BSPLND, A B-Spline N-Dimensional Package for Scattered Data Interpolation

Michael P. Weis
Tracker Business Systems
1835 Terminal Drive, Suite 220
Richland, WA 99352
1-509-946-5414

mike@vidian.net

Robert R. Lewis
Washington State University, School of EECS
2710 University Drive
Richland, WA 99352
1-509-372-7178

bobl@tricity.wsu.edu

ABSTRACT

The problem of scattered data interpolation is the fitting of a smooth surface (or, more generally, a manifold) through a set of non-uniformly distributed data points that extends to all positions in a domain. Common sources of scattered data include experiments, physical measurements, and computational values. Scattered data interpolation assists in interpreting such data through the calculation of values at arbitrary positions in the domain. Despite much attention in the literature, scattered data interpolation remains a difficult and computationally expensive problem to solve. BSPLND is a software package that solves this problem. It uses the scattered data interpolation technique presented in [1] (hereafter, LWS). This technique is fast and produces a C^2-continuous interpolation function for any set of scattered data using a hierarchical set of cubic B-splines. BSPLND extends the technique to work with data having an arbitrary number of dimensions for both its domain and range.

Categories and Subject Descriptors

I.3.4 [**Graphics Utilities**], I.3.5 [**Computational Geometry and Object Modeling**], G.1.2 [**Approximation**], E.2 [**Data Storage Representations**].

General Terms

Algorithms, Performance, Design.

Keywords

Scattered data interpolation, multilevel B-splines, data approximation.

1. INTRODUCTION

The scattered data interpolation technique presented by LWS is discussed in the context of bivariate data where the independent data is in 2D and the dependent data is a scalar. LWS develop the multilevel B-spline approximation method and the simpler algorithm on which it depends, the B-spline approximation method. The B-spline approximation method defines an approximation function for a set of scattered data in terms of uniform cubic B-spline basis functions on its own merit, and the multilevel technique improves upon it.

In this paper, we will set up the problem in the context of bivariate data in section 2, and then refer you to [1] for the details of the mathematical development. In section 3, we extend the math to data having an arbitrary number of dimensions in both its domain and range. The design and implementation of BSPLND are presented in sections 4 and 5, respectively.

2. Problem Definition for Bivariate Data

Let $\Omega = \{(x, y) \mid 0 \leq x < m, 0 \leq y < n\}$ be a rectangular domain in the xy-plane, and $P = \{(x_c, y_c, z_c)\}$ be a set of scattered points in 3D space. LWS define an approximation function f of this data as a uniform bicubic B-spline function in terms of a control lattice Φ overlaid on domain Ω. The control lattice is a $(m+3) \times (n+3)$ set of control points that spans the integer grid in Ω. Restricting the control points to integer values simplifies the math, without loss of generality. Figure 1, illustrates the relationship of Φ to Ω.
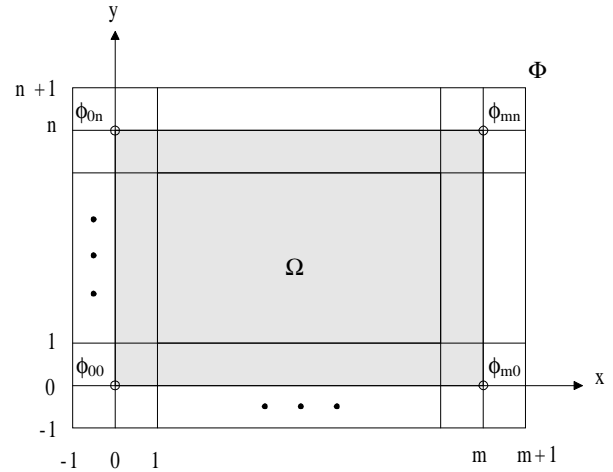

Figure 1. Control lattice configuration (from LWS).

Let ϕ_{ij} be the value of the ij-th control point in lattice Φ, located at (i, j) for $i = -1, 0, \ldots, m+1$ and $j = -1, 0, \ldots, n+1$. The approximation function f is defined in terms of these control points by

$$f(x, y) = \sum_{k=0}^{3} \sum_{l=0}^{3} B_k(s) B_l(t) \phi_{(i+k)(j+l)}, \quad (1)$$

where $i = \lfloor x \rfloor - 1$, $j = \lfloor y \rfloor - 1$, $s = x - \lfloor x \rfloor$, and $t = y - \lfloor y \rfloor$. B_k and B_l are uniform cubic B-spline functions defined as

$$B_o(t) = (1-t)^3 / 6,$$

$$B_1(t) = (3t^3 - 6t^2 + 4)/6,$$

$$B_2(t) = (-3t^3 + 3t^2 + 3t + 1)/6,$$

$$B_3(t) = t^3 / 6,$$

where $0 \le t < 1$. These functions serve as blending functions, weighing the contribution of each control point to $f(x, y)$ based on its distance to (x, y). Note that as B-spline curves, they sum to a value of one at each value of t. The problem of determining f then, is reduced to solving for the control points in Φ that best approximate the scattered data in P.

We refer the reader to [1] for the 2D mathematical development of calculating control lattice Φ, which defines an approximation function for fitting the scattered data. As mentioned in the introduction, LWS present two methods for its calculation, the B-spline approximation method, or BA algorithm, and its more capable descendant, the multilevel B-spline approximation method, or MBA algorithm. In the next section, we extend their mathematical development to data in any number of dimensions.

3. Extending the Technique to Data of Arbitrary Dimensionality

In handling scattered data having an arbitrary number of dimensions in both its domain and range, we denote the number of dimensions in the domain and range by D and R, respectively. Because each of the R data values in the range of a data point are independent of each other, we can determine one approximation function in terms of the independent data for each dimension of the range. Solving this problem then, can be reduced to solving R approximation functions with domain D and range $R = 1$ and grouping the R functions in some manner. The following is a derivation of the BA algorithm for a domain of dimension D and the range a scalar.

3.1 The BA Algorithm

The development is similar to the 2D case. Let the domain be defined as $\Omega = \{(x_0, x_1, ..., x_{D-1}) \mid 0 \le x_0 < m_0, 0 \le x_1 < m_1, ..., 0 \le x_{D-1} < m_{D-1}\}$ and the scattered data be defined as $P = \{(x_{0,c}, x_{1,c}, ..., x_{D-1,c}, z_c)\}$. The control lattice Φ overlaid on Ω is a $(m_0+3) \times (m_1+3) \times \cdots \times (m_{D-1}+3)$ set of control points that spans the integer grid in Ω. Let $\phi_{i_0 i_1 \cdots i_{D-1}}$ be the $i_0 i_1 \cdots i_{D-1}$-th control point in lattice Φ located at $(i_0, i_1, ..., i_{D-1})$ for $i_0 = -1, 0, ..., m_0 + 1$, $i_1 = -1, 0, ..., m_1 + 1$, ..., $i_{D-1} = -1, 0, ..., m_{D-1} + 1$. The approximation function is defined in terms of these control points by

$$f(x_0, x_1, \cdots, x_{D-1}) = \sum_{k_0=0}^{3} \sum_{k_1=0}^{3} \cdots \sum_{k_{D-1}=0}^{3} B_{k_0}(t_0) B_{k_1}(t_1) \cdots B_{k_{D-1}}(t_{D-1}) \phi_{(i_0+k_0)(i_1+k_1)\cdots(i_{D-1}+k_{D-1})}$$

$$= \sum_{k_0=0}^{3} \sum_{k_1=0}^{3} \cdots \sum_{k_{D-1}=0}^{3} \left(\prod_{d=0}^{D-1} B_{k_d}(t_d) \right) \phi_{(i_0+k_0)(i_1+k_1)\cdots(i_{D-1}+k_{D-1})}$$

where $i_d = \lfloor x_d \rfloor - 1$ and $t_d = x_d - \lfloor x_d \rfloor$ for $d = 0, 1, ..., D$-1 (one per dimension d of the domain). Each of the B_{k_d} are uniform cubic B-spline functions defined earlier, where each of the D subscripts $k_d = 0$, 1, 2, 3, and the D arguments t_d range from zero to one: $0 \le t_d < 1$. Thus

in (2) we extend the tensor product form of (1) to generate the function f for a D-dimension domain.

Similar to the 2D case, the problem of determining f is to solve for the control points in Φ that best approximate the scattered data in P. To start, we consider one data point in P, $(x_{0,c}, x_{1,c}, ..., x_{D-1,c}, z_c)$ and its relationship to the 4^D control points in its neighborhood. Assuming that $1 \le x_{0,c}, x_{1,c}, ..., x_{D-1,c} < 2$ as LWS do in the 2D case, then control points $\phi_{k_0 k_1 \cdots k_{D-1}}$ for $k_0, k_1, ..., k_{D-1} = 0$, 1, 2, 3 determine the value of f at $(x_{0,c}, x_{1,c}, ..., x_{D-1,c})$. For f to evaluate to z_c at $(x_{0,c}, x_{1,c}, ..., x_{D-1,c})$, the control points $\phi_{k_0 \cdots k_{D-1}}$ must satisfy

$$z_c = \sum_{k_0=0}^{3} \sum_{k_1=0}^{3} \cdots \sum_{k_{D-1}=0}^{3} w_{k_0 k_1 \cdots k_{D-1}} \phi_{k_0 \cdots k_{D-1}}, \quad (3a)$$

where $w_{k_0 k_1 \cdots k_{D-1}} = \prod_{d=0}^{D-1} B_{k_d}(t_d)$ and $t_d = x_{d,c} - 1$ for $d = 0, 1, ..., D$-1. We convert (3a) to matrix notation, by defining column vectors $\mathbf{W} = [w_{k_0 \cdots k_{D-1}} \mid k_0, k_1, ..., k_{D-1} = 0, 1, 2, 3]^T$ and $\mathbf{\Phi} = [\phi_{k_0 k_1 \cdots k_{D-1}} \mid k_0, k_1, ..., k_{D-1} = 0, 1, 2, 3]^T$:

$$z_c = \mathbf{W}^T \mathbf{\Phi}. \quad (3b)$$

Note that Equation (3) is a hyperplane in 4^D-space in unknowns $\phi_{k_0 k_1 \cdots k_{D-1}}$. Specifying the same minimization constraint as LWS for the 2D case, that the sum of the squares of the 4^D variables $\phi_{k_0 \cdots k_{D-1}}$ ($\mathbf{\Phi}^T \mathbf{\Phi}$) be minimized, amounts to locating the point on this hyperplane that is closest to the origin. This constraint is chosen to minimize the deviation of f from zero over the domain Ω. The parametric form of the line through the origin perpendicular to the hyperplane is given by

$$\mathbf{\Phi} = u\mathbf{W}, \quad (4)$$

where $\mathbf{W}$ represents a normal vector to the hyperplane and u is a scalar. Substituting (4) into (3b) yields

$$z_c = u\mathbf{W}^T \mathbf{W}$$

Solving for u and substituting into (4) gives the solution:

$$\mathbf{\Phi} = \frac{\mathbf{W}z_c}{\mathbf{W}^T \mathbf{W}}. \quad (5)$$

Summarizing, the 4^D equations given by (5) are the solution to control points $\phi_{k_0 k_1 \cdots k_{D-1}}$ in the vicinity of data point $(x_{0,c}, x_{1,c}, ..., x_{D-1,c})$. Therefore, by direct substitution of (5) into (3b) we see that function f evaluates to z_c at data point $(x_{0,c}, x_{1,c}, ..., x_{D-1,c})$. Equation (5) is the general solution to the 2D case presented in LWS, where there are $4^2 = 16$ control points in the vicinity of any given data point.

Next we consider the effects of all the data points in P on control lattice Φ. For each data point, (5) can be used to evaluate the 4^D control points in its neighborhood. When the neighborhoods of two data points overlap, multiple assignments to a control point occur via (5). To resolve multiple assignments to a control point, LWS consider all the data points in its neighborhood, calling this set of data points its proximity data set. The proximity data set of the single control point $\phi_{i_0 i_1 \cdots i_{D-1}}$ is given by $P_{i_0 i_1 \cdots i_{D-1}} = \{(x_{0,c}, ..., x_{D-1,c}, z_c) \in P \mid i_0 - 2 \le x_{0,c} < i_0$

+ 2, …, $i_{D-1} - 2 \leq x_{D-1,c} < i_{D-1} + 2$}. For each data point in $P_{i_0 i_1 \cdots i_{D-1}}$, (5) gives the single control point $\phi_{i_0 i_1 \cdots i_{D-1}}$ a different value ϕ_c:

$$\phi_c = \frac{w_c z_c}{\mathbf{W}^T \mathbf{W}}, \quad (6)$$

where $w_c = w_{k_0 k_1 \cdots k_{D-1}} = \prod_{d=0}^{D-1} B_{k_d}(t_d)$, and $k_d = i_d - (\lfloor x_{d,c} \rfloor - 1)$ and $t_d = x_{d,c} - \lfloor x_{d,c} \rfloor$ for $d = 0, 1, …, D$-1. To resolve multiple assignments to control point $\phi_{i_0 i_1 \cdots i_{D-1}}$, we follow LWS in their 2D approach by minimizing the error $e(\phi_{i_0 i_1 \cdots i_{D-1}}) = \Sigma_c (w_c \phi_{i_0 i_1 \cdots i_{D-1}} - w_c \phi_c)^2$, where the term $(w_c \phi_{i_0 i_1 \cdots i_{D-1}} - w_c \phi_c)$ is the difference between the real $(w_c \phi_{i_0 i_1 \cdots i_{D-1}})$ and expected $(w_c \phi_c)$ contributions of $\phi_{i_0 i_1 \cdots i_{D-1}}$ to function f at $(x_{0,c}, x_{1,c}, …, x_{D-1,c})$. Taking the derivative of approximation error e with respect to $\phi_{i_0 i_1 \cdots i_{D-1}}$ and equating the result to zero to find a minimum gives:

$$\phi_{i_0 i_1 \cdots i_{D-1}} = \frac{\sum_c w_c^2 \phi_c}{\sum_c w_c^2}. \quad (7)$$

Developing the pseudocode for the BA algorithm to handle an arbitrary number dimensions in the domain is a straightforward extension of the algorithm by LWS using the derivation given here. Recall that this derivation assumes that $R = 1$, and as such the solution is applied once for each dimension of the range of dimension R. This aspect of the algorithm shall be presented in Section 4 in the discussion of BSPLND. Here is the BA algorithm extended to a D-dimensional domain.

BA Algorithm ($R = 1$)
Input: scattered data $P = \{(x_{0,c}, x_{1,c}, …, x_{D-1,c}, z_c)\}$
Output: control lattice $\Phi = \{\phi_{i_0 i_1 \cdots i_{D-1}}\}$

for all $i_0, i_1, …, i_{D-1}$ **do**
    let $\delta_{i_0 i_1 \cdots i_{D-1}} = 0$ and $\omega_{i_0 i_1 \cdots i_{D-1}} = 0$
for each point $(x_{0,c}, x_{1,c}, …, x_{D-1,c}, z_c)$ in P **do**
    for each dimension $d = 0$ **to** D-1 **do**
        let $i_d = \lfloor x_{d,c} \rfloor - 1$
        let $t_d = x_{d,c} - \lfloor x_{d,c} \rfloor$
    end
    compute $\sum_{k_0=0}^{3} \sum_{k_1=0}^{3} \cdots \sum_{k_{D-1}=0}^{3} w_{k_0 k_1 \cdots k_{D-1}}^2$ ($\mathbf{W}^T \mathbf{W}$)
    for $k_0, k_1, …, k_{D-1} = 0, 1, 2, 3$ **do**
        compute $w_{k_0 k_1 \cdots k_{D-1}}$, an element of $\mathbf{W}$
        compute $\phi_{k_0 k_1 \cdots k_{D-1}}$, an element of Φ, using (5)
        add $w_{k_0 k_1 \cdots k_{D-1}}^2 \phi_{k_0 k_1 \cdots k_{D-1}}$ to $\delta_{(i_0+k_0)(i_1+k_1)\cdots(i_{D-1}+k_{D-1})}$
        add $w_{k_0 k_1 \cdots k_{D-1}}^2$ to $\omega_{(i_0+k_0)(i_1+k_1)\cdots(i_{D-1}+k_{D-1})}$
    end
end

for all $i_0, i_1, …, i_{D-1}$ **do**
    if $\omega_{i_0 i_1 \cdots i_{D-1}} \neq 0$ **then**
        compute $\phi_{i_0 i_1 \cdots i_{D-1}} = \delta_{i_0 i_1 \cdots i_{D-1}} / \omega_{i_0 i_1 \cdots i_{D-1}}$, using (7)
    else let $\phi_{i_0 i_1 \cdots i_{D-1}} = 0$
end

The BA algorithm has three major loops. In the first loop, the algorithm initializes the numerator and denominator of (7) for each control point in Φ. The second loop visits each data point, first calculating its effect on the 4^D control points in its neighborhood with (5), and second accumulating the numerator and denominator of (7). The third and final loop optimizes the multiple assignments to each control point by applying (7), if its particular denominator is nonzero; otherwise it is assigned a value of zero.

3.2 The MBA Algorithm

A tradeoff exists between the shape smoothness and the approximation accuracy of B-spline function f generated by the BA algorithm. LWS develop the multilevel B-spline approximation method, or MBA algorithm, to generate a function that is both smooth in shape and that closely approximates the data in P.

This MBA algorithm generates a hierarchy of control lattices, each via the BA algorithm, that represent a sequence of functions, the sum of which is the desired approximation function. The first function in the sequence is derived from a coarse lattice. The accuracy of this initial function on data in P is improved upon by subsequent functions in the sequence derived from finer lattices. Improving further upon this multilevel algorithm, LWS formulate a technique called B-spline refinement to reduce this sum of B-spline functions into one equivalent B-spline function. We refer the reader to [1] for the detailed development of the multilevel approach in 2D. However, we will briefly summarize the algorithm through the reproduction of a graphical illustration and the algorithm pseudocode from [1].

The MBA algorithm comes in two forms, each generating the same approximation function f. The first form, the basic MBA, generates a hierarchy of coarse to fine control lattices $\Phi_0, \Phi_1, …, \Phi_h$, overlaid on domain Ω and representing a sequence of functions f_k, the sum of which is the desired result f. The coarsest lattice Φ_0 is chosen to approximate the data in P via the BA algorithm. The difference between the data in P and the initial function f_0 evaluated at the data in P (specifically, $z_c - f_0(x_{0,c}, x_{1,c}, …, x_{D-1,c})$) serves as input to the next application of the BA algorithm to generate the next control lattice Φ_1, chosen by LWS to be twice the size (or density). This process continues for the calculation of subsequent lattices, each twice the size as the previous in the sequence, as applications of the BA algorithm for the lattices beyond Φ_0 serve to remove the residual error. Figure 2a illustrates the basic MBA for the 2D case.
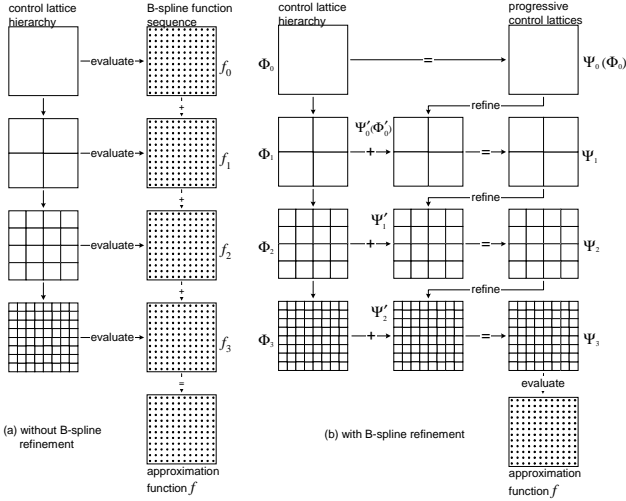
Figure 2. Approximation function evaluation in the MBA algorithm (from LWS).

Here is the algorithm for the basic MBA from LWS, modified for a domain of dimension D. Note that $\Delta^0 z_c = z_c$.

Basic MBA Algorithm ($R = 1$)
Input: scattered data $P = \{(x_{0,c}, x_{1,c}, \ldots, x_{D-1,c}, z_c)\}$
Output: control lattice hierarchy $\Phi_0, \Phi_1, \ldots, \Phi_h$
let k = 0
while $k \le h$ **do**
 let $P_k = \{(x_{0,c}, x_{1,c}, \ldots, x_{D-1,c}, \Delta^k z_c)$
 compute Φ_k from P_k by the BA algorithm
 compute $\Delta^{k+1} z_c = \Delta^k z_c - f_k(x_{0,c}, x_{1,c}, \ldots, x_{D-1,c})$ at each data pt.
 let $k = k + 1$
end

Notice that evaluating f requires ($h+1$) evaluations of a B-spline function by (1), rather than a single evaluation afforded by the BA algorithm (except when $h=0$). To eliminate this extra overhead in computation, which can be significant if f is evaluated for a large number of points in Ω, LWS formulate a technique they call B-spline refinement, illustrated in Figure 2b. This technique will reduce the ($h+1$) B-spline functions into one equivalent B-spline function. This is the second form of the MBA algorithm.

The B-spline refinement technique is progressive in that it is applied at each level in the control lattice hierarchy. Figure 2b illustrates the process described here. Let $F(\Phi)$ denote the B-spline function generated by control lattice Φ and let $|\Phi|$ denote the lattice size of Φ. Starting with the initial control lattice in the hierarchy Φ_0, a new control lattice Φ'_0 can be derived through B-spline refinement such that $F(\Phi'_0) = F(\Phi_0) = f_0$ and $|\Phi'_0| = |\Phi_1|$. In other words, Φ'_0 can be derived which defines an equivalent B-spline function to Φ_0, but which has the size of the next lattice in the hierarchy Φ_1. Then the sum of functions $f_0 + f_1$ can be satisfied by a single lattice Ψ_1 which results from the sum of each corresponding control point in Φ'_0 and Φ_1. Referring to Figure 2b, we continue the progression at the next level with lattice Ψ_1, applying B-spline refinement to derive lattice Ψ'_1 such that $F(\Psi'_1) = F(\Psi_1)$ and $|\Psi'_1| = |\Phi_2|$. The progression ends at the finest lattice in the hierarchy.

The process is generalized by the following pseudocode by LWS, modified for a domain of dimension D. In the algorithm, $P - F(\Phi)$ represents the updated data $\{(x_{0,c}, x_{1,c}, \ldots, x_{D-1,c}, \Delta^{k+1} z_c)\}$, where $P = \{(x_{0,c}, x_{1,c}, \ldots, x_{D-1,c}, \Delta^k z_c)\}$ and $f_k = F(\Phi)$.

MBA Algorithm ($R = 1$)
Input: scattered data $P = \{(x_{0,c}, x_{1,c}, \ldots, x_{D-1,c}, z_c)\}$
Output: control lattice Ψ
Let Φ be the coarsest lattice
Let $\Psi' = \mathbf{0}$
while Φ does not exceed the finest control lattice **do**
 compute Φ from P by the BA algorithm
 compute $P = P - F(\Phi)$
 compute $\Psi = \Psi' + \Phi$
 let Φ be the next finer control lattice in the hierarchy
 refine Ψ into Ψ' whereby $F(\Psi') = F(\Psi)$ and $|\Psi'| = |\Phi|$
end

The B-spline refinement step, the last one in the loop, can be performed by a variety of methods as noted by LWS. For the 2D case, LWS choose to refine a $(m + 3) \times (n + 3)$ control lattice Φ into a $(2m + 3) \times (2n + 3)$ lattice Φ' whose control point spacing is half as large as that of Φ. Then the refined lattice is the same size as the next lattice in the hierarchy and the sum of the two lattices Ψ' and Φ in the algorithm is a simple matter of summing their corresponding control points. Letting ϕ_{ij} and ϕ'_{ij} be the ij-th control point in Φ and Φ', respectively, then the position of control point $\phi'_{2i,2j}$ in Φ' corresponds to control point ϕ_{ij} in Φ. LWS define the following relationships between the control points in Φ' and those in Φ:

$$\phi'_{2i,2j} = \frac{1}{64}[\phi_{i-1,j-1} + \phi_{i-1,j+1} + \phi_{i+1,j-1} + \phi_{i+1,j+1} + 6(\phi_{i-1,j} + \phi_{i,j-1} + \phi_{i,j+1} + \phi_{i+1,j}) + 36\phi_{ij}]$$

$$\phi'_{2i,2j+1} = \frac{1}{16}[\phi_{i-1,j} + \phi_{i-1,j+1} + \phi_{i+1,j} + \phi_{i+1,j+1} + 6(\phi_{ij} + \phi_{i,j+1})]$$

$$\phi'_{2i+1,2j} = \frac{1}{16}[\phi_{i,j-1} + \phi_{i,j+1} + \phi_{i+1,j-1} + \phi_{i+1,j+1} + 6(\phi_{ij} + \phi_{i+1,j})]$$

$$\phi'_{2i+1,2j+1} = \frac{1}{4}[\phi_{i,j} + \phi_{i,j+1} + \phi_{i+1,j} + \phi_{i+1,j+1}]$$

Note there are four equations for the 2D case. In general, there are 2^D unique equations, each relating a control point in Φ to up to 3^D control points in Φ' in its vicinity. Notice that the factors that blend the control points in Φ' sum to one (for example, the four factors of ¼ in the last of the four equations given here).

4. BSPLND Design

The BSPLND package implements the MBA algorithm, for data having an arbitrary number of dimensions in its domain and range. BSPLND provides its user with two options for calculating an approximation function via the MBA algorithm given a set of scattered data points – *bsplnd_fit* and *bsplnd_fitToTol*. Each routine returns a data structure containing the control lattice that defines the approximation function. The control lattice in this form can then be passed to BSPLND's *bsplnd_eval* routine to evaluate the function at any independent data value, thereby implementing the D-dimensional form of the approximation function in (2). Because the data structure is allocated dynamically at run time, the package provides *bsplnd_delete* to free this memory.

BSPLND provides the user with flexibility in its two separate implementations of the MBA algorithm. The routine *bsplnd_fit*

accepts as input the number of levels of B-spline refinement (defined by h - see the basic MBA), so that the user predetermines the size of the control lattice hierarchy. When given $h = 0$ for no refinement, *bsplnd_fit* implements the BA algorithm. The second routine, *bsplnd_fitToTol* accepts a tolerance, or measure of accuracy, that the approximation function must achieve in its evaluation at the scattered data points. Therefore, this routine determines the number of levels of B-spline refinement required to achieve this tolerance.

The primary challenge in implementing the MBA algorithm and the supporting routines such as *bsplnd_eval* and *bsplnd_delete*, is the programming and data structure design required to handle data with an arbitrary number of dimensions in its domain and range. Because BSPLND routines must accept the dimensionalities D and R of the domain and range as input in order to properly interpret the scattered data they receive, the dimensions of an array to store the D-dimensional control lattice cannot be specified at compile time, but rather must be determined at run time. The solution to this problem, along with other features of the design, shall be discussed in this section.

First we shall present the data structure design for storing the D-dimensional control lattice and follow that discussion with the procedural design for accessing the structure. This will be accomplished through a set of procedures, or algorithms, that led to the implementation of the BSPLND routines mentioned above. We will start with the algorithm for *bsplnd_eval*, which demonstrates read access to the data structure. This algorithm assumes that the control lattice has already been calculated. It is the simplest of the algorithm set and will help us introduce the more complex multidimensional BA algorithm, which will now be presented in the context of the control lattice data structure. This algorithm demonstrates write access to the data structure, for it calculates the control lattice. We will follow this with the multidimensional algorithm for B-spline refinement, which, given a control lattice, generates a finer control lattice defining the same approximation function. Then, we can use both the BA algorithm and the B-spline refinement algorithm to build the MBA algorithm, as illustrated in Figure 2b. This algorithm led to the implementation of BSPLND routine *bsplnd_fit* and, with slight modification, the routine *bsplnd_fitToTol*. We name a BSPLND algorithm the same as its implemented counterpart when the algorithm led directly to the implementation. Whether referring to the algorithm, which led to the implementation, or the implementation itself, should be obvious from the context.

4.1 Storage and Access to the D-Dimensional Control Lattice

In the following discussion about the control lattice data structure we will frequently reference the BA algorithm rather than the MBA algorithm, because the BA algorithm actually calculates the control lattice. The MBA algorithm uses the BA algorithm to calculate the (h + 1) lattices in the control lattice hierarchy.

The principal input to the BA algorithm is the set of scattered data points P and the principal output is a D-dimensional control lattice Ψ overlaid on the domain Ω of the data. In order to properly interpret the scattered data the BA algorithm must also receive the number of dimensions in the domain and range of the data, D and R, and the number of data points, p. One method for storing the scattered data in memory is with two separate data stores, a $p \times D$ two-dimensional array for the independent data, and a $p \times R$ two-dimensional array for the dependent data, where each of the p rows in these corresponding

arrays represent the abscissa and ordinate of one data point, respectively. Let $x = \{(x_{0,c}, x_{1,c}, \ldots, x_{D-1,c})\}$ and $z = \{(z_{0,c}, z_{1,c}, \ldots, z_{R-1,c})\}$ denote these two data sets, where x and z are the domain and range of P, respectively. While the scattered data can conveniently be stored in a two-dimensional array structure as detailed here, storage for the output data, the control lattice, requires a D-dimensional array, because the lattice is overlaid on the domain Ω of the data, as illustrated in Figure 1. Herein lies the difficulty in storing the lattice. Because D is specified as input, the array dimensions cannot be specified at compile time, but must be determined when the implementation of the BA algorithm runs.

A review of how a two-dimensional array may be accessed as a one-dimensional array will provide some insight into solving the problem we face in storing and accessing the data for the control lattice. A caller of the BA algorithm can pass scattered data x and z (along with D, R, and p) as two-dimensional arrays. A conceptual view of x in memory as a two-dimensional array, where subscript c varies over the p data points from 0 to p - 1, is illustrated in Figure 3a.
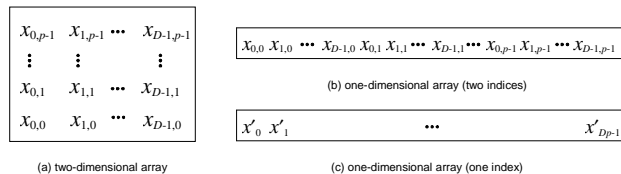

Figure 3. The independent data x in a two-dimensional array and corresponding one-dimensional array.

In this figure, we use a Cartesian indexing scheme that is consistent with the indexing used for control points (see Figure 1), where the first of the two indices is the fastest varying index, or subscript, indicating contiguous memory locations. We shall continue with this convention when introducing the multidimensional control lattice shortly. This is contrary to the C-convention of indexing two-dimensional arrays where the second index varies the fastest (row-major order). Either of these two conventions can be used in converting a two-dimensional index into a one-dimensional index.

From the caller's perspective, passing x as a two-dimensional array makes the most sense, for this is how the data is logically organized. However, because the two-dimensional array is stored in a contiguous set of memory locations, the BA algorithm can access the data as a one-dimensional array of $p \times D$ elements, as illustrated in Figure 3b. Figure 3b is derived from Figure 3a by traversing the data in Figure 3a from left to right and bottom to top. As long as parameter x is passed to the BA algorithm as a pointer to the first data value $x_{0,0}$, the algorithm can index it as a one dimensional array, using the following relationship:

$$x'_{i + Dj} = x_{i, j}, \quad (8)$$

where i and j represent the two-dimensional array indices. Figure 3c illustrates the one-dimensional indexing of array x. The BA algorithm accesses the scattered data in input x and z in this fashion. This gives the caller the freedom to pass x (and z) as a two-dimensional or a one-dimensional array, as long as it passes the algorithm a pointer to the first element in the array (along with values for D, R, and p).

We can store the D-dimensional control lattice in a one-dimensional array and accesses it in a similar manner. The array is allocated at run time with a capacity of $(m_0+3) \times (m_1+3) \times \cdots \times (m_{D-1}+3) \times R$ elements, one for each control point in the lattice. Recall that when $R > 1$ we can calculate a separate lattice for each dimension of the range under the assumption that $R = 1$ for that particular dimension. Taking this discussion in steps, we assume $R = 1$ for now. Afterwards we shall elaborate on how the storage and access of the lattice is affected when $R > 1$. When $R = 1$, the one-dimensional array consists of $(m_0+3) \times (m_1+3) \times \cdots \times (m_{D-1}+3)$ elements. To demonstrate, consider a three dimensional domain ($D = 3$). Figure 4 illustrates the three-dimensional view of the control lattice using the Cartesian indexing scheme.
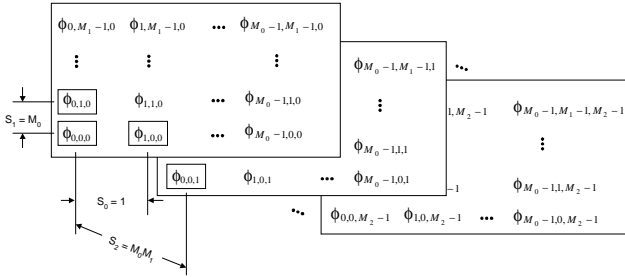
Figure 4. Three-dimensional control lattice.

Similar to the subscripting of independent data store x, we use zero-based indexing for the control points and have simplified subscripting with a change of variable, so that

$$M_d = m_d + 3, \quad (9)$$

for $d = 0, 1, \ldots, D$-1. M_d is the number of contiguous control points in dimension d of the domain. Similar to the two-dimensional example for independent data store x, the first dimension ($d = 0$) is along the horizontal from left to right and represented by the first subscript, the second ($d = 1$) is along the vertical from bottom to top and represented by the second subscript, and the third dimension ($d = 2$) is into the page from front to back and represented by the third subscript. The figure introduces the concept of the stride represented by variable S. The stride will provide the means of indexing the one-dimensional array in this example that is similar to the two-dimensional case in (8). If all $M_0 M_1 M_2$ elements in the three dimensional array were laid out end-to-end, then S_d is the number of elements between two adjacent elements in dimension d. Figure 5 illustrates the one-dimensional array storage of the three-dimensional control lattice and the stride S.
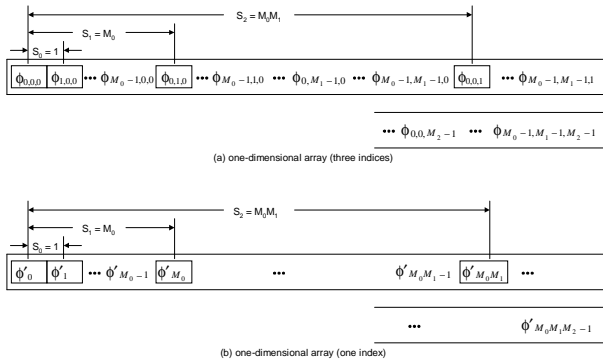

Figure 5. Storing the control lattice in a one-dimensional array.

Notice how the stride maps from the three-dimensional view in Figure 4 to the one-dimensional view of the data in Figure 5a. We can specify the one-dimensional array index for the control lattice, illustrated in Figure 5b, as a function of the three-dimensional array indices and the stride S as

$$\phi'_{iS_0 + jS_1 + kS_2} = \phi_{i,j,k}. \quad (10)$$

where $S_0 = 1$, $S_1 = M_0$, and $S_2 = M_0 M_1$ from Figures 4 and 5. For example for control point $\phi_{1,0,1}$, the one-dimensional array index is

$1 \times S_0 + 0 \times S_1 + 1 \times S_2 =$

$1 \times 1 + 0 \times M_0 + 1 \times M_0 M_1 = M_0 M_1 + 1.$

Explaining this result, we notice from Figure 4 that starting at the initial control point $\phi_{0,0,0}$, and traversing them from left to right, bottom to top, and front to back in the order we would lay them out in one dimension, that control point $\phi_{1,0,1}$ is the $(M_0 M_1 + 2)$th control point in the sequence. With zero-based indexing, the index to the corresponding one-dimensional array is $M_0 M_1 + 1$, as calculated.

We can extend this three-dimensional example to the general case where the domain is of D dimensions. Notice that in this example that the stride is increasing as the dimension of the data (d) increases, as more elements must be skipped to reach the adjacent element in dimension d. We apply the pattern that develops in the three-dimensional case to D dimensions to define the stride recursively as

$$S_d = S_{d-1} M_{d-1}, \quad (11)$$

where $S_0 = 1$ and $d = 1, 2, \ldots, D$-1. The one-dimensional array index is then a function of the D-dimensional indices and the stride:

$$\phi'_{I_0 S_0 + I_1 S_1 \cdots I_{D-1} S_{D-1}} = \phi_{I_0, I_1, \cdots, I_{D-1}}$$
$$\phi'_{\sum_{d=0}^{D-1} I_d S_d} = \phi_{I_0, I_1, \cdots, I_{D-1}}, \quad (12)$$

where $I_d = 0, 1, \ldots, M_d - 1$, and $I_d = i_d + 1$ for $d = 0, 1, \ldots, D$-1. Recall that i_d was used in the (-1)-based indexing for the multidimensional control lattice introduced in Section 3.1. Notice that (8) is the two-dimensional application of the general case.

We now elaborate on the case where $R > 1$. As we mentioned in Section 3, when the range of the scattered data is a vector ($R > 1$), we solve R approximation functions, one for each dimension of the range, and group them in some manner. We can accommodate the R control lattices defining these R functions by increasing the control lattice array from $M_0 M_1 \cdots M_{D-1}$ elements to $M_0 M_1 \cdots M_{D-1} \times R$ elements. In essence, each control point is an R-element vector. Figure 6 illustrates a two-dimensional conceptual view of this storage, where the first array index is for the dimension of the range and the second is the one-dimensional index calculated with (12), so that the R values for each control point "vector" are adjacent in memory.

$$\begin{array}{cccc} \phi_{0,M_0M_1\cdots M_D-1} & \phi_{1,M_0M_1\cdots M_D-1} & \cdots & \phi_{R-1,M_0M_1\cdots M_D-1} \\ \vdots & \vdots & & \vdots \\ \phi_{0,1} & \phi_{1,1} & \cdots & \phi_{R-1,1} \\ \phi_{0,0} & \phi_{1,0} & \cdots & \phi_{R-1,0} \end{array}$$

Figure 6. Two-dimensional view of complete control lattice.

In this case the two values of stride are $S_0 = 1$ and $S_1 = R$. Accessing this as a one-dimensional array is straightforward from (8):

$$\phi'_{r+jR} = \phi_{r,j}, \quad (13)$$

where $r = 0, 1, \ldots, R\text{-}1$ and $j = 0, 1, \ldots, M_0M_1\cdots M_{D-1} - 1$. Thus, we index the control lattice as a one-dimensional array with two steps: first by (12), then by (13). Combining the two relationships yields,

$$\phi'_{r+\left[\sum_{d=0}^{D-1} I_d S_d\right]R} = \phi_{r,I_0,I_1,\cdots I_{D-1}}. \quad (14)$$

4.2 BSPLND Algorithms

4.2.1 Evaluation of the Approximation Function with bsplnd_eval

The *bsplnd_eval* algorithm evaluates an approximation function defined by control lattice Ψ at one position $(x_0, x_1, \ldots, x_{D-1})$ in the domain Ω; in other words, it implements (2). This algorithm shall demonstrate the procedural design for accessing the one-dimensional array of the D-dimensional control lattice with (14). It assumes that the control lattice has been calculated (for example, via the BA algorithm).

First, we introduce the data structure of the object that encapsulates the control lattice data store and several other data elements. The additional data or fields of this structure include data required of the user of BSPLND to calculate the control lattice (e.g., D and R) and calculated data (e.g., S) useful to several of the BSPLND algorithms. Rather than pass them as separate parameters or require their calculation in each algorithm, they are packaged into this one structure and the structure (or a pointer to it) is communicated as a whole. Table 1 lists the data fields of this structure.

Table 1. Data structure for storing control lattice.

Field	Array Size	Description
ϕ	$M[0]M[1]\ldots M[D\text{-}1]R$	Number of control points in control lattice.
S	D	The stride.
D	Scalar	No. of dimensions in the domain.
R	Scalar	No. of dimensions in the range.
M	D	No. of control points in one dimension of the domain.

xMin, xMax	D, D	Lower and upper boundary values of the domain, one per dimension of the domain.
slope, intercept	D, D	Fields *slope* and *intercept* represent a transformation function from the domain of the scattered data to the domain of the control lattice.
h	Scalar	No. of levels of B-spline refinement used in calculating this control lattice.

Data fields D, R, S, and M maintain their definitions. Fields *xMin* and *xMax* are D-element arrays of real numbers that specify the lower and upper bound of the domain Ω in each dimension d. Fields *slope* and *intercept* are D-element arrays of real numbers that define a linear mapping function from domain Ω of the scattered data to the domain of the control lattice. We define the domain of the control lattice as follows. It is not reasonable that the BSPLND user be required to specify the domain Ω as defined in Section 3.1, such that the control points span the integer grid in Ω. Therefore, we define the domain of the control lattice to be one defined similar to the one in Section 3.1 for Ω as $\{(x'_0, x'_1, \ldots, x'_{D-1}) \mid 0 \le x'_0 < M_0 - 3, 0 \le x'_1 < M_1 - 3, \ldots, 0 \le x'_{D-1} < M_{D-1} - 3\}$ and map domain Ω to it, so that 0 (zero) corresponds to $xMin[d]$ in Ω and $M_{d-1} - 3 = M[d] - 3$ corresponds to $xMax[d]$ in Ω, for all d. We then can map any position $(x_0, x_1, \ldots, x_{D-1})$ in the scattered data's domain Ω to this domain using fields *slope* and *intercept*, and still use the development in Section 3.1. Field h of this data structure is the number of levels of B-spline refinement used to calculate this lattice. Finally, data field ϕ is a $M[0]M[1]\ldots M[D-1]R$-element array of real numbers that are the control points of the control lattice, initially zero elements in size.

All array sizes but that of array ϕ can be preset with some reasonable upper limit on D without consuming too much memory, but because the capacity of ϕ can become rather large in practice and can vary widely, we dynamically allocate its storage prior to calculating it in the BA algorithm.

Before listing the *bsplnd_eval* algorithm, we introduce three algorithms. The first calculates the stride S:

```
calc_stride(D, M, S{↑})
1    S[0] ← 1
2    for d ← 1 to D – 1 do
3        S[d] ← S[d–1] · M[d–1]
4    end
```

The mapping function stored in fields *slope* and *intercept* is calculated in the *bsplnd_new* algorithm. The *bsplnd_new* algorithm initializes the control lattice object Φ (Table 1). It accepts D, R, M, *xMin*, and *xMax* as input (ultimately from the BSPLND user) and control lattice object Φ as output. It populates Φ, including fields *slope* and *intercept*, for the BA algorithm before the BA calculates the control lattice in field ϕ. We shall use it later.

```
bsplnd_new(D, R, M, xMin, xMax, Φ{↑})
```

```
1    Φ.D ← D
2    Φ.R ← R
3    Φ.h ← 0
4    for d ← 0 to D – 1 do
5        Φ.M[d] ← M[d]
6        Φ.xMin[d] ← xMin[d]
7        Φ.xMax[d] ← xMax[d]
8        range ← xMax[d] – xMin[d]
9        Φ.slope[d] ← (M[d] – 3)/range
10       Φ.intercept[d] ← ((M[d] – 3) · xMin[d])/range
11   end
12   calc_stride(D, M, Φ.S)
13   ▷ Calculate the capacity for field φ and allocate its memory
14   size ← M[D –1] · Φ.S[D –1] · R
15   Alloc(Φ.φ, size)
```

The BSPLND algorithms *calc_stride* and *bsplnd_new* use a pseudocode that more closely resembles an actual programming language like C or Pascal than do the algorithms by LWS in previous sections. The BSPLND algorithms generally follow the pseudocoding conventions in [2]. Note that an output parameter is indicated by {↑} in the formal parameter list of an algorithm, and an input/output parameter in indicated by {↓↑}. Absence of a symbol after the parameter denotes an input parameter. Also, for arrays in greater than one dimension, we use the C subscripting convention of row-major order.

Before listing the *bsplnd_eval* algorithm we present one more algorithm, called e*val_basis_functions*, which evaluates the four uniform cubic B-spline functions at a given value *t*. This algorithm is called by *bsplnd_eval*.

```
eval_basis_functions(t, B{↑})
1    a ← 1 – t
2    b ← aa
3    c ← tt
4    d ← ct
5    B[0] ← ab/6
6    B[1] ← d/2 – c + 2/3
7    B[2] ← -d/2 + c/2 + t/2 + 1/6
8    B[3] ← d/6
```

After execution of this algorithm, array *B* contains the four B-spline function evaluated at *t*: $B_0(t)$, $B_1(t)$, $B_2(t)$, and $B_3(t)$. The four B-spline functions are not readily recognizable in e*val_basis_functions*, because the algorithm calculates them with the minimum number of operations possible. We used the software application Mathematica [3] to determine the quickest means of calculating these four functions, because this algorithm is called *D* times for each scattered data point input into the BA algorithm given later.

We now list the *bsplnd_eval* algorithm, which implements (2) by evaluating the approximation function stored in control lattice object Ψ at independent data stored in *D*-element array *X*. It returns the result in the *R*-element array *Z*.

```
bsplnd_eval(Ψ, X, Z{↑})
1    ▷ Locate the independent data X relative to the control lattice
2    ▷ with the "anchor" index i[d] for each dim. d of the domain
3    ▷ and evaluate the four B-spline functions for each dim. d.
4    for d ← 0 to Ψ.D – 1 do              ▷ Loop #1
5        Xmap ← Ψ.slope[d] · X[d] + Ψ.intercept[d]
6        i[d] ← ⌊Xmap⌋ – 1        ▷ i_d in (2)
7        t[d] ← Xmap – ⌊Xmap⌋     ▷ t_d in (2)
8        eval_basis_functions(t[d], CubicBsplines[d])
```

```
9    end
10   ▷ Evaluate control lattice for each dim. r of the range.
11   for r ← 0 to Ψ.R - 1 do                    ▷ Loop #2
12       Z[r] ← 0
13       ▷ Calculate the sum of the 4^D terms in (2)'s summation.
14       for TermCnt ← 0 to (1 << (2 · Ψ.D)) – 1 do     ▷ Loop #2.1
15           Index1D ← 0
16           BsplineProd ← 1
17           ▷ For this term, generate unique combination of k_d;
18           ▷ calc. B-spline func. product; accum; 1D index to Ψ.φ .
19           for d ← 0 to Ψ.D – 1 do            ▷ Loop #2.1.1
20               k[d] ← (TermCnt >> (2d)) & 0x3    ▷ k_d in (2)
21               ▷ Accumulate ∏_{d=0}^{D-1} B_{k_d}(t_d) in (2).
22               BsplineProd ← BsplineProd · CubicBsplines[d][k[d]]
23               ▷ Accum. 1D index to Ψ.φ with (12), assuming R=1.
24               Index1D ← Index1D + Ψ.S[d] · ((i[d] + 1) + k[d])
25           end
26           ▷ Finish 1D index to Ψ.φ via (13), to account for dim. r.
27           Index1D ← r + Index1D · Ψ.R
28           ▷ Add current term to the summation.
29           Z[r] ← Z[r] + BsplineProd · Ψ.φ[Index1D]
30       end
31   end
```

The algorithm for *bsplnd_eval* follows directly from (2) and the mathematical development given here for accessing the *D*-dimensional control lattice stored in a one-dimensional array. It may be helpful to the reader to refer back to equation (2) at this point, as many of the terms in this equation are referenced in walking through *bsplnd_eval*.

The first loop of the algorithm calculates data to be used in the second loop. First, it maps the independent data in *X* to the control lattice domain. Using this result, it calculates the *D*-dimensional, (-1)-based indices i_d in array *i*, where each i_d is the first of the four control point indices in dimension *d* that effect the value of the approximation function at *X*. Later, in Loop #2, these participate in generating the one-dimensional array index to the control lattice in Ψ.φ for accessing one control point for one term in (2). After calculating in array *t* the arguments t_d to the four B-spline functions, Loop #1 evaluates these functions with a call to *eval_basis_functions*. Upon completion of Loop #1, the array *CubicBsplines* contains 4*D* values as illustrated in Figure 7. We conserve execution time by calculating the 4*D* unique evaluations of the B-spline functions in the first loop and storing them in memory. If calculated in Loop #2 on an as needed basis, (2) requires $D4^D$ evaluations of these functions.

d=0	$B_0(t[0])$	$B_1(t[0])$	$B_2(t[0])$	$B_3(t[0])$
d=1	$B_0(t[1])$	$B_1(t[1])$	$B_2(t[1])$	$B_3(t[1])$
	⋮	⋮	⋮	⋮
d=D-1	$B_0(t[D-1])$	$B_1(t[D-1])$	$B_2(t[D-1])$	$B_3(t[D-1])$

Figure 7. The populated CubicBsplines array at the conclusion of Loop #1 in *bsplnd_eval*.

The second loop accumulates the 4^D terms of the summation in (2) for each dimension of the range independently, accessing the appropriate 4^D control point elements in array Ψ.φ. As noted earlier, there is a

separate set of $M[0]M[1]\ldots M[D$-$1]$ control points in $\Psi.\phi$ for each dimension r of the range. When Loop #2 completes it has stored the R separate evaluations in output array Z. Nested Loop #2.1 calculates the summation for dimension r of the range; therefore, each pass through this loop generates one term in the summation of (2). Key to the algorithm, nested Loop #2.1.1 generates a unique combination of the summation variables k_d in array k (one per dimension d, where $k_d = 0$, 1, 2, or 3) as a function of the loop iteration variable *TermCnt*. The summation variables k_d index the B-spline functions in the B-spline function product $\prod_{d=0}^{D-1} B_{k_d}(t_d)$ of (2), and assist in indexing the one-dimensional control lattice array $\Psi.\phi$. At the conclusion of Loop #2.1.1 then, we have calculated $\prod_{d=0}^{D-1} B_{k_d}(t_d)$ for one term in the summation, and indexed the one-dimensional array $\Psi.\phi$ with (12). This indexing, which occurs in line 24, increments the array element $i[d]$ by one to convert to zero-based indexing. Then in line 27, the algorithm completes the indexing of the one-dimensional control lattice, taking into account the correct dimension of the range with (13). Finally, in line 29, the algorithm adds the current term to the accumulating sum in Z for the current dimension of the range.

4.2.2 The BSPLND BA Algorithm: bsplnd_fitUL

In this section we present BSPLND's multidimensional BA algorithm. This algorithm and BSPLND's B-spline refinement algorithm in the next section are the basic building blocks of BSPLND's MBA algorithm, which will complete the algorithm set. The name of the BSPLND version of the BA algorithm is *bsplnd_fitUL*. (The *UL* stands for "uni-level" as the BA algorithm is used to calculate a control lattice at one level in the control lattice hierarchy in the MBA algorithm.)

The *bsplnd_fitUL* algorithm, listed below, accepts scattered data in parameters x and z, where x is an array of the independent data and z is an array of the dependent data. Given input parameters D and R for the domain and range, respectively, and p for the number of scattered data points, the algorithm assumes that x is a $(p \times D)$-element array and that z is a $(p \times R)$-element array. Furthermore, it assumes the D components of each data point's independent data are stored in contiguous memory locations, as well as the R components of each data point's dependent data. The boundaries of the scattered data domain Ω are passed to *bsplnd_fitUL* in parameters *xMin* and *xMax*, D-element arrays, so that it can properly map the independent data to the control lattice domain defined by input parameter M, also a D-element array. Input parameter *return_dz* is a boolean variable which if true will cause *bsplnd_fitUL* to calculate the deviation of the calculated approximation function from the data points, overwriting the range of the data points in input/output parameter z. This feature will be useful to the MBA algorithm in calculating the control lattice hierarchy. The sole output parameter Φ represents the control lattice object. The algorithm assumes the object contains no meaningful data, including a zero-element control lattice array in data field $\Psi.\phi$. Thus, *bsplnd_fitUL* will determine the control lattice's size and allocate the memory for it.

bsplnd_fitUL($p, D, x, R, z\{\downarrow\uparrow\}$, *return_dz*, M, *xMin*, *xMax*, $\Phi\{\uparrow\}$)

```
1    ▷ Populate control lattice object
2    bsplnd_new(D, R, M, xMin, xMax, Φ)
3    ▷ Calculate size of control lattice array and allocate size elements
4    ▷ for numerator δ and denominator ω of (7)
5    size ← M[D −1] · Φ.S[D −1] · R
6    Alloc(δ, size)
```

```
7    Alloc(ω, size)
8    for j ← 0 to size – 1 do                      ▷ Loop #1
9        δ[j] ← ω[j] ← 0
10   ▷ For each data point, calculate its effect on the 4^D control points
11   ▷ nearby (actually, one set of 4^D points per dimension of range).
12   for c ← 0 to p – 1 do                          ▷ Loop #2
13       ▷ Locate data point relative to the D-dim. control lattice by
14       ▷ calcg. the "anchor" index i[d] for each dim. d of the domain;
15       ▷ Evaluate the four B-spline functions for each dimension d.
16       for d ← 0 to D – 1 do                      ▷ Loop #2.1
17           Xmap ← Φ.slope[d] · x[d + cD] + Φ.intercept[d]
18           i[d] ← ⌊Xmap⌋ – 1
19           t[d] ← Xmap – ⌊Xmap⌋
20           eval_basis_functions(t[d], CubicBsplines[d])
21       end
22       ▷ Calculate the SS of the 4^D B-spline products, W^T W in (6).
23       BsplineProdSumSqrs ← 0
24       for TermCnt ← 0 to (1 << (2D)) – 1 do  ▷ Loop #2.2
25           BsplineProd ← 1
26           for d ← 0 to D – 1 do                  ▷ Loop #2.2.1
27               k[d] ← (TermCnt >> (2d)) & 0x3
28               BsplineProd ← BsplineProd · CubicBsplines[d][k[d]]
29           end
30           BsplineProdSumSqrs ← BsplineProdSumSqrs + BsplineProd · BsplineProd
31       end
32       ▷ Calculate a control lattice for each dimension r of the range
33       for r ← 0 to R - 1 do                      ▷ Loop #2.3
34           ▷ Accum. effect of data point on the 4^D cntrl. Pts. nearby for
35           ▷ dim. r of range, using the results from Loops #2.1 and #2.2
36           for TermCnt ← 0 to (1 << (2D)) – 1 do      ▷ Loop #2.3.1
37               Index1D ← 0
38               BsplineProd ← 1
39               ▷ Gen. 1 B-spline product (w_c in (6)); calc. 1D array
40               ▷ index to Φ.φ independent of r (using i from Loop #2.1)
41               for d ← 0 to D – 1 do              ▷ Loop #2.3.1.1
42                   k[d] ← (TermCnt >> (2d)) & 0x3
43                   BsplineProd ← BsplineProd · CubicBsplines[d][k[d]]
44                   Index1D ← Index1D + Φ.S[d] · ((i[d] + 1) + k[d])
45               end
46               ▷ Complete 1D index to Φ.φ, accounting for dim. r of range
47               Index1D ← r + Index1D · R
48               ▷ Calculate data point's effect on one control point with (6)
49               phi_c ← (BsplineProd · z[r + cR])/BsplineProdSumSqrs
50               ▷ Indexing this one control point, accumulate the
51               ▷ numerator and denominator of (7)
52               δ[Index1D] ← δ[Index1D] + BsplineProd · BsplineProd · phi_c
53               ω[Index1D] ← ω[Index1D] + BsplineProd · BsplineProd
54           end   ▷ end for each control point in the vicinity of data point
55       end   ▷ end for each dimension r of range
56   end   ▷ end for each data point
57   ▷ Calculate the control lattice using (7)
58   for j ← 0 to size – 1 do                      ▷ Loop #3
59       if ω[j] <> 0 then
60           Φ.φ[j] ← δ[j]/ω [j]
61       else
62           Φ.φ[j] ← 0
63   end
64   ▷ Return the deviation of the approx. function defined by the lattice
65   ▷ from the scattered data points, storing in i/o parameter z.
66   if return_dz then
67       for c ← 0 to p – 1 do                      ▷ Loop #4
```

```
68        bsplnd_eval(Φ, Addr(x[cD]), Z)
69        for r ← 0 to R - 1 do                    ▷ Loop #4.1
70            z[r + cR] ← z[r + cR] − Z[r]
71        end
72    ▷ Cleanup
73    Free(δ)
74    Free(ω)
```

A walk through the code of *bsplnd_fitUL* reveals that the BSPLND version of the BA algorithm closely follows the logic of the extended BA algorithm from Section 3.1. This version however, brings the BA algorithm closer to implementation as it considers the data store for the D-dimensional control lattice and it accounts for multiple dimensions in the range of the scattered data.

The algorithm first performs some setup by initializing the control lattice object in parameter Φ with a call to *bsplnd_new*. It then allocates storage, one element per control point, for local arrays δ and ω for accumulating the numerator and denominator of (7) for each control point. After this setup, the algorithm calculates the contribution of each data point to the 4^D control points in its vicinity, storing its contribution in the appropriate 4^D elements of δ and ω. This is actually done for each dimension of the range independently, so that one data point effects $4^D R$ elements of δ and ω. After each data point has been visited, the algorithm calculates each control point by dividing ω into δ element by element for a nonzero denominator or sets the control point to zero otherwise. A control point in the latter category does not have any data points in its proximity data set. Finally, the algorithm calculates the deviation of the calculated approximation function from the data points, if so requested.

In essence, *bsplnd_fitUL* calculates R separate control lattices in parallel, the algorithm viewing each component of the range as a separate scalar range ($R = 1$) applied against the full domain of the data. Control lattice object Φ has been designed to accommodate them, and the algorithm is coded to store them in Φ separate from each other.

4.2.3 The BSPLND B-spline Refinement Algorithm: bsplnd_refine

In this section we present BSPLND's multidimensional B-spline refinement algorithm, the second basic building block of BSPLND's MBA algorithm. The name of the algorithm, listed below, is *bsplnd_refine*. The *bsplnd_refine* algorithm accepts a control lattice Ψ that has already been calculated, and calculates a finer lattice defining an equivalent approximation function, returning it in output parameter Ψ'.

```
bsplnd_refine(Ψ, Ψ'{↑})
1     D ← Ψ.D
2     R ← Ψ.R
3     ▷ Allocate memory for coarse lattice indices that will contribute
4     ▷ to each control point in the fine lattice and memory for weights.
5     capacity ← 1
6     for d ← 0 to D – 1 do                    ▷ Loop #1
7         capacity ← capacity · 3
8         M_tmp[d] ← 2 · Ψ.M[d] – 3    ▷ refined lattice to be twice as fine
9     end
10    Alloc(CoarseIdxs, capacity)
11    Alloc(Weights, capacity)
12    ▷ Populate refined lattice to be twice as fine as the coarse
13    bsplnd_new(D, R, M_tmp, Ψ.xMin, Ψ.xMax, Ψ')
14    size ← Ψ'.M[D –1] · Ψ'.S[D –1]    ▷ Assume R=1
```

```
15    ▷ Calc. the fine lattice, one control point at a time.  Assume R=1.
16    for j ← 0 to size – 1 do                    ▷ Loop #2
17        ▷ Resolve the 1D index j into its D-dimensional indices
18        rem ← j
19        for d ← D – 1 downto 0 do              ▷ Loop #2.1
20            FineIdxs[d] ← (rem div Ψ'.S[d]) – 1    ▷ (-1)-based indexing
21            rem ← rem mod Ψ'.S[d]
22        end
23        ▷ Calculate the 1D array indices of the coarse lattice Ψ and their
24        ▷ weights that contribute to the current control point j in the
25        ▷ refined lattice Ψ'.  To do so, use the D-dimensional indices of
26        ▷ control pt. j to locate the related D-dimensional indices in the
27        ▷ coarse lattice and convert these back to a 1D index.
28        CoarseIdxs[0] ← 0
29        Weights[0] ← 1.0
30        NumWgts ← 1
31        for d ← 0 to D – 1 do                    ▷ Loop #2.2
32            ▷ odd index
33            if FineIdxs[d] & 0x1 <> 0 then begin
34                MinIdx ← (FineIdxs[d] – 1)/2 + 1    ▷ 0-based indexing
35                for i ← NumWgts – 1 downto 0 do     ▷ Loop #2.2.1
36                    CoarseIdxs[2i + 1] ← CoarseIdxs[i] + (MinIdx + 1) ·
                          Ψ.S[d]
37                    CoarseIdxs[2i] ← CoarseIdxs[i] + (MinIdx) · Ψ.S[d]
38                    Weights [2i + 1] ← Weights [i] · (0.5)
39                    Weights [2i] ← Weights [i] · (0.5)
40                end
41                NumWgts ← NumWgts · 2
42            end
43            ▷ even index
44            else begin
45                MinIdx ← (FineIdxs[d]/2 – 1) + 1    ▷ 0-based indexing
46                for i ← NumWgts – 1 downto 0 do     ▷ Loop #2.2.2
47                    CoarseIdxs[3i + 2] ← CoarseIdxs[i] + (MinIdx + 2) ·
                          Ψ.S[d]
48                    CoarseIdxs[3i + 1] ← CoarseIdxs[i] + (MinIdx + 1) ·
                          Ψ.S[d]
49                    CoarseIdxs[3i] ← CoarseIdxs[i] + (MinIdx) · Ψ.S[d]
50                    Weights[3i + 2] ← Weights[i] · (0.125)
51                    Weights[3i + 1] ← Weights[i] · (0.75)
52                    Weights[3i] ← Weights[i] · (0.125)
53                end
54                NumWgts ← NumWgts · 3
55            end
56        end   ▷ end for dimension of the domain
57        ▷ Calculate the control points.  Account for R>1, so that we are
58        ▷ computing each of the R lattices in Ψ' in parallel.
59        for r ← 0 to R - 1 do                    ▷ Loop #2.3
60            Ψ'.φ[r + jR] ← 0
61        for r ← 0 to R - 1 do                    ▷ Loop #2.4
62            for i ← 0 to NumWgts – 1 do
63                Ψ'.φ[r + jR] ← Ψ'.φ[r + jR] + Ψ.φ[r + CoarseIdxs[i] · R] ·
                      Weights[i]
64    end
```

The algorithm first performs some setup by populating Ψ' with a call to *bsplnd_new*. This lattice will have the same values as Ψ for data fields D, R, *xMin*, and *xMax* because the domain and range do not change; however, it will have different values in fields M, S, *slope*, and *intercept* which are dependent on the size of the lattice. The setup process also includes memory allocation for the 3^D control points in coarse lattice Ψ that each control point in refined lattice Ψ' can

potentially relate to (in the 2D case there are nine), and the corresponding weights that will blend these control points into the one. Then in line 14, it calculates the number of control points to calculate in Ψ', assuming R=1. After the setup, the algorithm fills these two data stores *CoarseIdxs* and *Weights* for each control point it must calculate in Ψ'. This occurs in the Loop #2, where the one-dimensional index for the current control point in Ψ' is given by loop counter j. In Loop #2.1, the one dimensional index j is resolved into its (-1)-based D-dimensional indices and stored in *FineIdxs*. Recall that this is the indexing scheme presented by LWS in Figure 1. Then in Loop #2.2, the two arrays *CoarseIdxs* and *Weights* are calculated solely as a function of these D indices (in *FineIdxs*).

Finally, in Loops #2.3 and #2.4, the control points in Ψ (*CoarseIdxs*) that contribute to the value of control point j in Ψ' are blended (via *Weights*) to give its value. Because the calculation of *CoarseIdxs* and *Weights* is strictly a function of the geometric position of control point j in the fine lattice, here we can perform this calculation for each dimension r of the range, independently.

4.2.4 The BSPLND MBA Algorithm: bsplnd_fit

In this section we present BSPLND's multidimensional MBA algorithm to complete the algorithm set. It is built from the BA algorithm *bsplnd_fitUL* and the B-spline refinement algorithm *bsplnd_refine*. It is the predecessor to the BSPLND package's *bsplnd_fit* routine. After presenting this algorithm we'll discuss how to modify it to be tolerance-based, calculating the levels of B-spline refinement required to achieve a given level of accuracy in the approximation function.

The *bsplnd_fit* algorithm follows the logic displayed in Figure 2b, the LWS diagram for the MBA. Before listing it however, we list three short algorithms that it calls that correspond to this figure nicely. The first, *bsplnd_finer*, calculates the next finer lattice Φ_{k+1} in the control lattice hierarchy (see Figure 2b) given the current lattice Φ_k and the current deviation of the hierarchy from the scattered data. It also calculates the new deviation from the data of the now extended hierarchy, to be fed into a subsequent call to *bsplnd_finer*. The algorithm *bsplnd_add* calculates a lattice as the sum of two equally sized lattices defined on the same domain of the scattered data (see Figure 2b). Finally, the algorithm *bsplnd_delete* frees the memory dynamically allocated in data field ϕ of the control lattice object, allowing *bsplnd_fit* to reuse memory used by previous levels in the hierarchy as the algorithm progresses.

bsplnd_finer(Φ, p, x, $dz\{\downarrow\uparrow\}$, $\Phi\_next\{\uparrow\}$)
1.   ▷ Next lattice in hierarchy is to be twice as fine.
2.   **for** $d \leftarrow 0$ **to** $\Phi.D - 1$ **do**
3. $M\_tmp[d] \leftarrow 2 \cdot \Phi.M[d] - 3$
4.   ▷ Calculate next lattice in hierarchy and return deviation in dz
5.   bsplnd_fitUL(p, $\Phi.D$, x, $\Phi.R$, dz, TRUE, M_tmp, $\Phi.xMin$, $\Phi.xMax$, Φ_next)

bsplnd_add($\Phi1$, $\Phi2$, $\Phi\_sum\{\uparrow\}$)
1.   ▷ Populate sum.
2.   bsplnd_new($\Phi1.D$, $\Phi1.R$, $\Phi1.M$, $\Phi1.xMin$, $\Phi1.xMax$, Φ_sum)
3.   ▷ Sum corresponding control points.
4. $size \leftarrow \Phi1.M[\Phi1.D - 1] \cdot \Phi1.S[\Phi1.D - 1] \cdot \Phi1.R$
5.   **for** $i \leftarrow 0$ **to** $size - 1$ **do**
6. $\Phi\_sum.\phi[i] \leftarrow \Phi1.\phi[i] + \Phi2.\phi[i]$

bsplnd_delete($\Phi\{\downarrow\uparrow\}$)
1.   **if** $\Phi.\phi \ <>$ NIL **then**
2.   Free($\Phi.\phi$)

The listing for *bsplnd_fit* follows. Notice that the parameter list matches the BA algorithm *bsplnd_fitUL*, except that the dependent data is strictly an input parameter (z), and the addition of parameter h, the number of levels of B-spline refinement. Indeed, when $h = 0$ *bsplnd_fit* reduces to *bsplnd_fitUL*. Since *bsplnd_fit* is a public routine in implementation (as opposed to *bsplnd_fitUL*) we don't edit the user's data. Note that in returned lattice Ψ, data field M is not the same as input parameter M. Each level in the hierarchy above level zero doubles the lattice size in the call to *bsplnd_finer*. The algorithm directly maps to Figure 2b.

bsplnd_fit(p, D, x, R, z, M, $xMin$, $xMax$, h, $\Psi\{\uparrow\}$)
1   ▷ current lattice in hierarchy: Φ
2   ▷ current unrefined lattice: Ψ
3   ▷ current refined lattice: Ψ'
4     $\Psi.\phi \leftarrow$ NIL   ▷ initialize; important for memory reuse
5   ▷ Storage for the delta of the lattice from the scattered data points.
6   ▷ Start with the data itself.
7   Alloc(dz, pR)
8   **for** $c \leftarrow 0$ **to** $p - 1$ **do**   ▷ Loop #1
9   **for** $r \leftarrow 0$ **to** $R - 1$ **do**   ▷ Loop #1.1
10           $dz[r + cR] \leftarrow z[r + cR]$
11   ▷ Calculate one control lattice Ψ = to the control lattice hierarchy
12   **for** $k \leftarrow 0$ **to** h **do**   ▷ Loop #2
13       $AtFirstLevel \leftarrow (k = 0)$
14   **if** $\Psi.\phi \ <>$ NIL **then**
15   bsplnd_delete(Ψ)   ▷ memory reuse
16   ▷ $\Psi = \Phi + \Psi'$ (Figure 7b)
17   **if** $AtFirstLevel$ **then**
18   bsplnd_fitUL(p, D, x, R, dz, TRUE, M, $xMin$, $xMax$, Ψ)
19   **else**
20   bsplnd_add(Φ, Ψ', Ψ)
21       $AtLastLevel \leftarrow (k = h)$
22   **if not** $AtLastLevel$ **then**
23   bsplnd_delete(Ψ')   ▷ memory reuse
24   ▷ Refine Ψ into Ψ' (Figure 2b)
25   bsplnd_refine(Ψ, Ψ')
26   ▷ Next finer Φ in hierarchy from current deviation (Figure 2b)
27   **if** $AtFirstLevel$ **then**
28   bsplnd_finer(Ψ, p, x, dz, Φ)
29   **else**
30   bsplnd_finer(Φ, p, x, dz, Φ)
31   **end**

```
32  end
33  Ψ.h ← h              ▷ copy levels of refinement into lattice object
34  bsplnd_delete(Ψ′)    ▷ cleanup
35  bsplnd_delete(Φ)
36  Free(dz)
```

4.2.4.1 The BSPLND Tolerance-Based MBA Algorithm

The bsplnd_fit algorithm can be modified to accept a tolerance or measure of accuracy the function must achieve in its approximation of the scattered data. After calculating the current value of Ψ in line 18 (or 20), we can modify *bsplnd_fit* to measure the approximation error of Ψ and exit the loop if it falls below the user-specified tolerance (a new parameter). Of course, we also modify Loop #2 to be condition-based on the error meeting the tolerance rather than iteration-based on input h. Given p instances of scattered dependent data $z = \{(z_{0,c}, z_{1,c}, \ldots, z_{R-1,c})\}$, we denote the deviation of Ψ from z as $\Delta z = \{(\Delta z_{0,c}, \Delta z_{1,c}, \ldots, \Delta z_{R-1,c})\}$. We calculate the root mean square error with

$$e = \sqrt{\frac{\sum_{r=0}^{R-1}\sum_{c=0}^{p-1} \Delta z_{r,c}^2}{p}} , \quad (15)$$

where as before p is the number of scattered data points.

Note that if the scattered data is not truly functional in nature, then the tolerance specified by the user may never be reached. The tolerance-based BSPLND MBA algorithm, *bsplnd_fitToTol*, requires a maximum number of levels of B-spline refinement to use (a maximum value for h) as input so that it can halt if the calculated function is not meeting the tolerance.

5. BSPLND Implementation

The BSPLND package has been written in the C programming language. The library consists of four primary routines: *bsplnd_fit*, *bsplnd_fitToTol*, *bsplnd_eval*, and *bsplnd_delete*. In addition to these, the library comes with three additional routines that are used by the MBA implementers *bsplnd_fit* and *bsplnd_fitToTol* to perform their work: *bsplnd_finer*, *bsplnd_refine*, and *bsplnd_add*. They are included in the library for the advanced user who would like to write routines similar to *bsplnd_fit* and *bsplnd_fitToTol*. A user's guide to the library is in available in the form of a manual page. The user's guide includes the function prototype and gives a detailed specification of each routine. It also lists all error conditions reported by the library. The BSPLND algorithms mentioned earlier that are not in the public interface to this library have a corresponding private routine in the library (e.g. *bsplndfit_UL*).

5.1 Application

BSPLND is a general-purpose program that works with data having an arbitrary number of dimensions for both its domain and range. Therefore it can be applied in a variety of applications, such as

- wind velocity in a three-dimensional volume
- altitudes on a map
- a compressed multicolor image
- fluid flow in a river
- tissue density in a CAT or MRI scan
- temperature in a furnace

5.2 Future Work

The implementation of one additional algorithm in [1], the Adaptive BA algorithm would be a valuable addition to BSPLND. The algorithm ensures interpolation of the data by using the MBA algorithm to a number of levels in the hierarchy such that each control point has a single point in its proximity data set. This is similar to the BSPLND routine *bsplnd_fitToTol*. However, because a single pair of close data points may require Φ_h to become very dense even though all other data points are sparsely distributed, the adaptive approach is to store only those control points that lie in the 4×4 neighborhood of each data point (in the 2D case), thereby conserving memory. Another useful addition to the library would be an integration routine that would accept the approximation function defined by a control lattice Φ, along with an interval that is some subset of the domain, and integrate the function over the specified interval. For example, integrating a density function over a three-dimensional volume would give the total mass of the substance in that volume.

6. ACKNOWLEDGMENTS

7. REFERENCES

[1] Lee, Wolberg, Shin, "Scattered Data Interpolation with Multilevel B-Splines", *IEEE Transactions on Visualization and Computer Graphics*, Vol. 3, No. 3, July-September 1997

[2] Cormen, et. al., "Introduction to Algorithms", The MIT Press, Cambridge, Massachusetts, 1990.

[3] Wolfram, S., "Mathematica: A System for Doing Mathematics by Computer", 2nd edition, Addison-Wesley Publishing Company, Reading, Massachusetts, 1993.

[4] Phillips, et. al., "Geomview Manual", Software Development Group, The Geometry Center, University of Minnesota.