PRESENTATION AND ANALYSIS OF A MULTI-DIMENSIONAL INTERPOLATION FUNCTION FOR NON-UNIFORM DATA: MICROSPHERE PROJECTION

A Thesis

Presented to

The Graduate Faculty of The University of Akron

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

William Dudziak

August, 2007

PRESENTATION AND ANALYSIS OF A MULTI-DIMENSIONAL INTERPOLATION FUNCTION FOR NON-UNIFORM DATA: MICROSPHERE PROJECTION

William Dudziak

Thesis

Approved:

Advisor
Yingcai Xiao

Faculty Reader
Zhong-Hui Duan

Faculty Reader
Kathy J. Liszka

Department Chair
Wolfgang Pelz

Accepted:

Dean of the College
Roger B. Creel

Dean of the Graduate School
George R. Newkome

Date

ABSTRACT

When dealing with randomly located or clustered data, interpolation error will vary as the distance to the nearest sample or cluster of samples. The current predominant methods for interpolating non-uniform data are not guaranteed to handle this variability of error well. The non-uniformity of the error surface can easily lead to gross misinterpretations of the interpolated values by the end user.

In order to address this limitation of the existing algorithms, this paper examines a method based on the physical structure of an infinitesimally small sphere at the point of interpolation. Using this structure we are able to interpolate based on the 'illumination' of nearby sample points.

Our analysis shows that Microsphere Projection is a viable interpolation technique, and in some cases surpasses the abilities of existing techniques. In one dimension, Microsphere Projection proves to be as accurate as piecewise cubic spline interpolation. In two dimensions, the accuracy of Microsphere Projection seems to outperform thin-plate spline interpolation; and in three dimensions its performance is at least on par with existing techniques. In hyper dimensions it is expected that Microsphere Projection will be even more useful due to its stable extrapolation properties.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Because of modern digital image processing, there exist many extremely precise and well-researched algorithms for interpolating values between regular, abundant sample points. Digital images can be represented as a perfectly-ordered two-dimensional grid of known color values. This grid can be resized or distorted in a number of ways using a host of algorithms, most common of which are Nearest-Neighbor, Bilinear, and Bicubic interpolations [7]. Although these algorithms perform remarkably well with a perfectly-ordered grid of sample points, their usefulness can be quickly outlived when the provided data is non-uniformly distributed across the sample space.

With variability of location, variability of interpolation error increases as well. The existing methods have weaknesses when dealing with the most error-prone areas. These weaknesses include over-smoothing of the interpolation region, and large instabilities of the interpolation surface near the edges of the sampled region. Both over-smoothness and instability can easily lead to misinterpretations by the end user when visualized [8].

Though perfectly gridded 2 or 3-dimensional sample locations are the ideal, the practicality of sensing data at precisely the correct grid locations is difficult if not impossible in many applications.

Examples of non-uniform data sources include:

- Detecting soil pollution levels at various depths in an area.
- Measurements of furnace temperature at various locations.
- Mineral concentrations at various depths.
- Pressure values at various points on the surface of a wing.
- EEG measurements from electrodes attached to the scalp.

The most common non-uniform patterns of samples include [5]:

- Linear arrangements of sample points intersecting the volume (e.g., drill holes).
- Planar arrangements of sample points intersecting the volume (e.g., slices).
- Clusters of sample points such that there are many groups of samples close together with large distances between the groups.

The algorithm introduced in this paper aims to provide a means of interpolating multi-dimensional data which is accurate, stable, and can assure more intuitive results across the extremes of the interpolation surface. Since Microsphere Projection is designed primarily to address non-uniform data, this paper will restrict its discussion to the set of algorithms which are designed to handle non-uniform data. This will exclude both Bilinear and Trilinear interpolations.

In Chapter 2, we discuss existing non-uniform interpolation methods and their implementations. Chapter 3 provides a detailed analysis of the Microsphere Projection algorithm. In Chapter 4 we present an analysis and comparison of various 1D, 2D and

3D case studies and the experimental results obtained from those studies. Chapter 5 contains concluding remarks and notes for future work.

1.1 Local vs. Global Interpolation

All interpolations are based on a set of sample points; these are points in space with known values. Local interpolations are methods which make use of the information from only a small set of nearby sample points, and global interpolations attempt to make use of the entire set of sample points. Local interpolations are common in one-dimensional interpolations, some of which will be discussed later; however these localization methods come at a cost in higher dimensions. In two-dimensions and higher, it is increasingly difficult to 'localize' the sample points without loosing differentiability (a key property) of the interpolation, so all of the common interpolation techniques in 2D and 3D tend to be global techniques.

Depending on how the terms 'local' and 'global' are interpreted, Microsphere Projection may be placed in either category. In the degenerate one-dimensional case, Microsphere Projection mimics the behavior of a local interpolation. However, in higher dimensions, the number of sample points affecting the interpolated value is not necessarily limited.

1.2 Exact vs. Inexact Interpolation

Depending on the application, the values sampled at the sample locations may have an error range. If the values are not known with certainty, then it is a common

practice to use an inexact approximation which follows the general trend of the data, and is not guaranteed to pass through any of the data points exactly. See Figure 1.1.

However, if the primary loss of information (the primary difficulty) with the data is not regarding error in the sample value, but in scarcity of data locations, then an exact approximation is better suited to interpolate the data. The Microsphere Projection algorithm is an exact interpolation; and since we would prefer to compare the algorithm to others with similar behavior characteristics, all of the algorithms and methods discussed in this paper are various forms of exact interpolation functions.

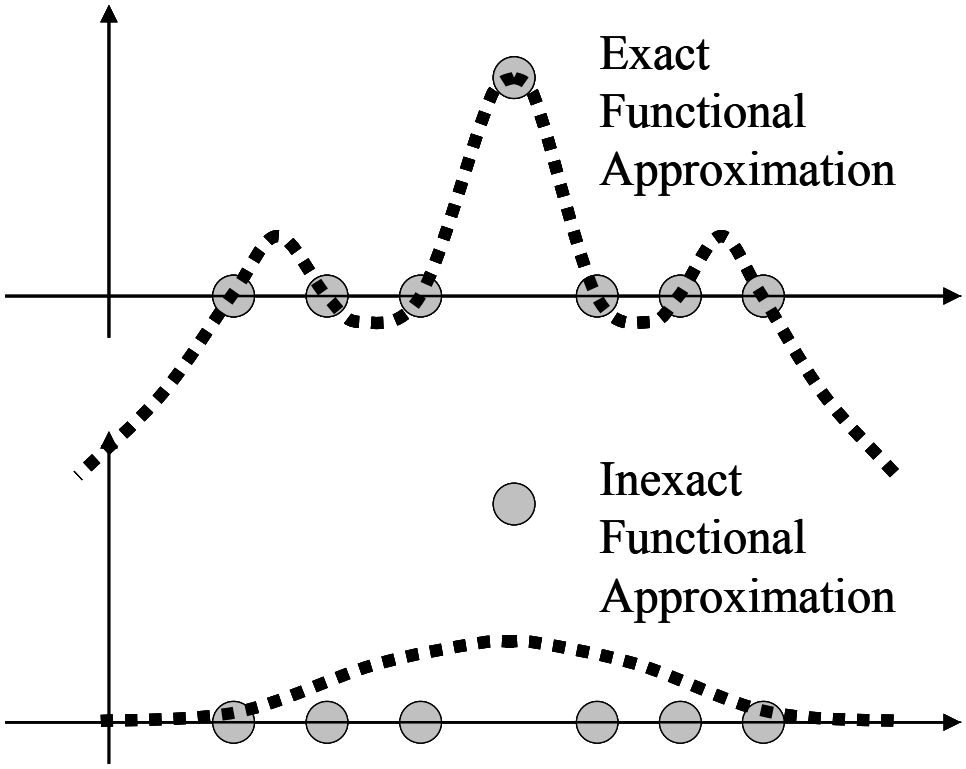


Figure 1.1. Comparison of exact and inexact functional approximations.

1.3 Differentiability Classes

Differentiability is a valuable property of an interpolation algorithm. It provides not only mathematical benefits, but also guarantees a visually smooth image. Differentiability refers to the ability to take derivatives over the line, surface or volume. Differentiability Classes are differentiated by the number of derivatives which one can take before the function becomes either zero throughout or non differentiable. These classes are written C^0, C^1, C^2, etc. If a function is C^0 then this indicates that the function is either non-differentiable, has a discontinuous first derivative, or the first derivative is 0 everywhere. A function that is C^N has a continuous N-1(th) derivative, however the Nth derivative is either non-differentiable or 0 everywhere.

1.4 Interpolation vs. Extrapolation

In general, interpolation is defined as the "guessing" of values within the convex hull formed by the sample point locations. "Guessing" at values beyond the convex hull constitutes extrapolation; even if the points are within the bounding box. For a visual example of this relationship, see Figure 4.37.

CHAPTER II

SURVEY OF EXISTING NON-UNIFORM DATA INTERPOLATION METHODS

Non-uniform data interpolation is a well-researched field with a wide variety of existing algorithms. These algorithms have may strengths and weaknesses dependant on the context and dimensionality in which they are used. The following sections discuss a few of the popular algorithms.

2.1 Nearest Neighbor Interpolation

Nearest Neighbor Interpolation is perhaps the most simplistic method for interpolating data. As the name implies, the algorithm chooses the interpolated value to be equal to the value of the sample point which is closest to the interpolation location. See Figure 2.1.

Though exhibiting excellent execution time, NN interpolation has several drawbacks when applied to real data. These include non-differentiability (class C^0), extremely high error rates, and non-intuitive visual results.

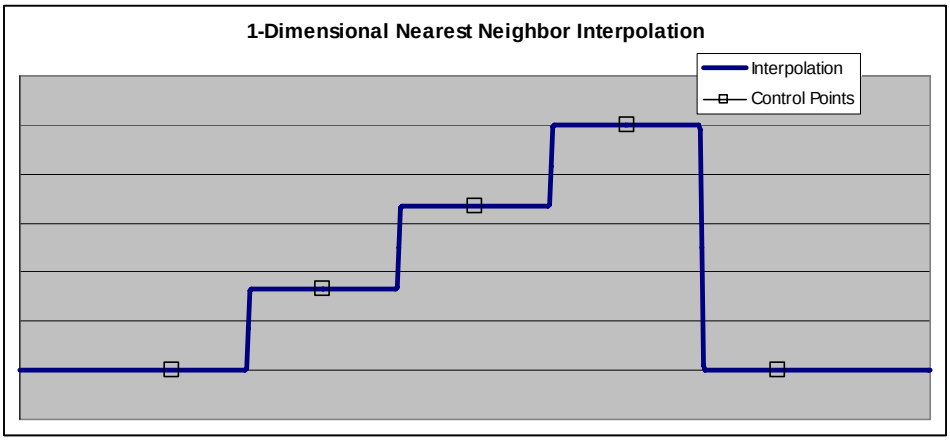


Figure 2.1. Example of Nearest Neighbor interpolation.

2.2 Polynomial Interpolation

Polynomial interpolation involves the mapping of a polynomial function to approximate the sampled data values. The constants in the polynomial function can be derived easily by solving a Vandermonde matrix populated with values derived from the sampled points [4]. See Formula 2.1.

$$\begin{bmatrix} 1 & x_0 & x_0^{\,2} & \cdots & x_0^{\,n-1} \\ 1 & x_1 & x_1^{\,2} & \cdots & x_1^{\,n-1} \\ 1 & x_2 & x_2^{\,2} & \cdots & x_2^{\,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^{\,2} & \cdots & x_{n-1}^{\,n-1} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} \qquad 2.1$$

Definition of an interpolating polynomial using a Vandermonde matrix. x_n represnts the 1D location of the sample, y_n represnts the value sampled at that location. The one-dimensional interpolating polynomial is defined by $c_0 + c_1x + c_2x^2 + \ldots + c_{n-1}x^{n-1}$.

A pure polynomial model of the data is very straightforward; however this approach has significant drawbacks. Even simple polynomial regressions can produce functions that oscillate wildly between sample points. The oscillation is directly related to the degree of the polynomial and the distance from the center of mass of the sample points. This property of polynomial interpolations is referred to as Runge's Phenomena. See Figure 2.2.

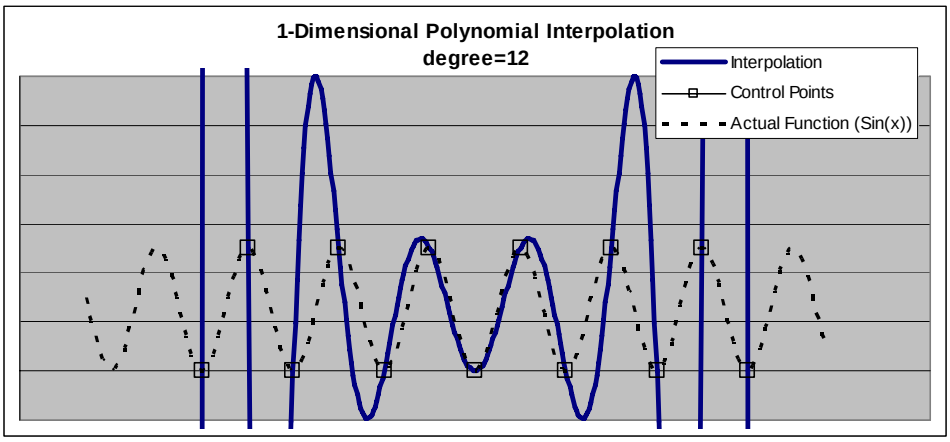


Figure 2.2. Example of polynomial functional approximation.

2.3 Shepard's Method Interpolation (Inverse-Distance Weighting)

Aside from using Nearest Neighbor, Shepard's Method is the most simplistic method to interpolate irregular data. The implementation of Shepard's Method is very similar to the method for calculating a body's center of mass. First, the algorithm introduces a 'weight' assigned to each sampled point which is inversely proportional to the distance between the sample and the interpolation location. The final interpolated value is given by: Sum(weight*sampleValue) / Sum(weight) [6] (see Formula 2.2). Given perfectly random sample locations, Shepard's Method will excel in simplicity and

accuracy; however there are a few drawbacks when dimensionality and clustering of sample locations are taken into consideration.

$$f(\mathrm{V}) = \frac{\sum_{i=1}^{N} v_i d_i^{-p}}{\sum_{i=1}^{N} d_i^{-p}} \qquad 2.2$$

Mathematical form of Shepard's Method of interpolation. The term d_i is the Euclidian distance between location V and sample point i. The term p is an arbitrary inverse distance propagation power ($p > 0$, $p = 2$ is accepted as standard). v_i is the value of sample point i. N is the number of sample points.

Shepard's Method does not adequately handle dense clusters of redundant data. Since the only information taken into consideration when determining weights is that of distance from the interpolation location, sample points with identical values and nearly the same coordinates will inappropriately bias the interpolated value. An extreme case of this can be seen in Figure 2.3 where there are only 3 sample points, two of which have the same value and nearly the same coordinate position.

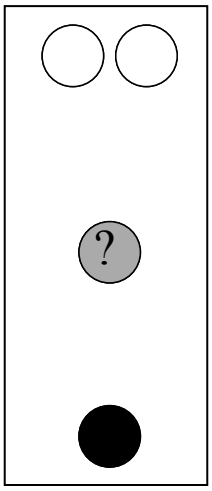


Figure 2.3. Illustration of the problem with naïve inverse distance weighting. The interpolation in the middle should have a value close to ½ black, ½ white… however naïve inverse-distance weighting suggests it should be 2/3 white, 1/3 black.

Another difficulty with Shepard's Method is the choice of the 'propagation of influence' power. Classically, this variable is set to '2', as physical propagation through three dimensions typically occurs as the inverse of the distance squared. However, when dealing with 2-dimensional or N-dimensional interpolation problems, an obvious choice for this value does not present itself easily [5].

2.4 Cubic Spline Interpolation

The term 'spline' originated in the architect's draft room where when a curve was needed, a very thin piece of wood would be fit between the points, bent slightly, and traced. Cubic spline interpolation is designed for 1-dimension, and is based on fitting localized cubic polynomials to each segment of the graph such that the entire interpolation has a continuous second derivative (class C^2). See Figure 2.4.

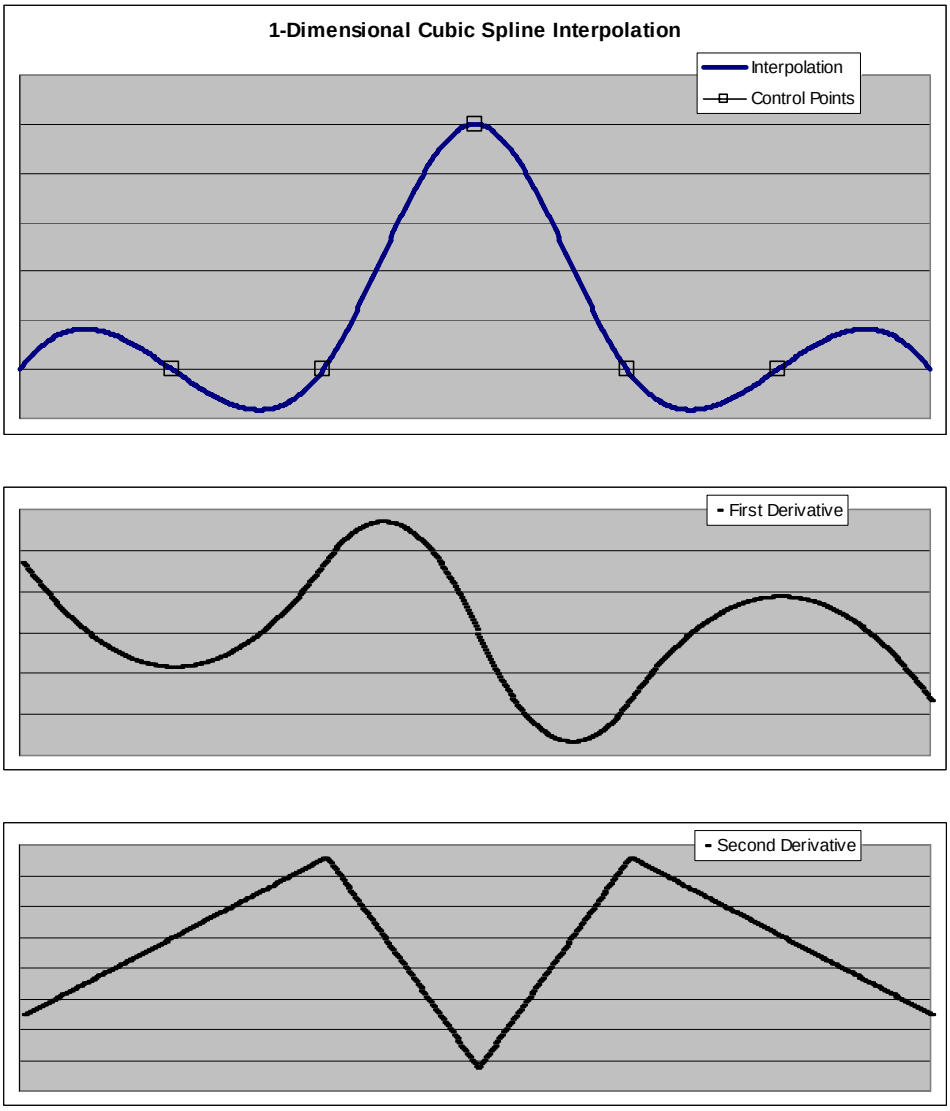


Figure 2.4. Interpolation of a set of data points using a cubic spline. The derivates have been included to demonstrate the second-differentiability of the interpolation (Class C^2).

The cubic spline interpolation is very good at providing a smooth approximation of 1-dimenional data. The major drawbacks are the inherently oscillatory nature of the interpolation, and the inextensibility to higher dimensions.

2.5 Thin-Plate Spline Interpolation

Thin-plate spline interpolation is a common interpolation method for 2-dimensional data. Much like cubic spline interpolation for 1-dimension, thin-plate splines are based on a physical process. The physical process approximated is that of 'bending' what would be a thin, flat metal plate over the x-y coordinate grid, to intersect the values of the sample points in the z-direction above or below the x-y plane [1]. See Formulas 2.3 and 2.4.

$$f(x,\mathrm{y}) = \mathrm{c}_1 + c_2\mathrm{x} + c_3 y + \sum_{i=1}^{N} b_i {d_i}^2 \log(d_i) \qquad 2.3$$

Mathematical form of the Thin-Plate Spline interpolation. The term d_i is the Euclidian distance between location (x,y) and sample point i [1].

$$\begin{bmatrix} 0 & {d_{12}}^2\log(d_{12}) & {d_{13}}^2\log(d_{13}) & \cdots & {d_{1n}}^2\log(d_{1n}) & 1 & x_1 & y_1 \\ {d_{21}}^2\log(d_{21}) & 0 & {d_{23}}^2\log(d_{23}) & \cdots & {d_{2n}}^2\log(d_{2n}) & 1 & x_2 & y_2 \\ {d_{31}}^2\log(d_{31}) & {d_{32}}^2\log(d_{32}) & 0 & \cdots & {d_{3n}}^2\log(d_{3n}) & 1 & x_3 & y_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ {d_{n1}}^2\log(d_{n1}) & {d_{n2}}^2\log(d_{n1}) & {d_{n3}}^2\log(d_{n3}) & \cdots & 0 & 1 & x_n & y_n \\ 1 & 1 & 1 & \cdots & 1 & 0 & 0 & 0 \\ x_1 & x_2 & x_3 & \cdots & x_n & 0 & 0 & 0 \\ y_1 & y_2 & y_3 & \cdots & y_n & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_n \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad 2.4$$

Matrix to determine constants b_1 through b_n u, and c_1, c_2, c_3 used in Thin-Plate Spline interpolation. The term d_{ij} is the Euclidian distance between sample point i and sample point j. The term v_i is the value at sample point i [1].

2.6 Volume Spline Interpolation

An extension of the Thin-Plate Spline interpolation to three dimensions, Volume Spline interpolation represents the theoretical 'bending' of a three-dimensional volume. This interpolation method is a very common method used in the interpolation of data in three-dimens. See Formulas 2.5 and 2.6.

$$f(x,y,z) = c_1 + c_2 x + c_3 y + c_4 z + \sum_{i=1}^{N} b_i d_i^{\,3} \qquad 2.5$$

Mathematical form of the Volume Spline interpolation. The term d_i is the Euclidian distance between location (x,y) and sample point i [3].

$$\begin{bmatrix} 0 & d_{12}^{\,3} & d_{13}^{\,3} & \cdots & d_{1n}^{\,3} & 1 & x_1 & y_1 & z_1 \\ d_{21}^{\,3} & 0 & d_{23}^{\,3} & \cdots & d_{2n}^{\,3} & 1 & x_2 & y_2 & z_2 \\ d_{31}^{\,3} & d_{32}^{\,3} & 0 & \cdots & d_{3n}^{\,3} & 1 & x_3 & y_3 & z_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\ d_{n1}^{\,3} & d_{n2}^{\,3} & d_{n3}^{\,3} & \cdots & 0 & 1 & x_n & y_n & z_n \\ 1 & 1 & 1 & \cdots & 1 & 0 & 0 & 0 & 0 \\ x_1 & x_2 & x_3 & \cdots & x_n & 0 & 0 & 0 & 0 \\ y_1 & y_2 & y_3 & \cdots & y_n & 0 & 0 & 0 & 0 \\ z_1 & z_2 & z_3 & \cdots & z_n & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \\ c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_n \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad 2.6$$

Matrix to determine b_1 through b_n, and c_1, c_2, c_3, c_4 used in Volume Spline interpolation. The term d_{ij} is the Euclidian distance between sample point i and sample point j. The term v_i is the value at sample point i [3]..

The primary difference between volume spline interpolation and thin-plate spline interpolation is the modification of the radial basis function from $d^2\log(d)$ in thin-plate spline to d^3 in volume spline interpolation. This can be seen clearly when comparing the sets of linear equations used in calculation of the functional coefficients (see Formulas 2.3 and 2.5)

2.7 Multiquadric Interpolation

Multiquadric Interpolation is an often-used 3-dimensional interpolation which is calculated very similarly to volume splines. The derived function is nearly the same form as volume splines, however the non-radial coefficients are removed and the radial basis function is modified slightly. The radial function includes a term β with β>0, β = 1 is common practice. See Formulas 2.7 and 2.8.

$$f(x,y,z)=\sum_{i=1}^{N} b_i \sqrt{d_i^{\,2}+\beta^2} \qquad 2.7$$

Mathematical form of the Volume Spline interpolation. The term d_i is the Euclidian distance between location (x,y) and sample point i. β is an arbitrary constant, β>0 [3].

$$\begin{bmatrix} 0 & \sqrt{d_{12}^2+\beta^2} & \sqrt{d_{13}^2+\beta^2} & \cdots & \sqrt{d_{1n}^2+\beta^2} \\ \sqrt{d_{21}^2+\beta^2} & 0 & \sqrt{d_{23}^2+\beta^2} & \cdots & \sqrt{d_{2n}^2+\beta^2} \\ \sqrt{d_{31}^2+\beta^2} & \sqrt{d_{32}^2+\beta^2} & 0 & \cdots & \sqrt{d_{3n}^2+\beta^2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sqrt{d_{n1}^2+\beta^2} & \sqrt{d_{n2}^2+\beta^2} & \sqrt{d_{n3}^2+\beta^2} & \cdots & 0 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_n \end{bmatrix} \qquad 2.8$$

Matrix to determine b_1 through b_n used in Multiquadric interpolation. The term d_{ij} is the Euclidian distance between sample point i and sample point j. The term v_i is the value at sample point i. β is an arbitrary constant, $\beta>0$ [3].

CHAPTER III

MICROSPHERE PROJECTION: DESIGN AND IMPLEMENTATION

3.1 Algorithm Description

3.1.1 Physical Premise

Microsphere projection is based on the physical model of an infinitesimally small sphere located at the point of interpolation. This tiny sphere is then 'illuminated' by the surrounding sample points. Based on the degree of illumination on various parts of the sphere by various sample points, a series of weights for all the sample points are assigned. These weights, when applied, yield our interpolated value for the location.

3.1.2 Description of the Sphere

The surface of the Microsphere is divided into a large number of equally-spaced regions. Each region records for itself which sample point has illuminated it the most, and what illumination that sample point has provided. Each surface region is represented by a single unit vector pointing out from the center of the sphere to the center of that region. "S[i].Vector" will be used to represent the unit vector for surface region i. The more regions used, the greater the precision of the interpolation. Throughout Chapter 4: Experimental Results, the value of 2000 was used for the number of surface regions.

For each region, two values are recorded: one recording the index of which sample point has illuminated this section the greatest, and the second recording the degree of illumination from this point. These will be referred to as "S[i].Brightest_Sample" and "S[i]. Max_Illumination", respectively.

Since determining an arbitrarily large number of equally-spaced regions on the surface of a sphere is no small task, we accept that a large number of randomly placed unit vectors will provide a fairly uniform distribution. The vectors are generated using the algorithm in Figure 3.1.

```
do
        // x,y,z are uniformly-distributed random numbers in the range (-1,1)
        x := rand(-1,1)
        y := rand(-1,1)
        z := rand(-1,1)
        vectorSize := sqrt (x*x + y*y + z*z)

// if the vector these points form is outside the unit sphere,
// disregard and find a new vector.
while ( vectorSize > 1 )

// normalize the vector, so that it forms a unit vector for the surface of our sphere.
x := x / vectorSize
y := y / vectorSize
z := z / vectorSize
```

Figure 3.1. Pseudo-code controlling the creation of unit vectors defining spherical segments of a Microsphere in 3-dimensions.

3.1.3 Applying Illumination to the Sphere

Net illumination is applied to the microsphere by iterating through each of the sample points, and applying illumination to the sphere one-by-one. It should be noted that illumination on various parts of the sphere decreases proportionally to the acuteness

of the angle between the surface of the sphere and the direction of the sample point. Illumination also decreases as the distance between the microsphere and the sample point increase. See Figure 3.2. Much like Shepard's Method, this inverse relationship between distance and 'brightness' is governed by a power value 'p' specified by the user where p>0, p=1 and p=2 are typical values. p=1 yields an interpolation that is C^0 (non-differentiable), p > 1 is C^1 (first-derivative is continuous). Similar to Shepard's Method, as $p \rightarrow \infty$, the closest points dominate the interpolation and the algorithm becomes the equivalent of Nearest Neighbor.

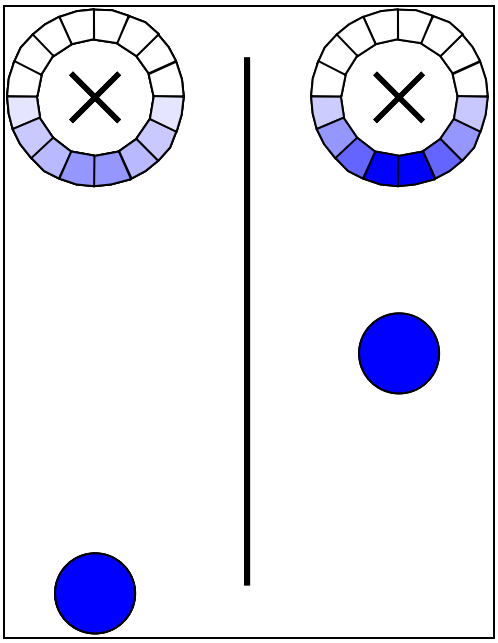


Figure 3.2. Illumination of a 2-D Microsphere by a single sample point in two separate cases. Note how the acuteness of the surface section and the distance of the sample point are taken into consideration. Sphere precision = 16.

Because we are using an inverse distance weight for each point (ie, $1/d^p$), when d=0 we have an issue. Thus, a simple check is implemented early on in the algorithm to see if we are interpolating at the location of a sample point. Since Microsphere Projection is an exact interpolation method, we simply return the value of the sample as our interpolated value. The following algorithm located in Figure 3.3 is the algorithm used to calculate the illumination values for each segment of the sphere:

```
for i := 0 to Number of Samples

        // vector connecting the current sample to the interpolation location
        vector1 := sample[i].XYZLocation - interpolation.XYZLocation

        // the distance-modified weight of this point
        // p > 0, typically p=1 or p=2.
        weight := pow(vector1.Size, -p)

        // the value of 'Precision' represents how many subdivisions
        // of the surface of the microsphere we are working with.
        for j := 0 to Precision

                // each sample only 'shines' on one hemisphere.
                // as the angle becomes more acute, the intensity
                // of that shine decreases as the cosine function
                cosValue := CosValueBetweenVectors(vector1, S[j].Vector)

                // if the brightness of the shine on this section of the sphere
                // is more than any other point thus far checked, update our
                // 'Brightest_Sample' and 'Illumination' data.
                if (cosValue * weight > S[j].Max_Illumination)
                        S[j].Max_Illumination := cosValue * weight
                        S[j].Brightest_Sample := i
                endif
        endfor
endfor
```

Figure 3.3. Pseudo-code controlling the application of illumination to the Microsphere.

In the inner-most loop of our algorithm, we have a function call and a simple if-statement. The if-statement requires very little execution time. To determine the cosine of an angle between two vectors, a function is implemented using Formula 3.1.

$$\cos(a,b) = \frac{a \bullet b}{\|a\| \, \|b\|} \qquad 3.1$$

Formula to calculate the cosine of the angle between vector 'a' and vector 'b'.

Since the sizes of both vectors are known in our case, the only calculations needed are with regard to the dot-product. These are fairly straightforward, and require very little execution time by the processor.

3.1.4 Accumulation of the Final Values from the Sphere

Once all the calculations are complete regarding the maximum illuminations on the various sections of the sphere, we must make use of this data to produce a single interpolated value. To do this, we assign a weight to each sample point equal to the total illumination that point provided to the sections of the sphere. Note that each section of the sphere only records data regarding the point which provided the most illumination; sample points which did not out-shine any other points on any section of the sphere are assigned a weight of 0. See Figure 3.4.

```
// accumulate the data from our sphere, and determine final interpolation
value := 0
totalWeight := 0
for i := 0 to Precision
        value := value + S[j].Max_Illumination * sample[S[j].Brightest_Sample].SampledValue
        totalWeight := totalWeight + S[j].Max_Illumination
endfor

// the final interpolated value generated by the algorithm
interpolation := value / totalWeight
```

Figure 3.4. Pseudo-code controlling the accumulation of data from the sphere, and determination of final interpolation value.

3.2 Mathematical Form

Together with Formula 3.1, Formulas 3.2-3.4 present the final mathematical form of the interpolation.

$$w_i = \left\{ \max\left(\left\| l_j - I \right\|^{-p} \cos(s_i, l_j - I) \right) \mid j \in \{1,2,3,\ldots,N\} \right\} \tag{3.2}$$

$$m_i = \left\{ v_j \mid \max\left(\left\| l_j - I \right\|^{-p} \cos(s_i, l_j - I) \right), j \in \{1,2,3,\ldots,N\} \right\} \tag{3.3}$$

$$f(I) = \begin{cases} \{ v_i \mid I = l_i, i \in \{1,2,3,\ldots,N\} \} \\ \dfrac{\sum_{i=1}^{P} m_i w_i}{\sum_{i=1}^{P} w_i} \quad otherwise \end{cases} \tag{3.4}$$

I = Location of interpolation
p = Propagation of influence power, p>0
v_i = Value of sample i, $i \in \{1,2,3,\ldots,N\}$
l_i = Location of sample i, $i \in \{1,2,3,\ldots,N\}$
N = Number of samples
s_i = Evenly spaced unit vector on surface of sphere, $i \in \{1,2,3,\ldots,P\}$
P = Precision (number of unit vectors on sphere), $P >>$ 2D
d = Dimensionality of data (d=2 is planar)

3.3 Runtime Analysis

Runtime analysis is important in understanding how the running time of the algorithm will increase or decrease with the change in quantity of inputs. The next few lines describe the behavior of the running time for Microsphere Projection.

N = Number of samples

P = Precision (number of unit vectors on sphere)

Determining the position of the unit vectors: O(P).

Calculating maximum illumination and illuminating point for each surface section: O(P*N).

Accumulating the final interpolation value: O(P).

Overall Runtime: O(P*N).

3.4 Strengths of Microsphere Projection

MS Interpolation exhibits the Maximum Principle. In other words, the interpolated value is guaranteed to lie in the range between the minimum sampled value and the maximum sampled value. Other interpolation techniques which demonstrate this quality include Nearest Neighbor interpolation, and Shepard's Method (naïve inverse-distance weighting). This feature was chosen because it is guaranteed to provide intuitive results for bounded data.

MS Interpolation is guaranteed to preserve monotonic and strict monotonic behavior over any set or subset of sample points. For example, if the set or subset of

sample points is increasing or strictly increasing over a range, then the interpolation is guaranteed to be increasing or strictly increasing over the same range.

MS Interpolation demonstrates no oscillatory behavior between sample points, unlike functional approximations which are designed to preserve high differentiability. See Figures 2.3 and 2.6 for examples of oscillatory functional behavior.

MS Interpolation provides a stable extrapolation ability. Functional approximations tend to produce extremely volatile extrapolation results beyond the range of the data points (see Figure 2.3). This can cause serious issues in higher dimensions where the differentiation between interpolation and extrapolation within the volume is difficult to determine (see Section 4.4). Because MS Interpolation provides a stable extrapolation, it has considerable benefits over functional approximations when visualizing higher dimensional data. This will be discussed further in following sections.

3.5 Weaknesses of Microsphere Projection

MS Interpolation is only class C^1 so long as $p>1$, C^0 otherwise. This means that the interpolation will have a continuous first derivative, however no guarantees are made of the second derivative of the interpolation.

Depending on the nature of the problem, the fact that MS Interpolation exhibits the Maximum Principle can be an issue. That the interpolation method is unable to interpolate a value beyond the scope of the sampled values can cause problems depending on the context.

Depending on the size of the data set and other considerations, MS Interpolation can require more computation time than some of the other interpolation algorithms.

Though the overall runtime is O(P*N), this set of calculations must be run every time a point is to be interpolated. Radial Basis Function (RBF) interpolations such as Thin-Plate Spline, Multiquadric, and Volume Spline are all O(N^2) (using Gaussian elimination) for the first interpolation and O(N) (with a very small overhead) for subsequent interpolations.

As interpolation location approaches a sample point, the first derivative in all dimensions approaches 0 when p>1. In most contexts this is undesirable behavior; however it is necessary if we wish to preserve the Maximum Principle in conjunction with differentiability.

CHAPTER IV

EXPERIMENTAL RESULTS

4.1 One-Dimensional Interpolation

4.1.1 Aberration Test – Case Study

This test simulates a series of constant values with a single aberrant value placed in the middle. Ideal interpolative behavior should show the values towards the edges approaching the horizontal asymptote formed by the lower sample points. This test is representative of many situations where the sampling period is larger than the period of the underlying function or event. In other words, the data could be said to be 'sparse' or the underlying function to be 'volatile' or 'turbulent'.

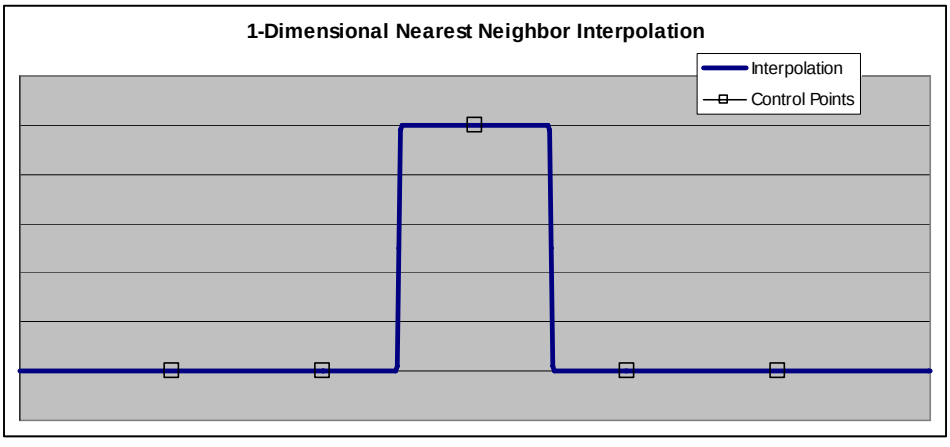


Figure 4.1. Nearest-Neighbor interpolation of simple aberration data set.

Nearest-Neighbor interpolation forms a fairly 'boxy' interpolation of the points and is discontinuous between sample points with different values. See Figure 4.1.

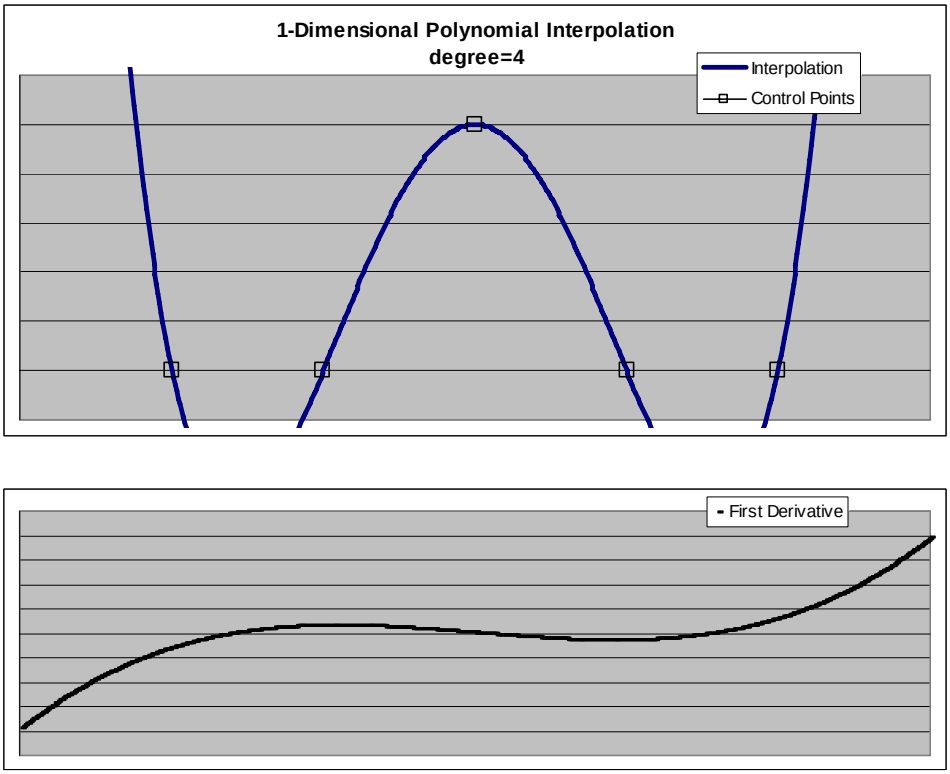


Figure 4.2. Polynomial interpolation of simple aberration data set.

Polynomial interpolation provides high differentiability (C^3), though gross error margins when interpolating farther from the center. Extrapolation is not tenable with this model. See Figure 4.2.

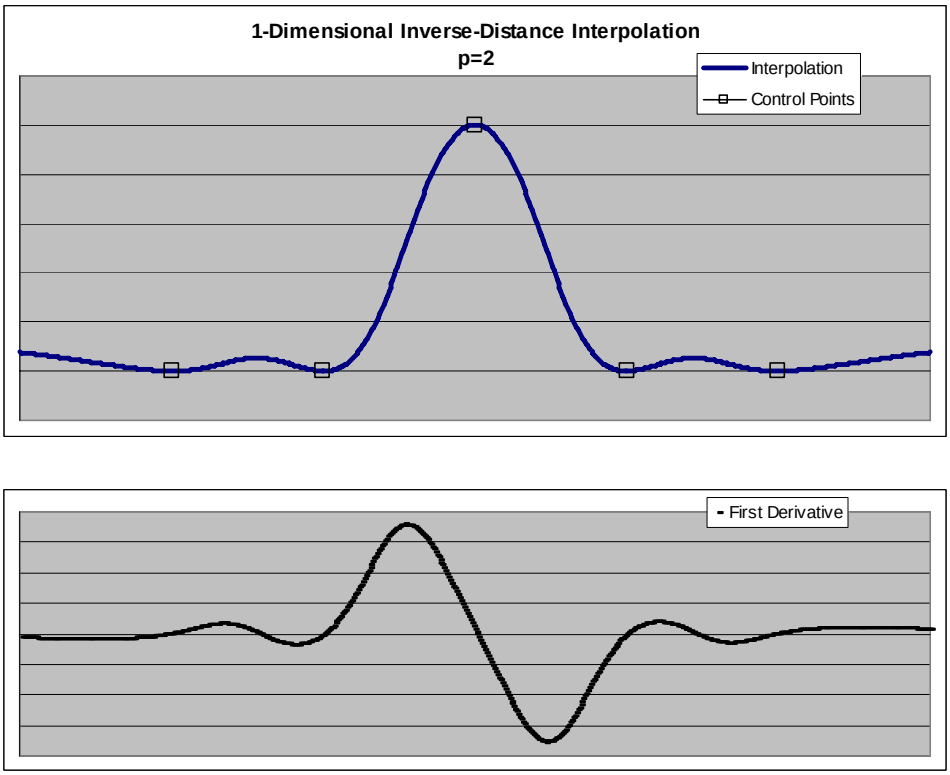


Figure 4.3. Shepard's Method (naïve inverse-distance) interpolation of simple aberration data set.

Inverse-Distance interpolation forms a smooth C^1 interpolation, however the interpolation values on the edges are not 'flat'. Extrapolation values approach the average value of all sample points. See Figure 4.3.

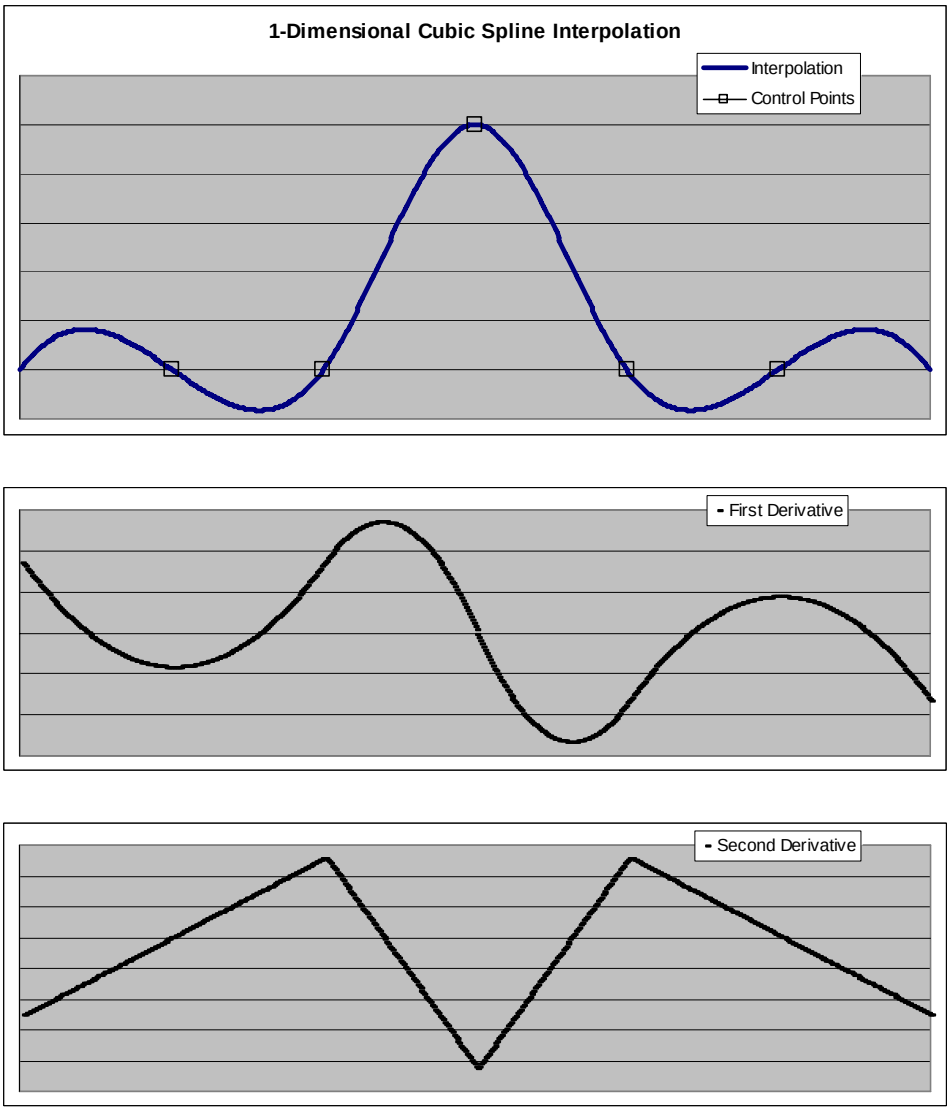


Figure 4.4. Cubic Spline interpolation of simple aberration data set.

Cubic Spline does not provide the 'flatness' on the edges we would hope for. Extrapolation is not tenable with this model. See Figure 4.4.

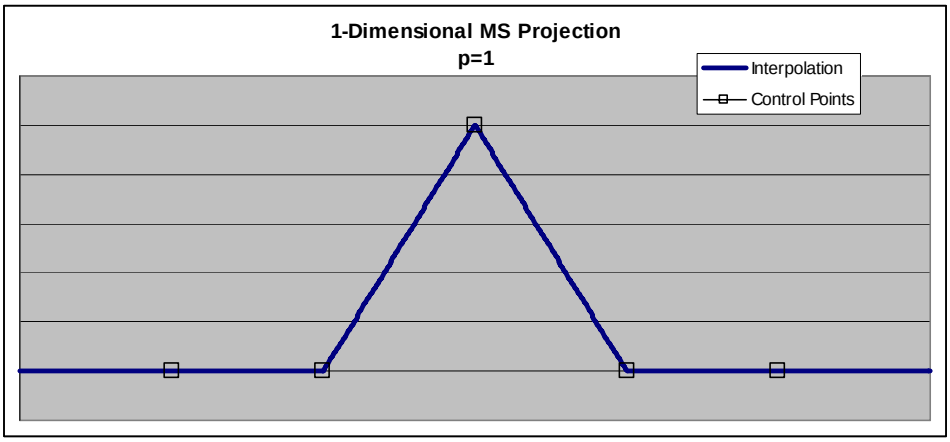


Figure 4.5. Microsphere Projection, p=1 interpolation of simple aberration data set.

Microsphere Projection with p=1. Setting p=1, Microsphere Projection becomes the 1-dimensional equivalent of piecewise linear interpolation. This model satisfies our endpoint criteria, however its non-differentiability may cause problems.

Note that this method is not equilivent to naïve inverse-distance interpolation because it generates a piecewise interpolation. Shepard's method assigns non-zero weights to all points in 1D, however MS Interpolation in 1D only assigns non-zero weights to the local points (those on the left and right of the interpolation location). See Figure 4.5.

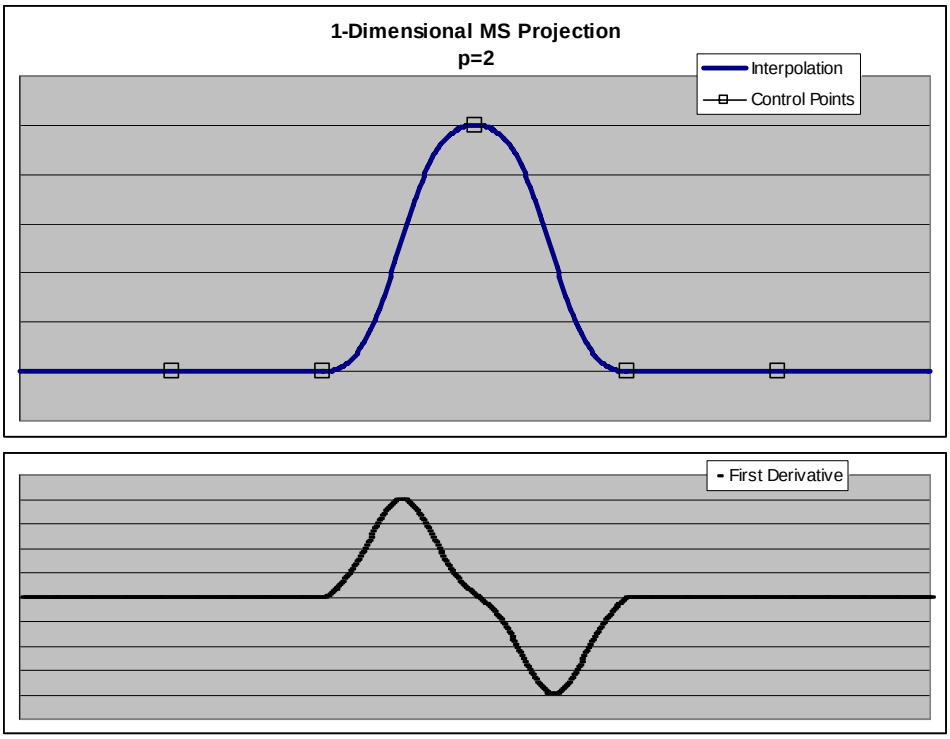


Figure 4.6. Microsphere Projection, p=2 interpolation of simple aberration data set.

Microsphere Projection with p=2. Setting p=2 provides the differentiability that is lacking with p=1. The graph can be described as a smoothed version of piecewise-linear or nearest neighbor interpolation. This method satisfies our endpoint expectations. See Figure 4.6.

4.1.2 Stair-Climb Test – Case Study

This test simulates a series of linearly increasing values with an aberrant value at the end of the sequence. Ideal interpolative behavior should linearly connect the sample points or at the very least preserve the monotonic behavior of the first four points, with a smooth curve to the aberrant point at the end. This test is representative of many

situations where the period of the underlying function is much larger than the period of the samples. In other words, the data set could be regarded as smooth or the sampling as being dense.

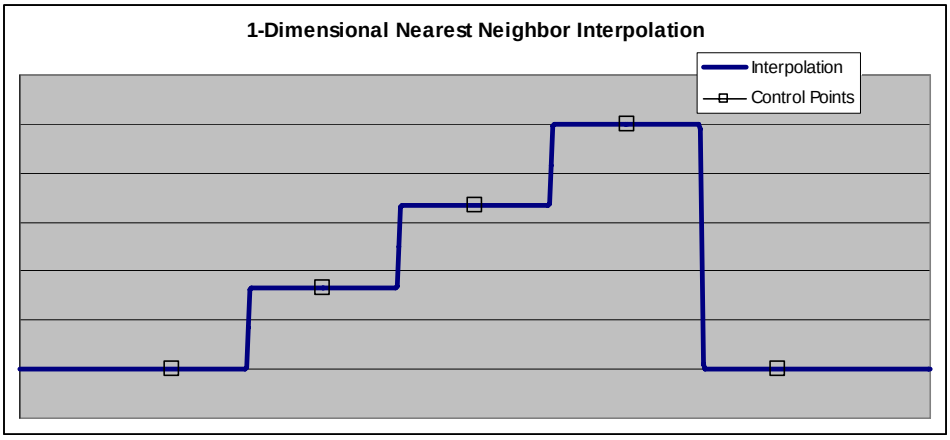


Figure 4.7. Nearest-Neighbor interpolation of simple smooth data set.

Nearest-Neighbor's 'boxy' interpolation of the points is undesired. The discontinuities (upward jumps in the graph) between the sample points are undesirable as well. See Figure 4.7.

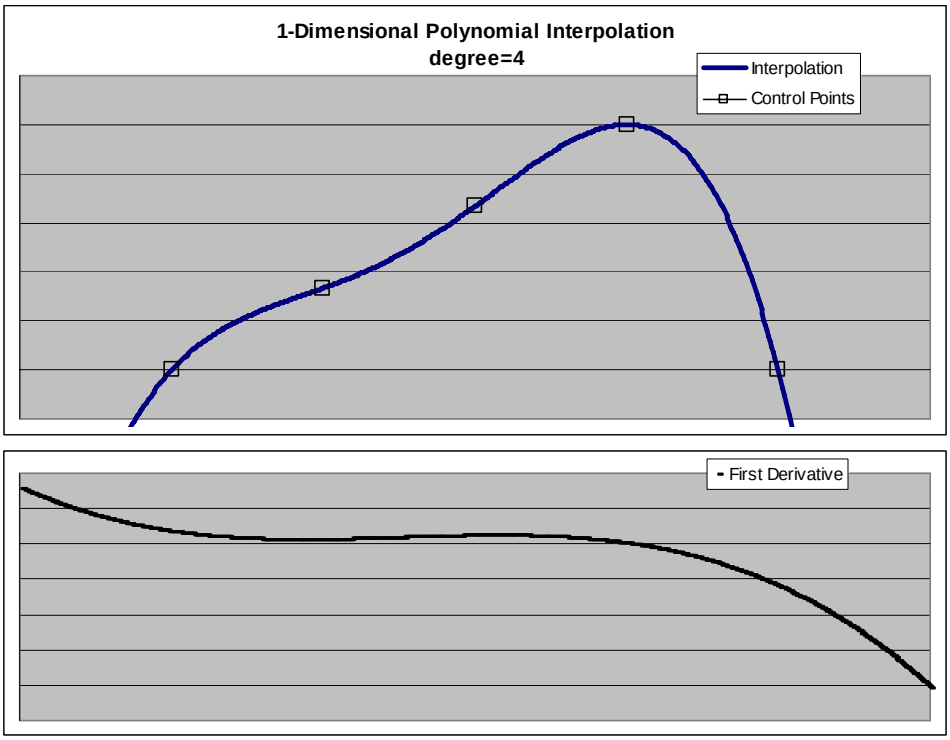


Figure 4.8. Polynomial interpolation of simple smooth data set.

Polynomial interpolation provides high differentiability (C^3), and seems to work very well for this data. Extrapolation is not tenable using this interpolation model. See Figure 4.8.

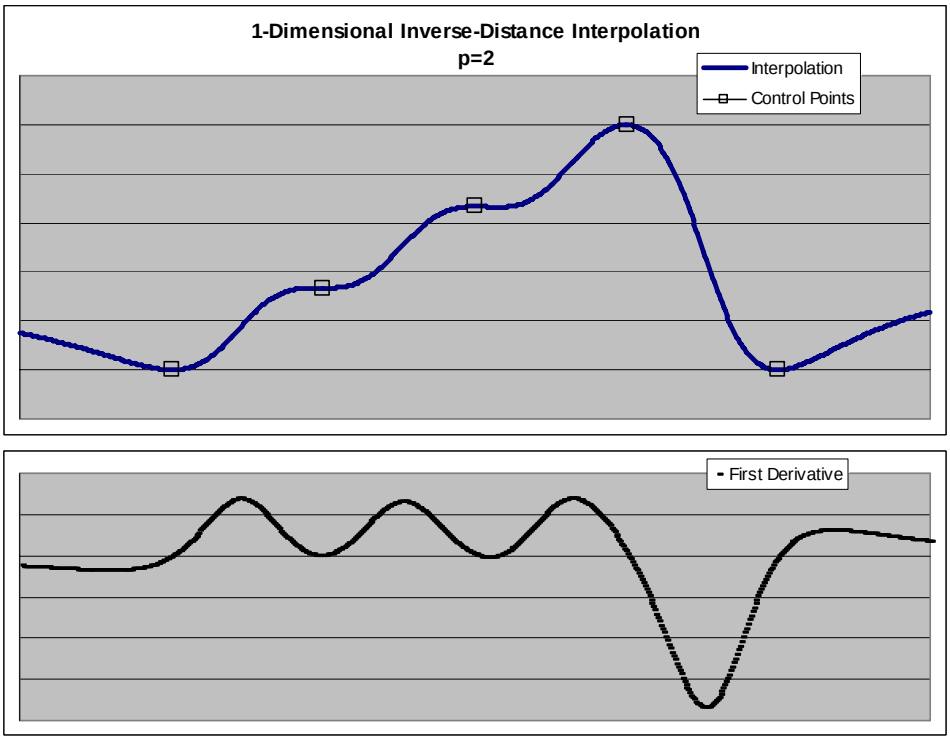


Figure 4.9. Shepard's Method (inverse-distance) interpolation of simple smooth data set

Inverse-Distance interpolation forms a smooth C^1 interpolation, however the derivative values at the intersection of the sample points is always 0 and the monotonic behavior of the first four sample points is not preserved in the interpolated values. Extrapolation values approach the average value of all sample points at the edges. See Figure 4.9.

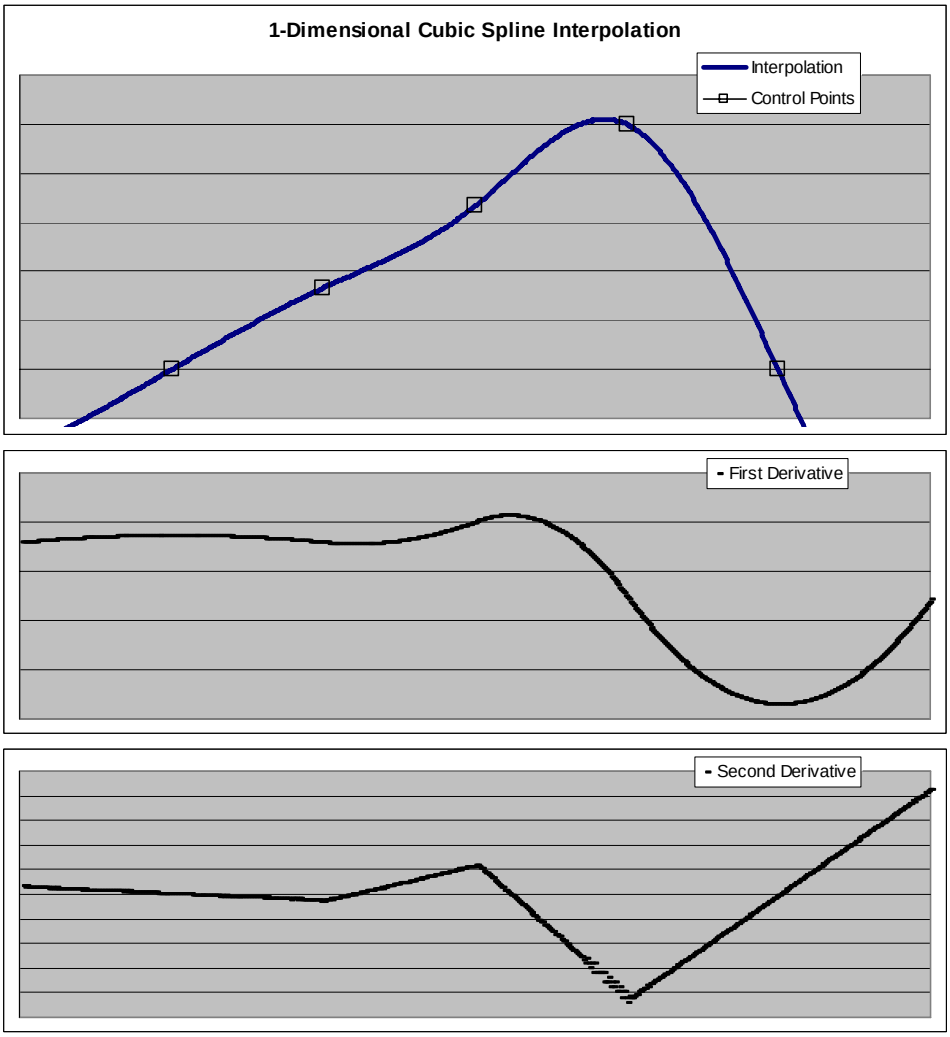


Figure 4.10. Cubic Spline interpolation of simple smooth data set.

Inverse-Cubic Spline interpolation works fairly well in this case. Only for a very short period at the peak is the monotonic behavior of the interpolation incorrect. Extrapolation is not tenable using this interpolation model. See Figure 4.10.

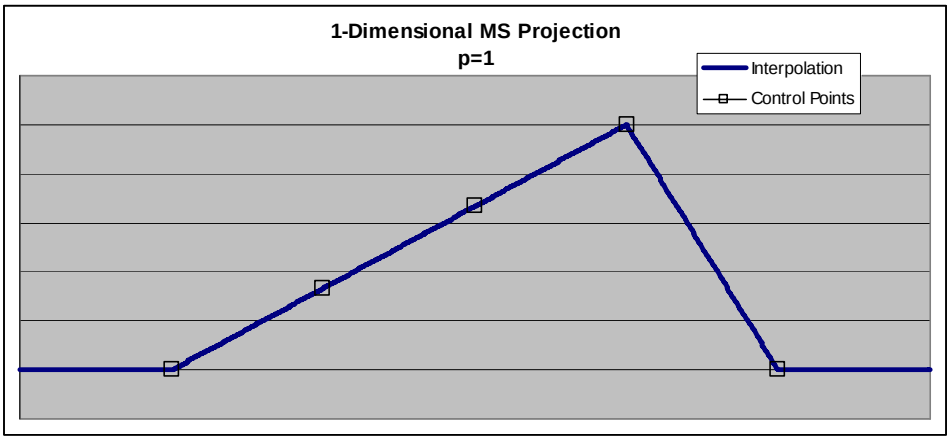


Figure 4.11. Microsphere Projection, p=1 interpolation of simple smooth data set.

Microsphere Projection with p=1. Setting p=1, Microsphere Projection becomes the 1-dimensional equivalent of piecewise linear interpolation. This model satisfies all of our main criteria; however its non-differentiability may cause problems. See Figure 4.11.

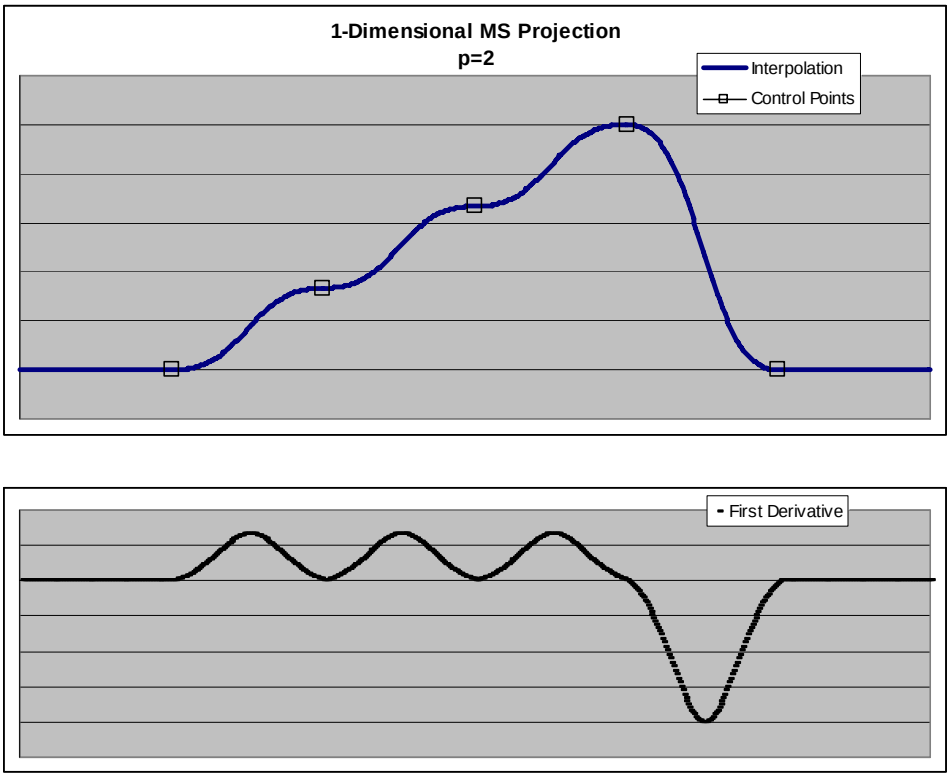


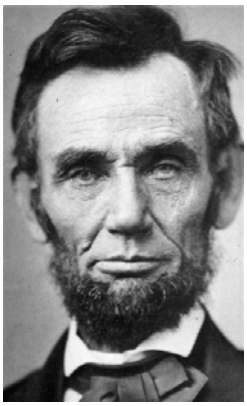
Figure 4.12. Microsphere Projection, p=2 interpolation of simple smooth data set.

Microsphere Projection with p=2. Setting p=2 provides the differentiability that is lacking with p=1. The graph could be described as a smoothed version of nearest neighbor interpolation. This method satisfies our criteria of differentiability and preservation of monotonic behavior. See Figure 4.12.

4.1.3 One-Dimensional Interpolation: Analysis

One-dimensional tests were conducted using the grayscale images in Figure 4.13. Each image was dissected row-by-row, a single row of pixels representing a set of 1-dimensional data. See Figure 4.14. From a row of pixels, a random collection of pixels were sampled on a percentage basis. These 'known' values formed the basis of the

interpolation functions. At this point, a single pixel was randomly selected and the interpolated value was compared against the actual value. This process was repeated 1000 times for each pixel row in each image for a total test size of 3642000.



(w:310 h:512)

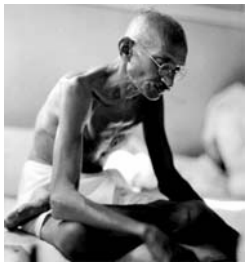
(w:377 h:450)

(w:500 h:316)

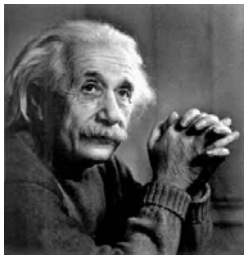
(w:599 h:480)

(w:600 h:465)

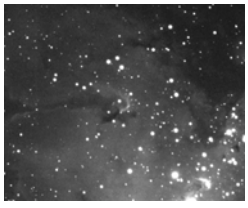
(w:302 h:307)



(w:187 h:200)



(w:299 h:312)



(w:401 h:327)

(w:216 h:273)

Figure 4.13. Grayscale images used in 1-Dimensional and 2-Dimensional testing.

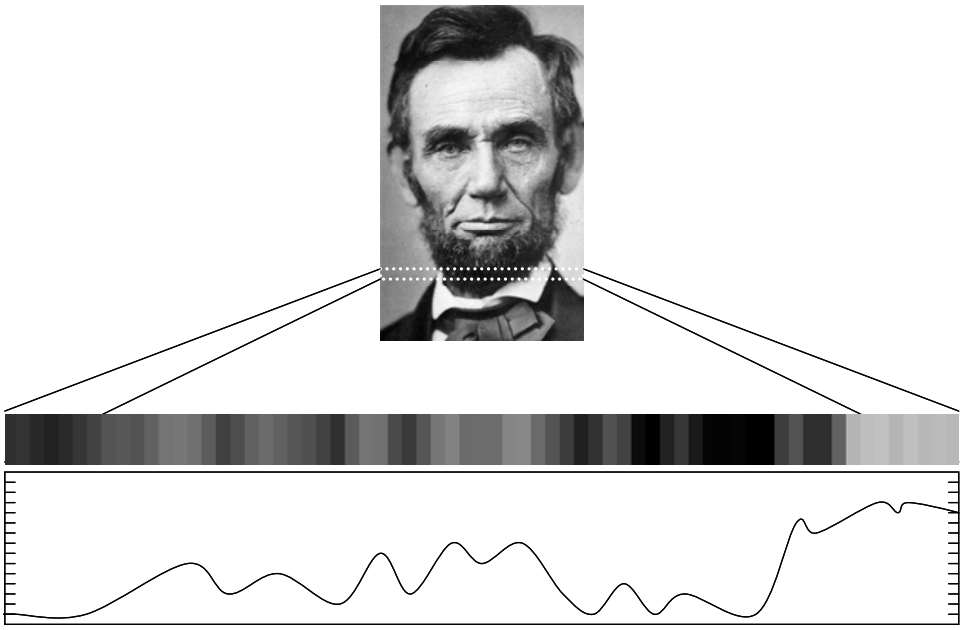


Figure 4.14. Example of how 1-dimensional testing data was extracted from existing grayscale images.

The 1-dimensional interpolation analysis was conducted in two phases. The first phase used strict interpolation, meaning that the values tested were strictly limited to the range of the randomly sampled pixels. The second phase did not restrict the testing to the range of the sampled pixels. Instead it tested the accuracy of the interpolation method along the entire length of the sample space. See Figure 4.15 for details. All error values were calculated by using relative Root Mean Squared error as written in Formula 3.1

$$\varepsilon_{RMS} = \frac{\sqrt{\sum_{i=1}^{N}(k_i - v_i)^2}}{(v_{\max} - v_{\min})} \qquad 3.1$$

Definition of relative Root-Mean-Squared (RMS) error. k is the interpolation value, v is the actual value, N is the number of interpolation values that have been gathered.

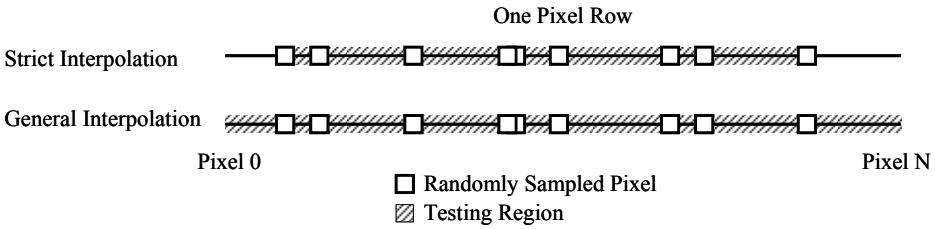


Figure 4.15. Differences between one-dimensional testing sets. Strict Interpolation tests strictly on the inclusive set of randomly sampled pixels. General Interpolation includes a small amount of extrapolation on the ends of the set.

Table 4.1. Relative RMS error of various 1-dimensional interpolation methods using Strict Interpolation.

	Percentage of Data "Known" to the Algorithm					
Method of Interpolation	**2.5%**	**5%**	**10%**	**25%**	**50%**	**90%**
Microsphere Projection, p=2	0.1731	0.1429	0.1135	0.0828	0.0651	0.0541
Microsphere Projection, p=1 (piecewise linear)	0.1688	0.1392	0.1105	0.0806	0.0635	0.0531
Piecewise Cubic Spline	0.1744	0.1438	0.1137	0.0825	0.0644	0.0533
Shepard's Method, p=2 (inverse distance)	0.1698	0.1410	0.1136	0.0848	0.0689	0.0601
Nearest Neighbor	0.1906	0.1592	0.1281	0.0951	0.0769	0.0668
Average Value	0.2186	0.2155	0.2134	0.2120	0.2117	0.2111
N (number of samples)	3.642e6	3.642e6	3.642e6	1.821e6	7.284e5	3.642e5

Table 4.2. Relative RMS error of various 1-dimensional interpolation methods using General Interpolation.

	Percentage of Data "Known" to the Algorithm					
Method of Interpolation	**2.5%**	**5%**	**10%**	**25%**	**50%**	**90%**
Microsphere Projection, p=2	0.1913	0.1503	0.1167	0.0838	0.0660	0.0545
Microsphere Projection, p=1 (piecewise linear)	0.1882	0.1472	0.1139	0.0817	0.0645	0.0535
Piecewise Cubic Spline	3733	696.1	70.33	2.830	0.3406	0.0584
Shepard's Method, p=2 (inverse distance)	0.1827	0.1464	0.1161	0.0857	0.0697	0.0606
Nearest Neighbor	0.2038	0.1640	0.1299	0.0957	0.0777	0.0675
Average Value	0.2240	0.2175	0.2142	0.2124	0.2118	0.2114
N (number of samples)	3.642e6	3.642e6	3.642e6	1.821e6	7.284e5	3.642e5

The data presented in Tables 4.1 and 4.2 were gathered in order to present a more useful picture of how accurate Microsphere Projection is compared to other one-dimensional interpolation techniques.

Included in the list of tested techniques is 'Average Value'; this method interpolates any value as being the average of all of the values of the sample points. It is not a serious interpolation technique; however it is useful to form a baseline opinion of performance.

Based on the data in Table 4.1, Microsphere Projection is able to perform on par with Piecewise Cubic Splines. One caveat is that the data these tests are based on is non-functional, meaning that the data points were not derived from any known function; they were sampled from photographs. If the sample points are known to be functionally-based and the underlying function fairly smooth (or a smooth grey-scale picture), it is expected that the Piecewise Cubic Spline would be able to interpolate with greater precision.

Because of Piecewise Cubic Spline's simplicity and continuous second-differentiability, it is tempting to extend the spline functions into the realms of extrapolation. The data presented in Table 4.2 demonstrates that this is wholly undesirable. The extremely poor results generated by piecewise cubic splines in Table 4.2 can be justified by understanding that in cases where only a small number of points are sampled, there exists the chance of running into a situation like that depicted in Figure 4.16. Since the test is run 3.6 million times to generate each of the numbers in table 4.2, situations such as Figure 4.16 are bound to happen, and when they do, they skew the root-mean-squared error significantly. If a differentiable function is needed that can extrapolate as well as interpolate, Microsphere Projection works well in these 1-dimensional test cases.

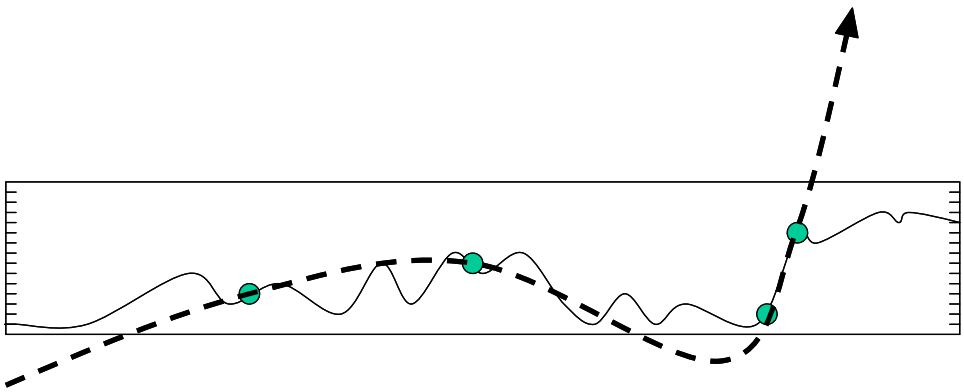


Figure 4.16. Depiction of problem when using cubic splines to perform even small amounts of extrapolation.

4.2 Two-Dimensional Interpolation

4.2.1 Random Control Point Locations – Case Study

For this experiment, a simple 100x100 pixel image was chosen to be interpolated. From the image, 50 random sample points were chosen (0.5% of original image). Using the values of the sample points, various interpolation methods were selected. The results of these interpolation methods are displayed in Figures 4.17-4.23.

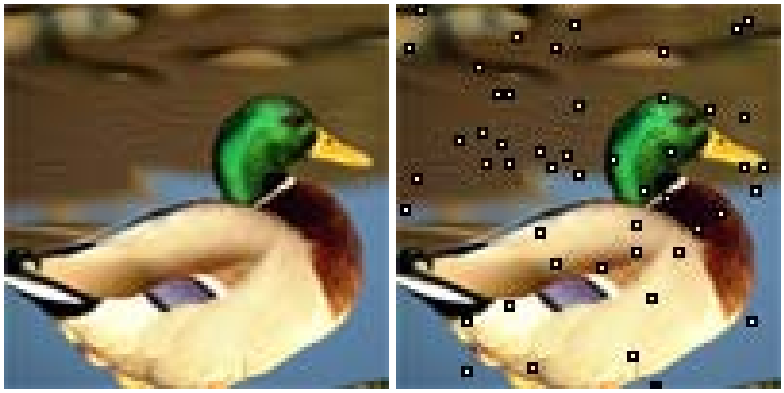


Figure 4.17. Study of random control point locations: original image with and without sample points highlighted.

This image was chosen because it provides:

- An obvious shape we wish to see reproduced by an interpolation function.
- Distinct and varying brightness in different areas of the picture.
- An interesting variation in color (depending on how this document was printed, it may be in color or not).

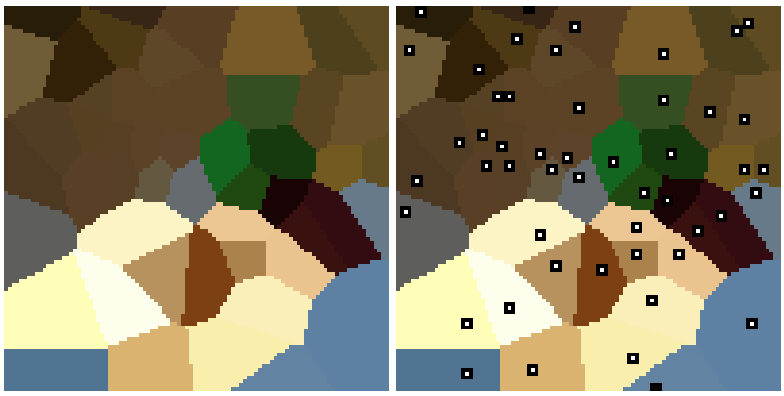


Figure 4.18. Study of random control point locations: interpolation using Nearest Neighbor.

Nearest Neighbor interpolation is not very accurate, however while viewing the following series of interpolations it should be noted that the equivalent of this interpolation is generated by both Shepard's Method and Microsphere Projection as $p \rightarrow \infty$. See Figure 4.18.

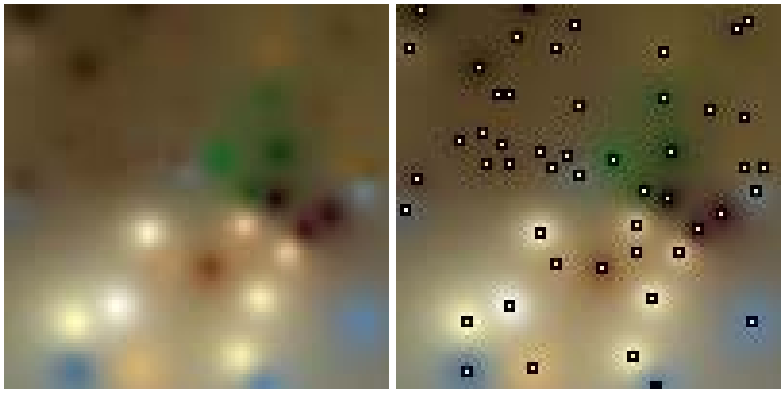


Figure 4.19. Study of random control point locations: interpolation using Shepard's Method (inverse distance weighting), p=2.

It turns out in this example that the value of p=2 is perhaps less than ideal. However, since the general case of Shepard's Method suggests p=2 and a typical user will have little information on how to change the value to improve their results, this should be accepted as a typical interpolation case. See Figure 4.19.

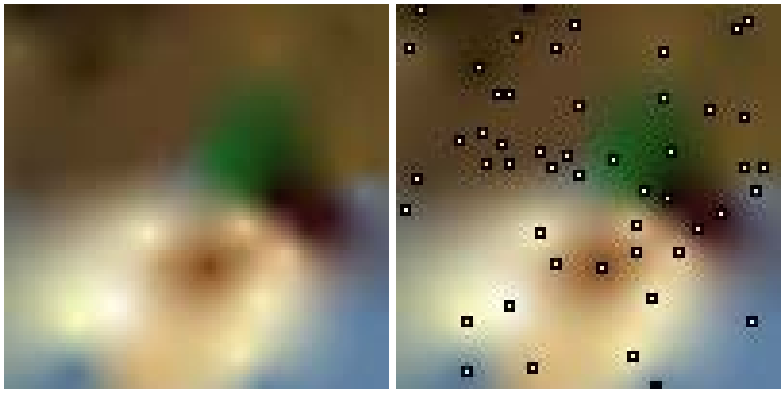


Figure 4.20. Study of random control point locations: interpolation using Microsphere Projection, p=1.

Microsphere Projection with p=1 provides a reasonable interpolation. Setting the p-value very low allows points that are farther away to influence the interpolation values to a larger degree. The resultant image is one which looks a little bit 'smeared'. The non-differentiability when p=1 is also noticeable in that the sample points are extremely pronounced in the interpolation image. See Figure 4.20.

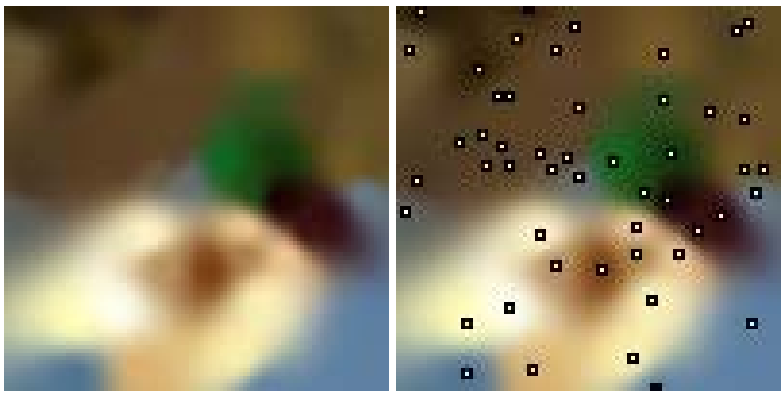


Figure 4.21. Study of random control point locations: interpolation using Microsphere Projection, p=2.

Using Microsphere Projection with p=2 provides a very appealing result. The 'smearing' effect generated when p=1 is greatly reduced with p=2. With the loss of the smearing, we are able to distinguish some shapes in the object a little more clearly. Arguably, this seems to produce the best interpolation of any of the algorithms. See Figure 4.21.

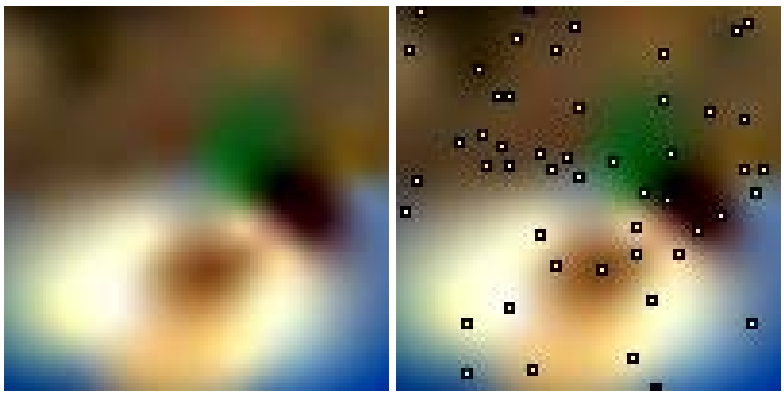


Figure 4.22. Study of random control point locations: Interpolation using Thin-Plate Spline method.

Thin-Plate Spline interpolation is a fairly common technique for interpolating non-uniform two-dimensional data. This method produces a good interpolation, however certain aspects of the interpolation are lacking. Besides some extreme interpolated values (which will be addressed in the next few paragraphs), the only problem with this interpolation is its inherent 'roundness'. Because it is based on a radial function, the interpolation around the data points is extremely rounded, and does not necessarily connote the desired shapes of the original image.

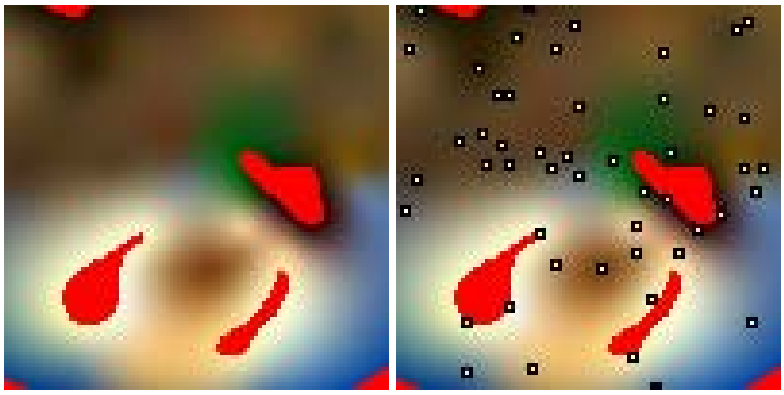


Figure 4.23. Study of random control point locations: Interpolation using Thin-Plate Spline method. The areas which generated non-real values (either the R, G, or B component of the pixel interpolated >255 or <0) are highlighted in red.

One other consideration should be made when choosing to use a radial basis function (RBF) interpolation such as Thin-Plate Spline: That of unintended interpolation to non-real values. Unfortunately the high degree of smoothness that these functions enjoy comes at the cost of the resulting interpolation being beyond the range of the

original sampled values (see Figure 4.23). In some cases this is a reasonable result, however in many cases (including those involving image interpolation), they can cause issues.

The simplest mechanism to avoid non-real values is to truncate the values when they surpass the ranges deemed acceptable. This mechanism was employed in Figure 4.22. Though it works, it defeats the purpose of using these interpolation techniques in the first place – their high smoothness and differentiability. Truncating the values renders the interpolation C^0 (non-differentiable).

4.2.2 Controlled Selection of Sample Points Located in Area of Interest – Case Study

This experiment differs slightly from the previous in that the sample points selected are not wholly random. The image and sample points were chosen to represent the concept of 'area of interest'. In many cases, the end user will desire a great degree of information regarding only a certain area of the full interpolation space, and thus will take a large number of sample points in that area. It is the responsibility of the interpolation function to handle this case without producing aberrant values in the areas where there are few or no sample points.

A grayscale elevation map of the New York coastline was chosen (w:401, h:327). 150 random pixels were chosen (0.11% of the original image) as sample points with the condition that they must be within 10 pixels of the shoreline. The restriction to within 10 pixels of the shoreline represents our 'area of interest' – anyone who would be interested in elevation data would not be surveying the elevation of the Atlantic Ocean. The results of interpolating this data with a variety of algorithms is displayed in Figures 4.24-4.29.

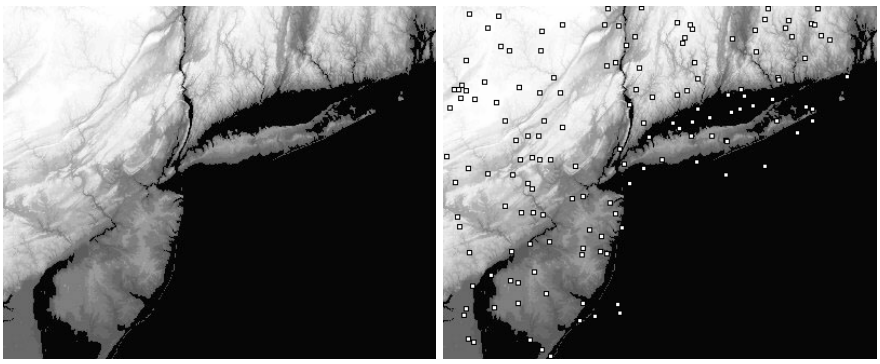


Figure 4.24. Study of controlled selection of control point locations: Original Image with and without sample points highlighted.

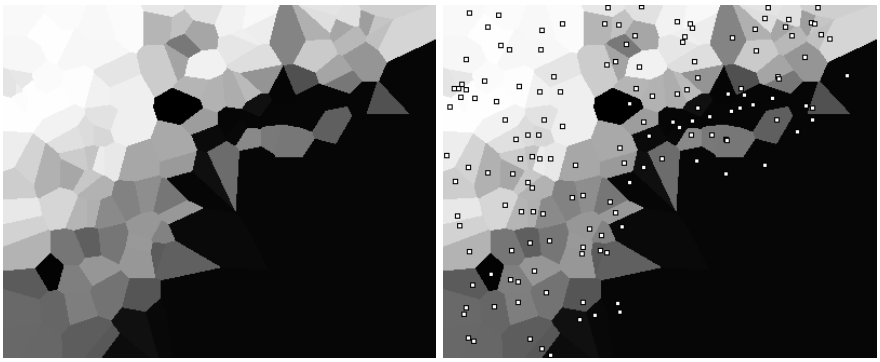


Figure 4.25. Study of controlled selection of control point locations: interpolation using Nearest Neighbor.

Nearest Neighbor is included because the equivalent of this interpolation is generated by both Shepard's Method and Microsphere Projection as $p \rightarrow \infty$. See Figure 4.25.

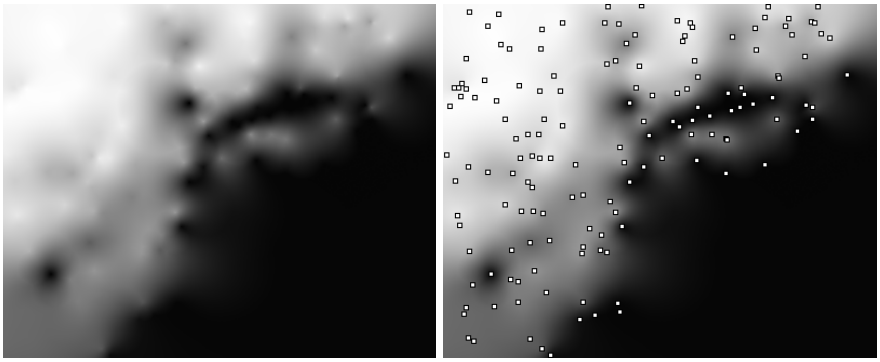


Figure 4.26. Study of restricted control point locations: interpolation using Microsphere Projection, p=1.

Using Microsphere Projection with p=1, we are able to recreate the coastline fairly well. The only visual miscues generated by this interpolation are at the locations of the sample points where the non-differentiability (non-smoothness) is noticeable. See Figure 4.26.

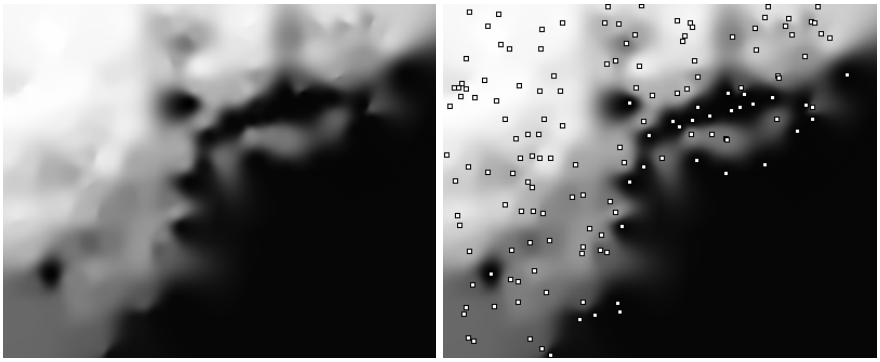


Figure 4.27. Study of restricted control point locations: interpolation using Microsphere Projection, p=2.

Using a propagation of influence power of 2, we are able to create an interpolation that is smoother than p=1 and generates less ‘smearing’ around the edges of the coastline. See Figure 4.27.

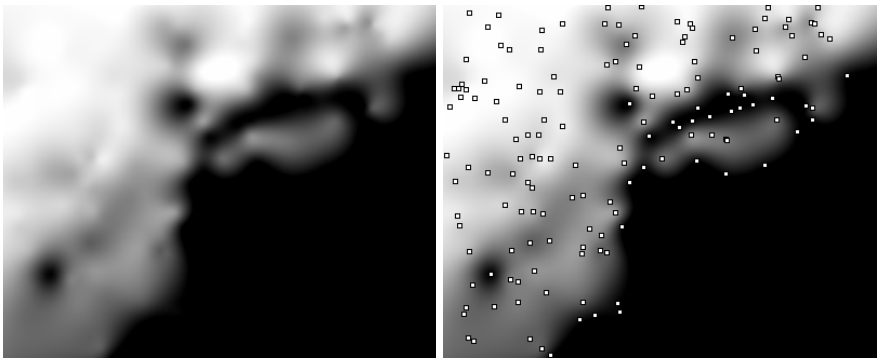


Figure 4.28. Study of restricted control point locations: interpolation using Thin-Plate Spline method.

Thin-Plate Spline produces a very smooth interpolation in comparison to the others shown. Visually, we can see the cost of using a Radial Basis Function (RBF) interpolation when looking at the interpolation of the long thin island in the original image(see Figure 4.28). Where Micosphere Interpolation was able to preserve the narrowness of the island, RBF-based interpolations such as TPS are more apt to produce a more ‘rounded’ look. This distorts the image slightly, making the island seem wider than it is in the original. Also, though smooth, TPS’s smoothness (high differentiability) comes at a high cost, as can be seen in Figure 4.29.

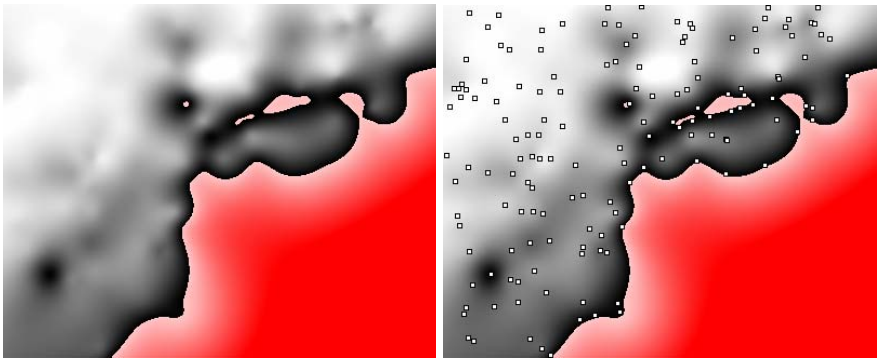


Figure 4.29. Study of restricted control point locations: interpolation using Thin-Plate Spline method. The areas which generated non-real values (the pixel was interpolated as >255 or <0) are highlighted in shades of red.

4.2.3 Two-Dimensional Interpolation: Analysis

Two-dimensional tests were conducted using the grayscale images displayed in Figure 4.13. For each image, a random set of sample pixels were chosen. Following this, 100 random pixels were selected and compared against their interpolated values. This procedure was performed on each picture 100 times for a total test size of 100000 between all ten pictures.

The 2-dimensional interpolation analysis was conducted in two phases. The first phase used strict interpolation, meaning that the values tested were strictly limited to the convex hull formed by the sample pixels. The second phase did not restrict the testing to the convex hull; instead, pixels from the entire image were tested for accuracy. See Figure 4.30 for details.

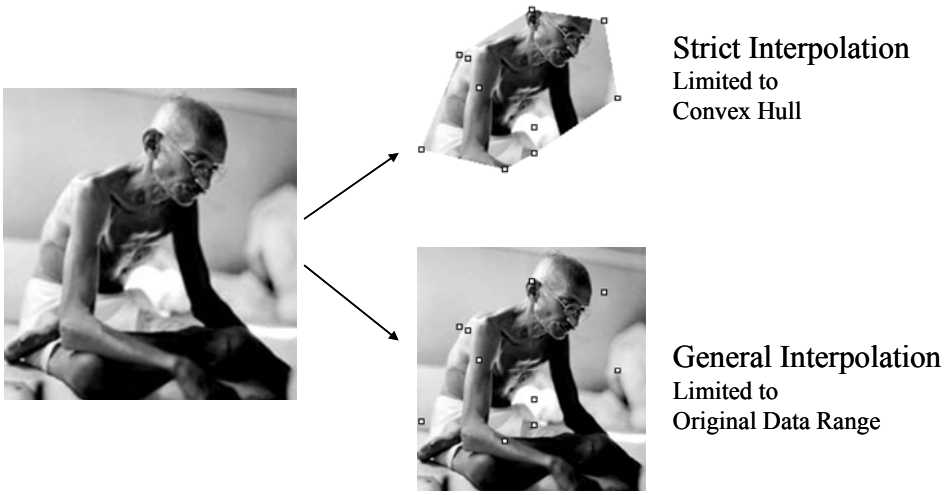


Figure 4.30. Differences between two-dimensional testing sets. Strict Interpolation tests strictly within the convex hull of sampled pixels. General Interpolation includes all pixels from the original data range.

Table 4.3. Relative RMS error of various two-dimensional interpolation methods using Strict Interpolation.

	Number of Points Sampled					
Method of Interpolation	**10**	**20**	**50**	**100**	**500**	**1000**
Microsphere Projection, p=2	0.251	0.231	0.201	0.178	0.133	0.116
Microsphere Projection, p=1 (2D version of piecewise linear)	0.242	0.222	0.193	0.172	0.128	0.112
Thin-Plate Spline	0.269	0.248	0.214	0.189	0.138	0.121
Shepard's Method, p=2 (inverse distance)	0.241	0.222	0.197	0.180	0.150	0.140
Nearest Neighbor	0.289	0.268	0.234	0.210	0.159	0.140
Average Value	0.265	0.259	0.255	0.251	0.250	0.249
N (number of samples)	1.0e5	1.0e5	1.0e5	1.0e5	1.0e5	1.0e5

Table 4.4. Relative RMS error of various two-dimensional interpolation methods using General Interpolation (includes areas outside convex hull).

	Number of Points Sampled					
Method of Interpolation	**10**	**20**	**50**	**100**	**500**	**1000**
Microsphere Projection, p=2	0.258	0.234	0.204	0.180	0.133	0.116
Microsphere Projection, p=1 (2D version of piecewise linear)	0.247	0.225	0.196	0.173	0.129	0.112
Thin-Plate Spline	0.320	0.273	0.226	0.195	0.140	0.122
Shepard's Method, p=2 (inverse distance)	0.242	0.221	0.197	0.181	0.151	0.139
Nearest Neighbor	0.292	0.268	0.237	0.211	0.160	0.141
Average Value	0.260	0.254	0.250	0.250	0.249	0.248
N (number of samples)	1.0e5	1.0e5	1.0e5	1.0e5	1.0e5	1.0e5
Convex Hull Coverage	43.6%	62.3%	80.0%	88.2%	96.6%	98.1%

Looking at the data gathered from 'strict interpolation', we can see that Microsphere Projection outperforms TPS for our test cases. The difference in performance is not large, however it is more significant and consistently larger than our error margin.

In the 'general interpolation' tests we see a slightly more profound difference between the values. While Microsphere Projection is slightly penalized by having some testing points outside the convex hull of the sample points, the penalty to TPS is far more significant. As the number of sample points grows larger, the convex hull approaches the size of the entire sample space, and thus the values in the '1000' column of both charts are very similar.

One interesting and unanticipated result of these tests is the general dominance of Microsphere Projection with p=1. Though having the downside of non-differentiability,

using p=1 seems to provide a clear advantage over p=2 as far as accuracy is concerned. This can be justified by considering that the data we are modeling is non-functional, and that p=1 is simply a multidimensional version of piecewise linear interpolation.

In all, Microsphere Projection displays a very strong performance when compared to the standard two-dimensional interpolation technique of Thin-Plate Spline. The only extra considerations might be that the data used for evaluation was not functionally based, and was limited to a range (0-255). Now, this should not be a problem for a robust interpolation technique, however TPS should not be written off entirely. TPS has been shown to perform admirably on smoother functionally based data sets. Therefore, if the underlying data were functionally based and fairly smooth, it might make more sense to use TPS rather than Microsphere Projection.

4.3 Three-Dimensional Interpolation

To experiment with Microsphere Projection in three dimensions, we chose to interpolate based on data gathered from soil pollution readings at various locations and depths throughout an industrial complex. The data gathered is unfortunately of a form different from our previous two dimensionality tests in that it is non-uniform. In the previous two sections we have been able to test against known uniform distributions of one and two dimensional data. The fact that the soil pollution readings are in non-uniform locations will force us to modify the testing mechanism slightly.

The data was gathered by drilling several bore wholes into the ground and measuring pollution levels in parts-per-million (PPM). Thus, the data is organized in linear sets of samples of varying depths, each originating from the surface (top) of the

data. The majority of the pollution sampled was at ground level or slightly below. Samples taken at lower depths nearly all returned 0 PPM. Based on this understanding, we should expect a correct interpolation to display 'hotspots' of pollution near the top, and the lowest parts of the graph to be at or extremely close to 0 PPM. Figures 4.31-4.36 display our results of interpolating this data set.

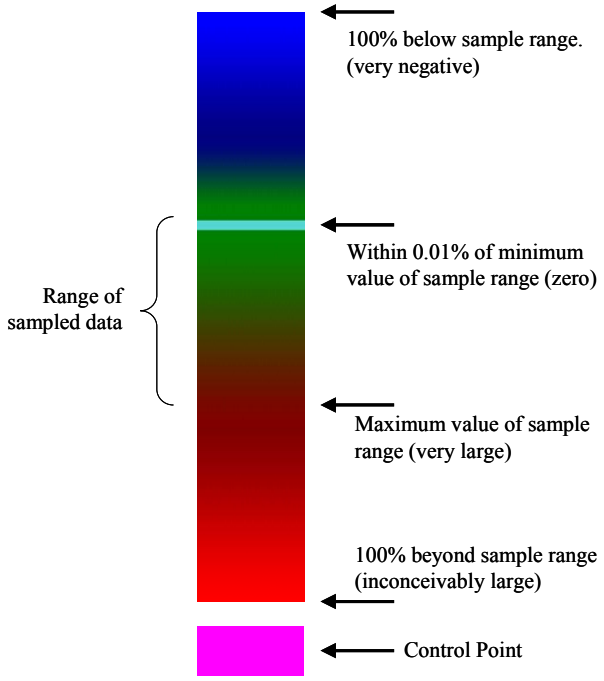


Figure 4.31. Legend for use in figures 4.32-4.36.

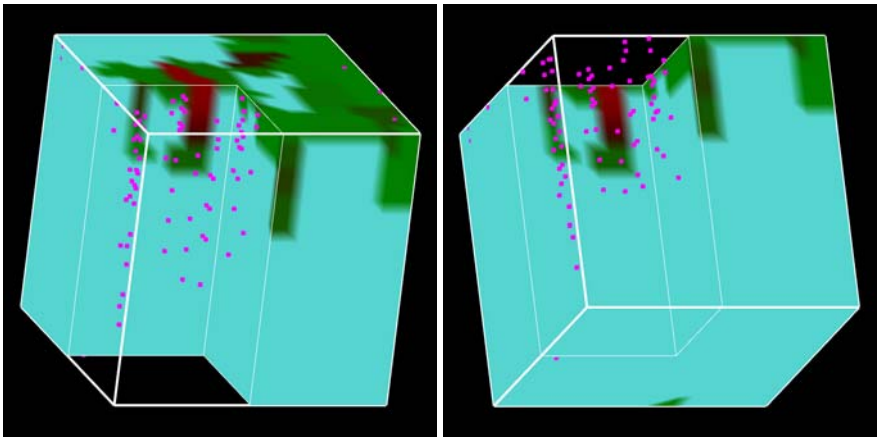


Figure 4.32. Front-top and front-bottom views of Nearest Neighbor interpolation, with one quadrant cut-away.

Nearest Neighbor interpolation works mildly well with this data set, however due to its inability to take distance to sample points into account and its non-differentiability, it can not bet trusted to generate a reasonable interpolation. See Figure 4.32.

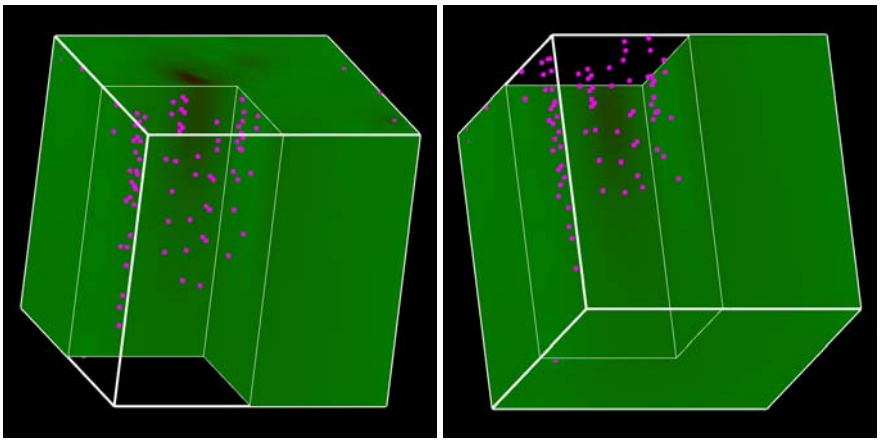


Figure 4.33. Front-top and front-bottom views of Shepard's Method p=2 interpolation, with one quadrant cut-away.

Using inverse-distance (Shepard's Method) for interpolating this data set, we can see reasonable results near the center of mass of the sample points; however certain aspects of the interpolation are blatantly incorrect. The most notable problem with using Shepard's method in this situation is that as we move farther away from the sample points (move farther down into the ground), the interpolated value approaches the average value of all of our samples (non-zero). This model would seem to imply that no matter how far down we test, there will always be an 'ambient' pollution. Our empirical data and our basic intuition do not support this because our deepest sample points all registered 0 PPM. This behavior is a major drawback of using Shepard's method. See Figure 4.33.

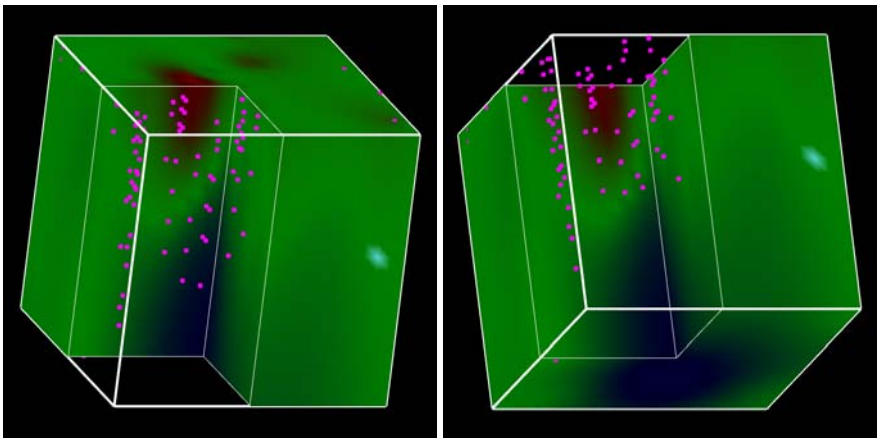


Figure 4.34. Front-top and front-bottom views of Multiquadric interpolation, with one quadrant cut-away.

Using Multiquadric interpolation, we are able to generate reasonable results for interpolations contained within the convex hull, though interpolating much beyond the

convex hull yields unrealistic results. Most notable of the unrealistic results are the large negative values displayed in figure 4.34 as a blue section at the bottom of the bounding box. This result is incorrect for two reasons, first it is non-zero, and second, it is not physically possible. Negative PPM does not make any physical sense. If only for these two reasons, this interpolation method is a complete failure on this data set.

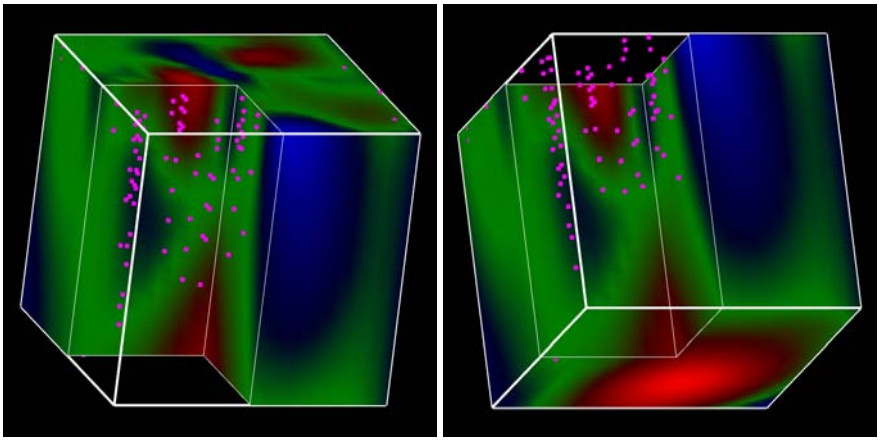


Figure 4.35. Front-top and front-bottom views of Volume Spline interpolation, with one quadrant cut-away.

Using volume spline interpolation yields a whole slew of incorrect and unrealistic results when applied to this data set. Though volume spline interpolation may produce reasonable results within the convex hull, the bounding box encompasses far greater area, and requires a good amount of extrapolation. Volume spline interpolation fails miserably at this extrapolation as we can see demonstrated in the huge areas of negative PPM (not physically possible), and a huge area of pollution at depths beyond where sensors had recorded none. See Figure 4.35.

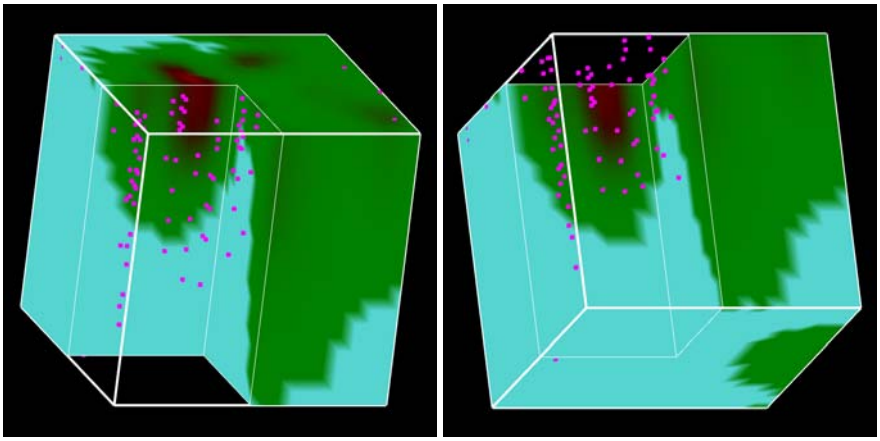


Figure 4.36. Front-top and front-bottom views of Microsphere Projection p=2, with one quadrant cut-away.

Visually, Microsphere Projection almost perfectly reproduces the results which we would expect from this data: the 'hotspots' near the surface seem to be modeled well, the interpolation approaches 0 PPM as depth increases, and there are no negative interpolated values. The only potential problem with this data set is almost a complete lack of data points in the front-right. This lack of data points makes interpolation particularly difficult in this area of the graph, and Microsphere projection interpolates some of the very deep points in this area to have very small (non-zero) pollution as can be seen represented as a large green patch in figure 4.36. This is undesirable; however the behavior in this region of the graph is far better than any of the other interpolation methods.

Table 4.5. Small sample of the soil pollution data. X-Y ground coordinates are listed first, Z depth coordinate listed third, and PPM of pollution is listed fourth.

8200.590551	7234.219160	-0.050000	790.000000
8188.877953	7167.158793	-0.050000	8190.000000
8070.439633	7322.736220	-1.000000	2.000000
8070.439633	7322.736220	-5.000000	2.000000
8041.141732	7277.723097	-1.000000	3520.000000
8041.141732	7277.723097	-5.000000	2.000000
8041.141732	7277.723097	-20.000000	2.000000
8030.118110	7271.227034	-5.000000	42400.000000
8030.118110	7271.227034	-10.000000	38500.000000
8030.118110	7271.227034	-15.000000	29300.000000
8030.118110	7271.227034	-20.000000	18300.000000
8030.118110	7271.227034	-25.000000	5500.000000
8030.118110	7271.227034	-30.000000	1.000000
8106.725722	7291.797900	-2.000000	3490.000000
8106.725722	7291.797900	-5.000000	2.000000
7982.939633	7274.409449	-5.000000	1.000000
7982.939633	7274.409449	-1.000000	2030.000000
7943.077428	7192.814961	-5.000000	0.000000
7943.077428	7192.814961	-30.000000	0.000000
7943.077428	7192.814961	-1.000000	0.000000
7943.077428	7192.814961	-10.000000	0.000000
7943.077428	7192.814961	-20.000000	0.000000
7943.077428	7192.814961	-45.000000	0.000000
7966.141732	7237.762467	-10.000000	13500.000000
7966.141732	7237.762467	-15.000000	4380.000000
7966.141732	7237.762467	-20.000000	943.000000
7966.141732	7237.762467	-30.000000	1.000000
7966.141732	7237.762467	-35.000000	0.000000

To test our algorithms against this data set, it was decided to use the single-point-removal test for modeling error. This test is conducted by removing a single point from the data set, and then comparing the interpolated value to the real value for the removed point. In our case, this process was conducted for each of the 145 points in the original

dataset (a small section of this data set is shown in table 4.5). The results of this procedure are located in Table 4.6.

Table 4.6. Relative RMS error of various three-dimensional interpolation methods using single-point-removal testing when applied to pollution data.

Method of Interpolation	Relative RMS Error
Microsphere Projection, p=2	0.080
Microsphere Projection, p=1 (piecewise linear)	0.081
Volume Spline	0.093
Multiquadric, r=1	0.077
Shepard's Method, p=2 (inverse distance)	0.100
Nearest Neighbor	0.110
Average Value	0.168

Because of the low size of N (145) and the non-uniformity of the source data, the results of the error rates presented in Table 4.6 can be debated, however it is clear that Microsphere Projection is at the very least able to provide similar error rates to that of Volume Spline and Multiquadric interpolation.

4.4 Hyper-Dimensional Interpolation

Interpolating data in higher dimensions poses certain difficulties; the chief difficulty to be addressed in this section is that of 'infinite corners'. We have seen in Tables 4.2, and 4.4 that spline interpolations falter when they are asked to extrapolate (extrapolation being defined as guessing at values beyond the convex hull of sample points). Thankfully, the bounding box around the sample points is very similar to the convex hull in lower dimensions; in fact when d=1, the bounding box and convex hulls

are the same. However, in higher dimensions things can become strange. See Figure 4.37 for an illustration of the difference between the convex hull and the bounding box.

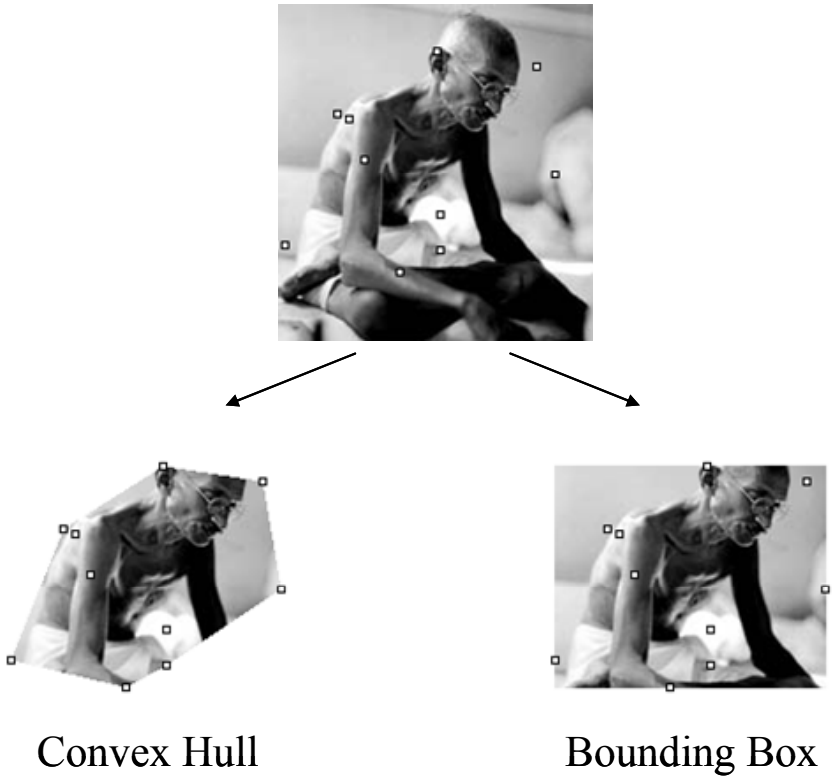


Figure 4.37. Illustration of 'Convex Hull' and 'Bounding Box'.

Most applications of non-uniform interpolation involve mapping a set of scattered data points on to a regular grid. This grid often takes the form of the bounding box surrounding the sample points such as in figures 4.32-4.36. Our problem arises in that in higher dimensions, the bounding box is often a great deal larger than the convex hull. In fact, the convex hull may represent only a very small fraction of the bounding box in very high dimensions. See Formula 4.1 for a simple definition of the relationship. The extra

space beyond the convex hull means that in order to fill out the regular grid in higher dimensions, we must not only be able to interpolate, but extrapolate as well.

$$as \quad d \to \infty, \quad \frac{\left\| ch^d \right\|}{\left\| bb^d \right\|} \to 0 \qquad 4.1$$

bb^d is the d-dimensional bounding box
ch^d is the d-dimensional convex hull
$\| x \|$ is the volume of space enclosed by x

At present, the RBF-based spline interpolations are inadequate at handling extrapolation beyond the convex hull as can be seen by the differences in Figures 4.34 and 4.35 and Tables 4.3 and 4.4. Looking at these tables, the Microsphere Projection function seems hardly disturbed when handling data outside the convex hull. Based on this information, it seems that Microsphere Projection is well-suited to high-dimensionality interpolation, whereas many of the other most popular interpolation techniques including RBF-based techniques are not.

CHAPTER V

CONCLUSIONS AND FUTURE WORK

In this thesis we have presented a new algorithm for interpolating sparse data: Microsphere Projection. This algorithm is based on the physical structure of an infinitesimally small sphere. Using this structure we are able to interpolate based on the 'illumination' of nearby sample points.

Our analysis has shown that Microsphere Projection is a viable interpolation technique, and in the test cases conducted in this thesis surpasses the abilities of some existing methods. In one dimension, MS interpolation with p=2 proves to be almost as accurate as piecewise cubic spline interpolation, with p=1 (the non-differentiable case), the accuracy surpasses cubic splines. See Table 4.2. In two dimensions, the accuracy of MS interpolation easily outperforms thin-plate spline interpolation; and in three dimensions its performance is at least on par with existing techniques. See Tables 4.3, 4.4, and 4.6. In hyper dimensions it is expected that MS interpolation will be even more useful due to its stable extrapolation properties.

Future work should include more extensive three-dimensional testing, along with comparisons between a wider range of alternative interpolation functions including Volume MMN Networks [5] and Bézier Surfaces [2]. All tests in 1D and 2D were performed based on a set of grayscale images; it would be beneficial to test the algorithm

against more varieties of one and two-dimensional data. Hyper-dimensional testing would be interesting, though the availability of original-source hyper-dimensional data is fairly limited.

Microsphere projection is able to provide a new and innovative interpolation method which is differentiable, preserves monotonic behavior, generates very low error rates, and is applicable in any dimensionality. These characteristics along with others discussed in this paper make Microsphere Projection a unique and robust tool with many applications.

REFERENCES

[1] F. L.Bookstein, “Principal Warps: Thin Plate Splines and the Decomposition of Deformations,” IEEE Trans. Pattern Anal. Mach. Intell. 11, 567-585, 1989.

[2] Gerald Farin. “Curves and Surfaces for CAGD, 5th ed.,” Morgan-Kaufmann Press, 2002.

[3] R. Franke and G. Neilson, “Scattered Data Interpolation and Applications: A Tutorial and Survey,” in Geometric Modelling: Methods and Their Applications, H. Hagen and D. Roller, eds., Springer, Berlin. 1990. pp. 131-160.

[4] Roger A. Horn and Charles R. Johnson, “Topics in Matrix Analysis,” Cambridge University Press, 1991. (See Section 6.1)

[5] Gregory M. Nielson, “Scattered Data Modeling,” IEEE Computer Graphics and Applications, pp. 60-70, 1993.

[6] Donald Shepard, “Two-Dimensional Interpolation Function for Irregularly-spaced Data,” Proc ACM National Conference: 517-524, 1968.

[7] Eric W. Weisstein, “Bicubic Spline,” From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/BicubicSpline.html

[8] Yincai Xiao, et al, “The Challenges of Visualizing and Modeling Environmental Data,” IEEE Visualization 96 Conference Proceeding. San Francisco, California, pp. 413-416, 1996.

APPENDIX

CRITICAL SOURCE CODE

```
/*********************************************************************
main.cpp
*********************************************************************/

#include <iostream>
#include <fstream>

#include "all_sparse_interpolators.h"
#include "bitmap.h"
#include "testfunctions.h"
#include "random.h"
#include "imageManip.h"
#include "opengl_code.h"

void Test1D ();
void Test2D ();
void Test3D ();
void MakeImage ();
void Test();

int main (int argc, char **argv)
{
    std::cout << "working...";

    //Test1D(); // perform 1-d testing and output error rate results
    //Test2D(); // perform 2-d testing and output error rate results
    //Test3D(); // perform 3-d testing and output error rate results
    //MakeImage(); // create 2-d images for viewing
    Test3D_2(argc, argv); // create 3-d images for viewing using openGL

    std::cout << "\n\npress any character & enter to exit.\n";
    char hold = 0;
    std::cin >> hold;
}
```

```
void Test()
{
    bitmapHeader img(100,100);

    point3D pts[16] =
    {
        point3D(0,0,0),
        point3D(33,0,0),
        point3D(67,0,0),
        point3D(100,0,0),
        point3D(0,33,0),
        point3D(33,33,0),
        point3D(67,33,0),
        point3D(100,33,0),
        point3D(0,67,0),
        point3D(33,67,0),
        point3D(67,67,0),
        point3D(100,67,0),
        point3D(0,100,0),
        point3D(33,100,0),
        point3D(67,100,0),
        point3D(100,100,0)
    };

    double valsR[16] =
    {
        255,0,0,255,
        0,255,255,0,
        0,255,255,0,
        255,0,0,255
    };

    double valsG[16] =
    {
        0,255,0,0,
        0,255,255,255,
        255,255,255,0,
        0,0,255,0
    };

    double valsB[16] =
    {
        0,0,255,0,
        255,255,255,0,
        0,255,255,255,
```

```
        0,255,0,0
    };

    spherical_interpolator iR, iG, iB;
    iR.RegisterSparseData(pts, valsR, 16);
    iG.RegisterSparseData(pts, valsG, 16);
    iB.RegisterSparseData(pts, valsB, 16);

    //for (int i=0;i<100;i++)
    //    for (int j=0;j<100;j++)
    //    {
    //        int r = iR.InterpolatePoint(point3D(i,j,0),2,true);
    //        int g = iG.InterpolatePoint(point3D(i,j,0),2,true);
    //        int b = iB.InterpolatePoint(point3D(i,j,0),2,true);
    //        img.setVals(i,j,r,g,b);
    //    }

    //img.SaveFile("output.bmp");

    bitmapHeader img2(4,4);
    img2.setVals(0,0,255,0,0);
    img2.setVals(1,0,0,255,0);
    img2.setVals(2,0,0,0,255);
    img2.setVals(3,0,255,0,0);
    img2.setVals(0,1,0,0,255);
    img2.setVals(1,1,255,255,255);
    img2.setVals(2,1,255,255,255);
    img2.setVals(3,1,0,255,0);
    img2.setVals(0,2,0,255,0);
    img2.setVals(1,2,255,255,255);
    img2.setVals(2,2,255,255,255);
    img2.setVals(3,2,0,0,255);
    img2.setVals(0,3,255,0,0);
    img2.setVals(1,3,0,0,255);
    img2.setVals(2,3,0,255,0);
    img2.setVals(3,3,255,0,0);

    bitmapHeader img3(100,100);

    resampleBicubicBSpline(img3, img2);

    img2.SaveFile("output2.bmp");
    img3.SaveFile("output3.bmp");

}
```

```
void MakeImage ()
{

    bitmapHeader img(1,1);
    img.LoadFile("2D_7.bmp");
    //spherical_interpolator interpolator1, interpolator2, interpolator3;
    //nearestneighbor_interpolator interpolator1, interpolator2, interpolator3;
    //inversedistance_interpolator interpolator1, interpolator2, interpolator3;
    thinplatespline_interpolator interpolator1, interpolator2, interpolator3;


    #define KNOWN_COUNT 10 // number or known pixels

    point3D knownPoints[KNOWN_COUNT];
    double knownValuesR[KNOWN_COUNT];
    double knownValuesG[KNOWN_COUNT];
    double knownValuesB[KNOWN_COUNT];

    for (int i=0; i < KNOWN_COUNT; i++)
    {
        int x=0,y=0;
        bool sea;
        do
        {
            x = knownPoints[i].x = rand()%img.width;
            y = knownPoints[i].y = rand()%img.height;
            knownValuesR[i] = img.getVal(x,y,RR);
            knownValuesG[i] = img.getVal(x,y,GG);
            knownValuesB[i] = img.getVal(x,y,BB);

            sea = true;
            for (int xxx = x-15; xxx<x+15; xxx++)
                for (int yyy = y-15;yyy<y+15;yyy++)
                    if (img.getVal(xxx,yyy,RR) > 10) sea = false;

        }
        while (sea);
    }

    interpolator1.RegisterSparseData(knownPoints, knownValuesR, KNOWN_COUNT);
    interpolator2.RegisterSparseData(knownPoints, knownValuesG, KNOWN_COUNT);
    interpolator3.RegisterSparseData(knownPoints, knownValuesB, KNOWN_COUNT);

    img.Whitewash();
```

```
for (int x=0;x<img.width;x++)
{
    cout << "col " << x << "\n";
    for (int y=0;y<img.height;y++)
    {

        point3D pt(x,y,0);
        int r=0,g=0,b=0;

        for (int i=0; i < KNOWN_COUNT; i++)
            if (pt == knownPoints[i])
            {
                //img.setVals(x,y,255,255,255);
                //img.setVals(x-1,y-1,0,0,0);
                //img.setVals(x,y-1,0,0,0);
                //img.setVals(x+1,y-1,0,0,0);
                //img.setVals(x-1,y,0,0,0);
                //img.setVals(x+1,y,0,0,0);
                //img.setVals(x-1,y+1,0,0,0);
                //img.setVals(x,y+1,0,0,0);
                //img.setVals(x+1,y+1,0,0,0);
                //img.setVals(x,y,255,255,255);
                //img.setVals(x-1,y-1,255,255,255);
                //img.setVals(x,y-1,255,255,255);
                //img.setVals(x+1,y-1,255,255,255);
                //img.setVals(x-1,y,255,255,255);
                //img.setVals(x+1,y,255,255,255);
                //img.setVals(x-1,y+1,255,255,255);
                //img.setVals(x,y+1,255,255,255);
                //img.setVals(x+1,y+1,255,255,255);
                //img.setVals(x-2,y-2,0,0,0);
                //img.setVals(x-1,y-2,0,0,0);
                //img.setVals(x,y-2,0,0,0);
                //img.setVals(x+1,y-2,0,0,0);
                //img.setVals(x+2,y-2,0,0,0);
                //img.setVals(x-2,y-1,0,0,0);
                //img.setVals(x+2,y-1,0,0,0);
                //img.setVals(x-2,y,0,0,0);
                //img.setVals(x+2,y,0,0,0);
                //img.setVals(x-2,y+1,0,0,0);
                //img.setVals(x+2,y+1,0,0,0);
                //img.setVals(x-2,y+2,0,0,0);
                //img.setVals(x-1,y+2,0,0,0);
                //img.setVals(x,y+2,0,0,0);
                //img.setVals(x+1,y+2,0,0,0);
```

```
                    //img.setVals(x+2,y+2,0,0,0);
                }


            r=g=b = (int)(interpolator1.InterpolatePoint(pt,20));
            //g = (int)(interpolator2.InterpolatePoint(pt));
            //b = (int)(interpolator3.InterpolatePoint(pt));
            if (r < 0 || r > 255 || g < 0 || g > 255 || b < 0 || b > 255)
            {
                int min = r;
                if (g < min) min = g;
                if (b < min) min = b;
                r = 255; g=b=200+min;
            }

            r = r<0?0:r>255?255:r;
            g = g<0?0:g>255?255:g;
            b = b<0?0:b>255?255:b;
            //img.setVals(x,y,r,g,b);

            //if (!interpolator1.InsideConvexHull_2D(pt))
            if (pt.x < interpolator1.minX() || pt.x > interpolator1.maxX() || pt.y <
interpolator1.minY() || pt.y > interpolator1.maxY())
                img.setVals(x,y,255,255,255);
            else
                img.setVals(x,y,0,0,0);
        }
    }
    img.SaveFile("output.bmp");

}

void Test3D ()
{
    MTRand r;
    unsigned long tstart = time(0);
    unsigned long tlast = 0;

    nearestneighbor_interpolator nni;
    inversedistance_interpolator invdi;
    average_interpolator avgi;
    volumespline_interpolator vsi;
    multiquadric_interpolator mqi;
    spherical_interpolator lini;
    spherical_interpolator msi;
```

```
double sum_error[7] = {0};
double sum_error_squared[7] = {0};
int error_count = 0;

std::ifstream fin ("DIESEL0.txt", std::ios::in);
#define DATASIZE 145

double vals[DATASIZE];
point3D locs[DATASIZE];

for (int i=0;i<DATASIZE; i++)
{
    fin >> locs[i].x;
    fin >> locs[i].y;
    fin >> locs[i].z;
    fin >> vals[i];
}

for (int i=0;i<DATASIZE; i++)
{
    if ((time(0) - tlast) > 4)
    {
        for (int k=0;k<7;k++)
            cout << sqrt(sum_error_squared[k]/error_count) << "\n";
        cout << error_count << "\n\n";
        tlast = time(0);
    }

    double vals2[DATASIZE-1];
    point3D locs2[DATASIZE-1];

    for (int j=0;j<DATASIZE;j++)
    {
        if (j < i)
        {
            vals2[j] = vals[j];
            locs2[j] = locs[j];
        }
        else if (j > i)
        {
            vals2[j-1] = vals[j];
            locs2[j-1] = locs[j];
        }
    }
```

```
    nni.RegisterSparseData(locs2, vals2, DATASIZE-1);
    invdi.RegisterSparseData(locs2, vals2, DATASIZE-1);
    avgi.RegisterSparseData(locs2, vals2, DATASIZE-1);
    vsi.RegisterSparseData(locs2, vals2, DATASIZE-1);
    mqi.RegisterSparseData(locs2, vals2, DATASIZE-1);
    lini.RegisterSparseData(locs2, vals2, DATASIZE-1);
    msi.RegisterSparseData(locs2, vals2, DATASIZE-1);

    double actual = vals[i];
    point3D testval = locs[i];
    double ee;

    ee = nni.InterpolatePoint(testval)-actual;
    sum_error[0] += fabs(ee);
    sum_error_squared[0] += ee*ee;

    ee = invdi.InterpolatePoint(testval)-actual;
    sum_error[1] += fabs(ee);
    sum_error_squared[1] += ee*ee;

    ee = avgi.InterpolatePoint(testval)-actual;
    sum_error[2] += fabs(ee);
    sum_error_squared[2] += ee*ee;

    ee = vsi.InterpolatePoint(testval)-actual;
    sum_error[3] += fabs(ee);
    sum_error_squared[3] += ee*ee;

    ee = mqi.InterpolatePoint(testval)-actual;
    sum_error[4] += fabs(ee);
    sum_error_squared[4] += ee*ee;

    ee = lini.InterpolatePoint(testval, 1)-actual;
    sum_error[5] += fabs(ee);
    sum_error_squared[5] += ee*ee;

    ee = msi.InterpolatePoint(testval, 2)-actual;
    sum_error[6] += fabs(ee);
    sum_error_squared[6] += ee*ee;

    error_count++;
}

for (int k=0;k<7;k++)
```

```
            cout << sqrt(sum_error_squared[k]/error_count) << "\n";
        cout << error_count << "\n";

}

void Test2D ()
{
    MTRand r;
    unsigned long tstart = time(0);
    unsigned long tlast = 0;

    nearestneighbor_interpolator nni;
    inversedistance_interpolator invdi;
    average_interpolator avgi;
    thinplatespline_interpolator tpsi;
    spherical_interpolator lini;
    spherical_interpolator msi;

    //double PERCENTAGE = .1;
    int NUMBER = 1000;  //(int)(img.width*img.height*PERCENTAGE)
    bool INCLUSIVE = false;

    double sum_error[6] = {0};
    double sum_error_squared[6] = {0};
    int error_count = 0;
    int inside_count = 0;

    for (int i=0; i<10; i++)
    {
        bitmapHeader img(1,1);
        char fname[100] = "2D_";
        char fnumber[100] = "";
        itoa(i, fnumber, 100);
        char extention [100] = ".bmp";
        img.LoadFile(strcat(strcat(fname, fnumber), extention));

        point3D * coords = new point3D[NUMBER];
        double * vals = new double[NUMBER];

        for (int l=0;l<1000;l++)
        {

            //if ((time(0) - tlast) > 4)
            if (false)
            {
```

```
    for (int k=0;k<6;k++)
        cout << sqrt(sum_error_squared[k]/error_count) << "\n";
    cout << error_count << "\n\n";
    tlast = time(0);
}

int count = 0;
do
{
    point3D pt;
    pt.x = (int)(r.rand() * img.width);
    pt.y = (int)(r.rand() * img.height);
    bool continuedo = false;
    for (int k=0;k<count;k++)
        if (pt==coords[k]) continuedo=true;
    if (continuedo)
        continue;
    coords[count] = pt;
    vals[count] = img.getVal(pt.x, pt.y, RR);
    count++;
}
while (count < NUMBER);

nni.RegisterSparseData (coords, vals, NUMBER);
invdi.RegisterSparseData (coords, vals, NUMBER);
avgi.RegisterSparseData (coords, vals, NUMBER);
tpsi.RegisterSparseData (coords, vals, NUMBER);
lini.RegisterSparseData (coords, vals, NUMBER);
msi.RegisterSparseData (coords, vals, NUMBER);

for (int k=0;k<10;k++)
{
    point3D testval;
    bool canuse=false;
    do
    {
        testval.x = (int)(r.rand()*img.width);
        testval.y = (int)(r.rand()*img.height);

        canuse = true;
        if (INCLUSIVE && !nni.InsideConvexHull_2D(testval))
            canuse = false;

        for (int l=0;l<NUMBER;l++)
            if (coords[l] == testval)
```

```
                    canuse = false;
                }
                while (!canuse);

                double actual = img.getVal(testval.x, testval.y, RR);
                double ee;

                ee = nni.InterpolatePoint(testval)-actual;
                sum_error[4] += fabs(ee);
                sum_error_squared[4] += ee*ee;

                ee = invdi.InterpolatePoint(testval)-actual;
                sum_error[3] += fabs(ee);
                sum_error_squared[3] += ee*ee;

                ee = avgi.InterpolatePoint(testval)-actual;
                sum_error[5] += fabs(ee);
                sum_error_squared[5] += ee*ee;

                ee = tpsi.InterpolatePoint(testval, 100)-actual;
                if (ee == ee && ee*ee < 1e20)
                {
                    sum_error[2] += fabs(ee);
                    sum_error_squared[2] += ee*ee;
                }

                ee = lini.InterpolatePoint(testval, 1, true)-actual;
                sum_error[1] += fabs(ee);
                sum_error_squared[1] += ee*ee;

                ee = msi.InterpolatePoint(testval, 2, true)-actual;
                sum_error[0] += fabs(ee);
                sum_error_squared[0] += ee*ee;

                error_count++;

                if (nni.InsideConvexHull_2D(testval))
                    inside_count++;
            }
        }

}

cout << NUMBER << "\n" << INCLUSIVE << "\n";
for (int k=0;k<6;k++)
```

```
            cout << sqrt(sum_error_squared[k]/error_count) << "\n";
        cout << error_count << "\n";
        cout << inside_count << "\n\n";

}

void Test1D ()
{
    MTRand r;
    unsigned long tstart = time(0);
    unsigned long tlast = 0;

    nearestneighbor_interpolator nni;
    inversedistance_interpolator invdi;
    average_interpolator avgi;
    cubicspline_interpolator csi;
    spherical_interpolator lini;
    spherical_interpolator msi;

    double PERCENTAGE = .9;
    bool INCLUSIVE = false;

    double sum_error[6] = {0};
    double sum_error_squared[6] = {0};
    int error_count = 0;

    for (int i=0; i<10; i++)
    {
        bitmapHeader img(1,1);
        char fname[100] = "2D_";
        char fnumber[100] = "";
        itoa(i, fnumber, 100);
        char extention [100] = ".bmp";
        img.LoadFile(strcat(strcat(fname, fnumber), extention));

        double * xvals = new double[(int)(img.width*PERCENTAGE)];
        double * yvals = new double[(int)(img.width*PERCENTAGE)];
        for (int j=0;j<(int)(img.width*PERCENTAGE);j++)
            xvals[j] = j;

        for (int j=0;j<img.height;j++)
        {

            for (int l=0;l<10;l++)
            {
```

```
                if ((time(0) - tlast) > 4)
                //if (false)
                {
                    for (int k=0;k<6;k++)
                        cout << sqrt(sum_error_squared[k]/error_count) << "\n";
                    cout << error_count << "\n\n";
                    tlast = time(0);
                }

                int count = 0;
                do
                {
                    int x = r.rand() * img.width;
                    bool continuedo = false;
                    for (int k=0;k<count;k++)
                        if (x==xvals[k]) continuedo = true;
                    if (continuedo) continue;
                    xvals[count] = x;
                    yvals[count] = img.getVal(x, j, RR);
                    count++;
                }
                while (count < (int)(img.width*PERCENTAGE));

                nni.RegisterSparseData (xvals, yvals, (int)(img.width*PERCENTAGE));
                invdi.RegisterSparseData (xvals, yvals,
(int)(img.width*PERCENTAGE));
                avgi.RegisterSparseData (xvals, yvals, (int)(img.width*PERCENTAGE));
                csi.RegisterSparseData (xvals, yvals, (int)(img.width*PERCENTAGE));
                lini.RegisterSparseData (xvals, yvals, (int)(img.width*PERCENTAGE));
                msi.RegisterSparseData (xvals, yvals, (int)(img.width*PERCENTAGE));

                for (int k=0;k<10;k++)
                {
                    point3D testval;
                    bool inuse=false;
                    do
                    {
                        if (INCLUSIVE)
                            testval.x = (int)(nni.minX() + r.rand()*(nni.maxX()-
nni.minX()));
                        else
                            testval.x = (int)(r.rand()*img.width);

                        inuse=false;
```

```
                    for (int l=0;l<(int)(img.width*PERCENTAGE);l++)
                        if (xvals[l] == testval.x)
                            inuse = true;
                }
                while (inuse);

                double actual = img.getVal(testval.x, j, RR);
                double ee;

                ee = nni.InterpolatePoint(testval)-actual;
                sum_error[0] += fabs(ee);
                sum_error_squared[0] += ee*ee;

                ee = invdi.InterpolatePoint(testval)-actual;
                sum_error[1] += fabs(ee);
                sum_error_squared[1] += ee*ee;

                ee = avgi.InterpolatePoint(testval)-actual;
                sum_error[2] += fabs(ee);
                sum_error_squared[2] += ee*ee;

                ee = csi.InterpolatePoint(testval)-actual;
                if (ee == ee && ee*ee < 1e20)
                {
                    //if (fabs(ee) > 1000)
                        //csi.QuickPrint();
                    sum_error[3] += fabs(ee);
                    sum_error_squared[3] += ee*ee;
                }

                ee = lini.InterpolatePoint(testval, 1, true)-actual;
                sum_error[4] += fabs(ee);
                sum_error_squared[4] += ee*ee;

                ee = msi.InterpolatePoint(testval, 2, true)-actual;
                sum_error[5] += fabs(ee);
                sum_error_squared[5] += ee*ee;

                error_count++;
            }
        }

    }

}
```

```
    cout << PERCENTAGE << "\n" << INCLUSIVE << "\n";
    for (int k=0;k<6;k++)
        cout << sqrt(sum_error_squared[k]/error_count) << "\n";
    cout << error_count << "\n\n";

}

/*************************************************************************
3Dgeomertry.h
*************************************************************************/


#ifndef D_3DGEOMETRY_H
#define D_3DGEOMETRY_H


#include <iostream>
#include <math.h>
using namespace std;

class point3D
{
public:

    double x,y,z;

    point3D (double xval = 0, double yval = 0, double zval = 0)
    {
        Set(xval, yval, zval);
    }

    point3D (const point3D & inp)
    {
        *this = inp;
    }

    point3D (float* xyz)
    {
        Set(xyz[0], xyz[1], xyz[2]);
    }

    point3D (double* xyz)
    {
        Set(xyz[0], xyz[1], xyz[2]);
```

```
}

point3D & operator = (const point3D & i)
{
    x=i.x;
    y=i.y;
    z=i.z;
    return *this;
}

const point3D operator + (const point3D & i) const
{
    point3D a;
    a.x = x+i.x;
    a.y = y+i.y;
    a.z = z+i.z;
    return a;
}

const point3D operator - (const point3D & i) const
{
    point3D a;
    a.x = x-i.x;
    a.y = y-i.y;
    a.z = z-i.z;
    return a;
}

const point3D operator - () const
{
    point3D a;
    a.x = -x;
    a.y = -y;
    a.z = -z;
    return a;
}

bool operator == (const point3D & i) const
{
    return (x==i.x && y==i.y && z==i.z);
}

bool operator != (const point3D & i) const
{
    return !(*this == i);
```

```
    }

    void Set (double xval = 0, double yval = 0, double zval = 0)
    {
        x=xval;
        y=yval;
        z=zval;
    }

    double DistanceFromOrigin () const
    {
        return sqrt(x*x + y*y + z*z);
    }

    ostream & Print (ostream & out) const
    {
        out << x << "," << y << "," << z;
        return out;
    }

};

class vector3D
{
public:

    point3D p;

    vector3D (double xval = 0, double yval = 0, double zval = 0)
    {
        Set(xval, yval, zval);
    }

    vector3D (const vector3D & inp)
    {
        *this = inp;
    }

    vector3D (const point3D & inp)
    {
        p = inp;
    }

    vector3D & operator = (const vector3D & i)
    {
```

```
        p=i.p;
        return *this;
    }

    vector3D & operator = (const point3D & i)
    {
        p=i;
    }

    const vector3D operator - () const
    {
        vector3D result;
        result.p=-p;
        return result;
    }

    const point3D operator + (const point3D & i) const
    {
        point3D result;
        result = i+p;
        return result;
    }

    const vector3D operator + (const vector3D & i) const
    {
        vector3D result;
        result = i.p+p;
        return result;
    }

    static double AngleBetweenVectors (const vector3D &v1, const vector3D &v2)
    {
        double cos = DotProduct(v1,v2) / (v1.Length()*v2.Length());
        if (cos > 1) cos = 1; // (sometimes this ends up 1.000000001 if vectors are very
close)
        return acos(cos);
    }

    static double AngleBetweenVectors (const vector3D &v1, const vector3D &v2,
double v1Length, double v2Length)
    {
        double cos = DotProduct(v1,v2) / (v1Length*v2Length);
        if (cos > 1) cos = 1; // (sometimes this ends up 1.000000001 if vectors are very
close)
        return acos(cos);
```

```
    }

    static double CosAngleBetweenVectors (const vector3D &v1, const vector3D &v2)
    {
        double cos = DotProduct(v1,v2) / (v1.Length()*v2.Length());
        if (cos > 1) cos = 1;
        return cos;
    }

    static double CosAngleBetweenVectors (const vector3D &v1, const vector3D &v2,
double v1Length, double v2Length)
    {
        double cos = DotProduct(v1,v2) / (v1Length*v2Length);
        if (cos > 1) cos = 1; // (sometimes this ends up 1.000000001 if vectors are very
close)
        return cos;
    }

    double AngleTo (const vector3D &v1) const
    {
        return AngleBetweenVectors(*this, v1);
    }

    static const vector3D & CrossProduct (const vector3D &v1, const vector3D &v2)
    {
        static vector3D result;
        result.p.x = v1.p.x*v2.p.z - v1.p.z*v2.p.y;
        result.p.y = v1.p.z*v2.p.x - v1.p.x*v2.p.z;
        result.p.z = v1.p.x*v2.p.y - v1.p.y*v2.p.x;
        return result;
    }

    static double DotProduct (const vector3D &v1, const vector3D &v2)
    {
        return v1.p.x*v2.p.x + v1.p.y*v2.p.y + v1.p.z*v2.p.z;
    }

    double Length () const
    {
        return p.DistanceFromOrigin();
    }

    vector3D & Scale (double scalingFactor)
    {
        p.x *= scalingFactor;
```

```
        p.y *= scalingFactor;
        p.z *= scalingFactor;
        return *this;
    }

    vector3D & Normalize ()
    {
        Scale(1.0/Length());
        return *this;
    }

    static const vector3D & Project (const vector3D &Projecting, const vector3D
&OnTo)
    {
        static vector3D result;
        double ontolength = OnTo.Length();
        result = OnTo;
        result.Scale(DotProduct(Projecting, OnTo) / (ontolength*ontolength));
        return result;
    }

    ostream & Print (ostream & out) const
    {
        return p.Print(out);
    }

    void Set (double xval = 0, double yval = 0, double zval = 0)
    {
        p.Set(xval, yval, zval);
    }

};

class plane3D
{
public:

    //ax + by + cz + d = 0;
    //[a,b,c] is the normalized normal vector (n)
    vector3D n; // normal (x^2+y^2+z^2=1 AT ALL TIMES) (invariant)
    double d;

    plane3D (double aval=0, double bval=0, double cval=0, double dval=0)
    {
        Set(aval, bval, cval, dval);
```

```
}

plane3D (const point3D &p1, const point3D &p2, const point3D &p3)
{
    Set(p1,p2,p3);
}

plane3D (const vector3D &normal, double dval)
{
    Set(normal,dval);
}

plane3D (const vector3D &normal, const point3D p)
{
    Set(normal,p);
}

plane3D (const plane3D & inp)
{
    *this = inp;
}

plane3D & operator = (const plane3D & inp)
{
    Set(inp.n, inp.d);
}

void Set (double aval=0, double bval=0, double cval=0, double dval=0)
{
    n.Set(aval, bval, cval);
    d=dval;
}

void Set (const point3D &p1, const point3D &p2, const point3D &p3)
{
    if (p1 == p2 || p1 == p3 || p2 == p3) return;
    vector3D n = vector3D::CrossProduct(p2-p1, p3-p1);
    n.Normalize();
    d = -(vector3D::DotProduct(n, p1));
}

void Set (const vector3D &normal, const double dval)
{
    n = normal;
    d = dval;
```

```
    }

    void Set (const vector3D &normal, const point3D p)
    {
        n = normal;
        d = -(vector3D::DotProduct(n, p));
    }

    double ClosestDistanceTo (const point3D & pt)
    {
        /*
        returns positive if point is on the side of the normal,
        returns negative if the point is on the opposite side of normal,
        */
        return vector3D::DotProduct(n, pt) + d;
    }

    const point3D ClosestPointTo (const point3D &pt)
    {
        vector3D tmp = n;
        double dist = ClosestDistanceTo(pt);
        return pt - tmp.Scale(dist).p;
    }

    ostream & Print (ostream & out) const
    {
        n.Print(out);
        out << ",";
        out << d;
        return out;
    }


};

#endif

/*************************************************************************
opengl_code.h
*************************************************************************/

struct gl_cube
{
    struct
    {
```

```
        float pos[3];
        float col[3];
    }ver[8];

    struct
    {
        unsigned int ver[4];
    } quad[6];

    bool isedge;
};

struct gl_color
{
    float rgb[3];
};

int Test3D_2(int argc, char **argv);


/*********************************************************************
opengl_code.cpp
*********************************************************************/

#include <windows.h>
#include <gl\gl.h>
#include <gl\glu.h>
#include <gl\glut.h>

#include <iostream>
#include <fstream>

#include "all_sparse_interpolators.h"
#include "random.h"
#include "3Dgeometry.h"
#include "opengl_code.h"
#include "zpr.h"


static void redraw();
int main(int argc, char **argv);
void initCube(gl_cube & cube, point3D center, double width, gl_color * colors, double
        xstretch=1, double ystretch=1, double zstretch=1, bool singleColor=false);
void remapColor (gl_color & c, double val);
void drawObjects();
```

```
void performCalculations();

gl_cube *cubes;
int cubeCount;

gl_cube *controlpts;
int controlptCount;

void performCalculations()
{
    int resolution = 20;
    cubeCount = resolution*resolution*resolution;
    cubes = new gl_cube[cubeCount];

    std::ifstream fin ("DIESEL0.txt", std::ios::in);
    #define DATASIZE 145

    double vals[DATASIZE];
    point3D locs[DATASIZE];
    point3D locs2[DATASIZE];

    for (int i=0;i<DATASIZE; i++)
    {
        fin >> locs[i].x;
        fin >> locs[i].y;
        fin >> locs[i].z;
        fin >> vals[i];
    }

    //spherical_interpolator msi;
    //volumespline_interpolator msi;
    //inversedistance_interpolator msi;
    //multiquadric_interpolator msi;
    nearestneighbor_interpolator msi;
    msi.RegisterSparseData(locs, vals, DATASIZE-1);

    double xdiam = (msi.maxX()-msi.minX())/resolution;
    double ydiam = (msi.maxY()-msi.minY())/resolution;
    double zdiam = (msi.maxZ()-msi.minZ())/resolution;
    cubeCount = 0;
    for (int x=0;x<resolution;x++)
    {
        cout << "row " << x << " complete\n";
        for (int y=0;y<resolution;y++)
            for (int z=0;z<resolution;z++)
```

```
    {
        if (x<10&&y<10) continue;

        gl_color c[8];
   c[0].rgb[0]=c[0].rgb[1]=c[0].rgb[2] =
(msi.InterpolatePoint(point3D(msi.minX()+(x)*xdiam,msi.minY()+(y+1)*ydiam,
msi.minZ()+(z+1)*zdiam)) - msi.minV()) / (msi.maxV()-msi.minV());
   c[1].rgb[0]=c[1].rgb[1]=c[1].rgb[2] =
(msi.InterpolatePoint(point3D(msi.minX()+(x+1)*xdiam,msi.minY()+(y+1)*ydia
m,msi.minZ()+(z+1)*zdiam)) - msi.minV()) / (msi.maxV()-msi.minV());
   c[2].rgb[0]=c[2].rgb[1]=c[2].rgb[2] =
(msi.InterpolatePoint(point3D(msi.minX()+(x+1)*xdiam,msi.minY()+(y+1)*ydia
m,msi.minZ()+(z)*zdiam)) - msi.minV()) / (msi.maxV()-msi.minV());
   c[3].rgb[0]=c[3].rgb[1]=c[3].rgb[2] =
(msi.InterpolatePoint(point3D(msi.minX()+(x)*xdiam,msi.minY()+(y+1)*ydiam,
msi.minZ()+(z)*zdiam)) - msi.minV()) / (msi.maxV()-msi.minV());
   c[4].rgb[0]=c[4].rgb[1]=c[4].rgb[2] =
(msi.InterpolatePoint(point3D(msi.minX()+(x)*xdiam,msi.minY()+(y)*ydiam,ms
i.minZ()+(z+1)*zdiam)) - msi.minV()) / (msi.maxV()-msi.minV());
   c[5].rgb[0]=c[5].rgb[1]=c[5].rgb[2] =
(msi.InterpolatePoint(point3D(msi.minX()+(x+1)*xdiam,msi.minY()+(y)*ydiam,
msi.minZ()+(z+1)*zdiam)) - msi.minV()) / (msi.maxV()-msi.minV());
   c[6].rgb[0]=c[6].rgb[1]=c[6].rgb[2] =
(msi.InterpolatePoint(point3D(msi.minX()+(x+1)*xdiam,msi.minY()+(y)*ydiam,
msi.minZ()+(z)*zdiam)) - msi.minV()) / (msi.maxV()-msi.minV());
   c[7].rgb[0]=c[7].rgb[1]=c[7].rgb[2] =
(msi.InterpolatePoint(point3D(msi.minX()+(x)*xdiam,msi.minY()+(y)*ydiam,ms
i.minZ()+(z)*zdiam)) - msi.minV()) / (msi.maxV()-msi.minV());

        remapColor(c[0], c[0].rgb[0]);
        remapColor(c[1], c[1].rgb[0]);
        remapColor(c[2], c[2].rgb[0]);
        remapColor(c[3], c[3].rgb[0]);
        remapColor(c[4], c[4].rgb[0]);
        remapColor(c[5], c[5].rgb[0]);
        remapColor(c[6], c[6].rgb[0]);
        remapColor(c[7], c[7].rgb[0]);

        double d = 20/resolution;
        double xcenter = x*d + d/2;
        double ycenter = y*d + d/2;
        double zcenter = z*d + d/2;
        initCube(cubes[cubeCount], point3D(xcenter, ycenter, zcenter),
20/resolution, c);
```

```
                cubes[cubeCount].isedge = x==0||x==resolution-1||y==0||y==resolution-
        1||z==0||z==resolution-1;
                cubeCount++;
            }
    }

    controlptCount = DATASIZE;
    controlpts = new gl_cube[controlptCount];
    float boxcolor[3] = {1,0,1};
    for (int i=0;i<controlptCount;i++)
    {
        gl_color c;
        c.rgb[0] = 1; c.rgb[1]=0; c.rgb[2] = 1;
        point3D pt;
        pt.x = 20*(locs[i].x-msi.minX())/(msi.maxX()-msi.minX());
        pt.y = 20*(locs[i].y-msi.minY())/(msi.maxY()-msi.minY());
        pt.z = 20*(locs[i].z-msi.minZ())/(msi.maxZ()-msi.minZ());
        initCube(controlpts[i], pt, .25, &c,1,1,1,true);
    }

}

void remapColor (gl_color & c, double val)
{
    if (val < 0)
    {
        c.rgb[0] = 0;
        c.rgb[1] = .5+val*3;
        c.rgb[2] = -val;
        if (c.rgb[2] > 1) c.rgb[2] = 1;
        if (c.rgb[1] < 0) c.rgb[1] = 0;
    }
    else if (val >= 0 && val < .0001)
    {
        c.rgb[0] = 85/255.;
        c.rgb[1] = 212/255.;
        c.rgb[2] = 208/255.;
    }
    else if (val >= .0001)
    {
        c.rgb[0] = val/2;
        c.rgb[1] = .5-val;
        c.rgb[2] = 0;
    }
```

```
}


void initCube(gl_cube & cube, point3D center, double width, gl_color * colors, double
        xstretch, double ystretch, double zstretch, bool singleColor)
{
    float r=width/2;
    float vertexPosDat[8][3]=
    {
        {center.x-r*xstretch,center.y+r*ystretch,center.z+r*zstretch}, //left,top,front
        {center.x+r*xstretch,center.y+r*ystretch,center.z+r*zstretch}, //right,top,front
        {center.x+r*xstretch,center.y+r*ystretch,center.z-r*zstretch}, //right,top,back
        {center.x-r*xstretch,center.y+r*ystretch,center.z-r*zstretch}, //left, top,back
        {center.x-r*xstretch,center.y-r*ystretch,center.z+r*zstretch}, //left,bottom,front
        {center.x+r*xstretch,center.y-r*ystretch,center.z+r*zstretch}, //right,bottom,front
        {center.x+r*xstretch,center.y-r*ystretch,center.z-r*zstretch}, //right,bottom,back
        {center.x-r*xstretch,center.y-r*ystretch,center.z-r*zstretch}  //left,bottom,back
    };

    //defines the vertexes of each quad in anti-clockwise order
    unsigned int quadVerDat[6][4]=
                                        {
                                                {0,1,2,3}, //top
                                                {0,3,7,4}, //left
                                                {3,2,6,7}, //back
                                                {2,1,5,6}, //right
                                                {0,4,5,1}, //front
                                                {4,7,6,5}, //bottom
                                        };


    int a,b;
    //put the vertex data into the cube struct
    for (a=0;a<8;++a)
    {
        for (b=0;b<3;++b)
        {
            cube.ver[a].pos[b]=vertexPosDat[a][b];
            if (colors)
            {
                if (singleColor)
                    cube.ver[a].col[b]=colors[0].rgb[b];
                else
                    cube.ver[a].col[b]=colors[a].rgb[b];
            }
```

```
        }
    }

    //put the quad data into the cube struct
    for (a=0;a<6;++a)
    {
        for (b=0;b<4;++b)
        {
            cube.quad[a].ver[b]=quadVerDat[a][b];
        }
    }

}

static void drawObjects()
{

    glPushMatrix();

        glTranslatef(-2,-2,-2);
        glRotatef(20,0,.3,0);
        glRotatef(-110,1,0,0);
        glScalef(.2,.2,.2);

        for (int i=0;i<cubeCount; i++)
        {
            glBegin(GL_QUADS);
            for (int j=0;j<6;++j)
                for (int k=0;k<4;++k)
                {
                    int currentVer=cubes[i].quad[j].ver[k];
                    glColor3fv(cubes[i].ver[ currentVer ].col);
                    glVertex3fv(cubes[i].ver[ currentVer ].pos);
                }
            glEnd();
        }

        for (int i=0;i<controlptCount; i++)
        {
            glBegin(GL_QUADS);
            for (int j=0;j<6;++j)
                for (int k=0;k<4;++k)
                {
                    int currentVer=controlpts[i].quad[j].ver[k];
                    glColor3fv(controlpts[i].ver[ currentVer ].col);
```

```
                glVertex3fv(controlpts[i].ver[ currentVer ].pos);
            }
        glEnd();
    }

    glLineWidth(4);
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    gl_cube box, box2;
    initCube(box, point3D(10,10,10), 20, 0);
    float boxcolor[3] = {1,1,1};
    glBegin(GL_QUADS);
    for (int j=0;j<6;++j)
        for (int k=0;k<4;++k)
        {
            int currentVer=box.quad[j].ver[k];
            glColor3fv(boxcolor);
            glVertex3fv(box.ver[ currentVer ].pos);
        }
    glEnd();
    initCube(box2, point3D(5,5,10), 9.95, 0,1,1,2);
    glLineWidth(1);
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    glBegin(GL_QUADS);
    for (int j=0;j<6;++j)
        for (int k=0;k<4;++k)
        {
            int currentVer=box2.quad[j].ver[k];
            glColor3fv(boxcolor);
            glVertex3fv(box2.ver[ currentVer ].pos);
        }
    glEnd();
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);

  glPopMatrix();

  glPushMatrix();
    glTranslatef(0,0,-100);
  glPopMatrix();

}


void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
  drawObjects();
  glutSwapBuffers();
}

void pick(GLint name)
{
  printf("Pick: %d\n",name);
  fflush(stdout);
}

int Test3D_2(int argc, char **argv)
{
   /* Initialise GLUT and create a window */

   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize(900,900);
   glutCreateWindow("GLT Mouse Zoom-Pan-Rotate");

   /* Configure GLUT callback functions */

   glutDisplayFunc(display);

   glScalef(0.25,0.25,0.25);

   // Configure ZPR module -
     // this is used to easily manipulate the 3D image with the mouse.

   zprInit();
   zprSelectionFunc(drawObjects);     /* Selection mode draw function */
   zprPickFunc(pick);              /* Pick event client callback   */

    /* Initialise OpenGL */

   glDepthFunc(GL_LESS);
   glEnable(GL_DEPTH_TEST);
   glEnable(GL_NORMALIZE);
   glEnable(GL_COLOR_MATERIAL);
     glMatrixMode(GL_MODELVIEW);

     /* Enter GLUT event loop */

     performCalculations();

   glutMainLoop();
```

```
   return 0;
}

/*********************************************************************
bitmap.h
*********************************************************************/


#ifndef BITMAPHEADER_H
#define BITMAPHEADER_H

#include <fstream>


typedef enum {
    Alpha = 262144,
    Canonical = 2097152,
    DontCare = 0,
    Extended = 1048576,
    Format16bppArgb1555 = 397319,
    Format16bppGrayScale = 1052676,
    Format16bppRgb555 = 135173,
    Format16bppRgb565 = 135174,
    Format1bppIndexed = 196865,
    Format24bppRgb = 137224,
    Format32bppArgb = 2498570,
    Format32bppPArgb = 925707,
    Format32bppRgb = 139273,
    Format48bppRgb = 1060876,
    Format4bppIndexed = 197634,
    Format64bppArgb = 3424269,
    Format64bppPArgb = 1851406,
    Format8bppIndexed = 198659,
    Gdi = 131072,
    Indexed = 65536,
    Max = 15,
    PAlpha = 524288,
    Undefined = 0
} PixelFormat;


#define RR 0
#define GG 1
#define BB 2
```

```
#define AA 3
#define TYPE_ARGB32 2498570
#define TYPE_RGB24 137224
#define TYPE_RGB32 139273

class bitmapHeader {

    public:
        unsigned long width;
        unsigned long height;
        unsigned long stride;
        unsigned long type;
        unsigned char * data;

    public:
        unsigned char & getVal (int x, int y, int rgba) const {
            static unsigned char defRet = 100;
            if (x >= (int)width) x = width - 1; else if (x < 0) x = 0;
            if (y >= (int)height) y = height - 1; else if (y < 0) y = 0;
            if (type == TYPE_ARGB32) {
                if (rgba == BB) return data[y*stride + x*4];
                if (rgba == GG) return data[y*stride + x*4+1];
                if (rgba == RR) return data[y*stride + x*4+2];
                if (rgba == AA) return data[y*stride + x*4+3];
            } else if (type == TYPE_RGB24) {
                if (rgba == BB) return data[y*stride + x*3];
                if (rgba == GG) return data[y*stride + x*3+1];
                if (rgba == RR) return data[y*stride + x*3+2];
                if (rgba == AA) return defRet=255;
            } else if (type == TYPE_RGB32) {
                if (rgba == BB) return data[y*stride + x*4];
                if (rgba == GG) return data[y*stride + x*4+1];
                if (rgba == RR) return data[y*stride + x*4+2];
                if (rgba == AA) return defRet=255;
            } else if (type == Format8bppIndexed) {
                if (rgba != AA) return data[y*stride + x];
                if (rgba == AA) return defRet=255;
            }
            return defRet;
        }

        void getVals (int x, int y, int &r, int &g, int &b, int &a) const
        {
            if (x >= (int)width) x = width - 1; else if (x < 0) x = 0;
            if (y >= (int)height) y = height - 1; else if (y < 0) y = 0;
```

```
    if (type == TYPE_ARGB32)
    {
        unsigned char* ptr = data + (y*stride + x*4);
        b = ptr[0];
        g = ptr[1];
        r = ptr[2];
        a = ptr[3];
    }
    else if (type == TYPE_RGB24)
    {
        unsigned char* ptr = data + (y*stride + x*3);
        b = ptr[0];
        g = ptr[1];
        r = ptr[2];
        a = 255;
    }
    else if (type == TYPE_RGB32)
    {
        unsigned char* ptr = data + (y*stride + x*4);
        b = ptr[0];
        g = ptr[1];
        r = ptr[2];
        a = 255;
    }
    else if (type == Format8bppIndexed)
    {
        unsigned char* ptr = data + (y*stride + x);
        r = g = b = ptr[0];
        a = 255;
    }
}

void setVals (int x, int y, int r, int g, int b, int a = 255)
{
    if (x >= (int)width) x = width - 1; else if (x < 0) x = 0;
    if (y >= (int)height) y = height - 1; else if (y < 0) y = 0;
    if (type == TYPE_ARGB32)
    {
        unsigned char* ptr = data + (y*stride + x*4);
        ptr[0] = b;
        ptr[1] = g;
        ptr[2] = r;
        ptr[3] = a;
    }
    else if (type == TYPE_RGB24)
```

```
    {
        unsigned char* ptr = data + (y*stride + x*3);
        ptr[0] = b;
        ptr[1] = g;
        ptr[2] = r;
    }
    else if (type == TYPE_RGB32)
    {
        unsigned char* ptr = data + (y*stride + x*4);
        ptr[0] = b;
        ptr[1] = g;
        ptr[2] = r;
    }
    else if (type == Format8bppIndexed)
    {
        unsigned char* ptr = data + (y*stride + x);
        ptr[0] = r;
    }
}

void getHSLA (int x, int y, float &h, float &s, float &l, int &a) const
{
    int intr,intg,intb;
    getVals(x,y,intr,intg,intb,a);
    double r = intr/255.;
    double g = intg/255.;
    double b = intb/255.;

    if (r==g && r==b)
    {
        h = (float)-1;
        s = (float)0;
        l = (float)r;
        return;
    }

    double max = (r >= g ? (r >= b ? r : b) : (g >= b ? g : b));
    double min = (r <= g ? (r <= b ? r : b) : (g <= b ? g : b));
    double delta = max-min;

    if (r == max)
        h = (float)((60./360.) * ((g-b)/delta));
    else if (g == max)
        h = (float)((60./360.) * ((b-r)/delta) + (120./360.));
    else if (b == max)
```

```
        h = (float)((60./360.) * ((r-g)/delta) + (240./360.));
    if (h < 0 || h >= 1) h = h-floor(h);

    l = (float)((max+min)/2);
    s = (float)((l<.5 ? (max-min)/(max+min) : (max-min)/(2-max-min)));
}

void setHSLA (int x, int y, float h, float s, float l, int a = 255)
{
    if (h < 0 || h >= 1) h = h-floor(h);
    if (s < 0) s = 0;
    if (s > 1) s = 1;
    if (l < 0) l = 0;
    if (l > 1) l = 1;

    int r,g,b;
    if (s == 0)
    {
        r = g = b = (int)floor(l*255+.5);
        setVals(x,y,r,g,b,a);
        return;
    }

    float v1,v2;
    if (l<.5) v2 = l*(1+s);
    else v2 = (l+s)-(l*s);
    v1 = 2*l - v2;

    r = (int)floor(HueToRGB(v1,v2,(float)(h+(1./3.)))*255 + .5);
    g = (int)floor(HueToRGB(v1,v2,h)*255 + .5);
    b = (int)floor(HueToRGB(v1,v2,h-(1./3.))*255 + .5);

    setVals(x,y,r,g,b,a);
}

float HueToRGB(float v1, float v2, float h)
{
    if (h < 0 || h >= 1) h = h-floor(h);
    if (6*h < 1) return v1 + (v2-v1)*6*h;
    if (2*h < 1) return v2;
    if (3*h < 2) return v1 + (v2-v1)*((2./3.)-h)*6;
    return v1;
}

void getHSVA (int x, int y, float &h, float &s, float &v, int &a) const
```

```
{
    int intr,intg,intb;
    getVals(x,y,intr,intg,intb,a);
    double r = intr/255.;
    double g = intg/255.;
    double b = intb/255.;

    if (r==g && r==b)
    {
        s = 0;
        h = -1;
        v = r;
        return;
    }

    double max = (r >= g ? (r >= b ? r : b) : (g >= b ? g : b));
    double min = (r <= g ? (r <= b ? r : b) : (g <= b ? g : b));
    double delta = max-min;

    if (r == max)
        h = (60./360.) * ((g-b)/delta);
    else if (g == max)
        h = (60./360.) * ((b-r)/delta) + (120./360.);
    else if (b == max)
        h = (60./360.) * ((r-g)/delta) + (240./360.);
    if (h < 0 || h >= 1) h = h-floor(h);

    s = (max-min) / max;
    v = max;
}

void setHSVA (int x, int y, float h, float s, float v, int a = 255)
{
    if (h < 0 || h >= 1) h = h-floor(h);
    if (s < 0) s = 0;
    if (s > 1) s = 1;
    if (v < 0) v = 0;
    if (v > 1) v = 1;

    int r,g,b;
    if (s == 0)
    {
        r = g = b = (int)floor(v*255+.5);
        setVals(x,y,r,g,b,a);
        return;
```

```
}

int h_i = (int)h*6;
float f = h*6-h_i;
float p = v * (1. - s);
float q = v * (1. - f*s);
float t = v * (1. - s*(1.-f));

if (h_i == 0)
{
    r = (int)floor(v*255+.5);
    g = (int)floor(t*255+.5);
    b = (int)floor(p*255+.5);
}
else if (h_i == 1)
{
    r = (int)floor(q*255+.5);
    g = (int)floor(v*255+.5);
    b = (int)floor(p*255+.5);
}
else if (h_i == 2)
{
    r = (int)floor(p*255+.5);
    g = (int)floor(v*255+.5);
    b = (int)floor(t*255+.5);
}
else if (h_i == 3)
{
    r = (int)floor(p*255+.5);
    g = (int)floor(q*255+.5);
    b = (int)floor(v*255+.5);
}
else if (h_i == 4)
{
    r = (int)floor(t*255+.5);
    g = (int)floor(p*255+.5);
    b = (int)floor(v*255+.5);
}
else if (h_i == 5)
{
    r = (int)floor(v*255+.5);
    g = (int)floor(p*255+.5);
    b = (int)floor(q*255+.5);
}
```

```
    setVals(x,y,r,g,b,a);
}


unsigned char avgTone (int x, int y) const { // returns average tone of the pixel
    return (getVal(x, y, RR) + getVal(x, y, GG) + getVal(x, y, BB)) / 3;
}
/*unsigned long & getPixel32 (int x, int y) const {
    static unsigned long defRet = 0xffaa9988;
    if (type == TYPE_ARGB32) {
        return ((unsigned long *)data)[y*stride/4 + x];
    } else return defRet;
}*/
unsigned char * getData() const {
    return data;
}

int BitCount() const
{
    if (type == TYPE_ARGB32 || type == TYPE_RGB32)
        return 32;
    if (type == TYPE_RGB24)
        return 24;
}

unsigned long & PixelInt(int x, int y)
{
    int bits = BitCount();
    if (bits == 32)
        return *((unsigned long *) &(data[y*stride + x*4]));
    else if (bits == 24)
        return *((unsigned long *) &(data[y*stride + x*3]));
}

bitmapHeader (unsigned long w, unsigned long h, unsigned long t =
TYPE_ARGB32) {
    int multiplier;
    if (w < 1) width = 1; else width = w;
    if (h < 1) height = 1; else height = h;
    type = t;
    if (type == TYPE_ARGB32 || type == TYPE_RGB32) {
        multiplier = 4;
    } else if (type == TYPE_RGB24) {
        multiplier = 3;
    }
```

```
    stride = (((w * multiplier)-1)/4+1)*4;
    data = new unsigned char[height * stride];
}
~bitmapHeader () {
    delete data;
}
bitmapHeader & operator = (const bitmapHeader & rhs) {
    int cw, ch;
    if (width < rhs.width) cw = width; // determine width constraint
        else cw = rhs.width;
    if (height < rhs.height) ch = height; // determine height constraint
        else ch = rhs.height;
    for (int i = 0; i < cw; i++) {
        for (int j = 0; j < ch; j++) {
            getVal(i,j,RR) = rhs.getVal(i,j,RR);
            getVal(i,j,GG) = rhs.getVal(i,j,GG);
            getVal(i,j,BB) = rhs.getVal(i,j,BB);
        }
    }
    return *this;
}
void resize (unsigned long w, unsigned long h, unsigned long t =
TYPE_ARGB32) {
    int multiplier;
    if (w < 1) width = 1; else width = w;
    if (h < 1) height = 1; else height = h;
    type = t;
    if (type == TYPE_ARGB32 || type == TYPE_RGB32) {
        multiplier = 4;
    } else if (type == TYPE_RGB24) {
        multiplier = 3;
    }
    stride = (((w * multiplier)-1)/4+1)*4;
    delete [] data;
    data = new unsigned char[height * stride];
}

void Whitewash (int r = 255, int g = 255, int b = 255, int a = 255)
{
    for (int y=0;y<height;y++)
        for (int x=0;x<width;x++)
            this->setVals(x,y,r,g,b,a);
}

void LoadFile (char * fname)
```

```
{
    std::ifstream fin (fname, std::ios::in|std::ios::binary);

    unsigned long dataOffset;
    unsigned long filesize;
    unsigned long zero;
    unsigned long headerSize;
    unsigned long w;
    unsigned long h;
    unsigned short planes;
    unsigned short bitsperpixel;
    unsigned long compression;

    fin.ignore();
    fin.ignore();
    fin.read((char*)&filesize, sizeof(filesize));
    fin.read((char*)&zero, sizeof(zero));
    fin.read((char*)&dataOffset, sizeof(dataOffset));
    fin.read((char*)&headerSize, sizeof(headerSize));
    fin.read((char*)&w, sizeof(w));
    fin.read((char*)&h, sizeof(h));
    fin.read((char*)&planes, sizeof(planes));
    fin.read((char*)&bitsperpixel, sizeof(bitsperpixel));
    fin.read((char*)&compression, sizeof(compression));

    resize(w,h);

    fin.seekg(dataOffset, std::ios::beg);

    for (int i = height-1; i >= 0; i--)
    {

            for (int j = 0; j < width; j++)
            {
                if (bitsperpixel == 24)
                {
                    getVal(j, i, BB) = fin.get();
                    getVal(j, i, GG) = fin.get();
                    getVal(j, i, RR) = fin.get();
                }
            }
        for (int j = 0; j < (((width*3)-1)/4+1)*4-(width*3); j++)
            fin.ignore(); // extract 4bit wide padding
    }
```

```
    fin.close();
}

void SaveFile (char * fname) const
{
    std::ofstream fout (fname, std::ios::out|std::ios::binary);

    unsigned long dataOffset = 54;
    unsigned long filesize = dataOffset+(  (((width*3)-1)/4+1)*4  *height);
    unsigned long zero = 0;
    unsigned long headerSize = 0x28; // used with windows bmp
    unsigned long w = width;
    unsigned long h = height;
    unsigned short planes = 1;
    unsigned short bitsperpixel = 24;
    unsigned long compression = 0;
    unsigned long bitmapdatasize = 0;
    unsigned long resolution = 2834; // px/meter

    fout.write("BM",2*sizeof(char));
    fout.write((char*)&filesize, sizeof(filesize));
    fout.write((char*)&zero, sizeof(zero));
    fout.write((char*)&dataOffset, sizeof(dataOffset));
    fout.write((char*)&headerSize, sizeof(headerSize));
    fout.write((char*)&w, sizeof(w));
    fout.write((char*)&h, sizeof(h));
    fout.write((char*)&planes, sizeof(planes));
    fout.write((char*)&bitsperpixel, sizeof(bitsperpixel));
    fout.write((char*)&compression, sizeof(compression));
    fout.write((char*)&bitmapdatasize, sizeof(bitmapdatasize));
    fout.write((char*)&resolution, sizeof(resolution));
    fout.write((char*)&resolution, sizeof(resolution));
    fout.write((char*)&zero, sizeof(zero));
    fout.write((char*)&zero, sizeof(zero));

    for (int i = height-1; i >= 0; i--)
    {
        for (int j = 0; j < width; j++)
        {
            fout.put(getVal(j, i, BB));
            fout.put(getVal(j, i, GG));
            fout.put(getVal(j, i, RR));
        }
        for (int j = 0; j < (((width*3)-1)/4+1)*4-(width*3); j++)
            fout.put(1); // pad to 4 bytes wide
```

```
    }
    fout.put(0);
    fout.put(0);

    fout.close();
}

static void WriteBitmapHeader (char * fname, unsigned long w, unsigned long h,
unsigned long DPM = 2834)
{
    std::ofstream fout (fname, std::ios::out|std::ios::binary);
    WriteBitmapHeader(fout, w, h, DPM);
    fout.close();
}

static void WriteBitmapHeader (std::ofstream &file, unsigned long w, unsigned
long h, unsigned long DPM = 2834)
{
    unsigned long dataOffset = 54;
    unsigned long filesize = dataOffset+(  (((w*3)-1)/4+1)*4  *h);
    unsigned long zero = 0;
    unsigned long headerSize = 0x28; // used with windows bmp
    unsigned short planes = 1;
    unsigned short bitsperpixel = 24;
    unsigned long compression = 0;
    unsigned long bitmapdatasize = 0;
    unsigned long resolution = DPM; // px/meter

    file.write("BM",2*sizeof(char));
    file.write((char*)&filesize, sizeof(filesize));
    file.write((char*)&zero, sizeof(zero));
    file.write((char*)&dataOffset, sizeof(dataOffset));
    file.write((char*)&headerSize, sizeof(headerSize));
    file.write((char*)&w, sizeof(w));
    file.write((char*)&h, sizeof(h));
    file.write((char*)&planes, sizeof(planes));
    file.write((char*)&bitsperpixel, sizeof(bitsperpixel));
    file.write((char*)&compression, sizeof(compression));
    file.write((char*)&bitmapdatasize, sizeof(bitmapdatasize));
    file.write((char*)&resolution, sizeof(resolution));
    file.write((char*)&resolution, sizeof(resolution));
    file.write((char*)&zero, sizeof(zero));
    file.write((char*)&zero, sizeof(zero));
}
```

```
        void WriteRowToBitmapFile (char * fname, int row) const
        {
            std::ofstream fout (fname, std::ios::out|std::ios::binary|std::ios::app);
            WriteRowToBitmapFile(fout, row);
            fout.close();
        }

        void WriteRowToBitmapFile (std::ofstream &file, int row) const
        {
            for (int j = 0; j < width; j++)
            {
                //file.put(getVal(j, row, BB));
                //file.put(getVal(j, row, GG));
                //file.put(getVal(j, row, RR));
                int r,g,b,a;
                getVals(j, row, r,g,b,a);
                file.put((unsigned char)b);
                file.put((unsigned char)g);
                file.put((unsigned char)r);
            }
            for (int j = 0; j < (((width*3)-1)/4+1)*4-(width*3); j++)
                file.put(1); // pad to 4 bytes wide
        }

        static void FinalizeBitmapFile (char * fname)
        {
            std::ofstream fout (fname, std::ios::out|std::ios::binary|std::ios::app);
            FinalizeBitmapFile(fout);
            fout.close();
        }

        static void FinalizeBitmapFile (std::ofstream &file)
        {
            file.put(0);
            file.put(0);
        }

};

#endif

/*************************************************************************
simplematrix.h
*************************************************************************/
```

```
#ifndef SIMPLEMATRIX
#define SIMPLEMATRIX

#include <math.h>
#include <iostream>
using namespace std;
#include <fstream>

class SimpleMatrix
{
private:
    void Swap(double & a, double & b)
    {
        double tmp = a;
        a = b;
        b = tmp;
    }

    int RowWithLargestValueInColumn (int c, int minRow = 0)
    {
        int r = minRow;
        double val = fabs(m[minRow][c]);
        for (int i=minRow+1;i<h;i++)
            if (fabs(m[i][c]) > val)
            {
                val = fabs(m[i][c]);
                r = i;
            }
        return r;
    }

    void QuickCOUT()
    {
        for (int i=0;i<h;i++)
        {
            for (int j=0;j<w;j++)
                cout << m[i][j] << ", ";
            cout << "\n";
        }
        cout <<"\n";
    }

    void QuickPrint()
    {
```

```
        std::ofstream fout ("simplematrixOutput.txt", std::ios::out);

        for (int i=0;i<h;i++)
        {
            for (int j=0;j<w;j++)
                fout << m[i][j] << ", ";
            fout << "\n";
        }
        fout <<"\n";

        fout.close();
    }

public:
    double ** m;
    int w;
    int h;

    SimpleMatrix() : m(0), w(0), h(0)
    {
    }

    SimpleMatrix(int w, int h) : m(0), w(0), h(0)
    {
        SetDimensions(w,h);
    }

    SimpleMatrix(SimpleMatrix & v) : m(0), w(0), h(0)
    {
        *this = v;
    }

    ~SimpleMatrix()
    {
        Clear();
    }

    void Clear()
    {
        for (int i=0;i<h;i++)
            delete [] m[i];
        delete [] m;
        m=0;
        w = h = 0;
    }
```

```
void SetDimensions (int w, int h)
{
    Clear();
    this->h = h;
    this->w = w;
    m = new double *[h];
    for (int i=0;i<h;i++)
        m[i] = new double [w];
    for (int i=0;i<h;i++)
        for (int j=0;j<w;j++)
            m[i][j] = 0;
}

SimpleMatrix & operator = (SimpleMatrix & m2)
{
    SetDimensions(m2.w, m2.h);
    for (int i=0;i<h;i++)
        for (int j=0;j<w;j++)
            m[i][j] = m2.m[i][j];
    return *this;
}

void AppendMatrixRight(SimpleMatrix & v)
{
    SimpleMatrix tmp(w+v.w, h);
    for (int i=0;i<h;i++)
        for (int j=0;j<w;j++)
            tmp.m[i][j] = m[i][j];
    for (int i=0;i<h;i++)
        for (int j=0;j<v.w;j++)
            tmp.m[i][w+j] = v.m[i][j];
    *this = tmp;
}

void AppendMatrixBottom (SimpleMatrix & v)
{
    SimpleMatrix tmp(w, h+v.h);
    for (int i=0;i<h;i++)
        for (int j=0;j<w;j++)
            tmp.m[i][j] = m[i][j];
    for (int i=0;i<v.h;i++)
        for (int j=0;j<w;j++)
            tmp.m[h+i][j] = v.m[i][j];
    *this = tmp;
```

```
}

SimpleMatrix SolveGaussianAgumented()
{


    SimpleMatrix v(*this);

    //v.QuickCOUT();
    //v.QuickPrint();

    for (int i=0;i<v.h;i++)
    {
        int swap = v.RowWithLargestValueInColumn(i, i);
        v.SwapRows(i, swap);

        //v.QuickCOUT();

        for (int j=0;j<v.h;j++)
        {
            if (j==i) continue;
            double scaleby = v.m[j][i] / v.m[i][i];
            for (int k=0;k<v.w;k++)
                v.m[j][k] -= v.m[i][k] * scaleby;
            v.m[j][i] = 0;
        }

        //v.QuickCOUT();
    }

    // gather together the result
    SimpleMatrix retval(1, v.h);
    for (int i=0;i<v.h;i++)
        retval.m[i][0] = v.m[i][w-1] / v.m[i][i];

    //v.QuickCOUT();
    //v.QuickPrint();

    return retval;
}

void SwapRows (int r1, int r2)
{
    for (int i=0;i<w;i++)
        Swap(m[r1][i], m[r2][i]);
```

```
    }

};

#endif

/*************************************************************************
sorts.h
*************************************************************************/


#ifndef SORTS_H
#define SORTS_H


template <class x>
void TriQuickSort (x * data, int length, bool sortAscending = true, bool (*lessthan)(const
        x&, const x&) = 0, bool (*greaterthan)(const x&, const x&) = 0) {

    // This function does not sort the data, it sorts the indices.

    if (lessthan == 0 || greaterthan == 0) {
        TriQuickSort2(data, sortAscending, 0, length-1, 4);
        InsertionSort(data, sortAscending, 0, length-1);
    } else {
        TriQuickSort2(data, sortAscending, 0, length-1, lessthan, greaterthan, 4);
        InsertionSort(data, sortAscending, 0, length-1, lessthan, greaterthan);
    }

}

template <class x>
void TriQuickSort2 (x * data, bool sortDecending, int min, int max, int splitLength = 4) {

    if (max - min > splitLength) {

        int i = (max + min) / 2;

        if (data[min] > data[i]) SwapAny(data[min], data[i]);
        if (data[min] > data[max]) SwapAny(data[min], data[max]);
        if (data[i] > data[max]) SwapAny(data[i], data[max]);

        int j = max-1;
        SwapAny(data[i], data[j]);
        i = min;
```

```
        x itemp = data[j]; //indices[j];

        do {
            do {
                i++;
            } while (data[i] < itemp);

            do {
                j--;
            } while (data[j] > itemp);

            if (j < i) break;
            SwapAny(data[i], data[j]);

        } while (true);

        SwapAny(data[i], data[max-1]);

        TriQuickSort2(data, sortDecending, min, j, splitLength);
        TriQuickSort2(data, sortDecending, i+1, max, splitLength);

    }

}

template <class x>
void TriQuickSort2 (x * data, bool sortDecending, int min, int max, bool
        (*lessthan)(const x&, const x&), bool (*greaterthan)(const x&, const x&), int
        splitLength = 4) {

    if (max - min > splitLength) {

        int i = (max + min) / 2;

        if (greaterthan(data[min], data[i])) SwapAny(data[min], data[i]);
        if (greaterthan(data[min], data[max])) SwapAny(data[min], data[max]);
        if (greaterthan(data[i], data[max])) SwapAny(data[i], data[max]);

        int j = max-1;
        SwapAny(data[i], data[j]);
        i = min;
        x itemp = data[j]; //indices[j];

        do {
            do {
```

```
            i++;
        } while (lessthan(data[i], itemp));

        do {
            j--;
        } while (greaterthan(data[j], itemp));

        if (j < i) break;
        SwapAny(data[i], data[j]);

    } while (true);

    SwapAny(data[i], data[max-1]);

    TriQuickSort2(data, sortDecending, min, j, lessthan, greaterthan, splitLength);
    TriQuickSort2(data, sortDecending, i+1, max, lessthan, greaterthan, splitLength);

}

}



template <class x>
void InsertionSort (x * data, bool sortDecending, int min, int max) {

    x dtemp;
    int j;
    for (int i = min+1; i <= max; i++) {

        //itemp = data[i];
        dtemp = data[i];

        j = i;
        while (j > min) {
            if (data[j-1] <= dtemp) break;
            data[j] = data[j-1];
            j--;
        }

        data[j] = dtemp;

    }

}
```

```
template <class x>
void InsertionSort (x * data, bool sortDecending, int min, int max, bool (*lessthan)(const
        x&, const x&), bool (*greaterthan)(const x&, const x&)) {

    x dtemp;
    int j;
    for (int i = min+1; i <= max; i++) {

        //itemp = data[i];
        dtemp = data[i];

        j = i;
        while (j > min) {
            if (!greaterthan(data[j-1], dtemp)) break;
            data[j] = data[j-1];
            j--;
        }

        data[j] = dtemp;

    }

}


template <class x>
void TriQuickSortIndices (x * data, int * indices, int length, bool sortDecending = true) {

    // This function does not sort the data, it sorts the indices.

    TriQuickSortIndices2(data, indices, sortDecending, 0, length-1, 4);
    InsertionSortIndices(data, indices, sortDecending, 0, length-1);

}

template <class x>
void TriQuickSortIndices2 (x * data, int * indices, bool sortDecending, int min, int max,
        int splitLength = 4) {

    if (max - min > splitLength) {

        int i = (max + min) / 2;
```

```
        if (data[indices[min]] > data[indices[i]]) SwapAny(indices[min], indices[i]);
        if (data[indices[min]] > data[indices[max]]) SwapAny(indices[min],
        indices[max]);
        if (data[indices[i]] > data[indices[max]]) SwapAny(indices[i], indices[max]);

        int j = max-1;
        SwapAny(indices[i], indices[j]);
        i = min;
        x itemp = data[indices[j]]; //indices[j];

        do {
            do {
                i++;
            } while (data[indices[i]] < itemp);

            do {
                j--;
            } while (data[indices[j]] > itemp);

            if (j < i) break;
            SwapAny(indices[i], indices[j]);

        } while (true);

        SwapAny(indices[i], indices[max-1]);

        TriQuickSortIndices2(data, indices, sortDecending, min, j, splitLength);
        TriQuickSortIndices2(data, indices, sortDecending, i+1, max, splitLength);

    }

}


template <class some>
__inline void SwapAny (some & x, some & y) {
    static some temp;
    temp = x;
    x = y;
    y = temp;
}

template <class x>
void InsertionSortIndices (x * data, int * indices, bool sortDecending, int min, int max) {
```

```
    x dtemp;
    int j, itemp;
    for (int i = min+1; i <= max; i++) {

        itemp = indices[i];
        dtemp = data[itemp];

        j = i;
        while (j > min) {
            if (data[indices[j-1]] <= dtemp) break;
            indices[j] = indices[j-1];
            j--;
        }

        indices[j] = itemp;

    }

}

#endif

/*****************************************************************************
all_sparse_interpolators.h
*****************************************************************************/

#ifndef AlL_SPARSE_INTERPOLATORS_H
#define ALL_SPARSE_INTERPOLATORS_H

#include "nearestneighbor_interpolator.h"
#include "inversedistance_interpolator.h"
#include "spherical_interpolator.h"
//#include "visibility_interpolator.h"
#include "thinplatespline_interpolator.h"
#include "volumespline_interpolator.h"
#include "multiquadric_interpolator.h"
//#include "regional_interpolator.h"
#include "cubicspline_interpolator.h"
#include "polynomial_interpolator.h"
#include "average_interpolator.h"

#endif

/*****************************************************************************
```

```
sparse_interpolator.cpp
*****************************************************************************/

#ifndef SPARSE_INTERPOLATOR_H
#define SPARSE_INTERPOLATOR_H

#include <math.h>
#include "3Dgeometry.h"
#include "simplematrix.h"
#include "sorts.h"

class sparse_interpolator
{
protected:
    point3D *points;
    double *values;
    int pointCount;
    double *weights;
    bool dataHasChanged;

    void Clear()
    {
        delete[] points;
        delete[] values;
        delete[] weights;
        points = 0;
        values = 0;
        weights = 0;
        pointCount = 0;
    }

public:

    sparse_interpolator()
    {
        points = 0;
        values = 0;
        weights = 0;
        pointCount = 0;
        dataHasChanged = false;
    }

    char* InterpolationType();
    double InterpolatePoint(point3D);
```

```
void RegisterSparseData(point3D points[], double values[], int count)
{
    if (count != pointCount)
    {
        Clear();
        this->points = new point3D[count];
        this->values = new double[count];
        weights = new double[count];
    }

    pointCount = count;
    for (int i=0;i<pointCount;i++)
    {
        this->points[i] = points[i];
        this->values[i] = values[i];
    }
    NormalizeWeights();
    dataHasChanged = true;
}

void RegisterSparseData(double points[], double values[], int count)
{
    point3D *pts = new point3D[count];
    for (int i=0;i<count;i++)
        pts[i].x = points[i];
    RegisterSparseData(pts, values, count);
    delete[] pts;
}

point3D RandomPointInsideDataBox()
{
    point3D retval;
    if (pointCount < 1) return retval;

    retval.x = minX() + (maxX()-minX()) * (double)rand() / RAND_MAX;
    retval.y = minY() + (maxY()-minY()) * (double)rand() / RAND_MAX;
    retval.z = minZ() + (maxZ()-minZ()) * (double)rand() / RAND_MAX;

    return retval;
}

double minX ()
{
    if (pointCount < 1) return 0;
    double val = points[0].x;
```

```
    for (int i=1;i<pointCount;i++)
        if (points[i].x<val)
            val = points[i].x;
    return val;
}

double maxX ()
{
    if (pointCount < 1) return 0;
    double val = points[0].x;
    for (int i=1;i<pointCount;i++)
        if (points[i].x>val)
            val = points[i].x;
    return val;
}

double minY()
{
    if (pointCount < 1) return 0;
    double val = points[0].y;
    for (int i=1;i<pointCount;i++)
        if (points[i].y<val)
            val = points[i].y;
    return val;
}

double maxY ()
{
    if (pointCount < 1) return 0;
    double val = points[0].y;
    for (int i=1;i<pointCount;i++)
        if (points[i].y>val)
            val = points[i].y;
    return val;
}

double minZ ()
{
    if (pointCount < 1) return 0;
    double val = points[0].z;
    for (int i=1;i<pointCount;i++)
        if (points[i].z<val)
            val = points[i].z;
    return val;
}
```

```
double maxZ ()
{
    if (pointCount < 1) return 0;
    double val = points[0].z;
    for (int i=1;i<pointCount;i++)
        if (points[i].z>val)
            val = points[i].z;
    return val;
}

double minV ()
{
    if (pointCount < 1) return 0;
    double val = values[0];
    for (int i=1;i<pointCount;i++)
        if (values[i]<val)
            val = values[i];
    return val;
}

double maxV ()
{
    if (pointCount < 1) return 0;
    double val = values[0];
    for (int i=1;i<pointCount;i++)
        if (values[i]>val)
            val = values[i];
    return val;
}

bool InsideConvexHull_2D (point3D pt)
{
    if (pointCount < 3) return false;

    vector3D unitVector;
    unitVector.p.x = 1;
    double *angles = new double[pointCount];

    for (int i=0; i<pointCount; i++)
    {
        if (points[i] == pt) return true;
        angles[i] = vector3D::AngleBetweenVectors((points[i]-pt),unitVector);
        if ((points[i]-pt).y < 0) angles[i] = 2*3.1415926-angles[i];
    }
```

```
        sort(angles, pointCount);

        double largestSpan = 0;
        for (int i=1; i<pointCount; i++)
        {
            double span = angles[i]-angles[i-1];
            if (span > largestSpan)
                largestSpan = span;
        }
        double span = angles[0]-angles[pointCount-1]+2*3.1415926;
        if (span > largestSpan)
            largestSpan = span;

        delete[] angles;

        return largestSpan < 3.1415926;
    }

    void QuickPrint ()
    {
        for (int i=0;i<pointCount;i++)
            cout <<
                points[i].x << ", " <<
                points[i].y << ", " <<
                points[i].z << ", " <<
                values[i] << "\n";
        cout << "\n";
    }

protected:

    void sortPointsByDistance (point3D center = point3D(0,0,0))
    {
        double *distances = new double[pointCount];
        int *indices = new int[pointCount];

        for (int i=0;i<pointCount;i++)
        {
            distances[i] = (points[i]-center).DistanceFromOrigin();
            indices[i] = i;
        }

        TriQuickSortIndices(distances, indices, pointCount);
```

```
    point3D *newpts = new point3D[pointCount];
    double *newvals = new double[pointCount];
    for (int i=0;i<pointCount;i++)
    {
        newpts[i] = points[indices[i]];
        newvals[i] = values[indices[i]];
    }

    RegisterSparseData(newpts, newvals, pointCount);

    delete[] distances;
    delete[] indices;
    delete[] newpts;
    delete[] newvals;
}

void sortPoints (vector3D vectorNormal)
{
    double *distances = new double[pointCount];
    int *indices = new int[pointCount];

    vectorNormal.Normalize(); // just in case;

    for (int i=0;i<pointCount;i++)
    {
        distances[i] = vector3D::Project(points[i], vectorNormal).Length();
        indices[i] = i;
    }

    TriQuickSortIndices(distances, indices, pointCount);

    point3D *newpts = new point3D[pointCount];
    double *newvals = new double[pointCount];
    for (int i=0;i<pointCount;i++)
    {
        newpts[i] = points[indices[i]];
        newvals[i] = values[indices[i]];
    }

    RegisterSparseData(newpts, newvals, pointCount);

    delete[] distances;
    delete[] indices;
    delete[] newpts;
    delete[] newvals;
```

```
}

void sort (double *arr, int len)
{
    quicksort(arr, 0, len-1);
}

void quicksort(double *arr, int left, int right)
{
    int p;
    if(left>=right)
    return;
    p = partition(arr,left, right);

    quicksort(arr,left,p-1);
    quicksort(arr,p+1,right);
}

int partition(double *arr, int left, int right)
{
    int first=left, pivot=right--;
    while(left<=right)
    {
        while(arr[left]<arr[pivot])
            left++;
        while((right>=first)&&(arr[right]>=arr[pivot]))
            right--;
        if(left<right)
        {
            swap(arr[left],arr[right]);
            left++;
        }
    }
    if(left!=pivot)
    swap(arr[left],arr[pivot]);

    return left;
}

template <class x>
void swap(x &i, x &j)
{
    x temp=i;
    i=j;
    j=temp;
```

```
}

void MakeSimpleWeightDistribution()
{
    MakeSimpleWeightDistribution(weights);
}

void MakeSimpleWeightDistribution (double *w)
{
    for (int i=0; i<pointCount; i++)
        w[i] = 1.0/pointCount;
}

double ApplyWeights ()
{
    double interpolant = 0;
    for (int i=0; i < pointCount; i++)
        interpolant += weights[i]*values[i];
    return interpolant;
}

void NormalizeWeights()
{
    NormalizeWeights(weights);
}

void NormalizeWeights (double *w)
{
    // normalize the weights
    double sum = 0;
    for (int i=0; i < pointCount; i++) sum += w[i];
    for (int i=0; i < pointCount; i++) w[i] /= sum;
}

void MergeWeights (double *w2)
{
    for (int i=0; i < pointCount; i++)
        weights[i] *= w2[i];
    NormalizeWeights();
}

void CreateDistanceWeights (const point3D target, double distancePower, double*
    w)
{
    MakeSimpleWeightDistribution(w);
```

```
        if (pointCount < 1) return;

        if (distancePower == 0)
        {
            MakeSimpleWeightDistribution(w);
            return;
        }

        // resolve semantic issues
        if (distancePower > 0) distancePower = -distancePower;

        // save problems later on by checking to see if any known points are exactly the
        same as the target
        for (int i=0; i < pointCount; i++)
            if (target == points[i])
            {
                for (int j=0; j < pointCount; j++)
                    w[j] = 0;
                w[i] = 1.0;
                return;
            }

        // calc weights
        if (distancePower == 1.0)
        {
            for (int i=0; i < pointCount; i++)
                w[i] *= 1.0 / (target-points[i]).DistanceFromOrigin();
        }
        else
        {
            for (int i=0; i < pointCount; i++)
                w[i] *= pow((target-points[i]).DistanceFromOrigin(), distancePower);
        }

        // normalize the weights
        NormalizeWeights(w);
    }

};

#endif

/*****************************************************************************
average_interpolator.h
```

```
**********************************************************************/


#ifndef AVERAGE_INTERPOLATOR_H
#define AVERAGE_INTERPOLATOR_H

#include "sparse_interpolator.h"

class average_interpolator: public sparse_interpolator
{
private:


public:
    average_interpolator ()
    {
    }

    ~average_interpolator()
    {
        Clear();
    }

    char* InterpolationType()
    {
        return "Average";
    }

    double InterpolatePoint(point3D target)
    {
        return InterpolatePoint();
    }

    double InterpolatePoint()
    {
        if (pointCount < 1) return 0;
        double sum = 0;
        for (int i=0; i < pointCount; i++)
            sum += values[i];
        return sum/pointCount;
    }

};
```

```
#endif

/*************************************************************************
cubicspline_interpolator.h
*************************************************************************/


#ifndef CUBICSPLINE_INTERPOLATOR_H
#define CUBICSPLINE_INTERPOLATOR_H


#include "sparse_interpolator.h"

class cubicspline_interpolator: public sparse_interpolator
{
private:
    SimpleMatrix s;

public:
    cubicspline_interpolator ()
    {
    }

    ~cubicspline_interpolator()
    {
        Clear();
    }

    char* InterpolationType()
    {
        return "Cubic Spline";
    }

    double InterpolatePoint(point3D target)
    {
        return InterpolatePoint(target.x);
    }

    double InterpolatePoint(double target)
    {
        if (pointCount < 4) return 0;
        if (dataHasChanged) FindCoef();
        dataHasChanged = false;
        return ApplyCoef(target);
    }
```

```
private:

    double ApplyCoef (double target)
    {
        double x = target;
        double val = 0;

        for (int j=0;j<pointCount;j++)
            if (x == points[j].x)
                return values[j];

        for (int j=0;j<pointCount-1;j++)
        {
            if ((x > points[j].x && x < points[j+1].x) || (x < points[0].x && j == 0) || (x >
points[pointCount-1].x && j == pointCount-2))
            {
                double a = values[j];
                double b = s.m[j][0];
                double c = 3*(values[j+1]-values[j])-2*s.m[j][0]-s.m[j+1][0];
                double d = 2*(values[j]-values[j+1])+s.m[j][0]+s.m[j+1][0];
                double p = (x-points[j].x)/(points[j+1].x-points[j].x);
                val = a + b*p + c*p*p + d*p*p*p;
            }
        }

        return val;
    }

    void FindCoef()
    {
        sortPoints(vector3D(1,0,0));

        SimpleMatrix m (pointCount+1, pointCount);

        m.m[0][0] = 2;
        m.m[0][1] = 1;
        m.m[pointCount-1][pointCount-1] = 2;
        m.m[pointCount-1][pointCount-2] = 1;
        for (int i=1;i<pointCount-1;i++)
        {
            m.m[i][i-1] = 1;
            m.m[i][i] = 4;
            m.m[i][i+1] = 1;
        }
```

```
        m.m[0][pointCount] = 3*(values[1]-values[0]);
        m.m[pointCount-1][pointCount] = 3*(values[pointCount-1]-values[pointCount-
2]);
        for (int i=1;i<pointCount-1;i++)
            m.m[i][pointCount] = 3*(values[i+1]-values[i-1]);

        s = m.SolveGaussianAgumented();
    }

};


#endif

/*************************************************************************
inversedistance_interpolator.h
*************************************************************************/

#ifndef INVERSEDISTANCE_INTERPOLATOR_H
#define INVERSEDISTANCE_INTERPOLATOR_H

#include "sparse_interpolator.h"

class inversedistance_interpolator: public sparse_interpolator
{
private:
    void Clear()
    {
        sparse_interpolator::Clear();
    }

public:
    inversedistance_interpolator ()
    {
    }

    ~inversedistance_interpolator()
    {
        Clear();
    }

    char* InterpolationType()
    {
        return "Inverse Distance";
```

```
    }

    double InterpolatePoint(point3D target, double distancePower = 2)
    {
        if (pointCount < 1) return 0;

        CreateDistanceWeights(target, distancePower, weights);
        return ApplyWeights();
    }

};

#endif

/*************************************************************************
multiquadric_interpolator.h
*************************************************************************/

#ifndef MULTIQUADRIC_INTERPOLATOR_H
#define MULTIQUADRIC_INTERPOLATOR_H


#include "sparse_interpolator.h"
#include "simplematrix.h"
#include <math.h>


class multiquadric_interpolator: public sparse_interpolator
{
private:

    SimpleMatrix coef;
    double r_squared;

    void Clear()
    {
        sparse_interpolator::Clear();
    }

public:
    multiquadric_interpolator ()
    {
        r_squared = 1.0;
    }
```

```
    ~multiquadric_interpolator()
    {
        Clear();
    }

    char* InterpolationType()
    {
        return "Volume Spline";
    }

    double InterpolatePoint(point3D target, double r_squared = 1.0)
    {
        if (pointCount < 3) return 0;

        if (r_squared != this->r_squared) dataHasChanged = true;
        this->r_squared = r_squared;

        if (dataHasChanged) FindCoef();
        dataHasChanged = false;

        return ApplyCoef(target);
    }
private:

    double ApplyCoef(point3D target)
    {
        double retval = 0;
        for (int i=0;i<pointCount;i++)
            retval += coef.m[i][0] * kernelFunction((points[i]-
target).DistanceFromOrigin());
        return retval;
    }

    double kernelFunction(double d)
    {
        return sqrt(d*d + r_squared);
    }

    void FindCoef ()
    {
        SimpleMatrix x(pointCount+1,pointCount);
        for (int i=0;i<pointCount;i++)
            for (int j=i;j<pointCount;j++)
                x.m[i][j] = x.m[j][i] = kernelFunction((points[i]-
points[j]).DistanceFromOrigin());
```

```
        for (int i=0;i<pointCount;i++)
            x.m[i][pointCount] = values[i];

        coef = x.SolveGaussianAgumented();
    }

};

#endif

/*************************************************************************
nearestneighbor_interpolator.h
*************************************************************************/

#ifndef NEARESTNEIGHBOR_INTERPOLATOR_H
#define NEARESTNEIGHBOR_INTERPOLATOR_H

#include "sparse_interpolator.h"

class nearestneighbor_interpolator: public sparse_interpolator
{
private:


public:
    nearestneighbor_interpolator ()
    {
    }

    ~nearestneighbor_interpolator()
    {
        Clear();
    }

    char* InterpolationType()
    {
        return "Nearest Neighbor";
    }

    double InterpolatePoint(point3D target)
    {
        if (pointCount < 1) return 0;

        int closest = 0;
```

```
        double smallestDistance = (target-points[0]).DistanceFromOrigin();
        for (int i=1; i < pointCount; i++)
        {
            double dist = (target-points[i]).DistanceFromOrigin();
            if (dist < smallestDistance)
            {
                closest = i;
                smallestDistance = dist;
            }
        }

        return values[closest];
    }

};

#endif

/*************************************************************************
polynomial_interpolator.h
*************************************************************************/

#ifndef POLYNOMIAL_INTERPOLATOR_H
#define POLYNOMIAL_INTERPOLATOR_H

#include "sparse_interpolator.h"
#include "simplematrix.h"
#include <math.h>

class polynomial_interpolator: public sparse_interpolator
{
private:
    SimpleMatrix s;

public:
    polynomial_interpolator ()
    {
    }

    ~polynomial_interpolator()
    {
        Clear();
    }

    char* InterpolationType()
```

```
    {
        return "Polynomial";
    }

    double InterpolatePoint(point3D target)
    {
        return InterpolatePoint(target.x);
    }

    double InterpolatePoint(double target)
    {
        if (pointCount < 4) return 0;
        if (dataHasChanged) FindCoef();
        dataHasChanged = false;
        return ApplyCoef(target);
    }

private:

    double ApplyCoef (double target)
    {
        double sum = 0;
        double coef = 1;
        for (int j=0; j<s.h;j++)
        {
            sum += coef * s.m[j][0];
            coef *= target;
        }
        return sum;
    }

    void FindCoef()
    {
        SimpleMatrix m(pointCount+1,pointCount);
        for (int i=0;i<pointCount;i++)
        {
            for (int j=0;j<pointCount;j++)
                m.m[i][j] = pow(points[i].x, j);
            m.m[i][0] = 1;
            m.m[i][pointCount] = values[i];
        }

        s = m.SolveGaussianAgumented();
    }
```

```
};

#endif

/*************************************************************************
thinplatespline_interpolator.h
*************************************************************************/

#ifndef THINPLATESPLINE_INTERPOLATOR_H
#define THINPLATESPLINE_INTERPOLATOR_H

#include "sparse_interpolator.h"
#include "simplematrix.h"
#include <math.h>

class thinplatespline_interpolator: public sparse_interpolator
{
private:

    SimpleMatrix coef;

    void Clear()
    {
        sparse_interpolator::Clear();
    }

public:
    thinplatespline_interpolator ()
    {
    }

    ~thinplatespline_interpolator()
    {
        Clear();
    }

    char* InterpolationType()
    {
        return "Thin-Plate Spline";
    }

    double InterpolatePoint(point3D target, int use_N_closest_points = -1)
    {
        if (pointCount < 3) return 0;
        if (use_N_closest_points != -1 && use_N_closest_points < pointCount)
```

```
        {
            sortPointsByDistance(target);
            point3D *newpts = new point3D[use_N_closest_points];
            double *newvals = new double[use_N_closest_points];
            for (int i=0;i<use_N_closest_points;i++)
            {
                newpts[i] = points[i];
                newvals[i] = values[i];
            }
            thinplatespline_interpolator interp;
            interp.RegisterSparseData(newpts, newvals, use_N_closest_points);
            delete [] newpts;
            delete [] newvals;
            return interp.InterpolatePoint(target);
        }
        if (dataHasChanged) FindCoef();
        dataHasChanged = false;
        return ApplyCoef(target);
    }
private:

    double ApplyCoef(point3D target)
    {
        double retval = 0;
        retval += coef.m[pointCount][0];
        retval += coef.m[pointCount+1][0] * target.x;
        retval += coef.m[pointCount+2][0] * target.y;
        for (int i=0;i<pointCount;i++)
            retval += coef.m[i][0] * kernelFunction((points[i]-
target).DistanceFromOrigin());
        return retval;
    }

    double kernelFunction(double r)
    {
        if (r == 0) return 0;
        return r*r*log(r);
    }

    void FindCoef ()
    {
        SimpleMatrix x(pointCount+3+1,pointCount+3);
        for (int i=0;i<pointCount;i++)
            for (int j=i;j<pointCount;j++)
```

```
                x.m[i][j] = x.m[j][i] = kernelFunction((points[i]-
points[j]).DistanceFromOrigin());

        for (int i=0;i<pointCount;i++)
        {
            x.m[pointCount][i] = x.m[i][pointCount] = 1;
            x.m[pointCount+1][i] = x.m[i][pointCount+1] = points[i].x;
            x.m[pointCount+2][i] = x.m[i][pointCount+2] = points[i].y;
        }

        for (int i=0;i<pointCount;i++)
            x.m[i][pointCount+3] = values[i];

        coef = x.SolveGaussianAgumented();
    }

};

#endif

/*************************************************************************
main.cpp
*************************************************************************/

#ifndef VOLUMESPLINE_INTERPOLATOR_H
#define VOLUMESPLINE_INTERPOLATOR_H

#include "sparse_interpolator.h"
#include "simplematrix.h"
#include <math.h>


class volumespline_interpolator: public sparse_interpolator
{
private:

    SimpleMatrix coef;

    void Clear()
    {
        sparse_interpolator::Clear();
    }

public:
    volumespline_interpolator ()
```

```
    {
    }

    ~volumespline_interpolator()
    {
        Clear();
    }

    char* InterpolationType()
    {
        return "Volume Spline";
    }

    double InterpolatePoint(point3D target)
    {
        if (pointCount < 3) return 0;
        if (dataHasChanged) FindCoef();
        dataHasChanged = false;
        return ApplyCoef(target);
    }
private:

    double ApplyCoef(point3D target)
    {
        double retval = 0;
        retval += coef.m[pointCount][0];
        retval += coef.m[pointCount+1][0] * target.x;
        retval += coef.m[pointCount+2][0] * target.y;
        retval += coef.m[pointCount+3][0] * target.z;
        for (int i=0;i<pointCount;i++)
            retval += coef.m[i][0] * kernelFunction((points[i]-
target).DistanceFromOrigin());
        return retval;
    }

    double kernelFunction(double r)
    {
        return r*r*r;
    }

    void FindCoef ()
    {
        SimpleMatrix x(pointCount+4+1,pointCount+4);
        for (int i=0;i<pointCount;i++)
            for (int j=i;j<pointCount;j++)
```

```
                x.m[i][j] = x.m[j][i] = kernelFunction((points[i]-
points[j]).DistanceFromOrigin());

        for (int i=0;i<pointCount;i++)
        {
            x.m[pointCount][i] = x.m[i][pointCount] = 1;
            x.m[pointCount+1][i] = x.m[i][pointCount+1] = points[i].x;
            x.m[pointCount+2][i] = x.m[i][pointCount+2] = points[i].y;
            x.m[pointCount+3][i] = x.m[i][pointCount+3] = points[i].z;
        }

        for (int i=0;i<pointCount;i++)
            x.m[i][pointCount+4] = values[i];

        coef = x.SolveGaussianAgumented();
    }

};

#endif

/*************************************************************************
spherical_interpolator.cpp (A.K.A. MICROSPHERE PROJECTION)
*************************************************************************/

#ifndef SPHERICAL_INTERPOLATOR_H
#define SPHERICAL_INTERPOLATOR_H

#define RESOLUTION_2D 1000 //MUST BE DIVISBLE BY 4
#define RESOLUTION_3D 2000
#define PI 3.141592653
#define VERY_SMALL_WEIGHT 0.001

#include "sparse_interpolator.h"
#include <math.h>
#include "random.h"

class spherical_interpolator: public sparse_interpolator
{
private:
    double *applicationDensity;
    point3D *sphereLocations;

    void Clear()
    {
```

```
        delete[] applicationDensity;
        applicationDensity = 0;
        delete[] sphereLocations;
        sphereLocations = 0;
        sparse_interpolator::Clear();
    }

public:
    spherical_interpolator ()
    {
        applicationDensity = 0;
        sphereLocations = 0;
    }

    ~spherical_interpolator()
    {
        Clear();
    }

    char* InterpolationType()
    {
        return "Spherical";
    }

    double InterpolatePoint(point3D target, double distancePower = 2, bool is2D = false)
    {
        if (pointCount < 1) return 0;

        CreateWeights(target, distancePower, is2D);
        return ApplyWeights();
    }

private:

    void MakeApplicationDensity()
    {
        if (!applicationDensity)
        {
            applicationDensity = new double [RESOLUTION_2D/2];
            for (int i=0; i < RESOLUTION_2D/2; i++)
                applicationDensity[i] = sin(i*PI*2/RESOLUTION_2D);
        }
    }

    void Make3DSphereLocations()
```

```
    {
        if (!sphereLocations)
        {
            MTRand r;
            r.seed();
            sphereLocations = new point3D [RESOLUTION_3D];
            // set aside our random points on the sphere
            for (int i=0; i < RESOLUTION_3D; i++)
            {
                do
                {
                    sphereLocations[i].x = (2*r.rand())-1;
                    sphereLocations[i].y = (2*r.rand())-1;
                    sphereLocations[i].z = (2*r.rand())-1;
                } while (sphereLocations[i].DistanceFromOrigin() == 0 ||
sphereLocations[i].DistanceFromOrigin() > 1);
                sphereLocations[i] = ((vector3D)sphereLocations[i]).Normalize().p;
            }
        }
    }

    void CreateWeights2D(const point3D target, double distancePower)
    {
        MakeApplicationDensity();

        // allocate memory and zero out our knowledge of the circle
        static int *strongestIndex = new int [RESOLUTION_2D];
        static double strongestValue[RESOLUTION_2D];
        for (int i=0; i < RESOLUTION_2D; i++)
        {
            strongestIndex[i] = -1;
            strongestValue[i] = 0;
        }

        CreateDistanceWeights(target, distancePower, weights);

        vector3D origin(1,0,0); // origin vector (points along x axis)
        for (int i=0; i < pointCount; i++)
        {
            // this is the core multiplier.  if very small, just skip.
            if (weights[i] < VERY_SMALL_WEIGHT/pointCount) continue;

            double angle = vector3D::AngleBetweenVectors(points[i]-target, origin);
            if ((points[i]-target).y < 0) angle = 2*PI - angle; // ex: if angle = 90degrees &
y-value is negative, circular angle is 270degrees from the origin
```

```
            int baseIndex = (int)(angle/(2*PI) * RESOLUTION_2D); // where is the
center of this distribution

            // start at the leftmost position of the distribution.
            int start = baseIndex-RESOLUTION_2D/4;
            for (int j=0; j < RESOLUTION_2D/2; j++)
            {
                int index = (start+j+RESOLUTION_2D)%RESOLUTION_2D; // ensure
index is positive and no greater than resolution-1
                if (applicationDensity[j]*weights[i] > strongestValue[index])
                { // we have a new strongest value, replace previous one in this sector.
                    strongestIndex[index] = i;
                    strongestValue[index] = applicationDensity[j]*weights[i];
                }
            }
        }

        for (int i=0; i < pointCount; i++) weights[i] = 0;
        for (int j=0; j < RESOLUTION_2D; j++) weights[strongestIndex[j]] +=
strongestValue[j];

        sparse_interpolator::NormalizeWeights();
    }

    void CreateWeights3D(const point3D target, double distancePower)
    {
        Make3DSphereLocations();

        // allocate memory and zero out our knowledge of the circle
        static int *strongestIndex = new int [RESOLUTION_3D];
        static double *strongestValue = new double [RESOLUTION_3D];
        for (int i=0; i < RESOLUTION_3D; i++)
        {
            strongestIndex[i] = -1;
            strongestValue[i] = 0;
        }

        CreateDistanceWeights(target, distancePower, weights);

        for (int i=0; i < pointCount; i++)
        {
            // this is the core multiplier.  if super small, just skip.
            if (weights[i] < VERY_SMALL_WEIGHT/pointCount) continue;

            vector3D pointvector = points[i]-target;
```

```
            pointvector.Normalize();
            for (int j=0; j < RESOLUTION_3D; j++)
            {
                double cosangle = vector3D::CosAngleBetweenVectors(pointvector,
sphereLocations[j], 1, 1);
                if (cosangle > 0)
                { // if the data point is on the same side as the sphereLocation
                    if (cosangle*weights[i] > strongestValue[j])
                    { // we have a new strongest value, replace previous one in this sector.
                        strongestIndex[j] = i;
                        strongestValue[j] = cosangle*weights[i];
                    }
                }
            }
        }

        // final weight = % of circle covered-ish
        for (int i=0; i < pointCount; i++)
        {
            double count = 0;
            for (int j=0; j < RESOLUTION_3D; j++)
                if (strongestIndex[j] == i) count += strongestValue[j];
            weights[i] = count;
        }

        sparse_interpolator::NormalizeWeights();
    }

    void CreateWeights(const point3D target, double distancePower, bool is2D)
    {
        MakeSimpleWeightDistribution(weights);
        if (pointCount < 1) return;

        // save problems later on by checking to see if any known points are exactly the
same as the target
        for (int i=0; i < pointCount; i++)
            if (target == points[i])
            {
                for (int j=0; j < pointCount; j++)
                    weights[j] = 0;
                weights[i] = 1.0;
                return;
            }

        //this section conatins code which adresses the 'uniqueness' of the point
```

```
        //points which are more unique will have their relative visibility value increase
        //points which are closer to other known points will see their relative visibility
values decrease
        if (is2D)
            CreateWeights2D(target, distancePower);
        else
            CreateWeights3D(target, distancePower); // 3D. YIKES!
    }

};

#endif
```