A FAST LAYERED ALTERNATIVE TO KRIGING

by

Michael J. Thiesen

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

June 2007

APPROVAL

of a thesis submitted by

Michael J. Thiesen

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

Dr. Denbigh Starkey

Approved for the Department of Computer Science

Dr. John Paxton

Approved for the College of Graduate Studies

Dr. Carl Fox

STATEMENT OF PERMISSION TO USE

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Empirically gathered scientific data often comes in the form of scattered locations, each with an associated measurement. Visualizing scattered data is challenging because we need to estimate the measured values at many regularly spaced intervals in order to render the data to modern displays. Kriging is a common technique for visualizing scattered data that produces high quality output, but is often too slow for large data sets. In this thesis I present Layered Interpolation, an alternative to Kriging based on the idea of fitting fractal noise functions to scattered data. This technique produces output with quality that is comparable to Kriging, but with greatly reduced running time. Layered Interpolation's speed makes it an ideal choice for rendering large scattered data sets to modern high-resolution displays.

1. INTRODUCTION

Computer software is frequently used to display and analyze spatial data gathered in the field. This data usually comes in the form of a set of points, each with an x and y positional component, and one or more real valued measurements for that position. Often this data is not laid out in an organized grid, but instead scattered throughout the area being measured. An example of this would be data gathered from a farmer's field by means of moisture sensors and GPS equipment mounted to a tractor. The tractor moves through the field and the sensors record a new data point every few seconds. Unless the field is perfectly rectangular and flat, the tractor will have to follow the contours of the field and also work around obstacles. This generates a large amount of data, but it is inconveniently scattered throughout the field. In order to visualize the data, it is desirable to have measurements at regularly spaced grid points corresponding either to pixels on screen or in a texture map.

Many interpolation algorithms exist, but most of them are only suitable for input that is already regularly spaced. If the input is scattered, there are fewer choices. Furthermore, modern high-resolution video hardware requires a great many interpolations to be performed. For example, modern video cards can easily handle textures that are 2048×2048 texels in size, which would require roughly four million interpolations to fill. This prompted the design of an algorithm that can accept a large number of sample points and also perform a large number of interpolations efficiently.

My approach is inspired by fractal Perlin noise. Fractal Perlin noise combines multiple layers of a randomly generated noise function to mimic natural phenomena. Perlin noise is used throughout the computer graphics industry to generate realistic looking terrain [1], clouds [2], water surfaces [3], and other natural phenomena [4]. My algorithm is based on the idea that if randomly generated fractal noise can accurately mimic natural processes, perhaps fractal “noise” with carefully chosen parameters can be used to interpolate natural processes. As we will see, my technique produces pleasing results and scales well enough to display large data sets at high resolutions.

Problem Description

Sampling irregularly spaced data at regular grid intervals is an interpolation problem. We wish to make a reasonable estimate of an unknown function’s value at regular grid intervals based on known measurements. When evaluating an interpolation algorithm, there are several desirable qualities to look for. Perhaps the most obvious is accuracy. We would like our interpolated value to be as close as possible to the actual value of the function being measured, and also to be visually pleasing when rendered. Another important factor is speed. The speed of the interpolation is most dependent on two values: The number of sampled points, and the number of points to be interpolated. Throughout this paper, the number of sample points will be referred to as n, and the number of interpolation points will be referred to as m. Usually m is much larger than n due to the large number of interpolations needed for high-resolution displays. A somewhat less obvious quality is numeric stability. When operations are performed on real valued variables, some rounding error usually results. This is especially true for division by small numbers. Repeating an error-causing operation propagates the error through intermediate results and causes the total error to accumulate as the algorithm runs. Unstable algorithms may require the use of double precision (64 bit) floating point values to reduce rounding error, which essentialy doubles the memory consumed by the algorithm. Thus, we wish to minimize our use of operations that introduce error.

Notation and Terminology

Throughout this paper I will use notation that is consistent with existing literature whenever possible. Terms and variables commonly used throughout this paper are explained below.

n — The number of points sampled

m — The number of points to be interpolated

$\mathbf{x}_i$ — A vector representing the location of the i^{th} sampled data point, commonly a 2D coordinate

$\bar{\mathbf{X}}$ — The list of the sampled locations

$Z(\mathbf{x})$ — The unknown function or process that our sample points have been gathered from

z_i — A scalar representing the observed value of $Z(\mathbf{x}_i)$

$\bar{\mathbf{Z}}$ — The list of the sampled values

$\hat{\mathbf{x}}$ — A vector representing a location at which we wish to perform an interpolation

$\hat{x}, \hat{y}$ — The x and y coordinates for $\hat{\mathbf{x}}$

$\hat{Z}(\hat{\mathbf{x}})$ — An interpolation function used to estimate Z at location $\hat{\mathbf{x}}$

Field — Refers to the region over which interpolation will be performed; the range of $\mathbf{x}$ and $\hat{\mathbf{x}}$

2. CURRENT TECHNIQUES

In this chapter I will describe some of the existing interpolation methods that can accept scattered data. I will describe each method briefly, but still provide enough information to compare them against my Layered Interpolator. The references provide more details on each method. The curve fitting technique described in section 2 of this chapter is used as part of the Layered Interpolation algorithm. Being familiar with the terminology in that section will help make chapter 4 more clear.

Shepard's Method

Shepard's Method [5] is more commonly known as inverse distance weighting. Shepard's method is one of the simplest interpolation algorithms for irregular data. Given n data points sampled from an unknown function Z, we estimate the value of Z at point $\hat{\mathbf{x}}$ by taking a weighted average of the sampled values:

$$\hat{Z}(\hat{\mathbf{x}}) = \frac{\displaystyle\sum_{i=1}^{n} w_i(\hat{\mathbf{x}}) z_i}{\displaystyle\sum_{i=1}^{n} w_i(\hat{\mathbf{x}})} \tag{2.1}$$

The weights are computed by

$$w_i(\hat{\mathbf{x}}) = \frac{1}{d(\hat{\mathbf{x}}, \mathbf{x}_i)^k} \tag{2.2}$$

where d is a distance function and k is a user specified power parameter. The Euclidean distance is the most commonly used distance function, but other functions may also be used. The most commonly used value for k is 2, which simplifies the equation to:

$$w_i(\hat{\mathbf{x}}) = \frac{1}{\hat{\mathbf{x}} \cdot x_i} \tag{2.3}$$

where $\cdot$ denotes the dot product. Higher values of k bias the result toward nearby points, whereas lower values of k bias the result toward the global average. Shepard's method has the benefit of being easy to implement, and runs reasonably fast. For n data points and m interpolation points, the running time will be $O(mn)$. It is also fairly numerically stable, unless an interpolated point lies very near to one of the sampled points. It suffers in regard to accuracy however. For $k = 2$ the estimated values tend to be biased too much toward the mean, except near sampled points where the estimation function forms spikes (See Fig. 6(c) on page 31). Other values of k can be used, but this essentially requires the user to guess the proper value, or just choose a value that looks good. Values not equal to 2 also hurt performance since the more complicated weighting function (2.2) must be used.

Curve Fitting

Curve fitting is used to fit parameters of a known function to observed data points. If the form of the function Z is known or at least can be approximated, curve fitting is an appropriate technique. There are many curve fitting techniques, but I will briefly describe the linear least squares method because it is used in the Layered Interpolation algorithm. For more details, see [6].

Linear least squares can be used if the function $\hat{Z}$ can be expressed as a linear sum of basis functions:

$$\hat{Z}(\hat{x}) = c_1 f_1(\hat{x}) + c_2 f_2(\hat{x}) + \cdots + c_b f_b(\hat{x}) \tag{2.4}$$

where f_j are the basis functions, c_j are the coefficients that will be adjusted to fit $\hat{Z}$ to the data, and b is the number of basis functions. The basis functions themselves do not have to be linear, as long as $\hat{Z}$ can be expressed as a linear sum of the basis functions. Linear least squares works by adjusting the coefficients c_j such that the error is minimized. Let $\mathbf{c}$ be the vector of coefficients, and let

$$A = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \dots & f_b(x_1) \\ f_1(x_2) & f_2(x_2) & \dots & f_b(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_n) & f_2(x_n) & \dots & f_b(x_n) \end{pmatrix} \tag{2.5}$$

The error vector η is defined as

$$\eta_i = \hat{Z}(x_i) - Z(x_i) \tag{2.6}$$

or in matrix form:

$$\eta = A\mathbf{c} - \mathbf{z} \tag{2.7}$$

We wish to chose coefficients that minimize the sum of squares of the errors $\|\eta\|^2$ which can be computed by

$$\|\eta\|^2 = \|A\mathbf{c} - \mathbf{z}\|^2 = \sum_{i=1}^{n} \left(\sum_{j=1}^{b} a_{ij} c_j - z_i \right)^2 \tag{2.8}$$

The minimum of $\|\eta\|^2$ occurs where the gradient vector is zero. We find the gradient by taking the partial derivatives of $\|\eta\|^2$ with respect to each c_j. Setting the gradient to zero and solving for $\mathbf{c}$ yields

$$\mathbf{c} = (A^T A)^{-1}(A^T \mathbf{z}) \tag{2.9}$$

When implementing linear least squares curve fitting, it isn't necessary to actually store A. $A^T A$ and $A^T\mathbf{z}$ can be first set to zero, and then updated on the fly as each point is added. This reduces the memory consumption significantly since b is usually much smaller than n. A is an $n \times b$ matrix whereas $A^T A$ is only a $b \times b$ matrix. Each update costs $O(b^2)$ and inverting A costs $O(b^3)$. Once $\mathbf{c}$ is known, evaluating $\hat{Z}$ costs $O(b)$. This makes the total cost of curve fitting $O(nb^2 + b^3 + mb)$. When b is constant, the running time is simply $O(n+m)$. This makes linear least squares curve fitting one of the fastest interpolation techniques, but also one of the least flexible. It requires that the process responsible for Z be known, or at least that the process can be approximated by a function. Furthermore, this function must be expressible as a linear sum of basis functions. The layered approach described in chapter 4 lifts these restrictions.

Kriging

Kriging [7] is a popular family of algorithms for interpolating geographic data. Kriging was developed by Georges Matheron [8] based on the Master's thesis of Daniel

Krige [9]. Kriging algorithms calculate statistics for the sampled data points, and use these statistics to make predictions at unknown locations. While there is no strict definition for what classifies an algorithm as a Kriging algorithm, it is generally accepted that Kriging algorithms minimize the variance of the value predicted at an unknown location.

There are many Kriging algorithms [10], but only a few are commonly used. The mathematics behind these algorithms are very similar. Simple Kriging assumes that the mean value of the field is known. Ordinary Kriging is a somewhat more robust algorithm that does not require the mean value of the field to be known, but does assume that the mean is constant throughout the region being sampled. Universal Kriging assumes that the mean value can be approximated with a linear function. Ordinary Kriging is the most common form of Kriging, and is the algorithm I will summarize below.

The Variogram

Ordinary Kriging uses the variogram of the sampled data points to make predictions at unsampled locations. A variogram is a function that predicts the variance of the values measured at two locations. More formally:

$$2\gamma(x_1, x_2) = E((Z(\mathbf{x_1}) - Z(\mathbf{x_2}))^2) \tag{2.10}$$

where E is the expected value, and 2γ is the variogram. The function $\gamma(x_1, x_2)$ is known as the semi-variogram and is often more algebraically convenient than 2γ. If the variance for two points depends only on the distance between those points, then Z is isotropic. Because many natural processes are either isotropic or near isotropic, the implementation described below assumes isotropy. For an isotropic function, the variogram can be expressed as

$$2\gamma(h) = E((Z(x) - Z(x+h))^2) \tag{2.11}$$

where h is the distance between two points. More intuitively, this function tells us how much variance we can expect from two samples when given their distance apart. We can compute the variogram empirically from a given set of sampled points. For a given *approximate* distance h, the empirical variogram is defined as follows:

$$2\gamma^*(h) = \frac{1}{n(h)} \sum_{i,j \in N(h)} (z_i - z_j)^2 \tag{2.12}$$

Here, $2\gamma^*$ denotes an empirical variogram as opposed to a true variogram or theoretical variogram, and $n(h)$ represents the number of points approximately distance h apart. The term $i, j \in N(h)$ denotes the set of every pair of points whose distance is approximately h. The approximate distance is not strictly defined. Usually points are included when their distance is within a preset tolerance of h. In practice, the empirical variogram tends to be a fairly noisy, uneven function (see fig. 1). Using the empirical variogram to perform Kriging will usually produce similarly noisy and jagged results. Furthermore, even with a large number of samples sometimes there is not enough data to estimate the variogram for every desired value of h. Due to these undesirable properties of the empirical variogram, a theoretical variogram is fit

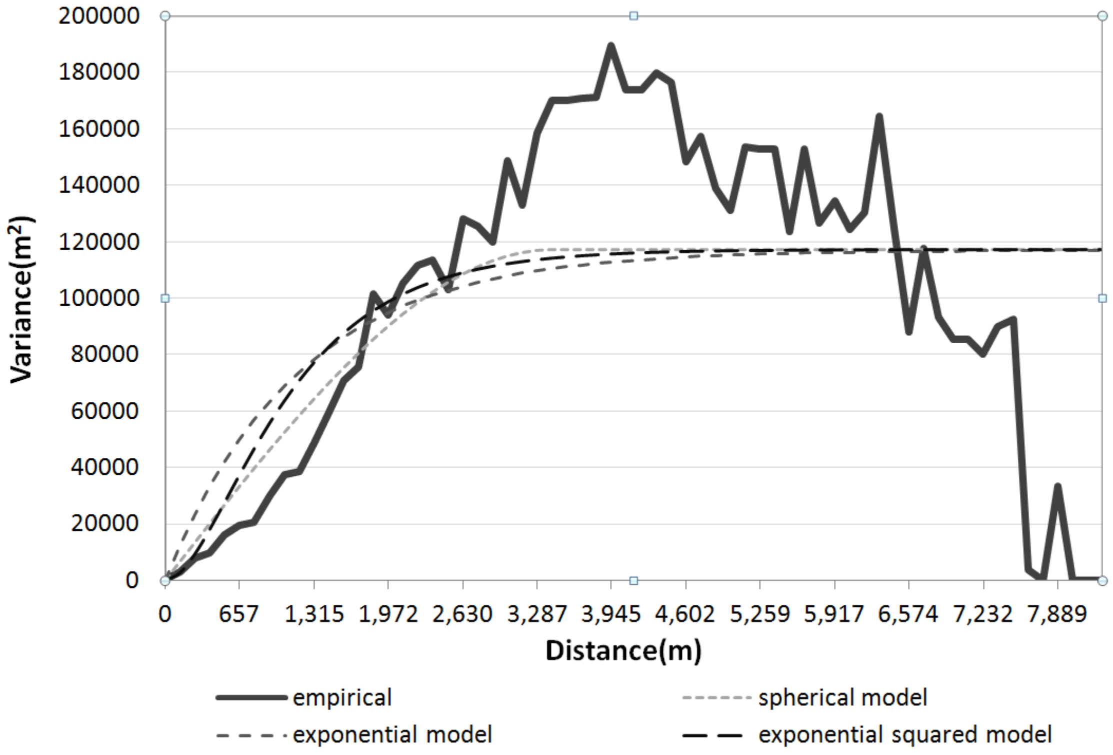


Figure 1. Variogram Example.

to the empirical data and used in the Kriging computations instead. Just about any continuous function that approximates the empirical variogram can be used. Some common model functions are listed below.

Spherical Model:

$$\gamma(h) = C(\frac{3h}{2a} - \frac{h^3}{2a^3}) \qquad \text{For } h < a$$

$$= C \qquad \text{For } h \geq a$$

Exponential Model:

$$\gamma(h) = C(1 - e^{-h/a})$$

Exponential Squared Model:

$$\gamma(h) = C(1 - e^{-h/a})^2$$

These models are based on the theoretical properties of the variogram. The values C and a are user specified parameters that are chosen such that the model variogram closely matches the empirically observed variogram. Curve fitting techniques are sometimes used for this, or the values may instead be chosen by hand.

Kriging Equations

Armed with a reasonable estimate of the variogram function, we are ready to begin the Kriging process. Our estimation function $\hat{Z}$ will be a weighted sum of the observed values:

$$\hat{Z}(\hat{\mathbf{x}}) = \sum_{i=1}^{n} w_i(\hat{\mathbf{x}}) Z(\mathbf{x}_i) \tag{2.13}$$

At a given location $\hat{\mathbf{x}}$, we wish to choose a value $\hat{z}$ such that the mean error is zero, and the variance (Kriging error) is as small as possible. For each location to be interpolated, the optimal weights must be recalculated, thus $w_i(\hat{\mathbf{x}})$ denotes the i^{th} Kriging weight for point $\hat{\mathbf{x}}$. The Kriging error is calculated as follows:

$$var(\hat{Z}(x_0) - Z(x_0)) = \sum_{i=0}^{n} \sum_{j=0}^{n} w_i(x_0) w_j(x_0) \gamma(x_i, x_j) \tag{2.14}$$

Here, we set $x_0 \equiv \hat{\mathbf{x}}$, which greatly simplifies the summations. We will choose weights which minimize this equation, subject to the following unbiasedness condition:

$$\sum_{i=1}^{n} w_i(\hat{\mathbf{x}}) = 1 \tag{2.15}$$

Requiring that the weights sum to one makes the predicted value a weighted average of the observed values and satisfies the condition that $E(\hat{Z}(\hat{\mathbf{x}}) - Z(\hat{\mathbf{x}})) = 0$. Finding the optimal Kriging weights is very similar to finding optimal basis function weights in curve fitting. The minimum variance occurs at the location where the gradient with respect to the weights is zero. Taking the partial derivative of equation (2.14) gives us the gradient vector. Setting the partial derivatives to zero will allow us to solve for the optimal weights. The resulting equations are best expressed in matrix form:

$$\begin{pmatrix} w_1 \\ \vdots \\ w_n \\ \lambda \end{pmatrix} = \begin{pmatrix} \gamma(\mathbf{x_1}, \mathbf{x_1}) & \dots & \gamma(\mathbf{x_1}, \mathbf{x_n}) & 1 \\ \vdots & \ddots & \vdots & \vdots \\ \gamma(\mathbf{x_n}, \mathbf{x_1}) & \dots & \gamma(\mathbf{x_n}, \mathbf{x_n}) & 1 \\ 1 & \dots & 1 & 0 \end{pmatrix}^{-1} \begin{pmatrix} \gamma(\mathbf{x_1}, \hat{\mathbf{x}}) \\ \vdots \\ \gamma(\mathbf{x_n}, \hat{\mathbf{x}}) \\ 1 \end{pmatrix} \tag{2.16}$$

$$\mathbf{w} = A^{-1}\mathbf{b} \tag{2.17}$$

Once the optimal weights are known, $\hat{z}$ is easily calculated using equation (2.13). The final row and column in A exists to satisfy the unbiasedness condition. The value λ is a Lagrange multiplier and also is used to satisfy the unbiasedness condition, but is not used for calculating the predicted value $\hat{z}$. For each new location $\hat{\mathbf{x}}$, the weights must be recalculated. However, A^{-1} is constant throughout the field and only needs to be computed once.

As we will see in chapter 5, Kriging makes good predictions compared to the other algorithms tested, but it is also the slowest algorithm tested. The time complexity for ordinary Kriging is $O(n^3 + mn^2)$. The n^3 term comes from the inversion of A, and the mn^2 term results from the matrix-vector multiplication required for each interpolated point. In most cases the mn^2 term dominates since m is usually much greater than n. Faster inversion algorithms such as Strassen's [11] or Coppersmith-Winograd [12] can be used to reduce the n^3 term for very large values of n, but this usually does not improve performance significantly since mn^2 dominates. Kriging also suffers from stability problems as n grows. As matrix A gets larger, more division operations are required to invert it. This can lead to substantial rounding errors, especially if 32 bit floating point numbers are used instead of 64 bit double precision numbers.

Thiessen Method

The Thiessen method [13] is another technique often used in the geosciences to interpolate spatial data. This technique was developed by American meteorologist Alfred H. Thiessen (not related to the author of this thesis) to interpolate meteorological measurements. The Thiessen method is a simple nearest neighbor technique. For an interpolation point $\hat{\mathbf{x}}$, set the interpolated value $\hat{z}$ equal to the known value sampled at the nearest location in $\bar{\mathbf{X}}$.

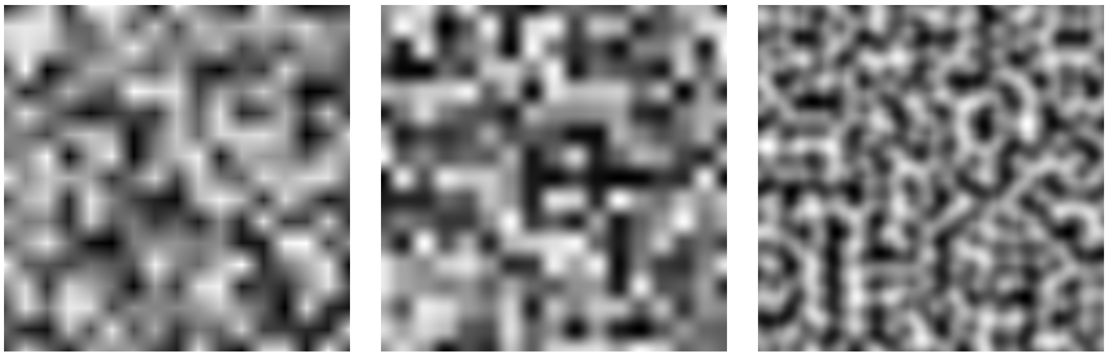
While the description of the Thiessen method is simple, a fast implementation is somewhat more complicated. For each point $\mathbf{x}_i$ in $\bar{\mathbf{X}}$, there is a surrounding region for which every point in that region is closer to $\mathbf{x}_i$ than any other point. These regions are a Voronoi Tessellation. When used in the geosciences, these tessellations are often known as Thiessen Polygons. For n input points, the Thiessen Polygons can be computed in $O(n \lg n)$ time [14]. Once the Thiessen Polygons have been computed, it takes $O(\lg n)$ time to find the sample point nearest to some given interpolation point $\hat{\mathbf{x}}$ [15]. Thus, for n sample points and m interpolation points, the total running time for the Thiessen method will be $O(n \lg n + m \lg n)$. A good implementation of the Thiessen method is very fast, but the resulting interpolation surface is not continuous. This makes the Thiessen method a good choice for analyzing data, but not an ideal choice for visualizing data.

3. FRACTAL NOISE

Fractal Noise is a noise generation technique widely used to mimic natural phenomena, and also to add artificial detail to digital renderings. Although Fractal Noise is not an interpolation algorithm, the concept is closely related to the Layered Interpolator described in chapter 4. Fractal noise is comprised of a repeated sum of some base noise function $\hat{Z}(\hat{\mathbf{x}})$. Even though noise functions aren't interpolation functions, I will use the same variable names for clarity. $\hat{\mathbf{x}}$ may be a scalar, but is more often a two or three dimensional vector. There are several choices for base functions. Generally, it is desirable for the base function to be continuous.

Base Noise Functions

Because we want the base noise function to be continuous, it should be defined for all real values of $\hat{\mathbf{x}}$. One simple way to create continuous noise is to randomly generate a value for all integer values of $\hat{\mathbf{x}}$. For one dimensional noise, this will be a regularly spaced array of random values. For two dimensional noise, this will take the form of a grid of random values. In order to evaluate the noise function for non-integer values of $\hat{\mathbf{x}}$, we simply perform a linear interpolation of the nearest integer values of $\hat{\mathbf{x}}$ (see fig. 2a). Higher quality noise can be produced by using cubic weighted interpolation instead of linear interpolation (see fig. 2b).



(a) 2D Linear Noise (b) 2D Cubic Noise (c) 2D Perlin Noise

Figure 2. Comparison of noise functions.

The noise functions described above tend to be unnatural looking. This is because their values are defined at regular intervals, and few processes in nature result from values defined at regular intervals. Perlin noise [3] produces a much more natural looking noise (see fig. 2c), and is a popular choice for graphical applications because it can be computed quickly.

Perlin Noise randomly generates its slope at regular intervals instead of defining its value at regular intervals. Let $\mathbf{i}$ be an integer valued vector, and let $\mathbf{g_i}$ be a randomly generated, real valued gradient vector for $\mathbf{i}$. For one dimensional Perlin noise, $\mathbf{g_i}$ will be a scalar, but for n dimensional noise, $\mathbf{g_i}$ will be an n dimensional vector. Each gradient vector $\mathbf{g_i}$ can be used to define a plane that passes through $\mathbf{i}$. The equation for this plane is

$$G(\mathbf{i}, \hat{\mathbf{x}}) = \mathbf{g_i} \cdot (\hat{\mathbf{x}} - \mathbf{i}) \tag{3.1}$$

To evaluate the noise function for a real valued position $\hat{\mathbf{x}}$, we sample the gradient function $G(\mathbf{i}, \hat{\mathbf{x}})$ for each neighboring integer grid point. We then use a weighted sum to blend these values into the final result (see fig. 3). For one dimensional Perlin noise, $\hat{\mathbf{x}}$ is a scalar and only has two neighboring gradients, $\mathbf{g}_{\lfloor \hat{x} \rfloor}$ and $\mathbf{g}_{\lceil \hat{x} \rceil}$. Two dimensional noise has four neighboring gradients, $\mathbf{g}_{(\lfloor \hat{x} \rfloor, \lfloor \hat{y} \rfloor)}$, $\mathbf{g}_{(\lceil \hat{x} \rceil, \lfloor \hat{y} \rfloor)}$, $\mathbf{g}_{(\lfloor \hat{x} \rfloor, \lceil \hat{y} \rceil)}$ and $\mathbf{g}_{(\lceil \hat{x} \rceil, \lceil \hat{y} \rceil)}$.

To perform the weighted sum, we define a cubic blending function $\alpha(v)$, where v is a scalar.

$$\alpha(v) = 2(v - \lfloor v \rfloor)^3 - 3(v - \lfloor v \rfloor)^2 + 1 \tag{3.2}$$

The alpha function is then used to blend the gradient planes into the final result. The one dimensional version is simple:

$$\hat{Z}(\hat{\mathbf{x}}) = (1 - \alpha(\hat{x}))G(\lfloor \hat{x} \rfloor, \hat{\mathbf{x}}) + \alpha(\hat{x})G(\lceil \hat{x} \rceil, \hat{\mathbf{x}}) \tag{3.3}$$

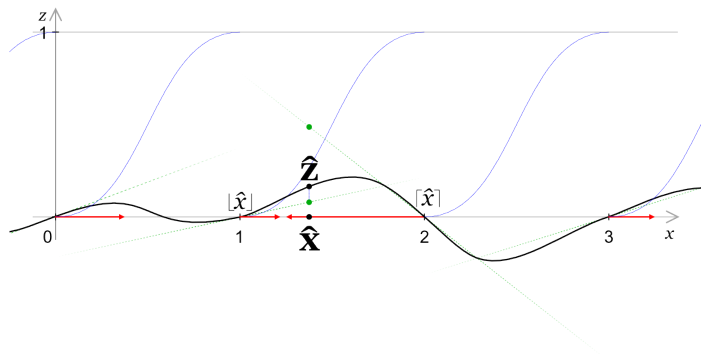
The two dimensional version first computes two intermediate values along the x axis and then blends those together along the y axis for the final result:

$$v_0(\hat{\mathbf{x}}) = (1 - \alpha(\hat{x}))G(\,[\lfloor \hat{x} \rfloor, \lfloor \hat{y} \rfloor]\,, \hat{\mathbf{x}}) + \alpha(\hat{x})G(\,[\lceil \hat{x} \rceil, \lfloor \hat{y} \rfloor]\,, \hat{\mathbf{x}}) \tag{3.4}$$

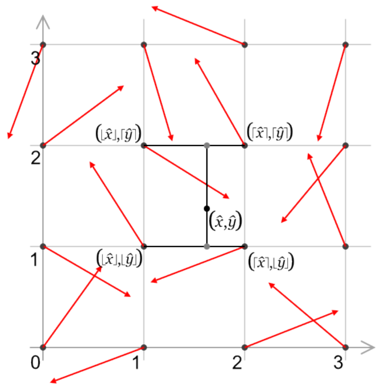
$$v_1(\hat{\mathbf{x}}) = (1 - \alpha(\hat{x}))G(\,[\lfloor \hat{x} \rfloor, \lceil \hat{y} \rceil]\,, \hat{\mathbf{x}}) + \alpha(\hat{x})G(\,[\lceil \hat{x} \rceil, \lceil \hat{y} \rceil]\,, \hat{\mathbf{x}}) \tag{3.5}$$

$$\hat{Z}(\hat{\mathbf{x}}) = (1 - \alpha(\hat{y}))v_0(\hat{\mathbf{x}}) + \alpha(\hat{y})v_1(\hat{\mathbf{x}}) \tag{3.6}$$

Here $\hat{x}$ and $\hat{y}$ are the scalar components of $\hat{\mathbf{x}}$, and $[x, y]$ denotes a 2D vector with the given x, y coordinates. The three dimensional version requires four intermediate results blended along the x axis, and then two more blended along the y axis. The



(a) 1D Perlin Noise


3
2
1
0
1
2
3
$(\lfloor\hat{x}\rfloor,\lceil\hat{y}\rceil)$
$(\lceil\hat{x}\rceil,\lceil\hat{y}\rceil)$
$(\hat{x},\hat{y})$
$(\lfloor\hat{x}\rfloor,\lfloor\hat{y}\rfloor)$
$(\lceil\hat{x}\rceil,\lfloor\hat{y}\rfloor)$

(b) 2D Perlin Noise

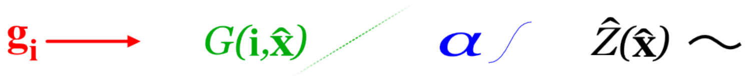
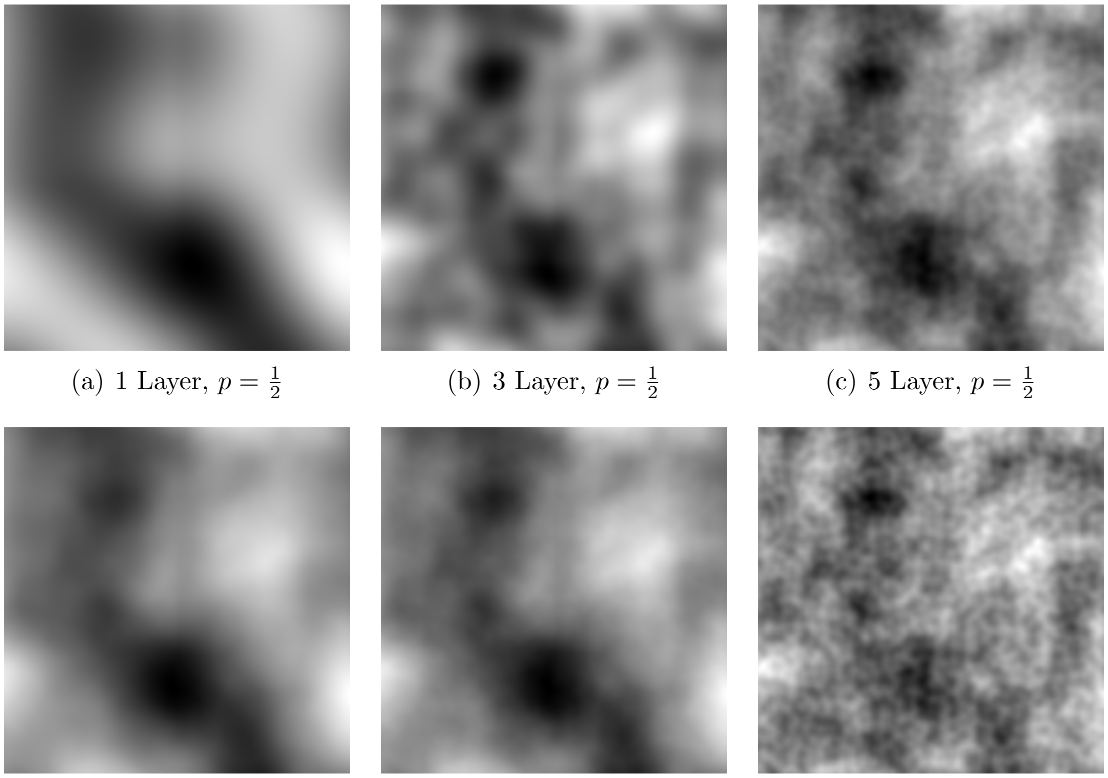


Figure 3. Perlin Noise with gradient vectors and gradient planes displayed.

final value is blended along the z axis. Higher dimensional versions continue in this fashion.

Producing Fractal Noise

In order to generate fractal noise, multiple layers of a noise function are added together with the frequency and amplitude varying for each layer. We can alter the frequency and amplitude of the noise simply by scaling the input and output of the noise function $\hat{Z}(\mathbf{x})$. For example, $\frac{1}{3}\hat{Z}(2\mathbf{x})$ would have double the frequency and one third the amplitude of $\hat{Z}(\mathbf{x})$. The first layer l_0 is computed with a user defined starting frequency f_0 and amplitude a_0. Each additional layer l_n will have double the frequency: $f_n = 2f_{n-1}$, and a user specified ratio of the previous amplitude: $a_n = pa_{n-1}$. Each layer adds detail at smaller scales, while preserving the existing



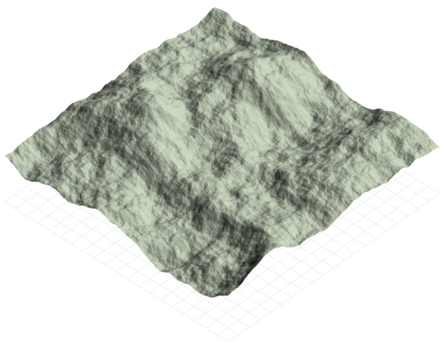
(a) 1 Layer, $p = \frac{1}{2}$ (b) 3 Layer, $p = \frac{1}{2}$ (c) 5 Layer, $p = \frac{1}{2}$

(d) 5 Layer, $p = \frac{1}{4}$ (e) 5 Layer, $p = \frac{1}{3}$ (f) 5 Layer, $p = \frac{2}{3}$

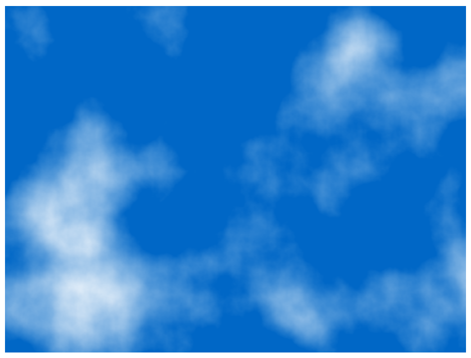
Figure 4. Two Dimensional fractal Perlin noise with varying parameters. Brighter regions correspond to higher noise values.

larger scaled structure. The ratio p is known as the persistence. $\frac{1}{2}$ is the most common persistence value, but any value between 0 and 1 will work. A lower persistence will result in smoother noise with large details exaggerated and fine details muted. A higher persistence causes smaller scales to be more pronounced and results in more jagged noise (see fig. 4).

Using a combination of noise with differing scales creates a final function that has distinct structures at every level of scale (see fig. 4). Multi-layered noise possesses statistical self-similarity, meaning that statistics gathered at a large scale will be



(a) Perlin Terrain



(b) Perlin Clouds

Figure 5. Fractal Perlin noise used to mimic natural phenomena.

similar to statistics gathered at a small scale. This property qualifies the layered noise function as a fractal function. Fractal noise is very useful for mimicking natural phenomena in graphics applications. For example, mapping the output of 2D noise to elevation can be used to produce natural looking terrain. Mapping the output to the opacity of a texture produces a cloud-like effect (see fig. 5).

4. LAYERED INTERPOLATOR

My algorithm combines curve fitting and the Perlin noise function to interpolate irregular data. Like Perlin noise, the interpolation function is composed of multiple layers, each having twice the frequency of the previous layer. Unlike Perlin noise, the parameters for each layer are chosen in such a way that the difference between the interpolated values and the observed values is minimized.

Layered Approach

The final interpolation function $\hat{Z}$ is the sum of the interpolation function for each layer L_i. Each layer is composed of a regularly spaced grid of interpolation nodes. The spacing between these nodes will be referred to as d, and its inverse is analogous to the frequency. Each node defines a surface, much like the gradient function G in Perlin noise. Similar to fractal noise, the distance between each node is halved for every successive layer.

Algorithm 1 Perform Layered Interpolation

```
1: function INTERPOLATE($\bar{\mathbf{L}}, \hat{\mathbf{x}}$)      ▷ Evaluates the list of layers $\bar{\mathbf{L}}$ at $\hat{\mathbf{x}}$
2:     $sum \leftarrow 0$
3:     for $i = 1..$COUNT($\bar{\mathbf{L}}$) do      ▷ Loop through each layer
4:         $sum \leftarrow sum +$ EVALLAYER($L_i, \hat{\mathbf{x}}$)
5:     end for
6:     return $sum$
7: end function
```

EvalLayer is described in more detail in the next section. The first layer is created with a large, user defined starting distance for d. With a large distance, this layer will provide only a rough fit for the existing data points. The error list $\bar{\mathbf{E}}$ for the layer has an error value e_j for each observed value z_j. e_j is the difference between the observed value and the Layered Interpolator. The next layer will have twice as many nodes, and therefore provide a closer fit. Rather than fit the next layer to the observed values z, we will fit it to the error from the previous layer, and then re-compute the error after adding the new layer. As more layers are added, the overall error will rapidly converge to zero.

Algorithm 2 Create Layers

```
1: function CreateLayers($\bar{\mathbf{X}}, \bar{\mathbf{Z}}$)      ▷ The sample locations, and sample values
2:     $n \leftarrow$ Count($\bar{\mathbf{X}}$)
3:     $lcount \leftarrow \lg(\mathbf{DETAIL} * n)$
4:     $\bar{\mathbf{L}} \leftarrow$ new LayerList[1..$lcount$]      ▷ Create a list of $lcount$ layers
5:     $d \leftarrow$ STARTDIST
6:     $\bar{\mathbf{E}} \leftarrow \bar{\mathbf{Z}}$      ▷ Initialize the error list
7:     for $i = 1..lcount$ do
8:         $L_i \leftarrow$ ComputeLayer($d, \bar{\mathbf{X}}, \bar{\mathbf{E}}$)      ▷ Compute each layer
9:         for $j = 1..n$ do
10:            $e_j \leftarrow e_j -$ EvalLayer($L_i, \mathbf{x}_j$)      ▷ Update the error list
11:        end for
12:        $d \leftarrow d/2$      ▷ Increase detail for next layer
13:    end for
14:    return $\bar{\mathbf{L}}$      ▷ Return the list of layers
15: end function
```

The ComputeLayer function returns a layer that has been fit to the given error points, and is described in algorithm 4. At first it may seem that only a single high resolution layer is needed and that multiple stacked layers are a waste of computation time. As we will see, the high detail layers make poor predictions in regions where there are few sample points. A sum of multiple layers of increasing detail makes better overall predictions than a single high-detail layer.

Evaluating a Layer

A layer is composed from a set of interpolation nodes arranged in a regular grid. Each node η has a position $\mathbf{x}_\eta$, A curve fitter F_η, and a surface function S_η. The surface function is similar to the gradient functions in Perlin noise. Perlin noise only defines the gradient (slope) for each surface, but the Layered Interpolator can use more complicated functions to achieve a better fit. When evaluating a layer at $\hat{\mathbf{x}}$, we will evaluate the surface function for each node neighboring $\hat{\mathbf{x}}$, and then interpolate the results.

Algorithm 3 Evaluate Layer

```
1: function EvalLayer($L, \hat{\mathbf{x}}$)                      ▷ Evaluates layer $L$ at location $\hat{\mathbf{x}}$
2:     $d \leftarrow$ NodeSpacing($L$)
3:     $sum \leftarrow 0$
4:     for all $\eta \in$ Neighbors($L, \hat{\mathbf{x}}$) do      ▷ Iterate through nodes neighboring $\hat{\mathbf{x}}$
5:         $sum \leftarrow sum + w(\eta, \hat{\mathbf{x}}, d) * S_\eta(\hat{\mathbf{x}})$      ▷ Compute weighted sum
6:     end for
7:     return $sum$                                       ▷ Return the list of layers
8: end function
```

In the above code, w is a weighting function used to blend the surface values. For my implementation, I chose a cubic weighting function similar to that found in

Perlin Noise. It is based on an s curve:

$$\begin{aligned} s(x) &= 1 - 3|x|^2 + 2|x|^3 && \text{For } |x| < 1 \\ &= 0 && \text{For } |x| \geq 1 \end{aligned}$$

One Dimensional:

$$w(\eta, \mathbf{x}, d) = s\left(\frac{|x - x_\eta|}{d}\right) \tag{4.1}$$

Two Dimensional:

$$w(\eta, \mathbf{x}, d) = s\left(\frac{|x - x_\eta|}{d}\right) \cdot s\left(\frac{|y - y_\eta|}{d}\right) \tag{4.2}$$

Here $\cdot$ is multiplication. If any of the components of $\mathbf{x}$ are greater than d, the weight will be zero. This weighting system allows each individual node to completely ignore most of the sample points. I refer to the region around a node where the weight is non-zero as the node's neighborhood. The NEIGHBORS function returns the set of nodes that have $\hat{\mathbf{x}}$ in their neighborhood. The size of this set is $2^{\mathbb{D}}$, where $\mathbb{D}$ is the dimension of the sample space. As a result of this exponentially growing neighbor set, this system becomes impractical for high dimensions. Interpolation is most commonly performed in two or three dimensions, so this isn't usually a problem. See chapter 6 for possible alternatives that are practical in higher dimensions.

Fitting a Layer

To fit a layer, the surface functions S_η of each node will be a localized approximation of the error left over from the last layer. The algorithm moves through each error point and adds it to every neighboring node. Then the curve fitter F_η is used to compute the surface function. The curve fitter F_η is used to adjust the parameters of the surface function such that the surface approximates neighboring error points. F_η could represent any curve fitting algorithm, but for my implementation I used the linear least squares technique with some small modifications described below.

Algorithm 4 Compute Layer

```
1:  function ComputeLayer($d, \bar{\mathbf{X}}, \bar{\mathbf{E}}$)        ▷ Returns a layer fit to the error $\bar{\mathbf{E}}$
2:      $L \leftarrow$ InitLayer($d$)                         ▷ Create layer $L$ and fill with nodes
3:      $n \leftarrow$ Count($\bar{\mathbf{X}}$)
4:      for $i = 1..n$ do
5:          for all $\eta \in$ Neighbors($L, \mathbf{x}_i$) do ▷ Iterate through nodes neighboring $\mathbf{x}_i$
6:              AddPoint($F_\eta, \mathbf{x}_i, e_i, w(\eta, \mathbf{x}_i, d)$)
7:          end for
8:      end for
9:      for all $\eta \in$ Nodes($L$) do                     ▷ Iterate through every node in $L$
10:         ComputeFit($F_\eta, S_\eta$)                         ▷ Use curve fitting to fit $S_\eta$
11:     end for
12:     return L
13: end function
```

InitLayer creates the nodes for the layer and lays out their positions $\mathbf{x}_\eta$ in a grid with spacing d. Its implementation will depend on the dimension of the input. The AddPoint($F, \mathbf{x}, z, w$) procedure adds the location $\mathbf{x}$ and value z to the curve fitter F, with a weighted importance of w. I used a variant of the linear least squares curve fitter that accepts a weight, or importance value for each of the input points. This is used to make the curve fitter to focus more on fitting the nearest points, whereas points near the edge of the neighborhood will mostly be the concern of adjacent nodes. ComputeFit(F, S) uses the curve fitter F to produce the surface function S. A plane is a straightforward choice for a surface function, although I also experimented with quadratic surface functions. The two dimensional versions of these functions are:

$$S(\mathbf{x}) = c_1 + c_2x + c_3y \qquad \text{Linear Basis, } b = 3$$

$$S(\mathbf{x}) = c_1 + c_2x + c_3y + c_4xy + c_5x^2 + c_6y^2 \qquad \text{Quadratic Basis, } b = 6$$

As we will see in chapter 5, these surface functions produce promising results. It is probably possible to further improve the results with more advanced surface functions and curve fitting techniques. This is an area for further research.

When using the linear least squares curve fitting method, a problem arises when there are very few points within the neighborhood of a node. If there are fewer points than basis functions, the matrix A^TA will be under-determined (see the Curve Fitting section of chapter 2). Even if the number of points and basis functions are equal, the curve fitting algorithm may over-fit the curve and produce a node with an extremely steep slope. Such slopes look like spikes in the final output, and result in an unsightly, inaccurate interpolation. To avoid these spikes, the following bias matrix β is added to A^TA before calculating the curve fit:

$$\beta = Ik \tag{4.3}$$

Where I is the identity matrix and k is a small scalar constant, usually in the range 0.01 to 0.1. This bias matrix creates a small penalty for non-zero coefficients that increases as coefficients grow in size. This causes the curve fitting algorithm to favor solutions that have small coefficients, preventing any extreme slopes from forming. This results in a slightly less accurate fit for the current layer, but subsequent layers will quickly chip away at the error introduced by the bias matrix.

Performance

The layered approach is very fast, both from a theoretical and practical standpoint. For each layer, we iterate through each input point and influence that point's neighboring interpolation nodes. Then, we compute the optimal fit at each node. Let $\mathbb{D}$ be the dimension of the input, b be the number of basis functions, and j be the number of nodes in the layer. The number of neighbors per node is $2^{\mathbb{D}}$. To compute the layer we have to make $n2^{\mathbb{D}}$ calls to AddPoint each costing $O(b^2)$, and then fit j curves, each fit costing $O(b^3)$ for a total cost of $O(n2^{\mathbb{D}}b^2 + jb^3)$. Because b, and $\mathbb{D}$ are constants, and (as I will show below) j is less then some constant multiplier of n, the time complexity to fit a layer is $O(n)$.

How many layers are needed for a good fit? Let l be the total number of layers. j grows exponentially as we add layers. As soon as we reach the point where each node only has b points or less in its neighborhood, there is almost no benefit to adding any additional layers. This is because each node produces an almost perfect fit for b or less points (it would be a perfect fit if not for the bias matrix). If the density of the input points is roughly constant throughout the field, then the number of nodes

j for the final layer will be some proportion of n. Because j grows exponentially with respect to l, the total number of layers needed is $O(\lg n)$. This makes the running time to fit all of the layers $n\lg(n)$. Each interpolation costs $O(\lg n)$ because each layer must be evaluated once. Thus, the total running time is $O(n\lg(n) + m\lg(n))$.

We will set the layer count to $\lg(kn)$ where k is a detail constant. If the data points are scattered roughly uniformly throughout the field, every node in each layer will have roughly the same number of data points in its neighborhood. Even if the points aren't distributed uniformly, Layered Interpolation still makes good predictions. Increasing k to a value greater than what would be needed for a uniform density will allow the algorithm to work well in areas where the data points are unusually dense. The bias matrix β ensures stable results in areas where the data is sparse.

In addition to being a very fast technique, Layered Interpolation produces accurate results, as we will see in chapter 5. Furthermore, the algorithm is also numerically stable. The only division operations are those required to invert each curve fitting A^TA matrix at each node. These matrices are usually small since few basis functions are needed, greatly reducing the number of divisions performed when they are inverted. This stability makes it practical to use single precision floating point numbers. Most modern video hardware only accepts 3D geometry with 32 bit coordinates, so this algorithm is an ideal choice for 3D plotting applications.

5. RESULTS

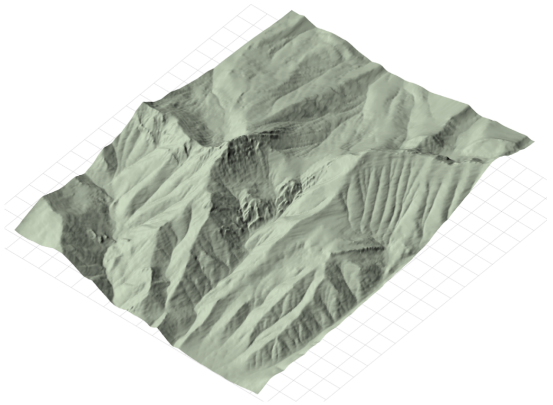
The following data sets with known Z functions were used to test the Layered Interpolation technique against other interpolation techniques. The data sets used are described below:

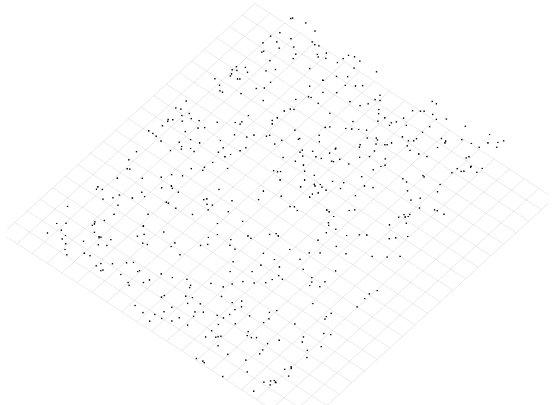
Bridgers, Fig. 6(a): A 5km by 7km section of the Bridger Range near Bozeman, Montana with $Z(x)$ corresponding to elevation

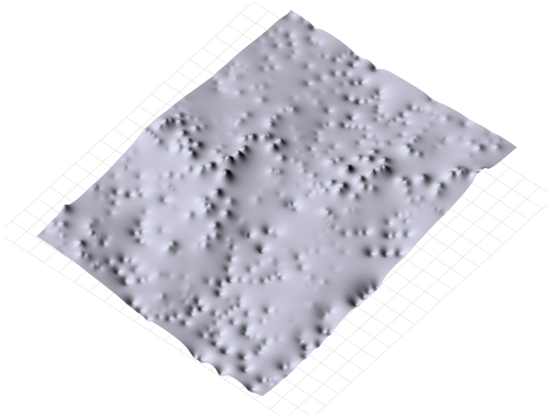
Clouds, Fig. 7(a): An image of clouds, with $Z(x)$ corresponding to brightness

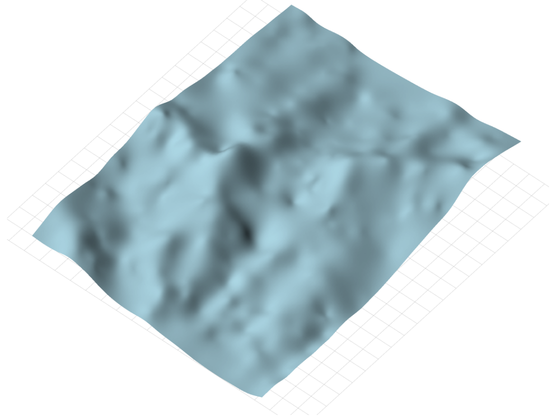
Visual Quality

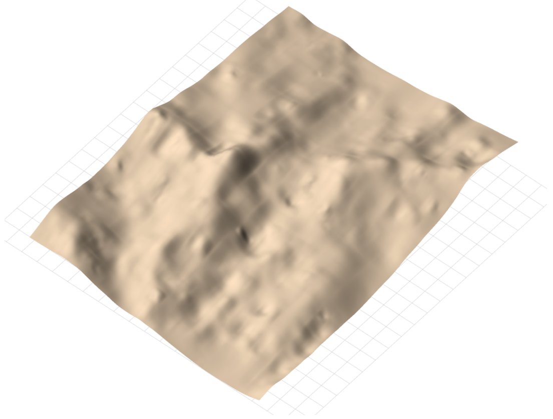
To demonstrate visual quality, n points are sampled at random locations from Z. Each interpolation method is then used to render $\hat{Z}$ using only the sampled points. The visual quality of Layered Interpolation is much better than Shepard's method, and comparable to Kriging. At a distance the output of Layered Interpolation is almost indistinguishable from Kriging. A closer look reveals that Layered Interpolation produces some minor artifacts, especially with a linear basis (see fig. 6e). These appear to be due to the cubic weighted interpolation used to blend values between nodes. A higher quality interpolation function would likely eliminate these artifacts. The simplex approach described in chapter 6 would also probably eliminate these problems.


(a) Bridger Range section


(b) Sample Locations


(c) Shepard's Method, $k = 2$


(d) Ordinary Kriging, exponential squared model


(e) Layered Interpolation, linear basis

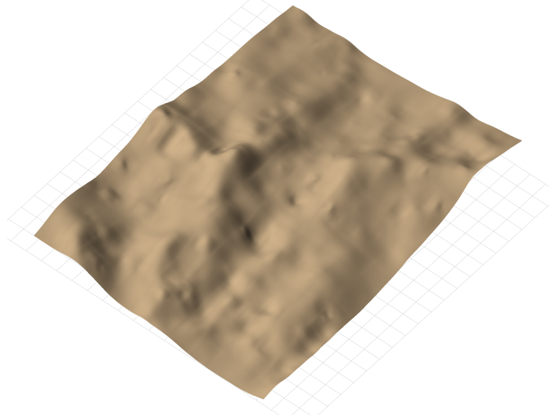
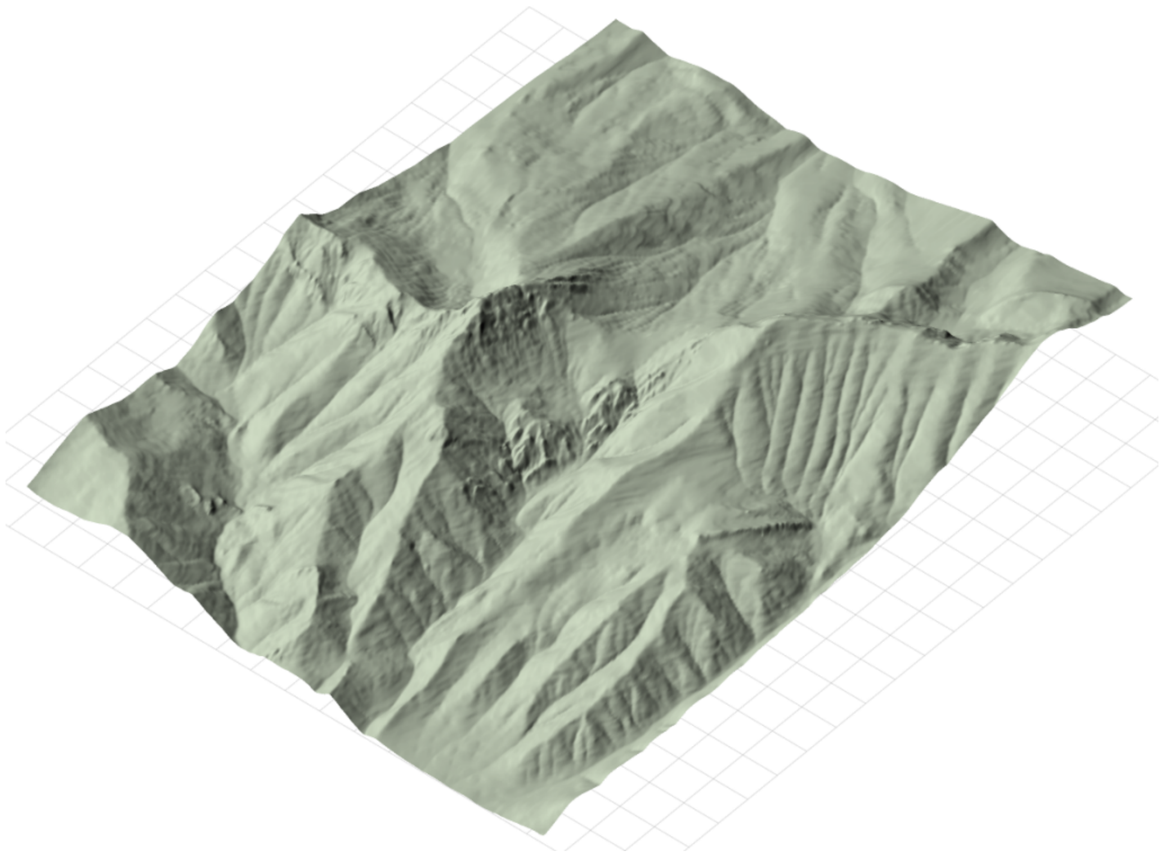
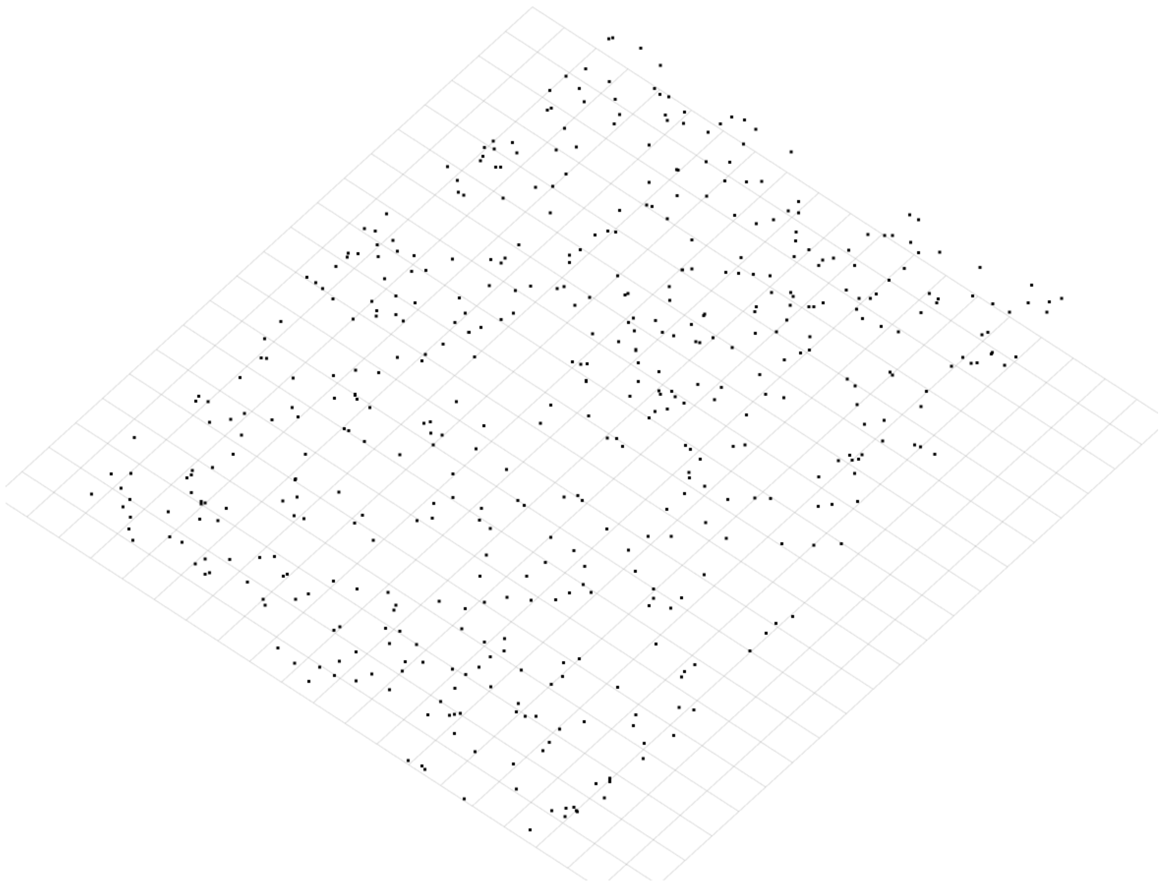

(f) Layered Interpolation, quadratic basis

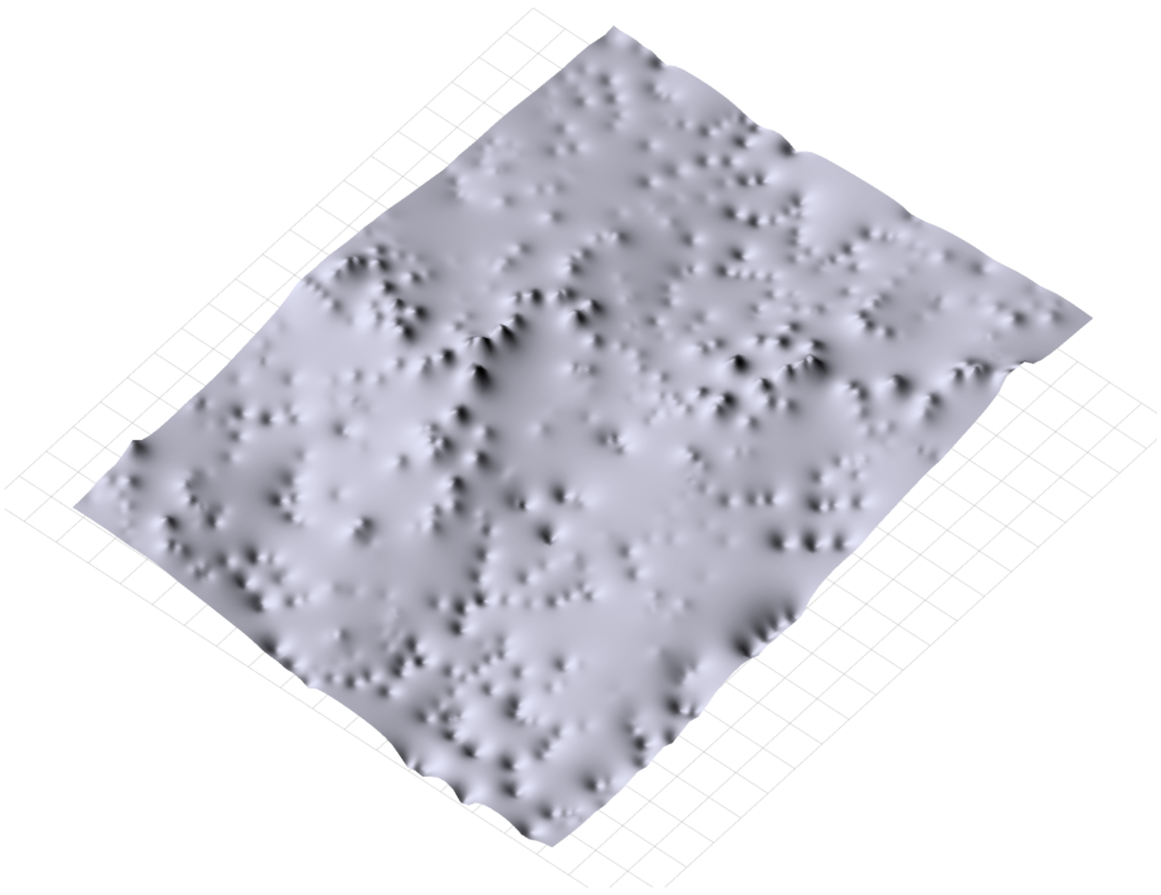
Figure 6. Visual demonstration of interpolation methods on Bridger Range data with $n = 512$ and $m = 512 \times 512$.



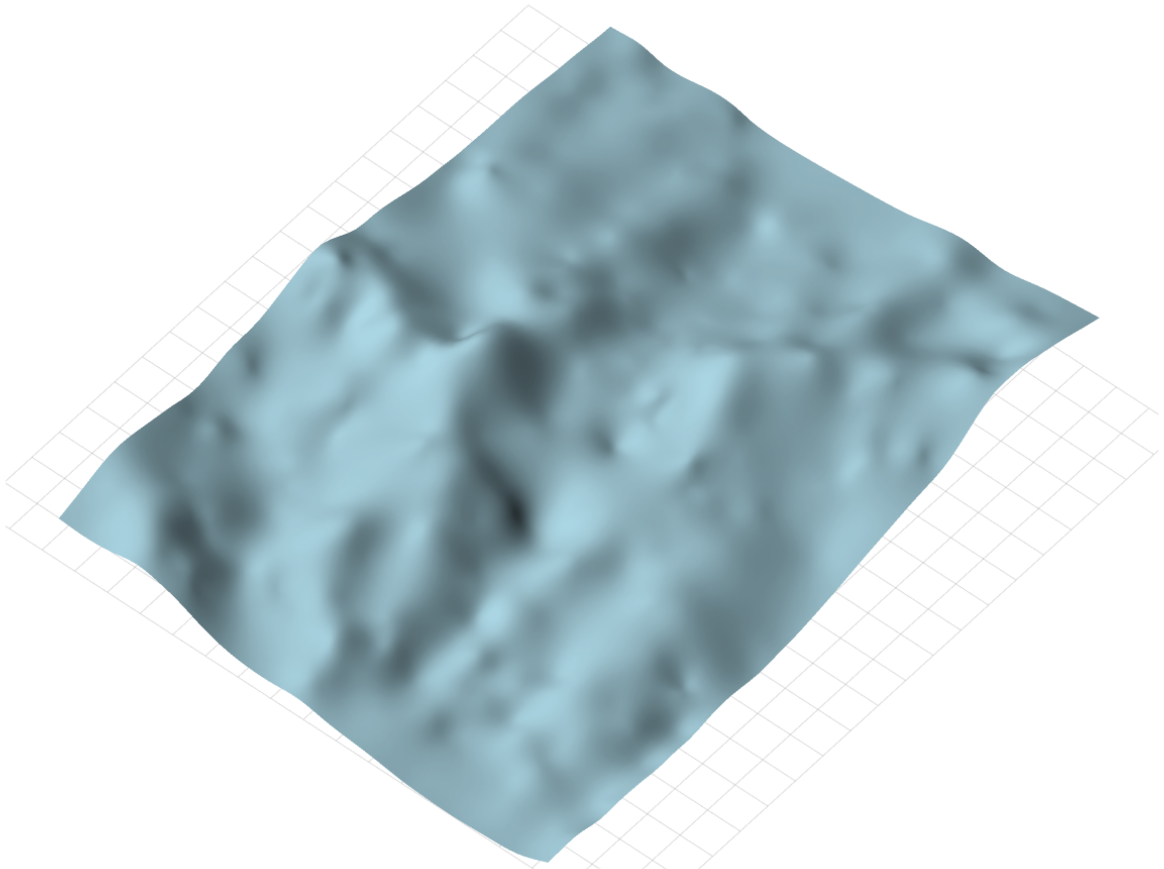
Expanded view of figure 6(a): Bridger Range section

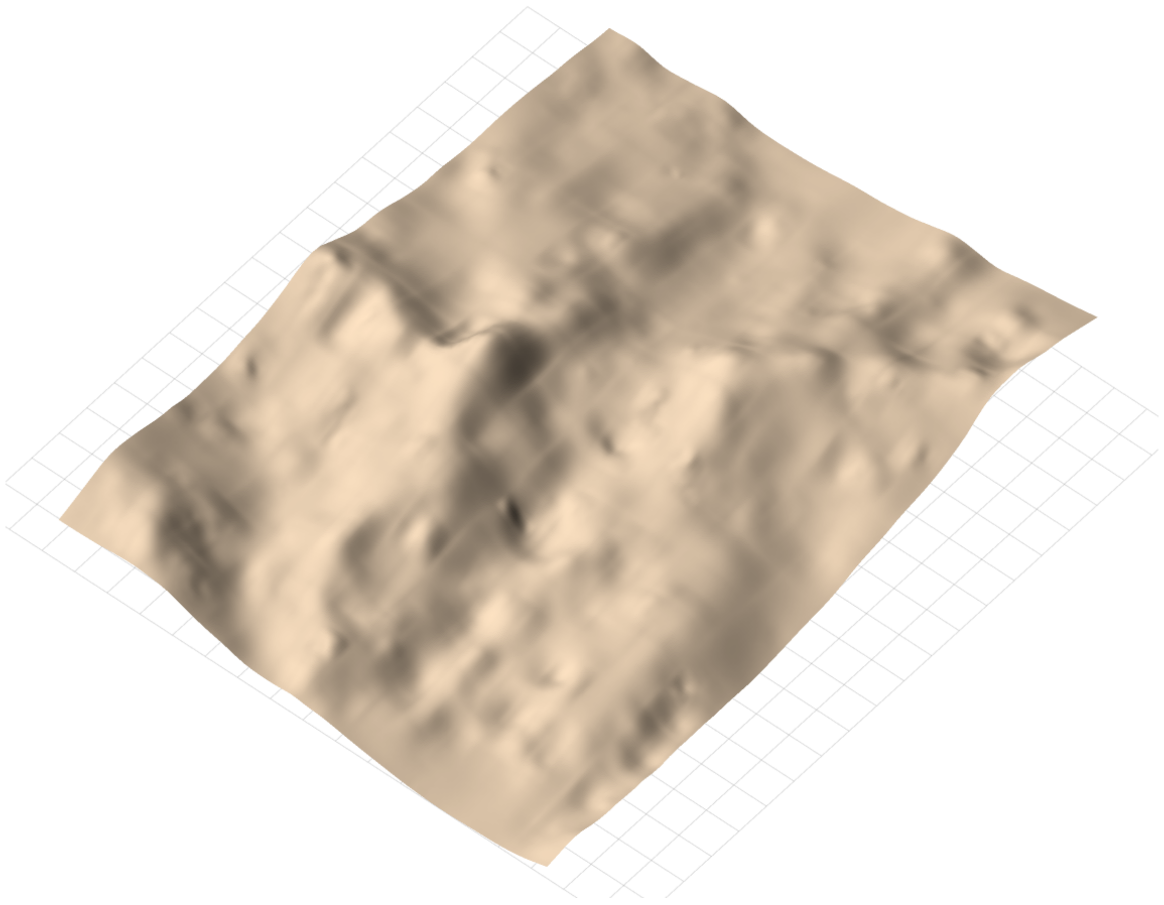


Expanded view of figure 6(b): Sample Locations

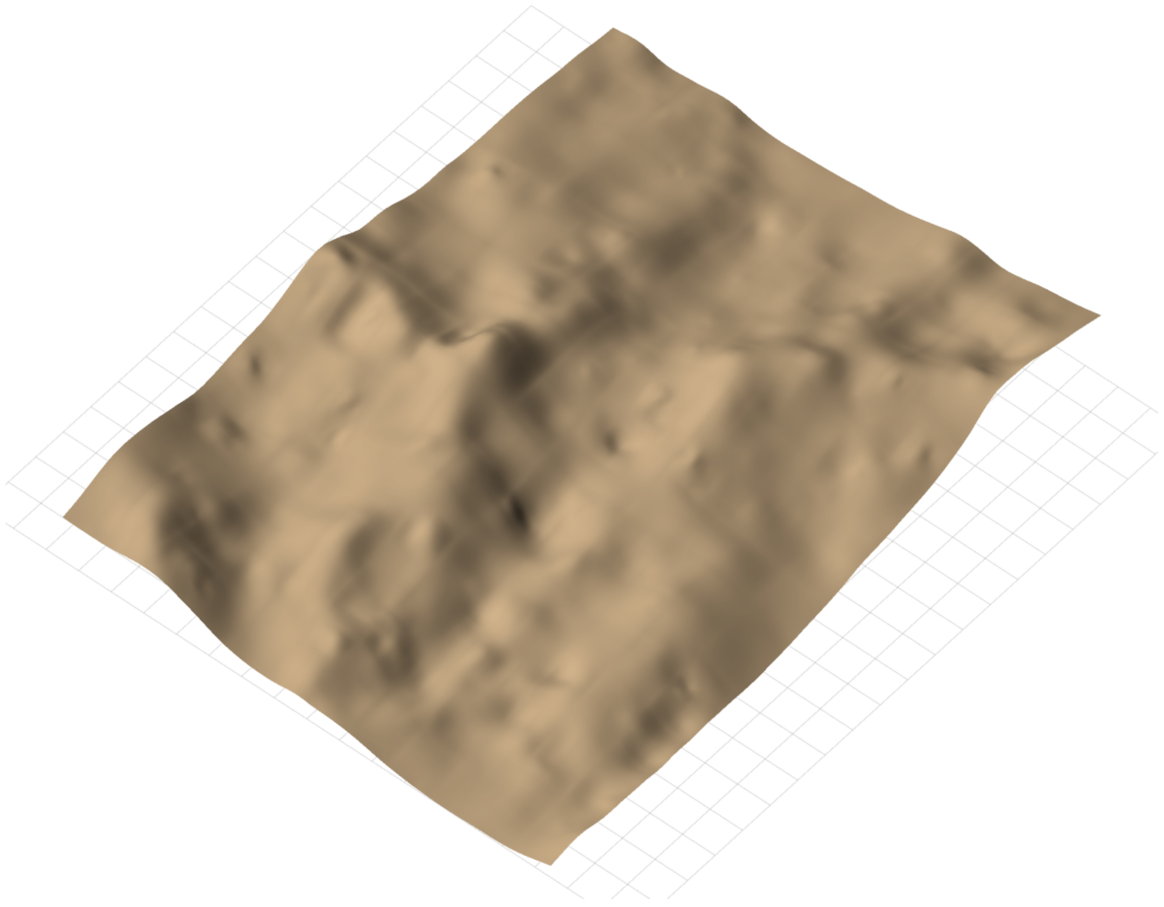


Expanded view of figure 6(c): Shepard's Method, $k = 2$

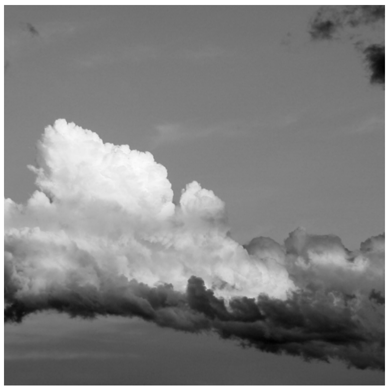

Expanded view of figure 6(d): Ordinary Kriging, exponential squared model

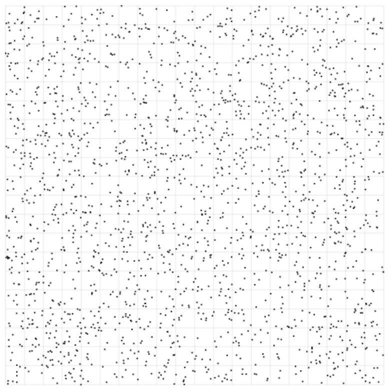


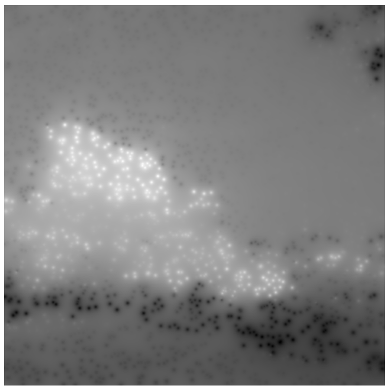
Expanded view of figure 6(e): Layered Interpolation, linear basis



Expanded view of figure 6(f): Layered Interpolation, quadratic basis


(a) Clouds Image


(b) Sample Locations

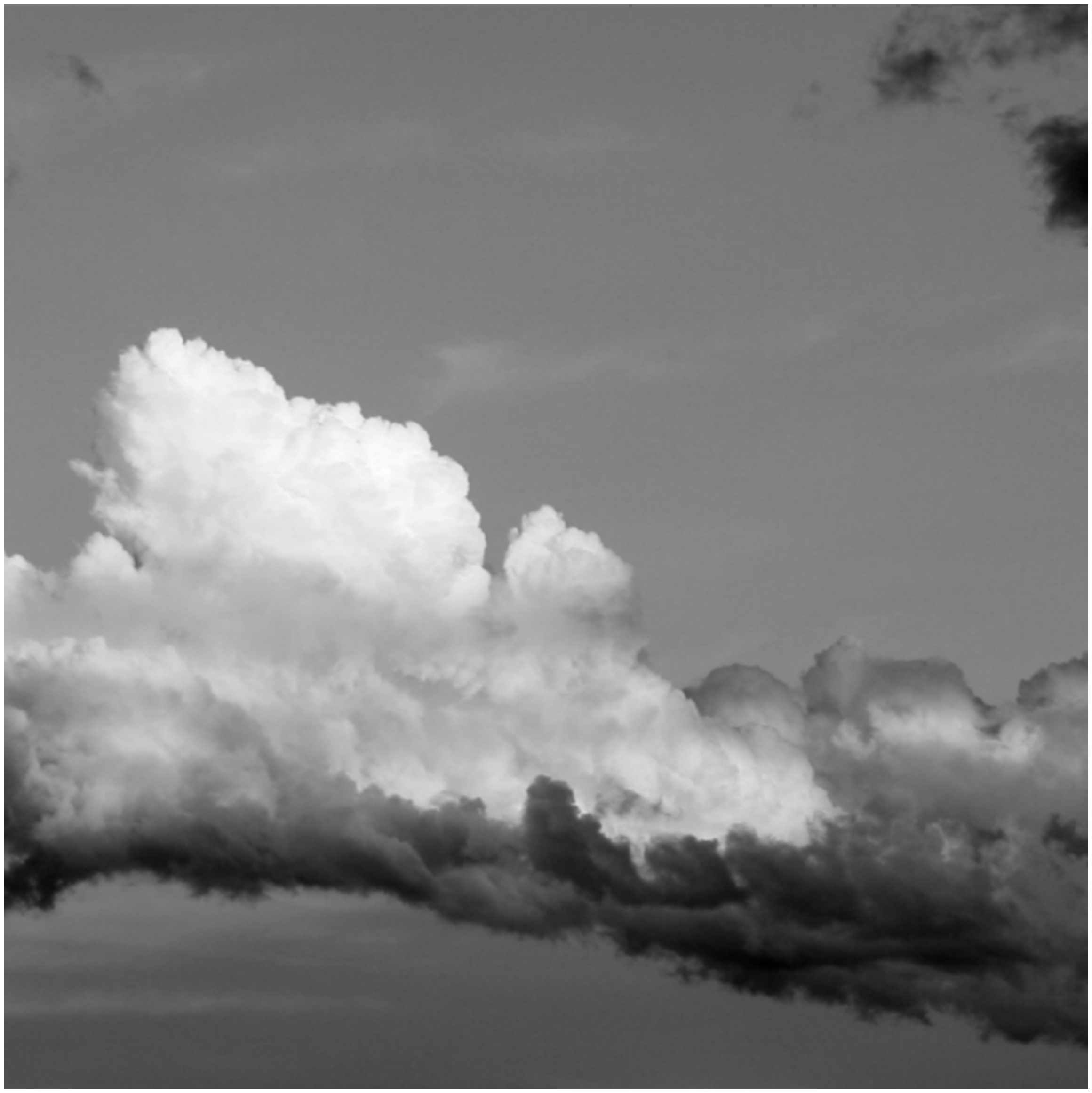

(c) Shepard's Method, $k = 2$

(d) Ordinary Kriging, Exponential Squared model

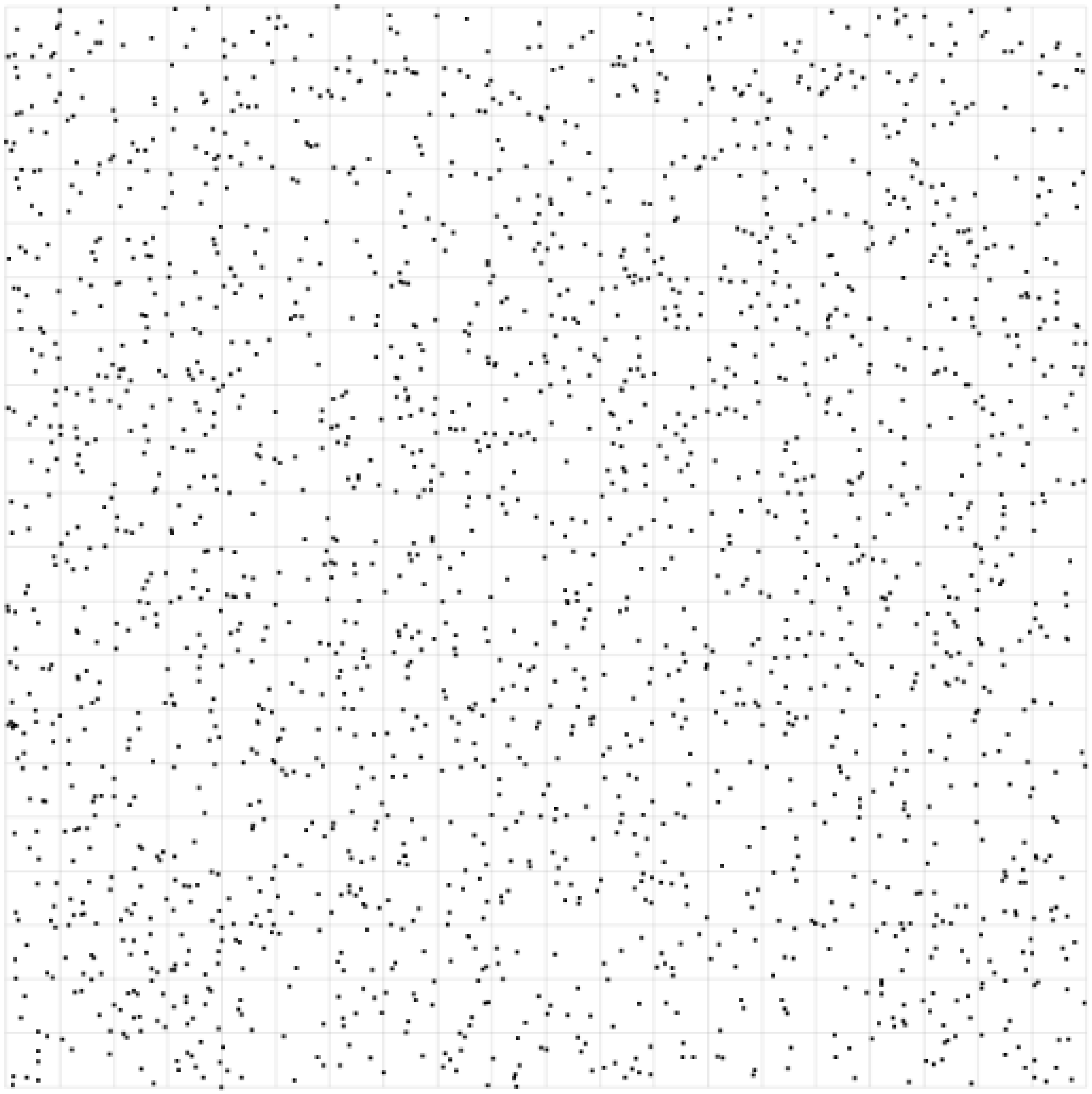
(e) Layered Interpolation, linear basis

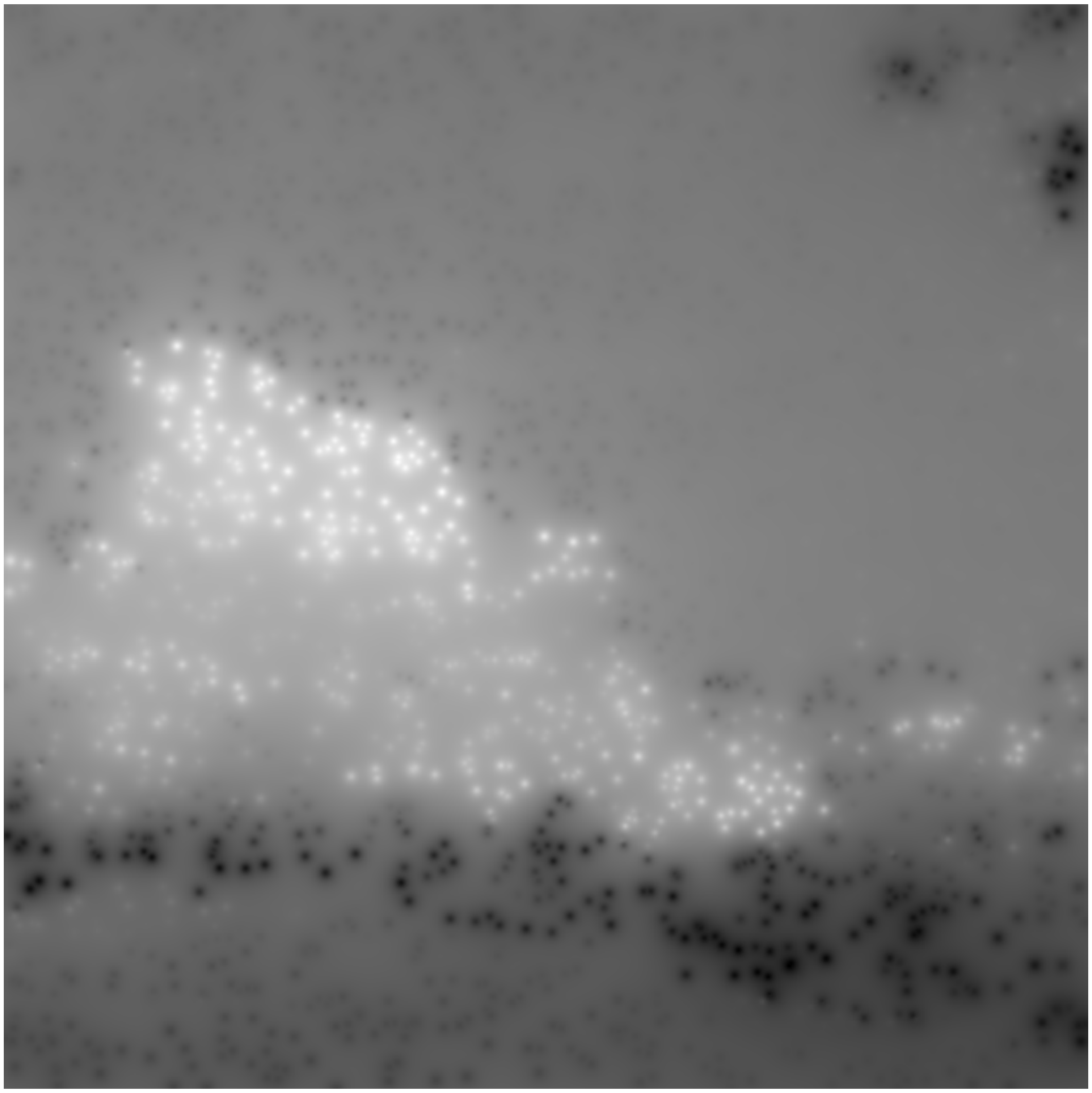
(f) Layered Interpolation, quadratic basis

Figure 7. Visual demonstration of interpolation methods on an image of clouds with $n = 2048$ and $m = 1024 \times 1024$.



Expanded view of figure 7(a): Clouds Image


Expanded view of figure 7(b): Sample Locations



Expanded view of figure 7(c): Shepard's Method, $k = 2$

Expanded view of figure 7(d): Ordinary Kriging, Exponential Squared model

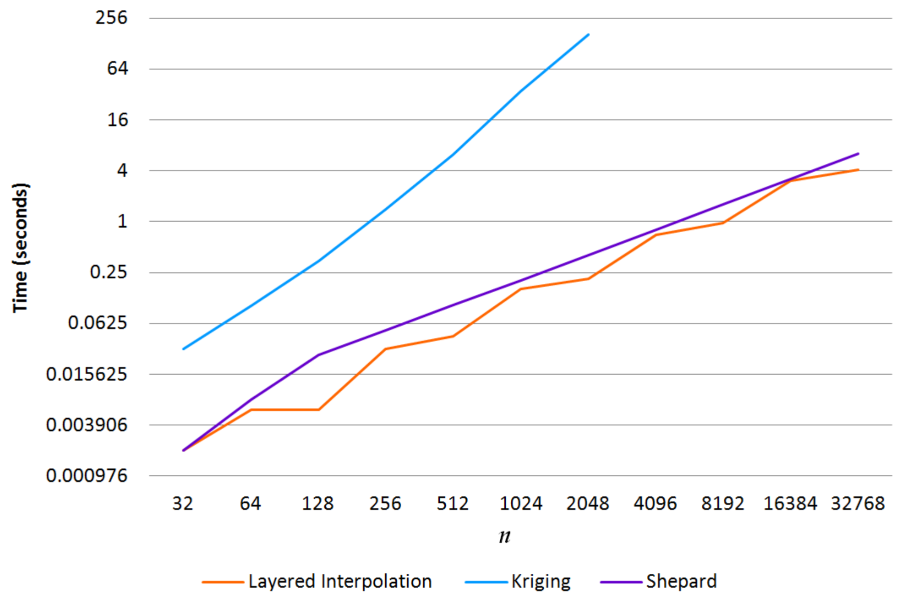
Expanded view of figure 7(e): Layered Interpolation, linear basis

Expanded view of figure 7(f): Layered Interpolation, quadratic basis

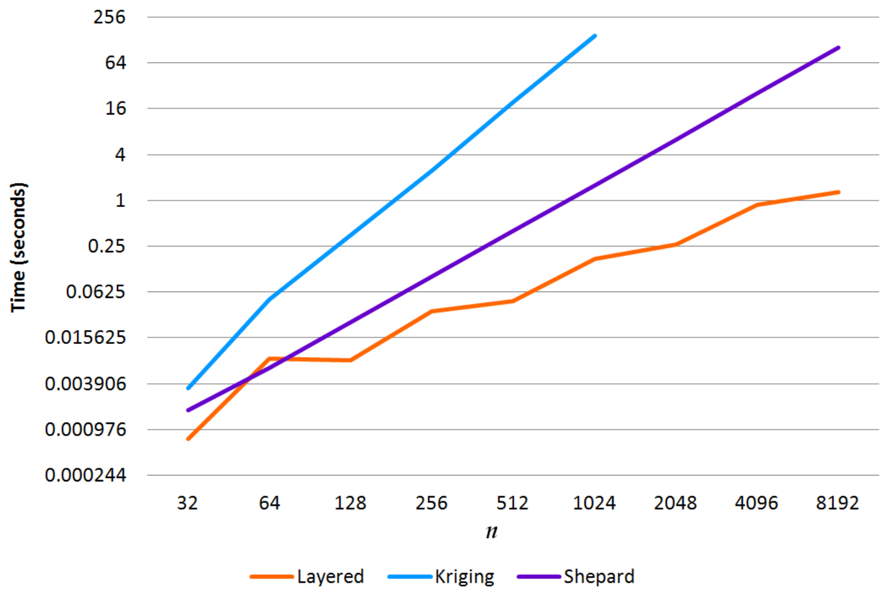
Running Time

To compare speed, each algorithm was run on the sin data set with varying n and m values and the running time was recorded. Layered Interpolation with a linear basis and Layered Interpolation with a quadratic basis have nearly identical running times, so only the quadratic version is shown for clarity. These tests were run on a Pentium 4 2.5Ghz desktop PC. The layered algorithm has a noticeable stair-step pattern (see fig. 8). These steps are formed at n values that caused the number of layers to increase. The layer count is 4 for $n = 32$ and increases every other power of 2 up to 9 layers for $n = 32768$.

Two graphs are shown. For the first graph, m is set to a constant value of 1024. The second graph uses the somewhat more realistic value of $m = 8n$. The layered approach is the fastest in all cases, but just barely so when m is constant. For increasing m values, The layered $m \lg n$ term starts to dominate and outperforms Shepard's mn running time. It also should be noted that both axes in these graphs are logarithmic.



(a) Constant $m = 1024$



(b) Increasing $m = 8n$

Figure 8. Running times of selected algorithms.

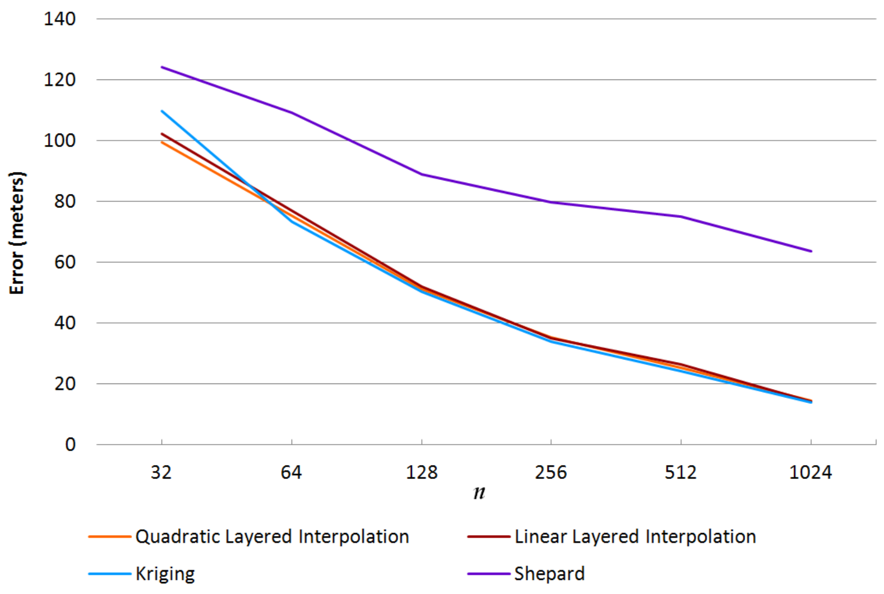
Accuracy

To compare accuracy, selected algorithms were run on each data set for varying values of n. For each trial, n points are randomly sampled, and $m = 8n$ interpolations are performed at random locations $\hat{\mathbf{x}}_i$. The error of each interpolation is $\hat{Z}(\hat{\mathbf{x}}_i) - Z(\hat{\mathbf{x}}_i)$, and the overall error ϵ is computed using the root mean square method:

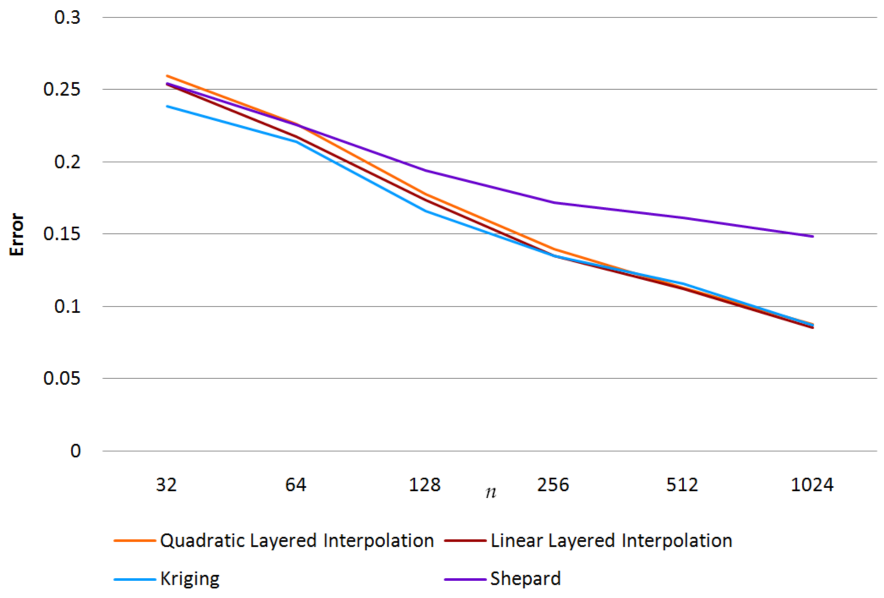
$$\epsilon = \sqrt{\frac{\sum_{i=1}^{m} (\hat{Z}(\hat{\mathbf{x}}_i) - Z(\hat{\mathbf{x}}_i))^2}{m}} \tag{5.1}$$

For smaller values of n, multiple trials were run with new random sample locations and the overall errors averaged together. This reduces bias caused by a small set of random locations favoring one algorithm over another.

Figure 9 shows the average interpolation error for varying values of n. These graphs show that Layered Interpolation is very close to Kriging in terms of accuracy. Changing the basis functions for Layered Interpolation from linear to quadratic does not significantly boost accuracy. In fact, the linear basis is more accurate for most n values in the clouds data. It seems that the main benefit of the quadratic basis is visual quality.



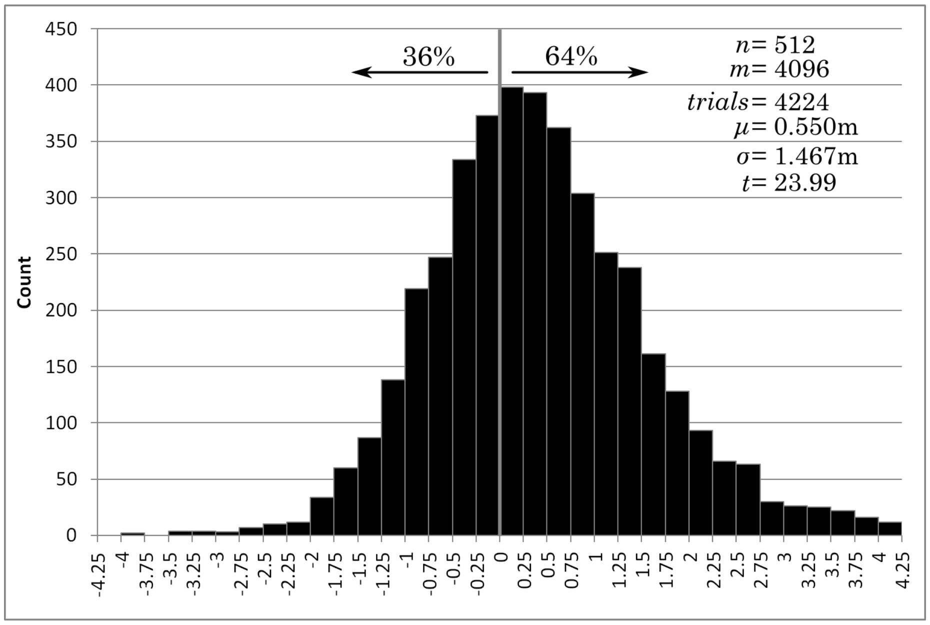
(a) Bridger Interpolation Error



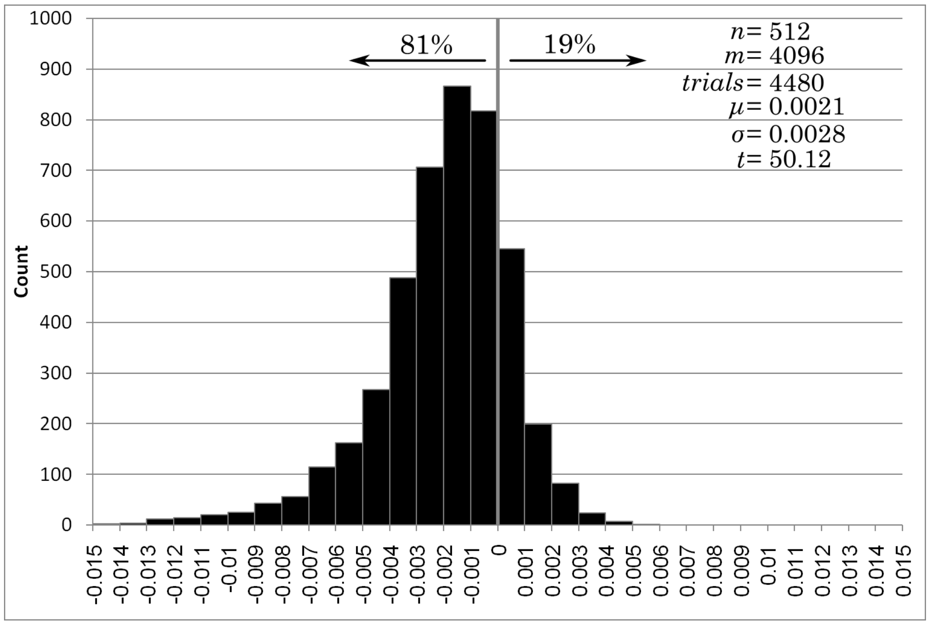
(b) Clouds Interpolation Error

Figure 9. Error of interpolation algorithms.

The difference between Kriging and Layered Interpolation is too small to draw any conclusions from the graphs shown in figure 9. A statistical analysis of Kriging versus quadratic Layered Interpolation is more useful. For this analysis, many trials were run on the Bridgers data and the Clouds data with $n = 512$ and $m = 4096$. For each trial, the difference between the Layered Interpolation RMS error and the Kriging RMS error was recorded. Histograms of these differences are shown in Figure 10. Difference values less than zero indicate trials in which Layered interpolation is more accurate than Kriging. For the Bridgers data, we can see that Kriging is more accurate, as the Layered Interpolation RMS error is on average 0.550m greater than Kriging. Despite this, Layered Interpolation still manages to outperform Kriging in 36% of the trials on the Bridgers data. When run on the clouds data, Layered Interpolation produces the most accurate results. The mean RMS error difference is 0.0028 in favor of Layered Interpolation, and Layered Interpolation produces the most accurate interpolation 81% of the time. At first glance, these error differences seem so small as to be insignificant. However, a statistical t test run on the data reveals that the differences are significant. The t values obtained for the Bridgers and Clouds data are 23.99 and -50.12 respectively. These t values are easily high enough to say with 99.9% confidence that Kriging outperforms Layered Interpolation for the Bridgers data, and that Layered Interpolation outperforms Kriging for the clouds data. These results suggest that Layered Interpolation's performance versus Kriging depends largely on the data source.



(a) Bridger Data Error Differences



(b) Clouds Data Error Differences

Figure 10. Histogram of Layered Interpolation error minus Kriging error over multiple trials with $n = 512$ and $m = 4096$. Bars to the left of zero indicate trials in which Layered Interpolation outperformed Kriging.

6. CONCLUSIONS

In this thesis I have described a fast multi-layer interpolation algorithm that can accept irregularly spaced data. My results indicate that this technique is much faster than Kriging, and even somewhat faster than Shepard's method. Both the visual quality and accuracy are comparable to Kriging. The speed of the layered algorithm makes it a good candidate for rendering plots of large, scattered data sets to modern high-resolution displays.

The Layered Interpolator performs well on two or three dimensional data, but does not scale well to higher dimensions. In higher dimensions, classic Perlin noise partitions the input space into hyper-cubes. Each hyper-cube has $2^{\mathbb{D}}$ corners, and therefore $2^{\mathbb{D}}$ gradient surfaces must be evaluated and interpolated for each point. Simplex noise [16] is a newer noise generation technique also designed by Ken Perlin. The simplex noise algorithm partitions the input space into hyper-tetrahedrons. Each hyper-tetrahedron only has $\mathbb{D} + 1$ corners, resulting in far fewer gradient surfaces being evaluated. Modifying the Layered Interpolation algorithm to partition its input into hyper-tetrahedrons should greatly improve performance in higher dimensions. Simplex noise also produces higher quality output with fewer artifacts than Perlin noise, so the simplex technique would likely improve the output quality of Layered Interpolation as well.

Although the Layered Interpolation algorithm is based on a fractal noise function, it can not be considered a fractal algorithm. Fractal noise is statistically self-similar at varying scales, and natural processes also often possess this property. A true fractal interpolation algorithm should take advantage of that fact when making predictions where data is sparse. The algorithm described in this thesis uses a simple bias matrix when an interpolation node has few points in its neighborhood. I suspect that instead it may be possible to use statistics gathered from previous layers to create a fit that is statistically similar to previous layers. This fractal bias would make the Layered Interpolation algorithm a true fractal interpolation algorithm, and is an area for further research.

REFERENCES CITED

[1] F. K. Musgrave, C. E. Kolb, and R. S. Mace. The synthesis and rendering of eroded fractal terrains. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 41–50, New York, NY, USA, 1989. ACM Press.

[2] Joshua Schpok, Joseph Simons, David S. Ebert, and Charles Hansen. A real-time cloud modeling, rendering, and animation system. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 160–166, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[3] Ken Perlin. An image synthesizer. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 287–296, New York, NY, USA, 1985. ACM Press.

[4] K. Perlin and E. M. Hoffert. Hypertexture. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 253–262, New York, NY, USA, 1989. ACM Press.

[5] Donald Shepard. A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the 1968 23rd ACM national conference*, pages 517–524, New York, NY, USA, 1968. ACM Press.

[6] T. H. Cormen et al. *Introduction to Algorithms.* MIT Press, second edition, 2001.

[7] Isobel Clark. *Practical Geostatistics.* Elsevier Applied Science, 1979.

[8] G. Matheron. Principles of geostatistics. *Economic Geology*, 58:1246–1266, 1963.

[9] D. G. Krige. A statistical approach to some mine valuations and allied problems at the witwatersrand. Master's thesis, University of Witwatersrand, 1951.

[10] Wikipedia. Kriging — wikipedia, the free encyclopedia, 2007. [Online; accessed 6-June-2007].

[11] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14:354–356, 1969.

[12] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9:251280, 1990.

[13] A. H. Thiessen. Precipitation averages for large areas. *Monthly Weather Report*, 39:1082–1084, 1911.

[14] S Fortune. A sweepline algorithm for voronoi diagrams. In *SCG '86: Proceedings of the second annual symposium on Computational geometry*, pages 313–322, New York, NY, USA, 1986. ACM Press.

[15] David G. Kirkpatrick. Optimal search in planar subdivisions. Technical report, Vancouver, BC, Canada, Canada, 1981.

[16] Robert Bridson, Ronald Fedkiw, and Matthias Muller-Fischer. Real-time shading siggraph course notes. In *ACM SIGGRAPH 2001 Courses*, New York, NY, USA, 2001. ACM Press.