Data Interpolation and Its Effects on Digital Sound Quality

Thesis

Presented to the Honors Committee of McMurry University

In Fulfillment of the Requirements for Undergraduate Departmental Honors in Mathematics

By Lynn Blair

April 8, 2008

Acknowledgements

This project would not have been possible without the support of my family friends, and the wonderful faculty members of the McMurry mathematics department. My academic advisor and honors project director, Dr. Mark Thornburg, and my thesis committee members Mr. Mike Swanson and Mr. Rich Brozovic have worked very hard and I wouldn't have been able to complete this project without their guidance and support. I owe my deepest gratitude to my parents, younger sister, and my grandmother for their support through the tough times and the times that I would get down on myself. I wish to thank my wonderful friend Rebekah Bryson and her husband Dan for their thoughts, prayers, and support through the process of putting this project together, as well as my predecessor Nicole Tunmire for her insights and support. Finally, I would like to thank my bagpipe band for always making sure I didn't get so preoccupied with this project that I forgot to practice the pipes, for without them (and my bagpipe) I would have lost my sanity.

Dedication

Dedicated to my great-grandmother, Kimiko Holland (December 10, 1918 – December 19, 2002), who watched over me as a little child while my mom was forced to work long hours to support us. I wouldn't be here today without her.

Abstract

Sound is a continuous analog wave. In modern times, technology has managed to compress sound into digital files. This involves taking certain points along the sound wave and throwing the wave itself away, thus potentially losing minor details in the sound therefore compromising the overall quality of the sound. This project is an aim to attempt to improve the sound quality of digital sound by implementing different interpolation techniques to fill in data that is missing from a digital sound file. These techniques were tried on two samples (one perfect sound wave, and one imperfect sound wave) to determine any change in the overall quality. This was accomplished by implementing a random number generator to generate data points which would then become the basis points for a sound file. These data points were read into a separate program written to implement the chosen interpolation method, and then saved the interpolated data. In doing so, I found that one method of performing Cubic Spline interpolation results in moderate improvement to the file.

Contents

Chapter 1: Introduction to Digital Sound

1.1 What is digital Sound?

Sound by its very nature is a continuous wave that can be regular or irregular in pattern. We call this kind of wave “analog.” These are the types of waves that reach our ear and are interpreted and translated by our eardrums into electrical impulses that the brain interprets as the sound we hear. Sound can be stored on many different media, each producing its own sound quality. The earliest form of recording and storing sound was by using flat disks called “records.” These were recorded on by etching a groove on a flat disk varying in depth to create the wave, which is then played back by a needle contacting the groove and vibrating in the pattern of the sound wave. This is an example of analog recording.

There is, however, a newer and more convenient method of storing sound. Such media as compact disks and various file formats that a computer can store on its hard drive for playback are examples of this type of storage. This is digital sound. Rather than storing the whole wave itself (which requires a large media as a means of doing so) points along the wave are chosen and stored. This allows for more space on the chosen media (hence a CD can store much more information than a record) . A digital sound player (such as a CD player or an MP3 player) reads these points and interprets them to create an analog sound wave. The problem with this is there are some parts of the sound (particularly higher-frequency harmonics) that are lost due to the fact that the points selected

aren't close enough to capture them. This is because these harmonics vibrate so fast the waves are very close together in these areas.

1.2 Converting from Analog to Digital and Back to Analog

1.2.1 Analog to Digital

When converting analog sound into digital sound, the first step is for the Analog-to-Digital Converter (ADC) to sample the analog sound wave itself and then select points from the sound at a set frequency. For example, the CDA standard sampling rate is 44.1 MHz. In other words, it takes samples of the sound wave at a rate of 44,100 samples per second. Other audio signals (such as those created by an electronic keyboard or a synthesizer) produce audio by means of "digital synthesis." Such signals start as digital audio and only undergo the second phase of conversion, going from the digital signal to the actual analog sound that is heard (Kientzle 53).

After the conversion to a digital signal has taken place, one of two things can happen. The file can be either stored on a digital storage media or played back. When digital sound is stored, it can be stored in a number of ways, ranging from a CD, a hard drive, or in flash memory. Several file formats are used to store digital audio. CDA (the CD standard), WAVE, AAC, and MPEG (MP3) are the most common file formats. These are all stored using lossy compression techniques, which is the reason that certain harmonics are lost in the conversion process as noted above.

1.2.2 Digital to Analog

In order for the sound to be heard, the digital sound must be decoded to produce an analog wave that is then played by speakers or other sound-producing devices. A Digital-to-Analog Converter (DAC) is responsible for this. Unlike the ADC that samples the analog sound, the DAC samples the stored digital file at varying bit rates. The DAC connects the points from the digital file to create a continuous wave that can then be played back. The DAC can use one of three techniques, which will be discussed here.

Oversampling is the process of sampling a signal with a sampling frequency higher than twice the frequency of the signal being sampled (Kientzle 31).

1.2.2.1 Definition An oversampled signal is said to be oversampled by a factor of ß, as defined by the formula

$$ß = \frac{f_s}{2 f_H}$$

Or as

$$f_s = 2ßf_H$$

Where f_s is the sampling frequency and f_H is the highest frequency of the signal being sampled (Kientzle 31).

Oversampling has two notable advantages. First, the higher sampling rate used in oversampling is capable of achieving better resolution in the digital-to-

audio conversion. Second, it has the advantage of having a great level of noise reduction/cancellation.

Two other methods commonly used by DACs are upsampling and downsampling. These are done by simply increasing or decreasing the sample rate used by the ADC when it translated the analog to the digital. It is the sampling rate that the DAC uses for playback that gives the output bit rate of a particular sound file (Kientzle 36). These methods are quicker and require less programming code, but these methods don't give the noise canceling capability that oversampling gives.

Note that the sampling rate converting back to analog from digital might be different from the rate used to convert the original analog into digital. This can result in major discrepancies between the original sound wave and the sound wave the DAC produces. Any of these discrepancies have a negative effect on the overall sound quality.

1.3 The WAVE File Format

This thesis will primarily consider sound files that are stored in the WAVE format. The WAVE file format was developed as the standard for storing audio on PCs by Microsoft and IBM.

WAVE file formats are always stored using the method of placing the least significant byte of information first in an array. This array is stored in memory as a string, which is a set of characters. This string contains indicators that point out significant notes, point labels, etc. that the audio player will need to recognize to play the sound file correctly. (The Sonic Spot).

The very clear advantage of WAVE files is that they support several different sampling rates, audio channels, and bit resolutions. Though they do tend to be rather large files, they are one of the most popular choices for recording artists who record in digital sound. This thesis will take into consideration WAVE files that are equal to the CDA standard for compact disc digital audio, which use a sampling rate of 44.1 MHz (that is, 44,100 samples per second).

Chapter 2: Data Interpolation Techniques

Data interpolation is a means of taking two points and fitting a function in between them to estimate the location of a third point in between the two given points. There are several different means of data interpolation, however in this thesis we will take a look at three selected methods to be applied to our sound samples. These are Piecewise Linear Interpolation, Lagrange Polynomial Interpolation, and Cubic Spline Interpolation.

2.1 Piecewise Linear Interpolation

Perhaps the simplest and quickest of all known interpolation methods is the Piecewise Linear Interpolation method. If you recall the midpoint formula from high school algebra for finding the point exactly halfway between two given points, this is very similar. Piecewise Linear Interpolation calculates the slope and y-intercept of the particular line that passes through two points to allow one to estimate the location of any point chosen between the two points.

2.1.1 Definition Given two known points A := (x_0, y_0) and B := (x_1, y_1) the *linear interpolant* is defined as the straight line between the two points. For a value x in the interval (x_0, x_1), the value y along the straight line is given from the equation

$$\frac{y - y_0}{y_1 - y_0} = \frac{x - x_0}{x_1 - x_0}$$

This can then be solved for y to yield

$$y = y_0 + (x - x_0)\frac{y_1 - y_0}{x_1 - x_0}$$

Which is the formula for the linear interpolation in the interval (x_0, x_1) (Bradie 387).

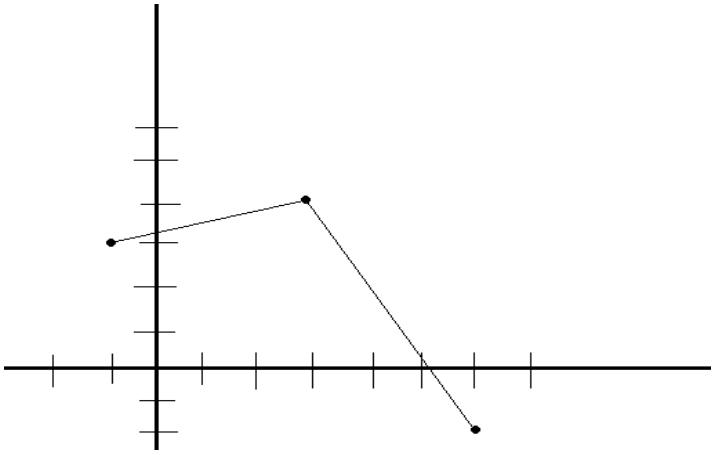


Figure 2.1.2 an example of Piecewise Linear Interpolation Using Example 2.1.3

2.1.3 Example Suppose we wanted to do a Piecewise Linear Interpolation on the points A = (-1, 3), B = (3, 4), and C = (6, -2). First we calculate the interpolant between A and B, the results of the calculation give us the following:

$$y = 3 + (x+1)\frac{4-3}{3+1}$$

$y = \frac{x+13}{4}$, which is the line that connects the points A and B.

We now do a similar step to calculate the interpolant between the points B and C:

$$y = -2 + (x-3)\frac{-2-4}{6-3}$$

$y = -2x + 4$, which is the line connecting B and C.

It's obvious from Figure 2.1.2 that Piecewise Linear Interpolation and our example are not very precise at all. Using straight lines to connect the points results in a very jagged connection, and the chances of missing points lying on straight lines between points are very unlikely. I highly suspect that this will prove to be detrimental to interpolating data points in a sound file as the wave will flatten out to close to a straight line that does not accurately reproduce sound. We will show this in chapter 3 section 1 when the results are analyzed.

2.2 Lagrange Polynomial Interpolation

Polynomial Interpolation is a method of fitting a single non-linear polynomial to a set of data points. Upon determining the number and location of the points involved, the Polynomial Interpolation method creates a polynomial with a degree one less than the number of points in the data set that passes through each of the given points (e.g. with 3 data points a quadratic is created, with 4 data points a cubic is created, etc.). There are several different forms of the Interpolating Polynomial, but this section will focus on one: The Lagrange Form of the Interpolating Polynomial.

2.2.1 Definition Given a set of k+1 points in the Cartesian coordinate system, the *interpolation polynomial* is given by the linear combination:

$$\sum_{j=0}^{k} y_j l_j(x)$$

Where $l_j(x)$ is the combination of *Lagrange basis polynomials* as follows:

$$l_j(x) = \prod_{i=0, i\neq j}^{k} \frac{x - x_i}{x_j - x_i}$$

The result is a polynomial with degree k. (Bradie 341)

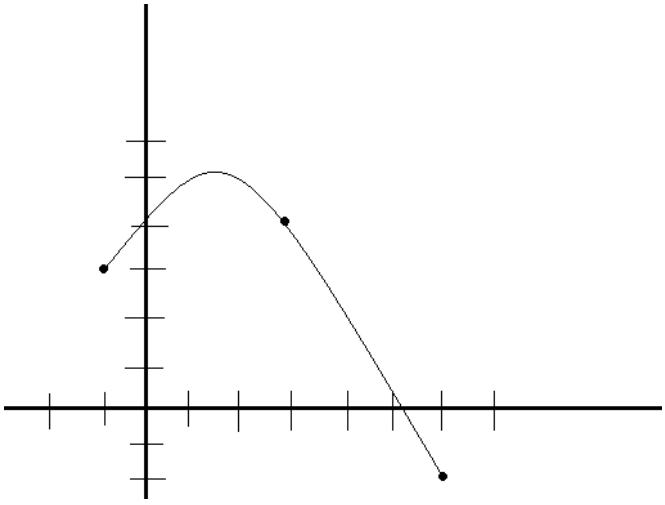


Figure 2.2.2 the data set in example 2.1.3 with the 2nd degree interpolation polynomial.

2.2.3 Example Suppose we want to perform a polynomial interpolation on the points used in the Piecewise Linear example. We would first calculate all of the Lagrange basis polynomials as follows:

$$l_0(x) = -\frac{x-3}{4} * -\frac{x-6}{7} = \frac{1}{28}(x^2 - 9x + 18)$$

$$l_1(x) = \frac{x+1}{4} * -\frac{x-6}{3} = -\frac{1}{12}(x^2 - 5x - 6)$$

$$l_2(x) = \frac{x+1}{7} * \frac{(x-3)}{3} = \frac{1}{21}(x^2 - 2x^2 - 3)$$

The interpolation polynomial is then the sum of the corresponding y value times each of the basis polynomials:

$$L(x) = 3(\frac{1}{28}(x^2 - 9x + 18)) - 1(\frac{1}{3}(x^2 - 5x - 6)) - 2(\frac{1}{21}(x^2 - 2x - 3))$$

$\approx$ -.32143x^2 +.89286x $+ 4.21426$, which is the approximate Lagrange polynomial for this data set.

As we can note, the curved graphs of the interpolants in this method would more accurately predict where missing data lies than the Piecewise Linear Interpolation method would. However, for our purposes of digital sound, my tests demonstrate that such an interpolation would make the sound behave very erratically, in addition to the interpolation being highly inefficient. As the number of data points increases, so does the degree of the interpolating polynomial. When the standard for digital sound on compact disks is 44,100 samples per second (meaning there are 44,100 data points for every second of sound), the interpolating polynomial would then be of degree 44,099 for a mere one second of sound. When the degree of a given polynomial rises, the height of the maxima and minima above and below the X-axis also tends to increase. For uses in digital sound, this would create a lot of unwanted and unnecessary background noise. Thus Lagrange Polynomial Interpolation turned out to be the worst choice of the three for the interpolation of digital sound. We will take a look at the results of Lagrange Polynomial interpolation as applied to a digital sound file in section 2 of chapter 3.

2.3 Cubic Spline Interpolation

We have seen the various issues raised by the two above interpolation techniques. Even though they are much easier to implement and calculate than the method presented in this section, they have negative effects on the sound solely due to their unnatural behavior when compared to analog sound.

In this section, we'll look at spline interpolation. Spline interpolation has certain similarities in regards to both of the above methods, as well as some unique aspects. Like Piecewise Linear Interpolation (which is actually an example of Spline Interpolation), the interpolant is a piecewise function. This means that the degree of the polynomial is restricted to the degree that is selected. For example, Quadratic Splines are restricted to polynomials of degree 2, Cubic Splines to polynomials of degree 3, etc. However, unlike Piecewise Linear Interpolation, Spline Interpolations have smooth curves, which more accurately represent the behavior of sound waves. We will look at two ways of performing splines using cubic polynomials.

For the first method, finding the piecewise function id done by fitting a cubic function that will pass through three points, and require that the right endpoint derivative to be equal to the derivative of the left endpoint on the next piece, which gives us the smoothness we desire from this interpolation method.

2.3.1 Example Suppose we want to do a Cubic Spline Interpolation to the following 5 points: (-1, -1), (0, 0), (1, 1), (2, 0), (3, -1). First we find a cubic function that passes through the first three points. The easy choice is $f_1(x) = x^3$. Let the second piece of the function be defined as $f_2(x) = c_1x^3 + c_2x^2 + c_3x + c_4$. Now, the derivative of the first function evaluated at 1 will be equal to the derivative of our second cubic at 1. This gives us the following:

$$f_1(x) = x^3$$
$$f_1'(x) = 3x^2$$
$$f_1'(1) = 3(1)^2 = 3$$

From this we know that the derivative of our second cubic function must equal 3 when evaluated at 1. Taking the derivative of f_2, we find that $f_2'(x) = 3c_1x^2 + 2c_2x + c_3$. We also know that the function itself must equal 1 when evaluated at 1, must equal 0 when evaluated at 2, and must equal –1 when evaluated at 3. Given this information, we obtain the following system of equations:

$$3c_1 + 2c_2 + c_3 = 3$$
$$c_1 + c_2 + c_3 + c_4 = 1$$
$$8c_1 + 4c_2 + 2c_3 + c_4 = 0$$
$$27c_1 + 9c_2 + 3c_3 + c_4 = -1$$

To obtain the cubic satisfying the above, we solve the system of equations for our constants. Using a TI-83 calculator and Gauss-Jordan elimination, the result yields that $c_1=2$, $c_2=-12$, $c_3=21$ and $c_4=-10$. Thus the second part of our interpolant can be expressed as $2x^3-12x^2+21x-10$. So we say our interpolant

function $f(x) = x^3$ on the interval [-1, 1] and $f(x) = 2x^3-12x^2+21x-10$ on the interval (1, 3].

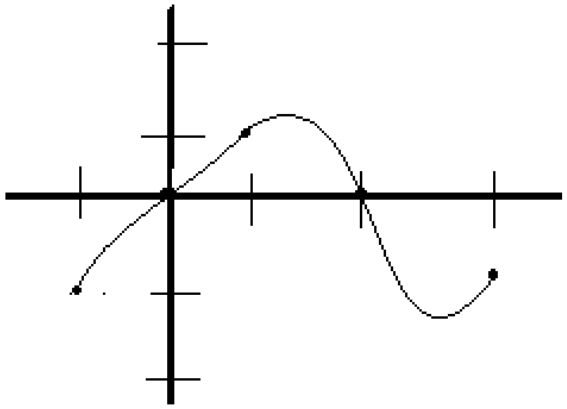


Figure 2.3.2 Example 2.3.1 Graphed

Another method of finding Cubic Splines (and more commonly used) is to use only two data points from the set, and require that the first and second derivatives agree at the connecting points. There are several constrains we can use, however in the following examples we'll look at "natural splines." These set the second derivatives equal to 0 at the far left and far right nodes. This is the version of Cubic Spline Interpolation that I chose to implement due to its ease of coding.

2.2.2 Definition *A Cubic Spline Interpolant* of f relative to the partition $a = x_0 < x_1 <\ldots<x_n = b$ is a function S that satisfies the following:

1) $S(x) = S_j(x) = a_j + b_j(x-x_j) + c_j(x-x_j)^2 + d_j(x-x_j)^3$ on the interval $[x_j, x_{j+1}]$.
2) S interpolates f at $x_0, x_1,\ldots, x_n$.
3) S, S', and S'' are continuous on [a, b]. (Bradie 393).

2.3.3 Example Let's return to the three data points we used in the Piecewise Linear and Lagrange examples. Suppose we wanted to do the second method to create a Cubic Spline on these data points.

Cubic Spline Interpolation has notable advantages and disadvantages over the previous two methods we looked at. The advantage to Cubic Splines is the high level of smoothness within the curve, and that the function being a piecewise function of degree 3 ensures controlled behavior within the interpolation. This allows us to get a better estimate of where these missing points lie when executed on a digital sound file. The disadvantage with this technique is that it's very computationally intensive, and therefore in terms of execution time it's rather slow. Despite its advantages, Cubic Spline interpolation performed disappointingly in the interpolation of sound. The results from testing this method will be in section 3.3.

Chapter 3: Results

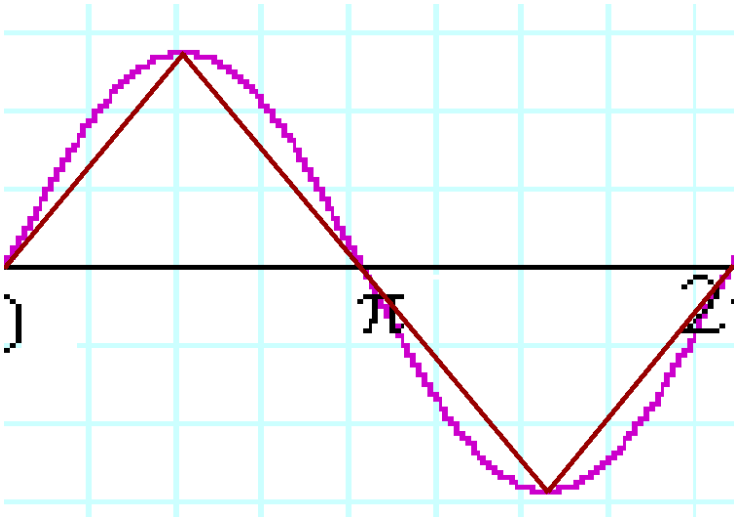
For the results of testing contained herein, to demonstrate each of the hypotheses given in the previous examples, let our first test sample be defined by the equation f(x)=3sin(x). This is an example of a perfect sound wave (meaning a constant tone, volume, and pitch is assumed). We will do three different samplings on this equation, each time increasing the sample rate. We'll start by just sampling the zeros and the extrema of the function (so that our sample domain is 0, π/2, π, 3π/2, 2π), then the second time through the halfway points will be added in (so our sample domain will be 0, π/4, π/2, 3π/4, π, 5π/4 3π/2, etc.). In our last sample, we'll double the sampling rate again, and add in the halfway points between those points.

Our second test sample used a random number generator to generate 11 y values between -10 and 10 for x values of 0 through 10. I then connected these points by hand to make a sound wave (note with 11 data points this sound would be of extremely short duration: about 4 milliseconds of sound), but this is more than enough to demonstrate the behavior of an actual sound wave.

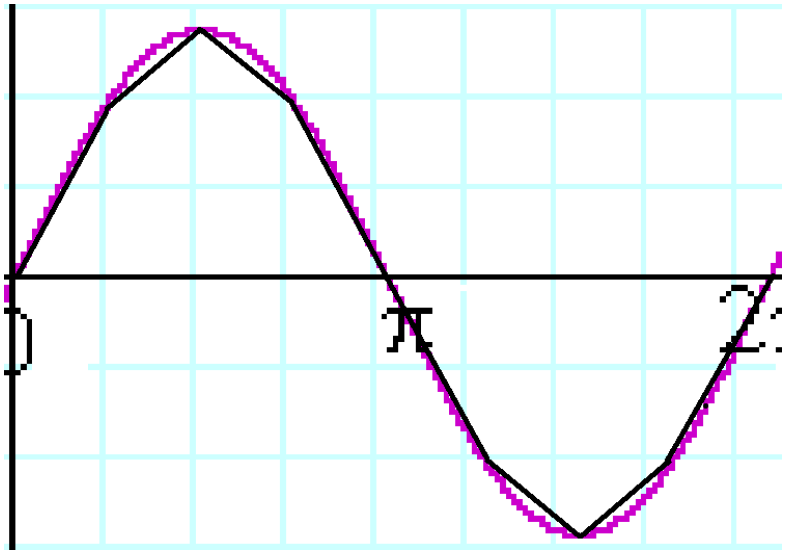
3.1 Piecewise Linear Results

Our first low sample rate returned very inaccurate interpolation results using this method. As the only end points for our lines were the extrema and zeros of our function, this gave a very inaccurate interpolation of the data, as seen in the diagram on the next page. The distances between the interpolants and the actual sound wave might not seem like much at first glance, but when

put in perspective that it's sound we're dealing with, then the difference in the harmonics becomes very large.

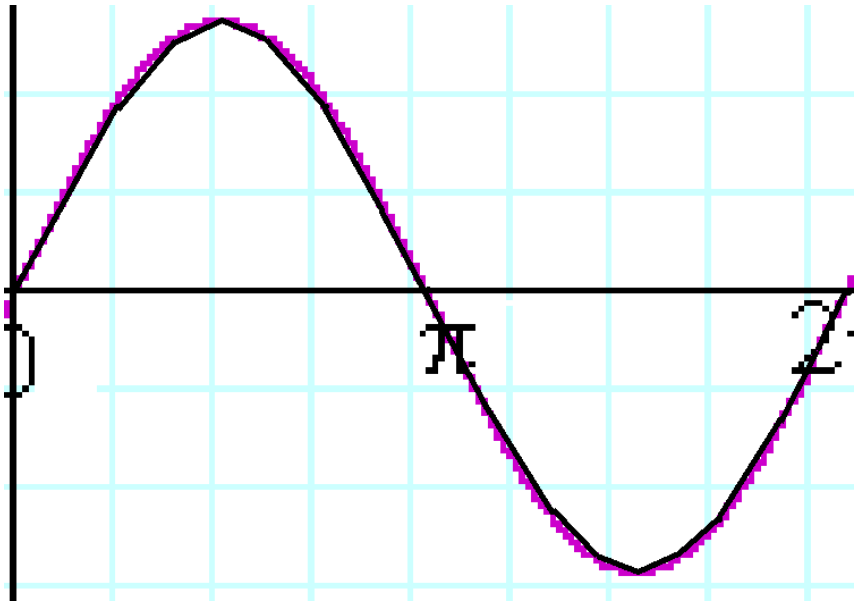


The next sample I used included points exactly halfway between our initial 5 interpolating points. The error improved, though it's still quite a ways off the mark in several places:



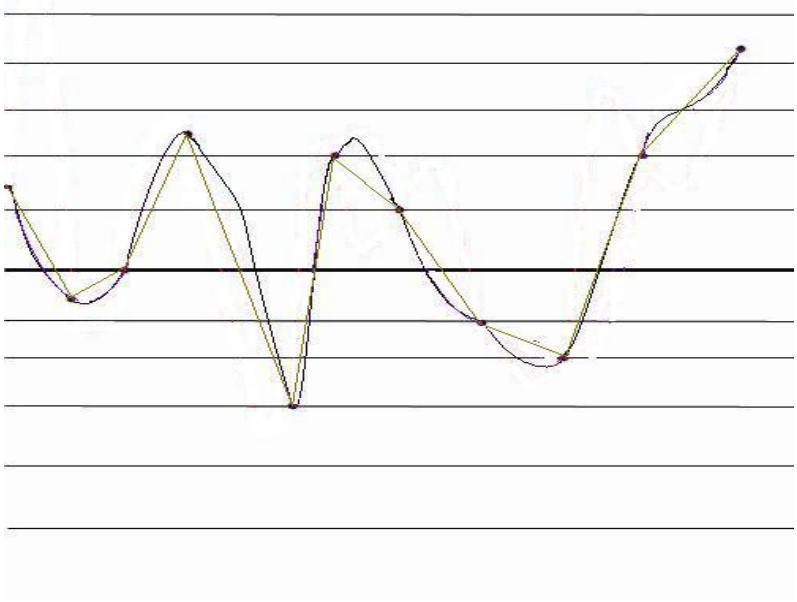
Notice from the previous diagram that the places that have the greatest amount of error are those places that the wave doesn't have a very sharp incline or decline. Shaper inclines result in smaller error.

Lastly, using the highest sampling rate, we can obtain even greater accuracy. This one actually did very well. The error was very minimal, as can be seen from this diagram:



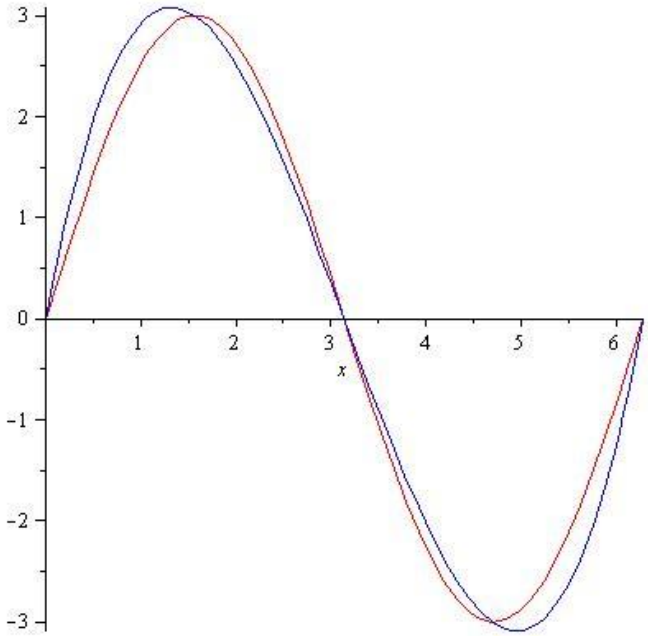
We conclude that with a moderate sample rating, Piecewise Linear Interpolation works reasonably well when dealing with perfect sounds. However, realistically, there are no sounds that are absolutely perfect, so it's not possible to go by these results alone to interpolate sound.

As for the Imperfect sound wave, Piecewise Linear did a relatively poor job. Although not as bad as polynomial interpolation, it was still very poor. This can be seen in the diagram on the following page, where the sound wave is the smooth curve and the interpolant is the set of straight lines:

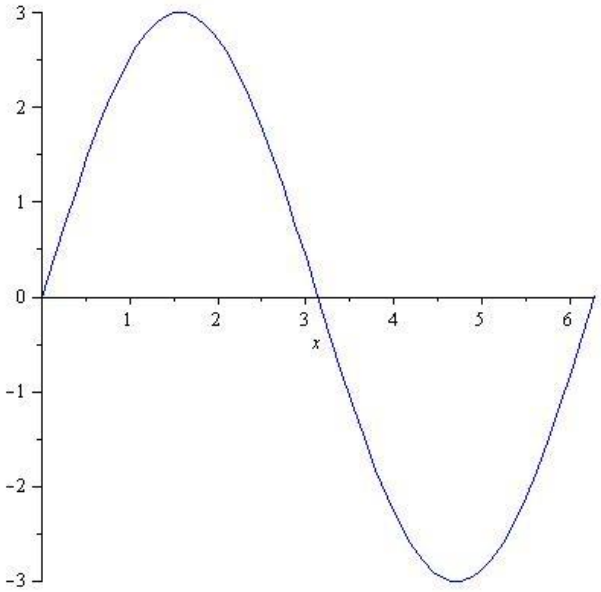


3.2 Lagrange Results

For our example of a perfect sound wave, the Lagrange Polynomial results were very good. The first example, given by the picture on the following page demonstrates the need for a fair amount of data points however. The lighter graph represents our original sound wave, and the darker represents the interpolating polynomial. Keep in mind that any significant deviation from the original sound wave has an extremely detrimental effect to the sound (if it's large enough, it can even make a completely different sound than what the original sound was).



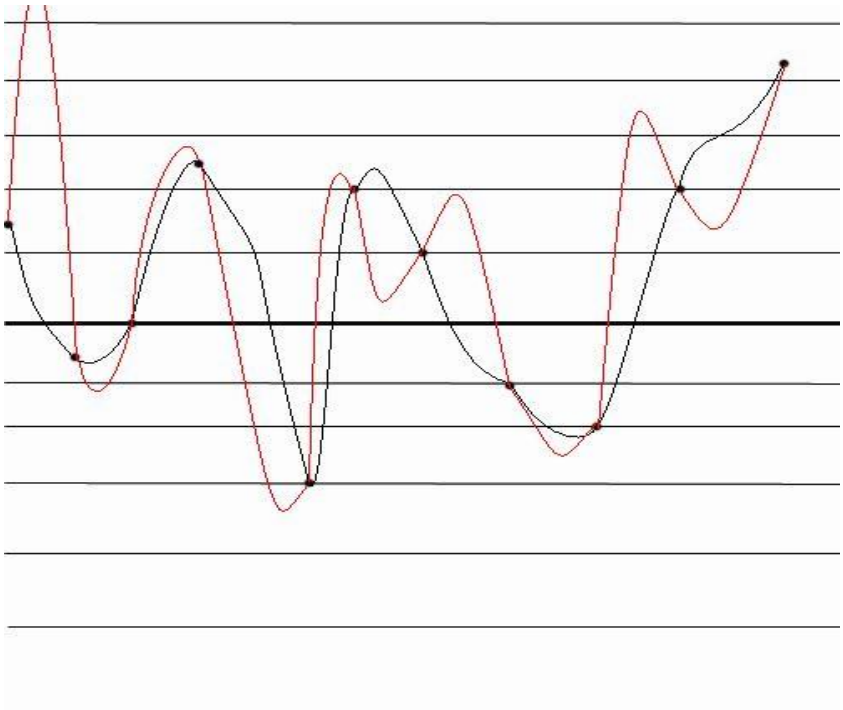
The next two trial runs resulted in the functions lying directly on top of each other when graphed using Maple, as shown in this example:



From this, we discover that if enough data points are used, we can come up with a great interpolation of a perfect sound wave. However, as with the previous

section, no sound wave is exactly perfect save a single constant steady musical note, thus these results aren't accurate to model most sounds.

In testing the imperfect sound file, I found the interpolation to behave erratically. As mentioned in the previous chapter, the higher the degree the polynomial, generally the higher the absolute maxima and minima become. This is demonstrated by the following image, where the darkler line is the actual sound wave, and the lighter line is the interpolating polynomial:

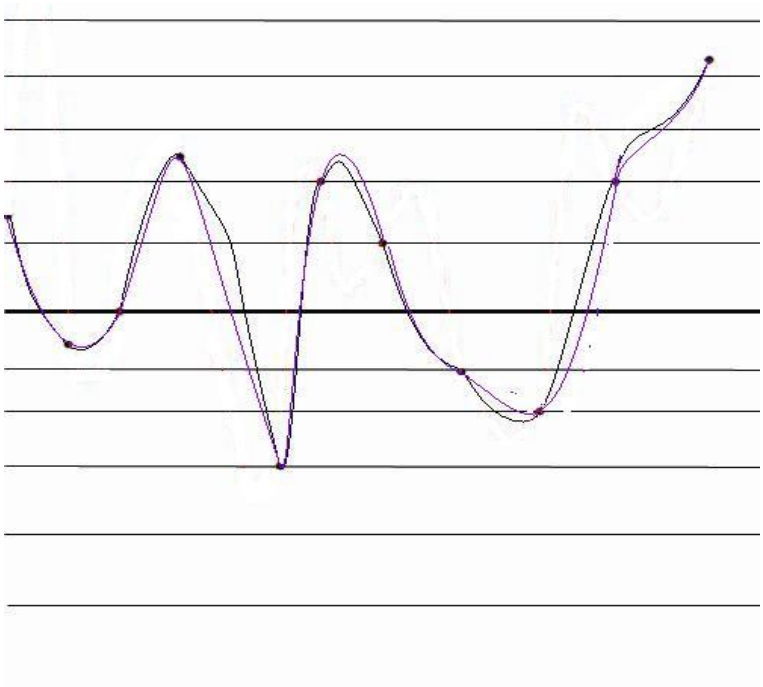


3.3 Cubic Spline Results

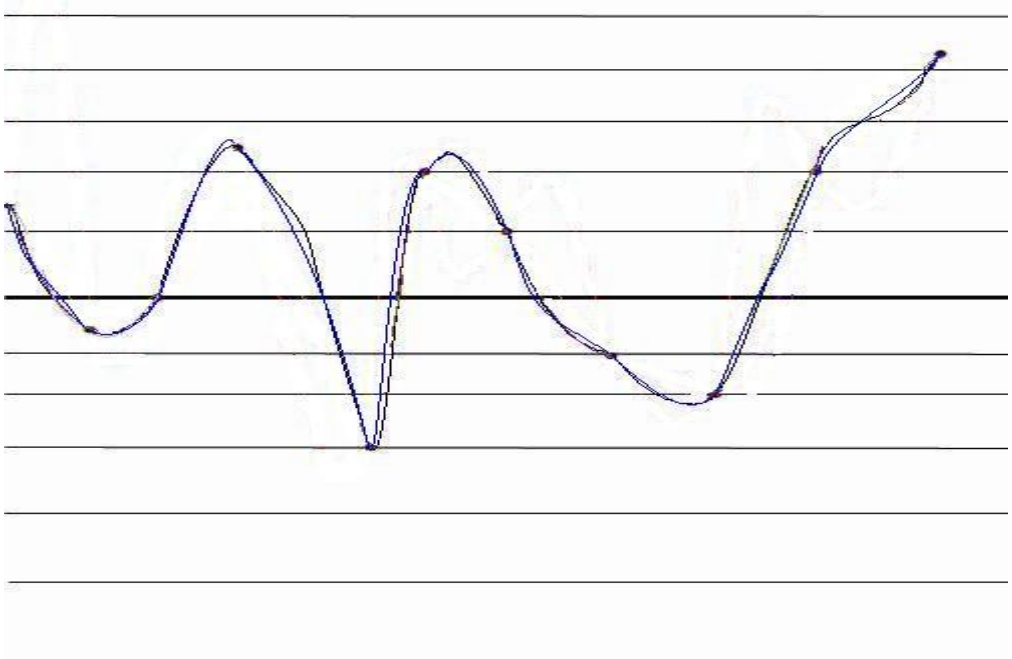
For results of the perfect sound wave, refer to the above section 3.2, as the results were identical to that of performing polynomial interpolation on the wave.

For the imperfect sound wave using the first method, overall the results were fairly good, though there are still some areas where the error is quite large.

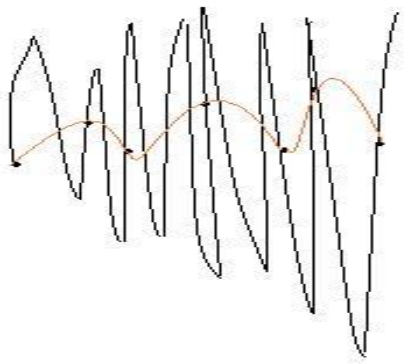
The following picture shows this, where the wave is dark and the interpolant is light.



For the second method of doing Cubic Splines (which only two points are used and the second derivatives are equated), we got much more accurate results still. The cubics were much closer to the original wave (as I had suspected would happen), though there is still some deviation in some spots. Nonetheless, given the graphical interpretation, this could dramatically improve sound quality in certain spots. However in others it might not have as much effect. The darker wave is the sound wave, and the lighter is the interpolant:



However, in the very high harmonics where the wave is oscillating very quickly, either version of this interpolation method wasn't quite enough to fully capture that harmonic (one would have to have a very high sampling rate, which would result in less compression and would essentially defeat the purpose of digital sound). However, the claim is these aren't audible to the human ear. Whether it is or not, it just can't capture it. On the following page is an image demonstrating that (dark line is sound, light is interpolant).



3.4 Conclusion

Unfortunately, given our limited knowledge about the sound files in question, we cannot devise a perfect method for performing an accurate interpolation. The second method of Cubic Spline interpolation provided decent results, though still fails to capture in full the very high harmonics that are lost. The other methods discussed all resulted in the sound quality being compromised further. Knowledge of the original sound file would have to be available for devising the best way of performing interpolation, if that's even possible at all. This would be a topic for further research.

Appendices

Appendix A: Program Header Files

A.a: Linear.h

```
/*
 *  linear.h
 *  Linear
 *
 *  Created by Lynn Blair on 10/13/07.
 *
 */

#ifndef MATHS_INTERPOLATION_LINEAR_H
#define MATHS_INTERPOLATION_LINEAR_H

namespace Maths
{

namespace Interpolation
{



class Linear {
public:



Linear(int n, double *x, double *y) {

        m_x = new double[n];
        m_y = new double[n];

        for (int i = 0; i < n; ++i) {
            m_x[i] = x[i];
            m_y[i] = y[i];
        }

    }
```

```
~Linear() {

        delete [] m_x;
        delete [] m_y;

    }



double getValue(double x) {

        int i = 0;
        while (x > m_x[++i]);

        double a = (x - m_x[i - 1]) / (m_x[i] - m_x[i - 1]);
        return m_y[i - 1] + a * (m_y[i] - m_y[i - 1]);

    }

private:

double *m_x, *m_y;
};


};

};

#endif
```

A.b: Lagrange.h

```
/*
 *  lagrange.h
 *  Lagrange
 *
 *  Created by Lynn Blair on 10/13/07.
 *
 */

#ifndef MATHS_INTERPOLATION_LAGRANGE_H
#define MATHS_INTERPOLATION_LAGRANGE_H

#include <assert.h>
```

```
namespace Maths
{

namespace Interpolation
{



class Lagrange {
public:



Lagrange(int n, double *x, double *y) {

        m_n = n;
        m_x = new double[n];
        m_y = new double[n];

        for (int i = 0; i < n; ++i) {
            m_x[i] = x[i];
            m_y[i] = y[i];
        }

    }


~Lagrange() {

        delete [] m_x;
        delete [] m_y;

    }



double getValue(double x, int l) {

        if (l > 1)
            assert(x >= m_x[0] && x <= m_x[m_n - l + 1]);

        int i = 0;
        for (; x > m_x[i]; ++i);

        double y = 0.0;
        for (int k = 0; k <= l; ++k) {
```

```
            double aux = 1;
            for (int j = 0; j <= l; ++j)
                if (j != k) {
                    double tmp = m_x[j + i - 1];
                    aux *= (x - tmp) / (m_x[i + k - 1] - tmp);
                }
            y += aux * m_y[i + k - 1];

        }

        return y;

    }

private:

int m_n;

double *m_x, *m_y;
};


};

};

#endif
```

A.c: Cubic.h

```
/*
 *  cubic.h
 *  Cubic
 *
 *  Created by Lynn Blair on 10/13/07.
 *
 */

#ifndef MATHS_INTERPOLATION_CUBIC_H
#define MATHS_INTERPOLATION_CUBIC_H

#include <assert.h>

namespace Maths
{
```

```
namespace Interpolation
{


class Cubic{
public:



Cubic(int n, double *x, double *y){
        m_n = n;
        m_x = new double[n + 1];
        m_y = new double[n + 1];
        m_b = new double[n + 1];
        m_c = new double[n + 1];
        m_d = new double[n + 1];

        for (int i = 1; i <= n; ++i) {
            m_x[i] = x[i - 1];
            m_y[i] = y[i - 1];
        }

        if (n < 3) {

            m_b[2] = m_b[1] = (m_y[2] - m_y[1]) / (m_x[2] - m_x[1]);
            m_c[1] = m_c[2] = m_d[1] = m_d[2] = 0.0;
            return;

        }

        m_d[1] = m_x[2] - m_x[1];
        m_b[1] = - m_d[1];
        m_c[2] = (m_y[2] - m_y[1]) / m_d[1];
        m_c[1] = m_c[n] = 0.0;

        for (int i = 2; i < n; ++i) {
            m_d[i] = m_x[i + 1] - m_x[i];
            m_b[i] = 2.0 * (m_d[i - 1] + m_d[i]);
            m_c[i + 1] = (m_y[i + 1] - m_y[i]) / m_d[i];
            m_c[i] = m_c[i + 1] - m_c[i];
        }
        m_b[n] = - m_d[n - 1];

        if (n != 3) {
```

```
            m_c[1] = m_c[3] / (m_x[4] - m_x[2]) - m_c[2] / (m_x[3] - m_x[1]);
            m_c[n] = m_c[n - 1] / (m_x[n] - m_x[n - 2]) - m_c[n - 2] / (m_x[n - 1] -
m_x[n - 3]);
            m_c[1] *= m_d[1] * m_d[1] / (m_x[4] - m_x[1]);
            m_c[n] *= - m_d[n - 1] * m_d[n - 1] / (m_x[n] - m_x[n - 3]);
        }

        for (int i = 2; i <= n; ++i) {
            double T = m_d[i - 1] / m_b[i - 1];
            m_b[i] -= T * m_d[i - 1];
            m_c[i] -= T * m_c[i - 1];
        }

        m_c[n] /= m_b[n];
        for  (int i = n - 1; i > 0; --i)
            m_c[i] = (m_c[i] - m_d[i] * m_c[i + 1]) / m_b[i];

        m_b[n] = (m_y[n] - m_y[n - 1]) / m_d[n - 1] + m_d[n - 1] * (m_c[n - 1] + 2.0
* m_c[n]);
        for (int i = 1; i < n; ++i) {
            m_b[i] = (m_y[i + 1] - m_y[i]) / m_d[i] - m_d[i] * (m_c[i + 1] + 2.0 *
m_c[i]);
            m_d[i] = (m_c[i + 1] - m_c[i]) / m_d[i];
            m_c[i] *= 3.0;
            m_c[n] *= 3.0;
        }
        m_d[n] = m_d[n - 1];

    }


~Cubic(){
        delete [] m_x;
        delete [] m_y;
        delete [] m_b;
        delete [] m_c;
        delete [] m_d;
    }



double getValue(double x){
  assert(x >= m_x[1] && x <= m_x[m_n]);

  if (x >= m_x[1] && x <= m_x[2])
  {
```

```
    double dx = x - m_x[1];
    return m_y[1] + dx * (m_b[1] + dx * (m_c[1] + dx * m_d[1]));
  }

  int i = 1, j = m_n + 1;
  do {
    int k = (i + j) / 2;
    (x < m_x[k]) ? j = k : i = k;
  } while (j > i + 1);

  double dx = x - m_x[i];
  return m_y[i] + dx * (m_b[i] + dx * (m_c[i] + dx * m_d[i]));
}

private:

int m_n;

double *m_x, *m_y, *m_b, *m_c, *m_d;
};


}

}

#endif
```

Bibliography

Bradie, Brian. A Friendly Introduction To Numerical Analysis. Prentice Hall, 2006

Kientzle, Tim. A Programmer's Guide to Sound. Addison-Wesley, 1998

Unknown Author. "WAVE File Format, The Sonic Spot." World Wide Web, accessed 9/30/2007. URL: http://www.sonicspot.com/guide/wavefiles.html