Table of Contents

Abstract

Using a computer to realistically synthesize instrument sounds is an important capability for composers and performers. Much of the sound's authenticity is a result of accurate portrayal of musical inflections available to a performer, such as attacks, decays, and vibrato. A performer would require sound reproduction including these inflections to be reproduced in real time. Composers, however, usually do not require a real-time output, making this type of synthesis very helpful. Described here is a study that concentrates on algorithms for authentic reproductions of musical instruments, specifically wind instruments, that take into account a list of variables including these inflections. The study suggests methods for communicating musical inflections to a computer so that sounds may be synthesized using the discussed methods of reproduction. Unlike much research in this field, the goal is to create a tool that does not function in real-time, but produces realistic output reflecting many different playing styles. As computing speeds increase, these techniques may be applied to a real-time reproduction.

Introduction

A major subset of the study of music synthesis is true reproduction of wind instruments. Brasswinds, closely followed by woodwinds, are perhaps the most complex orchestral instruments to analyze and, as a result, synthesize. Due to this problem, the advancement of synthesis of brasswinds and woodwinds (as well as other orchestral instruments, such as the string family, for the same reason) has progressed much more slowly than that of pianos and electric keyboard instruments. Another reason for the lag is the nature of the interface used with synthesis devices. This interface has traditionally been a keyboard derivative, giving reproduction of keyboard instruments an inherent advantage over other families of instruments. The focus of efforts for synthesis has also gravitated to keyboard instruments because of their wide utility.

Recently, an emphasis has been placed on closing this gap. Many new algorithms have been developed or are being developed to synthesize wind instruments more realistically. The benefits are tremendous for composers, orchestrators, and even performers. While the synthesis focus has shifted, the gap still remains formidable, and much work remains. To be most useful, the final product must be able to reproduce these sounds in a manner such that the original and its synthetic reproduction are indistinguishable to the human ear. Criteria for this product include not only tone and timbre of notes, but reproduction of attack, crescendo and decrescendo, vibrato, tremolo, register shifts, movement between notes, and more.

The general goal of the project is to synthesize one or more wind instruments so that they are indistinguishable from the original. The test for success will be the human

ear itself. If a person cannot recognize the difference, the goal has been achieved. The synthesis method will then be enhanced to include realistic forms of attacks, decays, and other musical inflections used on that instrument. The attack is especially important because it is the most difficult part to synthesize and is closely scrutinized by the listener to discern what instrument is heard. A non-authentic synthesized attack is easily noticed, and these poor reproductions are the primary reason that current efforts to synthesize wind instruments do not sound authentic. Once these inflections are incorporated into the synthesis for the particular instrument, a more practical interface to communicate them to the computer for reproduction can be created. In order to accomplish these goals, several intermediate steps must be reached along the way.

The difference between this and many other synthesis projects is the intended utility of the end result. Much of the study in this area is motivated to produce sounds in real-time for performance. Real-time syntheses are difficult because of the complexity of the sound to be reproduced. As a result, the final product often does not sound authentic. This study is more interested in creating a tool for composers to use that allows them to hear a realistic reproduction of their compositions without a performing ensemble. In this scenario, real-time sound production is not an important concern. However, as computing speeds increase, this project's results might eventually be able to run in real-time.

Background

Sounds produced by musical instruments result from creating a vibration in air that perturbs a membrane in the ear. The brain interprets this vibration as a sound. The vibrations in air are called sound waves. To understand, for example, the difference between the sound wave of a trumpet and that of hands clapping, certain basic concepts of waves must be understood.

One of the most basic waveforms is a sine wave, which describes a pure oscillation at a definite frequency. If this frequency is between about 20 and 20000 cycles per second, or Hertz (Hz), and the amplitude is sufficient, the wave is audible to the human ear[*]. Although humans can hear all of these frequencies, the ear is biased toward a smaller subset, concentrated toward the bottom of the available range. Sounds on the outside limits of this range are often difficult for the ear to resolve to a particular pitch. This concept will become important in the discussion of frequency spectra. In music, letter names are given to certain frequencies, which are called pitches. Several methods for assigning these pitches exist based on certain principles. The most common is the American standard system based on the pitch designated A. The pitch in this system to which orchestras and other musical groups tune their instruments is 440 Hz (often called A 440). The pitch A is not completely unique; in the American system there are several different frequencies with the name A. All pitches designated as A have related frequencies, and are separated by a distance known as an octave. The difference between

[*] The upper limit to this range can decrease dramatically with age, especially in males.

a pitch and the one an octave below is that the lower pitch is exactly half the frequency of the higher pitch. For example, the pitch an octave below A 440 has a frequency of 220 Hz. Similarly, the pitch an octave above A 440 has a frequency of 880 Hz. If they are heard at different times, these pitches' sine waves are distinguished by the brain to be three distinct notes. However, if heard simultaneously, the brain interprets the combination of all three waves as one note at 220 Hz, the lowest of the three A's.

The phenomenon of combining multiple waves as a single note is responsible for each instrument's unique sound. The difference between a single sine wave at 220 Hz and this new wave, which includes pitches at 220, 440, and 880 Hz, is that the two have a different timbre, or tone color. Most instruments produce sounds that combine tens of frequencies, called a frequency spectrum. All instruments that are considered pitched have only specific frequencies in their frequency spectrum[*]. These frequencies are all related to the lowest, or fundamental, frequency, and are called overtones. The first overtone is the frequency in the spectrum immediately above the fundamental; the second is the following frequency, and so on. The frequencies of the overtones are multiples of the fundamental, e.g. A 110 could have overtones of 220, 330, 440, 550, 660, 770, 880 Hz, etc. These overtones correspond different notes all related to the fundamental. From the example above:

- 110 is the fundamental A
- 220 is A one octave above the fundamental
- 330 is E an octave and a fifth above the fundamental

[*] In some cases, non-harmonic frequencies can exist in pitched instruments. These frequencies can make it very difficult for the ear to discern the exact pitch of the instrument. Non-harmonic frequencies also exist in the attack of a note, but die away quickly (see Attacks).

- 440 is A two octaves above the fundamental
- 550 is C# two octaves and a major third above the fundamental;
- 660 is E two octaves and a fifth above the fundamental
- 770 is G two octaves and a minor seventh above the fundamental.
- 880 is A three octaves above the fundamental.

The notes continue to get closer together as the pitch rises. Generally, the most important overtones in a frequency spectrum are the first few above the fundamental. As the frequencies of the overtones increase, the amplitude decreases. This reduction in amplitude is often rather disjunct, occurring between two adjacent harmonics rather than smoothly declining. Some of the richest sounding instruments sound in the range of a few hundred Hz. This leaves room for many overtones before reaching frequencies that are hard for humans to discern. Higher pitched instruments timbres are sometimes harder to recognize because their overtones reach high frequencies after only a few harmonics. Since the amplitude drop occurs so quickly the harmonic spectrum is fairly bare and the sound is not very original as a result.

Two notes with the same fundamental frequencies coming from different instruments can have frequency spectra with dramatically different characteristics. It is the pattern of the overtones that allows the ear to distinguish between an A 440 played on a clarinet, violin, piano, or any other pitched musical instrument. In order to reproduce an instrument's sound, the frequency spectrum of the desired note is required. Once the frequency spectrum has been determined, it can be used as a formula for recreating a waveform that sounds like the original. The frequency spectrum for a musical instrument can change over time, especially at the beginning and the ending of a note. It will later be

important to see just how the frequency spectrum does change; however, to do this, it is necessary to isolate the important frequencies from a portion of a waveform using Fourier analysis. This method can be applied not to only sound waves, but also to light waves, waves in water, and any other type of waveform. An understanding of Fourier analysis is crucial.

Fourier Analysis

Thus far, waves have been discussed only qualitatively as sounds detected by the human ear. However, to understand how to separate frequencies out of a waveform, it is necessary to understand the mathematical representation of waves. The most basic wave is the sine (sin) or cosine (cos) wave. The sin and cos functions oscillate between –1 and 1, and are periodic functions. These cos and sin waves are the building blocks in Fourier analysis for creating and analyzing more complicated waveforms.

An important concept about waveforms is the superposition principle. This principle states that waveforms can be added linearly to each other. In other words, a waveform w and a second waveform x, can be added together to produce a third waveform, y. Let w represent sin(u) and x represent cos(v) where u and v are proportional to time; y would then necessarily be sin(u) + cos(v). Waves may also be represented by complex numbers, e.g. $\cos(x) + i\sin(x)$ or e^{ix}. Both of these expressions are mathematically equivalent. These complex waves use a mathematical trick to represent the phase angle of the wave. The phase angle dictates where on the oscillation time t_0 is defined; for sin the amplitude at t_0 is 0, for cos it is 1 when the phase angle is an integer multiple of 2π. However, in general the starting amplitude could be at any point between -1 and 1. When describing physical phenomena with complex waveforms, it is always necessary to take the real part of the solution after any calculation.

Using superposition, it is possible to represent any waveform as the sum of sin waves of different frequencies. This is important in the case of a waveform for a particular instrument. If the waveform can be represented by the sum of a particular set

of sin waves, then the frequencies that make up that wave can be determined. Representing a waveform by a set of frequencies is an important tool which will be discussed later. It is, however, necessary to discuss a method of calculating these sin waves. This method is called Fourier analysis.

Classical Fourier Transform

One way of describing a waveform is considering an amplitude *h* as a function of time *t*, which can be symbolized as $h(t)$. $h(t)$ is called the time domain, or time space, of a function. The same waveform can be written in the frequency domain, or frequency space, so that the amplitude *H*, which could be complex, is written as a function of frequency *f*, symbolized by $H(f)$, with the restriction that $-\infty < f < \infty$. A Fourier transform equation is used to move from one representation of the function to the other, and is written in these forms:

$$H(f) = \int_{-\infty}^{\infty} h(t) e^{2\pi i f t} dt$$

$$h(t) = \int_{-\infty}^{\infty} H(f) e^{-2\pi i f t} df$$

Since the amplitude of the data is complex in the frequency domain, it is often desirable to look at the power spectrum instead. To do this it is necessary to take the sum of the squares of the real and imaginary parts of the complex amplitude in the frequency domain, creating a consolidated amplitude for each frequency. This can be compared to the square of the amplitude in the time domain. In many cases the amplitude in the time domain can also be complex. As previously mentioned when dealing with physical phenomena, only the real part need be considered. Consequently, we can assume that the

sound waves consist of all real data with the imaginary part being equal to zero. This fact will become important later when Fourier analysis algorithms are used to calculate the Fourier transform. This does not mean that the resultant transform data is all real. It will still be necessary to calculate the power spectrum from the resultant complex data.

$$Total\ Power \equiv \int_{-\infty}^{\infty} |h(t)|^2\, dt = \int_{-\infty}^{\infty} |H(f)|^2\, df$$

Working with Discrete Data

Working with discrete data is somewhat different from working with the continuous data above. Waveforms sampled on the computer are an example of discrete data. For the discrete data in this study, the data is recorded at evenly spaced intervals. Let Δ represent the time interval between samples, or the sample interval. The inverse of Δ is called the sampling rate or sampling frequency. For the purpose of this study, time is measured in seconds (s) and frequency in Hertz (Hz).

One of the drawbacks of discretely sampled data versus continuous data is a lack of resolution. To represent a sin wave of a certain frequency, at least two data points, one at the positive peak and one at the negative peak, must exist to recover that wave. This can be represented by defining the cutoff frequency (also called the Nyquist critical frequency) in the following way: $f_c \equiv \frac{1}{2\Delta}$. In other words, the highest frequency that can be accurately recovered from a discretely sampled waveform is exactly one-half the sampling frequency. The values recovered for frequencies greater than f_c are aliased or falsely translated. The only way to overcome aliasing is to allow at least two samples per cycle for the highest frequency present.

For this application, because the human ear cannot discern frequencies above about 22 kHz, higher frequencies are not of concern in this application. This requires use of a minimum sampling rate of 44 kHz. Both DATs (digital audio tapes) and CDs (compact discs) use sampling rates above this (48 and 44.1 kHz respectively) making them good sampling devices. Some argue that humans can hear frequencies higher than 22 kHz. Proponents of this argument claim that CD reproductions sound somewhat tinny, and a 48 kHz minimum should be used as the recording standard for digital sound.

Classical Fourier Transform for Discrete Data

In order to calculate the Fourier transform of a discretely sampled waveform with sampling interval Δ and N samples, N must be an even number. Since there are only N samples, the output of the transform may only contain N numbers. It is necessary only to output values for discrete frequencies, determined by the equation $f_n \equiv \frac{n}{N\Delta}, \quad n = -\frac{N}{2},\ldots,\frac{N}{2}$. Finally, the integral from the continuous data must be approximated by a discrete sum:

$$H(f_n) = \int_{-\infty}^{\infty} h(t) e^{2\pi i f_n t} dt \approx \sum_{k=0}^{N-1} h_k e^{2\pi i f_n t_k} \Delta = \Delta \sum_{k=0}^{N-1} h_k e^{2\pi i k n / N}$$

The last summation in the equation is the discrete Fourier transform of N points of value h_k, which is represented by H_n: $H_n \equiv \sum_{k=0}^{N-1} h_k e^{2\pi i k n / N}$. It is also possible to recover the discrete waveform from the discrete transform with a very similar calculation: $h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n e^{-2\pi i k n / N}$. There is a method similar to the continuous method for representing the power series:

$$Total\ Power \equiv \sum_{k=0}^{N-1} |h_k|^2 = \frac{1}{N} \sum_{n=0}^{N-1} |H_n|^2$$

Fast Fourier Transform

The classical Fourier transform method is somewhat tedious. In order to be useful, the computation between time and frequency space needs to be fast. A computer is required to do a Fourier transform quickly. The use of a computer requires the waveform data to be digital, or discretely sampled. The algorithm to compute the Fourier transform described above using the discrete method appears to be an $O(N^2)$ process, making the number of calculations required to process the data proportional to the square of the length of the data set. For example, if processing a data set of length 20 takes 30 minutes, an array of length 40 would take 2 hours.

In 1942, Danielson and Lanczos developed an algorithm that was popularized by J.W. Cooley and J.W. Tukey in the mid-1960's (Sundararajan, 1997). This algorithm allows the discrete Fourier transform to be calculated in $O(N \log_2 N)$ operations and is called a fast Fourier transform. This technique saves significant computer time. For example, a transform with 10^6 samples requires a computer to perform 10^{12} calculations using the classical discrete Fourier transform method, and $2*10^7$ calculations using the fast Fourier transform method. The result makes transforms like this possible in seconds rather than weeks.

The key to the speed of the fast Fourier transform, or FFT, is that a discrete Fourier transform of length N can be rewritten as the sum of two discrete Fourier transforms of length N/2, one with all the even elements and one with all the odd elements. This process can execute recursively, taking the two new groups of N/2

elements and dividing them each into two new groups of N/4 elements. The groups can now be classified. One group contains the evens from the first division and evens from the second division, and is called even, even. The other three groups can then be called even, odd; odd even; and odd, odd. The original N must be a power of 2 so that the process can be continued until there is only a set of transforms of one element each. The Fourier transform of one element simply copies its value into an output slot. The number of the output slot that the final transform copies into is a binary number created from its even and odd labels. Taking the order of evens and odds that characterize each new transform of length 1, reversing the order, and substituting even=0 and odd=1 results in the binary number for the slot in which to put the value. See *Numerical Recipes* (Price, 1992).

Fast Fourier Transform of Real Input Data

The fast Fourier transform method allows for complex input data. In this study, all of the input data will be real data. Rather than inputting zeros for all the imaginary values, the data can be repacked such that the real part of the second half of the data is stored in the imaginary slots for the first half of the data. The transform is now half as long, which eliminates half of the necessary calculations to compute the transform. The output can then easily be reordered such that it is identical to the output of the original unpacked fast Fourier transform[*].

Other Methods

Until recently, Fourier analysis was the generally accepted method for doing waveform analysis. However, a new method of waveform analysis with applications in acoustic analysis and synthesis is currently being developed. This new analysis tool is called wavelet analysis. Wavelet analysis processes a waveform and generates wavelets. Like Fourier analysis, wavelets convert the data from time-space to frequency-space. However, wavelet analysis does not use sin or cos functions as a basis for the frequency space of the wave like Fourier analysis, but uses any functions that are localized in space. Whether it is analyzing gross features or small features, wavelet analysis has the capability of choosing an interval of appropriate length to match.

A major advantage to this technique is for compression of audio data. Since Wavelet analysis can use an arbitrary set of functions, it is possible to choose both long, low frequency functions and short, high frequency functions. This creates wave envelopes, or packets, which mimic the behavior of actual audio data. It is easy to use very few functions to reproduce very complex audio data[*]. This method was used originally in MPEG audio compression. Its most recent version, MPEG layer 3 compression, compresses digital audio at a rate of 12:1. Attempts have been made to apply wavelet analysis to acoustic synthesis, but the efforts have been largely problematic. One reason for this is due to a major disadvantage of this technique--its

[*] In actuality, there is a slight loss of generality, namely the loss of all the negative frequencies present in the fast Fourier transform. These frequencies are a mathematical trick handle complex data. In this study it is not necessary to consider these negative frequencies since only real data is used.
[*] This is not to imply that in order to use Wavelet analysis, it is necessary to look at an entire wave envelope. This method can apply to portions of the wave envelope just as easily.

difficult calculations. Its analysis techniques can take a lot of computer time, especially for complicated audio data.

Wavelet analysis uses a process that is very similar to the Fast Fourier Transform to move from time-space to frequency space. The number of calculations required for this process depends on the functions using to convert to frequency space. Several families of functions have been studied, such as the Mallat and Daubechies families. However, the large calculations result from determining which set of functions yield the best results for analysis.

Part of the reason that wavelet analysis has not been widely used at this point in music analysis and synthesis is the large amount of computer time needed to make these calculations. However, this may change quickly as computer speeds increase. Since wavelets can study an arbitrarily big or small time-interval of the waveform. The small time intervals make wavelets an excellent choice to study the attack portion of the note where frequencies change so quickly with time. This sort of study may become very prevalent in the field of music analysis and synthesis in the near future.

Software

After the decision had been made to use the fast Fourier transform algorithm to analyze and reconstruct waveforms, it was necessary to find or create a software package which utilized the fast Fourier transform and had the desired flexibility of input and output parameters and organization. Another very important criterion was that the pre-existing package should run the fast Fourier transform calculations quickly and do file input and output operations efficiently. This made a package written in a low-level language such as C or Fortran more useful than an extension of Maple or Mathematica, or even a higher level language like C++ or Java.

A pre-existing package that satisfied all of the criteria was not available. What *was* available from several sources, however, were routines that could be incorporated into code in C or Fortran that had previously been written. C was chosen because of several advantages over Fortran. Although C is not significantly faster than Fortran, it enjoys much wider platform support than Fortran, making it easily portable from platform to platform. C code is also backward compatible with different compiler versions, which is not the case with Fortran. Finally, a familiarity with similarly styled languages such as Perl and Java cemented the decision, but did not eliminate the need to learn C.

Once the language was chosen, the source for the actual fast Fourier transform routine was required. Finding well-documented code from a free source such as the Internet was very difficult. Much of the code found there did not compile, and some of it was so long that it was not efficient. Some of it, like the code found in several books specifically about Fourier transforms, was so poorly documented that the code designed

to run the transform, and the code used for handling other tasks were indistinguishable to a C novice. The most extensively documented code was found in *Numerical Recipes in C: The Art of Scientific Computing* (Press, 1992). It contained well-commented code and a complete explanation of the theory behind it. It was still a challenge to completely understand the core fast Fourier transform routine, because intuitiveness of the operations was compromised in favor of speed.

The code and documentation in the book were far from perfect. The C code had been ported from old Fortran code. Because this was not typical C code and contained remnants of Fortran, it was very difficult for a C novice with no Fortran experience to work with. Much functionality needed to be added to the core fast Fourier transform routine to make it a good tool for this project. First the decision needed to be made about what form the data in and data out should follow.

It was most logical for the software to read the amplitude data out of a standardized audio file format. The Microsoft wav file format, which can handle both 8 and 16 bit files, stereophonic or monophonic data, and sampling rates up to 48 kHz was chosen. The reason wavs were selected is that they have become an industry standard, and documentation on the file format seemed to be plentiful. Unfortunately, the documentation, while plentiful, was never authoritative and often contradictory. Piecing together a viable model of the file format required a combination of three or four sources. Formatting the output correctly was another issue. It was written out to a data file that was compatible with Gnuplot, a freeware plotting program available for UNIX platforms. The resultant code from these criteria yielded a software package that could read a wav

file, take the fast Fourier transform, and output data that could be displayed and analyzed with Gnuplot.

After the basic program was functional, several features were added to make the it more useful. A modification was included that increased the efficiency of using only real input data. This modification was outlined in *IEEE Transactions on Signal Processing* (Sundararajan, 1997). The input data array originally needed to have space for a real and an imaginary component for each sample, regardless of whether the complex value was zero. However, if one is only interested in the positive frequency amplitudes, then by reordering the real data and eliminating the complex values, it is still possible to take the fast Fourier transform of this half-length data array and have no loss of generality. The real and complex values for the positive frequencies and 0 are returned. The re-ordering time is fairly insignificant compared to the time needed by the fast Fourier transform. Just over half the original calculation time is required to do the transform.

In addition, the program needed the ability to do multiple transforms in succession and output the data so that it could be viewed in a 3-dimensional format, one axis representing the frequency range, the second the time, and the third representing the amplitude. Also, one needed to be able to specify the approximate time interval for each transform, the number of overall transforms to take, and the initial time-offset from the beginning of the file. Unfortunately, it is not possible to arbitrarily set the time interval because the number of samples per transform must be a power of two.

After creating a method for interpreting the samples on the computer, it was necessary to acquire samples to analyze. Two methods exist for acquiring samples for

analysis: recording the samples with an appropriate digital medium or using prerecorded samples that fit specifications which must be determined.

Instrument Sampling

Once we are able to take the Fourier transform of a waveform, a method for saving the waveform on the computer is needed. It is logical to choose an existing industry standard, especially one that is easily interpreted. Microsoft's wav file format is utilized in this study because it has the most easily available documentation and most existing software supports this file format. The wav format contains a file header that relates important information about the amplitude data around which a filter to interpret the file can be written. The rest of the file contains raw amplitude data, which makes it a prime candidate for use in this project. Most programs that read digital audio from either CD-ROM or DAT directly onto the computer support this format.

For this project, sampling rates of 44.1 kHz and 48 kHz are used. The amplitude data is stored using16 bit numbers (or 16 place binary numbers) that allow a range of values from –32768 to 32768. This allows much more flexibility and accuracy than 8 bit values, which only allow a range of –128 to 128. For simplicity, this study uses monophonic data, or one channel. For stereophonic data, one channel should be selected or both channels should be added linearly using the superposition principle.

Specifications

This study will take into account the effects of several musical characteristics on the frequency spectrum of the samples. It should be possible to look at the frequency spectrum and the change of the frequency spectrum over time to determine its behavior. The frequency spectrum for each variable should be unique. In some cases it could be

related to other frequency spectra and correlations may be discovered. It should also be possible to combine the effects of different variables and reproduce two or more of these effects at once.

As mentioned before, each instrument produces a characteristic frequency spectrum that distinguishes it from other instruments. However, if one note of an instrument is examined, the frequency spectrum changes over time. The beginning of the note often contains very strange behavior in the frequency spectrum. Overtones may arbitrarily fluctuate violently. Frequencies other than overtones may also be prevalent, and are likely to be changing rapidly as well. This region of the note is called the attack, or articulation. The fluctuations in the overtones and other frequencies are called transients. Their behavior is dictated by the initial conditions of the physical system. These transients quickly die away to leave the "tone" of the instrument. The part of the sound without transients is called the steady state portion of the note, where the initial conditions are irrelevant. In instruments capable of sustaining a note, a third portion of the note, the release, exists. The release may be as simple as a fast damping of the steady state harmonic spectrum, but transients may again be introduced.

I. Steady State for Sustain-Capable Instruments

In musical instruments, and generally in any physical system, the steady state portion of the note is the easiest part of the waveform to calculate and analyze. Sound synthesis is no exception. For this reason, reproduction of the steady state portion will be the starting point. In application, it is necessary to use a slightly different technique when

dealing with a sustain-capable instrument versus a percussive instrument. For a sustain-capable instrument, once a set of samples from an instrument has been acquired, it is possible to look at the fast Fourier transform of this waveform's steady state solution using the computer program described above. For this purpose, it is important to select a sample that sustains a note at constant volume and pitch. The portion of the note between the attack and the release should be analyzed to avoid the transients. Since the note could waver slightly (as is the case with wind instruments and string instruments played with a bow) due to imperfections of the performer, the number of samples should be fairly large to achieve a good average value and good accuracy. The transform that results is the harmonic recipe of that particular note on that particular instrument.

This procedure should be followed to get a harmonic recipe for all the notes on the compass of the instrument, keeping all variables as close to the same as possible. The test for this criterion should be interpretation by the human ear, not a measure of the waveform's amplitude. Once all of these samples have been run through the phase vocoder program, it is possible to compare the harmonic recipes of all the notes on the compass. Of course, all of the recipes are different because they start on different fundamental frequencies. The spacing between overtones is different as well, because overtones are generally integer multiples of the fundamental frequency. However, if one speaks about a harmonic recipe in terms of amplitudes of fundamental, first overtone, second overtone, etc. It is possible to see relationships in the harmonic recipe of notes on the same instrument.

To eliminate the need to discuss the raw amplitude of a particular frequency, it is useful to talk about the percentage of total amplitude that is contained in the frequency.

This percentage is calculated by dividing the amplitude of the frequency of interest by the sum of the amplitudes of all the contributing frequencies in the harmonic recipe. Generally, adjacent notes on an instrument have relatively similar harmonic recipes; however, changes in register on an instrument can dramatically change the harmonic recipe between adjacent notes. Examples of this include changing partials on a brasswind, changing registers on a woodwind, or changing strings on a violin. Working within these restraints, it may be possible to create an algorithm for reproducing the steady state portion of all the notes on the compass based on slight modification of the harmonic recipe of one standard note or notes chosen from each register or grouping.

As mentioned before, an instrument will have imperfections when played by a person, and a synthetic reproduction using these techniques will be "perfect." However, in this case, perfect is not desirable, because the pitch will sound stagnant. It is necessary to replicate these imperfections through some sort of randomization algorithm which slightly changes the pitch and dynamic of the sustained note.

II. Steady State for Non-Sustained Instruments

For pitched instruments that are not sustainable, such as pitched percussion, a piano, stringed instruments played pizzicato, etc., reproducing the steady state part of the wave is a little more complicated. Rather than taking a fast Fourier transform over a large time scale, it is necessary to take a grouping of short time scale transforms to determine how the harmonic recipe changes over time. This adds an additional variable, decay. During the decay, the overtones will decay at different rates, and these rates will be

specific to each pitch on the compass. Now, not only can the harmonic recipe change according to the pitch being considered, it can change with time. For a violin playing pizzicato, three variables, pitch, timbre of the register, and harmonic decay rate must be introduced just to analyze the steady state portion of the waveform.

III. Dynamics

A very important part of musical composition is the use of dynamic contrast, or volumes ranging from extremely soft to extremely loud. When the dynamic is changed, the harmonic recipe of a note can change dramatically or very little depending on the instrument. To study this, samples must be taken at several different volumes and their steady state harmonic recipes analyzed. Again, an algorithm must be created to reproduce the effect. This algorithm may be incorporated with some of the techniques of the above algorithm because decrescendo and decay are similar phenomena.

IV. Attack

As mentioned before, the steady state portion of a note is likely the easiest part to reproduce. However, this synthesized steady state portion is almost useless without the attack. The attack has complicated behavior. During this portion of the note the transients are clearly visible. Because of this, it is very difficult to synthesize. A compounding problem is that the attack lasts only for a very short time and the transients die away quickly. The fast Fourier transform loses accuracy when the time interval and,

consequently, the number of samples, are reduced. It is for these reasons that knowledge of synthesis of the attack lags behind that of every other aspect.

In addition, an infinite number of different kinds of attacks can occur according to the particular instrument. Our analysis will be limited to several accepted categories of attack. The attack containing the fewest transients is not actually an attack, but a transition from one note to another called a slur. On a wind instrument this corresponds to switching notes without stopping the airstream. This form of attack generates very little noise between notes and therefore very little in the way of transients. On those same lines, the breath attack or soft articulation tries to emulate the slur in the sense that the steady state portion of the note occurs quickly with very little in the way of attack or transients. It might be more accurate to describe this as an increase in volume of the steady state form rather than an attack. Regular articulation, where the note has a definite beginning, and a more accented attack will be difficult to synthesize. Both will have transients, the more accented attacks having less predictable behavior.

V. Releases

Once the attack and the sustained portion of the note have been accurately synthesized, the end of the note must be synthesized. This effect should be quite similar to the dynamic effect because the effect of a release is almost equivalent to a fast decrescendo ending in silence. However, since different techniques are utilized for releases than decrescendos, the behavior of the frequency spectrum will likely be at least slightly different as well. Several different kinds of releases are employed and will need

to be synthesized. Slow releases will act more like dynamic changes rather than releases. Fast releases are generally utilized for shorter notes. Because of this, a fast release will interface more with the attacks and the steady state portion of the wave may not be utilized.

VI. Steady State Effects

A complete note can be synthesized with the above parameters. However, on some instruments, sustained pitches have characteristics that affect the steady state portion of the wave. Without these nuances, the pitches will not sound authentic. Wind instruments like the oboe and trumpet utilize a periodic raising and lowering of a pitch called vibrato. This change in pitch will slightly change the harmonic recipe toward the adjacent notes above and below. In addition, an instrument's vibrato will never fluctuate between exactly the same frequencies, much like a steady pitch does not always hold exactly the same frequency. As with the steady sustained pitch, it will be necessary to add a bit of a random effect to make the pitch sound more realistic. In addition, players may use vibrato from the beginning of a note, or only on part of a note. It could be added after the note is begun, ended before the note is completed, or both. It is necessary to be able to add vibrato gradually, since that is a technique used by musicians. One other variable is the amplitude of the pitch change. Again, this could change throughout the duration of the note.

Some instruments, such as the flute and the vibraphone, use an effect called tremolo. Like vibrato, this is a periodic effect. However, instead of changing the pitch

slightly, it changes the volume slightly. As shown in the discussion on dynamics, this also changes the harmonic recipe. Like vibrato, this effect can be added for a whole note or a fraction of a note, can change frequencies over time, and can change levels of dynamic contrast.

Sample Analysis

Once samples that meet the specifications have been acquired, it is possible to begin to examine how these different variables affect the frequency spectrum. Some of the hypothesized effects on the harmonic recipe that correspond with a particular variable have already been mentioned. To keep the variables ordered, an N dimensional space can be initialized such that N corresponds to each variable mentioned above. The first step is to try to produce an algorithm for each variable that modifies a base frequency spectrum. If the variables are uncorrelated, the modifications to the frequency spectrum can be combined. In order to test for independence, the waveform can be assumed to have independent variables. A synthetic waveform can be created with this assumption. If the reproduction sounds authentic and has the planned characteristics, the analysis has succeeded. If the reproduction is unacceptable, it may be necessary to explore a dependent relationship between these variables and find ways to adjust the one-dimensional algorithms to fit two, three, or more dimensions.

Sample Type 1: Tuning Fork

The tuning fork is a simplistic sound producer. It is designed to produce a very predominant fundamental, and few small amplitude higher harmonics. However, the tuning fork is not a completely trivial case. It still has a harmonic recipe with an attack and a decay. The fundamental and higher harmonics decay at different rates, the higher harmonics decaying much faster than the fundamental. This quickly eliminates more of

the already few higher harmonics. This behavior is useful in practice because the tuning fork is designed to produce exactly one pitch. Large amounts of overtones, especially with significant amplitudes and overtones that are not perfect intervals with the fundamental, are hard for the brain to process and often make the fundamental pitch less clear. It is for that reason that it is harder to tune an instrument to, for example, a clarinet than to a tuning fork.

Because of this simplicity, this study begins with an analysis of a 256 Hz tuning fork. The tuning fork can be sampled over a fairly long time-scale, on the order of 30 to 60 seconds. This time period includes the attack, which occurs when the tuning fork is struck, through to the decay, where the oscillation begins to be damped. The sample is then run through the fast Fourier transform program to isolate the important frequencies in the harmonic recipe. As predicted, few higher harmonics appear above the fundamental. In the case of the 256 Hz tuning fork, spikes appear at 256, 512, 768, 1560 Hz.

The next step is to see how the amplitude of each of these four harmonics behaves over time. To accomplish this, several iterations of the fast Fourier transform must be taken over time. Each transform iteration should use a fairly small time-scale compared to the sample's overall time-scale. Once the sequence of transforms has been taken, it is possible to see how the frequency amplitude changes over time. Since only a few distinct frequencies contribute significantly to the harmonic recipe, it can be useful to filter out the other non-contributing frequencies, thus making it easier to see what happens to the important frequencies as time progresses.

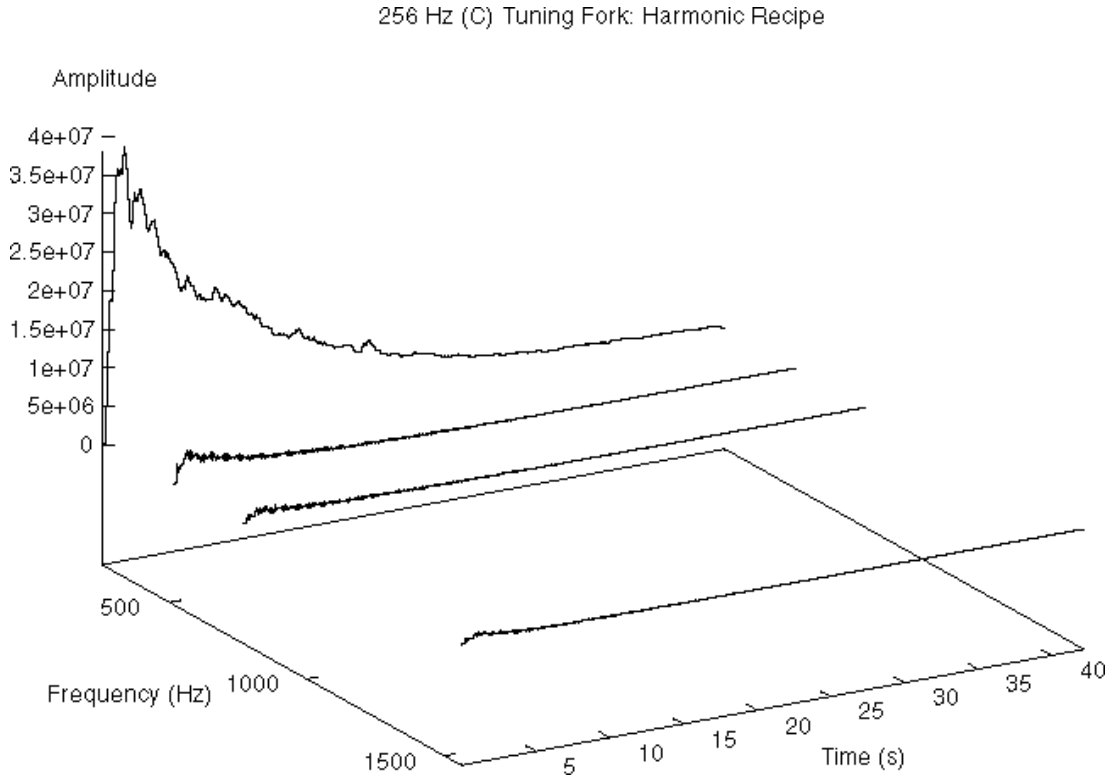

Figure 1. The fundamental and first three harmonics. Unimportant frequencies known as noise have been eliminated to clarify the graph. The higher order harmonics decay more quickly than the fundamental, and have much smaller initial amplitudes. The fluctuations apparent in the fundamental are a product of the sampling technique.

Figure 1 displays the four important frequencies as a function of time and amplitude. It is clear that the fundamental has the most significant contribution to the sound as well as the longest decay time. The amplitudes corresponding to the other frequencies dissipate quickly and contribute much less significantly to the total sound. Their contribution is still important because they help create the tuning fork's timbre during the attack portion of the sound. Without them, a realistic sound could not be achieved. To properly reproduce this sound, each harmonic must be isolated. It is then possible to try to fit a curve to the experimental data. In this case, an available software package, called *Physica* by TRIUMF Software, can be used. It can fit variable parameters

to an assumed equation. It requires looking at each isolated harmonic and guessing an

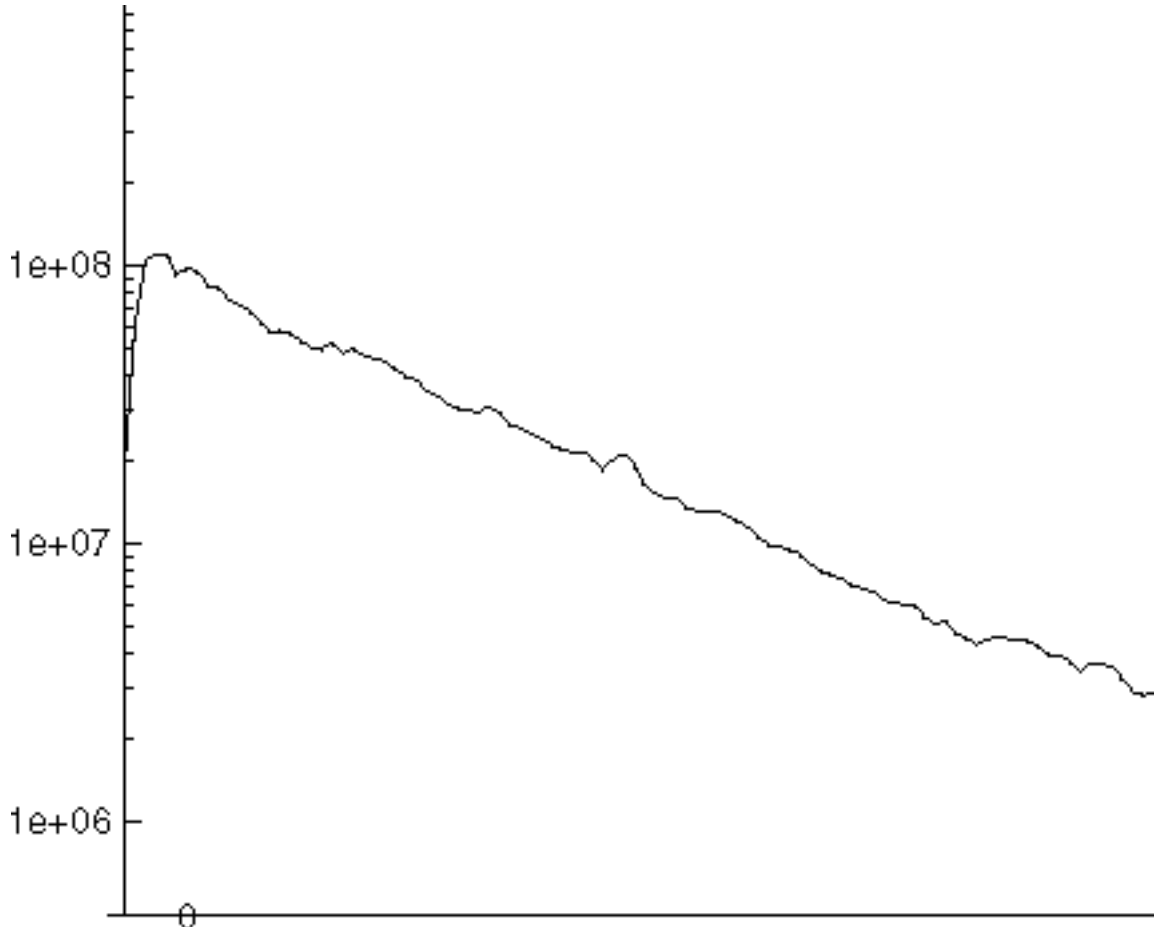

Figure 2. Logarithmic Plot of Amplitude versus Time for 256 Hz Tuning Fork Fundamental. Ignoring noise, the decay appears to be a smooth straight line, corresponding to an one term exponential decay.

equation that will fit.

Figure 2 shows a logarithmic plot of the fundamental frequency for the 256 Hz tuning fork. The decay appears to be a straight line on this plot, signaling that an exponential decay equation should be used to fit the data. The line representing the decay does not have any obvious kinks in it, which suggests that it will likely be possible to represent this line with only one exponential term. The rise at the beginning can be represented by t^a. The proposed curve is as follows: $p_1 t^{p_2} e^{\frac{-t}{p_3}} + p_4$, where p_x are the

Frequency (Hz)	p1	p2	p3	p4	p5	p6
256	2.64E+07	1.0063	5.0117	3.57E+06	7.795	1.061
512	7.72E+06	0.19955	3.794	1.78E+05		
768	4.80E+06	1.0729	1.0543	1.33E+05		
1560	1.11E+07	1	1.049	1.91E+05		

Table 1. *Physica* Fit Parameters for 256 Hz Tuning Fork Harmonics

variable parameters. *Physica* required a few extra parameters before it was able to fit the experimental curve for the fundamental frequency. The final form of the equation is:

$$p_1 t^{p_2}\left(e^{\frac{-t}{p_3}} + p_5 e^{\frac{-t}{p_6}}\right) + p_4 .$$

The higher harmonics must also be approximated. *Physica* was able to fit the higher harmonics with the simpler version of the proposed equation. The values for each

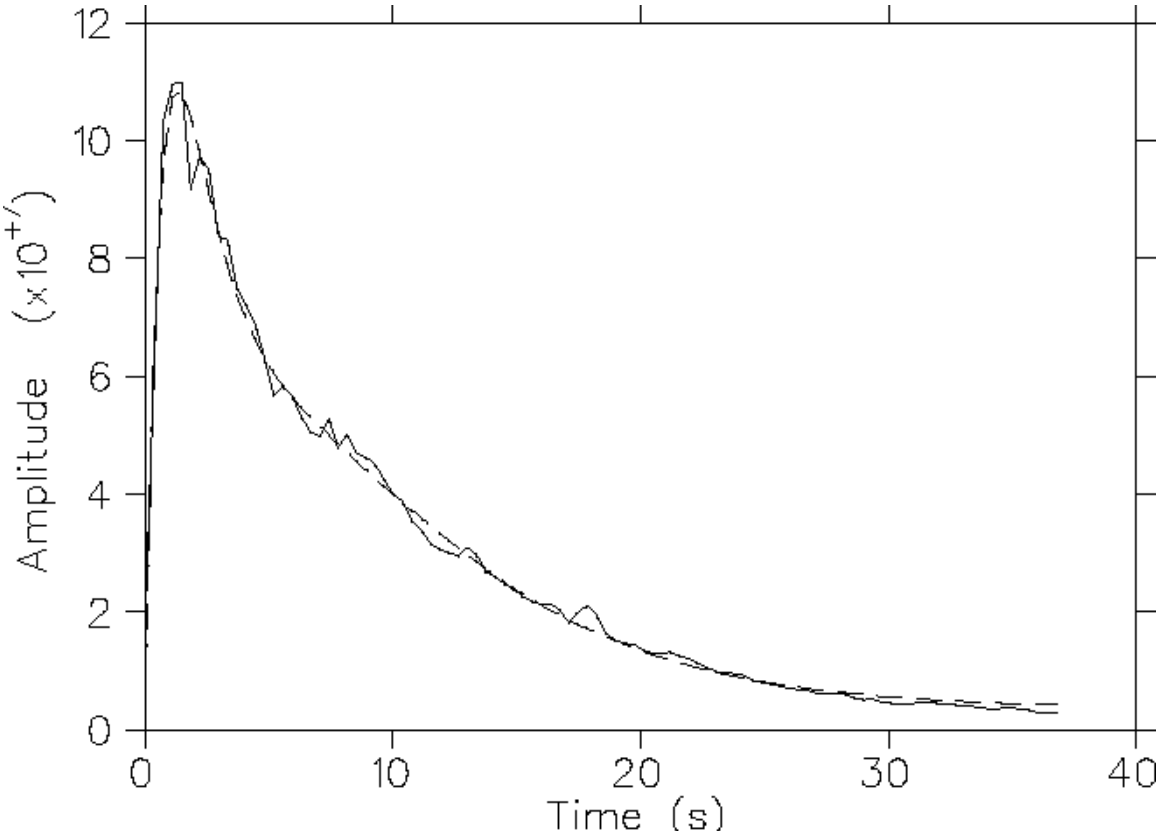

Figure 3. Fundamental (256 Hz) Experimental and Fitted Data. This frequency comprises most of the sound produced by the tuning fork. It decays much more slowly than the higher harmonics. The tuning fork is designed with this characteristic so that it is easy for use in tuning an instrument.

of the variables is listed in Table 1. Figures 4-7 show the experimental data for each harmonic and the fit that *Physica* was able to produce.

After the equation parameters have been fit with *Physica*, it is now possible to recreate the sound. Using fairly simple C code, a discretely sampled waveform can be produced from the equations and saved in the wav file format. The equation fitted for each particular harmonic is now multiplied by $\sin(\omega_n t)$ where $\omega = 2\pi f$ and n corresponds to the harmonic (0 for fundamental, 1 for first, …). The frequency f is the first number in Table 1. It is important to note that the parameter p_1 for these equations is the overall amplitude factor, and p_4 is just a zero offset. Thus, the ratio of the p_1 values for each

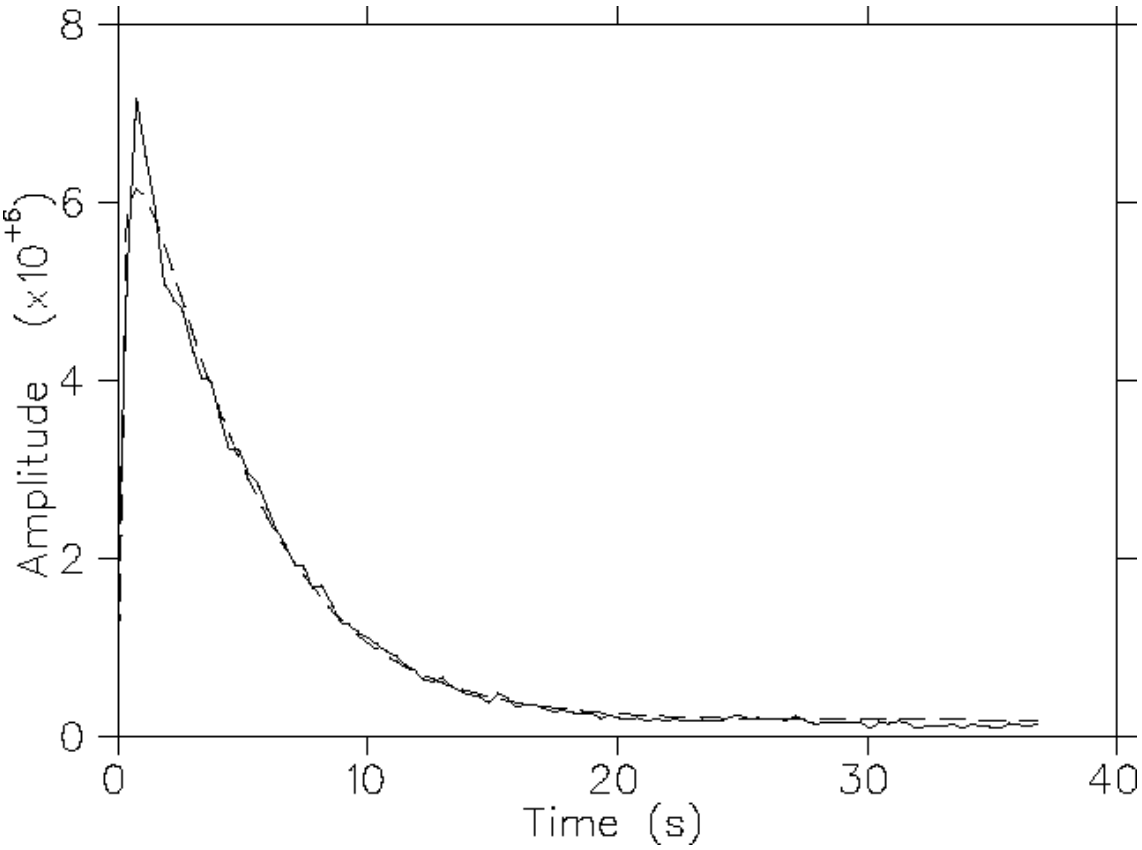

Figure 4. First Harmonic (512 Hz) Experimental and Fitted Data. This frequency decays much more rapidly than the fundamental, and has a lower peak amplitude. By 20 seconds, it has almost completely died away.

harmonic is more important than the actual values fit by *Physica*. The zero offset can also be disregarded in a sound reproduction because it corresponds to the noise both in the recording and in the fast Fourier transform program. The actual synthesized sound with these data did sound realistic.

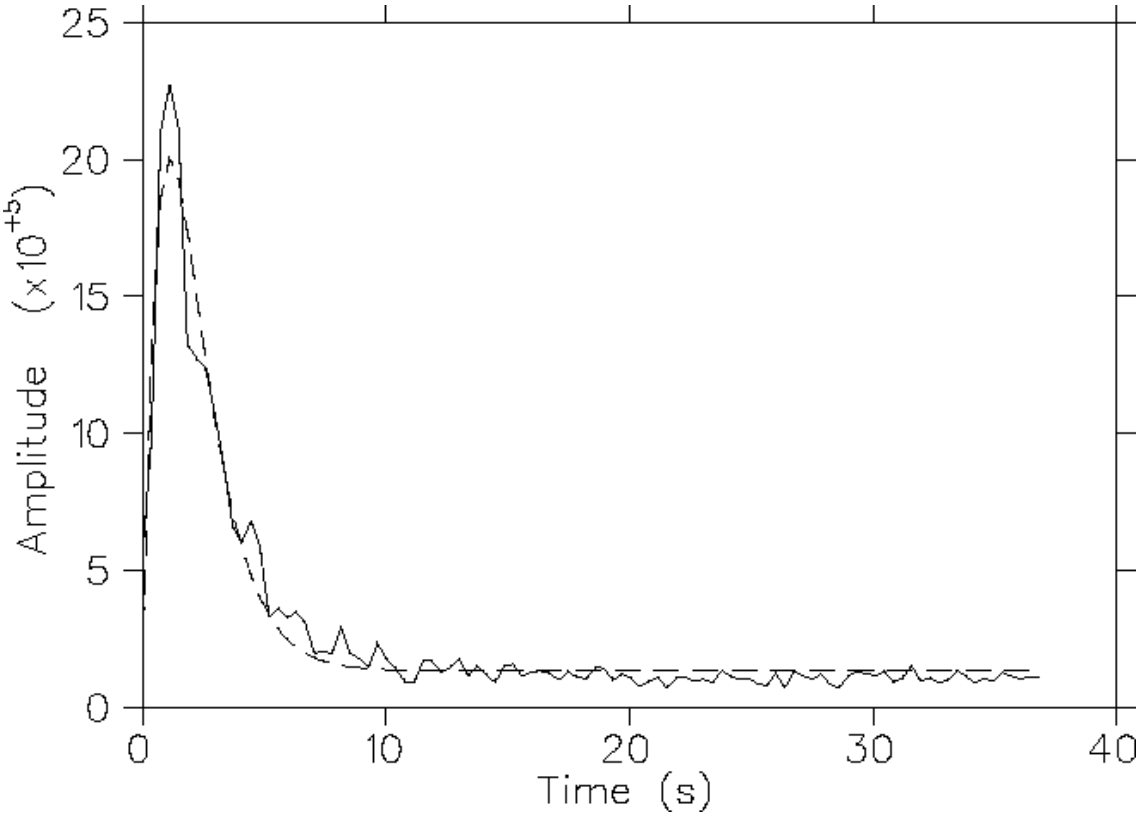

Figure 5. Second Harmonic (768 Hz) Experimental and Fitted Data. Again, the fit is good and the decay is exponential, falling off more quickly than the first harmonic.

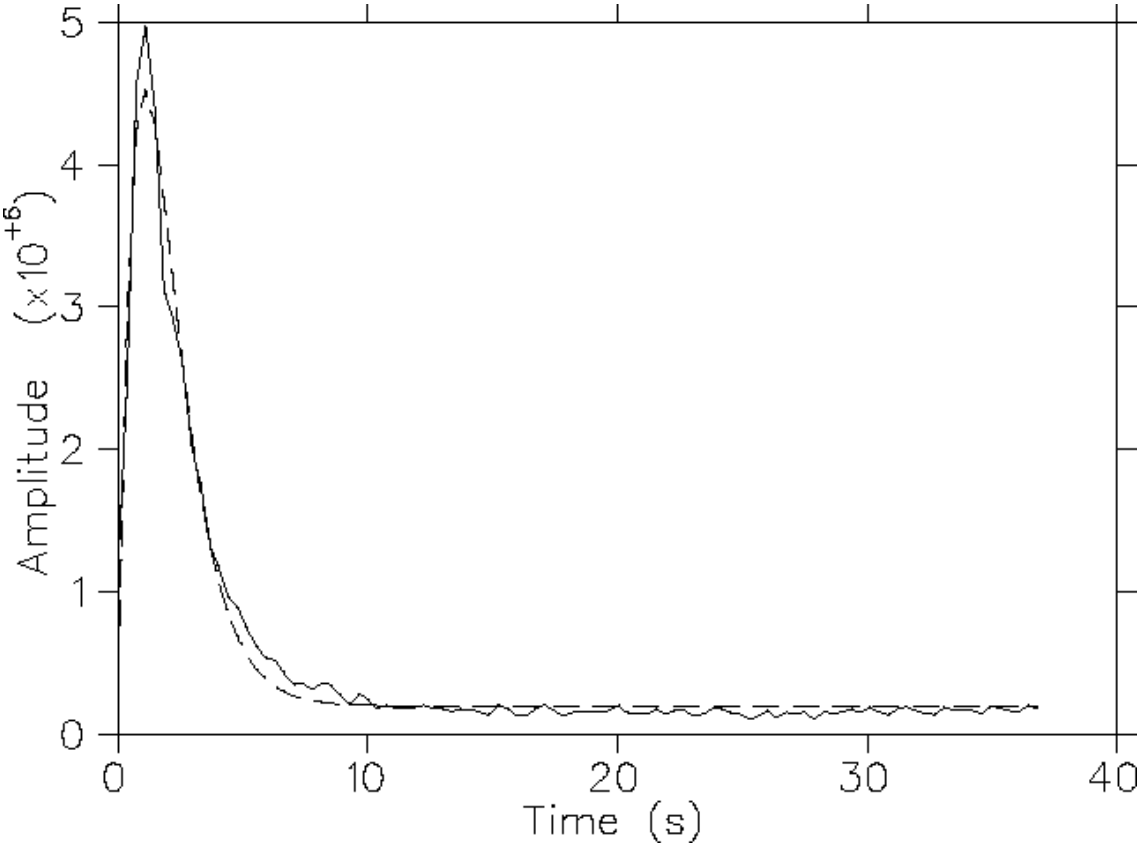

Figure 6. Third Harmonic (1560 Hz) Experimental and Fitted Data. Although it has less amplitude and similar decay to Figure 5, it sounds louder to the human ear because of its higher frequency.

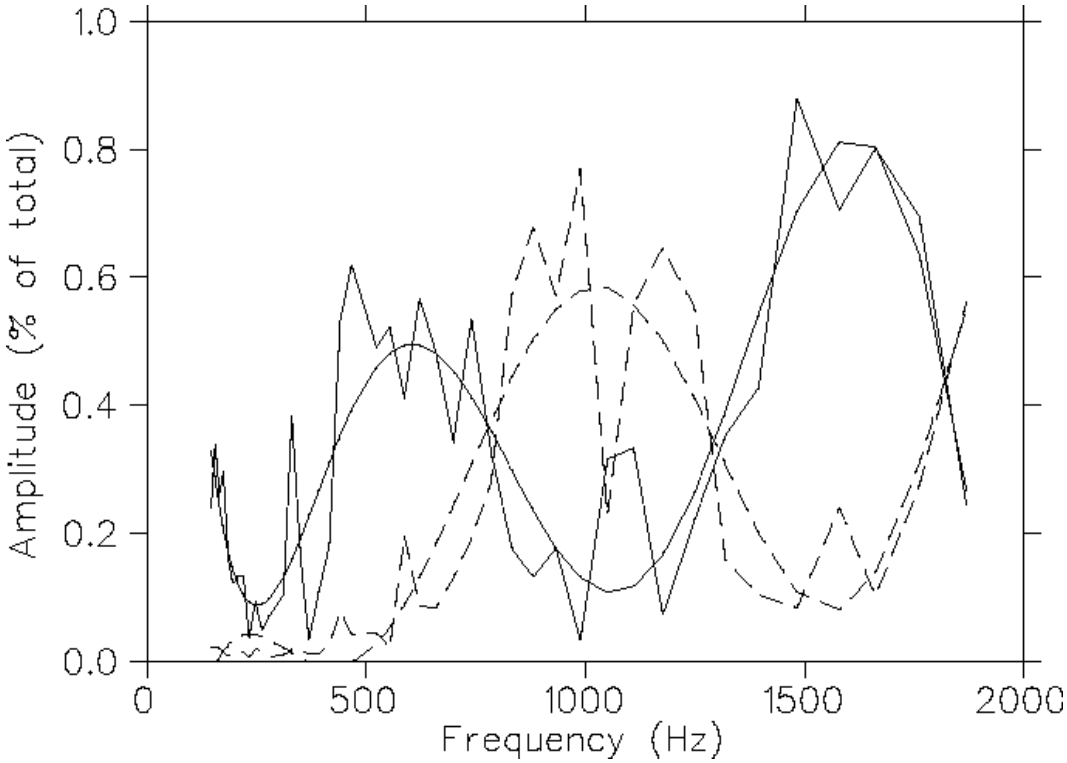

Figure 7. Frequency Spectrum of F in Chalameau Register. This note has many higher order harmonics that contribute to its timbre. As a result, the sound is rich, full, and dark. It is notable that the highest contributing frequencies are fundamental, second harmonic, fourth harmonic, etc. This is the classic pattern for the resonance of an closed cylinder or tube.

Sample Type 2: Clarinet

Unlike the tuning fork, the clarinet has a fairly complicated harmonic recipe. Synthesizing the steady state portion of each note individually, however, is fairly simple. Again, it is only necessary to take one fast Fourier transform to look at the steady state portion of a note. For a fairly homogeneous note, or in this case a note sustained without any musical inflections, taking the fast Fourier transform over a fairly long time scale (a few seconds) is helpful because it yields a very precise transform. A long transform on a note with inflections such as vibrato or crescendo could cause the transform to be less accurate. The resultant transform can be clarified by eliminating noncontributing frequencies, i.e., those that are indistinguishable from zero. However, the transform will

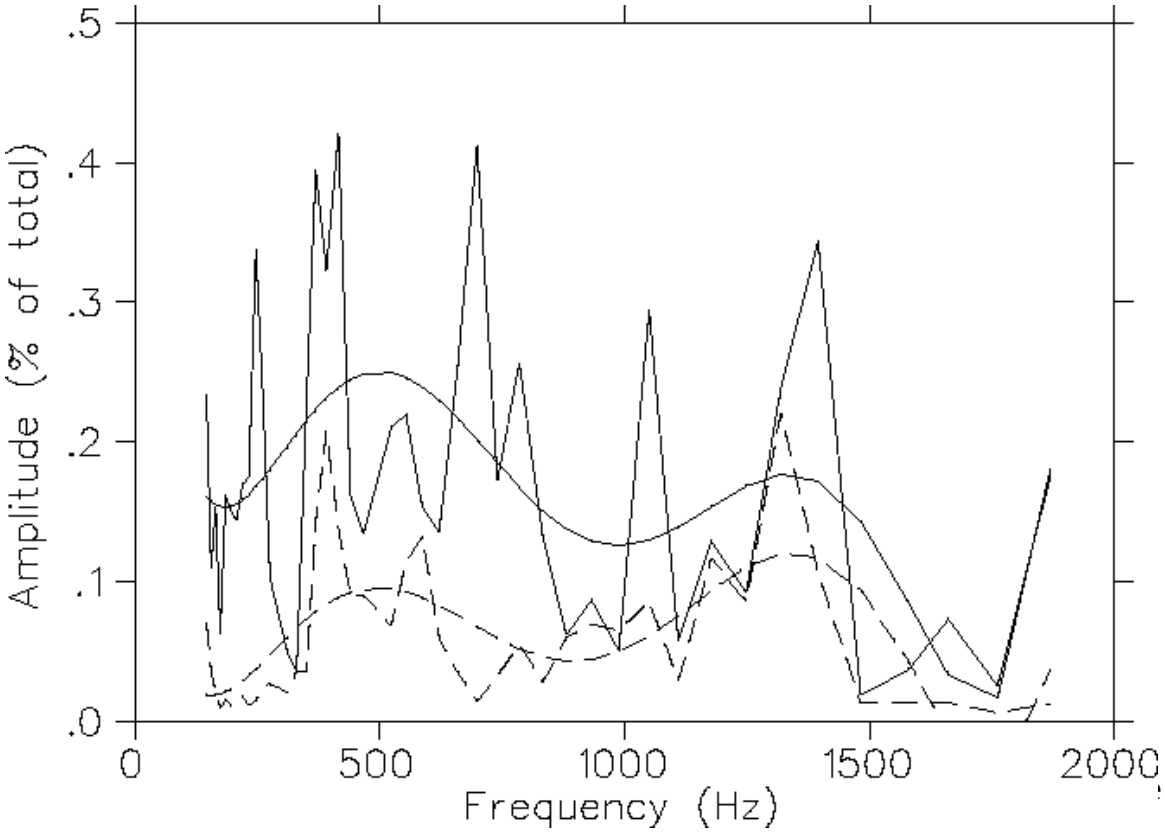

Figure 8. Frequency Spectrum of F in Upper Register. This graph shows fewer harmonics than Figure 7. The fundamental frequency contributes a huge majority of the sound for this note. The tone reflects this by sounding less full and rich than the chalameau register.

$\sum_n Au_n \sin(\omega_n t)$ where A is the overall amplitude and u_n is the overall percentage of amplitude for that harmonic compared to all the harmonics in each recipe. Figures 7-9 show a few characteristic harmonic spectra at different points on the clarinet compass.

Once it is possible to reproduce the steady state portion of the note from the harmonic recipe of that note derived from experimental data, it is necessary to find relationships between the recipes. By analyzing the experimental data of all the notes, an algorithm which reproduces each note's harmonic recipe based on its relationships with other nearby notes or note groupings can be created. Figure 10 shows the experimental data for the fundmental (solid line) and first harmonic (dashed line) for the clarinet over

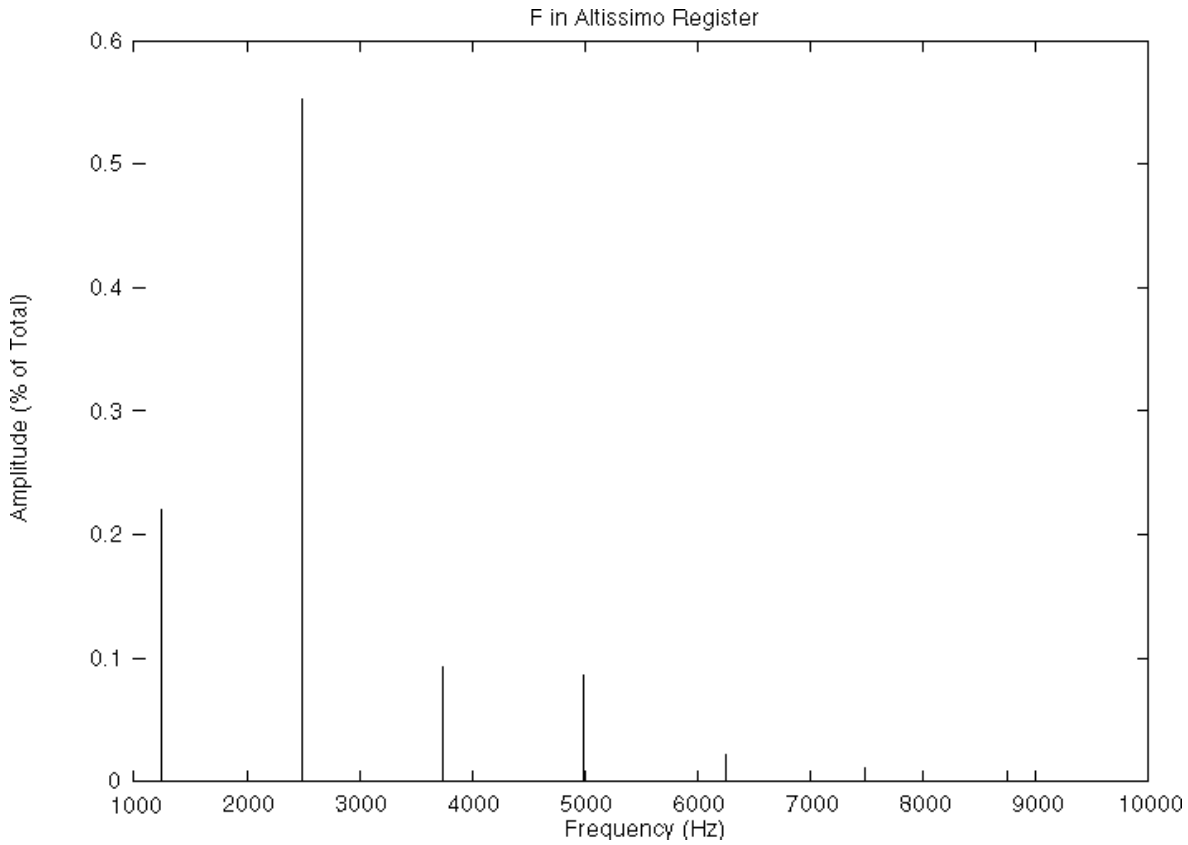

Figure 9. Frequency Spectrum of F in Altissimo Register. The number of harmonics has decreased significantly compared to Figure 8. This reduction in number of harmonics causes the instrument's tone to sound thinner than the upper register note. Interestingly, the first harmonic is the dominant frequency, which also contributes to the difference in tone color.

the entire compass of the instrument.

The clarinet, like many wind instruments, has several different registers. The lowest register, or chalemeau, contains the lowest D, or D_0 (147 Hz) through F_1 (349 Hz), 1.5 octaves above. The throat register, characterized by the keys near the top of the instrument, contains those notes between $F^{\#}_1$ (370 Hz), just above F_1, and A^{b}_2 (415 Hz). The upper register includes the notes between A_2 (440 Hz) and B^{b}_3 (932 Hz), and the third register, or the altissimo register, contains B_3 (988 Hz) through B^{b}_4 (1865 Hz). Figure 10 demonstrates that both the behavior of the fundamental and the first harmonic stay relatively constant within these partitions and change fairly dramatically at their

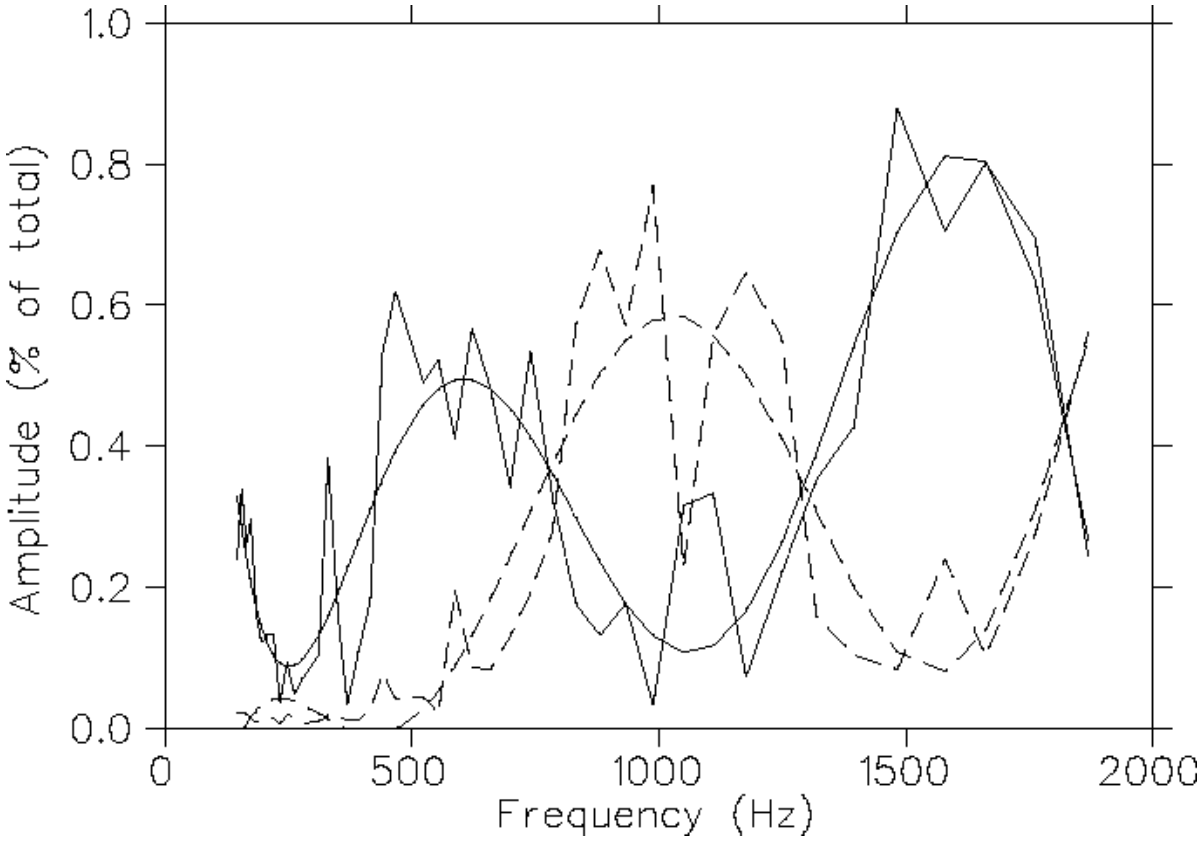

Figure 10. Fundamental and First Harmonic Experimental and Fitted Data. The data fits with existing register groupings on the instrument. The polynomial fit function works quite well for these harmonics. However, the higher harmonics (see Figure 11) do not fit nearly as well.

Frequency	p1	p2	p3	p4	p5	p6	p7
Fundamental	2.27700	-0.02367	9.08E-05	-1.53E-07	1.25E-10	-4.87E-14	7.21E-18
Fisrt Harmonic	-0.81462	0.01040	-4.56E-05	8.90E-08	-8.16E-11	3.48E-14	-5.56E-18
Second Harmonic	0.45671	-0.00429	2.09E-05	-4.28E-08	4.17E-11	-1.93E-14	3.39E-18
Third Harmonic	0.15866	-0.00226	1.22E-05	-2.64E-08	2.68E-11	-1.27E-14	2.28E-18

Table 2 Fit Parameters for fundamental, first, second, and third harmonics of the clarinet displayed in Figures 10 and 11.

boundaries. It is also notable that the fundamental and the first harmonic seem to have an inverse relationship.

Using *Physica*, a polynomial fit of the form was used to extrapolate a way to move between notes on each harmonic. The fit was fairly successful on the lowest harmonics. However, it fit less well as the harmonics got higher. The fits for each harmonic are displayed on the same graph as the experimental data for that harmonic for

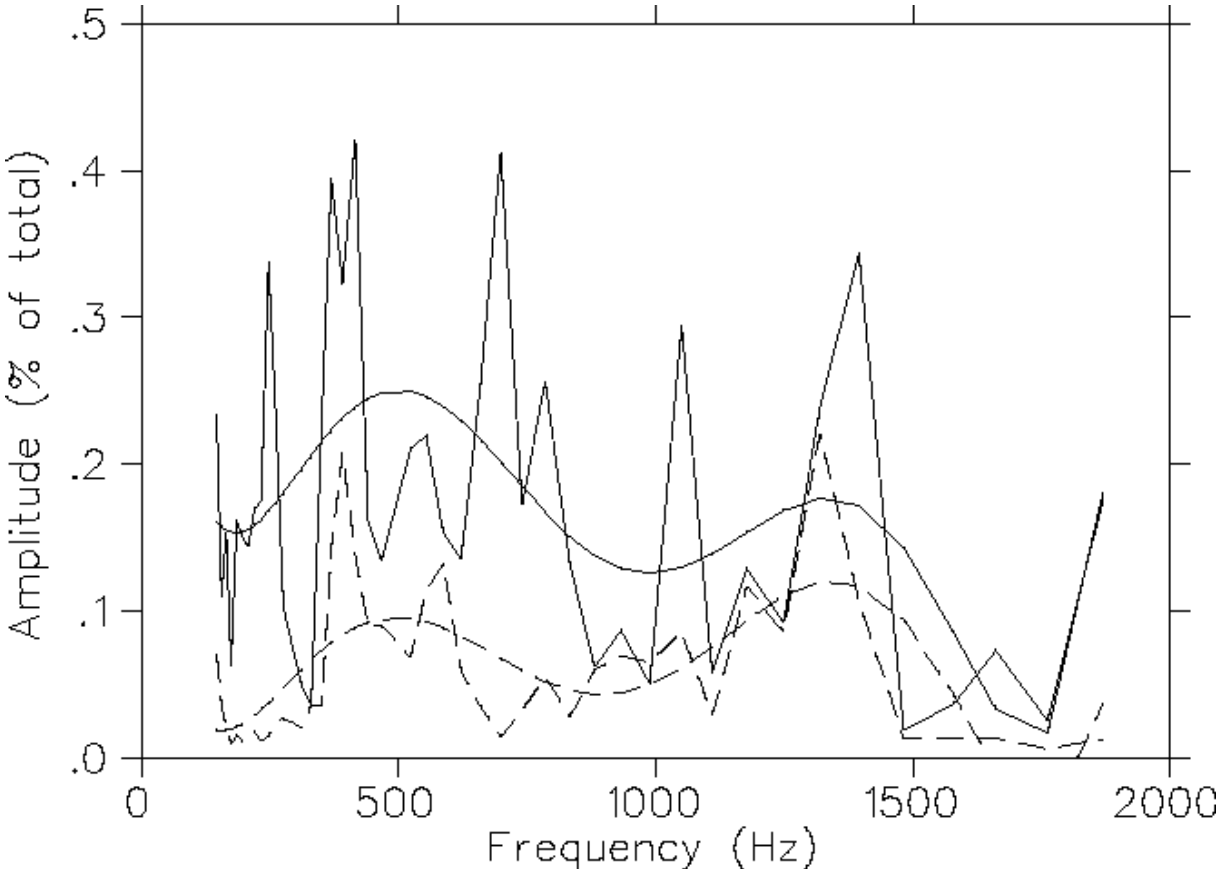

Figure 11. Second and Third Harmonics, Experimental and Fitted Data. The jaggedness of the data makes it very difficult to get a good fit with a polynomial function. It should be noted that the jumps are not as drastic as might appear at first glance, only around 20% of the total contribution. Other variables such as dynamic may contribute significantly to this discrepancy.

If dramatic changes are seen in the harmonic recipe at different volumes, it may help explain why the behavior of the higher order harmonics between notes is hard to predict. Using this new variable, it may be possible to generate a new set of samples which more accurately display the smooth change in harmonic recipe between adjacent notes, and the sharp changes between notes at cross registeral shifts. If fairly insignificant changes are observed for vastly different dynamics, then the poor fits noted above will be difficult to improve.

Conclusion

From the data gathered to this point, it is unclear exactly what would be the best way to represent the basic harmonic recipe for synthesis. If the data is tightened slightly by normalizing the samples' dynamics, it might be possible to produce much better algorithms spanning the compass of the instrument. However, if the data remains as widely varied as it is now, a general approximation may not be possible, and each individual harmonic recipe may need to be stored separately. The true test is to synthesize notes based on the curve approximations and decide whether the reproductions sound like the original.

In the case of many different wind instruments, it is likely that while the data will show general groupings by register as we have already seen, it may not be possible to generate an approximation that is good enough to fool the ear. The clarinet has notes which are considered *stuffy* and others that are considered more *free blowing*. Often these notes are right next to each other on the instruments compass. Acoustically, this translates into fairly different frequency spectra. This effect may necessitate storing the basic harmonic recipe for each note separately.

Regardless of which method is chosen to obtain the harmonic recipe for a particular note, it is now possible to realistically reproduce at least the steady state portion of each note of the clarinet. The method described in this study should be the same for any other instrument, making it a good starting point for synthesis. Once it is seen how some of the steady state effects such as dynamic changes change the recipe of the clarinet,

these effects can be included the synthetic reproduction of the clarinet sound. This has been done with the case of the tuning fork by synthesizing the sound decay. While the clarinet's higher harmonics will not decay away over time as was the case with the tuning fork, it should be possible to approximate their behavior and include these steady state effects in the synthetic reproduction.

Further Research

This study has dealt primarily with steady state reproduction of sounds. It had intended to progress beyond steady state and deal with attacks and releases as well. Time constraints did not allow this, but it should be noted that to achieve an accurate reproduction of an instrument, the attacks and releases must be studied in some detail. The attack is a vital part of the note because it is the first thing heard by the listener, and it helps him identify the instrument he is listening to. Each instrument has characteristic attacks just as it has a characteristic sustained sound.

In addition, the attack is very complicated because it contains many frequencies that would not be expected in a generic harmonic spectrum, and these frequencies change very quickly in time. Also, many different kinds of attacks are possible, and for the synthesis to be a useful tool, as many as possible of these attacks must be studied and synthesized. It may be useful to consider using wavelet analysis for this portion of the study. Since this project uses techniques independent of real-time synthesis, the longer calculation times may be acceptable if the synthesis is markedly improved from fast Fourier techniques.

Releases, while not quite as difficult to study, are also key. The note ending is the last sound the listener hears, and it is what he remembers. For that reason, performers can use many different releases for different purposes. Consequently, several different varieties of releases must be studied and synthesized.

After each note can be recreated with all of the desired inflections, it is optimal to build an interface that allows us to communicate the values for all the variables to the

computer. The original plan was to modify a clarinet by mounting sensors inside that connect to and communicate with a computer. The computer could then decode the information from the sensors, calculate values for the variables, and reproduce a pitch with the proper inflections of a synthesized, yet realistic-sounding, instrument. This would be an appropriate continuation of the project once a realistic model for one or more instruments has successfully been created.

Bibliography

Chuma, Joseph L. *Physica*. TRIUMF Software. V. 1.43, 1995.

Graps, Amara L. "An Introduction to Wavelets," *IEEE Computational Sciences and Engineering* 2(2):50-61, 1995.

Oates, Shaun. "Analytical Methods for Group Additive Synthesis," *Computer Music Journal* 21(2):21-39, 1997.

Hourdin, Christophe. "A Multidimensional Scaling Analysis of Musical Instruments' Time Varying Spectra," *Computer Music Journal* 21(2):40-55, 1997.

Hourdin, Christophe. "A Sound-Synthesis Technique Based on Multidimensional Scaling of Spectra," *Computer Music Journal* 21(2):56-68.

Press, William H., Saul A. Teukolsky, William T. Vettering, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* Second Ed. New York: Cambridge, 1992.

Fletcher, Neville H. *The Physics of Musical Instruments*. New York: Springer-Verlag, 1991.

Sundararajan, Duraisamy. "Fast Computation of the Discrete Fourier Transform of Real Data" *IEEE Transactions on Signal Processing*, 45(8):2010-2022, 1997.

Shlien, Seymour. "The Modulated Lapped Transform, its Time-Varying Forms, and its Applications to Audio Coding Standards," *IEEE Transactions on Speech and Audio Processing*, 5(4):359-366.

Appendix A: Fast Fourier Transform Package

```
/*************************************************************************
   fft.c  Version 0.05                                          03/28/98

  Fast Fourier Analysis program for acoustic samples.  Works currently with
  Microsoft wave file format (2's complement little endian).  Outputs data
  to work with gnuplot and to mimic data displayed in journals about
  acoustic synthesis.

*************************************************************************/

#include <math.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include "unitypes.h"
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <ctype.h>

/*************************************************************************
  global vars

*************************************************************************/
float *data,      /* The array used to take the transform of sampled data  */
  *product,       /* An array of half the size which ultimately contains
                  the result of the transform                            */
  **output,       /* An array of product variables */
  time_interval=1,/* The time interval over which to take the transform */
  time_offset=0;  /* The time offset from the beginning of the data in
                     in seconds, default is 0                            */
int sampling_rate,/* The sampling rate of the input sample (in Hz)         */
  bits,           /* The number of bits per sample (either 8 or 16)       */
  file,           /* The file variable for the input file                 */
  channels,       /* The number of channels specified in the input file
                  (either 1, mono or 2, stereo)                          */
  total,
  iter=1,         /* The number of iterations to take the transform       */
  slow=0;         /* Fast algorithm verses original algorithm             */
long datasize;    /* The number of samples in the input file              */
FILE *outfile,    /* The output file for the transform(s)                 */
  *outwave;       /* The output file for the parsed waveform              */

/*************************************************************************
  die

 Used to prematurely end the program in the case of an error.
 Writes a message to the screen to describe the error.
*************************************************************************/
void die(char *errstr) {
  printf("Error: %s\n", errstr);
  exit(1);
}
```

```
#define atof atof_fix  /* replaces atof which has a bug on our linux cluster */

/*****************************************************************************
  atof_fix

 Replacement for atof which has a bug on our linux cluster.
*****************************************************************************/
double atof_fix(char *str) {
  double d;

  if (sscanf(str,"%lf",&d) == 1) return d;
  else return 0;  /* an error occurred */
}

#define POW2(EXP) (1<<(EXP))  /* More efficient 2^n than pow */



/*****************************************************************************
  print_help_screen

 prints the usage information to the screen
*****************************************************************************/
void print_help_screen() {

  printf("\n\n");
  printf("usage: fft -i input [-h] [-o output] [-f outwave] [-t interval]\n");
  printf("       [-s offset] [-y iteration]\n\n");
  printf("  h : This screen\n\n");
  printf("  i input    : specify input file, no default, manditory\n");
  printf("  o output   : specify output file for the transform, default is");
  printf(" output.dat\n");
  printf("  f outwave  : specify output file for the waveform, default is");
  printf(" none\n");
  printf("  t interval : specify transform time interval, default is 1");
  printf(" second\n");
  printf("  s offset   : specify time offset from the beginning, default is");
  printf(" 0 seconds\n");
  printf("  y iteration: specify how many iterations to take the transform,");
  printf(" default is 1\n");
  printf("  m method   : specify between fft and original ft method,default ");
  printf("is fft.\n\n");

}
/*****************************************************************************
  parse_input

 Parses and records all of the values entered as flags on the commandline.
*****************************************************************************/
int parse_input(int argc, char **argv) {

  int curopt, ifile=0;
  char input_file[50],
    output_file[50]="output.dat",
    output_wave[50];

  opterr = 0;  /* disable error messages */

  while (1) {
    curopt = getopt(argc,argv,"i:o:t:y:s:f:mh");
```

```
    if (curopt < 0) break;

    switch (curopt) {
      case 'i':                          /* Flag for the name of the input    */
      strncpy(input_file,optarg,49);  /* file containing a sound sample    */
      input_file[49]='\0';
      ifile=1;
        break;
      case 'o':                          /* Flag for the name of the ouput    */
      strncpy(output_file,optarg,49); /* file to write the transform       */
      output_file[49]='\0';

        break;
      case 'f':                          /* Flag for the name of the ouput    */
      strncpy(output_wave,optarg,49); /* file to write the waveform        */
      output_wave[49]='\0';
      outwave = fopen(output_wave, "w");
      break;
      case 't':                          /* Flag for the time interval for    */
      time_interval = atof(optarg);   /* each computed transform           */
      break;
      case 's':                          /* Flag for the time offset from the */
      time_offset = atof(optarg);     /* beginning of the input file to    */
      break;                          /* to start reading data             */

      case 'y':                          /* Flag for the number of iterations */
      iter = atoi(optarg);            /* that the transform will occur     */
      break;
      case 'm':                          /* Flag which selects classic ft     */
      slow = 1;
      break;
      case 'h':                          /* Flag to request the flag listing  */
      print_help_screen();
      exit(0);
      case '?':                          /* Case an invalid flag was used     */
        printf("\nInvalid argument: %c\n",(char)optopt);
      print_help_screen();
      exit(1);
    }
  }
  if (ifile) {                            /* If all the input was specified   */
    file = open(input_file, O_RDONLY);   /* Opens the file for reading       */
    outfile = fopen(output_file, "w");   /* Opens the file for writing       */
    return 1;
  }
  else                                    /* halt on no input file            */
    die("You must specify an input file!\n");
  return 0;

}

/*****************************************************************************
  write_output

 Writes the output of the transform into the specified variable.
 Handles both two dimensional and three dimesional output.
*****************************************************************************/
int write_output(int length) {
  int i,j;

  for (i=0; i<length;i++) {
```

```
    for (j=0; j<iter; j++) {

    /* Prints the frequency of the corresponding data cell */
    fprintf(outfile, "%g\t", ((float)i*sampling_rate/(2*(length-1))));

    /* If there are multiple iterations of data prints out the beginning time
       for the current set of transform data */
    if (iter > 1) fprintf (outfile, "%f\t", j * (2.0*length/sampling_rate));

    /* Prints the amplitude for the current frequency */
    fprintf(outfile, "%d\n", (int)output[j][i]);
    }
    if (iter > 1) fprintf(outfile, "\n\n");
  }
  return 1;

}

/*****************************************************************************
  write_wave

 Writes out the specified file the waveform of the data which we are currently
 taking the transform of.
*****************************************************************************/
int write_wave(int length, int curiter) {
  int i,off;

  /* Calculates the time offset due to the current iteration of read data */
  off = (curiter)*length;
  for (i=0; i<length;i++) {

    /* Prints the time for the current amplitude of original wave data.  Uses
       the off variable calculated above and the orginal offset from the
       beginning of the file inputted from the command line */
    fprintf(outwave, "%f\t", (float)(off+i)/sampling_rate+time_offset);

    /* Prints the wave amplitude */
    fprintf(outwave, "%d\n", (int)data[i]);
    }
  return 1;

}

/*****************************************************************************
  adjust_phase

 Takes the output of the transform which returns real and imaginary data for
 each frequency and combines them into one value for each frequency.  It
 also puts the values in an array of half the original length since we are
 eliminating half the numbers through this process.
*****************************************************************************/
int adjust_phase(int total, int curiter) {
  int i;

  output[curiter][0] = abs(data[0]);
  output[curiter][total/2] = abs(data[1]);

  for (i=2; i < total; i+=2) {

    /* takes the square of the real and imaginary parts of the transform and
       combines them.  */
```

```
    output[curiter][i/2] = hypot(data[i], data[i+1]);
  }
  return total/2+1;
}

/*****************************************************************************
  next_pow2

 This function takes a number and finds the closest number (greater than the
 original) which is of the form 2^n
*****************************************************************************/
int next_pow2(int length) {
  int i=0;

  while (1) {
    if (POW2(i) >= (length-1)) break;
    i++;
  }
  return POW2(i);
}

/*****************************************************************************
  read_number

 Takes the input file and the length of the "word" in bytes and returns the
 value of the number.  This function expects to read a little endian number
 in 2's compliment form.
*****************************************************************************/
long read_number(int length) {
  int error, i;
  unsigned char uc;
  char c;
  long total=0;

  /* Reads byte by byte.  For numbers greater than 1 byte, the read numbers
     don't contain "sign" information, so we treat them as unsigned and read
     them from lowest order to highest order. */
  for (i=0; i < (length-1); i++) {
    error = read(file, &uc, 1);
    if (error <=0 ) die("Unexpected EOF or error!\n\n");
    total += uc * pow(16, 2*i);
  }

  /* The last byte we read (or only in the case of 1 byte numbers) contains
     "sign" information, so we treat it as signed and highest order. */
  error = read(file, &c, 1);
  if (error <=0) die("Unexpected EOF or error!\n\n");
  total += c * pow(16, 2*(length-1));

  return total;
}

/*****************************************************************************
  check_string

 Function takes the number of bytes to read out of the file, the string to
 compare it to, and whether it should be an exact match or if the input string
 should just be found somewhere in the specified block of the file.
*****************************************************************************/
int check_string(char *text, int num, int exact) {
```

```
  char temp[20];
  int error;

  error = read(file, &temp, num);
  temp[num] = 0;
  if (exact) {
    if (strcmp(temp, text) == 0) return 1;
    else return 0;
  }
  else {
    if (strstr(temp, text)) return 1;
    else return 0;
  }
}

/*****************************************************************************
  find_data_tag

 Finds the next occurance of "data" in the file and moves the pointer directly
 after it.
*****************************************************************************/
int find_data_tag() {
  int error=1;
  char c;

  while (error) {

    /* finds the next occurance of "d" in the file */
    while (error) {
      error = read (file, &c, 1);
      if (c == 'd') break;
    }
    /* after it has found a "d" make sure the next letter is an "a" otherwise
       start over */
    read (file, &c, 1);
    if (c != 'a') continue;
    /* after we have "da" make sure next is "t" */
    read (file, &c, 1);
    if (c != 't') continue;
    /* after we have "dat" make sure next is "a" */
    read (file, &c, 1);
    if (c != 'a') continue;
    /* if we got here we found "data" in the file, return true */
    else return 1;
  }
  return 0;  /* the file ran out before we found "data" return false */
}

/*****************************************************************************
  read_wav_header

 Opens and reads the header of the .wav file to get the sampling rate, number
 of bits, number of channels, and length of the file.  Also checks to make
 sure the header is valid to ensure it is a properly constructed wav file
*****************************************************************************/
int read_wav_header() {
  int file, fmtsize;

  if (!check_string("RIFF", 4, 1)) /* Check that the first 4 bytes     */
    die("Incorrect Wav File Header!\n"); /* contain "RIFF" (exact match)     */
```

```
  read_number(4);                  /* read the next four bytes which
                                   contain the length of data + fmt
                                   chunk, not currently used       */

  if (!check_string("WAVE", 4, 1)) /* Check that the next 4 bytes      */
    die("Incorrect wID!\n");             /* contain "WAVE" (exact match)     */

  if (!check_string("fmt", 4, 0))  /* Check that the next 4 bytes      */
    die("Incorrect fID!\n");             /* conatin "fmt" (inexact match)    */

  if ((fmtsize = read_number(4)) < 16) /* reads the size of the format */
    die("Format Chunk is Invalid!\n");   /* chunk (4 bytes), which has a min
                                   16 bytes of length              */

  read_number(2);                  /* not used, there is a discrepancy
                                   whether this stands for stereo/mono
                                   or wformat tag (ie PCM): 2 bytes */

  channels = read_number(2);       /* reads/checks the number of       */
  if (!(channels == 1) && !(channels == 2)) /* channels to be either 1,2   */
    die ("Wrong number of channels!\n");

  sampling_rate = read_number(4); /* reads the sampling rate          */

  read_number(6);                  /* not used: bytes per second (4
                                   bytes) bytes per sample (2 bytes)*/


  bits = read_number(2);           /* reads bits per sample, 2 bytes   */

  if (!find_data_tag(file))               /* search for the "data" tag        */
    die("Could not find data tag!\n");

  datasize = read_number(4);       /* reads the amount of amplitude data
                                   in the file (in bytes): 4 bytes  */

  return file;                            /* If we got this far, its a well
                                   formed wav header.  Returns the
                                   file handle for later reading    */
}

/*************************************************************************
  setup_data

 Takes the data from the wav header and command line, and sets up a float
 array of the proper length to be used to do the transform.  Also moves the
 pointer ahead in the file by the time offset specified on the command line
*************************************************************************/
int setup_data() {
  int temp, i;

  /* calculates the number of samples to ignore from the beginning */
  temp = ((int)(sampling_rate * time_offset)) * (bits/8) * channels;

  /* adjust datasize for the time offset from the beginning */
  datasize -= temp;

  /* discards the number of samples just calcuated */
  if (temp > 0) read_number(temp);

  /* calculates the number of samples to be used for each transform iteration*/
```

```
  temp = sampling_rate*time_interval;

  /* adjusts the number of samples to be of the form 2^n which is required for
     the transform to run */
  total = next_pow2(temp);

  /* adjusts the time interval to match the new number of samples */
  time_interval = (float)total/sampling_rate;

  /* finally setup the data array with the proper length */
  data = (float *)malloc(total * sizeof(float));

  /* allocates the proper length for the phase adjusted transform data in the
     product array */
  product = (float *)malloc((total/2+1) * sizeof(float));

  /* allocates the proper length for all of the product arrays to be used in
     the output file */

  output = (float **)malloc((iter+1) * sizeof(int));

  for (i=0; i<iter; i++)
    output[i] = (float *)malloc((total/2+1) * sizeof(float));

  return 1;
}

/*****************************************************************************
  read_input_wav

 Reads the appropriate number of samples from the wav file into the data
 array.
*****************************************************************************/
int read_input_wav() {
  int i=0, left, right;

  /* checks for stereo input, and combines the left and right channel in that
     case before inserting it into the data array.  Handles both 8 and 16 bit
     files */
  if (channels == 2)
    for (i=0; i < total; i++) {
      left = read_number(bits/8);
      right = read_number(bits/8);
      data[i] = (float)(left+right);
    }
  /* otherwise reads the samples directly into the array */
  else
    for (i=0; i < total; i++) {
      data[i] = read_number(bits/8);
    }
  return total;
}

/*****************************************************************************
  ispowerof2

 Checks to see if the input number is of the form 2^n
*****************************************************************************/
#define ispowerof2(X) !(X & (X-1))

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

```
  add_zeros

 Adds zero imaginary values to an array of real values.
*****************************************************************************/
void add_zeros(float data[], int N) {
  int i;

  for (i=N-1; i>=0; i-=2) {
    data[i-1] = data[i/2];
    data[i] = 0;
  }
}
/*****************************************************************************
  slow_fft

 Computes the fourier transform of an array of floats which is a discretely
 sampled waveform, using classical methods
*****************************************************************************/
void slow_fft(float data[], int N) {

  /* discrete fourier tranform of data, with N real and
     N imaginary elements, interleaved.  data[0]=real1,
     data[1]=imag1, data[2]=real2, etc. */


  void four1(float data[], unsigned long nn, int isign);
  double *trans;
  double tmpsin, tmpcos, x;
  int k, n, kk, nn;

  data = (float *)realloc(data,  2*N * sizeof(float));
  add_zeros(data, 2*N);
  for(n=0;n<2*N;n++)
    printf("%g\n", data[n]);
  printf("\n");
  trans = (double *)malloc(2*N*sizeof(double));
  for(n=0;n<N;n++)
    {
      nn = 2*n;
      trans[nn]=trans[nn+1]=0.0;
      for(k=0;k<N;k++)
      {
        kk = 2*k;
        x = 2 * M_PI * k * n / (double)N;
        tmpcos = cos(x);
        tmpsin = sin(x);
        trans[nn] += data[kk]*tmpcos;
        trans[nn] -= data[kk+1]*tmpsin;
        trans[nn+1] += data[kk]*tmpsin;
        trans[nn+1] += data[kk+1]*tmpcos;
      }
    }
  four1(data-1, N, 1);
  printf("\n");
  for(n=0;n<2*N;n++)
    printf("%g\n", trans[n]);
  printf("\n");
  for(n=0;n<2*N;n++)
    printf("%g\n", data[n]);

}
```

```
/*************************************************************************
  four1

 Computes the fast fourier transform of an array of floats which is a
 discretely sampled waveform.
*************************************************************************/
#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr

void four1(float data[], unsigned long nn, int isign)
{
      unsigned long n,mmax,m,j,istep,i;
      double wtemp,wr,wpr,wpi,wi,theta;
      float tempr,tempi;

      if (!ispowerof2(nn)) die("four1 received a data array without 2^n
elements.\n");
      if (nn == 0) die("four1 received an empty array.\n");

      n=nn << 1;
      j=1;
      for (i=1;i<n;i+=2) {
            if (j > i) {
                  SWAP(data[j],data[i]);
                  SWAP(data[j+1],data[i+1]);
            }
            m=n >> 1;
            while (m >= 2 && j > m) {
                  j -= m;
                  m >>= 1;
            }
            j += m;
      }
      mmax=2;
      while (n > mmax) {
            istep=mmax << 1;
            theta=isign*(6.28318530717959/mmax);
            wtemp=sin(0.5*theta);
            wpr = -2.0*wtemp*wtemp;
            wpi=sin(theta);
            wr=1.0;
            wi=0.0;
            for (m=1;m<mmax;m+=2) {
                  for (i=m;i<=n;i+=istep) {
                        j=i+mmax;
                        tempr=wr*data[j]-wi*data[j+1];
                        tempi=wr*data[j+1]+wi*data[j];
                        data[j]=data[i]-tempr;
                        data[j+1]=data[i+1]-tempi;
                        data[i] += tempr;
                        data[i+1] += tempi;
                  }
                  wr=(wtemp=wr)*wpr-wi*wpi+wr;
                  wi=wi*wpr+wtemp*wpi+wi;
            }
            mmax=istep;
      }
}
#undef SWAP

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

```
  realft

 Since the fast fourier transform function expects a real and imaginary part
 for each sample and all of our data is real, we can reassemble our data so
 the first half of the data is in all the of the "real" cells and the second
 half is in the "imaginary" cells, run the transform, and then decode the data
 back at the end.  This allows us to do twice the real data at once.
*****************************************************************************/
void realft(float data[], unsigned long n, int isign)
{
      void four1(float data[], unsigned long nn, int isign);
      unsigned long i,i1,i2,i3,i4,np3;
      float c1=0.5,c2,h1r,h1i,h2r,h2i;
      double wr,wi,wpr,wpi,wtemp,theta;

      theta=3.141592653589793/(double) (n>>1);
      if (isign == 1) {
            c2 = -0.5;
            four1(data,n>>1,1);
      } else {
            c2=0.5;
            theta = -theta;
      }
      wtemp=sin(0.5*theta);
      wpr = -2.0*wtemp*wtemp;
      wpi=sin(theta);
      wr=1.0+wpr;
      wi=wpi;
      np3=n+3;
      for (i=2;i<=(n>>2);i++) {
            i4=1+(i3=np3-(i2=1+(i1=i+i-1)));
            h1r=c1*(data[i1]+data[i3]);
            h1i=c1*(data[i2]-data[i4]);
            h2r = -c2*(data[i2]+data[i4]);
            h2i=c2*(data[i1]-data[i3]);
            data[i1]=h1r+wr*h2r-wi*h2i;
            data[i2]=h1i+wr*h2i+wi*h2r;
            data[i3]=h1r-wr*h2r+wi*h2i;
            data[i4] = -h1i+wr*h2i+wi*h2r;
            wr=(wtemp=wr)*wpr-wi*wpi+wr;
            wi=wi*wpr+wtemp*wpi+wi;
      }
      if (isign == 1) {
            data[1] = (h1r=data[1])+data[2];
            data[2] = h1r-data[2];
      } else {
            data[1]=c1*((h1r=data[1])+data[2]);
            data[2]=c1*(h1r-data[2]);
            four1(data,n>>1,-1);
      }
}

/*****************************************************************************
  initialize

 Gets the command line input, sets up the data array, reads the file header.
*****************************************************************************/
int initialize(int argc, char **argv) {

  printf("fft Version 0.05\n");
```

```
  parse_input(argc, argv);

  read_wav_header();

  setup_data();

  printf("Adjusted Time Interval: %f seconds\n", time_interval);

  return 1;

}

/*****************************************************************************
  calculate

 calculates the transforms for each iteration, and writes out the results as
 well as the original waveform.
*****************************************************************************/
int calculate() {
  int
    i=0,
    ntotal;

  while (i*total < datasize/(channels*bits/8)) {
    if (i >= iter) break;
    ntotal=read_input_wav();
    if (outwave != NULL)
      write_wave(ntotal, i);
    if (slow) slow_fft(data, ntotal);
    else
      realft(data-1,ntotal,1);
    /*    printf("\n");
    for (j=0; j<ntotal; j++)
      printf("%g\n", data[j]);
    printf("\n"); */
    ntotal=adjust_phase(ntotal, i++);
  }
  write_output(ntotal);
  return 1;

}
/*****************************************************************************
 main
*****************************************************************************/
int main (int argc, char **argv) {

  initialize(argc, argv);

  calculate();

  return 0;
}
```

Appendix B: Microsoft Wav File Format

Number Format

- All numbers are read little-endian. This means the highest order byte comes first.
- All values are 2's complement.

RIFF Chunk

RIFF Chunk bytes	12 bytes long
0-3	"RIFF"
4-7	length of entire file
8-11	"WAVE"

FMT Chunk

FMT Chunk bytes	24 Bytes Long (possibly longer)
0-3	"FMT "
4-7	length of the rest of the format chunk (16)
8-9	Amplitude Type, 1 for Pulse Code Modulation (???)
10-11	Number of Channels
12-15	Sample Rate (Hz)
16-19	Bytes per second
20-21	Bytes per sample: 1=8-bit mono, 2=8-bit stereo or 16-bit stereo, 4=16-bit stereo
22-23	Bits per sample

Data Chunk

DATA Chunk bytes	Variable Length
0-3	"data"
4-7	length of data chunk
8-end	amplitude data

Amplitude Data

- 8-bit mono: each byte is a sample
- 8-bit stereo: one byte is the left channel and the next is the right channel
- 16-bit mono: each sample is two bytes (little-endian, 2's complement)
- 16-bit stereo: each sample is four bytes, first two bytes are the left channel next two are the right channel (little endian, 2's complement)