University of Arkansas, Fayetteville

ScholarWorks@UARK

Computer Science and Computer Engineering Undergraduate Honors Theses

Computer Science and Computer Engineering

5-2008

Computer Generation and Processing of Music: Pitch Correction for the Human Voice

Jason Hardy
University of Arkansas, Fayetteville

Follow this and additional works at: https://scholarworks.uark.edu/csceuht

 Part of the Composition Commons, and the Graphics and Human Computer Interfaces Commons

Citation

Hardy, J. (2008). Computer Generation and Processing of Music: Pitch Correction for the Human Voice. *Computer Science and Computer Engineering Undergraduate Honors Theses* Retrieved from https://scholarworks.uark.edu/csceuht/14

This Thesis is brought to you for free and open access by the Computer Science and Computer Engineering at ScholarWorks@UARK. It has been accepted for inclusion in Computer Science and Computer Engineering Undergraduate Honors Theses by an authorized administrator of ScholarWorks@UARK. For more information, please contact scholar@uark.edu.

THE UNIVERSITY OF ARKANSAS: COLLEGE OF ENGINEERING, COMPUTER SCIENCE

Computer Generation and Processing of Music

Pitch Correction for the Human Voice

Jason Drew Hardy, Undergraduate CSCE
Dr. John Lusth, Advisor
04.29.2008

A thesis of research to develop and implement an algorithm to modify a digital waveform to a set of desired frequencies without changing the characteristics of the wave itself.

Introduction

Music is something everyone can enjoy. Shows like *American Idol* and *Dancing with the Stars* show us that most everyone, namely more than those who enjoy voting, does enjoy it. Not everyone can make music, and only a special few can make it well. The musically impaired have an ally, though, and it is technology. There are many synthetic ways one may become the dream superstar, especially if they are willing to pay enough. The majority of music generation, processing, and correction software is highly proprietary to their respective owners and carries a high price. Thus, it is difficult in an individual or academic setting to learn about these topics. It is for these reasons the research described below has partnered with the *SongLib* project to bring a suite of musical processing software to the open source community. Specifically, it aims to develop and implement an algorithm to manipulate the pitch of a recorded human voice to some desired value.

Background

Software production is a traditionally a business. A vendor will develop, test, copyright, market, and distribute some software and sell licenses to use it. After the sale, they will issue support and updates provided the user continues to pay for their services. This is great in most settings and provides consumers with a competitive market for software, but it does pose several challenges in special cases. High costs, low support-levels, and limited detailed knowledge about the software can be a problem for some users but especially for those in an individual or academic environment where funds are limited or intricate understanding is needed to aid in the teaching process.

The above challenges are magnified when it comes to music processing software. The number of programs which have pitch correction abilities is limited and the underlying

algorithms are not publicly documented as they are proprietary to their respective owners. *Apple* provides a post-recording suite for $499 and *Antares* provides a real-time suite for $399, both of which are priced for individual license and home use. Studios, which offer more advanced capabilities, can reach into the tens-of-thousands of dollars. Few can afford to pay hundreds of dollars for software, even if it is discounted for academic use, or thousands to have a personal studio. Perhaps more of a problem is the fact that the underlying algorithms and processes are documented only to the extent of the program's regular use. They may be glazed over at high level, but they are not detailed enough for a user to gain an intricate understanding. This makes teaching and learning past the point of regular use difficult. Developing an open-source solution for music processing with pitch correction would solve, or at least ease, most of these problems.

Open-source is a movement to *open* the *source* of software to a community of developers and users by making the source code of a project freely available and modifiable. It applies directly to the problems mentioned previously by lowering costs, increasing support channels, and providing all related files. It lowers the cost of software because most open-source licenses *require* the program be made publically available at no charge. While many proprietary programs make use of open-source projects, they must follow the exact terms of the attached license which usually means giving due credit and opening the part of the software that makes use of the aforementioned project. Support channels are opened because the project is available to a seemingly infinite number of independent developers of differing proficiencies who can contribute to the project. This requires a strict paper trail for how the software changes over time but can lead to a better, more functional program over all. Finally and most importantly, open-source licenses *require* the source code and documentation be made readily available any time it is used. This is of the most benefit to the academic world because it allows both professors and

students to gain an intricate understanding of the software. Unfortunately though, music processing software is essentially untouched in the open-source community [1].

No open-source music processing projects exist, or if they do exist they are small and underdeveloped. Furthermore, few algorithms for pitch correction have been publically published. One algorithm, called PSOLA, has been documented to some extent, but it is complex and implementations are hard to find. **P**itch-**S**ynchronous **O**ver**l**ap and **A**dd operates on the transitions between sounds by breaking the signal into windows and modifying the frequency by adding or removing cycles appropriately. To recreate the original spectrum with the new pitch, the resulting windows are joined together and overlapped [2]. Open implementations of PSOLA are even more rare, or non-existent, and the algorithm itself is sparsely documented. The research described below is a simplification of the PSOLA technique.

Approach

All source code is written for *Java* version 1.6.0 and makes use of several of the ***J****ava* ***D****evelopment* ***K****it* (JDK) libraries, and the ***R****eadily* ***R****eadable* ***A****udio* (RRA) file format is used because it stores amplitude values as a set of easily accessible sample points. The first of many simplifications is the program will know prior to processing: known frequencies, desired frequencies, and durations. These values are input as an annotation to the audio stream and are assumed to be of *Double* data type, but no checks are made as to their appropriateness or correctness. It is up to the user to provide accurate values. These three values allow the user to specify any number of tonal modifications throughout the wave spectrum. Also, the program is provided the name of the RRA file to be modified. It is instantiated at runtime as an RRA object which provides the necessary accessor and mutator methods.

The program runs through a series of computations to generate several values required before processing can begin. The number that guides the correction process is called the *ResampleFactor* and is calculated by dividing the desired frequency by the known frequency. Second, *SamplesPerCycle* is readily computed because the original frequency is provided to the program. It is calculated by dividing the sample rate, provided by the RRA object, by the known frequency at that window. Finally, to aid in later processing *SamplesNeeded* and *CyclesNeeded* are computed by multiplying the total samples in this window by the *ResampleFactor* and dividing *SamplesNeeded* by *SamplesPerCycle*, respectively. Both values are considered *Integers*, taken as the ceiling of the noted computations, and represent the total number of samples or cycles after pitch modification. All are stored in parallel arrays with each index representing a different window to allow for easy access in the future. Here, another simplification comes into play: only full cycle values are allowed at several points in processing (noted later). The program is still in the *setup* phase up until this point.

The *processing* phase begins by comparing each index in the *ResampleFactor* array and executing the appropriate method. A factor of greater than one signifies the desired frequency is greater than the known frequency, meaning the tone is flat, and the *CorrectFlat* method is called. Otherwise, the factor is less than one, i.e. *DesiredFrequency* < *KnownFrequency* and thus the tone is sharp, and the *CorrectSharp* method is called.

A flat pitch is denoted by the desired frequency being greater than the known frequency and is corrected by adding more fluctuations to the spectrum in the given window. This will be accomplished by copying cycles within the window and re-sampling down to the original number of samples. A temporary array of size *SamplesNeeded*, which is greater than the original window size, is created to hold the new data and is initially filled with the first few cycles of the

original window. This is done because a "warm-up" period exists before a given tone can be reached during which large fluctuations and sharp transitions can occur. Fortunately, a few cycles does not add nor take away from the overall sound so they are left alone. For the same reasons, the last few cycles as the sound transitions into a new pitch are also left alone. The number of cycles to skip both at the beginning and at the end of the window is configurable, and the amount of data left to fill is consequently (*CyclesNeeded – (2 * CyclesToSkip))*. Both *CyclesNeeded* and *SamplesNeeded* represent the amount of data required to achieve the desired frequency after the resulting window is re-sampled. Cycles are added to the spectrum in the following loop:

```
count = (CyclesToSkip * SamplesPerCycle);
placeHolder = count;
for i = 0 to SamplesPerCycle
{
        temp[count] = rra.getSample(placeHolder);
        temp[count + SamplesPerCycle] = rra.getSample(placeHolder);
        count++;
        placeHolder++;
}
count = count + SamplesPerCycle;
```

In this way, a given cycle is duplicated in the indices immediately following it. The entire loop runs until *SamplesNeeded* is fulfilled with a separate loop to take care of the odd cycle, if necessary. This produces a spectrum ready for re-sampling to produce a stream with the desired frequency inside the original window.

Re-sampling a modified flat pitch requires the removal of samples until the original window's characteristics are met, a sort of compression. Think of taking a window and squeezing it to fit into a smaller one, ensuring the tone will sound in the same amount of time as and match up with the original samples. The *Resample* method is provided with the data array,

the required size of the array, the amount of samples to skip, and the index corresponding appropriate place in the *CyclesPerSample* array. In this case, the size of the data is greater than the desired size so samples must be removed with the number of samples to remove being simply *data.length – DesiredSize*. A temporary array is created to hold the modified data and another array holds the sample indices to remove. The samples to remove are generated via a random number generator, seeded with time, and adjusted to yield a random sample in a random cycle within the bounds of the skipped cycles. The algorithm is as follows:

```
for i = 0 to NumberToRemove
{
        do
        {
                cycle = Math.round(rand.nextDouble() * (data.length / SamplesPerCycle))
                cycle = cycle + CyclesToSkip;
                sample = Math.round(rand.nextDouble() * SamplesPerCycle));
        } while (((cycle * SamplesPerCycle) + sample) > (data.length – SamplesToSkip));
        samplesToRemove.add((cycle * SamplesPerCycle) + sample);
}
```

The loop results in a sample index to remove and runs until enough are generated to produce an array of the same length as the original data. Once the indices are generated, a simple loop runs which actually removes the respective samples. Finally, the corrected array is passed to the W*riteCorrectedRRA* method which replaces the RRA object's sample points with the corrected ones and calls the W*riteRRA* method to actually write out the new RRA file. The new file will have "_corr" appended to the end of the name before the file extension, if there is one. (Figure 1 illustrates the *CorrectFlat* and *Resample* process.)

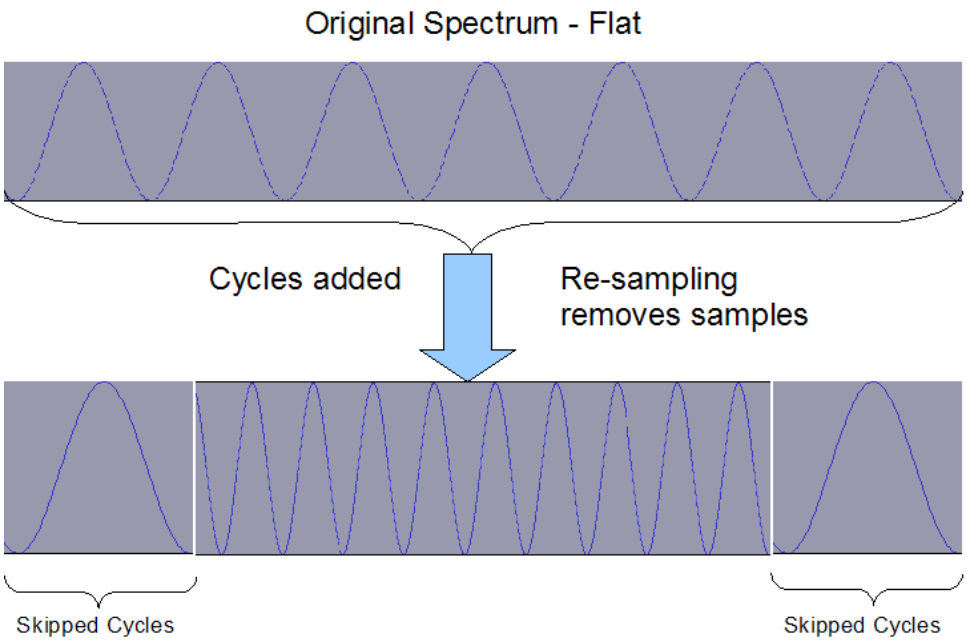

Figure 1 - *CorrectFlat* and *Resample*

A sharp pitch is denoted by the desired frequency being less than the known frequency and is corrected by removing cycles from the given window. Consequently, cycles must later be copied back in to maintain the original characteristics of the wave. A temporary array of size *SamplesNeeded*, which is less than the original window size, is created to hold the modified data and is initially populated, for the same reasons as before, with the first and last few cycles from the original spectrum at the beginning and end of the array. The middle portion of the array is filled with just enough cycles from the original stream resulting in an array ready to for re-sampling to produce a stream with the desired frequency inside the original window.

Re-sampling a modified sharp pitch requires the addition of samples until the original window's characteristics are met. Think of stretching a wave until the desired frequency is reached. As before, the *Resample* method is provided with the data array, the required size of the array, the amount of samples to skip, and the index corresponding appropriate place in the *CyclesPerSample* array. In this case, the desired size of the data is less than the desired size so

samples will need to be added with the number of samples needed being simply *DesiredSize – data.length*. A temporary array is created to hold the modified data and another array holds the indices at which to add a sample. The indices at which to add a sample are generated via a random number generator, seeded with time, and adjusted to yield a random sample in a random cycle within the bounds of the skipped cycles. The algorithm is the same as for generating indices for a flat tone. The simplest method of adding samples is to copy it, insert it at the following index, and shift the remaining indices up by one. This will result in a stair step pattern in the waveform and interrupts the continuous nature of the wave. A better way is to use some interpolation function to smooth the addition of samples. The method used in this algorithm is:

```
for i = 0 to NumberToAdd
{
        PrevValue = data.getSample(IndicesToAdd[i] – 1);
        temp.add(IndicesToAdd[i], (data.getSample(IndicesToAdd[i]) + PrevValue) / 2);
}
```

The *add* method automatically increments every following index by one to allow for the inserted sample. This loops until *temp.length* equals *DesiredSize* at which point the processing is finished. Finally, the corrected array is passed to the *WriteCorrectedRRA* method which replaces the RRA object's sample points with the corrected ones and calls the W*riteRRA* method to actually write out the new RRA file. The new file will have "_corr" appended to the end of the name before the file extension, if there is one. (Figure 2 illustrates the *CorrectSharp* and *Resample* process.)

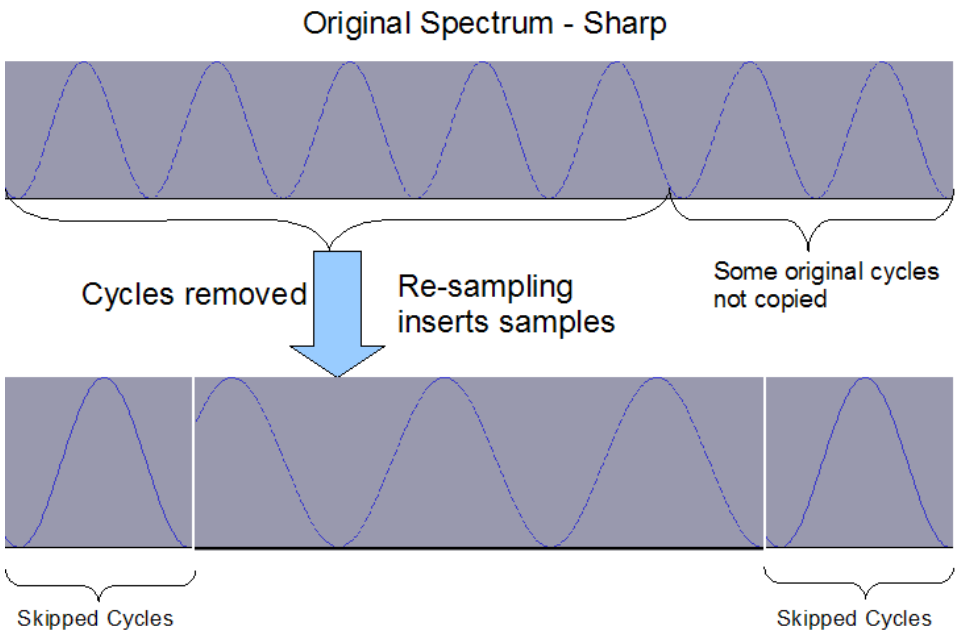

Figure 2 - *CorrectSharp* and *Resample*

Conclusion

The algorithm was only executed on perfect sinusoidal waves generated using *Audacity* for the sake of simplicity. For the algorithm to run successfully on more complex waves, like the human voice, it would need several improvements which will be discussed in the *Future Work* section. Four different cases were tested: 440 to 445 Hz, 440 to 435 Hz, 392 to 397 Hz, and 392 to 387 Hz each with duration of five seconds. The resulting RRA file was put through a frequency analyzer program which computes the frequency using the Zero Crossing Rate (ZCR) method. ZCR computes the frequency by examining the number of times the waveform changes from positive to negative, or vice versa, and thereby computes the number of cycles in the stream. This method works well for perfect waves, but does not for complex ones because the behavior may be erratic near the zero-point causing the frequency to be overestimated [3]. Table 1 shows the results of the trial runs.

Table 1: Test Run Results

Known Frequency (Hz)	Desired Frequency (Hz)	Time Length (s)	Resulting Frequency (Hz)	Error (%)
440	445	5	443.401	.359
440	435	5	435.199	.046
392	397	5	396.501	.126
392	387	5	387.298	.077

The above numbers may look exciting, but they are deceitful. The algorithm produces a tone close to the desired frequency, but the resulting sound is full of static. This is most likely due to the simplifications and assumptions mentioned throughout the paper.

One simplification causing a major concern is how the skipped cycles are added to the modified data stream. Currently, the skipped cycles are copied to the front and end of the temporary array before the modified data is populated, but there is no check to see if the modified data matches up with the skipped cycles. Both Figure 3 and Figure 4 show the consequences of this simplification.

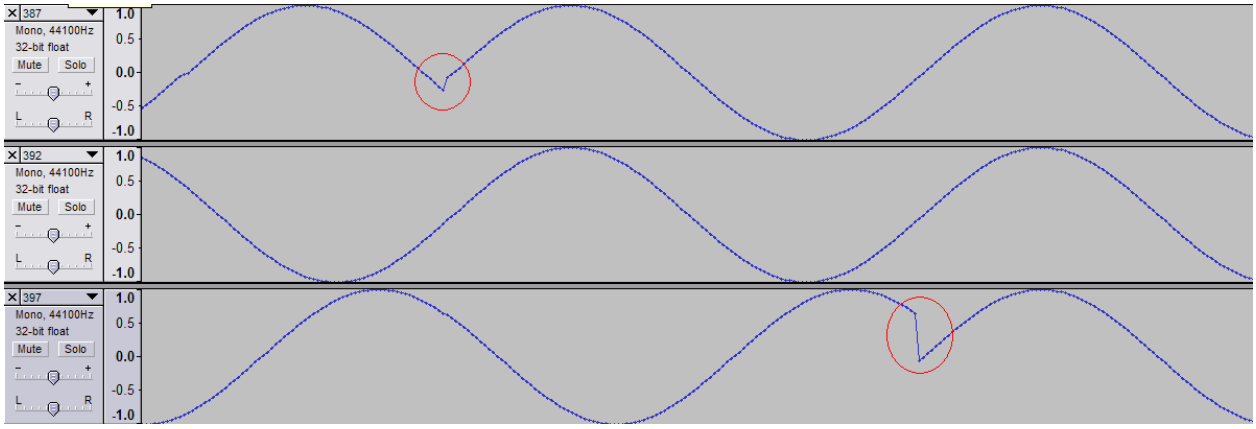

Figure 3 - 392 Hz Test Runs

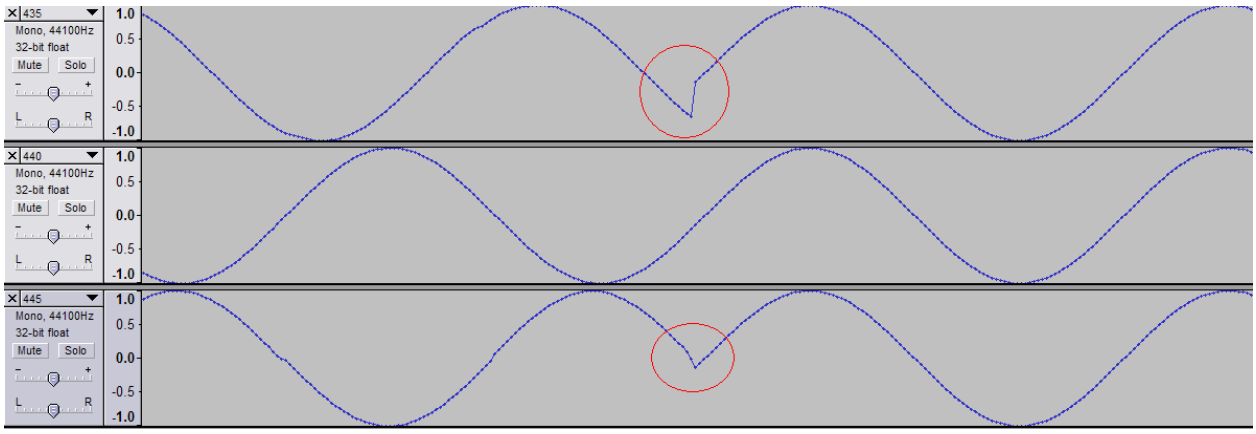

Figure 4 - 440 Hz Test Runs

The breaks could be avoided by adding a post-processing check to ensure the added data matches the existing skipped cycles. Also, the algorithm should only manipulate full cycles to avoid fractional breaks and discontinuities.

Re-sampling is causing the majority of the static, though. Although it is highly subtle, discontinuous breaks in the waveform cause a short, quiet section of white noise to be heard in the background of the tone. Figures 5 and 6 depict the minuscule breaks which occur just frequently enough to cause the sound to distort. In more extreme runs, like correcting the frequency from 440 to 400 Hz, the static is much more prevalent. This makes sense because a greater change in frequency leads to more manipulated samples which results in more blips.

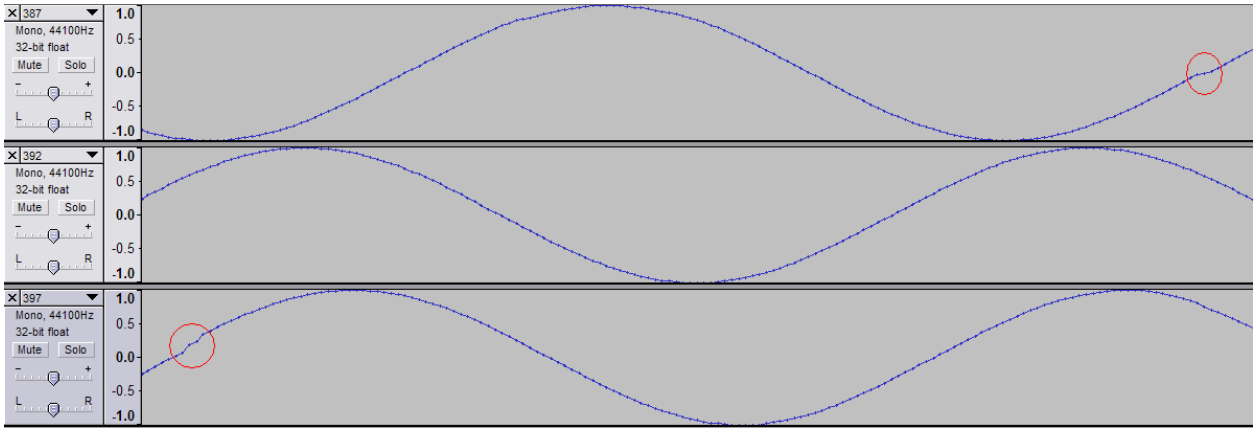

Figure 5 - 392 Hz Test Runs

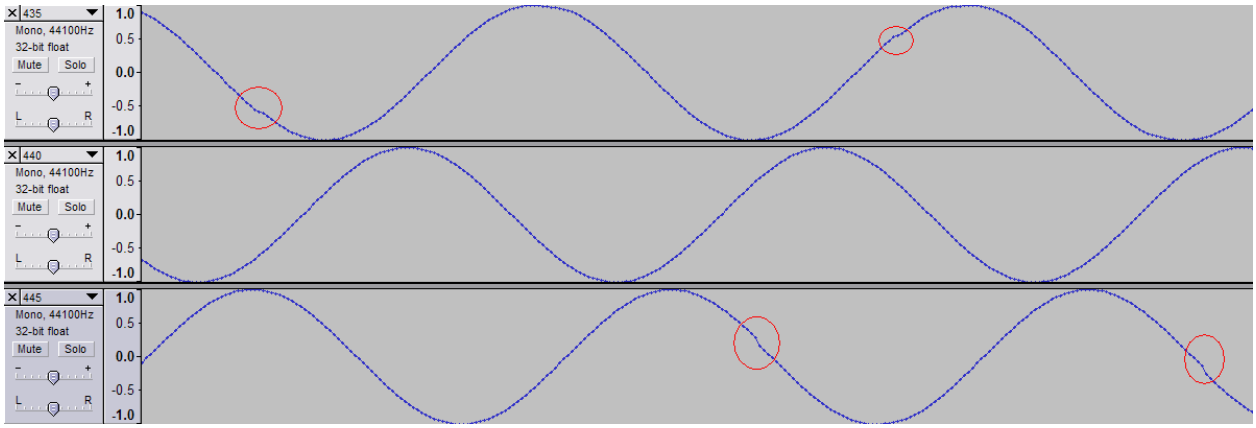

Figure 6 - 440 Hz Test Runs

The static could be greatly reduced by enhancing the interpolation function. Currently, samples are removed by simple deletion and samples are added by taking the average of the two surrounding data points. A more thorough re-sampling function would allow for smoother additions and deletions and thus lead to a more continuous wave.

The algorithm succeeds in showing that a simplified PSOLA technique does work to both increase and decrease the frequency of a given waveform to some desired value. Theoretically, the move from perfect wave to complex wave would only require minor additions and adjustments.

Future Work

The most prudent addition to this algorithm would be to add the necessary processes to allow for the manipulation of the human voice. As it stands, the program would not perform well on samples from a voice because of all the simplifications and assumptions made throughout. The idea is the same because all sound is fundamentally a wave, but most are much more complex than a generated tone.

Second would be to take steps to solve the problems outlined in the *Conclusion* section. Each modification that allows for smoother additions and deletions to the stream would make for

a better sounding output. Furthermore, the addition of a smoothing function to run post-processing would be highly beneficial because it could be called any number of times over any given window.

Lastly, the program itself should be more modular and configurable. The user should be able to plug in their own addition, deletion, re-sampling, interpolation, and smoothing functions to customize it to fit their needs. The number of cycles to skip and error threshold should also be configurable via command line options or an outside configuration file to allow for more flexibility. Along these lines, the algorithm should be enhanced to allow the original frequency to be calculated at runtime rather than hard-coded.

References

[1] J. Dionisio, C. Dickson, S. August, P. Dorin, R. Toal, “An Open Source Software Culture in the Undergraduate Computer Science Curriculum," inroads, vol. 39, no. 2, pp. 70-74, 2007.

[2] H. Valbret, E. Moulines, J. Tubach, “Voice Transformation Using PSOLA Technique,” Acoustics, Speech, and Signal Processing, vol. 1, no. 23, pp. 145-148, 1992.

[3] D. Gerhard, “Pitch Extraction and Fundamental Frequency: History and Current Techniques,” Technical Report, vol. 2003, no. 6, pp. 1-22, 2003.