Signal Harmonization by Resampling to Design a Guitar Effect Pedal

Cosmo Alto
Professor Erik Cheever
May 6th, 2016

Abstract

A melodic audio signal harmonization algorithm was designed in MATLAB, implemented in C code, then run on a C2000 Piccolo F28069 Control Card and Experimenter's Kit from Texas Instruments. Using four output buffers, the algorithm generates a higher-octave harmony signal for an electric instrument signal. This is done by resampling, a process by which the signal, originally sampled at 44.1 kHz, is resampled at 22.05 kHz to produce output buffers containing every other data point of the input, causing a periodic signal to oscillate twice as quickly as the original signal. The raw guitar pickup output was amplified with an amplifier with a gain of 10 so that it could be read into the device. Crossfading between two output channels was implemented to eliminate an audible clicking artifacts that were a direct result of discontinuities present in the output signal. The output signal was sent directly from the device to a guitar amplifier to be played.

Introduction

The goal of this project was to create a signal processing unit that synthesizes a higher-octave harmony signal in real time from a live input signal that is intended to be generated by an electric guitar or other electric instruments. All of the processing of the final module is done by an algorithm written in C and implemented with a digital signal processing (DSP) chip, the TI C2000 Piccolo F28069 Control Card and Experimenter's Kit. The harmonization algorithm itself was designed majoritally using MATLAB.

To synthesize a harmony signal in real time, I designed and implemented a time-domain-based resampling harmonization algorithm on the TMS320C2000 Piccolo with the assistance of Professor Erik Cheever. The algorithm synthesizes the harmony signal directly from the input data on a per-data-point basis. Ideally, the output of the chip would be combined with the input signal so that the resulting signal sounds as though two instruments are being played simultaneously. This step was never fully realized. This project did not require a large budget because it is implemented using a TI C2000 series Delfino, which is available for free in Swarthmore College's engineering laboratory.

In this proposal, the motivation for building a harmonizing guitar pedal is explained and the prerequisite musical theory and background are also explained. Most importantly, the resampling algorithm and digital signal processing techniques used in the project are explained as well. The major problems that arose in designing this guitar pedal are also explained, along with the solutions implemented to fix those problems.

The required components as well as a description of how these components were used to produce the end result is included in this report.

Theory: Music

Harmony is defined generally as the combination of multiple pitches to form chords, and is a fundamental element of music. The nature of a harmony is defined by the relative frequencies of each of the pitches present in a given chord. Harmony *intervals* determine the nature of the harmony and are defined by the proportion of the frequency of one pitch to the frequency of others. The simplest frequency proportions are most pleasing to the human ear, and are thus the most common harmony intervals found in music. As some examples, an octave harmony is defined as a 2:1 ratio of the frequency of one pitch to another, a perfect fifth is a 3:2 frequency ratio and a fourth is a 4:3 ratio (1). These frequency ratios form the basis of our approach to the problem of melodic signal harmonization.

It can be difficult for a musician to play a harmony line with a melody on a single instrument, either because playing the harmony parts requires greater mechanical skill of the musician, the desired harmony pitch lies outside of the tonal range of the instrument, or it is possible that the instrument can only produce a single pitch at a time. For these reasons, it can be desirable to use a digital device to aid a musician to generate harmony lines to complement a melody. This was the major motivation for this project.

The harmonization unit designed in this project generates a higher-octave harmony signal to accompany a melodic input signal produced by an electric

instrument, namely an electric guitar. Octaves were chosen as the harmony interval because the frequency proportion of a higher-octave to its bass note is the aforementioned straightforward 2 to 1 ratio. The simplicity of this ratio means the task at hand becomes that of doubling the frequency (or halving the wavelength) of a given input signal. Similarly, to generate a lower-octave harmony, the task is to double the wavelength of the input signal. Although there are several possible approaches to signal harmonization, I chose to use a technique called *resampling* due to its computational simplicity. In later sections I will present the steps that were taken to design the resampling algorithm to produce a higher-octave harmony output signal of high sound quality for a given melodic input. I will also present how this algorithm was implemented in real-time using a digital signal processing (DSP) chip, the TI C2000 Piccolo F28069.

Theory: Resampling Algorithm

The resampling algorithm begins by sampling an input signal at a fixed sampling rate. The most common sampling rate for audio applications is 44.1 kHz because at this rate there is absolutely no worry of possible aliasing present the resultant sampled signal. We know this because the Nyquist frequency for a sampling rate of 44.1 kHz is 22.05 kHz and the human auditory spectrum is known to exist in a range from roughly 20 Hz to 20 kHz. After resampling, the Nyquist frequency becomes 11.025 kHz, which still ensures that there will be no aliasing issues as long as the input signal comes from a guitar, whose frequency range exists in a range from roughly 30 Hz to 2 kHz (2).

The theory behind a resampling harmonization algorithm is best explained through an example. Consider an array of audio data of length n data points that was sampled initially at a rate of 44.1 kHz, called an *analysis frame*. Resampling the analysis frame at a frequency that is one half of the original sampling frequency results in an output frame of length $n/2$ data points. In general, to harmonize with a periodic input signal, the ratio of the resampling rate to the original sampling rate yields both the ratio of the output frame length to the input frame length as well as the ratio of the output signal frequency to the input signal frequency. Thus, to generate an octave signal of double the frequency of the input, the resampling rate is simply half of the original sampling rate. This is true as long as the resampled data is played back at the original sampling frequency.

Although it was not implemented in the final device, lower-octave harmonies were also considered in the design of this harmonization algorithm. To generate lower-octave harmonies, each data point in the analysis frame has an identical copy of itself inserted directly next to it, doubling the number of data points. The result is an output frame of length $2n$ whose frequency components have been halved. The signal must then be cropped to a length of n to be combined with the other higher-frequency signals to be output together.

A major problem that arises when using the resampling method to generate a higher-octave harmony is that the resampled signal, while it has been successfully pitch-shifted, is of an insufficient length to match the original input signal. There are

two potential solutions to consider when facing the problem of lost signal information due to resampling.

A simple solution to this problem would be to periodically extend each individual output frame in order to fill in the missing data. By creating a second copy of the output buffer immediately after the first and thus periodically extending the output frame to produce an output frame of twice its duration, we can compensate for the missing data. The problem with periodically extending our output is that by doing so the algorithm adapts to changes in the pitch of the input half as quickly as it otherwise would, since the output contains the same frequency information for twice as long as is achieveable through other methods.

The second solution to be considered to compensate for this loss of data is to continually run two or more of the resampling algorithms simultaneously to produce twice as much output data. The doubled algorithm would be implemented such that the generated outputs are 180 degrees out of phase. For example, if the time it takes to fill an analysis frame is 10 ms, then the higher-octave harmony signal is 5 ms in length and 5 ms of data are lost to resampling. By having a 5 ms delay between two sampling and harmonization algorithms, that 5 ms of lost information from one half of the algorithm is compensated for by the other half. The output frames can then be connected beginning-to-end such that the output signal is continuous.

A complication facing this method is the fact if the beginning and end points of the analysis frames aren't equal in magnitude, discontinuities between the end of the first harmony clip and the beginning of the second harmony clip are introduced. This

phenomenon occurs when the analysis frame length n is not an integer multiple of the wavelength of the signal and thus the beginning and end points of the analysis and output frames are nonzero. These types of discontinuities have audible effects that negatively impact the quality of sound of the output.

In order to avoid this problem in implementing the harmonization algorithm, there are again two possible solutions. A phase vocoder can be used to ensure that the analysis frame length n is constantly being updated to be an integer multiple of the period of the signal such that the first and last data points sampled are very near zero, minimizing the discontinuity as the frame shifts. Implementing this solution involves a feedback control loop that uses measurements of the fundamental frequency of the input signal to update the analysis frame such that the phase difference is minimized and there is no discontinuity or resulting click (3). Due to the computational complexity of this solution, it is desirable to seek an alternative method.

A different, less computationally intensive method to remove the audible clicks from the output signal is by crossfading between two output channels. Crossfading requires having two output signals, the volume of which is oscillating between 0 and 1 such that one signal is at max volume when the other is minimized (4). The output signals generated by resampling contain clicking from discontinuities that occur at a fixed rate, determined by the analysis frame length n. Thus, in order to implement a crossfade, the resampling algorithm described above must be scaled up once more by a factor of two. This is easily done in code by replicating the resampling process again, thus creating 4 outputs.

The first step of crossfading is to phase-shift the two output channels such that the clicks of the respective channels are 180 out of phase with one another, thereby doubling the frequency of the click if both channels are played back together. The next step is to modulate the amplitude of each output frame by a triangular distribution, such that the amplitude of the signal at the onset of the click is 0 and 1 in the center of the output frame (4). The final output channel is generated by combining these two channels. Theoretically, the clicks are inaudible after this step is complete because the output is smoothly transitioning between the purely periodic sections of the two output channels. While crossfading would significantly improve the quality of the output by removing the audible clicks, it is still not ideal because the crossfade modulation would be slightly audible.

In summary, by resampling an input signal into four separate output channels we can create two continuous streams of harmonized output data which can then be crossfaded to eliminate clicking artifacts that result from resampling. The result is a higher-octave replica of the input signal whose signal resolution is half of the original input, since half of the data is lost during the resampling process. Amplitude modulation due to crossfading would be audible, however it is still a great improvement over audible discontinuities. This is the process that I strove to replicate both in the MATLAB and C implementations of this algorithm.

Algorithm Development

The majority of this project was spent theorizing and developing the algorithm that would produce octave-harmony outputs for a given input signal. The

harmonization algorithm was designed in three distinct phases to implement each of the three processes described previously in the theory section. In this section I describe how each phase of the algorithm was designed and implemented using MATLAB. This design phase was meant as proofs of concepts for the eventual real-time harmonization algorithm, but these versions do not correspond directly with the final C implementation, since the MATLAB algorithms processed static datasets.

Phase 1: Resampling

The first iteration of the algorithm implemented resampling and signal doubling to generate a higher-octave harmony and a lower-octave harmony, respectively. First, a section of data is clipped from the original audio file to act as the analysis frame. Using a for-loop, every other data point from the analysis frame of length n is taken and stored into an output array of length $n/2$. When played back at the sampling frequency, this signal is an octave in frequency higher than the original signal. To generate a lower-octave signal, this algorithm uses a second for-loop to create two copies of each data point in the analysis frame and store them sequentially in the output frame, which has resultant length $2n$. The result, when played back at the original sampling rate, is an audio signal that is an octave lower than the original signal in frequency. In order to play all three of these signals back simultaneously, they have to be cropped to the same length in samples and thus are cropped to length of the higher octave output, length $n/2$. The MATLAB code to implement this algorithm is appended under Version 1.

To ensure that the algorithm was performing the frequency-shifting operations as desired, fourier spectra of each of the respective signals were taken and plotted for

comparison. If the algorithm was working as desired, the higher-octave signal would have doubled frequency components compared to the original and the lower-octave signal would have halved frequency components compared to the original signal. The frequency spectra are shown in Figure 1 below for comparison:

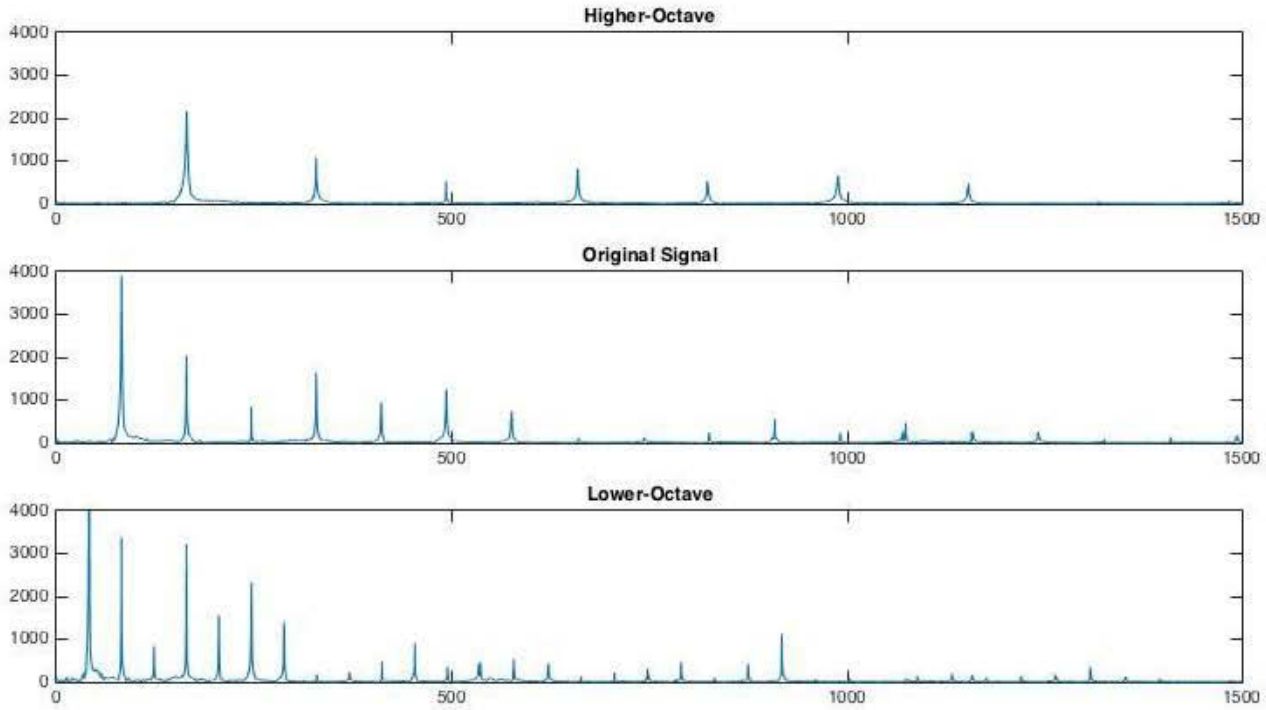

Figure 1: Frequency spectra of the higher-octave signal, the original signal and the lower-octave signal generated by resampling of the entire input, from top to bottom. Amplitude (y-axis) vs. frequency in hertz (x-axis).

The frequency components were shifted as desired for both octaves. The higher-octave signal has doubled frequency components and the lower-octave signal has halved frequency components. Furthermore, when played through a speaker, the audio sounds as is desired for both octaves.

Although the sound quality is good for a melodic input signal, there are two problems with this algorithm in its current version. First, the resampling process

modulates frequency as well as time and thus the duration of the resultant signals do not match the original signal. This is a problem that must be fixed in order to implement a real-time version of this algorithm. As a result of this time scale modification, changes in the pitch of the input signal do not occur at the same time as changes in the output. For example, if a 4 second signal has a pitch of frequency 440 Hz for the first 2 seconds and then shifts to a 550 Hz pitch for the next 2 seconds, the resampled higher-octave would be 880 Hz for 1 second then shift to 1100 Hz for 1 second. Similarly, the lower-octave signal would just be a 220 Hz pitch for 4 seconds. The outputs would not shift in pitch in synchrony, which is nonideal. Thus, in order to work for real-time inputs, changes needed to be made to this algorithm to allow for it to dynamically adjust to changes in the pitch of the input. The solution to this problem is discussed in phase 2.

Phase 2: Segmenting

Instead of resampling the entire signal of length n in a single for-loop iteration, the input signal was split into many smaller analysis frames of length 4200 data points. The reason this specific analysis frame length was selected will be explained with results from this version of the algorithm. Each analysis frame is then resampled independently, producing output frames of length ½ of the analysis frames. These frames were organized such that the first 2100 data points of frame N_i overlapped with the last 2100 data points of frame N_{i-1} so that each data point in the input was processed twice by the algorithm. This overlapping was done because the resultant output frames are half the length of the analysis frames and thus twice as many of them are required to

fill the same duration as the original signal. This step was implemented to compensate for the time-scale modification that is introduced as a result of resampling.

By splitting the input signal into many smaller analysis frames, the output harmony is able to adapt to changes in the pitch of the input because a new harmony output is generated with each shift of an analysis frame, obtaining new pitch information with each shift. The algorithm can adapt to changes in the pitch of the input signal at the rate at which the analysis frame shifts, which is inversely proportional to the analysis frame length, n_a. The frequency of analysis frame-shifting is called the *analysis frequency* (f_a) of the algorithm and is given in hertz by the proportionality $f_a = f_s / n_a$, where f_s is the sampling frequency of the input signal. The MATLAB code for this version of the algorithm is appended under Version 2. To verify that the algorithm works as intended, the frequency spectra of the resampled signals are shown below in Figure 2:

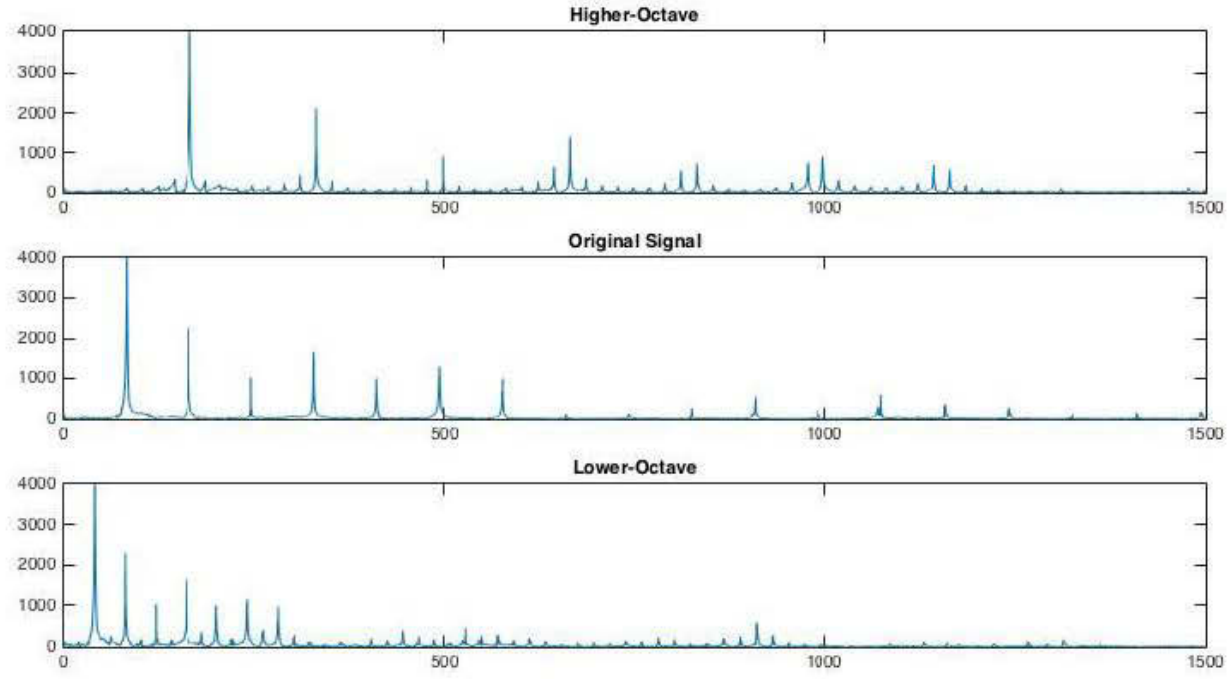

Figure 2: frequency spectra of the higher-octave signal, the original signal and the lower-octave signal generated by iterative resampling of analysis frames, from top to bottom.

We see that the frequency components of both signals are shifted in the intended and expected manner, once again. However, you may also notice many small frequency artifacts that appear at a fixed frequency interval. When the audio output is played back through a speaker, there is an audible clicking that is present in both harmony output signals. To understand the cause of these clicks, the time-domain representations of each signal are shown below in Figure 3:

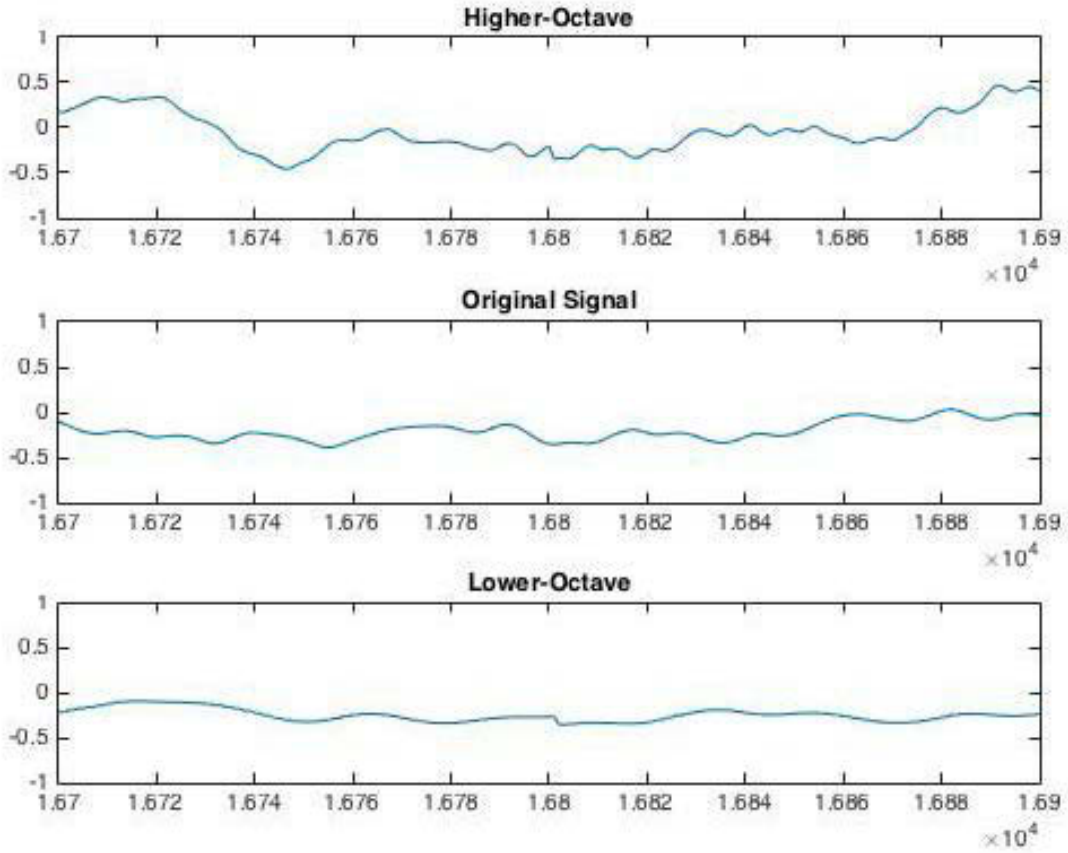

Figure 3: Output signal amplitude (y-axis) plotted against sample number (x-axis). 200 sampled points (~4.5 ms at 44.1 kHz) of audio signal data are shown for the higher-octave, original signal, and lower-octave signals, from top to bottom.

Upon further inspection of the output signal shown in Figure 3, we observe one of many small discontinuities that occur at intervals of 2100 data points (at 1.68*10^4 in Figure 3). The audible click heard in the signal is caused by these discontinuities, which are a result of the shifting of the analysis frame. The analysis frame length is 4200 samples but the algorithm only shifts the frame by 2100 samples after each resampling iteration so that there is enough overlap in the analysis frames to compensate for data lost in the resampling process. Therefore the start- and end-points of the output frames are not always of equal value and thus the signal jumps in a discontinuous fashion from one level to another, as was predicted in theory. This discontinuity is heard as an audible click when the output is played through a speaker.

To test the hypothesis that the clicking was occurring as a result of the shifting analysis frame and was thus occurring at a fixed rate, a simple sinusoidal test was implemented. Two sine waves, one whose wavelength was an integer factor of the analysis frame length and one whose wavelength was *not* each served as inputs into the algorithm. It was expected that the sine wave whose wavelength is an integer factor of the analysis window length will have no discontinuities after resampling, since the endpoints of each analysis frame and output frame are always 0. Conversely, a sine wave whose wavelength is irregularly related to the analysis frame length will have nonzero endpoint values and thus discontinuities will occur where the analysis frame shifts. The results of these tests are shown below in Figures 4 & 5:

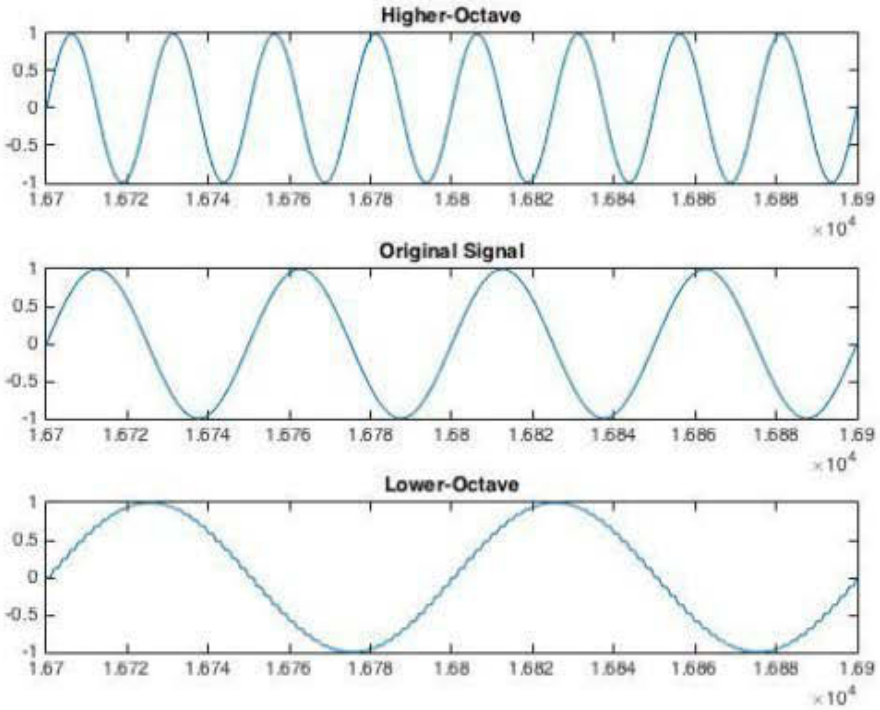

Figure 4: Sinusoid (f = 882 Hz, 1/50 Hz normalized to 44.1 kHz) whose wavelength is an integer factor of the analysis frame length (with octaves) plotted in time domain. (Analysis frame shifts at sample $n = 1.68*10\hat{}4$)

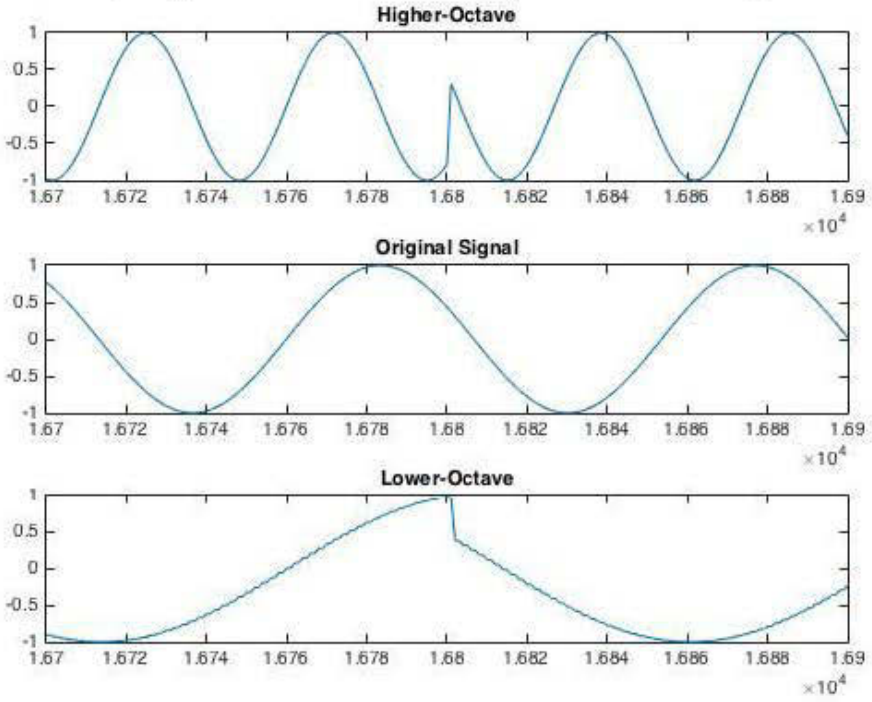

Figure 5: Sinusoid (f = 471 Hz, ~1/93.63 Hz normalized to 44.1 kHz) whose wavelength is *not* an integer factor of the analysis frame length (with octaves) plotted in time domain. (Analysis frame shifts at sample $n = 1.68*10\hat{}4$)

As you can see, the discontinuity occurs exactly where we expected in the signal whose wavelength does not adequately fit the analysis frame length (Figure 5). In the case where the wavelength does fit into the analysis frame, we see no discontinuity (Figure 4). Thus we know the discontinuities and resultant clicks occur at a fixed and predictable rate, which allows us to more easily deal with the problem of removing them from our outputs.

Phase 3: Crossfading

It is desirable to remove the audible discontinuities from the output signal. As was discussed previously in theory, the chosen technique to do this in this project is through crossfading. In order to implement crossfading, the resampling algorithm demonstrated in Phase 2 must once again be scaled up by a factor of two so that there are two output signals that are phase-shifted so that the clicks occur 180 degrees out of sync with one another. Once this is done, the algorithm modulates the amplitude of each of the output signal frames by a triangular distribution of equal length of the output frames so that the endpoints have 0 amplitude and the center sample is modulated by a factor of 1. By doing this, the clicks are diminished and the combined output of the two signals is a linear combination of the purely periodic components of the signals. To implement this in MATLAB, a triangular distribution of the desired length was created and stored in an array to be referenced inside the resampling for-loops to perform the amplitude modulation. The for-loops then operate in much the same way as they did in the second version of the algorithm, but produce twice as much data. Each output frame that is generated by the resampling for-loops are then

modulated point-by-point by the triangular distribution and a final output that is a summation and concatenation of all of the output frames is generated. This MATLAB algorithm that generates enough output data to then implement a crossfaded output is appended in Version 3.

To see if the crossfading method successfully removed the discontinuities while still maintaining the melodic qualities of the signal, the fourier spectra of the crossfaded outputs were calculated and plotted, shown below in Figure 6.

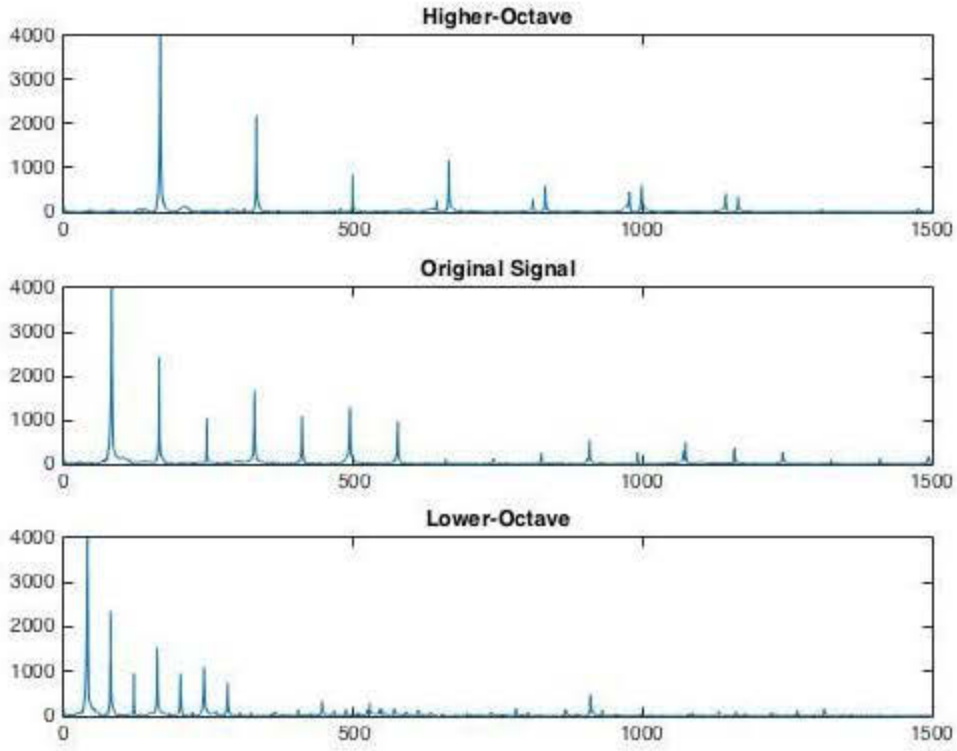

Figure 6: Fourier spectrum magnitude (y-axis) vs. frequency in hertz (x-axis) of the crossfaded output signal.

As you can see, the small frequency artifacts that were a result of frame-shifting have been greatly diminished by crossfading, although the spectrum is not perfectly doubled and halved for the respective higher and lower octaves. There are still small

frequency artifacts that are likely introduced by the amplitude modulation by a triangular distribution, but the result is still a vast improvement from before implementing crossfading. To see how the clicks themselves have been affected by crossfading, sample-time-domain plots of where the discontinuities occur in each signal are shown below in Figure 7.

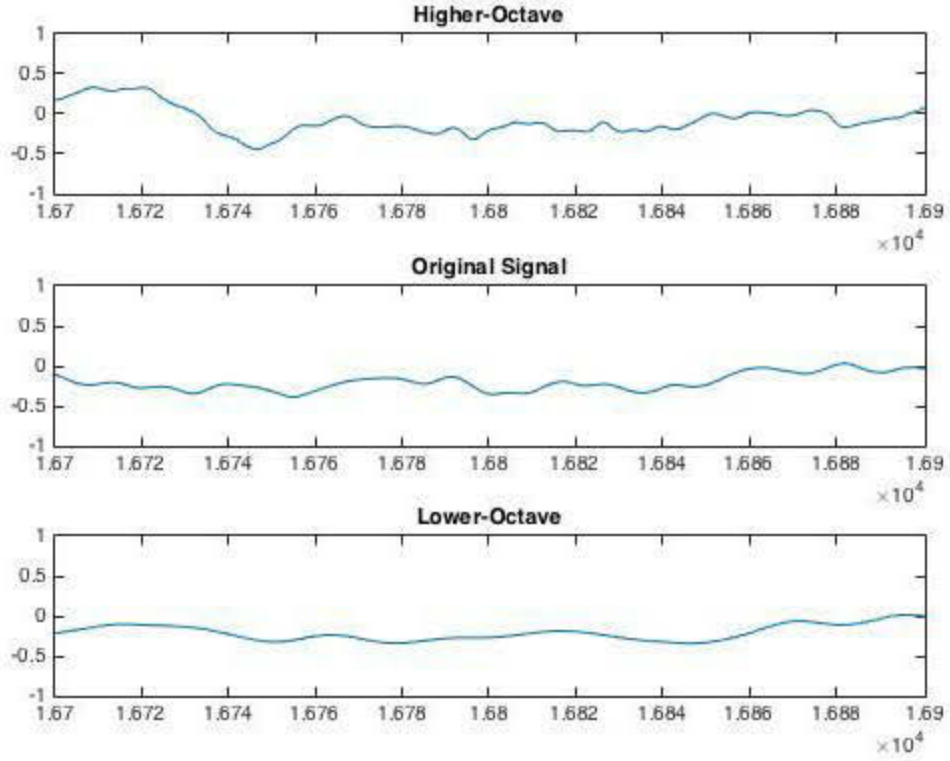

Figure 7: Sample-time-domain plot of input signal and generated harmonies after cross fading. (Analysis frame shifts at sample *n = 1.68*10^4*)

Compared to the same signal resampled iteratively without crossfading, the above is a great improvement. There is no noticeable jump in the signal as the analysis shift occurs. The output is drastically improved when played through a speaker because the click is no longer audible. There is, however, a slightly audible wavering in the signal due to the amplitude modulation implemented by crossfading. Even so,

crossfading improves the quality of the output greatly and thus the final implementation of the algorithm on the DSP chip is to include crossfading to smooth the transitions between analysis frames. Thus the third version of the algorithm described in this section served as the working prototype upon which the real-time algorithm in C would be based.

Real-Time Implementation in C and Hardware

The ultimate goal of this project was to build a guitar pedal by using a digital signal processing (DSP) chip that implemented real-time signal harmonization with an algorithm written in C. The C program was written in the free integrated development environment (IDE) provided by Texas Instruments with the device, Code Composer Studio version 6.1.0. The prototype MATLAB algorithm described previously worked only for static data and iteratively generated octave outputs with for-loops. While a real-time version of a harmonization algorithm by resampling would implement many of the same concepts in theory, the practical implementation would be fundamentally different. Due to time limitations for the project, the real-time C algorithm was not written to generate lower-octave harmonies and thus only higher-octave harmonies were generated by the chip for real-time harmonization.

A large amount of time was spent during the early stages of this project trying to use the TI C2000 Piccolo F28027 DSP chip. This device is a newer, smaller, faster version of the chip used in the final implementation. I hit many dead ends in trying to write my program on this device. In particular, it was very difficult to write code to initialize the CPU and peripherals as well as communication between these modules and

the analog-to-digital converter (ADC) and the digital-to-analog converter (DAC) to implement even simple input/output tasks. I tried using examples from TI's website and other sources but many of these resources led to problems involving the TI controlSUITE software and device connectivity issues. After several weeks of finding nothing but dead ends in trying to program on the F28027, I decided to use a DSP chip that I was more familiar with, the F28069.

The final implementation of the harmonization algorithm was done on the TI C2000 F28069 Control Card and Experimenter's Kit. The F28069 is a device that I was using at the same time as this project for a course in digital signal processing. In this course we wrote C programs within a code shell that was written by the Technical Training Organization (TTO) at Texas Instruments and adapted for the DSP course by Professor Erik Cheever to allow for easy initialization of the device. I decided to use this code shell as the framework within which I could write my C harmonization algorithm. This shell code "E71Shell.c" can be found on Erik Cheever's webpage, which is given in the References section of this report.

The F28069 also has two 3.5 mm audio jacks that make interfacing with guitar signals and other audio signals very easy to implement. The device also has built-in level-shifting circuitry to allow a guitar signal, centered around 0 V, to be automatically shifted to be centered around 1.65 V to be transmitted into the device. Thus no external circuitry was required to perform level-shifting; however, I will note that a simple analog op-amp inverting amplifier circuit with gain -10 was used to adjust the peak-to-peak voltage of the input signal to a workable range for the ADC on the F28069.

Now I will explain explicitly how the C program works to generate octave harmonies from an input signal. Within the shell code a sampling rate is selected and the ADC begins converting the sampled signal from the input ports to digital values that readable from the ADC. Each time a new value is converted by the ADC, the harmonization function "E90func.c" that was written for this project is called. To implement resampling for octave harmony generation, every other data point is discarded using an if-statement and by toggling a conditional bit. For each data point that is saved, four output buffers are filled using another if-statement. This if-statement uses an index to keep track of where within each of the output buffers the data is being stored. A constant value of half of the buffer length are added and subtracted from the index such that the buffers are 180 degrees out of phase so that they can later be summed and concatenated to form a continuous output signal. As the data is read into the output buffers, it is scaled by a triangular distribution of the same length as the output buffer that is stored in RAM to implement crossfading. Due to restrictions of the size of the RAM available on this device, the maximum output buffer size I could use was 128 samples in length. This corresponds with an input buffer size of 256 data points in the MATLAB algorithm, which is more than an order of magnitude smaller than the ideal analysis frame length of 4200 data points that was used in the final prototype.

The four output buffers were combined into a single output to be sent to the DAC and transmitted to the electric guitar amplifier. This was done once the buffer index reached the maximum size of the output buffer, at which point a new output array was filled with the summed and concatenated data from two of the four output buffers at a

time. Finally, this combined output was sent to the DAC, which transmitted the resampled, harmonized output signal as an analog signal to the output port. This signal was sent to the guitar amplifier to be played through the speaker. To interface 3.5 mm audio output of the DSP chip with the guitar amplifier, the 3.5 mm output was converted to a quarter-inch output to be plugged into the guitar amplifier.

Results

The resampling harmonization algorithm implemented on the DSP chip worked very well for high frequency ($f > \sim 2\ kHz$) periodic signals. When high frequency sine waves were input into the device and viewed on an oscilloscope, it was observable that the algorithm was working as intended and the higher-octave output that was generated had few distortions and no discontinuities. The results obtained from inputting various sine waves to the F28069 DSP chip are shown below in Figures 8 and 9.

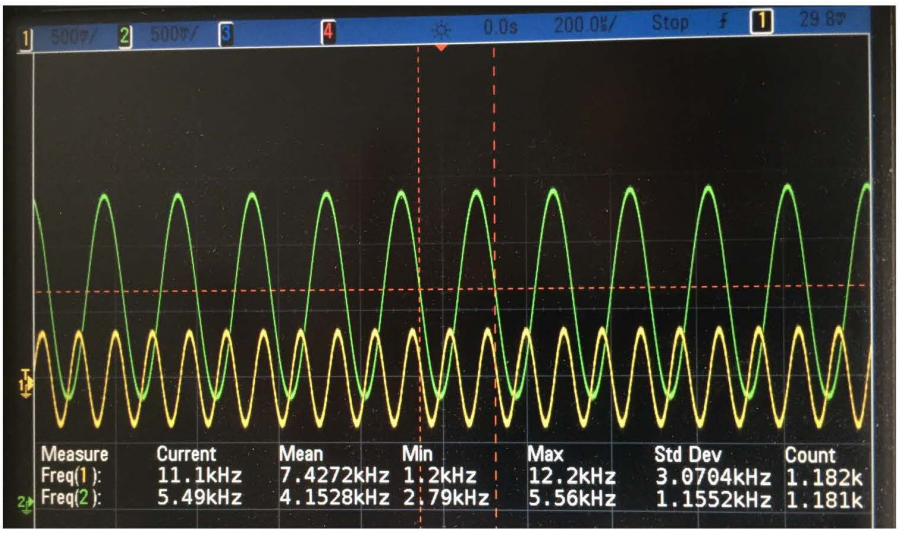

Figure 8: 5.49 kHz sinewave input (green) and 11.1 kHz harmony output (green) generated by the the DSP chip running the resampling algorithm.

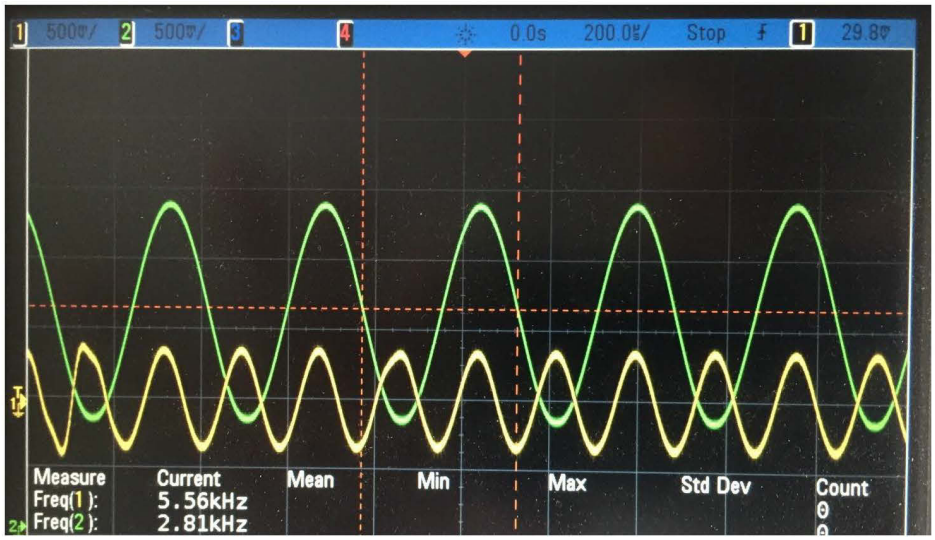

Figure 9: 2.81 kHz sinewave input (green) and 5.56 kHz harmony output (yellow) generated by the DSP chip running the resampling algorithm.

In Figure 9, you can observe two points where crossfading smoothed out discontinuities where the algorithm shifted between output buffers. The result is an octave harmony with only small distortions. In Figure 8, there are no observable buffer shifts and the output appears to be a pure sinusoid. The frequency of these inputs was high enough that the frequency of shifting buffers was much lower than the frequency of the resampled output and thus the effects of shifting buffers were not overbearing. However, if the input signal were of a much lower frequency (as we expect the input for guitar signal inputs will) the small buffer size will mean that the buffer is switching almost as frequently as the signal is oscillating. Thus one can expect that the output in this case would be very distorted since many smoothed discontinuities from shifting and crossfading will be present in the signal and thus not much of the output will be the periodic information we desire. Figures 10 and 11 below show the output of the device given various inputs from a guitar.

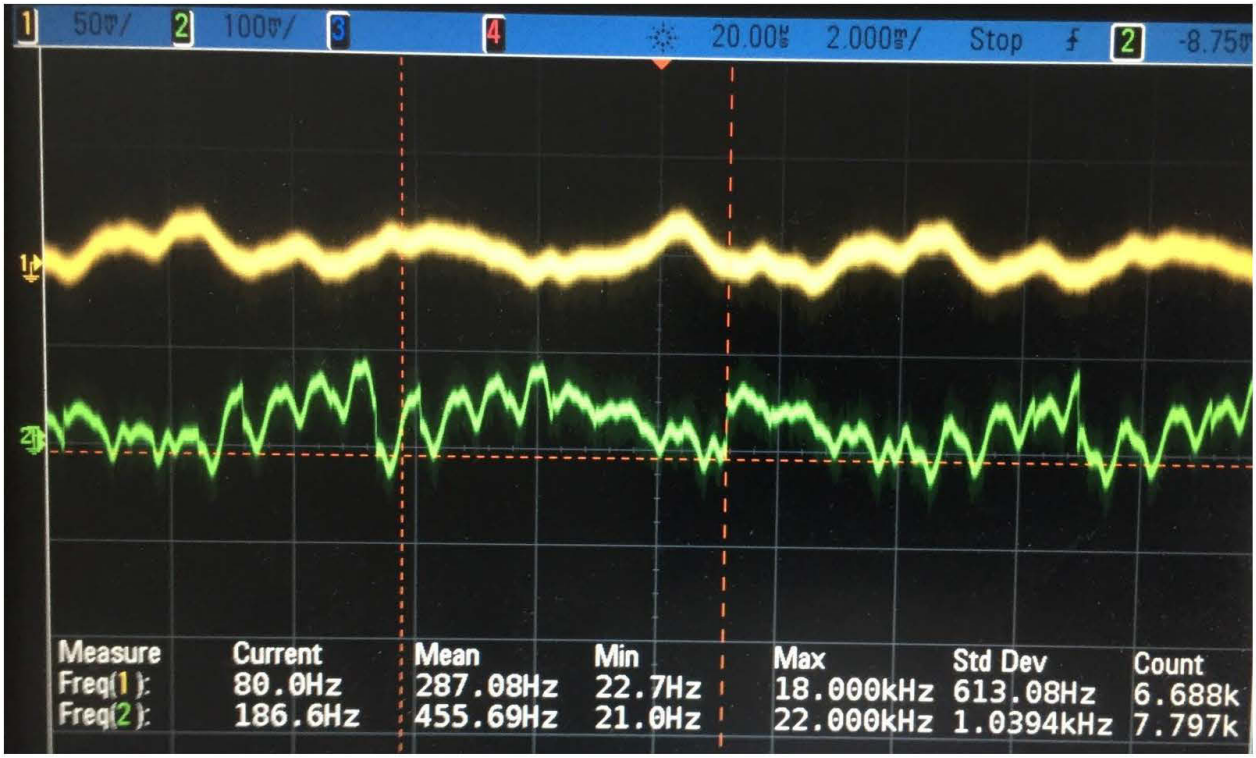

Figure 10: Input (yellow) from a guitar signal and higher-octave output (green) generated by the DSP chip running the resampling algorithm.

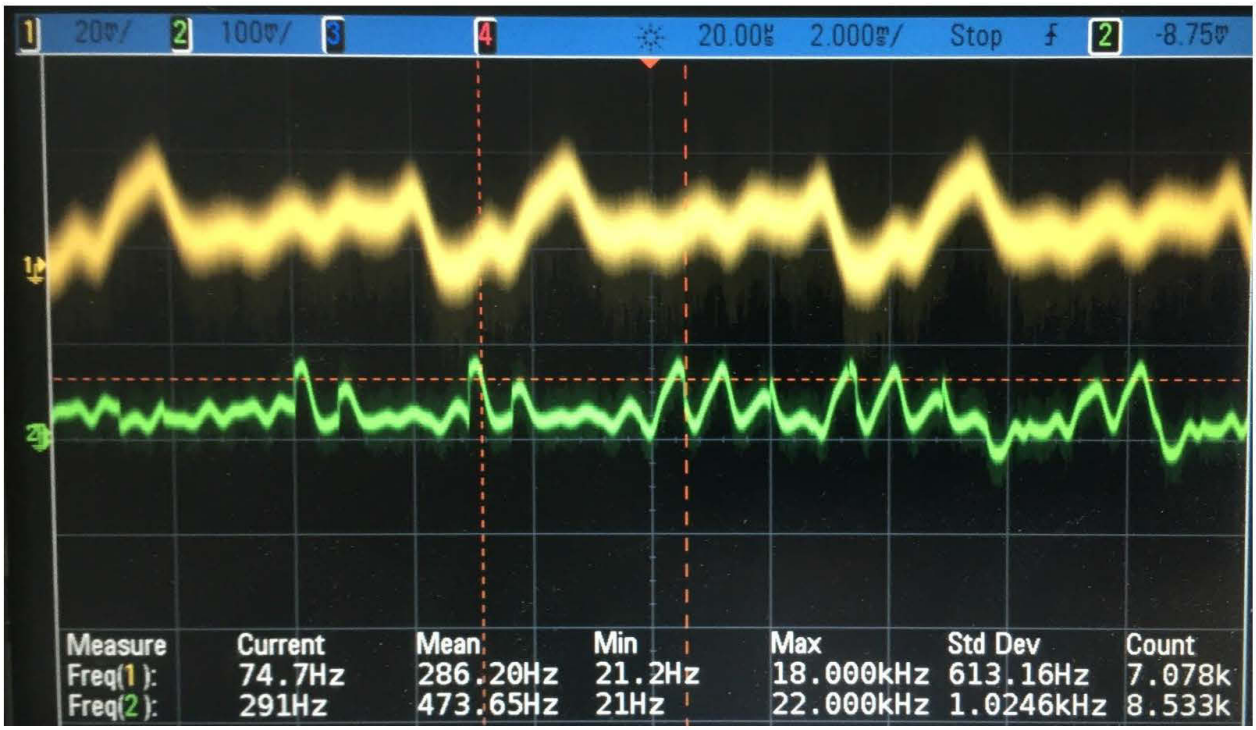

Figure 11: Input (yellow) from a guitar and higher-octave harmony output (green) generated by the DSP chip running the resampling algorithm.

The output signal given an input from an electric guitar was transmitted successfully through the device to the guitar amplifier. The sound it produces vaguely resembles that of a guitar, but is in many ways very distorted and unpleasant. The algorithm is working exactly as it is intended to work, but a buffer size of 128 samples is unideal and causes the frequency of output buffer shifting to be very high. This introduces many discontinuities into the output, but these are to some extent smoothed by crossfading. As you can see from Figures 10 and 11, the distortions introduced by switching buffers are in some cases somewhat smoother than a purely vertical leap in the signal, but there are still many jumps in the output signal. These distortions dominate the output signal, causing it to contain lots of noise and sound very abrasive, even if these distortions are diminished by crossfading.

Discussion and Conclusions

The algorithm works exactly as intended and as expected. By resampling the input signal at half the original sampling frequency into several output buffers and combining them, it is possible to generate a continuous output signal that is twice the frequency of the input signal in real-time. The algorithm did this with some success for most signals, however the main shortcoming of this harmonization module was its limited buffer size due to RAM restrictions in the DSP chip. The very small buffer size of 128 samples meant that the frequency at which the output buffers were switched and crossfaded was very high, greatly distorting the output signal. This resulted in greatly reduced signal quality for audio applications, however this could be easily fixed by implementing the same algorithm on a device with greater RAM capabilities.

The idealized version of the algorithm implemented in MATLAB produces both higher- and lower-octave harmonies by resampling and signal doubling respectively. The output buffer size of 2100 samples introduces a semi-frequent click that was then smoothly diminished by crossfading the output signals by scaling by a triangular distribution. The end result, as was observed in the fourier spectrum of the outputs, was a signal whose frequencies were properly doubled and halved for the respective octaves and with few frequency artifacts introduced. The sound quality of this algorithm was very good, the buffer length was short enough that the algorithm adapted quickly to changes in pitch of the input, but long enough to contain primarily periodic signal information. Thus this algorithm would be suitable for use in a real-time system with improved hardware capabilities to design a guitar effect pedal.

Acknowledgements

I would like to thank Professors Erik Cheever and Matt Zucker for advising me throughout the duration of this project. Without the help of Professor Erik Cheever in particular, I would not have been able to complete many aspects of this project and I am very thankful for his assistance. I would also like to thank Perry Nguyen '16 and Simon Bloch '17 for teaching me some basic coding techniques for the C programming language. Their help allowed me to begin to translate my MATLAB script to a functioning C program. Finally, I would like to thank the Ed Jaodi and the Swarthmore College Engineering Department for providing limitless assistance with hardware to implement many of the steps of this project.

Literature Cited

1. Smits van Waesberghe, Joseph. "*A Textbook of Melody: A course in functional melodic analysis*" American Institute of Musicology. 1955.
2. Case, A. "*Guitars*" (http://recordingology.com/in-the-studio/guitars/)
3. Flanagan, J.L. & Golden, R.M. "*Phase Vocoder*" The Bell System Technical Journal. pp. 1493-1509. November, 1966.
4. Rumsey, F. & McCormick, T. "*Sound and Recording*" Burlington: Focal Press. pp. 241, 282-284. 1992.
5. Prof. Cheever's webpage: http://www.swarthmore.edu/NatSci/echeeve1/Class/e71/E71Index.html

Code Listings:

MATLAB Scripts:

Version 1:

```
%% Harmonizer Version 1
% Cosmo Alto
% Resample Entire Signal

clear
close all

% Read in .wav file, obtain data (y) and sampling frequency (fs)
file = 'guitarnotes.wav';
[y, fs] = audioread(file);

% Choose start time in seconds and end time of recording and clip original file to
% within those bounds
start = fs*7;
finish = fs*13;
clip = y(start:finish,1);

% Find the start of the nonzero data to synchronize the beginning of the
% input and harmony. Data just represents the
data = find(clip);
nstart = data(1);
nfinish = length(data);
clip = clip(nstart:nfinish-1,1);

% Create empty array to store harmony
cliplow = [];
cliphigh = [];

% Double each point in clip to generate lower-frequency octave
for i = 1:L
    cliplow((2*i)-1,1) = clip(i);
    cliplow(2*i,1) = clip(i);
end

% Resample each point in clip to generate higher-frequency octave
for i = 1:(L/2)
    cliphigh(i,1) = clip(2*i);
end

% Edit the original clip to be the same length as the lower-frequency clip
Llow = length(cliplow);
xlow = linspace(1,Llow,Llow);
```

```
% Edit the original clip to be the same length as the lower-frequency clip
Lhigh = length(cliphigh);
xhigh = linspace(1,Lhigh,Lhigh);

% Clip audio to same length
cliplow = cliplow(1:Lhigh,1);
clip = clip(1:Lhigh,1);
clipall = (clip+cliplow+cliphigh)/2;

% Play audio
player = audioplayer(clipall, fs); % prep for player
play(player); % play the original signal in the specified window
```

Version 2:

```
%% Harmonizer Version 2
% Cosmo Alto
% Incrementally Resample Signal

clear
close all

% Read in .wav file, obtain data (y) and sampling frequency (fs)
file = 'guitarnotes.wav';
[y, fs] = audioread(file);

% Choose start time and end time of recording and edit original file to
% within those bounds

start = fs*7;
finish = fs*13;
clip = y(start:finish-1,1);

Ls = 4200;          % Ls length of buffer segments
Ns = len/(2*Ls);    % Ns number of buffer segments

for i = 1:Ns
   full = (i-1)*(Ls/2); % calculate amount of data already filled in each iteration to shift
index

   for j = 1:Ls/2      % half of analysis frame since we are doubling
      cliplow((full + 2*j), 1) = clip((full + j), 1);
      cliplow((full + 2*j + 1), 1) = clip((full + j), 1);
   end

   for j = 1:Ls        % double each of the input points to generate
      cliphigh((full + j), 1) = clip((full + 2*j), 1);
   end
```

```
end

% Combine all outputs into one continuous output signal
L = length(cliphigh);
cliplow = cliplow((1:L), 1);
clip = clip((1:L), 1);
clipall = (cliplow + cliphigh + clip)/3;

player = audioplayer(clipall, fs); % prep for player
play(player); % play the original signal in the specified window
```

Version 3:

```
%% Harmonizer Version 3
% Cosmo Alto
% 4x Output Buffer and Crossfade Implementation

clear
close all

% Read in .wav file, obtain data (y) and sampling frequency (fs)
file = 'guitarnotes.wav';
[y, fs] = audioread(file);

% Choose start time and end time of recording and edit original file to
% within those bounds
start = fs*7;
finish = fs*13;

% Clip the audio to the parts we want
clip = y(start:finish-1,1);
len = length(clip);

% Clip the audio we want
clip = y(start:finish-1,1);
len = length(clip);

Lb = 8400;       % Lb length of analysis frame
Li = Lb/4;       % Li amount to increment each iteration
Lo = Li*2;       % Lo length of output buffer, half of analysis frame
Ns = len/Li - 7; % Ns = number of analysis frames/iterations of analysis frame

% Triangular distribution for crossfading
pd = makedist('Triangular','a',0,'b',Li,'c',Lo);
pdx = 1:1:Lo;
pdf1 = pdf(pd,pdx);      % generate triangle distribution pdf
pdf1 = pdf1/max(pdf1); % maximum amplitude = 1
```

```
% Create output arrays
cliplow = zeros(len,1);
cliphigh = zeros(len,1);

for i = 1:Ns     % for each analysis frame
   full = (i-1)*(Li); % calculate amount of data already filled, shift analysis frame
% Generate lower-octave in temporary array
   for j = 1:Li      % half of analysis frame since we are doubling
      lowtemp((2*j), 1) = clip((full + j), 1);
      lowtemp((2*j + 1), 1) = clip((full + j), 1);
   end
   lowtemp = lowtemp(1:Lo, 1);  % cut to size of output buffer
% Modulate output by triangular dist for xfade
   for j = 1:Lo
      lowtemp(j) = lowtemp(j)*pdf1(j);
   end

   for j = 1:Lo
      cliplow(full + j) = cliplow(full + j) + lowtemp(j);
   end

% Generate higher-octave
   for j = 1:Lo        % sample every other point in original clip
      hitemp((j), 1) = clip((full + 2*j), 1);
   end
% Modulate both outputs by triangular dist for xfade
   for j = 1:Lo
      hitemp(j) = hitemp(j)*pdf1(j);
   end

   for j = 1:Lo
      cliphigh(full + j) = cliphigh(full + j) + hitemp(j);
   end
end

% Combine outputs
clipall = (cliplow + cliphigh + clip)/3;

player = audioplayer(clipall, fs); % prep for player
play(player); % play the original signal in the specified window
```

C Function within "E71Shell.c"

```
void E90Func(void) // function called after each ADC conversion
{
      static int inData, outData;  // declare input and output arrays
      static int bufInd = 0;       // declare buffer index
      static int getData = 0;      // declare ping-pong bit
      static int obufInd = 0;      // declare output buffer index

      inData = adcRight;  // Collect data and put into arrays
   if (getData==0){
         bufInd++;
         if (bufInd == BUF)  bufInd=0;  // fill output buffers
         d[bufInd] = inData*TRI[bufInd];
         c[(bufInd + INC) % BUF] = inData*TRI[(bufInd + INC) % BUF];
         b[(bufInd + 2*INC) % BUF] = inData*TRI[(bufInd + 2*INC) % BUF];
         a[(bufInd + 3*INC) % BUF] = inData*TRI[(bufInd + 3*INC) % BUF];
         getData = 1; // toggle to discard next data point
   } else {
         getData=0;  // discard every other data point
   }

   // Process data and combine output buffers
   obufInd++;
   if (obufInd==BUF) obufInd=0;
   switch (obufInd/(BUF/4)) {
   case 0:
         outData = d[obufInd]+c[obufInd + BUF/2];
         break;
   case 1:
         outData = d[obufInd]+a[obufInd - BUF/2];
         break;
   case 2:
         outData = b[obufInd]+a[obufInd + BUF/2];
         break;
   case 3:
         outData = b[obufInd]+c[obufInd - BUF/2];
          break;
   }
dacLeft = outData;
}
```