2019

Refining a Phase Vocoder for Vocal Modulation

Benjamin E. Crystal
University of Vermont

Refining a Phase Vocoder for Vocal Modulation

Ben Crystal

University of Vermont College of Engineering and Mathematical Science

Honors College Thesis

March 31 2019

The University of Vermont

Contents

1 Introduction

1.1 Motivation

Vocal harmonies are a highly sought-after effect in the music industry, as they allow singers to portray more emotion and meaning through their voices. The chords one hears when listening to nearly any modern song are constructed through common ratios of frequencies (e.g., the recipe for a major triad is 4:5:6). Currently, vocal melodies are only readily obtainable through a few methods, including backup singers, looper-effects systems, and post-process overdubbing. The issue with these is that there is currently no publicly-available code that allows solo-artists to modulate input audio to whatever chord structure is desired while maintaining the same duration and timbre in the successive layers.

This thesis plans to address this issue using the phase vocoder method. If this modulation technique is successful, this could revolutionize the way vocalists perform. The introduction of real-time self harmonization would allow artists to have access to emphasized lyrical phrases and vocals without needing to hire and train backup vocalists. This phase vocoder would also allow for more vocal improvisation, as the individual would only need to know how to harmonize with themselves and would thus not be relying on interpreting how backup vocalists plan on moving the melody when creating more spontaneously.

1.2 Prior Solutions for Audio Modulation

People have been using technology to layer effects in music since the early 1980s. This is due to the creation of the sequencer, which is a device that allows a performer to repeat specific recordings in whatever pattern they like, and DAWs, or digital audio workshops that allow people to post-process and edit recorded audio. However, there are two devices that have been primarily used to create harmonies in live performances. The first is the "looper-effects pedal", originally created for guitars, which artists can use to record individual layers within their desired time frame. This limits live performers by requiring them to spend a minimum of the amount of time of each section times the number of layers. If they want to repeat a 10 second sentence three times in harmony to form a major triad, for example, it will take a minimum of thirty seconds to create the desired effect if no time was wasted. This can ruin the fluency of the story that they are trying to convey, and also requires all recorded layers to be perfectly matched to one another manually. A one-of-a-kind solution to creating harmonies in live performances is Jacob Collier's "Harmoniser" [1]. The musician controls this instrument by singing into a microphone and selecting desired frequencies on a synthesizer keyboard. The audio coming through the microphone is then modulated so that the raw center frequency aligns with the expected center frequency of all pressed keys. However, this instrument requires the user to

not only be a skilled singer but also to have technical piano prowess, which extremely limits the potential user base of the process.

1.3 Contributions

This thesis will fulfill two major gaps in the field. First, it will provide a clear, concise understanding of how a phase vocoder is able to modulate a recording's pitch without altering its duration. This thesis will not show examples of the implementation itself, but rather the general knowledge and theory to understand what is happening to the raw audio data as it is streamlined from a single voice to many.

Second, it will provide a novel attempt to implement this phase vocoder methods for the application of real-time audio modulation. This would be a major feat in the music industry, as generally live performance and real-time audio "stacking" are mutually exclusive, and to allow artists to coordinate voices with themselves would allow for more meaningful and deliberate performances with less hassle than ever before.

1.4 Outline

Section 2, *Background*, begins with a brief history of harmony in music, which can show the importance of the implementation of this technique in music genres and groups that don't historically have access to an assembly of vocalists, as well as the importance and omnipresence of the desired effect in the music of today. Next, it contains both technical and musical background information to assist in understanding how a phase vocoder modulates pitch and provides the vocabulary that will be used further throughout this document. Then, a brief overview of the uses and implementations of the phase vocoder technique will be addressed.

Section 3, *Methods*, walks the reader through a large flow chart (visible in Figure 4). Each section attempts to explain the calculations through the full process that makes audio modulation possible.

Section 4, *Results*, evaluates how the phase vocoder described in this thesis compares to the requirements set forth. First, whether the phase vocoder aspect was accurate to expectations in the recreation of chords from a fixed audio input or not was discussed, as well as how this was tested. Then, the success of whether or not not this phase vocoder lives up to the requirements for real-time output delay is addressed, as well as how tests proving or denying such are completed.

Section 5, *Conclusion and Future Work*, discusses the shortcomings of this project, what could be done to solve these issues, and what may be potential next steps in furthering the implementation of real-time phase vocoder techniques in live musical performances.

2 Background

This chapter will provide the reader with a brief description of the history of harmonies, the musical and technical background needed to understand this thesis, and finally an understanding of where and how the phase vocoder process is currently used.

2.1 A Brief History

The concept of harmony is one of the most fundamental components of music theory. Harmony, which even outside of the realm of music means "agreement", allows for the progression of multiple sources in tandem through whatever medium they come from, be it a song, a story, or a set of machines. It is no wonder, then, why harmonies are so prevalent within music- it allows the progression of a story from multiple sources at once, holding a much stronger connotation than a single source could exhibit.

The world of electronics has been trying to dive into music since the early 1900s, which essentially were modification to previously existing devices such as electric guitars with amplifiers. The first fully electronic instruments have only made their appearance within the last 50 years- synthesizers, which produced fully computer-generated sounds, often played in a keyboard layout. In today's digital age, processors have become fast enough for electronic instruments and instrument modifiers to become almost as commonplace as their analog counterparts (e.g. autotuning devices, guitar pedals, and new-age instruments like the Roli *Seaboard*$^{\text{TM}}$ [2] and the Artiphon *Instrument 1*$^{\text{TM}}$ [3] that attempt to bridge the gap between electronic and acoustic by allowing digital instruments to be played in an analog-like manner).

One of the most ancient applications of harmony was, however, far before digital instruments. In Western music, harmonic form began developing around the year 1600 through religious Catholic music. Originally called counterpoint (note against note), it involved the coincident singing of two tales without any dissonance. As instruments evolved, it became easier for a fewer number of performers to be able to tell more and more intricate musical tales. In contrast to the instrument evolution, it was not until early 1990s where a solo performer could share multiple tails through voice with the creation of looping systems. Almost thirty years later, this ability has become easier, but is still very restrictive in its application.

2.2 Musical Foundation

This section will briefly explain the musical terminology and theory needed to understand the phase vocoder code.

To understand the effect that this code has on audio, one must first understand harmony. Though the

application history was discussed briefly above in the "A Brief History" section, what makes something sound harmonious and not dissonant is not a strict rule. Generally, one perceives a set of sounds as "more harmonious" when the individual pitches are of a strong relation with each other. In Figure 1 below, a sine wave is plotted in blue to represent a "tonic", which will always be the fundamental frequency with regards to this project. Musical notation defines a tonic as a reference point for the intervals between notes in a segment of a song. An "octave", a note where the frequency is twice the tonic, is the most harmonious note that can be played with a tonic other than itself. This is because the period is as low as possible, which in Figure 1 in red repeats every 2 *π. The next most harmonious note is a "perfect 5th", seen in green, which has a period when placed against the tonic of 4 * π. From here, additional notes become more dissonant (less harmonic) as they stray from a simple ratio of how these notes fit together. In Table 1 [4] [5] [6], one can see the ratios to the tonic for the most harmonic and frequent intervals.

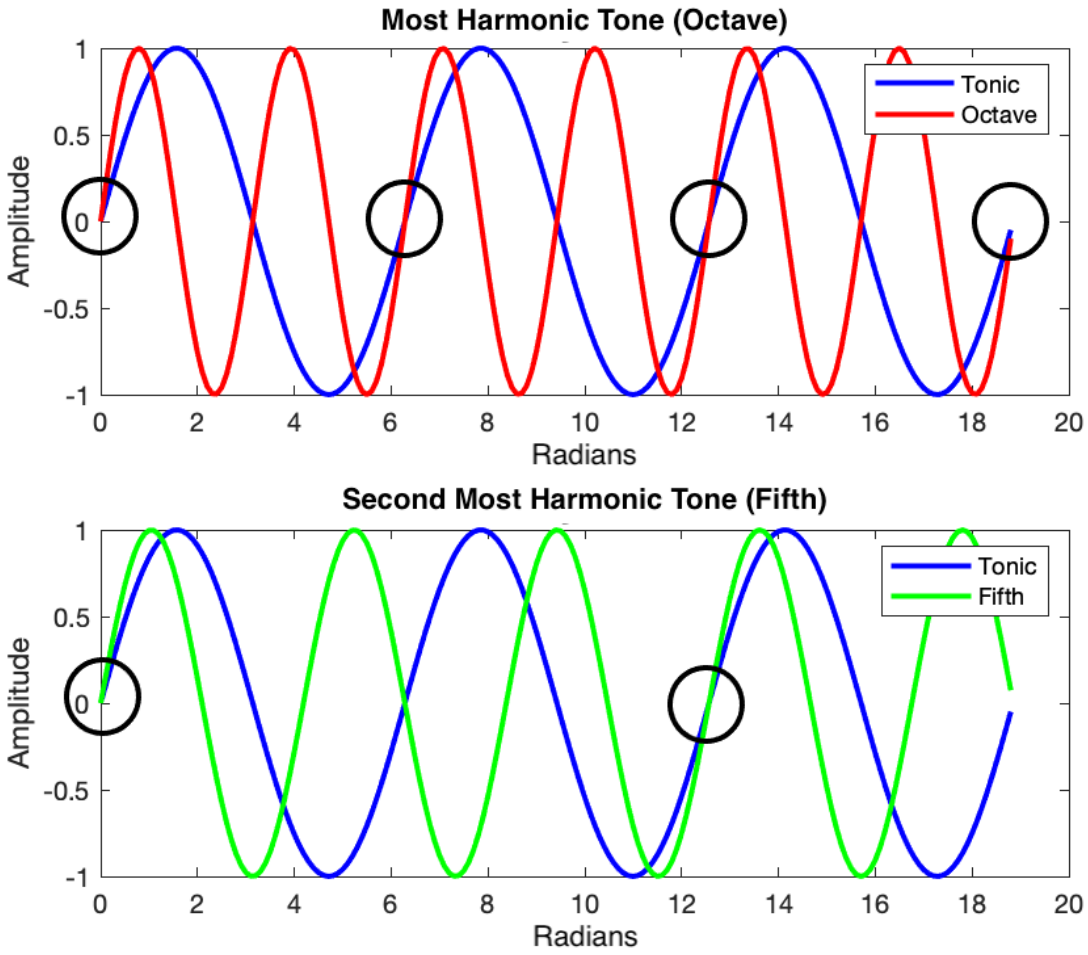

Figure 1: Period of octave (top) and fifth (bottom) against tonic

Note Name	Relationship to Tonic
Tonic	1 : 1
Major 2	9 : 8
Minor 3	6 : 5
Major 3	5 : 4
Perfect 4	4 : 3
Diminished 5	$\sqrt{2} : 1$
Perfect 5	3 : 2
Augmented 5	8 : 5
Diminished 7	5 : 3
Minor 7	16 : 9
Major 7	15 : 8
Octave	2 : 1

Table 1: Interval ratios relating generated tones to the fundamental frequency (tonic)

The code developed to create the phase vocoder analyzed in this thesis is capable of reproducing 16 musical note structures, which can be seen in Table 2. Although these chords were selected as being representative of the most common chord structures in popular Western music, all of these chords and their sequencing were chosen subjectively based on a physical implementation of the phase vocoder separate from this thesis, which is discussed in further detail in the "Conclusions and Future Work" section.

Number	Note Structure/Chord	Intervals
1	Tonic	Tonic
2	Major 3rd	Tonic — Major 3
3	Minor 3rd	Tonic — Minor 3
4	Major Triad	Tonic — Major 3 — Perfect 5
5	Perfect 5th	Tonic — Perfect 5
6	Suspended 4th	Tonic — Perfect 4 — Perfect 5
7	Diminished Triad	Tonic — Minor 3 — Diminished 5
8	Major 7th	Tonic — Major 3 — Perfect 5 — Major 7
9	Octave	Tonic — Octave
10	Augmented Triad	Tonic — Major 3 — Augmented 5
11	Suspended 2nd	Tonic — Major 2 — Perfect 5
12	Half Diminished 7th	Tonic — Minor 3 — Diminished 5 — Minor 7
13	Minor Triad	Tonic — Minor 3 — Perfect 5
14	Diminished 7th	Tonic — Minor 3 — Diminished 5 — Diminished 7
15	Minor 7th	Tonic — Minor 3 — Perfect 5 — Minor 7
16	Dominant 7th	Tonic — Major 3 — Perfect 5 — Minor 7

Table 2: Chord list and their generated tone compositions

Two additional terms that should be understood are tempo and timbres. Tempo is the speed of a particular section of a song. Timbre describes the qualities that differentiate sounds from one another. For example, a piano, trumpet, and a violin could all be playing "Middle C", or 261.6 Hz, at the same volume, but it is very likely that these sounds are distinguishable from one another to even an untrained ear. That is

because the way that the instruments shape and produce their acoustics are different, and the signals that they output are composed of different balances of frequency components around, in this case, middle C.

2.3 Technical Foundation

This section will briefly explain the technical information needed to understand the phase vocoder code. First and foremost, one must understand that the Just Noticeable Difference (JND) that humans can distinguish between sounds is 20 milliseconds, which is what several of the parameters in the code abide by. To capture the full human-range of hearing, which spans from 20 to 20,000 Hz, one must define the sampling rate of their device by following the Nyquist theorem: $f_s \geq 2 * f$, whereas f_s is the sampling frequency and f is the maximum frequency of the signal wished to be captured. Originally, for the phase vocoder that this thesis is based on, the 20,000 Hz doubled up to 40,000 Hz, which was rounded up to 44,100 Hz, as it is the closest standard recording rate, matching with that of CDs.

However, reconsidering optimization for processing times, this sampling rate could be lowered significantly. In choirs, a soprano is the role of the singers with the highest pitched voices. Other than in a few specific cases, this means that their voice is expected to cover the frequency spectrum as high as the note C6, which resides at 1,046.5 Hz. For comparison, the highest note on a standard 88 key piano is C8, at 4,186 Hz, which this is far below. If we wanted to capture the first 6 harmonics, which contains a vast majority of the power in a signal, this would mean that we'd need to capture up to 6,279 Hz (1,046.5 Hz * 6) to properly capture the highest expected human sang note. Though this would require a much higher value for a piano, since this application is focused on vocals, a sampling rate of 16,000 instead of 44,100 Hz would be sufficient for capturing frequencies of up to 8,000 Hz, whereas 6,279 Hz is well within this realm. This allows for what sounds like the same information to be captured using just over one third of the resources at just over one third of the sampling rate.

To allow for the potential that audio being sampled could be reproduced within the JND of 20 milliseconds, one must calculate the size of each audio slice that is modulated at a time based on the sampling frequency using the following equation: $seconds = size/f_s$, whereas size is the amount of samples in each recorded slice and seconds is the desired amount of seconds of each slice. Since this case requires a maximum of 20 milliseconds, this allows for a maximum size of 882 samples when f_s is 44,100 Hz. This value has been rounded down to 512 to allow a bit of time for modulation between when this sample is input and output without increasing the total number of code-cycles drastically. 512 samples equates to just under 12 milliseconds for recording, which leaves 8 milliseconds for the calculations for modulation to be performed.

One crucial mathematical tool utilized in this phase vocoder, as well as many other signal processing

techniques, is the Short Time Fourier Transform (STFT). For a brief understanding of this, the Fourier Transform will first be discussed. The Fourier Transform is a method that allows one to access the frequency domain representations that compose any signal. When this technique is implemented, the output is a variety of complex numbers, whereas the magnitude represents the amount of a specific frequency present within said signal, and the imaginary component represents the phase offset, or where that piece of information is present in one period of a waveform cycle. The equation representing this process is:

$$F(\omega) = \int_{-\infty}^{\infty} f(t) e^{-j\omega t} dt$$

Whereas $F(\omega)$ is the signal's representation in the frequency domain, f(t) is the signal information in the time domain, j is the imaginary number $\sqrt{-1}$, f is the angular frequency and ω is $2 * \pi * f$. Additionally, this process can be reversed to convert from the complex frequency components back into the original signal through the Inverse Fourier Transform:

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(\omega) e^{j\omega t} d\omega$$

In practice, signals can not be measured in truly continuous time, so to get around this for computational purposes, the Discrete Fourier Transform (DFT) is used, which is computationally known as the Fast Fourier Transform (FFT). This particular equation, the Exponential Fourier Series, was developed to perform the same operation on a discrete dataset, and can be seen here:

$$x(t) = \sum_{k=-\infty}^{\infty} X[k] e^{jk\omega_\circ t}$$

Furthermore, the FFT only takes into account the raw data that composes a signal by crafting the frequency spectrum of a full slice at a time. In order to maintain the importance of time aspect of a signal, the Short Time Fourier Transform (STFT) was developed, which is the technique implemented in this project. This uses a discrete FFT with a small sliding window that finds the frequency components that make up small sections of a signal at a time, and can be stitched back together into a representation that allows one to see what frequencies are present at any particular time in the signal.

Another important term to understand for this process is frequency resolution, or the distance between data points in Hz. The higher the frequency resolution, the more precisely one can recreate the signal and understand exactly what frequencies make up the sound at any given time. This ties in well with sampling rate, or the number of samples per second when a signal is recorded, which simply creates more data points per given time period. The higher the sampling rate, the more that a digital representation of a sound will

be able to resemble its analog state. Additionally, high sampling rate leads to a high time resolution as well, which is the amount precision in time between recorded data points. The quality of tracing with a lower and higher sampling rate can be seen below in Figure 2.

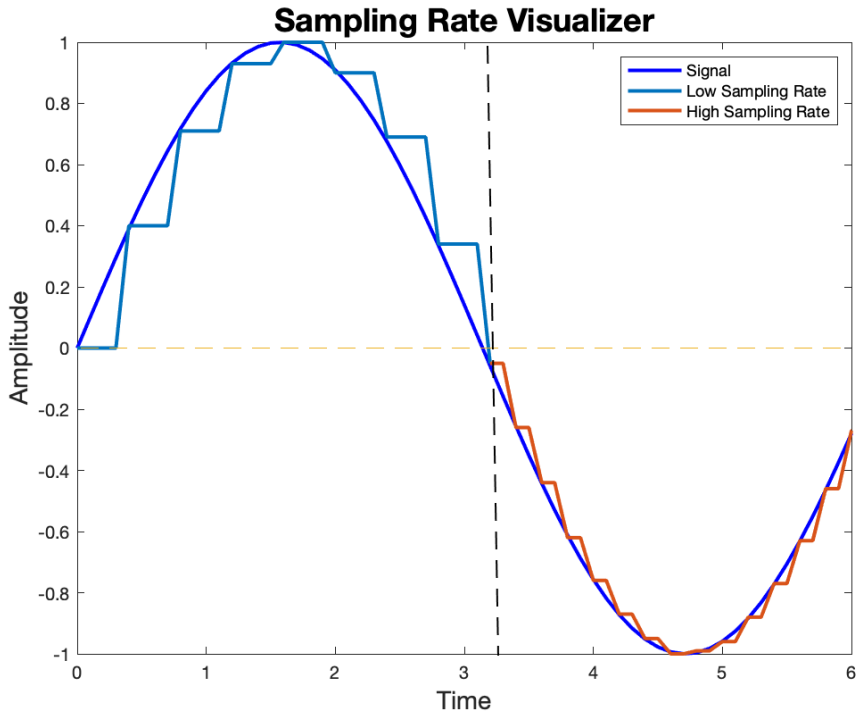

Figure 2: Depiction of analog to digital conversion with a relatively low (left) and high (right) sampling rate

Note that the quality of an analog to digital conversion can be found in reference to error. In Figure 2, the black dotted line divides low and high relative sampling rates on the left and right respectively. The low sampling rate lags for a long time before readjusting the notes, and thus has a lot more white space between the sampled amplitude and reality than the high sampling rate. This white space is the difference between analog and digital, and it can be viewed as error. The higher sampling rate on the right is a much closer representation of the original signal, and thus has less pitch quantization "white space" error.

Finally, it is important to be aware of the potential of aliasing in signal processing, which is the appearance of distortion or odd artifacts in any modified or reconstructed signal. This is often due to frequencies being recorded outside of an expected frequency range and not being accounted for in the filtering process. In an audio signal-processing problem such as this, it is commonplace to implement a low pass filter that helps to eradicate any audio that exists outside of the human audible spectrum (e.g. to nullify any frequency components recorded above 20,000 Hz), which is included in MATLAB's resample function.

2.4 Phase Vocoder Prior Work

This section will expand on how others have implemented phase vocoders in manners similar to this project. Since phase vocoders were invented in 1966 by J. L. Flanagan and R. M. Golden [7], the use of this process swelled into being omnipresent in the music industry through one medium in particular- Auto-Tune^{TM} [8]. A recent iteration of this pitch correction software can be viewed in Figure 3 below. This is a post-production process where after a musician (primarily singers, but the effect is often implemented on other single-tone producing instruments like saxophones) has recorded a take of their part with either slight mistakes or by being slightly out of key for certain sections of the song. Studio engineers are able to use the phase vocoder process through a digital audio interface to "round" the vocal recording to the nearest standard musical note, which will cause it to sound better and in key with the other instruments present. The program essentially shifts the pitch of whatever section has been selected to a desired center frequency, and modulates the signal as such without altering the duration of time that it would be played for. The notes that this process produced would generally be very discreet and distinguishable for even an untrained ear, which is what the blue boxes in Figure 3 represent.

Traditionally, this process was used for slight alterations- occasionally for pitch shifting up to a semitone (the smallest increment in pitch playable on most Western instruments, such as a fret on a guitar or an adjacent piano key), but generally would only alter vocals by a few cents (one one-hundredth of a semitone) to get the cleanest-sounding recording possible. The after-effect of this process being used with notes so close to their end-goal would be that the reproduced signal at the desired pitch would still sound very human, the transitions between notes would be fairly indistinguishable from unedited notes that a human sang, and they would aim to be transparent for the listener, imitating a "perfect" singer. However, in the past few decades, starting with the popularity of the song "Believe" by Cher, many artists have been opting to intensify the effect of the Auto-Tuning phase vocoder. This phenomenon entails music being recorded with Auto-Tune running live and "rounding" all notes to their nearest standard frequency, creating inhuman transition times and odd artifacts in sounds that give artists a sound they had never before been able to produce. It is also commonly used in the Rap and Hip Hop genres for quick transitions between speaking and singing, as well as for artists who are not formally trained in singing to get them closer to the desired note while still being very raw in their performance [9].

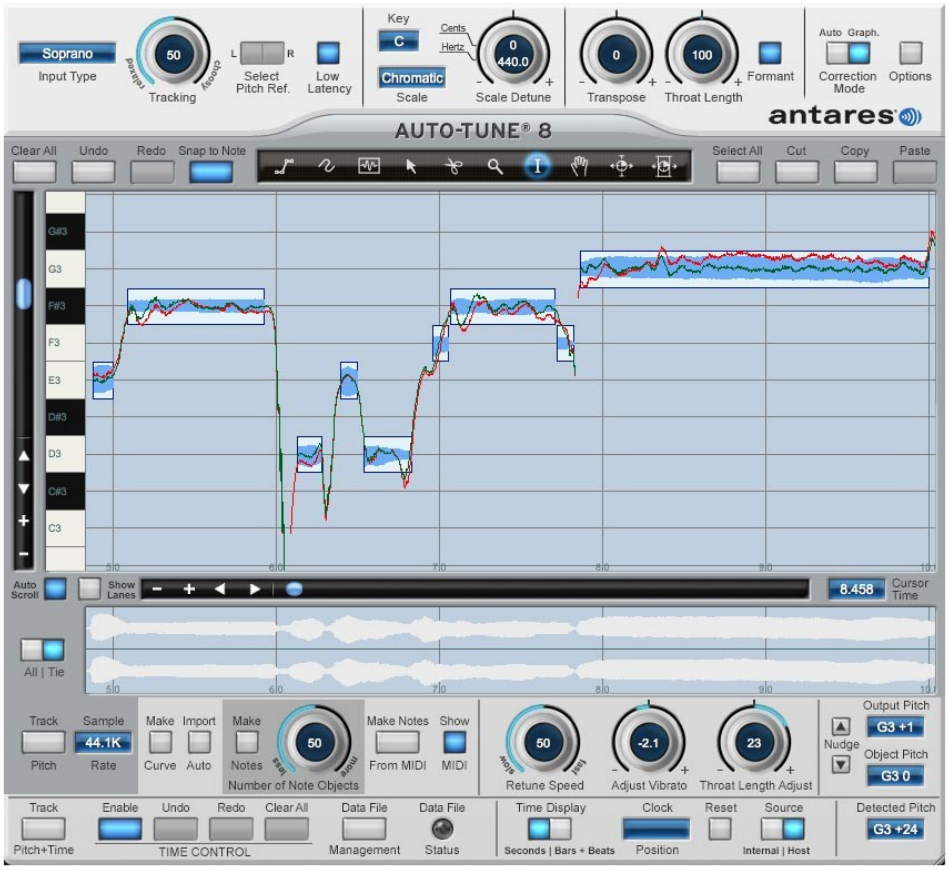

Figure 3: Auto-Tune Discrete Vocal Pitch Shifting [8]

As can be seen in Figure 3, in this program, the "sang" pitch of raw audio data is the orange line, and the green is the "snap to pitch" version where the audio slices are centered around the user-specified notes of the blue blocks. This allows the user to specify what they want human-like audio to be re-pitched to without making the note sound extremely auto-tuned. However, this interface is limited to function with post-processed audio only, and can not be implemented in real time.

The industry standard for pitch modulation is unable to perform the desired function, and so the next section will describe the audio processing method of a phase vocoder and how it could be implemented to allow for such functionality.

2.5 Conclusion

Now that a basic background on the motivation for the project, musical knowledge for what is trying to be achieved, and technical knowledge for audio and frequency domain signal processing have been established, the next chapter, "Methods", attempts to walk the reader through how the phase vocoder process is

accomplished.

3 Methods

This chapter will discuss the overall implementation of this phase vocoder [10] [11]. For the sake of coherence, only the MATLAB code will be discussed unless otherwise noted [12]. A high-level flow chart can be viewed below in Figure 4, and each block will be discussed in detail in the section noted in brackets.

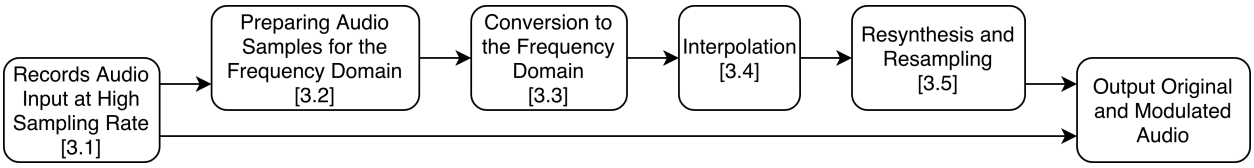

Figure 4: Flow Chart of Phase Vocoder Process

3.1 Audio Streaming

The most crucial part of any audio-modulating system is the input. Since this particular modulator is an attempt to emulate a "real-time" signal, or ideally with a delay below 20 milliseconds, the sampling in the beginning of the process seen above in Figure 4 needs to allow for such modulation. To enable a delay this long means that each audio slice needing to be modulated must be output within 20 milliseconds from when it was created, so as to leave time for the calculations, the duration of each slice would be set to about 10 milliseconds. As noted in the "Technical Foundation" section, at a sampling rate of 44,100 Hz, this results in 441 samples per slice. One audio slice can be viewed in Figure 5. In Python, these samples are read through a package called PyAudio, and MATLAB utilizes the "audiorecorder" functions.

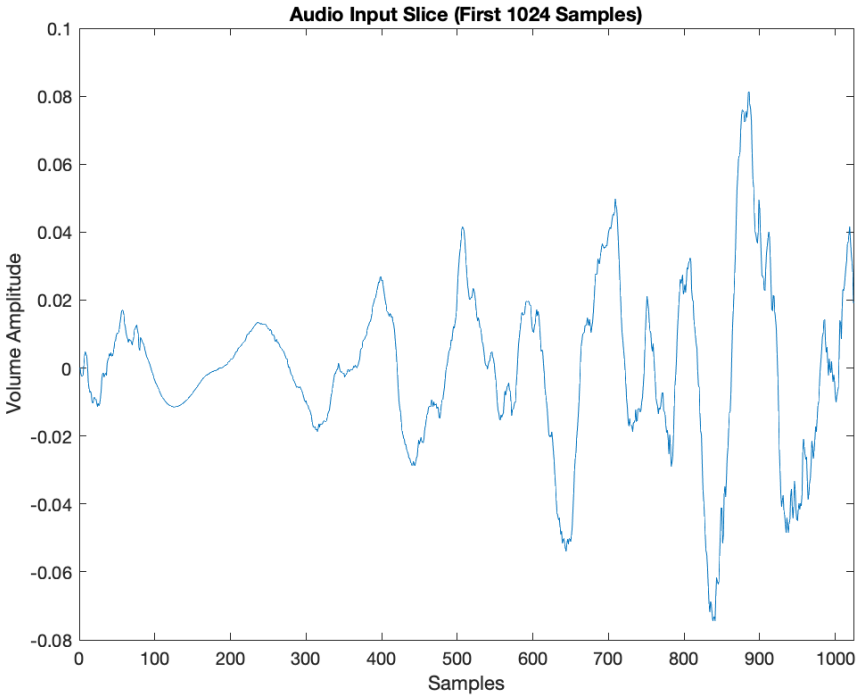

Figure 5: First 1024 samples of MATLAB audio input slice

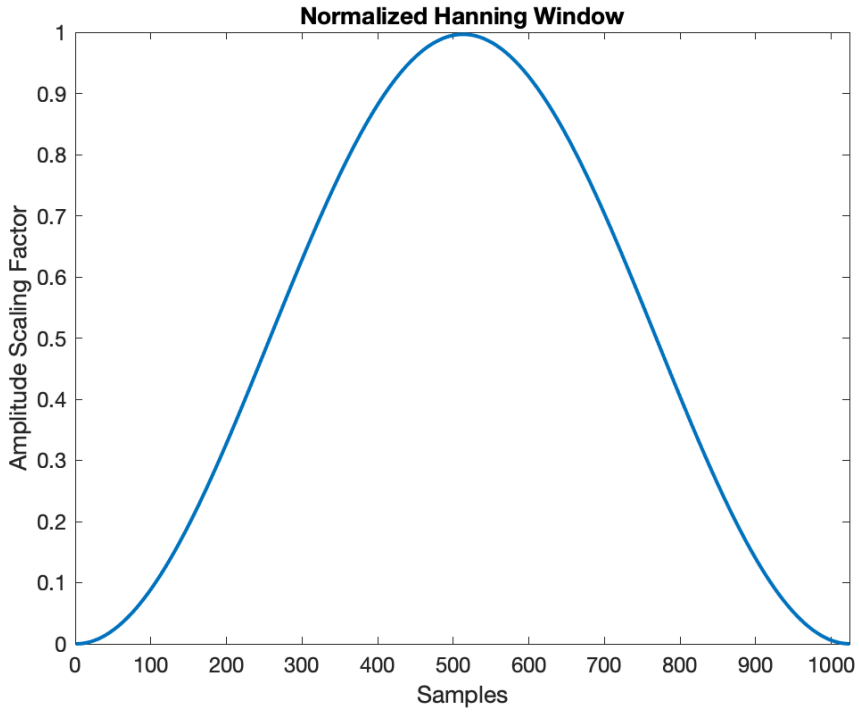

Figure 6: Hanning window on MATLAB

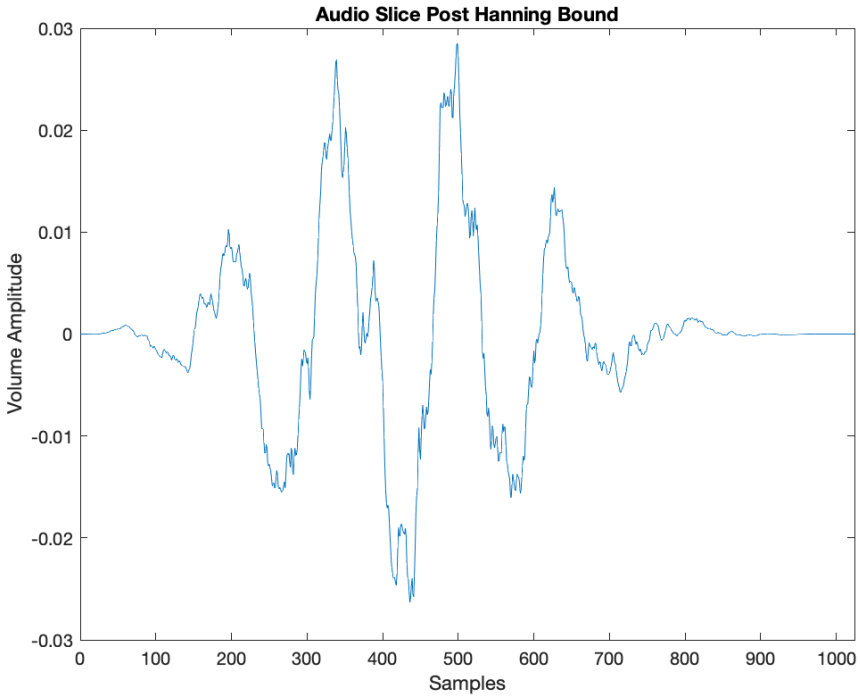

Figure 7: Scaled post Hanning windowed audio slice [14] [15]

3.2 Preparing for the Frequency Domain

The slice that has been stored is now converted to data that can be moved into the frequency domain, and other variables and reference points for further calculations are established here as well. This first entails converting all of the information into a vector of integers in Python and doubles in MATLAB. The discrepancy is a result of the methods used to read audio from microphones in MATLAB (as doubles) or in Python (as a byte-array that represents integer values).

As for definitions, the windowing size for the "Hanning Bound" was initialized, as is discussed next in the subsection "STFT to the Frequency Domain", to be as long as the number of samples, and be composed of a rectified cosine wave with two periods, each half of the samples of the slice in length. The other crucial variable is the hop size, which for the phase vocoder described in this thesis, can be defined as 1/4th of the number of samples in each slice. There will be an overlap size then of four, which means that any section of the reformed audio signals that will be discussed below are composed of the sum of up to four sub-slices at a time, which creates a smoother blurring sound and recreation in practice.

When pitch shifting up, the data stored in the new array will be played faster. Since one of the goals of resynthesis is to maintain the same time duration on these audio signals, the phase vocoder interpolates data in these higher pitch data streams using the overlapping windows, generating even more samples that

create a larger vector. Merging all of the interpolated and old samples together is why the higher amount of overlapping windows leads to a smoother reconstruction. An example of the effect of hop sizes on these sub-slices can be viewed in Figure 8. Here, frame size is the number of frames per sub-slice, hop size is the indexing point between adjacent sub-slices, and overlap size is the maximum number of potential sub-slices utilizing any data points for reconstruction. Note that the reconstruction of this array would likely be more choppy than this thesis' method, as it has an overlap size of two as opposed to four, creating a less blurred and averaged signal.

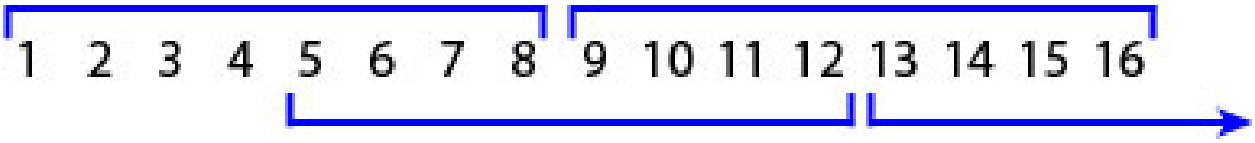

Figure 8: Hop size visualization with a frame size of eight, hop size of four, and overlap size of two [13]

3.3 STFT to the Frequency Domain

The Short-Time Fourier Transform (STFT) process translates the information from its temporal representation to its frequency representation, which allows for both the duration and pitch to be modulated. This is crucial in reproducing "musical", satisfactory signals.

This process begins with the Hanning Bound. Each audio slice is broken further into smaller segments that are multiplied by a common "window", as can be seen in Figure 6. Additionally, this window was used in the MATLAB code and was applied to longer recording slices (several thousand samples for several seconds of recording at a time), which is why it can contain as many samples as it does.

The combination of the audio slice in Figure 5 and Figure 6 can be seen in Figure 7. Note that the overall trend of this signal follows a Gaussian distribution, bell curve shape, where the "center" information has more value and weight than the edges of each segment, which will overlap with adjacent segments.

One should note that the window needs to be this shape for a number of reasons. First and foremost, it is rectified as to ensure that whatever it modulates will not flip signs, and this keeps the original data aligned in relationship with itself. Secondly, the sides of the Hanning Window force whatever it is modulating to zero, which can be viewed in the "Post-Hanning Bound" image. One can note that both ends of each sub-slice are normalized to zero, which helps eliminate sharp bursts of differentiation in sound data between adjacent windows. To understand the importance of this, for a signal in the time domain without this normalization, the transition between amplitudes of adjacent slices could be drastic, which would reproduce static or popping sounds when played back and the audio would be unrecognizable. If the window was a

rectangle, for example, that captured the full signal, it would have non-normalized sides where adjacent windowed samples would clash.

When each slice is multiplied and scaled by these weights, they are then Fourier transformed and summed back together with an appropriate overlap to reform a new signal. The output of the Fourier transform is a complex set of magnitudes and phases, representing the volume and pitch respectively. One section of the data at any given point in time is also known as a bin. Now that the temporal aspect of the recordings is maintained by interpolating the magnitudes of the complex numbers, the phase needs to be adjusted to create new pitches in the reformed signals.

3.4 Interpolation

To create data that represent higher frequencies in the frequency domain, one needs to interpolate the collection of windowed frames in the frequency domain so that there is more information when the signal is brought back into the time domain. This will allow for resampling for pitch shifting, which will be described in the following section.

First, we start with finding the magnitude of new datapoints. The amount of points interpolated is increased depending on how much higher the desired pitch is than the tonic. With small step sizes, occasionally the vectors used for interpolation will be accounted for twice, which is what allows for more information to be created. One by one, the magnitude differences between each adjacent windowed frame is calculated. The magnitudes of the frames are adjusted according to the step size to make reconstruction more reliable, as can be seen in this equation:

$$dmag = (1 - tf) * abs(dcols(:, 1)) + tf * (abs(dcols(:, 2)))$$

In this case, dmag is the difference in magnitude, tf is the difference in indexing values between two adjacent frames, and dcols is a matrix containing both the "current" and "previous" complex vectors. The first complex samples can be seen in Figure 9 here.

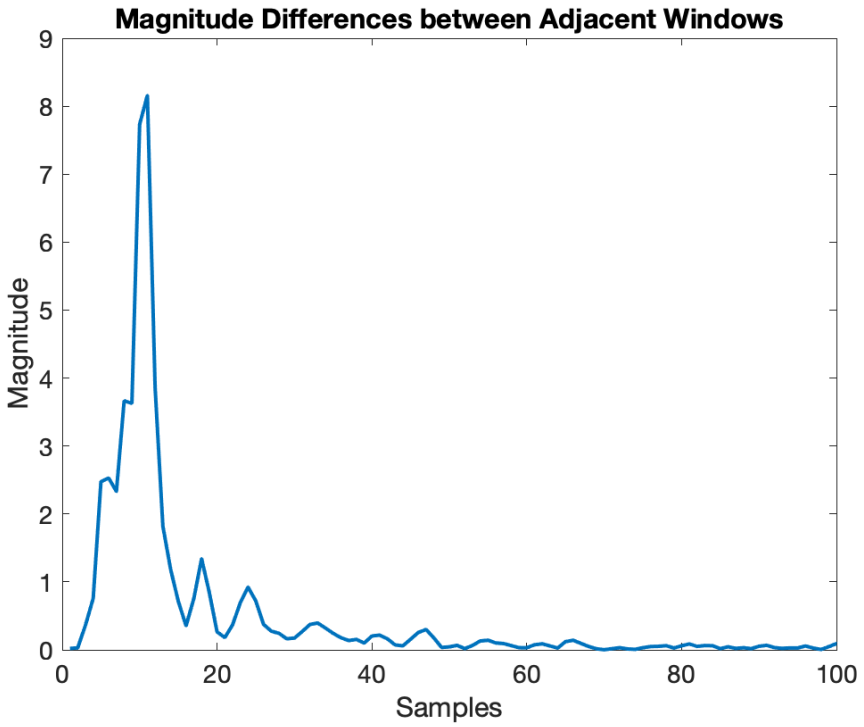

Figure 9: Magnitude difference between adjacent Hanning windowed frames

Meanwhile, to find the true phase of an audio sample, one subtracts the phase associated with the "previous" windowed reading of the sample from the "current" windowed reading of the sample. The phases of each bin is calculated using the "angle" function in MATLAB. The equation for the difference in phase can be shown here:

$$dp = angle(dcols(:, 2)) - angle(dcols(:, 1)) - dphi'$$

In this equation, dp is the difference in phase, with the first term representing the "current" frame, the second representing the "previous" frame, and dphi' being the expected difference in phase between adjacent frames. Note that dphi is a vector increasing at a constant rate, where the rate is dependent on the ratio of the desired pitch to that of the tonic. These phase differences can be seen here, in Figure 10.

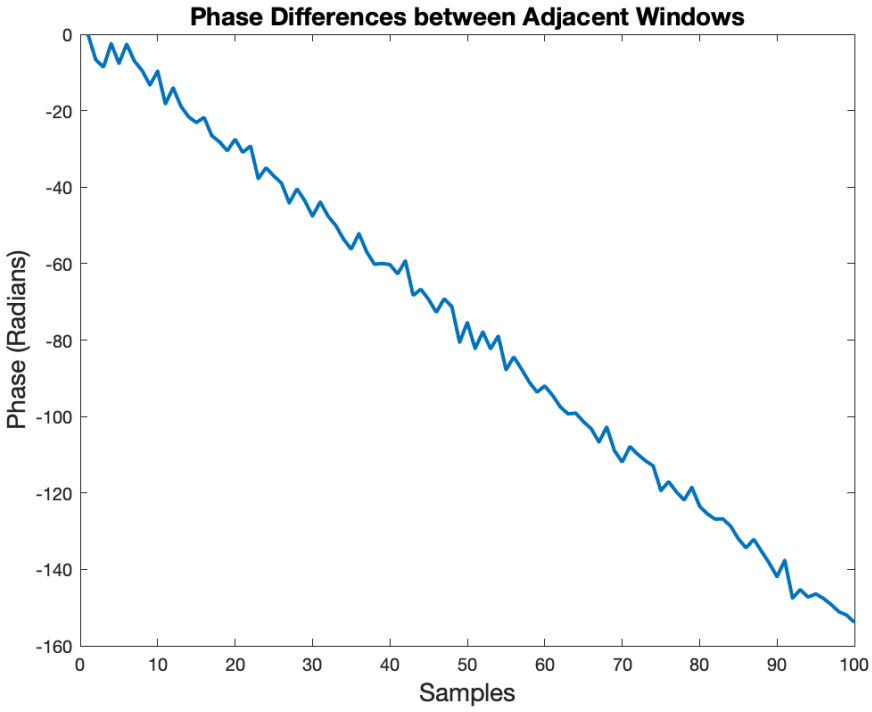

Figure 10: Phase differences between adjacent Hanning windowed frames

Note the sharp decline, which is the result of subtracting the expected difference in phase between adjacent bins from the actual difference. The jagged components within the broader decline contain the core information with respect to the expected information, and allow for more reliable reconstruction in the interpolated vectors. These differences are normalized to lie between $\pm\pi$. Adjusting the phase information allows a phase vocoder to play back any audio sample at the same pitch at any tempo. The scaled phases can be seen in Figure 11.

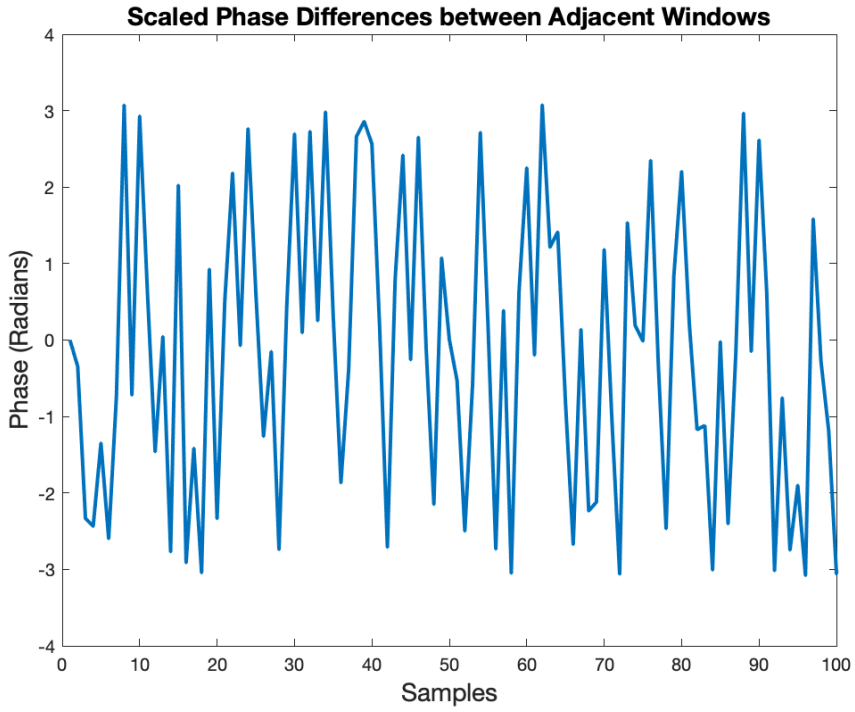

Figure 11: Phase differences between adjacent Hanning windowed frames scaled between $\pm\pi$

As opposed to the result described earlier, this phase vocoder's goal is the opposite- to maintain tempo, but modulate pitch. However, both results are accomplished with the same method. The magnitude differences and phase differences are recombined into a new interpolated matrix, which can be seen here:

$$c(:, ocol) = dmag. * exp(j * ph)$$

In this equation, c is the interpolated matrix and ocol is the current column being appended to in a larger for loop that iterates through each adjacent combination of frames, with the rest of the variables are the same as described previously. The c matrix now contains extra columns representing interpolated frames. When the columns of this matrix go through the IFFT process that will be described in the next section, the new time domain vector will be longer than the original, scaled to a length representative of the desired output tone. The vectors of magnitudes and phases have now interpolated and expanded to a length corresponding with the ratio of the desired pitch, as can be seen in Table 1. For example, to pitch shift up a major third, the length of the new expanded magnitude vector would be:

$$Length(Major3^{\text{rd}}Vector) = Length(MagnitudeVector) * 5/4$$

After the audio vector has been increased to fit its new desired length, the phase information is adjusted so that when the information is resampled and played out of the speaker, it will play at the new desired frequency.

3.5 Resynthesis and Resampling

Resynthesis allows the magnitudes and phases to be compiled back into integers, and resampling is the final step that shapes the integer vectors back into a size and representation that can be played through a speaker at the desired pitch.

Next, the newly calculated "true" phases for the resynthesized audio samples are assigned to their respective bins, and this new dataset is brought back into the time domain using the Inverse Fast Fourier Transform (IFFT). Here, the complex numbers representing the magnitude and recalculated phases of each bin are recombined into vectors of integers that can be rescaled and played out loud.

Finally, the audio is ready to be converted back into a playable vector. Using the resampling process, the magnitudes stay relatively the same, meaning that the volume maintains consistency between the altered and raw recordings. Resampling treats the input signal essentially as an analog signal in a sampling process similar to that used to record the initial audio. The primary difference is that the sampling rate used in the resample function is a ratio relative to both the desired new tone ratio and the original sampling rate. To continue with the example of a Major 3^{rd}, it can be noted that the length of the vector holding that information is still:

$$Length(Major3^{\text{rd}}Vector) = Length(MagnitudeVector) * 5/4$$

To bring this back to just the length of the magnitude vector so that it can be played out loud at the original sampling rate (44,100 Hz in this case), the resample sampling rate is:

$$Major3^{\text{rd}}ResamplingRate = (OriginalSamplingRate) * 4/5$$

Bringing these two together results in an output vector containing integers representing the pitch-modulated, time-consistent signal at a Major 3^{rd} higher frequency than the input. This is done by sampling the Major 3^{rd} vector at the new sampling rate, creating a vector of the original audio's length, yet still maintaining the pitch and phase information of a Major 3^{rd}.

One should also note that MATLAB's resample function by default applies an anti-aliasing low-pass filter, which enhances the quality of the reconstructed signal by reducing the distortion and potential appearance

of artifacts. Additionally, one can note that this particular phase vocoder only pitch shifts audio samples up, so it strictly compresses the post IFFT data during the resampling process. Expansion, however, is possible as well with the help of more data interpolation, and this would lower the pitch of the original sample while still maintaining the same duration in time.

The author has implemented this process in a MATLAB code. The next section will discuss the results of how the phase vocoder method fairs in terms of pitch modulation, as well as providing an analysis method for audio delay.

3.6 Conclusion

Now that the reader has a proper understanding of the phase vocoder process, the next chapter, "Results", discusses quantifiable results of the effectiveness of the implementation of this phase vocoder method for pitch modulation, as well as the methods used to calculate them.

4 Results

Two tests have been devised to prove the success of the attempt of a real-time phase vocoder. First, there is the pitch-based aspect, which will be addressed in the "Signal Reconstruction Verification" test, and second is the time-delay aspect, which is addressed in the "Audio Delay Verification" test.

4.1 Signal Reconstruction Verification

To verify that all 16 chord types are modulated correctly, an overlay was created to compare the expected frequencies in a given chord against actual modulated frequencies in a spectrogram [17], as can be seen below in Figure 13.

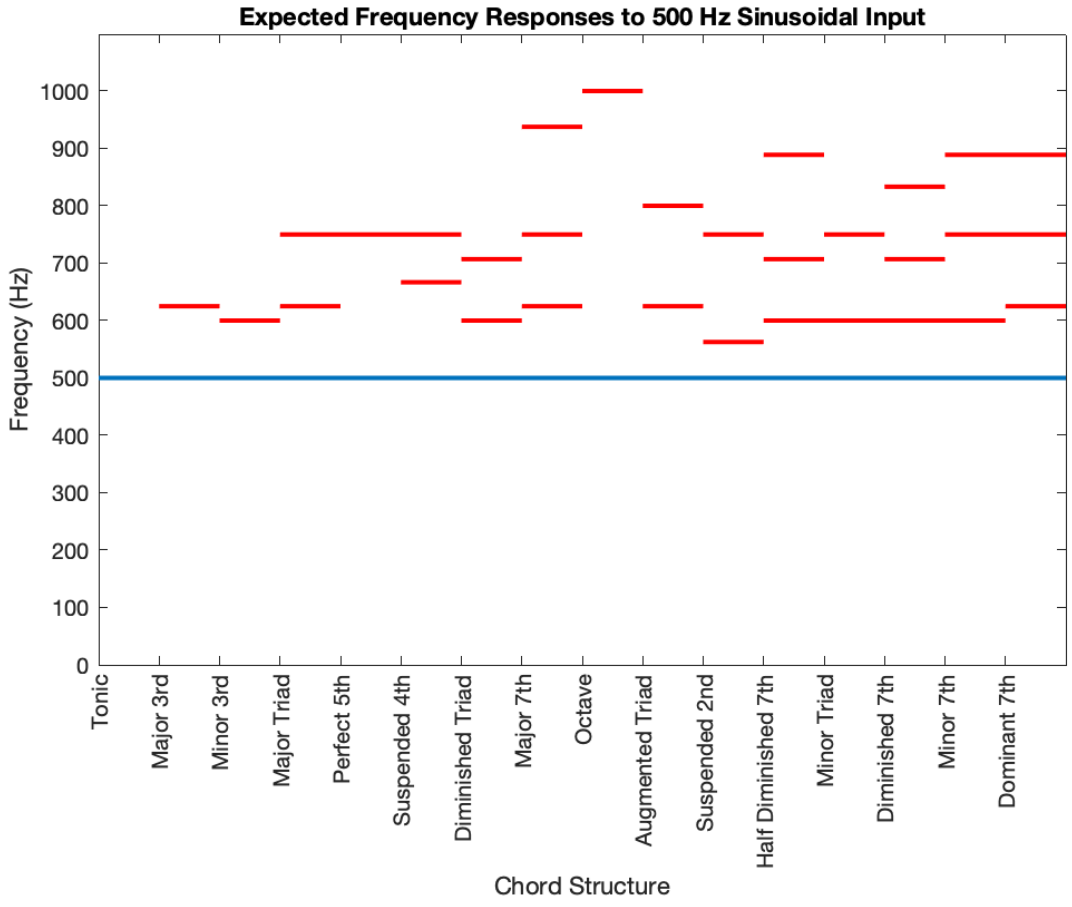

Figure 12: Overlay representing all 16 chords' expected spectrogram responses to 500 Hz sinusoidal input

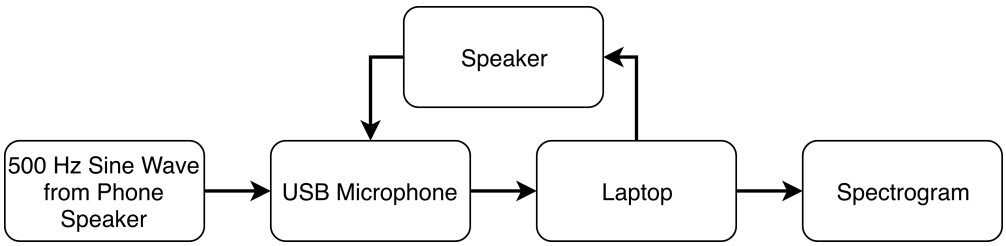

Figure 13: Flow chart depicting the setup used for the spectrogram test

First and foremost, the test was completed in an apartment during the daytime with as little ambient noise as possible. The microphone in use was a Blue Snowball, and the input is a 500 Hz sinusoid output from an iPhone speaker. Though 500 Hz does not perfectly align with any standard musical note, the value was selected as it is easy to consider and modulate mentally to expected frequencies. 500 Hz is a non-ear piercing yet audible frequency where the highest expected modulation, an octave above at 1,000 Hz, is within that same definition. The clarity of the signal can be viewed in Figure 14 below.

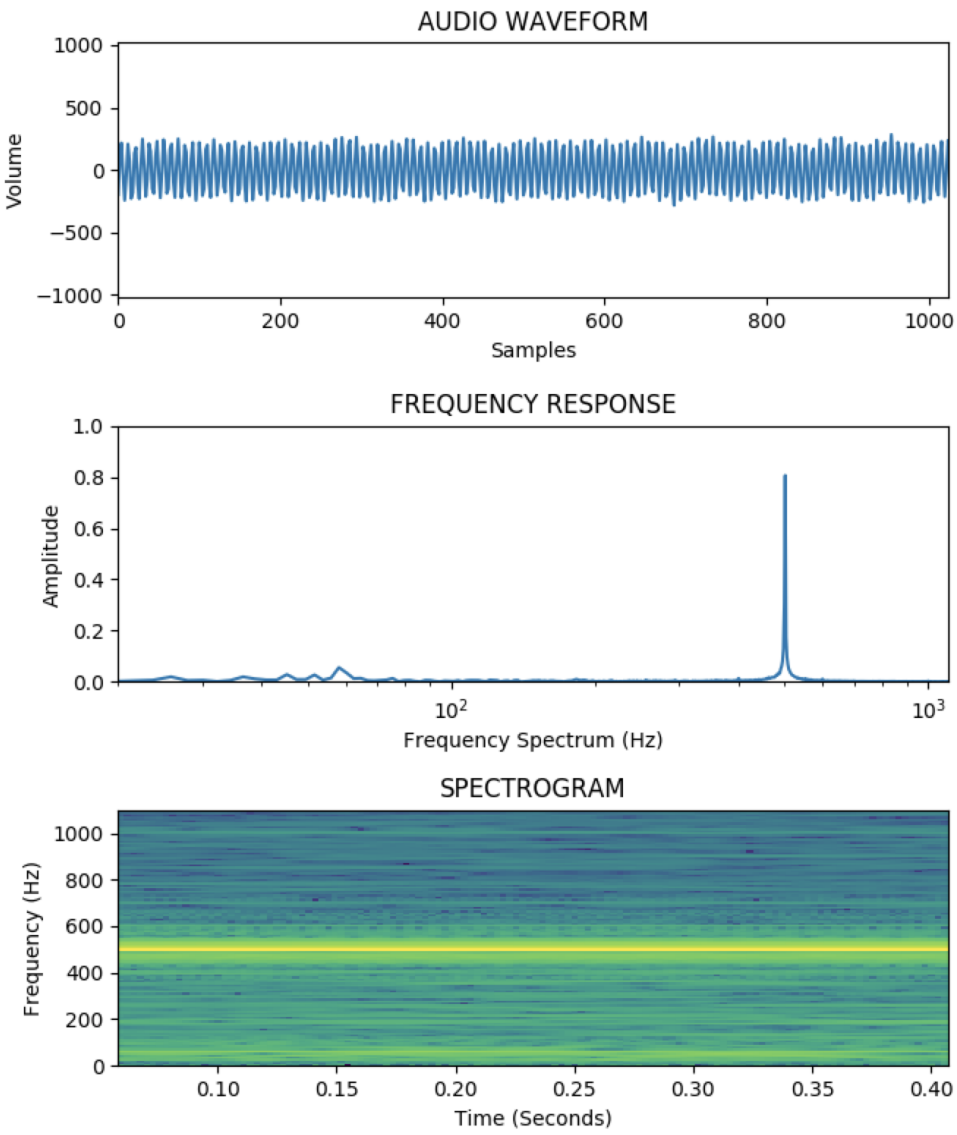

Figure 14: Time domain, frequency domain, and spectrogram of 500 Hz sinusoidal input

This image depicts three representations of the sinusoidal input- the topmost is its standard, temporal domain representation where one can note that the signal is fairly periodic and "clean", as it is simply a sine wave with very little background noise. The center plot depicts the frequency spectrum, where one can see the relative of all of the frequencies available for a specific tone. One can note here that the single, very sharp peak at 500 Hz is exactly what is expected from a clean sine wave. Finally, the bottom plot is a real-time spectrogram implemented in Python. This plots the frequency components over time, with frequency density on an increasing gradient from blue to green to yellow.

Then, each of the 16 selected chord types are modulated and output into the microphone through the

MATLAB phase vocoder code. Their generated spectrograms were recorded, and a small slice of each was put under the overlay. The spectral representation of each chord type can be viewed below in Figure 15.

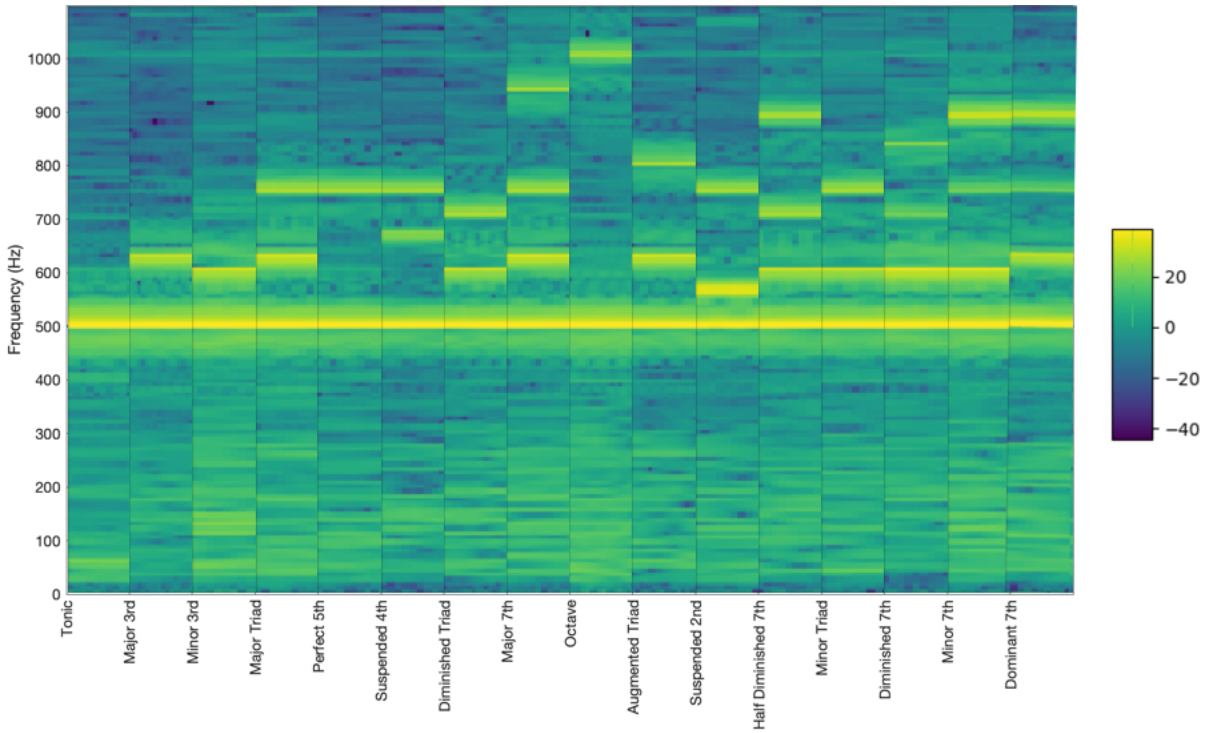

Figure 15: Spectrogram of all 16 resynthesized chords

Note that the color-mapping seen in the legend on the right is in dB. Additionally, this test only plots up to 1,100 Hz to make it easier for a reader to see the plot. This is acceptable, as the highest frequency tested to be generated is an octave, or twice the original frequency. For a 500 Hz signal, this equates to a maximum non-harmonic recreation of 1,000 Hz, so every possible combination should be visible under the overlay. The spectrogram with the addition of the overlay can be viewed here in Figure 16.

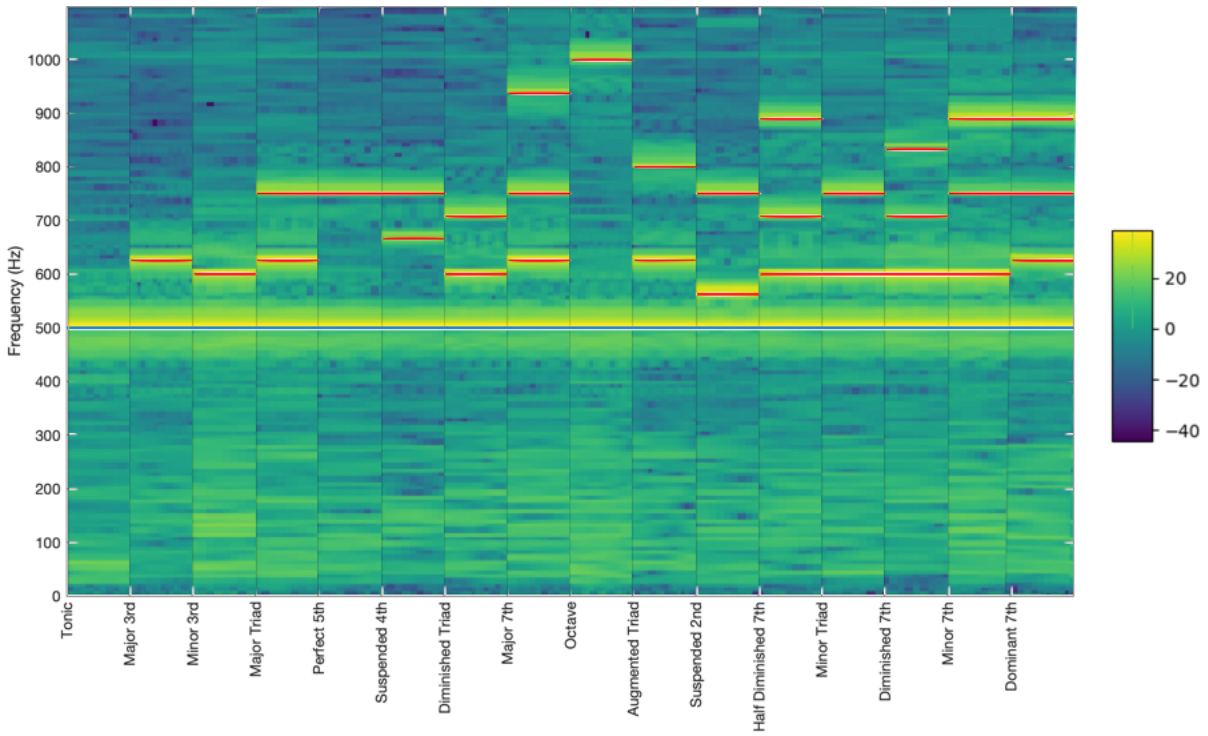

Figure 16: Spectrogram of all 16 chords with overlay

Visual inspection confirms that all of the tones produced through this phase vocoder directly align with the expected frequencies. There are very small alignment issues with the diminished fifth note, as is visible in the Diminished Triad, Half Diminished 7th, and Diminished 7th chords. Since the relationship of that particular note to the tonic is $\sqrt{2} : 1$, as MATLAB's resample function only accepts integers as its input, the ration of 10,000 : 7,071 was used as a substitute.

One interesting thing to note is that the spectrogram images contain a faint trace of the second harmonic at 1,000 Hz, double the fundamental frequency. This is expected, as frequencies repeat in increasing multiples of themselves in the frequency domain until infinity with rapidly decreasing power. This means that there would also be a consistent existence of frequency components at about 1,500 Hz, 2,000 Hz, and so on. However, the power drop-off is so significant that it is unlikely to be distinguishable from the background noise, in this case, past the second harmonic of 1,000 Hz.

4.2 Audio Delay Verification

The following test procedure depicts how one could verify audio delay. Unfortunately, as will be mentioned in the "Shortcomings and Solutions" section, the engineer was unable to combine a real-time implementation with the phase modulation code.

Using a DAW like Logic Pro X, the tempo would be set to 60 BPM to ease calculations, as this equates to 1 second per "beat". Audio would be recorded into said program, and the "Measure Markers" could be used to calculate how far apart these sounds are. For example, two snaps were recorded in Figure 17.

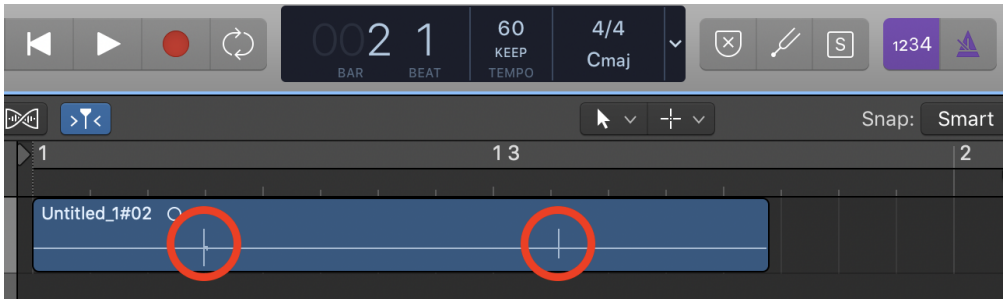

Figure 17: Two snaps in Logic Pro X circled in red

Zooming in on these ticks, as can be seen in Figure 18, reveals that they started at time instances "1 — 1 — 4 — 3" and "1 — 3 — 2 — 38.5" respectively (these time instances are read in 4/4 time signature as "Measure — Beat — Quarter Beat — 240th of a Quarter Beat".

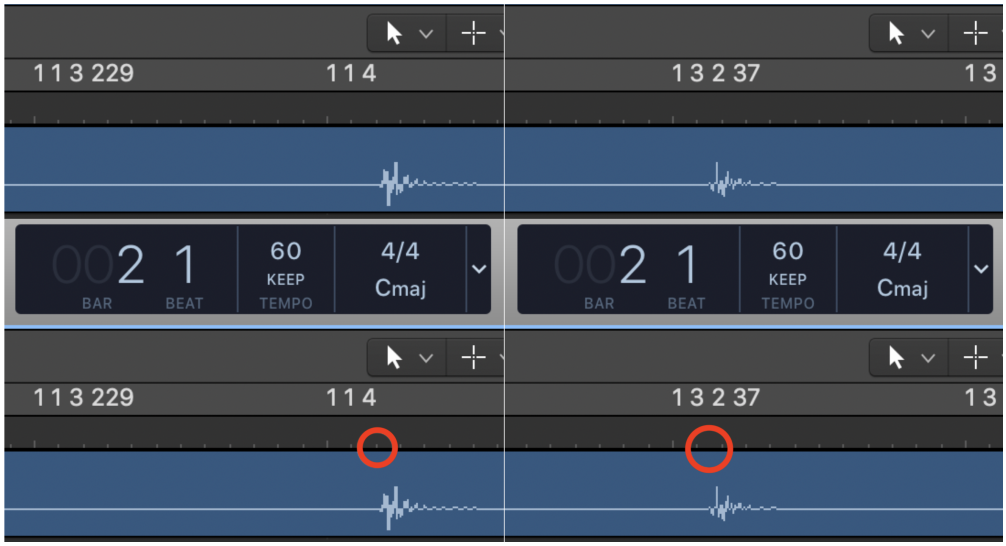

Figure 18: Zoom on measurement markers in Logic Pro X (occurrence "tick marks" circled in red)

One can note that these snaps occurred at what seems like a discrete time in the unzoomed Figure 17, and is still very short in the zoomed Figure 18, and the benefit of this sound is that there is virtually no attack or delay to a snap - the sound occurs with no swelling or residual sound on either side of the climax volume they produce. Additionally, the 240^{th} of a quarter beat is an industry standard for the level of precision that MIDI devices are able to record at, which is why Logic Pro X caps out at this level of accuracy [16]. This gives us measurable steps in time of approximately 4 milliseconds per tick, or five times the accuracy

needed to detect a real-time delay in an audio signal.

From the calculations done in Table 3, it can be calculated that at 60 BPM, these sounds are at times 0.75208 and 2.28906 seconds respectively, placing them 1.53698 seconds apart. It is important to note that this program, and many like this, can quantify timings in even smaller intervals than this, which is more than sufficient for verifying a delay of less than 20 milliseconds. This is due to the program recording at a very high sampling rate and storing it as such, where it is able to differentiate extremely precise differences in sound, much more precisely than a human can.

	Measure	Beat	Quarter Beat	240th of a Quarter Beat	Total Time
Snap 1	1	1	4	3	
Snap 1 (Seconds)	0	0	3/4 * 1	2/240 * 1/4 * 1	0.75208 seconds
Snap 2	1	3	2	38.5	
Snap 2 (Seconds)	0	2	1/4 * 1	37.5/240 * 1/4 * 1	2.28906 seconds

Table 3: Time delay measurement calculation table

Although this test was unable to be performed on a real-time modulation code, its use will be applicable when an appropriate code is implemented. In the next section, this thesis will contain a discussion of the shortcomings of expectations with audio delay and why they failed to be reached.

4.3 Conclusion

Now that the reader has an understanding of the results of the implementation of the phase vocoder method created in this thesis, the next chapter, “Conclusions and Future Work”, will describe the contributions this thesis has made to the field, shortcomings, plans for bettering the implementation in the future, and acknowledgements.

5 Conclusions and Future Work

Finally, this section will discuss the key contributions to the overarching subject and note where they are found in the thesis, the shortcomings of the work that has been accomplished during this thesis, and a personal statement by the author.

5.1 Contributions

This thesis was aimed to accomplish two major tasks within the field. First, it was an attempt to provide a more easily understood walk-through of the phase vocoder process through the real time implementation of a vocal chord selection.

Additionally, this thesis aimed to provide an understanding of how a real-time usage of this phase vocoder process could be implemented, as well as tested, and to discuss the importance of bringing this into the field of music technology.

5.2 Shortcomings and Solutions

One of the core goals of this thesis was to implement a phase vocoder that functions in real time. However, MATLAB's only implementation of reading audio data in real time was through an expensive package in its topographical modeling software, Simulink. Due to this limitation, an alternative route using Python was attempted to produce the real time effect. Python was selected in particular as it also allowed said code to be implemented onto a microcontroller such as a Raspberry Pi. Unfortunately, the author had no prior experience working in Python, and encountered a plethora of problems.

First and foremost, a code was developed that read audio in small time samples and played them back immediately as proof that the audio could be sampled in real time. However, there was consistent stuttering or blurring in the audio occurred depending on the sampling rate and number of samples per slice. This was primarily due to the idea that the code was being run on a single processor on a computer, and would only become worse when modulation code was implemented driving up the time between recording and outputting the signal. By only using a single processor at a time, to maintain real time ($\leq$ 20 milliseconds between playback of recorded audio), the program would have to alternate when it was recording and when it was playing. In other words, at the first time instance, it would record 10 milliseconds of audio. At the second time instance when the first had completed, it would play that back for 10 milliseconds, but would be unable to capture the audio existing during that 10 milliseconds duration. The code would then in the third time instance record another 10 milliseconds duration of audio, but the middle slice would be missing

from the listener's ear. If modulation was added into the code, the amount of time between a recording time instance and a playback instance would have to increase to leave room for calculations.

One particular method to counter this is the implementation of multiprocessing, or using multiple cores on a computer or microcontroller to segment the tasks required to run this code in parallel. For example, with a computer with 4-processors, the tasks could be divided amongst them as seen in Figure 19.

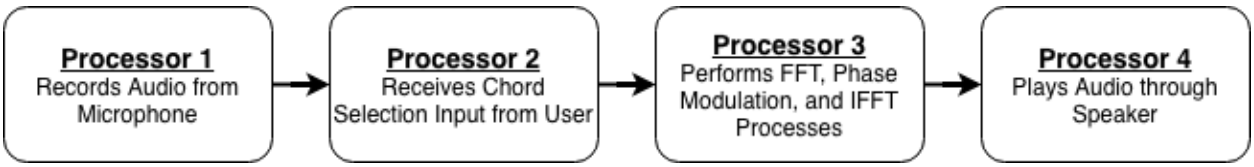

Figure 19: Potential quad-processor solution to the "stuttering" problem

Processor 1 would record small slices of audio (e.g. 10 milliseconds each) at a time, and stream that to Processor 2. This would retrieve a desired chord input selection from the user, either via input in the code or through a physical or GUI usage, as is addressed in the "Future Plans" section. Both of these streams of information are then placed into Processor 3, which would convert the audio information to the frequency domain, perform phase shifting to the desired pitches, and then convert the information back into playable audio. This would then be output through the Processor 4. The beauty of this system is that at any given time, these processors would be functioning in parallel. In other words, this would in theory allow for every time instance of an audio sample to be recorded, as Processor 1, which is recording audio, would ship its data off to the next core after each time step and be empty and ready for a new recording infinitely and instantaneously after the completion of the first.

5.3 Future Plans

One hope for this project after it is implemented into real time is to embody the code. For example, the author's Senior Experience in Engineering Design (SEED) team at the University of Vermont, composed of Ben Crystal, Margot Criscitiello, Haley Greenyer, and Elias Levinson, have developed a controller instrument that would allow vocal performers to select what chord they would be modulating to in real time. A SolidWorks schematic can be viewed in Figure 20.

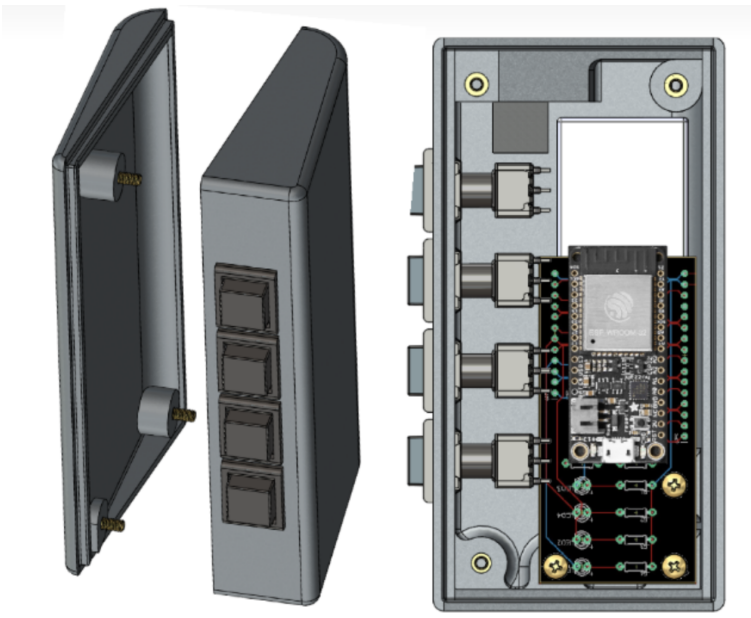

Figure 20: Controller for phase vocoder embodiment

The user interacts with this instrument through an array of four buttons. Holding down different combinations of the buttons allows for 16 potential outputs, which correlates with the chords visible in Table 1. While the user is singing into a microphone, their selection is sent via Bluetooth to a microcontroller connecting the microphone input to a speaker output. This microcontroller would ideally perform the phase vocoder process that this thesis described, and would output the fully developed chords to the speaker in real time.

It can be noted that the ergonomics of this device designated how the order of chords have been referenced throughout this document, such as in Table 2 and in Figures 13 through 16. Additionally, Python was chosen as the language to attempt the real time implementation in due to various constraints from the Senior Design project.

Another simpler potential implementation of this method is to install it as a computer-keyboard controlled Virtual Studio Technology (VST). This particular VST could be used to allow music producers to either quickly mark multiple ideas down on a track through singing, or as a permanent effect in a song to guarantee note alignment without needing to purchase an Auto-Tune software in a shorter period of time. Although this idea is primarily fixated on real-time processing for live applications, it is always beneficial to increase the efficiency of a music studio engineer's work flow. To have a virtual software that allows them to track multiple ideas at a time using nothing but, say, keyboard input could be an extremely beneficial time saver across just a few recording sessions.

5.4 Personal Statement

Music has been a fascination and passion of mine throughout my entire life. When I came to college to pursue engineering, I'd had an idea that mixing the two was possible but I had no idea how to initiate this blend. Working on this thesis in combination with my Senior Design project has opened up my eyes to a world of possibilities and potential projects for the future. It has been an honor and a privilege to be able to explore my hobby through my academic career path, and vice versa.

5.5 Acknowledgements

First and foremost, I would like to thank Professor Jeff Frolik for all of the guidance and patience he has provided, as well as for helping me stay motivated and on track throughout this process. I would also like to thank Professor Hamid Ossareh for enthusing all of my future plans with this thesis, as well as all of the help he as given me when overcoming obstacles. I would like to thank my SEED team mates for keeping me motivated to push the project through to completion, and for giving me a reason to work as hard as I did. Finally, I'd like to thank my parents for for believing in my academic pursuit and supporting me no matter how overwhelmed I become.

Thank you for reading this thesis, and I hope you learned something useful.

6 Bibliography

References

[1] Canvas "Jacob Collier: WTF is a Harmoniser? — EFG London Jazz Festival Preview". November 19, 2015.
https://www.youtube.com/watch?v=DnpVAyPjxDA

[2] "The future of the keyboard", Roli, March 29, 2019.
https://roli.com/products/seaboard/

[3] "All Your Instruments in 1", Artiphon, March 29, 2019.
https://artiphon.com/

[4] "Why Music Math? Musical Ratio and Musical Proportion", Earlham College: A Feeling for Harmony, Book 1 Chapter 1M Ratios, Part 1, March 29, 2019.
http://legacy.earlham.edu/ tobeyfo/musictheory/Book1/FFH1$_C$H1/1$M_R$atios$C$ommas1.*html*

[5] Seng Piel, "Chord Name Finder by Note Entry", Tontechnik-Rechner - sengpielaudio, March 29, 2019.
http://www.sengpielaudio.com/calculator-notenames.htm

[6] Bryan H. Suits "Chords - Frequency Ratios", Physics of Music - Notes, 2018.
https://pages.mtu.edu/ suits/chords.html

[7] J.L. Flanagan, R.M. Golden, "Phase Vocoder", Bell System Technical Journal, Pages 1493-1509, 1966.
http://www.ee.columbia.edu/ dpwe/e6820/papers/FlanG66.pdf

[8] "Antares AutoTune Vocal Studio", studiocare, March 29, 2019.
https://www.studiocare.com/antares-autotune-vocal-studio-bundle-native-inc-autotune-and-avox.html

[9] Charlie Harding and Nate Sloan "Do You Believe in Life after Autotune?", Vox: Switched on Pop, Episode 102, January 22, 2019.
http://www.switchedonpop.com/102-do-you-believe-in-life-after-autotune/

[10] Richard Dudas and Cort Lippe, "The Phase Vocoder - Part I", Cycling74, November 2, 2006.
https://cycling74.com/tutorials/the-phase-vocoder-%E2%80%93-part-i

[11] François Grondin, "Algorithm", Guitar Pitch Shifter, Section 3: Algorithm, 2009-2019.
http://www.guitarpitchshifter.com/algorithm.html

[12] D. P. W. Ellis "A Phase Vocoder in Matlab". Columbia University, New York, NY, 2002.
http://www.ee.columbia.edu/ln/labrosa/matlab/pvoc/

[13] Jeffrey Hass "Synthesis by Analysis: Phase Vocoding", Indiana University Introduction to Computer Music: Volume One, Chapter 4, 2017-2018.
http://iub.edu/ emusic/etext/synthesis/chapter4$_p v.shtml$

[14] "The Hanning Window", Azima DLI, 2009.
http://www.azimadli.com/vibman/thehanningwindow.htm

[15] Kui Fu Chen, Shu Li Mei "Composite Interpolated Fast Fourier Transform With the Hanning Window", IEEE Transactions on Instrumentation and Measurement, Volume 59, Issue 6, Pages 1571-1579, 2010.
https://ieeexplore.ieee.org/document/5456142

[16] B. Graziano Degazio "The Transformation Engine", University of Michigan International Computer Music Organization, Volume 2004, 2004.
https://quod.lib.umich.edu/i/icmc/bbp2372.2004.088/1/–transformation-engine?page=root;size=100;view=text

[17] Mark Jay "Let's Build an Audio Spectrum Analyzer in Python!", Youtube, September 9, 2017.
https://www.youtube.com/watch?v=AShHJdSIxkY

[18] "UVM Logo Solid", March 29, 2019.
https://www.uvm.edu/sites/default/files/UVMLogoSolidBlack.png