Pitch Correction

of digital audio

by Walter Smuts

Prepared for Associate Professor Fred Nicolls
Department of Electrical Engineering
University of Cape Town
South Africa

2018 (Generated on September 7, 2025)

Abstract

This thesis investigates the design, implementation and testing of a pitch correction system in the context of digital audio. A modular design is proposed which contains a frequency detection module and a frequency scaling module. Testing metrics are designed to evaluate the performance of the system as a whole as well as individual modules separately. The algorithms implemented and tested for the frequency detection module are the zero-crossing method and autocorrelation method. For the frequency scaling module a simple overlap and add method was tested a swell as the phase vocoder approach. The combination that produces the best results is the phase vocoder and zero crossing method combination. This approach achieves a pitch improvement factor of 4.38 and a similarity of 44% to the original signal.

Declaration

I declare that:

1. I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.
2. I have used the IEEE convention for citation and referencing. Each contribution to, and quotation in, this report from the work(s) of other people has been attributed, and has been cited and referenced.
3. This report is my own work.
4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as his or her own work.

Name:	Walter Smuts
Date:	2018

[illegible]

Signature

CONTENTS

INTRODUCTION

Chapter 1

1.1 Problem Statement

Musicians are faced with the challenge of creating good music. This comes down to controlling many different qualities of the sound they produce. The tempo, rhythm, pitch and volume are some examples of these qualities. They often make mistakes under the pressure of live performances or even when making studio recordings. Pitch is specifically hard to control for vocalists or musicians who play instruments in the horn or fretless string families. Errors in pitch accuracy are known as intonation errors and effect both the harmonic and melodic structure of the music, degrading it's aesthetic appeal. Since all the efforts of the musician culminates in a sound wave, it is subject to analysis and manipulation by the tools, algorithms and effects developed in the field of signal processing.

Pitch correction is an effect applied to audio that attempts to fix errors in intonation. This involves minimally changing the waveform to have the desired pitch while retaining all the other qualities of the waveform. This effect is mostly discussed in the context of music, where pitch is considered an important quality, but some considerations in other fields such as speech comprehension can be conceived.

Figure 1.1 shows an example of what is meant by errors in intonation. The graph is a pitch contour diagram, plotting the quality of pitch over time. One of the goals of the musician is to make the pitch contour follow the pitch indicated by music notation as closely as possible during the duration of the note. The red regions indicate errors in intonation, i.e. the times which the

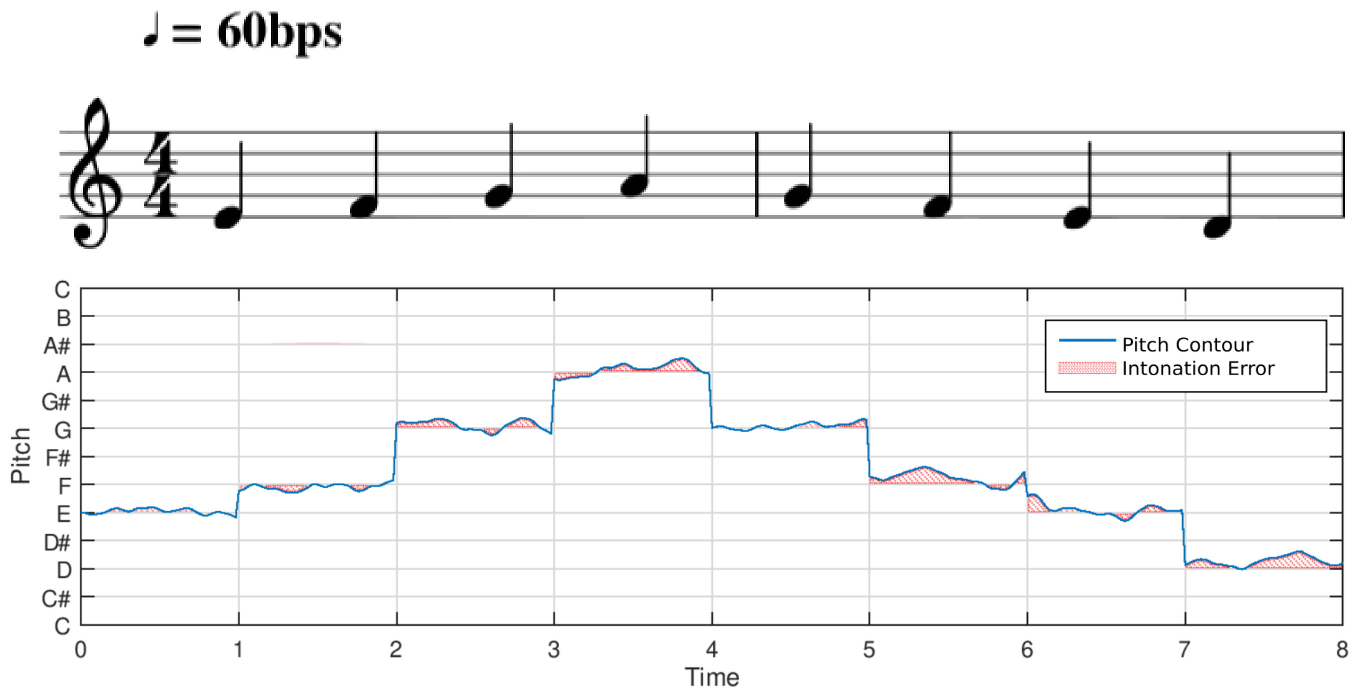


Figure 1.1: Illustration of Intonation Errors in a Pitch Contour Diagram

pitch contour does not follow the pitch indicated by the note. The goal of pitch correction is to alter the waveform to minimize these regions of intonation error while minimally affecting other properties of the waveform.

Pitch correction is difficult because it fundamentally require two non-trivial operations. These operations are pitch detection and pitch shifting. What's more is that these operations need to be efficiently implemented in order to allow for real time, minimal latency throughput for when pitch correction is needed for a live performance.

There are many different algorithms for pitch detection, each optimized for different situations. In the context of monophonic audio, the pitch detector needs to have robust performance in the presence of noise and prominent harmonics. The pitch shifter is fundamentally a harder problem to solve. This is explained by giving a naive example of how one could consider approaching pitch shifting. One could consider just playing the audio faster or slower, depending on the type of shift required. While this does successfully shift the pitch it alters some other qualities of the audio as well. The most important of which is duration, which also affects the tempo and rhythm. Another quality that this approach would alter is the formant structure of the sound. This is related to the harmonics of the sound and is explained later in the report. Altering this quality essentially comes down to the sound having an unnatural feeling to it.

The desire to have a pitch correction system can be summarised in a few use cases:

- A vocalist records some tracks at a recording studio. The whole song is perfect except for one out of tune note. This is only realised a few days later in the mixing phase of producing the song. Going back to the recording studio would be inconvenient and expensive. Having a tool to simply fix the intonation of this note, without the need to involve the vocalist or recording studio, would be strongly desired.

- A poor mumble rapper, call him Lil Pump, needs to deliver an important message. This message comes in the form of his latest song, sung to his followers in a live performance. Unfortunately he's not very talented and cannot sing in tune. A live pitch correction system, that would make his voice sound tolerable, would help him deliver his important message.

- A computer gaming company wants to create a game with an evil artificial intelligence antagonist. This character needs to have an eerily "perfect" voice. A system that can apply an effect on a human voice to sound perfectly in tune may create the desired effect.

1.2 History of Audio Pitch Correction

One of the first occurrences of pitch manipulation in music, at least the first occurrence that was found by the author, was of a song[1] called "The Chipmunk Song" from the animated music group "Alvin and the Chipmunks". The goal was to raise the pitch of the voice of a singer to sound an octave higher than his actual voice. This was accomplished by recording the song at half the wanted tempo and playing back the recording at twice the speed when mixing. The technology used in audio recordings was still analog tape, and the whole process was done using these tapes. The effect of speeding up a recording to achieve a pitch shift became known as the "Chipmunk Effect". This approach earned the group two Grammy awards[2] in 1958.

[1]The Chipmunk Song: `https://www.youtube.com/watch?v=b3p7tZw6Mps`

[2]`https://www.grammy.com/grammys/artists/ross-bagdasarian-sr`

In 1977 Eventide introduced a new product, the Eventide H949 Harmonizer[3], capable of incremental pitch shifting. This was sold as a "de-glitch pitch shifter" and was intended to be used to fix intonation errors and add harmonization effects after recording. Other features it was commonly used for was to stretch the duration of radio advertisements recordings, to be an exact duration, without affecting the pitch. The pitch shifting was implemented using single side-band modulation techniques and was done digitally.

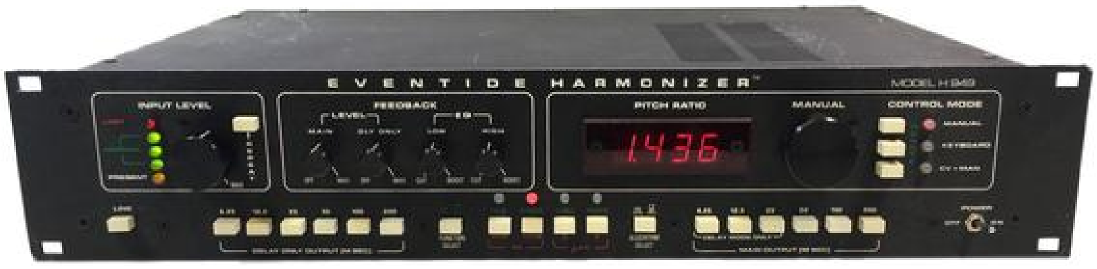


Figure 1.2: Eventide H949 Harmonizer Rack Mounted Unit

In 1996 Andy Hildebrand, an electrical engineer, was investigating seismic data, when he realised that the same techniques he was using to investigate the data, could be used to alter the pitch of audio files. His techniques for detecting pitch, using a simplification of the autocorrelation function, was considered superior to the state of the art at that time[4]. He implemented the first version of his pitch correcting algorithm on his Macintosh computer. His first demonstration was considered a success and the company "Antares Audio" was founded. A patent was filed in 1997 for a "Pitch detection and intonation correction apparatus and method". They created a product called "AutoTune" which was popularised in 1998 by Cher in her song "Believe" which made heavy use of the AutoTune pitch correction effect. AutoTune was the first product capable of automatic real time pitch correction[1].

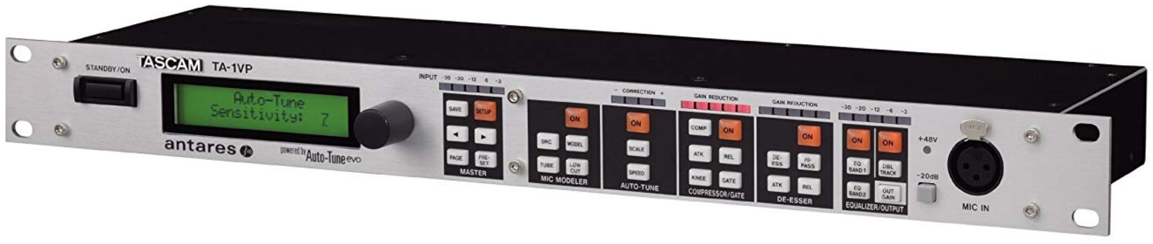


Figure 1.3: AutoTune Rack Mounted Unit

AutoTune comes in a rack mounted unit for live performances shown in figure 1.3 or as a VST plugin. The software has evolved to allow for many more features than the original pitch correction effect that the name suggests. Modern AutoTune is still considered a state of the art product by audio engineers.

In 2009 Tom Baran, an electrical engineer, wrote an open source pitch corrector. This pitch corrector was written in the C programming language as a VST (Virtual Studio Technology) plugin. At least two ports have been written and are up on GitHub. The fact that this project is open source allows for inspection of the code and provides insight in what design choices has been made. It uses an autocorrelation approach to find the pitch, a PSOLA algorithm to shift the pitch and a cubic spline interpolation algorithm for re-sampling[2].

[3]`https://www.eventideaudio.com/products/clockworks-legacy/harmonizer/h949-harmonizer`
[4]`https://priceonomics.com/the-inventor-of-auto-tune`

Finally Smule, a mobile phone application, was released in 2013. It's an application for amateur singers to record themselves singing songs and sharing the recordings on social media. The application features a pitch correction effect to improve the quality of the recording before it is shared. This pitch correction is done based on knowing what the harmony of the current note should be since it uses a pre-recorded backing track. A patent was filled by Smule in 2011 for "Pitch-correction of vocal performance in accord with score-coded harmonies"[3].

1.3 Approach Taken

The report follows the follow ng structure, starting with the literature review chapter. The literature review starts with investigates music theory, defining the concepts required for pitch correction and fundamentally answering the question of what is considered correct. This is done by using the concept of a tuning system to create a list of correct frequencies. Next, the pitch correction structure as a whole is investigated and an idea of the required modules are formed. The important modules are then investigate further, finalizing the literature review.

Thereafter, the implementation chapter discusses how the project was put together. Assessment metrics are rigorously defined that can measure the success or failure of a given pitch correcting system. These metrics aim to measure the pitch accuracy improvement the system provides and the distortion caused by the system. Thereafter a modular design is proposed for the pitch corrector, shown in figure 1.4.

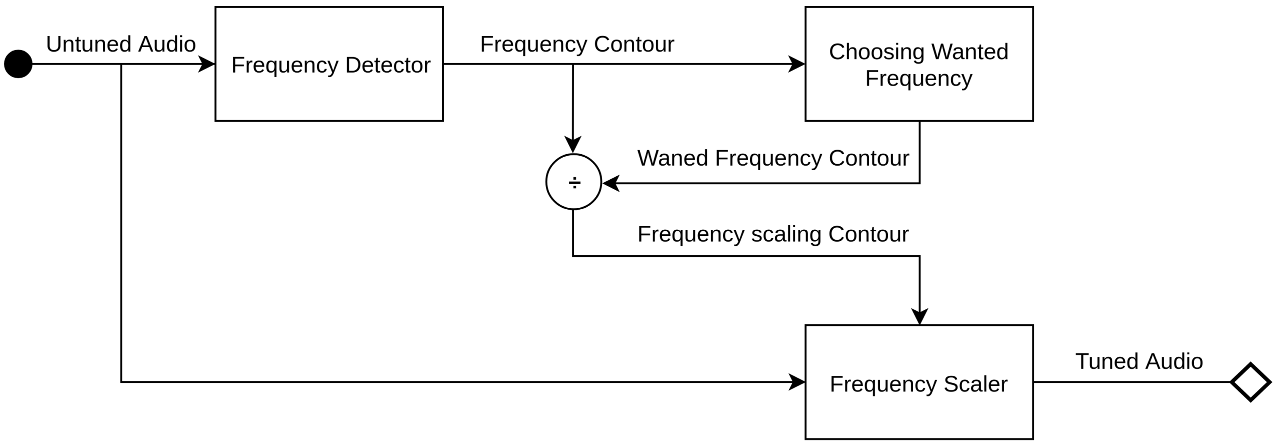


Figure 1.4: Flow Diagram of Pitch Corrector

Two implementations for the frequency detection modules are created and described. One based on the zero-crossing method and the other based on the autocorrelation approach. They were tested for noise robustness in the results chapter. The zero-crossing method required a signal to noise ratio of 4.5db while the autocorrelation method required a signal to noise ratio of 17.8db.

Next, two implementations for the frequency scaler were created and described. A simple overlap and add method; and a phase vocoder approach was implemented. Basic tests were run to compare the two approaches and roughly determine their pitch resolution. It was found that the phase vocoder approach had sufficient pitch resolution, acceptable for pitch correction, while the simple overlap and add approach did not.

The results chapter contains the results of the evaluation metrics defined in the implementation chapter. These rigorous metrics were only run on one combination of sub-modules, the zero-crossing method approach as the frequency detector and the phase vocoder as the frequency scaler. The metrics found that this combination produced a pitch accuracy improvement factor of 4.38 and a similarity of 44% to the original signal. A demo contour diagram is shown of how this system performs with a real vocal recording. Figure 1.5 shows the pitch contours of the demo.

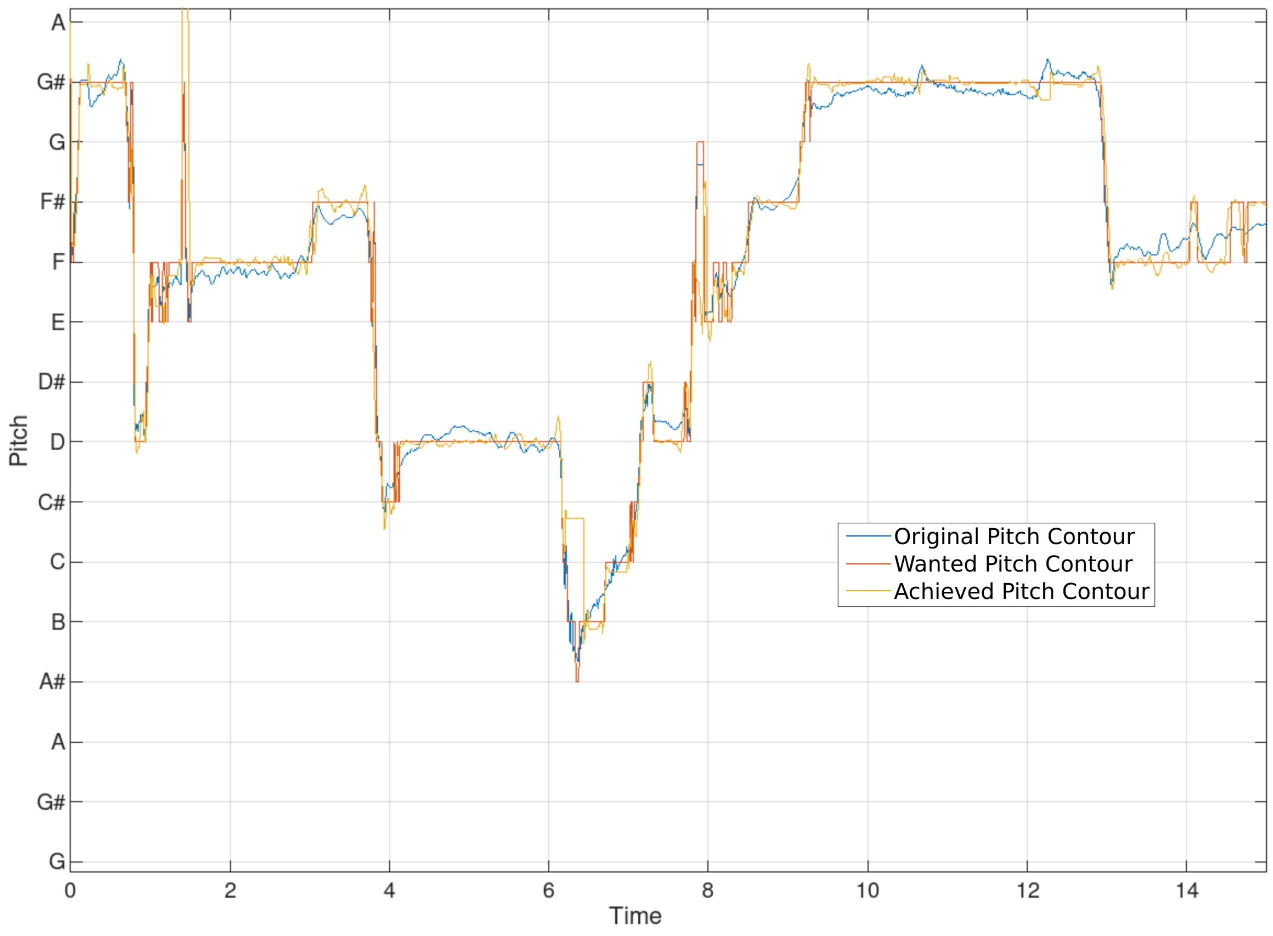


Figure 1.5: Vocal recording demonstration of pitch correction

The report concludes to say that the system, as it stand currently, is not capable of non-cherry picked data and further investigation and development is required. Recommendations are given on what such an investigation needs to consider and what to investigate first.

LITERATURE REVIEW

Chapter 2

This literature review attempts to cover all the subjects relevant to pitch correction. Music theory is investigated first, covering the topics of human pitch perception and tuning. This section attempts to determine what it means for a pitch to be correct and how this relates to frequency. The idea is to get a list of correct frequencies derived from a tuning system. After it is well-known what is musically considered to be a correct pitch, the general structure of how pitch correction is approached is investigated. This structure naturally splits up into modules and each module is investigated further.

2.1 Music Theory

The intent of this section is to come up with a definition of what will be considered a correct pitch. Some foundational work is required to define basic musical concepts in a rigorous way. To start we need to define exactly what is meant by a musical note.

A note is a sound made from a musical instrument. It has a pitch, volume, timbre and length. These are the characteristics a musician would consider when composing a piece of music. Each of these characteristics have a more rigorous scientific counterpart they are related to. Pitch relates to frequency; Volume relates to amplitude; timbre relates to harmonic content and length relates to duration. The characteristic pair relevant to pitch correction is the pitch and frequency pair and needs to be covered in more depth.

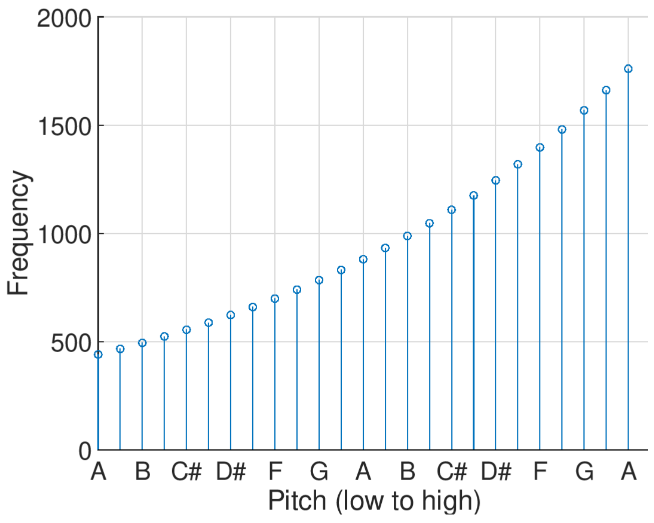


Figure 2.1: "Frequency vs Pitch"

Pitch and frequency are terms often used interchangeably but does not refer to the same concept. Pitch is a *sensation* experienced by humans when they hear notes containing frequencies between 31 Hz and 17.6 kHz[4]. The sensation of pitch is scaled on a subjective “high” and “low” scale. The higher the frequency, the higher the pitch perceived and the lower the frequency, the lower the pitch perceived. This relationship between pitch and frequency is generally considered to be logarithmic. Slight deviations from the expected logarithmic relationship was found[5] but was deemed minor and unnecessary to incorporate for this project.

In Figure 2.1 the relationship of frequency and pitch is shown. Pitch is generally denoted by letters ranging from A to G with sharps($\sharp$) and flats($\flat$) called accidentals. This is due to a long history of convention and is irrelevant for now. The main takeaway is that the perceived change in pitch is constant for each successive note. From this graph it can be seen that frequency is exponentially dependent on pitch, or inversely, pitch is logarithmically dependent on frequency.

From Figure 2.1 a list of correct frequencies is obtained. The method by which this list is obtained, originates from a concept called pitch harmony. Two notes sounds good or in tune if their frequencies produce a simple ratio[6]. For example an octave produces a ratio of 1:2 and a perfect 5th produces a ratio of 3:2. Figure 2.2 shows how these two harmonies are seen when their waveforms are plotted. The superposition created by the two notes creating the interval has a short repeating pattern. The fact that this pattern is short corresponds to the property of a simple ratio of frequencies. This short pattern/simple ratio is the origin of why two notes sound harmonious.

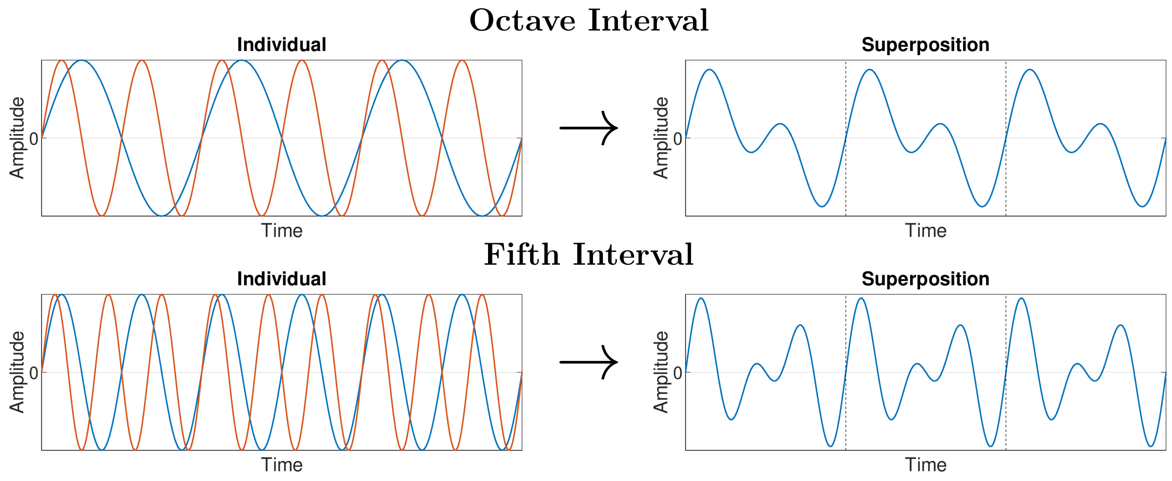


Figure 2.2: Harmonic interference of two intervals

This concept of harmony gives a simple method to start creating a list of frequencies that would sound good together. This list is created by starting with an arbitrary frequency (440 Hz) and using these simple harmonic ratio's to create a scale. In table 2.1 a list of these ratios and corresponding interval names are given. This is referred to as *just tuning*. This tuning method was naturally discovered because it corresponds to the different modes by which a string vibrates when plucked[7].

Table 2.1: Interval Names and Ratio's of Just Tuning

Interval Name	**Ratio**	Frequency relative to 440 Hz
Unison	$1/1 = 1.000$	440 Hz
Minor Second	$16/15 = 1.067$	469 Hz
Major Second	$9/8 = 1.125$	495 Hz
Minor Third	$6/5 = 1.200$	528 Hz
Major Third	$5/4 = 1.250$	550 Hz
Perfect Fourth	$4/3 = 1.333$	587 Hz
Tritone	$7/5 = 1.400$	616 Hz
Perfect Fifth	$3/2 = 1.500$	660 Hz
Minor Sixth	$8/5 = 1.600$	704 Hz
Major Sixth	$5/3 = 1.667$	733 Hz
Minor Seventh	$15/8 = 1.778$	782 Hz
Major Seventh	$15/8 = 1.875$	825 Hz
Octave	$2/1 = 2.000$	880 Hz

Unfortunately this just tuning method posed a problem when harpsichords and clavichords (piano-like instruments) were introduced. These instruments required to be in tune for any starting note. For a tuning system to allow this, a small interval is required that touches all the wanted intervals when each successive note is found using this small interval. Since pitch is logarithmically dependent on frequency, we require a single real number, r, raised to the power of a subset, n, of positive integers such that the result contains all the desired interval ratios. Equation 2.1 shows what is required in unambiguous notation.

$$\exists r \in \mathbb{R} \text{ s.t. } R \subseteq \{r^n | n \in \mathbb{N}_0\} \text{ where } R = \{\frac{1}{1}, \frac{16}{15}, \frac{9}{8}, \frac{6}{5}, \dots\} \tag{2.1}$$

This in not possible and is proved in Appendix A. The best one can do is choose which interval to keep perfect across all starting notes and choose a sufficiently small sub-division such that each of the wanted harmonic ratio's is close enough to an available note such that it is imperceptible to the listener. This is exactly what equal tempered tuning does. The interval kept constant is the octave and it is divided into 12 sub-intervals called semitones. Each semitone interval is equal to $\sqrt[12]{2}$. This interval has the property of when successively applied 12 times it produces an octave interval. It also very closely matches all the required just tuning ratios found in table 2.1.

Table 2.2: Interval Names and Ratio's of Equal Tempered Tuning

Interval Name	**Ratio**	Frequency relative to 440 Hz
Unison	$(\sqrt[12]{2})^0 = 1.000$	440 Hz
Minor Second	$(\sqrt[12]{2})^1 = 1.059$	466 Hz
Major Second	$(\sqrt[12]{2})^2 = 1.122$	494 Hz
Minor Third	$(\sqrt[12]{2})^3 = 1.189$	523 Hz
Major Third	$(\sqrt[12]{2})^4 = 1.259$	554 Hz
Perfect Fourth	$(\sqrt[12]{2})^5 = 1.334$	587 Hz
Tritone	$(\sqrt[12]{2})^6 = 1.414$	622 Hz
Perfect Fifth	$(\sqrt[12]{2})^7 = 1.498$	659 Hz
Minor Sixth	$(\sqrt[12]{2})^8 = 1.587$	598 Hz
Major Sixth	$(\sqrt[12]{2})^9 = 1.681$	740 Hz
Minor Seventh	$(\sqrt[12]{2})^{10} = 1.781$	784 Hz
Major Seventh	$(\sqrt[12]{2})^{11} = 1.887$	831 Hz
Octave	$(\sqrt[12]{2})^{12} = 2.000$	880 Hz

Table 2.2 shows the modern equal tempered tuning ratios and their related interval names. For all western scales a subset of these frequencies are taken to produce the desired scale. Figure 2.3 shows a scatter plot of the ratio's of just tuning and equal tempered tuning and a plot of the percentage error for each note. It can be seen that for each just tuning ratio needed, an equal tempered tuning note is not far off. The largest error that the equal tempered tuning scale has, is less than 2%. This is considered small enough to be mostly imperceptible. This twelve tone equal temperament tuning system is what most modern digital instruments are based on[8].

Table 2.2 finally gives us a definition of what is considered to be correct frequency. Other tuning methods exist but since twelve tone equal tempered tuning is the most used, it will be considered the objective of this project. Now that the idea of a correct pitch is well-known, the general pitch correction structure is investigated.

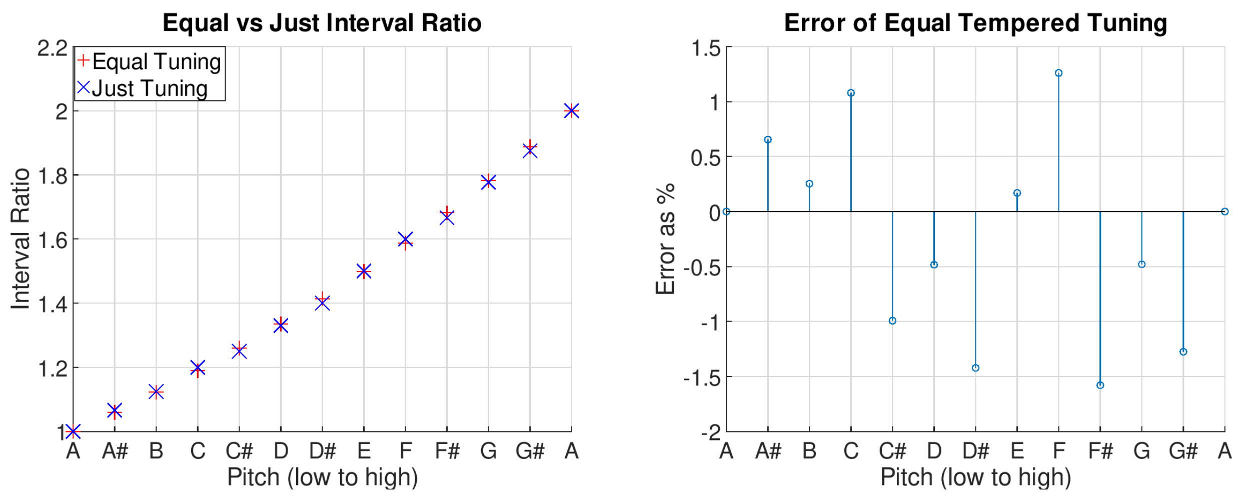


Figure 2.3: Comparison of just tuning and equal tempered tuning

2.2 General Pitch Correction Structure

As discussed in the "History of Audio Pitch Correction" section in the introduction, there are currently three modern pitch correction systems accessible for guidance: AutoTalent[2], AutoTune[1] and Smule[3]. The easiest system to get information from is the open source AutoTalent code. The other two are proprietary and the only reliable information found is that contained in their patents.

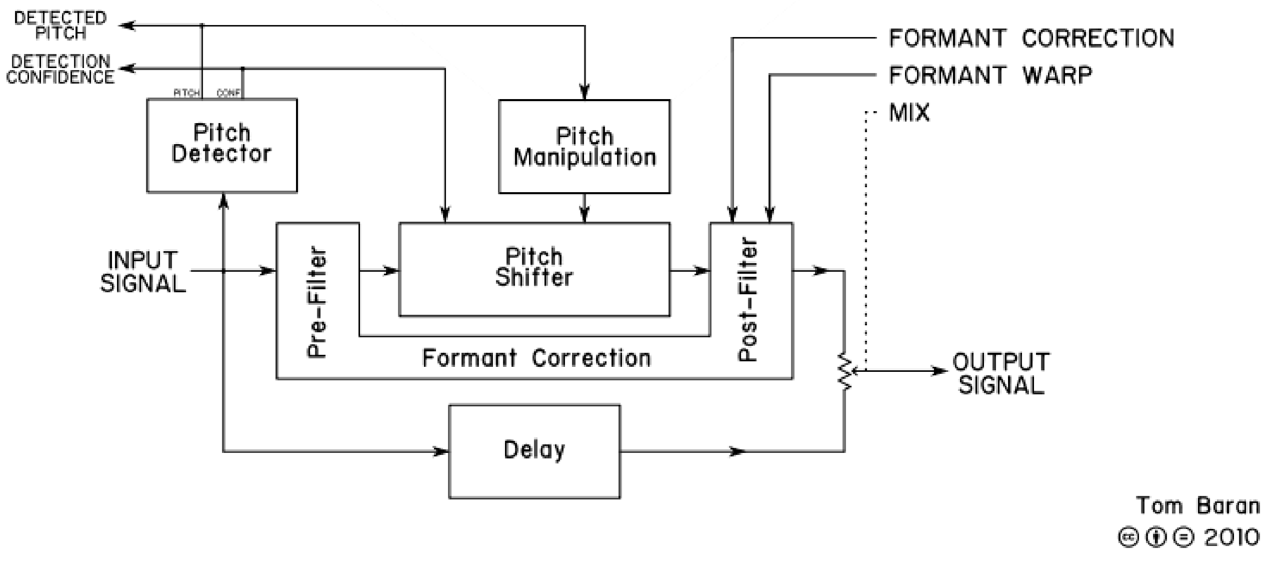


Figure 2.4: AutoTalent Flow diagram[2]

On Tom Baran's AutoTalent website[2], there is some explanation of how his pitch corrector works. Figure 2.4 shows a section of the flow diagram provided by him for his explanation. The full flow diagram is in Appendix B. The input signal is supplied to a pitch detector. This detector works using a type of autocorrelation function mentioned in his explanation on the website. The detected pitch is passed to a pitch manipulation block which is a slightly complicated algorithm(see Appendix B) to choose which pitch the pitch shifter will attempt to shift the output to. A confidence metric is also provided by the pitch detector and eventually used by the pitch shifter to reduce the effect when confidence is low. The pitch shifter works using a

PSOLA algorithm and is claimed to produce less artifacts than a phase vocoder based pitch shifter. A formant correction pre and post filter is also provided.

Formants are the regions in the spectrum where the harmonics of a vocal signal as a group, peaks[9]. The location and size of these formants don't naturally change, but do when pitch is synthetically shifted. This formant filter is an attempt at keeping the regions of the formants consistent before and after the synthetic pitch shift. In the source code for AutoTalent, Tom Baran mentions that he considers the formant filter experimental[2].

The AutoTune patent[1] contains some flow diagrams explaining the workings of the pitch corrector invention. These are very cumbersome and deemed to be unnecessary, even for an appendix. The basic structure is very similar to figure 2.4 and figure 2.5. The patent suggests the use of a modified autocorrelation function to detect the pitch and a method by Lent[10] to shift the pitch.

The Smule patent[3] was investigated for hints about what structure and sub modules were used for their application. From figure 2.5 it can be seen that they have two streams. A melody stream and a harmony stream. The harmony stream is irrelevant for pitch correction and will be ignored. The first block in the melody stream is a low pass filter, downsampler and decimater. This block represents the analog to digital converter and some pre-processing for the rest of the system. Thereafter a pitch detection algorithm finds the pitch being sung. This information, with the pre-processed signal and a target pitch is fed into a pitch shifting algorithm(PSOLA) to correct the pitch. The target pitch is found by some pitch choosing algorithm that uses the knowledge of what harmonies and scale the music is written in.

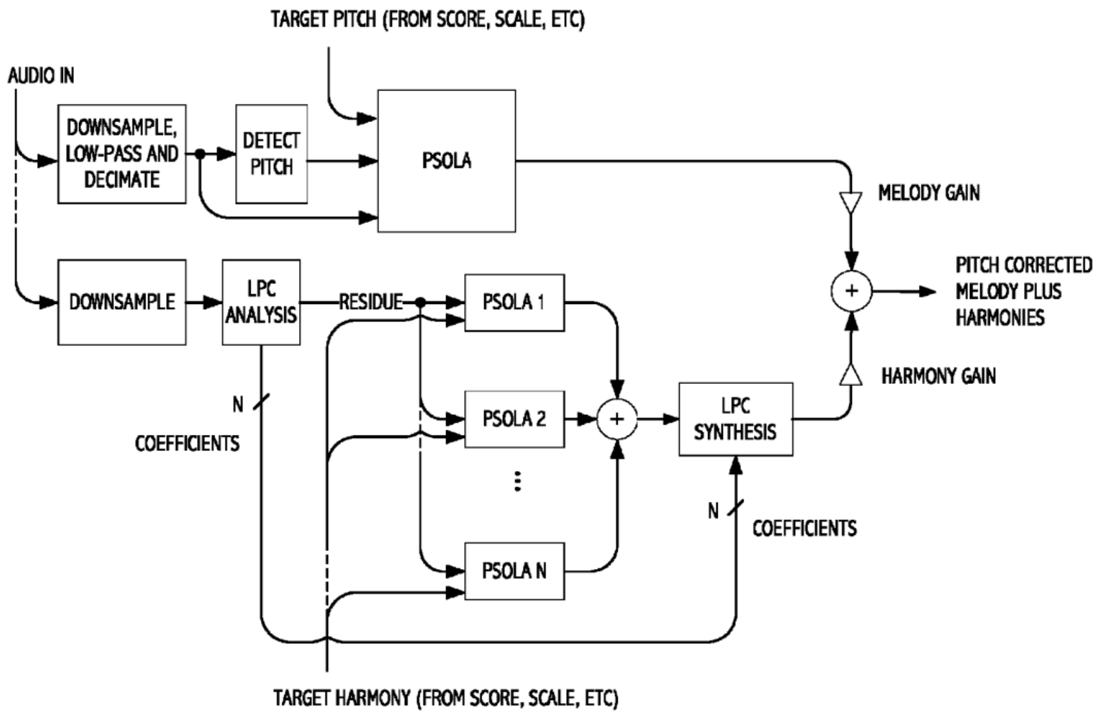

TARGET PITCH (FROM SCORE, SCALE, ETC)
AUDIO IN
DOWNSAMPLE, LOW-PASS AND DECIMATE
DETECT PITCH
PSOLA
MELODY GAIN
PITCH CORRECTED MELODY PLUS HARMONIES
DOWNSAMPLE
LPC ANALYSIS
RESIDUE
PSOLA 1
PSOLA 2
PSOLA N
+
LPC SYNTHESIS
HARMONY GAIN
N
COEFFICIENTS
N
COEFFICIENTS
TARGET HARMONY (FROM SCORE, SCALE, ETC)

Figure 2.5: Smule patent Flow diagram[3]

All three pitch correction systems seem to have some common aspects. They are separated into a frequency detector, a pitch choosing algorithm and a frequency scaling algorithm. Each system has some additional structures other than the basic pitch correction added, either to improve the result or for additional features. The major modules will be investigated further.

2.3 Frequency Detection

Frequency detection is an essential step in the pitch corrector system. Before the pitch can be corrected, it must be known what the current pitch value is. In the field of study, the term pitch detection and frequency detection are interchangeable terms. Both refer to finding the fundamental frequency of a periodic or semi-periodic signal[11]. There are several methods used depending on the application, but all fall under two categories: Time domain methods and frequency domain methods[12].

The time domain methods detects the frequency or timing of some time domain feature that has a known relation to the fundamental frequency of the signal. Examples of features commonly used are peaks and zero crossings. Some time domain methods use auto-correlation to measure similarity between a signal and time-lagged visions of that signal[12].

Frequency domain methods use operations similar to a Fourier-Transform to inspect the frequency components of a signal. Windowing the signal is recommended to reduce spectral smearing[13].

The methods chosen to be investigated is the zero-crossing method and the autocorrelation method. These were chosen because of their simplicity and surprisingly good performance[11]. Frequency domain methods were planned to be investigated and implemented but was left out due to a lack of time.

2.3.1 Zero Crossing Method

The zero-crossing method is a time domain method which uses zero-crossings to determine the period, and hence, the fundamental frequency of the signal. There are two zero crossings in a single period, therefore this detector can produce results at a rate of twice the lowest frequency of the signal. Frequency is related to period by equation 2.2.

$$F_0 = \frac{1}{p} = \frac{1}{2\Delta s} \tag{2.2}$$

Before the signal is passed to the zero-crossing detector some preprocessing is done to accentuate some features. Linear predictive coding is a common preprocessing step used when a zero-crossing detector is used[12]. Figure 2.6 shows a simple graph of how the zero-crossing detector works. It calculates the time elapsed between two consecutive zero crossings and uses equation 2.2 to calculate the fundamental frequency.

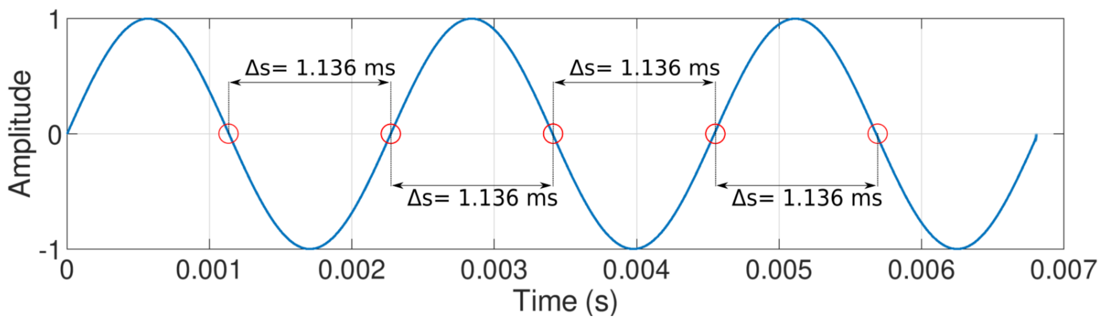


Figure 2.6: Zero crossing detector on 440 Hz sinusoidal signal

Some obvious implementation details and issues can be conceived but was not found to be exhaustively investigated in the literature on the subject. Since this is regarded as a very basic pitch detection algorithm these issues were personally solved and is further discussed in the implementation section.

2.3.2 Autocorrelation Method

As mentioned previously, the autocorrelation method or variants of it seem to be popular among prior art. The AutoTune patent[1] mentions that it uses a variant of the Algorithm and it appears in the source code and explanations of AutoTalent[2]. Smule mentions autocorrelation, among other pitch detection methods, in their patent[3].

The original algorithm for pitch extraction using autocorrelation was first described in a paper written in 1967[14]. A flow diagram of the method described in the original paper was produced by a paper comparing different pitch detection algorithms[11]. This flow diagram is shown in figure 2.7. The algorithm in the flow diagram differs slightly to the algorithm suggested by the original paper. The original algorithm does not have an infinite peak clipper and calculates the clipping threshold differently.

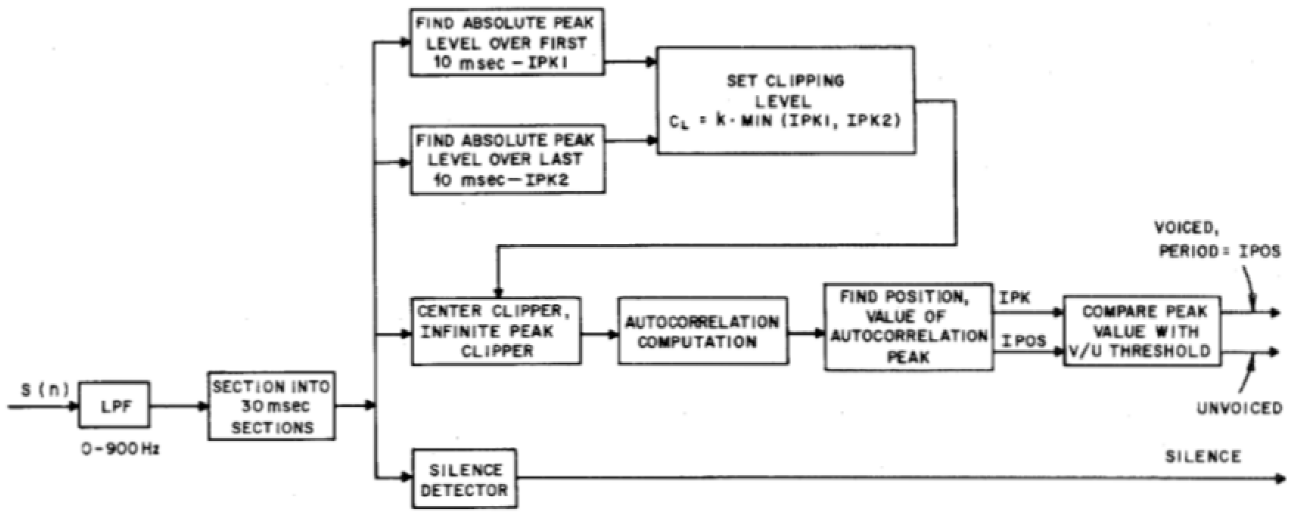

S (n)
LPF
0-900 Hz
SECTION INTO 30 msec SECTIONS
FIND ABSOLUTE PEAK LEVEL OVER FIRST 10 msec – IPK1
FIND ABSOLUTE PEAK LEVEL OVER LAST 10 msec – IPK2
SET CLIPPING LEVEL C_L = k · MIN (IPK1, IPK2)
CENTER CLIPPER, INFINITE PEAK CLIPPER
AUTOCORRELATION COMPUTATION
FIND POSITION, VALUE OF AUTOCORRELATION PEAK
IPK
IPOS
COMPARE PEAK VALUE WITH V/U THRESHOLD
VOICED, PERIOD = IPOS
UNVOICED
SILENCE DETECTOR
SILENCE

Figure 2.7: Autocorrelation Algorithm Flow diagram[11]

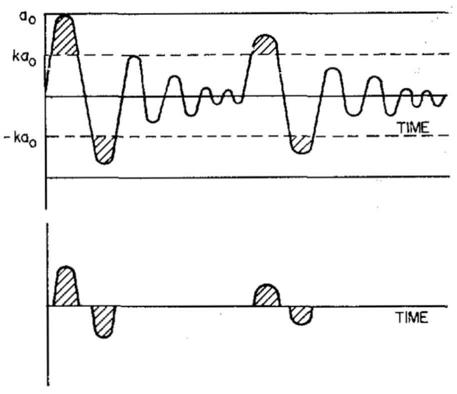


Figure 2.8: Center Clipping [14]

The flow diagram algorithm has a pre-processing step that applies a low pass filter to the signal and breaks the signal up into 30ms overlapping segments. Next, center clipping is applied to a segment of the signal. The thresholds for the clipper, ka_0, is calculated by first obtaining IPK_1 and IPK_2, the maximum of the absolute value of the first and last third of the signal segment respectively. The threshold is set to k times the minimum value of IPK_1 and IPK_2. The value of k is a tuning parameter and a suggested value of 0.67 is given. A center clipper is then applied to the signal as shown in figure 2.8. The autocorrelation of this center clipped signal is then calculated. The difference in position of the first peak after the center peak is calculated. The size of this difference is the period of the signal and the peak of the signal, compared with the main peak, provides a confidence metric.

Center clipping removes information on formants but preserves periodicity[15]. The presence of formants is considered the cause for inaccurate pitch detection using autocorrelation methods.

2.4 Frequency Scaling

Since pitch is logarithmically dependent on frequency, as shown in figure 2.1, the act of pitch shifting is analogous to frequency scaling. This is due to one of the logarithmic identities and is shown in equation 2.3.

$$\begin{aligned} p &= \log(f) && \text{(given)} \\ p_{shifted} &= p + \Delta p && \text{(pitch shifting)} \\ &\downarrow \\ \log(f_{scaled}) &= \log(f) + \log(\Delta f) && \text{(substitute)} \\ f_{scaled} &= f \cdot \Delta f && \text{(frequency scaling)} \end{aligned} \tag{2.3}$$

The general approach taken when doing frequency scaling is to expand or shorten the signal without affecting the pitch and thereafter re-sampling this expanded or shortened version of the signal, to be the same length as the original. This expansion or shortening of the signal is called time scale modification(TSM). Like frequency detection, frequency scaling algorithms can also be divided into time and frequency domain approaches. Both, however, tend to be implemented using a frame-based approach[16].

Figure 2.9 shows the basic idea behind a frame based approach for the TSM process. This signal is stretched to be longer in duration but has the same frequency of the original signal. The exact details of how the synthesis frames are produces is what differentiates approaches. To acquire a frequency scaled signal the signal needs to be re sampled at a new sampling rate equal to the inverse of the time scaling factor.

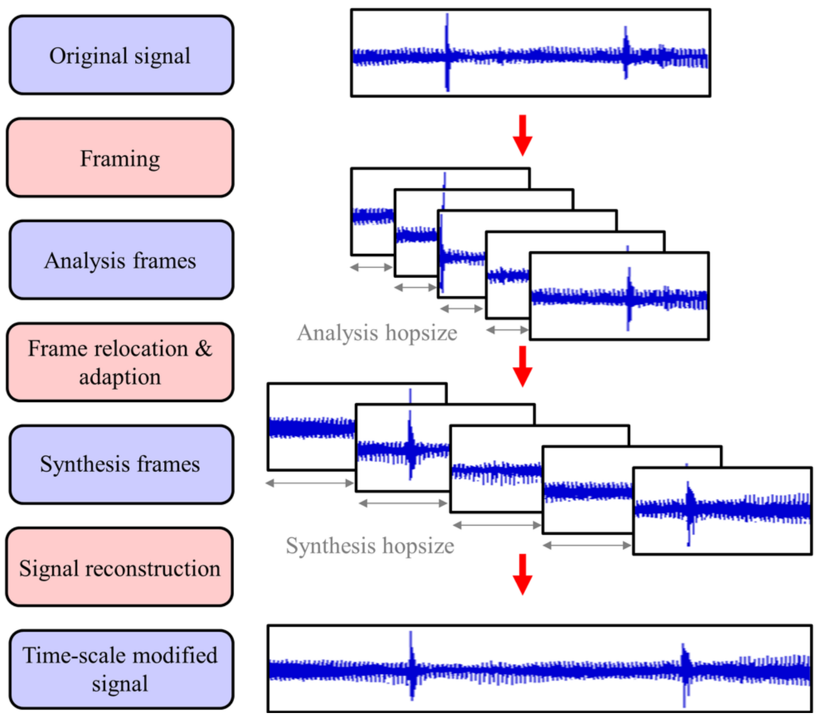


Figure 2.9: Frame based approach of many TSM procedures[1]

[1]https://en.wikipedia.org/wiki/Audio_time_stretching_and_pitch_scaling by Jonathan Driedger

2.4.1 Simple Overlap and Add

The most simple approach to time scale modification, is to simply have the synthesis frames identical to the analysis frames[16]. This approach comes with a couple of unwanted effects but forms a basis to point out the problems that other approaches tries to fix. It points out some subtleties in implementation that is not immediately evident in figure 2.9.

The overlap and add method's complications come in at the overlapping areas. The approach taken in to use a Hanning window when the analysis frames are created. This is to keep a constant magnitude on the overlapping sections. To achieve this a hop size of $H_s = N/2$ is required. Equation 2.4 shows the definition of the Hanning window and equation 2.5 shows the property that preserves amplitude. N is defined as the segment size[16].

$$w(n) = \frac{1}{2}\left(1 - cos\left(\frac{2\pi n}{N-1}\right)\right) \tag{2.4}$$

$$\sum_{a\in\mathbb{Z}} w(n - aH_s) = 1 \text{ for all n, when } H_s = N/2 \tag{2.5}$$

The problems with this approach is that it does not preserve periodic structures. This is because it does no work to align the two waveforms when the are fading into one another. Figure 2.10 shows this visually for a time scale modification which increases the duration of the signal by a factor of 1.8. The output signals duration is increased but shows little resemblance to the initial signal.

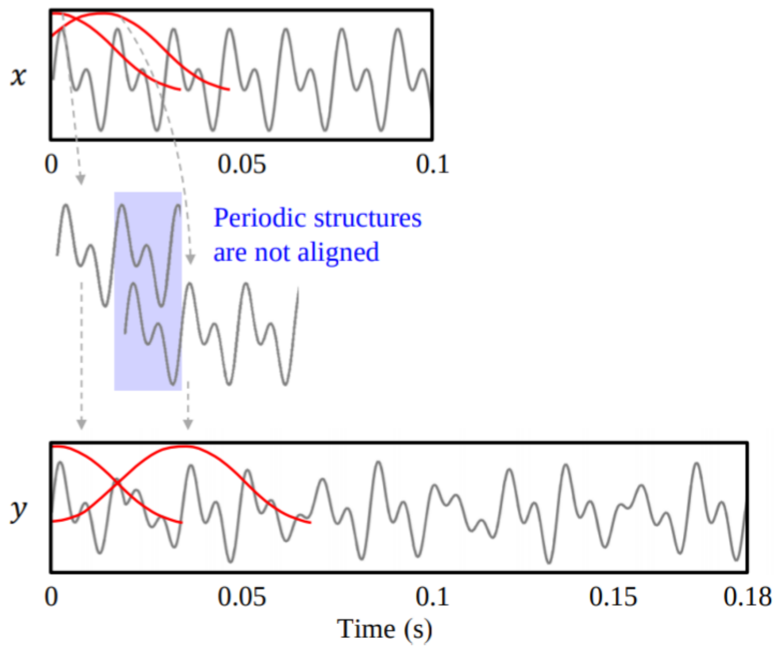


Figure 2.10: OLA harmonic structure distortion[16]

This approach is considered unsuitable for signals with harmonic content but works surprisingly well with percussive signals[16]. The reason for investigating this approach is to be able to give a basis for understanding other approaches and the reasons they implement complex algorithms.

It is recommended to use very small frames sizes, less than 10ms, in order to reduce transient doubling. The effect can be easily imagined when considering an impulse signal. The simple overlap and add approach will cause some doubling of this impulse at the same rate the synthesis frames are constructed. Further details of the implementation of the simple overlap and add method will be discussed in the implementation chapter.

2.4.2 Phase Vocoder

The phase vocoder originates from a different problem but was eventually discovered to be very useful for time scale modification. The algorithm essentially boils down to a complex approach for creating synthesis frames, but has a more interesting story for how it was invented.

The first paper to describe the workings of the phase vocoder was written in 1966 by Flanagan and Golden[17]. They were interested in improving the standard channel vocoder at the time to allow for higher fidelity sound transmission across telephone lines. This was an important field of research at the time because telephone lines were very limited in bandwidth. A prominent figure in the investigation of the phase vocoder is Mark Dolson, an Electrical Engineer focusing on research in music, speech and perception. He wrote a more intuitive tutorial on the phase vocoder and it's various interpretations in 1986[18].

The first interpretation is how the original paper first envisioned it. This is of a filter bank, each band-passed at different frequencies. Each of these sections represent a frequency range. The standard vocoder would only send over the information of each band-pass filters's output amplitude. The phase vocoder, however, sends the phase information too. Figure 2.11 shows how this information is extracted coming from a BPF. This allows the receiver to calculate the actual frequency in that bandpass filter (not just assuming the frequency to be the middle of the BPF) and reproduce a much more accurate signal with these two pieces of information.

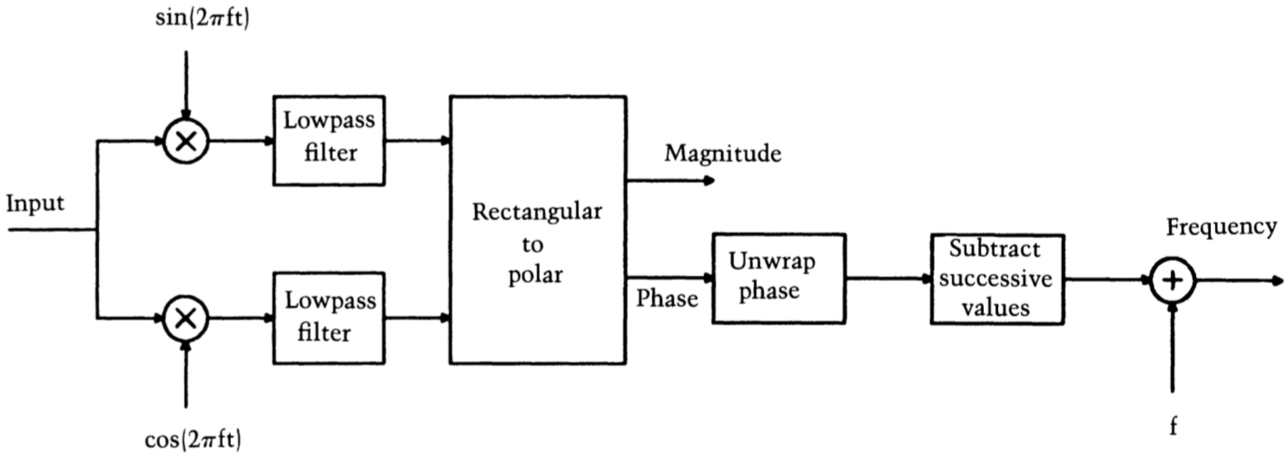


Figure 2.11: Flow Diagram showing Frequency and Phase Extraction from BPF[17]

Interesting blocks in figure 2.7 are the unwrap phase block and the subtract successive values block. These blocks are essentially calculating the offset for the middle frequency of the BPF. The "phase unwrap" block just ensures the phase is within the 0 to 2π domain and the "subtract successive values" block is where the frequency offset calculation happens.

The other interpretation is that of the Short Time Fourier Transform(STFT). The STFT is an operation on a signal where successive overlapping windows of the signal of interest are Fourier transformed to form frames of the Fourier view that change with time. This structure is shown in figure 2.12. The STFT view has been shown to be mathematically equivalent to the filter bank interpretation by Pornoff in 1976[19] and is the approach taken today because of the rise of digital computers and the computationally efficient FFT algorithm.

How the phase vocoder actually manages to change the time base still needs to be explained. The explanation will use the STFT interpretation but a equally valid interpretation in the filter

[2] https://www.researchgate.net/profile/Rajmil_Fischman/publication/231828310/figure/fig6/AS:300578681966597@1448674950601/Fourier-transform-taken-every-eight-samples.png

bank view may help for conceptual clarity. The result of the STFT on the signal is a complex valued two dimensional array. This array can be sampled at no integer time step values, basically in between frames. This new STFT array still needs to be phase corrected. The phase needs to be scaled by the same factor that the time has been expanded by[18]. This operation can be done at the same time the interpolation is done. This algorithm ensures individual frequencies to be lined up between frames. Specifically the algorithm preserves horizontal phase coherence.

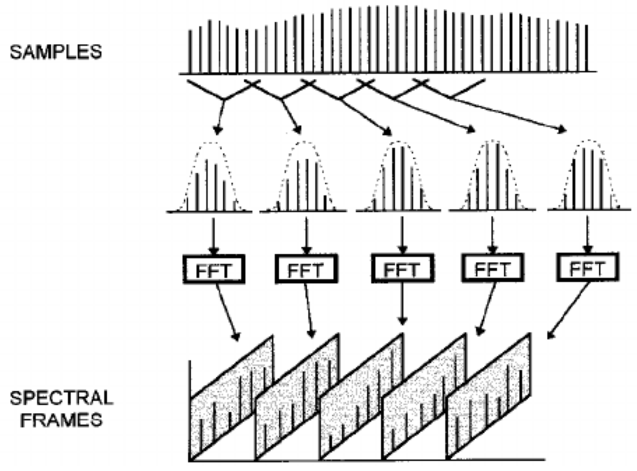


Figure 2.12: STFT frames structure[2]

The phase vocoder is regarded as a extremely high fidelity tool by which to alter the time base of a signal. The artifacts that are introduces have been studied in great detail and improvements on the algorithm have been proposed[20]. These artifacts are due to spectral leakage and the fact that it does not preserve vertical phase coherence. This introduces a reverberation effect or loss of presence that is commonly called "phasiness"[20].

There exist many different implementation of the phase vocoder online. Since this project looks to implement the pitch corrector in Octave, a open source alternative to Matlab, only implementations in Matlab and Octave were inspected. Dan Ellis has an implementation availale on his website[3]. This implementation is written for Matlab but was found to be compatible with Octave. An example of time base modification comes with this implementation. Other Matlab implementations[45] were also inspected for clarity and unbiased implementation details.

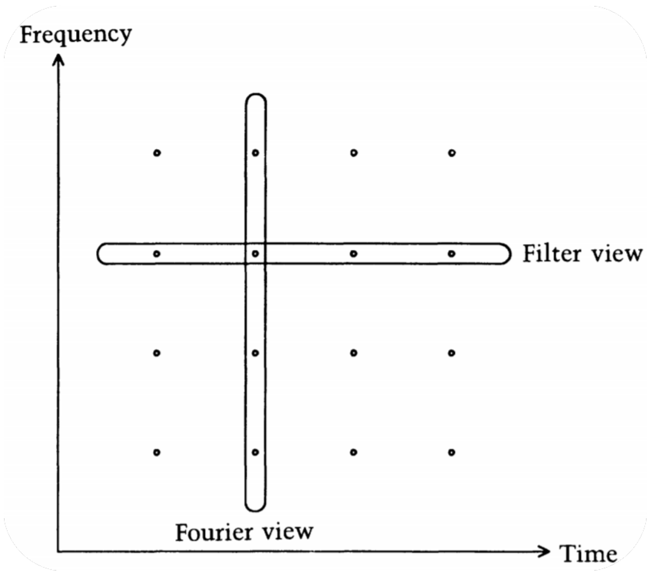


Figure 2.13: Relation between the Fourier view and filter view of the Phase Vocoder[18]

[3]`http://www.ee.columbia.edu/ln/rosa/matlab/pvoc/`

[4]`http://sethares.engr.wisc.edu/vocoders/matlabphasevocoder.html`

[5]`http://www.daimi.au.dk/~pmn/sound/pVoc/index.html`

IMPLEMENTATION

Chapter 3

This chapter describes the design, module implemetation and final implemetations of the pitch correction system. The implementation code was written in Octave, a free and open source alternative to Matlab. Some important, short code-snips are shown and explaind to give deeper insight into the details of the implementation.

Since one of the goals of this report is to compare the performance of the pitch corrector when using different modules, the evaluation metrics used are described first. Theses metrics are described first to reflect the fact that the metrics were designed before the structure and submodules were implemented in an effort to minimise biased results. Once these metrics are well understood, the desing and implementation of the pitch corrector and its submodules are described.

3.1 Evaluation Metrics Design

The goal of using metrics is to give a meaningful quantitative way of measuring the success of the pitch corrector. It provides way to compare the performance of different sub modules and choose the configuration that produces the best result. Since the ultimate goal of an audio pitch corrector is to produce better sounding music, the most rigorous way to judge the effectiveness of the pitch corrector would be to run a psychological survey on which combination of sub modules sound best. This is beyond the scope of this project. Instead, some simplifications are made and metrics are designed based on the research done on music theory. These metrics act as a proxy for the results given by a physiological survey and are much easier and cheaper to run. Two aspects of the pitch corrector are chosen to be assessed. The effectiveness of it and the noise or distortion of the audio signal introduced by the pitch corrector.

3.1.1 Pitch Corrector Metrics

The effectiveness metric is a way of measuring how much the pitch corrector corrects the pitch. This metric will answer the question of how in tune the audio is before and after the correction is applied and produce a number indicating the improvement. The algorithm to achieve this will make use of the frequency contour function. This is a function of the fundamental frequency of a signal over time. Exactly how the frequency contour is extracted from the signal will be discussed in the “Pitch Correction Design” section. From this frequency contour function, it is possible to find the closest correct frequency to this function as shown in listing 3.1.

```
function outContour = getClosestFreqContour(inContour)
  valid = getValidFrequencies();
  for i = 1:length(inContour)
    [dummy index(i)] = min(abs(log(inContour(i)).-log(valid)));
  endfor
  outContour = valid(index);
endfunction
```

Listing 3.1: getClosestFreqContour.m

The exact details of how the correct frequencies are calculated will be explained in the "Target Frequency Contour" section. Listing 3.1 shows how the closest frequency contour is calculated.

Once the closest frequency contour is found, both are converted into a pitch contour by taking the log_2 of the functions. This is done because using frequency contours would eventually weigh the importance of the higher frequencies more than the lower frequencies. A plot of these two contours would allow one to see the intonation error as shown in figure 3.1.

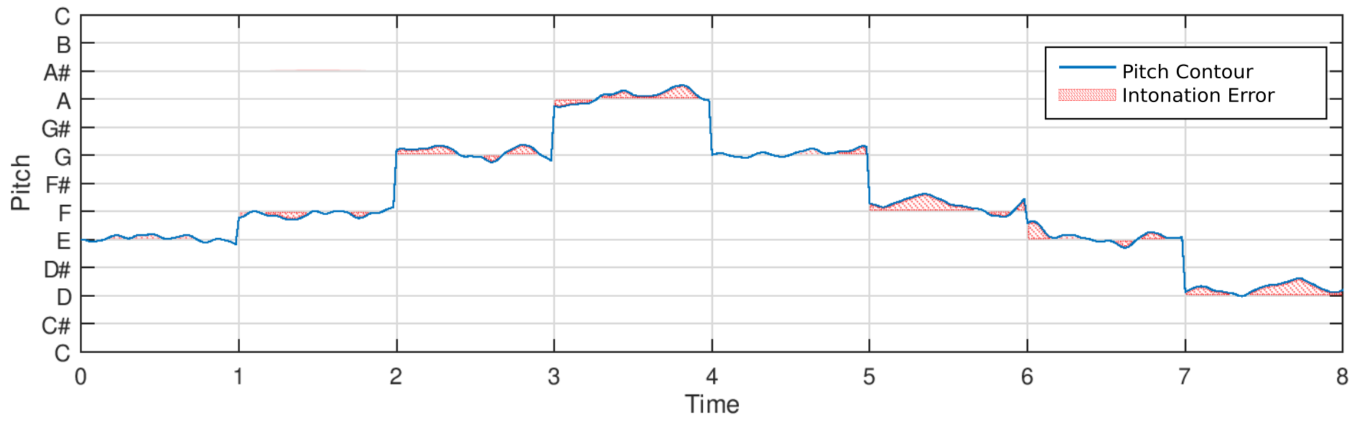


Figure 3.1: Illustration of Intonation Errors in a Pitch Contour Diagram

The red regions in figure 3.1 are intonation errors. To be perfectly in tune would be saying there are no red regions, or equivalently, the error function is zero everywhere. Therefore the smaller the red regions are the better the pitch corrector. A common metric to measure this is the mean square error metric and even has a built-in function in Octave. Therefore the mean square error would indicate how in tune the audio recording is, weighing larger intonation errors greater than small intonation errors. This metric is referred to as the mean squared pitch error and can fundamentally not be greater than 1.736×10^{-3} in the equal tempered tuning system. How this number is calculated is shown in equation 3.1 and is essentially half the distance between two notes in the logarithmic pitch scale.

$$\left(\frac{1}{2}log_2(\sqrt[12]{2}) - log_2(1)\right)^2 \approx 1.736 \times 10^{-3} \tag{3.1}$$

To formalise this metric algorithm, a standard recording needs to be chosen to test with. An exponential chirp signal is chosen, starting at 110Hz, ending at 220Hz and lasting 5 seconds. Figure 3.2 shows the pitch contour of the signal with the closest correct pitch contour. This signal has a mean squared pitch error of 0.59×10^{-3}. The algorithm will measure the mean squared pitch error after the pitch correction is applied and return a ratio of the original and corrected mean squared pitch error. The idea being that the metric essentially says the pitch correction effect causes the recording to be X times more in tune. The goal of the pitch corrector is to maximise this metric.

The effectiveness metric unfortunately relies on the frequency detector to be absolutely correct. This is because it depends on the results of the frequency contour before and after the correction was made. The choice to use a clean chirp signal for this metric was made to minimise dependence on the frequency detector.

Now that the effectiveness of the pitch corrector can be measured, a distortion metric needs to be designed to measure how much the pitch corrector distorts the signal unnecessarily. The idea is to have a signal in a state that it shouldn't require a correction anymore, i.e. Already pitch

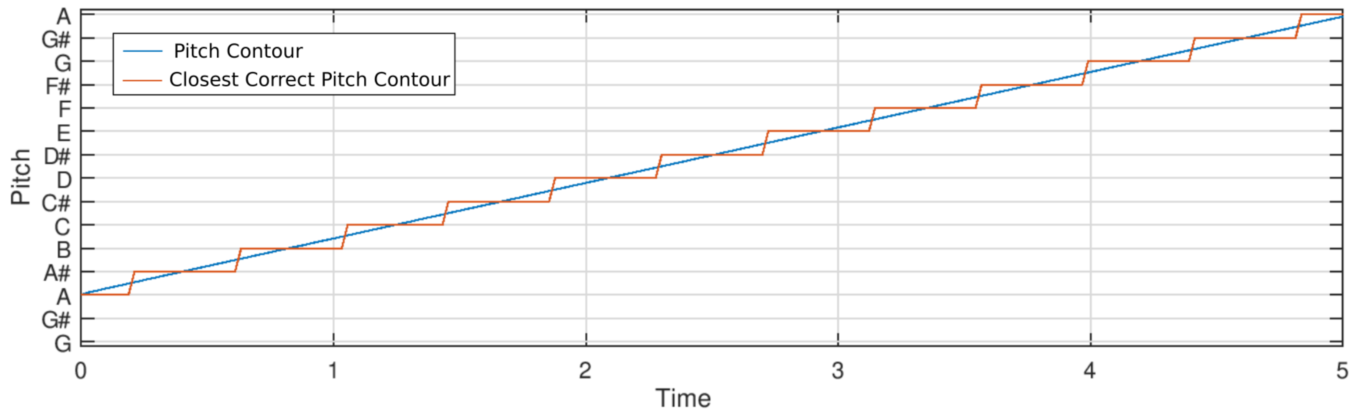


Figure 3.2: Exponential Chirp Contour with Closest Frequency Contour

corrected, and then apply the pitch correction effect and see how much the signal change. The idea is to measure the peak of the autocorrelation of a signal after applying the pitch correction effect. This is the baseline value. Then apply the pitch correction effect again and calculate the maximum value of the cross-correlation between the signal before and after the second pitch correction effect has been applied. The ratio between the maximum cross-correlation value and the baseline value gives in indication to how much the pitch corrector distorts the signal unnecessarily. This ratio is called the distortion metric, measured as a percentage. A pitch correction algorithm will attempt to produce the highest similarity ratio, 100% being a perfect score. Listing 3.2 shows the distortion metric algorithm implemented in Octave.

```
function result = distortionMetric(original,windowSize,hopSize,sf)
  % Do double pitch correction
  first = getCorrectedPitch(original,windowSize,hopSize,sf);
  second = getCorrectedPitch(first,windowSize,hopSize,sf);

  % Get percentage of correlation after second application
  firstCorr = max(xcorr(first));
  bothCorr  = max(xcorr(first,second));
  result = bothCorr/firstCorr*100;
endfunction
```

Listing 3.2: distortionMetric.m

The testing recording for the distortion metric is chosen to be the same as the one for the effectiveness metric except for one thing. The distortion signal needs some added white noise to have the distortion in the whole frequency range accounted for. To avoid the white noise from interfering with the frequency detector, the level is set to -20db since both the detectors are shown to be robust enough to that level of noise in the results chapter.

3.1.2 Noise Robustness Metric

As has already been mentioned, the effectiveness metric relies on the fact that the frequency detector is absolutely correct. To give some sense that the frequency detector is capable of producing valid results, a metric is designed to measure how much noise the frequency detector can deal with before producing unacceptable results. Unacceptable results are seen as a mean square error of more than 0.59×10^{-6}, i.e. three orders of magnitude greater than the mean squared error of the testing signal in figure 3.2. The testing signal will be the same as the one in figure 3.2 since the exact pitch contour of the signal is known. The metric will produce a db level of noise that can be added before the unacceptable mean squared error value has been reached. The goal of the pitch detector would be to maximise this db level of noise added.

3.2 Pitch Corrector Design

The design of the pitch corrector is guided by what was found in the literature review. Specifically the "General Pitch Correction Structure" section was found useful. All the pitch correctors investigated had a modular design that included a frequency detector and a frequency scaler. Some extra modules, unique to a specific pitch corrector, were mostly ignored in the pursuit of simplicity. The "pitch manipulation" and "target pitch" modules represents essentially the same function and was named the "Target Frequency Contour" module.

This chapter describes the general structure of the pitch corrector first, explaining how the data flows through the pitch corrector. Thereafter the concept of segmentation is explained and the exact details of how the pitch contour is calculated is laid out.

3.2.1 Structure

The overall structure of the pitch corrector can be represented in a flow diagram. This structure is shown in figure 3.3. The blocks represent modules that receives inputs and produces outputs. The arrows represents the data and is labeled according to what kind of data it is, pointing in the direction the data flows. The overall structure receives untuned audio and produces tuned audio.

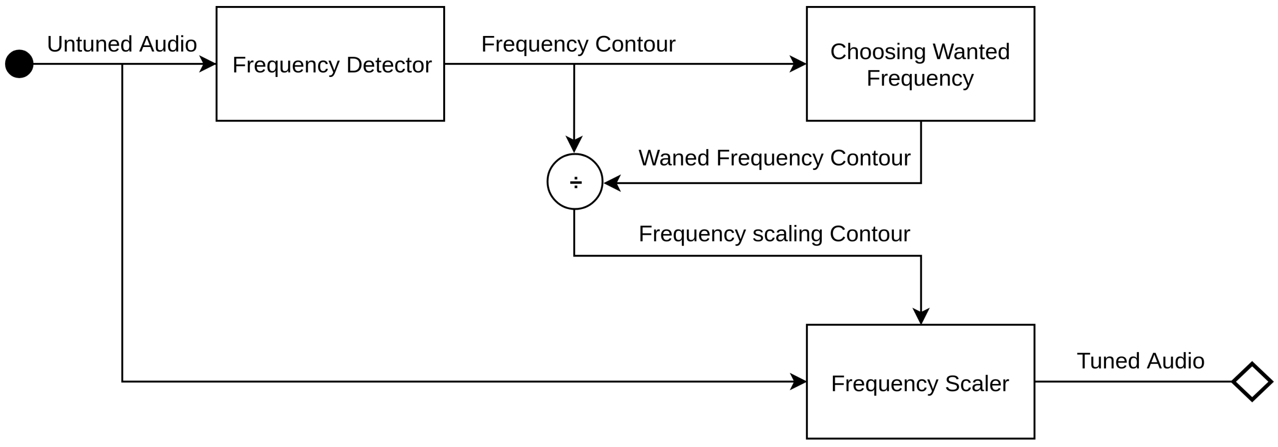


Figure 3.3: Flow Diagram of Pitch Corrector

The Octave code of the structure seen in figure 3.3 can be seen in listing 3.3. From this code, it can be seen that some extra information, other than that provided by the arrows, are needed. This information includes the sampling frequency, the segment size and the hop size and is needed by each module to function correctly. The sampling frequency is self explanatory but segment size and hop size relates to the concept of segmentation, not yet explained. This extends naturally to the idea of a frequency contour and is why the wrapper function `getFrequencyContour` exists, providing an intermediate step to the frequency detection module.

```
function correctedPitch = getCorrectedPitch(original,segmentSize,hopSize,sf)
  % Get frequency scaling contour
  originalFreqContour = getFrequencyContour(original, segmentSize, hopSize, sf);
  closestFreqContour = getClosestFreqContour(originalFreqContour);
  scalingFreqContour = closestFreqContour ./ originalFreqContour;

  % Do pitch shifting
  correctedPitch = getScaledSample(original,segmentSize,hopSize,sf,
   scalingFreqContour);
endfunction
```

Listing 3.3: getCorrectedPitch.m

3.2.2 Segmentation

Segmentation is a concept required by the structure of the pitch corrector. It essentially involves breaking up the input signal into segments, and doing computations on these individual segments. The frequency detector and frequency scaler performs it's operation in the context of a single segment. The idea of segmentation can be seen in figure 3.4. The figure shows a signal being segmented into chunks of 2 048 samples and a 50% overlap. This corresponds to a segment duration of around 46ms, assuming a sampling rate of 44 100 samples per second.

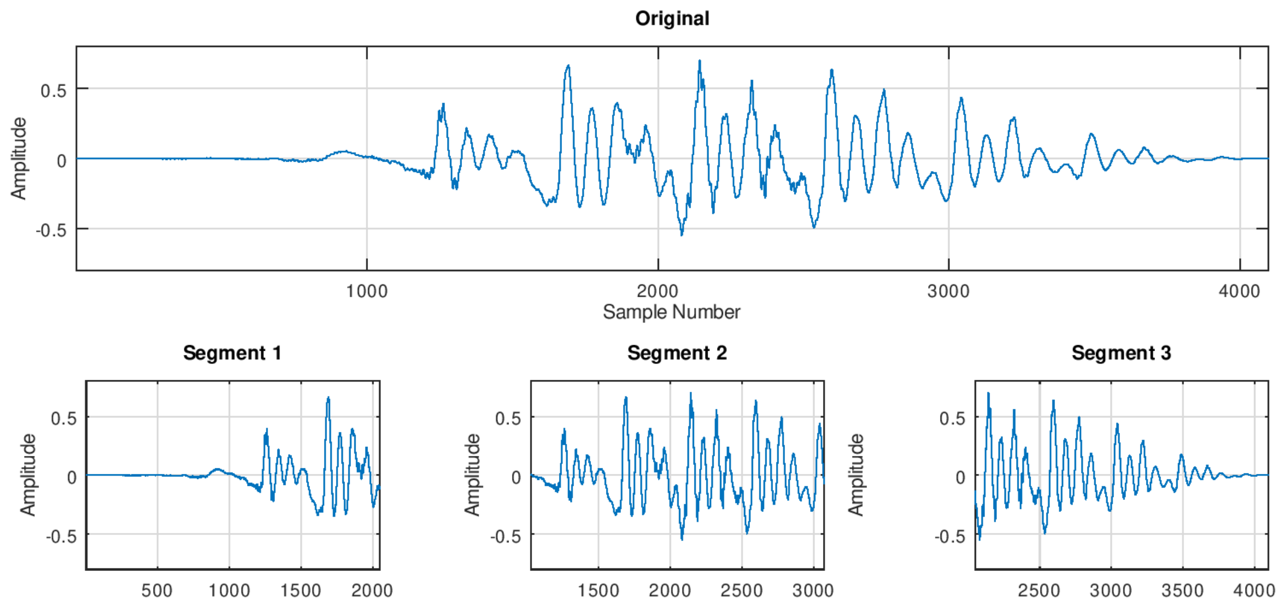


Figure 3.4: Segmentation Visualization

There is an obvious trade-off when considering the overlapping percentage. The more the signal overlaps, the more computational power is needed but it does usually provide more gradual changes when stitching segments back together. For this implementation, the overlap percentage was chosen at whatever was most natural for the pitch shifter(50% for simple overlap and add and 75% for the phase vocoder). A less obvious trade-off is that of segment size. The larger the segment size, the more latency the system will have but will increase the fundamental resolution in frequency for the frequency detection and frequency scaling modules. Practically the pitch detectors needs al least 2 periods to calculate the frequency properly, meaning the segment size needs to be at least 1 764 samples long. This corresponds to two periods of a 50Hz signal at 44 100 samples per second. Listing 3.4 shows how these segments are obtained.

```
function out = segment(input,segmentSize,hopSize)
  i=1;
  j=1;
  while ( j + segmentSize - 1 <= length(input) )
    out(:,i) = input(j:(j+segmentSize-1));
    i = i + 1;
    j = j + hopSize;
  endwhile
endfunction
```

Listing 3.4: segment.m

From this array of segments, it is possible to create the frequency contour. The basic idea is to pass each segment into the frequency detector, producing a list of frequencies over time. Unfortunately implementing exactly this would produce erratic results. The problem comes in when the specified segment is an unvoiced one. This means no instrument is producing sound in that frame and the frequency detector will find a frequency based on the background noise. This has been found to be very high values and can cause issues in the frequency scaler later on. This is unwanted behaviour and a mechanism is built in to allow the frequency detector to simply hold the frequency of the previous segment if it regards the segment as unvoiced. This produces a frequency contour that's much easier to work with. Listing 3.5 shows the Octave implementation of how this is achieved.

```
function contour = getFrequencyContour(sample,segmentSize,hopSize,sf)
  % Pre-filter specific to frequency detector
  [b,a] = butter(8, 250/sf*2);
  sample = filter(b,a,sample);

  % Segment data (Input should be exact length!!!)
  sample = segment(sample,segmentSize,hopSize);

  % Pre populate array
  contour = ones(1,length(sample)/segmentSize)*220;

  % Get Frequency of each segment
  prev = 220;
  for (i = 1:columns(sample))
    contour(i) = getFrequency(sample(:,i),prev,sf);
    prev = contour(i);
  endfor
endfunction
```

Listing 3.5: getFrequencyContour.m

It can be seen that a pre-filter is needed for the frequency detector. The filter is applied in the `getFrequencyContour` function to allow the frequency detector to only consider individual segments, essentially allowing for more elegant code. This pre-filter is specific to the frequency detector and will be discussed more in the frequency detector section.

3.3 Target Frequency Contour

As discussed in the literature review on music theory, the tuning system that will be used in this pitch correction system is the equal tempered tuning system based on the interval ratio of $\sqrt[12]{2}$. This tuning system works by choosing a starting frequency, 440 Hz, and calculating all the other valid frequencies by successively applying the chosen interval ratio. All the produced frequencies in the tuning system will be relative to 440 Hz. Listing 3.6 shows how a list of these valid frequencies are obtained. The list only extends from 55Hz to 3 520Hz but can easily be extended.

```
function valid = getValidFrequencies()
  multiplier = nthroot(2,12);
  base   = 440;
  i = -12*3:12*3;
  valid = 440*multiplier.^i;
endfunction
```

Listing 3.6: getValidFrequencies.m

Now a target frequency needs to be chosen based on the information provided by the frequency contour and the list of available frequencies. The naive approach would be to choose the target contour based on the closest pitch contour, as shown in listing 3.1. This produces unwanted rapid transitions when a slightly noisy contour transitions between two notes. Figure 3.5 shows these unwanted rapid transitions.

These noisy transitions are common in the field of electrical engineering. The solution, in the context of electrical engineering, is to use a Schmitt Trigger, a device that bases it's output, not only on it's current input, but on the previous output supplied by the device. This dependence on the previous state of the device is a form of hysteresis and is a common non-linear effect. Listing 3.7 shows how this Schmitt trigger approach is implemented in Octave.

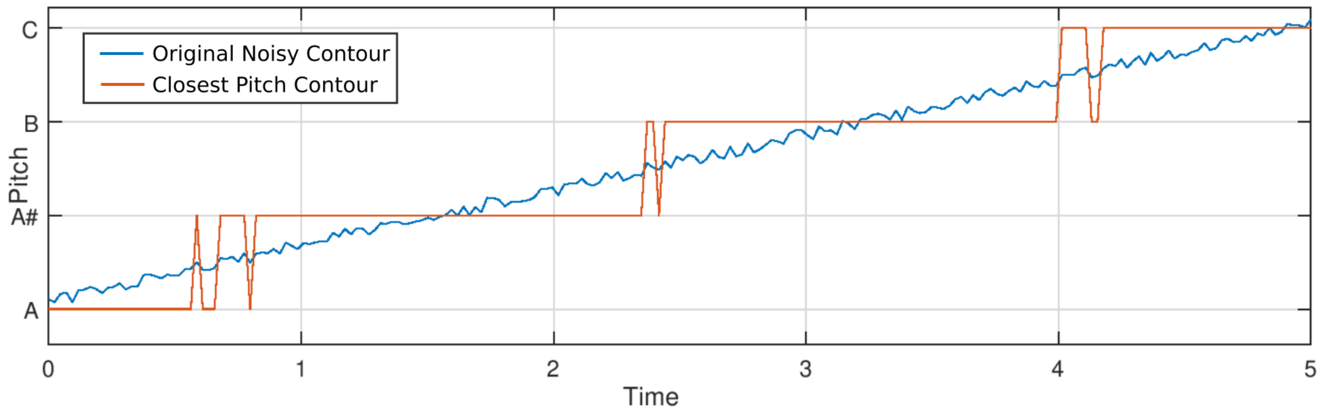


Figure 3.5: Rapid Closest Pitch Contour transitions of a noisy Pitch Contour

```
function outContour = getWantedFreqContour(inContour)
  valid = getValidFrequencies();

  % Get Closest Frequencies
  for i = 1:length(inContour)
    [dummy index(i)]= min(abs(log(inContour(i)).-log((valid))));
  endfor
  outContour = valid(index);

  % Correct according to Schmitt Trigger concept
  prevState = outContour(1);
  threshold = 5;
  for i = 1:length(outContour)
    d = abs(prevState - inContour(i));
    if (d < threshold)
      outContour(i) = prevState;
    else
      prevState = outContour(i);
    endif
  endfor
endfunction
```

Listing 3.7: getWantedFreqContour.m

The approach is to set a threshold that needs to be exceeded in order to change the pitch. The exact value of the threshold is a "tuning value" and would depend on the preference of the performer. This "Schmitt Trigger" approach produces the desired affect when the same noisy input contour is applied. This is shown in figure 3.6.

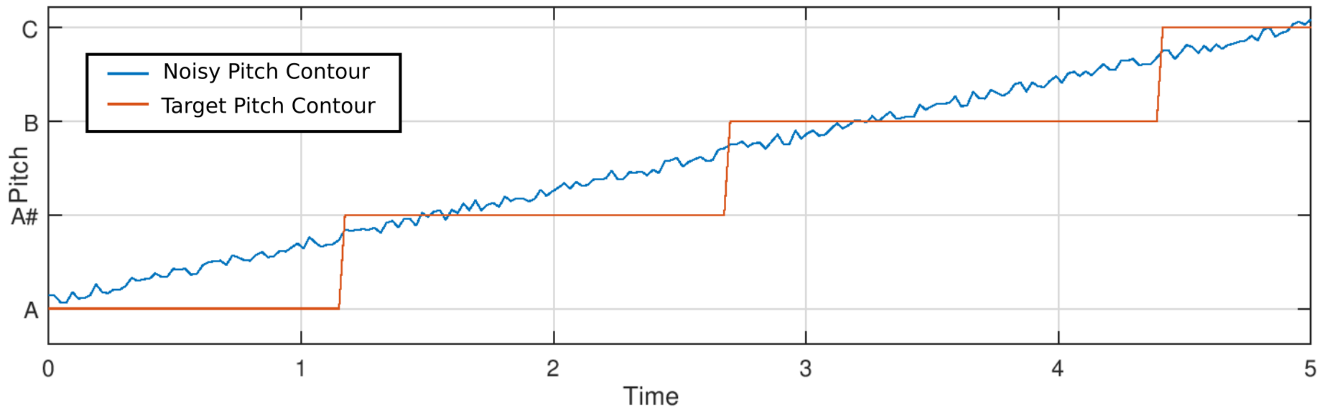


Figure 3.6: Result of Schmitt Trigger approach on Noisy Pitch Contour

3.4 Frequency Detector

The frequency detector module is the basis for the frequency shifter and it's accuracy is vital for the success of the whole frequency correction system. Each implementation has a pre-filter that prepares the signal for the detector. Thereafter the specific algorithm for frequency detection is applied to each segment as shown in listing 3.5. The first thing this algorithm does is determine if the segment is voiced or not. The reason the presence check is done inside the frequency detector function is because it was initially thought each detection approach could yield a different approach to a presence check. This was eventually not investigated further and a single presence check was used for both algorithms. The presence checking algorithm is shown in listing 3.8.

```
function voiced = isVoiced(signal)
  voiced = max(signal) > 0.02;
endfunction
```

Listing 3.8: isVoiced.m

For now, all the algorithm does is check if the amplitude of the incoming signal reaches a threshold. This threshold was found empirically and solved the presence check for all my test recordings. More sophisticated algorithms exist but efforts were focussed on other modules since this was not deemed the module limiting performance of the pitch corrector.

3.4.1 Zero Crossing Method

Before the algorithm for the zero crossing method receives the segments, a pre-processing filter needs to be applied to prepare the frames for the zero crossing detector. LPC was considered but was found to be quite complicated and a simpler option suggested by a website[1] was implemented. This involved a low pass filter, to filter out all the harmonic content and noise that might cause unwanted zero crossings. An 8th order Butterworth filter at 250 Hz was needed to produce results that contained no errors. The algorithm applied to each frame is shown in listing 3.9.

```
function freq = getFrequencyZCM(signal, prevFreq,sf)
  % Somewhat a voice presence check
  if !isVoiced(signal);
    freq = prevFreq;
    return;
  endif

  % Calculate crossing points
  zc = zerocrossing(1:length(signal),signal);

  % If fewer than 3
  if (length(zc) < 3)
    freq = prevFreq;
    return;
  endif

  % Force odd zero crossing amount
  if ( rem(length(zc),2) == 0 )
    zc = zc(1:end-1);
  endif

  % Get considered length
  consideredLength = zc(end) - zc(1);

  % Calculate frequency
  sum = length(zc)-1;
  freq = sum/2*(sf/consideredLength);
endfunction
```

Listing 3.9: getFrequencyZCM.m

[1]https://sound.eti.pg.gda.pl/student/eim/synteza/leszczyna/index_ang.htm

The first block in listing 3.9 calls the voicing check function and only if it return true, continues the algorithm. This is an attempt to reduce unnecessary computation. Next, the algorithm uses the built-in Octave function to find the location of the zero-crossings. The next two blocks relates to a quirk of the zero crossing method illustrated in figure 3.4.1. This figure shows what happens when the segment under inspection has a slight DC-value.

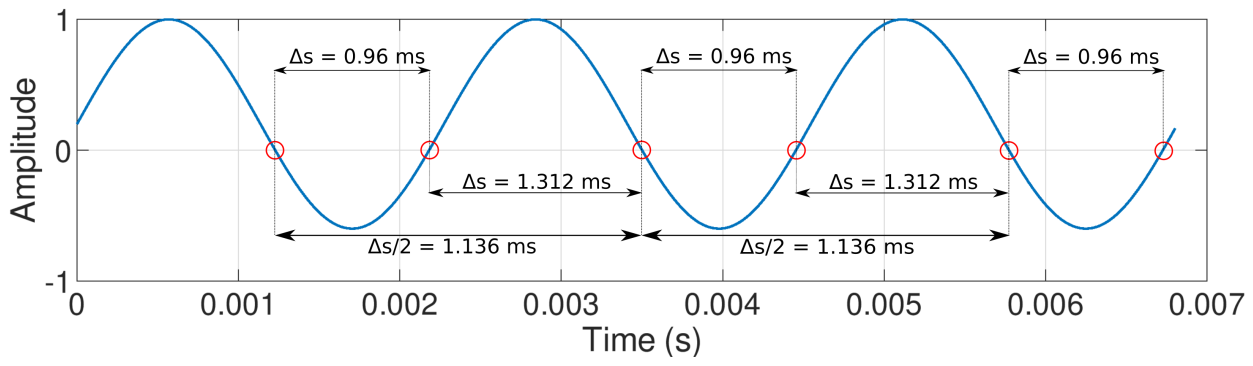


Figure 3.7: Zero crossing detector on 440 Hz sinusoidal signal

The frequency of the signal in the graph is a 440 Hz sinusoid. The formula in equation 3.2 requires Δs in order to to calculate the fundamental frequency of the signal. Since the length of Δs changes for each successive zero crossing, using the naive implementation of the formula would estimate frequencies of 521 Hz and 381 Hz even though the fundamental frequency of the segment is 440 Hz. Therefore some averaging needs to be done. Averaging two consecutive Δs values would produce the correct result as shown in figure 3.4.1. More generally, averaging an even amount of Δs values would produce much more accurate results. This involves ensuring the considered length of zero crossings is more than three and odd valued. This is exactly what the Octave implementation does.

$$F_0 = \frac{1}{p} = \frac{1}{2\Delta s} \tag{3.2}$$

The second to last block of code gets the length of the region from the first to the last zero crossings considered. This is going to be the region being averaged. The last block calculates the frequency from the information of the amount of zero crossings considered and the length of the region considered. With this algorithm, an estimate of the fundamental frequency for a segment can be calculated.

3.4.2 Autocorrelation Method

As with the zero crossing method, a pre-possessing LPF is applied. It was found that this filter did not need as strict parameters and was set to 900 Hz. This was on the recommendation in the paper comparing pitch detection methods[11]. This was found to be sufficient for all testing recordings.

Before the auto-correlation method could be implemented, a center clipping function needed to be written. This center-clipping is needed to remove harmonic content and formants but retain the fundamental periodicity of the signal. Listing 3.10 shows the implementation of this clipping function. This is a straight forward implementation and the code is considered self explanatory.

```
function clipped = clip(signal,threshold)
  % Force all values under threshold to zero
  signal(abs(signal)<threshold) = 0;

  % Relevant indices
  positiveAndGreater = (signal>0) & (signal>threshold);
  negativeAndGreater = (signal<0) & (signal<-threshold);

  % Reduce values past threshold
  signal(positiveAndGreater) = signal(positiveAndGreater) - threshold;
  signal(negativeAndGreater) = signal(negativeAndGreater) + threshold;
  clipped = signal;
endfunction
```

Listing 3.10: clip.m

Now that a center-clipping function has been written, the autocorrelation approach to frequency detection is implemented. The Octave code is shown in listing 3.11. A voice checking, as in the zero-crossing method, is done first. Thereafter the signal needs to be center-clipped. The literature review suggested taking the maximum value of the first and third part of the segment and scaling that by 0.67. The reason for only considering the first and last third is not fully understood and the decision was made to consider the whole segment. Now the clipped signal is autocorrelated. The peaks of this auto-correlated signal needs to be found using a built-in Octave function. The `findpeaks()` function requires the input to be greater than 0. All the peaks int the second half, greater than 80% of the center peak, are returned. The distance to the second peak is considered the period of the signal.

```
function freq = getFrequencyAutoCorr(signal, prevFreq,sf)
  % Somewhat a voice presence check
  if !isVoiced(signal);
    freq = prevFreq;
    return;
  endif

  % Center Clip Signal
  signal = clip(signal,0.6*max(abs(signal)));

  % Do correlation operation on clipped signal
  x = xcorr(signal);

  % Format and normalize correlation output
  x = x(floor(length(x)/2):end);
  x = x - min(x);

  % Find peaks of correlation
  [peak peakIndex] = findpeaks(x,"MinPeakHeight",0.8*x(1));

  % Check if valid period exists
  if (length(peakIndex)>1)
    freq = 1/peakIndex(2)*44100;
  else
    freq = prevFreq;
  endif
endfunction
```

Listing 3.11: getFrequencyAutoCorr.m

To demonstrate the function of the center clipping, figure 3.4.2 shows the result of the same segment with and without the clipping applied. The segment is of a vocal recording containing all the formants and harmonics a signal is expected to contain.

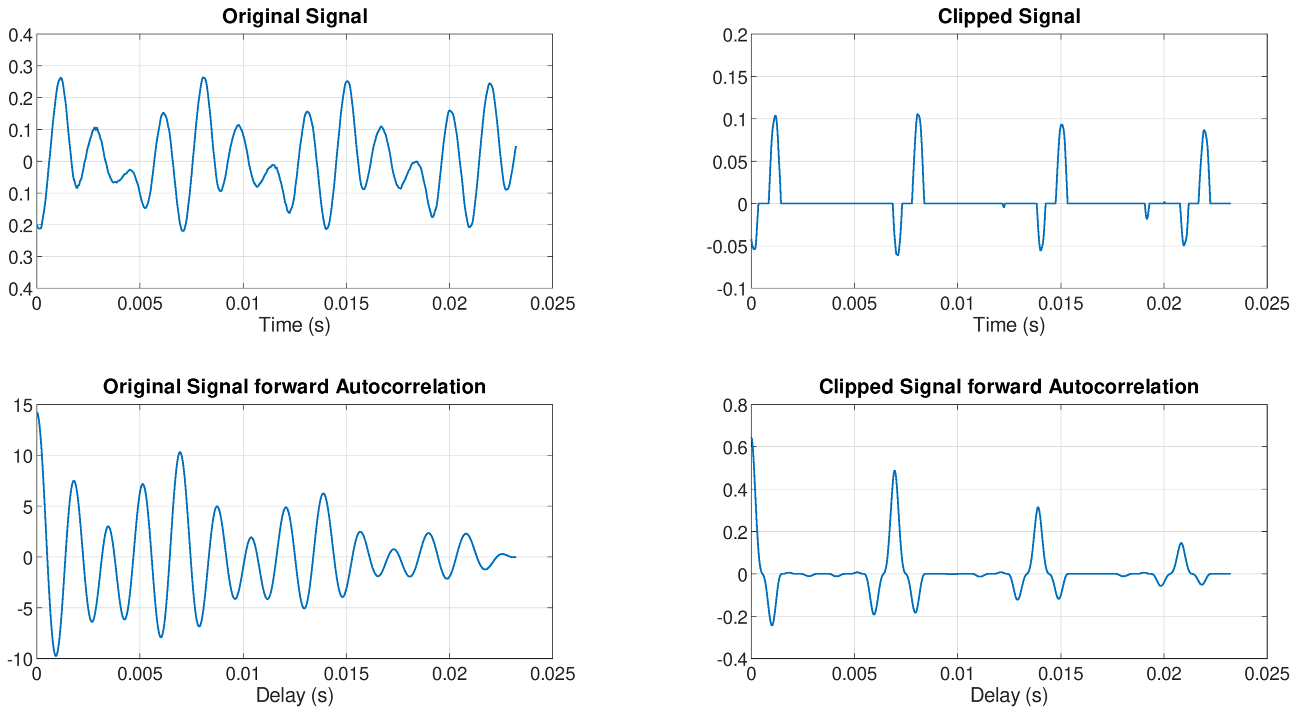


Figure 3.8: Comparison of center-clipped vs non-clipped signal on autocorrelation detector

Below is the result of the second half of the autocorrelation function of the clipped and non-clipped signal. The distance from the value at zero delay and the first peak is the period of the signal. The figure makes is obvious that center clipping produces a less ambiguous period. Further testing could to be done to calculate the ideal values for the clipping threshold and peak finding threshold. The chosen values of 0.6 and 0.8 respectively, was considered to perform good enough and time was spent working on other parts of the pitch correction system.

3.5 Frequency Scaler

The frequency scaler was considered the most difficult module in the pitch correction system. In-depth understanding needed to be acquired before anything worked at all. A test was devised to see if the pitch shifter improved. The test was to scale a 220 Hz sinusoid sound wave, five seconds in duration, up and down by 50% according sin function of a two and a half second period. The quality of this scaled 440 Hz pitch was listened to and changes were made depending on intuition gained from the listening experience. This evaluation method can not be considered a rigorous metric but was found to be very useful in the development of the algorithms.

3.5.1 Simple Overlap and Add

The simple overlap and add method was understood first but actually implemented after the phase vocoder. This was because it was not regarded as a serious approach. This intuition was correct, as will be shown. The merit for including this method is to illustrate it's shortcomings and to be a precursor for the synchronized overlap and add method used by most pitch shifters. Unfortunately the synchronized overlap and add approach was not implemented due to time constraints. This method feels a bit like a useless appendage but should still be considered a stepping stone and reference point to proper methods.

The Octave code of this method is cumbersome and a straight forward implementation of what is described in the literature review. For this reason the implementation code for this method is added in Appendix C. Instead of discussing the Octave code, the results of the frequency scaler will be inspected.

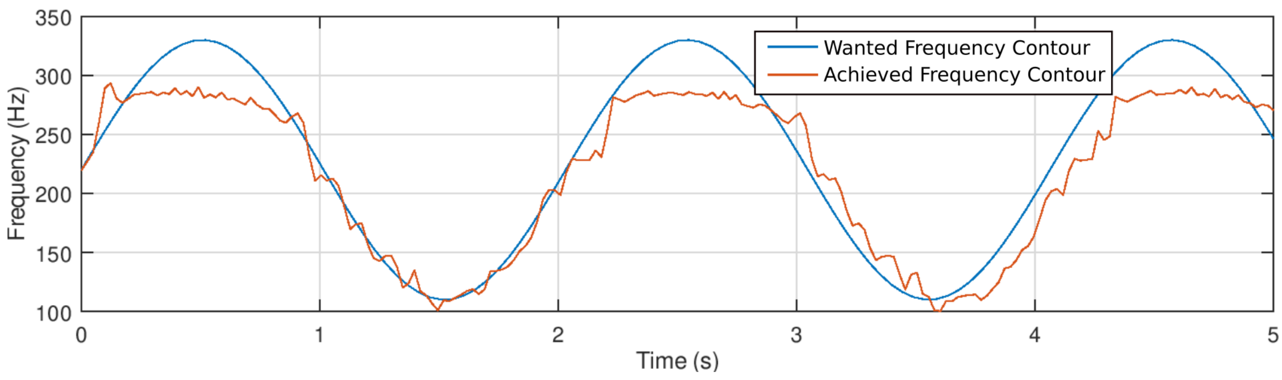


Figure 3.9: Scaling Test of Simple Overlap and Add Algorithm

From figure 3.9 it can be seen that somewhat pitch shifting is achieved. This is not nearly good enough for a high-fidelity application, but is a step in the right direction. Some aspects to point out on figure 3.9 is that the algorithm seems to have trouble with scaling to a higher frequency. The achieved contour follows the wanted contour much closer on the downward section. What's happening here is slightly misleading, or rather, not the full story. The frequency scaler seems to achieve scaling in discrete steps. These are log frequency steps, or alternatively, pitch steps. Figure 3.10 shows the same plot, except with pitch on the y-axis. Here it can be seen that the pitch shifting algorithm has a resolution of 3 semitones. The goal of the pitch shifter is to shift pitch within a semitone, making this pitch shifter not nearly good enough.

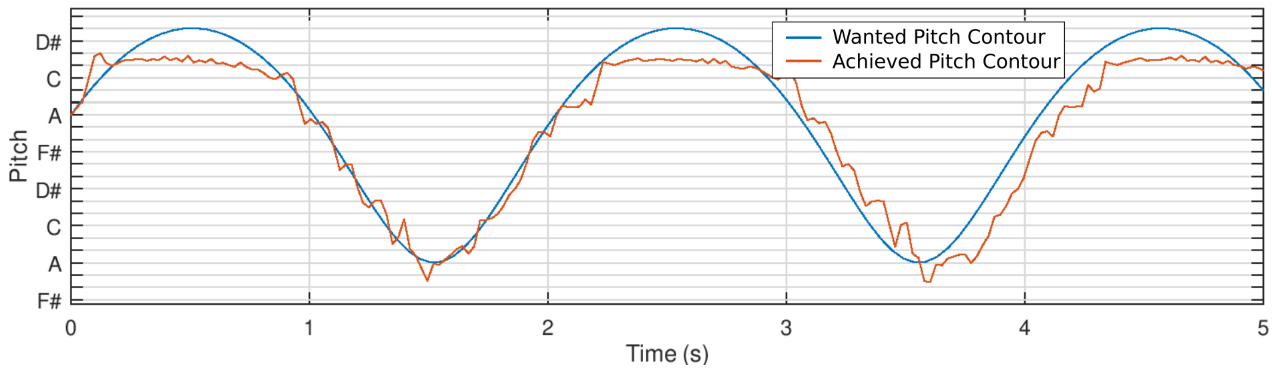


Figure 3.10: Pitch graph of figure 3.9

3.5.2 Phase Vocoder

The Phase vocoder was the most interesting algorithm investigated. The details of how the approach works has been explained in the literature review. The implantation Dan Ellis provides on his website[2] was used. Other implementations were also inspected, but the one provided by Professor Ellis was the cleanest, simplest approach. The implementation itself will not be explained but rather the interface provided will be explained. It will be viewed as a library, equivalent to the other Octave functions provided built in. The Octave code needed to implement arbitrary pitch shifting, using this library, will be explained.

The implementation consists of three functions:

- `stft(x, f, w, h, sr)`
  A function that receives a signal x, does f-point FFT's on w-sized windows spaced h samples apart and returns a form of STFT that the rest of the library knows how to deal with.

- `pvsample(b, t, hop)`
  A function that receives a STFT array b, a new time-base t, and a hop size h, and returns a STFT array, re-sampled at the new time-base according to the phase vocoder algorithm.

- `istft(d, ftsize, w, h)`
  A function that receives a STFT array d, with a FFT size specified by ftsize, window size specified by w, hops size specified by h and returns a signal created by an overlap and add method from the IFFT's of the STFT frames.

The website suggests values for the window size and fft size of 2 048 and 512 respectively and that the functions be executed in the order shown above. The only input to really consider is the new timebase t. The normal timebase is the integer values, starting from zero and having a length equal to the amount of STFT frames. Providing an array t, of 0.5 unit increments and executing the code is equivalent to stretching the input recoding by a factor of two. Re-sampling this recording at half the original sampling rate will cause all the frequencies to be scaled by a factor of two and now the sample will have a length equal to the original. This is how pitch shifting by a constant factor is achieved using the phase vocoder.

The timebase, however, does not need to be spaced equally. It can be spaced at arbitrary intervals. The re-sampling operation can also be done arbitrarily using spline interpolation. This theoretically allows one to scale the frequency by an arbitrary time dependent factor. This is exactly what is required for pitch correction. The function indicating the time dependent scaling factor is referred to as the scaling ratio contour. This algorithm is shown in listing 3.12

[2] `http://www.ee.columbia.edu/ln/rosa/matlab/pvoc/`

```
function scaledSample = getScaledSample(original,segmentSize,hopSize,sf,
   scalingRatioContour)
  addpath("lib/");
  % Get STFT of original
  spec = stft(original',segmentSize,segmentSize,hopSize);

  % Get Sampling Points
  [samplePointsSTFT samplePointsX]  = getSamplePoints(scalingRatioContour,
   hopSize,segmentSize); % CHECK n-1!!!

  % Actually apply arbitrary re-sampling of STFT
  newSpec = pvsample(spec,samplePointsSTFT,hopSize);

  % Get the time stretched signal x
  stretchedX = istft(newSpec,segmentSize,segmentSize,hopSize)';

  % Interpolate to pitch shifted sample by arbitrary function
  scaledSample = interp1(stretchedX, [samplePointsX], 'spline');

  % Remove any NAN values that creeps in
  scaledSample(isnan(scaledSample)) = 0;
endfunction
```

Listing 3.12: getScaledSamplePV.m

All the steps have already been explained and should be straight forward. The only step that needs further elaboration is the `getSamplePoints` function. This is a custom function and is essentially a mapping of the scaling ratio contour to sample points for the STFT array and the spline interpolation function. The Octave code is shown in listing 3.13.

```
function [samplePointsSTFT samplePointsX] = getSamplePoints(scalingRatio,hop,
   windowSize)
  % Initialize variables
  samplePointsX = 0;
  pointerOut = 1;
  samplePointsSTFT(pointerOut) = 0;

  % Loop over each scaling ratio point
  for (i = 1:length(scalingRatio)-2)

    % Fill sample points in STFT space until next scaling ratio is reached
    while (samplePointsSTFT(pointerOut) < i-1)
      samplePointsSTFT(pointerOut+1) = samplePointsSTFT(pointerOut) + 1/
   scalingRatio(i);

      % Add sample points in time domain every 4th STFT point !!!75% overlap
   assumed!!!
      if (rem(pointerOut,4)==0)
        samplePointsX = [samplePointsX (samplePointsX(end) : scalingRatio(i) : (
   samplePointsX(end)+windowSize))];
      endif
      pointerOut = pointerOut + 1;
    endwhile
  endfor
endfunction
```

Listing 3.13: getSamplePoints.m

The code in listing 3.13 essentially packs the STFT sample points at intervals equal to the inverse of the shifting ratio required at that point, until the next integer value is reached. This creates a density of sampling points, equal to the inverse of the shifting ratio, for the STFT frames. While this STFT sampling points are being obtained, the corresponding re-sapling points for the spline re-sampler are obtained. This is done by considering every 4th STFT frame, assuming an overlap of 75%, and creating corresponding sampling points spaced at intervals proportional to the scaling contour.

This approach was written as a quick attempt to doing arbitrary frequency scaling. Once it was written it wasn't even properly tested for a couple of weeks because it was thought to be too inelegant. When tested, it produced surprisingly good results. These results can be seen in a scaling test shown in 3.11.

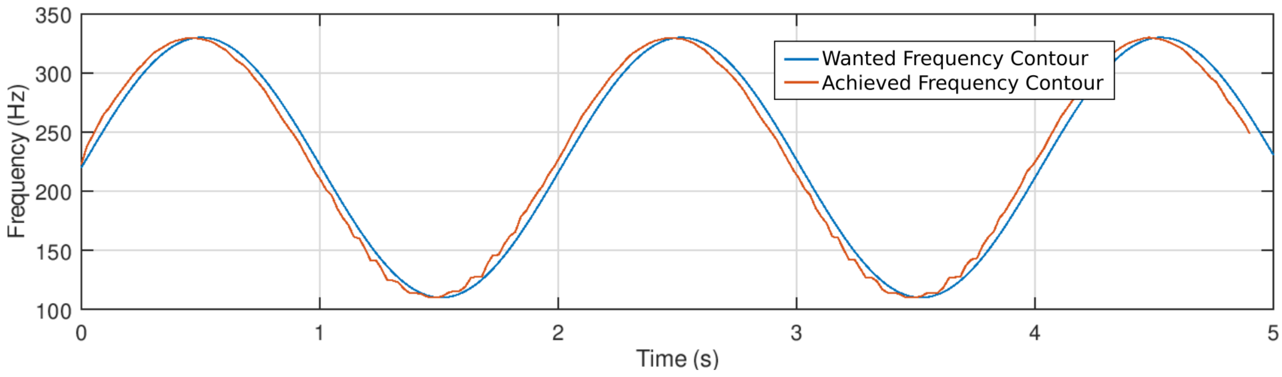


Figure 3.11: Scaling Test of Phase Vocoder

It not only outperformed the simple overlap and add method but when the pitch shifting resolution was inspected it seemed to produce results capable of use pitch correction. This is shown in figure 3.12. The slight phase lead in both figures are misleading. The reason they arise is because the vocoder considers the next frame when doing the scaling. In reality the audio will be delayed by a frame and the phase lead will be non-existent. The effect was left in the graphs because it was difficult to distinguish the two contours when they lined up almost perfectly.

The mapping function is still considered a hack and not the correct implementation. But since it produced results capable of doing pitch correction, it was left as is and not investigated further.

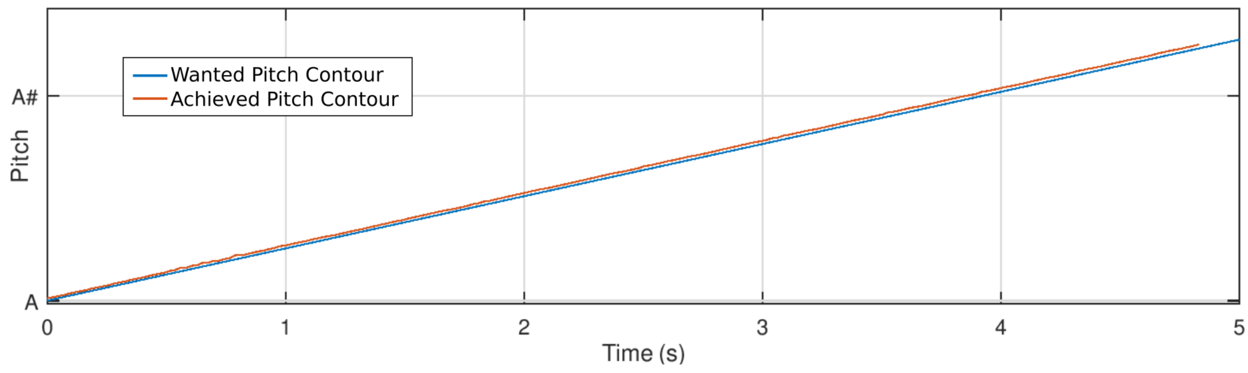


Figure 3.12: Pitch Resolution of Phase Vocoder

3.6 Concept Expansion

During development, some ideas arose of other, similar, effects that can be applied using the tools developed. Some quick implementations of these tools were written to see if they were fruitful. The ones that worked will be quickly described in this section.

One, obvious, wanted effect would be to allow a user to specify the wanted pitch contour during the mixing stage of a song. This would allow the singer to have a recording of an arbitrary pitch contour and this contour can be shaped into and arbitrary different contour, perhaps containing a melody. This affect could be referred to as “post-correction”. This is in contrast to the main algorithm being developed that intends to correct in a “Real Time” context.

A similar idea is to allow the user to shift by a constant factor. This could allow the performer to sing at a different vocal range than would naturally be possible. Multiple streams of these pitch shifts can be created and combined to create a harmonization effect. These effects could be applied in real time alongside the pitch correction algorithm or during a ”post-correction” stage.

RESULTS

Chapter 4

In this chapter, the results of metrics designed for the frequency detector and pitch correction system will be given and interpreted. The frequency detector will be discussed first and the pitch correction system will follow. Some real vocal recordings will also be corrected using a few different configurations of the pitch correction system. This is to give some confidence that the system can hadle real data and doesn't just rely on clean signals the metrics have been designed for. These examples are not rigorous by nature and should just be considered as a final real world demonstaration.

4.1 Frequency Detector

The noise robustness metric for each of the two pitch detection algorithms were run. This checks how much noise can be introduced into a signal with a known pitch contour before the pitch detector produces unacceptable results. Unacceptable is seen as having a mean squared pitch error of more than 0.59×10^-6. The metric will be quoted in decibels of noise.

Figure 4.1 shows a graph plotting the square pitch error, of the Zero Crossing frequency detector, as a function of additive noise in decibels. The square pitch error is plotted using log_{10} scaling to make the graph more readable.

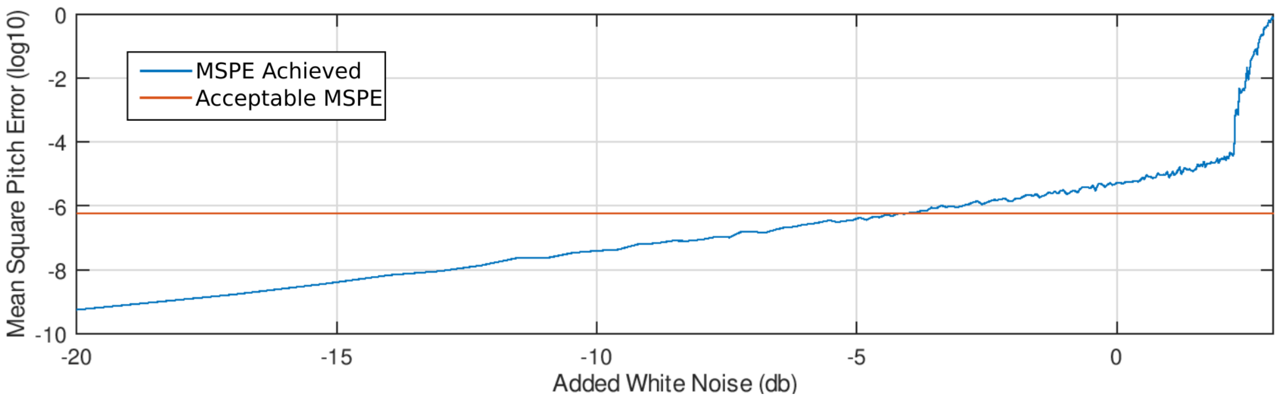


Figure 4.1: Zero Crossing Method Noise Robustness

Since the input signal has an amplitude of 0 decibel, the signal to noise ratio can easily be determined. The graph shows that the zero crossing method requires a SNR of 4.5db in order to function at a acceptable level.

Figure 4.2 shows a similar graph but of the autocorrelation method. The SNR required for acceptable performance is 17.8db. This is worse than the expected and indicates an error in implementation. It is postulated that this is a resolution error and oversampling of the signal before autocorrelation is required. It is also worth mentioning that very little time was spent inspecting the autocorrelation code and that mistakes are highly likely.

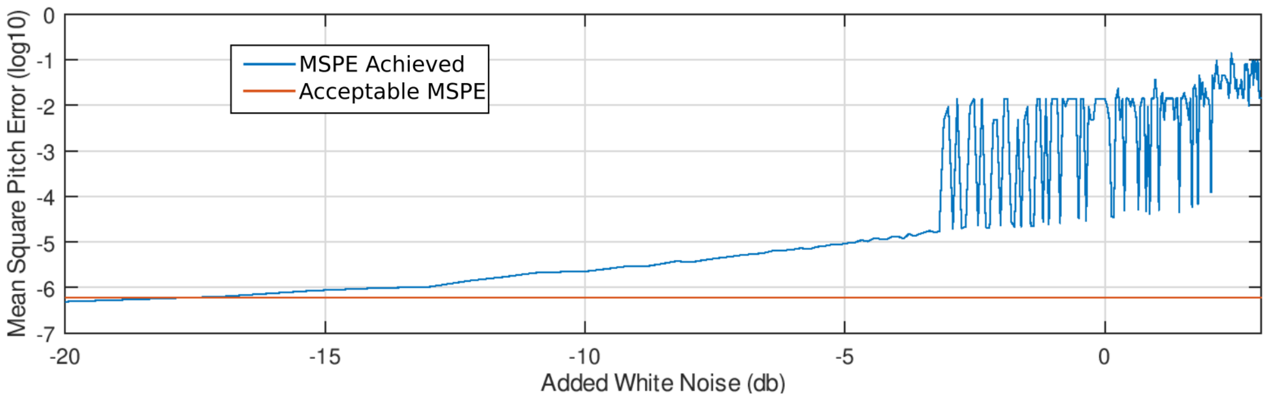


Figure 4.2: Autocorrelation Method Noise Robustness

4.2 Pitch Corrector

Because of time constraints, only the combination of the zero-crossing method and the phase vocoder, as sub-modules, will be tested. This combination does also seem like the most likely combination to produce the best results. This is because of the results seen when testing the sub-modules individually and during the development of the modules.

The first metric to test is the effectiveness metric. The goal is to produce a number to say: “The pitch corrector improves the pitch accuracy by X times”. Figure 4.3 shows the pitch contour plot of the test defined in the implementation chapter.

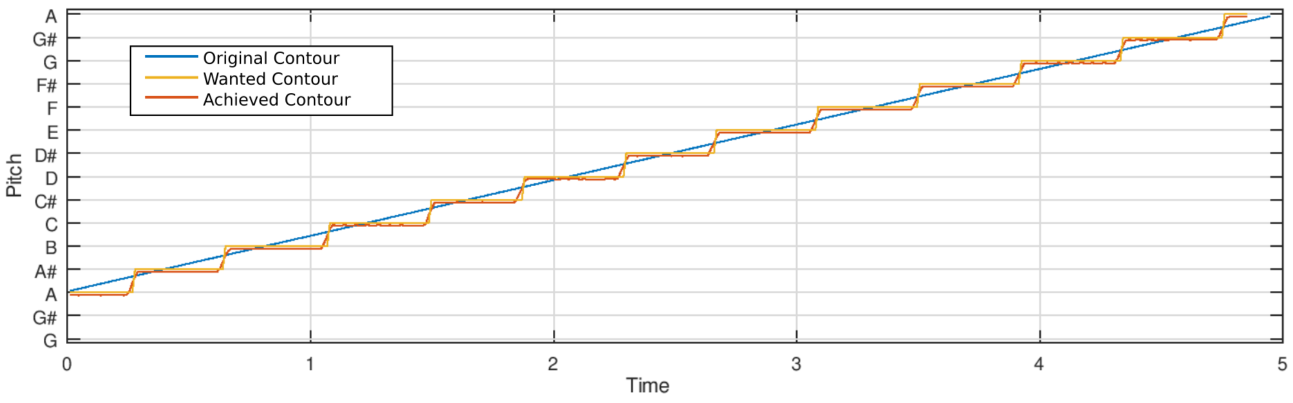


Figure 4.3: Visual Depiction of Effectiveness metric using Phase Vocoder and ZCM

The achieved pitch contour very narrowly follows the wanted pitch contour. The mean squared pitch error before the correction was 0.59×10^{-3}, and after the correction, 0.13×10^{-3}. This results in a pitch accuracy improvement factor of 4.38.

Now that it is known by how much the system improves the pitch accuracy of the signal, an indication of how much the signal is getting distorted by the system needs to be acquired. This is given by the distortion metric defined in the implementation chapter. To give some visual indication of what the distortion metric is measuring, a logarithmically scaled spectrogram of the signal is shown in figure 4.4.

This spectrogram shows the original signal on the left, followed by the result of a single application and double application of the pitch correction effect. The pitch correction system sees to introduce high frequency content on transitions between notes. The second application of

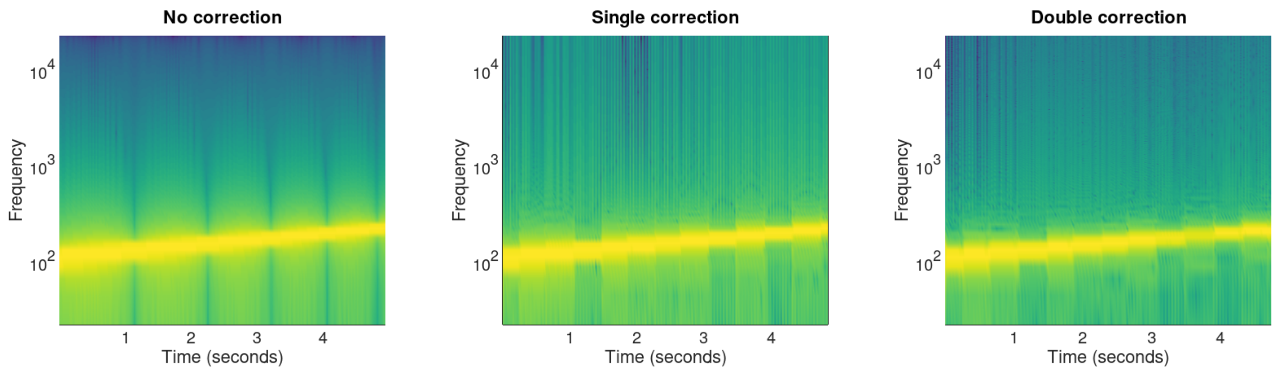


Figure 4.4: Visual Depiction of Distortion metric using Phase Vocoder and ZCM

the effect, however, seems to slightly remove the high frequency content on the transitions. This could be seen as a positive effect and should be kept in mind when considering the distortion metric.

The result of the distortion metric is a quote of a “percentage similar”. The goal of a pitch corrector would be to maximise this similarity percentage. The result of the distortion metric is 44% similarity. Another, perhaps even more meaningful, assessment of similarity may be to normalize the maximum value of the autocorrelation between the first and second application of the correction effect, not with the maximum value of the autocorrelation of the middle signal, but rather the third. The result of this similarity quote is 61%.

These metrics are a proxy to the performance of the system on real recordings. To give some evidence that the system does work on a real vocal recording, a plot of the original pitch contour, wanted pitch contour, and corrected contours are shown in figure 4.5. This is a recording of the author singing the tune of “Hansie Slim”... badly.

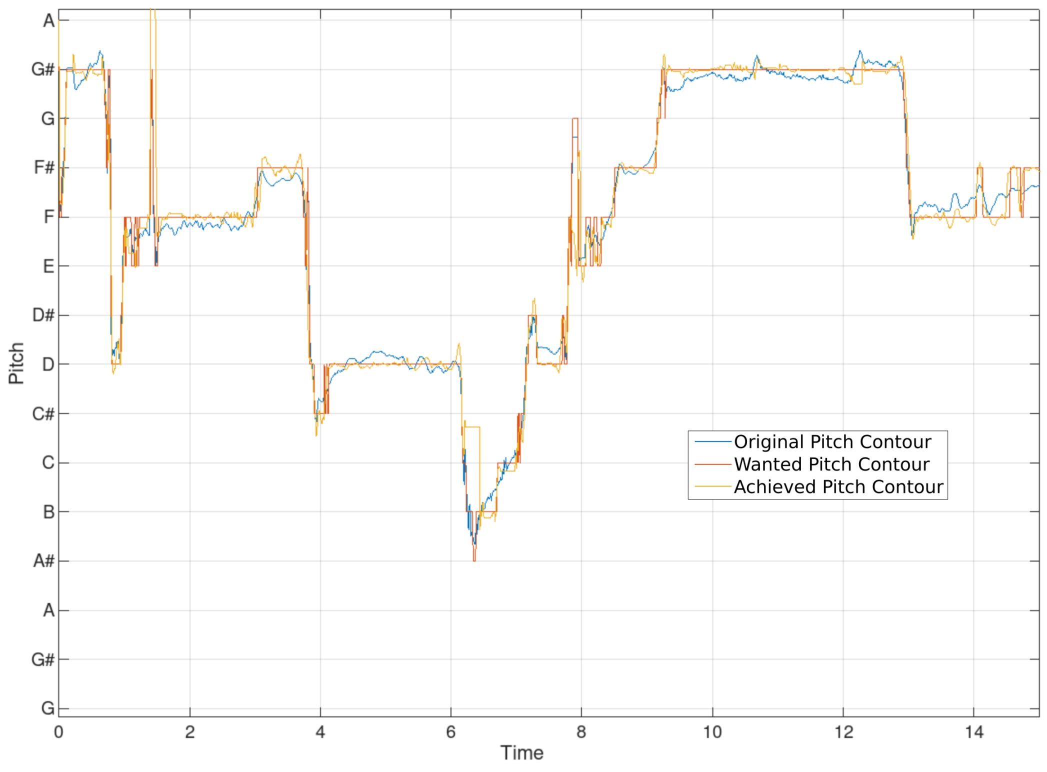


Figure 4.5: Vocal recording demonstration of pitch correction

CONCLUSIONS

Chapter 5

5.1 Findings

The objective of this thesis was to design and test a pitch correction system. Metrics were designed to be a proxy measurement for aesthetic improvement of the recordings. These metrics tried to capture the improvement in pitch accuracy and the distortion caused by the effect. A frequency detector metric was designed to measure the robustness the frequency detection module has towards additive white noise.

Two frequency detection modules were implemented. One implementation was based on using the zero-crossing method and the other was based on an autocorrelation algorithm. The zero-crossing method displayed more noise robustness than the autocorrelation method. The zero-crossing method required a signal to noise ration of 4.5db while the autocorrelation method required a ratio of 17.8db.

Two frequency scaling modules were implemented. One using a very basic overlap and add approach, and another using a sophisticated phase vocoder approach. From tests in the implementation chapter, it was obvious that the overlap and add method was not an appropriate method to use for pitch correction. The overlap and add method produced a pitch shifting resolution of greater than 3 semitones while the phase vocoder approach showed pitch shifting resolution greater that the test could measure.

Due to time constraints, the metrics developed were only tested on one combination of sub-modules. This combination is that of the zero-crossing method as a frequency detection module and the phase vocoder as the frequency scaling module. This combination produced a pitch improvement factor of 4.38 and a similarity percentage of 44% to the uncorrected recording. Real vocal recordings were also tested, proving the system can work with real data. Some manipulation and re-recording was required to produce satisfactory results.

Due to the facts presented above, the pitch correction system is not yet considered robust to real audio recordings. It works in laboratory conditions and requires cherry picked data to perform well. This report is considered a stepping stone towards a system that can perform well in a real world scenario. Further investigation is required to produce a system that has the desired effect. Recommendations are made to inform such an investigation.

5.2 Recommendations

- The design of the system as a whole needs to be relative to a time base. This means sending a time-base array, t, along with any calculation done on a segment This will make some implementation details easy and elegant. Debugging will also be much easier.
- A frame based approach needs to be taken more seriously. Currently the system calculates the contours separately and passes information about the signal as a whole between functions. This was easier to implement but caused much larger hassle later on. Functions passing single frames will work much easier.
- The state of the art for small pitch shifting is considered the synchronised overlap and add algorithm. The next frequency scaling algorithm to implement should be the synchronised overlap and add approach.
- The autocorrelation frequency detector should outperform the zero-crossing method. The reason it does not is most probably due to interpolation being required to provide enough resolution to function properly.
- The autocorrelation frequency detector needs to be tuned scientifically. The threshold values are currently picked from intuition but tests to calculate their optimal value is considered to have a chance of significantly improving performance.
- The inelegant mapping algorithm used by the phase vocoder, mapping an arbitrary frequency scaling contour to sample points for the STFT array and spline interpolation function requires some investigation. An email was sent to Professor Dan Ellis, the creator of the phase vocoder implementation being used, to get some ideas. He responded(too late) with a link to a GitHub repository[1] that has a correct implementation of the time stretching counterpart of the mapping function.
- Some parameters, such as pitch acceptableness error, was taken at an arbitrary value. These values need to be determined psychologically.

[1]`https://github.com/dpwe/pitchfilter/blob/master/resample_map.m`

REFERENCES

[1] H. A. Hildebrand, "Pitch detection and intonation correction apparatus and method," Oct. 26 1999. US Patent 5,973,252.

[2] T. Baren, "Autotalent v0.2." `tombaran.info/autotalent.html`. Accessed: 2018-10-5.

[3] P. R. Cook, A. Lazier, T. Lieber, and T. E. Kirk, "Pitch-correction of vocal performance in accord with score-coded harmonies," Oct. 21 2014. US Patent 8,868,411.

[4] H. E. Heffner and R. S. Heffner, "Hearing ranges of laboratory animals," *Journal of the American Association for Laboratory Animal Science*, vol. 46, no. 1, pp. 20–22, 2007.

[5] S. S. Stevens and J. Volkmann, "The relation of pitch to frequency: A revised scale," *The American Journal of Psychology*, vol. 53, no. 3, pp. 329–353, 1940.

[6] E. Terhardt, "Pitch, consonance, and harmony," *The Journal of the Acoustical Society of America*, vol. 55, no. 5, pp. 1061–1069, 1974.

[7] N. H. Fletcher and T. D. Rossing, *The physics of musical instruments*. Springer Science & Business Media, 2012.

[8] J. P. Burkholder, D. J. Grout, and C. V. Palisca, *A history of western music*. WW Norton & Company, Inc., 2010.

[9] J. Sundberg, "The acoustics of the singing voice," *Scientific American*, vol. 236, no. 3, pp. 82–91, 1977.

[10] K. Lent, "An efficient method for pitch shifting digitally sampled sounds," *Computer Music Journal*, vol. 13, no. 4, pp. 65–71, 1989.

[11] L. Rabiner, M. Cheng, A. Rosenberg, and C. McGonegal, "A comparative performance study of several pitch detection algorithms," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 24, no. 5, pp. 399–418, 1976.

[12] W. Hess, *Pitch determination of speech signals: algorithms and devices*, vol. 3. Springer Science & Business Media, 2012.

[13] F. J. Harris, "On the use of windows for harmonic analysis with the discrete fourier transform," *Proceedings of the IEEE*, vol. 66, no. 1, pp. 51–83, 1978.

[14] M. Sondhi, "New methods of pitch extraction," *IEEE Transactions on audio and electroacoustics*, vol. 16, no. 2, pp. 262–266, 1968.

[15] J. C. R. Licklider and I. Pollack, "Effects of differentiation, integration, and infinite peak clipping upon the intelligibility of speech," *The Journal of the Acoustical Society of America*, vol. 20, no. 1, pp. 42–51, 1948.

[16] J. Driedger and M. MÃller, "A review of time-scale modification of music signals," *Applied Sciences*, vol. 6, no. 2, 2016.

[17] J. L. Flanagan and R. Golden, "Phase vocoder," *Bell System Technical Journal*, vol. 45, no. 9, pp. 1493–1509, 1966.

[18] M. Dolson, "The phase vocoder: A tutorial," *Computer Music Journal*, vol. 10, no. 4, pp. 14–27, 1986.

[19] M. Portnoff, "Implementation of the digital phase vocoder using the fast fourier transform," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 24, no. 3, pp. 243–248, 1976.

[20] J. Laroche and M. Dolson, "Improved phase vocoder time-scale modification of audio," *IEEE Transactions on Speech and Audio processing*, vol. 7, no. 3, pp. 323–332, 1999.

APPENDIX

Chapter 5

A) Imperfect Tuning System Proof

To prove[2] that no perfect tuning system exists, is equivalent to proving the following proposition:

$$\nexists r \in \mathbb{R} \text{ s.t. } R \subseteq \{r^n | n \in \mathbb{N}_0\} \text{ where } R = \{\frac{1}{1}, \frac{15}{16}, \frac{9}{8}, \frac{6}{5}, \dots\} \tag{5.1}$$

R is the set of all the harmonic ratios required in the wanted tuning system. Only two of these ratios, the most fundamental ratios in any tuning system, are required for the proof. The octave interval, ratio 2/1, and the perfect fifth, ratio 3/2. The proof approach is a proof by contradiction.

Assume there exists:

$$r^n = \frac{2}{1} \text{ and } r^m = \frac{3}{2} \text{ s.t. } r \in \mathbb{R} \text{ and } n, m \in \mathbb{N}_0 \tag{5.2}$$

Taking the $\log_2$ of each equation in 5.2 the following two equations are obtained:

$$\log_2(r) = \frac{\log_2(2)}{n} \tag{5.3}$$

$$\log_2(r) = \frac{\log_2(3) - \log_2(2)}{m} \tag{5.4}$$

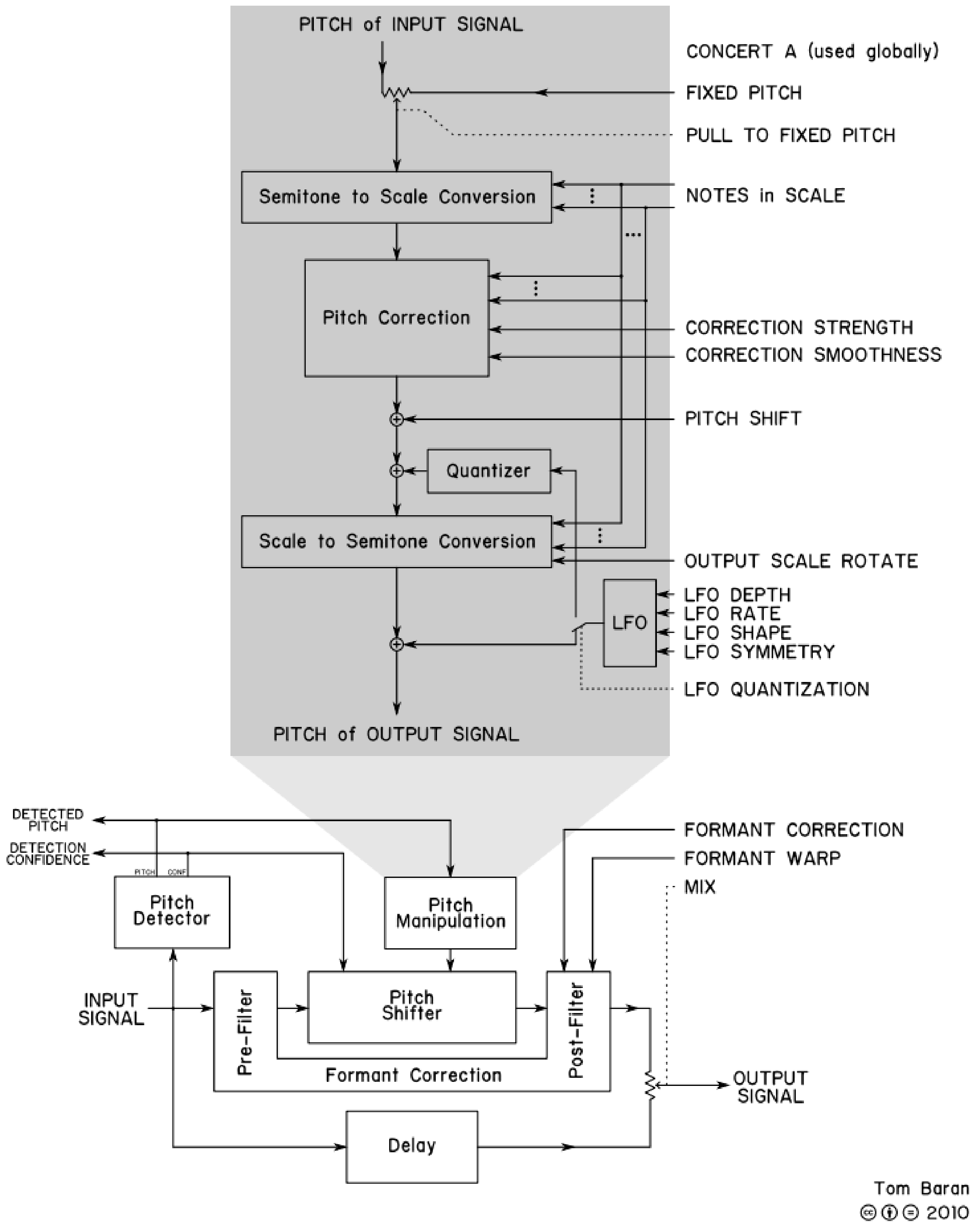
Equating equation 5.3 and 5.4 and simplifying, the following equation is obtained:

$$\log_2(3) = \frac{m+n}{n} \tag{5.5}$$

Since both n and m are elements of $\mathbb{N}_0$, the RHS is rational. The number $\log_2(3)$ is known to be irrational. This is a contradiction. Q.E.D.

[2]This proof was provided by Dr Neill Robertson of the UCT Mathematics department

B) Full AutoTalent Flow Diagram



C) Simple Overlap and Add Octave Code

All the code for the whole system is available on GitHub[3]

```
function scaledSample = getScaledSampleOLA(original,segmentSize,hopSize,sf,shiftRatioContour)
  % Get Sampling Points
  [windowStartPoints samplePointsX]  = getSamplePointsOLA(shiftRatioContour,hopSize,segmentSize);

  % Window at sampled points
  sampledSegments = getSegmentedSample(original,windowStartPoints,segmentSize);

  % Get the time stretched signal x
  stretchedX = overlapAndAdd(sampledSegments);

  % Interpolate to pitch shifted sample by arbitrary function
  scaledSample = interp1(stretchedX, [samplePointsX], 'spline');

  % Remove any NAN values that creeps in
  scaledSample(isnan(scaledSample)) = 0;
endfunction
```

Listing 5.1: getScaledSampleOLA.m

```
function [samplePointsOLA samplePointsX] = getSamplePoints(shiftRatio,hop,windowSize)
  % Initialize variables
  samplePointsX = 0;
  pointerOut = 1;
  samplePointsOLA(pointerOut) = 0;

  % Loop over each shift ratio point
  for (i = 1:length(shiftRatio)-2)

    % Fill sample points in STFT space until next shift ratio is reached
    while (samplePointsOLA(pointerOut) < i-1)
      samplePointsOLA(pointerOut+1) = samplePointsOLA(pointerOut) + 0.5/shiftRatio(i);

      % Add sample points in time domain every 2nd STFT point !!!50% overlap assumed!!!
      if (rem(pointerOut,2)==0)
        samplePointsX = [samplePointsX (samplePointsX(end) : shiftRatio(i) : (samplePointsX(end)+windowSize))];
      endif
      pointerOut = pointerOut + 1;
    endwhile
  endfor

  % Scale to whole song length
  samplePointsOLA = round(samplePointsOLA*windowSize);
endfunction
```

Listing 5.2: getSamplePointsOLA.m

[3]https://github.com/WalterSmuts/Pitch-Correction-Octave-Scripts

```
function segmentedSample = getSegmentedSample(original,windowStartPoints,
   segmentSize)
  segmentedSample = [];
  windowStartPoints(1) = 1;
  for (i = 1:length(windowStartPoints))
    start = windowStartPoints(i);
    stop  = windowStartPoints(i) + segmentSize-1);
    next  = original(start:stop).*hann(segmentSize);
    segmentedSample = [segmentedSample next];
  endfor
endfunction
```

Listing 5.3: getSegmentedSample.m

```
function stretchedX = overlapAndAdd(seg)
  % Constants
  windowSize = rows(seg);
  hopSize = windowSize/2;

  % Pre-allocate array
  stretchedX = zeros((rows(seg)*columns(seg)+rows(seg))/2,1);

  % Overlap and add
  begin = 1;
  for (i = 1:(columns(seg)))
    last = begin + windowSize-1;
    stretchedX(begin:last) = stretchedX(begin:last) .+ seg(:,i);
    begin = begin + hopSize;
  endfor
endfunction
```

Listing 5.4: overlapAndAdd.m