# 18-551 Final Project Report

# Karaoke Coach

**A pitch detection/pitch correction approach**

**Bryan Wang (bsw)**
**Seung Hyo Lee (seung)**
**Yedeh Ying (ypy)**

## TABLE OF CONTENTS

## 1.0 INTRODUCTION

As we enter into an era where people have more and more money, the entertainment industry and lifestyle has changed dramatically over the years. People go out and spend money frivolously on movies, games and several other forms of entertainment. Karaoke has been one of the fastest growing entertainment industries in this generation. Our project attempts to capture the qualities of the karaoke as well as enhance its use.

People have been flocking to karaoke bars in more recent times. The main reason of the large karaoke industry surge would be the want to show of their vocals to the newest songs, and just hanging out with friends. The machines used in the karaoke bars have expensive equipment, extensive music collections as well as several other amenities. When we approached this project, all of these could easily have been incorporated to home use with a computer, DSP chip, and speakers. The music collection can be easily downloaded off the web through mp3 sites or other forms of media. All this would be obtained for a significantly lower price, and become affordable to the average college student.

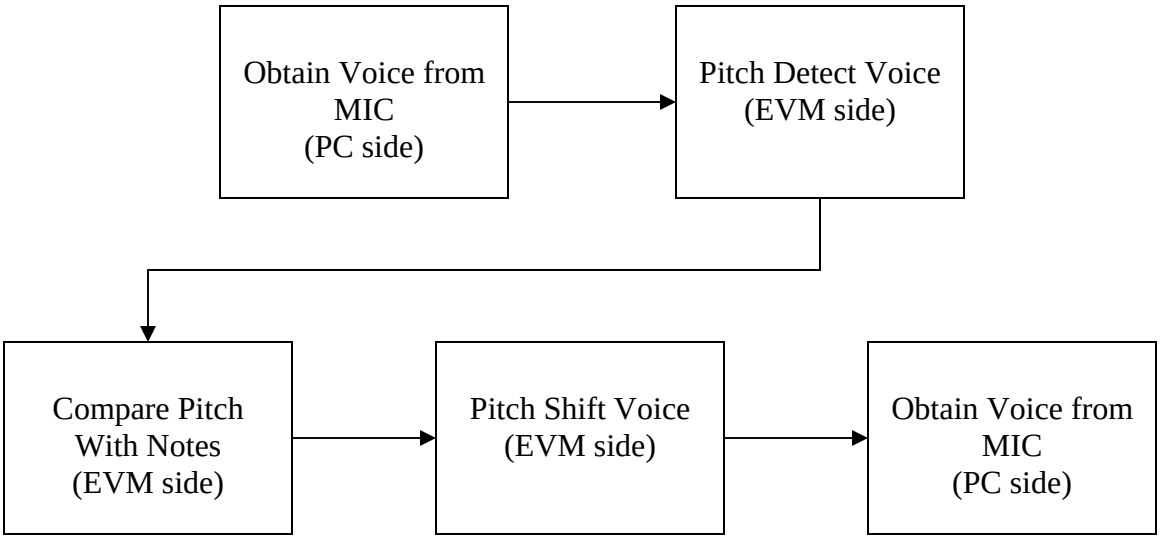
Not only would you be able to sing along with the songs that you download off the internet, the main focus of our project would be to allow the user to sing to the song of their choice but have it correct their voice so that they would sound as close as possible to the pitch of the original singer.

## 2.0 METHOD

In order to make this project possible there are two main concepts. The applications of pitch detection as well as pitch correction are some interesting concepts that we desired to explore. The user would be able to select a piece to sing to, and at the same time record their voice into a file. From that file, we would run it through the DSP chip and have it extract the pitches of the singers voice over a continuous sample. From the original music piece, we would also extract the pitches of the piece from a source file perhaps containing the singer's voice, or main notes of the piece. Comparing the two files with each other, every incorrect note will be flagged and corrected or shifted to the correct pitch. The output of the project would be a corrected voice of how the song 'should' have been sung, thus giving the singer a reference on how to sing for future songs.

Further research on polyphonic pitch detection is still very difficult to do, and hardly anyone has even had a successful algorithm that would allow us to detect the pitches of a highly intricate music piece. The easiest answer to our problem would be to just detect the singer's voice. Having the singer singing and a pitch detection algorithm running in the background, we would be able to accurately grab the pitches of the sung piece. Then having the an actually reference to the file where we would thus be able to compare the piece with the actual sung pitch.

## 2.1 SIGNAL FLOW




## 2.2 ALGORITHMS

The two algorithms we went with for this project was to actually implement the pitch detection using an autocorrelation window, and implementing the pitch shifting using a simple time stretching method and then re-sampling the new signal at the desired rate in order to give a new frequency.

### 2.2.1 Pre-Processing

The user first must obtain a sound file in which the format would be a wave file. This file will be the reference file that we will be able to obtain the pitches of the note to be sung at specific instances. A second wave file must be obtained from the users voice of the sung piece. This file will be the actual file that will be modified in order to have the user be able to tell how well she/he has sung to the music. A sampling rate of 11025 Hz was chosen since this would allow a great number of samples to be taken and hopefully be able to keep the quality of the sung samples.

### 2.2.2 Main Processing

#### 2.2.2.1 Pitch Detection

Unlike detecting the pitch of speech voice, detecting the pitch of the playing instruments is very hard because musical instruments have an extremely large frequency range, so the pitch detection algorithm must be able to process a very wide bandwidth. However, detecting the pitch of singing voice is even harder than that of instruments. It is because singing voice fluctuates so many times even within one time segment that it is hard to detect the 'correct' pitch that the singer is trying to sing. Instruments raise the pitch at the very beginning and then slowly fade away until the next segment of 'attack' follows and get mixed with the previous segment. In other words, the sound of instruments are very harmonic and less variant whereas the singing voice is less harmonic and has many attacks in the same amount of time.

There is handful of existing pitch-detecting algorithms. These algorithms are categorized in two. The first one is spectral-domain based algorithms like Cepstrum, Maximum Likelihood, and Autocorrelation methods. The other one is time-domain based. We decided to go with Autocorrelation method because it was proven to work in some previous projects in the course. Our first attempt was to find any prior working source or algorithm on signal processing of autocorrelation on the Internet. After searching the website for a while, we found out that Mississippi State University had a handful of C++ source on Autocorrelation including autocorrelation.h which was functions for autocorrelation [1]. However, we could not port it onto our EVM board for some unknown reason. It could have been that some of the functions were not compatible with C compilers of EVM, but we were not able to pin-point and resolve the problems. So rather than tinkering around with unreliable resources, we decided to write our own

autocorrelation program based on the labs we did in the early part of class.

Our pitch detection algorithm has two main parts: slicing the signal into fixed sliding window size and then processes each window with multiplications and summation with sliding window size of 100 and 8 KHz of sampling rate. We placed these two parts in FOR loops and assigned our sliding window size to be 100. So for every 100 samples, multiplied samples and then added together. We kept on moving the windows until the end of the signal. Autocorrelation is known to be able to detect peaks even in the noisy data so we thought this would enable us to detect the correct pitch. For detecting peaks in the autocorrelation, as we looped the process, we looked for a point where the point changed from positive to negative. In order to check that the detected peak was the ‘meaningful’ peak that we were looking for, we also checked peaks nearby samples from the given peak at the same time. If the given peak was lower than the peaks around it, the peak was not counted as one of our ‘meaningful’ peaks to calculate the fundamental frequencies. [2]

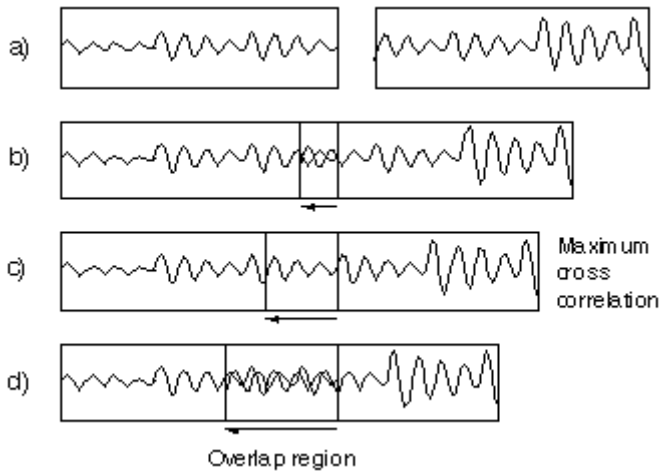
Our first attempt was to increase the window size and the sampling rate. The window size was incremented by 100 to the size of 200, but the detection of speech voice was not good enough. Even our recorded instrument (piano) playing at one tone was not detected well. Considering that the instrument has much less vibration than human voice, we were expecting to have much better pitch detection, but the resulted pitch detection was varied from as little as 5 Hz to over 200 Hz. After boosting up the window size to 400, and the sampling rate to 11.574 KHz, we were able to detect the pitches reasonably well. With larger window size, the peaks in the autocorrelation were much bigger and less while the peaks with smaller window size had many low

peaks – which could have caused bad pitch detection. Now the detected pitch variance was less than 10 Hz except a few extreme cases. However, this was not as consistent as we expected. We were able to detect the pitches for some wave files but some wave files with similar conditions did not give us the right pitch rating.

#### 2.2.2.2 Pitch Correction

Our method of pitch detection was a simple way that Professor Sullivan told us should work. It would be to stretch the sample in the time domain and then to actually resample that sample at the new desired frequency. So assuming that we had a ratio of 'x' between the frequencies, we would time stretch the given sample to 'x' times the original size, and then resample the new signal at a new rate 'x' times the normal sampling rate. [3]

A very efficient method of time stretching our samples would be to use the synchronized overlap-add method. This is very simple computationally, and if it had been possible this algorithm would have been ideal for a real time application. The method actually consists of shifting the beginning of a new speech segment over the end of a preceding segment in order to find the point of the highest cross correlation. At this point the points are overlapped and averaged together. In this way we are able to preserve the time-dependant pitch, magnitude and phase of the signal. The graphic below may actually give a better representation of the algorithm than stated in words.




In order to smooth the signals in between segments we used a windowing function in order to cross fade between the two segments. This would help to produce fewer artifacts than regular sampling techniques.

From the new stretched time sample, would we actually be able to extract a new pitch depending on how we now resample the signal. Hopefully this would be able to provide a robust enough algorithm to actually shift the voice of the user up or down, with out having a problem with intelligibility of the actually spoken words. Intelligibility meaning that the sung word still actually sounded like the original singer's voice and having it still be a new pitch.

### 2.2.3 Post-Processing

After the signals have been pitch detected and pitch corrected, the outputted data is put into a text file. The actual format of this text file is a table of floats containing the frequency of the note at each sample. Since there are From there on the PC side, we will be able to read this into a wave file format and output the new file through the sound card.

## 3.0 RESULTS & CONCLUSIONS

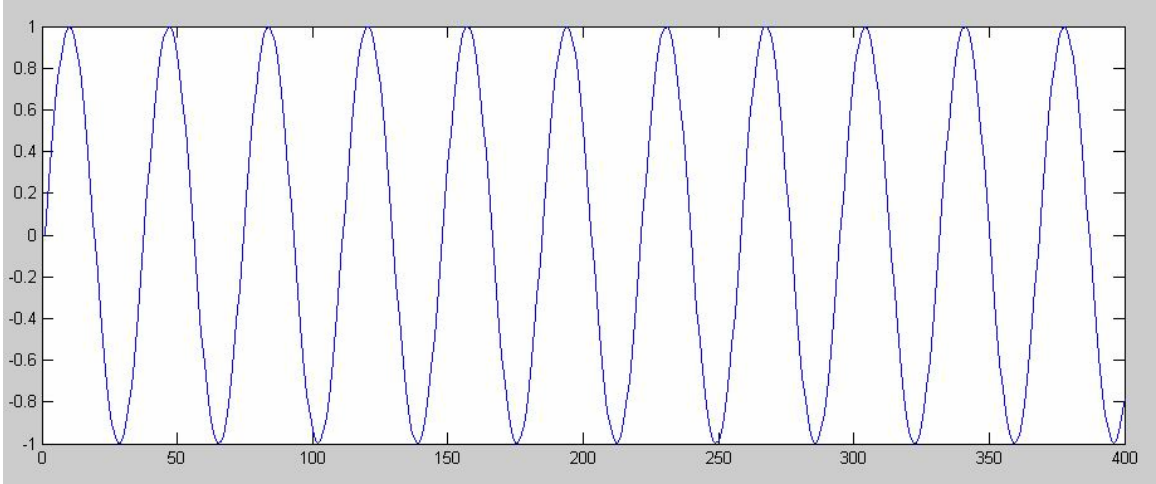
In order to see if our algorithm actually worked in the real settings, we had to test it in a sequential order. Each level of testing actually had to be passed before we were able to proceed on to the next set of test data. For each level, we would process signals of increasing difficulty. For example we started off with a simple sinusoidal curve that would produce a constant tone. From their we hoped to progress to a singer actually singing a musical piece.

### 3.1 SINUSOIDAL TONE

For this part of the testing, we just read in files that contained values for a 300 Hz sine wave, 605 Hz sine wave, and a succession of sine waves at 200 Hz, 300 Hz, and 450 Hz. These three test cases were needed to see if our pitch detection actually worked on a simple signal. We expected decent results with these signals.

#### 3.1.1 300 Hz SINE WAVE

The detection of this signal actually was a lot better than we had imagined. From the results of the output, we detected the sine wave within 1 Hz of the actual signal. From that detected signal we tried to shift the signal down to 293 Hz. Looking at those results, we were able to see that it actually shifted the signal quite well within 2 Hz of the desired pitch. Below is one window of the sine curve.



### 3.1.2 605 Hz SINE WAVE

This signal was to show the detection and up-shifting of a signal, whereas the previous signal was to show how well our algorithm would shift it down. The desired shift was to make the signal go up to a frequency of 621 Hz. This also was quite accurate and the shifted signal was within a range of 2 Hz of the desired pitch.

### 3.1.3 200 Hz, 300 Hz , and 450 SINE WAVES

This case would actually be for us to see if our pitch detection algorithm would be able to detect a series of notes. Having this would be the first major step in order to detect a multitude of frequencies. We were hoping that this would actually be able to simulate a wave file in which the singer was able to sing a multitude of different pitches and have it corrected to the desired pitch as well.

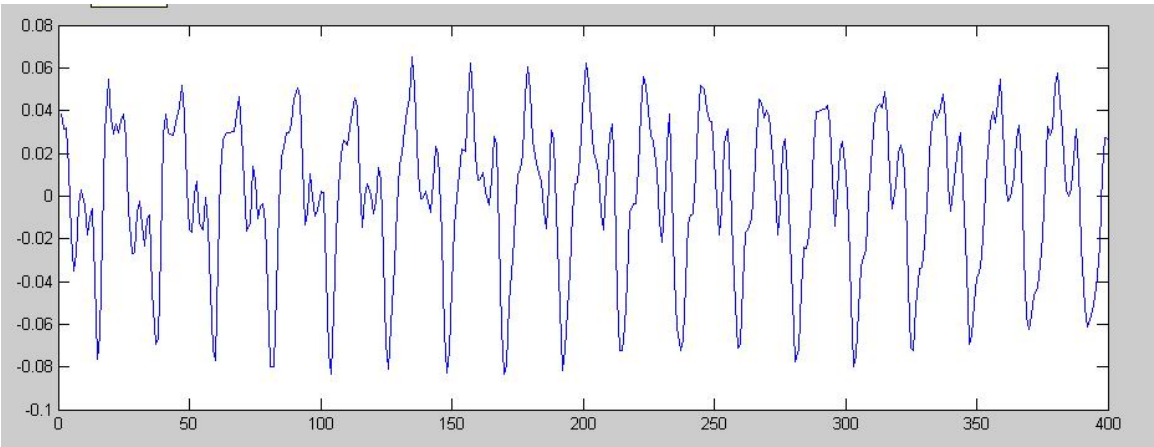
As we had seen earlier, we were able to detect single tones very well and shift them to whatever desired frequency with little problem. When it came to the detection of the signal with 3 different tones the pitch detection algorithm was just as flawless. The pitch was detected within 1 Hz of the given frequencies through out the signal. The pitch shifting was almost as good as well. The qualitative values of the pitch that it was shifted was actually within 2 Hz as well, but from the listening stand point, the actual sound of the signal was a little bit clipped in between the changing of the tones.

After all these fixing and updating, the final version of our pitch detection algorithm was able to handle almost all the incoming wave files. Now it detected the various sine waves within the variance of 1 Hz, and it was also able to detect stair-step series of sine waves without a problem. With the stair-step series of sine waves, we put together 3 sine waves with each step

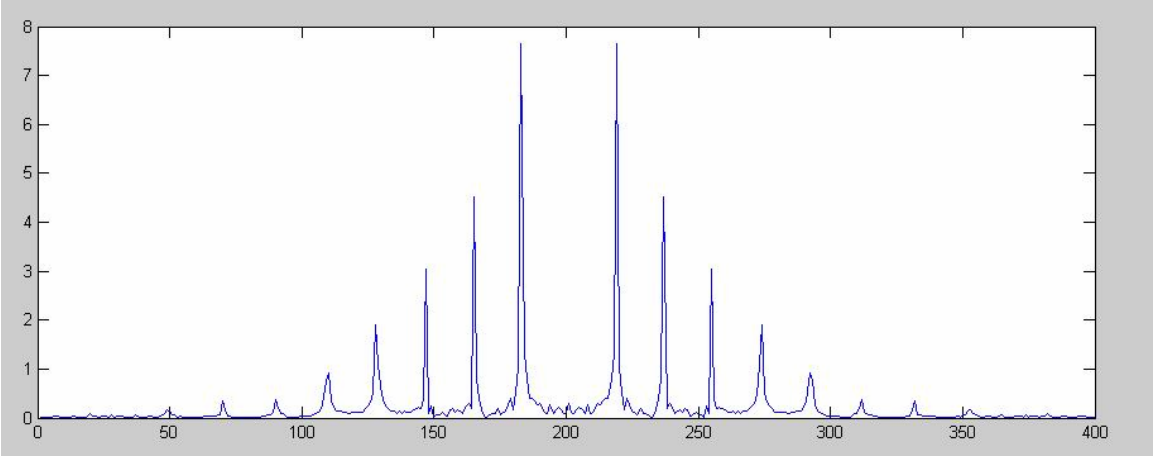
increased the frequency 150 Hz, and the program detected each sign waves within 1 Hz of error.

### 3.2 PIANO

The sample for this was actually a piano playing a few notes and recorded as a wave file. We knew that real life signals would not be as perfect as sine waves all mashed together, so we hoped that our algorithm would have been robust enough to handle the mixing of pitches in between notes of a piano. The played sampled was actually a grand piano playing two notes at B and A. The expected frequencies for these two notes would be at 493.9 Hz and 440 Hz respectively.



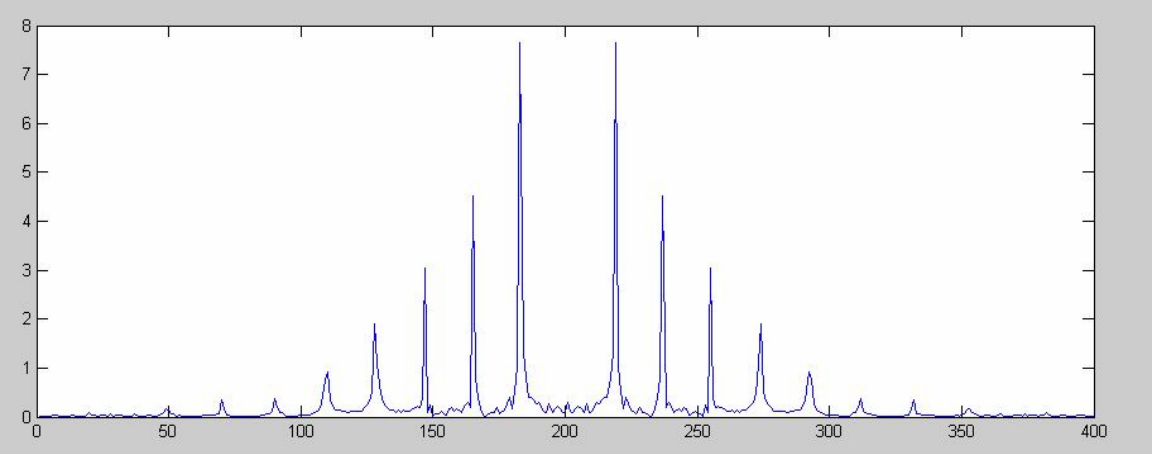

One window of piano wave




Spectrum of the piano wave

The detected pitches were not as close as before, but these errors could be attributed to noise, as well as the intermixing of the different notes. For example when the B notes was played, the A noted would be played shortly there after and there would be a mixing of the two waves together before the B note actually faded out. The detected pitches were within 3.02 Hz of the actual notes, which was pretty good in our opinion. When we attempted to shift the pitch back up to the normal pitch from eh detected values, the variance got higher, but still was within 5 Hz of the desired frequencies. The actual output of the wave file no discernable difference could be made from the wave files that we outputted in terms of pitch, but that was expected since the piano should have been playing the notes at either exactly the right note or at least near it, assuming that the piano itself was in tune. The only noticeable thing was when we hear the clipping in between the changing of the notes; we were unable to determine the actual reason for this.

| Pitch Detected (Hz) | Pitch Detected (Hz) |
|---|---|
| pitch before shift : 510.416687 | pitch after shift : 491.212891 |
| pitch before shift : 496.125031 | pitch after shift : 491.212891 |
| pitch before shift : 493.656738 | pitch after shift : 496.125031 |
| pitch before shift : 493.656738 | pitch after shift : 493.656738 |
| pitch before shift : 491.212891 | pitch after shift : 498.618103 |
| pitch before shift : 488.793121 | pitch after shift : 498.618103 |
| pitch before shift : 491.212891 | pitch after shift : 496.125031 |
| pitch before shift : 488.793121 | pitch after shift : 501.136383 |
| pitch before shift : 493.656738 | pitch after shift : 496.125031 |
| pitch before shift : 493.656738 | pitch after shift : 493.656738 |
| pitch before shift : 493.656738 | pitch after shift : 493.656738 |
| pitch before shift : 496.125031 | pitch after shift : 491.212891 |
| pitch before shift : 491.212891 | pitch after shift : 493.656738 |
| pitch before shift : 496.125031 | pitch after shift : 491.212891 |
| pitch before shift : 369.966431 | pitch after shift : 508.846161 |
| pitch before shift : 316.810333 | pitch after shift : 434.482788 |
| pitch before shift : 459.375000 | pitch after shift : 446.959473 |
| pitch before shift : 330.750031 | pitch after shift : 441.000031 |
| pitch before shift : 443.216095 | pitch after shift : 438.805969 |
| pitch before shift : 443.216095 | pitch after shift : 438.805969 |
| pitch before shift : 443.216095 | pitch after shift : 437.500000 |
| pitch before shift : 329.104492 | pitch after shift : 330.750031 |
| pitch before shift : 441.000031 | pitch after shift : 441.000031 |




Spectrum of piano wave

## 3.3 VOICE

For this part of the testing we were hoping to see what our algorithm would be like if we actually tested it with a human voice. The first sample was actually of a person singing and holding a specific note. From that we would be able to tell if our algorithm actually worked with voice even with all the background noise. After that we were hoping to test the algorithm of with an actual sung piece.

### 3.3.1 SUNG NOTE

The test file was made by having a person sing a know note with known frequency using a low quality microphone. We hoped that the noise from the background/wiring would be picked up along with the actual sung note. We had the person sing a high C note. The corresponding frequency for this note would be at 523.3 Hz.

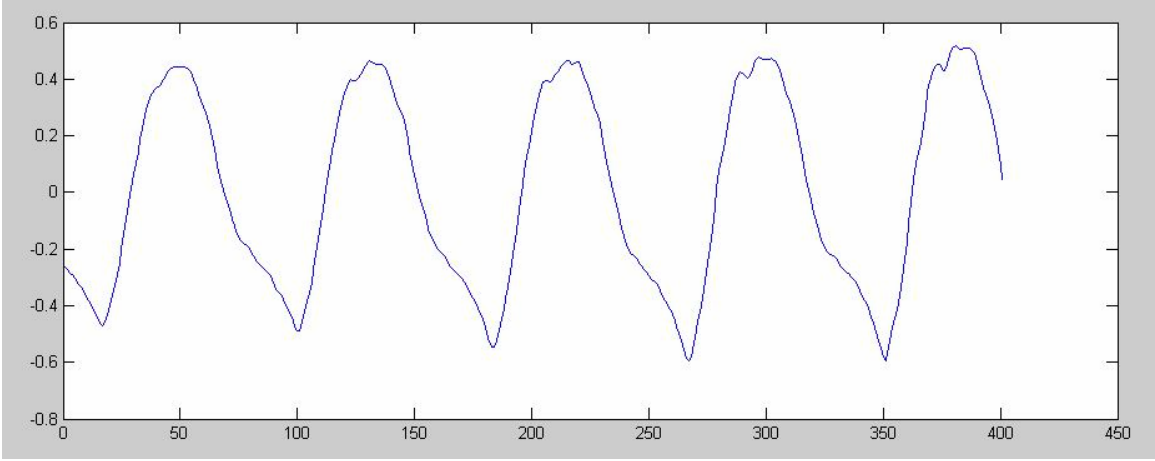
| Pitch Detected (Hz) | Pitch Detected (Hz) |
|---|---|
| pitch before shift : 511.469086 | pitch after shift : 522.511841 |
| pitch before shift : 514.119141 | pitch after shift : 520.047180 |
| pitch before shift : 511.469086 | pitch after shift : 525.000000 |
| pitch before shift : 514.119141 | pitch after shift : 520.047180 |
| pitch before shift : 511.469086 | pitch after shift : 525.000000 |
| pitch before shift : 514.119141 | pitch after shift : 522.511841 |
| pitch before shift : 516.796875 | pitch after shift : 522.511841 |
| pitch before shift : 511.469086 | pitch after shift : 522.511841 |
| pitch before shift : 511.469086 | pitch after shift : 525.000000 |
| pitch before shift : 508.846161 | pitch after shift : 522.511841 |
| pitch before shift : 508.846161 | pitch after shift : 525.000000 |
| pitch before shift : 511.469086 | pitch after shift : 522.511841 |
| pitch before shift : 508.846161 | pitch after shift : 525.000000 |
| pitch before shift : 511.469086 | pitch after shift : 522.511841 |
| pitch before shift : 511.469086 | pitch after shift : 522.511841 |
| pitch before shift : 511.469086 | pitch after shift : 522.511841 |
| pitch before shift : 511.469086 | pitch after shift : 522.511841 |
| pitch before shift : 511.469086 | pitch after shift : 522.511841 |
| pitch before shift : 508.846161 | pitch after shift : 522.511841 |
| pitch before shift : 508.846161 | pitch after shift : 525.000000 |
| pitch before shift : 511.469086 | pitch after shift : 522.511841 |

Our pitch detection algorithm detected this singer's voice at an average of 510 Hz. We took this to be within the reasonable range of the desired pitch, as the variance of this detected frequency was around 4 Hz. When we applied the pitch correction of the voice, sine they were off key slightly, it corrected it to the desired 523.3 Hz frequency with greater accuracy at a variance of 2 Hz.

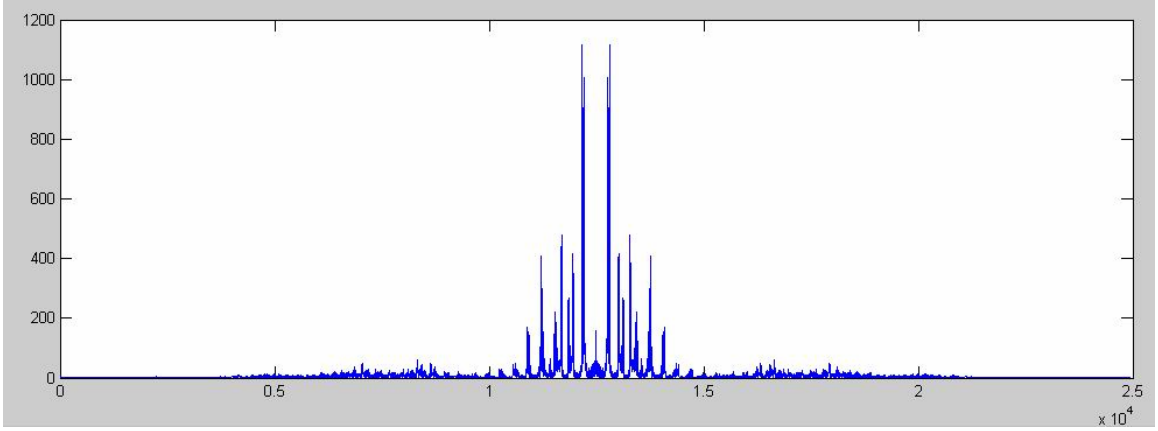
### 3.3.2 SUNG WORD

We had a user sing into the microphone a song that was well known. From that we analyzed the results from a clip of that where one of the words was analyzed with our pitch detection and shifting algorithms. Since we had first tested our algorithm with code-generated sine wave, we were confident that it was working properly. The results were satisfactory given that the resulted pitch was less than 1 Hz different. So we decided to move on with the voice. Using in-lab microphone and soundcard, we recorded our own speech voice and saved it as wave

file. When we tried to run the pitch detecting program with the recorded voice, we found out that our algorithm was not able to detect the pitch right. At first we thought that it could be caused by the fluctuating voice even if it was simple speech voice. After trying several times with different voice files and continuous failures of detecting pitches (inconsistent pitch detection), we concluded that our algorithm was not robust enough to handle speech voice.




One window of sung word




Spectrum of the sung word

The pitch detection at this point seemed to break down when there was a syllable being pronounced through out the song. At any point

there was a change in the syllables of the word, the pitch detection of that word just dropped off with large changes in frequencies and it was not able to track it as well as it had been previously. However it seemed as the voice stabilized, the pitch detection worked again. For example in the word ‘Mary’, the first part of the word being pronounced would be ‘Ma’. The pitch detection algorithm actually worked for this part of the signal. When the user proceeded to sign the ‘ry’ part of the word, the transition between the symbols produced a lot of garbage in terms of pitch. But as the time went on, and the syllable ‘ry’ was held, the pitch detection stabilized again, and a correct pitch was detected. The pitch correction part of this algorithm was not implemented since the pitch of the sung words could not be detected accurately enough to give the user a accurate correction of the desired pitch.

| Pitch Detected (Hz) | Pitch Detected (Hz) |
|---|---|
| pitch before shift : 136.111115 | pitch after shift : 136.111115 |
| pitch before shift : 131.250000 | pitch after shift : 131.250000 |
| pitch before shift : 131.250000 | pitch after shift : 131.250000 |
| pitch before shift : 141.346161 | pitch after shift : 141.346161 |
| pitch before shift : 136.111115 | pitch after shift : 136.111115 |
| pitch before shift : 137.812500 | pitch after shift : 137.812500 |
| pitch before shift : 137.812500 | pitch after shift : 137.812500 |
| pitch before shift : 137.812500 | pitch after shift : 137.812500 |
| pitch before shift : 139.556961 | pitch after shift : 139.556961 |
| pitch before shift : 142.258072 | pitch after shift : 142.258072 |
| pitch before shift : 141.346161 | pitch after shift : 141.346161 |
| pitch before shift : 408.333344 | pitch after shift : 214.077682 |
| pitch before shift : 255.208344 | pitch after shift : 146.026489 |
| pitch before shift : 216.176483 | pitch after shift : 416.037750 |
| pitch before shift : 318.641632 | pitch after shift : 277.358490 |
| pitch before shift : 304.558014 | pitch after shift : 258.802826 |
| pitch before shift : 139.556961 | pitch after shift : 139.556961 |
| pitch before shift : 117.287239 | pitch after shift : 117.287239 |
| pitch before shift : 237.096786 | pitch after shift : 264.071869 |
| pitch before shift : 271.106567 | pitch after shift : 243.646408 |
| pitch before shift : 117.914444 | pitch after shift : 117.914444 |
| pitch before shift : 117.287239 | pitch after shift : 117.287239 |
| pitch before shift : 237.096786 | pitch after shift : 158.633087 |
| pitch before shift : 351.861725 | pitch after shift : 266.733887 |
| pitch before shift : 117.287239 | pitch after shift : 117.287239 |
| pitch before shift : 116.052635 | pitch after shift : 116.052635 |
| pitch before shift : 234.574478 | pitch after shift : 206.718750 |
| pitch before shift : 266.733887 | pitch after shift : 260.433075 |

Looking at this table, we can see from approximately where the pronunciation of the first syllable of the word is being sung which is the first 10 or so cells. As the second syllable of the word is being pronounced the pitch detection just varies too much for any change to be made.

## 4.0 MEMORY & PAGING RESULTS

Originally we had though that we would have to load the entire clip of the song into the buffer requiring a lot of memory to be transferred too and from the DSP chip. That would also hinder the speed of the actually process since the communication would happen on board. However we were able to keep everything on chip since we just had the EVM communicate with the PC in order to grab the windows of data for each pitch detection frame.

So basically our whole program for the EVM side fit onto on chip memory and actually only occupied about 10 K. On the PC side, our code was whatever about the same as one of the labs where we were trying to get the board to communicate with the PC.

At the time we were also worried about having our code run really slowly. We were auto-correlating 400 samples for indefinite lengths of songs and music. With each song being sampled at 11025Hz, which would be a lot of windows to be correlating not to mention the math that would have to be involved. But once we ran it, it ran surprisingly quick and we actually didn't worry too much about it. Before optimization our code ran at 1,102,597 cycles for one correlated window. This also included pitch shifting and re-detecting the new pitch shifted window.

We then set out to do the loop unrolling and have it parallelize the autocorrelation. When we finished doing this, we were actually able to get the speed to improve by a little bit. The final number of cycles for our project was 724,149.

## 5.0 REFERENCES


1. Mississippi State University – Code for autocorrelation methods. (http://www.isip.msstate.edu/projects/speech/software/tutorials/tutorial_archive/c++/html/page_01.html)

2. Pitch Period Estimation Using Auto-Correlation pg 150 – 154 *Digital Processing of Speech Signals.*

3. Professor Tom Sullivan, ECE Dept Carnegie Mellon University.

4. http://kt-lab.ics.nitech.ac.jp/~tokuda/SPTK/ just for learning about speech signals, some reference code was looked at.


Some previous projects were that covered Pitch detection was slightly different from ours, but their reports were helpful in helping us understand exactly what had to be done for the project. We believe ours is the first to actually try to pitch detect and Pitch shift it to a desired pitch.