Python Implementation of the Auto-Tune Patent

Eric D. Stevens

Abstract— **This paper consists of a detailed description of (1) the Auto-Tune patent, (2) an overview of a Python programming language implementation of the patents key principles, and (3) a review of the performance of these algorithms. Since the software and the audio files cannot be included in this file, the will accompany this document.**

I. INTRODUCTION

At the time of the early 90's there was no way to automatically force an out of tune singer's voice into tune. At that time, Harold Hildebrand worked for Exxon Mobil as an electrical engineer, applying digital signal processing skills to geology in an effort to help Exxon use ground penetrating signals to find oil. Hildebrand was also an accomplished musician. One day, while out to lunch with a group of his musical colleagues, he queried the groups opinion about what the music industry could need that an electrical engineer might provide. One of his friends jokingly proclaimed a desire to have a tool that would make her sing in tune all the time. The group laughed it off but the idea was implanted in Hildebrand's head.

Within a few years Hildebrand had applied the same techniques he was applying working with Exxon to the effort of changing a singers voice, in real time, to be perfectly tune. He did this by deriving a novel way to utilize the auto-correlation function to track the fundamental frequency of a singers voice. Armed with the powerful equations, he designed an integrated system that consisted of a micro-controller loaded with a program capable of executing the mathematics he had computed. The system was dubbed Auto-Tune. The patent was filed in 1998 to the company of Hildebrand's founding, Antares Audio Technologies. Since then, it's various utility has been a staple of both the recording industry and modern music culture.

This paper should serve as an in depth exploration of how Auto-Tune uses the auto-correlation function to quickly and accurately obtain the fundamental frequency of a voice signal. There will also be a description of how capturing the fundamental frequency of the voice signal fits into the system as a whole. Finally, a Python implementation of the key concepts of Auto-Tune will be presented and its performance will be analyzed. All equations and works presented in this paper and works are directly derived from the 1998 Auto-Tune patent itself.

II. AUTO-TUNE SYSTEM OVERVIEW

The Auto-Tune system was originally designed to be a standalone computational tool for recording artist to use in studio. It is a micro controller with the program memory holding the instructions to continuously loop through one of three states at any given time. The three states are the operating modes which are described below.

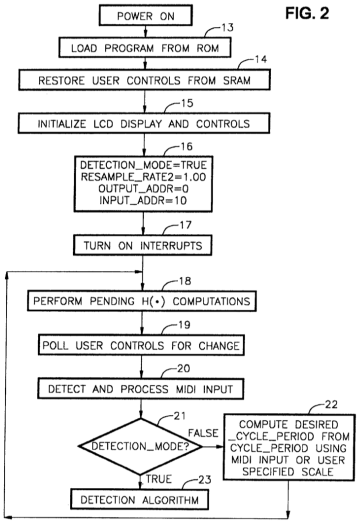

Fig. 1. Auto-Tune patent figure 2, the full program.

A. Pitch Detection Mode

Pitch detection mode is the start up mode of the system. When the system is in pitch detection mode it means that there is no information about the fundamental frequency of the previous time step. This is the most computationally expensive mode of operation. It requires the search across the entire range of possible fundamental frequency possibilities of a singer (2,756 Hz down to 50 Hz in this system) until if finds one that satisfies the criteria of the fundamental frequency. Once a fundamental frequency has been identified from within the pitch detection state, the pitch detection flag set to false and the pitch tracking state is entered. The mathematics behind pitch detection will be discussed in a following section.

B. Pitch Tracking

Pitch tracking assumes that there is knowledge if the fundamental frequency in the time-step prior to the one the system is currently analyzing. Making this assumption allows for a drastic reduction in the use of computational resources by assuming that at the current time point the fundamental frequency will be fairly similar to the fundamental frequency of the previous time point. This paper will not go into detail about how the pitch tracking state works because the

mathematics are for the most part identical to those of the pitch detection state.

C. Pitch Correction

The pitch correction state of the system assumes knowledge of the fundamental frequency of the current time step and uses that information to re-sample the incoming audio data of the current time step in order to alter it's playback pitch to dome desired fundamental frequency. The system as described in the original patent allows for a number of was to have the user input the desired frequency such as pulling the input to the closest note in a given musical scale or altering the input signal to match the frequency that is specified by the concurrent input from a midi controller.

III. METHODS

The key innovation of the system was the manipulation of the auto-correlation function to enable rapid and continuous detection of the fundamental frequency in a voice signal. This section is a detailed derivation of the mathematical techniques developed in this effort. Please note that an effort has been made to lift all of the following equations directly from the original patent without alteration in order to preserve the original idea.

A. Auto-correlation

The patent uses an un-normalized version of the auto-correlation function Φ.

$$\Phi_L(n) = \sum_{j=0}^{L} x_j x_{j-n} \tag{1}$$

Here x is assumed to be periodic with a fundamental period L. This auto-correlation function is also periodic. It is important to notice here that if n, referred to as the 'lag', is equal to 0 then we just have x^2. The periodic nature of the equation means that if $n = kL$, where k is any integer, this will equal the same thing as the function with a lag of 0. The auto-correlation is not generally used to track periodicity because of its computational expense. The key claim of the Auto-Tune patent was to speed up the computation of the auto-correlation.

At a time i given a sequence of a sampled periodic waveform with period L, x_j, the auto-correlation of the function can be expressed as,

$$\Phi_{i,L}(n) = \sum_{j=i-L-1}^{i} x_j x_{j-n} \tag{2}$$

B. $E(.)$ and $H(.)$ equations

In order to reduce the computational complexity involved in calculation of the auto-correlation, two related equations are derived, $E_i(L)$ and $H_i(L)$.

$E_i(L)$ is the auto-correlation of two cycles of a periodic waveform evaluated at 0 defined as dependent on the guess of the period of the cycle L.

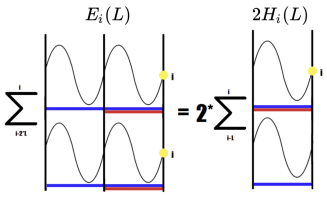

Fig. 2. In the situation where L is the true period of the signal, $E_i(L) = 2H_i(L)$

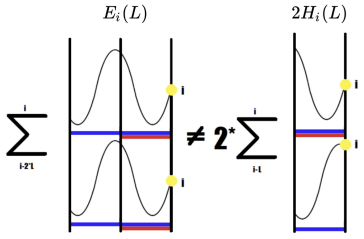

Fig. 3. In the situation where L is not the true period of the signal, $E_i(L) \neq 2H_i(L)$

$$E_i(L) = \Phi_{i,2L}(0) = \sum_{j=0}^{2L} x_j^2 \tag{3}$$

The $H_i(L)$ is the auto-correlation of a single cycle of the periodic waveform evaluated and the guess of the period L. Again, this function is defined with the guess of the period L as the dependent variable.

$$H_i(L) = \Phi_{i,L}(L) = \sum_{j=0}^{L} x_j x_{j-L} \tag{4}$$

At any given time point i these two functions allow us to determine the period of a periodic waveform by evaluating the following equation for L,

$$E_i(L) - 2H_i(L) = 0 \tag{5}$$

Equation 5 states that when L is set to the true period of the periodic waveform x_j it becomes true that equation 3 is equal to two times equation 2. In other words, if L is the true period of repetition of x_i then the square of two L is the square of two periods which is equal to two times the of the previous L times the previous $2L$ through L. This concept is more easily understood visually in ***Fig. 2*** and ***Fig. 3***.

In the actual software running on the system, the fundamental frequency of the signal being analyzed at a given time point is very unlikely to have a period that matches up perfect with the integer sample rate, or for that matter, to have a signal that is truly periodic in the first place. Therefore, when analyzing the real sampled data, L is said to be the fundamental period of the waveform if the following equation is satisfied,

$$E_i(L) - 2H_i(L) \leq \varepsilon E_i(L) \tag{6}$$

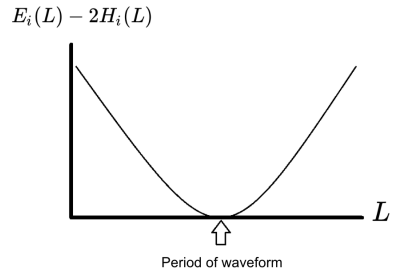

Fig. 4. Using polynomial approximation, the true floating point period of the the waveform can be inferred.

Where ε can range from 0.00 to 0.40. Once a value that satisfies equation 6 has been found, a range of L values surrounding the satisfying L values are used to preform a polynomial approximation of $E_i(L) - 2H_i(L)$ in order to find the local minimum ***Fig. 4***. This floating point minimum represents the true period of the waveform.

C. Speeding up successive $E_i(L)$ and $H_i(L)$ calculations

Calculating $E_i(L)$ and $H_i(L)$ is very efficient when the need to be performed in succession and this fact is the reason why this algorithm has such a performance advantage. Since at each time step we only move forward one sample, we do not need to repeat most of the multiply and adds that were performed for the initial calculation in the. The following equations demonstrate this,

$$E_i(L) = E_{i-1} + x_i^2 - x_{i-2L}^2 \tag{7}$$

$$H_i(L) = H_{i-1} + x_i x_{i-L} - x_{i-L} x_{i-2L} \tag{8}$$

Equations 7 and 8 are derived from equations 3 and 4 assuming knowledge of the previous E and H. In this way, we reduce the number of multiply adds that need to be done for each value of L from $3L$ to just 4.

IV. RESULTS

The implementation of the Auto-Tune system that accompanies this paper is incomplete when compared to the patented system as a whole. Nonetheless, the provided implementation does provide a complete end-to-end system for altering the fundamental frequency of human voice. While a variety of decisions were made to shrink the scope of the project and therefore decrease the time performance and fidelity well below what would be required of a production system, the use of the auto-correlation for pitch detection as described above was successful.

A. Differences between patent and Python implementation

There were a number of concessions made in order to fit this project into workload it was intended to fill.

The pitch state was ignored, resulting in the need to perform pitch detection at every time step. This brings rate at which the fundamental frequency can be captured far slower than real time. It probably takes somewhere around 10 seconds of pitch detection for every one second of audio as the system is. The lack of pitch tracking also made decreases the fidelity of the the f_o detection system, thus resulting outputs the will skip around in frequency for slots of time.

That being said, due to the required length of this paper there are many aspects of the system that are not even covered in this paper that were implemented by hand. The calculation of the re-sample rate is not discussed in this paper but is implemented by hand. The reassembly of wave forms to keep phone time fidelity was not discussed in this paper but.

The code base is very small for how much it accomplishes the reader is encouraged to view the code in system code in 'AutoTune.py' and run the demo in Jupyter with the 'Demo.ipynb' file.

B. System description

The 'AutoTune.py' file contains a class called AutoTune(). In this class there are a number of member variables:

- wav - the original .wav file sent into the class.
- $down\_wav$ - a 1/8 decimation of the original .wav file used for calculation of the of E and H
- Ei/Hi_table - $(110xlen(down\_wav))$ sized table that hold the calculation of E and H fore each time point in $down\_wav$.
- sub_table - table of the same size holding $E - 2H$ at any given time point and L value.
- $real\_freq$ - array the same size as the input .wav file that holds the calculated true frequency at the time coresponding to the index it shares with the input wav file.

When a class object is initialized, the constructor automatically performs the pitch detection for all time points. When initialization is complete that means you have a and array of calculated fundamental frequencies sitting in the $real\_freq$ member variable.

After this initialization has been performed, the user can alter the waves fundamental frequency over time using the $build_output()$ member function. Please see the 'Demo.ipynb' Jupyter notebook for examples of how this is done. Once the output has been build it will be sitting in the member variable $output$ and can be played through the users speakers using the member function $self.play()$.

C. Performance of the system

The way that we can evaluate performance of the pitch detection system is by feeding it wave forms of increasing complexity or noise and looking to see how it performs on its approximation of the current pitch. As mentioned in the previous section, upon initialization of an object the pitch detection is performed. For this results section a few pairs of graphs will be presented with one being a slice of a wave to demonstrate the complexity of that wave, and the other being the pitch tracking output to see the kind of noise that is seen in it.

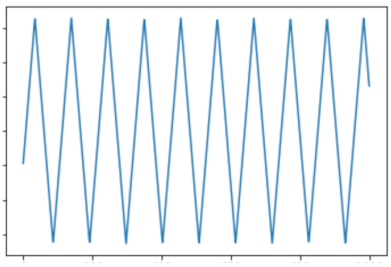

Fig. 5. Slice of a machine generated triangle wave up-down sweep.

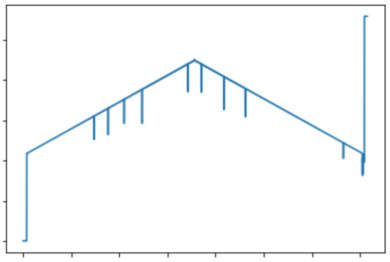

Fig. 6. Pitch detection perfectly captures frequency.

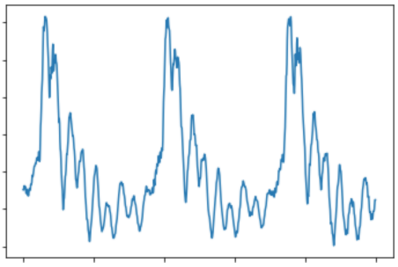

Fig. 7. Slice of single note voice signal.

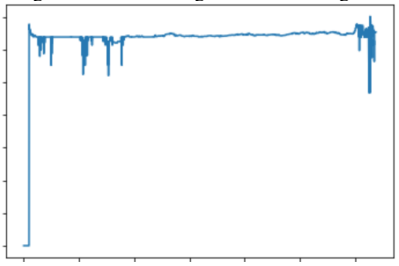

Fig. 8. Pitch detection able to capture almost all of it with a little noise in the beginning and end.

D. Perfect wave form sweep

The first test the system was put through was a perfect triangle wave. The results can be seen in figures 5 and 6.

E. Simple voice signal

The next test was to look at what the results of tracking would be with the most simple voice signal. This attempt at a monotone not being hit by a male voice resulted in figures 7 and 8.

F. Complex voice signal

Finally a whole line of a song was attempted. While the results are visually rough, it is very surprising how well the audio comes out. These results can be seen in figures 9 and 10.

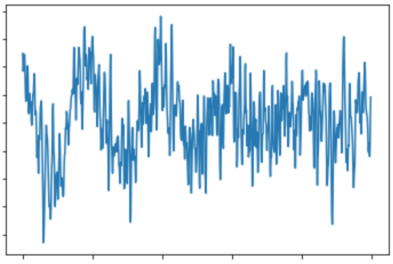

Fig. 9. Slice form acapella recording of Tom's Diner by Suzanne Vega.

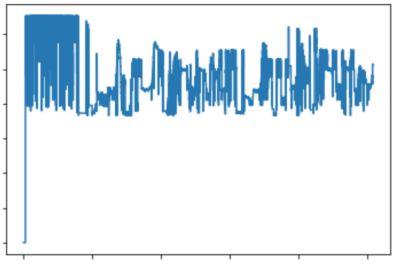

Fig. 10. Very rough frequency tracking but on average notes are captured.

V. Discussion

A. Audio performance

While it is difficult to see the performance of this system from the images above, when you hear the result you will notice that the level of signal noise is less than what the images might have you believe. Please be sure to check out the 'Demo.ipynb' file included with this submission if you care to hear the audio that resulted from these experiments or to play with your own audio files.

B. Future works

There is much that can be done with this system. This implementation should be viewed as a very slow, very incomplete prototype to show that the equations are capable of doing what they were intended to do. The parts of the system that were left out for this project should be added in.

After the system has been fully implemented, it should be ported to a lower level language so that the operations can be handled in an efficient in order to achieve the intended goal of real-time processing. For audio applications, the whole project, once working properly, should be written in non-blocking C++.

References

[1] Hildebrand H.A.,(1999), *US5973252A*, https://patents.google.com/patent/US5973252, Pitch detection and intonation correction apparatus and method