AUTOTUNE IMPLEMENTATION IN MATLAB

Jacob Melchi, Max Pagnucco, Evan Lo

University of Rochester

ABSTRACT

The goal of this project is to explore and develop a deeper understanding of singing and spoken word manipulation through various algorithms. Implemented in MATLAB, the project will contain two main steps, pitch detection and pitch correction, and then additional features will be added in order to increase the usability and functionality of our software. In order to make our project approachable to someone without the background of an audio engineer, we will create an easy-to-use GUI which will be accessible to understand on the first use.

1. INTRODUCTION

Autotune is an audio effect that automatically 'tunes' the input signal to a desired scale. This type of processing is often used on vocal tracks to correct anywhere that the singer may have been slightly out of tune. Many believe this type of effect is 'cheating,' and produces an artificial and inauthentic sound. However, when used in the proper context, autotune can be both a powerful creative tool as well as an invaluable effect to bring vocal recordings closer to perfection. In this paper, we will present how we achieved this effect both using methods learned throughout this course, and by exploring methods and ideas novel to us.

Our paper will be organized by first explaining each of the different processing stages that come together to create our live autotune function, as well as the many attempts that we chose not to use. The program first needs to detect the fundamental frequency of the (monophonic) input. Using that value, it then computes the frequency closest that maps directly to a note in an equal temperament scale, tuned to 440 Hz. Finally, the pitch of the signal is shifted to the specified equal temperament frequency, without altering the playback time. The program both works with an pre-recorded input file, as well has can be used with live input/playback.

2. PITCH DETECTION

The first stage of autotune is determining the pitch of the given input signal, so that we can figure out how much it needs to be corrected. We tried several analysis methods to accurately return the fundamental frequency of the input.

2.1. Using FFT & Peak Detection

When starting and researching how to detect pitch efficiently, we thought that by taking the spectral analysis of the input, we would then be able to extract the fundamental frequency by locating the highest peak. We also tried using cross correlation to determine the underlying frequency information in the input signal. However, although both these methods were accurate in analyzing the frequency of test sine waves, these methods were unreliable for returning the correct fundamental frequency when analyzing harmonically rich input signals (such as a human voice). This is most likely because our implementation, in order to be fast enough for live output, couldn't be as in-depth as it needed to be, so we moved on to another idea.

2.2. Using Pitch Function

After trying the methods listed above, we ended up settling on using MATLAB's 'pitch' function, which was the most efficient and reliable means for returning the fundamental frequency. Using this function forces us to use a window size that is 0.042 * sample rate, which at 44.1kHz gives a window size of about 2250 samples. Using arrays that contain pitch and frequency information for all the notes in a given scale, our program then compares the analyzed fundamental frequency against all the given values in the frequency array, until it finds the one that is closest (the absolute value between the two in minimized). This process returns two pieces of information for the next stage: the fundamental frequency of the input and the target frequency for the scaled output audio file. Figure 1 plots the fundamental frequency detected by the program from an example recording against the corrected frequency sent to the pitch correction function.

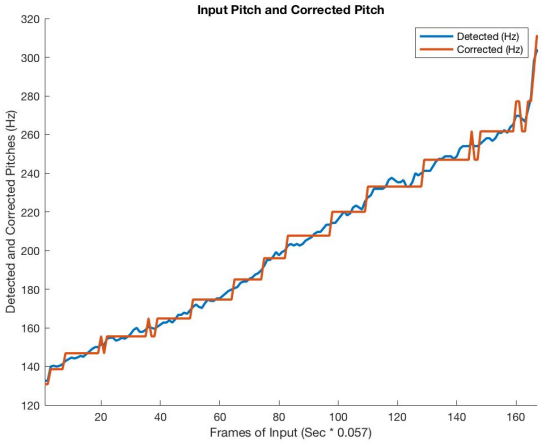

Fig. 1. Input Signal Pitch vs. Corrected Pitch. As shown, the program corrects the input pitch to the closest of a selection of predetermined, equal-tempered pitches.

3. PITCH SHIFTING

3.1. Approach 1: Phase Vocoder

Initially, we attempted to use the Phase Vocoder technique to perform the pitch shifting component of the autotune. Phase Vocoder is a technique that acts on an input signal in both the time and frequency domains; it first changes a signal's pitch by resampling it, and then compensates for the unwanted time change by linearly interpolating between the first and last frames of the signal's spectrogram. The phase component of the output is accounted for by applying the same phase *advances* from the original signal's spectrogram to the resulting spectrogram.

All of these steps, as one can imagine, are quite computationally expensive. This method, particularly when trying to apply it in real time, causes any host machine we tried to use to lag and stutter. All of the necessary processing does not happen fast enough to keep up with the sampling rate of the output DAC, and the resulting sound is quite egregious. Therefore, we decided to seek out an alternative method for pitch shifting.

3.2. Approach 2: PSOLA algorithm

The technique we decided to research and ultimately implement is called Pitch Synchronous Overlap Add (PSOLA). This method relies on detecting the fundamental frequency of an input signal first, and "stretching" or "squeezing" the signal based on this information.

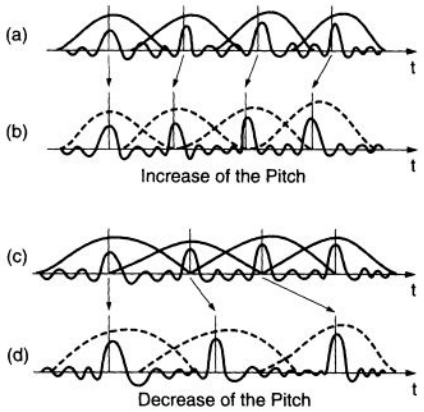

FIGURE 1 Example of the PSOLA method (from Sagisaka, 1990).

Fig. 2. Visualization of the PSOLA method

Once the fundamental frequency of an audio signal is found, markers are placed on the signal at every period of this frequency (preferably at each peak of this frequency). These markers become midpoints for a very small-scale windowing process. The signal is split up into windows at these markers, using an appropriate window function. For our implementation, we decided on the Hanning window. We use this function because it begins and ends at zero, which lowers the chances of the next step in the PSOLA process creating any unwanted audio artifacts.

Next, the chopped-up windows of the input signal are spread apart or pushed together based on a *new* desired fundamental frequency. The goal of this step is to match the period of the desired frequency to the spacing of these windows. This effectively alters the fundamental frequency of the input signal. The only thing left to do is to use the overlap-add technique to construct a single output signal.

Because these shifts in time are so miniscule, the output signal's character is only minimally altered, sometimes not at all. If an instance of PSOLA pushes the windows together too much such that the output signal is not the same length as the input signal, some frames can be duplicated to compensate. The choice of frame here is mostly arbitrary. In our implementation, we first count the number of frames that need to be duplicated. If copying every other frame is found to be insufficient, we duplicate every other frame *twice* until copying one time is once again sufficient.

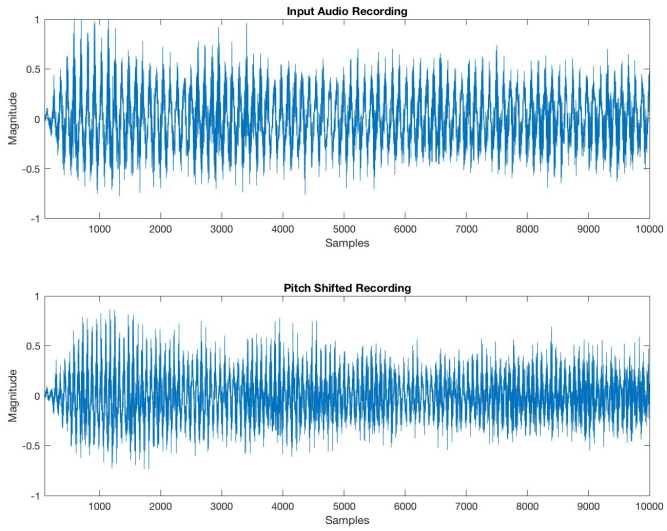

Fig. 3. Comparison of unaltered audio recording (top) and pitch-shifted audio recording (bottom, up 7 semitones). It is apparent that both signals have the same envelope, yet the shifted signal's fundamental period is significantly shorter.

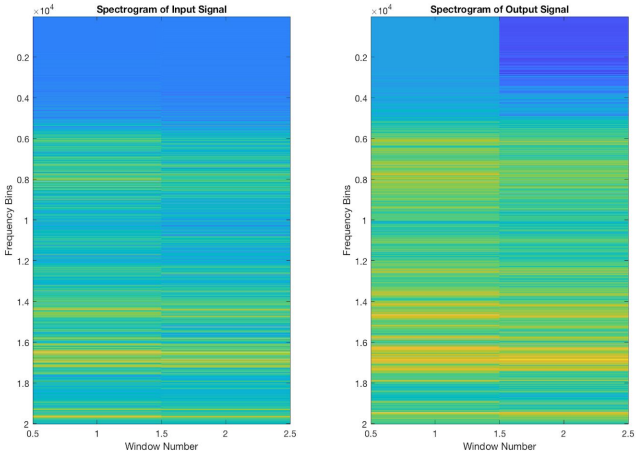

Fig. 4. Comparison of spectrograms of unaltered (left) and pitch-shifted (right, up 7 semitones) recordings. It is apparent that *some* noise is introduced, but it is also apparent that the fundamental frequency (brightest yellow line) is pushed upwards. It should be noted that while the altered signal looks much noisier, qualitatively the only discernible difference between the two is the shifted pitch.

4. LIVE IMPLEMENTATION

We used MATLAB's audio device reader to take audio input through the computer's microphone. The audio is windowed into frames for analysis. We experimented with different window types to try to make the output reconstruction as seamless as possible; in the live implementation the output frames are added end to end rather than using constant overlap-add (which allows for smoother reconstruction). This means there are some discernible clicking or 'gaps' in the output signal during live usage of our autotune effect.

To measure just how many gaps in the output there are, we took advantage of all of the capabilities of MATLAB's audioDeviceReader function. This function has the option to return the number of samples "underrun" or omitted. This provides us with a better understanding of what's causing the output to stutter. Comparing the number of samples underrun with and without the autotune applied, we found that there are *zero* samples underrun without. This obviously points to the fact that there are still issues with the program being able to catch up with the output sample rate.

5. GENERATING SCALE ARRAYS

In order to allow the user to select the scale they wish to tune their input to, we needed arrays that contained the pitch and frequency information for each scale. By using a sequence of half steps and whole steps to represent the 'jump' intervals along a major or minor scale, we were able to create a function that can generate a pitch & frequency array spanning two octaves to a user specified key and quality (major or minor). It generates this within a while loop by consulting the arrays of steps and selecting frequencies/keys from an array of chromatic frequency and pitch information. This gives the autotune program the information that it uses to tune the input signal accordingly.

6. CREATING A USER INTERFACE IN MATLAB

With the processing complete, emphasis was turned to creating a user interface that would be easy to use for the live implementation. We wanted one that could send live messages to alter what was being done to the input signal, and to be able to control all the parameters stated before. Out final GUI can be seen in Fig 4, where there are options for turning on the auto-tune processing or just returning the original input, stopping the program from running, and also selecting what key the outputs should map to. On top of that, while the program is running the user can choose whether they want to pitch up or down their voice, and how many semitones they want to do so. To the right of that, the first slider will "detune" the output signal by an amount of cents ranging from -50 to 50, equalling half of a semitone in either direction. Lastly the second slider will change the level of the threshold for triggering the program to input

audio. Essentially this slider is altering the sensitivity of the input, and ranges from 0 to 0.707 RMS. This threshold helps to save processing power, so the program isn't constantly running with no input, just background noise.

Creating this GUI was very straightforward once we got the hang of it, and consisted of radio buttons, button groups, state buttons, and sliders. Using the ".Value" notation, returning the state and value of the buttons was easy to implement. Only when we used radio-style buttons, like for the semitone selection, did we have to create seperate functions to return a value that corresponds to the label we gave it, not simply an on or off (1 or 0) like that state buttons. For the sliders, the range is set from 0 to 1 bottom to top, respectively, so after returning that value, only a little processing was needed to convert that to the range that we desired, whether it be cents or RMS values.

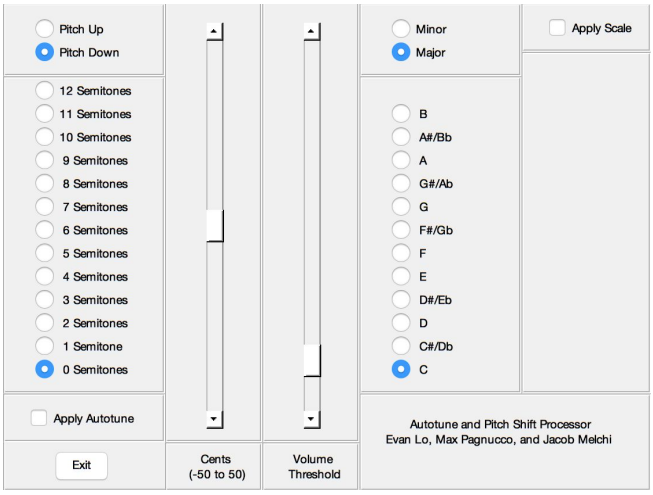

Fig. 5. Graphical User Interface for live implementation of autotune and pitch shifting. Created using MATLAB, and opens as a figure when running the project code.

7. CONCLUSIONS

With a working model of autotune completed, we can conclude definitively that we learned a lot about how to shift pitch using time-domain processing. Not only were we able to create a program that could correct the pitch of a pre-recorded file, but also we were able to fairly successfully adapt it to work for live processing. All of our processing was also able to be presented in a clear, easy-to-understand user interface, with all of the desired functionality that we set out to create.

Our final product, however, is not without its imperfections and downfalls; the most obvious being the issue where the live processing drops frames and returns a choppy output signal. In future work we would like to continue to troubleshoot this issue, and hopefully by reducing CPU usage and finding more efficient functions to input and output signal, we will be able to have much smoother playback in the live implementation. Similarly, we would like to find a better way to address the latency issue in the live playback. Whether it be creating a method to use a smaller window size, or some other answer, making the live implementation truly "real-time" processing would be great.

Lastly, it would be a big improvement if we were able to export our project as a VST or AU file, and then it could be used in a DAW for audio production and recording. We knew it would be hard to do so, but with more time we're sure that we'd be able to create this independent plug-in, and would be able to share out project that much easier.

8. REFERENCES

[1] B.H. Suits, "Frequencies for equal-tempered scale, A4 = 440 Hz," *Physics Department*, MTU, Web, 1998.

[2] Daniel Shub, "Using xcorr in pitch detections," *Help Forum*, MathWorks, Web, 21 August, 2011.

[3] Zhiyao Duan, "Assignment: Homework 5," *MATLAB template*, University of Rochester, Web, 7 March, 2019.

[4] National Academy of Sciences. "Voice Communication Between Humans and Machines," Washington, DC: The National Academies Press, 1994.i

[5] Peimani, Michael A., "Pitch Correction for the Human Voice," *Web Paper,* University of California Santa Cruz, 10 June, 2010

[6] J. Yuan, Y. Chen, Z. Zhao, S. Liu, "EECS 451 Team Project: Conan's Bowtie," University of Michigan, Web, 7 March, 2019