Pitch Shifting with Matlab

by Adam Croutworst, Mohammed Alhubail,
Ashli Nguyen, and Ross Holzworth

Summary

For our final project, our concept is to experiment with pitch shifting in the time domain. To achieve this project, we built a GUI in Matlab where the user would be prompted to record their voice for 1.5 seconds. Upon recording the user's voice, they will have a selection of eight filters to shift their voice. If time permitted, we anticipated to accomplish a pitch detection and pitch correction program that intakes a user's voice, detects the pitch, and performs pitch correction. Our new path has allowed us to achieve pitch correction, as pitch detection detracted a plethora of time away from our deliverables. These hindrances are highlighted in the sections Alternative Approaches, Limitations, and Problems.

The GUI will display the original signal captured and the manipulated signal. Both signals are plotted in the time domain, with milliseconds as the x-axis and frequency as the y-axis. The eight filters – or "features" as we call them – include the following: echo, doubling, fraction, high pass, three-point averager, slow down, speed up, and lower volume. Each of these features are a reflection of the lecture topics that we have learned for each week. For instance, three-point averager was a Finite Impulse Response filter that we discussed during Feed Forward filter section, speeding up / slowing down uses sampling rates to affect their voice speed, and high pass filter reviews our chapter concepts that introduced low, high, and band stop filters. Other aspects of our application incorporate coping the manipulating signal to replace the original signal, and resetting the program with a clean slate to compute a new recording. Our final project encapsulates an array of filters to solidify our knowledge of our course concepts.

Background

In its simplest form, pitch shifting involves taking an audio recording and speeding up or slowing down its playback in order to increase or decrease the pitch a listener perceives. This can be accomplished digitally through the use of simple linear filters as we have done in our program. This approach has the drawback of causing very obvious distortion to the original sound recording, and it also changes the time it takes for a recording to play for a listener.

The term pitch correction is frequently used to distinguish modern techniques of sound manipulation from basic pitch shifting. These techniques vary heavily, but they generally all arise from the desire to minimise the distortion caused by pitch shifting and increase the precision of manipulation.

As it stands, our project includes a very basic implementation of pitch shifting. The eventual goal was to implement an algorithm for pitch detection and correction that was much more sophisticated, but time ran out.

Technique

In this section, we will discuss the ideas we implemented to achieve our different purposes for this project.

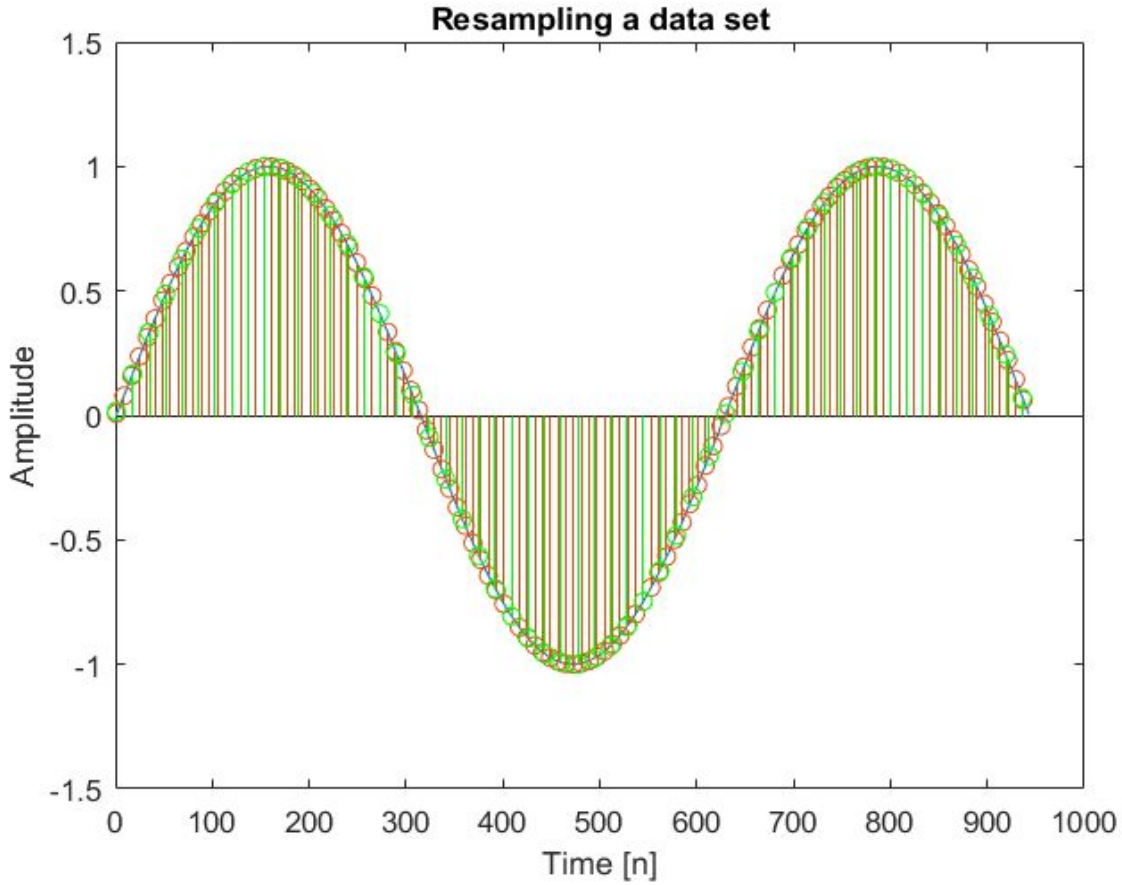
The first part of audio manipulation we wanted to implement was decreasing the volume of a recorded sound. We chose this as a starting point because of its simplicity. Intuitively, we believed that lowering the values inside the audio data vector would do the trick. Thus, we created the 'Fraction' button currently available on our GUI. However, this caused us to run with issues with Matlab's soundsc function. We realized that the function normalized the values of the audio data, and then played it at the highest volume. After more investigation, we realized that the soundsc function normalized the data in relation to the maximum value in the dataset. So, as a workaround to achieve our purpose, we created a new value that is larger than the maximum by a set percentage. Then, we replace the first element in the dataset with this new value. This way, we are able to use the normalization of Matlab's soundsc function to our advantage. The results of this method can be observed by using the 'Lower Volume' button in our program.

Next, we wanted to study a change on the data set we thought of. This change is to simply double each element in the vector. So, a vector of values [1 2 3] would change to become a vector of [1 1 2 2 3 3]. We achieved this by using Matlab's repelem function, which takes a data vector, and the number of times you want each element repeated. We were not sure what to expect as the result of this manipulation at first. Once we've implemented it, we noticed that the pitch was significantly lowered when compared to the pitch of the original dataset. Eventually, we linked the cause of this pitch change to the change of the wave length. Generally, it is hard to define pitch. However, there are certain facts that we know about pitch. For example, increasing the frequency results in higher pitch. Now, we noticed that the manipulation we did to the data set caused a longer wave length. Having a longer wave length lowers the frequency. This decrease in frequency causes a lower pitch. Thus, the result of doubling the elements in a data set is a longer wave length, which causes a much lower pitch. The results of this manipulation can be observed by using the 'Doubling' button in our program.

After observing the changes caused by the doubling function, we wanted to find a way to have more control over the wave length, and thus more control over the pitch. The easiest method we found that would allow us to have this control was being able to speed up, or speed down the audio data. To do this, we change the sampling rate of the function of the data set. This will result in increasing or decreasing the size of the data set. One way to think about this is having stem plots over our data set, where the points pointed to by stem values represent our new data set. Figure x shows an example of how this resampling can be observed. In the figure, it can be observed that we have two stem plots. The red stem plot has a higher sampling frequency than

the green stem plot. So, the data set represented by the red stem plot would be larger than the data set represented by the green stem plot. We achieve this by using the resample function in Matlab, and resampling by a set percentage of the sampling frequency. The idea to consider here is that a higher sampling frequency would result in a larger data set, which would result in a longer wave length. As discussed earlier, a longer wave length causes lower frequency, and therefore lower pitch. On the other hand, a smaller sampling frequency would result in a shorter wave length, which would result in higher frequency, and higher pitch. It is important to note that continuously applying resampling to speed up the data set would eventually cause aliasing. The results of the speeding up, and slowing down the data set can be observed by using the 'Speed Up,' and the 'Slow Down' buttons in our program. Overall, we believe that doing the speed up, and speed down functions greatly improved our understanding of how sampling frequency affects a data set, and more specifically, an audio segment.

Fig. 1 resampling a data set.



Next, we wanted to implement general filters that would allow certain frequencies in, and see how they affect our audio data. As a low pass filter, we implemented the three point averager

filter. We chose this filter because we felt more familiar with it. Intuitively, a low pass filter allows lower frequencies to pass more freely. So, the resultant data set will have lower pitch (because more of the lower frequencies passed). The results of the low pass filter did indeed follow our intuition, and they can be observed by using the '3-Point Averager' button in our program.

To have something to compare the results of the low pass filter to, we created a high pass filter with b = [1 -2/3], which gives us a zero at 2/3. Following the same reasoning as the low pass filter, the higher pass filter would allow the higher frequencies to pass more freely. So, the resultant data set would have higher pitch. The results of the filter matched our expectations, and can be observed by using the 'High Pass' button in our program.

We can use the High Pass, and the Low Pass filters multiple times in the program by recording the signal, copying it using the 'Copy' button, and the applying the filter again. This is what we studied and referred to as cascading filters. We know from class that it is possible to create a single filter that does the process of the multiple filters. To create this filter, we would need to multiply the transfer functions of the independent filters, and that will give us the transfer function of the single filter that can do the full operation alone.

Next, we wanted to be able to create echo to the audio data. The idea behind echo is that we want to repeat our data after a certain amount of delay, which will be referred to as td for time delay. Another idea that can be used is that the sampling frequency can be used to represent the amount of audio data the will be used to play the sound per second. Now, let us thinking about just repeating the data. One way to do this would be to use Matlab's conv function to do convolution on our data set, and the vector [1, 1] (we do not need to worry about how the addition will affect our signal right now). Usually, echo sounds weaker than the original sound, so we can change our vector to become [1, ⅓], so that the repeated data sounds weaker. Now, we need to create a delay between the signal, and the weakened repeated signal. To do this, we will go back to the idea of time delay, td, and sampling frequency, fs. To create the delay, we can simply add zeros between the values in the vector we use to create the repeated weakened signal. In other words, Our new vector will become [1, 0, 0, …, 1]. However, we want to be able to control the delay. So, we can use the zeros function, and create a number of zeros relative to the time delay we want, based on the sampling frequency. So, we would create the following vector
[1, zeros(1, round(td * fs)), 1/3]. Now, we can see that if we wanted a 0.3 second delay, we would choose a td value = td. We round the values of td * fs because the zeros function only accepts integer values. However, this is not good enough to create godo echo. Echo usually repeats, and even the echo can have an echo. Therefore, we get the data set that results from convolving our original data set with the vector, and convolve it with the vector again. This results in a nicer echo where the sound will be heard 4 times. Creating echo definitely helped us better understand the conv function, and how to set its boundaries. The results of this manipulation can be seen by pressing the 'Echo' button in our program.

Design

Our design is based on the idea that you have GUI (Graphic User Interface) that is relatively easy to set up. You can use the Guide section in Matlab to create buttons and each button can be renamed with a tag. This tag is referenced in a CallBack figure.m file that can be used to manipulate what that button does with an handles object.

There's two portions to the design, the recorded signal and the signal to be changed is always on top. The manipulated signal is displayed on the bottom for comparison to see what's been changed. The user can also play the signal from the before and after manipulation to hear the difference. If repeated manipulation on the initial signal is desired they can push on the copy button and the manipulated signal will be sent to the above signal to be manipulated next.
The reset button is for if the user wants to start all over.

This design is easy for the next person to come into and continue development. All they would need is to right click the .fig file and open it in Guide (Figure 2). They can expand/decrease the size of the signal display, the buttons. They could also add in new buttons and expand different functions and filter manipulation to the signal (Figure 3). To change the tag name and the button name, right click on the button and view the property inspector (Figure 4). To save the changes, one can do control S or press the play button to view it in GUI view versus Guide view. Then go to the figure.m file to put their code for signal manipulation into the callback function of that button, can be directed there by right clicking the button and scroll over to view callback (Figure 5/Figure 6). The create functions has already been completed for us by matlab.
See the picture Below.

Figure 2: How to open the GUIDE

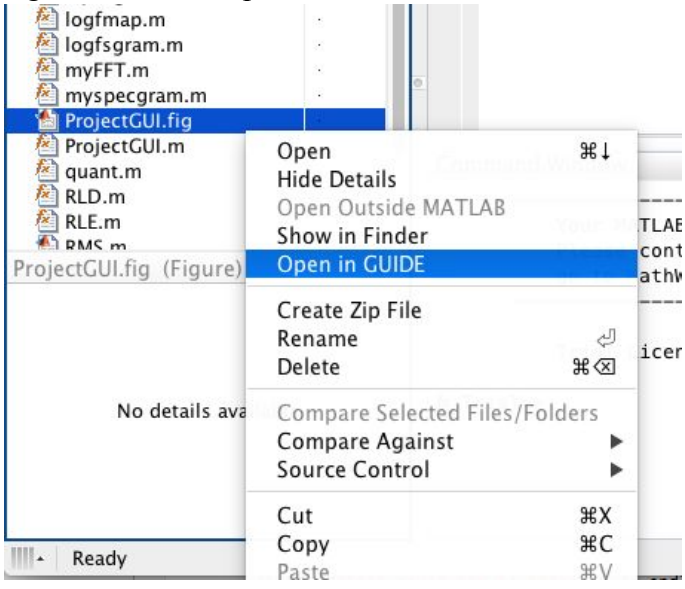
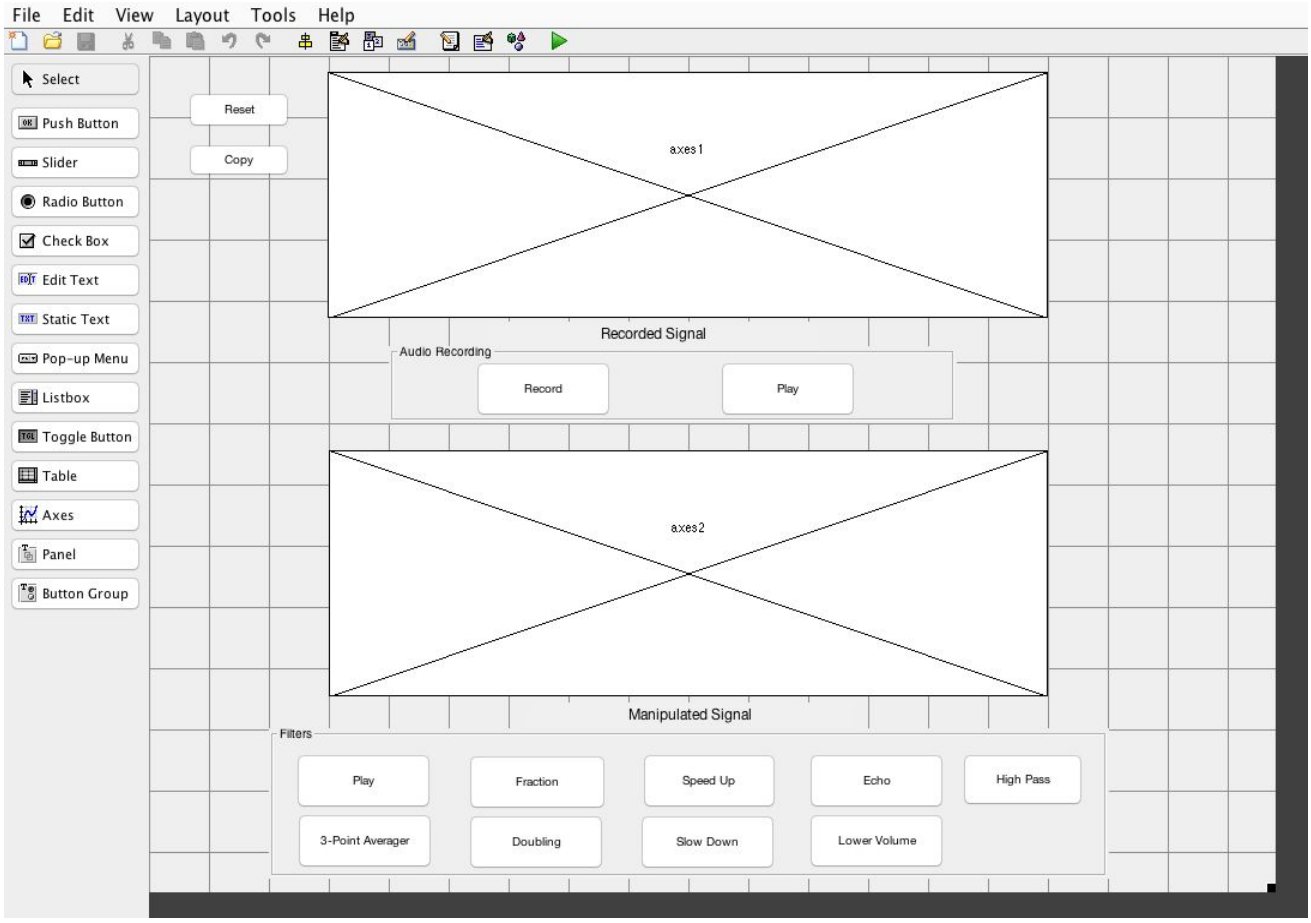


Figure 3: The GUIDE view of the GUI



Buttons that can be added are listed on the left side. The Green Play button on top will build the current GUIDE to a GUI.

Figure 4: The Property Inspector

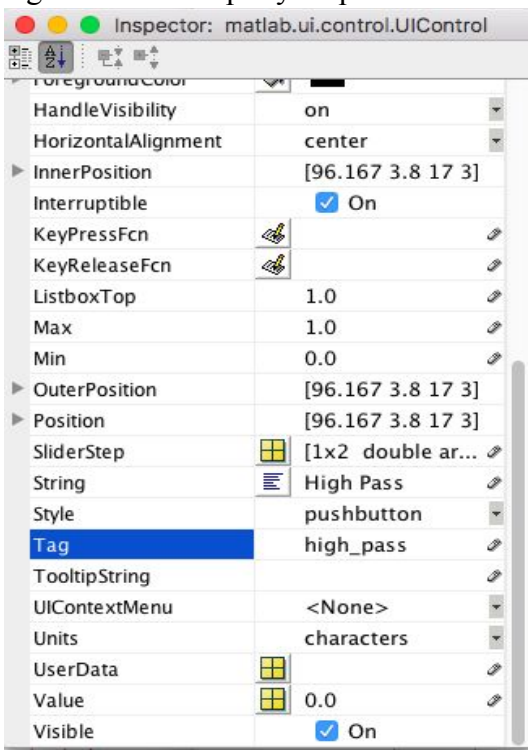


Figure 5: View Callbacks for HighPass Button

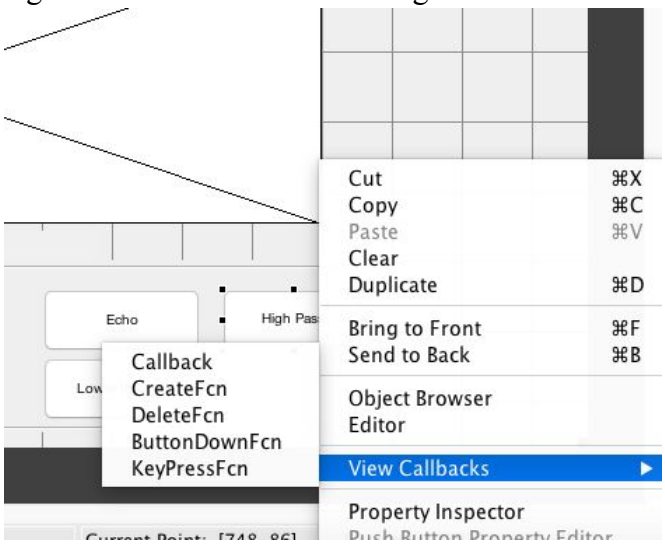
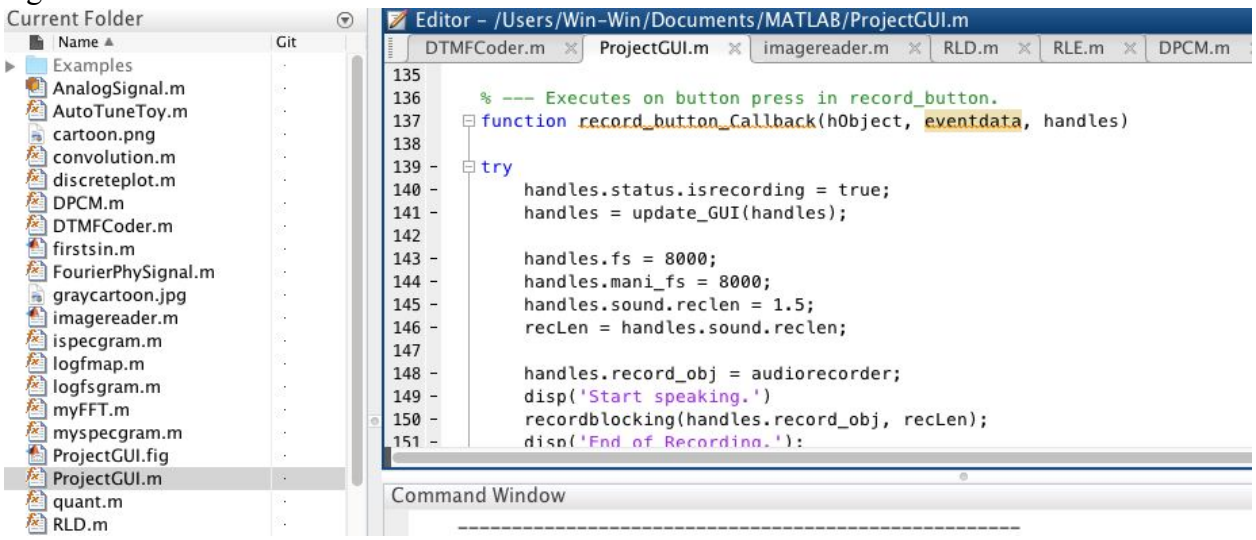


Figure 6: The Callback function for record button in the .m File



User Guide

The way the program works is that you can download our ProjectGUI.m and ProjectGUI.fig files and load it into MATLAB. You can type ProjectGUI into the command line and the GUI will pop up(Figure 7). Click the record button and start talking, the microphone, if it's working, will record 1.5 seconds of speech and draw the corresponding signal onto the recorded upper figure. To listen to the signal, press the play button directly under the recorded signal (Figure 8). To manipulate the signal, press any of the option buttons below the Manipulated Signal graph besides Play (Will only work when a signal in displayed in the manipulated signal graph). The corresponding manipulated signal will be displayed in the second graph and can then be played to listen to the changes in the signal. In Figure 9, the double button has been pressed and the signal displays the manipulated signal below. The lower the voice button is clicked in Figure 10 and the signal has been shifted down to sound softer. Can press play for both signals to compare and contrast the sound. If user wants to further manipulate the original signal, continuously press the copy button, a manipulation type and repeat. The copy button copies the manipulated signal into the recorded graph and can be further manipulated (Figure 11). If user wants to reset, they can click the reset button and the GUI will reload and reset. Another way is to just click record and it'll record over the recorded Signal, ready to be manipulated.

Figure 7: Load file in and start program

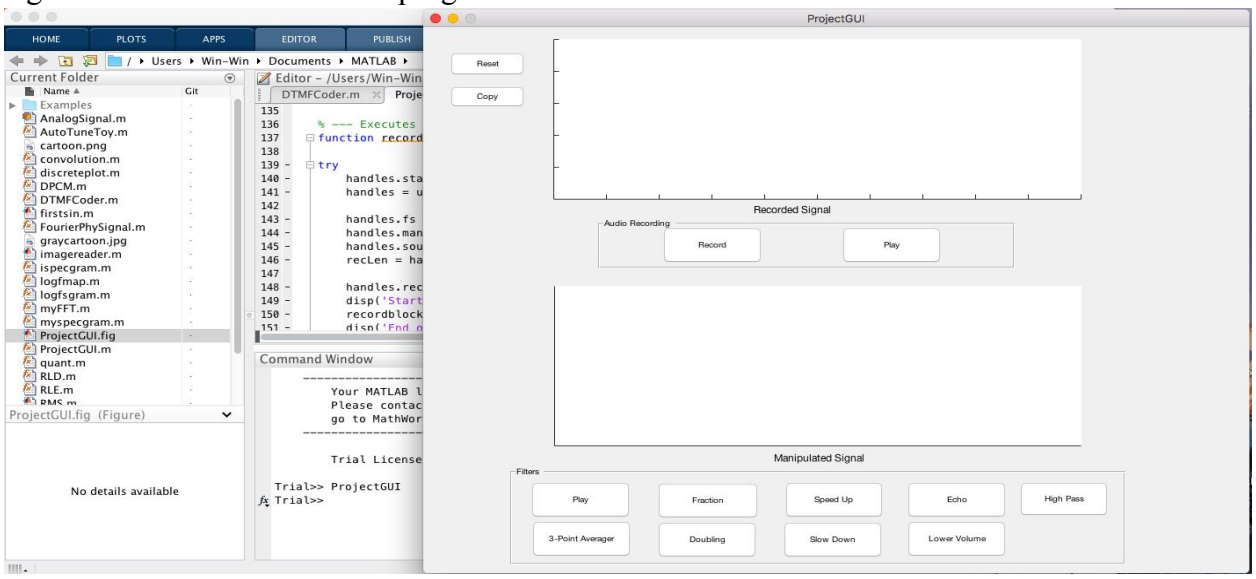


Figure 8: Upper Record and Play Button

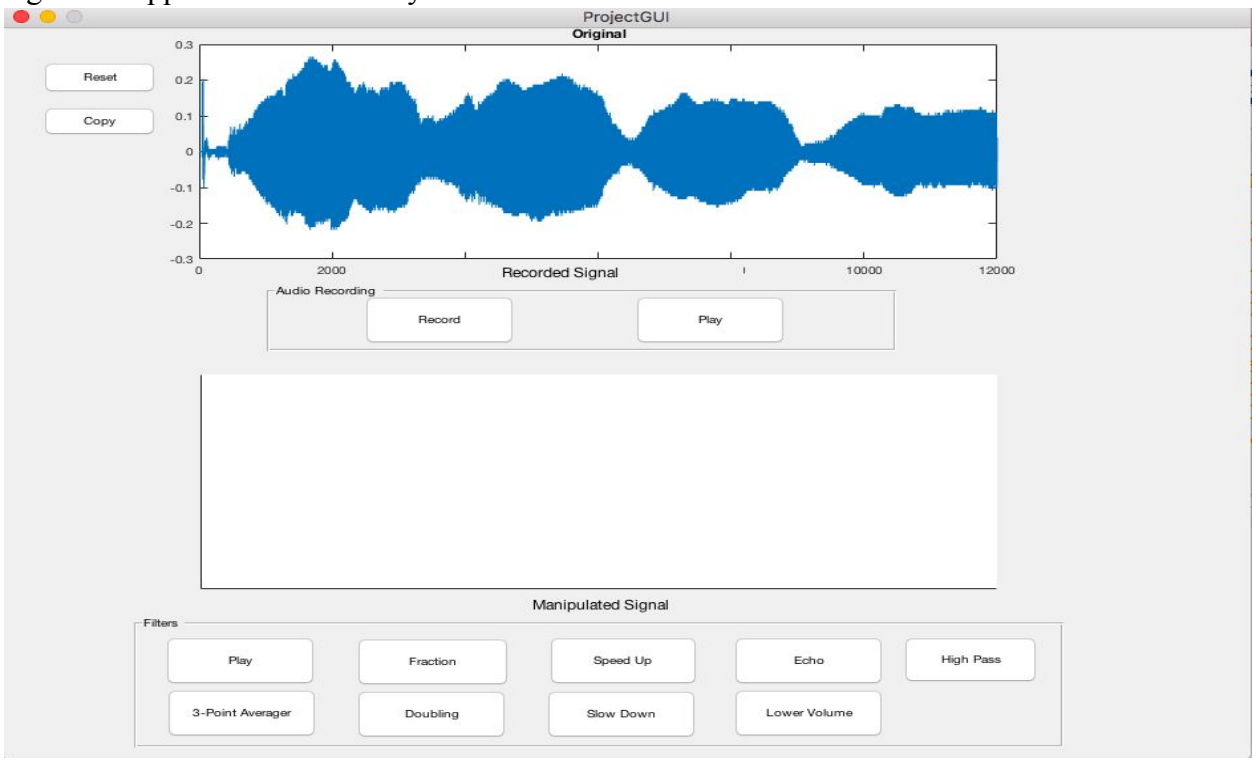


Figure 9: Doubling the Signal

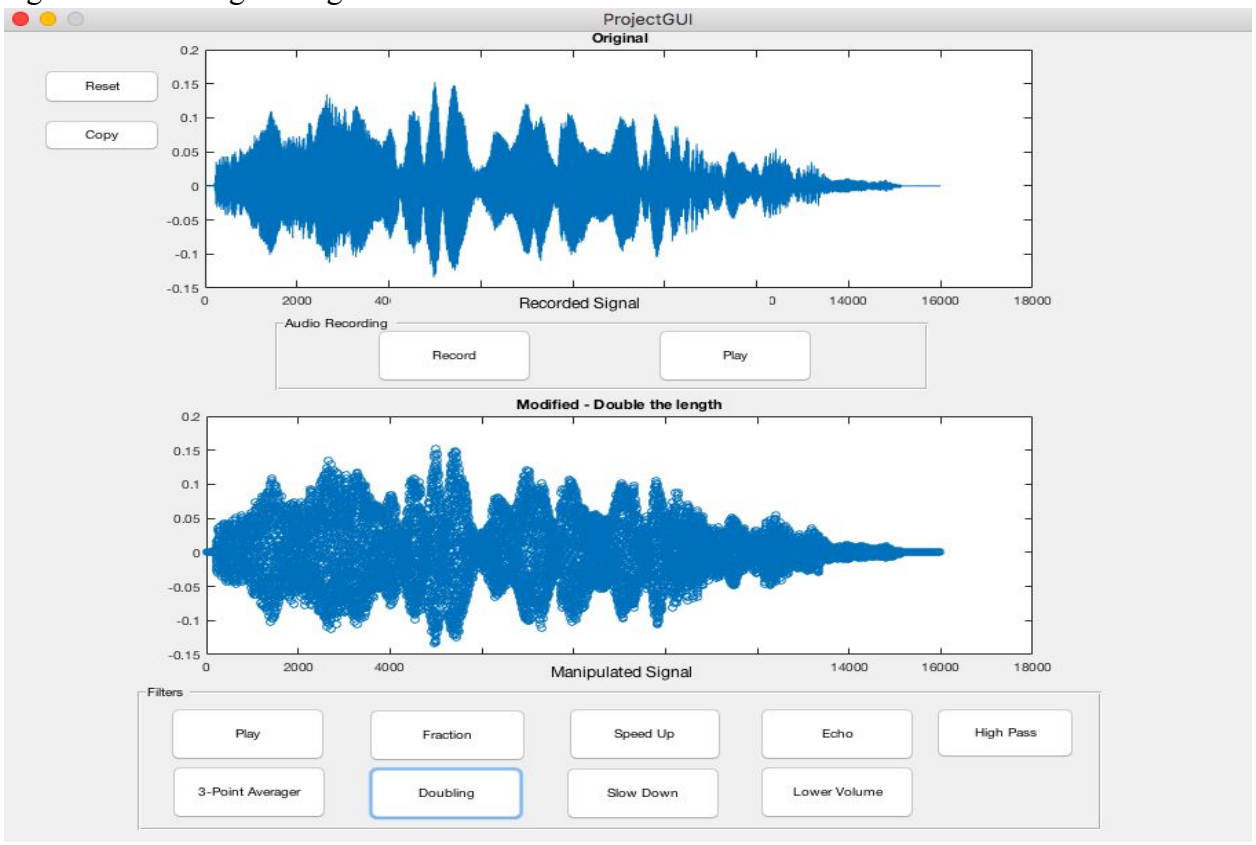


Figure 10: Lower the Voice Button

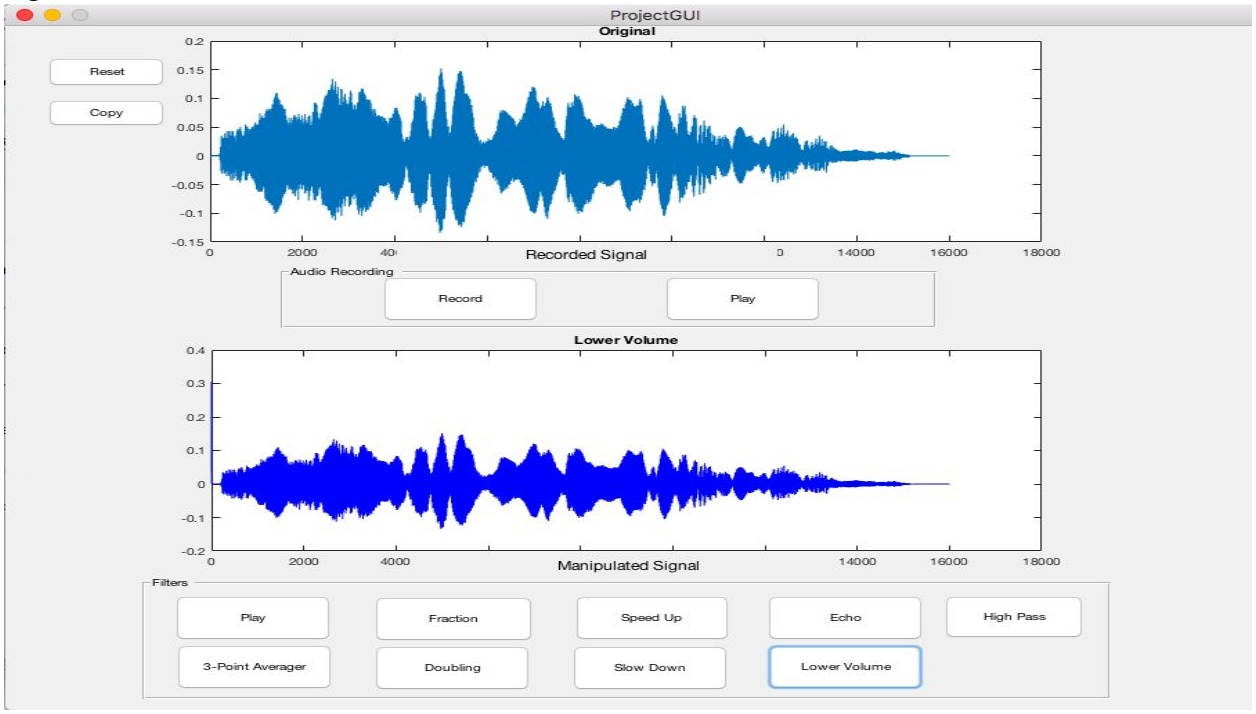
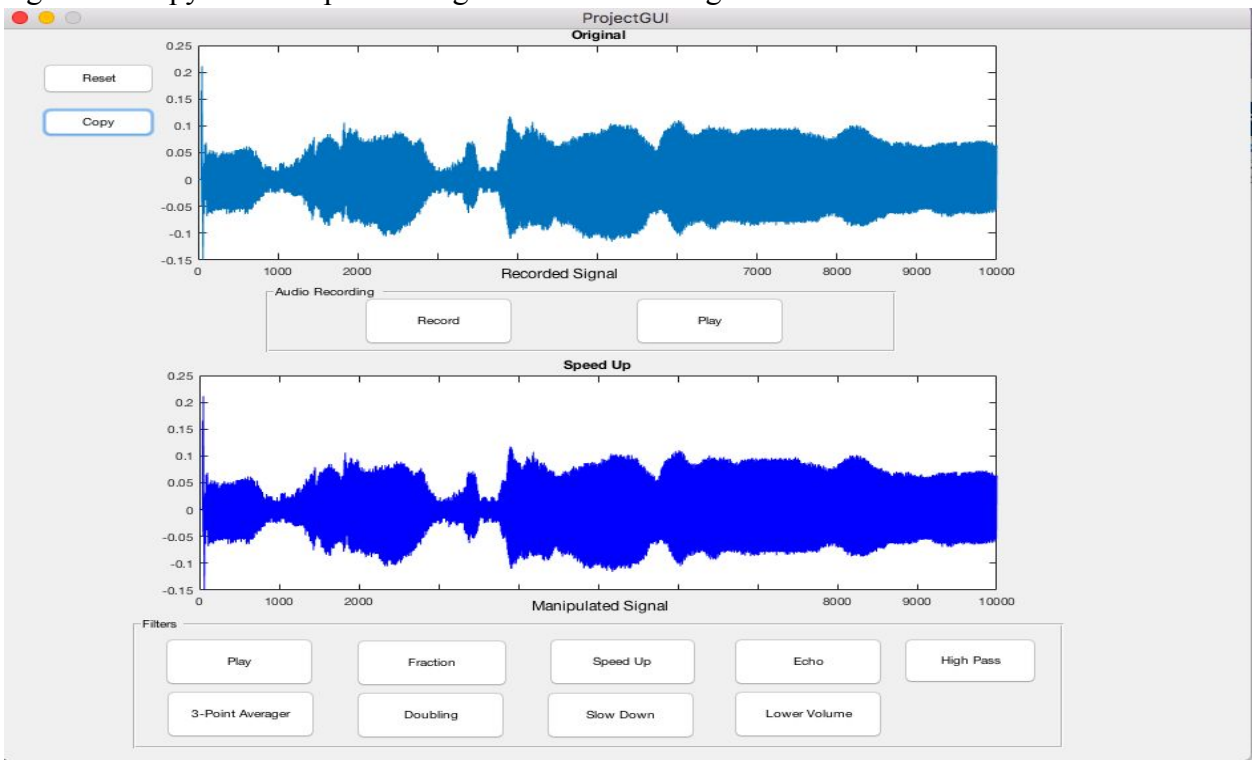


Figure 11:Copy the Manipulated Signal to Recorded Signal



Alternative Approaches

1.Cepstral: inverse DFT of the log magnitude of the DFT of a signal
Cepstrum equation: c[n]=F^-1 {log|F[x[n]}|}
Useful for speech processing by being able to separate who's speaking and for phase based speech deconstruction. Cepstral domain is the frequency represented in logarithmic magnitude spectrum. Given in "quefrency" , it represents a pitch lag, where there's the most energy and thus that is where the pitch is. To detect the pitch in Cepstral domain you can pick the peak of a signal within a certain range.

2. Phase Vocoder: short-time Fourier transform (STFT) of the signal
Useful for timescale modification of audio signals and for performing time/pitch modifications such as time stretching, pitch shifting, and pitch scaling. A phase vocoder is a time-frequency representation of the signal, using the short-time Fourier transform of the signal. The time-frequency representation then can modified through time/frequency manipulation and played back by finding the inverse STFT (ISTFT).

Limitations

Given time and motivation, our project would have included pitch detection, modified and focused pitch correction, and an accurate method of how to measure our successful pitch correction. Throughout the early stages of our project, we researched about the first phase of pitch correction which is actually detecting the inputted pitch. Our research encompassed topics such as Cepstral Analysis – analyzing pitch in the Cepstral domain, autocorrelation and adaptive filtering in the time domain, and the perceptual pitch detector based on human perception of pitch. After the examination of a myriad of methods to implement, in multiple domains, we quickly decided that there is too many unknown variables and information about signal computing that we are unfamiliar with to make an informed decision about pitch detection. Another issue arose. What is pitch? How do we define pitch to even detect it? Is pitch just frequency, or does it include human perception and how do we measure human perception? The realm of unknown lead us to the decision to bypass the pitch detection process and utilize pitch correction and pitch shifting instead. However, if we had the time to complete this extension, our team would further research and implement cepstrum – Cepstral Analysis. This was our most researched topic before starting to work on pitch shifting; from our research, we determined that Cepstrum is the spectrum of a spectrum and is computed from the inverse DFT of the logarithmic magnitude of the DFT of a signal. This is discussed in more detail within the Alternative Approaches section of our report.

Moving onto our next improvement, our project team would modify our pitch shifting filters and create a focused pitch correction algorithm to enhance one feature of a person's voice. In our current status of our project, we are a bit spread out in the topics we are covering. The next approach will be to narrow our focus and limit this to one filter. More likely than not, we would use the Phase Vocoder for time scale modifications. A Phase Vocoder is the short-time Fourier Transform of a signal and can be used in collaboration with Cepstrum. If we can first detect a pitch, then the Phase Vocoder will perform our pitch correction to modify the user's manipulated pitch. The goal of using the Phase Vocoder would be to clarify and enhance a user's vocal pitch. For example, a singer / musician would use our program to input a signal and output a more pleasing sound than originally registered. These type of applications are already on the market, but it would help our team to fully comprehend pitch detection / correction and expand upon the market's state of the art algorithms. The Phase Vocoder is provided greater detail in the section above – Alternative Approaches. Our final stretch goal / objective is to measure the effectiveness of our pitch detection and correction software.

Two avenues are available to us when discussing the measurement of our pitch correction program. The first option is to build a testing suite or program to measure the effectiveness of our pitch correction. When building the testing program, we have to take into account the actual pitch we are trying to achieve as well as differentiating the inputted pitch compared to the disrupting noise. This method of measuring pitch is accurate and precise, although, will take an extensive amount of time to complete. The alternative option is to test our pitch correction program with a musician and / or a professional vocalist. This route will eliminate the extra time to build a test suite. Although, the outcome has the potential to not be as accurate. If we stay motivated to complete this project extension, we will have to compare the pros and cons with both options. Stemming from our limitations, given time and motivation, we would have completed our initial project design and built an end-to-end Pitch Detection and Pitch Correction program, along with the necessary measurements to effectively correct a user's vocal pitch.

Future Work

An extension to our work would to be use a phase vocoder, which would allow us to do both time stretching, and pitch shifting, and identify the pitch of the signal. Once the pitch is identified, we can work on doing auto-tuning as the next phase. Another way to extend our current project would be to work on different forms of signal shifting and manipulations. For example we were thinking of doing the deep monotone voice, or chewbacca from Starwars. There's an unlimited way to change the recorded voice signal!