ABSTRACT

We know, that the musical scales are divided into 12 pitches each separated by a semitone – the difference in note between two adjacent keys on a piano or frets on a guitar neck. The goal of pitch correction is to retune a slightly high or low note to the nearest semitone.

In the system usually used by MIDI instruments in which pitch is assigned a number, with the 440-Hz A being 69 and each semitone increasing or decreasing the pitch number by 1, it is related to frequency *f* by a simple formula.

If during a singing attempt, suppose the A note came out at, say, 445 Hz instead, then we can code to correct the frequency back down would ensure that the recording sounds in tune. It's easier with digital signals, but for analog signals, we can sample and perform the same function. Although it is possible to alter analog signals – those based directly on the electrical signal generated by a microphone, or by a guitar pickup – a wider range of effects is possible when working with digital signals.

A digital signal uses discrete values rather than continuous ones, so converting an analog signal requires taking sets of discrete points or samples. These digital signals can be altered so that a sound produces the correct musical note by using a phase vocoder. This works by first changing the duration of the sound without altering its frequency, and then changing the frequency to both hit the correct pitch and restore the original duration.

It breaks an audio signal down into many small, overlapping frames and then changes the spacing of those frames to change the total duration of the sound. In practice, this is a complicated task that requires the use of the advanced math of Fourier transforms to convert the signal into a form that can be manipulated in this way.

The sound is then resampled to take it back to its original duration and hit the desired note.

ACKNOWLEDGEMENT

There is consistently a sense of gratitude one communicates to others for the supportive and impecunious service they render amid all phases of life. We would like to express our sincere gratitude to all the people who have helped and supported us throughout. We are deeply indebted to Professor Ivan Selesnik, Abhishek Vasu (Teaching Assistant) and Lei Yin (Teaching Assistant) for guiding us throughout efficiently and providing us valuable resources and for their cooperation and readiness to share their expertise and information and to commit their precious time to examine related topics.

CONTENTS

1. INTRODUCTION

Midi File

MIDI (Musical Instrument Digital Interface) is a technical standard that describes a communications protocol, digital interface and electrical connectors and allows a wide variety of electronic musical instruments, computers and other related music and audio devices to connect and communicate with one another. A single MIDI link can carry up to sixteen channels of information, each of which can be routed to a separate device.

MIDI carries event messages that specify notation, pitch and velocity (loudness or softness), control signals for parameters such as volume, vibrato, audio panning from left to right, cues in theatre, and clock signals that set and synchronize tempo between multiple devices. These messages are sent via a MIDI cable to other devices where they control sound generation and other features. A simple example of a MIDI setup is the use of a MIDI controller such as an electronic musical keyboard to trigger sounds created by a sound module, which is in turn plugged into a keyboard amplifier. This MIDI data can also be recorded into a hardware or software device called a sequencer, which can be used to edit the data and to play it back at a later time.

Advantages of MIDI include small file size, ease of modification and manipulation and a wide choice of electronic instruments and synthesizer or digitally-sampled sounds.[3] Prior to the development of MIDI, electronic musical instruments from different manufacturers could generally not communicate with each other. With MIDI, any MIDI-compatible keyboard (or other controller device) can be connected to any other MIDI-compatible sequencer, sound module, drum machine, synthesizer, or computer, even if they are made by different manufacturers.

MIDI technology was standardized in 1983 by a panel of music industry representatives, and is maintained by the MIDI Manufacturers Association (MMA). All official MIDI standards are jointly developed and published by the MMA in Los Angeles, and the MIDI Committee of the Association of Musical Electronics Industry (AMEI) in Tokyo. In 2016, the MMA established The MIDI Association (TMA) to support a global community of people who work, play, or create with MIDI.

Phase Vocoder

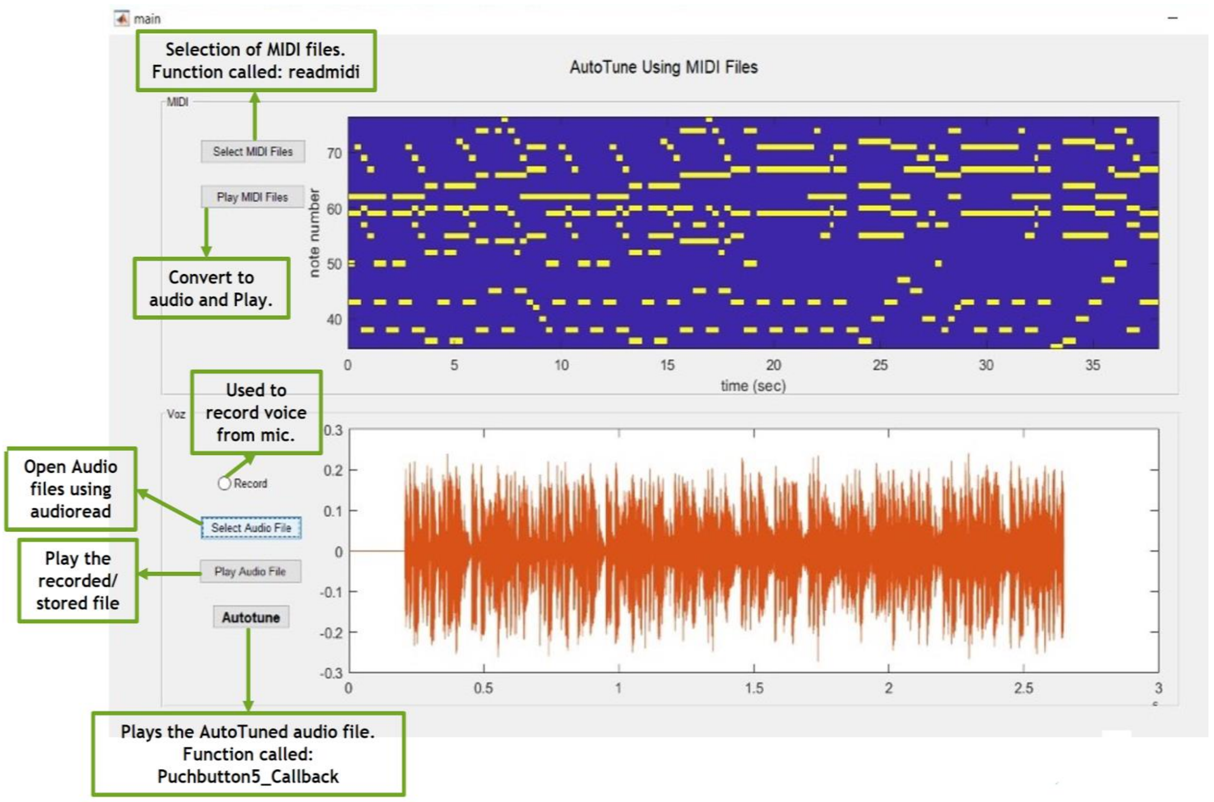
The Phase Vocoder [FlanG66, Dols86, LaroD99] is an algorithm for time scale modification of audio. One way of understanding it is to think of it as stretching or compressing the time-base of a spectrogram to change the temporal characteristics of a sound while retaining its short-time spectral characteristics; if the spectrogram is narrowband (analysis window longer than a pitch cycle, so the individual harmonics are resolved), then preserving the spectral characteristics implies preserving the pitch, and avoiding the 'slowing down the tape' pitch drop. The only complication to the algorithm is that the phases associated with each bin in the modified spectrogram image have to be 'fixed up' to maintain the dphase/dtime of the original, thereby ensuring the correct alignment of successive windows in the overlap-add reconstruction.

We wrote a phase vocoder function called the 'pvoc' unit generator. It first calculates the short-time Fourier transform of the signal using 'stft'; 'pvsample' then builds a modified spectrogram array by sampling the original array at a sequence of fractional time values, interpolating the magnitudes and fixing-up the phases as it goes along. The resulting time-frequency array can be inverted back into a sound with 'istft'. The 'pvoc' script is a wrapper to perform all three of these steps for a fixed time-scaling factor (larger than one for speeding up; smaller than one to slow down). But the underlying pvsample routine would also support arbitrary timebase variation (freezing, reversal, modulation) if one wished to write a suitable interface to specify the time path.

2. DISPLAY

GUI:

We made the GUI in MATLAB. It has two plots one which displays the MIDI waveform and other displays frequency response of the audio file. We have Total of 5 push buttons and one radio button. The first button is use to select the midi file. The second button plays the midi files. Radio button is used to record the audio files on the spot. When the radio button is selected the audio is in recording mode and when it is deselected audio has been recorded or not recording. The third push button is used to select pre recorded audio files and fourth button plays the audio file which we have selected or recorded. The last push button is where all the process happens. It compares the audio file to MIDI files and changes the pitch of the note and plays it.


main
AutoTune Using MIDI Files
Selection of MIDI files. Function called: readmidi
MIDI
Select MIDI Files
Play MIDI Files
Convert to audio and Play.
note number
time (sec)
Voz
Used to record voice from mic.
Record
Open Audio files using audioread
Select Audio File
Play the recorded/ stored file
Play Audio File
Autotune
Plays the AutoTuned audio file. Function called: Puchbutton5_Callback

3. IMPLEMENTATION

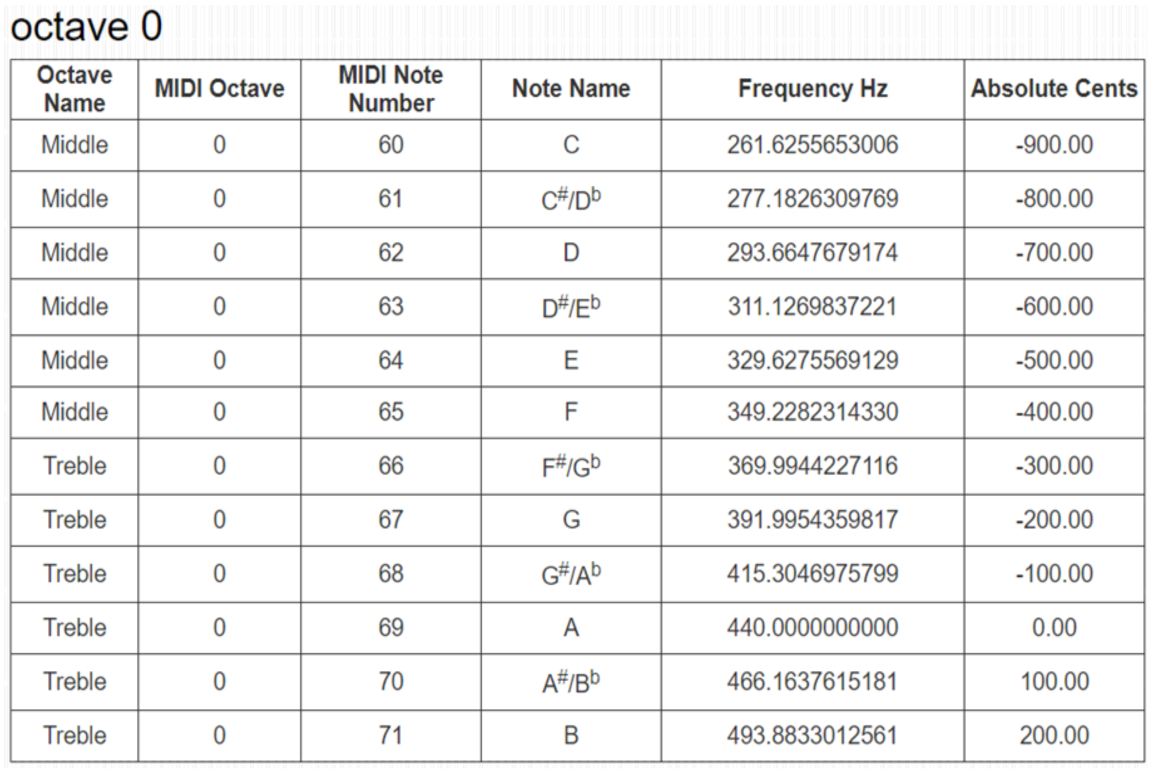
Writing the MIDI files:

We can write our own midi files or we can use the already available midi files. Basically the midi function is our code only generates piano notes. So if we want to write the midi file we have to give:

I. track number
II. channel number
III. note number (midi encoding of pitch)
IV. Velocity
V. start time (seconds)
VI. end time (seconds)
VII. message number of note_on
VIII. message number of note_off

This will create the midi files and store in in .mid file

How to give the correct note number? This can be seen from the Internet. You just choose the song which you want to autotune and check the note of the particular words which should be correct. Or if you are creating your own song then you know which note you want to sing. Just for the reference we have attached the octave note below:



octave 0

Octave Name	MIDI Octave	MIDI Note Number	Note Name	Frequency Hz	Absolute Cents
Middle	0	60	C	261.6255653006	-900.00
Middle	0	61	$C^{\#}/D^{b}$	277.1826309769	-800.00
Middle	0	62	D	293.6647679174	-700.00
Middle	0	63	$D^{\#}/E^{b}$	311.1269837221	-600.00
Middle	0	64	E	329.6275569129	-500.00
Middle	0	65	F	349.2282314330	-400.00
Treble	0	66	$F^{\#}/G^{b}$	369.9944227116	-300.00
Treble	0	67	G	391.9954359817	-200.00
Treble	0	68	$G^{\#}/A^{b}$	415.3046975799	-100.00
Treble	0	69	A	440.0000000000	0.00
Treble	0	70	$A^{\#}/B^{b}$	466.1637615181	100.00
Treble	0	71	B	493.8833012561	200.00

Audio File:

You can record your own audio file in any software of your choice. Audio file can be any file extension you want, it can be .mp3, .wav, etc.

AutoTune:

So after we have the MIDI and the audio files, we want that wherever the singer has misses the pitch of the note that should be corrected. So here is how it works. Midi file is converted into the Matlab structure which is then converted into audio vector. We can get the information about the midi files also. We convert the midi vector to frequency domain. Now we read the audio file. Then we get the desired frequency of every note from the midi files (Target frequency). Then the audio file is converted to frequency and sampled. Now we know the desired frequency of the note and the we get the actual frequency from the audio file (Portion Frequency), so we calculate the Target factor which is ratio of target factor over portion frequency. We have used Phase vocoder to change or shift the frequency. So phase vocoder calculates the stft , then the phase advance is calculated, i.e we change the phase of the sample. Finally the inverse stft is computed and then we get the final shifted audio file. This shifted audio is played and also is stored in the wav file.

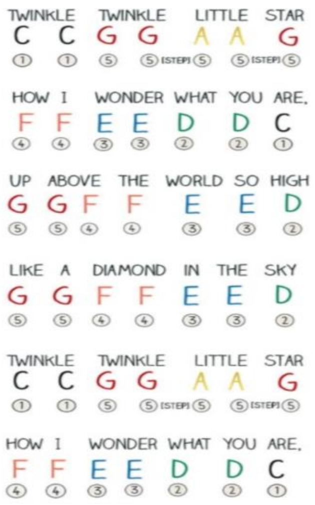
Issues Faced:

The major issue which was faced is while recording the audio files. So when we change the phase of the audio it shrinks it time domain. So if the audio is normally recorded a lot of words are missed and finally audio is not as expected. So one solution what we did is record the audio file for every note and giving ample of time for the audio to shrink. We can also find a way better solution for this problem.

4. TESTING

To test the code and the GUI for the AutoTune, we created a bunch of test MIDI files and also recorded Audio files to be AutoTuned.

We'll explain the testing process using the example of the file we used for the Demo. So, first we created a MIDI file (using writemidi.m) for the song 'Twinkle Twinkle Little Star' using the Musical Notes below.


TWINKLE TWINKLE LITTLE STAR
C C G G A A G
HOW I WONDER WHAT YOU ARE.
F F E E D D C
UP ABOVE THE WORLD SO HIGH
G G F F E E D
LIKE A DIAMOND IN THE SKY
G G F F E E D
TWINKLE TWINKLE LITTLE STAR
C C G G A A G
HOW I WONDER WHAT YOU ARE.
F F E E D D C

Once the MIDI file was created, we recorded the song 'Twinkle Twinkle Little Star', which was completely out of tune. Then this file was AutoTuned using the code we designed on the GUI above.

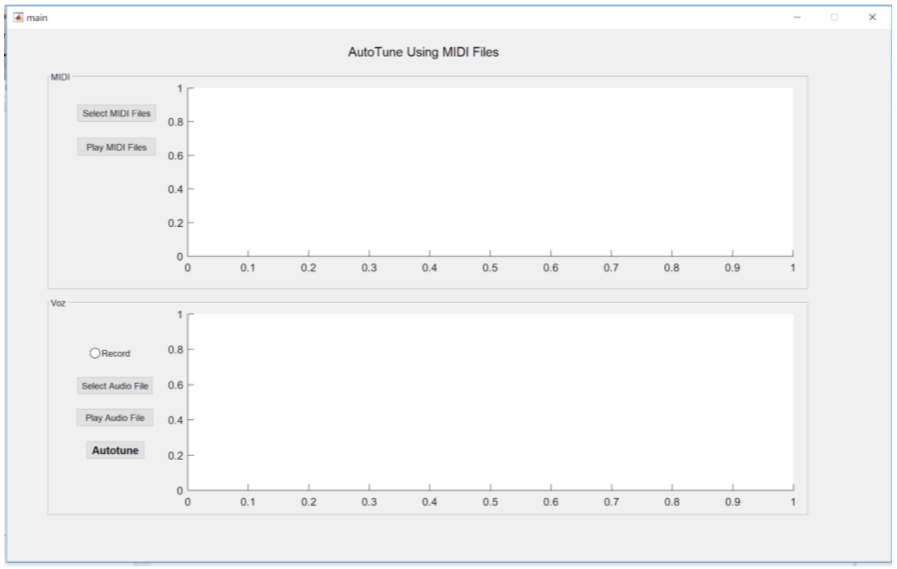
Next we analyzed the pitch of the AutoTuned file and checked whether the code worked. The output of the test file was correct and the output was an AutoTuned file. AutoTuning turned the original file into a slightly robotic sounding file, but it was correct in pitch.

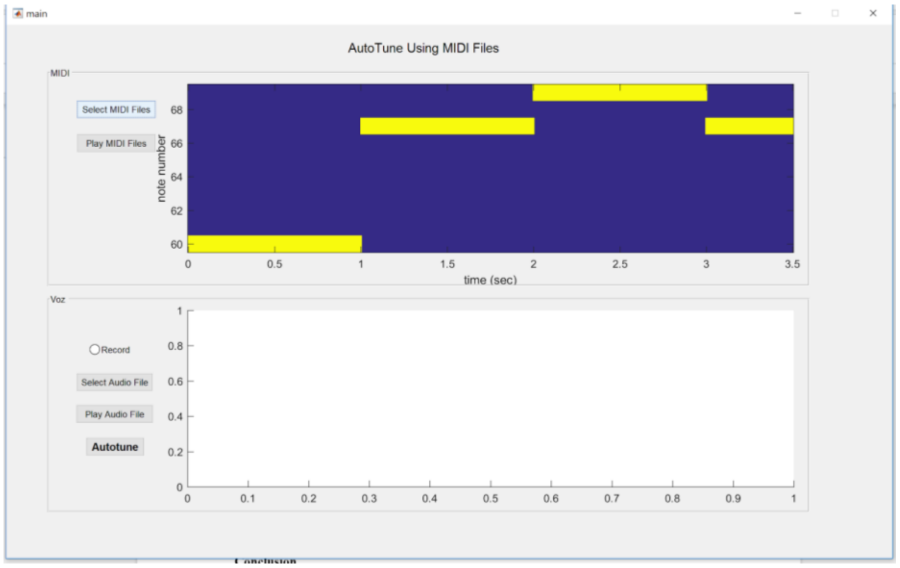
5. RESULTS

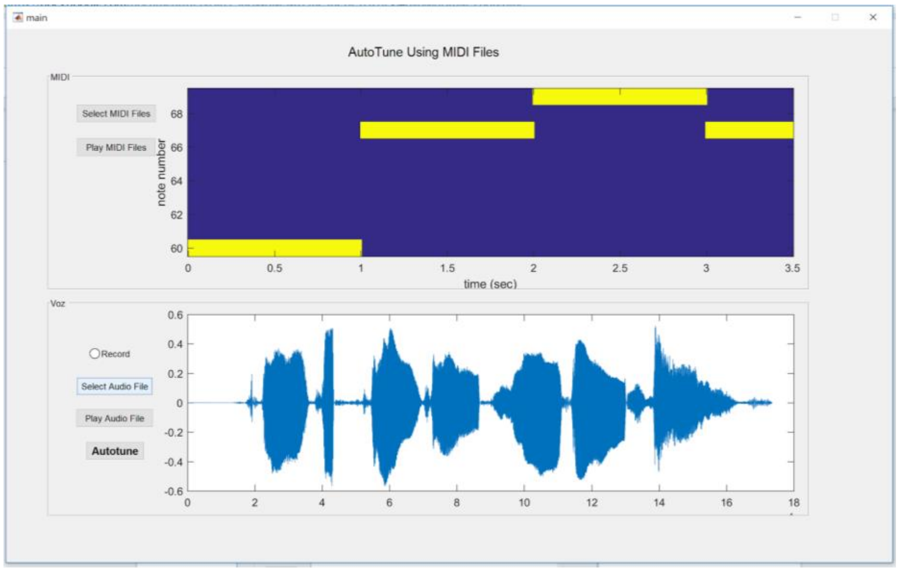
The result of the AutoTuned file is as expected. It was presented to the Professor in class.

For the demo, we created a MIDI file for the song ‘Twinkle Twinkle Star’ using the notes given in the above section. Then one of us sang the song and we used this as the base file to be AutoTuned. When we AutoTuned the song sung by us, it was pitch corrected at the output and it was clearly noticeable.

What happens is, when we listen to normal audio it is not in tune nor it has particular pitch for every note. But the autotuned file has tune and pitch of every note is as expected.







Conclusion

We compare the MIDI files and audio files and phase shift the pitch of the note which are not as desired and get it to correct pitch. We also found an issue in implementing in this way and we can use other ways too.

Future Scope

Right now we have managed to successfully AutoTune a variety of songs. But the Auto Tuned version loses the human quality when we hear it. It could sound slightly robotic to hear.

So our next step would be he capture the human quality in the AutoTuned versions of the songs. This could be achieved by using effects such Vibrato.

REFERENCES

a. http://kenschutte.com/midi
b. http://www.ee.columbia.edu/ln/labrosa/matlab/pvoc/