An Auto-Tune Harmonizer in Max

Ryan, Racicot

rracico3@u.rochester.edu

University of Rochester, Rochester, USA

The voice is not typically used as a traditional instrument is, outside of a choir or purely vocal ensemble (Barbershop, A Capella, etc.) the voice is only used to carry the main melody. Except for back-up vocalists which partially bridge the gap between using the voice as a soloist and an instrument. Traditionally there does not exist a means of a single person using their voice as a polyphonic instrument. This essay describes the design and implementation details of creating a pitch corrected harmonizer virtual instrument using Cycling ‘74’s *Max 8*.

Project Motivation

I have always loved the sound of Auto-Tuned vocals, I grew up listening to artists like T-Pain and Chris Brown who heavily relied on pitch corrected vocals to create their brand of Pop and RnB music. I thought this was a super interesting way to use the voice as a digital instrument, but without losing the essence of the human voice, but not everyone agrees with this viewpoint.

When many people speak about Auto-Tune, or more generally pitch correction, it is often carried with a negative connotation. The rationale being that if the singer is using pitch correction, that they are not a good singer, and that a more skilled singer would not need any pitch correction. Critics diminish the artistry of singers using pitch correction, yet insist on perfect, consistent, accurate pitch when listening to vocals, live and recorded. Which is provenly very difficult to accomplish.

When compared to more classical techniques of singing, most vocalists in Top 40, Rap, RnB, and other popular genres today sing with a technique that – without going into too much detail – often requires more movement of more muscles, with singers using many different techniques interchangeably to perform different vowel sounds and formants in various registers of their voice. However, in the classical style the focus of the vocalist is to minimize the amount of muscle strain and movement as much as possible, in order to have consistency across all pitches, registers, vowels, formants, etc. Because of this difference in the popular style, it is physically more difficult and demanding to have accurate pitch consistently, simply because the number of variables in the equation is much higher.

Although, thankfully in more recent years, societal acceptance and appreciation of Auto-Tune has grown substantially, to the point where nearly all recorded and performed music has some amount of pitch correction done [3].

But I digress. The motivation behind my project was to incorporate Auto-Tune style pitch correction in a new artistic manner that would not only be useful but fun to use. By incorporating live vocals with a physical instrument to increase the potential for artist expression.

Design Concept

On the broadest scale, this concept of transforming the voice into a polyphonic instrument can be realized by taking a signal of a vocal input and routing it to a set of notes to which the vocal will be pitch shifted to, and all played back simultaneously. Which can be broken down further into the following discrete steps:

1. Receive vocal input signal

2. Pitch correct the vocal signal, to create the auto-tune effect:

• Fourier analysis to determine the frequency of signal

• Calculating the distance to the closest midi note in Hz

• Shifting the vocal signal by altering magnitude values from Fourier analysis

3. Receive midi input from midi controller or keyboard, and for each note:

• Create a duplicated version of the pitch shifted vocal track

• Pitch shift the vocal signal to the desired midi note

• Manage the created polyphonic voices when midi inputs change or turn off.

4. Resynthesize all the “voices” created in the last step to a single output signal

IMPLEMENTATION

**Pitch Correction – ‘*Auto Tune’***

To achieve the aggressive pitch correction à la T-Pain, Max’s ***retune~*** object. Which performs a Fourier analysis of an input audio signal and returns: the detected frequency of the input, the closest midi note, and the vocals deviation (in cents) from that note, and most importantly the input signal pitch shifted to the determined closest midi frequency.

Some consideration had to be made for noise reduction in both the pitch correction and dealing with signals with frequency fundamentals within a reasonable range for the human voice. Otherwise, if unrestricted, would result in very harsh retuned signals from plosive consonants or the high frequencies from breathing in too close to the microphone. To reduce this, the ***retune~*** object attempts to pitch correct any incoming signal, but the output signal is only played back if it is below some reasonable threshold for the human voice to sing at. This value is set at 2000Hz (~C7), which is octaves above even the highest soprano’s range [1].

**Pitch Shifting**

Given the current frequency of the retuned vocal input signal (*Fcurrent)*, and a desired target frequency (*Ftarget*), the magnitude and direction to shift the signal in cents can be calculated by simple algebra.

*Δ = 100 \* (Ftarget - Fcurrent)*

Which is bound by **|***Δ****| ≤*** *2000Hz,* to reduce unwanted popping noises and harsh retuning of extraneous noise in the input. Meaning if a pitch shift value exceeds 2000Hz, then simply ignore the pitch shift and just play the dry signal.

**Polyphony**

Conceptually, extending the previously described functionality to polyphony is simple. For each midi note input, create a new pitch shifted signal as described above, combining all of the “voices” into a single output channel. However, because *Max* is a message based programming language, managing the state of threads playing specific notes and handling midi input is extremely complicated.

The ***poly~*** object in *Max* encapsulates pitch shifting procedure defined above, and given a list of Midi tuples, consisting of pitch and velocity values, creates a separate thread for each note. However, the object does not maintain the midi information with which each thread was created with. So as soon as the thread is created it is unknown what pitch any thread is playing. This makes it markedly difficult to turn a voice off, i.e. stop playing a note once the key has been lifted on the keyboard.

The final step requires to scale the gain of each voice output inversely to the number of voices currently being played to avoid clipping. The gain for each polyphonic voice is scaled by 1 / *N* where *N* is the number of currently active voices.

**Limitations**

However, *Max* is message based, with objects communicating with each other via either continuous signals, or individual messages. Individual objects are designed with two types of input ports, hot and cold. “Hot” inputs upon receiving input will automatically update values and calculations done in the object, where “cold” inputs upon receiving input will update values but will not perform any calculations or continue sending messages along. The *Max* objects responsible for list operations, ***zl.union, zl.unique, zl.reg,*** etc. Have both hot and cold inputs but have the added complexity of not continuing along it’s result message if the result of the calculation is an empty list. Because of this, it is impossible – using the built-in set manipulating objects – to perfectly maintain a list that can become empty in *Max*.

Because of this, the current version of this project does not perfectly manage polyphony. Because not all Midi signals to turn-off a note are properly handled. So in the current implementation, the performer must actively assign notes to voice numbers, and once any number of keys are pressed, at least one of the voices will be on at any time because of the lazy evaluation of empty lists (turning off voices) described above only being triggered when a new midi input is received, i.e. when another key is pressed down.

A partial solution to this problem encapsulated in the sub-patch ***update\_current\_notes*** attempts to maintain a set of actively pressed down midi notes by managing Midi messages. A velocity value of 0 is sent when a key is released, along with the corresponding midi value. So, a non-zero velocity value corresponds to adding a midi note to the set of active notes, and a 0 value for velocity removes that midi value from the active set if present.

In the ***poly~*** object, while each thread is actively receiving an input signal from the retuned vocal input, each thread must check that the latest note it was given to play is in the active notes set. This solves the problems of maintaining the note that each thread is responsible for and maintaining a list of actively pressed down notes.

conclusion

For this plugin to be usably functional a solution must be found for the managing polyphony. An alternated solution may be when a 0-velocity midi note event is triggered, send the message to each voice, and compare with the current midi note the voice is playing, and if the matching note is found, turn off that voice. However, this method requires much more real-time computation, and scales up very poorly as the number of voices increases. And the instrument relies on having the lowest amount of latency possible so it may be performed with live.

Also, there exists the possibility to extend the number of options available to the performer through the graphical interface. Allowing the performer to turn off the pitch correction for example, or only correct the pitch when any keys are being pressed down, in such a way that the plug-in can turn itself on and off.

References

[1] J. C. McKinney, The diagnosis & correction of vocal faults: a manual for teachers of singing and for choir directors. Long Grove, Il.: Waveland Press, 2005.

[2] H. A. Hildebrand, “Pitch detection and intonation correction apparatus and method,” U.S. Patent 5 973 252 A, October 26, 1999.

[3] S. Reynolds, “How Auto-Tune Revolutionized the Sound of Popular Music,” *Pitchfork,* September 2018. [Online]. Available: https://pitchfork.com/features/article/how-auto-tune-revolutionized-the-sound-of-popular-music/