Pitch corrector Voice Effect

Damien Ronssin damien.ronssin@epfl.ch
Jonathan Collaud jonathan.collaud@epfl.ch

***Abstract*—This report presents the implementation of a basic and real-time pitch corrector. It allows to take an out of tune monotonic melody and to correct the pitch of each note to the closest one in a given key. The method described in this paper uses windowing and processes signals in frequency domain. The algorithm achieves pitch correction over different kinds of signals in human voice frequency range. However this method modifies voice timber making it sounds more "artificial" or "robotic".**

I. Introduction

To present our implementation of a real time pitch corrector, we will start by reviewing different existing techniques of this well studied topic. We will then detail every aspect of the algorithm. To continue, we will discuss its performances and drawbacks, explore the influence of the different parameters on the quality of the output and finally propose possible improvements.

II. Review of existing techniques

The methods for pitch correction are numerous and the following review will not be exhaustive, we will only present general aspects of most used techniques.

A. Pitch detection

Pitch detection is a necessary part of any pitch corrector algorithm. Several methods exists to estimate the pitch of a signal. They can be divided between time domain and frequency domain approaches.

In frequency domain, the main technique is to first detect the main peak of the frequency spectrum and then to precise this first frequency estimation. This can be done by observing the phase difference between two consecutive windows for the peak frequency component, this gives information on the underlying precise frequency of the peak [9]. It is also possible to fit a parabola on the bins around the peak and find its maximum [5].

An example of a time domain approach for pitch detection is the use of signal auto-correlation. If the signal is periodic then for certain shifts, the auto correlation will be important. From these shifts the information of the pitch can be recovered [9].

For varying pitch signals, the above methods still works if one cuts the full signal in windows where the signal can be assumed to have an almost constant pitch. One then gets a sequence of pitch, one for each window.

B. Pitch Shifting

Pitch shifting can be achieved by many different methods that can also be categorised in time and frequency domain approaches. Once again, signals are cut in overlapping windows so that one can assume a single pitch per window.

For time domain approaches, the main idea is to modify the pitch of the signal by compressing (higher pitch) or expanding (lower pitch) it and then re-sampling to original rate. This changes the pitch of the signal and conserves the harmonic relations but it also obviously changes the duration of the sound. To get back the original length, a time stretching algorithm is applied. Time stretching consists in changing the duration of a signal without affecting its pitch [9]. A widely used time domain algorithm is the Pitch-Synchronous Overlap-Add (PSOLA) described in [9] and [6]. This algorithm is based on periodic pattern and on peak detection in the time domain, it can encounter some issues with sounds consisting essentially of high frequencies.

Frequency methods consists in taking FFT of each window, to shift the spectrum to get the desired pitch, most of the times using interpolation of the Fourier coefficients. Then the inverse Fourier transform is computed and the different windows are combined (overlap-add).

The methods used in our implementation for both pitch detection and shift are frequency domain techniques.

III. Algorithm description

Main steps of the algorithm

This pitch shifting algorithm has a classical Short Time Fourier Transform (STFT) processing structure as presented in [1]. The first step consists in reading the signal in overlapping windows. This implementation allows for 0%, 50% and 75% overlap. Then, an analysis window function is applied to each window and the Fourier transform is computed. Once one has the frequency representation, a peak detector finds the different peaks in the spectrum. The one with lowest frequency is considered to be the pitch of the window, this value will also be called the fundamental frequency or the note of the window later in this report.

Now that the pitch of the window is known, the spectrum will be scaled, via complex interpolation of the Fourier coefficients, so that the frequency of the first peak matches to the frequency of the closest note in the key specified by the user. Having now the re-tuned spectrum, a phase adjustment is done in order to maintain the phase coherency. Then the inverse Fourier transform is computed, the synthesis window function is applied and finally the different windows are combined

(overlap-add). A schematic view of these general steps is presented in Figure 1.

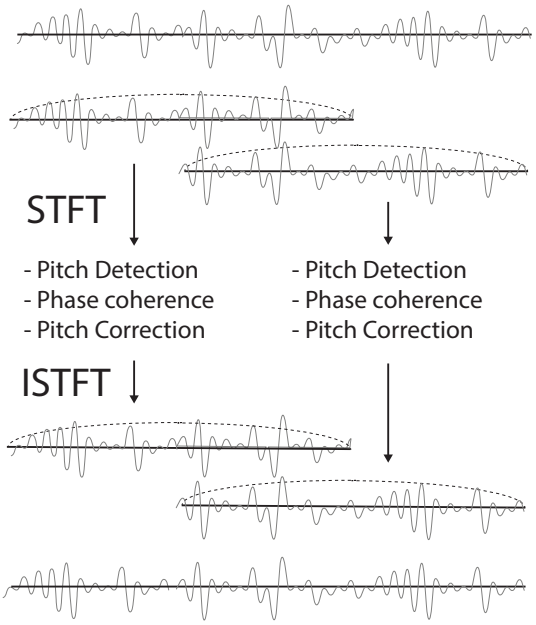

Fig. 1: Representation of the different steps of the algorithm

All this generally presented steps will now be precised.

A. Before sound processing

Some parameters have to be specified by the user for the algorithm to work. It is the case for the following values:

- Rate of the sound, usually 44100 Hz
- Window size: the number of samples in each window, usually a power of 2 between 512 and 4096.
- The overlap for the windows, either 0%, 50% or 75%.
- The FFT size, number of values in the array on which the Fast Fourier transform will be computed with (FFT_size - Window_Size) giving the number of padded 0 (this point will be precised later).
- The analysis and synthesis window type (sine, hann, hamm or rect).
- The song's key, for example if all the notes sung in the signal should be in F#, specify F#. It is possible to set the key to chromatic to get all possible notes.

An array containing all the frequencies of the notes composing the key is constructed. This is done using the fact that, for equal temperament system, each semitone is separated in frequency by a factor equal to $2^{\frac{1}{12}}$. Stated differently:

$$f(C\#) = 2^{\frac{1}{12}} \cdot f(C) \tag{1}$$

with f the function giving the frequency of a note. This holds for all notes a semitone apart from each other.

The lowest note that we chose to include in this array, and thus the lowest note that the algorithm can change a pitch to, is A1 (55 Hz).

B. Going to STFT domain

As said before, the signal is cut in overlapping windows and an analysis window function is applied. We will now detail the processing of each of these windows.

For shifting the frequency spectrum to the correct pitch, one first needs to detect the pitch of the signal. This requires a relatively high resolution in the frequency domain (the difference in frequency between two Fourier bins). This resolution is directly related to the number of samples N on which the Fourier transform is computed since:

$$\Delta F = \frac{f_s}{N} \tag{2}$$

with f_s and N respectively the sampling frequency and the number of samples on which one computes the Fourier transform.

Such a high resolution is needed for low frequencies for which the frequency difference between two notes is small : ≈ 3.3 Hz of difference between A1 and A#1, while ≈ 13.2 Hz between A3 (220 HZ) and A#3. To avoid using too long windows, one can artificially increase the number of samples for the Fourier transform by applying zero padding to the signal. This simply consists in adding 0 at the end of the time signal. The amount of zero padding is specified by the user and its influence on quality of the result will be discussed in the next part.

Once padding is done, the Fourier transform is computed using the Numpy [7] implementation for FFT on real signals: `numpy.fft.rfft`.

C. Pitch and peak detector

The pitch is needed to then compute the shift factor. This shift factor is the factor by which each frequency bin will be shifted in order to get a re-tuned spectrum. This value is critically important for the whole algorithm.

The pitch is defined as the peak with lowest frequency. The different peaks of spectrum are found using the method `find_peaks` from `scipy.signal` [2] applied on the norm of the Fourier transform array.

The parameters to adjust in order to detect the right amount of peaks are the minimal distance between peaks and the prominence of each peaks. This last parameters is a measure of how much a peak stands out from its surrounding. The right prominence choice is set-up specific, so this parameter should be adjusted if for example the microphone is changed.

The function `find_peaks` returns only the indexes of the peaks in the Fourier transform array. Thus, in order to have a more precise estimation of the fundamental frequency, a parabola interpolation (polynomial of degree 2) is done on the three values with indexes near the peak:

```
(peak_idx - 1, peak_idx, peak_idx + 1)
```

and the frequency for which the parabola is maximal is taken to be the fundamental frequency as suggested in [5]. The other peak locations will be used later to maintain phase coherency between different windows.

Knowing the fundamental frequency (pitch), one can now find the closest note belonging to the key and compute the shift factor S as follow:

$$S = \frac{f(\text{closest\_note})}{f(\text{pitch})} \tag{3}$$

D. Spectrum scaling

As said before, the frequency spectrum has to be shifted according to the shift factor S. Each Fourier coefficient pair corresponding to a frequency f_i should now be assigned to frequency:

$$f_i^* = S \cdot f_i \tag{4}$$

The term 'shift' is not entirely appropriate since each frequency bins are not shifted by a constant amount but scaled by one as it can be seen in equation 4. However, we chose to sometimes keep this formulation for simplicity.

To perform the inverse Fourier transform, one needs to have the Fourier coefficients corresponding to the original frequency bins:

$$f_i = \frac{f_s i}{N} \qquad \text{for } i = 0, 1, ..., N/2 \tag{5}$$

To get this, interpolation is used on the norm of the Fourier coefficients. Phase interpolation is avoided because of the discontinuity at π, $-\pi$. Instead the phase value of nearest bin is used. Supposing now that one has the tuned spectrum, i.e. pairs $(S \cdot f_i, y_i)$ with y_i the Fourier complex coefficient, interpolation is performed to get the values for the original frequency axis (equation 5). Piece-wise linear or cubic spline interpolations can be used in our implementation.

E. Phase Coherency

The so called phasiness problem is a well known issue for all kinds of pitch shifter as explained in [3]. The problem is the result of the combination of non phase-coherent signals. To avoid it, different phase modification techniques have been developed. In this implementation the identity phase locking method is used.

There are two types of phase coherency to maintain: horizontal and vertical. The horizontal phase coherency is the phase coherency from frame to frame, between the adjacent windows. In this case, phase coherency is necessary for the overlapping windows to add up in phase, otherwise the overlap add step would result in a important loss of amplitude (sometimes cancellation) of the output signal.

The vertical phase coherency corresponds to the phase relation between the different frequency components of the same window. It is especially important to maintain the phase relations of close frequency components. For example, the STFT representation of a pure sinusoidal signal is a single peak that can spreads on more than one bin, depending on the size of FFT window. The phase relation between the different bins composing the peak is important for recomposing the original sound from its STFT representation.

a) A basic idea to maintain horizontal phase coherency: Each frequency component of the spectrum will be shifted in frequency by:

$$\Delta f_i = (S - 1) f_i \tag{6}$$

with S the shift factor.
The difference in phase between the start of two consecutive windows is the following:

$$\Delta\phi_i = 2\pi \Delta f_i \Delta T = 2\pi \Delta f_i \frac{\text{Chunk}}{\text{Rate}} \tag{7}$$

with Chunk the number of samples between the beginning of two consecutive window. One can express it differently using the window size and the overlap:

$$\text{Chunk} = \text{Window\_size} \cdot (1 - \text{Overlap}) \tag{8}$$

The simplest idea to maintain horizontal phase coherency is to rotate the Fourier coefficient for each frequency component i by $\Delta\phi_i$. This can be done by multiplying all these Fourier coefficient of each frequency bin of index i by a phasor Z defined as follow:

$$Z_{k,i} = Z_{k-1,i} \exp\left(j \Delta\phi_i\right) \tag{9}$$

with $Z_{0,i} = 1 \quad \forall i$.

With this method, every frequency component will add up in phase but each's phase will be adjusted by a different amount. So the vertical phase coherency is not satisfied.

b) Identity phase locking: Identity phase locking [4] is a modification of the Phase locking method from [8]. Instead of adjusting a bin phase according to its own frequency, identity phase locking adjusts the phase of each bin using the frequency of the nearest peak. Thus each phase is adjusted by $\Delta\phi_{p(i)}$ with $p(i)$ giving the index of the nearest peak from bin i. This methods conserves the horizontal phase coherency since the frequency of bin i and bin $p(i)$ are close enough so that the window add up almost in phase and it also maintains vertical phase coherency for the different bins surrounding a peak.

The identity phase locking is used in our implementation and the peaks location are the one computed before with the Scipy method `find_peaks`.

F. Back to time domain

Once all this spectrum processing is done, the inverse Fourier transform is taken and the signal cropped to remove the zero padding. Then the synthesis window function is applied and the windows are overlap-added.

IV. Results and discussion

Our algorithm achieves to detect and correct pitch correctly over different types of signal, in the frequency range of human voice. However, the signal is modified in a way that does note keep the voice timber unchanged. The output of our algorithm often results in a sound that one could qualify of artificial or robotic. Although, this kind of pitch correction effect resulting in an artificial sounding voice is nowadays in modern music more of a conscious artistic choice than a real necessity for bad singers (or sometimes both at a time).

A. Choice of overlap

As said before, our implementation allows for 0%, 50% and 75% overlap. As one could have expected, the 0% overlaps gives very bad results since every transition between window can strongly be heard. The 50% is not smooth enough either. It introduces oscillations in the amplitude of the signal as it can be seen in figure 2. Thus 75% overlap has been chosen. The transition between different windows is inaudible. One can see its result on the same signal as in Fig 2 in Fig 3.

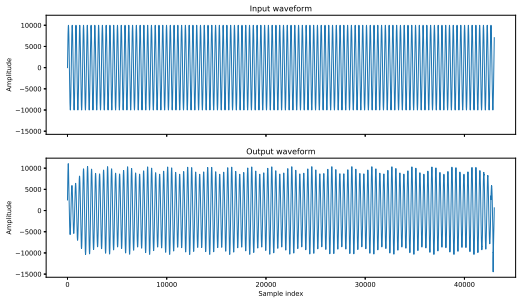

Fig. 2: Input and output of sinusoid signal (116 Hz corrected to 110 Hz) with 50% overlap Window_size = 4096

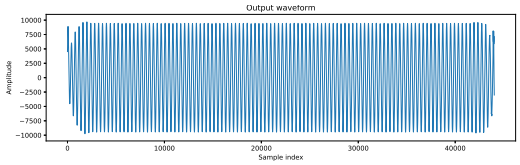

Fig. 3: Output of same signal as Fig in 2 with 75% overlap

B. Effect of zero padding

As stated before, zero padding is performed in order to artificially increase the resolution in frequency domain. A quite high resolution is needed to detect correctly the pitch of a signal and so to compute the shift factor. One has to know that zero padding do not bring any new information to our spectrum, it just increases the number of frequency bins but has other undesirable effects.

As it can be seen in Figures 4 and 5, the zero padding in Figure 5 kind of interpolates the spectrum of Figure 4 but it adds also more pronounced side lobes.

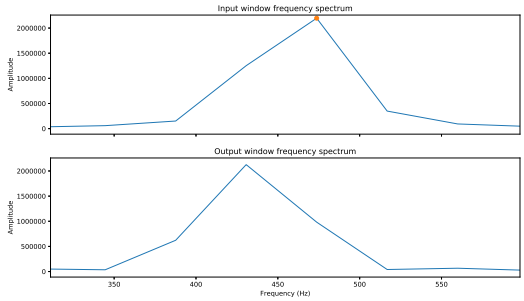

Fig. 4: Input and output of sinusoid signal (466 Hz corrected to 440 Hz) with Window_size = 1024 and no zero padding

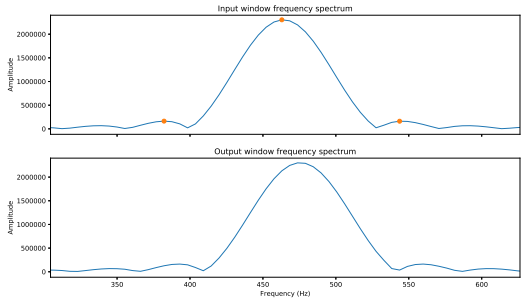

Fig. 5: Input and output of sinusoid signal (466 Hz incorrectly shifted) with Window_size = 1024 and 7168 zeros padded

The orange dots are the detected peaks. The peak with lowest frequency is considered as the peak whose frequency is the pitch of the window. In figure 5, we can see that the peak is wrongly detected on a side lobe and thus the spectrum is wrongly scaled. Figure 5 is an extreme case since 1024 is a really small window and a very important padding is used. But this shows that zero padding has an effect and that the parameters of `find_peaks` methods has to be chosen carefully taking into account the amount of zero padding and its consequences.

C. Effect on voice

This algorithm does not only modify the pitch of the signal but also introduces some artefacts. One can observe that on voice samples, it gives kind of a ”tube” or ”robotic” effect to the voice. The user of the algorithm has to choose a combination of parameters that minimizes those artefacts and maintains a working pitch correction.

After several tests, the combination that was working best on different samples (voices and pure sines) was with:

- Window size: 4096
- FFT size: 8192 (4096 padded zeros)
- Overlap: 75%

This combination gives a good frequency resolution with a limited amount of zero padding compared to the original signal size.

D. Real time issues

We have just seen that some good and flexible parameters are a window size of 4096 sample (with a rate of 44.1 kHz) and an overlap of 75%. For real-time application those parameters are different because of the delay. The delay is a major concern for real-time usage of any sound processing algorithm. In our case the delay δ can be approximated by a lower bound as a function of the window size, the rate and the overlap. The lower bound approximation comes from the fact that computation time is not taken into account since we weren't able to measure it. We only take into account the intrinsic delay resulting from the structure of our algorithm. With this we get:

$$\delta \approx (1 + \text{Overlap}) \cdot \frac{\text{WS}}{\text{Rate}} \tag{10}$$

with WS the window size, Rate the sampling rate (41.1 kHz) and Overlap the overlap factor equal to 0, 0.5 or 0.75.

For a window size of 4096 and overlap of 75%, the delay would be of 162 ms, which is way too important for any musical application. In order to have a reasonable delay, good parameters would be to use 512 or 1024 as window size with overlap of 0.5 or 0.75. With this you get a delay between 17 ms (for 512 and 0.5) and 40 ms (for 1024 and 0.75). The lack of samples per window leads to a too coarse frequency grid, so zero padding is used to get to a FFT computed on 2048 (for high voices) or 4096 values. The output with such parameters will not be as good as with larger window size but will allow real-time usage with an acceptable delay.

For real-time applications, time domain algorithms have an advantage since they do not need a lot of sample per window to get a good resolution in frequency domain. In time domain approaches, the delay can be reduced more easily without altering much the performances of the algorithm. In a frequency domain approach like our, a trade-off has to be made.

E. Comments on window functions

For analysis and synthesis window functions, we implemented these different couples of functions for both 50% and 75% overlap:

- Sine-Sine
- Hann-Rect
- Hamming-Rect

The tests done with the different function couples have shown that artefacts in high frequencies are produced when sine-sine analysis-synthesis is not used. So all our test concerning other parameters have been done using sine window functions.

F. Further improvements

One of the main limitation of this pitch corrector is that a pitch can only be corrected to the closest note in a given key. This point is necessary for a real-time implementation, however it would be convenient to be able to choose the note the algorithm corrects the pitch to in a second time, for non-live usage. Indeed it sometimes happens that the algorithm shifts the signal to an unwanted note, either because of the algorithm or because the closest note is not the desired one, so allowing for correction of this unwanted behaviour in post-processing would be a great improvement.

A recurrent problem with our implementation occurs during the pitch detection step. The right choice of the peak prominence threshold argument of `find_peaks` strongly depends on the input amplitude and also on the amount of zero padding. Pitch detection is a crucial step in the algorithm since if the pitch is wrongly estimated, then it is wrongly corrected and the output could be less in tune than the input. The use of an adaptive algorithm based on the signal properties, for example the harmonic relation between different frequency component, could be an idea of improvement.

State of the art pitch corrector have parameters allowing for a more natural sounding result. For example, some have a "Response time" parameters to adjust. This sets the time that the algorithm will take to bring the signal to the correct pitch. In our implementation, the shift is instantaneous and this results in sometimes quite rough transitions that sound artificial. Adding such a parameter would be possible and a good improvement of our implementation.

V. Conclusion

Real time sound processing algorithms are often implemented in C/C++ allowing for fast computations. This pitch corrector was fully implemented in Python 3 and achieves real-time pitch correction thanks to Numpy operations on arrays to get a fast enough processing.

This "auto-tune" algorithm uses a simple overlapping windows structure, goes in frequency domain for each window for both pitch correction and shift. For phase coherency, which is the main issue with those kind of algorithm, it uses identity phase locking method.

The algorithm achieves pitch correction but introduce some artefacts that modifies the voice timber.

References

[1] Chirstof Faller and Mihailo Kolundžija. *Audio and Acoustic Signal Processing*. 2019.

[2] Eric Jones, Travis Oliphant, Pearu Peterson, et al. Scipy: Open source scientific tools for python, 2001.

[3] Jean Laroche and Mark Dolson. Phase-vocoder: About this phasiness business. In *Proceedings of 1997 Workshop on Applications of Signal Processing to Audio and Acoustics*, pages 4–pp. IEEE, 1997.

[4] Jean Laroche and Mark Dolson. Improved phase vocoder time-scale modification of audio. *IEEE Transactions on Speech and Audio processing*, 7(3):323–332, 1999.

[5] Jean Laroche and Mark Dolson. New phase-vocoder techniques for pitch-shifting, harmonizing and other exotic effects. In *Proceedings of the 1999 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics. WASPAA'99 (Cat. No. 99TH8452)*, pages 91–94. IEEE, 1999.

[6] Gareth Middleton. Time domain pitch correction. In *ECE 301 Projects Fall 2003*. OpenStax CNX, 2003.

[7] Travis Oliphant. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006–.

[8] Miller Puckette. Phase-locked vocoder. In *Proceedings of 1995 Workshop on Applications of Signal Processing to Audio and Accoustics*, pages 222–225. IEEE, 1995.

[9] Udo Zölzer. *DAFX: digital audio effects*. John Wiley & Sons, 2011.