Auto-Tuning Using Fourier Coefficients

Math 56

Tom Whalen

May 20, 2013

The Fourier transform is an integral part of signal processing of any kind. To be able to analyze an input signal as a superposition of infinitely many signals of different frequency allows for manipulation of the coefficients and, after completing the inverse Fourier transform, interesting effects on the initial signal. Thus, in my project I focused on creating pitch correction effects as seen in software Auto-Tune, and in this paper I will talk about the process of creating a Matlab code that performs this effect.

The Fast Fourier Transform algorithm is effective at breaking down mathematical functions into a Fourier series of different frequency complex exponentials and the corresponding coefficients. However, when analyzing audio signals it is impractical and inefficient to take one Fouier Transform of an entire audio clip; for a band-limited signal, most of the $44100t$ Fourier coefficients will contain no imformation about the original signal and there is no way to tell which frequencies are dominant at different times in the signal. Thus, the way that audio signals are most often processed is using something called the Short-Time Fourier Transform (STFT). The basic idea of the STFT is that one can window the actual signal multiple times and take the FFT of the resulting modified input. The windowing process is done simply by multiplying the signal by another function that is mostly zero and has a maximum of 1. This will kill off the signal at other times that are not of interest and allow for getting the frequencies at a certain point in time via the FFT.

There are several types of windowing functions that can be used in the STFT process. The most commonly used set of windows are of the form

$$\alpha - (1-\alpha)\cos\left(\frac{2\pi n}{N-1}\right), 0 \leq n < N$$

where α is a constant that determines the shape of the function and N is the length of the windowed signal. The windowing function that I used most in my analysis was the Hanning window, which is the above equation with $\alpha = .5$.

Some other possible widowing functions are the Hamming window ($\alpha = .54$) and the square function ($\alpha = 1$). One can even form functions like $\cos^3\left(\frac{\pi n}{N-1}\right)$ where $0 \leq n < N$ that are very similar in shape to the Hanning window but have

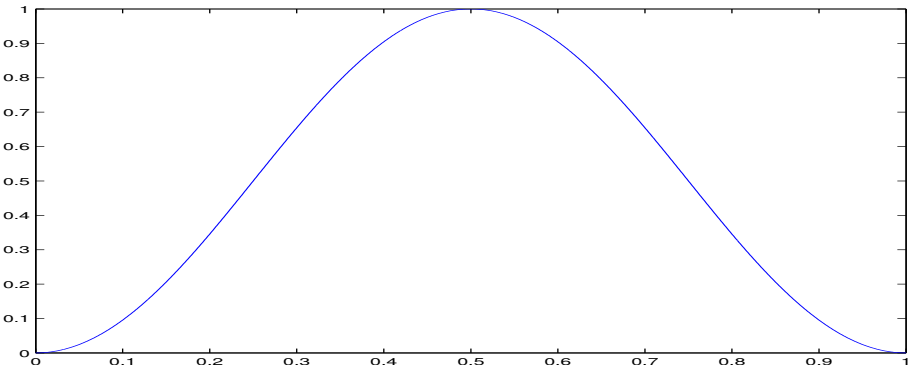

Figure 1: The Hanning window: $\frac{1}{2}(1 - \cos(\frac{2\pi n}{N-1}))$, $0 \leq n < N$

a more defined peak near $N/2$. Windows like the Hanning are much prefered to the square window because the jumps in the square window cause the Fourier coefficients to decay much more gradually than those obtained via one of the smooth windows. However, when using the cosine-based window, it is necessary to have a jump size between windows that enables full reconstruction of the original signal. This reconstruction condition is

$$f_j w(j - t_0) + f_j w(j - t_1) + ... = 1$$

which means that each spectral frame will be weighted evenly. For the square, the obvious choice for the jump size is N, the number of samples in the windowed signal, which will prevent any component of the signal from being processed more than once. For the other windows, the jumpsize is $N/4$, but this will cause the signal to be interpreted as having twice the actual value, hence a scaling factor of $1/2$ to recover the signal.

Now this brings us to the actual form of the STFT:

$$\hat{f}_{m,t} = \sum_{j=0}^{\infty} w(j - t) f_j \omega^{-mj}$$

In practice the result of the running the ST-FFT algorithm is a matrix with each row representing the Fourier coefficients of the signal at some point in time. The final step in the STFT process is choosing a suitable value for N, the number of samples of the signal included in each window. For a typical sampling rate of 44.1 kHz, the most common N is 1024, which is large enough to ensure that the Fourier coefficients will decay and there will be no aliasing effects. It is also a small enough that the time resolution of the signal is decent, as there will be roughly 43 spectral frames per second of signal. Here is a generic audio signal and its STFT displayed as a colorscale image with time on the vertical axis and the frequency on the horizontal.

To invert the STFT matrix, one must loop through all of the spectral frames and essentially add up all of the resuting signals that are produced by each set

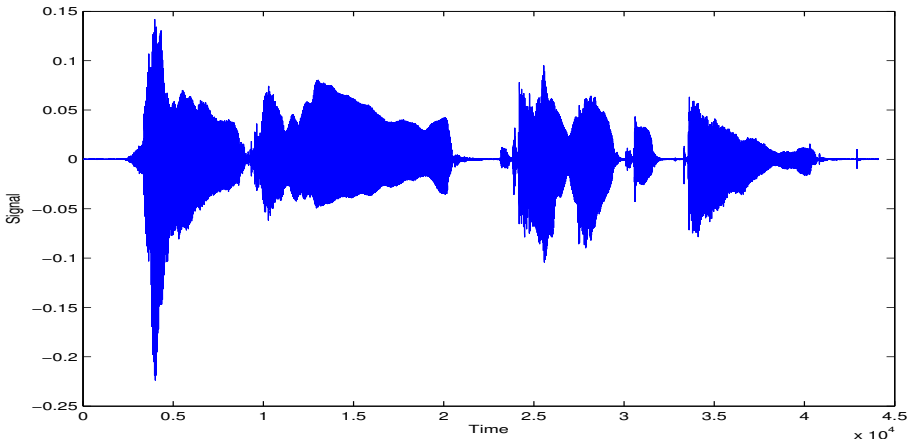

Figure 2: The input signal

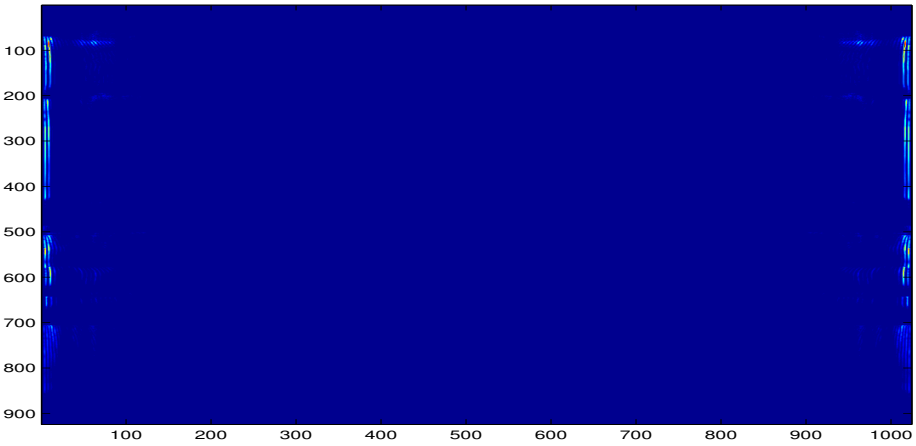

Figure 3: Its STFT on a color-scale map

of coefficients, which is commonly referred to as the overlap-add method of reconstruction. Shifting by the jumpsize each iteration, multiplying the ISTFT output by the window, and then adding it to the total output signal will perfectly reconstruct a signal from its set of STFT coefficients.

The problem with the STFT is that, for any given spectral frame, it doesn't tell us the instantaneous frequencies due to the fact that there isn't much energy resolution. The solution to this problem relies in the fact that a complex coefficient $a + ib$ can be written in the polar form βe^{ϕ} where $\beta = \sqrt{a^2 + b^2}$ and $\phi = \tan^{-1} \frac{b}{a}$. The ϕ term acts as a phase of the signal, due to the shift of a particular frequency in the windowing process. Looking at the term in the Fourier series for which the coefficient is $a + ib$, we can now write this as $\beta e^{i\phi} e^{im} = \beta e^{i(\phi+m)}$. The new interpretation of this is that the new amplitude of the wave is the magnitude of the original coefficient and the new frequency is actually the old frequency shifted by the phase ϕ. Thus, by converting to polar coordinates we can keep track of actual values of the frequencies in an audio signal rather than those determined by the length of the DFT matrix.

The implementation of the phase vocoder is illustrated by finding the change in phase for a specified energy bin between successive spectral frames, as shown in the following equations:

$$\Delta\omega(k,t) = \frac{\phi(k,t) - \phi(k,t)}{\Delta t} - \omega(k,t)$$

$$\Delta\omega_{wrapped}(k,t) = [(\Delta\omega(k,t) + \pi)\%2\pi] - \pi$$

$$\omega_{new}(k,t) = \Delta\omega_{wrapped}(k,t) - \omega(k,t)$$

Because the phases are wrapped from -π to π, we can't simply calculate the new frequency by finding the phase difference and adding it to the frequency denoted by k. We have to instead find the phase difference, subtract the expected value, make it positive and mod it with 2π and finally add it with the frequency value of the bin to get the true frequency of the signal.

Now that we know exactly which frequencies we are dealing with, it is possible to directly manipulate the coefficients. Pitch correction is carried out by first finding out the frequency values for the musical notes and making them target bins for the signal coefficients. Once, this has been done the next step is to find the dominant overtone of the input signal. Since pitch correction is employed mainly for human singing, there will normally only be one tone that dominates the spectrum of Fourier coefficients. It is also necessary to convert the frequencies, which have now been processed by the phase vocoder, into Hertz in order to correctly coorelate the target bins, which is given by the equation:

$\omega_{Hz}(k) = \frac{k(\text{sampling rate})}{\text{length of window}}$ where k is the bin number and ranges from 1 to the length of the window

Once this has been done, the next step is to loop through all of the spectral frames, identifying the dominant tone in each one, and pushing each into the target bin of the nearest musical note. However, because Matlab's implementation

of the FFT doesn't allow for changing the actual frequencies that correspond to the coefficients, I had to effectively change the frequencies by altering the values of the amplitudes. Since we want to rebin the dominant tone of the signal, we want to go from frequency m to frequency $m + \phi$, where ϕ is some phase factor that moves this frequency to the desired one. Now writing out the series term as:

$$\hat{f_m} e^{i(m+\phi)} = \hat{f_m} e^{i\phi} e^{im}$$

This tells us that we should multiply the Fourier coefficients by a factor $e^{i\phi}$ that depends on the distance from the nearest desired frequency. This will shift all the frequencies to output the pitch corrected signal that should, in theory, produce a much more euphonic sound than the original clip.

There are a few restrictions, however. For a real-valued input like an audio clip, we have the relation that $f_{-m} = f_m^*$, which means that the coefficient with the maximum amplitude will appear twice. Thus, we need only analyze the coefficients f_0 to $f_{N/2}$, which will give us the correct difference between the actual frequency corresponding to the coefficient and the goal musical note. Furthermore, there isn't enough energy resolution to separate the lower frequencies present in the signal. With a window size of 1024 and a sampling rate of approximately 43 Hz, the information of the entire lowest musical octave, with values below 43 Hz, is all contained in the first Fourier coefficient, so it is impossible to resolve out which notes are dominant in the signal. Thus, it is best just to avoid this difficulty and begin analyzing the dominant tones starting with the second coefficient.

This all sounds easy enough in theory, but the actual implementation was a bit more difficult. After a long amount of time put into making the code work, I wasn't able to get a functional auto-tune script up and running. Though the computational aspect of this project was not completely fulfilled, the theory behind pitch correction programs is very interesting creates a basis for other neat effects to implemented on audio signals.

References

[1] Boulanger, Richard; Lazzarini, Victor. *The Audio Programming Book*. Massachusetts Institute of Technology, 2011.