# Making a Phase Vocoder in Python

Cypress Frankenfeld
Jennifer Wei
Computational Signal Processing, Fall 2014
https://github.com/cypressf/phase-vocoder

## I. Introduction

A phase vocoder is an analysis and synthesis system that can scale both frequency and time using phase information from an audio signal to lengthen, compress, or change the pitch of the audio. Phase vocoders are quite common as they are used in modern music production (i.e., autotune) and are now part of YouTube videos (i.e., the speed controller). In this case study, we examine the importance and uses of phase vocoders, how they work, and how to implement them.

## II. Purpose

This phase vocoder is for altering the speed (stretching or compressing) of an audio signal while maintaining the original pitch. In order to scale time while maintaining pitch, it is important to divide a signal into small segments, estimate the frequency accurately for each segment, and add the segments while spacing them farther apart than when they were created. This requires high-resolution in both time and frequency.

Unfortunately, using the typical method of taking the FFT of segments of the signal, there is a direct tradeoff between time-resolution (segment size) and frequency-resolution.

> Frequency resolution = (sampling rate) / (segment size).

A phase vocoder is able to get around this tradeoff by using phase information to make more precise estimates of the frequencies of segments with very small time windows.

## III. How It Works

It works by taking an audio signal and a windowing function and multiplying them to get small segments of the original signal. From there, each windowed segment is analyzed. It takes the FFT of each segment and extracts magnitude *and* phase information. At this point, a typical best estimate of frequency peaks would be limited to the frequency resolution of the FFT, but the phase vocoder uses the phase information to improve that estimate.

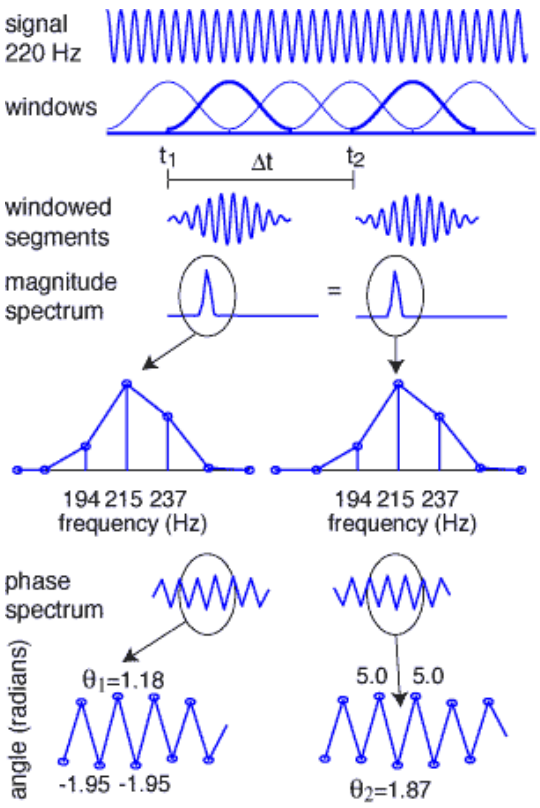


Figure 1. Above is a visual representation of how a phase vocoder works.

In Figure 1, we see an example signal (220 Hz wave), from which two segments are extracted. Next, you see plots of the magnitude and phase of the spectrum of each segment. The highest magnitude frequency is 215 Hz, which is a little off the true frequency (220 Hz). If you look at the phase spectrum at that frequency 215 Hz, you can observe a change in phase of $\theta_2 - \theta_1$ = 1.87 rad - 1.18 rad = .69 rad. We also know the difference in time between the two segments, $\Delta t = t_2 - t_1 = 0.023$. We know that the 2*pi*frequency*$\Delta t = \theta_2 - \theta_1$, so we can solve to get frequency*n = ($\theta_2 - \theta_1$ + 2*pi*n)/ (2*pi*$\Delta t$). Thus, we can pin frequency to an integer multiple of 2pi.

The first few of these frequencies are 47.7472, 90.8136, 133.8800, 176.946, 220.0129, and 263.0793 Hz. Matching it with the estimate from the magnitude spectrum, 220.0129 Hz emerges as the best estimate. This is considerably closer to the true frequency (220 Hz) than the magnitude spectrum was.

## IV. Our implementation

We implement the phase vocoder by reading a .wav file, and then chopping it into segments with a hanning window. Each segment is spaced apart by a “hop length.” The resulting segments are then stitched back together, this time separated by an output “hop length” with a different size. The ratio between the two hop lengths determines whether or not the vocoder compresses, maintains, or stretches the audio.

Using the spectrum magnitudes in find_peaks.py we find all the peaks and identify the biggest peaks that are above a defined threshold for a peak amplitude and also less than or equal to the maximum number of peaks we want to evaluate, for example, 50. In Figure 2, you can see a portion of the spectrum of a segment, and some peaks that have been identified in red.

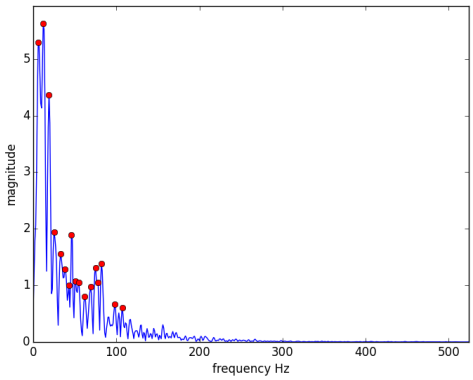


Figure 2. Some peaks identified in a spectrum of a segment by find_peaks.py

After we have identified the frequency of our 50 largest peaks, we find a better estimate for each frequency by using phase information, using the method described in Section III. Next, we create a spectrum for our estimates and our phase information, and we recreate a wave using this spectrum. Using all these new segments, we reassemble the signal time-scaled by a factor given by the ratio of hop lengths to create the altered audio signal.

To run the phase vocoder on sample audio, type

```
python phase_vocoder.py
```

This should create an output file called “yoursoundchanged.py.” This file will slow have a tempo that is ⅔ of the original, but without a frequency shift.

Our Python implementation of a phase vocoder is inspired by the Matlab phase vocoder written by William Sethares, which we have also included (slightly modified) in our repo.

## V. References

[1] Antares Tech. “How Does Auto-tune Pitch Correction Work?” *How Does Auto-tune Pitch Correction Work?* Physics.org, n.d. Web. 09 Dec. 2014.

[2] Dolson, Mark. "The Phase Vocoder: A Tutorial." *The Phase Vocoder: A Tutorial*. Jens Johansson, 23 Apr. 2014. Web. 09 Dec. 2014.

[3] Sethares, William A. "A Phase Vocoder in Matlab." *Phase Vocoder in Matlab*. N.p., n.d. Web. 09 Dec. 2014.

[4] "Theory behind Autotune/vocoder." *Theory behind Autotune/vocoder*. Stack Overflow, May 2012. Web. 09 Dec. 2014.