A Real-time Channel Vocoder AudioUnits Plug-in Emulating a Vocal Harmonizer

Sam Drazin
Music Engineering – University of Miami
MMI 505: Advanced Audio Signal Processing

***Abstract*—The goal of this project was to create a software plug-in which emulates the characteristics of a vocal harmonizer. To accomplish this goal, a Channel Vocoder was implemented within the AudioUnit Effect framework. The Short-Time Fourier Transform of a windowed signal was taken of two input streams audio: one of a carrier, and one of a modulator. The frequency-domain signal of the modulator (usually vocal) was broken up into banks of bins, and a weighted average of their magnitude calculated. This weighted average was used to modify the amplitude of the corresponding frequency bins of the carrier (typically a sustained synth instrument). The modulated frequency-domain signal was then returned to the time-domain and output from the plug-in.**

This report will describe the history of the vocoder, as well as it's current use in popular music as well as audio engineering. An in-depth discussion of the channel vocoder and it's implementation will be included as well. Lastly, an overview of the development process as well as encountered obstacles will also be discussed.

***Index Terms*—Harmonizer, Channel Vocoder, AudioUnits, Short-Time Fourier Transform**

I. Introduction

WHETHER or not one is familiar with the functionality of a *vocoder* or *harmonizer*, the sounds of vocal synthesis have been utilized in music for many years. Pop songs since the 1970s have featured such effects, and contemporary uses continue to stretch the possibilities of how vocals can be used to enhance the texture of music. Inspired by one of many songs that utilize a vocal harmonizer in such a manner, this project was chosen to emulate the effects of a vocal harmonizer through the implementation of a channel vocoder. With a brief introduction to various aspects of digital signal processing (DSP), the process of implementing a channel vocoder is described in the following report. Before covering the methods of implementation, however, the basics behind a channel vocoder must be covered, and tributes paid to those who developed these (and varying) concepts and systems.

II. Background

A. The Vocoder

The term *vocoder* (or **vo**ice en**coder**) describes a set of analysis/synthesis techniques used to process a voiced signal. Traditionally, the encoding process involves passing an input signal through a series of multi-band filter banks. Envelop followers store the envelops for each of the bands of the filter bank, which are then sent to the decoder. The original signal can be intelligibly reconstructed given a known filter bank configuration by both the encoder and decoder.

A.1 History

First developed by Homer Dudley in 1928, scientists in Bell Labs were experimenting with methods of encryption for voiced signals. Figure 1 shows a diagram of Dudley's early design of the system. Telecommunication needs required the secure and rapid transfer of messages over a radio signal. The use of a vocoder aided this need, reducing the amount of information needed to transmit an interpretable signal. The signal itself was not sent; instead, the envelops of each band of the filter banks were sent. Once received, a reconstructed signal was produced from the envelops received. Vocoding signals allowed for a variance of encryption methods; altering the format, order, size, or depth of the envelops transferred would render a worthless signal for any receiver without the proper decoding scheme to interpret the incoming messages. For this reason, this technique would later serve to secure communication transmissions made during the second World War.

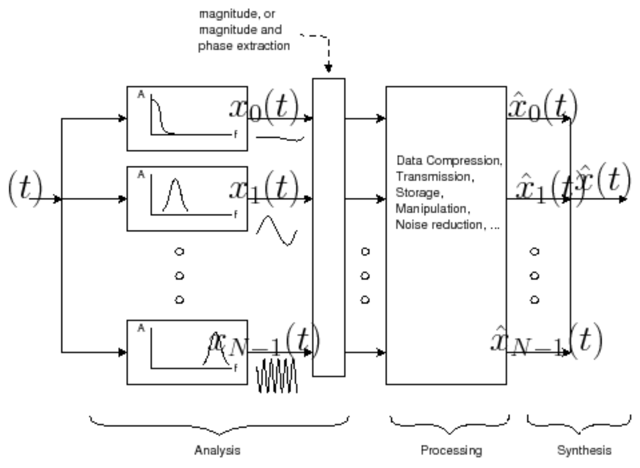
magnitude, or magnitude and phase extraction
$x_0(t)$
$x_1(t)$
$x_{N-1}(t)$
(t)
Data Compression, Transmission, Storage, Manipulation, Noise reduction, ...
$\hat{x}_0(t)$
$\hat{x}_1(t)$
$\hat{x}(t)$
$\hat{x}_{N-1}(t)$
Analysis
Processing
Synthesis

Fig. 1: Diagram of Dudley's design of a vocoder.

When decoding the output sent from a vocoder, the reconstruction process is simply the reversed encoding process. An impulse must excite a filter bank with the received output's envelop (changing over time), and this will effectively recreate a facsimile of the original signal. This result, however, is far from a perfect reconstruction of the original signal. After a vocoder encodes a voiced signal, all that is kept is each band's envelop; the original signal itself is discarded. Lost with the original signal is the unique excitation created by the sound source that was being vocoded. To perfectly reconstruct the original signal, the precise excitation that created the sound in the first place would need to be sounded and run through the envelope train received and inverted by the decoder. This was not only unpractical, but unnecessary in achieving successful transmissions of intelligible messages over a radio line. Although the reproduced signal from the decoder was commonly driven by either a tone generator or other sound source, the effect of a vocoded signal became desirable to musicians and those interested in effected sounds.

In the 1969, the first vocoder designed for musical purposes was build by Bruce Haack, a Canadian composer and early pioneer of electronic music. Named *Farad* after physicist Michael Faraday, his design was programmed with touch and proximity relays. Haack used his vocoder to produce several albums, the wake from which would inspire Robert Moog to develop a model of his own. The Moog vocoder used a carrier signal from the built-in Moog synthesizer, and featured an attached microphone input for the modulator signal. Making cameo appearances on several electronic music albums, the vocoder would soon be featured on many pop albums, including those of Neil Young, Phil Collins, and Madonna. The vocoder has maintained a solid presence in popular music to this day.

III. Proposed System

The proposed goal of this assignment was to create a plug-in that emulates a *vocal harmonizer*, or an instrument-controlled voice duplicator that takes an input vocal track and maps copies of the voice to the pitches played by the instrument (MIDI keyboard). In actuality, a vocal harmonizer would perform either a pitch-tracking or pitch-correction algorithm on the vocal track, and map duplicated vocal tracks to notes according to the pitches played on the keyboard. These were features that were secondary to the initial goal of producing a system that would alter a vocal input signal with the characteristics of a polyphonic source, preferably driven by a synthesizer.

At first, much focus went into researching the requirements of implementing a *phase vocoder*. A phase vocoder effects a digital signal in the frequency domain by altering its phase and magnitude components to produce the effect of pitch shifting or time expansion/compression. Although numerous instances of this algorithm exist and are available through open-source projects, this method tends to be computationally expensive, and often leaves audible artifacts to the synthesized signal. Nevertheless, plans were explored to utilize this process to achieve emulating a harmonizer. The initial plan involved duplicating a vocal input signal for every note that was held on a keyboard, and pitch-shifting each copy to sound at a corresponding pitch. This technique, although computationally intensive, would be extremely effective in maintaining the formant structure of the input vocal signal, or the characterized spectral envelop of an excitation. As several algorithms were experimented with, however, this approach began to seem excessive for accomplishing a goal that could be satisfactorily achieved in several less strenuous ways.

Simpler forms of a vocoder were explored, and upon reading through the mechanics of a channel vocoder, sights were set to experiment with a real-time implementation. An algorithm written by Bill Sethares was tested in MATLAB, and its effects were very similar to those sought after, as well as being a relatively light-weight process. The diagram in Figure 2 is an oversimplified graphical representation of the behavior of a channel vocoder, referencing its similarity to a vocal harmonizer. The rest of the development phase would involve porting this algorithm to C++, as well as implementing it as a part of the AudioUnit plug-in architecture. Although computationally much simpler than other algorithms, understanding the inner-workings of a channel vocoder is essential in the scope of this project.

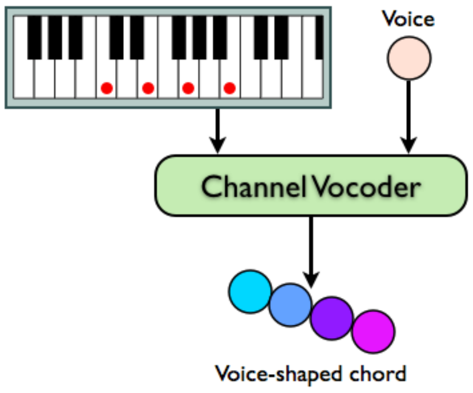

Fig. 2: Overview of the input/output signals to a harmonizer implemented with a channel vocoder.

A. *Channel Vocoder*

A channel vocoder describes a particular implementation of a vocoder. The function of a channel vocoder is to essentially map the spectral amplitude of a *modulator* signal (typically voice) to that of a *carrier* signal (synthesized patch of one or more tones). With this effect, one could emulate the sound of a synthesized instrument effectively "speaking" with a similar spectral timbre of a human's voice. Linked here is an example of this effect with the carrier of a pipe organ, and a modulator of someone reciting the United States Pledge of Allegiance. Although the output of this effect is powerful and interesting, the process by which such a signal is produced does not require immense computational power. Thanks to the growing availability of personal computers and computing devices, any machine with the ability to perform FFTs on audio files would be able to reproduce this effect.

A.1 The Fourier Transform

As an implementation of a basic vocoder, the input signals are analyzed and re-synthesized into an output signal. In the analysis stage, two signals required to produce output; both a carrier and modulator are read into the system. Once both signals are acquired, they are both transformed to the frequency domain via the Fourier Transform. When taking a signal to the frequency domain, there are several characteristics of the resulting signal which impact the manner in which the signal is handled. A closer look at the Fourier Transform will better illustrate this point.

Transforming a signal into the frequency domain is the process of taking a *Time Domain* signal, and representing its contents in terms of its frequency content over a certain duration. The frequency content of a signal is what gives sounds the timbre that individualizes them from other sounds. Your grandmother's voice sounds different than that of your nephew because of their unique harmonic structure, or strengths of frequency content in each of their voices. Despite the fact that both are voices, differentiating the two is trivial and automatic to a familiar listener. Inspecting a plot of the frequency-domain magnitude of human speech (see Figure 3) shows the relative strength or magnitude (represented on the y-axis) of frequencies (denoted on the x-axis) present in just two spoken words. The lowest peak would commonly be referred to as the fundamental frequency, or lowest sounding frequency of a complex signal. Peaks following to the right of the fundamental illustrate the harmonics of the signal, or integer multiples of the fundamental. Although all aural excitations exhibit similar structure with a fundamental and harmonics, the strength and prominence of each harmonic contributes greatly to the perceived timbre of each sound.

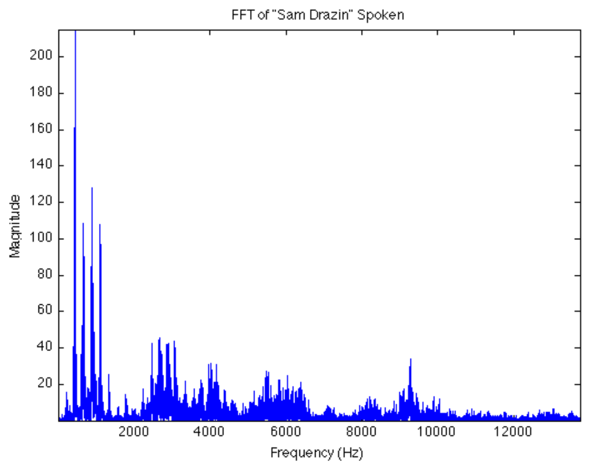

Fig. 3: Magnitude plot of an FFT of "Sam Drazin" spoken.

Taking advantage of this fact, the channel vocoder will summarize the magnitude of a modulator's spectrum (similar to that of the plot in Figure 3), and modify the spectrum of a carrier with the envelop of the modulator. In order to obtain a summary of the magnitude spectrum, several parameters of the channel vocoder dictate the number of channels that will be utilized to represent the frequency content of each signal. Each *channel* will represent a collection of frequency bins of the transformed frequency-domain signal. A bin will typically contain several frequencies itself, implying that the larger the number of channels, the higher the resolution of the synthesized signal. The number of channels, however, must be less than the number of bins produced by the Fourier Transform; otherwise, the resultant frequency bins would need to be split up via interpolation into smaller bins, which would be computationally expensive, as well as introducing artifacts of creating resolution in the signal that was not there to begin with.

The frequency-domain signal is split up into the number of specified channels, and a weighted average is calculated of the magnitudes of each channel's bins. What is left of the signal is a collection of weighted averages which will then be used to alter the magnitude of the corresponding channels of the frequency-domain carrier signal. Modified with the corresponding amplitudes, the carrier signal will maintain a magnitude spectrum heavily influenced by that of the modulator. Once returned to the time-domain via Inverse Fourier Transform, the carrier signal will exhibit a similar timbre of the modulator, while maintaining its original content of the synthesized tone or tones. Returning to the time-domain completes the channel vocoding process, and a sample of the resultant signal (mentioned above) can be heard at this link.

There were many concerns taken in implementing this process to pairs of signals that were not discussed in the description above. These difficulties were concerns of their own, and required experimentation and development completely aside from the recreation of the algorithm used to perform the channel vocoding.

B. AudioUnits

In order to successfully perform the channel vocoder algorithm described above, several decisions had to be made with regards to the forum of synthesis. Executing this effect was a trivial offline process; reading in two files and processing them entirely in one run through the length of each clip was a programmatically simple approach. A test environment was constructed in Xcode, and with the help of two third-party C libraries, (*libsndfile* and *FFTW*), the tasks reading and writing files, as well as performing the Fourier transform were subsidized greatly by each of the mentioned API's provided methods. The test environment did not embody the ideal behavior of a commercial plug-in, however, in that it needed to read in two files, run the algorithm, and then output a processed file. Though the program was fast, it was not able to process audio in (effectively) real-time, or from a live input-stream of audio. To achieve this goal, the AudioUnits architecture was used.

AudioUnits (AU) is a plug-in architecture supported in Mac's CoreAudio framework, and provides a template for low-latency processing of audio-streams, amongst other features. Transferring the existing algorithm for the channel vocoder to this platform would be the largest struggle encountered for this assignment. To better explain the obstacles faced in AU, a brief understanding of the AUEffect template is needed.

As mentioned previously, a majority of the development of this project was completed in Xcode, Mac's IDE and host of several AU templates. Two primary templates are offered that take advantage of the AU libraries: the AudioUnit Effect, and the AudioUnit Instrument. An unfortunate source of confusion and wasted effort came from the selection between these two templates, and in the decision of which would best serve the needs of this plug-in. Tempted by the support for MIDI note capture and handling, the AUInstrument template was heavily explored before deciding that an AUEffect template would more than suffice for the needs of a channel vocoder. Exploring the AUEffect template in more detail, the code provided can be described in two main sections: *initialization*, and *processing*. The initialization sections of the template provide pre-built constructors and destructors along with a class instantiation of a subclassed AUEffect. This section holds the declarations any members and methods needed by the developer other than the main *process* function, which handles the actual DSP work of the plug-in. The process function is structured to handle incoming audio in buffer-sized chunks, cycling through samples one-by-one. The size of the buffer that the process function receives is decided by the host program, and is commonly a power of two ranging from 64 to 4096 samples.

The structure provided by this template provides much of the needed scaffolding of a working plug-in, saving many hours of coding a similar construct. It falls short, however, in providing several vital needs of the chosen implementation of the channel vocoder. First, the processing performed by the channel vocoder algorithm would function on frames of audio at a time (eg: one buffer's worth of audio), not on a sample-by-sample basis. To achieve this, a buffering scheme would need to be built around the existing IO routing. Second, a channel vocoder needs two input streams of audio to render one output channel; by default, the AU templates support only mono effects. To handle more than one channel in or out, custom channel allocation would need to be handled.

B.1 Buffering

The channel vocoder algorithm explained in the previous subsection discusses processing both a carrier and modulator file with the Fourier Transform. The same technique was used in the final implementation, but instead of processing two entire files, small segments (or frames) of the incoming files were processed continuously. The existing structure of the AUEffect template gave access to only one buffer of samples at a time. This was an impeding restriction because processing one buffer at a time through the frequency domain implied some-what harsh changes to the incoming files; without the ability to process adjacent frames with an overlap-and-add scheme, the transition from one buffer to the next would render an audibly unacceptable transition manifested as periodic clicks in the output from the plug-in. For this reason, access to more than one frame at a time was needed. Secondary buffers twice the size of the process-buffer were declared, and as samples came into the default process buffer, they were stored into the secondary buffers.

Processing buffers of audio at a time does not change the overall effect of the channel vocoder; nevertheless, care was taken in ensuring that a perfect overlap-and-add scheme was utilized to properly analyze and synthesize the incoming audio stream. A 50% overlap was arbitrarily decided for simplicity of implementation and debugging. When the plug-in initially loads, the first buffer of audio loaded into the secondary (or input) buffer is processed once by itself. From this point on, every buffer received incurred two processing calls; one for the overlapped segment between it and the old buffer, and then one for the new buffer exclusively. The diagram in Figure 4 best illustrates the processing calls for each incoming buffer of samples. Waiting for the first frame of samples marks the longest wait needed for enough samples to process; after that initial buffer is received, the only half the number of samples filling a single frame need to be received before another processing call can be made. With the assistance of an incremental processing-flag and several circular buffers, this implementation was remarkably effective.

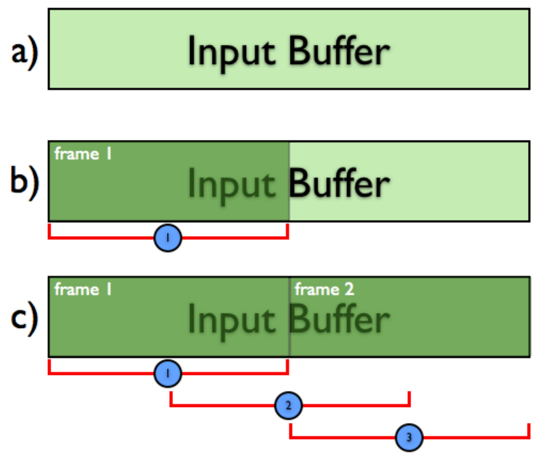

Fig. 4: Buffering scheme illustrated from the initial sequence (a), to one frame added (b), and finally a second frame added (c). Each red bar under the input buffer represents the span of a frame of audio being processed, and the order of processing is indicated by the blue labels for each red bar.

To ensure a smooth transition between frames of audio, a hanning window was used to taper each frame before being processed. After a frame was processed, the result would be added to an output buffer, which was declared similarly to the input buffer. The output buffer would serve as the source for samples to be taken from and sent out to be sounded. Taking advantage of the AUEffect process function, each iteration of the *while* loop sent incoming samples into the input buffer, and likewise, an output sample from the corresponding point in the output buffer would be added. Although initially, this would send out null values (as processing will not have occurred before enough samples had been stored to fill a frame), the initialized output buffer was zero-loaded to avoid random output. Much thanks to Stephen Molfetta for his guidance in constructing the aforementioned buffering system.

With a stable infrastructure to buffer an incoming signal and process it with low latency, the plug-in was capable of performing the algorithm in a real-time. The next issue arose in attempting to get multiple input streams into the plug-in while running. The default setup of an AUEffect handles mono input and output, and no support is available from Apple's AU documentation for any alternatives.

B.2 Multichannel Handling

The nature of a channel vocoder implies that two input streams of audio are needed to map the magnitude spectrum of a modulator signal to a carrier signal. To overcome this issue, efforts were initially spent attempting to work with the AUInstrument template; in subclassing the AUInstrumentBase, the hope was to create an instrument with the capabilities of an AUEffect plug-in; the instrument's output would serve as the carrier, and a the modulator would utilize the traditional input stream into the plug-in. After hours of researching Audio-programming blogs and developer websites, two realizations were made: 1) no suitable documentation was available to explain the inner-workings of an AUInstrument, and 2) using an AUInstrument would not solve the problem at hand regardless. Only handling a mono channel configuration, an AUEffect would only ever support a single input and output channel; the fact that the plug-in was also an instrument would not add the ability to process more than one channel at a time. This made the objective clear: support for a multi-channel configuration would be needed to move forward in the implementation. Thanks to the assistance from Brian Gerstle (former MUE - 2009), several example projects were made available to inspect which had already broken ground on this very issue.

In order to configure multi-channel support for an AUEffect, several methods needed to be overwritten. Measures needed to be taken to inform the host application of the channel layout as well, taking the form of an extended set of initialization calls. With multi-channel support, more than one channel could be read into the plug-in's architecture, and would therefore provide the needed access to read two steady input streams simultaneously. To avoid unusual channel mapping requirements or conversions, a stereo input and output configuration was decided upon. Passing in the modulator on the left channel and the carrier on the right, each channel was then accessible independently through the overwritten process function that was implemented. Post-processing, the single output channel was duplicated and sent out both channels. Keeping a stereo configuration for both input and output streams made the plug-in usable on any stereo channel, as opposed to a stereo-in mono-out plug-in, which would require a specialized channel layout in any host application to properly function. Figure 5 outlines the algorithm implemented by the channel vocoder plug-in.

IV. Evaluation

After completing the implementation of the channel vocoder as an AUEffect, the results were initially disappointing. A factor of the channel vocoding process is sig-

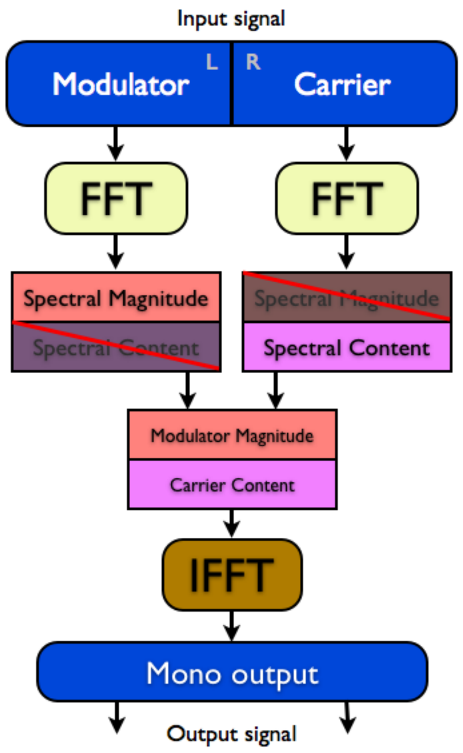

Fig. 5: Flow diagram of the channel vocoder algorithm. The red lines through the rectangular fields in the frequency domain signify the discarding of the respective information of each signal.

nificant attenuation of both signals coming out of the frequency domain. As the magnitude information for the carrier signal is effectively overwritten, the output is a signal is driven with the frequency content of the carrier and the spectral amplitude of the modulator. This process would consistently reduce the level of the output post-processing. After testing the output of the plug-in at several stages of processing, I recognized that the signal was not being normalized as it had been in the MATLAB implementation, which presented a problem to the design. Sethares' script in MATLAB read through two files offline, performed processing on the clips in entirety, and output a single chunk of processed information. Having seen the entire output file, normalization would not have any impacts on the perceived signal. In a real-time implementation, however, such normalization cannot occur. If each buffer, after being processed, were to be normalize, there could be no guarantee that a suitable maximum sample value for the given buffer would be present. Even if there was a suitable maximum sample value in the current buffer, chances would be that the next buffer would lack such a broad range of sample values. Normalization with access to such a small number of samples at a time ended up adding significant noise to the signal, and the results produced were unnatural and undesired. The gain-drop from the channel vocoding procedure was still an issue, and measures needed to be taken to condition the signal to an audible level. Experimenting with various constants, a fixed *normalization gain* was applied to every output sample (experiments ranged in value from 35-45). This seemed to boost the signal (although arbitrarily) substantially to a level where the output from the vocoder was intelligible.

A. Harmonizer Emulation

Reflecting on the initial goal, the channel vocoder plug-in does emulate the nature of a vocal harmonizer. With a vocal stimulus, the plug-in will synthesized a signal which is sounded through the spectral content of the carrier (notes or chords sounded by a synthesizer) with the spectral amplitude of the modulator (voice). The plug-in does not take into account any pitch information from the vocal modulator being received. Behaviors such as this would make the effect true to the ideals of a vocal harmonizer, but the given time constraints did not permit such thorough development.

V. Future Work

The primary objective of future work would be to condition the output of the channel vocoder to a more audible and controllable level. Overcoming the normalization issue is a critical step in this process. Perhaps instead of adding an arbitrary gain to the output signal from the frequency domain, a pseudo-normalization procedure can occur within each buffer, normalizing the buffer to a fixed *estimate* of what the maximum sample value should be.

The eventual implementation of a pitch-shifting feature would be a powerful addition to the plug-in. With a robust implementation of the phase vocoder running in real time, synthesized copies of the modulator signal could be pitch-shifted to match the pitches specified by the keys played on a synthesizer producing the carrier signal. The added benefit of incorporating a phase vocoder instead of a channel vocoder would be maintaining the original formants of the modulator signal, which would in turn result in an output signal sounding much more human and less robotic or synthesized.