# Phase Vocoder Implementation of a Harmonizer

Elec 484: Audio Signal Processing

August 11 2016

By Maitreya Panse and Trevor Lee

# Table of Contents

# Abstract

This report discusses an implementation of a harmonizer using a phase vocoder. A phase vocoder is used to describe a group of sound analysis synthesis techniques where the processing of the signal is performed in the frequency domain. This report outlines the process and challenges of developing a phase vocoder plug-in with the JUCE framework. The designers of this plug-in implemented real time pitch shifting to an audio input. To make the effect more interesting, they decided to mix the pitch shifted effect with the original audio to create a harmonizer. The designers believe their harmonizer is an important effect that can be utilized by a wide audience of music producers. This report contains brief code snippets of the phase vocoder plug-in to help explain design choices and implementation. The entire project can be downloaded from the GitHub account https://github.com/mpanse/Pitch-Shift-Phase-Vocoder-. Additionally, a video demo of the harmonizer can be found at https://www.youtube.com/watch?v=b54_V8P_4ng.


# 1.0 Introduction

The phase vocoder is used to describe a group of sound analysis synthesis techniques where the processing of the signal is performed in the frequency domain. Common effects that are implemented with a phase vocoder are pitch shifting, time stretching, robotization, whisperization, denoising, and mutation between two sounds.

This report discusses our implementation of a phase vocoder with a harmonizing effect. The harmonizing effect was created by performing pitch shifting to an audio input and mixing the result with its original input. A plug-in for Digital Audio Workstations (DAWs) such as Audacity, Reaper, and Ableton was created to prove that a pitch shifting effect can be done based on phase vocoder theory presented in lectures and textbooks. The phase vocoder effects plug-in was created using the JUCE framework. This framework included a user friendly Graphical User Interface (GUI) library that was accessible through JUCE's project management and code editing tool, known as Projucer. With Projucer, control dials were added to our GUI to vary the sound "mix" and "pitch" parameters. This resulted in real time changes to the output audio signal.

# 2.0 Phase Vocoder Basics

## 2.1 The Phase Vocoder

At a high level, a phase vocoder is created by following the steps below.

1) Given an input signal of arbitrary length, choose a frame of N consecutive samples. The value N is known as the frame size.

2) Multiply the signal by a window function of length N. The window function is defined to have nonzero value for N consecutive samples and a value of zero everywhere else. By multiplying the input signal and the window function, only the N samples in the frame will remain; the rest will be set to zero.

3) After the signal has been windowed, apply a fast Fourier transform (FFT). The combination of window function and FFT constitutes the short-time Fourier transform.

4) The output of the FFT will be a collection of M frequency domain bins containing magnitude and phase information for each frequency. Each phase vocoder effect will apply a different type of processing in this step (for our phase vocoder, we applied a pitch shifting effect).

5) Apply inverse fast Fourier transform (IFFT) to the output of step 4.

6) Add the samples from step 5 to an output buffer, which holds the output signal effect.

7) Move on to the next frame and return to step 1.

The following figures overview the steps of designing a phase vocoder. Figure 1 and Figure 2 show the same idea but in different graphical ways.

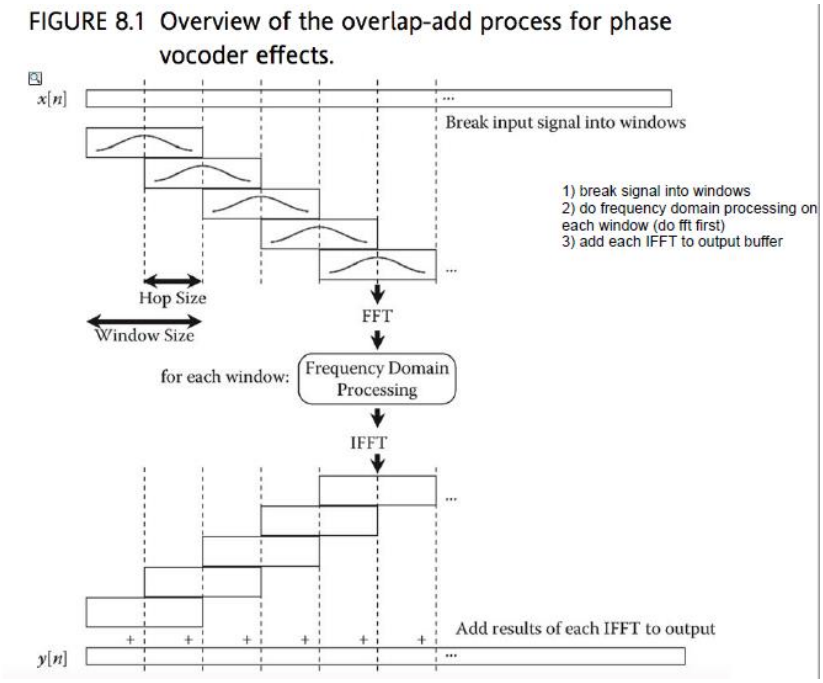


*Figure 1: Overview of Overlap and Add Process*

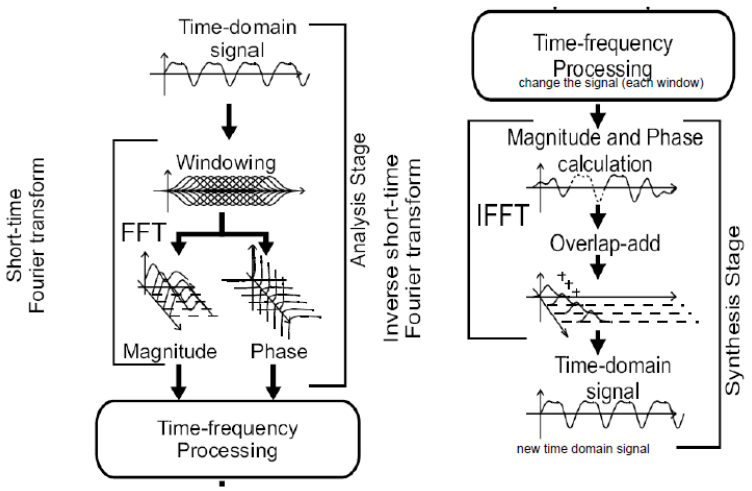


*Figure 2: Overview of Phase Vocoder Implementation*

## 2.2 Pitch Shifting

To create our harmonizing effect, we needed our phase vocoder to perform pitch shifting. The pitch shifting effect was programmed to perform in real time on an audio track. There are several ways to shift the pitch of a signal without changing its speed using a phase vocoder. The approach we used was a filter bank approach. The basic idea of pitch shifting using a filter bank approach is described below.

1) Calculate the phase increment per sample by $d\psi(k) = \Delta\varphi(k)/R_a$.

2) Multiply the phase increment by the transposition factor and integrate the modified phase increment according to $\psi(n+1,k) = \psi(n,k) + transpositionfactor * \Delta\varphi(k)/ R_a$.

3) Calculate the sum of the sinusoids.

In the steps above, $R_a$ is the hop size and the phase difference is calculated according to $d\psi(k) = transpositionfactor* \Delta\varphi(k)$. Figure 3 below shows pitch shifting with the filter bank approach. The analysis gives the time-frequency grid with analysis hop size $R_a$.

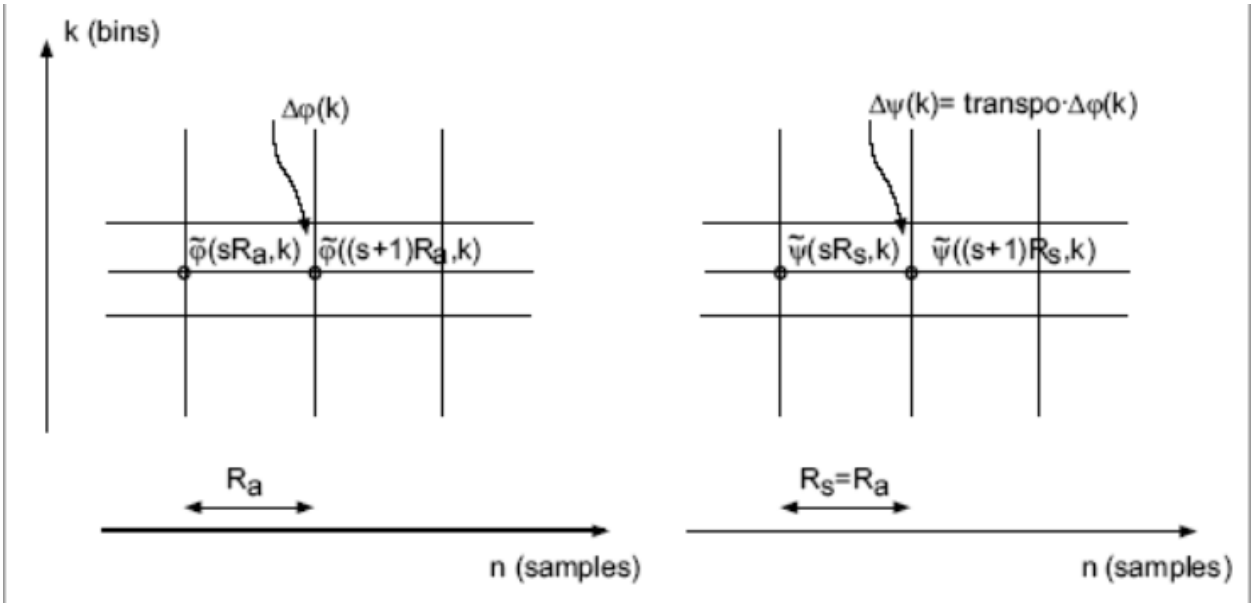


*Figure 3: Time-frequency grid with analysis hop size Ra.*

## 3.0 Detailed Description of our Phase Vocoder

The phase vocoder harmonizer that we created was developed for the audio production application called Reaper. Real time pitch shifting of audio input was implemented with our phase vocoder. To make the effect more interesting, we decided to mix the pitch shifted effect with the original audio to create a harmonizer. Figure 4 shows our application running in parallel with Reaper. Figure 5 shows a close up of our application's interface.

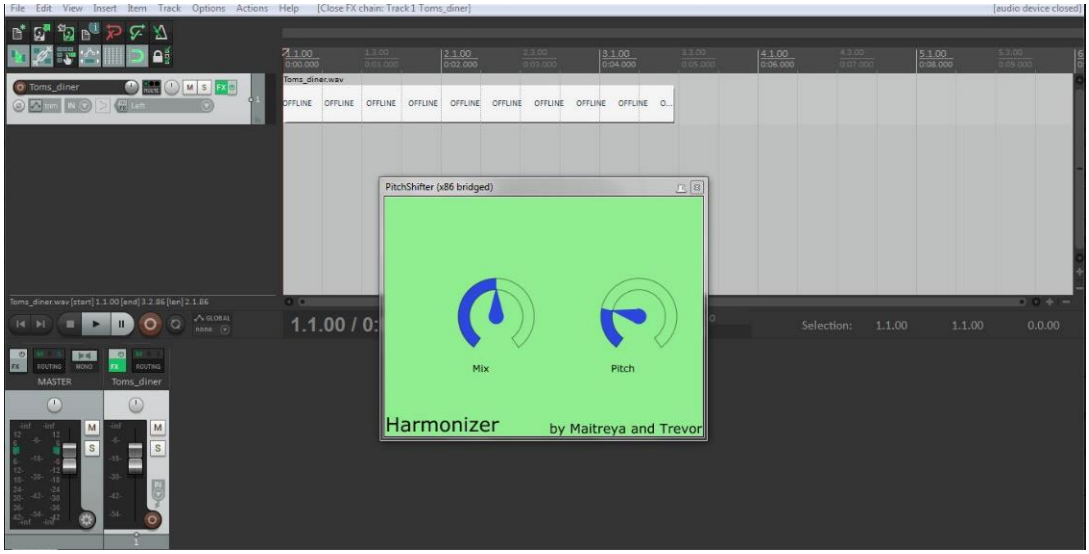


*Figure 4: Phase vocoder plugin running with Reaper*

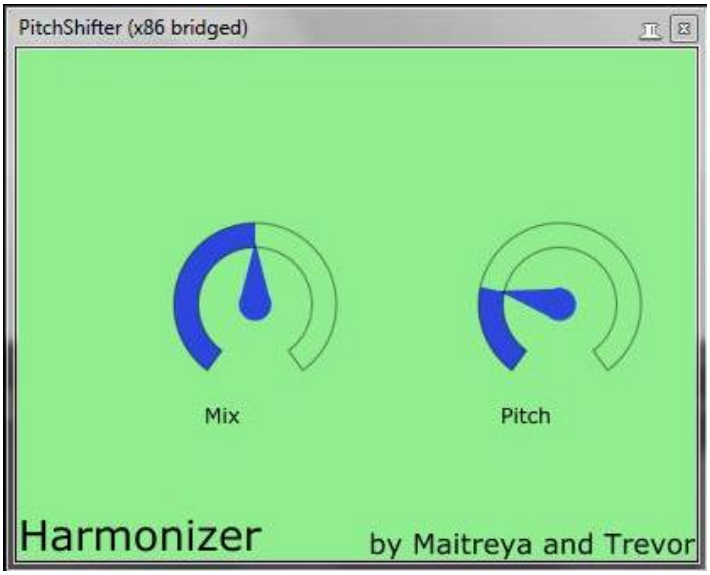


*Figure 5: Phase vocoder plug-in interface*

As shown in Figure 5, our application contains two knobs called Mix and Pitch. The Pitch knob changes the pitch of an audio signal depending on the knob placement. The further right the knob is positioned, the higher the pitch; the further left, the lower the pitch. The Mix knob is used to control the amount of pitch shifted output that is mixed with the original audio input. The further right the Mix knob is positioned, the more output audio is mixed with the input audio; the further left the knob, the less output audio is mixed with the input. If the Mix knob is positioned as far left as it can go, the output will be identical to the original input as this means zero pitch shifted output is being combined with the input.

To build our phase vocoder, we implemented the steps described in section 2.0 Phase Vocoder Basics, using C++ code. At a high level, we implemented pitch shifting and combined it with the original input. The following steps outline the basics of our harmonizer.

1) Capture the input signal (time domain)
2) Window the data
3) Perform FFT on windowed data
4) Convert the data to magnitude and phase form
5) Multiply the phase and frequencies with pitch changing parameters
6) Perform IFFT on the data
7) Overlap and add signals

## 3.1 Windowing

The first step in building a phase vocoder is to split the input audio into frames. The frame size is a predetermined number of samples known as the window size. Each frame is multiplied by a window function. By multiplying the frame by the window function, only the samples in the frame will remain, and the rest will be set to zero.

For our phase vocoder, after conducting extensive research and testing in MATLAB we decided to use a frame size of 2048 samples and a hop size of 512 samples (frame size/4). The window function that we used was a Hanning window. Figure 6 shows the definition of our frame size and hop size. A library that we used for our project contained a function called “hanning()” that allowed us to apply a Hanning window to our frame size. Figure 7 shows how we used the Hanning window function.

```
#define WINDOW_SIZE             1024 //Was 4096
#define HOP_SIZE                (WINDOW_SIZE/2) //Was 1/4Window Size
```

*Figure 6: Window size and hop size defined*

```
// Apply hanning window to our main window (less overlap)
hanning(dat->win, WINDOW_SIZE);
```

*Figure 7: Hanning function shown*

## 3.2 Cyclic Shift

Cyclic shifts ensure a centered impulse is zero phase, resulting in a centered Fast Fourier Transform (FFT). Without cyclic shifts, there will be oscillating phase values between consecutive frequency bins. Phase unwrapping will occur in opposite directions for even and odd FFT bins.

The library that we used for our project contained a function called “fftshift()” that easily performed cyclic shifts. Figure 8 shows how we called this function to perform the cyclic shift.

```
// Obtain minimum phase by shifting time domain data before taking FFT
fftshift(myData->cur_win, WINDOW_SIZE);
```

*Figure 8: The function that performs a cyclic shift and FFT*

## 3.3 Fast Fourier Transform

The Fast Fourier Transform is performed on each windowed and shifted segment. The FFT is a computationally inexpensive algorithm for finding the discrete Fourier transform of each segment. The results of the FFT gives a collection of frequency domain bins containing magnitude and phase information for each frequency. With this output, we applied pitch shifting processing. In our project, we used a function called “rfft()” that performed the real FFT on the selected segment. Figure 8 above

shows how we called this function.  This function had a "forward" and "backward" FFT argument which upon specification performed either the Fourier transform or the Inverse Fourier transform.

## 3.4 Pitch Shifting

After obtaining the magnitude and phase information from the FFT for each bin in the frame, time frequency processing is done to capture the instantaneous frequencies that were not captured due to the frequency resolution of the FFT. This method is done by taking the phase difference between adjacent bins and wrapping them to the –pi to pi interval and then computing the true instantaneous frequency. This computation loop is shown below in Figure 9.  Note that only half the size of the FFT was used due to the symmetry property of the FFT for real valued signals.

```
# pragma mark - Time-Frequency Processing
        // Get frequencies of FFT'd signal (analysis stage)
        for (j = 0; j < WINDOW_SIZE/2; j++)
        {
            // Get phase shift (true frequency w/ unwrapped phase shift)
            tmp = myData->cur_phs[j] - myData->prev_phs[j];
            // Set prev_phase to cur_phase
            myData->prev_phs[j] = myData->cur_phs[j];

            // Get Frequency Deviation (convert to radians)
            tmp -= j*omega;

            // Wrap Frequency Deviation to +/- Pi interval
            tmp = fmod(tmp + M_PI, -2 * M_PI) + M_PI;

            // Get deviation from bin freq from the +/- pi interval (convert from radians)
            tmp = osamp * tmp / (2. * M_PI);

            // Compute true frequency (new phase of freq bin j) by adding phase shift
            tmp = (long)j * freqPerBin + tmp * freqPerBin;

            // Store true frequency
            myData->anaFreq[j] = tmp;
        }
```

*Figure 9: Computation loop*

After obtaining the accurate frequencies and their corresponding magnitude and phase values, the pitch-shifting is done. The pitch shifting algorithm used in this case is the sum of sinusoids filter bank approach where the frequencies are represented as sum of sinusoids and then are modulated in frequency and magnitude. Figure 10 below shows the high level block diagram of the filter bank. Each Analysis branch below represents processing of a single frame.

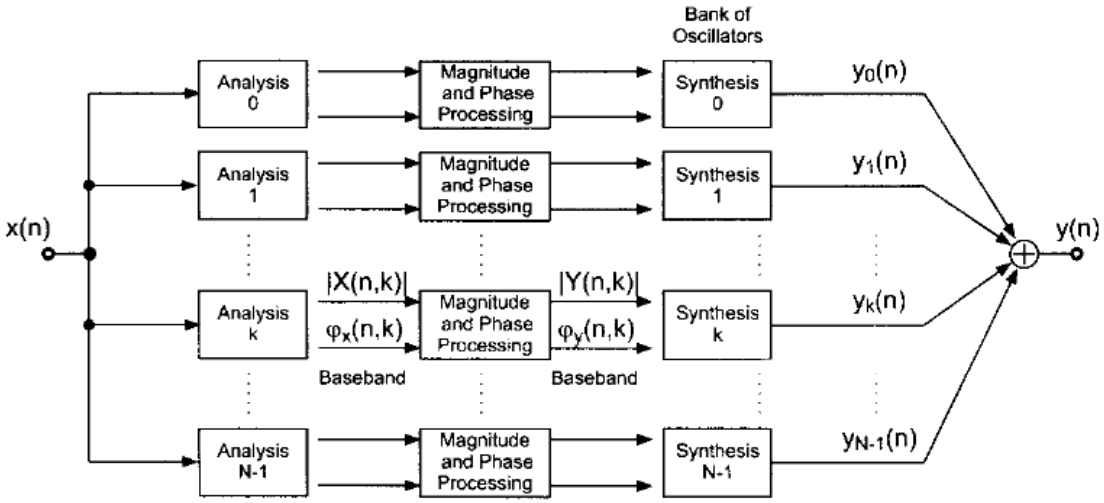


*Figure 10: High level block diagram of the filter bank*

In the magnitude and phase processing block, the bin index of the phase to be adjusted is determined by the user controlled pitch factor. If this index is less than half the window size then the current magnitudes and frequencies are adjusted accordingly. In the synthesis block the pitch shifting is done by first obtaining the adjusted true magnitude and frequencies. Then, the bandwidth of each frequency bin is subtracted from the new frequencies to obtain the actual pitch shifted frequency. From these frequencies the overlapped phase is adjusted and the phase difference between the bins is added to obtain the new pitch shifted phase. Lastly, the complex magnitudes are converted back to Cartesian co-ordinates for going back to the time domain. Figure 11 below shows the Synthesis block implementation of the pitch-shifter.

```
pragma mark - Synthesis
    // Write our new magnitudes and phases
    for (j = 0; j < WINDOW_SIZE/2; j++) {
        // get magnitude and true frequency from synthesis arrays
        myData->cur_mag[j] = myData->synMagn[j];

        // Subtract mid frequency bin (get actual pitch-shifted frequency from position j bin)
        tmp = myData->synFreq[j] - (float)j * freqPerBin;

        // Get bin deviation from frequency deviation (get actual pitch shifted frequency from frequency bandwidth)
        tmp /= freqPerBin;

        // Factor in overlap factor
        tmp = 2. * M_PI * tmp / (float)osamp;

        // Add the overlap phase back in (convert to radians)
        tmp += j*omega;

        // Accumulate delta phase to get bin phase
        myData->sumPhase[j] += tmp;
        myData->cur_phs[j] = myData->sumPhase[j];
    }

    // Back to Cartesian coordinates
    for (j = 0; j < WINDOW_SIZE/2; j++) {
        cbuf[j].re = myData->cur_mag[j] * cosf(myData->cur_phs[j]);
        cbuf[j].im = myData->cur_mag[j] * sinf(myData->cur_phs[j]);
    }
```

*Figure 11: Synthesis block implementation*

Lastly the inverse FFT of the Cartesian co-ordinates is computed to go back into the time domain. This process is done frame by frame and the data is added to an intermediate output buffer which is added to the output buffer. The next frame starts at the nth input sample determined by the hop size, then this process is repeated.

## 3.5 Inverse Fast Fourier Transform

The Inverse Fast Transform (IFFT) is a fast algorithm to compute the inverse Discrete Fourier Transform This converts the frequency domain data which had the phase and magnitude information corresponding to the frequencies to the time domain. The "rfft()" function defined in the fft file with an argument of "backward" is used to convert the frequency domain data back to the time domain. Note that the time domain data is required to be aligned with appropriate time.

## 3.6 Overlap and Add

The final step in building our phase vocoder was the overlap and add stage. Overlap and add takes the newly processed time domain frame and moves it to an output buffer. The frame is placed in the output buffer at its proper time. Since we have implemented pitch shifting in a way that does not require time stretching, we know that the resynthesis hop size will always be equal to the analysis hop size. The analysis and resynthesis hop sizes are both defined to be quarter of the window size so there is some overlap between consecutive segments.

## 3.7 Harmonization

To produce the harmonizer, the phase vocoder pitch shifted output is added to the input signal to produce a harmony. The amount of the output signal mixed in with input signal is controlled by the mix factor which is controlled by the user.  This results in real-time mixing of the input and output audio signals.

## 3.8 Design Decisions and Testing

The testing of the pitch shifting phase vocoder was initially done in MATLAB for three window sizes and two hop sizes for a total of six combinations. These window sizes were namely 1024, 2048, and 4096. And the two hop sizes were half the window size and quarter of the window size for each window. The best results for high and low pitch extremes were obtained in MATLAB for the 4096 window size and 1024 hop size. Therefore, initially these hop sizes were chosen for implementation of our phase vocoder in JUCE. However, after implementing the pitch shifting plug-in and conducting testing in two DAWs namely Reaper and Audacity, changing the pitch parameter caused the software to crash. Hence, the next best window and hop size were chosen, 2048 and 512 respectively.

One of the main factors observed in testing the plug-in was that even though the pitch shifting algorithm used was the same as the MATLAB implementation, the sound quality had noticeably degraded especially when operating the plug in Audacity. At this point the root of this problem was attributed to audacity and hence further testing was done in Reaper. Additional problems that caused the poor sound

quality were due to inefficient programming practices and array sizing errors which caused mismatches in the output especially at the hop. These errors were corrected immediately upon discovery, however the audio quality was still not as clean as it was in MATLAB. In addition, it was also observed the audio quality varied noticeably when running the plug-in on our different machines. The reason for this can be attributed to the difference in the available computing power.

Another interesting thing that was observed was the difference in audio quality when different version JUCE modules were used. The 4.2.2 JUCE version resulted in higher distortion than the latest 4.2.3 version hence the plug-in was built using the latest modules.

# 4.0 Video Demo

A video demo of our phase vocoder harmonizer can be seen at https://www.youtube.com/watch?v=b54_V8P_4ng. In this video, we explain the basic parts of our plug-in and show its functions.

# 5.0 Problems Encountered

## 5.1 JUCE Framework

The most challenging part of this project was working with the JUCE Framework. Due to a lack of documentation, problems that were encountered needed to be solved with brute force trial and error. The majority of the problems were unrelated to the implementation of our phase vocoder, but had to do with manipulating the JUCE environment. Major problem we had with JUCE were the initial setup and incorporating their libraries. All tutorials on the JUCE website and other websites assumed the initial setup was already completed and jumped directly into coding elements. In the appendix of this report, we explain how we set up the environment and incorporated their libraries.

## 5.2 Audacity

Initially the target DAW for our plug-in was Audacity, however after creating the plug-in and beginning testing the plug-in outright failed to work in Audacity on one of our computers. But, it managed to work on another computer with high amounts of distortion. Initially we were confident that the source of our problem was due to bugs in our phase vocoder code which we were able to identify and fix but the distortion in Audacity was persistent. Therefore, we tested our plug-in in Reaper and Ableton and discovered that the sound quality was considerably better. After identifying this problem with Audacity we switched over to testing the plug-in using Reaper because we were now able to isolate our testing to our phase vocoder.

## 5.3 Distortions Heard with Real Time Pitch Shifting

An interesting problem that appeared when we created our harmonizer was the noticeable distortion in the output. The distortion was even present when we did not mix the pitch shifted output with the

original input.  The distortion became worse when we did mix the pitch shifted output with the input. To solve this problem, we adjusted the window size from 1024 samples to 2048 samples, and changed our hop size from half the window size to a quarter of the window size. This change reduced the amount of distortion in the output, but not all. The remaining distortions were found to come from a mismatch in placing the processed output into the output buffer. The processed output was not being placed at the hop, rather it was being placed 20 samples to the right of the hop, and hence, there was sudden changes in sample amplitudes which resulted in the audible distortion.

## 6.0 Recommendations

If the goal of a project is to create a plug-in for Reaper or Ableton, the JUCE framework can effectively be used to create one. However, if the plug-in is intended for Audacity, results may vary depending on the computer that is being used. For our project, we found that our plug-in did not work in Audacity with one of our computers, but worked with another computer (with high amounts of distortion and lag). We also recommend that anyone planning on creating a plug-in with the JUCE framework should have a moderate amount of programming experience. We recommend this because there is a lack of documentation and resources to help once a user becomes stuck. Overall, we do not believe JUCE is an effective framework to create plug-ins. We found the lack of documentation and incomplete aspects frustrating to work with.

## 7.0 Conclusion

The phase vocoder is a very versatile audio processing tool because of the number of effects it can create. Once the basic design of a phase vocoder is created, it is relatively easy to add different effects. In this project, we implemented a harmonizer plug-in by creating a phase vocoder with a pitch shifting effect. We followed general steps for creating a phase vocoder and implement it using the JUCE framework. Our complete plug-in can be accessed from the GitHub account at https://github.com/mpanse/Pitch-Shift-Phase-Vocoder-.

From building a phase vocoder, we have developed a tremendous amount of insight into digital audio effects. We now understand how powerful signal processing in the frequency domain can be, but also how tedious. Overall we have developed a great appreciation for the work that is put into creating audio processing tools.

## 8.0 References

[1]Audio Effects Theory, Implementation and Application. CRC Press, 2015, pp. 189-213.

[2]DAFX-Digital Audio Effects. Chichester,: John Wiley & Sons, Ltd, 2002, pp. 237-299.

[3]G. Wang, fft.c/h. New Jersey: Princeton Education, 2016.

[3]R. Cook, fft.c/h. New Jersey: Princeton Education, 2016.

[4]"Tutorials | JUCE", JUCE, 2016. [Online]. Available: https://www.juce.com/tutorials. [Accessed: 16- Jul- 2016].

[5] Cocell, Step By Step How To Make A VST/AU Plug-in Using JUCE. 2016. Available: https://www.youtube.com/watch?v=5CgNQk5A0nk

## Appendix: JUCE Framework Setup

The JUCE Framework set-up for creating an audio plug-in is described in this section for Visual Basic 2015. There are several different target options such as Xcode and Code Blocks that can be used for creating in audio plug-in.

**1)** Download VST 3.0 SDK which is available from the following website: http://www.steinberg.net/en/company/developers.html and place it in a folder where you can access it later.

**2)** Download JUCE from JUCE's website and place it in folder of your choice.

**3)** Go to the folder where you downloaded JUCE and run Projucer.

**4)** Click on Audio Plug-in and then give your project a name, select the folder where you want to save the project, and select the target platform(s).

**5)** Go to Tools---> Global Preferences and under VST3 SDK add in the path where you saved the VST3 SDK folder.

**6)** In the left taskbar select Modules and then in the right window pane set the path for the modules folder, this is folder should be in the same directory where JUCE was installed to.

**7)** Now click on the project name in the taskbar this brings up all the settings for the project in the right window. In this window make sure Build VST and VST3 is checked and for channel configurations specify the number of channels you have this can be done by typing in "{1,1},{2,2}" in the channel configuration line.

**8)** Now click on files in the left window, you should see four files there namely "PluginProcessor.cpp", "PluginProcessor.h", "PluginEditor.cpp", and "PluginEditor.h". Here the editor cpp file is the GUI file and the processor file is where the audio processing is done.

**9)** To make the "PluginEditor.cpp" file the GUI right click on it and select add new GUI component and select the PluginEditor.cpp. Now when you click on it you should see a nice friendly interface where you can build your GUI.

**10)** Under subcomponents class you can add dials, sliders, and set their resolution and range also you can choose the background colour.

**11)** After you have designed the dials you want and the background colour click on "save project and open in IDE"

**12)** Now you have your GUI interface set up and now all you have to do is add a listener, this is described in the online JUCE tutorial.

**13)** To program the actual effect you will using the PluginProcessor.cpp and PluginProcessor.h files so you can add any variables and functions you would want in here.

**14)** In particular, the effect processing will be done under the process block in the Pluginprocessor.cpp file.

**15)** This completes the JUCE framework set-up and you can begin programming your plug-in.

**16)** Additionally, note that when you are adding external files to the project you need to do it through Projucer and not the IDE else the changes will not be saved.

**17)** The following youtube video by Redwood Audio https://www.youtube.com/watch?v=5CgNQk5A0nk goes into a step by step implementation of a gain plug-in which is helpful in setting up the JUCE environment.