Implementing an Acoustic Filter & VST Compressor

Alexander Hamidi, Matt Skrzypczyk

UIUC Physics 406 Project, Spring 2015

Contents

1 Project 1: PVC Band-Stop Acoustic Filter

The goal of our first project was to construct and analyze the spectral characteristics of a Band-Stop acoustic filter made out of PVC. Filters of any kind take signals and selectively reduce their frequency content. In the frequency domain this corresponds to taking "bands" or continuous segments of the frequency axis and attenuating the amplitudes of the frequencies in these bands, while leaving the rest of the spectrum untouched, ideally. Filters are most commonly recognized in the context of electronic circuits and digital processing, but the same theory carries over into broader classes of electromechanical systems that exhibit selective frequency responses.

1.1 Plane Wave Propagation in Pipes/Resonators

In having some ability to predict the resonant and cutoff features of pressure wave propagation in pipes, we need to analyze the pipe in the long-wavelength limit. If all dimensions of the wave-carrying medium are small relative to the wavelengths of propagation, invoke lumped-element analysis. Since the pressure and particle velocity wave do not change much over the pipe's dimensions, impedance mismatches are localized to points or "elements", as in electrical circuits (e.g. resistors, capacitors, inductors). In essence, the following analysis is a low-frequency method.

Start with plane waves propagating along the positive and negative z axis, which is oriented along the length of the main pipe.

$$\vec{P}(z,t) = P_+ e^{(kz-\omega t)} + P_- e^{(kz+\omega t)} \quad [N/m^2] \tag{1}$$

This is the general solution to the one dimensional wave equation for acoustic pressure in a waveguide.

$$\frac{\partial^2 \vec{P}(z,t)}{\partial t^2} = c^2 \frac{\partial^2 \vec{P}(z,t)}{\partial z^2} \tag{2}$$

The parameter "c" is the speed of sound in the medium of propagation, which in our case is air at standard temperature and pressure, STP. Analysis of the appropriate boundary conditions leads this expression for the transmission coefficient of a band-stop acoustic filter, as a function of frequency.

$$T = \frac{1}{1 + \frac{c/2A}{\omega L'/S - c^2/\omega V}} \tag{3}$$

"S" refers to the area of the neck of the Helmholtz resonator. "V" is the corresponding volume, "L'" is the neck length with the end correction, and "A" is the area of the opening that leads from the main pipe to the resonator, which may be different from the neck area. The transmission coefficient is a fractional quantity that describes the percentage of energy in the original acoustic signal that is "transmitted" or passed to the output. This description conveniently characterizes the properties of a filter; the quantity is approximately 1 in the pass-band, and 0 in the stop-band. Equation 3 predicts that the transmission coefficients goes to zero at the resonant frequency of the Helmholtz resonator. While Helmholtz resonators are interesting in their own right and are deserving of much longer discussions, the important takeaway is that their resonance frequency occurs when the reactance in the neck approaches zero.

$$\omega_o = c\sqrt{\frac{A}{VL'}} \tag{4}$$

While band-stop acoustic filters are generally constructed using Helmholtz resonators characterized by a single resonant frequency, ours was constructed using an Open-Closed pipe. This pipe, instead of a resonator of arbitrary volume, was necessitated by our design goal of making the cut-off frequency continuously variable. In turn, by constructing a piston, which slides up and down the pipe, we've effectively closed off the top while leaving the bottom open and connected to the main pipe. The resonances of an Open-Closed Pipe, including the end correction, are given by the following equation.

$$F_r = \frac{c}{4(H + .61r)} \tag{5}$$

"c", again, is the speed of sound in air, and "H" represents the height of the piston above the main pipe, which determines the length of the Open-Closed Pipe. The end correction, ".61*(r = pipe radius)", is necessary to account for the observation that acoustic resonances in pipes don't occur exactly at multiples of quarter or half wavelengths. In our simple model of band-stop filter characteristics, it is these resonances that should match up with the notches in the output spectral curves. On a log scale, these dips or notches don't necessarily have to go to negative infinity, but should take the form as local minimums.

1.2 Construction, Experiment

To construct our Band-Stop Filter, we used PVC pipes and connectors of dimensions below.

Incident Pipe
$Length = 30 \pm 0.1cm$
$InnerDiameter = 2.5 \pm 0.1cm$
$OuterDiameter = 3.2 \pm 0.1cm$
Transmitted Pipe
$L = 30.9 \pm 0.1cm$
$InnerDiameter = 2.5 \pm 0.1cm$
$OuterDiameter = 3.2 \pm 0.1cm$
T Joint
$HorizontalLength = 7.9 \pm 0.1cm$
$HorizontalInnerDiameter = 3.4 \pm 0.1cm$
$HorizontalOuterDiameter = 4 \pm 0.1cm$
$VerticalLength = 5.9cm \pm 0.1cm$
$VerticalInnerDiameter = 3.4 \pm 0.1cm$
$IntrusionLength = 1.5cm \pm 0.1cm$
$ProtrusionLength = 2.4cm \pm 0.1cm$
$EffectiveHorizontalLength = 65.7 \pm 0.5cm$

Making a piston allowed us to change the height of the pipe above the main pipe. We used cork plugs by shaping them into cylinders to match the diameter of the Open-Closed pipe, and gluing felt around the edge such that the cork could slide up and down the pipe with relative ease. The protruding pipe was long enough such that the piston could be elevated to 50 cm above the the main pipe, a length that violates the long-wavelength limit. We therefore expect any sensible predictions obtained from lumped element analysis to be markedly wrong when the open-closed pipe has this length.

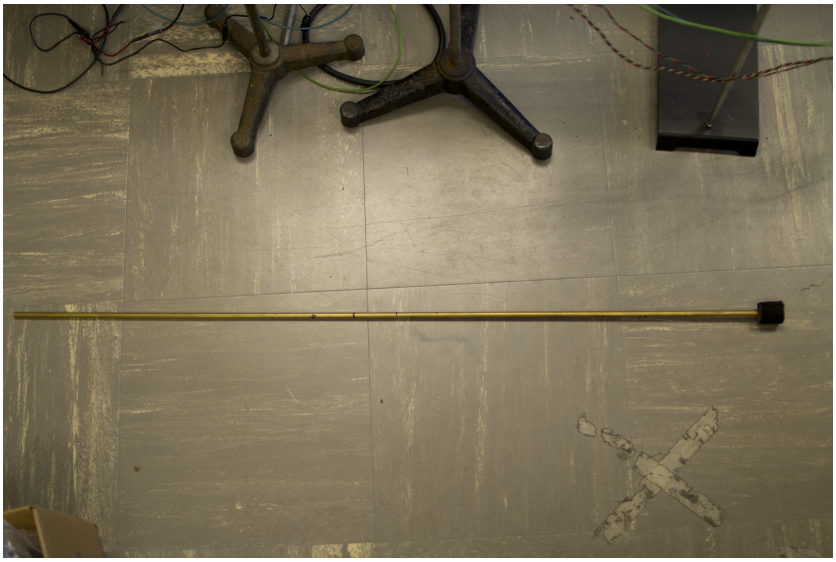

Figure 1: Piston

Figure 2 depicts what the filter looks like when prepped for measurements. The ring stands hold the main pipe in place, else the weight of the Open-Closed pipe would tip the the entire network over. The top of the Open-Closed pipe is not shown in the picture, but there is where the piston is stuck into. The pipe itself has a length of roughly 60 centimeters, whose effective length is controlled by the piston.

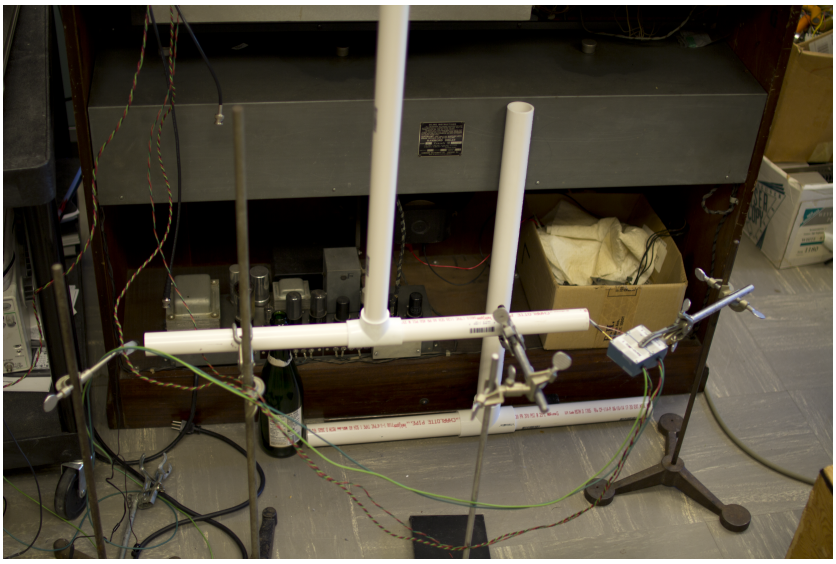

Figure 2: Filter & Stand

We used an HP 3562A Dynamic Spectrum Analyzer (Figure 3) to generate random or white noise to excite the entire audio frequency spectrum within the pipe. The signal, created by a pseudo random number generator internal to the spectrum analyzer, is fed to the one of a pair of Monster Inspiration Direct ear-buds.

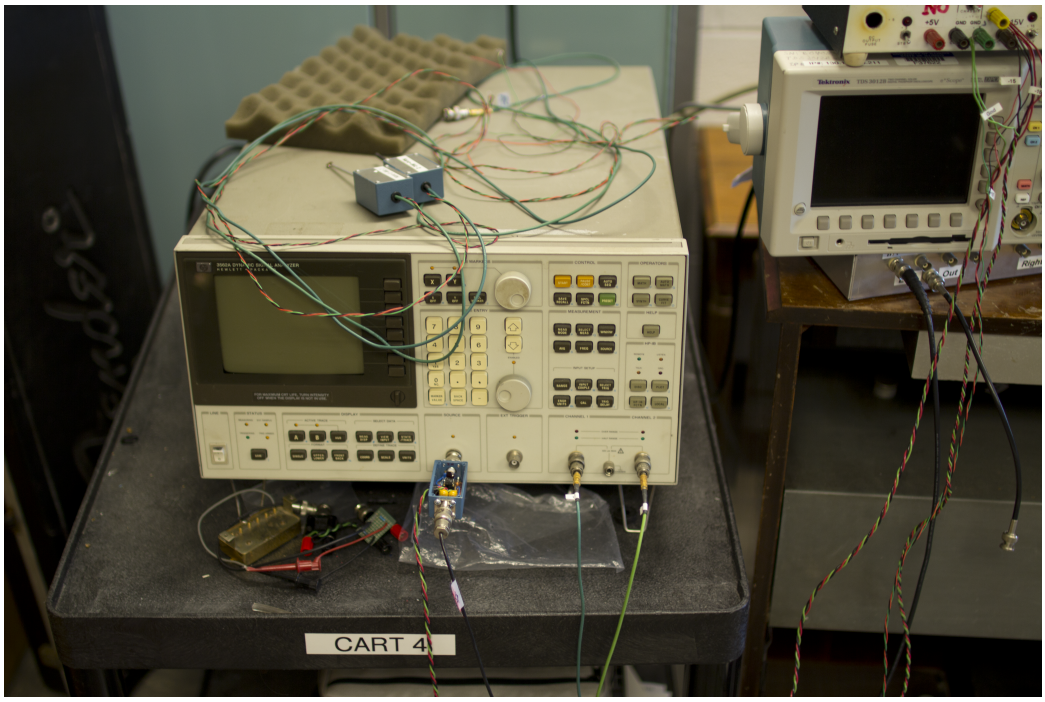

Figure 3: Spectrum Analyzer and Pressure Microphones

The spectrum analyzer is also used to read out data from a calibrated Knowles Acoustics .1" diameter FG-23329-C05 omni-directional electret condenser pressure microphones, driven by an op-amp pre-amplifier, shown in Figure 4. The pressure microphone's placements alternates between the two ends of the main pipe, either to take incident or transmission measurements. It is necessary to provide a ring-stand to hold the microphones and ear-buds, which amounts to 4 ring-stands total.

One possible source of error, not to be discussed in the following section, is how the microphone placement affects the data. Depending on how the microphone tip is oriented just inside the pipe ends, reflections occur off the ring stand and microphone itself. Certainly everything in close proximity to the microphone such as chairs, tables, and small items, also produce reflections, but the microphone itself is in much closer the source of waves. Being much closer, we expect the microphone and ring stand reflections to be much greater than other miscellaneous items in the room.

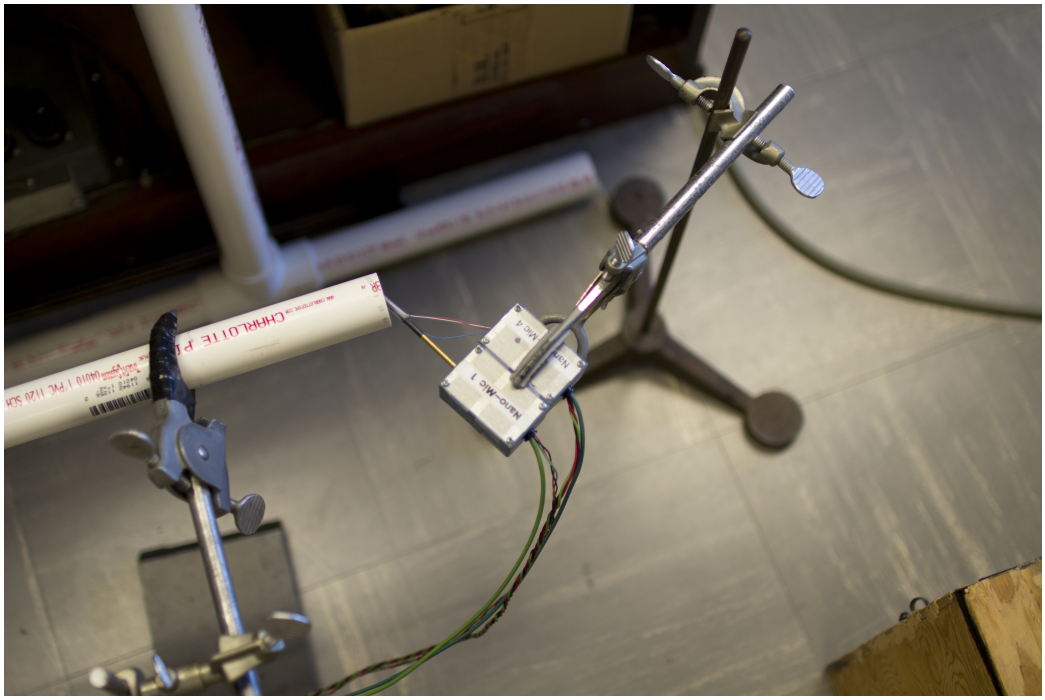

Figure 4: Microphones

The full experimental setup, excluding the computer and GPIB connections, is shown in Figure 5. The ear-bud, which sources the noise from the spectrum analyzer, feeds the main pipe's incident end (left hand side), while the pressure microphones read out power spectral density data on the transmission end. We start the procedure by taking all the data at the incident end, changing the Open-Closed pipe's height in 5 centimeter increments, then switching to the transmission end. This removes the need to move the microphone between the two ends for every measurement at a specified height, which is cumbersome. The data is read out to LAB-VIEW, a graphical programming language designed specifically to interface with and control electronic instruments like spectrum analyzers and lock-in amplifiers. Every time a measurement is taken, the spectrum analyzer is told to capture a power spectral density that is averaged over many sequential measurements. Once the analyzer is finished averaging, LAB-VIEW can be told to acquire the same data from it.

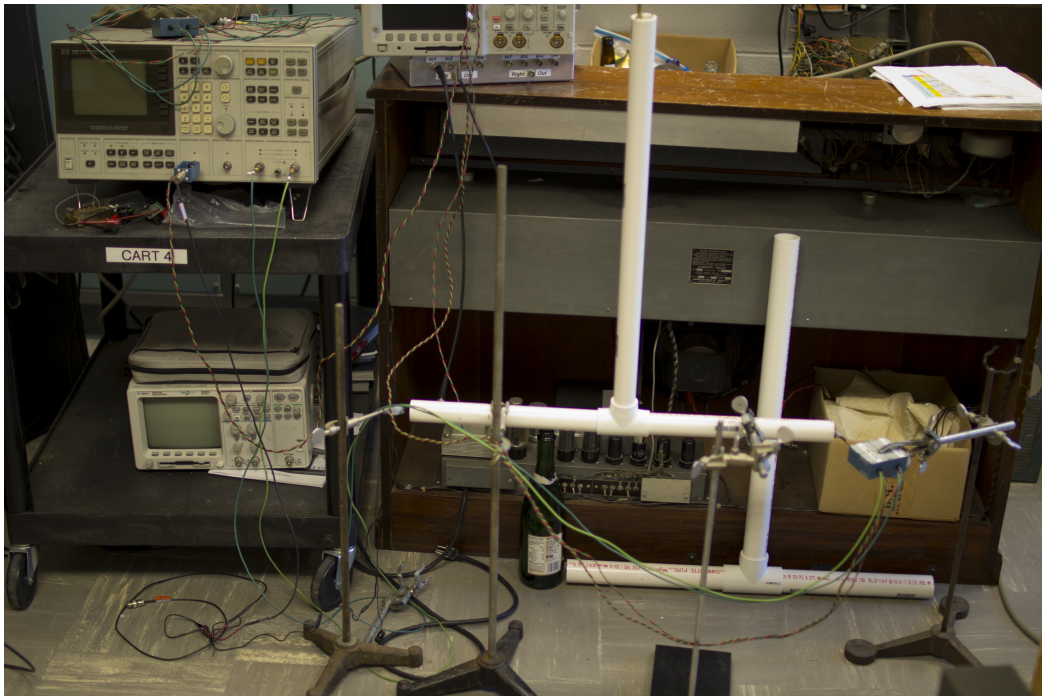

Figure 5: Entire Setup

1.3 Results, Real-World Effects, and Sources of Error

With our Power Spectral Density, or "PSD", data, we created plots of the PSD at the output divided by the PSD at the input. By taking this ratio of spectral plots, the intrinsic frequency response of the ear-buds are cancelled out of the data, since both the input and output data are taken with the ear-bud as the source of noise. These plots are not exactly transmission coefficients, but they carry the same information, telling us the relative weighting of the output energy at a particular frequency, versus the input energy. From these plots we should be able to see the band-stop filter curve, which is a "well" or notch-like shape at the resonant frequency, and flat everywhere else in the audio spectrum. Here are plots of the PSD ratio at a piston height of 10, 20, 30, 40, and 50 centimeters. A quick glance at all the graphs is that there is no readily apparent correlation between the piston height and the minimums. We quickly realized that searching for a single, dominant notch is a hopeless endeavor, since the higher resonances have non-zero overlap.

At a height of 10 centimeters, only one of the experimental minimums correlated nicely with theoretical value. We decided that if the measurement of a minimum falls within 10 hertz of the predicted resonance, then we could accept the experimental value as confirmation of the theoretical minimum. For each of the piston heights and corresponding graphs, we list the experimental and theoretical minimums whose differences are within 10 hertz. For theoretical values, the harmonic is listed in parenthesis next to the frequency.

Theoretical[2330.8 (3rd)]

Experimental[2325]

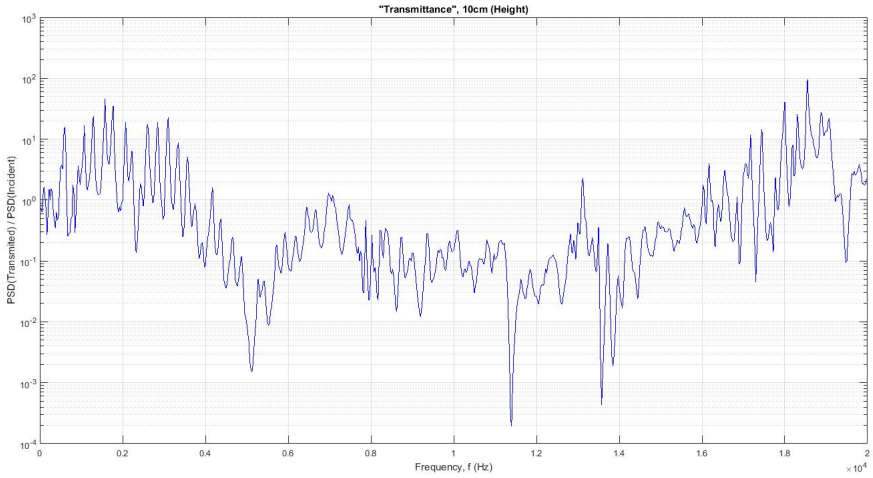

Figure 6: Transmittance, Piston Height = 10 cm

Theoretical[1222.85 (3rd) 2853.31 (7th)]

Experimental[1225 2850]

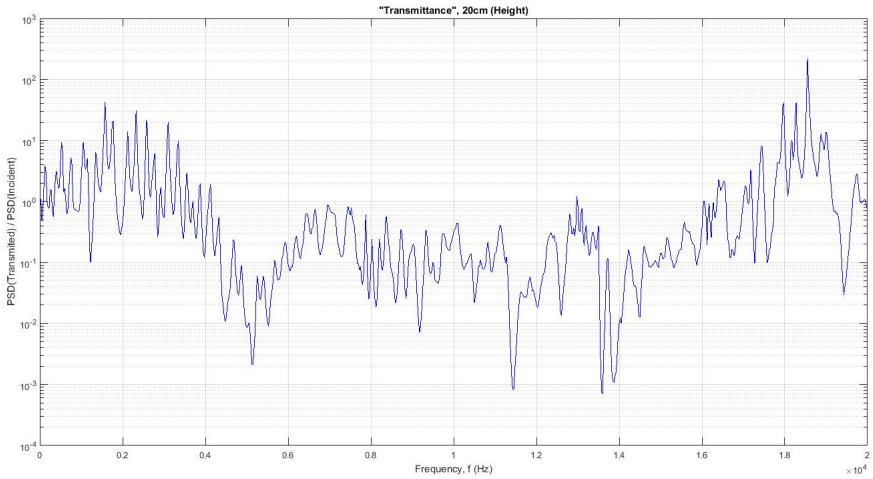

Figure 7: Transmittance, Piston Height = 20 cm

Theoretical[828 (3rd) 1381.4 (5th) 1933.9 (7th)]

Experimental[825 1375 1925]

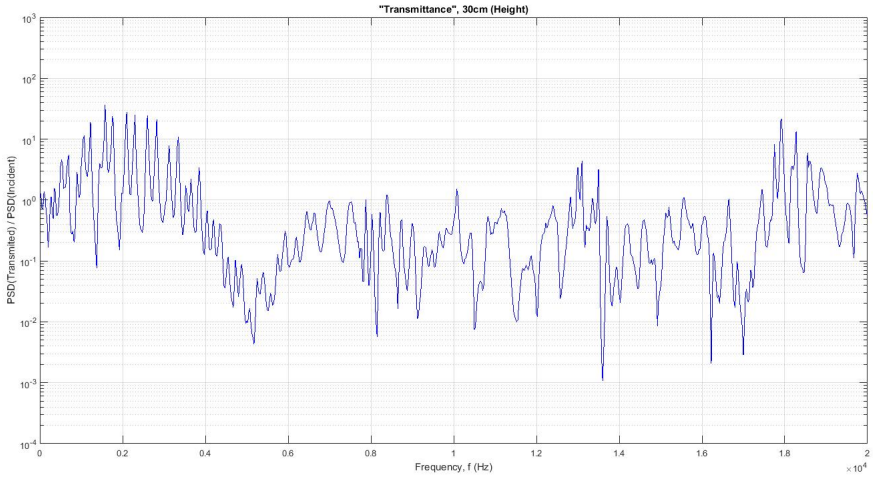

Figure 8: Transmittance, Piston Height = 30 cm

Theoretical[1880.6 (9th)]

Experimental[1875]

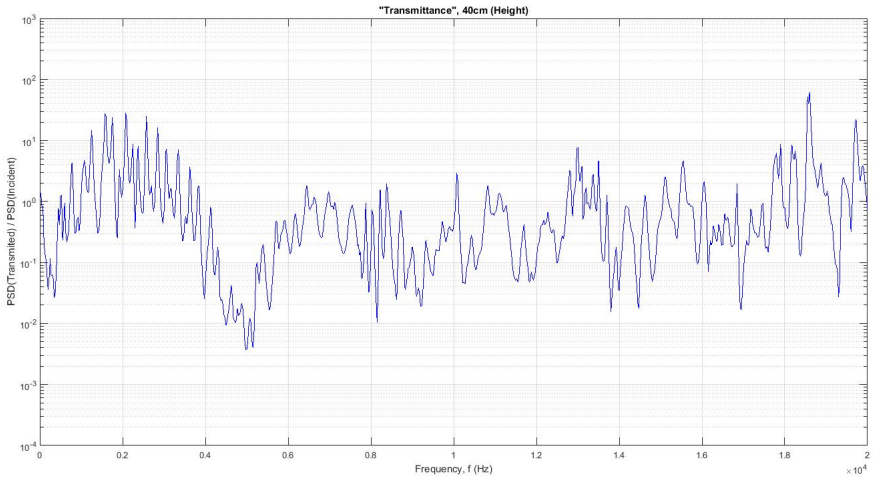

Figure 9: Transmittance, Piston Height = 40 cm

Theoretical[1848.17(11)]

Experimental[1850]

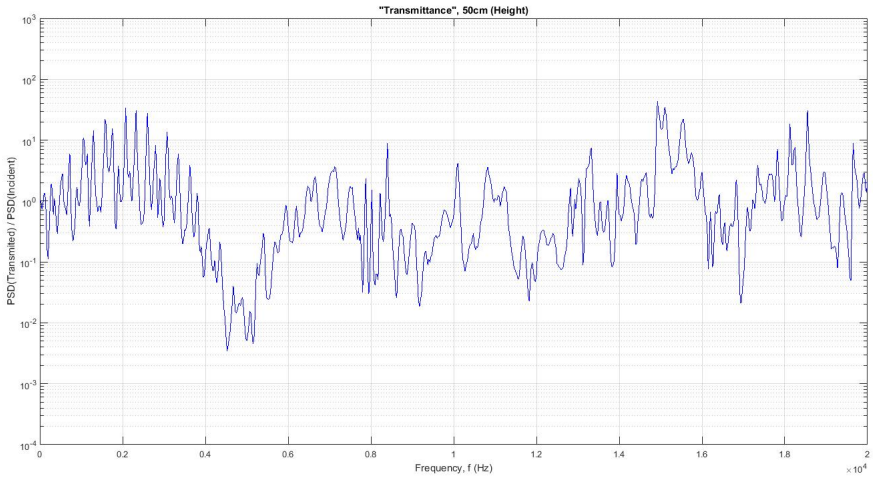

Figure 10: Transmittance, Piston Height = 50 cm

A interesting feature of all these graphs is the presence of a notch centered close to 5 kilohertz. Nothing in the theory, as far as we can tell, suggests that there be a notch frequency independent of the Open-Closed pipe's height. For some reason, it is easier to excite a resonance within the Open-Closed Pipe at this frequency.

As was described in the theory of Open-Closed pipes, these pipes have a fundamental resonance frequency and an infinite set of higher harmonics, which are progressively weaker in amplitude. Because of this complicated structure, compounded with the finite length of the pipes, we observe an uncontrolled, highly spurious interference pattern that masks the resonance frequencies as well as gives the transmittance curves a jagged shape.

As the height of the piston increases, Equation 5 predicts that the harmonics are moved to lower and lower frequencies. At at a height of 50 centimeters, the first 20 harmonics are all below 4000 hertz. When the resonances are this closely spaced, it is virtually impossible to distinguish them in a spectral curve unless the corresponding transition-band roll-off for each of those corresponding notches is very, very large. In simpler terms, these notches interfere with each other, distorting what we would recognize as individual notches.

While we may have been able to see some alignment among one or a few experimental and theoretical resonance frequencies, the degree to which those minimums were distorted, in the collected data, may have produced minimums coincidentally at the predicted values. Unfortunately, the standing wave interference behavior of several resonances is too complicated to analyze further without stronger computational methods and more controlled experimental setups.

1.4 Improvements

Constructing acoustic filters with a band-stop characteristic is significantly harder to accomplish with Open-Closed pipes that break the long-wavelength, low-frequency limit behavior of the analysis. Open-Closed pipes with pistons offer the potential to vary the cutoff frequency, but in doing so extremely complicate the filter behavior due to additional interference effects. We are certainly not saying that this particular construction will not give a working filter, but that it would require far more predictive analysis, including simulations and more controlled pipe dimensions. We suggest to any potential experimenters or designers who wish to follow up on this work, that if you're not going to precisely control the parameters and properties of the pipe and its environment, please use a set of Helmholtz resonators that you can swap out of a hole in your main pipe. This way you only have to worry finding a single resonance in your data. Otherwise, other filter characteristics might be of more interest, such as high-pass or low-pass configurations.

2 Project 2: JUCE Based VST Compressor

Our second project was to program a compressor VST and see how they process instruments of standard types. Compressors are signal processing tools widely considered central to modern audio mixing and mastering. VST or "Virtual Studio Technology" are software implementations of audio effects that are usually loaded from DAW's or "Digital Audio Workstations", which are a class of software used to record, produce, mix, and master digital audio. VST's are not usually standalone, rather they are treated as plug-ins to more general audio editing software.

2.1 Dynamic Range Compression

An audio signal, like any signal, has a time domain and frequency domain representation. A signal is characterized by a set of frequencies that make up the signal and a set of amplitudes that give relative weightings to those frequencies (Fourier's Theorem). At any given instant in the time domain, there exists a lower and upper bound on the set of amplitudes, which in audio are functions of time. Compressors take the lower and upper bounds, the difference of which is called the dynamic range, and bring them closer together, usually by decreasing the upper bound and keeping the lower bound fixed. In audio, qualitatively, this means that louder sounds sound quieter and soft sounds sound louder. Above which sound level are sounds attenuated is set by the "threshold". How large these new levels are relative to the unattenuated levels is set by the "ratio". This process, specified in the most part by the above two parameters, effectively reduces the amount of data needed to represent each sample of the audio and allows the audio to be recorded on equipment that are limited in dynamic range representation, or bits, if the recording is done digitally.

2.2 JUCE, Program Structure, and Algorithm

In order to implement our VST we used the Introjucer project manager in conjunction with the JUCE open source library and Microsoft Visual Studios 2013. The Introjucer program provided us with a base skeleton for the audio plugin and a GUI editor for the produced VST, while the JUCE C++ library provided us with object classes and functions that were designed for processing audio signals. Because Introjucer relies on an IDE to be linked to the project, most of the programming

for the audio processing was done in Visual Studios so that the source code files could be cleanly managed and built into a solution.

When a VST project is first opened in Introjucer, the user is provided with the skeleton C++ files for the audio processing component of the VST and the GUI editor component of the VST. Introjucer takes care to include comments in the code that specify where code should be placed to ensure that the solution build goes smoothly and that there are no conflicts between code altered by the IDE and Introjucer. In order to learn the basics of programming the VST we read some documentation from the JUCE project website and a tutorial on how to make a Stereo Width Controller VST from Redwood Audio ($http://www.redwoodaudio.net/Tutorials/juce\_for\_vst\_development\_\_intro.html$). This tutorial showed us the general flow of the VST design which starts with setting up the correct directory paths for the VST SDK that the Introjucer links to, as well as some information on details of the VST such as number of inputs and outputs, types of audio-plugins that should be build, etc. For our compressor we set out to design a VST which had two inputs/outputs in order to have the stereo effect.

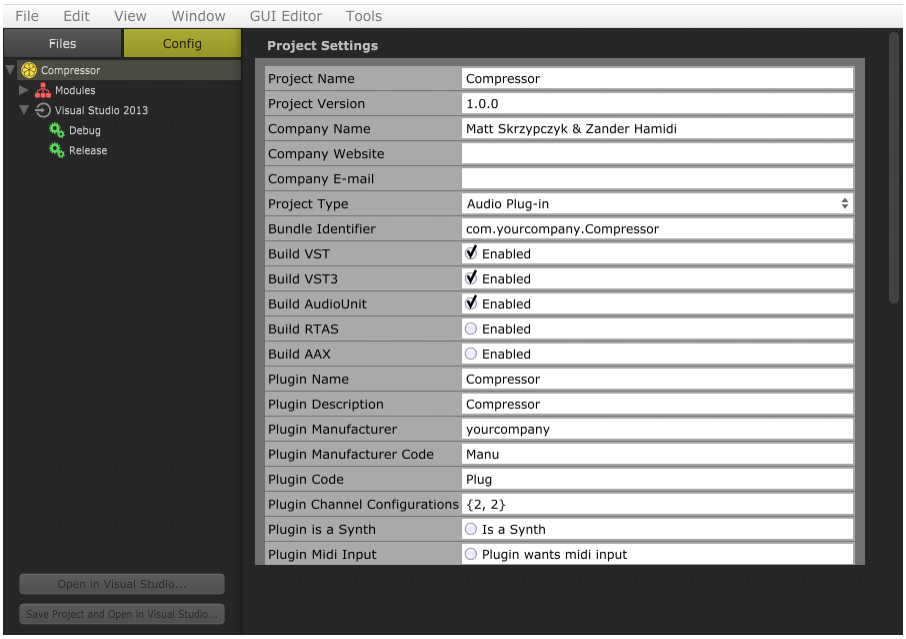

Figure 11: Project settings for the VST

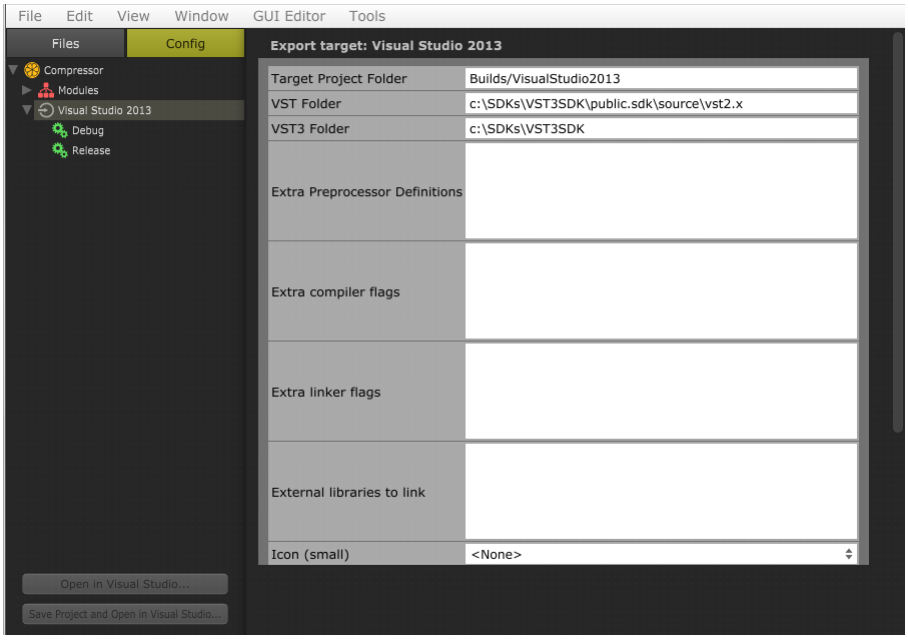

Figure 12: VST SDK Path

After modifying these fields under the main project settings we went under the yellow 'Files' tab and modified the class settings under the PluginEditor.cpp. We modified the class name to $CompressorAudioProcessorEditor$, the parent classes to $publicAudioProcessorEditor, publicTimer$ so that the $CompressorAudioProcessorEditor$ would inherit the public and private member variables from these classes. This property of inheritance is vital to object oriented programming and allows VSTs implemented in JUCE to inherit functions from the classes and libraries provided. The $AudioProcessorEditor$ class provides us with a useful function $getAudioProcessor$ which returns a pointer to the $AudioProcessor$ object (our $CompressorAudioProcessor$ in this case) so that we may access functions specific to the processor that allow us to modify the internal variables controlling the VST's operations. In this case these member variables of the $AudioProcessor$ are the ratio, threshold, attack time, and delay time for compression.

Once these main project settings have been put in place we began working on the code for the compressor in Visual Studios. The useful thing about the Introjucer program is that modifications to files are immediately updated and reflected within the IDE, so the appropriate fields are created within the code. In the $PluginEditor.h$ we modified the header files to include the $PluginProcessor.h$. We then added functions to the $UserMethods$ section to control the exchange of variable information from the VST GUI and the processor as well as a $getProcessor()$ function to return a pointer to our $CompressorAudioProcessor$ object. Under the $Consructor$ section we added the function $startTimer(200)$; from the $publicTimer$ class in order to start a timer with an interval of 200 ms in our VST. We then implemented the skeleton of $timerCallback()$ function for future modification with our VST parameters.

In order to implement the compressor we found open source code that provided an attack-release compressor object. The original posting of this code was on the Music-DSP Sound Archive, where

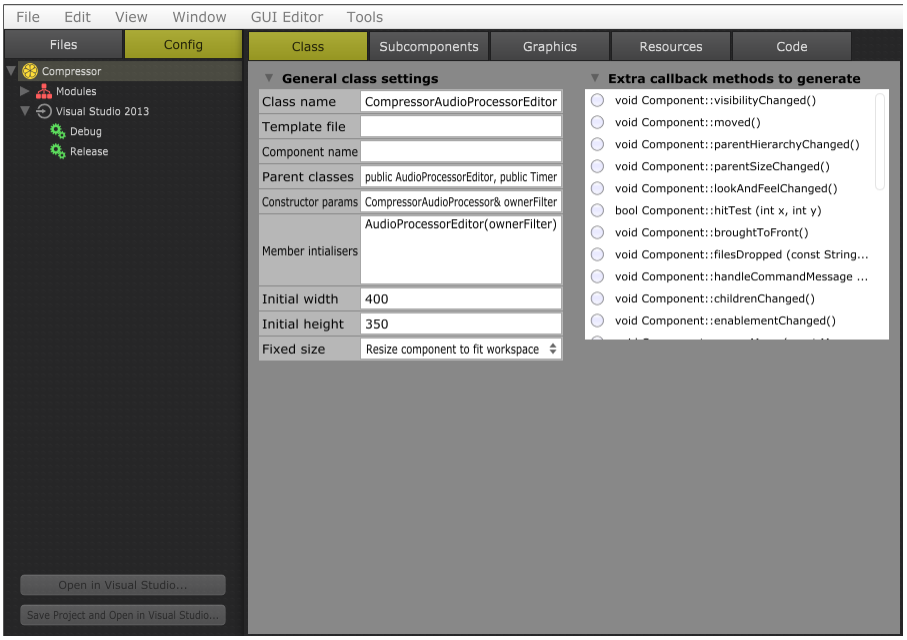

Figure 13: Class settings for the VST

we found a link to a Github repository hosting the source files. The Github link to the source code will be provided in the references section of this report. We downloaded these files and incorporated them into the project by doing the following: Right click the "Source" folder, click "Add new group", type the name, hit ¡Enter¿, then right click the new folder, select "add existing files...", and selecting the files. After saving the project the files were incorporated in the Visual Studios solution.

At this point we worked on the GUI for the VST so that we would have all the variables within the project to ensure linkage between the processing code and the VST controls. In the Introjucer program we selected the files tab and the *PluginEditor.cpp* file and were shown a small black box that represented the VST's physical layout By right-clicking in this box the program allows us to select various graphical components to add to our VST such as text boxes, parameter sliders, labels, and toggle buttons. For our VST we used the slider components to control the ratio, threshold, attack, and release times for our VST and a toggle button to control a bypass setting so that we could turn the VST off. When adding graphical components that control parameters it is important to modify the settings for the component, specifically the *membername* field. This field controls the variable name associated with the slider within the code, so it is good to have a descriptive name. For this VST we modified the member names for the ratio, threshold, attack, and release sliders to be RatioSld, ThresholdSld, AttackSld, and ReleaseSld respectively. After modifying the member names we also modified the *minimum* and *maximum* fields to control the range of these sliders as well as the interval of range. In addition, we added a Bypass button with member name BypassBtn as well as a few labels to distinguish the sliders on the VST.

Now that the graphical parameters have been set up appropriately in the Introjucer, we were able to finish the implementation in Visual Studios. We included the header file for the compressor source code in *pluginprocessor.h*, added custom methods for handling UI updates from the VST,

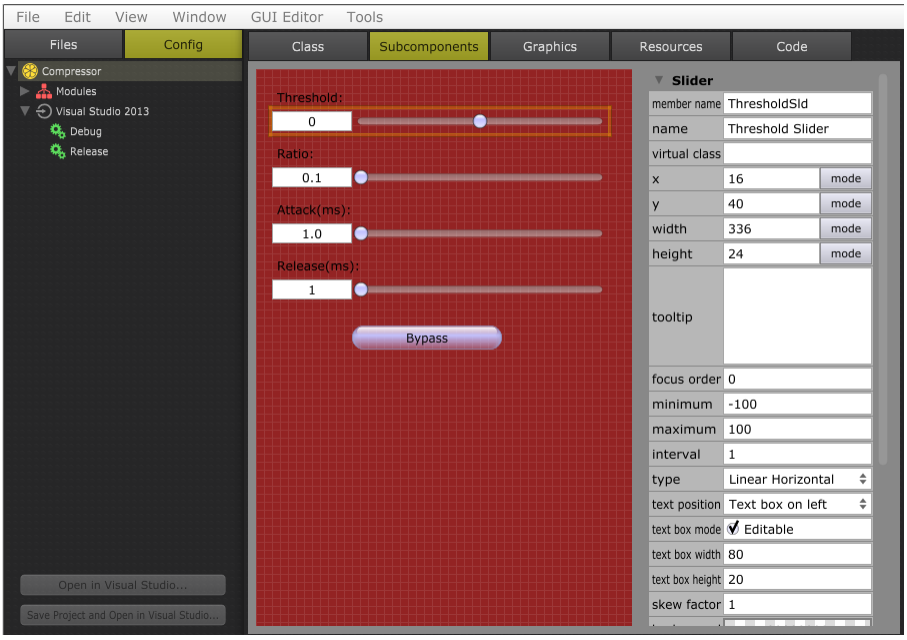

Figure 14: GUI Editor for the VST

and declared private members for the compressor object and parameters associated with it. We then added functions to initialize the *CompressorAudioProcessor* object, get-set parameters from the VST and within the object, and get parameter text from the VST. After these had been done we created the main processing component for the VST in the *processBlock* function. The following shows our processBlock code.

```
for (int i = getNumInputChannels(); i < getNumOutputChannels(); ++i)
        buffer.clear (i, 0, buffer.getNumSamples());

if (getNumInputChannels() < 2 || UserParams[MasterBypass]){}
else{
    float* leftData = buffer.getWritePointer(0);
    float* rightData = buffer.getWritePointer(1);

    for (int i = 0; i < buffer.getNumSamples(); i++){
        double curr_left = static_cast<double>(leftData[i]);
        double curr_right = static_cast<double>(rightData[i]);
        mCompressorControl.process(curr_left, curr_right);
        leftData[i] = static_cast<float>(curr_left);
        rightData[i] = static_cast<float>(curr_right);
    }
}
```

Figure 15: Process block code for compressor VST

The code starts by checking if the number of output channels is greater than the number of input channels, then clears data on the output that we don't want. Next, it checks whether the number of input channels is less than two to ensure we are not modifying data we shouldn't be. If we pass these checks, we receive pointers to the left and right signal information. The buffer object contains all the samples for the audio signal so we iterate over the number of samples in the buffer, pass these pointers by reference into our compressor's process function, and rewrite them into the leftData and rightData pointers to modify the buffer. While we were experimenting with this code we discovered we had to perform these static casts in order to operate on the data correctly, otherwise there was no modification done to the data of the sample. The following diagram shows how the compressor's process function operates.

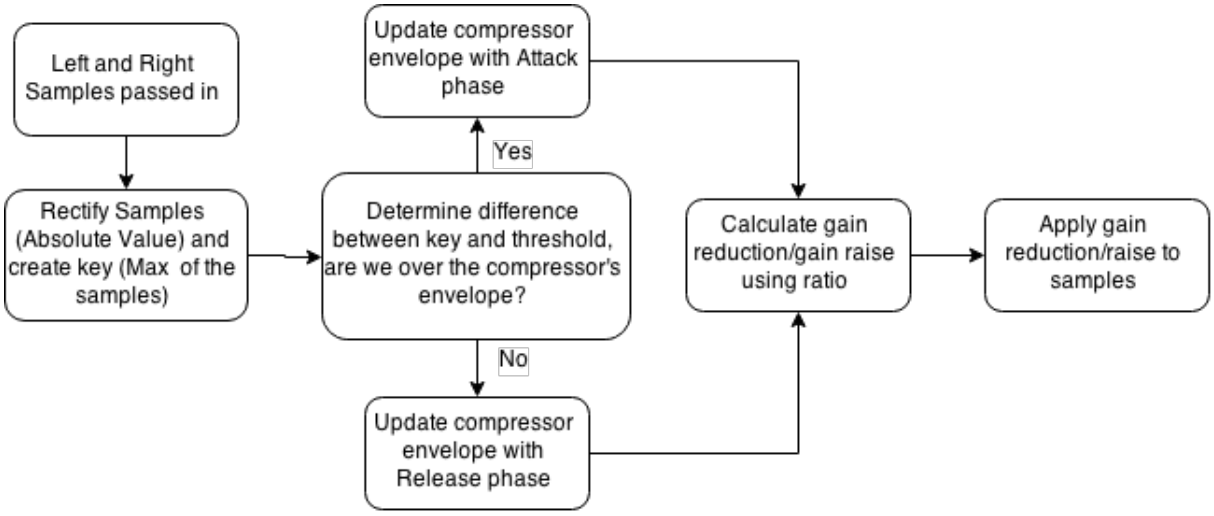

Figure 16: Flow diagram of compressor algorithm

After the processing algorithm was implemented we finished the VST by implementing functions that would retrieve values from the VST UI to update inner parameters.

2.3 Analysis of Instruments by Category

In order to analyze our compressor more, we loaded WAV files into a DAW and processed the samples with our VST. We then extracted contiguous 100,000 point samples from the uncompressed and compressed versions of the clips so that we could process and analyze the effect of our compressor on the sound files. We compressed the samples at a threshold of -80 decibels with a ratio of 0.3, attack time of 10 milliseconds and release time of 100 milliseconds for each of the uncompressed sound clips. We wanted to study the effects by category of instrument so we analyzed a cello sample, a clarinet sample, a gong sample, and a vocal sample.

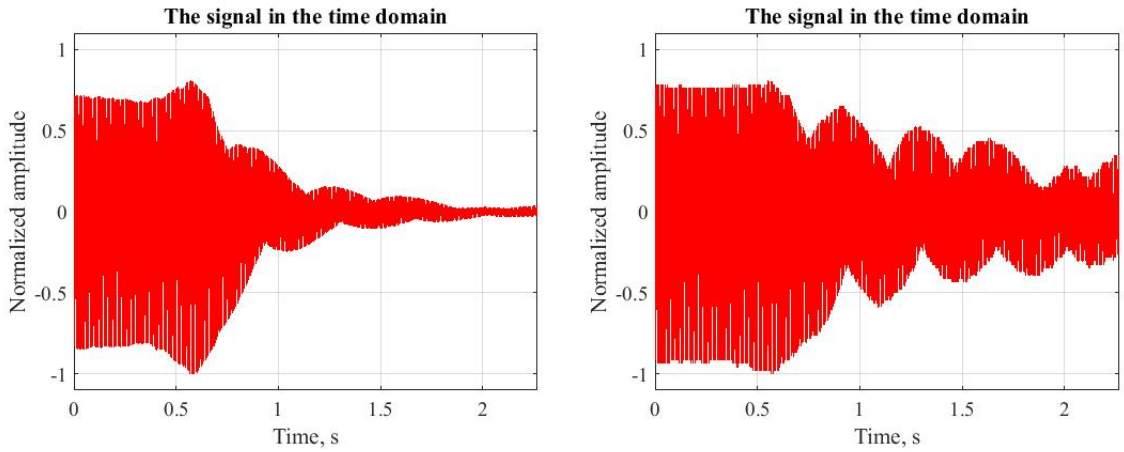

Figure 17: Cello Time Domain, Uncompressed(Left), Compressed(Right)

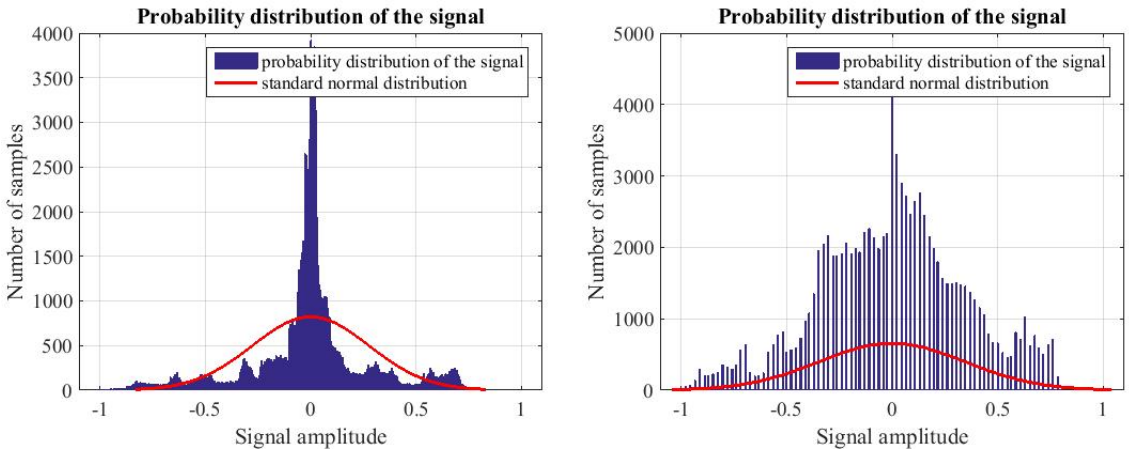

Figure 18: Cello Amplitude Distribution, Uncompressed(Left), Compressed(Right)

Above are the normalized time domain signals and the distribution of signal amplitudes for the compressed and uncompressed samples. We can see that after compression the signal amplitudes have become more broadly distributed rather than populating small amplitudes strongly. In addition, we can see that the signal amplitudes have taken on distinct values, as shown by the gaps in between the signal amplitudes observed in the compressed sample. From this we can confirm the compressors ability to reduce dynamic range. The Matlab Sound Analysis code also confirms this by telling us that the dynamic range of the uncompressed sample was reduced from 33.624 to 33.255 decibels.

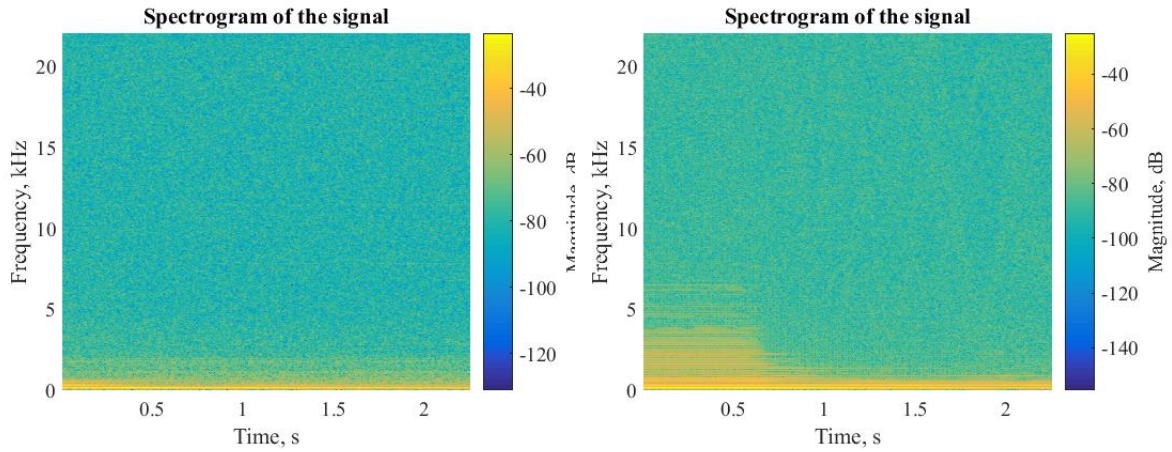

Figure 19: Cello Spectrograms, Uncompressed(Left), Compressed(Right)

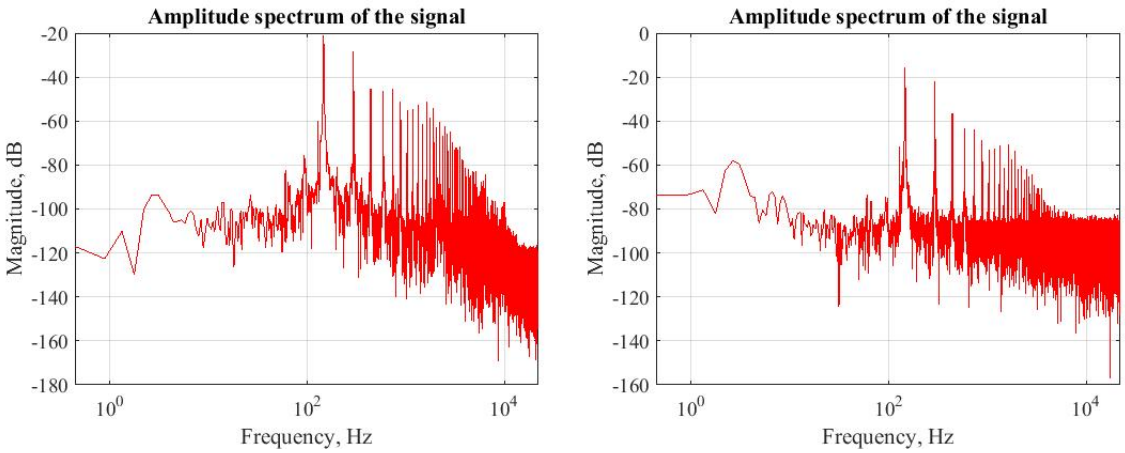

Figure 20: Cello Amplitude Spectrums, Uncompressed(Left), Compressed(Right)

Here we can look at the spectrograms and amplitude spectrum of the compressed and uncompressed Cello samples. When looking at the amplitude spectrums we see that there is a very noticeable trend around -80 decibels on the compressed spectrum which corresponds with our -80 decibel threshold on the VST. One may notice that there are a few "pops" above this threshold, these are most likely a result of the VST performing a release on the compression when these frequencies occur in the sample. Upon observation of the spectrograms one will see that the lower decibel level, blue areas have turned into more green colors that correspond to -80 decibels. One interesting area of observation is how the blue area between 0 and 10,000 decibels had actually raised to the -60 to -40 dB range within the first 0.5 seconds of the sample. These levels are reflected in the amplitude spectrum and are also related to the release of the compression.

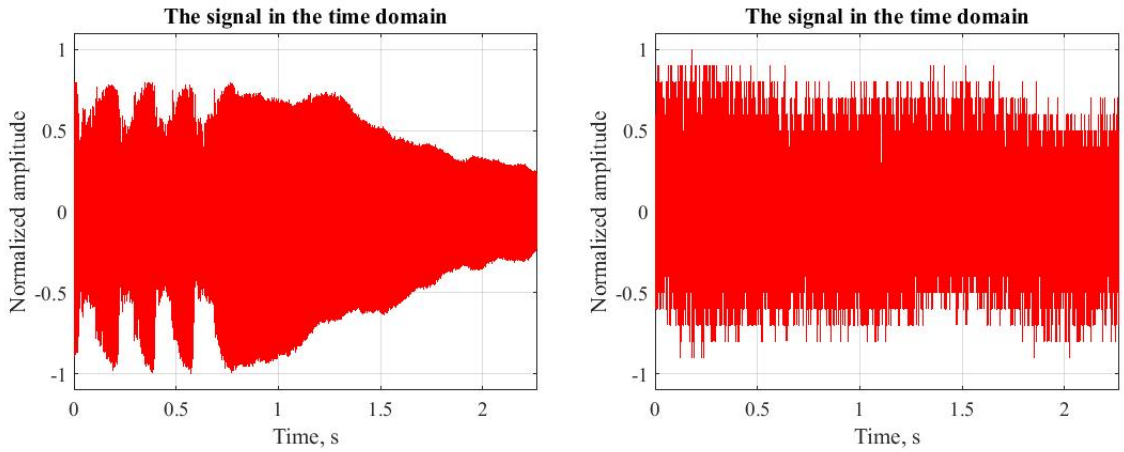

Figure 21: Clarinet Time Domain, Uncompressed(Left), Compressed(Right)

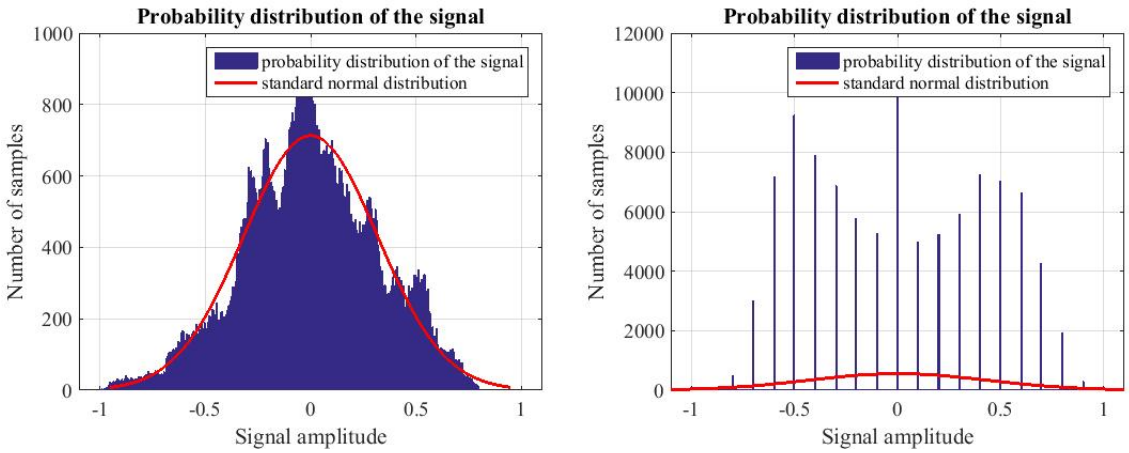

Figure 22: Clarinet Signal Distribution, Uncompressed(Left), Compressed(Right)

These graphs show the normalized signal amplitudes and distribution of those amplitudes for the clarinet in the time domain. The compression effect is much more noticeable on these samples than the cello samples. In the distribution of signal amplitudes we see again that the amplitudes are more disperse after compression.

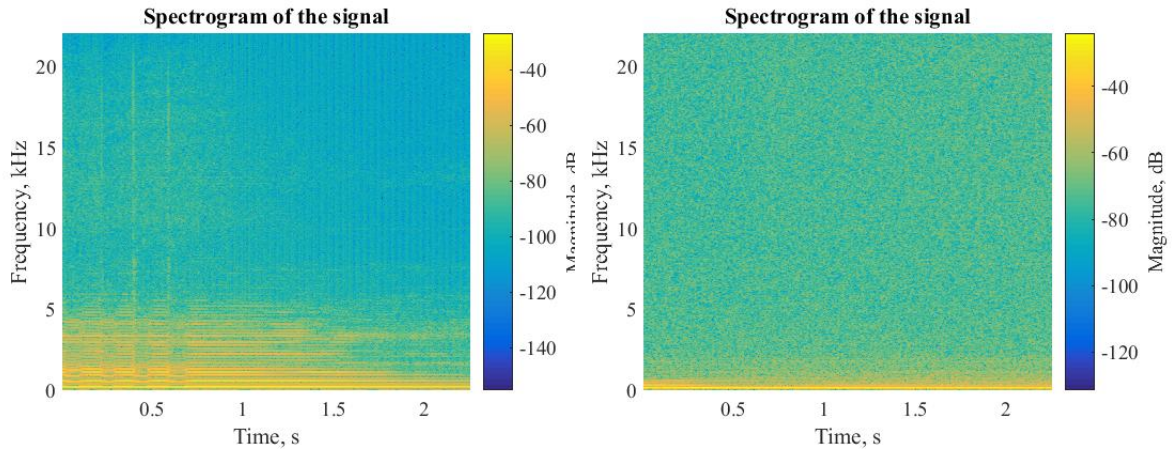

Figure 23: Clarinet Spectrograms, Uncompressed(Left), Compressed(Right)

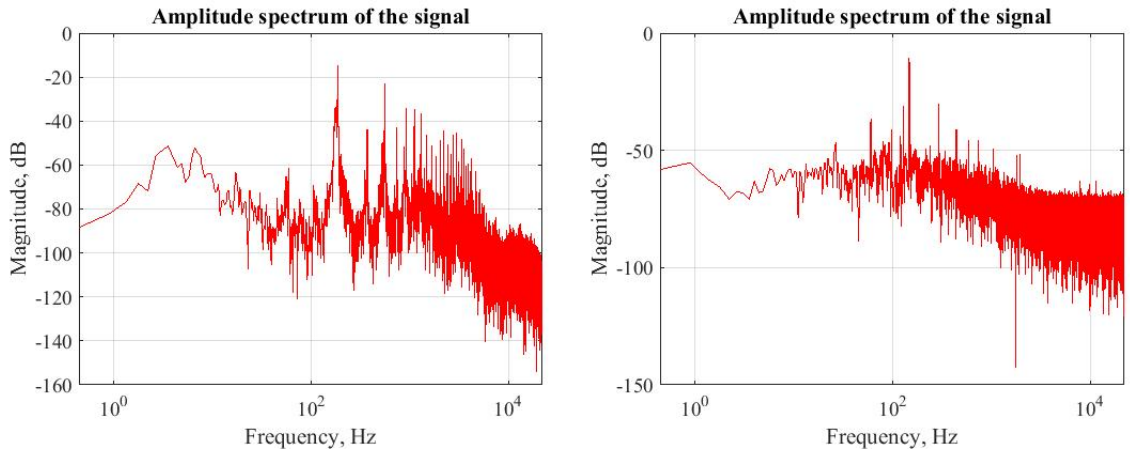

Figure 24: Clarinet Amplitude Spectrums, Uncompressed(Left), Compressed(Right)

The clarinet spectrograms are slightly different from the cello spectrograms in that there were no signal amplitudes raised to the -40 to -60 decibel range after compression. Most of the amplitudes were reduced or increased to be around the -80 decibel threshold. Some "pops" in the amplitude spectrum are visible after compression. With the clarinet we were able to achieve a compression on the dynamic range from 75.69 to 20 decibel.

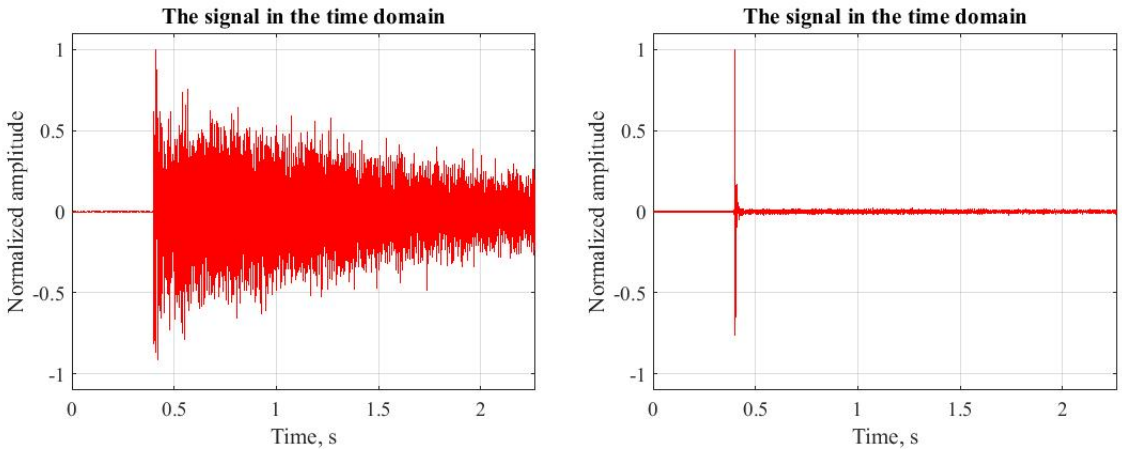

Figure 25: Gong Time Domain, Uncompressed(Left), Compressed(Right)

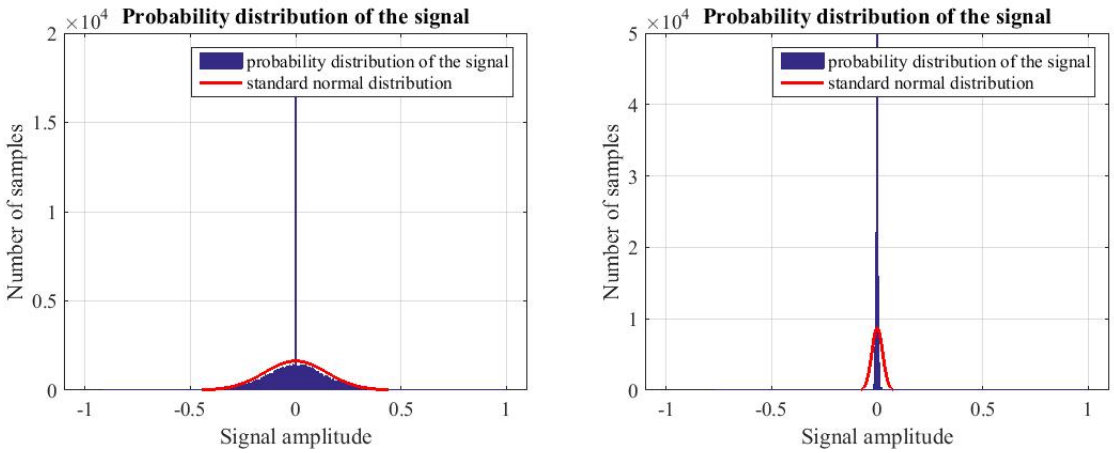

Figure 26: Gong Signal Distribution, Uncompressed(Left), Compressed(Right)

We found the time domain and distribution of normalized signal amplitudes for the compressed gong sample to be a little strange. The normalized signal amplitude seems to show extreme compression even though we had the same settings for the other instruments. Upon further investigation we believe that due to the attack time of our compressor. The large sample seen in the normalized signal was not compressed fully and the further samples were compressed for the entire duration of the sample. This would result in the rest of the samples appearing extremely small compared to the few samples that are larger around the 0.5 second mark of the signal. We also see in the distribution of signal amplitudes that the low amplitudes are more present than any other amplitude.

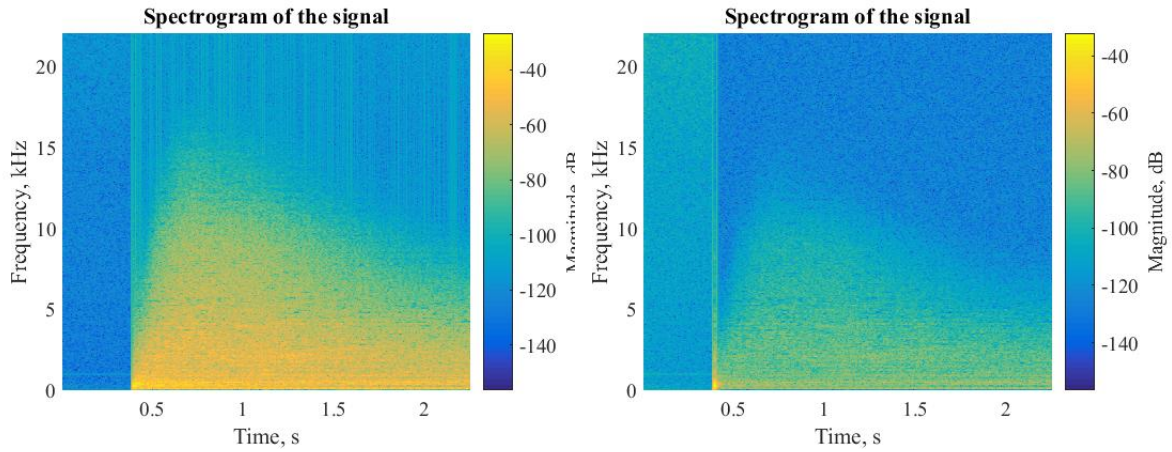

Figure 27: Gong Spectrograms, Uncompressed(Left), Compressed(Right)

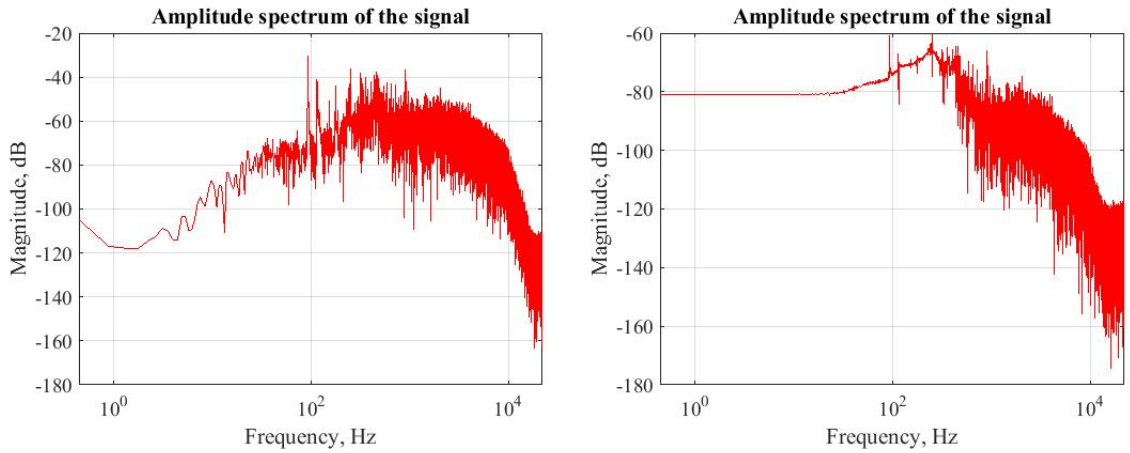

Figure 28: Gong Amplitude Spectrums, Uncompressed(Left), Compressed(Right)

In the spectrogram for the gong we see that most of the yellow, tan spaces have been reduced to the green, -80 decibel region, except for where the first sample that surpassed our compressor's threshold is. Here we see that yellow color spans the entire frequency range. We believe that due to the sound color of a gong, the initial strike will contain extremely complex frequency properties so the frequencies were not attenuated at this point in the sample. An interesting feature of the amplitude spectrum is how the higher frequencies remained at low decibel levels even after compression, this makes sense because compression seems to have been applied for the duration of most of the signal, and due to the gong's complex frequency arrangement we were most likely compressing higher frequencies as well. From the Matlab Sound Analysis we found that the gong's dynamic range was ultimately reduced from 88.737 to 70.56 decibels.

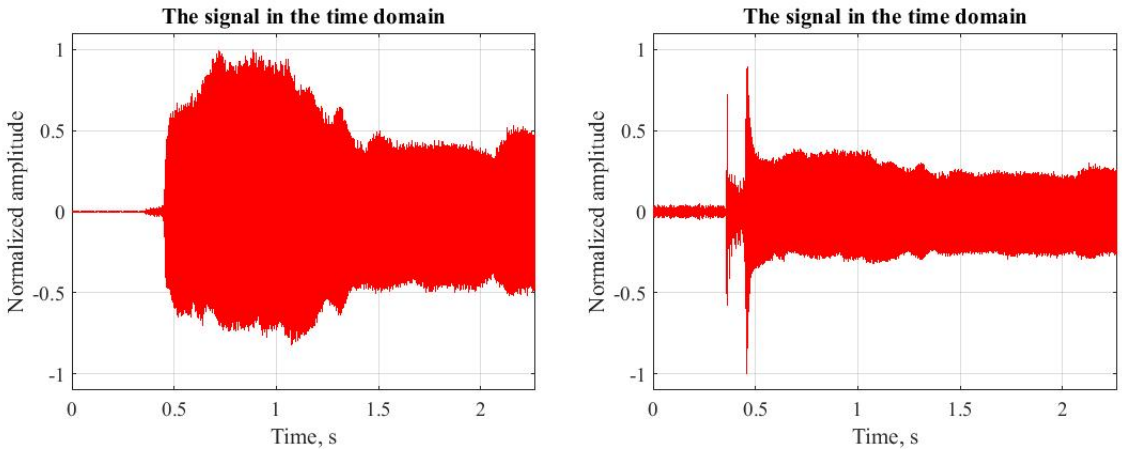

Figure 29: Vocal Time Domain, Uncompressed(Left), Compressed(Right)

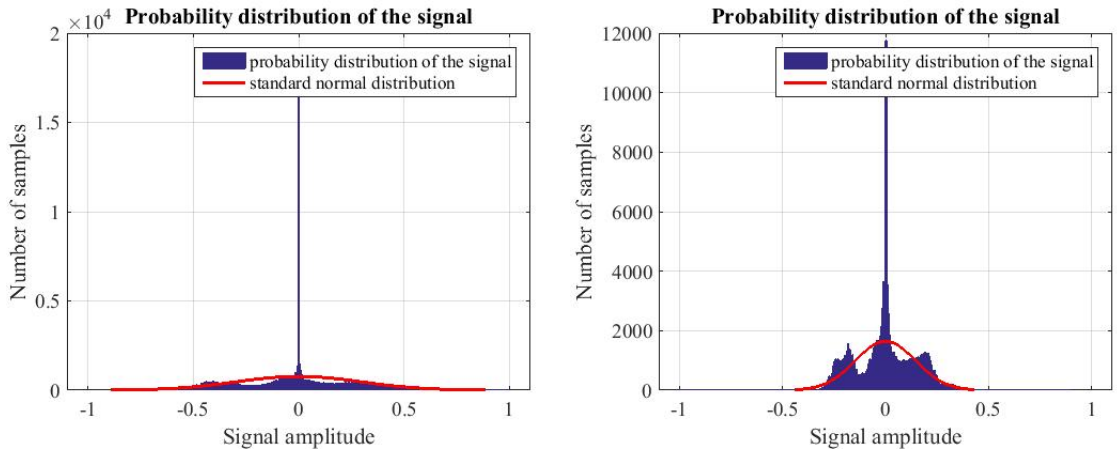

Figure 30: Vocal Signal Distribution, Uncompressed(Left), Compressed(Right)

As with the normalized signal amplitude graphs seen with the gong samples, the compressed vocal sample observes and initial pop where samples first begin. In addition, we see that even samples of extremely quiet noise have been raised when compared in the normalized version. From these graphs we may also observe that more of the samples have had their amplitudes reduced.

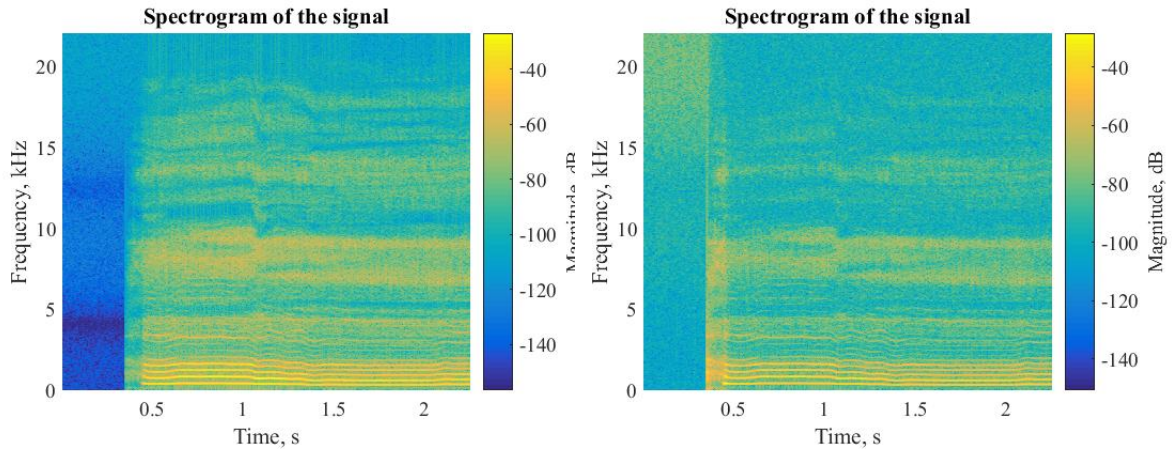

Figure 31: Vocal Spectrograms, Uncompressed(Left), Compressed(Right)

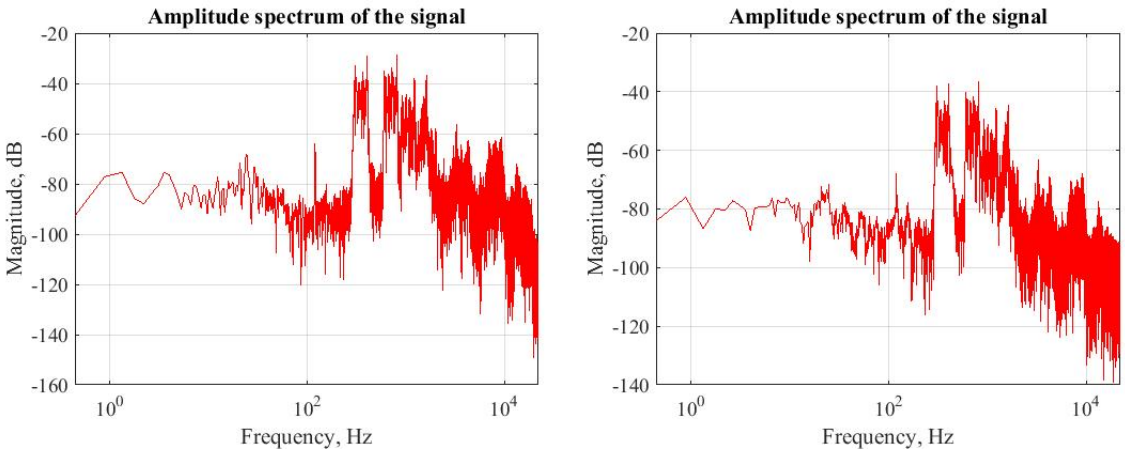

Figure 32: Vocal Amplitude Spectrum, Uncompressed(Left), Compressed(Right)

Looking at the spectrogram and amplitude spectrum plots for the vocal samples we see that they are not as straightforward to analyze as the other instruments. We found that vocal sound is extremely complex similar to the gong's sound. We can see that the magnitudes of many of the frequencies in the spectrogram were reduced heavily compared to the uncompressed sample while some of them remained at the same level. Analyzing the amplitude spectrum does not show us much due to the complex nature of vocal sound, but from the Matlab Sound Analysis we found that the dynamic range of the sample was reduced from 77.59 to 42.73 decibels.

2.4 Further Modularity

We considered the implementation of hard and soft knees as well as a built in spectrum analyzer. For testing purposes we used a separate VST called "Freakascope" and chained it with the compressor in order to see the changes made to the sound samples in real time with adjustment of the compressor parameters. Studying the effects of hard and soft knees on the individual audio samples would be an interesting future endeavor.

3 References

Music-DSP Source Code Archive. Smart Electronix, 2005. Web. 5 Mar. 2015. <http://www.musicdsp.org/showArchiveComment.php?ArchiveID=204>.

Markovic, Bojan. "Music-dsp-collection/chunkware-simple-dynamics." GitHub. 24 Nov. 2012. Web. 5 Mar. 2015. <https://github.com/music-dsp-collection/chunkware-simple-dynamics/tree/master/simpleSource>.

"JUCE for VST Development - Intro." JUCE for VST Development - Intro. Redwood Audio. Web. 11 Feb. 2015. <http://www.redwoodaudio.net/Tutorials/juce_for_vst_development__intro.html>.

Bukac, Hubert, "Acoustic Filters: Part I - Design and Analysis" (2006). International Compressor Engineering Conference. Paper 1751. <http://docs.lib.purdue.edu/icec/1751>

Russell, Daniel. "Acoustic High-Pass, Low-Pass, and Band-Stop Filters." 15 Apr. 2008. Web. 15 Mar. 2015. <http://users.cms.caltech.edu/ ps/All.pdf >.

Desmet, W., and W. Vandepitte. "Finite Element Modeling for Acoustics." LMS Engineering Innovations, 2002. Web. 19 Feb. 2015. <http://pbadupws.nrc.gov/docs/ML0831/ML083150740.pdf >.

Dickens, J.D. "METHODS TO MEASURE THE FOUR-POLE PARAMETERS OF VIBRATION ISOLATORS." Acoustics of Australia, 2000. Web. 11 May 2015. <http://www.acoustics.asn.au/journal/2000/2000_28_1_Dickens.pdf >.

Kinsler, Lawrence E. December 30, 1999. Fundamentals of Acoustics. 4th Edition. New York, NY: John Wiley & Sons. 548.