DEVELOPING A DYNAMIC RANGE COMPRESSOR USING JUCE

Computer Sound, CS 489, Winter Term 2017
Faculty of Mathematics, University of Waterloo

April 3, 2017

Christopher Zaworski,
3B Physics and Astronomy, Faculty of Science

Table of Contents

1.0 Introduction……3
1.1 VST Signal Flow……3
2.0 Compressor Algorithm……3
2.1 Python Prototype……..5
3.0 Implementing a Compressor using JUCE……..8
4.0 Distortion from the Compressor……14
5.0 Conclusion……18
6.0 References……18

1.0 Introduction

This report will highlight my experience using JUCE to develop a simple VST (Virtual Studio Technology) and AU (Audio Unit) plug-in to work within a Digital Audio Workstation (DAW).

JUCE is a series of C++ libraries and toolkits for developing graphical and audio applications. JUCE is developed by the company ROLI, known for releasing their Seaboard midi controllers and more recently their Blocks—a series of compact controllers that allow users to compose using iOS, Mac and PC apps. It is industry standard and used by many music hardware and software companies including Akai Professional, M Audio, Image Line and Korg.

Designed to be simple to use, convenient and tailored for developing real time applications. JUCE is available for free for use with Open-Source projects, with options for both indie developers and businesses to license the software for commercial use.

1.1 VST Signal Flow

VST and AU programs are the industry standard format for most music applications and software. Nearly all DAWs allow for 3rd Party VSTs to be run within them.

DAWs typically

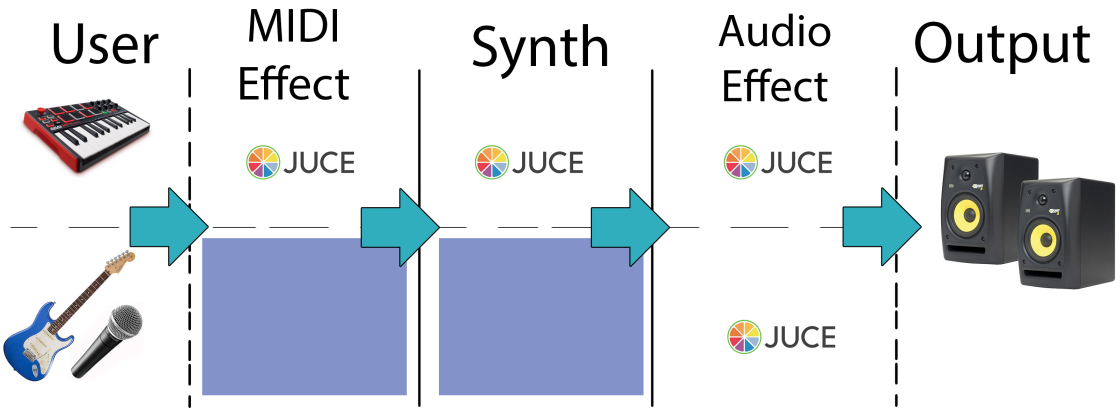

Figure 1: *The steps along the signal pipeline in a DAW where a VST can change the signal, and were a JUCE application can be developed.*

2.0 Compressor Algorithm

There are many mathematical descriptions of how a compressor works. Its effect can very simply be understood as making the loud parts of a signal quieter and the quiet parts of a signal louder. This has the effect of reducing the dynamic range of a sound, making it more or less a consistent volume. Compressors are used at all stages in music production including sound design, mixing and mastering. They can be used for very

subtle effects, such as reducing the amount of “bite” in a kick drum, to very drastic effects used musically.

In general, a compressor “compresses” a signal by reducing the loudest parts of a signal, and then boosting the whole signal. A very simple compressor can be made from three simple parameters: Threshold, Ratio and Gain. Compressors generally follow the very simple equation:

$$Output = Threshold + \frac{Input - Threshold}{Ratio} - Gain \qquad (1)$$

This equation leaves intact the part of the signal below the Threshold, while compressor the signal that rises about the Threshold by a set Ratio. Afterwards an overall Gain can be applied to the signal to bring its level back to the original level. This equation has the effect of “compressing” the whole sound signal down to a constant level, thereby reducing the dynamic range.

One more piece needs to be added to this compressor equation before a suitable VST plug-in can be created. Compressors almost always have a time component to their signal processing. There is a delay to the activation and deactivation of the compressor, as well as a ramp up and ramp down curve. This adds a level of dynamic processing to the compressor equation, as the amount of compression will vary with time.

In order to implement this, the Wet variable needed to be introduced. The Dry/Wet variable is the common terminology in music applications to describe the amount of affected signal that is played. Therefore, the final sound signal will be described by the equation:

$$Final = Compressed * Wet + Input * (1 - Wet); 0 \leq Wet \leq 1 \qquad (2)$$

Where Compressed represents the Output of equation (1), and Input will be the original signal. This then defines a compression amount, and allows a signal to be partially compressed. This will form the basis of the ramp up and ramp down functions.
In musical terms, the ramp up of an effect or process is typically called the Attack, while the ramp down of the effect is the Release. The equation for the attack when implemented into code will look something like this:

$$Wet = \frac{\Delta t_{attack}}{Attack * fs} \qquad (3)$$

Where Attack is a parameter set by the user, in units of seconds (usually milliseconds). fs is the Sample Rate of the signal, and Δt_{attack} is the time passed (in number of samples) since the attack phase begun. Since the Wet variable is limited to be less than 1, this function describes what the wet value will be for each sample in the signal. The Release equation is nearly identical:

$$Wet = 1 - \frac{\Delta t_{Release}}{Release * fs} \qquad (4)$$

Only working in inverse once the compressor begins to release the signal. Again this equation will stop when the Wet variable reaches zero, because of the constraint defined in equation (2).

2.1 Python Prototype

The section above outlines an effective algorithm for creating a dynamic range compressor. Moving this algorithm into working code was first done in python. Mainly for python's easy readability, plotting functions, simple packages for reading and writing audio files and since it is an object-oriented language—it is fairly straight-forward to port the python code over to C++ in order to make a working VST.

The first part of the code is simply importing the packages that will be used, as well as defining the empty variables needed for the compressor.

```
1  import soundfile as sf
2  import matplotlib.pyplot as plt
3  import numpy as np
4
5  file = '440.wav'#'0002 7-Audio.aif'
6
7  INPUT, fs = sf.read(file)
8
9
10 OUTPUT=[]
11 WET = []
12 THRESHOLD = []
13
14
15 Threshold = -12.0
16 Ratio = 2.0
17
18 Attack = 50.0
19 Release = 200.0
20
21 Gain = 0.0
22
23
24 t_att = 0
25 t_rel = 0
26 numSamples = 0
27 attack = False
28 wet = 0.0
```

Figure 2: *defining the initial variables for the python prototype compressor.*

OUTPUT is the list that will eventually hold the data to write to the wave file. *WET* and *THRESHOLD* are empty lists to store the threshold and wet values for plotting purposes. *Threshold*, *Ratio*, *Attack*, *Release* and *Gain* are all parameters that the user can manipulate, and indeed in JUCE will be programmed to be changed in real time. *t_att* and *t_rel* are the Δt values from equations (3) and (4), the same for the *wet* variable. *numSamples* will store the total number of samples in the audio file. When this algorithm is moved to real-time, *numSamples* will be replaced with a buffer of samples that the plugin will then process continually. The *attack* flag, is a Boolean variable that defines whether or not to use the attack part of the algorithm to define the value for *wet* or the release part.

The whole compressor algorithm fits neatly into a python while loop:

```
while numSamples < len(INPUT):

    x_L = INPUT[numSamples][0]
    x_R = INPUT[numSamples][1]

    x_LdB = 20.0*np.log10(abs(x_L))
    x_RdB = 20.0*np.log10(abs(x_R))


    if x_LdB > Threshold or x_RdB > Threshold:
        attack = True


    if attack == True:
        if t_att < (Attack/1000.0)*fs:
            t_rel = 0
            t_att += 1
            wet = float(t_att)/((Attack/1000.0)*fs)
        else:
            attack = False


    if attack == False:
        if t_rel < (Release/1000.0)*fs:
            t_att -= 1
            t_rel += 1
        wet = 1.0 - float(t_rel)/((Release/1000.0)*fs)


    if abs(wet) > 1.0:
        wet = 1.0
        attack = False
        t_att = 0

    if abs(wet) < 0.0:
        wet = 0.0
        t_rel = 0


    wet_LdB = Threshold + (x_LdB - Threshold)/Ratio
    wet_RdB = Threshold + (x_RdB - Threshold)/Ratio

    dry_LdB = x_LdB
    dry_RdB = x_RdB


    y_L = np.sign(x_L)*10.0**((wet_LdB*wet + dry_LdB*(1.0-wet))/20.0)
    y_R = np.sign(x_R)*10.0**((wet_RdB*wet + dry_RdB*(1.0-wet))/20.0)


    OUTPUT.append((y_L,y_R))
    WET.append(wet/2.0)
    THRESHOLD.append(10**(Threshold/20.0))

    numSamples += 1
```

Figure 3: *The compressor algorithm written it python. The while loop loops through the samples in the signal and the if conditionals determine what parts of the compressor are active.*

In this code I use *x* to designate an input signal and *y* to designate the output as a convention. There are two inputs/outputs because most wave files consist of a stereo signal—one for the left (L) channel and one for the right (R). The *x* arrays, which contain audio data as a number between -0.5 and 0.5 for each sample, are first converted into a decibel scale for processing. After which the *x* is first checked if it is above the threshold. If it is then the *attack* flag is true and attack relation is engaged.

Assuming the signal initially does not pass above the threshold, it's important to note

that the *attack* flag is initialized as false, and *wet* is initialized with a value of 0.0. It's important to note that the compressor is bypassed completely if those conditions are met.

Moving on, the $0 \leq wet \leq 1$ constraint is enforced. This is simply a safeguard against the float number, since the condition is also enforced by having maximum *t_att* and *t_rel* values. Note also, how the attack and release 'circuits' are reset when *wet* reaches the corresponding value. Important to note, the 'release circuit' is a 'hard reset' while the 'attack circuit' is a 'soft reset'.

By 'soft reset', I mean to say that the attack calculation will still track the *t_att* variable even if the compressor is in the release phase. This means that if the signal rises above the threshold while the compressor is releasing, the compressor will then attack from the corresponding wet value. By contrast, the release circuit is always 'hard reset' down to zero. Since the compressor will always begin to release from the same point, namely *wet==1.0*.

Below equation (1), the compressor equation, is applied to the data and used to calculate the *wet,* or compressed, signal levels—and the *dry* signal levels are set to the original input *x.* Then equation (2) is applied to the data, converted back from the dB scale and placed in the *y* array for output.

Running this code on a simple drumbeat and plotting it produces:

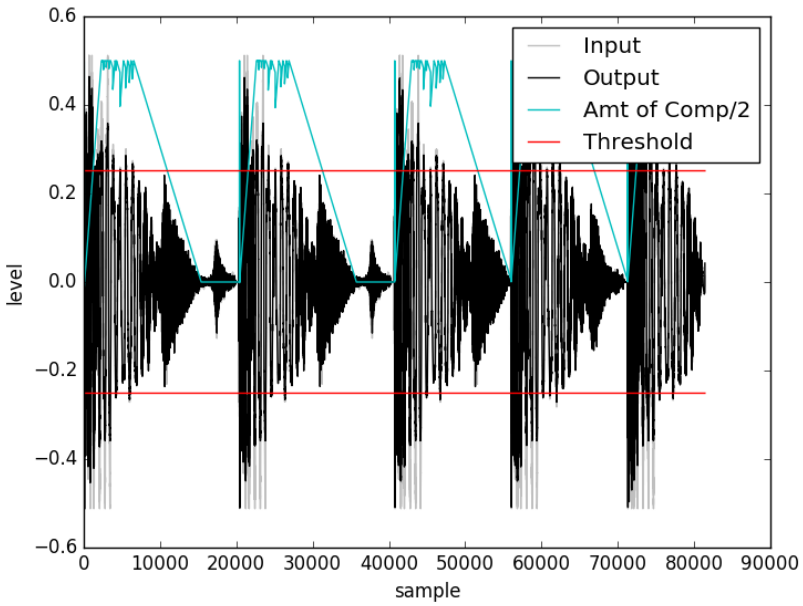

Figure 4: *Running a drumbeat through the compressor algorithm. The grey line is the input signal and the black is the output. The red line represents the threshold value while the blue line is the wet value of the compressor, divided by 2 to fit on the plot—since it varies between 0 and 1.*

Zooming in on one of the beats in this drum look makes it's easier to see what the compressor is doing:

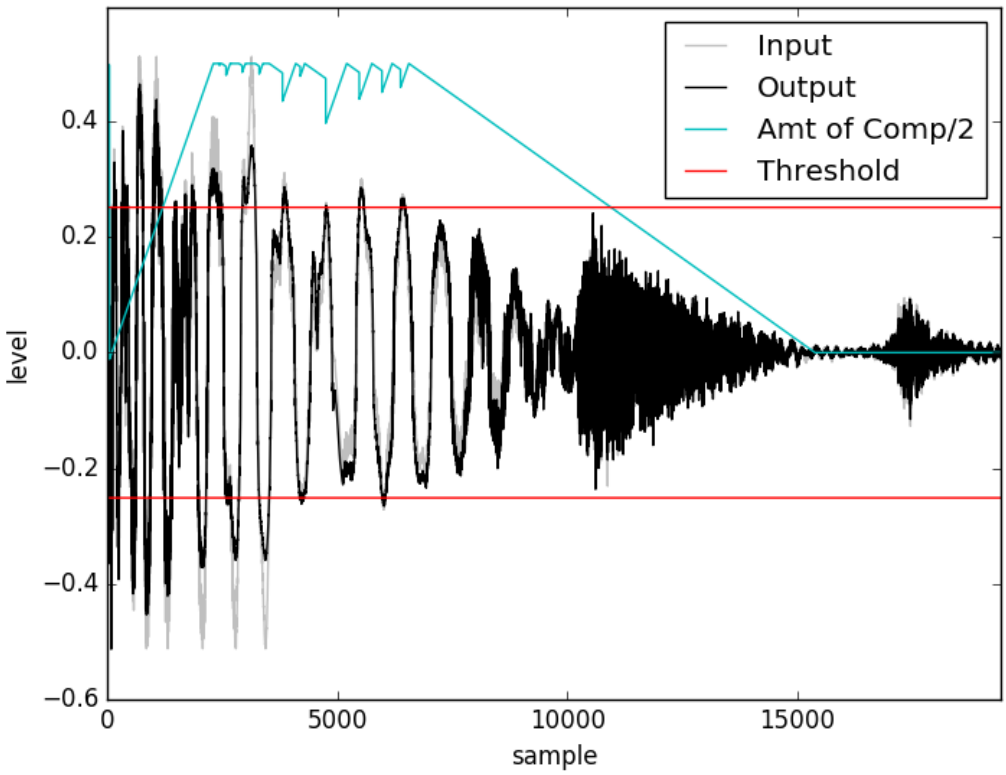

Figure 5: *Zooming in on one beat, the compressor begins to activate once the Input rises above the red line.*

As you can see, the compressor activates nearly instantly since the drumbeat begins with a very loud kick drum. The linear ramp up by the attack function is seen. Once the *wet* value reaches *1.0* it stays relatively constant. The release circuit is engaged momentarily but quickly interrupted as the drummer isn't outputting a constant sound. The second drum hit is not nearly as loud as the first, and does not activate the attack circuit. Thus the release circuit gradually turns the compressor off.

3.0 Implementing a Compressor using JUCE

JUCE is a C++ library, it also ships with the free Projucer application. Projucer once started allows the user to choose from several template projects. Selecting "Audio Plug-In" will create a very simple VST that displays "Hello World" when started.

The next step after choosing what kind of project to create is to choose the Integrated Development Environment (IDE). This is beyond the scope of this report, I simply chose Xcode because I used my Mac to test and compile the code for this project.

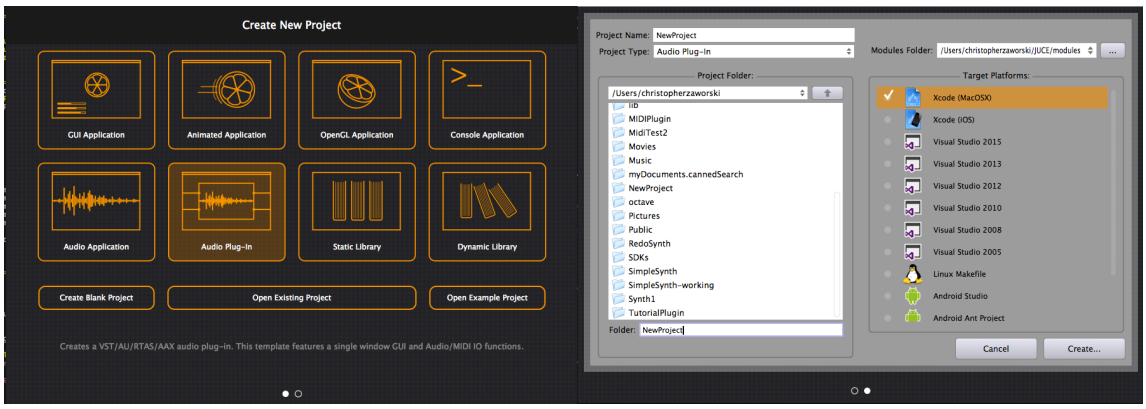

Figure 6: *Creating a new project in JUCE is as simple as choosing a project template and a platform to develop for.*

After creating a template project and compiling it in the IDE, you can then load the template project in a DAW. Here is the “Hello World” AU plug-in loaded in my DAW Ableton Live:

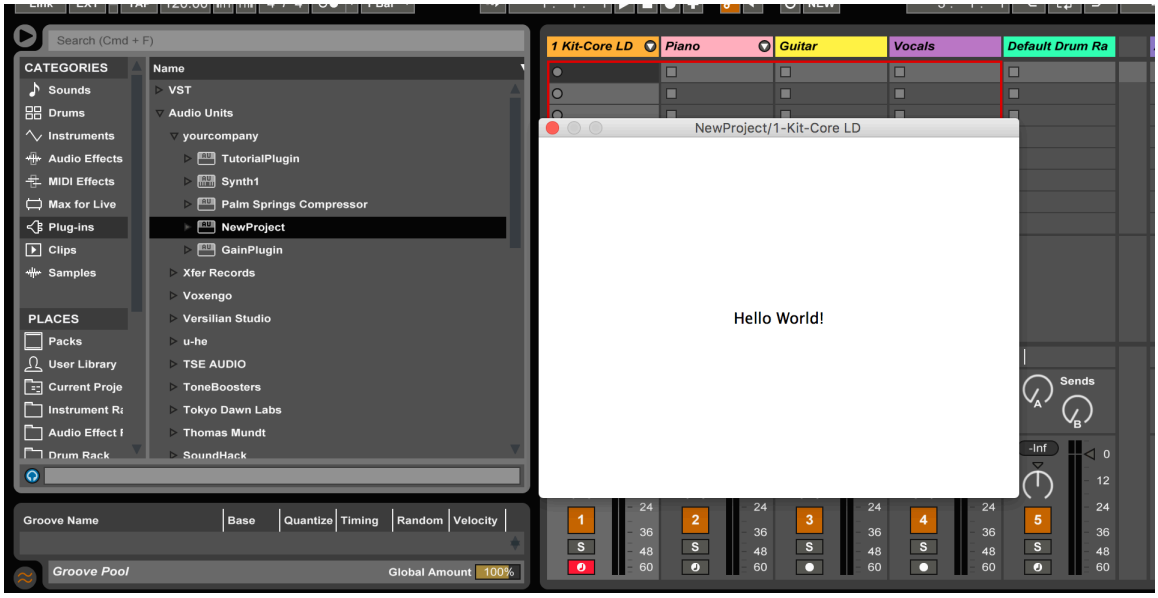

Figure 7: *The initial template created by JUCE, running inside of Ableton Live.*

The Projucer has done most of the heavy lifting in terms of creating an organized file structure and getting the application in a VST/AU format. Looking at the files the Projucer has automatically created:

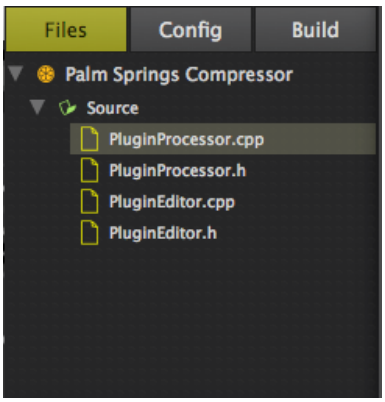

Figure 8: *The files automatically created by JUCE.*

It has created two sets of .cpp (c plus plus) and .h (header) files. The PluginProcessor contains the code that will be affecting the incoming audio signal, it is here the compressor algorithm will be coded. The PluginEditor files contain the code that will define the graphical interface. Luckily Projucer contains a very easy to use graphical editor for designing a simple user interface.

Looking back to the python prototype compressor algorithm, there are five user parameters: *Threshold*, *Ratio*, *Attack*, *Release* and *Gain.* Therefore, the interface will need to have five knobs. In Projucer it's very easy to add the five knobs, as well as some simple clip art and a background to the GUI.

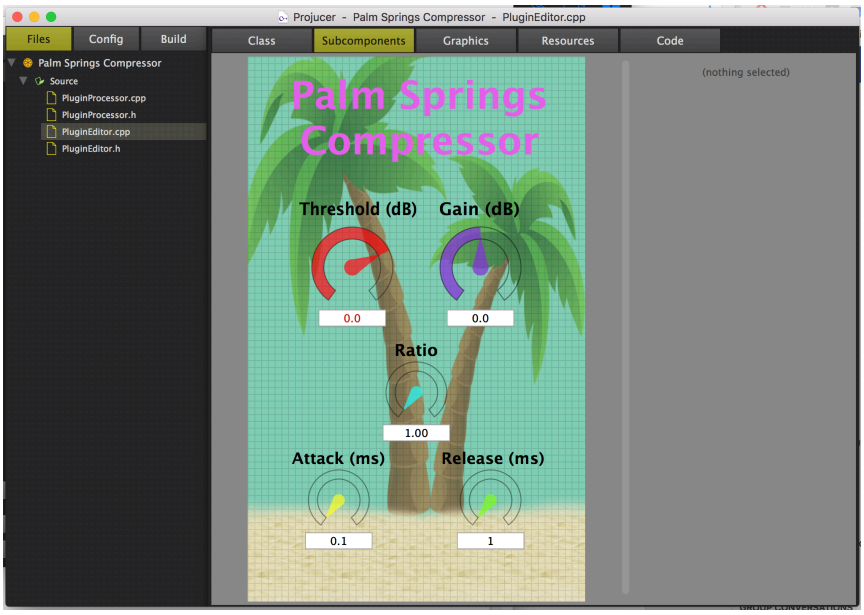

Figure 9: *The GUI editor included in Projucer. Adding interface and modifying objects is handled by drop down menus and standard editing gestures. The Projucer automatically updates the PluginEditor files to reflect any changes made in the graphical editor.*

After the GUI is created, the *UserParams* need to be defined in the PluginProcessor header, in order to serve a functional role. They are assigned to the knobs in the PluginEditor header file:

```
67
68
69      enum Parameters {
70          Gain, //parameter 0, separate by commas - CZ
71          Threshold, // parameter 1 etc
72          Ratio,
73          Attack,
74          Release,
75          totalNumParam
76
77      };
78
79
80  private:
81      float UserParams[totalNumParam];
82
83
```

Figure 10: *Defining the UserParams in the PluginProcessor header file. Here the knobs are assigned to the parameters of the compressor that the user will be able to manipulate.*

The last bit of housekeeping in order to enable the knobs that were created in the PluginEditor is to establish the default methods in JUCE's API. For example the *getParameter* function returns the current value of the parameter and the *setParameter* function will give the parameter a new value. There are quite a few of these methods to update, so I'll just show updating one of them. The rest follow a similar format:

```
73  float GainPluginAudioProcessor::getParameter (int index)
74  {
75      switch(index)
76      {
77          case Gain:// gain is index 0
78              return UserParams[Gain];
79
80          case Threshold:
81              return UserParams[Threshold];
82
83          case Ratio:
84              return UserParams[Ratio];
85
86          case Attack:
87              return UserParams[Attack];
88
89          case Release:
90              return UserParams[Release];
91          default: return 0.0f;
92      }
```

Figure 11: *Assigning standing JUCE methods to the knobs created in the PluginEditor and their corresponding parameters. This is done in the PluginProcessor.cpp file.*

So at this point we have a simple GUI with control knobs that are user-controllable and will update their values in our code. The last step is to implement the compressor

algorithm to process the audio samples that will be entering our VST.

This will be placed in the *ProcessBlock,* method. Here is where the C++ translation of the python prototype code will go. First we must define all the variables that will be used:

```
void GainPluginAudioProcessor::processBlock (AudioSampleBuffer& buffer, MidiBuffer& midiMessages)
//THIS IS WHERE YOU PUT ALL THE AUDIO PROCESSING
{
    int numberOfChannels = 2;

    if (numberOfChannels == 2)
    {
        float* samples0 = buffer.getWritePointer(0);
        float* samples1 = buffer.getWritePointer(1);
    }

        float* samples0 = buffer.getWritePointer(0);
        float* samples1 = buffer.getWritePointer(1);

        bool attack(false);

        float x_L;
        float x_R;
        float x_LdB;
        float x_RdB;

        float y_L;
        float y_R;

        float wet(0);

        int t_att(0);
        int t_rel(0);

        float dry_LdB;
        float dry_RdB;

        float wet_LdB;
        float wet_RdB;

        int fs(getSampleRate());

        int numSamples = buffer.getNumSamples();
        float RMS = buffer.getRMSLevel(0, 0, numSamples);

```

Figure 12: *Initializing all the variables required for the C++ version of the Compressor algorithm. Since C++ is statically typed and compiled, all variables must be pre-defined before they can be used within the code.*

After the variables are defined, the python algorithm can then be recreated verbatim. A key difference includes now defining the *x_L* and *x_R* as pointers to the original samples. This allows the code to modify the sample value then put the modified value back in the location the original pointer was pointing to.

The algorithm also makes use of the default *buffer* class generated by JUCE. This is the key to processing signals in real time. DAWs automatically created a buffer (usually of user determined size) where samples are placed to be processed by the various effects the user can set up. JUCE provides the interface for interacting with this buffer. Allowing the code to loop through all the samples inside the buffer and process them. We must only call the pre-built *getNumSamples()* method from the *buffer* class to get the number of samples and establish the while loop.

```
        while (numSamples > 0)
        {

            x_L = *samples0;
            x_R = *samples1;

            x_LdB = 20.f*log10(std::abs(x_L));
            x_RdB = 20.f*log10(std::abs(x_R));


             if (x_LdB > UserParams[Threshold] || x_RdB > UserParams[Threshold]) {
                 attack = true;
                }

            if (attack == true) {
                if (t_att < (UserParams[Attack]/1000.f)*fs){
                    t_rel = 0;
                    t_att++;
                    wet = (float)t_att/((UserParams[Attack]/1000.f)*fs);
                }
            }
                    else {
                        wet = 1.0;
                        attack = false;
                    }

                if (attack == false){
                    if (t_rel < (UserParams[Release]/1000.0)*fs){
                        t_att--;
                        t_rel++;
                    }
                    wet = 1.0 - (float)(t_rel)/((UserParams[Release]/1000.0)*fs);
                }

            if (wet > 1.0) {
                wet = 1.0;
                attack = false;
                t_att = 0;
            }
            if (wet < 0.0) {
                wet = 0.0;
                t_rel = 0;
            }

            dry_LdB = x_LdB;
            dry_RdB = x_RdB;

            wet_LdB = UserParams[Threshold] + ((x_LdB - UserParams[Threshold])/UserParams[Ratio]);
            wet_RdB = UserParams[Threshold] + ((x_RdB - UserParams[Threshold])/UserParams[Ratio]);

            y_L = ((x_L > 0.0) - (x_L < 0.0))*pow(10.f,(wet_LdB*wet + dry_LdB*(1.f-wet) + UserParams[Gain])/20.0);
            y_R = ((x_R > 0.0) - (x_R < 0.0))*pow(10.f,(wet_RdB*wet + dry_RdB*(1.f-wet) + UserParams[Gain])/20.0) ;

            *samples0 = y_L;
            *samples1 = y_R;

            *samples0++;
            *samples1++;
            numSamples --;
        }
```

Figure 13: *The main compressor algorithm translated from python to C++.*

The code above is functionally identical to the python prototype, only translated into C++ syntax:

That's it! The PalmSpringsCompressor is ready to be complied and run. Opening up the project in Xcode and compiling it will create a functional VST and AU plug-in that will compress audio signals in real-time.

Figure 14: *The final product! A working VST and AU plug-in running inside of Ableton Live and compressing audio signals in real time.*

4.0 Distortion from the Compressor

For this report I've implemented an extremely simple and barebones algorithm for compressing an audio signal. As a consequence of these decisions and the extremely simple calculations used to determine the output signal.

This harmonic distortion is caused by the nonlinear way in which the compressor varies the input signal (McLean, 2012). Professional compressors, both analog and software, feature this same distortion. However great care is taken to ensure that this distortion is pleasant and adds to the harmonic content of the signal.

This is primarily done by the way the compressor handles the attack and release phases of compression. Smoother curves would result in less distortion, as would changing the "knee" of the compressor (the compressor knee wasn't discussed in thi

Running a 440 Hz sine wave through the Palm Springs Compressor can give us a way of characterizing and hopefully fixing the distortion caused by the compressor, the sine wave was generated by Ableton's Operator synthesizer. Below I plotted the Fourier transform of the original sine wave:

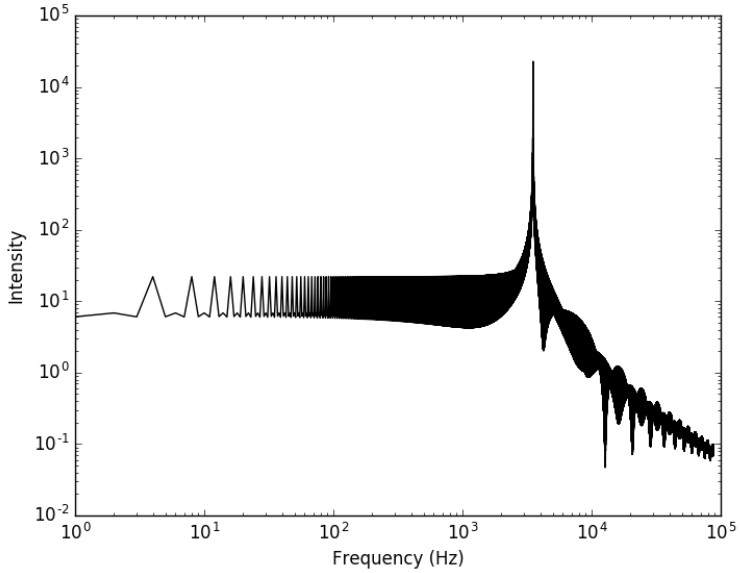

Figure 15: *A plot of the Fourier transform of a sine wave generated at 440 Hz by Ableton's Operator synthesizer.*

Next I ran the sine wave through the Palm Springs Compressor running as a plug-in inside of Ableton:

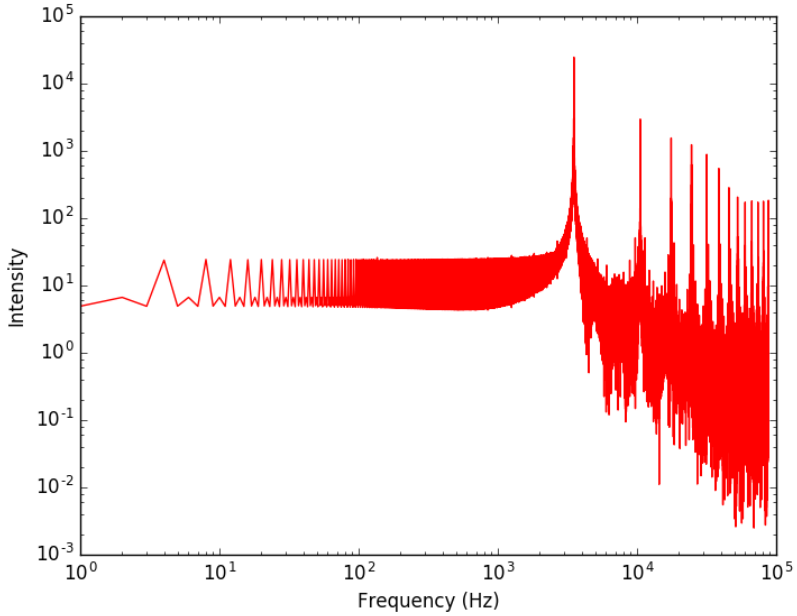

Figure 15: *A plot of the Fourier transform of a sine wave generated at 440 Hz by Ableton's Operator synthesizer and processed by the Palm Springs Compressor using a Threshold of -12 dB, Ratio of 2.00, Attack of 50.0 ms and Release of 200.0 ms.*

In this plot the harmonic distortion is clearly visible as multiples of the original 440 Hz signal. Lastly, for comparison, I ran the signal through Ableton's default Compressor plug-in under identical parameters and took the Fourier Transform:

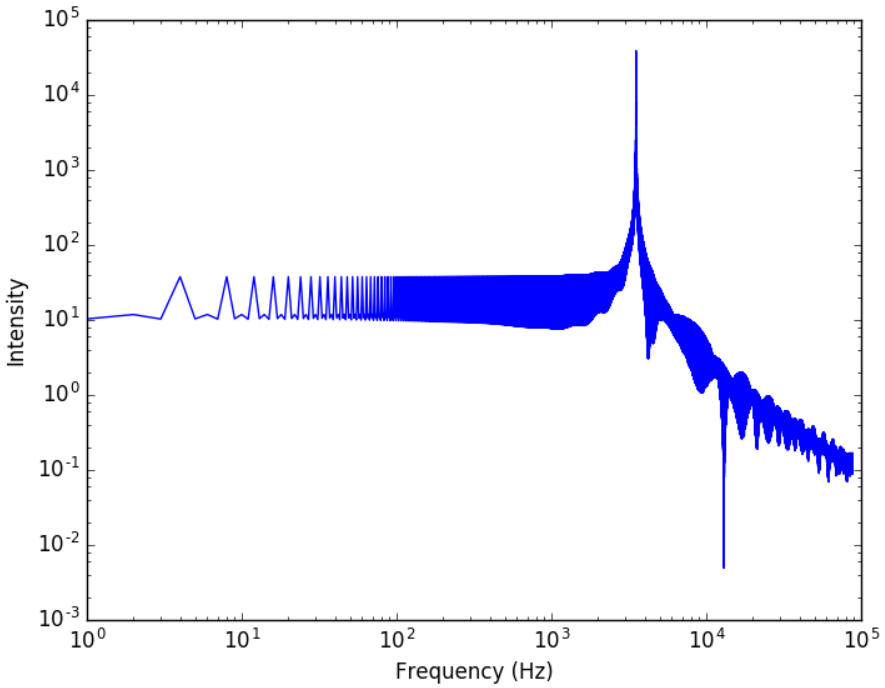

Figure 16: *A plot of the Fourier transform of a sine wave generated at 440 Hz by Ableton's Operator synthesizer and processed by the default Ableton Compressor using a Threshold of -12 dB, Ratio of 2.00, Attack of 50.0 ms and Release of 200.0 ms.*

Unsurprisingly, Ableton's compressor performed much more admirably than the Palm Springs Compressor. Indeed the distortion under identical conditions is barely registering in the Fourier domain.

A clue into the nature of this distortion and how to address it in the future, can be shown when I set the attack and decay of Ableton's Default compressor to 0.1 ms:

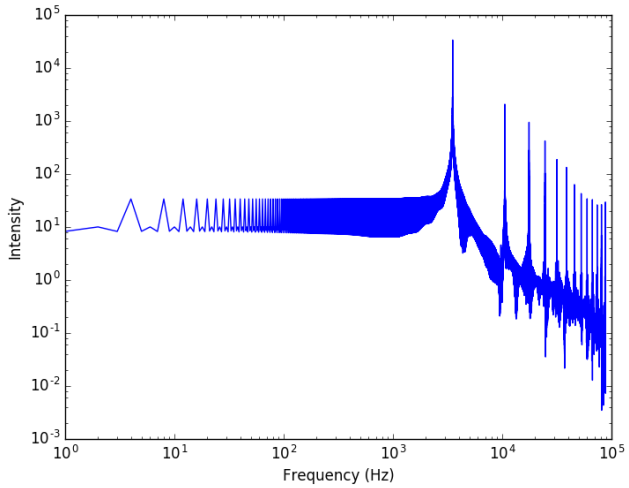

Figure 17: *A plot of the Fourier transform of a sine wave generated at 440 Hz by Ableton's Operator synthesizer and processed by the default Ableton Compressor using a Threshold of -12 dB, Ratio of 2.00, Attack of 0.01 ms and Release of 0.01 ms.*

This plots supports the hypothesis that the key to minimizing this distortion is a much smoother Attack and Release phase. No doubt that Ableton spent many months designing how the Compressor dynamically responds to sound signals in order to create the amount of distortion that they wanted. Plotting all three Fourier Transforms onto a single graph shows how well the Palm Springs Compressor performs for now:

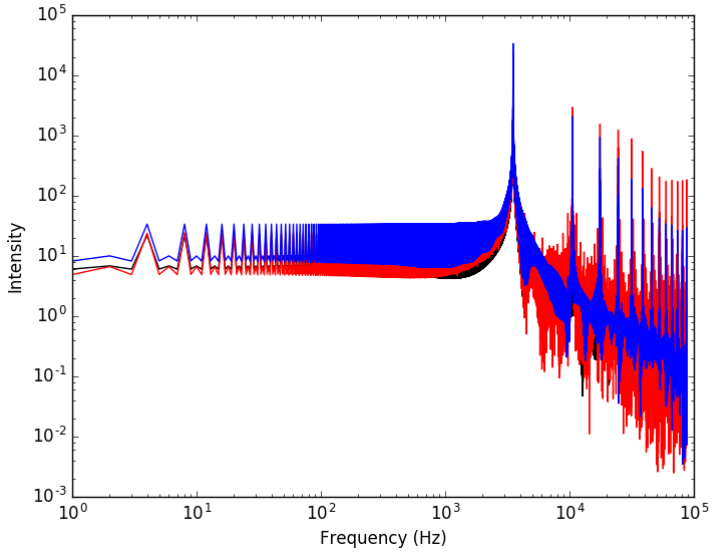

Figure 18: *A plot of the Fourier transform of a 440 Hz sine wave processed by the default Ableton Compressor (blue) and the Palm Springs Compressor (Red). The uncompressed signal is in black.*

This graph makes me very hopeful that given further development and research into developing a softer attack and release curve, as well as possible modifications to the original compressor equation would yield a software compressor that could perform on par with a commercially available VST product.

5.0 Conclusion

JUCE was a very straightforward, simple and fully capable C++ library to use. With minimal C++ and real time programming experience I was able to create a VST/AU plugin and run it inside of my DAW. While the compressor algorithm utilized in this report was functional, it was extremely simplified and limited in what it could accomplish. However this code will serve as an excellent starting point for creating a more complex and functional compressor—or other kind of audio processor VST.

6.0 References:

P. McLean, *Modeling Dynamic Range Compression in the Digital Domain*. Faculty of Architecture, Design and Planning, The University of Sidney, 2012.

I'd like to thank JUCE for developing and providing a great C++ library for developing Audio Plug-ins and Applications, as well as Dr. Mann at the University of Waterloo for his CS489 course for which this report was created for.

JUCE is available for download at juce.com. It is free to use for Open-Source projects such as the one in this report, licensing options are also available for commercial use.

All code used in this report is available at https://github.com/chrizzed/PalmSpringsCompressor. Feel free to use it in your own projects. I only ask that you provide credit and email a link to what you've made to christopherzaworski@gmail.com!