UNIVERSITY OF
WEST LONDON
The Career University

TC70032E – Audio Programming II

Assessment 2: Ping Pong Delay VST3 Plugin

June 2022

Harriet Rena Cerys Drury

21474551

Abstract

This technical report highlights work undertaken to fulfil the assessment brief to create a VST3 ping pong delay effect using C++ and JUCE framework. Included is an analysis and breakdown of research materials to implement VST3 plugins, alongside signal processing theory for delay, distortion and filtering, with a focus on delays and circular buffers. A novel approach of focusing on creating a delay aimed at monophonic melodies is picked, with research highlighting and influencing the design process.

Initial implementation of the ping pong delay is created in MATLAB, to generate graphical results and allow for analysis of distortion types. This is replicated in C++ via the JUCE framework to allow for fast implementation. The additional signal processing is also added at this stage.

Auditory and visual results conclude the ping pong delay works as intended and tested with monophonic samples, it provided a satisfactory tone. It is noted that the implementation of the distortion is minimal, both in the user interface and processing sections. This leads to the conclusion that future work would look at a less simplified user interface and give further control to how distortion is implemented and where in the ping pong delay block chain it occurs.

Table of Contents

Table of Figures

Table of Equations

1. Introduction

This technical report highlights the research, signal processing, implementation and final program to fulfil assessment criteria of creating a VST3 stereo ping pong delay effect with additional signal effects. The focus of this plugin is usage within a digital audio workstation as an effect for monophonic melody lines.

The structure of the report includes a focus of relevant background information on audio plugins, delays, feedback delays and stereo channels. Additional background is given on distortion and filtering, although minimal to highlight the focus on the delay. Prototyping and implementation are included with results and testing occurring. Finally, the future work and conclusion is given to critically evaluate the created program.

VST3 Plugins

For this assignment, creation of a VST3 plugin form is defined. The VST3 is the current version of Steinberg's specification. VST2 was made obsolete in 2018, with the removal of the available software development kit (SDK) (Pirkle, 2019, pp.32). The VST3 defines a set of audio plugin parameters to implement various digital signal processing attributes and behaviours. Plugins are encapsulated C++ objects, requiring an external host application function (Pirkle, 2019, pp.15). The audio plugin is dependent on a host application, figure 1.1 below highlights the signal processing for this.

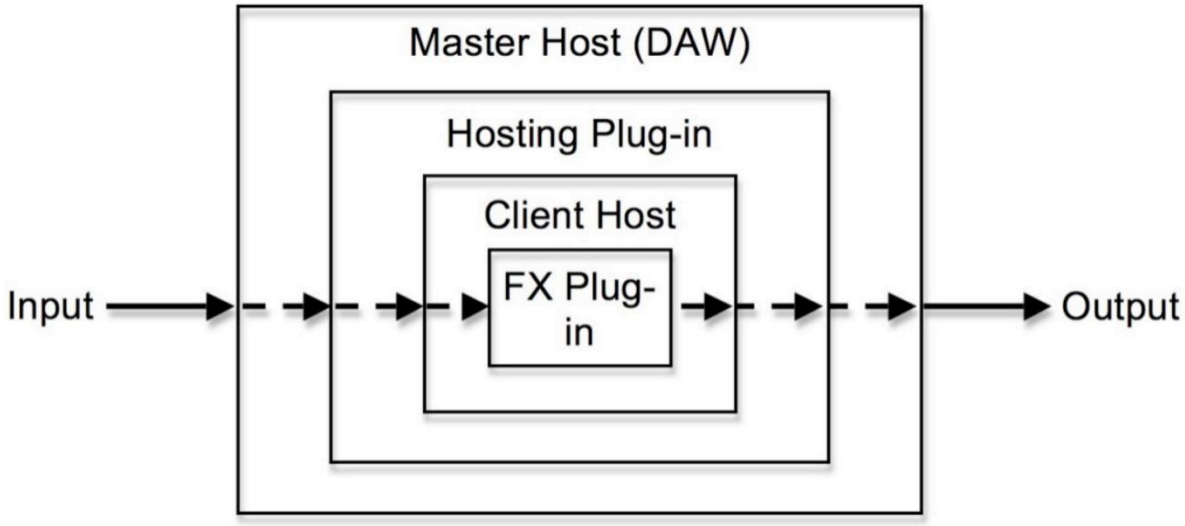

Figure 1.1 Architecture of a Hosting Plug-in (Gibson and Polfreman, 2011)

To create an audio plugin with VST3 formatting, a programmer can refer to the Steinberg Virtual Studio Technology Software Development Kit, allowing developers to create plugins in VST3 format. However, frameworks for creation such as JUCE are commonplace in the creation (Pirkle, 2019, pp.15).

The JUCE framework allows for developers to quickly build an audio plugin from pre formatted classes. JUCE also includes various classes to aid the processing of signals and

MIDI information. This project uses JUCE to complete the requirement of creating a VST3 plugin. The main bulk of the digital signal processing can be seen within the process block of the JUCE project, this is where the delay functions are implemented and changed.

An Overview of The Audio Process

Delay Overview

A delay repeats a signal back to itself with a time delay implemented (Reiss and McPherson, 2015, pp. 21). This effect is utilised for many types of effects, such as a reverb, vibrato and chorus. A delayed signal will play back a signal with a specified delay time, figure 1.2 below highlights a basic delay with a gain function as a block diagram.

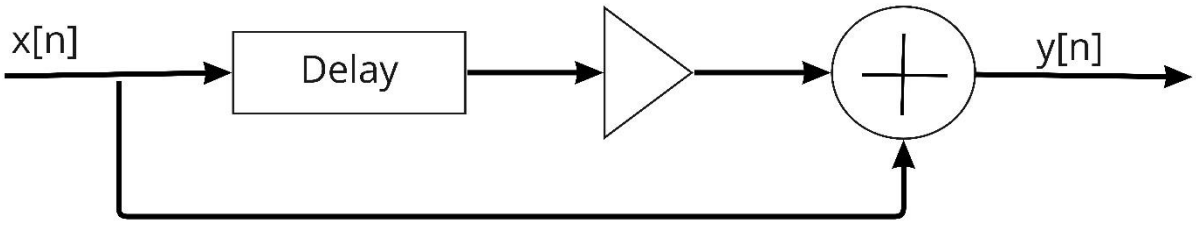

Figure 1.2 A Basic Delay Block Diagram

This can be shown in equation 1 below.

$$y[n] = x[n] + gx[n - N]$$

Equation 1. A Time Delayed Signal Effect

Where y[n] is the outputted signal, x[n] is the input to the system, g is the gain of the delayed signal and N is the time delay in samples. From this, the transfer function of the delay can be defined (equation 2), where z is the z domain, g is gain and N is the number of delaying samples.

$$H(z) = 1 + gz^{-N}$$

Equation 2. Transfer Function for a Delayed Signal

A delay effect is a linear, time-invariant system. This means the relationship between the input (x[n]) and the output (y[n]) are linear. If a constant is added to the input signal, the same weighted sum is observed in the output signal (Tan, 2008, pp. 64). Time-invariance is defined as a time shift that produces the system output (y[n]) to be time shifted by the same amount as the system input (x[n]) (Tan, 2008, pp. 68).

Figure 1.3 is an example of a time delayed sine wave signal. MATLAB code for this is available in appendix 1.

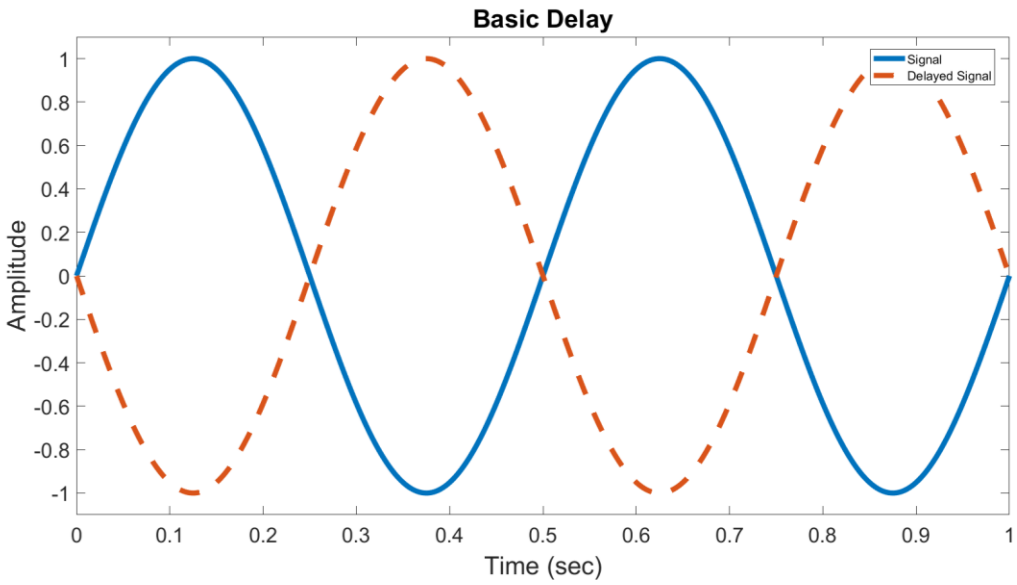

Figure 1.3 A Basic Delay Magnitude Response

A system can include multiple delays which are named delay lines. These may vary the amount of signal delayed or the number of samples which the signal is delayed by. Delay lines build multiple effects and are applied within this project with feedback delay and stereo panning. This creates the targeted ping pong effect.

Delay with Feedback

The aforementioned feedforward domain limits the variability of the output signal. Therefore, audio delay units have a regeneration control to feedback the delayed signal (Zolzer, 2011, pp. 71). This allows for a scaled copy of the output to be sent back to the input, shown with a block diagram in figure 1.4.

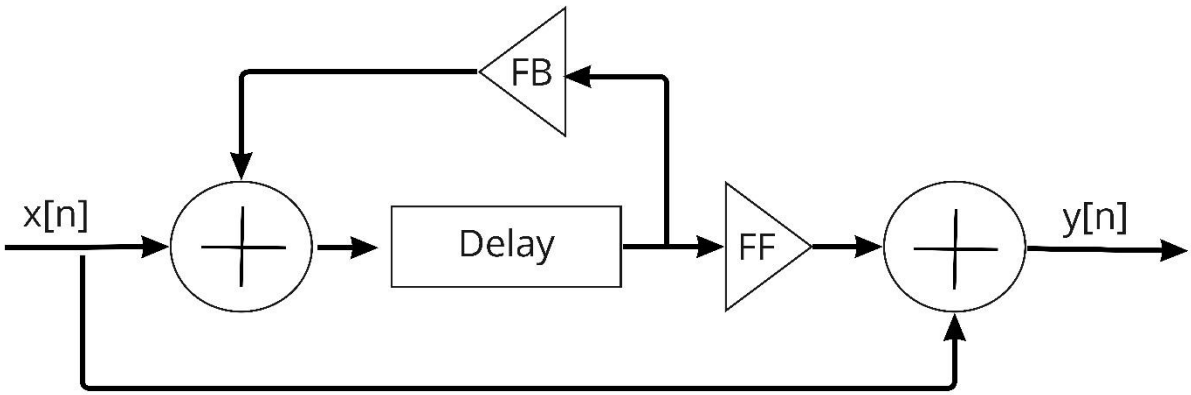

Figure 1.4 A Feedback Delay Block Diagram

Where ‘FF’ denotes the feedforward gain and ‘FB’ denotes the feedback gain. Each signal that goes through the delay line and copied back is subject to the feedback gain, each signal convolved with the unaffected signal is subject to the feedforward gain. Equation 3 is the difference equation for this feedback delay.

$$y[n] = x[n] + g_{FF}\, d[n] \qquad \text{where} \qquad d[n] = x[n-N] + g_{FB}\, d[n-N]$$

Equation 3. Feedback Delay Equation

Where y[n] is the output signal, x[n] is the input signal and d[n] is the feedback signal. Two gains are observed, the feedforward (g_{FF}) and feedback (g_{FB}). 'N' is the number of samples the delay signal is delayed by. This equation provides three changeable variables for the user, the two gain controls and delay amount. The feedback delay is shown in figure 1.5 below. This MATLAB simulation highlights how the inclusion of gain can change the delayed signal. Full code is in appendix 2.

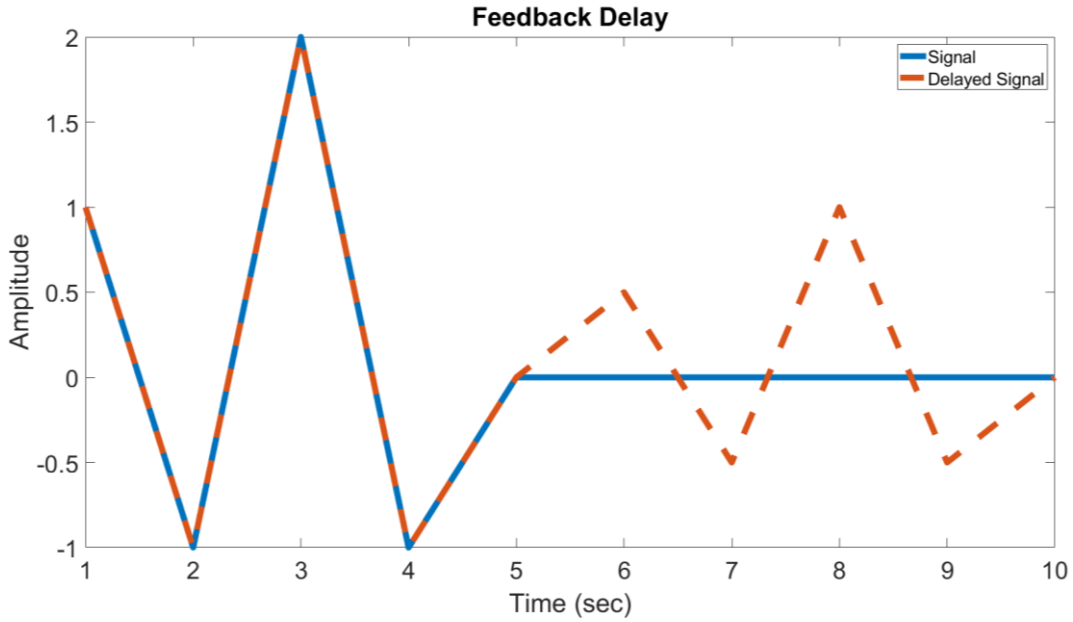

Figure 1.5 A Feedback Delay Magnitude Response

This figure highlights the reduction in the delayed signal with a feedback gain of 0.5. While the figure stops at 10 seconds, this signal would further decrease closer to zero amplitude. It is noted that the delayed signal copies the input signal in the first instance. This feedback delay utilises a buffer to record values.

Buffers

A delay buffer is a method of storing values via allocating an array or vector to store previous values of a signal (Tarr, 2019, pp.273). Within a delay buffer, the data stored is read after the delay time has elapsed. A simple delay will include one read operation (retrieving the delayed signal) and one write operation (storing the current signal) for each sampling period (Reiss and McPherson, 2015, pp.25).

The Circular Buffer

When the end of the buffer is reached, the buffer is looped back to the beginning of the buffer. This is known as a circular buffer. For real time processing, the output signal is produced at the same time as the input signal being acquired. Although there is a little delay

(10 milliseconds), this is not recognisable to the human ear (Smith, 1997, pp.507). Figure 1.6 below is a circular buffer implemented in real time.

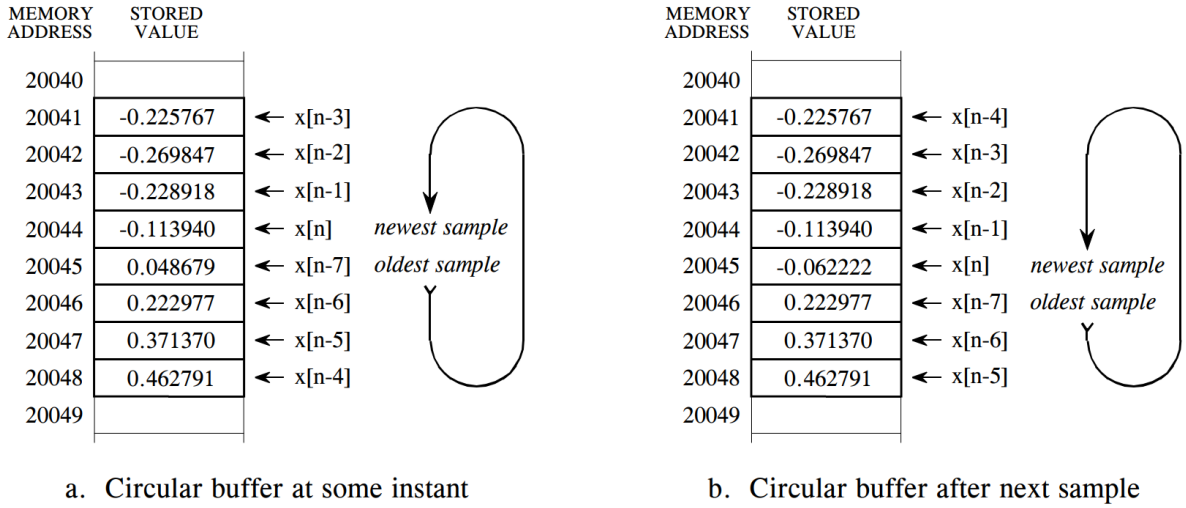

Figure 1.6 Circular Buffer Operation (Smith, 1997, pp.507)

This highlights how the circular buffer is efficient. This is due to the sole use of one pointer to store the current address of the required memory. For the delay, this can be used with a delay length defined as the difference between the indexing pointer and delay pointer.

Stereo Audio Systems

Within audio production, it is common for audio to be outputted from two speakers. This may be headphones or loudspeakers (Tarr, 2019, pp.131). For this, audio may be duplicated to a two channel, stereo signal to be played through one or both speakers at the same time. From this, the perceived location of an audio signal through the stereo field can be changed.

Stereo Panning

Stereo panning is the processing of a mono audio signal to be a stereo signal. Panning is performed by changing the amplitude of the signal in the left and right channels (Tarr, 2019, pp.131). This process produces two signals that can be varied. Within a delay, these channels place samples into the system in a linear format. One channel is affected by the system, then another.

Ping Pong Delay

A ping pong or stereo delay implements two feedback delay lines with the channels of audio signals feedbacking to one another (Reiss and McPherson, 2015, pp.25). The input to each delay line is the channel audio signals. The outputs however, rather than feeding back to itself, feedback to the other channel. This produces a sound that is perceived to bounce between left and right channels. Figure 1.7 is a block diagram for a ping pong delay.

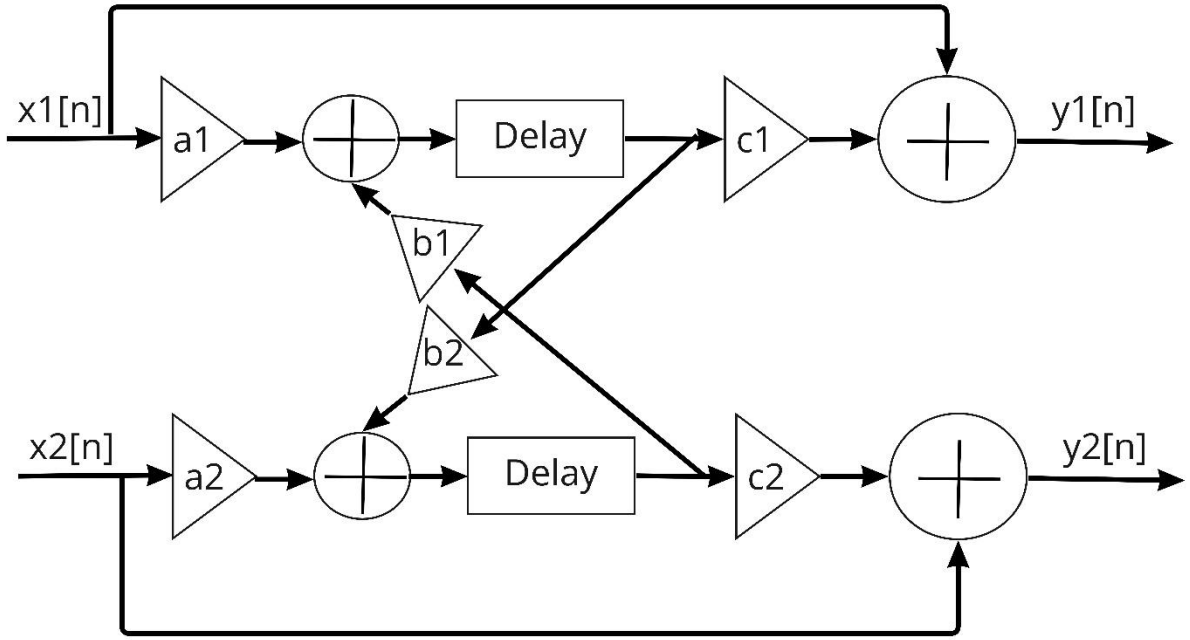

Figure 1.7 The Ping Pong Delay Block Diagram

This block diagram highlights the two delay lines. This requires the usage of two circular buffers, for left and right channels, with two read and write pointers for this. The gain controls provide control of the amplitude at various stages. a_1 and a_2 provide gain control for the inputted signals. b_1 and b_2 provide control of the feedback gain. These may be included as one variable, to control both signals symmetrically. A dry signal is one that has no effects or modifications from a system. A wet mix is the signal that has been affected by a system. c_1 and c_2 are used as wet/ dry mixes. This controls how much of the original signal vs the affected signal is heard when the signals are added together. Delay is the delay of N samples within the Z domain (z^{-N}). It is shown that the delays occur parallel to each other.

In comparison to a multitap delay, the ping pong delay uses multiple memory buffers with two read/write positions. However, the multitap delay uses one buffer with multiple read positions.

For the delay, this feedback of samples at a set number of samples in distance provides the required effect. Further work into these delay lines looks at the usage of delay line interpolation. This is to achieve a smooth change of delay that changes over time to fit the specifications of effects such as the flanger or chorus.

2. Methodology

The methodology of this technical report highlights the implementation of a ping pong delay VST3 plugin via JUCE framework. The focus is the building of two channel delay lines that are connected via a feedback loop to each channel delay. A hard clipping stage is also added as an additional and unique effect.

Designing the Ping Pong Delay & Distortion

As shown in figure 1.7, the ping pong delay can be broken down into stages. Figure 2.1 is one delay line from the diagram. As both are identical, this can be the focus of explaining the sections.

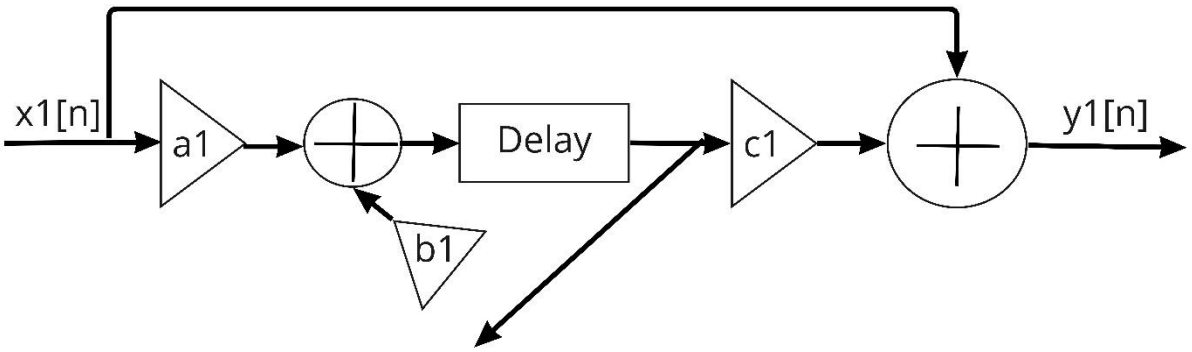

Figure 2.1 One Delay Line Ping Pong Delay Block Diagram

From this diagram, we see similarities with the feedback delay more apparently from figure 1.4. The delay is created by taking N number of samples and delaying the read pointer in comparison to it.

The inclusion of adding other channel signal back is also included. This highlights a diminishing value for the feedback delay. The second phase of the block diagram is the input into a hard clipping distortion with a pre-set threshold. While this threshold can be given to a user as a changeable variable, it is set for this program. Figure 2.2 below is the hard threshold block diagram and equation 4 provides an overview to the effect.

Figure 2.2 The Hard Clip Distortion Block Diagram

Where x[n] is the input sample per channel and y[n] is the output per sample per channel.

$$y[n] = \begin{cases} thresh, & if\ x[n] > thresh \\ x[n], & if -thresh < x[n] < thresh \\ -thresh, & if\ x[n] < -thresh \end{cases}$$

Equation 4. The Branch Equation for Threshold Limiting Hard Clipping

Equation 4 highlights how the signal is affected. This distortion is implemented after the delay phase, to affect the wet delayed signal. Hard clipping is a distortion effect which limits the maximum amplitude of a signal based on the threshold programmed (Tarr, 2019, pp.157).

Infinite Impulse Response Filters

For the methodology, two infinite impulse response filters are added to the program. This is to control the frequency response after the distortion is applied. Previous work with distortion highlights the low-end frequencies being more affected by distortion equations. Therefore, adding a high pass and low pass filter for control of the signal frequencies is added to this VST3 plugin.

An infinite impulse response filter not only relies on the input and past input but, can also rely on the past output of the system (Tan, 2008, pp.304). The output of a digital filter system can be shown in equation 5 below.

$$y(k) = x(k)H(k)$$

Equation 5. The Relationship of a Digital Filter and Digital Signal

Where $H(k)$ is the digital filter system response, $x(k)$ is the Discrete Fourier Transform of the input signal $y(k)$.

MATLAB Prototype

The digital signal processing for the ping pong delay theorised above is primarily prototyped in MATLAB. This allows for the removal of graphic user interface concerns and fast programming implementation. In comparison, C++ has fast runtimes and does not provide limits to parameters and variables. Therefore, a MATLAB prototype can allow for focus on the signal processing, rather than programming.

The focus of the MATLAB code in this instance is the usage of feedback to fill delay buffers. The feedback is not fully inclusive of the ping pong. Once the prototype is complete, these features can be added into the C++ program. Full code can be found in appendix 3.

The focus of the MATLAB prototype is to highlight the multi-channel programming required. As JUCE obtains channel information within a base class, this is not the focus of the C++ programming. Missing also is the hard clip distortion, IIR low pass and high pass filters.

Testing

For testing purposes, no audio is inputted to the MATLAB prototype. Instead, two sine waves of varying frequencies are created for left and right channels. This allows for clear visual magnitude responses to highlight how a signal is changed. The signals are created as the ‘in’ signal. They are contained using two columns in MATLAB. A sampling rate of 44100 is used, as this is common for audio signals. Figure 2.3 is the graph generated from the in signal array of left and right channel.

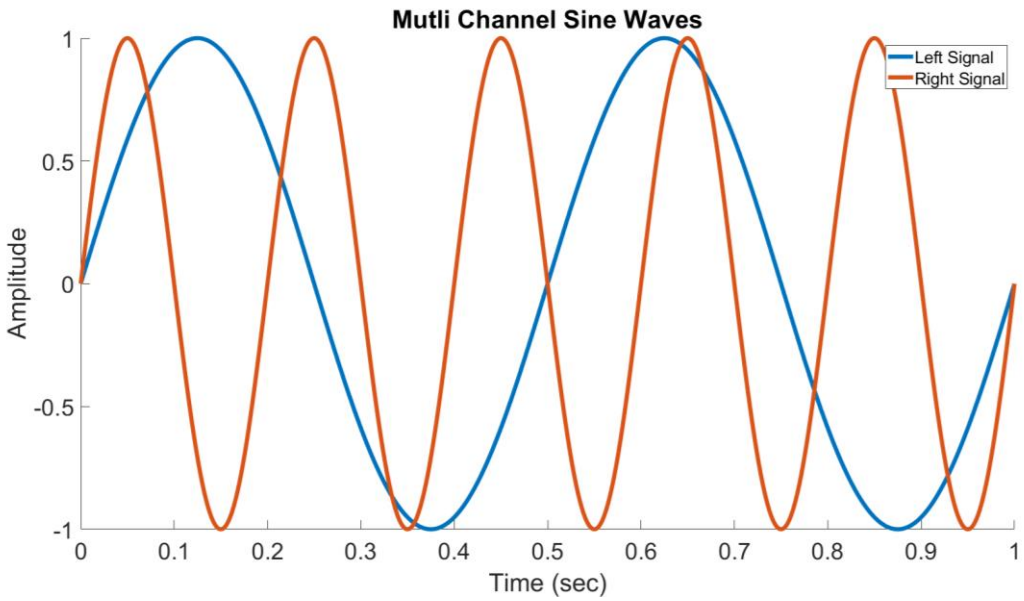

Figure 2.3 A Multi-Channel Sine Wave of Varying Frequency

These signals are then placed into an altered version of the feedback delay used in figure 1.5. This is then added to a dry, unaffected version of the signal with a wet gain control. Figure 2.4 is the result of both channels for this.

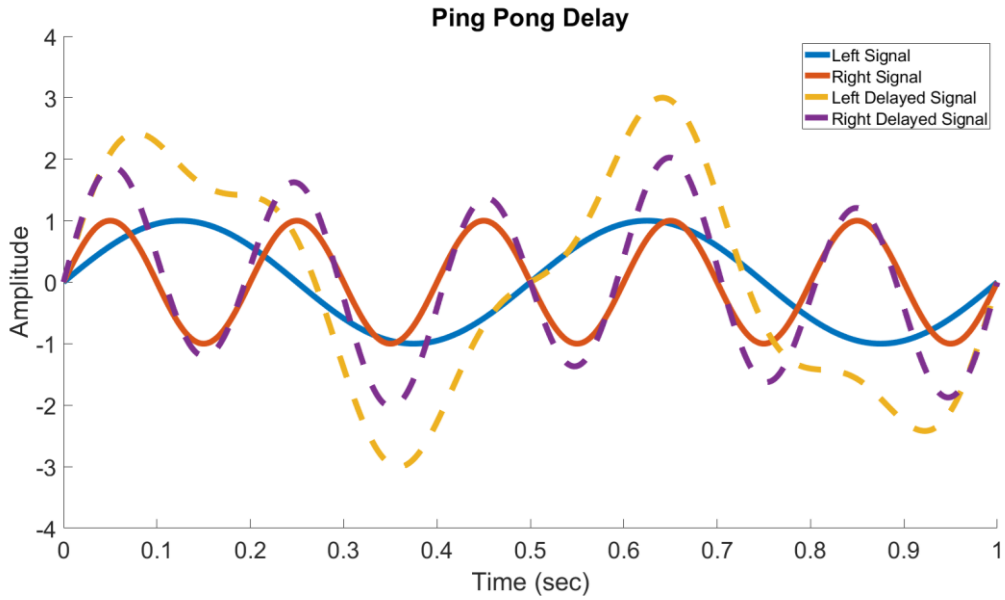

Figure 2.4 The Ping Pong Delay Prototype Magnitude Response

The magnitude response of the prototype ping pong delay shows the delayed signal mixed with the original signal. Missing is the feedback loop between left and right channels. To further this prototype, a larger buffer would be included that can contain the delayed samples. This is due to the linearity of the buffer used. In contrast, the circular buffer allows for the read and write position to change, rather than the usage of more storage.

Creation of a VST3 Plugin

By utilsing the JUCE framework, the creation of a distortion VST3 plugin an occur via the creation of dependency files within the ‘projucer’ JUCE user interface. For this plugin, an extra JUCE module is added; the ‘juce_dsp’ module. This is used for the implementation of filtering. Figure 2.5 is the projucer interface with the additional module circled in red.

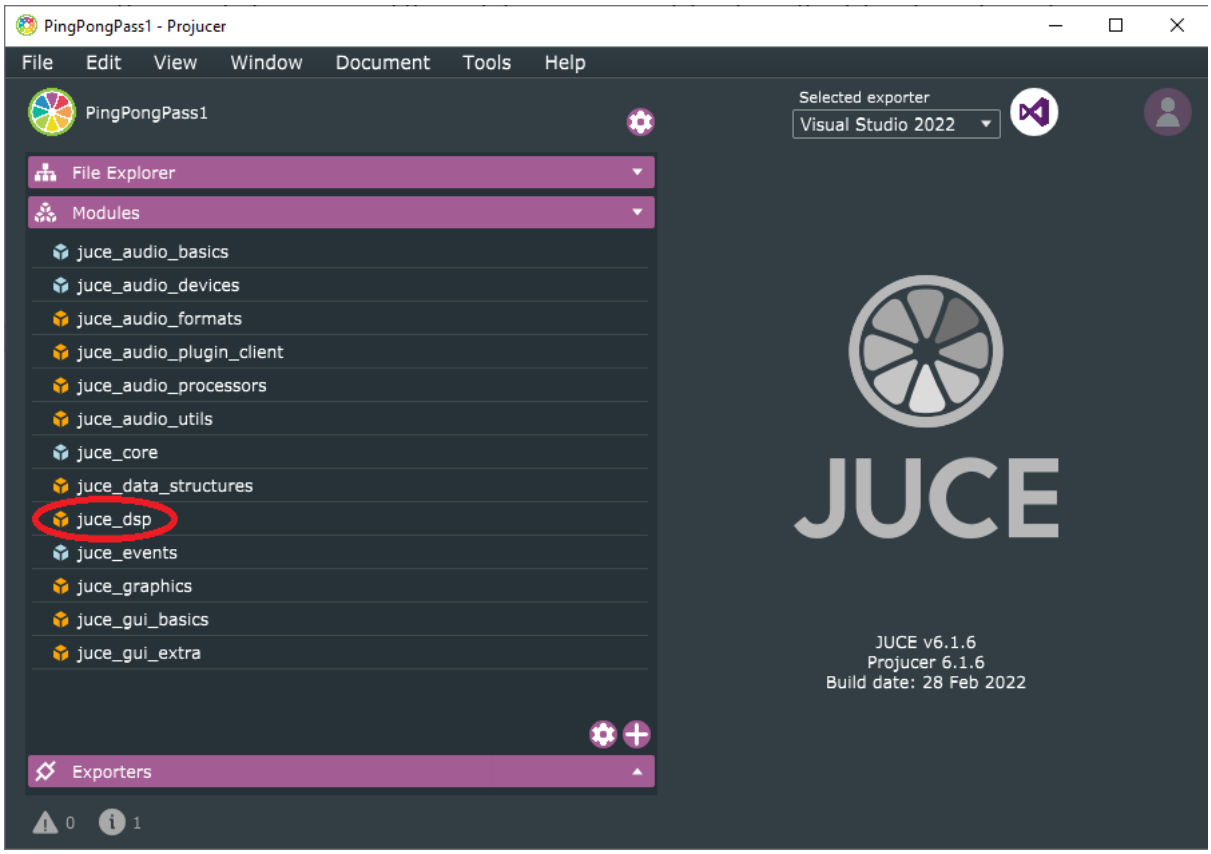

Figure 2.5 The Projucer User Interface & Modules

From the projucer, the JUCE framework is used to create the required base classes and prerequisites for a standalone program and plugin for the code. This create the required framework for rapid implementation in C++. There are two header files (.h) and two coding files (.cpp). The main focus of this report is the signal processing side of the code, which occurs in ‘PluginProcessor.cpp’ and ‘PluginProcessor.h’. The input from users and user interface are the focus of the ‘PluginEditor.cpp and PluginEditor.h’ files.

Program Overview

It is noted that the focus of this report is the code built on top of the JUCE framework. The dependencies of the JUCE framework, while touched upon, is not fully explored. Figure 2.6 provides a flowchart for the project.

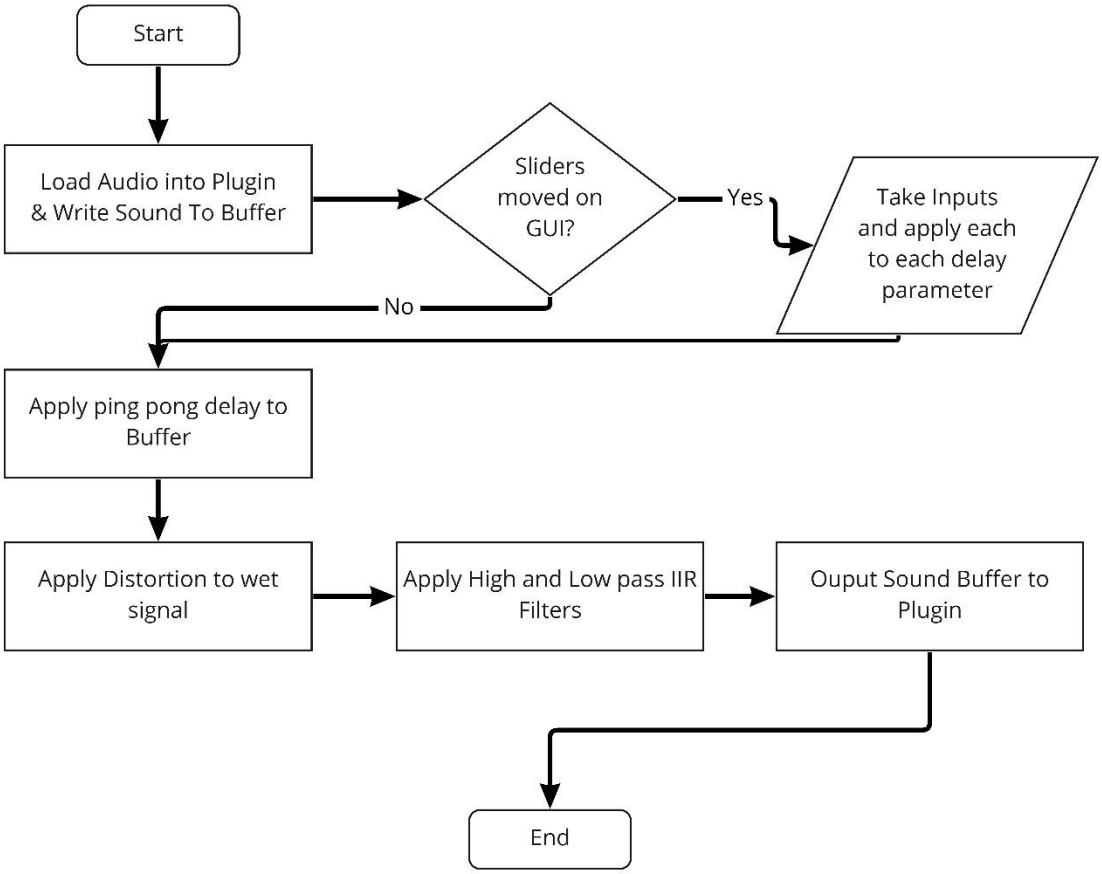

Figure 2.6 A Flowchart Overview of The Ping Pong Delay Plugin

To obtain clear code, sections are split into delay, distortion and filtering functions. These are called within 'PluginProcessor.cpp'. It is noted that for usage as a standalone plugin, line 23 requires a change to the lowpass and highpass declaration. Instead of using the 'getSampleRate()' function, it is required to replace both with '44100' as the sample rate.

PluginProcessor Source Code Design

Created functions are declared in 'PluginProcessor.h', as part of the private portion of the class. These are shown in figure 2.7 below.

```
private:

    juce::AudioProcessorValueTreeState::ParameterLayout createParameters();

    juce::AudioBuffer<float> delayBuffer;
    int delayBufferSamples;
    int delayBufferChannels;
    int delayWritePosition;

    //======================Delay Functions=====================================
    void PingPongPass1AudioProcessor::applyPingPong(juce::AudioBuffer<float>& buffer, const int numSamples);
    void PingPongPass1AudioProcessor::movePosition(int localReadPosition, int localWritePosition, float
readPosition, float* delayDataL, float* delayDataR, const float inL, const float inR, float outL, float outR,
float* dataL, float* dataR, float feedback, float mix, int sample);

    //=================Distortion Functions=====================================
    void PingPongPass1AudioProcessor::applyDistortion(juce::AudioBuffer<float>& buffer, const int numSamples,
float* channelData);
    float PingPongPass1AudioProcessor::hardClip(const float& sample, float thresh);

    //====================IIR Filter=============================================
    void PingPongPass1AudioProcessor::updateFilter();
    juce::dsp::ProcessorDuplicator<juce::dsp::IIR::Filter <float>, juce::dsp::IIR::Coefficients <float>>
lowPassFilter;
    juce::dsp::ProcessorDuplicator<juce::dsp::IIR::Filter <float>, juce::dsp::IIR::Coefficients <float>>
highPassFilter;
```

Figure 2.7 The Functions and Parameters within PluginProcessor.h

It is noted that the values from the user interface are also included within this figure. The functions have been split into sections; delay, distortion and filtering. This is for ease of editing and reading. To add the ping pong delay, the signal is sent through the delay buffer and manipulated using applyPingPong() and movePosition(). Figures 2.8 and 2.9 below are these functions.

```
void PingPongPass1AudioProcessor::applyPingPong(juce::AudioBuffer<float>& buffer, const int numSamples) {

    // Obtaining values for delay from UI
    auto balanceValue = apvts.getRawParameterValue("BALANCE");
    float balance = balanceValue->load();
    auto delayValue = apvts.getRawParameterValue("DELAY");
    float delay = balanceValue->load();
    auto feedbackValue = apvts.getRawParameterValue("FEEDBACK");
    float feedback = balanceValue->load();
    auto mixValue = apvts.getRawParameterValue("MIX");
    float mix = balanceValue->load();

    int localWritePosition = delayWritePosition;

    // Write Pointers for left and right
    float* dataL = buffer.getWritePointer(0);
    float* dataR = buffer.getWritePointer(1);
    float* delayDataL = delayBuffer.getWritePointer(0);
    float* delayDataR = delayBuffer.getWritePointer(1);


    for (int sample = 0; sample < numSamples; ++sample) {

        const float inL = (1.0f - balance) * dataL[sample];
        const float inR = balance * dataR[sample];
        float outL = 0.0f;
        float outR = 0.0f;

        // Read position via abs value
        float readPosition =
            fmodf((float)localWritePosition - delay + (float)delayBufferSamples, delayBufferSamples);
        int localReadPosition = floorf(readPosition);

        if (localReadPosition != localWritePosition) {
            movePosition(localReadPosition, localWritePosition, readPosition, delayDataL, delayDataR, inL,
inR, outL, outR, dataL, dataR, feedback, mix, sample);
        }

        if (++localWritePosition >= delayBufferSamples)
            localWritePosition -= delayBufferSamples;
    }

    delayWritePosition = localWritePosition;
}
```

Figure 2.8 The applyPingPong() Function

```
void PingPongPass1AudioProcessor::movePosition(int localReadPosition, int localWritePosition, float
readPosition, float* delayDataL, float* delayDataR, const float inL, const float inR, float outL, float outR,
float* dataL, float* dataR, float feedback, float mix, int sample) {
    float fraction = readPosition - (float)localReadPosition;
    float delayed1L = delayDataL[(localReadPosition + 0)];
    float delayed1R = delayDataR[(localReadPosition + 0)];
    float delayed2L = delayDataL[(localReadPosition + 1) % delayBufferSamples];
    float delayed2R = delayDataR[(localReadPosition + 1) % delayBufferSamples];
    outL = delayed1L + fraction * (delayed2L - delayed1L);
    outR = delayed1R + fraction * (delayed2R - delayed1R);

    dataL[sample] = inL + mix * (outL - inL);
    dataR[sample] = inR + mix * (outR - inR);

    delayDataL[localWritePosition] = inL + outR * feedback;
    delayDataR[localWritePosition] = inR + outL * feedback;

    //========Distortion on Wet Mix==========
    hardClip(delayDataL[localWritePosition], 0.4f);
    hardClip(delayDataR[localWritePosition], 0.4f);
    // Uncomment this ^^^^^ For wet distortion
}
```

Figure 2.9 The movePosition() Function

These functions are created to minimise the code in the process block and create easy to read code. Distortion is then added with applyDistortion() and hardClip() functions shown in figures 2.10 and 2.11 below.

```
void PingPongPass1AudioProcessor::applyDistortion(juce::AudioBuffer<float>& buffer, const int numSamples,
float* channelData) {
    for (int sample = 0; sample < numSamples; ++sample) {
        for (int channel = 0; channel < buffer.getNumChannels(); ++channel) {
            float out; float thresh = 0.7f; // Thresh value is set for the user. This could be added as a UI
feature
            const float in = buffer.getSample(channel, sample);
            out = hardClip(in, thresh);

            buffer.setSample(channel, sample, out);
        }

    }
}
```

Figure 2.10 The applyDistortion() Function

```
float PingPongPass1AudioProcessor::hardClip(const float& sample, float thresh) {
    float out;
    if (sample >= thresh) {
        out = thresh;
    }
    else if (sample <= -thresh) {
        out = -thresh;
    }
    else {
        out = sample;
    }
    return out;
}
```

Figure 2.11 The hardClip() Function

For the purpose of this plugin, no changeable parameters are given to the user for the distortion block. This is to focus on the delay portion of code. Similarly, the filtering process is applied with no control to the cutoff frequencies or resolutions. These can easily be added to the user interface for more control of the tone. However, the simplicity of the user interface is kept via only delay controls.

The Filtering Block

The JUCE base classes are explicitly used for filtering purposes of this plugin. This allows for fast implementation and clean code. It is also a secondary digital signal processing focus for this report. The filters are created within 'PluginProcessor.h' and require initial setup with the sample rate and channel number. Figure 2.12 is the changing of the filter response with the inputted signal. This is the sole programming implemented for this.

```
void PingPongPass1AudioProcessor::updateFilter() {

    // Low Pass
    float resLow = 2.0f; // Values are set for the user. This could be added as a UI feature
    float freqLow = 15000.0f;
    *lowPassFilter.state = *juce::dsp::IIR::Coefficients<float>::makeLowPass(getSampleRate(), freqLow,
resLow);

    // High Pass
    float resHigh = 2.0f;
    float freqHigh = 200;
    *highPassFilter.state = *juce::dsp::IIR::Coefficients<float>::makeHighPass(getSampleRate(), freqHigh,
resHigh);
}
```

Figure 2.12 The updateFilter() Function

While rudimentary in application, this filter works as anticipated and can be further built upon with the inclusion of the program designers own filter coefficients.

The function ‘processBlock’ is a predefined function by JUCE. This is where the aforementioned functions are called. It functions in real time via the audio buffer provided. This obtains data from the channel signals and allow for implementation of the delay, distortion and filters. Figure 2.13 is the process block for this VST3 Plugin. Included is an optional delay which affects the mixed wet/dry signal. This inclusion is to highlight the differences that can be placed when designing this plugin. A gain control within this sector may provide more control.

```
void PingPongPass1AudioProcessor::processBlock (juce::AudioBuffer<float>& buffer, juce::MidiBuffer&
midiMessages)
{
    juce::ScopedNoDenormals noDenormals;

    auto totalNumInputChannels  = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();
    const int numSamples = buffer.getNumSamples();
    //=========Ping Pong============
    applyPingPong(buffer, numSamples);

    //======Optional Distortion=====
    // Affects the Channel data separately. Wet/dry mix affected, comment out if wanting wet mix distorted and
uncomment in delay block (movePosition())
    // Comment From here
    /*for (int channel = 0; channel < totalNumInputChannels; ++channel)
    {
        float* channelData = buffer.getWritePointer(channel);
        applyDistortion(buffer, numSamples, channelData);
    } // Comment out to here */

    for (int channel = totalNumInputChannels; channel < totalNumOutputChannels; ++channel)
        buffer.clear(channel, 0, numSamples);

    //===========Filters============
    juce::dsp::AudioBlock<float> block(buffer);
    updateFilter();
    lowPassFilter.process(juce::dsp::ProcessContextReplacing<float> (block));
    highPassFilter.process(juce::dsp::ProcessContextReplacing<float>(block));
}
```

Figure 2.13 The Ping Pong Delay Process Block

PluginEditor Source Code Design

This section highlights the user interface and values obtained from the user to implement within the PluginProcessor code. All of this work utilises the pre created classes within JUCE. This is to allow focus on the signal processing design.

The user’s inputs are passed into a value tree state to be accessed within the PluginProcessor. In comparison to the slider listener class, this has been observed as a more efficient and less latent way of obtaining values. Figure 2.14 is the creation of the tree state values for the delay. Figure 2.15 is the accessing of the values in ‘PluginProcessor.cpp’.

```
private:
    juce::Slider balanceSlider;
    juce::Slider delaySlider;
    juce::Slider feedbackSlider;
    juce::Slider mixSlider;

    std::unique_ptr<juce::AudioProcessorValueTreeState::SliderAttachment> balanceSliderAttachment;
    std::unique_ptr<juce::AudioProcessorValueTreeState::SliderAttachment> delaySliderAttachment;
    std::unique_ptr<juce::AudioProcessorValueTreeState::SliderAttachment> feedbackSliderAttachment;
    std::unique_ptr<juce::AudioProcessorValueTreeState::SliderAttachment> mixSliderAttachment;
```

Figure 2.14 The Creation of Tree State Values in PluginEditor.h

```
juce::AudioProcessorValueTreeState::ParameterLayout PingPongPass1AudioProcessor::createParameters() {
    std::vector < std::unique_ptr < juce::RangedAudioParameter>> params;

    params.push_back(std::make_unique<juce::AudioParameterFloat>("BALANCE", "Balance", 0.0f, 0.9f, 0.5f));
    params.push_back(std::make_unique<juce::AudioParameterFloat>("DELAY", "Delay", 0.0f, 3.0f, 0.1f));
    params.push_back(std::make_unique<juce::AudioParameterFloat>("FEEDBACK", "Feedback", 0.0f, 0.9f, 0.7f));
    params.push_back(std::make_unique<juce::AudioParameterFloat>("MIX", "Mix", 0.0f, 1.0f, 1.0f));

    return { params.begin(), params.end() };

}
```

Figure 2.15 Accessing Tree State Values in PluginProcessor.cpp

The source files implemented creates a graphic user interface (figure 2.16) with functional digital signal processing as a VST3 plugin. This can further be improved upon and edited as required.

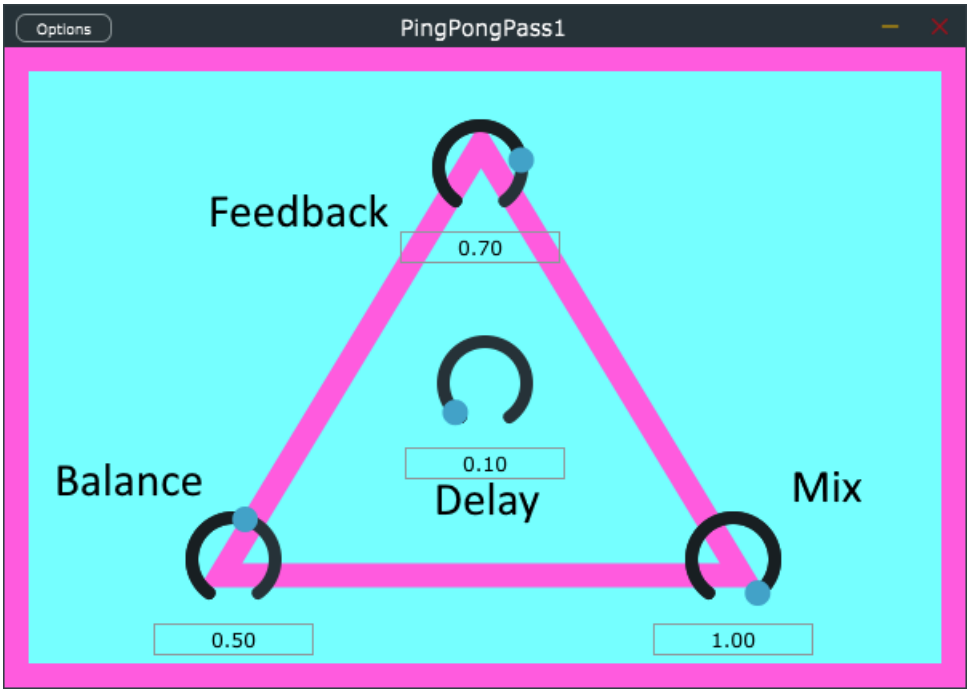

Figure 2.16 The Ping Pong Delay User Interface

3. Results & Discussion

The result of the signal processing techniques described above is the implantation of a novel VST3 plugin that distorts the delay and filters high and low frequencies. The result is a minimal distortion and ping pong delay that includes a stereo mix of sound. Previous MATLAB prototype testing highlights the characteristics of the delay on a sine wave. We can therefore conclude that the ping pong delay is working as expected.

An auditory assessment concludes that the delay is panned between the left and right channels effectively. The distortion provides minimal change, although is audible on a guitar sample. This can be heard when using the VST3 plugin format through a digital audio workstation. The filters and distortion are tested by changing parameters and listening for the result. Extremes in cut off frequencies and hard clip thresholds highlight the implementation as functional.

With both qualitative and quantitative data suggesting these outcomes, it is supported that the delay implemented works as expected. The feedback loop is implemented within the C++ code that is missing from the MATLAB prototype successfully. Filters and distortion are working as expected also.

Future Work

This novel VST3 plugin implores the focus of the delay aspect as the focus. This is due to the key sound change that is heard and other signal processing effects being added upon the created delay. The implemented delay solely uses stereo tracks. The use of 3D audio is a potential area for future improvement, without the limit on the number of channels.

A downside to this focus is that the distortion is very minimal. On the guitar tracks tested, it can be heard well when listening to the delayed signal only. Future work may look to further integrate both the distortion and delay phases. Such as a switch to affect both the wet and dry signal, as well as a gain value for this mix.

This modular design would also work with the inclusion of a three-band equaliser. The use of shelving and peaking filters to shape the frequency response of the outputted signal would allow for further control and design. There is the problem of overcomplication with this, however.

Conclusion

To conclude, this technical report highlights the theories and signal processing techniques utilised for the creation of a delay plugin. This aligns with the given brief for this assignment and further signal processing is completed via the distortion and filtering blocks. The final design is theorised first as a MATLAB prototype before moving into the JUCE framework to create the VST3 plugin within C++.

The resulting plugin, gives control of a ping pong delay, bouncing feedback to opposing channels. It is limited to stereo input, which is a building block for future work. The signal

processing occurs in real time in circular buffers. The usage of the JUCE buffers allow for hot reloads and immediate responses.

It is concluded via testing and auditory conclusions that the plugin works as expected, with filtering allowing for further control to the frequency response.

4. Reference List

Gibson, D. and Polfreman, R. (2011). An Architecture for Creating Hosting Plug - Ins for Use in Digitial Audio Workstations. International Computer Music Conference. Huddersfield.

Pirkle, W.C. (2019). *Designing Audio Effect Plugins in C++ : for AAX, AU, and VST3 with DSP Theory*. New York: Routledge, pp.15–30, 535–563.

Reiss, J.D. and McPherson, A.P. (2015). *Audio Effects theory, Implementation and Application*. London: Taylor & Francis Group.

Smith, S.W. (1997). *The Scientist and Engineer's Guide to Digital Signal Processing*. San Diego, Calif.: California Technical Pub, pp.503–534.

Tan, L. (2008). *Digital Signal Processing : Fundamentals and Applications*. Amsterdam: Academic Press.

Tarr, E. (2019). *Hack Audio: an Introduction to Computer Programming and Digital Signal Processing in MATLAB*. London: Routledge.

Zolzer, U. (2011). *DAFX : Digital Audio Effects*. Chichester, West Sussex, England: Wiley.

Appendix 1. A Basic Delay MATLAB Code

```
%% A basic delay on a sine wave

%% Creation of the sine wave
Fs = 48000;
Ts = 1/Fs;
f = 2;
t = [0:Ts:1].';
in = sin(2*pi*f*t);

%% Delay
N = .25;
delayt = t-N;
delay = sin(2*pi*f*delayt);

%% Plotting Results
figure(1);
plot(t,in, t, delay, '--', 'LineWidth',5); axis([0 1 -1.1 1.1]);
set(gca,'FontSize',20);
xlabel('Time (sec)', 'FontSize', 24); ylabel('Amplitude', 'FontSize', 24);
title('Basic Delay', 'FontSize', 24); legend('Signal', 'Delayed Signal',
'FontSize', 12);
```

Appendix 2. A Feedback Delay MATLAB Code

Main Script:

```
clear all; clc;
% Input Signal
in = [1;-1;2;-1;zeros(6,1)];

buffer = zeros(20,1); % Long Delay

delay = 5; % Number of Samples to Delay by

% Feedback Gain
fbGain = 0.5;

% Output Vector
N = length(in);
out = zeros(N,1);


for n = 1:N
    [out(n,1), buffer] = feedbackDelay(in(n,1),buffer,delay,fbGain);
end

t = [1:1:N];
plot(t,in,t,out,'--', 'LineWidth',7);set(gca,'FontSize',28);
xlabel('Time (sec)', 'FontSize', 30); ylabel('Amplitude', 'FontSize', 30);
title('Feedback Delay', 'FontSize', 30); legend('Signal', 'Delayed Signal',
'FontSize', 20);
```

FeedbackDelay Function:

```
function [out,buffer] = feedbackDelay(in,buffer,delay,fbGain)
%FEEDBACKDELAY Summary of this function goes here
%   Detailed explanation goes here

out = in+fbGain*buffer(delay,1);
buffer = [out;buffer(1:end-1,1)];
end
```

Appendix 3. Ping Pong Delay MATLAB Prototype

Main:

```
%% The DSP design for a JUCE Ping Pong Delay
%Changable variables:
%   fbGainL -> gain amount for left channel
%   fbGainR -> gain amount for right channel
%   delay -> number of samples to delay by
%   wetGainL -> Left Wet Mix
%   wetGainR -> Right Wet Mix

clear;clc;
%% Creating a multi channel sine wave
% Allows for prototyping with clear results
Fs = 48000;
Ts = 1/Fs;
fl = 2; % Left channel
fr = 5; % Right channel
t = [0:Ts:1].';
in = [sin(2*pi*fl*t),sin(2*pi*fr*t)]; % Creation of a sine wave channel array

%% Plotting the Original Signal
% For comparsion
figure(1)
hold on
plot(t,in,'LineWidth',5)
set(gca,'FontSize',28);
xlabel('Time (sec)', 'FontSize', 30); ylabel('Amplitude', 'FontSize', 30);
title('Mutli Channel Sine Waves', 'FontSize', 30); legend('Left Signal', 'Right Signal', 'FontSize', 20);
hold off

%% Ping Pong Delay
% Build around a feedback delay
N = length(in);
data = zeros(N,1);
%buffer = [zeros(N,1)';zeros(N,1)'];
buffer = [data, data];
delay = 9; % Number of Samples to Delay by

% Feedback Gains
fbGainL = 0.5;
fbGainR = 0.7;

% Wet Gains
wetGainL = 0.9;
wetGainR = 0.4;

% Output Vector
out = [data, data];

for n = 1:N
    [out(n,1), buffer] = PingPongDelay(in(n,1),buffer,delay,fbGainL);
    [out(n,2), buffer] = PingPongDelay(in(n,2),buffer,delay,fbGainR);

end

for n = 1:N

    out(n,1) = wetGainL.*out(n,1)+in(n,1);
    out(n,2) = wetGainR.*out(n,2)+in(n,2);

end

%% Plotting the Delayed Signal
% For comparsion
figure(2)
hold on
plot(t,in,t,out,'--', 'LineWidth',7)
set(gca,'FontSize',28);
xlabel('Time (sec)', 'FontSize', 30); ylabel('Amplitude', 'FontSize', 30);
title('Ping Pong Delay', 'FontSize', 30); legend('Left Signal', 'Right Signal','Left Delayed Signal','Right
Delayed Signal' ,'FontSize', 20);
hold off
```

PingPongDelay Function:

```
function [out,buffer] = PingPongDelay(in,buffer,delay,fbGain)
%PINGPONGDELAY Feedback Delay to Opposite Stereo Channels
%   A copy of the feedback delay with feedback to other channels

out = in+fbGain*buffer(delay,1);
buffer = [out;buffer(1:end-1,1)];
end
```