Assignment 2 - Original Audio Effect

Luke Brosnahan ec19129@qmul.ac.uk
190155597

1. Introduction

The purpose of this assignment was to implement an original audio effect using either the Matlab Audio Toolbox or Juce/C++. The 1970s saw the introduction of the first commercially available and affordable delay pedals with Boss and Roland (it's parent company) producing a range of delay effect models. Ever since, delay effects have been a mainstay across a multitude musical genres. Including a delayed copy of a signal can breath new life into a stale sound while longer delays can create fascinating syncopated rhythms out of a simple pattern or allow a solo instrument to duet with itself. Delay lines are also a key foundation of many of the most widely used digital audio effects used today. Delay effects are often implemented with just a single delay line. However, limiting the user to a single delay may close off creative possibilities and ideas. It may seem trivial to add another copy of the same effect in order to delay the same signal in multiple ways but doing so requires careful selection of parameters of both delays in order to avoid a cluttered and messy sound. This is made more difficult when the user must switch between interfaces in order to make small adjustments that can make the difference between two delayed signals sitting nicely together or clashing massively. The audio effect described in this paper is an attempt to provide a delay effect which allows multiple delays of the same signal with specific controls for each delay on the same interface.

As well as all the usual benefits of a single delay, some potential uses for this delay would be to create complex polyrhythms from very simple section of MIDI drums, apply delays to multiple instruments from a single audio recording separately and delay different frequency bands within a single sound individually.

2. Implementation

2.1 System Overview

In order to begin the implementation, a full list of system requirements was first created for guidance. The requirements for the system were as follows:

- 3 separate delays of the same signal with the following adjustable parameters:
  - Delay time, in terms of milliseconds or sixteenth notes
  - Feedback
  - Filter Type e.g. No-Filter/Low-Pass/High-Pass
  - Filter Cutoff, in Hz
  - Ping Pong Delay e.g On/Off
  - Gain, in dB
- Dry/Wet knob for specifying mix of the Dry and Wet signals
- Power switches for each delay to activate/deactivate delay

For this implementation, the audio effect was created in Juce using C++, with Ableton Live 10 and Microsoft Visual Studio used for debugging.

2.2 Method

In order to apply multiple delays to the signal, several copies of the signal would first need to be made and stored separately. One copy is made and stored as the Dry signal, undergoing no further processing, while three more copies are made so that distinct delays may be applied to each. For each copy of the signal, audio samples are stored in a preallocated memory buffer via a write operation while previously stored samples are extracted via a read operation. The relative distance of the read position in relation to the write position is dependent on the length of the delay time. When either the read or write pointer overrun the memory buffer they loop back to the beginning creating a circular buffer. In order to incorporate feedback a slight variation is made to this method as the write operation stores both the input signal and the delayed signal, scaled by the feedback parameter. In the case that the delay is a Ping Pong delay (figure 1), two memory buffers must be used to store the left and right channel signals separately. The output of the delay line for each channel is then fed back into the input of the opposite channel's buffer, creating an effect that the delayed signal is bouncing from left to right. This effect will of course only be perceived in a stereo track.

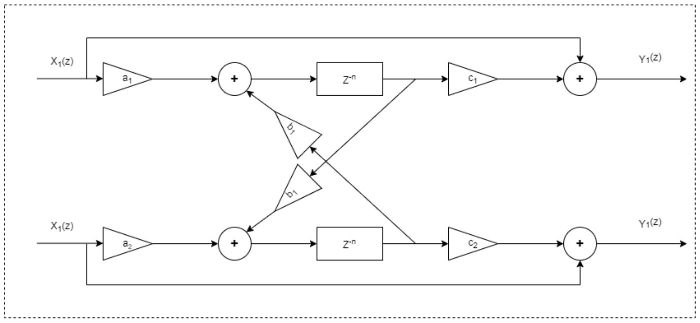


Figure 1: Diagram of a Ping Pong delay

Once the delay has been applied to the signal, further control over the delay is provided in the form of an optional filter with adjustable type and cutoff frequency. 4th-order IIR low-pass and high pass filters are used to filter the output of each delay allowing the user to attenuate frequencies above or below the filter cutoff of the delay's output. As long delays ring out, it may be useful to remove frequencies so as not to clutter certain frequency ranges. Finally, gain is applied to each delay output individually before the signals are added into the output buffer along with the dry signal, weighted by the Dry/Wet parameter. Figure 2 presents a diagram of the signal flow chain and served as a reference point throughout the design process.

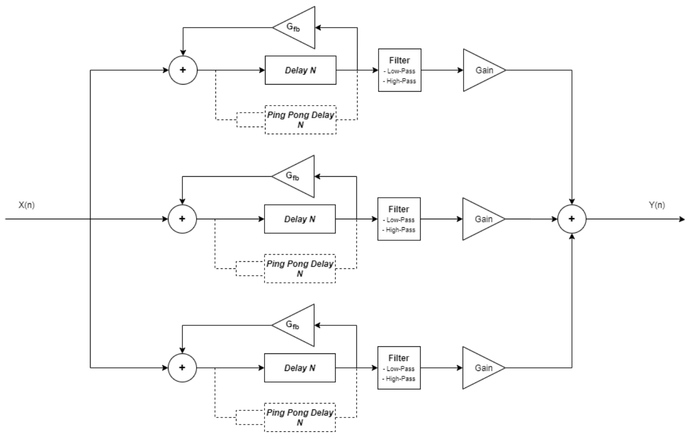


Figure 2: Diagram of the Multi-Delay unit

2.3 Classification

This effect can be classified as a standard linear time invariant filter, with magnitude and phase response fully describing the effect. We can confirm the stability of the effect by starting with the time-domain transfer function of a single delay with feedback.

$$y[n] = x[n] + G_{FF}d[n], \text{ where } d[n] = x[n-N] + G_{FB}d[n-N] \tag{1}$$

d[n] is similar to the delayed output signal y[n-N], allowing substitution to write y[n] in terms of x[n]:

$$y[n-N] = x[n-N] + g_{FF}d[n-N] \tag{2}$$

$$d[n] = \frac{g_{FB}}{g_{FF}}y[n-N] + (1 - \frac{g_{FB}}{g_{FF}})x[n-N] \tag{3}$$

$$y[n] = g_{FB}y[n-N] + x[n] + (g_{FF} - g_{FB})x[n-N] \tag{4}$$

We can then describe the frequency domain transfer function by applying the Z-transform:

$$Y(z) - g_{FB}z^{-N}Y(z) = X(Z) + (g_{FF} - g_{FF})z^{-N}X(z) \tag{5}$$

$$H(z) = \frac{Y(z)}{X(z)} = \frac{1 + z^{-n}(g_{FF} - g_{FB})}{1 - z^{-n}g_{FB}} = \frac{z^N + g_{FF} - g_{FB}}{z^N - g_{FB}} \tag{6}$$

Inspecting this transfer function, we see that the system has poles at complex roots of $\mathrm{g_{FB}}$ and that the effect will be stable when the pole lie inside the unit circle i.e. when $\mid g_{\mathrm{FB}} \mid <1$. Given that the full effect is an addition of multiple of these delay lines together along with possible filtering using stable filters, the same principle still applies.

The system can also be classed memory dependent as the current input depends on previous inputs to the system and causal as it does not depend on future inputs.

2.4 Code

The audio effect was implemented in code using two different classes.

- PluginProcessor.cpp
- PluginEditor.cpp

2.4.1 PluginProcessor.cpp

The PluginProcessor class contains any of the necessary setup for the audio effect plugin and, in effect, implements Figure 2. Apart from the usual audio plug in setup, the constructor for plugin processor importantly defines the value tree state which is used to store the parameters of the plugin. The value tree state plays a vital role in the plugins operation as it allows the plugin parameter's values to be stored when the GUI is closed. This also allows the plugin parameters to be automated using automation controls in whatever DAW the plugin is being hosted in. The *updateParameters()* method is used to update the parameters of the plugin by retrieving the values from tree state and store them in variables so that they may be used by the processor to process the input signal.

The *prepareToPlay()* method is called when the effect is first loaded and does some important initialisation of variables. The number of input channels is stored for later use while the sample rate is used to define a size for the delay buffers along with the maximum delay time allowed, in seconds(5). Next, an initial set of parameters are set by calling the *updateParameters()* method. These parameters will simply be the default values as defined in the parameter constructors in value tree.

```cpp
void MultidelayAudioProcessor::prepareToPlay (double sampleRate, int
    samplesPerBlock)
{
    const int numInputChannels = getTotalNumInputChannels();
    const int delayBufferSize = 5 * (sampleRate + samplesPerBlock);
    mSampleRate = sampleRate;

    updateParameters();

    //Initialize delay buffers, clear them and reset write positions to 0
    for (int i = 0; i < NUM_DELAYS; i++) {
        mDelayBuffers[i].setSize(numInputChannels, delayBufferSize);
        mDelayBuffers[i].clear();
        mWritePositions[i] = 0;
    }
}
```

The *processBlock()* method contains the guts of the plugin's processing and begins by defining some key variables to be used later in the method, such as the total number of input and output channels and the lengths of the input and delay buffers. Next, memory buffers for the dry output and the three delay outputs are initialised and cleared. A check is made to see if the parameters need to be updated based on whether a boolean, *shouldUpdateParameters*, has been set true by the value tree listener method, *valueTreePropertyChanged*. This check ensures that efficiency is not sacrificed by updating the parameters every time the process block runs even when the user has not changed anything on the GUI. The last step before the delay processing occurs is to clear the main buffer.

A conditional statement checks whether the delay should be processed as a regular delay or as a ping pong delay based on one of the delay's parameters. The code implements the process described in the Method section above but it is worth mentioning that it makes use of a fractional delay using linear interpolation. The current output sample for the delay line is calculated based on a weighted fraction of the two samples that the localReadPosition reads between. This method helps to clicking in the output of the signal. Although this is still an approximation, it is quick to calculate and produces much better results than the nearest neighbour approximation. The code snippet below displays the implementation which was also adapted for both channels in the ping pong delay processing condition. Finally the write position for the delay is updated with the local write position of the sample that it finished on.

```cpp
 //Calculate output sample as by interpolating between the two closest sample
    positions, weighted by proximity
    if (localReadPosition != localWritePosition[i]) {
        float fraction = readPosition - (float)localReadPosition;
        float delayed1 = delayBufferData[(localReadPosition + 0)];
        float delayed2 = delayBufferData[(localReadPosition + 1) %
    delayBufferLength];
        out = delayed1 + fraction * (delayed2 - delayed1);

        //Write the sample to the delays output buffer and back into the delay
     buffer with feedback
        outputData[sample] = out;
        delayBufferData[localWritePosition[i]] = in + out * feedbackValues[i];
                    }
```

The last action of the process block is to filter the delay outputs, apply individual gain to each and then add them to the main buffer along with the dry buffer. As mentioned previously, the mix of Dry to Wet signal is also applied as the outputs are added.

2.4.2 PluginEditor.cpp

The PluginEditor class implements the GUI which is used to tune the parameters of the multi-delay effect. This code sets the size of the GUI and paints the main shapes which make up the base of the interface. It also sets up the positions of the sliders, buttons and drop down menus. Importantly, this class also attaches the sliders and buttons to the value tree parameters so that they will be update as the user interacts with them. The final GUI design can be seen in Figure 3 below.

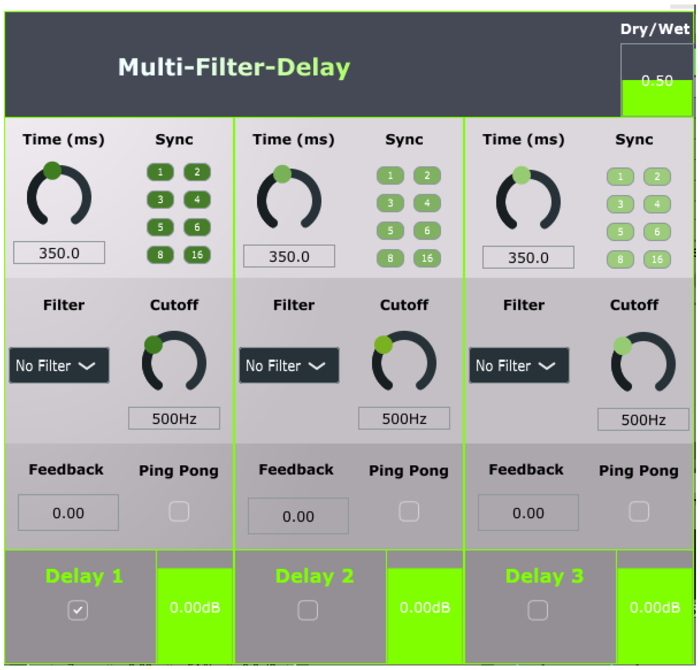


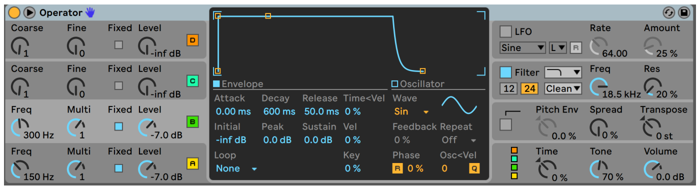
Figure 3: Final design of GUI

3. Evaluation

In order to analyse the operation of the plugin on different signals, the effect was applied to a number of test signals, exported and then investigated using Sonic Visualiser. The experiments were designed to verify the operation of the elements of the signal chain diagram in Figure 2.

3.1 Filter Delay Experiment

The first experiment looked to investigate whether a sound could be separated using the filters and delayed at different times. Ableton's operator was used to generate a tone comprised of two sine wave oscillators, one at 150Hz and the other at 300Hz.



Coarse 1 | Fine 0 | Level -inf dB
Coarse 1 | Fine 0 | Level -inf dB
Freq 300 Hz | Multi 1 | Fixed | Level -7.0 dB
Freq 150 Hz | Multi 1 | Fixed | Level -7.0 dB

Attack 0.00 ms | Decay 600 ms | Release 50.0 ms | Time<Vel 0 %
Initial -inf dB | Peak 0.0 dB | Sustain 0.0 dB | Vel 0 %
Loop None | Key 0 %
Wave Sin | Feedback 0 % | Repeat Off | Phase 0 % | Osc<Vel 0

LFO Sine | Rate 64.00 | Amount 25 %
Filter 24 Clean | Freq 18.5 kHz | Res 20 %
Pitch Env 0.0 % | Spread 0 % | Transpose 0 st
Time 0 % | Tone 70 % | Volume 0.0 dB

Figure 4: Operator setup

The plugin was then setup with two separate delays, one with a delay time of 283ms and the other with a delay time of 424ms. The first delay applied a low pass filter to it's output with a cutoff frequency of 270Hz while the second delay applied a high pass filter at the same cutoff frequency. The plugin should therefore separate the two sine waves which make up up the tone and delay the the high frequency wave directly after the low frequency wave. The Dry/Wet parameter was left at 50% so that both the dry and wet signal were included in the resulting audio. Figure 5 below displays the spectrogram of the resulting audio. As we can see the hypotheses is proven correct and plugin operates as expected.

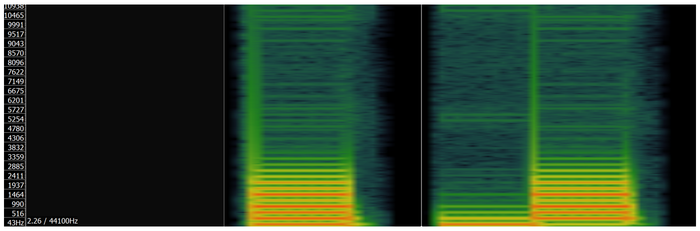


Figure 5: Spectrogram of the result

3.2 Ping Pong Delay Experiment

Another experiment was carried out to test whether the ping pong delay was operating as it should. Ableton's operator was once again used to generate a tone, however, this time it was the sine wave oscillators were at 150Hz and 1000Hz respectively. A single delay was used and delay time was set to 283ms. The delay applied a high pass filter with a cutoff

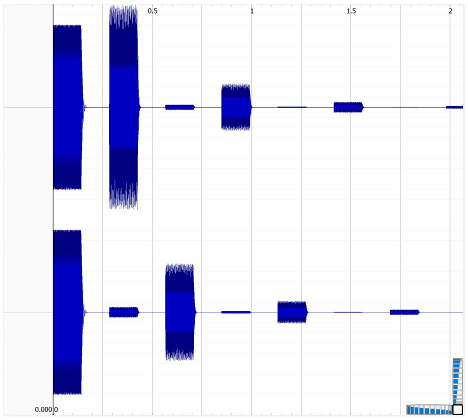


Figure 6: Waveforms for both channels.

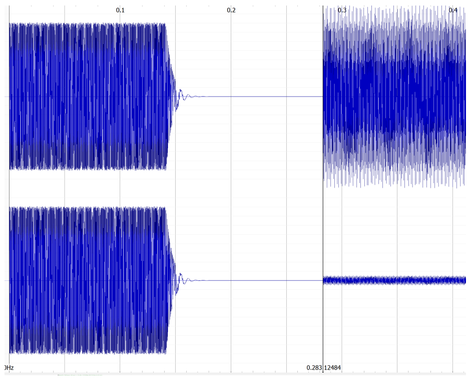


Figure 7: Timestamp of first delay.

frequency of 964Hz in an attempt to filter out the lower oscillator and ping pong delay the higher oscillator. The feedback was set to 0.5 to allow the delay to ring out longer and bounce from channel to channel.

Figure 6 displays the waveforms for both channels of the resulting signal. After the initial combination tone plays, we can see the high filtered delayed signal bouncing from channel to channel as time progresses. We can also verify the delay time is correct by inspecting the time stamp of the first waveform, seen in 7.

4. Conclusion

The described implementation can be considered a successful execution of a useful digital audio effect. The plugin allows the user to delay a single input signal in multiple ways concurrently and offers control over the delays to avoid messy results. The above implementation is also scalable to add many more delays, computational power allowing, with relatively little work in terms of code. The experiments and results displayed above demonstrate the plugin's successful operation and potential uses. One outstanding improvement that could be made is in the method of interpolation of the delay samples. Although linear interpolation does provide better results than nearest neighbour, a crackling sound resulting from discontinuities occurs when the delay time is altered during use. This is undoubtedly a potential block to creative use of the effect. Employing some form of polynomial interpolation is an obvious future addition to the plugin. Another potential area for improvement is increasing control further. For example, the Ping Pong delay balance setting could be user controlled. A default balance setting of 0.95 was set for each delay so as not to clutter the GUI with too many user parameters but with more time to plan the user interface parameters such as this could be user controlled.