Audio Processing Software

Tommy Burholt - 2023

Table of Contents

Project Overview

For beginners getting started with audio processing, there are several solutions they could use, varying from specialist software to specific plug-ins. However, across every solution the quality of the included effects may vary causing the user to pay for an upgraded version of the software or fork out a lot of money for better software. I plan to create a simple yet versatile, open-source effects bundle which works on every piece of software that supports VST2. It will contain 4 distinct effects, allowing the user to grasp the basic concepts of audio processing without the conventional paywall. The 4 effects I will create are Delay, Distortion, Flanger and Compression. These cover some of the essential effect beginners should know how to use. The main effects I am missing are Phaser, Chorus, Reverb and a parametric EQ however due to their complexity, it is outside the scope of this project. However, in the future I will add these to the bundle. Alongside my plug-ins, I will create a website using Github pages to allow users to download each plug-in and find documentation on how each effect works. Seeing as the project will be open source, there will be documentation on how essential classes work for each effect allowing other programmers to improve and reuse my code.

Personally, I have been producing music for over 4 years and I have always wanted to create my own audio processing software and plug-ins. This project allows me to understand the basic concepts of Digital Signal Processing (DSP) and the backend of audio processing, furthering my knowledge in the field and allowing me to implement more of my ideas in the future. It will also serve as a guide for anyone else who is interested in getting a taste for programming audio processing software.

After some consideration, I decided to name the project Chroma Audio since after I finish this, I plan to create more melodic and colourful plug-ins. Finally, before I start here is a link to the Github repository and the website. Throughout the project I will refer to audio examples which can be found in their respective folder in the repository.

Github: https://github.com/Psonite/Audio-Processing-Software

Website: https://psonite.github.io/Audio-Processing-Software

Computational Methods

Once I outlined the problem and assessed my objectives and I decomposed the problem into different sections. Since I am creating 4 different plug-ins, I will take a divide and conquer approach to the project. Each plug-in will have its own dedicated section which will encompass Design, Implementation, Development and iterative testing. This allows me to keep everything relevant to each plug-in in one section and allows me to work on multiple plug-ins at the same time since I can easily move between plug-ins. Since each section will have 4 main focuses, I'll explain the purpose of them here.

- Design – This aims to outline what the effect does, provide a brief history on the effect and outline any key ideas / signal diagrams which may be useful for the reader.
- Implementation – This aims to outline how I am going to implement the effect and explain all the different modules I am going to use with both visual and audio examples.
- Development – This will show how I developed the plug-in, from a brand-new project to the finished product. It will also show any problems I have encountered and how I approached fixing them.
- Iterative testing – Although there will be a dedicated section for testing later in this document, this focus will be embedded within the Development section, and it will show how I tested the software during development.

Since audio-based programming is quite a niche topic, I will have a section explaining the basics of Audio-Based DSP which should allow anyone with a basic understanding of programming to follow along. After this I will have a section for how the framework, I'm using works and a section on how I created the GUI for each plug-in.

When deciding how I should create my solution, I had two options. Either create the solution digitally or with electronics. After assessing each approach, the benefits of a digital based solution outweighed their physical counterpart. A digital based solution lends itself to; High-Speed, High-Quality processing alongside flexibility and expandability. Alongside this, it makes my solution more accessible since anyone with an internet connection and the required software can access and contribute to it. Although a physical based solution would practically eliminate latency and remove the need for a computer, making the solution more viable for live performances. This approach contradicts the whole point of my project being easily accessible and free to use. In addition to this, I would need a specialist knowledge in electronics and metal working which I don't have.

Creating my solution digitally also gives me the option to expand it to a physical system in the future, this would allow me to create custom software for a Rasbery PI which could run my plug-ins when connected to the appropriate hardware. This led me to discover 'Norns Sheild' by Monome [1] which takes this concept of Rasbery PI based music hardware to another level. In the future I plan to create something similar because it would allow me to learn how to program low-level code and work directly with the hardware.

Researching the problem

From my own research and experience, there are other solutions such as the MFreeFXBundle by Melda Production [2] and the Kilohearts Essentials Bundle [3]. Both offer a wide range of high-quality and versatile effects. However, upon discussion with different stakeholders, the number of effects and complex UI could make it harder for complete beginners to use. Since both solutions are free, I decided to research a paid solution. I came across the FabFilter Total Bundle [4] which offers a selection of High-Quality plug-ins which are renowned for their quality and stability. Although I would like to compare my plugins to their FabFilter counterparts, I don't have the money to buy all their plug-ins since it comes to almost £800. Alongside third-party effects bundles, I will compare my plug-ins to Ableton's stock effects [5]. They will act as a benchmark, since Ableton is used widely in music production and their stock effects are highly praised. If you don't want to go with a bundle, you could search the internet to find each plug-in individually, however this is time consuming, and the quality of each plug-in will vary between developers.

When researching how to create audio processing software I evaluated the top 2 frameworks for developing audio plug-ins; JUCE [6] and iPlug2 [7]. Both are written in C++ owing to its object orientated approach, memory management and low-level manipulation, which is essential when writing software that requires low latency during use. Here is a short overview for each:

- JUCE is the industry standard for developing audio processing software owing to its wide range of features and large amount of community support. It supports both desktop and mobile operating systems without having to drastically alter any code. It supports all modern plug-in formats such as VST3 and AUv3 and every modern Audio format. However, depending on which plan you chose it could cost a lot of money. But it does have a free tier for beginner developers to get started with at the draw back of a watermark when opening the plug-in.

- IPlug2 is a lesser-known framework developed by Oli Larkin. When compared to JUCE it has less features and support, however it has less clutter which makes it more straightforward to use. Similarly to JUCE, it supports all modern plug-in and audio formats and has support for different operating systems. IPlug2 is completely free to use and doesn't require any purchases to access all its features.

After this, I chose to use iPlug2. Although it doesn't have as much support as JUCE, it is completely free, and the implementation of each effect remains the same. There are two main versions that I found, WDL-OL and iPlug2. WDL-OL was the predecessor to iPlug2 and is what most of the resources I found implementing plug-ins use. However, since the concepts are the same there will only be slight differences between the two, I will use iPlug2 owing to its larger feature set and amount of support.

Limitations

Since I don't have an infinite amount of time to make everything, there will be a set of limitations to the project. However, these can be addressed when the project is complete.

Although I already know how all the effects work from a user standpoint, I still need to research how everything works under the hood. This will take up a significant portion of my time however, this should reduce overall development since I'll know how to program everything. This does pose the limitation of how much research on different plug-ins I can do before I need to start programming the software. As stated earlier it would be better to create a full bundle of effects however, from the plugins I have chosen, this should serve as a starting point to develop the bundle in the future.

Another limitation will be the framework I'm using, since I am only using iPlug2 I could encounter a flaw in the framework's code later in the project preventing me from working on a plugin. Although it is unlikely that this will happen, it is something to consider.

Since I am running Windows on both my computers, I won't be able to support every system natively. For users running MacOS, I won't be able to guarantee stability and support since I don't own an Apple device on which I can test my software. Similarly, to MacOS, I won't be able to guarantee stability and support for Linux at the moment. However, I plan to install a virtual machine to emulate Linux so I can test my plug-ins in programs such as Reaper [8] and Bitwig Studio [9].

This leads on to another point about support. At the time of writing, I have only been able to successfully build for VST2 and EXE. However, this is mostly due to me not having the required SDK's for VST3 and AAX. I plan to address this in the future however, it isn't important at the moment owing to VST2 being the predominant plug-in format which is used by a wide range of software.

Finally, since I will only have a limited time for Beta testing and a limited number of testers, I won't be able to find every bug and fix everything before the end of this project. However, this limitation shouldn't impact the project hugely since I should be able to find any large bugs affecting major functionality.

Overall, these limitations shouldn't impact the project moving forward and I should be able to address them in the future.

Requirements

Depending on who is using my solution there will be different sets of requirements. For users, any piece of software that can run VST2 and run the plugins natively will work and I will also include an executable which can be run independently since the iPlug2 has the option to build as an EXE. Although this does prevent users of MacOS from running a standalone version of the software since I am building on a Windows machine. No external hardware is required to run any of the plug-ins however, all the plug-ins will support MIDI, allowing them to be controlled with an external controller. Owing to the nature of VST2, there isn't a required operating system to run the plug-ins in this format since they are run within their respective software. As far as I'm aware iPlug2 doesn't support the new M1 and M2 MacBook since they use an ARM architecture, and I don't own one so I can't test if my software works on them. As mentioned earlier the standalone app will only run on Windows however I plan to include MacOS support in the future.

For developers, the requirements are slightly different. In addition to what was mentioned earlier, they will also need iPlug2 installed on their machine. It isn't required for users since they only need the VST2 or standalone application to run the software. The developer will also need either Visual Studio or Xcode to open and edit the project since that is what iPlug2 natively supports. Anything else of note will be documented on my website.

Essential Features

After I decomposed the problem, I created a set of essential features which all plug-ins should have. They aim to cover all the needs and requirements from each group of stakeholders and improve useability for beginners.

- All plug-ins should have the ability to work as both standalone software and in VST format. This allows any user, whether they have the required software or not to use my solution.
- All plug-ins should process audio at the highest-quality possible without sacrificing on performance. Since my plug-ins are targeted at both beginners and professionals it is essential not to reduce the quality of the incoming signal.
- Since the plug-ins are aimed at beginners, all the controls should have responsive and meaningful controls which improves first time user experience.
- Since any plug-in could be saved within a project and reopened later, they must be able to save their state, so any adjustments made aren't lost. This should be handled by the software itself, however I will allow users to save and load presets if this isn't the case.
- Each plug-in should be completely stable to prevent crashes and loss of data.

Stakeholders

After researching the problem, I identified who would take interest in my solution. Below is a list of stakeholders and their reasons for interest in my solution.

- Music Producers – Modern music production relies heavily digital effects and wouldn't be where it is without them. Being a music producer myself I know how much high-quality and affordable effects are needed. My package will include the essential effects, alongside experimental features which will allow both novice and experienced producers to create unique and interesting sounds.

- Mix Engineers – Every modern song has been mixed in one way shape or form. It is the process of conveying an idea the producer has created in an audibly pleasing fashion. Their job can range from fixing errors in the recording to creating a lush soundscape with digital effects. They rely heavily on basic effects because they are supposed to help convey the idea rather than edit the existing one.

- Sound Designers – Sound design is a wide and interesting field with uses in both Music and Film. Either way they heavily use digital effects when creating interesting and unique sounds. The effects used can range from simple guitar pedal emulations to crazy spectral processing, presenting a need for high-quality effects.

- Content Creators – During content creation, the use of audio effects is commonplace. From fixing audio to normalising the level on a voice-over, simple yet effective effects are essential to any beginner looking to get into the career. Effects such as compression are commonplace making the idea of an open-source effects bundle more appealing.

I interviewed the different stakeholders and asked for feedback on my ideas for each plug-in. From this I will add and remove features from my plug-ins in accordance with their feedback.

- Delay – There wasn't much feedback apart from the addition of a Tone and Warmth control to emulate an analogue tape delay. [Suggested by ME_2]
- Distortion – Overall, they were satisfied with the features and proposed two features I should include, the addition of a Bias and Rectify control. [Suggested by SD_1]
- Flanger - The addition of a Tone and Warmth control was suggested alongside the ability to control the feedback from the flanger. [Suggested by ME_1]
- Compressor - Across all stakeholders a sidechain input was heavily requested and the addition of a Dry/Wet control for parallel compression. [Suggested by MP_2]
- Synthesiser - The synthesiser should have full MIDI support which allows the user to use external hardware to control it. There was a suggestion to implement Karplus-Strong synthesis into the plug-in however after some basic research I believe it should be implemented as its own plug-in after I finish this project. [Suggested by SD_2]

Success Criteria

For my success criteria I have created a table outlining each and how I can achieve them. Although my project is split into many sub-projects, I will evaluate the overall success of each criterion between projects to see if I was successful. I have given each criteria an ID allowing for later reference.

Success ID	Success Criteria	Appropriate fulfilment of criteria
SC0	Optimised performance: The plug-in must be easy to run during intended use. This allows the user to use multiple instances of the plug-in without a drastic increase in load on the CPU.	• The impact on the CPU should not exceed 2x the load when compared to Ableton's stock effects. [SC0_A] • Each plug-in should not take over 1 second to open. [SC0_B] • Each plug-in should be responsive to user input. Each control should not take more than 100ms to respond to an input. [SC0_C]
SC1	Ease of use: Since the software is aimed at beginners, ease of use is essential. This encompasses both the plug-in itself and its documentation.	• Every essential control must not be hidden inside of menus. [SC1_A] • Each control must be labelled clearly. [SC1_B] • The w9ebsite must have user focused documentation for operation of each plug-in. [SC1_C] • Every plug-in should be accessible for the colourblind. There should be sufficient contrast between controls allowing the colourblind to use the plug-in. [SC1_D]
SC2	Compatibility between software and operating systems: Since each stakeholder's use of the solution spans across different software and operating systems it must work across all of them.	• Every plug-in must work on different operating systems. [SC2_A] • Every plug-in must work on different DAWs. [SC2_B] • Every plug-in must work in both audio and video editing software. [SC2_C]
SC3	Accurate emulation of desired effect: It isn't a successful solution unless each plug-in achieves its goal.	• Each control must have the desired effect on the signal. [SC3_A] • Each plug-in should be comparable to similar solutions. [SC3_B]
SC4	Reduced file size for each plug-in: Plug-ins should not take up an excessive amount of space.	• Each plug-in should not exceed 20mb in size. [SC4_A] • The entire package should not exceed 100mb in size. [SC4_B]

SC5	Retention of audio quality: Since my plug-ins could be used by professionals, there should be no deterioration in quality throughout the processing of the audio.	• There should be no unintentional compression while processing the audio. [SC5_A] • There shouldn't be any artifacts or ghost signals after processing. [SC5_B] • Each plug-in must adapt to changes in sample rate. Where applicable there should be no errors when the sample rate is changed. [SC5_C]
SC6	Ability to save the current state of the plug-in: For faster use of my solution, the user should be able to save and load previous states of each plug-in.	• The current state of the plug-in can be saved to an external file. [SC6_A] • External files can be opened, loading a previous state of the plug-in. [SC6_B]
SC7	Ability to reuse code: For both faster development of my solution and reuse of my code by other programmers, essential modules should be implemented as classes.	• Where appropriate, different modules should be implemented as classes allowing for re-use across projects. [SC7_A] • The website must have a dedicated section explaining how each modules work and how to interface with the class. [SC7_B]
SC8	Stability of code: During normal use, each plug-in must run in a stable state and not crash which could cause loss of data.	• Each plug-in must run in a stable state. They must run for over 5 minutes with a constant signal without critical errors. [SC8_A] • Each plug-in must not use an excessive amount of system resources. They should not use over 100mb memory during use. [SC8_B]

Definitions

Due to the technical nature of my project, I have created a definitions table containing any essential language I will use throughout my project.

Term	Definition
Plug-in	A piece of software which can be loaded within an existing piece of software adding functionality.
Transfer Curve	A transfer curve / function is a way of mapping a set of inputs to a set of outputs.
Transient	A very shot, loud sound generally at the start of a sound. Like the smack when you hit a snare drum.
Aliasing	A form of error where you have sampled at the wrong sampling rate. This is generally created when analogue signals are converted to digital signals.
Artifacts	Errors introduced to the signal during processing.
Clipping	A form of error where the volume of a signal is increased beyond the recommended limit introducing errors.

Bibliography

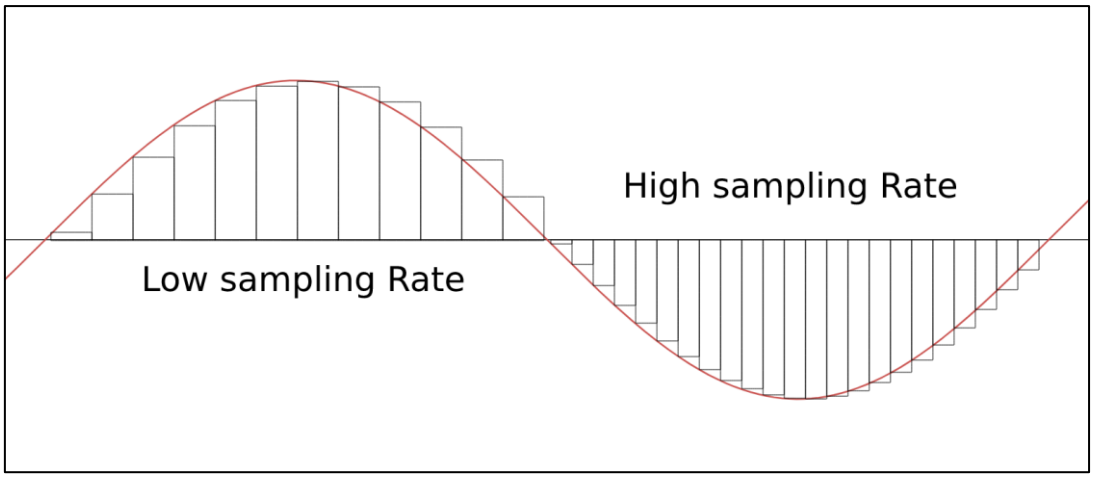
1. Monome, “Norns Shield”
2. Melda, “MFreeFXBundle”
3. Kilohearts, “Kilohearts Essentials”
4. FabFilter, “Total Bundle”
5. Ableton, “Live Audio Effect Reference”
6. JUCE, “JUCE”
7. Oli Larkin, “IPlug2 - C++ audio plug-in framework"
8. Bitwig Studio, “Bitwig Studio”
9. Reaper, “Reaper”

The Basics of Audio-Based DSP

Before I get into the core of my project, I should explain some preliminary material regarding the basics of audio-based DSP and the use of iPlug2. This should allow anyone regardless of their background in audio-based DSP to understand the basics of how each effect works.

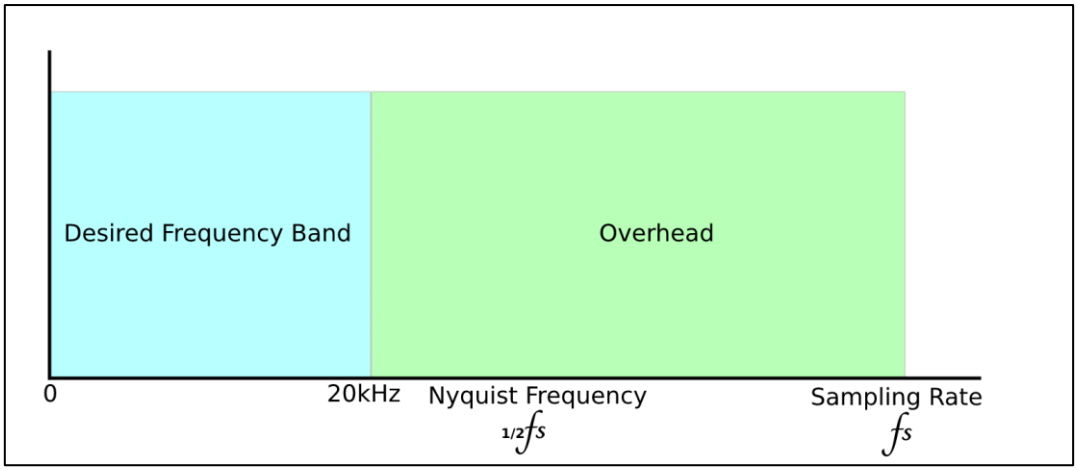
Audio-Based DSP

To kick things off let's start from the beginning, Analogue to Digital Conversion (ADC). Building on from what's covered in the course, the first step is figuring out how to turn a continuous, analogue signal into a discrete, digital set of data. The standard way of achieving this is to take Samples (small pieces of data normally 16-32 bits in size which represent the amplitude of the signal) at a fixed interval. The rate at which we take samples is called the Sampling Rate. This varies on the use case however, for consumer grade audio it is normally 44.1kHz and for professional recordings it is normally 48kHz. It is also important to know that Sampling Rate / Sampling Frequency can often be noted as f_s.



We also define a sampling depth often called Bit-Depth which states how many bits are used to represent the resolution for the amplitude of the signal. Where a higher Bit-Depth allows for a more accurate reading of the amplitude of the signal. We can think about the sampling rate being the resolution on the X-Axis and the Bit-Depth as the resolution on the Y-Axis.

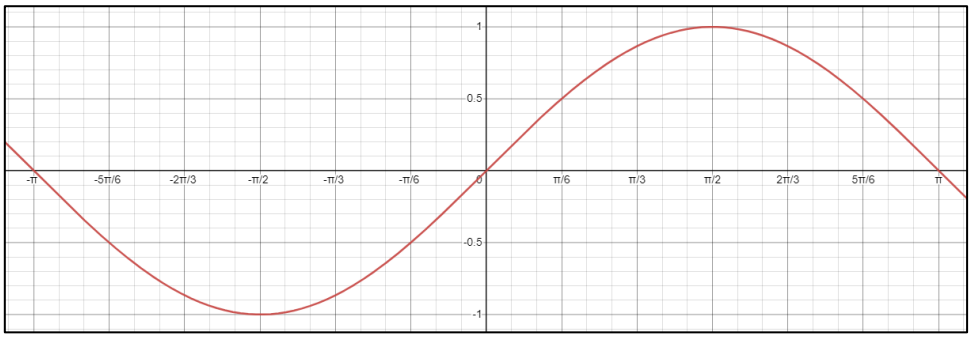
You might wonder why we use 44.1kHz or 48kHz for sampling, but there is some interesting physics behind it. If we take human hearing for example, we can hear between 20-20,000 Hz. If we sampled at 20kHz, we could lose information which would cause aliasing during processing. To combat this, we can use Nyquist's theorem to find a reasonable sampling rate. Nyquist's theorem states that you should sample at over double the highest frequency of you desired frequency band. So, if we want to work between 0-20kHz, the minimum sampling rate without aliasing is 40kHz. But for redundancy we sample at 44.1kHz to reduce quantisation errors. Professions use 48kHz because it is slightly higher quality without a significant increase in file size.



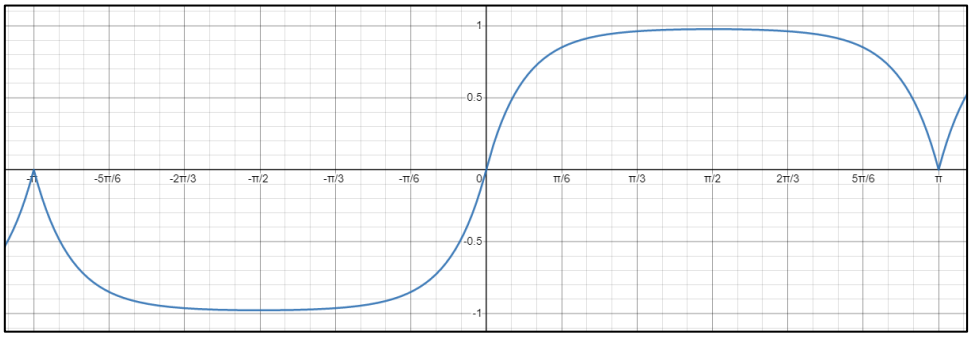
After all this preparation we finally have our signal in a usable medium. This allows us to process the signal in two main ways: Basic processing and Spectral processing. Below is a description of how each works.

Basic processing

Basic processing only manipulates the raw data as it is. Whether this be through altering the amplitude of the wave or repeating it in a buffer. We can see its use in effects such as Distortion where we use a transfer function to change the amplitude of the wave. This is the easiest form of processing to understand because it is the simplest to visualise.



Pre-Transfer Function

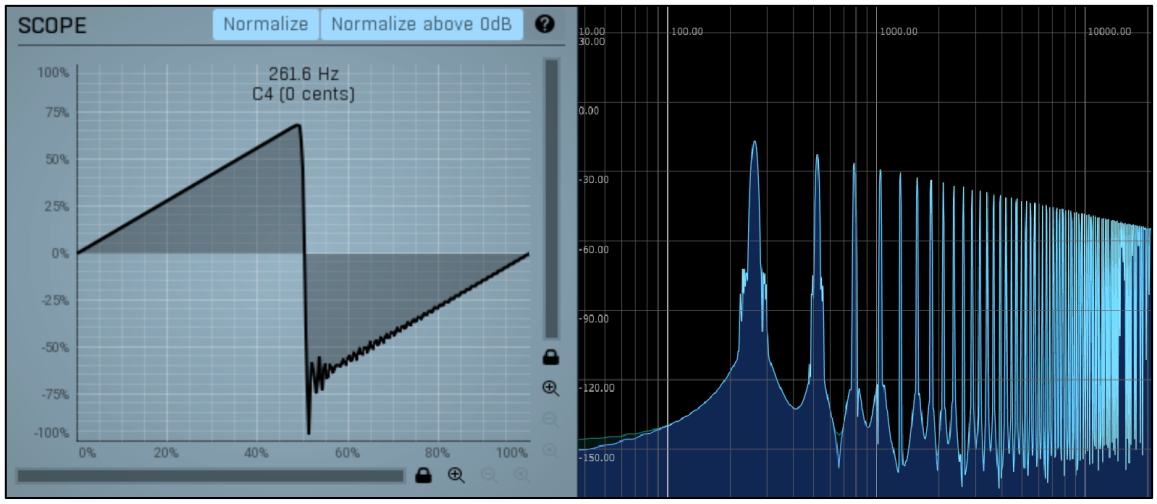


Post-Transfer Function

Spectral processing

Spectral processing converts the signal from amplitude over time to amplitude over frequency which allows for more complex processing of a signal. An easy visualisation is that amplitude over time is a Oscilloscope whereas amplitude over frequency is a frequency spectrum. Seeing as this allows us to alter the frequency content of a signal it is essential to any modern plug-in package because some effects would be impossible to create without it.

It is called Spectral processing because the process of running a signal through a FFT (Fast Fourier Transform) algorithm breaks down the signal into its component frequencies. This idea of using the partials (harmonics) of a sound for composition is the basis for a movement called Spectralism and is the namesake for this type of processing. After the signal has processed, we can recombine all these partials back into a normal signal using an IFFT (Inverse Fast Fourier Transform) algorithm.



Pre FFT Post FFT

History

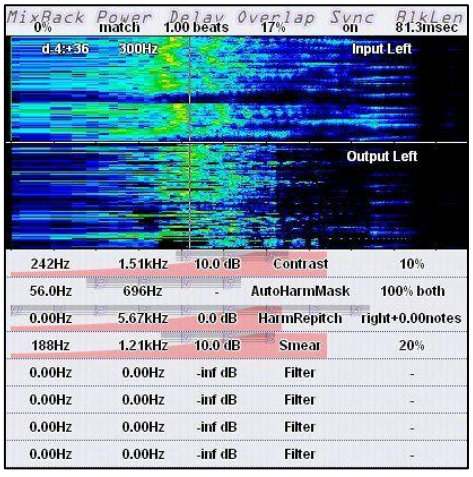
The Fourier Transform was originally derived by Joseph Fourier in 1822 when he was trying to find a method to decompose a signal into a sum of sine waves. He found that if you multiply your signal by a sine wave at a desired frequency e.g. 20Hz. If this frequency is present in the original signal, then the area under the curve would be a nonzero value. This area under the curve is the relative amplitude of that sinewave in the original signal. If we repeated this process for every frequency, we would get an accurate representation of how much each frequency is present in the signal. However, if we repeat the same process with cosine waves, we can account for the phase of that frequency as well. If we want to simplify this further, we can use Euler's formula to represent the sum of our sine and cosine wave in one expression $e^{2\pi ift}$, where f denotes frequency and t denotes the time. This exponential can then be broken down into $A\cos(2\pi ft) + Bi\sin(2\pi ft)$ where A and B are their relative amplitudes. Traditionally the Fourier Transform can be defined as $h(f) = \int_{-\infty}^{\infty} h(t)E^{-2\pi ift}\,dx$. This method works perfectly well in principle however it only works if we are using a continuous signal. This is fine for analogue signals however when we want to process them digitally, we run into problems. This leads us to using the DFT (Discrete Fourier Transform) algorithm, which achieves the same effect the Fourier Transform however has a few drawbacks. The output Frequency range depends on the Sample Rate I.e., the higher the Sample Rate the larger band of frequencies you can measure because there isn't enough resolution to capture high-

frequency noise. The resolution between frequencies (how easy it is to tell different frequencies apart) depends on the length of the inputted data. The shorter the duration of the signal, the wider each frequency bin is. When we apply the DFT algorithm to each sample we must multiply every sample by every frequency between 0 and the number of samples being processed. This leads to a time complexity of $O(N^2)$ which isn't ideal for Audio-Processing because it requires low latency.

This leads on to the FFT (Fast Fourier Transform) algorithm. In 1956 John Tukey developed the FFT algorithm, and with the help of James Cooley implemented it at IBM's Watson Research centre. The FFT algorithm can be defined as $g(f) = \sum_{n=0}^{N-1} g(t)e^{\frac{-2\pi ifn}{N}}$. The FFT algorithm is an improvement over the Fourier Transform owing to it reducing the number of calculations that need to be performed. When looking at how the Fourier Transform worked Tukey found patterns within the algorithm which allowed him to reduce the complexity from $O(N^2)$ to $O(N\log_2 N)$. This means that the FFT algorithm gets even faster with a larger sample size and more importantly makes it viable for low latency software. A more in-depth explanation of the Fourier Transform, and Fast Fourier Transform can be found in Veritasium's video 'The Remarkable Story Behind the Most Important Algorithm of All Time' [1].

Uses

FFT processing can traditionally be seen on effects such as filters, spectrograms and auto-tune owing to them using the frequency spectrum. However, these effects only scratch the surface of what the FFT algorithm is capable of. A good example of experimental spectral processing is DtBlkFX [2] by Darrell Barrell, which combines traditional filters with complex spectral processing to create unique sounds.



Bibliography:

1. Veritasium, "The Remarkable Story Behind the Most Important Algorithm of All Time"
2. Darrell Barrell, "DtBlkFX"

An Overview of iPlug2

To save time later I will give a basic overview of how iPlug2 works and any key features and functions that I will be using during development.

Firstly, I'll explain how the audio handling works from input to output. The plug-in receives an audio stream which can contain one or more channels of audio, each channel represents the amplitude of its signal at the current sample stored as a double. These channels are processed in the ProcessBlock function, which is called every sample.

```
#if IPLUG_DSP
void IPlugEffect::ProcessBlock(sample** inputs, sample** outputs, int nFrames)
{
  const double gain = GetParam(kGain)->Value() / 100.;
  const int nChans = NOutChansConnected();

  for (int s = 0; s < nFrames; s++) {
    for (int c = 0; c < nChans; c++) {
      outputs[c][s] = inputs[c][s] * gain;
    }
  }
}
#endif
```

When outputting the processed audio IPlug2 works as a flushed system. Audio can only be sent to the host software each sample to prevent timing errors. This also allows the host software to calculate the amount of latency a plug-in creates, allowing for correction later. This system can be visualised as a bus timetable. The bus leaves at set times and if you miss the bus you must wait for the next one. However, if you arrive at the bus stop early you must wait for the next bus giving us a window to complete the calculation. The processing doesn't have to be the fastest, just fast enough to arrive before the bus leaves.

This puts optimisation of code at the forefront of the project. Ideally, there would be no latency when using any effect allowing the user to hear changes back in real time. However, when there is a noticeable amount of latency from one plug-in or the build-up of latency from many plug-ins chained together, it makes it incredibly difficult to play an instrument and hear yourself owing to the slight delay.

During programming there is a set of useful functions built into iPlug2 which can be used to get values from the host software. Such functions include; GetTempo(), GetSamplePos(), GetSamplesPerBeat() and GetSampleRate(). These do what they say on the tin and are very useful for certain effects. For example, when creating a time synced delay GetSamplesPerBeat() is a useful function when implementing tempo sync reducing the need to derive an equation for samples per beat yourself.

When handling the GUI, iPlug2 creates a canvas which; controls, text and images can be attached to. These controls are attached in the constructor and can be used to manipulate specified variables. All

of this is defined in the constructor and the variables we modify with the controls are defined in an Enum; EParams. I can add new parameters by including them in the main header inside of Eparams.

```
mLayoutFunc = [&](IGraphics* pGraphics) {

  // Create Canvas
  pGraphics->AttachBackground(GUI_FN);
  pGraphics->LoadFont("Roboto-Regular", ROBOTO_FN);

  // Load Images
  const IBitmap Bias = pGraphics->LoadBitmap(BIAS_FN, 128);
  const IBitmap Dynamics = pGraphics->LoadBitmap(DYNAMICS_FN, 128);
  const IBitmap Filter = pGraphics->LoadBitmap(FILTER_FN, 128);
  const IBitmap Gain = pGraphics->LoadBitmap(GAIN_FN, 128);
  const IBitmap Rectify = pGraphics->LoadBitmap(RECTIFY_FN, 2);
  const IBitmap Bypass = pGraphics->LoadBitmap(BYPASS_FN, 2);
  const IBitmap Mode = pGraphics->LoadBitmap(MODE_FN, 5);

  // Attach controls
  pGraphics->AttachControl(new IBKnobControl(204, 33, Bias, kBias));
  pGraphics->AttachControl(new IBKnobControl(55, 136, Dynamics, kDynamics));
  pGraphics->AttachControl(new IBKnobControl(204, 136, Filter, kTone));
  pGraphics->AttachControl(new IBKnobControl(55, 33, Gain, kGain));
  pGraphics->AttachControl(new IBSwitchControl(194, 277, Rectify, kRectify));
  pGraphics->AttachControl(new IBSwitchControl(46, 261, Mode, kMode));
  pGraphics->AttachControl(new IBSwitchControl(211, 392, Bypass, kButton));

};
#endif
}
```

When creating a new control, we define the type and range for the variable we'll be modifying. This is best shown through an example. We use GetParam() to grab the variable we created in the Enum and set its initial conditions and constraints. When deciding on the data type we have a choice from; Double, Enum, Int and Bool.

```
// Bias Knob
GetParam(kBias)->InitDouble("Bias", 0.0, -0.5, 0.5, 0.01, "%");

// Gain Knob
GetParam(kGain)->InitDouble("Gain", 1.0, 1.0, 36.0, 0.01, "%");

// Bypass Switch
GetParam(kButton)->InitEnum("On Off", 0, 1);

// Mode Switch
GetParam(kMode)->InitInt("Mode", 0, 0, 4);

// Rectify button
GetParam(kRectify)->InitEnum("Rectify", 0, 1);

// Dynamics Knob
GetParam(kDynamics)->InitDouble("Dynamics", 1.0, 0.0, 1.0, -0.01, "%");

// Tone Knob
GetParam(kTone)->InitDouble("Tone", 0.0, -8.0, 8.0, 0.01, "db");
```

In this example we are initialising kGain and are settings its type to double. As shown in the comment we define a set of initial conditions and constraints allow the control to work. For kGain we name it "Gain", give it a default value of 50.0 and set its range between 0-100, we allow it to increase in increments of 0.1 and give it a label of "%" which is displayed in the host software allowing the user to interact with the plug-in without opening the GUI.

Next, we load different images into the canvas which can be attached to different controls. To put an image into the framework we first move it into “resources\img” and then define it in the project Resource Script file. After this we include it the config.h file and it is ready for use. We can then define a new image in the canvas and name it accordingly, we load the image we just linked, and we can define a frame count. This allows us to create animated controls with 1 image.

```
// Assets
// Fonts
#define ROBOTO_FN "Roboto-Regular.ttf"

//Images
#define BIAS_FN "BiasKnob.png"
#define BYPASS_FN "Bypass.png"
#define DYNAMICS_FN "DynamicsKnob.png"
#define FILTER_FN "FilterKnob.png"
#define GAIN_FN "GainKnob.png"
#define GUI_FN "gui.png"
#define MODE_FN "Mode.png"
#define RECTIFY_FN "Rectify.png"
```

Finally, we can define our new control. When creating a new control, we have a wide range of choices however the most useful controls are; IBKnobContol, IBSwitchControl and ITextControl. We do this through pGraphics->AttachControl() which again does what it says on the tin. In this we can define the type of control, place it on the GUI and assign the relevant image to the control. After all of this we have finally created a new image.

```
// Attach controls
pGraphics->AttachControl(new IBKnobControl(204, 33, Bias, kBias));
pGraphics->AttachControl(new IBKnobControl(55, 136, Dynamics, kDynamics));
pGraphics->AttachControl(new IBKnobControl(204, 136, Filter, kTone));
pGraphics->AttachControl(new IBKnobControl(55, 33, Gain, kGain));
pGraphics->AttachControl(new IBSwitchControl(194, 277, Rectify, kRectify));
pGraphics->AttachControl(new IBSwitchControl(46, 261, Mode, kMode));
pGraphics->AttachControl(new IBSwitchControl(211, 392, Bypass, kButton));
```

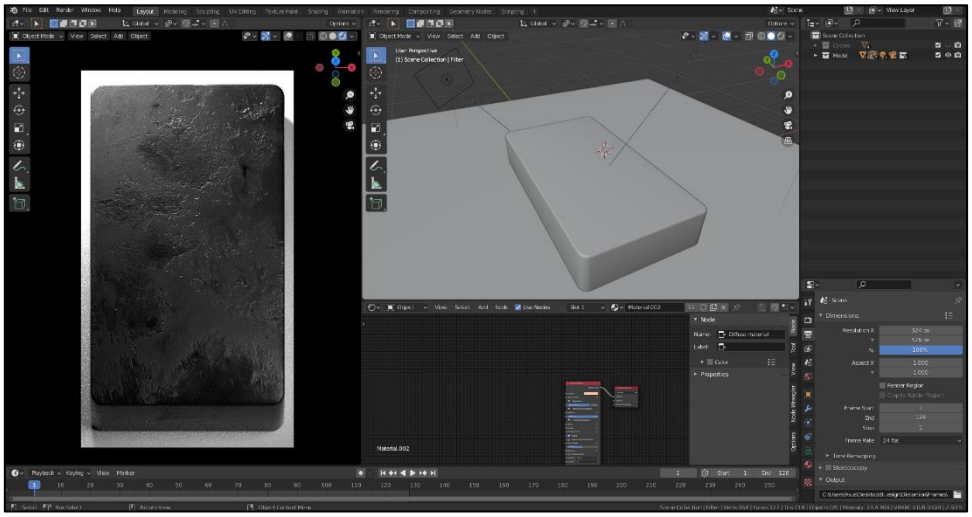
IPlug2 also has native support for presets which fully addresses [SC6] however, if possible, I will reimplement this system allowing for everything to be contained within the plugin. Finally, Oli Larkin goes into more detail about iPlug2 and its innerworkings in a presentation he made about on ‘The Audio Programmer’ YouTube channel [1].

Bibliography:

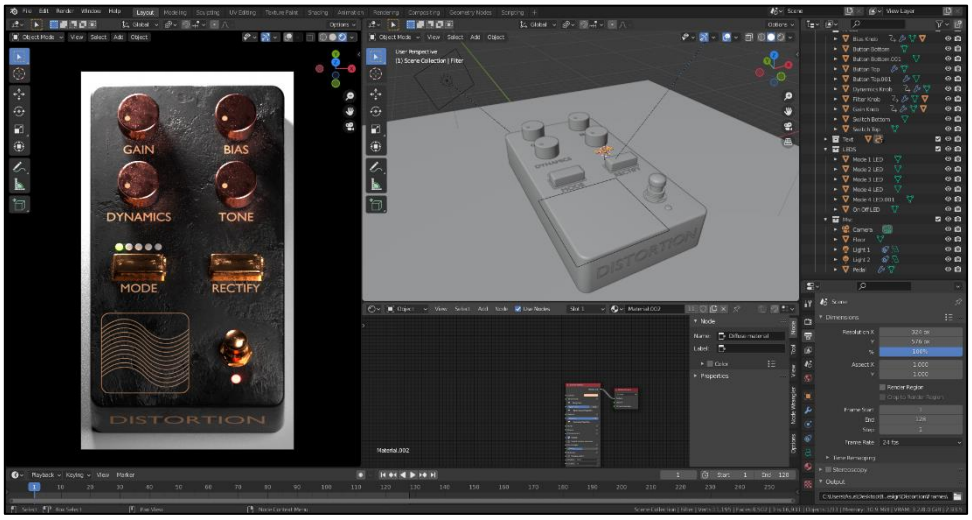
1. The Audio Programmer, "Oli Larkin - An Introduction to iPlug2"

GUI

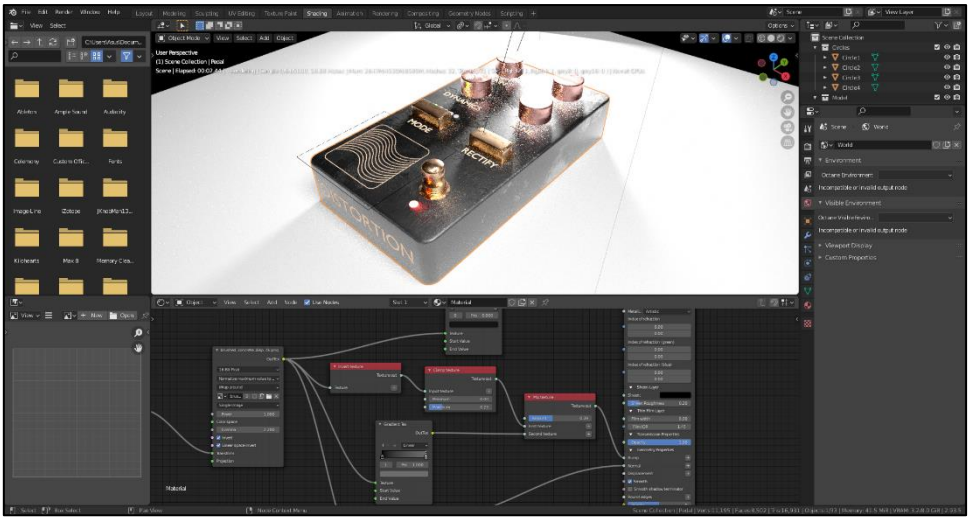
When designing the GUI, I want to make every control easily accessible without the whole design becoming cluttered. After assessing [SC1] I decided to base my GUI on a guitar pedal because it fulfils [SC1_A], [SC1_B] and [SC1_D]. It also has the added benefit of looking more familiar to beginners helping with ease of use. I modelled a basic guitar pedal in Blender [1] which I could then modify for each plugin.



From this template I can add any controls and labels I need. The added benefit of making the pedal in Blender is that I both have a lot of experience with the software, and I can make the pedal look photorealistic. This makes the plugin look more professional with is an added benefit when attracting new users.



In Blender I can customise the lighting and materials for every object in the scene. This allows me to give the whole GUI a photorealistic effect. It also allows me to create an illusion that when a knob is moved the lighting changes accordingly. I chose to make the body of the pedal out of a dark stone with the controls and labels being a shiny, coloured metal.



When animating the different controls, I must follow certain steps to make everything look cohesive. Firstly, I render a picture of the pedal at the right dimensions so I can calculate the width and height of each control's picture. Then I change the camera's resolution to match this and animate the controls. After I have animated all the controls, I have a sequence of images that I can stitch together to create 1 big image.

To animate a control in iPlug2 each frame needs to appear underneath the previous frame in one picture. To stich all these images together vertically, I found a website which conveniently does this [2]. I have already explained how to import an image into iPlug2 in the previous section, so I won't go over it again. After all of this I have completed GUI for a plugin.

Bibliography:

1. Blender, “Blender”
2. jgraph, “Draw.io Tools - Merge”

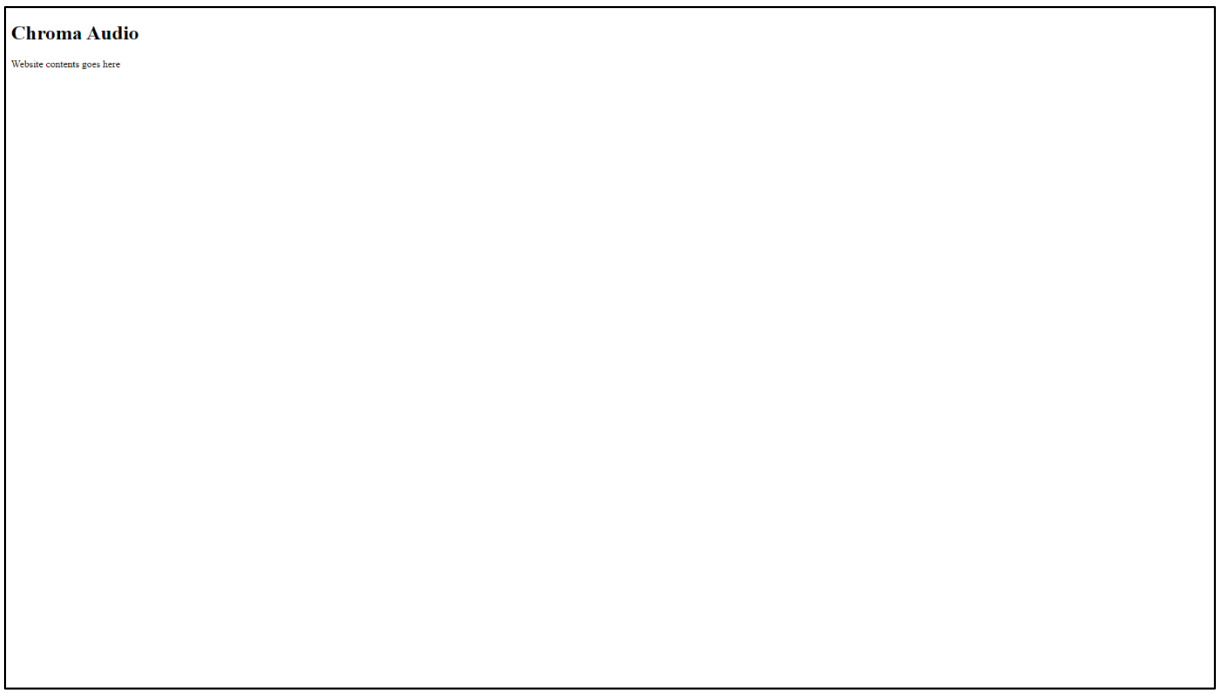
Website

Since it is essential to have a website with any commercial product, I will create my own using, HTML and CSS. This allows me to both improve the user experience and use concepts that I have learned from the course. Alongside HTML and CSS, I will use Bootstrap 5 [1] to help with the presentation of my website. After researching the designs of different websites, I created a plan which details which pages I need to program:

About	Landing page / show general information.
Blog	Show any progress updates on software.
Plug-ins: • Delay • Distortion • Flanger • Compressor	Show information about each plug-in.
Contact	Has a form to report bugs and suggest features.
Other: • Downloads & Installation • Presets • Documentation • FAQ • Terms and Conditions	• Information on downloading and installing software. • Information about presets. • Has links to documentation for each plug-in. • Frequently asked questions. • Self-explanatory.

Although it looks like a lot of work, once I get a template for each page designed it's just a case of customising each page's contents.

Initially I set up a website using Github pages, since I was already backing up my plug-ins on Github and I've used it before. I created my index.html and set up some basic HTML to fill out the page a bit, I also added a favicon with the logo I designed for the project. I also set up some folders and created my master CSS document which I will link to all my webpages. At this stage the website and its code looked like this.



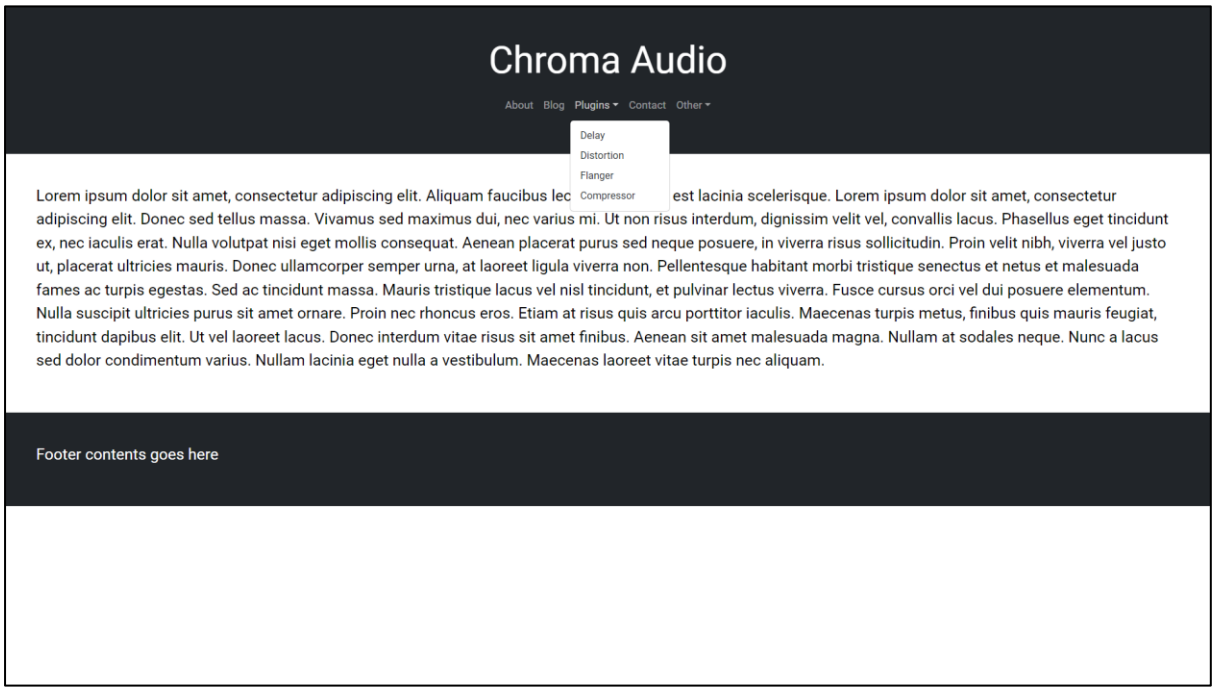

```html
<!DOCTYPE html>
<html>
  <head>
    <title>Chroma Audio</title>
    <link rel="icon" type="image/x-icon" href="Website/Favicon.png">
  </head>
  <body>
    <h1>Chroma Audio</h1>
    <p>Website contents goes here</p>
  </body>
</html>
```

With no styling to the website, I thought I should start using bootstrap. I included it in the header and got to work. I created three main sections, the header, the content and the footer. This split the page up into three manageable chunks which I could apply separate bootstrap styles to. I opted to go for a white and dark grey aesthetic since I could use the build in colours with Bootstrap and it looks nice. It also allows me to use contrasting colours making the website more accessible to the visually impaired addressing [SC1]. I created the three sections using the tags, within these I added some Bootstrap code which allowed me to change the colour of the section, add some padding and change the text colour. As with all my divs, I am using a fluid container since I want them to span the whole width of the page.


```html
<!DOCTYPE html>
<html>
  <head>
    <title>Chroma Audio</title>
    <link rel="icon" type="image/x-icon" href="Website/Favicon.png">

    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css" rel="stylesheet">
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/js/bootstrap.bundle.min.js"></script>

    <link rel="preconnect" href="https://fonts.googleapis.com">
    <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
    <link rel="stylesheet" href="Website/Website.css">
    <link href="https://fonts.googleapis.com/css2?family=Roboto&display=swap" rel="stylesheet">
  </head>
  <body>
    <!-- Header -->
    <div class="container-fluid p-5 bg-dark text-white">
      <h1>Chroma Audio</h1>
    </div>

    <!-- Content -->
    <div class="container-fluid p-5 text-black">
      <p>
        Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam faucibus lectus commodo est lacinia scelerisque.
        Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec sed tellus massa. Vivamus sed maximus dui, nec varius mi.
        Ut non risus interdum, dignissim velit vel, convallis lacus. Phasellus eget tincidunt ex, nec iaculis erat. Nulla volutpat nisi eget mollis consequat.
        Aenean placerat purus sed neque posuere, in viverra risus sollicitudin. Proin velit nibh, viverra vel justo ut, placerat ultricies mauris.
        Donec ullamcorper semper urna, at laoreet ligula viverra non. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.
        Sed ac tincidunt massa. Mauris tristique lacus vel nisl tincidunt, et pulvinar lectus viverra. Fusce cursus orci vel dui posuere elementum.
        Nulla suscipit ultricies purus sit amet ornare. Proin nec rhoncus eros. Etiam at risus quis arcu porttitor iaculis.
        Maecenas turpis metus, finibus quis mauris feugiat, tincidunt dapibus elit. Ut vel laoreet lacus. Donec interdum vitae risus sit amet finibus.
        Aenean sit amet malesuada magna. Nullam at sodales neque. Nunc a lacus sed dolor condimentum varius. Nullam lacinia eget nulla a vestibulum. Maecenas laoreet vitae turpis nec aliquam.
      </p>
    </div>

    <!-- Footer -->
    <div class="container-fluid p-5 bg-dark text-white">
      <p>Footer contents goes here</p>
    </div>
  </body>
</html>
```

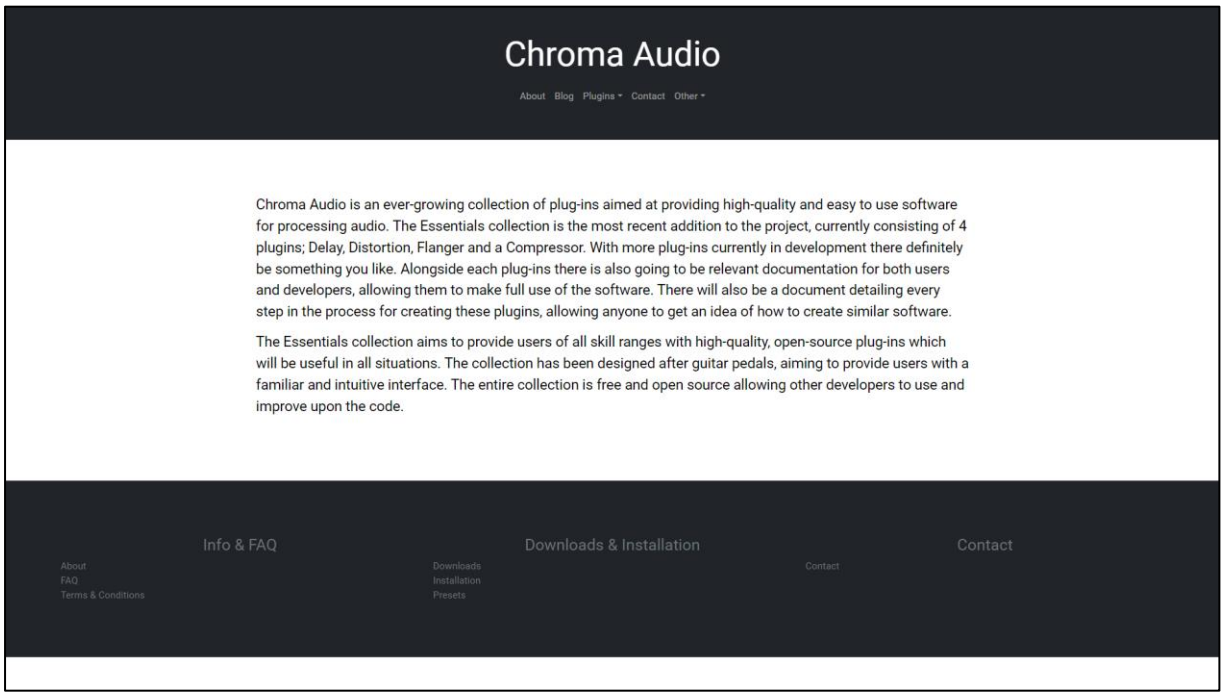
I filled out the context section with some Lorem Ipsum within the <p> tag just so I could test out different fonts and get a feel for how the page would look. Apart from filling out the footer I also added a navbar using Bootstrap. I also looked through Google Fonts to find a good-looking font. I settled on ‘Roboto’ since it has a modern but simple look.

I initially tried to make the navbar using CSS however I couldn't get dropdown menus within the navbar to work. After finding out that I can implement a navbar with dropdown menus in Bootstrap I got it to work. I also styled the navbar using more code from Bootstrap.




```
<!-- Header -->
<div class="container-fluid p-5 bg-dark text-white">
  <h1>Chroma Audio</h1>

  <nav class="navbar navbar-expand-sm bg-dark navbar-dark" id="centre">
    <div class="container-fluid">
      <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#collapsibleNavbar">
        <span class="navbar-toggler-icon"></span>
      </button>
      <div class="collapse navbar-collapse justify-content-center" id="collapsibleNavbar">
        <ul class="navbar-nav">
          <li class="nav-item">
            <a class="nav-link" href="">About</a>
          </li>
          <li class="nav-item">
            <a class="nav-link" href="">Blog</a>
          </li>
          <li class="nav-item dropdown">
            <a class="nav-link dropdown-toggle" href="#" role="button" data-bs-toggle="dropdown">Plugins</a>
            <ul class="dropdown-menu">
              <li><a class="dropdown-item" href="">Delay</a></li>
              <li><a class="dropdown-item" href="">Distortion</a></li>
              <li><a class="dropdown-item" href="">Flanger</a></li>
              <li><a class="dropdown-item" href="">Compressor</a></li>
            </ul>
          </li>
          <li class="nav-item">
            <a class="nav-link" href="">Contact</a>
          </li>
          <li class="nav-item dropdown">
            <a class="nav-link dropdown-toggle" href="#" role="button" data-bs-toggle="dropdown">Other</a>
            <ul class="dropdown-menu">
              <li><a class="dropdown-item" href="">Downloads & Installation</a></li>
              <li><a class="dropdown-item" href="">Presets</a></li>
              <li><a class="dropdown-item" href="">Documentation</a></li>
              <li><a class="dropdown-item" href="">FAQ</a></li>
              <li><a class="dropdown-item" href="">Terms and Conditions</a></li>
            </ul>
          </li>
        </ul>
      </div>
    </div>
  </nav>
</div>
```

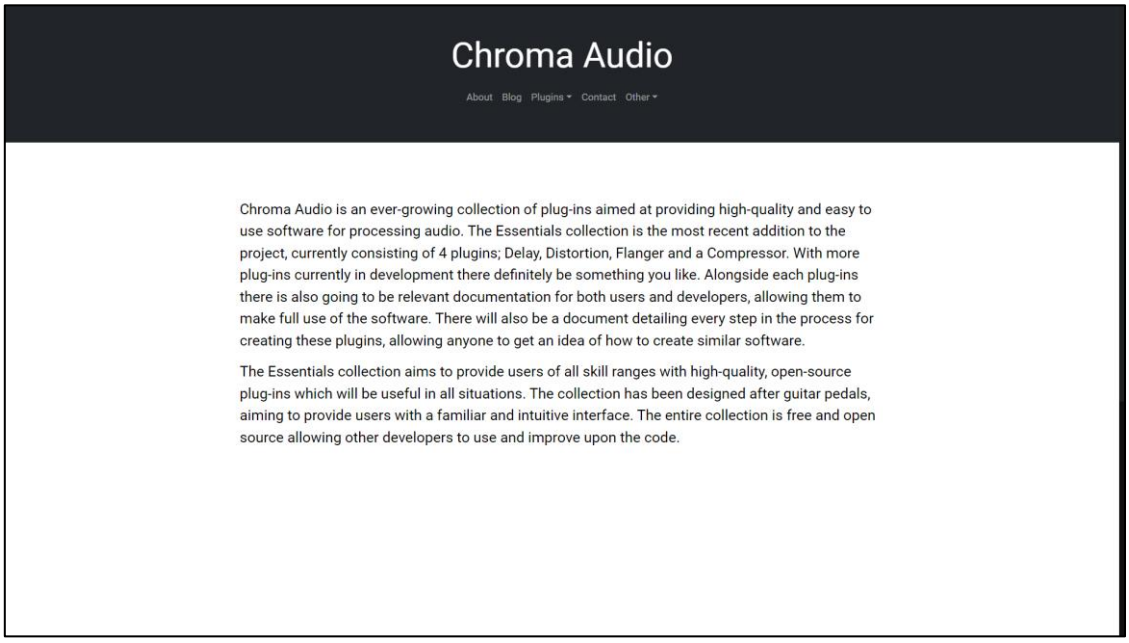
After finishing off the header I moved onto the footer. I used Bootstrap's grid system to create three equally sized grids where I could organise any links I want at the bottom of the page. The three sections are; Info and FAQ, Downloads & Installation and Contact. Within each grid, I head a heading and then links to different pages on my website. All the links are empty at the moment, but I will populate them when I have created all the different pages. I also decided to do the grid system with the content section, both for aesthetics and for functionality since I could add more content within the space I've created. Alongside this I replaced the Lorem ipsum with the actual text I'm going to use.




```
61        <!-- Content -->
62        <div class="container-fluid p-5 text-black">
63          <div class="row">
64            <div class="col-sm-2"></div>
65            <div class="col-sm-8">
66              <div class="container-fluid p-5 text-black">
67                <!-- Text -->
68                <p>
69                  Chroma Audio is an ever-growing collection of plug-ins aimed at providing high-quality and easy to use software for processing audio.
70                  The Essentials collection is the most recent addition to the project, currently consisting of 4 plugins; Delay, Distortion, Flanger and a Compressor.
71                  With more plug-ins currently in development there definitely be something you like.
72                  Alongside each plug-ins there is also going to be relevant documentation for both users and developers, allowing them to make full use of the software.
73                  There will also be a document detailing every step in the process for creating these plugins, allowing anyone to get an idea of how to create similar software.
74                </p>
75
76                <!-- Text -->
77                <p></p>
78                <p>
79                  The Essentials collection aims to provide users of all skill ranges with high-quality, open-source plug-ins which will be useful in all situations.
80                  The collection has been designed after guitar pedals, aiming to provide users with a familiar and intuitive interface.
81                  The entire collection is free and open source allowing other developers to use and improve upon the code.
82                </p>
83              </div>
84            </div>
85            <div class="col-sm-2"></div>
86          </div>
87        </div>
```

When I was looking at my website, the footer was under the content section however, it wasn't sticking to the bottom of the page. I initially tried to use some Bootstrap code which allows you to stick divs to the top and bottom of the page however this kept it at the bottom of the page all the

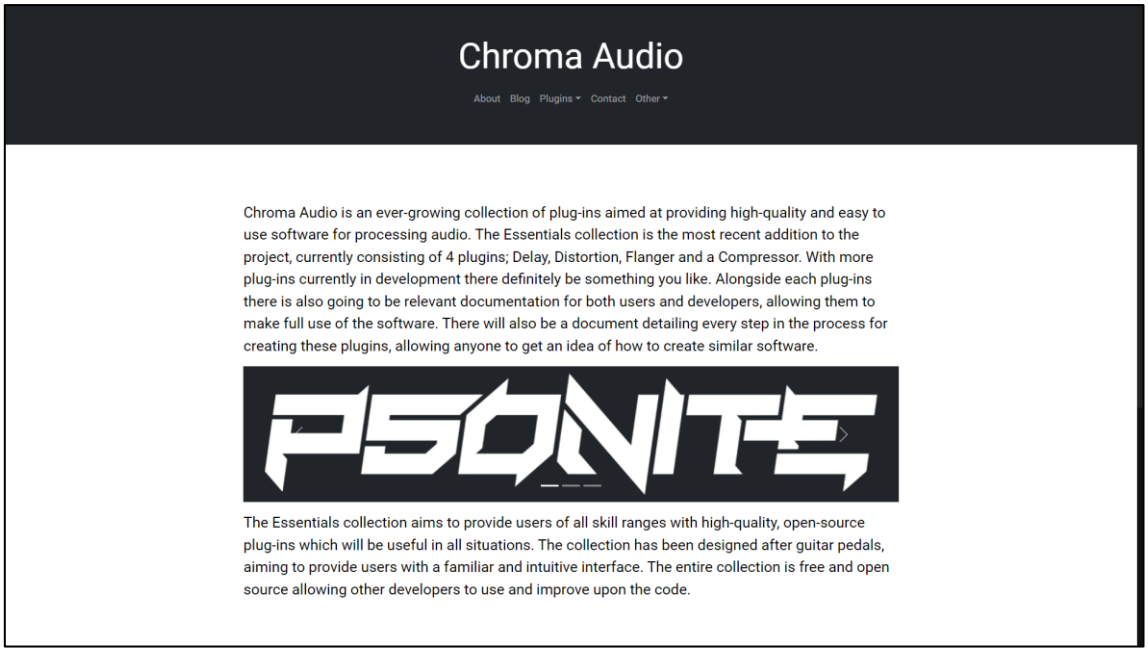
time, instead of only showing itself when you're at the bottom of the page. I fixed this by creating a new class in CSS called filler, in this class I used the min-height attribute to set the minimum height of the applied div to be the same as the full height of the webpage. This meant that when there wasn't enough content on the page, blank space will fill up the rest of the page, so you have to scroll down to fill the footer. However, if there is enough content in this div to fill up the page then nothing happens. This completed the template for each page allowing me to start customising each page. On the topic of CSS, apart from using it to style certain elements like always centring <h3> I used it to create a black scroll bar to fit with the theme of the website. I did this using the [::-webkit-scrollbar] pseudo element in CSS which allows you to customise elements of the browser.



```css
body {
  font-family: 'Roboto', sans-serif;
  font-size: 17px;
}

h1 {
  text-align: center;
  font-size: 60px;
}

h2 {
  text-align: left;
  text-decoration: underline;
}

h3 {
  text-align: center;
}

p {
 font-size: 25px;
}

.filler {
  min-height: 100vh;
}

::-webkit-scrollbar {
    width: 10px;
}

::-webkit-scrollbar-track {
    background: #111;
}

::-webkit-scrollbar-thumb {
    background: #222;
}

::-webkit-scrollbar-thumb:hover {
    background: #333;
}
```

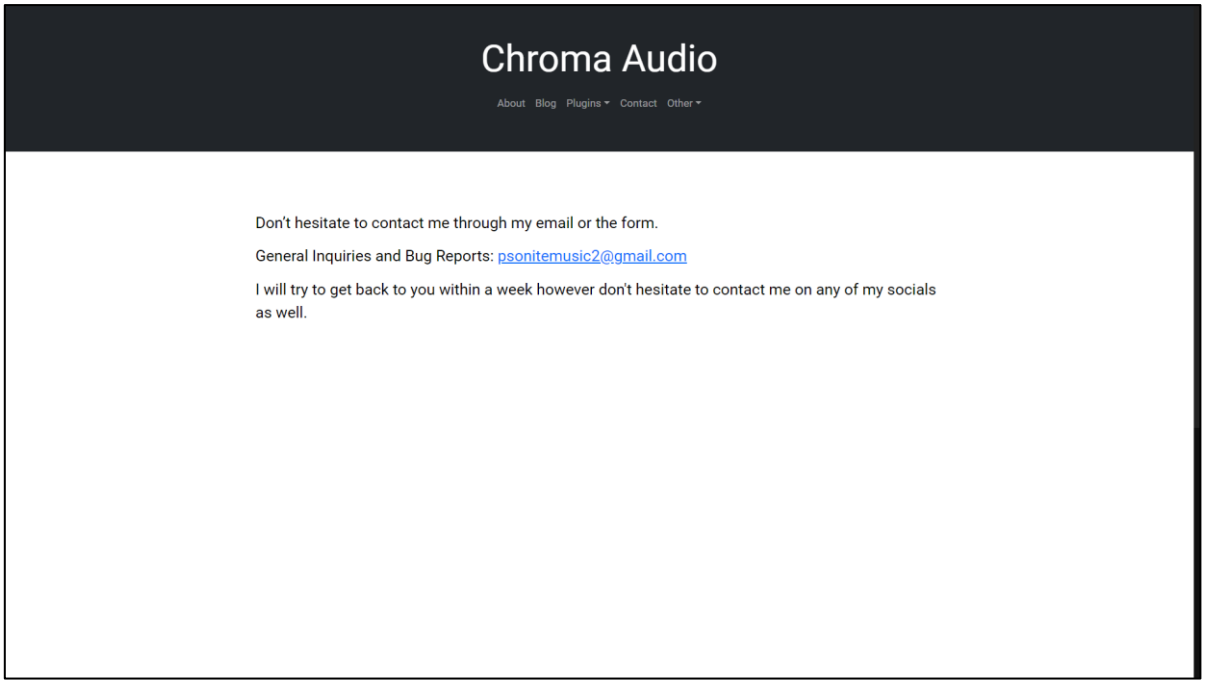
Since the only thing differing from each webpage is the content, I will only go over any new uses of HTML, CSS or any elements of Bootstrap. All the code for the website is available on my Github if you want to look at anything in particular.

I really wanted to implement a carousel on the landing page since I can use it to display any information and picture of plug-ins. I implemented this using Bootstrap which was simple after following a tutorial on W3 schools' website [2]. To fill the carousel, I put my music Alias's logo inside it, however I will change the images in it when I'm finalising the website. Here is the code for the carousel.




```
<!-- Carousel -->
<div class="container-fluid bg-dark">
  <div id="demo" class="carousel slide" data-bs-ride="carousel">

    <!-- Indicators/dots -->
    <div class="carousel-indicators">
      <button type="button" data-bs-target="#demo" data-bs-slide-to="0" class="active"></button>
      <button type="button" data-bs-target="#demo" data-bs-slide-to="1"></button>
      <button type="button" data-bs-target="#demo" data-bs-slide-to="2"></button>
    </div>

    <!-- The slideshow/carousel -->
    <div class="carousel-inner">
      <div class="carousel-item active">
        <img src="Website/logo.png" alt="Logo" class="d-block" style="width:100%">
      </div>
      <div class="carousel-item">
         <img src="Website/logo.png" alt="Logo" class="d-block" style="width:100%">
      </div>
      <div class="carousel-item">
        <img src="Website/logo.png" alt="Logo" class="d-block" style="width:100%">
      </div>
    </div>

    <!-- Left and right controls/icons -->
    <button class="carousel-control-prev" type="button" data-bs-target="#demo" data-bs-slide="prev">
      <span class="carousel-control-prev-icon"></span>
    </button>
    <button class="carousel-control-next" type="button" data-bs-target="#demo" data-bs-slide="next">
      <span class="carousel-control-next-icon"></span>
    </button>
  </div>
</div>
```

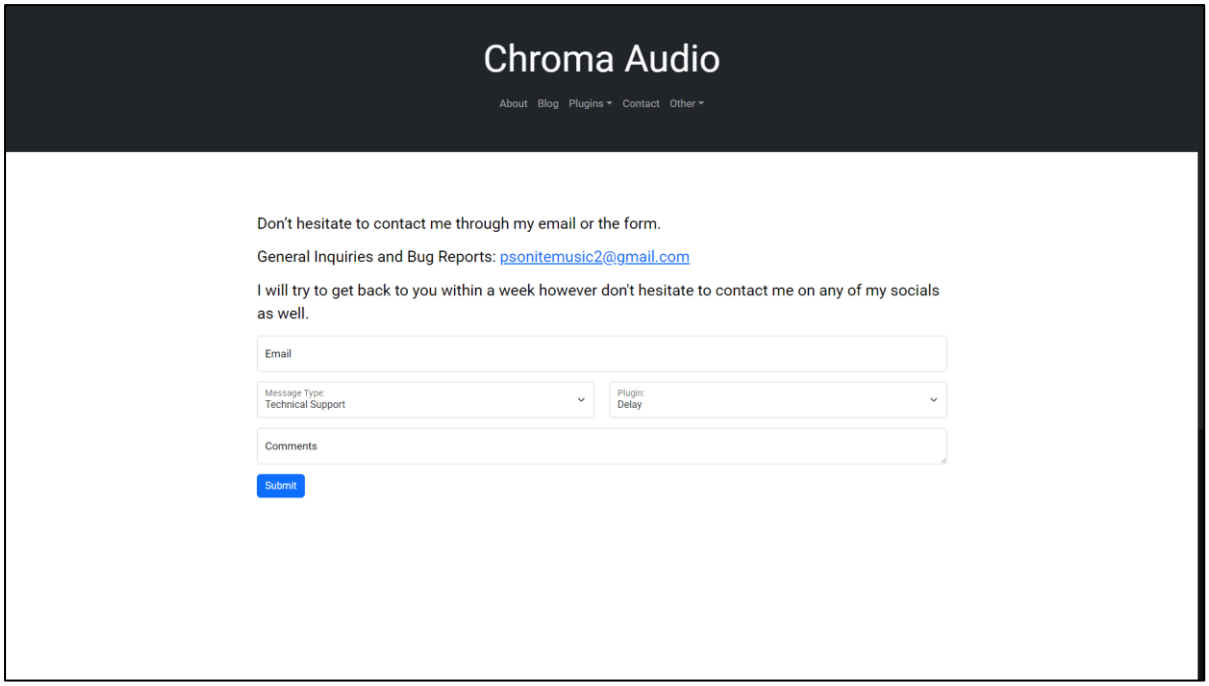
Since the blog is just a collection of headings and paragraphs there isn't anything worth talking about. I decided to move onto the contact page since I could use Bootstrap to implement a form. I created some other folders within my repository to organise my files and then I started work on creating a form. I filled out the top of the webpage with some information on how to contact me. I included a link within a paragraph allowing the user to click on my email address and have the browser open their preferred email software, allowing them to quickly send me an email. I did this through the <a> tag within my paragraph, utilising the 'mailto:____' within the href open the email software.




```
<!-- Content -->
<div class="container-fluid p-5 text-black">
  <div class="row filler">
    <div class="col-sm-2"></div>
    <div class="col-sm-8">
      <!-- Text -->
      <div class="container-fluid p-5 text-black">
        <p>Don't hesitate to contact me through my email or the form.</p>
        <p>General Inquiries and Bug Reports: <a href="mailto:psonitemusic2@gmail.com">psonitemusic2@gmail.com</a></p>
        <p>I will try to get back to you within a week however don't hesitate to contact me on any of my socials as well.</p>
      </div>
    </div>
    <div class="col-sm-2"></div>
  </div>
</div>
```

After this I moved onto the form. When looking at other contact forms I decided to have 4 inputs. Their email, what type of query their making, what plug-in it referring to and a text input box where they can detail their problem or anything else they want to tell me. Since I had the choice of which type of inputs I used, I opted for floating label inputs since they look fancier. This makes the label for each input float above the user's input instead of disappearing.

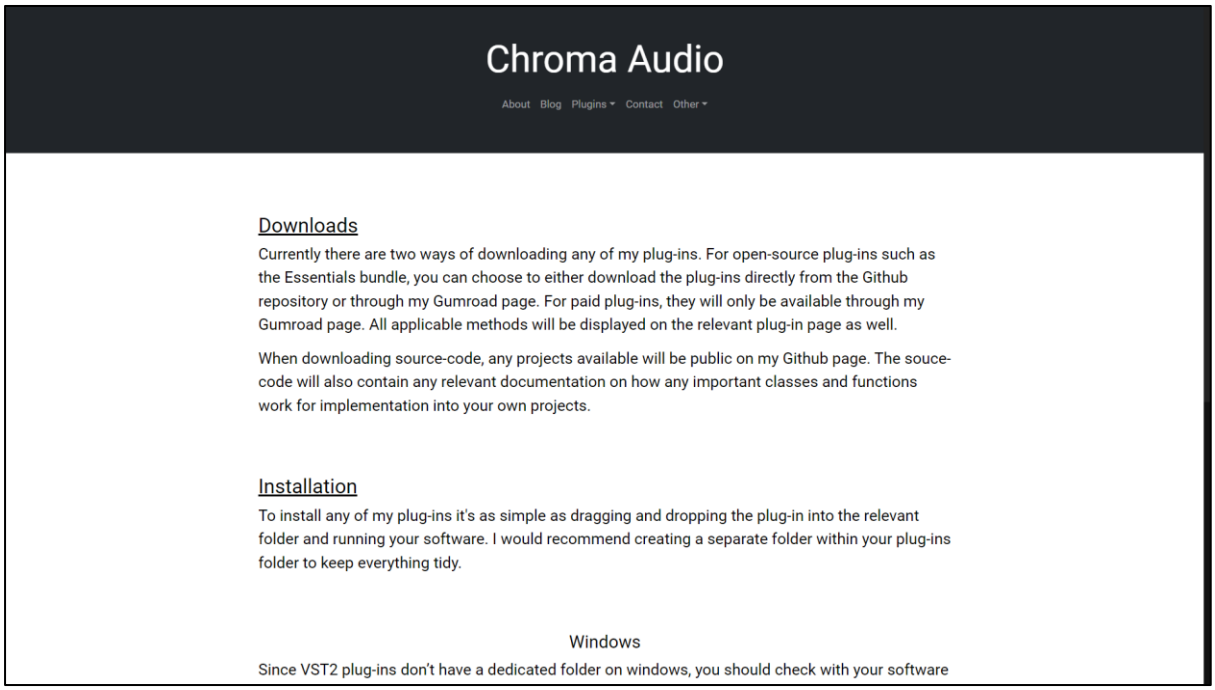
I created a div for the form without any padding so it would fit underneath the information above and stay aligned. I created a text input for emails, then I used Bootstrap's row and column feature to put the Query type and Plugin dropdown menus side by side since they don't need the full width of the div. Underneath this, I added a textarea which allows the user to resize the input if they have more things to say.




```
<!-- Form -->
<form action="Contact.html">
  <!-- Enter email-->
  <div class="form-floating mb-3 mt-3">
    <input type="text" class="form-control" id="email" placeholder="Enter email" name="email">
    <label for="email">Email</label>
  </div>
  <!-- Select Message type-->
  <div class="row mb-3 mt-3">
    <div class="col">
      <div class="form-floating">
        <select class="form-select" id="sel1" name="sellist">
          <option>Technical Support</option>
          <option>Bug Report</option>
          <option>Feature Request</option>
          <option>General Inquiry</option>
        </select>
        <label for="sel1" class="form-label">Message Type:</label>
      </div>
    </div>
    <!-- Select plugin-->
    <div class="col">
      <div class="form-floating">
        <select class="form-select" id="sel1" name="sellist">
          <option>Delay</option>
          <option>Distortion</option>
          <option>Flanger</option>
          <option>Compressor</option>
        </select>
        <label for="sel1" class="form-label">Plugin:</label>
      </div>
    </div>
  </div>
  <!-- Enter Message-->
  <div class="form-floating mb-3 mt-3">
    <textarea class="form-control" id="comment" name="text" placeholder="Comment goes here"></textarea>
    <label for="comment">Comments</label>
  </div>
  <!-- Submit Button-->
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

When I programmed the form there is a piece of code at the start which says [action="Contact.html"]. The 'action' defines what should happen when the form is submitted. Currently I have it set to redirect you back to the same webpage. However, in the future I will create a script in php to handle the inputs and store them on the server. This does mean that the form doesn't do anything apart from look pretty, however I do plan to fix this later.

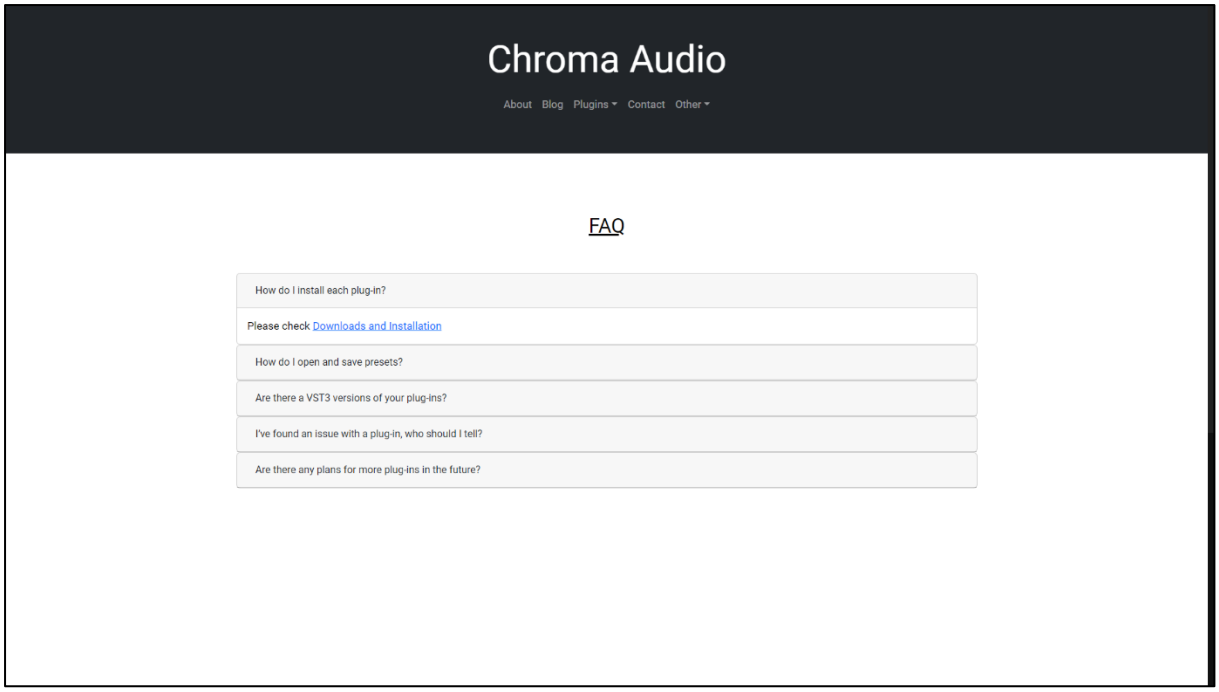
Following this I moved onto the 'other' section of webpages. Most of these aren't that interesting with 'Documentation' having some links in the page. When I created the footer for each webpage, I included two separate links to Downloads and Installation. Although both sections are contained on the same page, I can use HTML's id system to jump to the correct section. I gave the Downloads header the 'Dl' id and the Installation header the 'Ins' id. Then to jump straight to that section when the page is loaded, I just to include #Dl or #Ins at the end of the link at the footer.



```
<!-- Content -->
<div class="container-fluid p-5 text-black filler">
  <div class="row filler">
    <div class="col-sm-2"></div>
    <div class="col-sm-8">
      <div class="container-fluid p-5 text-black filler">
        <!-- Downloads -->
        <h2 id="Dl">Downloads</h2>
        <!-- Text -->
        <p>
          Currently there are two ways of downloading any of my plug-ins. For open-source plug-ins such as the Essentials bundle,
          you can choose to either download the plug-ins directly from the Github repository or through my Gumroad page.
          For paid plug-ins, they will only be available through my Gumroad page. All applicable methods will be displayed on the relevant plug-in page as well.<br>
        </p>
        <!-- Text -->
        <p>
          When downloading source-code, any projects available will be public on my Github page.
          The souce-code will also contain any relevant documentation on how any important classes and functions work for implementation into your own projects.<br><br><br>
        </p>
        <!-- Installation -->
        <h2 id="Ins">Installation</h2>
        <!-- Text -->
        <p>
          To install any of my plug-ins it's as simple as dragging and dropping the plug-in into the relevant folder and running your software.
          I would recommend creating a separate folder within your plug-ins folder to keep everything tidy.<br><br><br>
        </p>
        <h3>Windows</h3>
        <p>
          Since VST2 plug-ins don't have a dedicated folder on windows, you should check with your software first before installing the plug-in.
          However here are some common places where you can find you VST2 plug-ins:
        </p>
        <h5>C:\Program FIles\VSTPlugins<br>C:\Program FIles\Steinberg\VSTPlugins<br>C:\Program FIles\Common Files\VST2<br>C:\Program FIles\Common Files\Steinberg\VST2<br><br><br></h5>
        <!-- Text -->
        <h3>MacOS</h3>
        <p>For MacOS VST2 plug-ins should be moved into this folder:</p>
        <h5>Library/Audio/Plug-ins/VST </h5>
      </div>
    </div>
    <div class="col-sm-2"></div>
  </div>
</div>
```

```
<!-- Contact / FAQ bar -->
<div class="container-fluid p-5 bg-dark text-white">
  <div class="row p-5 text-secondary bg-dark footer">
    <div class="col">
      <h3>Info & FAQ</h3>
      <a class="nav-link" href="">About</a>
      <a class="nav-link" href="Website/Webpages/FAQ">FAQ</a>
      <a class="nav-link" href="Website/Webpages/T&C">Terms & Conditions</a>
    </div>
    <div class="col">
      <h3>Downloads & Installation</h3>
      <a class="nav-link" href="Website/Webpages/Downloads#Dl">Downloads</a>
      <a class="nav-link" href="Website/Webpages/Downloads#Ins">Installation</a>
      <a class="nav-link" href="Website/Webpages/Presets">Presets</a>
    </div>
    <div class="col">
      <h3>Contact</h3>
      <a class="nav-link" href="Website/Webpages/Contact.html">Contact</a>
    </div>
  </div>
</div>
```

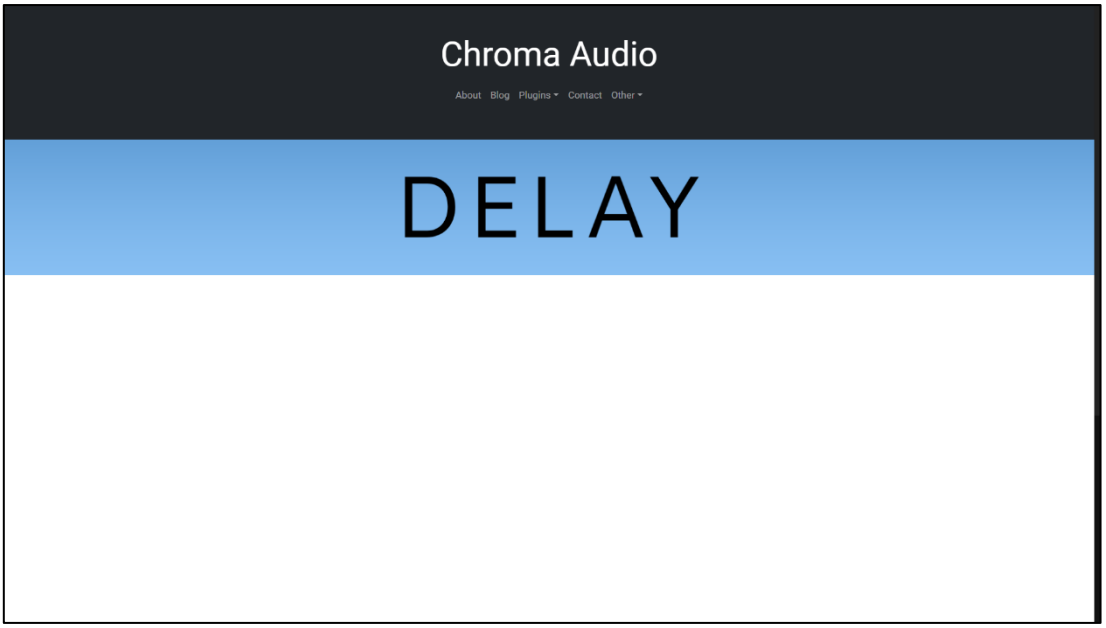
However, since I wanted the website to be more interactive, I used an accordion on the 'FAQ' page which allows the user to open and close the different questions. I populated the different sections with questions asked by stakeholders when they were testing the website.




```
<!-- Content -->
<div class="container-fluid p-5 text-black filler">
  <div class="row filler">
    <div class="col-sm-2"></div>
    <div class="col-sm-8">
      <div class="container-fluid p-5">
        <h2 style="text-align:center">FAQ</h2>
      </div>
      <div class="container-fluid">
        <div id="accordion">
          <div class="card">
            <div class="card-header">
              <a class="btn" data-bs-toggle="collapse" href="#collapseOne">
                How do I install each plug-in?
              </a>
            </div>
            <div id="collapseOne" class="collapse show" data-bs-parent="#accordion">
              <div class="card-body">
                Please check <a href="Downloads.html">Downloads and Installation</a>
              </div>
            </div>
          </div>
          <div class="card">
            <div class="card-header">
              <a class="collapsed btn" data-bs-toggle="collapse" href="#collapseTwo">
                How do I open and save presets?
              </a>
            </div>
            <div id="collapseTwo" class="collapse" data-bs-parent="#accordion">
              <div class="card-body">
                Please check the <a href="Presets.html">Presets</a> page
              </div>
            </div>
          </div>
          <div class="card">
            <div class="card-header">
              <a class="collapsed btn" data-bs-toggle="collapse" href="#collapseThree">
                Are there a VST3 versions of your plug-ins?
              </a>
            </div>
```

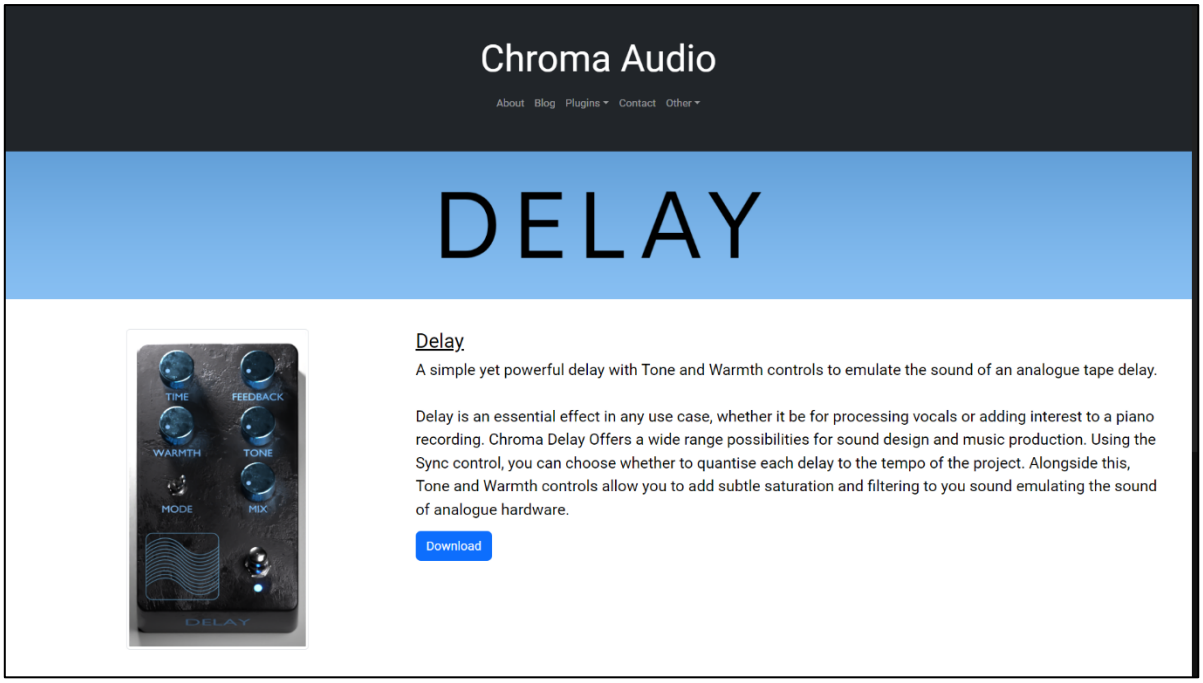
```
                  <div id="collapseThree" class="collapse" data-bs-parent="#accordion">
                    <div class="card-body">
                      Not at the moment, however I am currently working on a fix
                    </div>
                  </div>
                </div>
                <div class="card">
                  <div class="card-header">
                    <a class="collapsed btn" data-bs-toggle="collapse" href="#collapseFour">
                      I've found an issue with a plug-in, who should I tell?
                    </a>
                  </div>
                  <div id="collapseFour" class="collapse" data-bs-parent="#accordion">
                    <div class="card-body">
                      You can contact me through the Please check the <a href="Contact.html">Contact</a> page page
                    </div>
                  </div>
                </div>
                <div class="card">
                  <div class="card-header">
                    <a class="collapsed btn" data-bs-toggle="collapse" href="#collapseFive">
                      Are there any plans for more plug-ins in the future?
                    </a>
                  </div>
                  <div id="collapseFive" class="collapse" data-bs-parent="#accordion">
                    <div class="card-body">
                      Yes! Once the Essentials collection is finished I plan to start development on new plug-ins
                    </div>
                  </div>
                </div>
              </div>
            </div>
          </div>
          <div class="col-sm-2"></div>
        </div>
      </div>
```

Finally, last thing to finish was the 'Plug-in' section. I initially had the idea of a banner with the name of the plug-in beneath the header, followed by a picture of the plug-in and a description of the plug-in. Since I want the banner to stretch across the screen, I have two choices; place it outside of the content div, messing up my filler class. Or I remove the padding on the main content div and create another div for the rest of the content. I chose the second option since it doesn't have the unnecessary space beneath the main content. Adding the image was simple since I just use the <img> tag.




```
61          <!-- Content -->
62          <div class="container-fluid p-0 text-black filler">
63            <!-- Image -->
64            <img src="../../BannerDelay.png" alt="Banner" class="d-block" style="width:100%">
65          </div>
```

Following this, I used the grid system again to create two sections. The one on the left is for the image of the plug-in and it is smaler than the one on the right, which is meant for the content. Beneath the plug-in information I have two download buttons. One takes you to the Github repository where you can choose to download the Plug-in or the source code for the plug-in. The other takes you to my Gumroad page where you can only download the plug-in. I implemented these buttons using Bootstrap using its built-in buttons.




```
<!-- Content -->
<div class="container-fluid p-0 text-black filler">
  <!-- Image -->
  <img src="../../BannerDelay.png" alt="Banner" class="d-block" style="width:100%">

  <!-- Text Div -->
  <div class="container-fluid p-5 text-black">

    <div class="row">
      <div class="col-sm-4">
        <img src="../../DelayGui.png" alt="DelayGui" class="d-block mx-auto d-block img-fluid img-thumbnail" style="width:50%">
      </div>
      <div class="col-sm-8">
        <h2>Delay</h2>
        <p>
          A simple yet powerful delay with Tone and Warmth controls to emulate the sound of an analogue tape delay.
          <br><br>
          Delay is an essential effect in any use case, whether it be for processing vocals or adding interest to a piano recording.
          Chroma Delay Offers a wide range possibilities for sound design and music production.
          Using the Sync control, you can choose whether to quantise each delay to the tempo of the project. Alongside this,
          Tone and Warmth controls allow you to add subtle saturation and filtering to you sound emulating the sound of analogue hardware.
        </p>
        <a href="../Downloads.html" class="btn btn-primary btn-lg">Download</a>
      </div>
    </div>
  </div>
</div>
```

After making a separate page for each plug-in using this template, all the different pages were complete. The only thing to do now is add in any other images and descriptions which I haven't filled in already.

Bibliography:

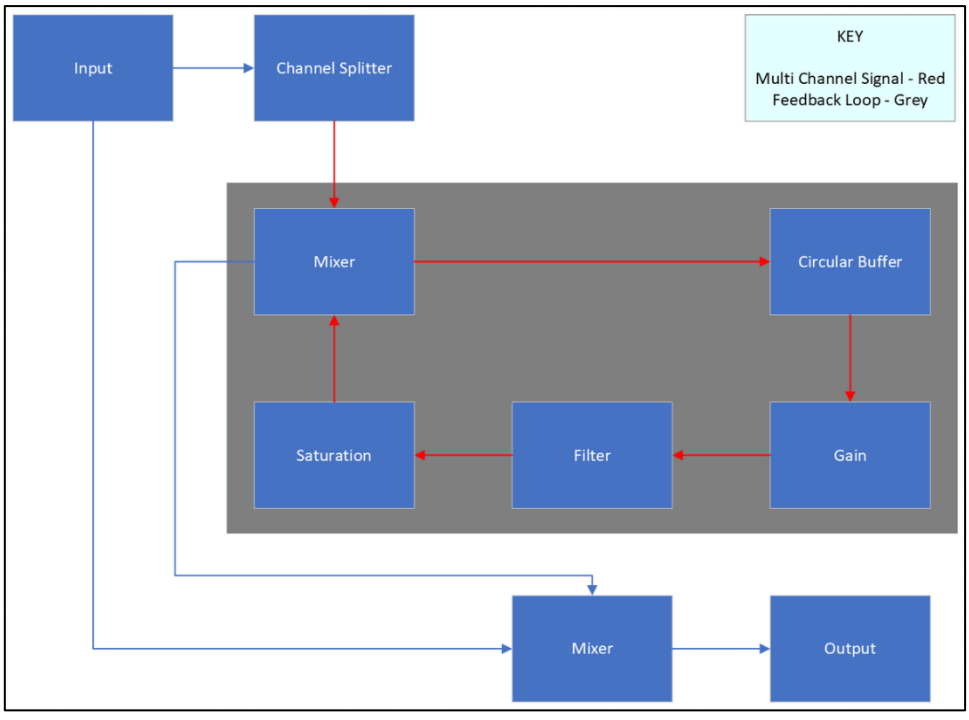
1. Bootstrap, "Bootstrap"
2. W3 Schools, "Bootstrap 5 Tutorial"

Delay Plugin

Design

A delay is an effect where a signal is stored and then played back after a specified amount of time has passed. It is one of the simplest effects to create however it is essential to both commercial music and DSP. Originally, delay was created by making a feedback loop on a strip of magnetic tape. By recording onto the tape whilst using the output as an input creates a feedback loop. Using tape allowed the speed to be changed easily creating different delay times. Due to the fidelity of the equipment, it creates a warmer tone compared to digital delay which is why analogue tape delays are still used today. The amount of feedback could be controlled through applying gain in the feedback loop which also prevented the signal from getting too loud.

My delay will aim to combine both Digital and Tape Delay into one plugin. I will add a warmth and tone parameter which will apply saturation and filtering to the delays allowing for a more analogue sounding tone to the delay. I will base the foundation of my plugin off Luke Zeitlin's design [1] because it shows how to set up a basic delay within WDL-OL without all the features I want to implement. From this I have created a signal diagram detailing the signal flow of the delay with the main elements I want to implement.



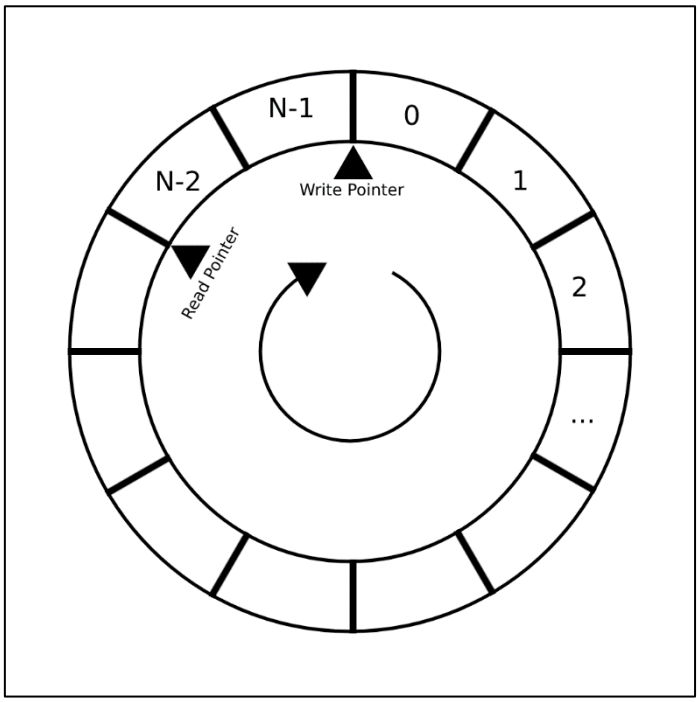
Implementation

Firstly, the input is sent through the processing chain and to a separate mixer, this allows us to blend between the dry and wet signals. Then the signal is split into the left and right channels, each channel is sent to its own circular buffer owing to each buffer working in mono. A circular buffer is an implementation of a queue, which is covered in the A-Level course, I will also go into more detail later about how I am implementing it. After the signal has been delayed, it is sent through a set of modules which both allow the user to control the amount of feedback and whether the plug-in should emulate a tape delay. The signal goes through a gain module which adjusts the amount of feedback in the delay. Then it is sent through a high-shelf filter and a saturator. These modules can be used to emulate the warmth of an analogue tape delay. Finally, this signal is recombined with the input signal before it is sent into the circular buffer creating a feedback loop. This effected signal is also sent to our initial mixer letting us control the amount of delay in the signal.

Circular Buffers

As stated earlier I will be using a circular buffer to create my delay, I will implement this through an array, this allows me to store the last X amount of time in a data structure and read that data at a set offset. Since we will step through the array each sample, we need to figure out how long it needs to be to hold the longest possible delay. To do this we multiply sample rate by longest delay time. This gives us the required length of our array.

When using the buffer, we have a read head and a write head. The write head increments every sample and writes the value of the input to the current index. The read head is a set offset behind the write head and increments every sample. We can change the offset between the two pointers to change the delay time.



Delay Modes

When researching about different delay modes I came across 3 main types; Mono, Stereo and Ping Pong delay. I have chosen to implement two of them, Mono delay and Ping Pong delay. Although I could implement Stereo delay as well, it isn't essential, and I would have to add another control to my GUI contradicting [SC1].

Mono Delay

Mono Delay is the simplest type of delay, each channel shares the same delay time which gives the effect of the signal being repeated at regular intervals just a bit quieter every time. [See Delay Example 1]

Ping Pong Delay

Ping Pong Delay is where the signal is repeated at regular intervals, alternating on the Left and Right channel. To implement this, you can double the delay time on each buffer but offset the right channel by the actual delay time. This makes every delay the desired amount, but it swaps channel on every delay. [See Delay Example 2]

Stereo Delay

Stereo Delay is where the left and right channels have independent delay times. It is easy to implement because you have a separate control for each channel. Since each channel has its own buffer, you can change the offset between the read and write head independently, creating this effect. [See Delay Example 3]

Tempo Sync

An essential part of a delay plugin is the ability to sync it to the host tempo. Without this, it would force the user to find the specific delay they want, making it more complicated to use when compared to other solutions. When I originally thought about how to do this, I needed to find the samples per beat. This relies on both the Beats Per Minute and Sample Rate. This was my working to find the samples per beat.

$$\frac{\text{Beats}}{\text{Minute}} \times \frac{1}{60} = \frac{\text{Beats}}{\text{Sec}}$$

$$\frac{\text{Sec}}{\text{Beat}} \times \frac{\text{Fs}}{\text{Sec}} = \frac{\text{Samples}}{\text{Beat}}$$

When implementing this I can use the equation Samples/Beat = (SampleRate / (Tempo / 60)). To get different divisions of this beat I can multiply by 0.5, 0.75, 1 etc.

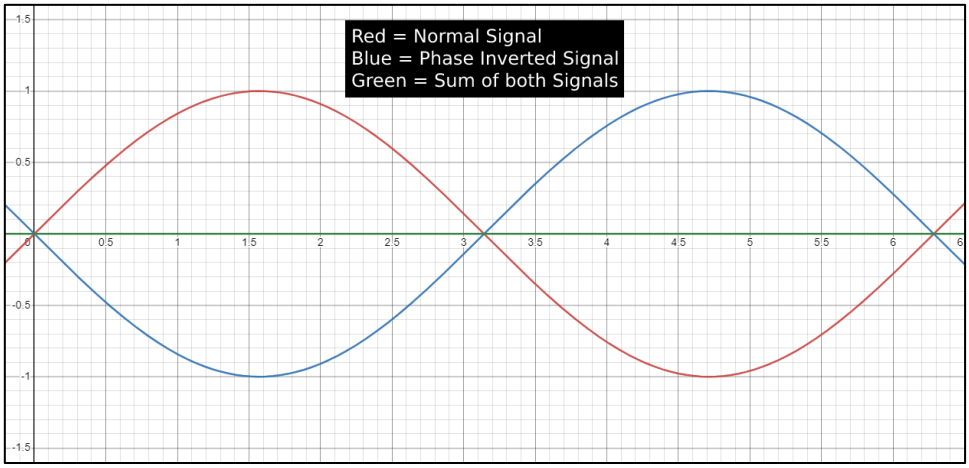
Tape Emulation

To imitate the warmth of Tape Delay, I am using both a filter and a saturator. This warmth is created through the fidelity of the equipment being used to record the tape. The filter muffles the sound and then the saturator creates the warmer tone emulating this effect.

Digital Filter

In a general sense, a filter is a medium which modifies the input when it is passed through it. In the real world this could be holding up a sheet of fabric over a speaker to muffle the sound or a set of electronics which modify an analogue signal in a synthesiser. A digital filter achieves the same goal; however, it operates on a digital domain. A digital filter can do anything an analogue filter can do to an arbitrary degree of accuracy.

Although there are many ways of implementing a digital filter, I will outline two methods. Firstly, you can create filtering through phase cancelation. If you sum two signals where the second is identical to the original however its phase has been inverted, it will produce silence.

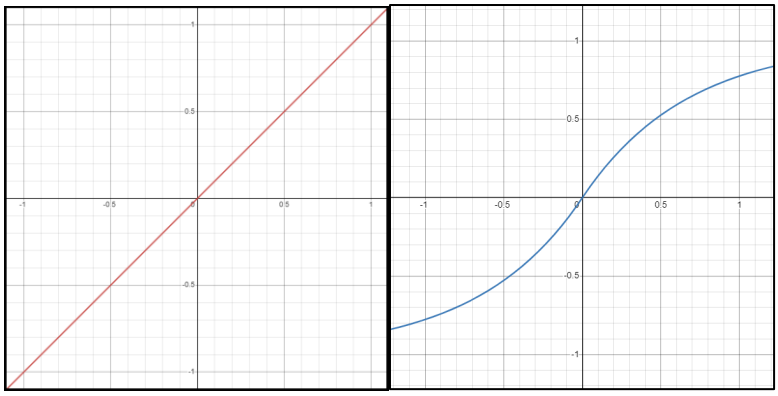


Although it seems like a straightforward approach, you would firstly need to extract the frequency band you want to attenuate. Which would involve Fourier transforms making the whole thing way too complicated. Another approach is to use either Finite Impulse Response (FIR) or Infinite Impulse Response (IIR) filters. Although I don't fully understand the maths behind these filters, it is the most popular approach to implementing a filter, so I am going to use it. Since I want my filters to be quick and responsive, I will be implementing an IIR filter since it is much faster than a FIR filter at the cost of some accuracy which in my use case is negligible. After some research I will use a Biquad filter since it's an IIR filter and it allows me to create a High-pass filter. I won't be implementing my own however I will use an implementation created by Nigel Redmon [2].

On guitar pedals, the tone control attenuates the high-frequency content of a signal, this is desirable when creating tape emulation since a physical often provides a subtle amount of filtering when being written to and read from.

Saturation

Although I will cover this in-depth later, I'll give a brief overview of what saturation is and how it works. Saturation is a form of distortion which is less aggressive and focused on creating a warmer tone through adding harmonics to the signal. Since I am trying to emulate the effect of a physical tape it makes logical sense to emulate tape saturation. To create this saturation, I will make use of a transfer curve which maps a set of inputs to a set of outputs. I will be using the transfer function suggested by Matt L in response to a question on the DSP Stack Exchange [3]. Since it tends towards –1 and 1 as you approach positive and negative infinity.



Default Curve **Transfer Curve**

The warmth control will add gain to the signal, allowing the effect of the transfer curve to become more noticeable. Although it isn't subtle on a sine wave, when applied to a guitar it creates a slight amount of warmth, hence the name.

Normal sine wave: [See Delay Example 4]

Sine wave with saturation: [See Delay Example 5]

Circular buffer class

Since the circular buffer algorithm can be abstracted to a separate class I shall do so in accordance with [SC7_A]. Here I will outline the different functions and variables that I'll need to create the class and their purpose.

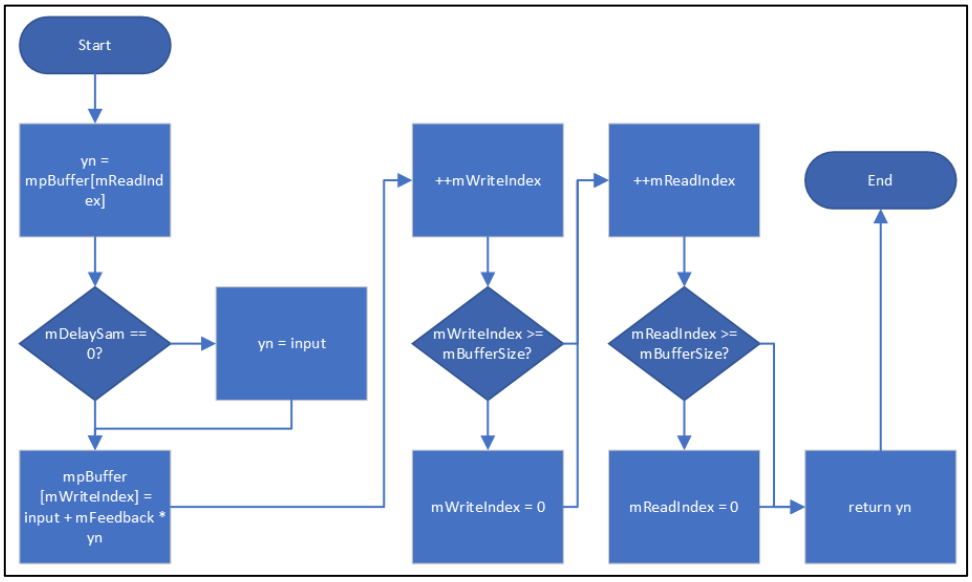
Functions:

- Destroy – Used to delete the buffer when the plug-in is unloaded from memory.
- cookVars – Used to set variables and create the offset between the read pointer and the write pointer.
- resetDelay – Initialises the buffer by setting every value in it to 0 and sets the read and write pointers to 0.
- Reset – This is called when the sample rate changes so we can delete the old buffer and create a new buffer. We then call resetDelay cookVars.
- ProcessSample – This reads from and writes to the buffer, increaments both pointers and returns the delayed sample.

Variables:

- mDelaySam – double, stores the length of the delay in samples.
- mFeedback – double, stores the amount of feedback.
- SampleRate – double, stores the current sample rate.
- mBuffer – double, stores the buffer array where each value in the array is a double.
- mReadIndex – int, stores the value of the read pointer.
- mWriteIndex – int, stores the value of the write pointer.
- mBufferSize – int, stores the size of the buffer.

I'll explain the PrcessSample function since the others and much simpler. I have created a flow chart to show the algorithm works.



We start off by reading from the buffer and setting its value to an arbitrary variable yn. After that we check if there is a delay, there is no difference between the read pointer and the write pointer we set yn to the input. Once we've read from the buffer, we can write to it. By doing input + mFeedback * yn we can both write the input to the buffer while adding the feedback at the same time. After this we increment each pointer, if they are the same size or greater than the size of the whole buffer, we set them back to the start allowing us to loop through the buffer. Finally, we return yn and the sample has been processed.

Development

To start things off, I created a new project and began to follow Luke Zeitlin's delay implementation. Since his implementation was programmed for iPlug1 I had to change and adapt a few things so it works in iPlug2 however all the DSP remained the same. In iPlug1 there are two functions which he uses in his implementation which aren't initially present in iPlug2; Reset and OnParamChange. To use these functions, you define them in your header and in your main program however in the header you add 'override' at the end which allows you to edit the functions, another thing to note is that Reset has been changed to OnReset. I also set up some initial variables that I will need later.

```
#pragma once
#include "IPlug_include_in_plug_hdr.h"

const int kNumPresets = 1;

enum EParams
{
  kDelayMs = 0,
  kFeedback,
  kMix,
  kNumParams
};

using namespace iplug;
using namespace igraphics;

class Delay final : public Plugin
{
public:
  Delay(const InstanceInfo& info);

private:
  double mDelaySam;
  double mFeedback;
  double mWet;

#if IPLUG_DSP // http://bit.ly/2S64BDd
  void OnParamChange(int paramIdx) override;
  void OnReset() override;
  void cookVars();
  void ProcessBlock(sample** inputs, sample** outputs, int nFrames) override;
#endif
};
```

After I did this, I set up the cookVars function which assigns the values of the controls to the variables I just created, and I call this function in OnParamChange and OnReset. I call it in OnParamChance since a control has changes, so I need to update my variables. I also call it in OnReset since when the plug-in is reset, controls can change meaning their variables wouldn't be updated.

```
void Delay::cookVars()
{
  mDelaySam = GetParam(kDelayMs)->Value() * GetSampleRate() / 1000.0;
  mFeedback = GetParam(kFeedback)->Value() / 100.0;
  mWet = GetParam(kMix)->Value() / 100.0;
}

void Delay::OnParamChange(int paramIdx)
{
  cookVars();
}

void Delay::OnReset()
{
  cookVars();
}
```

After all this setup I can finally implement the circular buffer. I created mpBuffer and assigned NULL to it since I both don't know the sample rate yet and the size could change during processing. By putting double before it since I want to store data with a double type. After this I created three integer variables which are self-explanatory.

```
private:
  double mDelaySam;
  double mFeedback;
  double mWet;

  double* mpBuffer = NULL;
  int mReadIndex = 0;
  int mWriteIndex = 0;
  int mBufferSize = 0;
```

Now I've sorted all the backend for the buffer I can finally start implementing it. In the OnReset function I define the buffer size, check if there already is a buffer and delete it if there is one and create a new buffer. I set mBufferSize to 2 * Sample Rate since I want my longest delay to be 2 seconds long. I also added some code to delete the buffer in the destructor when the plug-in is closed just encase the host software doesn't clear its memory.

```
void Delay::OnReset()
{
  mBufferSize = 2 * GetSampleRate();
  if (mpBuffer)
  {
    delete[] mpBuffer;
  }
  mpBuffer = new double[mBufferSize];
  cookVars();
}
```

```
Delay::~Delay() {
  if (mpBuffer)
  {
    delete[] mpBuffer;
  }
}
```

I then defined a new function resetDelay, which fills the whole buffer with 0's and resets the read and write buffer to 0. If we didn't do this, on our first pass through the buffer we would just hear random noise until we write over it. This just ensures that we will hear silence for the first pass. I also called this function in the OnReset function but before cookVars.

```
void Delay::resetDelay()
{
  if (mpBuffer)
  {
    memset(mpBuffer, 0, mBufferSize * sizeof(double));
  }

  mWriteIndex = 0;
  mReadIndex = 0;
}
```

```
void Delay::OnReset()
{
  mBufferSize = 2 * GetSampleRate();
  if (mpBuffer)
  {
    delete[] mpBuffer;
  }
  mpBuffer = new double[mBufferSize];
  resetDelay();
  cookVars();
}
```

Since I'll increment the write index every sample with the read index following behind by a set delay. To set up this delay between the two pointers, I just set my read index to the write index minus my delay in samples. If the read index becomes negative because the write index has just wrapped around the buffer, I just add the length of the buffer to it, so it remains in the right place.

```cpp
void Delay::cookVars()
{
  mDelaySam = GetParam(kDelayMs)->Value() * GetSampleRate() / 1000.0;
  mFeedback = GetParam(kFeedback)->Value() / 100.0;
  mWet = GetParam(kMix)->Value() / 100.0;

  mReadIndex = mWriteIndex - (int)mDelaySam;
  if (mReadIndex < 0)
  {
    mReadIndex += mBufferSize;
  }
}
```

With all the backend done I can finally start processing the audio. Since iPlug2's default processing function is slightly different to iPlug1 I chose to use iPlug1's processing function since both samples are processed and outputted on the same sample. When I was working on my distortion plug-in, I encountered a bug where the left and right channels were delayed by a sample owing to this so to ensure that it works, I went with iPlug1's implementation.

```cpp
void Delay::ProcessBlock(sample** inputs, sample** outputs, int nFrames)
{
  const int nChans = NOutChansConnected();

  double* in1 = inputs[0];
  double* in2 = inputs[1];
  double* out1 = outputs[0];
  double* out2 = outputs[1];

  for (int s = 0; s < nFrames; ++s, ++in1, ++in2, ++out1, ++out2)
  {
    // Read from buffer
    double yn = mpBuffer[mReadIndex];

    if (mDelaySam == 0)
    {
      yn = *in1;
    }

    // Write to buffer
    mpBuffer[mWriteIndex] = *in1 + mFeedback * yn;

    // Lerp dry and wet signals
    *out1 = (mWet * yn + (1 - mWet) * *in1);
```

```
        // Increment buffers
        ++mWriteIndex;
        if (mWriteIndex >= mBufferSize)
        {
            mWriteIndex = 0;
        }
        ++mReadIndex;
        if (mReadIndex >= mBufferSize)
        {
            mReadIndex = 0;
        }

        // Coppy left to right
        *out2 = *out1;
    }
}
```

Firstly, I created a new variable yn which stores the delayed output. Then if there is no delay, we set yn to the input. Following this we write to the delay buffer; we write our current input and delayed signal which is multiplied by mFeedback which allows us to control how many delays we hear. Then to create the Mix control we can use linear interpolation to smoothly blend between the unprocessed input signal and the delyed signal.

Now after all the processing we can deal with the pointers. Each sample we increment each pointer by one and check if the pointer is greater than or equal to the buffer size. If so, we set it to 0 which wraps the pointer back to the start of the buffer. Finally, since this is only working in mono for the minute, we duplicate the left output to the right output.

After building the plug-in and following some initial testing everything was working as intended. Since I might need to use this circular buffer code when programming my flanger I decided to abstract it to a class in accordance with [SC7_A]. This will also make it easier to implement stereo delay since I just create two instances of this circular buffer class. After moving everything over here is the code for the CircularBuffer class. I also created a new variable SampleRate, since I can't call the GetSampleRate function in this class I have to pass it in.

```cpp
#include "cmath"
#pragma once
class CircularBuffer
{
public:
    void Destroy();
    void cookVars(double delay, double feedback);
    void resetDelay();
    void Reset(double samplerate, double delay, double feedback);
    double ProcessSample(double input);

private:
    double mDelaySam;
    double mFeedback;
    double SampleRate;

    double* mpBuffer = NULL;
    int mReadIndex = 0;
    int mWriteIndex = 0;
    int mBufferSize = 0;
};
```

```cpp
#include "CircularBuffer.h"
#include <string.h>

void CircularBuffer::Destroy()
{
    if (mpBuffer)
    {
        delete[] mpBuffer;
    }
}

void CircularBuffer::cookVars(double delay, double feedback) {
    mDelaySam = delay * SampleRate / 1000.0;
    mFeedback = feedback / 100.0;

    mReadIndex = mWriteIndex - (int)mDelaySam;
    if (mReadIndex < 0)
    {
        mReadIndex += mBufferSize;
    }
}

void CircularBuffer::resetDelay()
{
    if (mpBuffer)
    {
        memset(mpBuffer, 0, mBufferSize * sizeof(double));
    }

    mWriteIndex = 0;
    mReadIndex = 0;
}
```

```cpp
void CircularBuffer::Reset(double samplerate, double delay, double feedback)
{
  SampleRate = samplerate;
  mBufferSize = 2 * SampleRate;
  if (mpBuffer)
  {
    delete[] mpBuffer;
  }

  mpBuffer = new double[mBufferSize];
  resetDelay();
  cookVars(delay, feedback);
}
```

```cpp
double CircularBuffer::ProcessSample(double input)
{
  // Read from buffer
  double yn = mpBuffer[mReadIndex];

  if (mDelaySam == 0)
  {
    yn = input;
  }

  // Write to buffer
  mpBuffer[mWriteIndex] = input + mFeedback * yn;

  // Increment buffers
  ++mWriteIndex;
  if (mWriteIndex >= mBufferSize)
  {
    mWriteIndex = 0;
  }
  ++mReadIndex;
  if (mReadIndex >= mBufferSize)
  {
    mReadIndex = 0;
  }

  // Return output
  return yn;
}
```

Since I'll need to modify this class to implement all the features I want for my delay, and I want to use the class as it is for my flanger, I created a copy of this class and renamed it to ProcessDelay. That way I can have a template for a circular buffer whenever I need to use one. Following on from this, I decided to implement the sync function. Since I want eight different synced timings, I divided my maximum delay time (2000) by eight which gave me 250. Then I just check if DelayMs < 250 in increments of 250, essentially giving me eight different sections on the control.

```
// Sync Calculation
if (mSync == 0)
{
  mDelaySam = mDelayMs * SampleRate / 1000.0;
}
else
{
  if (mDelayMs < 250.0)
  {
    mDelaySam = 0.25 * (SampleRate / (mTempo / 60.0));
  }
  else if (mDelayMs < 500.0)
  {
    mDelaySam = 0.5 * (SampleRate / (mTempo / 60.0));
  }
  else if (mDelayMs < 750.0)
  {
    mDelaySam = 0.75 * (SampleRate / (mTempo / 60.0));
  }
  else if (mDelayMs < 1000.0)
  {
    mDelaySam = 1.0 * (SampleRate / (mTempo / 60.0));
  }
  else if (mDelayMs < 1250.0)
  {
    mDelaySam = 1.25 * (SampleRate / (mTempo / 60.0));
  }
  else if (mDelayMs < 1500.0)
  {
    mDelaySam = 1.5 * (SampleRate / (mTempo / 60.0));
  }
  else if (mDelayMs < 1750.0)
  {
    mDelaySam = 2.0 * (SampleRate / (mTempo / 60.0));
  }
  else
  {
    mDelaySam = 4.0 * (SampleRate / (mTempo / 60.0));
  }
}
```

To do this I had to create two new variables. mTempo stores the current tempo and mSync is used to check if we want tempo sync. When I initially implemented this, I had an issue where I only got 1 delay time when I turned on sync. This happened because I forgot to put brackets around (mTempo / 60).

Finally, the last delay related feature I wanted to implement was the Ping-Pong mode. Since my class only creates one buffer and I'll have to instance the class twice. This means that I'll have to make a workaround to implement the Ping-Pong mode. I created a two new variables mPT which stores which channel the channel the buffer is supposed to be (left or right) and mPingPong which if we want Ping-Pong delay.

```
// Ping Pong
if (mPingPong == 1 && mDelayMs != 0.)
{
  if (mPT == 0)
  {
    mReadIndex = mWriteIndex - (2 * (int)mDelaySam) + (int)mDelaySam;
  }
  else
  {
    mReadIndex = mWriteIndex - (2 * (int)mDelaySam);
  }
}
else
{
  mReadIndex = mWriteIndex - (int)mDelaySam;
}
```

After I finished this I thought I would tidy up the class since I was having to pass a lot of variables into cookVars at this stage and it would be much simpler to have a SetVars function. Here is the new SetVars code.

```
void ProcessDelay::SetVars(double delay, double feedback, int sync, int pingPong, double tempo, int pt)
{
  mDelayMs = delay;
  mFeedback = feedback / 100.;
  mSync = sync;
  mPingPong = pingPong;
  mTempo = tempo;
  mPT = pt;
}
```

I won't put all the code for the rest of the class here, but it is available on my Github page if you want to take a look. Now the final feature I want to implement is the tape emulation. I created a new function in my main program called ToneWarmth which first filters the signal then adds saturation. To implement the filter, I included the header and created an instance of the filter class.

```
#include "DelayPlugin.h"
#include "IPlug_include_in_plug_src.h"
#include "IControls.h"
#include "ProcessDelay.h"
#include "Biquad.h"

// Instance imported classes
ProcessDelay BL;
ProcessDelay BR;
Biquad* hsFilter = new Biquad();
```

Since the filter relies on the samplerate of the project I put a setup function for the filter in the OnReset function. This is also where I defined the Cut-off and the Q.

```
void DelayPlugin::OnReset()
{

  // Calls buffer's reset function
  BL.Reset(GetSampleRate());
  BR.Reset(GetSampleRate());
  hsFilter->setBiquad(bq_type_highshelf, 3500 / GetSampleRate(), 0.707, 0);
}
```

To implement the saturation, I just used Mat L's implementation. However, I added in the warmth control since I could use that to increase the gain of the signal, adding more saturation. Putting all of this together here is the ToneWarmth function.

```
// Tone Warmth Processing
double DelayPlugin::ToneWarmth(double signal, double warmth)
{

  // Filter Processing
  signal = hsFilter->process(signal);

  // Saturation
  if (warmth > 1.0)
  {
    if (signal > 0)
    {

      signal = 1 - exp(-1 * signal * warmth);
    }
    else
    {

      signal = -1 + exp(signal * warmth);
    }
  }

  return signal;
}
```

By moving all the processing to separate classes and functions it heavily simplifies the processing block. After some initial testing whenever I was listening to the fully wet signal it was way louder than it should be. This was because the mix control goes from 0-100 in the gui however for my linear interpolation algorithm this needs to be between 0-1. To fix this I just divided the mix by 100 when I defined it earlier in the processing block. After this everything seemed to work fine so I finished up the GUI.

```cpp
#if IPLUG_DSP
void DelayPlugin::ProcessBlock(sample** inputs, sample** outputs, int nFrames)
{
  const double mix = GetParam(kMix)->Value() / 100.0;
  const bool bypass = GetParam(kBypass)->Value();
  const double tone = GetParam(kTone)->Value();
  const double warmth = GetParam(kWarmth)->Value();

  const int nChans = NOutChansConnected();

  double* in1 = inputs[0];
  double* in2 = inputs[1];
  double* out1 = outputs[0];
  double* out2 = outputs[1];

  for (int s = 0; s < nFrames; s++, ++in1, ++in2, ++out1, ++out2)
  {
    // Read from buffers
    double LPS = BL.ProcessSample(*in1);
    double RPS = BR.ProcessSample(*in2);

    if (bypass == 1)
    {
      // Tone + Warmth
      hsFilter->setPeakGain(tone);
      LPS = ToneWarmth(LPS, warmth);
      RPS = ToneWarmth(RPS, warmth);

      // Mix
      *out1 = (mix * LPS + (1 - mix) * *in1);
      *out2 = (mix * RPS + (1 - mix) * *in2);
    }
    else
    {
      *out1 = *in1;
      *out2 = *in2;
    }
  }
}
```

When creating the GUI, I modified my plug-in template and rendered out all the imaged. After stitching them together and importing them into iPlug2 this was the result.

Here is all the code for the delay code:

```h
#pragma once

#include "IPlug_include_in_plug_hdr.h"

const int kNumPresets = 1;

enum EParams
{
  kDelayMs = 0,
  kFeedback,
  kMix,
  kTone,
  kWarmth,
  kBypass,
  kSync,
  kPingPong,
  kNumParams
};

using namespace iplug;
using namespace igraphics;

class DelayPlugin final : public Plugin
{
public:
  DelayPlugin(const InstanceInfo& info);
  ~DelayPlugin();

private:


#if IPLUG_DSP // http://bit.ly/2S64BDd
  void OnParamChange(int paramIdx) override;
  void OnReset() override;
  double ToneWarmth(double signal, double warmth);
  void ProcessBlock(sample** inputs, sample** outputs, int nFrames) override;
#endif
};
```

```cpp
#include "DelayPlugin.h"
#include "IPlug_include_in_plug_src.h"
#include "IControls.h"
#include "ProcessDelay.h"
#include "Biquad.h"

// Instance imported classes
ProcessDelay BL;
ProcessDelay BR;
Biquad* hsFilter = new Biquad();

DelayPlugin::DelayPlugin(const InstanceInfo& info)
: Plugin(info, MakeConfig(kNumParams, kNumPresets))
{
  GetParam(kDelayMs)->InitDouble("Delay", 10., 0.1, 2000., 0.01, "Ms");
  GetParam(kFeedback)->InitDouble("Feedback", 50., 0., 100., 0.01, "%");
  GetParam(kMix)->InitDouble("Mix", 100., 0., 100., 0.01, "%");
  GetParam(kTone)->InitDouble("Tone", 0.0, -4.0, 0.0, 0.01, "");
  GetParam(kWarmth)->InitDouble("Warmth", 1.0, 1.0, 3.0, 0.01, "");
  GetParam(kBypass)->InitInt("Bypass", 1, 0, 1, "");
  GetParam(kSync)->InitInt("Sync", 0, 0, 1, "");
  GetParam(kPingPong)->InitInt("PingPong", 0, 0, 1, "");

#if IPLUG_EDITOR // http://bit.ly/2S64BDd
  mMakeGraphicsFunc = [&]() {
    return MakeGraphics(*this, PLUG_WIDTH, PLUG_HEIGHT, PLUG_FPS, GetScaleForScreen(PLUG_WIDTH, PLUG_HEIGHT));
  };

  mLayoutFunc = [&](IGraphics* pGraphics) {
    pGraphics->AttachCornerResizer(EUIResizerMode::Scale, false);
    pGraphics->AttachBackground(GUI_FN);
    pGraphics->LoadFont("Roboto-Regular", ROBOTO_FN);
```

```
    const IBitmap time = pGraphics->LoadBitmap(TIME_FN, 128);
    const IBitmap feedback = pGraphics->LoadBitmap(FEEDBACK_FN, 128);
    const IBitmap warmth = pGraphics->LoadBitmap(WARMTH_FN, 128);
    const IBitmap tone = pGraphics->LoadBitmap(TONE_FN, 128);
    const IBitmap mix = pGraphics->LoadBitmap(MIX_FN, 128);
    const IBitmap bypass = pGraphics->LoadBitmap(BYPASS_FN, 2);
    const IBitmap mode = pGraphics->LoadBitmap(MODE_FN, 2);
    const IBitmap sync = pGraphics->LoadBitmap(SYNC_FN, 2);

    pGraphics->AttachControl(new IBKnobControl(55, 33, time, kDelayMs));
    pGraphics->AttachControl(new IBKnobControl(204, 33, feedback, kFeedback));
    pGraphics->AttachControl(new IBKnobControl(204, 239, mix, kMix));
    pGraphics->AttachControl(new IBKnobControl(204, 136, tone, kTone));
    pGraphics->AttachControl(new IBKnobControl(55, 136, warmth, kWarmth));
    pGraphics->AttachControl(new IBSwitchControl(211, 392, bypass, kBypass));
    pGraphics->AttachControl(new IBSwitchControl(104, 254, sync, kSync));
    pGraphics->AttachControl(new IBSwitchControl(35, 254, mode, kPingPong));

  };
#endif
}

#if IPLUG_DSP
void DelayPlugin::ProcessBlock(sample** inputs, sample** outputs, int nFrames)
{
  const double mix = GetParam(kMix)->Value() / 100.0;
  const bool bypass = GetParam(kBypass)->Value();
  const double tone = GetParam(kTone)->Value();
  const double warmth = GetParam(kWarmth)->Value();

  const int nChans = NOutChansConnected();

  double* in1 = inputs[0];
  double* in2 = inputs[1];
  double* out1 = outputs[0];
  double* out2 = outputs[1];
```

```
  for (int s = 0; s < nFrames; s++, ++in1, ++in2, ++out1, ++out2)
  {
    // Read from buffers
    double LPS = BL.ProcessSample(*in1);
    double RPS = BR.ProcessSample(*in2);

    if (bypass == 1)
    {
      // Tone + Warmth
      hsFilter->setPeakGain(tone);
      LPS = ToneWarmth(LPS, warmth);
      RPS = ToneWarmth(RPS, warmth);

      // Mix
      *out1 = (mix * LPS + (1 - mix) * *in1);
      *out2 = (mix * RPS + (1 - mix) * *in2);
    }
    else
    {
      *out1 = *in1;
      *out2 = *in2;
    }
  }
}

// Tone Warmth Processing
double DelayPlugin::ToneWarmth(double signal, double warmth)
{

  // Filter Processing
  signal = hsFilter->process(signal);

  // Saturation
  if (warmth > 1.0)
  {
    if (signal > 0)
    {
```

```
        signal = 1 - exp(-1 * signal * warmth);
      }
      else
      {

        signal = -1 + exp(signal * warmth);
      }
    }

    return signal;
}

void DelayPlugin::OnReset()
{

    // Calls buffer's reset function
    BL.Reset(GetSampleRate());
    BR.Reset(GetSampleRate());
    hsFilter->setBiquad(bq_type_highshelf, 3500 / GetSampleRate(), 0.707, 0);
}

void DelayPlugin::OnParamChange(int paramIdx)
{
    double delayMs = GetParam(kDelayMs)->Value();
    double feedback = GetParam(kFeedback)->Value();
    int sync = GetParam(kSync)->Value();
    int pingPong = GetParam(kPingPong)->Value();

    // Cook vars for buffers
    BL.SetVars(delayMs, feedback, sync, pingPong, GetTempo(), 0);
    BR.SetVars(delayMs, feedback, sync, pingPong, GetTempo(), 1);

    BL.cookVars();
    BR.cookVars();
}
#endif
```

```
DelayPlugin::~DelayPlugin()
{
    // Delete both buffers by calling their destructors
    BL.Destroy();
    BR.Destroy();
}
```

Here is all the code for the ProcessDelay class:

```h
#include "cmath"
#pragma once
class ProcessDelay
{
public:
  void Destroy();
  void cookVars();
  void resetDelay();
  void Reset(double samplerate);
  void SetVars(double delay, double feedback, int sync, int pingPong, double tempo, int pt);
  double ProcessSample(double input);

private:
  double mDelaySam;
  double mDelayMs;
  double mFeedback;
  double SampleRate;
  int mSync;
  int mPingPong;
  double mTempo;
  int mPT;

  double* mpBuffer = NULL;
  int mReadIndex = 0;
  int mWriteIndex = 0;
  int mBufferSize = 0;
};
```

```cpp
#include "ProcessDelay.h"
#include <string.h>

void ProcessDelay::Destroy()
{
  if (mpBuffer)
  {
    delete[] mpBuffer;
  }
}

void ProcessDelay::cookVars()
{
  // Sync Calculation
  mTempo = 150.0;
  if (mSync == 0)
  {
    mDelaySam = mDelayMs * SampleRate / 1000.0;
  }
  else
  {
    if (mDelayMs < 250.0)
    {
      mDelaySam = 0.25 * (SampleRate / (mTempo / 60.0));
    }
    else if (mDelayMs < 500.0)
    {
      mDelaySam = 0.5 * (SampleRate / (mTempo / 60.0));
    }
    else if (mDelayMs < 750.0)
    {
      mDelaySam = 0.75 * (SampleRate / (mTempo / 60.0));
    }
    else if (mDelayMs < 1000.0)
    {
      mDelaySam = 1.0 * (SampleRate / (mTempo / 60.0));
    }
    else if (mDelayMs < 1250.0)
    {
```

```
        mDelaySam = 1.25 * (SampleRate / (mTempo / 60.0));
      }
      else if (mDelayMs < 1500.0)
      {
        mDelaySam = 1.5 * (SampleRate / (mTempo / 60.0));
      }
      else if (mDelayMs < 1750.0)
      {
        mDelaySam = 2.0 * (SampleRate / (mTempo / 60.0));
      }
      else
      {
        mDelaySam = 4.0 * (SampleRate / (mTempo / 60.0));
      }
    }

    // Ping Pong
    if (mPingPong == 1 && mDelayMs != 0.)
    {
      if (mPT == 0)
      {
        mReadIndex = mWriteIndex - (2 * (int)mDelaySam) + (int)mDelaySam;
      }
      else
      {
        mReadIndex = mWriteIndex - (2 * (int)mDelaySam);
      }
    }
    else
    {
      mReadIndex = mWriteIndex - (int)mDelaySam;
    }
```

```
  if (mReadIndex < 0)
  {
    mReadIndex += mBufferSize;
  }
}

void ProcessDelay::resetDelay()
{
  if (mpBuffer)
  {
    memset(mpBuffer, 0, mBufferSize * sizeof(double));
  }

  mWriteIndex = 0;
  mReadIndex = 0;
}

void ProcessDelay::Reset(double samplerate)
{
  SampleRate = samplerate;
  mBufferSize = 2 * SampleRate;
  if (mpBuffer)
  {
    delete[] mpBuffer;
  }

  mpBuffer = new double[mBufferSize];
  resetDelay();
  cookVars();
}

void ProcessDelay::SetVars(double delay, double feedback, int sync, int pingPong, double tempo, int pt)
{
  mDelayMs = delay;
  mFeedback = feedback / 100.;
  mSync = sync;
  mPingPong = pingPong;
  mTempo = tempo;
  mPT = pt;
}
```

```cpp
double ProcessDelay::ProcessSample(double input)
{
  // first we read our delayed output
  double yn = mpBuffer[mReadIndex];

  // if the delay is 0 samples we just feed it the input
  if (mDelaySam == 0)
  {
    yn = input;
  }

  // now write to our delay buffer
  mpBuffer[mWriteIndex] = input + mFeedback * yn;

  // increment the write index, wrapping if it goes out of bounds.
  mWriteIndex++;
  if (mWriteIndex >= mBufferSize)
  {
    mWriteIndex = 0;
  }

  // same with the read index
  mReadIndex++;
  if (mReadIndex >= mBufferSize)
  {
    mReadIndex = 0;
  }

  // because we are working in mono we'll just copy the left output to the right output.
  return yn;
}
```

As stated before, all the code is available on my Github profile and may have been updated since the creation of this document.

Bibliography:

1. Luke Zeitlin, “How Delays Work”
2. Ear Level Engineering, “Biquad C++”
3. Matt L, "Digital Distortion Effect Algorithm"

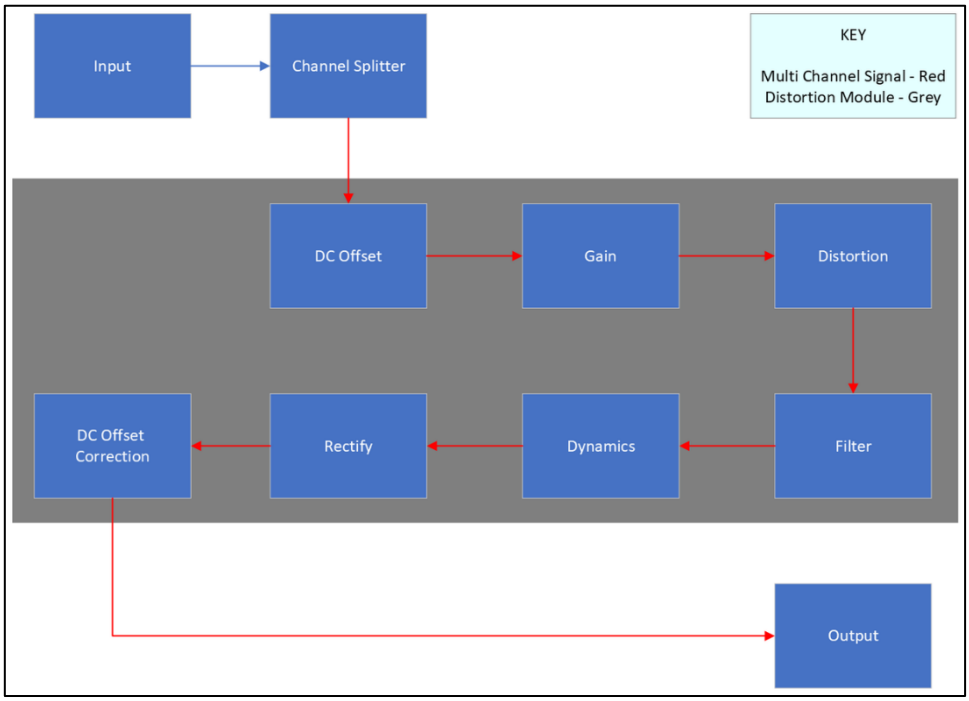
Distortion Plugin

Design

Distortion is an effect which adds a ‘warmer’, ‘fuzzy’ tone to the inputted signal. We can hear distortion on almost every modern song owing to the range of tones it creates. Its uses range from creating a warmer tone, to completely distorting the signal beyond recognition. I plan to implement three distinct types of distortion into my plugin. Hard Clipping, Overdrive and Wavefolding. This will enable the plugin to have more versatility and allow the user to experiment with their own sound design.

The earliest types of distortion were created on Vacuum Tube Amplifiers when the volume was turned up too high. To prevent the vacuum tubes from breaking, they compress the inputted signal creating the first type of guitar distortion, Overdrive. Other types of distortion were originally created through damaging the speaker cone of an amplifier. This created more distorted tones when compared to the original overdrive. However, most people didn’t want to destroy their amplifiers to achieve this effect. Maestro created the first distortion pedal, the Fuzz-Tone which replicated a faulty connection on a mixing desk creating a distorted tone.

I have designed a signal diagram which shows the different modules I need to program to create my distortion. The Modules in the grey box are all contained within one function which allows me to reuse the code in future plugins.

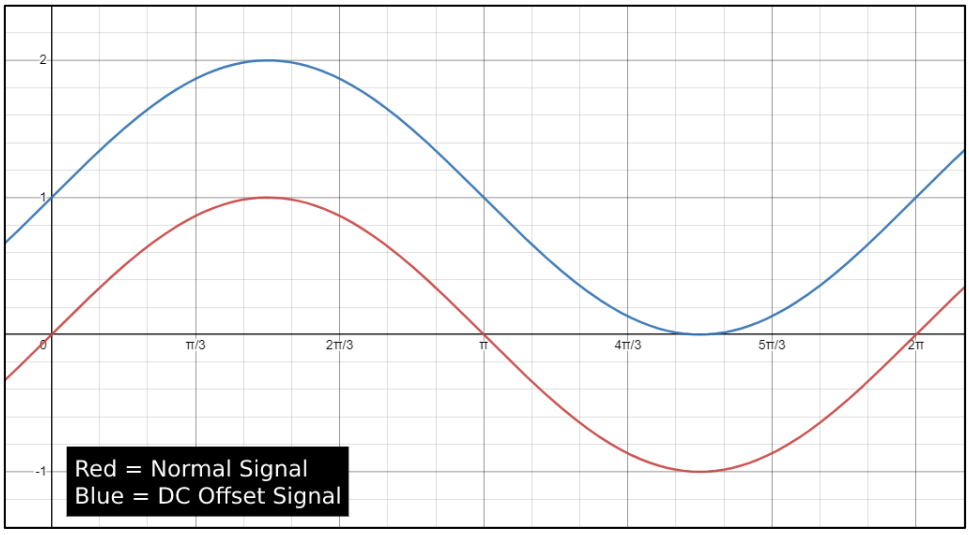


Implementation

Firstly, the left and right channels are processed separately, allowing the plugin to work in stereo with the same processing for each channel. The signal goes through a DC Offset module which can either be used to correct an outlying offset or introduce one to create different distortion tones. Then it is run through a gain module, which is the main control for distortion. Then the signal is run through a filter which replicates the tone control on a distortion pedal. Then the signal is run through a dynamics algorithm. This will adaptively adjust the volume of the output to match the input signal, because distortion can reduce the dynamic content of a signal. Finally, the signal is run through a DC Offset correction filter to eliminate any outlying offset from the signal. Differing from my other plugins, I don't need a mix control because distortion is achieved through adding more gain which makes a mix control redundant.

DC Offset

DC offset, conventionally called 'Bias' in distortion plugins, occurs when a signal has been offset from the origin. This is best explained with a simple diagram.



As we can see the sine wave has been moved up by 1 unit on the diagram. Generally, this undesirable when producing music because it can introduce unwanted artifacts when working with the audio. By introducing a 'Bias' control I can attenuate the signal before the distortion, limiting the number of unwanted artifacts. However, in some cases you want to introduce errors in the distortion because of the sound it creates. This feature was suggested after initial feedback from stakeholders.

However, DC Offset can carry through after distortion which is why I'll add a correction filter which will remove the DC Offset before the signal is outputted. There are many ways of implementing a correction filter, you could take an average of the signal and try to determine if there is an offset however this is more complicated than it needs to be. The simplest method is to implement a High-Pass filter below 10hz which removes this offset entirely. When you look at a spectrogram which can

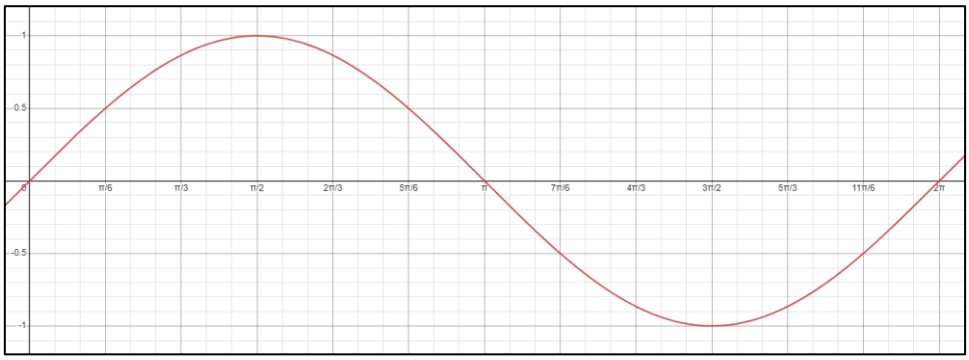
go as low as 1hz, there is a visible signal which is inaudible to the human ear. Since there will be no perceivable loss of quality, and it is the simplest method it is the obvious choice.

Distortion Modes

Since distortion is such a wide field in DSP, I will implement three different modes to give versatility and diversity to the user. Each type of distortion has its own transfer function which creates its signature tone. Below is a brief description of each mode and a simple explanation of how to implement it.

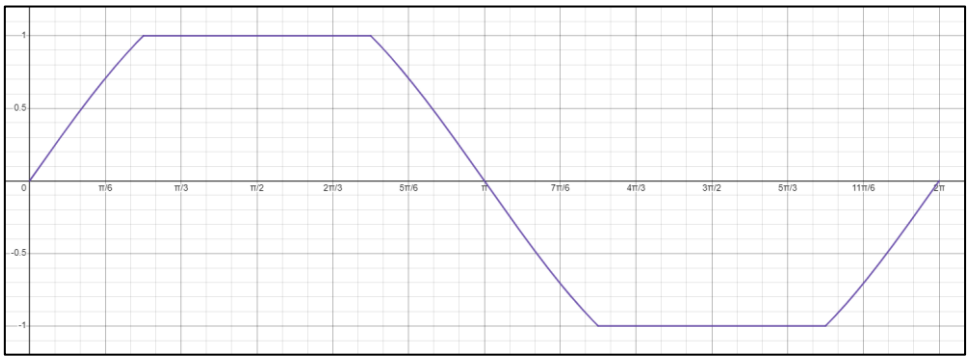
Hard-Clipping

Hard clipping is the simplest type of distortion to implement, and it is essential to any distortion plugin. As the gain of the inputted signal increases, it reaches a predetermined threshold. When it reaches this threshold, it gets set to it. Creating harsh, distorted tones. Hard clipping can easily be shown on a diagram.



Sine Wave

[Distortion_Example_1]



Sine Wave + Hard-Clipping

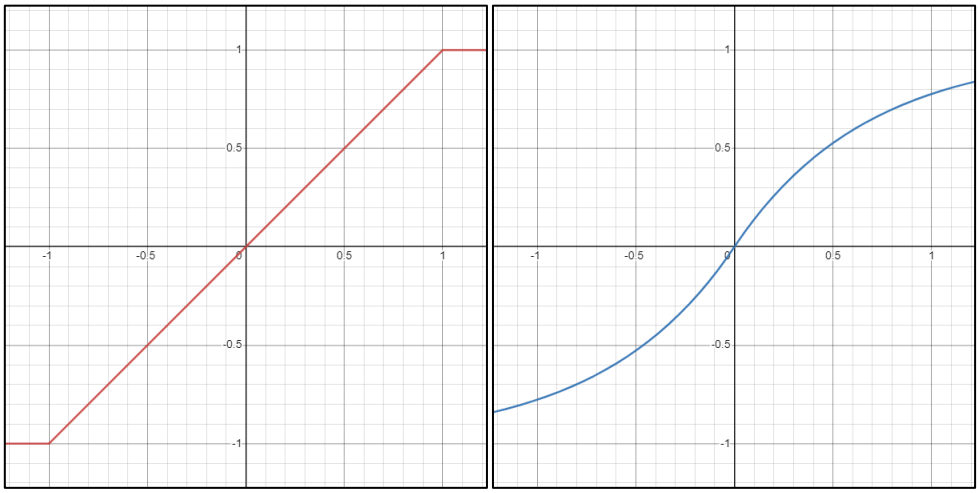
[Distortion_Example_2]

C++ has a handy set of functions which can create this effect for us, fmin() and fmax() which returns the smaller or larger inputted values, respectively. Since the signal can be between 1 and –1, if its greater than 1 we use fmin() and if it's less than 1 we use Fmax. This creates a harsh sound which is comparable to a square wave when the gain is set high enough, which can be seen as undesirable. Which leads on to the next distortion mode, Overdrive.

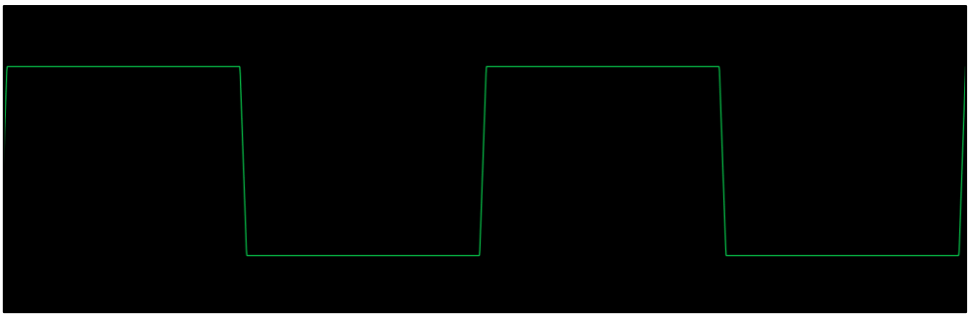
Overdrive

Overdrive, also known as Soft-Clipping, is similar to Hard-Clipping, however it uses a transfer function to create distortion. A transfer function maps an input to an output. Soft-Clipping uses a smoother curve when compared to hard clipping, hence the name. Here are the diagrams for each of the transfer functions and their effects on a sinewave shown on an oscilloscope.

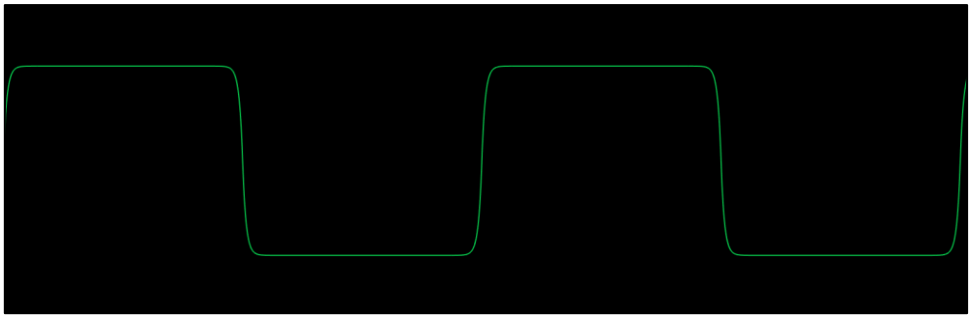
Transfer Functions



Hard-Clipping curve Soft-Clipping curve



Hard Clipping



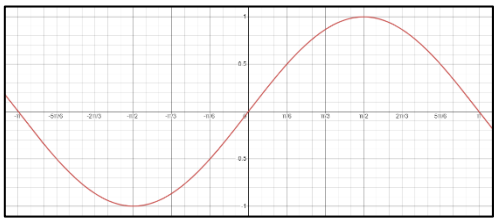
Soft Clipping

I will be using the transfer function suggested by Matt L in response to a question on the DSP Stack Exchange [1]. The equation $y = sgn(x) \cdot \left(1 - e^{(-|x|)}\right)$ gives us the desired curve and Matt provides a simple implementation of the algorithm which I will be using later. Prior to this I considered using a tanh curve owing to it having a similar shape and it approaches -1 and 1 as it tends to infinity, however when comparing performance, a tanh curve has more exponentials which would decrease performance.

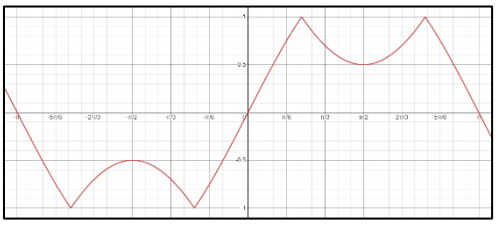
Here is an example of Soft-Clipping on a sine wave. [Distortion_Example_3]

Wavefolding

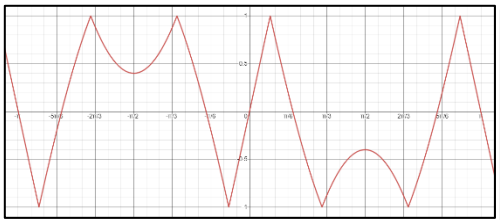
Wavefolding put simply, is where the signal folds over itself after exceeding a certain threshold. Although it's a simple concept, depending on the implementation it can create drastically different sounds. This is best explained using a diagram.



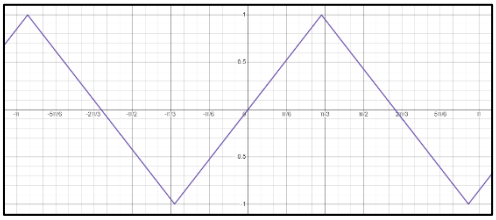
Sine Wave



Sine Wave + Triangle Fold



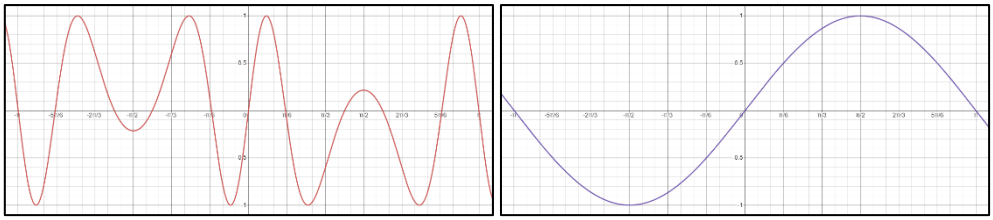
Sine Wave + Triangle Fold



Triangle Function

[Distortion_Example_4]

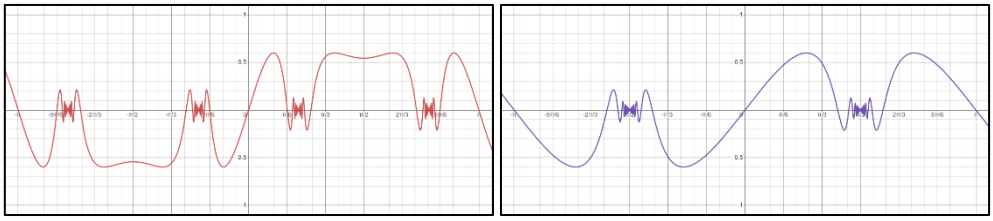
This is an example of triangle fold over, owing to the transfer function being a triangle wave. As the gain increases, the input folds over itself more. This causes triangle waves to appear between the original input. This led me to use a sine wave as a transfer function instead, which created a drastically different sound.



Sine Wave + Sine Fold Sine Function

[Distortion_Example_5]

Since this creates unique and interesting sounds, I've implemented 3 different transfer functions into my plugin; Triangle, Sine and HRM. The first two are self-explanatory, with the HRM mode being a custom function I coded with the equation $y = \cos(x) \cdot (\sin(\tan x))$. The triangle function I am using was suggested by 2DaT on kvraudio [2], I'm using it because it is faster than deriving the function myself. Here is a diagram of my custom function.

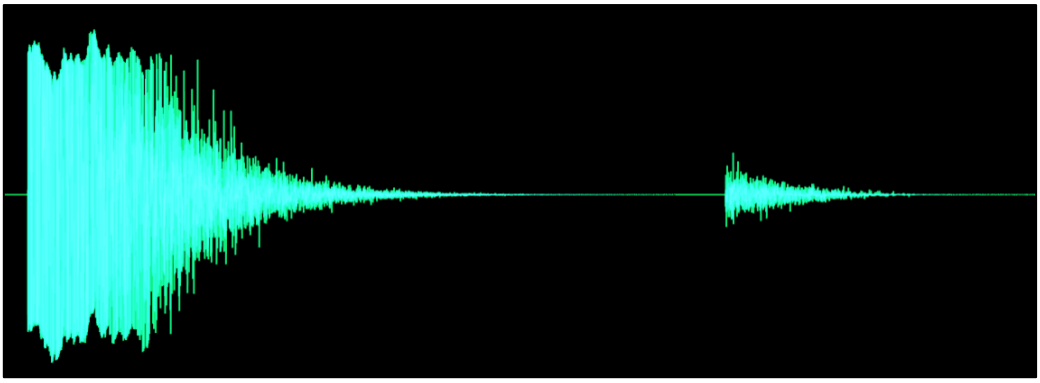


Sine + HRM Fold HRM Function

[Distortion_Example_6]

Dynamics

Distortion can remove the dynamic content of the signal which can be undesirable. Dynamics mode multiplies the processed signal with the inputted signal, reintroducing the dynamics of the input whilst retaining the tone of the distortion. I will use linear interpolation to blend between the normal and dynamic signal. This concept can be shown on a diagram.

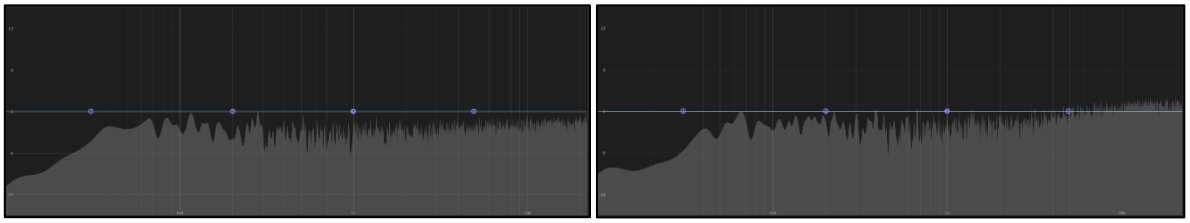


Dynamics Off Dynamics On

[Distortion_Example_7]

Tone Filter

After researching what the tone knob does on guitar pedals, it's a simple high-shelf filter which is used to boost / cut the high-end of a signal. After some experimentation I chose to have it boost above 3.5khz and with a range of 16db centred around 0. I will be reusing the filter code from my delay plugin because it already has high-shelf as a filter mode.

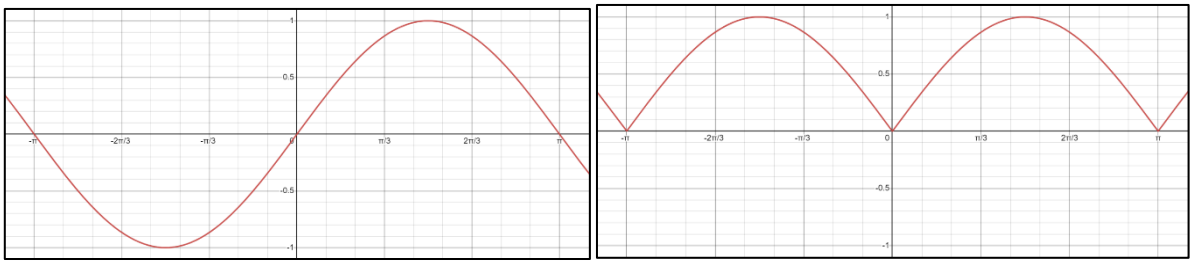


Tone Knob Off Tone Knob On

[Distortion_Example_8]

Rectify

Rectify is where the polarity of the inputted signal is inverted when its value is less than 0. This has the perceived effect of the frequency being doubled due to the wave's period being halved. This can introduce unwanted DC Offset. However, this will be corrected at the end of the signal chain.



Rectify Off Rectify On

[Distortion_Example_9]

ProcessDistortion Class

Since the distortion algorithm can be abstracted to a separate class I shall do so in accordance with [SC7_A]. Here I will outline the different functions and variables that I'll need to create the class and their purpose.

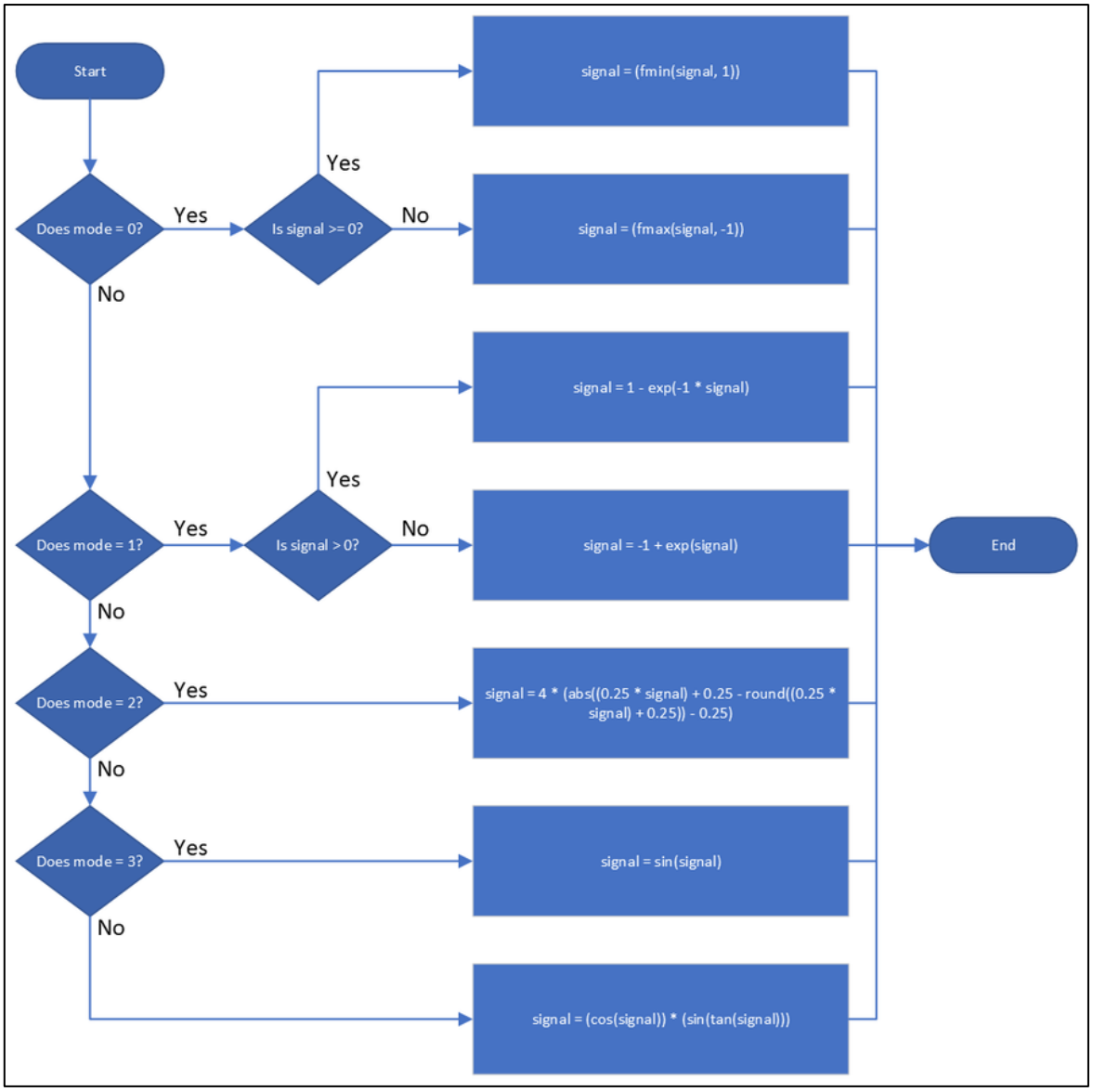
Functions:

- SetFilters – This function is called to set the sample rate of the filters to the current sample rate. This should be run when the plug-in is initialised.
- DCOffset – Applies DC offset to the input depending on the bias parameter.
- Gain – Applies Gain to the input depending on the gain parameter.
- Distortion – Applies a distortion algorithm to the input signal depending on the mode parameter.
- ToneFilter – Applies filtering to the input depending on the tone parameter.
- Dynamics – Applies dynamics to the input depending on the dynamics parameter.
- Rectify – Applies Rectify to the input.
- DCCorrection – Applies a DC correction filter to the input.

Variables:

- hsFilter – Creates a new High-Shelf filter for the ToneFilter function.
- hpFilter - Creates a new High-Pass filter for the DCCorrection function.

Since most of the functions and variables are self-explanatory, I will only explain Distortion function although it is quite simple. Here is the flowchart for the algorithm.



The first chunk of decisions are used to determine which mode we want to use. In order the different modes are; Hard-Clipping, Soft-Clipping, Triangle Fold, Sine Fold and HRM Fold. Each of which I have already discussed. After the audio has been processed, it is outputted.

Development

Firstly, I duplicated the default project from WDL-OL and set up the resource.h file. I wanted to get the simplest distortion modes implemented first before adding any extra features so I could have a working plugin before adding extra features. Starting with Hard-Clipping, I used the fmin and fmax functions described above to create this effect.

```
48  void DistortionPlugin::ProcessDoubleReplacing(double** inputs, double** outputs, int nFrames)
49  {
50      // Mutex is already locked for us.
51
52      int const channelCount = 2;
53
54      for (int i = 0; i < channelCount; i++) {
55          double* input = inputs[i];
56          double* output = outputs[i];
57
58          for (int s = 0; s < nFrames; ++s, ++input, ++output) {
59              if (*input >= 0) {
60
61                  *output = fmin(*input, mThreshold);
62              }
63              else {
64
65                  *output = fmax(*input, -mThreshold);
66              }
67          }
68      }
69  }
```

I initially created a variable mThreshold, which sets the threshold at which distortion will be applied. During initial testing, the signal kept getting quieter when I increased the amount of distortion. The distortion worked fine, however because I wasn't scaling the output afterwards the max volume was the threshold, which got smaller as I increased the amount of distortion. Instead of scaling the output I decided to change my approach designing the distortion algorithms. Looking back at distortion pedals it was the logical choice to replace my threshold control with a gain control. It drops the problem of having to scale the output and allows for finer control over the amount of distortion. Firstly, I removed mThreshold and replaced the values in Fmin and Fmax with 1 and –1, then I remade the gain control.

```
48  void DistortionPlugin::ProcessDoubleReplacing(double** inputs, double** outputs, int nFrames)
49  {
50      // Mutex is already locked for us.
51
52      int const channelCount = 2;
53
54      for (int i = 0; i < channelCount; i++) {
55          double* input = inputs[i];
56          double* output = outputs[i];
57
58          for (int s = 0; s < nFrames; ++s, ++input, ++output) {
59
60              // Gain
61              *output = mGain * *input;
62
63              // Hard Clipping
64              if (*output >= 0) {
65
66                  *output = fmin(*output, 1);
67              }
68              else {
69
70                  *output = fmax(*output, -1);
71              }
72          }
73      }
74  }
```

Implementing the gain module is as simple as multiplying the signal with the mGain variable. After fixing this, I moved on implementing Soft-Clipping. To swap between the different distortion modes I creates a switch statement which checks the value of mMode which is updated with a button on the GUI. As described above Soft-Clipping uses a transfer function which maps an input to an output. Using the function by Matt L it translates to this in code.

```
// Soft Clipping
if (*output > 0) {

    *output = 1 - exp(-1 * *output);
}
else {

    *output = -1 + exp(*output);
}
break;
```

It looks much simpler than the equation because the $\mathrm{sgn}(x)$ function sets the output to 1 if its positive and –1 if its negative so we can just use an if statement and change the signs around to make it work. Moving on to the mode selection, I created a new int variable mMode which will store the current distortion mode. Alongside this I created a button in the GUI which allows the user to select which distortion mode they want.

```
69          switch (mMode) {
70
71          case 0:
72
73              // Hard Clipping
74              if (*output >= 0) {
75
76                  *output = fmin(*output, 1);
77              }
78              else {
79
80                  *output = fmax(*output, -1);
81              }
82              break;
83
84          case 1:
85
86              // Soft Clipping
87              if (*output > 0) {
88
89                  *output = 1 - exp(-1 * *output);
90              }
91              else {
92
93                  *output = -1 + exp(*output);
94              }
95              break;
96          }
```

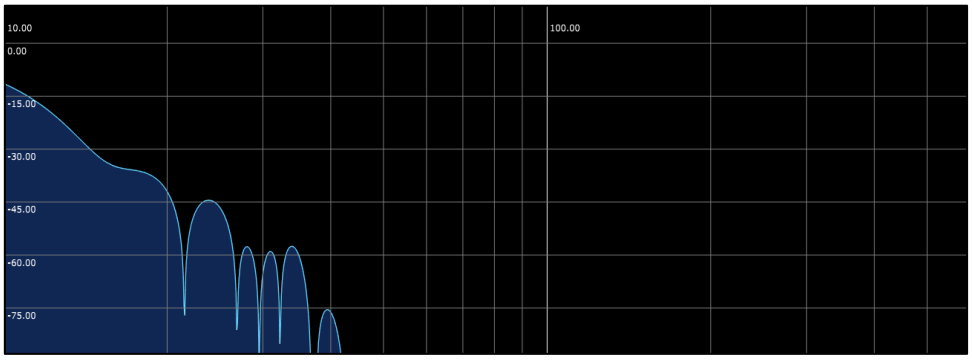
After getting the first two distortion algorithms working, I created my DC Offset code. As described earlier it's as simple as adding a value to the input. I created a new double called mBias and created a knob on the GUI to control it.

```
// DC Offset
*output = *input + mBias;
```

At this point discovered I discovered a ghost signal after adding in DC Offset. I thought a straightforward way of correcting it would be to subtract the offset I added after the distortion module which should still give me the tone of DC Offset without the ghost signal. This worked while mBias was small however as it got larger the ghost signal reappeared. Here is the simple offset correction code.

```
// DC Offset Correction
*output = *output - mBias;
```

After some experimentation and research, I discovered that a high-pass filter around 10hz would fix this issue because the offset creates a ghost signal inaudible to the human ear blow 10hz. I opened a spectrogram and found my ghost signal. Reusing the filter code from my delay plugin I created a new high-pass filter at 10hz and put it at the end of my processing chain. This fixed the issue and didn't require much effort too.



[Frequency Spectrum showing ghost signal]

```
// DC Correction Filter
Biquad *hpFilter = new Biquad();
```

```
hpFilter->setBiquad(bq_type_highpass, 19 / GetSampleRate(), 0.9, 0);
```

```
// DC Offset Correction filter
signal = hpFilter->process(signal);
```

Moving on, I wanted to see what would happen when I introduced DC Offset into my wavefolding algorithms, so I decided to program them next. Starting off with triangle fold, I translated 2DaT's function into code and tested it out. During initial testing, the wavefolding created a loud distorted sound no matter what the input gain was. However, I forgot to add a decimal place in one of the coefficients and after fixing this it worked as intended.

```
case 2:

    // Triangle Foldover
    *output = 4 * (abs((0.25 * *output) + 0.25 - round((0.25 * *output) + 0.25)) - 0.25);
    break;
}
```

Following on, I implemented sine and HRM foldover. When looking at the waveform as the gain increased, I stated to see copies of the original wave between each fold. Interestingly when using the sine foldover on a sine wave, as the gain increases you can start to hear each harmonic appear as the gain increases which I thought was quite interesting.

```
case 3:

    // Sine Foldover
    *output = sin(*output);
    break;

case 4:

    // HRM Foldover
    *output = (cos(*output)) * (sin(tan(*output)));
    break;

}
```

After finishing all the distortion algorithms, I decided to implement the bypass switch. I created an int variable called mButton which determines if the plugin is bypassed or not. Again, it's simple to implement by just using an if statement in the processing function to check if the plugin is bypassed. However, initially I forgot to update mButton when the switch on the GUI was pressed, this firstly meant that the switch didn't work however when the button was pressed the output sounded like it was bit crushed. After amending this, the switch worked and the bit crush was fixed as well.

```
// Check if bypassed
if (mButton == 1) {
```

When implementing the dynamics module, I want to interpolate between the distorted signal and the distorted signal multiplied by the magnitude of the dry signal. If I didn't multiply by the magnitude of the dry signal the output would only be positive which is undesirable. I found this out when originally coding the linear interpolation and the output sounding like I had turned on rectify mode. Which leads on to implementing rectify mode. I created a new int variable called mRectify and if it is set to 1, I return the magnitude of the inputted signal. Here is the code for both modules.

```
// Dynamics
*output = (mDynamics * *output) + (1 - mDynamics) * (*output * abs(*input));

// Rectify Mode
if (mRectify == 1) {

    *output = abs(*output);

}
```

Finally, moving onto the tone control, it's a simple high-shelf filter where the only thing changing is the peak gain of the filter. I created a new high-shelf filter with the cut-off set to 3.5khz and the peak gain set to a double variable mTone. Before the signal is ran through the filter the peak gain is updated to mTone because it could have changed from one sample to the next.

```
// High Shelf Filter
Biquad *hsFilter = new Biquad();
```

```
// Filter
hsFilter->setPeakGain(mTone);
signal = hsFilter->process(signal);
```

When I originally implemented the filter, I had a phasing issue between the left and right channels whenever the filter was active. After some debugging, I figured out that there was a fundamental issue when setting up my processing loop. I processed each channel in sequence which wasn't an issue when there wasn't any latency in the processing chain. However, my filter implementation has 1 sample worth of latency. Since I'm running the signal through two filters each the left channel is delay by 2 samples when compared to the right. This led me to put all my processing into a function and process each channel before moving onto the next sample. I also encountered a bug when changing the sample rate of my DAW, because I forgot to update the sample rate in the filter when it was changed the cut-off was only accurate for 44.1 khz. To fix this I added a few lines of code in the reset function which is called whenever the sample rate is changed. This fixed the issue and finished all the DSP in my plugin.

```
183  void DigitalDistortion::ProcessDoubleReplacing(double** inputs, double** outputs, int nFrames)
184  {
185      // Mutex is already locked for us.
186
187      double* in1 = inputs[0];
188      double* in2 = inputs[1];
189      double* out1 = outputs[0];
190      double* out2 = outputs[1];
191
192      for (int s = 0; s < nFrames; ++s, ++in1, ++in2, ++out1, ++out2)
193      {
194          *out1 = ProcessDistortion(*in1);
195          *out2 = ProcessDistortion(*in2);
196      }
197  }
```

```
void DigitalDistortion::Reset()
{
  hsFilter->setBiquad(bq_type_highshelf, 3500 / GetSampleRate(), 0.707, 0);
  hpFilter->setBiquad(bq_type_highpass, 19 / GetSampleRate(), 0.9, 0);
  TRACE;
  IMutexLock lock(this);
}
```

After I finished the plugin, I realised that I was programming in the wrong framework. I was programming in iPlug1 compared to iPlug2. Luckily, all my code should still work in iPlug2 because I didn't use too many iPlug1 specific functions and quirks. After I transferred all my code over, I initially tested the UI to check that every control was in the right place and moved. This is when I found out that the new GUI engine refreshes the image with your screen when compared to the 60hz of iPlug1.

When I tested if my DSP worked nothing happened, it was just outputting the input which is what should happen if the plugin is bypassed. I crosschecked it with my code from iPlug1 and it was identical. To fix this I tried passing every variable I was using into the function. This did fix the issue, but I still don't know why it didn't work in the first place.

```
for (int s = 0; s < nFrames; ++s, ++in1, ++in2, ++out1, ++out2)
{

  *out1 = ProcessDistortion(*in1, mBias, mGain, mButton, mMode, mRectify, mDynamics, mTone);
  *out2 = ProcessDistortion(*in2, mBias, mGain, mButton, mMode, mRectify, mDynamics, mTone);

}
```

When moving over to iPLug2 I had to make a few slight changes to how the controls were handled however this is mainly due to iPlug2 lacking the OnParamChange() function. The solution is to update the variable storing the data of the control every sample, although not as elegant as iPlug1 it still works.

```
#if IPLUG_DSP
void PsoniteTest::ProcessBlock(sample** inputs, sample** outputs, int nFrames)
{
  const double mBias = GetParam(kBias)->Value();
  const double mButton = GetParam(kButton)->Value();
  const double mGain = GetParam(kGain)->Value();
  const double mMode = GetParam(kMode)->Value();
  const double mRectify = GetParam(kRectify)->Value();
  const double mDynamics = GetParam(kDynamics)->Value();
  const double mTone = GetParam(kTone)->Value();
```

After assessing my success criteria realised that I should reimplement my distortion processing as a class. This would address [SC7_A] and allow me to use it in my delay plugin. Firstly, I set up the header file with all the functions and filters I need.

```h
#include "Biquad.h"

#pragma once
class ProcessDistortion
{
public:
  // Functions
  void SetFilters(double sampleRate);

  double DCOffset(double signal, double bias);
  double Gain(double signal, double gain);
  double HardClipping(double signal);
  double SoftClipping(double signal);
  double TriangleFold(double signal);
  double SineFold(double signal);
  double WeirdFold(double signal);
  double ToneFilter(double signal, double tone);
  double Dynamics(double signal, double preSignal, double dynamics);
  double Rectify(double signal);
  double DCCorrection(double signal);

private:
  // High Shelf Filter
  Biquad* hsFilter = new Biquad();

  // DC Correction Filter
  Biquad* hpFilter = new Biquad();
};
```

After this I moved the processing code into each function, checking that all the relevant parameters and being passed to each function. Here is the code for every function.

```cpp
#include "ProcessDistortion.h"
#include "Biquad.h"
#include "cmath"

void ProcessDistortion::SetFilters(double sampleRate)
{
  hsFilter->setBiquad(bq_type_highshelf, 3500 / sampleRate, 0.707, 0);
  hpFilter->setBiquad(bq_type_highpass, 19 / sampleRate, 0.9, 0);
}

double ProcessDistortion::DCOffset(double signal, double bias)
{
  signal += bias;
  return signal;
}

double ProcessDistortion::Gain(double signal, double gain)
{
  signal *= gain;
  return signal;
}

double ProcessDistortion::HardClipping(double signal)
{
  if (signal >= 0)
  {
    signal = (fmin(signal, 1));
  }
  else
  {
    signal = (fmax(signal, -1));
  }
  return signal;
}
```

```
double ProcessDistortion::SoftClipping(double signal)
{
    if (signal > 0)
    {

        signal = 1 - exp(-1 * signal);
    }
    else
    {

        signal = -1 + exp(signal);
    }
    return signal;
}

double ProcessDistortion::TriangleFold(double signal)
{
    signal = 4 * (abs((0.25 * signal) + 0.25 - round((0.25 * signal) + 0.25)) - 0.25);
    return signal;
}

double ProcessDistortion::SineFold(double signal)
{
    signal = sin(signal);
    return signal;
}

double ProcessDistortion::WeirdFold(double signal)
{
    signal = (cos(signal)) * (sin(tan(signal)));
    return signal;
}
```

```
double ProcessDistortion::ToneFilter(double signal, double tone)
{
    hsFilter->setPeakGain(tone);
    signal = hsFilter->process(signal);
    return signal;
}

double ProcessDistortion::Dynamics(double signal, double preSignal, double dynamics)
{
    signal = (dynamics * signal) + (1 - dynamics) * (signal * abs(preSignal));
    return signal;
}

double ProcessDistortion::Rectify(double signal)
{
    signal = abs(signal);
    return signal;
}

double ProcessDistortion::DCCorrection(double signal)
{
    signal = hpFilter->process(signal);
    return signal;
}
```

After including the processing class, I set up all the functions similarly to how my original processing function worked. Although it seems backwards in this case, when implementing different distortion modes into other plugins it will heavily speed up the process. I also had to include a small piece of code in the constructor to tell the processing class to create the filters.

```
// Check if bypassed
if (mButton == 1)
{
  // Apply Bias
  *out1 = Processing.DCOffset(*in1, mBias);
  *out2 = Processing.DCOffset(*in2, mBias);

  // Apply Gain
  *out1 = Processing.Gain(*out1, mGain);
  *out2 = Processing.Gain(*out2, mGain);

  // Distortion modes
  switch (mMode)
  {
    case 0:
      // Hard Clipping
      *out1 = Processing.HardClipping(*out1);
      *out2 = Processing.HardClipping(*out2);
      break;

    case 1:
      // Soft Clipping
      *out1 = Processing.SoftClipping(*out1);
      *out2 = Processing.SoftClipping(*out2);
      break;
```

```
// Set up filters
Processing.SetFilters(GetSampleRate());
```

Finally, when I was reassessing my code, I realised that it would be better to only implement one function to apply distortion. This both simplifies the program, making the class easier to implement and reduces duplicate code.

```cpp
double ProcessDistortion::Distortion(double signal, int mode)
{
  switch (mode)
    {
    // Hard Clipping
    case 0:
      if (signal >= 0)
      {
        signal = (fmin(signal, 1));
      }
      else
      {
        signal = (fmax(signal, -1));
      }
      break;

    // Soft Clipping
    case 1:
      if (signal > 0)
      {

        signal = 1 - exp(-1 * signal);
      }
      else
      {

        signal = -1 + exp(signal);
      }
      break;

    // Triangle Fold
    case 2:
      signal = 4 * (abs((0.25 * signal) + 0.25 - round((0.25 * signal) + 0.25)) - 0.25);
      break;

    // Sine Fold
    case 3:
      signal = sin(signal);
      break;

    // HRM Fold
    case 4:
      signal = (cos(signal)) * (sin(tan(signal)));
      break;
    }
  return signal;
}
```

Once this was complete the plugin was finished. I created a new GUI from my blender template and attached all the images leaving me with the result.

All the tests for the plugin will appear later in the document but here is the full code for the main program and distortion class.

Main program code:

```cpp
#include "PsoniteTest.h"
#include "IPlug_include_in_plug_src.h"
#include "IControls.h"
#include "ProcessDistortion.h"

ProcessDistortion Processing;

PsoniteTest::PsoniteTest(const InstanceInfo& info)
: Plugin(info, MakeConfig(kNumParams, kNumPresets))
{
  // Set up filters
  Processing.SetFilters(GetSampleRate());

  // Bias Knob
  GetParam(kBias)->InitDouble("Bias", 0.0, -0.5, 0.5, 0.01, "%");

  // Gain Knob
  GetParam(kGain)->InitDouble("Gain", 1.0, 1.0, 36.0, 0.01, "%");

  // Bypass Switch
  GetParam(kButton)->InitEnum("On Off", 0, 1);

  // Mode Switch
  GetParam(kMode)->InitInt("Mode", 0, 0, 4);

  // Rectify button
  GetParam(kRectify)->InitEnum("Rectify", 0, 1);

  // Dynamics Knob
  GetParam(kDynamics)->InitDouble("Dynamics", 1.0, 0.0, 1.0, -0.01, "%");

  // Tone Knob
  GetParam(kTone)->InitDouble("Tone", 0.0, -8.0, 8.0, 0.01, "db");
```

```cpp
#if IPLUG_EDITOR // http://bit.ly/2S64BDd
  mMakeGraphicsFunc = [&]() {
    return MakeGraphics(*this, PLUG_WIDTH, PLUG_HEIGHT, PLUG_FPS, GetScaleForScreen(PLUG_WIDTH, PLUG_HEIGHT));
  };

  mLayoutFunc = [&](IGraphics* pGraphics) {

    // Create Canvas
    pGraphics->AttachBackground(GUI_FN);
    pGraphics->LoadFont("Roboto-Regular", ROBOTO_FN);

    // Load Images
    const IBitmap Bias = pGraphics->LoadBitmap(BIAS_FN, 128);
    const IBitmap Dynamics = pGraphics->LoadBitmap(DYNAMICS_FN, 128);
    const IBitmap Filter = pGraphics->LoadBitmap(FILTER_FN, 128);
    const IBitmap Gain = pGraphics->LoadBitmap(GAIN_FN, 128);
    const IBitmap Rectify = pGraphics->LoadBitmap(RECTIFY_FN, 2);
    const IBitmap Bypass = pGraphics->LoadBitmap(BYPASS_FN, 2);
    const IBitmap Mode = pGraphics->LoadBitmap(MODE_FN, 5);

    // Attach controls
    pGraphics->AttachControl(new IBKnobControl(204, 33, Bias, kBias));
    pGraphics->AttachControl(new IBKnobControl(55, 136, Dynamics, kDynamics));
    pGraphics->AttachControl(new IBKnobControl(204, 136, Filter, kTone));
    pGraphics->AttachControl(new IBKnobControl(55, 33, Gain, kGain));
    pGraphics->AttachControl(new IBSwitchControl(194, 277, Rectify, kRectify));
    pGraphics->AttachControl(new IBSwitchControl(46, 261, Mode, kMode));
    pGraphics->AttachControl(new IBSwitchControl(211, 392, Bypass, kButton));
```

```
  };
#endif
}

#if IPLUG_DSP
void DistortionPlugin::ProcessBlock(sample** inputs, sample** outputs, int nFrames)
{
  const double mBias = GetParam(kBias)->Value();
  const int mButton = GetParam(kButton)->Value();
  const double mGain = GetParam(kGain)->Value();
  const int mMode = GetParam(kMode)->Value();
  const int mRectify = GetParam(kRectify)->Value();
  const double mDynamics = GetParam(kDynamics)->Value();
  const double mTone = GetParam(kTone)->Value();

  double* in1 = inputs[0];
  double* in2 = inputs[1];
  double* out1 = outputs[0];
  double* out2 = outputs[1];

  for (int s = 0; s < nFrames; ++s, ++in1, ++in2, ++out1, ++out2)
  {
    // Check if bypassed
    if (mButton == 1)
    {
      // Apply Bias
      *out1 = Processing.DCOffset(*in1, mBias);
      *out2 = Processing.DCOffset(*in2, mBias);
```

```
      // Apply Gain
      *out1 = Processing.Gain(*out1, mGain);
      *out2 = Processing.Gain(*out2, mGain);

      // Distortion modes
      *out1 = Processing.Distortion(*out1, mMode);
      *out2 = Processing.Distortion(*out2, mMode);

      // Filter
      *out1 = Processing.ToneFilter(*out1, mTone);
      *out2 = Processing.ToneFilter(*out2, mTone);

      // Dynamics
      *out1 = Processing.Dynamics(*out1, *in1, mDynamics);
      *out2 = Processing.Dynamics(*out2, *in2, mDynamics);

      // Rectify
      if (mRectify == 1)
      {
        *out1 = Processing.Rectify(*out1);
        *out2 = Processing.Rectify(*out2);
      }

      // DC Offset Correction Filter
      *out1 = Processing.DCCorrection(*out1);
      *out2 = Processing.DCCorrection(*out2);

    }
    else
    {
      *out1 = *in1;
      *out2 = *in2;
```

```
    }
  }
}
#endif
```

ProcessDistortion code:

```h
#include "Biquad.h"

#pragma once
class ProcessDistortion
{
public:
  // Functions
  void SetFilters(double sampleRate);

  double DCOffset(double signal, double bias);
  double Gain(double signal, double gain);
  double Distortion(double signal, int mode);
  double ToneFilter(double signal, double tone);
  double Dynamics(double signal, double preSignal, double dynamics);
  double Rectify(double signal);
  double DCCorrection(double signal);

private:
  // High Shelf Filter
  Biquad* hsFilter = new Biquad();

  // DC Correction Filter
  Biquad* hpFilter = new Biquad();
};
```

```cpp
#include "ProcessDistortion.h"
#include "Biquad.h"
#include "cmath"

void ProcessDistortion::SetFilters(double sampleRate)
{
  hsFilter->setBiquad(bq_type_highshelf, 3500 / sampleRate, 0.707, 0);
  hpFilter->setBiquad(bq_type_highpass, 19 / sampleRate, 0.9, 0);
}

double ProcessDistortion::DCOffset(double signal, double bias)
{
  signal += bias;
  return signal;
}

double ProcessDistortion::Gain(double signal, double gain)
{
  signal *= gain;
  return signal;
}

double ProcessDistortion::Distortion(double signal, int mode)
{
  switch (mode)
  {
  // Hard Clipping
  case 0:
    if (signal >= 0)
    {
      signal = (fmin(signal, 1));
    }
    else
    {
      signal = (fmax(signal, -1));
    }
    break;
```

```
    // Soft Clipping
    case 1:
      if (signal > 0)
      {

        signal = 1 - exp(-1 * signal);
      }
      else
      {

        signal = -1 + exp(signal);
      }
      break;

    // Triangle Fold
    case 2:
      signal = 4 * (abs((0.25 * signal) + 0.25 - round((0.25 * signal) + 0.25)) - 0.25);
      break;

    // Sine Fold
    case 3:
      signal = sin(signal);
      break;

    // HRM Fold
    case 4:
      signal = (cos(signal)) * (sin(tan(signal)));
      break;
    }
  return signal;
}

double ProcessDistortion::ToneFilter(double signal, double tone)
{
  hsFilter->setPeakGain(tone);
  signal = hsFilter->process(signal);
  return signal;
}
```

```
double ProcessDistortion::Dynamics(double signal, double preSignal, double dynamics)
{
  signal = (dynamics * signal) + (1 - dynamics) * (signal * abs(preSignal));
  return signal;
}

double ProcessDistortion::Rectify(double signal)
{
  signal = abs(signal);
  return signal;
}

double ProcessDistortion::DCCorrection(double signal)
{
  signal = hpFilter->process(signal);
  return signal;
}
```

As stated before, all the code is available on my Github profile and may have been updated since the creation of this document.

Bibliography:

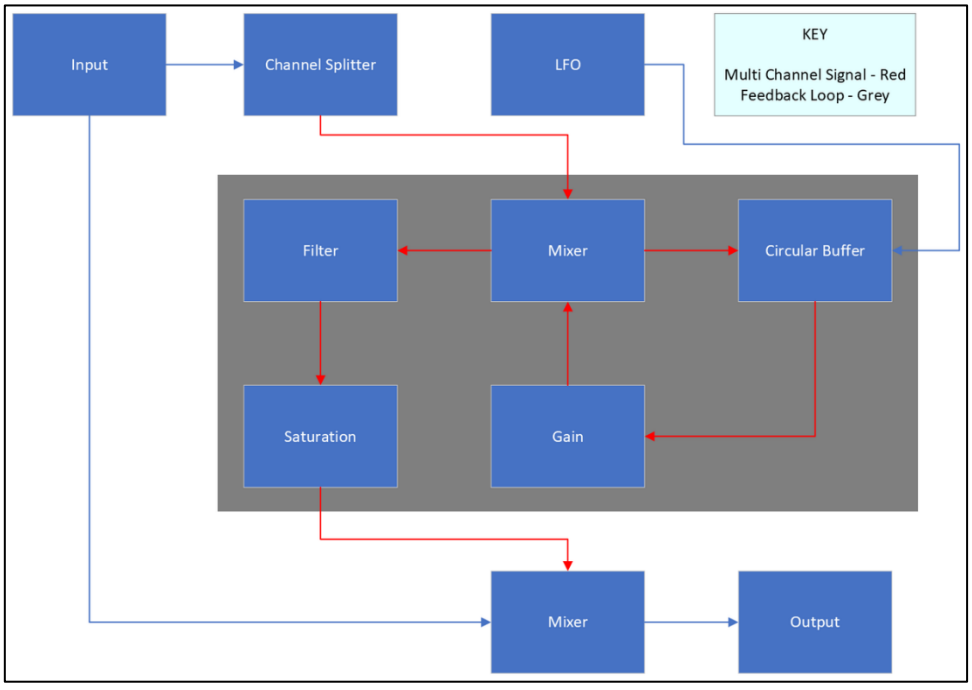
1. Matt L, "Digital Distortion Effect Algorithm"
2. 2DaT, “Wavefolding?”

Flanger Outline

Design

Building on my design for a delay, a Flanger is very similar with a few changes. Its delay times are much shorter, normally between 1-5ms with an LFO (Low Frequency Oscillator) modulating the delay time. This creates a metallic, 'flanging' effect which is typically heard on guitars and vocals. The first form of 'Flanging' was created when two tape recorders were fed an identical signal however a slight amount of pressure was applied to the flange (rim) of one reel. This causes the tape to slow down, delaying the two signals, producing comb filtering. From this, the effect has most notably been used on guitars and vocals, for example we can see it used on the vocals to the Beatles' track 'Tomorrow Never Knows'.

Since a Flanger shares many similarities with a delay I will modify my delay to function as a Flanger. The main change is the addition of an LFO and the order of each module. I will also keep the tone and warmth controls since from initial testing with my delay I feel that they are good additions to the plug-ins. The signal diagram below shows the main modules of the plugin that I'll have to implement.



Implementation

As explained earlier I will reuse most of the code from my delay so all I need to do is program the LFO and modify a few things, so everything works together. To link the LFO with the delay time, I will multiply the output of the oscillator with a constant and then add this to the delay creating the varying delay time. To create the LFO Amount I multiply this output by the LFO Amount Control. This will create a form of scaling, controlling the amount of modulation. This will also create a static Comb Filter when the LFO Amount is 0.

To create a stereo effect, I will use the same premiss for stereo delay. I will use a separate buffer for the left and right channels with the same LFO modulating their delay times. This allows both channels to be controlled by one LFO simplifying the plug-in.

Comb Filtering

When a sound is put through a Flanger it produces comb filtering. A comb filter combined both constructive and destructive interference by summing the original signal with a delayed signal.

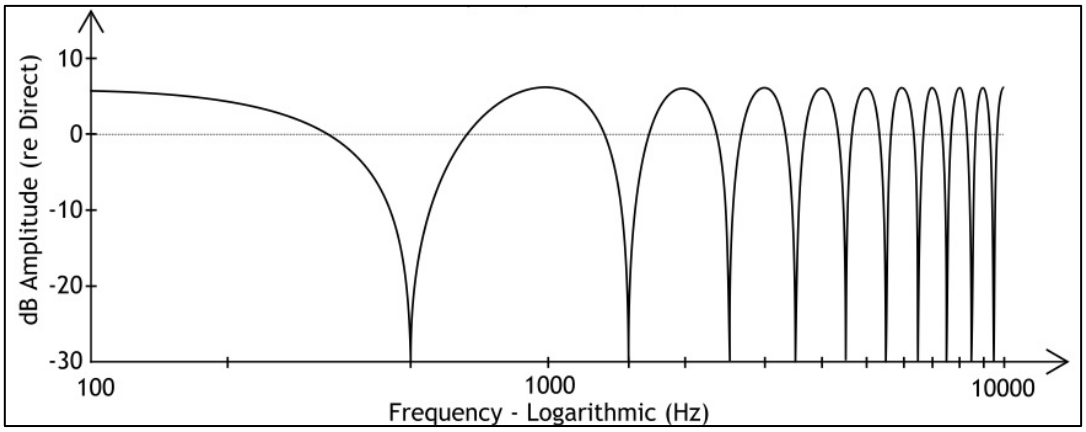
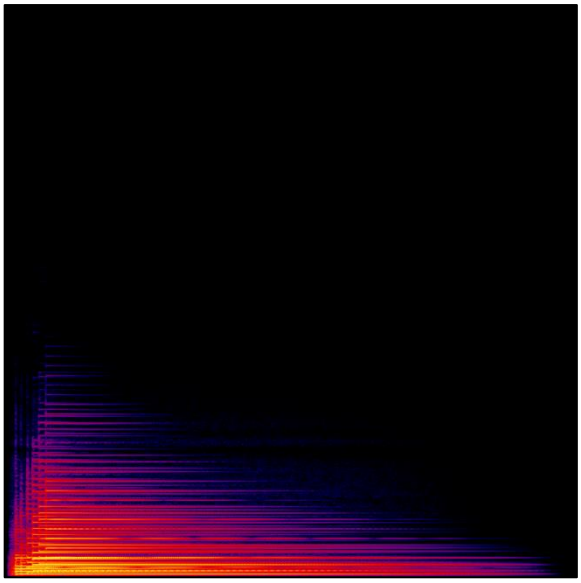
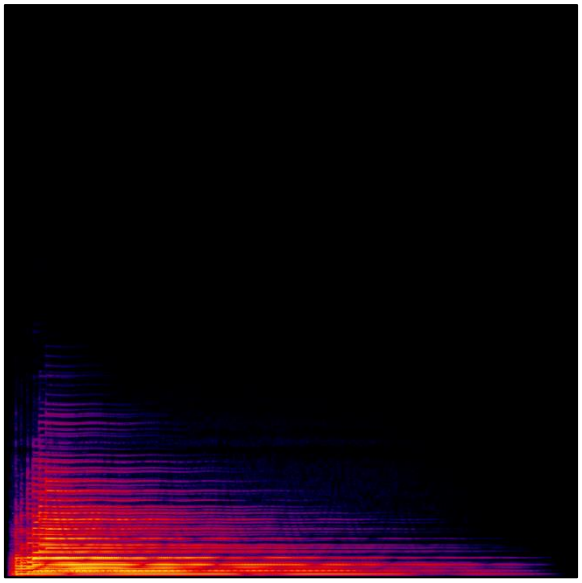


Image From Recordingology [1]

It's called a comb filter because of the shape of the frequency response it produces. Here is an example of a guitar chord [Flanger_Example_1] being run through a dry signal chain. Now when we add a Comb Filter it sounds like this [Flanger_Example_2]. If we slowly modulate the cut-off of the filter at 20hz, it sounds like this [Flanger_Example_3]. If we compare this to a Flanger [Flanger_Example_4] we can hear that it is producing the same effect.



Dry Guitar Chord



Flanged Guitar Chord

The LFO

An LFO is created from an oscillator which runs at low frequencies (go figure), typically for an LFO this is bellow 20hz. The oscillator will be made of two components, the waveforms and the phase calculations. I will be following Martin Finke's implementation of an oscillator since it is simple, and he created it for iPlug1 so it should still work [2].

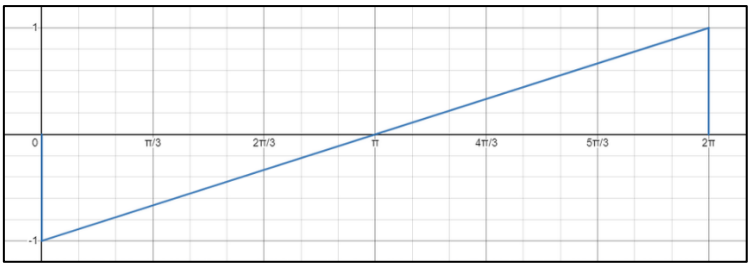
The Waveforms are made through looping functions bound between $0 \leq x \leq 2\pi$. I will have four different waveforms for the LFO; Sine, Saw, Triangle and Square. The phase calculations link the frequency with the sample rate allowing me to set the oscillator to the desired frequency. Using {Phase Increment = (Frequency * 2π) / Sample Rate} all I have to do is increment the phase of each function every sample, producing the desired waveform at the correct frequency regardless of sample rate.

Sine Wave

The Sine waveform is the easiest waveform to create, C++ already has a sine function so I will just pass my 'current phase' variable through it.

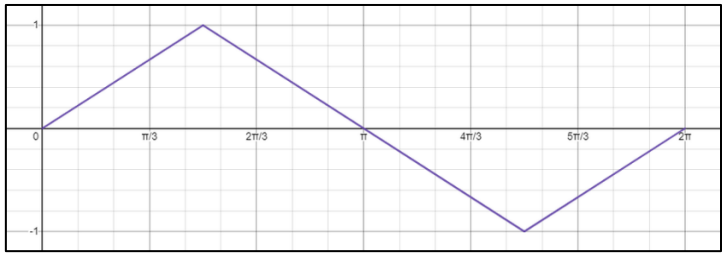
Saw Wave

The Saw waveform is created by manipulating the $y = x$ equation slightly. I want it to start and –1 and finish at 1, so the equation becomes $y = x - 1$. Finally, I want it to reach 1 at 2 π so I will give it a gradient of 1/ π. This gives me my final equation of $y = \frac{1}{\pi}x - 1$, were x is my 'current phase' variable and y is my output.



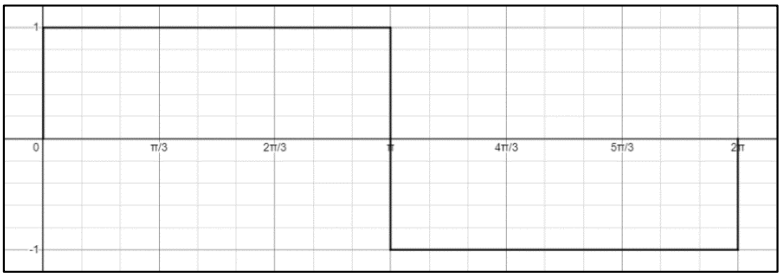
Triangle Wave

The Triangle waveform is the most complicated out of the four. The final equation is $y = 2\left|\frac{1}{\pi}x - 1\right| - 1$, quite complex but it is very similar to the equation for a Saw wave. We have the same $y = \frac{1}{\pi}x - 1$ however, we are the absolute value of it. This creates the change in gradient for the curve. Taking away 0.5 and multiplying by 2 just centres it around 0 and puts it between –1 and 1. If you plot the equation I have created, you will notice that is it out of phase with my graph. Since the graph loops between $0 \leq x \leq 2\pi$ the phase doesn't matter because it will produce the same result.



Square Wave

The Square waveform is created by setting the output (y) to either 1 or 0. Between $0 \leq x \leq \pi$, the output will be set to 1 and for $\pi \leq x \leq 2\pi$, the output will be set to –1. Where x is my 'current phase' variable and y is my output.



Oscillator class

Since the oscillator can be abstracted to a separate class I shall do so in accordance with [SC7_A]. Here I will outline the different functions and variables that I'll need to create the class and their purpose.

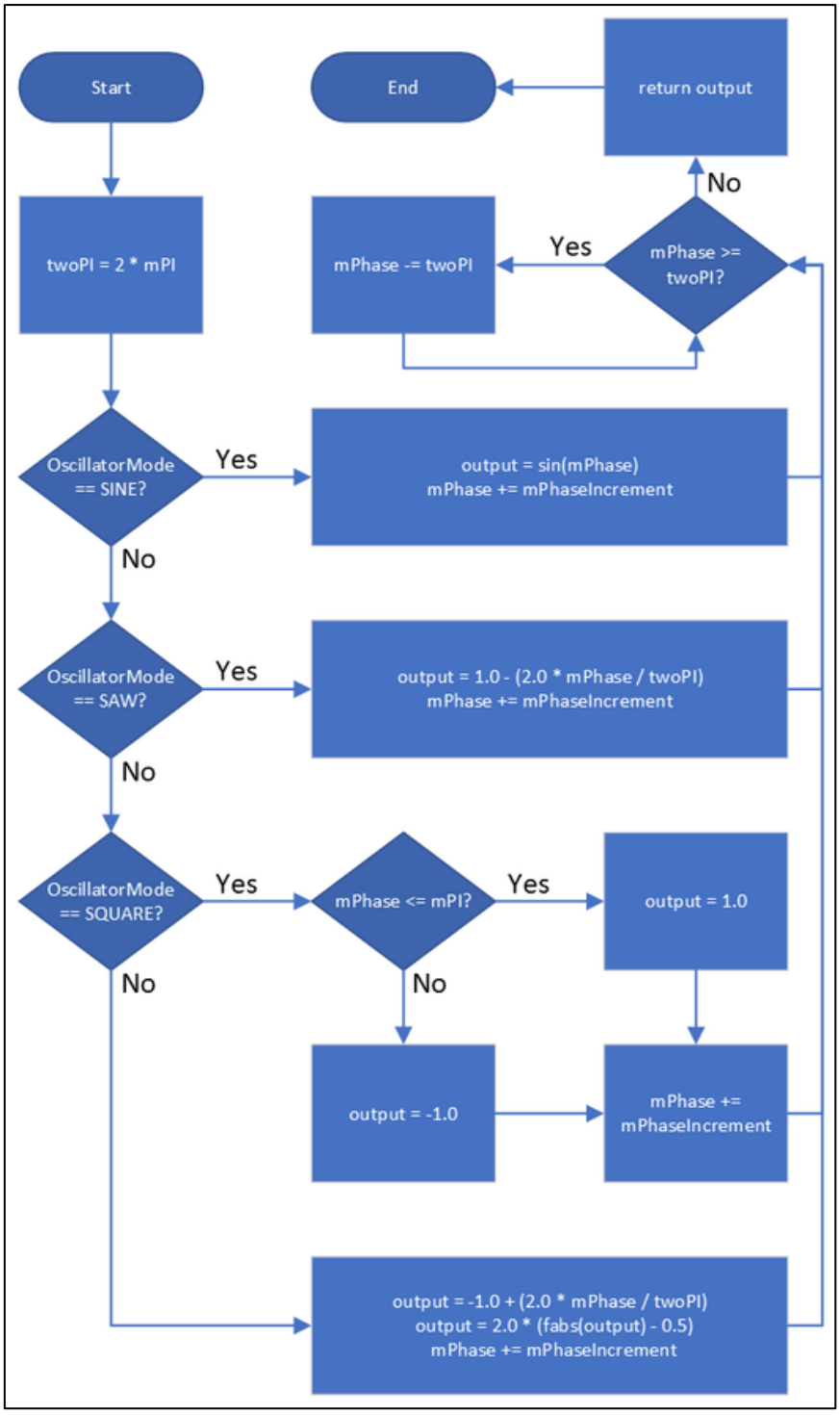
Functions:

- generate – synthesises the audio.
- setMode – sets the mode of the oscillator.
- setFrequency – set the frequency of the oscillator.
- setSampleRate – sets the sample rate.
- updateIncrement – updates mPhaseIncrement.

Variables:

- OscillatorMode – enum, stores oscillator mode we are in.
- mPI – double, stores the value of pi so we don't have to constantly calculate it.
- mFrequency – double, stores the frequency of the oscillator.
- mPhase – double, stores the current phase of the oscillator.
- mSampleRate – double, stores the current sample rate.
- mPhaseIncrement – double, stores how much we increment the phase by each sample.
- output – double, stores the output of the oscillator.

Since the rest of the functions are self-explanatory, I will show how generate function works with a flowchart.



Initially we define twoPI since we will be using it a lot later and it will reduce repeated code. Then we check which waveform we want to synthesise. When sintering each waveform, we use the equations I stated earlier which will produce the correct waveform scaled to fit between 0 -> 2pi and –1 -> 1. After running the calculation, we increment the phase and check if it's greater than 2pi if it is we just minus 2pi resetting the phase back to the start of the waveform. If mPhase < twoPI we just return the output. This function gets called every sample, synthesising the waveform and for our use case, creating the LFO.

Development

I initialised a new project and started implementing the oscillator class. I started by creating all the functions and variables I would need. I also initialised some variables in the header since it could cause issues later. I am also modifying Martin Finke's design slightly since I want to control the mode with a knob which only outputs values not the values in the Enum. I will use a switch statement to select the current mode. I also set up the basic setter functions.

```cpp
#pragma once
#include <math.h>

enum class OscillatorMode
{
    SINE,
    SAW,
    TRIANGLE,
    SQUARE
};

class Oscillator
{
public:
    void setMode(int mode);
    void setFrequency(double frequency);
    void setSampleRate(double sampleRate);
    double generate();

private:
    OscillatorMode mOscillatorMode = (OscillatorMode::SINE);
    const double mPI = (2 * acos(0.0));
    double mFrequency;
    double mPhase = 0.0;
    double mSampleRate;
    double mPhaseIncrement;
    void updateIncrement();
};
```

```cpp
#include "Oscillator.h"

void Oscillator::setMode(int mode)
{
    switch (mode)
    {
    case 0:
        mOscillatorMode = OscillatorMode::SINE;
        break;

    case 1:
        mOscillatorMode = OscillatorMode::SAW;
        break;

    case 2:
        mOscillatorMode = OscillatorMode::TRIANGLE;
        break;

    case 3:
        mOscillatorMode = OscillatorMode::SQUARE;
        break;
    }
}

void Oscillator::setFrequency(double frequency)
{
    mFrequency = frequency;
    updateIncrement();
}

void Oscillator::setSampleRate(double sampleRate)
{
    mSampleRate = sampleRate;
    updateIncrement();
}

void Oscillator::updateIncrement() { mPhaseIncrement = mFrequency * 2 * mPI / mSampleRate; }
```

After this I implemented the generate function with some changes. Since the original implementation fills an output buffer, this isn't useful when I want to multiply the output of the LFO with other values. So, I modified it by setting the output of the waveform calculations to a new double variable 'output'. Here is the completed generate function.

```
double Oscillator::generate()
{
    const double twoPI = 2 * mPI;
    switch (mOscillatorMode)
    {
    case (OscillatorMode::SINE):
        output = sin(mPhase);
        mPhase += mPhaseIncrement;
        while (mPhase >= twoPI)
        {
            mPhase -= twoPI;
        }
        break;

    case (OscillatorMode::SAW):
        output = 1.0 - (2.0 * mPhase / twoPI);
        mPhase += mPhaseIncrement;
        while (mPhase >= twoPI)
        {
            mPhase -= twoPI;
        }
        break;

    case (OscillatorMode::SQUARE):
        if (mPhase <= mPI)
        {
            output = 1.0;
        }
        else
        {
            output = -1.0;
        }
        mPhase += mPhaseIncrement;
        while (mPhase >= twoPI)
        {
            mPhase -= twoPI;
        }
        break;

    case (OscillatorMode::TRIANGLE):
        output = -1.0 + (2.0 * mPhase / twoPI);
        output = 2.0 * (fabs(output) - 0.5);
        mPhase += mPhaseIncrement;
        while (mPhase >= twoPI)
        {
            mPhase -= twoPI;
        }
        break;
    }
    return output;
}
```

Moving back to the main program, I initialised some parameters and variables I would need alongside adding in the OnReset and OnParamChange functions.

```
enum EParams
{
  kDepth,     // How much lfo affects delay time
  kFeedback,  // Feedback
  kWarmth,    // Warmth
  kTone,      // Tone
  kFrequency, // Frequency of Oscillator
  kMix,       // Mix
  kOscMode,   // LFO mode
  kBypass,    // Bypass
  kNumParams
};
```

```
private:
  double mFrequency;
  double mDelay;
  double mDepth;
  double mFeedback = 0.0;
  double mMix;

  int mBypass;
  int mMode;

  double mTone;
  double mWarmth;
```

```
#if IPLUG_DSP // http://bit.ly/2S64BDd
  void ProcessBlock(sample** inputs, sample** outputs, int nFrames) override;
  void OnParamChange(int paramIdx) override;
  void OnReset() override;
  double ToneWarmth(double signal, double warmth);
#endif
};
```

I included the oscillator class in the main program and created a new instance of the class. After this I modified the ProcessBlock function to my preferred state and then I set the output to the oscillator to make sure it's working. I also put the setSampleRate and setFrequency functions in the OnReset and OnParamChange functions respectively.

```
#if IPLUG_DSP
void Flanger::ProcessBlock(sample** inputs, sample** outputs, int nFrames)
{
  const int nChans = NOutChansConnected();

  double* in1 = inputs[0];
  double* in2 = inputs[1];
  double* out1 = outputs[0];
  double* out2 = outputs[1];

  for (int s = 0; s < nFrames; s++) {
    *out1 = mOscillator.generate();
    *out2 = *out1;
  }
}

void Flanger::OnReset()
{
  mOscillator.setSampleRate(GetSampleRate());
}

void Flanger::OnParamChange(int paramIdex)
{
  mFrequency = GetParam(kFrequency)->Value();
  mOscillator.setFrequency(mFrequency);
}
#endif
```

After some initial testing the oscillator was working, I could change the frequency and when I did it moved smoothly. Now with the LFO out the way it's just a case of implementing the delay with the circular buffer class I made earlier. I included the circular buffer class in the main program and created an instance of the class. I set up the buffer the same as when I created my delay so there isn't much to talk about.

```
void Flanger::OnReset()
{
  BL.Reset(GetSampleRate(), 1.0, 0.0);
  mOscillator.setSampleRate(GetSampleRate());
}

void Flanger::OnParamChange(int paramIdex)
{
  mFrequency = GetParam(kFrequency)->Value();
  mDelay = GetParam(kDepth)->Value();
  mFeedback = GetParam(kFeedback)->Value();

  BL.cookVars(mDelay, mFeedback);
  mOscillator.setFrequency(mFrequency);
}
#endif
```

```
#if IPLUG_DSP
void Flanger::ProcessBlock(sample** inputs, sample** outputs, int nFrames)
{
  const int nChans = NOutChansConnected();

  double* in1 = inputs[0];
  double* in2 = inputs[1];
  double* out1 = outputs[0];
  double* out2 = outputs[1];

  for (int s = 0; s < nFrames; s++)
  {
    double LPS = BL.ProcessSample(*in1);

    // *out1 = mOscillator.generate();
    *out1 = (mMix * LPS + (1. - mMix) * *in1);
    *out2 = *out1;
  }
}
```

After testing to see if the delay worked by temporarily linking the depth control to the delay time. The delay did work however the feedback control wasn't working. After debugging my code using breakpoints, I figures out that I was dividing the feedback by 100 twice. After removing the division in OnParamChange it worked fine and I could start linking the two together.

Since I wanted my delay time to modulate between 1 and 20, I needed a way to keep a constant offset of 1 and add on a varying offset. I ended up creating an equation to do this for me; 1 + ((19 * mDepth) * (mOscillator.generate)). When I tested this, I was surprised that it worked however it wasn't fully functional yet since it was creating some errors with the delay buffer. I looked at my code again and realised that the read head was moving in front of the write head at certain points. To fix this, I added 1 to the output of the oscillator and halved it so it stayed positive and had a maximum value of 1. I also moved the (19 * mDepth) into the OnParamChange so I could simplify the processing function.

```
#if IPLUG_DSP
void Flanger::ProcessBlock(sample** inputs, sample** outputs, int nFrames)
{
  const int nChans = NOutChansConnected();

  double* in1 = inputs[0];
  double* in2 = inputs[1];
  double* out1 = outputs[0];
  double* out2 = outputs[1];

  for (int s = 0; s < nFrames; s++)
  {
    double LPS = BL.ProcessSample(*in1);

    mDelay = 1. + (mDepth * (1. + mOscillator.generate()));
    BL.cookVars(mDelay, mFeedback);

    // *out1 = mOscillator.generate();
    *out1 = (mMix * LPS + (1. - mMix) * *in1);
    *out2 = *out1;
  }
}
```

I also noticed an issue with the audio quality when I was modulating the delay time. I was having the same issue with my delay plug-in however since the delay time shouldn't be changing during normal use it didn't seem like an issue however it is here. After a few weeks of researching different solutions to the problem I either had to completely reimplement by circular buffer design with a variable sampling rate or implement a complex smoothing algorithm which I didn't want to do. However, I had an idea to average out the last few samples since that would also smooth out the soundwave reducing the artefacts since when you modulate the delay time each sample won't perfectly flow into each other. I initially tried to average out the last 2 samples however this didn't solve my problem. In a last-ditch effort, I averaged out the past 4 samples and somehow it worked! Although this does contradict [SC5_B], in normal use cases these artifacts won't be noticeable I don't have time to implement a technically perfect solution.

```cpp
#if IPLUG_DSP
void Flanger::ProcessBlock(sample** inputs, sample** outputs, int nFrames)
{
  const int nChans = NOutChansConnected();

  double* in1 = inputs[0];
  double* in2 = inputs[1];
  double* out1 = outputs[0];
  double* out2 = outputs[1];

  for (int s = 0; s < nFrames; s++)
  {
    double LPS = BL.ProcessSample(*in1);

    mDelay = 1. + (mDepth * (1. + mOscillator.generate()));
    BL.cookVars(mDelay, mFeedback);

    // Anti-Aliasting code
    LPS = (LPS + Lps1 + Lps2 + Lps3) / 4;
    Lps3 = Lps2;
    Lps2 = Lps1;
    Lps1 = LPS;


    // *out1 = mOscillator.generate();
    *out1 = (mMix * LPS + (1. - mMix) * *in1);
    *out2 = *out1;
  }
}
```

With the core of the plug-in now working I decided to implement the tone and warmth control since it worked well on the delay plug-in. I imported the filter class again and created a new instance of it. I then implemented the same ToneWarmth function I used in my delay plug-in. I also implemented the bypass switch while I was at it.

```cpp
// Tone Warmth Processing
double Flanger::ToneWarmth(double signal, double warmth)
{

  // Filter Processing
  signal = hsFilter->process(signal);

  // Saturation
  if (warmth > 1.0)
  {
    if (signal > 0)
    {

      signal = 1 - exp(-1 * signal * warmth);
    }
    else
    {

      signal = -1 + exp(signal * warmth);
    }
  }

  return signal;
}
#endif
```

```
for (int s = 0; s < nFrames; s++)
{
  if (mBypass == 1)
  {
    double LPS = BL.ProcessSample(*in1);

    mDelay = 1. + (mDepth * (1. + mOscillator.generate()));
    BL.cookVars(mDelay, mFeedback);

    // Anti-Aliasting code
    LPS = (LPS + Lps1 + Lps2 + Lps3) / 4;
    Lps3 = Lps2;
    Lps2 = Lps1;
    Lps1 = LPS;

    hsFilter->setPeakGain(mTone);
    LPS = ToneWarmth(LPS, mWarmth);

    *out1 = (mMix * LPS + (1. - mMix) * *in1);
    *out2 = *out1;
  }
  else
  {
    *out1 = *in1;
    *out2 = *in2;
  }
}
```

With all this working, all I need to do now is make the plug-in stereo and attach the GUI. To make the plug-in stereo I created another buffer for the right channel and called the same functions for setting variables and resetting the delay as with the left buffer. After giving both channels the exact same processing, I tried to test it however I was getting an error since I was trying to access a memory location I didn't have access to. Since I hadn't changed any code, I didn't know what was going wrong. After again debugging the program using breakpoints, I found out that at some point the read head was getting set to a huge negative value. After a log of trial and error I ended up setting mDelay to 0 in my processing function since after that it would be assigned its actual value. I still don't know why this works but I'm not complaining.

```
void Flanger::ProcessBlock(sample** inputs, sample** outputs, int nFrames)
{
  const int nChans = NOutChansConnected();
  double* in1 = inputs[0];
  double* in2 = inputs[1];
  double* out1 = outputs[0];
  double* out2 = outputs[1];
  mDelay = 1.0;
  for (int s = 0; s < nFrames; s++)
  {
    double LPS = BL.ProcessSample(*in1);
    double RPS = BR.ProcessSample(*in2);

    if (mBypass == 1)
    {
      mDelay = 1. + (mDepth * (0.5 * (1. + mOscillator.generate())));
      BL.cookVars(mDelay, mFeedback);
      BR.cookVars(mDelay, mFeedback);

      // Anti-Aliasting code
      LPS = (LPS + Lps1 + Lps2 + Lps3) / 4;
      Lps3 = Lps2;
      Lps2 = Lps1;
      Lps1 = LPS;

      RPS = (RPS + Rps1 + Rps2 + Rps3) / 4;
      Rps3 = Rps2;
      Rps2 = Rps1;
      Rps1 = RPS;

      hsFilter->setPeakGain(mTone);
      LPS = ToneWarmth(LPS, mWarmth);
      RPS = ToneWarmth(RPS, mWarmth);

      // Mix
      *out1 = (mMix * LPS + (1. - mMix) * *in1);
      *out2 = (mMix * RPS + (1. - mMix) * *in2);
    }
  }
}
```

After some initial testing, my plug-in was now working in stereo, and I could implement the GUI. I modified my GUI template and made this.

All the tests for the plugin will appear later in the document but here is the full code for the main program and distortion class.

Main program code:

```
#pragma once
#include "IPlug_include_in_plug_hdr.h"

const int kNumPresets = 1;

enum EParams
{
  kDepth,     // How much lfo affects delay time
  kFeedback,  // Feedback
  kWarmth,    // Warmth
  kTone,      // Tone
  kFrequency, // Frequency of Oscillator
  kMix,       // Mix
  kOscMode,   // LFO mode
  kBypass,    // Bypass
  kNumParams
};

using namespace iplug;
using namespace igraphics;

class FlangerPlugin final : public Plugin
{
public:
  FlangerPlugin(const InstanceInfo& info);
  ~FlangerPlugin();
```

```
private:
  double mFrequency;
  double mDelay;
  double mDepth;
  double mFeedback = 0.0;
  double mMix;

  int mBypass;
  int mMode;

  double mTone;
  double mWarmth;

  double Lps1 = 0.;
  double Lps2 = 0.;
  double Lps3 = 0.;

  double Rps1 = 0.;
  double Rps2 = 0.;
  double Rps3 = 0.;
```

```
#if IPLUG_DSP // http://bit.ly/2S64BDd
  void ProcessBlock(sample** inputs, sample** outputs, int nFrames) override;
  void OnParamChange(int paramIdx) override;
  void OnReset() override;
  double ToneWarmth(double signal, double warmth);
#endif
};
```

```
#include "FlangerPlugin.h"
#include "IPlug_include_in_plug_src.h"
#include "IControls.h"
#include "CircularBuffer.h"
#include <projects/Oscillator.h>
#include "Biquad.h"

Oscillator mOscillator;
CircularBuffer BL;
CircularBuffer BR;
Biquad* hsFilter = new Biquad();

FlangerPlugin::FlangerPlugin(const InstanceInfo& info)
: Plugin(info, MakeConfig(kNumParams, kNumPresets))
{
  GetParam(kFrequency)->InitDouble("Frequency", 0.2, 0.01, 10.0, 0.01, "Hz", IParam::kFlagsNone, "", IParam::ShapePowCurve(3.));
  GetParam(kDepth)->InitDouble("Depth", 1., 0., 1., 0.01, "%");
  GetParam(kMix)->InitDouble("Mix", 100., 0., 100., 0.01, "%");
  GetParam(kFeedback)->InitDouble("Feedback", 0., 0., 100., 0.01, "%");
  GetParam(kTone)->InitDouble("Tone", 0.0, -4.0, 0.0, 0.01, "");
  GetParam(kWarmth)->InitDouble("Warmth", 1.0, 1.0, 3.0, 0.01, "");

  GetParam(kBypass)->InitInt("Bypass", 1, 0, 1);
  GetParam(kOscMode)->InitInt("mode", 0, 0, 3);

#if IPLUG_EDITOR // http://bit.ly/2S64BDd
  mMakeGraphicsFunc = [&]() {
    return MakeGraphics(*this, PLUG_WIDTH, PLUG_HEIGHT, PLUG_FPS, GetScaleForScreen(PLUG_WIDTH, PLUG_HEIGHT));
  };
```

```
  mLayoutFunc = [&](IGraphics* pGraphics) {

    pGraphics->AttachBackground(GUI_FN);
    pGraphics->LoadFont("Roboto-Regular", ROBOTO_FN);

    // Load images
    const IBitmap depth = pGraphics->LoadBitmap(DEPTH_FN, 128);
    const IBitmap feedback = pGraphics->LoadBitmap(FEEDBACK_FN, 128);
    const IBitmap warmth = pGraphics->LoadBitmap(WARMTH_FN, 128);
    const IBitmap tone = pGraphics->LoadBitmap(TONE_FN, 128);
    const IBitmap frequency = pGraphics->LoadBitmap(FREQUENCY_FN, 128);
    const IBitmap mix = pGraphics->LoadBitmap(MIX_FN, 128);

    const IBitmap mode = pGraphics->LoadBitmap(MODE_FN, 4);
    const IBitmap bypass = pGraphics->LoadBitmap(BYPASS_FN, 2);

    // Attach controls

    pGraphics->AttachControl(new IBKnobControl(55, 33, depth, kDepth));
    pGraphics->AttachControl(new IBKnobControl(204, 33, feedback, kFeedback));
    pGraphics->AttachControl(new IBKnobControl(55, 136, warmth, kWarmth));
    pGraphics->AttachControl(new IBKnobControl(204, 136, tone, kTone));
    pGraphics->AttachControl(new IBKnobControl(55, 239, frequency, kFrequency));
    pGraphics->AttachControl(new IBKnobControl(204, 239, mix, kMix));

    pGraphics->AttachControl(new IBSwitchControl(199, 344, mode, kOscMode));
    pGraphics->AttachControl(new IBSwitchControl(211, 392, bypass, kBypass));

  };
#endif
```

```
#if IPLUG_DSP
void FlangerPlugin::ProcessBlock(sample** inputs, sample** outputs, int nFrames)
{
  const int nChans = NOutChansConnected();

  double* in1 = inputs[0];
  double* in2 = inputs[1];
  double* out1 = outputs[0];
  double* out2 = outputs[1];

  mDelay = 1.0;

  for (int s = 0; s < nFrames; s++, ++in1, ++in2, ++out1, ++out2)
  {
    double LPS = BL.ProcessSample(*in1);
    double RPS = BR.ProcessSample(*in2);

    if (mBypass == 1)
    {
      mDelay = 1. + (mDepth * (0.5 * (1. + mOscillator.generate())));
      BL.cookVars(mDelay, mFeedback);
      BR.cookVars(mDelay, mFeedback);

      // Anti-Aliasting code
      LPS = (LPS + Lps1 + Lps2 + Lps3) / 4;
      Lps3 = Lps2;
      Lps2 = Lps1;
      Lps1 = LPS;
```

```
      RPS = (RPS + Rps1 + Rps2 + Rps3) / 4;
      Rps3 = Rps2;
      Rps2 = Rps1;
      Rps1 = RPS;


      hsFilter->setPeakGain(mTone);
      LPS = ToneWarmth(LPS, mWarmth);
      RPS = ToneWarmth(RPS, mWarmth);

      // Mix
      *out1 = (mMix * LPS + (1. - mMix) * *in1);
      *out2 = (mMix * RPS + (1. - mMix) * *in2);
    }
    else
    {
      *out1 = *in1;
      *out2 = *in2;
    }
  }
}
```

```cpp
void FlangerPlugin::OnParamChange(int paramIdx)
{
  mFrequency = GetParam(kFrequency)->Value();
  mMix = (GetParam(kMix)->Value()) / 200.;
  mDepth = (GetParam(kDepth)->Value() * 19.);
  mTone = (GetParam(kTone)->Value());
  mWarmth = GetParam(kWarmth)->Value();
  mBypass = GetParam(kBypass)->Value();
  mMode = GetParam(kOscMode)->Value();
  mFeedback = GetParam(kFeedback)->Value();

  mOscillator.setMode(mMode);
  mOscillator.setFrequency(mFrequency);
}

void FlangerPlugin::OnReset()
{
  mOscillator.setSampleRate(GetSampleRate());
  BL.Reset(GetSampleRate(), 1.0, 0.0);
  BR.Reset(GetSampleRate(), 1.0, 0.0);
  hsFilter->setBiquad(bq_type_highshelf, 3500 / GetSampleRate(), 0.707, 0);
}

// Tone Warmth Processing
double FlangerPlugin::ToneWarmth(double signal, double warmth)
{

  // Filter Processing
  signal = hsFilter->process(signal);
```

```cpp
  // Saturation
  if (warmth > 1.0)
  {
    if (signal > 0)
    {

      signal = 1 - exp(-1 * signal * warmth);
    }
    else
    {

      signal = -1 + exp(signal * warmth);
    }
  }

  return signal;
}
#endif

FlangerPlugin::~FlangerPlugin()
{
  BL.Destroy();
  BR.Destroy();
}
```

Oscillator class code:

```h
#pragma once
#include <math.h>

enum class OscillatorMode
{
  SINE,
  SAW,
  TRIANGLE,
  SQUARE
};

class Oscillator
{
public:
  void setMode(int mode);
  void setFrequency(double frequency);
  void setSampleRate(double sampleRate);
  double generate();

private:
  OscillatorMode mOscillatorMode = (OscillatorMode::SINE);
  const double mPI = (2 * acos(0.0));
  double mFrequency;
  double mPhase = 0.0;
  double mSampleRate;
  double mPhaseIncrement;
  void updateIncrement();
  double output;
};
```

```cpp
#include "Oscillator.h"

void Oscillator::setMode(int mode)
{
  switch (mode)
  {
  case 0:
    mOscillatorMode = OscillatorMode::SINE;
    break;

  case 1:
    mOscillatorMode = OscillatorMode::SAW;
    break;

  case 2:
    mOscillatorMode = OscillatorMode::TRIANGLE;
    break;

  case 3:
    mOscillatorMode = OscillatorMode::SQUARE;
    break;
  }
}

void Oscillator::setFrequency(double frequency)
{
  mFrequency = frequency;
  updateIncrement();
}
```

```cpp
void Oscillator::setSampleRate(double sampleRate)
{
    mSampleRate = sampleRate;
    updateIncrement();
}

double Oscillator::generate()
{
    const double twoPI = 2 * mPI;
    switch (mOscillatorMode)
    {
    case (OscillatorMode::SINE):
        output = sin(mPhase);
        mPhase += mPhaseIncrement;
        while (mPhase >= twoPI)
        {
            mPhase -= twoPI;
        }
        break;

    case (OscillatorMode::SAW):
        output = 1.0 - (2.0 * mPhase / twoPI);
        mPhase += mPhaseIncrement;
        while (mPhase >= twoPI)
        {
            mPhase -= twoPI;
        }
        break;
```

```
    case (OscillatorMode::SQUARE):
        if (mPhase <= mPI)
        {
            output = 1.0;
        }
        else
        {
            output = -1.0;
        }
        mPhase += mPhaseIncrement;
        while (mPhase >= twoPI)
        {
            mPhase -= twoPI;
        }
        break;

    case (OscillatorMode::TRIANGLE):
        output = -1.0 + (2.0 * mPhase / twoPI);
        output = 2.0 * (fabs(output) - 0.5);
        mPhase += mPhaseIncrement;
        while (mPhase >= twoPI)
        {
            mPhase -= twoPI;
        }
        break;
    }
    return output;
}

void Oscillator::updateIncrement() { mPhaseIncrement = mFrequency * 2 * mPI / mSampleRate; }
```

As stated before, all the code is available on my Github profile and may have been updated since the creation of this document.

Bibliography:

1. Recordingology, “Comb FIlter Calculations”
2. Martin Finke, “Making Audio Plugins Part 8: Synthesizing Waveforms”

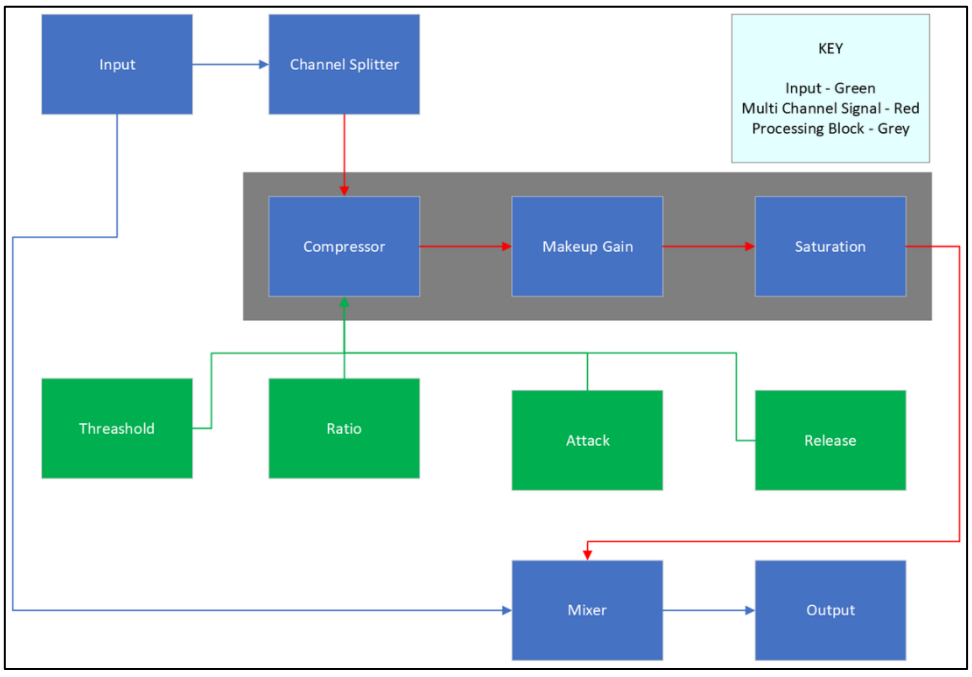
Compressor Outline

Design

Compression is an effect which controls the volume of a given signal. You set a threshold where you want to start compressing the signal. When the signal reaches this threshold, it will reduce the gain of the signal according to a given ratio. The speed at which it enters and returns from this state is called the Attack and Release. In music production compression can be found almost every instrument since it is a great tool for normalising the level of a signal. Alongside this, compressors have a sidechain input which allows a different signal to determine whether to compress the signal.

Compression was originally created to help control the volume of Radio and TV presenters since they could go from whispering to talking loudly very quickly, requiring someone to manually change the volume of their voice. As time went on, more traditional compressors were created and with them the introduction of more controls to dial in the sound you wanted. Around this time sidechain compression was introduced which would later create its own effect 'sidechain' which is prominent in modern EDM. Finally, multi-band compression was introduced which essentially had multiple compressors on one channel, each listening to their own frequency band and applying compression accordingly.

I plan to create a single-band compressor with built-in saturation. It will have controls for; Threshold, Ratio, Attack, Release, Makeup Gain, and a Dry/Wet control. Since I am emulating a guitar pedal compressor, I won't include a multi-band mode or a sidechain input simplifying the design in accordance with [SC1]. I have created a signal diagram which shows all the different modules I need to program and the interplay between them.



Implementation

As with my delay and flanger plug-in, the input is sent to two processing chains, one straight to a mixer for Dry/Wet control and the other to the compression algorithm. There aren't many modules in this implementation since the 'Compressor' module is just one algorithm with five inputs. After it is sent through the compression algorithm some makeup gain can be applied since compression can reduce the volume of the input and finally this is sent through a saturator to add some subtle warmth before heading into the Dry/Wet mixer.

Compression

Like most audio equipment, when dealing with levels you work with dB. However, dB doesn't only refer to volume, dB is used to denote that you are using a logarithmic scale. For music we generally use dBV where the V stands for volts, owing to electronics working in volts. After some trial and a large amount of error, I realised that I should work in amplitude for reasons I will cover later. So, I will only need to convert from dBV to amplitude to convert my controls into a usable form.

To go from dBV to amplitude I can use the equation $y = 10^{\left(\frac{x}{20}\right)}$where x is the threshold in dBV and y is the amplitude. To convert from amplitude to dBV you use the equation $y = 20 \cdot \log_{10}(x)$ where is your input signal and is your output in dBV. Both equations and a good explanation of this is provided by Demofox on their blog [1].

After I have completed the conversion, I can start processing the input. Firstly, we compare if the input signal is greater than the threshold if it is we start processing the signal. There are three core parts to this stage of the processing chain; Ratio, Attack and Release. Ratio determines the amount of gain reduction applied to the signal. Ratio is the relationship between the input and output signal, if the ratio is set to 3 then when the input increased by 3dBV then the output only increased by 1dBV. If the ratio is set to 1 then no compression is applied. My ratio control will be notched, meaning that it can only be changed to set values; 1, 2, 3, 5, 10, 20, 30. Attack determines how long it takes to reach this amount of gain reduction. Release determines how long it takes to return to a normal level when the input has dropped below the threshold.

ProcessCompressor class

Since the compression algorithm can be abstracted to a separate class I shall do so in accordance with [SC7_A]. Here I will outline the different functions and variables that I'll need to create the class and their purpose.

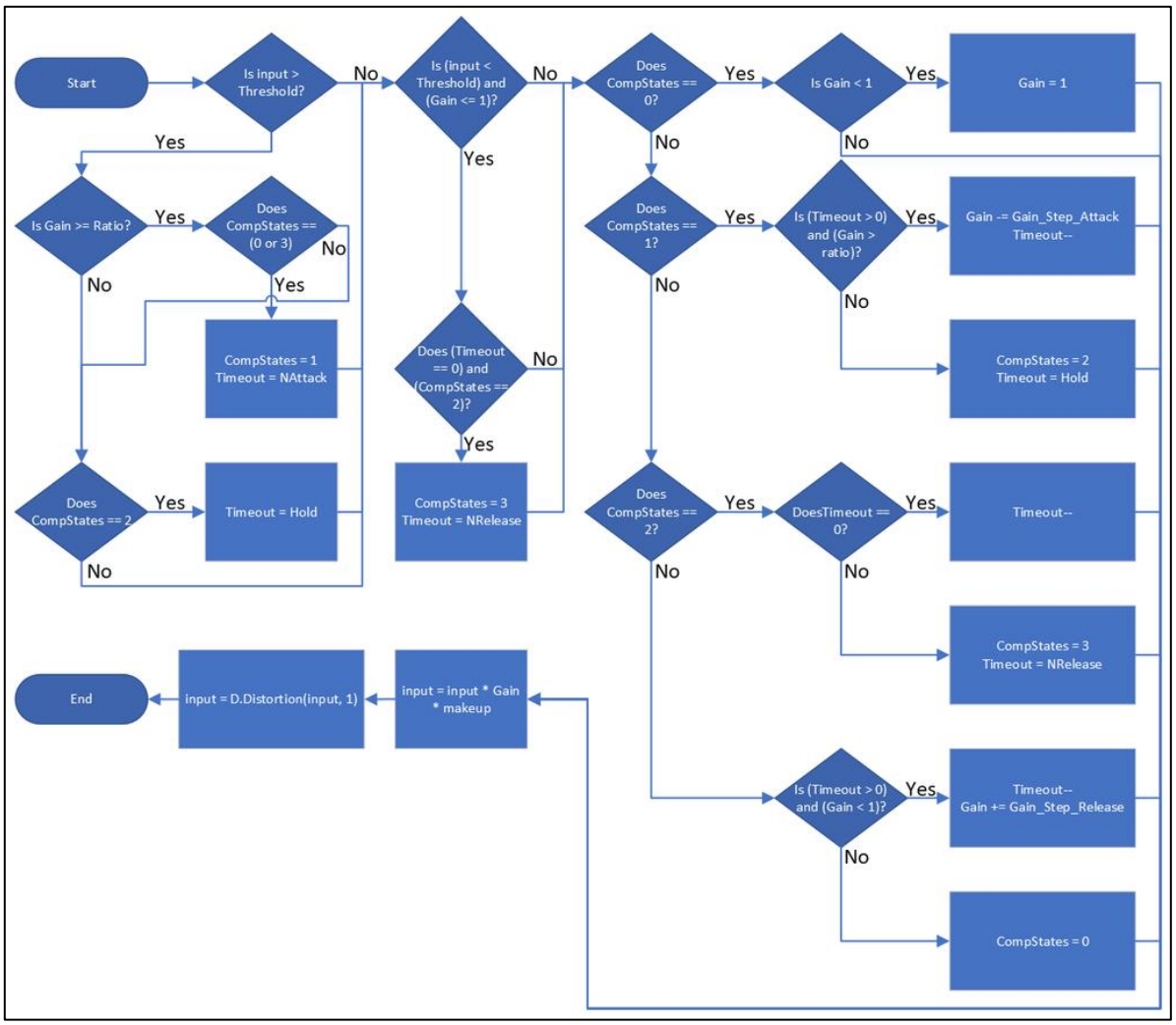
Functions:

- Compression – Processes incoming audio using passed parameters to determine compression

Variables:

- Hold – Integer, used to store how many samples we should stay in the hold stage before checking if the input is above the threshold.
- Timeout – Integer, used to store the value of a timer, which reduces by 1 every sample.
- NAttack – Integer, used to store how many samples long the attack is.
- NRelease – Integer, used to store how many samples long the release is.
- Gain – Double, used to store the amount of gain reduction being applied to the input.
- Gain_Step_Attack – Double, used to store how much the gain should increment each sample during the attack stage.
- Gain_Step_Release – Double, used to store how much the gain should increment each sample during the release stage.
- CompStates – Double, used to store which stage of the program we are in.

Since there is only one function, I have created a flowchart on how the compression algorithm works.



Although it looks very complicated, it's just the rules on how to transition between each state of compression. The four states are; No Gain Reduction, Attack, Gain Reduction and Release. Firstly, I'll explain what each stage does then I'll explain the conditions on how we transition to a new state.

- No Gain Reduction: In this state there is no compression on the input signal.
- Attack: In this state we are transitioning from an uncompressed signal to a compressed signal.
- Gain Reduction: In this state there is compression on the input signal.
- Release: In this state we are transitioning from a compressed signal to an uncompressed signal.

We start in the 'No Gain Reduction' stage, to move to the attack stage we need to meet two conations; [input > threshold] and [gain > ratio]. This means that we have exceeded the threshold and the ratio will influence the signal. Since gain is set to 1 during this stage if the ratio = 1 then there is no point going to through the rest of the stages since there will be no compression anyway.

When we move to the 'Attack' stage, we set Timeout to NAttack. Since Timeout acts as a timer, we set it to NAttack which is the length of out attack in samples. This allows us to reduce the gain by a fixed amount, reaching the desired value when the timer runs out. Each sample the Timeout is reduced by 1 and Gain_Step_Attack is subtracted from gain creating the transition. To move to the 'Gain Reduction' stage we can meet either of these conditions [Timeout == 0] or [gain == ratio]. The first condition says that the timer has finished, and we can move onto the next stage. The second condition says we have reached the desired gain so we can move onto the next stage, the reason for this condition will become apparent later.

When we move to the 'Gain Reduction' stage, we set Timeout to Hold. We have a timer in this state owing to how sound works. Since a sound oscillates between –1 and 1, we could be above the threshold one sample but bellow on the next. This timer allows us to keep a steady amount of compression for a set period before checking if we need to return to this state, reducing artifacts in accordance with [SC5_B]. Once [Timeout == 0] we can transition to the next stage 'Release'.

Finally, when we reach the 'Release' stage we set Timeout to NRelease allowing us to smoothly transition to an uncompressed signal. From this stage we can either transition back to the 'Attack' stage or to the 'No Gain Reduction' stage. To transition to the 'Attack' stage we need to meet two conditions [input > threshold] and [gain > ratio]. We do this for the same reason as transitioning from 'No Gain Reduction' to 'Attack' since during the release stage we could exceed the threshold again. To transition to the 'No Gain Reduction' stage, we can meet either of these conditions [Timeout == 0] or [gain == 1]. For the first condition, the timer has finished and we have returned to an uncompressed signal. For the second condition, we have already returned to an uncompressed signal.

All this processing allows us to determine what value Gain should be. To apply the compression and makeup gain to our signal we simply multiply the three values together. Finally, to apply saturation to out signal we run it through the Distortion function from the ProcessDistortion class we created earlier. We pass the signal and 1 into the function since we want to use the soft-clipping mode of the function. This completes the compression algorithm.

Audio Examples of compression

In most cases compression is an effect that is felt not heard, this makes it hard to point out where it is being used unless it's supposed to be noticeable. Owing to this I have created some examples which should help you understand compression better.

Firstly, we will start off with clean guitar riff [See Compression_Example_1]. As you can hear, there is a lot of dynamic contrast in this riff. To make it obvious, I will use a low threshold of –20dB and a high ratio of 10 with some makeup gain to match the level to the clean signal [See Compression_Example_2]. As we can here there is way less dynamic contrast in this riff however the start of each transient is being squashed which could be undesirable. To fix this we increase the attack on the compressor, allowing more time for the transient to play before the compression kicks in [See Compression_Example_3]. Finally, after we dip below the threshold, I want to return to a normal state since the compressor is creating a pumping effect. To fix this I will reduce the release on the compressor making it return to a state of no compression faster [See Compression_Example_4]. If that wasn't clear, then I highly recommend you check out a video on compression by Hyperbits [2].

Development

So, I started as usual by creating a new project, setting up controls for any parameters I want to control and creating a new class called ProcessCompression. One thing to note is that this algorithm isn't the final algorithm described in this document. I ended up reworking the algorithm since I already had most of the core ideas in my design and it was simpler to modify it than start again.

I started by setting up the header file, I created all the variables and functions I would need. Compression contained the compression algorithm. UpdateVars would be called every sample to update the variables within the class (With hindsight I could have just passed all of these into the Compression function). Interpolation was a linear interpolation algorithm. InterpolationSamples was used to calculate the number of samples for whichever state was passed into it (Attack or Release). Setup was called when the program was initialised to load the sample rate into the class (Again, in hindsight this could have been put in a constructor).

```
#include "ProcessDistortion.h"
#include "IPlugUtilities.h"

#pragma once
class ProcessCompression
{
public:

    double Compression(double input);
    void UpdateVars(double threshold, double ratio, double attack, double release, double makeup);
    double Interpolation(double a, double b, double f);
    int InterpolationSamples(double stage);
    void SetUp(double sampleRate);

private:
    double Threshold;
    double Attack;
    double Release;
    int Ratio;
    double Makeup;

    double Signal;
    double CSignal;
    double FSignal;
    int CStage = 0;
    double SampleRate;
    int SAttack;
    int SRelease;
    int PAttack;
    int PRelease;
};
```

Apart from the obviously named controls, Signal was used to store the input to Compression. CSignal stored the compressed signal. FSignal stored the final signal. CStage stored what stage of compression we were on. SAttack and SRelease stored how long each stage was in samples. PAttack and PRelease were used to store how far into each stage we were. I started by implementing the simple functions, since they would be used within the main Compression function.

```
void ProcessCompression::UpdateVars(double threshold, double ratio, double attack, double release, double makeup)
{
    Threshold = threshold;
    Ratio = ratio;
    Attack = attack;
    Release = release;
    Makeup = makeup;
}

double ProcessCompression::Interpolation(double a, double b, double f) { return ((b - a) * f + a); }

void ProcessCompression::SetUp(double sampleRate) { SampleRate = sampleRate; }

int ProcessCompression::InterpolationSamples(double stage) { return SampleRate * stage; }
```

Following this, I implemented the whole of the Compression function in one go. Since I had already created a flowchart, I programmed the whole thing since I knew what to do without testing any of it. This was silly since I might have realised the flaw in this algorithm earlier. However, I started by using a switch statement to determine what Ratio should be set to since when I originally designed the algorithm, I wanted to have a notched ratio control.

```
double ProcessCompression::Compression(double input)
{
    // Ratio Switch statement
    switch (Ratio)
    {
    case 0:
        // 1:1
        Ratio = 1;
        break;

    case 1:
        // 2:1
        Ratio = 0.5;
        break;

    case 2:
        // 3:1
        Ratio = 0.3333;
        break;

    case 3:
        // 5:1
        Ratio = 0.2;
        break;

    case 4:
        // 10:1
        Ratio = 0.01;
        break;

    case 5:
        // 20:1
        Ratio = 0.02;
        break;

    case 6:
        // 30:1
        Ratio = 0.0333;
        break;
    }
```

I calculated the values by doing [1/(ratio)] for example a 5:1 ratio would give me 0.2. After this, I started to program the compression algorithm. Since I commented everything out, I won't explain how it works but imagine it as a slightly worse version of my final algorithm.

```
// Assign input to Signal
Signal = input;

// Is Signal > Threshold?
if (Signal > Threshold)
{
    // Run Ratio Calculation
    CSignal = Signal - ((Signal - Threshold) + ((Signal - Threshold) / Ratio));

    // Does Attack need to be calculated?
    if (CStage == 0)
    {
        // Calculate attack length in samples if interpolation isn't already running
        if (PAttack == 0)
        {
            SAttack = InterpolationSamples(Attack);
        }

        // Attack interpolation
        FSignal = Interpolation(Signal, CSignal, ((PAttack / SAttack) * 100));
        PAttack++;

        // Has interpolation finished?
        if (PAttack == SAttack)
        {
            PAttack = 0;
            CStage = 1;
        }
    }
    else
    {
        // Sustained Compression Stage
        FSignal = CSignal;
    }
}
```

```
else
{
    // Does Release need to be calculated?
    if (CStage = 1)
    {
        // Calculate release length in samples if interpolation isn't already running
        if (PRelease == 0)
        {
            SRelease = InterpolationSamples(Release);
        }

        // Release interpolation
        FSignal = Interpolation(Signal, CSignal, ((PRelease / SRelease) * 100));
        PRelease++;

        // Has interpolation finished?
        if (PRelease == SRelease)
        {
            PRelease = 0;
            CStage = 0;
        }
    }
    else
    {
        // No compression needs to be applied
        FSignal = Signal;
    }
}

// Perform Makeup Gain
FSignal += Makeup;

// Convert back to Amplitude
FSignal = pow(10, (FSignal / 20.0));
```

```
// Add Saturation
FSignal = D.Distortion(FSignal, 1);

return FSignal;
```

When I tested the build for the first time, I noticed a few things; when I changed the ratio control no compression was being applied, the attack, release and makeup controls didn't do anything, and no saturation was being applied. So basically, nothing worked. The first issue I found was that I added too many brackets in the ratio calculation messing up the whole thing. Here is the fix:

```
// Run Ratio Calculation
CSignal = Signal - (Signal - Threshold) + ((Signal - Threshold) / Ratio);
```

After realising that this was a complete failure, I decided to strip my program back to the basics and test everything individually, gradually building back up to my finished program. I started by testing Threshold and Ratio. At this point, the code looked like this:

```
// Assign input to Signal
Signal = input;

// Is Signal > Threshold?
if (Signal > Threshold)
{
    // Run Ratio Calculation
    FSignal = 0
}
else
{
    FSignal = Signal;
}
return FSignal;
```

Initially, when testing on a since wave the threshold didn't work, and the sinewave sounded like it was an octave up. Going back to me code I realised I should be taking an absolute value of my signal since when the signal is < 0 it will always be less than the threshold. This created the illusion of the sine wave sounding an octave up since I could only hear half of the waveform. However, this only partially fixed the problem since the threshold still wasn't working. After looking back through my code, I realised that threshold was still in dBV when my signal was in amplitude. To fix this I modified the UpdateVars function, updating Makeup as well.

```
void ProcessCompression::UpdateVars(double threshold, double ratio, double attack, double release, double makeup)
{
    Threshold = 10*pow(10, (threshold/20));
    Ratio = ratio;
    Attack = attack;
    Release = release;
    Makeup = 10*pow(10, (makeup/20));
}
```

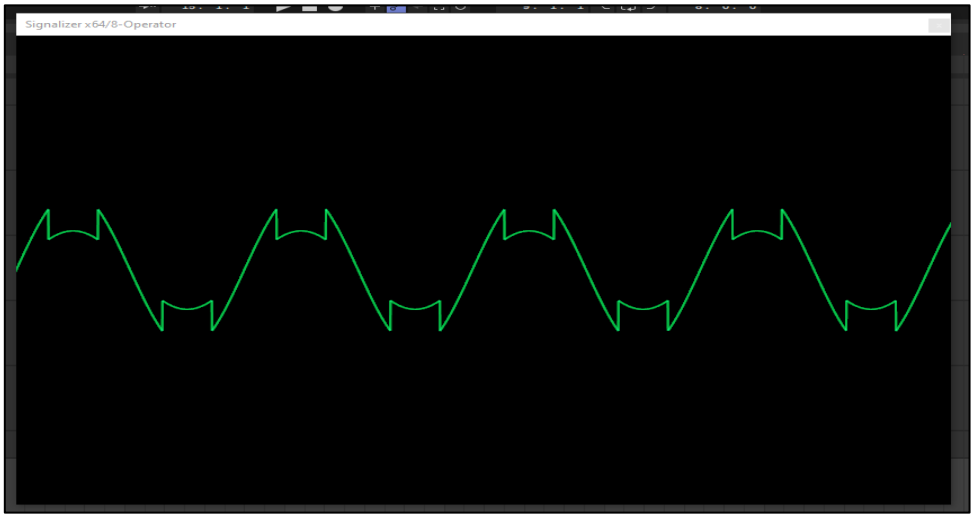
This did fix the threshold allowing me to move on to implementing the ratio control. I realised that I was calculating my ratio values incorrectly. To calculate the correct values, you run the amount of gain reduction you want e.g. -2 through the dBV to Amplitude equation stated earlier, in this case it yields 0.7079.

```
// Ratio Switch statement
switch (Ratio)
{
case 0:
    // 1:1
    Ratio = 1;
    break;

case 1:
    // 2:1
    Ratio = 0.7943;
    break;

case 2:
    // 3:1
    Ratio = 0.7079;
    break;

case 3:
    // 5:1
    Ratio = 0.5623;
    break;

case 4:
    // 10:1
    Ratio = 0.3162;
    break;

case 5:
    // 20:1
    Ratio = 0.1;
    break;

case 6:
    // 30:1
    Ratio = 0.0316;
    break;
}
```

I also realised that my ratio calculation was working in dBV not amplitude, so I promptly fixed it. Luckily to work in amplitude you just multiply the two values together.

```
// Run Ratio Calculation
FSignal = Signal * Ratio;
```

However, when I tested this new version all I heard was silence. Looking back at the code I realised that Ratio was an integer, and I was trying to set it to a double. To fix this I created a new double called RatioC which would store the calculated ratio value. I updated all the instances of Ratio inside the switch statement and in the ratio calculation accordingly. After again updating the code and testing the algorithm I finally hear some form of compression. However, it wasn't what I was expecting so I checked the sound out on an oscilloscope. What I saw was this:



After looking back through the code, everything was working just, not as I intended. I wanted it to compress the whole signal when it reached the threshold not just one section of each oscillation. At this point I immediately thought about implementing an RMS (Root Mean Square) detector since it would take an average of x samples to determine the average volume. RMS is a way of determining the average volume of a signal. To calculate it you square root the average of your samples squared. After some thought I could use a circular buffer to store x samples and when I want to calculate RMS, I just sum the values in this buffer and square root them. Here is the code for the RMS detector.

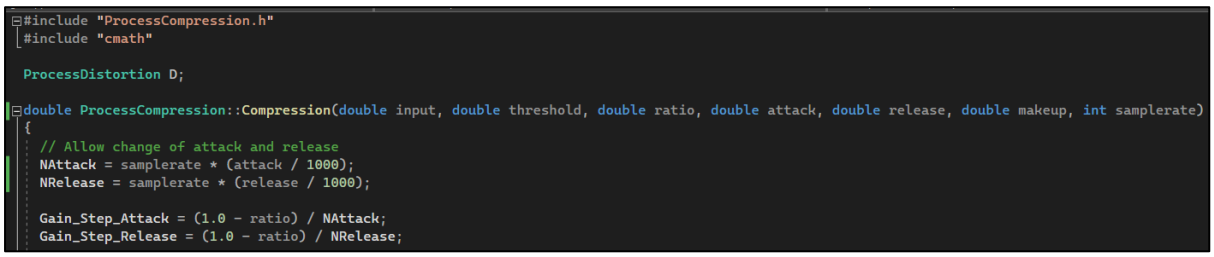
```
// RMS Stuff
// Wrap pointer around buffer
if (RMSPointer != 1323)
{
  RMSPointer++;
}
else
{
  RMSPointer = 0;
}

// Perform RMS calculation
RMSTotal = 0.0;
for (int i = 0; i < 1323; i++)
{
  RMSTotal += pow(RMSBuffer[i], 2);
}
RMS = sqrt(RMSTotal / 1323);

// Assign input to Signal and convert to dBV
Signal = input;

if (RMS > Threshold)
{
```

RMSBuffer as an array of size 1323 which stores the squared values. RMSPointer points to the next index where we want to save a value. RMSTotal stores the sum of the array. RMS stores the actual RMS of the array. (Again, in hindsight I could have implemented this more efficiently). After making these changed to my algorithm I tested it out. To my surprise it did work. However, since it was performing thousands of calculations per sample it was horribly inefficient and my laptop couldn't even compute the compression in real time. I ended up changing the array size to 100 and even then, it was using up over 40% of my CPU. This would completely violate [SC0_A] and [SC0_C].

Finally, at this point I realised that it would be better to rethink my algorithm from the ground up when compared to fixing my current implementation. Which lead me to create the algorithm I presented earlier. I saved a copy of my old algorithm for later reference and started again. I initially set up the header file with all the functions and variables I needed.

```
#include "ProcessDistortion.h"
#include "IPlugUtilities.h"

#pragma once
class ProcessCompression
{
public:

  double Compression(double input, double threshold, double ratio, double attack, double release, double makeup, int samplerate);

private:
  int Hold = 1000;
  int Timeout;
  int NAttack;
  int NRelease;
  double Gain = 1.0;
  double Gain_Step_Attack;
  double Gain_Step_Release;
  int CompStates = 0;
};
```

I then started to program the compression algorithm. Since the attack and release could change every sample, I need to calculate NAttack, NRelease, Gain_Step_Attack and Gain_Step_Release before carrying out any other calculations. To calculate NAttack and NRelease, I first convert attack and release into seconds and then multiply them with the sample rate. To calculate Gain_Step_Attack and Gain_Step_Release I divide the difference between 1 (no compression) and the calculated ratio value with NAttack or NRelease to give me how much to increase / decrease the gain by each sample during their respective states.



```
#include "ProcessCompression.h"
#include "cmath"

ProcessDistortion D;

double ProcessCompression::Compression(double input, double threshold, double ratio, double attack, double release, double makeup, int samplerate)
{
  // Allow change of attack and release
  NAttack = samplerate * (attack / 1000);
  NRelease = samplerate * (release / 1000);

  Gain_Step_Attack = (1.0 - ratio) / NAttack;
  Gain_Step_Release = (1.0 - ratio) / NRelease;
```

Following this I implemented the state checker; this will calculate which stage of compression we should be in. The first if statement checks whether we have exceeded the threshold. If we have then we check if gain >= ratio. If it is, we haven't reached a compressed state yet and we need to check which state we are in. If we are in 'No Gain Reduction' (0) or 'Release' (3) then we can transition into the attack state. We do this by setting CompStates to 1 and setting the timeout to NAttack. If our gain is less than the ratio and we have still exceeded the threshold, we check if we are in the 'Gain Reduction' state. If we are in this state, we set Timeout to Hold.

If we haven't exceeded the threshold and gain <= equal to 1, we check if Timeout == 0 and we are in the 'Gain Reduction' state. If so, we can transition into the 'Release' state by setting CompStates to 3 and setting Timeout to NRelease.

```
// Stage checker
if (fabs(input) > threshold)
{
  if (Gain >= ratio)
  {
    if (CompStates == 0 || CompStates == 3)
    {
      CompStates = 1;
      Timeout = NAttack;
    }
  }
  if (CompStates == 2)
  {
    Timeout = Hold;
  }
}
if (fabs(input) < threshold && Gain <= 1.0)
{
  if (Timeout == 0 && CompStates == 2)
  {
    CompStates = 3;
    Timeout = NRelease;
  }
}
```

Now we know what state we are in; we can carry out the following calculations to compress our signal. The simplest state is 'No Gain Reduction' since we just set gain to 1, because when we multiply it with our signal there will be no change in volume.

The next state is 'Attack', we initially check if Timeout is > 0 and our gain is greater than our ratio. If we meet these conditions, then we decrease the gain by Gain_Step_Attack and decrease Timeout by 1. If we haven't met these conditions, then the 'Attack' state has finished, and we can move to 'Gain Reduction' by setting CompStates to 2 and setting Timeout to Hold.

```
// Compression calculation
switch (CompStates)
{
case 0:
  // No Gain Reduction
  if (Gain < 1.0)
  {
    Gain = 1.0;
  }
  break;

case 1:
  // Attack
  if (Timeout > 0 && Gain > ratio)
  {
    Gain -= Gain_Step_Attack;
    Timeout--;
  }
  else
  {
    CompStates = 2;
    Timeout = Hold;
  }
  break;
```

When we are in the 'Gain Reduction' state, we check if Timeout > 0. If it is, then we decrease Timeout by 1. Otherwise, we have finished this state and can move onto the 'Release' state. We do this by setting CompStates to 3 and setting the Timeout to NRelease.

When we are in the 'Release' state, we check if Timeout > 0 and Gain < 1. If we meet these conditions, we decrease Timeout by 1 and increase the gain by Gain_Step_Release. Otherwise, we have finished the 'Release' stage and we can move back to 'No Gain Reduction'.

```
case 2:
    // Gain Reduction
    if (Timeout > 0)
    {
        Timeout--;
    }
    else
    {
        CompStates = 3;
        Timeout = NRelease;
    }
    break;

case 3:
    // Release
    if (Timeout > 0 && Gain < 1.0)
    {
        Timeout--;
        Gain += Gain_Step_Release;
    }
    else
    {
        CompStates = 0;
    }
    break;
}
```

Finally, we can apply compression and makeup gain to the input by multiplying all the values together. After this we pass our input into the Distortion function from the ProcessDistortion class to apply Soft-Clipping to the signal. After this we can return out signal and we have finished compressing the signal.

```
    // Apply Compression and Makeup Gain
    input = input * Gain * makeup;
    // Apply Saturation
    input = D.Distortion(input, 1);

    return input;
}
```

After some initial testing and minor bug fixes, everything seemed to work, and I could start implementing the final GUI. But before this I'll quickly explain how this is being implemented inside the main program. I started by including ProcessCompression in the main program and creating an instance of the class.

```
#include "CompressorPlugin.h"
#include "IPlug_include_in_plug_src.h"
#include "IControls.h"
#include "ProcessCompression.h"

ProcessCompression C;
```

Aside from the usual setup of controls which I have explained in my section on how iPlug2 works, I finalised the ProcessBlock function implementing my compression algorithm, mix control and bypass switch. The mix control was implemented using linear interpolation between the raw input signal and the compressed signal.

```
#if IPLUG_DSP
void CompressorPlugin::ProcessBlock(sample** inputs, sample** outputs, int nFrames)
{
  // Update params
  const int nChans = NOutChansConnected();

  double* in1 = inputs[0];
  double* in2 = inputs[1];

  double* out1 = outputs[0];
  double* out2 = outputs[1];
  double Threshold = DBToAmp(GetParam(kThreshold)->Value());
  double Ratio = DBToAmp(GetParam(kRatio)->Value());
  double Attack = GetParam(kAttack)->Value();
  double Release = GetParam(kRelease)->Value();
  double Makeup = DBToAmp(GetParam(kMakeup)->Value());
  double Samplerate = GetSampleRate();

  double mix = GetParam(kMix)->Value() / 100;
  bool bypass = GetParam(kBypass)->Value();

  for (int s = 0; s < nFrames; ++s, ++in1, ++in2, ++out1, ++out2)
  {
    if (bypass)
    {
      // Mix
      *out1 = ((mix * (C.Compression(*in1, Threshold, Ratio, Attack, Release, Makeup, Samplerate))) + (1 - mix) * *in1);
      *out2 = ((mix * (C.Compression(*in2, Threshold, Ratio, Attack, Release, Makeup, Samplerate))) + (1 - mix) * *in2);
    }
    else
    {
      *out1 = *in1;
      *out2 = *in2;
    }

  }
}
#endif
```

Finally, I created a new GUI from my blender template and attached all the images to the controls leaving me with the result.

Here is all the code for the compressor:

```h
#pragma once

#include "IPlug_include_in_plug_hdr.h"

const int kNumPresets = 1;

enum EParams
{
  kMakeup = 0,
  kThreshold,
  kRatio,
  kAttack,
  kRelease,
  kMix,
  kHold,
  kBypass,
  kNumParams
};

using namespace iplug;
using namespace igraphics;

class CompressorPlugin final : public Plugin
{
public:
  CompressorPlugin(const InstanceInfo& info);

#if IPLUG_DSP // http://bit.ly/2S64BDd
  void ProcessBlock(sample** inputs, sample** outputs, int nFrames) override;
#endif
};
```

```cpp
#include "CompressorPlugin.h"
#include "IPlug_include_in_plug_src.h"
#include "IControls.h"
#include "ProcessCompression.h"

ProcessCompression C;

CompressorPlugin::CompressorPlugin(const InstanceInfo& info)
: Plugin(info, MakeConfig(kNumParams, kNumPresets))
{
  // Set Params
  GetParam(kMakeup)->InitGain("Makeup Gain", 0., 0., 12.0, 0.01);
  GetParam(kThreshold)->InitGain("Threshold", 0., -30., 0., 0.01);
  GetParam(kRatio)->InitDouble("Ratio", 1., -30., 1., 0.01, "dBV");

  GetParam(kAttack)->InitDouble("Attack", 20.0, 1., 500.0, 0.01, "Ms");
  GetParam(kRelease)->InitDouble("Release", 50.0, 1., 2000.0, 0.01, "Ms");
  GetParam(kMix)->InitDouble("Mix", 100.0, 0., 100.0, 0.01, "%");
  GetParam(kBypass)->InitBool("Bypass", 1);


#if IPLUG_EDITOR // http://bit.ly/2S64BDd
  mMakeGraphicsFunc = [&]() {
    return MakeGraphics(*this, PLUG_WIDTH, PLUG_HEIGHT, PLUG_FPS, GetScaleForScreen(PLUG_WIDTH, PLUG_HEIGHT));
  };

  mLayoutFunc = [&](IGraphics* pGraphics) {
    pGraphics->AttachBackground(GUI_FN);
    pGraphics->LoadFont("Roboto-Regular", ROBOTO_FN);

    // Load Images
    const IBitmap threshold = pGraphics->LoadBitmap(THRESHOLD_FN, 128);
    const IBitmap ratio = pGraphics->LoadBitmap(RATIO_FN, 128);
    const IBitmap attack = pGraphics->LoadBitmap(ATTACK_FN, 128);
    const IBitmap release = pGraphics->LoadBitmap(RELEASE_FN, 128);
    const IBitmap makeup = pGraphics->LoadBitmap(MAKEUP_FN, 128);
    const IBitmap mix = pGraphics->LoadBitmap(MIX_FN, 128);
    const IBitmap bypass = pGraphics->LoadBitmap(BYPASS_FN, 2);
```

```
    // Attach controls
    pGraphics->AttachControl(new IBKnobControl(55, 33, threshold, kThreshold));
    pGraphics->AttachControl(new IBKnobControl(204, 33, ratio, kRatio));
    pGraphics->AttachControl(new IBKnobControl(55, 136, attack, kAttack));
    pGraphics->AttachControl(new IBKnobControl(204, 136, release, kRelease));
    pGraphics->AttachControl(new IBKnobControl(55, 239, makeup, kMakeup));
    pGraphics->AttachControl(new IBKnobControl(204, 239, mix, kMix));
    pGraphics->AttachControl(new IBSwitchControl(211, 392, bypass, kBypass));

  };
#endif
}

#if IPLUG_DSP
void CompressorPlugin::ProcessBlock(sample** inputs, sample** outputs, int nFrames)
{
  // Update params
  const int nChans = NOutChansConnected();

  double* in1 = inputs[0];
  double* in2 = inputs[1];

  double* out1 = outputs[0];
  double* out2 = outputs[1];
  double Threshold = DBToAmp(GetParam(kThreshold)->Value());
  double Ratio = DBToAmp(GetParam(kRatio)->Value());
  double Attack = GetParam(kAttack)->Value();
  double Release = GetParam(kRelease)->Value();
  double Makeup = DBToAmp(GetParam(kMakeup)->Value());
  double Samplerate = GetSampleRate();

  double mix = GetParam(kMix)->Value() / 100;
  bool bypass = GetParam(kBypass)->Value();
```

```
  for (int s = 0; s < nFrames; ++s, ++in1, ++in2, ++out1, ++out2)
  {
    if (bypass)
    {
      // Mix
      *out1 = ((mix * (C.Compression(*in1, Threshold, Ratio, Attack, Release, Makeup, Samplerate))) + (1 - mix) * *in1);
      *out2 = ((mix * (C.Compression(*in2, Threshold, Ratio, Attack, Release, Makeup, Samplerate))) + (1 - mix) * *in2);
    }
    else
    {
      *out1 = *in1;
      *out2 = *in2;
    }

  }
}
#endif
```

Here is all the code for the ProcessCompression class:

```
#include "ProcessDistortion.h"
#include "IPlugUtilities.h"

#pragma once
class ProcessCompression
{
public:

  double Compression(double input, double threshold, double ratio, double attack, double release, double makeup, int samplerate);

private:
  int Hold = 1000;
  int Timeout;
  int NAttack;
  int NRelease;
  double Gain = 1.0;
  double Gain_Step_Attack;
  double Gain_Step_Release;
  int CompStates = 0;
};
```

```
orPlugin-vst2                                                    (Global Scope)
#include "ProcessCompression.h"
#include "cmath"

ProcessDistortion D;

double ProcessCompression::Compression(double input, double threshold, double ratio, double attack, double release, double makeup, int samplerate)
{
  // Allow change of attack and release
  NAttack = samplerate * (attack / 1000);
  NRelease = samplerate * (release / 1000);

  Gain_Step_Attack = (1.0 - ratio) / NAttack;
  Gain_Step_Release = (1.0 - ratio) / NRelease;

  // Stage checker
  if (fabs(input) > threshold)
  {
    if (Gain >= ratio)
    {
      if (CompStates == 0 || CompStates == 3)
      {
        CompStates = 1;
        Timeout = NAttack;
      }
    }
    if (CompStates == 2)
    {
      Timeout = Hold;
    }
  }
  if (fabs(input) < threshold && Gain <= 1.0)
  {
    if (Timeout == 0 && CompStates == 2)
    {
      CompStates = 3;
      Timeout = NRelease;
    }
  }
```

```
// Compression calculation
switch (CompStates)
{
case 0:
  // No Gain Reduction
  if (Gain < 1.0)
  {
    Gain = 1.0;
  }
  break;

case 1:
  // Attack
  if (Timeout > 0 && Gain > ratio)
  {
    Gain -= Gain_Step_Attack;
    Timeout--;
  }
  else
  {
    CompStates = 2;
    Timeout = Hold;
  }
  break;

case 2:
  // Gain Reduction
  if (Timeout > 0)
  {
    Timeout--;
  }
  else
  {
    CompStates = 3;
    Timeout = NRelease;
  }
  break;
```

```
case 3:
  // Release
  if (Timeout > 0 && Gain < 1.0)
  {
    Timeout--;
    Gain += Gain_Step_Release;
  }
  else
  {
    CompStates = 0;
  }
  break;
}

// Apply Compression and Makeup Gain
input = input * Gain * makeup;
// Apply Saturation
input = D.Distortion(input, 1);

return input;
}
```

As stated before, all the code is available on my Github profile and may have been updated since the creation of this document.

References

1. Demofox, “Decibels (dB) and Amplitude”
2. Hyperbits, “Music Production in 2 Minutes: Understanding Compression”

Testing

Since I have 4 separate plug-ins, I have created a set of test criteria for each. This allows me to go into detail about each feature in the plug-ins and compare against similar industry standard plug-ins. All these tests are aimed both addressing different test criteria and the functionality of the plug-ins themselves.

Delay

When testing the plugin, I shall compare it to two industry standard delay plugins. Ableton's stock delay [1] and Wave's H-Delay [2]. This is because Ableton's delay is a flexible digital delay and allows me to compare the Stereo Delay feature of my plugin against it, and Wave's H-Delay because it has a good analogue tape emulation feature. This also addresses [SC3_B]. To standardise my tests, I shall use a clap sound that I recorded myself to compare both the fidelity and accuracy of my delay.

I will perform 4 tests to judge the success of my design:

- Delay Accuracy
- Sonic Fidelity
- Accuracy of Tape Emulation
- Extreme cases

To test Delay Accuracy, I will manually place repetitions of my clap at each specified interval, then I will run the clap through the delay at the same time and record it to a new track, this allows me to visually see if the delay is accurate. I will do this for each delay mode to ensure that it is accurate. This test addresses [SC3_A].

To test Sonic Fidelity, I will run the clap through my delay and Ableton's delay with identical delay times and then I will phase invert my delay. If there is no loss in quality, then I should hear silence when the audio is played back. If I don't hear silence, then the only difference between the channels could be the volume of each delay because the feedback amounts won't be identical. This would make the delays quieter compared to loss in frequencies. This test addresses [SC5_A] and [SC5_B].

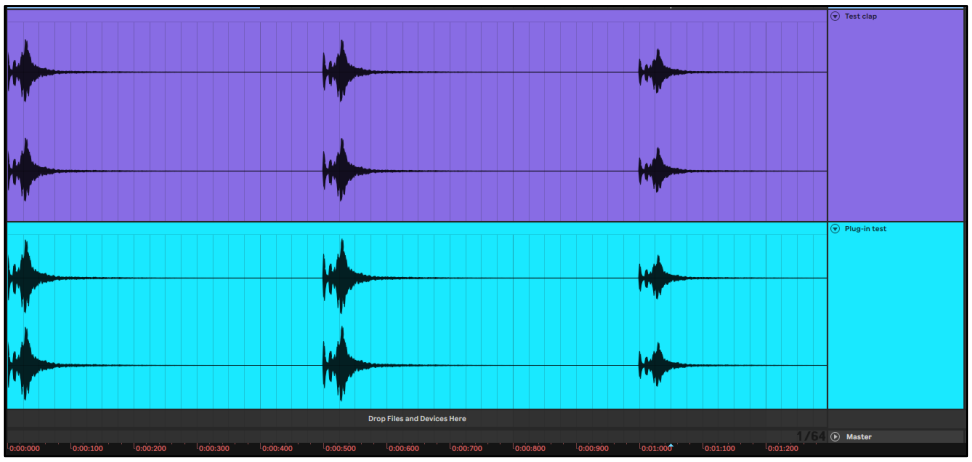
To test the accuracy of my Tape Emulation, I will compare the sound of my delay to Wave's H-Delay with its analogue setting turned on. Seeing as this setting is less about absolute accuracy and focuses on the sound it produces, there will be bias from me whether it sounds good. However, I plan to get feedback from alpha testers on this element of the plugin and I will adjust my design accordingly. This test addresses [SC3_A].

To test Extreme cases, I will see how the quality of the sound changes dependant of the input signal. I shall use audio with extreme volume, phase issues, dc offset issues and sample rate issues. Extreme volume could affect the quality of the signal when it fed into the buffer. Phase issues could affect the delays and cause phase cancelation when the delay is in stereo mode. Dc offset issues because it could cause distortion in the delay. Sample rate issues could affect the delay time since the buffer relies on the sample rate. This test addresses all of [SC5].

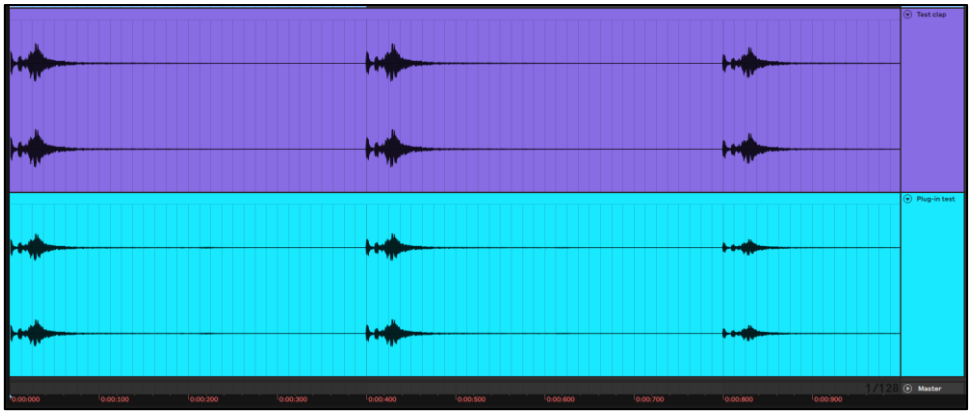
Delay Testing

Delay Accuracy:

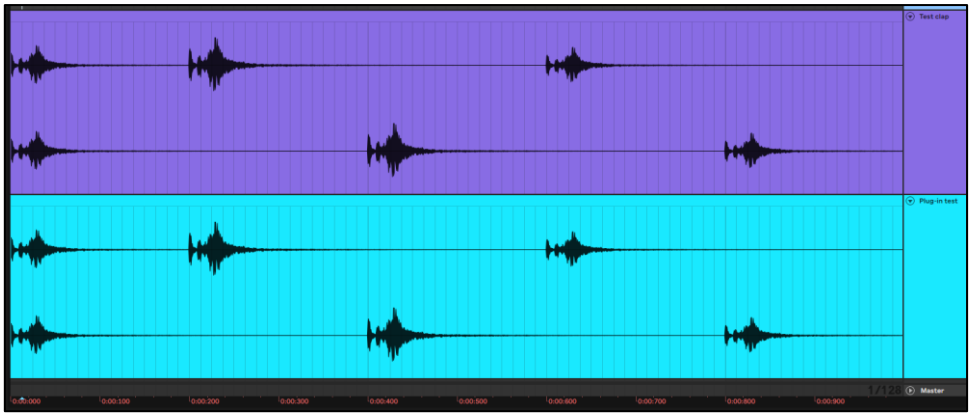
In the first test, I set the delay time to 500ms. The purple channel is the test clap which I manually placed, and the blue channel is the output from my delay. There is a ruler at the bottom of the image if you wish to check the timings yourself. However, they look identical and when I played them together, I didn't hear any phasing meaning that they are perfectly aligned.



In the second test, I set the delay time to 1 beat at 150 bpm. The channels are coloured the same as last time. Again, both claps are perfectly lined up meaning it's another successful test.

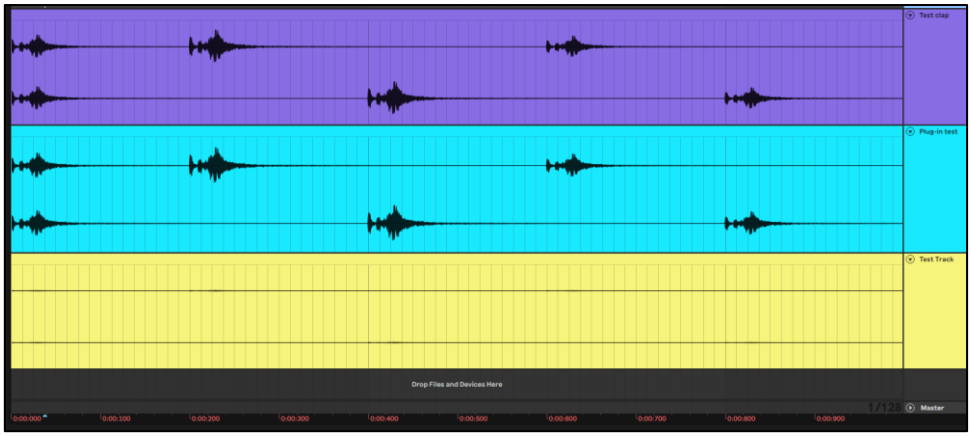


For the final delay accuracy test, I tested the accuracy of the ping-pong mode. I set the delay time to ½ a beat at 150 bpm. As expected, the signals are identical, meaning that my plug-in has passed this test.



Sonic fidelity:

Moving on, using the audio I recorded from the last test, I phase inverted Ableton's delay and recorded the two channels together. The purple channel is Ableton's delay, the blue channel is my delay, and the yellow channel is both sounds recorded together. As you can see it is almost completely silent meaning that the sound is identical however they are at different volumes. Therefore, there was no loss in sonic fidelity.



Accuracy of Tape Emulation:

Since I can't visually show this test easily, I got feedback from alpha testers to get their opinions on the feature. Here is what they said:

- Tester 1: I think the tape emulation can be extreme at times when the warmth is cranked all the way up however its nice seeing more controls in a delay plug-in.
- Tester 2: The tape emulation is fine, when comparing it to other plug-ins I own it doesn't compare however it is a great addition and if it was improved in the future that would be great.
- Tester 3: I loved the feature since it adds a nice amount of distortion and filtering, I don't have much else to say.
- Tester 4: I like the tone control the most out of the two since it simplifies my processing chain, but I do think the warmth can be a bit too much.

From what they've said my implementation can be a little extreme when turned up to the max however they still liked the addition of these controls since it can add some character to the sound. So, I will say it just about passed since only two testers said that it was a good emulation however since the overall opinion was positive, I think it justifies a pass.

Extreme cases:

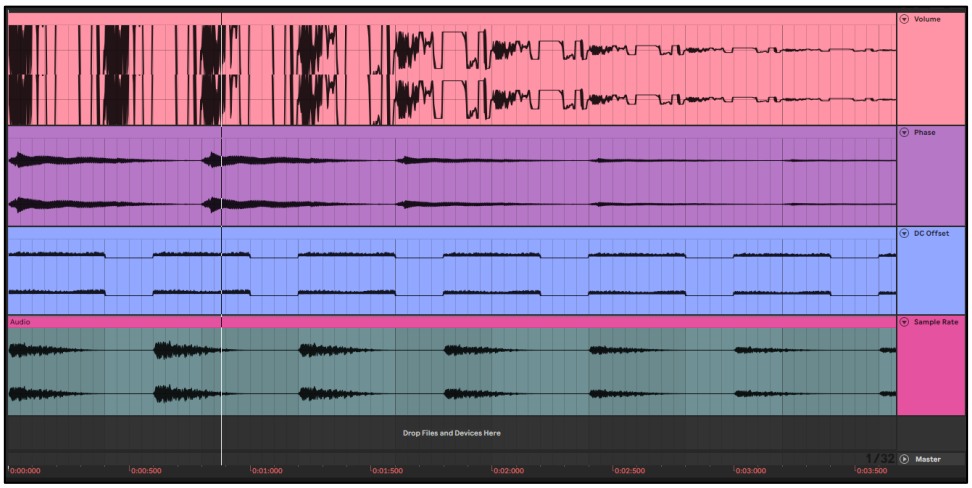
From top to bottom each audio clip addresses, Extreme Volume, Phase Issues, DC Offset Issues and Sample Rate Issues.

In the first audio clip the sound is clipping by 30db however when I reduced the volume of the clip there was no loss in audio quality apart from a bit of saturation which is a by-product of the soft clipper.

In the second audio clip the signal was put out of phase by phase inverting the left channel. As you can see the delay is still accurate and there was no phase cancellation.

In the third audio clip the signal has some DC Offset issues which can be seen through the audio hovering above the centre line. Apart from the distortion that DC Offset adds to the signal the delay worked perfectly fine.

In the final audio clip, the audio was recorded at 96kHz, and the delay was used at the same sample rate. Again, there was no issue with this, so this test is a success.



Distortion

When testing my plugin I will compare it against two industry standard plugins, Ableton's Saturator [3] and Kilohearts' Distortion [4]. Ableton's Saturator has both Soft and Hard clipping modes alongside a DC correction filter, Kilohearts' Distortion also has the same features as Ableton's Saturator alongside tone, dynamics and foldover distortion modes. This also addresses [SC3_B]. To standardise each test, I will use an identical sinewave as my input signal when performing each test.

I will perform 4 tests to judge the success of my design.

- Accuracy of distortion modes
- Sonic Fidelity
- Accuracy of Tone, Dynamics, and Bias controls
- Extreme cases

To test the accuracy of each distortion mode, I will compare each distortion mode with their counterparts in Ableton's 'Saturator' and Kilohearts' 'Distortion'. Each plugin will have no effects that will affect the signal enabled. This includes Tone, Dynamics, and Bias. I will record the output of each plugin to a new channel so I can visually compare the waveforms of each plugin. This test addresses [SC3_A].

To test Sonic fidelity, I will firstly compare the signal when there is no distortion being added. The plugin should not add anything to the signal when nothing is enabled. I will also test the quality of the signal before and after the filter since I had some phasing issues during my initial implementation. This test addresses [SC5_A] and [SC5_B].

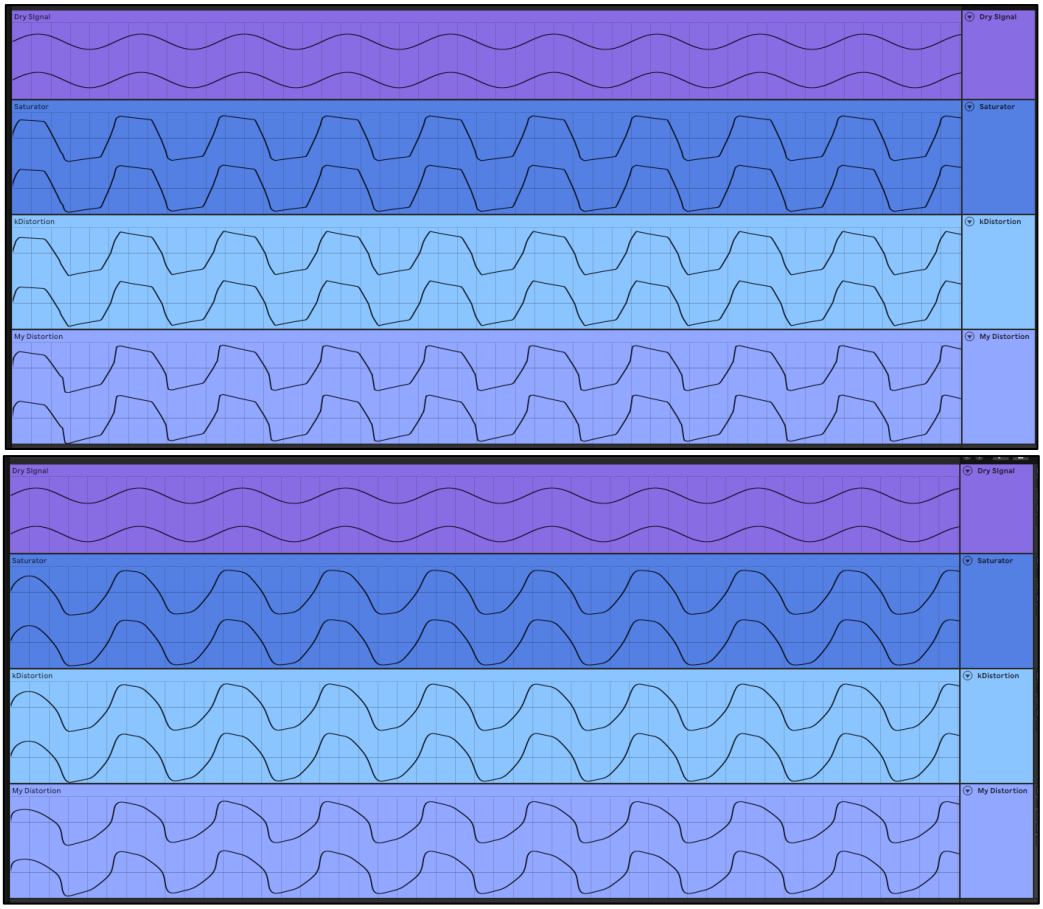
To test the accuracy of the Tone, Dynamics, and Bias controls, I will again compare the output of my plugin with Kilohearts' 'Distortion'. However, I will also manually recreate the effect of all my controls with Ableton's stock plugins and compare the results with my own plugins to check that they are working properly. This test addresses [SC3_A].

To test Extreme cases, I will see how the quality of the sound changes dependant of the input signal. I shall use audio with extreme volume, and sample rate issues. Extreme volume could affect the quality of the signal when it fed into the distortion algorithms causing them to behave in unintended ways. Sample rate issues since the filter depends on the sample rate. This test addresses all of [SC5].

Distortion Testing

Accuracy of distortion modes:

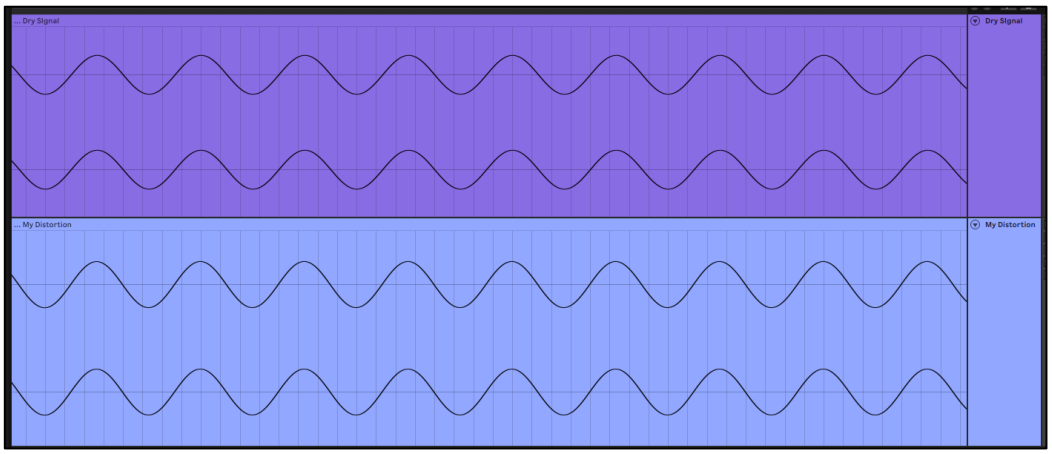
Since it would take up a lot of space showing each test, I'll only show a few of them so you can see how I tested everything. To test this, I recorded a sine wave at 120hz and placed it on four separate channels. From top to bottom; Unprocessed signal, Ableton's Saturator, Kiloheart's Distortion, and my Distortion plugin. The first screenshot shows a comparison for the hard-clipping mode and the second shows the comparison for soft-clipping mode.



As you can see all three plug-ins produce a similar looking and sounding waveform. I would call this test a complete success since when I tested out the other waveforms, I had similar results apart from my HRM mode since it is a custom mode which no other plug-in has.

Sonic Fidelity:

As with the previous test I have ran a sine wave at 120hz through my distortion plug-in when it is active but shouldn't be affecting the signal. If the waveform looks identical, then it isn't adding any distortion and if the waveforms are perfectly aligned then there is no phasing. From my test my plug-in retained sonic fidelity and has passed.



Accuracy of Tone, Dynamics, and Bias controls:

When assessing this test, I will be comparing the sound of each plug-in. If I compare the waveforms, they could like quite different owing to the different implementations of the controls. However, they should sound the same or similar otherwise it wouldn't be an accurate emulation of the effect.

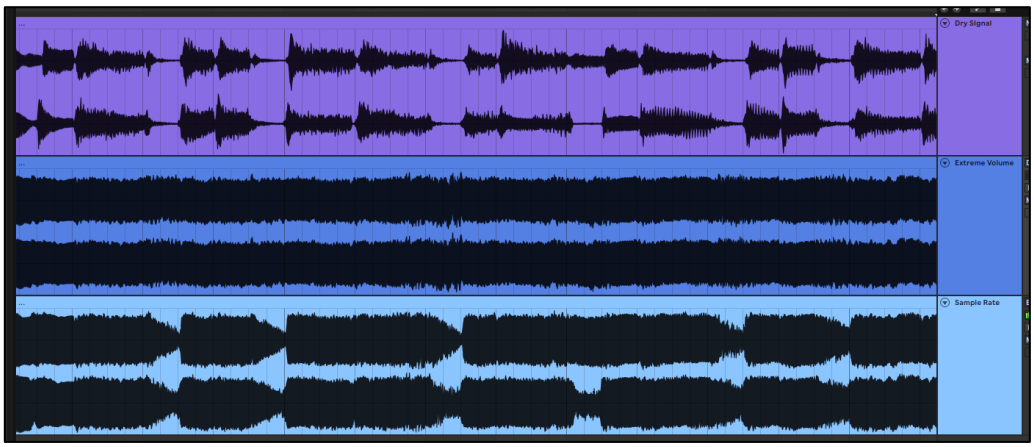
- Tone: Since Ableton's Saturator and Kilohearts' Distortion don't have a dedicated tone control, I will manually recreate the effect with a high-shelf eq. I set both my plug-in and Kiloheart's Distortion to the same settings as the first test, but I introduced the tone control. When I increased the amount, it had a similar effect on both signals meaning I have successfully recreated the control.
- Dynamics: When comparing Kilohearts' Distortion to my Distortion, both had a similar effect on the signal however they did sound distinctly different. Both did achieve the same effect of making the processed signal follow the dynamics of the original sound so I would call this implementation a success.
- Bias: Finally, I compared my Distortion plug-in against Kilohearts' Distortion since it has a bias control. Again, both sounds did sound slightly different when increasing the amount of bias however after playing around with it I found that my plug-in adds more bias than Kiloheart's since I turn my control less to achieve the same sound. Meaning that I have successfully implemented the control.

Extreme cases:

To Test extreme cases I will follow the same method as my Delay plug-in. The first clip is the dry signal, the second is the signal with extreme volume and the last is the clip with sample rate issues.

When testing all my distortion modes with extreme volume, they all functioned as intended. Although they did work, I wasn't sure whether my soft and hard clipping modes would still function. Although it doesn't look like it the original signal was boosted by 36dB and upon further inspection of the waveform the algorithm was working as intended.

When testing sample rate issues, the clip has been recorded at a different sample rate and the audio is being fed directly into the distortion. The filters still remove any DC Offset and attenuate the high-end properly. Similarly, as with the other test, it may not look like it, but the filters are working properly although I have introduced a sample rate issue in the original signal.



Flanger

When testing the plugin, I shall compare it to two industry standard Flanger plugins. Ableton's stock Flanger [5] and the Kilohearts' Flanger [6]. Despite both plugins having more functionality I will only be comparing the Flanger and any similar features. Doing this addresses [SC3_B].

I will perform 3 tests to judge the success of my design:

- Flanger Accuracy
- LFO Accuracy
- Extreme cases

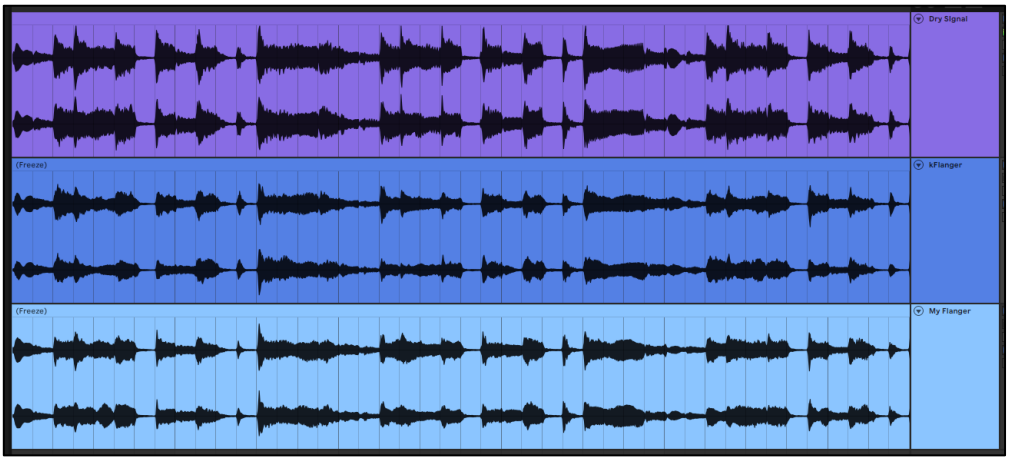
To test Flanger Accuracy, I will use each plugin with identical settings and compare the sounds of each Flanger. First and foremost, the 'sound' is the most important aspect of an effect plugin, so a comparison between them is logical. It will also highlight any glaring issues in the sound of the effect. It could sound good but if it doesn't sound like a Flanger then it isn't one. This test addresses [SC3_A].

To test LFO Accuracy, I will output the LFO's waveform through the plugin and record it at different frequencies and then compare it to Ableton's Operator [7]. I will use both visual comparison and phase inversion to check whether they are the same frequency. This test addresses [SC3_A].

To test Extreme cases, I will see how the quality of the sound changes depending on the input signal. I shall use audio with extreme volume, dc offset issues and sample rate issues. Extreme volume could affect the quality of the signal when it fed into the buffer. Dc offset issues because it could cause distortion in the Flanger. I won't test phase issues because Flanging creates comb filtering which messes with the phase of the original signal. Sample rate issues since the filter and buffer depend on the sample rate. This test addresses all of [SC5].

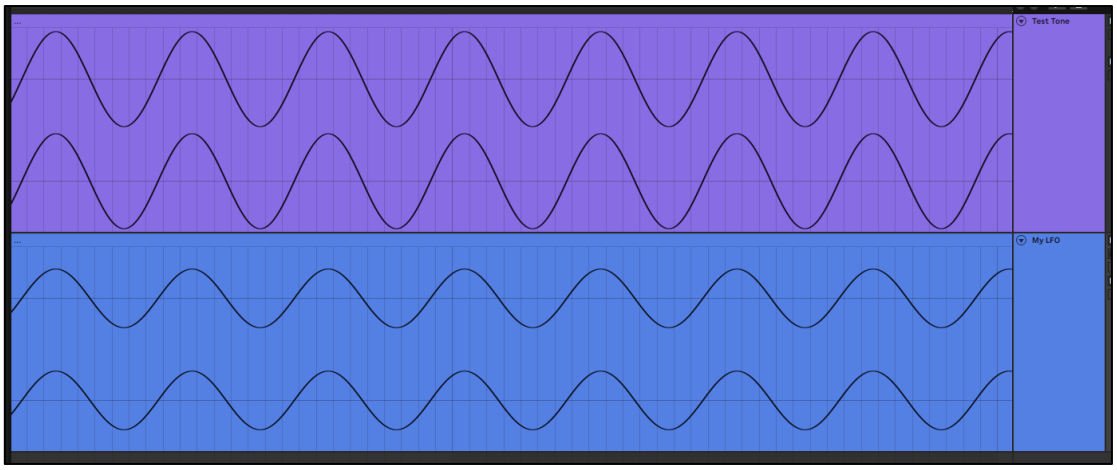
Flanger accuracy:

As per usual, I will use a visual comparison to compare the effectiveness of my flanger. The first channel is the dry signal, the second is being processed using Kiloheart's Flanger and the third channel is my flanger using identical settings to emulate the exact same effect. Although the two channels look different, the sound almost identical. This could be due to the two plug-ins LFO's not being aligned or their better implementation.



LFO Accuracy:

As mentioned earlier I will be comparing a sinewave at 10hz to my LFO to test whether my timing is correct. The first channel is the test signal and the second is the output from my LFO. As you can see, they line up perfectly meaning that the LFO timing is accurate. Although there is a slight difference in volume the frequency is identical.



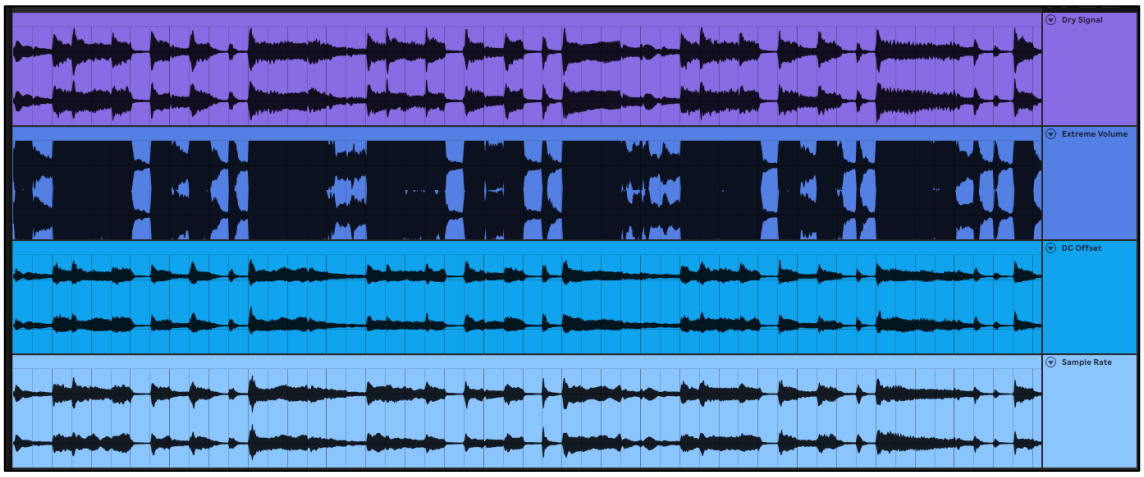
Extreme Cases:

I will be using the same method to test extreme cases as with my other plug-ins. From top to bottom the tracks are; Dry Signal, Extreme Volume, DC Offset issues and Sampe Rate issues.

When I fed extreme volume into the flanger, recorded it and then reduce the volume by the same amount there was no degradation in audio quality, meaning I've passed this test.

When adding DC Offset to the signal and passing it through my plug-in, there was no artificial distortion added and the output just sounded like the dry signal with a comb filter on it. Hence showing that it works.

When I fed a signal with sample rate issues into my flanger the buffer worked as intended which can be seen through the waveform. There were no unwanted artifacts in the signal meaning that it has passed this test.



Compressor

When testing the plugin, I shall compare it to two industry standard Compressor plugins. Ableton's stock Compressor [8] and the Kilohearts' Compressor [9]. I chose to use these two since they have a similar set of features to my compressor. I would have compared my implementation to Wave's API 2500 [10] since I own this plug-in. However, since it is emulating physical hardware the differences between compression algorithms would be too great to perform a fair test. Doing this addresses [SC3_B].

I will perform 5 tests to judge the success of my design:

- Threshold accuracy
- Attack and Release accuracy
- Ratio accuracy
- Makeup gain accuracy
- Extreme cases

To test Threshold accuracy, I will generate constant tones at specified volumes; -30dB, -20dB, -10dB, -5dB and 0dB. Then I will test whether the compressor is affecting the signal at the specified volumes. For example, If I set the threshold to –20dB, then there should be no compression for –30dB and –20dB but there should be compression for every other tone. To test fairly, makeup gain and saturation will be turned off since they could affect the volume of the output signal. This test addresses [SC3_A].

To test Attack and Release accuracy, I will send a tone through the compressor which triggers compression with a high ratio. I will record the whole process so I can visually analyse the result and check whether the attack and release are accurate. To test the release, I will suddenly drop the volume of the tone bellow the threshold since if I remove the signal then I won't be able to tell how long it takes for the signal to return to an uncompressed state. This test addresses [SC3_A].

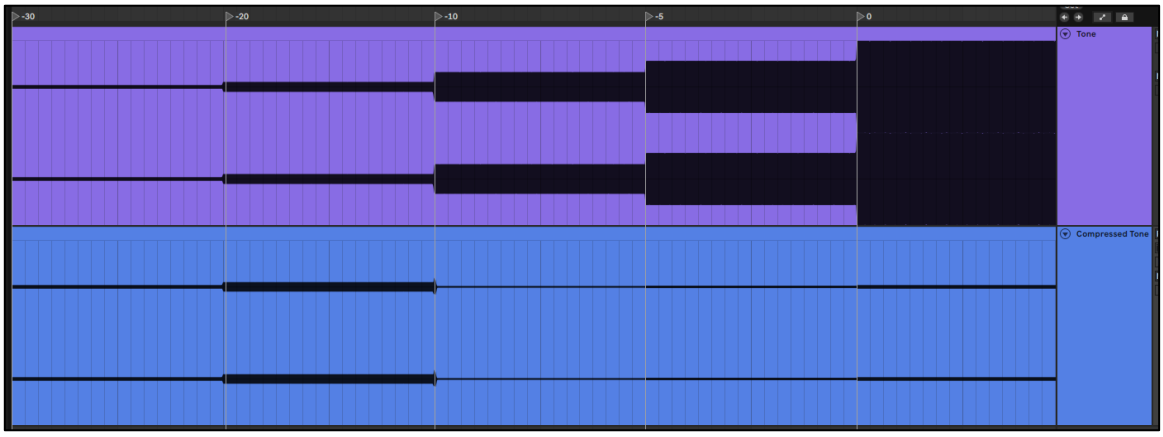
To test Ratio accuracy, I will test at each increment. I will start with a tone where its volume is at the threshold. Then I will increase the volume of this tone by the same amount as the ratio. So, if the ratio is 5, then I will increase the volume in increments of 5dB. This test addresses [SC3_A].

To test Makeup gain accuracy, I will start with no makeup gain and slowly increase it. Checking whether the volume increases in accordance with the makeup gain control. This test addresses [SC3_A].

Finally, to test Extreme cases I will see how the quality of the sound changes depending on the input signal. I shall use audio with extreme volume and dc offset issues. Extreme volume could affect how the ratio behaves when the volume is ludicrously high. DC offset issues could affect how the functionality of the threshold. This test addresses [SC5].

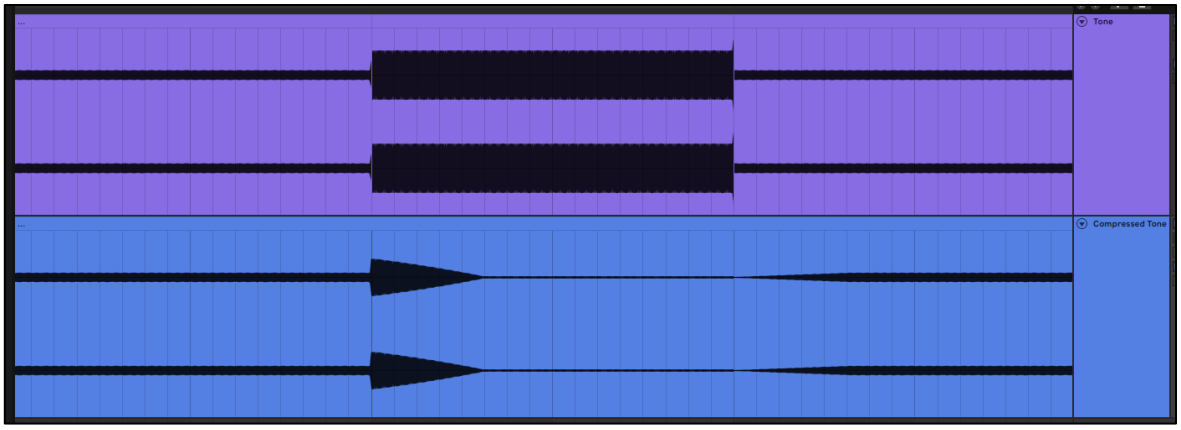
Threshold Accuracy:

For this test I set the threshold to –19.9dBV and the ratio to 30dBV. I set the ratio to this amount since it shouldn't compress –20dBV however if the control isn't accurate, it could compress the signal. I set the ratio to –30dBV since when the signal should be compressed there should be a noticeable difference in volume when compared to the uncompressed signal. After testing this at different volumes the threshold was accurate.



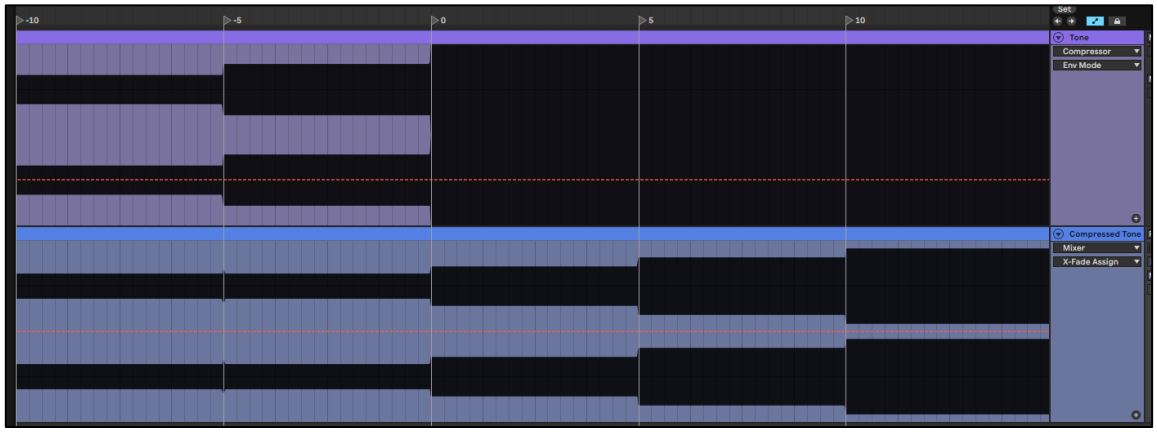
Attack and Release Accuracy:

When testing the Attack and Release times, I set both the attack and release to 500ms and suddenly changed from a quiet to a loud and back to a quiet signal. This will allow me to visually see how long the attack and release are taking so I can measure if they are the correct length. Once I tested this, I noticed that the attack and decay times were half as long as they should be. Both were only 200ms long, this is probably due to a dodgy calculation somewhere which I can easily fix by multiplying the attack and release values by 2. Although they weren't accurate when I tested them, I have amended the code to fix this issue since it wasn't hard to fix.



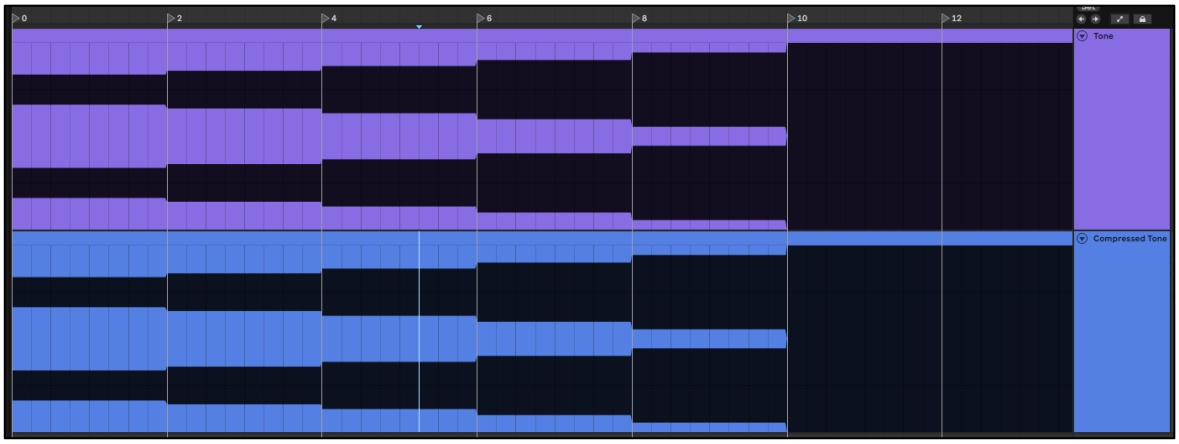
Ratio Accuracy:

Following the method I described earlier, I set the threshold to –10dBV and the ratio to 5dBV. When I increase the volume of the compressed signal in increments of 5dBV it should only increase by 1dBV. When testing this it didn't work exactly as I intended however, I thought I should compare it to Ableton's delay seeing if it was having the same issue. Surprisingly output from each plug-in was almost identical so although I didn't strictly pass the test, I am achieving the same effect as Ableton's compressor.



Makeup gain Accuracy:

Since this is as simple as turning up the makeup gain and recording the volume, the first clip shows what the output volume should look like, and the second clip shows what my compressor outputs. I will increase the makeup gain in increments of 2dBV. As you can see the result is identical meaning that I've successfully implemented this control.

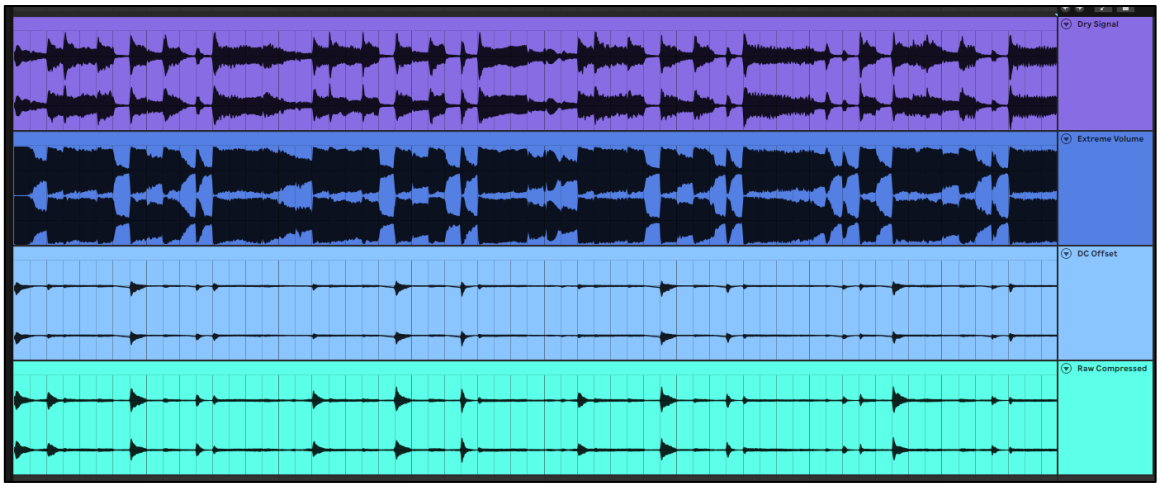


Extreme cases:

As usual, the clips are ordered from top to bottom; Dry Signal, Extreme volume, DC Offset issues and the signal just being compressed without any issues. Although I could check for sample rate issues, I have already performed some successful initial testing so it would be redundant to do it again. For these tests I set the threshold to –15dBV and the ratio to 20dBV.

When I sent the signal with extreme volume into the compressor, it was successfully compressing the signal and we can clearly see the reduction in dynamic range of the signal. From a few tests I didn't encounter any issues with extreme volume.

When adding DC Offset to the input signal there was no difference to the amount of compression and when the threshold was reached when compared to a raw signal. Although it didn't introduce any issues here, if the DC Offset was loud enough to trigger the compression, then it would always be compressed but this would be the same with any compressor without a DC Offset filter at the start of the processing chain. However, I feel that it has passed this test.



Usability testing and Alpha tester feedback

I also asked the alpha testers about the usability of the plug-ins. They were all positive about the GUI and layout of the plug-ins. From their feedback I think I've achieved my goal of making the GUI more accessible by basing it off a guitar pedal. However, when I finish this bundle I will take a different direction with the design, since it would make the plug-ins more user friendly if they could see the values that they're controlling on the GUI.

- Tester 1: I love the GUI, it's such an elegant and familiar design. I had no issues using it.
- Tester 2: Coming from a guitar-based background I love the design; you've got almost everything nailed apart from the input and output jacks lol. But great job.
- Tester 3: The GUI looks super professional and its very fun to use, I do wish some of the knobs didn't get so bright when turning them since it gets a bit hard to see the dial.
- Tester 4: I like how everything looks, if there was some kind of indication of the precise values the knob had on the GUI would be my only improvement but apart from that it's great!

I also asked the testers about which plug-in they liked and disliked the most and why. This allows me to create a comprehensive list of features users like and dislike letting me create better software in the future.

- Tester 1: I liked the Flanger the most since I love how it sound and I prefer its colour scheme. If I had to pick one, I didn't like the Distortion as much since for my music I don't need as extreme distortion modes.
- Tester 2: I preferred the Compressor over all the other plug-ins since compared to other plug-ins I like the sound it makes. I think the Delay needs more improvements since there is clicking when the time knob is changed, and I would prefer it if it acted more like an analogue tape delay where the delays speed up and slow down when you change the time.
- Tester 3: Since I make heavy bass music, I loved the different foldover algorithms for the Distortion plug-in since it creates such a unique sound. However, I think the delay could have some improvements since it sounds quite clunky.
- Tester 4: Out of all the plug-ins I preferred the Flanger since it creates interesting sounds by turning a few controls. Since I don't understand compression, I would have to say the Compressor was my least favourite plug-in because I haven't gotten much use out of it yet.

References

1. Ableton, "Live Audio Effect Reference" 24.11
2. Waves, "H-Delay Analog Delay Plugin"
3. Ableton, "Live Audio Effect Reference" 24.37
4. Kilohearts, "Distortion"
5. Ableton, "Live Audio Effect Reference" 24.11
6. Kilohearts, "Flanger"
7. Ableton, "Live Instrument Reference" 26.6
8. Ableton, "Live Audio Effect Reference" 24.9
9. Kilohearts, "Compressor"
10. Waves, "API 2500"

Evaluation

Addressing success criteria

When evaluating my success criteria, each criterion will have its own subsection where I'll discuss if and how I've met the requirements. Here is a copy of the success criteria so you don't have to go back to the start.

Success ID	Success Criteria	Appropriate fulfilment of criteria
SC0	Optimised performance: The plug-in must be easy to run during intended use. This allows the user to use multiple instances of the plug-in without a drastic increase in load on the CPU.	• The impact on the CPU should not exceed 2x the load when compared to Ableton's stock effects. [SC0_A] • Each plug-in should not take over 1 second to open. [SC0_B] • Each plug-in should be responsive to user input. Each control should not take more than 100ms to respond to an input. [SC0_C]
SC1	Ease of use: Since the software is aimed at beginners, ease of use is essential. This encompasses both the plug-in itself and its documentation.	• Every essential control must not be hidden inside of menus. [SC1_A] • Each control must be labelled clearly. [SC1_B] • Website must have user focused documentation for operation of each plug-in. [SC1_C] • Every plug-in should be accessible for the colourblind. There should be sufficient contrast between controls allowing the colourblind to use the plug-in. [SC1_D]
SC2	Compatibility between software and operating systems: Since each stakeholder's use of the solution spans across different software and operating systems it must work across all of them.	• Every plug-in must work on different operating systems. [SC2_A] • Every plug-in must work on different DAWs. [SC2_B] • Every plug-in must work in both audio and video editing software. [SC2_C]
SC3	Accurate emulation of desired effect: It isn't a successful solution unless each plug-in achieves its goal.	• Each control must have the desired effect on the signal. [SC3_A] • Each plug-in should be comparable to similar solutions. [SC3_B]
SC4	Reduced file size for each plug-in: Plug-ins should not take up an excessive amount of space.	• Each plug-in should not exceed 20mb in size. [SC4_A]

		• The entire package should not exceed 100mb in size. [SC4_B]
SC5	Retention of audio quality: Since my plug-ins could be used by professionals, there should be no deterioration in quality throughout the processing of the audio.	• There should be no unintentional compression while processing the audio. [SC5_A] • There shouldn't be any artifacts or ghost signals after processing. [SC5_B] • Each plug-in must adapt to changes in sample rate. Where applicable there should be no errors when the sample rate is changed. [SC5_C]
SC6	Ability to save the current state of the plug-in: For faster use of my solution, the user should be able to save and load previous states of each plug-in.	• The current state of the plug-in can be saved to an external file. [SC6_A] • External files can be opened, loading a previous state of the plug-in. [SC6_B]
SC7	Ability to reuse code: For both faster development of my solution and reuse of my code by other programmers, essential modules should be implemented as classes.	• Where appropriate, different modules should be implemented as classes allowing for re-use across projects. [SC7_A] • The website must have a dedicated section explaining how each modules work and how to interface with the class. [SC7_B]
SC8	Stability of code: During normal use, each plug-in must run in a stable state and not crash which could cause loss of data.	• Each plug-in must run in a stable state. They must run for over 5 minuets with a constant signal without critical errors. [SC8_A] • Each plug-in must not use an excessive amount of system resources. They should not use over 100mb memory during use. [SC8_B]

<u>SC0 - Optimised performance:</u>

Starting with [SC0_A], for each plug-in I loaded up 20 instances of them on a blank channel and looked at the CPU usage. The peak usage was 15%, when compared to their Ableton counterparts the Ableton plugins achieves a peak usage of 13%. Since this is less than 2x Ableton's performance I have met this criterion.

Moving on to [SC0_B] and [SC0_C], I loaded up some screen capture software and recorded the time it took to open the plug-in and for any control to influence the sound. I used screen capture software since it would be more accurate than me manually timing it. Each plugin took under 0.4 seconds to open and there was an effect on the sound in under 30 milliseconds on every plug-in.

With all of this considered I have successfully addressed [SC0].

SC1 - Ease of use:

Linking back to my GUI section, since I designed my plug-ins off guitar pedals there are no menus with extra controls and each control is labelled clearly addressing [SC1_A] and [SC1_B].

Owing to it being one of the last things on my to-do list, I haven't gotten round to creating any documentation on how each plug-in works outside of this document. Owing to this I haven't met [SC1_C]. However, in the future I will create this and link it on my website.

To test [SC1_D], I simulated what it would look like for colour blind people to view my plug-ins. After comparing with 8 different types of colour blindness, each control was clearly defined. Here is one example of the images I compared against.

With all of this considered I have mostly addressed [SC1].

SC2 - Compatibility between software and operating systems:

When testing [SC2_A], I interviewed one of my alpha testers since he uses macOS. He said that the plug-ins worked as expected and he didn't encounter any issues. However, since he wasn't running an ARM based system I don't know if my plug-ins will run on one. I also tested my plug-ins on Linux where the also performed as expected.

Since I own multiple DAWs on my laptop, I tested my plug-ins in Cubase [1] and FL Studio [2] to address [SC2_B]. Again, I didn't encounter any issues and they plug-ins performed identically to Ableton.

To assess [SC2_C], I loaded up my plug-ins in DaVinci Resolve [3]. I recorded some audio into the timeline and tested my plug-ins on it. Again, there were no issues, so I have met this criterion.

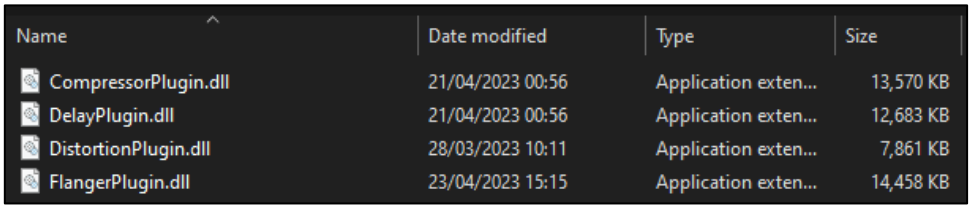
With all of this considered I have successfully addressed [SC2].

SC3 - Accurate emulation of desired effect:

Linking back to my testing section, I outlined my tests for [SC3] and from my results I have successfully addressed this criterion.

SC4 - Reduced file size for each plug-in:

Currently, each plug-in is under 20mb in size and the whole bundle comes to 46.5mb. This successfully addresses all of [SC4].



Name	Date modified	Type	Size
CompressorPlugin.dll	21/04/2023 00:56	Application exten...	13,570 KB
DelayPlugin.dll	21/04/2023 00:56	Application exten...	12,683 KB
DistortionPlugin.dll	28/03/2023 10:11	Application exten...	7,861 KB
FlangerPlugin.dll	23/04/2023 15:15	Application exten...	14,458 KB

SC5 - Retention of audio quality:

When assessing [SC5_A], I ran audio through my plug-ins and compared the waveforms before and after processing. If there was compression, there would be less dynamic range in the processed signal. When I tested all my plug-ins, there was no unintentional compression meeting this criterion.

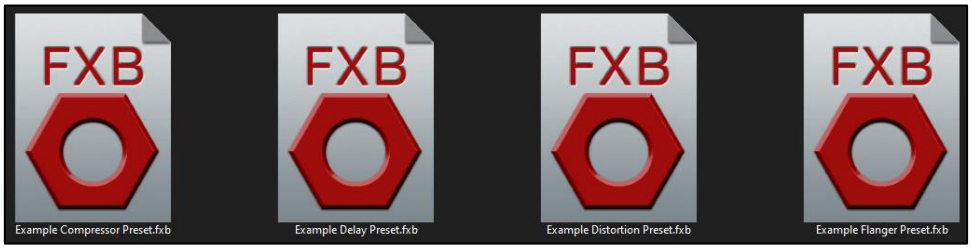
When I was initially testing my plug-ins, I tried putting them on the max settings trying to check whether they introduce artifacts. The Delay and Distortion plug-ins passed with flying colours however I found some issues with my Flanger and Compressor. Owing to my slightly dodgy implementation of a flanger it introduces a subtle number of artifacts which aren't heard during normal operation. When I was testing my compressor with the smallest attack and release values the compressor started to introduce some clicks and unwanted noise owing to the super short timings. Although this doesn't completely address [SC5_B] during normal use these issues won't be noticeable.

During this initial testing I didn't discover any sample rate issue when changing the sample rate or using a clip recorded at the wrong sample rate. Meaning I have met [SC5_C].

With all of this considered I have partially met [SC5] however I will talk about this more later.

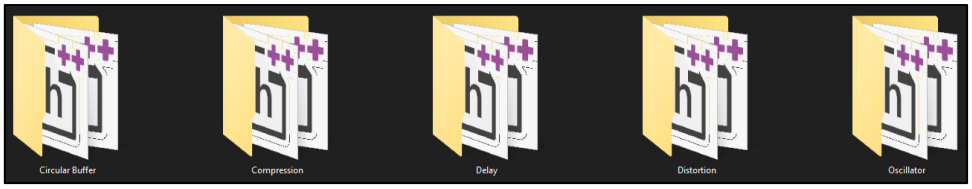
SC6 - Ability to save the current state of the plug-in:

Since iPlug2 natively supports presets, I didn't have to program in any preset handling code. The presets are saved as an fxb file which stores the value of each parameter so they can be loaded in later. After testing this I have achieved [SC6] although I'll talk about this feature more later.



SC7 - Ability to reuse code:

Throughout my project I have been abstracting my code into classes allowing anyone to use and improve upon my code. On my Github repository I have included all these classes in their own folders organised by effect so if I make any improvements or new versions, I can keep everything organised. This successfully addresses [SC7_A].



Alongside this, I have included documentation on how each class works under the documentation section. This explains how key functions work alongside what each variable is used for. I have also included some notes if there is anything I needed to mention. This allows me to meet [SC7_B].

With all of this considered I have successfully addressed [SC7].

SC8 - Stability of code:

When testing the stability of my code, I ran a constant tone through all my plug-ins for 5 minutes and waited to see if there were any errors. After waiting for 20 minutes all my code ran perfectly fine meaning that I have met [SC8_A].

Since it's not easy to see memory usage in a DAW, I ran the plug-ins as standalone applications to measure their memory usage in the task manager. After testing each plug-in, they were all well under the 100mb target even when they were under a heavy load, meaning that I have met [SC8_B].

Name	S...	38% CPU	64% Memory	4% Disk	0% Network	100% GPU
Apps (9)						
CompressorPlugin		0.4%	47.5 MB	0 MB/s	0 Mbps	0%
DelayPlugin		0.5%	50.9 MB	0 MB/s	0 Mbps	0%
DistortionPlugin		0.3%	47.6 MB	0 MB/s	0 Mbps	0%
FlangerPlugin		0.2%	51.0 MB	0 MB/s	0 Mbps	0%

With all of this considered I have successfully addressed [SC8].

Further developments for unmet and partially met criteria

Since I addressed most of my criteria, I don't have much to talk about. The main criteria I didn't meet were, [SC5_B] and [SC6]. With [SC5], I have two options, either I reimplement processing classes with a more technically accurate algorithm or I try to fix my existing algorithm. For the compressor it would be easier to try and fix my current algorithm since it could just be tweaking a few variables and change a few lines of code. However, with the flanger I would need to reimplement the entire processing function making use of a variable sample rate or by creating a new class which would fix any aliasing I created. Since these artifacts are unnoticeable during normal use, this isn't at the top of my priority list since none of the testers mentioned any of them. Although I did meet [SC6], I feel that it's slightly cheating since my framework provides this for me. I do plan to implement my own preset browser into all my plug-ins since it was a requested feature. I could either create my own file format which stores the data for presets, or I could use an xml file. I would also implement a feature to save and load preset banks which would contain multiple preset packs making it easier to distribute preset packs.

Overall limitations and maintenance issues

Since I have finished the project, I have reassessed my limitations and I have thought about some possible maintenance issues. Firstly, all my plug-ins are currently using basic signal processing. Although it is fine for the moment, if I want to expand the project with more interesting plug-ins, I'll need to explore spectral processing. Linking to this, I only have 4 plug-ins finished at the moment. Before I start marketing these plug-ins to a wider audience, I will refine what I have and create a few of the effects I have suggested to give users more incentive to use my software. My plug-ins also have a relatively simple number of features. Although this does fit with my initial goals, alpha testers did mention that sometimes the plug-ins felt like they were lacking more features.

Moving on to maintenance issues. I am a solo developer so finding and fixing bugs will take longer than if I was in a full team of developers. This means that unless users report bugs through my website there could be glaring issues I haven't discovered yet. Finally, since I have a limited knowledge in audio DSP, fixing certain issues like my dodgy implementation of a flanger could be hard and require a lot of time to fix again slowing down maintenance.

Further improvements to the program

Throughout my project I've been suggesting improvements to the bundle which I'll make in the future. Here is the compiled list of all the improvements and further developments to the project:

- Addition of a Phaser, Chorus and Reverb: I was originally planning to program them however I ended up deciding that I wouldn't have enough time to successfully implement all of them.
- Addition of a Parametric EQ: Since it is the only effect missing from my bundle it would allow me to fully complete it and allow me to reuse the code for later projects.
- Porting of software to a physical system: As mentioned at the start I've always wanted to create a physical implementation of my software however since it would be way too complicated to make this a commercial product it would end up being a personal project which I would include documentation for if anyone else wanted to do the same.

- Including more plug-in types: Although support for every plug-in type wasn't in my success criteria, I still believe it is an essential feature hence why it is my main priority after I finish this project.
- Contact form processing on website: Since this is a small project, I haven't added in the code to process the contact form however I do want to fix this when I have some time to learn php.
- Addition of more plug-ins to the project: Throughout feedback with testers and my own ideas I have had a few ideas for new plug-ins I could develop. One being a special type of convolution reverb where you can change the impulse in real time. This would allow me to delve into FFT processing however I would need more experience coding plug-ins before I would be comfortable starting this project.

Final thoughts

Overall, I think this project has been a huge success, although I didn't make as many plug-ins as I hoped I still have made a good start and I will continue this in the future. It has also allowed me to learn a practical application of C++ which will be helpful when furthering my studies at university. I hope that this document will be useful for others wishing to get started with programming audio plug-ins however they don't have any idea where to start since that was where I was at about a year ago.

References

1. Steinberg, "Cubase"
2. Image-Line, "FL Studio"
3. Blackmagic Design, "DaVinci Resolve 18"
