3-Band Parametric EQ

Raphael Sule

Analysis:

Introduction - What is a VST?:

VST* stands for Virtual Studio Technology, and refers to any third-party software that integrates software synthesisers and effects units into digital audio workstations (DAWs). They can emulate the effects of hardware outboard gear with ease, allowing users to compress, expand, maximise and more. VSTs are used by all music producers to add effects on sounds like reverb, distortion etc. and can greatly enhance quality of sound. Other VSTs are used to EQ and mix tracks, and are essential in the processing portion of the development of a song.

VSTis are a specific branch of VSTs. VSTi stands for Virtual Studio Technology Instrument and refers to any VST plugin that utilises one-shot .wav files to produce sound. This means that you can create melodies with this type of plugin. Instead of being created digitally, in order to make high quality VSTis, it's essential that developers find the actual instrument that they want to sample, and record high quality samples to use.

What are some issues with VSTs?

While there are many VSTs already in production, a lot of free VSTs don't match up with the standard that paid software brings, and some features of paid VSTs don't have any freeware that can emulate what they produce. Furthermore, while developers are slowly changing the way VSTs look, many are still quite retro in appearance and a lot harder to understand.

Many VSTs are yet to be updated to appear modernised, and often have several implementations that aren't used due to its difficulty to understand. Because of this, access becomes limited only to producers who have a lot of knowledge/ training, and those just starting out can feel discouraged.

Omnisphere, one of the most popular VSTis to date, hosts hundreds of built in sounds and has thousands community made sounds more available for download.

It also implements effects into itself, allowing easy manipulation of sound.
However, to be able to use the most basic instance of this software, you must be
willing to pay at least $400 or $290,which is the same price of many more popular DAWs themselves.

---

[1]https://www.spectrasonics.net/sales/techshop/?main_page=product_info&products_id=45
*Plugins and VSTs can be used interchangeably

Main characteristics

Upon research, I found some key factors that are used to make up a good VST:

1. An effective GUI: The best VST's in the industry all have an easy to use and aesthetically pleasing interface. The top selling VST of 2021*, is incredibly visual, with almost all methods of interaction coming via the images of the images themselves. As well as just looking nice, an effective GUI enables efficiency and clarity for the programmer's end-users, allowing for more productive usage.

2. Visualisation: I have noticed that a lot of VSTs visualise what they are doing for users to understand. It also helps the VST to function smoother, as it can assist in making more precise changes. One of the best examples of this method is EQ plugins. The sound wave is represented via variable intensities of light, where the brighter the light is the more prominent the frequency of sound. Users can simply drag a slider to soften or cut certain frequencies from their samples.

3. Simplicity: A lot of the best VSTs are recognised because they simply do what is advertised. Sausage Fattener, for example, has two simple sliders: one for 'fatness', or distortion levels, and another for 'colour' or stereo manipulation. A higher level of simplicity means that it is easier to both understand and use, especially for entry and beginners.

4. Likeness: Some VSTs use images of real instruments and even allow usage of the VST through the image. Users may feel more comfortable using a system that is represented by something that they are familiar with. AGML allows for people to use the image of the guitar to create MIDI events in the sample track. With this extra added likeness to actual instruments or objects, users could make higher-level music in a shorter amount of time. They also have an alternative method of creating music within DAWs, rather than just using sample sounds.

---

[1]https://www.spectrasonics.net/sales/techshop/?main_page=product_info&products_id=45
*Plugins and VSTs can be used interchangeably

Identifying end-users:

The primary end-user will also be the person who is testing my VST. He had bought FL Studio expecting to be able to make a wide variety of music, only to find that the stock plugins provided were somewhat limiting and didn't really offer any uniqueness. Some of the free VSTs that he found had effects on his song, but he wasn't actually sure what the audio parameters were. He also informed me that some features that he wanted to employ on his songs simply weren't available: they were gated by a paywall. By showing me his project, I immediately understood what he meant: while I could clearly hear the potential in his track, there was also an undeniable lack of refinement within it. I was asked to make a few different types of VSTs with the hopes of some sleek plugins that enable easy and free use.
With these objectives in mind, the amount of end-users this program can benefit could greatly increase. Many consumers have already spent a lot of money for the DAWs that they use, and will profit from a free but professional set of VSTs to complement their productions.

User Wants

When prompted to identify some VST effects that he wanted created, the primary end-user responded with:

- Reverb: Allows users to be able to manipulate sound in such a way that it persists after the sample has finished. Reverb can be controlled to sound more 'wet' or 'dry', and can also be altered to last for longer durations of time.
- Distortion/overdrive: These are forms of audio signal processing used to alter the sound of amplified electric musical instruments by increasing gain or drive, creating a buzzing effect. It is commonly used on bass or 808s, the lower frequencies of a track.
- A simple synthesiser: This focuses on being able to manipulate various aspects of a synth wave, such as pitch, atack, sustain and decay. This can be considered an incredibly simple VSTi, as it does also produce the sound which the effects are being placed on.
- 3-Band Parametric EQ: This will allow the manipulation of frequencies within a given sound.

When prompted to give other desires, the primary end user said:

- If possible, any plugin should have a sleek and simple design, preferably in darker colours.
    - This is more akin to GUI creation, which I don't have extensive knowledge of, so it will be difficult for me to be able to easily create a whole design for these plugins. Upon research, I found that most people create UI interfaces themselves in editing applications ie. Photoshop. However, C++ and JUCE allows for easy GUI manipulation.

- Any plugin should be simple and easy to use and understand.
    - To do this, a simple slider based system will make it easier to change effects and give more control to the user.

I decided that an EQ was the best option to make for my end user. This is because EQ's are arguably the biggest staple out of all VST types. A 3-band parametric EQ is preferable over a 7-band parametric EQ. This is due to its user-friendliness as there are less bands to manipulate.

Objectives

1. **Create a VST that:**
   - 1.1. Is able to take in sound.
     - 1.1.1. By sample.
     - 1.1.2. By MIDI controller.
   - 1.2. Can process sound.
     - **1.2.1. Create a chain of filters containing:**
       - 1.2.1.1. Low cut
       - 1.2.1.2. Peak
       - 1.2.1.3. High cut

         (evident if the EQ works)
     - **1.2.2. Initiate two chain instances:**
       - 1.2.2.1. left chain and right chain that will emulate stereo sound.

         (Evident if the EQ works)
2. **Use DSP to initialise and run effects on a given sample sound:**
   - 2.1. Create a basic initial state for each audio parameter.
   - 2.2. Allows the user to be able to layer the VST multiple times on one audio track with no errors.
     - 2.2.1. Can run two EQ's with identical settings concurrently.
     - 2.2.2. Can hear the difference between zero, one and two EQ's with identical settings running on one audio track, thus proving they can layer.
3. **Create a custom GUI with:**
   - 3.1. A dark background layer.
   - 3.2. Custom sliders for each audio parameter.
     - 3.2.1. Each slider will be in its own area.
     - 3.2.2. Each slider will have their name within them.
   - 3.3. An entry box that shows the current value that the slider is at.
     - 3.3.1. The entry box will serve as an alternative to moving the slider with the mouse.
     - 3.3.2. The entry box will snap to the nearest allowed value if a value out of range is entered.

4. Is able to output EQ'd sound:

4.1. EQ'd Sound plays when the sliders are turned:

Slopes:

4.1.1. The **low cut slope cuts more** of the lower frequencies when the **slider is increased.**

4.1.2. The **high cut slope cuts more** of the higher frequencies when the **slider is decreased.**

4.1.3. The **low cut slope cuts less** of the lower frequencies when the **slider is decreased.**

4.1.4. The **high cut slope cuts less** of the higher frequencies when the **slider is increased.**

4.1.5. The **low cut slope has a steeper attenuation when a higher cutoff is chosen** by the user.

4.1.6. The **low cut slope has a more gentle attenuation when a lower cutoff is chosen** by the user.

4.1.7. The **high cut slope has a steeper attenuation when a higher cutoff is chosen** by the user.

4.1.8. The **high cut slope has a more gentle attenuation when a lower cutoff is chosen** by the user.

Peak:

4.1.9. The **peak band moves to higher frequencies** when its **slider is increased.**

4.1.10. The **peak band moves to lower frequencies** when its **slider is decreased.**

4.1.11. The **quality of the peak band becomes slimmer** when the **slider is increased.**

4.1.12. The **quality of the peak band becomes wider** when the **slider is decreased.**

4.1.13. The **gain of the peak band increases** when the **slider is increased.**

4.1.14. The **gain of the peak band decreases** when **the slider is decreased.**

4.2. Outside of the EQ, I can bypass the filters and hear unfiltered audio.

Supplementary Objective

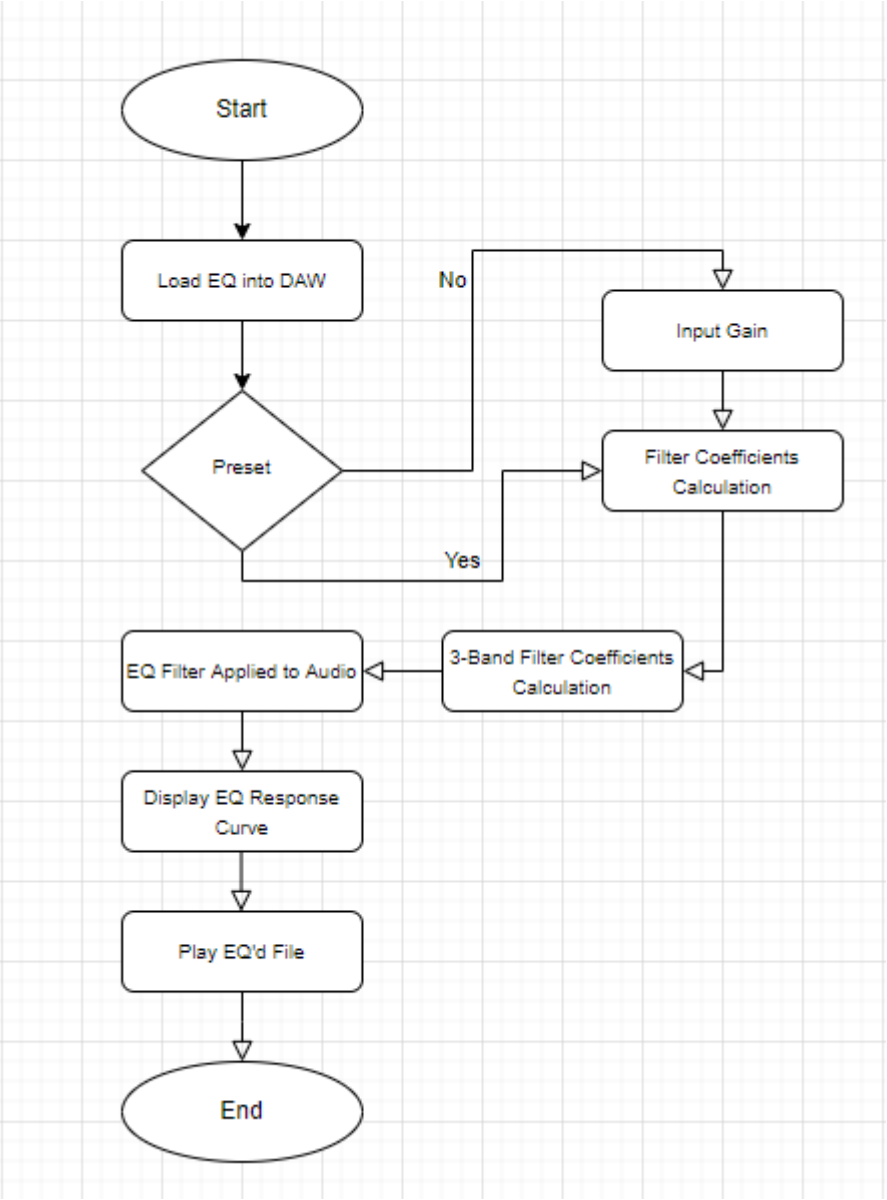
Usage of a response curve and spectrum analyser for visualisation of what the VST is doing.

The response curve(white) shows what the filters actually are filtering out against a of the sample sound.

The spectrum analyser (green) shows the frequencies and gain where the sample is making sound.

Simple Process Flowchart

The EQ will follow some simple steps:



1. The EQ must be located on the user's PC, either as a standalone plugin or through a DAW.
2. The user can input a EQ 'save state' if being used inside a DAW ie. '808 Preset'.
3. If a preset is not chosen the user must 'input gain', or manipulate their filter settings.
4. Concurrently, the EQ will begin to calculate filter coefficients and apply them to the audio, either raising or lowering frequencies of the sample. Some frequencies will be cut out altogether.
5. The frequency changes should also be displayed on the response curve in real-time.
6. The EQ'd audio sample should play with the effects implemented on to it, resulting in a sound that is fully filtered to the user's wants.

Similar Programs:

I will be taking some inspiration from these plugins:

Fruity Parametric EQ[2]**:** An EQ that comes as part of a VST expansion pack for FL Studio. While it is functional, it costs money to buy.

Prime EQ[2]**:** A real time parametric EQ VST with a real-time spectrum analyser.

Evidence of Analysis

I decided to take two precautions to better understand the issues the user was facing: as well as conducting an interview, I decided to try and create music first-hand, using the DAW that he had: FL Studio. Fortunately, the end-user was also able to bring a third-party to assist and teach me throughout this process, who greatly improved the VST usage and brought their potential out much more.

Interview

For the interview, my main goals were to:

- try and understand what users would want most from the VSTs
- understand what VSTs were and why they were so helpful
- See what the end-user thought were the reasons for VSTs being locked behind paywalls.

What is the most important outcome of this project to you?
I just want the VSTs to be easy to use. There are amazing plugins that are free but are archived on page 100 in Google, never to be found by any producer. Then there are so many FREE plugins that I find that seem like gems, only for them to end up not being used because I have no idea how to use them. When I try to find tutorials about them, the only things that come up are promoters showing ten second clips of beats that feature them, which doesn't really help at all. It isn't like there isn't anything that I can find, but it would be nice to have them all accessible in one place.

Why are VSTs so important to you?
VSTs are the easiest way to create variety in your music. They can do so much to your music, and elevate your game by levels. There isn't anyone at the top of the game who doesn't use VSTs: that makes no sense. It's not like VSTs are just important to me, it's a basic part of music production.

How would you like VSTs to be laid out?
Simple would definitely be best. Producers like to make music in the dark, so if you use a dark design with white sliders, then the design will be really good. Something to tie it all together to make it unique too, maybe a streak of blue or something would be cool. If instructions could be implemented, that would be even better.

Why do you think VSTs are so expensive?
At the end of the day, a lot of VSTs are directed towards a very small number of people. Only producers who are truly willing to dedicate their lives to the craft will end up buying them, and that's even if they can afford it. People who make these programs can choose how much they want to make them for, and that's why they cost as much money as they do.

Do you think your issues will be solved by this project?
I think that over a period of time, with many revisions to the coding in this project, it could become something that could potentially be useful for producers everywhere. Starting from scratch, I think the first objective should be to produce something that works first. From there, the code can be altered and refined into something more professional. I don't know much about coding, but judging from the fact that companies charge so much for their products, I'm guessing that the process could be either very difficult or just tedious.

[2]https://www.image-line.com/fl-studio-learning/fl-studio-online-manual/html/plugins/Fruity%20Parametric%20EQ%202.htm

[3]https://www.voxengo.com/product/primeeq/

Creating music

To best try and understand the type of effects that each VST would have, I created three tracks.I created a track with no effects used, and then used the stock plugins provided with the program. Finally, I used VSTs the interviewee's friend provided to create the most 'professional' version of the track, and see if there were any noticable differences between paid and stock plugins.

Track 1: Track one didn't use VSTs to augment the sound. The only VSTi that I used was FL Keys: a simple piano sound. While I could change parameters like the harshness of the sound, it only sounded average.

Plugins used: N/A

Track 2: Track two only used stock VSTs to augment the sound. The only VSTi that I used was FL Keys: a very simple piano sound. While I now could use a few effects, it still sounded just above average.

Plugins used: Fruity Delay Bank, Fruity EQ, Gross Beat

Track 3: Track three used both stock and third-party plugins to augment the sound. Using more professional plugins like Halftime rather than Gross Beat really enhanced the sound. Having free control over effect usage elevated the track. There was a noticeable difference between Track 2 and 3.

Plugins used: SimpRev, Halftime, Fruity EQ, Vinyl

The results of this test can be found here: [https://youtu.be/GYbLqeuqBwY]

After conducting the experiment and the interview, it became clear that VSTs were vital for music production.

Documented Design:
Parametric EQ

Design Overview

*note: By convention, .cpp files contain definitions and .h files are declarations. This means that header files (.h) are used to list accessible methods and variables . Implementation files (.cpp) are used to actually implement those methods and variables.

JUCE plugins have two main classes: PluginProcessor and PluginEditor. PluginProcessor handles the audio and processing logic, whereas PluginEditor handles visualisations and GUI controls.

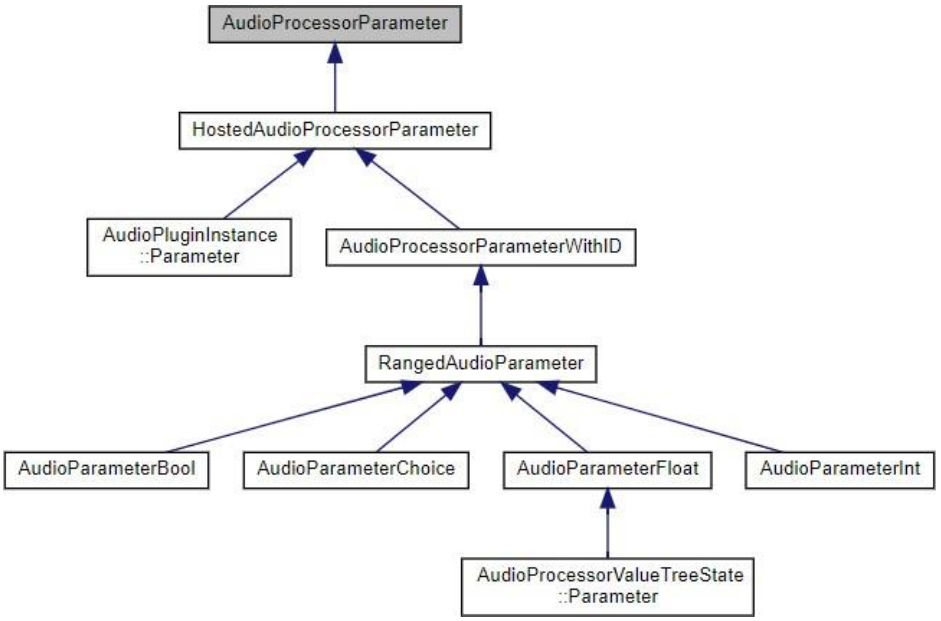
The PluginProcessor can be seen as the 'parent node': there can only be one PluginProcessor but there can be multiple editors.

PluginEditor.h / PluginEditor.cpp: The main method I will edit is the constructor, used to initialise the window and GUI components, and set up new custom GUI components (via paint()).

PluginProcessor.h / PluginProcessor.cpp: The main method I will edit is the processBlock() method, which takes and gives audio and MIDI data to the plugin output.

Setting Up Audio Parameters

Audio Parameters are used to connect the GUI and DSP. The AudioProcessorValueTreeState is what links together the AudioParamaters and the AudioProcessor class.



AudioProcesorValueTreeState Class

This class will enable the connection between the GUI and the data in the audio processor.

Juce has an AudioProcessorValueTreeState class, used to initiate parameters such as sliders and checkboxes. My audio parameters require me to use AudioParamaterFloats, which are those that take a range of real values. This is because I can use this to manipulate a continuous flow of values, for example the parametric bands represented using a slider.

I will also use the AudioProcesorValueTreeState constructor directly to create the layout of my VST. It can take a collection of AudioParameters and add them to the attached **AudioProcessor** directly. These are parameters it can take:

- Name: The name of the Parameter that will appear in the VST.
- Parameter Name: The name you give to the Parameter.
- Normalisabe Range: The lowest and highest values the slider can reach.
- Step: The increment value the slider will follow ie. 1 Hz or 10 Hz.

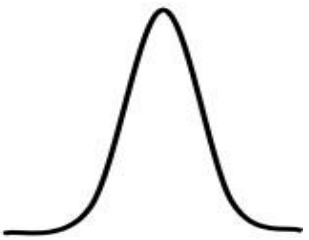
- Skew: Skew is the deviation from the normal distribution. This means that a larger portion of the slider could represent less values ie. 50% of the slider representing 10 Hz out of 100Hz rather than 50 Hz.

I used this parameter for the **Peak EQ, Peak Gain, Peak Quality, Low Cut, High Cut, and for both the Low and High Cut Slope cutoffs.**

Skew: When two frequencies have a ratio of 2^n:1 for any natural number n, they are said to be n octaves apart. This results in a phenomenon where if the sound goes up **one octave**, the frequency **doubles.** So, if an octave was between 16 and 32, the next octave would be double 64. This means that it will take double the frequencies of the one octave to reach the next octave. Thus, the skew of each parameter must be manipulated in such a way that this exponential relationship can be represented by a linear slider. To do this, I will make a greater proportion of low-end frequencies represent a larger proportion of the slider.

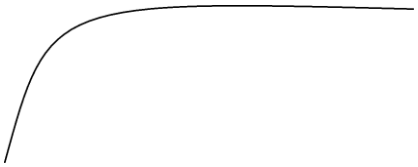
Parametric Peak & Quality/Gain

*Response curve representation

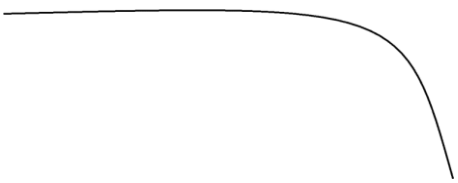


- Peak Gain refers to changes to the gain/ volume of the parametric band.
- Peak Quality refers to the broadness of the peak band.

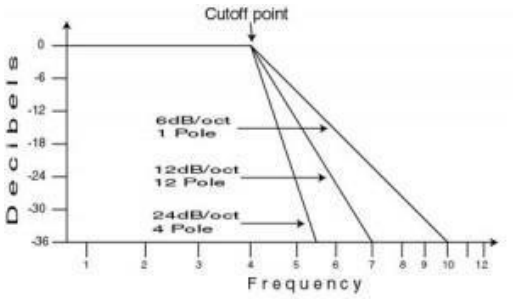
Low/High Cut



- A Low Cut is used to eliminate lower frequencies within a sound or sample. The further to the right you move the Low Cut filter, the bigger the range of lower frequencies it will remove from the sound.



- A High Cut is used to eliminate higher frequencies within a sound or sample. The further to the left you move the High Cut filter, the bigger the range of upper frequencies it will remove from the sound.



- The Low and High Cut Slope both have 4 separate steepness options, from the gentle -12 dB/oct choice to the steep -48 dB/oct choice, in increments of -12 db/oct. The steeper the slope, the harsher the frequency will be. A harsher frequency eliminator takes out more frequencies.

This is where I added the peak band parameters to the layout: the peak frequency, gain and quality.

```
//Parametric Peak
layout.add(std::make_unique<juce::AudioParameterFloat>("PeakFreq", "Peak Freq", juce::NormalisableRange<float>(20.f, 20000.f, 1.f, 0.25f), 750.f));
//Peak Gain
layout.add(std::make_unique<juce::AudioParameterFloat>("peakGain", "Peak Gain", juce::NormalisableRange<float>(-24.f, 24.f, 0.5f, 1.f), 0.0f));
//Peak Quality (How wide / narrow the peak is)
layout.add(std::make_unique<juce::AudioParameterFloat>("peakQuality", "Peak Quality", juce::NormalisableRange<float>(0.1f, 10.f, 0.05f, 1.f), 1.f));
```

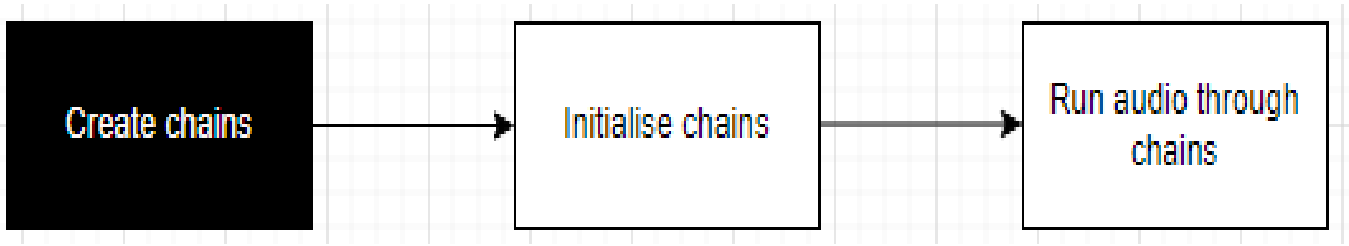
This is where I added both the low cut and the high cut frequencies to the layout.

```
//LowCut Frequency
layout.add(std::make_unique<juce::AudioParameterFloat>("lowcutFreq", "LowCut Freq", juce::NormalisableRange<float>(20.f, 20000.f, 1.f, 0.25f), 20.f));
//HighCut Frequency
layout.add(std::make_unique<juce::AudioParameterFloat>("highcutFreq", "HighCut Freq", juce::NormalisableRange<float>(20.f, 20000.f, 1.f, 0.25f), 20000.f));
```

This is the for loop used that I used to create the string for each choice within both the high cut and low cut slopes.

```
juce::StringArray stringArray;
for (int i = 0; i < 4; ++i) // I IN RANGE 1 TO 4, INTERVAL OF 1
{
    juce::String str; str << (12 + i * 12); str << " dB/Oct"; stringArray.add(str);
}
```

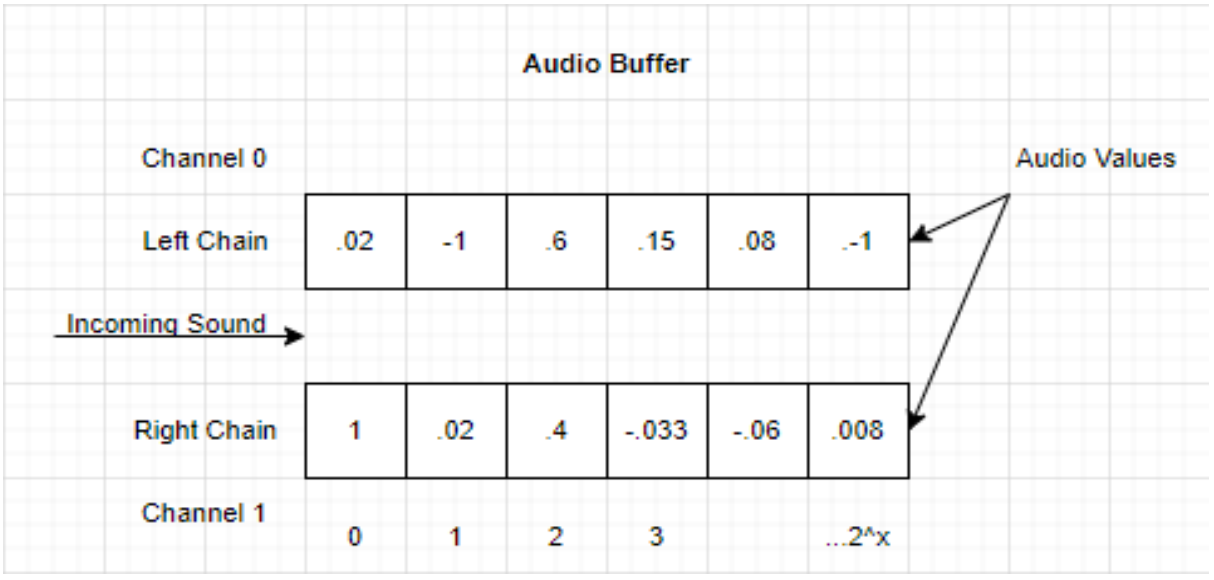
Initialising Digital Signal Processing



DSP is used to process and output filtered sound. For the EQ, I can create a chain of filters, and the DSP can run on each filter in sequence instead of me having to declare each filter separately every time I want to filter sound. Every SingleChain has 3 links within them: one for the LowCut, one for the peak band, and one for the HighCut.

DSP automatically processes audio in mono chains. This means that there is a singular channel where sound is manipulated; there is no way to process stereo sound, which has separate channels for the left and right ear. In order to rectify this issue, I will declare two mono channels, which is the same as one stereo channel.

```
using Filter = juce::dsp::IIR::Filter<float>;

using VariableCut = juce::dsp::ProcessorChain<Filter, Filter, Filter, Filter>; // 12 dB/Oct * 4 = 48dB/Oct

using SingleChain = juce::dsp::ProcessorChain<VariableCut, Filter, VariableCut>;
//dsp defaults as mono instead of stereo, so creating a left and right channel
//to play concurrently will result in stereo sound

SingleChain leftChain, rightChain;
```



The left chain represents Channel 0, or the left channel, and the right chain represents Channel 1, or the right channel.

```
const std::integral_constant<int, 0> LowCut;
const std::integral_constant<int, 1> Peak;
const std::integral_constant<int, 2> HighCut;
```

I also create 3 constants so that when I need to call each part of the chain, I can do so with them.

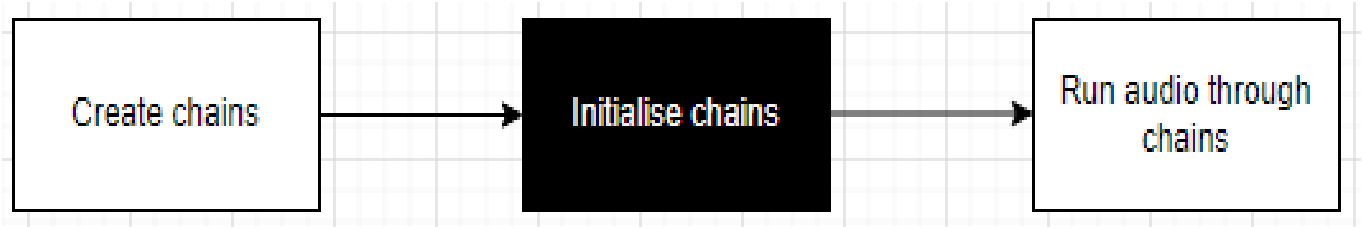
Filter is part of the IIR class, which means Infinite Impulse Response. It takes past and present inputs along with past outputs in order to generate output signals. IIR filters are recursive, which means that it can never respond with a value of 0, and the filter also results in feedback of the output. This means that the filter will return sound that has been filtered.

Using the IIR class rather than the FIR (Finite Impulse Response) allows for a:

Lower implementation cost: requires less coefficients and memory than FIR filters in order to satisfy a similar set of specifications i.e cutoff frequency and low and high cut attenuation.

Lower latency: suitable for real-time control and very high-speed applications due to the lower number of coefficients, and hence, calculations.

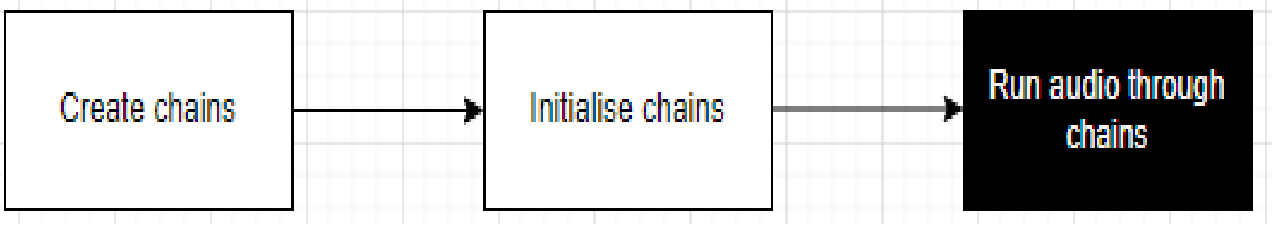
VariableCut refers to the Low and High CutSlope Cut. Juce VSTs has a convention of using multiples of -12 db/Oct options, and in order to achieve the options of -12 to -48 dB, I will use 4 -12dB filters.
Finally, I create the complete chain using VariableCut to represent the low and high cut, and a singular filter to represent the peak band.



- sampleRate: The rate at which data will be sent to the processor.
- maximumBlockSize: The maximum number of samples that can be in a block.
- numChannels: The number of channels the processor can expect to process. In this case, there are two separate channels, leftChain and rightChain.

To do this, there is a prebuilt x.**prepare()** function which prepares all inner processors with a given process specification. I can call the prepare function on both the left and right chain, resulting in both chains now being ready to have 'contexts' passed through them.

```
// pass a process spec object to both the left and right chain, this will be passed to each link in the chain
// this prepares each chain to be prepared for processing

juce::dsp::ProcessSpec shem;

shem.maximumBlockSize = samplesPerBlock;

shem.numChannels = 1;

shem.sampleRate = sampleRate;

leftChain.prepare(shem);
rightChain.prepare(shem);
```

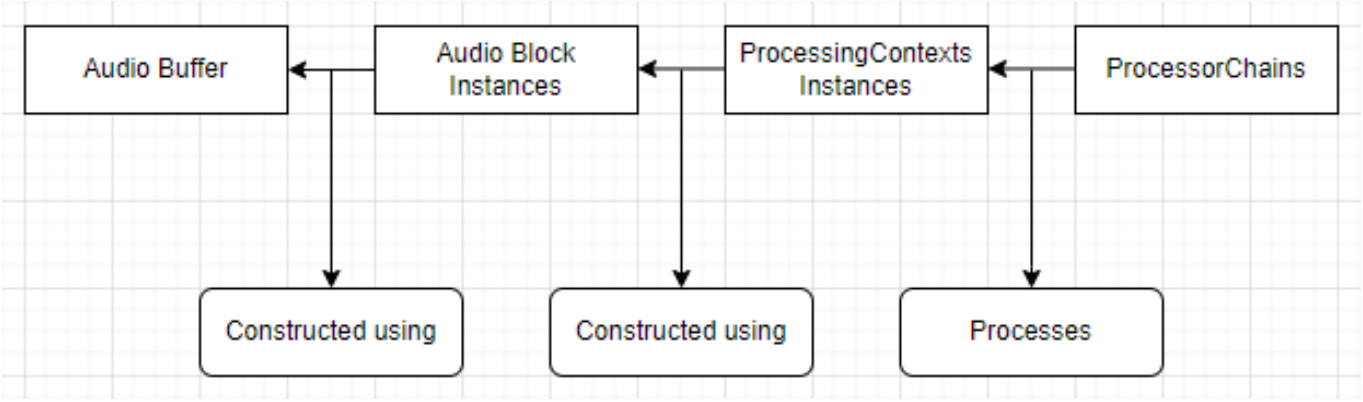


After initialising the chains, they are now ready to have audio run through them. Audio is run through the chains in the form of continuously updating values from the buffer. However, chains can only take these n values from a pre-built function

which requires the use of audio blocks, rather than the buffer itself. An audio block simply points to values in an audio buffer: this protects the incoming values and prevents the code from changing them in any way.

```
juce::dsp::AudioBlock<float> block(buffer);

auto leftBlock = block.getSingleChannelBlock(0);
auto rightBlock = block.getSingleChannelBlock(1);
```

getSingleChannelBlock returns the entire block, which points to all the values inside the buffer.



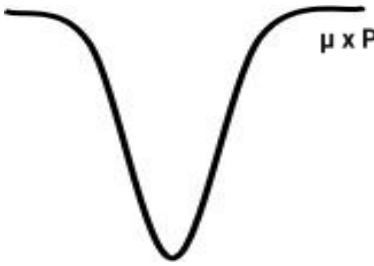
After getting both the left and right Block, I use the needed **ProcessContextReplacing** pre-built function so that contexts can be passed into the process function. The ProcessContextReplacing function runs through each link in the chain automatically, and returns instances of contexts that the processor can understand.
The ProcessContextReplacing instances are constructed with AudioBlocks, which is why I used the getSingleChannelBlock for both the left and right chain.

Because the audio buffer is always updating, the audio block instances are always changing, which the processing contexts use to create new instances to be processed.

Initiating and processing filters

Before the peak filter can be applied to sound, I need to first pull the coefficients of the peak filter. A coefficient is a multiplier or factor that measures a particular property. The reason why I need the coefficients for the peak filter in particular is because these are real-world, denormalised values. If the value that gets returned is normalised, it must begin with the integer 1. This means that the plugin would not be able to have any coefficient value of below 1. This is vital because a coefficient of below one would represent a cut rather than a boost of frequencies.

One example which demonstrates the need for denormalised values is the peak band. A peak band with a coefficient of more than one will result in a boosted peak band.



However, a peak band with a coefficient less than one will result in mellowed frequencies instead of boosted frequencies.

Firstly, I created a struct with all of the audio parameters, as well as a default value. This is the value that will be called in the prepareToPlay function. After this, I defined the function **getchainsettings** that will return the values of each parameter and store it in the tuple.

```
//creating a structure so that the apvts can pull these values every time it is called, rather than having to write them out over and over.
struct chainsettings
{
    float peakFreq{ 0 }, peakGain{ 0 }, peakQuality{ 1.f };
    float lowCutFreq{ 0 }, highCutFreq{ 0 };
    int lowCutSlope{ 0 }, highCutSlope{ 0 };
};

chainsettings getchainsettings(juce::AudioProcessorValueTreeState& apvts);
```

After this struct is created, I can define the function getchainsettings. getRawParameterValue returns a pointer to each parameters' RawParameterValue. load() returns this value. This function extracts the denormalized values I need, which will be in the range I defined when I defined each audio parameter.

```
chainsettings settings;

settings.lowCutFreq =
apvts.getRawParameterValue("lowcutFreq")->load();
settings.highCutFreq =
apvts.getRawParameterValue("highcutFreq")->load();
settings.peakFreq =
apvts.getRawParameterValue("PeakFreq")->load();
settings.peakGain =
apvts.getRawParameterValue("peakGain")->load();
settings.peakQuality =
apvts.getRawParameterValue("peakQuality")->load();
settings.lowCutSlope =
static_cast<Slope>(apvts.getRawParameterValue("lowcutSlope")->
load());
settings.highCutSlope = static_cast
<Slope>(apvts.getRawParameterValue("highcutSlope")->load());

return settings;
```

I created the variable getchainSettings and assigned getchainsettings(apvts) to it. This is so I can pull the raw values for each parameter and initiate each slope and filter.

I will use getchainSettings in two places: the prepareToPlay method and the processBlock method.

peakCoefficients:

```
juce::dsp::IIR::Coefficients<float>::makePeakFilter(getSampleR
ate(), getchainSettings.peakFreq,
getchainSettings.peakQuality,
juce::Decibels::decibelsToGain(getchainSettings.peakGain));
```

getchainSetting parameters: getchainSettings.peakFreq, getchainSettings.peakQuality, juce::Decibels::decibelsToGain(getchainSettings.peakGain))

makePeakFilter will return coefficients for a peak filter around a given frequency (getchainSettings.peakFreq), a given quality getchainSettings.peakQuality and a given gain (getchainSettings.peakGain).
By default the peak gain's value is in dB, but the function takes the value as gain, so it must be converted.

```
*leftChain.get<Peak>().coefficients = *peakCoefficients
*rightChain.get<Peak>().coefficients = *peakCoefficients;
```

These lines use the pointer peakCoefficients, dereferences it to access the data it points to, and then assigns these values to the coefficient that is held in the Peak band of the chain.

lowCutCoefficients:

```
juce::dsp::FilterDesign<float>::designIIRHighpassHighOrderButterworthMethod(getchainSettings.lowCutFreq, getSampleRate(), (getchainSettings.lowCutSlope + 1) * 2);
```

highCutCoefficients:

```
juce::dsp::FilterDesign<float>::designIIRLowpassHighOrderButterworthMethod(getchainSettings.highCutFreq, getSampleRate(), (getchainSettings.highCutSlope + 1) * 2);
```

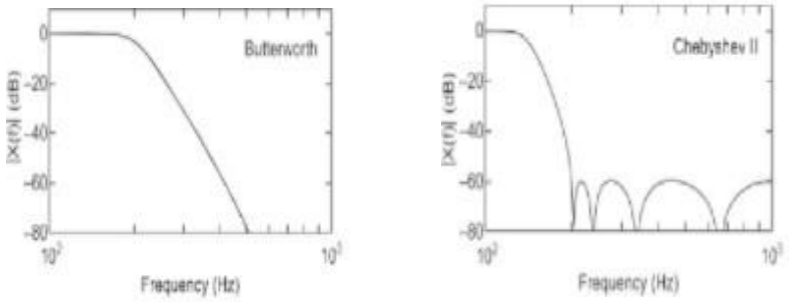
designIIRHighpassHighOrderButterworthMethod:
getchainSetting parameters: getchainSettings.lowCutFreq, getchainSettings.lowCutSlope

designIIRLowpassHighOrderButterworthMethod
getchainSetting parameters: getchainSettings.highCutFreq, getchainSettings.highCutSlope

The **designIIRHighpassHighOrderButterworthMethod** is the method used to generate the low cut IIR coefficients (Highpass also means low cut). The **designIIRLowpassHighOrderButterworthMethod** is the method used to generate the low cut IIR coefficients (low pass also means high cut).

These methods take 3 parameters: the cutoff frequency of the high-pass filter, the sample rate being used in the filter design, and the order of the resulting IIR filter with an attenuation of (-6 dB*order) / octave.
The Butterworth IIR methods I will be using default to a **-6 dB/oct cutoff**, which isn't as steep a cut as other filters ie. the Chebyshev type II filter. However,it is completely flat in

the passband and monotonic overall, leading to the most consistent filter.



source:https://www.sciencedirect.com/topics/engineering/iir-filters

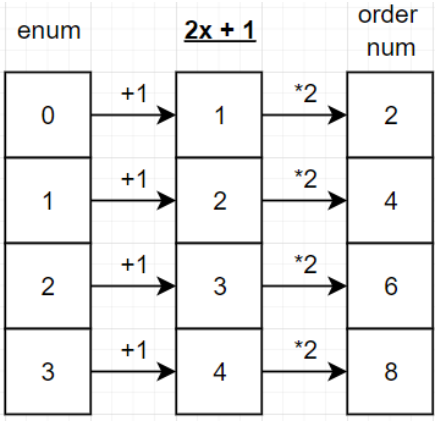
The Butterworth method takes integral constants rather than integers as the order of the IIR filter. To declare integers as constants, I created an enum Slope. Slope1 corresponds to the constant 0, Slope2 the constant 1, Slope3 the constant 2 and Slope4 the constant 3. This enum can be used for the Highpass Butterworth method as well as the Lowpass Butterworth method.

```
enum Slope
{
    Slope1,
    Slope2,
    Slope3,
    Slope4
};
```

I also updated the struct so that the low and high cut slopes hold the Slope enums rather than actual integers.

```
lowCutSlope{ Slope::Slope1 }, highCutSlope{ Slope::Slope1 };
```

The Slope attenuation options I have used are -12, -24, -36 and -48 dB/Oct. Each order attunes the cutoff by -6db. So for Slope1, which returns the integral 0, I need to produce an order value of 2 (-6*2=-12). For Slope2 needs an integral of 4, Slope3 needs an integral of 6 and Slope4 needs an integral of 8. The mathematical process of getting from each enum to order value is as follows:



Firstly, I add 1 to the enum's value. Then I multiply it by 2. This produces the correct order number that will be multiplied by -6db/Oct to produce my low and high slope frequency cutoff.

2 * -6dB/Oct = -12dB/Oct
4 * -6dB/Oct = -24dB/Oct
6 * -6dB/Oct = -36dB/Oct
8 * -6dB/Oct = -48dB/Oct

```
auto& RightLowCut = rightChain.get<lowCut>();
auto& leftLowCut = leftChain.get<lowCut>();
auto& RightHighCut = rightChain.get<highCut>();
auto& leftHighCut = leftChain.get<highCut>();
```

The low and high cuts are composed of 4 filters.

```
using VariableCut = juce::dsp::ProcessorChain<Filter, Filter, Filter, Filter>; // 12 dB/Oct * 4 = 48dB/Oct

using SingleChain = juce::dsp::ProcessorChain<VariableCut, Filter, VariableCut>;
```

I will first bypass all four filters.

```
auto& RightLowCut = rightChain.get<LowCut>();
RightLowCut.setBypassed<0>(true);
RightLowCut.setBypassed<1>(true);
RightLowCut.setBypassed<2>(true);
RightLowCut.setBypassed<3>(true);

auto& leftLowCut = leftChain.get<LowCut>();
leftLowCut.setBypassed<0>(true);
leftLowCut.setBypassed<1>(true);
leftLowCut.setBypassed<2>(true);
leftLowCut.setBypassed<3>(true);
```

For every slope attenuation option chosen, I will turn on that number of filters. So, for example, if -24 dB/Oct is chosen, I will set LeftLowCut.setBypassed<0> and LeftLowCut.setBypassed<1> to false, as there will be 2 -12dB/Oct filters which are active.

```
switch (getchainSettings.lowCutSlope)
```

getchainSettings.lowCutSlope and getchainSettings.lowCutSlope now hold slope enums instead of integers, meaning that I can use the Slope enums as cases in the switch-case statement.
A switch-case statement is similar to an if-else statement, but it does not break if a condition is met. This means I can implement fallthrough in my code.
If there is fallthrough, Whenever a case is ended in a switch, the code of the next case will get executed unless specified otherwise with a break statement.

```
switch (getchainSettings.lowCutSlope)
{
        case Slope4:
        {
            *RightLowCut.get<3>().coefficients = *lowCutCoefficients[3];
            RightLowCut.setBypassed<3>(false);
            *leftLowCut.get<3>().coefficients = *lowCutCoefficients[3];
            leftLowCut.setBypassed<3>(false);
            //[[fallthrough]];
        }

        case Slope3:
        {
            *RightLowCut.get<2>().coefficients = *lowCutCoefficients[2];
            RightLowCut.setBypassed<2>(false);
            *leftLowCut.get<2>().coefficients = *lowCutCoefficients[2];
            leftLowCut.setBypassed<2>(false);
            //[[fallthrough]];
        }
        case Slope2:
        {
            *RightLowCut.get<1>().coefficients = *lowCutCoefficients[1];
            RightLowCut.setBypassed<1>(false);
            *leftLowCut.get<1>().coefficients = *lowCutCoefficients[1];
            leftLowCut.setBypassed<1>(false);
            //[[fallthrough]];
        }
        case Slope1:
        {
            *RightLowCut.get<0>().coefficients = *lowCutCoefficients[0];
            RightLowCut.setBypassed<0>(false);
            *leftLowCut.get<0>().coefficients = *lowCutCoefficients[0];
            leftLowCut.setBypassed<0>(false);
            //[[fallthrough]];
        }
}
```

These lines use the pointer lowCutCoefficients[x], dereferences it to access the data it points to, and then assigns these values to the coefficient that is held in the Peak band of the chain.
For whatever slope you use, all the code underneath it will also be run. So for Slope4, where all the filters are active, there will be a ‘fallthrough’, where all the other filters will also be activated and filter sound.
The high cut filters use the same code as the low cut filters, but replaces every ‘LowCut’ with ‘HighCut’.

All of the code in processBlock is similar to the prepareToPlay, with the exception of the parameter getSampleRate() in the coefficients being changed to sampleRate. This is because the sample doesn’t need to continuously be updated.

Creating Custom GUI

In order to return my own GUI rather than JUCE's generic GUI, I first have to change the `createEditor()` function in pluginprocessor.cpp from:

```
return new juice::GenericAudioProcessor (*this);
```

to:

```
return new EQAudioProcessorEditor (*this);
```

I want my code to use rotary sliders, so that when the user turns the slider, it returns the frequency of the point that the slider has been turned to. I will use a rotary slider for all of the parameters previously declared within the code:

- Peak Frequency, Gain and Quality;
- Low cut slope and freq;
- High cut slope and freq;

```
struct RotarySlider : juce::Slider
{
    RotarySlider() : juce::Slider(juce::Slider::SliderStyle::RotaryHorizontalVerticalDrag, juce::Slider::TextEntryBoxPosition::NoTextBox)
    {

    }
};
```

I create a struct which will make instances of the rotaryslider derivative from the slider class. The slider style name is HorizontalVerticalDrag, and I have also chosen to create each instance without a text box. I then create an instance of each parameter.

```
RotarySlider peakFreqSlider;
RotarySlider peakGainSlider;
RotarySlider peakQualitySlider;
RotarySlider lowCutFreqSlider;
RotarySlider highCutFreqSlider;
RotarySlider lowCutSlopeSlider;
RotarySlider highCutSlopeSlider;
```

After this, I make them appear in the GUI's constructor.

```
addAndMakeVisible(lowCutSlopeSlider);
addAndMakeVisible(highCutSlopeSlider);
addAndMakeVisible(highCutFreqSlider);
addAndMakeVisible(lowCutFreqSlider);
addAndMakeVisible(peakQualitySlider);
addAndMakeVisible(peakGainSlider);
```

```
addAndMakeVisible(peakFreqSlider);
addAndMakeVisible(lowCutSlopeSlider);
addAndMakeVisible(highCutSlopeSlider);
```

Connecting the Sliders and Parameters

After creating the actual slider objects in the GUI, I then have to connect them to the parameters. If I were to run the code at this point, the sliders would all show up, but they would not do anything. The JUCE framework has a built in slider attachment object in the AudioProcessorValueTreeState class, which simply maintains a connection between a parameter and a slider.

Firstly, I create an instance for each parameter:

```
juce::AudioProcessorValueTreeState::SliderAttachment peakFreqAttachment;
juce::AudioProcessorValueTreeState::SliderAttachment peakGainAttachment;
juce::AudioProcessorValueTreeState::SliderAttachment peakQualityAttachment;
juce::AudioProcessorValueTreeState::SliderAttachment lowCutFreqAttachment;
juce::AudioProcessorValueTreeState::SliderAttachment highCutFreqAttachment;
juce::AudioProcessorValueTreeState::SliderAttachment lowCutSlopeAttachment;
juce::AudioProcessorValueTreeState::SliderAttachment highCutSlopeAttachment;
```

And then I declare each one in the construction of the GUI. Each slider attacher takes three parameters:

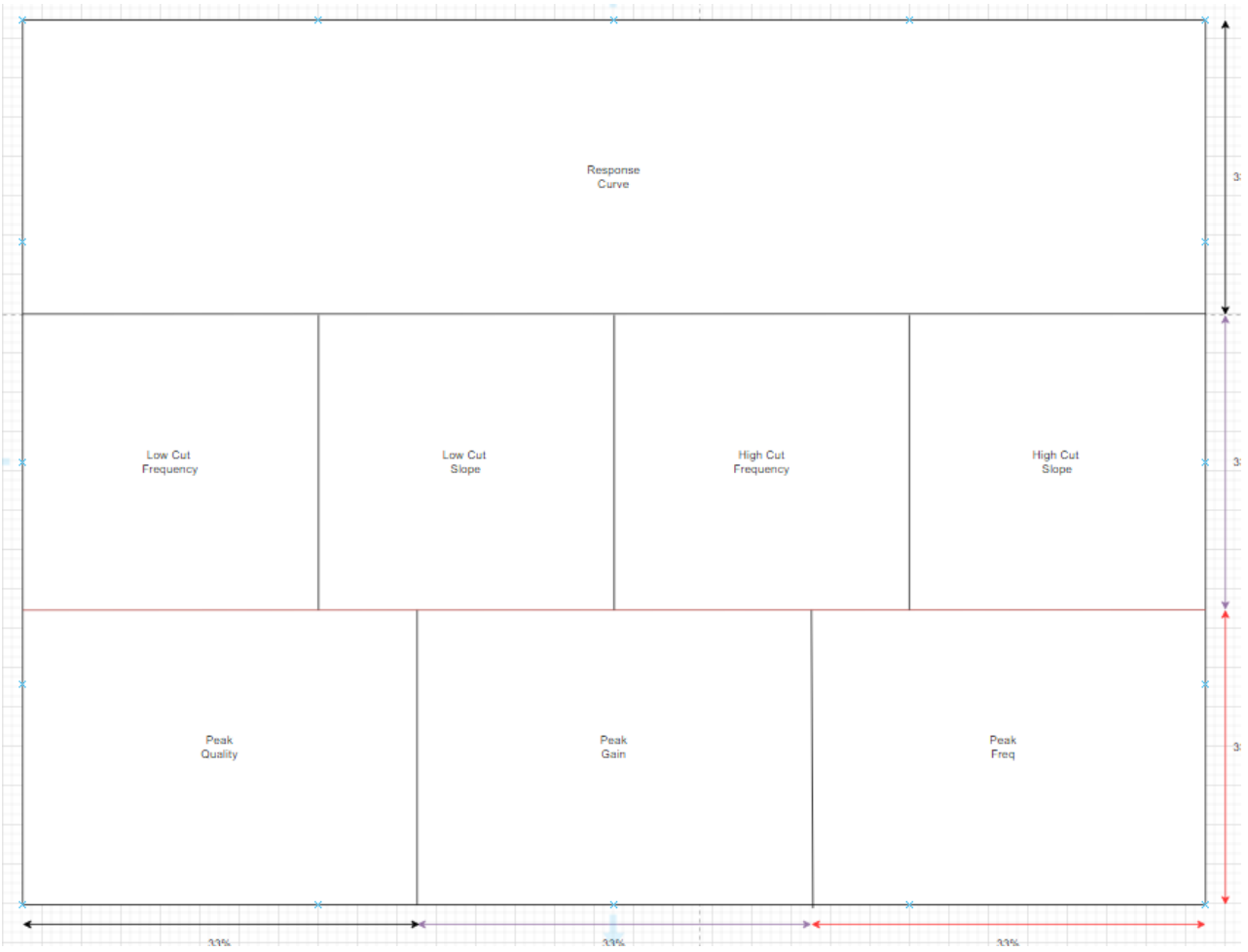
- The location of the parameter
- The parameter ID
- The slider to which the parameter is being attached.

```
peakFreqAttachment(audioProcessor.apvts, "peakFreq", peakFreqSlider),
peakGainAttachment(audioProcessor.apvts, "peakGain", peakGainSlider),
peakQualityAttachment(audioProcessor.apvts, "peakQaulity", peakQualitySlider),
lowCutSlopeAttachment(audioProcessor.apvts, "lowCutSlope", lowCutSlopeSlider),
highCutSlopeAttachment(audioProcessor.apvts, "highCutSlope", highCutSlopeSlider),
lowCutFreqAttachment(audioProcessor.apvts, "lowCutFreq", lowCutFreqSlider),
highCutFreqAttachment(audioProcessor.apvts, "highCutFreq", highCutFreqSlider)
```

Designing the GUI

To be able to successfully show each slider in a specific place in my GUI, I have to **set bounds** for each individual one. Then, when the addAndMakeVisible() function is called for each slider, it will correctly centre them in the bounds I create for each one.

Firstly, I designed the basic layout for what my GUI boundaries will be. In each boundary, I will display a different slider. Every boundary is represented as a rectangle.



In order to create these rectangles for each slider, I first have to set the size of the rectangle:

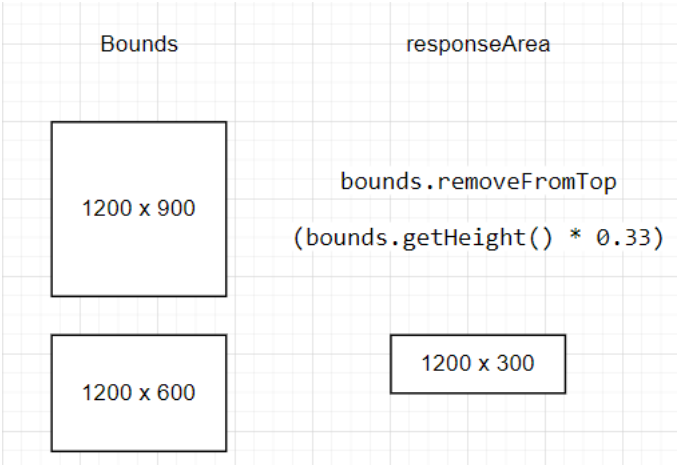
```
setSize(1200, 900);
```

These numbers can be changed, but it must always be called in the constructor method. This is because this is where an instance of the class is created, and so where all the sliders, as well as the window itself, will be created and made visible.

```
auto bounds = getLocalBounds();
auto responseArea = bounds.removeFromTop(bounds.getHeight() * 0.33);
auto freqAreaBounds = bounds.removeFromTop(bounds.getHeight() * 0.5);
```

The function getLocalBounds() simply returns the size of the rectangle that will hold my GUI: 1200x900. Then, I begin to cut rectangles out of this rectangle and allocate the spaces to variables I have made for each sliders' bounds.

Every time a rectangle is removed from an area, that rectangle's size is returned and assigned to the variable that I create. Also, the original size also decreases by that amount. For example, when I take 33% of the bounds area and allocate it to responseArea, bounds now has an area of 66% of the height it previously did.



I want for each row of sliders to take up 33% space of the rectangle, so I first remove a bound of 33% from the top of the rectangle. I do this by using a pre built function removeFromTop(), and remove a rectangle with a height of 33% the GUI window.

removeFromTop() and removeFromBottom() both assume that the width of the rectangle I want to cut will be the same as the width of the rectangle you are cutting from. romoveFromLeft() and removeFromRight() will assume that the rectangle I want to cut will have the same height as the rectangle I am cutting from.

Once I take the initial 33% strip from the top of the rectangle and assign it to repsonseArea, bounds now has a height of 600, which is 33% of 900. So, in order to take another 50% strip off, this time I multiply the height by 50% rather than 33%. This will result in whatever I assign the cut strip to having a height of 33%, and bounds also having a height of 33%.

```
auto lowCutArea = freqAreaBounds.removeFromLeft(bounds.getWidth() * 0.5);
auto highCutArea = freqAreaBounds;

auto lowCutFreqArea = lowCutArea.removeFromLeft(lowCutArea.getWidth() * 0.5);
auto lowCutSlopeArea = lowCutArea;

auto highCutFreqArea = highCutArea.removeFromLeft(highCutArea.getWidth() * 0.5);
auto highCutSlopeArea = highCutArea;
```

For the low and high cut sliders, I divided the frequency area by two. This left me with two equally sized rectangles in the middle strip. I then divided each one of those rectangles into two more equal parts, and then assigned each one to a slope or a cut.

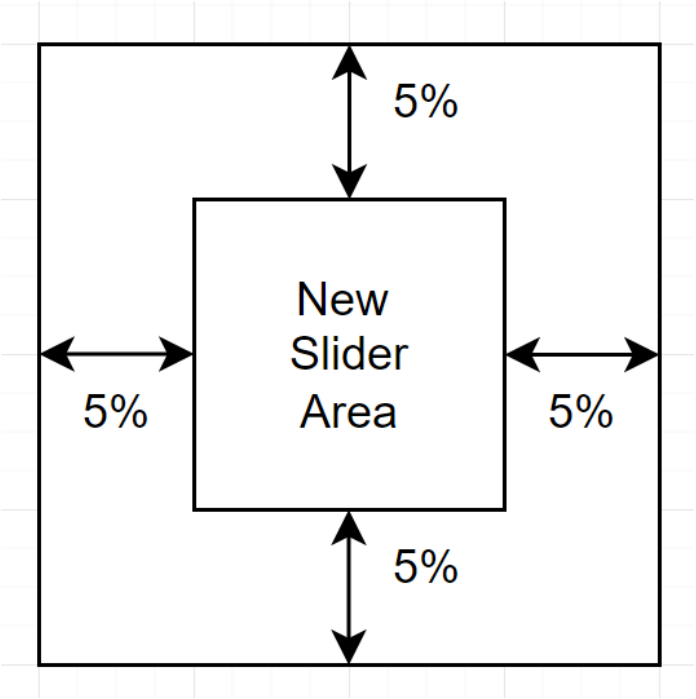
```
auto peakQualityArea = bounds.removeFromLeft(bounds.getWidth() * 0.33);
auto peakFreqArea = bounds.removeFromRight(bounds.getWidth() * 0.5);
auto peakGainArea = bounds;
```

Using the same division principle that I did for the rows, I created 3 columns in the last strip that was left for the peak Frequency, Gain, and Quality.

Finally, I assigned the bounds to their respective sliders.

```
lowCutFreqSlider.setBounds(lowCutFreqArea);
highCutFreqSlider.setBounds(highCutFreqArea);

lowCutSlopeSlider.setBounds(lowCutSlopeArea );
highCutSlopeSlider.setBounds(highCutSlopeArea);

peakFreqSlider.setBounds(peakFreqArea);
peakGainSlider.setBounds(peakGainArea);
peakQualitySlider.setBounds(peakQualityArea);
```

Decreasing the area size



After retrieving these initial rectangles for each slider, I wanted to make them smaller so it could have a more professional look. However, there is no built-in function that allows a rectangle to be multiplied by a scale factor. Instead, I removed strips from each side of the rectangle to make it smaller.
I used the same multiples and method for each rectangle, decreasing them on all 4 sides by 5%.
example:peakGainArea

```
auto peakGainArea = bounds;
auto peakGainArea1 = peakGainArea.removeFromLeft(peakGainArea.getWidth() * 0.95);
auto peakGainArea2 = peakGainArea1.removeFromRight(peakGainArea1.getWidth() * 0.9025);
auto peakGainArea3 = peakGainArea2.removeFromTop(peakGainArea2.getWidth() * 0.95);
auto peakGainArea4 = (peakGainArea3.removeFromBottom(peakGainArea3.getWidth() * 0.9025));
```

I used this area to print each slider's name in their centre.

```
g.drawFittedText("Low Cut Frequency",lowCutFreqArea4,
juce::Justification::centred, 1);
g.drawFittedText("High Cut Frequency", highCutFreqArea4,
juce::Justification::centred, 1);
g.drawFittedText("Low Cut Slope", lowCutSlopeArea4,
juce::Justification::centred, 1);
g.drawFittedText("High Cut Slope", highCutSlopeArea4,
juce::Justification::centred, 1);
g.drawFittedText("Peak Frequency", peakFreqArea4,
juce::Justification::centred, 1);
g.drawFittedText("Peak Gain", peakGainArea4,
juce::Justification::centred, 1);
g.drawFittedText("Peak Quality", peakQualityArea4,
juce::Justification::centred, 1);
```

Parameters: String,Rectangle,Justification,maxNumOfLines

Juce::Justification is a type of representation used when positioning graphical items.

Technical Solution: Parametric EQ

PluginProcessor.cpp

```
/*

==============================================================================
    This file contains the basic framework code for a JUCE plugin processor.

==============================================================================
*/
#include "PluginProcessor.h"
#include "PluginEditor.h"
//==============================================================================
EQAudioProcessor::EQAudioProcessor()
#ifndef JucePlugin_PreferredChannelConfigurations
    : AudioProcessor(BusesProperties()
#if ! JucePlugin_IsMidiEffect
#if ! JucePlugin_IsSynth
        .withInput("Input", juce::AudioChannelSet::stereo(), true)
#endif
        .withOutput("Output", juce::AudioChannelSet::stereo(), true)
#endif
    )
#endif
{
}

EQAudioProcessor::~EQAudioProcessor()
{
}
//==============================================================================
const juce::String EQAudioProcessor::getName() const
```

```
{
    return JucePlugin_Name;
}
bool EQAudioProcessor::acceptsMidi() const
{
#if JucePlugin_WantsMidiInput
    return true;
#else
    return false;
#endif
}
bool EQAudioProcessor::producesMidi() const
{
#if JucePlugin_ProducesMidiOutput
    return true;
#else
    return false;
#endif
}
bool EQAudioProcessor::isMidiEffect() const
{
#if JucePlugin_IsMidiEffect
    return true;
#else
    return false;
#endif
}
double EQAudioProcessor::getTailLengthSeconds() const
{
    return 0.0;
}
int EQAudioProcessor::getNumPrograms()
{
    return 1;   // NB: some hosts don't cope very well if you
tell them there are 0 programs,
                // so this should be at least 1, even if
you're not really implementing programs.
}
int EQAudioProcessor::getCurrentProgram()
{
    return 0;
}
void EQAudioProcessor::setCurrentProgram(int index)
{
```

```
}
const juce::String EQAudioProcessor::getProgramName(int index)
{
    return {};
}
void EQAudioProcessor::changeProgramName(int index, const
juce::String& newName)
{
}
//==============================================================
==================
void EQAudioProcessor::prepareToPlay(double sampleRate, int
samplesPerBlock)
{
    // pass a process spec object to both the left and right
chain, this will be passed to each link in the chain
    // this prepares each chain to be prepared for processing

    juce::dsp::ProcessSpec shem;

    shem.maximumBlockSize = samplesPerBlock;

    shem.numChannels = 1;

    shem.sampleRate = sampleRate;

    leftChain.prepare(shem);
    rightChain.prepare(shem);

    auto getchainSettings = getchainsettings(apvts);

    // IIR or Infinite-Duration Impulse Response Filters uses
a feedback mechanism where the previous output,
    //in conjunction with the present and past input,
    //is given as the present input.

    //coefficients for the peak filter:

    auto peakCoefficients =
juce::dsp::IIR::Coefficients<float>::makePeakFilter(sampleRate
, getchainSettings.peakFreq, getchainSettings.peakQuality,
juce::Decibels::decibelsToGain(getchainSettings.peakGain));

    //set the filter's coefficients accordingly:
```

```
    //IIR functions return instances on the heap, rather than
in the actual audio buffer.
    // To access these, you must dereference them

    *leftChain.get<Peak>().coefficients = *peakCoefficients;
//the star is used to deref.
    *rightChain.get<Peak>().coefficients = *peakCoefficients;


    // every 2 orders creates an IIRHighPass / LowCut filter
    // so for x iterations of the filter we need 2x filters
    //as the first slope choice starts from 0, we need 2x+1
filters

    auto lowCutCoefficients =
juce::dsp::FilterDesign<float>::designIIRHighpassHighOrderButt
erworthMethod(getchainSettings.lowCutFreq, sampleRate,
(getchainSettings.lowCutSlope + 1) * 2);

    auto& RightLowCut = rightChain.get<LowCut>();
    auto& leftLowCut = leftChain.get<LowCut>();

    RightLowCut.setBypassed<0>(true);
    RightLowCut.setBypassed<1>(true);
    RightLowCut.setBypassed<2>(true);
    RightLowCut.setBypassed<3>(true);

    leftLowCut.setBypassed<0>(true);
    leftLowCut.setBypassed<1>(true);
    leftLowCut.setBypassed<2>(true);
    leftLowCut.setBypassed<3>(true);

    switch (getchainSettings.lowCutSlope)
    {
    case Slope4:
    {
        *RightLowCut.get<3>().coefficients =
*lowCutCoefficients[3];
        RightLowCut.setBypassed<3>(false);
        *leftLowCut.get<3>().coefficients =
*lowCutCoefficients[3];
        leftLowCut.setBypassed<3>(false);
        [[fallthrough]]; // fallthrough is explicit
    }
```

```
    case Slope3:
    {
        *RightLowCut.get<2>().coefficients =
*lowCutCoefficients[2];
        RightLowCut.setBypassed<2>(false);
        *leftLowCut.get<2>().coefficients =
*lowCutCoefficients[2];
        leftLowCut.setBypassed<2>(false);
        [[fallthrough]]; // fallthrough is explicit
    }
    case Slope2:
    {
        *RightLowCut.get<1>().coefficients =
*lowCutCoefficients[1];
        RightLowCut.setBypassed<1>(false);
        *leftLowCut.get<1>().coefficients =
*lowCutCoefficients[1];
        leftLowCut.setBypassed<1>(false);
        [[fallthrough]]; // fallthrough is explicit
    }
    case Slope1:
    {
        *RightLowCut.get<0>().coefficients =
*lowCutCoefficients[0];
        RightLowCut.setBypassed<0>(false);
        *leftLowCut.get<0>().coefficients =
*lowCutCoefficients[0];
        leftLowCut.setBypassed<0>(false);
        [[fallthrough]]; // fallthrough is explicit
    }
    }

    auto highCutCoefficients =
juce::dsp::FilterDesign<float>::designIIRLowpassHighOrderButte
rworthMethod(getchainSettings.highCutFreq, sampleRate,
(getchainSettings.highCutSlope + 1) * 2);

    auto& RightHighCut = rightChain.get<HighCut>();
    auto& LeftHighCut = leftChain.get<HighCut>();

    RightHighCut.setBypassed<0>(true);
    RightHighCut.setBypassed<1>(true);
    RightHighCut.setBypassed<2>(true);
```

```
    RightHighCut.setBypassed<3>(true);

    LeftHighCut.setBypassed<0>(true);
    LeftHighCut.setBypassed<1>(true);
    LeftHighCut.setBypassed<2>(true);
    RightHighCut.setBypassed<3>(true);

    switch (getchainSettings.highCutSlope)
    {
    case Slope4:
    {

        *RightHighCut.get<3>().coefficients =
*highCutCoefficients[3];
        RightHighCut.setBypassed<3>(false);
        *LeftHighCut.get<3>().coefficients =
*highCutCoefficients[3];
        LeftHighCut.setBypassed<3>(false);
        //[[fallthrough]];
    }

    case Slope3:
    {
        *RightHighCut.get<2>().coefficients =
*highCutCoefficients[2];
        RightHighCut.setBypassed<2>(false);
        *LeftHighCut.get<2>().coefficients =
*highCutCoefficients[2];
        LeftHighCut.setBypassed<2>(false);
        //[[fallthrough]];
    }
    case Slope2:
    {
        *RightHighCut.get<1>().coefficients =
*highCutCoefficients[1];
        RightHighCut.setBypassed<1>(false);
        *LeftHighCut.get<1>().coefficients =
*highCutCoefficients[1];
        LeftHighCut.setBypassed<1>(false);
        //[[fallthrough]];
    }
    case Slope1:
    {
```

```
        *RightHighCut.get<0>().coefficients =
*highCutCoefficients[0];
        RightHighCut.setBypassed<0>(false);
        *LeftHighCut.get<0>().coefficients =
*highCutCoefficients[0];
        LeftHighCut.setBypassed<0>(false);
        //[[fallthrough]];
    }
    }
}

void EQAudioProcessor::releaseResources()
{
    // When playback stops,B you can use this as an
opportunity to free up any
    // spare memory, etc.
}
#ifndef JucePlugin_PreferredChannelConfigurations
bool EQAudioProcessor::isBusesLayoutSupported(const
BusesLayout& layouts) const
{
#if JucePlugin_IsMidiEffect
    juce::ignoreUnused(layouts);
    return true;
#else
    // This is the place where you check if the layout is
supported.
    // In this template code we only support mono or stereo.
    // Some plugin hosts, such as certain GarageBand versions,
will only
    // load plugins that support stereo bus layouts.
    if (layouts.getMainOutputChannelSet() !=
juce::AudioChannelSet::mono()
        && layouts.getMainOutputChannelSet() !=
juce::AudioChannelSet::stereo())
        return false;
    // This checks if the input layout matches the output
layout
#if ! JucePlugin_IsSynth
    if (layouts.getMainOutputChannelSet() !=
layouts.getMainInputChannelSet())
        return false;
#endif
    return true;
```

```
#endif
}
#endif
void EQAudioProcessor::processBlock(juce::AudioBuffer<float>&
buffer, juce::MidiBuffer& midiMessages)
{
    juce::ScopedNoDenormals noDenormals;
    auto totalNumInputChannels = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();
    // In case we have more outputs than inputs, this code
clears any output
    // channels that didn't contain input data, (because these
aren't
    // guaranteed to be empty - they may contain garbage).
    // This is here to avoid people getting screaming feedback
    // when they first compile a plugin, but obviously you
don't need to keep
    // this code if your algorithm always overwrites all the
output channels.
    for (auto i = totalNumInputChannels; i <
totalNumOutputChannels; ++i)
        buffer.clear(i, 0, buffer.getNumSamples());

    // this code must be implemented in processBlock AS WELL
as prepareToPlay
    // so that the filter can be updated with the new
coefficients when the user changes
    // the desired frequency.

    auto getchainSettings = getchainsettings(apvts);

    auto peakCoefficients =
juce::dsp::IIR::Coefficients<float>::makePeakFilter(getSampleR
ate(), getchainSettings.peakFreq,
getchainSettings.peakQuality,
juce::Decibels::decibelsToGain(getchainSettings.peakGain));

    //A Processerchain needs processing context to be passed
through it
    //in order to run audio via the links in the chain.
    //Processing contexts needs to be supplied by Audioblock
instances.
```

```
    *leftChain.get<Peak>().coefficients = *peakCoefficients;
//as these are references, they point to the addresses of
variables.
    *rightChain.get<Peak>().coefficients =
*peakCoefficients;// to get the data in these variables, we
must dereference them.

    auto lowCutCoefficients =
juce::dsp::FilterDesign<float>::designIIRHighpassHighOrderButt
erworthMethod(getchainSettings.lowCutFreq, getSampleRate(),
(getchainSettings.lowCutSlope + 1) * 2);

    auto& RightLowCut = rightChain.get<LowCut>();

    RightLowCut.setBypassed<0>(true);
    RightLowCut.setBypassed<1>(true);
    RightLowCut.setBypassed<2>(true);
    RightLowCut.setBypassed<3>(true);

    auto& leftLowCut = leftChain.get<LowCut>();

    leftLowCut.setBypassed<0>(true);
    leftLowCut.setBypassed<1>(true);
    leftLowCut.setBypassed<2>(true);
    leftLowCut.setBypassed<3>(true);
    switch (getchainSettings.lowCutSlope)
    {
    case Slope4:
    {
        *RightLowCut.get<3>().coefficients =
*lowCutCoefficients[3];
        RightLowCut.setBypassed<3>(false);
        *leftLowCut.get<3>().coefficients =
*lowCutCoefficients[3];
        leftLowCut.setBypassed<3>(false);
        //[[fallthrough]];
    }

    case Slope3:
    {
        *RightLowCut.get<2>().coefficients =
*lowCutCoefficients[2];
        RightLowCut.setBypassed<2>(false);
```

```
        *leftLowCut.get<2>().coefficients =
*lowCutCoefficients[2];
        leftLowCut.setBypassed<2>(false);
        //[[fallthrough]];
    }
    case Slope2:
    {
        *RightLowCut.get<1>().coefficients =
*lowCutCoefficients[1];
        RightLowCut.setBypassed<1>(false);
        *leftLowCut.get<1>().coefficients =
*lowCutCoefficients[1];
        leftLowCut.setBypassed<1>(false);
        //[[fallthrough]];
    }
    case Slope1:
    {
        *RightLowCut.get<0>().coefficients =
*lowCutCoefficients[0];
        RightLowCut.setBypassed<0>(false);
        *leftLowCut.get<0>().coefficients =
*lowCutCoefficients[0];
        leftLowCut.setBypassed<0>(false);
        //[[fallthrough]];
    }
    }

    auto highCutCoefficients =
juce::dsp::FilterDesign<float>::designIIRLowpassHighOrderButte
rworthMethod(getchainSettings.highCutFreq, getSampleRate(),
(getchainSettings.highCutSlope + 1) * 2);

    auto& RightHighCut = rightChain.get<HighCut>();
    auto& LeftHighCut = leftChain.get<HighCut>();

    RightHighCut.setBypassed<0>(true);
    RightHighCut.setBypassed<1>(true);
    RightHighCut.setBypassed<2>(true);
    RightHighCut.setBypassed<3>(true);

    LeftHighCut.setBypassed<0>(true);
    LeftHighCut.setBypassed<1>(true);
    LeftHighCut.setBypassed<2>(true);
    RightHighCut.setBypassed<3>(true);
```

```
    switch (getchainSettings.highCutSlope)
    {
    case Slope4:
    {
        *RightHighCut.get<3>().coefficients =
*highCutCoefficients[3];
        RightHighCut.setBypassed<3>(false);
        *LeftHighCut.get<3>().coefficients =
*highCutCoefficients[3];
        LeftHighCut.setBypassed<3>(false);
        //[[fallthrough]];
    }

    case Slope3:
    {
        *RightHighCut.get<2>().coefficients =
*highCutCoefficients[2];
        RightHighCut.setBypassed<2>(false);
        *LeftHighCut.get<2>().coefficients =
*highCutCoefficients[2];
        LeftHighCut.setBypassed<2>(false);
        //[[fallthrough]];
    }
    case Slope2:
    {
        *RightHighCut.get<1>().coefficients =
*highCutCoefficients[1];
        RightHighCut.setBypassed<1>(false);
        *LeftHighCut.get<1>().coefficients =
*highCutCoefficients[1];
        LeftHighCut.setBypassed<1>(false);
        //[[fallthrough]];
    }
    case Slope1:
    {
        *RightHighCut.get<0>().coefficients =
*highCutCoefficients[0];
        RightHighCut.setBypassed<0>(false);
        *LeftHighCut.get<0>().coefficients =
*highCutCoefficients[0];
        LeftHighCut.setBypassed<0>(false);
        //[[fallthrough]];
    }
```

```
    }

    juce::dsp::AudioBlock<float> block(buffer);

    auto leftBlock = block.getSingleChannelBlock(0);
    auto rightBlock = block.getSingleChannelBlock(1);

    juce::dsp::ProcessContextReplacing<float>
leftContextReplacing(leftBlock);
    juce::dsp::ProcessContextReplacing<float>
rightContextReplacing(rightBlock);

    leftChain.process(leftContextReplacing);
    rightChain.process(rightContextReplacing);

}
//==============================================================
==================
bool EQAudioProcessor::hasEditor() const
{
    return true; // (change this to false if you choose to not
supply an editor)
}
juce::AudioProcessorEditor* EQAudioProcessor::createEditor()
{
    return new EQAudioProcessorEditor(*this);
    //return new juce::GenericAudioProcessorEditor(*this);
}
//==============================================================
==================
void EQAudioProcessor::getStateInformation(juce::MemoryBlock&
destData)
{
    // You should use this method to store your parameters in
the memory block.
    // You could do that either as raw data, or use the XML or
ValueTree classes
    // as intermediaries to make it easy to save and load
complex data.
}
void EQAudioProcessor::setStateInformation(const void* data,
int sizeInBytes)
{
```

```
    // You should use this method to restore your parameters
from this memory block,
    // whose contents will have been created by the
getStateInformation() call.
}

chainsettings
getchainsettings(juce::AudioProcessorValueTreeState& apvts)
{
    chainsettings settings; // denormalises the values of each
parameter so that we can get the real-world values

    settings.lowCutFreq =
apvts.getRawParameterValue("lowcutFreq")->load();
    settings.highCutFreq =
apvts.getRawParameterValue("highcutFreq")->load();
    settings.peakFreq =
apvts.getRawParameterValue("PeakFreq")->load();
    settings.peakGain =
apvts.getRawParameterValue("peakGain")->load();
    settings.peakQuality =
apvts.getRawParameterValue("peakQuality")->load();
    settings.lowCutSlope =
static_cast<Slope>(apvts.getRawParameterValue("lowcutSlope")->
load());
    settings.highCutSlope = static_cast
<Slope>(apvts.getRawParameterValue("highcutSlope")->load());

    return settings;
}

juce::AudioProcessorValueTreeState::ParameterLayout
EQAudioProcessor::createParameterLayout()
{
    juce::AudioProcessorValueTreeState::ParameterLayout
layout;

    //RangeStart,RangeEnd,Interval, SkewFactor

    //LowCut Frequency


layout.add(std::make_unique<juce::AudioParameterFloat>("lowcut
```

```
Freq", "LowCut Freq", juce::NormalisableRange<float>(20.f,
20000.f, 1.f, 0.25f), 20.f));

    //HighCut Frequency


layout.add(std::make_unique<juce::AudioParameterFloat>("highcu
tFreq", "HighCut Freq", juce::NormalisableRange<float>(20.f,
20000.f, 1.f, 0.25f), 20000.f));

    //Parametric Peak


layout.add(std::make_unique<juce::AudioParameterFloat>("PeakFr
eq", "Peak Freq", juce::NormalisableRange<float>(20.f,
20000.f, 1.f, 0.25f), 750.f));

    //Peak Gain


layout.add(std::make_unique<juce::AudioParameterFloat>("peakGa
in", "Peak Gain", juce::NormalisableRange<float>(-24.f, 24.f,
0.5f, 1.f), 0.0f));

    //Peak Quality (How wide / narrow the peak is)


layout.add(std::make_unique<juce::AudioParameterFloat>("peakQu
ality", "Peak Quality", juce::NormalisableRange<float>(0.1f,
10.f, 0.05f, 1.f), 1.f));

    // stringArray dropdown table

    juce::StringArray stringArray;
    for (int i = 0; i < 4; ++i) // I IN RANGE 1 TO 4, INTERVAL
OF 1
    {
        juce::String str; str << (12 + i * 12); str << "
dB/Oct"; stringArray.add(str);
    }


layout.add(std::make_unique<juce::AudioParameterChoice>("lowcu
tSlope", "LowCut Slope", stringArray, 0));
```

```
layout.add(std::make_unique<juce::AudioParameterChoice>("highc
utSlope", "HighCut Slope", stringArray, 0));

    return layout;
}
//==============================================================
==================
// This creates new instances of the plugin..
juce::AudioProcessor* JUCE_CALLTYPE createPluginFilter()
{
    return new EQAudioProcessor();
}
```

PluginProcessor.h

```
/*

==============================================================================

    This file contains the basic framework code for a JUCE plugin processor.

==============================================================================

*/

#pragma once


#include <JuceHeader.h>


const std::integral_constant<int, 0> LowCut;

const std::integral_constant<int, 1> Peak;

const std::integral_constant<int, 2> HighCut;


enum Slope

{

    Slope1,

    Slope2,

    Slope3,

    Slope4

};
```

```
//creating a structure so that the apvts can pull these values
every time it is called, rather than having to write them out
over and over.
struct chainsettings
{
    float peakFreq{ 0 }, peakGain{ 0 }, peakQuality{ 1.f };
    float lowCutFreq{ 0 }, highCutFreq{ 0 };
    Slope lowCutSlope{ Slope::Slope1 }, highCutSlope{
Slope::Slope1 };
};

chainsettings
getchainsettings(juce::AudioProcessorValueTreeState& apvts);



//==============================================================
==================
/**
*/
class EQAudioProcessor : public juce::AudioProcessor
{
public:
//==============================================================
==================
    EQAudioProcessor();
    ~EQAudioProcessor() override;
//==============================================================
==================
```

```
    void prepareToPlay(double sampleRate, int samplesPerBlock)
override;

    void releaseResources() override;

#ifndef JucePlugin_PreferredChannelConfigurations

    bool isBusesLayoutSupported(const BusesLayout& layouts)
const override;

#endif

    void processBlock(juce::AudioBuffer<float>&,
juce::MidiBuffer&) override;


//==============================================================
==================

    juce::AudioProcessorEditor* createEditor() override;

    bool hasEditor() const override;


//==============================================================
==================

    const juce::String getName() const override;

    bool acceptsMidi() const override;

    bool producesMidi() const override;

    bool isMidiEffect() const override;

    double getTailLengthSeconds() const override;


//==============================================================
==================

    int getNumPrograms() override;

    int getCurrentProgram() override;

    void setCurrentProgram(int index) override;

    const juce::String getProgramName(int index) override;

    void changeProgramName(int index, const juce::String&
newName) override;
```

```
//==============================================================
==================

    void getStateInformation(juce::MemoryBlock& destData)
override;

    void setStateInformation(const void* data, int
sizeInBytes) override;

    static juce::AudioProcessorValueTreeState::ParameterLayout
createParameterLayout();

    juce::AudioProcessorValueTreeState apvts{ *this, nullptr,
"Parameters", createParameterLayout() };


    //slope of cut filters are multiples of 12dB/Oct and
filters defaults at 12dB/Oct, but we want up to 48 dB/Oct


private:

    using Filter = juce::dsp::IIR::Filter<float>;


    using VariableCut = juce::dsp::ProcessorChain<Filter,
Filter, Filter, Filter>; // 12 dB/Oct * 4 = 48dB/Oct


    using SingleChain = juce::dsp::ProcessorChain<VariableCut,
Filter, VariableCut>;

    //dsp defaults as mono instead of stereo, so creating a
left and right channel

    //to play concurrently will result in stereo sound


    SingleChain leftChain, rightChain;
//==============================================================
==================


JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR(EQAudioProcessor)

}
```

PluginEditor.cpp

```
/*

==============================================================================
    This file contains the basic framework code for a JUCE plugin editor.

==============================================================================
*/

#include "PluginProcessor.h"
#include "PluginEditor.h"
#include<iostream>
#include<list>


//==============================================================================
EQAudioProcessorEditor::EQAudioProcessorEditor(EQAudioProcessor& p)
    : AudioProcessorEditor(&p), audioProcessor(p),
    peakFreqAttachment(audioProcessor.apvts, "PeakFreq", peakFreqSlider),
    peakGainAttachment(audioProcessor.apvts, "peakGain", peakGainSlider),
    peakQualityAttachment(audioProcessor.apvts, "peakQuality", peakQualitySlider),
    lowCutSlopeAttachment(audioProcessor.apvts, "lowcutSlope", lowCutSlopeSlider),
    highCutSlopeAttachment(audioProcessor.apvts, "highcutSlope", highCutSlopeSlider),
    lowCutFreqAttachment(audioProcessor.apvts, "lowcutFreq", lowCutFreqSlider),
    highCutFreqAttachment(audioProcessor.apvts, "highcutFreq", highCutFreqSlider)
{
    // Make sure that before the constructor has finished, you've set the
    // editor's size to whatever you need it to be.

    addAndMakeVisible(highCutFreqSlider);
```

```
    addAndMakeVisible(lowCutFreqSlider);
    addAndMakeVisible(peakQualitySlider);
    addAndMakeVisible(peakGainSlider);
    addAndMakeVisible(peakFreqSlider);
    addAndMakeVisible(lowCutSlopeSlider);
    addAndMakeVisible(highCutSlopeSlider);

    setSize(1200, 900);
}

EQAudioProcessorEditor::~EQAudioProcessorEditor()
{
}

//==============================================================
==================
void EQAudioProcessorEditor::paint(juce::Graphics& g)
{
    // (Our component is opaque, so we must completely fill
the background with a solid colour)

    auto bounds = getLocalBounds();
    auto responseArea =
bounds.removeFromTop(bounds.getHeight() * 0.33);
    auto freqAreaBounds =
bounds.removeFromTop(bounds.getHeight() * 0.5);

    auto lowCutArea =
freqAreaBounds.removeFromLeft(bounds.getWidth() * 0.5);
    auto highCutArea = freqAreaBounds;




    auto lowCutFreqArea =
lowCutArea.removeFromLeft(lowCutArea.getWidth() * 0.5);
    auto lowCutFreqArea1 =
lowCutFreqArea.removeFromLeft(lowCutFreqArea.getWidth() *
0.95);
    auto lowCutFreqArea2 =
lowCutFreqArea1.removeFromRight(lowCutFreqArea1.getWidth() *
0.9025);
```

```
    auto lowCutFreqArea3 =
lowCutFreqArea2.removeFromTop(lowCutFreqArea2.getWidth() *
0.95);
    auto lowCutFreqArea4 =
lowCutFreqArea3.removeFromBottom(lowCutFreqArea3.getWidth() *
0.9025);



    auto lowCutSlopeArea = lowCutArea;
    auto lowCutSlopeArea1 =
lowCutSlopeArea.removeFromLeft(lowCutSlopeArea.getWidth() *
0.95);
    auto lowCutSlopeArea2 =
lowCutSlopeArea1.removeFromRight(lowCutSlopeArea1.getWidth() *
0.9025);
    auto lowCutSlopeArea3 =
lowCutSlopeArea2.removeFromTop(lowCutSlopeArea2.getWidth() *
0.95);
    auto lowCutSlopeArea4 =
lowCutSlopeArea3.removeFromBottom(lowCutSlopeArea3.getWidth()
* 0.9025);



    auto highCutFreqArea =
highCutArea.removeFromLeft(highCutArea.getWidth() * 0.5);
    auto highCutFreqArea1 =
highCutFreqArea.removeFromLeft(highCutFreqArea.getWidth() *
0.95);
    auto highCutFreqArea2 =
highCutFreqArea1.removeFromRight(highCutFreqArea1.getWidth() *
0.9025);
    auto highCutFreqArea3 =
highCutFreqArea2.removeFromTop(highCutFreqArea2.getWidth() *
0.95);
    auto highCutFreqArea4 =
highCutFreqArea3.removeFromBottom(highCutFreqArea3.getWidth()
* 0.9025);

    auto highCutSlopeArea = highCutArea;
    auto highCutSlopeArea1 =
highCutSlopeArea.removeFromLeft(highCutSlopeArea.getWidth() *
0.95);
```

```
    auto highCutSlopeArea2 =
highCutSlopeArea1.removeFromRight(highCutSlopeArea1.getWidth()
* 0.9025);
    auto highCutSlopeArea3 =
highCutSlopeArea2.removeFromTop(highCutSlopeArea2.getWidth() *
0.95);
    auto highCutSlopeArea4 =
highCutSlopeArea3.removeFromBottom(highCutSlopeArea3.getWidth(
) * 0.9025);


    auto peakQualityArea =
bounds.removeFromLeft(bounds.getWidth() * 0.33);
    auto peakQualityArea1 =
peakQualityArea.removeFromLeft(peakQualityArea.getWidth() *
0.95);
    auto peakQualityArea2 =
peakQualityArea1.removeFromRight(peakQualityArea1.getWidth() *
0.9025);
    auto peakQualityArea3 =
peakQualityArea2.removeFromTop(peakQualityArea2.getWidth() *
0.95);
    auto peakQualityArea4 =
peakQualityArea3.removeFromBottom(peakQualityArea3.getWidth()
* 0.9025);


    auto peakFreqArea =
bounds.removeFromRight(bounds.getWidth() * 0.5);
    auto peakFreqArea1 =
peakFreqArea.removeFromLeft(peakFreqArea.getWidth() * 0.95);
    auto peakFreqArea2 =
peakFreqArea1.removeFromRight(peakFreqArea1.getWidth() *
0.9025);
    auto peakFreqArea3 =
peakFreqArea2.removeFromTop(peakFreqArea2.getWidth() * 0.95);
    auto peakFreqArea4 =
peakFreqArea3.removeFromBottom(peakFreqArea3.getWidth() *
0.9025);

    auto peakGainArea = bounds;
    auto peakGainArea1 =
peakGainArea.removeFromLeft(peakGainArea.getWidth() * 0.95);
```

```
    auto peakGainArea2 =
peakGainArea1.removeFromRight(peakGainArea1.getWidth() *
0.9025);
    auto peakGainArea3 =
peakGainArea2.removeFromTop(peakGainArea2.getWidth() * 0.95);
    auto peakGainArea4 =
(peakGainArea3.removeFromBottom(peakGainArea3.getWidth() *
0.9025));

    lowCutFreqSlider.setBounds(lowCutFreqArea4);
    highCutFreqSlider.setBounds(highCutFreqArea4);

    lowCutSlopeSlider.setBounds(lowCutSlopeArea4);
    highCutSlopeSlider.setBounds(highCutSlopeArea4);

    peakFreqSlider.setBounds(peakFreqArea4);
    peakGainSlider.setBounds(peakGainArea4);
    peakQualitySlider.setBounds(peakQualityArea4);
    g.fillAll(juce::Colours::black);
    g.setColour(juce::Colours::white);
    g.setFont(15.0f);
    g.drawFittedText("Low Cut Frequency",lowCutFreqArea4,
juce::Justification::centred, 1);
    g.drawFittedText("High Cut Frequency", highCutFreqArea4,
juce::Justification::centred, 1);
    g.drawFittedText("Low Cut Slope", lowCutSlopeArea4,
juce::Justification::centred, 1);
    g.drawFittedText("High Cut Slope", highCutSlopeArea4,
juce::Justification::centred, 1);
    g.drawFittedText("Peak Frequency", peakFreqArea4,
juce::Justification::centred, 1);
    g.drawFittedText("Peak Gain", peakGainArea4,
juce::Justification::centred, 1);
    g.drawFittedText("Peak Quality", peakQualityArea4,
juce::Justification::centred, 1);
    g.setFont(45.0f);
    g.setColour(juce::Colours::lightblue);
    g.drawFittedText("3 Band Parametric EQ", responseArea,
juce::Justification::centred, 1);

}

void EQAudioProcessorEditor::resized()
{
```

```
    // This is generally where you'll want to lay out the
positions of any
    // subcomponents in your editor..

    auto bounds = getLocalBounds();
    auto responseArea =
bounds.removeFromTop(bounds.getHeight() * 0.33);
    auto freqAreaBounds =
bounds.removeFromTop(bounds.getHeight() * 0.5);

    auto lowCutArea =
freqAreaBounds.removeFromLeft(bounds.getWidth() * 0.5);
    auto highCutArea = freqAreaBounds;




    auto lowCutFreqArea =
lowCutArea.removeFromLeft(lowCutArea.getWidth() * 0.5);
    auto lowCutFreqArea1 =
lowCutFreqArea.removeFromLeft(lowCutFreqArea.getWidth() *
0.95);
    auto lowCutFreqArea2 =
lowCutFreqArea1.removeFromRight(lowCutFreqArea1.getWidth() *
0.9025);
    auto lowCutFreqArea3 =
lowCutFreqArea2.removeFromTop(lowCutFreqArea2.getWidth() *
0.95);
    auto lowCutFreqArea4 =
lowCutFreqArea3.removeFromBottom(lowCutFreqArea3.getWidth() *
0.9025);



    auto lowCutSlopeArea = lowCutArea;
    auto lowCutSlopeArea1 =
lowCutSlopeArea.removeFromLeft(lowCutSlopeArea.getWidth() *
0.95);
    auto lowCutSlopeArea2 =
lowCutSlopeArea1.removeFromRight(lowCutSlopeArea1.getWidth() *
0.9025);
    auto lowCutSlopeArea3 =
lowCutSlopeArea2.removeFromTop(lowCutSlopeArea2.getWidth() *
0.95);
```

```
    auto lowCutSlopeArea4 =
lowCutSlopeArea3.removeFromBottom(lowCutSlopeArea3.getWidth()
* 0.9025);


    auto highCutFreqArea =
highCutArea.removeFromLeft(highCutArea.getWidth() * 0.5);
    auto highCutFreqArea1 =
highCutFreqArea.removeFromLeft(highCutFreqArea.getWidth() *
0.95);
    auto highCutFreqArea2 =
highCutFreqArea1.removeFromRight(highCutFreqArea1.getWidth() *
0.9025);
    auto highCutFreqArea3 =
highCutFreqArea2.removeFromTop(highCutFreqArea2.getWidth() *
0.95);
    auto highCutFreqArea4 =
highCutFreqArea3.removeFromBottom(highCutFreqArea3.getWidth()
* 0.9025);

    auto highCutSlopeArea = highCutArea;
    auto highCutSlopeArea1 =
highCutSlopeArea.removeFromLeft(highCutSlopeArea.getWidth() *
0.95);
    auto highCutSlopeArea2 =
highCutSlopeArea1.removeFromRight(highCutSlopeArea1.getWidth()
* 0.9025);
    auto highCutSlopeArea3 =
highCutSlopeArea2.removeFromTop(highCutSlopeArea2.getWidth() *
0.95);
    auto highCutSlopeArea4 =
highCutSlopeArea3.removeFromBottom(highCutSlopeArea3.getWidth(
) * 0.9025);


    auto peakQualityArea =
bounds.removeFromLeft(bounds.getWidth() * 0.33);
    auto peakQualityArea1 =
peakQualityArea.removeFromLeft(peakQualityArea.getWidth() *
0.95);
    auto peakQualityArea2 =
peakQualityArea1.removeFromRight(peakQualityArea1.getWidth() *
0.9025);
```

```
    auto peakQualityArea3 =
peakQualityArea2.removeFromTop(peakQualityArea2.getWidth() *
0.95);
    auto peakQualityArea4 =
peakQualityArea3.removeFromBottom(peakQualityArea3.getWidth()
* 0.9025);


    auto peakFreqArea =
bounds.removeFromRight(bounds.getWidth() * 0.5);
    auto peakFreqArea1 =
peakFreqArea.removeFromLeft(peakFreqArea.getWidth() * 0.95);
    auto peakFreqArea2 =
peakFreqArea1.removeFromRight(peakFreqArea1.getWidth() *
0.9025);
    auto peakFreqArea3 =
peakFreqArea2.removeFromTop(peakFreqArea2.getWidth() * 0.95);
    auto peakFreqArea4 =
peakFreqArea3.removeFromBottom(peakFreqArea3.getWidth() *
0.9025);

    auto peakGainArea = bounds;
    auto peakGainArea1 =
peakGainArea.removeFromLeft(peakGainArea.getWidth() * 0.95);
    auto peakGainArea2 =
peakGainArea1.removeFromRight(peakGainArea1.getWidth() *
0.9025);
    auto peakGainArea3 =
peakGainArea2.removeFromTop(peakGainArea2.getWidth() * 0.95);
    auto peakGainArea4 =
(peakGainArea3.removeFromBottom(peakGainArea3.getWidth() *
0.9025));

    lowCutFreqSlider.setBounds(lowCutFreqArea4);
    highCutFreqSlider.setBounds(highCutFreqArea4);

    lowCutSlopeSlider.setBounds(lowCutSlopeArea4);
    highCutSlopeSlider.setBounds(highCutSlopeArea4);

    peakFreqSlider.setBounds(peakFreqArea4);
    peakGainSlider.setBounds(peakGainArea4);
    peakQualitySlider.setBounds(peakQualityArea4);

}
```

PluginEditor.h

```
/*

==============================================================
================
    This file contains the basic framework code for a JUCE
plugin editor.

==============================================================
================
*/

#pragma once

#include <JuceHeader.h>
#include "PluginProcessor.h"

struct RotarySlider : juce::Slider
{
    RotarySlider() :
juce::Slider(juce::Slider::SliderStyle::RotaryHorizontalVertic
alDrag, juce::Slider::TextEntryBoxPosition::TextBoxAbove)
    {
        bool stopAtEnd = true;
    }
};


//==============================================================
==================
/**
*/
class EQAudioProcessorEditor : public
juce::AudioProcessorEditor
{
public:
    EQAudioProcessorEditor(EQAudioProcessor&);
    ~EQAudioProcessorEditor() override;
```

```
//==============================================================
==================
    void paint(juce::Graphics&) override;
    void resized() override;



private:
    // This reference is provided as a quick way for your
editor to
    // access the processor object that created it.
    EQAudioProcessor& audioProcessor;

    RotarySlider peakFreqSlider;
    RotarySlider peakGainSlider;
    RotarySlider peakQualitySlider;
    RotarySlider lowCutFreqSlider;
    RotarySlider highCutFreqSlider;
    RotarySlider lowCutSlopeSlider;
    RotarySlider highCutSlopeSlider;

    juce::AudioProcessorValueTreeState::SliderAttachment
highCutSlopeAttachment;
    juce::AudioProcessorValueTreeState::SliderAttachment
lowCutSlopeAttachment;
    juce::AudioProcessorValueTreeState::SliderAttachment
peakFreqAttachment;
    juce::AudioProcessorValueTreeState::SliderAttachment
peakGainAttachment;
    juce::AudioProcessorValueTreeState::SliderAttachment
peakQualityAttachment;
    juce::AudioProcessorValueTreeState::SliderAttachment
lowCutFreqAttachment;
    juce::AudioProcessorValueTreeState::SliderAttachment
highCutFreqAttachment;


JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR(EQAudioProcessorE
ditor)
};
```

Testing:

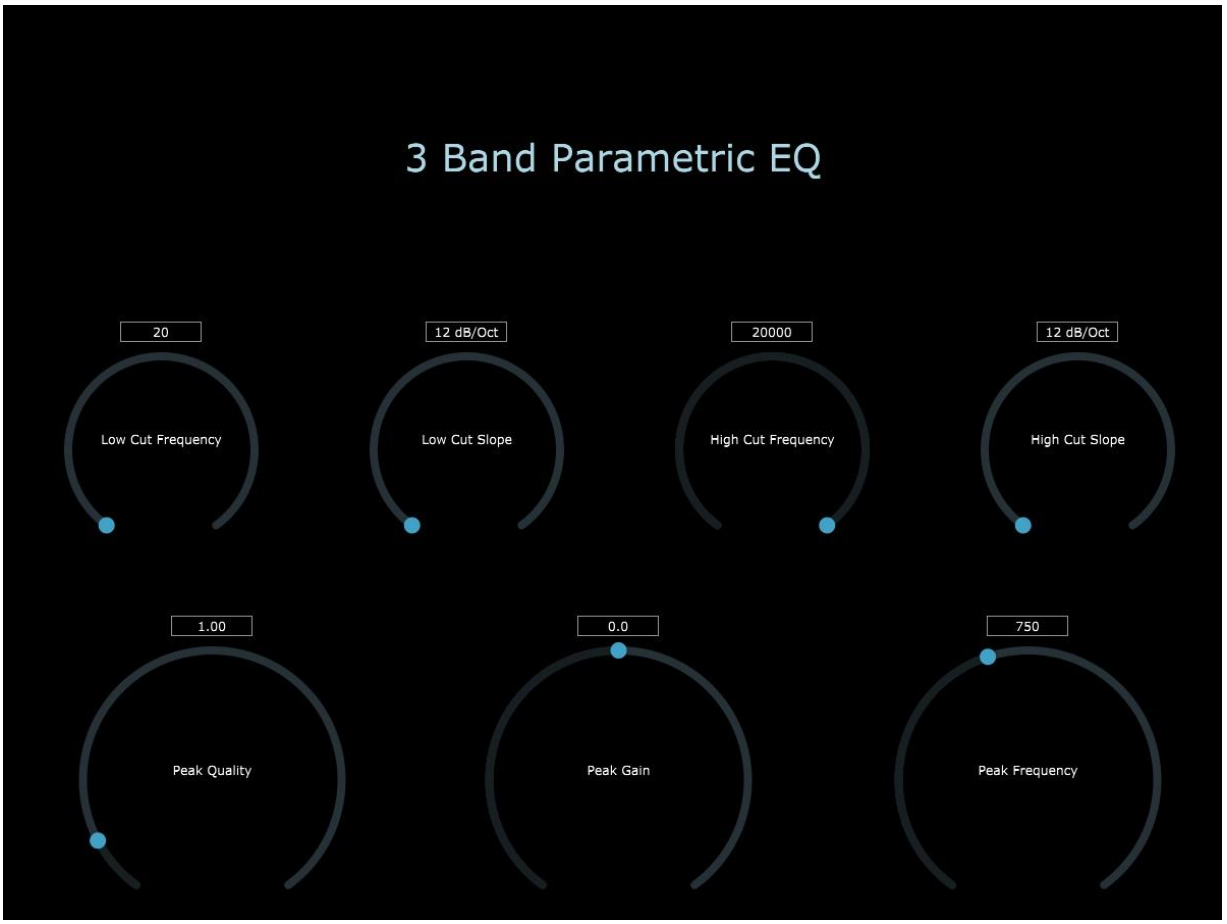
Reference Number	Description	Expected Result	Pass/Fail	Reference
1.1.1	EQ can take sound by sample.	If any sliders are moved, some filtering should be heard.	PASS	Video 1
1.1.2	EQ can take sound by MIDI controller.	If any sliders are moved, some filtering should be heard.	PASS	Video 1
1.2.1.1	The chain of filters should contain a low cut filter.	The low cut slope should cut off lower frequencies.	PASS	Video 1
1.2.1.2	The chain of filters should contain a peak frequency filter.	The peak frequency should be heard boosting/ mellowing certain frequencies.	PASS	Video 1
1.2.1.3	The chain of filters should contain a high cut filter.	The High Cut Slope should cut off higher frequencies.	PASS	Video 1
1.2.2.1	The left and right chain should both filter and output sound at the same rate.	The filtered sound played should sound stereo i.e. In the left and right ear.	PASS: With a bug that rarely appears.	Video 1 Video 2: Bug

2.1	Create a basic initial state for all the audio parameters.	When I load the EQ, each parameter should have a default value: low cut freq: 0 high cut freq: 20k Peak freq: 750 Low and high cut slope: 12 dB/oct Peak quality: 1.00 Peak gain: 0.0	PASS	Video 1
2.2.1	Can run 2 EQ's on top of each other	To be able to run 2 EQ's on one track.	PASS	Video 1
2.2.2	Can run 2 EQ's with different settings on top of each other.	If one EQ cuts more frequencies than another, there should be a difference in sound when that EQ is turned on or off.	PASS	Video 1
3.1	A dark custom background	The VST's background should be black.	PASS	Figure 1
3.2.1	Custom sliders for each parameter	There should be a total of 7 sliders on the screen.	PASS	Figure 1
3.2.2	Each slider should have their name.	Within the centre of each slider, the parameter ID should print.	PASS	Figure 1
3.3.1	The entry box should serve as an alternative to the sliders.	If you enter a valid value into any parameter's entry box, the slider should snap there.	PASS	Video 1

3.3.2	The entry box will snap to the nearest value position if a value out of range is inputted	Example: if -50 is entered onto high cut frequency, which has a min value of 20, the slider will snap to 20	PASS	Video1
4.1.1	The low cut slope cuts more of the lower frequencies when the slider is increased.	The resulting sound should sound more airy when the slider is increased.	PASS	Video 1
4.1.2	The high cut slope cuts more of the higher frequencies when the slider is decreased.	The resulting sound should sound more murky when the slider is decreased.	PASS	Video 1
4.1.3	The low cut slope cuts less of the lower frequencies when the slider is decreased.	The resulting sound should sound less airy when the slider is decreased.	PASS	Video 1
4.1.4	The high cut slope cuts less of the higher frequencies when the slider is increased.	The resulting sound should sound less murky when the slider is increased.	PASS	Video 1
4.1.5	The low cut slope has a steeper attenuation when a higher cutoff is chosen.	The resulting sound should have a slightly more airy sound.	PASS	Video 1
4.1.6	The high cut slope has a steeper attenuation when a higher cutoff is chosen.	The resulting sound should have a slightly more murky sound.	PASS	Video 1
4.1.7	The low cut slope has a gentler attenuation when a lower cutoff is chosen.	The resulting sound should have a slightly less airy sound.	PASS	Video 1

4.1.8	The low cut slope has a gentler attenuation when a lower cutoff is chosen.	The resulting sound should have a slightly less murky sound.	PASS	Video 1
4.1.9	The peak band moves to higher frequencies when the slider is increased.	Should be able to hear the boosted or mellowed frequencies move higher.	PASS	Video 1
4.1.10	The peak band moves to lower frequencies when the slider is increased.	Should be able to hear the boosted or mellowed frequencies move lower.	PASS	Video 1
4.1.11	The quality of the peak band widens when the slider is decreased.	Should be able to hear a larger range of boosted or mellowed frequencies.	PASS	Video 1
4.1.12	The quality of the peak band slims when the slider is increased.	Should be able to hear a smaller range of boosted or mellowed frequencies.	PASS	Video 1
4.1.13	The gain of the peak band increases when the slider is increased.	The peak band frequency should become louder.	PASS	Video 1
4.1.14	The gain of the peak band decreases when the slider is decreased.	The peak band frequency should become quieter.	PASS	Video 1
4.2.1	When the EQ is disabled, the filters are bypassed.	There will be no bugs where filters work even when off.	PASS	Video 1

Figure 1:



Video 1:
https://youtu.be/MJq-cnYQSP4

Video 2:
https://youtu.be/8PAWFueWjLc

Evaluation:

Objective Analysis

Objective	Objective met?	Comment
Allow the VST to take in an input and output filtered sound	**Yes**	The VST can take in sound in the form of a sample or from a MIDI controller, and output sound.
Use DSP to filter the incoming sound	**Yes, with a slight issue**	The sound is filtered however the user wants, with one bug.
Create a Custom GUI suited toward the end user	**Yes**	The end-user was content with the GUI.
Make the sliders represent values where the user wants to perform actions on the sound	**Yes**	The sliders work well and can be used to change the values of the audio parameters, as well as the entry boxes.
Usage of a response curve and spectrum analyser for visualisation of what the VST is doing.	**No**	I ran out of time before I could fully understand how to implement these two features. They required extensive knowledge beyond that of the average JUICE coder, using complex maths formulae i.e. mapping to pixels with magnitude. With more time, it would be a very good upgrade and addition to my program.

User Feedback:

How easy is the system to use?

I was really impressed with the ease of use of the VST. It does exactly what it says it needs to on the package, which is EQ'ing whatever I want it to. It is definitely universally accessible: you don't need to be a music-making prodigy to be able to understand how to use the system. I especially liked the entry text box system: not many VSTs use this feature, but it is very useful to be able to enter the exact value that I want any of the bands or their characteristics to be at.

How does the VST meet their objectives?

The VST meets the objectives almost perfectly. Everything I asked for was there. The colours were sleek and dark, the interface was simple and efficient, and the sound was actually being EQ'd. I did find one slight issue which I had to inform the coder of: there seemed to be some sort of issue with moving the sliders around a lot over a few minutes, where it seemed like the right ear became muffled, almost as though somebody had panned the audio to the left. Apart from that, it was 10/10!

What improvements or extensions would you recommend?

One improvement I would like to see would be if each slider's values could have their notation. As somebody who is quite experienced with music, I know that a lot of these sliders are measured in either decibels or hertz, but several lower or entry level musicians may not know this. It won't make it so that it won't work, but I feel like just having that notation will make the VST a lot more user friendly and just easier to understand. For example, if the slope cut off didn't have the db/oct notation, I'm sure many people would be confused as to what 12, 24, 36 and 48 meant. Eventually they may understand how to use it, but they still wouldn't understand what it really is.

Another thing that I would improve is the starting point of all of the sliders. What I mean by that is some of the sliders almost have some sort of dead space when it comes to certain sounds. For example, the peak quality slider seems to hit a point at around 5 where the difference between 5 and 10 is much much smaller than the difference between 1 and 5. If you could make it so that the slider felt just more linear across its entirety, it would make the experience smoother.

Analysis of user feedback:

The user gave very useful and constructive feedback that I think is very valid. He also went out of his way to report a bug he had found while utilising the VST, and brought it to my attention. However, he was also very content with what he received and was impressed with the level of professionalism he got. I am confident that I have met the vast majority of my objectives to a high degree from this feedback.

Possible extensions:

The bug that was found takes a while to initiate and results in the right chain processing sound at a slightly slower rate than the left chain. This can be heard by what sounds like the music panning towards the left ear, rather than it being centered.
I believe this bug occurs whenever you change the parameters too quickly several times over the course of around five minutes. Eventually, the strain becomes too great on the processors. However, every time I have managed to cause the bug to occur, it is always the right chain that lags behind. This explanation does not explain why it is always the right chain filters that are lagging behind, and never the left chain filters.
Still, I would like to be able to rectify this issue and make it so that there isn’t any muffling or panning of sounds within the VST.

Another extension I would like to make is the bypassing of certain filters within the actual VST. Many VSTs today have an in-built feature where users are able to bypass certain aspects of their VST. In my case, rather than just leaving an audio parameter at it's default value i.e. the low cut slope frequency being left at 20hz, there could be a button you could press instead that deactivates the low cut filter in both of the chains. This would be incredibly useful as instead of having to bypass the entire VST, users could just turn the parts that they don't require off. This also allows users to compare the original sound to certain parts of the filter more easily: so they can have a better understanding of the effect of what they're doing in the VST has on the music. This would make my VST a lot more user friendly, as well as add another layer of useful features.

A final extension I would like to add is a textbox that appears if you right-click, or hover, over each slider, detailing exactly what they do. For example, for the high cut frequency, it could say 'This slider changes where the processor will cut the high frequencies from.' This would add another layer of user friendliness: by explicitly being told by what each slider does, even those completely new and looking to get into music could use this VST to learn. This addition to my code would make it more accessible for all users.