Final Year Project
Music Technology

Author:

João Maurício

Title:

Assisted Spectral Rebalance Plug-In for Mixing and Mastering

Supervisor:

Phill Phelps

Programme:

Audio and Music Technology

Date:	Student No:	Word count:
May 2021	18030724	13179

UWE Bristol | University of the West of England

SUMMARY

This project will detail the creation of an audio plug-in in C++ using JUCE, which will assist audio engineers in rebalancing the spectrum of a piece of audio by comparing and matching it with a suitable reference provided by the user. It can be useful in cases where there are multiple vocal takes in a mixing session, recorded in different conditions (different preamp, different mic, distance to the mic or even the tilt of the mic) but the mixing engineer needs the spectral characteristics to be consistent across the entire performance. Another example could be if a mastering engineer is given a reference track from the artist and wants to quickly demo how the artist's track would sound with a similar spectral balance.

This report will oversee the development of the DSP needed for this system to be built, as well as explaining the main concepts and theories that need to be known in order to progress. Different implementations will be explained, paving the way for the final implementation to be detailed and tested.

The final implementation analyses the long-term average spectrum of the two audio excerpts and calculates the spectral difference between the two. An array of 256 Infinite Impulse Response peak filters is then used to spectrally match the Current to the Target, using the difference analysis as a reference for their gain, Q and centre frequency values.

AU and VST3 versions of the plug-in were created and made available in the appendix, and audio examples are presented for the reader to listen to the results.

ACKNOWLEDGMENTS

To my girlfriend Catarina, thank you for keeping me sane throughout this whole final year of university and supporting me through all of the setbacks and disheartening moments that came from this project. I can definitely promise I will never talk to you about FFTs and audio filters as much as I did this this year ever again. I'm honestly thankful for the fact that you have been on my side for all these years.

To my parents and my brother, thank you for believing in my potential and always backing up my decisions. Knowing I have your support has been incredibly important for my entire journey through university. Specially to my parents, thank you for providing me with the opportunity to come and study in a different country, I will always be grateful for the sacrifices you have made for me. This project is as much yours as it is mine.

To Phill, thank you for being the best project supervisor I could have asked for. Your positive approach to always helping every single student has struck me since year 1 and I think that is an incredible trait to have. It has been a great pleasure to work with you throughout this entire project.

TABLE OF CONTENTS

1. INTRODUCTION

In the recent years, the process of *overdubbing* during the recording stage of a song has become more and more common. “During this phase, additional tracks are added by monitoring the previously recorded tracks” (Huber, 2017). This is particularly useful, when one of the musicians makes some mistakes, as they can simply re-record the same parts various times, sometimes stacking dozens of different takes. The next step is to “refine the recorded tracks into their final form” by “pasting together (comping) the best section of several recorded takes to create a single master take” (Senior, 2017, p.386). However, because there is no guarantee that every single take would have been recorded in the same conditions (same microphone, preamp, distance to the mic or even the same room), they might sound different to each other, which can be noticeable. To solve this, mixing engineers can resort to equalisers, to aurally match the tone of every take used. However, this process can be time-consuming and, in a lot of cases, lost time is lost money for audio engineers.

Although there are some audio plug-ins that already can assist this process, referred to as “EQ Matching” software, this is a tool that still seems relatively unexplored, as there is a very limited number of options currently available.

The main goal of this project will be to research the technologies and concepts behind this type of tool, to then create an audio plug-in that will be able to analyse the spectral differences between two audio sources and use equalisation to assist the user with matching one source to the other. For examples of other uses, see Chapter 8.

Additionally, it is also of personal interest to the author to learn about the process of designing and implementing audio-plugins, so they can be distributed for different DAWs and be used in a professional environment. It is then important to mention that the author had no previous experience in fully building an audio plug-in before this project and, as such, a lot of personal development will come from this challenge.

With the main goal set, it can be divided into different objectives:

- Researching techniques and concepts behind spectral measurement and analysis, to be used in a programming environment.
- Studying and using JUCE for the first time as a suitable framework to create audio plug-ins with C++ that could be professionally used.
- Creating a plug-in that can analyse the spectrum of two given audio tracks and spectrally rebalance one to match the other.

- The plug-in should work with different types of source material (vocals, guitar, drums, full mixes, etc.)
- It should support different plug-in formats to work with most major DAWs.

For this report to remain concise and clear, it is important to limit the scope of this project. Although a significant amount of time will be spent learning how to program an audio plug-in with C++, the report will instead cover the theory and DSP concepts needed to make this tool, as well as combining the research with the digital signal processing design and implementation. This seems more useful for the author to solidify in the report, rather than explaining advanced audio programming concepts and techniques.

The following report will start by exploring existing technologies that solve similar problems to the ones mentioned above. This will be used as a starting point to then uncover the concepts that need to researched. Digital audio analysis, in the form of long-term average spectrums, will be studied, along with the Fast Fourier Transform and how it can be used for this project. After most of the research has been completed, the implementation of the different prototypes will be detailed, as well as its progression. Once the author gets to the final implementation, additional testing will be reported, to allow for the plug-in to be evaluated and for future improvements to be suggested.

2. EXISTING TECHNOLOGIES

As a starting point to this project, it is important to research existing plug-ins that already try to solve similar problems to the ones mentioned in the introduction. In this way, it will be possible to note down some key concepts to explore, as well as common (or distinguishable) features between the products. Later, this research will also reinforce the need for this project and some of the reasoning behind it.

There are three main existing plug-ins that are going to be studied:

- iZotope Match EQ
- FabFilter Pro-Q 3 (EQ Match feature)
- Logic Pro Match EQ

2.1. IZOTOPE MATCH EQ

iZotope (2011, p.37) defines their Match EQ as a tool that “works hand in hand with spectrum snapshots to “borrow” the spectrum of one audio clip and apply it to another”, which is also one of the goals of this project. They claim it is a “digital linear phase EQ, with the ability to use over 8,000 bands of frequencies for very precise matching”.

Currently, the most up-to-date Match EQ plug-in from iZotope is exclusively available through their Ozone 9 Advanced package, which is priced at 423.20£ (including VAT) (iZotope, 2021).

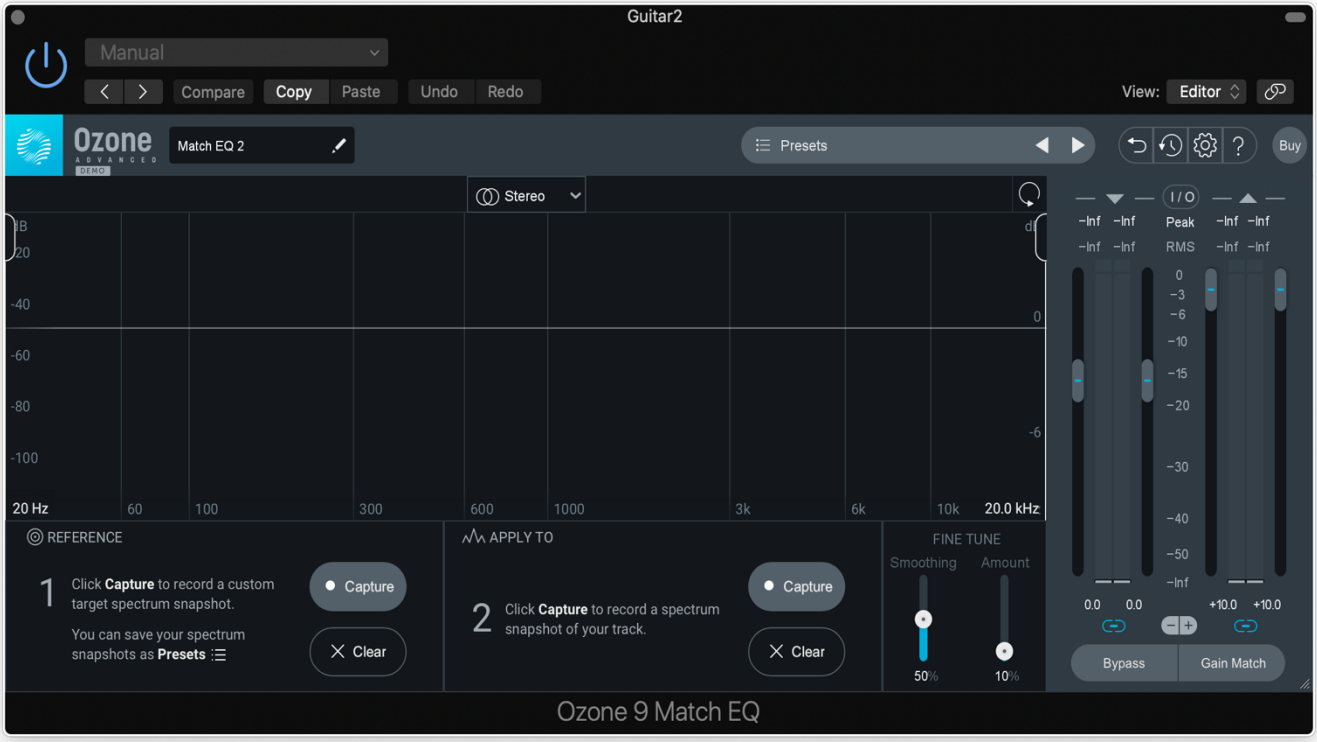

Figure 1 - Screenshot of iZotope's Match EQ user interface

After experimenting with this tool, and using the above figure as a reference, we can take a few notes regarding the used workflow:

1. Firstly, we load this plug-in on the reference track we want to use.
2. Turning our attention to the “Reference” section of the UI: after starting the DAW playback, we click on “Capture” to begin capturing the spectrum of the audio track this plug-in is loaded onto. Afterwards, a “Stop” button replaces “Capture”, and we can click on it whenever we have finished capturing the desired portion of the recording. Optionally, we can save this “target spectrum snapshot” as a preset for future use, or we can click on “Clear” to restart the capture process.
3. We then move this same instance the plug-in to the track that we want to approximate (or to a different playhead position if using the same DAW track), and repeat the previous step, this time in the “Apply To” section of the UI.
4. As shown in the figure below, an EQ curve (in grey) is then generated and displays the processing being now applied to the source material, which is based on the difference between the two captured spectrums, as well as the Fine Tune settings.

5. Looking at the “Fine Tune” section of the UI, the “Smoothing” slider “determines the amount of precision to apply to the matched curve, whereas the “Amount” slider “determines the amount of processing (intensity) to apply when matching the Source curve to the Target curve” (iZotope, 2019).

Figure 2 - iZotope's Match EQ UI after analysing two guitar takes played through different guitar cabinets. Files “GuitarTarget” and “GuitarCurrent” from the ReportExamples folder in the appendix.

One very important aspect to mention and that was also tested, is that this matching process seems to be independent from the gain differences of the two tracks used, as the resulting curves are the same even with drastically different gains between similar tracks.

2.2. FABFILTER PRO-Q 3 (EQ MATCH FEATURE)

Pro-Q 3 is an EQ plug-in from FabFilter which has a multitude of different features built into it, including EQ Matching. FabFilter explains the goal of this feature very similarly to iZotope, saying that it's a process that lets us "record a reference spectrum, compares it to the spectrum of the current input, and adds new EQ bands to make the audio sound like the reference signal" (FabFilter, 2018, p.27).

Even if users only want access to the EQ Match feature, they still need to pay the full price of the entire Pro-Q 3 software, which is currently priced at 134£ (FabFilter, 2021).

Figure 3 – FabFilter Pro-Q 3 UI.

After some testing, it is possible to say that the workflow to match two spectrums using Pro-Q 3 is fairly similar to iZotope's Match EQ. With that in mind, it is not worth going through the various steps, so only the main differences will be detailed.

Pro-Q allows the user to select an external side-chain input as the reference, which provides a way to analyse both the reference and source at the same time. This is in addition to the same options that iZotope offers, such as previously saved spectrums or main input capture.

After the analysis, “Pro-Q automatically calculates how many and what kind of EQ bands are needed to match the sound”. It “proposes a number of new bands and gives the opportunity to customise the matching detail, using the slider”, as show in the figure below (FabFilter, 2018, p.28).

Figure 4 – Pro-Q 3 after analysing two guitar takes played through different guitar cabinets. Same files from the iZotope example.

The slider allows the user to choose the number of bands that the EQ Match uses, although Pro-Q automatically sets a recommended number. “By choosing more bands, even the smallest differences in the analysed spectrums will be matched, while choosing less bands will only cover the main shape of the difference spectrum” (FabFilter, 2018, p.28).

2.3.LOGIC PRO MATCH EQ

Logic's Match EQ is a stock plug-in currently available for all Logic Pro X users at no extra cost. However, its price can be defined as 199.99£ (current price of Logic Pro X) (Apple, 2021) being that this tool is exclusively available for this DAW.

Although Logic's manual describes similar uses to those of the plug-ins mentioned earlier on, it gives more emphasis to using this tool with full mixes rather than short sections of single audio tracks.

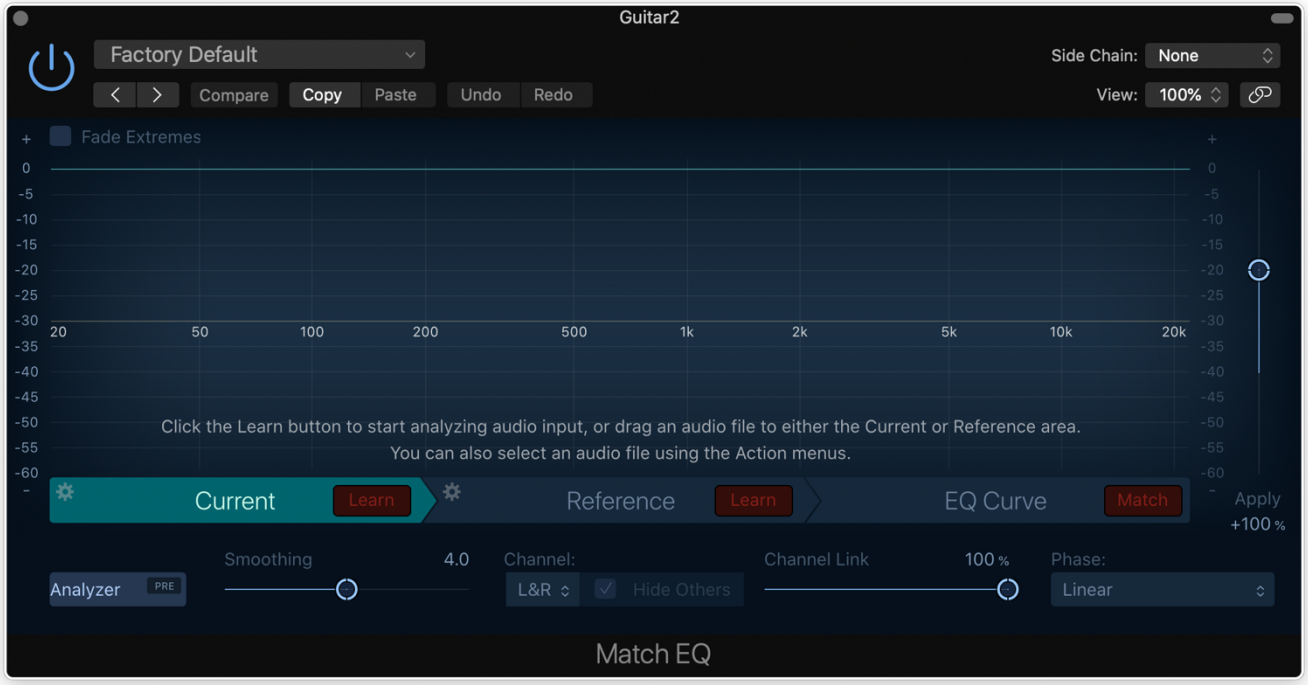

Figure 5 – Logic Pro's Match EQ UI.

Apart from getting the reference/source from the main input, sidechain, or previously saved spectrums, Logic's Match EQ also allows us to load an audio file directly into the plug-in, which is then analysed in just a few seconds.

When using stereo signals, a slider called “Channel Link” is enabled and allows the user to either have a common difference EQ curve for all channels (100%) or have different curves for each channel (0%). Values between 0% and 100% blend the filter curves.

Figure 6 – Logic’s Match EQ UI after analysing two guitar takes played through different guitar cabinets.

After the analysis, the EQ match can be customized by using the “Smoothing” and “Apply” sliders, which, respectively, “set the amount of smoothing for the filter curve, using a constant bandwidth set in semitone steps” and “determine the impact of the filter curve on the signal” (Apple, 2020, p.117). The operational principle of the filter curve can be changed using the “Phase” pop-up menu, which gives the following options: Linear, Minimal, or Minimal Zero Latency.

2.4. COMMENTS ON EXISTING PLUG-INS AND NEXT STEPS

Reflecting on these three existing plug-ins and the research of existing technologies, it is possible to say that this is still a very unexplored tool, as there was a very limited number of EQ matching plug-ins found during this process. Additionally, the ones which were found are not accessible for a lot of audio engineers: iZotope's Match EQ can only be bought as part of Ozone 9 Advanced (359.90£); FabFilter's EQ matching is only a feature (not a dedicated plug-in) of the Pro-Q 3 equaliser, which costs 134£; Logic's Match EQ is exclusive for Logic Pro (and MacOS) users, which costs 199.99£. This reinforced the curiosity and need behind this final year project.

It is worth mentioning that there were various common features between the explored plug-ins, such as:

- Being split into two different stages: **analysis** and **filters implementation**;
- When "analysing input or reference audio, **the spectrums average over time**" (FabFilter, 2018, p.28);
- The user almost always needs to fine-tune the resulting curve. iZotope (2019) mentions that "a matched curve amount of 100% and a smoothing amount of 0% might technically be the closest match to the "Reference" mix, but, in reality, it's probably not the most effective combination of settings. Those will try to capture every peak, valley, and level, which can result in extreme, unnatural EQs";
- FabFilter and Logic Pro allow the user to further tweak the resulting EQ curve by adjusting individual bands;
- iZotope and Logic Pro seem to normalise the spectrum analysis. Apple (2020, p.120) states that the resulting EQ curve "automatically compensates for differences in gain between the template and the current material" and is "referenced to 0dB".

With this in mind, next steps regarding concepts to explore are now able to be set. Being that this EQ tool relies heavily on the analysis of two signals, let us start by looking into **digital audio analysis,** specifically in the form of **average spectrums**.

3. BACKGROUND RESEARCH

To describe the development of this project, it is important to define some of the main concepts and theories that it relies on, as well as linking them to current and past research on said topics.

3.1. DIGITAL AUDIO ANALYSIS

The first concept that is fundamental to be mentioned is **digital audio analysis**. Digital audio systems "use time sampling and amplitude quantization to encode the infinitely variable analog waveform as amplitude values in time" (Pohlmann, 2010). Those values can then be played and converted back to its analogue counterpart, which we, as humans, can perceive through our auditory system. In the following figure, a 100Hz sine wave is represented in a time domain waveform plot, which Creasey (2016) states that it is "analogous to the air pressure variation that arrives at the human ear".

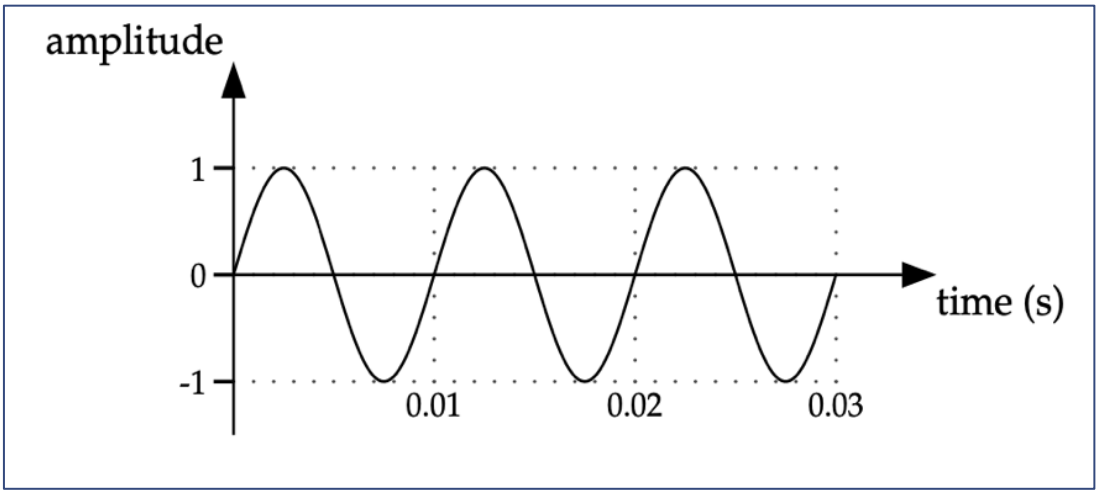

Figure 7 - 100Hz Sine wave represented in the time domain. X-axis represents seconds and Y-axis represents amplitude.

Observing solely the figure above, we could actually determine the frequency (and consequently, the pitch that we perceive) of this signal, through some simple calculations, given that frequency is related to the period of a wave (time it takes for that wave to complete one cycle). With that in mind:

$$\text{Frequency} = \frac{1}{Period} = \frac{1}{0.010\ seconds} = 100\text{Hz}$$

However, with more complex signals (such as the figure below), Creasey (2016) states that "there comes a point at which a simple plot of signal amplitude over time is not enough to explain everything that can be heard", as there is a "lack of a clear link between waveform shape and perception".

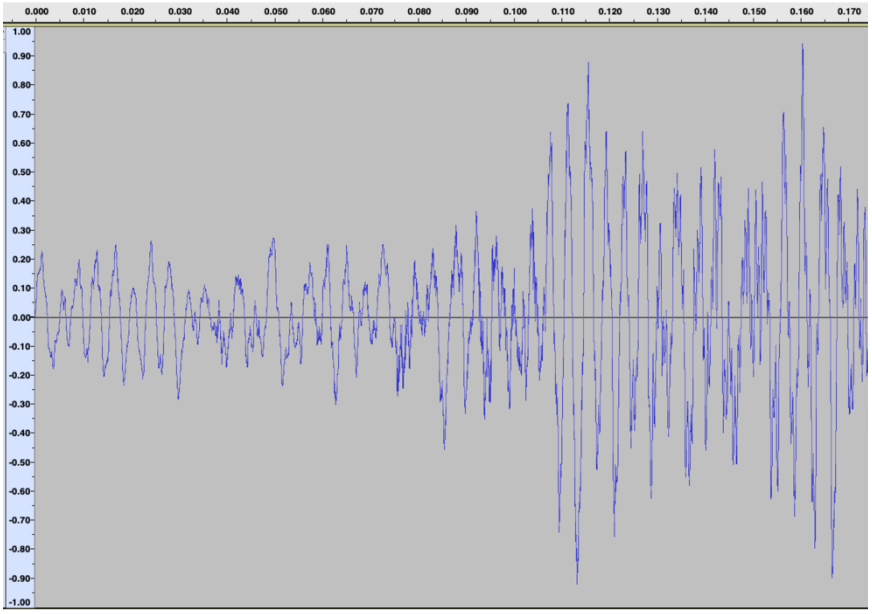

Figure 8 – Time domain representation of the first strum of the "GuitarTarget" file in the ReportExamples folder of the Appendix.

One solution is, instead, to use frequency domain analysis, commonly displayed through a **frequency spectrum plot**.

3.2. AUDIO SPECTRUMS

A frequency spectrum plot can be considered a snapshot of a signal in time that represents amplitude (Y-axis) according to frequency (X-axis). It relies on the Fourier Theorem, which states that it is possible to deconstruct complex sounds as a "combination of sinusoidal elements of different frequencies." Creasey (2016) believes that this frequency domain representation can be used to better understand features of the sound analysed, such as

harmonics, inharmonics, noisy elements, but most importantly for this project, the spectral envelope.

From a more musical point of view, Lerch (2012, p.41) states that “the features describing the spectral shape of an audio signal are closely related to the timbre of this signal” which is regularly “referred to as its sound colour, its quality or its texture”, and “allows humans to group together different sounds originating from the same source”.

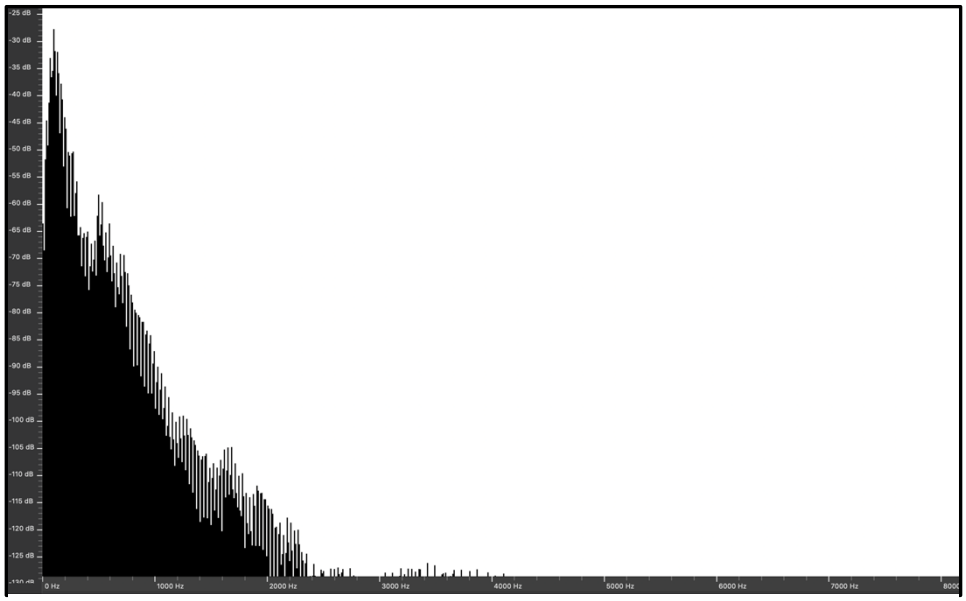

Figure 9 - Example of the spectrum of a bass guitar – “BluesBass” in the ReportExamples folder of the Appendix.

In the above figure, a prediction can be made that the analysed audio signal has a bass-heavy timbre as its spectrum conveys that most of the energy is located in the low frequencies area. That is exactly the case for the recording of the bass guitar analysed.

It can then be concluded that **frequency spectrums** are a useful concept to apply in this project. However, the reader might think: How can a representation of an audio signal be converted from the time domain to the frequency domain? That is where the **Fourier Transform** comes in.

3.3. FOURIER TRANSFORM

As mentioned previously, the Fourier Theorem is one of the backbones of spectral analysis, as it can deconstruct complex waveforms into a combination of sine waves. To deconstruct a sound, we can perform what is called a **Fourier transform.**

According to Park (2010), the Fourier transform is "one of the most powerful concepts in digital signal processing (as well as other fields of study such as mathematics, physics, electrical engineering, computer science, etc.) and incredibly important for various signal processing applications including (…) sound manipulation, filtering and **spectral analysis**."

3.4. FAST FOURIER TRANSFORM

The Fourier transform takes a time domain input and transforms it into the frequency domain, where we can extract information about the frequencies the input contains and their magnitudes, which can then be displayed graphically in a frequency spectrum plot (Creasey, 2016). However, it is important to mention that the Fourier transform is intended to be used on **continuous signals** (like analogue audio), and digital audio is a **discrete signal**. According to Park (2010) another technique that we can use instead is the **Discrete Fourier Transform (DFT)**, which is able to do the same conversion as the Fourier Transform but with discrete signals.

However, "the DFT requires an excessive amount of computation time, particularly when then number of samples, *N*, is high". This directly takes us to the **Fast Fourier Transform (FFT)**, which "is an algorithm to speed up DFT computations", by "reducing the number of calculations" (Sevgi, 2007).

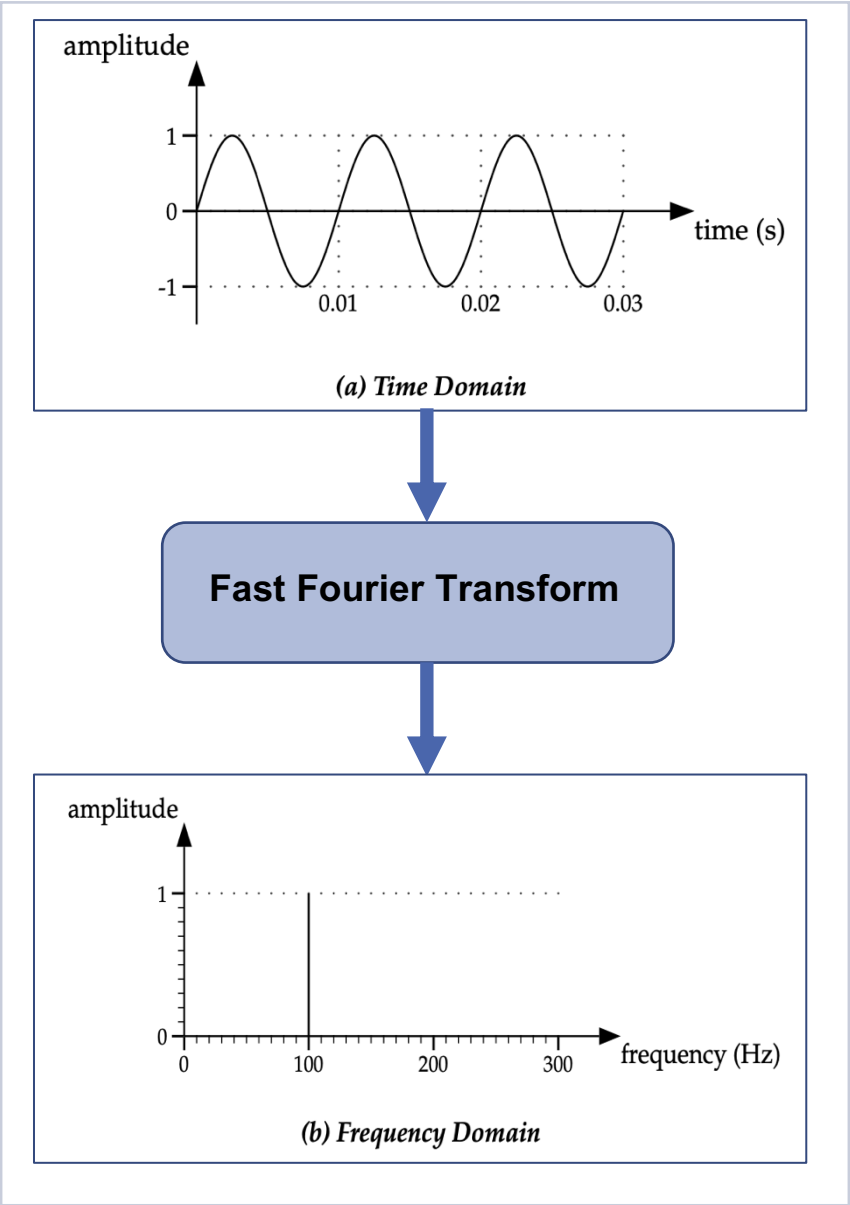

Figure 10 - Visual representation of the conversion that the Fast Fourier Transform performs.

The FFT receives a number of audio samples as a block (**the number of samples is a power of 2** (256, 512, 2048, etc.) and is called the **FFT size**) (Lerch, 2012). After receiving it, it processes the data and outputs a frame (set of values corresponding to different frequencies) composed of a number of frequency components that is the same number as samples received (FFT size) – these X-axis discrete frequency points are often called **FFT bins/Frequency bins/bins**, according to Park (2010).

In other words, “the X-axis transforms from a time-domain axis to a frequency-domain axis” and time sequences of *N* number of samples are then each represented by *N* frequency components, “making up a frame after the Fourier Transform” (Park, 2010). For example, a “512 samples signal becomes a frequency-domain representation of 512 sinusoids with their corresponding 512 frequency (X-axis) and magnitude/phase (Y-axis) values.” For now, we will only focus on the frequency and magnitude values of the FFT output.

3.5. WINDOWING FUNCTIONS

When breaking apart our signals into blocks (or buffers) of *N* samples to be processed, there is a very high chance that the first and last amplitude values of each block will not be zero. “When that is true, the FFT will produce noise in the output due to the discontinuity of the signal” (Bagley, 2020).

Before looking at what windowing functions are, let us have a look at the frequency spectrum plot of a 100Hz sine wave in Amadeus Pro. What one would expect, from reading the sections above, would most likely be a graph similar to this one:

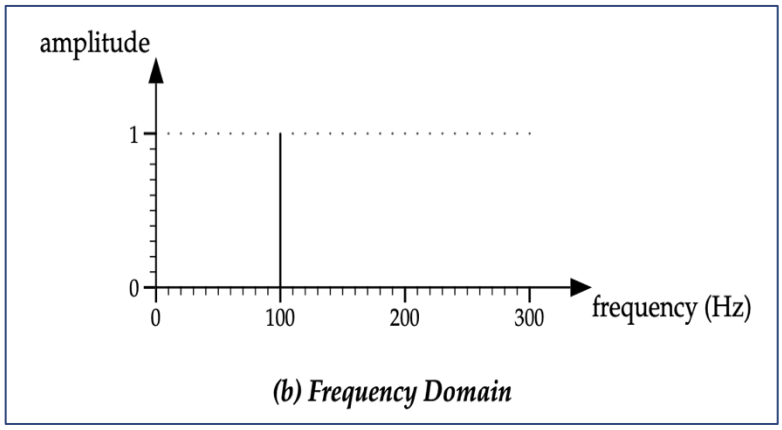

Figure 11 - Expected frequency domain representation of a 100 Hz sine wave.

However, what we get, **without using any windowing function**, looks like this:

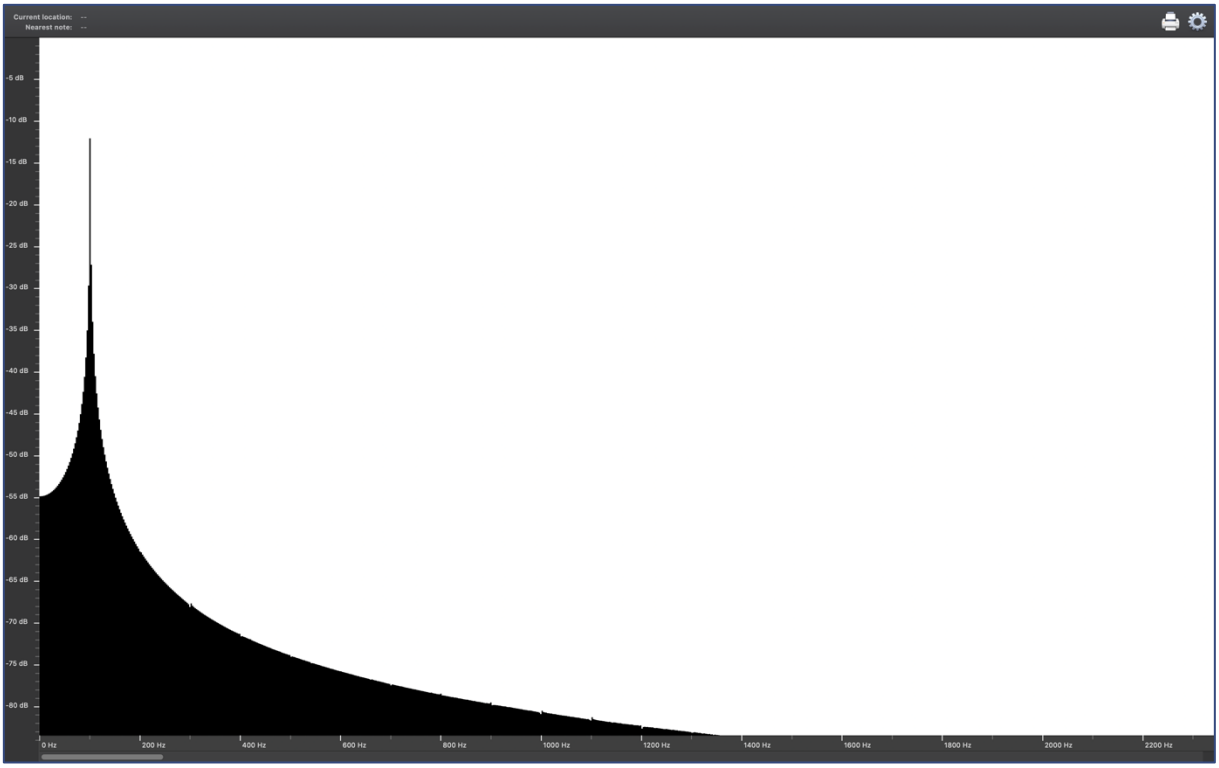

Figure 12 - Spectrum of a 100Hz sine wave from Amadeus Pro. FFT size of 8192 without any windowing function.

The frequency domain distortion that we observe in the above figure is one the of practical limitations of the FFT and it is described as spectral leakage, “characterised by the smearing of the spectrum” (Wickramarachi, 2003). Should the input signal have had a 200Hz sine wave with a lower amplitude, it would be masked by the smearing.

This happens “when a non-integer number of periods is acquired” from the input (Wickramarachi, 2003), causing “the spilling of energy centred at one frequency into the surrounding spectral regions” (Rapuano and Harris, 2007), as we can observe in the figure above.

To reduce this distortion, Bagley (2020) suggests that we use a windowing function to process the input samples, which essentially “smooths the beginning and end to make sure they transition smoothly”. There are various choices for windowing functions, each with different advantages and trade-offs. Some examples of these functions are the Hanning, Blackmann and Hamming functions.

If we now analyse the same 100Hz sine wave using the same FFT size but process the input samples with a windowing function (Hanning, in this case), we get this plot instead:

Figure 13 - Spectrum of a 100Hz sine wave using Amadeus Pro with an FFT size of 8192 and Hanning windowing.

This is now much closer to the results we would expect after reading about what an FFT does, supporting the fact that window functions can be used to reduce the effects of spectral leakage, while improving the readability of FFT spectrums.

3.6. LONG-TERM AVERAGE SPECTRUM

As discussed earlier on, spectrum plots are normally a snapshot of the frequency content of a signal in time, as the FFT outputs its results using a limited number of samples. This means that, for a given audio signal, the spectrum in one point in time might be very different from another point in time.

Wichern (2017) mentions that “while the spectrum analyser is a great tool for identifying resonant and fundamental frequencies, it provides too much information for analysing tonal balance”, which we can relate to timbre (as mentioned in chapter 3.2). To help with this, Wichern (2017) suggests that “for tonal balance, we want something that measures overall frequency content, so it should be averaging on the scale of several seconds or even over the entire track”. “If a source is producing a fairly consistent sound quality, it is possible to average the frequency description”, which “attenuates short (transient) details and the development of the sound over time in general.” This gives us the “consistent core structure of the sound” (Creasey, 2016).

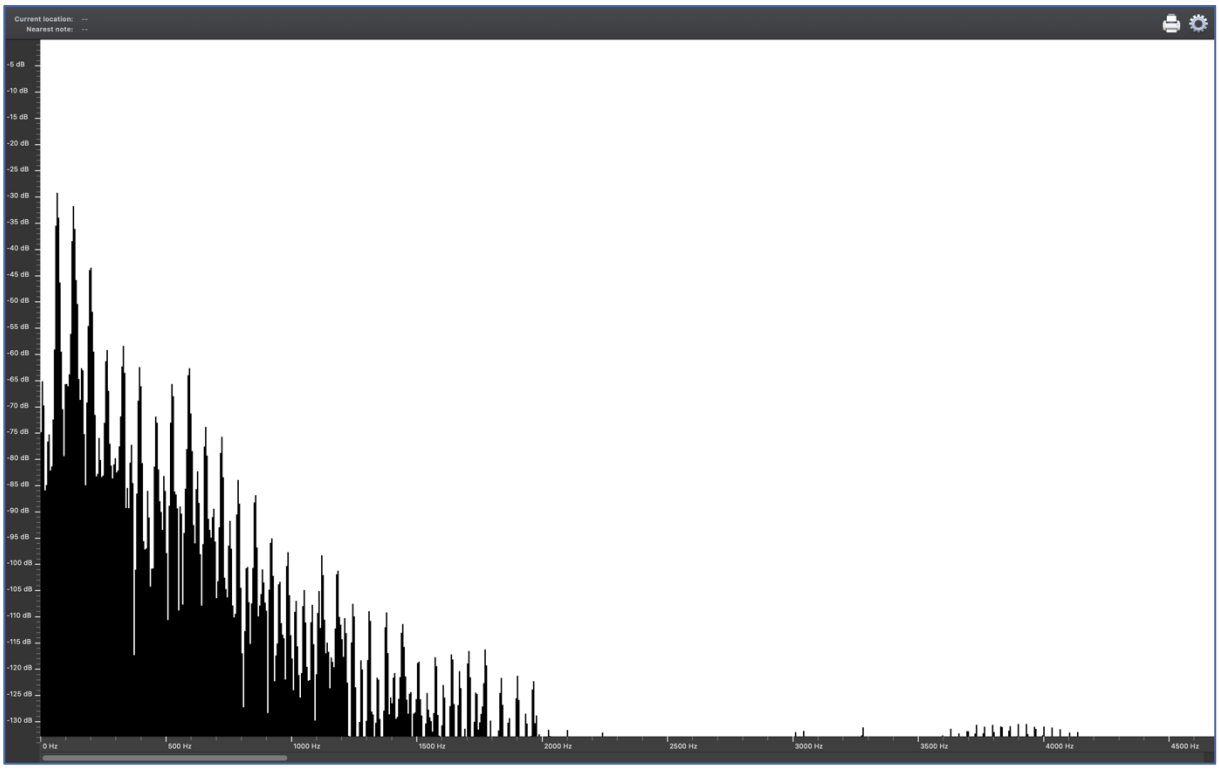

Figure 14 - Non-averaged spectrum of a bass – “BluesBass” file in the ReportExamples folder in the Appendix - analysing only start of the signal. FFT size of 4096 with Hanning windowing.

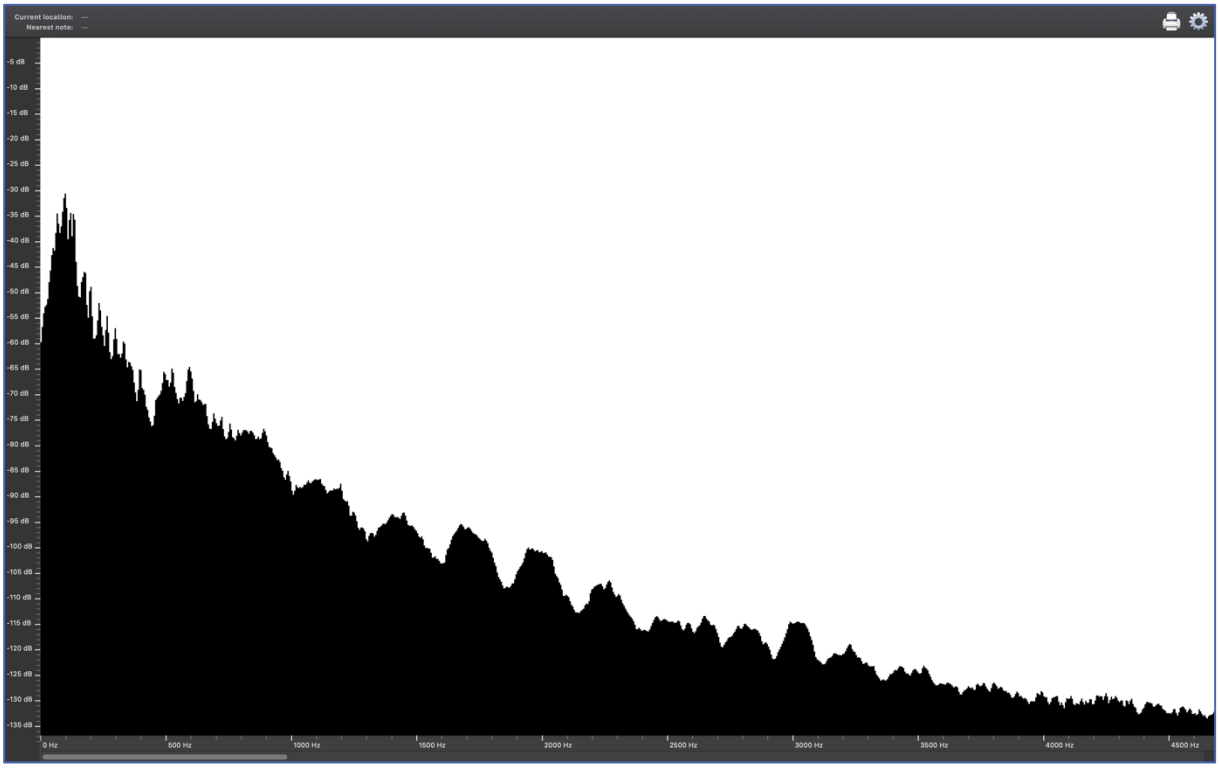

Figure 15 - Spectrum of the same bass guitar as Figure 14 but averaged over the entire duration of the file.

Observing the two figures above, we can see that, using the same audio signal, the long-term averaged spectrum (Figure 15) provides a much higher-level representation of the overall frequency content (and timbre), without all the potentially unnecessary detail of a real-time spectrum. This average representation seems to be much more useful to compare two different acoustic sound sources, which produce sounds that change over time.

Additionally, a study was conducted by Pestana *et al.* (2013) where a large dataset of number one songs in the US and UK charts was used to analyse the spectral characteristics of popular commercial recordings and its evolution between 1950 and 2010. To do this, the authors used the "monaural (left + right channel over two) average long-term spectrum of the aforementioned dataset". It was concluded that "the spectra of professionally produced commercial recordings show consistent trends, which can roughly be described as a linearly decaying distribution of around 5dB per octave between 100 and 4000 Hz, becoming gradually steeper with higher frequencies, and a severe low-cut around 60 Hz."

However, what is most important for us is the fact that Pestana *et al.* (2013) mentioned this knowledge "could be useful for a more informed implementation of match-EQ type plug-ins (…), as well as for automated equalization work". This further hints that average spectrums could be a very useful tool for the software that this final year project is aiming to develop.

4. PREPARING THE C++ IMPLEMENTATION

Going back to the project proposal, one of the main goals that was set for this project was to build an audio plug-in. However, it is very important to emphasise that the author had no previous experience on fully programming a plug-in, and, as such, a lot of the time spent on this project involved solidifying audio programming techniques and best practices to deploy software that would be opened in DAWs such as Logic Pro and Reaper, so that it could easily be used by mixing and mastering engineers.

According to the proposal, the first prototypes were going to be programmed in Max MSP, being that the author had some experience with it, and it provides a good environment to quickly design and implement audio processes as a way to experiment with the concepts learnt through the background research.

The next step planned was to study techniques to port the finished Max patches into a C++ implementation that could be exported as a plug-in. JUCE seemed to be a good resource for this, as it is “popular for developing audio applications and audio plug-ins” (Robinson, 2013). It is a “framework for developing cross-platform software in C++" which “provides classes to manage audio I/O, audio processing, and media reading and writing”, as well as user interface and graphics. JUCE enables their users to “build VST, VST3, AU, RTAS and AAX format plug-ins with ease” (JUCE, 2020).

Other factors that make JUCE a good way forward are the existence of a dedicated forum where a lot of users ask questions and share information, and the fact that their website also offers tutorials on some important concepts and techniques. These were particularly important considering this was the first time the author was using this framework.

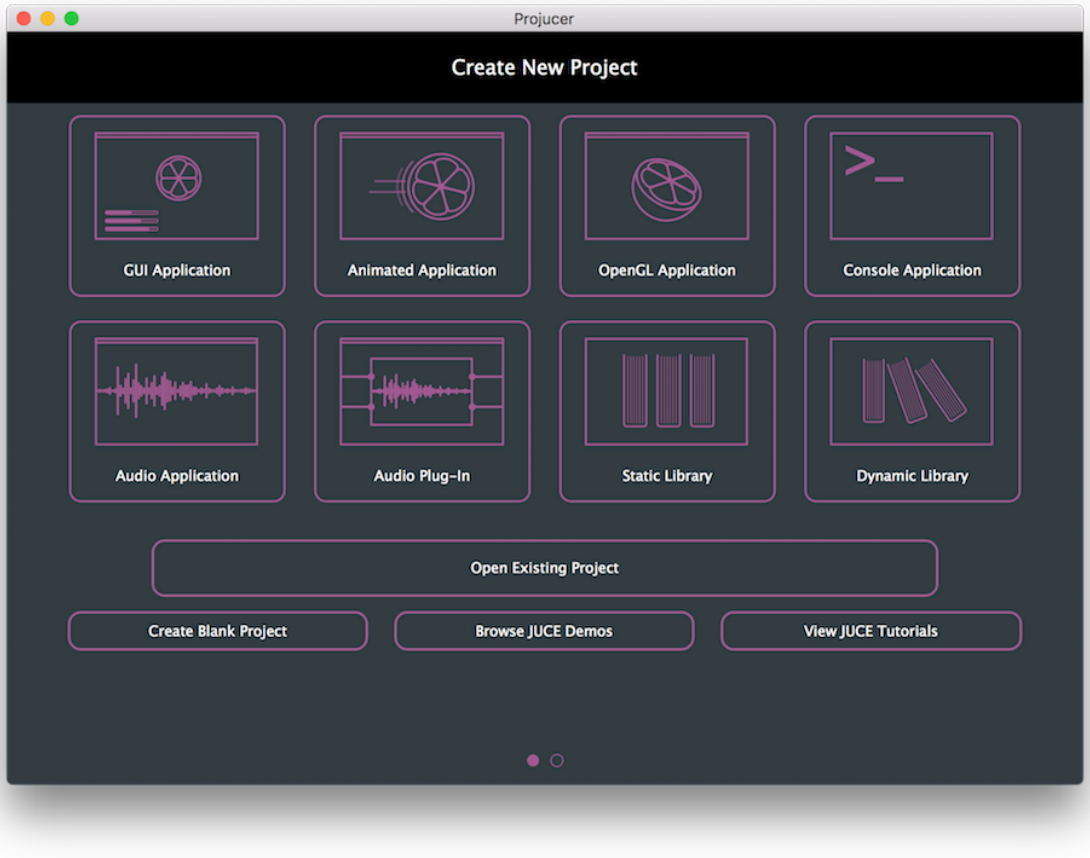

Figure 16 - Screenshot of the Projucer, a software made by JUCE to manage and create JUCE projects.

However, a few weeks after the project proposal had been submitted, the author started working with JUCE sooner than expected through the Software Development for Audio (SDA) module, which expanded materials covered in Introductory Audio Programming and Audio Process Design and Implementation, from Years 1 and 2, respectively.

SDA ended up being very important for the development of this project, as it covered lots of fundamental topics (such as threading, separating graphical user interface from audio processing, inheritance, working with JUCE, etc.), specific to building audio applications. This module also uncovered a lot of the constraints and difficulties of programming audio plug-ins: making sure the audio is continuously running without any glitches or drops; ensuring parameters are saved/recalled when closing/opening a DAW project; connecting the user interface and audio processing in a safe way; making sure the audio processing is still running with the graphical user interface closed, etc.

Because of this, it was decided that it would be better to skip the Max MSP implementation completely and start prototyping directly in C++ with JUCE, as there was a considerable

amount of material for the author to learn and practice (both from SDA and the JUCE website tutorials).

Additionally, it was decided that it would be best to start by prototyping an audio application rather than a plug-in, given that this was the process being studied in the Software Development for Audio module, and it involved less concerns (such as the ones mentioned in the paragraph above) for the first few implementations. Later, once the author was more comfortable with JUCE, the final implementation could be ported into a final plug-in.

5. IMPLEMENTATION - ANALYSIS

Having researched key topics and studied the JUCE framework for some time, it is now possible to start working on the C++ implementation. As was seen in the Existing Technologies section of this report, it is a suitable approach to divide this plug-in into two different steps: **analysis** and **filters implementation**.

5.1. CREATING A LONG-TERM AVERAGE SPECTRUM

Starting with the analysis, the first step will be to create a simple application that plays an audio file and graphically represents its real-time spectrum. Afterwards, we will change the implementation so that it continuously calculates the average spectrum for the audio file while playback is on.

In the Software Development for Audio, a rudimentary application had already been programmed in JUCE which opens and plays an audio file, to which the author attached a white component where the spectrum is going to be displayed, as shown below.

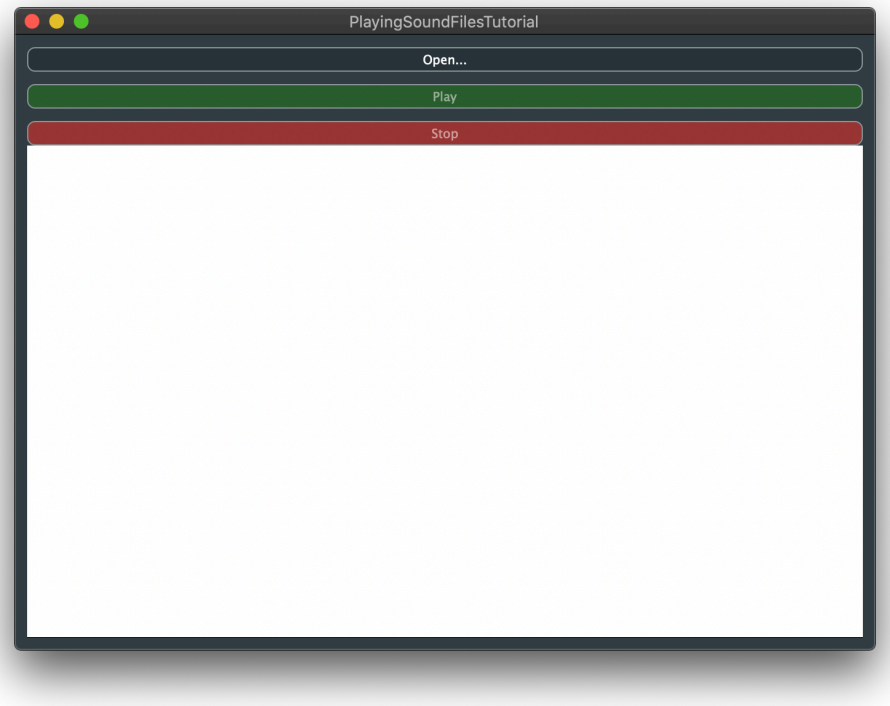

Figure 17 - Application developed in the SDA module that opens and plays an audio file.

From our previous research, it is now understood that, to display a spectrum, the audio needs be processed through an **FFT**, after being passed through a **windowing function**. One important feature from JUCE is that it already contains some of these techniques encapsulated as classes that we can use, such as the **dsp::FFT** and **dsp::WindowingFunction** classes.

We can set up the FFT by specifying its FFT order, which then also sets the FFT size:

```
enum
{
    fftOrder = 11, // 2 to the power of fftOrder = fftSize
    fftSize = 1 << fftOrder
}
```

JUCE suggests calculating the fftSize using the left bit shift operator, which produces the fftSize as a binary number, from the fftOrder. In the excerpt above, the fftSize is set as 2048, which we will use for the rest of the project. Higher FFT sizes were tested but they demanded more processing power and were making the software slower, as the dsp::FFT class is “only a simple low-footprint implementation and isn’t tuned for speed” (JUCE).

Regarding the windowing function, JUCE offers a few built-in options, such as Hann, Hamming and Blackman. In their “Visualise the frequencies of a signal in real time” tutorial, the Hann function is used, and is described as having “good dynamic range, good resolution” and being “typically used in narrow-band applications” (JUCE). This is also the default windowing function that iZotope’s Match EQ uses. For these reasons, the Hann function will be used as a starting point for this project.

When dealing with digital audio, it is very common to use buffers to store and read audio samples (Creasey, 2016). Audio interfaces and DAWs normally allow its users to configure the buffer size for the system – Logic allows users to choose between 32, 64, 128, 256, 512 and 1024. This means that, for an FFT to output data, it might need to be provided with more than one buffer of samples for each spectrum frame, depending on its FFT size. To overcome this, we are going to implement a FIFO (First In, First Out) array structure, which is going to be based on JUCE's "Visualise the frequencies of a signal in real time" tutorial.

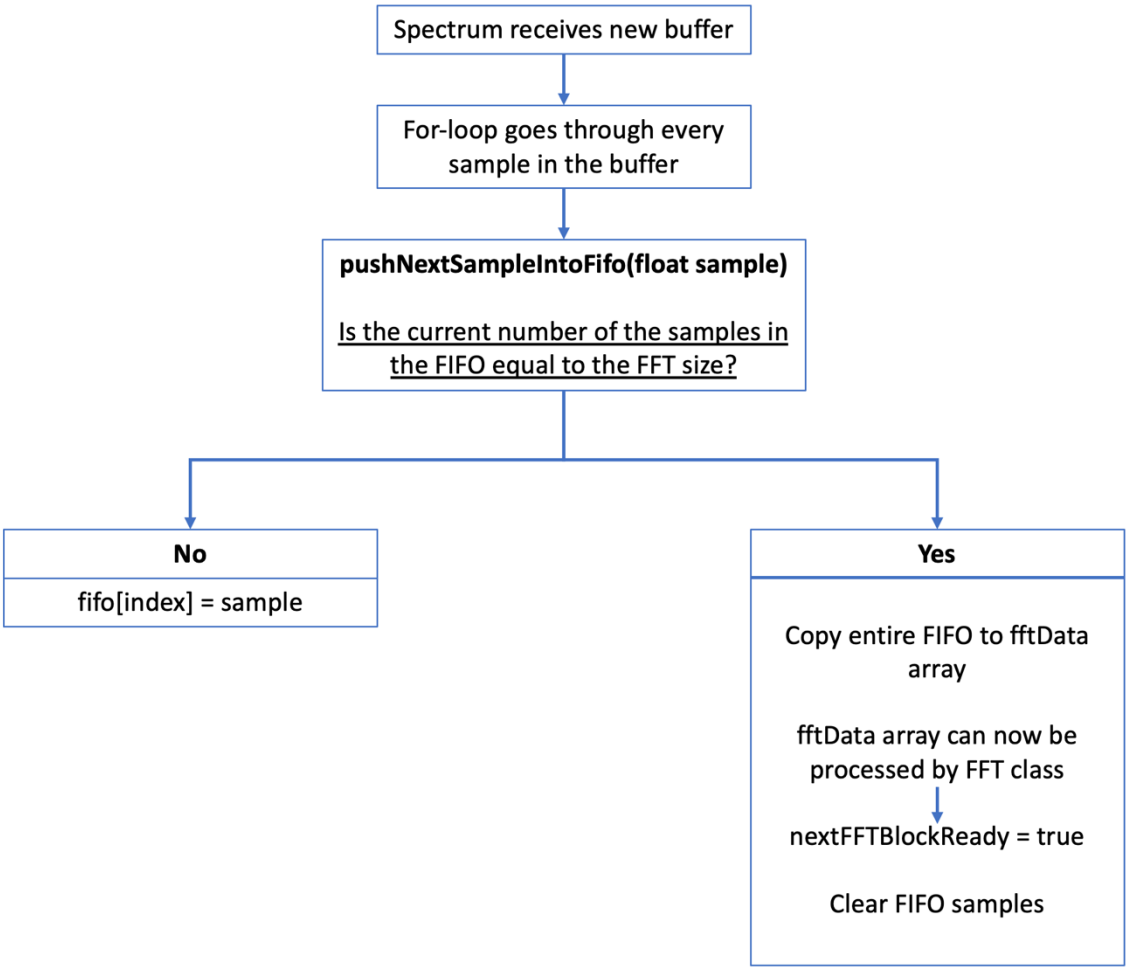

Figure 18 – Diagram explaining the FIFO structure.

It is worth mentioning that our Spectrum will only receive samples to analyse after the user has clicked on "Play" and the playback is on. Additionally, for a stereo setup, where an audio file might have 2 channels (left and right), we are only going to be analysing the **monaural** representation of that piece of audio. Each sample from the left channel will be summed with the respective sample from the right channel, and the total will be divided by 2. The result will then be pushed into the FIFO array.

After the FIFO array has been copied to the fftData array, fftData is then processed through the windowing function, and passed on to the dsp::FFT class to render an FFT frame that can be displayed as a spectrum.

In the excerpt of code below, *window* is an object from the dsp::WindowingFunction class, whereas *forwardFFT* is an object from the dsp::FFT class.

```
// Apply a windowing function to our data
window.multiplyWithWindowingTable(fftData, fftSize);

// Then render our FFT data
forwardFFT.performFrequencyOnlyForwardTransform(fftData);
```

As seen above, JUCE's dsp::FFT class provides a really useful function – **performFrequencyOnlyForwardTransform** – which transforms an array of audio samples into an array of magnitude values for each array index, or FFT bin. This means that we won't need to deal with complex numbers from an FFT or separate the phase and magnitude attributes in each FFT bin, as this function provides the magnitude values directly.

In an FFT, "the frequency resolution is the difference in frequency between each bin, and thus sets a limit on how precise the results can be. The **frequency resolution is equal to the sampling frequency divided by FFT size**" (Marsar, 2015).

$$\text{frequency Resolution} = \frac{\text{sampleRate}}{\text{fftSize}}$$

Imagining an fftSize of 2048 and a sampleRate of 44100Hz, the frequency resolution for the FFT would be approximately 21.5Hz. This means that, after the fftData array has been processed by the FFT class, its index 0 will represent the magnitude for the frequencies 0Hz (DC) to 21.5Hz, index 1 will represent 21.5Hz to 43Hz, index 3 will represent 43Hz to 64.5Hz, and so on. To determine the lowest frequency that a certain fftBin represents:

$$\text{FFTbinLowerFreq} = \text{fftBinIndex} * \frac{\text{sampleRate}}{\text{fftSize}} = \text{fftBinIndex} * \text{frequencyResolution}$$

The frequency range that a specific fftBin represents will then be *FFTbinLowerFreq* to *FFTbinLowerFreq + frequencyResolution*.

Respectively, to find the fftBin that a specific frequency is represented in:

$$\text{fftBinIndex} = \text{frequency} * \frac{\text{fftSize}}{\text{sampleRate}} = \text{frequency} * \frac{1}{\text{frequencyResolution}}$$

This means that, in an FFT output, the range of usable bins, for our case, will only be 0 to fftSize/2, as the fftSize/2 bin will represent the Nyquist frequency, which, for a sampleRate of 44100Hz, will be 22050Hz and is already above the human hearing limit.

The equations above were tested by playing various known sine waves into the application and analysing the magnitude values in specific fftData array indexes (FFT bins).

Now, we have values that can be transferred to a spectrum, as we know their frequency properties (X-axis) and their magnitude values (Y-axis). Rather than creating a plot with same number of points as fftSize/2, we can use a smaller number of points, as we only want to display the approximate spectral envelope of the sound analysed. In our case, we are going to start with a scope size of 128 points.

It is important to recognize that fftData, after being processed through the FFT class, has bins that are **linearly spaced between them**, which means that the X-axis wouldn't follow a logarithmic scale, normally used in audio spectrums. To overcome this, JUCE's "Visualise the frequencies of a signal in real time" tutorial presents the following algorithm, to assign specific fftData bins to another array called scopeData, which has 128 indexes (scope size) and will later be used for the graphical spectrum:

(Before reading the algorithm, it's worth mentioning that *jmap* and *jlimit* are both functions from JUCE. *Jmap* remaps a value from a source range to a target range – *jmap*(sourceValue, sourceRangeMin, sourceRangeMax, targetRangeMin, targetRangeMax) – whereas *jlimit* constraints a value to a specific range – *jlimit*(lowerLimit, upperLimit, valueToConstrain)).

```
auto mindB = -100.0;
auto maxdB = 0.0;

for (int i = 0; i < scopeSize; i++)
{
   auto skewedProportionX = 1.0 - std::exp(std::log(1.0 - i/scopeSize) *
0.2);

   auto fftDataIndex = jlimit(0, fftSize/2, (skewedProportionX * fftSize *
0.5) );

   auto level = jmap( jlimit(mindB, maxdB, Decibels::gainToDecibels
(fftData[fftDataIndex]) - Decibels::gainToDecibels(fftSize) ), mindB,
maxdB, 0.0, 1.0);

}

scopeData[i] = level;
```

Considering a scopeSize of 128 and an fftSize of 2048, then each scopeData index grabs its magnitude value to display from fftData as follows:

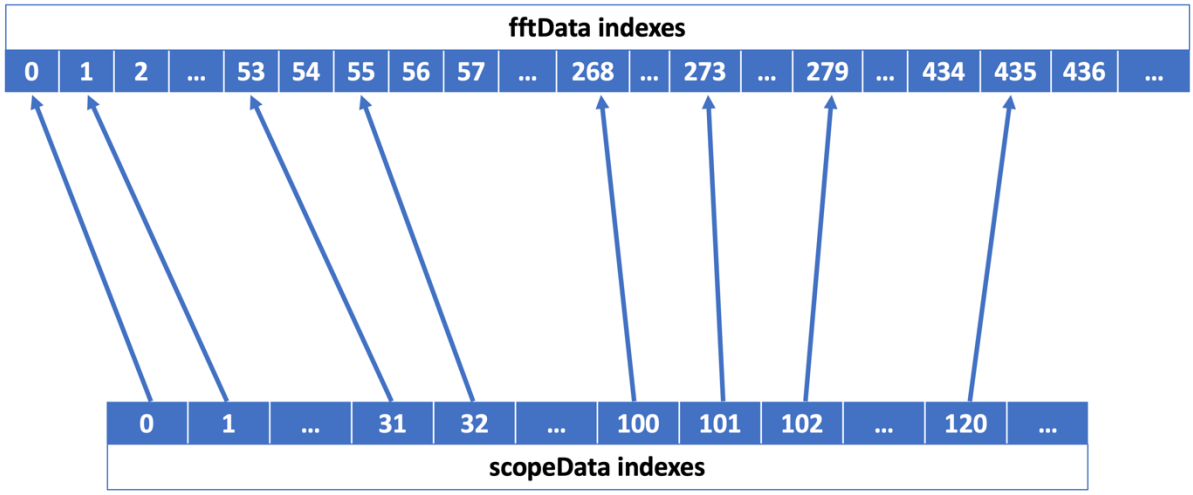

Figure 19 - Diagram showing the non-linear distribution for the spectrum plot.

This way, the spectrum plot can have a distribution that is much closer to a logarithmic scale in the X-axis, which is more suitable for this kind of application.

A fully logarithmic scale was also tried. However, it did not seem to be beneficial in this case, mostly because of the FFT having a lower resolution at low frequencies, which looked like this:

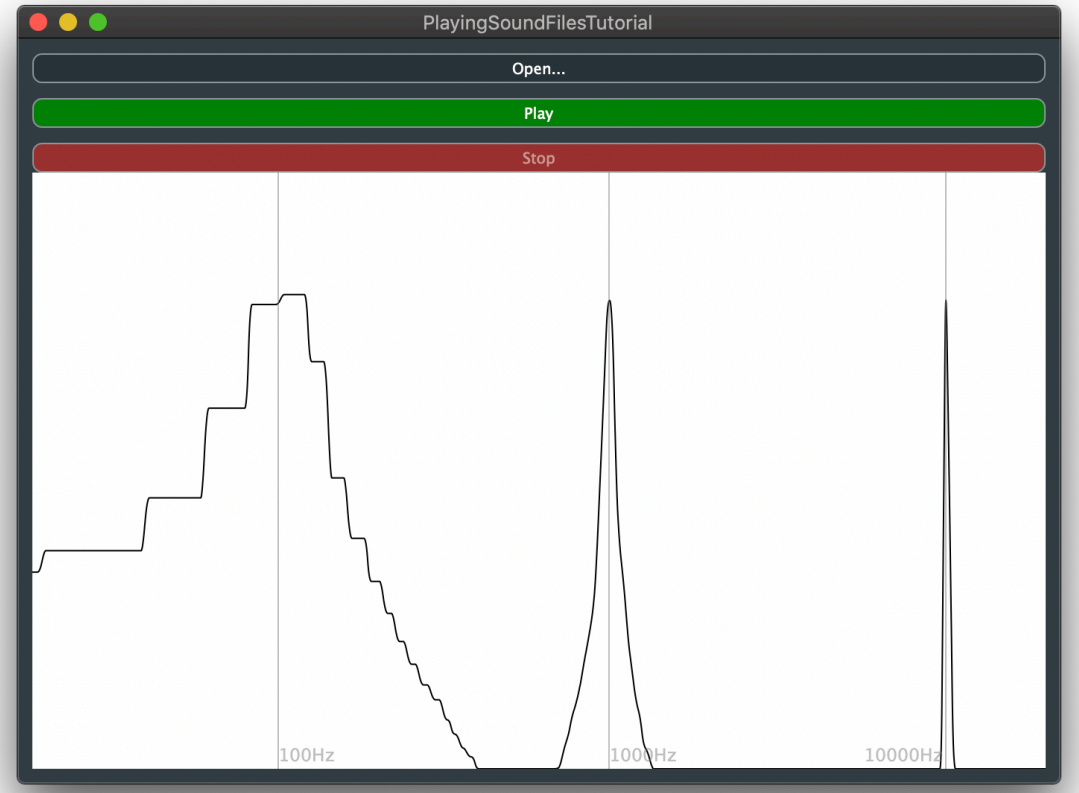

Figure 20 - Screenshot of the application using a logarithmic scale.

Using JUCE's algorithm, we can get, instead, an almost linear scale on the low frequencies, and then smoothly phase onto to a logarithmic X-axis for the higher frequencies, which will be analogous to the resolution that the FFT gives us.

Regarding the Y-axis and according to JUCE's "Visualise the frequencies of a signal in real time", the FFT class outputs magnitude values as gain. These can then be converted to decibel values by using the Decibels::gainToDecibels JUCE function, as shown in the algorithm above. This algorithm also limits the Y-axis scale to only show values between 0 dBFS and -100 dBFS.

With the X-axis and Y-axis values ready with a non-linear scale, we can now use JUCE's Path class to create a sequence of lines, that go from one point to another, in order to draw the spectrum. We can use the function Path::startNewSubPath(float startX, float start Y) to start a new path with a given position, and then use Path::lineTo(float endX, float endY) to add a line from the path's last position to a new end-point, as shown below:

```
void drawFrame(juce::Graphics& g)
{
    Path plotPath;

    auto width = getLocalBounds().getWidth();
    auto height = getLocalBounds().getHeight();

    for (int i = 0; i < scopeSize; i++)
    {
       if (i == 0)
       {
          plotPath.startNewSubPath( jmap(i, 0, scopeSize-1, 0, width),
jmap(scopeData[i], 0.0, 1.0, height, 0.0));
       }
       plotPath.lineTo (jmap(i, 0, scopeSize – 1, 0, width),
jmap(scopeData[i], 0.0, 1.0, height, 0.0f));
    }

    plotPath = plotPath.createPathWithRoundedCorners(50.0);
    g.strokePath(plotPath, juce::PathStrokeType(1.0));
}
```

As seen above, both the X-axis and Y-axis values are mapped to the height/width of the spectrum window, and then the g.strokePath function is used to draw the full frame of the spectrum, giving us a result like this:

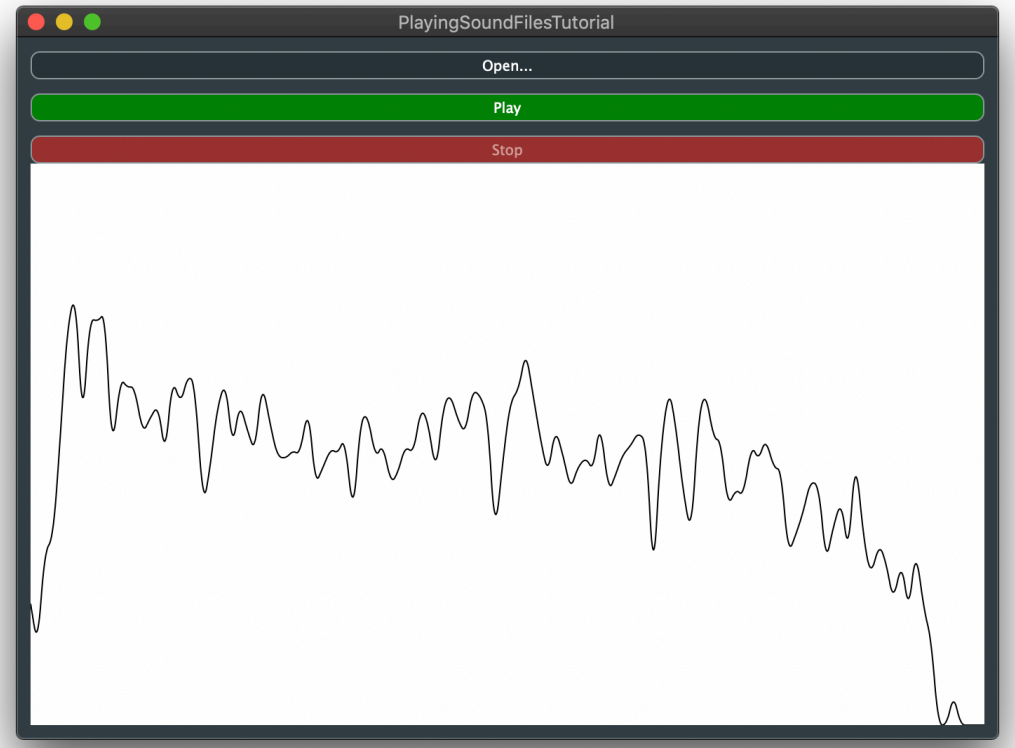

Figure 21 - Spectrum frame of a guitar excerpt.

To help with the visual analysis of the spectrum, 3 frequency lines were also added to the application, roughly specifying where 100Hz, 1000Hz and 10000Hz are displayed.

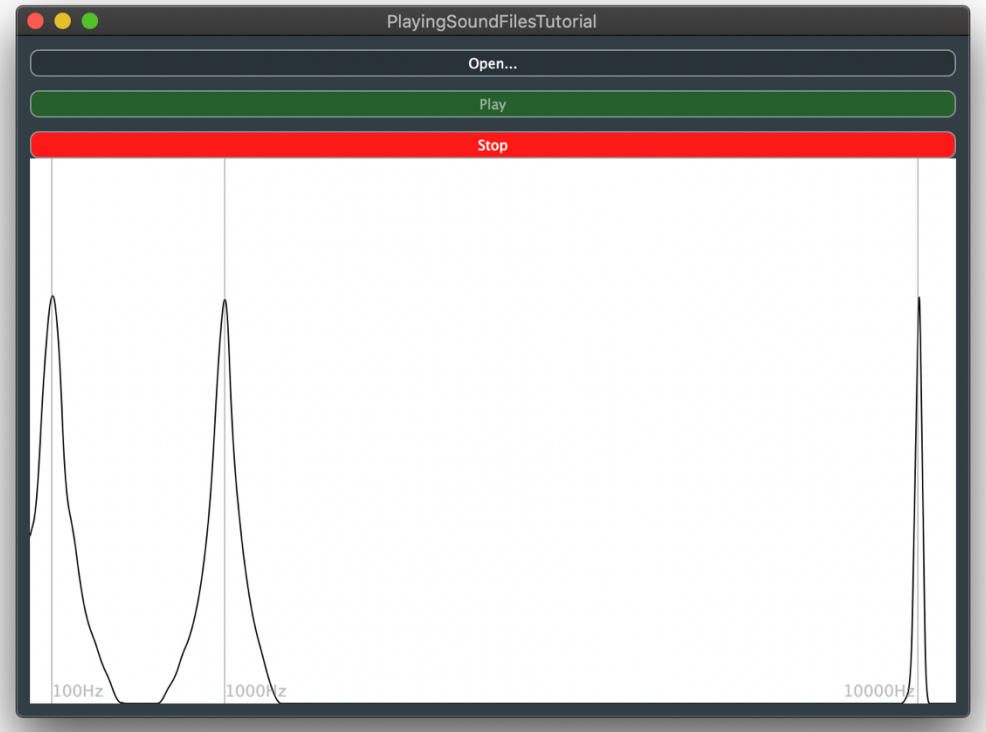

Figure 22 – Spectrum with 100Hz, 1000Hz and 10000Hz frequency lines.

As this is a real-time spectrum analyser, it needs to update the spectrum a few times per second. For this, a timer callback was set up in our Spectrum class that is triggered 30 times per second, checking if the next FFT frame is ready to be rendered. If it is ready, then it draws the frame, as shown below:

```
void timerCallback() override
{
    if (nextFFTBlockReady == true)
    {
       renderNextFrameOfSpectrum();
       nextFFTBlockReady = false;
    }
}
```

Now, we need to change the spectrum implementation to continuously average the spectrum while the file playback is on. To do this, we are going to create a variable that keeps track of the number of FFT frames rendered - *currentFrameIndex*. We can then create another array – *preAverageSum* – where all of each FFT bin magnitude values will be summed, before being divided by the number of frames rendered, calculating the average for as long as the user plays the audio file.

```
// Apply a windowing function to our data
window.multiplyWithWindowingTable(fftData, fftSize);

// Then render our FFT data
forwardFFT.performFrequencyOnlyForwardTransform(fftData);

currentFrameIndex++;
for (int i = 0; i < fftSize; i++)
{
    preAverageSum[i] += fftData[i];
    averageResults[i] = preAverageSum[i] / currentFrameIndex;
}
```

The only change that needs to be done afterwards, is for the spectrum plot (*scopeData)* to grab the amplitude (Y-axis) values from the *averageResults* array rather than directly from *fftData*, which will display the current average magnitudes.

We can then quickly test the average spectrum display using known audio examples, before moving on to the next steps.

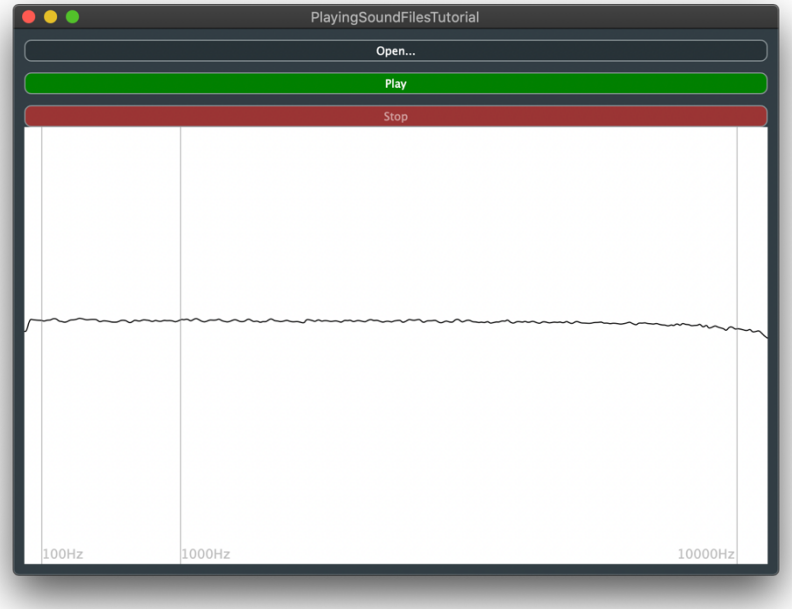

Figure 23 - Average spectrum of 30 seconds of white noise.

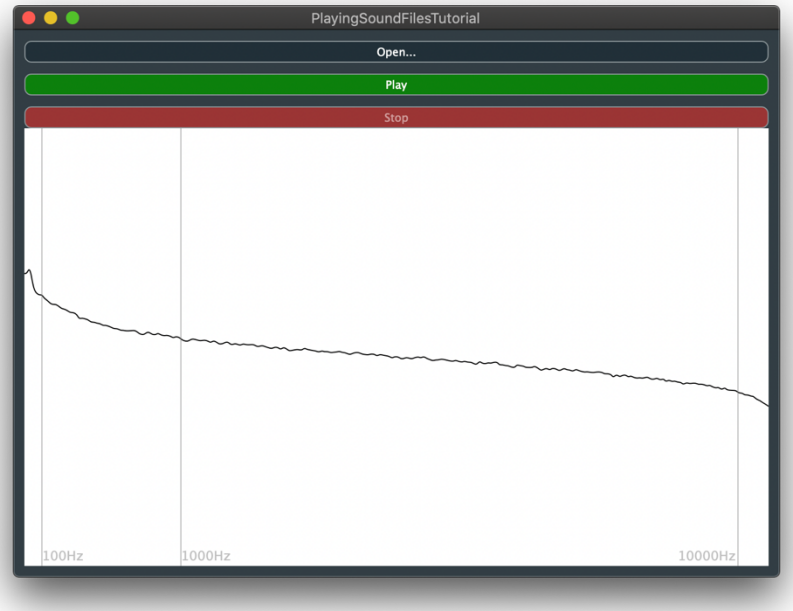

Figure 24 - Average spectrum of 30 seconds of pink noise.

5.2. SPECTRAL DIFFERENCE DISPLAY

With the average spectrum now implemented, we can simply duplicate it in our Xcode project, so that our application works with two different audio files. We will also add a third window, where we will display the spectral difference between the two files analysed. The “Target” window will be the audio spectrum that we want to approximate our “Current” spectrum to.

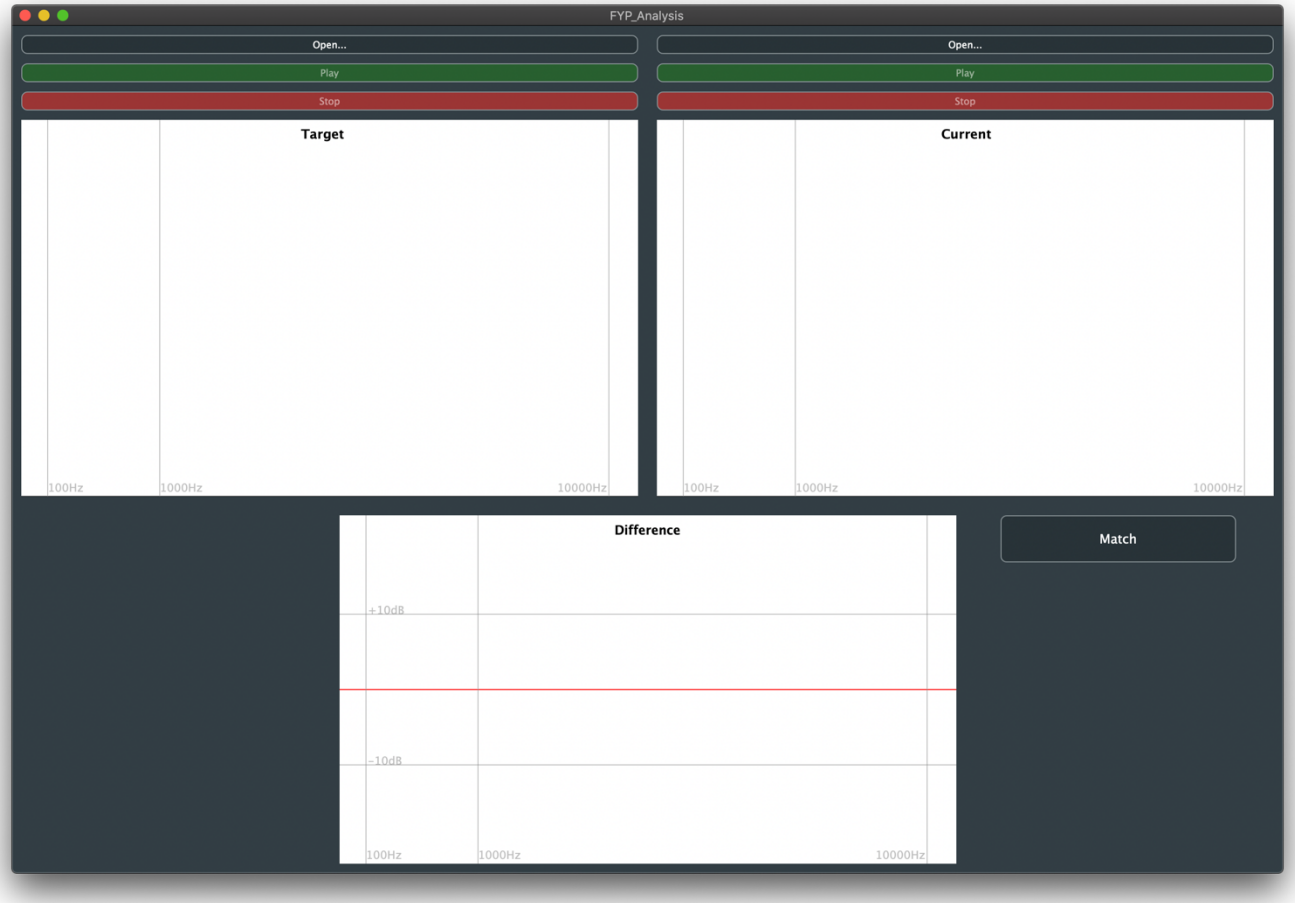

Figure 25 - Spectrum is now duplicated and difference window is displayed.

For the difference between the target and current spectrums to be calculated, when the user clicks on the “Match” button, we can simply extract the fftData array values from each spectrum, convert them from gain to decibel (as they will have the same references), and subtract the decibels for each FFT bin. We can then display the difference in a similar way to the two spectrums, by using a combination of JUCE’s Path class and the *jlimit* and *jmap* functions, as mentioned before. We won’t need to do any scaling for the level values as they will already be in decibels from the difference calculations.

```
void calculateDifferenceBetweenAverages()
{
   for (int i = 0; i < fftSize; i++)
   {
      differenceResults[i] = Decibels::gainToDecibels(averageTarget[i]) –
Decibels::gainToDecibels(averageCurrent[i]);
   }
}
```

In the excerpt of code above, *averageTarget* is a copy of the *averageResults* array from the Target spectrum, whereas *averageCurrent* is a copy of the *averageResults* array from the Current spectrum.

To test the difference calculations, one method that was used, was to create some known examples, where we would create a copy of a certain file, and process it through a known equaliser. If it this implementation works properly, the difference display should only show us the changes made by the equaliser.

Let us test this with the Target file being white noise with a +10dB boost at 100Hz, a -10dB attenuation at 1000Hz and a +10dB boost at 10000Hz, all with a Q of 1.0, using Logic's stock EQ. The Current file is the same white noise but without going through the equaliser.

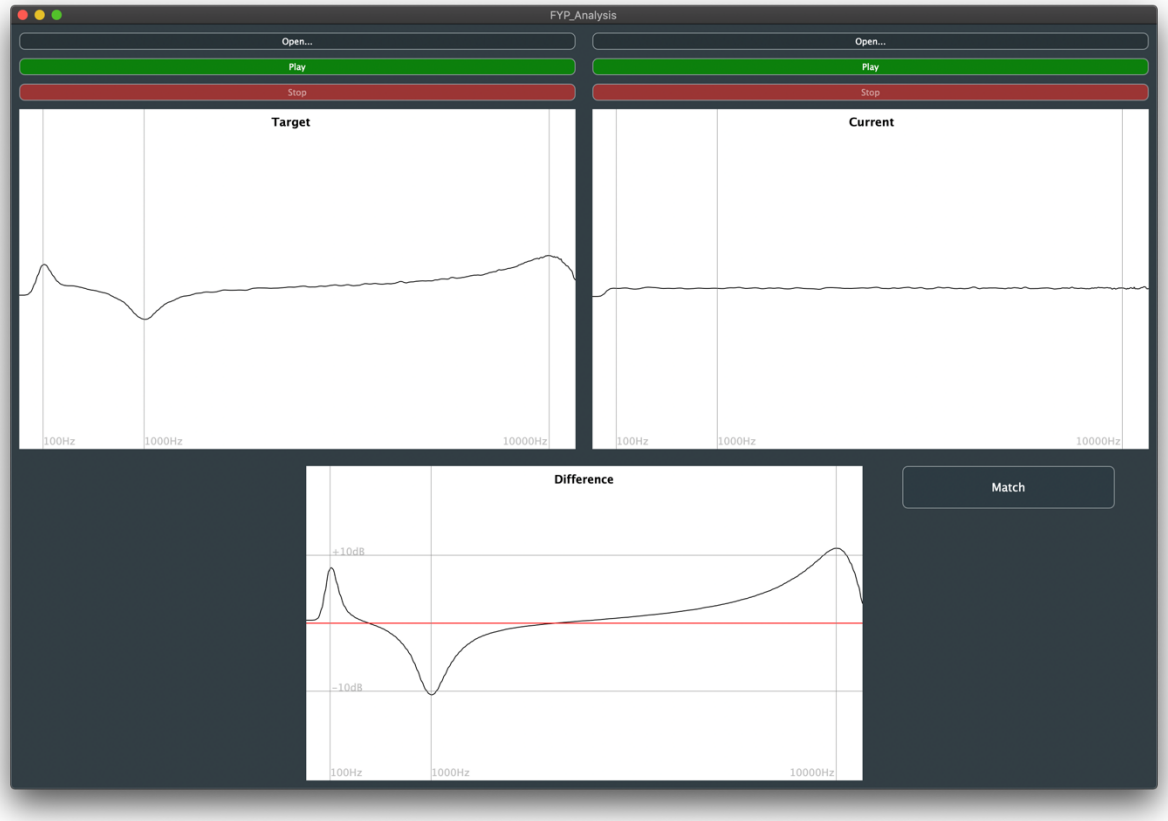

Figure 26 - Screenshot of the test mentioned above.

In the screenshot above, we can see that there is a slight deviation from the +/- 10dB mark for all the three peaks in the difference window. However, for our goal, we can consider this to be acceptable and sufficiently close.

The difference window shows us a very close approximation of what the equaliser is doing to the white noise file, essentially providing us with the frequency response of the processing we would need to apply to the Current spectrum, in order to match it to the Target.

Another example that was tested was recording two takes playing some electric guitar chords through an amplifier. However, the two takes were recorded with two different amplifiers, two distinct pickup settings in the guitar, and the pick strumming different parts of the strings:

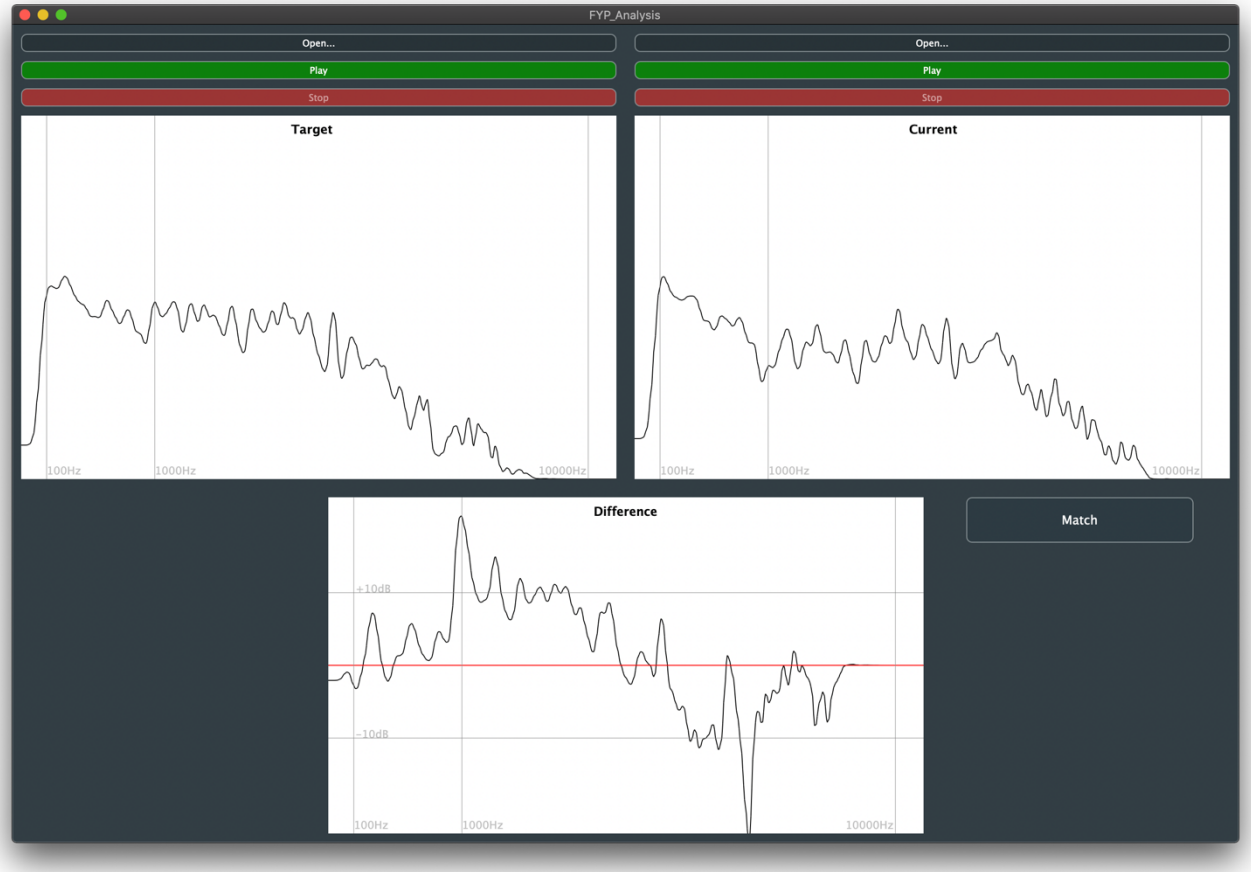

Figure 27 - Screenshot of the example specified above. "GuitarTarget" and "GuitarCurrent" files in the appendix.

We can see this gives us a much more drastic spectral difference throughout the entire spectrum.

It is important to emphasise here that our goal is not to build a dedicated analysis tool for scientific use, we simply need to use it to later inform the filters implementation. The important goal for us is to have the application display the rough spectral envelope of the signals, even if the horizontal scale is not exactly like the one found in other spectrums. This will not affect the quality of our data as the graphical user interface is separate from the audio process.

6. FROM APP TO PLUG-IN

Now that we have implemented the two long-term average spectrums, as well as their comparison display and calculations, we can start working on porting the application into an audio plug-in.

While it isn't worth going through the whole process, there are a few differences worth mentioning when working with JUCE plug-ins compared to applications:

- A plug-in needs to follow the audio settings of the host (DAW), whereas an application can have a dedicated sample rate and buffer size;
- The audio processing needs to be separate and independent from the GUI rendering – this was achieved by creating separate classes for each purpose, where the UI class keeps a pointer to its audio counterpart in order to control or display it (SpectrumAudio/SpectrumUI, DifferenceSpectrumAudio/DifferenceSpectrumUI). JUCE also helps with that, as it creates, by default in plug-in projects, an **AudioProcessor** class to handle audio and an **AudioProcessorEditor** class to handle the user interface;
- DAWs normally allow for automation of some plug-in parameters, which needs to be set up manually in the plug-in's code;
- Relevant parameters and values need to be saved (but not made automatable) when the DAW session is closed and recalled when it is opened again, which also needs to be set up manually.

To address the last two bullet points above, two JUCE tutorials from their website were essential: "*Adding plug-in parameters"* and *"Saving and loading your plug-in state",* in order to save and recall relevant parameters/values, using conversion to and from XML.

In preparation for the next stage of the project, where filters will be set up to match the Current audio to the Target, a slider was created and made automatable to later control the intensity of the matching. Apart from this, we will also need to make sure that the values from the Current and Target spectrums, as well as the Difference results will always be saved and recalled, although not automatable by the user.

To save the plug-in values (even non-automatable ones), JUCE's **ValueTree** class is being used, which can hold free-form data and easily convert it into XML. This saving process will happen whenever the user stops analysing an audio file or moves the intensity slider.
Refer to the "SavingAndRecallingPluginState" video in the VideoExamples folder of the appendix for a demonstration of this working.

To help with the rest of this project's development, JUCE provides an application called **AudioPluginHost**, which works as a host for audio plug-ins, allowing us to continue prototyping and debugging without having to always open and close a DAW whenever we change the code. It also allows us to connect different plug-ins and, as such, a rudimentary audio file player was inserted before our plug-in, to simulate playback from a DAW.

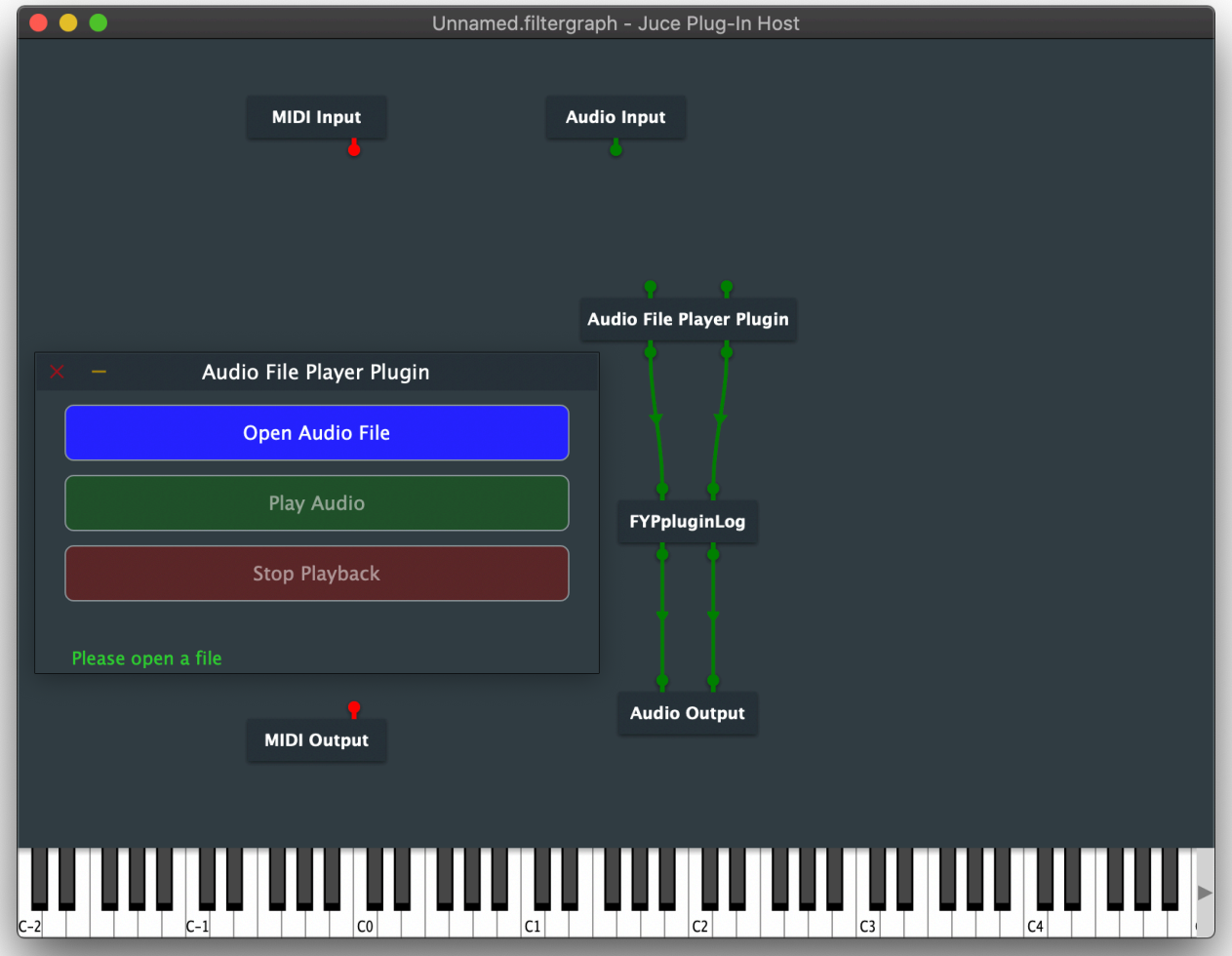

Figure 28 - Screenshot of JUCE's AudioPluginHost and the Audio File Player Plugin.

Once the porting process is finished, we can then load our software into a DAW. For this case, we will specify, in the *Projucer,* that we want AU and VST3 formats of our plug-in, which will be compatible with most DAWs currently. After compiling and building our ported code in Xcode, we can now open our software in Logic Pro and Reaper.

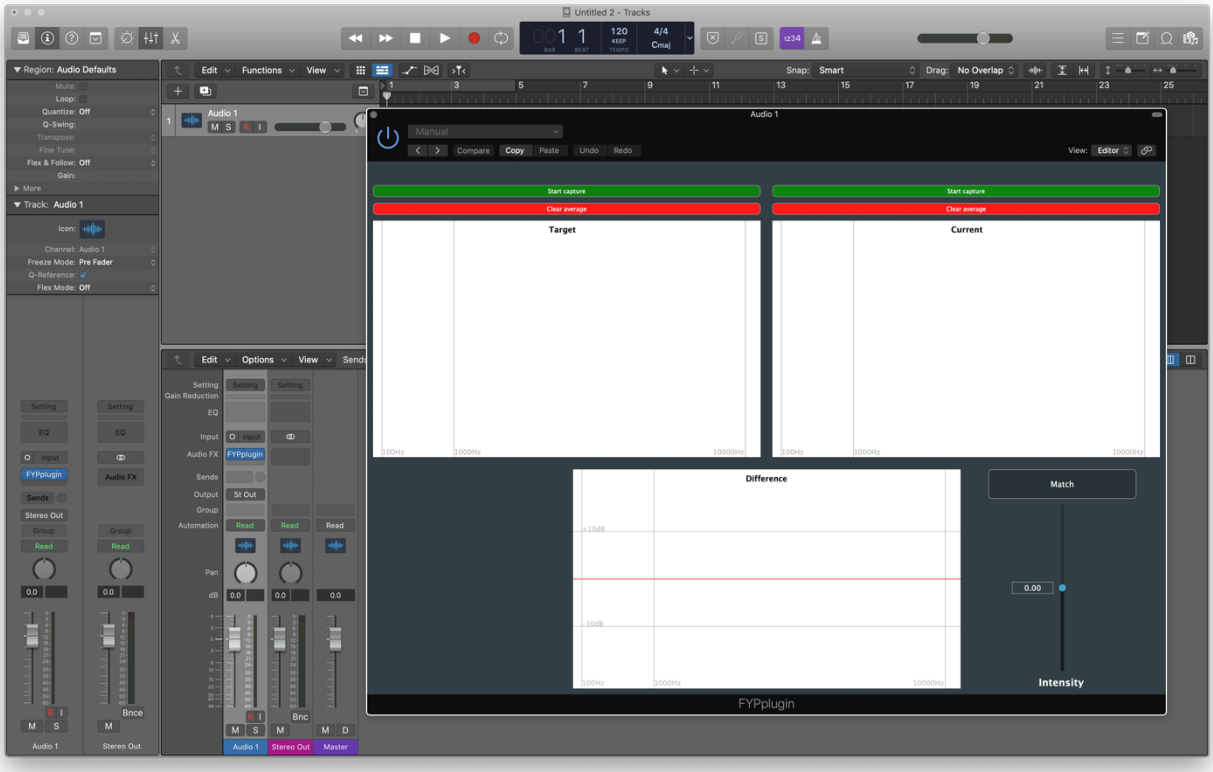

Figure 29 - Screenshot of the FYP plug-in open in Logic Pro.

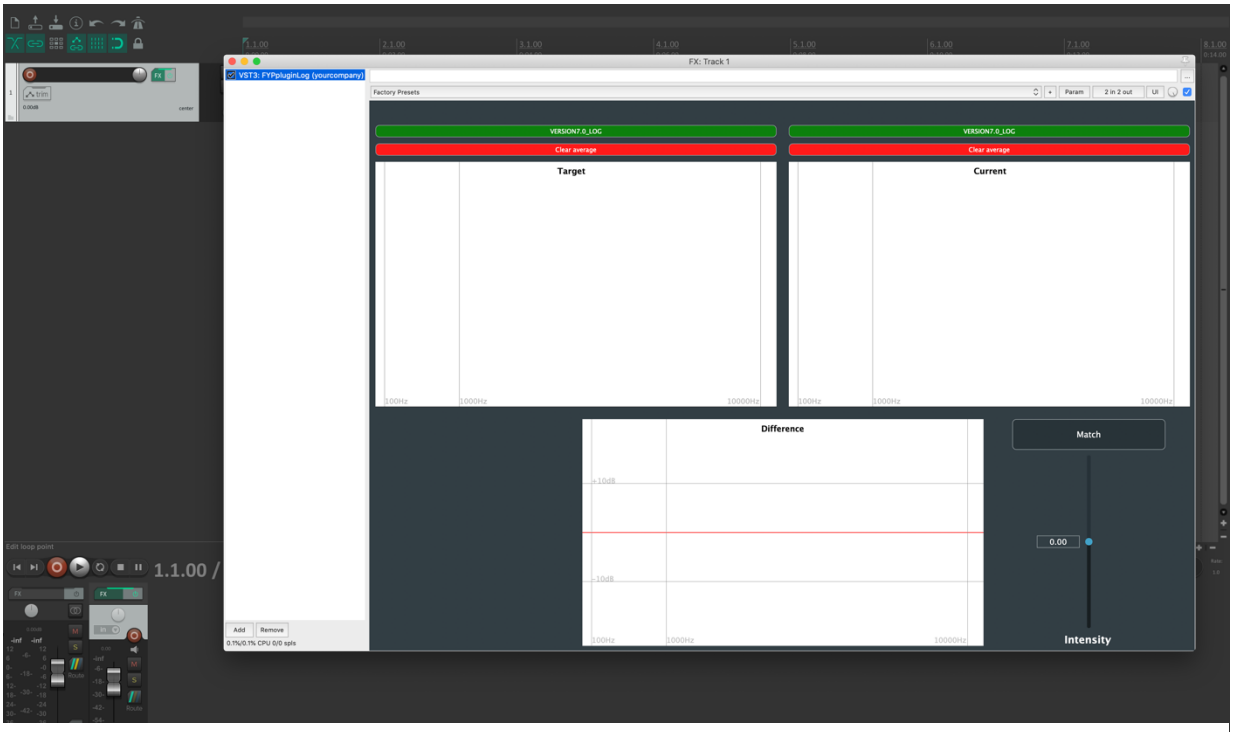

Figure 30 - Screenshot of the FYP plugin open in Reaper.

7. IMPLEMENTATION – FILTERS SETUP

Now that we have the values that we need from the analysis and comparison between the two different spectrums, as well as a plug-in version of our software, we can start implementing the audio processing to match the Current audio to the Target audio.

After weeks of research on filter design (such as Infinite Impulse Response vs Finite Impulse Response filters, linear-phase, minimum-phase, coefficients, convolution, etc.) it was recognisable that this was a deeply technical area, with lots of advanced mathematics and concepts, which, going back to the project proposal, are outside of the scope of this project. For this reason, we will be using JUCE's **dsp::IIR::Filter** class, as they provide a high-level approach to setting up various types of filters, such as bandpass, notch, peak/bell, high/low pass, etc.

Along with the **dsp::IIR::Coefficients** structure, we can set up multiple filters just by specifying their centre frequency, the sample rate of the system, their Q factor and, for some filter types, their gain.

Although a filter bank approach using bandpass filters would be a suitable idea, after some research on the dsp::IIR::Coefficients structure along with some experiments using that type of filter in JUCE, it was concluded that it would be better to use an array of peak/bell filters instead. This would provide an easier method of controlling the gain of each specific filter, as well as making sure that any filters that are not being used (gain factor of 1.0) are not altering the frequency or phase response in any way.

To test these approaches, two frequency response analysers were used: Bertom's *EQ Curve Analyzer*, and Pajczur's *pajEQanalyser*. Both allow us to "investigate frequency magnitude and phase shift caused by EQ or any other plug-in" (Pajczur, 2021). To use them, we need to insert their impulse generator before our plug-in, and the analyser after, which displays the frequency response of our own plug-in.

In the figure below, we can see the two frequency response analysers (on the top) displaying the frequency response of iZotope's Neutron 2 EQ (on the bottom). This is tested inside JUCE's AudioPluginHost:

Figure 31 - Screenshot of the two frequency response analysers (on the top), analysing iZotope's Neutron 2 EQ (on the bottom), within the AudioPluginHost.

Refer to the “FrequencyResponseAnalysers” video in the VideoExamples folder of the appendix to see the frequency analysers being set up and tested.

Because this stage of the project involved a considerable amount of testing, prototyping, and experimentation with different number of filters, different Q factors, different relations to the analysis, etc., only the most significant approaches will be mentioned here, as well as how they have led to the final implementation. Additionally, the majority of testing will be detailed only in the last implementation, due to the word limit of this report.

7.1. LINEAR DISTRIBUTION OF FILTERS

The first approach that was tried was to directly relate the results from the difference spectrum to the gains, Q factors and centre frequencies of the filters. Filter 1 will follow the difference results from FFT bin 1, Filter 2 follows FFT bin 2, and so on. As mentioned before, only the first half of the FFT bins would be important for the filter implementation, representing 0 Hz (DC) up until the Nyquist frequency (sampling frequency divided by two).

This means that, with an FFT size of 2048, we would have 1024 filters referencing 1024 bins. Being that we know the FFT's frequency resolution, the frequency range that each bin represents, and their difference values from the analysis, we could transfer this information directly into the filters' implementation.

This approach was then to create an array of *fftSize/2* (1024, in this case) filters, that would follow the values from their respective FFT bins. (In reality, in a stereo system there would need to be 2048 filters, as each filter can only process one channel of audio, even if using the exact same settings on the left and right channels. However, because JUCE lets us duplicate mono processors using the **dsp::ProcessorDuplicator** structure, for the simplicity of this report we will mention each filter as being able to process the left and right channels.)

Regarding the centre frequency for each filter, as we know how to find the range of frequencies that each FFT bin represents, we can calculate the following:

$$\text{centreFrequency} = \text{FFTbinLowerFrequency} + \frac{\text{frequencyResolution}}{2}$$

$$\text{centreFrequency} = (\text{fftBinIndex} * \text{frequencyResolution}) + \frac{\text{frequencyResolution}}{2}$$

In this way, the centreFrequency is essentially the median value of the range of frequencies that its specific FFT bin represents.

To set up an array of bell/peak filters, we also need to specify their Q factor values. The Q (quality factor) of a filter "may be defined as the *resonance frequency* divided by the resonator bandwidth". "The *resonance* frequency is typically defined as the frequency at which the peak gain occurs. The *bandwidth* is typically defined as the 3dB-bandwidth, i.e., the bandwidth between the -3dB points (half-power-gain points) straddling the resonance frequency" (Smith, 2007). We can then choose the bandwidth to be the distance (in Hz) between the frequencies that each *fftBin* represents (which will be our *FFTresolution*), whereas the centre frequency will be obtained as detailed above.

$$\text{Q factor} = \frac{\text{centreFrequency}}{\text{fftBinHigherFreq} - \text{fftBinLowerFreq}} = \frac{\text{centreFrequency}}{\text{FFTresolution}}$$

Regarding the gain for each filter, we get it from the difference results of its specific FFT bin. As these results will be in decibels, we simply need to convert them again into gain values using JUCE's **Decibels::decibelsToGain** function.

We can then set up the entirety of the filters as follows:

```
float freqRes = sampleRate / fftSize;
int numberOfFilters = fftSize/2;

for (int i = 0; i < numberOfFilters; i++)
{
   centreFrequency[i] = freqRes * i + (freqRes/2.0);
   Qvalue[i] = centreFrequency / freqRes;
   gain[i] = Decibels::decibelsToGain(differenceResults[i] *
intensitySlider);

   *peakFilterArray[i].state =
*dsp::IIR::Coefficients<float>::makePeakFilter(sampleRate,
centreFrequency[i], Qvalue[i], gain[i]);
}
```

In order to test this approach, we can use the audio files “GuitarTarget” and “GuitarCurrent”, found in the ReportExamples folder of the appendix. Apart from listening to the matched result, we can also compare the Difference spectrum to the frequency response of our plug-in, using Bertom’s *EQ Curve Analyser.*

If we analyse the two files, we get the following results:

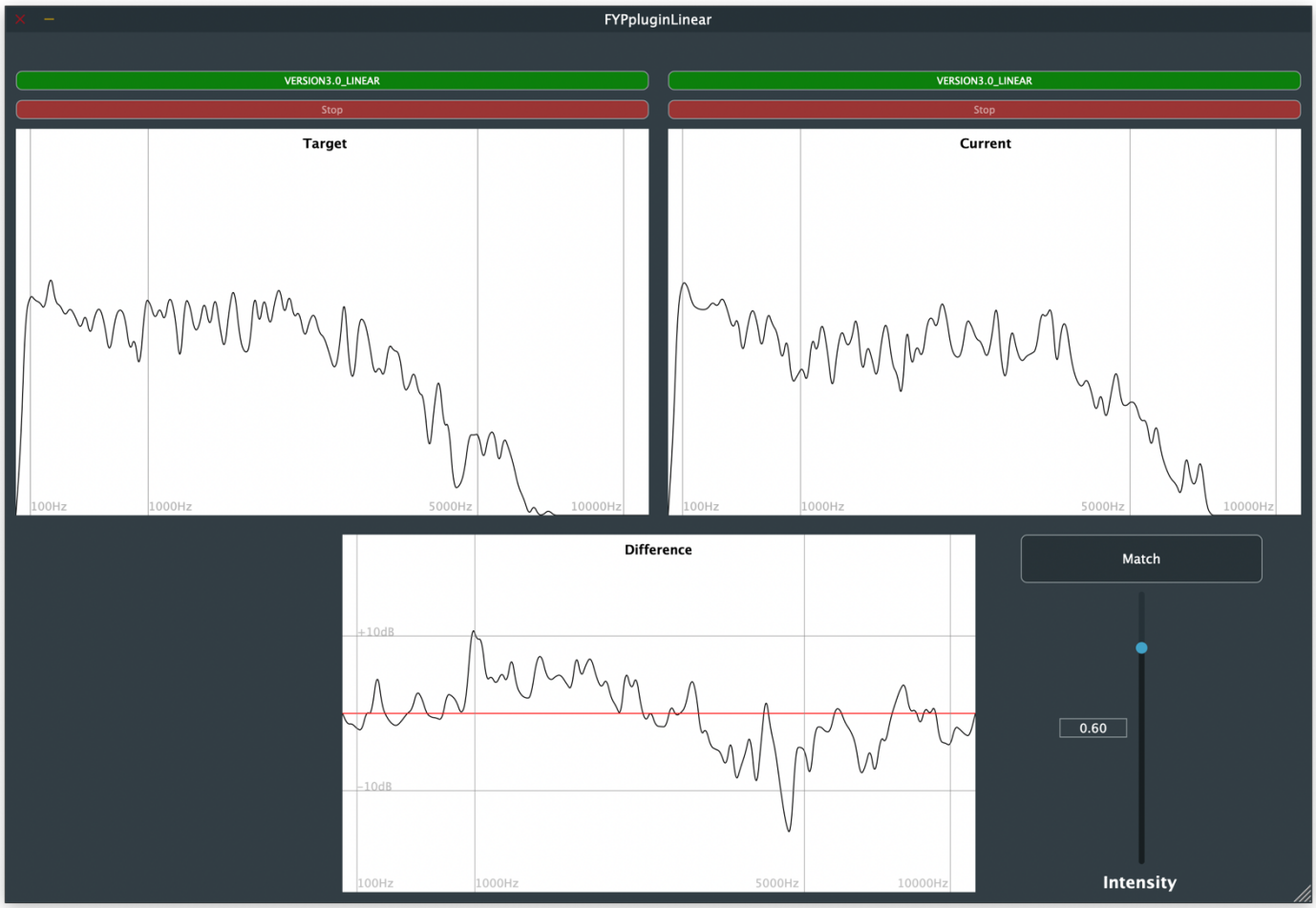

Figure 29 - Screenshot of the comparison between two guitars playing the same phrase but with different amps and different pickup settings.

Focusing on the difference display, the figure below should then be roughly the frequency response that our plug-in needs to have, to match the Current to the Target audio:

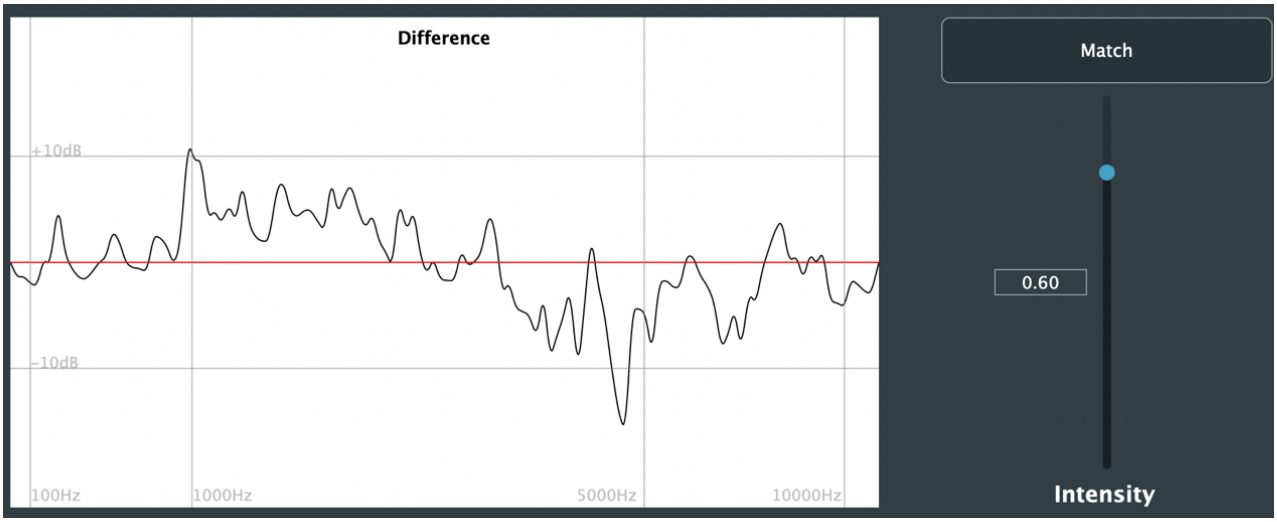

Figure 30 - Screenshot of our difference results from the example in figure 29.

After listening to the matched Current, it was decided that an intensity value of 0.6 (60%) was a good balance between sounding close the Target and taming resonances from the filters, for this example.

If we analyse our filters' frequency response with Bertom's *EQ Curve Analyser,* we get the following analysis:

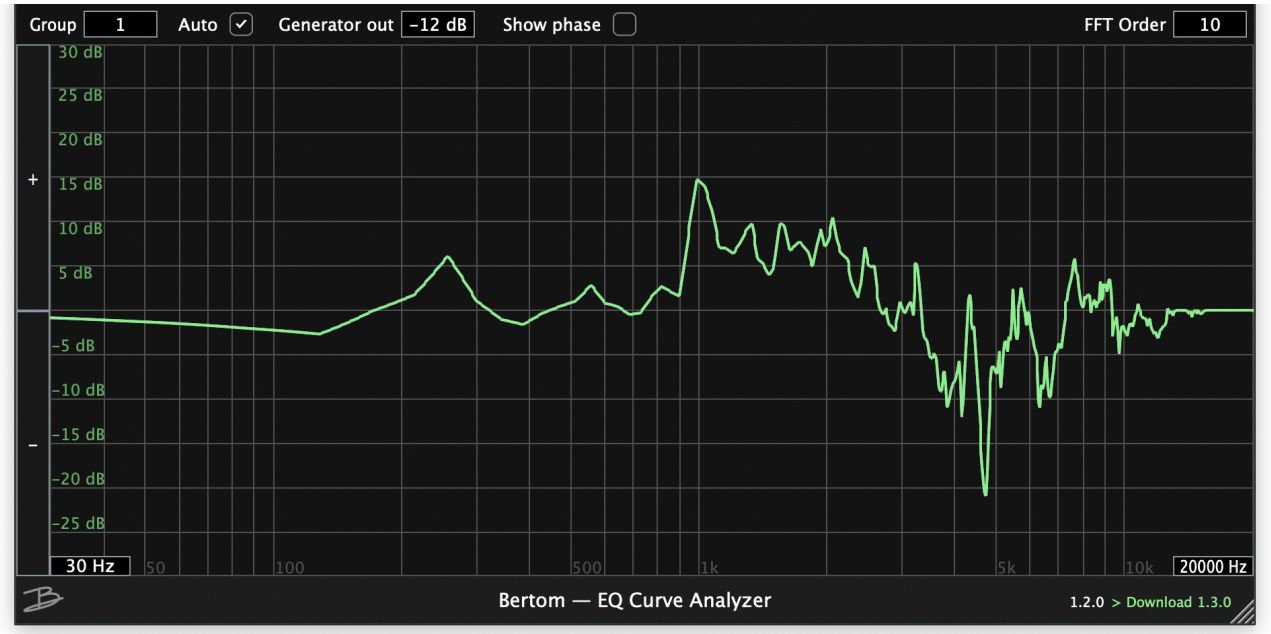

Figure 31 – Screenshot of the frequency response of our plug-in, using the guitar example with the linear filters approach.

Firstly, it's important to emphasise that the X-axis of our analysis and the one from Bertom's *EQ Curve Analyzer* are different, as Bertom's is purely logarithmic, while ours follows the distribution from JUCE's algorithm mentioned in the analysis chapter.

We can describe the result as a good starting point, as we can see the frequency response following our analysis very closely – we can identify the three peaks in the 100Hz to 1000Hz area, one peak at 1000Hz and another one roughly below 5000Hz, which are similar in both figures.

One discrepancy that we can note, however, is the decibel values between our analysis and our frequency response. This might be due to the fact that adjacent filters can be adding gain to each other, and, consequentially, the overall gain of the filter setup might be slightly increased – in our analysis, the 1000Hz peak is roughly situated in the Y-axis at +10dB, whereas our frequency response shows a gain of +15dB for that same peak. However, as the user will be able to manually change the intensity of the filters, this won't be a relevant concern.

Regarding some advantages and disadvantages of this linear filter setup approach, it is possible to say that, being able for the frequency response to follow the exact frequency resolution of our FFT results is a benefit, specifically in the lower frequencies. However, it does come at a cost, as it involves a high number of filters, particularly to cover the high frequencies area.

To minimize the number of filters, we can experiment with the following approach:

7.2. LOGARITHMIC DISTRIBUTION OF FILTERS

The second approach that was tried was distributing the filters logarithmically. This is closely related to the 31-band graphic EQ 1/3 octave representation, which “is strongly linked to the perception of sound by a human ear”, and “it allows a compression of the amount of information” (Ma, Reiss and Black, 2013). This compression of information lets us have a lower number of filters needed compared to the previous linear approach, specifically in the higher frequencies area.

It is important to have in mind that octaves are specifically defined as a doubling or halving of frequency, and, in a logarithmic scale, equal distances mean equal ratios between the values (as opposed to equal differences, in a linear scale). Knowing this, if we multiply any frequency by 2, we can find out the frequency that is one octave higher than the one we started with.

In Western music, it is very common to divide each octave into 12 different subdivisions (semitones), maintaining the same ratio between adjacent frequencies (Dobrian, 2019). We can then assume that there is going to be a ratio that, if multiplied with any frequency, will give us the frequency that is one semitone higher than the starting one. If we then keep multiplying the result by the ratio, once we have done it 12 times in total, we will get to the frequency that is one octave higher than the one we started with. To find that ratio, we will use 440Hz as a starting point, and 880Hz being one octave higher than 440Hz:

$$440 \times ratio^{12} = 880$$

$$\Leftrightarrow ratio^{12} = \frac{880}{440}$$

$$\Leftrightarrow ratio^{12} = 2$$

$$\Leftrightarrow ratio = \sqrt[12]{2} = 2^{\frac{1}{12}}$$

This also makes it simple to divide an octave into any other number of subdivisions, as we can assume that:

$$ratio = 2^{\frac{1}{numberOfSubdivisionsPerOctave}}$$

We can directly use this to set up the centre frequencies for our array of filters. Starting with the 1/3 octave approach, we can begin with 20Hz being the centre frequency of our first filter. Our *ratio* will then be $2^{\frac{1}{3}}$ and, in order cover the entire frequency spectrum (20Hz – 20kHz), we will need a total of 31 peak filters.

```
numberOfFilters = 31;
centreFrequency[0] = 20;
for (int i = 1; i < numberOfFilters; i++)
{
   centreFrequency[i] = centreFrequency[i-1] * std::pow(2.0, 1.0/3.0);
}
```

Regarding the Q factor values, every filter will use the same Q as the ratio between all of their centre frequencies is going to be constant. As such, we will calculate it by selecting three adjacent filter frequencies and replacing them in the formula that was presented before:

lowerFreq will be the median between the $centreFrequency_0$ and $centreFrequency_1$, while *higherFreq* will be the median between $centreFrequency_1$ and $centreFrequency_2$.

$centreFrequency_0 = 20Hz;\ centreFrequency_1 = 25.2Hz;\ centreFrequency_2 = 31.75Hz;$

$$\text{Q factor} = \frac{\text{centreFrequency}}{\text{higherFreq}-\text{lowerFreq}} = \frac{25.2}{28.475 - 22.6} \approx \mathbf{4.3}$$

The same Q value would be obtained with any other three adjacent filter frequencies.

Lastly, to set up the gain values for each filter, we can get them from the difference results' FFT bin where their centre frequencies are represented in, as follows:

```
numberOfFilters = 31;
double gain[numberOfFilters];
int fftBinToLookFor[numberOfFilters];

for (int i = 0; i < numberOfFilters; i++)
{
   fftBinToLookFor[i] = centreFrequency[i] * fftSize / sampleRate;
   gain[i] = Decibels::decibelsToGain(differenceResults[fftBinToLookFor[i]]
* intensitySlider);

   *peakFilterArray[i].state =
*dsp::IIR::Coefficients<float>::makePeakFilter(sampleRate,
centreFrequency[i], 4.3, gain[i]);
}
```

We can then test this 1/3 octave approach in the same way that we have tested the linear filters setup, by using the GuitarTarget and GuitarCurrent files to compare our difference analysis to our plug-in's frequency response. Using the same examples, we can assume the analysis step of the process to be similar to before, as shown below:

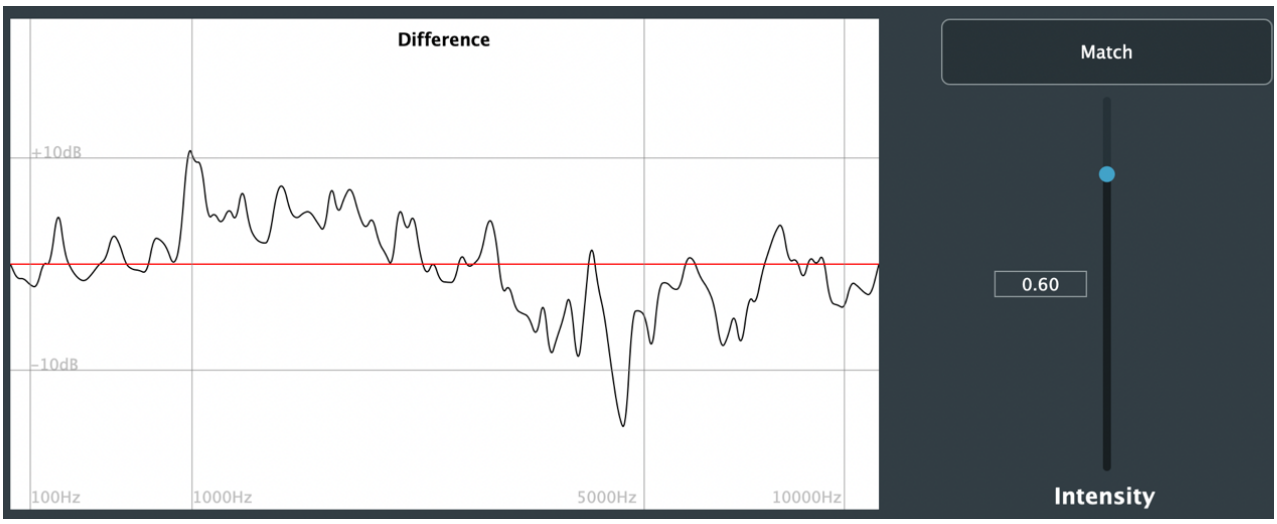

Figure 32 - Screenshot of our difference results from comparing the average spectrums of GuitarTarget and GuitarCurrent

The frequency response now looks like this:

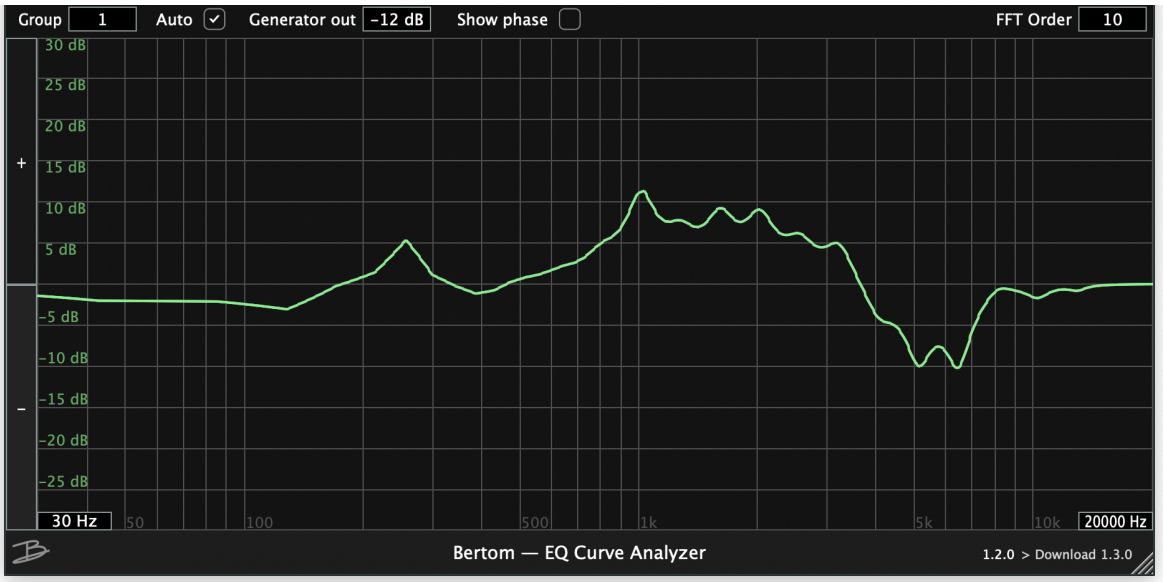

Figure 32 - Frequency response from the guitar example, using the 1/3 octave approach.

The first thing we can notice is that the frequency response now looks like a very smoothed version of the difference spectrum, without precisely detailing the peaks and valleys, as the linear setup did. To improve this, we can try diving each octave into more subdivisions.

If we then alter the number of subdivisions per octave to 12, we should need a total of 121 filters to cover the human hearing range, each with a Q factor value of 17.3. This would give us the following frequency response:

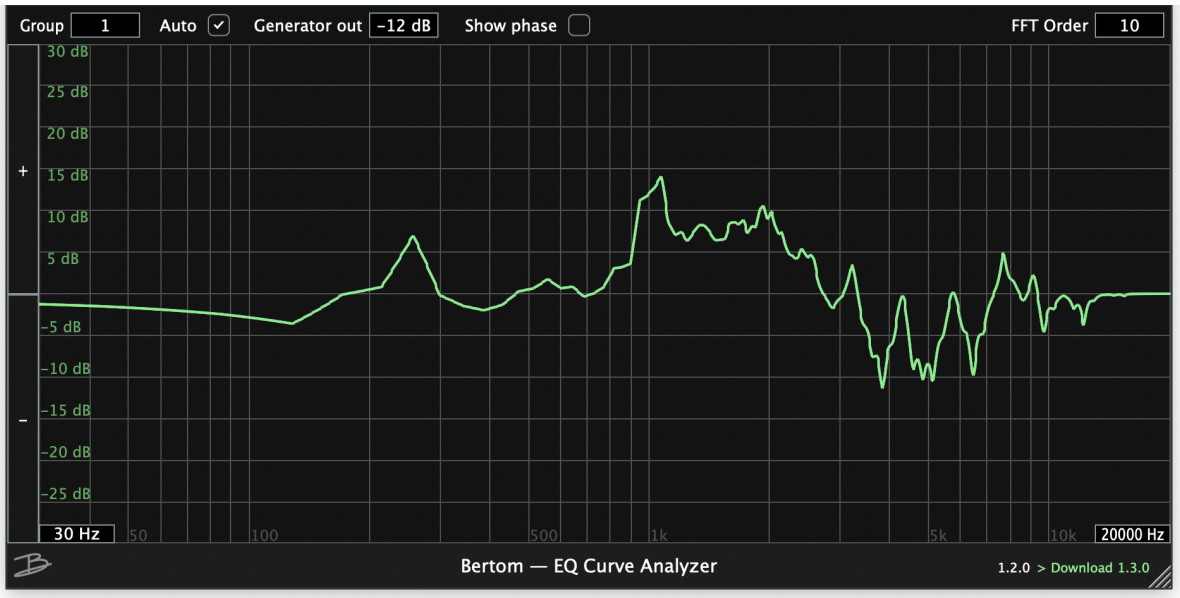

Figure 33 - Frequency response from the guitar example, using the 1/12 octave approach.

Using 24 subdivisions per octave, we would need 241 filters, all with a Q factor of 34.6:

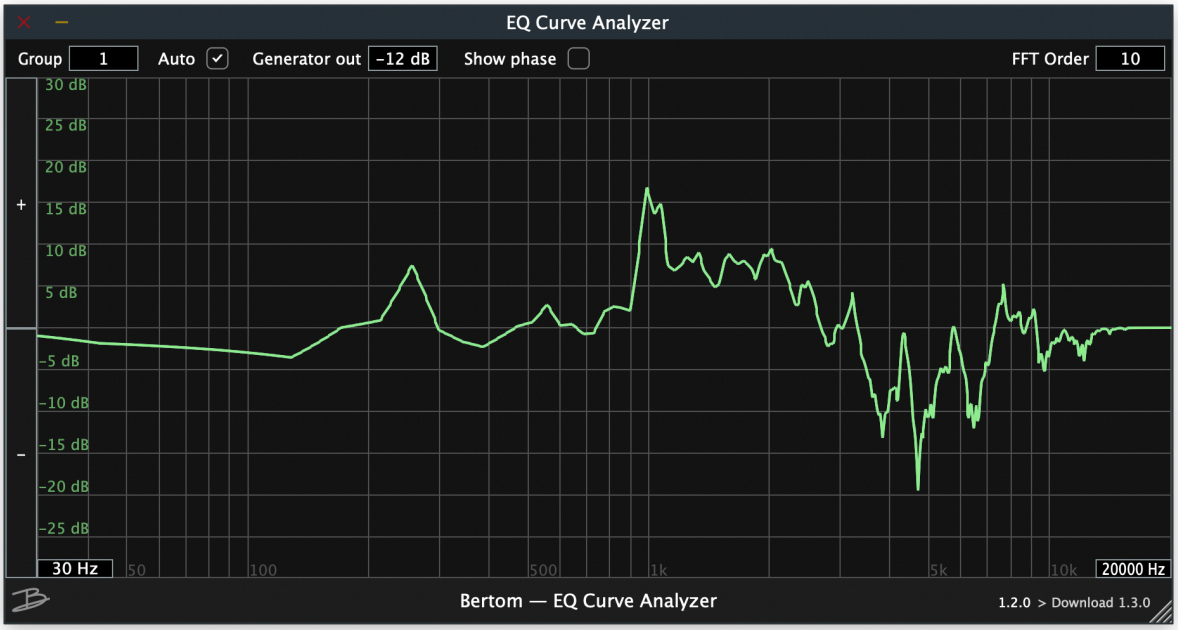

Figure 34 - Frequency response from the guitar example, using the 1/24 octave approach.

Lastly, 48 subdivisions per octave were also tried, involving a total of 480 filters, each with a Q factor of 69.2:

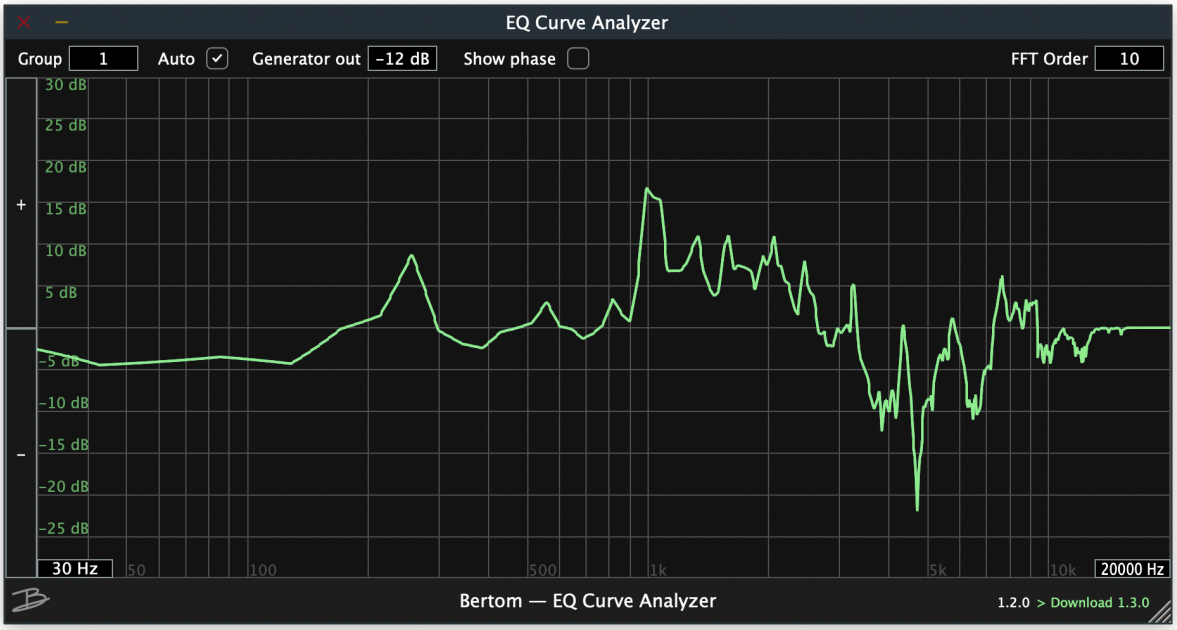

Figure 35 - Frequency response from the guitar example, using the 1/48 octave approach.

As we can see, we can still get the detail very close to the one that we had on the linear approach. Even if using a considerable number of subdivisions per octave, such as 48, we would still need a significantly smaller number of filters compared to the linear implementation. Additionally, this implementation is not directly dependent on the FFT size used to calculate the spectral differences between the two files.

However, there is another problem that arises from this implementation. For the 48 subdivisions per octave, let's detail the centre frequencies for the first 12 filters, as well as the FFT bin that each filter is getting its gain value from (considering an FFT size of 2048), as explained at the start of this chapter:

Filter Index	**Filters' centre frequency - Hz**	***FFTbinToLookFor***
0	20	0
1	20.29	0
2	20.58	0
3	20.88	0
4	21.19	0
5	21.49	0
6	21.81	1
7	22.13	1
8	22.45	1
9	22.77	1
10	23.10	1
11	23.44	1

As shown above, a lot of filters will be referencing the same FFT bin for the lower frequencies, since the FFT distributes its magnitude values in a linear way. This will continue roughly up until the 1000Hz area and, as such, we have simply shifted the problem that we had in the linear approach (where a lot of filters in the higher frequencies were unnecessary and inefficient) to the lower frequencies area, where a lot of filters will be redundant.

7.3. HYBRID DISTRIBUTION OF FILTERS

To overcome the problems mentioned in the linear and logarithmic implementations, the final implementation simply combines the two filter distributions in the same setup. Up until roughly the 1000Hz area, the filters are distributed linearly, with a specific Q value calculated for each of them, whereas from 1000Hz-20000Hz, the filters follow the logarithmic distribution with 48 subdivisions per octave and a Q factor of 69.2.

This approach is able to then follow the low-frequency resolution of our FFT size (2048), while still being able to compress the amount of information that exists in the higher-frequencies. This way, an even smaller number of filters needs to be used, totalling 256 filters. The first 47 will be distributed linearly, while the other 209 will follow the logarithmic distribution.

Using the same audio examples as in the other tests above, this implementation gives us the following frequency response:

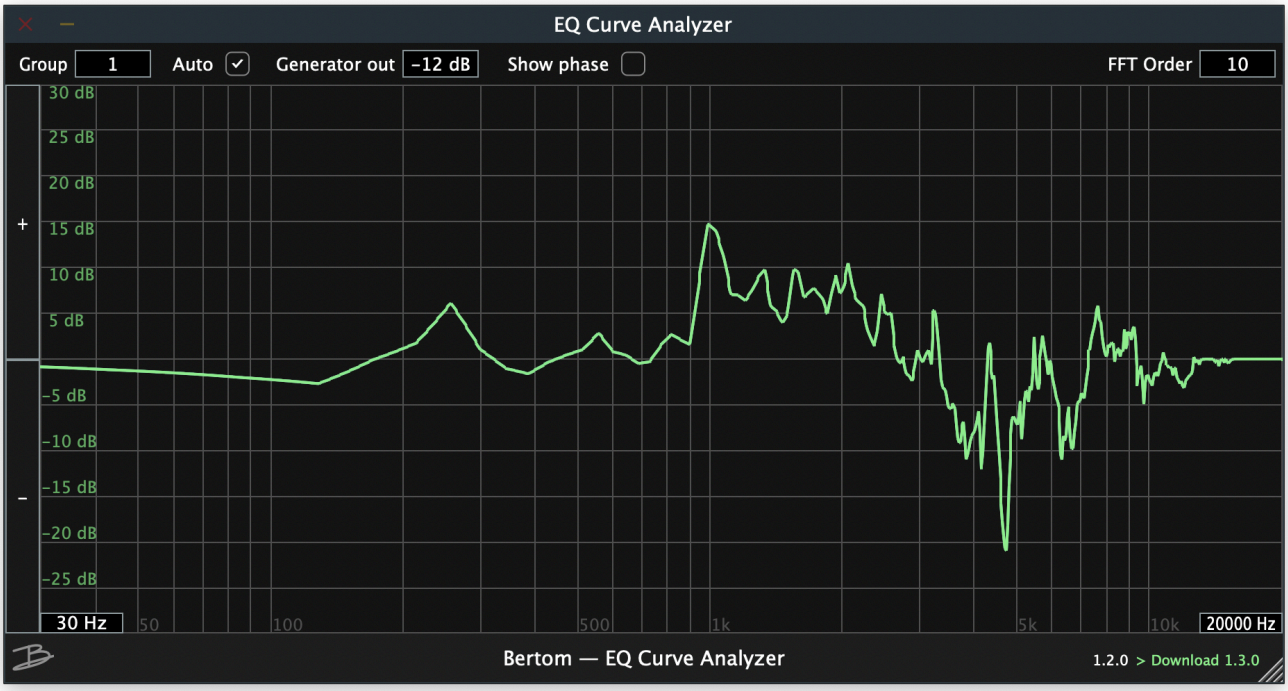

Figure 36 – Frequency response of the same guitar example as the last figures, but following the linear and logarithmic approaches combined.

We can see that the frequency response closely follows our spectral difference analysis, even with a much lower number of filters than the first two implementations mentioned:

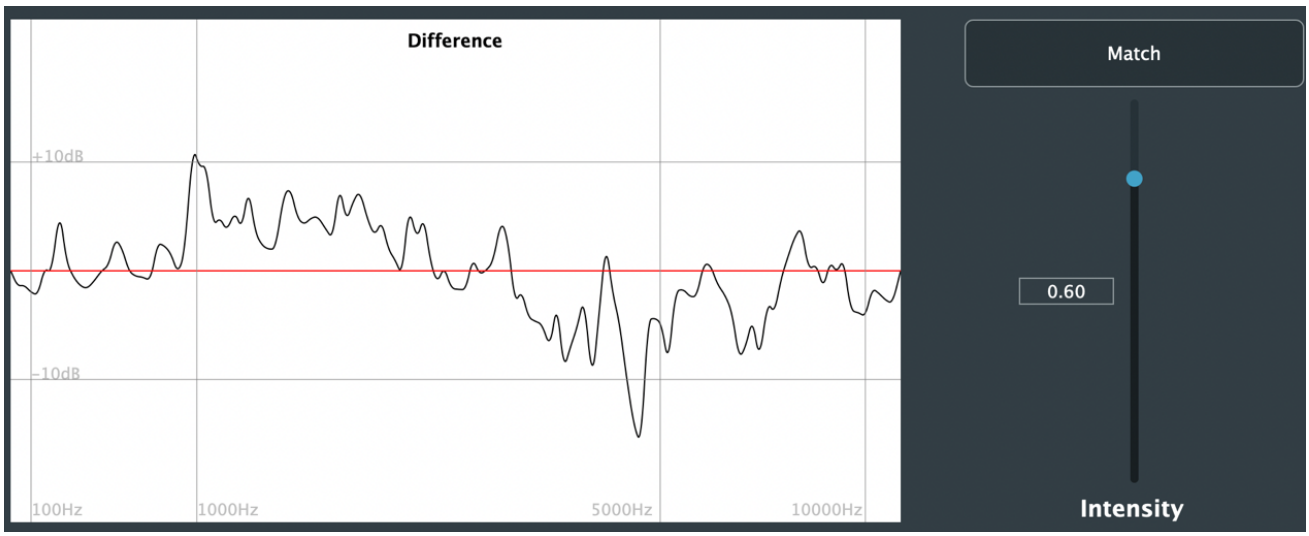

Figure 37 - Screenshot of the difference results from comparing the average spectrums of GuitarTarget and GuitarCurrent

7.4. TESTING THE FINAL IMPLEMENTATION

Assuming this to be the final implementation of the filters, we can now further explore the tests that were done during the prototyping and development of this project, by covering a range of different Target and Current comparisons.

This next set of tests was done by importing, into Logic Pro X, various pairs of audio files to analyse/match, which were mostly extracted from Logic Pro's Loops or recorded by the author. We then inserted our plug-in in the Target track, analysing the whole duration of the file. As soon as that is finished, we can move the same instance of our plug-in to the Current track (as the Target spectrum values will be saved), and then proceed to analyse the entire Current file provided. As soon as that is finished, we click on the "Match" button, and the array of filters is then set up to match our Current example to the Target. After auditioning the results and finding a suitable intensity with the slider, we bounce the matched version of the Current file.

(Refer to the User Manual and video in the FinalPlugin_Files/User Manual folder of the appendix to learn how to use the final plug-in.)

We now have, for each example, three audio files: Target, Current, and MatchedCurrent. To visually represent the effectiveness of our matching, we can first overlay the average spectrums of the Target and the Current, and in a different figure, overlay the average spectrums of the Target and the MatchedCurrent, which should look much closer to each other than before:

(Audio files included in Comparisons/GuitarExample folder in the appendix.)

This is the Target and the Current **before** the matching process (for the Guitar example):

Figure 37 – Target and Current average spectrums for the Guitar files, overlayed.

And this is the overlay of the Target and the MatchedCurrent **after** the matching process with 70% intensity:

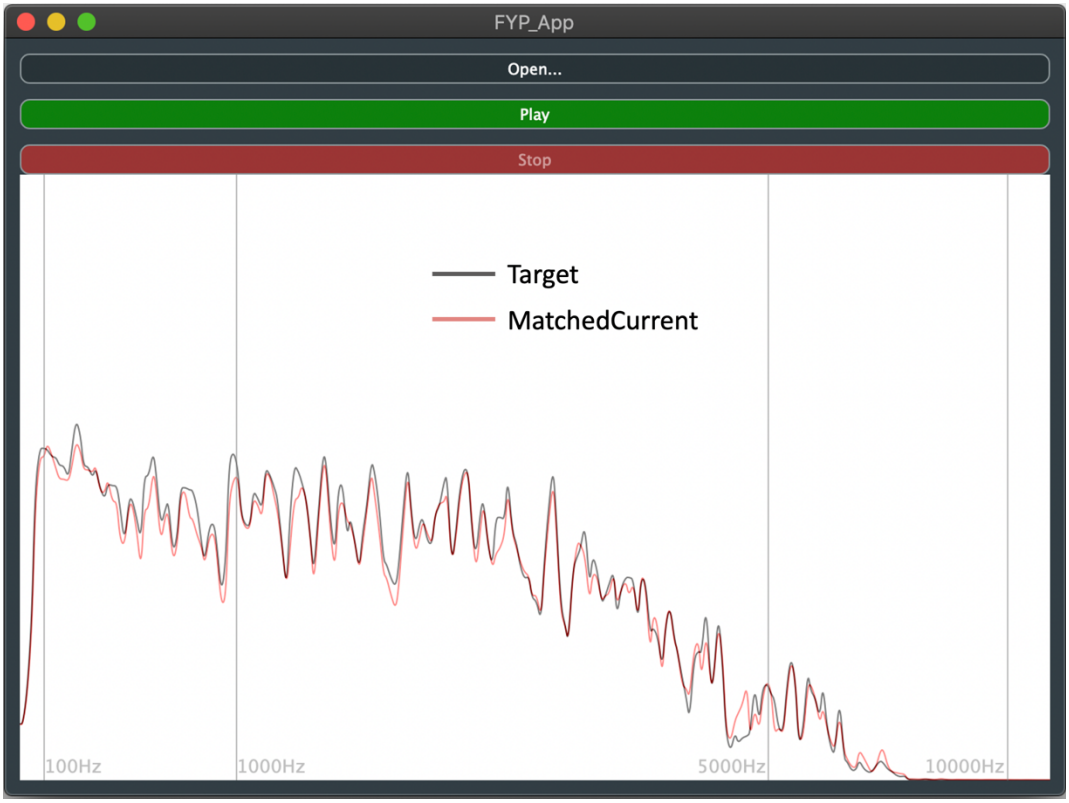

Figure 38 - Target and MatchedCurrent average spectrums for the Guitar files, overlayed.

We can now see that the average spectrums are much closer to each other, visually representing the fact that they also sound more similar now. Although there are still some audible differences between the Target and the Current, a mixing engineer should now be able to use them comped together in a take, without the listener realising they were recorded very differently.

In the case above, the intensity chosen was 60%.

We can try another similar example, using the introduction to “Smells Like Teen Spirit” by Nirvana as the Target (it has been edited so that we have the intro guitar for longer). As the Current, the author recorded the same pattern, but with drastically different sound to the Target. (Files available in the Comparisons/TeenSpirit folder.)

Below, we have the Target and the Current overlayed on the left side, and the Target overlayed with the MatchedCurrent on the right:

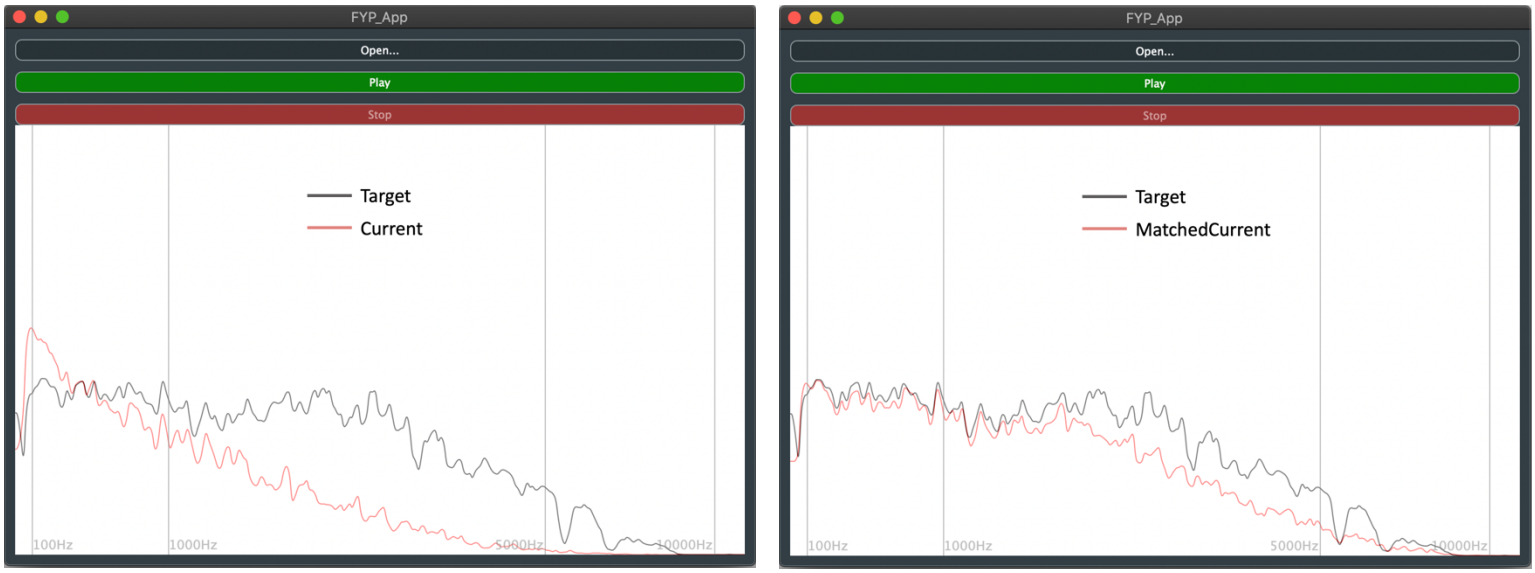

Figure 39 – Average spectrums of the Target and Current files (Smells Like Teen Spirit example) overlayed on the left side. Average spectrums of the Target and MatchedCurrent files on the right side.

Although the spectrums are now much closer to each other, there are still some differences that we can listen to, particularly in the 2000-5000Hz area. This might be due to the fact the Target also has some non-linear effects such as overdrive and some reverb, which the Current hasn't.

This is a good opportunity to emphasise that **our plug-in will only try to match spectral qualities of the audio**, which means that it won't be able to match attributes like dynamics, distortion, reverb, effects, etc.

It is fair to say that, for the two examples above, they are a bit simpler to process, as the performance between the two files, regarding dynamics, notes and expression is very similar.

To challenge this, we can try another example in which we use the same introduction to "Smells Like Teen Spirit" as the Target. However, as the Current, we will be using a guitar take recorded by the author playing a similar pattern but with very different dynamics/expression and transposed to another key (Files available in Comparisons/TeenSpirit2 folder):

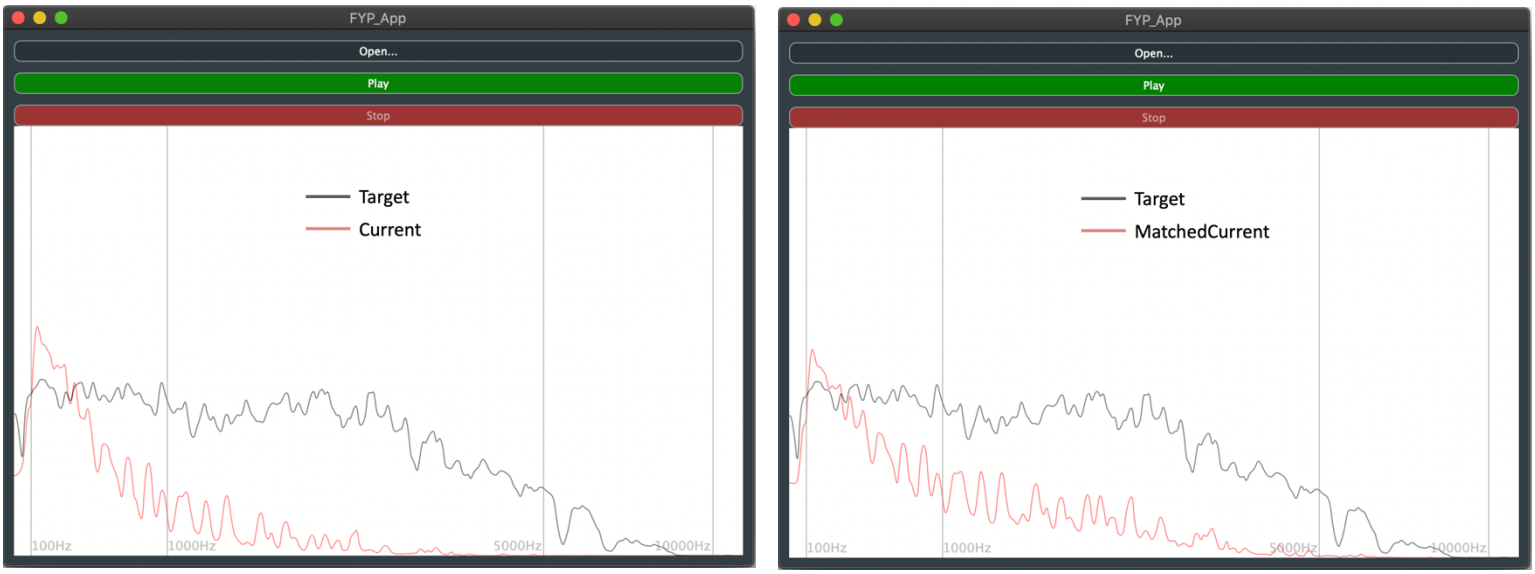

Figure 40 - Average spectrums of the Target and Current files overlayed on the left side. Average spectrums of the Target and MatchedCurrent files on the right side.

As we can see above, the overlay between the Target and the MatchedCurrent still shows some differences between them, which will support the fact that, **if the Target and Current are way too different, the matching process will not be successful** or useful, giving us results that sound unnatural. In a way, the two starting files need to be "compatible".

Because the analysis of two very different sources might give us a hugely contrasting Difference results (in decibels), there is the risk that some filters are set up with a very high gain, making the whole setup unstable and, in some cases, extremely loud.

To protect the user in these cases, we are limiting the gain values passed on to the filters using JUCE's ***jlimit*** function, constraining them to a maximum of +12dB and a minimum of -40dB **per filter**.

We can also try the same tests on excerpts of full mixes. In the following example, the files correspond to a chorus of two different songs, where the Target is "Wake Up Call" by Nothing But Thieves and the Current is an unmastered version of "Juicy Watermelon" by Cactus Haus, a song recorded and mixed by the author (Files available in Comparisons/FullMix folder):

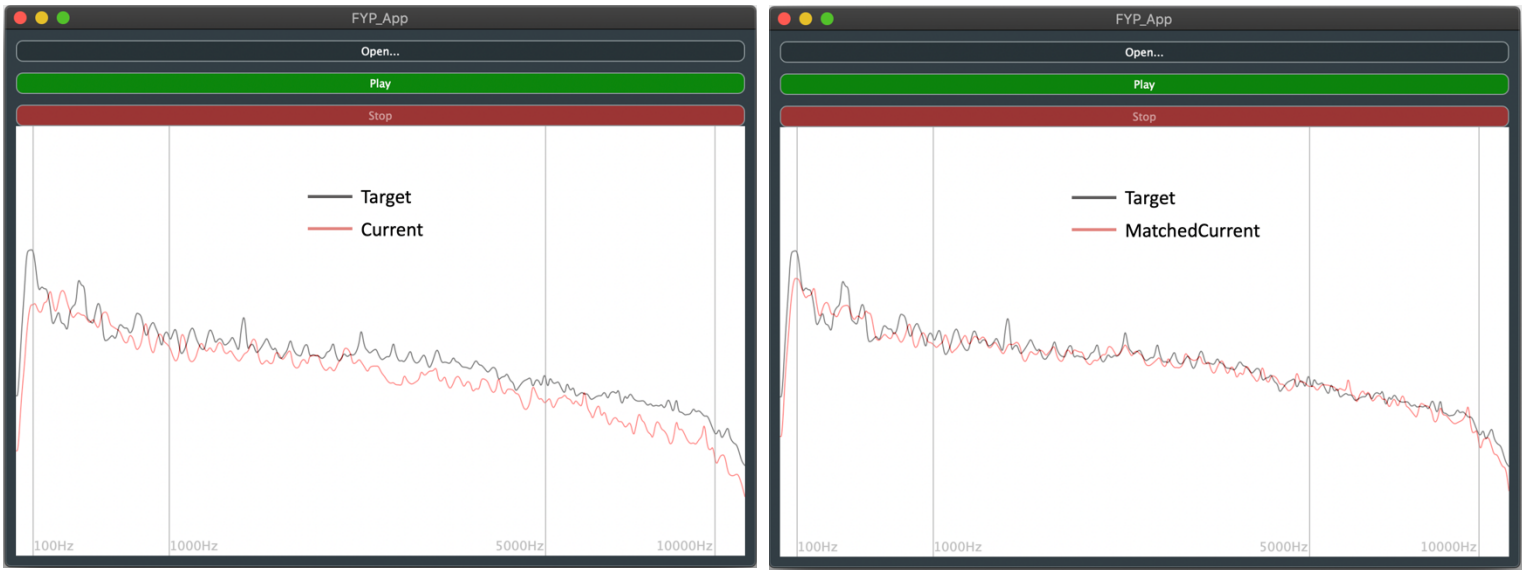

Figure 41 - Average spectrums of the Target and Current files overlayed on the left side. Average spectrums of the Target and MatchedCurrent files on the right side. Files used are mentioned above.

The MatchedCurrent now has a more prominent bass, as well as the high-end being more emphasised. It is worth mentioning that, with the Current being an unmastered version of "Juicy Watermelon", the overall level of the analysis was contrasting at the start. For this, we had to roughly balance the gain between the two excerpts before using our plug-in.

Having a built-in normalization step in our analysis would be a suitable idea, and this was tried during the prototyping, in the form of peak normalization. Whenever one of the spectrums finished the analysis, the algorithm would go over all of its magnitude FFT bins. After finding the peak magnitude value, it would divide all other bins by that same value, normalizing the array.

However, as Pestana *et al.* (2013) suggest, "strict normalization is not the answer, as spurious radical peaks in the frequency distribution might cause overall lower power levels, and the comparison would yield results that showed a variability that was greater than the real variability". For this reason, our peak normalization algorithm was removed from the final implementations, as it was making our analysis unreliable, thus affecting the filter gains. (For a suggestion of a suitable normalization approach, refer to the Future Improvements in Chapter 9.)

7.5. NOTES FROM TESTS

From the continuous testing that went on during the development of this project, and from some examples mentioned previously, we can note down a few conclusions regarding the usage of our plug-in:

- An intensity value of 100% was never chosen in our tests, as most examples would sound unnatural, with a lot of resonances. iZotope (2019) also mentioned a similar note about their own Match EQ software.
- Our plug-in, as expected, will only match spectral attributes of the audio, not being able to match dynamics, distortion, reverb, effects, etc.
- The two sources to match need to be “compatible” with each other for the matching process to be successful, specifically regarding the musical notes played and their dynamics/expression, as well as the overall level. In cases where the excerpts were from similar performances, most of our tests were successful.
- Although this wasn’t mentioned before, our plug-in does affect the phase of the processed audio, which might be a disadvantage in some cases, where the output needs to be mixed with correlated sources (such as individual microphones in a drum kit).
- The analysis results tended to be more accurate whenever the excerpts were long enough for the averaging to appear almost static after a few seconds (roughly more than 15 seconds long)

For additional comparisons, refer to the Comparisons folder of the appendix, where a comprehensive set of files is given for each example.

8. CREATIVE USES

Apart from the uses suggested for our plug-in, it might also be used creatively for other purposes, such as:

- Inverse EQ Matching – rather than matching two audio excerpts, we can also set the sources even further apart with intensity values below 0%. Because our analysis details the contrasting frequency regions in the two sources, negative intensities will emphasise those differences, rather than try to balance them. This could be particularly useful for mixing, to minimise frequency masking.
- Let's imagine the user has two microphones - A and B, where they will only have access to microphone B for one day, whereas microphone A is always available – and records a broad frequency range sound (such as white noise) through both of them (with the same preamp, gain, and with the capsules very close to each other). If the user then analysed the audio B (from microphone B) as the Target, and audio A as the Current, they would, theoretically, get the spectral rebalance that would need to be applied to microphone A to match microphone B. Although this use wasn't tested, and there are definitely other factors to have in mind, this could be an interesting idea to try with other equipment – such as guitar amps, hardware equalisers, etc.

9. FUTURE IMPROVEMENTS

As this project had to follow a strict deadline, it is worth mentioning a few future improvements that could have been made to our plug-in, had there been more time:

- Implementing a "Smoothing" slider, similar to iZotope's, which could control the number of subdivisions per octave that our logarithmic distribution would be using, lowering the number of filters used. This could essentially smooth our frequency response to only match the overall spectral envelope of the sources, rather than matching every peak and valley.

- Implementing a suitable normalizing step. For us not to use peak normalization, Pestana *et al.* (2013) suggest “scaling all spectral distributions so that the bin sum would be 1, followed by averaging the cumulative distribution function”.
- Implementing linear-phase processing, possibly by using FIR filters instead of IIR.

One other major approach that was tried but wasn’t fully included in this report, due to the word count limit, was using **convolution** for the matching process:

For this, there are two main findings that are important to have in mind: “A system’s frequency response is the Fourier Transform of its impulse response” (Smith, 1997), and the output of an FFT “can be converted back from the frequency domain into the time domain signal by applying the *Inverse Fourier Transform*” (Lerch, 2012, p.186).

As the Difference results that we get from the analysis closely constitute the frequency response that our plug-in needs to have in order to match the Current to the Target, we could process our spectral difference results through an inverse FFT, obtaining an impulse response. That impulse response could then be loaded onto JUCE’s **dsp::Convolution** class, without the need to manually set up any filters like it was done for our last implementations.

This idea was tried for roughly two weeks. However, its implementation was not successful and, thus, not included in this report. The author believes this might be because of the following reasons: we have been only dealing with the magnitude values from the FFT output so far. After some tests with the inverse FFT function in the dsp::FFT class, it became clear that the phase values also needed to be considered for an adequate impulse response to be constructed. This meant that we needed to directly handle the complex numbers from the FFT output (extracting them and separating magnitude and phase values); Additionally, the author believes that the phase values might not be able to be averaged like it has been done for the magnitude values, which would mean that a considerable percentage of our code would need to be redesigned.

Because this approach was getting rather complex and out of the scope of our project, it was decided that an array of IIR filters would be used instead, as they were simpler to setup and experiment with in JUCE. However, this might be a really interesting idea to explore in the future.

10. CONCLUSION

At the start of this project, the author set out to complete the following objectives:

- Researching techniques and concepts behind spectral measurement and analysis, to be used in a programming environment.
- Studying and using JUCE for the first time as a suitable framework to create audio plug-ins with C++ that could be professionally used.
- Creating a plug-in that is able to analyse the spectrum of two given audio tracks and rebalance one to match the other.
- The plug-in should work with different types of source material (vocals, guitar, drums, full mixes, etc.)
- It should support different plug-in formats to work with most major DAWs.

A final implementation of the audio plug-in was detailed, as well as suggestions for different approaches to the main goal of creating a tool that would be able to analyse the spectral differences between two audio sources and match their spectrums.

From our testing, it is possible to say that our plug-in also provided good results with different types of instruments, although the user needs to have in mind a few guidelines mentioned in chapter 7.5:

- The two sources to match need to be “compatible” with each other for the matching process to be successful, specifically regarding the musical notes played and their dynamics/expression, as well as the overall level. In cases where the excerpts were from similar performances, most of our tests were successful.
- A processing value of 100% was never chosen in our tests, as most examples would sound unnatural, with a lot of resonances.
- Our plug-in, as expected, will only match spectral attributes of the audio, not being able to match dynamics, distortion, reverb, effects, etc.
- Our plug-in does affect the phase of the processed audio, which might be a disadvantage in some cases.
- The analysis results tended to be more accurate whenever the excerpts were long enough for the averaging to appear almost static after a few seconds (roughly more than 15 seconds long)

Additionally, the final plug-in can be loaded onto different DAWs, such as Logic Pro X and Reaper, working as expected in a professional environment: it is able to save and recall its state, the user interface can be closed and opened without interfering with the audio processing, relevant parameters are automatable, etc. AU and VST3 formats of the plug-in were created.

After detailed testing, some future improvements were suggested, as well as another contrasting approach that could have been used in case the author had the option to restart this project with their current knowledge on the subject.

On a personal level, the author is now relatively comfortable with using the JUCE framework to develop other plug-ins in the future, as well as being able to better understand advanced tools such as the Fast Fourier Transform, which seemed to have a lot of potential for very different usages in audio processing systems.

Additionally, research skills were also improved, as there was the need to find relevant and suitable information that matched the present knowledge of the author. The area of filter design and Fourier transforms was daunting at the start, given the amount of advanced mathematics and concepts that go into it.

Although this report tries to be very concise and simple when explaining the entire process behind it, this was a very challenging project, which involved a lot of advanced research. It also involved learning JUCE – having never used it before, the author spent almost 2 months just going through audio programming techniques, JUCE tutorials and C++ concepts – as well as identifying and solving a considerable amount of bugs and problems with the code. This greatly improved the author's programming and problem-solving skills. Additionally, maths and statistical analysis skills were also further developed as there was the need to analyse incessant amounts of data (specifically FFT bins and audio samples) and recognise errors.

For all of these challenges to be overcome, it was important from the start to plan a very methodical and logical approach to the whole project, diving each objective in a huge number of smaller objectives, which were logged on to a daily checklist. For every day that was spent working on this project, a log was kept of what was accomplished, as well as any relevant information to later use in this report. This also helped the author to suitably adapt to any different approaches, ideas, and problems, changing the next steps where needed.

For the most part, weekly meetings were scheduled with the project supervisor, which helped shape the progress of the plug-in and the report, as the author always considered and explored the information and feedback that was given.

11. REFERENCES

Apple (2020) *Logic Pro Effects* [online]. Cupertino, US: Apple. [Accessed 14 October 2020].

Apple (2021) *Logic Pro.* Available from: https://apps.apple.com/gb/app/logic-pro-x/id634148309?mt=12 [Accessed 15 April 2021].

Bagley, J. (2020) *Making Spectrograms in JUCE.* Available from: https://artandlogic.com/2019/11/making-spectrograms-in-juce/amp/ [Accessed 03 January 2021].

Creasey, D. (2016) *Audio Processes: musical analysis, modification, synthesis and control* [online]. London: Focal Press. [Accessed 04 November 2020].

Dobrian, C. (2019) *Frequency and Pitch.* Available from: https://dobrian.github.io/cmp/topics/physics-of-sound/1.frequency-and-pitch.html [Accessed 10 March 2020].

FabFilter (2018) *FabFilter Pro-Q 3 User Manual* [online]. Amsterdam: FabFilter. [Accessed 14 October 2020].

FabFilter (2021) *Fabfilter Pro-Q 3.* Available from: https://www.fabfilter.com/shop/pro-q-3-equalizer-plug-in [Accessed 15 April 2021].

Huber, D. M. (2017) *Modern Recording Techniques* [online]. 6th. New York: Routledge. [Accessed 13 April 2021].

iZotope (2011) *Ozone 5 Help Documentation* [online]. Cambridge, Massachusetts: iZotope. [Accessed 13 October 2020].

iZotope (2019) *Ozone 9 Help Documentation: Match EQ*. Available from: https://s3.amazonaws.com/izotopedownloads/docs/ozone9/en/match-eq/index.html [Accessed 13 October 2020].

iZotope (2021) *Ozone 9 Advanced.* Available from: https://www.izotope.com/en/shop/ozone-9-advanced.html [Accessed 15 April 2021].

JUCE (2020) *JUCE.* Available from: https://juce.com/ [Accessed 2 November 2020]

JUCE (n.d.) *Tutorial: Visualise the frequencies of a signal in real time.* Available from: https://docs.juce.com/master/tutorial_spectrum_analyser.html [Accessed 18 December 2020]

JUCE (n.d.) *dsp::FFT Class Reference.* Available from: https://docs.juce.com/master/classdsp_1_1FFT.html [Accessed 18 December 2020]

Lerch, A. (2012) *An Introduction to Audio Content Analysis*. Hoboken, New Jersey: Wiley. [Accessed 04 November 2020].

Ma, Z..; Reiss, J. D.; Black D. A. (2013) Implementation of an intelligent equalization tool using Yule-Walker for music mixing and mastering. *Audio Engineering Convention 134,* Audio Engineering Society, Rome, 4-7 May 2013. AES E-Library [online]. Available from: https://www.aes.org/e-lib/browse.cfm?elib=16792 [Accessed 20 October 2020].

Marsar, J. (2015) Improving FFT Frequency Resolution. *Real-Time Digital Signal Processing* [online]. Available: http://www.add.ece.ufl.edu/4511/references/ImprovingFFTResoltuion.pdf [Acessed 20 December 2020]

Pajczur (2021) *pajEQanalyser.* Available from: https://pajczur.com/pajeqanalyser/ [Accessed 15 March 2021].

Park, T. H. (2010) Frequency-Domain and the Fourier Transform. In: Park, T. H. (2010) *Introduction to Digital Signal Processing: Computer Musically Speaking* [online]. New Jersey: World Scientific, pp. 276-333. [Accessed 06 November 2020].

Pestana, P.D.; Ma, Z.; Reiss, J.D.; Barbosa, A.; Black, D.A. (2013) Spectral Characteristics of Popular Commercial Recordings 1950-2010. *Audio Engineering Convention 135,* Audio Engineering Society, New York, 17-20 October 2013. AES E-Library [online]. Available from: https://www.aes.org/e-lib/browse.cfm?elib=17010 [Accessed 20 October 2020].

Pohlmann, K. C. (2010) Fundamentals of Digital Audio. In: Pohlmann, K. C., 6th ed. (2010) *Principles of Digital Audio* [online]. United States: McGraw-Hill, pp. 50-86. [Accessed 05 March 2021]

Rapuano, S. and Harris, F.J. (2007). An introduction to FFT and time domain windows. *IEEE instrumentation & measurement magazine* [online]. 10(6), pp.32-44 [Accessed 07 November 2020].

Robinson, M. (2013) *Getting Started With JUCE* [online]. United Kingdom: Packt Publishing. [Accessed 5 December 2020].

Senior, M. (2017) *Recording Secrets for the Small Studio*. New York: Routledge.

Sevgi, L. (2007) Numerical Fourier transforms: DFT and FFT. *IEEE Antennas and Propagation Magazine* [online]. 49(3), pp. 238-243. [Accessed 06 November 2020].

Smith, J. O. (2007) Quality Factor (Q). *Introduction to Digital Filters with Audio Applications* [online]. Available: https://ccrma.stanford.edu/~jos/fp/Quality_Factor_Q.html [Acessed 4 March 2021]

Smith, S. W. (1997) *The Scientist and Engineer's Guide to Digital Signal Processing* [online]. San Diego: California Technical Pub. [Accessed 06 February 2021].

Wickramarachi, P. (2003) Effects of windowing on the spectral content of a signal. *Sound and vibration* [online]. 37(1), pp. 10-13. [Accessed 07 November 2020].