UNIVERSITY OF
WEST LONDON

Disability Team
One Stop Shop
University of West London
St Mary's Road
Ealing
W5 5RF

Tel: 020 8231 2739

disability.support@uwl.ac.uk

15 August 2023

MSc Digital Audio Engineering
Starting: 19 September 2022
Finishing: 29 September 2023

RE: 21501990

Please do not accept a paper copy of this document, it is for electronic submission only

To Whom It May Concern,

This student is registered with Disability Support because of a disability.

Please refer to the University's Marking guidelines for students with Specific Learning difficulties (SpLD).

Yours faithfully,

Disability Support

* Please submit this file with your assessment.

UNIVERSITY OF
WEST LONDON

Audio Programming 2

VST3 JUCE Plug-in (Multi-Band Compressor)

Student name: Oberon Day-West.
Student ID: 21501990.
Course: MSc Digital Audio Engineering.

Assignment title: VST3 JUCE Plug-in (Assessment 2).
Module title: Audio Programming 2 (TC70032E).
Module leader: Dr Gerard Roma.
Date: 16th August 2023.
Word count: 3,680.

Abstract:
This report details the development of a basic compressor algorithm within the JUCE framework using C++ and the subsequent development of a multi-band compressor. Drawing from seminal works, the algorithm segments the audio signal into distinct frequency bands, enabling individualised compression. This innovative approach was tested rigorously, with visual representations highlighting the efficacy of the filters on audio signals. However, challenges, such as discrepancies in the GUI display, emerged, emphasising the need for further refinement. The project's complexity, combined with the intricate nature of compressors, suggests potential challenges for users. However, real-world testing within the DAW environment offers insights into the plugin's performance. In essence, this project merges classical filter theories with contemporary coding techniques, underscoring the potential for future development opportunities while emphasising the importance of continuous refinement and user feedback.

Table of contents

Code repositories: Note to reader

The project underwent several iterations and forms of prototyping, and a source control practice was utilised throughout (Git) to document and track the progress and feature development of the software. For reference, please access the repositories via the below links. If inaccessible, please contact the owner at 21501990@student.uwl.ac.uk.

Git repository 'one_MBComp':
https://github.com/0bi0n3/one_MBComp

The above Git repo contains the build, prototypes, README file and some iterations of the final plugin.

List of tables, figures and equations

1. Introduction

Dynamic audio effects manipulate audio signals by applying a time-varying gain, commonly a nonlinear function of the input signal level. These effects are frequently used to reshape the amplitude envelope of a signal, often resulting in the compression or expansion of its dynamic range. Two significant forms of dynamic processing are dynamic range compression and expansion, including their more extreme versions, limiting and noise gates. Dynamic range compression refers to mapping the perceived dynamic range of an audio signal to a smaller range. This effect is achieved by reducing the high signal levels and leaving the quieter parts unprocessed, thus compressing the dynamic range. This concept should not be confused with data compression as utilised in audio codecs, which serve a different purpose (Winer, 2018; Bennett, 2021).

The design of a compressor, and the associated design choices, have a considerable impact on the perceived sonic characteristics. Specific adjustable parameters often dictate compressor operations. A compressor typically controls variable gain, modulating the gain based on the input level. It applies attenuation when the signal level is high, softening louder passages and thus reducing the dynamic range. This modulation can be implemented as a feedforward or feedback device, as illustrated in Figures 1.1 and 1.2. Key compressor parameters include the threshold, which defines the level above which the compressor activates and reduces the signal level. The ratio controls the input/output ratio for signals exceeding the threshold level, determining the amount of compression applied.

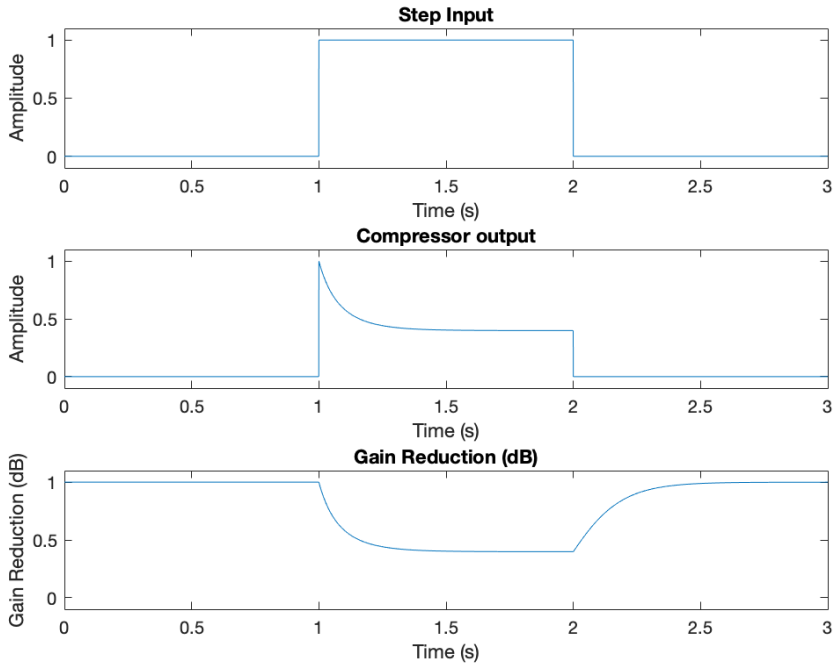

Figure 1.1: Compressor theory test.

Other important parameters include attack and release times, or time constants, which dictate the speed at which a compressor responds to signal-level changes. The attack time refers to the time the compressor takes to decrease the gain to the level dictated by the ratio once the signal overshoots the threshold. The release time defines how long it takes to restore the gain to normal once the signal falls below the threshold (Zolzer, 2011; Reiss and McPherson, 2015; Tarr, 2019).

Limiting is a more extreme form of compression with a very high ratio that limits the signal level hard. A makeup gain control is usually provided at the compressor output to compensate for the reduced signal level, allowing the matching of input and output loudness levels. The knee width option controls the transition in the response curve at the threshold, with a sharp transition providing more noticeable compression and a softer transition yielding a less perceptible effect. The signal entering the compressor is split into two copies; one is sent to a variable gain amplifier (the gain stage) and the other to an additional path, a sidechain. The central computer in the sidechain applies the necessary gain reduction. This sidechain signal is then transformed into a unipolar representation of the level. The gain stage attenuates the input signal over time, a function the voltage-controlled amplifier (VCA) performs. Modern compressor designs use specialised integrated VCA circuits, which provide predictability and improved specifications such as less harmonic distortion and a higher usable dynamic range (Lyons, 2010; Pirkle, 2012; 2019; Reiss and McPherson, 2015).

In a digital design, an ideal VCA can be modelled by multiplying the input signal by a control signal from the sidechain, and makeup gain is often used to add a constant gain back to the signal to match output and input levels. Ultimately, the precise functioning of the compressor and the quality of its output are determined by these processes and parameters.

This plugin, a dynamic range (DR) processor, aims to alter the amplitude of audio signals, thereby manipulating the signal's dynamic range. The dynamic range is defined as the disparity between the signal's maximum and minimum amplitudes, and the following sections in this report outline the design, reference to the implementation techniques and outcomes, finally, conclude with an analysis and review of the various tests and results obtained from the final plugin iteration.

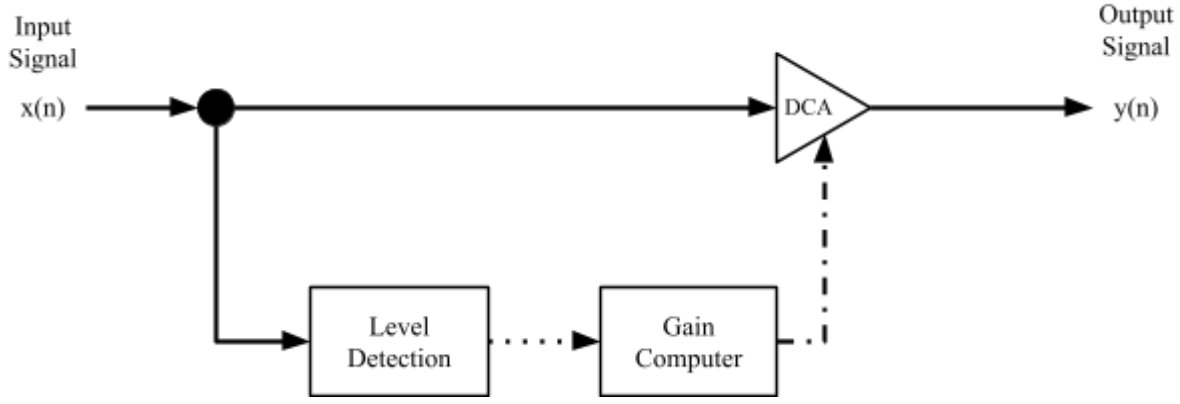

Figure 1.2: Signal flow diagram overview of feed-forward dynamic processor design.

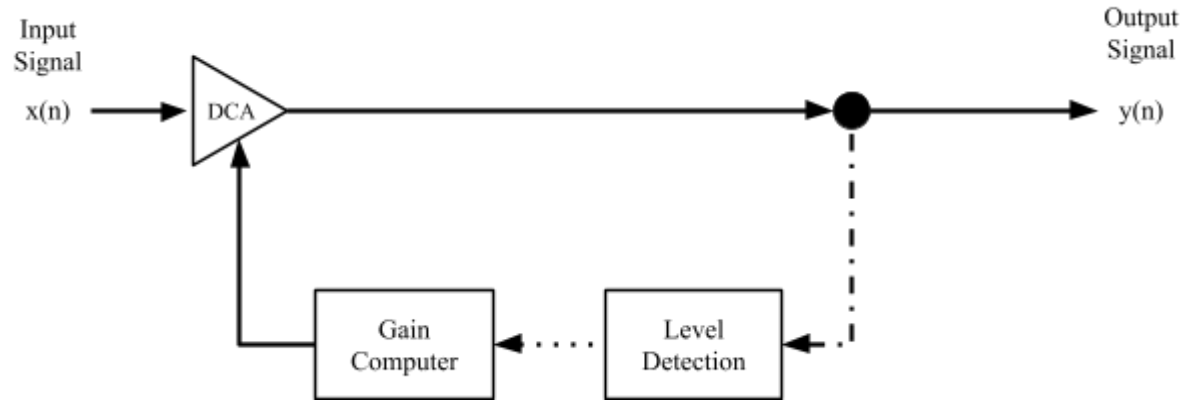

Figure 1.3: Signal flow diagram overview of feed-back dynamic processor design.

1.2 Audio plugin design

Multiband Compressor DSP Code

The digital signal processing (DSP) code forms the backbone of a multiband compressor. This code must be proficiently optimised to avoid audio errors and glitches that may disrupt the intended output—the process involves efficient handling of blocks of samples sequentially. The code ensures the seamless transformation of input signals to output, managing each frequency band separately. The efficiency of this process is critical as it directly impacts the real-time functionality of the compressor, affirming the need for well-crafted DSP code (Reiss and McPherson, 2015; Tarr, 2019; Bennett, 2021).

State-Based Parameters in Multiband Compressor

In multiband compression, parameters are state-based, influencing how the DSP code functions. These parameters, including the knee, threshold, and ratio, dictate the operation of the compressor. The knee defines the transition curve for the compressor, while the threshold sets the level at which compression starts. The ratio determines the degree of compression. Manipulating these parameters enables the precision tuning of the compressor to generate the desired output (Tarr, 2019; Schiermeyer, 2021b).

Preparing the DSP for Audio Processing

The preparation phase of DSP for audio processing involves several key considerations. The number of samples processed at once impacts the system's overall efficiency, and selecting an appropriate sample rate for processing is crucial for preserving audio fidelity. Moreover, the number of channels being processed will affect the computational requirements of the DSP (Schiermeyer, 2021b). These

parameters should be carefully configured to ensure that the multiband compressor can handle the audio data efficiently and effectively.

PrepareToPlay() Function in JUCE

In the JUCE audio programming framework, the prepareToPlay() function plays a significant role. This function sets up essential aspects such as the sample rate, number of channels, and buffer or block size before audio processing begins. It is often more manageable to send smaller blocks of samples for processing instead of larger ones to optimise system performance (Pirkle, 2019; Schiermeyer, 2021b). This function establishes the groundwork necessary for the subsequent DSP operations.

Multiband Compressor GUI

A user-friendly and intuitive graphical user interface (GUI) is crucial for operating a multiband compressor. Initially, developers may use the generic processor editor GUI provided by JUCE. However, as the project evolves, custom designs become a necessary direction. The GUI should allow users to adjust the knee, threshold, and ratio parameters to reflect that of previously designed plugins and effects.

2. Methodology

A basic compressor algorithm was explored as the foundation for developing a multi-band compressor within the JUCE framework using C++ (see Appendix A-I). To create a multi-band compressor, the algorithm can be extended to split the audio signal into multiple frequency bands using the digital filters theory of a Linkwitz-Riley filter (cascaded Butterworth filters) (Lyons, 2010; Pirkle, 2019; Schiermeyer, 2021b). Each band's signal is fed into separate instances of the primary compressor, applying compression independently to each frequency band. Afterwards, the compressed bands are recombined to create the final output signal. Additionally, parameters for each band, including threshold, ratio, attack time, and release time, can be exposed as plugin parameters to enable user control. Leveraging JUCE's GUI capabilities, an interface can be designed for real-time parameter adjustments.

Due to the project's scope and the system's complexity, all functions and processes contained within the VST plugin will not be entirely described or analysed herein. However, the following source file sections will be reviewed: PluginProcessor.cpp, basicCompressor.cpp and butterworthFilter.cpp. These three source files contain most of their processes' authored code and implementations.

The primary compressor algorithm:

The main algorithm to test and establish the compressor theory was referenced and adapted from the source material by Reiss and McPherson (2015) and Tarr (2019). The algorithm was tested with two sample audio loops, mainly music-focused, so the dynamic range effect could be more readily analysed. Figure 2.1 shows the control and output from the algorithm, which was prototyped in MATLAB (Appendix A). As can be seen from the graphs, the signal in the output has some of the dynamic range reduced on several of the transients. Most notably, the 'zoomed' portions of the graph show the control line, which attenuates the output signal's transients.

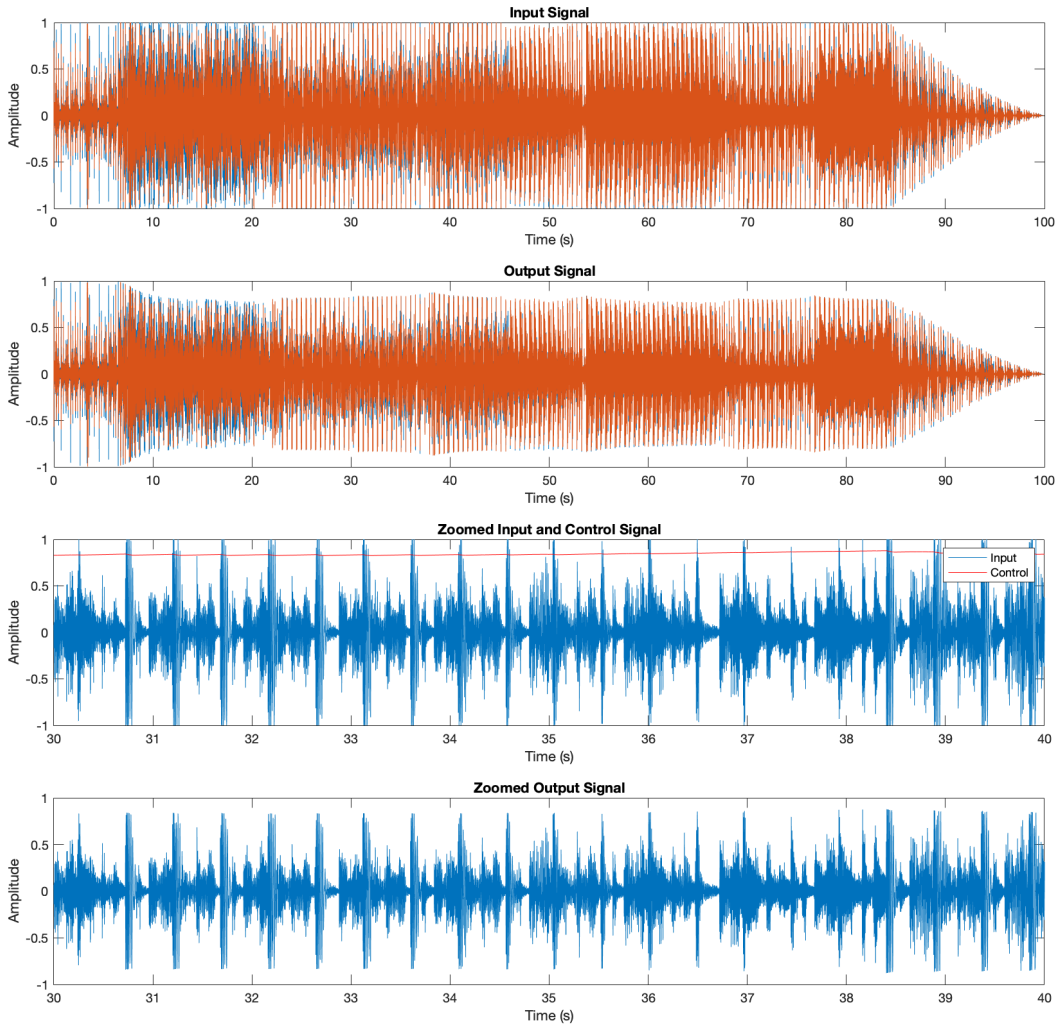

Figure 2.1: basicComp.m output graphs.

The code in Appendix K encapsulates the multiband compressor audio processor prototype developed in MATLAB. This piece builds upon prior works by Schiermeyer (2021a; 2021b), Pirkle (2019), and Tarr (2019), and it integrates modules for the primary compressor, along with Butterworth filtering.

The audio processor class, named `One_MBCompAudioProcessor` (in Appendix B and C), is the encapsulating multiband compression body, working distinctly on low, mid, and high-frequency

bands. Within the constructor, the audio buses' properties are set, and channel configurations are defined based on the nature of the plugin. In addition, various filters, such as lowpass and highpass filters, are initialised based on the current sample rate.

Parameter management is a pivotal segment of this code. To facilitate this, helper lambda functions — `floatHelper`, `choiceHelper`, and `boolHelper` — link the parameters with corresponding attributes in the compressor modules. This ensures that when a user modifies a parameter in the plugin's graphical interface, the underlying processor attribute updates accordingly.

Within the `process block` method, core audio processing occurs. Here, denormalisation is inhibited for performance efficiency (Schiermeyer, 2021a; 2021b). The plugin first applies input gain, segregates the signal into three frequency bands using filter modules, processes each band through its respective compressor, and subsequently merges the outputs. Depending on user settings, certain bands can be muted or soloed. Finally, output gain is applied before the audio is passed onward in the DAW's signal chain (see Appendix B and C).

Pre-playback initialisation occurs in `prepare to play`, which sets up requisite components with the given sample rate and block size. An editor component, visualising and allowing interaction with the plugin's parameters, is provided via the `createEditor` method.

Figure 2.2 provides an overview of the PluginProcessor.cpp file's processBlock function and the data flow from input to the processing that later occurs with the various implementations of the filtering and compression algorithms.

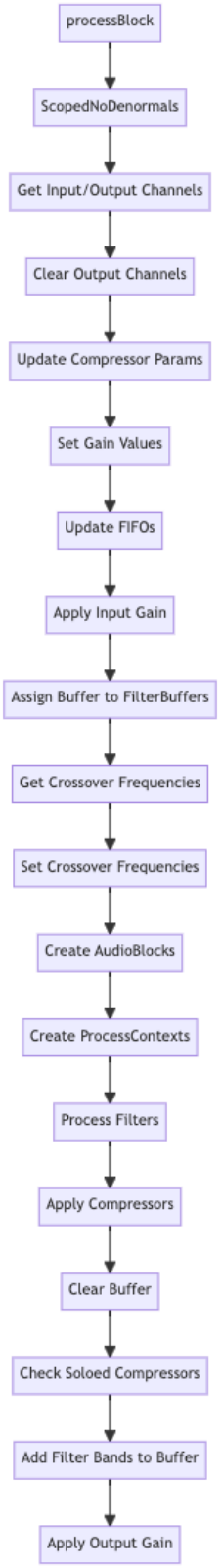

Figure 2.2: PluginProcessor.cpp processBlock overview of signal flow. The overview flow diagram provides a “roadmap” for navigating the general signal flow of the audio data as it enters the process block.

Compressor class

The basicCompressor class in the developed code (see Appendix F-M for complete source) is a simple implementation of an audio compressor using parts of the JUCE library. The class has several private member variables representing the parameters of the compressor:

m_thresholdLevelDb: The level above which compression is applied.

m_compressionRatio: The amount of compression applied to signals above the threshold.

m_attackTimeInMs and m_releaseTimeInMs: The time it takes for the compressor to start and stop compressing after the signal crosses the threshold.

m_newMakeUpGainDb: The gain applied after compression to bring the level back up.

m_alphaAttack and m_alphaRelease: Coefficients used in the envelope follower for the attack and release stages.

The class also has several public methods to set these parameters and to process audio data:

prepare: This method prepares the compressor by calculating the m_alphaAttack and m_alphaRelease coefficients based on the sample rate and the attack and release times.

setThresholdLevel, setCompressionRatio, setAttackTime, setReleaseTime, setMakeUpGain: These methods set the corresponding parameters.

process: This method applies compression to the audio data.

The compression is applied in the process method using the following equations:

The input level in decibels is calculated from the input sample,

$$inputLevelInDecibles = 20.0f \cdot log_{10}(abs(currentSample)) \quad (1.1)$$

The output level in decibels is calculated based on the input level and the threshold and compression ratio,

$$outputLevelInDecibels = m\_thresholdLevelDb + (inputLevelInDecibels - m\_thresholdLevelDb) / m\_compressionRatio \quad (1.2)$$

if the input level is above the threshold, otherwise it is the same as the input level.

The difference between the input and output levels is calculated,

$$levelDifference = inputLevelInDecibles - outputLevelInDecibles \quad (1.3)$$

An envelope follower is applied to the level difference to smooth the changes and avoid distortion. The envelope level is calculated based on the previous envelope level and the level difference, using the m_alphaAttack or m_alphaRelease coefficient, depending on whether the level is increasing or decreasing.

The gain to be applied to the sample is calculated from the make-up gain and the envelope level,

$$gainForSample = pow(10.0f \frac{(m\_newMakeUpGainDb = envelopeLevel)}{20.0f}) \quad (1.4)$$

The gain is then applied to the sample to get the output sample,

$$outputSample = currentSample \cdot gainForSample \quad (1.5)$$

This process is applied to each sample in the audio data, and the result is a compressed version of the input.

Butterworth & LinkwitzRiley classes

The provided code in Appendix H and I demonstrate the C++ implementation of two audio filters: the Butterworth filter and the Linkwitz-Riley filter. Both filters were explicitly implemented in the same source file for ease of reference and processing. The below outlines the functionality and code makeup of these classes.

Butterworth Filter (ButterFilter)

Variables:

- filterType: Enumerated type indicating the type of filter (lowpass, highpass, or allpass).
- coefficientA0, coefficientA1, coefficientA2, coefficientB1, coefficientB2: Coefficients used in the filter's operation.
- cutOffFrequency, qualityFactor: Parameters that define the filter's behaviour.
- sampleRate: The rate at which audio samples are processed.
- previousSamples1, previousSamples2: Vectors to store previous samples for filtering.
-

Functions:

- Constructor: initialises the filter with a given sample rate and filter type.

- prepare: Prepares the filter by resizing and resetting the vectors based on the number of channels provided.
- setFilterParameters: Sets the filter parameters and calculates the coefficients based on the filter type.
- processFilter: Processes an input sample through the filter and returns the filtered sample.
- updateSampleRate: Updates the sample rate and recalculates the filter parameters.
- process: Processes a block of samples through the filter.

Linkwitz-Riley Filter (LinkwitzRFilter)

Variables:

- filterType: Enumerated type indicating the type of filter.
- lowPassFilter, highPassFilter, allPassFilter: Instances of the Butterworth filter to achieve the Linkwitz-Riley filter's behaviour.

Functions:

- LinkwitzRFilter: Constructor that initialises the filter with a given sample rate.
- prepare: Forwards the preparation call to the inner Butterworth filters.
- setCrossoverFrequency: Sets the crossover frequency for the inner Butterworth filters.
- processFilter: Checks the filter type and processes the input sample accordingly.
- setType: Sets the filter type.
- process: Processes a block of samples through the filter.

Enumerations:

FilterType: Enumerated type with values lowpass, highpass, and allpass to indicate the type of filter.

Filter Coefficients Calculation:

In the setFilterParameters function of the ButterFilter class, filter coefficients are calculated based on the cutoff frequency, quality factor, and filter type. These coefficients determine the filter's behaviour.

Double Processing in Linkwitz-Riley:

The processFilter function of the LinkwitzRFilter class processes the input sample through the Butterworth filter twice, which is a characteristic of the Linkwitz-Riley filter (Schiermeyer, 2021b; Pirkle, 2019; Tarr, 2019).

Error Handling:

The code contains error handling mechanisms, such as checking for invalid filter parameters or channel indexes, and throws exceptions accordingly.

Use of JUCE Library:

The code uses the JUCE library's dsp::ProcessSpec and dsp::ProcessContextReplacing classes for audio processing, enabling the design for real-time audio applications. It is recognised that the project specifications detailed that using the JUCE DSP module was not encouraged; however, to achieve the desired results, certain aspects were leveraged to complete the project successfully.

In order to demonstrate the implementations of the band-filtering, the following section details the prototyping and analysis that took place to achieve the filter implementations. The scripts reference several sources, including Bristow-Johnson (2005), neotec (2007), Falco (2009), and Zolzer (2011), for their successful implementation.

In the linkWorth.m, linkWorth_HP.m and linkWorth_AP.m files, a programmatic representation of the Butterworth and the subsequent LinkwitzRiley filtering, is outlined. The central function, "ButterFilter", is fundamentally constructed to encapsulate the characteristics of a Butterworth filter, including filter parameters such as cutoff frequency, quality factor, and filter type. This filter uses the bilinear transform method to translate an analogue filter into a digital one, with the underlying coefficients derived from the filter's type - either lowpass, highpass, or allpass.

Demonstrating the prototyping involved generating test signals, applying the different filter types to the signal, and analysing the resulting output. Different analyses are presented below to provide a comprehensive insight into the prototypes.

Figure 2.3 outlines the lowpass filter function, showing the combined testing of the following parameters:

```
Fs = 44100; % Sample rate
t = 0:1/Fs:1; % 1 second of data
f1 = 400; % Frequency
f2 = 25; % Frequency
x = sin(2*pi*f1*t) + sin(2*pi*f2*t); % Sum of two sine waves
coefficients = setFilterParameters(300, 0.707, 'lowpass', sampleRate)
```

From the initialisation parameters, the resulting output shows the deduction of the higher frequency of 400Hz from the lower frequency of 25 Hz. As can be observed in the setFilterParameters statement, the cutoff frequency of 300 Hz demonstrates the successful filtering of the higher frequencies from the test signal.

Within the visual representation in Figure 2.4, two subplots distinctly contrast the magnitude spectra of both the original and filtered signals. The spectrum of the original signal prominently displays peaks corresponding to the frequencies of its two sine wave components, specifically at 3000 Hz and 10000 Hz. When observing the filtered signal's spectrum, the influence of the highpass filter

becomes evident. With a designated cutoff frequency set at 5000 Hz, there is a noticeable attenuation of the 3000 Hz component, while the 10000 Hz component largely retains its prominence. Both plots have been strategically zoomed to focus on the frequency range between 1000 Hz and 20000 Hz, highlighting the region of utmost interest to enhance clarity and provide a concentrated view.

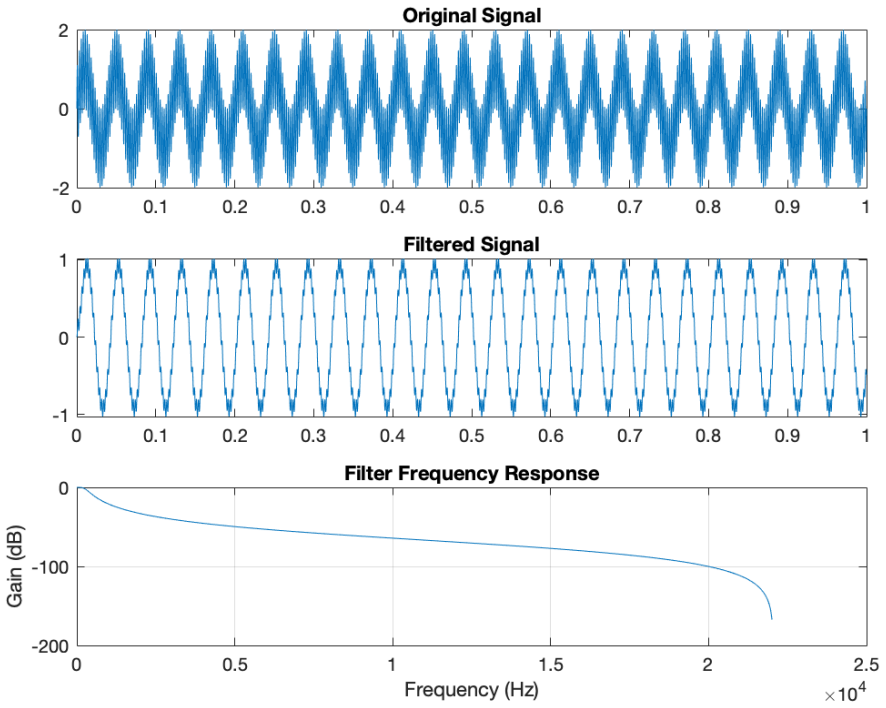

Figure 2.3: Lowpass filter frequency response and filtering.

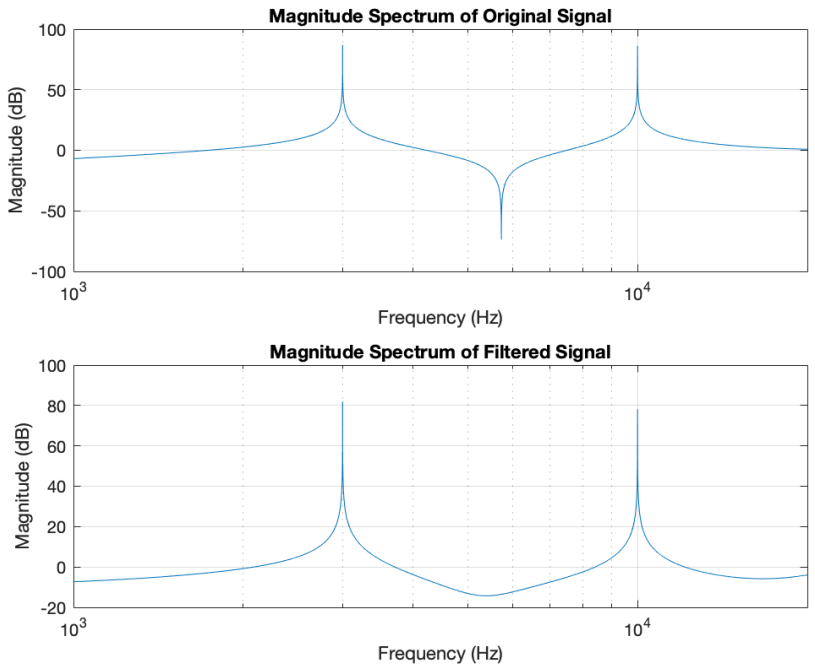

Figure 2.4: Highpass filter magnitude spectra response.

In Figure 2.5, the first plot offers a zoomed-in view of the initial 20ms, contrasting the original and filtered signals. A subsequent plot displays the entire duration of both signals for a broader comparison. One plot illustrates the filter's magnitude response with a marked 750 Hz cutoff (see Appendix K), while another highlights its phase shifts across frequencies. In the magnitude spectra, two plots juxtapose the original and filtered signals. The original showcases peaks at 500 and 1000 Hz, while the filtered signal emphasises phase changes from the allpass filter, maintaining consistent magnitude. Analysing these visuals, the allpass filter impacts the signal's temporal structure without significant amplitude changes. The frequency response plots reveal the filter's phase-altering characteristics. These plots elucidate the allpass Linkwitz/Butterworth filter's effect on a dual-frequency signal.

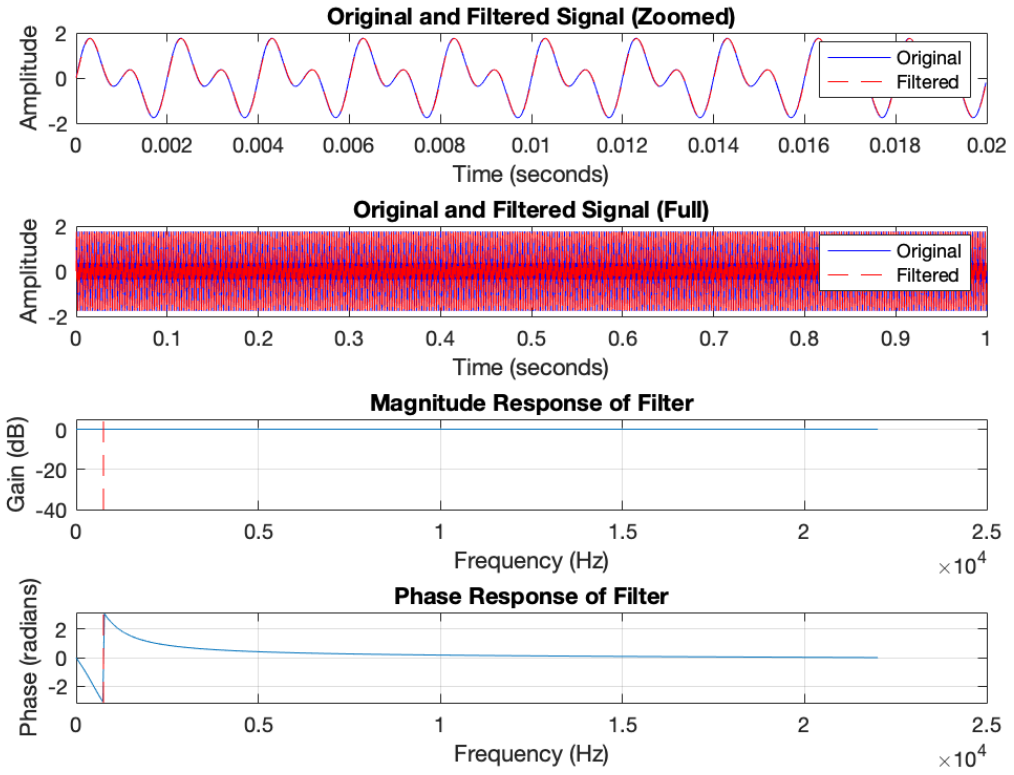

Figure 2.5: Allpass filter frequency, phase and magnitude response.

More generally, the filter's recursive structure is founded upon previous samples held within vectors, allowing the filter to function across multiple audio channels. These channels are processed individually, using the filter coefficients and past samples to deduce the current output sample. Notably, when parameters or the sampling rate are modified, the filter coefficients are recalculated, ensuring consistent and accurate filter operation. A higher-order filter is introduced through the LinkwitzRFilter class, which essentially chains the primary Butterworth filter in a sequence to attain

steeper roll-offs. The core notion here is to apply the primary Butterworth filter iteratively (twice in this case) on the input samples based on the set filter type (lowpass, highpass, or allpass). The code, inspired and adapted from previous works by Bristow-Johnson (2005), neotec (2007), Falco (2009), and Zolzer (2011), encompasses an example of classical filter theories with modern coding techniques.

2.2 Results & Testing

Analysing compressors is complex due to their nonlinear, time-dependent nature and memory involvement. They smoothly apply gain reduction, which deviates from simple static nonlinearity. The myriad of design choices further complicates the matter, making formulating a universally valid compressor diagram challenging. Real-world compressors vary in topology, stages, and digital design, with each deviation contributing to its unique character (Reiss and McPherson, 2015). However, despite these intricacies, we will delve into testing conducted within the final deployment environment of a DAW in the forthcoming results section.

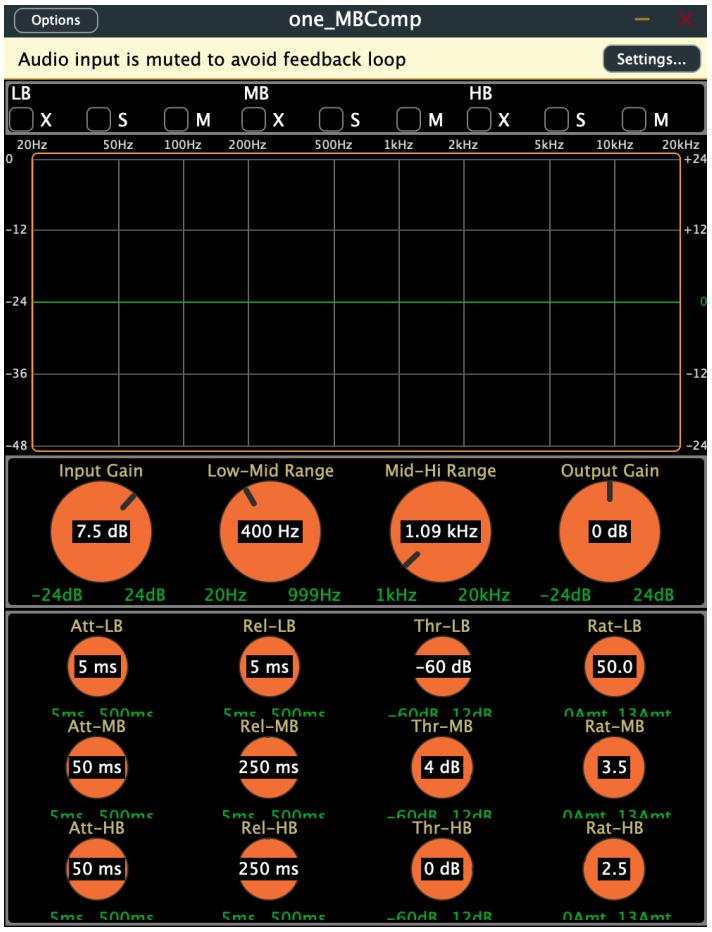

Figure 2.6: GUI overview of the plugin. The overall design was inspired by Tarr (2020a; 2020b) and Schiermeyer (2021a; 2021b).

Figure 2.6 presents the final GUI of the plugin, influenced by Tarr (2020a; 2020b) and Schiermeyer (2021a; 2021b). Despite rigorous testing, some GUI information does not display correctly, a bug yet to be addressed due to time constraints.

Subsequent figures depict a DAW session testing an audio signal with one control channel and another using the VST compressor plugin. While both signals start similarly, the test channel reveals altered parameters during output recording. Figure 2.7 highlights the unchanged orange control signal, while the pink effect channel showcases adjustments in attack, release, threshold, and ratio, impacting the test signal's final output.

Figures 2.8 and 2.9 further display audio processing with band adjustments. For a deeper analysis, readers can refer to the audio samples in the project's Git repository. This testing provided insights consistent with the current build's expectations, but further refinements are needed to enhance audio quality and control responsiveness.

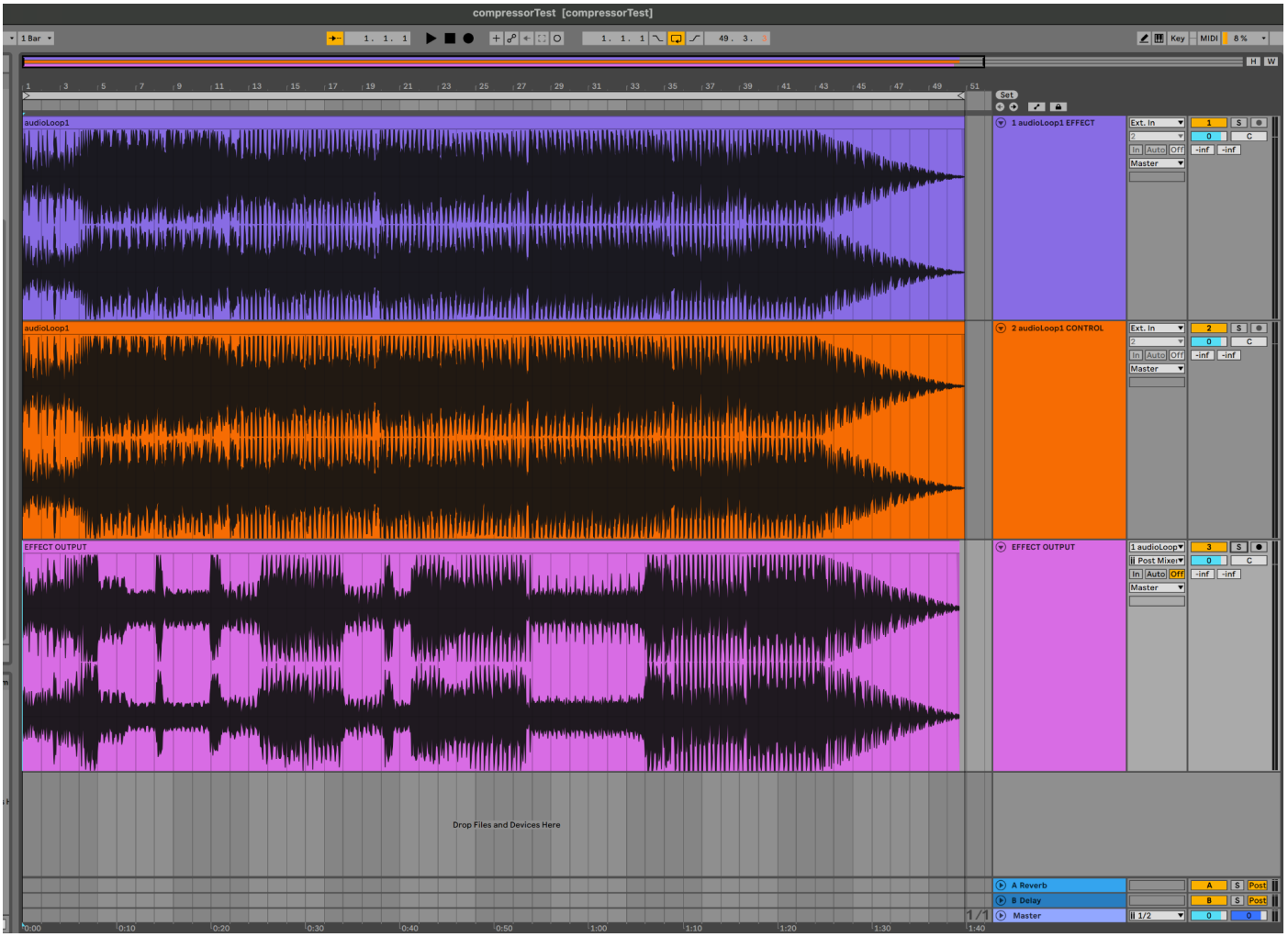

Figure 2.7: Ableton Live plugin testing. Demonstration of the overall test project, the original effect channel (purple), the control unaffected signal (orange) and output of the processed signal (pink).

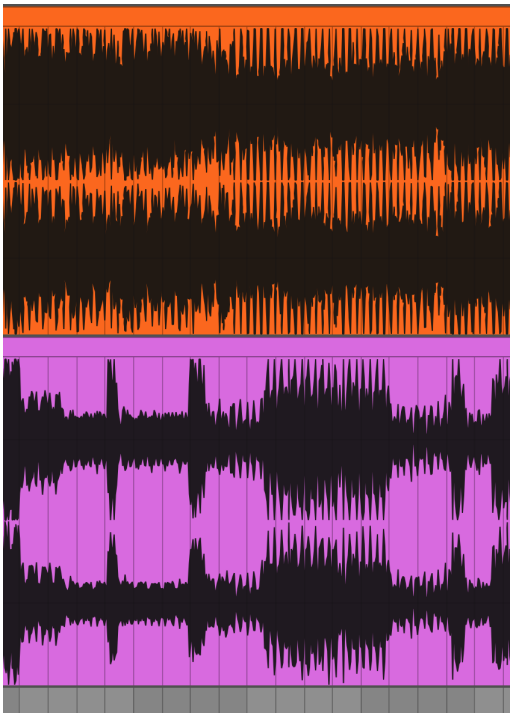

Figure 2.8: Ableton Live plugin attack, release, threshold and ratio tests. Demonstration of the test project, the control unaffected signal (orange) and output of the processed signal (pink).

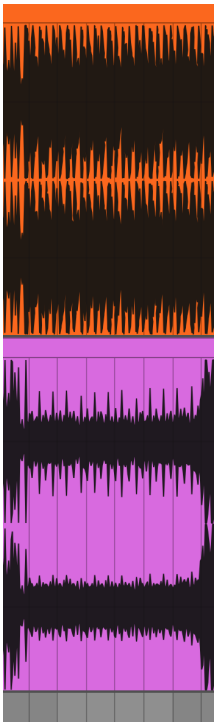

Figure 2.9: Ableton Live plugin mute and solo tests. Demonstration of the test project, the control unaffected signal (orange) and output of the processed signal (pink).

3. Conclusion

The exploration of a basic compressor algorithm within the JUCE framework using C++ has laid the groundwork for developing a multi-band compressor. Drawing from the works of Tarr, Schiermeyer, Pirkle, and others, the algorithm was expanded to segment the audio signal into multiple frequency bands, allowing for independent compression of each band. This approach is not without its challenges. Despite the comprehensive design and rigorous testing, discrepancies in the GUI display were observed, underscoring the need for further refinement (Appendix D and E).

Visualised in Figures 2.1, 2.3, 2.4, and 2.5, the testing phase provided valuable insights into the compressor's performance. Notably, the MATLAB prototyping of the multi-band compressor and the subsequent filtering processes demonstrated the algorithm's efficacy in manipulating audio signals. The visual representations, particularly the lowpass, highpass, and allpass filter responses, elucidated the impact of the filters on the audio signals, highlighting their potential in real-world applications.

However, the system's complexity, as evidenced by the detailed examination of the PluginProcessor.cpp, primary compressor.cpp, and Butterworth filter.cpp source files, suggests that a comprehensive understanding of the entire VST plugin's functionalities and processes might be overwhelming for some users. The intricate nature of compressors, with their nonlinear, time-dependent characteristics, further complicates the analysis. However, the testing conducted within the DAW environment offers a glimpse into the plugin's real-world performance, revealing its strengths and areas for improvement.

In summary, while the project showcases a promising start in developing a multi-band compressor, it underscores the complexities and challenges inherent in the project. The need for continuous refinement, testing, and user feedback is evident, ensuring that the final product meets the expectations of an audio engineer, and future development iterations would likely accomplish this.

References

Bennett, C. (2021). *Digital Audio Theory: A Practical Guide*. Abingdon, Oxon: Focal Press. pp.95-103, pp.109-124, pp.147-170, pp.205-214.

Bristow-Johnson, R. (2005). *RBJ Audio-EQ-Cookbook*. [Online]. musicdsp.org. Last Updated: 04 May 2005. Available at: https://www.musicdsp.org/en/latest/Filters/197-rbj-audio-eq-cookbook.html [Accessed 17 April 2023].

Falco, V. (2009). *Biquad, Butterworth, Chebyshev N-order, M-channel optimized filters*. [Online]. https://www.musicdsp.org/. Last Updated: 16 November 2009. Available at: https://www.musicdsp.org/en/latest/Filters/276-biquad-butterworth-chebyshev-n-order-m-channel-optimi [Accessed 20 July 2023].

Hodge, J. (2019). *Juce Tutorial 44- Audio Processor Value Tree State Update 2018 (Creating Plug-in Parameters)*. [Online]. YouTube.com. Last Updated: 2 Jan 2019. Available at: https://www.youtube.com/watch?v=NE8d91yYBJ8&list=PLLgJJsrdwhPxa6-02-CeHW8ocwSwl2jnu&index=47&ab_chan [Accessed 12 July 2023].

Lyons, R. (2010). *Understanding Digital Signal Processing*. 3rd ed. London: Pearson Education, Inc. pp.169-235, pp.893-902.

neotec. (2007). *Butterworth Optimized C++ Class*. [Online]. musicdsp.org. Last Updated: 20 January 2007. Available at: https://www.musicdsp.org/en/latest/Filters/243-butterworth-optimized-c-class.html [Accessed 20 July 2023].

Pirkle, W. (2012). *Designing Audio Effect Plug-Ins in C++: With Digital Audio Signal Processing Theory*. Abingdon, Oxon: Focal Press. pp.1-20, pp.453-487.

Pirkle, W. (2019). *Designing Audio Effect Plugins in C++: For AAX, AU, and VST3 With DSP Theory*. 2nd ed. Abingdon, Oxon: Routledge. pp.247, pp.430, pp.510-534.

Reiss, J. and McPherson, A. (2015). *Audio Effects: Theory, Implementation and Application*. Boca Raton, Florida: CRC Press. pp.59-88, pp.125-135, pp.141-167, pp.307-338.

Schiermeyer, C. (2021a). *Learn Modern C++ by Building an Audio Plugin (w/ JUCE Framework) - Full Course*. [Online]. YouTube. Last Updated: 20 May 2021. Available at: https://www.youtube.com/watch?v=i_Iq4_Kd7Rc&list=PLWKjhJtqVAbmUE5IqyfGYEYjrZBYzaT4m&index=4&t=1s&ab_ [Accessed 17 June 2023].

Schiermeyer, C. (2021b). *C++ Programming Tutorial - Build a 3-Band Compressor Audio Plugin (w/ JUCE Framework)*. [Online]. YouTube. Last Updated: 15 Nov 2021. Available at: <https://www.youtube.com/watch?v=Mo0Oco3Vimo&ab_channel=freeCodeCamp.org> [Accessed 20 July 2023].

Tarr, E. (2019). *Hack Audio: An Introduction to Computer Programming and Digital Signal Processing in MATLAB*. Abingdon, Oxon: Focal Press. pp.336-337, pp.409-454.

Tarr, E. (2020a). JUCE Tutorial. [Online]. hackaudio.com. Last Updated: 2020. Available at: <https://www.hackaudio.com/> [Accessed 10 July 2023].

Tarr, E. (2020b). *[2] Algorithmic Reverb Graphical User Interface*. [Online]. patreon.com. Last Updated: 11 April 2020. Available at: https://www.patreon.com/posts/2-algorithmic-35682889 [Accessed 10 July 2023].

Winer, E. (2018). *The Audio Expert*. 2nd ed. New York: Routledge. pp.371-396.

Zolzer, U. (2011). *DAFX: Digital Audio Effects*. 2nd ed. Chichester: John Wiley & Sons, Ltd. pp.48-57, pp.101-137.

Bibliography

Bennett, C. (2021). *Digital Audio Theory: A Practical Guide*. Abingdon, Oxon: Focal Press. pp.109-124, pp.147-170.

Bristow-Johnson, R. (2005). *RBJ Audio-EQ-Cookbook*. [Online]. musicdsp.org. Last Updated: 04 May 2005. Available at: https://www.musicdsp.org/en/latest/Filters/197-rbj-audio-eq-cookbook.html [Accessed 17 April 2023].

Falco, V. (2009). *Biquad, Butterworth, Chebyshev N-order, M-channel optimized filters*. [Online]. https://www.musicdsp.org/. Last Updated: 16 November 2009. Available at: https://www.musicdsp.org/en/latest/Filters/276-biquad-butterworth-chebyshev-n-order-m-channel-optimi [Accessed 20 July 2023].

Hashimoto, M. (2023). *My Approach to Building Large Technical Projects*. [Online]. mitchellh.com. Last Updated: 1 June 2023. Available at: https://mitchellh.com/writing/building-large-technical-projects [Accessed 3 June 2023].

Hill, M. (2021a). *Many More Much Smaller Steps – First Sketch*. [Online]. www.geepawhill.org. Last Updated: 29 Sept 2021. Available at: <https://www.geepawhill.org/2021/09/29/many-more-much-smaller-steps-first-sketch/> [Accessed 5 July2023].

Hill, M. (2021b). *MMMSS – A Closer Look at Steps*. [Online]. www.geepawhill.org. Last Updated: 26 Oct 2021. Available at: <https://www.geepawhill.org/2021/10/26/mmmss-a-closer-look-at-steps/> [Accessed 5 July2023].

Hodge, J. (2019). *Juce Tutorial 44- Audio Processor Value Tree State Update 2018 (Creating Plug-in Parameters)*. [Online]. YouTube.com. Last Updated: 2 Jan 2019. Available at: https://www.youtube.com/watch?v=NE8d91yYBJ8&list=PLLgJJsrdwhPxa6-02-CeHW8ocwSwl2jnu&index=47&ab_chan [Accessed 12 July 2023].

Lippman, S; Lajoie, J; Moo, B. (2012). *C++ Primer*. 5th ed. Massachusetts, USA: Addison-Wesley Professional. Pp.37-46, pp.47-51, pp.121-133, pp.332-360, pp.389-400.

Lyons, R. (2010). *Understanding Digital Signal Processing*. 3rd ed. London: Pearson Education, Inc. pp.169-235.

neotec. (2007). *Butterworth Optimized C++ Class*. [Online]. musicdsp.org. Last Updated: 20 January 2007. Available at: https://www.musicdsp.org/en/latest/Filters/243-butterworth-optimized-c-class.html [Accessed 20 July 2023].

Pirkle, W. (2012). *Designing Audio Effect Plug-Ins in C++: With Digital Audio Signal Processing Theory*. Abingdon, Oxon: Focal Press. pp.1-20.

Pirkle, W. (2019). *Designing Audio Effect Plugins in C++: For AAX, AU, and VST3 With DSP Theory*. 2nd ed. Abingdon, Oxon: Routledge. pp.32-37, pp.167-177, pp.178-193, pp.194-256, pp.384-415, pp.416-433

Reiss, J. and McPherson, A. (2015). *Audio Effects: Theory, Implementation and Application*. Boca Raton, Florida: CRC Press. pp.21-58, pp.59-88.

Schiermeyer, C. (2021a). *Learn Modern C++ by Building an Audio Plugin (w/ JUCE Framework)*. [Online]. YouTube. Last Updated: 20 May 2021. Available at: <https://www.youtube.com/watch?v=i_Iq4_Kd7Rc&pp=ygUeZnJlZWNvZGVjYW1wIGF1ZGlvIHByb2dyYW1taW5n> [Accessed 23 July2023].

Schiermeyer, C. (2021b). *C++ Programming Tutorial - Build a 3-Band Compressor Audio Plugin (w/ JUCE Framework)*. [Online]. YouTube. Last Updated: 15 Nov 2021. Available at: <https://www.youtube.com/watch?v=Mo0Oco3Vimo> [Accessed 23 July2023].

Stroustrup, B. (2000). *The C++ Programming Language: Special Edition*. 3rd ed. Indianapolis, IN: Addison-Wesley Professional. pp.69-75, pp.87-100, pp.108-125, pp.143-152, pp.197-210, pp.223-236, pp.431-434, pp.441-446, pp.579-584.

Tarr, E. (2019). *Hack Audio: An Introduction to Computer Programming and Digital Signal Processing in MATLAB*. Abingdon, Oxon: Focal Press. pp.183-204, pp.273-291, pp.293-347.

Tarr, E. (2020a). JUCE Tutorial. [Online]. hackaudio.com. Last Updated: 2020. Available at: <https://www.hackaudio.com/> [Accessed 10 July 2023].

Tarr, E. (2020b). *[2] Algorithmic Reverb Graphical User Interface*. [Online]. patreon.com. Last Updated: 11 April 2020. Available at: https://www.patreon.com/posts/2-algorithmic-35682889 [Accessed 10 July 2023].

Winer, E. (2018). *The Audio Expert*. 2nd ed. New York: Routledge. Pp.371-396.

Zolzer, U. (2011). *DAFX: Digital Audio Effects*. 2nd ed. Chichester: John Wiley & Sons, Ltd. pp.48-57, pp.101-137.

Appendix A - basicComp.m

The script was adapted and referenced from two sources: Reiss and McPherson (2015) and Tarr (2019). The script loads an input audio file named 'audioLoop1.wav' and prepares the necessary variables for processing. The input audio file is assumed to be stereo, and the script takes the mean of the two channels to create a mono audio signal. Parameters for the compressor are initialised, including the attack time constant (tauAttack_), release time constant (tauRelease_), threshold (in dB), compression ratio, and makeup gain (in dB).

The compression algorithm is then applied using a loop that processes each sample of the input audio:

Level detection: The level of the current sample is estimated using a peak detector and stored in x_g.

Gain computer: The output gain level y_g is determined based on the input level x_g, the threshold, and the compression ratio.

Ballistics: The gain reduction level y_l is smoothed based on the attack and release time constants and the previous gain reduction level yL_prev.

Control calculation: The control value c is calculated based on the makeup gain and the gain reduction level y_l.

The calculated control voltage c is applied to the input audio signal to obtain the output signal. The script creates a time vector t for plotting purposes. It specifies a zoomed region of interest in the input and output signals using the zoomStart_s and zoomEnd_s variables, representing the start and end times in seconds. The input and output signals are plotted in separate subplots along with the control signal in a zoomed view of the specified region. Overall, this script demonstrates a basic compressor, a commonly used audio processing technique that reduces the dynamic range of an audio signal by applying gain adjustments based on the signal level. It is used to control the loudness variations in audio recordings and is widely used in audio production, broadcasting, and music processing.

```
%% Basic compressor
%
% This script was adapted and referenced from Reiss and
McPherson (2015) and Tarr (2019).
% Please refer to accompanying report for full reference
list and details.
% Oberon Day-West (21501990).
%%
clc; clear;
% Prepare input signal
[inputBuffer, Fs] = audioread('audioLoop1.wav');
inputBuffer = mean(inputBuffer, 4);
bufferSize = length(inputBuffer);
samplerate = 44100; % Sample rate
% Initialize parameters
yL_prev = 0;
x_g = zeros(bufferSize, 1);
x_l = zeros(bufferSize, 1);
y_g = zeros(bufferSize, 1);
y_l = zeros(bufferSize, 1);
```

```
c = zeros(bufferSize, 1);
threshold_ = -10; % Set threshold (db)
ratio_ = 20; % Set ratio (10 = 10:1)
tauAttack_ = 2000; % Set attack time constant (ms)
tauRelease_ = 6000; % Set release time constant (ms)
makeUpGain_ = 0; % Set makeup gain (dB)
% Compression: calculates the control voltage
alphaAttack = exp(-1/(tauAttack_ * samplerate / 1000));
alphaRelease = exp(-1/(tauRelease_ * samplerate / 1000));
for i = 1:bufferSize
% Level detection- estimate level using peak detector
if abs(inputBuffer(i)) < 0.000001
x_g(i) = -120;
else
x_g(i) = 20*log10(abs(inputBuffer(i)));
end
% Gain computer - static apply input/output curve
if x_g(i) >= threshold_
y_g(i) = threshold_ + (x_g(i)-threshold_) / ratio_;
else
y_g(i) = x_g(i);
end
x_l(i) = x_g(i) - y_g(i);
% Ballistics - smoothing of the gain
if x_l(i) > yL_prev
y_l(i) = alphaAttack * yL_prev + (1 - alphaAttack) *
x_l(i);
else
y_l(i) = alphaRelease * yL_prev + (1 - alphaRelease) *
x_l(i);
end
% Find control
c(i) = 10^((makeUpGain_ - y_l(i)) / 20);
yL_prev = y_l(i);
end
% Apply control voltage to the audio signal
outputBuffer = inputBuffer .* c;
% Create time vector for plotting
t = (1:bufferSize) / samplerate;
% Specify zoom start and end in seconds
zoomStart_s = 30; % start at 30 seconds
zoomEnd_s = 40; % end at 40 seconds
% Convert to sample indices
```

```
zoomStart = round(zoomStart_s * samplerate);
zoomEnd = round(zoomEnd_s * samplerate);
% Plot results
figure
subplot(4,1,1);
plot(t, inputBuffer);
title('Input Signal');
xlabel('Time (s)');
ylabel('Amplitude');
subplot(4,1,2);
plot(t, outputBuffer);
title('Output Signal');
xlabel('Time (s)');
ylabel('Amplitude');
subplot(4,1,3);
plot(t(zoomStart:zoomEnd), inputBuffer(zoomStart:zoomEnd));
hold on;
plot(t(zoomStart:zoomEnd), c(zoomStart:zoomEnd), 'r');
title('Zoomed Input and Control Signal');
xlabel('Time (s)');
ylabel('Amplitude');
legend('Input','Control');
subplot(4,1,4);
plot(t(zoomStart:zoomEnd),
outputBuffer(zoomStart:zoomEnd));
title('Zoomed Output Signal');
xlabel('Time (s)');
ylabel('Amplitude');
```

Figure A.1: basicComp.m.

Appendix B - pluginProcessor.cpp

The provided code pertains to the implementation of an audio processing plugin, specifically a multiband compressor, using the JUCE framework. The code begins by including necessary header files for the plugin's processor, editor, a basic compressor, and a Butterworth filter.

The `One_MBCompAudioProcessor` class is defined, which inherits from the `AudioProcessor` class provided by JUCE. The constructor of this class initialises various audio processing parameters and sets up the audio buses for input and output. It also initialises multiple filters with the current sample rate.

Within the constructor, helper lambda functions (`floatHelper`, `choiceHelper`, and `boolHelper`) are defined to facilitate the dynamic casting of parameters and ensure their correct type. These functions are then utilised to initialise various parameters of the low, mid, and high band compressors, such as attack time, release time, threshold level, ratio, bypass, mute, and solo settings.

The `prepareToPlay` method is responsible for initialising various components before audio playback begins. It sets up the processing specifications, prepares the compressors and filters, and allocates memory for audio buffers.

The `processBlock` method is the core of the audio processing routine. It processes incoming audio buffers, applies the multiband compression, and outputs the processed audio. The method ensures that denormalized numbers, which can degrade performance, are disabled. It also updates compressor settings, applies input and output gains, and processes the audio through the defined filters.

The `hasEditor` method indicates that the plugin has a graphical user interface, and the `createEditor` method returns a new instance of the plugin's editor.

The `getStateInformation` and `setStateInformation` methods are responsible for saving and loading the plugin's state, respectively. This is essential for recalling user-defined settings when the plugin is reopened.

Lastly, the `createParameterLayout` method defines the layout of the plugin's parameters, specifying their types, ranges, and default values. This layout is crucial for the plugin's graphical user interface, allowing users to interact with and adjust the plugin's settings.

```
/*


==============================================================================
    This file contains the basic framework code for a JUCE plugin processor.
    Oberon Day-West (2023). #21501990.
    This code has been referenced and adapted from Schiermeyer (2021a; 2021b), Pirkle (2019) and Tarr
(2019).
    Please refer to the accompanying report for full list of references.


==============================================================================
*/
#include "PluginProcessor.h"
#include "PluginEditor.h"
#include "BasicCompressor.h"
#include "butterworthFilter.h"
//==============================================================================
=
One_MBCompAudioProcessor::One_MBCompAudioProcessor()
#ifndef JucePlugin_PreferredChannelConfigurations
    : AudioProcessor (BusesProperties()
                     #if ! JucePlugin_IsMidiEffect
                      #if ! JucePlugin_IsSynth
                       .withInput  ("Input",  juce::AudioChannelSet::stereo(), true)
```

```
                #endif
                 .withOutput ("Output", juce::AudioChannelSet::stereo(), true)
                #endif
                 ),
LP1(getSampleRate()), AP2(getSampleRate()), HP1(getSampleRate()), LP2(getSampleRate()),
HP2(getSampleRate())
#endif
{
   using namespace PluginParameters;
   const auto& parameters = GetParameters();

   auto floatHelper = [&apvts = this->apvts, &parameters](auto& parameter, const auto& ParamNames)
   {
      parameter =
dynamic_cast<juce::AudioParameterFloat*>(apvts.getParameter(parameters.at(ParamNames)));
      jassert( parameter != nullptr );
   };

   floatHelper(low_BandCompressor.attackTime,      ParamNames::Attack_LB);
   floatHelper(low_BandCompressor.releaseTime,     ParamNames::Release_LB);
   floatHelper(low_BandCompressor.thresholdLevel,  ParamNames::Threshold_LB);

   floatHelper(mid_BandCompressor.attackTime,      ParamNames::Attack_MB);
   floatHelper(mid_BandCompressor.releaseTime,     ParamNames::Release_MB);
   floatHelper(mid_BandCompressor.thresholdLevel,  ParamNames::Threshold_MB);

   floatHelper(high_BandCompressor.attackTime,     ParamNames::Attack_HB);
   floatHelper(high_BandCompressor.releaseTime,    ParamNames::Release_HB);
   floatHelper(high_BandCompressor.thresholdLevel, ParamNames::Threshold_HB);
   auto choiceHelper = [&apvts = this->apvts, &parameters](auto& parameter, const auto& ParamNames)
   {
      parameter =
dynamic_cast<juce::AudioParameterChoice*>(apvts.getParameter(parameters.at(ParamNames)));
      jassert( parameter != nullptr );
   };

   choiceHelper(low_BandCompressor.ratio,  ParamNames::Ratio_LB);
   choiceHelper(mid_BandCompressor.ratio,  ParamNames::Ratio_MB);
   choiceHelper(high_BandCompressor.ratio, ParamNames::Ratio_HB);


   auto boolHelper = [&apvts = this->apvts, &parameters](auto& parameter, const auto& ParamNames)
   {
      parameter =
dynamic_cast<juce::AudioParameterBool*>(apvts.getParameter(parameters.at(ParamNames)));
      jassert( parameter != nullptr );
   };

   boolHelper(low_BandCompressor.bypassed, ParamNames::Bypass_LB);
   boolHelper(mid_BandCompressor.bypassed, ParamNames::Bypass_MB);
   boolHelper(high_BandCompressor.bypassed, ParamNames::Bypass_HB);
```

```
  boolHelper(low_BandCompressor.mute, ParamNames::Mute_LB);
  boolHelper(mid_BandCompressor.mute, ParamNames::Mute_MB);
  boolHelper(high_BandCompressor.mute, ParamNames::Mute_HB);

  boolHelper(low_BandCompressor.solo, ParamNames::Solo_LB);
  boolHelper(mid_BandCompressor.solo, ParamNames::Solo_MB);
  boolHelper(high_BandCompressor.solo, ParamNames::Solo_HB);

  floatHelper(lowMidFreqXover, ParamNames::Low_Mid_XO_Frequency);
  floatHelper(midHighFreqXover, ParamNames::Mid_High_XO_Frequency);

  floatHelper(inputGainParameter, ParamNames::Gain_Input);
  floatHelper(outputGainParameter, ParamNames::Gain_Output);

  LP1.setType(FilterType::lowpass);
  HP1.setType(FilterType::highpass);
  AP2.setType(FilterType::allpass);
  LP2.setType(FilterType::lowpass);
  HP2.setType(FilterType::highpass);
}
One_MBCompAudioProcessor::~One_MBCompAudioProcessor()
{
}
//==============================================================================
=
const juce::String One_MBCompAudioProcessor::getName() const
{
   return JucePlugin_Name;
}
bool One_MBCompAudioProcessor::acceptsMidi() const
{
  #if JucePlugin_WantsMidiInput
   return true;
  #else
   return false;
  #endif
}
bool One_MBCompAudioProcessor::producesMidi() const
{
  #if JucePlugin_ProducesMidiOutput
   return true;
  #else
   return false;
  #endif
}
bool One_MBCompAudioProcessor::isMidiEffect() const
{
  #if JucePlugin_IsMidiEffect
   return true;
  #else
```

```
    return false;
  #endif
}
double One_MBCompAudioProcessor::getTailLengthSeconds() const
{
    return 0.0;
}
int One_MBCompAudioProcessor::getNumPrograms()
{
    return 1;   // NB: some hosts don't cope very well if you tell them there are 0 programs,
                // so this should be at least 1, even if you're not really implementing programs.
}
int One_MBCompAudioProcessor::getCurrentProgram()
{
    return 0;
}
void One_MBCompAudioProcessor::setCurrentProgram (int index)
{
}
const juce::String One_MBCompAudioProcessor::getProgramName (int index)
{
    return {};
}
void One_MBCompAudioProcessor::changeProgramName (int index, const juce::String& newName)
{
}
//==============================================================================
void One_MBCompAudioProcessor::prepareToPlay (double sampleRate, int samplesPerBlock)
{
    // Use this method as the place to do any pre-playback
    // initialisation that you need..

    juce::dsp::ProcessSpec spec;
    spec.maximumBlockSize = samplesPerBlock;
    spec.numChannels = getTotalNumOutputChannels();
    spec.sampleRate = sampleRate;

    for( auto& comp : compressors )
    {
        comp.prepareComp(spec);
    }

    LP1.prepare(spec);
    HP1.prepare(spec);
    AP2.prepare(spec);
    LP2.prepare(spec);
    HP2.prepare(spec);

    inputGain.prepare(spec);
    outputGain.prepare(spec);
```

```
    inputGain.setRampDurationSeconds(0.05); // ms
    outputGain.setRampDurationSeconds(0.05);

    for( auto& buffer : filterBuffers )
    {
        buffer.setSize(spec.numChannels, samplesPerBlock);
    }

    leftChannelFifo.prepare(samplesPerBlock);
    rightChannelFifo.prepare(samplesPerBlock);
}
void One_MBCompAudioProcessor::releaseResources()
{
    // When playback stops, you can use this as an opportunity to free up any
    // spare memory, etc.
}
#ifndef JucePlugin_PreferredChannelConfigurations
bool One_MBCompAudioProcessor::isBusesLayoutSupported (const BusesLayout& layouts) const
{
  #if JucePlugin_IsMidiEffect
    juce::ignoreUnused (layouts);
    return true;
  #else
    // This is the place where you check if the layout is supported.
    // In this template code we only support mono or stereo.
    // Some plugin hosts, such as certain GarageBand versions, will only
    // load plugins that support stereo bus layouts.
    if (layouts.getMainOutputChannelSet() != juce::AudioChannelSet::mono()
     && layouts.getMainOutputChannelSet() != juce::AudioChannelSet::stereo())
        return false;
    // This checks if the input layout matches the output layout
   #if ! JucePlugin_IsSynth
    if (layouts.getMainOutputChannelSet() != layouts.getMainInputChannelSet())
        return false;
   #endif
    return true;
  #endif
}
#endif
void One_MBCompAudioProcessor::processBlock (juce::AudioBuffer<float>& buffer, juce::MidiBuffer&
midiMessages)
{
    // juce::ScopedNoDenormals disables denormalised numbers, which can be a source of
    // performance issues in audio processing.
    juce::ScopedNoDenormals noDenormals;

    // Get the total number of input and output channels
    auto totalNumInputChannels  = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();
    // This loop clears any output channels that didn't contain input data
```

```
for (auto i = totalNumInputChannels; i < totalNumOutputChannels; ++i)
    buffer.clear (i, 0, buffer.getNumSamples());

// Loop through all compressors and update their parameter settings
for( auto& compressor : compressors )
{
    compressor.updateCompressorParamSettings();
}

// Set the input and output gain values in decibels
inputGain.setGainDecibels( inputGainParameter->get() );
outputGain.setGainDecibels( outputGainParameter->get() );

leftChannelFifo.update(buffer);
rightChannelFifo.update(buffer);

// Apply the gain to the buffer
applyGain(buffer, inputGain);

// Assign the contents of buffer to all elements in the filterBuffers vector
for( auto& filter_buffer : filterBuffers )
{
    filter_buffer = buffer;
}

// Get the crossover frequencies for the filters
auto filter_lowMidCutoff = lowMidFreqXover->get();
auto filter_midHighCutoff = midHighFreqXover->get();

// Set the cutoff frequencies for the filters
LP1.setCrossoverFrequency(filter_lowMidCutoff);
HP1.setCrossoverFrequency(filter_lowMidCutoff);
AP2.setCrossoverFrequency(filter_midHighCutoff);
LP2.setCrossoverFrequency(filter_midHighCutoff);
HP2.setCrossoverFrequency(filter_midHighCutoff);
// Create AudioBlocks from the filterBuffers
auto filter_bufferBlock0 = juce::dsp::AudioBlock<float>(filterBuffers[0]);
auto filter_bufferBlock1 = juce::dsp::AudioBlock<float>(filterBuffers[1]);
auto filter_bufferBlock2 = juce::dsp::AudioBlock<float>(filterBuffers[2]);

// Create ProcessContexts from the AudioBlocks
auto filter_bufferContext0 = juce::dsp::ProcessContextReplacing<float>(filter_bufferBlock0);
auto filter_bufferContext1 = juce::dsp::ProcessContextReplacing<float>(filter_bufferBlock1);
auto filter_bufferContext2 = juce::dsp::ProcessContextReplacing<float>(filter_bufferBlock2);

// Process the filters in a certain order (flow)
LP1.process(filter_bufferContext0);
AP2.process(filter_bufferContext0);
HP1.process(filter_bufferContext1);
filterBuffers[2] = filterBuffers[1];
LP2.process(filter_bufferContext1);
```

```
    HP2.process(filter_bufferContext2);

    // Loop through all compressors and apply them to the corresponding filter buffer
    for( size_t i = 0; i < filterBuffers.size(); ++i )
    {
        compressors[i].process(filterBuffers[i]);
    }

    // Clear the buffer
    buffer.clear();

    // A function that adds a source buffer to the input buffer
    auto addFilterBand = [numberChannels = buffer.getNumChannels(), numberSamples =
buffer.getNumSamples()]( auto& inputBuffer, const auto& source )
    {
        for( auto i = 0; i < numberChannels; ++i )
        {
            inputBuffer.addFrom(i, 0, source, i, 0, numberSamples);
        }
    };

    // Check if any compressors are soloed
    auto bandsAreSoloed = false;
    for( auto& comp : compressors )
    {
        if( comp.solo->get() )
        {
            bandsAreSoloed = true;
            break;
        }
    }

    // If any compressors are soloed, add only those filter bands to the buffer
    // Otherwise, add all unmuted filter bands to the buffer
    if( bandsAreSoloed )
    {
        for( size_t i = 0; i < compressors.size(); ++i )
        {
            auto& comp = compressors[i];
            if( comp.solo->get() )
            {
                addFilterBand(buffer, filterBuffers[i]);
            }
        }
    }
    else
    {
        for( size_t i = 0; i < compressors.size(); ++i )
        {
            auto& comp = compressors[i];
            if( ! comp.mute->get() )
```

```
            {
                addFilterBand(buffer, filterBuffers[i]);
            }
        }
    }

    // Apply the output gain to the buffer
    applyGain(buffer, outputGain);
}
//==============================================================================
bool One_MBCompAudioProcessor::hasEditor() const
{
    return true; // (change this to false if you choose to not supply an editor)
}
juce::AudioProcessorEditor* One_MBCompAudioProcessor::createEditor()
{
    return new One_MBCompAudioProcessorEditor (*this);
//    return new juce::GenericAudioProcessorEditor(*this);
}
//==============================================================================
void One_MBCompAudioProcessor::getStateInformation (juce::MemoryBlock& destData)
{
    // You should use this method to store your parameters in the memory block.
    // You could do that either as raw data, or use the XML or ValueTree classes
    // as intermediaries to make it easy to save and load complex data.
    juce::MemoryOutputStream memoryOutputStream(destData, true);
    apvts.state.writeToStream(memoryOutputStream);
}
void One_MBCompAudioProcessor::setStateInformation (const void* data, int sizeInBytes)
{
    // You should use this method to restore your parameters from this memory block,
    // whose contents will have been created by the getStateInformation() call.
    auto savedTree = juce::ValueTree::readFromData(data, sizeInBytes);
    if( savedTree.isValid() )
    {
        apvts.replaceState(savedTree);
    }
}
juce::AudioProcessorValueTreeState::ParameterLayout
One_MBCompAudioProcessor::createParameterLayout()
{
    // Define the layout for the plugin's parameters.
    APVTS::ParameterLayout PluginGUIlayout;

    // Import namespaces for JUCE and the parameters used in this plugin.
    using namespace juce;
    using namespace PluginParameters;
    const auto& parameters = GetParameters(); // Get the parameters defined for the plugin.
```

```
  // ===== Gain parameters
  // Define the range for the Gain parameters, -24 to 24 with steps of 0.5.
  auto gainRangeValues = NormalisableRange<float>(-24.f, 24.f, 0.5f, 1.f);

  // Add Gain Input and Output parameters to the layout.
  PluginGUIlayout.add(std::make_unique<AudioParameterFloat>(parameters.at(ParamNames::Gain_Input),
                                        parameters.at(ParamNames::Gain_Input),
                                        gainRangeValues,
                                        0));


PluginGUIlayout.add(std::make_unique<AudioParameterFloat>(parameters.at(ParamNames::Gain_Output),
                                        parameters.at(ParamNames::Gain_Output),
                                        gainRangeValues,
                                        0));

  // ===== Threshold parameters
  // Add threshold parameters to the layout.

PluginGUIlayout.add(std::make_unique<AudioParameterFloat>(parameters.at(ParamNames::Threshold_LB)
,
                                        parameters.at(ParamNames::Threshold_LB),
                                        NormalisableRange<float>(-60, 12, 1, 1),
                                        0));


PluginGUIlayout.add(std::make_unique<AudioParameterFloat>(parameters.at(ParamNames::Threshold_MB
),
                                        parameters.at(ParamNames::Threshold_MB),
                                        NormalisableRange<float>(-60, 12, 1, 1),
                                        0));


PluginGUIlayout.add(std::make_unique<AudioParameterFloat>(parameters.at(ParamNames::Threshold_HB)
,
                                        parameters.at(ParamNames::Threshold_HB),
                                        NormalisableRange<float>(-60, 12, 1, 1),
                                        0));

  // Define the range for the Attack and Release parameters, 5 to 500 with steps of 1.
  auto attkRelRange = NormalisableRange<float>(5, 500, 1, 1);

  // ===== Attack parameters
  // Add attack parameters to the layout.
  PluginGUIlayout.add(std::make_unique<AudioParameterFloat>(parameters.at(ParamNames::Attack_LB),
                                        parameters.at(ParamNames::Attack_LB),
                                        attkRelRange,
                                        50));

  PluginGUIlayout.add(std::make_unique<AudioParameterFloat>(parameters.at(ParamNames::Attack_MB),
                                        parameters.at(ParamNames::Attack_MB),
```

```
                                        attkRelRange,
                                        50));

    PluginGUIlayout.add(std::make_unique<AudioParameterFloat>(parameters.at(ParamNames::Attack_HB),
                                        parameters.at(ParamNames::Attack_HB),
                                        attkRelRange,
                                        50));
    // ===== Release parameters
    // Add release parameters to the layout.

PluginGUIlayout.add(std::make_unique<AudioParameterFloat>(parameters.at(ParamNames::Release_LB),
                                        parameters.at(ParamNames::Release_LB),
                                        attkRelRange,
                                        250));


PluginGUIlayout.add(std::make_unique<AudioParameterFloat>(parameters.at(ParamNames::Release_MB),
                                        parameters.at(ParamNames::Release_MB),
                                        attkRelRange,
                                        250));


PluginGUIlayout.add(std::make_unique<AudioParameterFloat>(parameters.at(ParamNames::Release_HB),
                                        parameters.at(ParamNames::Release_HB),
                                        attkRelRange,
                                        250));

    // Define the choices for the Ratio parameter as a vector of doubles.
    auto ratioChoices = std::vector<double>{ 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 8, 10, 15, 20, 50 };

    // Convert the ratio choices to a StringArray so they can be used with an AudioParameterChoice.
    juce::StringArray strArr;
    for( auto rChoice : ratioChoices )
    {
        strArr.add( juce::String(rChoice, 1) );
    }

    // ===== Ratio parameters
    // Add ratio parameters to the layout.
    PluginGUIlayout.add(std::make_unique<AudioParameterChoice>(parameters.at(ParamNames::Ratio_LB),
                                        parameters.at(ParamNames::Ratio_LB), strArr, 3));


PluginGUIlayout.add(std::make_unique<AudioParameterChoice>(parameters.at(ParamNames::Ratio_MB),
                                        parameters.at(ParamNames::Ratio_MB), strArr, 3));


PluginGUIlayout.add(std::make_unique<AudioParameterChoice>(parameters.at(ParamNames::Ratio_HB),
                                        parameters.at(ParamNames::Ratio_HB), strArr, 3));
    // ===== Bypass parameters
    // Add bypass parameters to the layout.
```

```
    PluginGUIlayout.add(std::make_unique<AudioParameterBool>(parameters.at(ParamNames::Bypass_LB),
                                            parameters.at(ParamNames::Bypass_LB), false));


PluginGUIlayout.add(std::make_unique<AudioParameterBool>(parameters.at(ParamNames::Bypass_MB),
                                            parameters.at(ParamNames::Bypass_MB), false));

    PluginGUIlayout.add(std::make_unique<AudioParameterBool>(parameters.at(ParamNames::Bypass_HB),
                                            parameters.at(ParamNames::Bypass_HB), false));

    // ====== Mute parameters
    // Add mute parameters to the layout.
    PluginGUIlayout.add(std::make_unique<AudioParameterBool>(parameters.at(ParamNames::Mute_LB),
                                            parameters.at(ParamNames::Mute_LB), false));

    PluginGUIlayout.add(std::make_unique<AudioParameterBool>(parameters.at(ParamNames::Mute_MB),
                                            parameters.at(ParamNames::Mute_MB), false));

    PluginGUIlayout.add(std::make_unique<AudioParameterBool>(parameters.at(ParamNames::Mute_HB),
                                            parameters.at(ParamNames::Mute_HB), false));

    // ====== Solo parameters
    // Add solo parameters to the layout.
    PluginGUIlayout.add(std::make_unique<AudioParameterBool>(parameters.at(ParamNames::Solo_LB),
                                            parameters.at(ParamNames::Solo_LB), false));

    PluginGUIlayout.add(std::make_unique<AudioParameterBool>(parameters.at(ParamNames::Solo_MB),
                                            parameters.at(ParamNames::Solo_MB), false));

    PluginGUIlayout.add(std::make_unique<AudioParameterBool>(parameters.at(ParamNames::Solo_HB),
                                            parameters.at(ParamNames::Solo_HB), false));

    // Add Frequency parameters for Low-Mid Crossover and Mid-High Crossover to the layout.

PluginGUIlayout.add(std::make_unique<AudioParameterFloat>(parameters.at(ParamNames::Low_Mid_XO
_Frequency),
                                            parameters.at(ParamNames::Low_Mid_XO_Frequency),
                                             NormalisableRange<float>(20, 999, 1, 1), 400));


PluginGUIlayout.add(std::make_unique<AudioParameterFloat>(parameters.at(ParamNames::Mid_High_XO
_Frequency),
                                            parameters.at(ParamNames::Mid_High_XO_Frequency),
                                             NormalisableRange<float>(1000, 20000, 1, 1), 2000));
    // Return the completed layout.
    return PluginGUIlayout;
}
//==============================================================================
=
// This creates new instances of the plugin..
juce::AudioProcessor* JUCE_CALLTYPE createPluginFilter()
```

```
{
    return new One_MBCompAudioProcessor();
}
```

Figure B.1: PluginProcessor.cpp file.

Appendix C - pluginProcessor.h

The code commences with the inclusion of necessary headers and definitions of classes related to audio processing. Notably, a specific section of the code, explicitly referenced from Schiermeyer (2021a; 2021b), demonstrates the spectrum in the plugin. This segment has not been originally crafted by the author, and its boundaries are clearly demarcated for the reader's convenience.

Within the code, various structures and classes are defined:

Fifo: A template structure that manages a First-In-First-Out (FIFO) buffer.

SingleChannelSampleFifo: A template structure designed to handle single-channel audio samples.

PluginParameters: A namespace that enumerates various parameter names and provides a mapping between parameter names and their string representations.

CompressorBand: A structure that encapsulates the functionalities of a basic compressor, including its parameters and processing methods.

One_MBCompAudioProcessor: The primary class that represents the audio processor for the plugin. This class encompasses methods for audio processing, parameter management, and state information handling. It also integrates multiple instances of the CompressorBand structure to manage different frequency bands and applies various filters and gains to the audio signal.

The code concludes with the declaration of a non-copyable macro for the One_MBCompAudioProcessor class, ensuring that instances of this class cannot be copied inadvertently.

```
/*
  ==============================================================================
    This file contains the basic framework code for a JUCE plugin processor.
    Oberon Day-West (2023). #21501990.
    This code has been referenced and adapted from Schiermeyer (2021a; 2021b), Pirkle (2019) and Tarr (2019).
    Please refer to the accompanying report for full list of references.
  ==============================================================================
*/
#pragma once
#include <array>
#include <JuceHeader.h>
#include "BasicCompressor.h"
#include "butterworthFilter.h"
/*
  ==============================================================================
    ********************************************************************
    NOTE TO READER: This following section of code was explicitly referenced from
    Schiermeyer (2021a; 2021b) to demonstrate the spectrum in the plugin. This
    code block has not be originally created by the author. The end of the block
    will be clearly stated for ease of reference to the reader.
    ********************************************************************

  ==============================================================================
*/
```

```
template<typename T>
struct Fifo
{
   void prepare(int numChannels, int numSamples)
   {
      static_assert( std::is_same_v<T, juce::AudioBuffer<float>>,
                  "prepare(numChannels, numSamples) should only be used when the Fifo is holding
juce::AudioBuffer<float>");
      for( auto& buffer : buffers)
      {
         buffer.setSize(numChannels,
                        numSamples,
                        false,   //clear everything?
                        true,    //including the extra space?
                        true);   //avoid reallocating if you can?
         buffer.clear();
      }
   }

   void prepare(size_t numElements)
   {
      static_assert( std::is_same_v<T, std::vector<float>>,
                  "prepare(numElements) should only be used when the Fifo is holding std::vector<float>");
      for( auto& buffer : buffers )
      {
         buffer.clear();
         buffer.resize(numElements, 0);
      }
   }

   bool push(const T& t)
   {
      auto write = fifo.write(1);
      if( write.blockSize1 > 0 )
      {
         buffers[write.startIndex1] = t;
         return true;
      }

      return false;
   }

   bool pull(T& t)
   {
      auto read = fifo.read(1);
      if( read.blockSize1 > 0 )
      {
         t = buffers[read.startIndex1];
         return true;
      }
```

```
        return false;
    }

    int getNumAvailableForReading() const
    {
        return fifo.getNumReady();
    }
private:
    static constexpr int Capacity = 30;
    std::array<T, Capacity> buffers;
    juce::AbstractFifo fifo {Capacity};
};
enum Channel
{
    Right, //effectively 0
    Left //effectively 1
};
template<typename BlockType>
struct SingleChannelSampleFifo
{
    SingleChannelSampleFifo(Channel ch) : channelToUse(ch)
    {
        prepared.set(false);
    }

    void update(const BlockType& buffer)
    {
        jassert(prepared.get());
        jassert(buffer.getNumChannels() > channelToUse );
        auto* channelPtr = buffer.getReadPointer(channelToUse);

        for( int i = 0; i < buffer.getNumSamples(); ++i )
        {
            pushNextSampleIntoFifo(channelPtr[i]);
        }
    }
    void prepare(int bufferSize)
    {
        prepared.set(false);
        size.set(bufferSize);

        bufferToFill.setSize(1,             //channel
                             bufferSize,    //num samples
                             false,         //keepExistingContent
                             true,          //clear extra space
                             true);         //avoid reallocating
        audioBufferFifo.prepare(1, bufferSize);
        fifoIndex = 0;
        prepared.set(true);
    }
```

```
//==============================================================================
=
    int getNumCompleteBuffersAvailable() const { return audioBufferFifo.getNumAvailableForReading(); }
    bool isPrepared() const { return prepared.get(); }
    int getSize() const { return size.get(); }

//==============================================================================
=
    bool getAudioBuffer(BlockType& buf) { return audioBufferFifo.pull(buf); }
private:
    Channel channelToUse;
    int fifoIndex = 0;
    Fifo<BlockType> audioBufferFifo;
    BlockType bufferToFill;
    juce::Atomic<bool> prepared = false;
    juce::Atomic<int> size = 0;

    void pushNextSampleIntoFifo(float sample)
    {
        if (fifoIndex == bufferToFill.getNumSamples())
        {
            auto ok = audioBufferFifo.push(bufferToFill);
            juce::ignoreUnused(ok);

            fifoIndex = 0;
        }

        bufferToFill.setSample(0, fifoIndex, sample);
        ++fifoIndex;
    }
};
/*


    ********************************************************************************
                            END OF CODE BLOCK
                        (Schiermeyer, 2021a; 2021b)
    ********************************************************************************


*/
namespace PluginParameters
{
enum ParamNames
{
    Low_Mid_XO_Frequency,
    Mid_High_XO_Frequency,

    Threshold_LB,
    Threshold_MB,
    Threshold_HB,
```

```
    Attack_LB,
    Attack_MB,
    Attack_HB,

    Release_LB,
    Release_MB,
    Release_HB,

    Ratio_LB,
    Ratio_MB,
    Ratio_HB,

    Bypass_LB,
    Bypass_MB,
    Bypass_HB,

    Mute_LB,
    Mute_MB,
    Mute_HB,

    Solo_LB,
    Solo_MB,
    Solo_HB,

    Gain_Input,
    Gain_Output,
};
inline const std::map<ParamNames, juce::String>& GetParameters()
{
    static std::map<ParamNames, juce::String> parameters =
    {
        { Low_Mid_XO_Frequency, "Low-Mid Crossover Frequency" },
        { Mid_High_XO_Frequency, "Mid-High Crossover Frequency" },

        { Threshold_LB, "Low-Band Threshold" },
        { Threshold_MB, "Mid-Band Threshold" },
        { Threshold_HB, "High-Band Threshold" },

        { Attack_LB, "Low-Band Attack" },
        { Attack_MB, "Mid-Band Attack" },
        { Attack_HB, "High-Band Attack" },

        { Release_LB, "Low-Band Release" },
        { Release_MB, "Mid-Band Release" },
        { Release_HB, "High-Band Release" },

        { Ratio_LB, "Low-Band Ratio" },
        { Ratio_MB, "Mid-Band Ratio" },
        { Ratio_HB, "High-Band Ratio" },
```

```
        { Bypass_LB, "Low-Band Bypass" },
        { Bypass_MB, "Mid-Band Bypass" },
        { Bypass_HB, "High-Band Bypass" },

        { Mute_LB, "Low-Band Mute" },
        { Mute_MB, "Mid-Band Mute" },
        { Mute_HB, "High-Band Mute" },

        { Solo_LB, "Low-Band Solo" },
        { Solo_MB, "Mid-Band Solo" },
        { Solo_HB, "High-Band Solo" },

        { Gain_Input, "Gain Input" },
        { Gain_Output, "Gain Output" },
    };

    return parameters;
}
}
struct CompressorBand
{
private:
    BasicCompressor compressor;

public:
    juce::AudioParameterFloat* attackTime = nullptr;
    juce::AudioParameterFloat* releaseTime = nullptr;
    juce::AudioParameterFloat* thresholdLevel = nullptr;
    juce::AudioParameterChoice* ratio = nullptr;
    juce::AudioParameterBool* bypassed = nullptr;
    juce::AudioParameterBool* mute = nullptr;
    juce::AudioParameterBool* solo = nullptr;

    void prepareComp( const juce::dsp::ProcessSpec& spec )
    {
        compressor.prepare(spec);
    }

    void updateCompressorParamSettings()
    {
        compressor.setAttackTime( attackTime->get() );
        compressor.setReleaseTime( releaseTime->get() );
        compressor.setThresholdLevel( thresholdLevel->get() );
        compressor.setCompressionRatio( ratio->getCurrentChoiceName().getFloatValue() );
    }

    void process(juce::AudioBuffer<float>& buffer)
    {
        auto sampleBlock = juce::dsp::AudioBlock<float>(buffer);
        auto context = juce::dsp::ProcessContextReplacing<float>(sampleBlock);
```

```
        context.isBypassed = bypassed->get();

        compressor.process(context);
    }
};
//==============================================================================
/**
*/
class One_MBCompAudioProcessor  : public juce::AudioProcessor
{
public:

//==============================================================================
    One_MBCompAudioProcessor();
    ~One_MBCompAudioProcessor() override;

//==============================================================================
    void prepareToPlay (double sampleRate, int samplesPerBlock) override;
    void releaseResources() override;
   #ifndef JucePlugin_PreferredChannelConfigurations
    bool isBusesLayoutSupported (const BusesLayout& layouts) const override;
   #endif
    void processBlock (juce::AudioBuffer<float>&, juce::MidiBuffer&) override;

//==============================================================================
    juce::AudioProcessorEditor* createEditor() override;
    bool hasEditor() const override;

//==============================================================================
    const juce::String getName() const override;
    bool acceptsMidi() const override;
    bool producesMidi() const override;
    bool isMidiEffect() const override;
    double getTailLengthSeconds() const override;

//==============================================================================
    int getNumPrograms() override;
    int getCurrentProgram() override;
    void setCurrentProgram (int index) override;
    const juce::String getProgramName (int index) override;
    void changeProgramName (int index, const juce::String& newName) override;

//==============================================================================
    void getStateInformation (juce::MemoryBlock& destData) override;
```

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

    // implemented short-hand object access
    using APVTS = juce::AudioProcessorValueTreeState;
    static APVTS::ParameterLayout createParameterLayout();

    APVTS apvts { *this, nullptr, "Parameters", createParameterLayout() };

    using BlockType = juce::AudioBuffer<float>;
    SingleChannelSampleFifo<BlockType> leftChannelFifo{Channel::Left};
    SingleChannelSampleFifo<BlockType> rightChannelFifo{Channel::Right};
private:
    std::array<CompressorBand, 3> compressors;
    CompressorBand& low_BandCompressor = compressors[0];
    CompressorBand& mid_BandCompressor = compressors[1];
    CompressorBand& high_BandCompressor = compressors[2];

//    LinkwitzRiley LPF, HPF;
    using Filters = LinkwitzRFilter;
    //      FC0     FC1
    Filters LP1,    AP2,
            HP1,    LP2,
                    HP2;

    juce::AudioParameterFloat* lowMidFreqXover { nullptr };
    juce::AudioParameterFloat* midHighFreqXover { nullptr };

    std::array<juce::AudioBuffer<float>, 3> filterBuffers;

    juce::dsp::Gain<float> inputGain, outputGain;
    juce::AudioParameterFloat* inputGainParameter { nullptr };
    juce::AudioParameterFloat* outputGainParameter { nullptr };

    template<typename T, typename U>
    void applyGain(T& buffer, U& gain)
    {
        auto block = juce::dsp::AudioBlock<float>(buffer);
        auto context = juce::dsp::ProcessContextReplacing<float>(block);
        gain.process(context);
    }


//==============================================================================
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (One_MBCompAudioProcessor)
};
```

Figure C.1: PluginProcessor.h file.

Appendix D - pluginEditor.cpp

The provided code defines several classes and methods related to audio processing using the JUCE framework. The ControlBar class initialises various buttons and labels, and provides methods for resizing and painting the UI components. The LookAndFeel class customises the appearance of rotary sliders and toggle buttons. The RotarySliderWithLabels class extends the functionality of a rotary slider by adding labels around it. The Placeholder class is a simple class that generates a random colour. The CompressorBandControls class initialises and manages the layout of a set of rotary sliders, each representing a parameter of an audio compressor. The code also includes utility functions for converting frequency values to kilohertz and obtaining parameter values as strings. The code references and adapts from works by Schiermeyer (2021a; 2021b).

NOTE TO READER:

The segmented section of code was explicitly referenced from Schiermeyer (2021a; 2021b) to demonstrate the spectrum in the plugin. This code block has not been originally created by the author. The end of the block will be clearly stated for ease of reference to the reader.

```
/*
  ==============================================================================
    This file contains the basic framework code for a JUCE plugin editor.
    This code has been referenced and adapted from Hodge (2019), Tarr (2020a; 2020b) and Schiermeyer
(2021a; 2021b).
  ==============================================================================
*/
#include "PluginProcessor.h"
#include "PluginEditor.h"
#include <array>
/*
  ==============================================================================
    **************************************************************************
    NOTE TO READER: This following section of code was explicitly referenced from
    Schiermeyer (2021a; 2021b) to demonstrate the spectrum in the plugin. This
    code block has not be originally created by the author. The end of the block
    will be clearly stated for ease of reference to the reader.
    **************************************************************************
  ==============================================================================
*/
SpectrumAnalyser::SpectrumAnalyser(One_MBCompAudioProcessor& p) :
audioProcessor(p),
leftPathProducer(audioProcessor.leftChannelFifo),
rightPathProducer(audioProcessor.rightChannelFifo)
{
    const auto& params = audioProcessor.getParameters();
    for( auto param : params )
    {
        param->addListener(this);
    }
```

```
    startTimerHz(60);
}
SpectrumAnalyser::~SpectrumAnalyser()
{
    const auto& params = audioProcessor.getParameters();
    for( auto param : params )
    {
        param->removeListener(this);
    }
}
void SpectrumAnalyser::paint (juce::Graphics& g)
{
    using namespace juce;
    // (Our component is opaque, so we must completely fill the background with a solid colour)
    g.fillAll (Colours::black);
    drawBackgroundGrid(g);

    auto responseArea = getAnalysisArea();

    if( shouldShowFFTAnalysis )
    {
        auto leftChannelFFTPath = leftPathProducer.getPath();
        leftChannelFFTPath.applyTransform(AffineTransform().translation(responseArea.getX(),
responseArea.getY()));

        g.setColour(Colour(97u, 18u, 167u)); //purple-
        g.strokePath(leftChannelFFTPath, PathStrokeType(1.f));

        auto rightChannelFFTPath = rightPathProducer.getPath();
        rightChannelFFTPath.applyTransform(AffineTransform().translation(responseArea.getX(),
responseArea.getY()));

        g.setColour(Colour(215u, 201u, 134u));
        g.strokePath(rightChannelFFTPath, PathStrokeType(1.f));
    }

    Path border;

    border.setUsingNonZeroWinding(false);

    border.addRoundedRectangle(getRenderArea(), 4);
    border.addRectangle(getLocalBounds());

    g.setColour(Colours::black);

    g.fillPath(border);

    drawTextLabels(g);

    g.setColour(Colours::orange);
    g.drawRoundedRectangle(getRenderArea().toFloat(), 4.f, 1.f);
```

```
}
std::vector<float> SpectrumAnalyser::getFrequencies()
{
    return std::vector<float>
    {
        20, /*30, 40,*/ 50, 100,
        200, /*300, 400,*/ 500, 1000,
        2000, /*3000, 4000,*/ 5000, 10000,
        20000
    };
}
std::vector<float> SpectrumAnalyser::getGains()
{
    return std::vector<float>
    {
        -24, -12, 0, 12, 24
    };
}
std::vector<float> SpectrumAnalyser::getXs(const std::vector<float> &freqs, float left, float width)
{
    std::vector<float> xs;
    for( auto f : freqs )
    {
        auto normX = juce::mapFromLog10(f, 20.f, 20000.f);
        xs.push_back( left + width * normX );
    }

    return xs;
}
void SpectrumAnalyser::drawBackgroundGrid(juce::Graphics &g)
{
    using namespace juce;
    auto freqs = getFrequencies();

    auto renderArea = getAnalysisArea();
    auto left = renderArea.getX();
    auto right = renderArea.getRight();
    auto top = renderArea.getY();
    auto bottom = renderArea.getBottom();
    auto width = renderArea.getWidth();

    auto xs = getXs(freqs, left, width);

    g.setColour(Colours::dimgrey);
    for( auto x : xs )
    {
        g.drawVerticalLine(x, top, bottom);
    }

    auto gain = getGains();
```

```
    for( auto gDb : gain )
    {
        auto y = jmap(gDb, -24.f, 24.f, float(bottom), float(top));

        g.setColour(gDb == 0.f ? Colour(0u, 172u, 1u) : Colours::darkgrey );
        g.drawHorizontalLine(y, left, right);
    }
}
void SpectrumAnalyser::drawTextLabels(juce::Graphics &g)
{
    using namespace juce;
    g.setColour(Colours::lightgrey);
    const int fontHeight = 10;
    g.setFont(fontHeight);

    auto renderArea = getAnalysisArea();
    auto left = renderArea.getX();

    auto top = renderArea.getY();
    auto bottom = renderArea.getBottom();
    auto width = renderArea.getWidth();

    auto freqs = getFrequencies();
    auto xs = getXs(freqs, left, width);

    for( int i = 0; i < freqs.size(); ++i )
    {
        auto f = freqs[i];
        auto x = xs[i];
        bool addK = false;
        String str;
        if( f > 999.f )
        {
            addK = true;
            f /= 1000.f;
        }
        str << f;
        if( addK )
            str << "k";
        str << "Hz";

        auto textWidth = g.getCurrentFont().getStringWidth(str);
        Rectangle<int> r;
        r.setSize(textWidth, fontHeight);
        r.setCentre(x, 0);
        r.setY(1);

        g.drawFittedText(str, r, juce::Justification::centred, 1);
    }

    auto gain = getGains();
```

```
    for( auto gDb : gain )
    {
        auto y = jmap(gDb, -24.f, 24.f, float(bottom), float(top));

        String str;
        if( gDb > 0 )
            str << "+";
        str << gDb;

        auto textWidth = g.getCurrentFont().getStringWidth(str);

        Rectangle<int> r;
        r.setSize(textWidth, fontHeight);
        r.setX(getWidth() - textWidth);
        r.setCentre(r.getCentreX(), y);

        g.setColour(gDb == 0.f ? Colour(0u, 172u, 1u) : Colours::lightgrey );

        g.drawFittedText(str, r, juce::Justification::centredLeft, 1);

        str.clear();
        str << (gDb - 24.f);
        r.setX(1);
        textWidth = g.getCurrentFont().getStringWidth(str);
        r.setSize(textWidth, fontHeight);
        g.setColour(Colours::lightgrey);
        g.drawFittedText(str, r, juce::Justification::centredLeft, 1);
    }
}
void SpectrumAnalyser::resized()
{
    using namespace juce;
}
void SpectrumAnalyser::parameterValueChanged(int parameterIndex, float newValue)
{
    parametersChanged.set(true);
}
void PathProducer::process(juce::Rectangle<float> fftBounds, double sampleRate)
{
    juce::AudioBuffer<float> tempIncomingBuffer;
    while( leftChannelFifo->getNumCompleteBuffersAvailable() > 0 )
    {
        if( leftChannelFifo->getAudioBuffer(tempIncomingBuffer) )
        {
            auto size = tempIncomingBuffer.getNumSamples();
            juce::FloatVectorOperations::copy(monoBuffer.getWritePointer(0, 0),
                                              monoBuffer.getReadPointer(0, size),
                                              monoBuffer.getNumSamples() - size);
            juce::FloatVectorOperations::copy(monoBuffer.getWritePointer(0, monoBuffer.getNumSamples() -
size),
                                              tempIncomingBuffer.getReadPointer(0, 0),
```

```
                                size);

            leftChannelFFTDataGenerator.produceFFTDataForRendering(monoBuffer, -48.f);
        }
    }

    const auto fftSize = leftChannelFFTDataGenerator.getFFTSize();
    const auto binWidth = sampleRate / double(fftSize);
    while( leftChannelFFTDataGenerator.getNumAvailableFFTDataBlocks() > 0 )
    {
        std::vector<float> fftData;
        if( leftChannelFFTDataGenerator.getFFTData( fftData) )
        {
            pathProducer.generatePath(fftData, fftBounds, fftSize, binWidth, -48.f);
        }
    }

    while( pathProducer.getNumPathsAvailable() > 0 )
    {
        pathProducer.getPath( leftChannelFFTPath );
    }
}
void SpectrumAnalyser::timerCallback()
{
    if( shouldShowFFTAnalysis )
    {
        auto fftBounds = getAnalysisArea().toFloat();
        auto sampleRate = audioProcessor.getSampleRate();

        leftPathProducer.process(fftBounds, sampleRate);
        rightPathProducer.process(fftBounds, sampleRate);
    }
    if( parametersChanged.compareAndSetBool(false, true) )
    {

    }

    repaint();
}
juce::Rectangle<int> SpectrumAnalyser::getRenderArea()
{
    auto bounds = getLocalBounds();

    bounds.removeFromTop(12);
    bounds.removeFromBottom(2);
    bounds.removeFromLeft(20);
    bounds.removeFromRight(20);

    return bounds;
}
juce::Rectangle<int> SpectrumAnalyser::getAnalysisArea()
```

```
{
    auto bounds = getRenderArea();
    bounds.removeFromTop(4);
    bounds.removeFromBottom(4);
    return bounds;
}
/*


==========================================================================================
    ************************************************************************
                        END OF SPECTRUM CODE BLOCK
                        (Schiermeyer, 2021a; 2021b)
    ************************************************************************


==========================================================================================
*/
ControlBar::ControlBar(juce::AudioProcessorValueTreeState& apvts)
{
    using namespace PluginParameters;
    const auto& parameters = GetParameters();

    // Use only the button label to initialize the ToggleButton.
    bypassButton1 = std::make_unique<Buttons>("X");
    soloButton1 = std::make_unique<Buttons>("S");
    muteButton1 = std::make_unique<Buttons>("M");
    titleLabel1.setText("LB", juce::NotificationType::dontSendNotification);

    bypassButton2 = std::make_unique<Buttons>("X");
    soloButton2 = std::make_unique<Buttons>("S");
    muteButton2 = std::make_unique<Buttons>("M");
    titleLabel2.setText("MB", juce::NotificationType::dontSendNotification);
    bypassButton3 = std::make_unique<Buttons>("X");
    soloButton3 = std::make_unique<Buttons>("S");
    muteButton3 = std::make_unique<Buttons>("M");
    titleLabel3.setText("HB", juce::NotificationType::dontSendNotification);
    auto makeAttachmentHelper = [&parameters, &apvts](auto& attachment, const auto& name, auto&
button)
    {
        makeBtnAttachment(attachment, apvts, parameters, name, button);
    };

    makeAttachmentHelper(bypassButtonAttachment1, ParamNames::Bypass_LB, *bypassButton1);
    makeAttachmentHelper(soloButtonAttachment1, ParamNames::Solo_LB, *soloButton1);
    makeAttachmentHelper(muteButtonAttachment1, ParamNames::Mute_LB, *muteButton1);

    makeAttachmentHelper(bypassButtonAttachment2, ParamNames::Bypass_MB, *bypassButton2);
    makeAttachmentHelper(soloButtonAttachment2, ParamNames::Solo_MB, *soloButton2);
    makeAttachmentHelper(muteButtonAttachment2, ParamNames::Mute_MB, *muteButton2);

    makeAttachmentHelper(bypassButtonAttachment3, ParamNames::Bypass_HB, *bypassButton3);
    makeAttachmentHelper(soloButtonAttachment3, ParamNames::Solo_HB, *soloButton3);
```

```
    makeAttachmentHelper(muteButtonAttachment3, ParamNames::Mute_HB, *muteButton3);


    // Add buttons and title labels to the component
    addAndMakeVisible(*bypassButton1);
    addAndMakeVisible(*soloButton1);
    addAndMakeVisible(*muteButton1);
    addAndMakeVisible(titleLabel1);

    addAndMakeVisible(*bypassButton2);
    addAndMakeVisible(*soloButton2);
    addAndMakeVisible(*muteButton2);
    addAndMakeVisible(titleLabel2);

    addAndMakeVisible(*bypassButton3);
    addAndMakeVisible(*soloButton3);
    addAndMakeVisible(*muteButton3);
    addAndMakeVisible(titleLabel3);
}
void ControlBar::resized()
{
    auto bounds = getLocalBounds();
    auto buttonWidth = bounds.getWidth() / 9;
    auto buttonHeight = 20;  // Adjust this as needed
    auto titleHeight = 18;   // Adjust this as needed
    // Position the low group
    titleLabel1.setBounds(bounds.removeFromTop(titleHeight));
    bypassButton1->setBounds(bounds.removeFromLeft(buttonWidth).withHeight(buttonHeight));
    soloButton1->setBounds(bounds.removeFromLeft(buttonWidth).withHeight(buttonHeight));
    muteButton1->setBounds(bounds.removeFromLeft(buttonWidth).withHeight(buttonHeight));

    // Reset bounds for mid group
    bounds = getLocalBounds();
    auto midColumn = bounds.removeFromRight(335);
    // Position the mid group
    titleLabel2.setBounds(midColumn.removeFromTop(titleHeight));
    bypassButton2->setBounds(midColumn.removeFromLeft(buttonWidth).withHeight(buttonHeight));
    soloButton2->setBounds(midColumn.removeFromLeft(buttonWidth).withHeight(buttonHeight));
    muteButton2->setBounds(midColumn.removeFromLeft(buttonWidth).withHeight(buttonHeight));
    // Reset bounds for high group
    bounds = getLocalBounds();
    auto rightColumn = bounds.removeFromRight(175);  // Adjust this as needed
    // Position the high group
    titleLabel3.setBounds(rightColumn.removeFromTop(titleHeight));
    bypassButton3->setBounds(rightColumn.removeFromLeft(buttonWidth).withHeight(buttonHeight));
    soloButton3->setBounds(rightColumn.removeFromLeft(buttonWidth).withHeight(buttonHeight));
    muteButton3->setBounds(rightColumn.removeFromLeft(buttonWidth).withHeight(buttonHeight));
}
void ControlBar::paint(juce::Graphics& g)
{
    using namespace juce;
```

```
    auto bounds = getLocalBounds();
    g.setColour(Colours::grey);
    g.fillAll();

    auto localBounds = bounds;

    bounds.reduce(3,3);
    g.setColour(Colours::black);
    g.fillRoundedRectangle(bounds.toFloat(), 3);

    g.drawRect(localBounds);
}
//======================referenced and adapted from Schiermeyer (2021a;
2021b)===========================================================
template<typename T>
bool addKilohertz(T& value)
{
    if( value > static_cast<T>(999) )
    {
        value /= static_cast<T>(1000);
        return true;
    }
    return false;
}
juce::String getValString(const juce::RangedAudioParameter& param,
                          bool getLow,
                          juce::String suffix)
{
    juce::String string;

    auto value = getLow ? param.getNormalisableRange().start : param.getNormalisableRange().end;

    bool useK = addKilohertz(value);
    string << value;

    if( useK )
    {
        string << "k";
    }

    string << suffix;

    return string;
}
//================================================================================
=
void LookAndFeel::drawRotarySlider(juce::Graphics &g,
                                   int x,
                                   int y,
                                   int width,
                                   int height,
```

```
                        float sliderPosProportional,
                        float rotaryStartAngle,
                        float rotaryEndAngle,
                        juce::Slider &slider)
{
  using namespace juce;
  auto bounds = Rectangle<float>(x, y, width, height);

  auto enabled = slider.isEnabled();

  g.setColour(enabled ? Colour(236u, 114u, 41u) : Colours::darkgrey);
  g.fillEllipse(bounds);

  g.setColour(enabled ? Colour(46u, 48u, 45u) : Colours::grey);
  g.drawEllipse(bounds, 1.f);

  if(auto* rswl = dynamic_cast<RotarySliderWithLabels*>(&slider))
  {
    auto center = bounds.getCentre();

    Path p;

    Rectangle<float> r;
    r.setLeft(center.getX() - 2);
    r.setRight(center.getX() + 2);
    r.setTop(bounds.getY());
    r.setBottom(center.getY() - rswl->getTextHeight() * 1.5);

    p.addRoundedRectangle(r, 2.f);

    jassert(rotaryStartAngle < rotaryEndAngle);

    auto sliderAngRad = jmap(sliderPosProportional, 0.f, 1.f, rotaryStartAngle, rotaryEndAngle);

    p.applyTransform(AffineTransform().rotated(sliderAngRad, center.getX(), center.getY()));

    g.fillPath(p);

    g.setFont(rswl->getTextHeight());
    auto text = rswl->getDisplayString();
    auto strWidth = g.getCurrentFont().getStringWidth(text);

    r.setSize(strWidth + 4, rswl->getTextHeight() + 2);
    r.setCentre(bounds.getCentre());

    g.setColour(enabled ? Colours::black : Colours::darkgrey);
    g.fillRect(r);

    g.setColour(enabled ? Colours::white : Colours::lightgrey);
    g.drawFittedText(text, r.toNearestInt(), juce::Justification::centred, 1);
  }
```

```
}
void LookAndFeel::drawToggleButton(juce::Graphics &g,
                                   juce::ToggleButton &toggleButton,
                                   bool shouldDrawButtonAsHighlighted,
                                   bool shouldDrawButtonAsDown)
{
    using namespace juce;

    if(auto* pb = dynamic_cast<PowerButton*>(&toggleButton))
    {
        Path powerButton;

        auto bounds = toggleButton.getLocalBounds();
        auto size = jmin(bounds.getWidth(), bounds.getHeight() - 6);
        auto r = bounds.withSizeKeepingCentre(size, size).toFloat();

        float ang = 30.f;
        size -= 6;

        powerButton.addCentredArc(r.getCentreX(),
                                  r.getCentreY(),
                                  size * 0.5,
                                  size * 0.5,
                                  0.f,
                                  degreesToRadians(ang),
                                  degreesToRadians(360.f - ang),
                                  true);

        powerButton.startNewSubPath(r.getCentreX(), r.getY());
        powerButton.lineTo(r.getCentre());

        PathStrokeType pst(2.f, PathStrokeType::JointStyle::curved);

        auto colour = toggleButton.getToggleState() ? Colours::dimgrey : Colours::red;

        g.setColour(colour);
        g.strokePath(powerButton, pst);
        g.drawEllipse(r, 2);
    }
}
//==============================================================================
void RotarySliderWithLabels::paint(juce::Graphics &g)
{
    using namespace juce;
    auto startAng = degreesToRadians(180.f + 45.f);
    auto endAng = degreesToRadians(180.f - 45.f) + MathConstants<float>::twoPi;

    auto range = getRange();

    auto sliderBounds = getSliderBounds();
```

```
    auto bounds = getLocalBounds();

    g.setColour(Colours::darkkhaki);
    g.drawFittedText(getName(), bounds.removeFromTop(getTextHeight() + 2), Justification::centredTop, 1);

    getLookAndFeel().drawRotarySlider(g,
                                      sliderBounds.getX(),
                                      sliderBounds.getY(),
                                      sliderBounds.getWidth(),
                                      sliderBounds.getHeight(),
                                      jmap(getValue(), range.getStart(), range.getEnd(), 0.0, 1.0),
                                      startAng,
                                      endAng,
                                      *this);

    auto center = sliderBounds.toFloat().getCentre();
    auto radius = sliderBounds.getWidth() * 0.5f;

    g.setColour(Colour(0u, 172u, 1u));
    g.setFont(getTextHeight());

    auto numChoices = labels.size();
    for(int i = 0; i < numChoices; ++i)
    {
        auto pos = labels[i].pos;
        jassert(0.f <= pos);
        jassert(pos <= 1.f);

        auto ang = jmap(pos, 0.f, 1.f, startAng, endAng);

        auto c = center.getPointOnCircumference(radius + getTextHeight() * 0.5f + 1, ang);

        Rectangle<float> r;
        auto str = labels[i].label;
        r.setSize(g.getCurrentFont().getStringWidth(str), getTextHeight());
        r.setCentre(c);
        r.setY(r.getY() + getTextHeight());

        g.drawFittedText(str, r.toNearestInt(), juce::Justification::centred, 1);

    }

}
juce::Rectangle<int> RotarySliderWithLabels::getSliderBounds() const
{
    auto bounds = getLocalBounds();

    bounds.removeFromTop(getTextHeight());

    auto size = juce::jmin(bounds.getWidth(), bounds.getHeight());
```

```
    size -= getTextHeight();

    juce::Rectangle<int> r;
    r.setSize(size, size);
    r.setCentre(bounds.getCentreX(), bounds.getCentreY());
//    r.setY(2);
    r.setY(bounds.getY());

    return r;
}
juce::String RotarySliderWithLabels::getDisplayString() const
{
    if(auto* choiceParam = dynamic_cast<juce::AudioParameterChoice*>(param))
        return choiceParam->getCurrentChoiceName();

    juce::String str;
    bool addK = false;

    if(auto* floatParam = dynamic_cast<juce::AudioParameterFloat*>(param))
    {
        float val = getValue();

//        if(val > 999.f)
//        {
//            val /= 1000.f;
//            addK = true;
//        }

        addK = addKilohertz(val);
        str = juce::String(val, (addK ? 2 : 0));
    }
    else
    {
        jassertfalse;
    }

    if(suffix.isNotEmpty())
    {
        str << " ";
        if(addK)
        {
            str << "k";
        }
        str << suffix;
    }

    return str;
}
//==============================end of
ref================================================
```

```
Placeholder::Placeholder()
{
    juce::Random r;
    customColour = juce::Colour(r.nextInt(255), r.nextInt(255), r.nextInt(255));
}
//==============================================================================
CompressorBandControls::CompressorBandControls(juce::AudioProcessorValueTreeState& apvts)
{
    using namespace PluginParameters;
    const auto& parameters = GetParameters();

    auto getParamHelper = [&parameters, &apvts](const auto& name) -> auto&
    {
        return getParameter(apvts, parameters, name);
    };

    // Create and assign parameters
    atkSlider1 = std::make_unique<RotarySliderWL2>(getParamHelper(ParamNames::Attack_LB),"ms",
"Att-LB");
    relSlider1 = std::make_unique<RotarySliderWL2>(getParamHelper(ParamNames::Release_LB),"ms",
"Rel-LB");
    thresSlider1 = std::make_unique<RotarySliderWL2>(getParamHelper(ParamNames::Threshold_LB),"dB",
"Thr-LB");
    ratiSlider1 = std::make_unique<RotarySliderWL2>(getParamHelper(ParamNames::Ratio_LB),"Amt",
"Rat-LB");

    atkSlider2 = std::make_unique<RotarySliderWL2>(getParamHelper(ParamNames::Attack_MB),"ms",
"Att-MB");
    relSlider2 = std::make_unique<RotarySliderWL2>(getParamHelper(ParamNames::Release_MB),"ms",
"Rel-MB");
    thresSlider2 =
std::make_unique<RotarySliderWL2>(getParamHelper(ParamNames::Threshold_MB),"dB", "Thr-MB");
    ratiSlider2 = std::make_unique<RotarySliderWL2>(getParamHelper(ParamNames::Ratio_MB),"Amt",
"Rat-MB");

    atkSlider3 = std::make_unique<RotarySliderWL2>(getParamHelper(ParamNames::Attack_HB),"ms",
"Att-HB");
    relSlider3 = std::make_unique<RotarySliderWL2>(getParamHelper(ParamNames::Release_HB),"ms",
"Rel-HB");
    thresSlider3 =
std::make_unique<RotarySliderWL2>(getParamHelper(ParamNames::Threshold_HB),"dB", "Thr-HB");
    ratiSlider3 = std::make_unique<RotarySliderWL2>(getParamHelper(ParamNames::Ratio_HB),"Amt",
"Rat-HB");

    auto makeAttachmentHelper = [&parameters, &apvts](auto& attachment, const auto& name, auto&
slider)
    {
        makeAttachment(attachment, apvts, parameters, name, slider);
    };
```

```
    makeAttachmentHelper(atkSlider1_Attachment, ParamNames::Attack_LB, *atkSlider1);
    makeAttachmentHelper(relSlider1_Attachment, ParamNames::Release_LB, *relSlider1);
    makeAttachmentHelper(thresSlider1_Attachment, ParamNames::Threshold_LB, *thresSlider1);
    makeAttachmentHelper(ratiSlider1_Attachment, ParamNames::Ratio_LB, *ratiSlider1);

    makeAttachmentHelper(atkSlider2_Attachment, ParamNames::Attack_MB, *atkSlider2);
    makeAttachmentHelper(relSlider2_Attachment, ParamNames::Release_MB, *relSlider2);
    makeAttachmentHelper(thresSlider2_Attachment, ParamNames::Threshold_MB, *thresSlider2);
    makeAttachmentHelper(ratiSlider2_Attachment, ParamNames::Ratio_MB, *ratiSlider2);

    makeAttachmentHelper(atkSlider3_Attachment, ParamNames::Attack_HB, *atkSlider3);
    makeAttachmentHelper(relSlider3_Attachment, ParamNames::Release_HB, *relSlider3);
    makeAttachmentHelper(thresSlider3_Attachment, ParamNames::Threshold_HB, *thresSlider3);
    makeAttachmentHelper(ratiSlider3_Attachment, ParamNames::Ratio_HB, *ratiSlider3);

    addLabelPairs(atkSlider1->labels, getParamHelper(ParamNames::Attack_LB), "ms");
    addLabelPairs(relSlider1->labels, getParamHelper(ParamNames::Release_LB), "ms");
    addLabelPairs(thresSlider1->labels, getParamHelper(ParamNames::Threshold_LB), "dB");
    addLabelPairs(ratiSlider1->labels, getParamHelper(ParamNames::Ratio_LB), "Amt");

    addLabelPairs(atkSlider2->labels, getParamHelper(ParamNames::Attack_MB), "ms");
    addLabelPairs(relSlider2->labels, getParamHelper(ParamNames::Release_MB), "ms");
    addLabelPairs(thresSlider2->labels, getParamHelper(ParamNames::Threshold_MB), "dB");
    addLabelPairs(ratiSlider2->labels, getParamHelper(ParamNames::Ratio_MB), "Amt");

    addLabelPairs(atkSlider3->labels, getParamHelper(ParamNames::Attack_HB), "ms");
    addLabelPairs(relSlider3->labels, getParamHelper(ParamNames::Release_HB), "ms");
    addLabelPairs(thresSlider3->labels, getParamHelper(ParamNames::Threshold_HB), "dB");
    addLabelPairs(ratiSlider3->labels, getParamHelper(ParamNames::Ratio_HB), "Amt");

    addAndMakeVisible(*atkSlider1);
    addAndMakeVisible(*relSlider1);
    addAndMakeVisible(*thresSlider1);
    addAndMakeVisible(*ratiSlider1);

    addAndMakeVisible(*atkSlider2);
    addAndMakeVisible(*relSlider2);
    addAndMakeVisible(*thresSlider2);
    addAndMakeVisible(*ratiSlider2);

    addAndMakeVisible(*atkSlider3);
    addAndMakeVisible(*relSlider3);
    addAndMakeVisible(*thresSlider3);
    addAndMakeVisible(*ratiSlider3);

}
void CompressorBandControls::resized()
{
    auto bounds = getLocalBounds().reduced(5);
    using namespace juce;
    // Main FlexBox to contain the Control FlexBoxes
```

```
    FlexBox mainFlex;
    mainFlex.flexDirection = FlexBox::Direction::column;  // Each group in a new line
    mainFlex.justifyContent = FlexBox::JustifyContent::center;
    // Individual Control FlexBoxes (for better structuring)
    FlexBox controlFlex1, controlFlex2, controlFlex3;
    // Configure as rows
    controlFlex1.flexDirection = FlexBox::Direction::row;
    controlFlex1.items.addArray({
        FlexItem(*atkSlider1).withFlex(1),
        FlexItem(*relSlider1).withFlex(1),
        FlexItem(*thresSlider1).withFlex(1),
        FlexItem(*ratiSlider1).withFlex(1)
        });

    controlFlex2.flexDirection = FlexBox::Direction::row;
    controlFlex2.items.addArray({
        FlexItem(*atkSlider2).withFlex(1),
        FlexItem(*relSlider2).withFlex(1),
        FlexItem(*thresSlider2).withFlex(1),
        FlexItem(*ratiSlider2).withFlex(1)
        });

    controlFlex3.flexDirection = FlexBox::Direction::row;
    controlFlex3.items.addArray({
        FlexItem(*atkSlider3).withFlex(1),
        FlexItem(*relSlider3).withFlex(1),
        FlexItem(*thresSlider3).withFlex(1),
        FlexItem(*ratiSlider3).withFlex(1)
        });

    mainFlex.items.addArray({
        FlexItem(controlFlex1).withFlex(1),
        FlexItem(controlFlex2).withFlex(1),
        FlexItem(controlFlex3).withFlex(1)
    });
    // Perform layout within the component's bounds
    mainFlex.performLayout(bounds);
}
void CompressorBandControls::paint(juce::Graphics& g)
{
    using namespace juce;
    auto bounds = getLocalBounds();
    g.setColour(Colours::grey);
    g.fillAll();

    auto localBounds = bounds;

    bounds.reduce(3,3);
    g.setColour(Colours::black);
    g.fillRoundedRectangle(bounds.toFloat(), 3);
```

```
    g.drawRect(localBounds);
}
//==============================================================================
GlobalControls::GlobalControls(juce::AudioProcessorValueTreeState& apvts)
{
    using namespace PluginParameters;
    const auto& parameters = GetParameters();

    auto getParamHelper = [&parameters, &apvts](const auto& name) -> auto&
    {
        return getParameter(apvts, parameters, name);
    };

    auto& gainInputParameter = getParamHelper(ParamNames::Gain_Input);
    auto& lowMidParameter = getParamHelper(ParamNames::Low_Mid_XO_Frequency);
    auto& midHighParameter = getParamHelper(ParamNames::Mid_High_XO_Frequency);
    auto& gainOutputParameter = getParamHelper(ParamNames::Gain_Output);

    inputGainSlider = std::make_unique<RotarySliderWL>(gainInputParameter,"dB", "Input Gain");
    lowMidCrossoverSlider = std::make_unique<RotarySliderWL>(lowMidParameter,"Hz", "Low-Mid
Range");
    midHighCrossoverSlider = std::make_unique<RotarySliderWL>(midHighParameter,"Hz", "Mid-Hi
Range");
    outputGainSlider = std::make_unique<RotarySliderWL>(gainOutputParameter,"dB", "Output Gain");

    auto makeAttachmentHelper = [&parameters, &apvts](auto& attachment,
                                        const auto& name,
                                        auto& slider)
    {
        makeAttachment(attachment, apvts, parameters, name, slider);
    };

    makeAttachmentHelper(inputGainSliderAttachment,
                         ParamNames::Gain_Input,
                         *inputGainSlider);
    makeAttachmentHelper(lowMidCrossoverSliderAttachment,
                         ParamNames::Low_Mid_XO_Frequency,
                         *lowMidCrossoverSlider);
    makeAttachmentHelper(midHighCrossoverSliderAttachment,
                         ParamNames::Mid_High_XO_Frequency,
                         *midHighCrossoverSlider);
    makeAttachmentHelper(outputGainSliderAttachment,
                         ParamNames::Gain_Output,
                         *outputGainSlider);

    addLabelPairs(inputGainSlider->labels,
                  gainInputParameter,
                  "dB");
    addLabelPairs(lowMidCrossoverSlider->labels,
                  lowMidParameter,
```

```
                "Hz");
    addLabelPairs(midHighCrossoverSlider->labels,
                midHighParameter,
                "Hz");
    addLabelPairs(outputGainSlider->labels,
                gainOutputParameter,
                "dB");

    addAndMakeVisible(*inputGainSlider);
    addAndMakeVisible(*lowMidCrossoverSlider);
    addAndMakeVisible(*midHighCrossoverSlider);
    addAndMakeVisible(*outputGainSlider);
}
void GlobalControls::paint(juce::Graphics& g)
{
    using namespace juce;
    auto bounds = getLocalBounds();
    g.setColour(Colours::grey);
    g.fillAll();

    auto localBounds = bounds;

    bounds.reduce(3,3);
    g.setColour(Colours::black);
    g.fillRoundedRectangle(bounds.toFloat(), 3);

    g.drawRect(localBounds);

}
void GlobalControls::resized()
{
    auto bounds = getLocalBounds().reduced(5);
    using namespace juce;

    FlexBox flexBox;
    flexBox.flexDirection = FlexBox::Direction::row;
    flexBox.flexWrap = FlexBox::Wrap::noWrap;

    auto spacer = FlexItem().withWidth(4);
    auto endCap = FlexItem().withWidth(6);

    flexBox.items.add(endCap);
    flexBox.items.add(FlexItem(*inputGainSlider).withFlex(1.f));
    flexBox.items.add(spacer);
    flexBox.items.add(FlexItem(*lowMidCrossoverSlider).withFlex(1.f));
    flexBox.items.add(spacer);
    flexBox.items.add(FlexItem(*midHighCrossoverSlider).withFlex(1.f));
    flexBox.items.add(spacer);
    flexBox.items.add(FlexItem(*outputGainSlider).withFlex(1.f));
    flexBox.items.add(endCap);
```

```
    flexBox.performLayout(bounds);
}
One_MBCompAudioProcessorEditor::One_MBCompAudioProcessorEditor
(One_MBCompAudioProcessor& processor)
    : AudioProcessorEditor (&processor), audioProcessor (processor)
{
    // Make sure that before the constructor has finished, you've set the
    // editor's size to whatever you need it to be.
    addAndMakeVisible(controlBar);
    addAndMakeVisible(specAnalyser);
    addAndMakeVisible(globalControls);
    addAndMakeVisible(bandControls);

    setSize (500, 600);
}
One_MBCompAudioProcessorEditor::~One_MBCompAudioProcessorEditor()
{
}
//==============================================================================
void One_MBCompAudioProcessorEditor::paint (juce::Graphics& g)
{
    // (Our component is opaque, so we must completely fill the background with a solid colour)
//    g.fillAll (getLookAndFeel().findColour (juce::ResizableWindow::backgroundColourId));
//
//    g.setColour (juce::Colours::white);
//    g.setFont (15.0f);
//    g.drawFittedText ("Hello World!", getLocalBounds(), juce::Justification::centred, 1);
//
    g.fillAll(juce::Colours::black);
}
void One_MBCompAudioProcessorEditor::resized()
{
    // This is generally where you'll want to lay out the positions of any
    // subcomponents in your editor..
    auto bounds = getLocalBounds();

    controlBar.setBounds(bounds.removeFromTop(40));

    bandControls.setBounds(bounds.removeFromBottom(225));

    specAnalyser.setBounds(bounds.removeFromTop(225));

    globalControls.setBounds(bounds);

}
```

Figure D.1: PluginEditor.cpp file.

Appendix E - pluginEditor.h

The code outlines the user interface components for the audio plugin and can be broken down into the following:

ControlBar: Represents a control section with toggle buttons (bypass, solo, mute) for multiple channels and associated labels.

LookAndFeel: Customises the appearance of rotary sliders and toggle buttons.

RotarySliderWithLabels: A rotary slider enhanced with labels.

PowerButton & Placeholder: Simple structures for a toggle button and a placeholder component.

RotarySlider: A basic rotary slider.

makeAttachment & related Functions: Template functions to link UI components to audio processor parameters and manage parameter retrieval.

CompressorBandControls: Controls for different bands in a multiband compressor, including attack, release, threshold, and ratio.

GlobalControls: Global settings for the plugin, including input gain and crossover frequencies.

One_MBCompAudioProcessorEditor: The main UI class for the audio plugin, integrating all the above components and handling visual rendering and resizing.

NOTE TO READER:

The segmented section of code was explicitly referenced from Schiermeyer (2021a; 2021b) to demonstrate the spectrum in the plugin. This code block has not been originally created by the author. The end of the block will be clearly stated for ease of reference to the reader.

```
/*
  ==============================================================================
    This file contains the basic framework code for a JUCE plugin editor.
    This code has been referenced and adapted from Hodge (2019), Tarr (2020a; 2020b) and Schiermeyer
(2021a; 2021b).
  ==============================================================================
*/
#pragma once
#include <JuceHeader.h>
#include "PluginProcessor.h"
struct ControlBar : juce::Component
{
    ControlBar(juce::AudioProcessorValueTreeState& apvts);
    void resized() override;
    void paint(juce::Graphics& g) override;
private:
```

```
    using Buttons = juce::ToggleButton;

    std::unique_ptr<Buttons>    bypassButton1, soloButton1, muteButton1,
                                bypassButton2, soloButton2, muteButton2,
                                bypassButton3, soloButton3, muteButton3;

    juce::Label titleLabel1, titleLabel2, titleLabel3;

    using Attachment = juce::AudioProcessorValueTreeState::ButtonAttachment;
    std::unique_ptr<Attachment>
        bypassButtonAttachment1, soloButtonAttachment1, muteButtonAttachment1,
        bypassButtonAttachment2, soloButtonAttachment2, muteButtonAttachment2,
        bypassButtonAttachment3, soloButtonAttachment3, muteButtonAttachment3;
};
/*

==============================================================================
    ******************************************************************************
    NOTE TO READER: This following section of code was explicitly referenced from
    Schiermeyer (2021a; 2021b) to demonstrate the spectrum in the plugin. This
    code block has not be originally created by the author. The end of the block
    will be clearly stated for ease of reference to the reader.
    ******************************************************************************

==============================================================================
*/
enum FFTOrder
{
    order2048 = 11,
    order4096 = 12,
    order8192 = 13
};
template<typename BlockType>
struct FFTDataGenerator
{
    /**
     produces the FFT data from an audio buffer.
     */
    void produceFFTDataForRendering(const juce::AudioBuffer<float>& audioData, const float
negativeInfinity)
    {
        const auto fftSize = getFFTSize();

        fftData.assign(fftData.size(), 0);
        auto* readIndex = audioData.getReadPointer(0);
        std::copy(readIndex, readIndex + fftSize, fftData.begin());

        // first apply a windowing function to our data
        window->multiplyWithWindowingTable (fftData.data(), fftSize);       // [1]

        // then render our FFT data..
```

```
        forwardFFT->performFrequencyOnlyForwardTransform (fftData.data());  // [2]

        int numBins = (int)fftSize / 2;

        //normalize the fft values.
        for( int i = 0; i < numBins; ++i )
        {
            auto v = fftData[i];
//            fftData[i] /= (float) numBins;
            if( !std::isinf(v) && !std::isnan(v) )
            {
                v /= float(numBins);
            }
            else
            {
                v = 0.f;
            }
            fftData[i] = v;
        }

        //convert them to decibels
        for( int i = 0; i < numBins; ++i )
        {
            fftData[i] = juce::Decibels::gainToDecibels(fftData[i], negativeInfinity);
        }

        fftDataFifo.push(fftData);
    }

    void changeOrder(FFTOrder newOrder)
    {
        //when you change order, recreate the window, forwardFFT, fifo, fftData
        //also reset the fifoIndex
        //things that need recreating should be created on the heap via std::make_unique<>

        order = newOrder;
        auto fftSize = getFFTSize();

        forwardFFT = std::make_unique<juce::dsp::FFT>(order);
        window = std::make_unique<juce::dsp::WindowingFunction<float>>(fftSize,
juce::dsp::WindowingFunction<float>::blackmanHarris);

        fftData.clear();
        fftData.resize(fftSize * 2, 0);
        fftDataFifo.prepare(fftData.size());
    }

//==========================================================================
=
    int getFFTSize() const { return 1 << order; }
    int getNumAvailableFFTDataBlocks() const { return fftDataFifo.getNumAvailableForReading(); }
```

```
//==============================================================================
=
    bool getFFTData(BlockType& fftData) { return fftDataFifo.pull(fftData); }
private:
    FFTOrder order;
    BlockType fftData;
    std::unique_ptr<juce::dsp::FFT> forwardFFT;
    std::unique_ptr<juce::dsp::WindowingFunction<float>> window;

    Fifo<BlockType> fftDataFifo;
};
template<typename PathType>
struct AnalyzerPathGenerator
{
    /*
     converts 'renderData[]' into a juce::Path
     */
    void generatePath(const std::vector<float>& renderData,
                      juce::Rectangle<float> fftBounds,
                      int fftSize,
                      float binWidth,
                      float negativeInfinity)
    {
        auto top = fftBounds.getY();
        auto bottom = fftBounds.getHeight();
        auto width = fftBounds.getWidth();
        int numBins = (int)fftSize / 2;
        PathType p;
        p.preallocateSpace(3 * (int)fftBounds.getWidth());
        auto map = [bottom, top, negativeInfinity](float v)
        {
            return juce::jmap(v,
                              negativeInfinity, 0.f,
                              float(bottom+10),   top);
        };
        auto y = map(renderData[0]);
//        jassert( !std::isnan(y) && !std::isinf(y) );
        if( std::isnan(y) || std::isinf(y) )
            y = bottom;

        p.startNewSubPath(0, y);
        const int pathResolution = 2; //you can draw line-to's every 'pathResolution' pixels.
        for( int binNum = 1; binNum < numBins; binNum += pathResolution )
        {
            y = map(renderData[binNum]);
//            jassert( !std::isnan(y) && !std::isinf(y) );
            if( !std::isnan(y) && !std::isinf(y) )
            {
                auto binFreq = binNum * binWidth;
                auto normalizedBinX = juce::mapFromLog10(binFreq, 20.f, 20000.f);
```

```
            int binX = std::floor(normalizedBinX * width);
            p.lineTo(binX, y);
          }
        }
        pathFifo.push(p);
    }
    int getNumPathsAvailable() const
    {
        return pathFifo.getNumAvailableForReading();
    }
    bool getPath(PathType& path)
    {
        return pathFifo.pull(path);
    }
private:
    Fifo<PathType> pathFifo;
};
struct PathProducer
{
    PathProducer(SingleChannelSampleFifo<One_MBCompAudioProcessor::BlockType>& scsf) :
    leftChannelFifo(&scsf)
    {
        leftChannelFFTDataGenerator.changeOrder(FFTOrder::order2048);
        monoBuffer.setSize(1, leftChannelFFTDataGenerator.getFFTSize());
    }
    void process(juce::Rectangle<float> fftBounds, double sampleRate);
    juce::Path getPath() { return leftChannelFFTPath; }
private:
    SingleChannelSampleFifo<One_MBCompAudioProcessor::BlockType>* leftChannelFifo;

    juce::AudioBuffer<float> monoBuffer;

    FFTDataGenerator<std::vector<float>> leftChannelFFTDataGenerator;

    AnalyzerPathGenerator<juce::Path> pathProducer;

    juce::Path leftChannelFFTPath;
};
struct SpectrumAnalyser: juce::Component,
juce::AudioProcessorParameter::Listener,
juce::Timer
{
    SpectrumAnalyser(One_MBCompAudioProcessor&);
    ~SpectrumAnalyser();

    void parameterValueChanged (int parameterIndex, float newValue) override;
    void parameterGestureChanged (int parameterIndex, bool gestureIsStarting) override { }

    void timerCallback() override;

    void paint(juce::Graphics& g) override;
```

```
    void resized() override;

    void toggleAnalysisEnablement(bool enabled)
    {
        shouldShowFFTAnalysis = enabled;
    }
private:
    One_MBCompAudioProcessor& audioProcessor;
    bool shouldShowFFTAnalysis = true;
    juce::Atomic<bool> parametersChanged { false };

    void drawBackgroundGrid(juce::Graphics& g);
    void drawTextLabels(juce::Graphics& g);

    std::vector<float> getFrequencies();
    std::vector<float> getGains();
    std::vector<float> getXs(const std::vector<float>& freqs, float left, float width);
    juce::Rectangle<int> getRenderArea();

    juce::Rectangle<int> getAnalysisArea();

    PathProducer leftPathProducer, rightPathProducer;
};
/*


    ******************************************************************************
                            END OF CODE BLOCK
                        (Schiermeyer, 2021a; 2021b)
    ******************************************************************************


*/
struct LookAndFeel : juce::LookAndFeel_V4
{
    void drawRotarySlider (juce::Graphics&,
                           int x, int y, int width, int height,
                           float sliderPosProportional,
                           float rotaryStartAngle,
                           float rotaryEndAngle,
                           juce::Slider&) override;

    void drawToggleButton (juce::Graphics &g,
                           juce::ToggleButton & toggleButton,
                           bool shouldDrawButtonAsHighlighted,
                           bool shouldDrawButtonAsDown) override;
};
struct RotarySliderWithLabels : juce::Slider
{
    RotarySliderWithLabels(juce::RangedAudioParameter& rap, const juce::String& unitSuffix, const
juce::String& title /*= no title */ ) : juce::Slider(juce::Slider::SliderStyle::RotaryHorizontalVerticalDrag,
```

```
juce::Slider::TextEntryBoxPosition::NoTextBox),
    param(&rap),
    suffix(unitSuffix)
    {
        setName(title);
        setLookAndFeel(&lnf);
    }

    ~RotarySliderWithLabels()
    {
        setLookAndFeel(nullptr);
    }

    struct LabelPos
    {
        float pos;
        juce::String label;
    };

    juce::Array<LabelPos> labels;

    void paint(juce::Graphics& g) override;
    juce::Rectangle<int> getSliderBounds() const;
    static int getTextHeight() { return 14; }
    juce::String getDisplayString() const;

private:
    LookAndFeel lnf;
    juce::RangedAudioParameter* param;
    juce::String suffix;
};
struct PowerButton : juce::ToggleButton {};
//==============================================================================
=
struct Placeholder : juce::Component
{
    Placeholder();

    void paint(juce::Graphics& g) override
    {
        g.fillAll(customColour);
    }

    juce::Colour customColour;
};
struct RotarySlider : juce::Slider
{
    RotarySlider() :
    juce::Slider(juce::Slider::SliderStyle::RotaryHorizontalVerticalDrag,
                 juce::Slider::TextEntryBoxPosition::NoTextBox)
    {}
```

```
};
template<
    typename Attachment,
    typename APVTS,
    typename PluginParameters,
    typename ParamNames,
    typename SliderType>
void makeAttachment(std::unique_ptr<Attachment>& attachment,
                    APVTS& apvts,
                    const PluginParameters& parameters,
                    const ParamNames& name,
                    SliderType& slider)
{
    attachment = std::make_unique<Attachment>(apvts,
                                              parameters.at(name),
                                              slider);
}
template<
    typename Attachment,
    typename APVTS,
    typename PluginParameters,
    typename ParamNames,
    typename ButtonType>
void makeBtnAttachment(std::unique_ptr<Attachment>& attachment,
                    APVTS& apvts,
                    const PluginParameters& parameters,
                    const ParamNames& name,
                    ButtonType& toggle)
{
    attachment = std::make_unique<Attachment>(apvts,
                                              parameters.at(name),
                                              toggle);
}
template<
    typename APVTS,
    typename PluginParameters,
    typename ParamNames>
juce::RangedAudioParameter& getParameter(APVTS& apvts, const PluginParameters& params, const
ParamNames& name)
{
    auto param = apvts.getParameter(params.at(name));
    jassert( param != nullptr );

    return *param;
}
juce::String getValString(const juce::RangedAudioParameter& param,
                          bool getLow,
                          juce::String suffix);
template<
    typename Labels,
    typename ParamType,
```

```
    typename SuffixType>
void addLabelPairs(Labels& labels, const ParamType& param, const SuffixType& suffix)
{
    labels.clear();
    labels.add({0.f, getValString(param, true, suffix)});
    labels.add({1.f, getValString(param, false, suffix)});
}
struct CompressorBandControls : juce::Component
{
    CompressorBandControls(juce::AudioProcessorValueTreeState& apvts);
    void resized() override;
    void paint(juce::Graphics& g) override;
private:
    using RotarySliderWL2 = RotarySliderWithLabels;

    std::unique_ptr<RotarySliderWL2> atkSlider1, atkSlider2, atkSlider3, relSlider1, relSlider2, relSlider3,
thresSlider1, thresSlider2, thresSlider3, ratiSlider1, ratiSlider2, ratiSlider3;

    using Attachment = juce::AudioProcessorValueTreeState::SliderAttachment;

    std::unique_ptr<Attachment> atkSlider1_Attachment,
                                atkSlider2_Attachment,
                                atkSlider3_Attachment,
                                relSlider1_Attachment,
                                relSlider2_Attachment,
                                relSlider3_Attachment,
                                thresSlider1_Attachment,
                                thresSlider2_Attachment,
                                thresSlider3_Attachment,
                                ratiSlider1_Attachment,
                                ratiSlider2_Attachment,
                                ratiSlider3_Attachment;
};
struct GlobalControls : juce::Component
{
    GlobalControls(juce::AudioProcessorValueTreeState& apvts);
    void paint(juce::Graphics& g) override;
    void resized() override;
private:
    using RotarySliderWL = RotarySliderWithLabels;
    std::unique_ptr<RotarySliderWL> inputGainSlider, lowMidCrossoverSlider, midHighCrossoverSlider,
outputGainSlider;

    using Attachment = juce::AudioProcessorValueTreeState::SliderAttachment;
    std::unique_ptr<Attachment> lowMidCrossoverSliderAttachment,
                                midHighCrossoverSliderAttachment,
                                inputGainSliderAttachment,
                                outputGainSliderAttachment;

};
/**
```

```
*/
class One_MBCompAudioProcessorEditor  : public juce::AudioProcessorEditor
{
public:
    One_MBCompAudioProcessorEditor (One_MBCompAudioProcessor&);
    ~One_MBCompAudioProcessorEditor() override;

//==============================================================================
    void paint (juce::Graphics&) override;
    void resized() override;
private:
    // This reference is provided as a quick way for your editor to
    // access the processor object that created it.
    One_MBCompAudioProcessor& audioProcessor;

    ControlBar controlBar { audioProcessor.apvts };

    GlobalControls globalControls { audioProcessor.apvts };
    CompressorBandControls bandControls { audioProcessor.apvts };
    SpectrumAnalyser specAnalyser { audioProcessor };
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR
(One_MBCompAudioProcessorEditor)
};
```

Figure E.1: PluginEditor.h file.

Appendix F - basicCompressor.cpp

The code delineates the implementation of the class named `BasicCompressor`. This class is a representation of an audio compressor.

The `prepare` method initialises the compressor with the provided specifications, particularly computing the alpha values for both the attack and release phases based on the sample rate and respective time values.

Several setter methods are provided to adjust the parameters of the compressor, including threshold level, compression ratio, attack time, release time, and make-up gain. Notably, when the attack and release times are updated, the alpha values are recomputed.

The primary method, `process`, handles the core functionality of the compressor. It processes blocks of audio samples, applying compression based on the set parameters. If the context is bypassed, the input audio is directly copied to the output without any processing. Otherwise, the method iterates over each channel and sample, computes the input level in decibels, determines the output level, calculates the difference between input and output levels, and applies an envelope to this difference. The gain for each sample is then computed and applied, producing the compressed audio output.

```
//
//  basicCompressor.cpp
//  one_MBComp
//
//  UWL #21501990.
//  Copyright © 2023 Oberon Day-West. All rights reserved.
//  This code has been referenced and adapted from Reiss and McPherson (2015), Pirkle (2019) and Tarr
```

```
(2019).
//  Please refer to the accompanying report for full list of references.
#include <JuceHeader.h>
#include "BasicCompressor.h"
// This method prepares the compressor with provided specifications
void BasicCompressor::prepare( const juce::dsp::ProcessSpec& compressorSpec )
{
   // Copy the provided specification to the class's specification member
   m_compressorSpecifications = compressorSpec;
   // Compute the alpha value for attack phase using provided specifications
   m_alphaAttack = std::exp(-1.0f / (m_releaseTimeInMs * m_compressorSpecifications.sampleRate /
1000.0f));

   // Compute the alpha value for release phase using provided specifications
   m_alphaRelease = std::exp(-1.0f / (m_releaseTimeInMs * m_compressorSpecifications.sampleRate /
1000.0f));
}
// This method sets the threshold level of the compressor
void BasicCompressor::setThresholdLevel(float newThresholdLevel)
{
   m_thresholdLevelDb = newThresholdLevel;
}
// This method sets the compression ratio of the compressor
void BasicCompressor::setCompressionRatio(float newCompressionRatio)
{
   m_compressionRatio = newCompressionRatio;
}
// This method sets the attack time of the compressor
void BasicCompressor::setAttackTime(float newAttackTimeInMs)
{
   m_attackTimeInMs = newAttackTimeInMs;
   // Compute the new alpha value for attack phase after attack time is updated
   m_alphaAttack = std::exp(-1.0f / (m_attackTimeInMs * m_compressorSpecifications.sampleRate /
1000.0f));
}
// This method sets the release time of the compressor
void BasicCompressor::setReleaseTime(float newReleaseTimeInMs)
{
   m_releaseTimeInMs = newReleaseTimeInMs;
   // Compute the new alpha value for release phase after release time is updated
   m_alphaRelease = std::exp(-1.0f / (m_releaseTimeInMs * m_compressorSpecifications.sampleRate /
1000.0f));
}
// This method sets the make-up gain of the compressor
void BasicCompressor::setMakeUpGain(float newMakeUpGain)
{
   m_newMakeUpGainDb = newMakeUpGain;
}
// Main compressor processing method
void BasicCompressor::process(juce::dsp::ProcessContextReplacing<float>& context)
{
```

```
    // Get input and output blocks from the context
    auto& inputAudioBlock = context.getInputBlock();
    auto& outputAudioBlock = context.getOutputBlock();

    // Get the number of samples and channels from the output block
    const int numberOfSamples { static_cast<int>(outputAudioBlock.getNumSamples()) };
    const int numberOfChannels { static_cast<int>(outputAudioBlock.getNumChannels()) };
    // If context is bypassed, copy input block to output block without processing
    if(context.isBypassed)
    {
        outputAudioBlock.copyFrom(inputAudioBlock);
    }
    else
    {
        // If not bypassed, iterate over each channel
        for (int currentChannel = 0; currentChannel < numberOfChannels; ++currentChannel)
        {
            // Get the data for current input and output channels
            const float* inputChannelData = inputAudioBlock.getChannelPointer(currentChannel);
            float* outputChannelData = outputAudioBlock.getChannelPointer(currentChannel);

            // Iterate over each sample in the channel
            for (int currentSampleIndex = 0; currentSampleIndex < numberOfSamples; ++currentSampleIndex)
            {
                // Get the current sample
                float currentSample = inputChannelData[currentSampleIndex];

                // Compute the input level in decibels
                float inputLevelInDecibels;
                if (std::abs(currentSample) < 0.000001f)
                    inputLevelInDecibels = -120.0f;
                else
                    inputLevelInDecibels = 20.0f * std::log10(std::abs(currentSample));

                // Compute the output level in decibels
                float outputLevelInDecibels;
                if (inputLevelInDecibels >= m_thresholdLevelDb)
                    outputLevelInDecibels = m_thresholdLevelDb + (inputLevelInDecibels - m_thresholdLevelDb) /
m_compressionRatio;
                else
                    outputLevelInDecibels = inputLevelInDecibels;

                // Compute the difference between input and output levels
                float levelDifference = inputLevelInDecibels - outputLevelInDecibels;

                // Apply attack or release envelope to the level difference
                float envelopeLevel;
                if (levelDifference > m_previousEnvelopeLevel)
                    envelopeLevel = m_alphaAttack * m_previousEnvelopeLevel + (1.0f - m_alphaAttack) *
levelDifference;
                else
```

```
                envelopeLevel = m_alphaRelease * m_previousEnvelopeLevel + (1.0f - m_alphaRelease) *
levelDifference;

            // Compute the gain to be applied on the sample
            float gainForSample = std::pow(10.0f, (m_newMakeUpGainDb - envelopeLevel) / 20.0f);
            m_previousEnvelopeLevel = envelopeLevel;

            // Apply gain to the sample and write it to the output
            outputChannelData[currentSampleIndex] = currentSample * gainForSample;
        }
    }
}
```

Figure B.1: basicCompressor.cpp file.

Appendix G - BasicCompressor.h

This code presents the declaration of a class named BasicCompressor, which appears to be designed for audio signal processing, leveraging the JUCE framework. This class encapsulates the fundamental attributes and operations associated with a dynamic range compressor.

Within the private section of the class, several member variables are defined to store the state and parameters of the compressor. These include the previous envelope level, threshold level in decibels, compression ratio, attack and release times in milliseconds, make-up gain in decibels, and coefficients for attack and release. Additionally, a juce::dsp::ProcessSpec object is declared to store the specifications for the compressor.

The public interface of the BasicCompressor class offers a set of methods that facilitate the configuration and operation of the compressor. These methods allow for the preparation of the compressor with specific specifications, processing of audio data, and adjustment of various compressor parameters such as attack time, release time, threshold level, compression ratio, and make-up gain.

```
//
//  BasicCompressor.h
//  one_MBComp
//
//  UWL #21501990.
//  Copyright © 2023 Oberon Day-West. All rights reserved.
//  This code has been referenced and adapted from Reiss and McPherson (2015), Pirkle (2019) and Tarr
(2019).
//  Please refer to the accompanying report for full list of references.
#ifndef BasicCompressor_h
#define BasicCompressor_h
#include <JuceHeader.h>
class BasicCompressor
{
private:
    float m_previousEnvelopeLevel = 0.0f;
    float m_thresholdLevelDb = -10.0f;
    float m_compressionRatio = 20.0f;
    float m_attackTimeInMs = 2000.0f;
```

```
    float m_releaseTimeInMs = 6000.0f;
    float m_newMakeUpGainDb = 0.0f;
    float m_alphaAttack;
    float m_alphaRelease;

    juce::dsp::ProcessSpec m_compressorSpecifications;
public:
    void prepare(const juce::dsp::ProcessSpec& compressorSpec);
    void process(juce::dsp::ProcessContextReplacing<float>& context);
    void setAttackTime(float newAttackTime);
    void setReleaseTime(float newReleaseTime);
    void setThresholdLevel(float newThreshold);
    void setCompressionRatio(float newCompressionRatio);
    void setMakeUpGain(float newMakeUpGain);
};
#endif /* BasicCompressor_h */
```

Figure G.1: basicCompressor.h file.

Appendix H - butterworthFilter.cpp

The provided code delineates the implementation of two audio filter classes, `ButterFilter` and `LinkwitzRFilter`, utilising the JUCE framework.

The `ButterFilter` class is designed to implement a Butterworth filter, a type of signal processing filter characterised by a maximally flat frequency response. The class contains methods to prepare the filter, set filter parameters, process individual samples, and handle blocks of audio data. The filter coefficients are calculated based on the specified cutoff frequency, quality factor, and filter type (lowpass, highpass, or allpass). The class also maintains a history of previous samples to ensure continuity between successive audio blocks.

The `LinkwitzRFilter` class, on the other hand, represents a composite filter that internally utilises three instances of the `ButterFilter` class: one each for lowpass, highpass, and allpass filtering. The class provides methods to set the crossover frequency and process audio data based on the selected filter type. The processing method cascades the chosen filter type twice, offering a steeper roll-off characteristic.

Both classes ensure data integrity by validating input parameters and channel indices, and they efficiently handle multi-channel audio data by iterating over each channel and sample within the provided audio blocks.

```
//
//  butterworthFilter.cpp
//  one_MBComp
//
//
//  Copyright © 2023 Oberon Day-West. All rights reserved.
//  This code has been referenced and adapted from Bristow-Johnson (2005), neotec (2007), Falco (2009) and Zolzer (2011).
//
#include <cmath>
#include <vector>
#include <stdexcept>
#include <JuceHeader.h>
#include "butterworthFilter.h"
// Constructor definition
```

```
ButterFilter::ButterFilter(double sampleRate, FilterType type) :    filterType(type),
                                                    sampleRate(sampleRate),
                                                    previousSamples1(2, 0),
                                                    previousSamples2(2, 0)
{}
void ButterFilter::prepare(const juce::dsp::ProcessSpec& spec)
{
   sampleRate = spec.sampleRate;

   // Resize the vectors to handle the number of channels provided
   previousSamples1.resize(spec.numChannels, 0);
   previousSamples2.resize(spec.numChannels, 0);

   // Reset the vectors to 0
   std::fill(previousSamples1.begin(), previousSamples1.end(), 0.0);
   std::fill(previousSamples2.begin(), previousSamples2.end(), 0.0);
}
void ButterFilter::setFilterParameters(double cutOffFrequency, double qualityFactor, FilterType filterType)
{
//    // Validate filter parameters
//    if (cutOffFrequency <= 0 || cutOffFrequency >= 1 || qualityFactor <= 0)
//    {
//        throw std::invalid_argument("Invalid filter parameters.");
//    }
   this->cutOffFrequency = cutOffFrequency;
   this->qualityFactor = qualityFactor;
   this->filterType = filterType;
   // Calculate w0 and alpha based on cutOffFrequency and qualityFactor
      double w0 = 2 * M_PI * cutOffFrequency / sampleRate;
      double alpha = std::sin(w0) / (2 * qualityFactor);
      // Calculate coefficients based on filter type
      if (filterType == FilterType::lowpass)
      {
         double a0 = 1 + alpha;
         coefficientA0 = (1 - std::cos(w0)) / 2 / a0;
         coefficientA1 = (1 - std::cos(w0)) / a0;
         coefficientA2 = coefficientA0;
         coefficientB1 = -2 * std::cos(w0) / a0;
         coefficientB2 = (1 - alpha) / a0;
      }
      else if (filterType == FilterType::highpass)
      {
         double a0 = 1 + alpha;
         coefficientA0 = (1 + std::cos(w0)) / 2 / a0;
         coefficientA1 = -(1 + std::cos(w0)) / a0;
         coefficientA2 = coefficientA0;
         coefficientB1 = -2 * std::cos(w0) / a0;
         coefficientB2 = (1 - alpha) / a0;
      }
      else if (filterType == FilterType::allpass)
      {
```

```
        double a0 = 1 + alpha;
        coefficientA0 = (1 - alpha) / a0;
        coefficientA1 = -2 * std::cos(w0) / a0;
        coefficientA2 = (1 + alpha) / a0;
        coefficientB1 = coefficientA1; // same as a1
        coefficientB2 = coefficientA0; // same as a0
    }
    else
    {
        throw std::invalid_argument("Invalid filter type.");
    }
}
double ButterFilter::processFilter(double inputSample, int channelNumber)
{
  // Validate channel index
  if (channelNumber < 0 || channelNumber >= previousSamples1.size())
  {
    throw std::out_of_range("Invalid channel index.");
  }
  // Filter the input sample
  double outputSample = coefficientA0 * inputSample + coefficientA1 *
previousSamples1[channelNumber] + coefficientA2 * previousSamples2[channelNumber] - coefficientB1 *
previousSamples1[channelNumber] - coefficientB2 * previousSamples2[channelNumber];

  // Update previous samples
  previousSamples2[channelNumber] = previousSamples1[channelNumber];
  previousSamples1[channelNumber] = inputSample;
  previousSamples1[channelNumber] = outputSample;
  // Return filtered sample
  return outputSample;
}
void ButterFilter::updateSampleRate(double newSampleRate)
{
  sampleRate = newSampleRate;
  // Recalculate the filter parameters with the new sample rate
  setFilterParameters(cutOffFrequency, qualityFactor, filterType);
}
void ButterFilter::process(const juce::dsp::ProcessContextReplacing<float>& context)
{
  auto& inputBlock = context.getInputBlock();
  auto& outputBlock = context.getOutputBlock();
  const int numChannels = inputBlock.getNumChannels();
  const int numSamples = inputBlock.getNumSamples();
  // You'd need to loop over all channels and samples
  for (int channel = 0; channel < numChannels; ++channel)
  {
    for (int i = 0; i < numSamples; ++i)
    {
      // Process the samples and fill the outputBlock
      outputBlock.getChannelPointer(channel)[i] = processFilter(inputBlock.getChannelPointer(channel)[i],
channel);
```

```
    }
  }
}
// ==============================================================================
// Constructor definition
LinkwitzRFilter::LinkwitzRFilter(double sampleRate) :   lowPassFilter(sampleRate,
                                        FilterType::lowpass),
                                        highPassFilter(sampleRate,
                                        FilterType::highpass),
                                        allPassFilter(sampleRate,
                                        FilterType::allpass)
{}
void LinkwitzRFilter::prepare(const juce::dsp::ProcessSpec& spec)
{
  // Forward the preparation call to the inner filters
  lowPassFilter.prepare(spec);
  highPassFilter.prepare(spec);
  allPassFilter.prepare(spec);
}
void LinkwitzRFilter::setType(FilterType newType)
{
  filterType = newType;
}
void LinkwitzRFilter::setCrossoverFrequency(double crossoverFrequency)
{
//    // Validate crossover frequency
//    if (crossoverFrequency <= 0 || crossoverFrequency >= 1) {
//        throw std::invalid_argument("Invalid crossover frequency.");
//    }
  // Set crossover frequency for low pass and high pass filter
  lowPassFilter.setFilterParameters(crossoverFrequency, 0.707, FilterType::lowpass);
  highPassFilter.setFilterParameters(crossoverFrequency, 0.707, FilterType::highpass);
  allPassFilter.setFilterParameters(crossoverFrequency, 0.707, FilterType::allpass);
}
double LinkwitzRFilter::processFilter(double inputSample, int channelNumber)
{
  // Check for filter type and process accordingly
  if (filterType == FilterType::lowpass)
  {
    // Process the input sample with the low pass filter twice
    return lowPassFilter.processFilter(lowPassFilter.processFilter(inputSample, channelNumber),
channelNumber);
  }
  else if (filterType == FilterType::highpass)
  {
    // Process the input sample with the high pass filter twice
    return highPassFilter.processFilter(highPassFilter.processFilter(inputSample, channelNumber),
channelNumber);
  }
  else // allpass
  {
```

```
        return allPassFilter.processFilter(allPassFilter.processFilter(inputSample, channelNumber),
channelNumber);
    }
    return 0.0; // Default return value
}
void LinkwitzRFilter::process(const juce::dsp::ProcessContextReplacing<float>& context)
{
    auto& inputBlock = context.getInputBlock();
    auto& outputBlock = context.getOutputBlock();
    const int numChannels = inputBlock.getNumChannels();
    const int numSamples = inputBlock.getNumSamples();
    // You'd need to loop over all channels and samples
    for (int channel = 0; channel < numChannels; ++channel)
    {
        for (int i = 0; i < numSamples; ++i)
        {
            // Process the samples and fill the outputBlock
            outputBlock.getChannelPointer(channel)[i] = processFilter(inputBlock.getChannelPointer(channel)[i],
channel);
        }
    }
}
```

Figure H.1: butterworthFilter.cpp file.

Appendix I - butterworthFilter.h

The provided code delineates the declaration of two classes, `ButterFilter` and `LinkwitzRFilter`, which are designed for audio signal processing using the JUCE framework. The code is encapsulated within a header guard to prevent multiple inclusions.

The `FilterType` enumeration class defines three potential filter types: lowpass, highpass, and allpass.

The `ButterFilter` class represents a Butterworth filter. It contains private member variables to store filter coefficients, filter parameters such as cutoff frequency and quality factor, the sample rate, and vectors to retain previous samples for filtering. The public interface of this class includes a constructor to initialise the filter, methods to prepare the filter, set filter parameters, process individual samples, update the sample rate, and process blocks of samples.

The `LinkwitzRFilter` class embodies a Linkwitz-Riley filter, which is constructed using Butterworth filters. The class contains a public member for each type of Butterworth filter: lowpass, highpass, and allpass. The public interface offers a constructor, methods to prepare the filter, set the crossover frequency, process individual samples, set the filter type, and process blocks of samples.

```
//
//  butterworthFilter.hpp
//  one_MBComp
//
//
//  Copyright © 2023 Oberon Day-West. All rights reserved.
//  This code has been referenced and adapted from Bristow-Johnson (2005), neotec (2007), Falco (2009) and
Zolzer (2011).
//
```

```
#ifndef butterworthFilter_h
#define butterworthFilter_h
#include <JuceHeader.h>
#include <cmath>
#include <vector>
#include <stdexcept>
enum class FilterType
{
    lowpass,
    highpass,
    allpass
};
// ======================Butterworth=========================
class ButterFilter
{
    FilterType filterType;

    // Coefficients for Butterworth filter
    double coefficientA0, coefficientA1, coefficientA2, coefficientB1, coefficientB2;

    // Filter parameters
    double cutOffFrequency, qualityFactor;

    double sampleRate;

    // Vectors to store previous samples
    std::vector<double> previousSamples1, previousSamples2;
public:
    // Constructor
    ButterFilter(double sampleRate, FilterType type);

    void prepare(const juce::dsp::ProcessSpec& spec);
    // Set filter parameters
    void setFilterParameters(double cutOffFrequency, double qualityFactor, FilterType filterType);

    // Process input sample through filter
    double processFilter(double inputSample, int channelNumber);

    // Update method to handle changes in sample rate
    void updateSampleRate(double newSampleRate);

    void process(const juce::dsp::ProcessContextReplacing<float>& context);
};
// ======================LinkwitzRiley=========================
class LinkwitzRFilter
{
    FilterType filterType;

public:
    // Low pass and high pass Butterworth filters
    ButterFilter lowPassFilter, highPassFilter, allPassFilter;
```

```
    // Constructor
    LinkwitzRFilter(double sampleRate);

    void prepare(const juce::dsp::ProcessSpec& spec);
    // Set crossover frequency
    void setCrossoverFrequency(double crossoverFrequency);

    // Process input sample through filters
    double processFilter(double inputSample, int channelNumber);

    // Method to set the filter type
    void setType(FilterType newType);

    void process(const juce::dsp::ProcessContextReplacing<float>& context);
};
#endif /* butterworthFilter_h */
```

Figure B.1: butterworthFilter.h file.

Appendix J - basicCompressorDesign.m

The provided code implements a basic audio compressor. The script is adapted from an example in the book "Hack Audio" by Eric Tarr (2019).

The compressor operates on a step input signal, which is a sequence of zeros, followed by ones, and then zeros again, all sampled at a frequency of 48,000 Hz (Fs). The length of this signal is stored in N.

The compressor's behaviour is defined by two parameters: a threshold T set at -12 dBFS and a compression ratio R of 3. If the input signal's amplitude in dB exceeds the threshold, it is compressed; otherwise, it bypasses the compressor unchanged.

To ensure the compressor doesn't react instantaneously to changes in the input signal, a smoothing mechanism is implemented using an exponential averaging formula. The response time of this smoothing is set to 0.25 seconds, and the smoothing factor alpha is derived from this time.

The main processing loop iterates over each sample of the input signal. For each sample, the code:

- Converts the amplitude to a decibel scale.
- Checks if the amplitude exceeds the threshold.
- If it does, the compressor calculates the required gain reduction; otherwise, it bypasses.
- Smooths the gain reduction over time using the previously mentioned exponential averaging.
- Converts the smoothed gain reduction from dB to a linear scale.
- Applies the linear gain to the input sample to produce the compressed output.

Finally, the script visualises the step input signal, the compressed output, and the gain reduction over time using three subplots. Each plot spans a time of 3 seconds and an amplitude range from -0.1 to 1.1.

```
%% Basic compressor
% This script has been referenced and adapted from the
basicComp.m example
```

```
% provided in Hack Audio by Eric Tarr.
% Step input signal
Fs = 48000;
Ts = 1/Fs;
x = [zeros(Fs,1); ones(Fs,1); zeros(Fs,1)];
N = length(x);
% Parameters for compressor
T = -12; % Compressor threshold in dBFS
R = 3; % Ratio for compression
responseTime = 0.25; % Time in seconds
alpha = exp(-log(9)/(Fs * responseTime));
gainSmoothPrev = 0; % Initalise smoothing of variable
y = zeros(N,1);
lin_A = zeros(N,1);
% Loop over each sample to see if it is above threshold
for n = 1:N
%%%%% Calculations
% Turn the input signal into a unipolar signal on the dB
scale
x_uni = abs(x(n,1));
x_dB = 20*log10(x_uni/1);
% Ensure there are no values of negative infinity
if x_dB < -96
x_dB = -96;
end
% Static characteristics
if x_dB > T
gainSC = T + (x_dB - T)/R; % perform downward compression
else
gainSC = x_dB; % bypass
end
gainChange_dB = gainSC - x_dB;
% Smooth over the gainChange_dB to alter response time
gainSmooth = ((1 - alpha) * gainChange_dB) + (alpha *
gainSmoothPrev);
% Convert to linear amplitude scalar
lin_A(n,1) = 10^(gainSmooth/20);
%%%%%% Apply linear amplitude from detection path to input
sample
y(n,1) = lin_A(n,1) * x(n,1);
% Update gainSmoothPrev used in the next sample of the loop
gainSmoothPrev = gainSmooth;
end
```

```
t = [0:N-1] * Ts; t = t(:);
subplot(3,1,1);
plot(t,x); title('Step Input'); axis([0 3 -0.1 1.1]);
subplot(3,1,2);
plot(t,y); title('Compressor output'); axis([0 3 -0.1
1.1]);
subplot(3,1,3);
plot(t,lin_A); title('Gain Reduction'); axis([0 3 -0.1
1.1]);
```

Figure J.1: basicCompressorDesign.m file.

Appendix K - linkWorth.m, linkWorth_HP.m, linkWorth_AP.m

The provided code demonstrates the implementation and application of a Linkwitz/Butterworth lowpass filter on a signal composed of two sine waves. The script is adapted from various sources, and users are advised to refer to an accompanying report for detailed references. Please refer to the accompanying prototypes in the source code folder for variations on the filter types (https://github.com/0bi0n3/one_MBComp).

Initially, the script sets up a sample rate (`Fs`) of 44,100 Hz and creates a 1-second signal `x` that is the sum of two sine waves with frequencies `f1` (400 Hz) and `f2` (25 Hz). The filter's parameters, including a cutoff frequency of 300 Hz and a quality factor of 0.707, are then defined. The `setFilterParameters` function is used to compute the filter coefficients based on these parameters.

The main loop processes each sample of the signal `x` through the `processFilter` function, which applies the filter and returns the filtered sample. The result is stored in the array `y`.

The script then visualises the original and filtered signals in the time domain using two subplots. Additionally, the frequency response of the filter is plotted, showing how the filter attenuates or amplifies different frequencies.

In the final section, the magnitude spectra of both the original and filtered signals are computed using the Fast Fourier Transform (FFT) and displayed. This visualisation allows users to see how the filter has affected the frequency content of the original signal, particularly around the filter's cutoff frequency.

```
%% Linkwitz/Butterworth
%
% This code has been referenced and adapted from
Bristow-Johnson (2005), neotec (2007), Falco (2009) and
Zolzer (2011).
% Please refer to accompanying report for full reference
list and details.
% Oberon Day-West (21501990).
%%
clc; clear;
% Initialization
Fs = 44100; % Sample rate
t = 0:1/Fs:1; % 1 second of data
f1 = 4000; % Frequency
f2 = 500; % Frequency
x = sin(2*pi*f1*t); %+ sin(2*pi*f2*t); % Sum of two sine
waves
sampleRate = Fs;
filterType = 'highpass';
previousSamples1 = zeros(2, 1);
previousSamples2 = zeros(2, 1);
spec.sampleRate = Fs;
spec.numChannels = 1;
% Reset samples
previousSamples1 = zeros(spec.numChannels, 1);
previousSamples2 = zeros(spec.numChannels, 1);
```

```
% Set filter parameters (cutOffFrequency, qualityFactor,
filterType, sampleRate)
coefficients = setFilterParameters(2000, 0.707, 'highpass',
sampleRate);
% Filter the signal
y = zeros(size(x));
for i = 1:length(x)
[y(i), previousSamples1, previousSamples2] =
processFilter(x(i), 1, previousSamples1, previousSamples2,
coefficients);
end
% Plot original and filtered signal
figure;
subplot(3, 1, 1);
plot(t, x);
title('Original Signal');
subplot(3, 1, 2);
plot(t, y);
title('Filtered Signal');
% Frequency response of the filter
[H, w] = freqz([coefficients.A0, coefficients.A1,
coefficients.A2], [1, coefficients.B1, coefficients.B2]);
subplot(3, 1, 3);
plot(w/pi*(Fs/2), 20*log10(abs(H)));
xlabel('Frequency (Hz)');
ylabel('Gain (dB)');
title('Filter Frequency Response');
grid on;
% Magnitude spectra comparison
figure;
f = (0:length(x)-1) * Fs / length(x);
X = abs(fft(x));
Y = abs(fft(y));
subplot(2, 1, 1);
plot(f, 20*log10(X));
xlabel('Frequency (Hz)');
ylabel('Magnitude (dB)');
title('Magnitude Spectrum of Original Signal');
xlim([0 Fs/2]);
grid on;
subplot(2, 1, 2);
plot(f, 20*log10(Y));
xlabel('Frequency (Hz)');
```

```
ylabel('Magnitude (dB)');
title('Magnitude Spectrum of Filtered Signal');
xlim([0 Fs/2]);
grid on;
```

Figure K.1: linkWorth.m file.

Appendix L - setFilterParameters.m

The provided code defines a function `setFilterParameters` that calculates the coefficients for different types of digital filters based on the specified parameters. The function takes in four arguments: `cutOffFrequency`, `qualityFactor`, `filterType`, and `sampleRate`. Initially, it computes the normalised frequency `w0` and a value `alpha` based on the given parameters.

Depending on the `filterType` specified, which can be 'lowpass', 'highpass', or 'allpass', the function calculates the respective coefficients (`coefficientA0`, `coefficientA1`, `coefficientA2`, `coefficientB1`, and `coefficientB2`). If an unrecognised filter type is provided, the function throws an error. The calculated coefficients are then returned as a structured array.

```
%% Filter parameters
%
% This code has been referenced and adapted from
Bristow-Johnson (2005), neotec (2007), Falco (2009) and
Zolzer (2011).
% Please refer to accompanying report for full reference
list and details.
% Oberon Day-West (21501990).
function coefficients =
setFilterParameters(cutOffFrequency, qualityFactor,
filterType, sampleRate)
w0 = 2 * pi * cutOffFrequency / sampleRate;
alpha = sin(w0) / (2 * qualityFactor);
if strcmp(filterType, 'lowpass')
a0 = 1 + alpha;
coefficientA0 = (1 - cos(w0)) / 2 / a0;
coefficientA1 = (1 - cos(w0)) / a0;
coefficientA2 = coefficientA0;
coefficientB1 = -2 * cos(w0) / a0;
coefficientB2 = (1 - alpha) / a0;
elseif strcmp(filterType, 'highpass')
a0 = 1 + alpha;
coefficientA0 = (1 + cos(w0)) / 2 / a0;
coefficientA1 = -(1 + cos(w0)) / a0;
coefficientA2 = coefficientA0;
coefficientB1 = -2 * cos(w0) / a0;
```

```
coefficientB2 = (1 - alpha) / a0;
elseif strcmp(filterType, 'allpass')
a0 = 1 + alpha;
coefficientA0 = (1 - alpha) / a0;
coefficientA1 = -2 * cos(w0) / a0;
coefficientA2 = (1 + alpha) / a0;
coefficientB1 = coefficientA1; % same as a1
coefficientB2 = coefficientA0; % same as a0
else
error('Invalid filter type.');
end
coefficients = struct('A0', coefficientA0, 'A1',
coefficientA1, 'A2', coefficientA2, ...
'B1', coefficientB1, 'B2', coefficientB2);
end
```

Figure L.1: setFilterParameters.m file.

Appendix M - processFilter.m

The provided code defines a function `processFilter` that applies a digital filter to an input sample using provided filter coefficients and previous samples. The function is designed to work with multi-channel audio, as indicated by the `channelNumber` parameter.

The function takes in five arguments:

1. `inputSample`: The current audio sample to be processed.
2. `channelNumber`: The specific channel of the audio being processed.
3. `previousSamples1`: The most recent sample processed for each channel.
4. `previousSamples2`: The second most recent sample processed for each channel.
5. `coefficients`: A structure containing the filter's coefficients.

The function calculates the `outputSample` by applying the filter coefficients to the `inputSample` and the previous samples. This is done using a difference equation that represents the filter's behaviour.

After computing the `outputSample`, the function updates the previous samples for the specified channel. The most recent sample (`previousSamples1`) is updated to the current `outputSample`, and the second most recent sample (`previousSamples2`) is updated to the value of the most recent sample before the update.

The function then returns the `outputSample` and the updated values of `previousSamples1` and `previousSamples2`.

```
%% Filter processing
%
% This code has been referenced and adapted from
Bristow-Johnson (2005), neotec (2007), Falco (2009) and
Zolzer (2011).
```

```
% Please refer to accompanying report for full reference
list and details.
% Oberon Day-West (21501990).
function [outputSample, previousSamples1, previousSamples2]
= processFilter(inputSample, channelNumber,
previousSamples1, previousSamples2, coefficients)
outputSample = coefficients.A0 * inputSample +
coefficients.A1 * previousSamples1(channelNumber) +
coefficients.A2 * previousSamples2(channelNumber) -
coefficients.B1 * previousSamples1(channelNumber) -
coefficients.B2 * previousSamples2(channelNumber);
% Update previous samples
previousSamples2(channelNumber) =
previousSamples1(channelNumber);
previousSamples1(channelNumber) = outputSample;
end
```

Figure M.1: processFilter.m file.