# DAFX
# Multiband Compressor Report

Jack Walters

## 1. Introduction

The report accompanies a multi-band compressor designed in C++ in the JUCE framework. This report is ordered around the file structure of the project, with each section describing one of the four C++ or header files that serve as as foundation for the project.

## 2. PluginEditor.h

### *2.1. Public*

This header file accompanies the PluginEditor.cpp file. The CompressorAudioProcessorEditor class inherits from the AudioProcessorEditor, Timer, Slider::Listener and Button::Listener classes. Listing 1 displays the public methods used by this class for various aspects of GUI interaction and aesthetic.

Listing 1: Public methods inside the CompressorAudioProcessorEditor class

```
    void timerCallback();
    void paint (Graphics& g);
    void resized();
    void buttonClicked (Button* buttonThatWasClicked);
    void sliderValueChanged (Slider* sliderThatWasMoved);
```

### *2.2. Private*

In the private section of CompressorAudioProcessorEditor class, various pointers of the JUCE GUI class are declared. An object of type Image is also created to that will cache the GUI background image in PluginEditor.cpp.

## 3. PluginEditor.cpp

### *3.1. Constructor and Destructor*

Through inline implementation, at first all the GUI parameters and text boxes are initialised to 0. In the constructor parameters, every object declared in the header file is then initialised with its class type, colour, range etc depending on its functionality. Every GUI object that will be interacted with by the user is also set as a listener, as its value will be called in the sliderValueChanged method implementation of this file. The destructor deletes and zeros the GUI pointers and object that have been implemented, upon termination of the program.

### *3.2. ::paint and ::resized*

Listing 2: CompressorAudioProcessorEditor::paint

```
 1 void CompressorAudioProcessorEditor::paint (Graphics& g)
 2 {
 3     g.fillAll (Colour (0xff3e3a3a));
 4 
 5     g.setColour (Colours::silver);
 6     g.fillRoundedRectangle (0.0f, (float) (-1), 800.0f, 60.0f,
 7         10.0000f);
 8 
 9     g.drawImage(cachedImage_Mattblack, 0, 60, 800, 740, 0, 0, 400,
10         400, false);
11 }
```

Listing 2 shows the GUI background colour being adjusted on line 1. Lines 6-8 place a silver rounded rectangle shape to go at the top of the GUI, and in lines 10 and 11 the cached image binary is used as the input parameter for the Graphics class drawImage method.

The ::resized method sets the location and physical size of every object in the GUI.

### *3.3. Object values*

Below the ::resized method, are the two methods ::buttonClicked and ::sliderValueChanged that are tasked with receiving values from the sliders and relaying them to methods in PluginProcessor.cpp that set the values to variables used by audio processing functions. The ::timerCallBack method checks in regular intervals whether the on/off button has been clicked to ensure low latency in response time when it is interacted with.

## 4. PluginProcessor.h

### *4.1. Public*

This header file accompanies the PluginProcessor.cpp file. The CompressorAudioProcessor class declares methods that are concerned with audio processing in the PluginProcessor.cpp file.

Listing 3: Compressor value setting methods in the CompressorAudioProcessor class

```
    float getThreshold();
    float getRatio();
    float getGain();
    float getAttackTime();
    float getReleaseTime();
    float getKneeWidth();
    void setThreshold(float T, int compressorIndex);
    void setGain(float G, int compressorIndex);
    void setRatio(float R, int compressorIndex);
    void setAttackTime(float A, int compressorIndex);
    void setReleaseTime(float R, int compressorIndex);
    void setKneeWidth(float K, int compressorIndex);
```

Listing 3 shows the declaration of the methods that will be initialised in PluginProcessor.cpp. For example, setThreshold is called in PluginEditor.cpp with two parameters: the threshold and the compressor index. For these values to be received in PluginProcessor.cpp, the method first needs to be declared here.

### *4.2. Private*

Inside the private section of the CompressorAudioProcessor is where inputBuffer, processBuffer and monoBuffer (all of type AudioSampleBuffer) are declared. Crucially, compressor parameters are also declared in arrays so that each of the 4 compressors can access their index separately, as shown in Listing 4.

Listing 4: Compressor parameter array declaration

```
    float ratio[3];
    float threshold[3];
    float makeUpGain[3];
    float tauAttack[3];
    float tauRelease[3];
    float alphaAttack[3];
    float alphaRelease[3];
    float kneeWidth[3];
    float yL_prev[3];
```

In the same manner, IIR filter objects are also declared in arrays to that individual filter objects cans be indexed in the ::crossover method in CompressorAudioProcessor.

## 5. PluginProcessor.h

### *5.1. Constructor, Destructor and prepareToPlay*

Through inline implementation, the three buffers used in the audio processes are initialised to (1,1), the number of compressors is set to 4 and the last audio playhead position is set to default. The ::prepareToPlay method initialises input channel number, sample rate and buffer size variables, which are used in subsequent methods. It also allocates memory for buffers to be used in gain calculation in the ::compressor method.

### *5.2. Getters and Setters*

Lines 233 to 314 exhibit methods that retrieve values from the GUI and set them to variables that are used in the audio processing functions. GUI parameter values are received from the ::buttonClicked and ::sliderValueChanged methods in PluginEditor.cpp described in section 3.3. Listing 5 shows an example of a setter: this method is called in PluginEditor.cpp with the GUI object value and compressor index as parameters, and subsequently sets the threshold variable of the corresponding index to the GUI object value.

Listing 5: CompressorAudioProcessor::setThreshold

```
void CompressorAudioProcessor::setThreshold(float T,
    int compressorIndex)
{
    threshold[compressorIndex] = T;
}
```

### *5.3. ::crossover*

This method receives the crossover frequency and filter Q values from the GUI as variables crossoverPoints1, crossoverPoints2 and crossoverPoints3. A for loop in processBlock iterates through from 0 to (numberOfCompressors - 1) (which in this case is the equivalent of 3), and on each index a different band is created through a combination of high-pass and low-pass filters. The 4 separate bands are created through the three crossover frequencies are a low-pass at crossover frequency 1, a high-pass at crossover frequency 3, a band-pass between crossover frequencies 1 and 2, and another band-pass between crossover frequencies 2 and 3, all of which are done in a Linkwitz-Riley crossover [2]. The filter coefficients are calculated in the JUCE IIRFilter method ::setCoefficients, and then the initialised filter processes the samples in the buffer.

### *5.4. ::compressor*

This section was inspired by a combination of Reiss and McPherson's compressor code [3] and GitHub user djmoffat's implementation of this code in a multiband scenario [1]. Firstly, the attack and release parameters are defined at the beginning of every block as the time it takes for the system to reach 1-1/e of its initial value. Then, the input value is converted to decibels by being multiplied by 20log(10). This value (x_g) is then used in gain computation where knee width, threshold and ratio are all used to calculate the gain that should be applied to the output (y_g). To calculate the gain smoothing the previous value of y is used in tandem with the alphaAttack and alphaRelease values that were computed on the first two lines of the method. This method ends with its calculation of the control voltage c[j], which is applied to the buffer in ::compress.

### *5.5. ::compress*

This method begins with two precautionary if statements to ensure the threshold is not positive and the makeup gain is not negative. Using GitHub user mjmoffat's ::compress function [1] as a starting point, a mono buffer is created in the size of the buffer, and it is filled from both channels of the audio buffer (both channels being multiplied by 0.5) as they are being combined into a mono file. This mono buffer is passed through the ::compressor function and a control voltage at each sample c[i] is obtained. This control voltage c[i] is then applied to both channels of the buffer, implementing the compression, demonstrated in Listing 6.

Listing 6: CompressorAudioProcessor::compress

```
for (int i = 0; i < bufferSize; ++i)
{
    buffer.getWritePointer(0)[i] *= c[i];
    buffer.getWritePointer(1)[i] *= c[i];
}
```

### *5.6. ::compress*

This method is concerned with iterating through the channels and indexed compressor, while applying all aforementioned audio processes to them. Firstly the buffer is assigned to variable inputBuffer and then cleared. Then, a for loop iterates through the input channels and a nested for loop inside of this iterates through the indexed compressors. Inside this nested for loop a processBuffer variable is made as a copy of inputBuffer, and is passed through the ::crossover and ::compress methods. Finally this channel data is added to its corresponding channel index of the original buffer.

## References

[1] djmoffat. *JUCE-multibandCompressor*, 2014 (accessed March 3, 2020).
[2] Siegfried Linkwitz and Russ Riley. Linkwitz-riley crossovers: A primer. 2005.
[3] Joshua D Reiss and Andrew McPherson. *Audio effects: theory, implementation and application.* CRC Press, 2014.