Computer Music – Languages and Systems

HW2

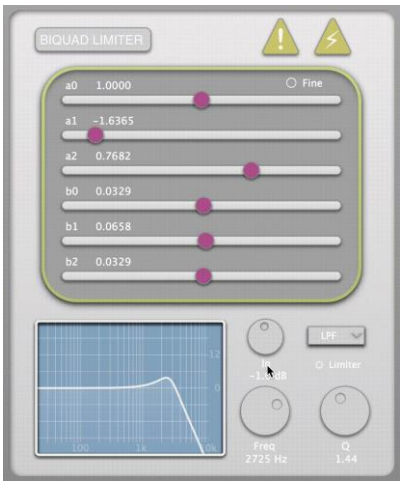
BIQUADLIMITER PLUG-IN ANALYSIS

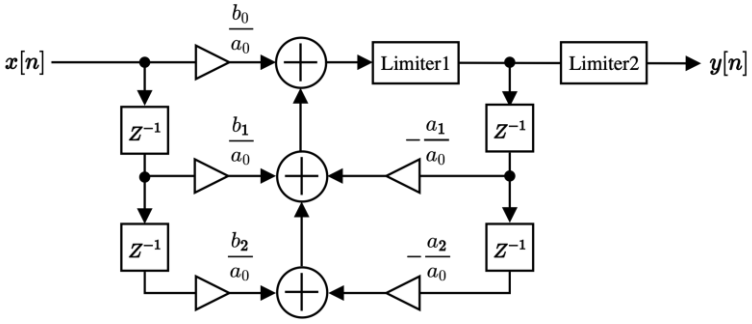
Emma Coletta 214391

Federico Ferreri 221557

Lorenzo Previati 222058

BiquadLimiter Plug-in implements a biquad filter which has a dynamic range limiter in front of the feedback section (Limiter1) and another limiter in the final section (Limiter2).





Features

- The six coefficients (a_0, a_1, a_2, b_0, b_1, b_2) of the biquad filter determine the behavior of the filter. Changing either Frequency / Q / Type will move the coefficient sliders.
- The coefficient values a_0, ..., b_2 can be set individually. This feature gives you a frequency response that is difficult to achieve with a regular 1-channel filter plug-in.
- The two limiters prevent the output from becoming too loud when the filter is unstable, or the input is too loud. In addition, the front limiter generates a unique sound by feeding the compressed sound back to the biquad filter.

From a theoretical standpoint, a digital biquad filter is a second order recursive linear filter, containing two poles and two zeros. "Biquad" is an abbreviation of "biquadratic", which refers to the fact that in the Z domain, its transfer function is the ratio of two quadratic functions:

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{a_0 + a_1 z^{-1} + a_2 z^{-2}}$$

The “Direct form 1” is the implementation presented in this plugin:

$$y[n] = \left(\frac{b_0}{a_0}\right)x[n] + \left(\frac{b_1}{a_0}\right)x[n-1] + \left(\frac{b_2}{a_0}\right)x[n-2] - \left(\frac{a_1}{a_0}\right)y[n-1] - \left(\frac{a_2}{a_0}\right)y[n-2]$$

Parameters and display items

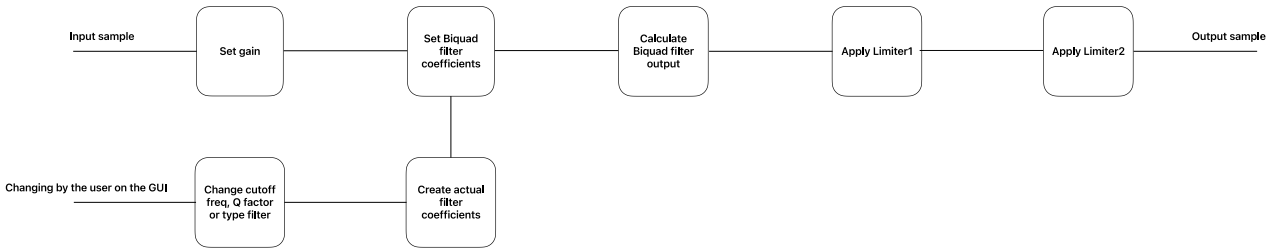
In: The input gain before the filter unit.
Freq: The cutoff frequency of biquad filter.
Q: The q of biquad filter.
Type: This is a combo box to choose filter type. You can select one among LPF (low-pass), BPF (band-pass) and HPF (high-pass).
a_0, ..., b_2: The coefficients of the biquad filter.
Fine: The button to turn on / off the fine adjustment mode of coefficient slider.
Limiter LED: This LED indicates whether the limiter is working.

Frequency display: The graph shows the ideal filter frequency response without the effect of the limiter. The y-axis shows the gain (dB) and the x-axis shows frequencies (Hz).

Code

The PluginProcessor class is the core of the Biquad Limiter plugin, responsible for processing incoming audio data and generating the output signal. It extends the AudioProcessor class which provides a set of functions for managing the plugin's input and output channels, handling parameter changes, and initializing the plugin's state. In the code we can find the use of the **AudioProcessorValueTreeState** object and the **dsp::ProcessSpec** object.
The first one stores the plugin's parameters as a tree of **AudioProcessorParameter** objects. It is useful to manage the plugin's parameters and keep their values synchronized between the user interface and the processing code.
The second one specifies the processing specifications for the plugin, including the sample rate, block size, and number of input and output channels. It is used to initialize the processing chain and ensure that it is configured correctly for the incoming audio data.

This is the general block diagram of how the plugin works:



PluginProcessor

This is the implementation file for the **BiquadLimiterAudioProcessor** which is the audio plugin processor. The constructor for the **BiquadLimiterAudioProcessor** initializes the variables:

- **parameters**: an **AudioProcessorValueTreeState** object that manages the plugin's parameters and their associated listeners;
- **defaultCoefficient**: an array that contains the default coefficient values for the biquad filter;
- **biquadLimiter**: a **BiquadFilter** object that implements the biquad filter using the default coefficient values;
- **coefIds**: parameter IDs for the coefficients;
- **coefNames**: parameter names for the coefficients;
- **coefValPtrs**: an array of pointers to the parameter values for the coefficients;
- **isGraphUpToDate**: a boolean flag that indicates whether the filter graph needs to be updated.

```
for (int i = 0; i < Bf::numCoef; ++i)
{
    parameters.createAndAddParameter (std::make_unique<Parameter> (coefIds[i], coefNames[i], ""
                                                                   , normalisableRanges[i]
                                                                   , defaultCoefficient[i]
                                                                   ,  valueToTextFunction
                                                                   ,  nullptr
                                                                   , true));

    coefValPtrs[i] = parameters.getRawParameterValue (coefIds[i]);
    parameters.addParameterListener (coefIds[i], this);
}
```

```
parameters.createAndAddParameter(std::make_unique<Parameter> ("frequency", "Freq", "", nrange3, 1.0f
                                                              ,  valueToFreqFunction,  nullptr, true));
parameters.createAndAddParameter(std::make_unique<Parameter> ("q", "Q", "", nrange3, 0.0f
                                                              ,  valueToQFunction,  nullptr, true));
parameters.createAndAddParameter(std::make_unique<Parameter> ("gain", "In", "", nrange3, 0.5f
                                                              , valueToGainFunction, nullptr));
parameters.createAndAddParameter(std::make_unique<AudioParameterChoice> ("filter_type", "Type", FILTER_TYPES, 0));
freq = parameters.getRawParameterValue("frequency");
q = parameters.getRawParameterValue("q");
gain = parameters.getRawParameterValue("gain");

parameters.addParameterListener ("frequency", this);
parameters.addParameterListener ("q", this);
parameters.addParameterListener ("filter_type", this);
```

The constructor creates also **AudioProcessorValueTreeState::Parameter** objects for each of the six Biquad filter coefficients, and adds them to the **parameters** object. The **Parameter** constructor takes the ID, the name of the parameter, its **NormalisableRange** that are used to specify the minimum and maximum values

for each of the six filter coefficients used in the Biquad filter, a default value, and a pointer to a function that converts the parameter's value to a string representation.
The **createAndAddParameter** method returns a unique pointer to the created parameter, which is used to initialize **coefValPtrs**, an array of pointers to the coefficient values. The filter coefficient parameters are added as listeners to the **parameters** object.
Then, three **Parameter** objects are defined for the filter frequency, Q, and gain, and an **AudioParameterChoice** object for the filter type. The **AudioParameterChoice** constructor takes the ID, name, and available choices for the parameter. The **createAndAddParameter** method is used to add all four parameters to the **parameters** object.

The function **prepareToPlay** is used to initialize the processor with the given sample rate and buffer size. It initializes the **previousGain** the current value of the **gain** parameter, it sets **spec** with the given sample rate and buffer size, and prepares the **clipper**, a **dsp::WaveShaper** object, by calling its **prepare** function with the given **spec** that is a **dsp::ProcessSpec** object. It initializes the processing chain based on the specifications provided in the **ProcessSpec**. Then the **isGraphUpToDate** is set to **false**.

The function **processBlock** takes the AudioBuffer as input. It sets the number of input and output channels and clears any output channels that didn't contain input data. Then, it retrieves the current gain of the plugin using the **getGainLinearVal** method. If the current gain is the same as the previous gain, the **applyGain** method is used to apply the same gain to the audio buffer. If the gain has changed, the **applyGainRamp** method is used to gradually ramp the gain up or down over the audio buffer.
Next, the coefficients for the biquad filter are retrieved and are passed to the **setCoefficient** method of the **biquadLimiter** object. This updates the coefficients of the biquad filter.
After that, the biquad limiter is applied to the audio buffer for each input channel if the channel is within the allowed range of channels through the **DoProcess** method. It then updates the working state of the biquad limiter.
At the end, if the buffer has any samples, the input buffer is wrapped using an **AudioBlock** object and a **ProcessContextReplacing** object is created using the **AudioBlock**. The **process** method of the **clipper** object is called with the **ProcessContextReplacing** object to apply the clipping function to the audio data. The result is then written back to the input buffer.

The function **setBiquadFilterCoef()** is used to update the filter coefficients of the biquad filter in response to changes in the filter type, frequency or Q values, allowing for dynamic filtering of audio signals. First, it checks if the sample rate falls within a certain range. Next, it gets the frequency and Q values from the respective **AudioProcessorValueTreeState** parameters, and it gets the current filter type from the "filter_type" **AudioParameterChoice** parameter (LPF, HPF, BPF).

Based on the calculated target frequency and Q values and the selected filter type, it creates the actual filter coefficients using the **dsp::IIR::Coefficients<float>** class. Then it stores the filter coefficients in an array called **currentCoefVals** and it sets the corresponding plugin parameter value to the appropriate value from **currentCoefVals**. The **setValueNotifyingHost()** function is used to set the parameter value, which also notifies the plugin host of the parameter change.

The function **parameterChanged** is called whenever a parameter in the audio processor is changed. It checks which parameter was changed by checking the **parameterID** argument. If the parameter is one among "frequency", "q", or "filter_type", then it calls the **setBiquadFilterCoef()** function to update the biquad filter coefficients based on the new parameter values.

BiquadLimiterDsp

The class implements a digital signal processing algorithm that applies a biquad filter to an audio signal with two limiters as shown in the above figure.

The main function of this class is the **DoProcess** which processes each audio sample by applying the biquad filter and limiters and updates the buffer with the filtered and limited audio signal.

First, it checks the sampling rate that will be used for the processing. Then, it calculates the difference between the smoothed coefficients and the current coefficients storing it in **delta_coef**:

```
delta_coef[k] = (smoothed_coef[k] - coef[k]) / bufferSize;
```

It copies the current coefficients from the **coef** vector to the **curr_coef** vector and adds the difference between the smoothed coefficients and the current coefficients for each coefficient:

```
curr_coef[k] += delta_coef[k];
```

It calculates the output of the biquad filter using the current coefficients, the input audio sample and the previous two input samples stored in **biquadFilterBuffer[channel].in1** and **biquadFilterBuffer[channel].in2**, as well as the previous two output samples stored in **biquadFilterBuffer[channel].out1** and **biquadFilterBuffer[channel].out2**:

```
float out0 = (curr_coef[3] * bufferPtr[i]
              + curr_coef[4] * biquadFilterBuffer[channel].in1
              + curr_coef[5] * biquadFilterBuffer[channel].in2
              - curr_coef[1] * biquadFilterBuffer[channel].out1
              - curr_coef[2] * biquadFilterBuffer[channel].out2
              ) / curr_coef[0];
```

The resulting output is divided by **curr_coef[0]** to normalize the signal.
Then, it applies the first limiter "**limiter1**" to the output **out0** calling the function **DoProcessOneSample** of the Limiter class and stores the result in **out0.** The output signal with the limiter applied is then buffered in **biquadFilterBuffer[channel].out1**. The **biquadFilterBuffer[channel].in1** is updated to be the current input sample.
Next, the second limiter "**limiter2**" is applied to the output signal.
The limiter1 is updated with information about the input and output signals and the current status of the limiter.
Finally, the processed output signal **out0** is written back into the input buffer **bufferPtr[i]**.

LimiterDsp

the **LimiterDsp** class is constructed with these parameters:

- threshold level in decibels (dB);
- knee width in dB, the attack time in seconds;
- the release time in seconds;
- the number of audio channels;
- a flag indicating whether to use a shared gain across all channels.

The main function is **DoProcessOneSample** that applies the limiter effect to a single audio sample.
It takes as input the **sidechainSigLinear** parameter which represents a sidechain signal used to calculate the gain reduction applied to the input signal. If this parameter is not provided, the **inputSigLinear** is used as the sidechain signal that it is the amplitude of the input signal.
First, it checks whether the limiter is in gain share mode. When gain share mode is enabled, the limiter uses a single shared gain for all channels. In this case, the **channel** parameter is ignored, and all samples are processed using channel 0.
It converts the sidechain signal from a linear value to a decibel (dB) value using the **convertLinearToDb** function.

The gain computer section calculates the static characteristic of the gain reduction curve using a soft knee function. If the sidechain signal level is below the threshold minus half the knee width, no gain reduction is applied:

```
if (InputScSigdB < threasholdDb - kneeWidthDb / 2.0f)
{
    staticCharacteristic = InputScSigdB;
}
```

If the level is within the knee region (between the threshold minus half the knee width and the threshold plus half the knee width), the gain reduction is calculated using a parabolic curve that smoothly transitions from no gain reduction to full gain reduction:

```
else if ( threasholdDb - kneeWidthDb / 2.0f <= InputScSigdB && InputScSigdB <= threasholdDb + kneeWidthDb / 2.0f)
{
    staticCharacteristic = InputScSigdB - std::pow(InputScSigdB - threasholdDb + kneeWidthDb / 2.0f, 2.0f) / (2 * kneeWidthDb);
}
```

If the level is above the threshold plus half the knee width, full gain reduction is applied:

```
else
{
    staticCharacteristic = threasholdDb;
}
```

The computed gain is obtained by computing the difference between the static characteristic and the input side chain level in dB. It is then smoothed out, acting like a time-varying low-pass-filter through the smoothing gains alphaAttack for the attack state and alphaRelease for the release state. They are calculated using the exponential function and the logarithmic conversion factor:

```
static constexpr float forwardVoltage = 90.0f;  // We define forward voltage as 90% of the peak voltage.
static const float kLogBaseConversion = forwardVoltage / 10.0f;
// alphaAttack : Smoothing gain used in attack state.
float alphaAttack = std::exp(- std::log(kLogBaseConversion) / (static_cast<float>(sampleRate) * attackTime));
// alphaRelease : Smoothing gain used in release state.
float alphaRelease = std::exp(- std::log(kLogBaseConversion) / (static_cast<float>(sampleRate) * releaseTime));
```

If the computed gain value is less than or equal to the previous smoothed gain value, the **alphaAttack** smoothing gain is used because it means that the gain is decreasing, so the filter uses it to gradually reduce the gain to the new computed value. The **alphaRelease** smoothing gain can also be used when the gain is increasing to gradually increase it to the new computed value. **gainSmoothedDb** is calculated as a weighted average of the previous smoothed gain value and the current computed gain value. The smoothing gain coefficients represent the weights applied to each value.

```
if (computedGainDb <= prevGainSmoothedDb[channel])
{
    gainSmoothedDb = alphaAttack * prevGainSmoothedDb[channel] + (1.0f - alphaAttack) * computedGainDb;
}
else
{
    gainSmoothedDb = alphaRelease * prevGainSmoothedDb[channel] + (1.0f - alphaRelease) * computedGainDb;
}
```

ClipperDsp

The aim of this class is to clip the input value through two different functions: **clippingFunctionTanh** and **clippingFunctionNormal.**

- **clippingFunctionTanh** applies a hyperbolic tangent function to the input value and compares the absolute value of the input with the absolute value of the hyperbolic tangent, clipping the first to the hyperbolic tangent of the input value where it is greater;

- **clippingFunctionNormal** does the same thing but instead of using an hyperbolic tangent function it sets a threshold of 2.0.

DspCommon

This class holds the functions **clampSampleRate()** and **defaultSampleRateIfOutOfRange()**, used to manage the input sampling rate.

The first function returns a new value that is clamped between the minimum and maximum allowed sample rates, while the second one returns a new value that is equal to the input sample rate if it is within the range of allowed sample rates, otherwise it returns the default sample rate.

We also have the function **smoothParamValue()** which by taking a new value, a previous smoothed one, and the current sample rate as inputs, returns a new smoothed value applying a smoothing filter.