POLITECNICO
MILANO 1863

052483 - COMPUTER MUSIC: LANGUAGES AND SYSTEMS

POLITECNICO DI MILANO

MUSIC AND ACOUSTIC ENGINEERING

Homework 2: Juce

Group:
Radical Geeks (ID: 7)
Authors:
Gerardo Cicalese (ID: 10776504)
Alberto Bollino (ID: 10865248)
Umberto Derme (ID: 10662564)
Giorgio Granello (ID: 10869436)

Date: May 9, 2022

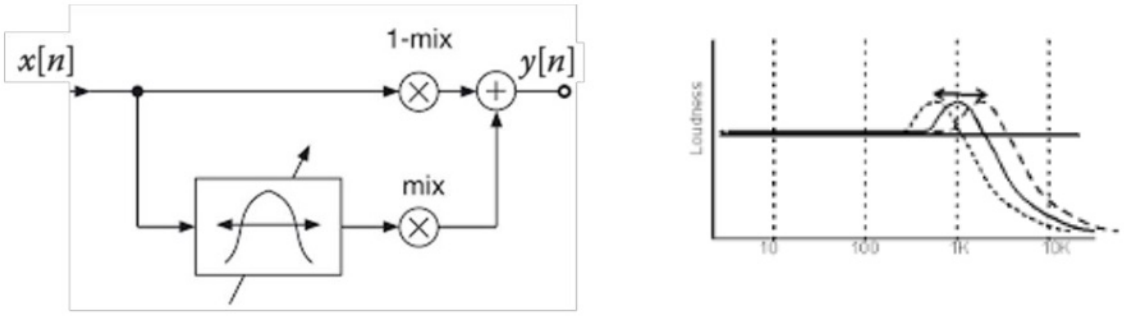

Figure 1: The figure on the left is the schema of signal path in a wah-wah effect. In the right figure we can see the frequency response of the wah-wah effect.

1 Purposes

1.1 Purposes of the project

The aim of this project is to implement a wah-wah effect plugin (see Fig. 1). The wah-wah effect consists in a passband filter, whose central frequency varies over time in a prescribed range and according to a modulating function that can be modified by a knob.
The input of the plugin is a signal (e.g. an instrument, a song, a midi instrument), and the output is the signal modified after applying the wah-wah effect. The user can choose the parameters by interacting with the GUI.

2 The wah-wah effect

The **wah-wah effect** (or simply wah effect) is an effect that alters the tone and frequencies of the input signal to create a distinctive sound, mimicking the human voice saying the onomatopeic name ”wah-wah”. More precisely, the wah-wah effect consists in a spectral glide, obtained by sweeping the peak of the frequency response of a pass-band filter (with a low quality factor) up and down in frequency.
The wah-wah effect originated in the 1920s with trumpet or trombone players finding they could produce an expressive crying tone by moving a mute in and out of the instrument’s bell. This was later simulated with electronic circuitry for the electric guitar when the wah-wah pedal was invented. This effect was used for guitar solos or just to create a funky groove (”wacka-wacka” funk-styled rhythm).

3 Implementation

The most famous device that implements the wah-wah effect is the **Cry Baby Wah**: the center frequency of the filter peak is controlled by a pedal connected to a potentiometer. Anyway, we want to implement it digitally. To do this, we have used the Juce environment.
First of all, we implemented a bandpass filter: the response of this type of filter is presented in Fig. 2.
The quality factor of the filter must be controlled by the user, to let he decide whether to emphasize more or less harmonics. If the bandwidth filter is very small, "only one harmonic will be emphasized by the narrow peak, and others will be greatly attenuated". That's how we create our peak filter, exploiting the Juce::DSP library:

```
Coefficients makePeakFilter(const ChainSettings& chainSettings, double
    sampleRate)
{
    return juce::dsp::IIR::Coefficients<float>::makePeakFilter(sampleRate,
        chainSettings.peakFreq,
        chainSettings.peakQuality,
        juce::Decibels::decibelsToGain(chainSettings.peakGainInDecibels));
}
```

It's also important to vary the center frequency of the filter, as each harmonic in a given range is approached and passed by. We have decided to automatize this process: there's a function handling the peak sweeping, which can be controlled by the user. That's how we implemented the automatic sweeping of the central frequency of the peak filter:

```
float fc, minf, maxf, sweepFreq;
minf = apvts.getRawParameterValue("Sweep Min Freq")->load();
maxf = apvts.getRawParameterValue("Sweep Max Freq")->load();
sweepFreq = apvts.getRawParameterValue("Sweep Freq")->load();
// Update filter parameters depending on phase
fc = (minf + maxf)/2 + (maxf - minf) / 2 * sweepFunction(phase);

// updating phase modulo 2PI
phase += 1/sr * sweepFreq;

if (phase > 2 * PI)
    phase -= 2 * PI;

// updating peak center frequency
apvts.getRawParameterValue("Peak Freq")->store(fc);
```

The user can choose the velocity of the sweep, the range and the function; the sweeping can be described by an harmonic signal or by a more complex signal (triangle,

sawtooth, inverted sawtooth and rectangle). That's how we have implemented these signals (shown in the figures 3, 4, 5, 6 and 7):

```
float YetAnotherAutoWahAudioProcessor::sweepFunction(float phase)
{
    auto sweepType = apvts.getRawParameterValue("Sweep Type")->load();

    // SINE
    if(sweepType == 0)
        return sin(phase);

    // TRIANGLE
    if (sweepType == 1) {
        if (phase < PI)
            return 2 * phase / PI - 1;
        else
            return 3 - 2 * phase / PI;
    }

    // SAWTOOTH
    if(sweepType == 2)
        return 1 - phase / PI;

    // INVERTED SAWTOOTH
    if (sweepType == 3)
        return phase / PI - 1;

    // RECTANGLE
    if (sweepType == 4)
        if (phase < PI)
            return 1;
        else
            return -1;

    return 0;
}
```

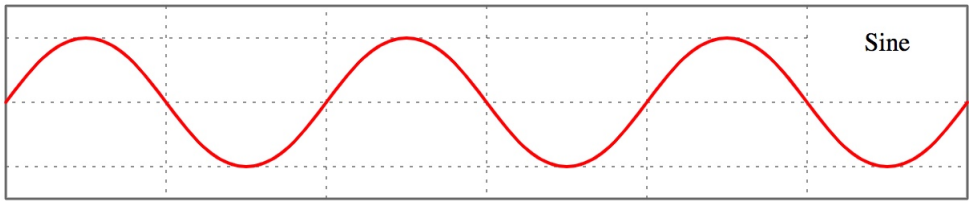

Figure 3: Sine wave.

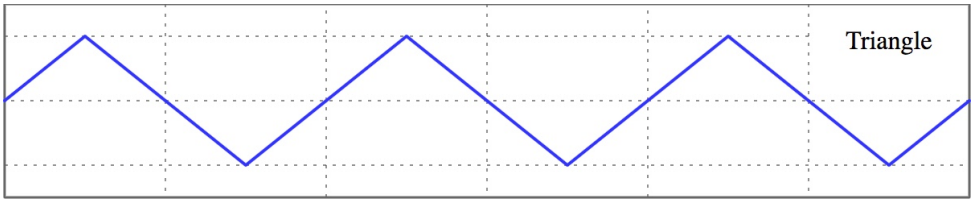

Figure 4: Triangle wave.

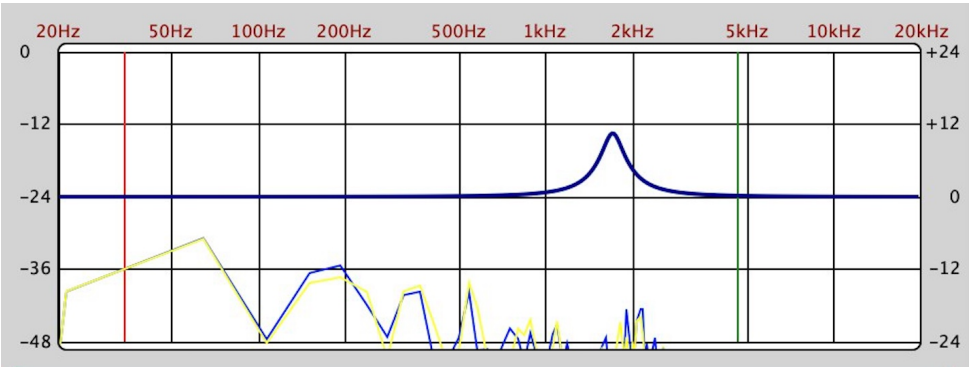

Figure 2: Frequency response of the band-pass filter used in our plugin for gain = $10.5dB$ and $Q = 3.6$.

Then, we added a Wet/Dry knob, in order to decide the percentage of the input signal that must be affected by the wah-wah effect. That's the code snippet where we copy the input into a buffer, process it and perform a weighted sum in order to implement the Wet/Dry feature:

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

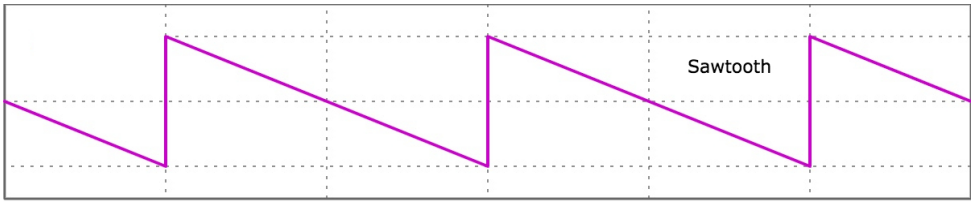

Figure 5: Sawtooth wave.

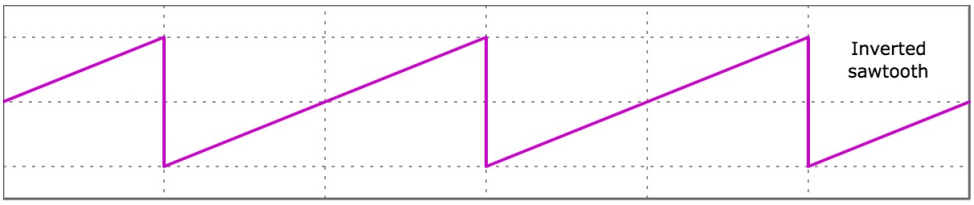

Figure 6: Inverted sawtooth wave.

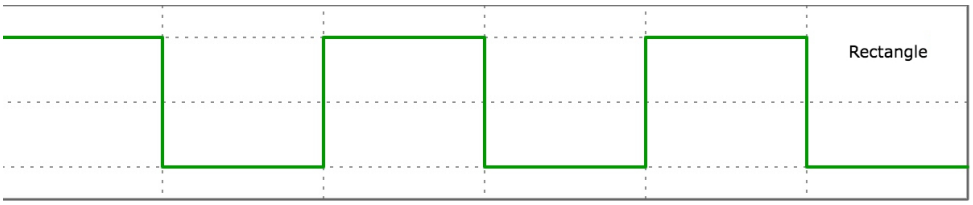

Figure 7: Rectangle wave.

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

// allocating buffer for wahwah wet
tempBufferLeft.copyFrom(leftBlock);
tempBufferRight.copyFrom(rightBlock);

auto mix = apvts.getRawParameterValue("Mix")->load() / 100;

// wet = mix
tempBufferLeft.multiplyBy(static_cast<float> (mix));
tempBufferRight.multiplyBy(static_cast<float> (mix));

juce::dsp::ProcessContextReplacing<float> leftContext(tempBufferLeft);
juce::dsp::ProcessContextReplacing<float> rightContext(tempBufferRight);

leftChain.process(leftContext);
rightChain.process(rightContext);

// dry = 1 - mix
leftBlock.multiplyBy(static_cast<float> (1.0 - mix));
rightBlock.multiplyBy(static_cast<float> (1.0 - mix));

// summing wet and dry
leftBlock.add(tempBufferLeft);
rightBlock.add(tempBufferRight);
```

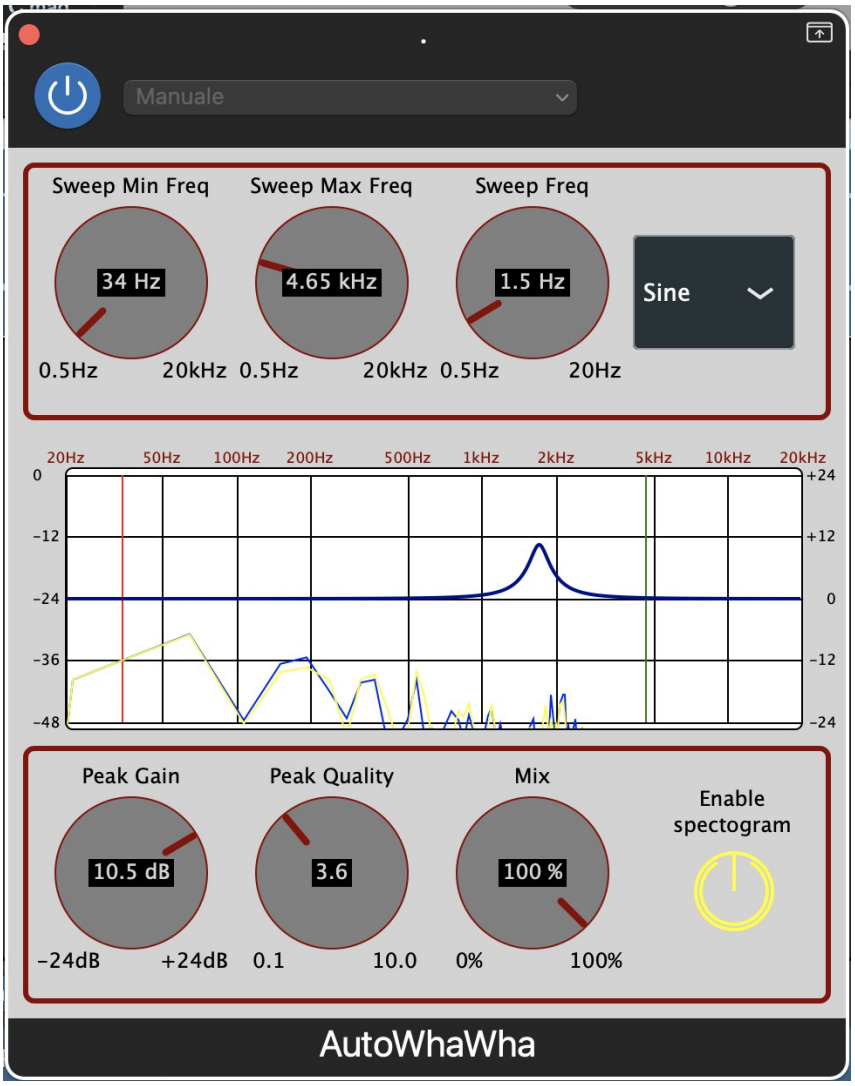

Figure 8: Screenshot of the GUI.

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

Finally, we have decided to let the user see what is doing: it's difficult to design an effect if you can't see the frequency response of the filter. So, we added a spectrogram view of the filter response function, which can also show the input signal's spectrum (FFT).

3.1 GUI

The GUI (Fig. 8) is user friendly and it's made of the following components:

- On the first row, there are two knobs controlling the limits (in the frequency do-

main) of the peak sweep, then there's the knob controlling the sweep velocity (frequency of the sweep function) and, finally, a drop-down menu to select the sweep function.

- At the center, there's a spectrogram visualization of the filter and, eventually, of the input signal.
- On the last row, there are two knobs controlling respectively the gain and the quality of the peak, then we have the Wet/Dry knob and, at last, a button that toggles the spectrogram visualization of the input signal.