Mu45 Final Project Report

Haizhou Yu-szz5ft

Project Description

The Wobbler is a plugin dedicated to produce a wobble effect on any instruments. This could be synths, bass, leads, or even other samples like vocals. It was inspired by Xfer Records' "*LFOTool*". The reason why I designed it is that I wanted to make a straightforward, simple yet useful plugin to create the effect of wobble chords and more commonly used in the genres of Future House, Future Bass, and more.

Signal Flow

This is an LFO-based plugin, which has a set of LFOs to control a gain of the track, to achieve the effect. Presented below is the signal flow diagram of the plugin. The two channels have two different LFOs and gains, and the LFO controls what I would call "pre gains", which then fed into a set of Low Pass filters, and the wet and dry signals are combine together.

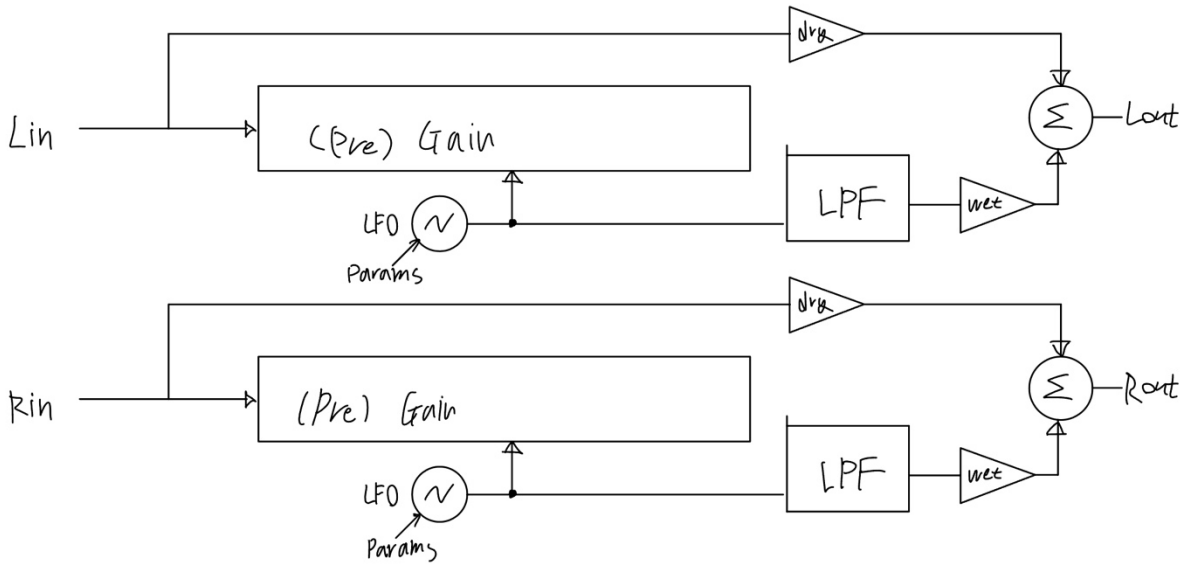

Figure 1: Signal Flow Diagram of the Wobbler

Parameters

The plugin has the following parameters:

- Attack – This controls the shift of the function which advances or retards the start of the trigger for the LFO. Range will be from 100 to 0, which is mapped to +0.8 to -0.8. This range is what I found useful using a desmos graphing calculator. The formula for the mapping is:

$$Shift = \frac{(parameter - 50) * N(number\ of\ samples)}{62.5 * 2}$$

- Speed – This controls the speed which the curvature of the curve, which could be curve up or down. This modulates the exponent of the function. Range will be from 0 to 100. 90 represent a linear function, while 1 - 89 is a curve that concaves up, 91 – 99 is a curve that concaves down. 100 is a square wave. The formula for the mapping is:

$$Power\ of\ the\ function = \frac{100 - parameter}{10}$$

- Rate – This will control the frequency of the LFO. The range is from 0.1Hz to 20Hz, with spacing of 0.1Hz. Anything over 20Hz will be too not useful.
- Rate Choice – This will control the frequency of the LFO, but the units are in notes. The choices are: 1/2, 1/4, 1/8, 1/16, 1/32, and their corresponding triplet notes.
- Rate Reference– This tells the plugin if it should stick with a Hz based or note based reference of the rate. The two choices are “Hz” and “Note”.
- Mix – This controls the mix of the dry and wet signals. It is a constant volume mixer. The algorithm is shown below:

```
// Ensure mix is within the valid range [0, 100]
float mix = std::max(0.0f, std::min(100.0f, MixPerc));
// Convert mix percentage to a scale factor in the range [0, 1]
float mixFactor = mix / 100.0f;
// Calculate dry and wet gains to maintain constant perceived volume
mDryGain = sqrt(1.0f - mixFactor);
mWetGain = sqrt(mixFactor);
```

- High Cut Frequency – This controls the cutoff frequency of the Low Pass filter. The range is from 30 to 22000Hz.

GUI

The following is a GUI of the Wobbler:

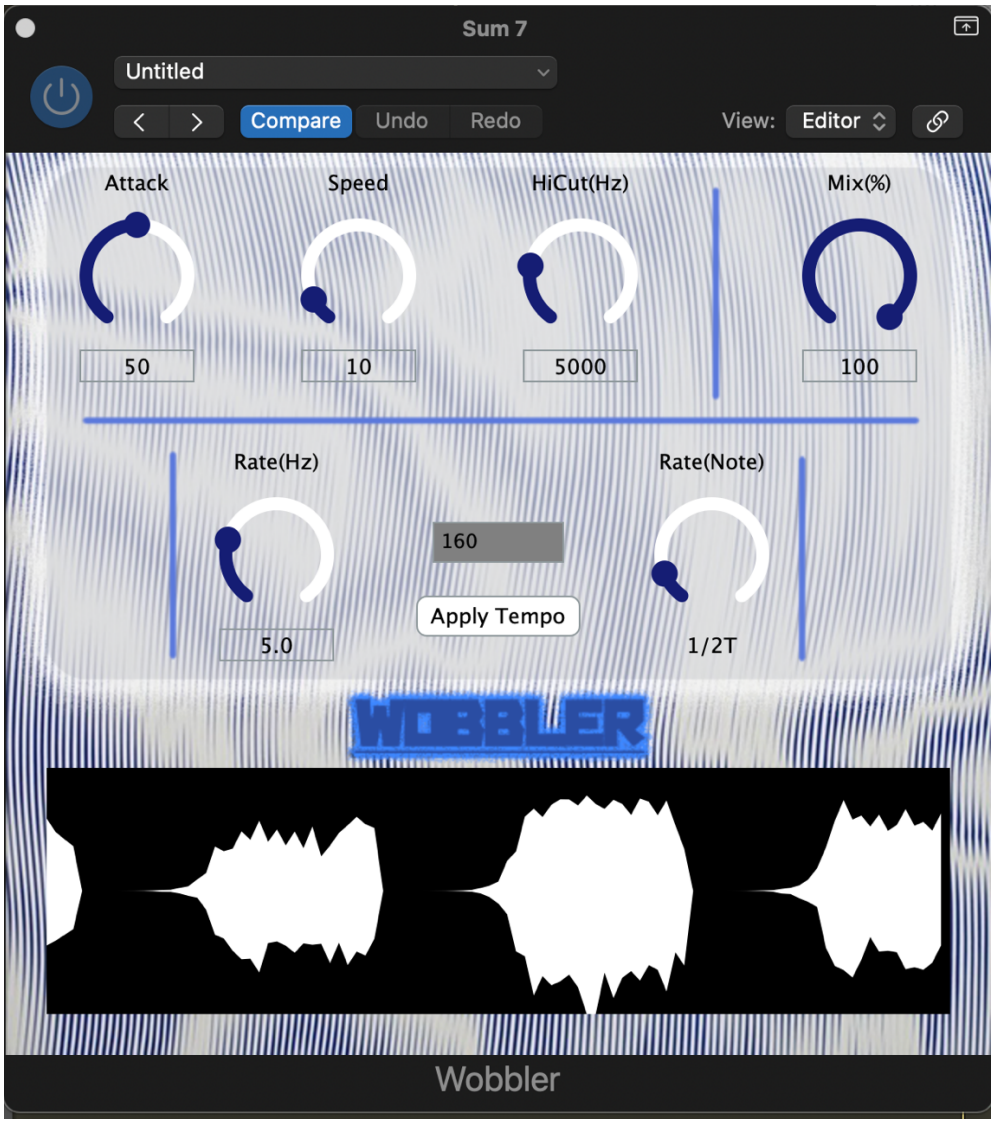

Figure 2: GUI Design of the Wobbler

I first layout the sliders, buttons, and visualizer for the plugin. I grouped the speed, attack, and high cut on the top right corner, because they are they define the shape of the waveform and the frequency spectrum. The mix knob is on its separate to the right, like the design on the kickstart. The rate sliders, BPM label and buttons are down one row. It is in a symmetrical design for better looks. After All this finished, I looked for a background picture to start the design. I came across this blue, wavy looking pattern, that reminds me of the word “wobble”. I used Photoshop to create different layers: one for the background, one for the sliders’ region with the round edge rectangular design, and one for the “Wobble” word logo and the bars in between the sliders to separate their region. I changed a lot of opacity, and blending with shadow, glow, and the background was done. I also changed the colors of the sliders and buttons, so they matched the color scheme, I even went into the detail of changed the RGB values of the color to fine tune them. The color of the rails, knobs, text, textbox background of the slider, and the color of the textbox text, textbox background, also the color of the text of the text-button and its background, as all been customized.

It took me one whole day from starting to design the GUI to finish, and I really put a lot of work and effort as well as craftsmanship into it, and I hope everyone else will be pleased by the looks of it as what I do.

Functions

The heart of the program is my LFO class that was modified out of our MU45LFO class. I added one crucial helper function, updateParams, which the code is shown in the figure below.

```
void Mu45LFO::updateParams(float attackParam, float speedParam, float rate, float mFs)
{
    if (speedParam == 100)
        attackParam += 60.0;

    double offset = ((attackParam - 50.0) / 62.5) < 0 ? 0 : ((attackParam - 50.0) / 62.5) * N / 2.0;
    for (int i = static_cast<int>(offset); i < N; i++) {
        if (i <= (N / 2 + (((attackParam - 50.0) / 62.5) * N / 2.0)) && i > static_cast<int>(offset)) {
            table[i] = pow((i - ((attackParam - 50.0) / 62.5) * N / 2.0) / (static_cast<double>(N) / 2), (100.0 -
                speedParam) / 10.0);
        } else if (i > (N / 2 + (((attackParam - 50.0) / 62.5) * N / 2.0))) {
            table[i] = 1.0;
        } else {
            table[i] = 0.0;
        }
    }
    setFreq(rate, mFs);
}
```

Figure 3: Code snippet of the updateParams function

This function constructs a new LFO waveform on the fly, so it changes the sound effect whenever the user makes an input. I also made adapts to the N constant, which is the size of the wavetable, so if the laptop is not powerful enough, a smaller size could be used instead.

The transient visualizer will give a user a clue on how the effect is affecting the sound. Here are two pictures that compare the waveform output in the visualizer with all the same setting, but different attack settings.

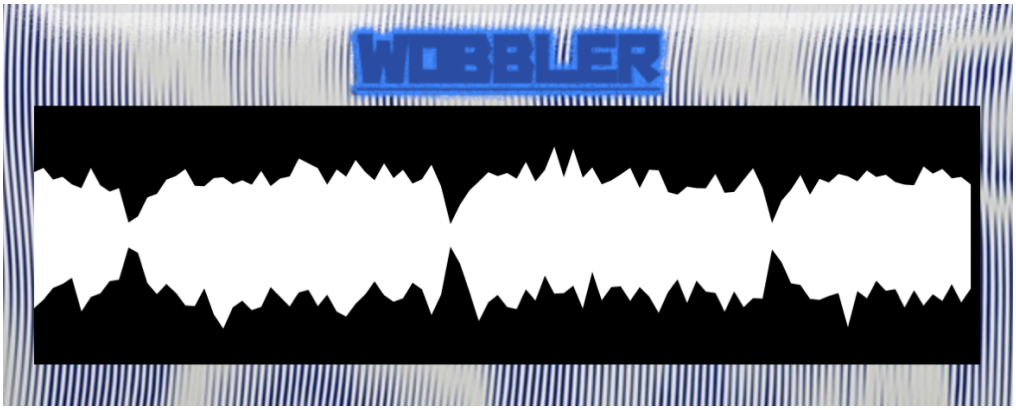

Figure 4: Visualizer when the attack is at 5

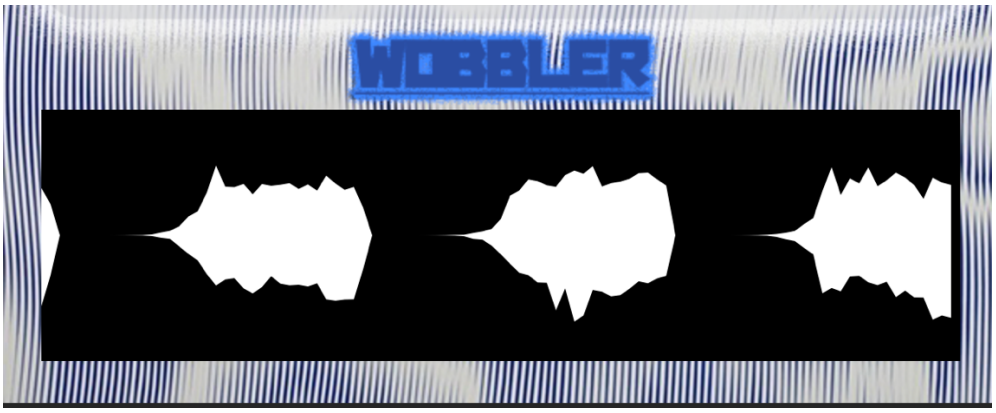

Figure 5: Visualizer when the attack is at 50

Because different frequency will produce a different period of the waveform, so there isn't any good timeframe for the visualizer, so it displays a good clear, magnified transient analysis for all the time. Therefore, I wrote a code that adjust the number of samples in the visualizer on the fly. I also made my refresh rate very slow, for 1.1Hz (1Hz will produce error somehow), so the users can see clearly the waveform without it flying around in a high frequency. I really made a lot of trials and errors on this, and the whole visualizer took me half a day to research and make it into reality. This is shown in the code snippet below:

```
Visualizer.setBufferSize((1/(rate->get()))*512);
```

Figure 6: Code snippet for the variable buffer size on the visualizer

I will be talking about how I made the Rate Choice sliders next. I initially tried to make custom sliders that have snapping function and a custom region, but later I found out that there is a type of variable in JUCE called "AudioParameterChoice". I used this for both my rate choice parameter and my rate reference parameter. The way I set them up is shown in the figure below:

```
addParameter(RateChoices = new juce::AudioParameterChoice ("Rate Choices", "Rate Choices", { "1/2", "1/2T", "1/4",
    "1/4T", "1/8", "1/8T", "1/16", "1/16T", "1/32", "1/32T" }, 2));
addParameter(HzOrNote = new juce::AudioParameterChoice("Rate Reference", "Rate Reference", {"Hz", "Note"}, 0));
```

Figure 7: Code snippet for the AudioParameterChoice setup

In order to make the slider work accordingly (because there is technically no range), I set the slider range to be the range of the index in the rate parameter choice. By doing this, the slider will act like a snap-on slider, which is very cool. Then, I made a separate label for displaying the rate choice, which is stored as string, besides the slider. I made a lot of conversion back and forth to sync this from the plugin processor and the editor. All of these are shown in the calcAlgorithmsParams function, which I highly recommend the reader to take a look. All in all, this whole part took me another full day to get it work.

Flaws

There are a couple flaws to the program, I admit. Since this is a route that no one takes, which is full of unknown, I know there are going to have lots of challenges. First, the rate choice would not work without the GUI opened, and my DAW, Logic, will crash. This may be caused some public variables that shared across the GUI (PluginEditor) and the PluginProcessor are not treated well (maybe memory errors or something that is in conflict).
There is a flaw that is minimal yet also major in terms of affecting the sound quality. Since there is no way for me to read the playhead information from the DAW (at least I tried with the official JUCE tutorial, and even I only declared the variables with two lines of code, my DAW crashes whenever the plugin is loaded). This means the frequency is not in sync with the beat, because I cannot reset the phase accordingly to the beat, so sometimes the LFO will be not in sync with the tempo. Also, that means I have to take in tempo information manually. If the user change the rate on every upbeat, this could be fine, but again this somewhat limits the usage of the plugin.

Testing

I will be attaching a python program that generates the waveform with two parameters. That is how I test my MU45 LFO class's algorithm. I've also recorded a video to explain how the effect works in action, with explaining how changing some parameters will lead to certain effects.