CMLS Homework 2 - Group 9

10724182 10743181 10766268 10768481

https://github.com/mttbernardini/cmls-hw2

1 Introduction

The aim of this document is to describe the realization of a flanger effect plug-in with feedback. For the implementation we relied on JUCE [1], a partially open-source cross-platform `C++` application framework.

Section 2 contains a detailed description of the algorithm of our digital flanger, starting from a simple delay line to a stereo flanger with identical settings except the phase of the low-frequency oscillator. Section 3 contains a detailed discussion on different interpolation techniques, used to handle fractional delay times in a discrete domain. Section 4 refers instead to the graphic user interface and to the management of user controls.

2 Flanger

Flanging is an audio effect produced by mixing two identical signals together, where one signal is delayed by a small and gradually changing period. It gets its name from the original analog method of running two tape machines slightly out of sync in respect to each other. It is based on the principle of constructive and destructive interference of signals.

2.1 Frequency response

A clear comprehension of the flanger, from a perceptive point of view, is highlighted by the frequency response of the comb filter. At the heart of the algorithm, comb filter is the simplest example of a recursive delay network. In fact, the flanger is a recursive comb filter with an interpolating and time-varying delay line. Changes in delay time correspond to the characteristic way peaks in the frequency response move up and down in frequency [3].

The comb filter is obtained by feeding a g_{FB} fraction of the output back to the input. Its typical difference equation is:

$$y[n] = g_{FB}\, y[n - M[n]] + x[n] + (g_{FF} - g_{FB})\, x[n - M[n]],$$

where g_{FF} is the *depth* of the flanger, and $M[n]$ is a function describing variable delay times.

In the flanger case, it is represented by a Low Frequency Oscillator, for each channel, of different and selectable waveforms. Thus, the frequency response of the flanger is equal to:

$$H[z] = \frac{1 + z^{-M[n]}\ (g_{FF} - g_{FB})}{1 - z^{-M[n]}\, g_{FB}}$$

By looking at the absolute value of the frequency response of the non-recirculating comb filter with $g_{FB} = 0$

$$|H(z)| = \sqrt{1 + 2 g_{FF} cos(\omega M[n]) + g_{FF}^2}$$

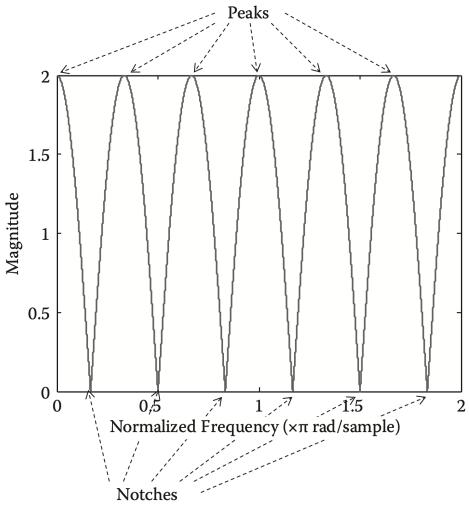


Figure 1: Frequency response of a flanger

we can appreciate the classical spectrum with peaks and notches lying respectively in

$$\omega_p = \frac{2\pi p}{M} \qquad p = 0, 1, 2, \ldots, M-1$$

and

$$\omega_n = \frac{(2n+1)\pi}{M} \qquad n = 0, 1, 2, \ldots, M-1$$

It is easy to see the dependence between enhanced frequencies and delay times of the flanger. The flanger is stable while $g_{FB} < 1$.

Our plug-in provides the feature to invert the polarity of the g_{FF} parameter. In this case, peaks and notches in the frequency response will trade places, with the first notch lying at DC frequency. Because of it, when switching to the negative polarity you can hear a thinner sound unless M is large.

2.2 Implementation

In order to reproduce the analog effect of the flanger given by the heads of the tape machine, a delay line with a read and a write pointer must be implemented. In the code we defined them as two variables dr and dw. The read index moves away from, then back to, the top or starting point of the delay. When the read pointer and the write one are superimposed there is no delay, while when they are moving in respect to each other there is a change of pitch. Looking at the algorithm, a loop over all samples is implemented to calculate, for each iteration, the distance between the two pointers and store the delay length into a buffer. More details can be found in the comments of the `FlangerProcessor::processBlock()` member function in the `PluginProcessor.cpp` file.

As can be seen from the block diagram in figure 2, our plug-in has a mono input that is duplicated in a stereo output. The wet lines are initially filtered by a high-pass filter (described in section 2.3) and then sent to the delay blocks. They are controlled by two distinct LFOs for each channel, with an adjustable relative phase to spread the stereo image. At the end, the wet and the dry lines are summed up to provide the two outputs $y_L[n]$ and $y_R[n]$.

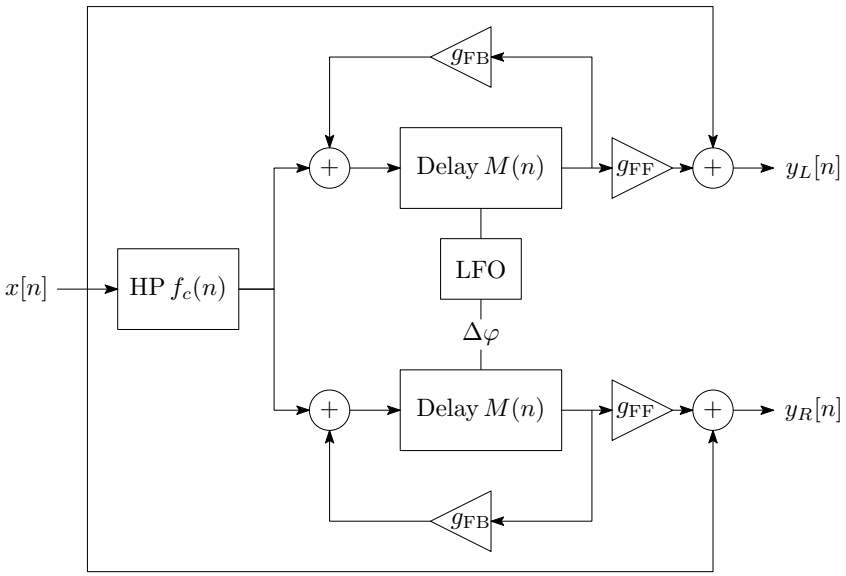


Figure 2: Complete flanger block diagram

2.3 High-Pass Filter

With high values of feedback gain and depth, we can occur in the risk of saturation, in particular at low frequencies. In order to avoid that, we decided to add a high-pass filter before the delay lines, by implementing a first-order recursive filter:

$$y[n] = \alpha \cdot (y[n-1] + x[n] - x[n-1]) \qquad 0 \leq \alpha \leq 1$$

where α has the same role as the time-constant of an analog RC circuit. In particular, it is related to the cut-off frequency f_c as:

$$\alpha = \left(\frac{2\pi f_c}{F_s} + 1\right)^{-1}$$

2.4 Parameters

The flanging effect is created by mixing the dry signal and the wet one (in fact if only the signals with modulated delays were sent out, we would have a sort of simple frequency modulation that would lead to a vibrato effect). In short, mixing an audio signal with a slightly delayed copy of itself results in a different frequency spectrum due to the suppression of some wave components and the enhancement of others.

Depth The mixing ratio between the two signals is a fundamental aspect of this type of effect. The control of the *depth* value, expressed in percentage, is used to decide the amount of wet signal to mix to the dry one. In particular 0% corresponds to $g_{FF} = 0$ and produces no effects, while 100% guarantees the maximum effect.

Feedback Another user adjustable parameter is the value of the *feedback gain*. It is expressed in percentage as well. We decided to set the range of this parameter from 0 to 99% to remain in a stable condition and prevent the risk of distortion of the sound.

Low Frequency Oscillators As said before, each delay line is controlled by an LFO. We decided to set another degree of freedom for the user and we introduced some other user-adjustable parameters related to it. In particular *sweep width* is the value of its amplitude, expressed in milliseconds, in a range from 0 to 25 ms. It is also possible to modify the *LFO frequency*, from 0 to 10 Hz. This parameter is useful to act on the speed of the read pointer.

The plug-in also provides the possibility to choose the type of wave. Besides the default "sinusoidal" wave, it is possible to set a "square", "sawtooth", "inverse sawtooth", "triangular" or a "random" wave. This part, is implemented with a simple `switch` over a `OscFunction` enum in the member function `FlangerProcessor::waveForm()` in the `PluginProcessor.cpp` file.

A final parameter is the *LFO phase R/L*, ranging from 0 to 360°, which allows the user to modify phase displacement in case the wave function is not random. In the latter case, this parameter is replaced by *LFO width*, which is a percentage representing how much the right channel should differ from the right one, by making a linear combination between the left-channel random value and another right-channel random value.

Cut-off frequency For the high-pass filter, the cut-off frequency is provided as a final user adjustable parameter, in a range from 20 Hz to 5 kHz. We observed that setting a higher frequency is not useful, as it would leave only the dry signal go through.

3 Interpolation

Digital delay lines are implemented using a memory buffer of discrete audio samples. To change the delay time, we change the distance in the buffer between where samples are written and where they are read back [4]. Separate read and write pointers are used and additionally, for good quality audio, it is necessary to interpolate the delay-line length rather than "jumping" between integer numbers of samples. This is typically accomplished using an interpolating read to make the delay line vary smoothly over time [5]. Since the delay of the flanger changes by small amounts each sample, it will inevitably take fractional values, so interpolation is always used when calculating the delayed signal of the flanger.

In our work we tried implementing both linear interpolation and polynomial interpolation and the second one seemed to be the smoothest one. We implemented them by writing the member function `FlangerProcessor::interpolate()` in the `PluginProcessor.cpp` file.

Linear Interpolation Linear interpolation is the most commonly used case because it is easy to implement and inexpensive. It works by effectively drawing a straight line between two neighboring samples and returning the appropriate point along that line (see figure 3).

This is given in the following equation:

$$x(t) = (1 - (t - n)) \cdot x[n] + (t - n) \cdot x[n + 1] \qquad n \leq t < n + 1$$

where $x[n]$ and $x[n+1]$ are two successive samples and $(t-n)$ is a fractional value representing the difference between the reader pointer dr and its floor rounding $\lfloor dr \rfloor$.

Polynomial Interpolation In a polynomial interpolation the function is estimated to be an Nth-order polynomial. A curve is drawn between the points (or a series of points), and then the interpolated value is found on that curve [2].

$$x(t) = c_N t^N + c_{N-1} t^{N-1} + \cdots + c_1 t^1 + c_0 \qquad n \leq t < n + 1$$

interpolation of sample values.png

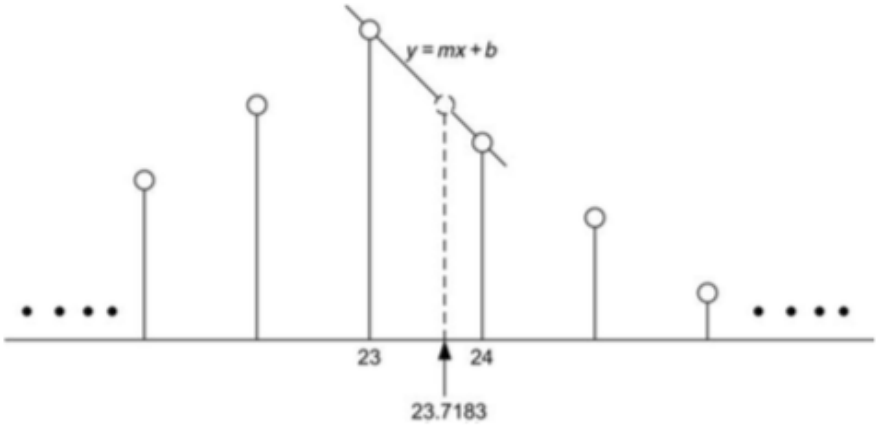


Figure 3: Linear interpolation of sample values

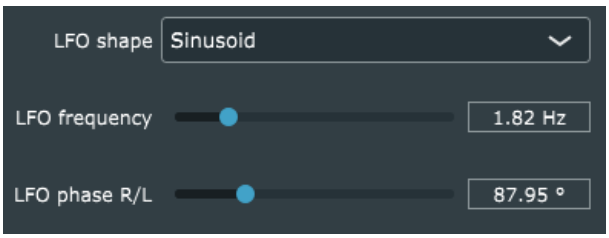
In particular, second-order polynomial interpolation considers three successive samples surrounding the interpolated location: $x[n-1]$, $x[n]$, and $x[n+1]$.

$$x(t) = c_2(t-n)^2 + c_1(t-n) + c_0 \qquad n < t < n+1$$

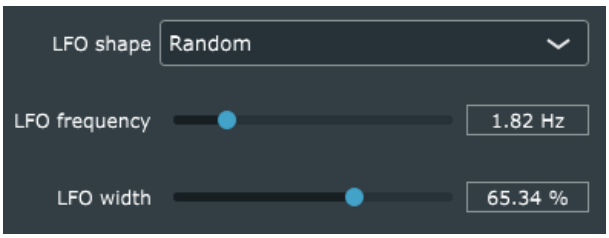
$$\begin{cases} c_0 = x[n] \\ c_1 = \dfrac{x[n+1] - x[n-1]}{2} \\ c_2 = \dfrac{x[n+1] - 2x[n] + x[n-1]}{2} \end{cases}$$

4 Interface

User interaction is accomplished through a simple Graphical User Interface, providing the most of the controls as sliders. The upper area controls the LFO parameters. In figure 4 we can notice how the controls are always relevant to the selected waveform, i.e. for the "random wave" the "phase" slider is replaced with a "width" slider (see section 2.4 for more details on LFO parameters).



(a) Sinusoidal shape



(b) Random shape

Figure 4: Different parameters depending on the LFO shape

The remaining controls affect the main effect parameters.

Sweep width controls the peak delay of the LFO;

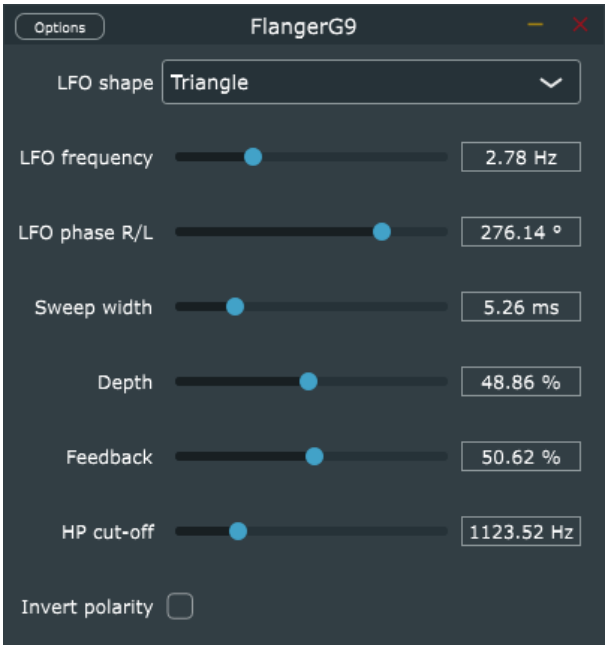


Figure 5: Complete Graphical User Interface

Depth controls the intensity of the effect, like a dry/wet control, and corresponds to the gain level of the delay line;

Feedback controls the intensity of the feedback, corresponds to the gain of the feedback loop;

HP cut-off controls the cut-off frequency of the high-pass filter;

Invert polarity controls whether the wet signal is added or subtracted from the dry signal.

The full GUI as it is presented to the user is shown in figure 5.

4.1 Implementation Details

In order to avoid code duplication, we adopted some techniques to define the GUI using a more descriptive approach rather than an imperative approach.

In particular, each `Slider` is dynamically generated in a `for` loop, which iterates over an array of `UISliders` objects describing the properties of each slider (i.e. label, unit, range and binding functions). The initial `ComboBox` is populated by iterating over a `std::map` which associates each `OscFunction` enum value to the corresponding label. The final `ToggleButton`, being just one, is added directly.

Each `Component` is then added to an `Array`, which is iterated over during the `resized()` method, and laid down using a vertical main-axis `FlexBox`, to prevent hard-coding numerical values for bounds and sizes. For the styling we just kept JUCE's default `LookAndFeel`, being already clear enough for this purpose.

Value changes are bound using lambda-functions, instead of subclassing `Listener`, for easier coding. In particular, the logic of alternating the display between the two "phase" and "width" `Sliders` is accomplished by keeping a map of "mutually exclusive" `Components`. The correct `Slider` is then displayed when the `ComboBox::onChange` lambda-function is invoked.

Value unit conversion is finally performed by `FlangerProcessor`'s setter/getter methods of each parameter, following the encapsulation pattern to bridge between UI and internal representations.

References

[1] Juce documentation. `https://docs.juce.com/master/index.html`.

[2] William C Pirkle. *Designing audio effect plug-ins in C++ with digital audio signal processing theory*. Taylor & Francis, 2013.

[3] Miller Puckette. Theory and techniques of electronic music. *Online] http://www-crca. ucsd. edu/˜ msp*, 2006.

[4] Joshua D Reiss and Andrew McPherson. *Audio effects: theory, implementation and application*. CRC Press, 2014.

[5] Julius Orion Smith. *Physical audio signal processing: For virtual musical instruments and audio effects*. W3K publishing, 2010.