[DSP Class] Final Project: Time-Lag Filter Audio Effect

Sauraen (sauraen@gmail.com)

May 2015

Abstract

A new digital audio effect is characterized, developed, and tested, both on an embedded DSP processor and as a software audio plugin. The theoretical foundation is discussed, and experimental results (both quantitative and qualitative) on each version are compared. The system is parameterized and deployed as an audio effect for use in workstation-based digital audio production.

1 Introduction

The Time-Lag Filter is a new, original audio effect that comes from a linear, time-invariant (LTI) or linear, time-varying (LTV) system. The system can be completely characterized with the traditional LTI models; however, it is expected that the user will vary the system parameters during playback, and the system will support this variation, making it a LTV system. The system is implemented both on the TI F28335 DSP controller as a hardware digital audio effect with analog input and output, and as a VST[1] audio plugin using the JUCE[2] C++ framework for use in professional software audio production.

The name Time-Lag Filter was chosen as an extension of a Phase-Lag Filter. The system has a flat amplitude response, but a group delay at high frequencies that is so extreme that those frequencies are delayed in time by an audible amount. This is accomplished by stringing together several dozen (or several hundred!) second-order all-pass filters, and taking the output from the end of the chain. Unlike a comb filter, the outputs from different stages are never mixed together, so no frequencies interfere destructively.

The system has three time-variable parameters:

1. The amount of delay at high frequency; that is, the number of all-pass filters in the chain. For ease of implementation, the largest number of filters will always be running; the system will simply take the output from a filter earlier in the chain, if this parameter is not at maximum.

2. The center frequency of the phase lag of each filter. This is equivalent to the center band frequency for a band-pass filter.

3. The width of the sloped section of the phase response of each all-pass filter, which is equivalent to the bandwidth for a band-pass filter.

In addition, in the software version of this system, the user can cause the latter two parameters to vary over the filters in the chain, according to several preset patterns with adjustable parameters.

[1]The VST format is an open standard developed by Steinberg: https://www.steinberg.net/en/company/technologies/vst3.html

[2]http://www.juce.com/

2 System theory

2.1 Continuous time

Transfer function The transfer function for a second-order all-pass filter is

$$h(s) = \frac{s^2 - 2\zeta\omega_0 s + \omega_0^2}{s^2 + 2\zeta\omega_0 s + \omega_0^2} \tag{1}$$

The poles are at

$$\begin{aligned} 0 &= s^2 + 2\zeta\omega_0 s + \omega_0^2 \\ s &= -\zeta\omega_0 \pm \frac{\sqrt{4\zeta^2\omega_0^2 - 4\omega_0^2}}{2} \\ &= \omega_0(-\zeta \pm \sqrt{\zeta^2 - 1}) \end{aligned}$$

Since we want the real part of these to be negative, we place the condition $\zeta \in (0, \mathrm{inf})$, and the poles are at

$$\omega_p = -\zeta\omega_0 \pm j\omega_0\sqrt{1 - \zeta^2} \tag{2}$$

if $\zeta < 1$, and

$$\omega_p = -\zeta\omega_0 \pm \omega_0\sqrt{\zeta^2 - 1} \tag{3}$$

if $\zeta >= 1$.

Amplitude response The amplitude response is

$$\begin{aligned} |h(j\omega)| &= \left| \frac{(j\omega)^2 - 2j\zeta\omega_0\omega + \omega_0^2}{(j\omega)^2 + 2j\zeta\omega_0\omega + \omega_0^2} \right| \\ &= \left| \frac{(\omega_0^2 - \omega^2) - 2j\zeta\omega_0\omega}{(\omega_0^2 - \omega^2) + 2j\zeta\omega_0\omega} \right| \\ &= \sqrt{\frac{(\omega_0^2 - \omega^2) - 2j\zeta\omega_0\omega}{(\omega_0^2 - \omega^2) + 2j\zeta\omega_0\omega} \cdot \frac{(\omega_0^2 - \omega^2) + 2j\zeta\omega_0\omega}{(\omega_0^2 - \omega^2) - 2j\zeta\omega_0\omega}} \\ &= 1 \end{aligned}$$

which proves that this is indeed an all-pass filter.

2.1.1 Phase response

The Time-Lag Filter was designed to have a characteristic phase response. The phase response of the whole system, composed of all the filters in a chain, is simply the sum of the phase responses of all the filters individually. This is because, for an input signal $u(\omega)$ and a string of LTI systems whose frequency responses are given by $A_n(\omega)e^{j\phi_n(\omega)}$, the output $y(\omega)$ is

$$\begin{aligned} y(\omega) &= u(\omega)\prod_{n=0}^{N} A_n(\omega)e^{j\phi_n(\omega)} \\ &= u(\omega)\cdot\left(\prod_{n=0}^{N} A_n(\omega)\right)\cdot e^{j\sum_{n=0}^{N}\phi_n(\omega)} \end{aligned}$$

Phase delay The phase delay is simply the phase component of the frequency response, with negative values representing delays:

$$h(j\omega) = \frac{(\omega_0^2-\omega^2)-2j\zeta\omega_0\omega}{(\omega_0^2-\omega^2)+2j\zeta\omega_0\omega} \tag{4}$$

Let $a=\omega_0^2-\omega^2$, $b=2\zeta\omega_0\omega$. Then

$$\begin{aligned} h(j\omega) &= \frac{a-jb}{a+jb} \\ &= \frac{\sqrt{a^2+b^2}\cdot e^{j\cdot\tan^{-1}(-b/a)}}{\sqrt{a^2+b^2}\cdot e^{j\cdot\tan^{-1}(b/a)}} \\ &= \frac{e^{-j\cdot\tan^{-1}(b/a)}}{e^{j\cdot\tan^{-1}(b/a)}} \\ &= e^{-2j\cdot\tan^{-1}(b/a)} \end{aligned}$$

and

$$\phi(h(j\omega)) = -2\tan^{-1}(b/a) \tag{5}$$

Group delay However, the phase, and thus the phase delay, of a single sine wave in a single audio channel is not audible—in fact, the waveforms for a 1-cycle delay and a 256-cycle delay are identical. In order for a time delay to be audible, several frequencies must be present in the input signal, whether a wide range of frequencies forming a transient, or many closely

spaced frequencies forming a wave packet. Here, what counts is not the phase delay of any of the component frequencies, but the relative delay of certain frequencies to others. For instance, a "sine wave pulse", that is a wave packet composed of a range of frequencies around a center frequency $\omega_0 \pm d\omega$, will be delayed in time a large amount if its component frequencies are shifted relative to each other by a large amount $d\phi$. Since those frequencies only occupy a range $d\omega$, we are interested in $d\phi/d\omega$, and we define the group delay as

$$\Delta_g = k \cdot \frac{d\phi}{d\omega} \tag{6}$$

Since higher frequencies have a more negative phase delay in the phase response above, $d\phi/d\omega$ is always negative for this system, and it is more negative the quicker the phase delay is changing; so we define $k = -1$ and ignore the units.

We now find the equation for the group delay at each frequency for a single filter:

$$\begin{aligned}
\Delta_g &= -1\frac{d\left(-2\tan^{-1}(b/a)\right)}{d\omega} \\
&= 2 \cdot \frac{1}{1+(b/a)^2} \cdot \frac{d(b/a)}{d\omega} \\
&= 2 \cdot \frac{1}{1+(b/a)^2} \cdot \frac{d}{d\omega}\frac{2\zeta\omega_0\omega}{\omega_0^2-\omega^2} \\
&= \qquad\qquad \vdots \\
&= \frac{4\zeta\omega_0\left(\omega_0^2-3\omega^2\right)}{a^2+b^2}
\end{aligned} \tag{7}$$

2.2 Discrete time

2.2.1 Parameterization

The parameters for a single filter are given in terms of the center frequency $f_0 = \omega_0/2\pi$ and the bandwidth ζ. Translating the former to discrete time using the bilinear transform is trivial; however, the latter is distorted, and the translation is more easily accomplished by means of analogy to that of a band-pass filter:

$$f_0 = \sqrt{f_H f_L} \tag{8}$$

$$B = 2\zeta f_0 = f_H - f_L \tag{9}$$

Since the parameters f_0 and ζ are the ones being varied, the inverse relations must be found:

$$\begin{aligned} f_0^2 &= f_L(f_L + B) \\ 0 &= f_L^2 + Bf_L - f_0^2 \\ f_L &= \frac{-B \pm \sqrt{B^2 + 4f_0^2}}{2} \\ &= -\zeta f_0 \pm \frac{\sqrt{4\zeta^2 f_0^2 + 4f_0^2}}{2} \\ &= f_0(-\zeta \pm \sqrt{\zeta^2 + 1}) \end{aligned} \tag{10}$$

The negative solution here gives a negative frequency for f_L, which we are not interested in, so we take the positive one

$$f_L = f_0(-\zeta + \sqrt{\zeta^2 + 1}) \tag{11}$$

and from Equation (9)

$$f_H = f_L + 2\zeta f_0 \tag{12}$$

Now these parameters can be converted into discrete time:

$$\theta_L = 2\pi \frac{f_L}{f_s} \tag{13}$$

$$\theta_H = 2\pi \frac{f_H}{f_s} \tag{14}$$

$$\Omega_L = \tan(\theta_L/2) = \tan\left(\pi \frac{f_L}{f_s}\right) \tag{15}$$

$$\Omega_H = \tan(\theta_H/2) = \tan\left(\pi \frac{f_H}{f_s}\right) \tag{16}$$

and

$$B_a = \Omega_H - \Omega_L \tag{17}$$

$$\Omega_0 = \sqrt{\Omega_H \Omega_L} \tag{18}$$

2.2.2 Signal flow algorithm

In the fake S domain,

$$H_a(S) = \frac{S^2 - B_a S + \Omega_0^2}{S^2 + B_a S + \Omega_0^2} \tag{19}$$

and by the bilinear transform,

$$H(z) = \frac{\left(\frac{z-1}{z+1}\right)^2 - B_a\left(\frac{z-1}{z+1}\right) + \Omega_0^2}{\left(\frac{z-1}{z+1}\right)^2 + B_a\left(\frac{z-1}{z+1}\right) + \Omega_0^2} \tag{20}$$

After a page's worth of handwritten algebra,

$$H(z) = \frac{\frac{\Omega_0^2 - B_a + 1}{\Omega_0^2 + B_a + 1} + \frac{2(\Omega_0^2 - 1)}{\Omega_0^2 + B_a + 1} z^{-1} + z^{-2}}{1 + \frac{2(\Omega_0^2 - 1)}{\Omega_0^2 + B_a + 1} z^{-1} + \frac{\Omega_0^2 - B_a + 1}{\Omega_0^2 + B_a + 1} z^{-2}} \tag{21}$$

Let

$$a = \frac{1}{\Omega_0^2 + B_a + 1},\ b = a \cdot 2(\Omega_0^2 - 1),\ c = a \cdot (\Omega_0^2 - B_a + 1) \tag{22}$$

so the coefficients can be calculated (from Ω_0 and B_a) with only one floating-point divide and only three floating-point multiplies. Then

$$H(z) = \frac{c + bz^{-1} + z^{-2}}{1 + bz^{-1} + cz^{-2}} \tag{23}$$

and this can easily be translated into a signal flow algorithm as

$$\begin{aligned} y_n &= cu_n + bu_{n-1} + u_{n-2} - by_{n-1} - cy_{n-2} \\ &= c(u_n - y_{n-2}) + b(u_{n-1} - y_{n-1}) + u_{n-2} \end{aligned} \tag{24}$$

meaning the actual calculation for the filtering only takes two floating-point multiplies and four adds.

2.2.3 Multi-filter implementation

The actual signal-processing subroutine for the embedded and plugin implementations is nearly identical, the only major exception being that for the plugin implementation, b and c are arrays, one for each filter, while in the embedded implementation, they are variables (constant throughout the subroutine over all filters).

To store the values of u and y, the program uses two circular arrays, one for the us and one for the ys, each of length three times the number of filters. At any given time, each filter uses a block of three of each, representing u_{n-2}, u_{n-1}, u_n, y_{n-2}, y_{n-1}, and y_n. Each filter stores its input into the element for u_n; uses that and the four delayed values to calculate y_n; and stores that back

in its appropriate element. Then, once all the filters have completed their processing, the starting index in the circular arrays is incremented by one, so that on the next frame the element previously holding u_n will be considered u_{n-1} and so on, with each filter overwriting what would have been the next filter's u_{n-3}.

This circular addressing requires that the filter can address coefficients off the end of the array and onto the beginning, which is why the actual indices `n`, `n_1`, and `n_2` are precalculated and wrapped to the correct range. In addition, if the filter currently being processed is the one whose output is desired, the current value of y_n is assigned to be returned; this feature is not used on the embedded version, due to the lack of a third analog control input to select the filter index, but the filter itself supports it.

See the Appendix for the filter code.

3 Embedded implementation

3.1 Overview

The Time-Lag Filter was implemented as an embedded system on the TI Peripheral Explorer Board for the F28335 DSP controller. Stereo audio inputs and outputs are provided by the AIC23 audio codec, and the two onboard potentiometers are used to adjust the center frequency and bandwidth in realtime. Initial testing was performed with 64 filters in the chain and at a sampling frequency of 8kHz; however, this sample rate was inadequate for extensive aural verification of the system's operation, and for final testing the system used 32 filters and sampled at 32kHz. The system uses only the left channel input (discarding the right channel input data); it sends the filtered signal to the left channel output, and the unmodified input signal to the right channel output.

3.2 System configuration

The F28335 runs native C code (developed in TI's Code Composer Studio) with no real-time operating system. Code optimization level 1 was used to resolve some issues that caused minor nonlinearities in the output (due to timing issues between different ISRs); code optimization level 3 was tried, but the output was not correct and the system did not function at all.

Three interrupts (and hence ISRs) are present. The first is from a CPU core timer, and simply services the watchdog. The second is from the transmit frame-sync interrupt of the McBSP serial communication interface to the AIC23 codec, which serves as the sample clock. This ISR reads the last received sample from the audio inputs, processes it through the filter (see Section 2.2.3), and sends the resulting sample to the codec to be output. The final ISR is from the analog-to-digital converter having finished reading the states of the potentiometers; it is triggered from an ePWM unit at 250Hz, which serves only as a timer. This ISR simply stores the potentiometer values in global variables and returns as quickly as possible.

The main background loop, in addition to servicing the watchdog, continually monitors the variables storing the potentiometer states, which correspond to the center frequency and bandwidth. When these parameters change by a significant amount (more than noise or error from the ADC), it recalculates the filter coefficients from these parameters according to Equations (11) through (18).

3.3 Testing and results

3.3.1 Qualitative aural testing

In addition to the measurements of the system's performance using standard equipment, its audio input was hooked to the headphone output of a smartphone, and its audio output was hooked to a pair of very-low-end computer speakers. Music from several different genres (rock, folk, electronic) was played through the system; the system was configured so that the unfiltered left channel was sent to the left speaker, and the filtered left channel was sent to the right speaker.

The effect of the system was clearly audible, though not nearly as intense as that of the plugin version. The primary effect that was noticeable was a "stereo spreading" effect between the two speakers, due to phase differences between the two (which varied over frequency). The human ear can detect even small phase differences between two audio sources; phase differences and echoes are the largest effects the brain uses to interpret the three-dimensional position of a sound source, with volume differences between the two ears being a relatively minor effect. (A monaural sound played in headphones is perceived to be coming from the center of the listener's head; panning the sound solely by changing the volume causes it to "move" in a straight line be-

tween the ears, but as soon as the phase of the two signals starts diverging, the listener perceives the sound as being from somewhere outside them.[3]) Since in this system, different frequencies have their phases changed by different amounts, the effect is that the sound is coming from many directions at once, though it also sounds somewhat artificial.

Changing the center frequency and bandwidth in realtime caused an even more noticeable phase effect, as the audio sounded like it was moving around in space around the listener. It also introduced some pitch shifting, proportional to the rate of change of the parameters (but not at all while they were in steady state). This is theoretically expected, as while the system parameters are changing, it is a linear time-varying system, which is equivalent to a nonlinear system, so the output can contain frequencies not present in the input. In addition, the effect of the filter chain is to be a sort of delay line, and a chorus is created by varying the recording or playback speeds from a short delay line. This is not an exact analogy, however, as varying the parameters of the Time-Lag Filter does not change the way the signals enter or exit the filter chain, only the parameters of the filters.

3.3.2 Measurement

All tests were performed at a sampling rate of 32kHz and with a filter chain of 32 filters.

Phase mangling A square wave of approximately 700Hz is the system's input. The top scope trace is the system's unfiltered output of that signal, and the bottom is the filtered version. The bottom waveform has a shape that is completely unrecognizable as a square wave, but its spectrum (shown in the second picture) remains the same. Three sets of images are shown, corresponding to three different (arbitrary) settings of the system's center frequency and bandwidth; the spectrums of the filtered output signals are visibly almost identical.

[3]Results of personal research; actual (scholarly) research pending.

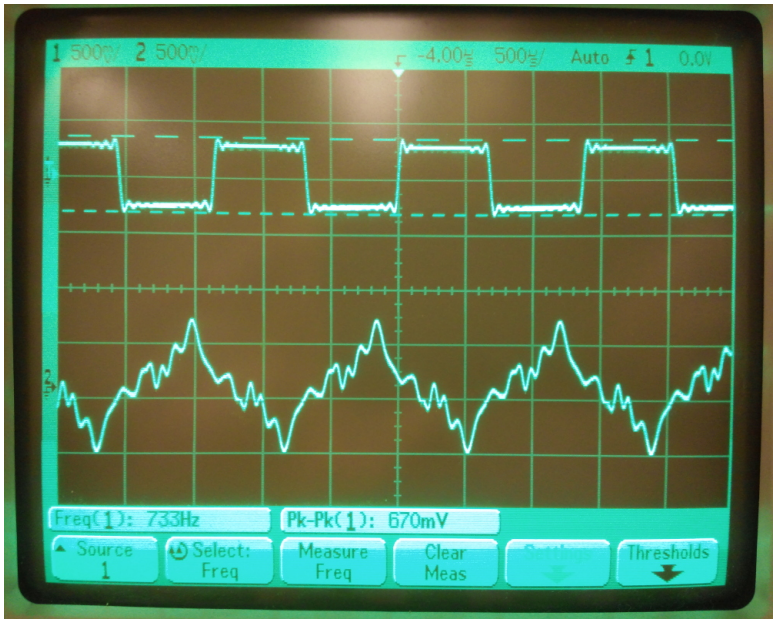

Figure 1: Unfiltered output and filtered output, for the first set of parameters.

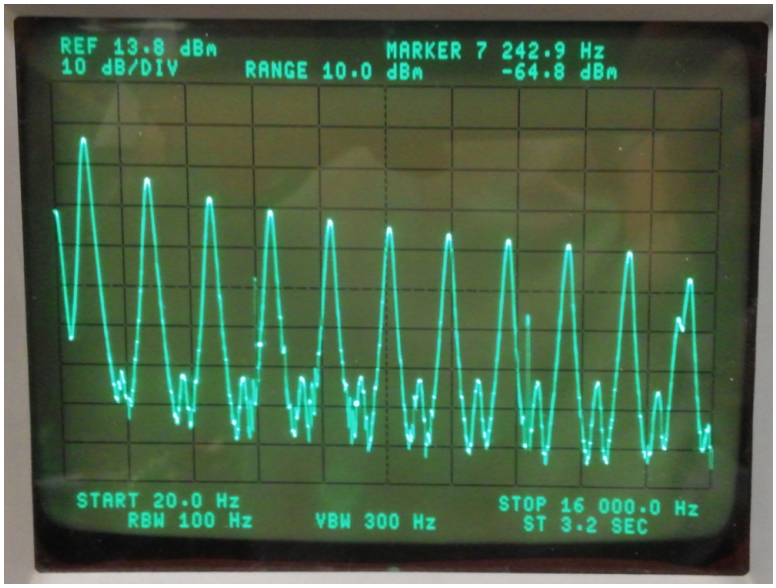

Figure 2: The spectrum of the lower signal in Figure 1.

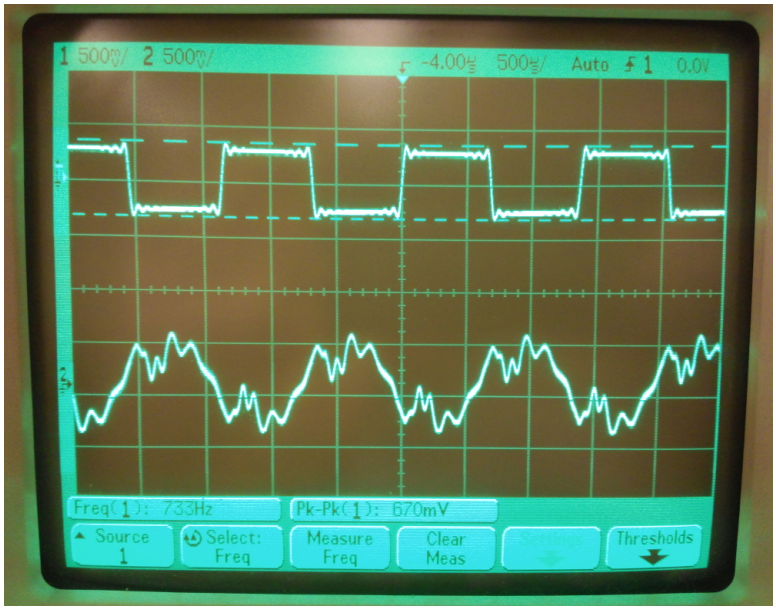

Figure 3: Unfiltered output and filtered output, for the second set of parameters.

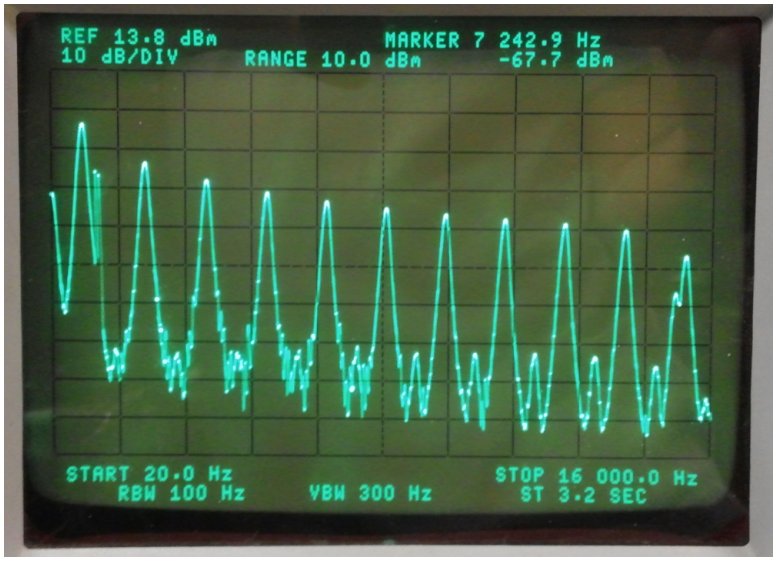

Figure 4: The spectrum of the lower signal in Figure 3.

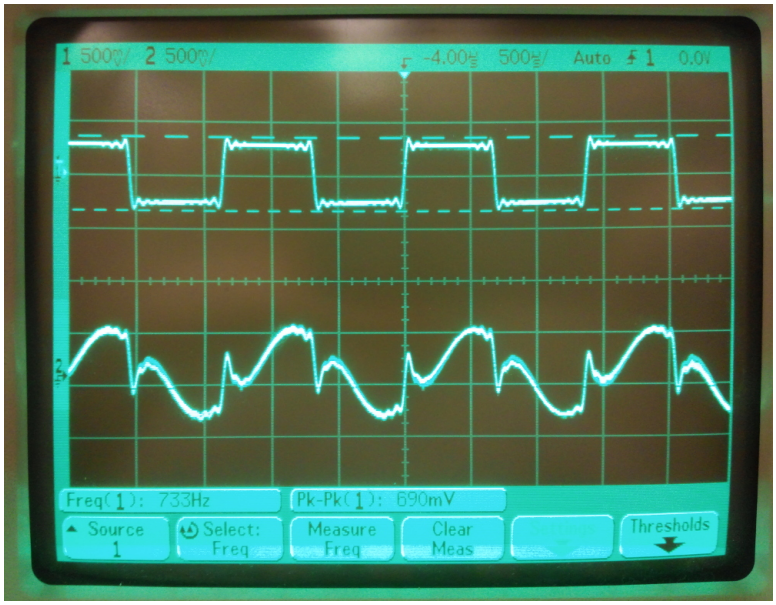

Figure 5: Unfiltered output and filtered output, for the third set of parameters.

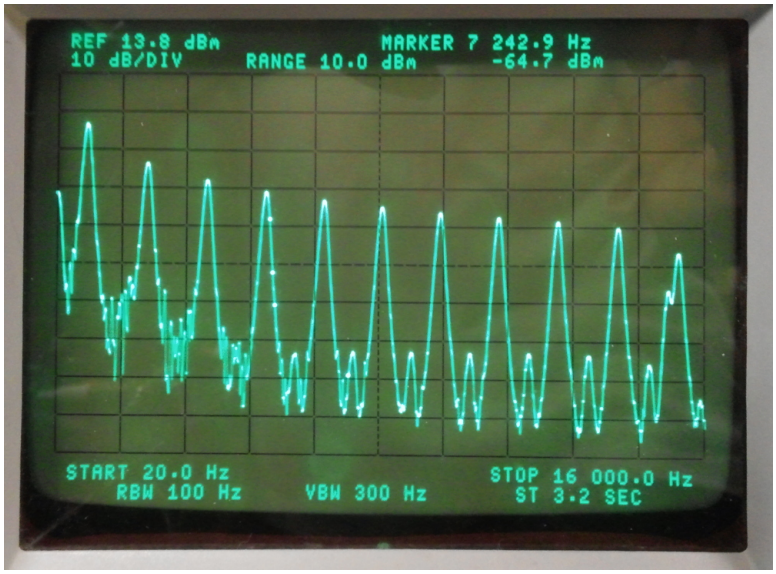

Figure 6: The spectrum of the lower signal in Figure 5.

Amplitude response and nonlinearities For the following tests, the system input is connected to the sweep output of the spectrum analyzer, so it is actually graphing the system's amplitude response. Each image corresponds to a different set of center frequency and bandwidth parameters. For the purposes of these results, the resonance $Q = \frac{1}{2\zeta}$ will be discussed instead of the bandwidth. The system allows the center frequency to range from 10Hz to about 4kHz, and the bandwidth to range from 2 to 0 (Q from 0.25 to ∞).

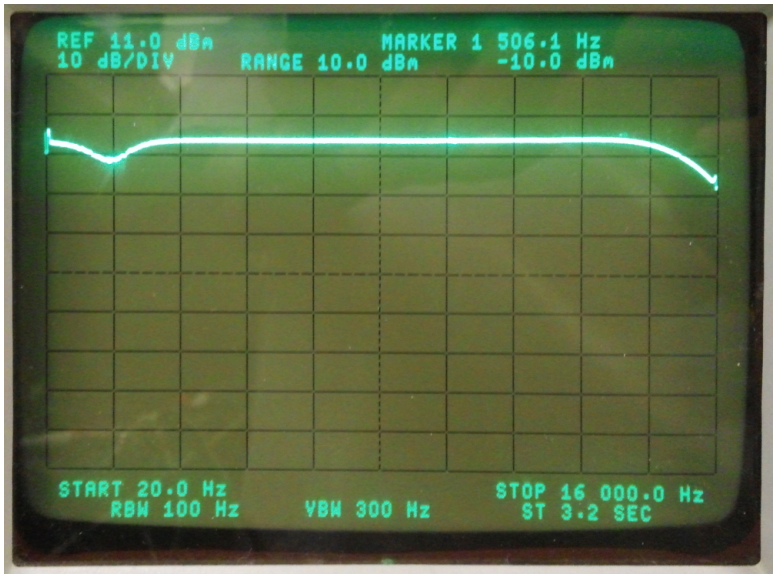

Figure 7: An average value of both center frequency and resonance. A dip vaguely near the center frequency (about 1.5kHz) of about -5dBm.

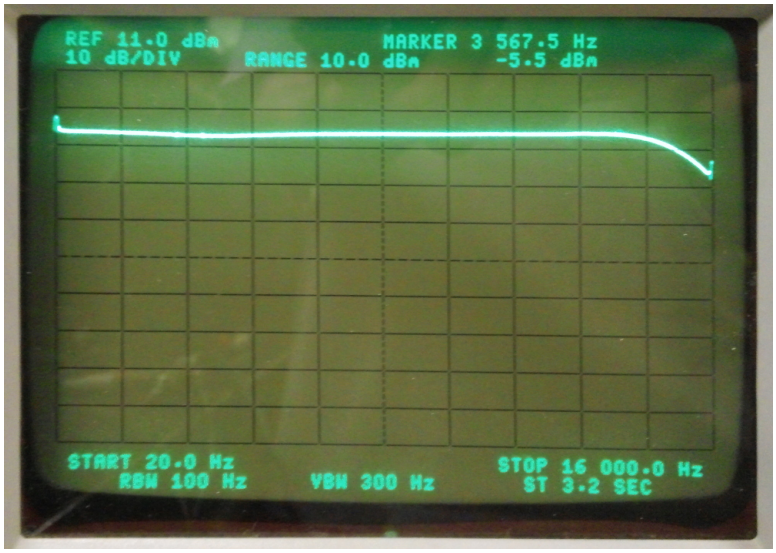

Figure 8: A higher value of center frequency, with the same middle value of resonance. The dip is much smaller, about -1dBm.

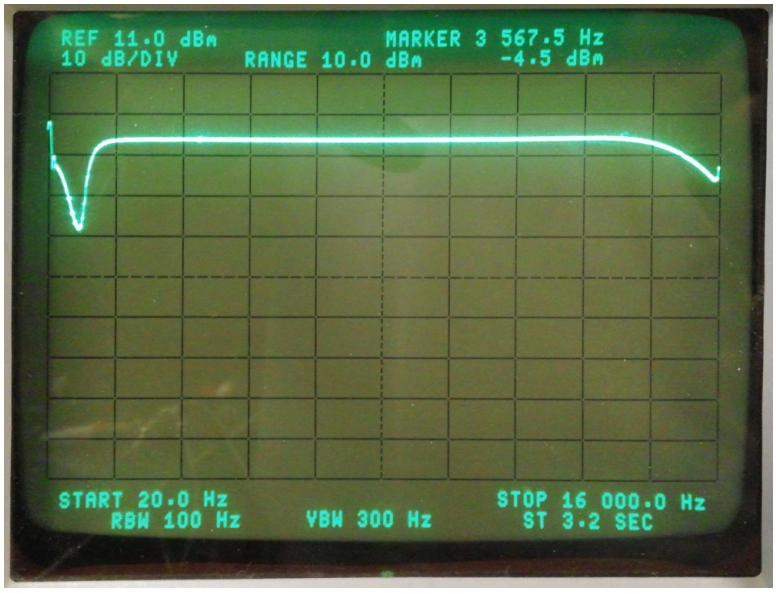

Figure 9: A lower value of center frequency, with the same middle value of resonance. The dip is much larger, about -25dBm.

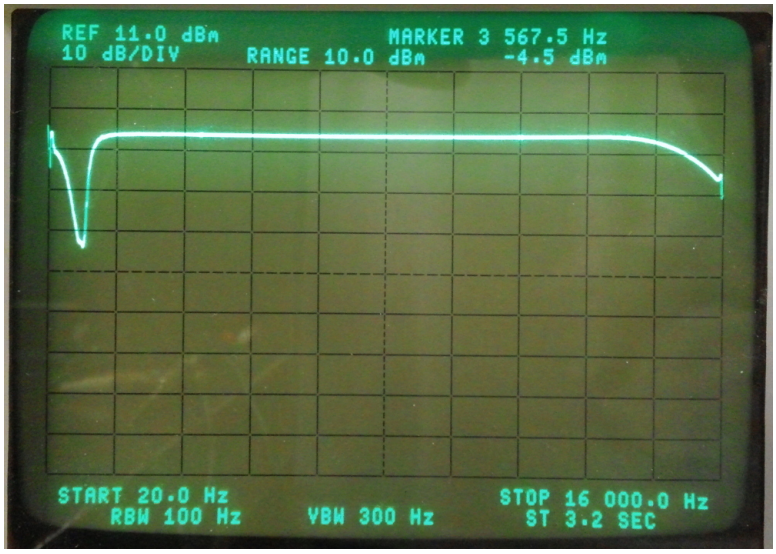

Figure 10: The same value of center frequency, with a higher resonance. The dip is slightly larger, about -30dBm, and the upper edge is noticeably sharper.

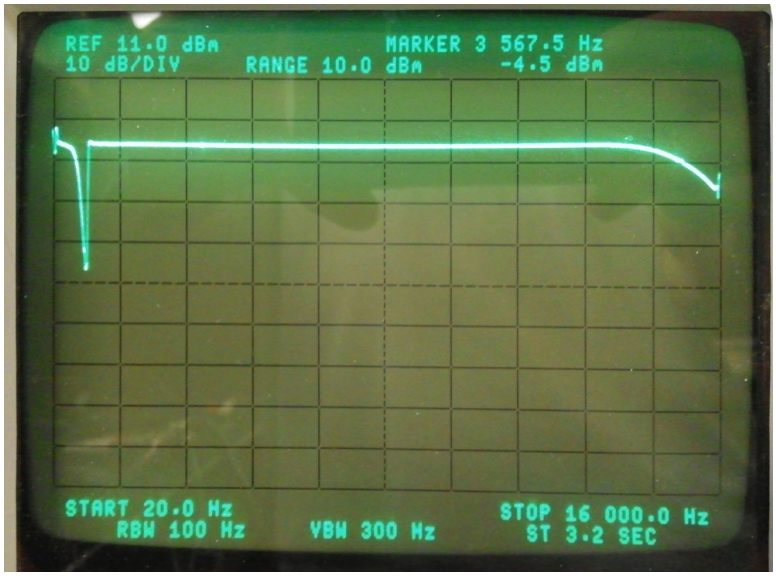

Figure 11: The same value of center frequency, with an even higher resonance. The dip is over -30dBm and very sharp.

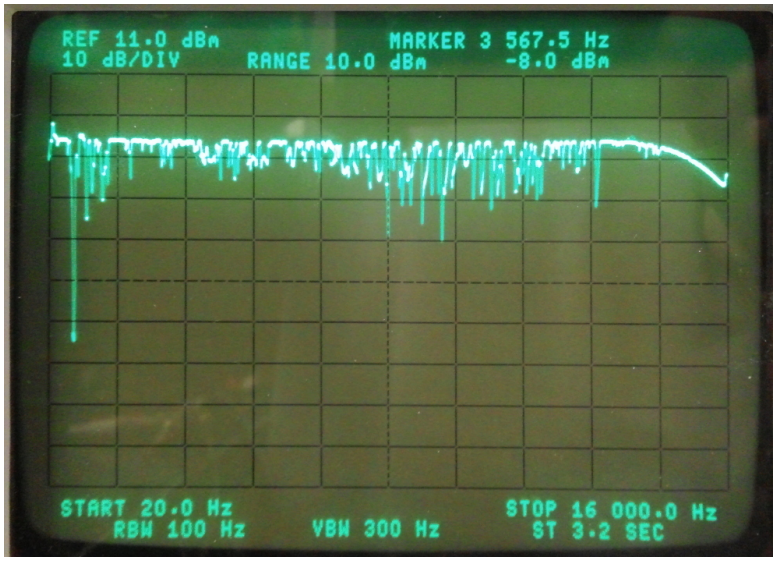

Figure 12: The same value of center frequency, with a bandwidth very close to zero ($Q \to \infty$). Obviously nonlinear behavior.

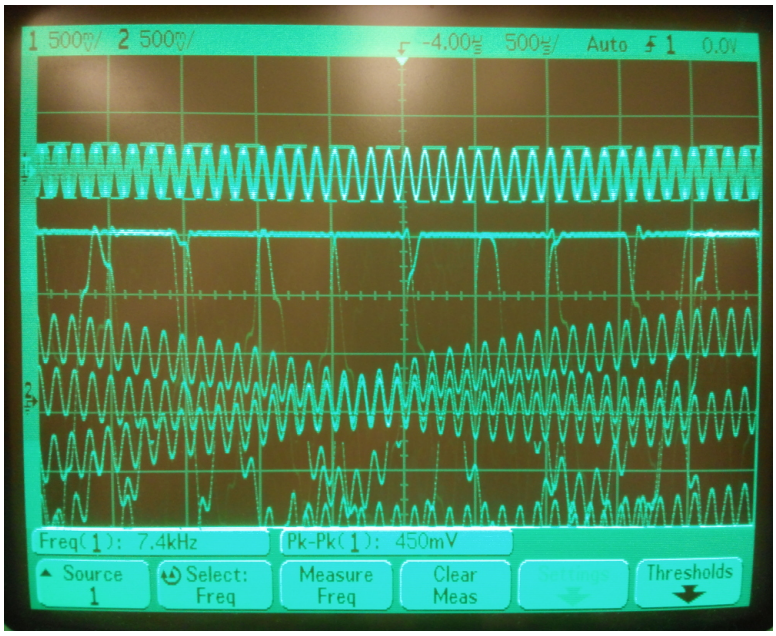

Figure 13: The output waveform whose spectrum is Figure 12. "Oh boy".

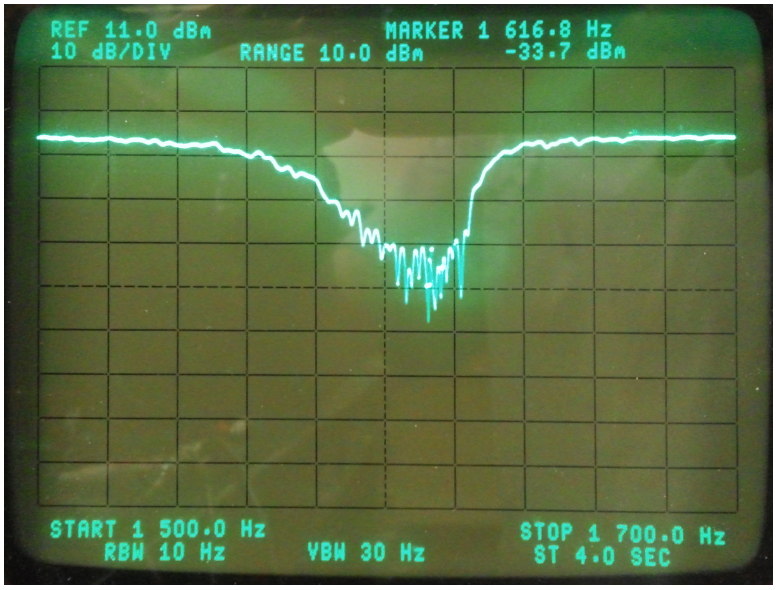

Figure 14: An average value of center frequency, with a high (but not extreme) resonance, zoomed in on the dip in amplitude response.

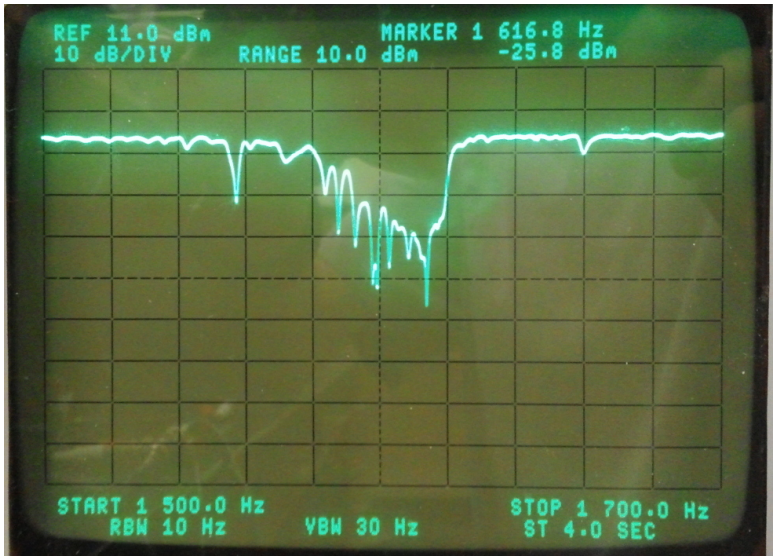

Figure 15: The same, with a resonance bordering on instability.

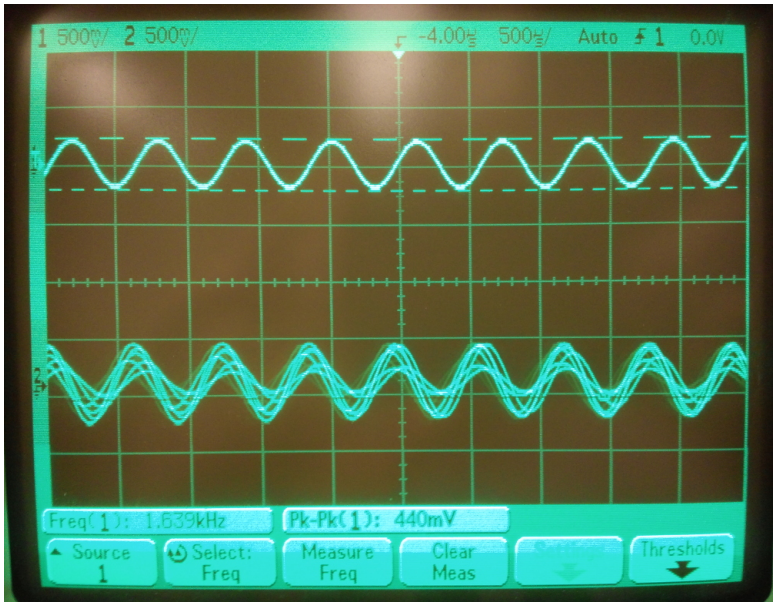

Figure 16: The waveform whose spectrum is shown above in Figure 15.

The severe distortion in these images can easily be accounted for by the fact that this is a digital system going unstable; the dips in amplitude response can be explained by the fact that the system is really just a 64th-order IIR filter (and some designers are scared by eighth-order IIR filters!). However, some additional explanation may also be warranted. The spectrum analyzer operates by generating a sine wave at a certain frequency, sending that to the system, filtering the received signal with a very sharp band-pass filter at the same frequency, and displaying that result. In the band around the center frequency, with a high resonance, the system is designed to generate a delay between the input signal and the output; so it is possible that the spectrum analyzer starts missing the resulting frequencies in that range, as it has moved on to slightly higher frequencies. Since the spectrum analyzer employs a frequency sweep, the input signal is always moving between different frequencies, and that motion constitutes modulation at other frequencies, making considerations of group delay potentially important. Another piece of evidence that these dips might be at least partly artifacts of the measurement is that a -30dB notch filter is quite audible—as at least some change or "weirdness" to an untrained ear, and identifiable as a notch filter at that frequency to a trained ear. Yet such a reduction of frequencies near the cen-

ter frequencies was not audible, even in the plugin version which had four times the length of the filter chain—in the latter version, frequencies near there were delayed in time by an audible amount, but they were not absent.

Further testing could be done on the system by setting it up in a state like this, putting in sine waves at single frequencies one-at-a-time, letting the system settle into steady state for a couple seconds for each, and measuring the amplitudes. Then again, the "frequency response" description of the system is no longer very helpful, as the typical conception of a frequency response is a "snapshot" of the behavior of a system (since it is not time-varying), whereas in this system the long time delays associated with certain *groups* of input sine waves makes it behave like a time-varying system. (This discussion ignores varying system parameters with time, which would make the system actually time-varying.)

4 Plugin implementation

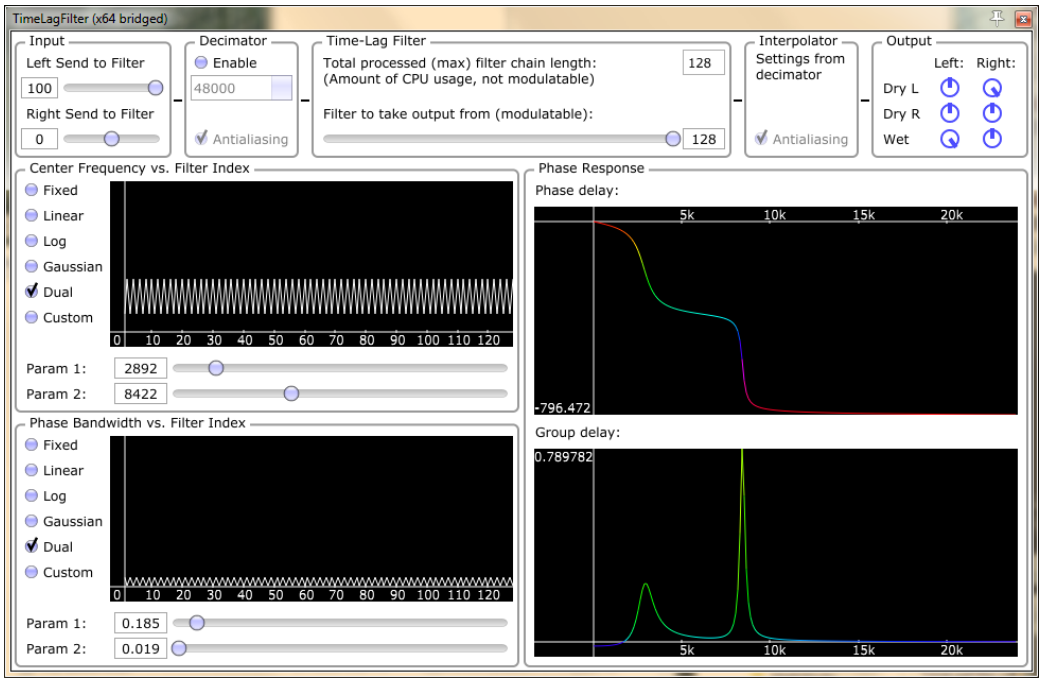

Figure 17: The GUI for the plugin version of the Time-Lag Filter.

4.1 Overview

The Time-Lag Filter is implemented on PC as a VST audio plugin, which can be used in a signal chain in all standard digital audio workstations (DAWs). The plugin is created using the JUCE C++ framework, an industry-standard cross-platform C++ library that focuses on audio software development. The project was developed in Juce's IDE, the Introjucer; compiled for Windows in Visual Studio 2013; and hosted for testing in REAPER[4].

In addition to the features included in the embedded version, the VST version offers:

- Center frequency and bandwidth are not necessarily the same for all the filters in the chain. Graphs of each parameter are shown on the GUI, and the user may select from several presets of how each parameter varies with filter index. Each preset has one or two variable parameters, and a "Custom" preset is available that lets the user adjust each filter's parameters independently.

- Graphs of the phase response and group delay for the entire array of filters are displayed on the GUI. These graphs update in real-time as the filter parameters are varied, and they are based on the theoretical continuous-time behavior of the filter chain, not a measurement of the actual response of the discrete-time system.

- The effective filter chain length is a variable parameter; the actual (maximum) chain length being processed by the system is changeable, but this causes an interruption of the audio and is not intended for real-time control.

- A mixer is provided, to let the user assign variable levels of both input channels to the filter, and either input or the filter's output to each output channel. The mixer levels include the negative half of the range, so that the phase of any assignment can be inverted.

- All variable parameters are modulatable in real-time through the VST framework's control paths. Graphs of the variation of parameters over

[4] A powerful, low-cost digital audio workstation that the author highly recommends. http://reaper.fm/

time can be drawn in the user's DAW, and the filter will respond accordingly. This feature is not yet functional in the current version of the plugin.

4.2 Software architecture

4.2.1 TimeLagFilterAudioProcessor (PluginProcessor)

A VST3 plugin is a Windows DLL that conforms to the Steinberg VST3 API. JUCE wraps this API in the `AudioProcessor` class, and this plugin is a subclass of that, `TimeLagFilterAudioProcessor`. It includes functions for interfacing with the host, responding to parameter changes, and the actual audio sample processing.

Audio is provided to `processBlock()` as a multichannel buffer of a certain number of samples, depending on the audio driver's settings and the tradeoff between CPU use and latency. This class iterates through the buffer, performing the mixing in the function itself but calling the `sample()` function from the filter class to actually perform the filtering. In addition, when playback is started, the class calls a function to setup the filter with the chain length specified by the user and the sample rate specified by the host.

4.2.2 TimeLagFilterAudioProcessorEditor (PluginEditor)

This plugin, like most VST plugins, has a GUI. The `AudioProcessor` class has a function that returns a new GUI component to the host if desired; the type of this object is the JUCE class `AudioProcessorEditor`, and the class `TimeLagFilterAudioProcessorEditor` is this plugin's subclass. It simply creates a window filled with a single `MainComponent`, which is the actual GUI component.

4.2.3 MainComponent

One of the core features of the Introjucer, Juce's IDE, is its ability to create complex GUI components using a drag-and-drop interface. This class, which is the main component visible to the user, was created this way. Much of the code of this class was generated by the Introjucer, with the actual content of interface callback functions filled in by the developer. Other than creating and setting the parameters of the GUI components, this class responds to

user-generated GUI events, checks their validity if applicable, and reports the changes to the `TimeLagFilterAudioProcessor`.

4.2.4 ParamGraph and ResponseGraph

These are custom GUI components; the former is the component that displays the graphs of the filter center and bandwidth values, and the latter displays the phase delay and group delay graphs. Each class contains a flag in its constructor that sets its mode to one of the two graphs it can display; since most of the code is common between the two graphs of each type, it was simpler to write the code this way than to create four different classes. If the functionality had been even more complex, inheritance would have been used, but at this level that would have introduced unnecessary function calls and made the code more confusing.

4.2.5 TimeLagFilterCore

This class performs the filtering. It contains the arrays of filter parameters and signal data, a variety of functions to read and write filter parameters, the `sample()` function that actually processes audio, two threads to perform calculations in parallel, and a locking system that allows for multithreaded access to the filter data.

Multithreading Several threads concurrently access the data in the filter core. GUI events are responded to on the GUI thread, which was created at some point by the window manager; this thread is the one actually changing center and bandwidth parameters. Upon changing these parameters, this thread also calls `recalcCtrs()` or `recalcBWs()` as appropriate, which calculate the center or bandwidth values for all the filters based on the currently-active preset. These functions also notify the second thread, `DTCalcThread`, that new data is available. When it gets a chance, it takes a snapshot of the continuous-time filter parameters by copying them to another array, and then for each filter calculates its discrete-time parameters (b and c). Finally, when it is done, it sets a flag to tell the sample function to swap to this new set of DT parameters.

The thread that actually calls `sample()` is a third thread, the audio thread, which was created by the host. Besides processing the sample through the filter as described in Section 2.2.3, it swaps the arrays of discrete-time

parameters if a new one is available. The final thread is `DlyCalcThread`, which is woken up at 10 Hz by a timer in `ResponseGraph`, and which calculates the phase delay and group delay for all the filters.

Concurrent data access The JUCE framework provides a set of classes that make multithreading relatively easy. This plugin uses two objects of the `ReadWriteLock` class in `TimeLagFilterCore`, one to lock access to the CT parameters and one for the DT parameters. This type of lock can be locked for reading by any number of threads, but only one thread can lock it for writing (and not at the same time as any readers). Local variables of type `ScopedReadLock` and `ScopedWriteLock` referencing the lock objects perform the actual interlocking; the constructors for these scoped locks will block until they get the lock, meaning the code cannot continue until execution is safe. When the function exits and the scoped locks go out of scope, their destructors release the locks.

`TimeLagFilterCore` does not use them exactly as intended. If the arrays were locked so that while one thread is writing to an array, no other thread can read from it, almost all the threads would have to wait for each other in most cases, which destroys the benefits of multithreading (as well as the performance of the plugin). Instead, the read lock is used for functions that read or write to the arrays, and the write lock for operations that modify the entire arrays (changing their size as in `setup()` or copying/swapping the data).

4.3 Testing and results

Summary Due to the lack of a software spectrum analyzer that could be used as an amplitude response finding tool, the same tests performed on the embedded version were not performed on the plugin version. Instead, the effects of much larger filter chains, varying filter parameters over filter index, and changing the input/output mixing to obtain non-flat amplitude responses were observed. A video[5] was created showing the results in detail; only the highlights will be discussed here.

- With a standard filter chain length of 128 filters, four times that of the embedded system, not only were the phase effects observed from that

[5]https://www.youtube.com/watch?v=jg1CMFPCZKk

system (section 3.3.1) observed here, an additional qualitative result appeared. Frequencies close to each other were delayed by different amounts of time, so transients (containing a variety of frequencies) were transformed into quick frequency sweeps of their consistuent frequencies. This was more noticeable with high resonance and at middle (1kHz) frequencies.

- Varying filter parameters over filter index produced results that were slightly different from those with fixed filter parameters. Group delays wider than normal, or with several peak frequencies, were produced.

- Using only a few filters from the chain and mixing some of the unfiltered input into the output produced comb filtering or phasor effects as expected, since some of the frequencies happened to be at phases that would cancel, and others would reinforce.

- After using a version of the plugin that had been compiled with optimization, the system was able to operate with a chain of 2048 filters. This is equivalent to a 4096-order IIR filter! This produced extremely long delays (in the seconds) at some frequencies, a variety of frequency sweeps corresponding to small transients, and in general gave the impression of pulling the audio apart in time. Setting the number of filters to 4096 caused the plugin to behave improperly, take an extremely large amount of CPU (two cores at 4GHz), and stop producing audio. This may have been due to saturation of the filters in the signal arrays, but probably it was simply too much processing for even the high-end test computer to handle.

Stability During development, it was found that values of the center frequency above about $f_s/3$ caused the filter chain to go unstable, as did a value of zero. Thus the center frequency was restricted to the range $[10, f_s/4]$. Similar problems were not observed with the bandwidth, and a bandwidth of zero (infinite resonance) is still permitted in the plugin; restricting the high end to 2 was simply to increase the resolution of the control at the low end.

Performance With the Visual Studio code optimizer enabled, the effect was able to use a very long filter chain, up to 2048 filters, and still only used about half the power of one CPU. However, when playback was stopped,

and the effect's input became near zero, CPU usage went up drastically. It is believed that this is because the floating-point values passing through the filter were so small they were in the denormalized range[6]; and for certain operations, the processor's floating point unit cannot handle these numbers, so it performs the floating poing calculations in software, which takes much longer[7]. A small amount of code was added to set very small ($< 10^{-30}$) values to zero on the filter's input, but this was not very effective, as the real problems are the values in the filter chain itself, and adding code there to check certain values would slow down the processing. The CPU does support a mode which treats denormalized floating-point numbers as zero, but this setting changes for each thread, and since the audio thread was not created by the plugin, it is not clear what the effects of changing processor settings on one of the host's threads would be.

5 Conclusions

The Time-Lag Filter was successfully implemented in both embedded and plugin versions. The plugin version was able to run a much longer filter chain, which made the results more audible and interesting, and hence improved the usability of the effect. The effect is flexible enough that configured in one way, it can add a subtle stereo separation to a sound without significantly distorting it, while configured differently it can tear the sound apart in the time domain—and the transition between these two can be smooth, which would be impossible with other effects that do some similar things (e.g. delay/chorus, phasor, or stereo separator).

However, several features still need to be developed before the plugin can be distributed in a complete form, most notably the resampling engine and correct response to the host's parameter modulation. In addition, the plugin should be tested on other computers with different hosts running at different sample rates, and ideally, other plugin formats that JUCE supports (e.g. AU and VST2, plus 32-bit versions of each) should be compiled and tested. Eventually, the plugin will be released in all of these formats, for free under the GPL license, and hopefully it will find a niche use in professional audio production.

[6] http://stackoverflow.com/questions/9314534/why-does-changing-0-1f-to-0-slow-down-performance-by-10x

[7] http://charm.cs.uiuc.edu/papers/SubnormalOSIHPA06.pdf

6 Appendix

Due to the length and complexity of the source code for both versions, the complete code is distributed separately. Only several key sections appear here.

6.1 Filter algorithm, embedded version

```
/*****************************************************************************
Run one time-step of the time lag filter.
Returns: The system output this time step.
Parameters:
  in: The system input this time step.
  nout: The index of the filter to take the output from; must be between 0 and
    NUM_FILTERS-1.
*****************************************************************************/
float timelagfilter(float in, int nout){
  float out = 0.0f;
  float un = in, yn = 0.0f;
  int filtercount, n, n_1, n_2;
  //Start indices for circular buffer
  n_2 = start_idx;
  if(n_2 >= 3*NUM_FILTERS) n_2 -= 3*NUM_FILTERS;
  n_1 = n_2 + 1;
  if(n_1 >= 3*NUM_FILTERS) n_1 -= 3*NUM_FILTERS;
  n   = n_1 + 1;
  if(n   >= 3*NUM_FILTERS) n   -= 3*NUM_FILTERS;
  //Run all filters
  for(filtercount=0; filtercount<NUM_FILTERS; ++filtercount){
    //LTI system
    u[n] = un;
    yn = (c * (u[n] - y[n_2])) + (b * (u[n_1] - y[n_1])) + u[n_2];
    y[n] = yn;
    //Check for outputting from this block
    if(nout == filtercount) out = yn;
    //Input to next system is output of this one
    un = yn;
    //Move to next block with circular addressing
    n_2 = n   + 1;
    if(n_2 >= 3*NUM_FILTERS) n_2 -= 3*NUM_FILTERS;
    n_1 = n_2 + 1;
    if(n_1 >= 3*NUM_FILTERS) n_1 -= 3*NUM_FILTERS;
    n   = n_1 + 1;
    if(n   >= 3*NUM_FILTERS) n   -= 3*NUM_FILTERS;
  }
  //Start at next buffer position next time
  ++start_idx;
  if(start_idx >= 3*NUM_FILTERS) start_idx -= 3*NUM_FILTERS;
  //Output
  return out;
}
```

6.2 DT filter parameter calculations, embedded version

```
/*****************************************************************************
Sets up the parameters for the time lag filter. Only call when the parameters
have changed, not on every frame please! Very computation-intensive.
Parameters:
  f0: The center frequency of the filter. Range 0 < f0 < fs/2
  Q: The bandwidth of the filter. Range 0 < Q < 1
  fs: The sampling frequency, e.g. 48000.0f
*****************************************************************************/
void timelagfilter_params(float f0, float Q, float fs){
  float fL, fH, pioverfs, OmegaL, OmegaH, Ba, Omega0squared, a;
  //Check input parameters
  if(f0 <= 0.0f || f0 >= fs * 0.5f){
    f0 = fs * 0.25f;
  }
  if(Q <= 0.0f || Q >= 1.0f){
    Q = 0.5f;
  }
  //Calculations
  fL = f0 * (sqrt(Q*Q + 1) - Q);
  fH = Q*f0;
  fH += fH + fL;
  pioverfs = PI / fs;
  OmegaL = tan(pioverfs * fL);
  OmegaH = tan(pioverfs * fH);
  Ba = OmegaH - OmegaL;
  Omega0squared = OmegaH * OmegaL;
  a = 1.0f / (Omega0squared + Ba + 1.0f);
  b = a * (Omega0squared - 1.0f);
  b += b;
  c = a * (Omega0squared - Ba + 1.0f);
}
```

6.3 Phase and group delay calculations, plugin version

```
//Calculate delays
dw = core.getMaxCtr() / ResponseGraph::NUM_RESP_W;
w = 0.0f;
for(nw = 0; nw < ResponseGraph::NUM_RESP_W; ++nw){
    phasesum = 0.0f;
    groupsum = 0.0f;
    wsquared = w * w;
    twow = 2.0f * w;
    for(f=0; f<core.num_filters; ++f){
        center = core.ct_write[f].center;
        bw = core.ct_write[f].bw;
        ctrsquared = center * center;
        a = ctrsquared - wsquared;
        b = twow * bw * center;
        phase = -2.0f * atan2(b, a);
        group = -4.0f * bw * center
                * (ctrsquared - (core.getMaxBW() * wsquared)) / (a*a + b*b);
```

```
            phasesum += phase;
            groupsum += group;
        }
        core.phasedelay[nw] = phasesum;
        core.groupdelay[nw] = groupsum;
        w += dw;
    }
```