ALF: Creating a Lo-fi Effects Plugin Using C++ and JUCE

Christopher Relyea
crelyea@bu.edu

Abstract

This project explores the development of ALF (All-Purpose Lo-fi Facilitator), an audio effects plug-in designed to emulate the aesthetic of lo-fi music through various signal processing techniques. Implemented using the JUCE framework in C++, the plug-in incorporates features such as downsampling, bit depth reduction, low-pass filtering, and the addition of vinyl noise to degrade and transform clean audio signals. The project aims to balance computational efficiency and auditory authenticity, introducing a novel randomization-based downsampling algorithm. This approach enhances performance while maintaining the "rough" sonic qualities characteristic of lo-fi aesthetics. Additionally, the plug-in allows extensive user control over parameters, enabling customized audio outputs. Future developments may include advanced modulation options, tape saturation emulation, and a preset system for greater usability. The project showcases the intersection of digital signal processing principles and artistic creativity in modern audio production.

1. Introduction

1.1 What is Lo-Fi?

The rise of the 'lo-fi' (short for 'low-fidelity') musical style, defined by a poor sound quality (intentional or not) and, more so today, a nostalgia for physical formats past their moment of marketplace dominance, has had a profound impact on the world of music both popular and esoteric since the emergence of the genre in the late-twentieth century. Modern musicologists trace the beginnings of the lo-fi craze to The Beach Boy's twelfth album *Smiley Smile* with its purposeful distortion of cassette sounds or inclusion of "tape hiss" noise in the final mix [1]. The music-consuming public would, over the following decades, become increasingly fascinated with sounds that were purposely not up to the highest of quality standards. In conjunction with the explosive re-emergence of LP record popularity, music released digitally today and most often listened to on a smartphone but made to sound like it was intended for older formats finds continued critical and popular success.

Figure 1: "Lofi Girl"

Paul McCartney's eponymous 1970 solo debut, entirely recorded in his own home, is another notable example which asserts itself as an early manifestation of the soon-to-be popular style of "do-it-yourself" music, now closely associated with lo-fi. Experimental American singer and songwriter Beck's smash hit "Loser" from 1994's *Mellow Gold*, also recorded in the musician's

home, as well as similar songs and artists made the low-quality, “dirty” sound increasingly popular [2]. In more recent years, some of the most acclaimed “indie” musicians claim to have recorded their most streamed tracks with only the low-budget equipment they had available before their respective rises to fame. Music recorded and produced in the absence of professional studio equipment is marked by the resulting sonic imperfections that audiences have come to find intimate and alluring. There is also the rising Internet popularity of the term. Lofi Girl, a popular livestreaming content creator, broadcasts uninterrupted streams of modern lo-fi tracks and has totaled over 2.1 billion views in its channel lifetime of just over nine years. The animated character featured in the unchanging looping visual which accompanies every stream (Figure 1) is now instantly recognizable to a generation of web-surfers.

1.2 Project

I’ve aimed to identify the digital signal processing techniques and psychoacoustical underpinnings that can be leveraged to create an audio plug-in to process clean audio for degradation and modification to emulate the characteristic lo-fi sound. The desired result is a JUCE-based VST3 and AU plug-in I’ve named ALF (All-Purpose Lo-fi Facilitator) that functions with most commercial digital audio workstations and allows the user the freedom to control several parameters to tweak the output lo-fi sound signal to their liking.

2. Definition of Tasks

2.1 Bitcrushing

“Bitcrushing”, a type of audio distortion, is a catch-all term for several possible manipulations of a source audio signal that result in output perceived as less refined or of a lower quality. Assuming little previous signal processing knowledge, I will describe the mathematical basis for the two most common bitcrushing techniques: downsampling and bit depth reduction. Either or both methods are typically used in commercial bitcrushing software plug-ins or hardware effects pedals.

2.2 Downsampling

Let’s start from the foundations of digital audio themselves. Sound in its natural form exists as a continuous signal. When this signal is recorded and digitized it is periodically sampled resulting in the digital representation of audio as, simply, a list of values (samples) from which a reconstructed digital waveform can be produced (Figure 2, taken from *The Complete Beginner’s Guide to Audio Plugin Development* by Matthijs Hollemans [3]).

The closeness of this digital representation to the continuous audio signal depends on the sample rate we have chosen for capturing the audio signal. Common high rates include 98 kilohertz, usually reserved for professional audio production and mastering, 48 kilohertz, used often for audio signals in the higher end of consumer video (movies, TV shows, etc.) and 44.1 kilohertz, the standard rate for both music streaming and CDs.

Figure 2: The digitization of an audio signal

Downsampling, also known as sample rate decimation, seeks to decrease the sampling rate of an audio signal to a lower value, resulting in a signal further in accuracy from the source and audibly lower in quality. The sound produced from downsampling a 44.1kHz signal just slightly to 40kHz for example, will result in small artifacts of distortion, creating a sound that it "rough around the edges" and thus sonically in line with lo-fi aesthetics.

2.3 Bit Depth Reduction

Continuing the discussion of sampling theory, note that when we plot sampled values of an audio signal, they must be associated with some y-value. This value is a measure of the signal's amplitude at a given instant. In the digital form, this is analogous to the signal's intensity or volume, while in a continuous analog signal the y measure is most closely associated with the displacement of a speaker or microphone's diaphragm in moving back and forth to produce the sound by vibrating the surrounding air.

Audio Bit Depth	Technologies/Formats
8-bit	NES, Game Boy, Commodore 64 SID Chip
12-bit	Akai MPC60
16-bit	Compact Disc (CD), MiniDisc, WAV, AIFF
24-bit	DVD-Audio, Blu-ray Audio
32-bit (float)	Most modern DAWs

Figure 3: Various audio formats/devices and their associated bit depths [4]

For our digital signal use, we need to consider the element of precision. We are first given that the y-values in an audio signal must lie between -1 and 1 (in keeping with the most conventional method of digital audio representation). A computer is not capable of representing an infinite number of distinct points in this range, so we must choose a level of precision which can be understood as a quantity of possible unique values between -1 and 1 that we can choose from in representing the y-values of our audio signal. At any given moment, the y-value of our signal is quantized, or rounded, to the nearest possible value of intensity.

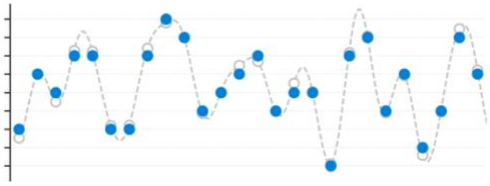

Figure 4: Quantization in a discrete signal

This level of precision is in digital signal practice is known as *bit depth*. For example, 16-bit audio utilizes a quantity of possible intensity values that is equivalent to $2^{16} = 65{,}536$ spread across the range [-1,1].

Quantization error is defined as the difference between the theoretical, continuous values we are trying to plot on the y-axis and the rounded, digital version dependent on the bit depth. Especially at lower bit depths (8-bit and lower), the presence of quantization error can have an impact the audible output of our digital signal by way of unclean sound artifacts, white noise, or the general "computer-y" sound you might expect from and old video game console–an off-shoot genre of the lo-fi music craze known as chiptune, while not a focus of this project, is based on this use of extra-low bitrates. The lowering of target bit depths can be used to emulate older digital formats in their decreased capabilities for storing precise data and is thus fit for a lo-fi plug-in.

2.4 Low-Pass Filters

When downsampling is used in a signal processing pipeline, it is useful to have the option of filtering out high frequencies from the signal. The reason for this comes out of the *Nyquist Theorem* which states that, to digitally represent an analog signal in a way that is faithful to the source signal (meaning that, if we wanted to, we could convert our digital signal back into the continuous waveform exactly how it appeared before digital conversion), we must sample the signal at a rate greater than twice the highest frequency present in the signal. It can be difficult to develop intuition for why this condition must be true for faithful digital representation, but it comes down to the fact that if we try to capture a frequency too high in the eyes of the Nyquist theorem, we will misrepresent it as an incorrect low frequency that is not actually present in the signal to begin with. This effect is known as *aliasing*. Figure 5 shows the result of this phenomenon; a source sine wave of frequency 5Hz is correctly sampled by the red points and waveform at a rate of 15Hz, following the Nyquist theorem, correctly reconstructing the 5Hz signal (red), while the original signal is also incorrectly sampled at the blue points and a rate of 4Hz that is too low. The result of sampling the 5Hz signal at a rate of 4Hz is a 1Hz sine wave, an incorrect representation of the original signal.

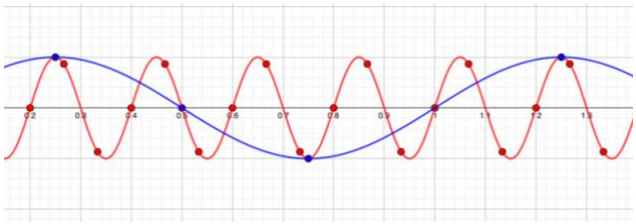

Figure 5: The effects of aliasing

If every signal can be represented as a sum of many frequencies (Fourier series), the job of a low-pass filter is to eliminate all frequencies greater a certain value, set by the user. A low-pass *anti-aliasing filter* can then be applied before downsampling to prevent the representation of new, non-existent frequencies. This is as easy as applying a low pass filter with a cutoff frequency determined by the downsampler's target sampling rate, following the Nyquist Theorem. For example, if we resample a 44.1kHz signal at 22.05kHz, we should first apply a low-pass filter with a cutoff frequency of half the new rate, or 11.025kHZ. Any aliasing that

would occur as a resulting of the downsampling process is eliminated since frequencies above the Nyquist limit will be eliminated before decimation occurs.

2.5 Vinyl Noise

One of the most popular ways for lo-fi artists to pay homage to physical formats of the past is to emulate the extra sounds outside of the music that occur because of a specific medium. One of the most recognizable noise profiles is that of a vinyl record and its player. By adding the low, subtle static of a phonograph player or the pops that occur when its needle encounters dust particles on the grooves of a record, any audio signal can be made to sound like it had been pressed onto a record and played through a turntable.

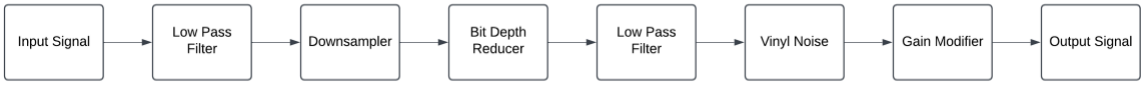

Figure 6: ALF's signal processing pipeline

3. Methods and Implementation

Figure 7: ALF plugin interface

3.1 JUCE

The JUCE framework, originally written in 2004 by developer Julian Storer, is a robust set of tools that allows for the efficient design of audio software. It is one of the most popular tools for the creation of audio effects or virtual instruments plugins mainly for its ease of cross-platform development (Windows, MacOS, Linux, iOS, Android supported) and support for all major audio plugin formats (VST, Audio Unit (AU), AAX, or a standalone application) that allows the developer to write plugins for nearly every DAW on the market. JUCE is relatively easy to jump into if you have some C++ experience and a basic understanding of digital audio, which is why as a novice audio programmer I've chosen to use it for this project.

3.2 Bitcrusher Implementation

I've decided to allow the user the option to apply downsampling, bit depth reduction, or both to the input signal as they see fit. This allows for a customizable pipeline (Figure 6). The downsampling factor and new bit depth as chosen by the user as well as the target sampling rate achieved by decimation (left of the slider in Figure 7) are clearly displayed in the GUI.

3.2.1 ALF Downsampler

In a DAW, an audio signal is processed in *buffers.* A buffer is a small chunk of audio data that the DAW processes and outputs, one chunk at a time, rather than handling the entire audio stream as a continuous flow. In JUCE programming, buffer sizes are set within the DAW then inherited by the plugin. Common sizes are 512 or 256 samples, but this size can be raised or lowered depending on the real-time needs of the audio pipeline or the capabilities of the machine (smaller buffer size requires more refreshing, creating more work for the CPU).

The main loop of a JUCE plugin is represented as the method *processBlock*, which takes as input the current buffer of audio samples and a set of all incoming MIDI events detected during the buffer's timespan (the latter is not important for this project). All of ALF's audio signal manipulation code is implemented in the body of *processBlock* directly or in helper functions which are called there.

It was mentioned earlier that the process of downsampling involves sampling the same analog/continuous signal at a lower rate. But what if we are only given the discrete values as they have already been collected and are not provided with the original signal to resample it? How can we decrease the sample rate of this discrete signal? It depends on our *downsampling factor*. DSP math splits the problem into two cases:

Integer: If we want to decimate a signal by an integer factor n, we can simply use every nth sample as our downsampled signal, discarding the remaining signals.

Non-Integer: If the factor is not an integer, the standard approach is to employ one of various interpolation techniques. For example, let factor $n = 1.5$. If the first sample of our resulting decimated signal occurs at time $t = 0$, the second sample would fall between the second and third samples in the original discrete sample. We do not know what the sampled value of the continuous signal would be at that instant, so we "guess" what the sample would be based on samples we do know by applying the interpolation formula of our choice (linear, spline, Lagrange, etc.) [5].

How can we adapt these two solutions in JUCE? While the framework does include tools for resampling a discrete signal that incorporate interpolation as needed, I've decided to avoid these prewritten methods in favor my own downsampling algorithm for reasons to be explained later in this report.

If the factor (which I also refer to as *block size* for how the new signal looks when plotted on a graph) is an integer, the solution is quite like the general approach as described above. Instead of eliminating samples outside of every *n*th value, though, every value that isn't an *n*th value is set to the preceding *n*th value. No interpolation is required and we also don't have to change the sampling rate within JUCE or delete any samples entirely. Thus, ALF's integer-factor downsampler implementation works as follows:

1) Decide on a downsampling factor *n*
2) Loop through the input buffer and select every *n*th sample
3) After every selected sample, select the next *n*-1 samples with the value of the *n*th sample
4) Repeat until the end of the buffer

When we apply this process to an incoming signal and take a close look at the sample values, it looks like this ($n = 3$):

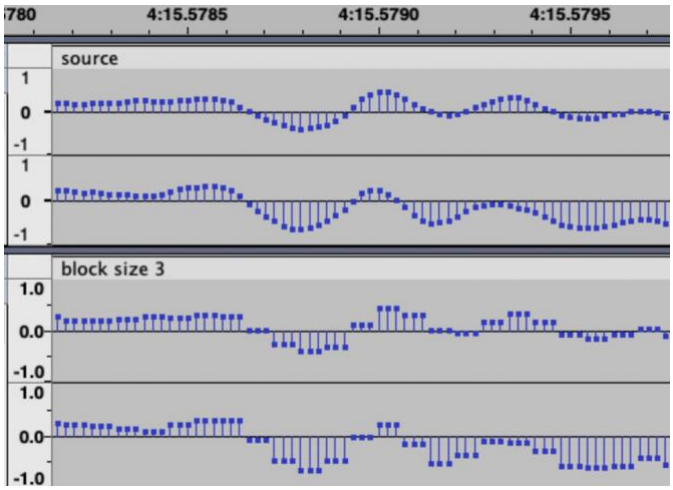

Figure 8: ALF downsampler applied to original signal, block size n = 3

We're now just left with the case of non-integer block sizes. It is at this point where I propose a novel downsampling algorithm that seems to exhibit greater computational efficiency than existing interpolation-based options. This algorithm, implemented in code as shown in Figure 9, works as follows:

1) Decide on a (non-integer) downsampling factor *n*
2) Determine and store n_{lower} and n_{upper} as the floor and ceiling values of *n,* respectively
3) Beginning at $t = 0$, choose between n_{lower} and n_{upper} as the next block size according to a probability function which accounts for the closeness of *n* to its floor and ceiling values:
    a. $P(use\ block\ size\ n_{lower}) = n_{upper} - n$
    b. $P(use\ block\ size\ n_{upper}) = 1 - P(use\ block\ size\ n_{lower}) = n - n_{lower}$
    c. For example, if $n = 1.2$, there is an 80% chance of selecting block size 1 and a 20% chance of selecting block size 2 for the next block
4) Repeat until the end of the buffer

```
// set the two possible block sizes (rounding up and rounding down)
int highBlockSize = intBlockSize + 1;
int lowBlockSize = intBlockSize;
float blockSizeDecimal = blockSize - intBlockSize; // decimal value of the float block size
// loop through all samples in the buffer
for (int i = 0; i < numSamples;) {
    float randomFloat = juce::Random::getSystemRandom().nextFloat();

    /* random float is compared to the decimal value to get the next block size
    * example: a size of 3.1f would compare a random float from 0.0 to 1.0 to the 0.1 decimal.
    * If the random float is higher than 0.1, the the lower block size is chosen. This makes sense because
    * 3.1 is much closer to 3.0 than 4.0, and thus the block size of 3 will be chosen most of the time. */
    int randomizedBlockSize = (randomFloat >= blockSizeDecimal) ? lowBlockSize : highBlockSize;

    int blockEnd = std::min(i + randomizedBlockSize, numSamples); // make sure we don't go out of bounds
    // fill in x values to be the value at the start of the block, where x is the randomly chosen block size
    for (int j = i; j < blockEnd; ++j) {
        channelData[j] = channelData[i];
    }
    // advance the iterator to the end of this block to get ready for the next
    i = blockEnd;
}
```

Figure 9: C++/JUCE implementation of the randomization-based algorithm

As described here, the algorithm seeks to downsample a signal by a non-integer factor by alternating between adjacent integer block sizes with a frequency determined by the decimal block size. Here's how an output signal looks using this method, applied to the same region in Figure 8 but with a non-integer block size of 3.2. Most of the observed block sizes are 3, while block sizes of 4 samples occur around twice in every ten blocks. Block sizes in the modified signal are labeled.

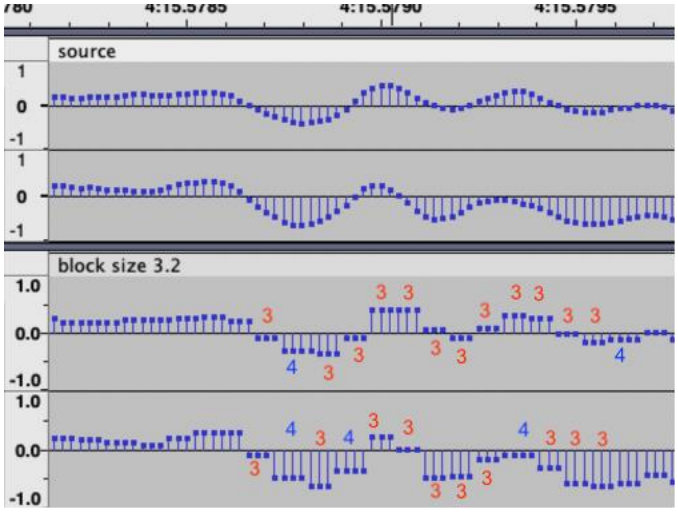

Figure 10: ALF downsampler applied to input signal, block size n = 3.2

Arguments for Randomization Algorithm

We can reason about the time complexity of this randomization-powered algorithm by first assuming that the random step (choosing the next block size) takes $O(1)$ time. Since we must make this decision for each block in the buffer, the number of random processes is $\frac{N}{block\ size}$ per buffer, where N is the total number of processed samples. Combining these steps together with the time to change the values of each sample in a block to be the first sample's value, we arrive at an overall time complexity of $O(\frac{N}{block\ size})$, reducing to simply $O(N)$.

This shows that the randomization algorithm is at least as good as typical methods for non-integer downsampling that would employ the fastest interpolation algorithms (linear or cubic, for example) which also have complexities of $O(N)$.

However, it can be reasoned that the new algorithm reduces the computational overhead required by other interpolation methods due to the simplicity of its calculations. Linear interpolation requires averaging adjacent samples, while the cubic alternative relies on solving polynomials. These operations must be employed for every signal in the output, which can create much more work for the processor especially with higher buffer sizes. Further, linear and cubic interpolation are only the fastest and simplest of the existing algorithms; others which place more of an emphasis on the quality of the reproduced signal are built around mathematical processes that are significantly more complex for a machine to perform. My proposed randomization algorithm requires no such complicated operations, only the randomized determination of the block size and the assignment of all sample values in that block to the value of the first sample. This keeps things quite simple for the machine.

Of course, the trade-off for faster and simpler processing in a real-time audio context is clear when we consider the quality of the final output. Depending on the needs of a system, it may make more sense to sacrifice speed and simplicity in favor of accuracy or fidelity of the reconstructed audio signal, making those algorithms which prioritize performance unusable. Based on the definition and aesthetics of the lo-fi genre, though, high quality audio is likely not the goal for a plug-in such as this, and it may even be antithetical to the signature "rough" sound we're trying to achieve. Thus, a fast algorithm that does not prioritize signal quality is favorable.

Joining Downsampled Buffers

When a buffer is split into blocks according to the downsampling factor, it is possible that this block size does not divide the buffer evenly. As such, it may occur that a buffer of size *n* must end in a block much smaller than the intended block size, or alternatively, that the next buffer must begin with a block that has a starting sample unusually far away from the value at the end of the last buffer. These mismatches can introduce discontinuities at the boundaries between buffers, resulting in audible artifacts such as high-frequency clicks or pops.

To address this issue, I have implemented a smoothing mechanism for the first and last blocks of each buffer. The samples at the end of each buffer are modified so that they gradually approach a value of 0, while the samples at the start of each buffer begin at 0 and slowly grow towards their actual values. This ensures a seamless handoff between buffers and an elimination of the artifacts that may occur during these instants of transitions between buffers. Figure 11 shows the result of this fading strategy on a discrete signal. The source discrete signal (top) remains unmodified while the ALF-processed signal (bottom) allows for a smooth transition into the next buffer.

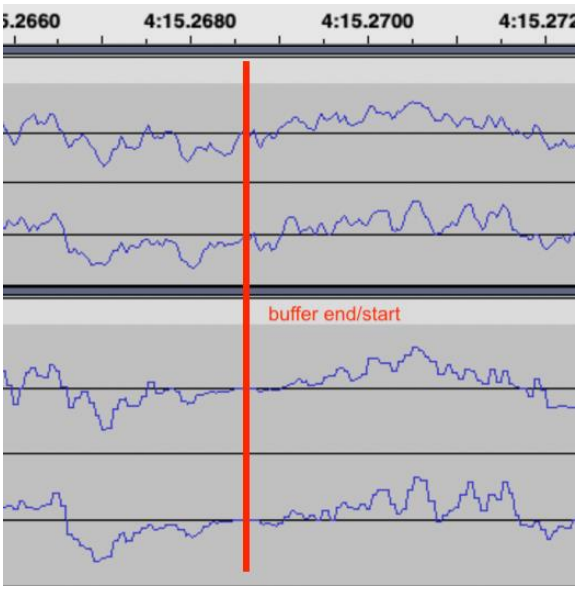

Figure 11: Buffer transition fade out/in

3.2.2 ALF Bit Depth Control

In *Designing Audio Effect Plugins in C++* by Will C. Pirkle, the author defines the following algorithm for bitcrushing using bit depth reduction (Figure 12), which maps *y(n)*, sample values, to values in the new, smaller set of possible values dictated by the new quantization level (*QL*) using *int* to cast a decimal value to its integer floor [6]:

$$QL = \frac{2}{\left(2^N - 1\right)}$$

$$y(n) = QL\left(\text{int}\left(\frac{x(n)}{QL}\right)\right)$$

where

N = new bit depth

Figure 12: Algorithm for bit depth-reduction bitcrushing

This can easily be implemented in C++ JUCE code for the purposes of this plugin (Figure 13).

```
auto* channelData = buffer.getWritePointer (channel);
int numSamples = buffer.getNumSamples();
int possibleVals = pow(2, bitDepthVal);

/* For each value in the buffer, convert it to the value that it would take in a system using the
 new bit depth (fewer distinct values between -1.0 and 1.0)
 1. Multiply original value by the number of possible vals of the new bit depth
 2. Cast to integer (truncation)
 3. Divide by number of possible vals to re-normalize to -1.0 to 1.0 range */

for (int i = 0; i < numSamples; i++) {
    channelData[i] = static_cast<float>(static_cast<int>(channelData[i] * possibleVals)) / possibleVals;
}
```

Figure 13: ALF bit depth reduction implementation

3.2.3 Filters

As mentioned earlier, it is a good idea to apply a simple low pass filter to a signal prior to any downsampling to mitigate the effects of aliasing. JUCE provides an easy-to-use filter tool as part of its DSP library. To incorporate such a filter into our pipeline, it is only a matter of initializing the filter object and supplying the cutoff frequency, easily calculated as half of our target sampling rate (recalling the Nyquist theorem).

I have supplied the reasoning for a pre-downsampling low pass filter, but have also included in ALF the option for the user to apply the same filter type *after* the downsampling and bit depth reduction steps. Where the pre-LP filter, as I've called it, is more necessary in its utility for eliminating unwanted, "fake" aliasing frequencies, the user's choice to enable the post-LP option is more of a matter of preference. In both the downsampling and bit depth-lowering steps of the processing pipeline, there is an opportunity for high-frequency artifacts to emerge in the output signal. These can be heard as high-frequency "clicks," harshness, or a general sense of distortion, particularly in signals with a significant amount of high-frequency content. While these artifacts may not always be intrusive or noticeable, especially in certain styles of music or audio content, they can become problematic in more exposed or critical listening contexts.

The post-LP filter serves to smooth out these potential high-frequency artifacts at the end of the pipeline should the user deem it necessary. This step helps ensure that any residual high-frequency noise or distortion introduced during the downsampling or bit depth reduction stages is minimized, resulting in a cleaner and more polished signal. For simplicity, this filter's cutoff value is set to be the same value as that of the pre-LP filter. ALF gives the user the option to use any combination of these two filters (or neither) as they see fit.

3.2.4 Vinyl Noise

To incorporate in JUCE the addition of vinyl noise on top of the ALF-processed signal as displayed in the pipeline (Figure 6), I first found a suitable royalty-free sound effect several minutes in length which includes light static and the popping of the turntable's needle. It was then only a matter of loading the entire sample as a buffer into the plugin's code and continually looping through this buffer, adding the value of every sample to the current sample being outputted from the processed signal after bitcrushing, bit depth alteration, and filtering (Figure 14). The vinyl noise sample repeats indefinitely using a pointer that jumps back to the start of the buffer after reaching the end (the current position in the vinyl buffer is a global variable accessed within *processBlock*).

```
// add vinyl noise
float noiseModifier = apvts.getRawParameterValue(noiseLevelParamID.getParamID())->load();
if (channel < vinylBuffer.getNumChannels())
{
    auto* vinylData = vinylBuffer.getReadPointer(channel);
    auto* channelData = buffer.getWritePointer (channel);
    for (int i = 0; i < numSamples; i++) {
        if (currentVinylIndex >= vinylBuffer.getNumSamples())
        {
        currentVinylIndex = 0; // Loop the vinyl buffer
        }
        channelData[i] += (vinylData[currentVinylIndex] * noiseModifier);
        currentVinylIndex++;
    }
}
```

Figure 14: Vinyl noise code

3.2.5 Gain Modification

To allow the user control over the gain level of the full, final processed signal after the finishing touch of vinyl noise has been added, I've employed JUCE's built-in *applyGain* method, multiplying every sample in the buffer by a factor dictated by the state of the gain dial.

4. Future Work/Planned Features

4.2 Evaluation

To make more concrete claims about the performance and output quality of the ALF plugin, it would be necessary to run tests in a controlled environment. Particularly, the load on the CPU could be compared between versions of ALF that use the proposed randomized block size approach or other types of interpolation for downsampling to get a better sense of the true merits of the algorithm. This could include a closer look at ALF's memory behavior, the average time taken to process a buffer of audio samples, or a visualization of the output signal in the frequency domain.

4.3 More Options for Modulation

There are many more quirks of the lo-fi sound that could be added as controls in this plugin that would allow much more creativity to the user. Notably, audio recorded through a tape machine can exhibit a phenomenon known "tape saturation" which leads to an iconic warmth in the output signal. Similarly, music captured or played with deteriorated physical equipment might have a "wobbling" pitch variability or increased levels of distortion. Further work can be done to implement such features in ALF that would more closely align the plug-in's capabilities with the classic lo-fi sound of dated physical music recording and playback.

4.4 Enhanced Customization

Commercial plug-ins almost always include a "presets" feature which allow the user to save a configuration they like for quick access in the future. JUCE does provide some built-in support for such elements making this a feasible future addition to the plug-in.

References

[1] A. Harper, *Lo-Fi Aesthetics in Popular Music Discourse*. Wadham College, 2014, pp. 2–3, 44. [Online]. Available: https://ora.ox.ac.uk/objects/uuid:cc84039c-3d30-484e-84b4-8535ba4a54f8/download_file?file_format=application%2Fpdf&safe_filename=AHarper%2B-%2BLo-Fi%2BAesthetics%2BThesis.pdf&type_of_work=Thesis.

[2] A. Price, "'We have the YouTube generation to thank for its explosion': Unpacking the modern lo-fi genre," *MusicRadar*, Oct. 22, 2024. https://www.musicradar.com/tutorials/we-have-the-youtube-generation-to-thank-for-its-explosion-unpacking-the-modern-lo-fi-genre.

[3] S. W. Smith, *Digital Signal Processing : a Practical Guide for Engineers and Scientists*. Saint Louis: Elsevier Science & Technology, 2013.

[4] M. Hollemans, *The Complete Beginner's Guide to Audio Plug-in Development*. Self-Published, 2024.

[5] K. Pohlmann, *Principles of Digital Audio*. McGraw-Hill Companies, 1995.

[6] W. Pirkle, *Designing audio effect plugins in C++ : for AAX, AU, and VST3 with DSP theory*. New York, NY: Routledge, 2019.