NIER: AUTOMATA TONE FILTER IN JUCE

Suomi
Redacted Lab
suomikp31@gnk.com

ABSTRACT

This paper presents the implementation of a tone filter effect inspired by the hacking sequences in the game NieR: Automata, using the JUCE framework. The effect transitions between the original background music (BGM) and an 8-bit version, or synthesizes an 8-bit-like version if none exists. The design utilizes 48 parallel Biquad filters to extract audio features, which are then used to synthesize audio in real-time. The Maister's implementation of the tone filter, employing biquad resonance filters, is referenced for its detailed filter design and SIMD optimization. Additionally, a distortion function is used to create 8-bit-like square waves. This project combines music information retrieval (MIR) knowledge with practical audio programming techniques, demonstrating the effectiveness of JUCE for audio processing and SIMD optimization.

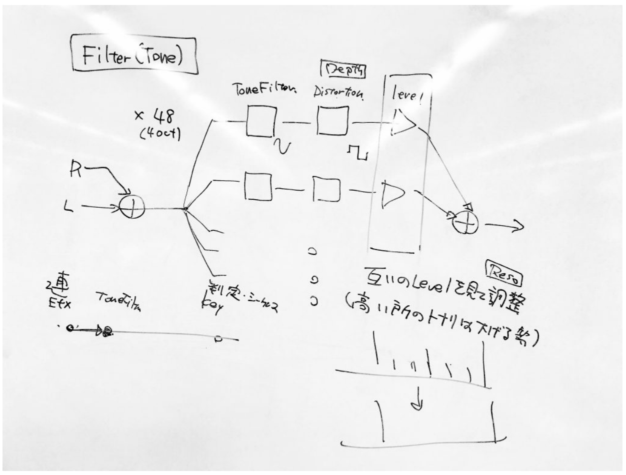

Figure 1. System Diagram from Masami's Blog

1. INTRODUCTION

This project for CSC575 aims to reproduce Platinum Games Studio's hacking audio effect from their game NieR:Automata. Platinum Games use this effect to create transitions between the original BGM track and its 8-bit version, and in case there's not a 8-bit version, it can synthesize a 8-bit-like version on the fly.

This effect utilizes IIR/FIR filters to extract audio feature, these features (in the form of 48 sine-wave-like signals at specific frequency) are then used to synthesize audio in real-time. We choose this effect as it combines the MIR knowledge with good audio programming practice, and the parallel filter design also exposes parallelism, which is perfect for some low-level SIMD coding practices.

2. ORIGINAL PLATINUM STUDIO BLOG

Said effect was acoomplished by using 48 parallel Biquad Filters alongside a distorted Chiptune synthesizer. This created a 8-bit like side track from the current BGM.

In the game NieR: Automata, the effect gradually fades in the tone-filtered version of when the protagonist starts a hacking sequence in the game, and swap to a handcrafted 8-bit version of the BGM when the hacking minigame starts. If there's not a handcrafted version, the game falls back to playing the tone filtered sidetrack mixing with the original track with an $80 : 20$ ratio. Lead Composer Masami Ueda shared their blog post *Happy Hacking: Music implementation in NieR:Automata* [1] regarding a few technical details and filter designs for this effect. Several demo videos can also be found there. Overall, this effect fits quite well for transitioning the game from a Triple-A adventure into the arcade-like hacking minigame.

Masami kindly offered us a white board diagram explaining what is happening in this transition effect. There are 48 4-Oct filters running in parallel, getting a sine wave of specific tone frequency as the output. The sine wave is then passed to a distortion function, generating 8-bit like square wave. Outputs are mixed up again with a stereo delay (not present in the diagram), forming the side track.

While this diagram explains the design of the system, the remainder of the blog has no technical details, so the design of the filters and the distortion function are unknown to us. Without more details on filter designs and the distortion functions, this tone filter will be very tough to do well and probably will take a lot of time for trial-and-error.

3. THE MAISTER'S IMPLEMENTATION

While Platinum Games did not release any details regarding their implementation of the Tone Filter, a very helpful reference solution is provided online by The Maister [2]. In their own C++ framework, Graphite, they implemented this tone filter effect as one of the audio capabilities of the framework.

This version of the effect has more detail about filter design, SIMD implementation and distortion function, and they are tested to work pretty well. This should help us to get straight to the implementation and optimization without spending too much time to tune it right.

3.1 Filter Design

The Maister chooses a biquad resonance filter to act as the tone filter. In the time domain, a biquad filter is defined by the following equation:

$$y[n] = b_0x[n]+b_1x[n-1]+b_2x[n-2]-a_1y[n-1]-a_2y[n-2] \tag{1}$$

Or, in the Z-domain,

$$H(z) = \frac{b_0 + b_1z^{-1} + b_2z^{-2}}{1 + a_1z^{-1} + a_2z^{-2}} \tag{2}$$

Here, we aim to make use of this filter to generate an ultra-sharp response, and thus we design the zeros and poles of the filter accordingly. Following The Maister's notation, we will now refer to the feed forward part of the filter $b_0x[n] + b_1x[n-1] + b_2x[n-2]$ as the FIR response and the feed back part $-a_1y[n-1] - a_2y[n-2]$ as the IIR response.

3.2 Pole / Zero Placement

The biquad filters' effect is entirely determined by the pole/zero placements, which dictates the frequency where we have infinite response and zero response respectively. [3]. The IIR response is determined by the poles, and their placement is pretty straightforward, we want a specific tone to be filtered in, so we place them at

$$w = \pm 2\pi\frac{f}{f_s} \tag{3}$$

where f is the desired frequency and f_s is the sample rate. The amplitude of the pole was chosen to be 0.9999, which ensures stability while provides an ultra-sharp response (or high Q-factor). We know that the poles in biquad filters are always complex conjugate pairs, so a set of a_i values can be pretty easily calculated for the IIR response.

$$a_1 = -2r\cos(w) \tag{4}$$

$$a_2 = r^2 \tag{5}$$

where r is the amplitude of the poles.

Note that these a_is need to be negated in the Z-domain. (As they are the denominator)

The zero placement determines the FIR response, and we need to adjust b_i accordingly. The Maister chooses to place one zero at DC $(w = 0)$ and another at the Nyquist Frequency $w = \pi, f = f_s/2$. On the Z plane, this puts two zeros at $(0, 0)$ and $(1, 0)$. We visualize the response using Matlab in Figure 2. We used a modified version of [4] which is fixed to be usable in Matlab R2024b.

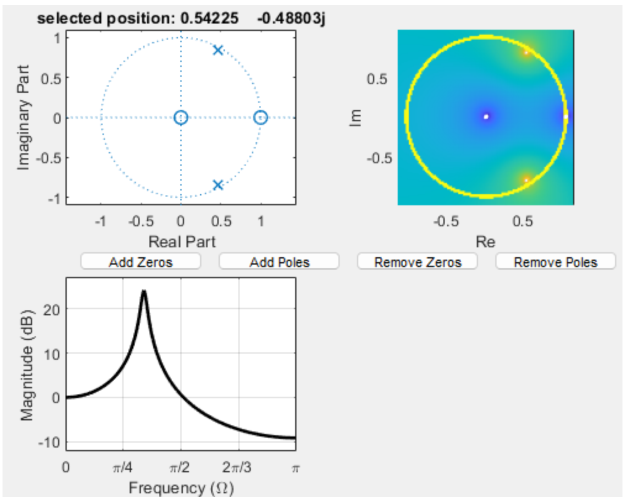

Figure 2. A sample poles/zeros placement

3.3 Normalization

Lastly, we don't want to amplify the target frequency. You can see from the response figure that we amplified the response on specific frequency by quite a lot, but in reality we just want that frequency preserved. Normalizing the response will remove noises and get us our desired response at the same time.

The Maister designed a evaluation function to ease the feed forward part of the response, lowering the gain to approximately 0 dB. We will pull this function from their VST plugin version of the effect [5], as it's more barebones and readable than the graphite version. Let's take a look at their code.

```cpp
std::complex<double> PoleZeroFilterDesigner::
    evaluate_response(double phase) const
{
  std::complex<double> num = 0.0f;
  std::complex<double> den = 0.0f;

  for (unsigned i = 0; i < numerator_count; i++)
    num += numerator[i] * rotor(-phase * double(
    i));
  for (unsigned i = 0; i < denominator_count; i
    ++)
    den += denominator[i] * rotor(-phase *
    double(i));
  return num / den;
}
```

This normalization process is surprisingly simple, see above code. We are essentially passing in a signal of the exact frequency we want to filter in, and see how much gain we get from the filter. Then, this returned number $A = \frac{num}{den}$ is used to normalize the response by:

$$b_i = \frac{1}{A} \cdot b_{i,peak} \tag{6}$$

This way, the peak filter is turned into a tone filter. We should be able to get a single sine wave, or at least a signal that composes of a few sine waves around the desired frequency, from a song input. There are more about how we arrange those 48 filters to different tones, which we will add to this report later.

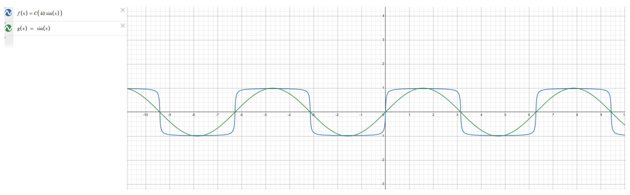

Figure 3. Distortion Function

3.4 Distortion Function

The last thing we need to do is to synthesize the square wave that resembles a 8-bit console. Since we already have a sine signal, distorting it to a square wave should do the trick. We don't really need to synthesize anything from noise as traditional synthesizers do.

The Maister choose to use the HDR tone mapping formula after doing a lot of research.

$$C(x) = \frac{x}{1+|x|} \tag{7}$$

$$f(x) = C(40 \cdot \sin(x)) \tag{8}$$

Visualize this on desmos gives Figure 3.

This function is cheap and SIMD-friendly, so we'll use it in our implementation as well.

4. JUCE FRAMEWORK

We choose to use JUCE framework for this demo. JUCE is a specialized audio programming framework, providing basic UI library, signal processing and capability of compiling to VST3 or native plugins.

Some previously made VST3 plugins will be used as supplementary tools for this project. At this moment, we will need a plugin to deliver the audio source. This functionality will be supported by my AudioFilePlayer VST3 plugin [6].

JUCE provides some basic implementations of filters. We'll explore the options to use existing code bases first, and if that ended up not working, we will build the biquad sharp filter from scratch.

JUCE also supports SIMD buffers, which should make SIMD optimization easier. A SIMD tutorial for JUCE is available at [7]. We are going to do quite literally this tutorial did - parallelize the filters.

5. SIMD OPTIMIZATION

Since the filters are largely doing the same thing to the same input at the same time, SIMD is a good way to accelerate this process. I originally thought about doing this on the GPU, but that was largely not feasible as the 44100 host-to-device transfer we need to do per second is not worth it.

Doing SIMD in this project should be fairly straightforward. JUCE provides a SIMDRegister class that should be a bit easier to use than The Maister's code, mainly because that it wraps different instructions under a layer of abstraction, so we don't have to handle different CPU types in our codes.

6. TEAM MEMBERS AND WORK DISTRIBUTION

Removed unimportant information.

7. DELIVERABLES AND TIMELINE

The project can be divided into the following sub-tasks. As this semester is my last semester, I'm not sure how much time commitment I will have for this project. Therefore, I listed my objectives modularly and the deliverables will be tagged as two groups, base and optional (stretch goals).

Base Deliverables:

1. Parallel filter design
2. VST3 with GUI
3. Chiptune Sine wave distortion

Optional Deliverables:

1. SIMD
2. Unity native plugin with demo scene

The project will last for 6 weeks, from Feb. 24 to Apr. 11. The deliverables will be updated bi-weekly.

Ideally, Base Deliverable 1,2 will be done in early March, Base Deliverable 3 will be done in mid-March. In late March, I'll have a (very inefficient) demo ready and usable. We will leave the remaining time to optional deliverables, with SIMD being the first priority.

8. PROGRESS REPORT

The bulk of the report is completed on Feb. 18, 2025. The mid-term update is completed on Mar.14.

8.1 Mar. 4 First Update

In the first half of the project, we successfully laid basis for our tone filter design in JUCE by exploring the codebase of JUCE's IIR filter design.

By reading JUCE's IIR Filter implementation [8], we can find we have a ready-to-use IIR filter at hand. It is compatible to 1-3 orders. In our case, we need the second order.

The implementation looks like this:

```
switch(order)
    case 1:
    ...
    case 2:
    {
        auto b0 = coeffs[0];
        auto b1 = coeffs[1];
        auto b2 = coeffs[2];
        auto a1 = coeffs[3];
        auto a2 = coeffs[4];

        auto lv1 = state[0];
        auto lv2 = state[1];

        for (size_t i = 0; i < numSamples; ++i)
        {
            auto input = src[i];
            auto output = (input * b0) + lv1;
```

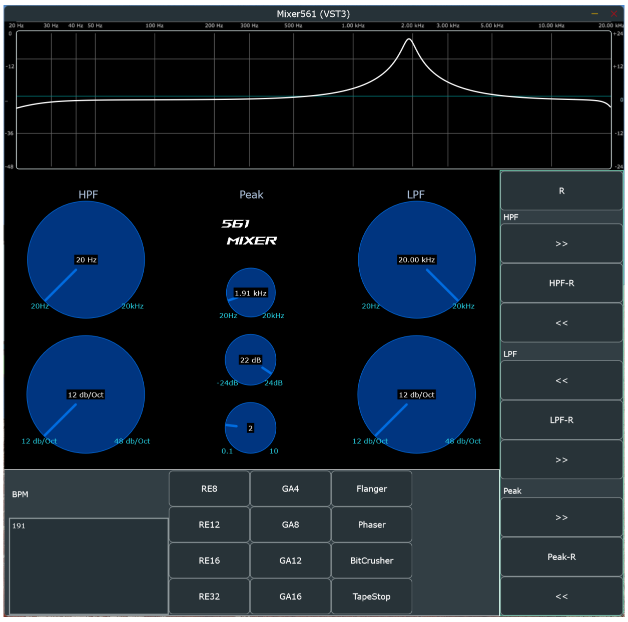

Figure 4. 561 Mixer

```
            dst[i] = bypassed ? input : output;

            lv1 = (input * b1) - (output* a1) +
    lv2;
            lv2 = (input * b2) - (output* a2);
        }

        util::snapToZero (lv1); state[0] = lv1;
        util::snapToZero (lv2); state[1] = lv2;
    }
    break;
    case 3:
    ...
```

We can clearly outline that we can manipulate these coefficients to get desired filter behaviors, as described above. Particularly, we need to implement The Maister's filter designer in JUCE to produce these coefficients and use them to design custom filters. The b_i here corresponds to the filter's FIR portion, while the a_i corresponds to the IIR portion.

In one of my previous projects [9], I extensively used JUCE's IIR filter, but the parameters are all generated from built-in static functions, as that project mostly uses conventional filters (LPF, HPF, Peak). I also implemented a lot of real-time effects there, but the most related (to this project) part is the peak filter.

What we are going to create is essentially a special peak filter array, which is normalized to unit gain at peak (desired frequency). Taking a peek at the peak filter coefficient generator:

```
template <typename NumericType>
std::array<NumericType, 6> ArrayCoefficients<
    NumericType>::makePeakFilter (double
    sampleRate,

                                  NumericType
    frequency,

                                  NumericType Q,

                                  NumericType
    gainFactor)
{
    jassert (sampleRate > 0);
    jassert (frequency > 0 && frequency <=
    static_cast<NumericType> (sampleRate * 0.5))
    ;
    jassert (Q > 0);
    jassert (gainFactor > 0);

    const auto A = std::sqrt (Decibels::
    gainWithLowerBound (gainFactor, (NumericType
    ) minimumDecibels));
    const auto omega = (2 * MathConstants<
    NumericType>::pi * jmax (frequency,
    static_cast<NumericType> (2.0))) /
    static_cast<NumericType> (sampleRate);
    const auto alpha = std::sin (omega) / (Q *
    2);
    const auto c2 = -2 * std::cos (omega);
    const auto alphaTimesA = alpha * A;
    const auto alphaOverA = alpha / A;

    return { { 1 + alphaTimesA, c2, 1 -
    alphaTimesA, 1 + alphaOverA, c2, 1 -
    alphaOverA } };
}
```

It returns 6 numbers corresponding to the second order case we listed above. We can write its tranfer function as:

$$H(z) = \frac{1 + \alpha \cdot A + z^{-1}(c_2) + z^{-2}(1 - \alpha \cdot A)}{1 + \alpha/A + z^{-1}(c_2) + z^{-2}(1 - \alpha/A)} \quad (9)$$

In similar fashion, we need to implement a `makeToneFilter` function, which puts one zero at DC ($z = 1$), another at Nyquist($z = -1$).

For the IIR portion, this means we need to satisfy:

$$b_0 + b_1 z^{-1} + b_2 z^{-2} = 0 \quad (10)$$

$$z = \pm 1 \quad (11)$$

This will give us:

$$b_0 = c \quad (12)$$

$$b_1 = 0 \quad (13)$$

$$b_2 = -c \quad (14)$$

As for the FIR portion, as discussed in Section 3.2, they will be placed at:

$$a_1 = -2r\cos(\omega) \quad (15)$$

$$a_2 = r^2 \quad (16)$$

where $r = 0.9999$.

We also need to evaluate the response at ω to ensure a unit gain. This process is essentially done by setting c to $1/A_f$ where A_f is the magnitude of the output when we evaluate the filter at the desired frequency.

That concludes the tone filter's mathematical design in JUCE. We then got a stripped down version of Mixer561 and created a new repo. We kept the 3 filter knobs there but modified the peak part to be our test for the tone filter.

8.2 Mar. 14 Codebase Update

This update will be fairly simple, we started to create a code base and some basic filter parameters. We branched from a very early version of Mixer 561, when there is only

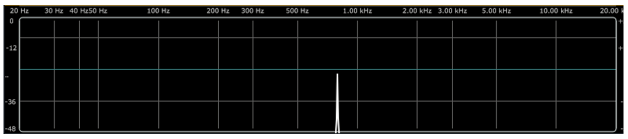

Figure 5. Tone Filter Response Curve (at 800Hz)

Peak and HPF/LPF implemented in the code base. Conveniently, I already have a response plot implemented there, thanks past me!

The first step is to implement an ultra sharp peak response. We started from a built-in ultra sharp response, where we set the Q factor to be 10 and gain to 24dB. This approach (as outlined in the last update) creates an audible ultra sharp response already. I used Hirayasu Matsudo's remixed song *Genesis At Oasis (Hirayasu Matsudo Remix)* to qualitatively test the response as it has bass on F5 (698Hz) playing throughout the song.

Next up, there're two more things to do: Customized filter coefficients and gain normalization.

Let's first do the easy one. Gain normalization should be as simple as substituting $z = e^{j\omega_0}$ to the transfer function and get a response. We then scale all the IIR parameters down by the (magnitude of) the response to normalize the gain. This turns the ultra-sharp peak filter into a tone filter, which silents all tones but the desired one.

This, turns out to be extremely easy to do. JUCE's IIR filter class implemented a function `getMagnitudeForFrequency`, which returns the magnitude $\frac{num}{dem}$ directly to us. All we have to do is scale b_i down with this factor (See formula 6), and we get a unit gain at the peak frequency and silenced all the other frequencies. The filter design is also fairly simple, we basically directly use the parameters calculated above with $c = 1$ and $a_0 = 1$ in JUCE's second order IIR filter, then normalized the c to produce a unit gain.

The response curve is shown in Figure 5.

Now, since we know how to construct the tone filters, it's time to roll the filter factory out. Following The Maister's practice, we constructed 48 tone filters in the project, spanning out 4 Octaves with 12 semitones. `440.f * std::exp2(float(i - 12) / 12.0f` will cover tones from A3 to A7.

Lastly, to test if those tone filters work properly, we started to do some basic modifications to our process loop. We introduced 48 tone buffers and two additionally buffers. The `TempTrackBuffer` will contain the mono version of the current audio frame, we fill it by directly averaging the two channels in the input. The `SideTrackBuffer` is used to store the tone-filtered result. We currently directly add each tone-filter output to it with a unit gain, so the result will be a quantized version of the original song. It worked as expected and generated a borderline piano-like sidetrack that retains the critical musical information of the input.

There are still a good amount of things to do but I think we are at a good position for a mid-term report. In the next update I'll implement the distortion function, which should make the sidetrack sounds chiptune-like. The Maister also did a few other tricks to balance the sidetrack, namely he calculates the energy contained in each track and discard the non-important ones to emulate the limited capacity of the old consoles.

8.3 Mar.28 Implementation - Leveling

One thing to keep in mind at this point is that the system will sound terrible and not 8-bit at all due to energy distributions in the audio. Typically in older consoles, there are limited tones available (it's why they are called 8-bit, because you only have 8 bits for a digital audio signal!), but the source audio we have here are much more complicated than that and tones may leak to each other.

Achieving good leveling can take time, but fortunately we have a tried and true way to deal with this issue. What we want to do here is a **running power estimation**, essentially estimating how much energy is in the current audio buffer that we are listening to. That power is critical for us to then determine the best tone to play among all the tones we are getting, as we can roughly estimate how "important" it is.

We set two fixed power threshold, with the "floor" being $2 \cdot 10^{-4}$ times the RTP, "ceiling" being 0.1 times the RTP. These values will later be changed to variables when I have time to explore more tracks other than the few from NieR: Automata that I used to test the filters.

The RTP is calculated by, in the time domain:

$$P_{rt}^{t} = P_{rt}^{t-1} * (1 - W_{tone}) + W_{tone} * s_t^2 \qquad (17)$$

, where s_t is the current sample in the main mono track, and W_{tone} is the weight of which you want the tone filter output to be. It is adjusted to the sample rate to compensate for more samples processed under high audio quality.

This is pretty straight forward, it ensures a smooth transition in total power level from the filters and generally keeps it proportionate to the total power from the main track, preventing it from sounding too abrupt.

We similarly do a running attenuation within the 48 tone outputs so that it drops smoothly when tone energy distribution changes. We aim to create a lasting harmonic edge that lingers a little bit longer than the original tone, which turns out to work pretty well.

8.4 Mar.31 Implementation - Distortion

We will stick to the distortion function provided by reference, listed in section 3.4. The implementation is pretty straight forward, the output from running leveling will be passed to a distortion function with its RMS value calculated beforehand, as the function is expected to output a unit square wave, we then multiply the RMS back to maintain the tone's amplitude.

The code is done at this point, albeit still very naively written with no-SIMD involved. It however performs surprisingly well, as JUCE does implemented a very compiler-friendly audio buffer structure that can be optimized to application-ready level even when written naively. Still, we plan to explore the SIMD API it offers.

CPU	Naive	IIR SIMD	Distort SIMD
i9-14900HX @5.7GHz	6.9 M/s	13.2 M/s	13.6M/s

Table 1. SIMD Performance

8.5 Apr. 7 SIMD Optimization - Maybe it's not really needed...

JUCE provided a very good abstraction, the `SIMDRegister`. This class removes the need to write SIMD calls for multiple instuction sets, instead, you create multiple write pointers to the data (in our case, 48 tone buffer outputs) and they can be operated in SIMD-fashion with IIR filters. The register is designed to support all contemporary instruction sets on a higher level.

What we do is very similar to the tutorial [7]. Instead of running 48 IIR filters, we create one IIR filters and use 48 sets of parameters which are predetermined and output to 48 write pointers.

We benchmark the system by manually run the `processBlock` function repeatedly on a 256-sample audio block and discard the output. We measure the samples the system can process each second, with max load possible.

The result is an approximately 80% performance gain, with distortion also SIMD-ed we can squeeze some more. Since we only need 44k samples processed per second, SIMD isn't really needed (even if you are using some lower-end chips), JUCE itself is optimized enough that you can write a few hundreds of filters in a for loop.

9. CONCLUSION

We successfully recreated the Maister's tone filter and it sounds reasonably similar to the one in the game! I run out of time to explore more stuff as this is my last term at UVic. I'll try to make more adjustable parameters and fiddle around with it should time allows in the future.

My great gratitude goes to the MIR course crew. I wish the Synthesizer Programming course were still offered!

If you want to learn JUCE, I have filters, effects and synthesizer example projects, available on my GitHub page https://github.com/SuomiKP31. This project will also be open-sourced soon (when I have time to clean it up).

10. REFERENCES

[1] M. Ueda, "Happy Hacking: Music implementation in NieR:Automata | PlatinumGames Official Blog — platinumgames.com," https://www.platinumgames.com/official-blog/article/9581, 31-07-2017, [Accessed 17-02-2025].

[2] T. Maister, "Recreating the tone filter from NieR:Automata 2013; Maister's Graphics Adventures — themaister.net," https://themaister.net/blog/2019/02/23/recreating-the-tone-filter-from-nierautomata/, [Accessed 17-02-2025].

[3] P. Cheung, "Poles, zeros & filters," n.d., accessed: 2025-02-18. [Online]. Available: http://www.ee.ic.ac.uk/pcheung/teaching/ee2_signals/Lecture%209%20-%20Poles%20Zeros%20&%20Filters.pdf

[4] Dadorran, "Zpgui," https://dadorran.wordpress.com/2012/04/07/zpgui/, [Accessed 18-02-2025].

[5] T. Maister, "GitHub - Themaister/ToneFilterVST: Basic recreation of the Nier Automata tone filter in VST form — github.com," https://github.com/Themaister/ToneFilterVST/tree/main, [Accessed 18-02-2025].

[6] [Redacted], "GitHub - SuomiKP31/AudioFilePlayerPlugin: Basic cross-platform/cross-API audio plugin for playing audio files. — github.com," https://github.com/SuomiKP31/AudioFilePlayerPlugin/tree/master, [Accessed 18-02-2025].

[7] J. Developers, "JUCE: Tutorial: Optimisation using the SIMDRegister class — docs.juce.com," https://docs.juce.com/master/tutorial_simd_register_optimisation.html, [Accessed 18-02-2025].

[8] JUCE, "Juce iir implementation." [Online]. Available: https://github.com/juce-framework/JUCE/blob/51a8a6d7aeae7326956d747737ccf1575e61e209/modules/juce_dsp/processors/juce_IIRFilter_Impl.h

[9] [Redacted], "Suomikp31/juce_vst3_mixer: A juce-made vst plugin." [Online]. Available: https://github.com/SuomiKP31/JUCE_VST3_Mixer