Vocal Distortion in Time-Domain using Amplitude Modulation

Maria Polak

Aalborg University Copenhagen, Sound and Music Computing, 7th Semester

1 Introduction

This project is a real-time implementation of a vocal distortion and roughness effect, described in detail in a paper by Gentilucci, Marta, Luc Ardaillon, and Marco Liuni (1). The effect was implemented as an audio effect plugin using the JUCE framework (2).

2 Algorithm

2.1 Voice distortion

2.1.1 Amplitude Modulation

In the original paper (1) authors explain that it is possible to add roughness to monotonic voices by using amplitude modulation, in general, described by equation

$$y(t) = x_m(t) * x_c(t) \tag{1}$$

where x_c and x_m denote carrier and modulating signals respectively.
Authors conclude that by using modulating signal of the following form

$$x_m(w_0, t) = 1 + h\cos(w_0 t) \tag{2}$$

where $w_0 = 2\pi f_0$ and f_0 being fundamental frequency of the carrier signal x_c, eq.(1) yields a result of the original signal enriched by its sub-harmonics

Later, they extend the definition, by saying that it is possible to use Eq 3. as the modulating signal, in order to generate more sub-harmonics.

$$x_m(w_0, t) = 1 + \sum_{k=1}^{K} h_k \cos(\frac{w_0}{k} t) \tag{3}$$

2.1.2 High Pass Filtering

Since modulation described above, does not yet result in a natural sounding voice, one more step is required. To get a more natural effect, high-pass filtering of sub-harmonics is needed. Authors explain that it is possible to isolate sub-harmonics by subtracting an original signal from the modified one $y_{sub} = y(t) - x_c(t)$. After applying high-pass filter, we can once again sum this signal with the original.

The final rough voice signal equation is:

$$y_{rough}(t) = x_c(t) + \alpha y_{sub}^{HP}(t) \tag{4}$$

2.2 Pitch Finding

Important step of the previously described amplitude modulation is defining the fundamental frequency of the carrier signal. To do so, authors of the original paper (1) suggested using real-time implementation of yin algorithm (3).

The yin algorithm consists of 6 steps. The steps 1-4 and 6 build upon each other, and while step 5 is independent, it still relies on the properties of those before.

2.2.1 Step 1 - Auto-correlation method

The auto-correlation is of the discrete signal is defined as

$$r_t(\tau) = \sum_{j=t+1}^{t+W-\tau} x_j x_{j+\tau} \tag{5}$$

where x is an input signal and W is a window size. Auto-correlation shows peaks of a periodic signal at the multiplies of the period τ.

2.2.2 Step 2 - Difference function

The difference function is defined as

$$d_t(\tau) = \sum_{j=1}^{W} (x_j - x_{j+\tau})^2 \tag{6}$$

This function reaches values near zero at τ equal to the period of a periodic signal.

2.2.3 Step 3 - Cumulative mean normalized difference function

While the essence of this step is extremely similar to the step 2, it greatly improves the results. Cumulative mean normalized difference function is defined as

$$d'_t = \begin{cases} 1, & \text{if } \tau = 0, \\ d_t(\tau) / [(1/\tau) \sum_{j=1}^{\tau} d_t(j)] & \text{otherwise.} \end{cases} \tag{7}$$

2.2.4 Step 4 - Absolute threshold

This steps selects the smallest τ which satisfies the following condition

$$d'_t(\tau) < T \text{ and } d'_t(\tau) < d'_t(\tau + 1)$$

where T is the threshold value. If no τ satisfies the condition, global minimum of d'_t is selected.

2.2.5 Step 5 - Parabolic interpolation

Since previous steps only take multiples of sampling rate as a possible period, the parabolic interpolation step is used. This allows for a more accurate estimation. The parabolic interpolation of 3 points can be defined as

$$\hat{x} = \frac{\frac{1}{2}(y_- - y_+)}{(2 * y - y_- - y_+)} \tag{8}$$

2.2.6 Step 6 - Best local estimate

Explanation of this step is omitted, as the best local estimate was skipped during the implementation process.

3 Implementation

The implementation is heavily based on the equations defined in the previous sections. The pseudo-code for general processing loop is shown below:

```
pitch = yinFindPitch(input_signal);
mod_signal = amplitudeModulation(input_signal, pitch);
sub_harmonies = mod_signal - input_signal;
filtered_signal = highPassFilter(sub_harmonies);
output_signal = input_signal + dry_wet * filtered_signal;
```

The original source code can be found in my GitHub Repository (4), and this particular part is located in processBlock function in PluginProcessor.cpp file.

3.1 Pitch Finding

Yin algorithm was implemented as a separate class and can be found in yin.h and yin.cpp files in the GitHub Repository. The main algorithm code is presented below. It is based on equations introduced in section 2.2. The code was optimized, so it does not reflect the step-by-step approach in a straightforward way. The most important parts are highlighted by comments.

```
float Yin::getPitch(const juce::AudioBuffer<float>& buffer)
{
        const auto x = buffer.getReadPointer(0);
        const auto bufferSize = buffer.getNumSamples();

        auto sum = 0.f;
        auto minimumValue = std::numeric_limits<float>::max();
        auto minTau = 1;


        dt[0] = 1; // the first value of dt' is 1
        for(auto tau = 1; tau < bufferSize; tau++){
                auto diffFunc = 0.f;
                for(auto j = 1; j + tau < bufferSize; j++){
                        const auto correlation = (x[j] - x[j+tau]); // STEP 1 - Correlation
                        diffFunc += correlation * correlation; // STEP 2 - Difference Function
                }
                sum += diffFunc; // Summation from Mean Normalized Difference
                dt[tau] = diffFunc / (sum / static_cast<float>(tau)); // STEP 3 - Mean Normalized Difference
        }

        // STEP 4 - Absolute threshold
        for(auto tau = 1; tau < bufferSize; tau++){
                if(dt[tau] < minimumValue){ // found value lower then current minimum
                        minimumValue = dt[tau]; // save the value
                        minTau = tau; // save the index

                        // if we have found value lower than threshold and it is a local minimum we have found the
                            answer -> break!
                        if((tau == bufferSize - 1 || dt[tau] < dt[tau + 1]) && minimumValue < threshold) break;
                }
        }

        // STEP 5 - Parabolic Interpolation
        const auto result = parabolicInterpolation(dt, minTau, bufferSize);

        if(result > 0.f){
                currentPitch = sampleRate / result;
        }

        return currentPitch;
}
```

3.2 Amplitude Modulation

The amplitude modulation, similarly to pitch finding, was implemented as a separate class, and can be found in roughness.h and roughness.cpp files in the GitHub Repository. The main processing algorithm code is presented below. It is based on equations introduced in section 2.1.1.

```
void Roughness::process(juce::AudioBuffer<float>& buffer){
        auto* leftChannel = buffer.getWritePointer(0);
        auto* rightChannel = buffer.getNumChannels() > 1 ? buffer.getWritePointer(1) : leftChannel;

        for(auto i = 0; i < buffer.getNumSamples(); i++){
                f0 = f0Smoothing.getNextValue();
                update();

                auto xm = 1;

                for(auto j = 0; j < subHarmonics; j++){
                        curPhase[j] += phaseStep[j];
                        if(curPhase[j] > 1.f){
                                curPhase[j] -= 1.f;
                        }
                        xm += hAmp * h[j] * cos(juce::MathConstants<float>::twoPi * curPhase[j]);
                }

                leftChannel[i] *= xm;
                rightChannel[i] *= xm;
        }

}
```

3.3 Real-Time Plugin

The project was implemented as a real-time effect audio plugin using JUCE framework (2). Compiled plugin can be opened in any macOS DAW which supports either AU or VST3 plugin formats. The implementation assumes single channel input signal.

The plugin allows the user control to values of the following parameters:

- Dry/Wet - α parameter from Eq 4. Controls the amount of filtered sub-harmonics which are summed back with the input signal.
- Sub-Harmonics - K parameter from Eq 3. Controls the number of cosines of which the modulating signal consists.
- Mod Amplitude - scaling factor of h_ks from Eq 3. A set of h_k values is predefined in the code, this parameter allows the user to scale all of them by the same factor.

- High Pass Frequency - frequency of the high-pass filter used for filtering the sub-harmonics, explained in section 2.1.2.

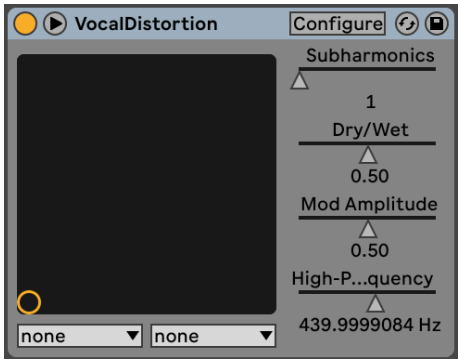


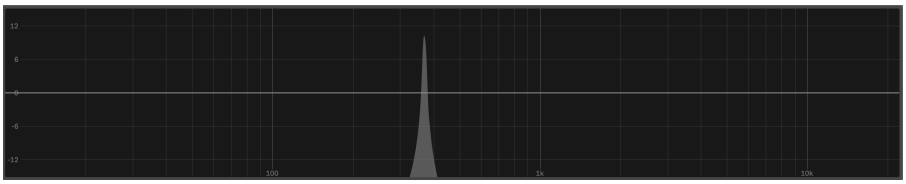
Figure 1: The plugin window

3.4 Results

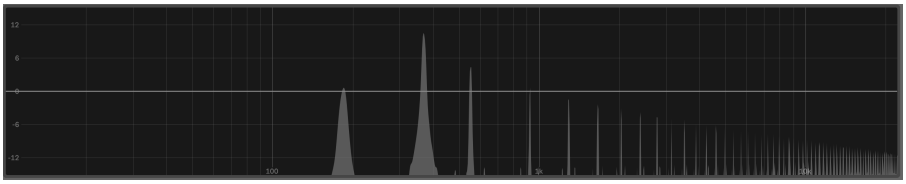
3.4.1 Voice Distortion

"In the spectral domain, a rough voice is characterized by a low Harmonic-to-Noise Ratio (HNR), with the presence of noise and subharmonics. [...] In the time domain, rough voices are mainly characterized by the presence of important degrees of jitter and shimmer." Gentilucci et al. (1)

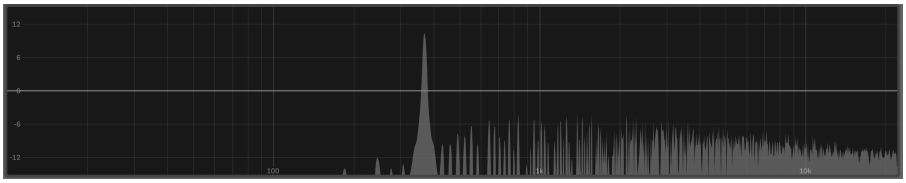
The results of the implementation are satisfactory and match the description above. Using this effect brings the roughness, which authors of original algorithm defined. By looking at frequency spectrum before and after the effect, it is clearly visible that the sub-harmonics are indeed added to the signal, at the same time creating a bit of noise at the higher frequencies.



(a) Before the effect



(b) After the effect



(c) After the effect with sub-harmonics parameter set to 3

Figure 2: Frequency spectrum of a sine wave processed by the plugin

3.4.2 Pitch Finding

A simple comparison was made between Ableton's in build "Tuner" audio effect and the yin algorithm implementation. With a simple sine wave as an input, the error rate of yin algorithm was 0.0005% and with a more complex sounds the error rose to roughly 0.2%. The performance achieved through the Yin algorithm implementation has exceeded initial expectations.

References

[1] Marta Gentilucci, Luc Ardaillon, and Marco Liuni. Vocal distortion and real-time processing of roughness. *International Computer Music Conference Proceedings*, 2018, 2018. ISSN 2223-3881.

[2] JUCE framework. `https://juce.com/`. Accessed: 2023-11-10.

[3] Alain Cheveigné and Hideki Kawahara. Yin, a fundamental frequency estimator for speech and music. *The Journal of the Acoustical Society of America*, 111:1917–30, 05 2002. doi: 10.1121/1.1458024.

[4] Github repository vocal distortion. `https://github.com/mp-smc23/vocal-distortion`.