Real-time implementation of vocal distortion

Marco Fiorini
Aalborg University Copenhagen, Sound and Music Computing, 7th Semester
Sound Processing

***Abstract*—This research focuses on the processing of the singing voice, implementing parametric controlled distortion. Specifically, a real-time VST plugin was developed in MATLAB, based on an existing algorithm, using amplitude modulation to produce distortion and high pass filtering to enhance the sub-harmonics generated from the modulation. The result is a software tool that facilitates the production of vocal distortion and roughness and can be controlled by the singer during the performance.**

I. Introduction

Vocal extended techniques, including distortion, roughness and noise production, have been widely used in the last century, from composers like Karlheinz Stockhausen (*Spirale*) to non-idiomatic improvisers as Sidsel Endresen and Stian Westerhus (*Bonita*), just to name a few examples.

As an improvising musician, working with singers able to produce and control these techniques has always been very inspiring and rewarding; that's why I got interested in the work of singer and researcher Marta Gentilucci. In their research, Gentilucci et al. focused on distortion as "a sound quality that can be composed as much as harmony, rhythm, pitch, etc." [1], to enlarge the classical vocal possibilities. Therefore, they aimed at implementing a steadily controlled manipulation of distortion, by nature fragile and unstable, specifically applied to a voice signal. Whereas many different approaches have been used by a wide variety of researchers to create the perceived effects of vocal distortion, Gentilucci et al. [1] decided to model such effects based on amplitude modulation and time-domain filtering, resulting in an open-source patch for Max/MSP [2], called *Angus* [3].

My work in this paper was then based on their algorithm, described in [1] and presented in detail in the following section, and implemented as a real-time plugin in MATLAB, with a few variations, as showed later.

A. Algorithm Description

In the case of amplitude modulation (AM), the focus is on altering the amplitude components of oscillators involved, namely carrier and modulator. The carrier refers to the oscillator that is being altered and the modulator refers to the oscillator that changes the carrier. Let $x_c(t) = A_c cos(w_c t)$ be the carrier signal, with angular frequency w_c[1] in (rad/sec) and amplitude $A_c \in [0, 1]$, and $x_m(t) = A_m cos(w_m t)$ be the modulating signal with angular frequency w_m and amplitude $A_m \in [0, 1]$ (also called modulation index or modulation depth). In [4], the classic AM synthesis is then mathematically defined by Park as:

$$y(t) = cos(w_c t) \cdot (A_m cos(w_m t) + A_c) \tag{1}$$

The resulting signal consists of the original carrier component and two new components with altered frequency and amplitude characteristics, as shown in Figure 1.

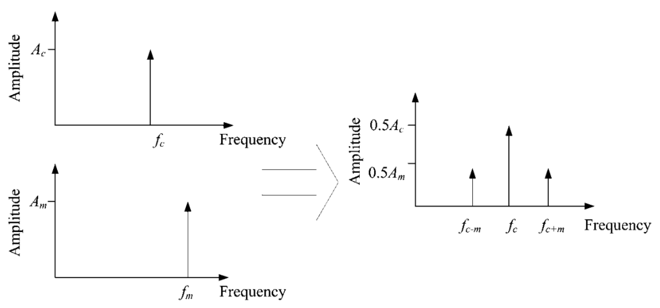


Fig. 1. Sum and difference and carrier component in output of AM synthesis [4]

However, in [1], Gentilucci et al. gave a slightly different implementation of this AM synthesis, which will be used in this study:

$$y(t) = x_c(t) \cdot (x_m(t) + 1) \tag{2}$$

The variation consists in a different handling of both carrier and modulation amplitudes, so that the two new sinusoids generated from the AM are:

$$y_{\pm}(t) = \frac{A_c A_m}{2} cos((w_c \pm w_m)t) \tag{3}$$

I believe that this choice is motivated by two factors:

[1] $w = 2\pi f$, where f is the frequency in Hz

- The voice is a complex signal, thus here $x_c(t)$ could be represented as a sum of N harmonic sinusoids: $x_c(t) = \sum_{i=1}^{N} A_i cos(iw_0 t)$, where $w_0 = 2\pi f_0$
- By choosing appropriate values for w_m, it is possible to generate sub-harmonics between each harmonics at frequencies $iw_0 \pm w_m$, the distance of each sub-harmonic to its related i harmonic being thus equal to w_m

Nevertheless, using only a modulation doesn't result in a natural-sounding effect for a voice signal. It is thus necessary to high pass filter the sub-harmonics. As the original signal $x_c(t)$ is fully preserved in the modulated signal, the sub-harmonics can easily be isolated by simply subtracting this original signal from the modulated one:

$$y_{sub}(t) = y(t) - x_c(t) \tag{4}$$

Once isolated, the sub-harmonics are high pass filtered before being added back to the original signal by a simple summation:

$$y_{rough}(t) = x_c(t) + \alpha y_{sub}^{HP}(t) \tag{5}$$

where $\alpha y_{sub}^{HP}(t)$ represents the high pass filtered sub-harmonics and $\alpha \geq 0$ is the mixing factor, or modulation index (in the VST implementation this parameter will be called Modulation Gain).

II. Implementation

My implementation of the algorithm proposed in [1] and presented in Section I-A was written in MATLAB as a real-time audio plugin.

As described in by DeVane in [5], every audio plugin in MATLAB must inherit from the `audioPlugin` class. First of all I defined a series of properties for the plugin, including a master `gain` control, `amplitude` and `freq` of the AM modulation signal, as well as `cutoffFrequency` and `Q` for the high pass filter. I also added some `private` variables to control the `sine` modulator and the coefficients of the high pass filter, as shown in Listing 1. Then I created a `sine audioOscillator` to modulate the input signal, as shown in Listing 2. The parameters for the GUI are then and mapped to a series of `set` methods, in order to control them.

```
properties
    gain = 1;
    amplitude = 1;
    freq = 200;
    cutoffFrequency = 20;
    Q = sqrt(2)/2;
    state_pass = zeros(2);
end

properties (Access = private)
    sine;
    num_pass = [.5 -1 .5];
    den_pass = [1 1 1];
end
```

Listing 1. Public and private properties of the plugin

```
function p = amplitudeModulationPlugin
    p.sine =  audioOscillator('SignalType',...
    'sine', 'Frequency',...
    p.freq, 'Amplitude', p.amplitude);
end
```

Listing 2. Audio oscillator used for the AM modulation

The main core of the plugin is contained in the `process` method, like in every MATLAB plugin. This method is shown in Listing 3 and implements the main algorithm as described earlier, using the private function `calculateCoeffientPass` to determine the numerator and denominator coefficients of the high pass filter (Listing 4).

```
function out = process(p, in)
    [frameSize,channels] = size(in);
    updateFrameSize(p, frameSize);

    sig = ones(size(in));
    sig = p.sine();

    if channels == 1
        sig_multiplier = sig;
    else
        sig_multiplier = [sig sig];
    end

    dBGain = 10 ^ (p.gain / 20);

    processedOut = (sig_multiplier + 1) .* in;
    subharmonics = processedOut - in;

    calculateCoefficientsPass(p);

    [filt_pass, p.state_pass] = filter(p.num_pass,...
    p.den_pass, subharmonics, p.state_pass);

    yRough = in + (p.amplitude .* filt_pass);
    out = dBGain .* yRough;

end
```

Listing 3. Process function of the plugin, implementing the algorithm

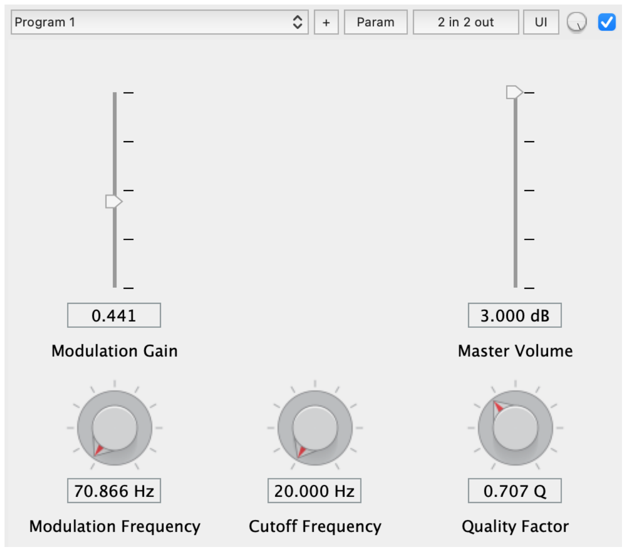
```
function calculateCoefficientsPass(p)
    w0 = 2 * pi * p.cutoffFrequency/getSampleRate(p);
    alpha = sin(w0)/(2 * p.Q);
    cosw0 = cos(w0);
    norm = 1/(1 + alpha);
    p.num_pass = (1 + cosw0)*norm * [.5 -1 .5];
    p.den_pass = [1 -2*cosw0*norm (1 - alpha)*norm];
end
```

Listing 4. Computing high pass filter coefficient based on [6]

Finally, in order for the plugin to work correctly in a DAW, a few passages were needed. First, `validateAudioPlugin` was run, to diagnose audio plugin constraint errors. This also checks for many code generation errors. The VST plugin was then generated using `generateAudioPlugin`.

III. Results

To test the plugin in a DAW, I installed the VST in Reaper [7]. Here I created a new audio track and added the generated plugin in the FX chain. I also connected a Korg Nanokontrol (Korg Inc. Tokyo, Japan) to my computer and mapped the main parameters of the plugin via MIDI, to adjust and control them in real-time using the sliding faders and rotating knobs of the Nanokontrol. The GUI of the plugin, as seen in Reaper, is shown in Figure 2.



Modulation Gain: 0.441
Master Volume: 3.000 dB
Modulation Frequency: 70.866 Hz
Cutoff Frequency: 20.000 Hz
Quality Factor: 0.707 Q

Fig. 2. GUI of the VST plugin, as seen in Reaper

The VST is available for free download[2], while an audio example of the resulting effect of the plugin on a vocal performance can be heard at the following url:

https://drive.google.com/file/d/16HY961YQ4C3kutcJ0vTCoxeLJt0kWgiy/view?usp=sharing.

This is a recording of me singing the first verse of the song *Hide and Seek* by Imogen Heap[3]. Here, the Modulation Frequency was kept constant at 70.866 Hz, in order to best suit the drone harmonizing effect wanted on the vocals. Cutoff Frequency was fixed at 20 Hz, Quality Factor at 0.707 and Master Volume at 3 dB, as shown in Figure 2. While singing, I moved the sliding fader on the Korg Nanokontrol mapped to the Modulation Gain control (the only parameter I changed during this recording), following the performance according to my taste (the Modulation Gain automation is displayed in Figure 3). I also added a reverb plugin (built-in in Reaper) to fit my personal taste and enhance the oniric qualities of the song.

[2]https://drive.google.com/file/d/1-2P3_67IuGOy6FtCfx-9QbtP54H7SQ29/view?usp=sharing

[3]https://open.spotify.com/track/121so7t3AeX6nLMvxy9ZP9?si=0d971e1075e24b04

IV. Discussion

The goal of the research described in this paper was to investigate the distortion of the singing voice. Specifically, a real-time VST plugin was implemented in MATLAB, based on an existing algorithm, using amplitude modulation to produce distortion and high pass filtering to enhance the sub-harmonics generated from the modulation. The result is a software tool that facilitates the production of vocal distortion and roughness and can be controlled by the singer during the performance. Because this VST does not depend on the spectral characteristics of the input sound but only on frequency content, its use can easily be extended to non-vocal sounds. The current model operates as synthesis tool with manual adjustment of all parameters.

A particular limitation, comparing this VST to the reference plugin generated in [1], has to do with f_0 estimation. In fact, in their implementation, Gentilucci et al. automatically detected the picth of the fundamental frequency f_0 of the input signal and thus the modulation frequency was set according to this f_0. Gentilucci et al. used a real-time implementation of the YIN algorithm presented by de Cheveigné & Kawahara in [8]. This algorithm is considered one of the best publicly available methods for pitch detection but unfortunately it cannot run entirely on MATLAB code, demanding routines implemented in low-level programming languages like C, as in the YIN Tuner published by Neuron in [9].

As future developments, real-time pitch detection could thus be used to tune the VST accordingly to the input fundamental frequency f_0. To improve the sound quality of the tool, compression and reverb could also be added to the processing chain.

V. Conclusions

The present implementation of sound processing techniques contributes to a larger context of studies on vocal manipulation and modulation synthesis literature. It is hoped that the research done during this study can work as a personal foundation for future investigations in the field of vocal processing and plugin development.

References

[1] M. Gentilucci, L. Ardaillon, and M. Liuni, "Vocal distortion and real-time processing of roughness," 2018. [Online]. Available: https://www.researchgate.net/publication/326876071

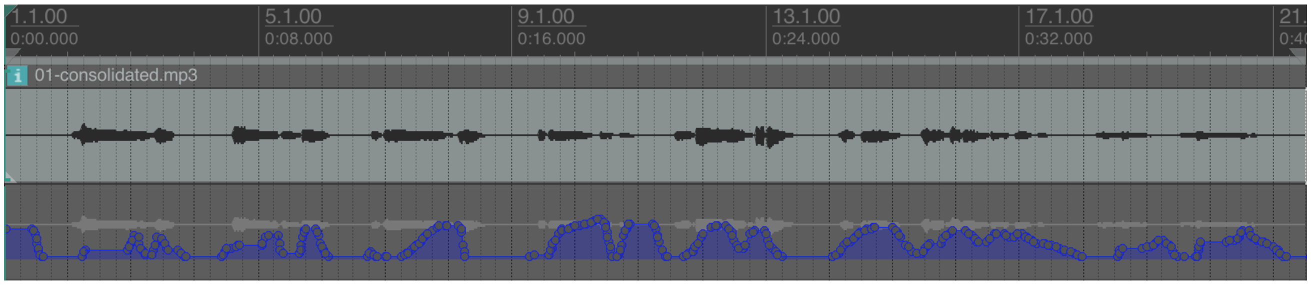


Fig. 3. Track recorded in Reaper. In the upper part of the figure, the waveform of the vocal audio file is visible. In the lower part, the blue curve represents the stored automation of the Modulation Gain parameter, controlled in real-time during the recording of the performance through a fading slider on a Korg Nanokontrol.

[2] "Max/MSP, Cycling '74." [Online]. Available: https://cycling74.com/

[3] "Angus: the highway to yell – cream." [Online]. Available: http://cream.ircam.fr/?p=787

[4] T. H. Park, "Introduction to digital signal processing: Computer musically speaking," *Introduction to Digital Signal Processing: Computer Musically Speaking*, pp. 1–429, 1 2009.

[5] C. DeVane, "Automatically generating vst plugins from matlab code," 2016. [Online]. Available: www.aes.org/e-lib/browse.cfm?elib=18142.

[6] R. Bristow-Johnson, "Cookbook formulae for audio eq biquad filter coefficients," 2004. [Online]. Available: https://webaudio.github.io/Audio-EQ-Cookbook/audio-eq-cookbook.html

[7] "Reaper | audio production without limits." [Online]. Available: https://www.reaper.fm/

[8] A. D. Cheveigné and H. Kawahara, "Yin, a fundamental frequency estimator for speech and music a)," 2002.

[9] Neuron, "Yin tuner - file exchange - matlab central," 2022. [Online]. Available: https://it.mathworks.com/matlabcentral/fileexchange/54663-yin-tuner