EFFECTS UNIT SIMULATION USING PYTHON: CHORUS AND REVERB

Ke Yang
yangk@kth.se

Haohao Yu
haohao@kth.se

ABSTRACT

Effect units are electronic or digital devices designed to alter the sound signal of musical instruments, creating a ton of sound textures and characteristics. Nowadays, software simulations of these physical units are also commonly used, allowing for a more accessible means to craft and manipulate sound. This report investigates two of these effects, chorus and reverb, and discusses their origins, principles, and simulations in Python.

1. INTRODUCTION

The chorus and reverb effects are essential pillars in the audio engineering landscape, shaping the texture and spatial quality of sound in recorded and live audio. This research explores how these effects are replicated and simulated through algorithms in Python, providing insight into their transition from analog to digital technologies.

The history of effects reflects humanity's enduring interest in manipulating sound as engineers have continuously searched for ways to enhance experiences. Early electromechanical innovations by Hammond and Leslie paved the way for the development of digital reverb and modulation effects. [1] Innovations like plate reverberators introduced by EMT in the mid 20th century furthered the quest for audio effects. [1]

In particular, the chorus effect, known for adding richness and depth to sound evolved alongside advancements such as multitrack recording and digital delay units. The introduction of devices like the Lexicon Delta T enabled precision in controlling chorus effects by incorporating delays and diverse modulations. [1] The effects based on time manipulation closely linked to flanging and echo were developed to create textures and were further enhanced through random modulation methods. [1]

Recreating reverberation initially emerged from acoustics. Studios achieved this using chambers before transitioning into digital realms. The digitization of reverb inspired by the decay and diffusion of sound in environments is closely associated with the research and advancements of Schroeder and Moorer, whose algorithms serve as the foundation for many modern digital reverb processors. [2]

This research's inspiration arises from an appreciation for the history of these effects and the transformative impact they have on contemporary music production. Python's computational capabilities provide an approach to exploring these effects. Through simulations of chorus and reverb, the goal of this study is to grasp their intricacies and utilize them creatively to push the boundaries of sound engineering.

This project seeks to contribute to discussions by blending insights with practical applications. By revisiting works that have shaped audio effects history and leveraging digital signal processing techniques, this research aims to connect the past with the present.

2. METHOD

2.1 Chorus

The chorus effect works by duplicating the audio signal and slightly altering the time delay and pitch of these copies, thereby creating the effect of multiple sound sources playing simultaneously. This effect mimics the rich, expansive sound produced when multiple similar sound sources, such as a choir or string orchestra, perform together.

2.1.1 Preliminary Research and Design Methodology

The chorus effect is closely related to the vibrato effect. [2] The chorus effect is designed to emulate the collective sound of a choir from a single voice, imparting a 'thickening' to the sound by mimicking the slight variations in timing and pitch inherent in a group performance. Similarly, the vibrato effect, although also reliant on time-delay modulation, aims not to create echo but to slightly alter the pitch of the signal to emulate the natural fluctuations that occur in live singing or instrument playing. This is achieved by varying the delay time, which in turn modulates the frequency, simulating a speeding up or slowing down of the audio signal. [3] The chorus effect is established by mixing a dry, unprocessed signal with a signal that has been affected by vibrato, typically within a delay range of 10–50 ms, to avoid creating an audible echo. This delay parameter is set close to the echo threshold of human hearing, thereby ensuring that the effect enhances the sound without generating discrete echoes.Fig. 1 depicts the block diagram of a feedforward modulated delay line that is fundamental to creating the chorus effect, illustrating the flow from input through modulation to the output, where the signal x[n] is subjected to an LFO-driven delay and subsequently mixed with the original signal in proportions governed byg to yield the output y[n]. The parameters governing the chorus effect, including rate and depth, modulate the amplitude and frequency of the low-

frequency oscillator (LFO) that affects the delay time, similar to how vibrato is controlled. An additional pre-delay parameter may be implemented to ensure the modulated delay remains within the desired temporal range.

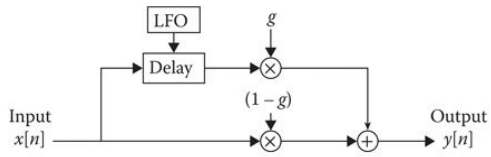

Figure 1. Block diagram of feedforward modulated delay

2.1.2 Replication in VCV Rack 2

Following the research phase, the study proceeded to replicate the selected chorus effect within VCV Rack 2, a modular synthesizer software that offers a tangible analog to digital comparison.

The simulation of the chorus effect within the VCV Rack 2 environment as Fig. 2 ,begins with the generation of an original audio signal by the WT VCO module, which is capable of producing a variety of waveforms. Subsequently, this initial signal undergoes duplication, resulting in more copies, each of which is imbued with a slight delay. The imposition of delay on these copies is not static. It is dynamically modulated by a Low Frequency Oscillator (LFO module). This modulation signifies that the quantum of delay imparted to each copy fluctuates over time, thereby instilling a temporal variation in the delay effect. The modulated copies are then merged back with the original audio signal using the VCA MIX module. This module mixes the original signal with its delayed versions, creating an effect as if multiple sound sources are playing simultaneously. However, these sounds differ slightly due to the variations in delay, introduced by the LFO. This merging results in the chorus effect, characterized by a more complex sound texture that expand the spatial dimension of the audio.

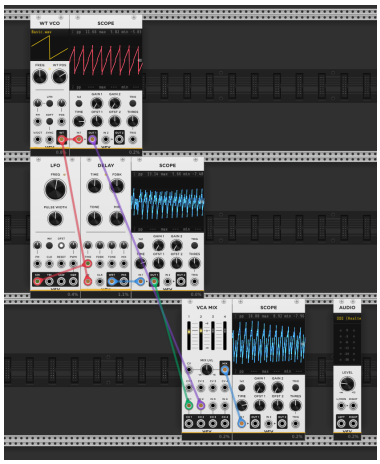

Figure 2. Modular synthesis setup for chorus effect simulation using VCV Rack 2

2.1.3 Implementation in Python

The transition from a modular synthesizer environment to a Python-based simulation is facilitated through a modular programming approach. This approach involves encapsulating each component of the chorus effect within distinct functions. Below is a comprehensive description and explanation of the Python code, illustrating the modular implementation of the chorus effect.

```python
#LFO
1 usage
def lfo(frequency, length, sample_rate, depth=1.0, phase=0):
    # Calculate the number of samples
    num_samples = int(length * sample_rate)

    # Create a time array
    t = np.linspace(0, length, num_samples, endpoint=False)

    # Generate LFO signal using a sine wave
    lfo_signal = depth * np.sin(2 * np.pi * frequency * t + phase)

    return lfo_signal
```

Figure 3. Code for LFO function

The first function lfo can be seen in Fig. 3. It generates a low-frequency oscillator signal using a sine wave as equation (1) , which is a sine wave that modulates the delay time of the audio signal. The parameters of the function include the frequency of the LFO, the length of the LFO signal, sample rate of the audio, and the depth and phase of modulation. The function calculates the number of samples and creates a time array, which is then used to generate the LFO signal.

$$lfo(t) = depth \cdot \sin(2\pi \cdot frequency \cdot t + phase) \quad (1)$$

The second function dynamic delay can be seen in Fig. 4. It implements a delay effect where the amount of delay is modulated over time by an LFO signal. The input audio signal is processed sample by sample, with the current delay d(n) determined by adding the static delay (ms) to the value of the LFO signal at that point in time. The current delay in samples is then calculated as (2) :

$$d(n) = int\left((delay\_ms + lfo(n)) \cdot \frac{sample\_rate}{1000}\right) \quad (2)$$

The function employs a delay buffer to store the delayed

```python
def dynamic_delay(audio_signal, delay_ms, lfo_signal, sample_rate, feedback=0.0):
    # Initialize the delay buffer
    max_delay_ms = np.max(delay_ms + lfo_signal)
    max_delay_samples = int(sample_rate * max_delay_ms / 1000)
    delay_buffer = np.zeros(max_delay_samples)
    output_signal = np.zeros_like(audio_signal)

    for i in range(len(audio_signal)):
        # Calculate the current delay in samples
        current_delay_samples = int((delay_ms + lfo_signal[i]) * sample_rate / 1000)

        # Make sure the delay is within the range of the delay buffer
        current_delay_samples = np.clip(current_delay_samples, 1, max_delay_samples - 1)

        # Read the delayed sample from the delay buffer using linear interpolation
        fraction = current_delay_samples % 1
        delayed_sample = (1 - fraction) * delay_buffer[int(current_delay_samples - 1)] + fraction * delay_buffer[
            int(current_delay_samples)]

        # Write the current input sample to the delay buffer
        delay_buffer = np.roll(delay_buffer, -1)
        delay_buffer[-1] = audio_signal[i] + delayed_sample * feedback

        # Store the output sample
        output_signal[i] = delayed_sample

    return output_signal
```

Figure 4. Code for Dynamic Delay function

samples. For samples that require a delay time falling between discrete indices within the buffer, linear interpolation is applied. The interpolated value y for a non-integer

delay d with a fractional part f is given by (4) :

$$y = (1 - f) \cdot d\_b[int(d)] + f \cdot d\_b[int(d) + 1] \quad (3)$$

```
#VCA Mix
!usage
def vca_mix(signal_a, signal_b, gain_a=1.0, gain_b=1.0):
    len_a = len(signal_a)
    len_b = len(signal_b)
    if len_a > len_b:
        signal_b = np.pad(signal_b, (0, len_a - len_b), 'constant')
    elif len_b > len_a:
        signal_a = np.pad(signal_a, (0, len_b - len_a), 'constant')

    # Apply the gains to each signal
    signal_a = signal_a * gain_a
    signal_b = signal_b * gain_b

    # Mix the signals
    mixed_signal = signal_a + signal_b

    return mixed_signal
```

Figure 5. Code for Dynamic Delay function

The third function can be seen in Fig. 5. takes two signals and applies a linear combination of them after scaling by their respective gain factors:

$$y[n] = gain_a \cdot x_1[n] + gain_b \cdot x_2[n] \quad (4)$$

2.2 Reverb

In audio effects, reverberation plays a role in adding depth and atmosphere to sound. This section dives into the algorithms used in reverb, mainly focusing on the work of Manfred Schroeder and the advancements made by James Moorer. [2] Their algorithms incorporate filters like comb filters, all-pass filters, early reflections and low-pass filters to recreate the acoustic reverberation found in different spaces. These techniques not only replicate real world environments in recordings but also enhance the immersive experience for listeners by making sounds resonate as they would naturally.

2.2.1 Preliminary Investigation and Design Approach

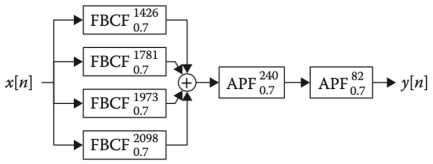

Figure 6. Block diagram of Schroeder reverb

Manfred Schroeder's innovations in reverberation paved the way for reverb techniques. His approach involves using comb filters followed by a series of all-pass filters to create a diffuse echo pattern that mimics the sound reflections within a room. [4] The spread of sound plays a role in preventing the ringing sounds often linked with early digital reverb setups.

James Moorer took Schroeder's idea further by adding early reflections and low-pass filters (LPF) into the feedback of the comb filters (FBCF) structure. [5] Early reflections, the echoes bouncing off walls, are essential for crafting a realistic sense of space and depth. The LPF deals with the problem of high-frequency buildup in the

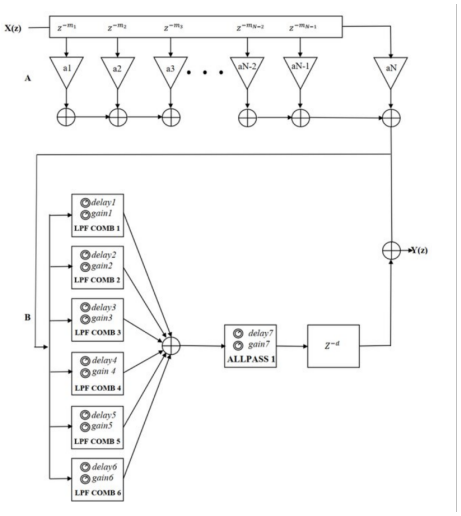

Figure 7. Block diagram of Moorer reverb

feedback loop, ensuring a fading away of sound over time similar to what happens in real-world spaces due to air absorption.

2.2.2 Implementation in Python

In the Python-based simulation of the Moorer Reverb algorithm, the methodology reflects the modularity found in synthesizer environments, delineating each effect component within separate functions for clarity and ease of manipulation.

```
def early_reflections(input_sample, buffer, fs, n):

    # Convert delay times from milliseconds to samples
    delay_times = np.array([0, 0.015, 0.016, 0.017, 0.018, 0.020, 0.025, 0.030, 0.035, 0.040,
                            0.045, 0.050, 0.055, 0.060, 0.065, 0.070, 0.075, 0.080, 0.085]) * fs
    delay_times = np.round(delay_times).astype(int)

    gains = np.array([1, 0.85, -0.75, 0.65, 0.55, 0.45, -0.35, -0.25, -0.15, 0.15,
                      -0.25, 0.35, 0.45, 0.55, 0.65, -0.75, 0.85, 0.95, 1])


    # Update circular buffer with current sample
    buffer[n % len(buffer)] = input_sample

    # Initialize output sample
    output_sample = 0

    # Convert delay times from milliseconds to samples and ensure they are integers
    delay_times = np.fix(fs * np.array([0, 0.01277, 0.01283, 0.01293, 0.01333,
                                        0.01566, 0.02404, 0.02679, 0.02731, 0.02737,
                                        0.02914, 0.02920, 0.02981, 0.03389, 0.04518,
                                        0.04522, 0.04527, 0.05452, 0.06958])).astype(int)

    # Loop through all taps
    for tap in range(len(delay_times)):
        # Calculate the circular buffer index for the current tap
        index_tdl = (n - delay_times[tap] - 1) % len(buffer)
        # Add current tap with output
        output_sample += gains[tap] * buffer[index_tdl]

    return output_sample, buffer
```

Figure 8. Code for Early Reflections function

The first step is to design a function to realize early reflections in the reverb effect. The early reflections are simulated by adding delayed versions of the input signal, each with its own gain. This simulates the effect of sound bouncing off various surfaces before reaching the listener. In this function, 'delays' and 'gains' correspond to τ_i and g_i in the mathematical formulation:

$$y_{er}(n) = \sum_{i=1}^{M} g_i \cdot x(n - \tau_i)$$

The function iterates through each early reflection, applying the specified delay and gain to the input signal, and accumulating the results.

```python
def lpcf(input_sample, buffer, fs, n, delay, fb_gain, amp, rate, fb_lpf):

    # Calculate time in seconds for the current sample
    t = (n - 1) / fs
    # Apply LFO modulation to the delay
    frac_delay = amp * np.sin(2 * np.pi * rate * t)
    int_delay = int(np.floor(frac_delay))
    frac = frac_delay - int_delay

    # Determine indexes for the circular buffer
    len_buffer = len(buffer)
    index_c = (n - 1) % len_buffer  # Current index
    index_d = (n - delay - 1 + int_delay) % len_buffer  # Delay index with LFO modulation
    index_f = (n - delay - 1 + int_delay + 1) % len_buffer  # Fractional index for interpolation

    # Interpolate between the delayed samples for fractional delay
    out = (1 - frac) * buffer[index_d] + frac * buffer[index_f]

    # Store the current output in the buffer and apply LPF in the feedback path
    buffer[index_c] = input_sample + fb_gain * (0.5 * out + 0.5 * fb_lpf)

    # Update feedback LPF value for the next sample
    fb_lpf = out

    return out, buffer, fb_lpf
```

Figure 9. Code for Low-pass Comb Filter function

Then, the feedback comb filter with a low-pass filter creates the dense reverb tail. It's implemented with a feedback loop that includes a low-pass filtering step to simulate frequency-dependent absorption.

$$y_{fbcf}(n) = x(n) + g \cdot LPF(y_{fbcf}(n - \delta))$$

Here, 'lpf' is an instance of a LowPassFilter class that simulates the LPF operation $LPF(y)$. The feedback comb filter's operation is realized by iterating over the input signal and applying the delay and feedback gain within a low-pass filtering context.

```python
def apf(input_sample, buffer, fs, n, delay, gain, amp, rate):

    # Calculate time in seconds for the current sample for LFO modulation
    t = (n - 1) / fs
    frac_delay = amp * np.sin(2 * np.pi * rate * t)
    int_delay = int(np.floor(frac_delay))
    frac = frac_delay - int_delay

    # Determine indices for the circular buffer
    len_buffer = len(buffer)
    index_c = (n - 1) % len_buffer  # Current index
    index_d = (n - delay - 1 + int_delay) % len_buffer  # Delay index
    index_f = (n - delay - 1 + int_delay + 1) % len_buffer  # Fractional index for interpolation

    # Interpolation for fractional delay adjustment
    w = (1 - frac) * buffer[index_d] + frac * buffer[index_f]

    # Compute the output of the APF
    v = input_sample - gain * w
    out = gain * v + w

    # Update the buffer with the current processed input
    buffer[index_c] = v

    return out, buffer
```

Figure 10. Code for All-pass Filter function

Finally, the all-pass filters are used to disperse the phase of the signal uniformly across the frequency spectrum without affecting its amplitude, enhancing the diffusion of the reverb. This function directly translates the all-pass filter formula into code:

$$y_{apf}(n) = -g \cdot x(n) + x(n - \phi) + g \cdot y_{apf}(n - \phi)$$

It uses a for-loop to process each sample of the input signal according to the all-pass filtering principle.

Through these modular functions and calculations, the Python implementation recreates the Moorer Reverb algorithm, demonstrating the intersection of digital signal processing theory and practical application in audio engineering.

3. RESULT

3.1 Chorus

In the results section of the project on the preliminary research and methodology of the chorus effect, the testing phase involved audio excerpts sourced from YouTube videos: `https://www.youtube.com/watch?v=zmN7fK3fKUE`. The excerpts were imported into an audio analysis suite, where their waveforms and spectrograms can be seen in Fig.12, 13, and 11. The spectrogram analysis revealed that

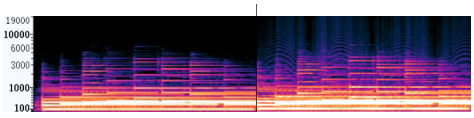

Figure 11. Spectrogram for Chorus Signal

the synthetic chorus effect injected a perceptible increase in sonic brightness as evidenced by a more vibrant energy distribution across various frequencies. Furthermore, upon close inspection of the original signal's harmonic structure within the spectrogram, a diffusion of energy was noted. This dispersion manifests as a broader spread across the frequency spectrum, indicating an amalgamation of sound waves that suggests a richer and more blended auditory experience.

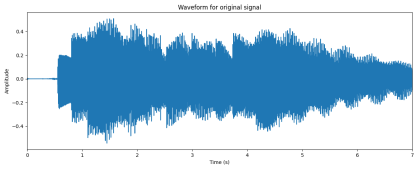

Figure 12. Waveform for Original Signal

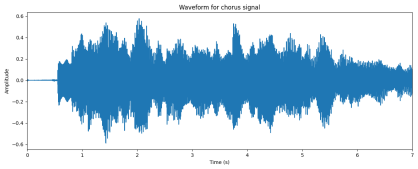

Figure 13. Waveform for Chorus Signal

Waveform inspection of the chorus-enhanced audio demonstrated a notable elevation in amplitude coupled with an escalation in complexity. This complexity is characterized by a denser and more erratic pattern of peaks and troughs in the waveform, diverging from the original's simpler and more uniform structure. These observed changes in amplitude and waveform intricacy are indicative of the depth and richness typically associated with the chorus effect, which mimics the acoustic phenomenon of multiple instruments or voices performing in unison.

3.2 Reverb

The evaluation for the reverb effect was also conducted by adding our algorithm code onto an audio clip cut from a YouTube video: `https://www.youtube.com/watch?v=TZG2K0J_A3c`.

In the waveform and spectrogram figures presented below, the top one represents the original audio file. The middle figure illustrates the audio with the reverb effect calculated using Python. Lastly, the bottom figure depicts a clip of audio processed with a physical effector.

When we look at how the reverb effect impacts a signal, we can see changes in both the wave shape and frequency range. These changes play a role in shaping how we perceive the sound.

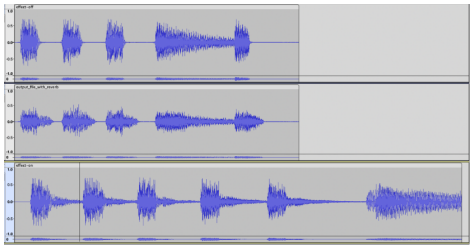

Figure 14. Waveform for Reverb Signal

One noticeable difference when adding reverb to an audio waveform is the sustain of sounds. As the sound fades away, its volume gradually decreases after the initial attack phase. This lingering quality captures the essence of reverberation found in real-world spaces. The reflections within the environment extend the decay resulting in a waveform that has a longer tail compared to the dry signal.

Additionally, peaks in the waveform representing sections of the audio signal appear to be softened with reverb. This smoothing effect occurs because the energy from the sound is spread out over time due to reflections introduced by reverb. As a result, the energy gets distributed throughout the sound's duration, creating a more consistent volume level and less distinct contrast between peaks and troughs in the waveform.

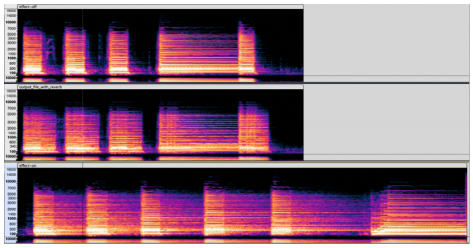

Figure 15. Spectrogram for Reverb Signal

When examining the frequency spectrogram of the signal processed with reverb, we noticed a filling effect in the spectrum. The energy distribution becomes more uniform due to the multiple reflections created by the reverb effect within the simulated space. These reflections add energy to frequency bands, enhancing the audio with a richer and livelier spectral presence. This enrichment of content is crucial to reverb as it fills in gaps in the domain, enhancing the overall fullness and complexity of the sound.

Additionally, this process mimics the behaviour of sound in physical settings, where high frequencies gradually diminish over distance and time due to air and surface absorption. Our digital reverb intentionally replicates this attenuation of high frequencies, which leads to a decay in the spectrogram, characterized by a smoother, darker trail at higher frequencies. This effect adds warmth and realism to the sound.

4. DISCUSSION

In the simulation of the chorus effect, our methodology was grounded in the fundamental understanding of its principles, leading us to employ a basic model comprising an LFO and a delay to replicate the effect. This approach, while foundational, revealed limitations in the richness and the adjustability of the simulated sound, which did not entirely match the nuanced quality of real-world chorus processors. Another disadvantage was the absence of more sophisticated models and a lack of intuitive parameter adjustment methods. This gap resulted in an analysis that could not directly and effectively discern the distinctions between the simulated chorus effect and the original audio signal.

For the reverb effect component, our focus lies in implementing Moorer's algorithm principles in Python. Nonetheless, the reverb effect algorithm undergoes continual optimization and refinement by researchers and engineers. In future endeavours, we aspire to enhance our code and algorithms by incorporating more advanced designs, such as the FDN (Feedback Delay Network) discussed in *Hack Audio*. Furthermore, drawing from feedback received during presentations, adjustments to the parameters within the reverb effect code are warranted to achieve a more pronounced and effective reverb effect. Additionally, for potential project extensions, we envisage the development of a user-friendly interface to facilitate easier parameter adjustment, thereby enabling comparison and deeper comprehension of the factors influencing audio effects.

5. CONCLUSION

In conclusion, this project successfully explored the chorus and reverb effects, starting with an extensive research phase that culminated in a practical application using Python. Subsequently, real-world audio excerpts were manipulated to demonstrate the chorus and reverb effect's capability to alter sound characteristics.

6. REFERENCES

[1] T. Wilmering, D. Moffat, A. Milo, and M. B. Sandler, "A history of audio effects," *Applied Sciences*, vol. 10, no. 3, p. 791, 2020.

[2] E. Tarr, *Hack Audio: An Introduction to Computer Programming and Digital Signal Processing in MATLAB*. Routledge, June 2018.

[3] P. Dutilleux and U. Zölzer, "Delays," in *DAFX: Digital Audio Effects*, U. Zölzer, Ed. John Wiley Sons, Ltd, 2002. [Online]. Available: https://doi.org/10.1002/047085863X.ch3

[4] M. R. Schroeder, "Natural sounding artificial reverberation," in *Audio Engineering Society Convention 13*. Audio Engineering Society, 1961.

[5] J. A. Moorer, "About this reverberation business," *Computer music journal*, pp. 13–28, 1979.