DSP Worksheet IV

Kyle Smith, Ishaan Jagyasi, Peyman Salimi

28th April 2025

1 Score Sonification

1.1 Implementation

We implemented two types of synthesis techniques, FM and Waveguide synthesis and added a reverb to both the synthesis files using a version of the Schroeder's reverb algorithm.

1.2 Documentation of Algorithm

1.2.1 MIDI Parsing (same for both the synthesis algorithms)

We first converted the score to MIDI by writing it in MuseScore and exporting it in MIDI. Then we proceeded to use the '`pretty_midi`' library in python to parse the midi file into individual midi note pitches, lengths and velocity.

We created an empty array to store the output of the synthesis process by finding the total length of the MIDI file. The next step was to iterate through notes and for each note find the start index (in samples), end index, note length (in samples) and total duration of the sample in samples. The total duration of the sample was then used to create a time ramp for each note's duration.

```
midi_data = pretty_midi.PrettyMIDI(midi_path)
total_duration = midi_data.get_end_time()
audio = np.zeros(int(fs * total_duration))

for instrument in midi_data.instruments:
        for note in instrument.notes:
            start_time = note.start
            end_time = note.end
            duration = end_time - start_time
            start_idx = int(start_time * fs)
            end_idx = int(end_time * fs)
            num_samples = end_idx - start_idx
            t = np.linspace(0, duration, num_samples, endpoint=
               False)
```

After parsing, we carried out synthesis process for each individual note so as to sonify the MIDI file with the chosen synthesis approach.

1.2.2 FM Synthesis

Our approach for FM synthesis was to create a modulator sine wave which in harmonic coherence with the main carrier frequency. To ensure this, we created a variable called '`harmonic_ratio`',

which defines the relation between the carrier tone frequency to modulator frequency as -

$$f_m = \text{harmonic\_ratio} \times f_0 \tag{1}$$

where f_m is the modulator frequency and f_0 is the carrier frequency (f_0 corresponded to the pitch of the midi note).

Since we created an array for each individual MIDI note, we could store specific carrier wave frequency for each MIDI note and find the harmonically relevant modulator wave frequency.

The formulae for modulator looked like following:-

$$modulator = sin(2\pi f_m t) \tag{2}$$

The output of the carrier wave after FM was calculated as follows -

$$carrier = sin(2\pi f_0 t + I_m sin(2\pi f_m t)) \tag{3}$$

We implemented a simple exponential envelope using `np.exp()`. To make the envelope longer or shorter, we added an option to change the `decay_rate` of the exponential envelope. With the envelope implemented, the final output was calculated as follows:

```
envelope = np.exp(-decay_rate * t)

f0 = pretty_midi.note_number_to_hz(note.pitch)
fm = harmonic_ratio * f0
beta = mod_index

modulator = np.sin(2 * np.pi * fm * t)
carrier = np.sin(2 * np.pi * f0 * t + beta * modulator)

signal = carrier * envelope * (note.velocity / 127.0)
```

In order to make the sound more dynamic, we implemented a first-order IIR filter with key tracking to expose more high-frequency content for higher frequency notes. in addition this, we also used the same exponential envelope as the filter envelope. To have extensive control over the cutoff frequency of the filter, we added a `min_cutoff` parameter that controls the minimum cutoff frequency and a `max_cutoff_mul` parameter that is a multiplier for f_0 to control how much above carrier frequency the filter should open (in order to expose the higher partials created as a result of frequency modulation).

```
y = np.zeros_like(signal)
y_prev = 0.0
for i in range(len(signal)):
    fc = min_cutoff + envelope[i] * (f0 * max_cutoff_mul -
       min_cutoff)
    alpha = (2 * np.pi * fc / fs) / (2 * np.pi * fc / fs + 1)
    y[i] = y_prev + alpha * (signal[i] - y_prev)
    y_prev = y[i]
```

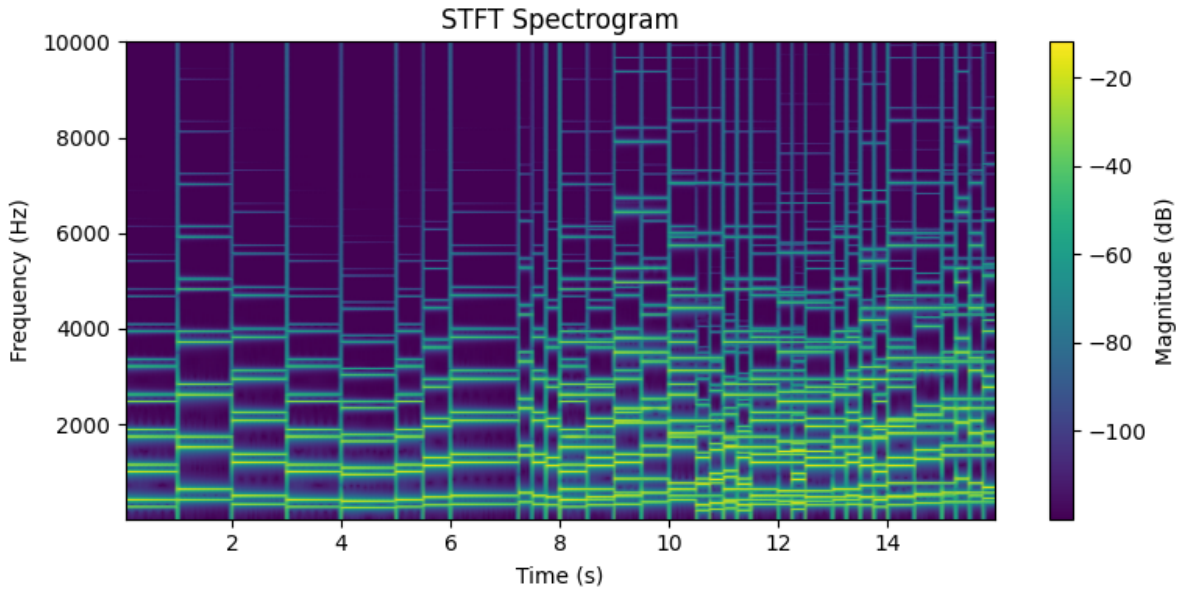


Figure 1: STFT of FM Synthesized Output - Pre Filter

(Figure 1) represents the STFT of the FM synthesized output signal **pre-filter**. Compared to (Figure 2), which is the output of the FM synthesized signal **post-filter**, we can see that the filter cuts the higher frequency content in the lower notes, but maintains the high frequency content in the higher notes, which are situated around the 8 second mark in the composition.

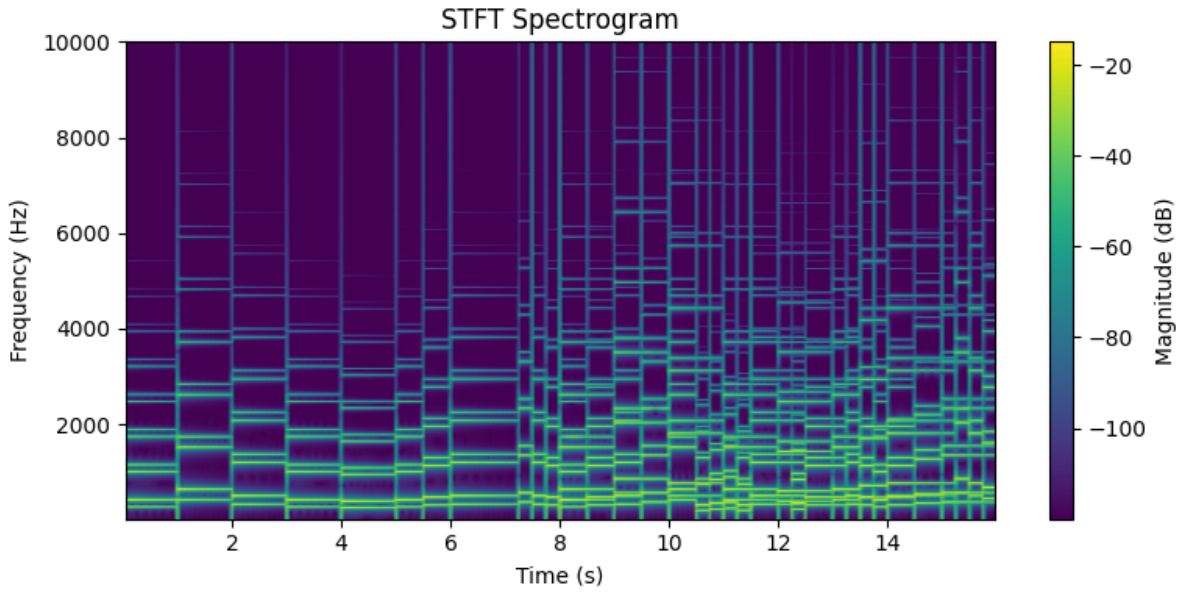


Figure 2: STFT of FM Synthesized Output - Post Filter

The frequency v.s. magnitude response of the one-pole dynamic filter is represented in (Figure 3)

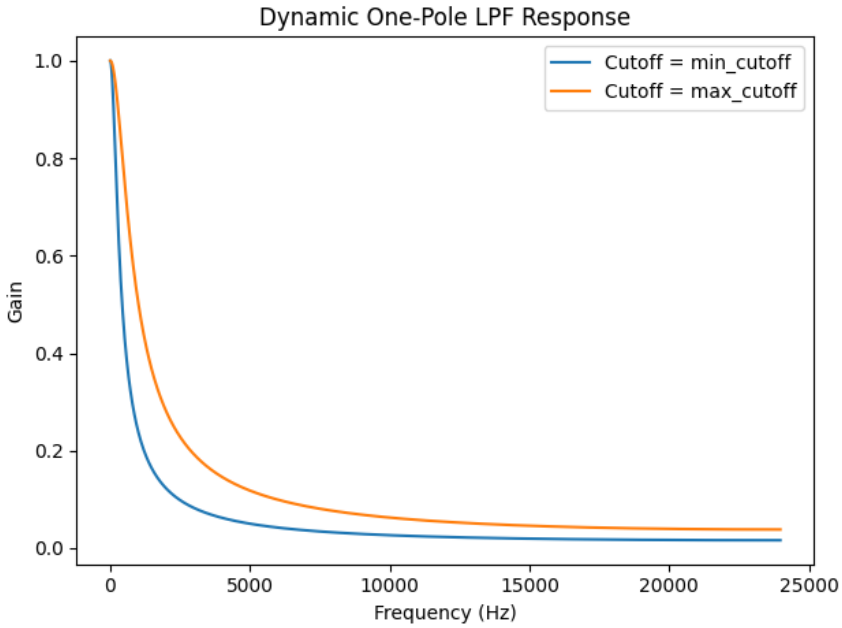


Figure 3: Frequency Response of the Dynamic One-Pole Filter

We can also take a look at the frequency content of first notes and is building step. We can see from (Figure 4b) and (Figure 4c) that the signal does get affected by the filter (loses frequencies at around 2000 Hz and above) but when we apply the envelope to the filter and the signal, we recover some of that higer frequency content, as shown in (Figure 4d).

1.2.3 Waveguide Synthesis

For waveguide synthesis we defined a class called `waveGuide` that mimics the movement of a wave on a double bounded string, to and fro from the bridge (left) and the nut (right). We set two delay lines `x_L` and `x_R` denoting left and right movement of the string along with a loss function `g()` consisting of a moving average filter at the boundaries.

The buffer simply moves from left to right through the loss function and that is done through the `np.roll()` function to push the buffer towards the right every sample.

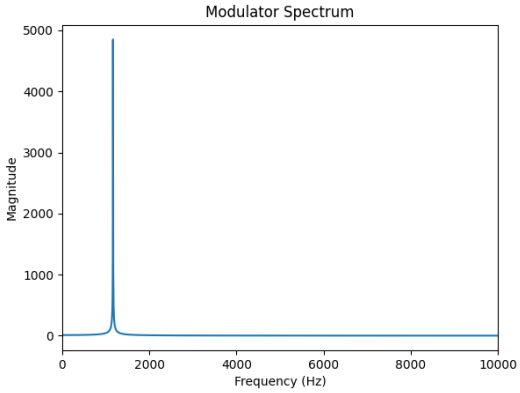
The excitation function used in this case is a simple triangular pluck function with maximum y-directional displacement at a place that can be defined while calling the synthesis function.

To synthesize the audio, we defined a `synthesize_midi()` function that takes in the MIDI file to be synthesized, relative position of the pluck and pickup position (in fraction of the length of the buffer).

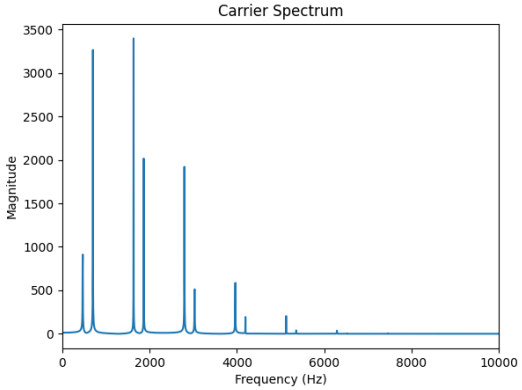
The base frequency of each note (extracted from `pretty_midi` is used to define the buffer size by the formula

$$L = f_s/2f \tag{4}$$

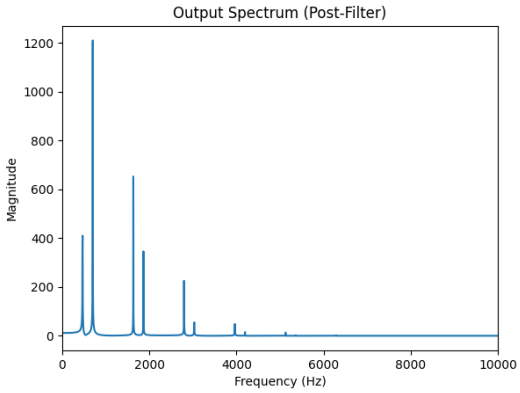
where f is the pitch of each note and f_s is the sampling rate. Since $1/f$ would define the time period of this frequency, we can multiply it by the sampling rate and double it to get the total number of samples required for the left plus right traveling buffers.



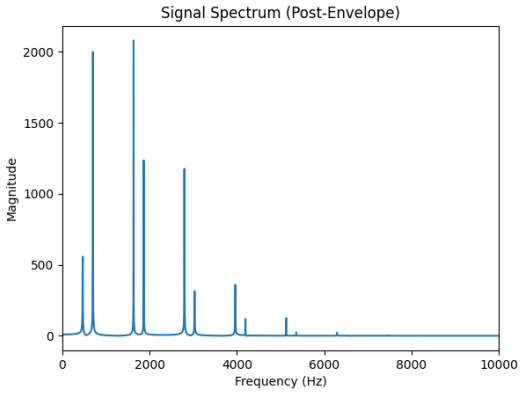
(a) Frequency Spectrum of Modulator Signal



(b) Carrier Signal Frequency Spectrum Pre-Filter and Pre-Envelope



(c) Carrier Signal Frequency Spectrum Post-Filter and Pre-Envelope



(d) Carrier Signal Frequency Spectrum Post-Filter and Post-Envelope

Figure 4: FFT spectrum of first note of the MIDI sequence at different stages

To model a more physical behavior of how higher pitched notes decay faster than lower pitched notes, we made our own loss function: -

$$loss\ factor = base\ loss\ factor - (pitch\ of\ note/127) * loss\ variation \tag{5}$$

We defined `pickup_ratio` and `pluck_ratio` to input position of the pickup and the pluck position as the fraction of the total length of the left traveling buffer. This makes it easier to do the relativistic calculation according to each note of the MIDI file and its length.

The `base_loss_factor` can be defined to always have certain amount of loss in energy (also defined as `loss_factor` in the `waveGuide` class). The `loss_variation` factor adds an additional feature to scale the overall loss factor by the pitch of the note. Higher the note, lesser is the `loss_factor`, hence more is the loss in the energy at the bounds, hence mimicking the higher loss of higher frequencies at the bounds in real physical world.

Another important factor to be taken into account was the duration of the note. If the note duration was lesser than the length of the buffer (calculation formula above), the output was clamped between the starting index of the note to the end index of the note. The ampplitude of

the note was also taken into account by the virtue of note velocity and converting it into a scaling factor.

```
start_sample = int(note.start * fs)
end_sample = int(note.end * fs)
note_duration_samples = end_sample - start_sample

amplitude = note.velocity / 127.0

# Generate audio on per note basis
note_buffer = np.zeros(note_duration_samples)
for j in range(note_duration_samples):
    note_buffer[j] = wg.wavePropagation() * amplitude

# Add to the output buffer
if start_sample + note_duration_samples <= len(output):
    output[
        start_sample : start_sample + note_duration_samples
    ] += note_buffer
```

The STFT of the output signal of waveguide synthesis shows us the sharp transients of note onsets and the dense harmonic content generated by the waveguide synthesis through its reflective delay-line interactions.

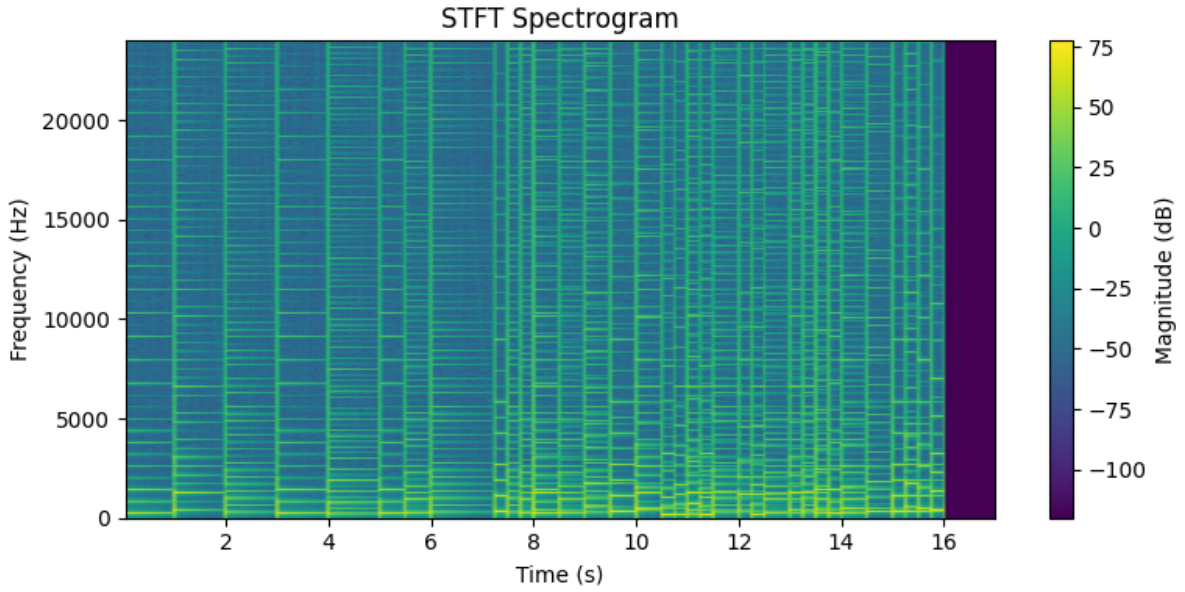


Figure 5: Spectrum of Physical Modeling synthesis using waveguide synthesis technique

1.2.4 Schroeder's reverb algorithm

To make the sounds more natural and to make them sound as if they are coming from a natural space, we designed and implemented a reverb based on Schroeder's reverb algorithm.

The algorithm consists of 4 arrays of gains and delay values for all pass filters arranged in series and comb filters arranged in parallel. The input signal is passed through the all-pass filter to introduce regular frequency dependent phase shifts and the output through the all-pass filters

is fed into a parallel arrangement of multiple comb filters which adds irregular phase shifts, which closely mimics real spaces. A small part of the original non-delayed signal is also mixed with the delayed signal to give a dry and wet signal balance.

The `SchroederReverb` class that we defined accepts delay and gain arrays for all pass filters and the comb filters along with the input signal and sampling rate.