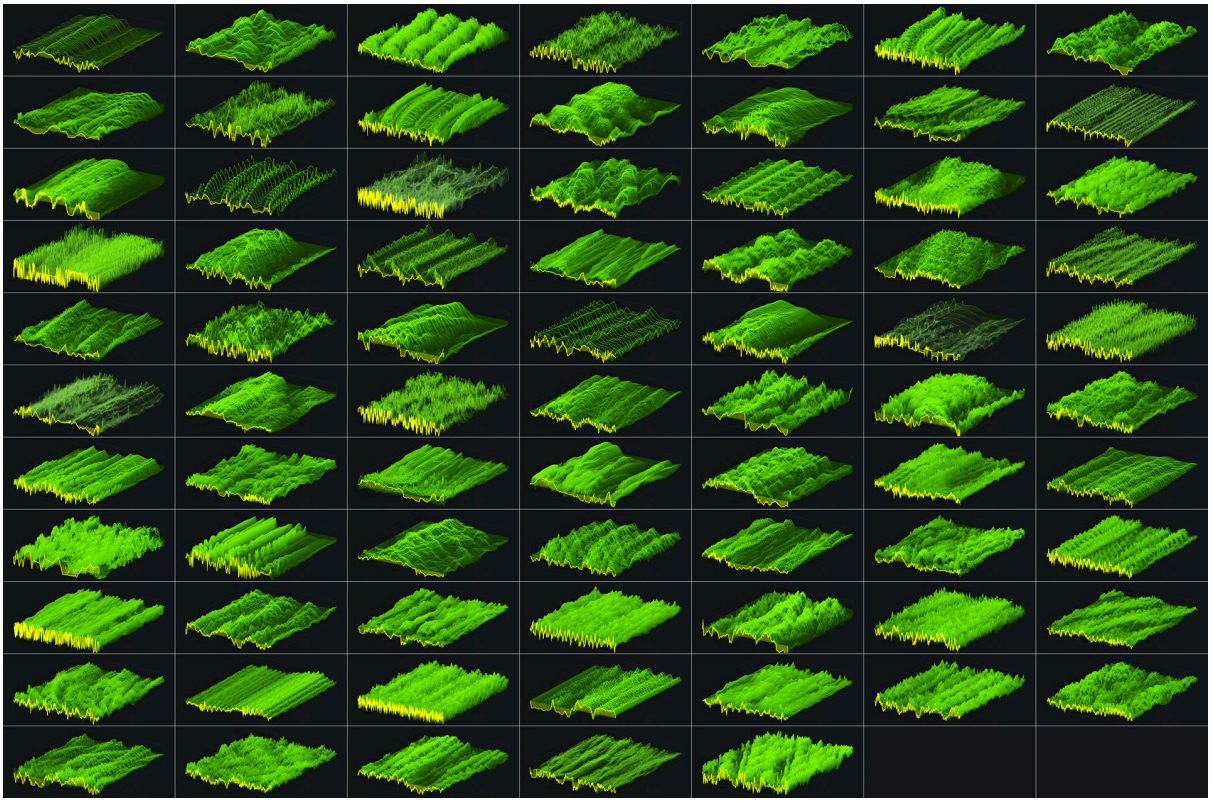
wave

Edward Wang
CS489

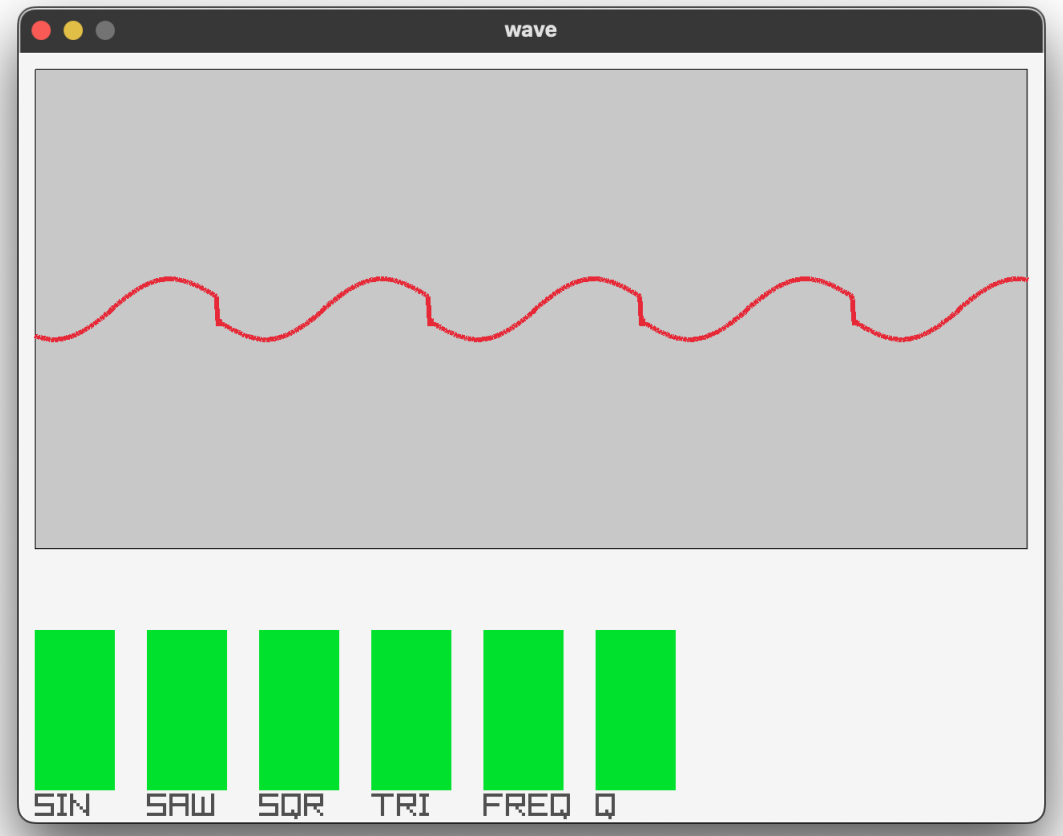


Introduction

For my CS489 project, I implemented a compact wavetable synthesizer from scratch. The synth is written in C and is optimized for performance, making it suitable for embedded systems and real time environments. To support real-world instrument tones, I wrote a Python script that converts one-shot .wav files into wavetables. This process includes normalization, frequency detection, and waveform extraction. It allows shrinking large audio files down to small binary tables while maintaining many sonic characteristics.

The structure of this report is as follows:

- The first section explains the basic principles of wavetable synthesis
- The next section is on the Python resampler script.
- After, I walk through a case study of recreating **Trumpet.wav** from class.
    - I have graphs comparing the original waveform to the reconstructed version
    - I will discuss how I reduced the file size by **95x**!!!
- Later, I show an experiment I did on aliasing sine() waveforms at various table resolutions.
    - I will plot many graphs, and include audio samples in the appendix.
- Lastly, I will discuss improvements.



Listing 0: The wave synthesizer

Wavetable Synthesis Principles

```c
#define TABLE_SIZE 1024

typedef struct {
	double phase;     // current phase (in table index units)
	double phase_inc; // phase increment per sample
	int wt_index;     // index into the shared wavetable array
} Osc;

typedef struct {
	float *data;      // float[1024] of samples
	Waveform type;    // enum: sine, tri, CUSTOM, etc.
} Wavetable;
```

Listing 1. Data representation

How does wavetable synthesis work?

- A wavetable is an array of discrete samples that represent one period of a waveform.

1. Each oscillator uses a floating-point phase value to determine its current position within the array.
2. Since the phase is usually not an exact integer, the oscillator performs linear interpolation between the two closest sample points to calculate a smooth output value.
3. After the sample is produced, the phase is increased by the “phase increment” and wraps around the wavetable.

```c
float wavetable_makesound(Osc *osc, Wavetable *wt) {
	double pos = osc->phase;
	// 1. index into table
	int index0 = (int)pos;
	int index1 = (index0 + 1) % len;
	double frac = pos - index0;
	// 2. this is the linear interpolation
	float sample = (float)((1.0 - frac) * wt->data[index0] + frac *
wt->data[index1]);
	// multiply by intended amplitude
	sample *= state->wt_levels[osc->wt_index];
	// 3. update phase
	osc->phase += osc->phase_inc;
	if (osc->phase >= TABLE_SIZE)
	osc->phase -= TABLE_SIZE;

	return sample;
}
```

Listing 2: Sound generation

What is the phase increment?

- The increment is directly related to the desired frequency of an oscillator:
  - A higher phase increment makes the phase move through the wavetable more quickly, resulting in a higher frequency output.
  - A lower increment produces a lower frequency.
- How is it calculated?

```c
#define SAMPLE_RATE 48000
void Osc_set_freq(Osc *osc, double freq) {
    osc->phase_inc = (TABLE_SIZE * freq) / SAMPLE_RATE;
}
```

Listing 3: Phase increment

Resampler Principles

The resampler is a python script that generates a wavetable based on an audio file.
How does it work?

1. Cleanup the audio
   a. Firstly, we normalise and trim silence from the file.
2. Fundamental frequency detection
   a. We use librosa to estimate the fundamental frequency.
   b. Improve detection by only considering voiced segments, and taking the median of all results
3. Cycle extraction
   a. We calculate the period length (in samples) based on the detected frequency.
   b. We then extract a single cycle that corresponds to 1 period of the waveform

```python
def extract_wavetable(audio_file):
      y, sr = librosa.load(audio_file, sr=None, mono=True)
      # 1. Cleanup
      y = normalize_audio(y)
      y_trimmed, _ = librosa.effects.trim(y, top_db=20)
      # 2. Detect fundamental frequency
      f0, voiced_flag, voiced_prob = librosa.pyin(y_trimmed)
      f0 = f0[voiced_flag]
      detected_freq = np.median(f0)
      # 3. Calculate period length
      period_length = sr / detected_freq
      period_samples = int(round(period_length))
      # 4. Extract cycle
      center = len(y_trimmed) // 2
      start = center - period_samples // 2
      end = start + period_samples
      wavetable = y_trimmed[start:end]
      return wavetable
```

Listing 4: Wavetable extraction

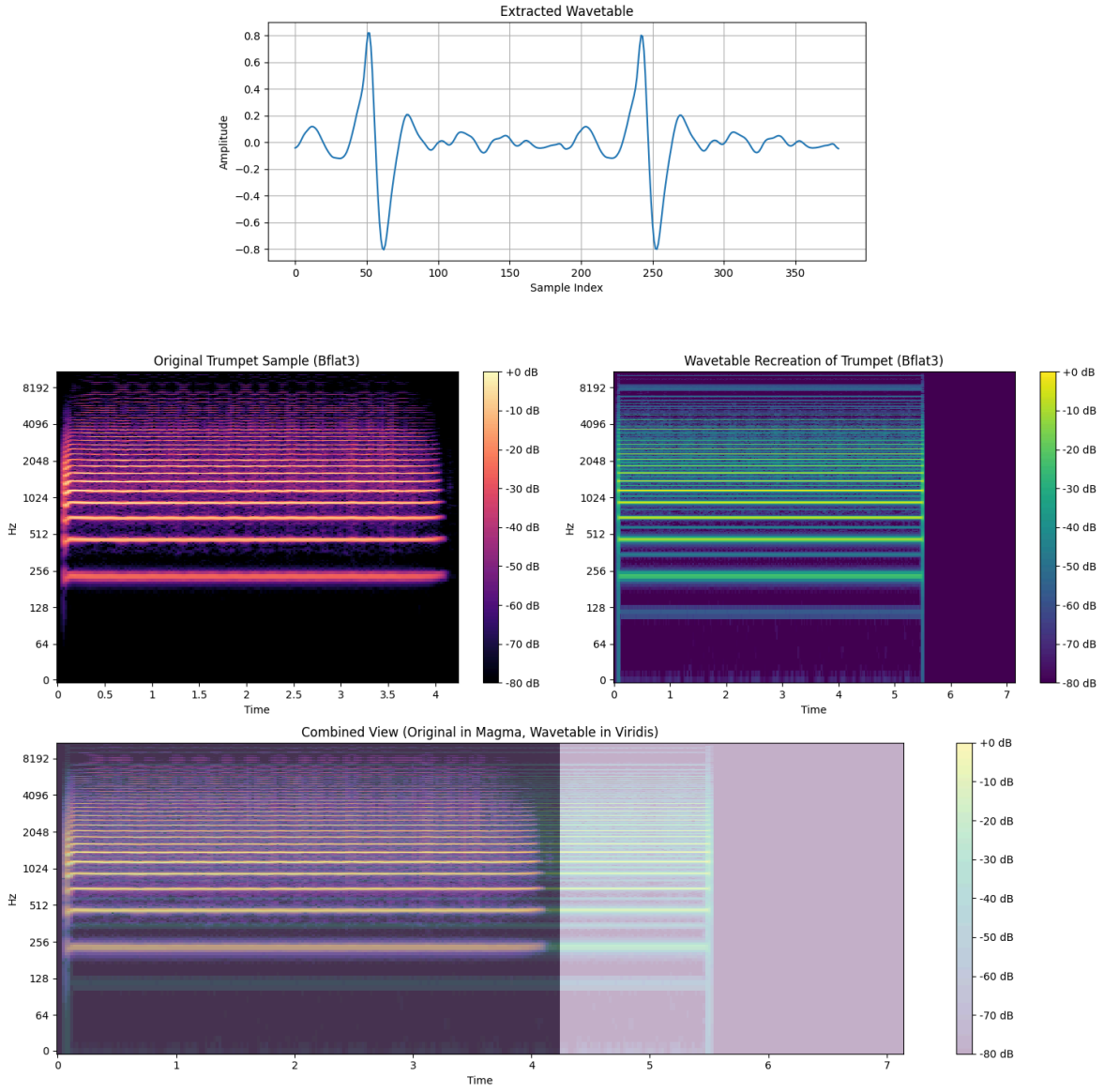
Trumpet.wav vs. Trumpet_wavetable.wav

To test my synthesizer, I wanted to try to recreate the Trumpet.wav sound we were given in the assignments.

My process:

1. Run resampler.py on Trumpet.wav
    a. This produces a binary “Trumpet.bin” of float32s.
    b. The format is:
        i. First 4 bytes: Length of the array (n)
        ii. Next N*4 bytes: Samples[0:1023]
2. Read the binary into the wavetable synthesizer
3. Play the corresponding note (B flat) and record it using OBS (a screen recorder).

Here are my results:

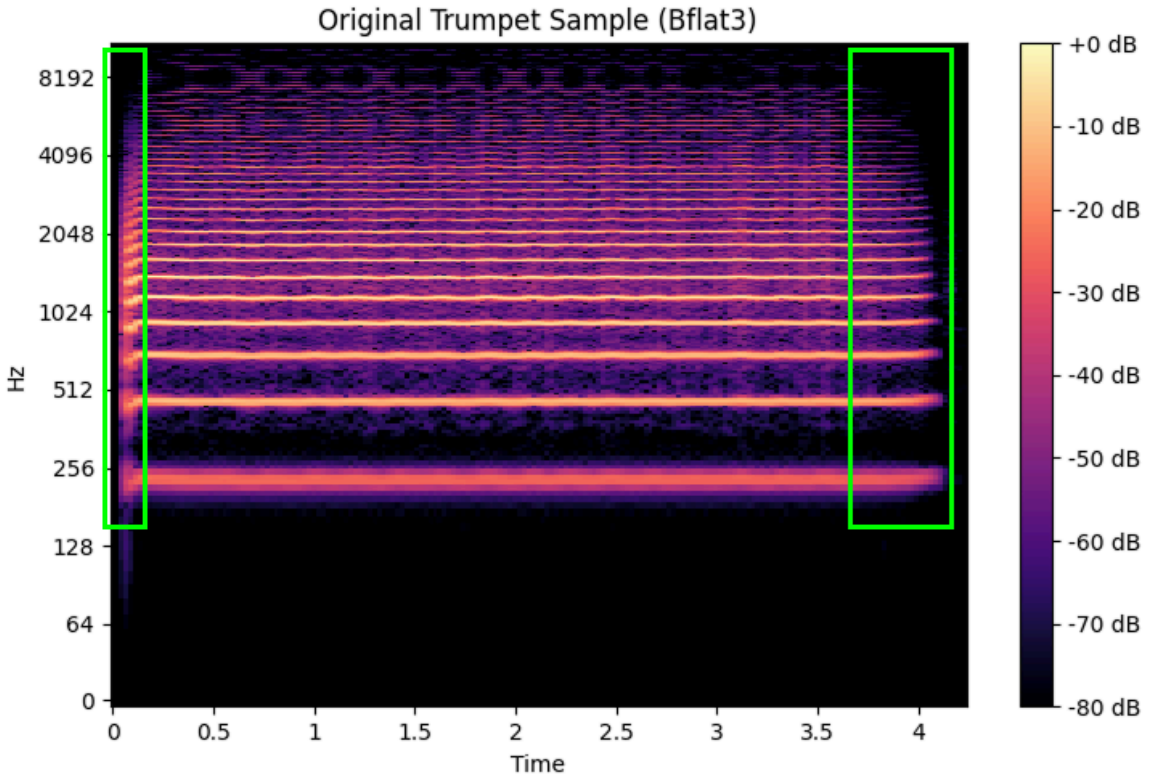


Listing 5: Trumpet.wav Recreation

1a. Differences in timbre (original)

There are several differences between the original and recreated sound.
Notably, the start and end behaviour is noticeably different.

We see a frequency sweep at the beginning of Trumpet.wav (left side).



Listing 6: Frequency sweeps

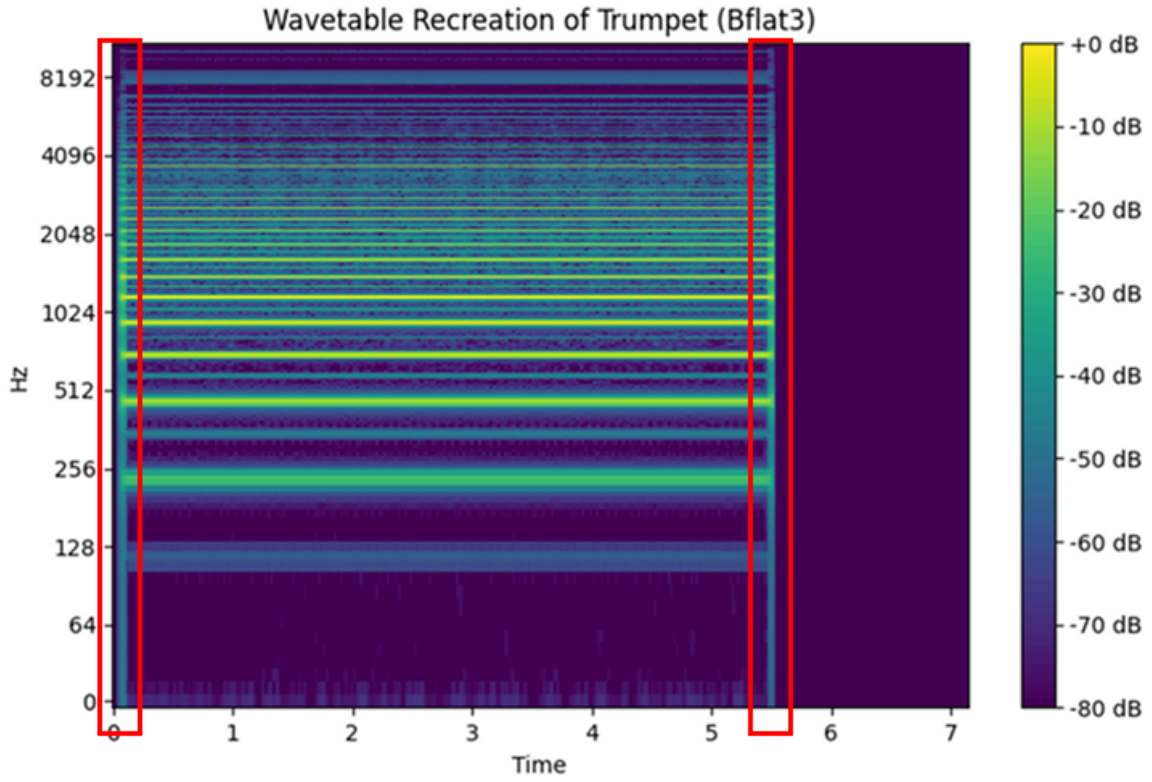
We can see that the lower frequencies arrive before the higher ones, which creates the “opening-up” kind of timbre. This corresponds to the “wah” sound of the trumpet.

- We see the opposite behaviour on the release of the note (right side).

1b. Differences in timbre (recreated)

In the wavetable recreation, such a sweep is not found.
This is because the synthesizer is not able to modulate the frequency response over time.
It simply repeats the same 1024 samples for the entire recording.

In an improved version of the synth, the “wah” behaviour could be achieved by using a filter with the cutoff frequency modulated by an ADSR envelope.



Listing 7: Hard edges

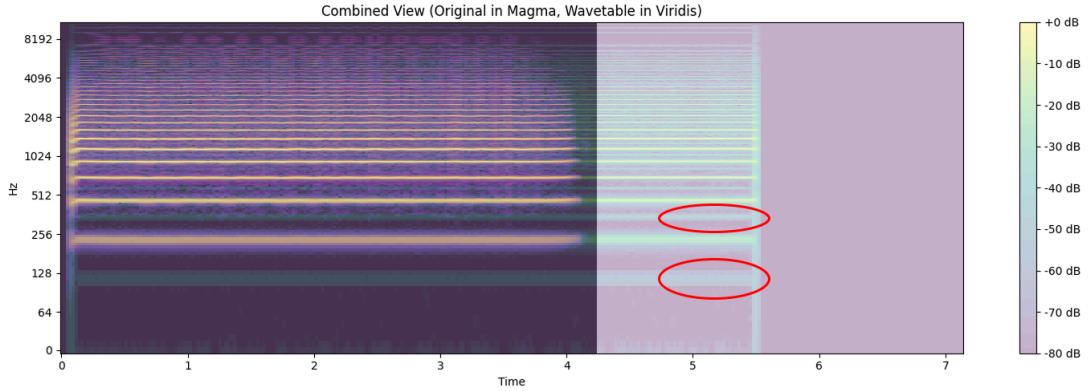
There is also a “click” effect found at the beginning and end of the recording.
I theorize this is because the synthesizer does not have an ADSR function.

- It is very very unlikely that at a random point in time the phase will be positioned so that wt->[data] == 0.
- So when the key is unpressed, the sound instantly jumps (non-zero) to 0. This discontinuity creates the click.

The click does not exist in the original trumpet recording because it is an analog instrument, and the sound will continuously propagate throughout the pipes after the blowing has stopped.

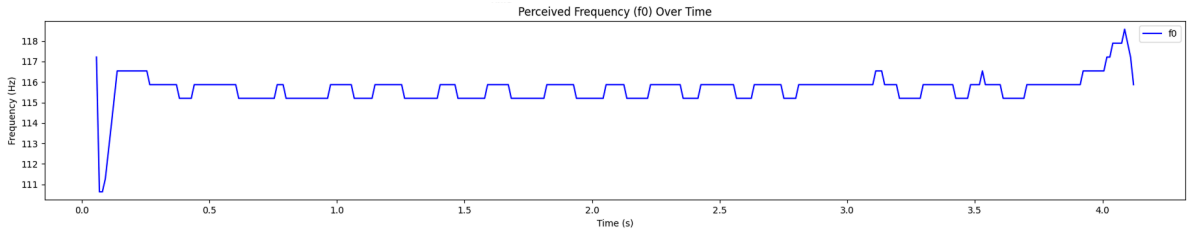
2. Differences in harmonics

I also noticed that there exist harmonics found in the recreation that did not exist in the original.



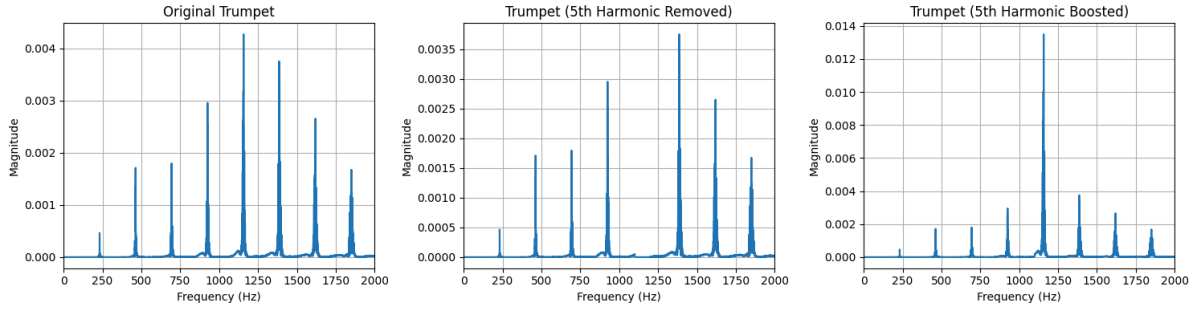
Listing 8: Harmonics

I think this is due to my resampler script not correctly identifying the fundamental frequency.



Listing 8: f0 estimate

The output of the script was A♯2 (115.87 Hz). But in a previous assignment I found the fundamental frequency to be around 240hz.



Listing 9: My work from A3

This makes sense. If the script is identifying the fundamental frequency as half the correct amount, it will generate double the harmonics. The extra ones will interleave the correct ones.

3. File size reduction

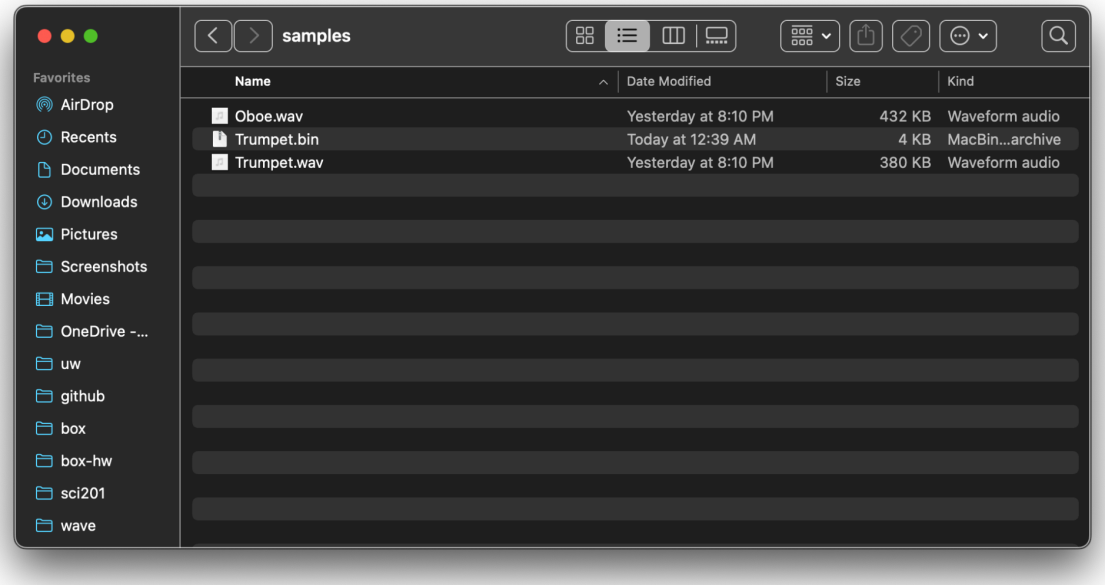
All in all, the two recordings sound pretty similar. I will attach them in the zip so that you can have a listen for yourself.

- Trumpet.wav (original)
- Trumpet_wavetable.wav (recreated)

That said, I was able to reduce the filesize from 380KB down to 4KB.
This is a 95x space reduction!!!!

I think this is very significant. Looking at the graphs, much of the original sound is preserved. Given a few more modulation options (like ADSR on a lowpass filter), we could recreate even more characteristics of the trumpet without increasing the file size!

Additionally, we could reduce the binary resolution from float32 to int16, and get it even smaller: 2KB! (but I didn't test how good the sound quality would be)



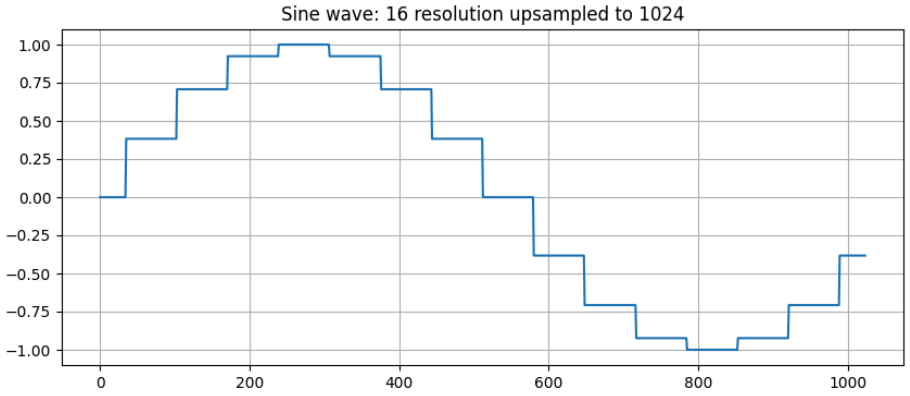
Listing 10: File size comparison

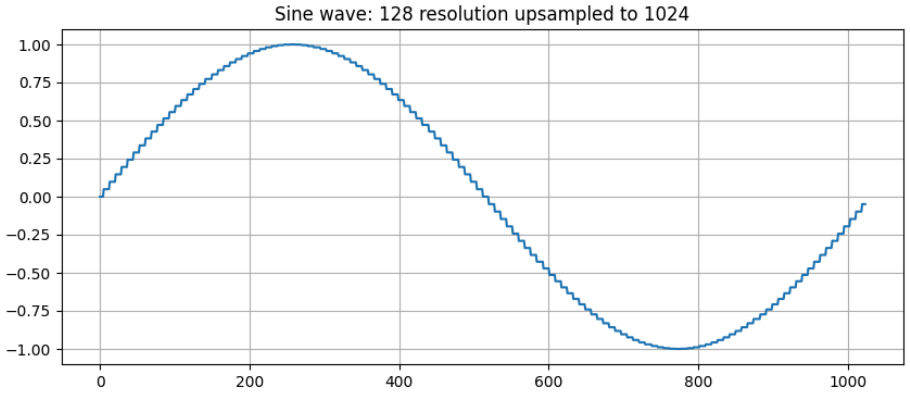
Sine Aliasing

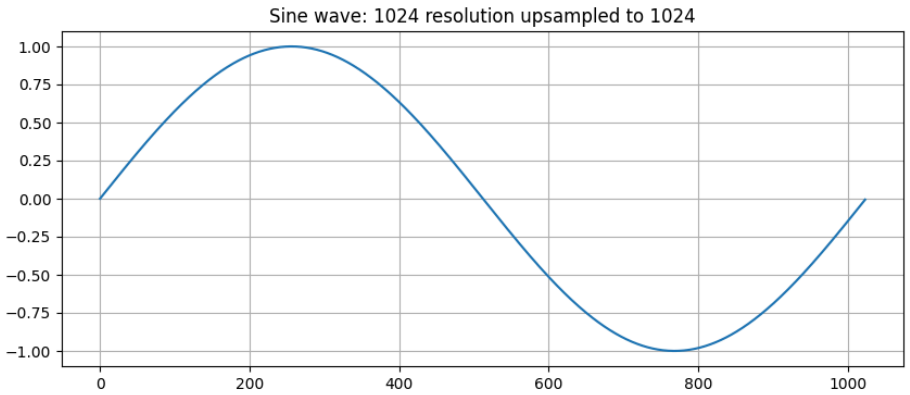
The main thing that drew me to wavetable synthesis is how it represented continuous waves. Thus, I wanted to see how rendering a sine wave at different resolutions impacted the sound.

I did this by forcing the signal into a smaller table size, then doing nearest-neighbour approximation back up to 1024, since the synth only takes in 1024 length binaries.
I rendered sine() at 16, 32, 64, 128, and 1024 table size.

- I also tried 61, to see if a prime number would cause an interesting effect

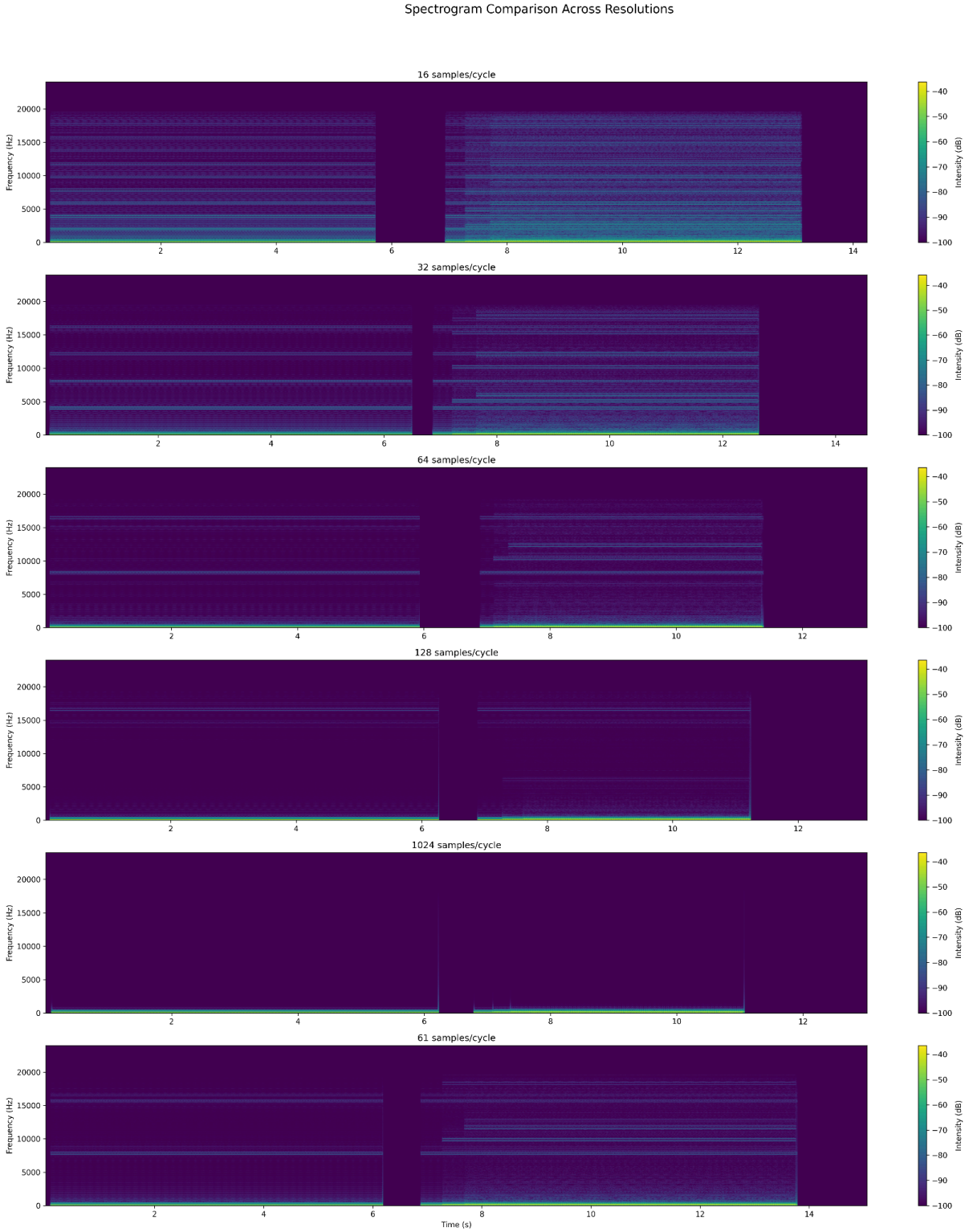






Spectrogram

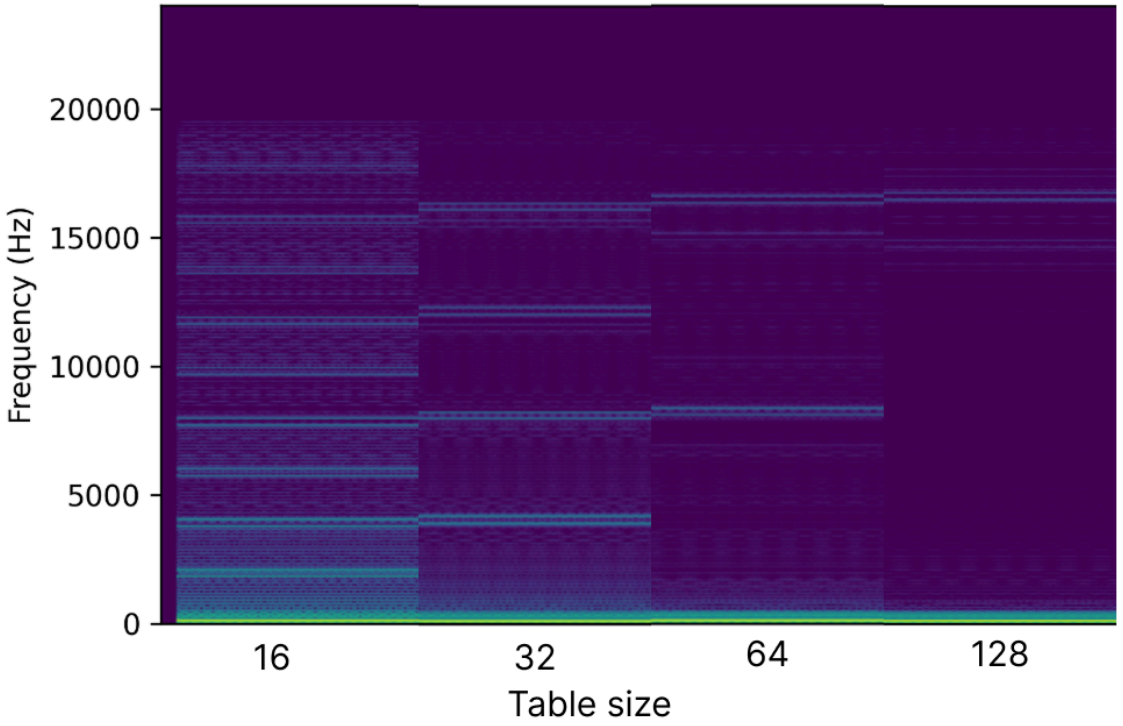
I also plotted a spectrogram.
I have included an audacity project file that will allow you to listen to the files themselves.
The pattern is: one single note (C3), followed by one major chord (C3, E3, G3)



Listing 11: Sine Spectrogram

Aliased Harmonics

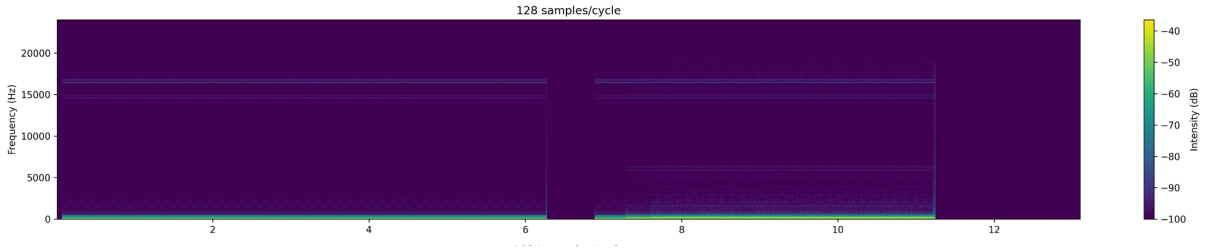
I noticed that at low table sizes, there are many extra harmonics introduced.
We can also see that at every doubling of the table size, the number of harmonics is halved.



Listing 12: Sine harmonics

Why does this happen?

As we learned in class, any periodic waveform can be expressed as a sum of sine and cosines. A perfect sine wave has only one frequency component: the fundamental. This is shown in the spectrogram for N=1024:



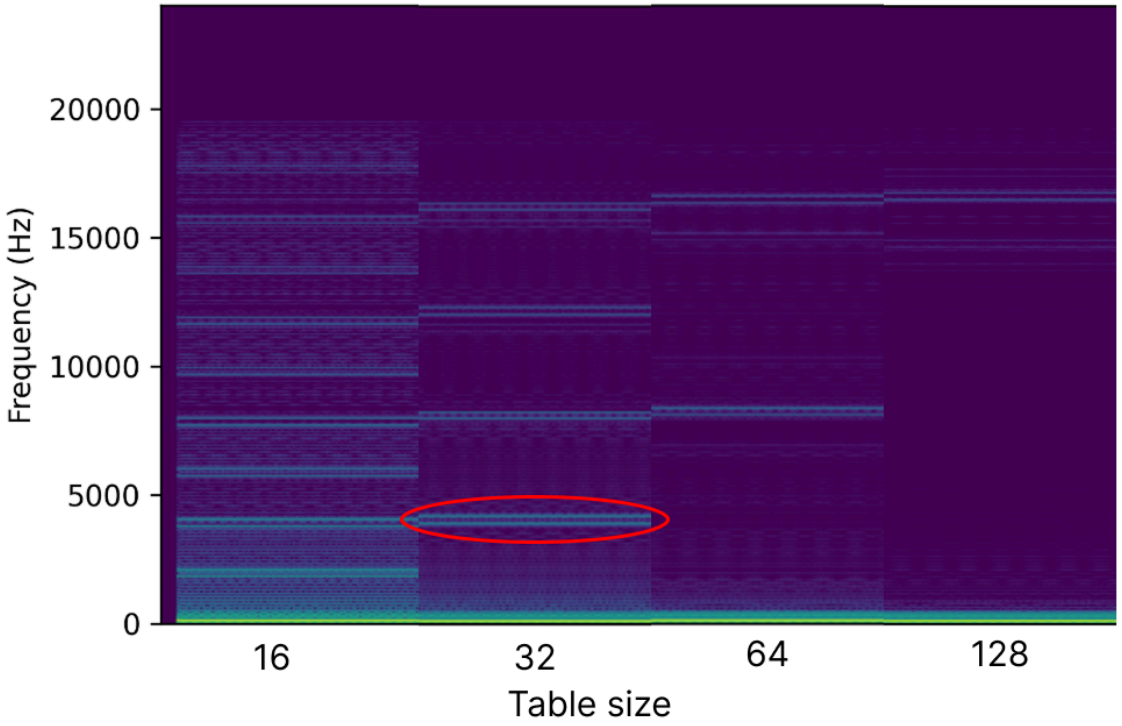
Listing 13: Sine with table size 1024

But when we approximate a sine using a small number of samples, we are creating a waveform that deviates from a perfect sine.

The Fourier representation of this approximation includes not just the fundamental, but also additional harmonics needed to recreate the jagged shape. As we increase the number of samples per cycle, the waveform gets smoother and closer to a true sine, so its Fourier series requires fewer harmonic terms.

Sidebanding

I also noticed the presence of double lines. This is interesting because I recorded these files while only playing one note.



Listing 14: Sidebanding

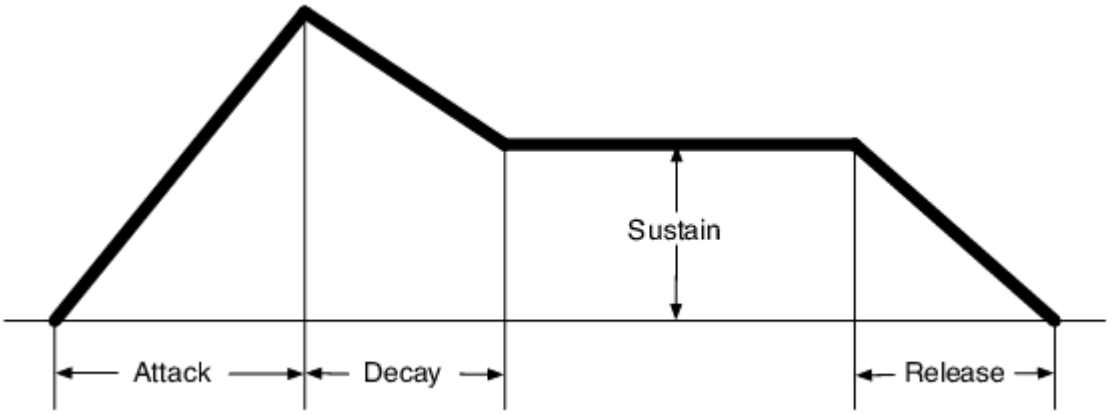
To be honest, I don't know why this happens.
My best guess is that it is due to the discontinuities creating an artifact that is higher than the Nyquist frequency (24,000 hz in this case), which then wraps around to the other side?

If you have a better explanation, please email me: e239wang@uwaterloo.ca :)

Improvements

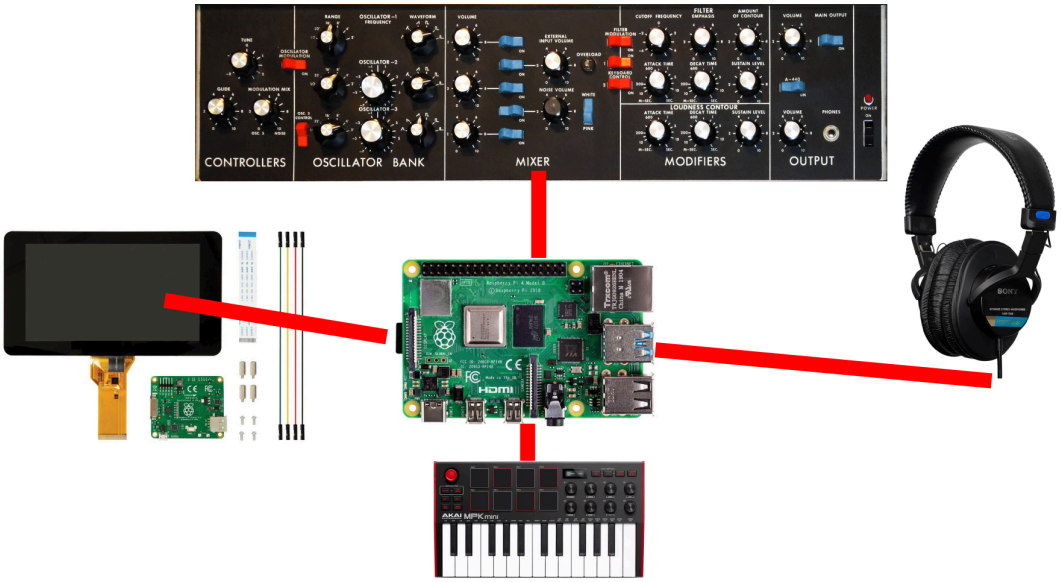
How could I make the synth better?

- Add more modulation options, like an ADSR envelope or a Low Frequency Oscillator



Listing 15: ADSR

- Have modulation between different signals
    - I.e. ADSR modulating Lowpass Filter Cutoff Frequency
    - As discussed before, this would allow us to represent more sounds, like Trumpet.wav
- Have a hardware interface



- I had this plan to build a whole setup for it, but it didn't pan out

Conclusion

I am satisfied with the project I created.
I learned a lot about DSP, C programming, and matplotlib.
Thank you for a great term.

Best, Edward