EE 367 Lab 2

Music Generation

Report by:
Ethan Vosburg (evosburg@calpoly.edu)
Isaac Lake (ilake@calpoly.edu)

February 6, 2024

Table of Contents

1 Identify specific requirements for note frequencies and duration

1.1 Part 1:

For part 1 we will be converting provided tables that include notes of a song into usable frequencies and durations to program a song in MatLab

1.2 Notes:

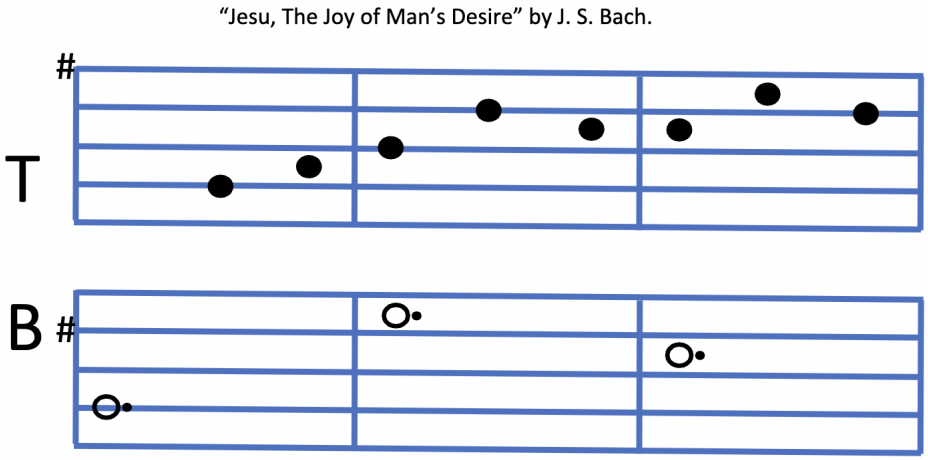

Figure 1: Notes for Analysis

T		G4	A4	B4	D5	C5	C5	E5	D5
B	G2			G3			E3		

Table 1: Table 1a

1.3 Note Numbers:

T		47	49	51	54	52	52	56	54
B	23			35			32		

Table 2: Table 1b

1.4 Calculated Frequencies and Beat Duration:

1. Calculate the frequency for each note (Hz).

$$f_n = 440 * 2^{\frac{n-49}{12}}$$

T		392	440	494	587	523	523	659	587
B	98			196			165		

Table 3: Table 1c

2. Specify the duration of each note (in beats).

T		1	1	1	1	1	1	1	1
B	3			3			3		

Table 4: Table 1d

2 Identify specific requirements for envelope and harmonics

2.1 Part 2:

In part 2, we will make our notes sound better by adding harmonics and creating an envelope for our music with a constant σ that determines the envelope decay rate.

1. Describe your method of determining the value of sigma for the envelope. What value did you choose?

   σ exists as a type of time constant in our case. Given it takes roughly 5 time constants before you can consider an RC circuit completely charged, I will use that as a baseline. Thus $\frac{t}{\tau} \approx 5$. Given a beat lasts $.285s$ our τ should be $\tau \approx \frac{.285}{5} = .057s$. Converting this τ into our preferred σ is as simple as multiplying τ by our sampling rate 16000, $\sigma = \tau * 16000 = .052 * 16000 = 912$. Thus we can safely conclude that we should use a σ of $912 samples$.
2. Describe your method of generating harmonics. Include any relevant parameters.

   A harmonic is an integer multiple of a given frequency. To generate harmonics we will run a simple for loop that will take in the provided frequency of a note and add a couple harmonics to it by multiplying by i.

3 Generate Music in Python

3.1 Part 3:

In part 3, we will finally put our preparation into practice and generate the sound using Python and MatLab.

1. Wav file of our music.
   We have uploaded the Wav file to Canvas.

4 Comparison to the real world

4.1 Part 4:

In part 4, we will take the spectrogram and music we created digitally and compare it to the spectrogram of the recorded harpists' music potentially giving us insight into the inner workings of a harp sound.

1. Spectrogram of our music:

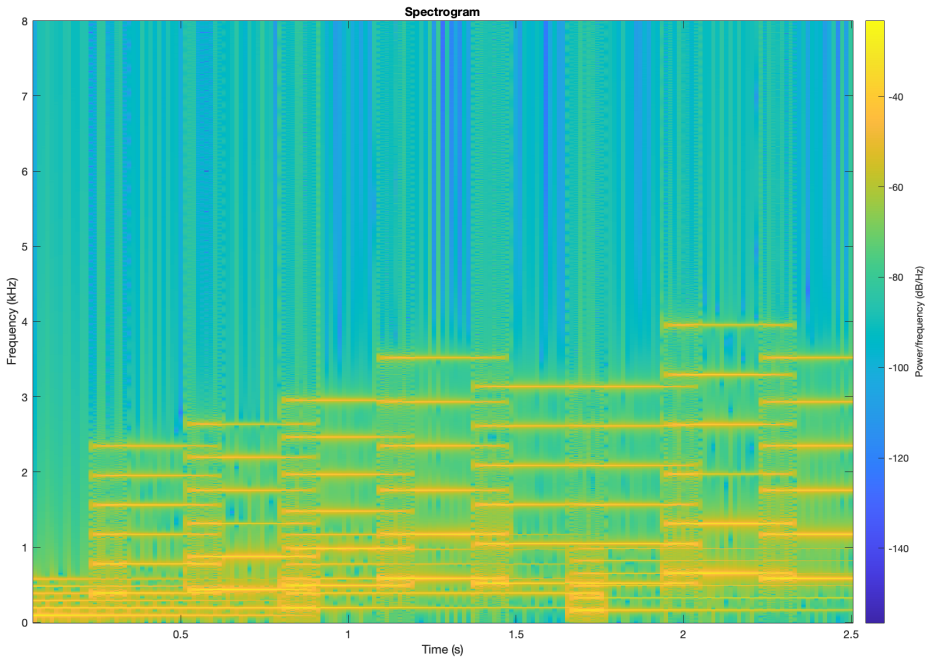

Figure 2: Spectrogram of the song generated by out Python script

2. Describe differences in the synthetic and harp spectrograms. Compare similarities or differences in envelopes or harmonics.

   There are noticeable differences between the music generated by our program and the real music played on a harp. One of the biggest differences is the lack of "fullness" in the generated music. The sound of a real harp is determined by the construction of the instrument, which creates complex tones with a large number of harmonics and decay. These tones are difficult to recreate in a program. Although we were able to match the fundamental frequencies of the harp, the program cannot fully replicate the complex tones of a real harp.

   Another difference is the timing. A harpist can slow down or speed up the tempo to add emphasis to the notes, which adds a layer of human variance to the performance. In contrast, our program can produce a perfectly timed piece of music. While this may be technically impressive, the natural human variances in timing add a layer of pleasure that is missing from our generated music.

   You can see all of these effects in the spectrographs as one of the synthesized music is much cleaner and void of variation. The digital nature of the music shows through as there are jagged edges and perfect lines. This is in stark contrast to the spectrograph of the real music where you can see the complexity of the signal increase. Beyond the fundamental frequency and the associated harmonics, you also see that some other sub-frequencies come in to play and mix with the music. This leads to a similar-looking spectrograph when you squint your eye but as you start to look closer at the signal you see the complexities of the sound show though.

3. Include sections of your add_note() function here, with comments, as well as sections of your program that call add_note()

   Below are snippets from the master Python file that show how the notes are added to the song, how the frequencies are made, and how the `add\_note` function was modified to generate the decaying sound. Note that we changed the σ from 912 to 5912 to make the decay less intense and smooth out the sound.

Listing 1: Python for getting the frequency of a midi note

```
def getMidiFreq(note):
    """This function returns the frequency of a midi note

    Args:
        note (int): midi defined note

    Returns:
        int: integer value of the frequency of the note
    """
    # Using the formula that was provided in the lab manual to convert midi note
    # to frequency
    return int(440 * 2 ** ((note - 49) / 12))
```

Listing 2: Python for modified add note routine

```
def add_note(xlist,amp,w0,nstart,nlen,harmonics):
   # initializing the decay to zero per sample added to the music
   decay = 0
   # iterate over the samples and harmonics adding each sample to the list
   for n in range(nstart,nstart+nlen):
      # iterate the decay as the samples progress
      decay += 1
      for harmonic in range(harmonics):
         if harmonic == 1:
            xlist[n] +=  (amp / (harmonic + 1)) * math.sin((harmonic + 1) * w0 * n)
         else:
            xlist[n] += (math.exp(-decay/5912)) * (amp / (harmonic + 1)) * math.sin((harmonic +
                 1) * w0 * n)
   return
```

Listing 3: Python for adding notes

```
   # define the total number of samples in relation to the sample rate and
   # beat count for the song
   total_num_samples = int(9 * 0.285 * 16000)
   print("Total Number of Samples:", total_num_samples)

   # allocate list of zeros to store an empty signal
   xlist = [0] * total_num_samples

   # define the notes that need to be played
   noteArray = [
      23, 0, 3,
      35, 3, 6,
      32, 6, 9,
      47, 1, 2,
      49, 2, 3,
      51, 3, 4,
      54, 4, 5,
      52, 5, 6,
      52, 6, 7,
      56, 7, 8,
      54, 8, 9
   ]

   # iterate over the notes and place them in to the song
   for i in range(int(len(noteArray)/3)):
      # define w1 with the midi note and the sample rate
      w1 = 2 * math.pi * getMidiFreq(noteArray[3 * i])/ 16000
      amp = 10000
      n_start = int(noteArray[3 * i + 1] * 0.285 * 16000)
      n_durr = int(noteArray[3 * i + 2] * 0.285 * 16000) - n_start

      # add note to the over all array
      add_note(xlist,amp,w1,n_start,n_durr, 6)
```