COMM 401: Signal & System Theory

Lab Project Report

Name: Mathew Hany Bestawy

ID: 52-21647

Tutorial: T16

Milestone 1 Steps:

1. We import all libraries we will use in our project.
    1.1. **numpy**: to do all mathematical calculations on the signals.
    1.2. **matplotlib**.**pyplot**: to plot the signals as graphs.
    1.3. **sounddevice**: to play the signal on the speakers.
2. We store frequencies of piano notes in variables so we can access them by their names instead of writing the numbers everywhere. We also make a variable **rest** so that any time we want to have a silent period in the song we will use it to make a constant signal with amplitude 0.
3. We store the song's duration in a variable (3 seconds)
4. We store the number of beats per minute in a variable, it will be used to control how fast the notes will be played. In our case, our song is played at 140 beats per minute.
5. We calculate how long a beat should be using the formula

$$Beat\ Duration = \frac{60\ Seconds}{Beats\ Per\ Minute}$$

6. We split the beat into smaller units so that we can have notes that have different uniform durations. We will have 3 units, quarter of a beat, half of a beat, and a whole beat. So a quarter note is a note that takes lasts for $\frac{Beat\ Duration}{4}$.
7. Using **np.linspace** function, we generate an array that has values starting from 0 to 3 (The song's duration), and we tell numpy to split this range into 12 * 1024 equal parts. So we have an array with 12 * 1024 values that are equally spaced from 0 to 3. This will be used as our x-axis (Time with 12*1024 samples).
8. We define a helper function called **note,** that given a frequency gives us a sine wave signal for that note. It works as following:

8.1. It uses **numpy** to generate 2 sine waves using the formulas $sin(2\pi ft)$ and $sin(2\pi Ft)$ where $F = 2f$. The idea is that in music multiplying a frequency by 2 gives the same note, just at a higher pitch, so for example $A3 = 220$ and $A4 = 440$.

8.2. Now we have the 2 sine waves of the same note at 2 different octaves. We can add them together using **numpy** to get a new wave that will represent our note and we return it.

9. Next we define our main function **generate_song**, that given any song will generate the signal for it. Here is how it works:

9.1. The song will be represented by an array of arrays. Each inner array contains 2 items. The first one is a note frequency, and the second item is a note duration. So each inner array represents a single note. And the outer array will contain these notes in order in which they will be played. For example **[[a, quarter], [c, half]]** represent a song that has 2 notes, A that will be played for quarter of a beat and C that will be played for half a beat.

9.2. We define an accumulator that starts at 0, where we will add our notes together so that at the end when we have added all notes the accumulator will become the song.

9.3. We define a variable to keep track of when the next note should be played. At the start it is set to 0 because the first note should be played at time 0.

9.4. We loop over the notes in the song and for each one:

9.4.1. Extract the frequency from index 0 and duration from index 1 and put them into variables for later use.

9.4.2. We create a pulse function that starts at the time where the note should start playing and ends at the time that the note should stop playing. The pulse is just a difference between 2 unit step functions

$u(t - t_i) - u(t - t_i - T_i)$. We use the variable that we defined for keeping track of when the next note should be played as the starting point of the pulse and we use that value + the duration of the note as the end point.

9.4.3. We update the variable that keeps track of when the next note should be played. We add to it the duration of the current note + a small time delay so that the next note won't be played directly after the current note. A delay of quarter beat / 4 works well for us.

9.4.4. We use the **note** helper method that we defined before and give it the frequency of the current note to construct the sine wave array representing the note. We then multiply that by the pulse that we have generated in step 9.4.2, so that we have a sine wave that starts at the required time and for the required duration. And we add that to our accumulator

9.5. After the loop, all notes should have been converted into sine waves using **note** helper method, multiplied by pulses so that they are played in the correct times and durations, and summed together in the accumulator. So our final song is ready in the accumulator, so we return it.

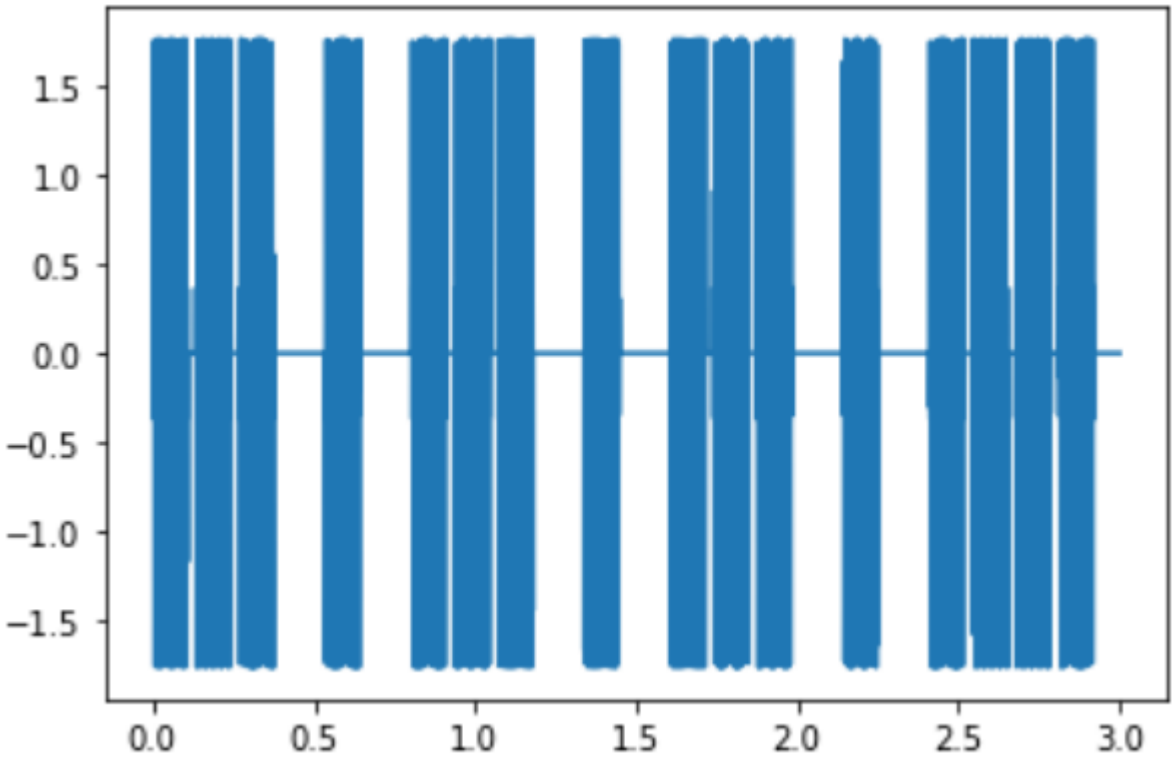
10. At the end we defined our song, we searched the internet for the piano notes for **The Pirates Of The Caribbean** song and their durations, and we put them in an array of arrays as we explained in point 9.1.

11. Finally we call the function **generate_song** on our song so that we get the signal of the song.

12. We can next plot the song's signal using **matplotlib.pyplot.plot** giving it the y-axis **t** and the y-axis (our generated signal for the song).

13. We can play the song on the speakers using **sounddevice.play** giving it **song's signal**, with sample rate of 3 * 1024.

Milestone 1 Plots



Milestone 2

Steps:

1. We import ***fft*** from ***scipy.fftpack*** to use it to convert our signals from time domain to frequency domain using fourier transform.
2. We define our frequency axis ranging from 0 to 512, and define our number of samples (3 * 1024) which is (song duration * 1024)
3. We convert our original song from time domain into frequency domain using the method **fft** from **scipy.**
4. We generate 2 random numbers from 0 to 512 that will be used as the 2 random frequencies that we will use as noise.
5. Using the 2 random frequencies we generate a sine wave for the noise signal.
6. We add the noise to our original song.
7. We convert the new song (with the added noise) from time domain to frequency domain using **fft** from **scipy**
8. Next we find the maximum frequency from the frequency domain representation of our original song.
9. We round that maximum frequency to the next integer so that we don't include the maximum frequency itself in the next step.
10. Next, in our new song (with added noise), we find any frequency that has a magnitude that is greater than the maximum magnitude of the frequencies of the original song (without noise). In particular, we are interested in the indices where these magnitudes happen. We use **np.where** to achieve that. These will be the indices of our 2 random noise that we have previously added.

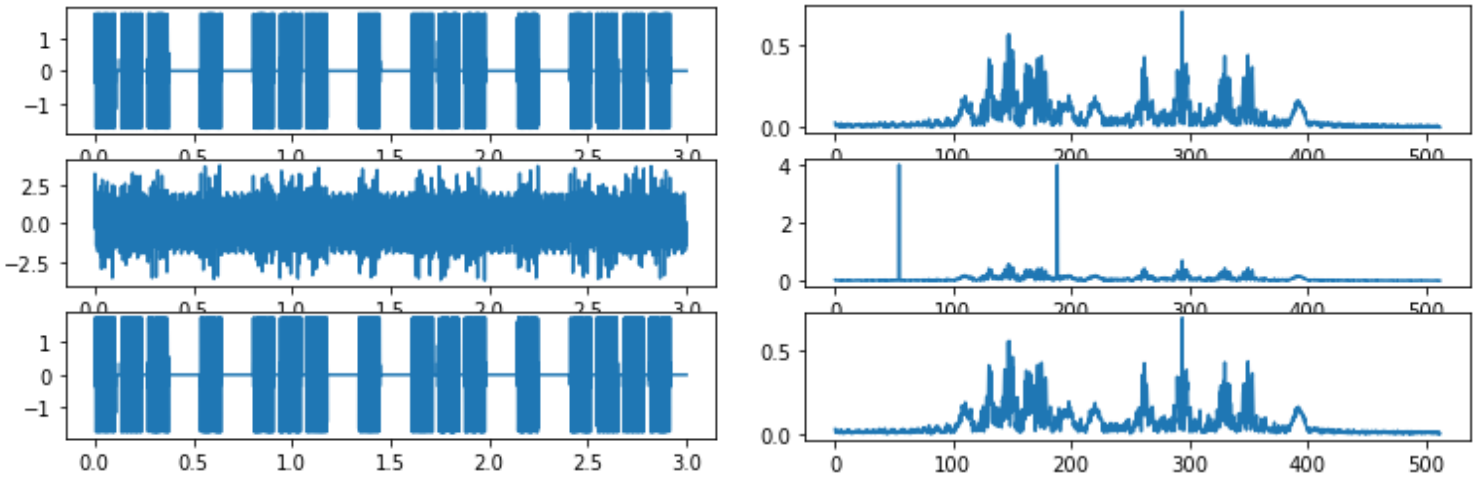
11. Next for each found index, we get the frequency corresponding to that index, we round it to an integer value (as our random frequencies were restricted to integer values). And we construct a sine wave for it. We accumulate all the noise frequencies sine waves into a single signal (**noise_cancel**), it will be the same as our original noise that we had added to our original song.

12. To remove the noise from the song, we just subtract that (**noise_cancel**) from the new song with the noise, we will get our original song back without any noise.

13. We use **matplotlib.pyplot** to plot the original song without noise, the new song with noise, and the new song after removing the noise. We do that for both time domain and frequency domain, so we end up with 6 plots in 2 figures.

14. We also play our song using **sd.play** to make sure it is the same as our original song without any noise.

Plots for Milestone 2:



On the left: Time domain for original song before noise, song after adding noise and song after filtering the added noise.
On the right: Frequency domain for the original song before noise, song after adding noise and song after filtering the added noise.