Std ID: 190041220

Name: Tasfia Tasneem Annesha

```
from IPython.display import Audio
import librosa
import librosa.display
from scipy.io.wavfile import write
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

```
plt.rcParams["figure.figsize"] = (14,10)
plt.rcParams.update({'font.size': 14})
```

Playing an Audio File

```
# choose a song of your liking (the more uncommon it is, the better), now play it using IPython.display's Audio Function
Audio("/content/happy-birthday-music-box.mp3")
```

0:00 / 0:19

Reading an Audio File into an array

Sample Rate - The sampling rate refers to the number of samples of audio recorded every second. The most common sample rate for digital audio is 44100 Hertz; this means that the CT sound-wave is sampled 44100 times every second.

Source: A [Mathematical] Analysis of Sample Rates and Audio Quality by Noah Sheridan

```
# read your music here # sr is the sample_rate
```

```
audio_array,sr = librosa.load("/content/happy-birthday-music-box.mp3") # ignore user warning
```

```
# clip the audio_array upto first 90000 samples
audio_array = audio_array[:90000]
```

```
# print the shape of audio_array
audio_array.shape
```

```
     (90000,)
```

```
# print the sampling rate sr
sr
```

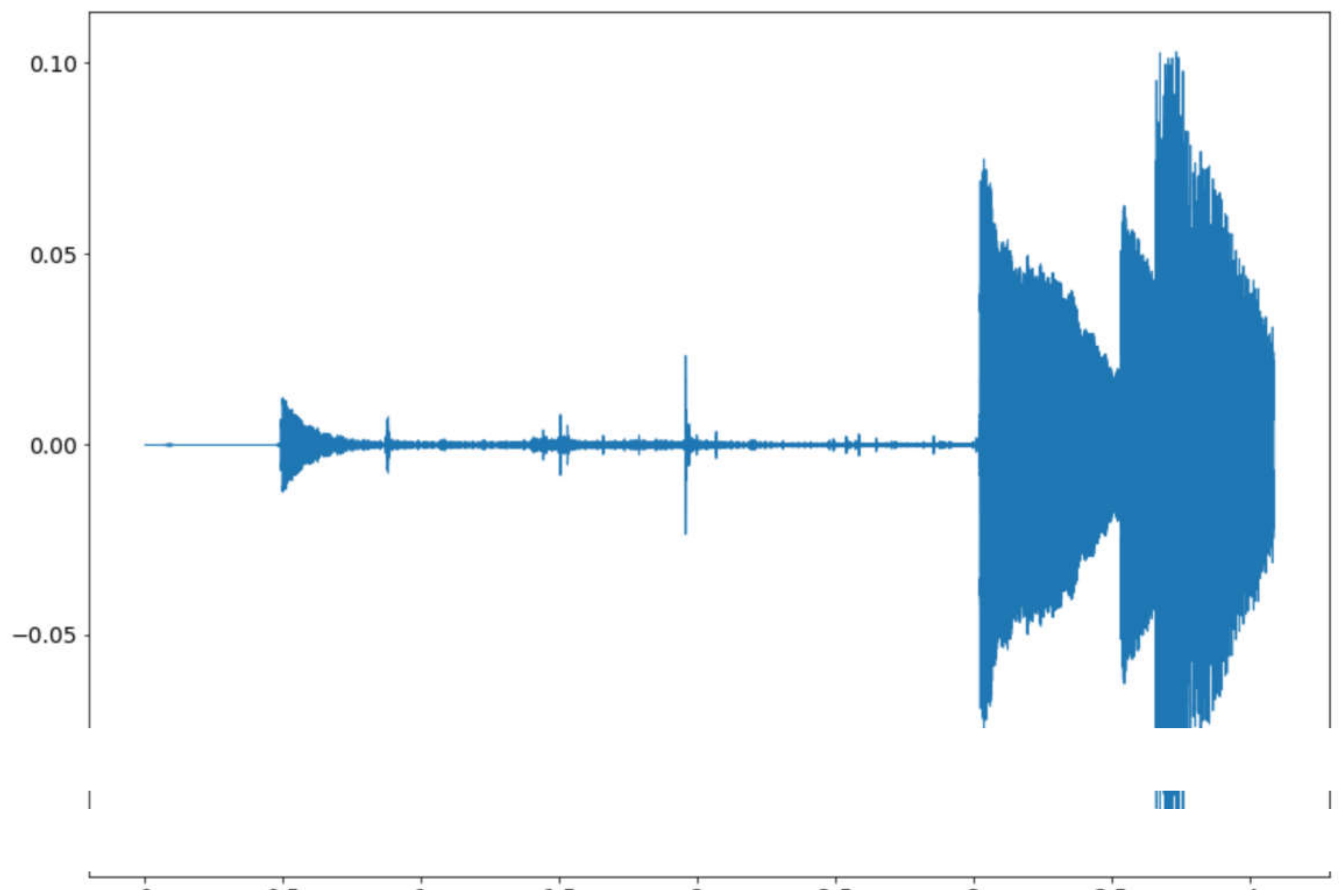
```
     22050
```

```
# play the clipped/truncated audio_array using Audio() function
# Audio(audio_array, rate=sampling_rate)

Audio(audio_array, rate=sr)
```

0:00 / 0:04

```
# plot the clipped/truncated audio_array using librosa.display.waveplot() function
import librosa.display
librosa.display.waveshow(audio_array, sr=sr);
```



Audio Manipulation

```
# In this part, you will implement the moving average filter as shown in the class

# make a function called moving_average( input_signal_I, neighborhood_size_N )
# it will output a signal with equal size of the input signal
# each output_signal sample will contain the average of that corresponding sample value of input signal and (N-1) samples before it.
# for the first N samples you will see that the neighborhood goes beyond the edges of input_signal, then consider those samples to be zero (jus
# don't implement it using any fancy automated library function. Just go over the signal and do some arithmatic calculation.
def moving_average(x, w):
    output = []
    x_cumsum = x.cumsum()
    output.append(x cumsum[0])
    output.append(x_cumsum[0])

    for i in range(1, len(x)):
      idx = max(i-w, 0)
      n = min(i+1, w)
      output.append((x_cumsum[i] - x_cumsum[idx])/n)
    return output

mov_avg_output = moving_average(audio_array, 1000)


# listen to the moving average output signal
Audio(mov_avg_output, rate=sr)
```

0:00 / 0:04

```
# now listen to the original signal again, can you hear any difference?
Audio(audio_array, rate=sr)
```

0:00 / 0:04

Explain what are the differences that you hear in between these two signals, does it match with the theory that was taught in the class?

Compared to the original, the moving average output is much quieter as higher frequency sample has decreased due to averaging and gives the impression that the music is coming from behind a surface. The low and somewhat restricted loudness makes sense since the moving average filter diminishes the clarity of the signal by reducing its smoothness.

```
# now we will implement the moving_avg using convolution. We will use np.convolve
# here, np.ones(w)/w is the kernel of the low pass filter (moving avg. filter)
# passing 'same' in the function means, I want to keep the garbage values (as shown in the class) on the both sides. If I passed 'valid', then

def moving_average_np_conv(x, w):
    return np.convolve(x, np.ones(w), 'same') / w

mov_avg_conv_output = moving_average_np_conv(audio_array,1000)

Audio(mov_avg_conv_output, rate=sr)
```

0:00 / 0:04

Creating an Audio File

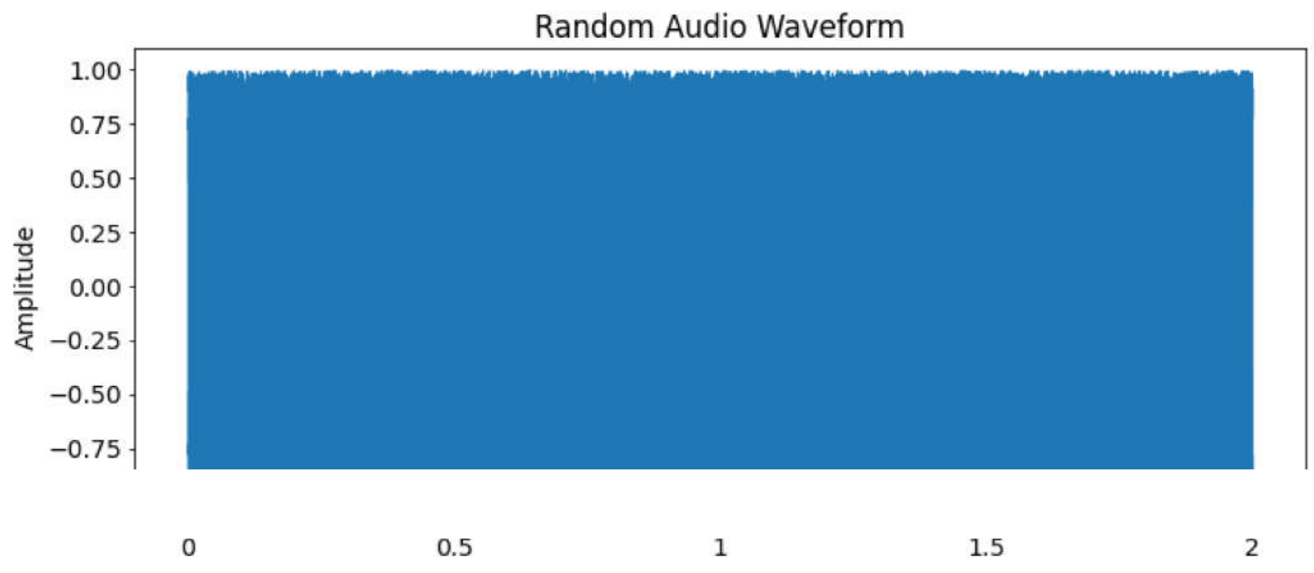
```
# first, let's create an audio of about 2 seconds with sample_rate = 22500
# use np.random.uniform to generate an array with (sample_rate*seconds) values. The values should be within -1 to 1, put it in the variable ran

# random_audio = ?
# print(radnom_audio.shape)
# now, let's play that audio
# Audio(random_audio, rate=22500)
sample_rate = 22500
seconds = 2
random_audio = np.random.uniform(size=sample_rate*seconds)
print(random_audio.shape)
# now, let's play that audio
Audio(random_audio, rate=22500)
```

```
(45000,)
```

0:00 / 0:02

```
# plot random_audio using librosa waveplot
plt.figure(figsize=(12, 5))
librosa.display.waveshow(random_audio, sr=sample_rate)
plt.title('Random Audio Waveform')
plt.xlabel('Time (seconds)')
plt.ylabel('Amplitude')
plt.show()
```



Creating audio from General Wave Equations

In this section, you will actually use the equations taught in the class to generate your own music. In the class, I showed you cosine waves, but sine waves can be used as well. For this task, we will be using sine.

Wave Function: $f(t) = (A)\ sin(2\pi\ Ft)$: where $A =$ Amplitude, $F =$frequency, $t =$ time

Source: How to Play Music Using Mathematics in Python by Nishu Jain

```
samplerate = 44100

def get_wave(freq, duration=0.5):
    '''
    Function takes the "frequecy" and "time_duration" for a wave
    as the input and returns a "numpy array" of values at all points
    in time
    '''
    amplitude = 4096
    # create an array t with values from 0 to duration. as the number of samples will be about (samplerate*duration). The values of t represent
    # now calculate wave function f(t) as shown in the text cell above, for each value in the array t
    # in other words, each value in the array t will give you an f(t). then put all the values of f(t) for each t in an array called 'wave'

    # this function can be written in two lines using numpy

    # your codes goes here
    t = np.linspace(0, duration, int(samplerate*duration))
    wave = amplitude * np.sin(2 * np.pi * freq * t)

    return wave
```

```
# now test your function get_wave()
# play it in low volume, some of the wave might be a little harsh to hear
# put these values in frequency [100, 500, 1500]
frequency = 1500 # change this frequency value and observe the effect
test_audio = get_wave(frequency, duration = 1.5)
Audio(test_audio, rate = samplerate)
```

0:01 / 0:01

```
frequency = 1000
test_audio = get_wave(frequency, duration = 1.5)
Audio(test_audio, rate = samplerate)
```

0:01 / 0:01

```
frequency = 1200 # change this frequency value and observe the effect
test_audio = get_wave(frequency, duration = 1.5)
Audio(test_audio, rate = samplerate)
```

0:00 / 0:01

Explain what are the differences that you hear in between the audios for different frequency values

The pitch of the song increases with increasing wave frequency. The sound is more harsher to the ear as we increase the frequency.

```
# read the given excel file using pandas
df = pd.read_excel('/content/note_frequencies.xlsx')
```

Source Data: [Frequencies for equal-tempered scale](#) by Bryan H. Suits

```
# this excel files contains the chords and their corresponding frequencies
df
```

	Note	Frequency (Hz)	Wavelength (cm)
0	C0	16.35	2109.89
1	C#0	17.32	1991.47
2	D0	18.35	1879.69
3	D#0	19.45	1774.20
4	E0	20.60	1674.62
...	...	...	...
103	G8	6271.93	5.50
104	G#8	6644.88	5.19
105	A8	7040.00	4.90
106	A#8	7458.62	4.63
107	B8	7902.13	4.37

108 rows × 3 columns

```
# So, If you know chords of a certain song, then you can find out the frequencies
# from the frequencies, you can generate short audio signals using get_wave() function, then merging/concatenating the audio signal for differe
```

```
def get_piano_notes():
    '''
    Returns a dict object for all the piano
    note's frequencies
    '''
    octave = df['Note'].to_list()
    freq = df['Frequency (Hz)'].to_list()

    note_freqs = {note:sound for note,sound in zip(octave,freq)}
    note_freqs[''] = 0.0

    return note_freqs

note_freqs = get_piano_notes()


def get_song_wave(music_notes):
    '''
    Function to concatenate all the waves (notes)
    '''
    note_freqs = get_piano_notes()
    song = [get_wave(note_freqs[note], duration=0.4) for note in music_notes.split('-')]
    song = np.concatenate(song)
    return song

# music_notes = 'F#3-F#4-A#4-C#5-D#5-F#5-F#4-C#5-G#4-F#4-F4-F#4-F#3-F#4-A#4-C#5-D#5-F#5-F#4-C#5-G#4'
# CHOOSE SOME NOTES FOR YOUR OWN MUSIC. TRY TO CREATE AN ORIGINAL MUSIC THAT YOU LIKE # An example is given above #
music_notes = 'A#3-D#4-A#4-G#4-D#5-F#5-C#4-C#5-G#4-F#4-F4-D#5-F#3-A#4-A#4-D#5-D#5-F#5-F#4-G#5-G#4-F#4-F4-D#4-D#3-D#4-A#4-A#5-D#5-F#5-F#4-C#5-G#
music = get_song_wave(music_notes)
music = music * (16300/np.max(music))

Audio(music, rate=samplerate)
# changing the 'duration', you can the speed of the music
# changing the music_notes, you can create audios for different songs
# this particular chords/song is from Avicii's Fade into the darkness
```

0:02 / 0:14

Help: How to Play Music Using Mathematics in Python by Nishu Jain

Save your music as my_music.wav

```
# save your music as a .wav file
write('my_music.wav', samplerate,music.astype(np.int16))
```

```
# play the saved music
Audio('/content/my_music.wav')
```

0:05 / 0:14

DFT and Spectral Analysis

Now you need to read the music you just created and do spectral analysis to find out the frequency composition.

In frequency composition, you should see peaks on the frequencies of the notes you took in the variable 'music_notes'.

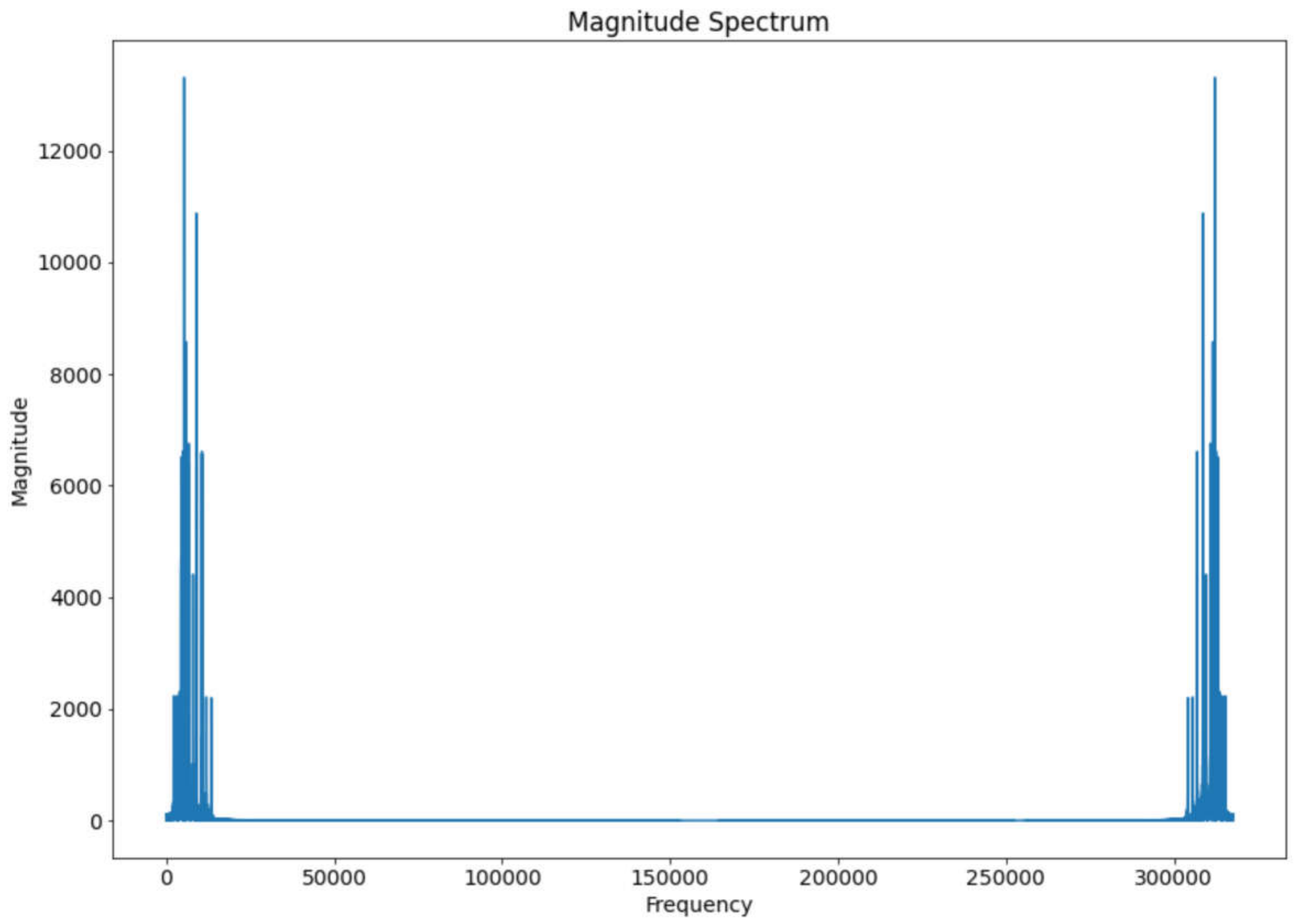
To implement DFT, you can either use numpy or your own implementations. Get the DFT is Polar format. Now, plot and analyze the magnitude array to see where the peaks are.

Then, convert the DFT into analog signal format, where the y axis is the values and xaxis is the analog frequencies, which ranges from 0 Hz to (SamplingRate/2) Hz.

Write down the frequencies of the peaks and match them with corresponding notes from the excel sheet to figure out what notes were played in the music.

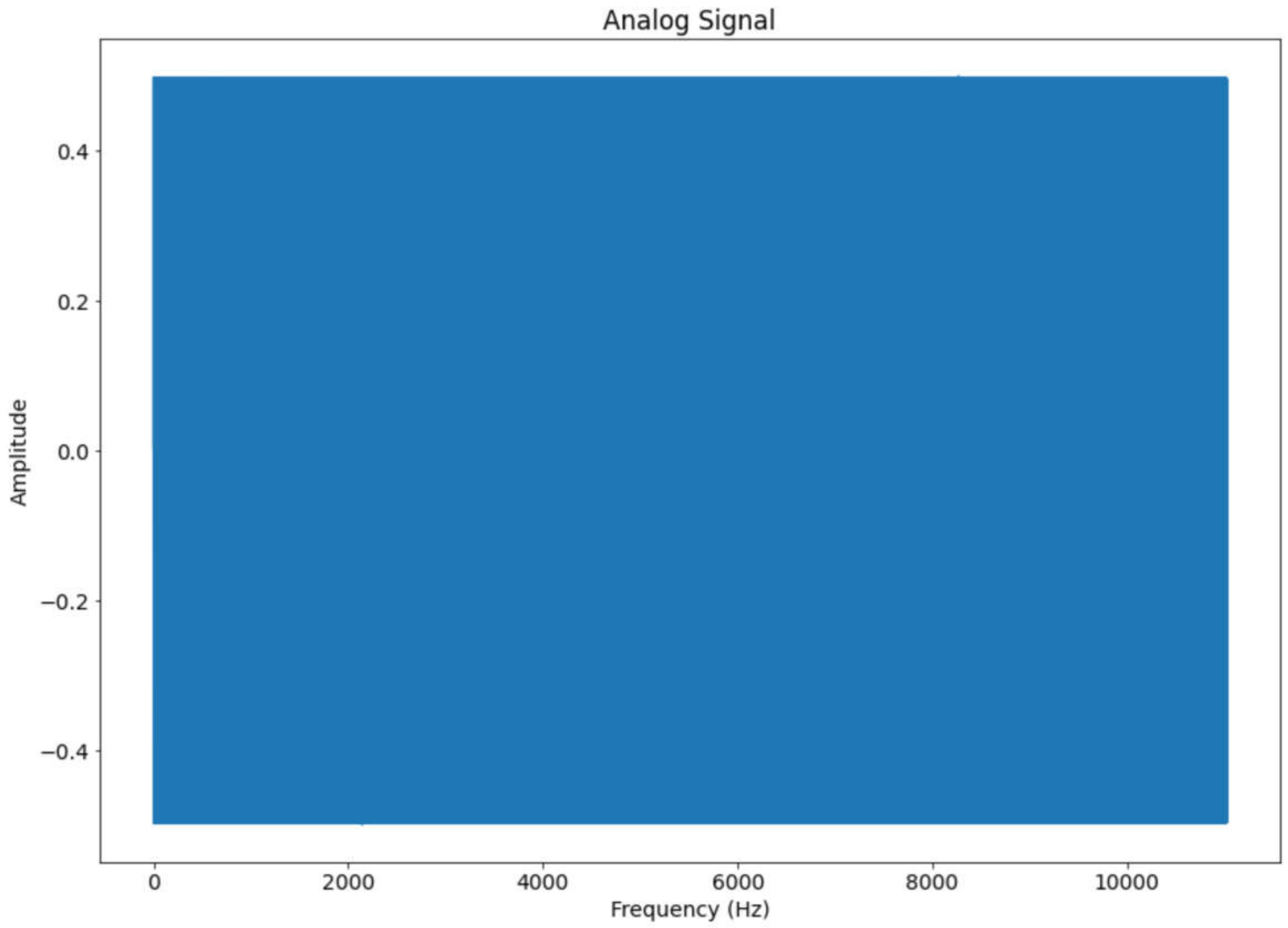
```
# your code goes here
music_array,sr = librosa.load("/content/my_music.wav")
dft = np.fft.fft(music_array)
mag = np.abs(dft)


plt.plot(mag)
plt.title('Magnitude Spectrum')
plt.xlabel('Frequency')
plt.ylabel('Magnitude')
plt.show()
```



```
analog_freqs = np.linspace(0, sr/2, len(mag)//2)
analog_dft = mag[:len(mag)//2]
analog_signal = analog_dft * np.exp(1j * np.angle(dft[:len(mag)//2]))
analog_signal = np.fft.ifft(analog_signal).real
```

```
plt.plot(analog_freqs, analog_signal)
plt.title('Analog Signal')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Amplitude')
plt.show()
```



```
print(analog_freqs)
```

```
     [0.00000000e+00 6.94448819e-02 1.38889764e-01 ... 1.10248611e+04
      1.10249306e+04 1.10250000e+04]
```

References and Additional Learning

Article

- Analysis of Sample Rates and Audio Quality by Noah Sheridan
- How to Play Music Using Mathematics in Python by Nishu Jain

Frequency Data

- Frequencies for equal-tempered scale by Bryan H. Suits

Video

- Mathematics of Music by Nishu Jain

Websites

- Jupyter Audio Basics from Notes on Music Information Retrieval
- librosa documentation
- Module display for IPython

Connect

- Feel free to connect with Adrian on YouTube, LinkedIn, Twitter and GitHub. Happy coding!

✓ 0s completed at 11:08 PM