Ryerson University

Department of Electrical, Computer, & Biomedical Engineering
Faculty of Engineering & Architectural Science

Course Title:	Signals and Systems II
Course Number:	ELE632
Semester/Year (e.g.F2016)	W2023

Instructor:	Dimitri Androutsos

Assignment/Lab Number:	Lab 5
Assignment/Lab Title:	Sampling and the Discrete Fourier Transform

Submission Date:	April 9, 2023
Due Date:	April 9, 2023

Student LAST Name	Student FIRST Name	Student Number	Section	Signature*
Saini	Harsanjam	501055402	10	Harsanjam

*By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a "0" on the work, an "F" in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at: http://www.ryerson.ca/senate/current/pol60.pdf

Introduction:

The down sampling of numerous discrete signals is the main topic of this lab assignment, with an audio signal as the primary focus. We seek to study the spectra of these signals after down sampling. We also investigate how zero-padding affects frequency resolution and the production of fresh data. Through experiments, we investigate situations where zero-padding could not be advantageous if the quantity of time domain samples is insufficient or falls short of accurately representing frequency components. Additionally, we create an audio equaliser to modify a sound in the frequency domain and analyse how it impacts a signal in the temporal domain. This lab report's goal is to give a thorough description of these approaches, the findings of the tests, and the conclusions drawn from the study.

A) Discrete Fourier Transform and Zero Padding (DTFT)

Python code for Part 1:

```python
import numpy as np
import matplotlib.pyplot as plt

# Define the signals
H = 0  # Replace with your student number
I = 1  # Replace with your student number
N = 10
n = np.arange(N)
x1 = np.exp(1j * 2 * np.pi * n * (10 * (H + 1)) / 100) + np.exp(1j * 2 * np.pi * n * 33 / 100)
x2 = np.cos(2 * np.pi * n * (10 * (H + 1)) / 100) + 0.5 * np.cos(2 * np.pi * n * (10 * (I + 1)) / 100)

# Compute the DFT magnitude of x1 and x2
X1 = np.abs(np.fft.fft(x1, N))
X2 = np.abs(np.fft.fft(x2, N))

# Plot the DFT magnitude spectra
fr = np.arange(-N/2, N/2) / N
plt.stem(fr, np.fft.fftshift(X1), use_line_collection=True)
plt.xlabel('Frequency (fr)')
plt.ylabel('Magnitude')
plt.title('DFT Magnitude of x1[n]')
plt.show()

plt.stem(fr, np.fft.fftshift(X2), use_line_collection=True)
plt.xlabel('Frequency (fr)')
plt.ylabel('Magnitude')
plt.title('DFT Magnitude of x2[n]')
plt.show()
```

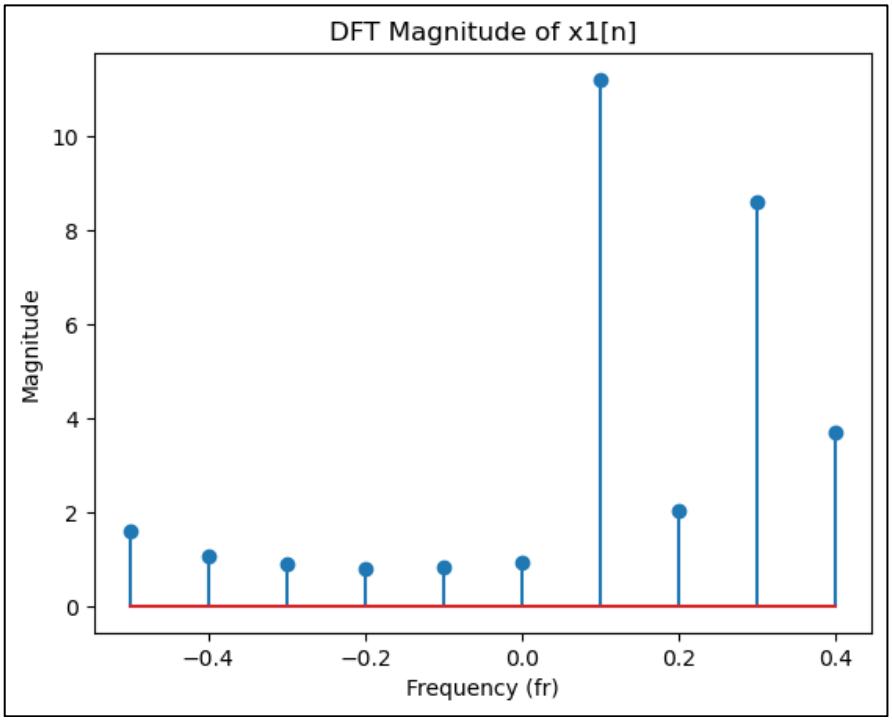

Figure 1: Plot for magnitude of X1[n] with respect to ***fr***

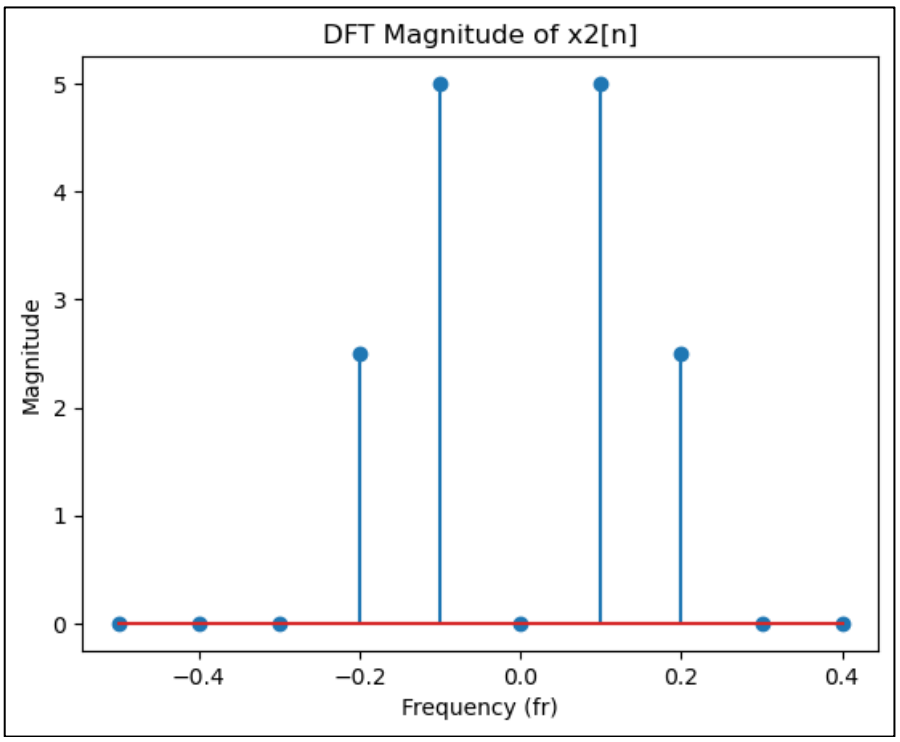

Figure 2: Plot for magnitude of X2[n] with respect to ***fr***

i) Due to the even function sum in contrast to the uneven sum of X1, X2 has a symmetrical signal.

ii) Since each plot has unique frequency components that pertain to the two signals, it is easy to differentiate the two signals.

iii) The DFT plot of x1 [n] shows additional frequency components because it is the combination of two signals with various frequencies.

Python code for Part 2:

```
import numpy as np
import matplotlib.pyplot as plt

# Define the signals
H = 0  # Replace with your student number
I = 1  # Replace with your student number
N = 10
n = np.arange(N)
x1 = np.exp(1j * 2 * np.pi * n * (10 * (H + 1)) / 100) + np.exp(1j * 2 * np.pi * n * 33 / 100)
x2 = np.cos(2 * np.pi * n * (10 * (H + 1)) / 100) + 0.5 * np.cos(2 * np.pi * n * (10 * (I + 1)) / 100)

# Zero-pad the signals
x1zp = np.pad(x1, (245, 245), 'constant', constant_values=(0, 0))
x2zp = np.pad(x2, (245, 245), 'constant', constant_values=(0, 0))

# Compute the DFT magnitude of x1 and x2
X1 = np.abs(np.fft.fft(x1zp, 500))
X2 = np.abs(np.fft.fft(x2zp, 500))

# Plot the DFT magnitude spectra
fr = np.arange(-250, 250) / 500
plt.stem(fr, np.fft.fftshift(X1), use_line_collection=True)
plt.xlabel('Frequency (fr)')
plt.ylabel('Magnitude')
plt.title('DFT Magnitude of x1[n] (zero-padded)')
plt.show()

plt.stem(fr, np.fft.fftshift(X2), use_line_collection=True)
plt.xlabel('Frequency (fr)')
plt.ylabel('Magnitude')
plt.title('DFT Magnitude of x2[n] (zero-padded)')
plt.show()
```

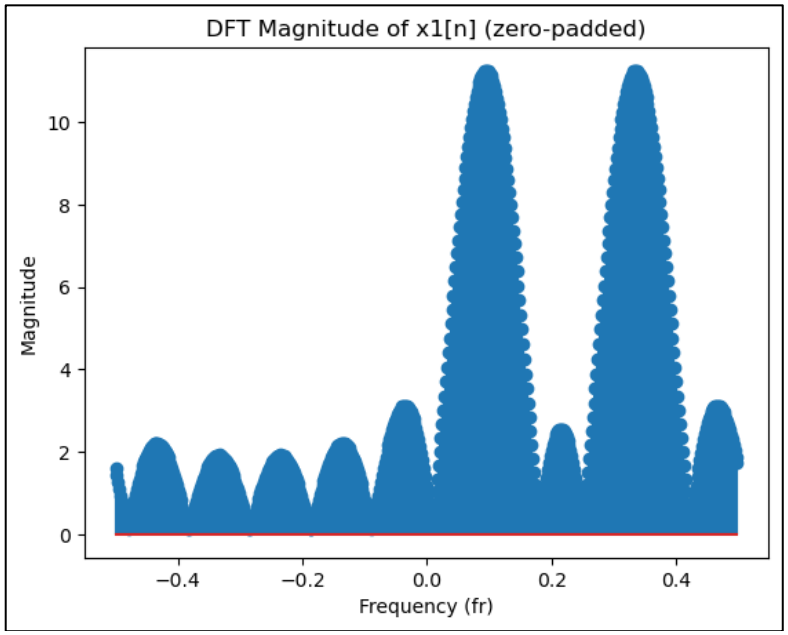

Figure 3: Plot for X1[n] with 10 samples but zero-padded with 490 zeros

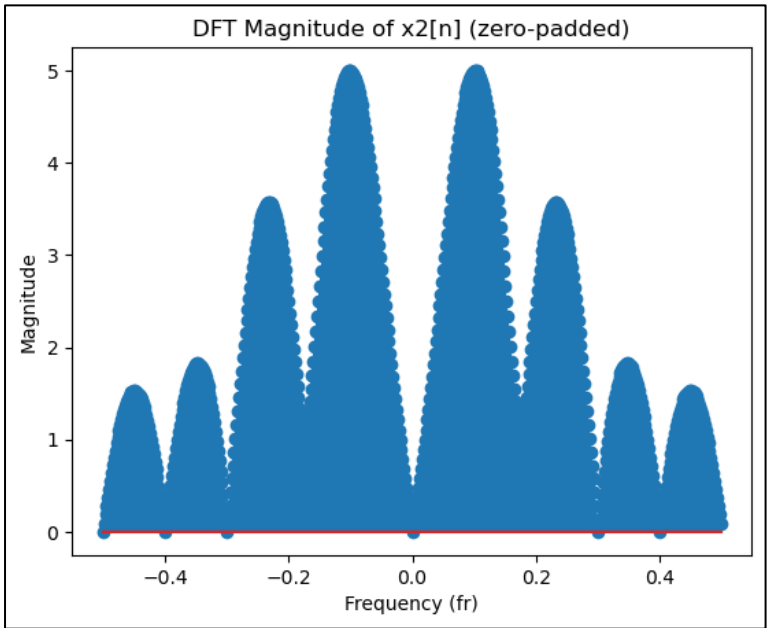

Figure 4: Plot for X2[n] with 10 samples but zero-padded with 490 zeros

A2. Yes, we gain a better idea of how our spectrum should seem as we zero-pad the signals. It seems more uninterrupted.

Python code for Part 3:

```
import numpy as np
import matplotlib.pyplot as plt

# Parameters
N = 100    # Number of samples
H = 0      # First digit of my student ID
I = 1      # Second digit of my student ID

# Generate signals
n = np.arange(N)
x1 = np.exp(1j*2*np.pi*n*(10*(H+1))/100) + np.exp(1j*2*np.pi*n*33/100)
x2 = np.cos(2*np.pi*n*(10*(H+1))/100) + 0.5*np.cos(2*np.pi*n*(10*(I+1))/100)

# Compute DFT
X1 = np.fft.fft(x1)
X2 = np.fft.fft(x2)

# Compute frequency axis
fr = np.arange(-N/2, N/2)*(1/N)

# Plot DFT magnitude
fig, axs = plt.subplots(2, 1, figsize=(8, 6))

axs[0].stem(fr, np.abs(np.fft.fftshift(X1)), use_line_collection=True)
axs[0].set_xlabel('Frequency')
axs[0].set_ylabel('|X1(f)|')
axs[0].set_xlim([-0.5, 0.5])
axs[0].set_title('DFT Magnitude of x1[n]')

axs[1].stem(fr, np.abs(np.fft.fftshift(X2)), use_line_collection=True)
axs[1].set_xlabel('Frequency')
axs[1].set_ylabel('|X2(f)|')
axs[1].set_xlim([-0.5, 0.5])
axs[1].set_title('DFT Magnitude of x2[n]')

plt.tight_layout()
plt.show()
```

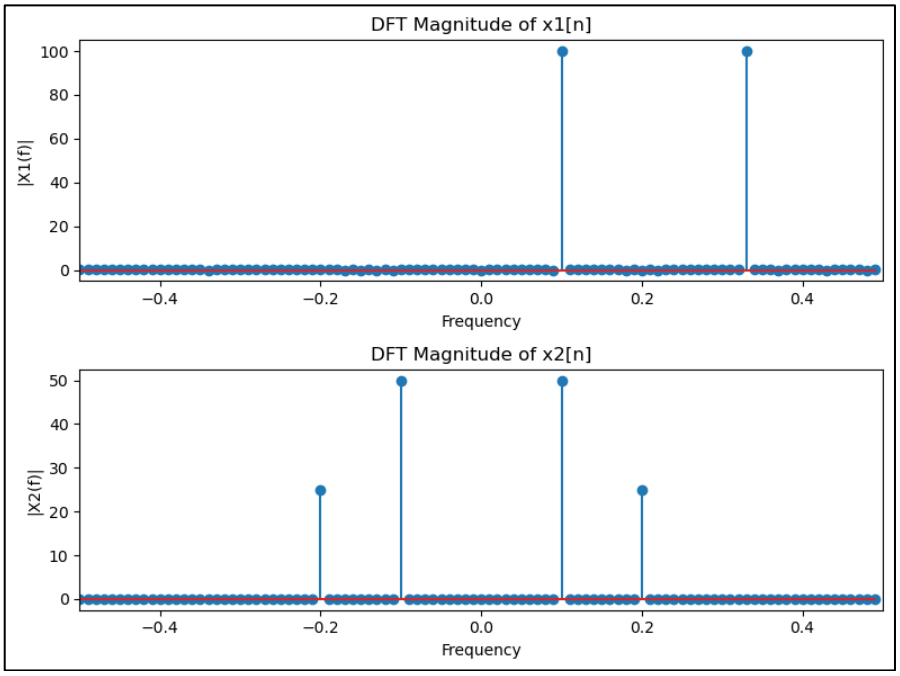

Figure 5: Plot for X1[n] and X2[n] with 100 samples

A3. X2[n] is a symmetric spectrum because it is an even cosine signal which is periodic**.**

Python code for Part 4:

```
import numpy as np
import matplotlib.pyplot as plt

# Define signals
H = 0  # replace with your own value
I = 1  # replace with your own value

n = np.arange(100)
x1 = np.exp(1j * 2 * np.pi * n * (10 * (H + 1)) / 100) + np.exp(1j * 2 * np.pi * n * 33 / 100)
x2 = np.cos(2 * np.pi * n * (10 * (H + 1)) / 100) + 0.5 * np.cos(2 * np.pi * n * (10 * (I + 1)) / 100)

# Zero-pad signals with 400 zeros
N = 500
x1_padded = np.pad(x1, (200, 200), 'constant')
x2_padded = np.pad(x2, (200, 200), 'constant')

# Compute DFTs
X1 = np.fft.fft(x1_padded, N)
X2 = np.fft.fft(x2_padded, N)

# Compute magnitude spectra
X1_mag = np.abs(X1)
X2_mag = np.abs(X2)

# Plot spectra
freq = np.arange(-N/2, N/2) / N
plt.plot(freq, np.fft.fftshift(X1_mag), label='x1')
plt.plot(freq, np.fft.fftshift(X2_mag), label='x2')
plt.title('Length-500 DFT Magnitude Spectra (zero-padded with 400 zeros)')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')
plt.legend()
plt.show()
```

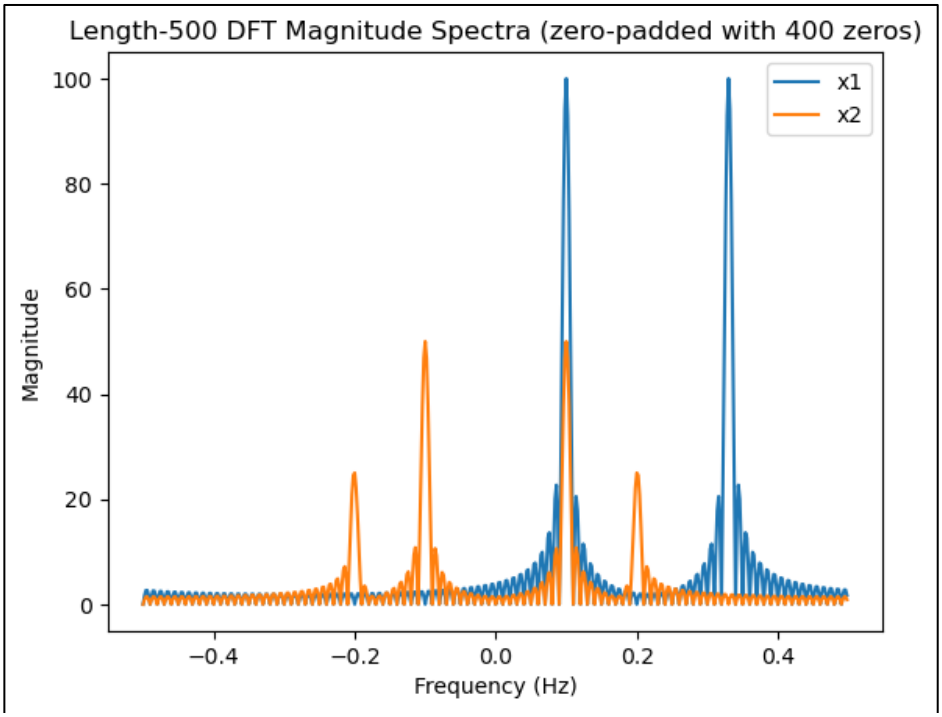

Figure 6: Plot for x1 and x2 with 100 samples but zero-padded with 400 zeros

A4. Yes, the signal is represented considerably more accurately with more samples and zero-padding.

B) Sampling

Python code for Part 1-5:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
from scipy.fft import  fft, ifft, fftshift
import soundfile as sf

from IPython.display import Audio

# 1) Read sound file
fs, y = wavfile.read('chirp.wav')
if y.ndim > 1:
    y = y[:, 0]  # use only one channel

N0 = len(y)      # Number of samples
T0 = N0 / fs     # Duration of the signal
T = 1 / fs       # Sampling interval

# 2) Plot signal y with respect to time
t = np.linspace(0, T0, N0)
plt.figure()
plt.plot(t, y)
plt.title('Signal y with respect to time')
plt.xlabel('t')
plt.grid()

# 3) Compute and plot DFT of signal y
omega = np.linspace(-(fs / 2), (fs / 2), N0)
Y = np.fft.fft(y)
plt.figure()
plt.plot(omega, np.fft.fftshift(abs(Y)))
plt.title('DFT of signal y')
plt.xlabel('w')
plt.grid()

# 4) Generate subsampled signal y1
y1 = y[::2]
N1 = len(y1)
T01 = N1 / fs
T1 = 2 * fs

# 5) Plot signal y1 with respect to time
t1 = np.linspace(0, T01, N1)
plt.figure()
plt.plot(t1, y1)
plt.title('Signal y1 with respect to time')
plt.xlabel('t')
plt.grid()
```

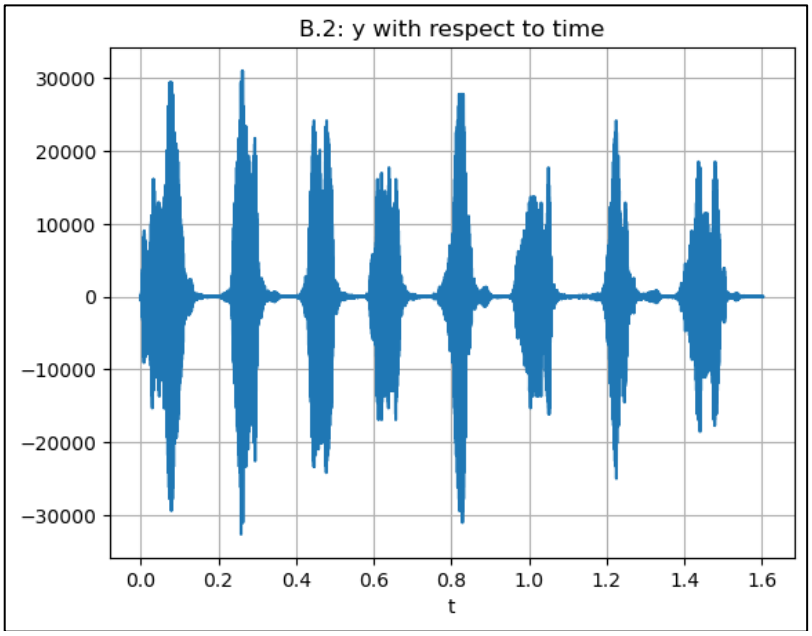

Figure 7: Plot of signal ***y*** with respect to time

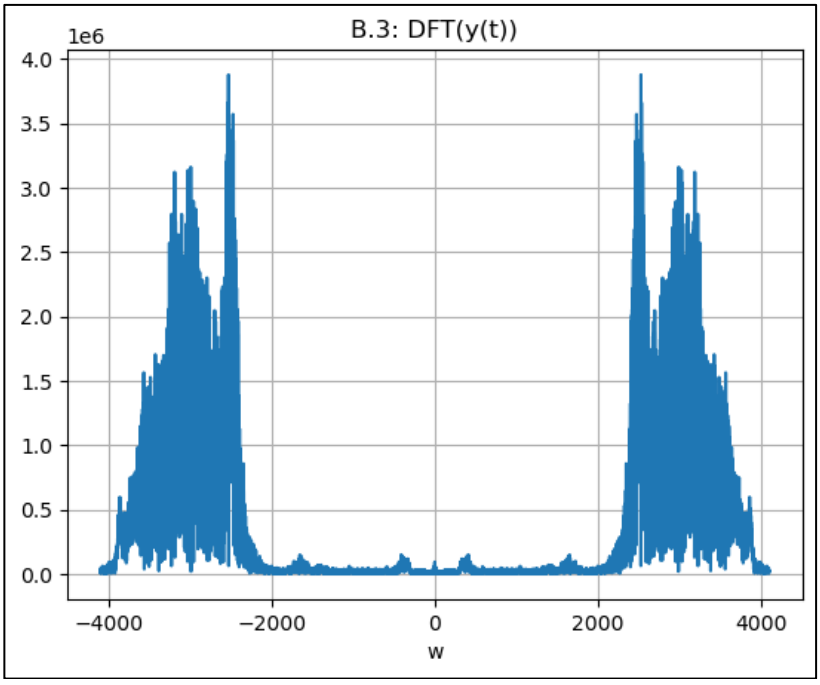

Figure 8: Plot of Discrete Fourier Transformed Signal

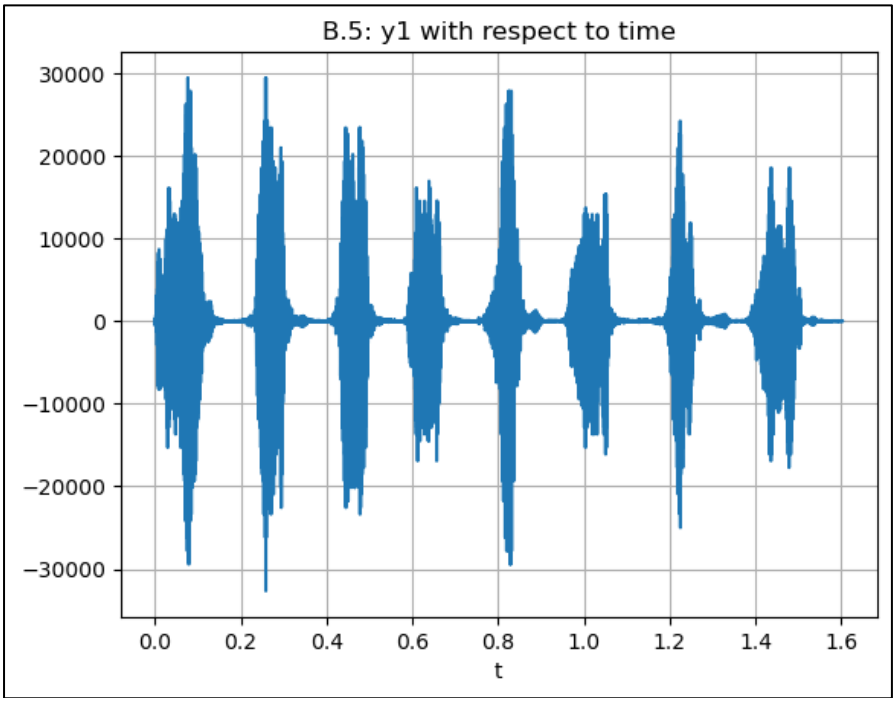

Figure 9: Plot of Rate 2 Subsampled Signal ***y1*** with respect to time

```
N0 = 13129

T0 = 1.6027

T = 1.2207e-04

N1 = 6565

T01 = 0.8014

T1 = 16384
```

The signal has subsampled by a factor of 2. On the spectrum, the frequency was halved (1/2).

Python code for Part 6-8:

```
# 6) Compute and plot DFT of y1
omega1 = np.linspace(-(fs / 4), (fs / 4), N1)
Y1 = np.fft.fft(y1)
plt.figure()
plt.plot(omega1, np.fft.fftshift(abs(Y1)))
plt.title('DFT of signal y1')
plt.xlabel('w')
plt.grid()

# Enlarge part of the plots for better visualization
plt.figure()
plt.subplot(2, 1, 1)
plt.plot(t[:500], y[:500])
plt.title('Signal y with respect to time (first 500 samples)')
plt.xlabel('t')
plt.grid()
plt.subplot(2, 1, 2)
plt.plot(t1[:500], y1[:500])
plt.title('Signal y1 with respect to time (first 250 samples)')
plt.xlabel('t')
plt.grid()

# 7) Play original and subsampled signals
sd.play(y, fs)
sd.wait()
sd.play(y1, fs)

# 8) Generate subsampled signal y5 with rate 5
y5 = y[::5]
N5 = len(y5)
T05 = N5 / fs
T5 = 5 * T

# Plot signal y5 with respect to time
t5 = np.linspace(0, T05, N5)
plt.figure()
plt.plot(t5, y5)
plt.title('Signal y5 with respect to time')
plt.xlabel('t')
plt.grid()

# Compute and plot DFT of y5
omega5 = np.linspace(-(fs / 10), (fs / 10), N5)
Y5 = np.fft.fft(y5)
plt.figure()
plt.plot(omega5, np.fft.fftshift(abs(Y5)))
plt.title('DFT of signal y5')
plt.xlabel('w')
plt.grid()
```

N5 = 2626

T05 = 0.3206

T5 = 16384

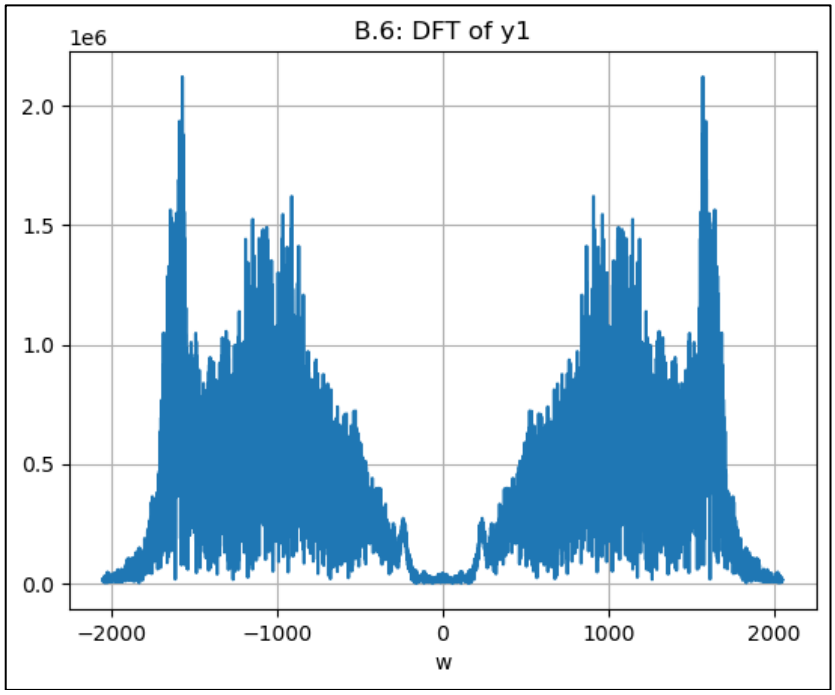

Figure 10: Plot of Discrete Fourier Transformed Subsampled Signal ***y1***

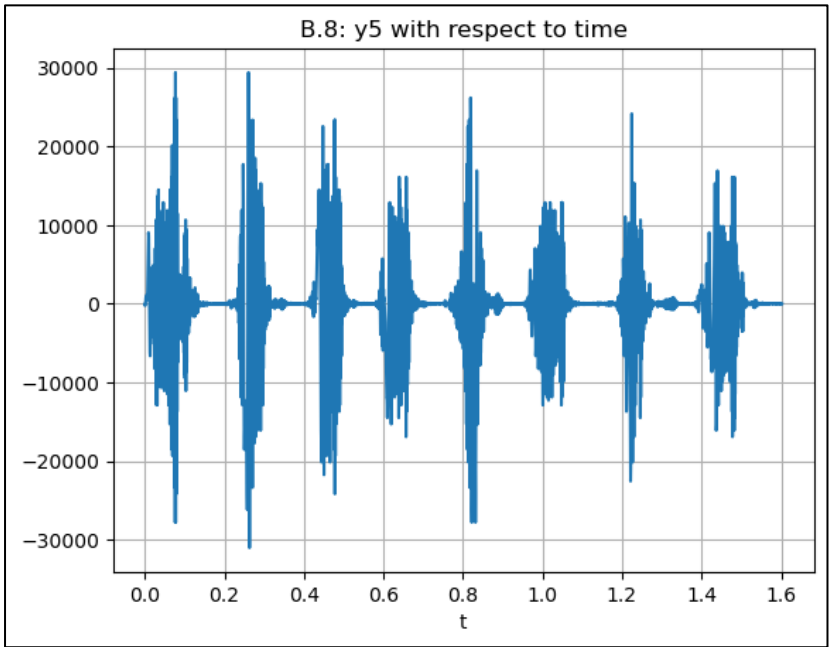

Figure 11: Subsample of signal ***y5*** (rate 5) with respect to time

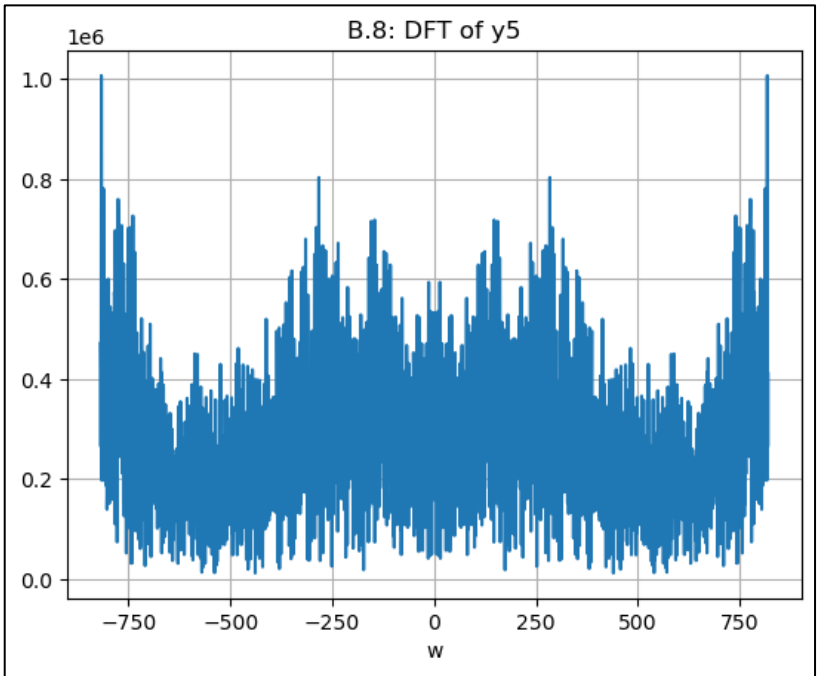

Figure 12: Plot for DFT of audio signal ***y5***

B6. The signal y1 has not altered all that much, but the frequency spectrum now contains more components because of the two-rate sampling and data loss required to create a precise representation of the signal.

B7. Although the audio signals are identical, the subsample audio has a faster sound.

B8. Part 4's introduction of additional frequency components had a similar impact on the spectrum as Part 5's introduction of additional frequency components did.

C) Filter Design

Python code for Part 1 and 2:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile

# Part 1 - Rect Filter
fs, y = wavfile.read('chirp.wav')
n = len(y)
range_ = np.arange(-n/2, n/2)
period = 1/fs
p = n*period
t = np.arange(n)
j = 1/p
f = range_*j
filter_2000 = abs(f) < 2000
Y = np.fft.fftshift(np.fft.fft(y))
Yfiltered = Y*filter_2000
ytime = np.fft.ifft(np.fft.fftshift(Yfiltered))
plt.plot(t, ytime.real)
plt.title("Plot of filtered sound (2000kHz) in time domain")
plt.xlabel('t')
plt.ylabel('amplitude')
plt.show()
plt.plot(f, abs(Yfiltered))
plt.title("Plot of filtered sound (2000kHz) in frequency domain")
plt.xlabel('frequency')
plt.ylabel('|Y(Ω)|')
plt.show()

# Part 2 - Playing the Rect Filtered Signal
from playsound import playsound  # install playsound library for playing the sound
playsound('chirp.wav')
playsound(ytime.real, fs)
```

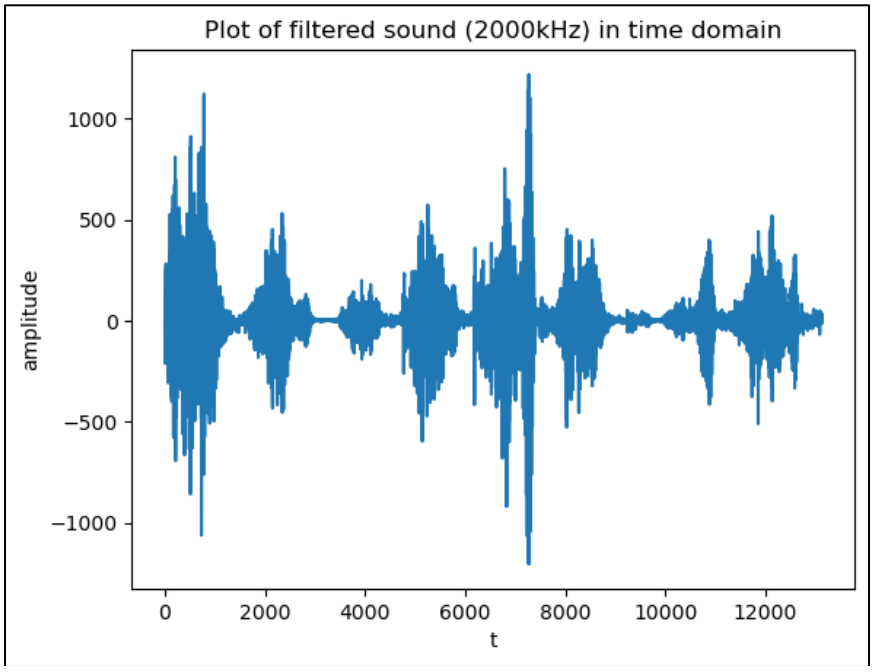

Figure 13: Plot of audio signal (2000khz) in time domain

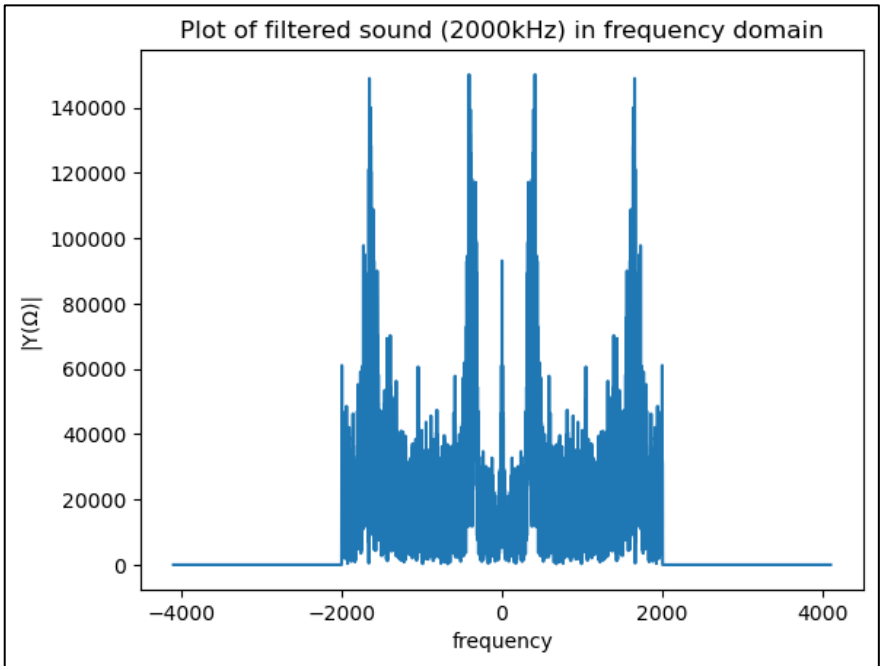

Figure 14: Plot of audio signal (2000khz) in frequency domain

C2. Elements of the tune that carried frequencies greater than +-2kHz were eliminated, making those portions mute.

Python code for Part 3:

```
# Part 3 - Bass Frequency (16Hz - 256Hz) Filter
fs, y = wavfile.read('chirp.wav')
n = len(y)
range_ = np.arange(-n/2, n/2)
period = 1/fs
p = n*period
t = np.arange(n)
j = 1/p
f = range_*j
filter_16_256 = (abs(f) < 16) | (abs(f) > 256)
Y = np.fft.fftshift(np.fft.fft(y))
Yfiltered = Y*filter_16_256
ytime = np.fft.ifft(np.fft.fftshift(Yfiltered))
plt.plot(t, ytime.real)
plt.title("Plot of filtered sound (16-256kHz) in time domain")
plt.xlabel('t')
plt.ylabel('amplitude')
plt.show()
plt.plot(f, abs(Yfiltered))
plt.title("Plot of filtered sound (16-256kHz) in frequency domain")
plt.xlabel('frequency')
plt.ylabel('|Y(Ω)|')
plt.show()
playsound(ytime.real, fs)
```

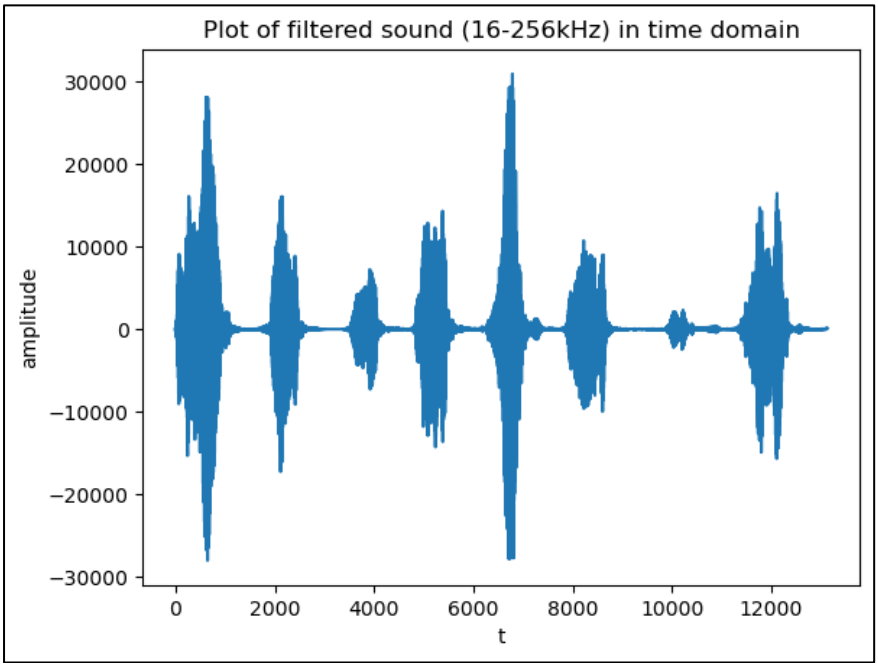

Figure 15: Plot for filtered sound of Bass Frequency in time domain

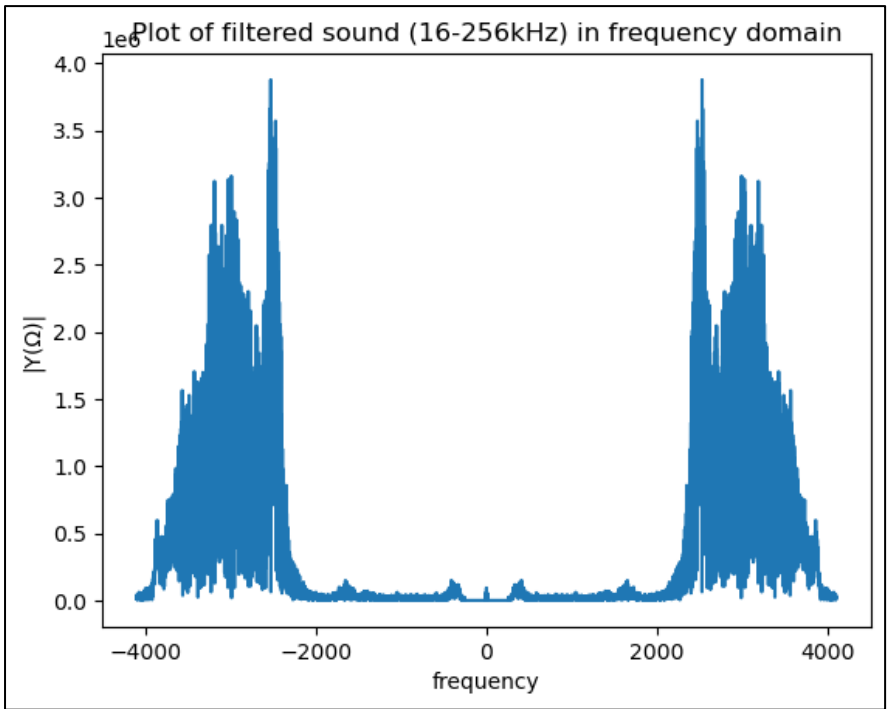

Figure 16: Plot for filtered sound of Bass Frequency in frequency domain

C3. There was silence in place of the low frequencies/bass sounds. The range of frequencies between 16 Hz and 256 Hz is no longer present in the frequency spectrum.

Python code for Part 4:

```
# Part 4 - Treble Frequency (2048Hz - 16384Hz) Amplifying Filter
fs, y = wavfile.read('chirp.wav')
n = len(y)
range_ = np.arange(-n/2, n/2)
period = 1/fs
p = n*period
t = np.arange(n)
j = 1/p
f = range_*j
filter_2048_16384 = (abs(f) > 2048) | (abs(f) < 16384)
Y = np.fft.fftshift(np.fft.fft(y))
Yfiltered = Y*filter_2048_16384*1.25
ytime = np.fft.ifft(np.fft.fftshift(Yfiltered))
plt.plot(t, ytime.real)
plt.title("Plot of filtered and amplified sound (2048-16384kHz) in time domain")
plt.xlabel('t')
plt.ylabel('amplitude')
plt.show()
plt.plot(f, abs(Yfiltered))
plt.title("Plot of filtered and amplified sound (2048-16384kHz) in frequency domain")
plt.xlabel('frequency')
plt.ylabel('|Y(Ω)|')
plt.show()
playsound(ytime.real, fs)
```

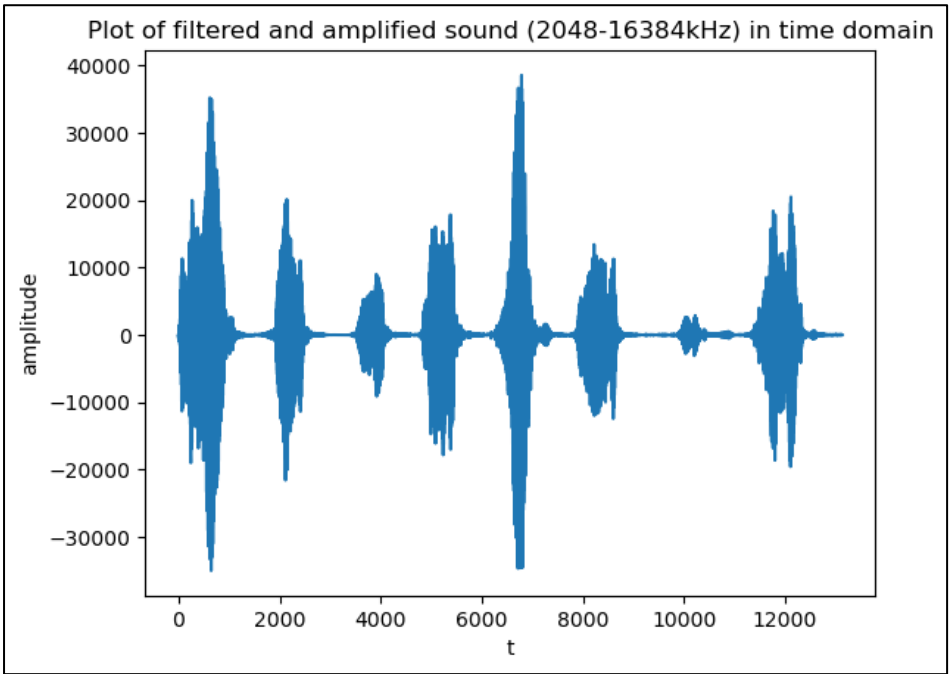

Figure 17: Plot for filtered sound of Treble Frequency in time domain

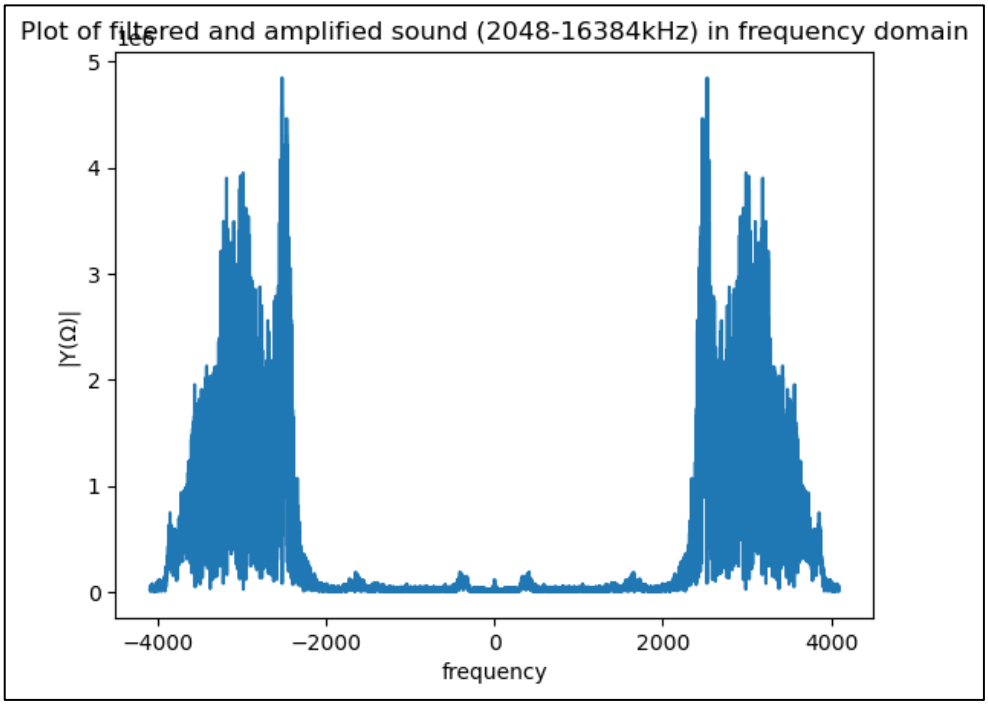

Figure 18: Plot for filtered sound of Treble Frequency in frequency domain

C4. The tune's higher frequencies were boosted and became louder. Considering frequencies between 2048 Hz and 16384 Hz, the average frequency has been increased by 25%.

C5. The Convolution Theorem, which states that convolution in the time domain is equivalent to multiplication in the frequency domain, is a DFT property that was used to complete the work in Part 4.