**CSC – 575 (Assignment -1)**

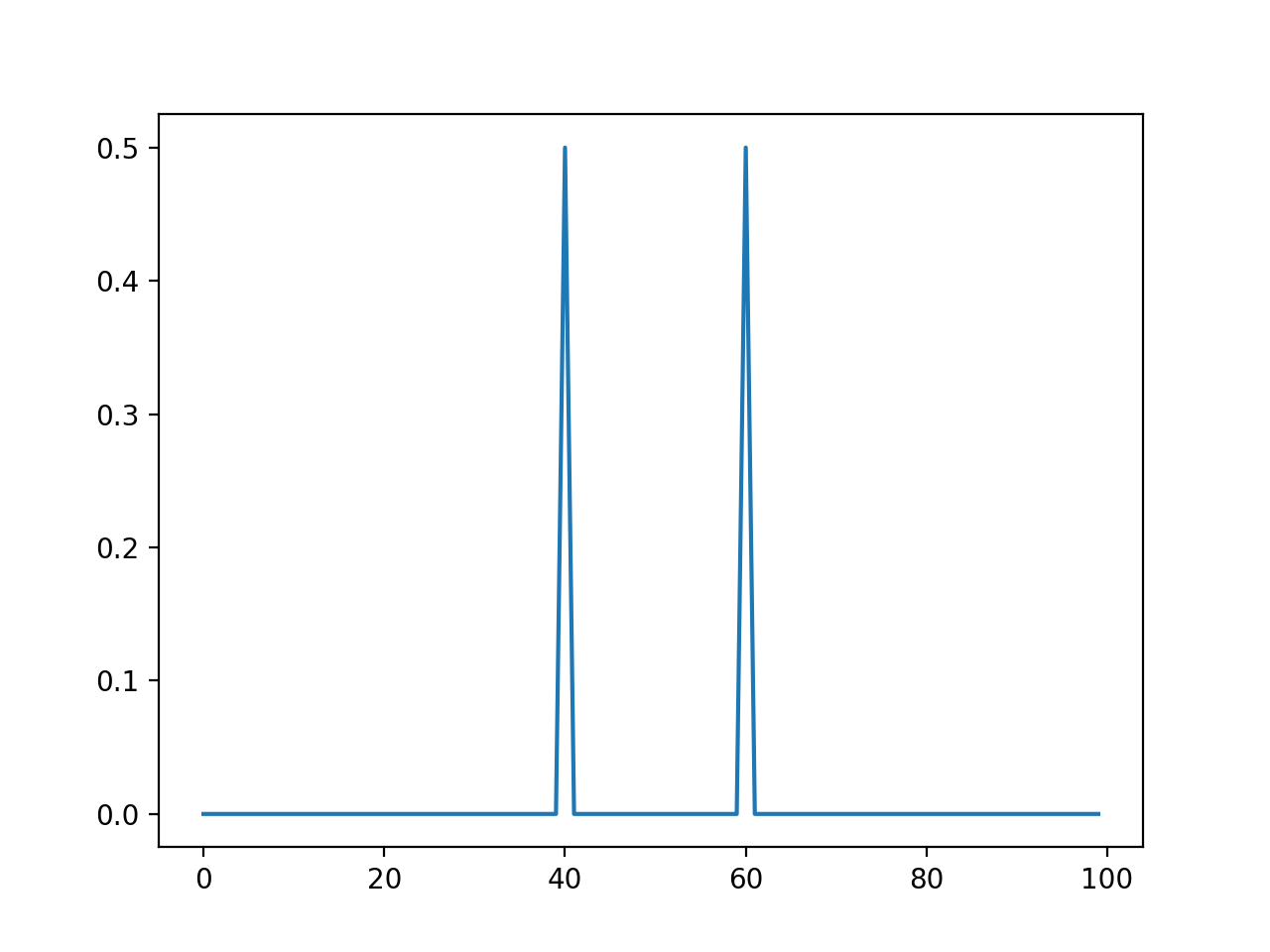
**Question1:**

Part – 1: Low level DFT Computation method

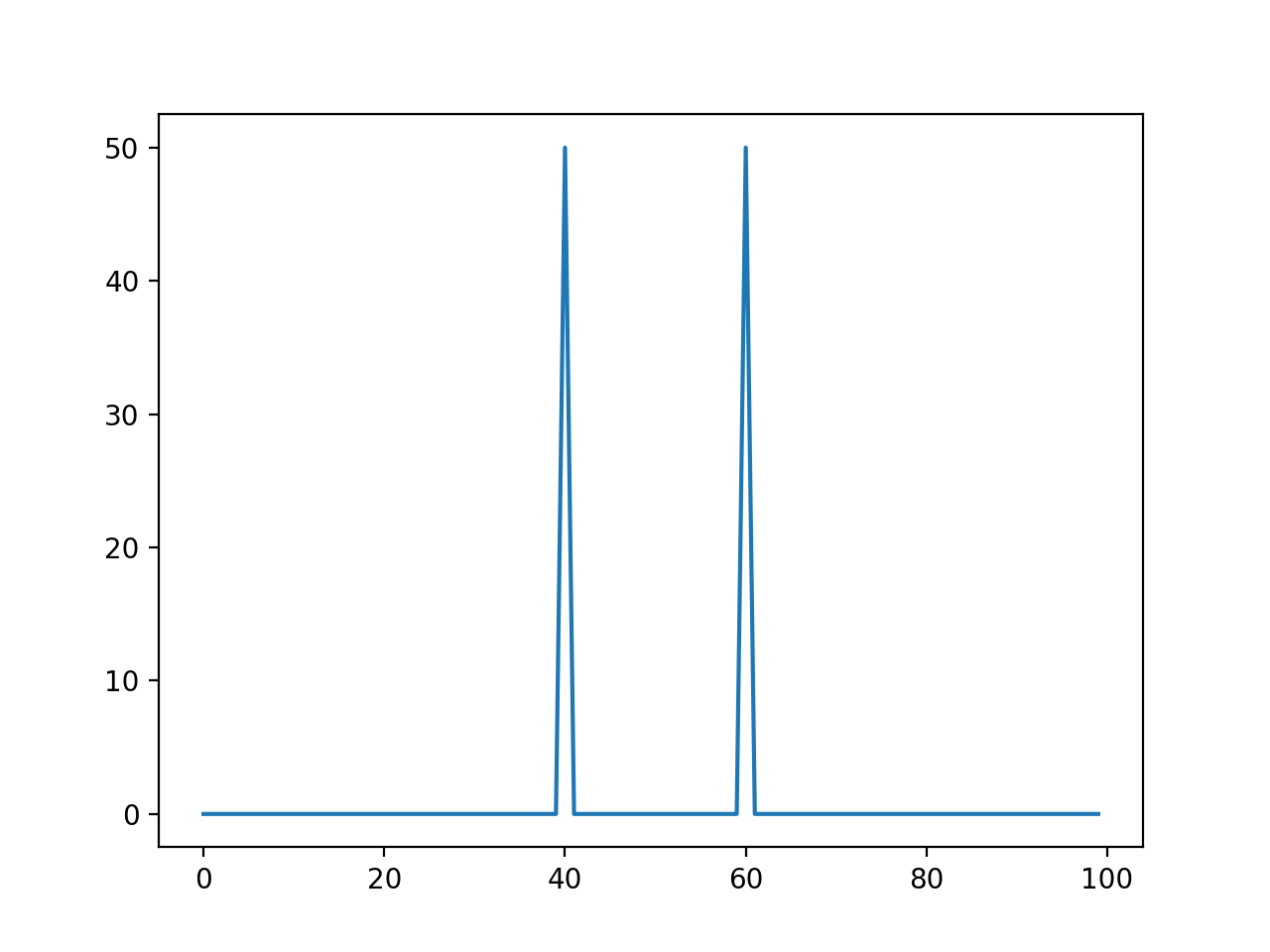
Code:

**import** numpy **as** np  
**import** matplotlib.pyplot **as** plt  
**import** math  
**import** mir  
**from** mir **import** Sinusoid  
*#get the sin wave*sin = Sinusoid(Fs=100)  
*#differentiate real and imaginary components*datar = sin.data.real  
datai = sin.data.imag  
N = len(datar)  
outputr = []  
outputi = []  
**for** k **in** range(N):  
 sumr = 0  
 sumi = 0  
 **for** n **in** range(N):  
 sumr += datar[n]\*math.cos(2\*math.pi\*k\*n/N)+datai[n]\*math.sin(2\*math.pi\*k\*n/N)  
 sumi += -datar[n]\*math.sin(2\*math.pi\*k\*n/N)+datai[n]\*math.cos(2\*math.pi\*k\*n/N)  
 outputr.append(sumr)  
 outputi.append(sumi)  
outputp = []  
*#final output***for** a **in** range(N):  
 outputp.append(math.sqrt((outputr[a]\*outputr[a])+(outputi[a]\*outputi[a])))  
*#plot output*plt.plot(outputp)  
plt.show()  
*#find the plot dft with built in function*outputk = sin.dft()  
plt.plot(abs(outputk))  
plt.show()

Plot:



Generated from user written DFT function.



Generated from built in DFT function.

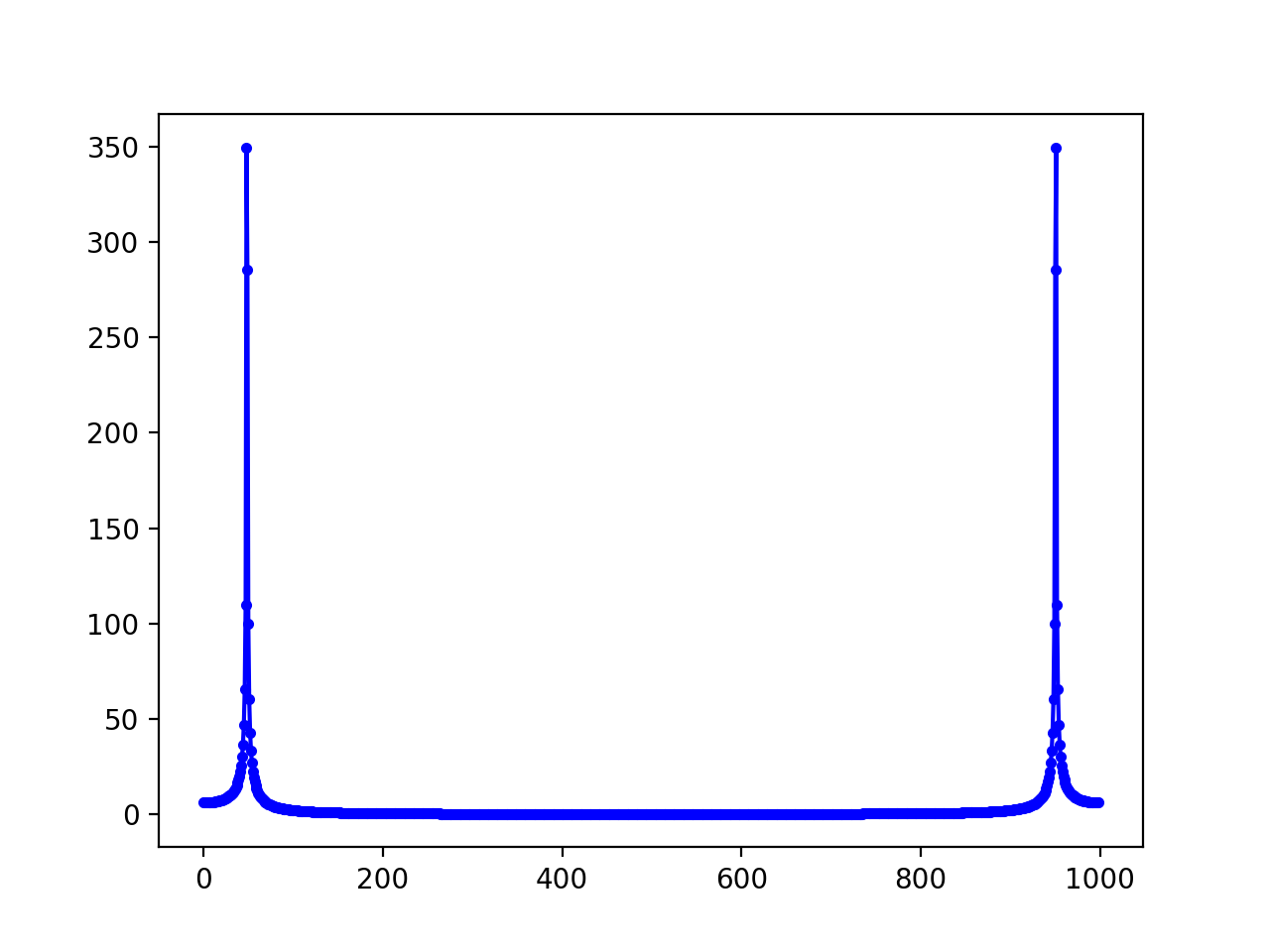
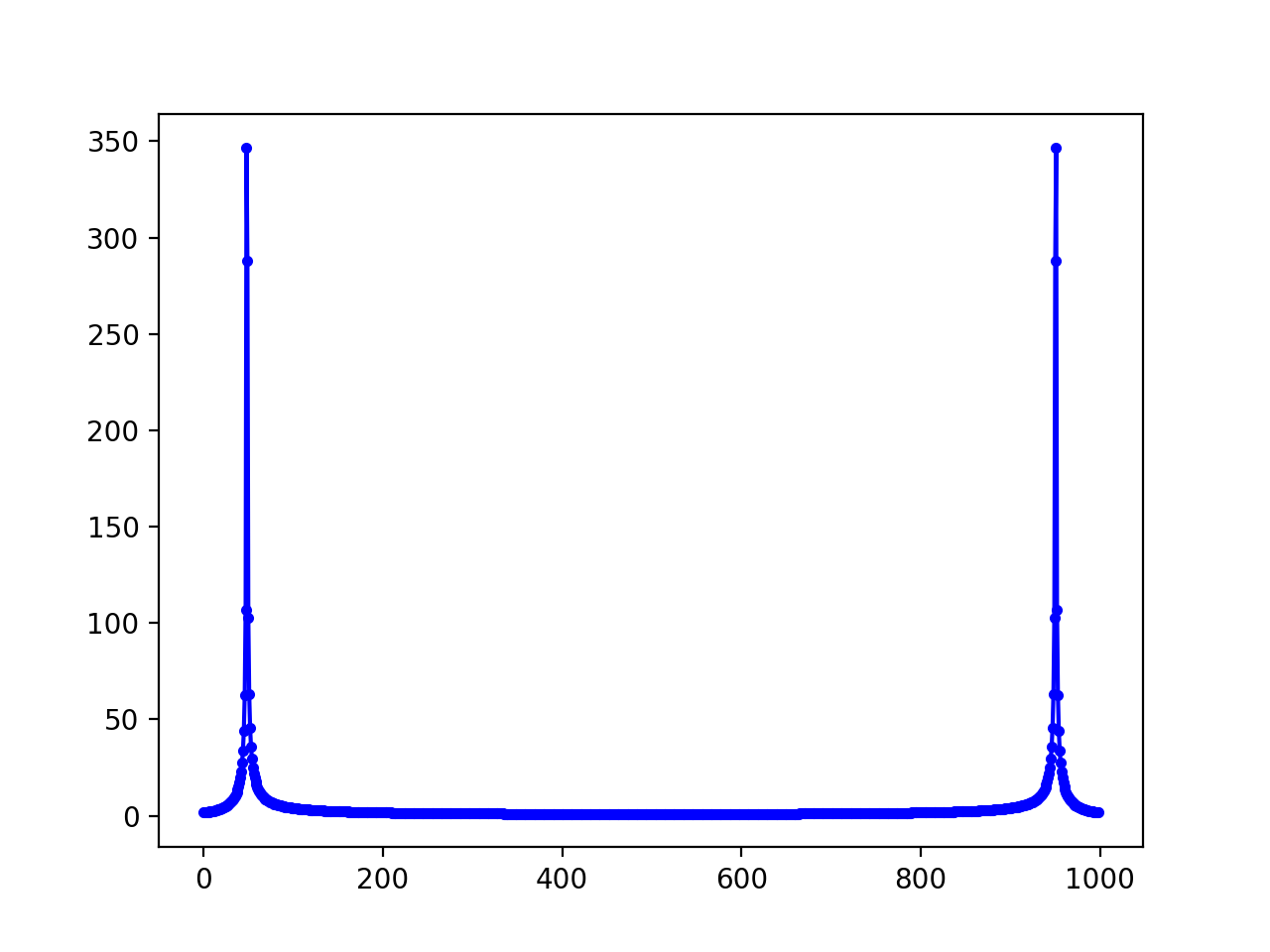
Part – 2: Linear Combination of 3 Sinusoidal

Code:

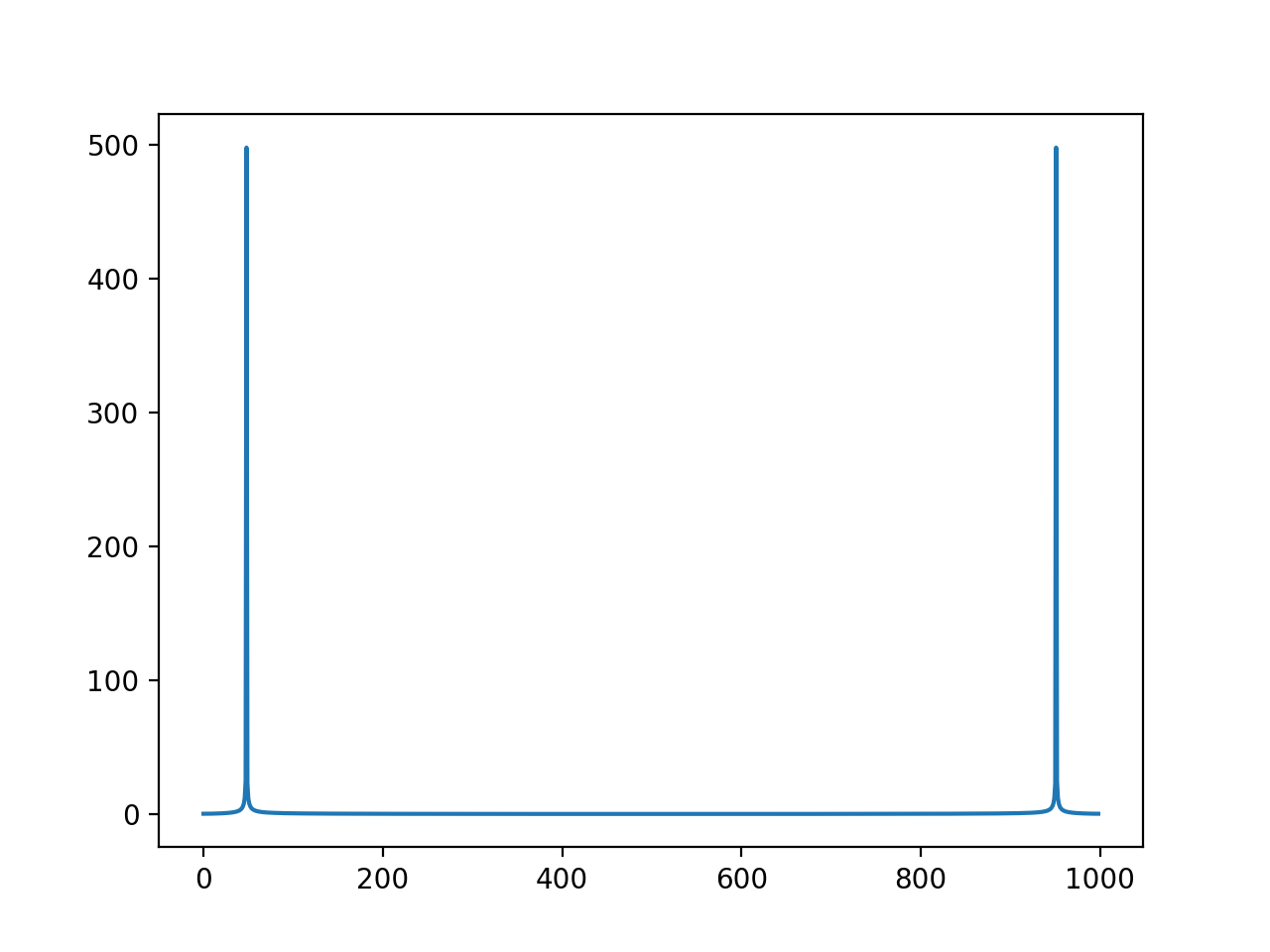
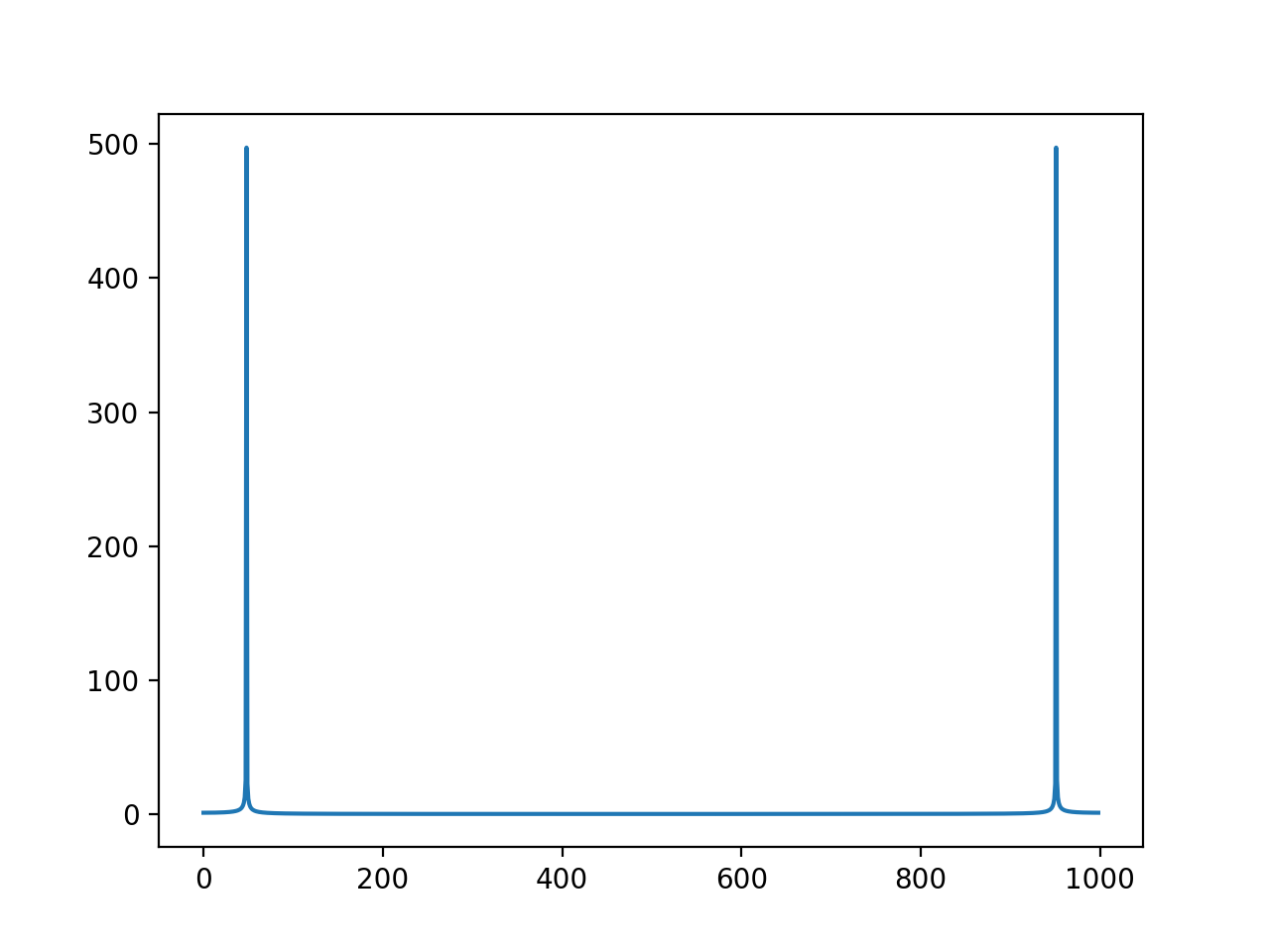
**import** numpy **as** np  
**import** matplotlib.pyplot **as** plt  
**import** math  
**import** mir  
**from** mir **import** Sinusoid  
*#fetch the 3 sinusoids*sin1 = Sinusoid(Fs=1000, amp=5, freq=20, phase=5)  
sin2 = Sinusoid(Fs=1000, amp=10, freq=40, phase=10)  
sin3 = Sinusoid(Fs=1000, amp=15, freq=80, phase=15)  
N = len(sin1.data)  
signal = []  
*#Make them one***for** a **in** range(N):  
 signal.append(sin1.data[a]+sin2.data[a]+sin3.data[a])  
plt.plot(signal)  
plt.show()  
*#calculating for bin 48*bin = 48  
k = bin  
cosa = []  
sina = []  
**for** n **in** range (N-1):  
 theta = 2\*math.pi\*k\*n/N  
 cosa.append(math.cos(theta))  
 sina.append(math.sin(theta))  
plt.plot(abs(np.fft.fft(cosa)))  
plt.show()  
plt.plot(abs(np.fft.fft(sina)))  
plt.show()  
*#calculating between bins*bin = 48.5  
k = bin  
cosa = []  
sina = []  
**for** n **in** range (N-1):  
 theta = 2\*math.pi\*k\*n/N  
 cosa.append(math.cos(theta))  
 sina.append(math.sin(theta))  
plt.plot(abs(np.fft.fft(cosa)),**'b.-'**)  
plt.show()  
plt.plot(abs(np.fft.fft(sina)),**'b.-'**)  
plt.show()

Plots:

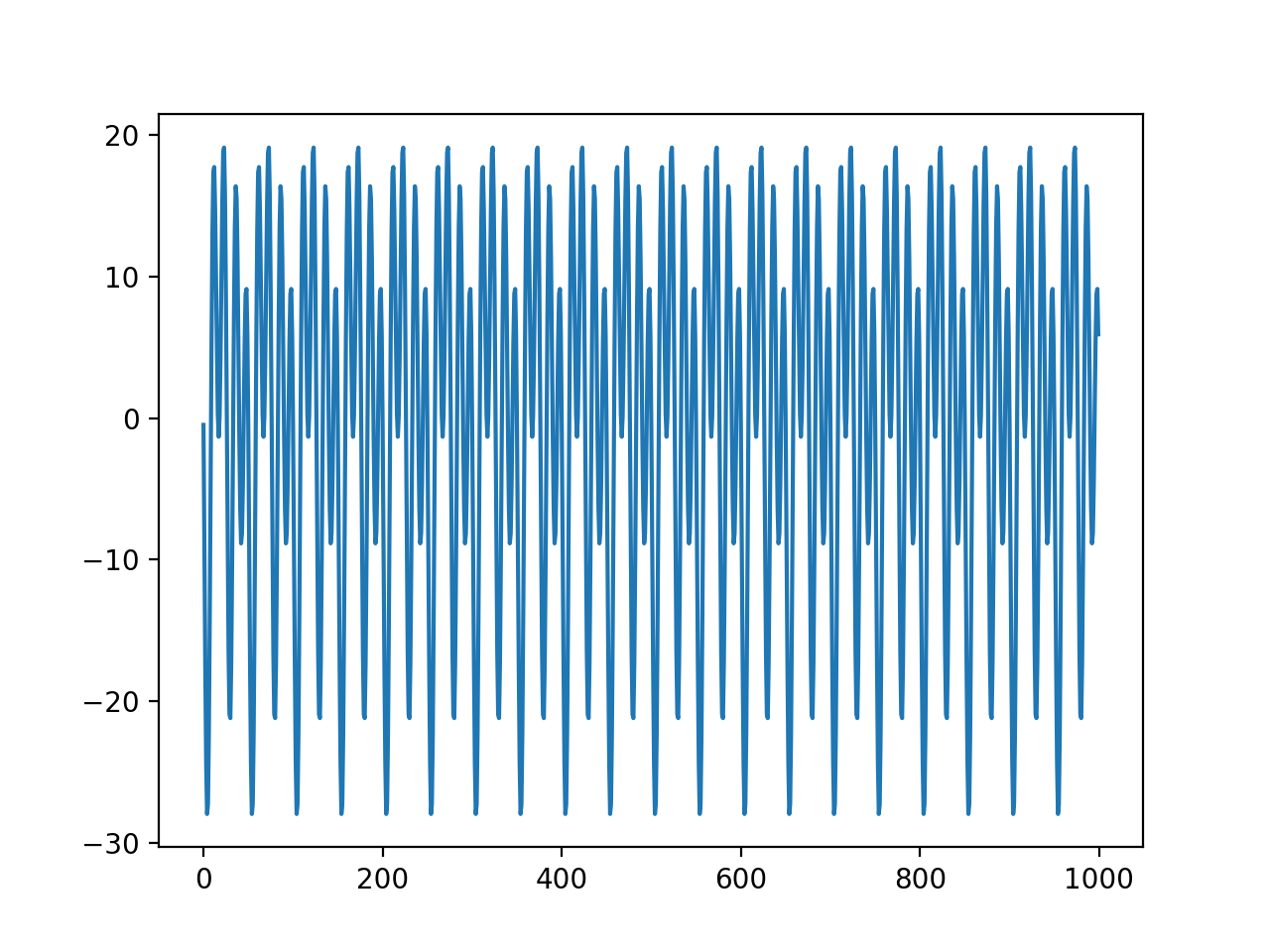
Plots for Cos and Sin with values within Bin



Plots for Cos and Sin with values on Bin



Collective all the 3 Sin additive

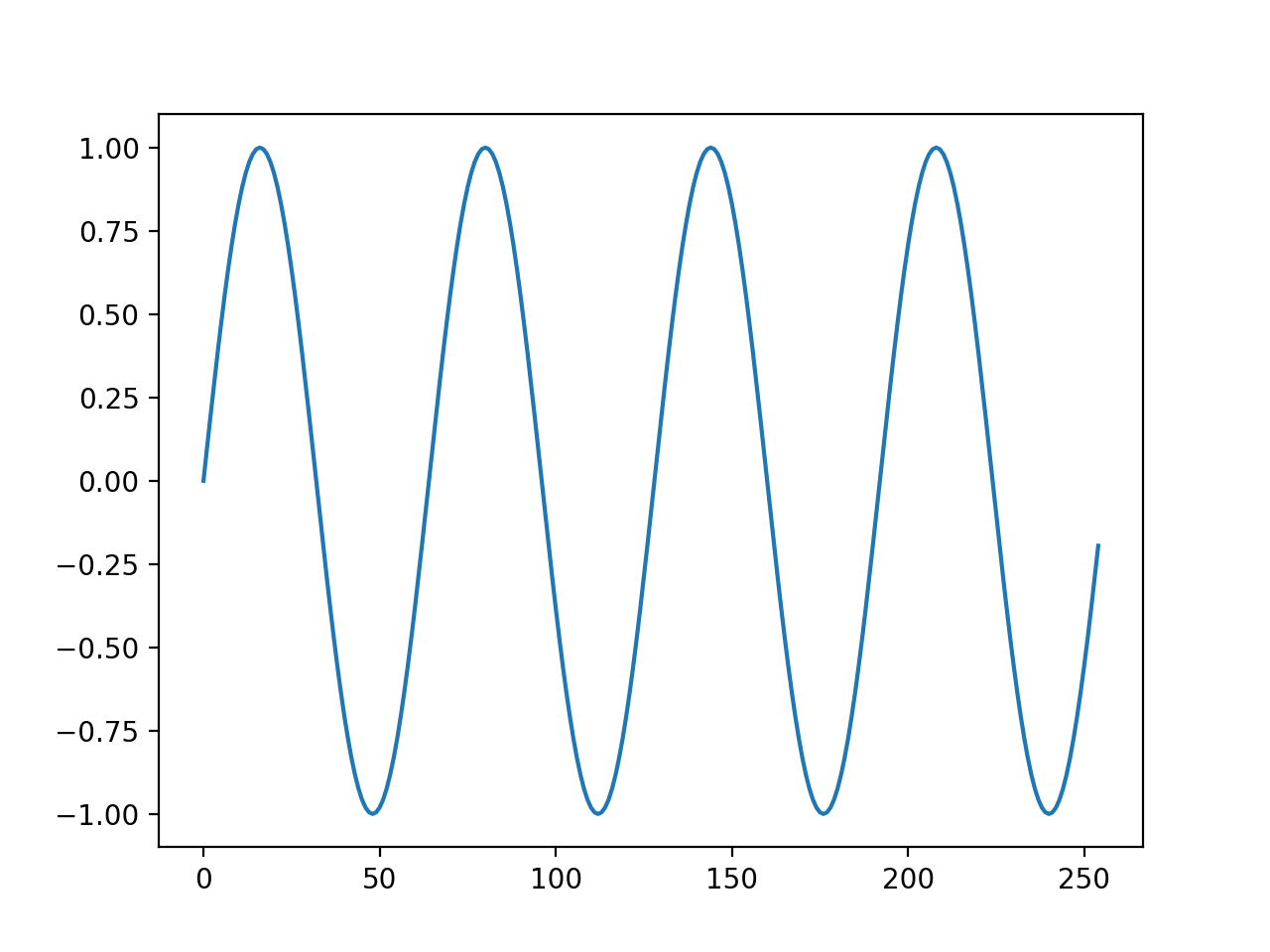
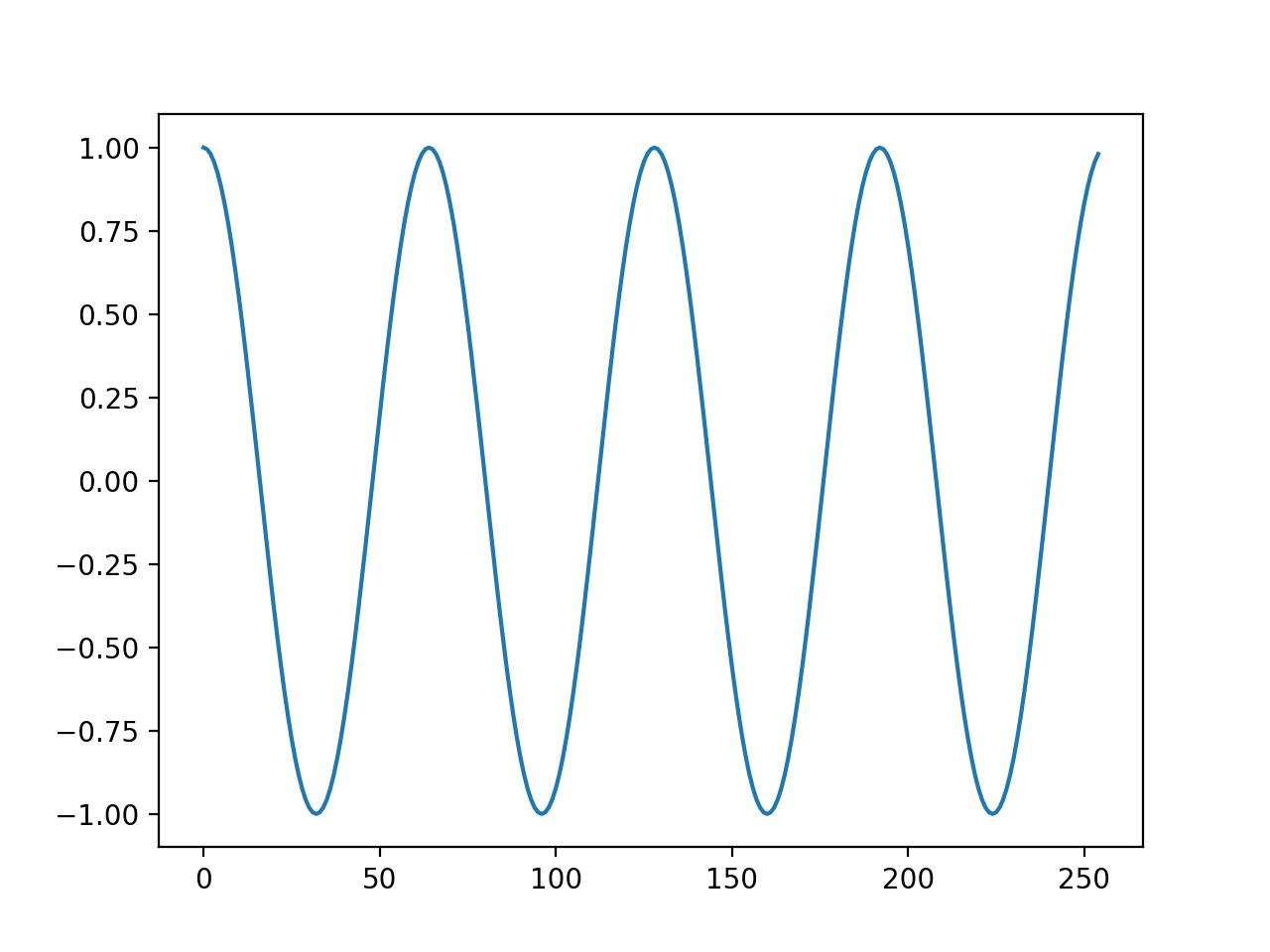


Part 3: Plot the Basis Function

Code:

**import** numpy **as** np  
**import** matplotlib.pyplot **as** plt  
**import** scipy.fftpack  
**import** math  
**import** mir  
**from** mir **import** Sinusoid  
  
sin1 = Sinusoid(Fs=256, amp=5, freq=20, phase=5)  
  
N = len(sin1.data)  
**print** N  
signal = []  
  
bin = 4  
k = bin  
  
cosa = []  
sina = []  
  
**for** n **in** range (N-1):  
 theta = 2\*math.pi\*k\*n/N  
 cosa.append(math.cos(theta))  
 sina.append(math.sin(theta))  
  
plt.plot(cosa)  
plt.show()  
plt.plot(sina)  
plt.show()

Plot:



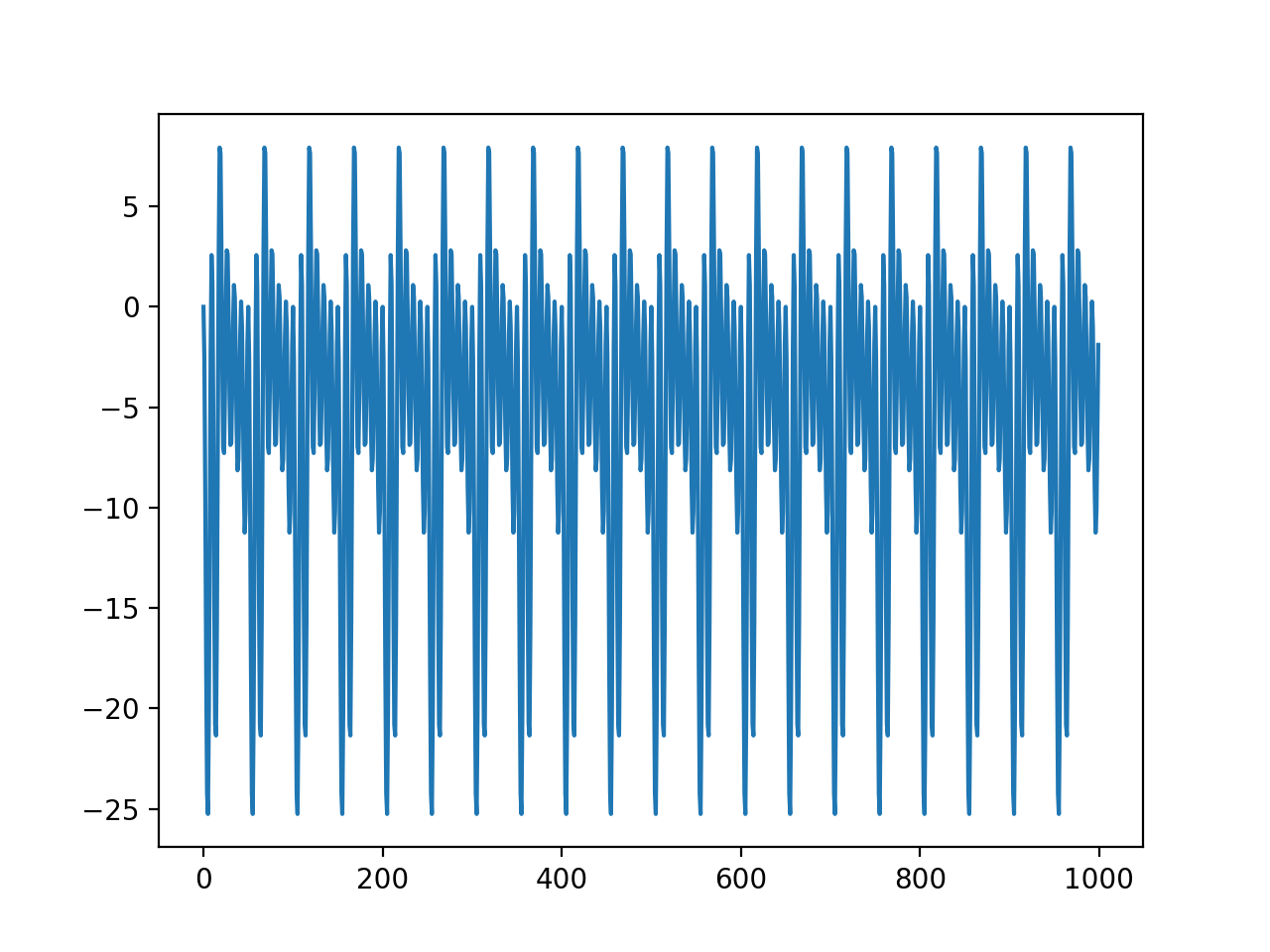
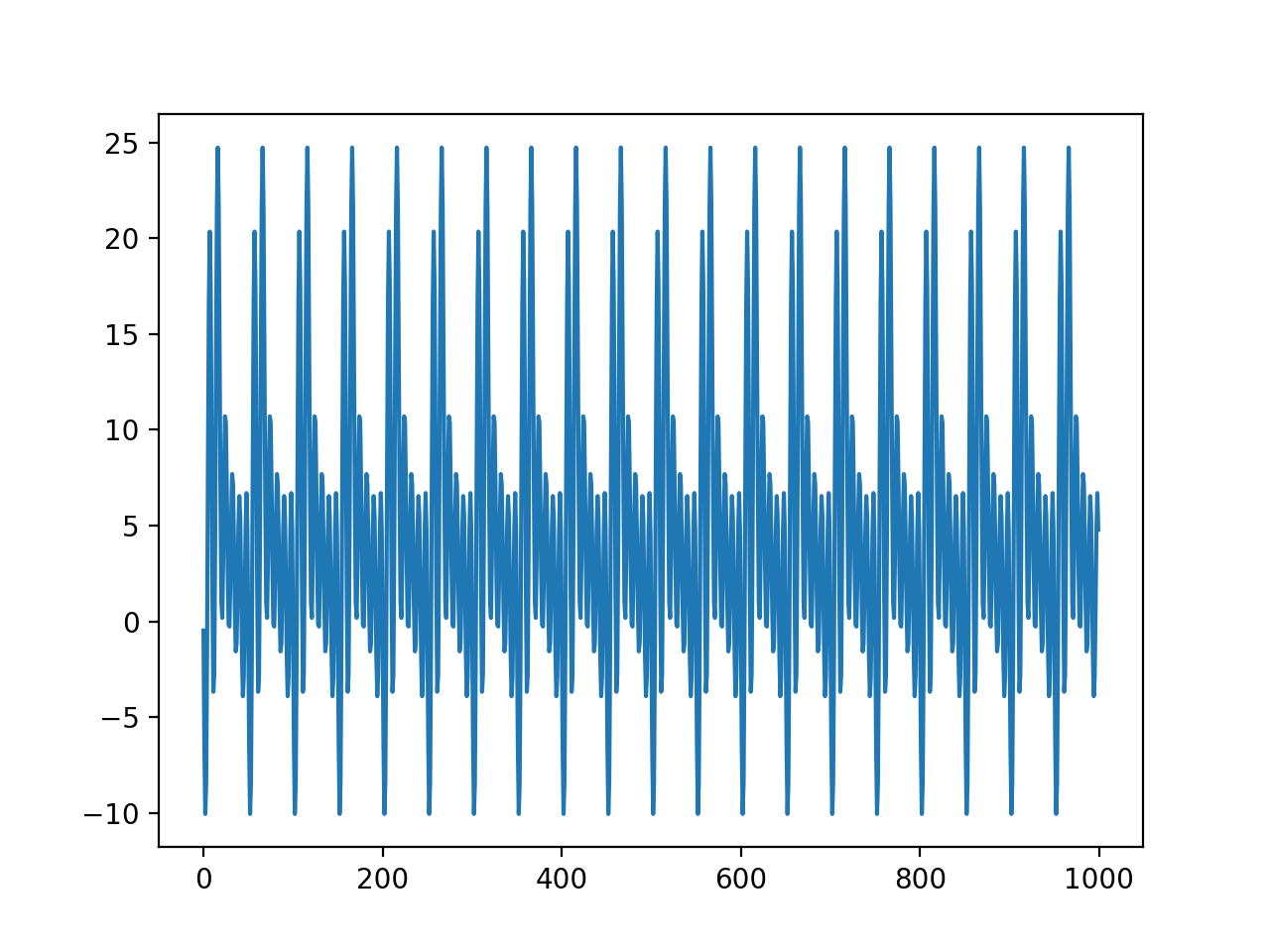
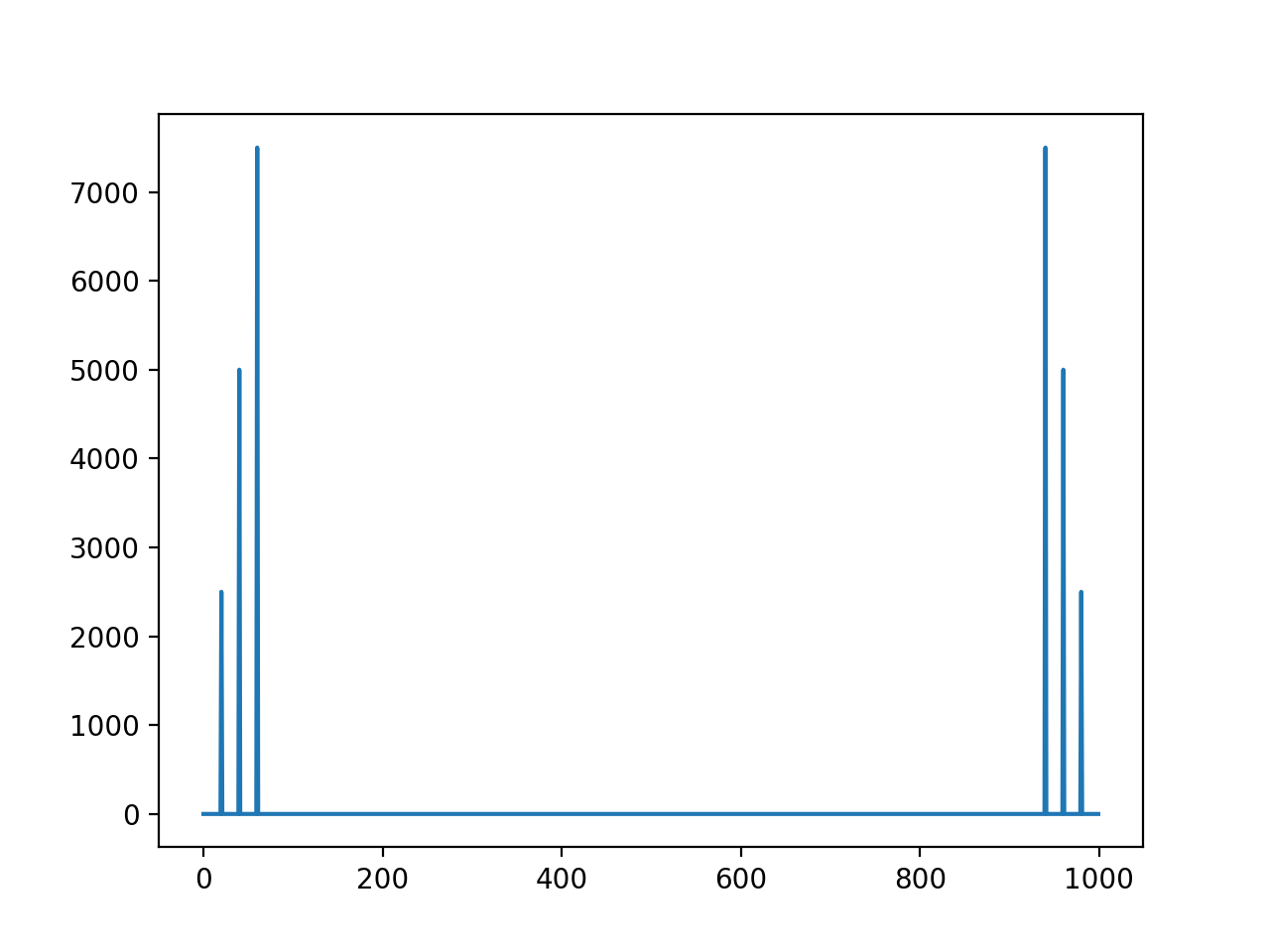
Part 4 : Point Wise Multiplication

Code:

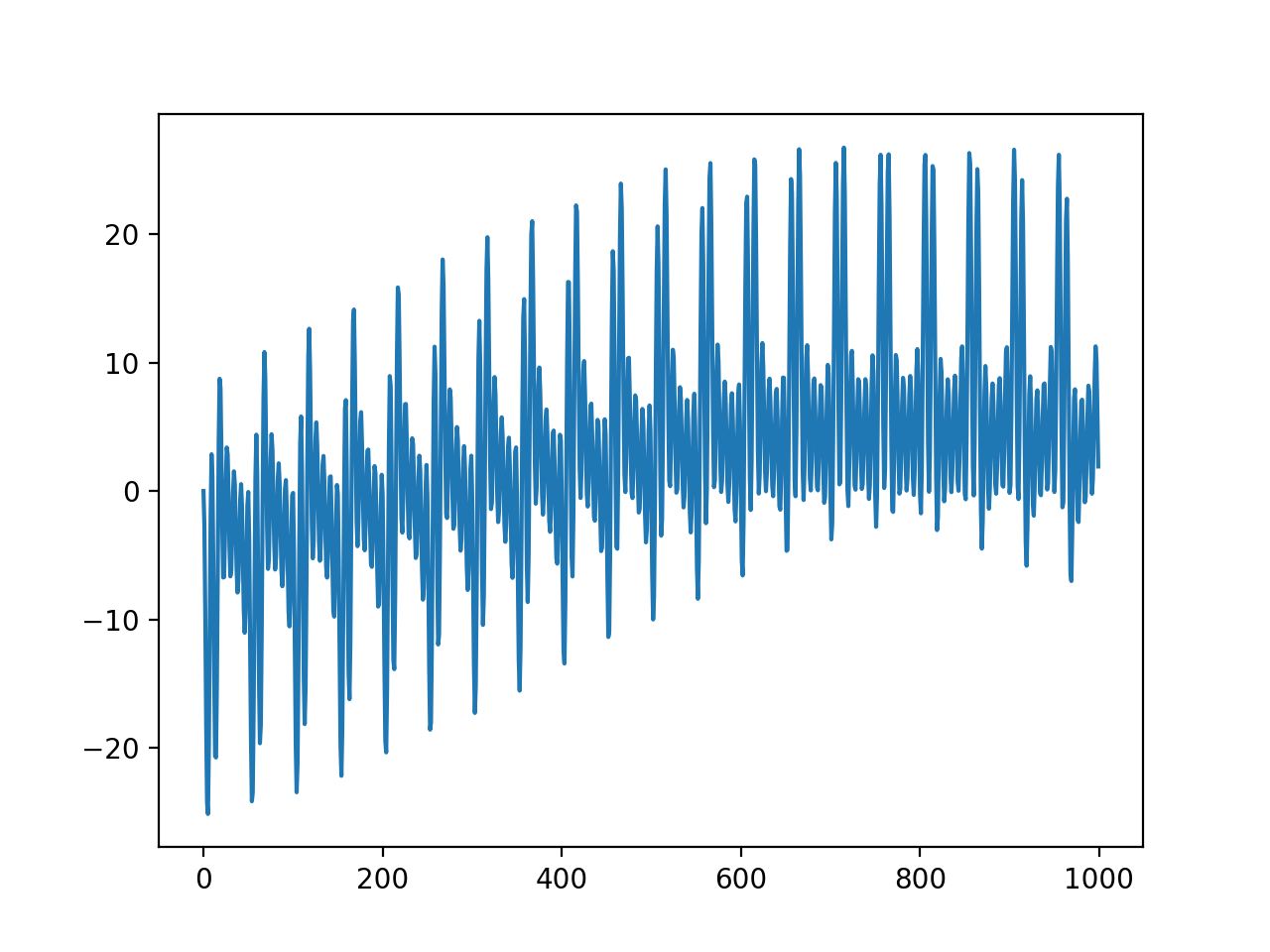
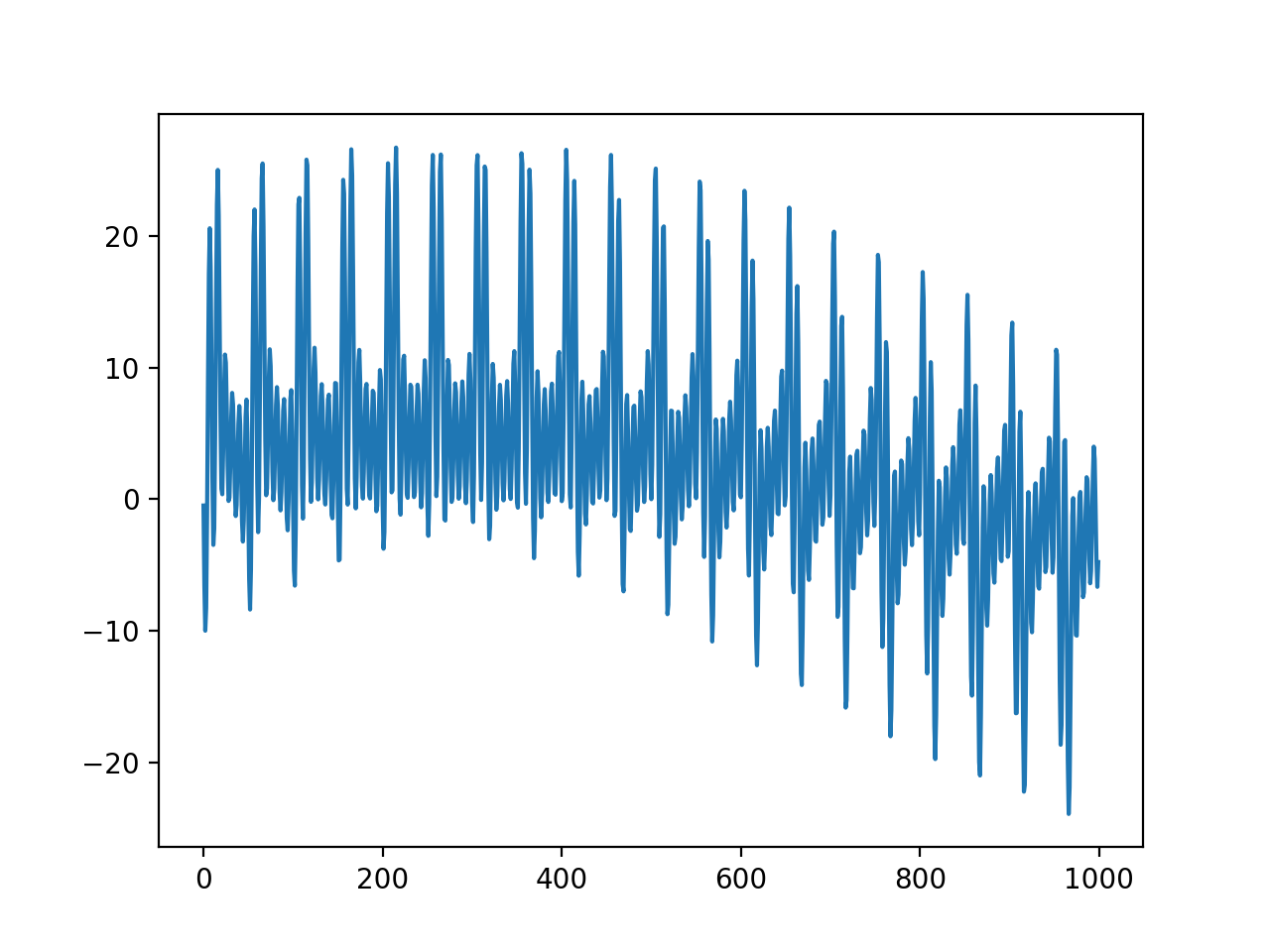
**import** numpy **as** np  
**import** matplotlib.pyplot **as** plt  
**import** math  
**import** mir  
**from** mir **import** Sinusoid  
  
sin1 = Sinusoid(Fs=1000, amp=5, freq=20, phase=5)  
sin2 = Sinusoid(Fs=1000, amp=10, freq=40, phase=10)  
sin3 = Sinusoid(Fs=1000, amp=15, freq=60, phase=15)  
N = len(sin1.data)  
  
signal = []  
**for** a **in** range(N):  
 signal.append(sin1.data[a]+sin2.data[a]+sin3.data[a])  
plt.plot(abs(np.fft.fft(signal)))  
plt.show()  
  
sin4 = Sinusoid(Fs=1000, amp=5, freq=20, phase=5)  
  
N = len(sin4.data)  
*#basic function with bin 60*bin = 60  
k = bin  
  
cosa = []  
sina = []  
  
**for** n **in** range (N):  
 theta = 2\*math.pi\*k\*n/N  
 cosa.append(math.cos(theta))  
 sina.append(math.sin(theta))  
*#point wise multiplication*ppc = []  
pps = []  
**print** len(signal)  
**print** len(cosa)  
**for** n **in** range(N):  
 ppc.append(cosa[n] \* signal[n])  
 pps.append(sina[n] \* signal[n])  
  
**print** np.arctan(sum (pps)/sum(ppc))  
plt.plot(ppc)  
plt.show()  
plt.plot(pps)  
plt.show()  
  
*#basic function with bin 60*bin = 60.5  
k = bin  
  
cosa = []  
sina = []  
  
**for** n **in** range (N):  
 theta = 2\*math.pi\*k\*n/N  
 cosa.append(math.cos(theta))  
 sina.append(math.sin(theta))  
*#point wise multiplication*ppc = []  
pps = []  
**print** len(signal)  
**print** len(cosa)  
**for** n **in** range(N):  
 ppc.append(cosa[n] \* signal[n])  
 pps.append(sina[n] \* signal[n])  
  
**print** np.arctan(sum (pps)/sum(ppc))  
plt.plot(ppc)  
plt.show()  
plt.plot(pps)  
plt.show()

Plot:

Plot of absolute magnitude of 3 sin waves



Plot of Sin and Cos with Closest DFT Bin



Plot of Sin and Cos with unrelated DFT Bin

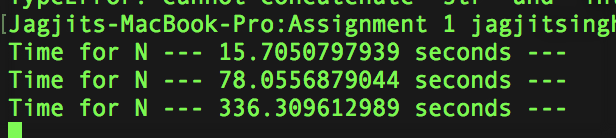
Part5:Finding Time

Code:

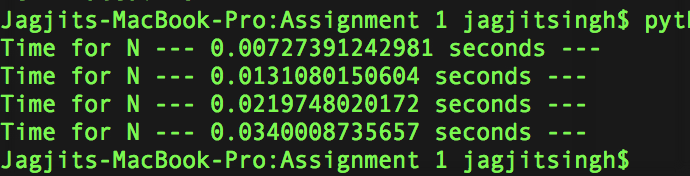
**import** numpy **as** np  
**import** matplotlib.pyplot **as** plt  
**import** math  
**import** mir  
**from** mir **import** Sinusoid  
**import** time  
start\_time = time.time()  
size = [256,512,1024,2048]  
  
**for** n1 **in** range(4):  
 **for** n2 **in** range(100):  
 *#get the sin wave* sin = Sinusoid(Fs=size[n1])  
 *#differentiate real and imaginary components* datar = sin.data.real  
 datai = sin.data.imag  
 N = len(datar)  
 outputr = []  
 outputi = []  
 **for** k **in** range(N):  
 sumr = 0  
 sumi = 0  
 **for** n **in** range(N):  
 sumr += datar[n]\*math.cos(2\*math.pi\*k\*n/N)+datai[n]\*math.sin(2\*math.pi\*k\*n/N)  
 sumi += -datar[n]\*math.sin(2\*math.pi\*k\*n/N)+datai[n]\*math.cos(2\*math.pi\*k\*n/N)  
 outputr.append(sumr)  
 outputi.append(sumi)  
 outputp = []  
 *#final output* **for** a **in** range(N):  
 outputp.append(math.sqrt((outputr[a]\*outputr[a])+(outputi[a]\*outputi[a])))  
 *#plot output* **print**(**"Time for N --- %s seconds ---"** % (time.time() - start\_time))  
  
start\_time = time.time()  
size = [256,512,1024,2048]  
output = []  
**for** n1 **in** range(4):  
 **for** n2 **in** range(100):  
 *#get the sin wave* sin = Sinusoid(Fs=size[n1])  
 output = sin.dft()  
 **print**(**"Time for N --- %s seconds ---"** % (time.time() - start\_time))

Results:

Time for 256, 512 and 1024 size with 100 iterations. (Still running for rest of the two)



Time taken by the Built in function



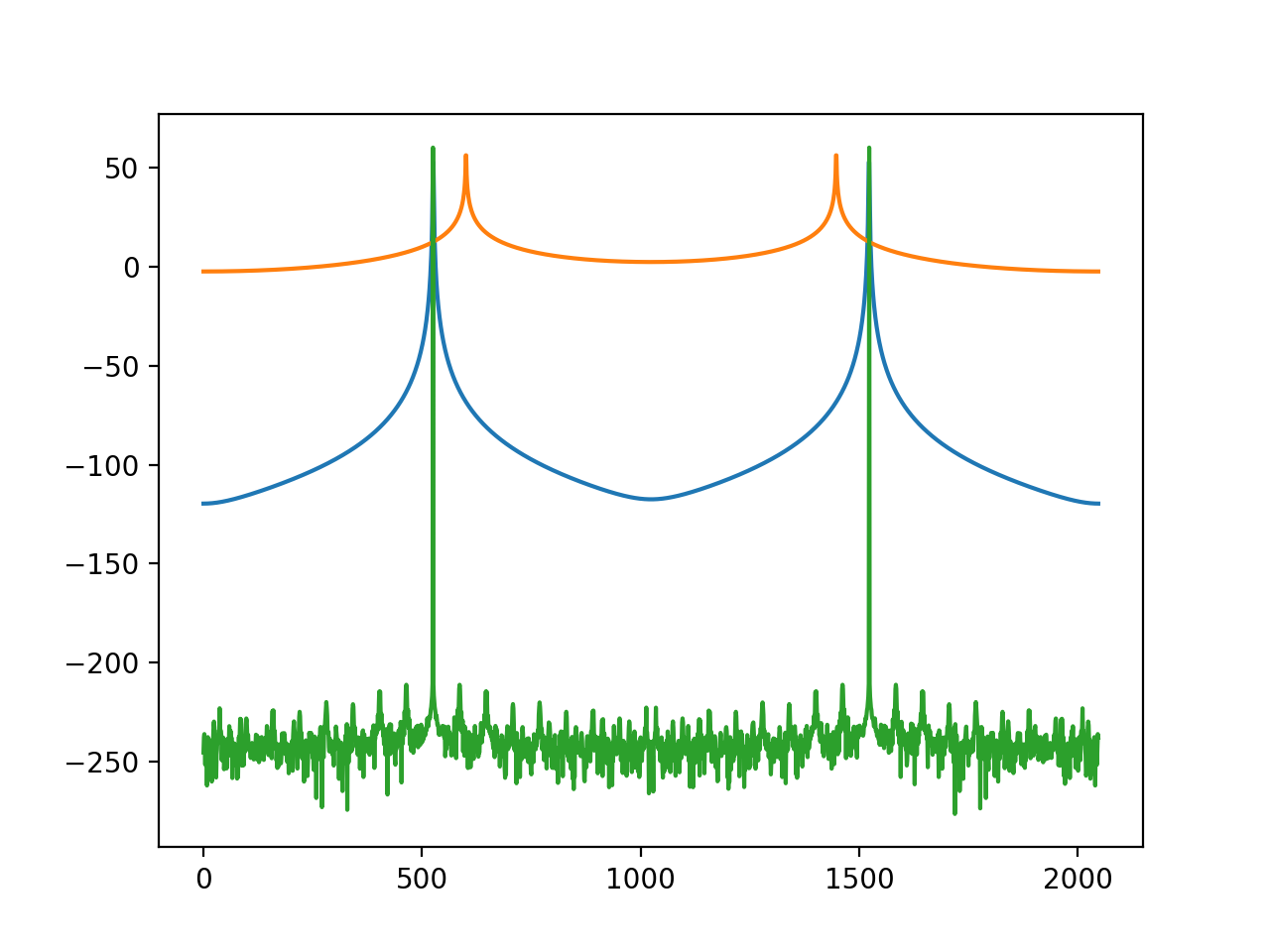
**Question2:**

Part 1: Hanning Windows

Code:

**import** numpy **as** np  
**import** matplotlib.pyplot **as** plt  
**import** scipy.fftpack  
**import** math  
**import** mir  
**from** mir **import** Sinusoid  
  
N = 2048  
  
bin = 525  
k = bin  
  
sina = []  
  
**for** n **in** range (N):  
 theta = 2\*math.pi\*k\*n/N  
 sina.append(math.sin(theta))  
  
N = 2048  
  
bin = 600.5  
k = bin  
  
sinaa = []  
  
**for** n **in** range (N):  
 theta = 2\*math.pi\*k\*n/N  
 sinaa.append(math.sin(theta))  
  
N = 2048  
  
bin = 525.5  
k = bin  
  
sinaaa = []  
  
**for** n **in** range (N):  
 theta = 2\*math.pi\*k\*n/N  
 sinaaa.append(math.sin(theta))  
  
hann = np.hanning(2048)  
wave =[]  
**for** n **in** range (N):  
 wave.append(hann[n]\*sinaaa[n])  
  
dft3 = np.fft.fft(wave)  
dft2 = np.fft.fft(sinaa)  
dft = np.fft.fft(sina)  
dft3t= abs(dft3)  
dft2t= abs(dft2)  
dftt= abs(dft)  
plt.plot(20\*np.log10(dft3t))  
plt.plot(20\*np.log10(dft2t))  
plt.plot(20\*np.log10(dftt))  
plt.show()

Plot:



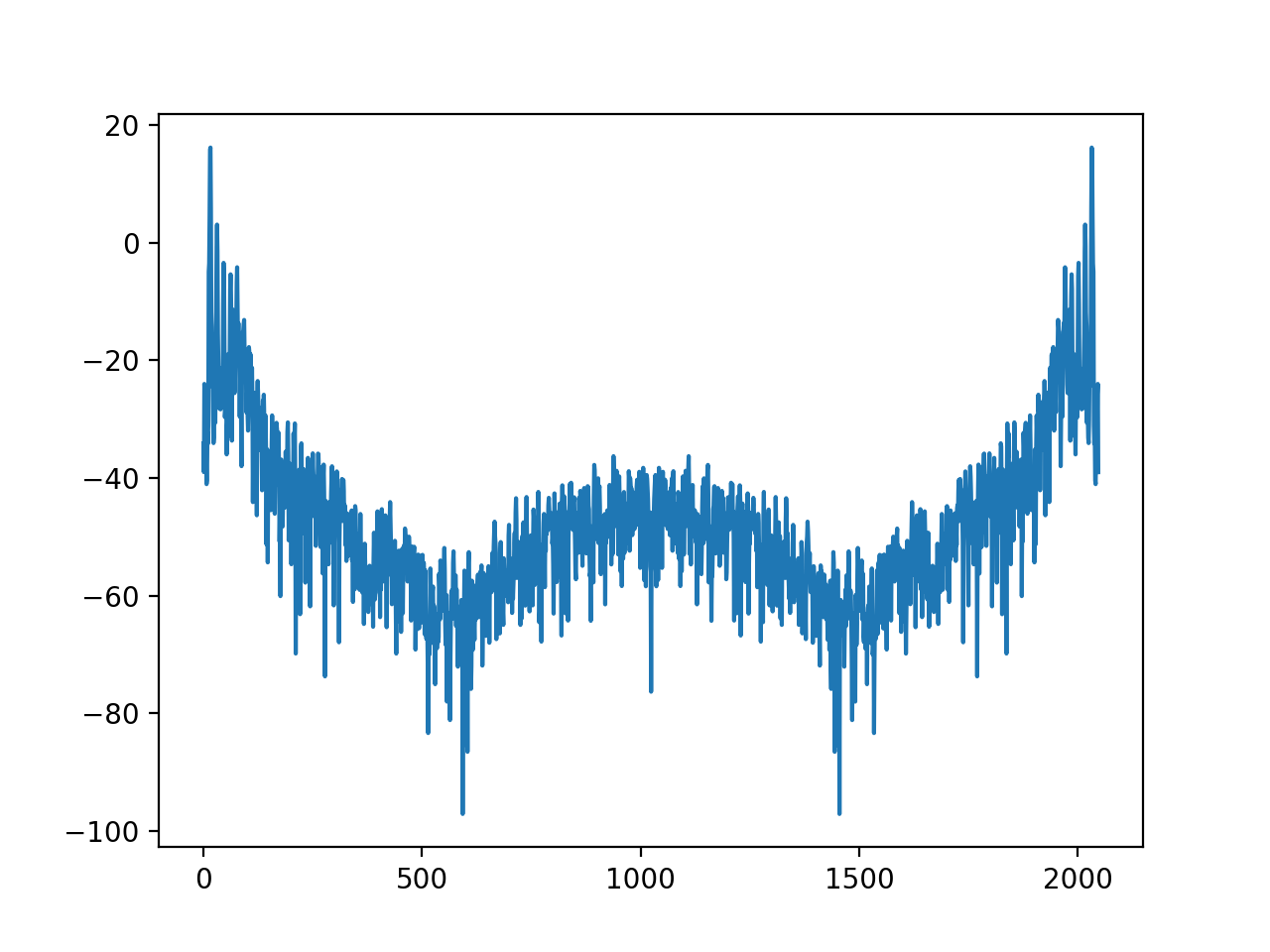
Two plots coincide that are within the bin and the other on the bin is different in Orange color.

Part2: Single Note Sound Analysis

Code:

**import** numpy **as** np  
**import** matplotlib.pyplot **as** plt  
**import** scipy.fftpack  
**import** math  
**import** mir  
**from** mir **import** Sinusoid  
**from** mir **import** Signal  
  
sin = Signal()  
sin.wav\_read(**'2.wav'**)  
wave = sin.data  
wave = wave[0:2048]  
  
hann = np.hanning(2048)  
wave1 =[]  
N = 2048  
**for** n **in** range (N):  
 wave1.append(hann[n]\*wave[n])  
  
dft = np.fft.fft(wave1)  
dftt= abs(dft)  
plt.plot(20\*np.log10(dftt))  
plt.show()

Plot



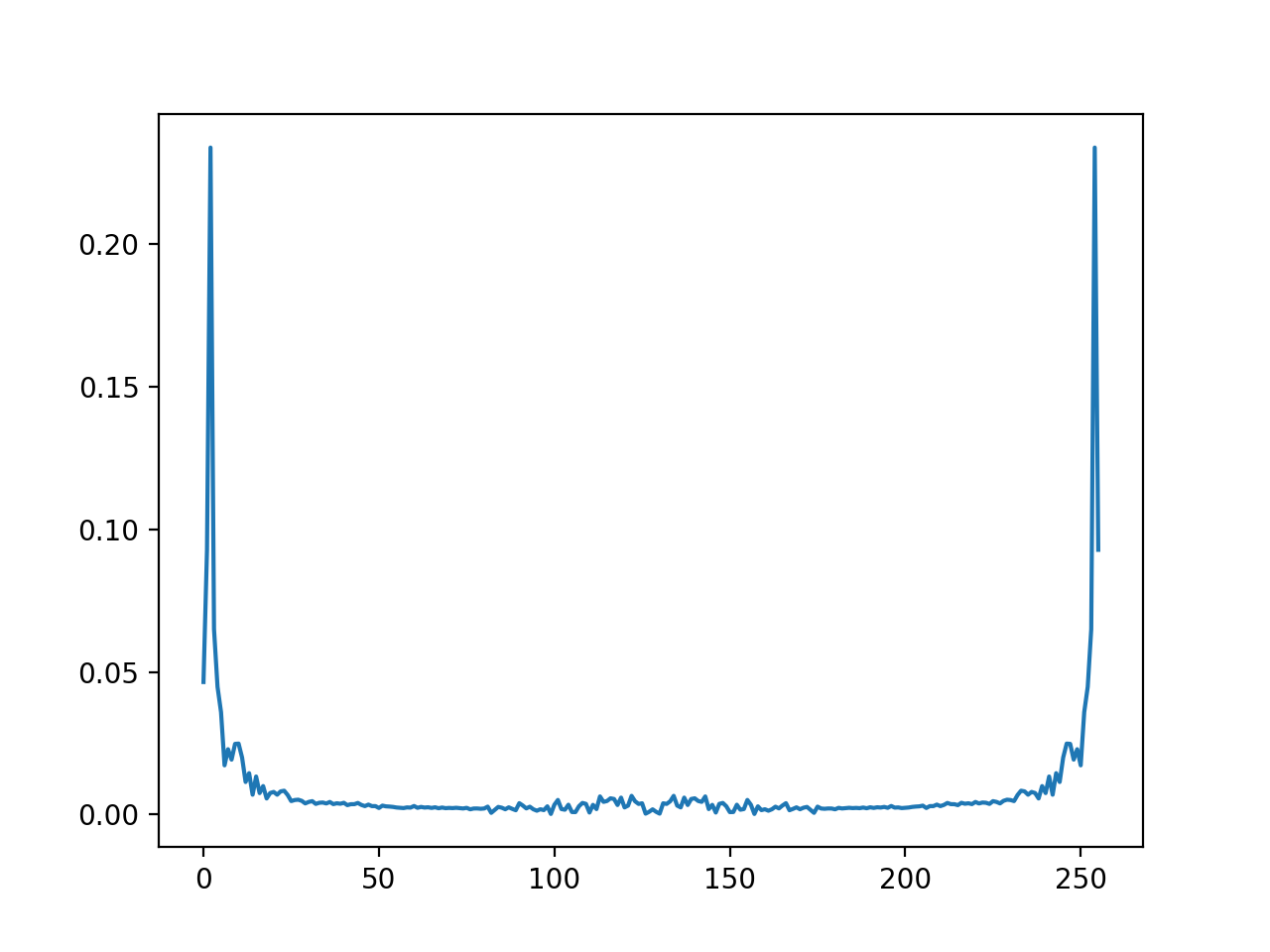
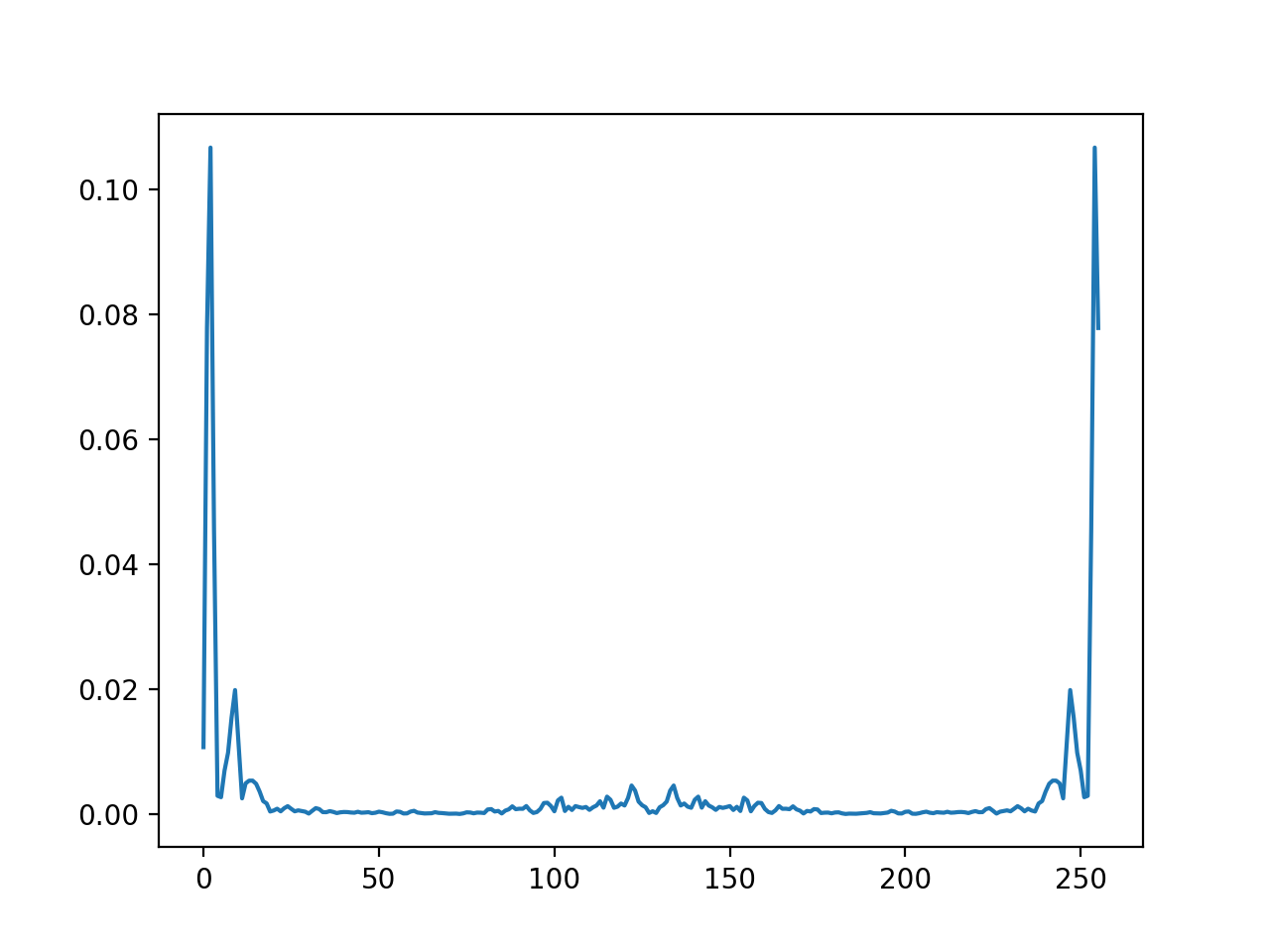
The resultant plot after the windowing will have reduced spectral leakage.

Part 3: Zero Padding Array

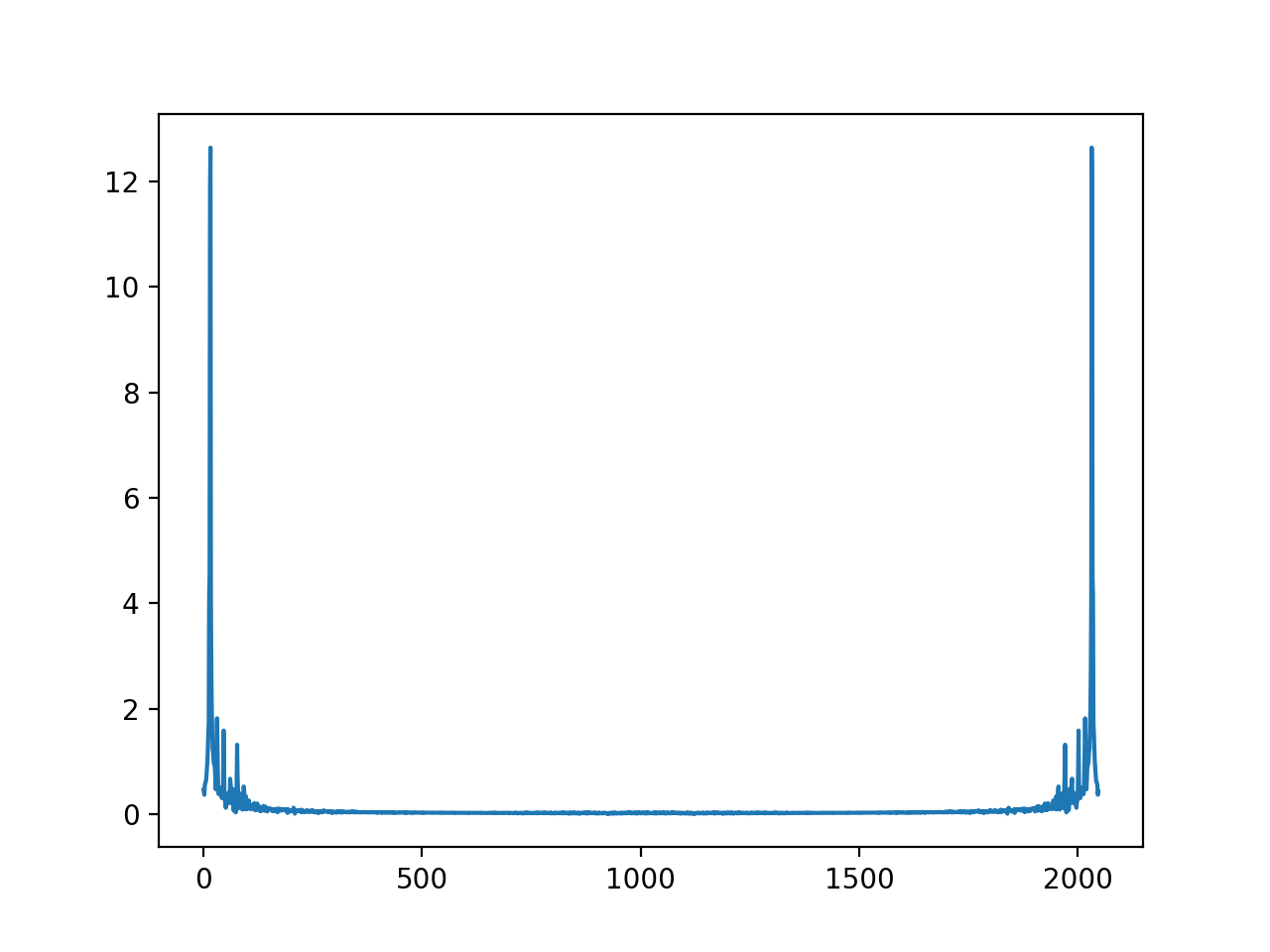
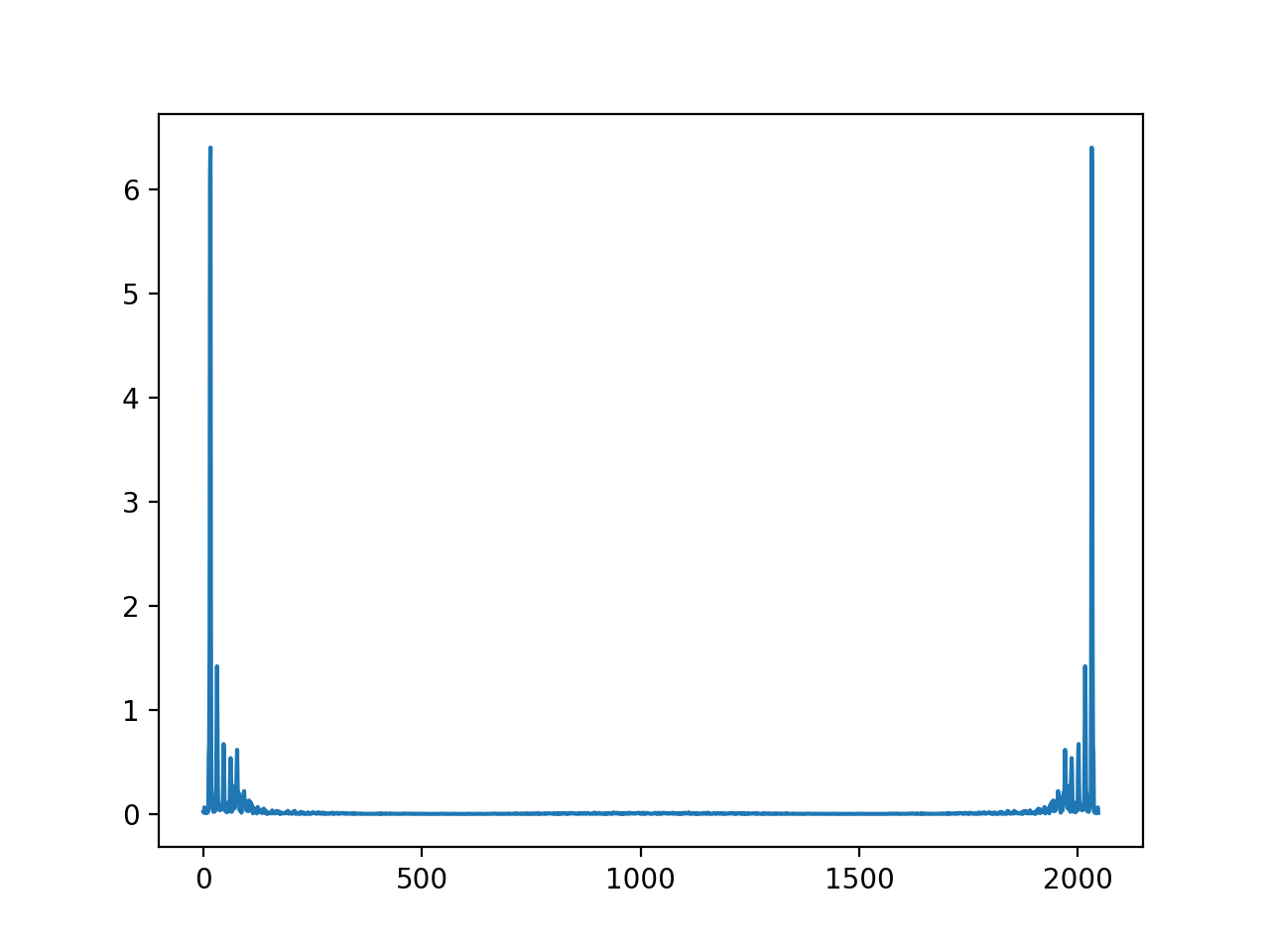
Code:

**import** numpy **as** np  
**import** matplotlib.pyplot **as** plt  
**import** scipy.fftpack  
**import** math  
**import** mir  
**from** mir **import** Sinusoid  
**from** mir **import** Signal  
  
sin = Signal()  
sin.wav\_read(**'2.wav'**)  
wave = sin.data  
wave = wave[0:2048]  
  
hann = np.hanning(2048)  
wave1 =[]  
N = 2048  
**for** n **in** range (N):  
 wave1.append(hann[n]\*wave[n])  
  
dft = np.fft.fft(wave1)  
dftt= abs(dft)  
plt.plot(dftt)  
plt.show()  
  
dft1 = np.fft.fft(wave)  
dft1t= abs(dft1)  
plt.plot(dft1t)  
plt.show()  
  
wave = wave[0:256]  
hann = np.hanning(256)  
wave1 =[]  
N = 256  
**for** n **in** range (N):  
 wave1.append(hann[n]\*wave[n])  
  
dft = np.fft.fft(wave1)  
dftt= abs(dft)  
plt.plot(dftt)  
plt.show()  
  
dft1 = np.fft.fft(wave)  
dft1t= abs(dft1)  
plt.plot(dft1t)  
plt.show()  
  
list = []  
N = 2048  
**for** n **in** range (N):  
 **if** n < 255:  
 list.append(wave[n])  
 **else**:  
 list.append(**'0'**)  
  
dft1 = np.fft.fft(list)  
dft1t= abs(dft1)  
plt.plot(dft1t)  
plt.show()  
  
hann = np.hanning(2048)  
wave1 = [x\*y **for** x,y **in** zip([hann],list)]  
  
  
  
  
dft = np.fft.fft(wave1)  
dftt= abs(dft)  
plt.plot(dftt)  
plt.show()

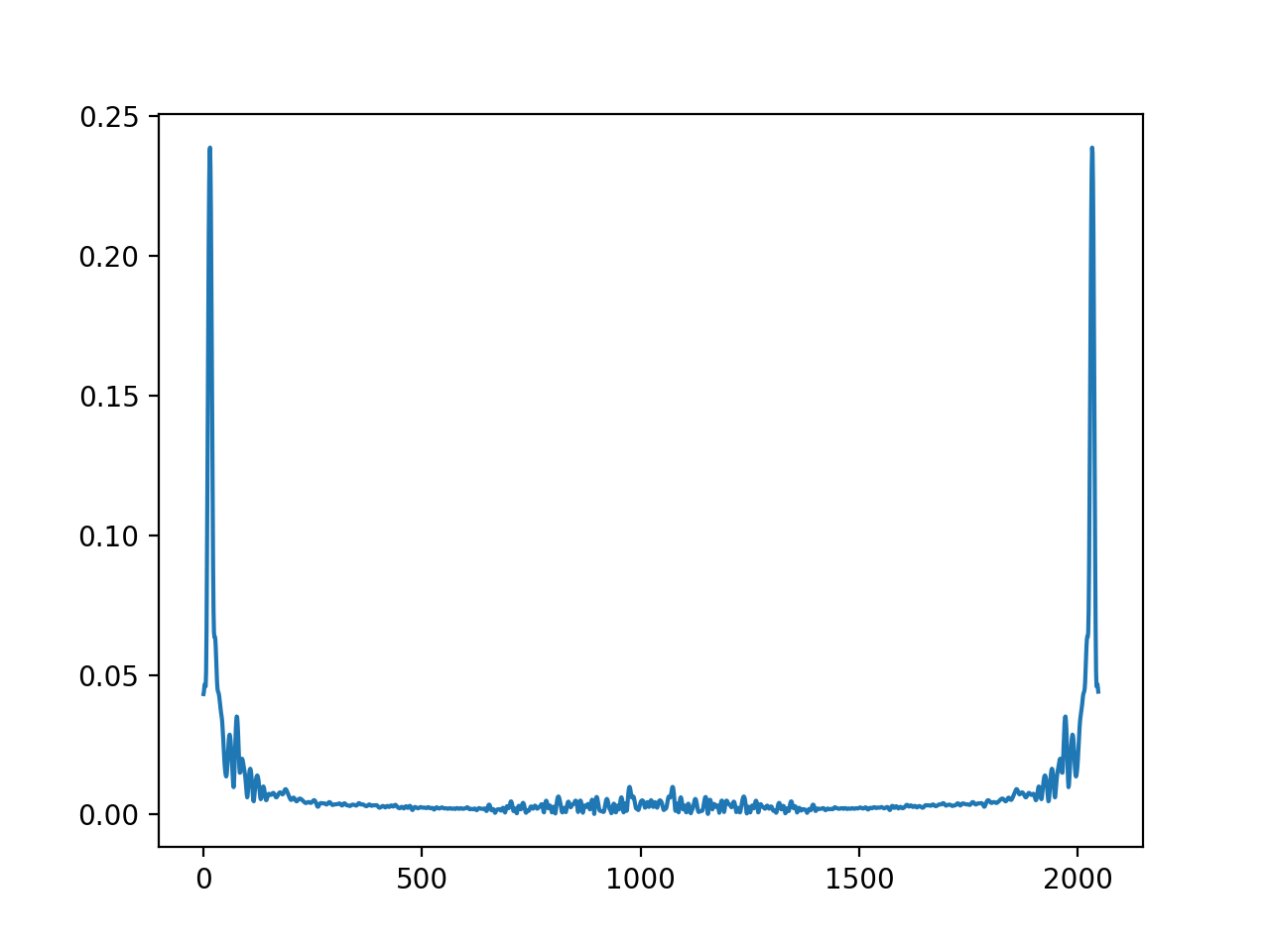
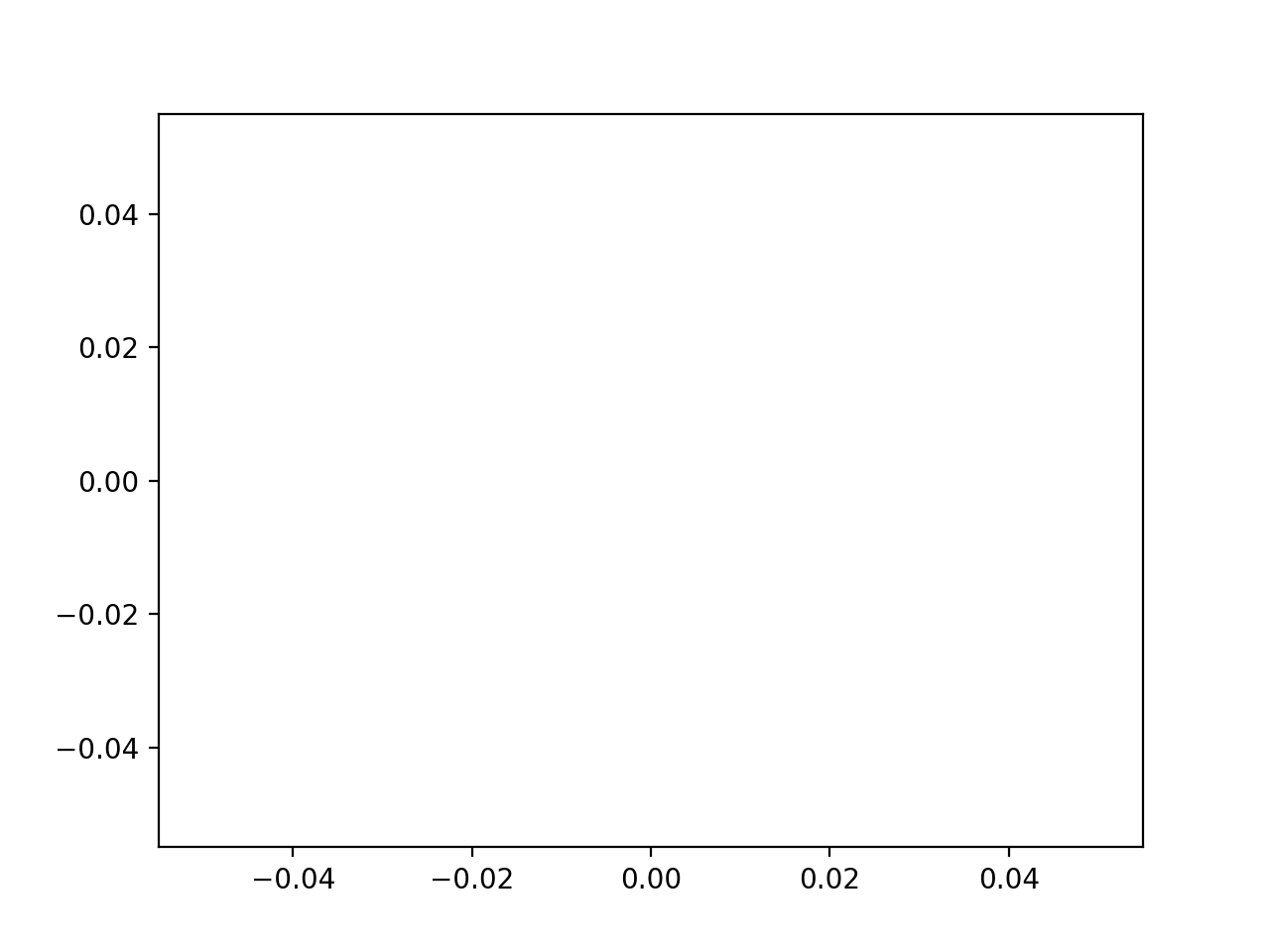
Plot:



With and without window 256



With and without Window 2048



With and without window Zero Padded

Effects – Initially the plots with higher sample size are dense and the plots with lower

Sample size are widely places, with hanning we avoided the spectral leakage in both the cases and the sound was soften by the hanning window effect. When we made an 256 sample signal with Zero padding the windowing showed a straight wave, that means the window size may be higher than 256 so it didn’t came in.

**Question3:**

For purpose of answering the question, I have selected the GTA San Andreas (Theme Song) - <https://www.youtube.com/watch?v=W4VTq0sa9yg>

Part -1

Time Scales – Short, Middle and Long Term

Dimensions – Timber, Orchestration, Acoustics, Rhythm, Melody, Harmony and Structure.

The opening of the music starts with Chimes followed by repetitive drum beats and piano.

**Rhythm** – The music has a time scale – Middle Term, where a sound is being repeating multiple times. The entire Sequence is coming back again.

**Melody** – The music after the initial opening has a middle term melody repetition, that is high and low drum beats.

**Timber** – There is a lot of classification in Timber (Short term) when the opening comes from the Chimes.

**Structure** – When seen for the long term the music has a lot of time scale differences.

Part – 2

**Casual Users –** Since casual users like listening and collecting music their request will be like (Finding the song that sounds like this) (I like this one, what other songs I may admire)(I have to organize my collection)

**Professional Users –**  They may be using music in advertisement and production, they need music for an application, their request information will be like (Given a rhythm give me the music that matches it) (I need this kind of soundtrack with this instrument)

**Music Scholars –** These kinds of people and interested in studying music, developing music (analysis of music tools)(I don’t want the music but the pitch it was sung in)

Part – 3

**Query By humming** – Humming is a way where in a casual user will use a combination of whistling and singing to produce a melody. Then that melody will be used to search/query the sound/music database to find the full version of the song/music corresponding to the melody.

**Query by Example –**  When a part of the music/example of the music is used to query the database to retrieve the full music it is termed as query by example. This will avoid mismatching of the retrieved music.

Part- 4

**Music XML**- This is a type of music representation which used XML as portable format for exchange of information. The best part is the industry wide support including the open-source and commercial projects. Since we can write XML in Latex so we can represent this using the latex.

**MIDI**- it stands for Musical Instrument Digital Interface. This is format where we do the data exchange within digital instruments. This requires sequential information transfers. SMF-Standard Midi File will be used for the future information transfer.

Part-5

Since this entire document deals with MIR and we are doing MIR for the end users, for me that’s the best part of explaining various users that uses MIR. There are three levels of users starting from the most basic going to the most complex ones. The casual users are the one who need music for fun and hobbits. The second level is for the professional student that may need music for their professional purposes, they are commercial users of music. The music scholars are most advanced users; they are one who study music and its composition. It also includes musicians.

Part-6

Paper – 1: Indexing Music Databases Using Automatic Extraction of Frequent Phrases

In the field of MIR the extraction of music for the database is a crucial and complex task, if we wish to have the offline database then storing the entire indexing file structure take a lot of memory, The best possible way to improve this is using the phrases instead of full content. Repetition is an important property of music, if we utilize that property and only store the most occurring phrase instead of entire file we can save a lot of space. We already have efficient text mining methods that can be applied with minor modifications.

The paper is based of Irish folk music, and concludes with a text-mining based indexing algorithm for music which extracts repeating patterns from the music.

Paper -2: A Multimodal Framework for Music Inputs

Music waves have one excellent property of being multimodal, since the previous research in this field have done a lot of work with the indexing and fetching of data but the technique outlined in this paper is essentially for the query. Since when users interact with computer they want the ease of use, that means that it should be possible for them to query the music database in any form they wish to, example – sing, play or sent symbols.

Since this idea is a new step in terms of the Human Computer Interaction (HCI), which is one of the most emerging topic and further it gives the user a flexibility to interact with the music system.

Paper - 3: Query By Humming -- Musical Information Retrieval in an Audio Database

If we have all the text in the database, we need to query it using a text, but what if we have a database full of music and audio files, it would be great if we could just hum the melody and fetch the music of our choice. Efficient pattern matching is an important aspect of this paper. It may be possible not to return just one but multiple matching music files.

It is equally essential to have an optimized version of query in the system. Since taking extra time will result in the user losing interest in the system. The authors use pith as the medium of detection between the hummed melody and original sound wave in the database.

**Question4:**

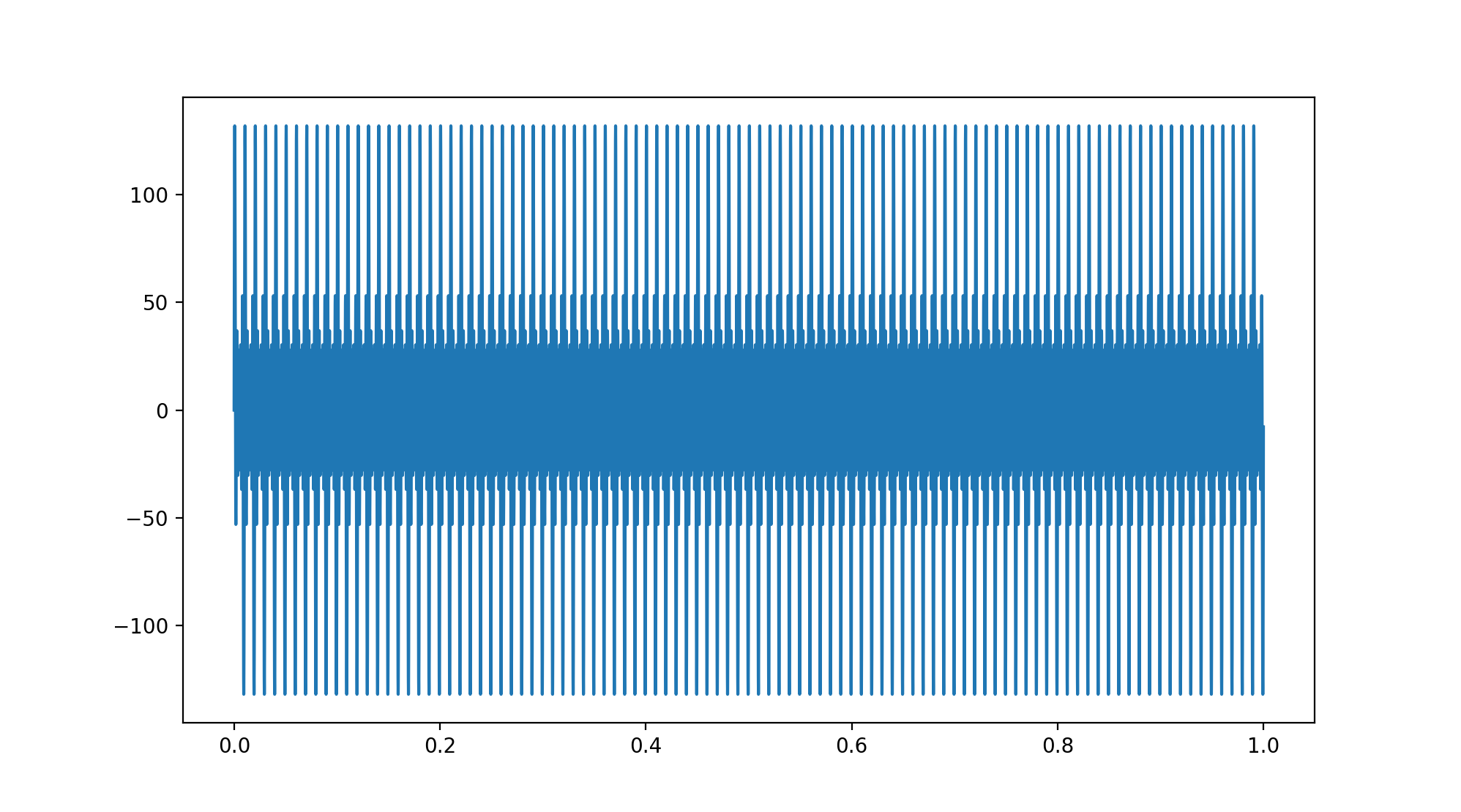
Part - 1

Additive Synthesis Instrument: The actual concept is to have n number of sin wave generator using the defined frequency and helper function – Mixture ()

Code:

**import** numpy **as** np  
**import** matplotlib.pyplot **as** plt  
**import** math  
**import** mir  
**from** mir **import** Sinusoid  
**from** mir **import** Signal  
**from** mir **import** Mixture  
  
*#Declare the frequency and amplitude array*freq = [100,200,300,400,500]  
amp = [10,20,30,40,50]  
*#Declare the empty list*list = []  
*#Making a list of all 5 sinusoids***for** n **in** range (5):  
 sin = Sinusoid(amp=amp[n], freq=freq[n])  
 list.append(sin)  
*#Making a mixture using the helper function in mir.py*mix = Mixture(\*list)  
*#Plot the mixture of all 5 waves*mix.plot()

Plot:



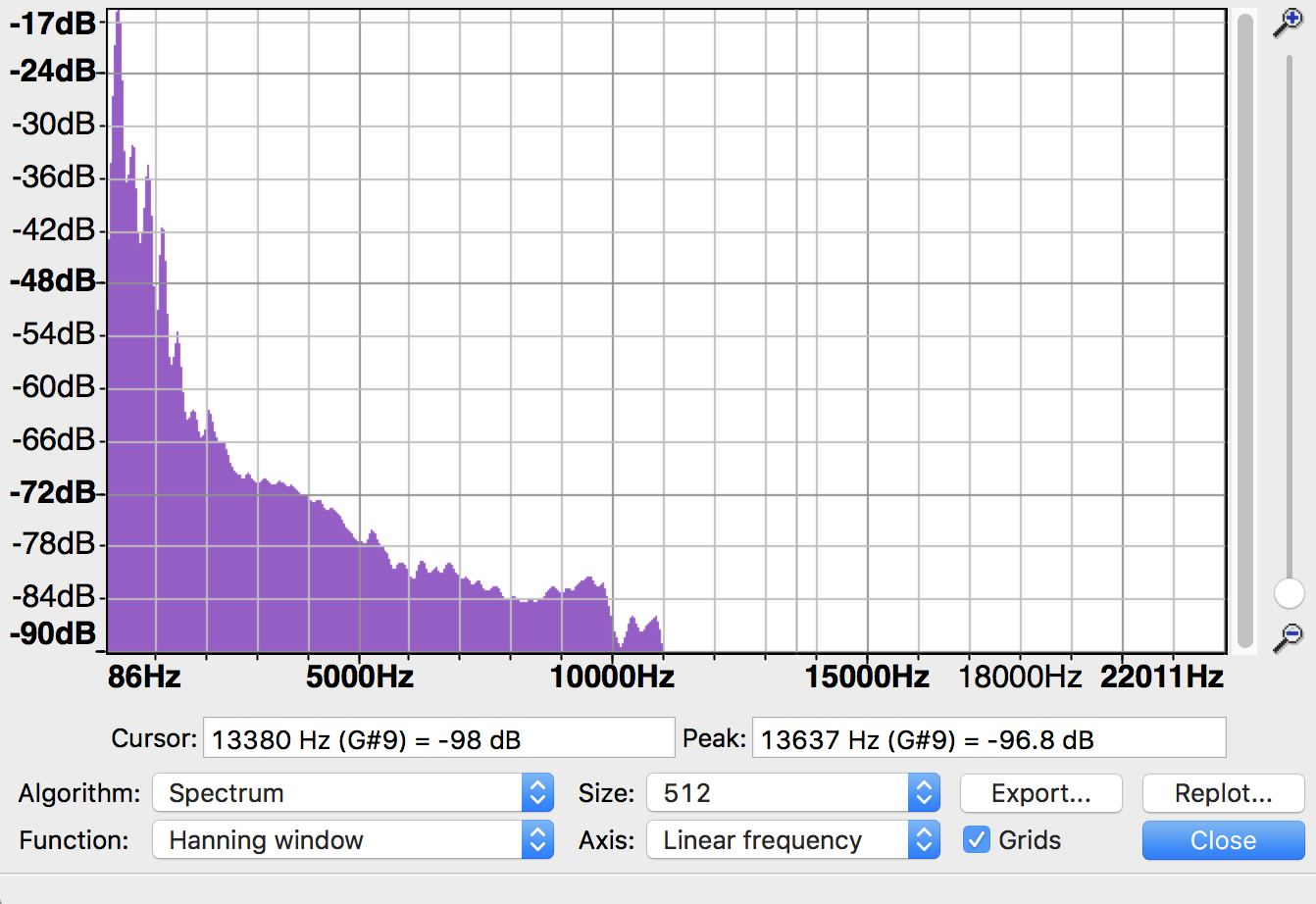
Part - 2

Different Sounds of Same Pitch – passing them to additive Instrument.

Sound 1 = Flute D Tone

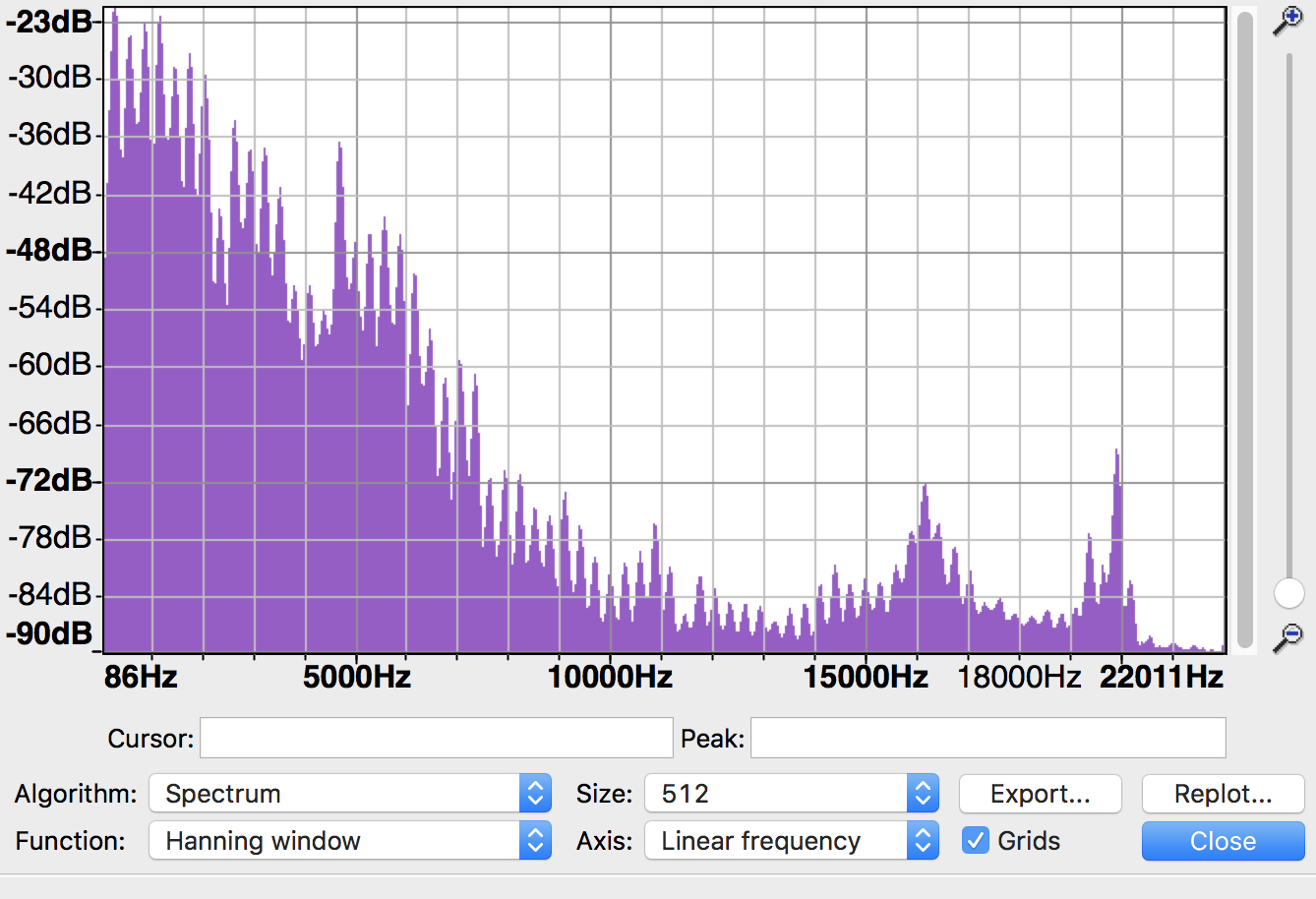
Sound 2 = Violin D Tone

Spectrum Analysis Sound 1



freq = [292,563,869,1160,1439,1767,2064,2828,3159,3447]  
amp = [0.15488166189124816,0.03162277660168379,0.019275249131909367,0.008511380382023767,0.0023173946499684774,0.0007328245331389037,0.0007673614893618193,0.0003388441561392024,0.00031988951096913973,0.00030549211132155157]

Spectrum Analysis Sound 2

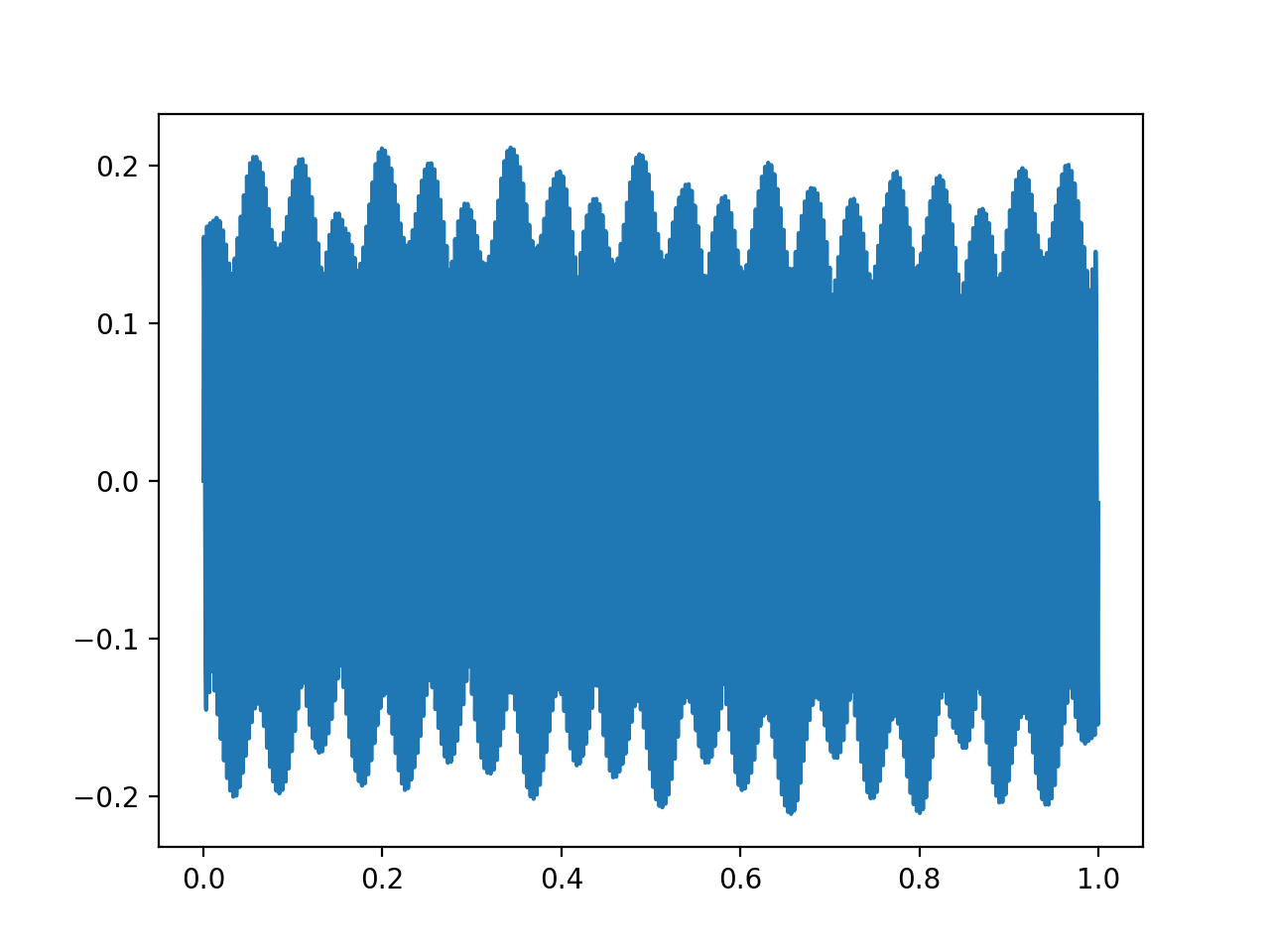


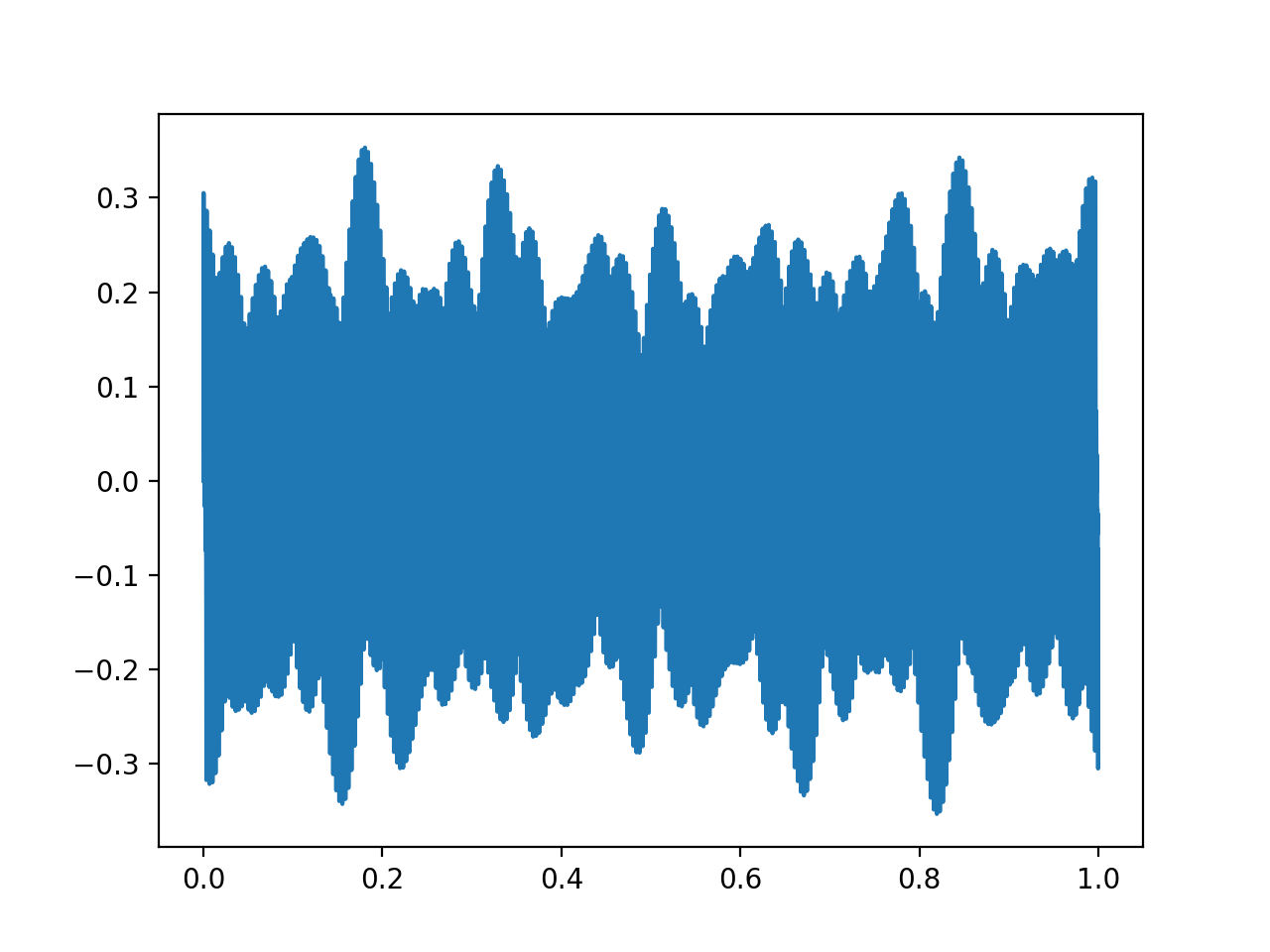
freq = [295,572,882,1165,1460,1764,2052,2343,2633,2931]  
amp = [0.0812830516164099,0.06165950018614822,0.06095368972401691,0.07852356346100718,0.03890451449942807,0.042169650342858224,0.033884415613920256,0.006760829753919818,0.022130947096056376,0.01428893958511103]

Code:

**import** numpy **as** np  
**import** matplotlib.pyplot **as** plt  
**import** math  
**import** mir  
**from** mir **import** Sinusoid  
**from** mir **import** Signal  
**from** mir **import** Mixture  
  
*#Frequency and Amplitude for Sound 1*freq = [292,563,869,1160,1439,1767,2064,2828,3159,3447]  
amp = [0.15488166189124816,0.03162277660168379,0.019275249131909367,0.008511380382023767,0.0023173946499684774,0.0007328245331389037,0.0007673614893618193,0.0003388441561392024,0.00031988951096913973,0.00030549211132155157]  
  
list = []  
**for** n **in** range (10):  
 sin = Sinusoid(amp=amp[n], freq=freq[n])  
 list.append(sin)  
*#make signal mixture*mix1 = Mixture(\*list)  
  
sig1 = Signal(data=mix1.data)  
*#output new audio file*sig1.wav\_write(**'flu\_new.wav'**,normalize=False)  
  
mix1.plot()  
*#Frequency and Amplitude for Sound 2*freq = [295,572,882,1165,1460,1764,2052,2343,2633,2931]  
amp = [0.0812830516164099,0.06165950018614822,0.06095368972401691,0.07852356346100718,0.03890451449942807,0.042169650342858224  
 ,0.033884415613920256,0.006760829753919818,0.022130947096056376,0.01428893958511103]  
  
list1 = []  
**for** n **in** range (10):  
 sin = Sinusoid(amp=amp[n], freq=freq[n])  
 list1.append(sin)  
*#make signal mixture*mix2 = Mixture(\*list1)  
  
sig2 = Signal(data=mix2.data)  
*#output new audio file*sig2.wav\_write(**'voi\_new.wav'**,normalize=False)  
  
mix2.plot()

Plot :





Effect – Its hard to tell which sound is which, we cannot make out the sound because its like an artificial sound without the musical instrument properties. No wood, metal noise.

Its just like a plain artificial note.

Part 3

MIDI note Numbers to Frequency Conversion – Used 4 MIDI numbers converted them to corresponding waves and then used the mixture to make a melody out of them.

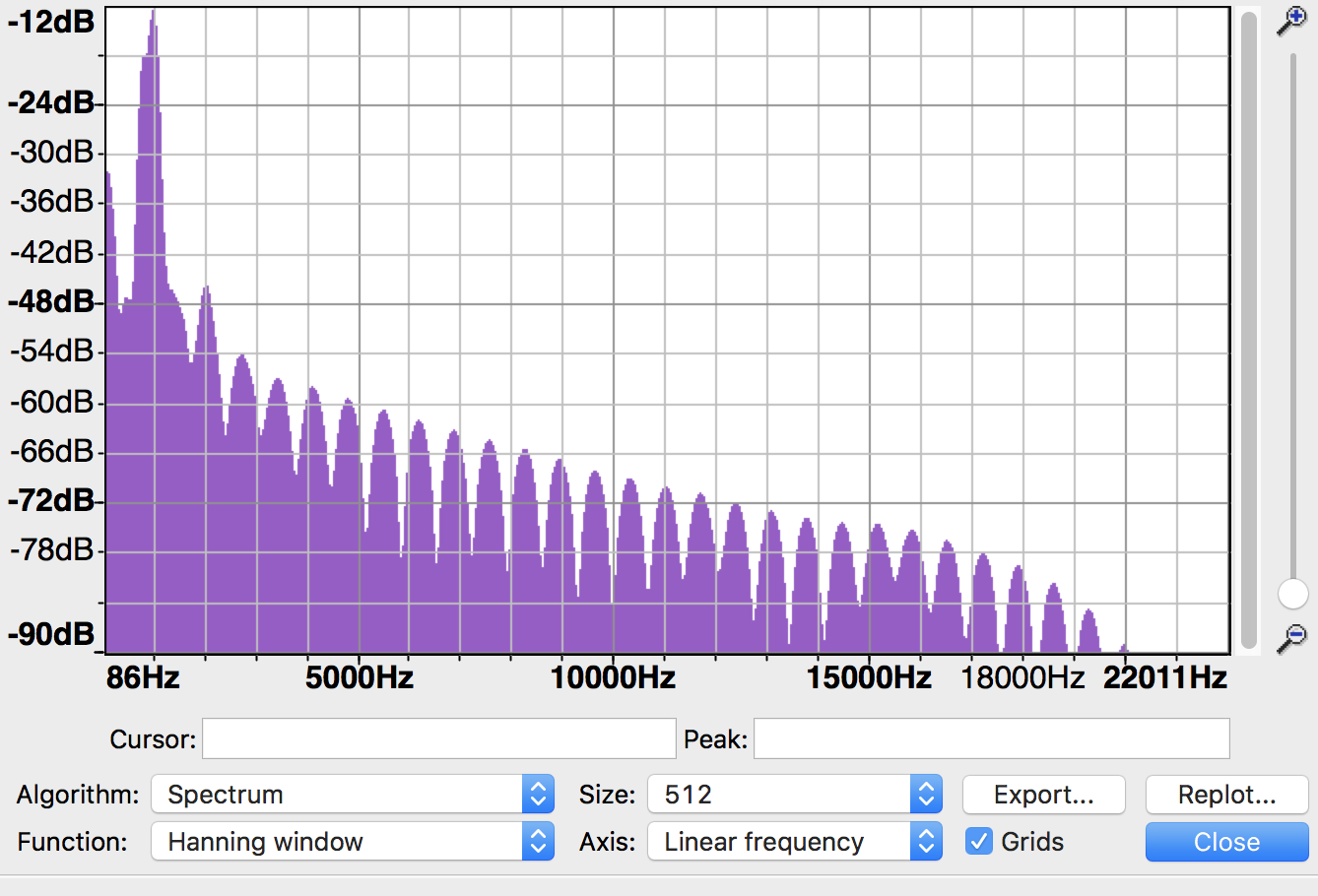
Code:

**import** numpy **as** np  
**import** matplotlib.pyplot **as** plt  
**import** math  
**import** mir  
**from** mir **import** Sinusoid  
**from** mir **import** Signal  
**from** mir **import** Mixture  
  
*#midi notes from 0-119*note = [7,8,9,10]  
freq = []  
list = []  
**for** n **in** range (4):  
 freq.append(440\*pow((note[n]-57)/12,2))  
 sin = Sinusoid(freq=freq[n])  
 list.append(sin)  
  
*#make signal mixture*mix1 = Mixture(\*list)  
  
sig1 = Signal(data=mix1.data)  
*#output new audio file*sig1.wav\_write(**'melody.wav'**,normalize=False)  
  
mix1.plot()

Melody File - <https://github.com/JagjitUvic/MIR/blob/master/Assignment%201/melody.wav>

Recorded Audio File- <https://github.com/JagjitUvic/MIR/blob/master/Assignment%201/Question4audio.wav>

Audacity Spectrum –



**Question5:**

Part – 1

Use the helping function provided in the file to read and write the audio file.

Code:

**import** numpy **as** np  
**import** matplotlib.pyplot **as** plt  
**import** scipy.fftpack  
**import** math  
**import** mir  
**from** mir **import** Sinusoid  
**from** mir **import** Signal  
  
sig = Signal()  
*#Read audio File*sig.wav\_read(**'Ques5Input.wav'**)  
wave = sig.data  
wavefft = np.fft.fft(wave)  
waveifft = np.fft.ifft(wavefft)  
*#Writing the audio file*sig = Signal(data=waveifft)  
sig.wav\_write(**'Ques5Output.wav'**)

Result: Both the audio files are identical.

Input Audio –https://github.com/JagjitUvic/MIR/blob/master/Assignment%201/Ques5Input.wav

Output Audio - https://github.com/JagjitUvic/MIR/blob/master/Assignment%201/Ques5Output.wav

Part – 2

Code:

**import** numpy **as** np  
**import** matplotlib.pyplot **as** plt  
**import** scipy.fftpack  
**import** math  
**import** mir  
**from** mir **import** Sinusoid  
**from** mir **import** Signal  
  
sig = Signal()  
*#Read audio File*sig.wav\_read(**'Ques5Input.wav'**)  
wave = sig.data  
wavefft = np.fft.fft(wave)  
length = len(wavefft)  
magnitude = []  
phase = []  
**for** n **in** range (length):  
 magnitude.append(np.absolute(wavefft[n]))  
 phase.append(np.angle(wavefft[n]))  
*#phase goes random*phase = np.random.randn(length)  
sigC = []  
**for** n **in** range (length):  
 val1 = magnitude[n]\*math.cos(phase[n])  
 val2 = magnitude[n]\*math.sin(phase[n])  
 sigC.append(np.complex(val1,val2))  
*#find inverse transform*waveifft = np.fft.ifft(sigC)  
*#write back the signal object*sig = Signal(data=waveifft)  
sig.wav\_write(**'Ques5\_2Output.wav'**)

Output File - <https://github.com/JagjitUvic/MIR/blob/master/Assignment%201/Ques5_2Output.wav>

Effects – the sound lost its original voice and Timber, it’s now just a rough ended music. After applying different window size we can hear distortion at the window size ends.

Part – 3

Code:

**import** numpy **as** np  
**import** matplotlib.pyplot **as** plt  
**import** scipy.fftpack  
**import** math  
**import** mir  
**from** mir **import** Sinusoid  
**from** mir **import** Signal  
  
sig = Signal()  
*#Read audio File*sig.wav\_read(**'Ques5Input.wav'**)  
wave = sig.data  
wavefft = np.fft.fft(wave)  
length = len(wavefft)  
magnitude = []  
phase = []  
**for** n **in** range (length):  
 magnitude.append(np.absolute(wavefft[n]))  
 phase.append(np.angle(wavefft[n]))  
*#fetching 4 high values of magnitude*index = []  
value = []  
**for** n **in** range (4):  
 index.append(np.argmax(magnitude))  
 value.append(np.amax(magnitude))  
 magnitude[np.argmax(magnitude)] = 0  
*#setting 4 high values rest all 0***for** n **in** range (length):  
 **if** n == index[0]:  
 magnitude[n] = value[0]  
 **elif** n == index[1]:  
 magnitude[n] = value[1]  
 **elif** n == index[2]:  
 magnitude[n] = value[2]  
 **elif** n == index[3]:  
 magnitude[n] = value[3]  
 **else**:  
 magnitude[n] = 0  
sigC = []  
**for** n **in** range (length):  
 val1 = magnitude[n]\*math.cos(phase[n])  
 val2 = magnitude[n]\*math.sin(phase[n])  
 sigC.append(np.complex(val1,val2))  
*#find inverse transform*waveifft = np.fft.ifft(sigC)  
*#write back the signal object*sig = Signal(data=waveifft)  
sig.wav\_write(**'Ques5\_3Output.wav'**)

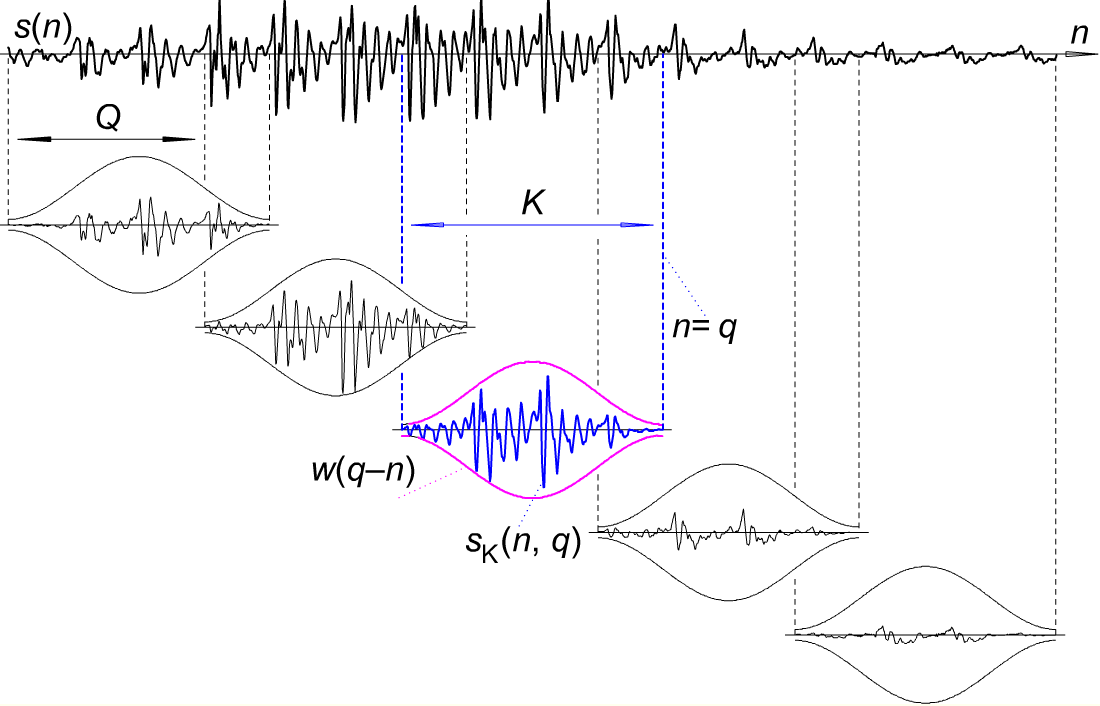
Effects - The output audio has 4 places where it gave different sound than the usual straight sound in the previous cases. There were high nodes at the corresponding high magnitudes that were available in the magnitude spectrum.

Output File - <https://github.com/JagjitUvic/MIR/blob/master/Assignment%201/Ques5_3Output.wav>

Part 4: Overlap-add – using window Size – 32768

**import** numpy **as** np  
**import** matplotlib.pyplot **as** plt  
**import** scipy.fftpack  
**import** math  
**import** mir  
**from** mir **import** Sinusoid  
**from** mir **import** Signal  
  
sin = Signal()  
*#Read audio File*sin.wav\_read(**'Ques5Input.wav'**)  
wave = sin.data  
wavefft = np.fft.fft(wave)  
length = len(wavefft)  
**print** length  
window = length/32768  
list = []  
**print** window  
*#overlapping window*list.append(wavefft[0:32769])  
list.append(wavefft[32768:65537])  
list.append(wavefft[65536:131073])  
list.append(wavefft[131072:262145])  
*#inverse and save back*waveifft = np.fft.ifft(list)  
*#write back the signal object*sig = Signal(data=waveifft)  
sig.wav\_write(**'Ques5\_4Output.wav'**)

Overlapping and Non-Overlapping Approach



(<https://i.stack.imgur.com/Jg5EG.png)>

The overlapping approach is producing the better audio which is without the distortions at the window edges. In this technique, I have used WindowIndex+1, I have the window size = 32768 but I am always taking 32769 to have that overlap.

If you wish to reconstruct the wave after reducing spectral leakage make sure you use windowing but to avoid artifacts use Overlap-Add.