assignment-3-lssp

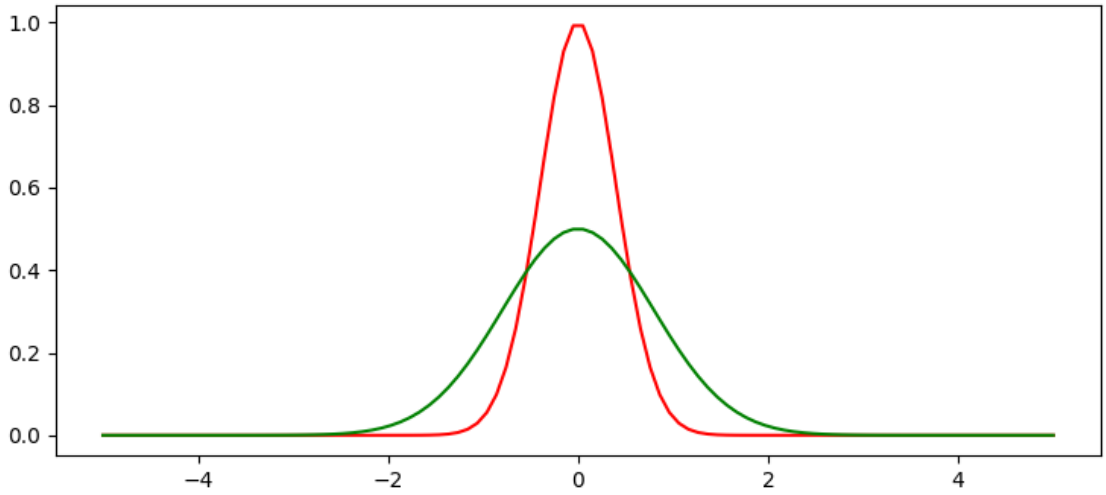
October 21, 2023

[12]:
```
# 2

import numpy as np
from matplotlib import pyplot as plt
import math
```

[13]:
```
plt.rcParams["figure.figsize"] = [7.50, 3.50]
plt.rcParams["figure.autolayout"] = True
```

[14]:
```
def f(x):
  a=1
  return np.exp(-1*(np.pi*(x**2))/a**2)/a

# Increasing the value of a
def g(x):
  a=2
  return np.exp(-1*(np.pi*(x**2))/a**2)/a
```

[15]:
```
x = np.linspace(-5, 5, 100)

plt.plot(x, f(x), color='red')
plt.plot(x, g(x), color='green')

plt.show()
```



As seen in the above graph, as we increase the value of a, the plot gets wider and flatter

$\mathcal{F}\{f(t)\} = \sum_{n=-\infty}^{\infty} f(nT)e^{-j2\pi fnT}/R_{\text{time}}$

Above represents the infinite summation approximation for the fourier transform

$\mathcal{F}\{f(t)\} = \sum_{n=-L/2R_{\text{time}}}^{L/2R_{\text{time}}} f(nT)e^{-j2\pi fnT}/R_{\text{time}}$

Above is the modified expression after applying the window

$\mathcal{F}\{f(t)\} = \sum_{n=-L/2R_{\text{time}}}^{L/2R_{\text{time}}} f(nT)e^{-j2\pi m(Rfreq)nT}/R_{\text{time}}$

Above is the approximate expression for the fourier transform at m(Rfreq)

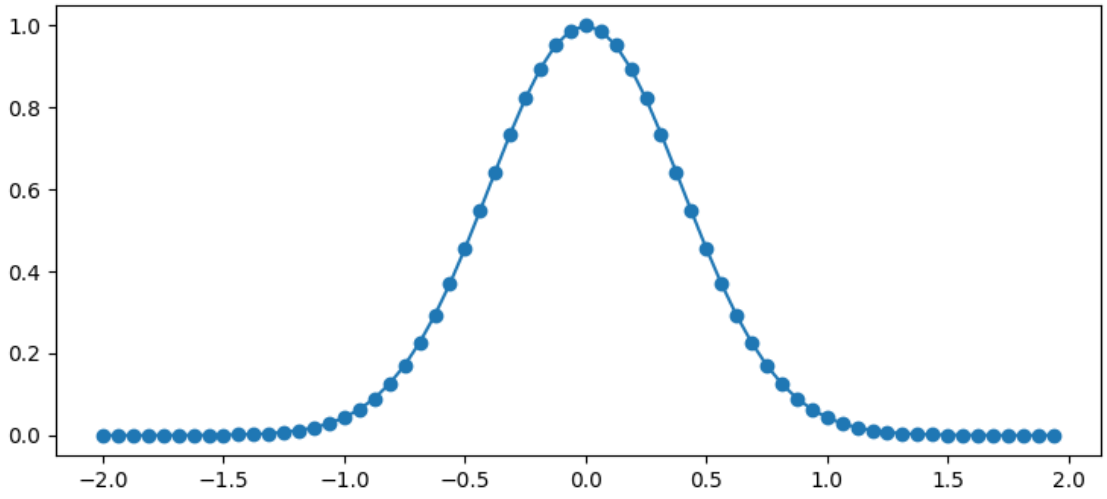
Number of samples in time domain = L * (sampling rate in time domain) = L*B

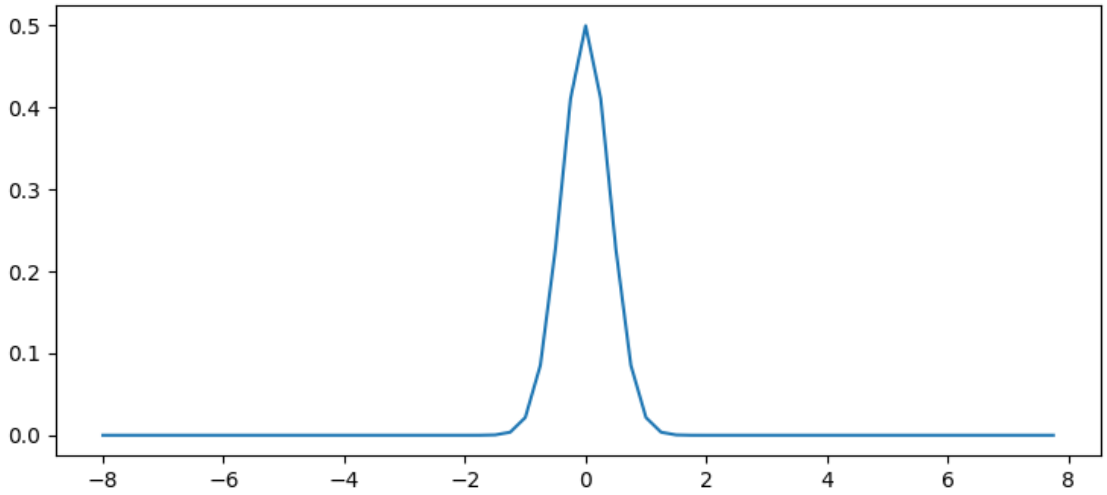
Number of samples in frequency domain = B * (sampling rate in frequency domain) = L*B

We pick the parameters in this way for it to obey the nyquist theorem. When the signal is approximately time and band limited, the sampling rates of one domain are related to the widths of the other domain reciprocally

[16]:
```
N = 64
B = 16
L = N/B
Rf = 1/L
Rt = 1/B

x_sampled_time = np.arange(-L/2, L/2, Rt)
y_sampled_time = f(x_sampled_time)

plt.plot(x_sampled_time, y_sampled_time, marker = 'o')
plt.show()
```

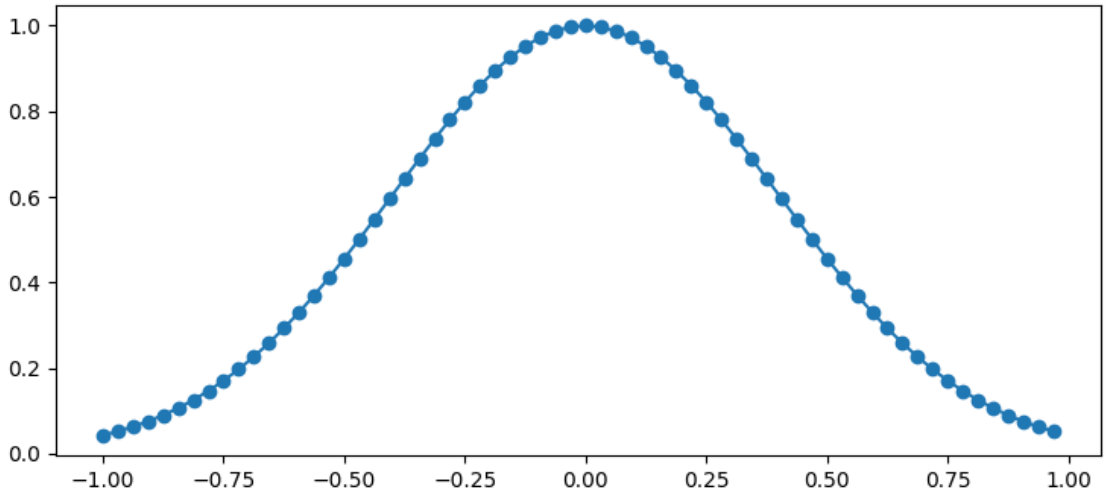
```
x_fft = np.arange(-B/2, B/2, Rf)
y_fft = 2*np.fft.fftshift(np.abs(np.fft.fft(y_sampled_time))) /␣
 ↪len(y_sampled_time)

plt.plot(x_fft, y_fft)
plt.show()
```

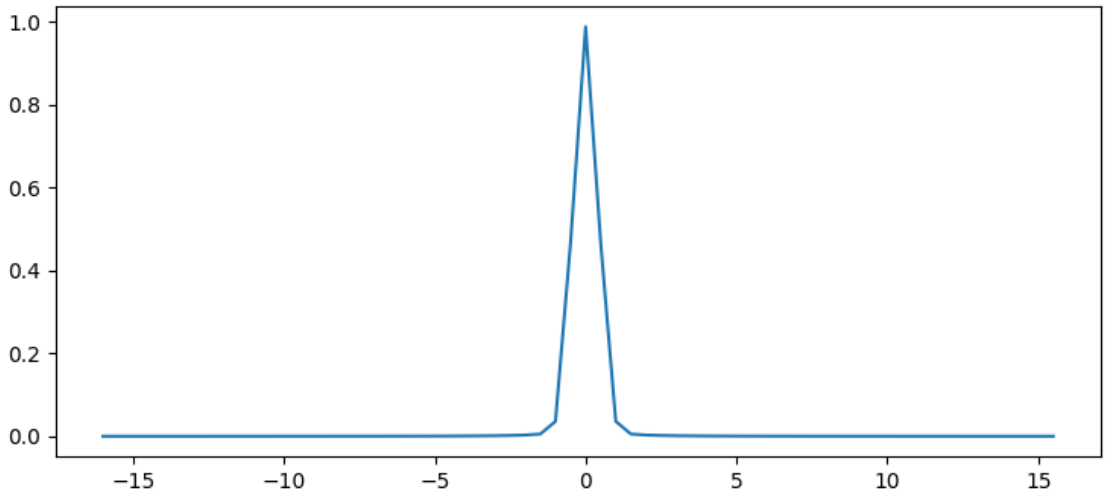




[17]:
```
N = 64
B = 32
L = N/B
Rf = 1/L
Rt = 1/B
```

```
x_sampled_time = np.arange(-L/2, L/2, Rt)
y_sampled_time = f(x_sampled_time)

plt.plot(x_sampled_time, y_sampled_time, marker = 'o')
plt.show()

x_fft = np.arange(-B/2, B/2, Rf)
y_fft = 2*np.fft.fftshift(np.abs(np.fft.fft(y_sampled_time))) /␣
  ↪len(y_sampled_time)

plt.plot(x_fft, y_fft)
plt.show()
```

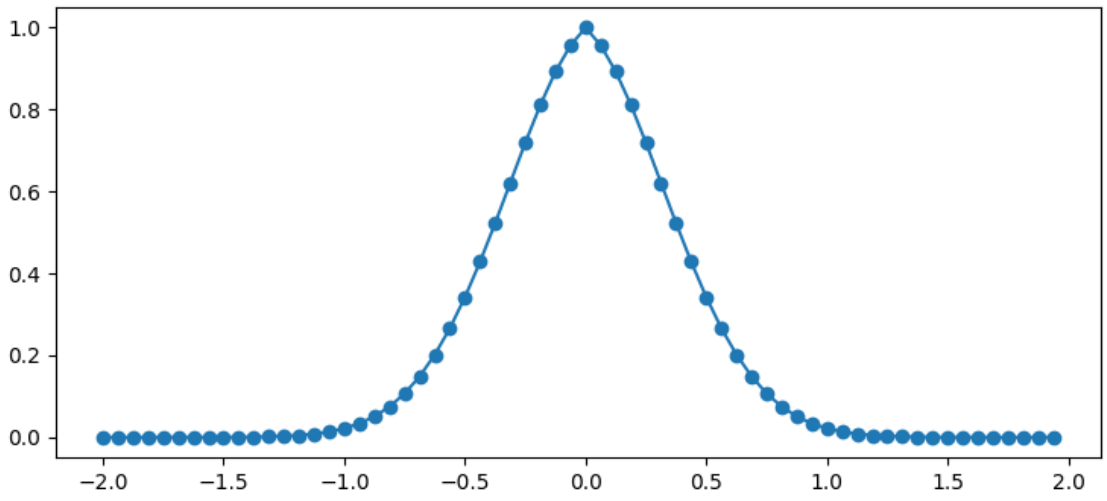


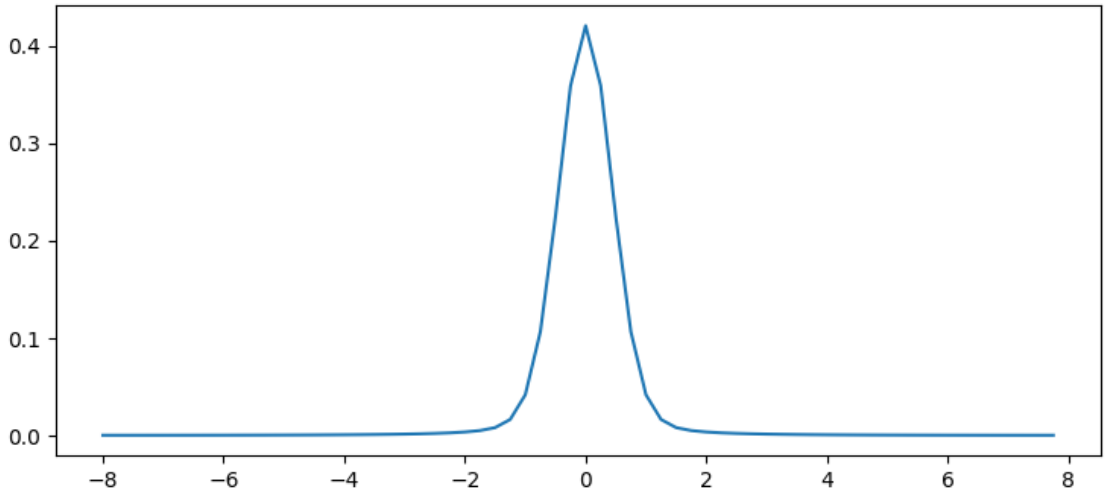


By plotting the above graphs, we observe that, as we increase the bandwidth, the time duration decreases and vice versa

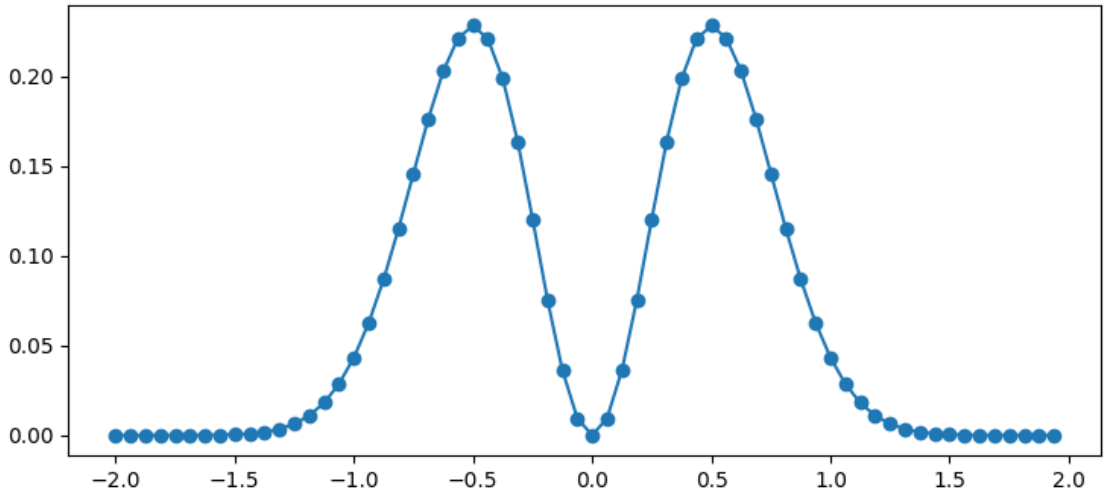
(Since all the parameters are inter-related, we only require values for two of the parameters to uniquely determine the other parameters. Hence we only need to check for varying one, since the rest of the variation of parameters will be present in this variation)

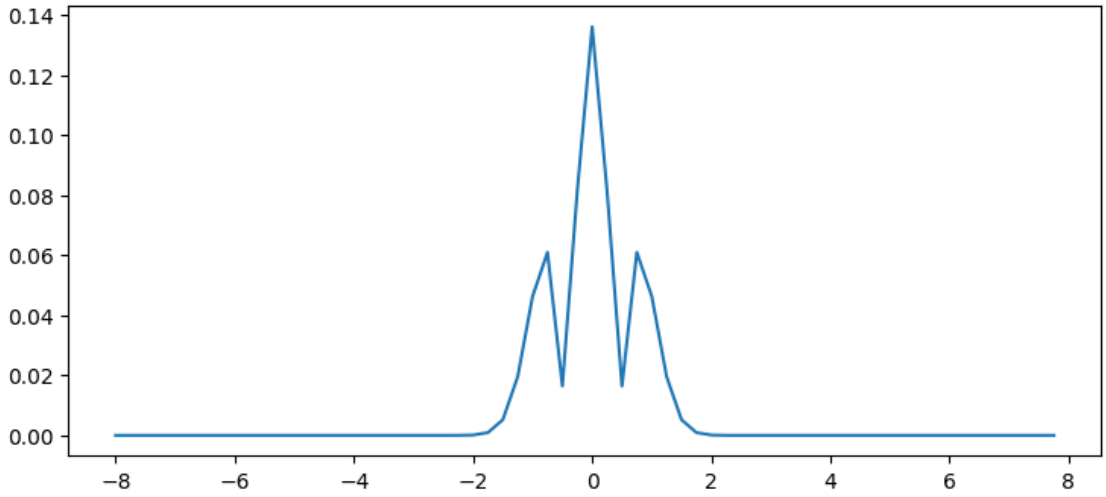
[18]:
```
N = 64
L = 4
B = N/L
Rf = 1/L
Rt = 1/B

x_sampled_time = np.arange(-L/2, L/2, Rt)
windowed_1 = f(x_sampled_time)*(1 - 2 * np.abs(x_sampled_time) / L)

plt.plot(x_sampled_time, windowed_1, marker = 'o')
plt.show()

x_fft = np.arange(-B/2, B/2, Rf)
y_fft = 2*np.fft.fftshift(np.abs(np.fft.fft(windowed_1))) / len(windowed_1)

plt.plot(x_fft, y_fft)
plt.show()
```





[19]:
```
N = 64
L = 4
B = N/L
Rf = 1/L
Rt = 1/B

x_sampled_time = np.arange(-L/2, L/2, Rt)
windowed_1 = f(x_sampled_time)*(np.sin(2*np.pi*x_sampled_time/L)**2)

plt.plot(x_sampled_time, windowed_1, marker = 'o')
plt.show()

x_fft = np.arange(-B/2, B/2, Rf)
y_fft = 2*np.fft.fftshift(np.abs(np.fft.fft(windowed_1))) / len(windowed_1)

plt.plot(x_fft, y_fft)
plt.show()
```





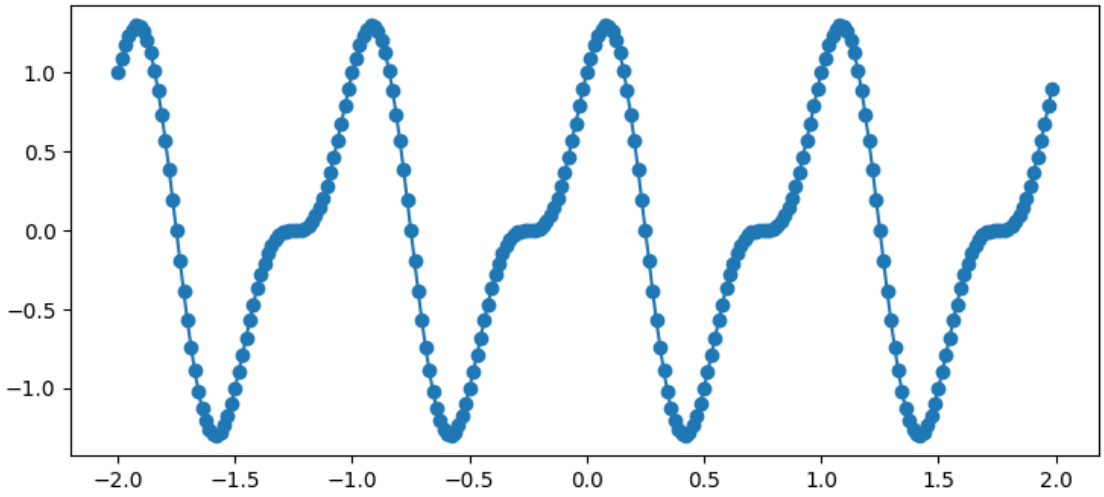
[20]:
```
import numpy as np
from matplotlib import pyplot as plt
import math
```

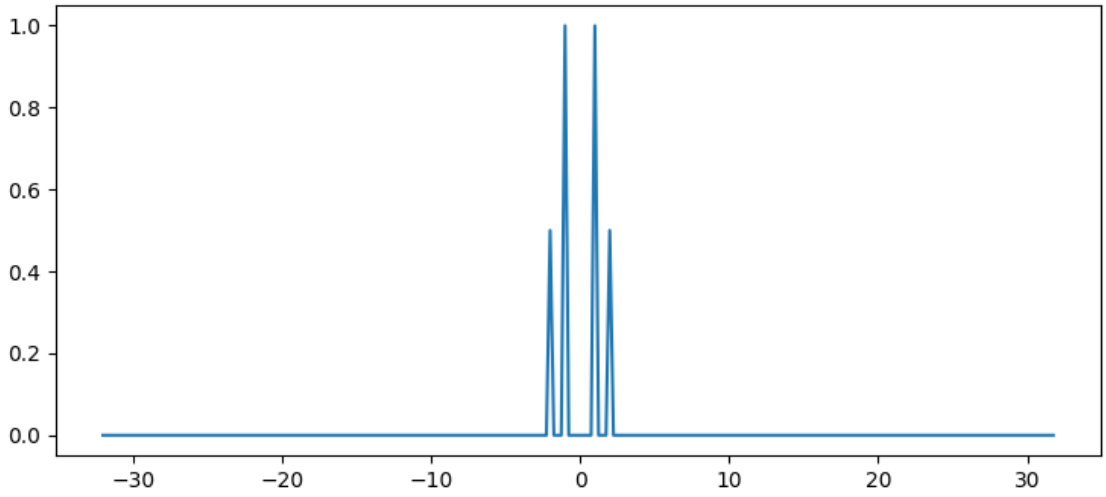
[21]:
```
plt.rcParams["figure.figsize"] = [7.50, 3.50]
plt.rcParams["figure.autolayout"] = True
```

[22]:
```
def f(x):
  return np.cos(2*np.pi*x) + np.sin(4*np.pi*x)/2
```

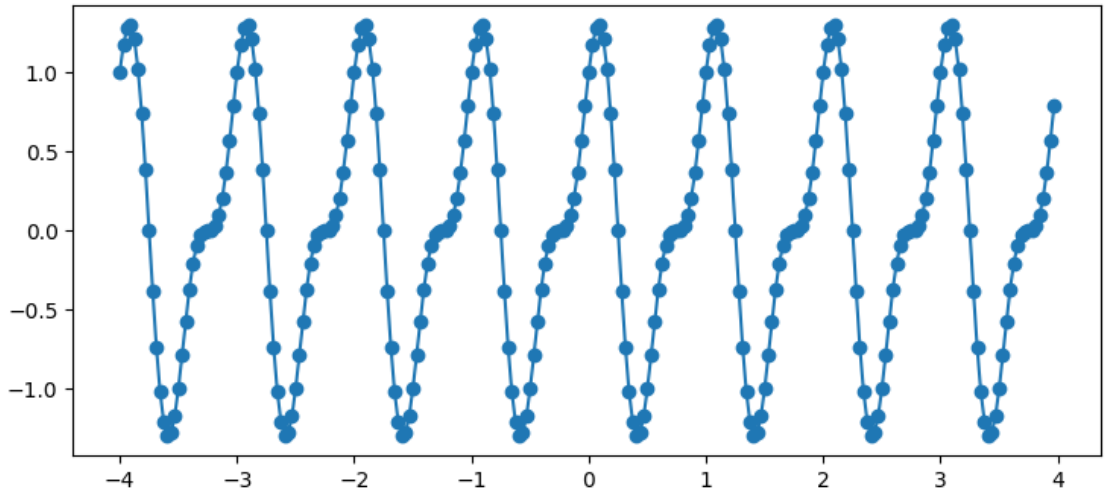
[23]:
```
N = 256
B = 64
```

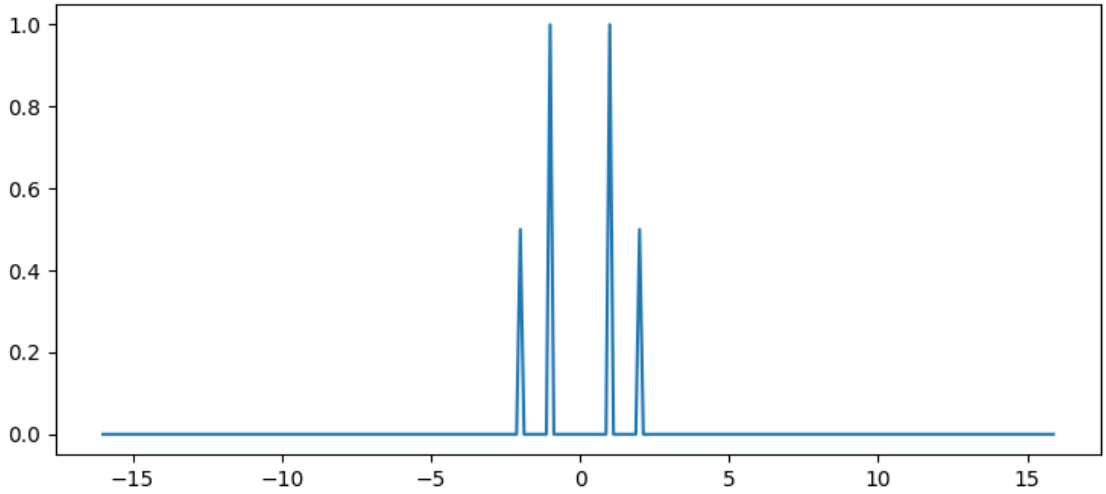
```
L = N/B
Rf = 1/L
Rt = 1/B

x_sampled_time = np.arange(-L/2, L/2, Rt)
y_sampled_time = f(x_sampled_time)

plt.plot(x_sampled_time, y_sampled_time, marker = 'o')
plt.show()

x_fft = np.arange(-B/2, B/2, Rf)
y_fft = 2*np.fft.fftshift(np.abs(np.fft.fft(y_sampled_time))) /␣
 ↪len(y_sampled_time)

plt.plot(x_fft, y_fft)
plt.show()
```





[24]:
```
N = 256
B = 32
L = N/B
Rf = 1/L
Rt = 1/B

x_sampled_time = np.arange(-L/2, L/2, Rt)
y_sampled_time = f(x_sampled_time)

plt.plot(x_sampled_time, y_sampled_time, marker = 'o')
plt.show()

x_fft = np.arange(-B/2, B/2, Rf)
y_fft = 2*np.fft.fftshift(np.abs(np.fft.fft(y_sampled_time))) /␣
 ↪len(y_sampled_time)

plt.plot(x_fft, y_fft)
plt.show()
```

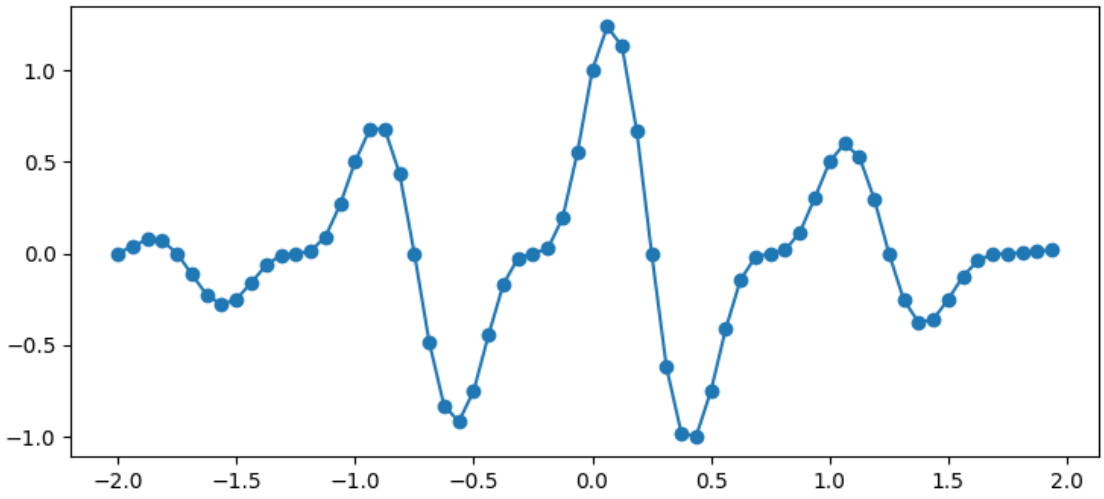


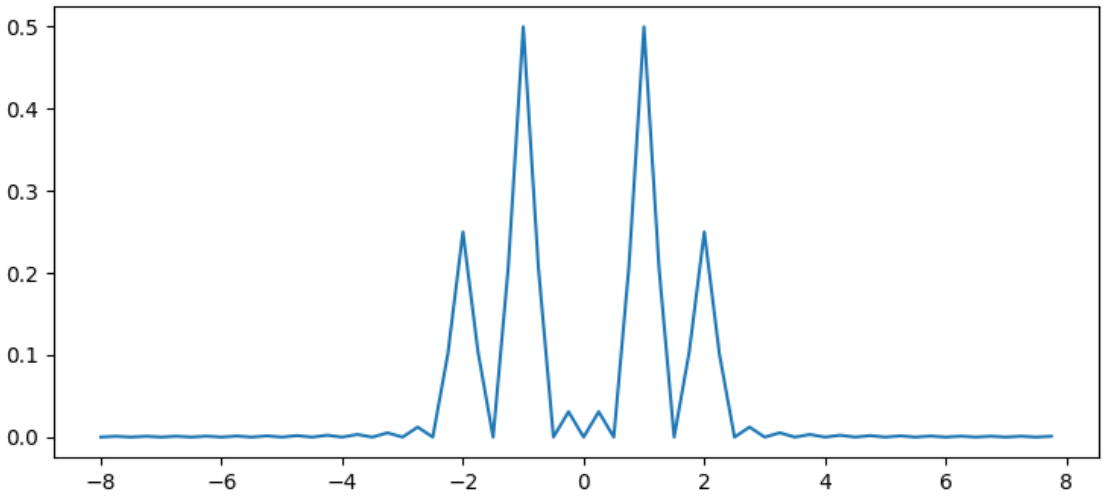


By plotting the above graphs, we observe that, as we increase the bandwidth, the time duration decreases and vice versa

[25]:
```
N = 64
L = 4
B = N/L
Rf = 1/L
Rt = 1/B

x_sampled_time = np.arange(-L/2, L/2, Rt)
windowed_1 = f(x_sampled_time)*(1 - 2 * np.abs(x_sampled_time) / L)

plt.plot(x_sampled_time, windowed_1, marker = 'o')
```

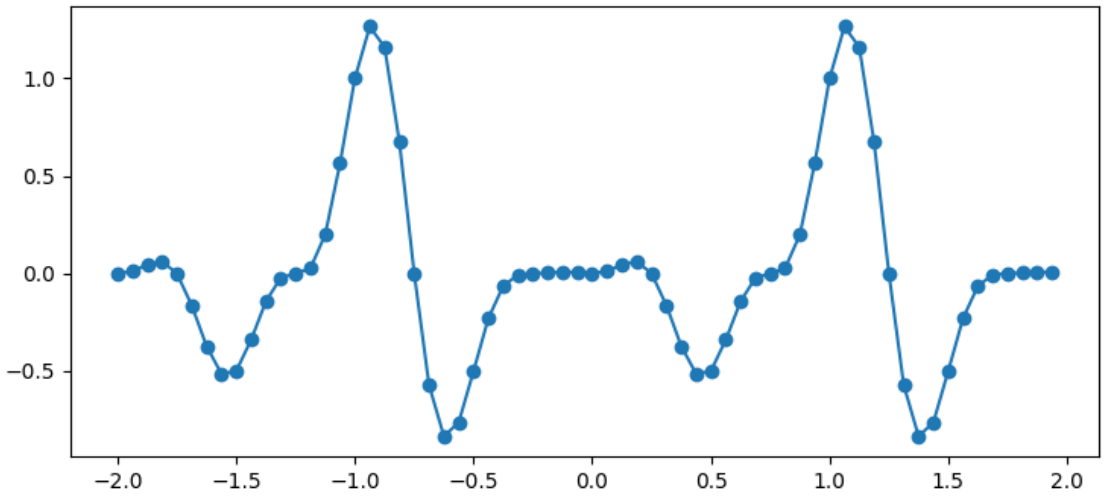
```
plt.show()

x_fft = np.arange(-B/2, B/2, Rf)
y_fft = 2*np.fft.fftshift(np.abs(np.fft.fft(windowed_1))) / len(windowed_1)

plt.plot(x_fft, y_fft)
plt.show()
```

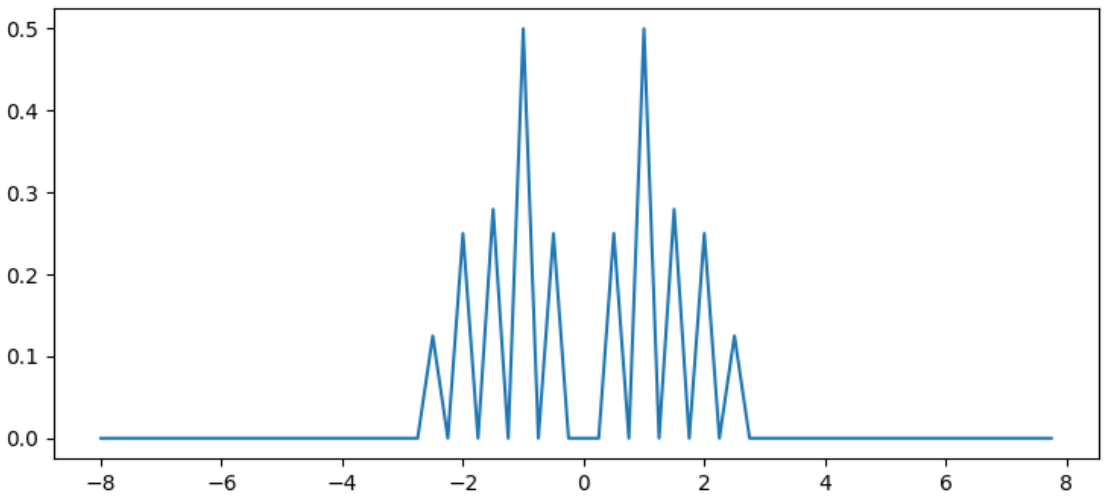




[26]:
```
N = 64
L = 4
B = N/L
Rf = 1/L
```

```
Rt = 1/B

x_sampled_time = np.arange(-L/2, L/2, Rt)
windowed_1 = f(x_sampled_time)*(np.sin(2*np.pi*x_sampled_time/L)**2)

plt.plot(x_sampled_time, windowed_1, marker = 'o')
plt.show()

x_fft = np.arange(-B/2, B/2, Rf)
y_fft = 2*np.fft.fftshift(np.abs(np.fft.fft(windowed_1))) / len(windowed_1)

plt.plot(x_fft, y_fft)
plt.show()
```





3

The camera samples images at a rate such that the rotor blades are at the same position at the instants of capture.

Between the times of each of the captures, the blades must have covered an angle of an integral multiple of $2\pi/5$. Therefore,

R1*2nπ/5 = 2πR2

R2 = n*R1/5

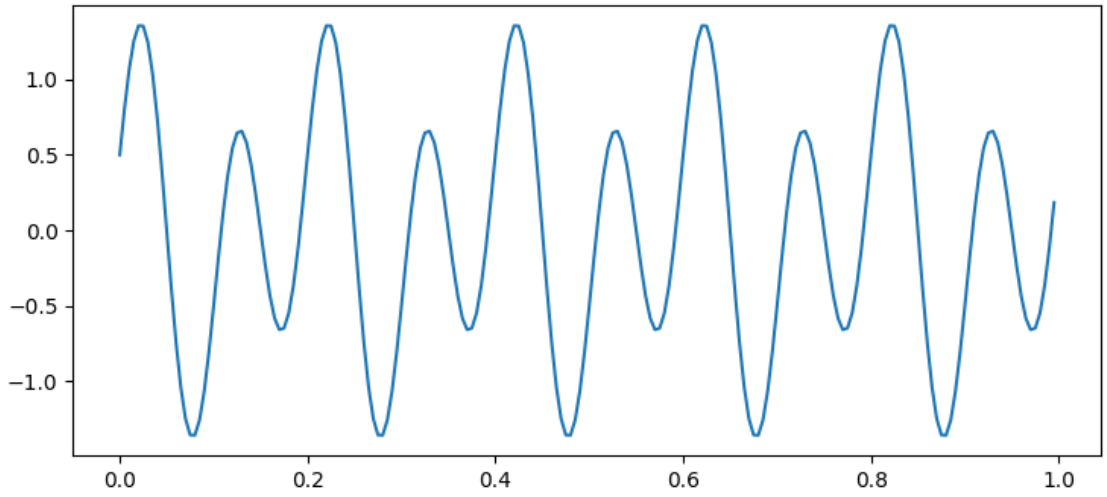
In case we see 6 blades, there could have been 3 or 2 blades. This is due to the aliasing of our eyesight: taking the average of the images perceived at different but close time instances. This is not possible with 5 blades, since it has no factors besides 1. (assuming 1 blade isn't valid)

[27]:
```
# 4

import numpy as np
from matplotlib import pyplot as plt
import math
```

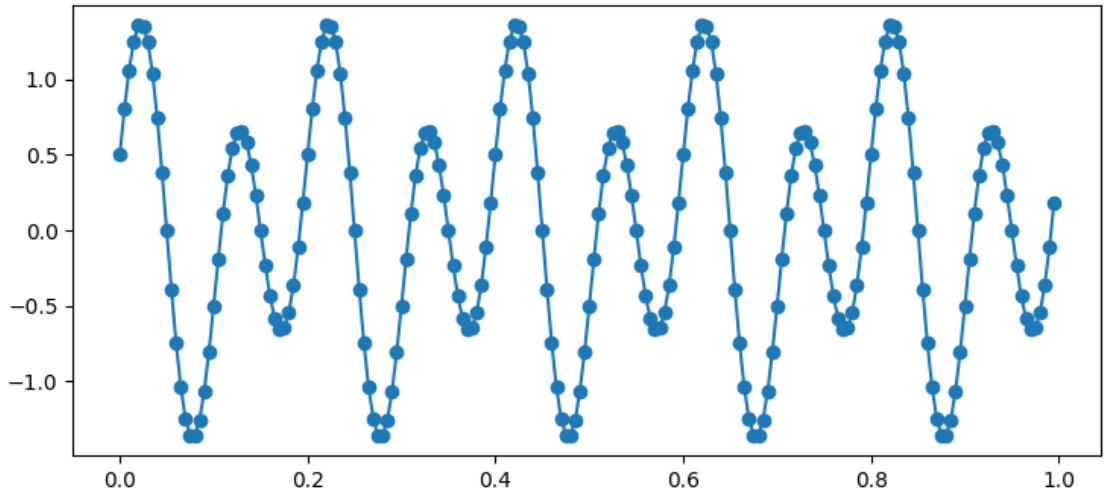
[28]:
```
plt.rcParams["figure.figsize"] = [7.50, 3.50]
plt.rcParams["figure.autolayout"] = True
```

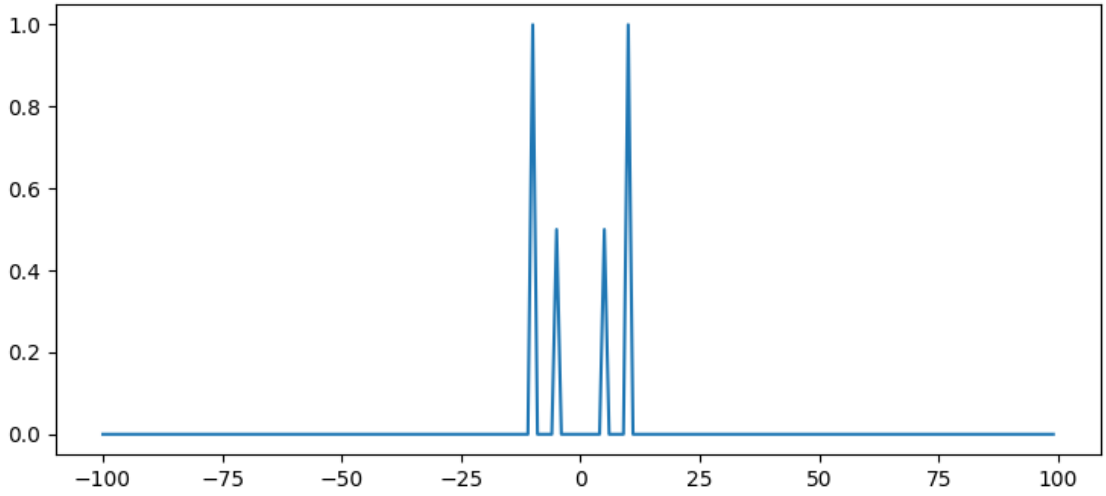
[29]:
```
# Sample function with the given frequencies

def f(x):
  return np.cos(10*np.pi*x)/2 + np.sin(20*np.pi*x)
```

[30]:
```
# Plotting the above function with a rate of Rs

Rs = 200
x = np.arange(0, 1, 1/Rs)
plt.plot(x, f(x))
plt.show()
```



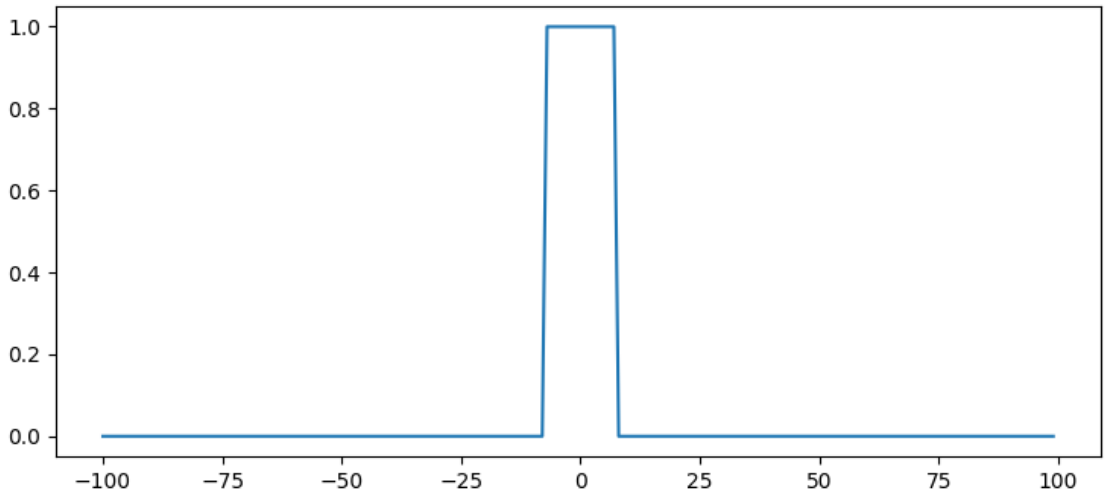
[31]:
```
N = Rs
L = 1
B = N/L
Rf = 1/L
Rt = 1/B

x_sampled_time = np.arange(0, 1, Rt)
y_sampled_time = f(x_sampled_time)

plt.plot(x_sampled_time, y_sampled_time, marker = 'o')
plt.show()

x_fft = np.arange(-B/2, B/2, Rf)
y_fft = 2*np.fft.fftshift(np.abs(np.fft.fft(y_sampled_time))) /␣
 ↪len(y_sampled_time)

plt.plot(x_fft, y_fft)
plt.show()
```

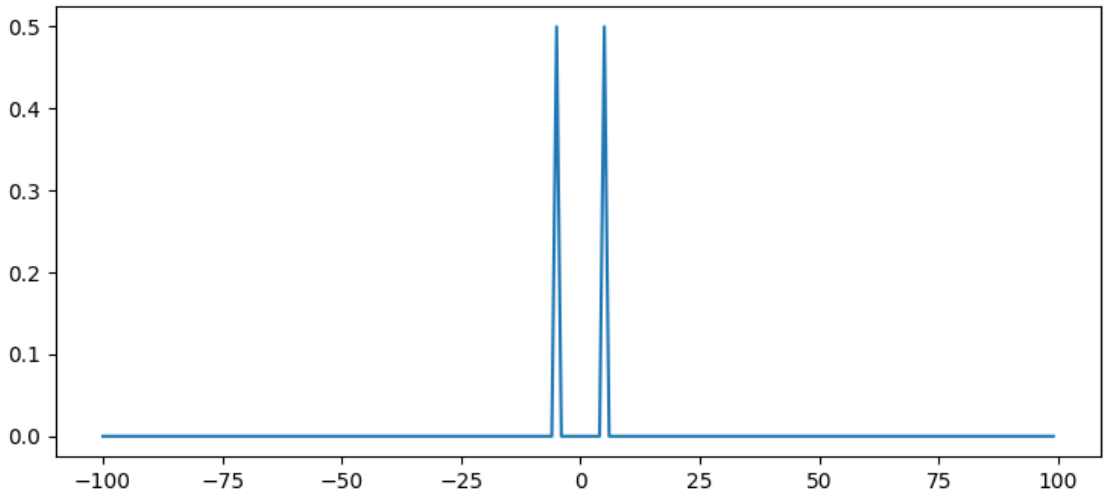




[32]:
```
def filter(x):
  cutoff = 7.5
  return (np.sign(cutoff-np.abs(x))+1)/2

x_samples = np.arange(-B/2, B/2, Rf)
y_filter = filter(x_samples)

plt.plot(x_samples, y_filter)
plt.show()

y_filtered = y_filter * y_fft
```

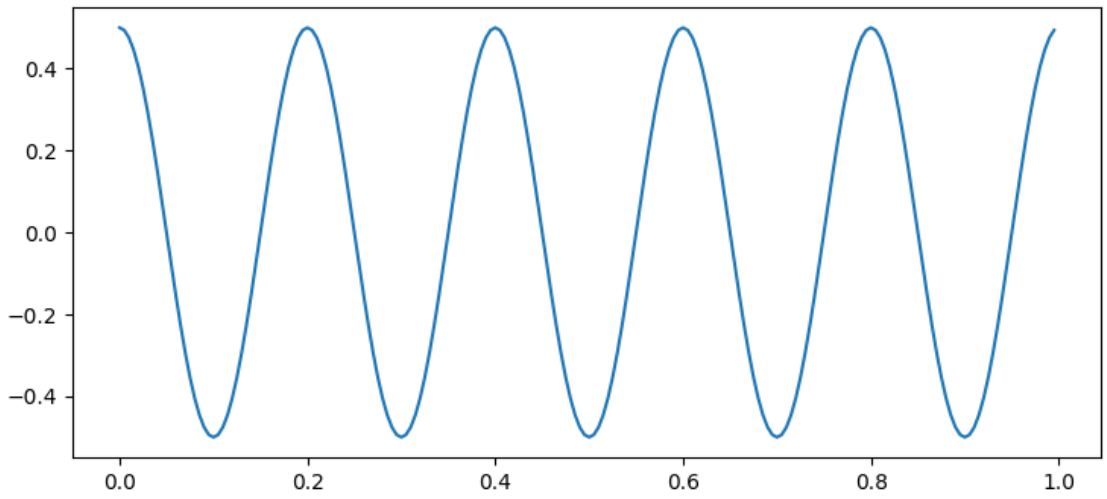
```
plt.plot(x_samples, y_filtered)
plt.show()
```





[33]:
```
y_ifft = np.fft.ifft(np.fft.fft(y_sampled_time) * np.fft.
  ↪ifftshift(filter(x_samples)))
plt.plot(x_sampled_time, y_ifft)
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/matplotlib/cbook/__init__.py:1335:
ComplexWarning: Casting complex values to real discards the imaginary part
  return np.asarray(x, float)
```



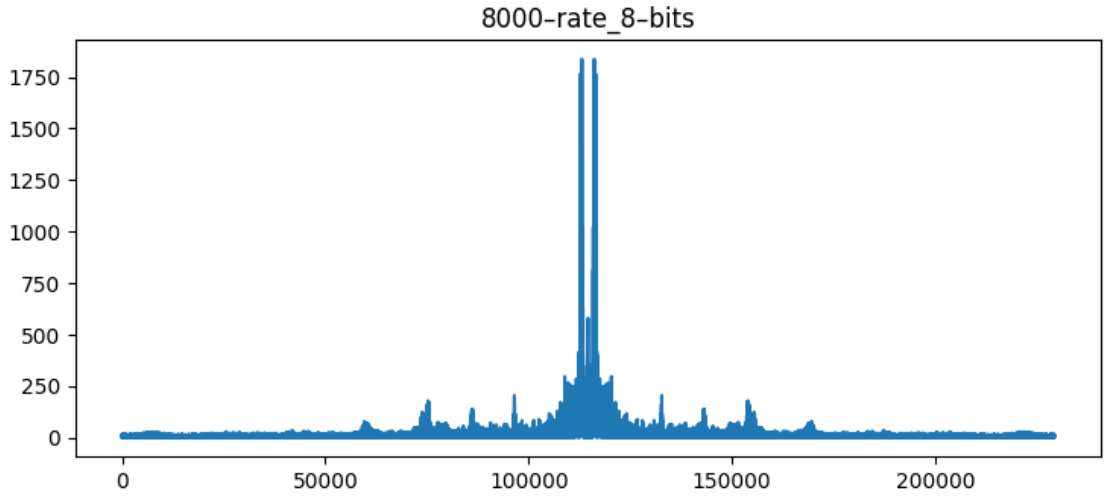
[34]:
```
# 5

import numpy as np
from matplotlib import pyplot as plt
import math
import scipy
```

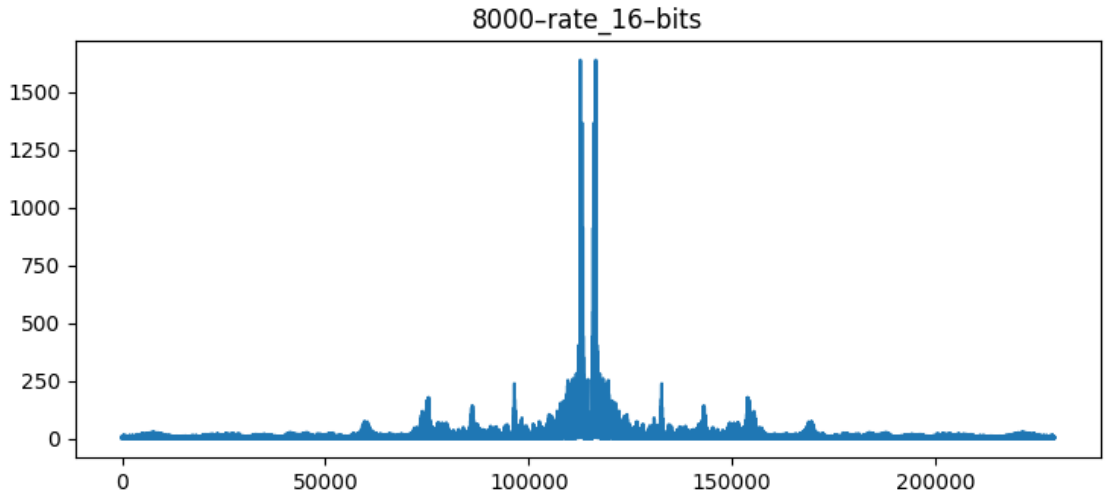
[35]:
```
original_rate, data_two_channel = scipy.io.wavfile.read('audio_lssp_islandy.
 ↪wav')
R_t = original_rate
```

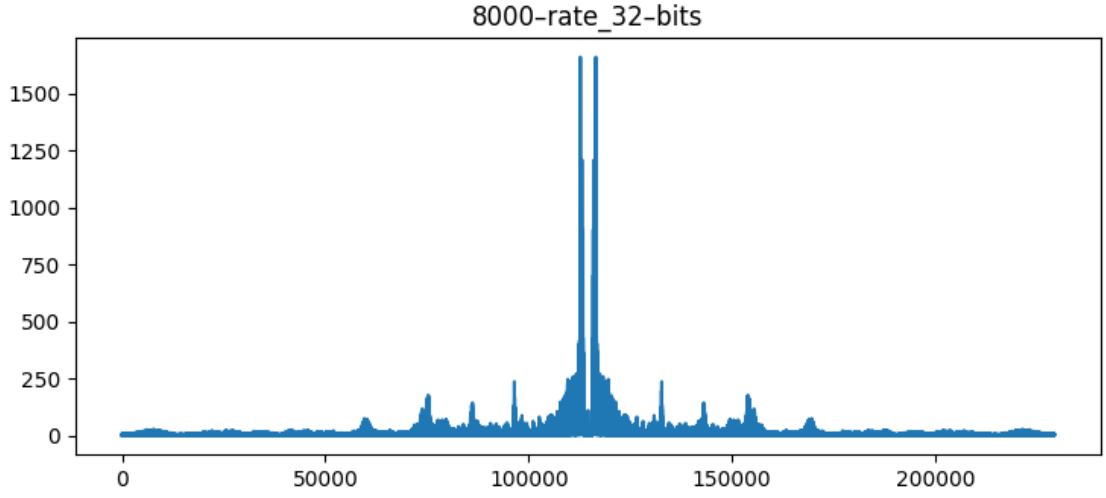
[36]:
```
original_data = data_two_channel[:, 0]
 scipy.io.wavfile.write("input.wav", original_rate, original_data)
```

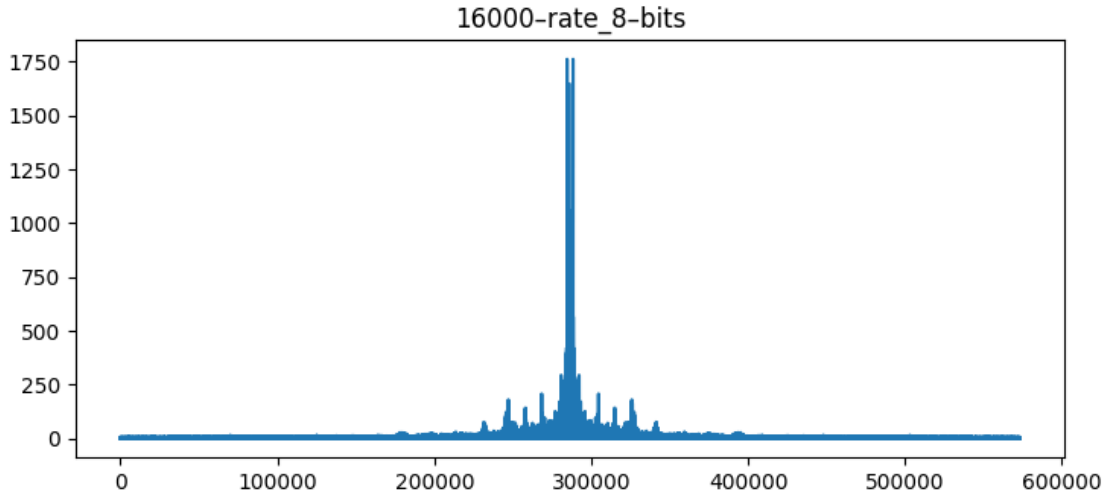
[37]:
```
def change_sampling_rate(original_data, original_rate, target_rate):
  # Considering only downsampling

  downsampling_ratio = original_rate//target_rate
  downsampled_signal = original_data[::downsampling_ratio]
  return downsampled_signal, target_rate

def quantize(data, bits):
  data = data.astype(np.int32)
  max_val = np.max(data)
  min_val = np.min(data)
  for i in range(len(data)):
    quantized_val = min_val + (max_val - min_val)/(bits - 1)*round((data[i] -␣
 ↪min_val) * (bits - 1)/(max_val - min_val))
    data[i] = quantized_val
```

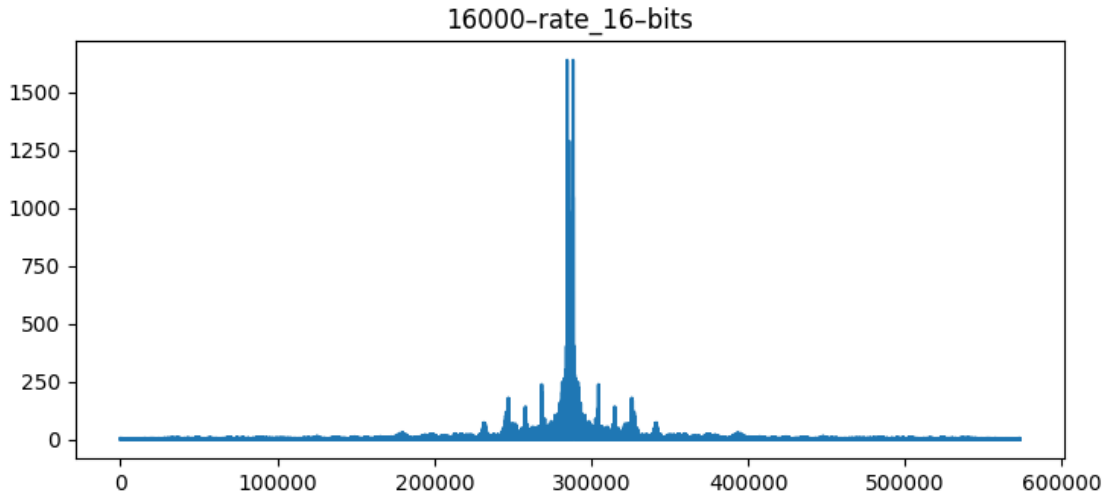
```
  data = data.astype(np.int16)
  return data


sampling_rates = [8000, 16000, 22050, 44100] # Different sampling rates to␣
 ↪test, for example
quantization_bits = [8, 16, 32] # Different quantization levels to test
for rate in sampling_rates:
  for bits in quantization_bits:
    # Change sampling rate
    new_data, new_rate = change_sampling_rate(original_data, original_rate,␣
 ↪rate)

    # Quantize the audio data
    quantized_data = quantize(new_data, bits)

    # Plotting the fourier transform
    plt.title(f"{rate}-rate_{bits}-bits")
    plt.plot(2*np.fft.fftshift(np.abs(np.fft.fft(quantized_data))) /␣
 ↪len(quantized_data))
    plt.show()

    # Save the quantized audio
    output_filename = f"output_{rate}-rate_{bits}-bits.wav"
    scipy.io.wavfile.write(output_filename, new_rate, quantized_data)
```

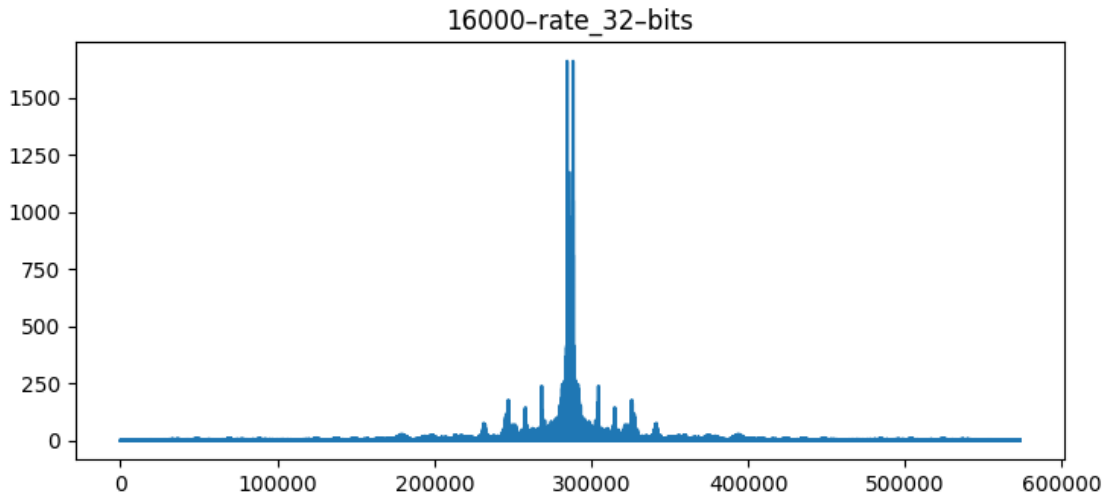


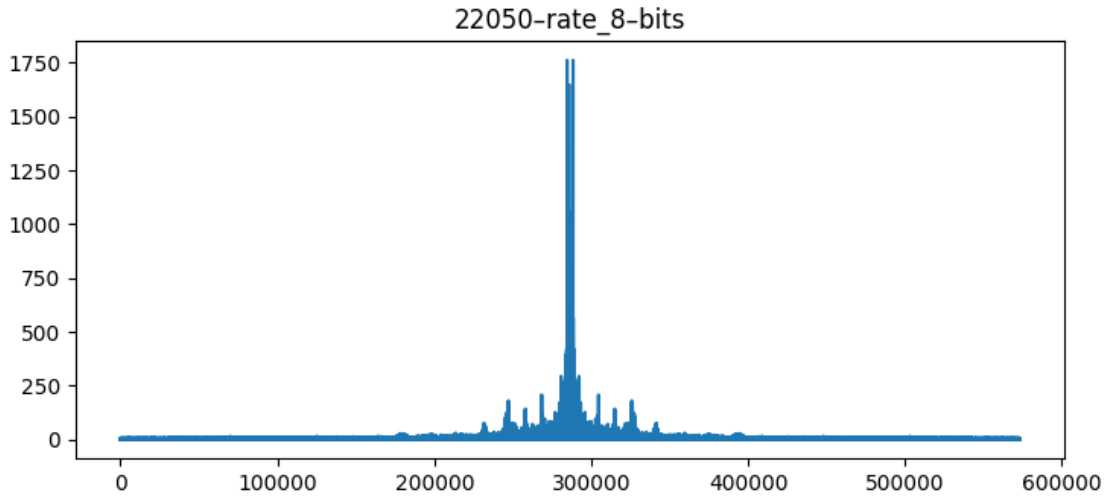

8000-rate_16-bits
1500
1250
1000
750
500
250
0
0
50000
100000
150000
200000

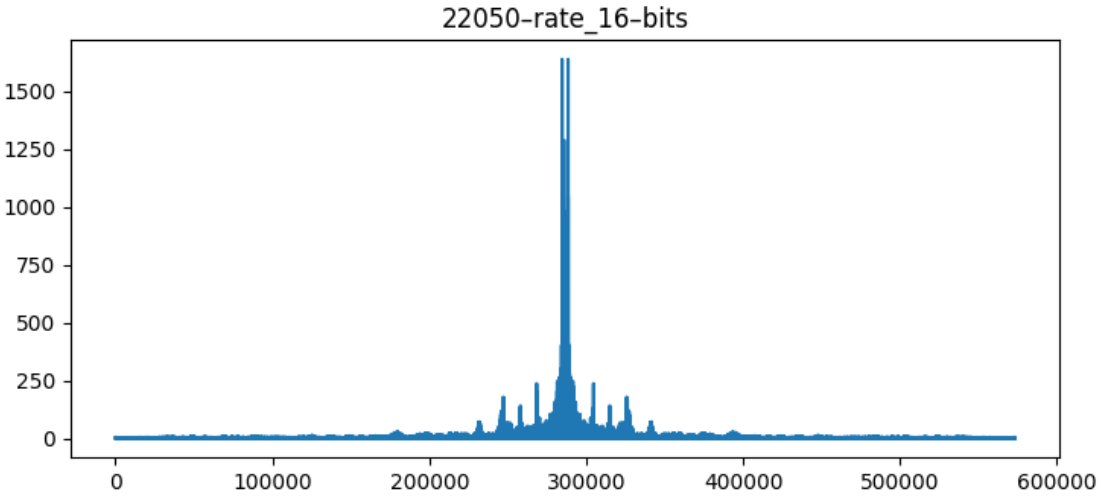

8000-rate_32-bits
1500
1250
1000
750
500
250
0
0
50000
100000
150000
200000

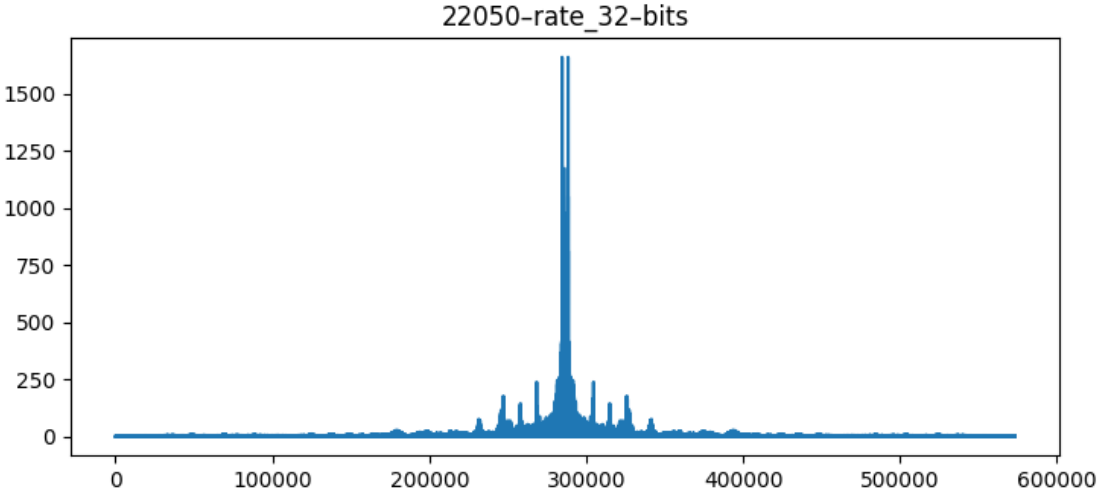

16000–rate_8–bits
1750
1500
1250
1000
750
500
250
0
0
100000
200000
300000
400000
500000
600000

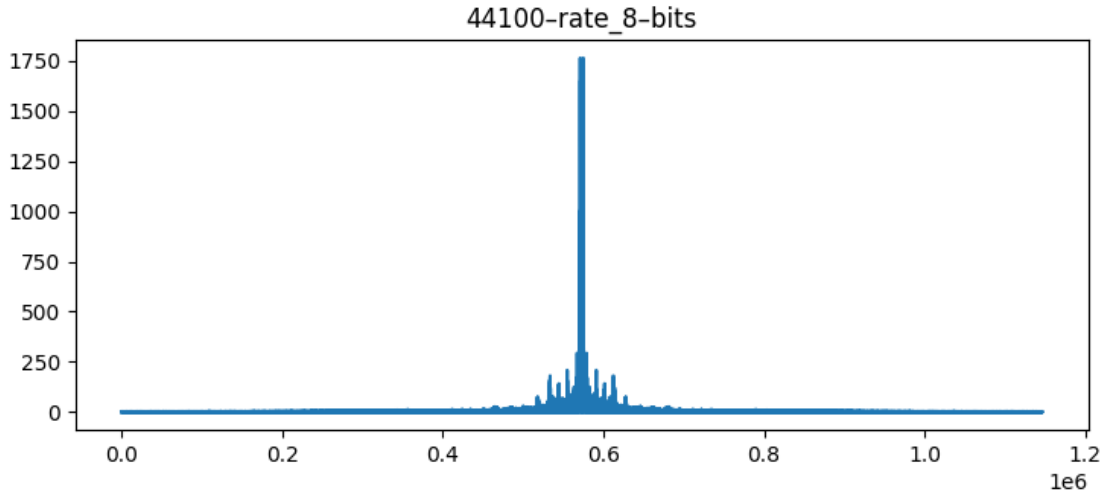

16000–rate_16–bits
1500
1250
1000
750
500
250
0
0
100000
200000
300000
400000
500000
600000

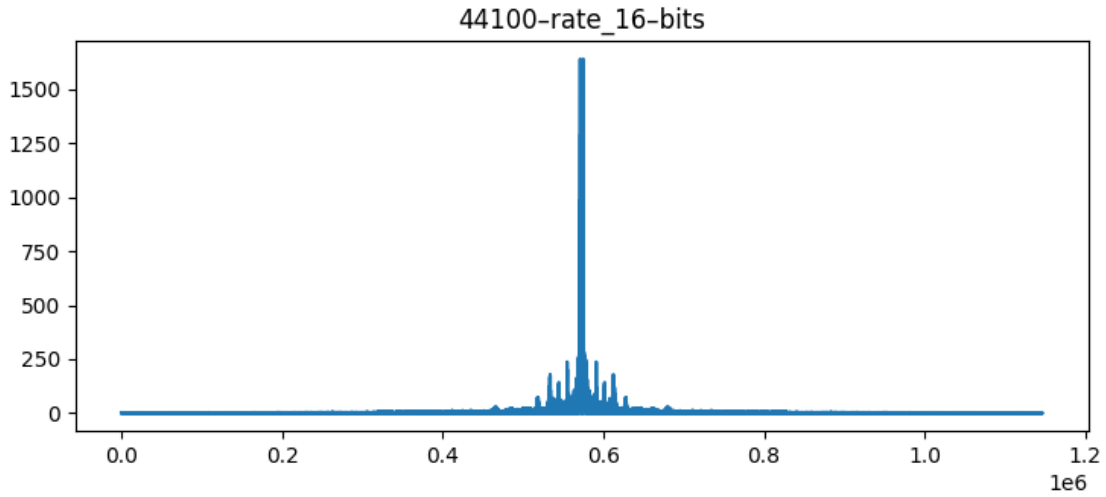

16000–rate_32–bits
1500
1250
1000
750
500
250
0
0
100000
200000
300000
400000
500000
600000

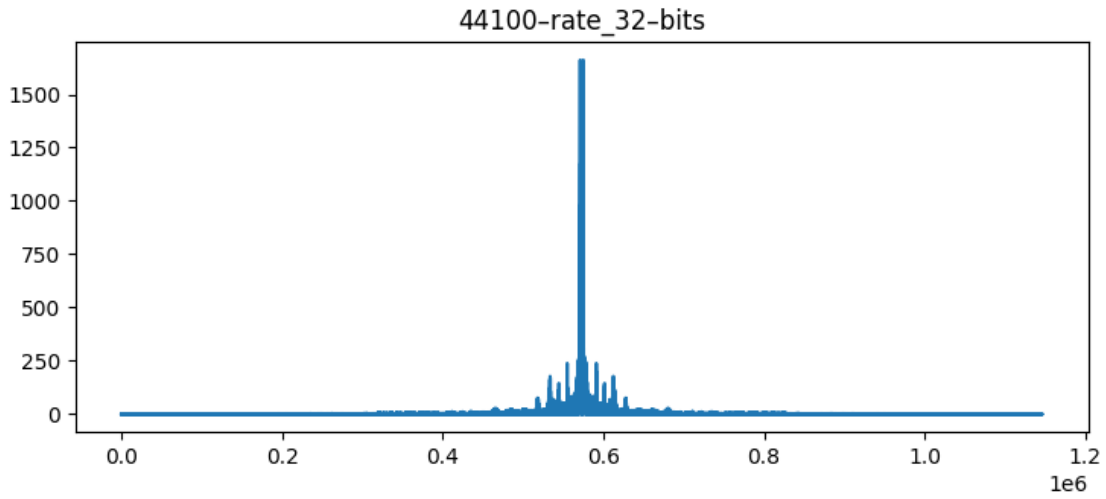

22050–rate_8–bits
1750
1500
1250
1000
750
500
250
0
0
100000
200000
300000
400000
500000
600000


22050–rate_16–bits
1500
1250
1000
750
500
250
0
0
100000
200000
300000
400000
500000
600000


22050–rate_32–bits
1500
1250
1000
750
500
250
0
0
100000
200000
300000
400000
500000
600000


44100–rate_8–bits
1750
1500
1250
1000
750
500
250
0
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1e6


44100–rate_16–bits
1500
1250
1000
750
500
250
0
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1e6



Analysis and Interpretation:

- On decreasing the sampling rate, there is a loss of the higher frequencies, and due to no filtering during downsampling, there is aliasing. This is due to the Nyquist theorem
- On quantizing, there is a reduction in audio quality. This is due to not being able to take dynamic values for the amplitude, instead discrete values are taken