Signal Processing

Syed Saqib Habeeb (BM20BTECH11015)

Q1) Plotting sine waves and chirps

```
import math
import matplotlib.pyplot as plt
import numpy as np
from scipy.fftpack import fftfreq
from scipy.fftpack import fft, ifft
from scipy.signal import chirp,spectrogram
import scipy as sp
import sounddevice as sd
```

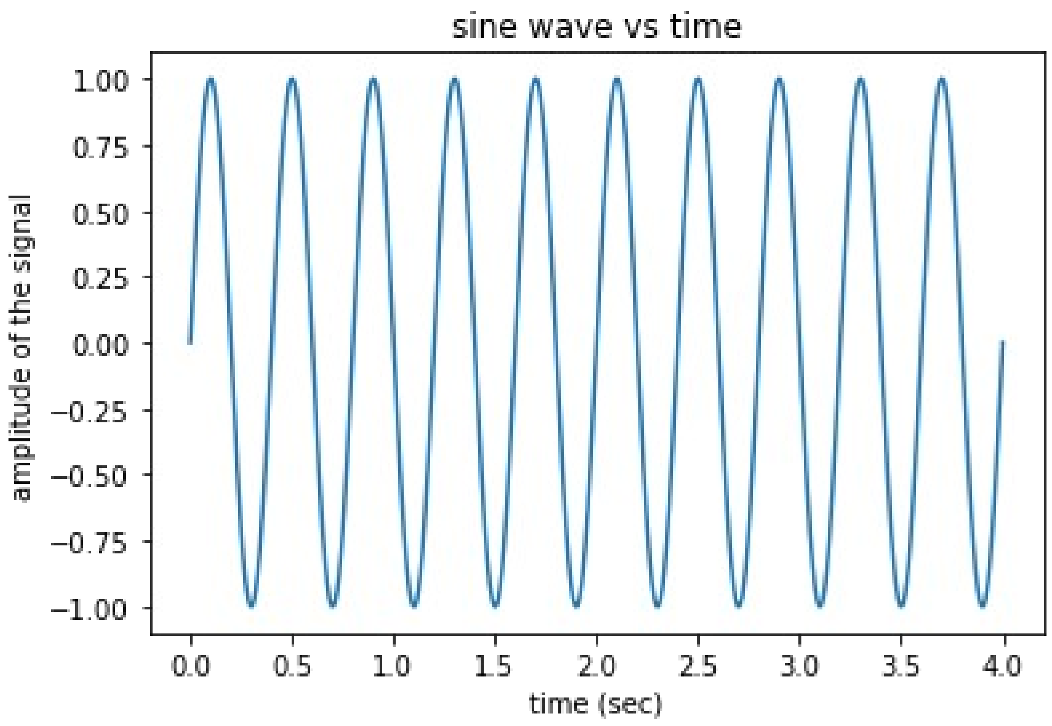
```
pi=math.pi
sin=[]
linearChirp=[]
quadChirp=[]
logChirp=[]
concaveChirp=[]
```

Sin wave

```
#the sin function is sin(5*pi*t)
t=np.linspace(0,4,5000)
w=5*pi
for i in range(len(t)):
    sin.append(np.sin(w*t[i]))

plt.title('sine wave vs time')
plt.xlabel('time (sec)')
plt.ylabel('amplitude of the signal')
plt.plot(t,sin)
```

```
sd.play(sin,50000)
```

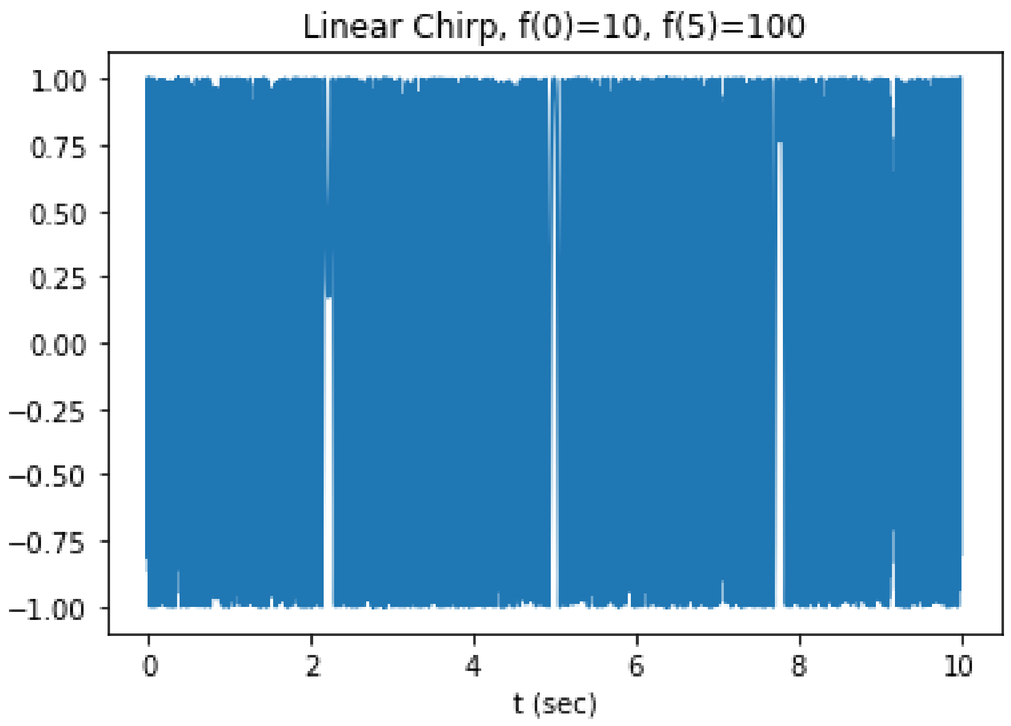
```
ALSA lib pcm.c:8526:(snd_pcm_recover) underrun occurred
ALSA lib pcm.c:8526:(snd_pcm_recover) underrun occurred
```



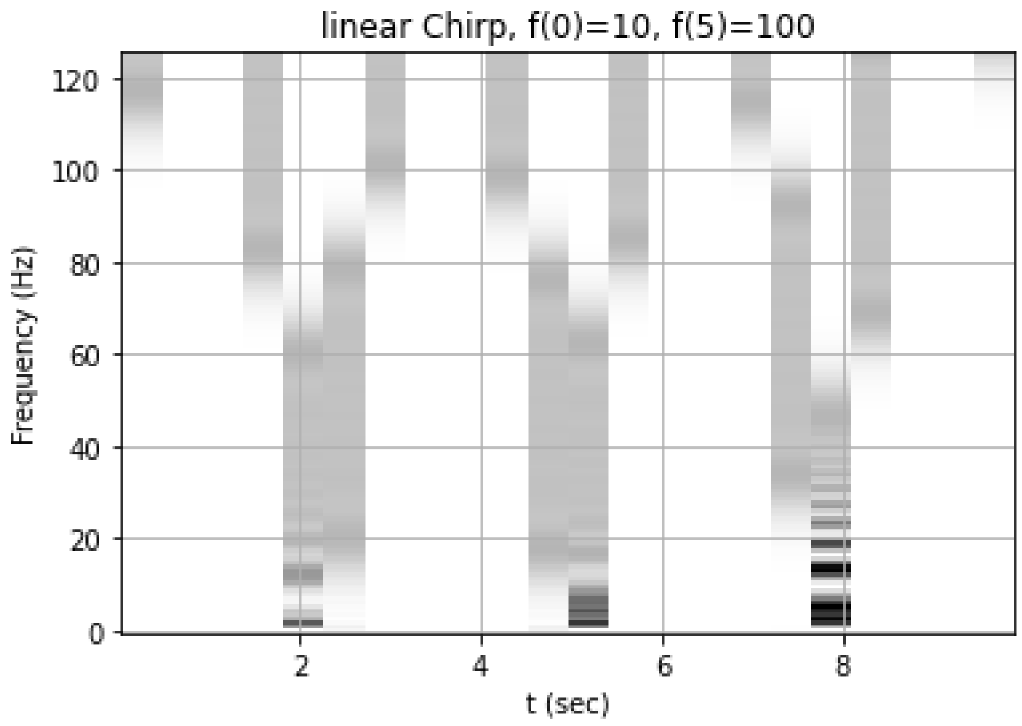
Here in the graph above we can see that for a sinus signal sin(5πt),the period of the signal is 2/5. On the X-axis we have the time from t=0 to t=4 and on the y axis we have the amplitude of the signal which ranges from -1 to 1. We know that the sampling frequency should be atleast twice the frequency of the signal, hence the sampling frequency is 10.

linear chirping

```
t = np.linspace(0, 10, 5000)
w = chirp(t, f0=100, f1=1000, t1=5, method='linear')
plt.plot(t, w)
plt.title("Linear Chirp, f(0)=10, f(5)=100")
plt.xlabel('t (sec)')
plt.show()
```



```
fs = 500
T = 5
t = np.arange(0, int(T*fs)) / fs

def plot_spectrogram(title, w, fs):
    ff, tt, Sxx = spectrogram(w, fs=fs, nperseg=256, nfft=576)
    fig, ax = plt.subplots()
    ax.pcolormesh(tt, ff[:145], Sxx[:145],cmap='gray_r')
    ax.set_title(title)
    ax.set_xlabel('t (sec)')
    ax.set_ylabel('Frequency (Hz)')
    ax.grid(True)

plot_spectrogram(f'linear Chirp, f(0)=10, f({T})=100', w, fs)
plt.show()
```



```
pip install sounddevice
```

```
Requirement already satisfied: sounddevice in
/home/saqib/anaconda3/lib/python3.9/site-packages (0.4.5)
Requirement already satisfied: CFFI>=1.0 in
/home/saqib/anaconda3/lib/python3.9/site-packages (from sounddevice)
(1.15.0)
Requirement already satisfied: pycparser in
/home/saqib/anaconda3/lib/python3.9/site-packages (from CFFI>=1.0-
>sounddevice) (2.20)
Note: you may need to restart the kernel to use updated packages.
```

```
import sounddevice as sd
```

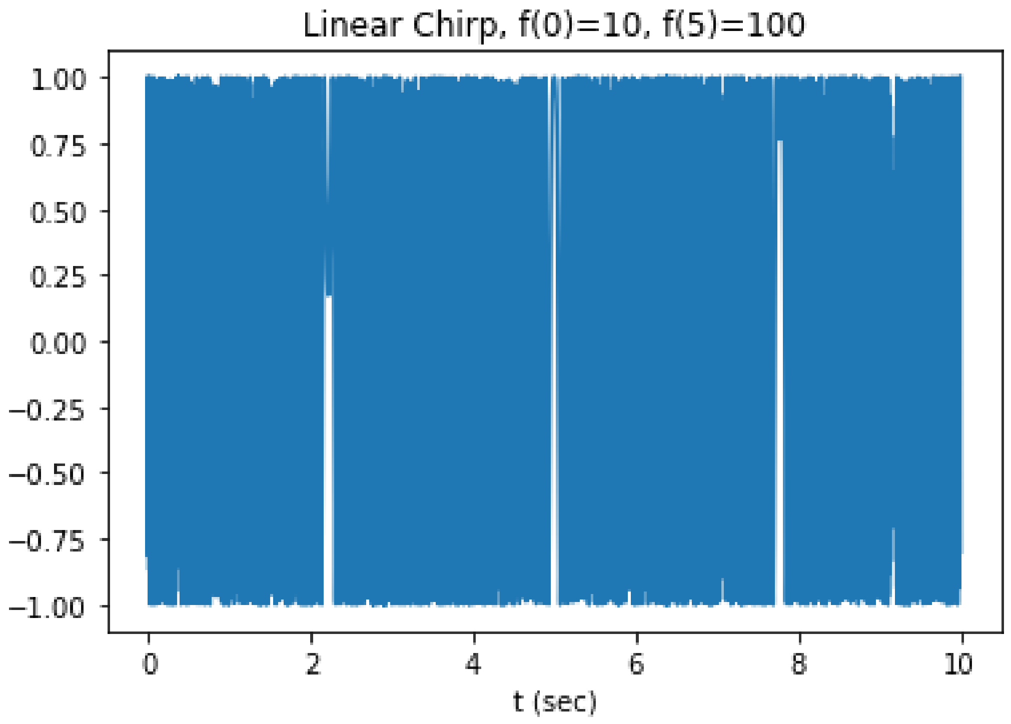
```
sd.play(w,50000)
```

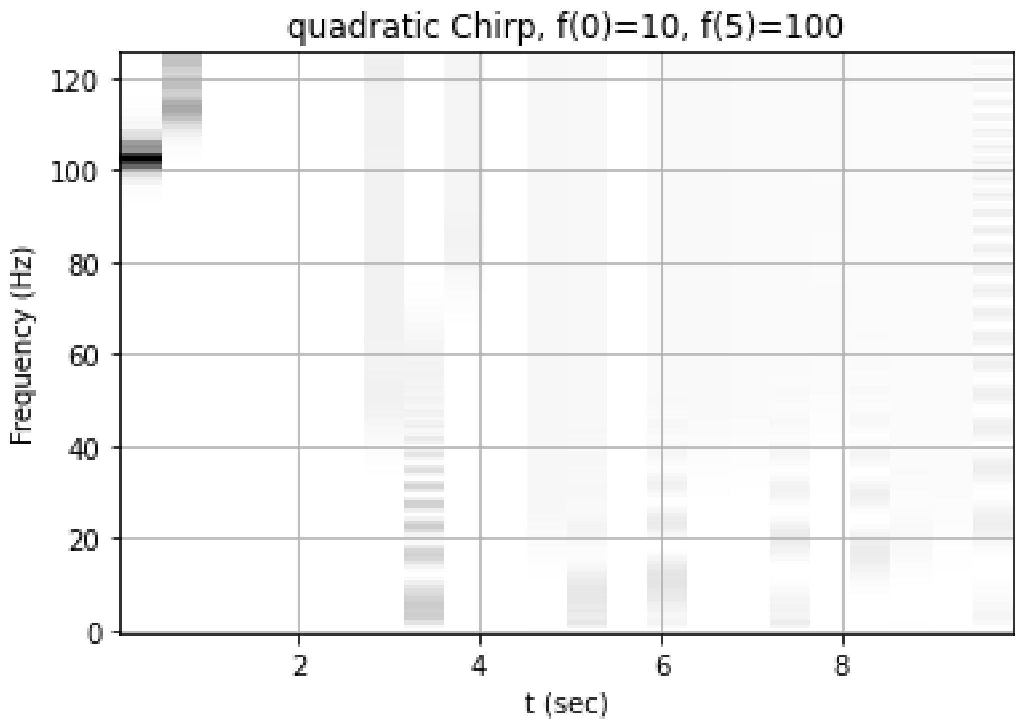
we can hear the signal because an average human has a hearing range for freguencies from f=20Hz to 20kHz.

Quadratic chirping

```
t = np.linspace(0, 10, 5000)
x = chirp(t, f0=100, f1=1000, t1=5, method='quadratic')
plt.plot(t, w)
plt.title("Linear Chirp, f(0)=10, f(5)=100")
plt.xlabel('t (sec)')
plt.show()
```

```
fs = 500
T = 5
t = np.arange(0, int(T*fs)) / fs

plot_spectrogram(f'quadratic Chirp, f(0)=10, f({T})=100', x, fs)
plt.show()

sd.play(x,50000)
```

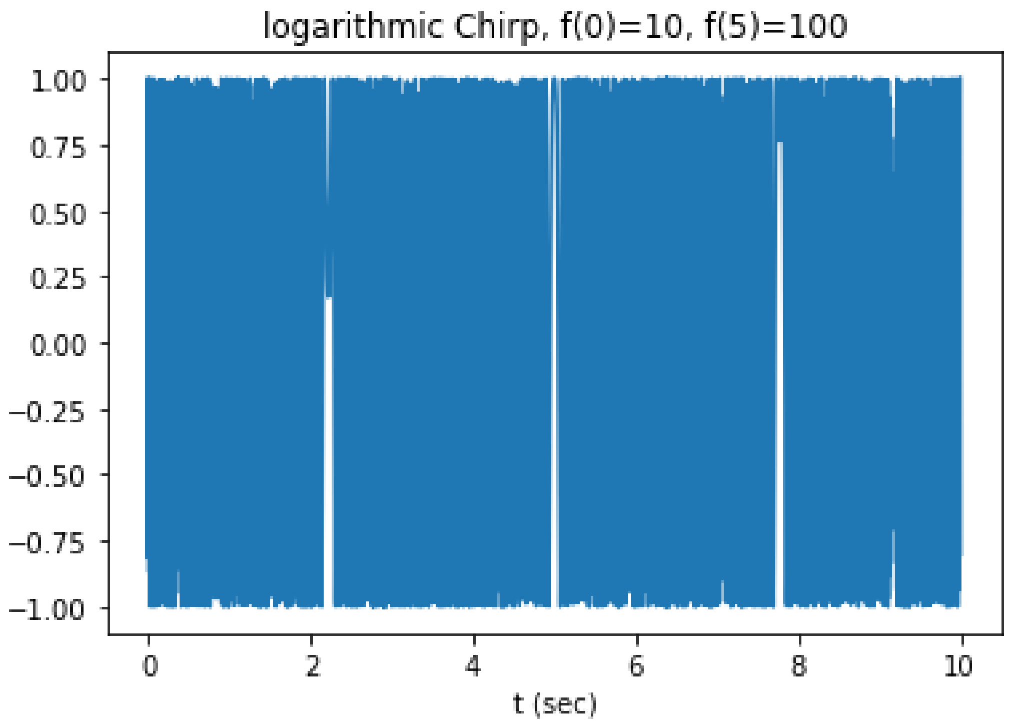


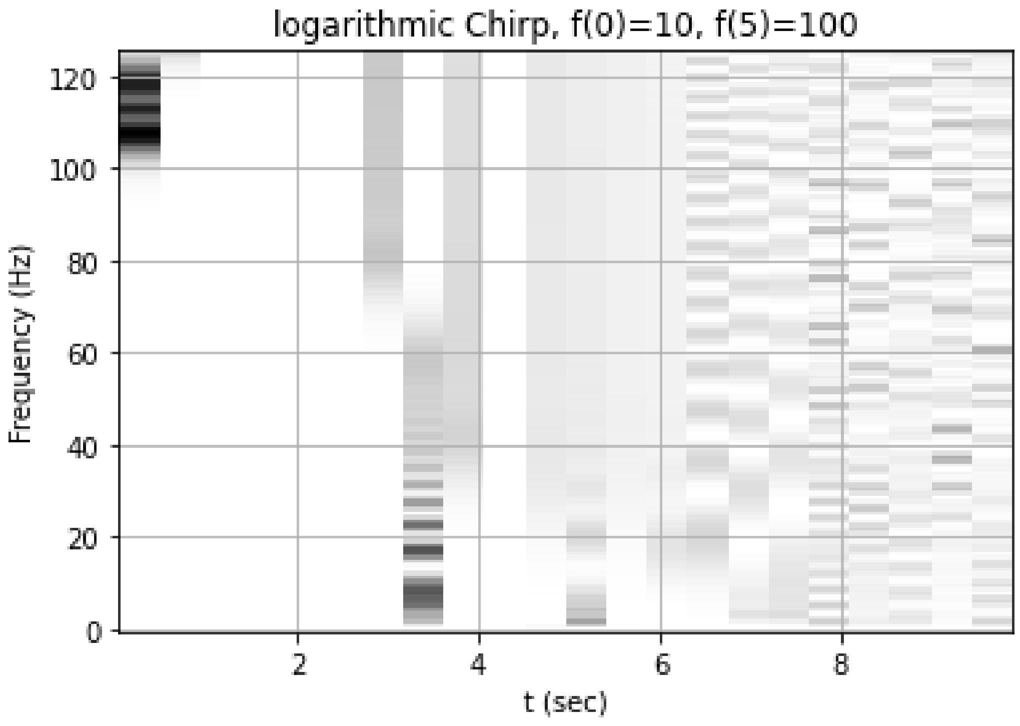


logarithmic chirping

```
t = np.linspace(0, 10, 5000)
y = chirp(t, f0=100, f1=1000, t1=5, method='logarithmic')
plt.plot(t, w)
plt.title("logarithmic Chirp, f(0)=10, f(5)=100")
plt.xlabel('t (sec)')
plt.show()


fs = 500
T = 5
t = np.arange(0, int(T*fs)) / fs

plot_spectrogram(f'logarithmic Chirp, f(0)=10, f({T})=100', y, fs)
plt.show()

sd.play(y,50000)
```

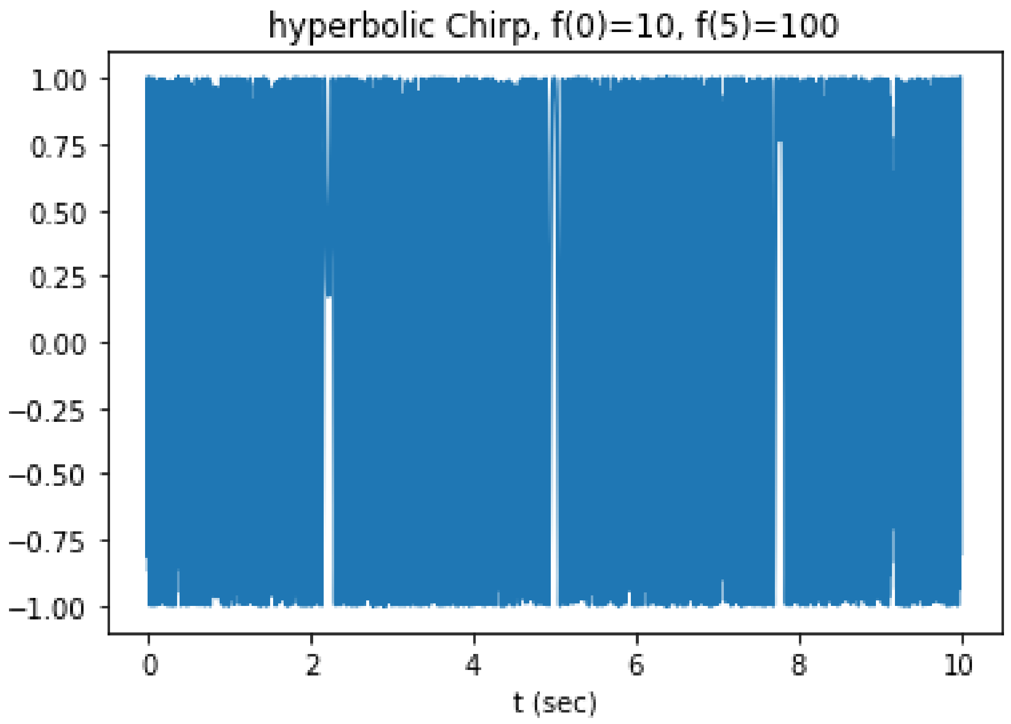
```
ALSA lib pcm.c:8526:(snd_pcm_recover) underrun occurred
```

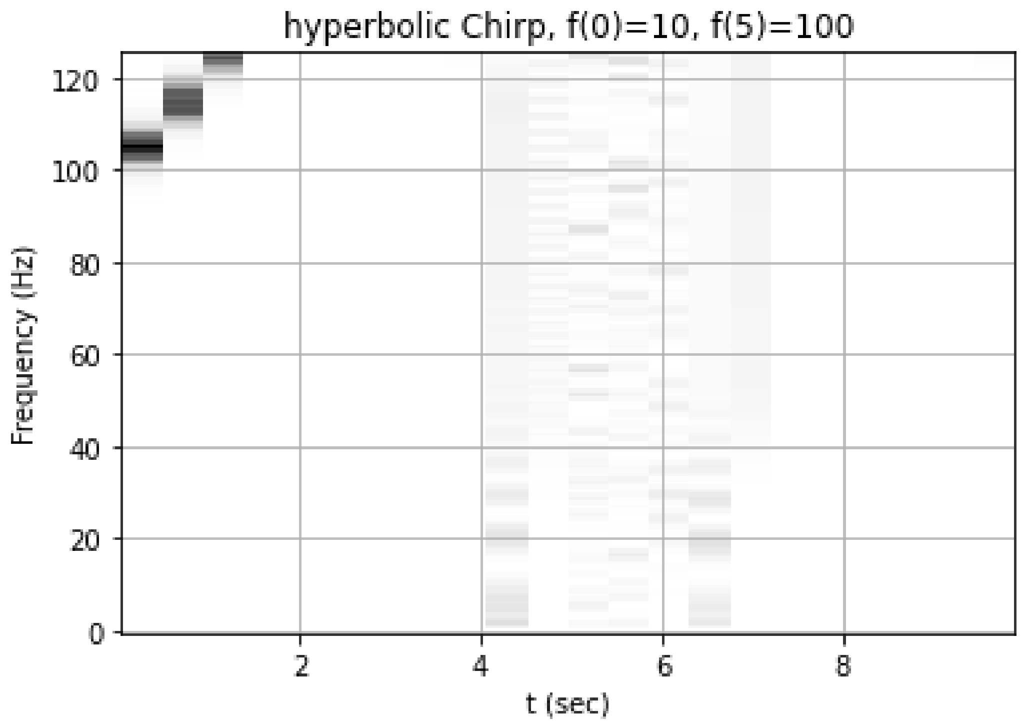




hyperbolic chirping

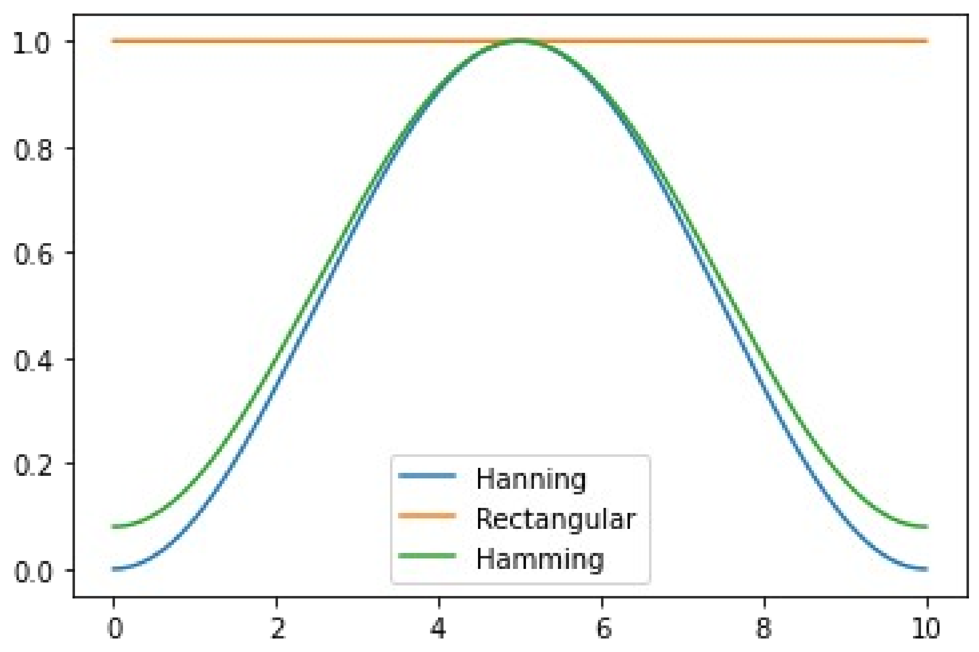
```
t = np.linspace(0, 10, 5000)
z= chirp(t, f0=100, f1=1000, t1=5, method='hyperbolic')
plt.plot(t, w)
plt.title("hyperbolic Chirp, f(0)=10, f(5)=100")
plt.xlabel('t (sec)')
plt.show()


fs = 500
T = 5
t = np.arange(0, int(T*fs)) / fs

plot_spectrogram(f'hyperbolic Chirp, f(0)=10, f({T})=100', z, fs)
plt.show()

sd.play(z,50000)
```



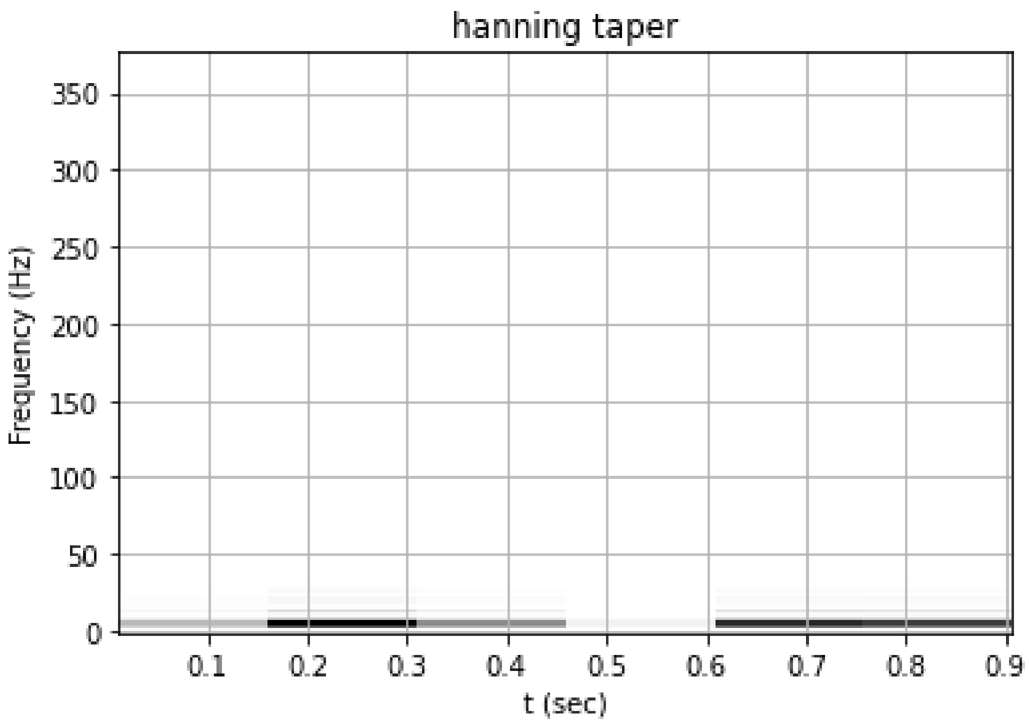


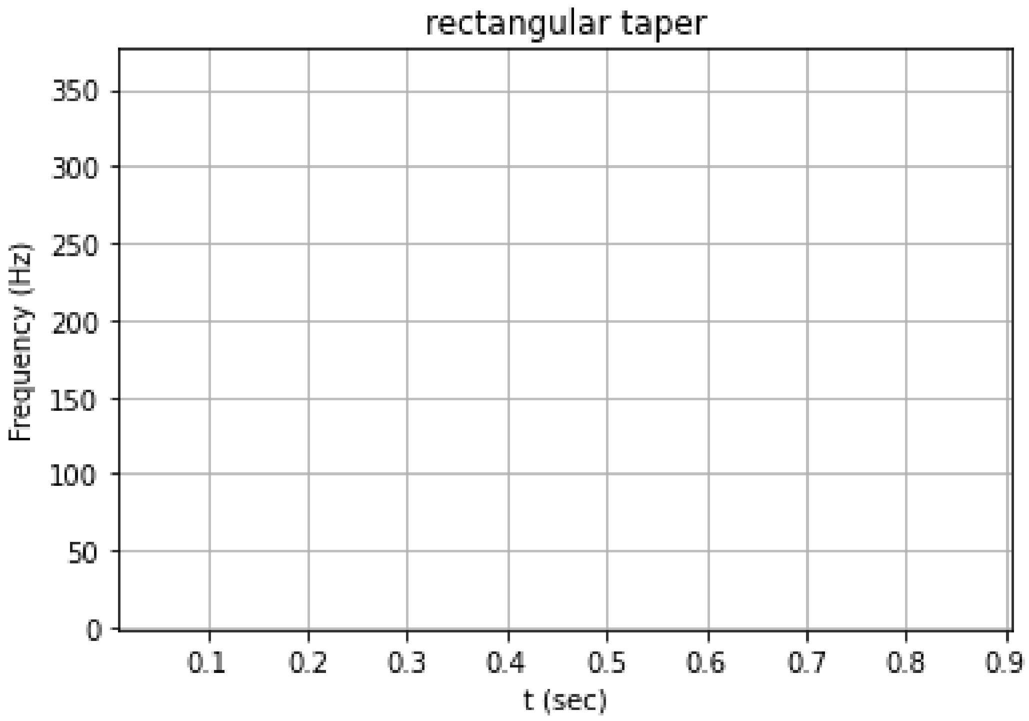
Tapers

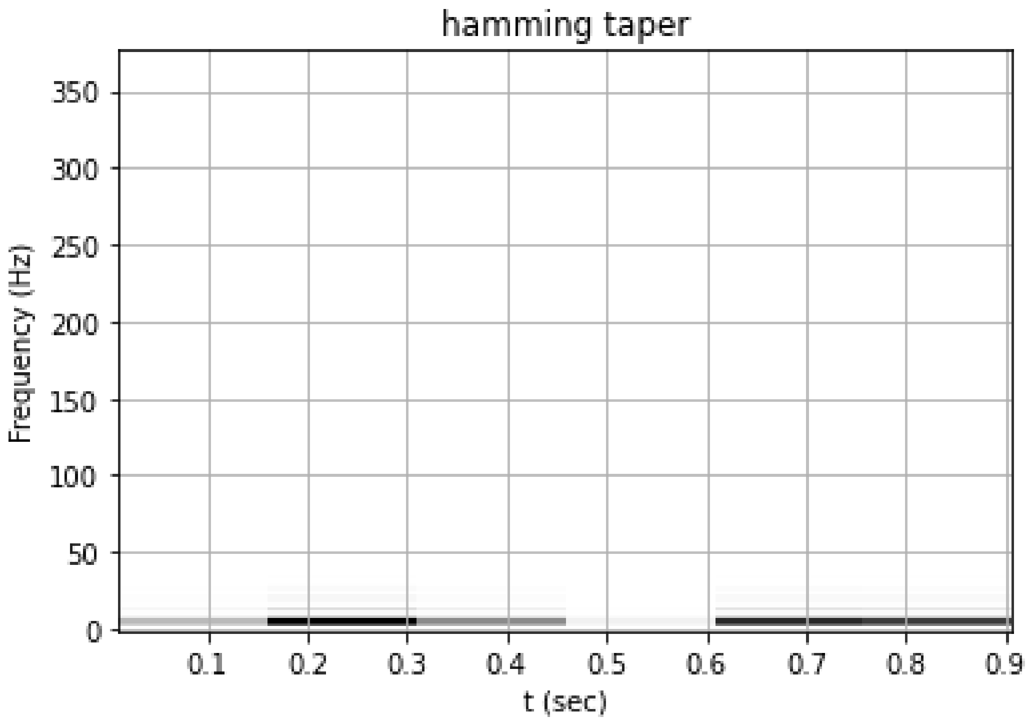
```
import scipy as sp

T=10
Ns=1500
t = np.linspace(0, T, Ns)
fs=3000


hanning_taper=sp.signal.windows.hann(Ns)
rectangular_taper=sp.signal.windows.boxcar(Ns)
hamming_taper=sp.signal.windows.hamming(Ns)
plt.plot(t,hanning_taper,label='Hanning')
plt.plot(t,rectangular_taper,label='Rectangular')
plt.plot(t,hamming_taper,label='Hamming')
plt.legend()
plt.show()
```



```
plot_spectrogram(f'hanning taper',hanning_taper, Ns)
plt.show()

plot_spectrogram(f'rectangular taper',rectangular_taper, Ns)
plt.show()

plot_spectrogram(f'hamming taper',hamming_taper, Ns)
plt.show()
```


hanning taper
Frequency (Hz)
t (sec)
0
50
100
150
200
250
300
350
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9


rectangular taper
Frequency (Hz)
t (sec)
0
50
100
150
200
250
300
350
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

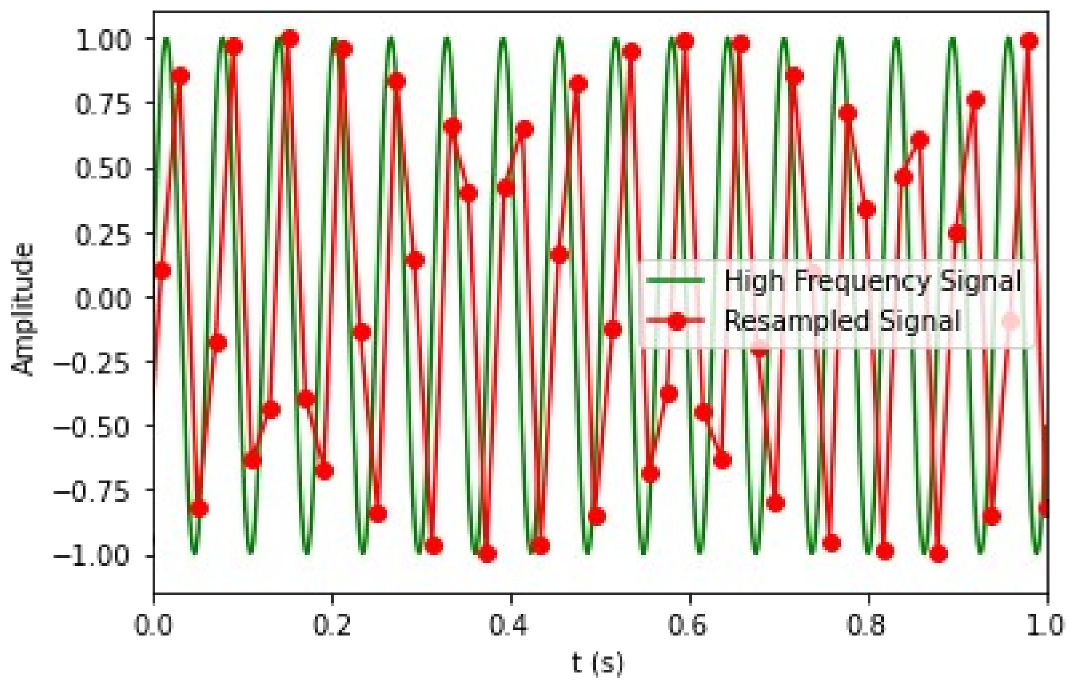


3) Above we can see the difference between the spectra of different windows

4) According to nyquist theorem sampling frequency should be atleast twice the maximum frequency signal present in the system.

5) The wrap-around of frequency refers to the phenomena of interpreting a high frequency signal as a lower frequency signal due to a low sampling rate, or in respect of Nyquist Shannon theorem the high frequency signal when sampled at a frequency lower than twice the highest frequency component of the signal gets interpreted as a low frequency signal.

```
t=np.linspace(-1,1,1000)
high_freq_signal=np.sin(100*t)
tnew=np.linspace(-1,1,100)
resampled_signal=sp.signal.resample(high_freq_signal,100)
plt.plot(t,high_freq_signal,'g-',label='High Frequency Signal')
plt.plot(tnew,resampled_signal,'ro-',label='Resampled Signal')
plt.xlim((0,1))
plt.xlabel('t (s)')
plt.ylabel('Amplitude')
plt.legend()
plt.show()
```



Here we can see that when the sampling frequency is not twice the max freq, the sampled signal is not equivalent to the input signal.

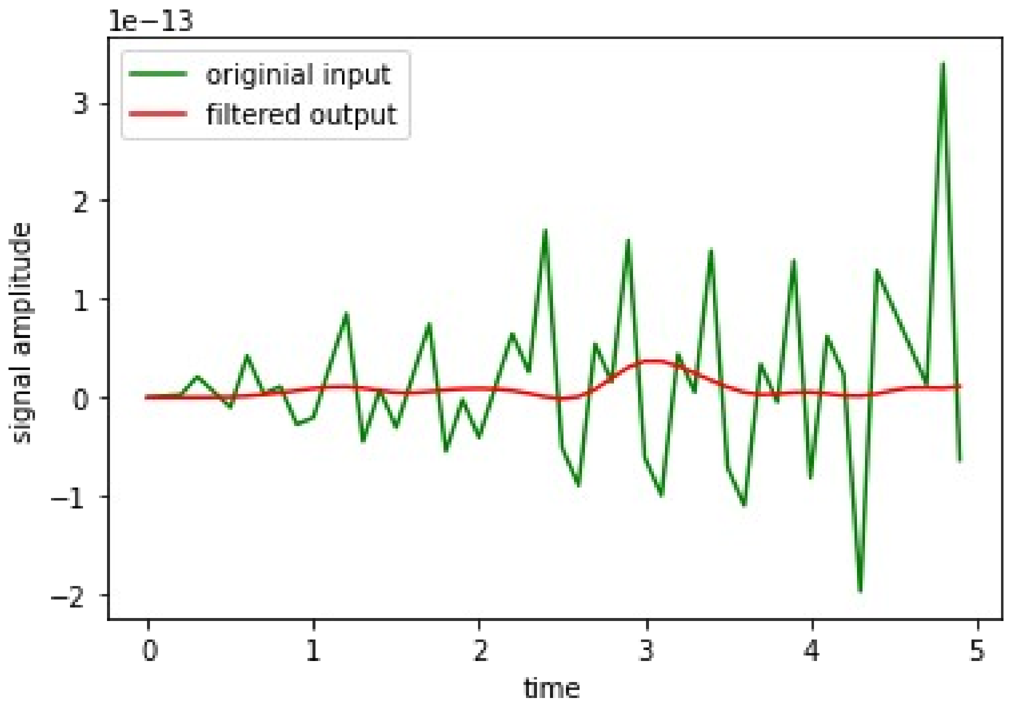
Filter Design

```
import math
import matplotlib.pyplot as plt
import numpy as np
from scipy.fftpack import fftfreq
from scipy.fftpack import fft, ifft
from scipy.signal import chirp,spectrogram
import scipy as sp
import sounddevice as sd
from scipy.signal import butter, lfilter,freqz
```

Lowpass filter

```
def butter_lowpass(cutoff, fs, order=5):
    nyquist = 0.5 * fs
    normal_cutoff = cutoff / nyquist
    b, a = butter(order, normal_cutoff, btype='low', analog=False)
    return b, a

def butter_lowpass_filter(data, cutoff, fs, order=5):
    b, a = butter_lowpass(cutoff, fs, order=order)
    y = lfilter(b, a, data)
    return y

fs = 1000
```

```
cutoff = 100
T = 5.0
nsamples = T * fs
t = np.linspace(0, T, 50, endpoint=False)
a = 0.02
f0 = 600.0
x = 0.1*np.sin(2*np.pi*f0*t)

y = butter_lowpass_filter(x, cutoff, fs, order=6)

plt.xlabel('time')
plt.ylabel('signal amplitude')
plt.plot(t,x,'g',label='originial input')
plt.plot(t,y,'r',label='filtered output')
plt.legend()
plt.show()
```

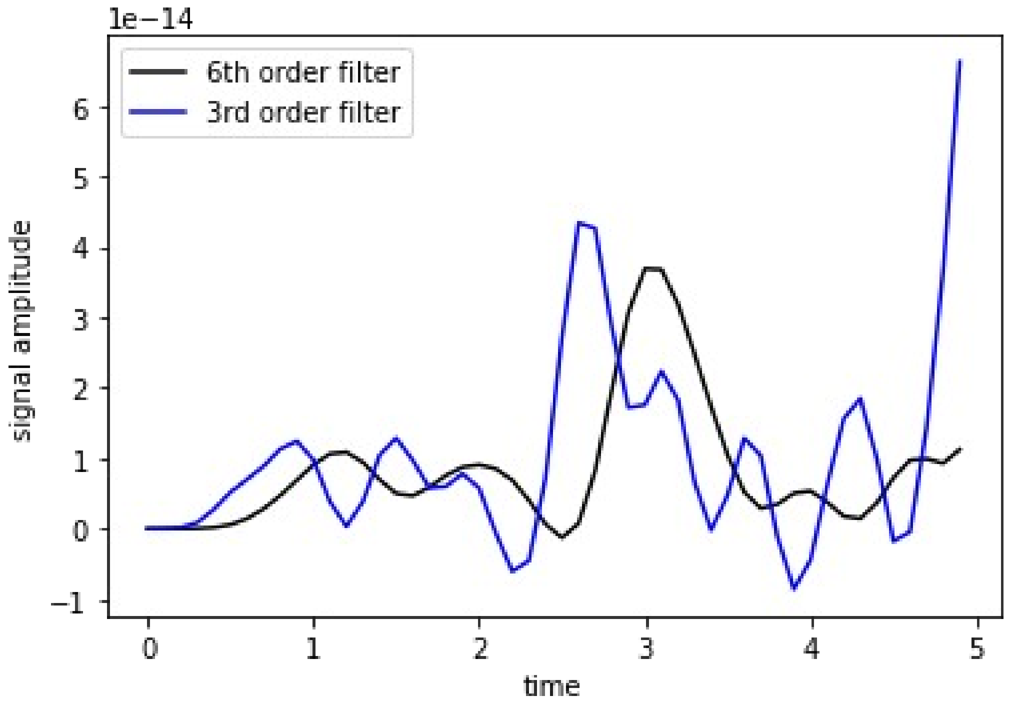


Here we can see that the filter only passes the low frequency input (The filter above is a 6th order filter).

the code for the filter has 2 functions defined. The butter_lowpass function returns the coeeficients a and b which are then passed to the butter_lowpass_filter function.

```
fs = 1000
cutoff = 100
T = 5.0
nsamples = T * fs
t = np.linspace(0, T, 50, endpoint=False)
a = 0.02
```

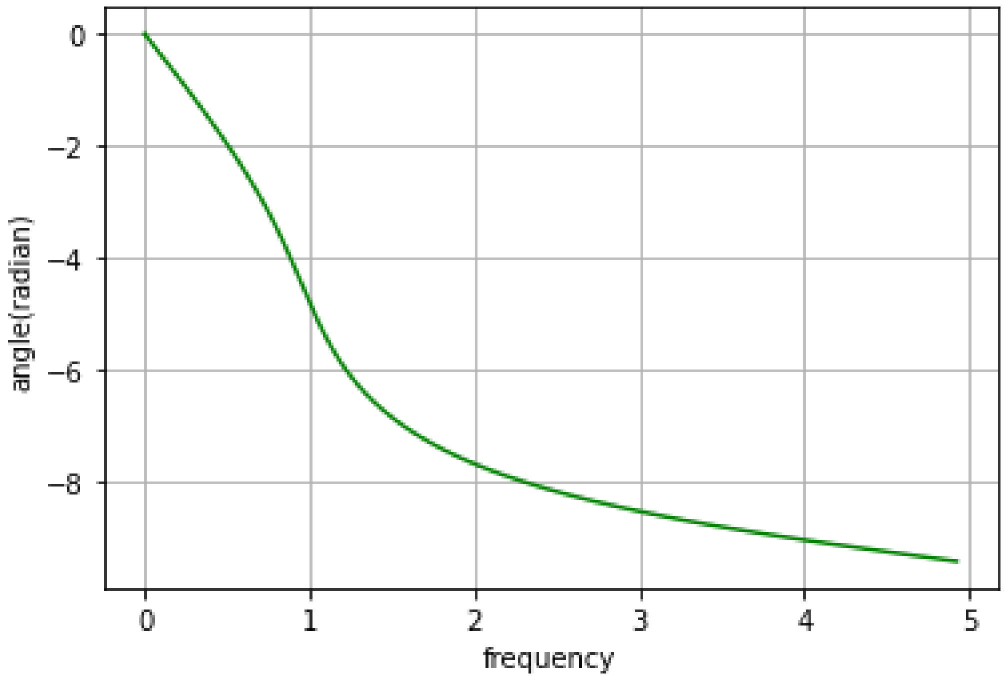
```
f0 = 600.0
x = 0.1*np.sin(2*np.pi*f0*t)

z = butter_lowpass_filter(x, cutoff, fs, order=3)

plt.xlabel('time')
plt.ylabel('signal amplitude')
plt.plot(t,y,'black',label='6th order filter')
plt.plot(t,z,'blue',label='3rd order filter')
plt.legend()
plt.show()
```



We can see in the above plot that the attenuation in the stopband is higher for the 6th order(higher order) filter. Hence we can infer that a higher order filter will have more attenuation in the stop band for lowpass filter. Therefore we can also say that the roll off rate increases with an increase in the order of the filter.

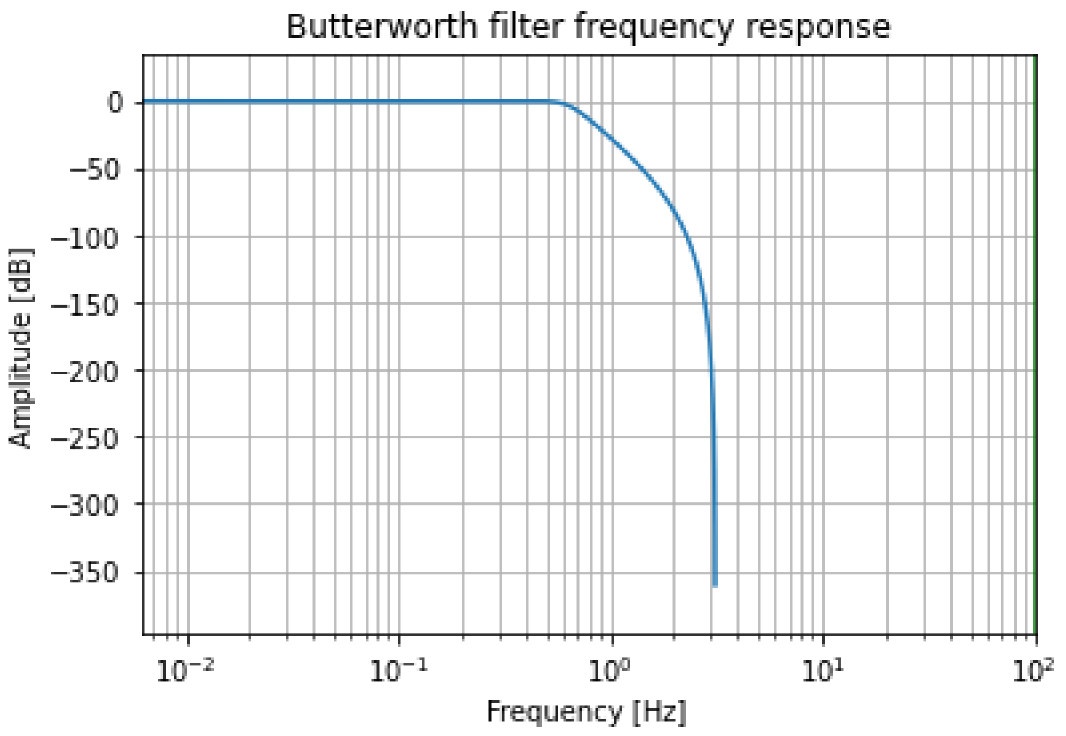
```
b, a = butter_lowpass(cutoff, fs, order=6)
w, h = freqz(b, a)

phase = np.unwrap(np.angle(h))

fig, ax1 = plt.subplots()
ax1.plot(w/2*np.pi, phase, 'g')
ax1.grid()
ax1.axis('tight')
ax1.set_xlabel('frequency')
```

```
ax1.set_ylabel('angle(radian)')
plt.show()
```



Above plot is the phase response of the lowpass filter

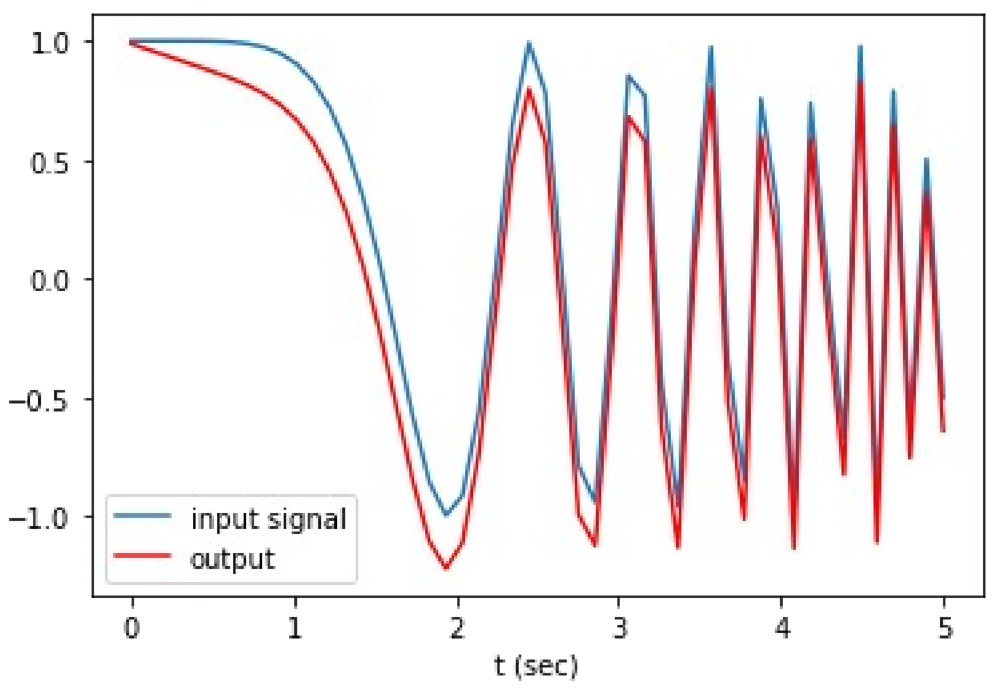
```
plt.semilogx(w, 20*np.log10(abs(h)))
plt.xscale('log')
plt.title('Butterworth filter frequency response')
plt.xlabel('Frequency [Hz]')
plt.ylabel('Amplitude [dB]')
plt.margins(0, 0.1)
plt.grid(which='both', axis='both')
plt.axvline(100, color='green')
plt.show()
```



Above plot if the freq response of the lowpass filter

High pass filter

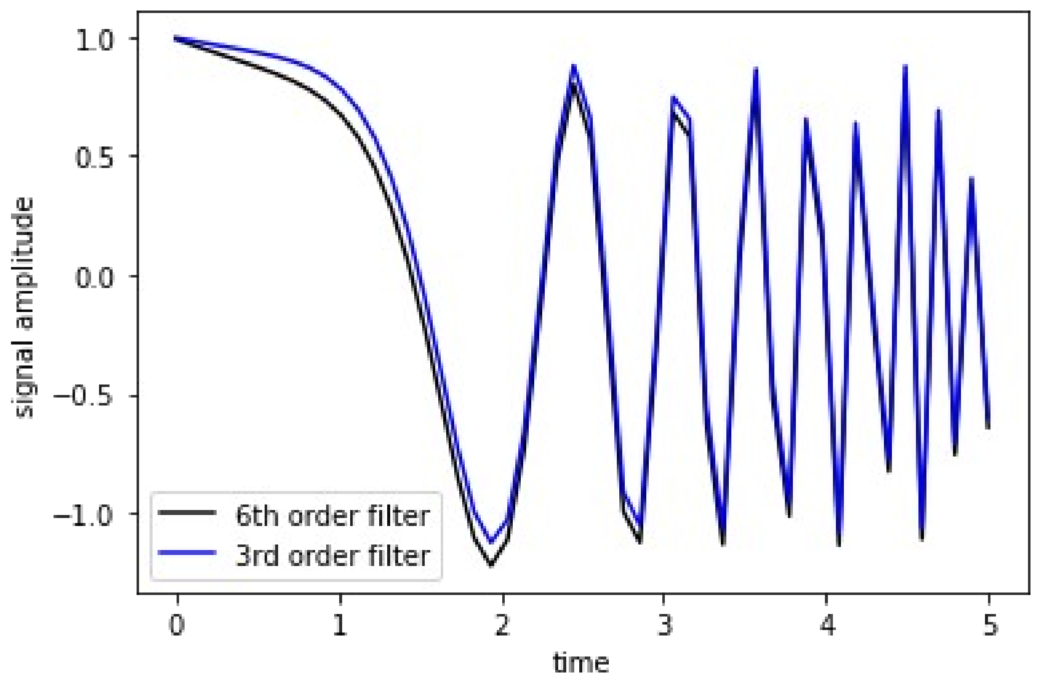
```
def butter_highpass(cutoff, fs, order=5):
    nyq = 0.5 * fs
    normal_cutoff = cutoff / nyq
    b, a = butter(order, normal_cutoff, btype='high', analog=False)
    return b, a

def butter_highpass_filter(data, cutoff, fs, order=5):
    b, a = butter_highpass(cutoff, fs, order=order)
    y = lfilter(b, a, data)
    return y


fs = 1000
cutoff = 1

t=np.linspace(0,5,50)
w= chirp(t, f0=0.0001, f1=5, t1=5, method='quadratic')
z=butter_highpass_filter(w, cutoff, fs,order=6)
plt.plot(t, w,label='input signal')
plt.plot(t,z,'r',label='output')
plt.xlabel('t (sec)')
plt.legend()
plt.show()
```



Here we can see that the filter only passes the high frequency input (The filter above is a 6th order filter).

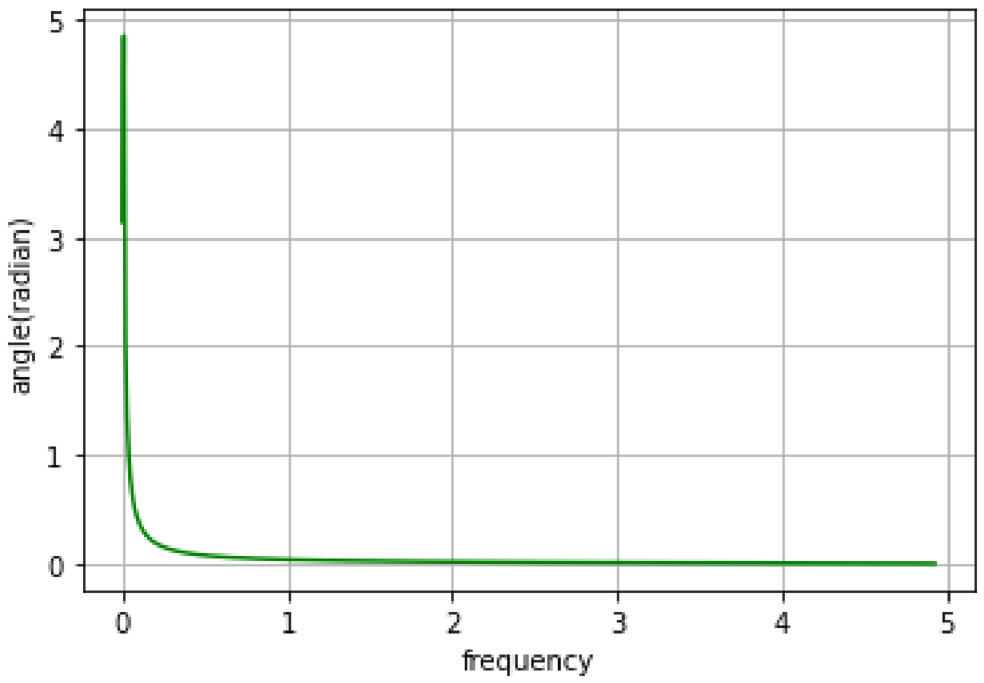
the code for the filter has 2 functions defined. The butter_highpass function returns the coeeficients a and b which are then passed to the butter_highpass_filter function.

```
z1= butter_highpass_filter(w, cutoff, fs,order=3)
plt.xlabel('time')
plt.ylabel('signal amplitude')
plt.plot(t,z,'black',label='6th order filter')
plt.plot(t,z1,'blue',label='3rd order filter')
plt.legend()
plt.show()
```



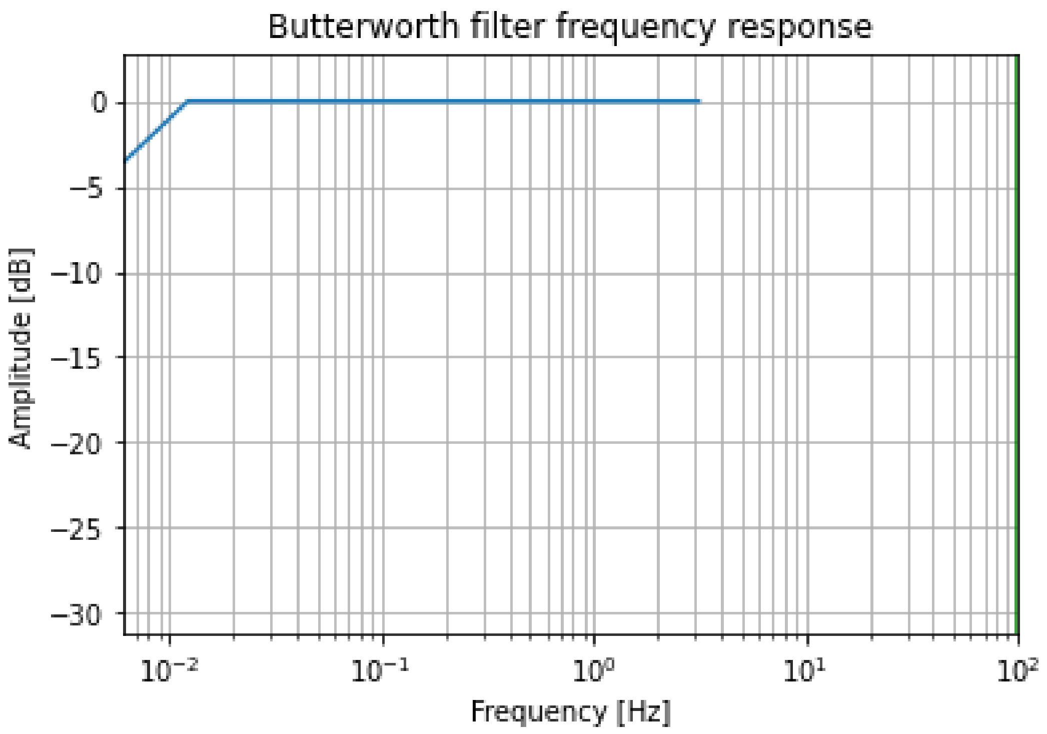
We can see in the above plot that the attenuation in the stopband is higher for the 6th order(higher order) filter. Hence we can infer that a higher order filter will have more attenuation in the stop band for highpass filter. Therefore we can also say that the roll off rate increases with an increase in the order of the filter.

```
b, a = butter_highpass(cutoff, fs, order=6)
w, h = freqz(b, a)

phase = np.unwrap(np.angle(h))

fig, ax1 = plt.subplots()
ax1.plot(w/2*np.pi, phase, 'g')
ax1.grid()
ax1.axis('tight')
ax1.set_xlabel('frequency')
ax1.set_ylabel('angle(radian)')
plt.show()
```



Above plot is the phase response of highpass filter

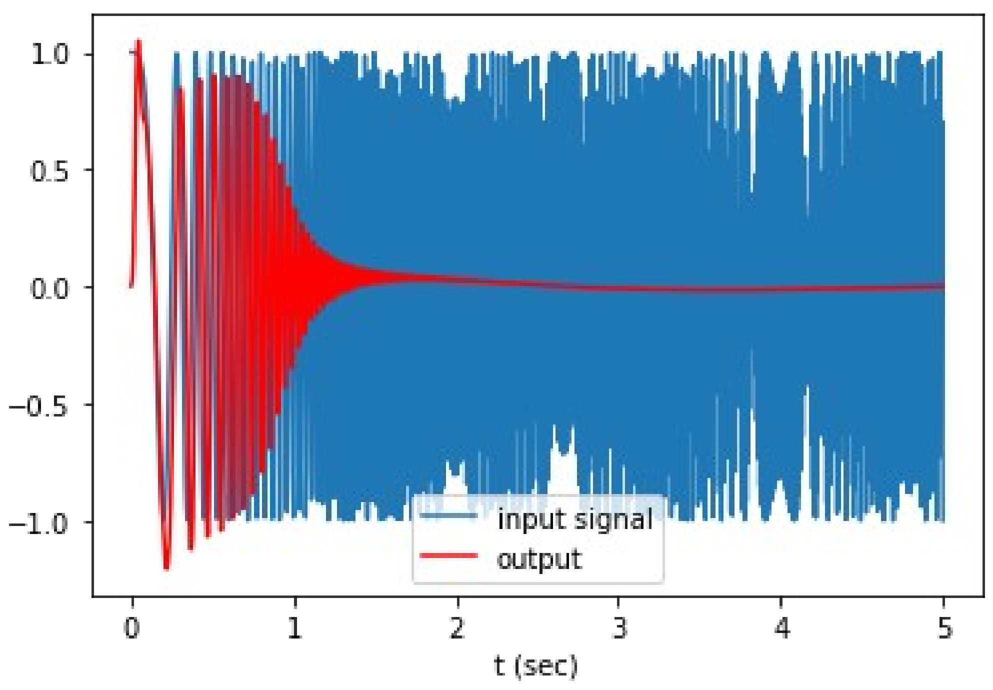
```
plt.semilogx(w, 20*np.log10(abs(h)))
plt.xscale('log')
plt.title('Butterworth filter frequency response')
plt.xlabel('Frequency [Hz]')
plt.ylabel('Amplitude [dB]')
plt.margins(0, 0.1)
plt.grid(which='both', axis='both')
plt.axvline(100, color='green')
plt.show()
```



Above plot is the freq response of the high pass filter

Band pass filter

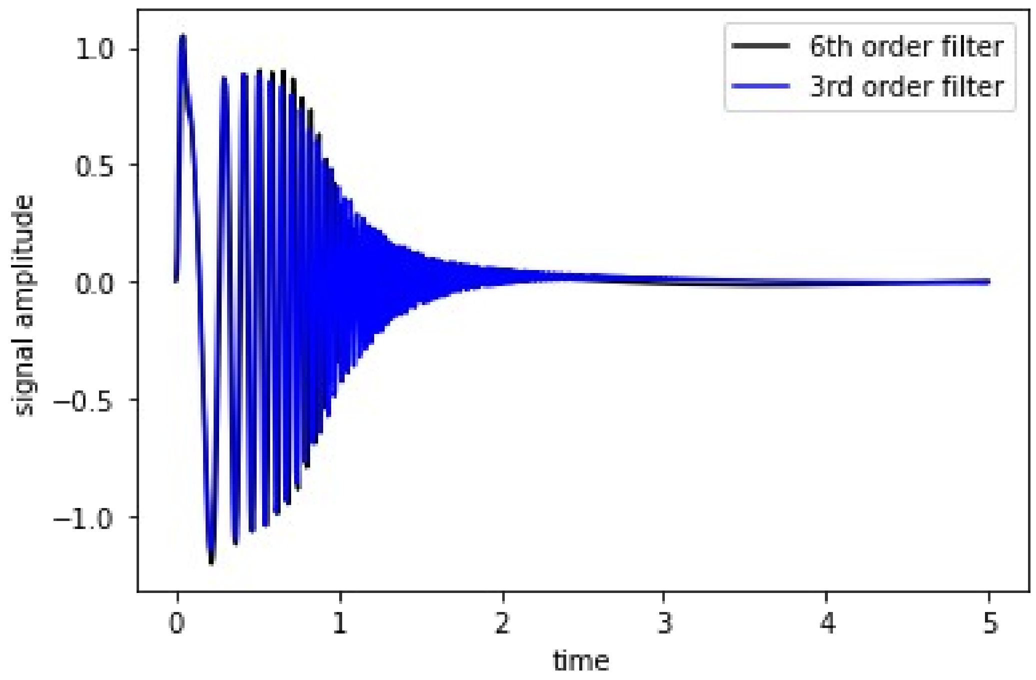
```
def butter_bandpass(lowcut, highcut, fs, order=5):
    nyq = 0.5 * fs
    low = lowcut / nyq
    high = highcut / nyq
    b, a = butter(order, [low, high], btype='band')
    return b, a

def butter_bandpass_filter(data, lowcut, highcut, fs, order=5):
    b, a = butter_bandpass(lowcut, highcut, fs, order=order)
    y = lfilter(b, a, data)
    return y

fs = 1000
lowcut = 1
highcut = 100
t=np.linspace(0,5,1000)
w= chirp(t, f0=0.0001, f1=250, t1=10, method='linear')
z= butter_bandpass_filter(w, lowcut, highcut, fs)
plt.plot(t, w,label='input signal')
plt.plot(t,z,'r',label='output')
plt.xlabel('t (sec)')
plt.legend()
plt.show()
```



Here we can see that the filter only passes a band frequency input (The filter above is a 6th order filter).

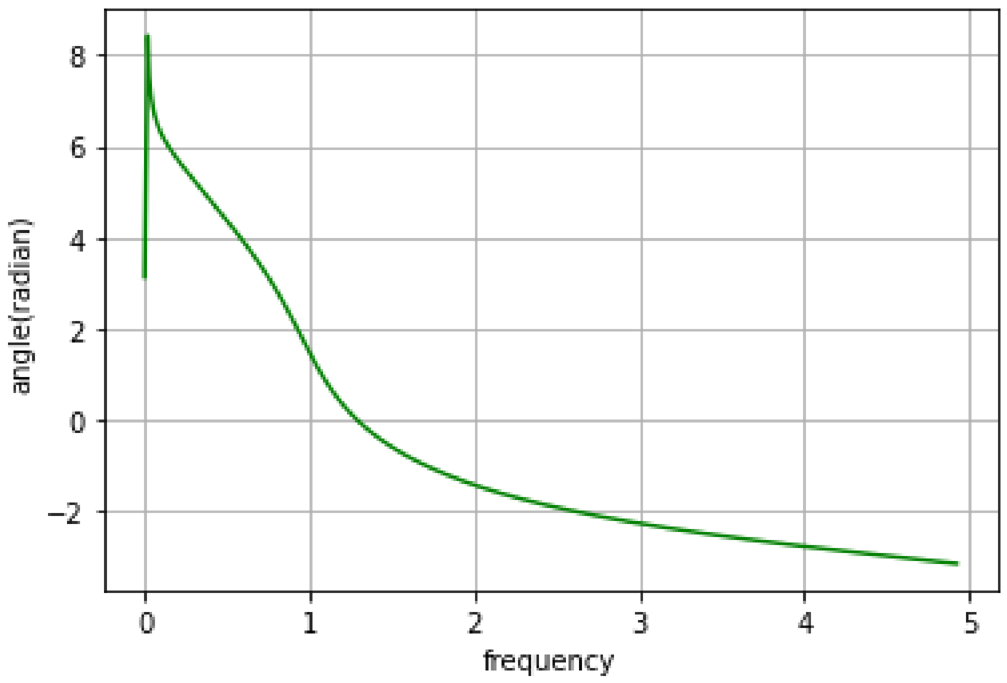
the code for the filter has 2 functions defined. The butter_bandpass function returns the coeeficients a and b which are then passed to the butter_bandpass_filter function.

```
z1= butter_bandpass_filter(w, lowcut,highcut, fs,order=3)
plt.xlabel('time')
plt.ylabel('signal amplitude')
plt.plot(t,z,'black',label='6th order filter')
plt.plot(t,z1,'blue',label='3rd order filter')
plt.legend()
plt.show()
```



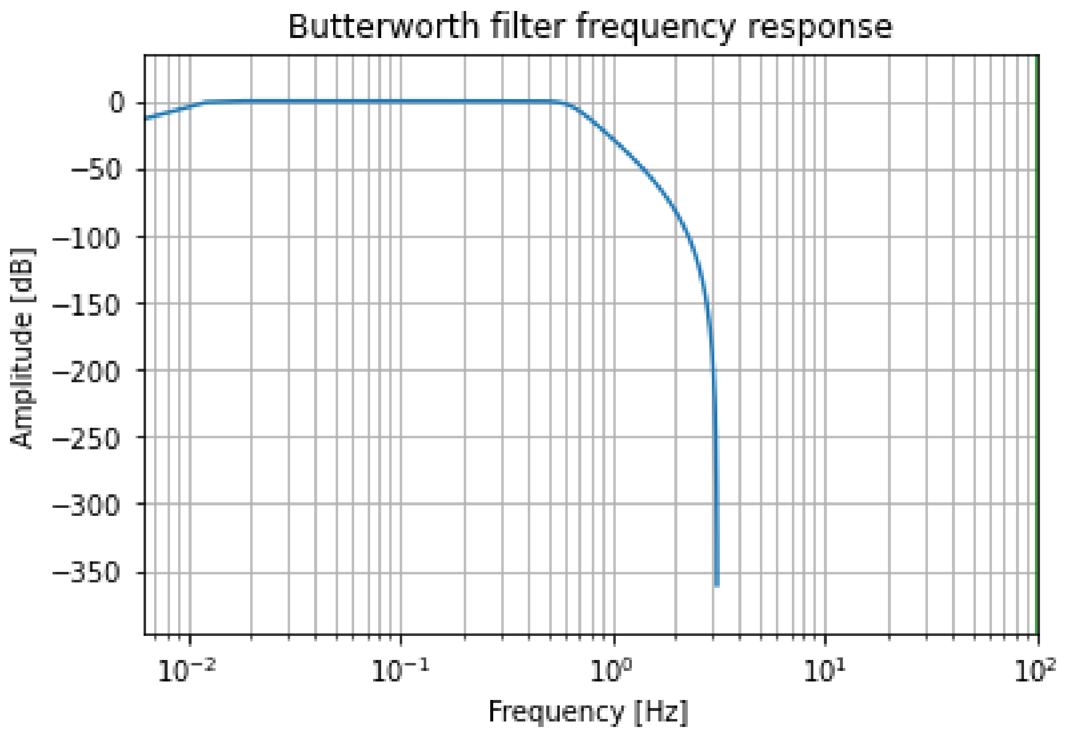
We can see in the above plot that the attenuation in the stopband is higher for the 6th order(higher order) filter. Hence we can infer that a higher order filter will have more attenuation in the stop band for bandpass filter. Therefore we can also say that the roll off rate increases with an increase in the order of the filter.

```
b, a = butter_bandpass(lowcut,highcut, fs, order=6)
w, h = freqz(b, a)

phase = np.unwrap(np.angle(h))

fig, ax1 = plt.subplots()
ax1.plot(w/2*np.pi, phase, 'g')
ax1.grid()
ax1.axis('tight')
ax1.set_xlabel('frequency')
ax1.set_ylabel('angle(radian)')
plt.show()
```



phaseresponse of the bandpass filter

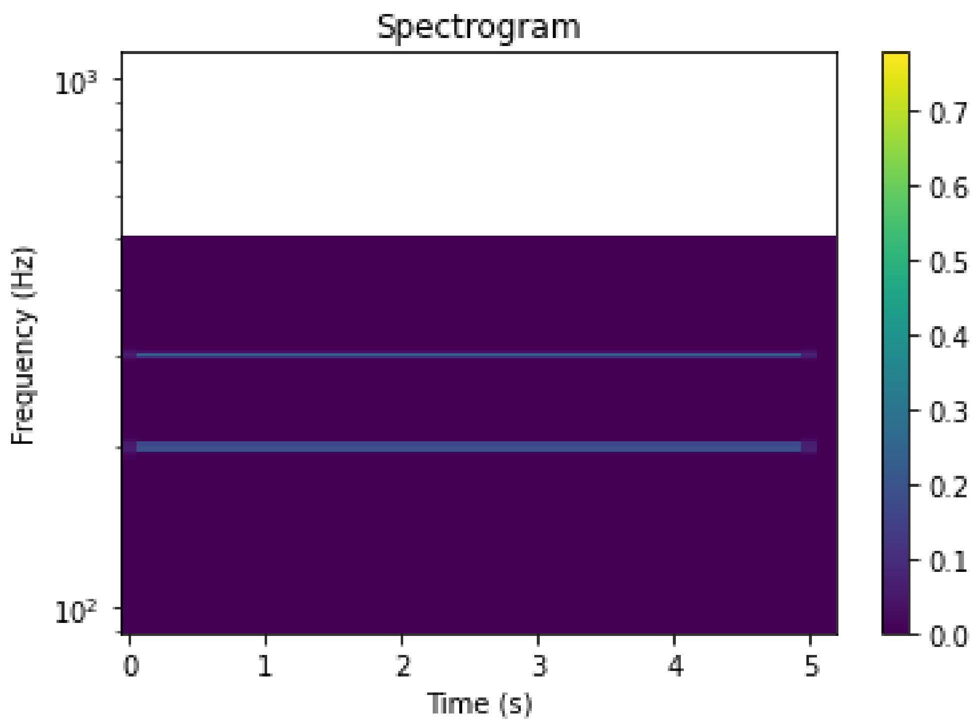
```
plt.semilogx(w, 20*np.log10(abs(h)))
plt.xscale('log')
plt.title('Butterworth filter frequency response')
plt.xlabel('Frequency [Hz]')
plt.ylabel('Amplitude [dB]')
plt.margins(0, 0.1)
plt.grid(which='both', axis='both')
plt.axvline(100, color='green')
plt.show()
```



freq response of the bandpass filter

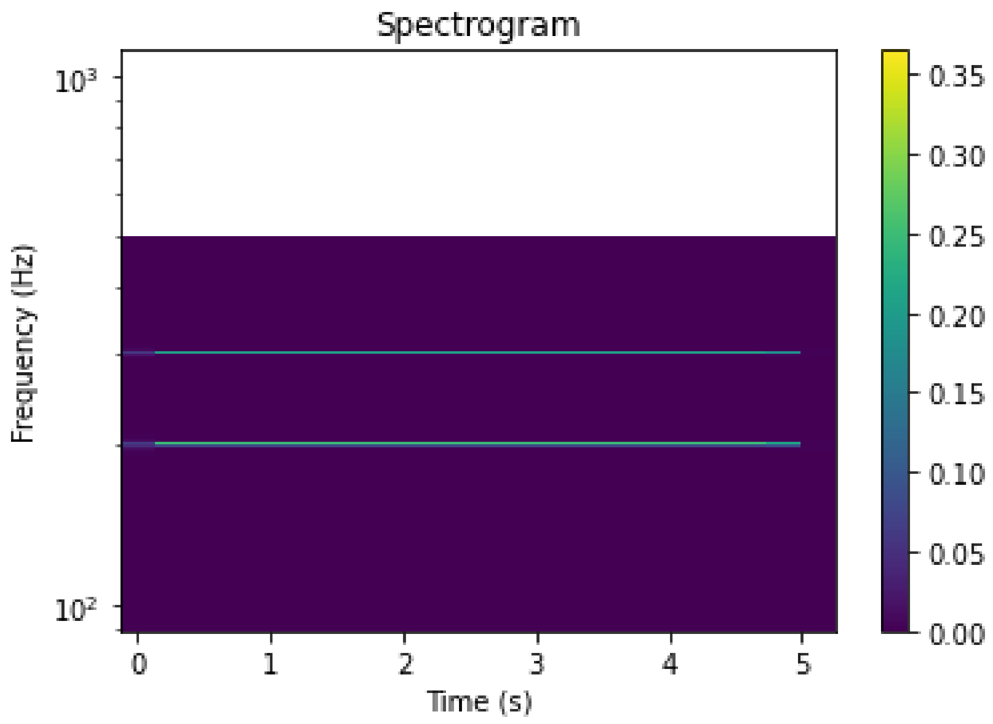
STFT

```
import numpy as np
from scipy.fftpack import fft
from scipy.signal import stft


fs = 1000
t = np.linspace(0, 5, 5000)
f = 300
x = np.sin(2 * np.pi * f * t) + np.sin(2*np.pi *f*0.67*t)
+np.sin(2*np.pi*1.67*t)

f, t, Zxx = stft(x, fs=fs)

tfr = np.abs(Zxx) ** 2

plt.pcolormesh(t, f, tfr)
plt.title('Spectrogram')
plt.xlabel('Time (s)')
plt.ylabel('Frequency (Hz)')
plt.yscale('log')
plt.colorbar()
plt.show()
```



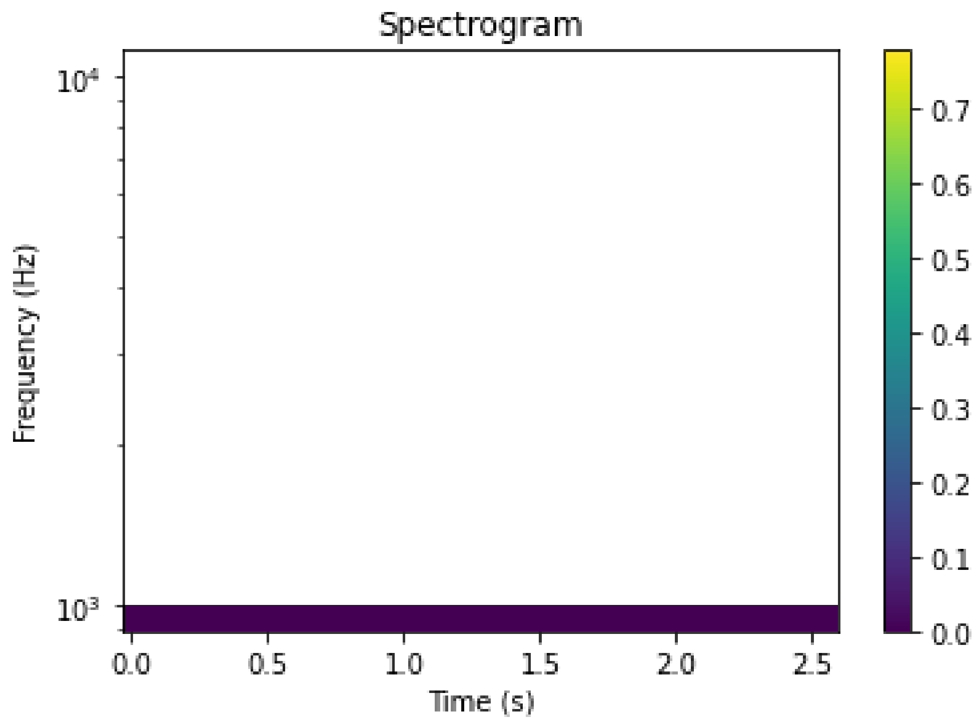
Here we see a horizontal line which indicates the presence of a signal of a particular frequency at that time in the signal

```
import numpy as np
from scipy.fftpack import fft
from scipy.signal import stft


fs = 1000
t = np.linspace(0, 5, 5000)
f = 300
x = np.sin(2 * np.pi * f * t) + np.sin(2*np.pi *f*0.67*t)
+np.sin(2*np.pi*1.67*t)

f, t, Zxx = stft(x, fs=fs,nperseg=512)

tfr = np.abs(Zxx) ** 2

plt.pcolormesh(t, f, tfr)
plt.title('Spectrogram')
plt.xlabel('Time (s)')
plt.ylabel('Frequency (Hz)')
plt.yscale('log')
plt.colorbar()
plt.show()
```



By doubling the window size, we see that the frequency response is much crisper

```
import numpy as np
from scipy.fftpack import fft
from scipy.signal import stft


fs = 2000
t = np.linspace(0, 5, 5000)
f = 300
x = np.sin(2 * np.pi * f * t) + np.sin(2*np.pi *f*0.67*t)
+np.sin(2*np.pi*1.67*t)

f, t, Zxx = stft(x, fs=fs)

tfr = np.abs(Zxx) ** 2


plt.pcolormesh(t, f, tfr)
plt.title('Spectrogram')
plt.xlabel('Time (s)')
plt.ylabel('Frequency (Hz)')
plt.yscale('log')
plt.colorbar()
plt.show()
```



We can double the freq response by doubling the sample rate and keeping the window size same. Essentially, we have to capture twice the number of signal inputs for double resolution. We can it by either doubling the sample rate or doubling the window size.