

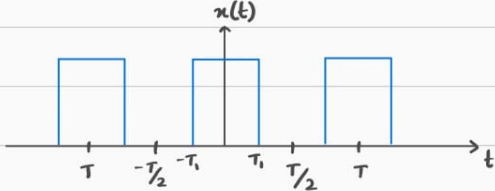
S2 Workshop 2

Signals in the Frequency Domain

by 220046R (Asuramuni S Y) and 220051D (Attanayake A Y) submitted on 13/03/2024

2.1 Fourier Series Approximation

The Fourier Series coefficients for the square wave given in Figure 1 is as below:


$$T_1 = \frac{T}{4} \quad \omega_0 = \frac{2\pi}{T}$$
$$a_0 = \frac{1}{T} \int_T x(t) dt = \frac{1}{T} \int_{-T/4}^{T/4} 1 dt = \frac{T/2}{T} = \frac{1}{2} //$$
$$a_k = \frac{1}{T} \int_T x(t) \cdot e^{-jk\omega_0 t} dt = \frac{1}{T} \int_{-T/4}^{T/4} (1) \cdot e^{-jk\omega_0 t} dt = \frac{1}{T} \left[\frac{e^{-jk\omega_0 t}}{-jk\omega_0} \right]_{-T/4}^{T/4}$$
$$= \frac{1}{T} \times \frac{e^{jk \cdot \frac{2\pi}{T} \cdot \frac{T}{4}} - e^{-jk \cdot \frac{2\pi}{T} \cdot \frac{T}{4}}}{jk \cdot \frac{2\pi}{T}}$$
$$= \frac{1}{k\pi} \times \sin\left(\frac{k\pi}{2}\right) //$$

Importing the required packages:

```
In [ ]: # Imports
import numpy as np
import matplotlib.pyplot as plt
from scipy.fftpack import fft,fftshift,ifft
from scipy import signal
```

Q: Taking $A = 1$ V, $T = 1$ s the Fourier series coefficients of the square wave for given any integer value of k is coded as follows:

```
In [ ]: # Square pulse
def square(t):
    if t % 1 < 0.25 or t % 1 > 0.75:
        s=1
    elif t % 1 == 0.25 or t % 1 == 0.75:
```

```

        s = 0.5
    else:
        s=0
    return s

# Fourier series coefficients
def a(k):
    # For k = 0 instance
    if k == 0:
        # a_k = Average value of the function
        a_k = 0.5
    else:
        # for k != 0 use coefficient formula
        a_k = np.sin(k * np.pi/2) / (k * np.pi)
    return a_k

```

the function `fs_approx(t,N)` is to be coded as follows where in return the value of a Fourier series approximated periodic signal, at any given time.

```

In [ ]: def fs_approx(t, N):
        x_t = 0.0

        # Parameters
        T = 1.0 # Period
        w0 = 2 * np.pi / T # Fundamental frequency

        # Looping through -N to N and add individual complex exponentials
        for i in range(-N, N+1):
            x_t += (a(i)*np.e**(1j*i*w0*t)).real
        return x_t

```

The updated python script according to the given guidelines is as follows:

```

In [ ]: # Fourier series approximation of the square wave
x = []
y = []
N = 5 # CHANGE HERE

# Creating a timestamp array
time = np.linspace(-2.5, 2.5,1000)

# Filling x and y arrays
for t in time:
    x.append(square(t)) #The true square wave function
    y.append(fs_approx(t,N)) #The Fourier series approximation

```

The following Python script plots the original signal, $x(t)$ and the approximated signal, $x_N(t)$ in the same figure for $N = 5$:

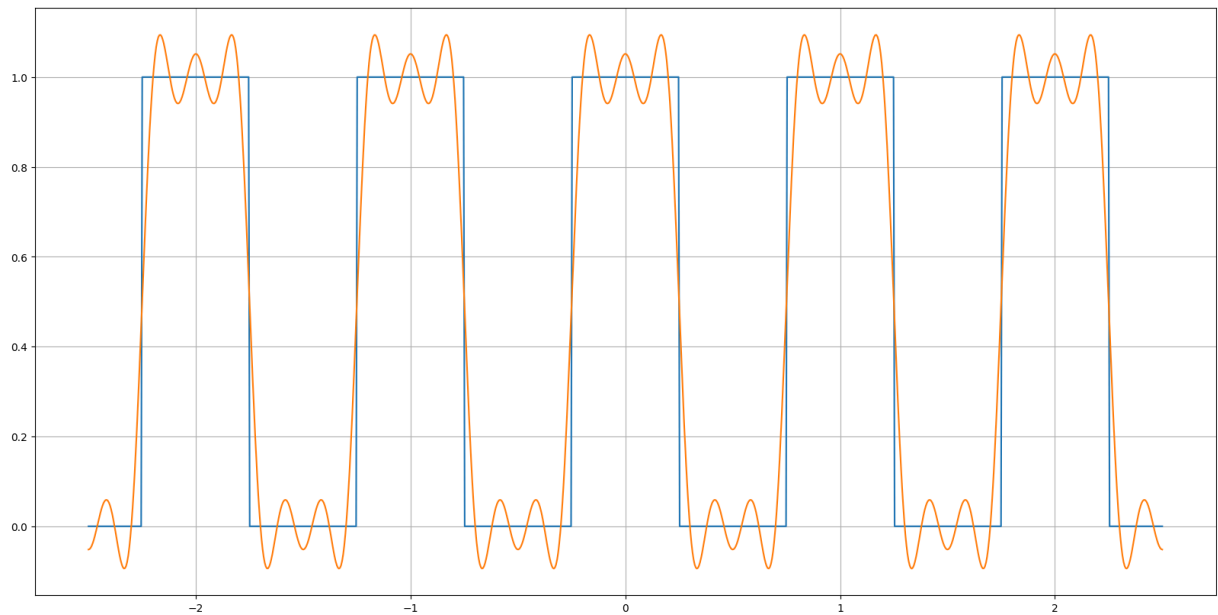
```

In [ ]: # Plotting the square wave and its Fourier series approximation
fig,ax=plt.subplots(figsize=(20,10))

# Editing plot parameters
ax.plot(time,x)

```

```
ax.plot(time,y)
ax.grid()
```



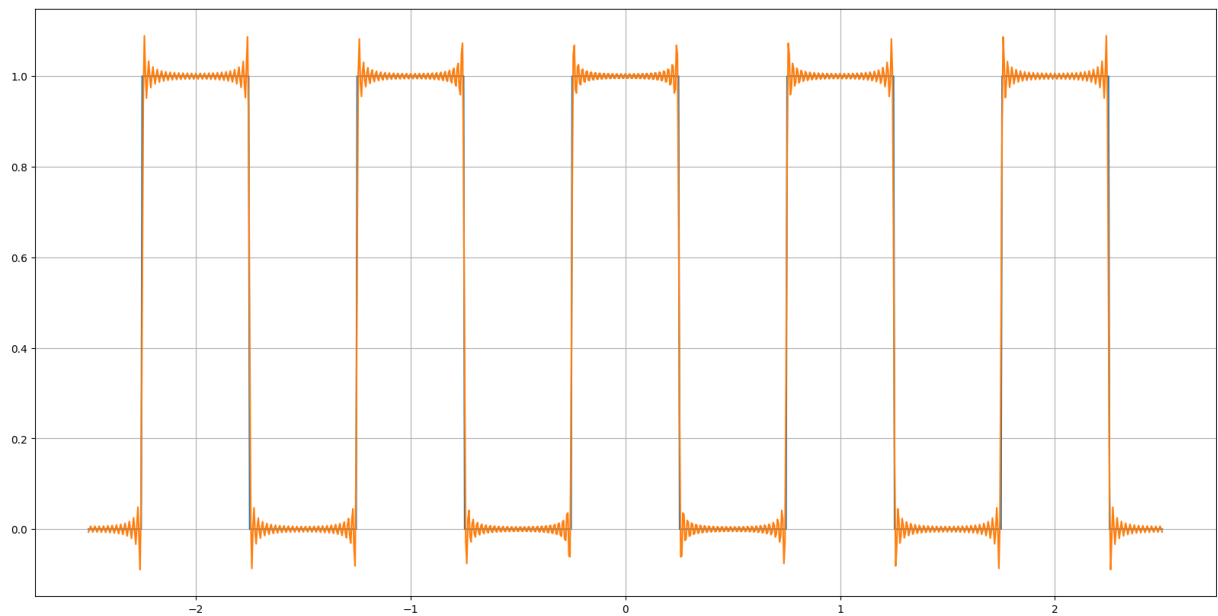
The following Python script plots the original signal, $x(t)$ and the approximated signal, $x_N(t)$ in the same figure for $N = 50$:

```
In [ ]: # New plot for N = 50
x_2 = []
y_2 = []
N = 50

# Filling x and y arrays
for t in time:
    x_2.append(square(t)) #The true square wave function
    y_2.append(fs_approx(t,N)) #The Fourier series approximation

# Plotting
fig,ax=plt.subplots(figsize=(20,10))

# Editing plot parameters
ax.plot(time,x_2)
ax.plot(time,y_2)
ax.grid()
```



The observations. (i.e. for $N = 5$ and $N = 50$)

When N is increased, the number of complex exponential terms in Fourier series expansion approximation of $x(t)$ increases as well which increases the accuracy and gets closer and closer to the original $x(t)$ square wave.

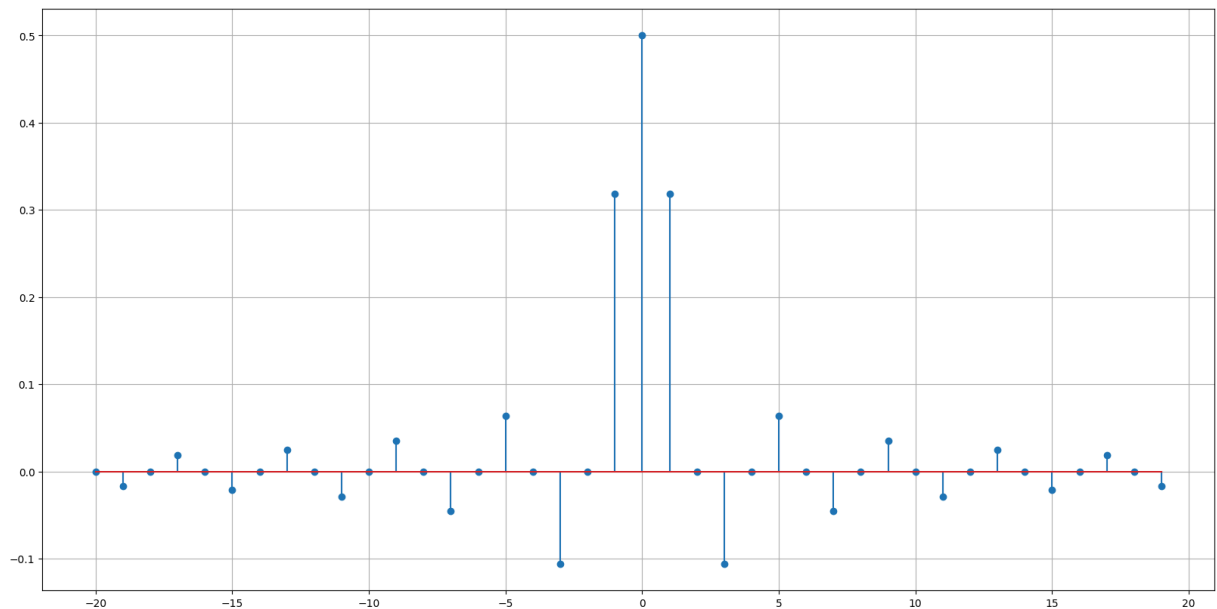
2.2 Fourier Series Coefficients

The following code creates the two arrays k and a_k with integers in the interval $k = -20, \dots, 20$ and plots the Fourier series coefficients of the square wave for each k value in the array, respectively.

```
In [ ]: k = [i for i in range(-20,20,1)]
a_k=[]
for i in k:
    a_k.append(a(i)) #appending the a_k values to the list

fig,ax = plt.subplots(figsize=(20,10))

ax.stem(k,a_k) #plotting the a_k values with respect to k
ax.grid()
```

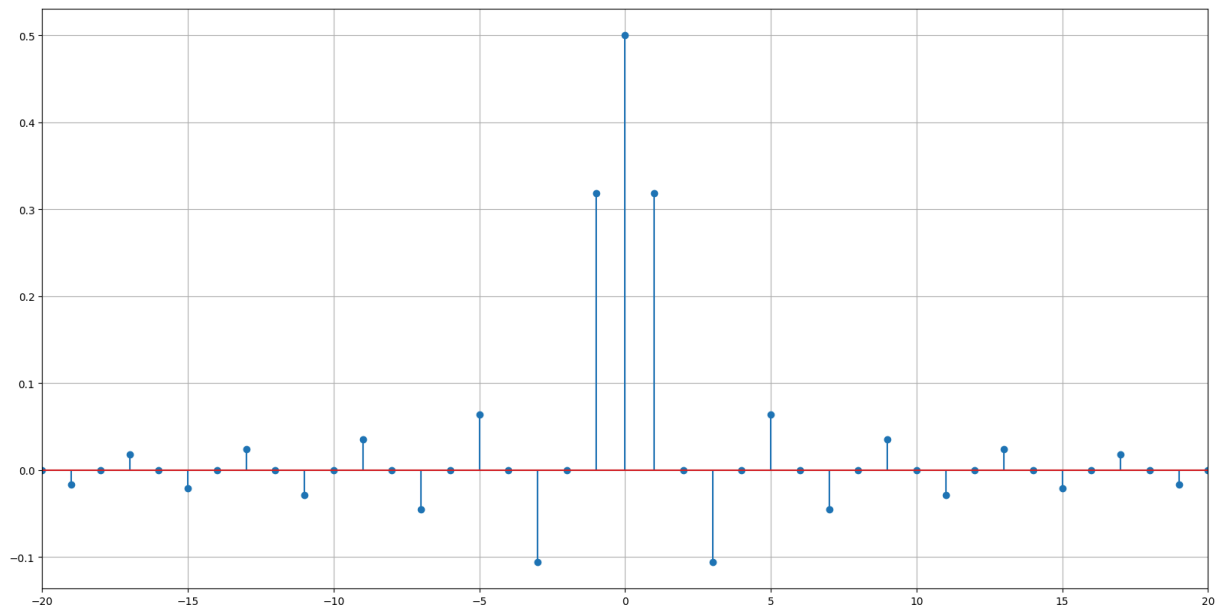


The code provided is modified by using `stem()` function to plot the Fourier series coefficients against k and to plot normalized Fast Fourier Transform (FFT) coefficients in X_{norm} vs k with `stem()` function. (Using `set_xlim()` function to limit the x-axis to the interval $[-20, 20]$)

```
In [ ]: N = 200
t = np.linspace(0, 1-1/N, N)
x = []
for i in t:
    x.append(square(i))
# Obtaining FFT coefficients
X = fftshift(fft(x))
X_norm = X.real/N
k = np.linspace(-N/2, N/2-1, N)

# plotting fft coefficients
# Your code goes here
fig, ax = plt.subplots(figsize=(20, 10))
ax.stem(k, X_norm) #plotting the a_k values generated from fft with respect to k

ax.set_xlim(-20, 20) #setting the x-axis limits
ax.grid()
```



The result from the calculated Fourier coefficients and the result from the FFT algorithm match correctly with each other. The envelop of the Fourier coefficients is seen to be a sinc function. The Fourier coefficients are symmetrical about the y axis. therefore is even.

2.3 Ideal Filters and Actual Filters

The below code generates a signal $x(t)$ consisting of 3 sinusoidal waveforms:

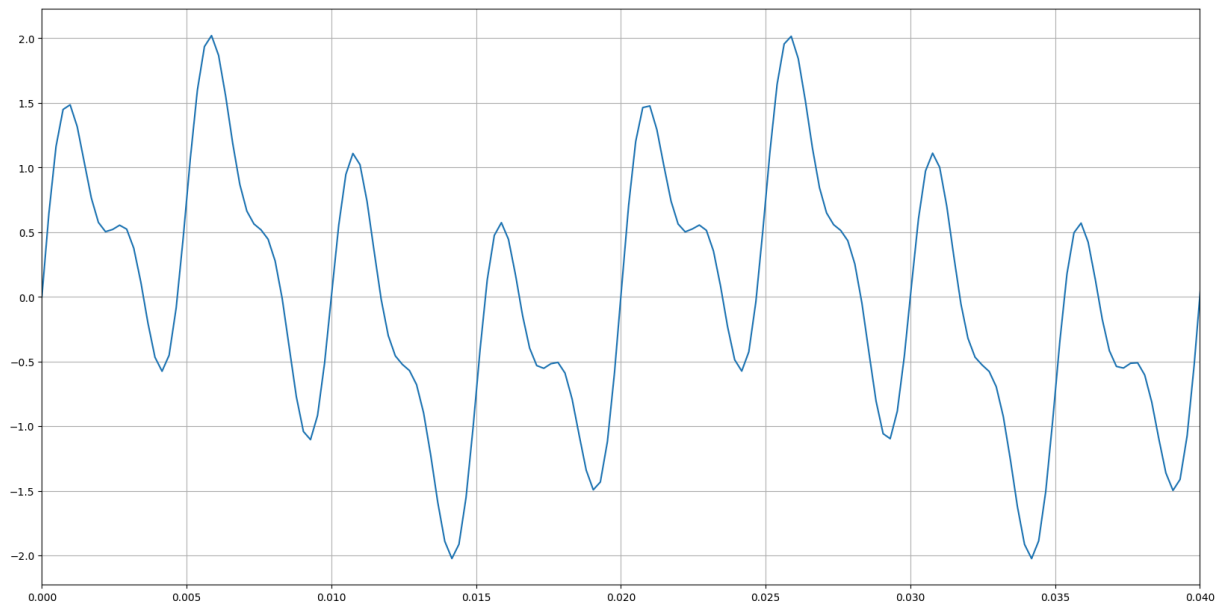
```
In [ ]: # Creating 3 sinusoidal signals
# Your code goes here
w1 = 100 * np.pi
w2 = 400 * np.pi
w3 = 800 * np.pi
a1 = 0.75
a2 = 1
a3 = 0.5

fs = 4095
ws = 2*np.pi*fs
def x(t):
    # Your code goes here
    x_t = a1 * np.sin(w1*t) + a2 * np.sin(w2*t) + a3 * np.sin(w3*t) #creating the s
    return x_t
```

The following code plots the waveform in the time domain (limiting the x axis to the interval [0, 0.04]):

```
In [ ]: time = np.linspace(0,1,fs+1) #creating the time array
xt = [x(t_) for t_ in time] #creating the x(t) array

fig,ax=plt.subplots(figsize=(20,10))
ax.plot(time,xt)
ax.set_xlim(0,0.04) #setting the x-axis limits from 0 to 0.04
ax.grid()
```

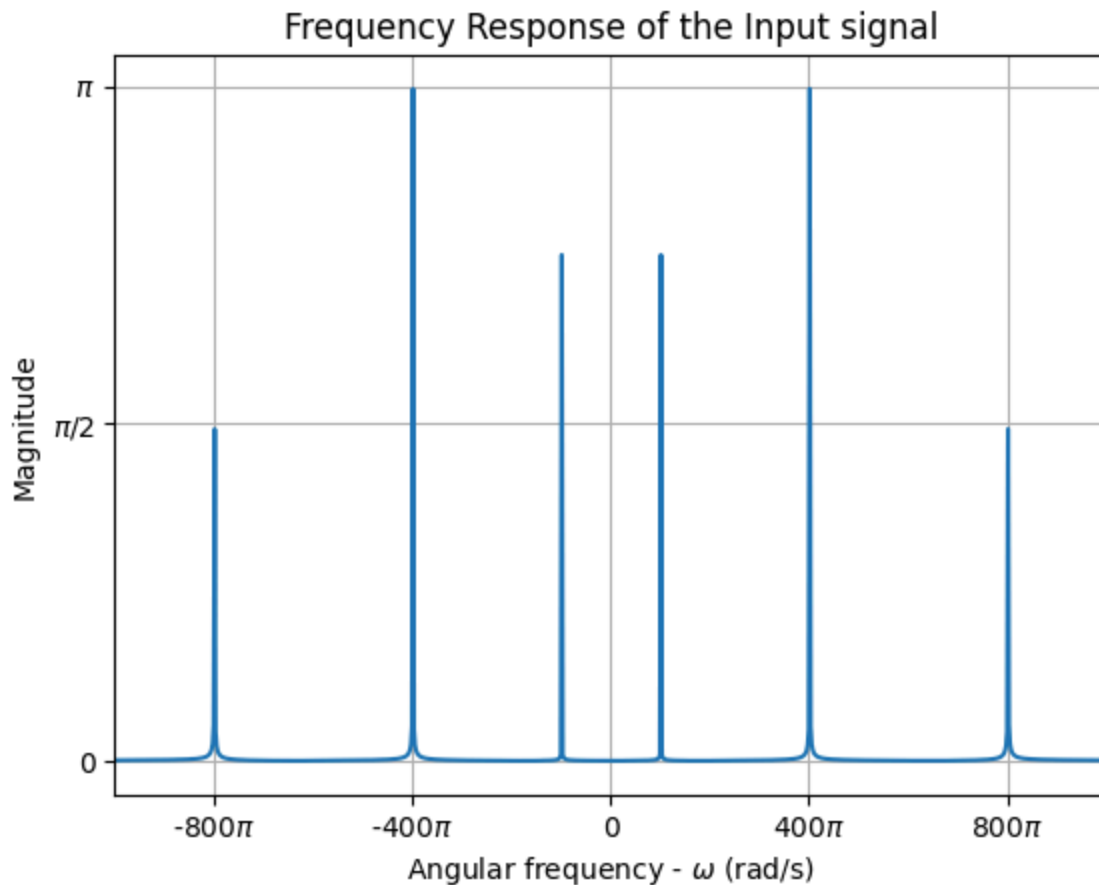


The following code plots the absolute value of the Fourier transform of $x(t)$, that is $X\omega$ against the angular frequency ω .

```
In [ ]: Xw = fft(xt, 4096)*2*np.pi/fs
Xw = fftshift(Xw)
k = np.arange(1,4097)
w = k/4096*ws - ws/2
# Plotting the input signal in frequency domain
fig, ax = plt.subplots()
# Your code goes here

ax.plot(w,abs(Xw)) #plotting the magnitude of the frequency response

ax.set_title('Frequency Response of the Input signal')
ax.set_xlabel('Angular frequency -'+r'$\omega$ (rad/s)')
ax.set_ylabel('Magnitude')
ax.set_xticks(np.arange(-1200*np.pi, 1200*np.pi+1,400*np.pi))
ax.set_xticklabels([str(i)+(r'$\pi$' if i else '') for i in range(-1200,1210,400)])
ax.set_xlim(-1000*np.pi, 1000*np.pi)
ax.set_yticks([0,np.pi/2,np.pi])
ax.set_yticklabels([0,r'$\pi$/2',r'$\pi$'])
plt.grid()
```



The following code snippet contains the function `ideal_filter(w)` which generate a function that acts as an ideal filter :

```
In [ ]: # Ideal filter
wc1 = (w1+w2)/2
wc2 = (w2+w3)/2

# Ideal filter function
def ideal_filter(w):
    # Your code goes here
    if abs(w) < wc1 or abs(w) > wc2:
        gain = 0
    else:
        gain = 1
    return gain
```

Ideal Filter: Part A

The provided code snippet was completed to execute the using of a ideal filter to filter out required signal:

```
In [ ]: k = np.arange(1,4097)
w = k/4096*ws - ws/2

# Your code goes here
```



```

H0w = [ideal_filter(w_) for w_ in w] #creating the frequency response of the ideal

# Simulation of Filtering
Y0w = np.multiply(Xw,H0w)

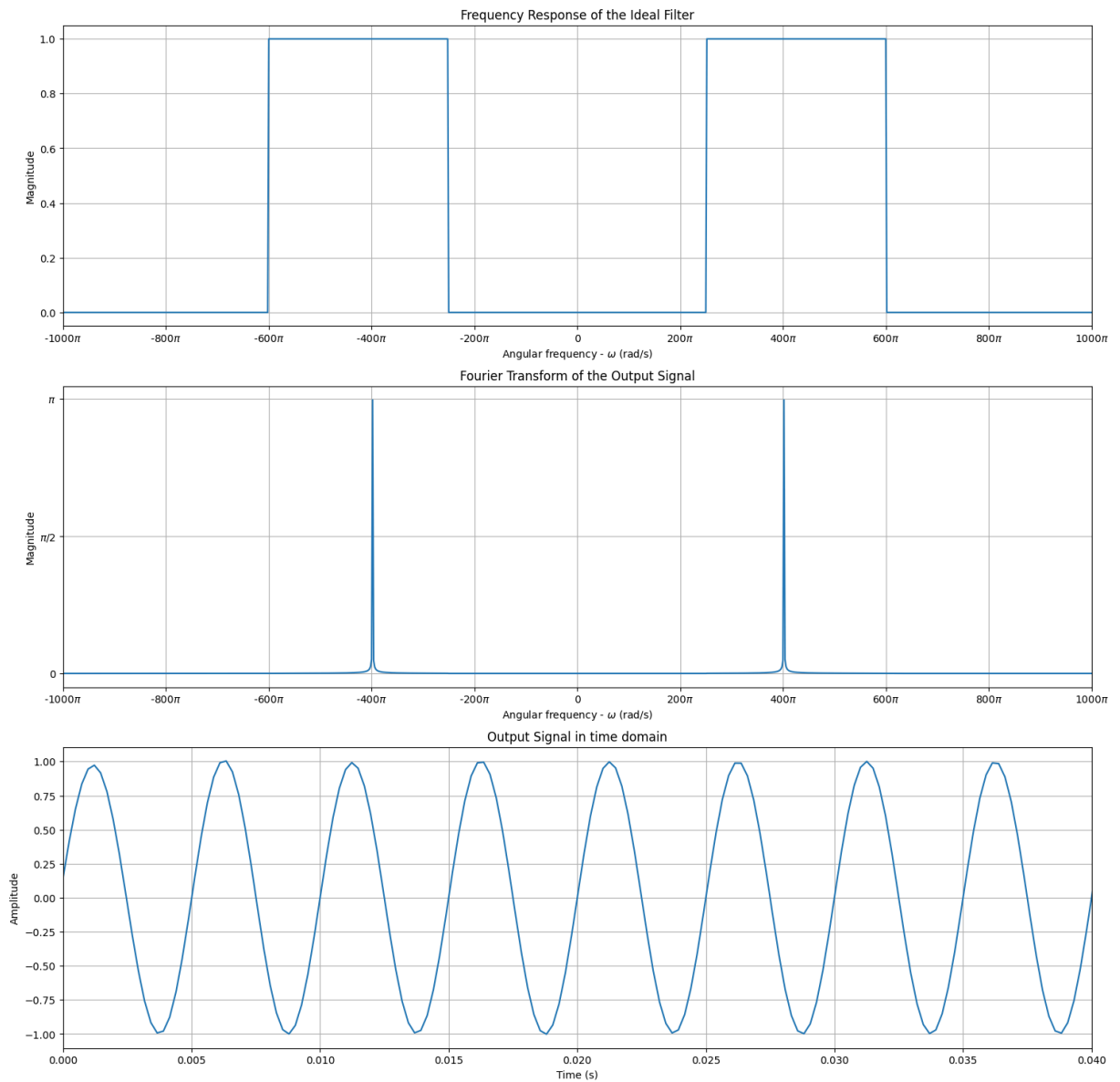
# Obtaining the time domain signal
y0t = ifft(fftshift(Y0w*fs/(2*np.pi)))

# Ideal filter frequency response (magnitude)
fig, axes = plt.subplots(3,1, figsize=(18,18))
axes[0].plot(w,H0w)
axes[0].set_title('Frequency Response of the Ideal Filter')
axes[0].set_xlabel('Angular frequency -'+r' $\omega$ (rad/s)')
axes[0].set_ylabel('Magnitude')
axes[0].set_xticks(np.arange(-1200*np.pi, 1200*np.pi+1,200*np.pi))
axes[0].set_xticklabels([str(i)+(r'$\pi$' if i else '') for i in range(-1200,1210,200)])
axes[0].set_xlim(-1000*np.pi, 1000*np.pi)
axes[0].grid()

# Frequency response of the ideal filter output (magnitude)
axes[1].plot(w,abs(Y0w))
axes[1].set_title('Fourier Transform of the Output Signal')
axes[1].set_xlabel('Angular frequency -'+r' $\omega$ (rad/s)')
axes[1].set_ylabel('Magnitude')
axes[1].set_xticks(np.arange(-1200*np.pi, 1200*np.pi+1,200*np.pi))
axes[1].set_xticklabels([str(i)+(r'$\pi$' if i else '') for i in range(-1200,1210,200)])
axes[1].set_xlim(-1000*np.pi, 1000*np.pi)
axes[1].set_yticks([0,np.pi/2,np.pi])
axes[1].set_yticklabels([0,r'$\pi/2$',r'$\pi$'])
axes[1].grid()

# Output signal in time domain
axes[2].plot(time,np.real(y0t))
axes[2].set_title('Output Signal in time domain')
axes[2].set_xlabel('Time (s)')
axes[2].set_ylabel('Amplitude')
axes[2].set_xlim(0, 0.04)
axes[2].grid()

```



Ideal filter: Part B

The following code was provided to be executed:

```
In [ ]: # Actual Filter
b, a = signal.butter(5, [2*wc1/ws, 2*wc2/ws], 'bandpass', analog=False)
ww, h = signal.freqz(b, a, 2047)
ww = np.append(-np.flipud(ww), ww)*ws/(2*np.pi)
h = np.append(np.flipud(h), h)
# Filtering
y = signal.lfilter(b,a,xt)

# Obtaining the frequency response of the output signal
Y = fft(y,4096)*2*np.pi/fs
Y = fftshift(Y)

# Actual filter frequency response (magnitude)
fig, axes = plt.subplots(3,1, figsize=(18,18))
axes[0].plot(ww, abs(h) )
```

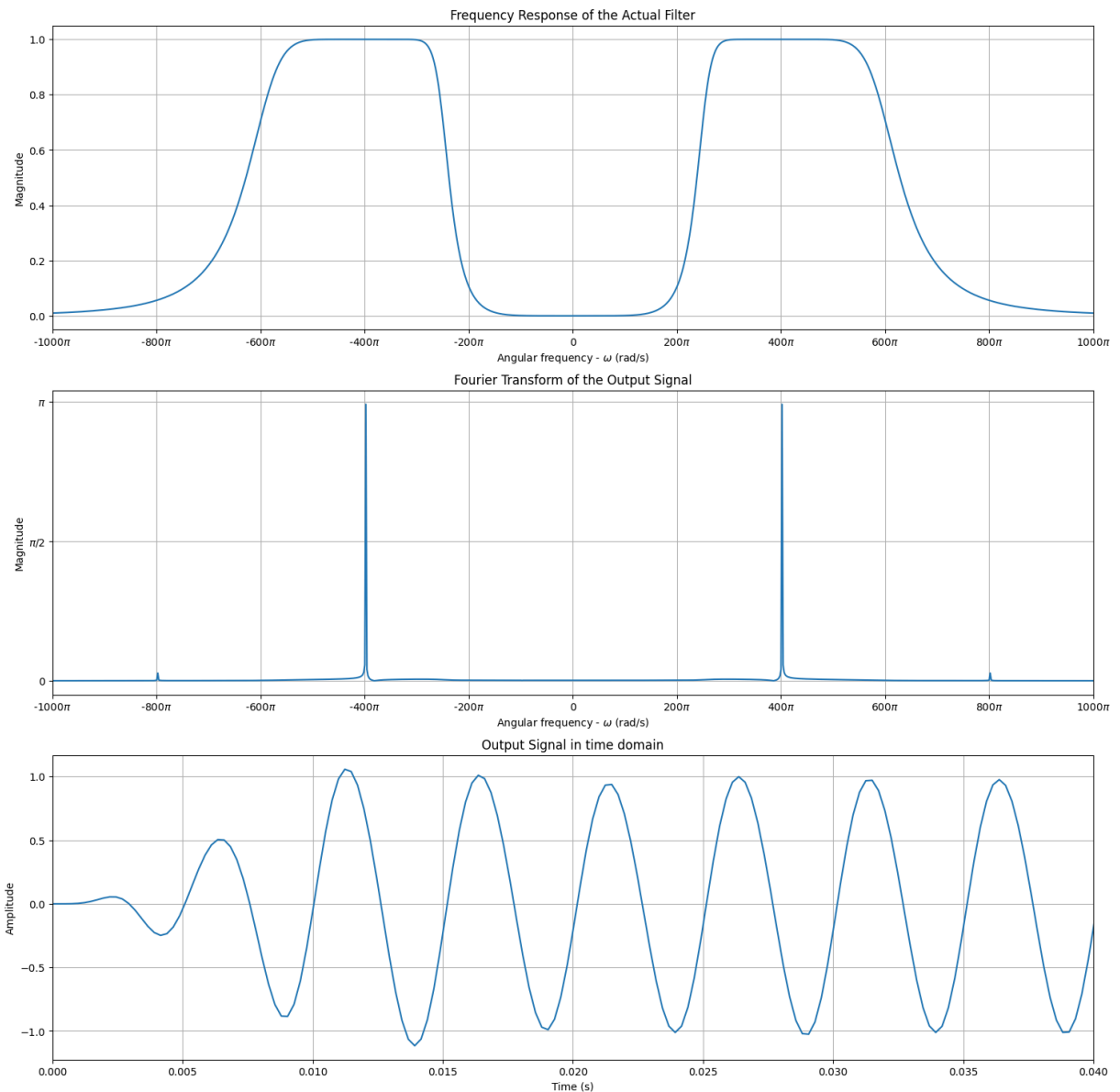
```

axes[0].set_xlabel('Angular frequency -'+r' $\omega$ (rad/s)')
axes[0].set_ylabel('Magnitude')
axes[0].set_title('Frequency Response of the Actual Filter')
axes[0].set_xticks(np.arange(-1200*np.pi, 1200*np.pi+1,200*np.pi))
axes[0].set_xticklabels([str(i)+(r'$\pi$' if i else '') for i in range(-1200,1210,200)])
axes[0].set_xlim(-1000*np.pi, 1000*np.pi)
axes[0].grid()

# Frequency response of the actual filter output (magnitude)
axes[1].plot(w,abs(Y))
axes[1].set_title('Fourier Transform of the Output Signal')
axes[1].set_xlabel('Angular frequency -'+r' $\omega$ (rad/s)')
axes[1].set_ylabel('Magnitude')
axes[1].set_xticks(np.arange(-1200*np.pi, 1200*np.pi+1,200*np.pi))
axes[1].set_xticklabels([str(i)+(r'$\pi$' if i else '') for i in range(-1200,1210,200)])
axes[1].set_xlim(-1000*np.pi, 1000*np.pi)
axes[1].set_yticks([0,np.pi/2,np.pi])
axes[1].set_yticklabels([0,r'$\pi/2$',r'$\pi$'])
axes[1].grid()

# # Output signal in time domain
axes[2].plot(time,np.real(y))
axes[2].set_title('Output Signal in time domain')
axes[2].set_xlabel('Time (s)')
axes[2].set_ylabel('Amplitude')
axes[2].set_xlim(0, 0.04)
axes[2].grid()

```



While the ideal filter perfectly transmits the desired range of 250-400 rad/s, the actual filter exhibits non-ideality. This manifests as a transition band around 500-800 rad/s, where significant leakage of unwanted frequencies above 600 rad/s occurs. Consequently, the filtered signal deviates from a pure sinusoid, becoming a combination of the intended 400 rad/s component and a spurious 800 rad/s component.

2.4 Removing Power Line noise in an ecg signal

The following code snippet reads the data from file `ecg_signal.csv` and completes a list named `ecg`:

```
In [ ]: # Reading the ECG data
ecg = []
# EDIT HERE
with open('ecg_signal.csv', 'r') as file: #reading the file
    for line in file:
        ecg.append(float(line)) #appending the values to the list
```

```

duration = 10 # seconds
T = duration/len(ecg)
Fs = 1/T

# Obtaining the fourier transform
F = fftshift(fft(ecg))
fr = np.linspace(-Fs/2, Fs/2, len(F))

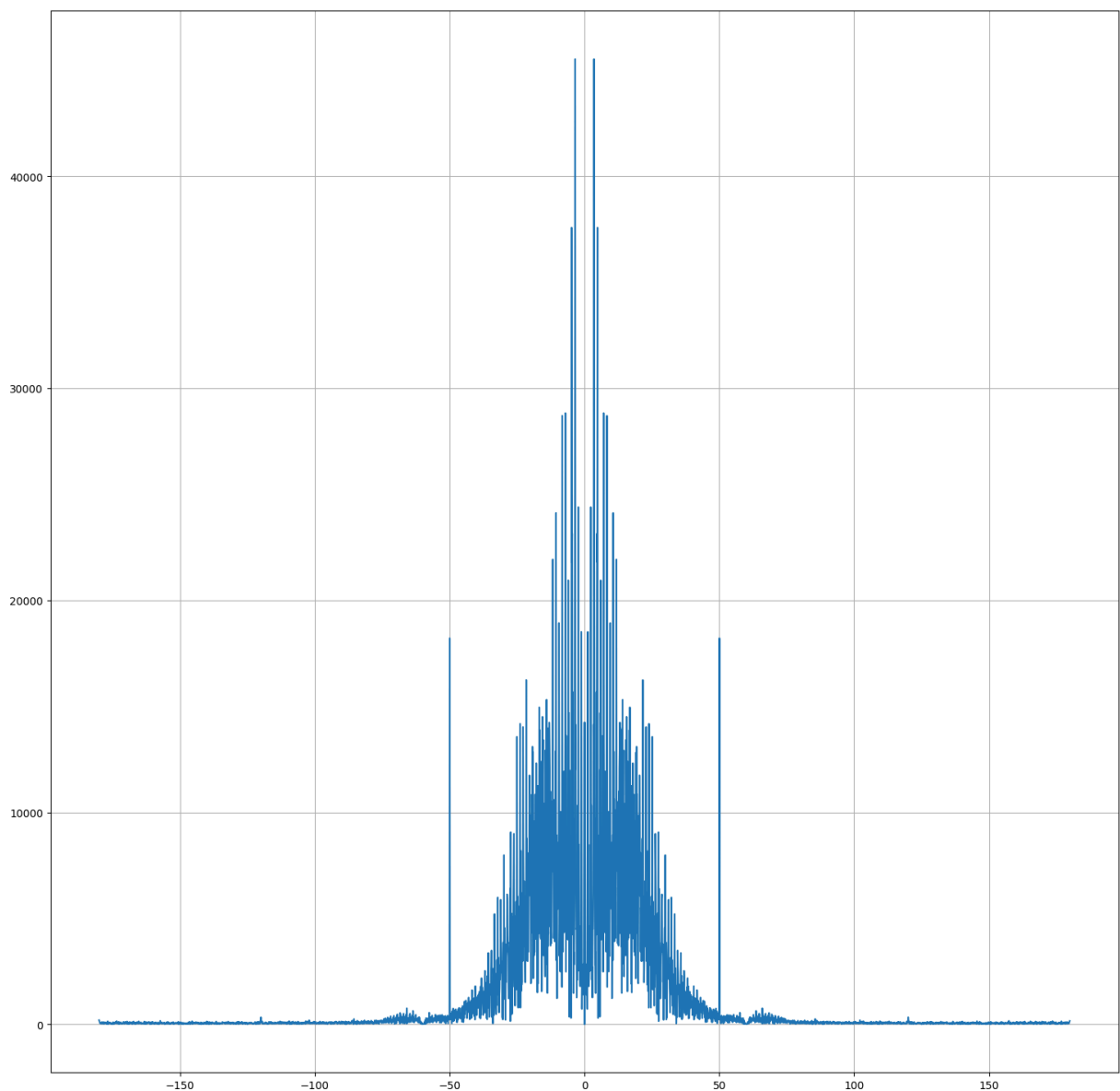
```

the following code snippet plots the absolute values of the fourier transform with respect to frequency while limiting the x axis to [-100, 100]

```

In [ ]: fig, ax = plt.subplots(figsize=(18,18))
        ax.plot(fr,abs(F))
        ax.grid()

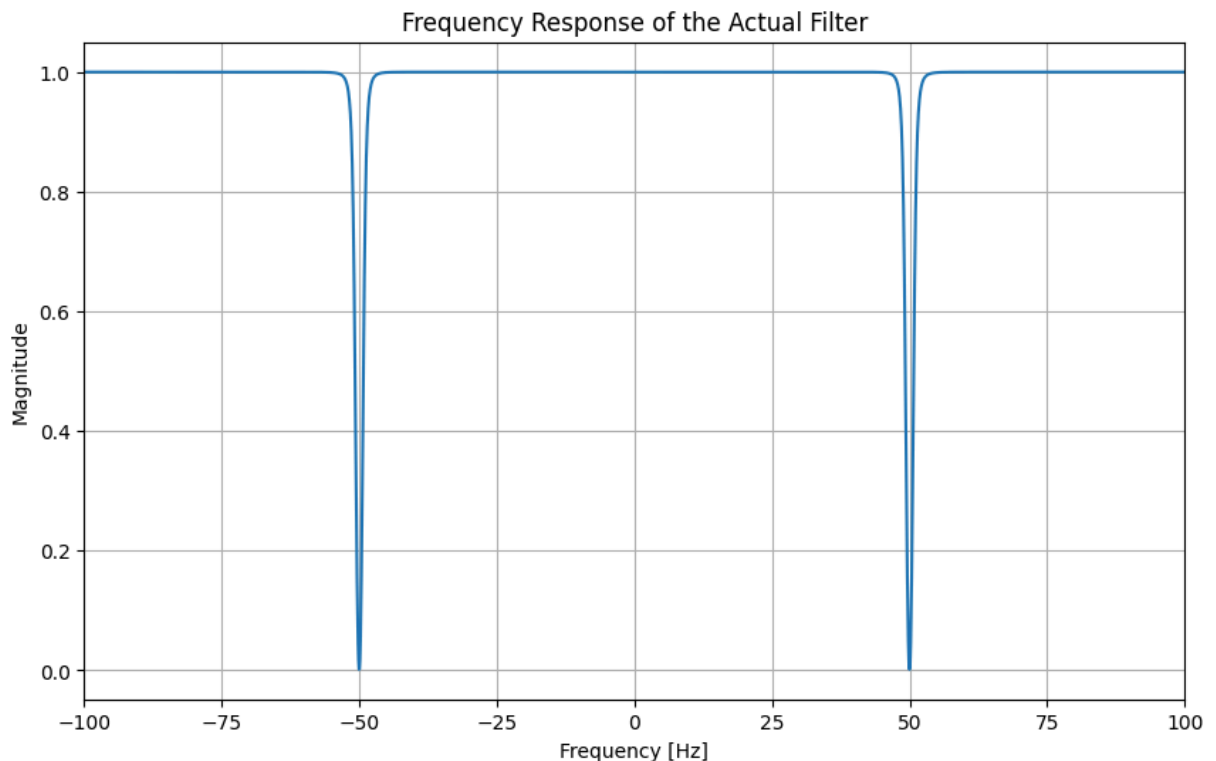
```



A band stop filter has to be used in order to remove the noise at 50 Hz

The following code provides the plot of the band stop filter in the frequency domain:

```
In [ ]: # Designing the filter
f1 = 49
f2 = 51
filter_type = 'bandstop' # EDIT HERE
b, a = signal.butter(2, [2*f1/Fs, 2*f2/Fs], filter_type, analog=False)
# Obtaining the frequency response of the filter
ww, h = signal.freqz(b, a, 2047)
ww = np.append(-np.flipud(ww), ww)
h = np.append(np.flipud(h), h)
# Plotting the frequency response
fig, ax = plt.subplots(figsize=(10,6))
ax.plot(ww*Fs/(2*np.pi), abs(h))
ax.set_title('Frequency Response of the Actual Filter')
ax.set_xlabel('Frequency [Hz]')
ax.set_ylabel('Magnitude')
ax.set_xlim(-100,100)
ax.grid()
```



The following code snippet provides the time domain representation of the input signal and the output signal after going through the stop band filter:

```
In [ ]: time = np.arange(T, duration+T, T)
# Filtering the ECG wavefoem
output = signal.lfilter(b, a, ecg)

fig, ax = plt.subplots(2,1,figsize=(20,10))

# Plotting the input signal
```

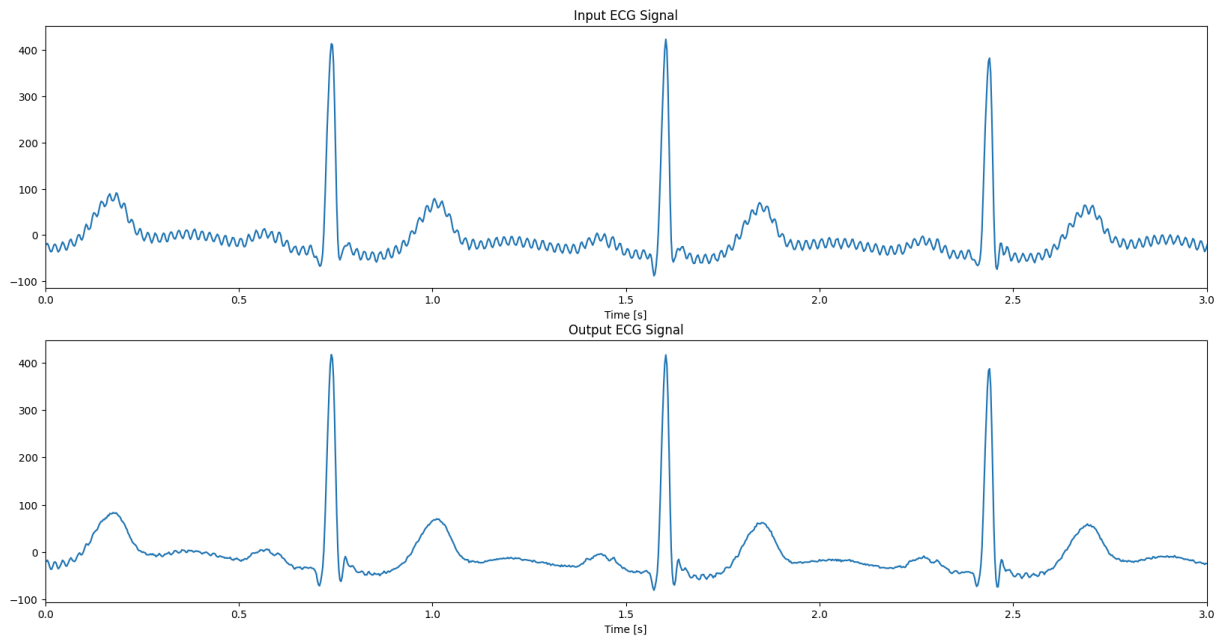
```

ax[0].plot(time,ecg)
ax[0].set_title('Input ECG Signal')
ax[0].set_xlabel('Time [s]')
ax[0].set_xlim(0, 3)

# Plotting the output signal
ax[1].plot(time,output)
ax[1].set_title('Output ECG Signal')
ax[1].set_xlabel('Time [s]')
ax[1].set_xlim(0, 3)

```

Out[]: (0.0, 3.0)



The following code snippet provided the plotting between the absolute value of Fourier transform of the output waveform with respect to the frequency (limiting the x axis to the interval $[-100, 100]$)

```

In [ ]: F = fftshift(fft(output))

# Plotting the frequency response of the output signal
fig, ax = plt.subplots(figsize=(12,12))
ax.plot(fr,abs(F))
ax.set_xlim(-100,100)
ax.set_title('Frequency Response of the Output Signal')

```

Out[]: Text(0.5, 1.0, 'Frequency Response of the Output Signal')

