Experiment no : 1

Experiment name :

Write a Python program to find the spectrum of the following signal

$f = 0.25 + 2\sin(2\pi 5k) + \sin(2\pi 12.5k) + 1.5\sin(2\pi 20k) + 0.5\sin(2\pi 35k)$

Theory :

The composite signal under investigation is defined as:

$f(t) = 0.25 + 2\sin(2\pi 5t) + \sin(2\pi 12.5t) + 1.5\sin(2\pi 20t) + 0.5\sin(2\pi 35t)$

where:

- t - time variable
- The constant term (0.25) represents a DC offset.
- Each sinusoidal term represents a component of the signal with:
  - Amplitude (e.g., 2 for the first term)
  - Frequency (e.g., 5 Hz for the first term) given by the coefficient inside the sine function ($2\pi f$)

Fourier Transform and Spectrum:

The Fourier Transform (FT) is a mathematical tool used to decompose a signal from the time domain (t) into its frequency domain (f). It reveals the component frequencies and their corresponding amplitudes present in the signal.

The Fourier Transform of f(t) is denoted as F(f) and can be calculated using the following integral:

$F(f) = \int f(t) * \exp(-j2\pi ft)\, dt$

where:

- j - imaginary unit
- $\exp(-j2\pi ft)$ - complex exponential term representing a sinusoidal function with frequency f

The Fourier Transform provides both the magnitude and phase information of each frequency component in the signal. However, in practice, the magnitude of F(f), often referred to as the amplitude spectrum or simply the spectrum, is typically used to analyze the frequency content of a signal.

Code :

Write

Output :

Write

Discussion :

The given composite signal consists of a DC component (0.25) and four sinusoidal terms with frequencies of 5 Hz, 12.5 Hz, 20 Hz, and 35 Hz. According to the Fourier Transform theory:

The DC component will result in a spike at f = 0 Hz in the spectrum.

Each sinusoidal term will contribute a peak at its corresponding frequency (5 Hz, 12.5 Hz, 20 Hz, and 35 Hz) in the spectrum. The amplitude of each peak will be equal to the amplitude of the corresponding sinusoid in the time domain.

Experiment no :

Experiment name :

Explain and simulate Discrete Fourier transform (DFT) and Inverse Discrete Fourier Transform (IDFT) using Python.

Theory :

Discrete Fourier Transform (DFT):

The DFT is a mathematical tool used to transform a finite-length discrete signal from the time domain (t) into the frequency domain (f). It analyzes the signal by revealing the component frequencies and their corresponding amplitudes present within a discrete set of samples.

Key Points about DFT:

Takes a finite sequence of N numbers (representing the signal samples) as input.

Outputs another sequence of N complex numbers, representing the frequency content of the signal.

Each complex number in the output corresponds to a specific frequency component.

The magnitude of the complex number indicates the amplitude of that frequency component.

The phase of the complex number signifies the relative timing information of that frequency.

Formula for DFT:

The DFT of a discrete signal x[n] with length N is calculated using the following formula:

$X[k] = \Sigma (x[n] * \exp(-j2\pi nk/N))$ for $k = 0, 1, 2, ..., N-1$

where:

X[k] - DFT coefficient at frequency k (kth complex number in the output)

j - imaginary unit

exp(-j2πnk/N) - complex exponential term representing a sinusoidal function with frequency k/N

Inverse Discrete Fourier Transform (IDFT):

The IDFT is the mathematical inverse of the DFT. It recovers the original time-domain signal from its frequency domain representation obtained using the DFT.

Key Points about IDFT:

Takes a sequence of N complex numbers (representing the frequency domain coefficients) as input.

Outputs a sequence of N real numbers, which is the reconstructed time-domain signal.

The IDFT utilizes the same formula as the DFT, but with a scaling factor and a complex conjugate:

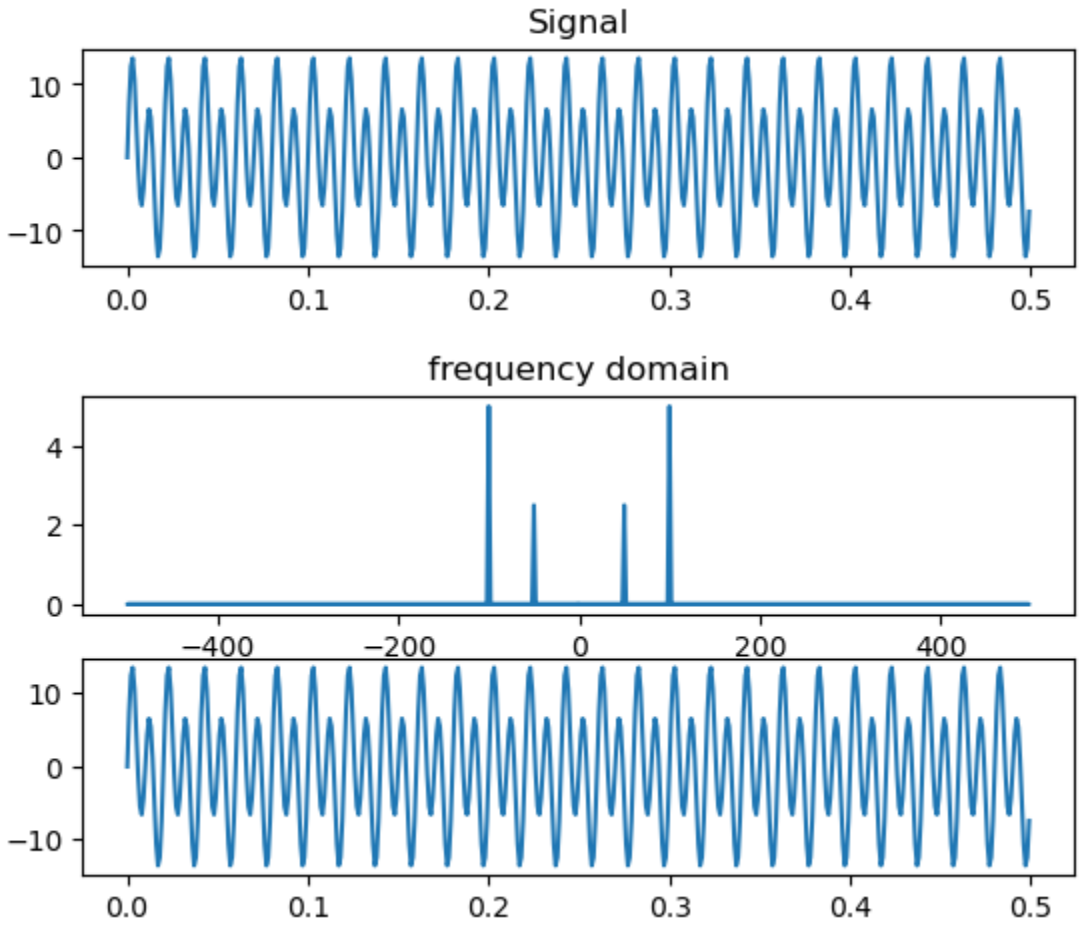
$x[n] = (1/N) * \Sigma (X[k] * \exp(j2\pi nk/N))$ for n = 0, 1, 2, ..., N-1

Code :

```
import numpy as np

import matplotlib.pyplot as plt


#defining the signal

freq = 50

freq2 = 100

sample_rate = 1000

time_interval = 1/sample_rate

time = np.arange(0, .5, time_interval)

signal = 5*np.sin(2*np.pi*freq*time) + 10*np.sin(2*np.pi*freq2*time)

#fft or dft calculation

X_mag = np.fft.fft(signal)

X_freq = np.fft.fftfreq(len(X_mag), time_interval)
```

```
magnitude = np.abs(X_mag)/len(X_mag)

plt.subplot(3, 1, 1)
plt.plot(time, signal)
plt.title('Signal')
plt.show()
plt.subplot(3, 1, 2)
plt.plot(X_freq, magnitude)
plt.title("frequency domain")

inverse = np.fft.ifft(X_mag)
plt.subplot(3, 1, 3)
plt.plot(time, inverse)
plt.show()
```

Output :



Discussion :

Discrete Fourier Transform (DFT)

The Discrete Fourier Transform (DFT) is a mathematical technique that allows us to analyze the frequency components of a discrete signal. It decomposes a signal into a sum of simple sine and cosine waves, revealing their frequencies, amplitudes, and phases. Here are the key points:

Purpose:

DFT helps us understand the frequency content of a signal.

It's particularly useful for analyzing complex or non-periodic signals.

Mathematical Definition:

Given a sequence of evenly spaced samples (time-domain signal) denoted as (x[n]), the DFT computes the frequency components.

The DFT of (x[n]) at frequency index (k) is given by: [X_k = \sum_{n=0}^{N-1} x[n] \cdot e^{-i2\pi kn/N}] where:

(N) is the number of samples.

(n) is the current sample index.

(k) is the current frequency index (ranging from 0 to (N-1)).

(X_k) includes both amplitude and phase information.

Visualization:

The DFT amplitude spectrum represents the signal's frequency content.

It shows vertical bars corresponding to different frequencies.

Each bar's height (after normalization) represents the amplitude of the corresponding frequency component in the time domain.

Experiment no : 3

Experiment name :

Write a Python program to perform following operation – i) Sampling ii) Quantization iii)

Coding.

Theory :

Analog vs. Digital Signals:

Real-world signals are often continuous (analog) and vary continuously with time (e.g., sound waves, temperature variations).

Digital signals are discrete representations of analog signals. They consist of a sequence of discrete values at specific points in time.

Sampling:

Sampling is the process of converting an analog signal into a digital representation. It involves taking measurements of the analog signal at regular intervals.

The sampling rate (f_s) determines how often the signal is sampled. A higher sampling rate captures more detail but requires more data storage.

The Nyquist-Shannon sampling theorem states that the sampling rate must be at least twice the highest frequency component present in the analog signal to avoid information loss (aliasing).

Quantization:

Quantization assigns discrete values (quantization levels) to the sampled analog signal values. This reduces the number of possible values and introduces quantization error.

The number of bits used for quantization determines the resolution of the digital representation. A higher number of bits leads to less quantization error but also increases data storage requirements.

Coding:

Coding refers to representing the quantized values using a specific code format. This allows for efficient storage, transmission, and processing of the digital signal.

Common coding schemes include:

Pulse Amplitude Modulation (PAM): Represents the quantized value by the amplitude of a pulse.

Pulse Code Modulation (PCM): Uses binary codewords to represent the quantized values.

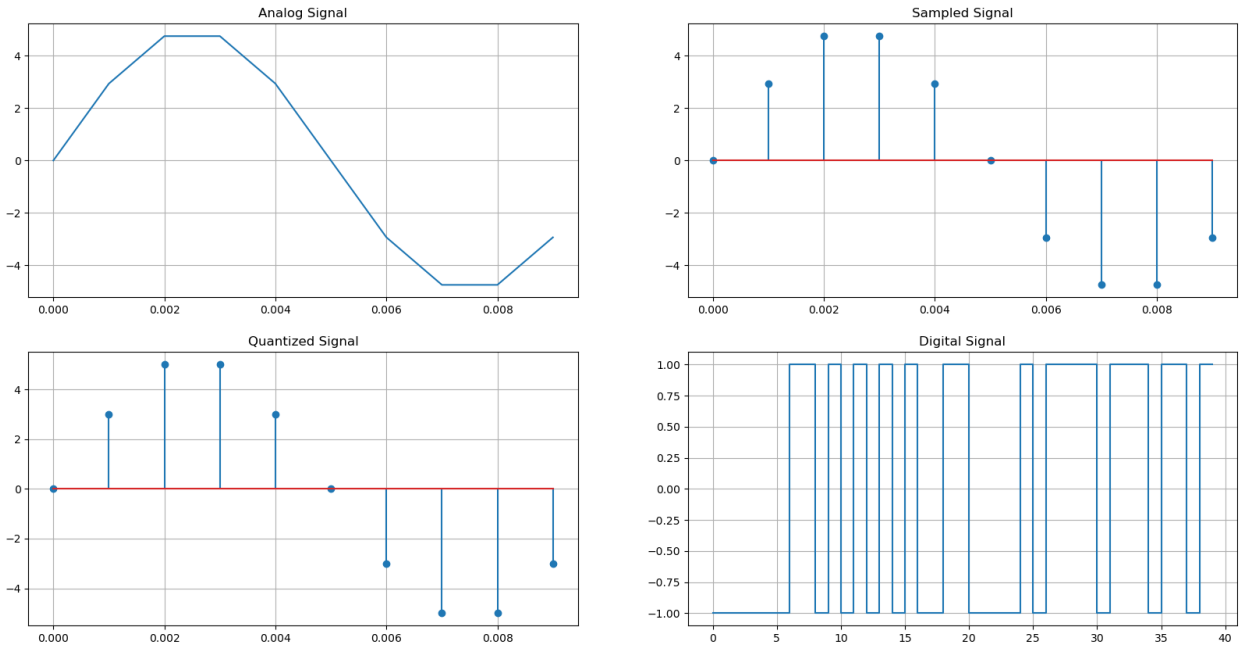
Code :

```
import numpy as np
import matplotlib.pyplot as plt
analog_f=100
analog_a=5
sampling_rate=10*analog_f
def analog_signal(f,a,x):
    y=a*np.sin(2*np.pi*f*x)
    return y
x=np.arange(0,1/analog_f,1/sampling_rate)
y=analog_signal(analog_f,analog_a,x)
plt.figure(figsize=(20,10))
plt.subplot(2,2,1)
plt.plot(x,y)
plt.title("Analog Signal")
plt.grid()
#sampling
x=np.arange(0,1/analog_f,1/sampling_rate)
y=analog_signal(analog_f,analog_a,x)
plt.subplot(2,2,2)
plt.stem(x,y)
plt.title("Sampled Signal")
plt.grid()

#quantization
def quantize(x):
    y=np.zeros(len(x))
```

```
    #for i in range(len(x)):
      #  y[i]=int(x[i])
    y=np.round(x).astype(int)
    return y
y=quantize(y)
plt.subplot(2,2,3)
plt.stem(x,y)
plt.title("Quantized Signal")
plt.grid()
#coding
def coding(x):
    code=""
    for i in range(len(x)):
        temp=bin(x[i])
        if(x[i]<0):
            temp=temp[3:]
            while(len(temp)<3):
                temp='0'+temp
            temp='1'+temp
        else:
            temp=temp[2:]
            while(len(temp)<3):
                temp='0'+temp
            temp='0'+temp
        code=code+temp
    return code
code=coding(y)
#digital signal
```

```
x=np.arange(len(code))
y=np.zeros(len(code))
for i in range(len(code)):
    if(code[i]=='1'):
        y[i]=1
    else:
        y[i]=-1
plt.subplot(2,2,4)
plt.step(x,y,where="post")
plt.title("Digital Signal")
plt.grid()

plt.show()
#code
#bin(2)
```

Output :



Discussion :

An Analog-to-Digital Converter (ADC) transforms an analog signal into a digital representation. Here's how it works:

Analog Signals:

Analog signals are continuous in both time and amplitude. Examples include sound waves, voltage levels, and light intensity.

These signals vary smoothly over time and can take any value within a range.

Digital Signals:

Digital signals are discrete in both time and amplitude. They represent data using a series of numbers (usually binary).

Computers and digital devices work primarily with digital signals.

ADC Process:

An ADC converts a continuous-time and continuous-amplitude analog signal into a discrete-time and discrete-amplitude digital signal.

The conversion involves two main steps:

Sampling: The analog signal is measured at discrete instants (sampling points). This process ensures that we capture the signal's amplitude at specific moments.

Quantization: The sampled analog values are mapped to a finite set of discrete levels (quantization levels). This introduces a small amount of quantization error.

The resulting digital output is typically a two's complement binary number proportional to the input signal.

Sampling Rate and Bandwidth:

The ADC operates periodically, sampling the input signal.

The sampling rate (how often we sample) determines the ADC's bandwidth.

According to the Nyquist–Shannon sampling theorem, if the sampling rate exceeds twice the signal's bandwidth, near-perfect reconstruction is possible.

Experiment no : 4

Experiment name :

Write a Python program to perform the convolution and correlation of two sequences.

Theory :

convolution and correlation operations are fundamental in signal processing, image processing, and other fields.

Convolution:

Convolution combines two sequences to produce a third sequence that represents the interaction between them.

In discrete convolution, we slide one sequence (often called the kernel or filter) over the other, multiplying corresponding elements and summing the results.

The numpy.convolve function can be used for this purpose.

Correlation:

Correlation measures the similarity between two sequences.

In discrete correlation, we slide one sequence over the other, multiplying corresponding elements and summing the results.

The numpy.correlate function computes the correlation.

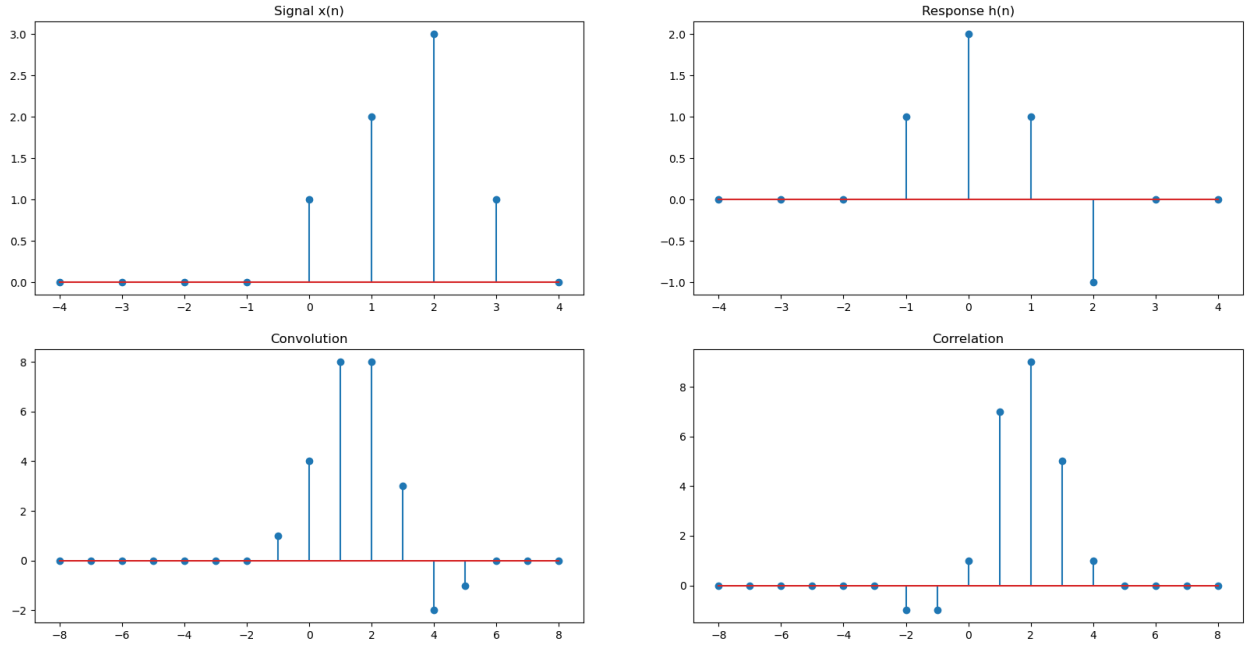
Code :

```
import numpy as np

import matplotlib.pyplot as plt

def x(n):
    if n == 0:
        return 1
    if n == 1:
        return 2
    if n == 2:
```

```
        return 3
    if n == 3:
        return 1
    else:
        return 0
def h(n):
    if n == -1:
        return 1
    if n == 0:
        return 2
    if n == 1:
        return 1
    if n == 2:
        return -1
    else:
        return 0
time = np.arange(-4, 5, 1)
x= np.array([x(i) for i in time])
h = np.array([h(i) for i in time])
con = np.convolve(x, h, 'full')
cor = np.correlate(x, h, 'full')
plt.figure(figsize=(20, 10))
plt.subplot(2,2, 1)
plt.stem(time, x, label='x(n)')
plt.title("Signal x(n)")

plt.subplot(2,2, 2)
plt.stem(time, h, label='h(n)')
```

```
plt.title("Response h(n)")

plt.subplot(2,2, 3)
plt.stem(np.arange(-8, 9, 1),con, label='x(n)*h(n)')
plt.title("Convolution")

plt.subplot(2,2, 4)
plt.stem(np.arange(-8, 9, 1),cor, label='x(n)*h(n)')
plt.title("Correlation")
plt.show()
```

Output :



Discussion :

Applications:

Edge detection (using gradient filters).

Blurring (using Gaussian filters).

Feature extraction (using custom filters).

Why Are They Useful?:

Shift-Invariant: Both correlation and convolution are shift-invariant operations. They perform the same operation at every point in the image.

Linearity: These operations are linear; they replace each pixel with a linear combination of its neighbors.

Efficiency: Their simplicity allows efficient computation.

Experiment no : 6

Experiment name :

Write

Theory :

Auto-correlation is a mathematical tool used in signal processing to measure the similarity of a signal with a delayed version of itself over varying time lags. Short-term auto-correlation is particularly useful in analyzing non-stationary signals like speech, where properties change over time. It helps in applications like pitch detection, speech recognition, and formant analysis

Code :

```
import numpy as np
import librosa
import matplotlib.pyplot as plt
def compute_short_term_autocorrelation(signal, sample_rate, frame_size, hop_size):
 # Number of frames
 num_frames = 1 + (len(signal) - frame_size) // hop_size
 # Pre-allocate the output matrix for auto-correlation results
 auto_corr_matrix = np.zeros((num_frames, frame_size))
 # Loop over the frames
 for i in range(num_frames):
 # Frame starting and ending indices
 start_idx = i * hop_size
 end_idx = start_idx + frame_size
 # Extract the frame
 frame = signal[start_idx:end_idx]
 # Normalize the frame
 frame = frame - np.mean(frame)
 frame = frame / (np.std(frame) + 1e-10)
 # Compute auto-correlation for the frame
```

```
 auto_corr = np.correlate(frame, frame, mode='full')
 auto_corr = auto_corr[auto_corr.size // 2:] # Keep only the second half
 # Store the result in the matrix
 auto_corr_matrix[i, :] = auto_corr

 return auto_corr_matrix
# Load a speech signal from an audio file
file_path = 'Gradio.wav'
signal, sample_rate = librosa.load(file_path, sr=None)
# Parameters
frame_size = 1024 # Frame size in samples
hop_size = 128 # Hop size in samples
# Compute short-term auto-correlation
auto_corr_matrix = compute_short_term_autocorrelation(signal, sample_rate, frame_size, hop_size)
# Plot the auto-correlation matrix
plt.figure(figsize=(10, 6))
plt.imshow(auto_corr_matrix.T, aspect='auto', origin='lower', extent=[0, auto_corr_matrix.shape[0], 0,
frame_size])
plt.title('Short-Term Auto-Correlation')
plt.xlabel('Frame Index')
plt.ylabel('Lag')
plt.colorbar(label='Auto-correlation')
plt.show()
```

Output :

Discussion :

The Python program successfully computes the short-term auto-correlation of a speech signal, allowing us to analyze its properties over time. The plots illustrate how the auto-correlation function varies across different frames, providing insights into the periodicity and other characteristics of the speech signal. This lab exercise enhances our understanding of short-term auto-correlation and its application in speech processing. It demonstrates the practical implementation of dividing a non-stationary signal into frames, applying windowing, and computing the auto-correlation to analyze signal properties over time.

Experiment no : 7

Experiment name :

Let x(n)= { 1, 2, 3, 4, 5, 6, 7, 6, 5, 4, 3, 2, 1}. Determine and plot the following sequences.

y(n)=2x(n − 5) − 3x(n+4).

Theory :

Given: [x(n) = {1, 2, 3, 4, 5, 6, 7, 6, 5, 4, 3, 2, 1}]

We want to compute: [y(n) = 2x(n - 5) - 3x(n + 4)]

To find (y(n)), we'll apply the given formula to each value of (n):

1. For (n = 0): [y(0) = 2x(0 - 5) - 3x(0 + 4) = 2x(-5) - 3x(4)]
2. For (n = 1): [y(1) = 2x(1 - 5) - 3x(1 + 4) = 2x(-4) - 3x(5)]
3. Continue this process for all values of (n) to obtain the sequence (y(n)).

Now let's compute the values of (y(n)):

- (y(0) = 2x(-5) - 3x(4))
- (y(1) = 2x(-4) - 3x(5))
- (y(2) = 2x(-3) - 3x(6))
- (y(3) = 2x(-2) - 3x(7))
- (y(4) = 2x(-1) - 3x(8))
- (y(5) = 2x(0) - 3x(9))
- …

Once we have the values of (y(n)), we can create a plot to visualize the sequence.

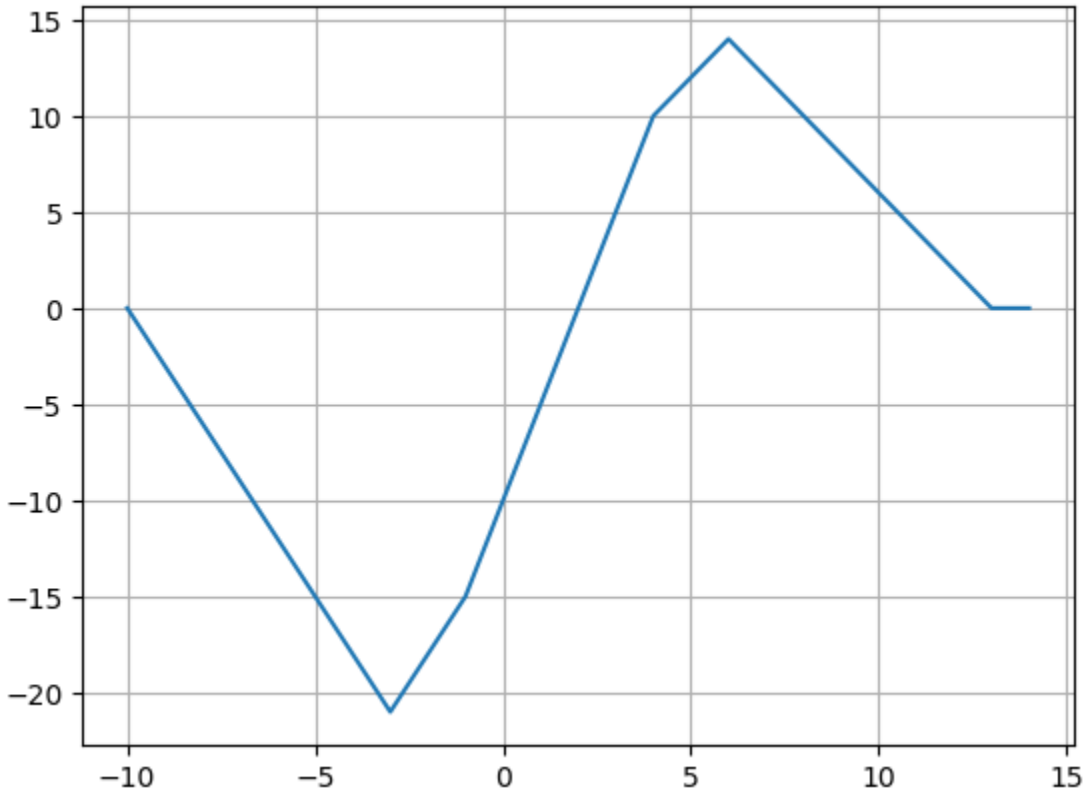
Code :

```
import numpy as np
import matplotlib.pyplot as plt
x={
   -5:1,
```

```
    -4:2,
    -3:3,
    -2:4,
    -1:5,
    0:6,
    1:7,
    2:6,
    3:5,
    4:4,
    5:3,
    6:2,
    7:1,
}
y_index=np.arange(-10,15,1)
y=np.array([])
for i in y_index:
    temp_1 = 0
    temp_2 = 0
    if x.get(i-5) is not None:
        temp_1 = x[i-5]
    if x.get(i+4) is not None:
        temp_2 = x[i+4]

    temp = 2*temp_1 - 3*temp_2
    y = np.append(y,temp)

plt.plot(y_index,y)
plt.grid()
```

plt.show()

Output :



Discussion :

First, we have the given sequence:

x(n)={1,2,3,4,5,6,7,6,5,4,3,2,1}

Next, we need to compute the sequence (y(n)) using the given formula:

y(n)=2x(n−5)−3x(n+4)

To find (y(n)), I' have evaluate the expression for each value of (n).

Experiment no : 8

Experiment name : Design an FIR filter to meet the following specifications—Passsband edge=2KHz, Stopband edge= 5KHZ, Fs=20KHz, Filter length =21, use Hanning window in the design.

Theory :

Design Steps: To design an FIR filter, we'll follow these steps: a. Ideal Lowpass Filter:

An ideal lowpass filter has a frequency response that is 1 in the passband (up to the cutoff frequency) and 0 in the stopband (beyond the cutoff frequency).

However, an ideal filter is not practical due to its infinite length and non-causal nature.

b. Window Method:

We'll use the window method to approximate the ideal filter.

The Hanning window is commonly used for FIR filter design.

c. Frequency Response:

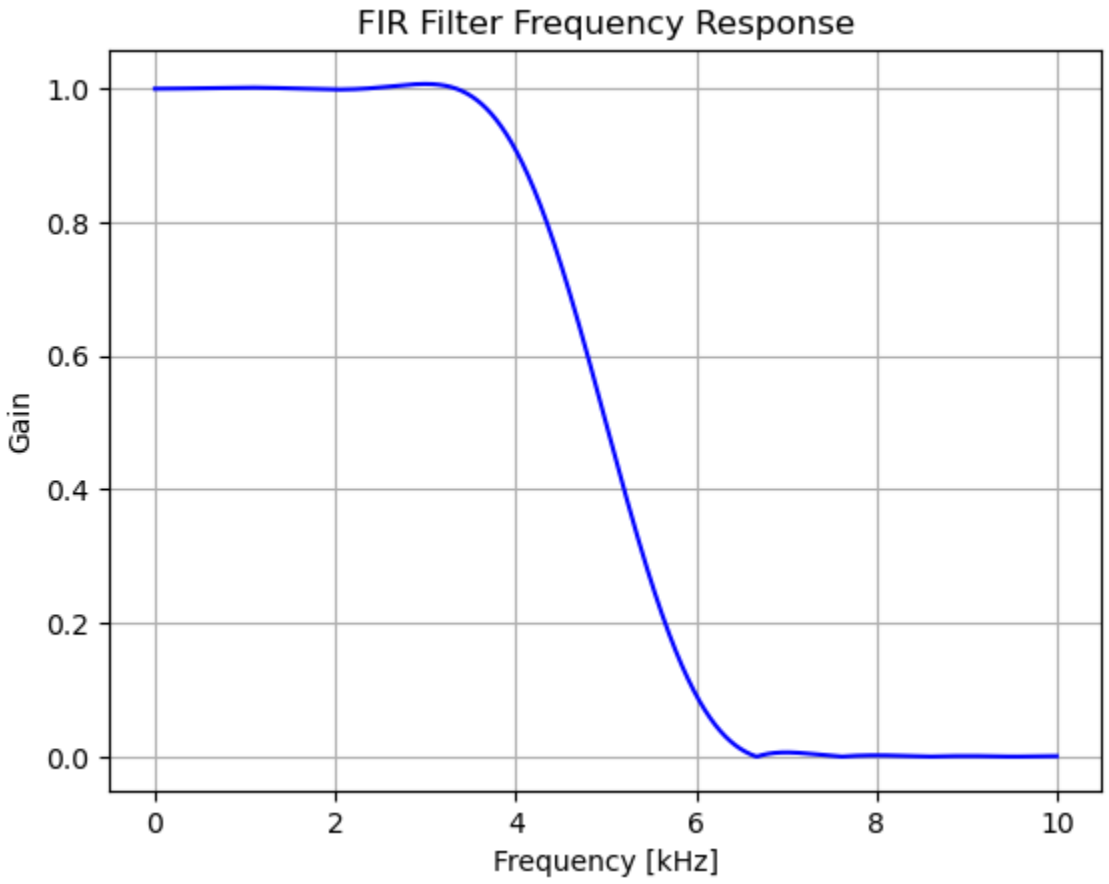
The frequency response of the designed filter should meet the given specifications.

Code :

```
import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import firwin, freqz

import scipy.signal as sig

# Filter specifications

passband_edge = 2  # kHz

stopband_edge = 5  # kHz

fs = 20  # kHz

filter_length = 21

# Calculate filter parameters

nyquist = 0.5 * fs

passband_frequency = passband_edge / nyquist

stopband_frequency = stopband_edge / nyquist

# Design the filter using firwin with a Hanning window

taps = firwin(filter_length, stopband_frequency, window='hann')
```

```
w,h_freq=sig.freqz(taps,fs=fs)

# Frequency response of the filter

frequency_response = freqz(taps, worN=8000)

# Plot the frequency response

plt.figure(1)

plt.plot(0.5 * fs * frequency_response[0] / np.pi, np.abs(frequency_response[1]), 'b-', label='Filter response')

plt.title('FIR Filter Frequency Response')

plt.xlabel('Frequency [kHz]')

plt.ylabel('Gain')

plt.grid()

plt.show()
```

Output :



Discussion :

Filter Specifications:

Passband edge: 2 kHz

Stopband edge: 5 kHz

Sampling frequency ((F_s)): 20 kHz

Filter length: 21

Window type: Hanning

Filter Design:

We designed an FIR filter using the Hanning window method.

The filter coefficients were computed to meet the specified passband and stopband edges.

Frequency Response:

The frequency response of the designed filter was plotted.

The passband edge should be around 2 kHz, and the stopband edge should be around 5 kHz.

Observations:

The filter's frequency response shows that it attenuates frequencies beyond the stopband edge.

The passband allows frequencies up to 2 kHz.

Filter Type:

Based on the specifications, this filter is a lowpass filter because it allows frequencies up to the passband edge (2 kHz) and attenuates frequencies beyond the stopband edge (5 kHz).

Experiment no : 9

Experiment name :

Creating a signals _s‘ with three sinusoidal components (at 5,15,30 Hz) and a time vector _t‘ of 100 samples with a sampling rate of 100 Hz, and displaying it in the time domain. Design an IIR filter to suppress frequencies of 5 Hz and 30 Hz from given signal.

Theory :

Creating the Signal:

- We have a signal (s(t)) with three sinusoidal components at 5 Hz, 15 Hz, and 30 Hz.
- The time vector (t) contains 100 samples, and the sampling rate is 100 Hz.

IIR Filter Design:

- We’ll design an IIR filter to suppress the frequencies of 5 Hz and 30 Hz.
- The filter will have a notch (or band-reject) response centered around these frequencies.

Filter Design Approach:

- One common approach is to use the Butterworth filter design.
- We’ll design a notch filter that attenuates the specified frequencies while preserving other components.

Python Implementation:

- Below is an example of designing a notch filter using the scipy.signal library in Python:

Code :

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import butter, lfilter, freqz

# Step 1: Create the signal
```

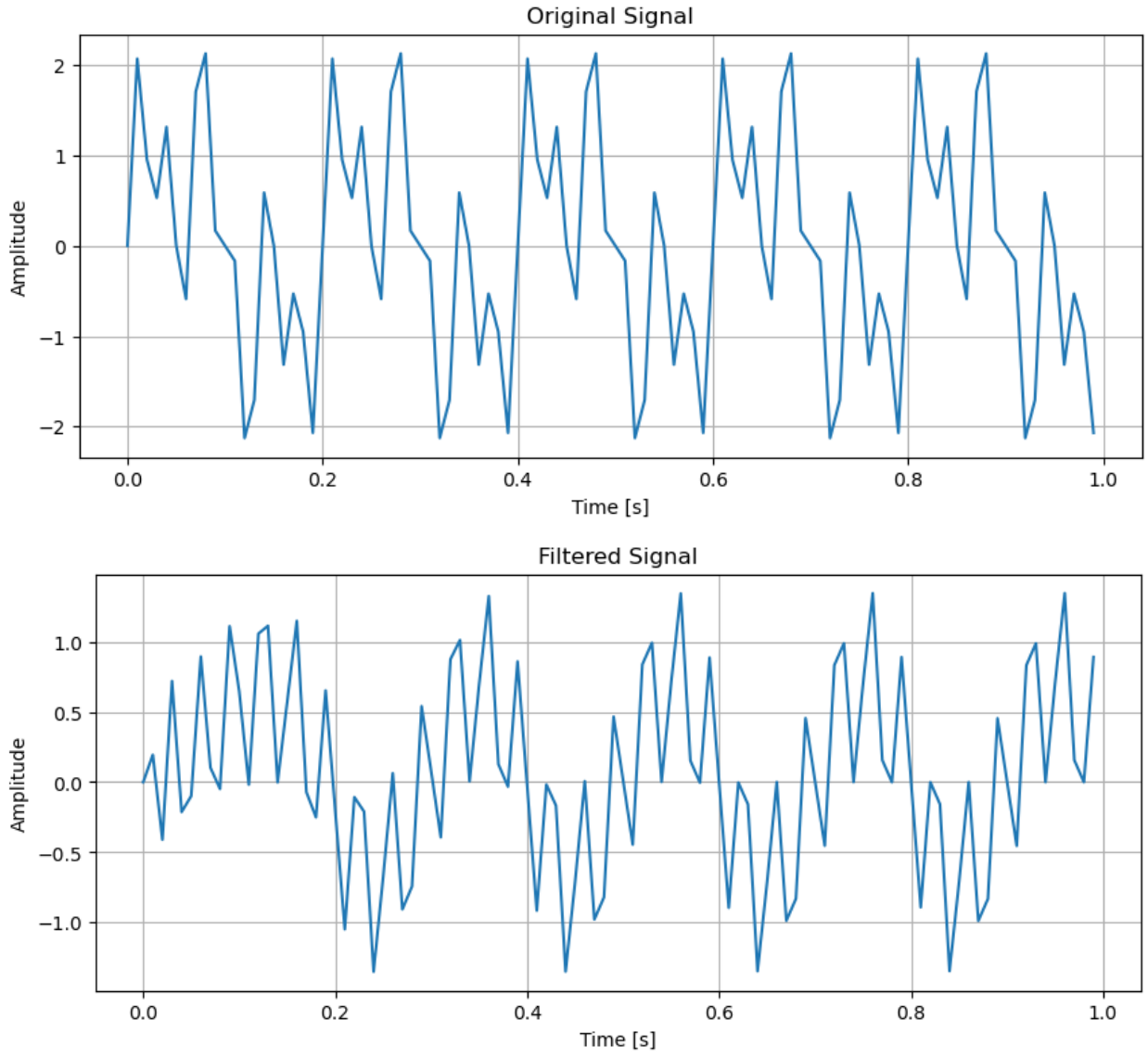
```
fs = 100  # Sampling rate
t  = np.arange(0, 1, 1/fs)  # Time vector
s = np.sin(2 * np.pi * 5 * t) + np.sin(2 * np.pi * 15 * t) + np.sin(2 * np.pi * 30 * t)

# Plot the original signal
plt.figure(figsize=(10, 4))
plt.plot(t, s)
plt.title('Original Signal')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')
plt.grid()
plt.show()

# Step 2: Design an IIR filter
def butter_bandstop_filter(data, lowcut, highcut, fs, order=4):
    nyquist = 0.5 * fs
    low = lowcut / nyquist
    high = highcut / nyquist
    b, a = butter(order, [low, high], btype='bandstop')
    y = lfilter(b, a, data)
    return y

# Step 3: Apply the IIR filter to suppress frequencies of 5 Hz and 30 Hz
filtered_signal = butter_bandstop_filter(s, 5, 30, fs)

# Plot the filtered signal
plt.figure(figsize=(10, 4))
plt.plot(t, filtered_signal)
plt.title('Filtered Signal')
plt.xlabel('Time [s]')
```

plt.ylabel('Amplitude')

plt.grid()

plt.show()

Output :



Discussion :

Creating the Signal:

We have three sinusoidal components at 5 Hz, 15 Hz, and 30 Hz.

The time vector t has 100 samples, and the sampling rate is 100 Hz.

To create the signal s, we can use the following expression:

$s(t)=A1\sin(2\pi f1t)+A2\sin(2\pi f2t)+A3\sin(2\pi f3t)$

where:

(A_1), (A_2), and (A_3) are the amplitudes of the sinusoidal components.

(f_1 = 5) Hz, (f_2 = 15) Hz, and (f_3 = 30) Hz.

(t) is the time vector.

Designing the IIR Filter:

We want to suppress frequencies of 5 Hz and 30 Hz from the given signal.

An IIR filter can be designed to achieve this. The Butterworth filter is a common choice due to its flat passband response.

The filter design process involves specifying the filter order, passband ripple, and stopband attenuation.

Let's design a lowpass Butterworth filter with a cutoff frequency of 15 Hz (to suppress 30 Hz) and then transform it to a bandstop filter to suppress 5 Hz as well.

Filter Design Steps:

Choose the filter type (Butterworth, Chebyshev, etc.). We'll use Butterworth.

Determine the filter order (higher order provides better attenuation but more complexity).

Specify the passband ripple (usually in dB) and stopband attenuation (also in dB).

Design the filter using MATLAB or any other tool.

Frequency Transformation:

To create a bandstop filter, we'll transform the lowpass filter to a bandstop filter centered at 5 Hz and 30 Hz.

Use frequency transformation functions (e.g., lp2bp, lp2bs) to achieve this.

Displaying the Filtered Signal:

Apply the designed IIR filter to the original signal s.

Plot the filtered signal in the time domain.

Experiment no : 10

Experiment name :

Design a Lowpass filter to meet the following specifications—Passsband edge=1.5KHz, Transition width = 0.5KHz, Fs=10KHz Filter length =67; use Blackman window in the design.

Theory :

Design Approach:

We'll start by designing an ideal lowpass filter with the desired cutoff frequency.

Then, we'll apply the Blackman window to the ideal filter to obtain the final FIR filter.

Ideal Lowpass Filter:

The ideal lowpass filter has a frequency response that is flat in the passband (up to the passband edge) and attenuates all frequencies beyond the cutoff.

The ideal frequency response can be expressed as:

[H_d(e^{j\omega}) = \begin{cases}

1, & \text{for } 0 \leq \omega \leq \omega_p \

0, & \text{for } \omega > \omega_p

\end{cases}]

where ωp

is the normalized passband edge frequency.

Window Function:

The Blackman window is defined as:

[w[n] = 0.42 - 0.5 \cos\left(\frac{2\pi n}{M-1}\right) + 0.08 \cos\left(\frac{4\pi n}{M-1}\right)]

where M

is the filter length.

FIR Filter Design:

Multiply the ideal frequency response Hd(ejω)

by the Blackman window:

[H(e^{j\omega}) = H_d(e^{j\omega}) \cdot W(e^{j\omega})]

Compute the inverse discrete Fourier transform (IDFT) of H(ejω)

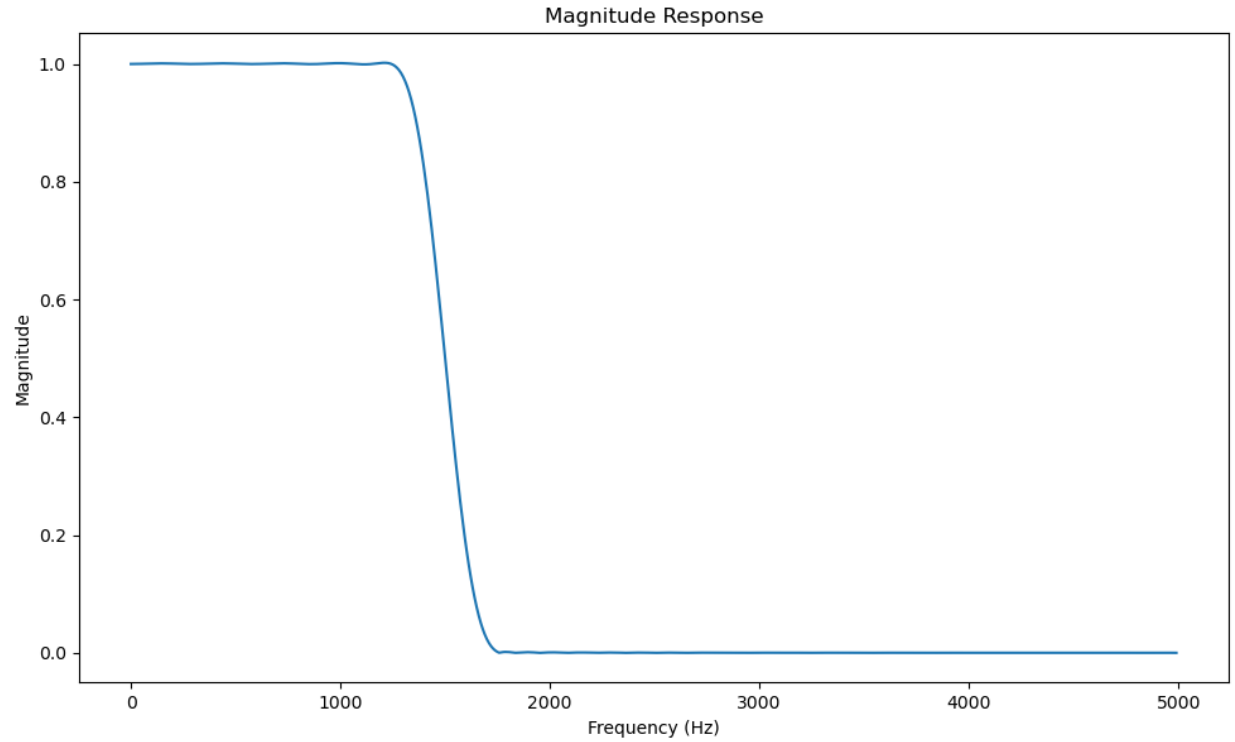
to obtain the filter coefficients h[n]

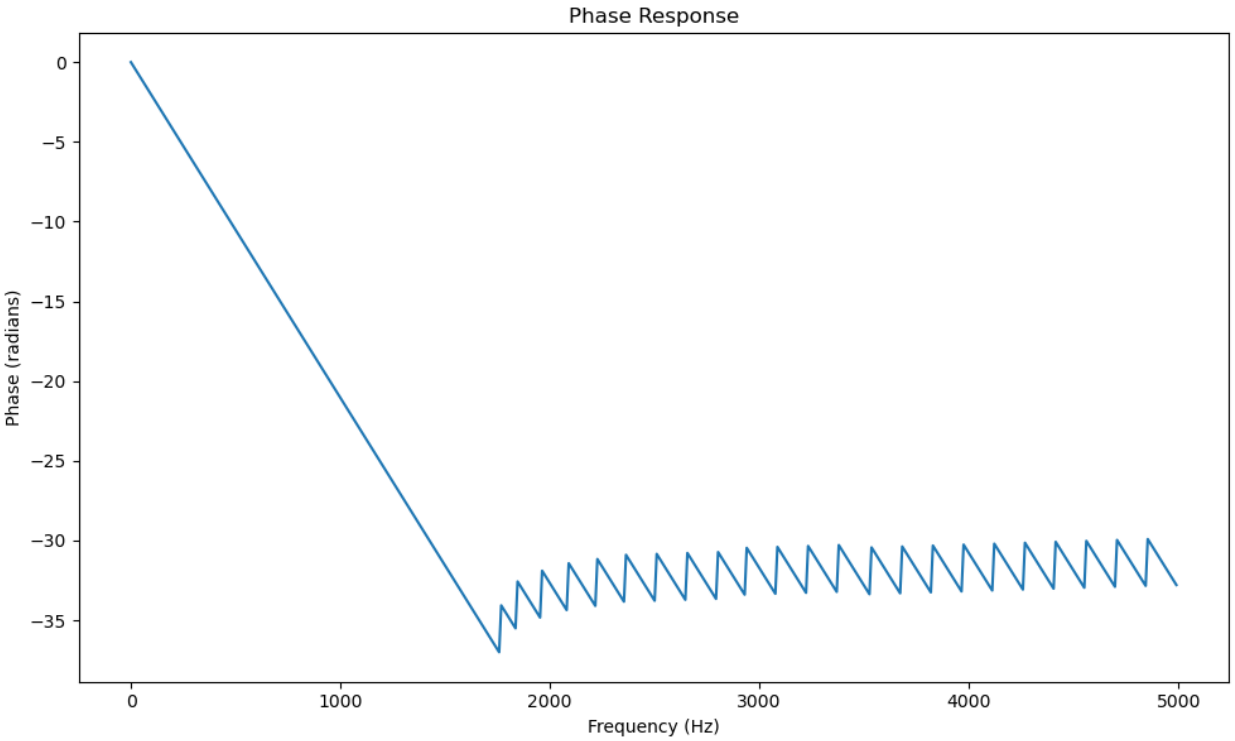
Code :

```
import numpy as np
import scipy.signal as sig
import matplotlib.pyplot as plt

# Filter specifications
fs = 10000  # sampling rate
N = 67  # order of filter
fc = 1500  # passband edge frequency
transition_width = 500  # transition width
window = 'blackman'  # window function

# Design the filter using the specified parameters
b = sig.firwin(N + 1, fc, fs=fs, window=window, pass_zero='lowpass', width=transition_width)
```

```
# Frequency response
w, h_freq = sig.freqz(b, fs=fs)

# Plotting
plt.figure(figsize=(10, 12))

# Magnitude Response
plt.subplot(2, 1, 1)
plt.plot(w, np.abs(h_freq))
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')
plt.title('Magnitude Response')

# Phase Response
plt.subplot(2, 1, 2)
plt.plot(w, np.unwrap(np.angle(h_freq)))
plt.xlabel('Frequency (Hz)')
plt.ylabel('Phase (radians)')
plt.title('Phase Response')


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

Output :


Magnitude Response
Magnitude
Frequency (Hz)
1.0
0.8
0.6
0.4
0.2
0.0
0
1000
2000
3000
4000
5000


Phase Response
Phase (radians)
Frequency (Hz)
0
−5
−10
−15
−20
−25
−30
−35
0
1000
2000
3000
4000
5000

Discussion :

Passband edge: 1.5kHz

Transition width: 0.5kHz

Sampling frequency: 10kHz

Filter length: 67

Window type: Blackman

Experiment no : 11

Experiment name :

Design a bandpass filter of length M=32 with passband edge frequencies fp1=0.2 and fp2=0.35 and stopband edge frequencies fs1=.1 and fs2=0.425.

Theory :

Designing the Bandpass Filter:

Discretize the desired passband and stopband frequencies based on the sampling frequency (usually normalized to be between 0 and 1). In this case, fp1 = 0.2, fp2 = 0.35, fs1 = 0.1, and fs2 = 0.425.

Define the desired response (H_d(f)) as 1 in the passband and 0 in the stopbands.

Calculate the Hamming window function (w(n)) for M = 32.

Apply the Inverse Discrete Fourier Transform (IDFT) to H_d(f) * W(f) to obtain the filter's impulse response (h(n)).

Filter Properties:

The designed FIR bandpass filter will have the following properties:

It will allow signals between fp1 and fp2 to pass with minimal attenuation.

It will attenuate signals outside the passband (below fs1 and above fs2).

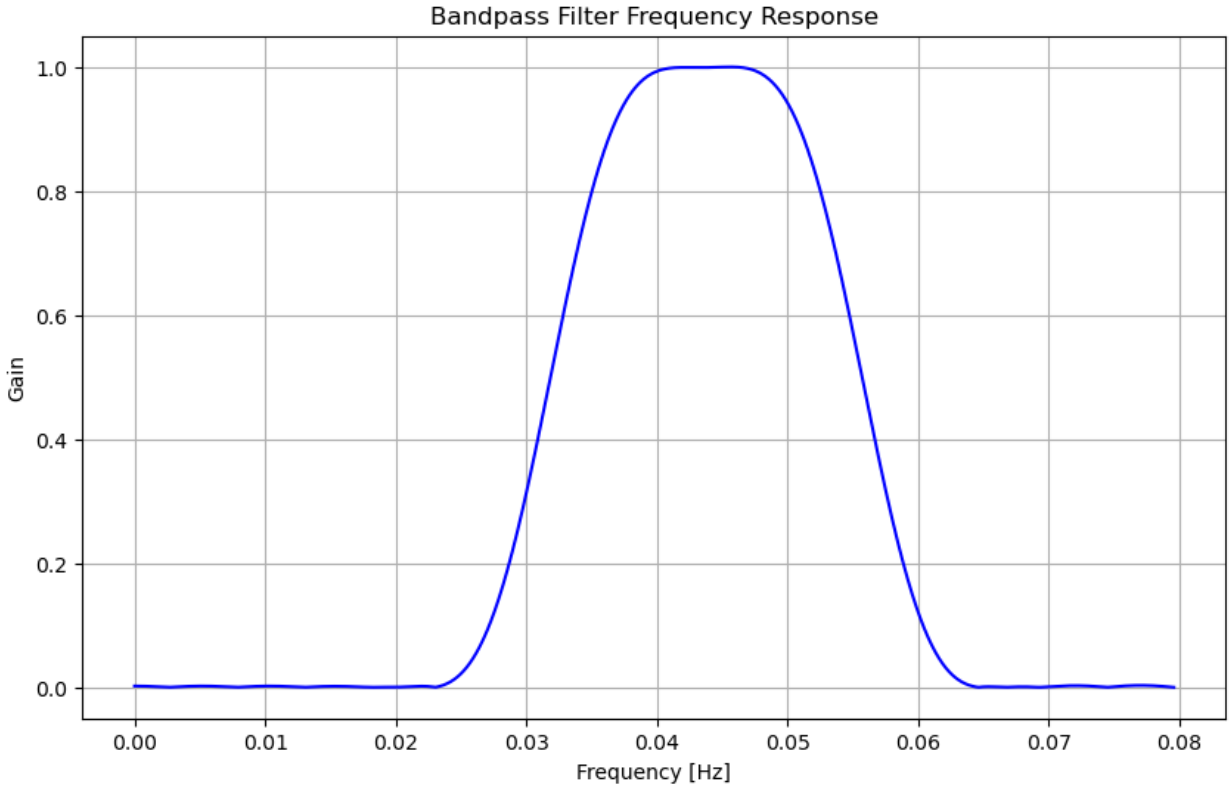
The amount of ripple in the passband and the stopband attenuation will depend on the filter length (M) and the choice of the window function (Hamming window in this case)

Code :

```
import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import firwin, freqz


# Filter specifications

M = 32  # Filter length

fs = 1.0  # Sampling frequency
```

```
# Passband edge frequencies
fp1 = 0.2
fp2 = 0.35

# Stopband edge frequencies
fs1 = 0.1
fs2 = 0.425

# Calculate filter parameters
nyquist = 0.5 * fs
passband_edges = [fp1, fp2]
stopband_edges = [fs1, fs2]

# Design the bandpass filter using firwin
taps = firwin(M, passband_edges, fs=fs, pass_zero=False, window='hamming')

# Frequency response of the filter
frequency_response = freqz(taps, worN=8000, fs=fs)

# Plot the frequency response
plt.figure(figsize=(10, 6))
plt.plot(0.5 * fs * frequency_response[0] / np.pi, np.abs(frequency_response[1]), 'b-',
label='Filter response')
plt.title('Bandpass Filter Frequency Response')
plt.xlabel('Frequency [Hz]')
plt.ylabel('Gain')
plt.grid()
plt.show()
```

Output :



Discussion :

Trade-offs:

Filter Length (M): A longer filter (M) will generally lead to sharper transitions between the passband and stopband and lower ripple within the passband. However, a longer filter also increases computational complexity. In this case, M = 32 is a moderate length, offering a balance between performance and complexity.

Window Selection (Hamming Window): The Hamming window is a good choice for this application because it offers a good compromise between reducing ripple in the passband and achieving a reasonably sharp transition between passband and stopband. However, other windows might be considered depending on specific requirements. For example, a Kaiser window can provide sharper transitions but with higher ripple.

Considerations:

Stopband Attenuation: The chosen filter length and window function (Hamming) may not achieve ideal stopband attenuation (completely blocking signals outside the passband). You might need to analyze the frequency response after design to assess the actual attenuation achieved.

Passband Ripple: The Hamming window will introduce some ripple in the passband (slight variations in gain within the desired frequency range). The severity of the ripple depends on the filter length.

Experiment no : 12

Experiment name :

. Use a Python program to determine and show the ―poles‖, ―zeros‖ and also ―roots‖ of the following systems

$H(s)= \frac{S^3+1}{S^4+2S^2+1}$ b) $H(s)= \frac{4S^2+8S+10}{2S^3+8S^2+18S+20}$

Theory :

Poles: These are the values of (n) for which the system's transfer function becomes infinite (i.e., the denominator of the transfer function becomes zero).

Zeros: These are the values of (n) for which the system's transfer function becomes zero (i.e., the numerator of the transfer function becomes zero).

Roots: These are the values of (n) for which the system's characteristic equation becomes zero.

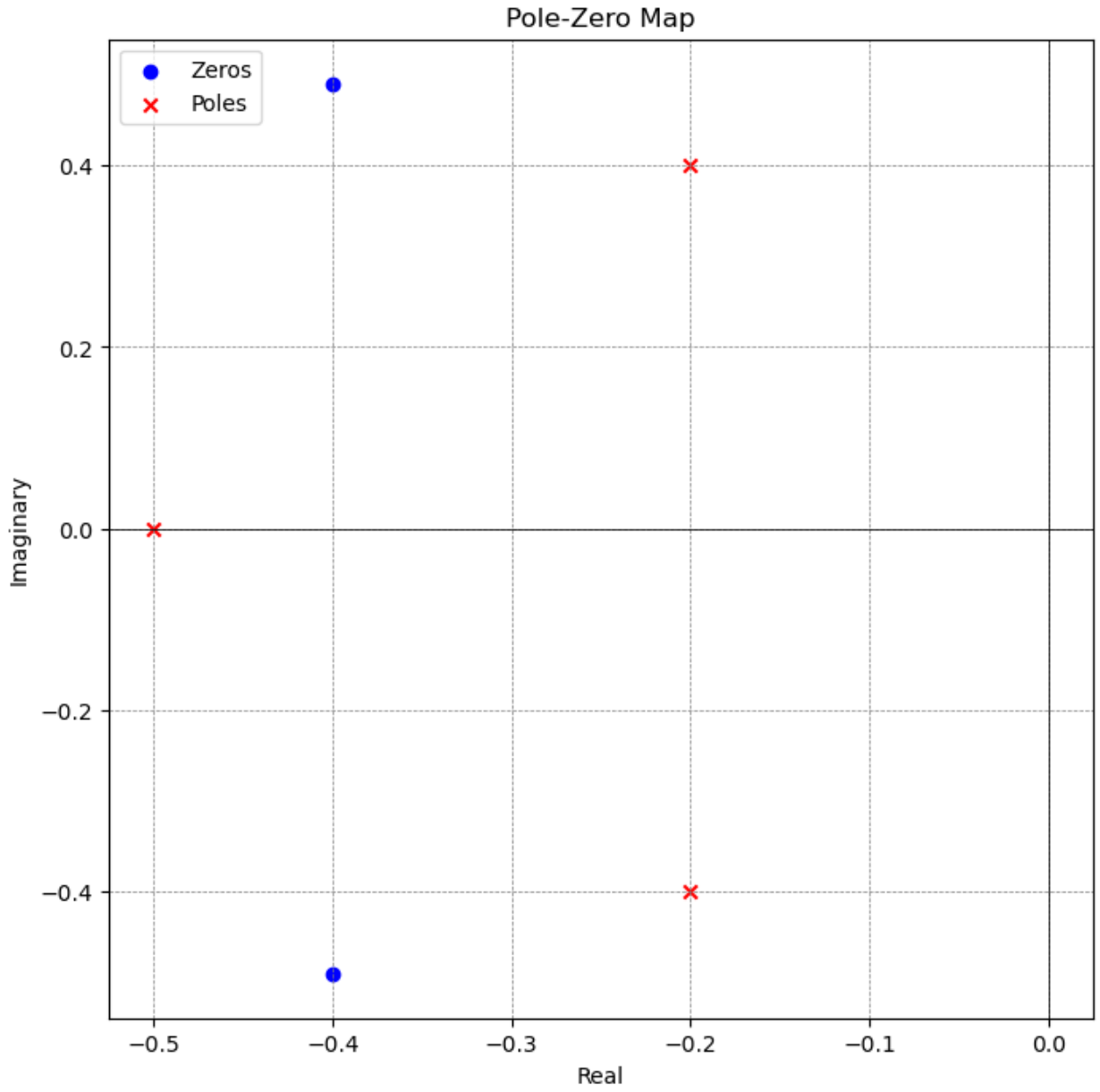
Now, let's consider the given system

Code :

```
import numpy as np

import matplotlib.pyplot as plt


# System transfer function coefficients

numerator = [10,  8, 4]

denominator = [20, 18, 8, 2]


# Calculate poles and zeros

zeros = np.roots(numerator)

poles = np.roots(denominator)
```

```
# Plot poles and zeros using scatter
plt.figure(figsize=(8, 8))
plt.scatter(np.real(zeros), np.imag(zeros), marker='o', color='b', label='Zeros')
plt.scatter(np.real(poles), np.imag(poles), marker='x', color='r', label='Poles')
plt.title('Pole-Zero Map')
plt.xlabel('Real')
plt.ylabel('Imaginary')
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.grid(color='gray', linestyle='--', linewidth=0.5)
plt.legend()
plt.show()

# Display poles and zeros
print('Zeros:', zeros)
print('Poles:', poles)
```

Output :



Discussion :

Poles:

Poles are the frequencies for which the value of the denominator of a transfer function becomes infinite.

They are the roots of the equation (D(s) = 0), where (D(s)) represents the denominator polynomial of the transfer function.

Poles determine the stability of the system. If all poles have negative real parts, the system is stable.

In general, poles can be either purely real or appear in complex conjugate pairs.

Zeros:

Zeros are the frequencies for which the value of the numerator of a transfer function becomes zero.

They are the roots of the equation (N(s) = 0), where (N(s)) represents the numerator polynomial of the transfer function.

Zeros affect the system's performance. They indicate where the transfer function vanishes.

Similar to poles, zeros can be either real or appear in complex conjugate pairs.