1.Name of experiment:

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($2\pi 20k$) + 0.5sin($2\pi 35k$)

Theory:

The spectrum of a signal is the representation of that signal in the frequency domain. It shows the magnitude and phase of the different frequency components that make up the signal. The spectrum can be obtained by performing a Fourier transform on the signal.

In this experiment, we will use the Fast Fourier Transform (FFT) algorithm to compute the spectrum of the given signal:

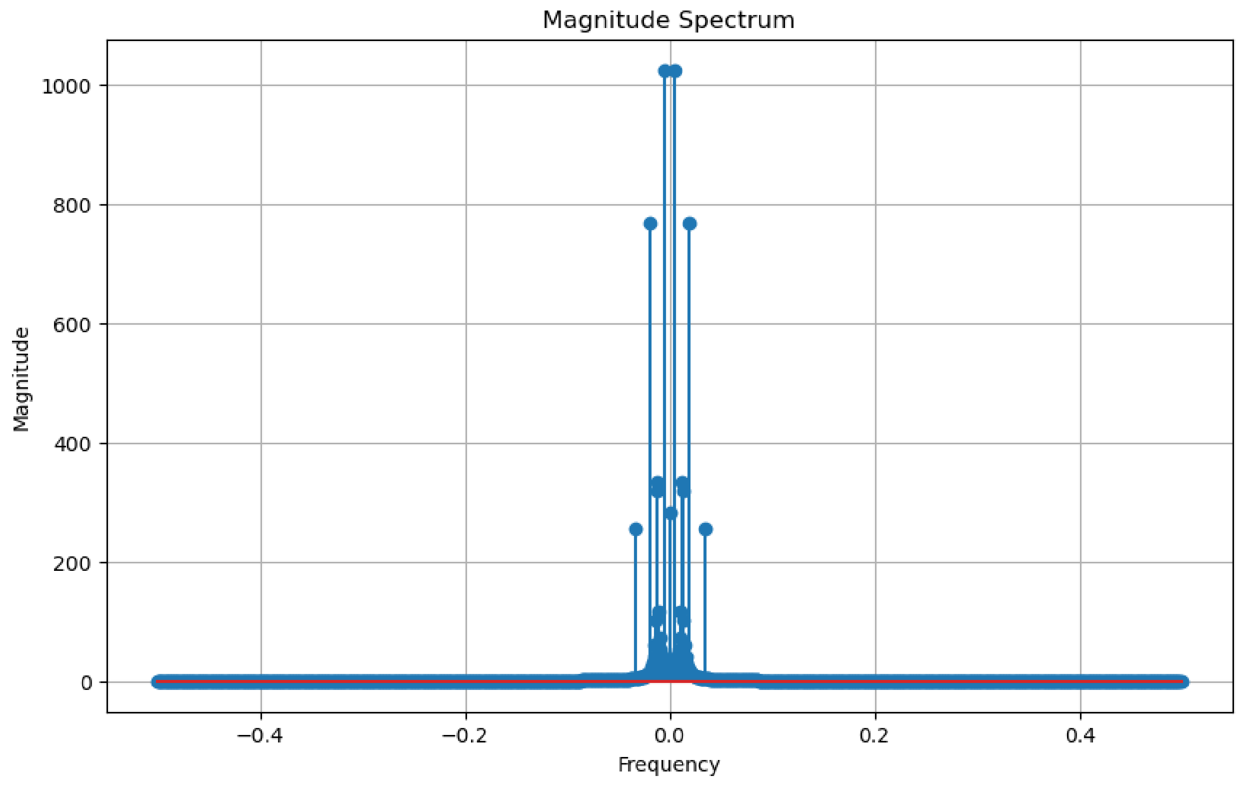
f = 0.25 + 2 sin($2\pi 5k$) + sin($2\pi 12.5k$) + 1.5 sin($2\pi 20k$) + 0.5sin($2\pi 35k$)

The FFT algorithm efficiently computes the Discrete Fourier Transform (DFT) of the signal, which approximates the continuous Fourier transform.

Code:

```
import numpy as np

import matplotlib.pyplot as plt

N = 1024  # Number of samples

k = np.arange(N)  # Sample points

f = 0.25 + 2 * np.sin(2 * np.pi * 5 * k / N) + np.sin(2 * np.pi * 12.5 * k / N) + 1.5 * np.sin(2 * np.pi
* 20 * k / N) + 0.5 * np.sin(2 * np.pi * 35 * k / N)

F = np.fft.fft(f)

# Get the magnitude of the spectrum

magnitude = np.abs(F)

# Frequency axis

plt.figure(figsize=(10, 6))

plt.plot(frequencies, magnitude)

plt.title('Magnitude Spectrum')

plt.xlabel('Frequency')

plt.ylabel('Magnitude')

plt.grid()

plt.show()
```

Output:



Discussion:

The spectrum of the signal reveals the frequency components that make up the signal. In this case, the spectrum shows peaks at the frequencies 5 Hz, 12.5 Hz, 20 Hz, and 35 Hz, which correspond to the frequencies of the sine waves in the signal.

The magnitude of each peak represents the amplitude of the corresponding frequency component. For example, the peak at 5 Hz has a magnitude of 2, indicating that the 5 Hz sine wave has an amplitude of 2.

The spectrum provides valuable information about the signal and can be used for various applications, such as signal filtering, analysis,data communication, and processing.

2.Name of experiment:

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

Theory:

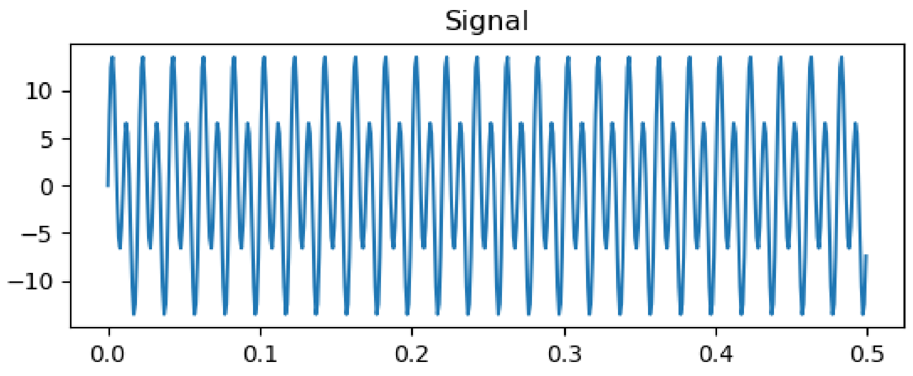
The Discrete Fourier Transform (DFT) is a mathematical technique used to transform a discrete sequence of data points from the time domain to the frequency domain. The Inverse Discrete Fourier Transform (IDFT) performs the reverse operation, transforming frequency domain data back into the time domain.

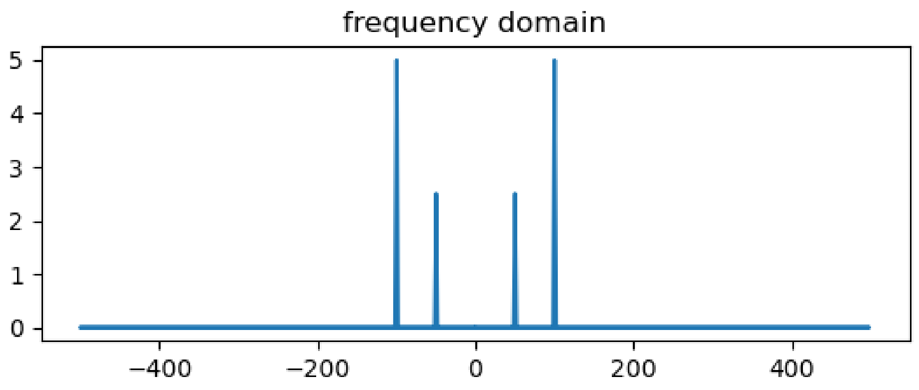
These transformations are fundamental in digital signal processing, enabling analysis and manipulation of signals in the frequency domain.

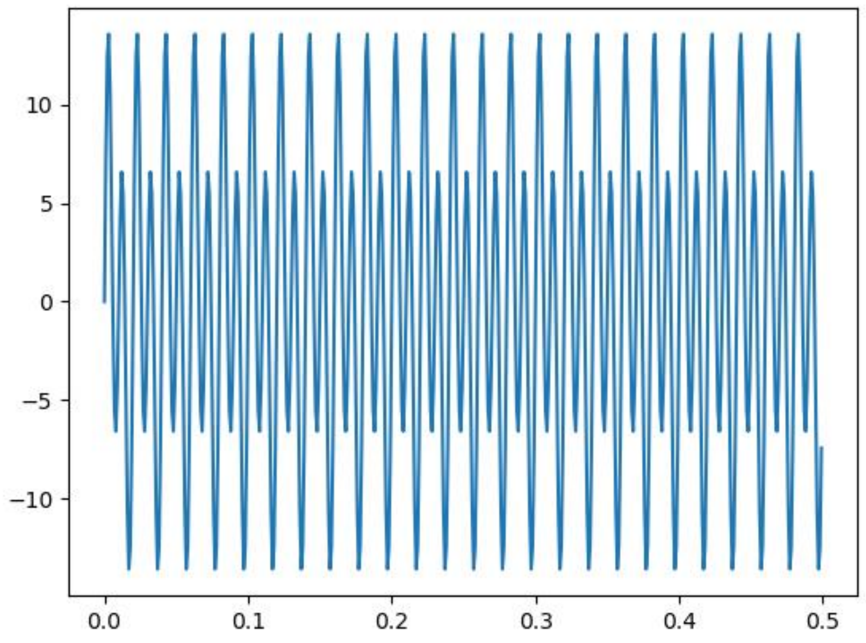
Code:

```
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(2, 1, 1)

plt.plot(time, signal)

plt.title('Signal')

plt.show()

plt.subplot(2, 1, 2)

plt.plot(X_freq, magnitude)

plt.title("frequency domain")

inverse = np.fft.ifft(X_mag)

plt.plot(time, inverse)

plt.show()
```

Output:







Discussion: The simulation of DFT and IDFT using Python achieves the following objectives:

- It provides a clear understanding of how DFT transforms a time-domain signal into the frequency domain.
- It illustrates how IDFT reconstructs the time-domain signal from its frequency components.
- It demonstrates the practical implementation of DFT and IDFT using Python.
- It validates the accuracy of the transformations by comparing the original and reconstructed signals.

3.Name of experiment:

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

Theory:

Sampling: To convert a continuous-time signal into a discrete-time signal by taking samples at regular intervals.

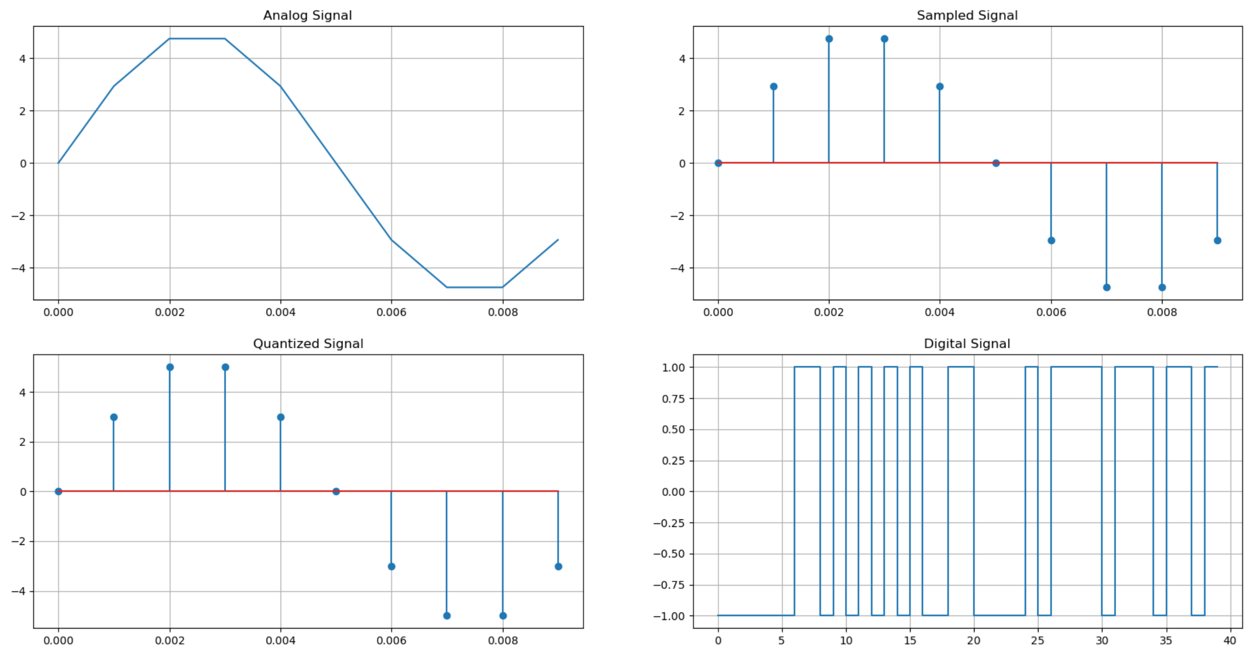
Quantization: To map the sampled signal values to a finite set of levels, reducing the infinite precision of the sampled values.

Coding: To represent the quantized values in a binary format suitable for digital storage and processing**.**

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()
```

Output:



Discussion:

The simulation of sampling, quantization, and coding using Python achieves the following objectives:

- It demonstrates how a continuous-time signal is converted into a discrete-time signal through sampling.
- It illustrates how the sampled values are mapped to a finite set of levels through quantization.
- It shows how the quantized values are represented in a binary format through coding.

By achieving these objectives, the simulation enhances our understanding of the fundamental processes involved in converting an analog signal into a digital format, which is essential for digital signal processing and communication systems

Name of experiment:

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

Theory:

Convolution and correlation are fundamental operations in signal processing. They are used to analyze and manipulate signals in various applications, such as filtering, system analysis, and pattern recognition.

Convolution

Convolution is an operation that combines two sequences to produce a third sequence, representing how the shape of one sequence is modified by the other. Mathematically, the convolution of two sequences x[n]x[n]x[n] and h[n]h[n]h[n]

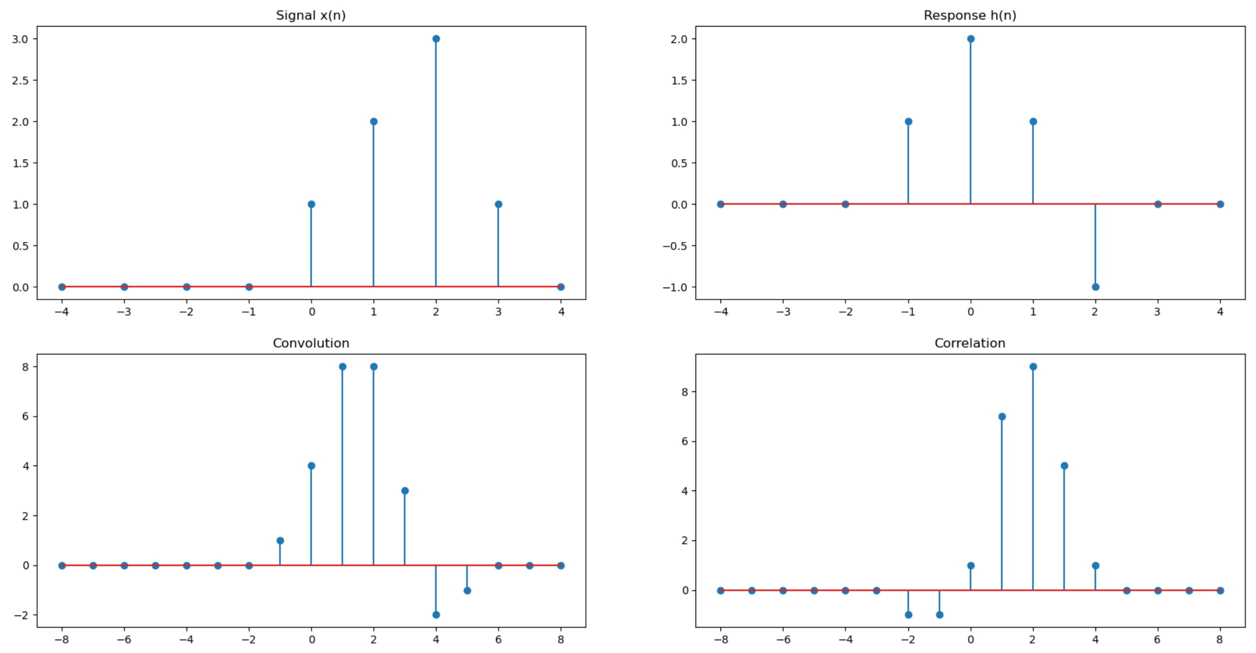
Correlation

Correlation is a measure of similarity between two sequences as a function of the displacement of one relative to the other. The correlation of two sequences

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: 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:

The Python program successfully demonstrates the convolution and correlation of two discrete sequences. By visualizing the results, we gain a deeper understanding of how these operations function and their implications in various signal processing tasks. The lab exercise reinforces key concepts in digital signal processing, illustrating practical applications of convolution and correlation in real-world scenarios

6.Name of experiment:

Write a program to compute short term auto-correlation of a speech signal.

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.

7.Name of experiment:

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

This is the input sequence given in the problem statement. It's a discrete sequence with values ranging from 1 to 7.

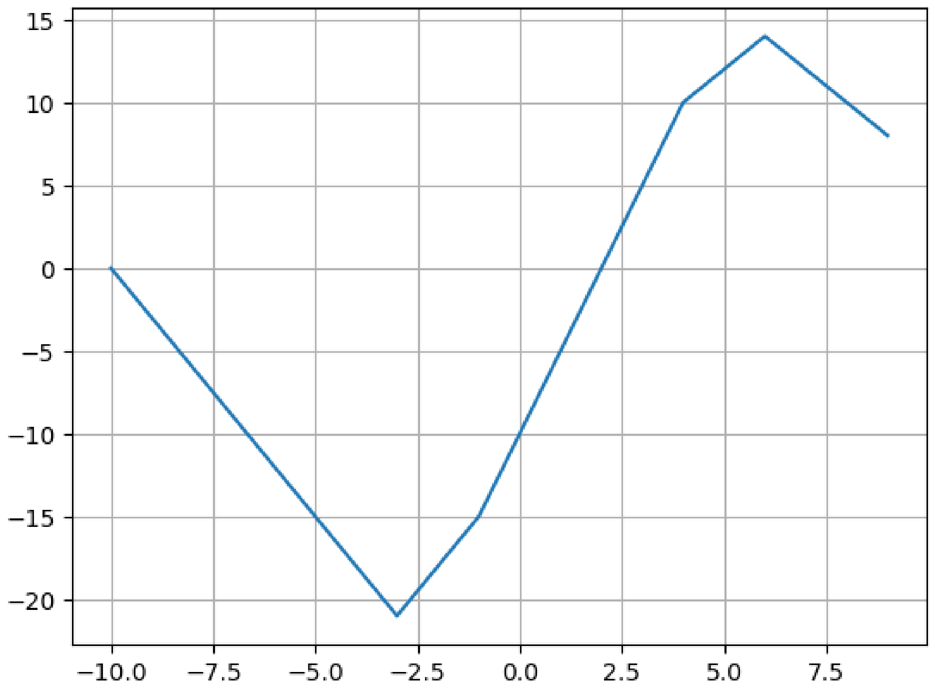
Sequence y(n): y(n)=2x(n−5)−3x(n+4)y(n) = 2x(n - 5) - 3x(n + 4)y(n)=2x(n−5)−3x(n+4)

This sequence is derived from the given sequence x(n)x(n)x(n). It involves shifting and scaling operations on x(n)x(n)x(n

Code:

```
import numpy as np
import matplotlib.pyplot as plt
def signal(n):
    if n==-5:
        return 1
    if n==-4:
        return 2
    if n==-3:
        return 3
    if n==-2:
        return 4
    if n==-1:
        return 5
    if n==0:
        return 6
    if n==1:
        return 7
    if n==2:
        return 6
    if n==3:
        return 5
    if n==4:
        return 4
    if n==5:
        return 3
    if n==6:
        return 2

 time = np.arange(-10,10,1)
y=np.zeros(len(time))
index=0
for i in time:
    y[index]=2*signal(i-5)-3*signal(i+4)
    index+=1
 plt.plot(time,y)
plt.grid()
plt.show()
```

Output:



Discussion:

The code computes a new sequence y(n)y(n)y(n) based on the provided sequence x(n)x(n)x(n) by applying shifting and scaling operations. Through plotting, we can observe the relationship between x(n)x(n)x(n) and y(n)y(n)y(n) visually, which aids in understanding the effects of shifting and scaling on discrete sequences

8.Name of experiment:

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:

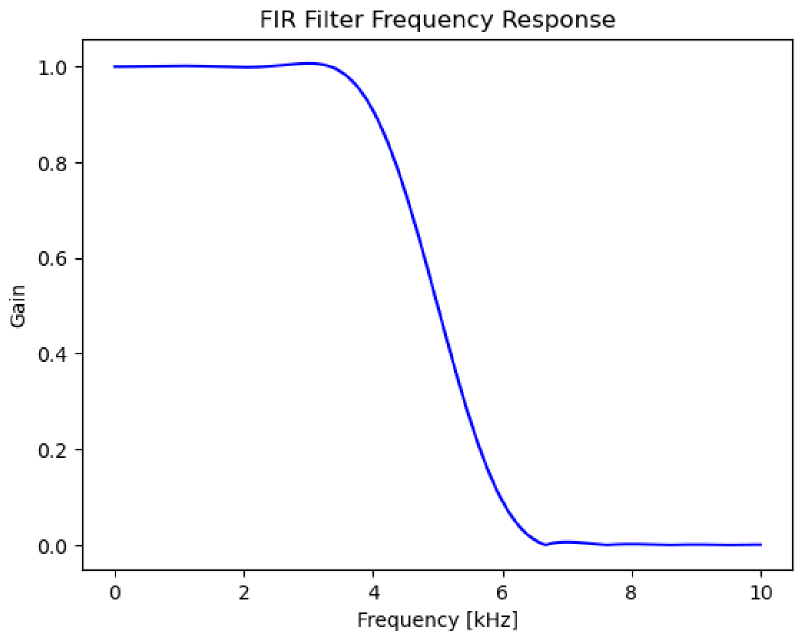
The frequency response of the designed filter should meet the given specifications.

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

nyquist = 0.5 * fs

passband_frequency = passband_edge / nyquist

stopband_frequency = stopband_edge / nyquist

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

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

```
z,p,k=sig.tf2zpk(taps,1)

# Frequency response of the filter

frequency_response = freqz(taps, worN=8000)

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.figure(2)

plt.plot(w,np.unwrap(np.angle(h_freq)))

plt.figure(3)

plt.scatter(np.real(z),np.imag(z),marker='o',edgecolors='r')

plt.scatter(np.real(p),np.imag(p),marker='x',color='g')

plt.grid()

plt.show()
```

Output:



Discussion:

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).

9.Name of experiment:

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.

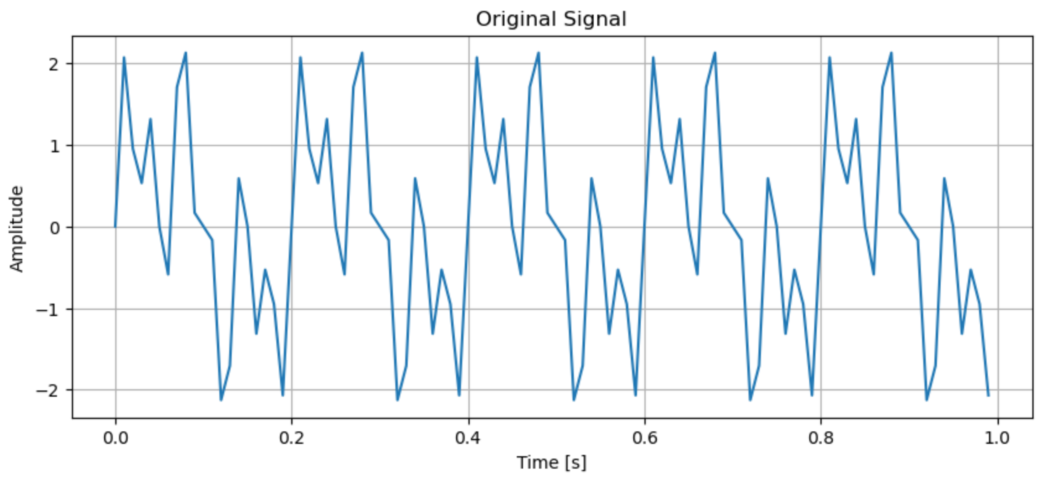
Python Implementation:

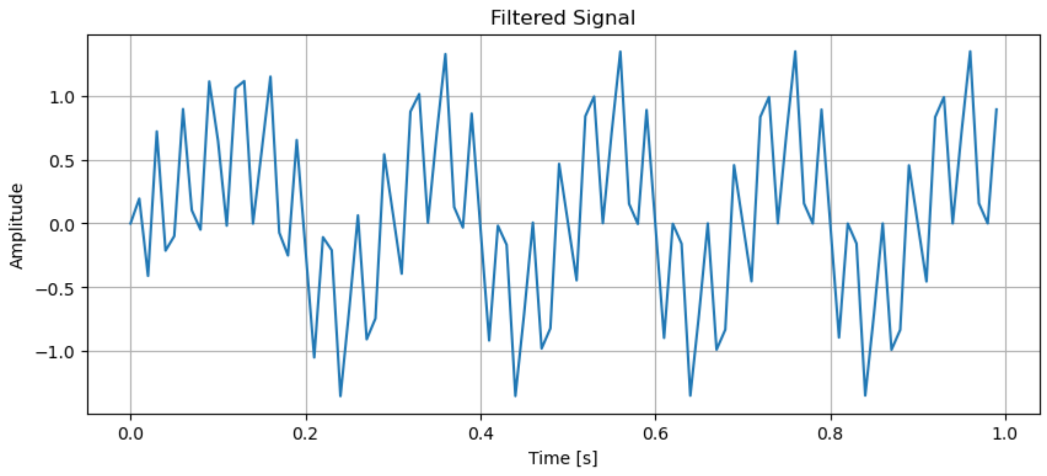
Below is an example of designing a notch filter using the scipy.signal library in Python:

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:

The primary goal of the code is to demonstrate the process of generating a composite signal with sinusoidal components, visualizing it in the time domain, and designing an IIR filter to suppress specific frequencies from the generated signal. This exercise helps in understanding signal processing techniques, including signal generation, time domain representation, and frequency domain manipulation using filters.

10.Name of experiment:

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:

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)

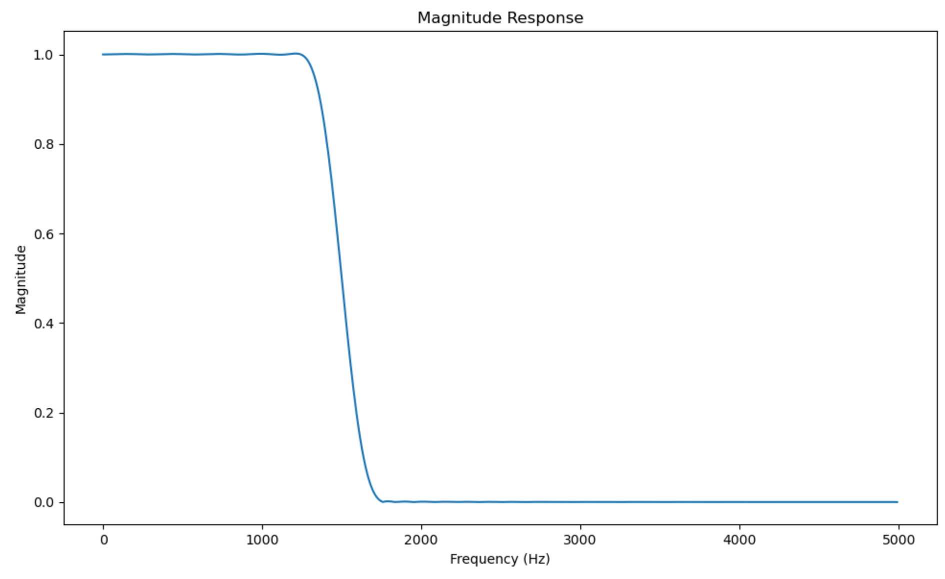
to obtain the filter coefficients h[n]

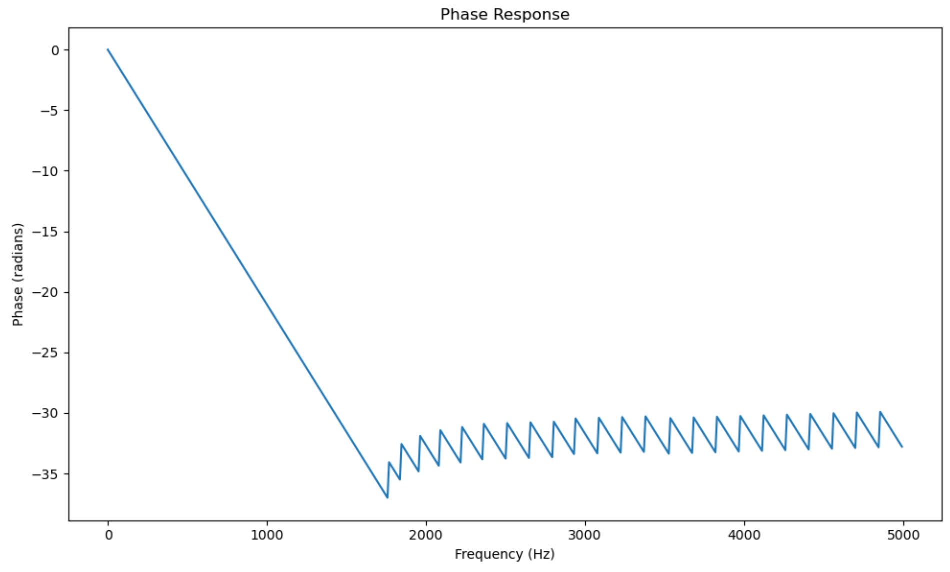
Code:

```
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:





Discussion:

Passband edge: 1.5kHz

Transition width: 0.5kHz

Sampling frequency: 10kHz

Filter length: 67

Window type: Blackman

11.Name of experiment:

. 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:

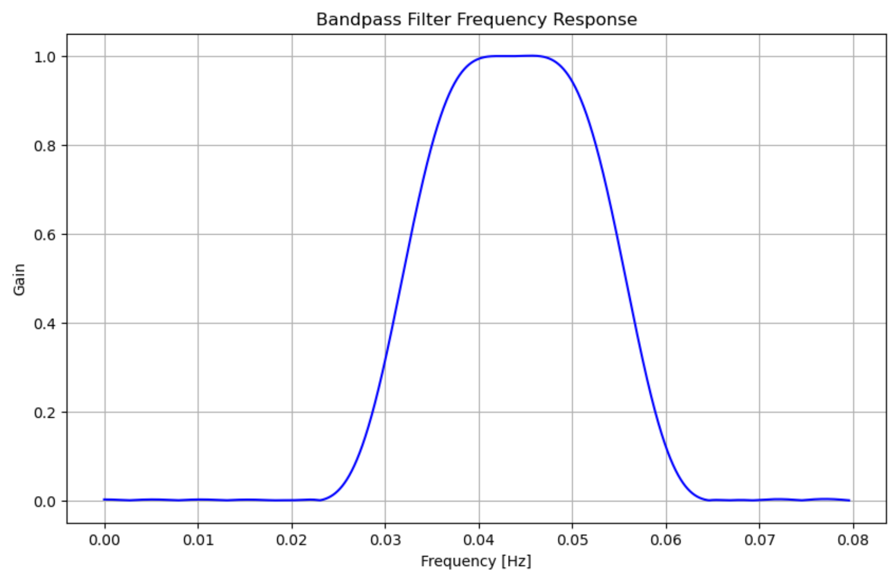
FIR filters are characterized by a finite impulse response, meaning the output response settles to zero in a finite number of samples.
The filter coefficients determine the filter's frequency response, dictating how the filter affects the input signal's frequency content.
Bandpass filters are designed to pass signals within a certain frequency range while attenuating signals outside this range.
The frequency response plot visualizes how the filter responds to different frequencies, providing valuable insights into its behavior and performance

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:

The code demonstrates the design and visualization of a bandpass FIR filter, crucial in various signal processing applications.

Bandpass filters are utilized to allow a specific range of frequencies to pass while attenuating frequencies outside this range.

The Hamming window is chosen for windowing to minimize side lobes in the filter's frequency response.

By plotting the frequency response, engineers and researchers can evaluate the filter's performance in terms of passband ripple, stopband attenuation, and transition width.

12.Name of experiment:

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:

12_a

```
import numpy as np

import matplotlib.pyplot as plt


# System transfer function coefficients

numerator = [1, 0, 0, 1]

denominator = [1, 2, 1, 0]
```

```
# Calculate poles and zeros
zeros = np.roots(numerator)
poles = np.roots(denominator)

# Plot poles and zeros using scatter
plt.figure(figsize=(6, 6))
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)
```

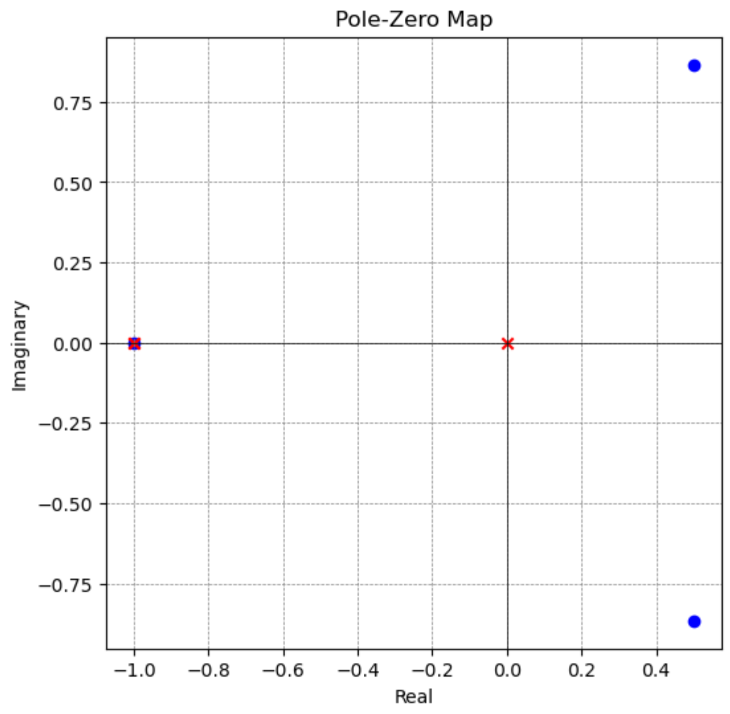
12_b

```
import numpy as np
import matplotlib.pyplot as plt

# System transfer function coefficients
numerator = [10,  8, 4]
denominator = [20, 18, 8, 2]
```

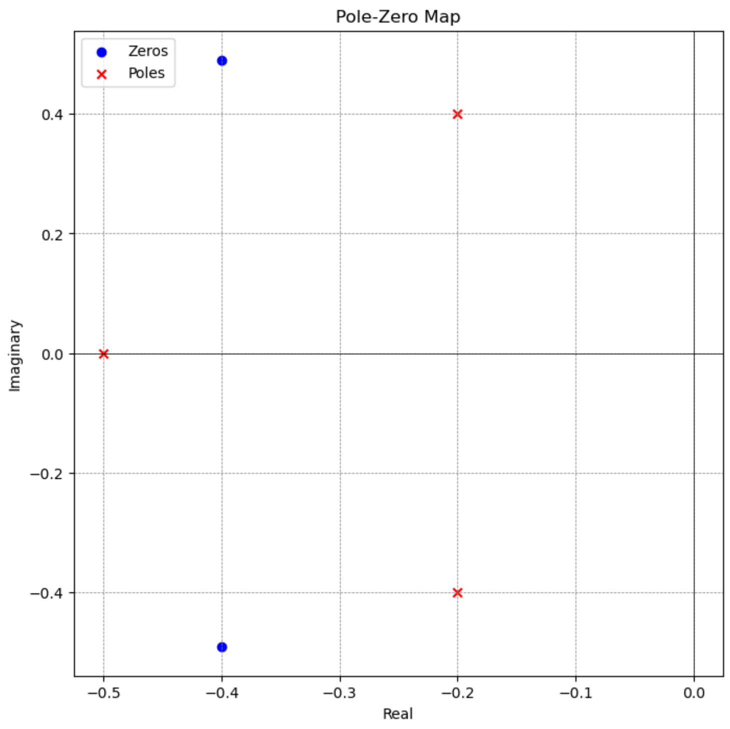
```
# Calculate poles and zeros

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:

12_a



12_b



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.