EXPERIMENT -01:

Write a Python program to find the spectrum of the following signal f = 0.25 + 2 sin(2π5k) + sin(2π12.5k) + 1.5 sin(2π20k) + 0.5sin(2π35k)

Theory:

To analyse a signal in the frequency domain, we use the Fourier Transform. The Fast Fourier Transform (FFT) is an algorithm to compute the Discrete Fourier Transform (DFT) efficiently.

The given signal is: 0.25 + 2 sin(2π5k) + sin(2π12.5k) + 1.5 sin(2π20k) + 0.5sin(2π35k)

This signal is composed of multiple sine waves with different frequencies and amplitudes, plus a constant offset of 0.25.

Steps to Find the Spectrum

1. Sample the Signal: Discretize the signal in the time domain.

2. Apply FFT: Use FFT to convert the time-domain signal into the frequency domain.

3. Calculate Magnitude: Determine the magnitude of the frequency components.

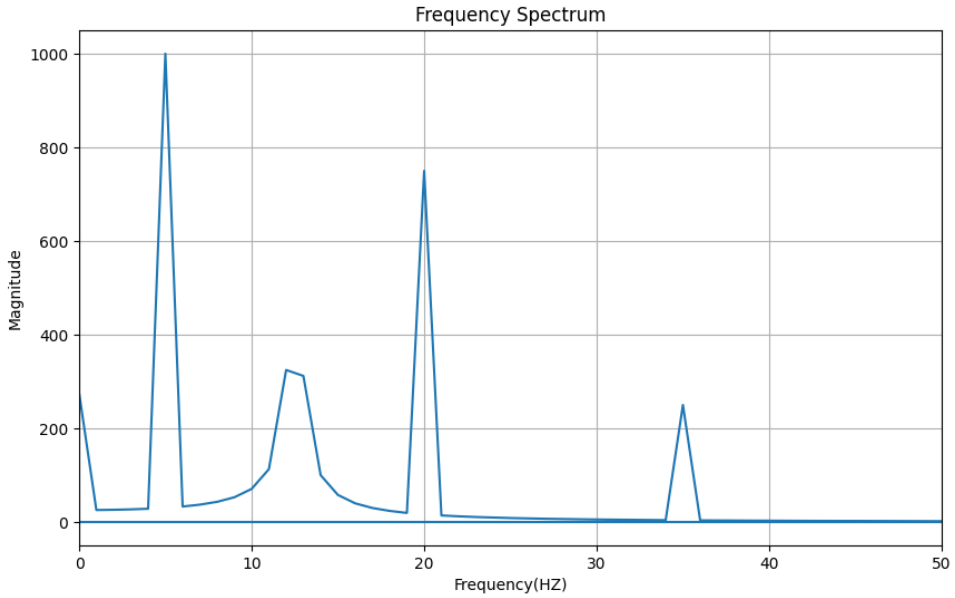
4. Plot the Spectrum: Visualize the spectrum by plotting the magnitude against the frequency.

Python Program:

```
import numpy as np # type: ignore

import matplotlib.pyplot as plt # type: ignore

sample_rate = 1000

duration = 1.0

t = np.linspace(0,duration,int(duration*sample_rate),endpoint=False)

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

f_fft = np.fft.fft(f)
```

```
frequency = np.fft.fftfreq(len(f),1/sample_rate)
magnitude = np.abs(f_fft)
plt.figure(figsize=(10,6))
plt.plot(frequency,magnitude)
plt.title("Frequency Spectrum")
plt.xlabel("Frequency(HZ)")
plt.ylabel("Magnitude")
plt.xlim(0,50)
plt.grid(1)
plt.show()
```

Output:



Discussion:

Signal Sampling: We create a time vector `t` using `np.linspace` with a sample rate of 1000 Hz and a duration of 1 second.

Signal Construction: The signal `f` is constructed by summing the given sine waves with their respective amplitudes and frequencies.

FFT Computation: The FFT of the signal is computed using `np.fft.fft`, which transforms the signal from the time domain to the frequency domain.

Frequency Bins: `np.fft.fftfreq` is used to get the frequency bins corresponding to the FFT output.

Magnitude Calculation: The magnitude of each frequency component is calculated using `np.abs`.

Spectrum Plotting: We plot the magnitude against the frequency. Only the positive frequencies are plotted as the FFT output is symmetric.

This code will produce a plot showing the spectrum of the signal, where the x-axis represents the frequency and the y-axis represents the magnitude of each frequency component. The plot will help visualize the different frequency components present in the signal.

EXPERIMENT -02:

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

Theory:

The Discrete Fourier Transform (DFT) and its inverse (IDFT) are fundamental tools in digital signal processing for analysing the frequency content of discrete signals.

Discrete Fourier Transform (DFT): The DFT transforms a sequence of complex numbers (x[n]) into another sequence of complex numbers (X[k]), representing the signal in the frequency domain. The DFT is defined by the formula:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j2\pi ft/N}$$

where: -

N is the number of samples.

k is the frequency index.

n is the time index.

j is the imaginary unit.

Inverse Discrete Fourier Transform (IDFT): The IDFT transforms the sequence of complex numbers (X[k]) back into the time domain sequence (x[n]). The IDFT is defined by the formula:

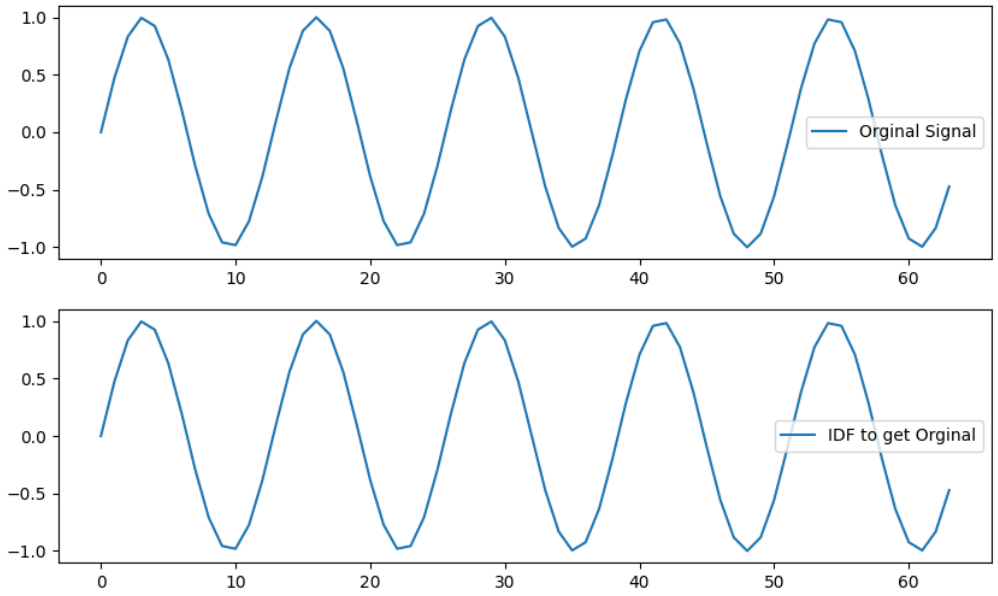
$$x[n] = \sum_{n=0}^{N-1} X[k] e^{j2\pi ft/N}$$

Python Program:

```
import numpy as np

import matplotlib.pyplot as plt

def DFT(x):

    N = len(x)

    X = np.zeros(N,dtype=complex)

    for k in range(N):
```

```
        for n in range(N):
            X[k] += x[n]*np.exp(-2j*np.pi*n*k/N)
    return X
def IDFT(X):
    N = len(X)
    x = np.zeros(N,dtype=complex)
    for k in range(N):
        for n in range(N):
            x[n] += X[k]*np.exp(2j*np.pi*n*k/N)
    return x/N
N = 64
t = np.arange(N)
freq = 5
x = np.sin(2*np.pi*freq*t/N)
X = DFT(x)
get_x = IDFT(X)
plt.figure(figsize=(10,6))
plt.subplot(2,1,1)
plt.plot(t,x,label="Orginal Signal")
plt.legend()
plt.subplot(2,1,2)
plt.plot(t,get_x.real,label="IDF to get Orginal")
plt.legend()
plt.show()
```

Output:



Discussion:

DFT and IDFT Functions: We define `DFT(x)` to compute the Discrete Fourier Transform manually by iterating over all indices and summing the complex exponentials.

Similarly, `IDFT(X)` computes the Inverse Discrete Fourier Transform by iterating and summing the complex exponentials, followed by normalization by (N).

Sample Signal: We generate a sample sine wave signal with a specified frequency.

DFT and IDFT Computation: We compute the DFT of the signal to transform it into the frequency domain. We then compute the IDFT to reconstruct the original signal from the frequency domain.

Plotting: We plot the original and reconstructed signals to visually verify the accuracy of the transformations.

This code demonstrates the process of transforming a signal from the time domain to the frequency domain and back, verifying the correctness of our manual implementation using Python's built-in functions.

EXPERIMENT -03:

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

Theory:

Sampling, Quantization, and Coding. Each of these steps is fundamental in converting an analog signal into a digital form.

1. **Sampling**: This is the process of converting a continuous-time signal into a discrete-time signal by taking samples at regular intervals.
2. **Quantization**: This step involves mapping the sampled signal to discrete levels.
3. **Coding**: This step converts the quantized values into binary code.

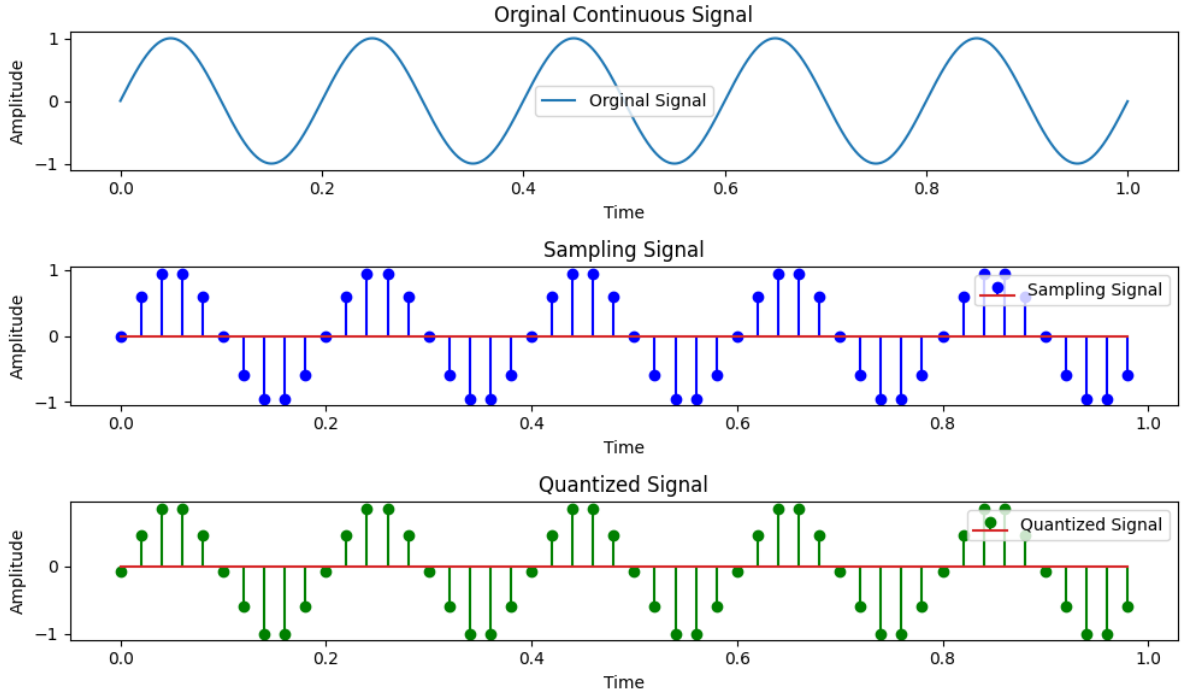
Python Program:

```
import numpy as np

import matplotlib.pyplot as plt

frequency = 5

amplitude = 1

duration = 1

sample_rate = 50

#Orginal Signal

orginal_time =
np.linspace(0,duration,int(duration*sample_rate*100),endpoint=False)

Orginal_amplitude = amplitude*np.sin(2*np.pi*frequency*orginal_time)

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

plt.subplot(3,1,1)

plt.plot(orginal_time,Orginal_amplitude,label="Orginal Signal")

plt.title("Orginal Continuous Signal")

plt.xlabel("Time")
```

```
plt.ylabel("Amplitude")
plt.legend()
#Sampling
sample_time =
np.linspace(0,duration,int(duration*sample_rate),endpoint=False)
sample_amplitude = amplitude*np.sin(2*np.pi*frequency*sample_time)
plt.subplot(3,1,2)
plt.stem(sample_time,sample_amplitude,'b',label="Sampling Signal")
plt.title("Sampling Signal")
plt.xlabel("Time")
plt.ylabel("Amplitude")
plt.legend()
#Quantized
num_level = 16
quantized_value = np.linspace(-amplitude,amplitude,num_level)
quantized_index = np.digitize(sample_amplitude,quantized_value)-1
quantized = quantized_value[quantized_index]
plt.subplot(3,1,3)
plt.stem(sample_time,quantized,'g',label="Quantized Signal")
plt.title("Quantized Signal")
plt.xlabel("Time")
plt.ylabel("Amplitude")
plt.legend()
#Coding
def decimal_to_binary(n,bits):
    return bin(n)[2:].zfill(bits)
bits_per_sample = int(np.ceil(np.log2(num_level)))
x_encoded = [decimal_to_binary(index,bits_per_sample) for index in
quantized_index]
```

```
print("Quantized Amplitudes and Corresponding Binary Codes:")
for i,code in enumerate(x_encoded):
    print(f'Sample {i} : Amplitude = {quantized[i]:.2f}, Binary Code: {code}')
plt.tight_layout()
plt.show()
```

Output:



Quantized Amplitudes and Corresponding Binary Codes:

Sample 0 : Amplitude = -0.07, Binary Code: 0111

Sample 1 : Amplitude = 0.47, Binary Code: 1011

Sample 2 : Amplitude = 0.87, Binary Code: 1110

Sample 3 : Amplitude = 0.87, Binary Code: 1110

Sample 4 : Amplitude = 0.47, Binary Code: 1011

Sample 5 : Amplitude = -0.07, Binary Code: 0111

Sample 6 : Amplitude = -0.60, Binary Code: 0011

Sample 7 : Amplitude = -1.00, Binary Code: 0000

Sample 8 : Amplitude = -1.00, Binary Code: 0000

Sample 9 : Amplitude = -0.60, Binary Code: 0011

Sample 10 : Amplitude = -0.07, Binary Code: 0111

Sample 11 : Amplitude = 0.47, Binary Code: 1011

Sample 12 : Amplitude = 0.87, Binary Code: 1110

Sample 13 : Amplitude = 0.87, Binary Code: 1110

Sample 14 : Amplitude = 0.47, Binary Code: 1011

Sample 15 : Amplitude = -0.07, Binary Code: 0111

Sample 16 : Amplitude = -0.60, Binary Code: 0011

Sample 17 : Amplitude = -1.00, Binary Code: 0000

Sample 18 : Amplitude = -1.00, Binary Code: 0000

Sample 19 : Amplitude = -0.60, Binary Code: 0011

Sample 20 : Amplitude = -0.07, Binary Code: 0111

Sample 21 : Amplitude = 0.47, Binary Code: 1011

Sample 22 : Amplitude = 0.87, Binary Code: 1110

Sample 23 : Amplitude = 0.87, Binary Code: 1110

Sample 24 : Amplitude = 0.47, Binary Code: 1011

Sample 25 : Amplitude = -0.07, Binary Code: 0111

Sample 26 : Amplitude = -0.60, Binary Code: 0011

Sample 27 : Amplitude = -1.00, Binary Code: 0000

Sample 28 : Amplitude = -1.00, Binary Code: 0000

Sample 29 : Amplitude = -0.60, Binary Code: 0011

Sample 30 : Amplitude = -0.07, Binary Code: 0111

Sample 31 : Amplitude = 0.47, Binary Code: 1011

Sample 32 : Amplitude = 0.87, Binary Code: 1110

Sample 33 : Amplitude = 0.87, Binary Code: 1110

Sample 34 : Amplitude = 0.47, Binary Code: 1011

Sample 35 : Amplitude = -0.07, Binary Code: 0111

Sample 36 : Amplitude = -0.60, Binary Code: 0011

Sample 37 : Amplitude = -1.00, Binary Code: 0000

Sample 38 : Amplitude = -1.00, Binary Code: 0000

Sample 39 : Amplitude = -0.60, Binary Code: 0011

Sample 40 : Amplitude = -0.07, Binary Code: 0111

Sample 41 : Amplitude = 0.47, Binary Code: 1011

Sample 42 : Amplitude = 0.87, Binary Code: 1110

Sample 43 : Amplitude = 0.87, Binary Code: 1110

Sample 44 : Amplitude = 0.47, Binary Code: 1011

Sample 45 : Amplitude = -0.07, Binary Code: 0111

Sample 46 : Amplitude = -0.60, Binary Code: 0011

Sample 47 : Amplitude = -1.00, Binary Code: 0000

Sample 48 : Amplitude = -1.00, Binary Code: 0000

Sample 49 : Amplitude = -0.60, Binary Code: 0011

Discussion:

Signal Generation: We generate a continuous-time sine wave signal with a specified frequency and amplitude.

Sampling: We sample the continuous signal at a specified sampling rate to get a discrete-time signal.

Quantization: We define a number of quantization levels. We map each sampled value to the nearest quantization level using **np.digitize**.

Coding: We convert each quantized value into a binary code using a custom function **decimal_to_binary**.

Plotting: We plot the continuous signal, sampled signal, and quantized signal for visualization.

Display Coding: We print out the binary codes for each quantized value.

This program demonstrates the process of converting an analog signal into a digital signal through sampling, quantization, and coding.

EXPERIMENT -04:

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

Theory:

Convolution: Convolution is a mathematical operation used to express the relationship between three functions. In signal processing, it's commonly used to combine two sequences to form a third sequence, showing how the shape of one sequence is modified by the other. For discrete signals, the convolution of two sequences x[n] and h[n] is defined as:

$$y[n] = x[n]*h[n]$$

Correlation: Correlation is a measure of similarity between two sequences as a function of the time-lag applied to one of them. The cross-correlation of two sequences x[n] and h[n] is defined as:

$$R(t) = \int_{-\infty}^{\infty} x_1(t)x_2(t-T)dt$$

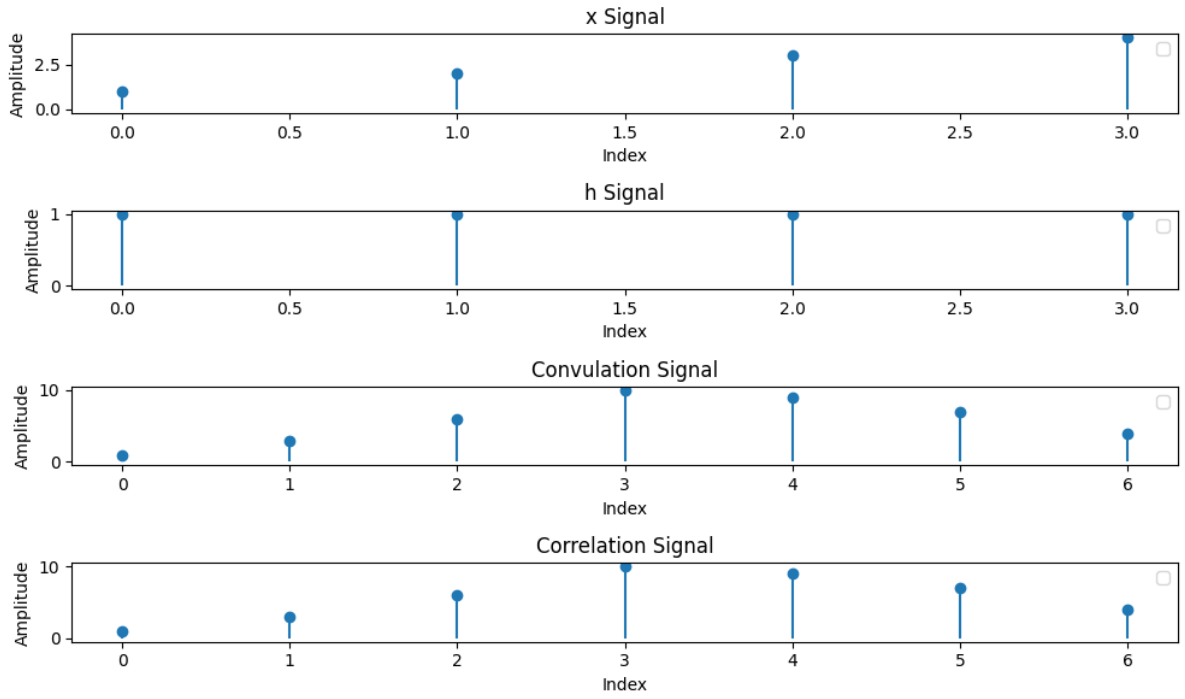
Correlation is often used to find patterns or matches between two signals.

Python Program:

```
import numpy as np

import matplotlib.pyplot as plt

x = np.array([1,2,3,4])

h = np.array([1,1,1,1])

Convulation = np.convolve(x,h,mode='full')

Correlation = np.correlate(x,h,mode='full')

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

plt.subplot(4,1,1)

plt.title("x Signal")

plt.stem(x,basefmt=" ")

plt.xlabel("Index")

plt.ylabel("Amplitude")
```

```
plt.legend()
plt.subplot(4,1,2)
plt.title("h Signal")
plt.stem(h,basefmt=" ")
plt.xlabel("Index")
plt.ylabel("Amplitude")
plt.legend()
plt.subplot(4,1,3)
plt.title("Convulation Signal")
plt.stem(Convulation,basefmt=" ")
plt.xlabel("Index")
plt.ylabel("Amplitude")
plt.legend()
plt.subplot(4,1,4)
plt.title("Correlation Signal")
plt.stem(Correlation,basefmt=" ")
plt.xlabel("Index")
plt.ylabel("Amplitude")
plt.legend()
print(f"Convulation Result: {Convulation}")
print(f"Correlation Result: {Correlation}")
plt.tight_layout()
plt.show()
```

Output:



Convolution Result: [1 3 6 10 9 7 4]

Correlation Result: [1 3 6 10 9 7 4]

Discussion:

Convolution Result: The convolution result combines the effect of both sequences. For example, in signal processing, if **x** is an input signal and **h** is an impulse response, their convolution represents the output signal of the system.

Correlation Result: The correlation result shows how similar the two sequences are at various time-lags. A peak in the correlation result indicates a high degree of similarity at that particular lag. This is useful in applications like signal matching and pattern recognition.

This program demonstrates the fundamental operations of convolution and correlation, which are essential tools in digital signal processing and various other fields involving discrete data analysis.

EXPERIMENT -05:

Write a python program to display the following region of a speech signal: i) Voiced region, ii) Unvoiced region, iii) Silence region.

Theory:

To display the different regions (voiced, unvoiced, and silence) of a speech signal, we'll follow these steps:

1. Load the speech signal.
2. Extract features that help distinguish between voiced, unvoiced, and silence regions.
3. Classify and segment the regions.
4. Plot the signal highlighting these regions.

For this task, we can use the 'librosa' library for loading and processing the audio signal, and matplotlib for plotting.

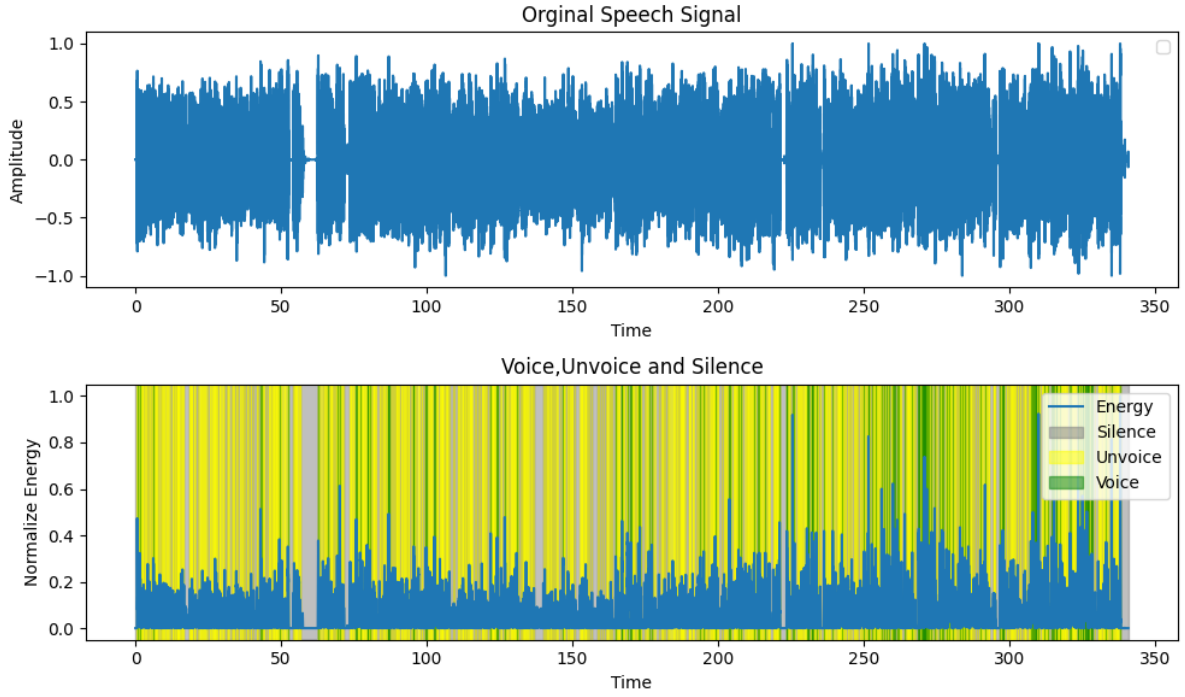
Python Program:

```
import librosa

import numpy as np

import matplotlib.pyplot as plt

file_path = r"C:\Users\SHUVO\Desktop\AlgoBangla29\948A.wav"

signal,sr = librosa.load(file_path,sr=None)

time = np.linspace(0,len(signal)/sr,num=len(signal))

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

plt.subplot(2,1,1)

plt.plot(time,signal)

plt.title("Orginal Speech Signal")

plt.xlabel("Time")

plt.ylabel("Amplitude")
```

```
plt.legend()
def classify_region(singnal,sr,frame_length=2048,hop_length=512):
    energy=np.array([
        np.sum(np.abs(signal[i:i+frame_length]**2))
        for i in range(0,len(signal),hop_length)
    ])
    #Normalize
    energy = (energy-np.min(energy)) / (np.max(energy)-np.min(energy))
    silence = 0.1
    unvoice = 0.3
    region = np.zeros_like(energy)
    region[energy>unvoice] = 2 #Voice
    region[(energy<=unvoice) & (energy>silence)] = 1 #Unvoice
    region[energy<=silence] = 0 #Silence
    return region,energy
region,energy = classify_region(signal,sr)
frame_time = np.linspace(0,len(signal)/sr,num=len(energy))
plt.subplot(2,1,2)
plt.plot(frame_time,energy,label="Energy")
plt.fill_between(frame_time,0,1,where=region==0,
color="gray",alpha=0.5,transform=plt.gca().get_xaxis_transform(),label="Silence")
plt.fill_between(frame_time,0,1,where=region==1,
color="yellow",alpha=0.5,transform=plt.gca().get_xaxis_transform(),label="Unvoice")
plt.fill_between(frame_time,0,1,where=region==2,
color="green",alpha=0.5,transform=plt.gca().get_xaxis_transform(),label="Voice")
plt.title("Voice,Unvoice and Silence")
plt.xlabel("Time")
plt.ylabel("Normalize Energy")
```

```
plt.legend(loc='upper right')
plt.tight_layout()
plt.show()
```

Output:



Discussion:

Loading the signal: We use **librosa.load** to load the audio signal.

Classifying regions: We compute the short-time energy of the signal to differentiate between voiced, unvoiced, and silence regions. Based on energy thresholds, we classify frames as silence, unvoiced, or voiced.

Plotting: The original speech signal is plotted in the first subplot. The energy and classified regions are plotted in the second subplot, with different colors representing different regions.

EXPERIMENT -06:

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

Theory:

Autocorrelation is a mathematical tool used in signal processing to measure the similarity between a signal and a delayed version of itself over varying time lags. It helps in identifying repeating patterns or periodicities within a signal, which is particularly useful in speech processing for tasks such as pitch detection.

Short-Term Autocorrelation involves dividing the signal into short overlapping frames and computing the autocorrelation for each frame separately. This approach is necessary because speech signals are non-stationary, meaning their statistical properties change over time. By analysing short segments, we can capture these time-varying properties.

Steps to Compute Short-Term Autocorrelation:

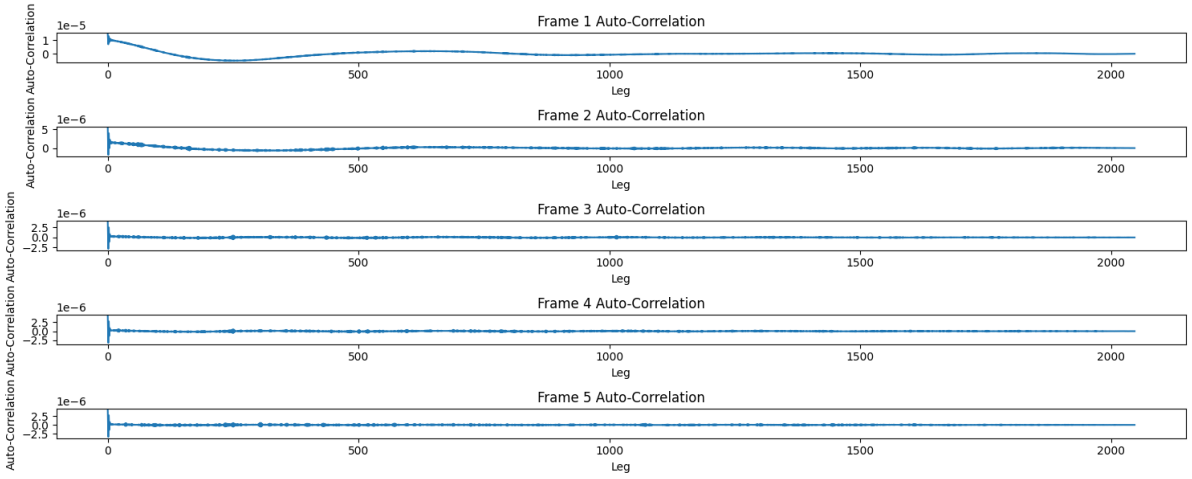
1. **Frame the signal**: Divide the signal into overlapping frames.
2. **Windowing**: Apply a window function (e.g., Hamming window) to each frame to minimize spectral leakage.
3. **Compute autocorrelation**: Calculate the autocorrelation for each windowed frame.
4. **Plot and analyses**: Visualize the autocorrelation to identify patterns.

Python Program:

```
import librosa

import numpy as np

import matplotlib.pyplot as plt

file_path = r"C:\Users\SHUVO\Desktop\AlgoBangla29\948A.wav"

signal,sr = librosa.load(file_path,sr=None)

def auto_correlation(signal,frame_length=2048,pop_length=512):

    autocor=[]
```

```
    for i in range(0,len(signal)-frame_length,pop_length):
        frame = signal[i:i+frame_length]
        frame_autoco = np.correlate(frame,frame,mode="full")
        autocor.append(frame_autoco[frame_length-1:])
    return np.array(autocor)
autoCorrelation = auto_correlation(signal)
plt.figure(figsize=(15,6))
for i in range(min(5,len(autoCorrelation))):
    plt.subplot(5,1,i+1)
    plt.title(f"Frame {i+1} Auto-Correlation")
    plt.plot(autoCorrelation[i])
    plt.xlabel("Leg")
    plt.ylabel("Auto-Correlation")
plt.tight_layout()
plt.show()
```

Output:



Discussion:

Frame Length and Hop Length:

- **Frame Length**: This determines the size of each segment of the signal that is analysed. A typical value is around 20-40 ms for speech signals. In this example, we use 2048 samples, which is approximately 46 ms at a 44.1 kHz sampling rate.
- **Hop Length**: This determines the overlap between consecutive frames. A typical value is half the frame length. In this example, we use 512 samples, which provides significant overlap and smoothest the autocorrelation results.

Windowing:

- **Hamming Window**: Applying a window function like the Hamming window reduces spectral leakage by tapering the edges of the frames. This is crucial for obtaining accurate autocorrelation results.

Autocorrelation:

- **Non-negative Lags**: Only the second half of the autocorrelation result is retained, corresponding to non-negative lags. This is because the autocorrelation function is symmetric.

Visualization:

- **Interpreting Plots**: The plots show how the signal's similarity varies with lag for different frames. Peaks in the autocorrelation plots correspond to periodic components in the signal, which can indicate the presence of fundamental frequencies (pitch) in voiced speech.

EXPERIMENT -07:

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 the sequence $x(n)$={1,2,3,4,5,6,7,6,5,4,3,2,1} we need to determine and plot the sequence $y(n)$.

Sequence Shifting: $x(n-5)$, this shifts the sequence x(*n*) 5 steps to the right. x(*n*+4), this shifts the sequence x(*n*) 4 steps to the left.

Scaling: Multiply the shifted sequence x(*n*−5) by 2. Multiply the shifted sequence x(*n*+4) by -3.

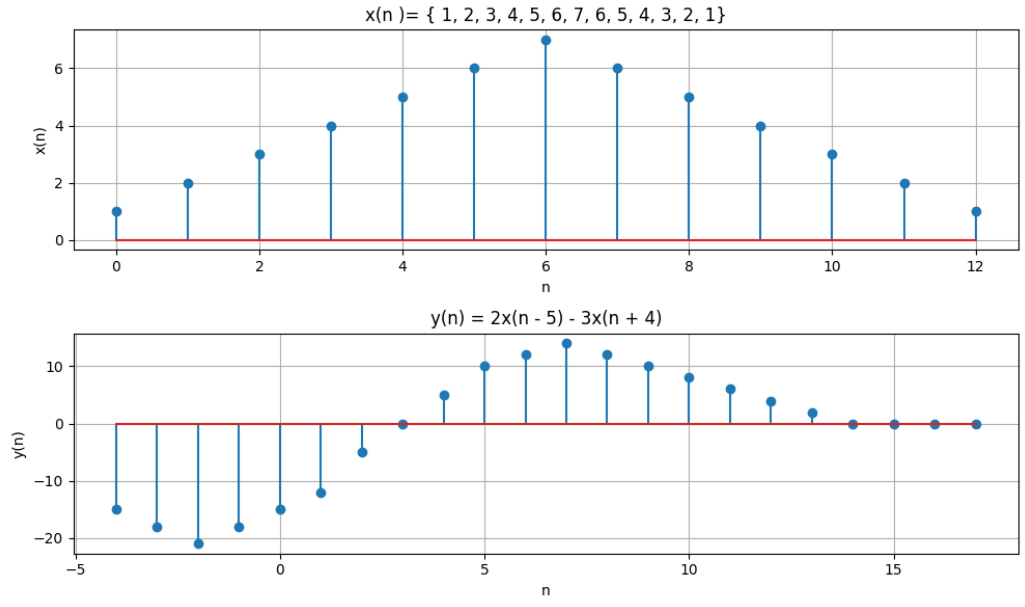
Combining: Combine the scaled and shifted sequences to get $y(n)$.

Python Program:

```
import numpy as np
import matplotlib.pyplot as plt
# x(n )= { 1, 2, 3, 4, 5, 6, 7, 6, 5, 4, 3, 2, 1}
x_n = np.array([1,2,3,4,5,6,7,6,5,4,3,2,1])
n_x = np.arange(0,len(x_n))
plt.figure(figsize=(10,6))
plt.subplot(2,1,1)
plt.stem(n_x,x_n)
plt.title('x(n )= { 1, 2, 3, 4, 5, 6, 7, 6, 5, 4, 3, 2, 1}')
plt.xlabel('n')
plt.ylabel('x(n)')
plt.grid(True)
n_y = np.arange(-4,len(x_n)+5)
y_n = np.zeros_like(n_y,dtype=float)
```

```
#y(n)=2x(n − 5) − 3x(n+4)
for i in range(len(n_y)):
    shift_minus5 = i-5
    shift_plus4 = i+4
    if 0<shift_minus5<len(x_n):
        y_n[i]+=2*x_n[shift_minus5]
    if 0<shift_plus4<len(x_n):
        y_n[i]-=3*x_n[shift_plus4]
plt.subplot(2,1,2)
plt.stem(n_y,y_n)
plt.title('y(n) = 2x(n - 5) - 3x(n + 4)')
plt.xlabel('n')
plt.ylabel('y(n)')
plt.grid(True)
plt.tight_layout()
plt.show()
```

Output:



Discussion:

Sequence Shifting:

- **Right Shift x(n - 5)**: Shifting $x(n)$ to the right by 5 positions aligns the sequence such that the value at position n moves to $n+5$.
- **Left Shift x(n + 4)**: Shifting $x(n)$ to the left by 4 positions aligns the sequence such that the value at position n moves to $n-4$.
- **Scaling and Combining**:
- **Scaling**: Each shifted sequence is scaled by a factor of 2 or -3, respectively.
- **Combining**: The scaled sequences are then summed to produce the final sequence $y(n)$.

Handling Range:

- To accommodate the shifts, the range of $y(n)$ is extended to ensure that the shifted indices fall within the valid range. Specifically, n is extended from -4 to $len(xn)+5$.

Plotting:

- **Plotting x(*n*)**: This provides a visual reference for the original sequence.
- **Plotting *y*(*n*)**: This shows the combined effect of shifting, scaling, and summing, making it clear how the original sequence transforms into $y(n)$.

By analysing these plots, we can visually understand the effect of the operations on the original sequence.

EXPERIMENT -08:

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

Theory:

Normalized Frequencies:

The normalized frequency (ω) is calculated as:

$$\omega p=2\pi fp/Fs,\ \omega s=2\pi fs/Fs$$

Ideal Impulse Response:

The ideal impulse response for a low-pass filter is given by the sinc function:

$$hideal[n]=\sin(\omega c(n-(N-1)/2))/\pi(n-(N-1)/2)$$

Here, ωc is the cutoff frequency, which we approximate by taking the average of the passband and stopband edges for simplicity:

$$\omega c = \omega p+\omega s/2 = 0.2\pi+0.5\pi/2 = 0.35\pi$$

The filter length N is 21, so the impulse response is centered around $(N-1)/2=10$.

Window Function:

The Hanning window is applied to the ideal impulse response to taper the edges, reducing the sidelobe levels in the frequency response:

$$w[n]=0.5-0.5\cos(2\pi n/N-1)$$

The final impulse response of the FIR filter is:

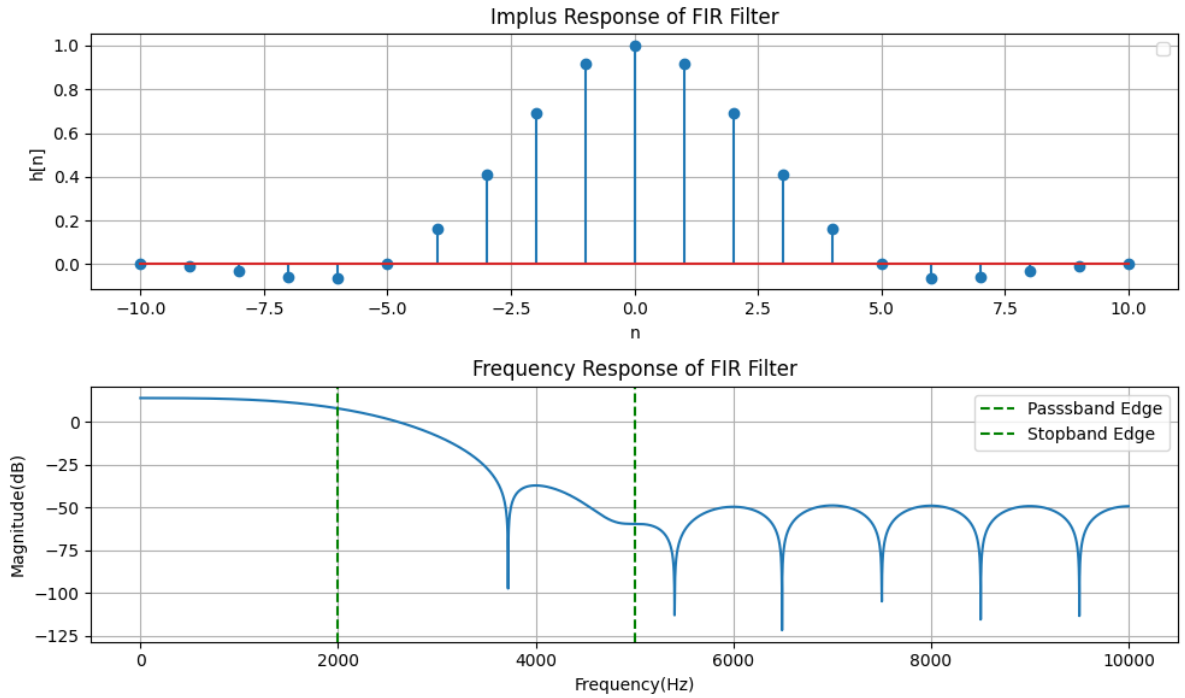
$$h[n]=hideal[n]\cdot w[n]$$

Python Program:

```
import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import freqz

fp = 2000
```

```
fs = 5000
Fs = 20000
N = 21
n = np.arange(N)-(N-1)/2
h_ideal = np.sinc((2*fp/Fs)*n)
hamming_window = np.hamming(N)
h = h_ideal*hamming_window
plt.figure(figsize=(10,6))
plt.subplot(2,1,1)
plt.stem(n,h)
plt.title("Implus Response of FIR Filter")
plt.xlabel("n")
plt.ylabel("h[n]")
plt.grid(True)
plt.legend()
W,H = freqz(h,worN=8000)
plt.subplot(2,1,2)
plt.plot(W/np.pi*(Fs/2),20*np.log10(np.abs(H)))
plt.title("Frequency Response of FIR Filter")
plt.xlabel("Frequency(Hz)")
plt.ylabel("Magnitude(dB)")
plt.grid(True)
plt.axvline(fp,color = "green",linestyle="--",label="Passsband Edge")
plt.axvline(fs,color = "green",linestyle="--",label="Stopband Edge")
plt.legend()
plt.tight_layout()
plt.show()
```

Output:



Discussion:

Impulse Response Calculation:

The **sinc** function is used to create the ideal low-pass filter response, which is centered around zero. The cutoff frequency ωc is the midpoint between the passband and stopband edges.

Window Application:

The Hanning window is applied to the ideal impulse response. This window reduces the sidelobes in the frequency response by tapering the filter coefficients smoothly to zero at the edges.

Frequency Response:

The frequency response plot shows the filter's attenuation in the passband and stopband regions. The transition band is smooth due to the Hanning window.

This approach provides a simple and effective method for designing FIR filters that meet the given specifications. The Hanning window helps to balance the trade-off between mainlobe width and sidelobe levels, providing a good compromise between passband ripple and stopband attenuation.

EXPERIMENT -09:

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:

Generate the Signal:

First, generate a time vector **t** with 100 samples at a sampling rate of 100 Hz. Construct the signal **s** by summing three sinusoidal components:

$$s(t)=\sin(2\pi\cdot 5\cdot t) +\sin(2\pi\cdot 15\cdot t) + \sin(2\pi\cdot 30\cdot t)$$

This creates a signal **s** with frequencies at 5 Hz, 15 Hz, and 30 Hz.

Design the IIR Filter:

To suppress frequencies at 5 Hz and 30 Hz, we'll design an IIR notch filter for each frequency. A notch filter is characterized by its center frequency ($f0$) and quality factor (Q). For each notch filter:

Compute the notch frequency ($f0$) as the frequency to be suppressed. Choose a suitable quality factor (Q) that determines the width of the notch. Design the IIR notch filter using methods like the bilinear transform or direct analog prototype designs.

Apply the Filter:

Apply the designed notch filters to the signal **s** using a filtering function (e.g., **scipy.signal.lfilter**).

Display the Results:

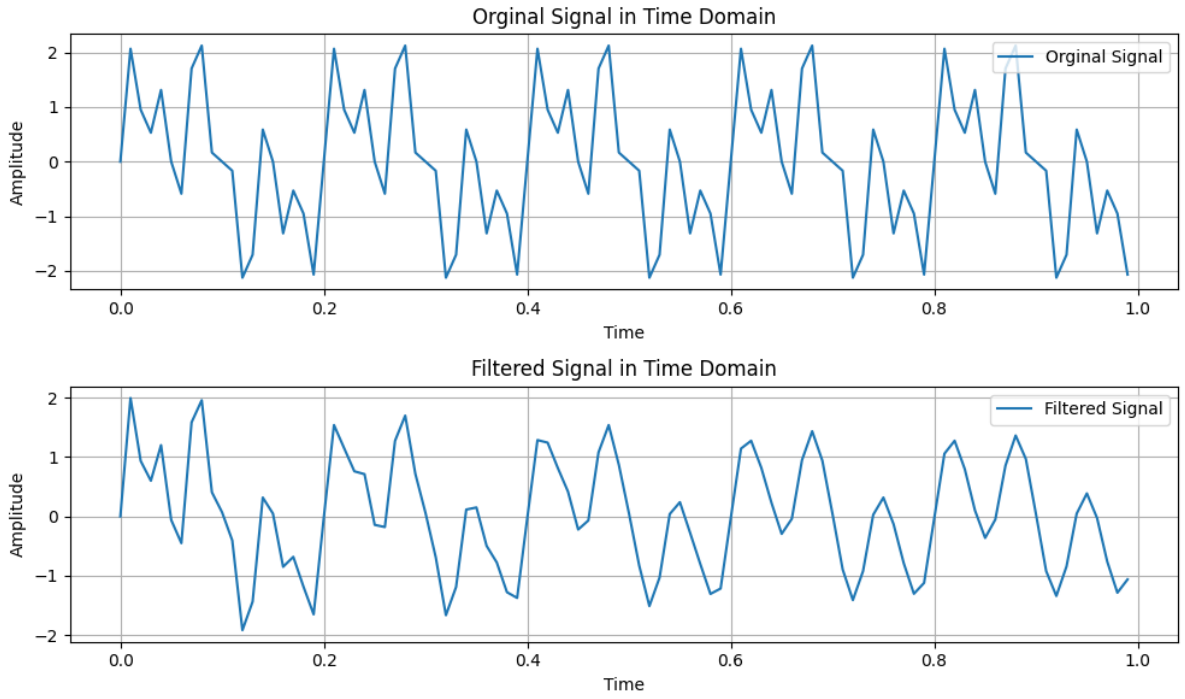
Plot the original signal **s** in the time domain. Plot the filtered signal to observe the suppression of frequencies at 5 Hz and 30 Hz.

Python Program:

```
import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import iirnotch,lfilter
```

```
fs = 100
t = np.arange(100)/fs
s = np.sin(2*np.pi*5*t)+np.sin(2*np.pi*15*t)+np.sin(2*np.pi*30*t)
plt.figure(figsize=(10,6))
plt.subplot(2,1,1)
plt.plot(t,s,label="Orginal Signal")
plt.title("Orginal Signal in Time Domain")
plt.xlabel("Time")
plt.ylabel("Amplitude")
plt.grid(True)
plt.legend()
f_notch1 = 5
f_notch2 = 30
Q = 30
b1,a1 = iirnotch(f_notch1,Q,fs)
b2,a2 = iirnotch(f_notch2,Q,fs)
s_filter = lfilter(b1,a1,s)
s_filter = lfilter(b2,a2,s_filter)
plt.subplot(2,1,2)
plt.plot(t,s_filter,label="Filtered Signal")
plt.title("Filtered Signal in Time Domain")
plt.xlabel("Time")
plt.ylabel("Amplitude")
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```

Output:



Discussion:

Signal Generation: The signal s is composed of three sinusoidal components at 5 Hz, 15 Hz, and 30 Hz, sampled at a rate of 100 Hz over 1 second.

IIR Notch Filters: Two IIR notch filters are designed using the iirnotch function from scipy.signal. Each notch filter is designed to suppress the specific frequencies of 5 Hz and 30 Hz with a quality factor Q of 10, which determines the width of the notch.

Filter Application: The lfilter function applies the notch filters sequentially to the original signal s. After applying both notch filters, s_filtered2 contains the signal with suppressed frequencies at both 5 Hz and 30 Hz.

Visualization: The results are visualized using matplotlib. Three subplots show: the original signal s, the signal after applying the notch filter for 5 Hz, the final filtered signal after applying notch filters for both 5 Hz and 30 Hz.

EXPERIMENT -10:

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

Theory:

Normalized Frequencies:

Convert the passband edge and transition width from Hz to normalized digital frequencies:

$$\omega p=2\pi \cdot fp/Fs,\ Wn=2\pi \cdot Wt/Fs$$

Filter Design:

Use the **scipy.signal.firwin** function to design the FIR filter:

python

```
h = firwin(N, omega_p / np.pi, window='blackman', fs=F_s)
```

This function designs a FIR filter of length N with a passband edge normalized frequency ($\omega p/\pi$) and applies a Blackman window to the coefficients.

Frequency Response:

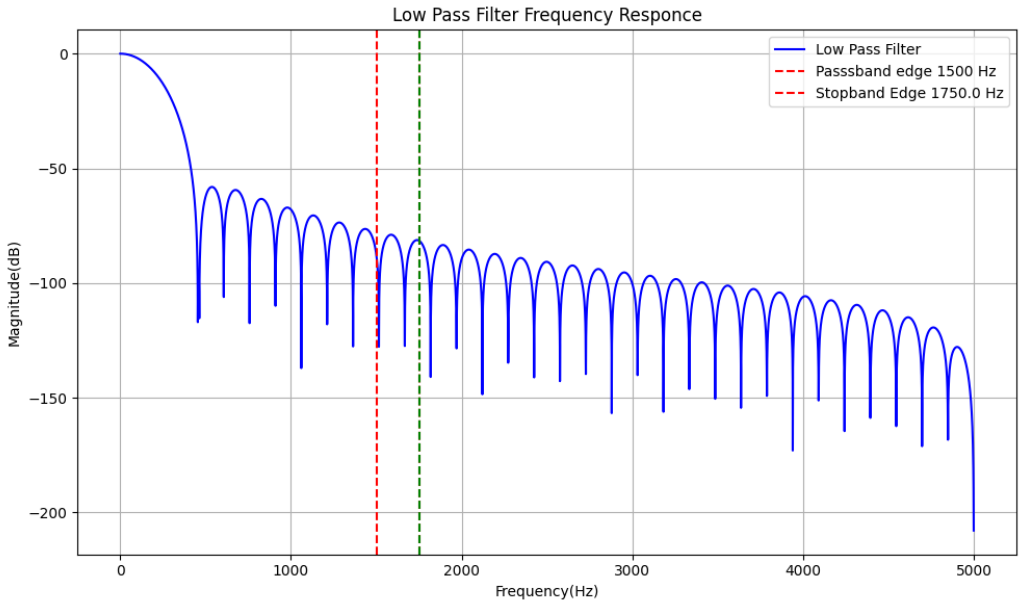
Compute and plot the frequency response of the designed filter to ensure it meets the desired specifications. The frequency response should show attenuation starting from the passband edge and continuing into the stopband.

Python Program:

```
import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import firwin,freqz

fp = 1500

wt = 500

Fs = 10000

N = 67
```

```
window = 'blackman'
Omega_P = 2*np.pi*fp/Fs
h = firwin(N,Omega_P/np.pi, window=window,fs=Fs)
W,H = freqz(h,worN=8000,fs=Fs)
plt.figure(figsize=(10,6))
plt.plot(W,20*np.log10(np.abs(H)),'b',label="Low Pass Filter")
plt.title("Low Pass Filter Frequency Responce")
plt.xlabel("Frequency(Hz)")
plt.ylabel("Magnitude(dB)")
plt.axvline(fp,color='red',linestyle='--',label=f"Passsband edge {fp} Hz")
plt.axvline(fp+wt/2,color='red',linestyle='--',label=f"Stopband Edge {fp+wt/2} Hz")
plt.axvline(fp+wt/2,color='green',linestyle='--')
plt.legend()
plt.grid()
plt.tight_layout()
plt.show()
```

Output:



Discussion:

Normalization: The passband edge frequency (fp) and transition width (Wt) are normalized to digital frequencies (ωp and Wn) by dividing by half the sampling frequency ($Fs/2$).

Filter Design: The **scipy.signal.firwin** function designs an FIR filter with a passband edge at $\omega p/\pi$, which corresponds to fp Hz. The Blackman window is applied to the filter coefficients to minimize sidelobe levels in the frequency response.

Frequency Response: The frequency response plot shows the magnitude response in decibels (dB) against frequency (in Hz). The red vertical line indicates the passband edge, and the green dashed lines indicate the edges of the stopband, defined by

$$fp \pm Wt/2 \text{ Hz.}$$

This approach effectively designs a lowpass FIR filter with a Blackman window to meet specific frequency domain specifications, ensuring smooth transition from passband to stopband with minimal ripple and attenuation characteristics as required. Adjusting the filter length N and the window type provides flexibility in achieving desired filter performance.

EXPERIMENT -11:

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:

Normalize Frequencies: Normalize the passband and stopband edge frequencies to the range [0,0.5], where 0.5 corresponds to the Nyquist frequency ($fs/2$).

Filter Design: Design the bandpass filter using appropriate methods such as windowed sinc, FIR filter design functions (scipy.signal.firwin), or other design techniques.

Windowing: Apply a suitable window function (e.g., Hamming, Blackman, etc.) to the filter coefficients to minimize sidelobes and improve frequency response characteristics.

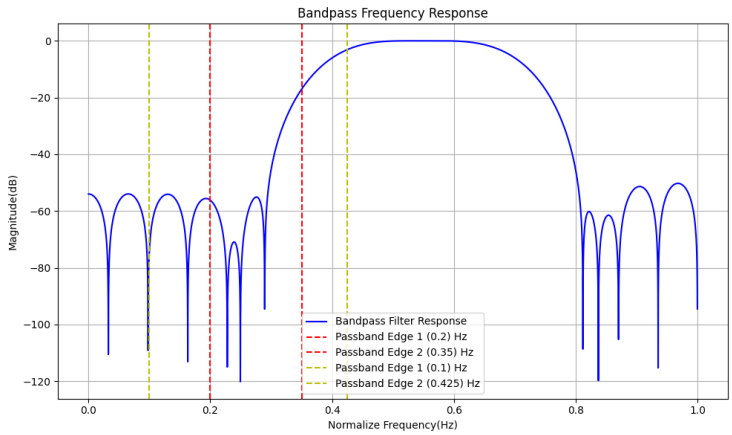
Plot the Frequency Response: Plot the magnitude and phase response of the designed filter to verify that it meets the desired specifications in the frequency domain.

Python Program:

```
import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import firwin,freqz

M = 32

fp1 = 0.2

fp2 = 0.35

fs1 = 0.1

fs2 = 0.425

normalize_fp1 = fp1

normalize_fp2 = fp2

normalize_fs1 = fs1
```

```
normalize_fs2 = fs2

h = firwin(M,[normalize_fp1,normalize_fp2],pass_zero=False,fs = 1.0,
window='hamming')

W,H = freqz(h,worN=8000)

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

plt.plot(W/np.pi,20*np.log10(np.abs(H)),'b',label = "Bandpass Filter Response")

plt.axvline(normalize_fp1,color="red",linestyle="--",label=f"Passband Edge 1
({fp1}) Hz")

plt.axvline(normalize_fp2,color="red",linestyle="--",label=f"Passband Edge 2
({fp2}) Hz")

plt.axvline(normalize_fs1,color="y",linestyle="--",label=f"Passband Edge 1 ({fs1})
Hz")

plt.axvline(normalize_fs2,color="y",linestyle="--",label=f"Passband Edge 2 ({fs2})
Hz")

plt.title("Bandpass Frequency Response")

plt.xlabel("Normalize Frequency(Hz)")

plt.ylabel("Magnitude(dB)")

plt.legend()

plt.grid(True)

plt.tight_layout()

plt.show()
```

Output:



Discussion:

Normalization: The given frequencies $fp1, fp2, fs1, fs2$ are normalized by dividing by the Nyquist frequency (0.5). This ensures they are within the valid range of [0,1] for FIR filter design.

Filter Design: The scipy.signal.firwin function designs the bandpass filter with a length M=32 using the Hamming window (window='hamming'). Passband edges are specified by the list [normalized_fp1, normalized_fp2], and pass_zero=False ensures it's a bandpass filter.

Frequency Response: The plotted frequency response shows the magnitude response in decibels (dB) against normalized frequency (ω/π). Red dashed lines mark the passband edges, and green dashed lines mark the stopband edges to visually confirm the filter's characteristics.

EXPERIMENT -12:

Use a Python program to determine and show the ''poles‖'' ''zeros‖'' and also ''roots‖'' of the following systems-

a) 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 and Zeros: Zeros of a transfer function $H(s)$ are the values of s that make the numerator equal to zero. Poles of a transfer function $H(s)$ are the values of s that make the denominator equal to zero.

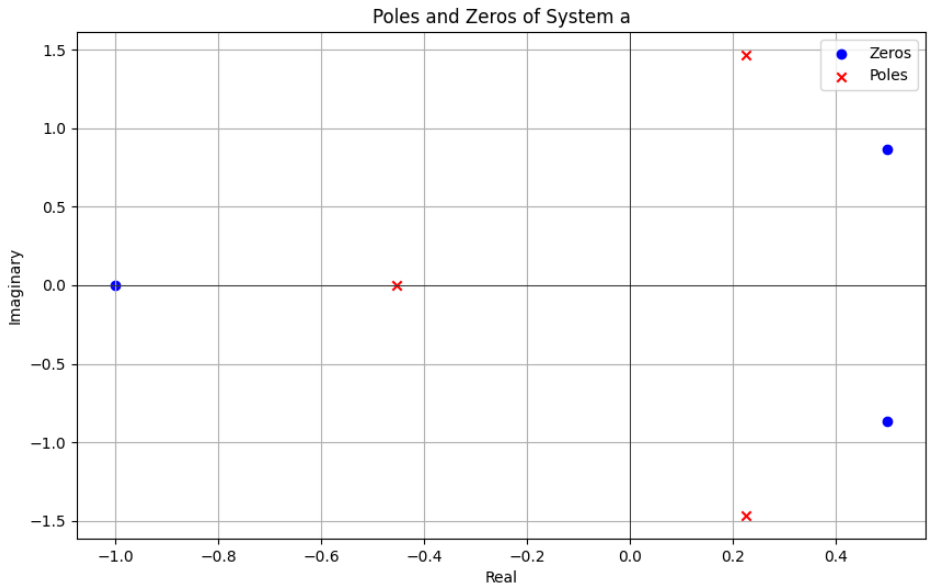
Roots: The roots of a polynomial are the values of the variable that make the polynomial equal to zero. In the context of transfer functions, these are the poles and zeros.

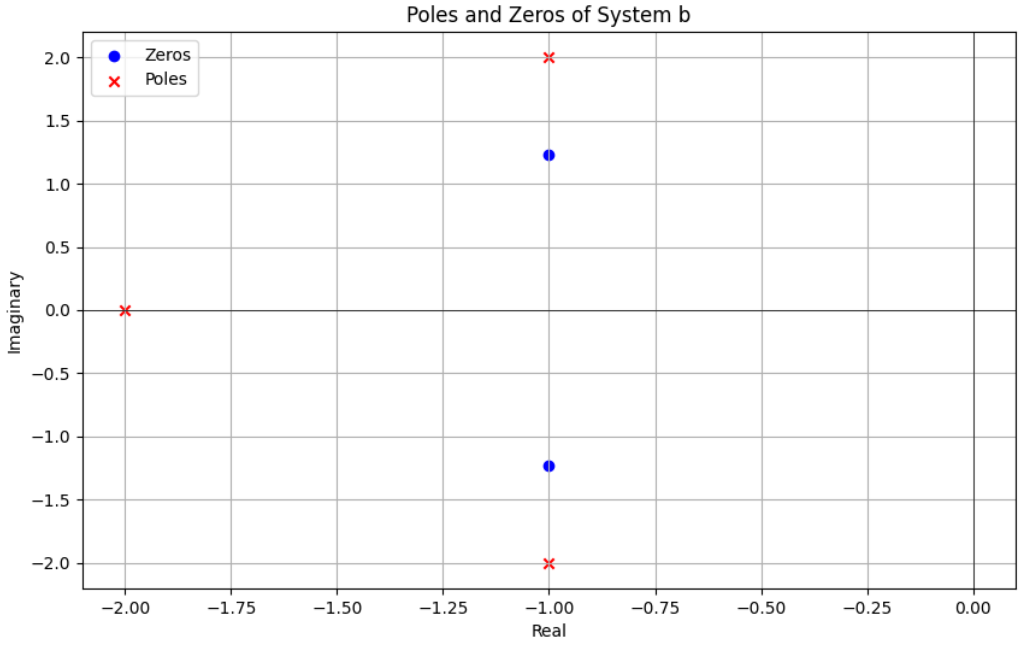
Python Program:

```
import numpy as np
import matplotlib.pyplot as plt
def plot_pole_zero(num,deno,title):
    zeros = np.roots(num)
    poles = np.roots(deno)
    plt.figure(figsize=(10,6))
    plt.scatter(np.real(zeros),np.imag(zeros),color='blue',marker='o',label="Zero
s")
    plt.scatter(np.real(poles),np.imag(poles),color='red',marker='x',label="Poles"
)
    plt.title(title)
    plt.axhline(0,color='black',lw=0.5)
    plt.axvline(0,color='black',lw=0.5)
    plt.grid(True)
    plt.xlabel("Real")
```

```
    plt.ylabel("Imaginary")
    plt.legend()
    plt.show()
    return zeros,poles
num_a = [1,0,0,1]
deno_a = [1,0,2,1]
num_b = [4,8,10]
deno_b = [2,8,18,20]
zero_a,pole_a = plot_pole_zero(num_a,deno_a,'Poles and Zeros of System a')
zero_b,pole_b = plot_pole_zero(num_b,deno_b,'Poles and Zeros of System b')
print("Sytem A:")
print("Zeros: ",zero_a)
print("Poles: ",pole_a)
print("Sytem B:")
print("Zeros: ",zero_a)
print("Poles: ",pole_b)
```

Output:





Sytem A:

Zeros: [-1. +0.j 0.5+0.8660254j 0.5-0.8660254j]

Poles: [0.22669883+1.46771151j 0.22669883-1.46771151j -0.45339765+0.j]

Sytem B:

Zeros: [-1. +0.j 0.5+0.8660254j 0.5-0.8660254j]

Poles: [-1.+2.j -1.-2.j -2.+0.j]

Discussion:

System a:

$$\mathbf{H(s) = \frac{s^3+1}{s^4+2s^2+1}}$$

Numerator (Zeros): $s^3 + 1$

Roots (Zeros): The roots of $s3+1=0$ are the zeros of the system.

Denominator (Poles): $s^4 + 2s^2 + 1$

Roots (Poles): The roots of $s^4 + 2s^2 + 1=0$ are the poles of the system.

System b:

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

Numerator (Zeros): $\boldsymbol{4s^2 + 8s + 10}$

Roots (Zeros): The roots of $\boldsymbol{4s^2 + 8s + 10}$ =0 are the zeros of the system.

Denominator (Poles): $\boldsymbol{2s^3 + 8s^2 + 18s + 20}$

Roots (Poles): The roots of $\boldsymbol{2s^3 + 8s^2 + 18s + 20}$=0 are the poles of the system.

Visualization

- **Poles** are marked with red crosses (x).
- **Zeros** are marked with blue circles (o).
- The plot shows these points on the complex plane, with the real part on the x-axis and the imaginary part on the y-axis.

This approach provides a clear graphical representation of the poles and zeros, allowing for analysis of system stability and frequency response characteristics. Poles in the right half of the complex plane indicate instability, while zeros impact the frequency response but not stability directly.