Experiment No: 01

Name of the Experiment: Write a Program to Sampling of a Sinusoidal Signal and Reconstruction of Analog Signal.

Objectives:

(1) To convert a continuous-time signal into a discrete-time signal.

(2) To understand the process of sampling and reconstruction of a sinusoidal signal and to analyze the accuracy of the reconstructed signal compared to the original signal.

Theory:

Sampling:

Sampling is the process of converting a continuous-time signal into a discrete-time signal. In other words, it involves taking regular samples or measurements of an analog signal at specific time intervals. The rate at which these samples are taken is known as the sampling rate, typically measured in Hertz (Hz). The Nyquist-Shannon sampling theorem states that to accurately represent an analog signal, the sampling rate must be at least twice the highest frequency component of the signal. Sampling is a critical step in digitizing analog signals for processing, transmission, or storage.

Sinusoidal Signal: A sinusoidal signal is a type of continuous-time signal that has a waveform resembling a sine or cosine function. It is characterized by its amplitude, frequency, and phase.The equation for a general sinusoidal signal is given by:

$$x(t) = A * \sin(2\pi f\, t + \varphi)$$

Where:

A is the amplitude of the signal.

f is the frequency of the signal in Hertz (Hz).

t is time.

φ (phi) is the phase angle.

Analog Signal:

An analog signal is a continuous-time signal that varies smoothly and continuously over time. It can take on any value within a certain range and is not quantized. Analog signals are found in many natural phenomena, including real-world sounds, voltage levels in electronic circuits, and physical measurements. Analog signals are typically represented by continuous waveforms.

Reconstruction:

Reconstruction, in the context of signal processing, refers to the process of converting a discrete-time signal (sampled signal) back into a continuous-time analog signal. This is typically done using a reconstruction filter or interpolating algorithm. The goal of reconstruction is to obtain a continuous signal that closely approximates the original continuous analog signal before it was sampled. Proper reconstruction is important to avoid aliasing and loss of information in the sampled signal.

Methods:

Setup: We began by setting up the function generator to generate a sinusoidal signal at a known frequency and amplitude.

Sampling: Using an oscilloscope, we measured the original analog signal's frequency and amplitude. We then used an analog-to-digital converter to sample the signal at different frequencies and analyze the results.

Aliasing Experiment: To explore the phenomenon of aliasing, we gradually reduced the sampling frequency below the Nyquist rate, and we observed the effects on the reconstructed signal.

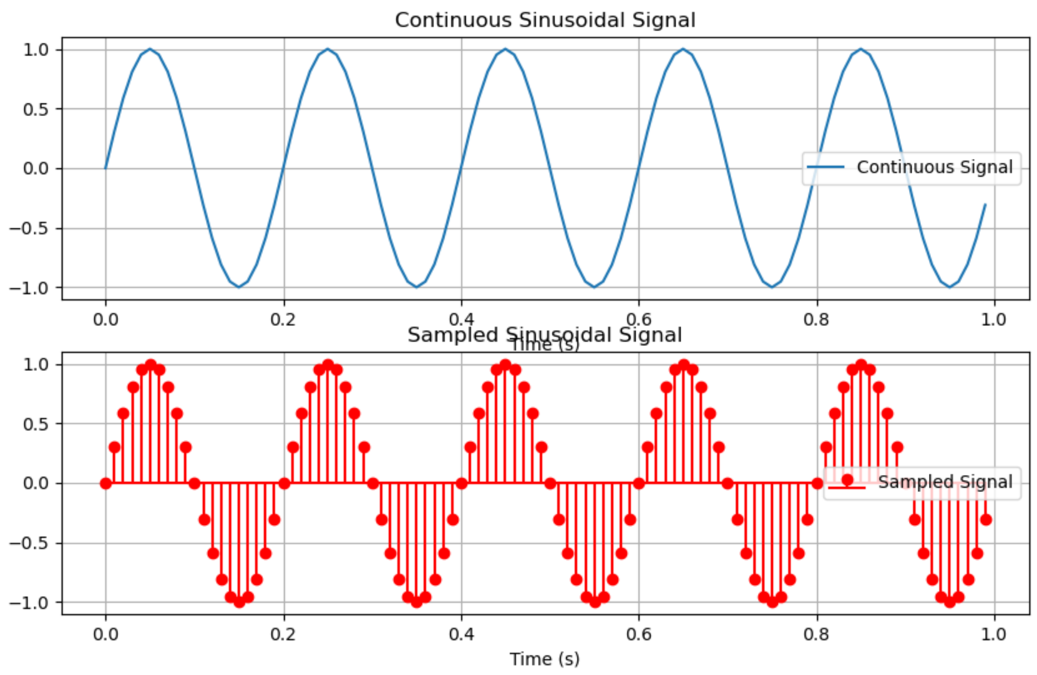
Reconstruction: After the aliasing experiment, we implemented the process of reconstructing the analog signal from the sampled data. This involved applying low-pass filters to the discrete signal.

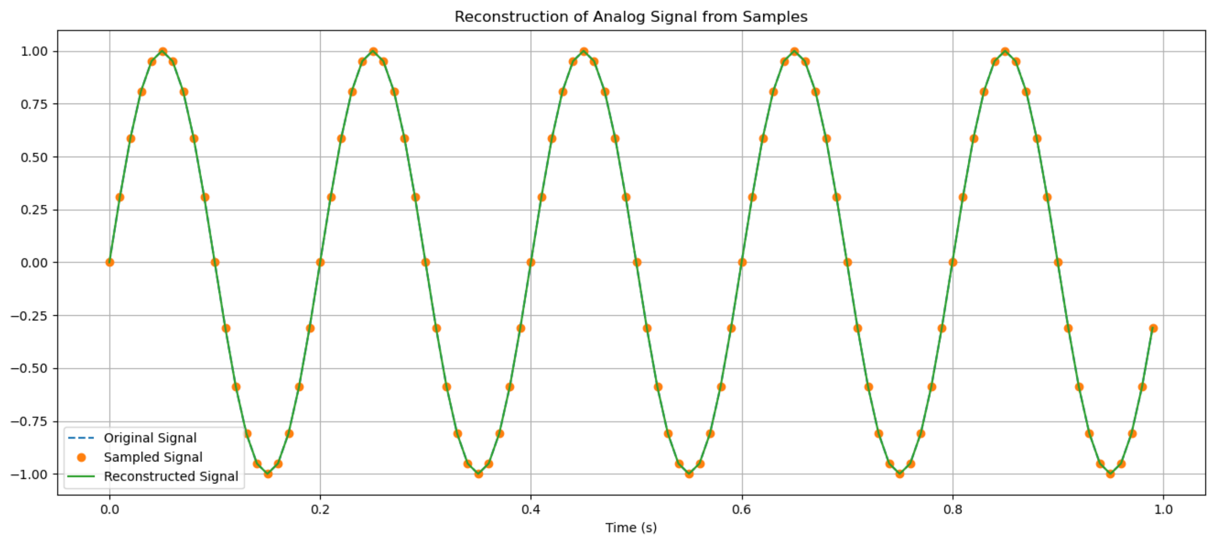
Data Analysis: We compared the original analog signal with the reconstructed signal and evaluated the accuracy of the reconstruction.

Source Code in Python:

```
import numpy as np

import matplotlib.pyplot as plt

# Parameters for the sinusoidal signal

amplitude = 1.0

frequency = 5.0  # Hz

sampling_frequency = 20.0 * frequency  # Must be greater than Nyquist rate (2
* frequency)

duration = 1.0  # seconds

# Time values for the continuous signal

t_continuous = np.linspace(0, duration, int(sampling_frequency * duration),
endpoint=False)

# Generate the continuous sinusoidal signal

continuous_signal = amplitude * np.sin(2 * np.pi * frequency * t_continuous)

# Time values for the discrete samples

t_samples = np.arange(0, duration, 1 / sampling_frequency)

# Sample the continuous signal

sampled_signal = amplitude * np.sin(2 * np.pi * frequency * t_samples)

# Plot the continuous and sampled signals

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

plt.subplot(2, 1, 1)

plt.plot(t_continuous, continuous_signal, label='Continuous Signal')

plt.title('Continuous Sinusoidal Signal')

plt.xlabel('Time (s)')

plt.grid(True)

plt.legend()
plt.subplot(2, 1, 2)
plt.stem(t_samples, sampled_signal, markerfmt='ro', linefmt='r-', basefmt='r-',
label='Sampled Signal')
plt.title('Sampled Sinusoidal Signal')
plt.xlabel('Time (s)')
plt.grid(True)
plt.legend()
# Reconstruction by simply interpolating between samples
reconstructed_signal = np.interp(t_continuous, t_samples, sampled_signal)
# Plot the reconstructed signal
plt.figure(figsize=(10, 4))
plt.plot(t_continuous, continuous_signal, label='Original Signal',
linestyle='dashed')
plt.plot(t_samples, sampled_signal, label='Sampled Signal', marker='o',
linestyle='None')
plt.plot(t_continuous, reconstructed_signal, label='Reconstructed Signal')
plt.title('Reconstruction of Analog Signal from Samples')
plt.xlabel('Time (s)')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
:
```

Output:


Continuous Sinusoidal Signal
Continuous Signal
Time (s)
Sampled Sinusoidal Signal
Sampled Signal
Time (s)


Reconstruction of Analog Signal from Samples
Original Signal
Sampled Signal
Reconstructed Signal
Time (s)

Experiment No: 02

Name of the Experiment: Write a Program to Implement Z-transform of a Discrete Time Function, Inverse Z-transform, Pole-zeros diagram and Root of a system.

Objectives:

- To convert a discrete-time signal into a representation in the Z-domain.
- To convert a function in the Z-domain back into the time-domain representation.
- To provide a graphical representation of the poles and zeros of a system in the Z-domain.

Theory:

Z-Transform: The Z-transform is a mathematical transformation used in discrete-time signal processing and digital control theory. It plays a fundamental role in analyzing and processing discrete-time signals and systems, particularly in the context of difference equations and discrete-time system representations. The Z-transform is analogous to the Laplace transform used in continuous-time (analog) signal processing and control systems.

Given a discrete-time signal $x[n]$, the Z-transform $X(z)$ is defined as:

$$X(z) = \sum_{n=-\infty}^{\infty} x[n] \cdot z^{-n}$$

Here, z is a complex variable, and $X(z)$ is a function of z that represents the Z-transform of the discrete-time signal $x[n]$.

Inverse Z-Transform: The inverse Z-transform is used to go from the Z-domain back to the time-domain. It finds the original discrete-time signal from its Z-transform representation. There are various techniques for finding the inverse Z-transform, including partial fraction expansion and power series expansion. The inverse Z-transform is a mathematical operation that allows us to convert a Z-transform function into a discrete-time function. It is the inverse operation of the Z-transform and is used to recover the original discrete-time signal from its Z transform. Mathematically, the inverse Z-transform of a function X(z) is given by the contour integral:

$$x[n] = \frac{1}{2\pi j} \oint_c X(z) z^{n-1} dz$$

Where, C is a closed contour in the complex plane that encloses all the poles of X(z). The contour C can be chosen in various ways depending on the properties of X(z) and the desired accuracy of the result.

Poles and Zeros:

Poles: Poles are the values of the complex variable that make the system's transfer equal to infinity. In other words, they are the roots of the denominator polynomial of the transfer function.

Zeros: Zeros are the values of the complex variable that make the system's transfer function equal to zero. They are the roots of the numerator polynomial of the transfer function.

The two polynomials, P(z) and Q(z) , allow us to find the poles and zeros of the Z-Transform.

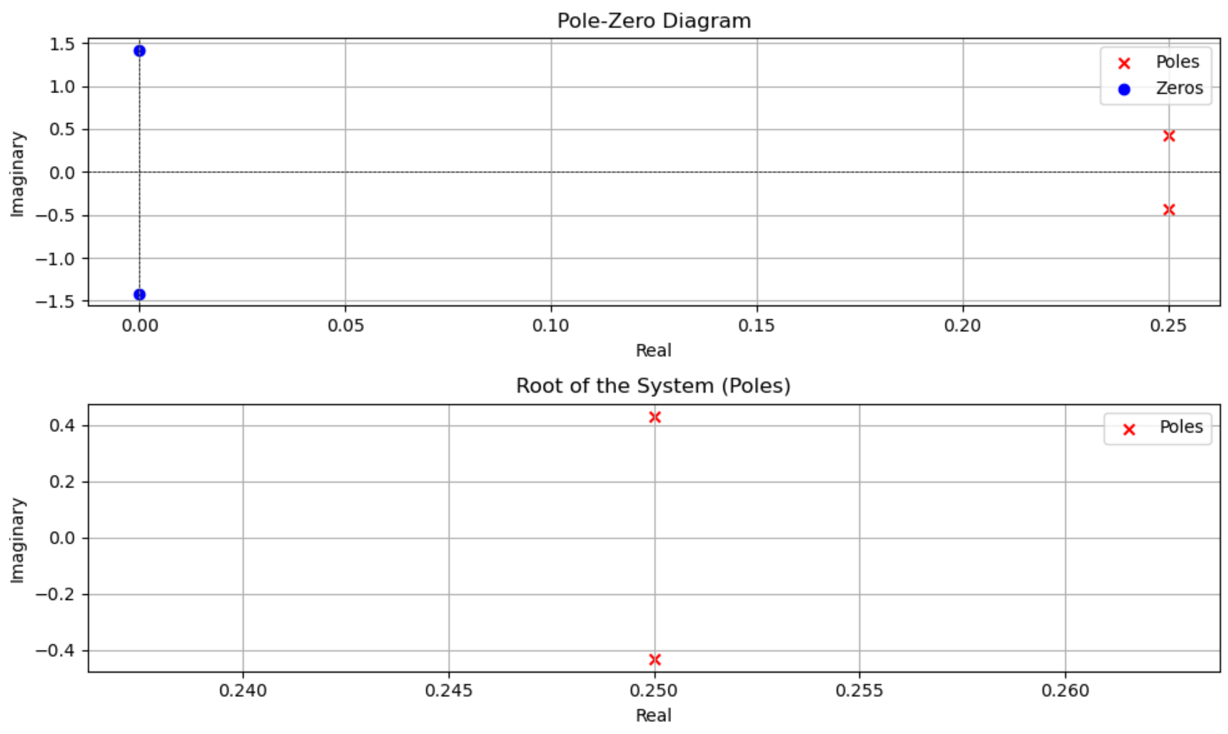
- Poles: The value(s) for z where Q(z)=0.
- Zeros: The value(s) for z where P(z)=0.

Source Code in Python:

```
import numpy as np

import matplotlib.pyplot as plt

from scipy import signal

# Define a discrete-time function as a list of coefficients (e.g., b0, b1, b2, ... for the numerator)

numerator = [1, 0, 2]  # Example: Z-transform of X(z) = 1 + 2z^-2


# Define a list of coefficients for the denominator
```

```
denominator = [1, -0.5, 0.25]  # Example: Z-transform of Y(z) = 1 - 0.5z^-1 + 0.25z^-
2

# Calculate the Z-transform

z_transform_result = signal.TransferFunction(numerator, denominator, dt=1.0)  # dt
is the sampling time (set to 1 for simplicity)

# Get the poles and zeros of the system

poles = z_transform_result.poles

zeros = z_transform_result.zeros

# Inverse Z-transform

t, inverse_z_transform_result = signal.dstep((numerator, denominator, 1.0))

# Plot pole-zero diagram

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

plt.subplot(2, 1, 1)

plt.scatter(np.real(poles), np.imag(poles), marker='x', color='red', label='Poles')

plt.scatter(np.real(zeros), np.imag(zeros), marker='o', color='blue', label='Zeros')

plt.axvline(x=0, color='black', linestyle='--', linewidth=0.5)

plt.axhline(y=0, color='black', linestyle='--', linewidth=0.5)

plt.title('Pole-Zero Diagram')

plt.xlabel('Real')

plt.ylabel('Imaginary')

plt.grid(True)

plt.legend()

# Plot the root of the system

plt.subplot(2, 1, 2)

plt.scatter(np.real(poles), np.imag(poles), marker='x', color='red', label='Poles')

plt.title('Root of the System (Poles)')
plt.xlabel('Real')
plt.ylabel('Imaginary')
plt.grid(True)
plt.legend()
# Show the plots
plt.tight_layout()
plt.show()
# Print the inverse Z-transform result (time-domain signal)
print("Inverse Z-Transform Result (Time-Domain Signal):")
print(inverse_z_transform_result)
```

Output:



Experiment No: 03

Name of the Experiment: Write a Program to Implement The Discrete Fourier Transform (DFT) and Fast Fourier Transform (FFT).

Objectives:

 To implement and understand the concept of DFT and FFT.

 To perform various signal processing tasks.

 To analyze the frequency content of a signal.

Theory:

DFT (Discrete Fourier Transform): The Discrete Fourier Transform (DFT) is a mathematical transformation used to analyze and represent a discrete-time signal in the frequency domain. It converts a sequence of N complex or real numbers (samples) into a sequence of complex numbers representing the signal's frequency components. The DFT is defined by the following formula:

$X[k]=\sum n=0N-1x[n]\cdot e-j(2\pi/N)kn$

Where:

- $X[k]$ is the complex result of the DFT at frequency bin k.
- $x[n]$ is the input signal's discrete-time sample at time index n.
- N is the number of samples in the input signal.
- j is the imaginary unit.

Key points about the DFT:

- It provides a way to analyze the frequency content of a discrete signal, allowing you to identify the presence and magnitude of various frequency components.

- The DFT returns a complex result for each frequency bin, consisting of both magnitude and phase information.

- The computation complexity of the DFT is O(N^2), which can be inefficient for long signals.

FFT (Fast Fourier Transform): The Fast Fourier Transform (FFT) is an algorithm for efficiently computing the Discrete Fourier Transform (DFT) of a sequence. It significantly speeds up the calculation of the DFT, reducing the computational complexity from O(N^2) to O(N log N), making it much faster for large datasets. The FFT is widely used in signal processing, engineering, and scientific computing.

Key points about the FFT:

-It is an algorithmic technique rather than a separate mathematical transform. The FFT computes the same result as the DFT but does so more efficiently.

-The FFT is particularly valuable for real-time applications, where rapid frequency analysis is required.

-Various FFT algorithms exist, with the Cooley-Tukey algorithm being one of the most common.

The solution to the given data set can be obtained either in time-domain or frequency-domain. Within this fram work, there are two commonly used FFTs. They are

1. Decimation-in-time FFT (Time-domain analysis).

2. Decimation-in-frequency FFT (Frequency-domain analysis).

Decimation-in-time FFT: Decimation in Time (DIT) Radix 2 FFT algorithm converts the time domain N point sequence x(n) to a frequency domain N-point sequence X(k).In Decimation in Time algorithm the time domain sequence x(n) is decimated and smaller point DFT are performed. The results of smaller point DFTs are combined to get the result of N-point DFT.

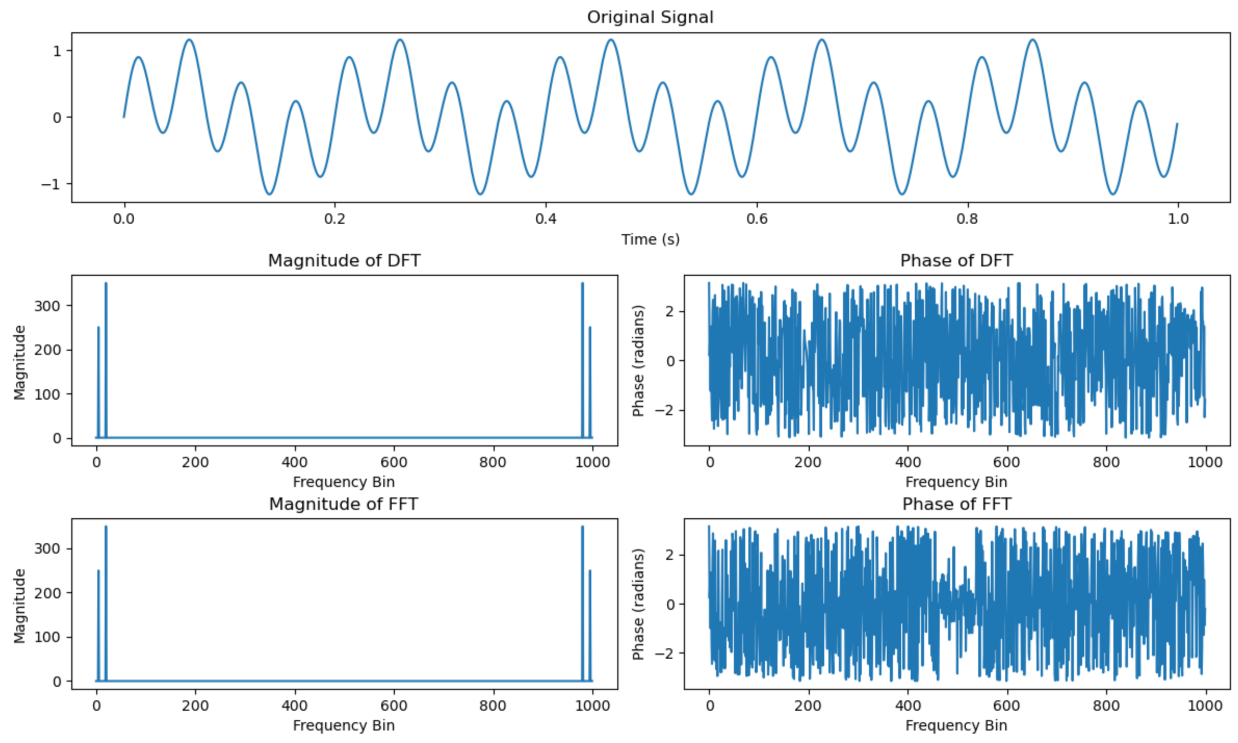
Decimation-in-Frequency FFT: Decimation-in-frequency (DIF) FFT algorithm, original sequence s(n) is decomposed into two subsequences as first half and second half of a sequence. There is no need of reordering (shuffling) the original sequence as in Radix-2 decimation-in-time (DIT) FFT algorithm.

Source Code in Python:

```
import numpy as np
import matplotlib.pyplot as plt
# Function to compute the Discrete Fourier Transform (DFT)
def compute_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 * k * n / N)
    return X
# Function to compute the Fast Fourier Transform (FFT)
def compute_fft(x):
    return np.fft.fft(x)
# Sample parameters
Fs = 1000  # Sampling frequency (Hz)
T = 1.0   # Duration (s)
N = int(T * Fs)  # Number of samples
t = np.linspace(0.0, T, N, endpoint=False)  # Time values
f1 = 5.0  # Frequency of the first sinusoid (Hz)
f2 = 20.0  # Frequency of the second sinusoid (Hz)
# Create a signal with two sinusoids
signal = 0.5 * np.sin(2 * np.pi * f1 * t) + 0.7 * np.sin(2 * np.pi * f2 * t)
# Compute the DFT and FFT of the signal
dft_result = compute_dft(signal)
```

```
fft_result = compute_fft(signal)
# Plot the original signal and its frequency domain representations
plt.figure(figsize=(12, 8))
# Original signal
plt.subplot(3, 1, 1)
plt.plot(t, signal)
plt.title('Original Signal')
plt.xlabel('Time (s)')
# Magnitude of DFT
plt.subplot(3, 2, 3)
plt.plot(np.abs(dft_result))
plt.title('Magnitude of DFT')
plt.xlabel('Frequency Bin')
plt.ylabel('Magnitude')
# Phase of DFT
plt.subplot(3, 2, 4)
plt.plot(np.angle(dft_result))
plt.title('Phase of DFT')
plt.xlabel('Frequency Bin')
plt.ylabel('Phase (radians)')
# Magnitude of FFT
plt.subplot(3, 2, 5)
plt.plot(np.abs(fft_result))
plt.title('Magnitude of FFT')
plt.xlabel('Frequency Bin')
plt.ylabel('Magnitude')
# Phase of FFT
plt.subplot(3, 2, 6)
plt.plot(np.angle(fft_result))
plt.title('Phase of FFT')
plt.xlabel('Frequency Bin')
plt.ylabel('Phase (radians)')
plt.tight_layout()
plt.show()
```

Output:



Experiment No: 04

Name of the Experiment: Write a Program to Designing Finite Impulse Response (FIR) Filters and Infinite Impulse Response (IIR) Filters.

Objectives:

- To design and understand the concept of FIR and IIR Filters.
- To understand how to filtering out noise using digital Filters.
- To calculate the lower and upper cut-off frequency.
- To know the smoothing a signal.
- To calculate the gain.

Theory:

Filter: A filter, in the context of signal processing and engineering, is a device, circuit, algorithm, or system that selectively allows certain frequencies or components of a signal to pass through while attenuating or blocking others. Filters are used to modify or process signals by emphasizing or de-emphasizing specific frequency components, smoothing out noise, removing unwanted interference, or shaping the signal's frequency response to meet desired specifications.

FIR Filter (Finite Impulse Response): An FIR filter has a finite-length impulse response. It is used in both digital and analog domains and is characterized by its finite-duration response to a pulse input.There are various types of filters, and they can be classified based on several factors, including their frequency response, order, and purpose. Some common types of filters include:

Low-Pass Filter (LPF): A low-pass filter allows signals with frequencies lower than a certain cutoff frequency to pass through while attenuating higher frequencies. It is commonly used to remove high-frequency noise or to smooth out a signal.

High-Pass Filter (HPF): A high-pass filter does the opposite of a low-pass filter; it allows signals with frequencies higher than a cutoff frequency to pass while attenuating lower frequencies. HPFs are useful for removing low-frequency interference.

Band-Pass Filter (BPF): A band-pass filter allows a specific range of frequencies to pass while attenuating frequencies outside that range. BPFs are used in applications where you need to isolate a specific band of frequencies.

Band-Stop Filter (BSF) or Notch Filter: A band-stop filter, also known as a notch filter, attenuates signals within a specific frequency range while allowing frequencies outside that range to pass. They are used to eliminate narrowband interference or unwanted frequencies.

FIR Notch Filter: A FIR (finite impulse response) notch filter is a type of digital filter that attenuates a narrow range of frequencies around a center frequency, while allowing other frequencies to pass through. It is called a notch filter because it creates a "notch" or dip in the frequency response at the center frequency.

To design a FIR notch filter, we need to specify the center frequency of the notch, the width of the notch (i.e., the range of frequencies to attenuate), and the desired level of attenuation. There are several methods to design FIR filters, including windowing, frequency-sampling, and leasts quares.

FIR Multi Band Filter: A Finite Impulse Response (FIR) Multi-Band Filter is a type of digital signal processing filter that is designed to attenuate or pass frequency bands in a signal.Overall, the FIR Multi-Band Filter is a powerful tool for signal processing that can be used to achieve precise control over the frequency content of a signal.

Digital Filter: Digital filters are implemented in software or digital signal processors (DSPs) to process discrete-time signals. They can be designed using various algorithms and have applications in fields like digital audio processing and image processing.

Analog Filter: Analog filters are typically implemented using electronic components, such as resistors, capacitors, and inductors, to process continuous analog signals. They have applications in analog electronics, communications, and control systems.

IIR Filter: Infinite Impulse Response (IIR) filters are one of two primary types of digital filters used in Digital Signal Processing (DSP) application. It is a property applying to many linear time-invariant systems that are distinguished by having an impulse response h(t) which does not become exactly zero past a certain point, but continues indefinitely. Also, FIR digital filter can be classified as

- Low Pass Filter (LPF).
- High Pass Filter (HPF).
- Band Pass Filter (BPF).
- Band Stop Filter (BSF)

IIR Low Pass Filter: An IIR (Infinite Impulse Response) Low Pass Filter is a type of digital filter that attenuates high-frequency signals while passing low-frequency signals. It achieves this by exploiting the feedback mechanism in its design, which causes the output of the filter to depend on both its input and previous outputs. The transfer function of an IIR low pass filter can be expressed as-

$$H(z) = 1 / (1 + a1z^{-1} + a2z^{-2} + ... + an*z^{-n})$$

Where, z^{-1} represents a delay of one sample, and a1, a2,..., an are coefficients that determine the characteristics of the filter's frequency response.

IIR High Pass Filter: An IIR (Infinite Impulse Response) High Pass Filter is a type of digital filter that attenuates low-frequency signals while passing high-frequency signals. It achieves this by using a similar feedback mechanism as the IIR low pass filter, but with the coefficients chosen to emphasize high-frequency signals instead. The transfer function of an IIR high pass filter can be expressed as:

$$H(z) = (1 - a1z^{-1} - a2z^{-2} - ... - anz^{-n}) / (1 + b1z^{-1} + b2z^{-2} + ... + bmz^{-m}).$$

IIR Band Pass Filter: An IIR (Infinite Impulse Response) Band Pass Filter is a type of digital filter that passes a range of frequencies while attenuating frequencies outside of that range. It achieves this by using a combination of low pass and high pass filters in its design. The transfer function of an IIR band pass filter can be expressed as:

$$H(z) = (1 - a1z^{-1} - a2z^{-2} - ... - anz^{-n}) / (1 + b1z^{-1} + b2z^{-2} + ... + bmz^{-m})$$

IIR Band Stop Filter: An IIR (Infinite Impulse Response) Band Stop Filter is a type of digital filter that attenuates a range of frequencies while passing frequencies outside of that range. It achieves this by using a combination of low pass and high pass filters in its design. The transfer function of an IIR band stop filter can be expressed as:

H(z) = (1 + a1z^-1 + a2z^-2 + ... + anz^-n) / (1 + b1z^-1 + b2z^-2 + ... + bmz^-m)

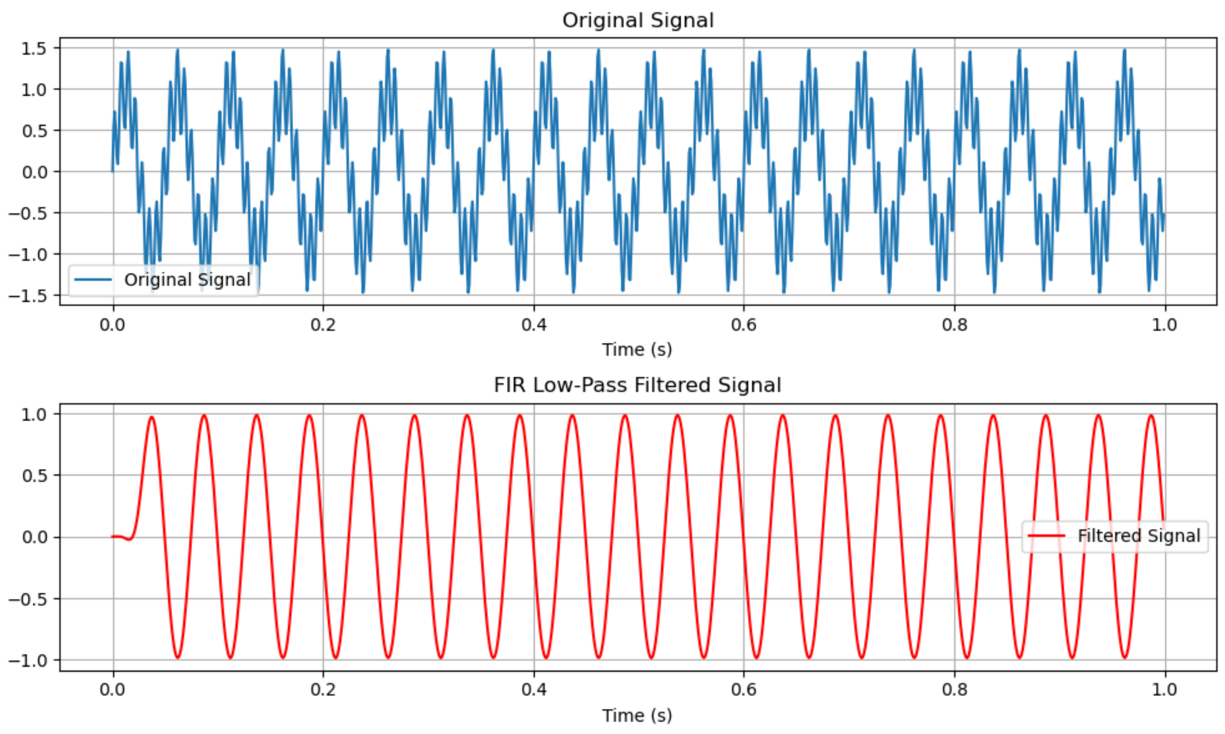
Where, z^-1 represents a delay of one sample, and a1, a2,..., an and b1, b2, ..., bm are coefficients that determine the characteristics of the filter's frequency response

Source Code in Python:

1.FIR Low Pass Filter

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import firwin, lfilter
# Define the filter parameters
cutoff_frequency = 0.1  # Adjust this value to set the cutoff frequency
filter_order = 50  # Adjust the filter order as needed
# Create the FIR low-pass filter coefficients
filter_coefficients = firwin(filter_order, cutoff=cutoff_frequency)
# Generate a test signal
sampling_frequency = 1000  # Sampling frequency (Hz)
t = np.arange(0, 1, 1 / sampling_frequency)  # Time vector
test_signal = np.sin(2 * np.pi * 20 * t) + 0.5 * np.sin(2 * np.pi * 150 * t)
# Apply the FIR filter to the test signal
filtered_signal = lfilter(filter_coefficients, 1.0, test_signal)
# Plot the original and filtered signals
plt.figure(figsize=(10, 6))
plt.subplot(2, 1, 1)
plt.plot(t, test_signal, label='Original Signal')
plt.title('Original Signal')
plt.xlabel('Time (s)')
plt.grid(True)
plt.legend()
plt.subplot(2, 1, 2)
plt.plot(t, filtered_signal, label='Filtered Signal', color='red')
plt.title('FIR Low-Pass Filtered Signal')
plt.xlabel('Time (s)')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```

Output:

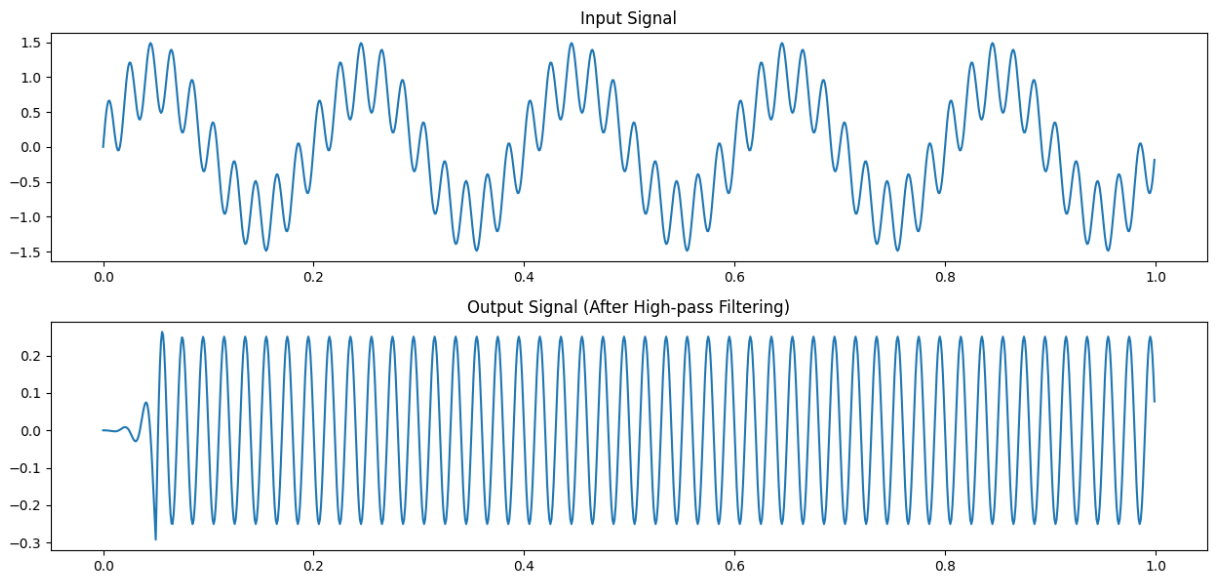


2.FIR High Pass Filter

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

```
# Define the parameters of the filter
num_taps = 101  # Number of filter taps (coefficients)
cutoff_freq = 0.1  # Cutoff frequency in fraction of Nyquist frequency

# Generate the FIR filter coefficients
coefficients = firwin(num_taps, cutoff=cutoff_freq, pass_zero=False)

# Generate a sample input signal (you can replace this with your own signal)
t = np.linspace(0, 1, 1000, endpoint=False)
input_signal = np.sin(2 * np.pi * 5 * t) + 0.5 * np.sin(2 * np.pi * 50 * t)

# Apply the filter to the input signal
output_signal = lfilter(coefficients, 1.0, input_signal)

# Plot the input and output signals
plt.figure(figsize=(10, 6))
plt.subplot(2, 1, 1)
plt.plot(t, input_signal)
plt.title('Input Signal')
plt.subplot(2, 1, 2)
plt.plot(t, output_signal)
plt.title('Output Signal (After High-pass Filtering)')
plt.tight_layout()
plt.show()
```

Output:

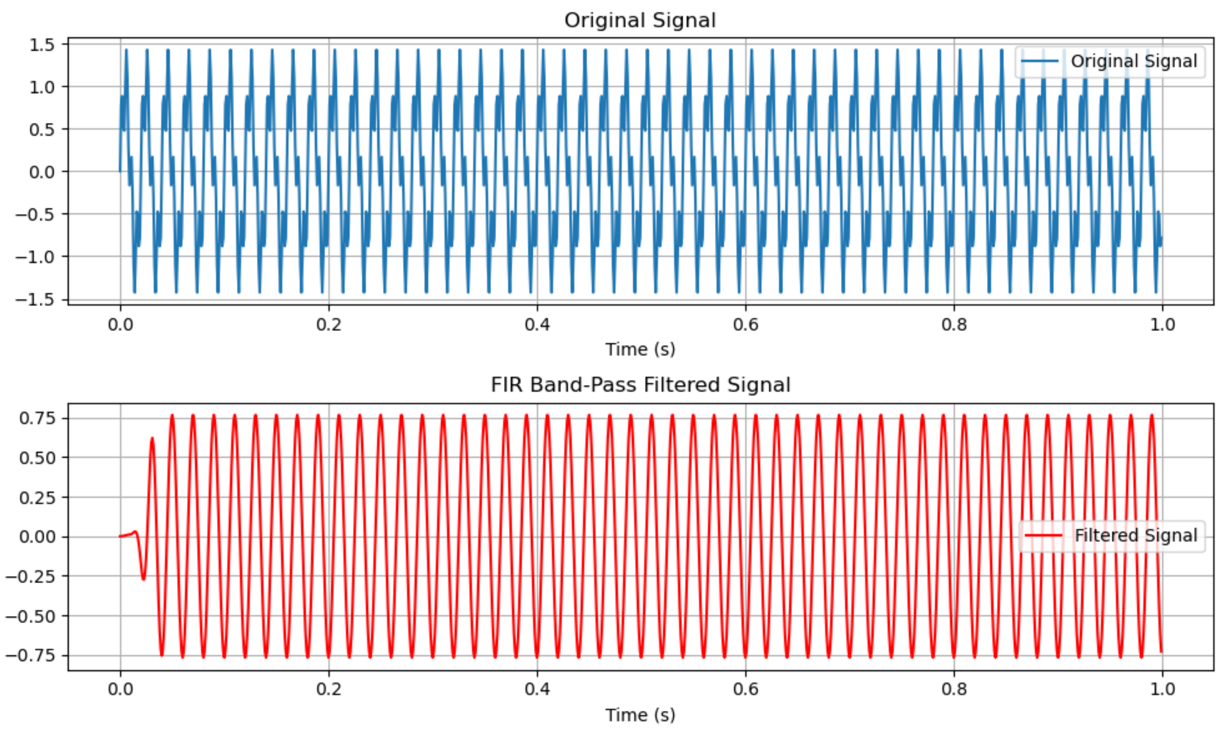


3.FIR Band Pass Filter

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import firwin, lfilter
# Define filter parameters
sampling_frequency = 1000  # Sampling frequency (Hz)
nyquist_frequency = 0.5 * sampling_frequency
filter_order = 51  # Filter order (adjust as needed)
low_cutoff = 40.0  # Lower cutoff frequency (Hz)
high_cutoff = 100.0  # Upper cutoff frequency (Hz)
# Create the FIR band-pass filter coefficients
filter_coefficients = firwin(filter_order, [low_cutoff / nyquist_frequency,
high_cutoff / nyquist_frequency], pass_zero=False)
# Generate a test signal
t = np.arange(0, 1, 1 / sampling_frequency)  # Time vector
test_signal = np.sin(2 * np.pi * 50 * t) + 0.5 * np.sin(2 * np.pi * 200 * t) + 0.2
* np.sin(2 * np.pi * 500 * t)
# Apply the FIR filter to the test signal
```

```
filtered_signal = lfilter(filter_coefficients, 1.0, test_signal)
# Plot the original and filtered signals
plt.figure(figsize=(10, 6))
plt.subplot(2, 1, 1)
plt.plot(t, test_signal, label='Original Signal')
plt.title('Original Signal')
plt.xlabel('Time (s)')
plt.grid(True)
plt.legend()
plt.subplot(2, 1, 2)
plt.plot(t, filtered_signal, label='Filtered Signal', color='red')
plt.title('FIR Band-Pass Filtered Signal')
plt.xlabel('Time (s)')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```

Output:

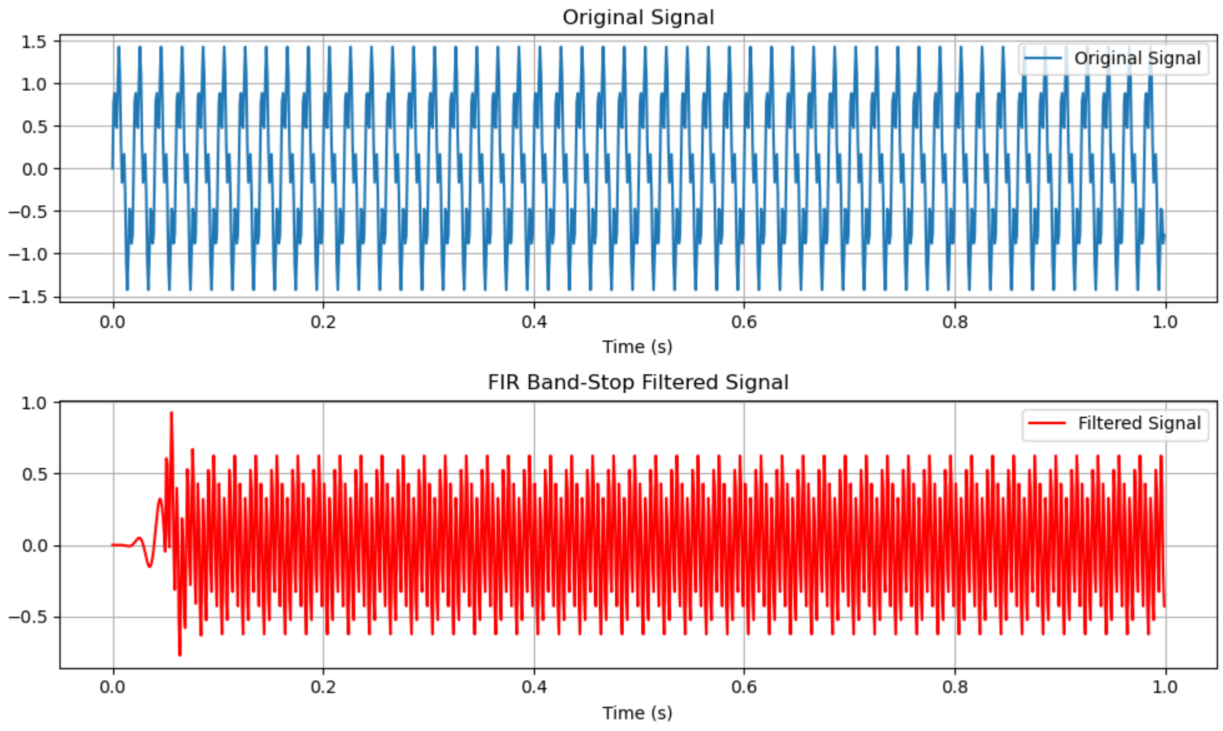


4.FIR Band Stop Filter

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import firwin, lfilter
# Define filter parameters
```

```
sampling_frequency = 1000  # Sampling frequency (Hz)
nyquist_frequency = 0.5 * sampling_frequency
filter_order = 101  # Filter order (adjust as needed)
low_cutoff = 40.0  # Lower cutoff frequency (Hz)
high_cutoff = 60.0  # Upper cutoff frequency (Hz)
# Create the FIR band-stop filter coefficients
filter_coefficients = firwin(filter_order, [low_cutoff / nyquist_frequency,
high_cutoff / nyquist_frequency], pass_zero=True)
# Generate a test signal
t = np.arange(0, 1, 1 / sampling_frequency)  # Time vector
test_signal = np.sin(2 * np.pi * 50 * t) + 0.5 * np.sin(2 * np.pi * 200 * t) + 0.2
* np.sin(2 * np.pi * 500 * t)
# Apply the FIR filter to the test signal
filtered_signal = lfilter(filter_coefficients, 1.0, test_signal)
# Plot the original and filtered signals
plt.figure(figsize=(10, 6))
plt.subplot(2, 1, 1)
plt.plot(t, test_signal, label='Original Signal')
plt.title('Original Signal')
plt.xlabel('Time (s)')
plt.grid(True)
plt.legend()
plt.subplot(2, 1, 2)
plt.plot(t, filtered_signal, label='Filtered Signal', color='red')
plt.title('FIR Band-Stop Filtered Signal')
plt.xlabel('Time (s)')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```

Output:

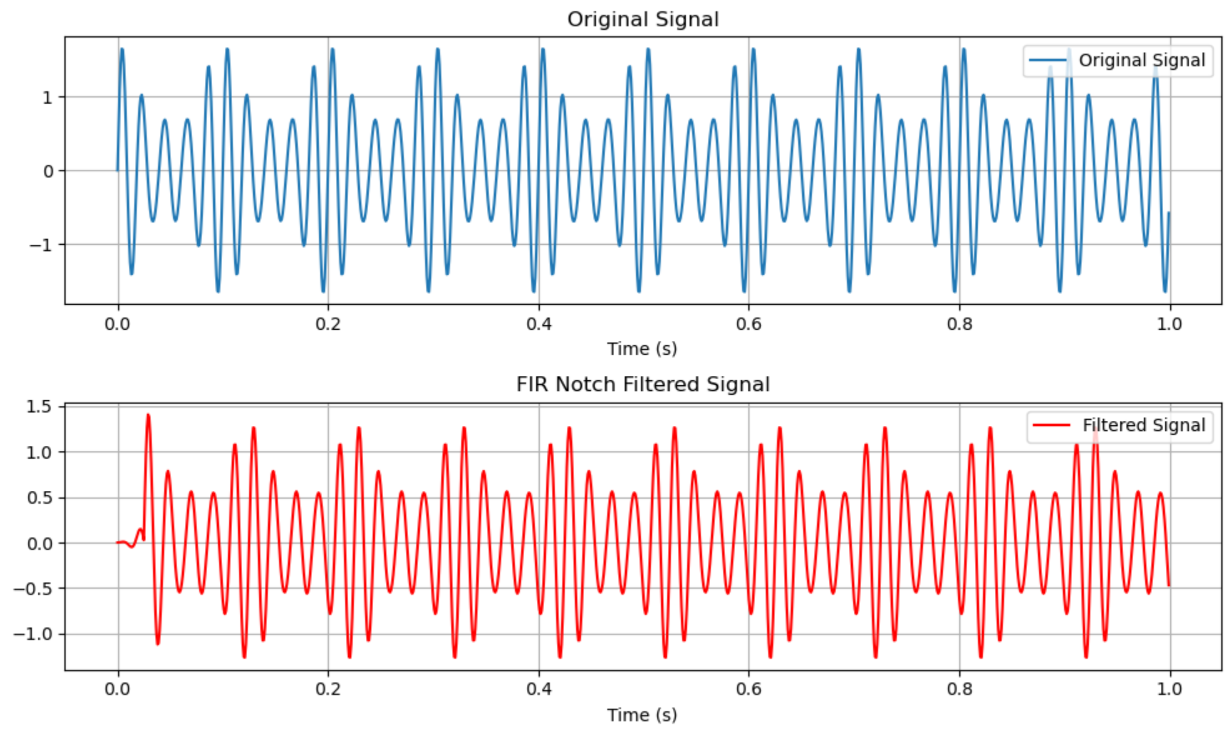


5.FIR Notch Filter

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import firwin, lfilter
# Define filter parameters
sampling_frequency = 1000  # Sampling frequency (Hz)
nyquist_frequency = 0.5 * sampling_frequency
filter_order = 51  # Filter order (adjust as needed)
center_frequency = 60.0  # Center frequency of the notch (Hz)
notch_bandwidth = 10.0  # Bandwidth of the notch (Hz)
# Calculate the notch frequencies
low_cutoff = center_frequency - 0.5 * notch_bandwidth
high_cutoff = center_frequency + 0.5 * notch_bandwidth
```

```
# Create the FIR notch filter coefficients
filter_coefficients = firwin(filter_order, [low_cutoff / nyquist_frequency,
high_cutoff / nyquist_frequency], pass_zero=True)
# Generate a test signal
t = np.arange(0, 1, 1 / sampling_frequency)  # Time vector
test_signal = np.sin(2 * np.pi * 50 * t) + 0.5 * np.sin(2 * np.pi * 60 * t) + 0.2
* np.sin(2 * np.pi * 70 * t)
# Apply the FIR filter to the test signal
filtered_signal = lfilter(filter_coefficients, 1.0, test_signal)
# Plot the original and filtered signals
plt.figure(figsize=(10, 6))
plt.subplot(2, 1, 1)
plt.plot(t, test_signal, label='Original Signal')
plt.title('Original Signal')
plt.xlabel('Time (s)')
plt.grid(True)
plt.legend()
plt.subplot(2, 1, 2)
plt.plot(t, filtered_signal, label='Filtered Signal', color='red')
plt.title('FIR Notch Filtered Signal')
plt.xlabel('Time (s)')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```

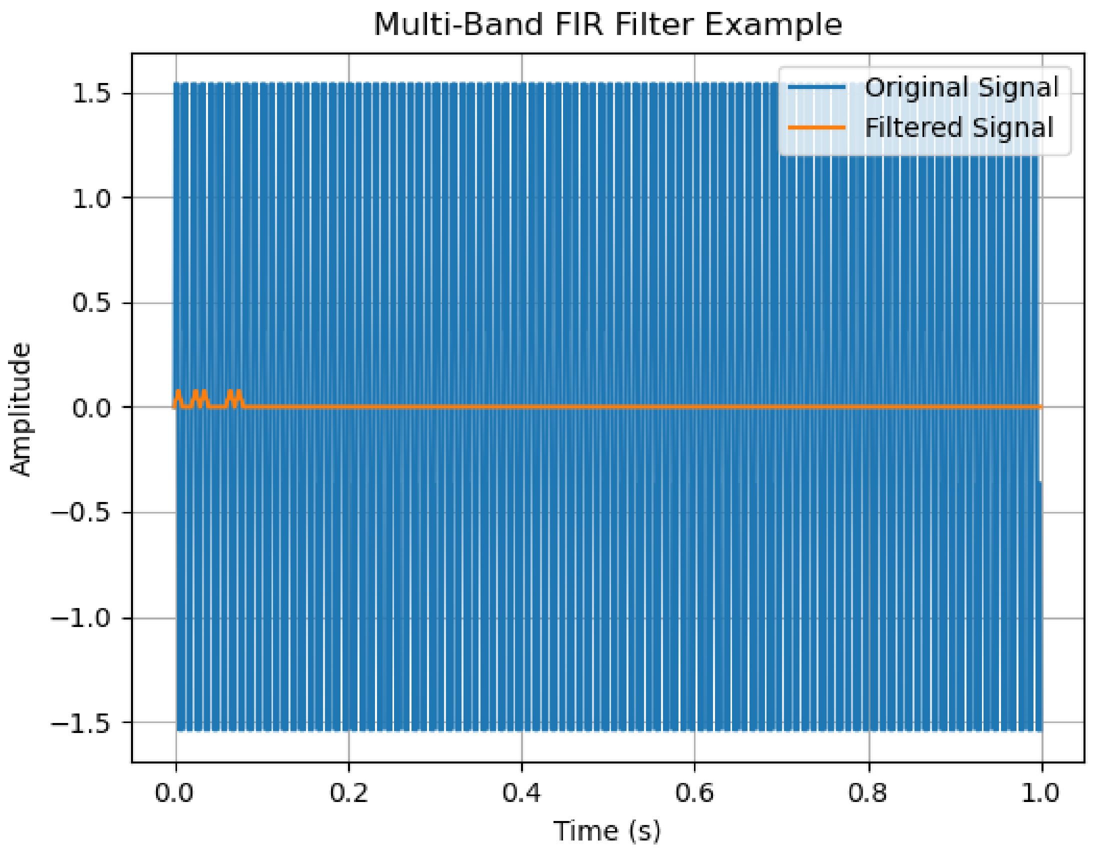
Output:



6. FIR Multi Band Filter

```
import numpy as np
from scipy.signal import lfilter
import matplotlib.pyplot as plt
# Define the sampling frequency and the length of the filter
fs = 1000  # Sampling frequency (Hz)
T = 1.0  # Length of the signal (s)
t = np.linspace(0, T, int(T * fs), endpoint=False)  # Time vector
# Define the frequency bands and their corresponding filter coefficients
bands = [(0, 50), (100, 200), (300, 400)]  # Define frequency bands in Hz
# Create a multi-band FIR filter
filter_order = 100  # Filter order
coefficients = []  # To store filter coefficients for each band
for band in bands:
    low, high = band
    nyquist = 0.5 * fs
    low_cutoff = low / nyquist
    high_cutoff = high / nyquist
    coefficients.append(np.zeros(filter_order + 1))
    coefficients[-1][int(low_cutoff    *    filter_order):int(high_cutoff    *
filter_order)] = 1.0
# Combine filter coefficients for all bands
combined_filter = np.sum(coefficients, axis=0)
# Normalize the filter so that its frequency response sums to 1
combined_filter /= np.sum(combined_filter)
# Generate a test signal
signal = np.sin(2 * np.pi * 100 * t) + np.sin(2 * np.pi * 300 * t) + np.sin(2 *
np.pi * 500 * t)

# Apply the multi-band FIR filter to the signal
filtered_signal = lfilter(combined_filter, 1, signal)
# Plot the original and filtered signals
plt.figure()
plt.plot(t, signal, label="Original Signal")
plt.plot(t, filtered_signal, label="Filtered Signal")
plt.legend()
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.title("Multi-Band FIR Filter Example")
plt.grid()
plt.show()
```

Output:



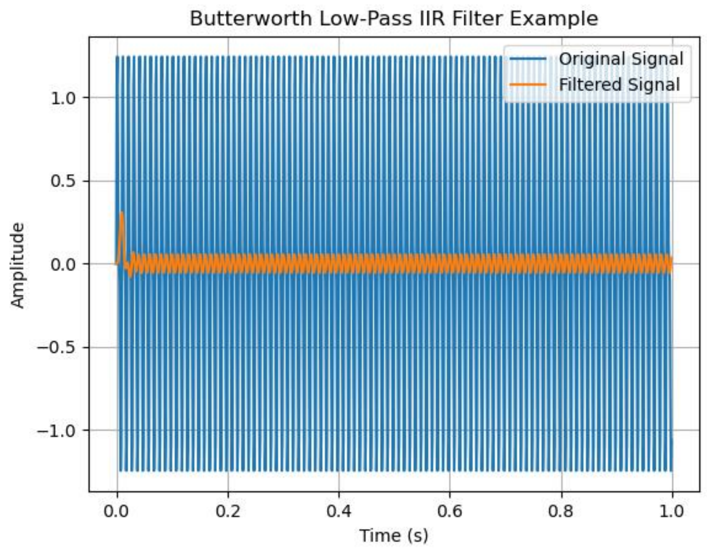
IIR Filters:

1.Low Pass Filter:

```
import numpy as np
from scipy.signal import butter, lfilter
import matplotlib.pyplot as plt
# Define the sampling frequency and the cutoff frequency of the low-pass filter
fs = 1000  # Sampling frequency (Hz)
cutoff_frequency = 50  # Cutoff frequency of the low-pass filter (Hz)
# Design the Butterworth low-pass filter
order = 4  # Filter order
nyquist = 0.5 * fs
normal_cutoff = cutoff_frequency / nyquist
b, a = butter(order, normal_cutoff, btype='low', analog=False)
```

```
# Generate a test signal
T = 1.0  # Length of the signal (s)
t = np.linspace(0, T, int(T * fs), endpoint=False)  # Time vector
signal = np.sin(2 * np.pi * 100 * t) + 0.5 * np.sin(2 * np.pi * 200 * t)
# Apply the Butterworth low-pass filter to the signal
filtered_signal = lfilter(b, a, signal)
# Plot the original and filtered signals
plt.figure()
plt.plot(t, signal, label="Original Signal")
plt.plot(t, filtered_signal, label="Filtered Signal")
plt.legend()
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.title("Butterworth Low-Pass IIR Filter Example")
plt.grid()
plt.show()
```

Output:

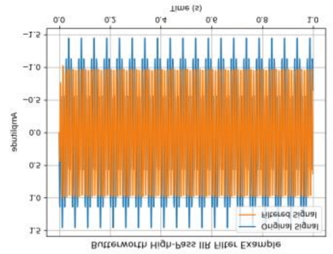


2. High Pass Filter:

```
import numpy as np
from scipy.signal import butter, lfilter
import matplotlib.pyplot as plt
# Define the sampling frequency and the cutoff frequency of the high-pass filter
```

```
fs = 1000  # Sampling frequency (Hz)
cutoff_frequency = 50  # Cutoff frequency of the high-pass filter (Hz)
# Design the Butterworth high-pass filter
order = 4  # Filter order
nyquist = 0.5 * fs
normal_cutoff = cutoff_frequency / nyquist
b, a = butter(order, normal_cutoff, btype='high', analog=False)
# Generate a test signal
T = 1.0  # Length of the signal (s)
t = np.linspace(0, T, int(T * fs), endpoint=False)  # Time vector
signal = np.sin(2 * np.pi * 100 * t) + 0.5 * np.sin(2 * np.pi * 20 * t)
# Apply the Butterworth high-pass filter to the signal
filtered_signal = lfilter(b, a, signal)
# Plot the original and filtered signals
plt.figure()
plt.plot(t, signal, label="Original Signal")
plt.plot(t, filtered_signal, label="Filtered Signal")
plt.legend()
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.title("Butterworth High-Pass IIR Filter Example")
plt.grid()
plt.show()
```

Output:

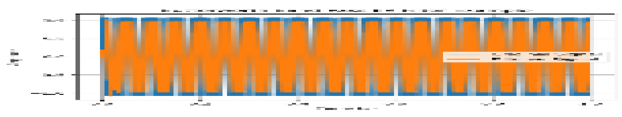


3. Band Pass Filter:

```
import numpy as np
from scipy.signal import butter, lfilter
import matplotlib.pyplot as plt
# Define the sampling frequency and the bandpass frequency range
fs = 1000  # Sampling frequency (Hz)
```

```
lowcut = 50  # Lower cutoff frequency (Hz)
highcut = 200  # Upper cutoff frequency (Hz)
# Design the Butterworth band-pass filter
order = 4  # Filter order
nyquist = 0.5 * fs
low = lowcut / nyquist
high = highcut / nyquist
b, a = butter(order, [low, high], btype='band')
# Generate a test signal
T = 1.0  # Length of the signal (s)
t = np.linspace(0, T, int(T * fs), endpoint=False)  # Time vector
signal = np.sin(2 * np.pi * 100 * t) + 0.5 * np.sin(2 * np.pi * 300 * t) + np.sin(2
* np.pi * 500 * t)
# Apply the Butterworth band-pass filter to the signal
filtered_signal = lfilter(b, a, signal)
# Plot the original and filtered signals
plt.figure()
plt.plot(t, signal, label="Original Signal")
plt.plot(t, filtered_signal, label="Filtered Signal")
plt.legend()
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.title("Butterworth Band-Pass IIR Filter Example")
plt.grid()
plt.show()
```

Output:

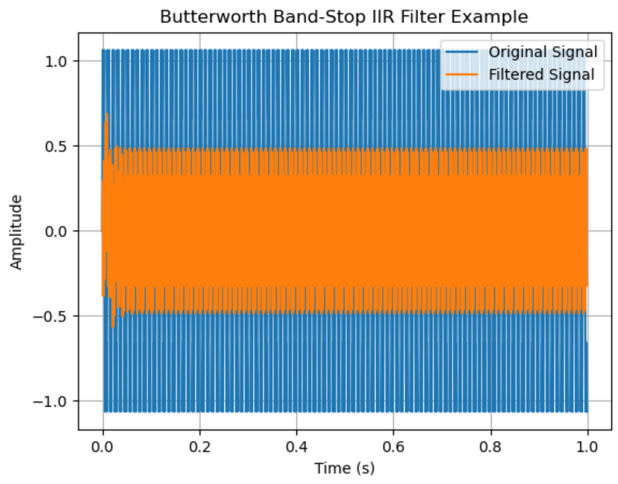


4.IIR Band Stop Filter

```
import numpy as np
from scipy.signal import butter, lfilter
import matplotlib.pyplot as plt
# Define the sampling frequency and the band-stop frequency range
fs = 1000  # Sampling frequency (Hz)
```

```
lowcut = 50  # Lower cutoff frequency (Hz)
highcut = 200  # Upper cutoff frequency (Hz)
# Design the Butterworth band-stop filter
order = 4  # Filter order
nyquist = 0.5 * fs
low = lowcut / nyquist
high = highcut / nyquist
b, a = butter(order, [low, high], btype='bandstop')
# Generate a test signal
T = 1.0  # Length of the signal (s)
t = np.linspace(0, T, int(T * fs), endpoint=False)  # Time vector
signal = np.sin(2 * np.pi * 100 * t) + 0.5 * np.sin(2 * np.pi * 300 * t) + np.sin(2
* np.pi * 500 * t)
# Apply the Butterworth band-stop filter to the signal
filtered_signal = lfilter(b, a, signal)
# Plot the original and filtered signals
plt.figure()
plt.plot(t, signal, label="Original Signal")
plt.plot(t, filtered_signal, label="Filtered Signal")
plt.legend()
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.title("Butterworth Band-Stop IIR Filter Example")
plt.grid()
plt.show()
```

Output:


Butterworth Band-Stop IIR Filter Example
Original Signal
Filtered Signal
Amplitude
Time (s)
1.0
0.5
0.0
−0.5
−1.0
0.0
0.2
0.4
0.6
0.8
1.0