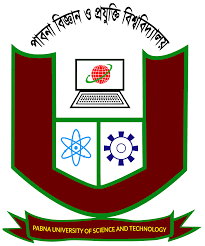
****

**PABNA UNIVERSITY OF SCIENCE AND TECHNOLOGY**

**Faculty of Science and Technology**

***Department of Information and Communication Engineering***

**LAB REPORT**

**Course Title: Signal and System Sessional**

**Course Code: ICE-2204**

**Submitted By:**

Name: Mst. Sabi Afrin Mim

Roll: 220615

Session: 2021-2022

2nd Year 2nd Semester

Dept.of ICE, PUST

**Submitted To:**

Dr. MD. Imran Hossain

Associate Professor

Department of ICE, Pabna University of Science and Technology, Pabna

**Submission Date: 02/03/2025**

**INDEX**

|  |  |
| --- | --- |
| Experiment No | Experiment Name |
| 01 | Signal Operations in Discrete-Time Systems(addition, multiplication, shifting, folding, scaling) |
| 02 | Discrete-Time Signal Representations: Impulse, Step, and Ramp Signals. |
| 03 | Convolution of Two Sinusoidal Signals with Phase Shift. |
| 04 | Cross-Correlation and Auto-Correlation of Sinusoidal Signals |
| 05 | PPG Signal Processing and Heart Rate Estimation. |
| 06 | Time Domain to Frequency Domain Analysis of a Signal Using Fast Fourier Transform (FFT) |
| 07 | **Signal Processing Using Discrete Fourier Transform (DFT) and Noise Removal Techniques** |

**Experiment No: 01**

**Title: Signal Operations in Discrete-Time Systems.**

**Objectives:**

The goal of this experiment is to explore various operations performed on discrete-time signals, including addition, multiplication, scaling, shifting, and folding. These operations are fundamental in signal processing and system analysis, helping in understanding how signals behave under various transformations.

**Theory:**

In signal processing, discrete-time signals can be manipulated in several ways. The following operations are commonly used:

1. **Signal Addition:** This operation involves adding two signals of the same length. If x1[n] and x2[n] are two discrete-time signals, their sum is given by:

y[n] = x1[n]+x2[n]

1. **Signal Multiplication:** In this operation, two signals are multiplied point by point. The result of multiplying two signals x1[n]x and x2[n] is:

y[n] = x1[n]⋅x2[n]

1. **Signal Scaling:** Scaling a signal involves multiplying it by a constant scalar. If α\alphaα is a scalar, the scaled version of x[n] is:

y[n] = α⋅x[n]

1. **Signal Shifting:** Shifting a signal involves shifting its time index. A right shift by k

units is represented as:

y[n] = x[n−k]

A left shift can be done similarly.

1. **Signal Folding:** Folding a signal, or reflection, is the operation of reversing the signal's time axis:

y[n]=x[−n]

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

# Define the signal operations

def signal\_addition(x1, x2):

    return x1 + x2

def signal\_multiplication(x1, x2):

    return x1 \* x2

def signal\_scaling(x, alpha):

    return alpha \* x

def signal\_shifting(n, shift):

    return np.roll(n, shift)  # Use np.roll for shifting the indices

def signal\_folding(x):

    return np.flip(x)

# Time and signal definitions

n = np.array([-3,-2, -1, 0, 1, 2])

x1 = np.array([4,3,6,1,4,9])

x2 = np.array([4,3,5,1,1,10])

# Operations

added\_signal = signal\_addition(x1, x2)

multiplied\_signal = signal\_multiplication(x1, x2)

scaled\_signal = signal\_scaling(x1, 2)

shifted\_signal1 = signal\_shifting(n, -2)

shifted\_signal2 = signal\_shifting(n, 2)

folded\_signal = signal\_folding(x1)

# Plotting the results with smaller figure size

plt.figure(figsize=(8, 10))

# Plot original signal x1

plt.subplot(4, 2, 1)

plt.stem(n, x1)

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.title("Original Signal x1")

plt.grid()

# Plot original signal x2

plt.subplot(4, 2, 2)

plt.stem(n, x2)

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.title("Original Signal x2")

plt.grid()

# Plot added signal

plt.subplot(4, 2, 3)

plt.stem(n, added\_signal)

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.title("Signal Addition")

plt.grid()

# Plot multiplied signal

plt.subplot(4, 2, 4)

plt.stem(n, multiplied\_signal)

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.title("Signal Multiplication")

plt.grid()

# Plot scaled signal

plt.subplot(4, 2, 5)

plt.stem(n, scaled\_signal)

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.title("Scaled Signal (x1 \* 2)")

plt.grid()

# Plot shifted signal (Shift = -2)

plt.subplot(4, 2, 6)

plt.stem(shifted\_signal1, x1)

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.title("Shifted Signal (Shift = -2)")

plt.grid()

# Plot shifted signal (Shift = +2)

plt.subplot(4, 2, 7)

plt.stem(shifted\_signal2, x1)

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.title("Shifted Signal (Shift = +2)")

plt.grid()

# Plot folded signal

plt.subplot(4, 2, 8)

plt.stem(n, folded\_signal)

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.title("Folded Signal (x1)")

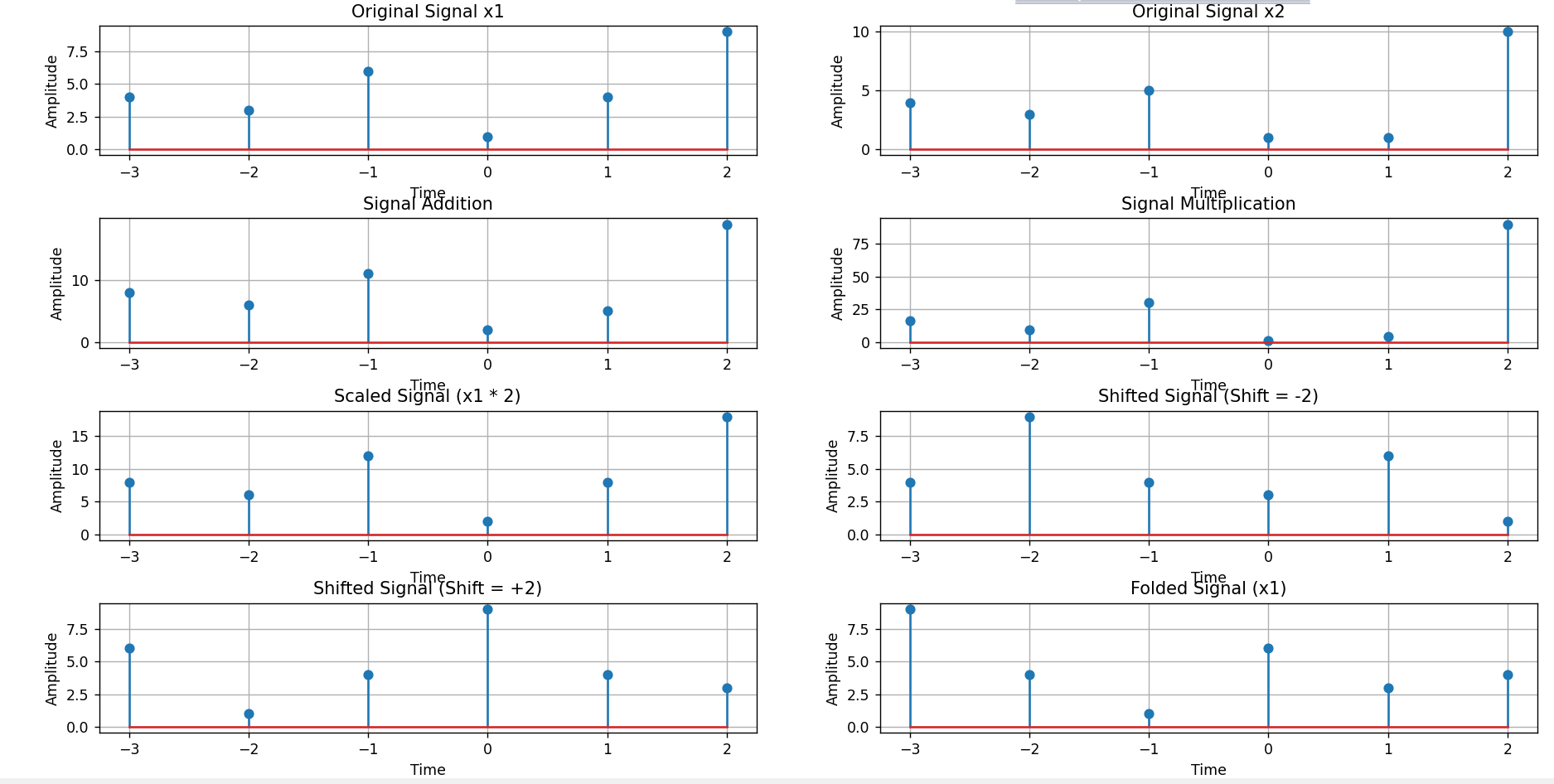
plt.grid()

# Adjust layout for better view

plt.tight\_layout()

plt.show()

**Output:**



**Experiment No**:  **02**

**Title:**  **Discrete-Time Signal Representations: Impulse, Step, and Ramp Signals.**

**Objectives:**

1. Illustrate the behavior of these basic signals.
2. Explore the role these signals play in the foundation of digital signal processing.
3. Provide a clear understanding of how these signals interact in different systems, helping in the design and analysis of complex systems.

**Theory:**

In signal processing, discrete-time signals are commonly represented as sequences of numbers defined at specific time instants. The unit impulse, step, and ramp signals are basic building blocks used to analyze and manipulate more complex signals.

**1. Unit Impulse Signal δ[n]:**

The unit impulse signal, often denoted as δ[n], is a sequence that is zero for all values of n, except at n=0 n ,where it is defined as δ[0]=1. Mathematically, it can be represented as:

δ[n]

This signal is used to represent an instantaneous, "impulsive" event in a system.

**2. Unit Step Signal u[n]:**

The unit step signal, denoted as u[n], is a sequence that is zero for negative values of n and one for zero or positive values of n. Its mathematical representation is:

u[n]

The step signal is used to model systems that begin at some point in time and remain constant thereafter.

**3. Ramp Signal r[n]:**

The ramp signal, denoted as r[n], is a sequence that increases linearly for non-negative values of n and is zero for negative values of n. Mathematically, it can be expressed as:

r[n]

The ramp signal is useful in modeling systems where a quantity increases gradually over time.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

# Define range of n

n = np.arange(-10, 11)  # From -10 to 10

# Impulse Signal δ[n]

impulse = np.where(n == 0, 1, 0)

# Step Signal u[n]

step = np.where(n >= 0, 1, 0)

# Ramp Signal r[n]

ramp = np.where(n >= 0, n, 0)

# Plot the Signals

plt.figure(figsize=(12, 8))

# Impulse Signal Plot

plt.subplot(3, 1, 1)

plt.stem(n, impulse)

plt.title("Unit Impulse Signal δ[n]")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

# Step Signal Plot

plt.subplot(3, 1, 2)

plt.stem(n, step)

plt.title("Unit Step Signal u[n]")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

# Ramp Signal Plot

plt.subplot(3, 1, 3)

plt.stem(n, ramp)

plt.title("Ramp Signal r[n]")

plt.xlabel("n")

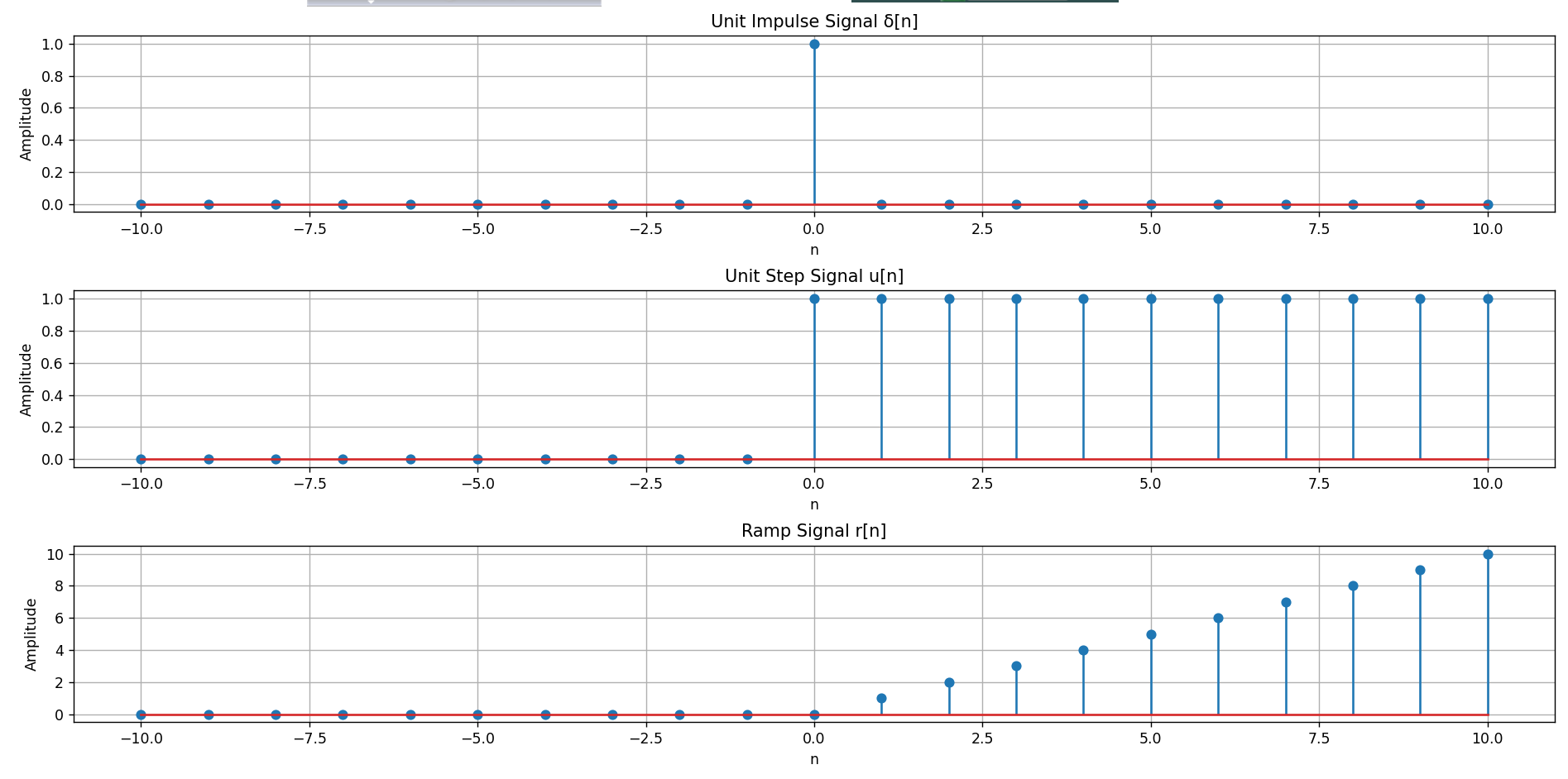
plt.ylabel("Amplitude")

plt.grid()

plt.tight\_layout()

plt.show()

**Output:**



**Experiment No: 03**

**Title:** **Convolution of Two Sinusoidal Signals with Phase Shift.**

**Objectives:**

To compute and visualize the convolution of two sinusoidal signals with the same frequency but a phase shift. This helps analyze their interaction over time, revealing the effect of the phase shift on the signals' alignment and similarity. The convolution result is plotted against time lag to observe the correlation between the two signals.

**Theory:**

Convolution is a mathematical operation that combines two signals to produce a third. In discrete-time signal processing, the convolution of two signals x[n] and h[n] is given by the formula:

[n]=

Where:

* x[n] is the first signal,
* h[n] is the second signal,
* y[n] is the resulting convolution output.

The **autoconvolution** of a signal is simply the convolution of the signal with itself. When the signal is shifted, the convolution gives insight into how the signal overlaps as one part shifts relative to another.

When noise is introduced to a signal, convolution helps us analyze the effect of noise on the system, which is important in filter design and signal processing.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import convolve

# Parameters for the signals

sample\_rate = 1000  # samples per second

duration = 1  # seconds

frequency1 = 5  # Hz for signal 1

frequency2 = 5  # Hz for signal 2, but shifted slightly in phase

# Generate the time array and original signals

t = np.linspace(0, duration, int(sample\_rate \* duration), endpoint=False)

signal1 = np.sin(2 \* np.pi \* frequency1 \* t)

signal2 = np.sin(2 \* np.pi \* frequency2 \* t + np.pi/4)  # Signal 2 is phase-shifted by pi/4

# Compute convolution between signal1 and signal2

convoluted\_signal = convolve(signal1, signal2, mode='full')

# Time lags for the convolution plot

lags = np.arange(-len(signal1) + 1, len(signal1))

# Plot convolution result

plt.figure(figsize=(12, 6))

plt.plot(lags / sample\_rate, convoluted\_signal)

plt.xlabel("Lag [s]")

plt.ylabel("Convolution Result")

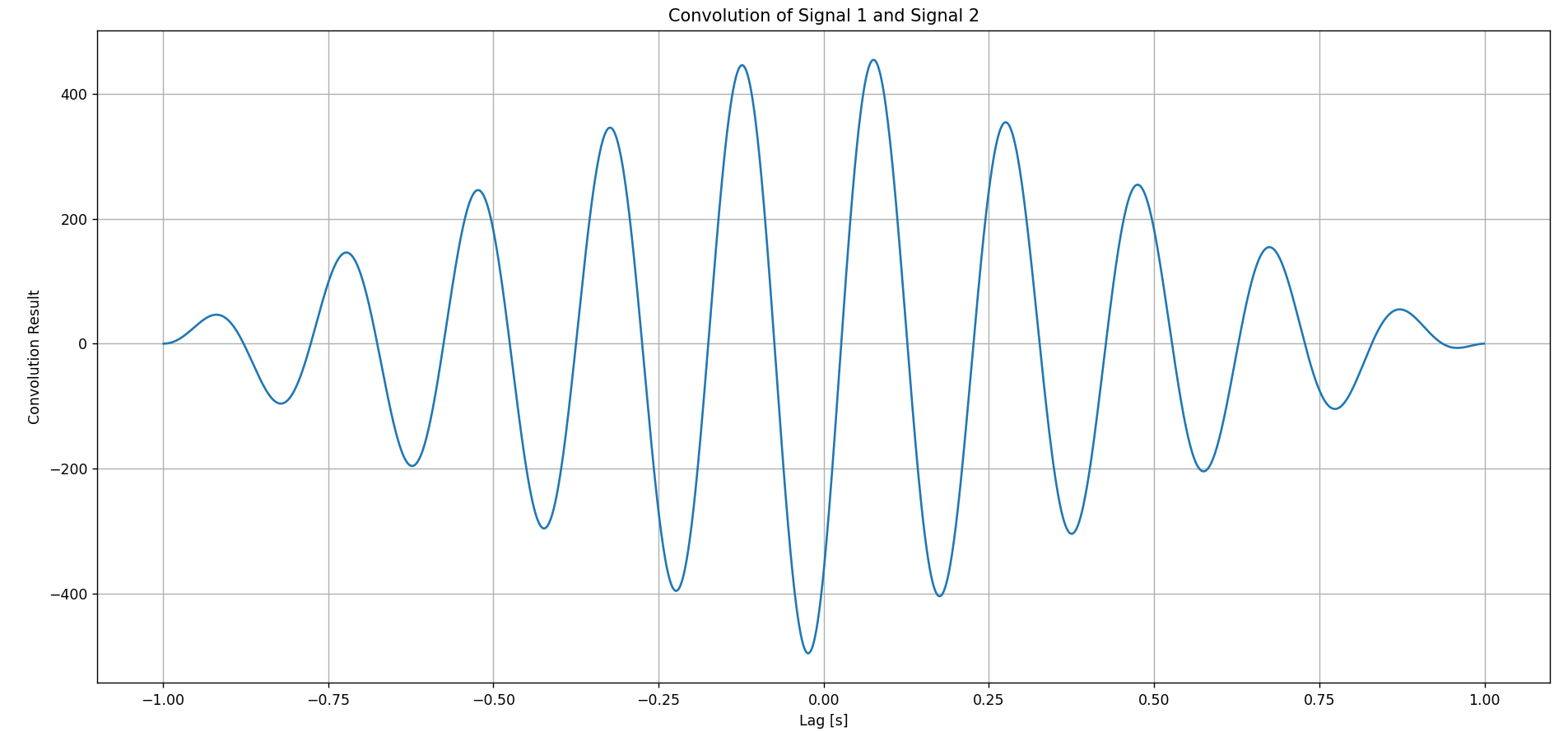
plt.title("Convolution of Signal 1 and Signal 2")

plt.grid(True)

plt.tight\_layout()

plt.show()

**Output:**



**Experiment No:**  **04**

**Title:** **Cross-Correlation and Auto-Correlation of Sinusoidal Signals**

**Objectives:**

The purpose of this experiment is to compute and analyze the **cross-correlation** and **auto-correlation** of two sinusoidal signals. By doing so, we aim to:

1. **Cross-correlation**: Investigate the similarity between two sinusoidal signals, one of which is phase-shifted, and observe how the signals correlate as a function of the time shift (lag). This helps us understand how a signal resembles a shifted version of another signal.
2. **Auto-correlation**: Analyze the similarity of a signal with itself over different time shifts (lags). By computing the auto-correlation of a sinusoidal signal, we aim to observe periodicity and how the signal correlates with itself.

Through these analyses, we aim to understand how signals interact with each other and how the correlation function can reveal properties like periodicity and similarity between signals.

**Theory:**

In signal processing, correlation is a measure of similarity between two signals. The correlation function provides information about how one signal relates to another over a given range of lags or shifts. There are two types of correlation used in this experiment:

* **Cross-correlation**: It measures the similarity between two different signals at different lags. Cross-correlation is used to detect if one signal is similar to a shifted version of another.

The mathematical formula for the cross-correlation of signals x(t) and y(t)is given by:

* **Auto-correlation**: It measures the similarity of a signal with itself at different lags. This is useful for analyzing periodic signals, as the highest correlation occurs when the signal is aligned with itself.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import correlate

# Parameters for the signals

sample\_rate = 1000  # samples per second

duration = 1  # seconds

frequency1 = 5  # Hz for signal 1

frequency2 = 5  # Hz for signal 2, but shifted slightly in phase

# Generate the time array and original signals

t = np.linspace(0, duration, int(sample\_rate \* duration), endpoint=False)

signal1 = np.sin(2 \* np.pi \* frequency1 \* t)

signal2 = np.sin(2 \* np.pi \* frequency2 \* t + np.pi/4)  # Signal 2 is phase-shifted by pi/4

# Compute cross-correlation

cross\_corr = correlate(signal1, signal2, mode='full')

# Compute auto-correlation for signal1

auto\_corr\_signal1 = correlate(signal1, signal1, mode='full')

lags = np.arange(-len(signal1) + 1, len(signal1))

# Plot cross-correlation and auto-correlation

plt.figure(figsize=(12, 6))

# Plot cross-correlation between signal1 and signal2

plt.subplot(2, 1, 1)

plt.plot(lags / sample\_rate, cross\_corr)

plt.xlabel("Lag [s]")

plt.ylabel("Cross-correlation")

plt.title("Cross-correlation between Signal 1 and Signal 2")

plt.grid(True)

# Plot auto-correlation for signal1

plt.subplot(2, 1, 2)

plt.plot(lags / sample\_rate, auto\_corr\_signal1)

plt.xlabel("Lag [s]")

plt.ylabel("Auto-correlation")

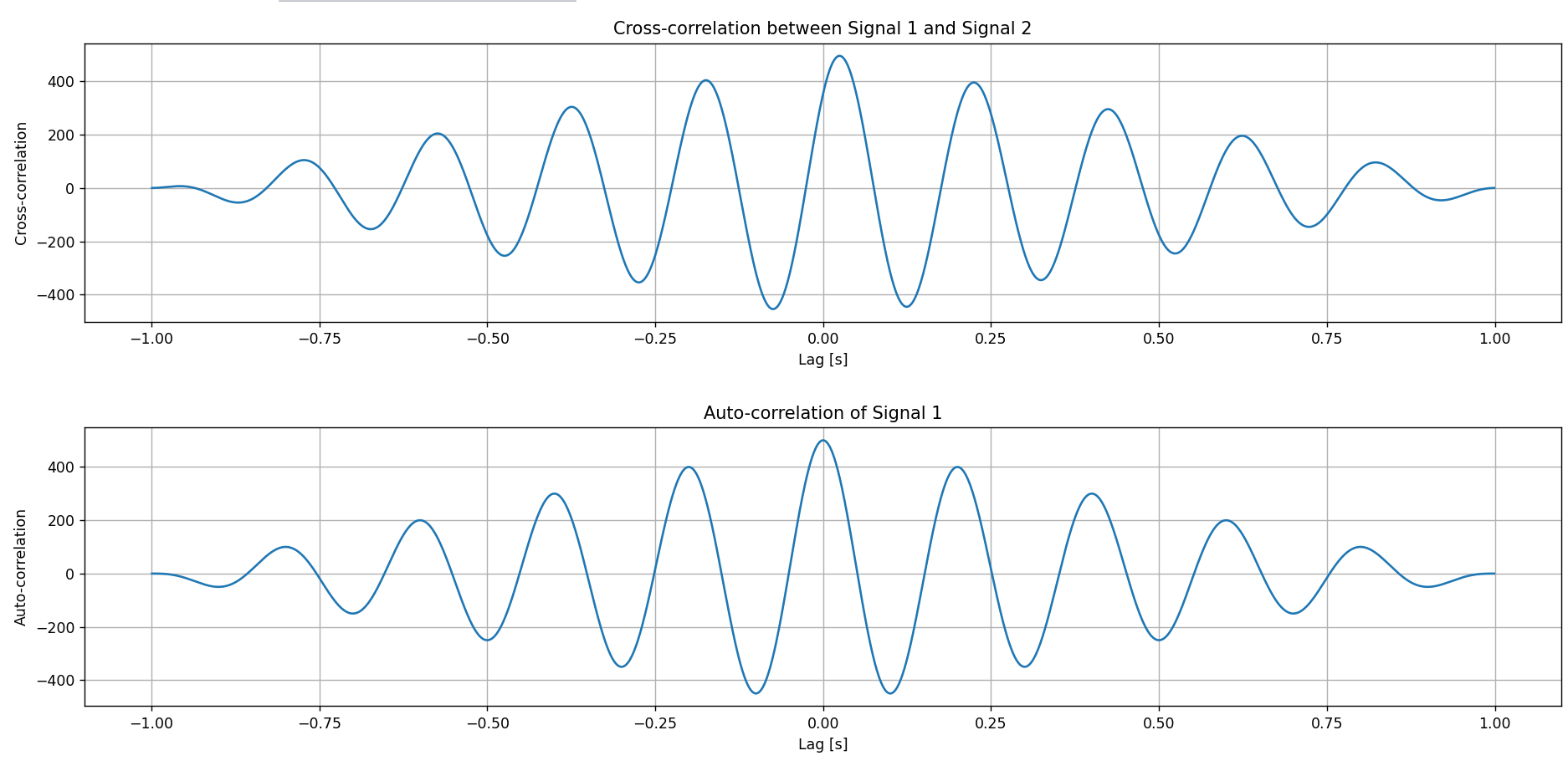
plt.title("Auto-correlation of Signal 1")

plt.grid(True)

plt.tight\_layout()

plt.show()

**Output:**



**Experiment No: 05**

**Title:** **PPG Signal Processing and Heart Rate Estimation.**

**Objectives:**

The objective of this experiment is to simulate, process, and analyze a Photoplethysmogram (PPG) signal, which is used to measure the changes in blood volume and estimate heart rate. The key steps involved are:

1. Simulating a noisy PPG signal.
2. Filtering the signal using a low-pass filter.
3. Normalizing the filtered signal for further analysis.
4. Detecting peaks in the signal (representing heartbeats).
5. Estimating the heart rate in beats per minute (BPM).

**Theory:**

 **PPG Signal**: A PPG signal is a simple optical measurement that provides information about blood volume changes in the microvascular bed of tissue. It is usually obtained using a light source and a photodetector placed on the skin.

 **Low-Pass Filter**: A low-pass filter allows signals with frequencies below a certain cutoff frequency to pass through while attenuating frequencies higher than the cutoff. It is used to remove high-frequency noise from the PPG signal.

 **Normalization**: Normalization refers to scaling the signal to a fixed range, typically [0, 1], which allows for easier comparison and interpretation of the signal.

 **Peak Detection**: In PPG signals, the peaks correspond to the heartbeat intervals. These are identified to estimate the heart rate.

 **Heart Rate Estimation**: The heart rate can be estimated by calculating the average time interval between peaks (Inter-Beat Interval, IBI) and converting it to beats per minute (BPM).

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import find\_peaks, butter, filtfilt

# Simulated PPG Signal

fs = 100  # Sampling frequency (Hz)

t = np.linspace(0, 10, fs \* 10)  # 10 seconds signal

ppg\_signal = 1 + 0.5 \* np.sin(2 \* np.pi \* 1.2 \* t) + 0.1 \* np.random.normal(size=len(t))  # Signal with noise

# Low-pass filter function

def butter\_lowpass\_filter(data, cutoff, fs, order=4):

    nyquist = 0.5 \* fs

    normal\_cutoff = cutoff / nyquist

    b, a = butter(order, normal\_cutoff, btype='low', analog=False)

    return filtfilt(b, a, data)

# Apply Low-pass filter to the signal

filtered\_signal = butter\_lowpass\_filter(ppg\_signal, cutoff=2.5, fs=fs)

# Normalize the filtered PPG signal

normalized\_ppg = (filtered\_signal - np.min(filtered\_signal)) / (np.max(filtered\_signal) - np.min(filtered\_signal))

# Detect peaks in the filtered signal (heartbeats)

peaks, \_ = find\_peaks(filtered\_signal, distance=fs \* 0.6)  # Minimum 0.6s apart

# Plotting

plt.figure(figsize=(12, 12))

# Plot 1: Raw PPG Signal

plt.subplot(4, 1, 1)

plt.plot(t, ppg\_signal, label="Raw PPG Signal", color='green')

plt.title("Raw PPG Signal (With Noise)")

plt.xlabel("Time (seconds)")

plt.ylabel("Amplitude")

plt.legend()

plt.grid()

# Plot 2: Filtered PPG Signal

plt.subplot(4, 1, 2)

plt.plot(t, filtered\_signal, label="Filtered Signal", color='blue')

plt.title("Filtered PPG Signal (Low-Pass Filtered)")

plt.xlabel("Time (seconds)")

plt.ylabel("Amplitude")

plt.legend()

plt.grid()

# Plot 3: Normalized PPG Signal

plt.subplot(4, 1, 3)

plt.plot(t, normalized\_ppg, label="Normalized PPG Signal", color='purple')

plt.title("Normalized PPG Signal")

plt.xlabel("Time (seconds)")

plt.ylabel("Normalized Amplitude")

plt.legend()

plt.grid()

# Plot 4: Filtered Signal with Detected Peaks (Heartbeats)

plt.subplot(4, 1, 4)

plt.plot(t, filtered\_signal, label="Filtered Signal", color='blue')

plt.scatter(t[peaks], filtered\_signal[peaks], color='red', label="Detected Peaks")

plt.title("PPG Signal with Detected Peaks (Heartbeats)")

plt.xlabel("Time (seconds)")

plt.ylabel("Amplitude")

plt.legend()

plt.grid()

plt.tight\_layout()

plt.show()

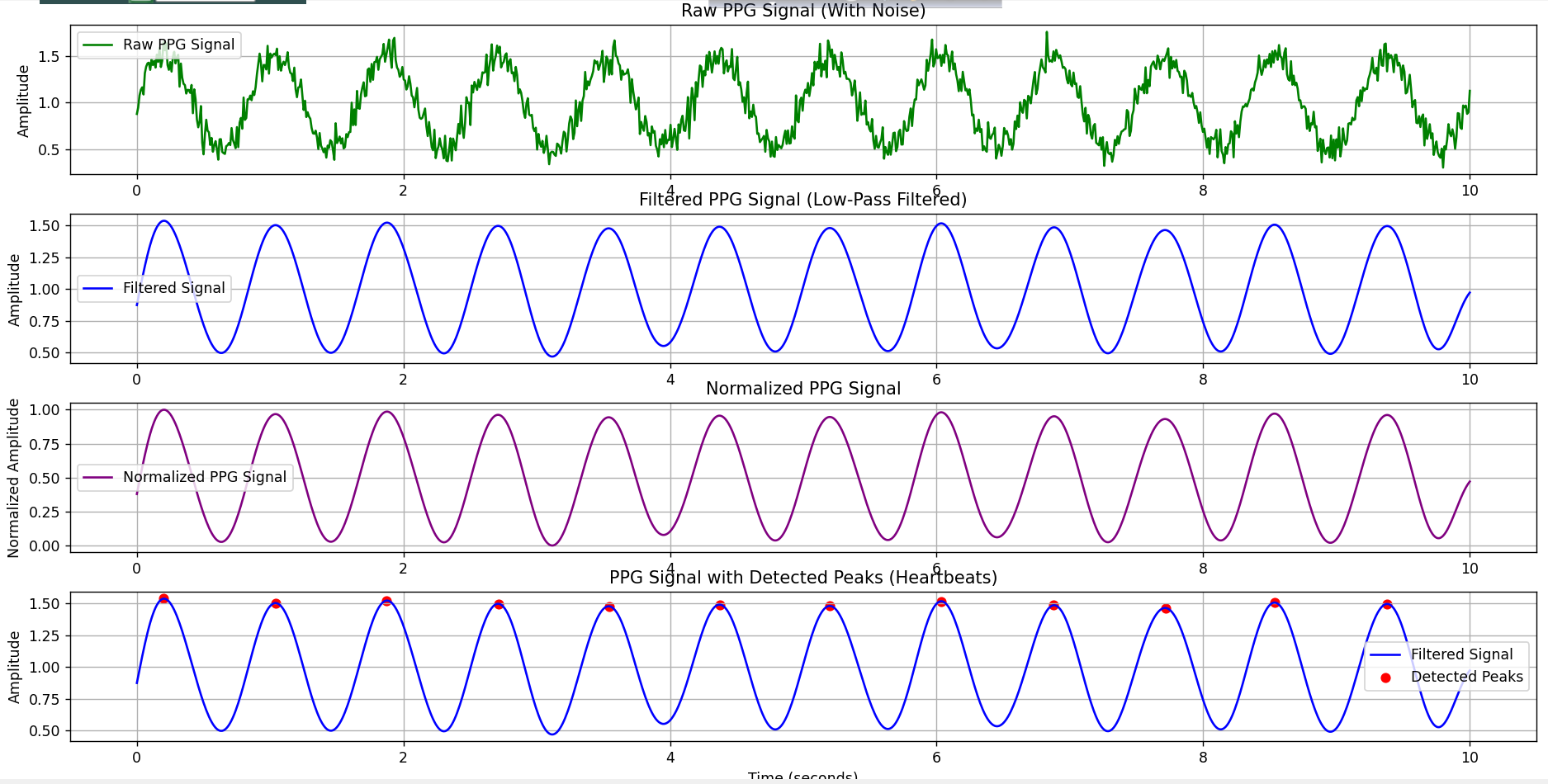
# Print the estimated heart rate

ibi = np.diff(peaks) / fs  # Inter-beat interval (in seconds)

heart\_rate = 60 / np.mean(ibi)  # Beats per minute (BPM)

print(f"Estimated Heart Rate: {heart\_rate:.2f} BPM")

**Output:**

****

**Experiment No: 06**

**Title: Time Domain to Frequency Domain Analysis of a Signal Using Fast Fourier Transform (FFT)**

**Objectives:**

 To generate a signal composed of two different sine waves.

 To apply the Fast Fourier Transform (FFT) to the generated signal to analyze its frequency content.

 To visualize the signal in both the time domain and frequency domain.

 To observe how the Fourier Transform reveals the frequency components of a mixed signal.

**Theory:**

The **Fourier Transform** is a mathematical technique used to convert a time-domain signal into its frequency-domain representation. It decomposes a signal into its constituent frequencies, providing information on the amplitude and phase of each frequency component. The **Fast Fourier Transform (FFT)** is an efficient algorithm for computing the Discrete Fourier Transform (DFT), which is used in digital signal processing.

For a continuous time signal x(t), the Fourier Transform X(f) is defined as:

Where:

* x(t) is the time-domain signal,
* X(f) is the frequency-domain representation,
* f is the frequency,
* j is the imaginary unit.

In practice, the FFT computes the DFT, which provides discrete values of the frequency content of the signal.

The signal in this experiment is a combination of two sine waves with different frequencies. The FFT will allow us to analyze the frequencies present in this mixed signal.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

# Define a signal (sum of two sine waves)

fs = 1000  # Sampling frequency in Hz

t = np.linspace(0, 1, fs, endpoint=False)  # Time vector (1 second duration)

f1, f2 = 2, 50  # Frequencies of sine waves

signal = np.sin(2 \* np.pi \* f1 \* t) + np.sin(2 \* np.pi \* f2 \* t)  # Mixed signal

# Compute Fourier Transform using FFT

fft\_result = np.fft.fft(signal)  # Compute FFT

frequencies = np.fft.fftfreq(len(fft\_result), 1/fs)  # Frequency axis

# Compute Magnitude Spectrum

magnitude = np.abs(fft\_result) / len(signal)  # Normalize

magnitude = magnitude[:fs // 2]  # Keep only positive frequencies

frequencies = frequencies[:fs // 2]  # Corresponding frequency axis

# Plot the original signal

plt.figure(figsize=(12, 6))

plt.subplot(2, 1, 1)

plt.plot(t, signal, color="b", label="Original Signal")

plt.title("Time-Domain Signal")

plt.xlabel("Time (s)")

plt.ylabel("Amplitude")

plt.legend()

plt.grid()

# Plot the magnitude spectrum

plt.subplot(2, 1, 2)

plt.plot(frequencies, magnitude, color="r", label="Magnitude Spectrum")

plt.title("Fourier Transform (Frequency Domain)")

plt.xlabel("Frequency (Hz)")

plt.ylabel("Magnitude")

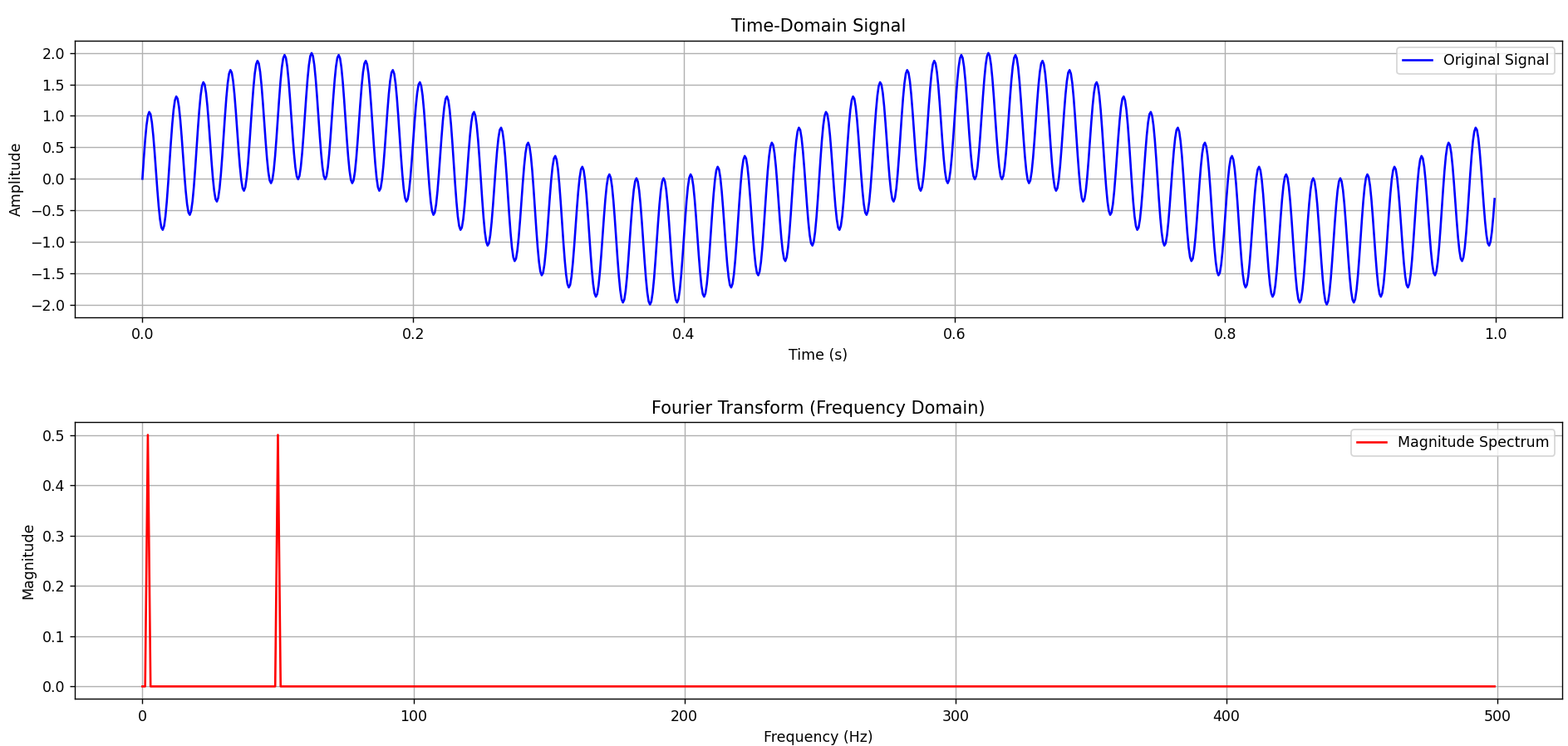
plt.legend()

plt.grid()

plt.tight\_layout()

plt.show()

**Output:**

****

**Experiment No: 07**

### Title: ****Signal Processing Using Discrete Fourier Transform (DFT) and Noise Removal Techniques****

**Objectives:**

* To implement and compute the **Discrete Fourier Transform (DFT)** of a signal using Python.
* To analyze the frequency spectrum of a signal by calculating the **frequency bins**.
* To perform **Noise Removal** from a signal by filtering out high-frequency components using **FFT** and **Inverse FFT**.

**Theory:**

The Discrete Fourier Transform (DFT) is a mathematical technique used to convert a signal from the time domain to the frequency domain. It represents the signal as a sum of sinusoidal functions, each with a specific frequency, amplitude, and phase.

The formula for DFT is given by:

X[k]=

Where:

* X[k] is the frequency-domain representation of the signal.
* x[n] is the time-domain signal.
* N is the total number of data points (samples).

The DFT provides the frequency spectrum of the signal, and the frequency bins indicate the different frequency components present in the signal.

The FFT (Fast Fourier Transform) is an optimized algorithm for computing DFT in less time, often used in practical applications.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

from scipy.fft import fft, ifft, fftfreq

# DFT Implementation

def DFT(x):

    """

    Compute the Discrete Fourier Transform (DFT) of a 1D signal.

    """

    N = len(x)

    X = np.zeros(N, dtype=complex)  # Output array (complex numbers)

    for k in range(N):  # Loop over frequency bins

        for n in range(N):  # Loop over time samples

            X[k] += x[n] \* np.exp(-2j \* np.pi \* k \* n / N)

    return X

# Create a sample signal (two sine waves)

Fs = 1000  # Sampling rate

T = 1 / Fs  # Sampling interval

t = np.linspace(0, 1, Fs, endpoint=False)  # 1 second duration

# Signal: Combination of 50 Hz and 120 Hz sine waves

f1, f2 = 50, 120

signal = np.sin(2 \* np.pi \* f1 \* t) + 0.5 \* np.sin(2 \* np.pi \* f2 \* t)

# Compute DFT

dft\_output = DFT(signal)

# Compute frequency bins

freqs = np.fft.fftfreq(len(dft\_output), T)

# Plot magnitude spectrum (single-sided)

plt.figure(figsize=(10, 5))

plt.plot(freqs[:Fs//2], np.abs(dft\_output[:Fs//2]))  # Single-sided spectrum

plt.title("DFT Frequency Spectrum")

plt.xlabel("Frequency (Hz)")

plt.ylabel("Magnitude")

plt.grid()

plt.show()

# Frequency Bins Calculation for Signal Cleaning

# Generate a sample audio signal with noise

pure\_signal\_freq = 440  # Frequency of pure sine wave (A4 note)

pure\_signal = np.sin(2 \* np.pi \* pure\_signal\_freq \* t)  # Pure sine wave

# Add random noise to the pure signal

noise = np.random.normal(0, 0.5, pure\_signal.shape)

noisy\_signal = pure\_signal + noise

# Apply FFT on noisy signal

fft\_signal = fft(noisy\_signal)

freqs = fftfreq(len(fft\_signal), T)  # Frequency bins

# Filter: Remove frequencies higher than 500 Hz (noise)

fft\_filtered = fft\_signal.copy()

fft\_filtered[np.abs(freqs) > 500] = 0  # Zero out high frequencies (noise)

# Apply Inverse FFT to get the cleaned signal

cleaned\_signal = ifft(fft\_filtered).real

# Plot the results

plt.figure(figsize=(12, 6))

# Plot 1: Original Pure Signal

plt.subplot(3, 1, 1)

plt.plot(t, pure\_signal, label="Original Signal (440 Hz)")

plt.legend()

plt.title("Original Pure Signal")

# Plot 2: Noisy Signal

plt.subplot(3, 1, 2)

plt.plot(t, noisy\_signal, label="Noisy Signal", color="red")

plt.legend()

plt.title("Noisy Signal")

# Plot 3: Cleaned Signal (after FFT filtering)

plt.subplot(3, 1, 3)

plt.plot(t, cleaned\_signal, label="Cleaned Signal (After FFT Filtering)", color="green")

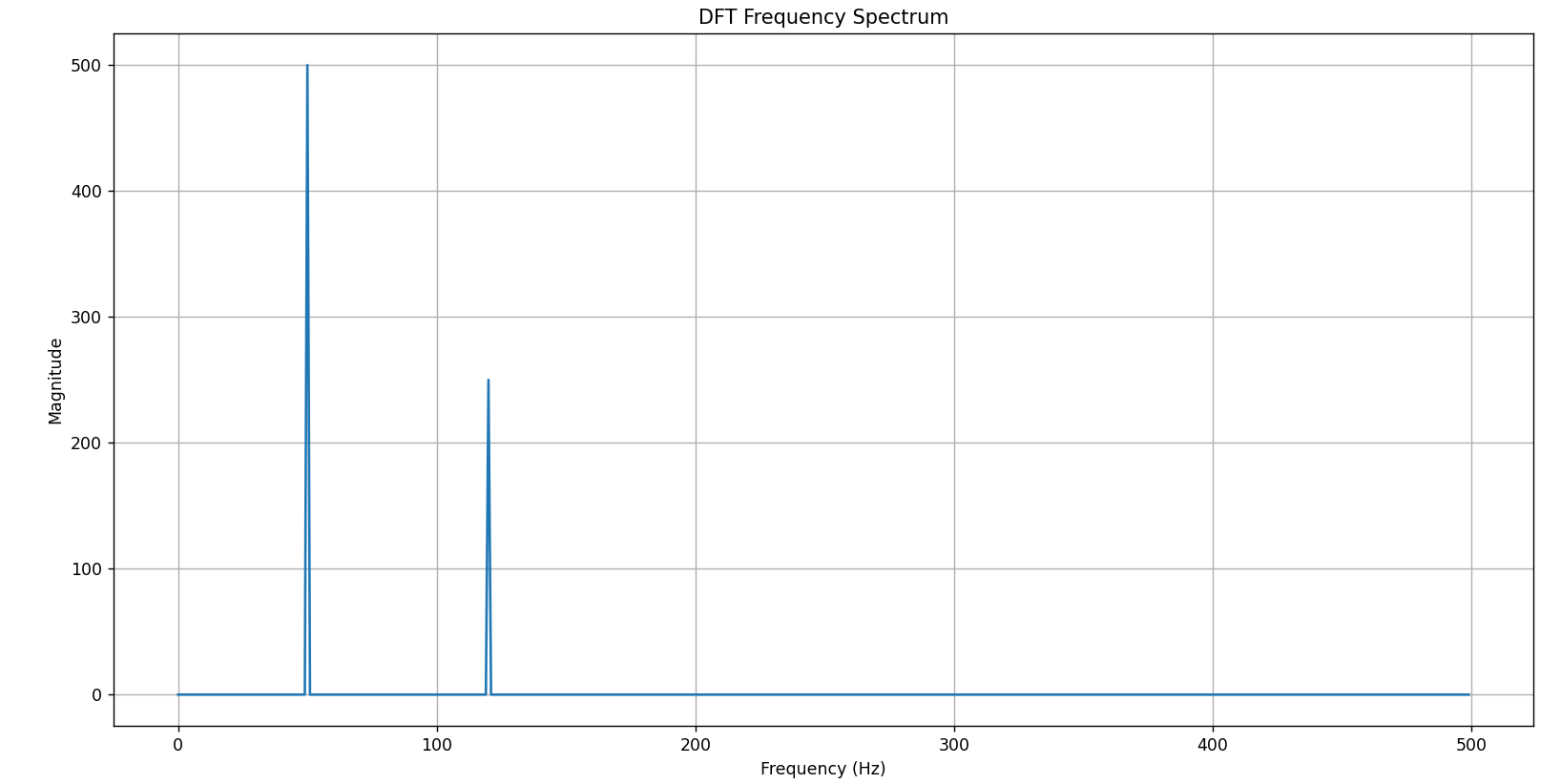
plt.legend()

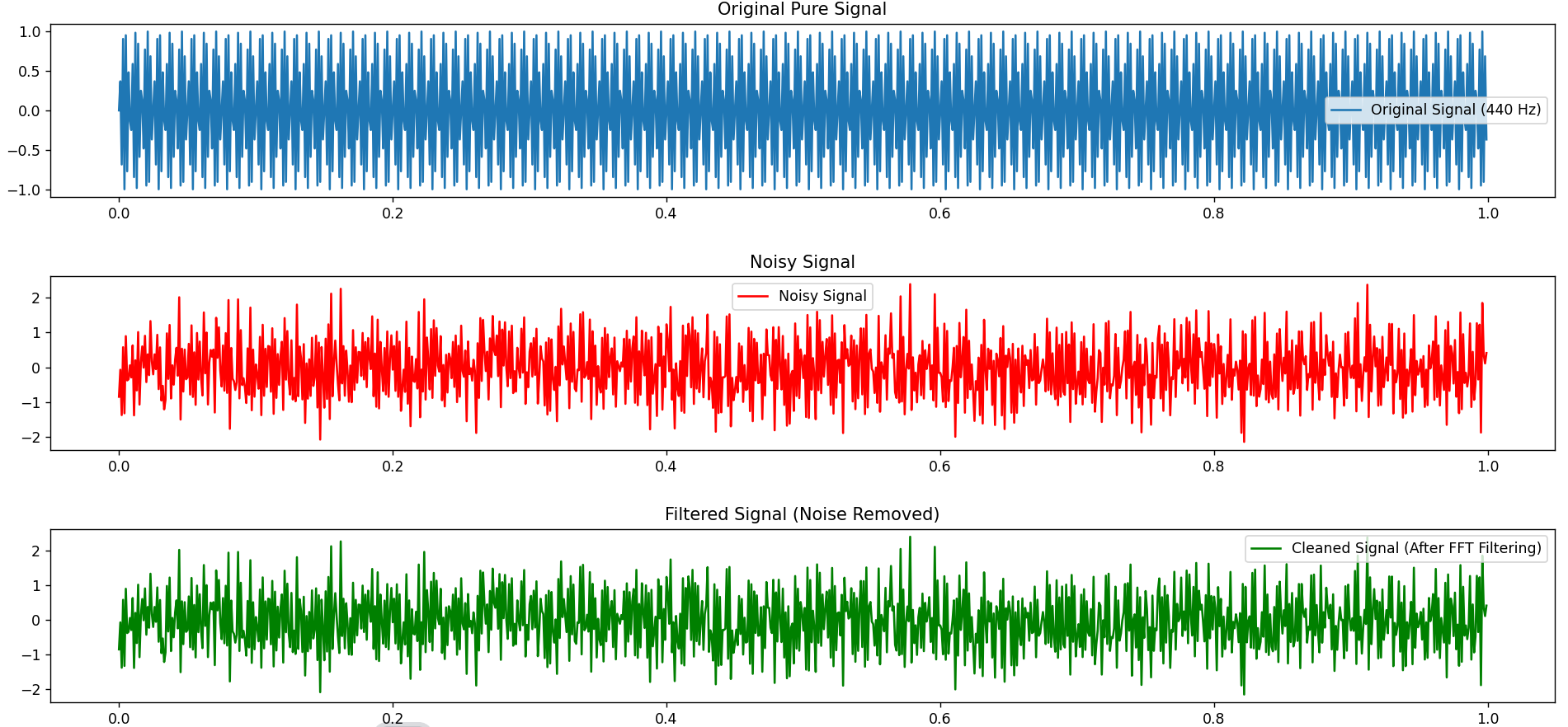
plt.title("Filtered Signal (Noise Removed)")

# Adjust layout and display the plots

plt.tight\_layout()

plt.show()

**Output:**

****