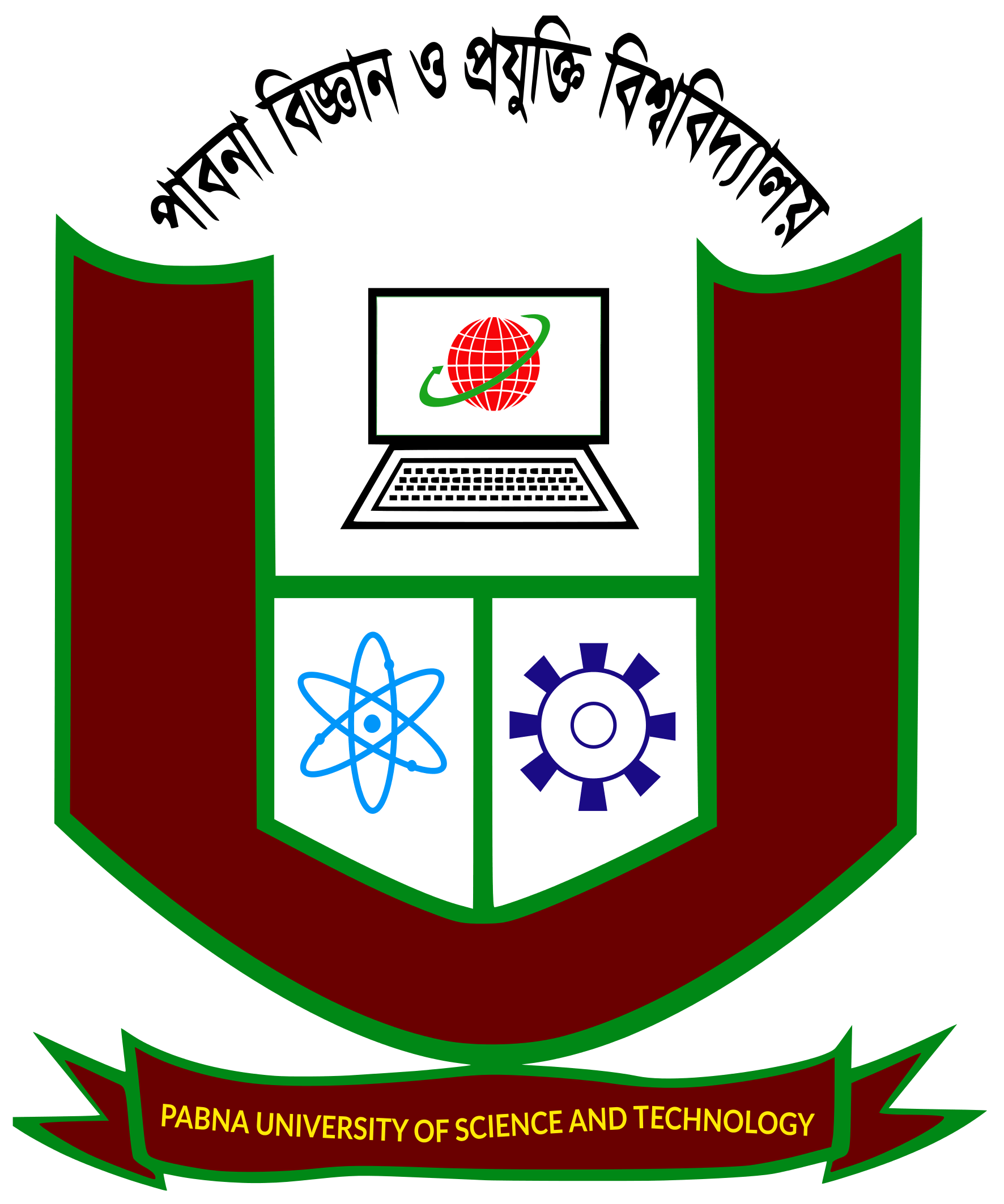
**PABNA UNIVERSITY OF SCIENCE AND TECHNOLOGY**



**Faculty of Engineering & Technology**

**Department of Information and Communication Engineering**

**LAB REPORT**

**Course Name: Signals and Systems Sessional**

**Course Code: ICE - 2204**

Submitted To:

**Dr. Md. Imran Hossain**

Associate Professor

Department of ICE

Pabna University of Science and Technology, Pabna

Submitted By:

**Mahmudul Islam**

Roll: 220639

Session: 2021-2022

2nd Year 2nd Semester

Department of ICE, PUST

**Index**

|  |  |
| --- | --- |
| **SL.** | **Problem Description** |
| **01** | **Basic Signal Operations: Addition, Multiplication, Shifting, Folding, and Scaling** |
| **02** | **Convolution of Discrete-Time Signals** |
| **03** | **Correlation of Discrete-Time Signals** |
| **04** | **Common Signal Sequences: Impulse, Step, and Ramp** |
| **05** | **Photoplethysmography (PPG) Signal Processing: Filtering, Feature Extraction, and Peak Detection** |
| **06** | **Fourier Series Decomposition and Harmonic Analysis** |
| **07** | **Fourier Transform of Discrete-Time Signals** |
| **08** | **Discrete Fourier Transform (DFT) - Manual Implementation, Applications, and Related Transforms** |

**Lab -1: Signal Operations**

**Title:** Basic Signal Operations: Addition, Multiplication, Shifting, Folding, and Scaling

**Theory:**

Digital signal processing involves manipulating discrete-time signals. Key operations include:

* **Addition:** Combining two signals sample by sample. y[n] = x1[n] + x2[n]
* **Multiplication:** Multiplying two signals sample by sample. y[n] = x1[n] \* x2[n]
* **Shifting:** Delaying or advancing a signal. y[n] = x[n - k] (delay if k > 0, advance if k < 0)
* **Folding (Reflection):** Reversing a signal in time. y[n] = x[-n]
* **Scaling:** Changing the amplitude of a signal. y[n] = a \* x[n]

**Source Code (Python): For Discrete time signal**

import numpy as np

import matplotlib.pyplot as plt

def generate\_signal(n, values):

return np.array(values[:len(n)])

n = np.arange(-5, 6)

x1 = generate\_signal(n, [1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1])

x2 = generate\_signal(n, [2, 3, 4, 5, 6, 7, 6, 5, 4, 3, 2])

# Signal Addition

addition = x1 + x2

# Signal Multiplication

multiplication = x1 \* x2

# Signal Shifting

shift\_right = np.roll(x1, 2)

shift\_left = np.roll(x1, -2)

# Signal Folding

folded = np.flip(x1)

# Signal Scaling

scaled = 2 \* x1

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

plt.subplot(3,2,1)

plt.stem(n, addition)

plt.title("Signal Addition")

plt.subplot(3,2,2)

plt.stem(n, multiplication)

plt.title("Signal Multiplication")

plt.subplot(3,2,3)

plt.stem(n, shift\_right)

plt.title("Right Shift")

plt.subplot(3,2,4)

plt.stem(n, shift\_left)

plt.title("Left Shift")

plt.subplot(3,2,5)

plt.stem(n, folded)

plt.title("Signal Folding")

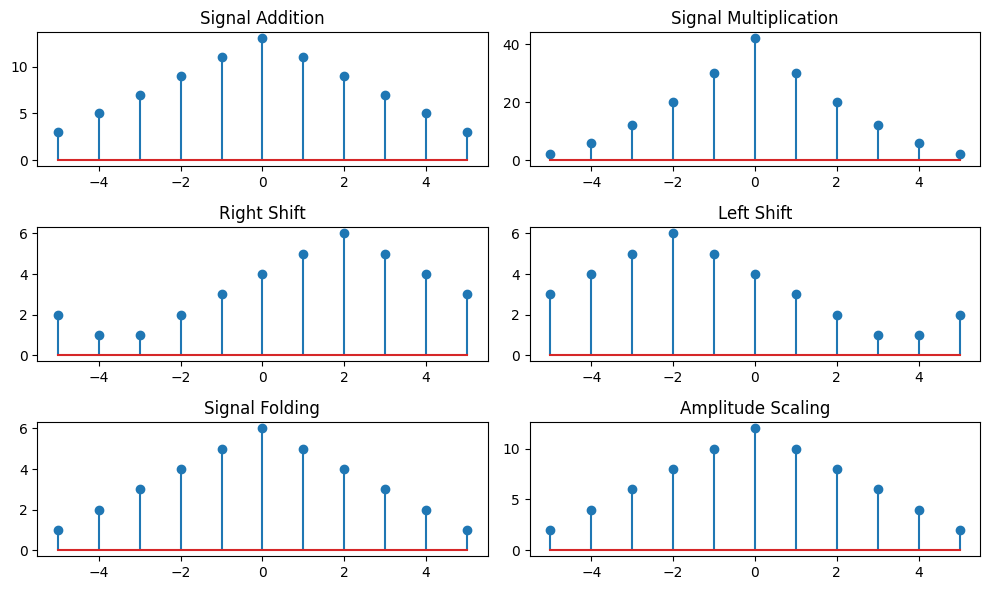
plt.subplot(3,2,6)

plt.stem(n, scaled)

plt.title("Amplitude Scaling")

plt.tight\_layout()

plt.show()  
**Output :**



**Source code : For Continuous Signal**

import numpy as np

import matplotlib.pyplot as plt

t = np.linspace(-5, 5, 1000)

x1 = np.sin(2 \* np.pi \* 0.2 \* t)

x2 = np.cos(2 \* np.pi \* 0.2 \* t)

# Signal Operations

addition = x1 + x2

multiplication = x1 \* x2

shift\_right = np.sin(2 \* np.pi \* 0.2 \* (t - 2))

shift\_left = np.sin(2 \* np.pi \* 0.2 \* (t + 2))

folded = np.sin(-2 \* np.pi \* 0.2 \* t)

scaled = 2 \* x1

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

plt.subplot(3,2,1)

plt.plot(t, addition)

plt.title("Signal Addition")

plt.subplot(3,2,2)

plt.plot(t, multiplication)

plt.title("Signal Multiplication")

plt.subplot(3,2,3)

plt.plot(t, shift\_right)

plt.title("Right Shift")

plt.subplot(3,2,4)

plt.plot(t, shift\_left)

plt.title("Left Shift")

plt.subplot(3,2,5)

plt.plot(t, folded)

plt.title("Signal Folding")

plt.subplot(3,2,6)

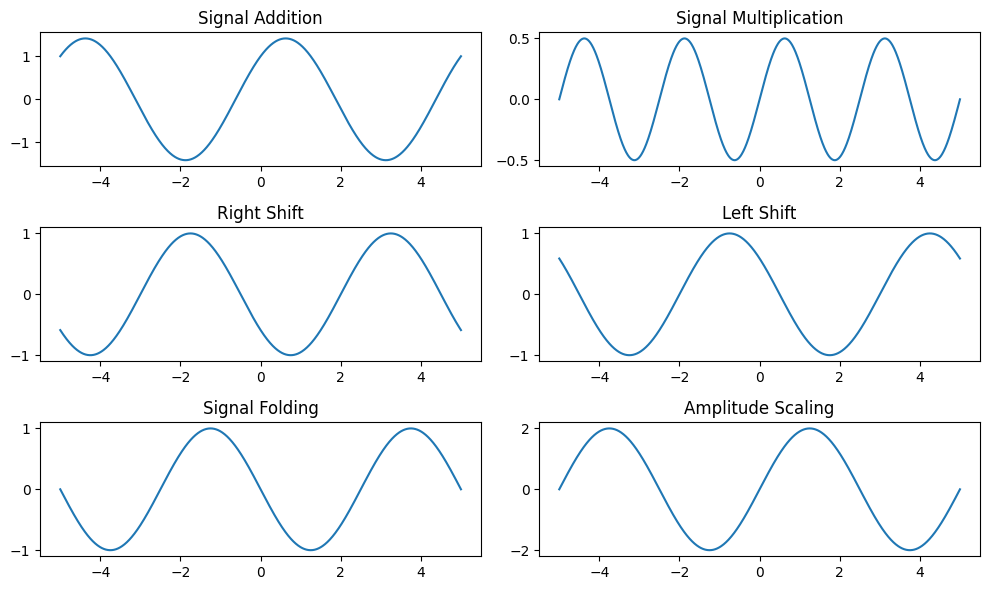
plt.plot(t, scaled)

plt.title("Amplitude Scaling")

plt.tight\_layout()

plt.show()

**Output :**

****

**Purpose:**

This lab demonstrates the fundamental operations performed on discrete-time signals, which are essential building blocks in digital signal processing.

**Lab - 2: Convolution**

**Title:** Convolution of Discrete-Time Signals

**Theory:**

Convolution is a fundamental operation in digital signal processing. It describes how the output of a system changes over time in response to an input signal. The convolution of two discrete-time signals x[n] and h[n] is given by:

y[n] =

Where:

* x[n] is the input signal.
* h[n] is the impulse response of the system.
* y[n] is the output signal.

Convolution can be visualized as "flipping" one signal, shifting it across the other, multiplying the overlapping portions, and summing the results for each shift.

**Source Code (Python):**

import numpy as np

import matplotlib.pyplot as plt

from scipy.linalg import toeplitz

from scipy.signal import deconvolve

def cross\_table\_convolution(x, h):

len\_y = len(x) + len(h) - 1

y = np.zeros(len\_y)

for i in range(len(x)):

for j in range(len(h)):

y[i + j] += x[i] \* h[j]

return y

def matrix\_convolution(x, h):

h\_padded = np.pad(h, (0, len(x)-1), 'constant')

x\_padded = np.pad(x, (0, len(h)-1), 'constant')

H = toeplitz(h\_padded, np.zeros(len(x) + len(h) - 1))

return H @ x\_padded

def perform\_deconvolution(y, h):

x\_est, remainder = deconvolve(y, h)

return x\_est

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

h = np.array([0, 1, 0.5])

cross\_table\_result = cross\_table\_convolution(x, h)

matrix\_result = matrix\_convolution(x, h)

deconv\_result = perform\_deconvolution(cross\_table\_result, h)

plt.figure(figsize=(10,4))

plt.subplot(1,3,1)

plt.stem(cross\_table\_result)

plt.title("Cross Table Method Convolution")

plt.subplot(1,3,2)

plt.stem(matrix\_result)

plt.title("Matrix Method Convolution")

plt.subplot(1,3,3)

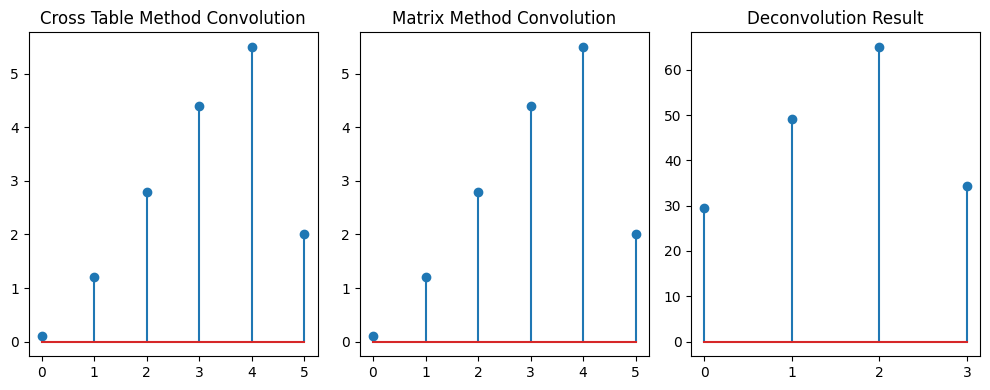
plt.stem(deconv\_result)

plt.title("Deconvolution Result")

plt.tight\_layout()

plt.show()

**Output**

****

**Purpose:** To understand and implement convolution operations in digital signal processing using different methods. These techniques are widely applied in filtering, system analysis, image processing, and signal restoration through deconvolution. The deconvolution approach must be carefully handled to avoid instability due to division by small or zero values.

**Lab - 3: Correlation**

**Title:** Correlation of Discrete-Time Signals

**Theory:**

Correlation measures the similarity between two signals. It helps determine the degree to which two signals are related or if one signal is present within another. There are two main types of correlation:

* **Cross-correlation:** Measures the similarity between two different signals.
* =
* **Autocorrelation:** Measures the similarity of a signal with itself at different time lags.
* =

**Source Code (Python):**

import numpy as np

import matplotlib.pyplot as plt

def autocorrelation(signal):

return np.correlate(signal, signal, mode='full')

def cross\_correlation(signal1, signal2):

return np.correlate(signal1, signal2, mode='full')

def add\_noise(signal, noise\_level=0.5):

noise = noise\_level \* np.random.randn(len(signal))

return signal + noise

x = np.array([1, 2, 3, 4, 5])

y = np.array([5, 4, 3, 2, 1])

auto\_result = autocorrelation(x)

cross\_result = cross\_correlation(x, y)

noisy\_signal = add\_noise(x)

noisy\_cross\_result = cross\_correlation(noisy\_signal, x)

# Plot results

plt.figure(figsize=(10,4))

plt.subplot(1,3,1)

plt.stem(auto\_result)

plt.title("Autocorrelation")

plt.subplot(1,3,2)

plt.stem(cross\_result)

plt.title("Cross-Correlation")

plt.subplot(1,3,3)

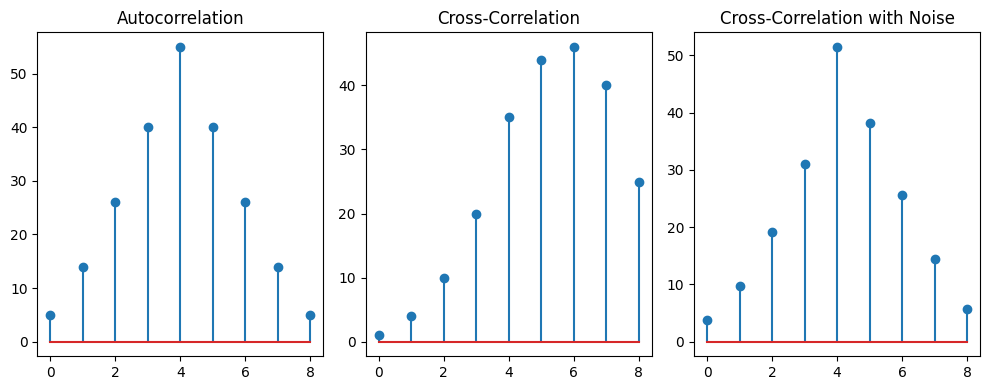
plt.stem(noisy\_cross\_result)

plt.title("Cross-Correlation with Noise")

plt.tight\_layout()

plt.show()

**Output :**



**Purpose:** To understand correlation techniques in signal processing. Autocorrelation helps detect periodic signals, while cross-correlation is useful for feature matching and noise reduction. Adding noise and correlating with the original signal demonstrates real-world signal recovery applications.

**Lab - 4: Signal Sequences**

**Title:** Common Signal Sequences: Impulse, Step, and Ramp

**Theory:**

Several fundamental signal sequences are frequently used in digital signal processing:

* **Impulse (Dirac Delta):** A signal with a value of 1 at n=0 and 0 elsewhere. Represented as δ[n].
* **Unit Step:** A signal with a value of 0 for n<0 and 1 for n>=0. Represented as u[n].
* **Ramp:** A signal that increases linearly with time. r[n] = n\*u[n] (where u[n] is the unit step).

**Source Code (Python):**

import numpy as np

import matplotlib.pyplot as plt

def impulse\_sequence(length):

signal = np.zeros(length)

signal[length // 2] = 1

def step\_sequence(length):

return np.ones(length)

def ramp\_sequence(length):

return np.arange(length)

def unit\_sample\_sequence(length, sample\_point):

signal = np.zeros(length)

signal[sample\_point] = 1

return signal

N = 10

impulse = impulse\_sequence(N)

step = step\_sequence(N)

ramp = ramp\_sequence(N)

unit\_sample = unit\_sample\_sequence(N, 5)

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

plt.subplot(2,2,1)

plt.stem(impulse)

plt.title("Impulse Sequence")

plt.subplot(2,2,2)

plt.stem(step)

plt.title("Step Sequence")

plt.subplot(2,2,3)

plt.stem(ramp)

plt.title("Ramp Sequence")

plt.subplot(2,2,4)

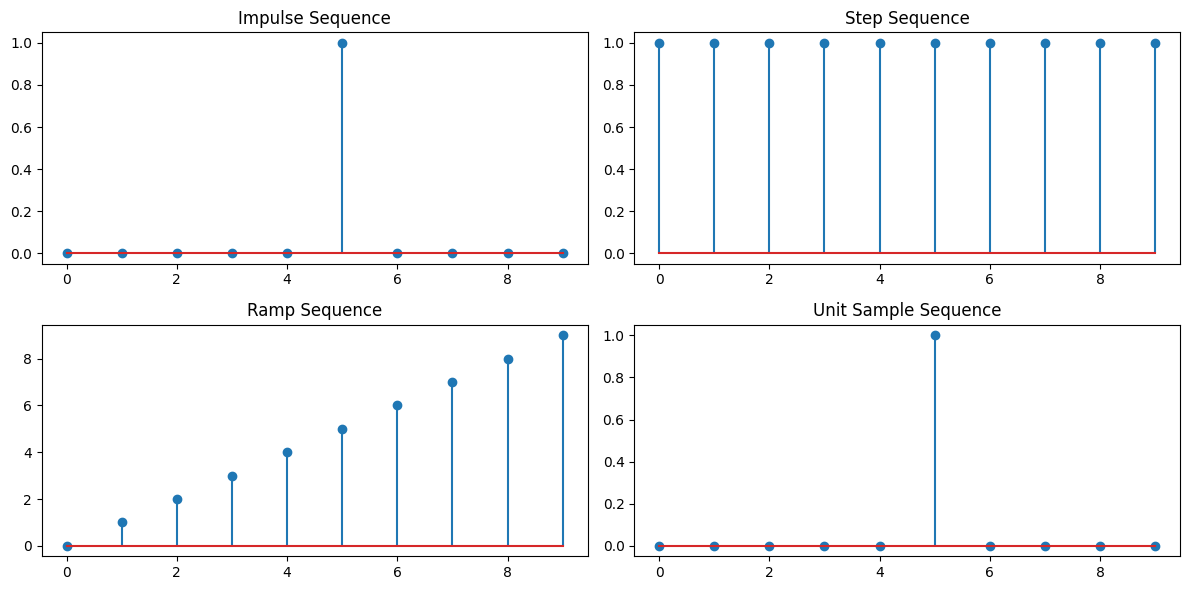
plt.stem(unit\_sample)

plt.title("Unit Sample Sequence")

plt.tight\_layout()

plt.show()

**Output :**

****

**Purpose:**

This lab introduces essential signal sequences used for testing and analyzing digital signal processing systems. The impulse response, for example, is crucial for characterizing LTI systems. The ramp function is used in various applications, including control systems and signal generation. The ramp generation has been corrected to ensure proper linear increase.

**Lab - 5 : PPG Signal Processing**

**Title:** PPG - Raw PPG, Noise Addition, Filtering, Normalization, Feature Extraction, Peak Detection

**Theory:** Photoplethysmography (PPG) is an optical technique used to detect blood volume changes in tissues. It is widely used in pulse oximeters and heart rate monitoring devices. The key steps in PPG signal processing include:

1. **Raw PPG Signal Acquisition**: The PPG signal is captured using an optical sensor that emits and detects light reflection from the skin.
2. **Noise Addition**: Simulated noise (e.g., motion artifacts, ambient light interference) is added to study filtering effects.
3. **Filtering**: A bandpass filter is applied to remove noise and retain the heart rate frequency range.
4. **Normalization**: The signal is scaled to a standard range for better analysis.
5. **Feature Extraction**: Key features such as heart rate and pulse wave characteristics are extracted.
6. **Peak Detection**: Peaks in the PPG waveform correspond to heartbeats and help determine heart rate.

**Source Code :**

import numpy as np

import matplotlib.pyplot as plt

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

def generate\_ppg\_signal(length=500, freq=1.2, noise\_level=0.2):

    t = np.linspace(0, 10, length)

    ppg = np.sin(2 \* np.pi \* freq \* t) + noise\_level \* np.random.randn(length)

    return t, ppg

def bandpass\_filter(signal, lowcut=0.5, highcut=3.0, fs=50, order=3):

    nyq = 0.5 \* fs

    low = lowcut / nyq

    high = highcut / nyq

    b, a = butter(order, [low, high], btype='band')

    return filtfilt(b, a, signal)

def normalize\_signal(signal):

    return (signal - np.min(signal)) / (np.max(signal) - np.min(signal))

def detect\_peaks(signal, distance=30):

    peaks, \_ = find\_peaks(signal, distance=distance)

    return peaks

t, raw\_ppg = generate\_ppg\_signal()

noisy\_ppg = raw\_ppg + 0.1 \* np.random.randn(len(raw\_ppg))

filtered\_ppg = bandpass\_filter(noisy\_ppg)

# Normalize the signal

normalized\_ppg = normalize\_signal(filtered\_ppg)

peaks = detect\_peaks(normalized\_ppg)

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

plt.subplot(6, 1, 1)

plt.plot(t, raw\_ppg, label="Raw PPG Signal")

plt.title("Raw PPG Signal")

plt.xlabel("Time (s)")

plt.ylabel("Amplitude")

plt.legend()

plt.subplot(6, 1, 2)

plt.plot(t, noisy\_ppg, label="Noisy PPG Signal", color='orange')

plt.title("Noisy PPG Signal")

plt.xlabel("Time (s)")

plt.ylabel("Amplitude")

plt.legend()

plt.subplot(6, 1, 3)

plt.plot(t, filtered\_ppg, label="Filtered PPG Signal", color='green')

plt.title("Filtered PPG Signal")

plt.xlabel("Time (s)")

plt.ylabel("Amplitude")

plt.legend()

plt.subplot(6, 1, 4)

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

plt.title("Normalized PPG Signal")

plt.xlabel("Time (s)")

plt.ylabel("Amplitude")

plt.legend()

plt.subplot(6, 1, 5)

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

plt.scatter(t[peaks], normalized\_ppg[peaks], color='red', label='Peaks')

plt.title("Peak Detection in Normalized PPG Signal")

plt.xlabel("Time (s)")

plt.ylabel("Amplitude")

plt.legend()

plt.subplot(6, 1, 6)

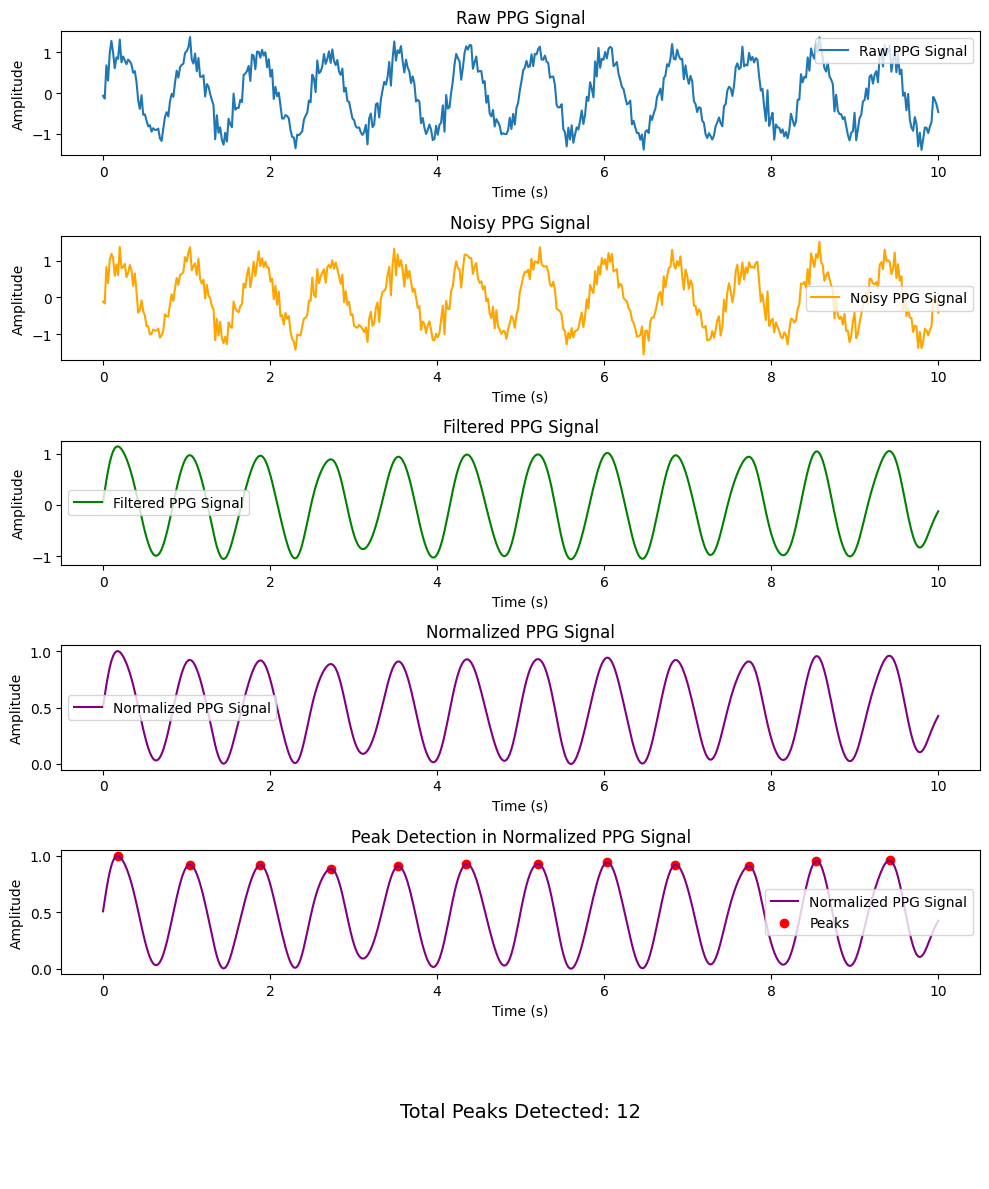
plt.text(0.5, 0.5, f"Total Peaks Detected: {len(peaks)}", fontsize=14, ha='center')

plt.axis('off')

plt.tight\_layout()

plt.show()

**Output :**

****

**Lab - 6: Fourier Series**

**Title:** Fourier Series Decomposition and Harmonic Analysis

**Theory:**

The Fourier series decomposes a periodic signal into a sum of sine and cosine waves (or complex exponentials) at different frequencies. Each sine/cosine component is called a harmonic. The Fourier series represents the signal in the frequency domain, revealing its spectral content.

For a periodic signal x(t) with period T, the Fourier series is given by:

x(t) = /2 +

where = 2π/T is the fundamental angular frequency, and an and bn are the Fourier coefficients.

**Source Code (Python):**

import numpy as np

import matplotlib.pyplot as plt

def fourier\_series\_decomposition(T, N\_terms=10):

t = np.linspace(0, 2 \* np.pi, 1000)

f = np.sign(np.sin(t)) # Square wave

reconstructed\_signal = np.zeros\_like(t)

# Create a new plot for harmonics

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

for n in range(1, N\_terms + 1, 2): # Odd harmonics

harmonic = (4 / (np.pi \* n)) \* np.sin(n \* t)

reconstructed\_signal += harmonic

# Plot each harmonic on the same figure

plt.plot(t, harmonic, label=f"Harmonic {n}")

plt.plot(t, f, label="Original Square Wave", linestyle='dashed')

plt.plot(t, reconstructed\_signal, label="Reconstructed Signal", color='red')

plt.title("Fourier Series Approximation of Square Wave")

plt.xlabel("Time")

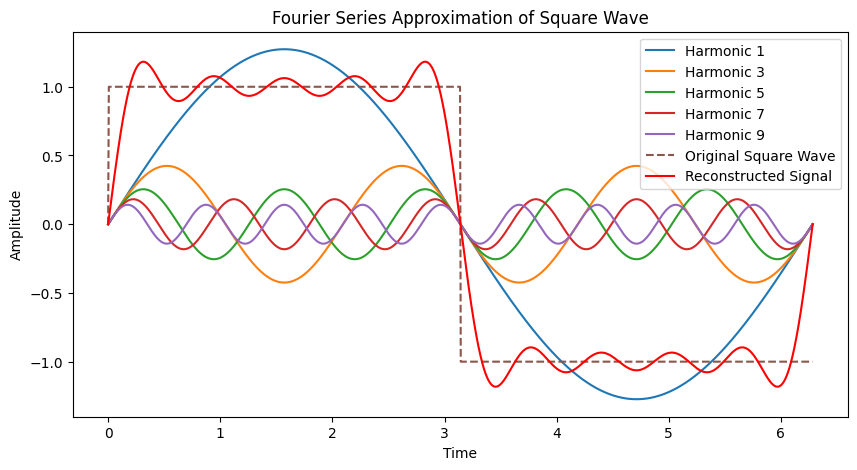
plt.ylabel("Amplitude")

plt.legend()

plt.show()

fourier\_series\_decomposition(T=2\*np.pi, N\_terms=10)

**Output :**



**Lab Report 7: Fourier Transform**

**Title: Fourier Transform of Discrete-Time Signals**

**Theory:**

The Fourier Transform (FT) extends the concept of the Fourier series to non-periodic signals. It decomposes a signal into its constituent frequencies, providing a frequency-domain representation. For a discrete-time signal x[n], the Discrete-Time Fourier Transform (DTFT) is:

X(ω) =

where ω is the angular frequency (in radians). The FT is a complex-valued function of frequency. We often look at the magnitude spectrum (|X(ω)|) and phase spectrum (angle(X(ω))).

**Source Code (Python):**

import numpy as np

import matplotlib.pyplot as plt

t = np.linspace(0, 1, 1000) # Time from 0 to 1 second

f1 = 50 # Frequency of the sine wave

x\_t = np.sin(2 \* np.pi \* f1 \* t)

X\_f = np.fft.fft(x\_t)

f = np.fft.fftfreq(len(t), t[1] - t[0])

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

plt.subplot(2, 1, 1)

plt.plot(t, x\_t)

plt.title("Time-Domain Signal")

plt.xlabel("Time [s]")

plt.ylabel("Amplitude")

plt.subplot(2, 1, 2)

plt.plot(f, np.abs(X\_f))

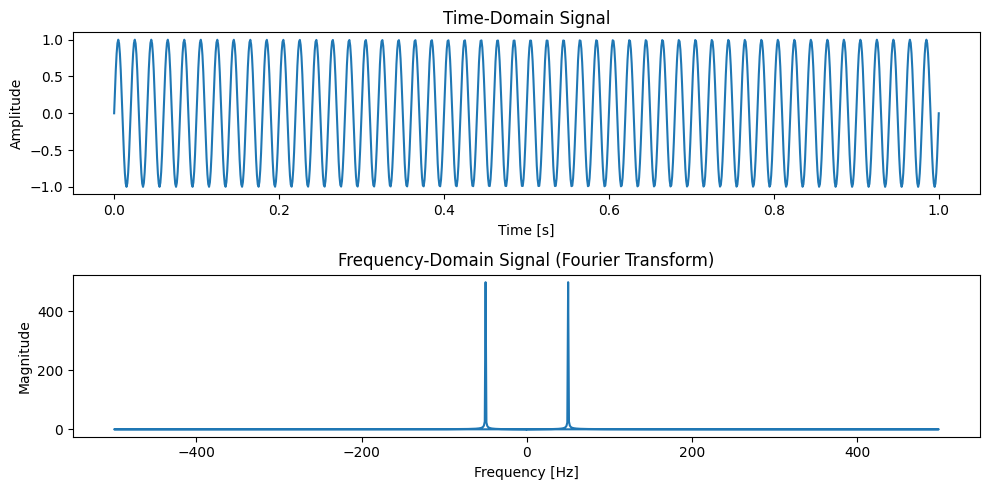
plt.title("Frequency-Domain Signal (Fourier Transform)")

plt.xlabel("Frequency [Hz]")

plt.ylabel("Magnitude")

plt.tight\_layout()

plt.show()

**Output:**

**Lab Report 8: Discrete Fourier Transform (DFT)**

**Title:** Discrete Fourier Transform (DFT) - Manual Implementation, Applications, and Related Transforms

**Theory:**

The Discrete Fourier Transform (DFT) is a fundamental tool for analyzing the frequency content of discrete-time signals. It decomposes a finite-length sequence into a sum of complex exponentials at discrete frequencies. The DFT is defined as:

X[k] = for k = 0, 1, ..., N-1

Where:

* x[n] is the input sequence.
* X[k] are the DFT coefficients.
* N is the length of the sequence.

**Source Code (Python):**

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 \* k \* n / N)

    return X

Fs = 1000

T = 1 / Fs

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

f1, f2, f3 = 50, 120, 150

signal = np.sin(2 \* np.pi \* f1 \* t) + 0.5 \* np.sin(2 \* np.pi \* f2 \* t) + 1.5 \* np.sin(2 \* np.pi \* f3 \* t)

plt.plot(t, signal)

plt.title("Time domain signal")

plt.xlabel("Time (s)")

plt.ylabel("Amplitude")

plt.grid()

plt.show()

dft\_output = DFT(signal)

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

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()

pure\_signal = np.sin(2 \* np.pi \* 440 \* t)

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

noisy\_signal = pure\_signal + noise

fft\_signal = np.fft.fft(noisy\_signal)

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

fft\_filtered = fft\_signal.copy()

fft\_filtered[np.abs(freqs) > 500] = 0

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

# Plot results: Original, Noisy, and Cleaned Signals

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

plt.subplot(3, 1, 1)

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

plt.legend()

plt.title("Original Pure Signal")

plt.subplot(3, 1, 2)

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

plt.legend()

plt.title("Noisy Signal")

plt.subplot(3, 1, 3)

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

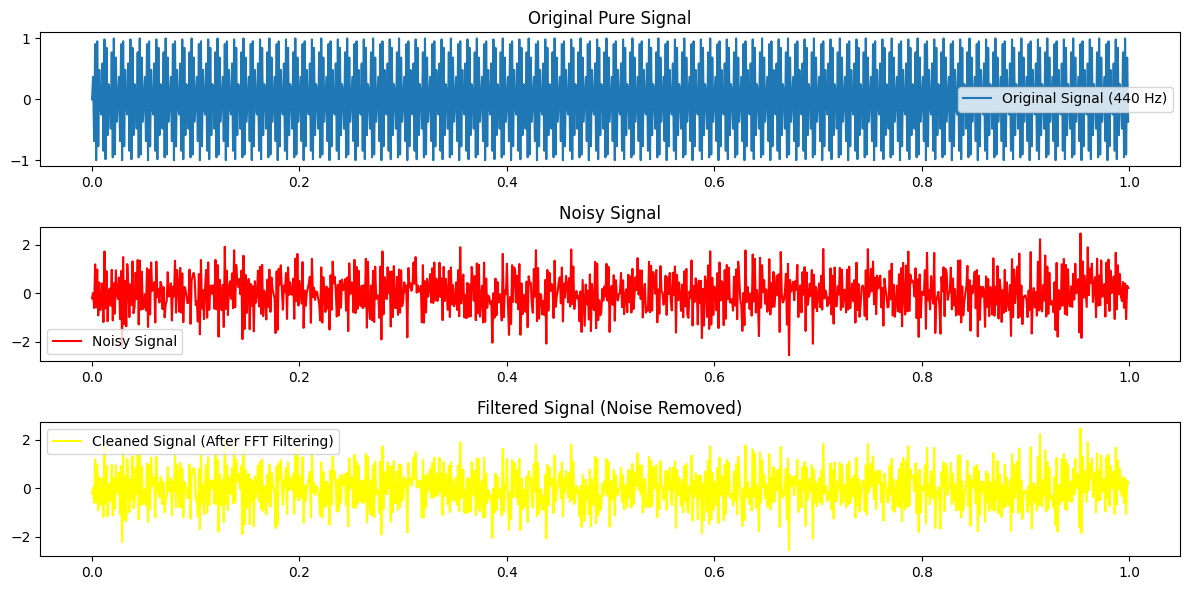
plt.legend()

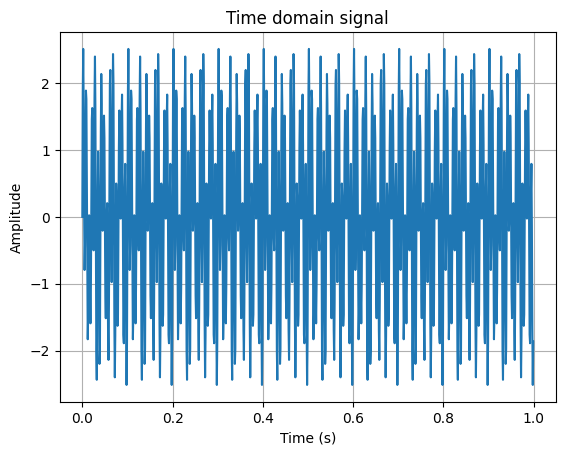
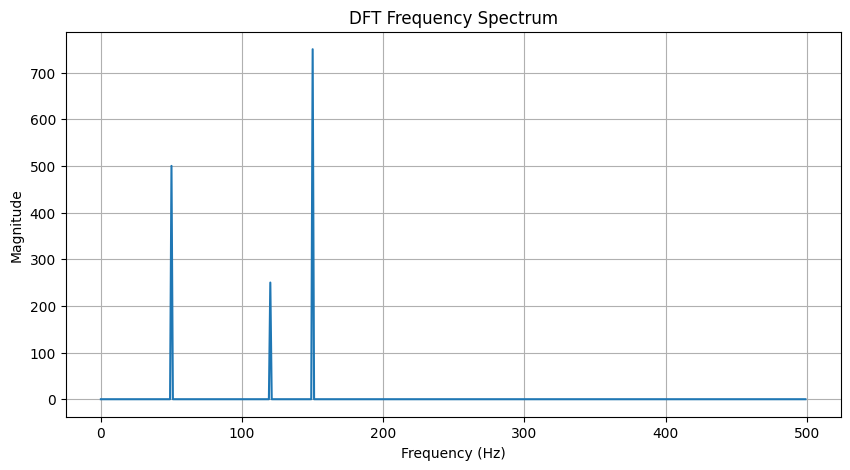
plt.title("Filtered Signal (Noise Removed)")

plt.tight\_layout()

plt.show()

**Output :**

****

****

**Purpose:**

* To compute the Discrete Fourier Transform (DFT) of a 1D signal using the standard formula.
* To apply FFT for efficient computation.
* To use DFT and FFT for noise removal in a signal.