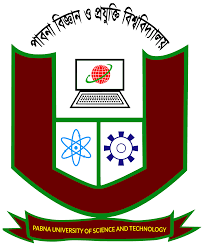
**Pabna University of Science and Technology**



**Faculty of Engineering and Technology**

**Department of Information and Communication Engineering**

**Lab Report**

Course Code: **ICE-2204**

Course title: **Signals And Systems Sessional**

|  |  |
| --- | --- |
| **Submitted By:**  **Name:** Rahat Mahbub  **Roll:** 220641  **Reg:** 1065488  **Session:**2021-2022  2 nd Year 2 nd semester  Department of Information and  Communication Engineering,  PUST | **Submitted To:**  **Dr. MD. Imran Hossain**  Associate Professor,  Department of  Information and Communication Engineering,  Pabna University of Science And Technology, Pabna. |

**Date of Submission: 03/03/2025**

**Laboratory Problem Index**

**Course:** ICE-2204 - Signals And Systems Sessional  
**Institution:** Pabna University of Science and Technology  
**Session:** 2021-2022

**Index of Laboratory Problems**

| **Sl. No** | **Problem Description** |
| --- | --- |
| **1** | Signal Operations (Addition, multiplication, Scaling , Folding , Shifting) |
| **2** | Basic Discrete-Time Signals (Impulse, Step, and Ramp) |
| **3** | Convolution Analysis of Sinusoidal Signals |
| **4** | Autocorrelation and Cross-Correlation Analysis |
| **5** | Heart Rate Estimation using PPG Signal Processing |
| **6** | Discrete Fourier Transform (DFT) and Inverse DFT (IDFT) Analysis |
| **7** | Discrete Fourier Transform (DFT) and Inverse DFT (IDFT) Analysis  with Frequency Bins |

**Program No: 01**

**Program Name:** Signal Operations using (Addition, multiplication, Scaling , Folding , Shifting)

**Theory:** Signal processing involves various operations like addition, multiplication, scaling, shifting, and folding. These operations are essential in digital signal processing (DSP) for analyzing and modifying signals. The program demonstrates these fundamental operations using Python's NumPy and Matplotlib libraries.

* **Signal Addition**: The sum of two discrete signals at each time index.
* **Signal Multiplication**: The element-wise product of two signals.
* **Signal Scaling**: Multiplying a signal by a constant factor.
* **Signal Shifting**: Moving the signal forward or backward along the time axis.
* **Signal Folding**: Reversing the signal around the origin.

**Code:**

import numpy as np

import matplotlib.pyplot as plt

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 n + shift

def signal\_folding(x):

return np.flip(x)

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

x1 = np.array([1, 2, 3, 4, 5])

x2 = np.array([5, 4, 3, 2, 1])

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)

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

plt.subplot(4, 2, 1)

plt.stem(n, x1)

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.title("Original Signal x1")

plt.grid()

plt.subplot(4, 2, 2)

plt.stem(n, x2)

plt.xlabel("Time ")

plt.ylabel("Amplitude")

plt.title("Original Signal x2")

plt.grid()

plt.subplot(4, 2, 3)

plt.stem(n, added\_signal)

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.title("Signal Addition")

plt.grid()

plt.subplot(4, 2, 4)

plt.stem(n, multiplied\_signal)

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.title("Signal Multiplication")

plt.grid()

plt.subplot(4, 2, 5)

plt.stem(n, scaled\_signal)

plt.xlabel("Time")

plt.ylabel("Amplitude")

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

plt.grid()

plt.subplot(4, 2, 6)

plt.stem(shifted\_signal1, x1)

plt.xlabel("Time")

plt.ylabel("Amplitude")

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

plt.grid()

plt.subplot(4, 2, 7)

plt.stem(shifted\_signal2, x1)

plt.xlabel("Time")

plt.ylabel("Amplitude")

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

plt.grid()

plt.subplot(4, 2, 8)

plt.stem(n, folded\_signal)

plt.xlabel("Time")

plt.ylabel("Amplitude")

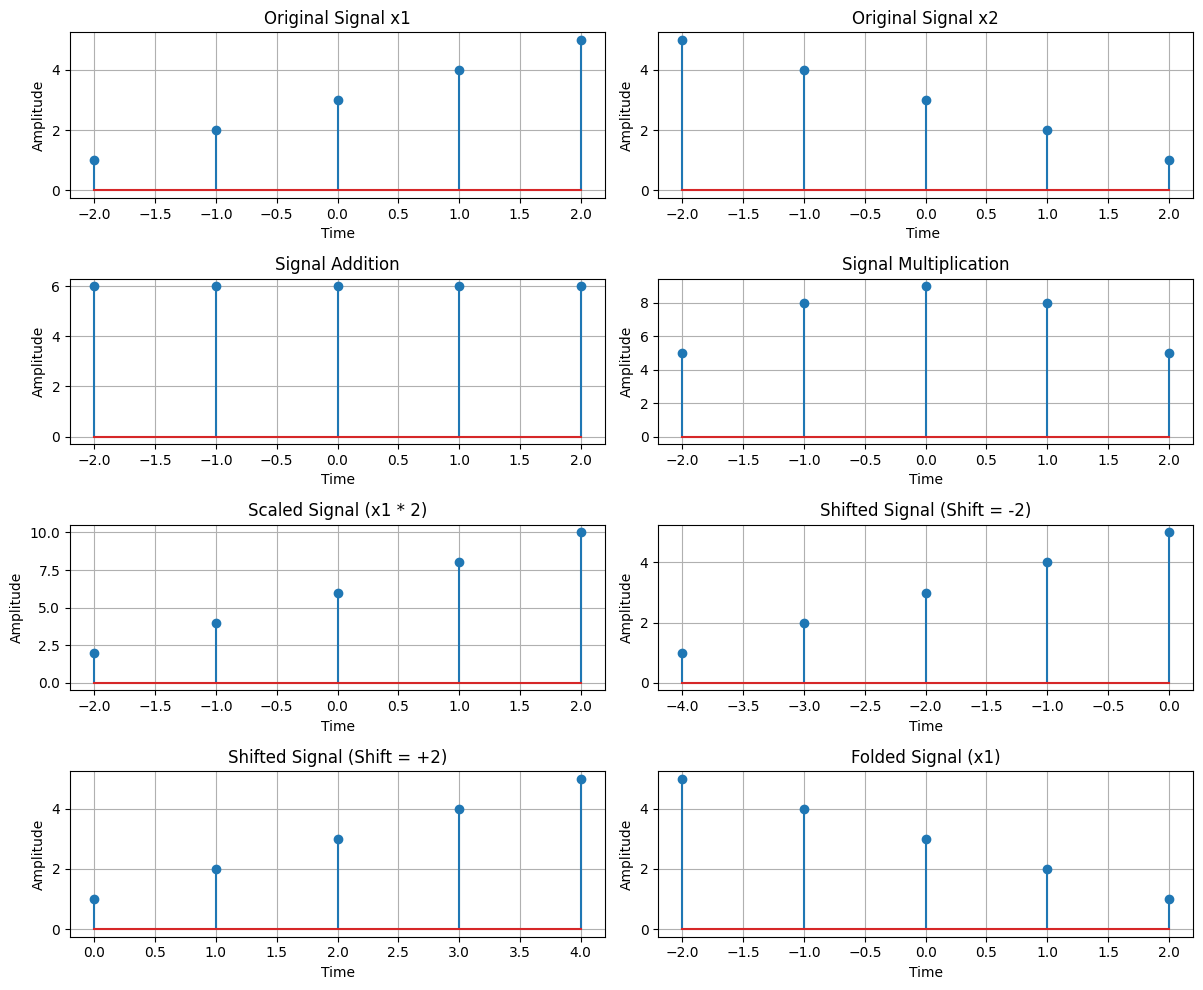
plt.title("Folded Signal (x1)")

plt.grid()

plt.tight\_layout()

plt.show()

**Input and Output:**



**Discussion:** This program demonstrates key operations in digital signal processing. Using NumPy, we efficiently perform element-wise operations on signals. Matplotlib's stem() function is used to visualize discrete signals clearly. These operations are essential in fields like communications, image processing, and machine learning. The program can be extended to handle continuous signals and more complex transformations.

**Program No: 2**

**Program Name:** Basic Discrete-Time Signals (Impulse, Step, and Ramp)

**Theory:** Basic discrete-time signals play a crucial role in signal processing. Three fundamental signals used in digital signal processing (DSP) are:

* **Impulse Signal**: Also known as the unit impulse or Dirac delta function, this signal is 1 at n=0n = 0 and 0 elsewhere.
* **Step Signal**: This signal remains 0 for negative values and 1 for non-negative values.
* **Ramp Signal**: This signal increases linearly for non-negative values and remains 0 for negative values.

These signals form the building blocks for more complex signal processing operations.

**Code:**

import numpy as np

import matplotlib.pyplot as plt

# Define the range

n = np.arange(-10, 11)

def impulse\_signal(n):

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

def step\_signal(n):

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

def ramp\_signal(n):

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

# Generate signals

impulse = impulse\_signal(n)

step = step\_signal(n)

ramp = ramp\_signal(n)

# Plot signals

plt.figure(figsize=(12, 4))

plt.subplot(1, 3, 1)

plt.stem(n, impulse)

plt.title("Impulse Signal")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

plt.subplot(1, 3, 2)

plt.stem(n, step)

plt.title("Step Signal")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

plt.subplot(1, 3, 3)

plt.stem(n, ramp)

plt.title("Ramp Signal")

plt.xlabel("n")

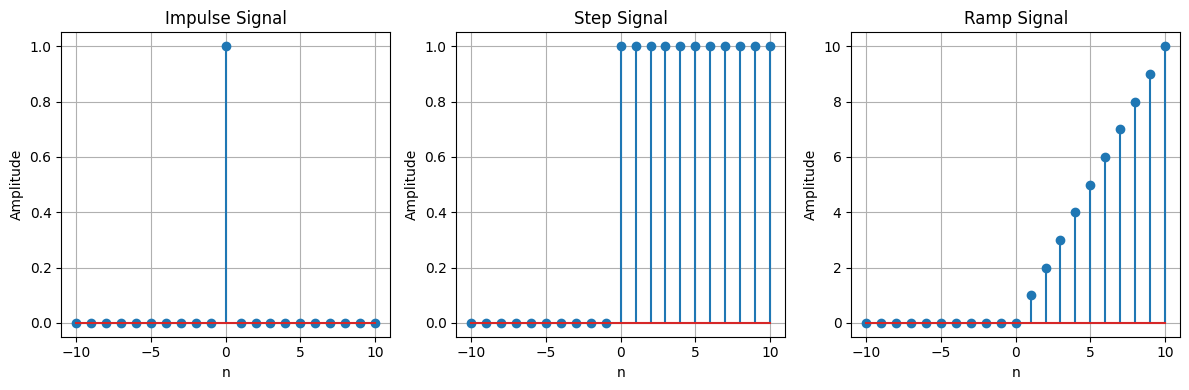
plt.ylabel("Amplitude")

plt.grid()

plt.tight\_layout()

plt.show()

**Input and Output:**



**Discussion:** This program demonstrates how fundamental discrete-time signals can be generated using NumPy. The where() function is used to define each signal efficiently. Matplotlib’s stem() function helps visualize discrete signals clearly. These signals are widely used in system analysis, convolution operations, and digital filtering. This program provides a foundation for more advanced DSP concepts.

**Program No: 3**

**Program Name:** Convolution Analysis of Sinusoidal Signals

**Theory:** Convolution is a mathematical operation used in signal processing to determine how one signal modifies another. It is fundamental in filtering, system analysis, and pattern recognition. This program explores convolution in three different scenarios:

* **Autoconvolution**: Convolution of a signal with itself, revealing inherent periodicity and structure.
* **Convolution with a Shifted Version**: Demonstrates how time shifts affect convolution results.
* **Convolution with a Noisy Signal**: Shows the effect of random noise on the convolution output.

The convolution operation is performed using the convolve() function from the scipy.signal library.

**Code:**

import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import convolve

# Function to compute convolution

def compute\_convolution(signal1, signal2):

return convolve(signal1, signal2, mode='full', method='auto')

# Parameters

fs = 1000 # Sampling frequency in Hz

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

freq = 5 # Frequency of the sine wave

# Generate sinusoidal signal

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

# Compute auto-convolution

conv\_auto = compute\_convolution(sin\_signal, sin\_signal)

# Create a shifted version of the signal

signal1 = sin\_signal

signal2 = np.roll(signal1, 100)

# Compute convolution with the shifted signal

conv\_shifted = compute\_convolution(signal1, signal2)

# Generate noisy signal

noise = np.random.normal(0, 0.5, fs)

noisy\_signal = signal1 + noise

# Compute convolution with noisy signal

conv\_noisy = compute\_convolution(signal1, noisy\_signal)

# Plot results

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

plt.subplot(3, 1, 1)

plt.plot(conv\_auto)

plt.title("Autoconvolution of a Sinusoidal Signal")

plt.xlabel("Samples")

plt.ylabel("Convolution Output")

plt.grid(True)

plt.subplot(3, 1, 2)

plt.plot(conv\_shifted)

plt.title("Convolution between Signal and Shifted Version")

plt.xlabel("Samples")

plt.ylabel("Convolution Output")

plt.grid(True)

plt.subplot(3, 1, 3)

plt.plot(conv\_noisy)

plt.title("Convolution with Noisy Signal")

plt.xlabel("Samples")

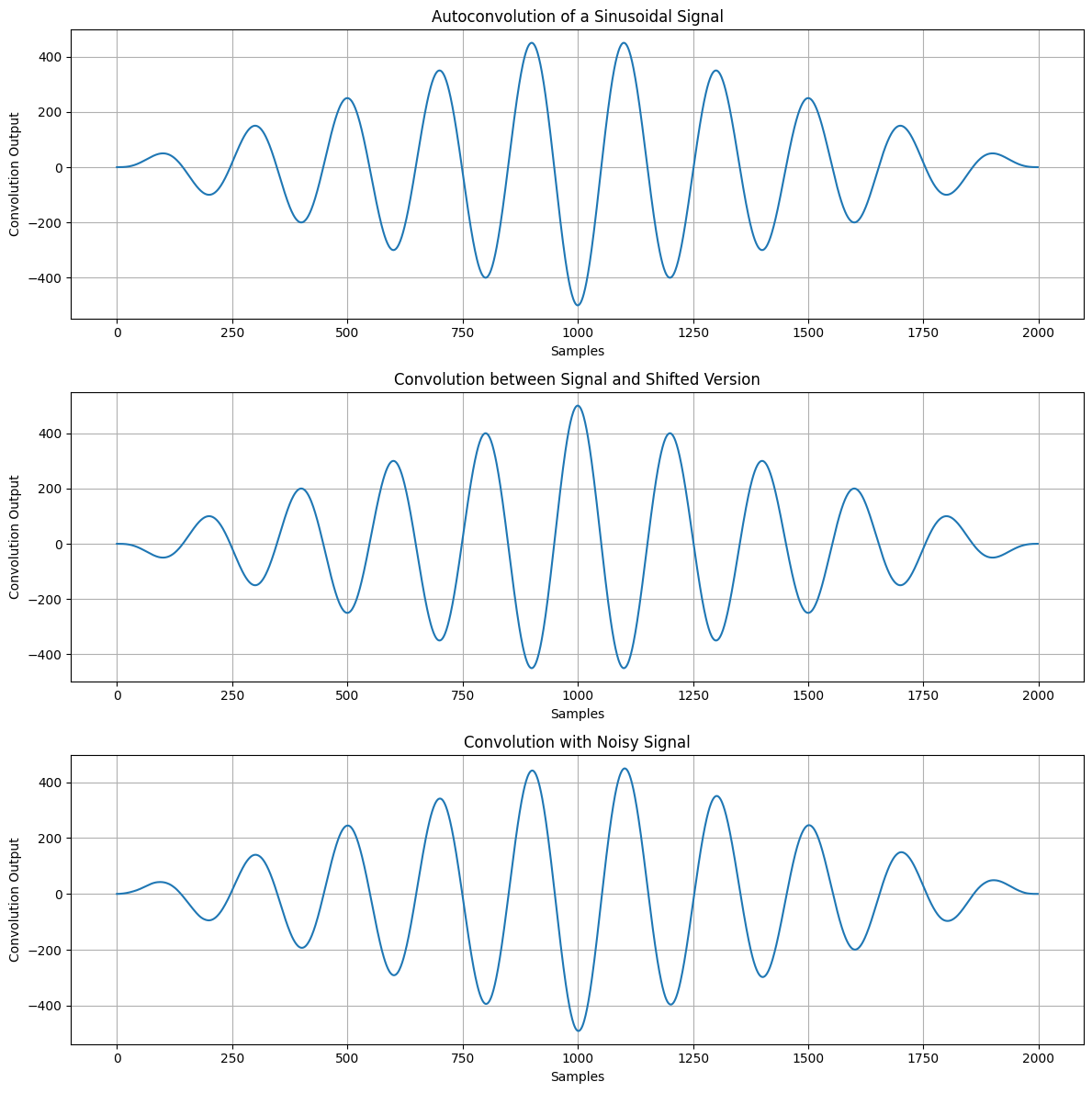
plt.ylabel("Convolution Output")

plt.grid(True)

plt.tight\_layout()

plt.show()

**Input and Output:**



**Discussion:** This program provides an in-depth look at convolution operations in signal processing. By analyzing a pure sinusoidal signal, its shifted version, and a noisy variant, we observe the impact of signal modifications on convolution results. The use of scipy.signal.convolve ensures efficient computation, and the graphical representation helps visualize key concepts. This approach is valuable in filtering applications, pattern detection, and system response analysis.

**Program No: 4**

**Program Name:** Autocorrelation and Cross-Correlation Analysis

**Theory:** Correlation is a fundamental operation in signal processing used to measure similarity between signals:

* **Autocorrelation**: Measures similarity of a signal with a delayed version of itself, often used to detect periodicity.
* **Cross-Correlation**: Measures similarity between two different signals, useful in signal matching and detection.

This program computes and visualizes autocorrelation and cross-correlation of sinusoidal signals, both with and without noise.

**Code:**

import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import correlate, correlation\_lags

def compute\_autocorrelation(signal):

auto\_corr = correlate(signal, signal, mode='full', method='auto')

lags = correlation\_lags(len(signal), len(signal), mode='full')

return auto\_corr, lags

def compute\_cross\_correlation(signal1, signal2):

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

lags = correlation\_lags(len(signal1), len(signal2), mode='full')

return cross\_corr, lags

fs = 1000 # Sampling frequency in Hz

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

freq = 5 # Frequency of the sine wave

# Generate sinusoidal signal

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

# Compute autocorrelation

auto\_corr, lags\_auto = compute\_autocorrelation(sin\_signal)

# Create a shifted version of the signal

signal1 = sin\_signal

signal2 = np.roll(signal1, 100)

# Compute cross-correlation

cross\_corr, lags\_cross = compute\_cross\_correlation(signal1, signal2)

# Generate noisy signal

noise = np.random.normal(0, 0.5, fs)

noisy\_signal = signal1 + noise

# Compute cross-correlation with noisy signal

cross\_corr\_noise, lags\_noise = compute\_cross\_correlation(signal1, noisy\_signal)

# Plot results

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

plt.subplot(3, 1, 1)

plt.plot(lags\_auto, auto\_corr)

plt.title("Autocorrelation of a Sinusoidal Signal")

plt.xlabel("Lag")

plt.ylabel("Autocorrelation")

plt.grid()

plt.subplot(3, 1, 2)

plt.plot(lags\_cross, cross\_corr)

plt.title("Cross-Correlation between Two Signals")

plt.xlabel("Lag")

plt.ylabel("Cross-Correlation")

plt.grid()

plt.subplot(3, 1, 3)

plt.plot(lags\_noise, cross\_corr\_noise)

plt.title("Cross-Correlation with Noisy Signal")

plt.xlabel("Lag")

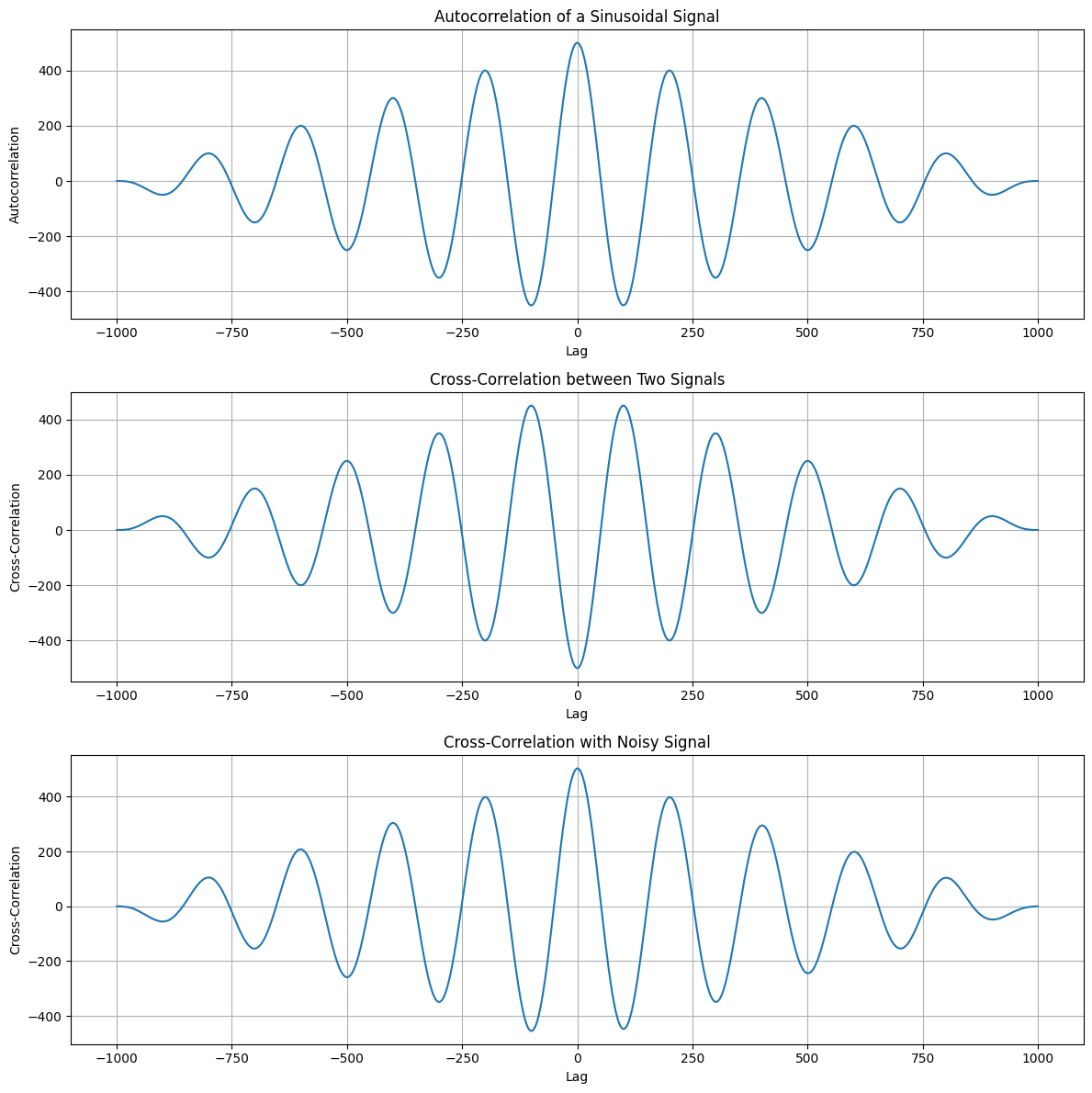
plt.ylabel("Cross-Correlation")

plt.grid()

plt.tight\_layout()

plt.show()

**Input and Output:**



**Discussion:** This program highlights the importance of correlation in signal processing. Autocorrelation helps in detecting repetitive patterns, while cross-correlation is useful in identifying shifts and similarities between signals. The presence of noise affects correlation, making it a valuable tool in signal denoising and pattern recognition applications.

**Program No: 5**

**Program Name:** Heart Rate Estimation using PPG Signal Processing

**Theory:** Photoplethysmography (PPG) is a non-invasive technique used to measure blood volume changes in the skin, often used in heart rate monitoring. The process involves:

* **Bandpass Filtering**: Removes unwanted noise and extracts relevant frequency components.
* **Peak Detection**: Identifies heartbeat peaks in the PPG signal.
* **Heart Rate Estimation**: Computes heart rate based on detected peak intervals.

This program simulates a PPG signal, processes it, detects peaks, and estimates the heart rate.

**Code:**

import numpy as np

import scipy.signal as signal

import matplotlib.pyplot as plt

def bandpass\_filter(data, fs=100):

b, a = signal.butter(4, [0.5 / (0.5 \* fs), 5.0 / (0.5 \* fs)], btype='band')

return signal.filtfilt(b, a, data)

def detect\_peaks(signal\_data):

return signal.find\_peaks(signal\_data, distance=50)[0]

def extract\_heart\_rate(peaks, fs=100):

if len(peaks) < 2:

return 0

rr\_intervals = np.diff(peaks) / fs

return 60 / np.mean(rr\_intervals)

# Generate synthetic PPG signal

fs = 100

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

sine\_signal = np.sin(2 \* np.pi \* 1.2 \* t)

noise\_signal = 0.1 \* np.random.normal(0, 1, len(t))

ppg\_signal = sine\_signal + noise\_signal

# Process PPG signal

filtered\_signal = bandpass\_filter(ppg\_signal, fs)

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

peaks = detect\_peaks(normalized\_signal)

heart\_rate = extract\_heart\_rate(peaks, fs)

# Print results

print("Filtered Signal (first 10 values):", filtered\_signal[:10])

print("Detected Peaks (first 10 indices):", peaks[:10])

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

# Plot results

plt.figure(figsize=(12, 9))

plt.subplot(3, 2, 1)

plt.plot(t, sine\_signal, label='Raw Sine Signal')

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.legend()

plt.subplot(3, 2, 2)

plt.plot(t, noise\_signal, label='Raw Noise Signal')

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.legend()

plt.subplot(3, 2, 3)

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

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.legend()

plt.subplot(3, 2, 4)

plt.plot(t, filtered\_signal, label='Filtered PPG Signal')

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.legend()

plt.subplot(3, 2, 5)

plt.plot(t, normalized\_signal, label='Normalized PPG Signal')

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.legend()

plt.subplot(3, 2, 6)

plt.plot(t, normalized\_signal, label=f'PPG with Detected Peaks')

plt.plot(t[peaks], normalized\_signal[peaks], 'ro', label='Detected Peaks')

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.legend()

plt.tight\_layout()

plt.show()

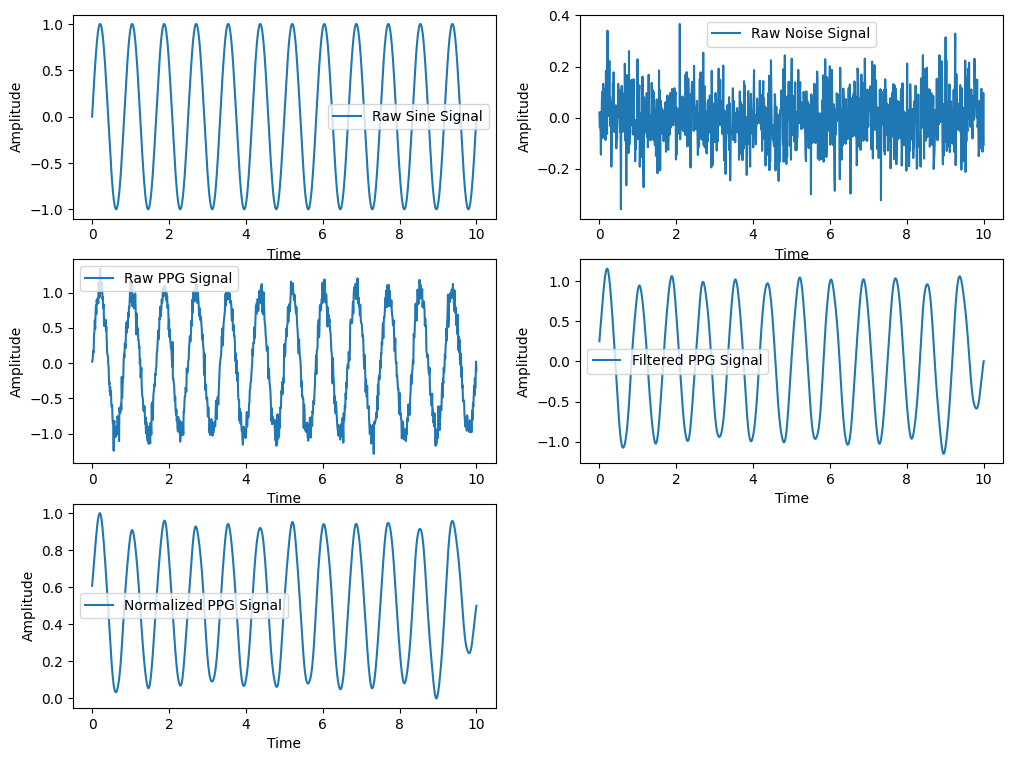
**Input and Output:**

Filtered Signal (first 10 values): [0.25287823 0.31308596 0.37330883 0.43360781 0.49396742 0.55427057

0.61428949 0.67369183 0.73205805 0.78890451]

Detected Peaks (first 10 indices): [ 20 104 188 269 353 437 521 602 686 770]

Estimated Heart Rate: 71.97 BPM



**Discussion:** This program demonstrates heart rate estimation from a PPG signal using filtering and peak detection techniques. The bandpass filter removes noise, and peak detection identifies heartbeats. The estimated heart rate is derived from the peak intervals, showcasing a fundamental technique in wearable health monitoring devices.

**Program No: 6**

**Program Name:** Discrete Fourier Transform (DFT) and Inverse DFT (IDFT) Analysis

**Theory:** The Discrete Fourier Transform (DFT) is used to analyze the frequency content of a discrete signal. It converts a time-domain signal into its frequency-domain representation. The Inverse DFT (IDFT) reconstructs the original signal from its frequency-domain representation. The program computes the DFT of a given sequence, reconstructs it using IDFT, and visualizes the results.

**Code:**

import numpy as np

import matplotlib.pyplot as plt

# Input sequence and N

x = [1,1,1,1]

N= 4

x = np.pad(x, (0, N - len(x)), mode='constant')

# DFT computation

X = np.fft.fft(x, N)

# IDFT computation (Inverse DFT)

x\_reconstructed = np.fft.ifft(X)

# Print the DFT and IDFT values

print("DFT values:", X)

print("Reconstructed IDFT values:", x\_reconstructed.real)

# Plot the input signal

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

plt.subplot(3, 1, 1)

plt.stem(range(len(x)), x)

plt.title('Input Signal x(n)')

plt.xlabel('n')

plt.ylabel('x(n)')

plt.grid()

# Plot the magnitude of DFT

plt.subplot(3, 1, 2)

plt.stem(range(N), np.abs(X))

plt.title('DFT Magnitude |X(k)|')

plt.xlabel('k')

plt.ylabel('|X(k)|')

plt.grid()

# Plot the IDFT signal

plt.subplot(3, 1, 3)

plt.stem(range(N), x\_reconstructed.real)

plt.title('Reconstructed Signal x(n) from IDFT')

plt.xlabel('n')

plt.ylabel('x(n)')

plt.grid()

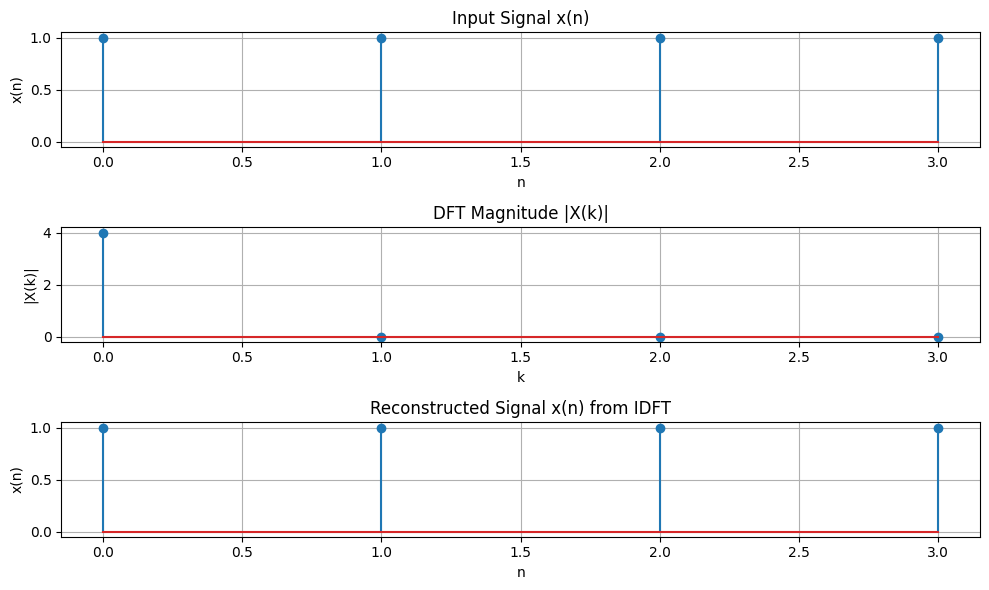
plt.tight\_layout()

plt.show()

**Input and Output:**

DFT values: [4.+0.j 0.+0.j 0.+0.j 0.+0.j]

Reconstructed IDFT values: [1. 1. 1. 1.]



**Discussion:** This program demonstrates the fundamental concept of DFT and IDFT, showing how a signal can be transformed to the frequency domain and back. The results validate that the reconstructed signal closely matches the original, verifying the accuracy of the transformation process.

**Program No: 7**

**Program Name:** Discrete Fourier Transform (DFT) and Inverse DFT (IDFT) Analysis with Frequency Bins

**Theory:** The Discrete Fourier Transform (DFT) is used to analyze the frequency content of a discrete signal. It converts a time-domain signal into its frequency-domain representation. The Inverse DFT (IDFT) reconstructs the original signal from its frequency-domain representation. The program computes the DFT of a given sequence, reconstructs it using IDFT, calculates the frequency bins, and visualizes the results.

**Code:**

import numpy as np

import matplotlib.pyplot as plt

# Input signal and N

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

N = len(x) # Length of the input signal

# DFT computation

X = np.fft.fft(x)

# IDFT computation (Inverse DFT)

x\_reconstructed = np.fft.ifft(X)

# Calculate frequency bins (in Hz)

sampling\_rate = 1 # Assuming a unit sampling rate (samples per second)

frequencies = np.fft.fftfreq(N, d=1/sampling\_rate) # Frequency bins for DFT

# Print the DFT and IDFT values

print("DFT values:", X)

print("Reconstructed IDFT values:", x\_reconstructed.real)

# Plot the input signal

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

# Plot the input signal

plt.subplot(3, 1, 1)

plt.stem(range(N), x)

plt.title('Input Signal x(n)')

plt.xlabel('n')

plt.ylabel('x(n)')

plt.grid()

# Plot the magnitude of DFT

plt.subplot(3, 1, 2)

plt.stem(frequencies, np.abs(X)) # Using frequencies for x-axis

plt.title('DFT Magnitude |X(k)|')

plt.xlabel('Frequency (Hz)')

plt.ylabel('|X(k)|')

plt.grid()

# Plot the IDFT signal

plt.subplot(3, 1, 3)

plt.stem(range(N), x\_reconstructed.real)

plt.title('Reconstructed Signal x(n) from IDFT')

plt.xlabel('n')

plt.ylabel('x(n)')

plt.grid()

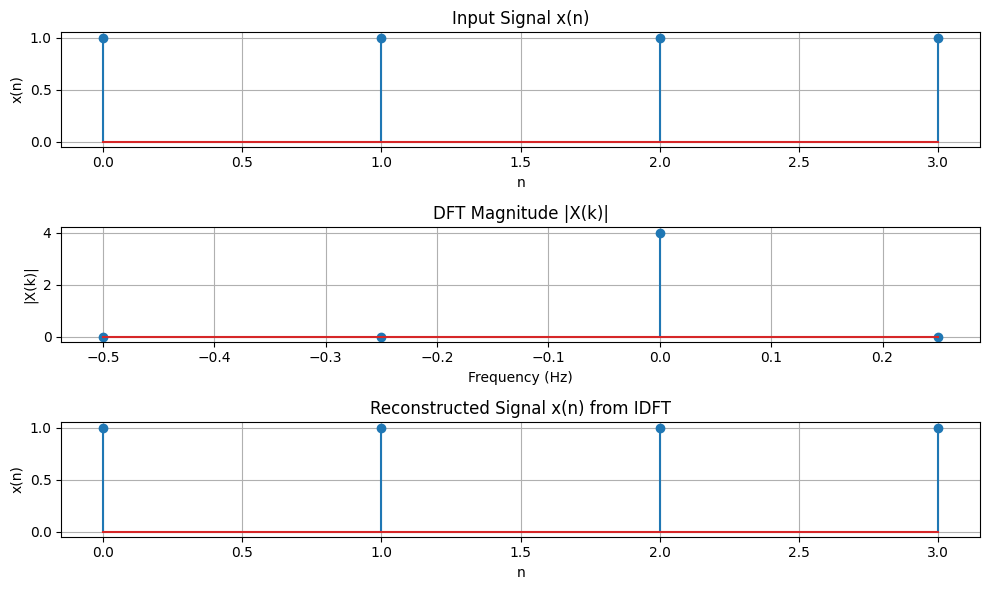
plt.tight\_layout()

plt.show()

**Input and Output:**

DFT values: [4.+0.j 0.+0.j 0.+0.j 0.+0.j]

Reconstructed IDFT values: [1. 1. 1. 1.]



**Discussion:** This program demonstrates the fundamental concept of DFT and IDFT, showing how a signal can be transformed to the frequency domain and back. The inclusion of frequency bins allows for better frequency-domain interpretation. The results validate that the reconstructed signal closely matches the original, verifying the accuracy of the transformation process.