**Pabna University of Science and Technology**

****

**Faculty of Engineering and Technology**

**Department of Information and Communication Engineering**

**Lab Report**

Course Code: **ICE-2204**

|  |  |
| --- | --- |
| **Submitted By:**  **Name:** Md Mahafug Ahmed  **Roll:** 220628  **Reg:** 1065475  **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. |

Course title: **Signals And Systems Sessional**

**Submission Date : 03-03-2025**

|  |  |
| --- | --- |
| **Sl No.** | **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 | PPG - Raw PPG, Noise Addition, Filtering, Normalization, Feature Extraction, 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 Index

**Problem 1: Signal Operations**

**Lab Title:** Basic Signal Operations (Addition, Multiplication, Shifting, Folding, and Scaling).

**Theory:**

Signals are mathematical functions representing physical quantities. In digital signal processing, basic operations on signals include:

* **Addition:**Combining two signals by summing their corresponding values.
* **Multiplication:** Point-wise multiplication of two signals.
* **Shifting:** Moving a signal left (advance) or right (delay).
* **Folding (Time Reversal):** Flipping a signal about the vertical axis.
* **Scaling:** Multiplying a signal by a constant to amplify or attenuate it.

Mathematically, if x[n]and y[n] are discrete-time signals:

1. **Addition:** z[n]=x[n]+y[n]z[n]
2. **Multiplication:** z[n]=x[n]⋅y[n]
3. **Shifting:** x[n−k] (right shift for k>0, left shift for k<0)
4. **Folding:** x[−n]
5. **Scaling:** y[n]=a⋅x[n], where a is a scalar

**Source Code (Python) :**

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")

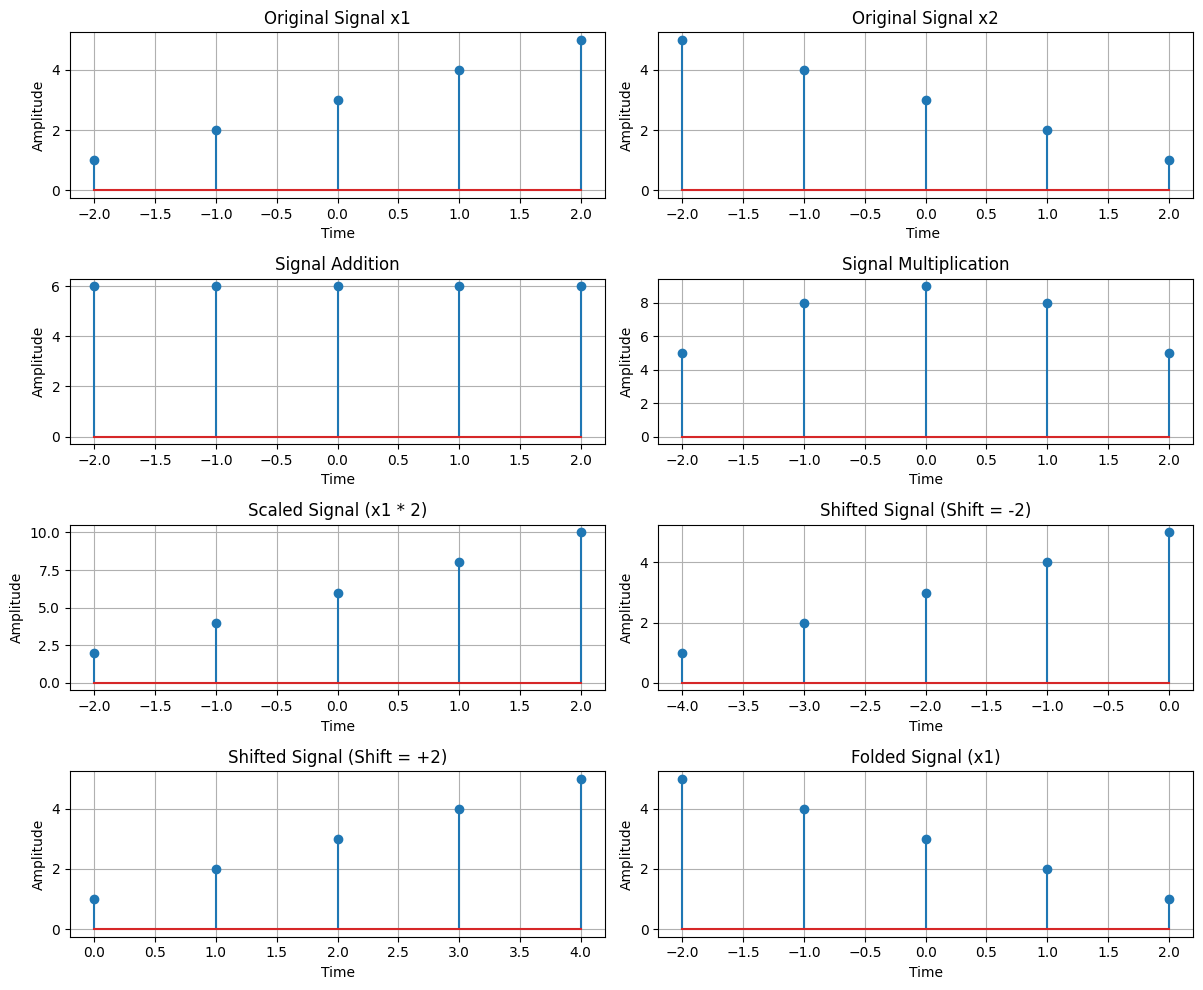
plt.title("Folded Signal (x1)")

plt.grid()

plt.tight\_layout()

plt.show()

**Output :**



**Purpose :**

The purpose of this lab is to understand basic signal operations and visualize their effects. These operations form the foundation of signal processing, used in applications like audio processing, image enhancement, and communications.

**Problem 2: Convolution**

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

**Theory :**

Convolution is a fundamental operation in digital signal processing used to determine the output of a linear time-invariant (LTI) system given an input signal and an impulse response. The convolution sum for discrete-time signals is defined as:

y[n] = Σ(k=-∞ to ∞) x[k]h[n-k]

where:

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

Convolution is widely used in filtering, system analysis, and signal transformation. The convolution sum can be computed manually using graphical methods or programmatically using algorithms.

**Source Code (Python) :**

import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import convolve

def compute\_convolution(signal1, signal2):

    conv\_result = convolve(signal1, signal2, mode='full', method='auto')

    return conv\_result

fs = 1000 # Sampling frequency in Hz

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

freq = 5 # Frequency of the sine wave

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

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

signal1 = sin\_signal

signal2 = np.roll(signal1, 100)

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

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

noisy\_signal = signal1 + noise

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

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()

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()

plt.subplot(3, 1, 3)

plt.plot(conv\_noisy)

plt.title("Convolution with Noisy Signal")

plt.xlabel("Samples")

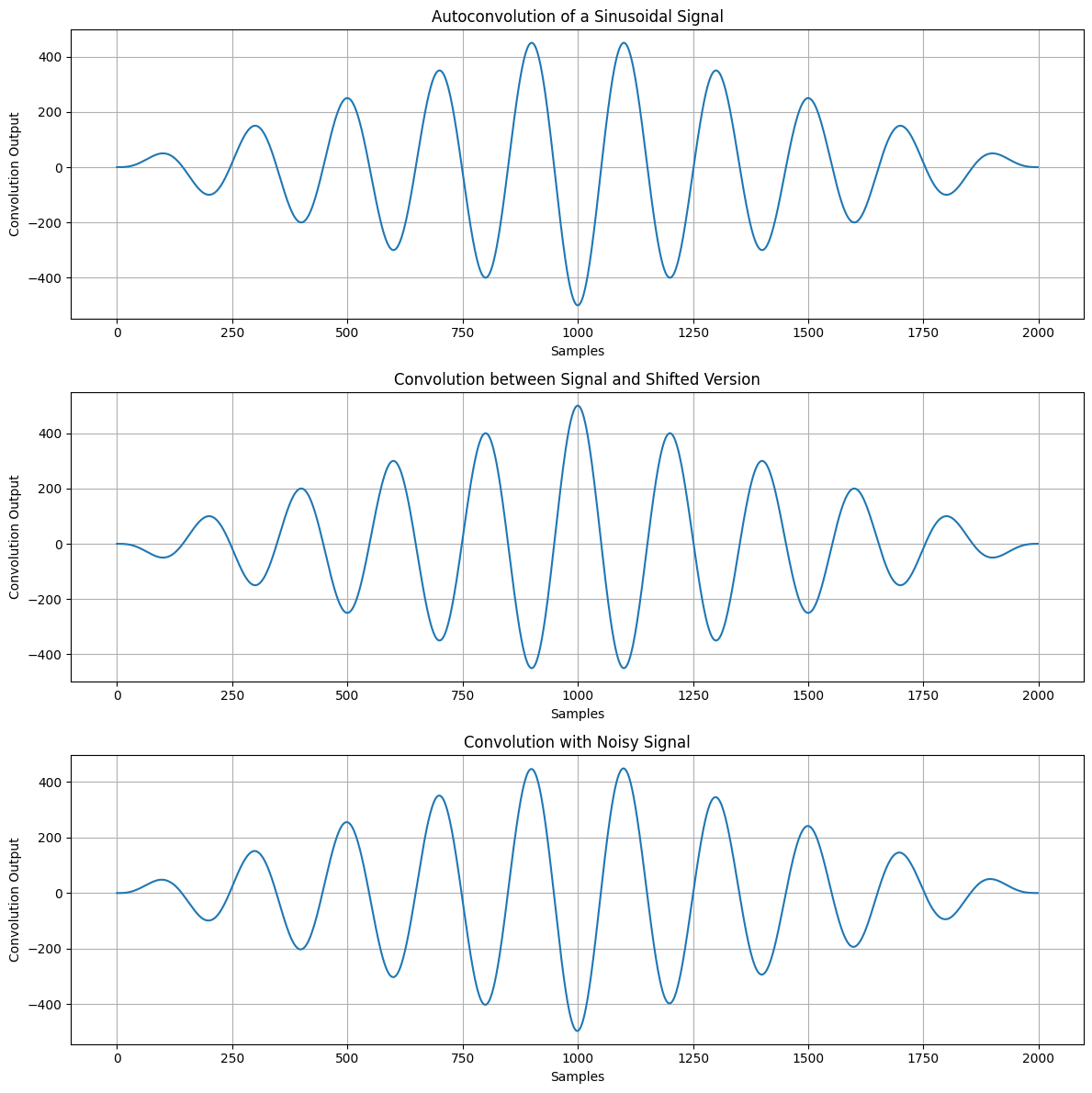
plt.ylabel("Convolution Output")

plt.grid()

plt.tight\_layout()

plt.show()

**Output :**

****

**Purpose :**

The objective of this lab is to understand and implement the convolution operation for discrete-time signals. By computing the convolution, we analyze how an LTI system responds to an input signal and observe how the impulse response affects the output. This experiment is crucial in digital signal processing applications such as filtering and system analysis.

**Problem 3: Correlation**

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

**Theory :**

Correlation is a mathematical operation used to measure the similarity between two discrete-time signals. It is commonly used in signal processing for pattern recognition, feature extraction, and detecting similarities between signals.

There are two main types of correlation:

1. **Auto-correlation**: Correlation of a signal with itself at different time shifts.
2. **Cross-correlation**: Correlation between two different signals.

For two discrete-time signals and , the cross-correlation is defined as:

* Rxy[n] = Σ(k=-∞ to ∞) x[k]y[n+k]

If x[n]=y[n], this becomes the auto-correlation function:

* Rxx[n] = Σ(k=-∞ to ∞) x[k]x[n+k]

The correlation function measures how well two signals match when shifted by samples.

**Source Code (Python) :**

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

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

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

signal1 = sin\_signal

signal2 = np.roll(signal1, 100)

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

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

noisy\_signal = signal1 + noise

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

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")

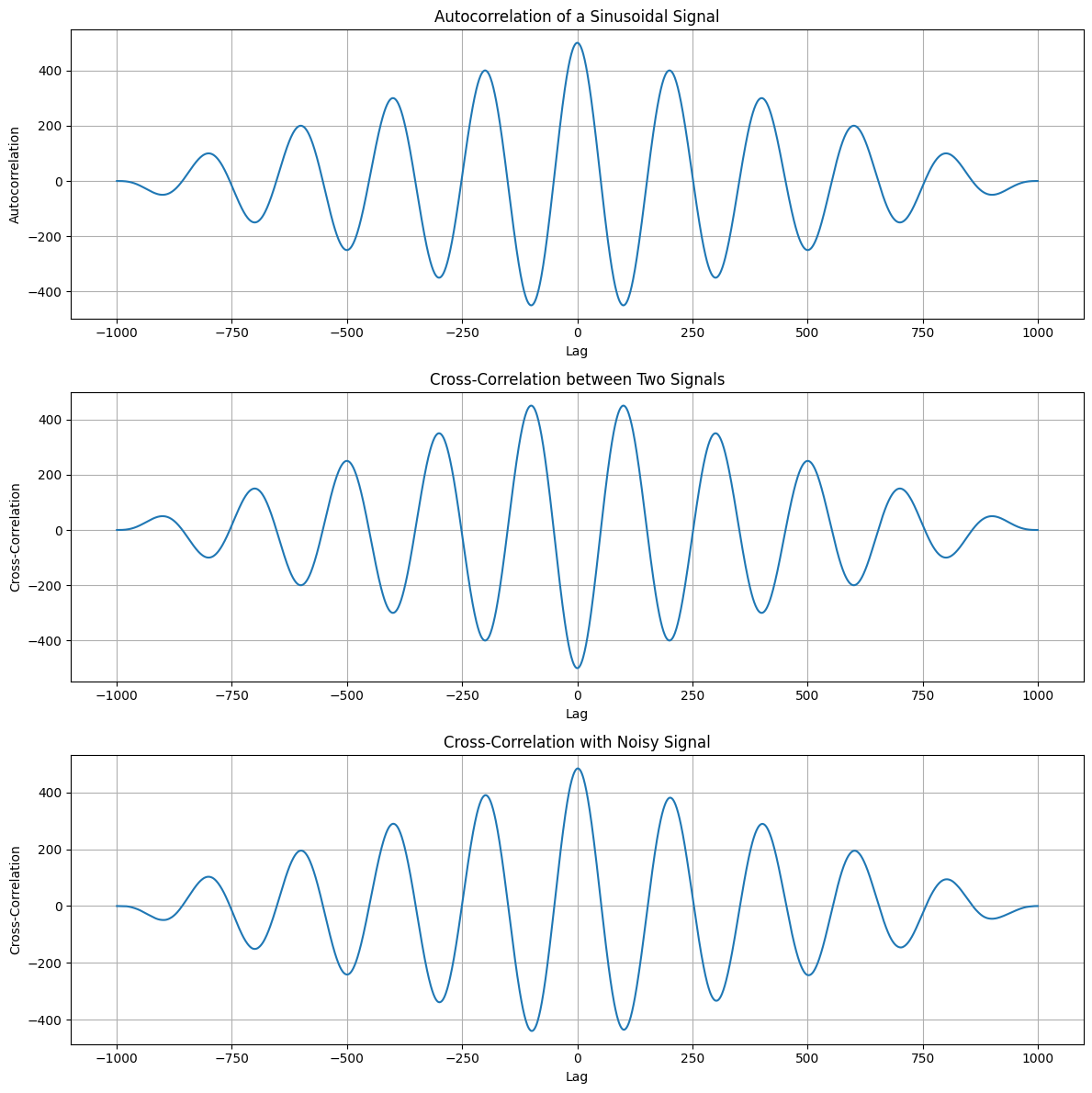
plt.ylabel("Cross-Correlation")

plt.grid()

plt.tight\_layout()

plt.show()

**Output :**

****

### **Purpose:**

The purpose of this experiment is to understand how correlation functions are used to measure similarity between discrete-time signals. By computing auto-correlation and cross-correlation, we analyze signal patterns, which is essential in applications like communication systems, speech processing, and image recognition.

**Problem 4: Signal Sequences**

**Lab Title:** Common Signal Sequences - Impulse, Step, and Ramp.

**Theory :**

Signal sequences are fundamental in signal processing and control systems. Three common types are:

* **Impulse Signal (Delta Function,)**: Defined as 1 at n=0 and 0 elsewhere. It is used to analyze system responses.
* **Step Signal (Unit Step,)**: Defined as 1 for n>=0 and 0 for n<0. It represents sudden changes in systems.
* **Ramp Signal ()**: A linearly increasing function where , r(n) = n for n>=0, modeling uniform acceleration.

These signals are used in various engineering applications such as system identification, control theory, and signal analysis.

**Source Code (Python) :**

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")

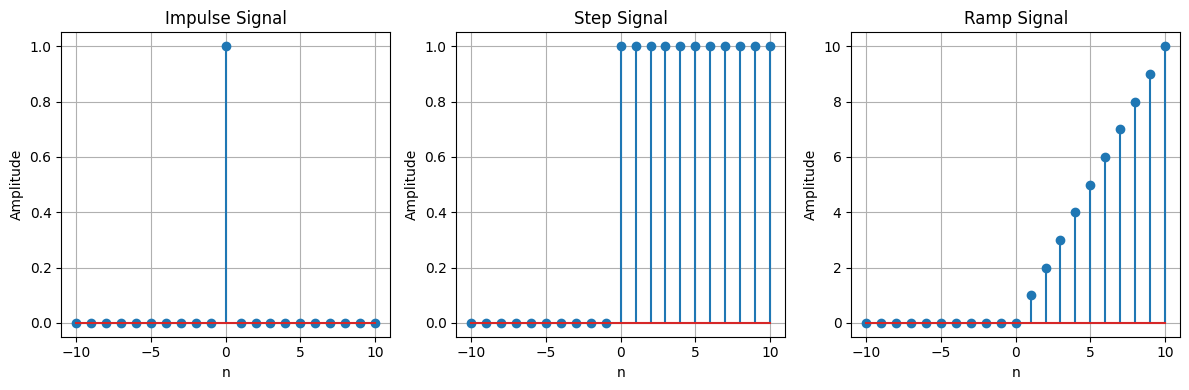
plt.ylabel("Amplitude")

plt.grid()

plt.tight\_layout()

plt.show()

**Output :**

****

**Purpose :**

The purpose of this lab is to understand and visualize fundamental discrete-time signals: impulse, step, and ramp. These signals are crucial in analyzing system behavior in digital signal processing and control systems.

**Problem 5: PPG Signal Processing**

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

**Theory:**

Photoplethysmography (PPG) is a non-invasive optical technique used to measure blood volume changes in tissues. PPG signals are widely used for heart rate monitoring and other physiological measurements. The raw PPG signal contains noise, which must be processed for accurate analysis.

The key steps in PPG signal processing include:

* **Noise Addition:** Introducing synthetic noise to test the robustness of filtering techniques.
* **Filtering:** Removing unwanted components such as motion artifacts and baseline drift using bandpass or moving average filters.
* **Normalization:** Standardizing the signal amplitude for consistent feature extraction.
* **Feature Extraction:** Identifying important parameters such as heart rate, systolic and diastolic points.
* **Peak Detection:** Locating the peaks corresponding to heartbeats for further analysis.

**Source Code (Python) :**

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")

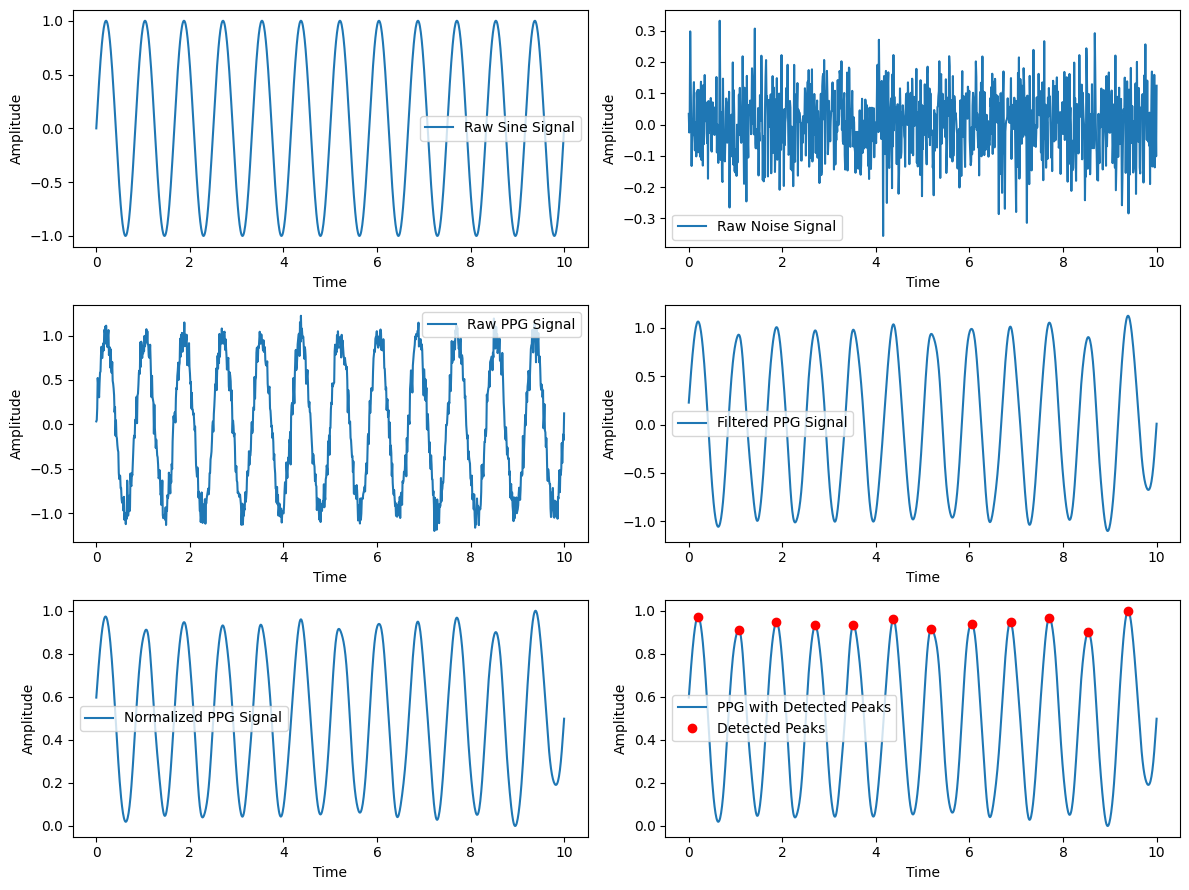
plt.ylabel("Amplitude")

plt.legend()

plt.tight\_layout()

plt.show()

**Output :**

**Purpose:**

This experiment demonstrates the essential steps in PPG signal processing. By implementing noise reduction, normalization, and peak detection, we can extract meaningful physiological information such as heart rate and pulse variability.

**Problem 6: Fourier series**

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

**Theory:**

The Fourier series is a mathematical tool used to represent a periodic function as a sum of sinusoidal components. It decomposes a function into a sum of sines and cosines, known as harmonics, each with a specific amplitude and frequency.

A periodic function f(x) with period T can be expressed as:

Where,

* is the DC component (average value of the function over one period).

Harmonic analysis is used to analyze the contribution of each harmonic in signal representation, often applied in signal processing, vibration analysis, and electrical engineering.

**Source Code (Python) :**

import numpy as np

import matplotlib.pyplot as plt

def fourier\_series(x, terms):

    if terms < 1:

        raise ValueError("Number of terms must be at least 1")

    result = x - x  # Initialize result to 0

    for n in range(1, terms + 1, 2):

        result += (4 / (np.pi \* n)) \* np.sin(n \* x)

    return result  # Return the result after the loop completes

# Define the original square wave function

def square\_wave(x):

    return np.where(np.sin(x) >= 0, 1, -1)

# Generate x values

t = np.linspace(-np.pi, np.pi, 400)

# Plot different approximations

plt.figure(figsize=(8, 6))

# Plot the original square wave

plt.plot(t, square\_wave(t), label='Original Square Wave', linestyle='--', color='black')

# Loop through different number of terms and plot the approximations

for terms in [1, 3, 5, 9]:

    plt.plot(t, fourier\_series(t, terms), label=f'{terms} terms') # Indented this line to be part of the for loop

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

plt.title('Fourier Series Approximation of a Square Wave')

plt.xlabel('Time')

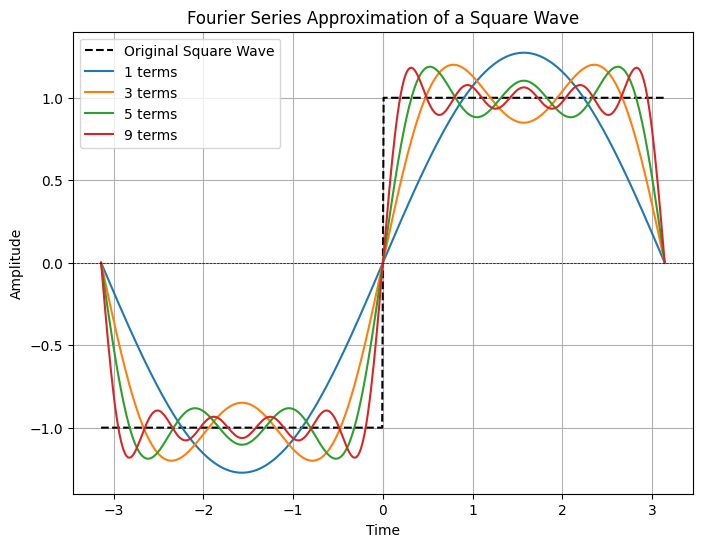
plt.ylabel('Amplitude')

plt.legend()

plt.grid()

plt.show()

**Output:**

****

## **Purpose :**

* To understand the concept of Fourier series decomposition and how it represents periodic functions.
* To compute Fourier coefficients for given functions and reconstruct them using harmonics.
* To visualize the impact of truncating Fourier series on function approximation.
* To apply harmonic analysis in signal processing applications.

**Problem 7: Fourier Transform**

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

**Theory :**

The Fourier Transform is a mathematical tool used to analyze the frequency components of signals. In the case of discrete-time signals, the **Discrete-Time Fourier Transform (DTFT)** and **Discrete Fourier Transform (DFT)** are commonly used.

1. **DTFT:** The DTFT of a discrete-time signal x[n] is given by:

X(e^jω) = Σ(n=-∞ to ∞) x[n] \* e^(-jωn) It is a continuous function of frequency ω .

1. **DFT:** The DFT is a sampled version of the DTFT and is used for numerical computation:

X[k] = ∑(n=0 to N-1) x[n]\* e^-j(2π/N)\*kn

Where N is the number of points in the transform.

The **Fast Fourier Transform (FFT)** is an efficient algorithm used to compute the DFT.

**Source Code (Python) :**

import numpy as np

import matplotlib.pyplot as plt

# Create a time-domain signal (e.g., a sine wave)

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)

# Apply Fourier Transform

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

# Frequency axis

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

# Plot the signal and its Fourier Transform

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)")

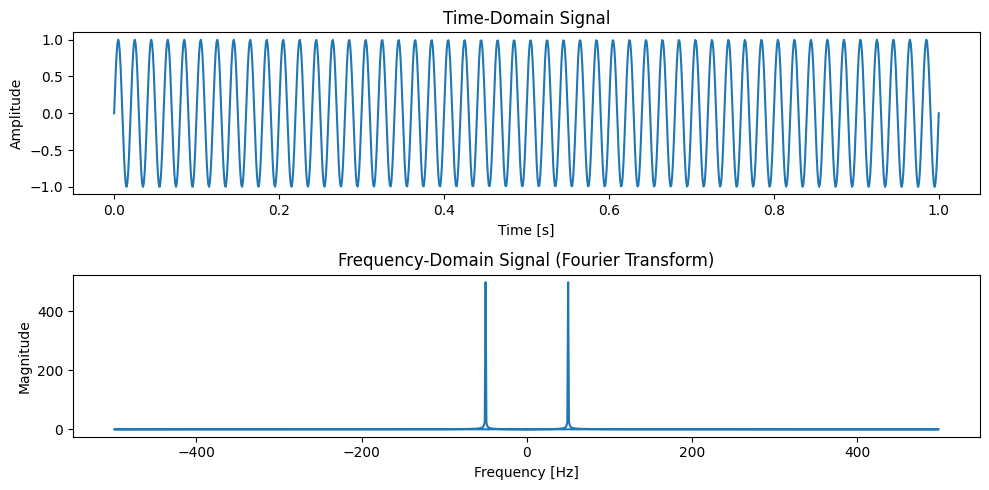
plt.xlabel("Frequency [Hz]")

plt.ylabel("Magnitude")

plt.tight\_layout()

plt.show()

**Output :**

****

**Purpose:**

The purpose of this lab is to understand the Fourier Transform of discrete-time signals and how it reveals frequency components. By implementing the DFT using FFT, we analyze how different signals behave in the frequency domain.

This experiment helps in applications such as signal processing, audio analysis, and digital communication systems.

**Problem 8 : Discrete Fourier Transform (DFT)**

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

**Theory :**

### **Introduction to Discrete Fourier Transform (DFT)**

The Discrete Fourier Transform (DFT) is a fundamental mathematical technique used to analyze frequency components of a discrete signal. It transforms a sequence of complex or real numbers from the time domain to the frequency domain.

### **Mathematical Definition**

The DFT of a sequence x[n] of length N is given by:

X[k] = Σ(n=0 to N-1) x[n] \* e^(-j2πkn/N) 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.

### **Inverse Discrete Fourier Transform (IDFT)**

To recover the original sequence, the Inverse Discrete Fourier Transform (IDFT) is used:

X[n] = 1/N ∑(K=0 to N-1)X[k]\* e^(2πkn/N)

### **Applications of DFT**

* Signal processing (speech and audio analysis)
* Image processing (frequency domain filtering)
* Data compression (JPEG and MP3 encoding)
* Communications (modulation and demodulation)
* Spectrum analysis in scientific research

### **Related Transforms**

* **Fast Fourier Transform (FFT):** An optimized version of DFT that significantly reduces computation time.
* **Short-Time Fourier Transform (STFT):** Used for time-frequency analysis of non-stationary signals.
* **Continuous Fourier Transform (CFT):** The integral-based transform used for continuous signals.

**Source Code (Python) :**

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')

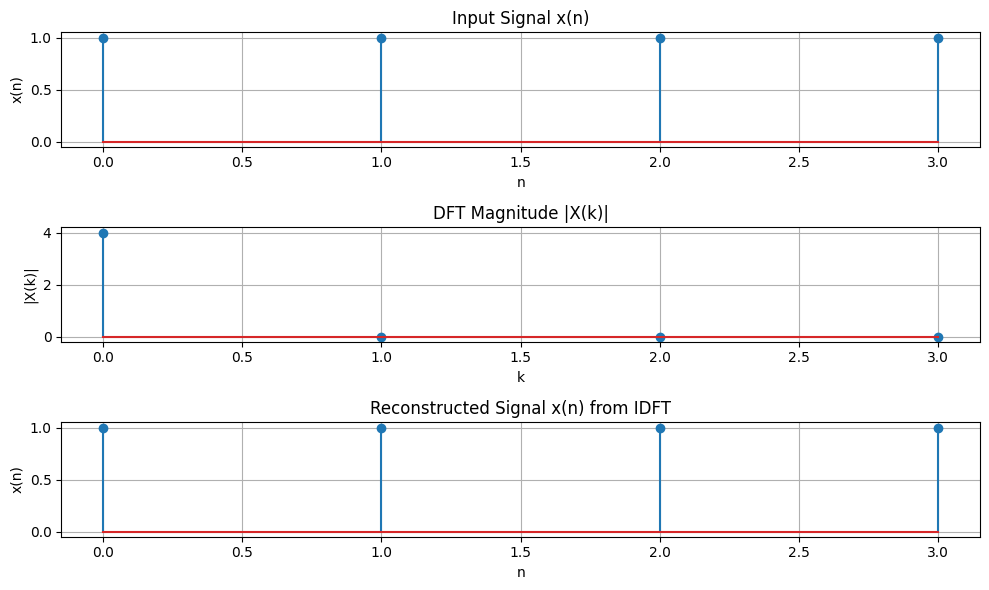
plt.ylabel('x(n)')

plt.grid()

plt.tight\_layout()

plt.show ()

**Output :**

****

## **Purpose:**

The purpose of this lab is to:

1. Understand the fundamental principles of the Discrete Fourier Transform (DFT).
2. Implement DFT and IDFT manually using Python without relying on built-in FFT libraries.
3. Analyze how DFT transforms signals from the time domain to the frequency domain.
4. Explore real-world applications where DFT is useful.
5. Compare DFT with related transforms such as FFT and STFT