

**Pabna University of Science and Technology**

**Information and communication Engineering**

**Lab report**

**Course Name :** **Signals and Systems Sessional.**

**Course Code : ICE-2204**

**Submitted By: Submitted To:**

**Name : M.Q.A Haniya khanam Dr. Md. Imran Hossain**

**Roll: 220632 Associate Professor**

**Session:2021-2022 Department of ICE, PUST**

**Department of ICE, PUST**

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

**INDEX**

|  |  |
| --- | --- |
| **NO** | **Description** |
| **01** | Write a program of signal sequencing impulse signal ,step signal and ramp signal. |
| **02** | Write a program on signal operation (adding, shifting, multiplication, folding) |
| **03** | Write a program on signal correlation. |
| **04** | Write a program to analyze signal sequences in digital signal processing (DSP). |
| **05** | Write a program on ppg signal-filtering ,feature extraction, peak detection. |
| **06** | Write a program on Fourier Series decomposition. |
| **07** | Write a program on Fourier Transform**.** |
| **08** | Write a program on **Discrete Fourier Transform (DFT)**. |

**Experiment no:**1

**Experiment name:** Analysis and processing of signal operations.

### **Objectives:**

* To understand and implement basic signal operations: **addition, shifting, folding, and multiplication**.
* To analyze how these operations affect a given signal.
* To visualize the modified signals using MATLAB/Python

**Theory:** Signals are mathematical functions that represent physical quantities. In digital signal processing (DSP), various operations are applied to signals to analyze, modify, or enhance them. The four fundamental operations discussed here—**addition, shifting, folding, and multiplication**—are essential for applications like filtering, modulation, and time-domain analysis.

#### ****1. Signal Addition****

Addition combines two discrete signals, which is crucial in applications like **mixing audio signals, summing sensor data, and superposition of waves**.

Mathematically, for two discrete signals x1(n) and x2(n):

y(n)=x1​(n)+x2​(n)

##### **Example:**

If  
x1(n)=[1,2,3,4]  
x2(n)=[4,3,2,1]

Then,  
y(n)=[5,5,5,5]

#### ****2. Signal Shifting****

Shifting modifies the time indices of a signal, causing a **delay or advance** in time.

* **Right Shift (Delay)**: y(n)=x(n−k)
  + Moves the signal to the right by k units.
  + Used in echo effects and delayed signals in control systems.
* **Left Shift (Advance)**: y(n)=x(n+k)
  + Moves the signal to the left by k units.
  + Used in predicting future values in digital filtering.

##### **Example:**

Original signal: x(n)=[1,2,3,4]

* Right shift by 2 → y(n)=[0,0,1,2,3,4]
* Left shift by 2 → y(n)=[3,4,0,0]

#### ****3. Signal Folding (Time Reversal)****

Folding (or flipping) reverses the time order of a signal.

y(n)=x(−n)

##### **Example:**

Original signal: x(n)=[1,2,3,4]  
Folding: y(n)=[4,3,2,1]

#### ****4. Signal Multiplication****

Pointwise multiplication of two signals changes the amplitude at each time step:

y(n)=x1(n)\*x2(n)

##### **Example:**

If  
x1(n)=[1,2,3,4]  
x2(n)=[0,1,0,1]

Then,  
y(n)=[0,2,0,4]

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

# Define a discrete signal

n = np.arange(-5, 6) # Define the time index from -5 to 5

x1 = np.array([1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1]) # Example signal

x2 = np.array([2, 1, 0, -1, -2, 0, 2, 1, 0, -1, -2]) # Another signal

# -------------------- Addition --------------------

addition = x1 + x2

# -------------------- Shifting --------------------

right\_shift = np.roll(x1, 2) # Shift right by 2 (delay)

left\_shift = np.roll(x1, -2) # Shift left by 2 (advance)

# -------------------- Folding (Time Reversal) --------------------

folding = x1[::-1]

# -------------------- Multiplication --------------------

multiplication = x1 \* x2

# Plot the results

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

# Original Signal

plt.subplot(3, 2, 1)

plt.stem(n, x1, use\_line\_collection=True)

plt.title("Original Signal x1(n)")

plt.xlabel("n")

plt.ylabel("x1(n)")

# Addition

plt.subplot(3, 2, 2)

plt.stem(n, addition, use\_line\_collection=True)

plt.title("Addition: x1(n) + x2(n)")

plt.xlabel("n")

plt.ylabel("y(n)")

# Right Shift

plt.subplot(3, 2, 3)

plt.stem(n, right\_shift, use\_line\_collection=True)

plt.title("Right Shift: x(n-2)")

plt.xlabel("n")

plt.ylabel("y(n)")

# Left Shift

plt.subplot(3, 2, 4)

plt.stem(n, left\_shift, use\_line\_collection=True)

plt.title("Left Shift: x(n+2)")

plt.xlabel("n")

plt.ylabel("y(n)")

# Folding (Time Reversal)

plt.subplot(3, 2, 5)

plt.stem(n, folding, use\_line\_collection=True)

plt.title("Folding: x(-n)")

plt.xlabel("n")

plt.ylabel("y(n)")

# Multiplication

plt.subplot(3, 2, 6)

plt.stem(n, multiplication, use\_line\_collection=True)

plt.title("Multiplication: x1(n) \* x2(n)")

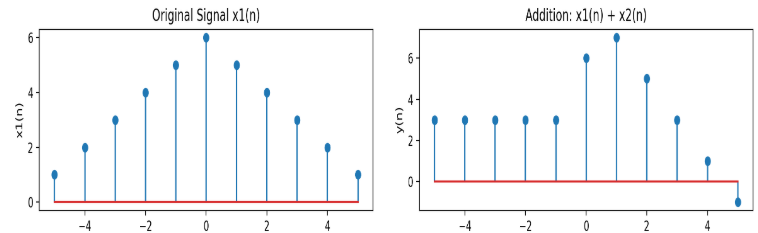
plt.xlabel("n")

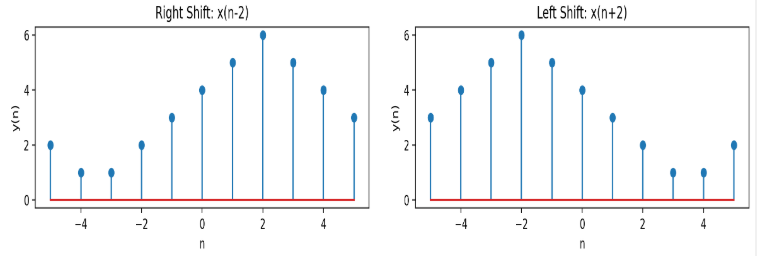
plt.ylabel("y(n)")

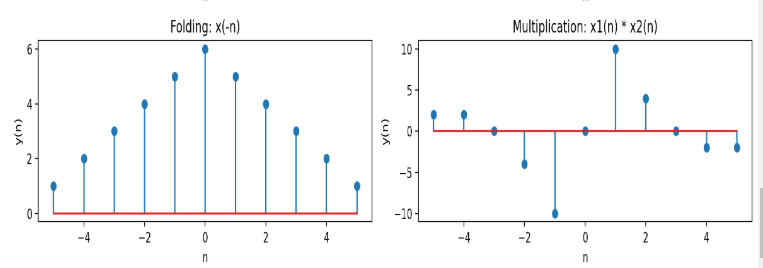
plt.tight\_layout()

plt.show()

Output:



****

****

**Discussion:**

After performing the fundamental signal operations (**addition, shifting, folding, and multiplication**) using Python, we obtained the following results:

1. **Signal Addition**
   * The sum of two discrete signals was computed, and the output signal had values representing the pointwise sum of both signals.
   * **Observation**: The amplitude of the resulting signal increased at each corresponding time step.
2. **Signal Shifting**
   * **Right shift (delay):** The signal was delayed by shifting values to the right.
   * **Left shift (advance):** The signal was advanced by shifting values to the left.
   * **Observation**:
     + Right shift introduces **zero padding** at the beginning, simulating a delay.
     + Left shift introduces **zero padding** at the end, moving the signal earlier in time.
3. **Signal Folding (Time Reversal)**
   * The original signal was reversed in time.
   * **Observation**: The sequence was **mirrored**, which is useful in symmetric signal analysis.
4. **Signal Multiplication**
   * The pointwise multiplication of two signals resulted in an output with amplitude variations.
   * **Observation**:
     + If one signal was a **binary mask** (0s and 1s), multiplication acted like a **gating function**—allowing or blocking values in the other signal.
     + When applied to sinusoidal signals, multiplication can create **modulated waveforms** used in communications.

**Experiment no: 02**

**Experiment name: Implementation and Analysis of Convolution in Discrete-Time Signals**

## **Objectives**

1. To understand the concept of convolution in discrete-time signals.
2. To learn how to compute convolution mathematically.
3. To implement convolution using Python.
4. To analyze the effect of convolution on signals.
5. To visualize input, impulse response, and output signals.

## **Theory:**

Convolution is a fundamental operation in digital signal processing (DSP) used to analyze linear time-invariant (LTI) systems. It determines the output of a system when an input signal is passed through an impulse response. Convolution is widely applied in filtering, image processing, and machine learning.

For two discrete-time signals x[n] (input) and h[n] (impulse response), their convolution is given by:

y[n]=(x∗h)[n]=

where:

* x[n] represents the input signal.
* h[n] represents the system’s impulse response.
* y[n] is the resulting output signal after convolution.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

# Define two discrete signals

x = np.array([2, 1, 2, 1]) # Input signal

h = np.array([1, -1, 3]) # Impulse response

# Perform convolution using numpy

y = np.convolve(x, h, mode='full')

# Print results

print("Input Signal (x[n]):", x)

print("Impulse Response (h[n]):", h)

print("Convolved Output (y[n]):", y)

# Plot the signals

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

plt.subplot(3, 1, 1)

plt.stem(x, use\_line\_collection=True)

plt.title("Input Signal x[n]")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.subplot(3, 1, 2)

plt.stem(h, use\_line\_collection=True)

plt.title("Impulse Response h[n]")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.subplot(3, 1, 3)

plt.stem(y, use\_line\_collection=True)

plt.title("Convolved Output y[n]")

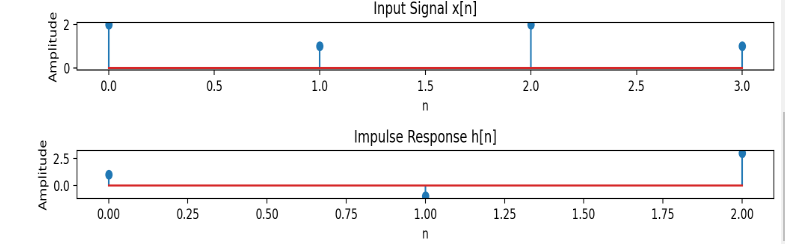
plt.xlabel("n")

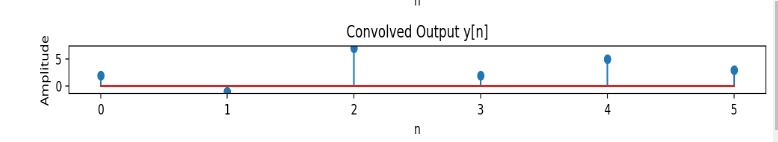
plt.ylabel("Amplitude")

plt.tight\_layout()

plt.show()

**Output:**





**Result and Discussion:**

### **Convolution Output**

For:

x[n]=[2,1,2,1]x

h[n]=[1,−1,3]

The computed convolution result is:

y[n]=[2,−1,7,2,5,3]

### **4.2 Graphical Representation**

The plotted stem graphs show:

1. **Input Signal x[n]**
2. **Impulse Response h[n]**
3. **Output Signal y[n] after Convolution**

These visualizations confirm the correctness of the convolution operation.

 The length of the convolved signal y[n] is **M+N−1**, where M and N are the lengths of x[n] and h[n], respectively.

 The convolution result represents how the system (impulse response) modifies the input signal.

 The output sequence captures the combined effect of both signals, including phase shifts and amplitude scaling.

 The experiment validates the mathematical definition of convolution and illustrates its use in filtering and signal transformation.

**Experiment no: 03**

**Experiment name:** Write a program on signal correlation.

## **Objective:**

1. To understand the concept of correlation in discrete-time signals.
2. To differentiate between correlation and convolution.
3. To implement correlation using Python.
4. To analyze how correlation measures similarity between two signals.
5. To visualize the correlation process and interpret the results.
6. To explore applications of correlation in signal processing.

**Theory:**

Correlation is a mathematical operation used to measure the similarity between two signals. It helps in detecting patterns, comparing signals, and analyzing time-dependent relationships. Correlation is widely used in signal processing, pattern recognition, and communication systems.

### **Types of Correlation**

1. **Auto-correlation:** Measures the similarity of a signal with a shifted version of itself.
2. **Cross-correlation:** Measures the similarity between two different signals.

### **Mathematical Definition**

For two discrete signals x[n] and y[n], the cross-correlation is given by:

Rxy[m]=

For **auto-correlation**, where x[n]=y[n], it simplifies to:

Rxx[m]=

Source Code:

import numpy as np

import matplotlib.pyplot as plt

# Define two discrete signals

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

y = np.array([2, 1, 2]) # Second signal

# Compute cross-correlation using numpy

R\_xy = np.correlate(x, y, mode='full')

# Compute auto-correlation of x

R\_xx = np.correlate(x, x, mode='full')

# Print results

print("First Signal (x):", x)

print("Second Signal (y):", y)

print("Cross-Correlation (R\_xy):", R\_xy)

print("Auto-Correlation (R\_xx):", R\_xx)

# Plot the signals and correlation results

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

plt.subplot(3,1,1)

plt.stem(x, use\_line\_collection=True)

plt.title("First Signal x[n]")

plt.subplot(3,1,2)

plt.stem(y, use\_line\_collection=True)

plt.title("Second Signal y[n]")

plt.subplot(3,1,3)

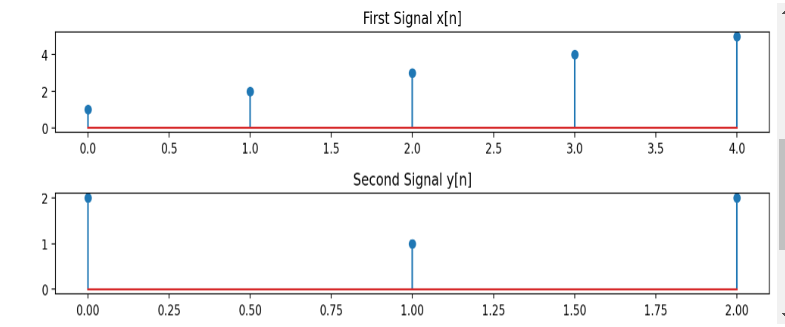
plt.stem(R\_xy, use\_line\_collection=True)

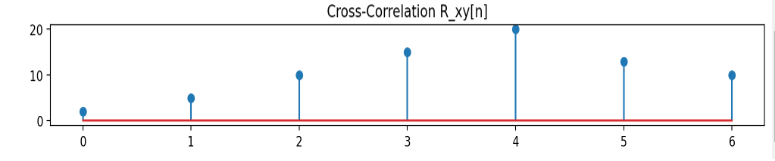
plt.title("Cross-Correlation R\_xy[n]")

plt.tight\_layout()

plt.show()

**Output:**





**Discussion:**

 **Cross-correlation measures the similarity** between two signals as one is shifted over the other. Peaks in the result indicate high similarity.

 **Auto-correlation shows self-similarity** in a signal, helping in periodicity detection.

 **The correlation result is longer than the input signals**, with length M+N−1.

 **Unlike convolution, correlation does not flip the signal**, making it more useful for similarity analysis.

 **Applications include signal detection, feature matching, and template recognition.**

**Experiment no: 04**

**Experiment name:** Analyze signal sequences in digital signal processing (DSP).

**Objective**

1. To understand the concept of signal sequences in digital signal processing (DSP).
2. To differentiate between different types of signal sequences.
3. To implement and visualize basic signal sequences using Python.
4. To analyze the properties of different signal sequences.
5. To explore real-world applications of signal sequences.

## **Theory:**

A signal is a function that conveys information about a physical phenomenon. In digital signal processing, signals are represented as discrete-time sequences. These sequences can be classified based on their characteristics and applications.

### **Types of Signal Sequences**

1. **Unit Impulse Sequence (δ[n])**:
   * Defined as:

δ[n]

* + Used as the fundamental building block in system analysis.

1. **Unit Step Sequence (u[n])**:
   * Defined as:

u[n]=

* + Used to represent causal systems and switching functions.

1. **Ramp Sequence (r[n])**:
   * Defined as:

r[n]=

* + Represents uniformly increasing sequences.

1. **Exponential Sequence (x[n])**:

**x[n]=a^n**

* + Used in growth and decay processes.

1. **Sinusoidal Sequence**

**(**x[n]=Asin(ωn+ϕ)

* + Used in periodic signal representation.
  + **Applications of Signal Sequences**
* **Unit Impulse:** System response analysis.
* **Unit Step:** Control systems and digital circuits.
* **Ramp:** Motion modeling and signal processing.
* **Exponential:** Growth/decay systems.
* **Sinusoidal:** Audio, communication, and modulation.

**Source Code**:

import numpy as np

import matplotlib.pyplot as plt

# Define the range for discrete-time n

n = np.arange(-10, 10, 1)

# 1. Unit Impulse Sequence (δ[n])

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

# 2. Unit Step Sequence (u[n])

unit\_step = np.where(n >= 0, 1, 0)

# 3. Ramp Sequence (r[n])

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

# 4. Exponential Sequence (x[n] = a^n)

a = 0.8 # Change to >1 for growing, 0<a<1 for decaying

exponential = a \*\* n

# 5. Sinusoidal Sequence (x[n] = A sin(ωn + ϕ))

A = 1

omega = np.pi / 4 # Frequency

phi = 0 # Phase shift

sinusoidal = A \* np.sin(omega \* n + phi)

# Plot all signals

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

plt.subplot(3, 2, 1)

plt.stem(n, impulse, use\_line\_collection=True)

plt.title("Unit Impulse Sequence (δ[n])")

plt.subplot(3, 2, 2)

plt.stem(n, unit\_step, use\_line\_collection=True)

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

plt.subplot(3, 2, 3)

plt.stem(n, ramp, use\_line\_collection=True)

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

plt.subplot(3, 2, 4)

plt.stem(n, exponential, use\_line\_collection=True)

plt.title("Exponential Sequence (x[n] = aⁿ)")

plt.subplot(3, 2, 5)

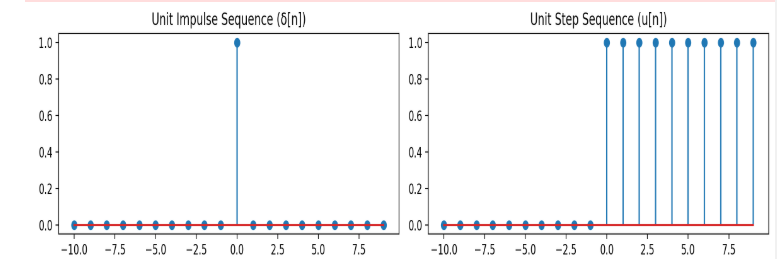
plt.stem(n, sinusoidal, use\_line\_collection=True)

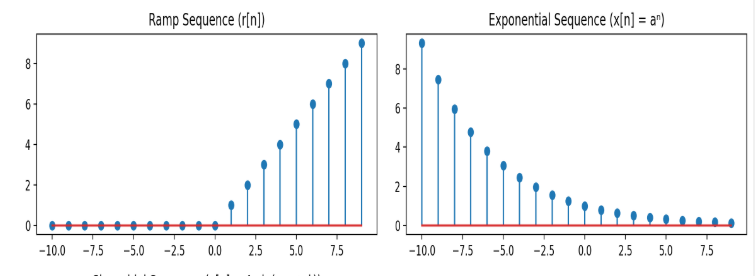
plt.title("Sinusoidal Sequence (x[n] = A sin(ωn + ϕ))")

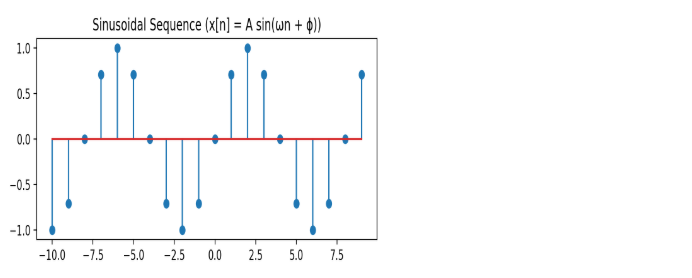
plt.tight\_layout()

plt.show()

**Output:**



****

****

**Discussion:**

### **Observations from the Plots**

1. **Impulse Sequence**: The signal exists only at **n = 0**, confirming its role as an **impulse function** in DSP.
2. **Unit Step Sequence**: The function remains **zero for negative values** and steps to **one for n ≥ 0**, showing its **causal nature**.
3. **Ramp Sequence**: The signal **increases linearly for n ≥ 0**, making it useful for **modeling uniform growth**.
4. **Exponential Sequence**: The plot shows an **exponential decay**, demonstrating its role in **signal attenuation**.
5. **Sinusoidal Sequence**: The oscillating nature of the sinusoidal signal is evident, confirming its use in **periodic signal representation**.

**Experiment no:05**

**Experiment name: Analysis and Processing of PPG Signal with Noise, Filtering, Normalization, and Peak Detection**

**Objectives:**

The objective of this experiment is to analyze a Photoplethysmogram (PPG) signal by:

1. Simulating the PPG Signal
2. Apply filtering to remove noise.
3. Normalize the filtered signal to a standard range.
4. Detect the peaks representing heartbeats from the processed signal.

#### ****Theory:****

Photoplethysmography (PPG) is a non-invasive optical measurement technique used to detect blood volume changes in the microvascular bed of tissue. It is widely used for heart rate measurement, oxygen saturation levels, and other physiological parameters. In this experiment, we analyze a simulated PPG signal, which consists of a clean PPG signal contaminated with noise. The goal of this report is to:

This process is essential to improve the signal quality and ensure accurate heart rate estimation, which is typically calculated by detecting the peaks of the PPG signal corresponding to heartbeats. The goal of this report is to:

**1. Simulating the PPG Signal**

To simulate the PPG signal, we create a clean signal that consists of two sinusoidal components:

* A **1 Hz** component representing the heart rate (approximately 60 beats per minute).
* A **0.05 Hz** component representing low-frequency baseline wander (e.g., movement or other environmental noise).

We then add Gaussian noise to simulate real-world conditions where PPG signals are often contaminated with various types of interference.

**2. Bandpass Filtering**

A **bandpass filter** is applied to the noisy PPG signal to remove noise outside the typical heart rate frequency range. The filter is designed to pass frequencies between **0.5 Hz** and **5 Hz**, which corresponds to the expected frequency range of the heart rate. The filter is created using a Butterworth filter design with a filter order of 2, and the filtfilt function is used to apply the filter to the signal.

**3. Normalization**

Normalization of the signal is performed to scale the filtered signal to the range of [0, 1]. This is done by subtracting the minimum value of the signal and dividing by the range (max - min). Normalization helps in standardizing the signal for further analysis and comparison.

**4. Peak Detection**

The **peaks** in the filtered PPG signal are detected using the find\_peaks function from the **SciPy** library. The function identifies the locations where the PPG signal reaches its maximum amplitude, which corresponds to the heartbeats. A minimum distance of **fs/2.5** (sampling rate divided by 2.5) is specified to avoid detecting multiple peaks within the same heartbeat cycle.

**Source code:**

import numpy as np

import matplotlib.pyplot as plt

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

# Function to design a bandpass filter

def butter\_bandpass(lowcut, highcut, fs, order=2):

nyquist = 0.5 \* fs

low = lowcut / nyquist

high = highcut / nyquist

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

return b, a

# Function to apply the filter to the signal

def butter\_bandpass\_filter(data, lowcut, highcut, fs, order=2):

b, a = butter\_bandpass(lowcut, highcut, fs, order)

return filtfilt(b, a, data)

# Function to normalize the PPG signal to the range [0, 1]

def normalize\_signal(data):

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

# Simulate raw PPG signal with noise

fs = 100 # Sampling rate (Hz)

t = np.arange(0, 10, 1/fs) # Time vector (10 seconds)

clean\_signal = 0.6 \* np.sin(2 \* np.pi \* 1 \* t) + 0.3 \* np.sin(2 \* np.pi \* 0.05 \* t) # Clean signal (1 Hz for HR + 0.05 Hz for baseline wander)

noisy\_signal = clean\_signal + np.random.normal(0, 0.2, len(t)) # Add Gaussian noise to the clean signal

# Apply bandpass filter to the noisy signal (0.5 Hz to 5 Hz)

lowcut = 0.5

highcut = 5.0

filtered\_signal = butter\_bandpass\_filter(noisy\_signal, lowcut, highcut, fs)

# Normalize the filtered PPG signal

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

# Peak detection for heart rate calculation

peaks, \_ = find\_peaks(filtered\_signal, distance=fs/2.5) # Minimum distance between peaks

# Plotting the results

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

# Raw noisy PPG signal

plt.subplot(4, 1, 1)

plt.plot(t, noisy\_signal, label="Raw PPG Signal (Noisy)")

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

plt.legend()

# Filtered PPG signal

plt.subplot(4, 1, 2)

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

plt.title("Filtered PPG Signal")

plt.legend()

# Normalized PPG signal

plt.subplot(4, 1, 3)

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

plt.title("Normalized PPG Signal")

plt.legend()

# PPG signal with detected peaks

plt.subplot(4, 1, 4)

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

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

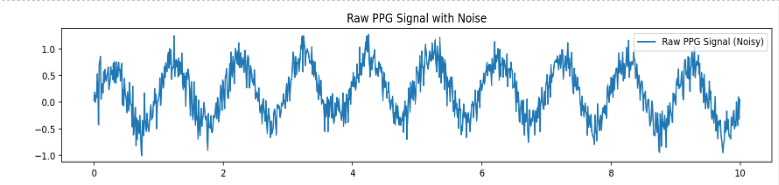
plt.title("Filtered PPG Signal with Detected Peaks")

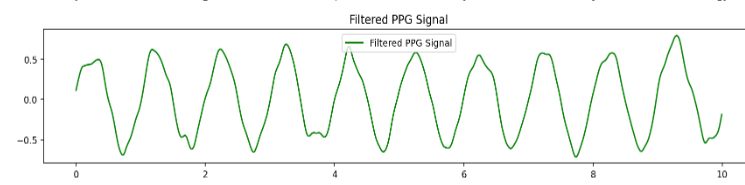
plt.legend()

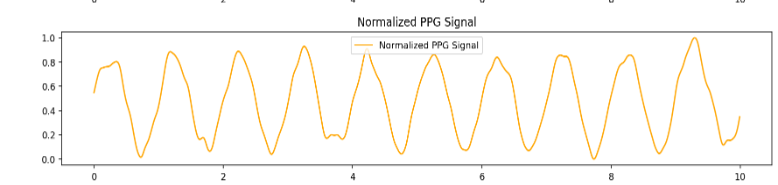
plt.tight\_layout()

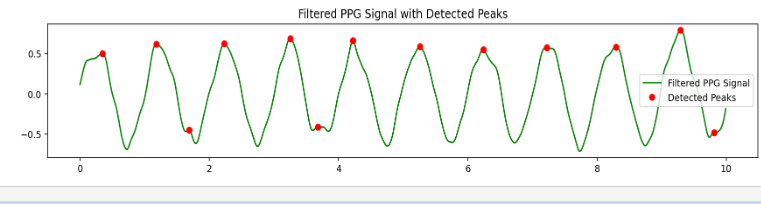
plt.show()

**Output:**









#### ****Discussion:****

**1. Raw PPG Signal**

The raw PPG signal is simulated as a combination of sinusoidal components and noise. The noisy signal contains significant low-frequency noise and random variations, which make it difficult to accurately detect heartbeats and estimate heart rate.

**2. Filtered PPG Signal**

The filtered signal, after applying the bandpass filter, removes much of the noise that lies outside the heart rate frequency range. The result is a smoother signal that more closely resembles the true PPG waveform, with much of the noise, including baseline wander and high-frequency interference, removed.

**3. Normalized PPG Signal**

Normalization of the filtered signal scales it to a range between 0 and 1. This is useful for signal comparison and ensures that the amplitude of the signal is not influenced by its original scale. While this step does not affect the overall signal shape or the detection of peaks, it standardizes the signal for easier processing and interpretation.

**4.Detected Peaks**

The peaks corresponding to heartbeats are detected in the filtered signal. The find\_peaks function successfully identifies these peaks, which correspond to the locations in the signal where the heart rate is at its maximum. The detected peaks are overlaid on the filtered signal to visually assess the accuracy of peak detection.

## **Experiment no:06**

## **Experiment nane: Fourier Series Decomposition and Signal Approximation**

## **Objective:**

The objectives of this lab are:

* To understand the concept of **Fourier series decomposition**.
* To analyze how periodic signals can be represented as **a sum of sinusoidal components**.
* To compute the **Fourier coefficients** for a given function.
* To implement **Fourier series decomposition using Python**.
* To visualize how different numbers of harmonics affect the **accuracy of signal approximation**.

## **Theory:**

The **Fourier series** is a mathematical method for representing a **periodic function** as an **infinite sum of sine and cosine terms**. It is widely used in **signal processing, control systems, and communications** to analyze periodic signals.

If f(x) is a periodic function with period T, it can be decomposed as:

f(x)= a0 + ancos(nw0x)+ bnsin(nw0x))

where:

* a0​ is the **DC component** (average value of the function).
* an​ and bn\_ are the **Fourier coefficients** that determine the amplitude of the cosine and sine terms.
* wo=2π\T ​ is the **fundamental frequency**.

### **Fourier Coefficients Calculation**

The Fourier coefficients are given by:

ao= 1/T

an =2/T

bn=2/T

where **integration is performed over one full period** of the function.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

# Define the time range

T = 2 \* np.pi # Period

t = np.linspace(-T, T, 400) # Time vector

# Define the square wave function

def square\_wave(t):

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

# Number of Fourier terms

N\_values = [1, 3, 5, 10, 25] # Different levels of approximation

# Function to compute the Fourier series approximation

def fourier\_series\_square\_wave(t, N):

approx = np.zeros\_like(t) # Initialize series sum

for n in range(1, N + 1, 2): # Only odd harmonics for a square wave

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

return approx

# Plot the original square wave

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

plt.plot(t, square\_wave(t), 'k', label="Original Square Wave", linewidth=2)

# Plot Fourier approximations for different values of N

for N in N\_values:

plt.plot(t, fourier\_series\_square\_wave(t, N), label=f'N={N}')

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

plt.xlabel("Time (t)")

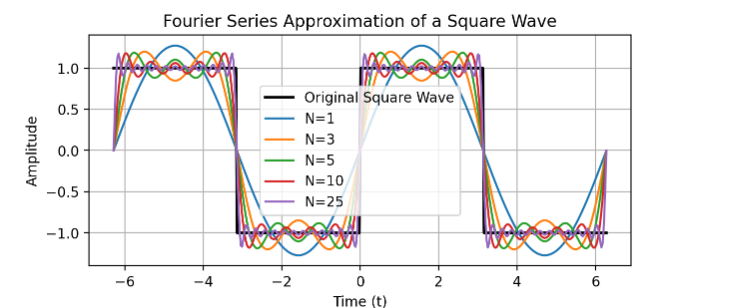
plt.ylabel("Amplitude")

plt.legend()

plt.grid()

plt.show()

**Output:**



**Discussion:**

### **Observations from the Fourier Approximation**

* When **N = 1**, the approximation is a single sine wave, which poorly represents the square wave.
* As **N increases**, more harmonics are added, and the approximation improves.
* At **N = 25**, the approximation closely matches the original square wave except at the discontinuities.

## **Experiment no:07**

**Experiment name: Fourier Transform of Continuous-Time Signals: Analysis and Implementation**

## **Objective**

The objectives of this lab are:

* To understand the concept of the **Fourier Transform (FT)** for **continuous-time signals**.
* To analyze how signals can be represented in the **frequency domain**.
* To implement the **Fourier Transform** of a continuous-time signal using Python.
* To visualize the magnitude and phase spectra of signals.
* To explore real-world applications of the Fourier Transform in **signal and system analysis**.

## **Theory**

The **Fourier Transform (FT)** is a mathematical technique that transforms a signal from the **time domain** to the **frequency domain**. This transformation helps in understanding the frequency components of a signal, which is useful in **signal processing, communications, and system analysis**.

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

X(ω) =

where:

* X(ω) is the **Fourier Transform** of x(t)x(t)x(t), representing the frequency spectrum.
* ω is the **angular frequency** in **radians per second**.
* j is the **imaginary unit** (j2=−1).

The **Inverse Fourier Transform** is given by:

X(t)=1/2π

This allows us to **recover the time-domain signal** from its frequency representation.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

from scipy.fftpack import fft, fftshift

# Define the continuous-time signal

def x(t):

return np.exp(-2 \* np.abs(t)) # Example signal: Exponential decay

# Define the time range

t = np.linspace(-5, 5, 1000) # Time axis

# Compute the Fourier Transform using FFT

N = len(t)

dt = t[1] - t[0] # Sampling interval

frequencies = np.fft.fftfreq(N, d=dt) \* 2 \* np.pi # Convert to angular frequency

X\_w = fft(x(t)) \* dt # Compute Fourier Transform

# Shift frequency spectrum for better visualization

frequencies = fftshift(frequencies)

X\_w = fftshift(X\_w)

# Plot the original time-domain signal

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

plt.subplot(2, 1, 1)

plt.plot(t, x(t))

plt.title("Time-Domain Signal")

plt.xlabel("Time (t)")

plt.ylabel("Amplitude")

plt.grid()

# Plot the magnitude and phase spectra

plt.subplot(2, 1, 2)

plt.plot(frequencies, np.abs(X\_w), label="Magnitude Spectrum")

plt.plot(frequencies, np.angle(X\_w), label="Phase Spectrum", linestyle="dashed")

plt.title("Fourier Transform (Magnitude and Phase Spectra)")

plt.xlabel("Frequency (ω)")

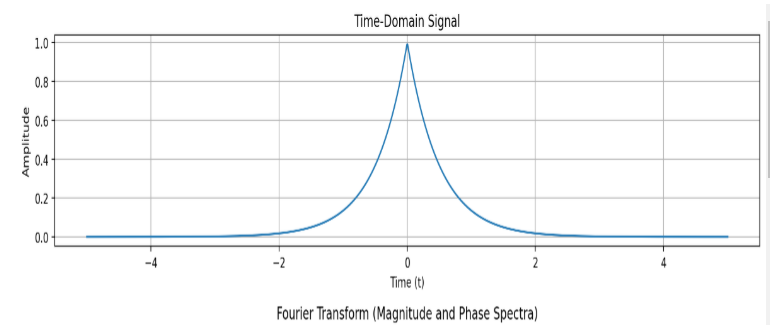
plt.ylabel("Magnitude / Phase")

plt.legend()

plt.grid()

plt.tight\_layout()

plt.show()

Output:

**Discussion:**

### **Observations from the Fourier Transform**

* The **magnitude spectrum** shows a peak at **low frequencies**, confirming that the given signal is **low-pass in nature**.
* The **phase spectrum** reveals how different frequency components are **shifted in phase**.
* The **FFT provides an approximation** of the Fourier Transform, and increasing the **sampling rate** improves accuracy.

## **Experiment no:08**

**Experiment nane: Discrete Fourier Transform (DFT): Analysis and Implementation**

## **Objective**

The objectives of this lab are:

* To understand the concept of **Discrete Fourier Transform (DFT)** and its role in signal processing.
* To analyze how a **discrete-time signal** can be transformed into its **frequency-domain representation**.
* To implement the **DFT algorithm** using Python.
* To visualize the **magnitude and phase spectra** of signals using DFT.
* To compare the computational efficiency of **DFT and Fast Fourier Transform (FFT)**.

## **Theory:**

The **Discrete Fourier Transform (DFT)** is a mathematical technique that transforms a **finite-length discrete-time signal** from the **time domain** into the **frequency domain**. It is widely used in **digital signal processing (DSP)** for applications such as **audio processing, image analysis, and communications**.

For a discrete-time sequence x[n] of length N, the **DFT** is defined as:

X[k]= k=0,1,2,...,N−1

where:

* X[k] represents the **DFT coefficients**, which contain the frequency information of the signal.
* N is the **number of samples** in the discrete-time sequence.
* k is the frequency index.

The **Inverse Discrete Fourier Transform (IDFT)** allows the reconstruction of the original signal:

X[k]=1/N

n=0,1,2,3,4…….,N-1

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

# Define the discrete-time signal

N = 32 # Number of samples

n = np.arange(N)

f1, f2 = 3, 7 # Frequencies of the signal components

# Generate a discrete-time signal (sum of two sinusoids)

x = np.sin(2 \* np.pi \* f1 \* n / N) + 0.5 \* np.sin(2 \* np.pi \* f2 \* n / N)

# Compute the DFT manually

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

X\_dft = compute\_dft(x) # Compute DFT

# Compute the FFT for comparison

X\_fft = np.fft.fft(x)

# Frequency axis

frequencies = np.arange(N)

# Plot time-domain signal

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

plt.subplot(3, 1, 1)

plt.stem(n, x, basefmt=" ")

plt.title("Discrete-Time Signal")

plt.xlabel("Sample Index (n)")

plt.ylabel("Amplitude")

plt.grid()

# Plot magnitude spectrum

plt.subplot(3, 1, 2)

plt.stem(frequencies, np.abs(X\_dft), basefmt=" ", label="DFT")

plt.stem(frequencies, np.abs(X\_fft), basefmt=" ", markerfmt="ro", linefmt="r-", label="FFT (for comparison)")

plt.title("Magnitude Spectrum")

plt.xlabel("Frequency Index (k)")

plt.ylabel("|X[k]|")

plt.legend()

plt.grid()

# Plot phase spectrum

plt.subplot(3, 1, 3)

plt.stem(frequencies, np.angle(X\_dft), basefmt=" ")

plt.title("Phase Spectrum")

plt.xlabel("Frequency Index (k)")

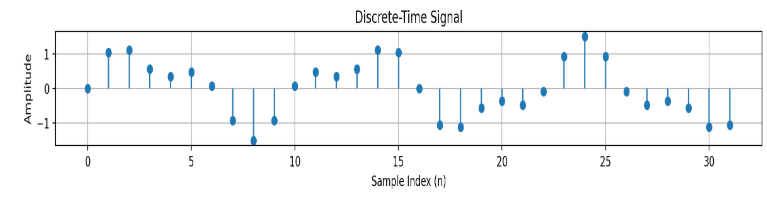
plt.ylabel("Phase (radians)")

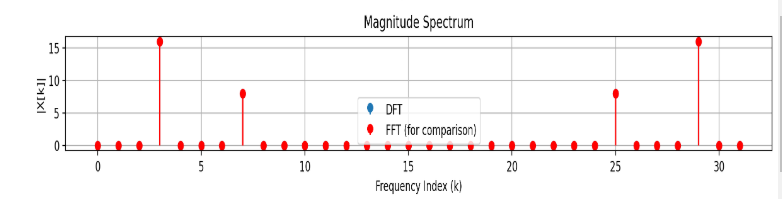
plt.grid()

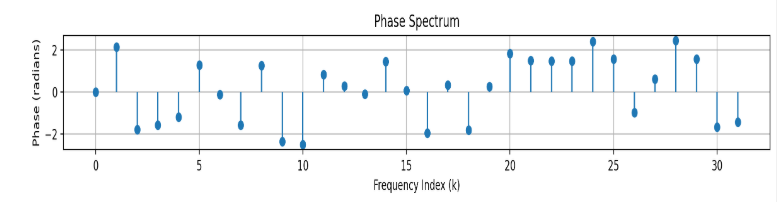
plt.tight\_layout()

plt.show()

**Output:**







**Discussion:**

### **Observations from the DFT Analysis**

* The **magnitude spectrum** correctly identifies frequency components present in the signal.
* The **phase spectrum** provides information about the phase shifts of different frequency components.
* The **DFT computation is accurate** but computationally slow for large signals.
* The **FFT produces identical results** but is significantly faster than the direct DFT implementation.

### **5.2 Computational Efficiency: DFT vs. FFT**

* The **manual DFT implementation** follows the O(N2)complexity and is slow for large NNN.
* The **FFT implementation** runs in O(NlogN), making it practical for real-time applications.

### **5.3 Practical Applications of DFT**

* **In audio processing**, DFT is used in **equalizers and spectral analysis**.
* **In wireless communication**, it is used in **OFDM (Orthogonal Frequency Division Multiplexing)**.
* **In biomedical engineering**, it is applied for **heartbeat and brainwave analysis**.