

|  |  |
| --- | --- |
| SL. | Problem Description |
| 01 | Lab Report on Signal Addition, Shifting, and Folding. |
| 02 | Lab Report on Linear Convolution of Discrete-Time Signals. |
| 03 | Lab Report on Cross-Correlation of Discrete-Time Signals. |
| 04 | Lab Report on Basic Discrete-Time Signals. |
| 05 | Lab Report on PPG Signal Processing and Heart Rate Estimation. |
| 06 | Lab Report on Fourier Series Approximation of a Square Wave. |
| 07 | Lab Report on Fourier Transform of a Signal Using FFT. |
| 08 | Lab Report on Discrete Fourier Transform (DFT) Analysis. |

**Problem No:01**

**Problem Name:** Lab Report on Signal Addition, Shifting, and Folding.

**Title:** Signal Addition, Shifting, and Folding in Discrete-Time Signals.

**Objective:** To perform and analyze basic operations on discrete-time signals, including:

* Addition of two discrete-time signals.
* Time shifting (left and right shift).
* Folding (time-reversal) of a signal.

### **Theory:**

### **1. Discrete-Time Signals:**

A discrete-time signal is a sequence of values that represent a signal sampled at discrete time intervals. It is denoted as *x[n]x[n]*x[n], where *nn*n represents discrete time indices.

### **2. Signal Addition:**

When two discrete-time signals *x1[n]x\_1[n]*x1 [n] and *x2[n]x\_2[n]*x2 [n] are added, the result is given by:

*y[n]=x1[n]+x2[n]y[n] = x\_1[n] + x\_2[n]*y[n]=x1 [n]+x2 [n]

If the two signals have different lengths, zero-padding is applied to match their lengths before addition.

### **3. Signal Shifting:**

Time shifting moves a signal left or right along the time axis:

* **Right Shift:** *x[n−k]x[n-k]*x[n−k] shifts the signal by *kk*k samples to the right. This delays the signal.
* **Left Shift:** *x[n+k]x[n+k]*x[n+k] shifts the signal by *kk*k samples to the left. This advances the signal.

### **4. Signal Folding (Time Reversal):**

Folding (or time reversal) inverts the time indices of a signal. The transformed signal is given by:

*y[n]=x[−n]y[n] = x[-n]*y[n]=x[−n]

This operation reflects the signal around *n=0n=0*n=0, reversing its order.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

def signal\_addition(x1, x2):

""" Adds two signals with zero-padding if needed """

len\_x1, len\_x2 = len(x1), len(x2)

max\_len = max(len\_x1, len\_x2)

# Zero-padding to match lengths

x1 = np.pad(x1, (0, max\_len - len\_x1), mode='constant')

x2 = np.pad(x2, (0, max\_len - len\_x2), mode='constant')

return x1 + x2

def signal\_shift(x, shift):

""" Shifts the signal right (positive shift) or left (negative shift) """

if shift > 0: # Right shift

return np.pad(x, (shift, 0), mode='constant')[:-shift]

elif shift < 0: # Left shift

return np.pad(x, (0, -shift), mode='constant')[-shift:]

return x

def signal\_folding(x):

""" Folds (reverses) a given signal """

return x[::-1]

# Example Signals

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

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

# Signal Addition

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

# Signal Shifting

right\_shifted = signal\_shift(x1, 2)

left\_shifted = signal\_shift(x1, -2)

# Signal Folding

folded\_signal = signal\_folding(x1)

# Plotting the results

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

# Original Signals

plt.subplot(2, 2, 1)

plt.stem(x1)

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

# Signal Addition

plt.subplot(2, 2, 2)

plt.stem(added\_signal)

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

# Right Shifted Signal

plt.subplot(2, 2, 3)

plt.stem(right\_shifted)

plt.title("Right Shifted Signal x1[n-2]")

# Folded Signal

plt.subplot(2, 2, 4)

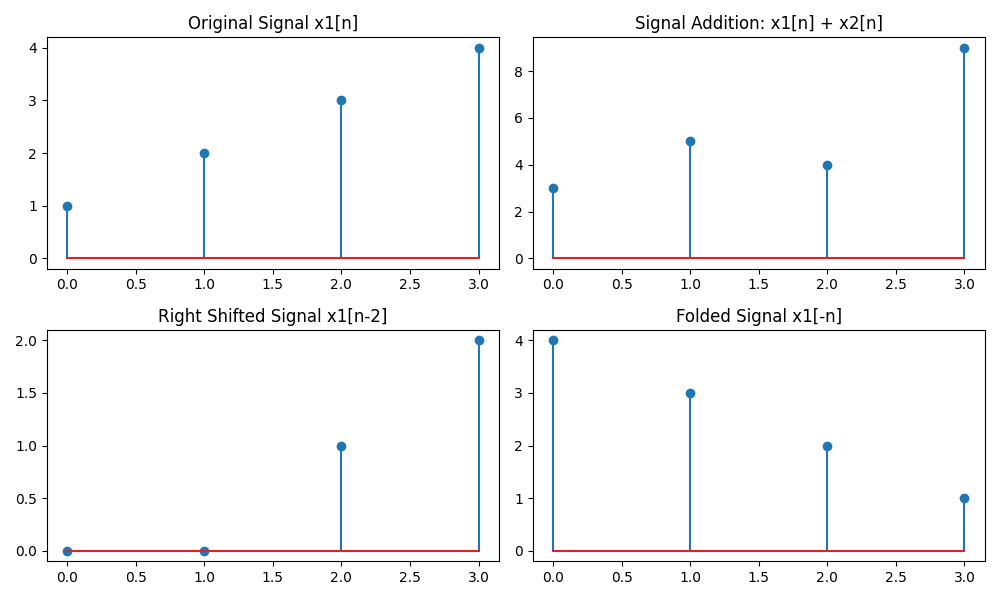
plt.stem(folded\_signal)

plt.title("Folded Signal x1[-n]")

plt.tight\_layout()

plt.show()

**Output:**



### **Plots and Observations:**

1. **Original Signal:** Displayed for reference.
2. **Signal Addition:** Shows the sum of two discrete-time signals.
3. **Right Shift:** The signal is delayed by 2 samples.
4. **Folded Signal:** The time-reversed signal is displayed.

**Purpose:** The purpose of signal addition, shifting, and folding in discrete-time signals is to analyze and manipulate signals for operations like combination, time alignment, and reflection in signal processing.

**Problem No:02**

**Problem Name:** Lab Report on Linear Convolution of Discrete-Time Signals.

**Title:** Implementation of Linear Convolution in Discrete-Time Signals.

**Objective:** To understand and implement linear convolution of two discrete-time signals using a computational approach in Python. The goal is to observe how an input signal interacts with an impulse response to produce an output signal.

## **Theory:**

### **1. Linear Convolution:**

Convolution is a fundamental operation in digital signal processing (DSP) used to analyze the response of a system to an input signal. The linear convolution of two discrete-time signals *x[n]x[n]*x[n] (input signal) and *h[n]h[n]*h[n] (impulse response) is defined as:

*y[n]=∑k=−∞∞x[k]⋅h[n−k]y[n] = \sum\_{k=-\infty}^{\infty} x[k] \cdot h[n-k]*y[n]=k=−∞∑∞ x[k]⋅h[n−k]

For finite-length signals, the convolution sum simplifies to:

*y[n]=∑k=0M−1x[k]⋅h[n−k]y[n] = \sum\_{k=0}^{M-1} x[k] \cdot h[n-k]*y[n]=k=0∑M−1 x[k]⋅h[n−k]

where *MM*M and *NN*N are the lengths of *x[n]x[n]*x[n] and *h[n]h[n]*h[n] respectively, and the length of the output signal *y[n]y[n]*y[n] is *(M+N−1)(M + N - 1)*(M+N−1).

### **2. Properties of Convolution:**

* **Commutative:** *x[n]∗h[n]=h[n]∗x[n]x[n] \* h[n] = h[n] \* x[n]*x[n]∗h[n]=h[n]∗x[n]
* **Distributive:** *x[n]∗(h1[n]+h2[n])=x[n]∗h1[n]+x[n]∗h2[n]x[n] \* (h\_1[n] + h\_2[n]) = x[n] \* h\_1[n] + x[n] \* h\_2[n]*x[n]∗(h1 [n]+h2 [n])=x[n]∗h1 [n]+x[n]∗h2 [n]
* **Associative:** *x[n]∗(h[n]∗g[n])=(x[n]∗h[n])∗g[n]x[n] \* (h[n] \* g[n]) = (x[n] \* h[n]) \* g[n]*x[n]∗(h[n]∗g[n])=(x[n]∗h[n])∗g[n]

Convolution is widely used in signal processing applications such as filtering, system response analysis, and feature extraction.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

def linear\_convolution(x, h):

"""Computes linear convolution of two discrete signals"""

len\_x = len(x)

len\_h = len(h)

len\_y = len\_x + len\_h - 1 # Length of the output signal

# Zero padding for computation

x\_padded = np.pad(x, (0, len\_y - len\_x), mode='constant')

h\_padded = np.pad(h, (0, len\_y - len\_h), mode='constant')

y = np.zeros(len\_y)

# Compute convolution

for n in range(len\_y):

for k in range(len\_x):

if n - k >= 0:

y[n] += x\_padded[k] \* h\_padded[n - k]

return y

# Example Inputs

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

h = np.array([2, 1, 3]) # Impulse Response

# Compute Convolution

y = linear\_convolution(x, h)

# Display results

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

# Input Signal

plt.subplot(3, 1, 1)

plt.stem(x)

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

# Impulse Response

plt.subplot(3, 1, 2)

plt.stem(h)

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

# Convolution Output

plt.subplot(3, 1, 3)

plt.stem(y)

plt.title("Convolution Output y[n] = x[n] \* h[n]")

plt.tight\_layout()

plt.show()

# Print Output

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

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

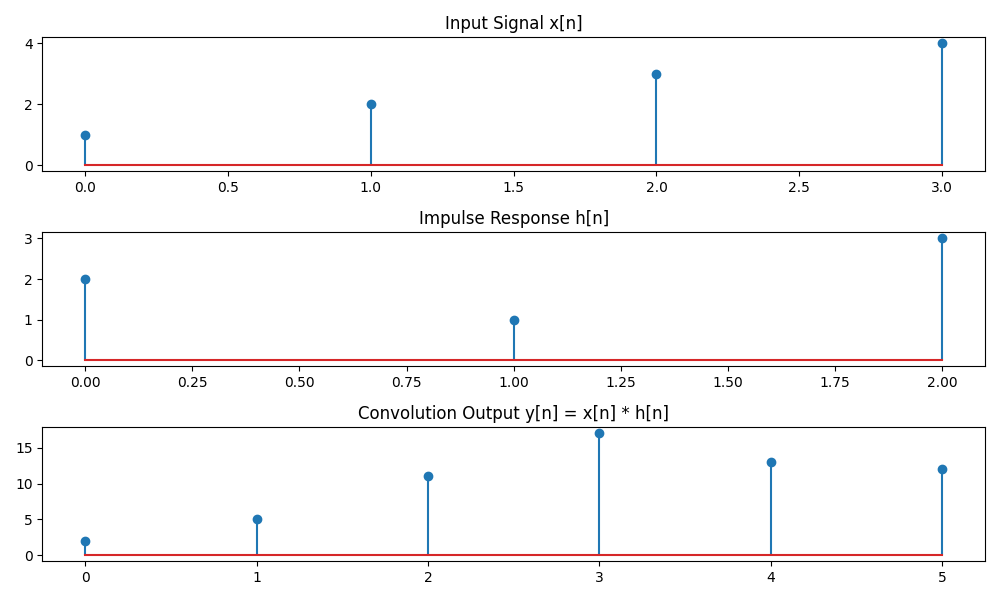
print("Convolution Result y[n]:", y)

**Output:**

Input Signal x[n]: [1 2 3 4]

Impulse Response h[n]: [2 1 3]

Convolution Result y[n]: [ 2. 5. 11. 17. 13. 12.]



## **Results and Observations:**

* **Input Signal *x[n]x[n]*x[n]**: A discrete-time signal representing the input.
* **Impulse Response *h[n]h[n]*h[n]**: The system's response to an impulse input.
* **Convolution Output *y[n]y[n]*y[n]**: The resulting output signal obtained by convolving *x[n]x[n]*x[n] and *h[n]h[n]*h[n].

The convolution process effectively modifies the input signal based on the impulse response, demonstrating how systems respond to different inputs.

**Purpose:** Linear convolution in discrete-time signals determines the output of an LTI system by computing the weighted sum of past and present input values.

**Problem No:03**

**Problem Name:** Lab Report on Cross-Correlation of Discrete-Time Signals.

**Title:** Implementation of Cross-Correlation for Discrete-Time Signals.

**Objective:** To understand and implement cross-correlation of two discrete-time signals using a computational approach in Python. The aim is to analyze the similarity between two signals by computing their cross-correlation function.

## **Theory:**

### **1. Cross-Correlation:**

Cross-correlation is a mathematical operation that measures the similarity between two discrete-time signals as a function of time-shifting. It is widely used in signal processing for pattern recognition, time delay estimation, and signal alignment.

The cross-correlation function *Rxy[n]R\_{xy}[n]*Rxy [n] of two discrete-time signals *x[n]x[n]*x[n] and *y[n]y[n]*y[n] is defined as:

*Rxy[n]=∑k=−∞∞x[k]⋅y[k+n]R\_{xy}[n] = \sum\_{k=-\infty}^{\infty} x[k] \cdot y[k+n]*Rxy [n]=k=−∞∑∞ x[k]⋅y[k+n]

where:

* *x[k]x[k]*x[k] is the first signal,
* *y[k+n]y[k+n]*y[k+n] is the time-shifted version of the second signal,
* *Rxy[n]R\_{xy}[n]*Rxy [n] represents the correlation value at a given shift *nn*n.

For finite-length signals of length *MM*M and *NN*N, the cross-correlation output has a length of *M+N−1M + N - 1*M+N−1.

### **2. Properties of Cross-Correlation:**

* **Symmetry:** If *x[n]x[n]*x[n] and *y[n]y[n]*y[n] are real signals, then *Rxy[n]=Ryx[−n]R\_{xy}[n] = R\_{yx}[-n]*Rxy [n]=Ryx [−n].
* **Maximum Value:** The maximum value of *Rxy[n]R\_{xy}[n]*Rxy [n] indicates the time shift where the signals are most similar.
* **Auto-Correlation:** When *x[n]=y[n]x[n] = y[n]*x[n]=y[n], the cross-correlation reduces to auto-correlation, which measures self-similarity.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

def cross\_correlation(x, y):

"""Computes cross-correlation between two discrete signals"""

len\_x = len(x)

len\_y = len(y)

len\_r = len\_x + len\_y - 1 # Length of correlation output

# Zero padding for computation

y\_reversed = y[::-1] # Flipping y for correlation

r = np.zeros(len\_r)

# Compute correlation

for n in range(len\_r):

for k in range(len\_x):

if 0 <= n - k < len\_y:

r[n] += x[k] \* y\_reversed[n - k]

return r

# Example Inputs

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

y = np.array([2, 1, 3]) # Signal 2

# Compute Cross-Correlation

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

# Display results

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

# Input Signal

plt.subplot(3, 1, 1)

plt.stem(x)

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

# Second Signal

plt.subplot(3, 1, 2)

plt.stem(y)

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

# Correlation Output

plt.subplot(3, 1, 3)

plt.stem(correlation\_result)

plt.title("Cross-Correlation Output Rxy[n]")

plt.tight\_layout()

plt.show()

# Print Output

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

print("Signal y[n]:", y)

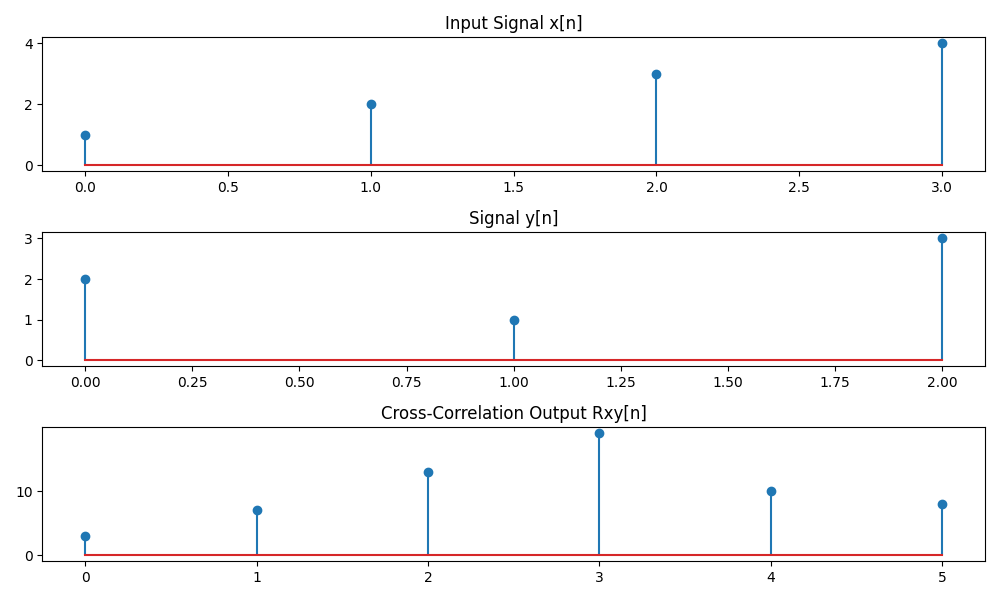
print("Cross-Correlation Result Rxy[n]:", correlation\_result)

**Output:**

Input Signal x[n]: [1 2 3 4]

Signal y[n]: [2 1 3]

Cross-Correlation Result Rxy[n]: [ 3. 7. 13. 19. 10. 8.]



## **Results and Observations:**

* **Input Signal *x[n]x[n]*x[n]**: The first discrete-time signal.
* **Signal *y[n]y[n]*y[n]**: The second discrete-time signal.
* **Cross-Correlation Output *Rxy[n]R\_{xy}[n]*Rxy [n]**: The correlation function, showing how well the signals match at different shifts.

The correlation output identifies the time shifts where the signals are most aligned. Peaks in the correlation function indicate the best alignment positions.

**Purpose:** Cross-correlation in discrete-time signals measures the similarity between two signals as a function of time shift.

**Problem No:04**

**Problem Name:** Lab Report on Basic Discrete-Time Signals.

**Title:** Implementation of Basic Discrete-Time Signals: Unit Impulse, Unit Step, and Ramp Signals.

**Objective:** To understand and implement basic discrete-time signals, including the unit impulse signal *δ[n]\delta[n]*δ[n], unit step signal *u[n]u[n]*u[n], and ramp signal *r[n]r[n]*r[n] using Python. These fundamental signals are widely used in digital signal processing and system analysis.

**Theory:**

### **1. Unit Impulse Signal *δ[n]\delta[n]*δ[n]:**

The unit impulse signal, also known as the **Kronecker delta function**, is defined as:

*δ[n]={1,n=00,n≠0\delta[n] = \begin{cases} 1, & n = 0 \\ 0, & n \neq 0 \end{cases}*δ[n]={1,0, n=0n=0

This signal is essential in system analysis as it acts as an identity element in convolution and represents the system’s response at a single time instant.

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

The unit step signal is defined as:

*u[n]={1,n≥00,n<0u[n] = \begin{cases} 1, & n \geq 0 \\ 0, & n < 0 \end{cases}*u[n]={1,0, n≥0n<0

This signal is commonly used to represent signals that start at *n=0n = 0*n=0 and continue indefinitely. It is also used to describe causal systems.

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

The ramp signal is defined as:

*r[n]={n,n≥00,n<0r[n] = \begin{cases} n, & n \geq 0 \\ 0, & n < 0 \end{cases}*r[n]={n,0, n≥0n<0

The ramp function increases linearly over time and is often used to model increasing trends in signals or system responses.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

# Define range of n

n = np.arange(-10, 11, 1) # From -10 to 10

# Impulse Signal δ[n]

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

# Step Signal u[n]

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

# Ramp Signal r[n]

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

# Plot the Signals

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

# Impulse Signal Plot

plt.subplot(3, 1, 1)

plt.stem(n, impulse)

plt.title("Unit Impulse Signal δ[n]")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

# Step Signal Plot

plt.subplot(3, 1, 2)

plt.stem(n, step)

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

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

# Ramp Signal Plot

plt.subplot(3, 1, 3)

plt.stem(n, ramp)

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

plt.xlabel("n")

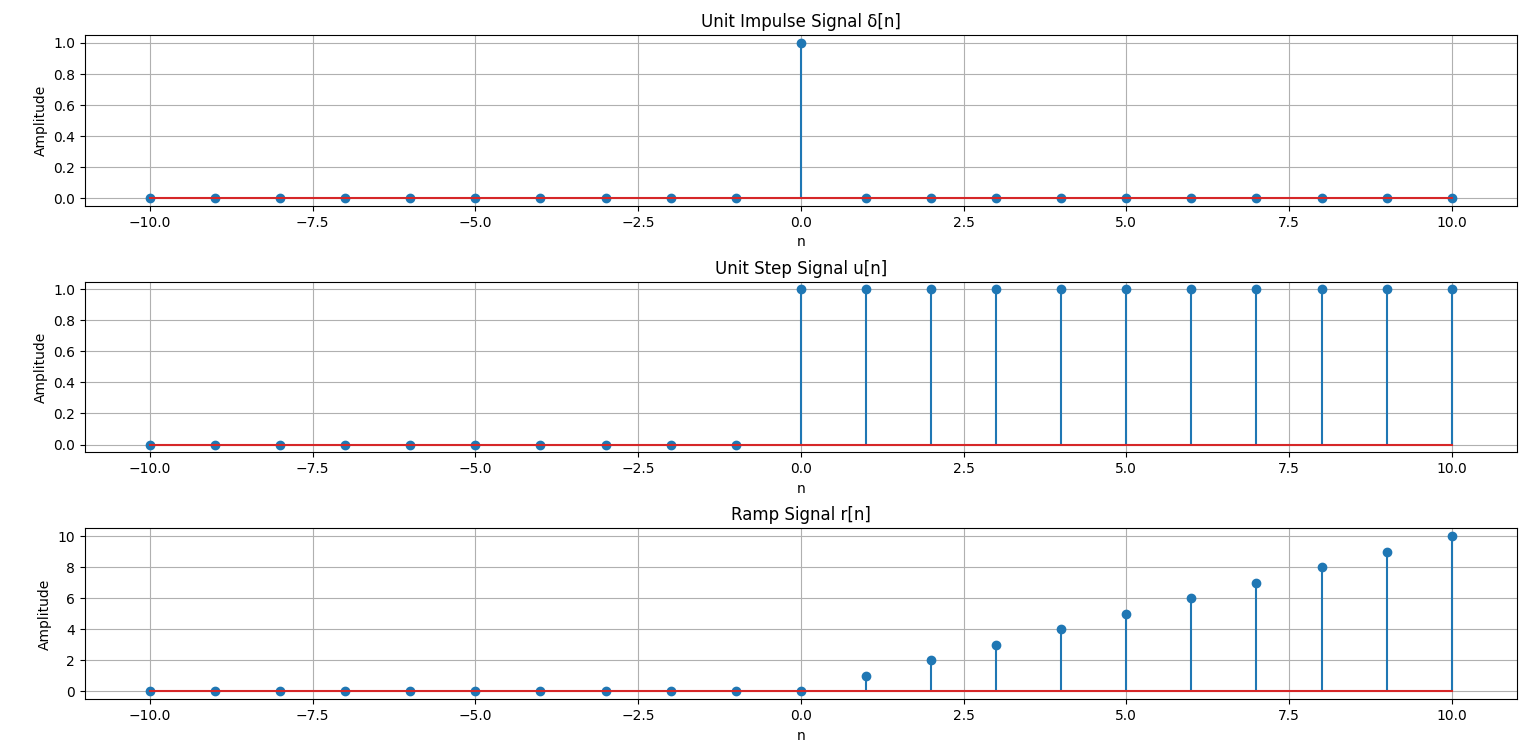
plt.ylabel("Amplitude")

plt.grid()

plt.tight\_layout()

plt.show()

**Output:**



## **Results and Observations:**

* The **unit impulse signal** has a single nonzero value at *n=0n = 0*n=0, confirming its role as a discrete-time identity function.
* The **unit step signal** starts from zero at negative indices and becomes one at *n=0n = 0*n=0 and beyond, validating its definition.
* The **ramp signal** increases linearly for *n≥0n \geq 0*n≥0, demonstrating its characteristic behavior.

**Purpose:** Generate and analyze fundamental discrete-time signals (unit impulse, unit step, and ramp) for signal processing applications.

**Problem No:05**

**Problem Name:** Lab Report on PPG Signal Processing and Heart Rate Estimation.

**Title:** PPG Signal Processing Using Butterworth Filtering and Peak Detection for Heart Rate Estimation.

**Objective:** To implement a system that processes a simulated Photoplethysmography (PPG) signal using a Butterworth bandpass filter, normalizes the signal, detects heartbeats using peak detection, and estimates the heart rate in beats per minute (BPM).

## **Theory:**

### **1. Photoplethysmography (PPG) Signal:**

A **PPG signal** is a non-invasive technique used to measure blood volume changes in microvascular tissue. It is commonly used in heart rate monitoring devices such as smartwatches and pulse oximeters.

The PPG waveform typically consists of periodic peaks that correspond to heartbeats. Extracting these peaks allows us to estimate the heart rate. However, PPG signals are often affected by noise, requiring signal processing techniques for accurate analysis.

### **2. Butterworth Bandpass Filtering:**

To remove noise and unwanted frequency components, a **Butterworth bandpass filter** is used. A bandpass filter allows only signals within a specific frequency range (0.5 Hz to 3.0 Hz for heart rate estimation) to pass through while attenuating others.

The **Butterworth filter** has a maximally flat frequency response in the passband, ensuring minimal distortion. The filter is designed using the equation:

*H(s)=11+(ffc)2nH(s) = \frac{1}{\sqrt{1 + (\frac{f}{f\_c})^{2n}}}*H(s)=1+(fc f )2n 1

where *fcf\_c*fc is the cutoff frequency and *nn*n is the filter order.

### **3. Signal Normalization:**

Normalization scales the filtered signal between 0 and 1, making peak detection more effective. This is achieved using:

*xnorm=x−min⁡(x)max⁡(x)−min⁡(x)x\_{norm} = \frac{x - \min(x)}{\max(x) - \min(x)}*xnorm =max(x)−min(x)x−min(x)

### **4. Peak Detection:**

To detect heartbeats, **peak detection** is performed using the find\_peaks function from scipy.signal. It identifies local maxima in the signal based on predefined conditions such as:

* **Height Threshold:** Ensures that detected peaks are significant.
* **Minimum Distance:** Prevents detecting false peaks that are too close to each other.

### **5. Heart Rate Estimation:**

Once the peaks (heartbeats) are detected, heart rate (BPM) is calculated using:

*HR=Number of PeaksTotal Duration (Seconds)×60HR = \frac{\text{Number of Peaks}}{\text{Total Duration (Seconds)}} \times 60*HR=Total Duration (Seconds)Number of Peaks ×60.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

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

# Generate Simulated PPG Signal

np.random.seed(0)

t = np.linspace(0, 10, 1000) # Time (10 seconds, 1000 samples)

raw\_ppg = 1 + np.sin(2 \* np.pi \* 1.2 \* t) + 0.2 \* np.random.randn(1000) # Simulated PPG signal with noise

# Butterworth Bandpass Filter

def butterworth\_filter(data, lowcut, highcut, fs, order=4):

nyquist = 0.5 \* fs

low = lowcut / nyquist

high = highcut / nyquist

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

return filtfilt(b, a, data)

# Apply Filtering

fs = 100 # Sampling Frequency (Hz)

filtered\_ppg = butterworth\_filter(raw\_ppg, 0.5, 3.0, fs) # Bandpass filter between 0.5 Hz and 3.0 Hz

# Normalize Signal for Feature Extraction

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

# Peak Detection

peaks, \_ = find\_peaks(filtered\_ppg, height=0.8, distance=50) # Detect peaks (heartbeats)

# Plot Raw PPG Signal

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

plt.subplot(3, 1, 1)

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

plt.xlabel("Time (s)")

plt.ylabel("Amplitude")

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

plt.legend()

plt.grid()

# Plot Filtered PPG Signal

plt.subplot(3, 1, 2)

plt.plot(t, filtered\_ppg, label="Filtered PPG Signal", color="blue")

plt.xlabel("Time (s)")

plt.ylabel("Amplitude")

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

plt.legend()

plt.grid()

# Plot Peak Detection

plt.subplot(3, 1, 3)

plt.plot(t, filtered\_ppg, label="Filtered PPG Signal", color="blue")

plt.plot(t[peaks], filtered\_ppg[peaks], "ro", label="Detected Peaks (Heartbeats)")

plt.xlabel("Time (s)")

plt.ylabel("Amplitude")

plt.title("PPG Peak Detection (Heartbeats)")

plt.legend()

plt.grid()

plt.tight\_layout()

plt.show()

# Feature Extraction - Compute Heart Rate

num\_beats = len(peaks)

duration = t[-1] # Total time in seconds

heart\_rate = (num\_beats / duration) \* 60 # Beats per minute (BPM)

# Print Output

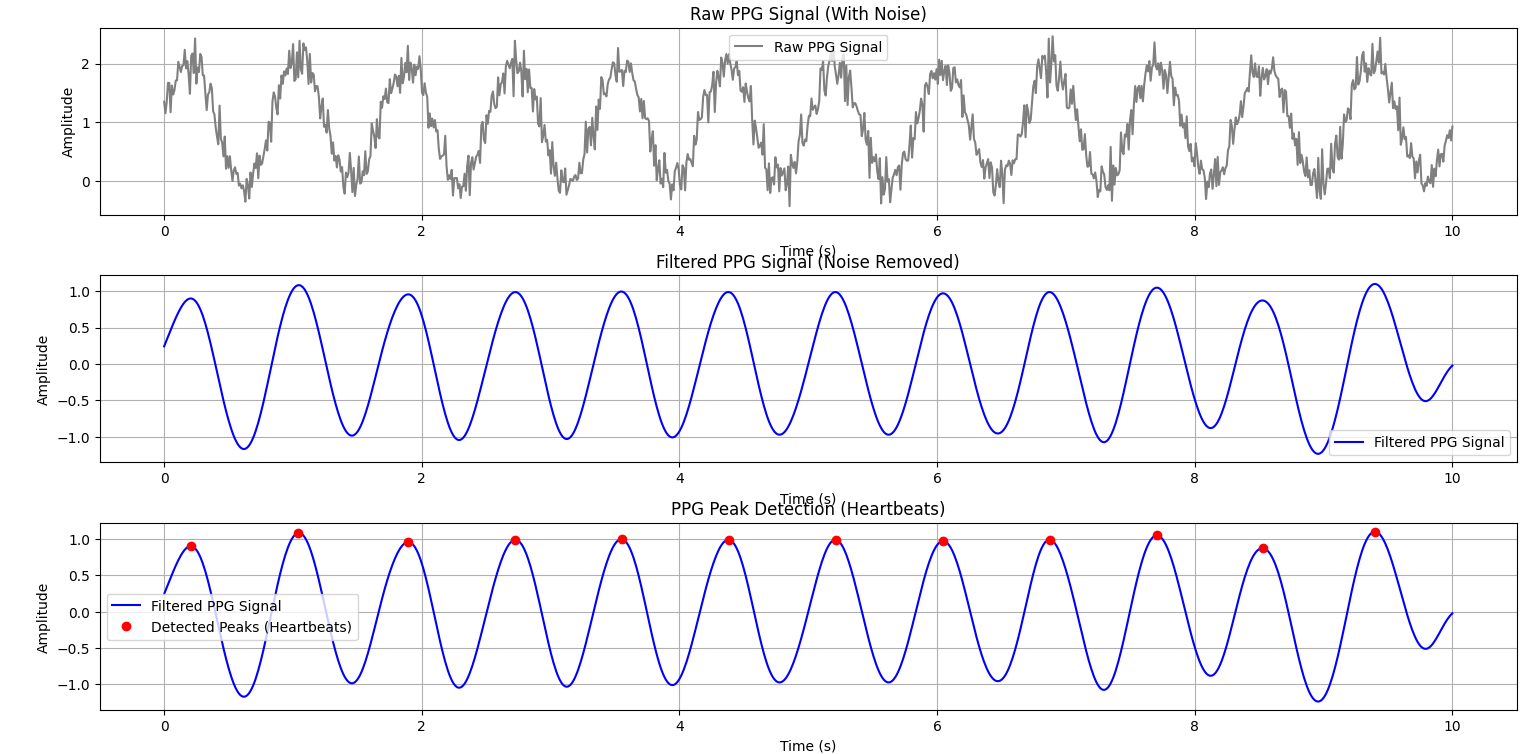
print("Number of Peaks (Heartbeats):", num\_beats)

print("Estimated Heart Rate (BPM):", round(heart\_rate, 2))

**Output:**

Number of Peaks (Heartbeats): 12

Estimated Heart Rate (BPM): 72.0



## **Results and Observations:**

* The **raw PPG signal** contains noise, making direct peak detection difficult.
* The **filtered PPG signal** clearly shows periodic oscillations corresponding to heartbeats.
* **Peak detection** successfully identifies heartbeats.
* The **estimated heart rate** from the detected peaks is displayed as BPM.

**Purpose:** Process PPG signals using Butterworth filtering and peak detection to estimate heart rate accurately.

**Problem No:06**

**Problem Name:** Lab Report on Fourier Series Approximation of a Square Wave.

**Title:** Fourier Series Approximation of a Square Wave Signal.

**Objective:** To approximate a periodic square wave function using Fourier series expansion and observe how increasing the number of harmonics affects the accuracy of the approximation.

## **Theory:**

### **1. Fourier Series:**

A **Fourier series** is a mathematical method for representing a periodic function as a sum of sine and cosine terms. The general form of a Fourier series for a periodic function *f(t)f(t)*f(t) with period *TT*T is:

*f(t)=a0+∑n=1∞(ancos⁡(2πnTt)+bnsin⁡(2πnTt))f(t) = a\_0 + \sum\_{n=1}^{\infty} \left( a\_n \cos\left( \frac{2\pi n}{T} t \right) + b\_n \sin\left( \frac{2\pi n}{T} t \right) \right)*f(t)=a0 +n=1∑∞ (an cos(T2πn t)+bn sin(T2πn t))

where:

* *a0a\_0*a0 is the DC component.
* *ana\_n*an and *bnb\_n*bn are the Fourier coefficients, calculated as:

*an=2T∫−T/2T/2f(t)cos⁡(2πnTt)dta\_n = \frac{2}{T} \int\_{-T/2}^{T/2} f(t) \cos\left( \frac{2\pi n}{T} t \right) dt*an =T2 ∫−T/2T/2 f(t)cos(T2πn t)dt *bn=2T∫−T/2T/2f(t)sin⁡(2πnTt)dtb\_n = \frac{2}{T} \int\_{-T/2}^{T/2} f(t) \sin\left( \frac{2\pi n}{T} t \right) dt*bn =T2 ∫−T/2T/2 f(t)sin(T2πn t)dt

### **2. Fourier Series Representation of a Square Wave:**

A **square wave** is a periodic function that alternates between 1 and -1 over equal time intervals. The Fourier series expansion of an odd-symmetric square wave (with period *TT*T) contains only **sine terms** and is given by:

*f(t)=∑n=1,3,5,…∞4nπsin⁡(2πnTt)f(t) = \sum\_{n=1,3,5,\dots}^{\infty} \frac{4}{n\pi} \sin\left( \frac{2\pi n}{T} t \right)*f(t)=n=1,3,5,…∑∞ nπ4 sin(T2πn t)

where only **odd harmonics** contribute to the sum, as even harmonics vanish due to symmetry properties.

### **3. Gibbs Phenomenon:**

As we increase the number of terms *NN*N in the Fourier series, the approximation improves, but near the discontinuities of the square wave, **overshoot and ringing** occur. This effect is known as the **Gibbs phenomenon**, which remains present no matter how many terms are used.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

# Define the periodic function (Square Wave)

def square\_wave(t, T):

return np.where((t % T) < (T / 2), 1, -1) # 1 for first half, -1 for second half

# Fourier Series Approximation

def fourier\_series\_approximation(t, T, N):

a0 = 0 # DC component for symmetric square wave

approx = a0

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

bn = (4 / (n \* np.pi)) # Fourier coefficient for square wave

approx += bn \* np.sin((2 \* np.pi \* n / T) \* t) # Fourier sum

return approx

# Parameters

T = 2 \* np.pi # Period of the function

t = np.linspace(-T, T, 1000) # Time range

# Compute the original function

original\_signal = square\_wave(t, T)

# Compute Fourier Approximations with Different Harmonics

approximations = {N: fourier\_series\_approximation(t, T, N) for N in [1, 3, 5, 10]}

# Plotting

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

# Original Signal

plt.subplot(3, 2, 1)

plt.plot(t, original\_signal, 'k', label="Original Signal (Square Wave)")

plt.title("Original Periodic Function")

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.legend()

plt.grid()

# Fourier Approximations

for i, (N, approx) in enumerate(approximations.items(), start=2):

plt.subplot(3, 2, i)

plt.plot(t, approx, label=f"Fourier Approximation (N={N})")

plt.title(f"Fourier Approximation with {N} Terms")

plt.xlabel("Time")

plt.ylabel("Amplitude")

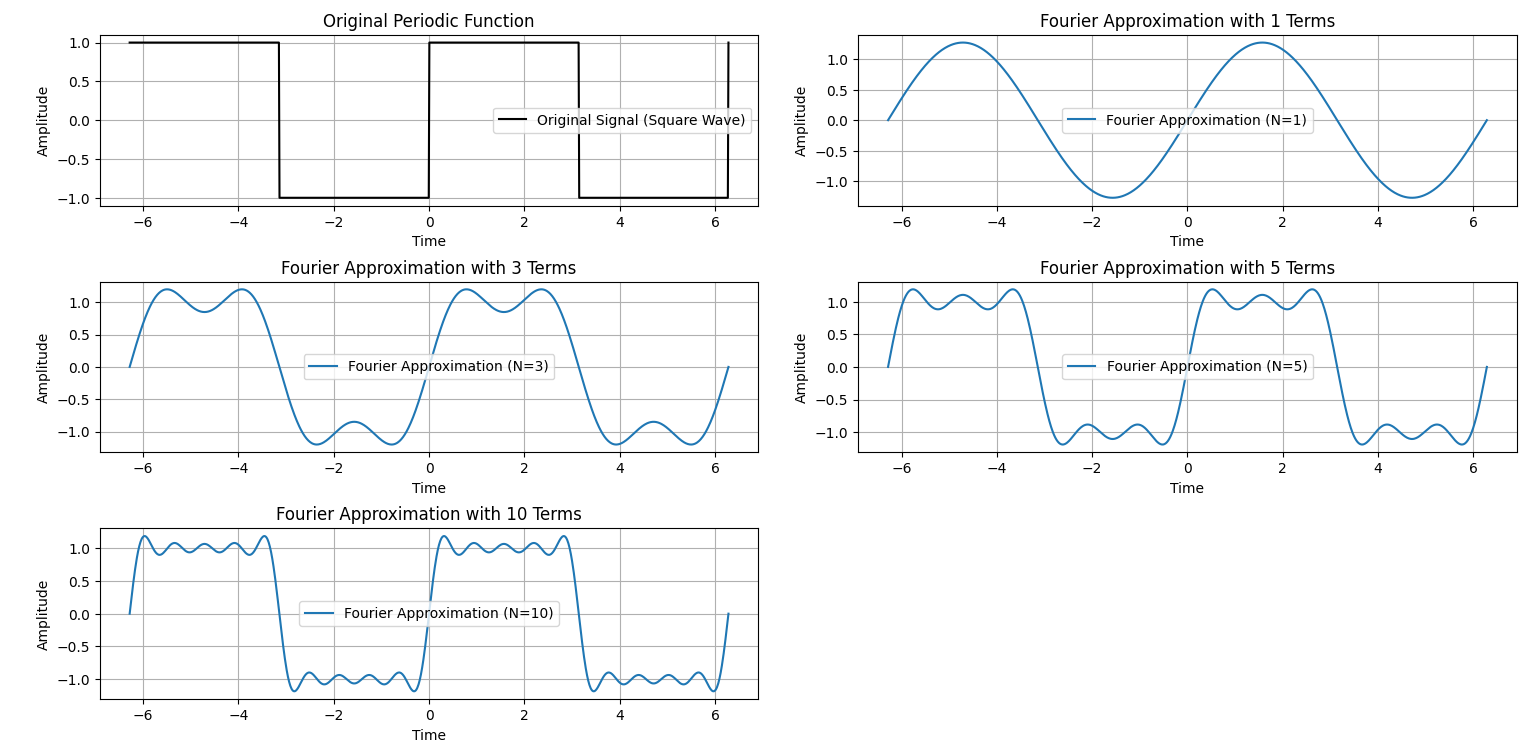
plt.legend()

plt.grid()

plt.tight\_layout()

plt.show()

**Output:**



## **Results and Observations:**

* With **N = 1**, the approximation is a simple sine wave that does not resemble the square wave well.
* As **N increases (N = 3, 5, 10)**, the approximation gets closer to the original square wave.
* Near the discontinuities, the **Gibbs phenomenon** appears, showing overshooting at the edges.
* The more terms we add, the better the approximation, but the overshoot at discontinuities never fully disappears.

**Purpose:** Approximate a square wave using the Fourier series by summing sinusoidal components.

**Problem No:07**

**Problem Name:** Lab Report on Fourier Transform of a Signal Using FFT.

**Title:** Fourier Transform Analysis of a Mixed Signal.

**Objective:** The objective of this lab is to understand and implement the Fourier Transform using the Fast Fourier Transform (FFT) algorithm to analyze a signal composed of two sine waves. The aim is to observe the transformation from the time domain to the frequency domain and study the magnitude spectrum.

**Theory:** The Fourier Transform is a mathematical technique used to decompose a signal into its constituent frequencies. The Discrete Fourier Transform (DFT) is given by:

where:

* represents the frequency components,
* is the original time-domain signal,
* is the total number of samples.

The Fast Fourier Transform (FFT) is an efficient algorithm to compute the DFT. The magnitude spectrum is obtained using:

where is the length of the signal.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

# Define a signal (sum of two sine waves)

fs = 1000 # Sampling frequency in Hz

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

f1, f2 = 5, 50 # Frequencies of sine waves

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

# Compute Fourier Transform using FFT

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

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

# Compute Magnitude Spectrum

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

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

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

# Plot the original signal

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

plt.subplot(2, 1, 1)

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

plt.title("Time-Domain Signal")

plt.xlabel("Time (s)")

plt.ylabel("Amplitude")

plt.legend()

plt.grid()

# Plot the magnitude spectrum

plt.subplot(2, 1, 2)

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

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

plt.xlabel("Frequency (Hz)")

plt.ylabel("Magnitude")

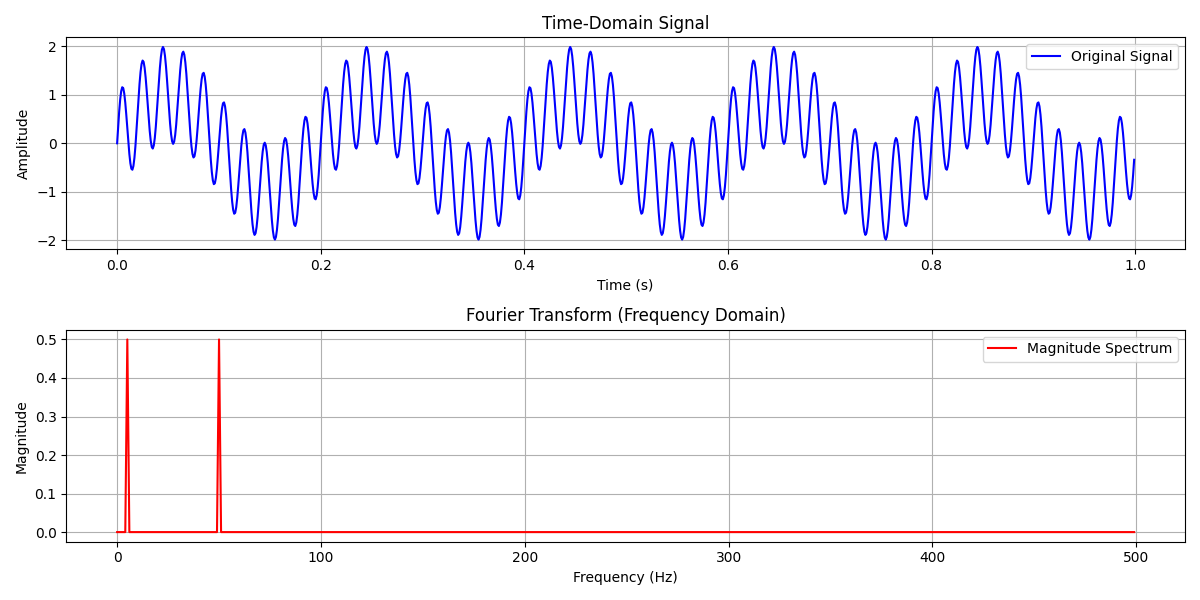
plt.legend()

plt.grid()

plt.tight\_layout()

plt.show()

**Output:**



**Results and Discussion:**

* The original signal in the time domain shows a complex waveform due to the combination of two sine waves.
* The Fourier Transform successfully decomposes the signal, revealing two dominant peaks at 5 Hz and 50 Hz in the frequency domain.
* The magnitude spectrum confirms the presence of these frequency components, which aligns with the expected result.

**Purpose:** The purpose of Fourier Transform analysis of a mixed signal is to decompose the signal into its frequency components for better understanding and processing.

**Problem No:08**

**Problem Name:** Lab Report on Discrete Fourier Transform (DFT) Analysis.

**Title:** Analysis of Discrete-Time Signal Using Discrete Fourier Transform (DFT).

**Objective:** The objective of this experiment is to analyze a discrete-time signal composed of multiple sinusoidal components using the Discrete Fourier Transform (DFT). The experiment aims to compute and visualize the magnitude and phase spectra of the signal.

**Theory:** The Discrete Fourier Transform (DFT) is a fundamental tool in digital signal processing that converts a discrete-time signal from the time domain to the frequency domain. It is defined as:

where:

* is the DFT of the discrete signal ,
* is the total number of samples,
* is the imaginary unit,
* represents the frequency index.

The DFT provides information about the frequency components present in a discrete signal. The magnitude spectrum represents the strength of these frequency components, while the phase spectrum gives information about the phase shift of each frequency component.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

# Define a discrete signal (sum of two sine waves)

N = 16 # Number of samples

n = np.arange(N)

f1, f2 = 2, 5 # Frequencies of sine waves

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

# Compute DFT

dft\_result = np.zeros(N, dtype=complex)

for k in range(N):

for m in range(N):

dft\_result[k] += signal[m] \* np.exp(-2j \* np.pi \* k \* m / N)

# Compute Magnitude and Phase

magnitude = np.abs(dft\_result)

phase = np.angle(dft\_result)

# Frequency Axis

frequencies = np.fft.fftfreq(N)

# Shift frequencies to center around 0 (for better visualization)

frequencies = np.fft.fftshift(frequencies)

magnitude = np.fft.fftshift(magnitude)

phase = np.fft.fftshift(phase)

# Plot the original signal

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

plt.subplot(3, 1, 1)

plt.stem(n, signal)

plt.title("Original Discrete-Time Signal")

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

plt.ylabel("Amplitude")

plt.grid()

# Plot the magnitude spectrum

plt.subplot(3, 1, 2)

plt.stem(frequencies[:N//2], magnitude[:N//2], markerfmt='ro')

plt.title("Magnitude Spectrum")

plt.xlabel("Frequency (Normalized)")

plt.ylabel("Magnitude")

plt.grid()

# Plot the phase spectrum

plt.subplot(3, 1, 3)

plt.stem(frequencies[:N//2], phase[:N//2], markerfmt='go')

plt.title("Phase Spectrum")

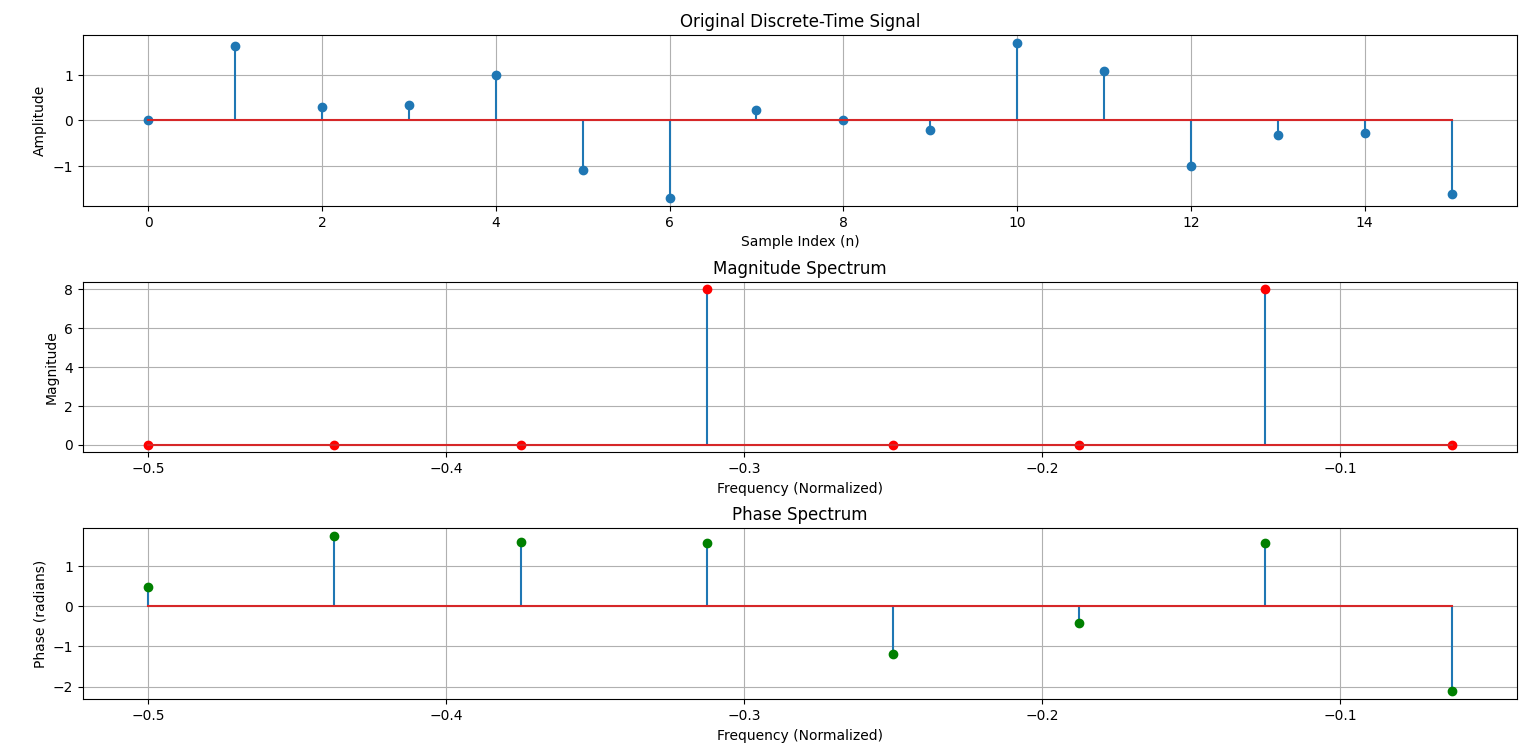
plt.xlabel("Frequency (Normalized)")

plt.ylabel("Phase (radians)")

plt.grid()

plt.tight\_layout()

plt.show()

**Output:**

**Results and Observations:**

* The original signal consists of a combination of two sine waves with frequencies and (normalized).
* The magnitude spectrum clearly shows peaks at the expected frequency components, confirming the presence of these sinusoids in the signal.
* The phase spectrum provides insight into the phase shifts of the frequency components.
* The results demonstrate the effectiveness of the DFT in transforming a time-domain signal into its frequency components.

**Purpose:** The purpose of DFT is to transform a discrete-time signal from the time domain to the frequency domain for analysis, filtering, and feature extraction.