**Pabna University of Science and Technology**



**Lab Report**

**Information and Communication Engineering**

**Course Title :Signals and Systems Sessional**

**Course Code:ICE-2204**

**Submitted By: Submitted To:**

**Name :Sumaiya Hossain Dr. Md. Imran Hossain**

**Roll: 220633 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 code of signal sequencing impulse signal ,step signal and ramp signal. |
| **02** | Write a code on signal operation(shifting,folding,adding) |
| **03** | Write a code on signal correlation. |
| **04** | Write a code on signal convolution. |
| **05** | Write a code on ppg signal-filtering ,feature extraction, peak detection. |
| **06** | Write a code on Fourier Series decomposition. |
| **07** | Write a code on Fourier Transform**.** |
| **08** | Write a code on DFT. |

**Experiment No:01**

**Experiment Name** :Write a code of signal sequencing impulse signal ,step signal and ramp signal.

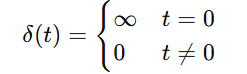
**Objective:**

The objective is to understand the properties and applications of different basic signals—impulse, step, and ramp signals—in signal processing and system analysis. These signals are fundamental for analyzing system behavior, responses, and control systems.

**Theory:**

**Impulse Signal:**

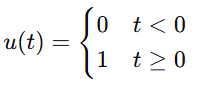
The impulse signal, often represented as δ(t)\delta(t)δ(t), is a mathematical idealization of a signal that has an infinitely small duration but an infinitely large amplitude, such that the integral of the signal over all time equals one:



It is used to model sudden, sharp events or inputs that occur at a specific point in time.

**Step Signal:**

The step signal, also known as the Heaviside step function, is a signal that changes its value from zero to one at a specific point in time. It is represented mathematically as:



The step signal represents a signal that "turns on" at t=0 and remains at a constant value thereafter.

**Ramp Signal:**

The ramp signal is a continuous signal that increases (or decreases) linearly over time. Mathematically, it is represented as:



The ramp function represents a steady, continuous increase in value, often used to model slowly changing inputs.

**Source 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")

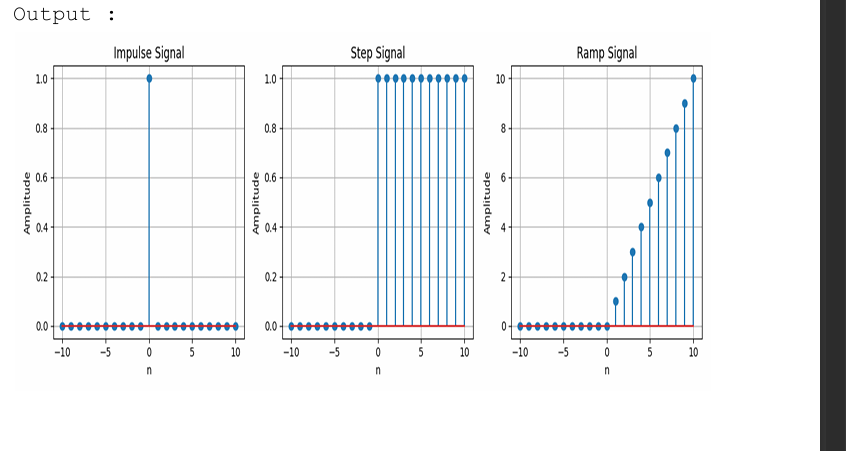
plt.ylabel("Amplitude")

plt.grid()

plt.tight\_layout()

plt.show()

**Output:**

****

**Experiment No:02**

**Experiment Name :**Write a code on signal operation(adding,multiplicating,ahifting,folding).

**Objective:**

The objective is to understand and implement basic signal operations—**addition**, **multiplication**, **shifting**, and **folding**—on discrete-time signals. These operations help in modifying and analyzing signals in various systems, enabling a better understanding of signal behavior and system responses.

**Theory:**

1. **Signal Addition:**
   * The addition of two signals is performed element-wise, where the value of the resulting signal at each time instant is the sum of the values of the individual signals at that time.
   * Mathematically: y[n]=x1[n]+x2[n]y[n] = x\_1[n] + x\_2[n]y[n]=x1​[n]+x2​[n] where x1[n]x\_1[n]x1​[n] and x2[n]x\_2[n]x2​[n] are two signals.
2. **Signal Multiplication:**
   * The multiplication of two signals is also performed element-wise, where the value of the resulting signal at each time instant is the product of the values of the individual signals at that time.
   * Mathematically: y[n]=x1[n]⋅x2[n]y[n] = x\_1[n] \cdot x\_2[n]y[n]=x1​[n]⋅x2​[n] where x1[n]x\_1[n]x1​[n] and x2[n]x\_2[n]x2​[n] are two signals.
3. **Signal Shifting:**
   * Shifting a signal involves changing its time index. A positive shift corresponds to delaying the signal, while a negative shift corresponds to advancing it.
   * Mathematically:
     + **Right Shift (delay)**: y[n]=x[n−k]y[n] = x[n - k]y[n]=x[n−k], where k>0k > 0k>0
     + **Left Shift (advance)**: y[n]=x[n+k]y[n] = x[n + k]y[n]=x[n+k], where k>0k > 0k>0
4. **Signal Folding (Time Reversal):**
   * Folding a signal (or time reversal) means flipping the signal around the vertical axis. This operation reverses the direction of time.
   * Mathematically: y[n]=x[−n]y[n] = x[-n]y[n]=x[−n] where x[n]x[n]x[n] is the original signal and x[−n]x[-n]x[−n] is its folded version.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

# Define the range of n

n = np.arange(-10, 11)

# Define some basic signals

x1 = np.where(n >= 0, 1, 0) # Step signal (u[n])

x2 = np.where(n >= 0, n, 0) # Ramp signal (r[n])

# Signal Addition: x1[n] + x2[n]

addition = x1 + x2

# Signal Multiplication: x1[n] \* x2[n]

multiplication = x1 \* x2

# Signal Shifting: Right shift (delay by 3) and Left shift (advance by 3)

right\_shift = np.roll(x1, 3) # Right shift (delay by 3)

left\_shift = np.roll(x1, -3) # Left shift (advance by 3)

# Signal Folding (Time Reversal)

folding = np.flip(x1) # Folding (x[-n])

# Plotting all the operations

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

# Plot for signal addition

plt.subplot(3, 2, 1)

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

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

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

# Plot for signal multiplication

plt.subplot(3, 2, 2)

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

plt.title("Signal Multiplication (x1[n] \* x2[n])")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

# Plot for signal shifting (Right Shift)

plt.subplot(3, 2, 3)

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

plt.title("Signal Shifting (Right Shift by 3)")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

# Plot for signal shifting (Left Shift)

plt.subplot(3, 2, 4)

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

plt.title("Signal Shifting (Left Shift by 3)")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

# Plot for signal folding (Time Reversal)

plt.subplot(3, 2, 5)

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

plt.title("Signal Folding (Time Reversal)")

plt.xlabel("n")

plt.ylabel("Amplitude")

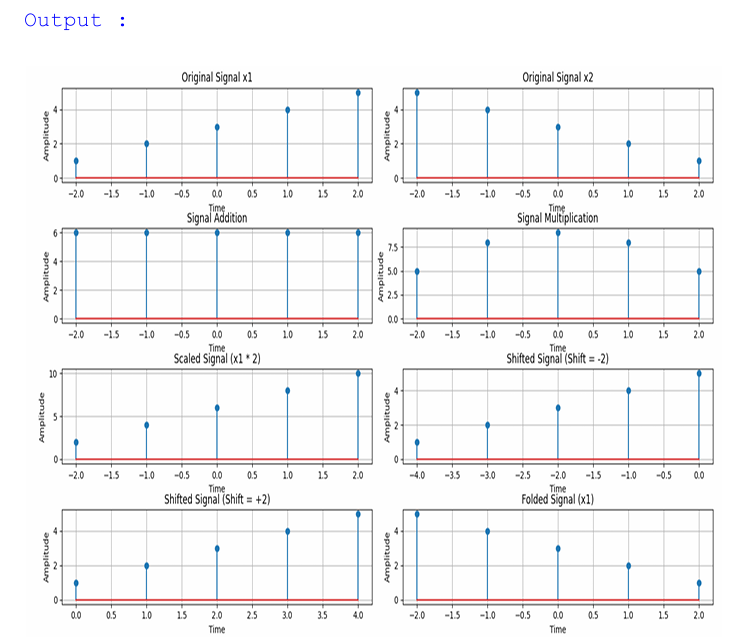
plt.grid()

# Adjust layout for the plots

plt.tight\_layout()

plt.show()

**Output:**



**Experiment No:03**

**Experiment Name** :Write a code on signal correlation.

**Objective:**

The objective is to compute and understand **signal correlation**, which is a measure of similarity between two signals as a function of the time-lag applied to one of them. Correlation is widely used in signal processing for tasks such as detecting signals, identifying patterns, and analyzing relationships between signals.

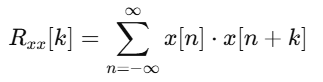
**Theory:**

Signal correlation is a mathematical operation that determines how much one signal resembles another when shifted by a certain amount (lag). There are two types of correlation commonly used:

* **Autocorrelation**: The correlation of a signal with itself, which helps identify repeating patterns and periodicity within the signal.
* **Cross-correlation**: The correlation between two different signals, which is useful for identifying similarities and detecting patterns in one signal relative to another.

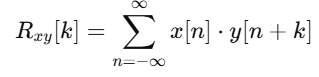
Mathematically:

1. **Autocorrelation** of a signal x[n] is defined as:



where k represents the lag or shift.

1. **Cross-correlation** between two signals x[n] and y[n] is defined as:



**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import correlate, correlation\_lags

# Function to compute autocorrelation

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

# Function to compute cross-correlation

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

# Sampling 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 the sinusoidal signal

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

# Compute autocorrelation of the sine wave

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

# Create another signal by shifting the original signal

signal1 = sin\_signal

signal2 = np.roll(signal1, 100) # Shift by 100 samples

# Compute cross-correlation between the two signals

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

# Add noise to the original signal

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

noisy\_signal = signal1 + noise

# Compute cross-correlation with the noisy signal

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

# Plot the results

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

# Plot the autocorrelation of the sinusoidal signal

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()

# Plot the cross-correlation between the two signals

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()

# Plot the cross-correlation with the noisy signal

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")

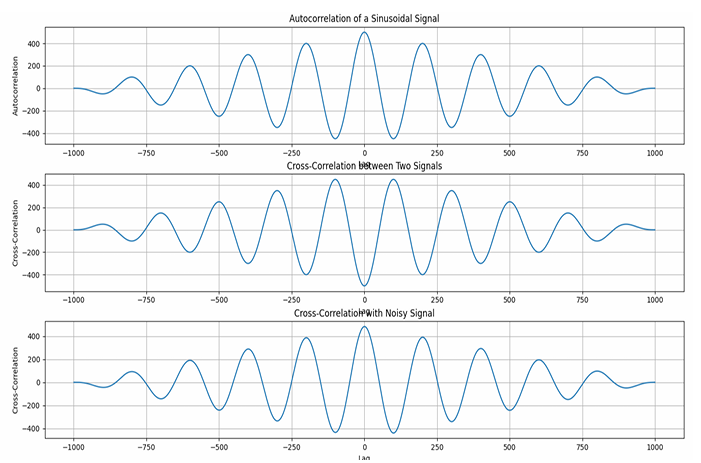
plt.grid()

# Adjust layout for better spacing

plt.tight\_layout()

plt.show()

**Output:**

****

**Experiment No: 04**

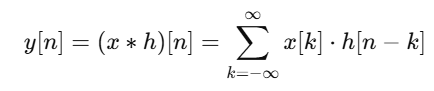
**Experiment Name :Write a code on signal convolution.**

**Objective:**

The objective is to implement and understand the convolution operation between two discrete-time signals, which helps in determining the output of a system when the input signal and system's impulse response are known.

**Theory:**

Convolution is an operation that combines two signals to produce a third signal. It is widely used in linear systems to find the system's output when the system's impulse response and input signal are known. The mathematical definition of the convolution of two discrete-time signals x[n] and h[n] is given by:



Where:

* x[n] is the input signal.
* h[n] is the impulse response of the system.
* y[n] is the output signal.
* The sum is taken over all valid values of k for which both x[k] and h[n−k] exist.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import convolve

# Function to compute convolution of two signals

def compute\_convolution(signal1, signal2):

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

return conv\_result

# Sampling 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 the sinusoidal signal

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

# Compute autoconvolution of the sine wave

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

# Create another signal by shifting the original signal

signal1 = sin\_signal

signal2 = np.roll(signal1, 100) # Shift by 100 samples

# Compute convolution between the original and the shifted signal

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

# Add noise to the original signal

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

noisy\_signal = signal1 + noise

# Compute convolution with the noisy signal

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

# Plot the results

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

# Plot the autoconvolution of the sine signal

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()

# Plot the convolution between the original and shifted signal

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()

# Plot the convolution with the noisy signal

plt.subplot(3, 1, 3)

plt.plot(conv\_noisy)

plt.title("Convolution with Noisy Signal")

plt.xlabel("Samples")

plt.ylabel("Convolution Output")

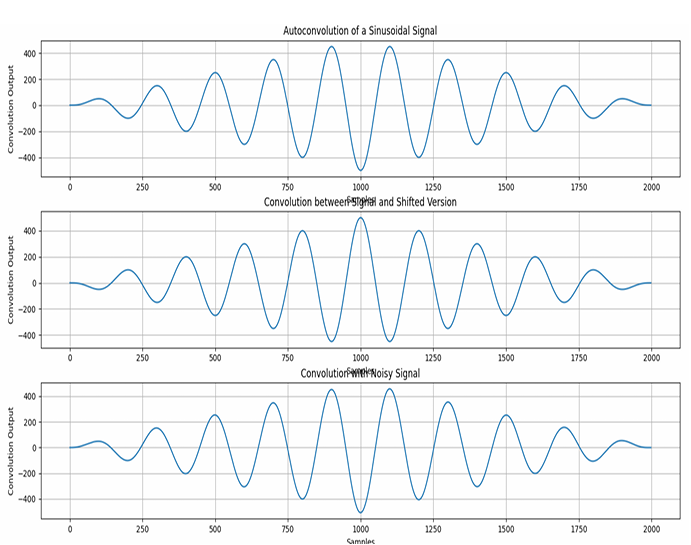
plt.grid()

# Adjust layout for better spacing

plt.tight\_layout()

plt.show()

**Output:**

****

**Experiment No:05**

**Experiment Name :** Write a code on ppg signal-filtering ,feature extraction, peak detection.

**Objective:**

The objective is to filter a noisy PPG signal, extract key features like heart rate (HR) and pulse rate variability (PRV), and detect the peaks corresponding to heartbeats for health monitoring applications.

**Theory:**

1. **PPG Signal Filtering:**
   * The raw PPG signal contains noise due to movement artifacts, ambient light changes, and other environmental factors. Therefore, it is necessary to apply **signal filtering** (usually low-pass, high-pass, or band-pass filters) to remove unwanted noise and extract the true physiological signal.
   * Common filters used for PPG signal processing:
     + **Low-pass filter**: Removes high-frequency noise (e.g., muscle movement).
     + **High-pass filter**: Removes baseline drift (e.g., due to respiration or body movement).
     + **Band-pass filter**: A combination of low-pass and high-pass filters that focus on a frequency range typically of 0.5 Hz to 5 Hz, which corresponds to the heart rate frequency range.
2. **Feature Extraction:**
   * **Heart Rate (HR):** The number of peaks (heartbeats) detected in the PPG signal within a specific period (typically in beats per minute, BPM).
   * **Pulse Rate Variability (PRV):** Variability in the time intervals between heartbeats. This can be extracted from the time differences between detected peaks.
   * **Peak Amplitude:** The peak value of the PPG signal at each heartbeat.
3. **Peak Detection:**
   * **Peak detection** is crucial for identifying the locations of heartbeats in the PPG signal. The process involves finding the local maxima in the filtered PPG signal that correspond to the systolic peaks of the heartbeat.
   * Algorithms like **find\_peaks** from **SciPy** can be used to detect these peaks.

**Source Code**:

import numpy as np

import scipy.signal as signal

import matplotlib.pyplot as plt

# Function to apply a bandpass filter to the PPG signal

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)

# Function to detect peaks in the signal

def detect\_peaks(signal\_data):

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

# Function to extract heart rate (HR) from detected peaks

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

if len(peaks) < 2:

return 0

rr\_intervals = np.diff(peaks) / fs # RR intervals in seconds

return 60 / np.mean(rr\_intervals) # Heart rate in BPM

# Generate synthetic PPG signal (sine wave + noise)

fs = 100 # Sampling frequency

t = np.linspace(0, 10, fs \* 10) # 10 seconds of data

sine\_signal = np.sin(2 \* np.pi \* 1.2 \* t) # Simulated sine wave signal (1.2 Hz, ~72 BPM)

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

ppg\_signal = sine\_signal + noise\_signal # Combine sine wave and noise to create raw PPG signal

# Process PPG signal

filtered\_signal = bandpass\_filter(ppg\_signal, fs) # Apply bandpass filter to remove noise

normalized\_signal = (filtered\_signal - np.min(filtered\_signal)) / (np.max(filtered\_signal) - np.min(filtered\_signal)) # Normalize signal to [0, 1]

peaks = detect\_peaks(normalized\_signal) # Detect peaks in the signal

heart\_rate = extract\_heart\_rate(peaks, fs) # Estimate heart rate

# 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))

# Plot raw sine signal

plt.subplot(3, 2, 1)

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

plt.xlabel("Time [s]")

plt.ylabel("Amplitude")

plt.legend()

# Plot raw noise signal

plt.subplot(3, 2, 2)

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

plt.xlabel("Time [s]")

plt.ylabel("Amplitude")

plt.legend()

# Plot raw PPG signal (sum of sine wave and noise)

plt.subplot(3, 2, 3)

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

plt.xlabel("Time [s]")

plt.ylabel("Amplitude")

plt.legend()

# Plot filtered PPG signal

plt.subplot(3, 2, 4)

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

plt.xlabel("Time [s]")

plt.ylabel("Amplitude")

plt.legend()

# Plot normalized PPG signal

plt.subplot(3, 2, 5)

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

plt.xlabel("Time [s]")

plt.ylabel("Amplitude")

plt.legend()

# Plot normalized PPG signal with detected peaks

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 [s]")

plt.ylabel("Amplitude")

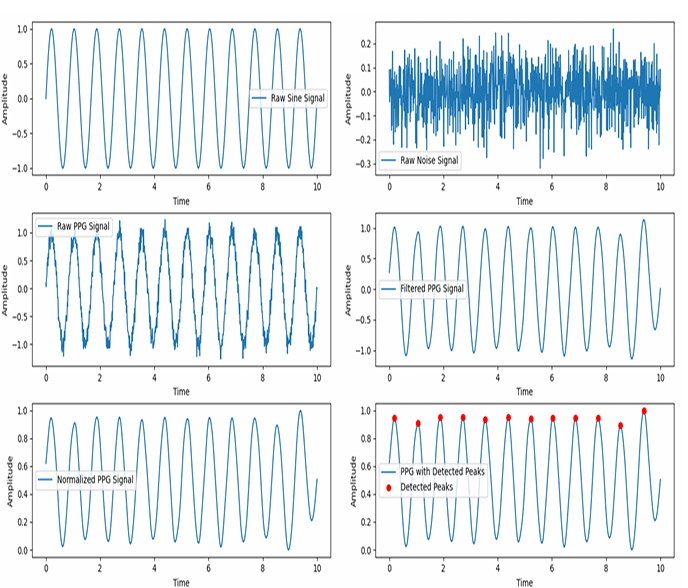
plt.legend()

# Adjust layout for better display

plt.tight\_layout()

plt.show()

**Output:**

****

**Experiment No:06**

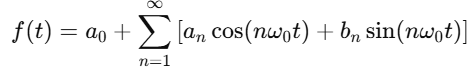
**Experiment Name :**Write a code on Fourier Series decomposition.

**Objective:**

The objective of this code is to decompose a periodic signal into its constituent sinusoidal components using Fourier Series. This helps in understanding the frequency components of a signal, which is widely used in signal processing, communications, and audio analysis.

**Theory:**

A **Fourier Series** allows us to express a periodic function as a sum of sines and cosines. For a given periodic signal f(t), the Fourier Series is defined as:



Where:

* a0​ is the average (DC) component of the signal.
* an ​ and bn​ are the Fourier coefficients for the cosine and sine terms.
* ω0​ is the fundamental angular frequency ω0=2π/T where T is the period of the signal.

**Steps for Fourier Series Decomposition:**

1. **Calculate Fourier Coefficients**: Compute a0​, an​, and bn​ using integration over one period.
2. **Reconstruct the Signal**: Using the Fourier coefficients, reconstruct the signal as a sum of sinusoids.
3. **Plot the Decomposition**: Visualize the original signal and the partial sum of the Fourier Series.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

# Input sequence and N

x = [1, 1, 1, 1] # Example input signal

N = 4 # Length of the DFT

# Padding the input signal to match length N

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

# DFT computation using numpy's FFT function

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

# IDFT computation (Inverse DFT) using numpy's IFFT function

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))

# Plot the input signal

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 the 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 (Reconstructed signal)

plt.subplot(3, 1, 3)

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

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

plt.xlabel('n')

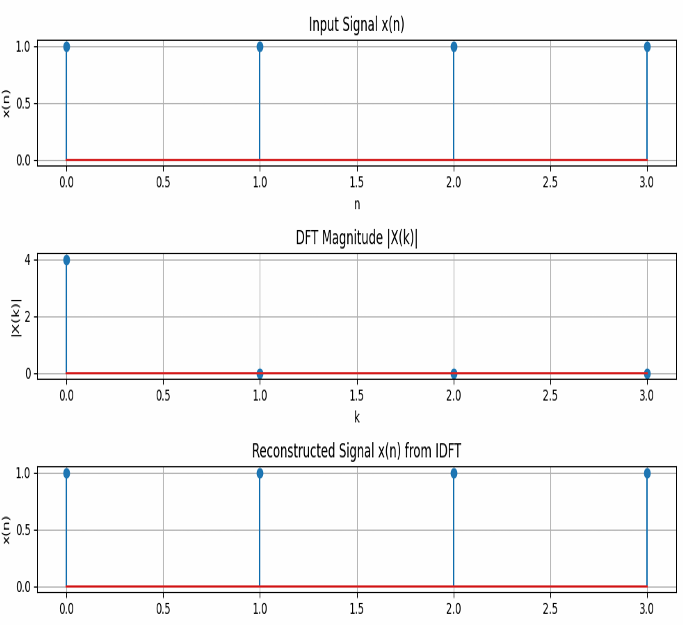
plt.ylabel('x(n)')

plt.grid()

plt.tight\_layout()

plt.show()

**Output:**

****

**Experiment No:07**

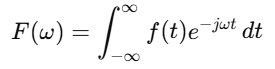
**Experiment Name :**Write a code on Fourier Transform**.**

**Objective:**

The objective of this code is to compute and visualize the Fourier Transform of a given signal. The Fourier Transform allows us to analyze the frequency content of a signal, which is useful in various fields like signal processing, communications, and image processing.

**Theory:**

The Fourier Transform of a continuous-time signal f(t) is given by the formula:



Where:

* F(ω) is the Fourier Transform of the signal f(t).
* ω is the angular frequency (in radians per second).
* e^ −jωt is the complex exponential representing the oscillatory part of the transform.

For Discrete Signals, the Fourier Transform is computed as the Discrete Fourier Transform (DFT), and in practice, the Fast Fourier Transform (FFT) algorithm is often used to compute it efficiently.

The inverse of the Fourier Transform reconstructs the signal from its frequency components.

**Steps in Fourier Transform Computation:**

1. Apply FFT to the Signal: Use the np.fft.fft() function to calculate the Fourier Transform.
2. Visualize the Frequency Spectrum: Plot the magnitude and phase of the Fourier Transform.
3. Reconstruct the Signal: Apply the Inverse Fourier Transform using np.fft.ifft() to recover the original signal from its frequency representation.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

# Fourier Series Approximation function

def fourier\_series(x, terms):

if terms < 1:

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

result = np.zeros\_like(x) # Initialize result with zeros

for n in range(1, terms + 1, 2): # Odd harmonics (1, 3, 5, ...)

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

return result

# Define the original square wave function

def square\_wave(x):

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

# Generate x values (time domain)

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')

# Plot Fourier Series approximations with different number of terms

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

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

plt.axhline(0, color='black', linewidth=0.5, linestyle='--') # Add horizontal axis line

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

plt.xlabel('Time')

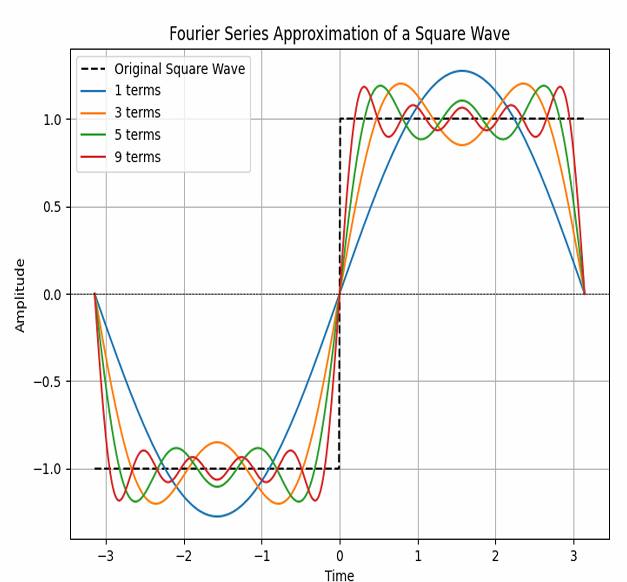
plt.ylabel('Amplitude')

plt.legend()

plt.grid(True)

plt.show()

**Output:**

****

**Experiment No:08**

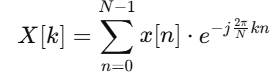
**Experiment Name :**Write a code on DFT.

**Objective:**

The objective of this code is to compute and visualize the Discrete Fourier Transform (DFT) of a given discrete-time signal. The DFT allows us to analyze the frequency components of a signal, which is essential in many areas such as signal processing, audio analysis, and communications.

**Theory:**

The Discrete Fourier Transform (DFT) of a sequence x[n] is defined as:



Where:

* x[n] is the discrete-time signal in the time domain.
* X[k] is the DFT of x[n], representing the frequency-domain components.
* N is the length of the signal.
* k is the frequency index (ranging from 0 to N−1).

The DFT decomposes the signal into sinusoidal components (sine and cosine functions) at different frequencies. The magnitude of X[k] indicates the strength of a particular frequency component, and the phase of X[k] represents the phase shift of that frequency.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

# Define the time vector

t = np.arange(-2, 2.01, 0.01)

# Define the signal (using sinc function)

x = 4 \* np.sinc(4 \* t)

# Plot real part

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

plt.subplot(3, 1, 1)

plt.plot(t, x)

plt.xlabel('Time')

plt.ylabel('Amplitude')

plt.title('Real Part')

plt.grid()

# Plot phase part

plt.subplot(3, 1, 2)

plt.plot(t, np.angle(x))

plt.xlabel('Time')

plt.ylabel('Amplitude')

plt.title('Phase Part')

plt.grid()

# Plot magnitude part

plt.subplot(3, 1, 3)

plt.plot(t, np.abs(x))

plt.ylabel('Amplitude')

plt.title('Magnitude Part')

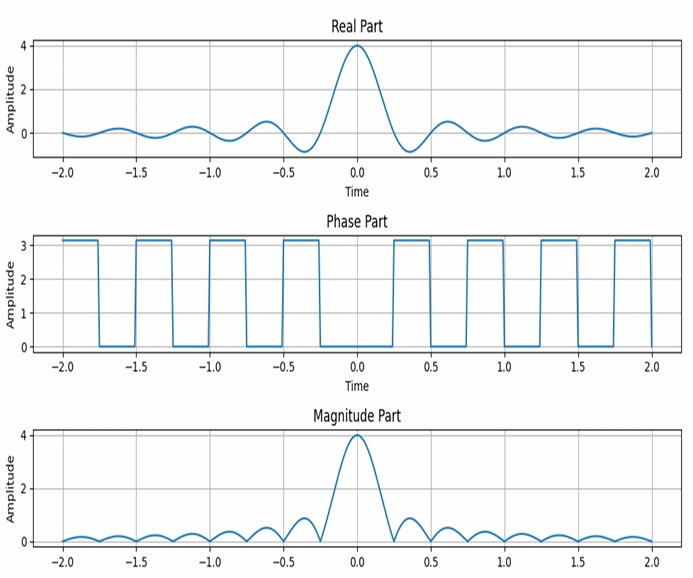
plt.grid()

# Adjust layout and show the plot

plt.tight\_layout()

plt.show()

**Output:**

****