 **LAB REPORT**

Department of

**INFORMATION AND COMMUNICATION ENGINEERING**

**PABNA UNIVERSITY OF SCIENCE AND TECHNOLOGY**

**Submitted To** :

Dr. Md. Imran Hossain

Associate Professor

Dept. of Information and Communication Engineering

**Submitted By** :

Md.Sabbir Hossain

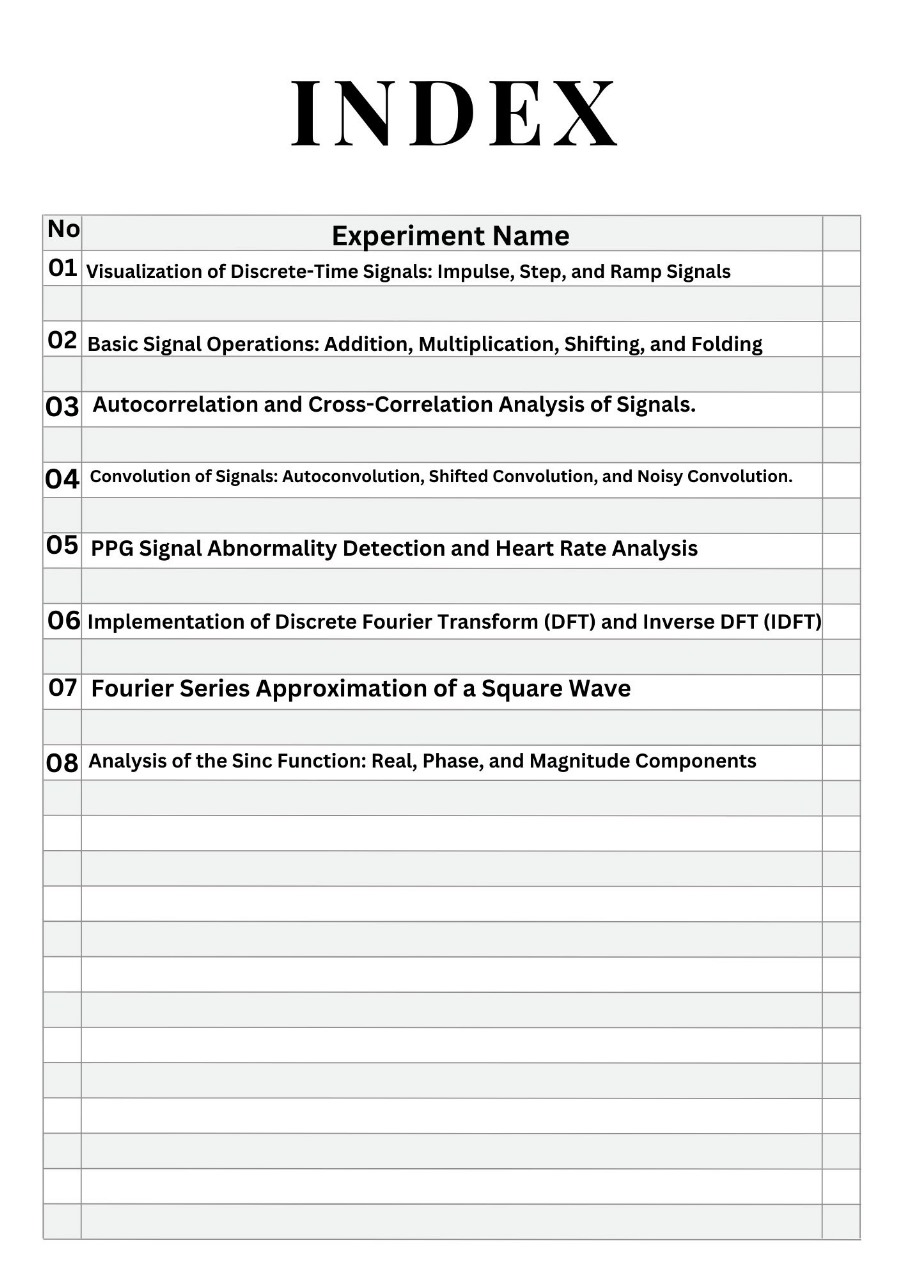
Roll : 220603

Session : 2021-22

Dept. of Information and Communication Engineering

**ICE-2204  
Signals and systems Sessional**

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

****

**Signals And Systems Lab Report**

**Experiment No-1:**

**Tittle:Visualization of Discrete-Time Signals: (Impulse,Step,and Ramp Signals )**

**Theory:**

 **Impulse Signal (δ[n])**: A discrete-time signal that is 1 at n = 0 and 0 elsewhere.

δ[n]={1if n=00if n≠0\delta[n] = \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{if } n \neq 0 \end{cases}δ[n]={10​if n=0if n=0​

 **Step Signal (u[n])**: A signal that is 0 for n < 0 and 1 for n ≥ 0.

u[n]={0if n<01if n≥0u[n] = \begin{cases} 0 & \text{if } n < 0 \\ 1 & \text{if } n \geq 0 \end{cases}u[n]={01​if n<0if n≥0​

 **Ramp Signal (r[n])**: A signal that increases linearly with n for n ≥ 0 and is 0 for n < 0.

r[n]={nif n≥00if n<0r[n] = \begin{cases} n & \text{if } n \geq 0 \\ 0 & \text{if } n < 0 \end{cases}r[n]={n0​if n≥0if n<0​

**Objectives:**

The goal of this experiment is to generate and analyze three fundamental discrete-time signals: Impulse, Step, and Ramp signals, and visualize their behavior.

**Code:**

import numpy as np

import matplotlib.pyplot as plt

# Define the range

n = np.arange(-10, 11)

def impulse\_signal(n):

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

def step\_signal(n):

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

def ramp\_signal(n):

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

# Generate signals

impulse = impulse\_signal(n)

step = step\_signal(n)

ramp = ramp\_signal(n)

# Plot signals

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

plt.subplot(1, 3, 1)

plt.stem(n, impulse)

plt.title("Impulse Signal")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

plt.subplot(1, 3, 2)

plt.stem(n, step)

plt.title("Step Signal")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

plt.subplot(1, 3, 3)

plt.stem(n, ramp)

plt.title("Ramp Signal")

plt.xlabel("n")

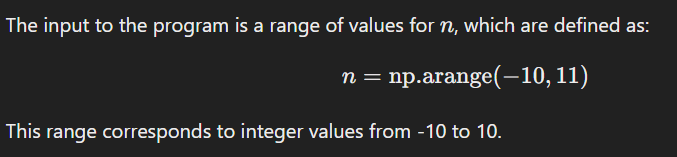
plt.ylabel("Amplitude")

plt.grid()

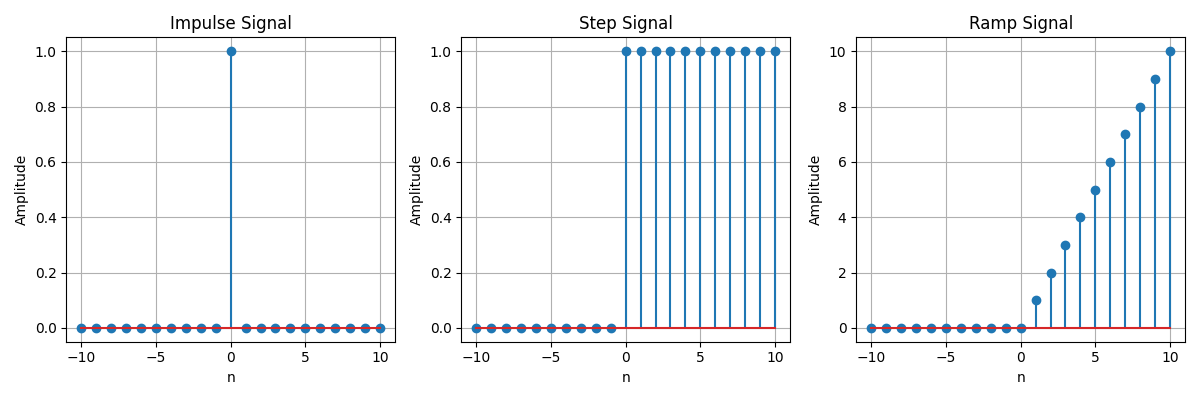
plt.tight\_layout()

plt.show()

Input:



Output :



Purpose:

The purpose of this lab is to:

* Understand the properties and definitions of impulse, step, and ramp signals.
* Use Python to generate these signals over a range of nnn values.
* Visualize the signals using plots to analyze their behaviors.

**Experiment No-02:**

**Tittle: Signal Operations: Addition, Multiplication, Scaling, Shifting, and Folding**

Theory:

Signal processing involves manipulating signals to modify or enhance them. Common operations include:

* **Addition:** Combining two signals element-wise.
* **Multiplication:** Element-wise product of two signals.
* **Scaling:** Multiplying a signal by a constant to change its amplitude.
* **Shifting:** Translating a signal along the time axis.
* **Folding:** Reversing the signal with respect to time.

These operations are essential in fields like communications, audio processing, and control systems.

Objectives:

 **Learn basic signal operations** (addition, multiplication, scaling, shifting, folding).

 **Visualize the effects** of these operations on signals.

 **Understand real-world applications** of these operations in signal processing.

 **Practice using Python** for signal manipulation and visualization.

Code:

import numpy as np

import matplotlib.pyplot as plt

def signal\_addition(x1, x2):

return x1 + x2

def signal\_multiplication(x1, x2):

return x1 \* x2

def signal\_scaling(x, alpha):

return alpha \* x

def signal\_shifting(n, shift):

return n + shift

def signal\_folding(x):

return np.flip(x)

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

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

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

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

multiplied\_signal = signal\_multiplication(x1, x2)

scaled\_signal = signal\_scaling(x1, 2)

shifted\_signal1 = signal\_shifting(n, -2)

shifted\_signal2 = signal\_shifting(n, 2)

folded\_signal = signal\_folding(x1)

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

plt.subplot(4, 2, 1)

plt.stem(n, x1)

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.title("Original Signal x1")

plt.grid()

plt.subplot(4, 2, 2)

plt.stem(n, x2)

plt.xlabel("Time ")

plt.ylabel("Amplitude")

plt.title("Original Signal x2")

plt.grid()

plt.subplot(4, 2, 3)

plt.stem(n, added\_signal)

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.title("Signal Addition")

plt.grid()

plt.subplot(4, 2, 4)

plt.stem(n, multiplied\_signal)

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.title("Signal Multiplication")

plt.grid()

plt.subplot(4, 2, 5)

plt.stem(n, scaled\_signal)

plt.xlabel("Time")

plt.ylabel("Amplitude")

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

plt.grid()

plt.subplot(4, 2, 6)

plt.stem(shifted\_signal1, x1)

plt.xlabel("Time")

plt.ylabel("Amplitude")

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

plt.grid()

plt.subplot(4, 2, 7)

plt.stem(shifted\_signal2, x1)

plt.xlabel("Time")

plt.ylabel("Amplitude")

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

plt.grid()

plt.subplot(4, 2, 8)

plt.stem(n, folded\_signal)

plt.xlabel("Time")

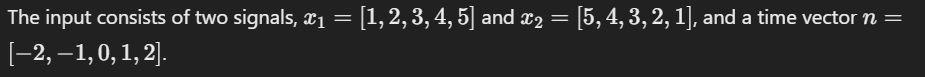
plt.ylabel("Amplitude")

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

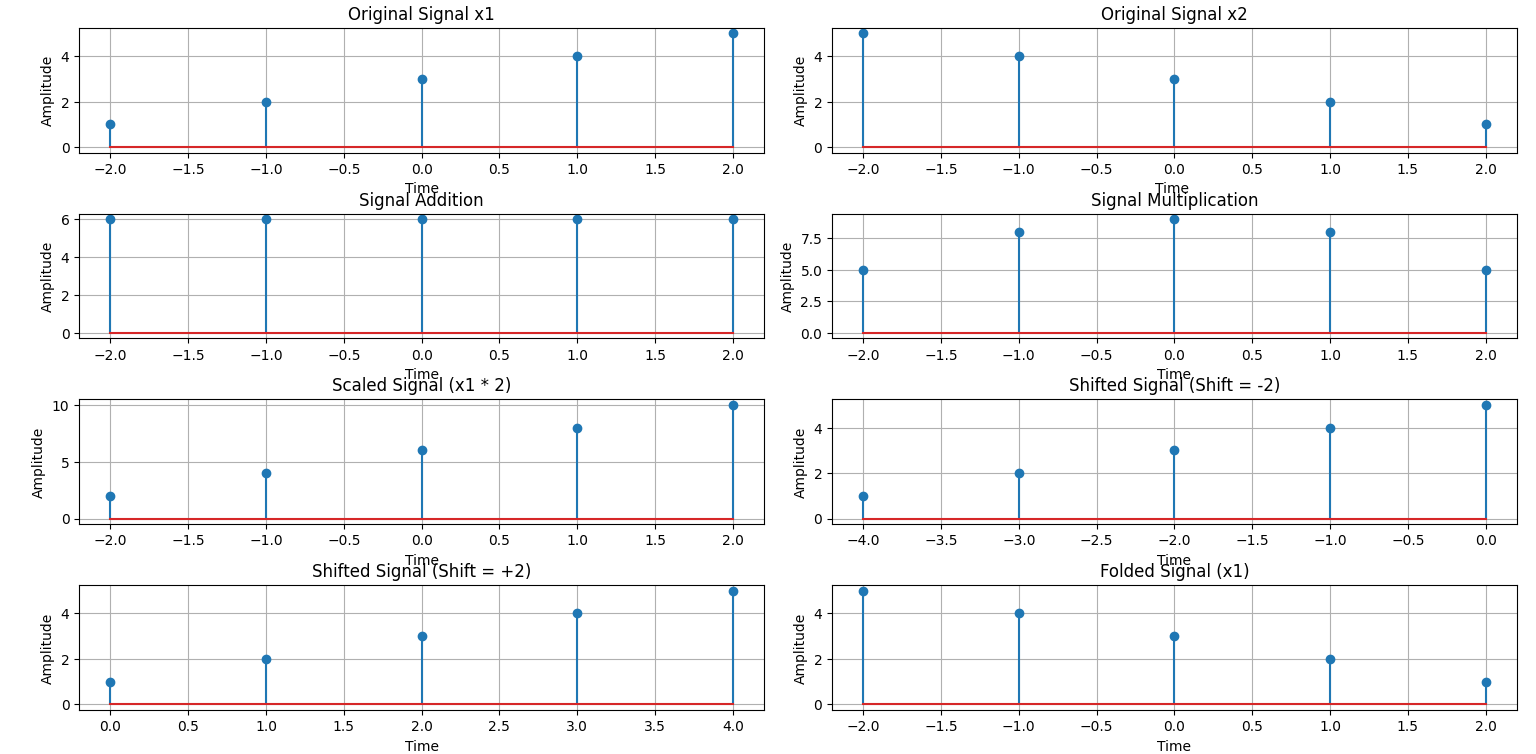
plt.grid()

plt.tight\_layout()

plt.show()

Input: 

Output :



Purpose:

The purpose of the program is to demonstrate common signal operations, including signal addition, multiplication, scaling, shifting, and folding, and visualize their effects on the input signals.

**Experiment No-03:**

**Tittle: Autocorrelation and Cross-Correlation Analysis of Signals with Noise**

Theory:

* **Autocorrelation** measures the similarity of a signal with itself over different time lags, helping to identify periodicity or patterns within the signal.
* **Cross-correlation** measures the similarity between two signals, helping to detect relationships and align signals in time.

These techniques are used in signal detection, noise reduction, and time delay estimation.

Objectives:

1. Understand and compute **autocorrelation** and **cross-correlation**.
2. Analyze signal periodicity and the relationship between signals.
3. Visualize how correlation can help in real-world signal processing tasks.
4. Learn to implement these methods using Python libraries.

Code:

import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import correlate, correlation\_lags

def compute\_autocorrelation(signal):

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

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

return auto\_corr, lags

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

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

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

return cross\_corr, lags

fs = 1000 # Sampling frequency in Hz

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

freq = 5 # Frequency of the sine wave

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

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

signal1 = sin\_signal

signal2 = np.roll(signal1, 100)

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

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

noisy\_signal = signal1 + noise

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

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

plt.subplot(3, 1, 1)

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

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

plt.xlabel("Lag")

plt.ylabel("Autocorrelation")

plt.grid()

plt.subplot(3, 1, 2)

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

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

plt.xlabel("Lag")

plt.ylabel("Cross-Correlation")

plt.grid()

plt.subplot(3, 1, 3)

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

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

plt.xlabel("Lag")

plt.ylabel("Cross-Correlation")

plt.grid()

plt.tight\_layout()

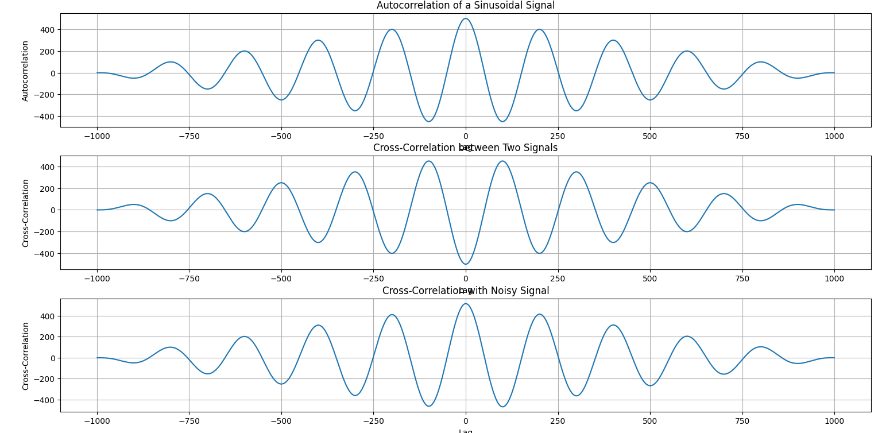
plt.show()

Input:

The input consists of:

* A sinusoidal signal with a frequency of 5 Hz, sampled at 1000 Hz.
* A shifted version of the sinusoidal signal for cross-correlation.
* A noisy version of the sinusoidal signal (created by adding random noise).

Output:



Purpose:

The purpose of this program is to compute and visualize:

1. The **autocorrelation** of a sinusoidal signal.
2. The **cross-correlation** between two signals (one being a shifted version of the other).
3. The **cross-correlation** between a sinusoidal signal and a noisy signal, to observe how noise affects the correlation.

**Experiment No: 04**

**Tittle: Convolution Analysis of Signals: Autoconvolution, Shifted Signals, and Noise Effects**

Theory:

Convolution is a mathematical operation that combines two signals to produce a third, showing their overlap as a function of time shift. Key concepts include:

* **Autoconvolution**: Convolving a signal with itself to detect patterns and periodicity.
* **Convolution with Shifted Signals**: Analyzing the similarity between a signal and its shifted version.
* **Convolution with Noisy Signals**: Studying the effect of noise on a signal through convolution.

Objectives:

 Understand and visualize **autoconvolution** and its properties.

 Analyze **convolution with shifted signals** to measure time shifts and similarity.

 Investigate the effect of **noise** in signal processing through convolution.

 Practice implementing convolution using **Python** and relevant libraries.

Code:

import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import convolve

def compute\_convolution(signal1, signal2):

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

return conv\_result

fs = 1000 # Sampling frequency in Hz

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

freq = 5 # Frequency of the sine wave

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

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

signal1 = sin\_signal

signal2 = np.roll(signal1, 100)

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

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

noisy\_signal = signal1 + noise

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

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

plt.subplot(3, 1, 1)

plt.plot(conv\_auto)

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

plt.xlabel("Samples")

plt.ylabel("Convolution Output")

plt.grid()

plt.subplot(3, 1, 2)

plt.plot(conv\_shifted)

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

plt.xlabel("Samples")

plt.ylabel("Convolution Output")

plt.grid()

plt.subplot(3, 1, 3)

plt.plot(conv\_noisy)

plt.title("Convolution with Noisy Signal")

plt.xlabel("Samples")

plt.ylabel("Convolution Output")

plt.grid()

plt.tight\_layout()

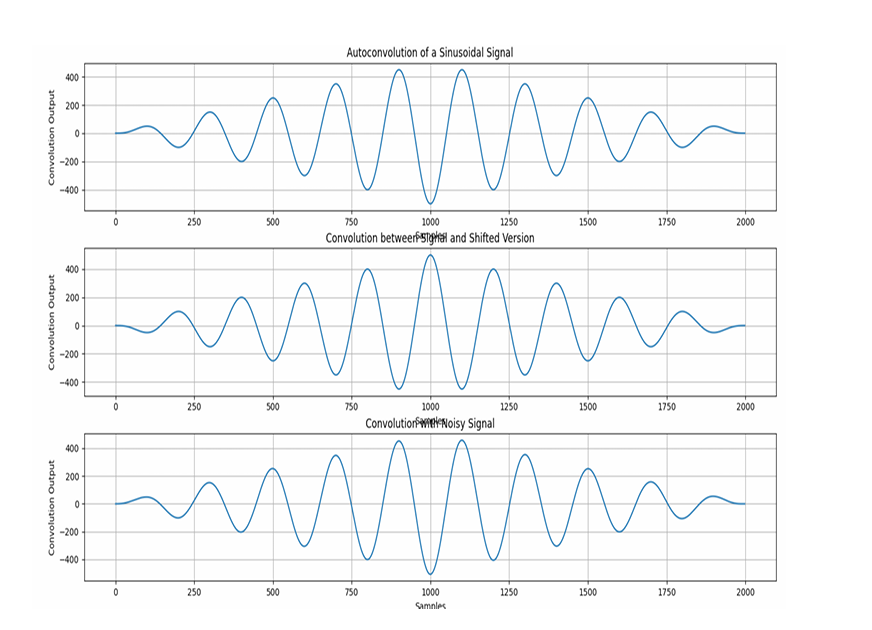
plt.show()

Input:

The input consists of:

* A sinusoidal signal with a frequency of 5 Hz, sampled at 1000 Hz.
* A shifted version of the sinusoidal signal for convolution.
* A noisy version of the sinusoidal signal (created by adding random noise)

Output :



Purpose:

The purpose of this program is to compute and visualize:

1. The **autoconvolution** of a sinusoidal signal.
2. The **convolution** between a sinusoidal signal and its shifted version.
3. The **convolution** between a sinusoidal signal and a noisy signal, to observe the effect of noise on the convolution output.

**Experiment No- 05**

**Tittle - Heart Rate Estimation from PPG Signal Using Bandpass Filtering and Peak Detection**

Theory:

This code processes a PPG signal to estimate heart rate by:

1. **Bandpass Filtering**: Removes noise and retains frequencies relevant to the heart rate (0.5 Hz to 5 Hz).
2. **Peak Detection**: Identifies peaks in the signal corresponding to heartbeats.
3. **Heart Rate Estimation**: Calculates the heart rate by averaging the time intervals between peaks (R-R intervals).
4. **Normalization**: Scales the signal for easier peak detection.

**Objectives:**

 Apply **bandpass filtering** to clean the PPG signal.

 **Detect peaks** in the filtered signal.

 **Estimate heart rate** by analyzing the R-R intervals.

 Visualize the **signal processing stages**.

 Understand how signal processing techniques are used in **health monitoring** for heart rate estimation.

Code:

import numpy as np

import scipy.signal as signal

import matplotlib.pyplot as plt

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

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

return signal.filtfilt(b, a, data)

def detect\_peaks(signal\_data):

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

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

if len(peaks) < 2:

return 0

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

return 60 / np.mean(rr\_intervals)

# Generate synthetic PPG signal

fs = 100

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

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

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

ppg\_signal = sine\_signal + noise\_signal

# Process PPG signal

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

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

np.min(filtered\_signal))

peaks = detect\_peaks(normalized\_signal)

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

# Print results

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

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

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

# Plot results

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

plt.subplot(3, 2, 1)

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

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.legend()

plt.subplot(3, 2, 2)

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

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.legend()

plt.subplot(3, 2, 3)

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

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.legend()

plt.subplot(3, 2, 4)

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

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.legend()

plt.subplot(3, 2, 5)

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

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.legend()

plt.subplot(3, 2, 6)

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

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

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.legend()

plt.tight\_layout()

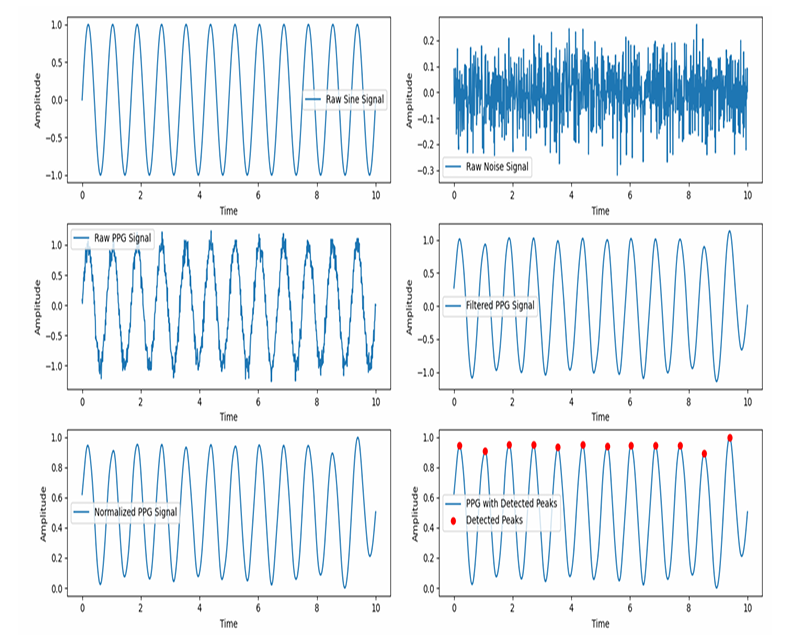
plt.show()

Input:

The input consists of:

* A synthetic PPG signal, created by adding a sine wave of 1.2 Hz frequency and Gaussian noise.
* A sampling frequency fs=100fs = 100fs=100 Hz.

**Output:**



Purpose: The purpose of this program is to:

1. **Filter** the PPG signal using a bandpass filter.
2. **Normalize** the filtered signal for peak detection.
3. **Detect peaks** in the PPG signal.
4. **Estimate heart rate** based on the detected peaks by calculating the RR intervals and computing the average heart rate in beats per minute (BPM).

**Experiment No: 06**

**Tittle: Discrete Fourier Transform (DFT) and Inverse DFT (IDFT) of a Signal**

Theory:

* **DFT (Discrete Fourier Transform)** converts a time-domain signal into its frequency components, revealing the signal's frequency content.
* **IDFT (Inverse Discrete Fourier Transform)** reconstructs the original time-domain signal from its frequency components.

These transforms are essential in signal processing for tasks like spectrum analysis and signal reconstruction.

Objectives:

 Compute the **DFT** of a signal and analyze its frequency components.

 **Reconstruct** the signal using the **IDFT**.

 Visualize the original signal, its frequency spectrum, and the reconstructed signal.

Code:

mport numpy as np

import matplotlib.pyplot as plt

# Input sequence and N

x = [1,1,1,1]

N= 4

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

# DFT computation

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

# IDFT computation (Inverse DFT)

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

# Print the DFT and IDFT values

print("DFT values:", X)

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

# Plot the input signal

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

plt.subplot(3, 1, 1)

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

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

plt.xlabel('n')

plt.ylabel('x(n)')

plt.grid()

# Plot the magnitude of DFT

plt.subplot(3, 1, 2)

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

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

plt.xlabel('k')

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

plt.grid()

# Plot the IDFT signal

plt.subplot(3, 1, 3)

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

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

plt.xlabel('n')

plt.ylabel('x(n)')

plt.grid()

plt.tight\_layout()

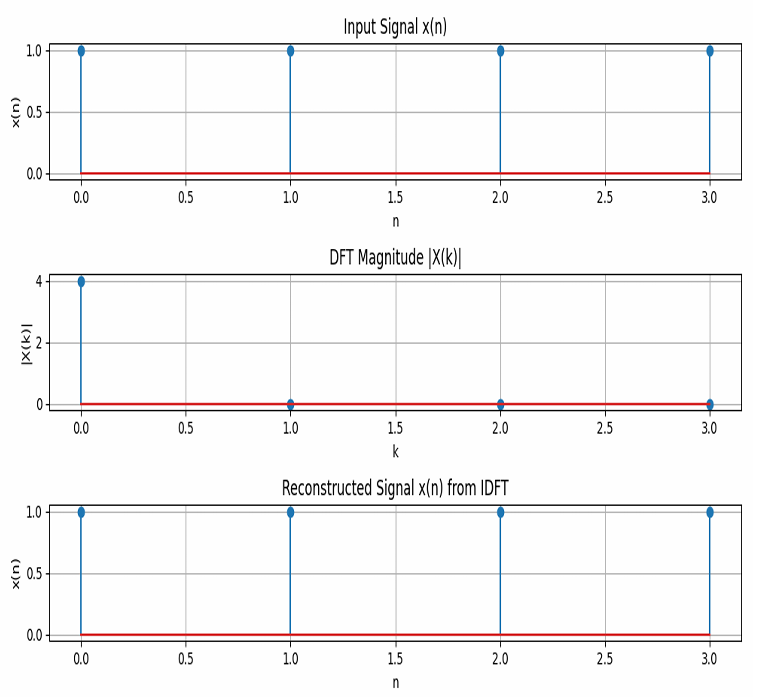
plt.show()

Input:

The input consists of:

* A sequence x=[1,1,1,1]
* The length N=4 for the DFT computation.

**Output:**



Purpose:

The purpose of this program is to:

1. **Compute the Discrete Fourier Transform (DFT)** of the input signal.
2. **Reconstruct the original signal** using the Inverse Discrete Fourier Transform (IDFT).
3. **Visualize** the input signal, the magnitude of the DFT, and the reconstructed signal.

**Experiment No: 07**

**Tittle : Fourier Series Approximation of a Square Wave**

Theory:

The **Fourier Series** is a mathematical tool used to express a periodic function as a sum of sine and cosine functions. It helps to approximate complex periodic signals using simpler sinusoidal components.

1. **Square Wave:**
   * A square wave is a periodic waveform that alternates between two levels, typically +1 and -1. It is a non-sinusoidal waveform, often used in digital signals and communications.
2. **Fourier Series Approximation:**
   * The Fourier series allows us to approximate any periodic function (such as a square wave) as a sum of sinusoidal functions (sines and cosines). For a square wave, the Fourier series consists only of odd harmonics (sine terms with odd multiples of the fundamental frequency).
   * As the number of terms in the Fourier series increases, the approximation becomes closer to the original square wave, exhibiting sharp transitions between the levels.

Objectives:

 **Understand Fourier Series:**

* Learn how Fourier series decompose a periodic function into a sum of sinusoidal components.

 **Approximate Square Wave:**

* Use Fourier series to approximate a square wave using a specified number of terms (odd harmonics).

 **Visualize Approximation:**

* Plot the original square wave and its Fourier series approximations with different numbers of terms to observe how the approximation improves as more terms are added.

 **Explore the Convergence of Fourier Series:**

* Study how increasing the number of terms in the Fourier series leads to a better approximation of the square wave, especially in terms of the sharp transitions.

Code:

import numpy as np

import matplotlib.pyplot as plt

def fourier\_series(x, terms):

if terms < 1:

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

result = x - x

for n in range(1, terms + 1, 2):

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

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')

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='--')

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

plt.xlabel('Time')

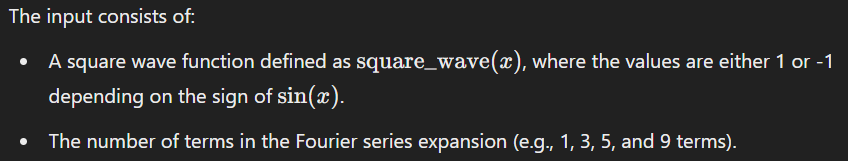
plt.ylabel('Amplitude')

plt.legend()

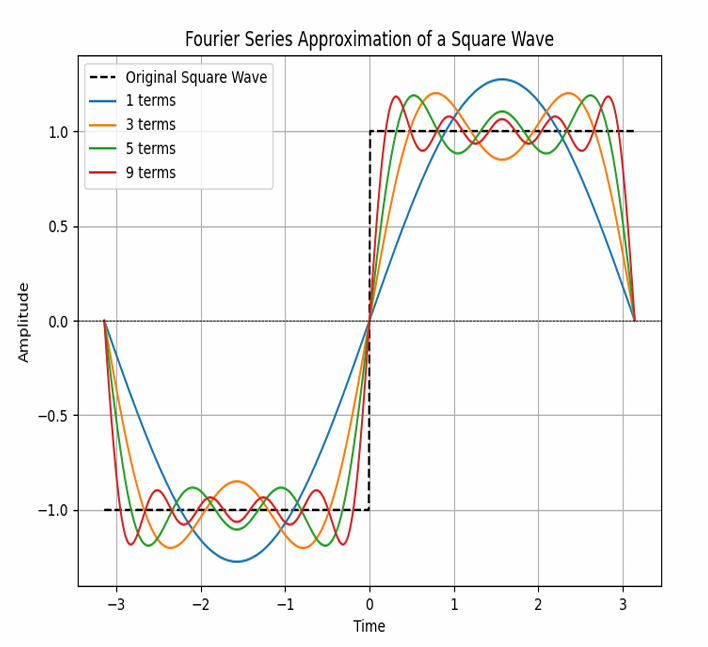
plt.grid()

plt.show()

**Input:**



Output:



Purpose:

The purpose of this program is to:

1. **Approximate** a square wave using a Fourier series with different numbers of terms.
2. **Visualize** the approximation of the square wave for different numbers of terms in the Fourier series and compare it to the original square wave.

**Experiment No: 08**

**Tittle: visualizing the real, phase, and magnitude components of the sinc function.**

Theory:

e **sinc function** is defined as:

sinc(x)=sin⁡(πx)πx\text{sinc}(x) = \frac{\sin(\pi x)}{\pi x}sinc(x)=πxsin(πx)​

However, in this case, the function is scaled as sinc(4t)\text{sinc}(4t)sinc(4t), which is a normalized version of the sinc function.

1. **Real Part**: The real part of a complex signal is the actual value of the signal at any given time.
2. **Phase Part**: The phase represents the shift of the signal in time. It can be obtained using the **angle** of the signal, angle(x)\text{angle}(x)angle(x), which gives the phase of the complex values.
3. **Magnitude Part**: The magnitude represents the absolute value of the signal at any point in time. It provides information about the amplitude of the signal.

Objectives:

 **Plot the Real Part** of the sinc function to show its time-domain behavior.

 **Plot the Phase Part** to visualize how the phase varies across time (though, for real signals, this may remain constant or be zero).

 **Plot the Magnitude Part** to observe the amplitude of the sinc function and its decay with time.

 Understand the **sinc function's properties** and how it relates to signal processing and frequency response.

Code:

import numpy as np

import matplotlib.pyplot as plt

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

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')

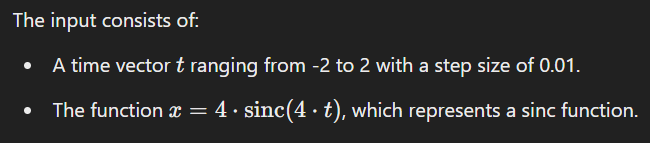
plt.title('Magnitude Part')

plt.grid()

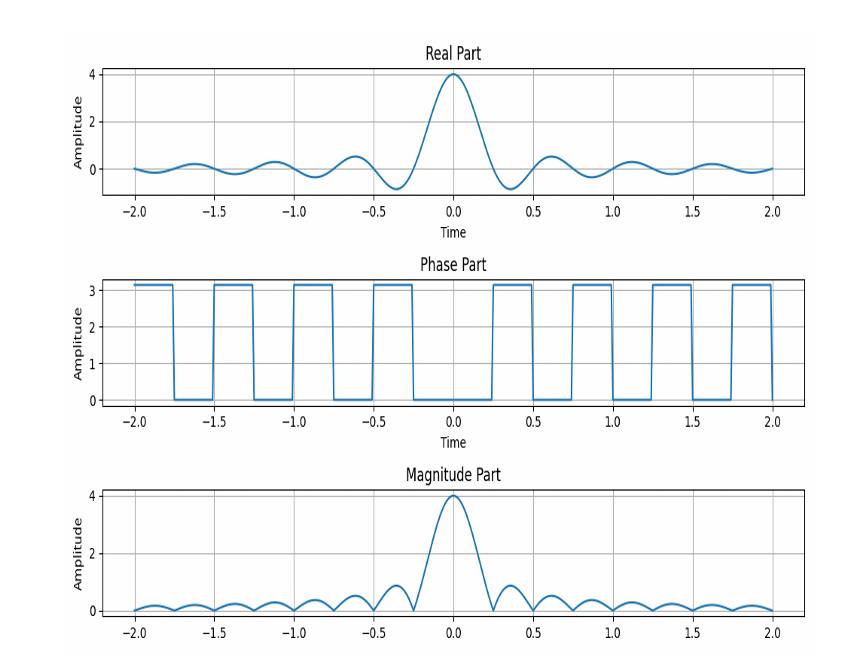
plt.tight\_layout()

plt.show()

Input:



Output:



Purpose:

The purpose of this program is to:

1. **Visualize** the real part, phase part, and magnitude part of the sinc function xxx.
2. **Plot** these components to understand their characteristics over the given time range.