

**Index**

| No | Experiment Name |
| --- | --- |
| 01 | System Sequence: Analysis of Impulse Signal, Step Signal, and Ramp Signal |
| 02 | Signal Operations: Addition, Multiplication, Scaling, Shifting, and Folding |
| 03 | Correlation: Analysis of Autocorrelation and Cross-Correlation of Signals |
| 04 | Convolution: Analysis of Autoconvolution and Convolution of Signals |
| 05 | PPG Signal Processing: Heart Rate and Abnormality Detection. |
| 06 | Fourier Series: Approximation of a Square Wave |
| 07 | Fourier Series Decomposition: Analysis of a Square Wave |
| 08 | Discrete Fourier Transform (DFT) and Inverse Discrete Fourier Transform (IDFT) |

## Experiment No: 01 Title:

System Sequence: Analysis of Impulse Signal, Step Signal, and Ramp Signal

**Theory:**

### 1. Impulse Signal:

An impulse signal, also known as a Dirac delta function, is a signal that is zero everywhere except at the origin, where it is infinitely high. However, its integral over time is 1. It is mathematically represented as:

In discrete-time systems, the impulse signal is defined as:  
   
  
**2. Step Signal:**

A step signal, also known as the unit step function, is a signal that is zero for all negative time and one for all positive time. It is mathematically represented as:

In discrete-time systems, the step signal is defined as:

### 3. Ramp Signal:

A ramp signal is a signal that increases linearly with time. It is mathematically represented as:

In discrete-time systems, the ramp signal is defined as:

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

# Define the range

n = np.arange(-10, 11)

# Define functions for signals

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))

# Impulse Signal

plt.subplot(1, 3, 1)

plt.stem(n, impulse)

plt.title("Impulse Signal")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

# Step Signal

plt.subplot(1, 3, 2)

plt.stem(n, step)

plt.title("Step Signal")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

# Ramp Signal

plt.subplot(1, 3, 3)

plt.stem(n, ramp)

plt.title("Ramp Signal")

plt.xlabel("n")

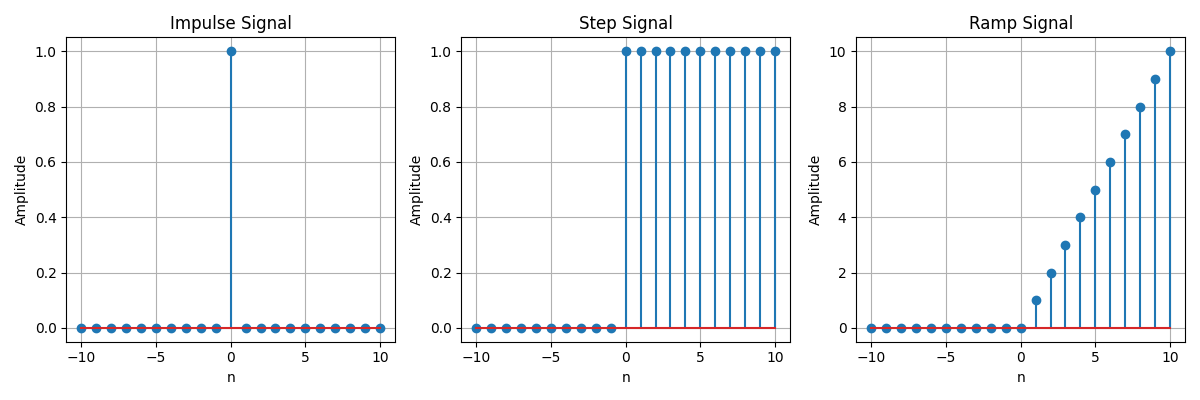
plt.ylabel("Amplitude")

plt.grid()

plt.tight\_layout()

plt.show()

Input:  
 n = np.arange(-5, 10)

**Output:  
  
  
Purpose:**

The purpose of this assignment is to:

1. Understand and analyze three fundamental discrete-time signals: Impulse Signal, Step Signal, and Ramp Signal.
2. Implement these signals programmatically using Python and visualize them using Matplotlib.
3. Gain insight into the characteristics and applications of these signals in signal processing and system analysis.
4. Develop skills in generating and plotting discrete-time signals for further analysis and system design.

**Experiment No:02  
Title:**

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

## Theory:

### 1. Signal Addition:

Signal addition involves adding two signals pointwise. If [*n*] and [n] are two signals, their sum *y*[*n*] is given by:

y[n] =[n] + [n] **2. Signal Multiplication:**

Signal multiplication involves multiplying two signals pointwise. If [n] and [n] are two signals, their product *y*[*n*] is given by:

y[n] =[n] [n]

### 3. Signal Scaling:

Signal scaling involves multiplying a signal by a constant factor. If *x*[*n*] is a signal and *α* is a scaling factor, the scaled signal *y*[*n*] is given by:

**4. Signal Shifting:**

Signal shifting involves shifting a signal in time. If is a signal and is the shift amount, the shifted signal is given by: **=**

**5. Signal Folding:**

Signal folding involves reflecting a signal about the origin. If x[n] is a signal, the folded signal *y*[*n*] is given by:

**=   
  
Source Code:**import numpy as np

import matplotlib.pyplot as plt

# Define signal operations

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)

# Define input signals and time axis

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

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

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

# Perform signal operations

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)

# Plot the results

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()

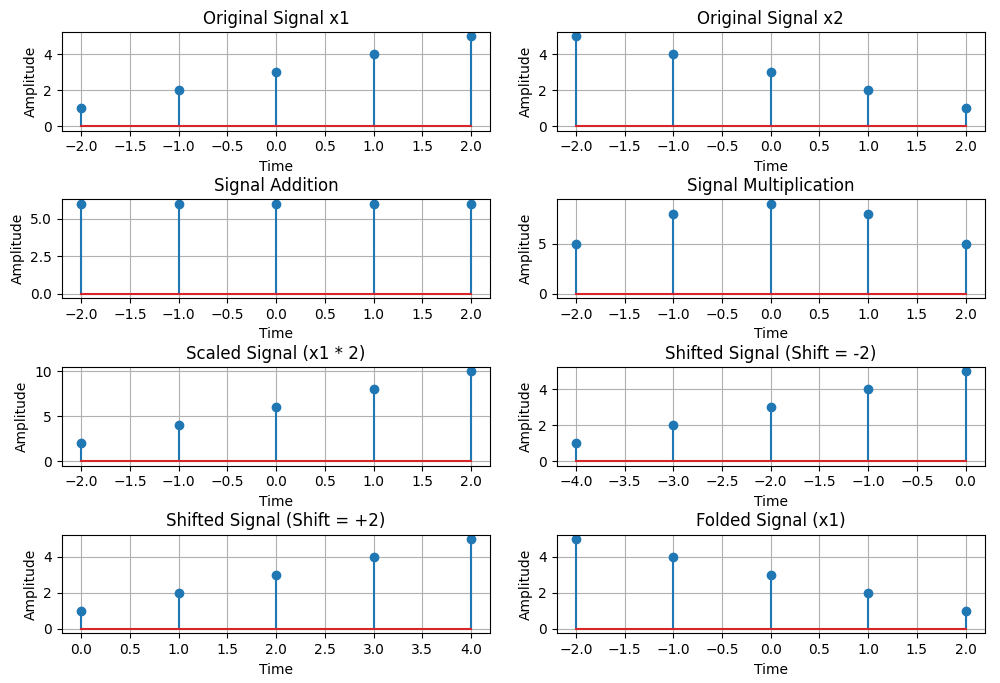
## Input:

1. **Input Signals:**
   * =[1,2,3,4,

=[5,4,3,2,1]

1. **Time Axis:**

*n*=[−2,−1,0,1,2]

**Output:  
  
  
Purpose:**

The purpose of this assignment is to:

1. Understand and implement basic signal operations such as addition, multiplication, scaling, shifting, and folding.
2. Visualize the effects of these operations on discrete-time signals.
3. Gain hands-on experience in manipulating signals using Python and Matplotlib.

**Experiment No:03.  
Title:**

Correlation: Analysis of Autocorrelation and Cross-Correlation of Signals

## Theory:

### Correlation:

Correlation is a mathematical operation used to measure the similarity between two signals. It is widely used in signal processing to identify patterns, detect delays, and analyze signal properties. There are two types of correlation:

* Autocorrelation: Measures the similarity of a signal with a delayed version of itself.
* Cross-Correlation: Measures the similarity between two different signals.

### 1. Autocorrelation:

Autocorrelation is the correlation of a signal with itself. It is used to identify repeating patterns or periodicities in a signal. Mathematically, the autocorrelation Rxxof a signal *x*(*t*) is defined as:

In discrete-time systems, it is defined as: **2. Cross-Correlation:**

Cross-correlation measures the similarity between two signals x(t) and *y*(*t*). It is used to determine the time delay between two signals or to detect the presence of a known signal in a noisy environment. Mathematically, the cross-correlation *Rxy*  is defined as:

In discrete-time systems, it 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

# Define parameters

fs = 1000

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

freq = 5

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

# Compute autocorrelation

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

# Compute cross-correlation between original and shifted signal

signal1 = sin\_signal

signal2 = np.roll(signal1, 100)

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

# Compute cross-correlation with a noisy signal

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

noisy\_signal = signal1 + noise

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

# Plot results

plt.figure(figsize=(10, 8))

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:**

**Sinusoidal Signal**:

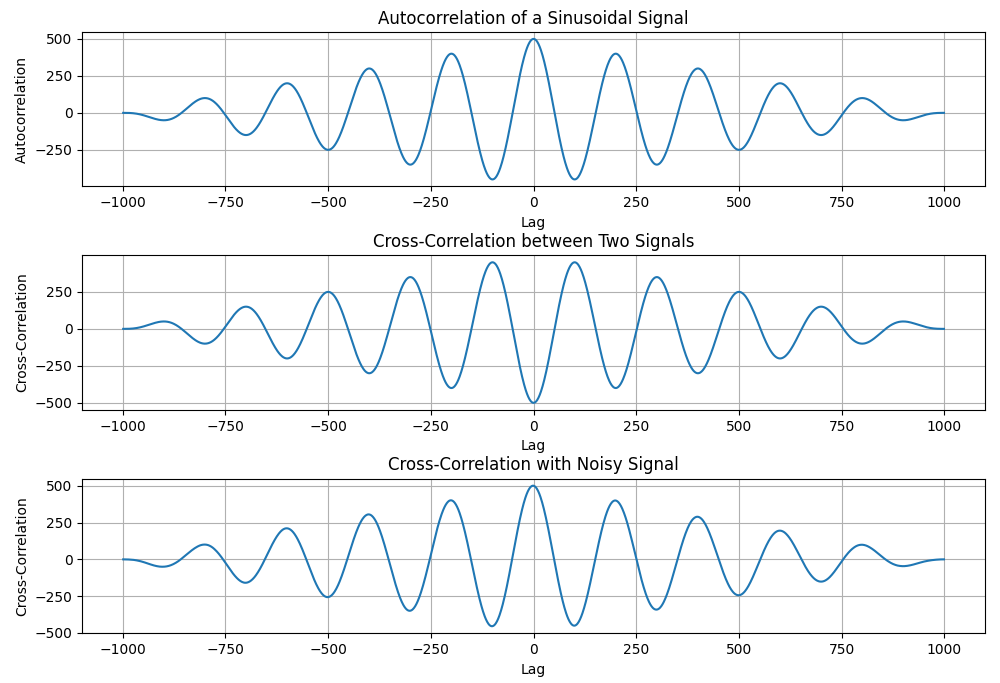
A sinusoidal signal with a frequency of 5 Hz is generated using:  
**sin\_signal = np.sin(2 \* np.pi \* freq \* t)**

**Shifted Signal**:

The original sinusoidal signal is shifted by 100 samples using:  
**signal2 = np.roll(signal1, 100)**

**Noisy Signal**:

Gaussian noise is added to the original signal using:  
**noisy\_signal = signal1 + noise**

**Output:**  
  
**Purpose:**  
The purpose of this assignment is to:

1. Understand the concept of correlation in signal processing, including autocorrelation and cross-correlation.
2. Implement autocorrelation and cross-correlation using Python and visualize the results.
3. Analyze the effects of signal shifting and noise on correlation.
4. Gain practical experience in using correlation to measure similarity between signals and detect time delays.

**Experiment No:04**  
  
**Title:**

Convolution: Analysis of Autoconvolution and Convolution of Signals

## Theory:

### Convolution:

Convolution is a mathematical operation used to combine two signals, producing a third signal that represents how the shape of one signal is modified by the other. It is widely used in signal processing, image processing, and system analysis. Mathematically, the convolution of two signals *x*(*t*) and *h*(*t*) is defined as:

**1. Autoconvolution:**

Autoconvolution is the convolution of a signal with itself. It is used to analyze the properties of a signal, such as its symmetry or periodicity. Mathematically, the autoconvolution of a signal *x*(*t*) is:

**2. Convolution with Shifted and Noisy Signals:**

Convolution can also be performed between a signal and its shifted version or a noisy version. This helps in understanding how delays or noise affect the output of a system.

**Source Code:**import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import convolve

# Function to compute convolution

def compute\_convolution(signal1, signal2):

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

return conv\_result

# Define parameters

fs = 1000

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

freq = 5

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

# Compute autoconvolution

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

# Compute convolution between original and shifted signal

signal1 = sin\_signal

signal2 = np.roll(signal1, 100)

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

# Compute convolution with a noisy signal

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

noisy\_signal = signal1 + noise

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

# Plot results

plt.figure(figsize=(10, 8))

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")

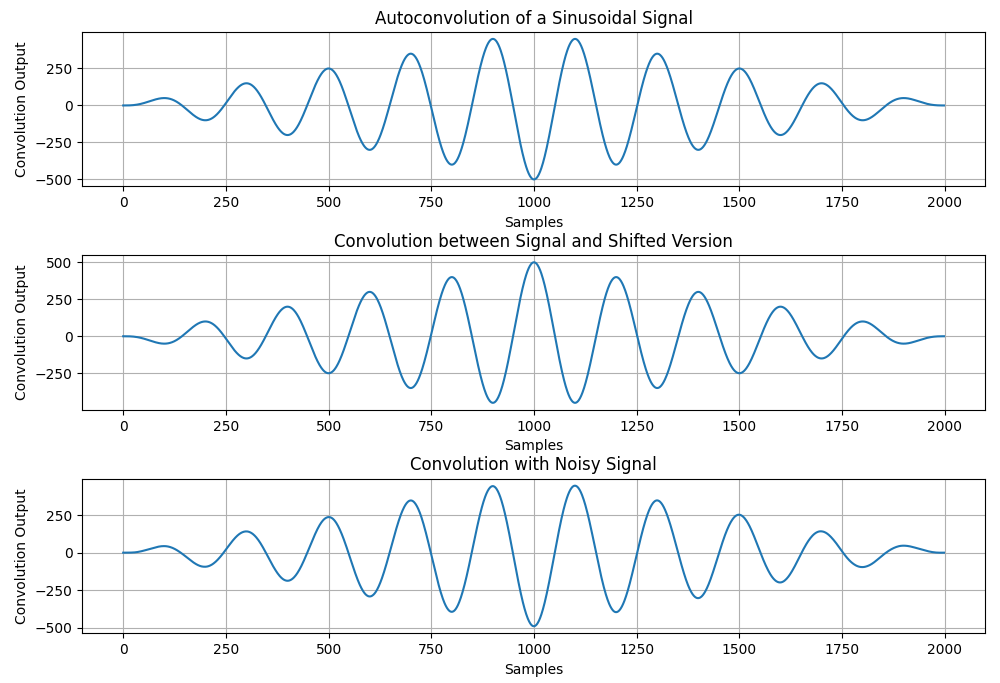
plt.ylabel("Convolution Output")

plt.grid()

plt.tight\_layout()

plt.show()

**Input:**

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

The purpose of this assignment is to:

1. Understand the concept of convolution in signal processing and its applications in analyzing linear time-invariant (LTI) systems.
2. Implement convolution operations using Python and visualize the results.
3. Analyze the effects of autoconvolution, convolution with a shifted signal, and convolution with a noisy signal.
4. Gain practical experience in using convolution to study signal interactions and system responses.
5. Develop skills in signal processing and visualization using Python libraries such as NumPy, SciPy, and Matplotlib.

**Experiment No:05**  
**Title:**

PPG Signal Processing: Heart Rate and Abnormality Detection.

**Theory:**

### 1. Photoplethysmogram (PPG) Signal:

The PPG signal is a non-invasive optical technique used to detect blood volume changes in the microvascular bed of tissue. It is commonly used in wearable devices to measure heart rate and oxygen saturation (SpO2). The PPG signal consists of:

* **Systolic Peak**: Corresponds to the heartbeat.
* **Diastolic Peak**: Corresponds to the relaxation phase of the heart.
* **Dicrotic Notch**: A small downward deflection in the signal.

### 2. Signal Processing Steps:

* **Bandpass Filtering**: Removes noise and artifacts from the raw PPG signal.
* **Normalization**: Scales the signal to a range of [0, 1] for easier analysis.
* **Peak Detection**: Identifies systolic peaks in the PPG signal.
* **Heart Rate Calculation**: Computes the heart rate using the time intervals between peaks.
* **Abnormality Detection**: Identifies abnormal peaks that deviate significantly from the average RR interval.

### 3. Applications:

* Heart rate monitoring.
* Detection of arrhythmias and other cardiovascular abnormalities.
* Wearable health monitoring devices.

**Source Code:**import numpy as np

import scipy.signal as signal

import matplotlib.pyplot as plt

# Bandpass filter function

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)

# Peak detection function

def detect\_peaks(signal\_data):

peaks, \_ = signal.find\_peaks(signal\_data, distance=50, height=0.3)

return peaks

# Heart rate extraction function

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)

# Abnormality detection function (if RR intervals vary drastically)

def detect\_abnormalities(peaks, fs=100):

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

mean\_rr = np.mean(rr\_intervals)

abnormal\_peaks = [i for i, rr in enumerate(rr\_intervals) if abs(rr - mean\_rr) > 0.2 \* mean\_rr]

return abnormal\_peaks

# 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)

abnormalities = detect\_abnormalities(peaks, fs)

# Print results

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

print(f"Detected Abnormalities: {len(abnormalities)}")

# Plot Results

plt.figure(figsize=(15, 10))

plt.subplot(4, 1, 1)

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

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.legend()

plt.grid()

plt.subplot(4, 1, 2)

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

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.legend()

plt.grid()

plt.subplot(4, 1, 3)

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

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.legend()

plt.grid()

plt.subplot(4, 1, 4)

plt.plot(t, normalized\_signal, label='Peak Detection')

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

for ab in abnormalities:

plt.plot(t[peaks[ab]], normalized\_signal[peaks[ab]], 'kx', label='Abnormal Peak')

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.legend()

plt.grid()

plt.tight\_layout()

plt.show()  
  
  
**Input:**

1. **Synthetic PPG Signal**:  
   * A synthetic PPG signal is generated using a sine wave with added noise:

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

**2.Sampling Frequency**:

The sampling frequency is set to 100 Hz:  
 fs = 100

## Output: Estimated Heart Rate: 71.90 BPM

## Detected Abnormalities: 0

## Purpose:

The purpose of this assignment is to:

1. Understand the Photoplethysmogram (PPG) signal and its applications in heart rate monitoring.
2. Implement signal processing techniques such as filtering, normalization, and peak detection on a synthetic PPG signal.
3. Extract heart rate and detect abnormalities in the PPG signal.
4. Gain practical experience in processing and analyzing biomedical signals using Python.
5. Develop skills in signal filtering, peak detection, and visualization using Python libraries such as NumPy, SciPy, and Matplotlib.

**Experiment No:06  
Title:**

Fourier Series: Approximation of a Square Wave

## Theory:

### 1. Fourier Series:

The Fourier series is a mathematical tool used to represent a periodic function as a sum of sine and cosine functions. It is widely used in signal processing, physics, and engineering to analyze and synthesize periodic signals. The Fourier series of a periodic function *f*(*x*) with period 2*π* is given by:

where:

**=   
 =   
  
2. Square Wave:**

A square wave is a periodic waveform that alternates between two levels (e.g., 1 and -1) with a 50% duty cycle. The Fourier series of a square wave consists of odd harmonics of sine functions:

**3. Fourier Series Approximation:**

The Fourier series approximation involves truncating the infinite series to a finite number of terms. As the number of terms increases, the approximation becomes more accurate.

## 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): # Iterate over odd numbers (1, 3, 5,...)

result += (4 / (np.pi \* n)) \* np.sin(n \* x) # Fourier series formula

return result

# Define the Original Square Wave Function

def square\_wave(x):

return np.where(np.sin(x) >= 0, 1, -1) # Generates a square wave

# Generate X Values

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

# Plot the 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 Terms

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

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

# Add Grid and Labels

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(True)

# Show the Plot

plt.show()

## Input: 1.Time Vector: t = np.linspace(-np.pi, np.pi, 400) This creates 400 points between −π and *π*.

## 2.Square Wave Function: def square\_wave(x):

## return np.where(np.sin(x) >= 0, 1, -1) 3.Fourier Series Terms: The Fourier series approximation is computed for different numbers of terms: 1, 3, 5, and 9.

## Output: Purpose:

## The purpose of this assignment is to:

## Understand the concept of Fourier Series and its application in approximating periodic signals, such as a square wave.

## Implement the Fourier Series approximation of a square wave using Python and visualize the results.

## Analyze how increasing the number of terms in the Fourier Series improves the approximation of the square wave.

## Gain practical experience in using Fourier Series to decompose and reconstruct signals.

## Experiment No:07 Title:

## Fourier Series Decomposition: Analysis of a Square Wave

## 

## Theory:

### 1. Fourier Series Decomposition:

## Fourier series decomposition is the process of breaking down a periodic function into a sum of sine and cosine functions. This is achieved by calculating the Fourier coefficients , and which represent the amplitude of the cosine and sine terms in the series. The Fourier series of a periodic function with period T is given by:

**where:**

**=   
  
 =   
 =   
  
2.Square Wave:**

A square wave is a periodic waveform that alternates between two levels (e.g., 1 and -1) with a 50% duty cycle. The Fourier series of a square wave consists of odd harmonics of sine functions:

### 3. Fourier Coefficients:

## The coefficients , and are calculated using numerical integration. These coefficients determine the contribution of each harmonic to the overall signal.

## Source Code: import numpy as np

## from scipy.integrate import quad

## 

## # Define the function to decompose (Example: Square Wave)

## def f(x):

## return np.where(np.sin(x) >= 0, 1, -1) # Square wave function

## 

## # Fourier Series Coefficients Calculation

## def fourier\_coefficients(f, T, N):

## L = T / 2 # Half period

## a0 = (1 / L) \* quad(lambda x: f(x), -L, L)[0] # Compute a0

## 

## an = np.zeros(N)

## bn = np.zeros(N)

## for n in range(1, N + 1):

## an[n-1] = (1 / L) \* quad(lambda x: f(x) \* np.cos(n \* np.pi \* x / L), -L, L)[0]

## bn[n-1] = (1 / L) \* quad(lambda x: f(x) \* np.sin(n \* np.pi \* x / L), -L, L)[0]

## 

## return a0, an, bn

## 

## # Parameters

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

## N = 10 # Number of terms

## 

## # Compute Fourier Coefficients

## a0, an, bn = fourier\_coefficients(f, T, N)

## 

## # Display Fourier Coefficients

## np.set\_printoptions(precision=5, suppress=True) # Format output for readability

## print("a0:", a0)

## print("an:", an)

## print("bn:", bn) Input:

## 1.Square Wave Function def f(x):

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

## 2.Period T: T = 2 \* np.pi 3.Number of Terms N: N = 10 Output: : 0.0

## : [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

## : [ 1.27324 0. 0.42441 0. 0.25465 0. 0.18189 -0.

## 0.14147 -0. ] Explanation of Output:

**:**The DC component is 0, which is expected for a square wave.

## :All cosine coefficients are 0, as the square wave is an odd function.

## :The sine coefficients are non-zero for odd harmonics, consistent with the Fourier series of a square wave.

## Purpose:

## The purpose of this assignment is to:

## 1.Understand the concept of Fourier series decomposition and its application in analyzing periodic signals.

## 2.Compute the Fourier coefficients ,,for a square wave.

## 3.Verify the theoretical properties of Fourier series, such as the absence of cosine terms (

## ) for odd functions like the square wave.

## 4.Gain hands-on experience in using numerical integration to calculate Fourier coefficients. Experiment No: 08 Title:

## Discrete Fourier Transform (DFT) and Inverse Discrete Fourier Transform (IDFT)

## Theory:

### 1. Discrete Fourier Transform (DFT):

## The Discrete Fourier Transform (DFT) is a mathematical technique used to transform a discrete-time signal from the time domain to the frequency domain. For a sequence *x*[*n*] of length *N*, the DFT *X*[*k*] is defined as:

## k=0,1,,,,,N-1 where:

## *X*[*k*] is the DFT of the sequence *x*[*n*].

## *k* is the frequency bin index.

## *N* is the length of the sequence.

### 2. Inverse Discrete Fourier Transform (IDFT):

## The Inverse Discrete Fourier Transform (IDFT) is used to reconstruct the original time-domain signal from its frequency-domain representation. The IDFT is defined as:

## n=0,1,,,,,N-1 where:

## *x*[*n*] is the reconstructed time-domain signal.

## *X*[*k*] is the DFT of the sequence.

### 3. Frequency Bins:

## The frequency bins represent the discrete frequencies at which the DFT is computed. For a sequence of length *N*, the frequency bins are given by:

k = 0,1,,,,N-1 **Source Code:**  
import numpy as np

import matplotlib.pyplot as plt

# Input sequence and N

x = [1,1,1,1]

N= 4

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

# DFT computation

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

# IDFT computation (Inverse DFT)

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

# Compute frequency bins

freq\_bins = np.fft.fftfreq(N)

# Print the DFT and IDFT values

print("DFT values:", X)

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

print("Frequency bins:", freq\_bins)

# Plot the input signal

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

plt.subplot(4, 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(4, 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 phase of DFT

plt.subplot(4, 1, 3)

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

plt.title('DFT Phase ∠X(k)')

plt.xlabel('k')

plt.ylabel('Phase (radians)')

plt.grid()

# Plot the IDFT signal

plt.subplot(4, 1, 4)

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()

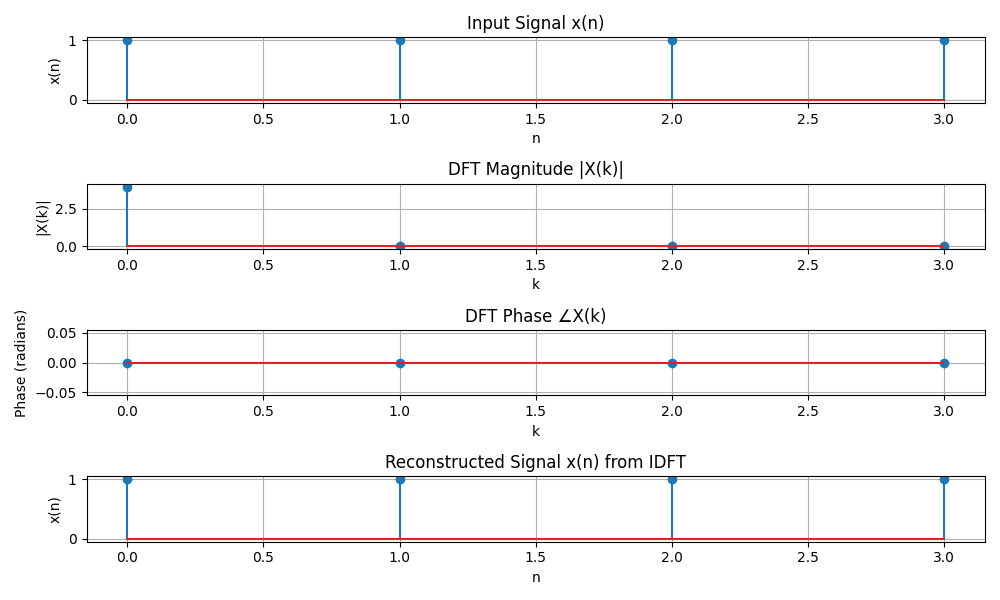
**Input:**

1.Input Sequence:

x = [1, 1, 1, 1]  
2.Length of DFT (*N*):  
 N = 4 **Output:**DFT values: [4.+0.j 0.+0.j 0.+0.j 0.+0.j]

Reconstructed IDFT values: [1. 1. 1. 1.]

Frequency bins: [ 0. 0.25 -0.5 -0.25]

**  
  
Purpose:**

The purpose of this assignment is to:

1. Understand the concept of Discrete Fourier Transform (DFT) and Inverse Discrete Fourier Transform (IDFT).
2. Compute the DFT and IDFT of a given sequence using Python.
3. Analyze the magnitude and phase spectrum of the DFT.
4. Reconstruct the original signal from its DFT using IDFT.
5. Visualize the input signal, DFT magnitude, DFT phase, and reconstructed signal.