

***.INDEX.***

|  |  |
| --- | --- |
| **S.L.** | **Name of The Problem** |
| **01** | Plot the following signal operations using user defined function - **a. adding ,b. multiplication, c. Scaling, d. shifting and e. folding.** |
| **02** | **Plot the following transformation x1(n) = 2x(n-5) - 3x(n+4).** |
| **03** | **Explain and Implementation the unit Impulse sequence, the unit Step sequence, the unit Ramp sequence.** |
| **04** | **Explain and Implement convolution of signal.** |
| **05** | **Explain and Implement correlation of signal.** |
| **06** | **Extract relevant features such as filtering, feature extraction, pick detection, heart rate etc. from PPG signal.** |
| **07** | **Explain and Implement Discrete Fourier Transform (DFT) using python.** |
| **08** | **Explain and Implement Frequency bin using python.** |

# Problem No: 01

## **Problem Name: Plot following signal operations using user defined function – a. adding, b. multiplication, c. Scaling, d. shifting and e. folding.**

### **Objective:**

To implement and visualize basic signal operations (addition, multiplication, scaling, shifting, and folding) using user-defined functions in Python.

### **Theory:**

Signal processing involves various operations to manipulate signals for analysis and transformation. The fundamental operations are:

1. **Addition of Two Signals:**  
   When two discrete-time signals x1[n] and x2[n] are added, the result is given by:

y[n]= x1​[n]+x2​[n]

This operation combines the amplitudes of both signals at corresponding time indices.

1. **Multiplication of Two Signals:**  
   Element-wise multiplication of two signals is given by:

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

This operation is used in modulation and filtering.

1. **Scaling of a Signal:**  
   A signal x[n] is scaled by a factor α as follows:

y[n]=α⋅x[n]

This amplifies or attenuates the signal’s amplitude.

1. **Shifting of a Signal:**  
   A signal is delayed or advanced by shifting its index by k:
   * **Right shift (delay)**:

y[n]=x[n−k]

* + **Left shift (advance)**:

y[n]=x[n+k]

1. **Folding (Time Reversal) of a Signal:**  
   Folding reflects the signal around the vertical axis:

y[n]=x[−n]

This is useful in symmetry analysis and convolution.

### **Source Code:**

import numpy as np

import matplotlib.pyplot as plt

def signal\_add(x1,x2):

    return x1+x2

def signal\_mul(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)

t = np.arange(-10,10,0.01)

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

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

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

p1 = np.sin(2 \* np.pi \* 1 \* t)

p2 = np.cos(2 \* np.pi \* 0.5 \* t)

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

multiplied\_signal = signal\_mul(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=(16,20))

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('multiplied\_signal')

plt.grid()

plt.subplot(4,2,5)

plt.stem(n,scaled\_signal)

plt.xlabel("time")

plt.ylabel('amplitude')

plt.title('Original signal (x1\*2)')

plt.grid()

plt.subplot(4,2,6)

plt.stem(shifted\_signal1,x1)

plt.xlabel("time")

plt.ylabel('amplitude')

plt.title('2 shifted signal')

plt.grid()

plt.subplot(4,2,7)

plt.stem(shifted\_signal2,x1)

plt.xlabel("time")

plt.ylabel('amplitude')

plt.title('-2 shifted signal')

plt.grid()

plt.subplot(4,2,8)

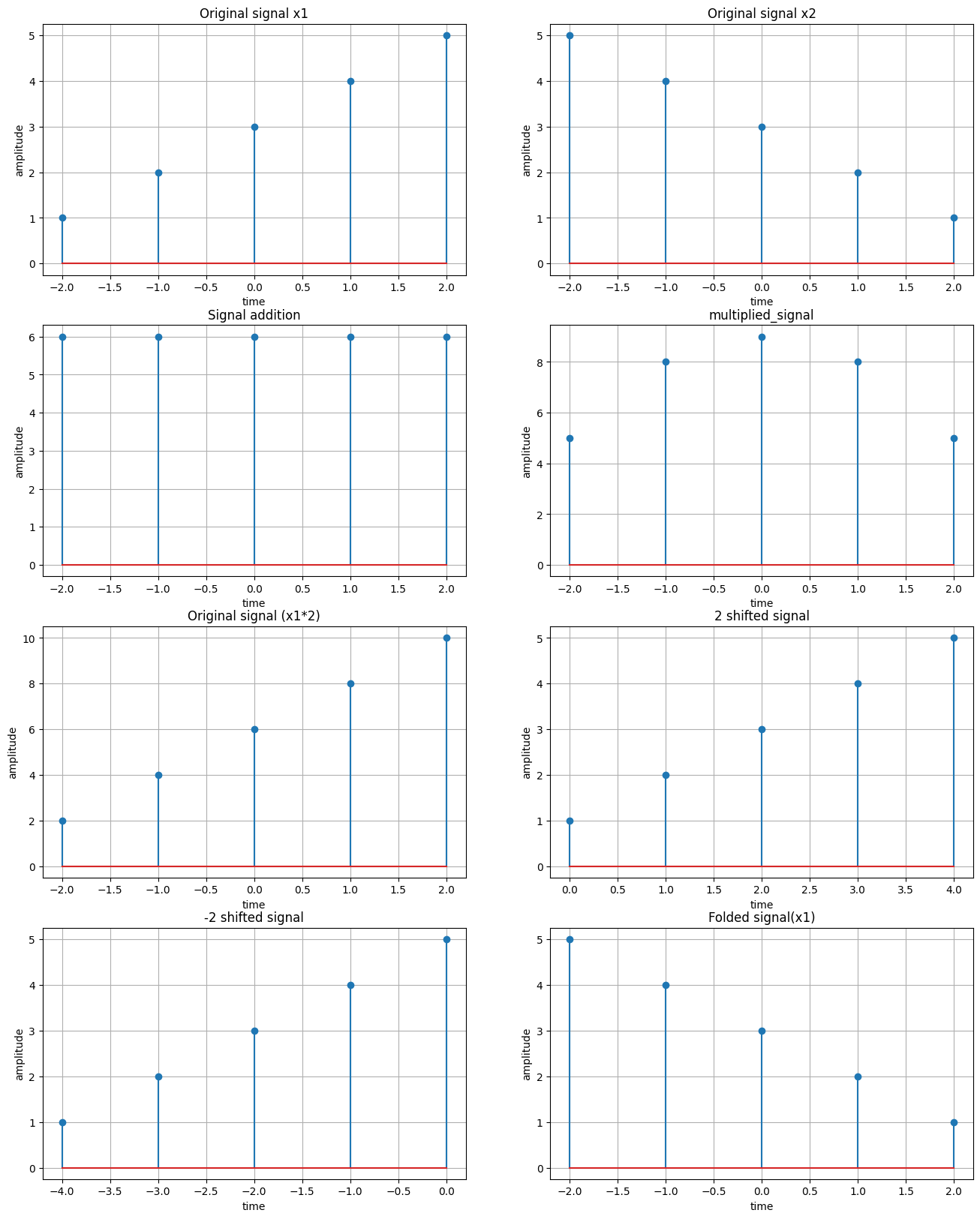
plt.stem(n,folded\_signal)

plt.xlabel("time")

plt.ylabel('amplitude')

plt.title('Folded signal(x1)')

plt.grid()

**Sample Input & Output:**

# Problem No: 02

## **Problem Name: Plot the following transformation x1(n) = 2x(n-5) - 3x(n+4)**

### **Objective:**

To plot the transformed signal x1(n) = 2x(n − 5) − 3x(n + 4) and visualize its effect on the original signal x(n).

### **Theory:**

A discrete-time signal x(n)x(n)x(n) can be modified using operations such as **scaling**, **shifting**, and **multiplication**.

The given transformation:

x1(n)=2x(n−5)−3x(n+4)

* **Time Shifting:**
  + x(n−5) shifts the signal **right** by 5 units (delayed).
  + x(n+4) shifts the signal **left** by 4 units (advanced).
* **Scaling:**
  + The signal x(n−5) is multiplied by **2**.
  + The signal x(n+4) is multiplied by **-3**.
* **Addition/Subtraction:**
  + The final signal is obtained by adding the two scaled versions.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

# Define the original signal x(n) (e.g., unit impulse or step function)

n = np.arange(-10, 10, 1)  # Discrete time range

x\_n = np.zeros\_like(n)  # Start with a zero array

# For simplicity, we will define x(n) as a unit impulse at n=0

x\_n[n == 0] = 1  # Unit impulse at n=0

# Apply the transformation x1(n) = 2x(n-5) - 3x(n+4)

x1\_n = 2 \* np.roll(x\_n, 5) - 3 \* np.roll(x\_n, -4)

# Plot the original signal and the transformed signal

plt.subplot(2, 1, 1)

plt.stem(n, x\_n)

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

Text(0.5, 1.0, 'Original Signal x(n)')

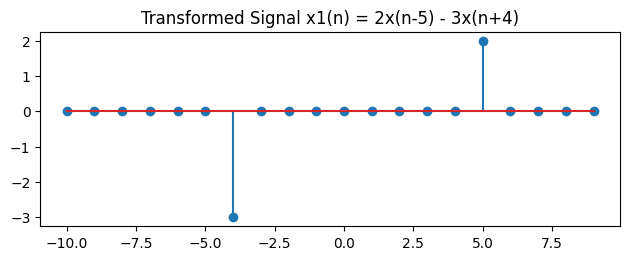
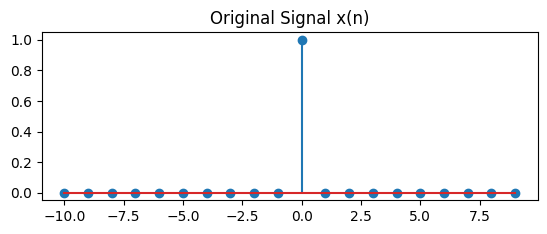
plt.subplot(2, 1, 2)

plt.stem(n, x1\_n)

plt.title("Transformed Signal x1(n) = 2x(n-5) - 3x(n+4)")

plt.tight\_layout()

plt.show()

**Sample Input & Output:**

# Problem No: 03

## **Problem Name: Explain and Implementation the unit Impulse sequence, the unit Step sequence, the unit Ramp sequence.**

### **Objective:**

To generate and understand three fundamental signals: unit impulse, unit step, and unit ramp.

### **Theory:**

In signal processing and discrete-time systems, the **Unit Impulse Sequence**, **Unit Step Sequence**, and **Unit Ramp Sequence** are fundamental discrete-time signals used for analysis and system characterization.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

# Generate a discrete-time index

n = np.arange(-10, 11)  # Range from -10 to 10

# Generate a random signal

random\_signal = np.random.uniform(-1, 1, len(n))  # Random values between -1 and 1

# Step Sequence (u[n])

step\_sequence = np.heaviside(n, 1)  # 1 for n >= 0, 0 for n < 0

# Impulse Sequence (δ[n])

impulse\_sequence = np.zeros\_like(n)  # All zeros

impulse\_sequence[n == 0] = 1  # 1 at n = 0

# Ramp Sequence (r[n])

ramp\_sequence = np.where(n >= 0, n, 0)  # n for n >= 0, else 0

# Plot the signals

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

# Plot Random Signal

plt.subplot(4, 1, 1)

plt.stem(n, random\_signal)

plt.title("Random Signal")

plt.xlabel("n")

plt.ylabel("Amplitude")

# Plot Step Sequence

plt.subplot(4, 1, 2)

plt.stem(n, step\_sequence)

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

plt.xlabel("n")

plt.ylabel("Amplitude")

# Plot Impulse Sequence

plt.subplot(4, 1, 3)

plt.stem(n, impulse\_sequence)

plt.title("Impulse Sequence δ[n]")

plt.xlabel("n")

plt.ylabel("Amplitude")

# Plot Ramp Sequence

plt.subplot(4, 1, 4)

plt.stem(n, ramp\_sequence)

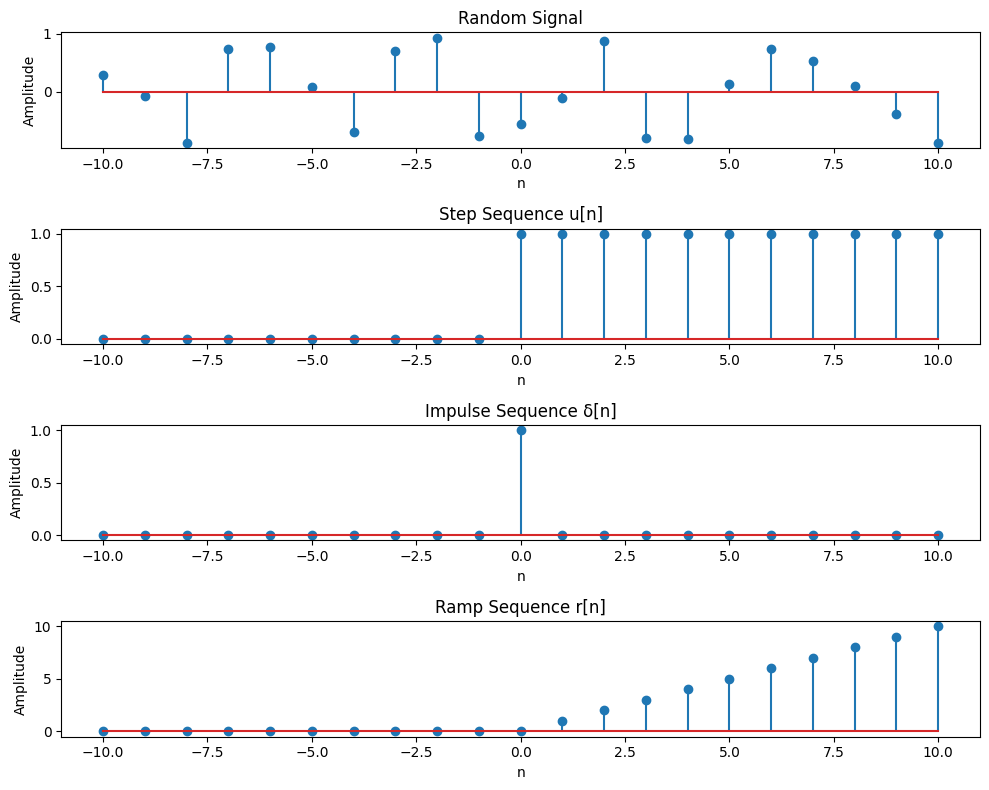
plt.title("Ramp Sequence r[n]")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.tight\_layout()

plt.show()

**  
Sample Input & Output:**

# Problem No: 04

## **Problem Name: Explain and Implement convolution of signal.**

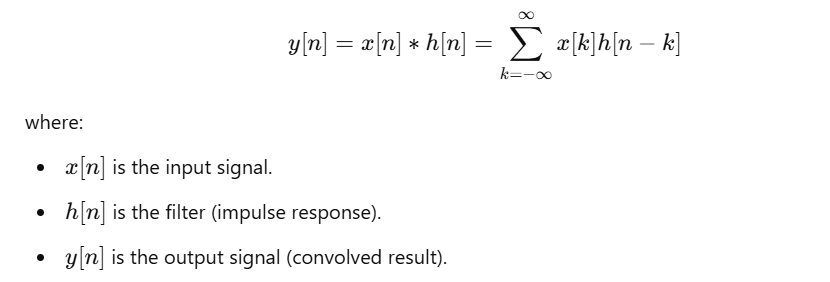
### **Objective:**

To compute and visualize the convolution of two discrete-time signals, demonstrating how the output signal is formed by combining the input signal and the system's impulse response.

### **Theory:**

**Convolution** is a mathematical operation that combines two signals to form a third signal. It describes the output of a linear time-invariant (LTI) system when the input signal is passed through the system's impulse response.

Mathematically, discrete-time convolution is given by:



# Source Code:

import numpy as np

import matplotlib.pyplot as plt

# Generate two random signals

np.random.seed(42)  # Ensures reproducibility

x = np.random.randint(0, 10, 10)  # First random signal (length 10)

h = np.random.randint(0, 10, 5)   # Second random signal (length 5)

# Compute convolution (It combines two signals to produce a third signal)

y = np.convolve(x, h, mode='full')  # mode='full' returns the complete convolution result of length (N+M-1).

# Normalize convolution result

y\_norm = y / np.max(np.abs(y))  # Normalize by maximum absolute value

# Define time indices

n\_x = np.arange(len(x))  # Time indices for x

n\_h = np.arange(len(h))  # Time indices for h

n\_y = np.arange(len(y))  # Time indices for y

# Plot original signals and convolution result

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

# Plot x[n]

plt.subplot(3, 1, 1)

plt.stem(n\_x, x, linefmt='b-', markerfmt='bo', basefmt='k')

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

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

# Plot h[n]

plt.subplot(3, 1, 2)

plt.stem(n\_h, h, linefmt='r-', markerfmt='ro', basefmt='k')

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

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

# Plot normalized convolution result y[n]

plt.subplot(3, 1, 3)

plt.stem(n\_y, y\_norm, linefmt='g-', markerfmt='go', basefmt='k')

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

plt.xlabel("n")

plt.ylabel("Normalized Amplitude")

plt.grid()

plt.tight\_layout()

plt.show()

# Print values

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

print("Filter h[n]:", h)

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

print("Normalized Convolution Output y[n]:", y\_norm)

**Sample Input & Output:**

Input Signal x[n]: [6 3 7 4 6 9 2 6 7 4]

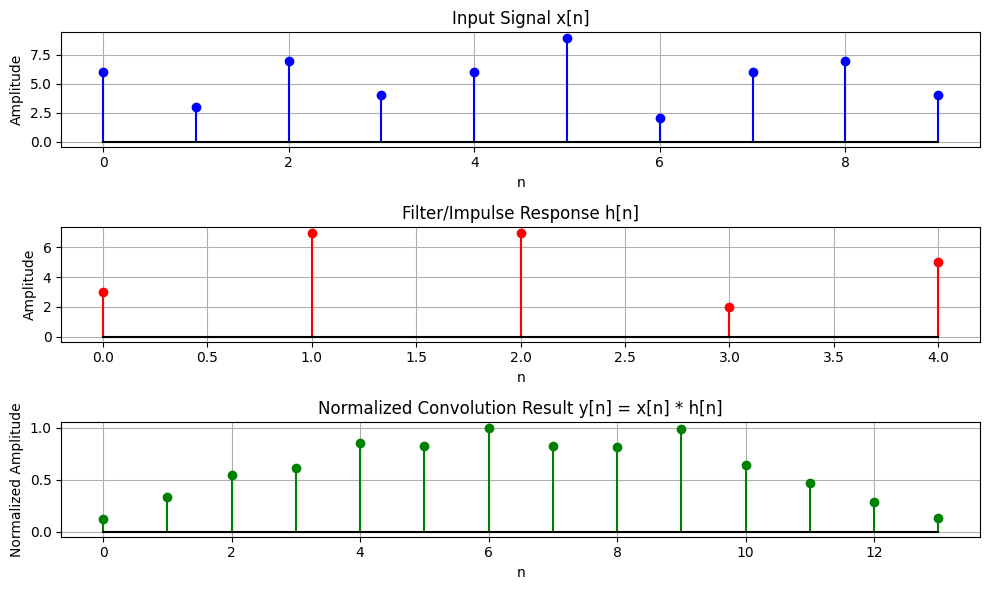
Filter h[n]: [3 7 7 2 5]

Convolution Output y[n]: [ 18 51 84 94 131 126 154 127 125 152 99 72 43 20]

Normalized Convolution Output y[n]: [0.11688312 0.33116883 0.54545455 0.61038961 0.85064935 0.81818182

1. 0.82467532 0.81168831 0.98701299 0.64285714 0.46753247

0.27922078 0.12987013]

****

# Problem No: 05

## **Problem Name: Explain and Implement Correlation of signal.**

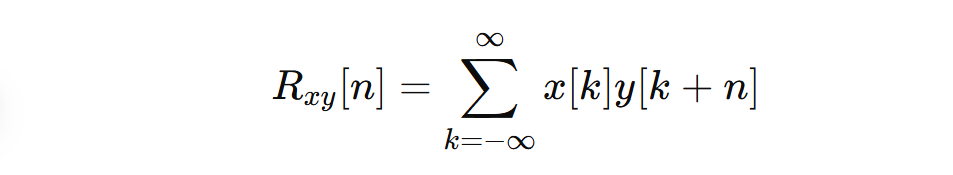
### **Objective:**

To compute and visualize the correlation of signals, including both **auto-correlation** (a signal with itself) and **cross-correlation** (between two signals), to analyze their similarity and periodicity.

### **Theory:**

**1. Cross-Correlation:**

Cross-correlation measures the similarity between two different signals as one is shifted relative to the other. It helps in detecting patterns and aligning signals. Mathematically, cross-correlation is defined as:

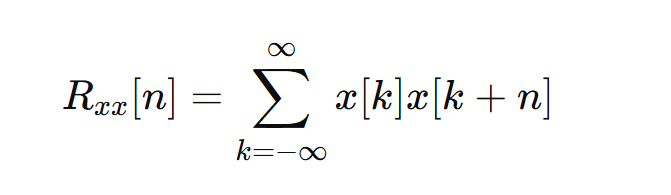


**Applications of Cross-Correlation:**

* **Signal synchronization** in communication systems.
* **Template matching** in image processing.
* **Feature detection** in machine learning.

**2. Auto-Correlation:**

Auto-correlation measures the similarity of a signal with itself over different shifts. It helps in identifying repetitive patterns within a signal. Mathematically, it is given by:



**Applications of Auto-Correlation:**

* **Speech processing:** Identifying periodicity in audio signals.
* **Radar and sonar:** Detecting objects by analyzing echo patterns.
* **DNA sequencing:** Finding repeated gene patterns.

# Source Code:

import numpy as np

import matplotlib.pyplot as plt

# Create two signals

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

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

# Compute Cross-Correlation (Cross-Correlation: Measures similarity between two different signals like x,y)

cross\_corr = np.correlate(x, y, mode='full')

# Compute Auto-Correlation of x (Auto-Correlation: Measures similarity of a signal with itself )

auto\_corr\_x = np.correlate(x, x, mode='full')

# Compute Auto-Correlation of y

auto\_corr\_y = np.correlate(y, y, mode='full')

# Normalize Cross-Correlation

norm\_factor = np.sqrt(np.correlate(x, x, mode='full')[len(x) - 1] \* np.correlate(y, y, mode='full')[len(y) - 1])

cross\_corr\_norm = cross\_corr / norm\_factor

# Normalize Auto-Correlations

auto\_corr\_x\_norm = auto\_corr\_x / auto\_corr\_x[len(x) - 1]  # Divide by max value

auto\_corr\_y\_norm = auto\_corr\_y / auto\_corr\_y[len(y) - 1]  # Divide by max value

# Plot Cross-Correlation (Shows how x and y align at different shifts)

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

plt.subplot(3, 1, 1)

plt.stem(range(-len(x) + 1, len(x)), cross\_corr\_norm)

plt.title("Normalized Cross-Correlation of x and y")

plt.xlabel("Lag")

plt.ylabel("Correlation (Normalized)")

# Plot Auto-Correlation of x

plt.subplot(3, 1, 2)

plt.stem(range(-len(x) + 1, len(x)), auto\_corr\_x\_norm)

plt.title("Normalized Auto-Correlation of x")

plt.xlabel("Lag")

plt.ylabel("Correlation (Normalized)")

# Plot Auto-Correlation of y

plt.subplot(3, 1, 3)

plt.stem(range(-len(y) + 1, len(y)), auto\_corr\_y\_norm)

plt.title("Normalized Auto-Correlation of y")

plt.xlabel("Lag")

plt.ylabel("Correlation (Normalized)")

plt.tight\_layout()

plt.show()

# Print correlation values

print("Normalized Cross-Correlation:", cross\_corr\_norm)

print("Normalized Auto-Correlation of x:", auto\_corr\_x\_norm)

print("Normalized Auto-Correlation of y:", auto\_corr\_y\_norm)

**Sample input & Output:**

Normalized Cross-Correlation: [0.01818182 0.07272727 0.18181818 0.36363636 0.63636364 0.8

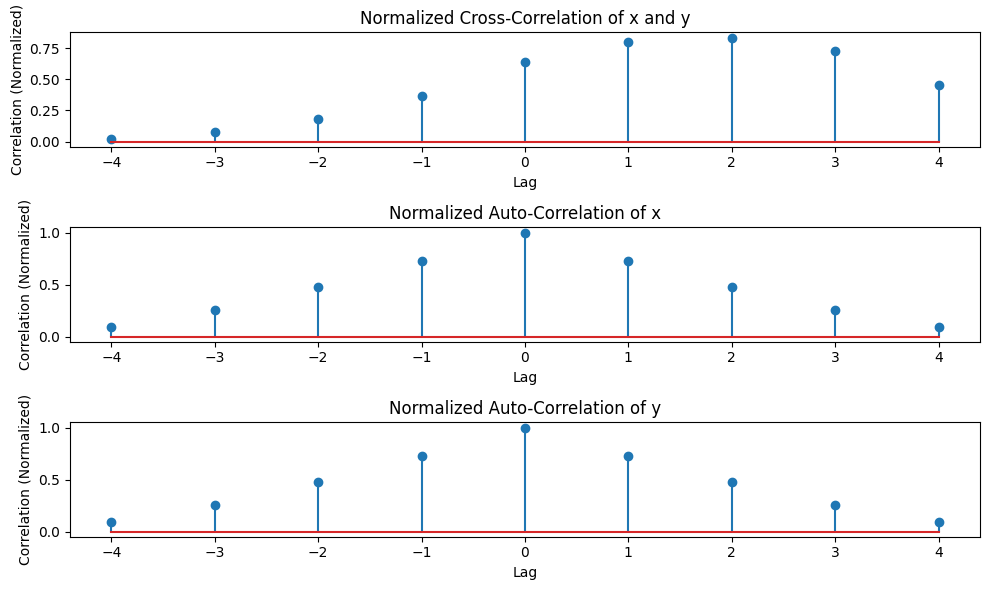
0.83636364 0.72727273 0.45454545]

Normalized Auto-Correlation of x: [0.09090909 0.25454545 0.47272727 0.72727273 1. 0.72727273

0.47272727 0.25454545 0.09090909]

Normalized Auto-Correlation of y: [0.09090909 0.25454545 0.47272727 0.72727273 1. 0.72727273

0.47272727 0.25454545 0.09090909]

****

# Problem No: 06

## **Problem Name: Extract relevant features such as filtering, feature extraction, pick detection, heart rate etc. from PPG signal.**

### **Objective:**

To extract relevant features such as filtering, peak detection, and heart rate calculation from a PPG (Photoplethysmogram) signal to analyze physiological parameters like heart rate and assess cardiovascular health.

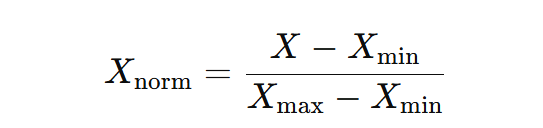
### **Theory:**

**1. PPG Signal:**

* The **Photoplethysmogram (PPG) signal** is an optical measurement of blood volume changes in tissue.
* It is used to monitor **heart rate (HR), blood oxygen levels (SpO₂), and vascular conditions**.
* The signal consists of a periodic waveform, reflecting heartbeats, and may contain **noise and artifacts**.

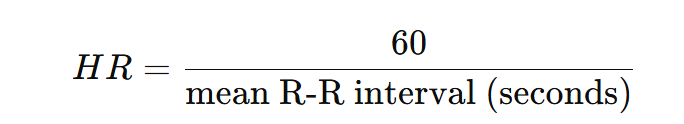
**2. Signal Processing Steps:**

1. **Signal Generation:**
   * A synthetic PPG signal is created using sinusoidal functions with added noise.
   * **Option to introduce abnormalities** such as irregular peaks for arrhythmia simulation.
2. **Normalization:**
   * The signal is normalized using **Min-Max Scaling**:



* + Ensures all values lie between **0 and 1**.

1. **Bandpass Filtering (Noise Removal):**
   * A **Butterworth bandpass filter** is used to retain relevant heart rate frequencies.
   * Removes low-frequency drift and high-frequency noise.
2. **Peak Detection:**
   * Identifies **heartbeats** by detecting peaks in the filtered signal.
   * Uses scipy.signal.find\_peaks() with parameters such as **minimum distance between peaks** and **prominence**.
3. **Feature Extraction:**
   * **R-R Intervals** (time between consecutive heartbeats) are computed.
   * **Heart Rate (HR) is estimated** as:



1. **Abnormality Detection:**
   * **Bradycardia**: HR < 60 BPM (slow heart rate).
   * **Tachycardia**: HR > 100 BPM (fast heart rate).
   * **Arrhythmia**: **Irregular heartbeat** detected if R-R interval standard deviation is high.

# Source Code:

import numpy as np

import matplotlib.pyplot as plt

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

# Simulating a synthetic PPG signal (for demonstration)

fs = 1000  # Sampling frequency (Hz)

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

ppg\_signal = 0.5 \* np.sin(2 \* np.pi \* 1.2 \* t) + 0.3 \* np.sin(2 \* np.pi \* 0.5 \* t) + 0.1 \* np.random.randn(len(t))

# Step 1: Bandpass Filtering (0.5-5 Hz)

def bandpass\_filter(signal, lowcut, highcut, fs, order=4):

    nyquist = 0.5 \* fs

    low = lowcut / nyquist

    high = highcut / nyquist

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

    filtered\_signal = filtfilt(b, a, signal)

    return filtered\_signal

# Apply bandpass filter to PPG signal

lowcut = 0.5  # Low frequency (0.5 Hz)

highcut = 5.0  # High frequency (5 Hz)

filtered\_ppg = bandpass\_filter(ppg\_signal, lowcut, highcut, fs)

# Step 2: Peak Detection

peaks, \_ = find\_peaks(filtered\_ppg, distance=fs\*0.6)  # Minimum distance between peaks (0.6 seconds)

# Step 3: Heart Rate Calculation

peak\_intervals = np.diff(peaks) / fs  # Time between successive peaks (in seconds)

heart\_rate = 60 / np.mean(peak\_intervals)  # Heart rate in beats per minute (bpm)

# Plot the results

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

# Original PPG Signal

plt.subplot(3, 1, 1)

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

plt.title('Raw PPG Signal')

plt.xlabel('Time (seconds)')

plt.ylabel('Amplitude')

plt.grid()

# Filtered PPG Signal

plt.subplot(3, 1, 2)

plt.plot(t, filtered\_ppg, label='Filtered PPG Signal', color='orange')

plt.scatter(t[peaks], filtered\_ppg[peaks], color='red', label='Detected Peaks')

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

plt.xlabel('Time (seconds)')

plt.ylabel('Amplitude')

plt.grid()

# Heart Rate Information

plt.subplot(3, 1, 3)

plt.text(0.5, 0.5, f'Heart Rate: {heart\_rate:.2f} bpm', fontsize=14, ha='center')

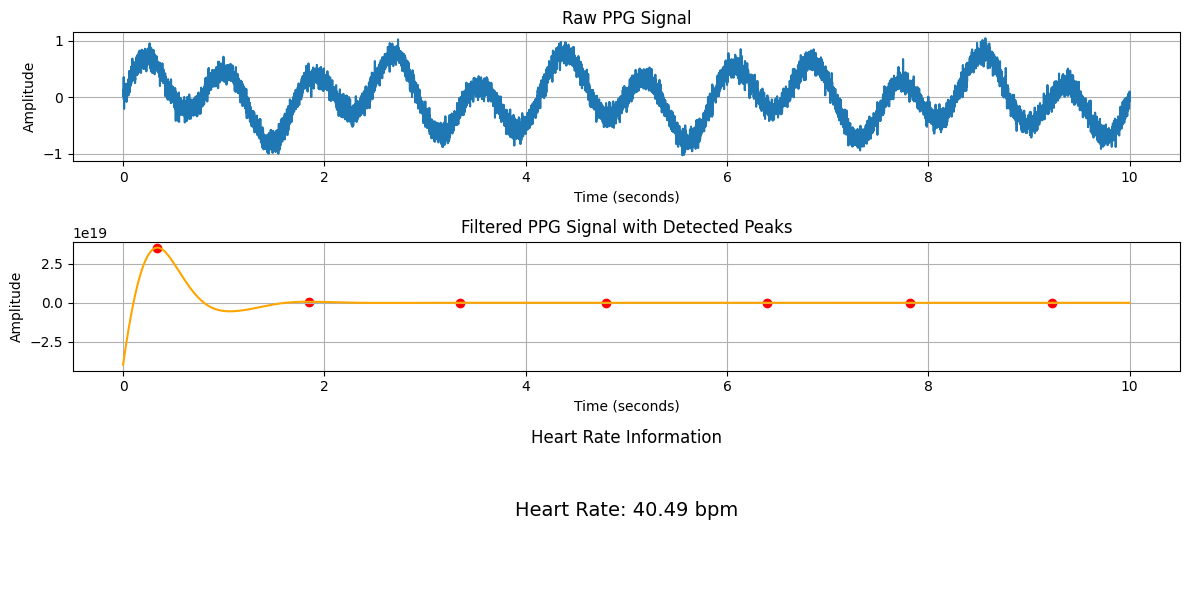
plt.axis('off')

plt.title('Heart Rate Information')

plt.tight\_layout()

plt.show()

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

****  
**Sample Input & Output:**

# Problem No: 07

## **Problem Name: Explain and Implement Discrete Fourier Transform (DFT) using python.**

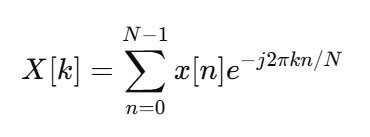
### **Objective:**

To explain and implement the Discrete Fourier Transform (DFT) using Python, and demonstrate how it can be used to analyze the frequency content of a discrete-time signal.

### **Theory:**

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

DFT transforms a **time-domain signal** into its **frequency-domain representation**. It is mathematically defined as:

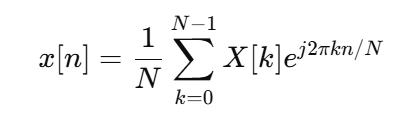


where:

* x[n] is the input signal.
* X[k] represents the frequency components of x[n].
* N is the total number of samples.
* k is the frequency index.

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

To recover the time-domain signal from its frequency representation, we use:



DFT is widely used for **signal analysis, filtering, and frequency domain processing**.

**3. Implementation Details:**

* The function DFT(x) computes the **DFT from scratch** using nested loops.
* A signal is generated with two frequency components (70 Hz and 90 Hz).
* The DFT is applied to analyze the frequency spectrum.
* The magnitude spectrum is plotted for interpretation.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

# Step 1: Create a sample discrete signal (for example, a sum of two sinusoids)

fs = 1000  # Sampling frequency (Hz)

t = np.arange(0, 1, 1/fs)  # Time vector for 1 second

f1 = 50  # Frequency of first sinusoid (Hz)

f2 = 120  # Frequency of second sinusoid (Hz)

# Create a signal that is the sum of two sinusoids

x = np.sin(2 \* np.pi \* f1 \* t) + np.sin(2 \* np.pi \* f2 \* t)

# Step 2: Compute the DFT of the signal manually using the DFT formula

N = len(x)  # Length of the signal

X = np.zeros(N, dtype=complex)  # Initialize the DFT array

# Calculate the DFT using the DFT formula

for k in range(N):

    sum\_val = 0

    for n in range(N):

        sum\_val += x[n] \* np.exp(-2j \* np.pi \* k \* n / N)

    X[k] = sum\_val

# Step 3: Compute the frequency axis for plotting

frequencies = np.fft.fftfreq(N, 1/fs)  # Frequencies corresponding to DFT bins

# Step 4: Plot the original signal and the magnitude of the DFT

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

# Plot the original signal

plt.subplot(2, 1, 1)

plt.plot(t, x)

plt.title('Original Signal (Sum of Two Sinusoids)')

plt.xlabel('Time (seconds)')

plt.ylabel('Amplitude')

plt.grid()

# Plot the magnitude of the DFT (frequency domain representation)

plt.subplot(2, 1, 2)

plt.plot(frequencies[:N//2], np.abs(X)[:N//2])  # Plot only the positive frequencies

plt.title('Magnitude of DFT (Frequency Domain)')

plt.xlabel('Frequency (Hz)')

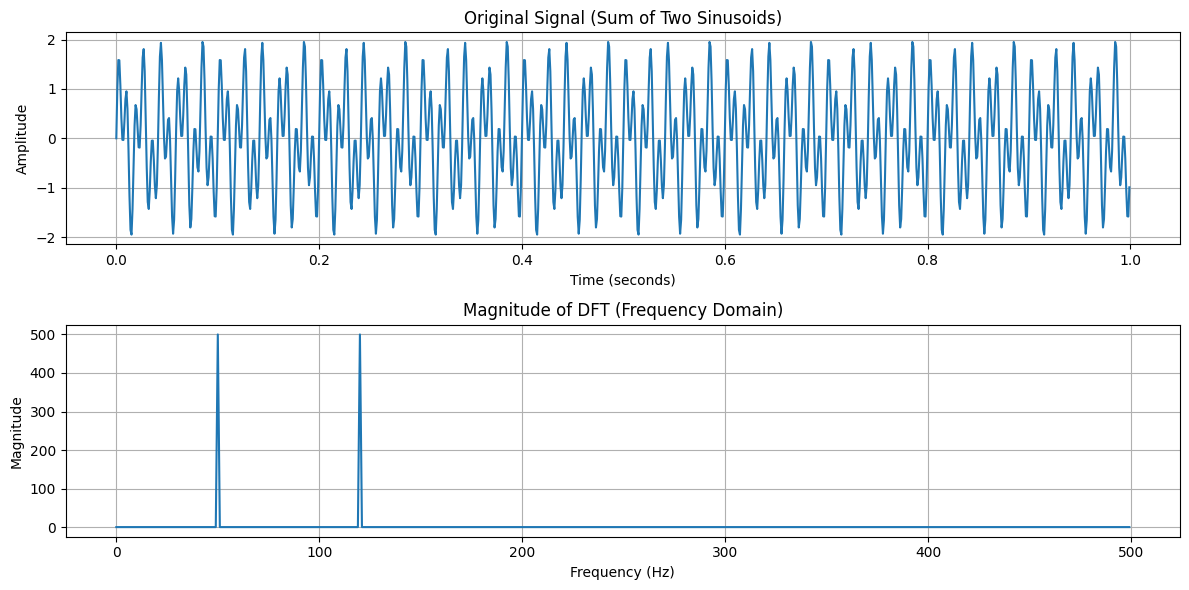
plt.ylabel('Magnitude')

plt.grid()

plt.tight\_layout()

plt.show()

**Sample Input & Output:**

****

# Problem No: 08

## **Problem Name: Explain and Implement Frequency bin using python.**

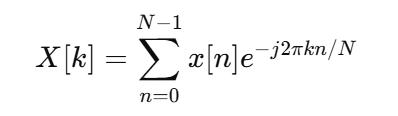
### **Objective:**

To understand and implement the Discrete Fourier Transform (DFT) and its frequency bins, and to analyze the frequency content of a discrete-time signal using Python.

### **Theory:**

**1. Fast Fourier Transform (FFT):**

FFT is an efficient algorithm to compute the **Discrete Fourier Transform (DFT)**. It converts a **time-domain signal** into the **frequency domain** using:

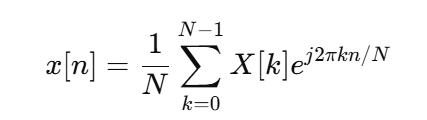


where:

* x[n] is the input signal.
* X[k] represents its frequency components.
* N is the number of samples.

**2. Frequency Filtering in FFT Domain:**

* High-frequency components **above 500 Hz** are removed by setting their magnitude to **zero** in the frequency spectrum.
* The cleaned signal is **reconstructed** using the **Inverse FFT (IFFT)**:



**3. Generating a Noisy Signal:**

* A **pure sinusoidal wave** at **440 Hz** (A4 note in music) is created.
* **Random noise** is added to simulate real-world signal corruption.
* FFT filtering helps **remove noise** and restore the original signal.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

from scipy.fft import fft, ifft, fftfreq

# Parameters

Fs = 1000  # Sampling frequency

T = 1 / Fs  # Sampling interval

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

# Generate signals

pure\_signal = np.sin(2 \* np.pi \* 440 \* t)  # 440 Hz sine wave

noisy\_signal = pure\_signal + np.random.normal(0, 0.5, t.shape)  # Add noise

# Apply FFT and filter

fft\_signal = fft(noisy\_signal)

freqs = fftfreq(len(fft\_signal), T)

fft\_signal[np.abs(freqs) > 500] = 0  # Remove frequencies > 500 Hz

cleaned\_signal = ifft(fft\_signal).real  # Inverse FFT to get cleaned signal

# Plotting

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

plt.subplot(3, 1, 1)

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

plt.subplot(3, 1, 2)

plt.plot(t, noisy\_signal, 'r', label="Noisy Signal")

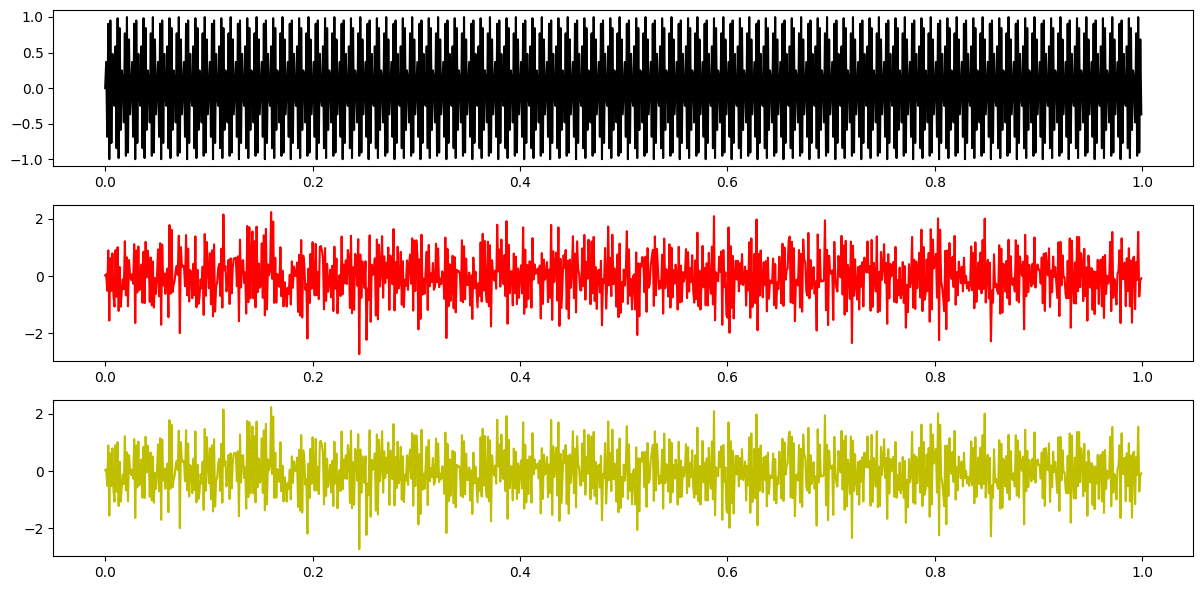
plt.subplot(3, 1, 3)

plt.plot(t, cleaned\_signal, 'y', label="Cleaned Signal")

plt.tight\_layout()

plt.show()

**Sample Input & Output:**

****