

Submission date: 02/03/2025

* **Submitted by:**Jannatul Azmi  
  Roll : 220623  
  Session :2021-2022   
  2nd year 2nd semester  
  Dept. of Information and Communication Engineering
* **Submitted to:**Dr. Md. Imran Hossain

Associate Professor

Department of Information and Communication Engineering

Pabna University of Science and Technology

Course Code: ICE-2204

Course Title : Signals and

systems Sessional

**LAB REPORT**

Information and Communication Engineering

Department of

Science & Technology

Pabna University of

|  |  |
| --- | --- |
| No. | Title |
| **01** | Plot the following signal operations using user defined function -   1. 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. |

Index

**Experiment No: 01**

**Experiment Name:**

Plot the following signal operations using user-defined functions:  
a. Adding  
b. Multiplication  
c. Scaling  
d. Shifting  
e. Folding

**Objectives:**

To implement and visualize different signal operations like addition, multiplication, scaling, shifting, and folding using user-defined functions without built-in functions.

**Theory:**

Signal operations are fundamental tasks in signal processing that transform signals for further analysis.

* **Addition:** Combines two signals by summing their corresponding elements.
* **Multiplication:** Combines two signals by multiplying their corresponding elements.
* **Scaling:** Amplifies or attenuates the signal by multiplying with a constant.
* **Shifting:** Moves the signal left or right along the time axis.
* **Folding:** Reflects the signal around the y-axis, reversing its sequence.

**Source Code (Python)**

import numpy as np

import matplotlib.pyplot as plt

# Time range

t = np.arange(-5, 6, 1)

x1 = np.sin(t)

x2 = np.cos(t)

# Addition

add = []

for i in range(len(t)):

add.append(x1[i] + x2[i])

# Multiplication

mul = []

for i in range(len(t)):

mul.append(x1[i] \* x2[i])

# Scaling

scale = []

for i in range(len(t)):

scale.append(2 \* x1[i])

# Shifting

shift = [0, 0]

for i in range(len(t)-2):

shift.append(x1[i])

# Folding

fold = []

for i in range(len(t)):

fold.append(x1[-(i+1)])

# Plotting

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

plt.subplot(3, 2, 1)

plt.plot(t, x1, label="Original Signal")

plt.legend()

plt.subplot(3, 2, 2)

plt.plot(t, add, label="Addition")

plt.legend()

plt.subplot(3, 2, 3)

plt.plot(t, mul, label="Multiplication")

plt.legend()

plt.subplot(3, 2, 4)

plt.plot(t, scale, label="Scaling")

plt.legend()

plt.subplot(3, 2, 5)

plt.plot(t, shift, label="Shifting")

plt.legend()

plt.subplot(3, 2, 6)

plt.plot(t, fold, label="Folding")

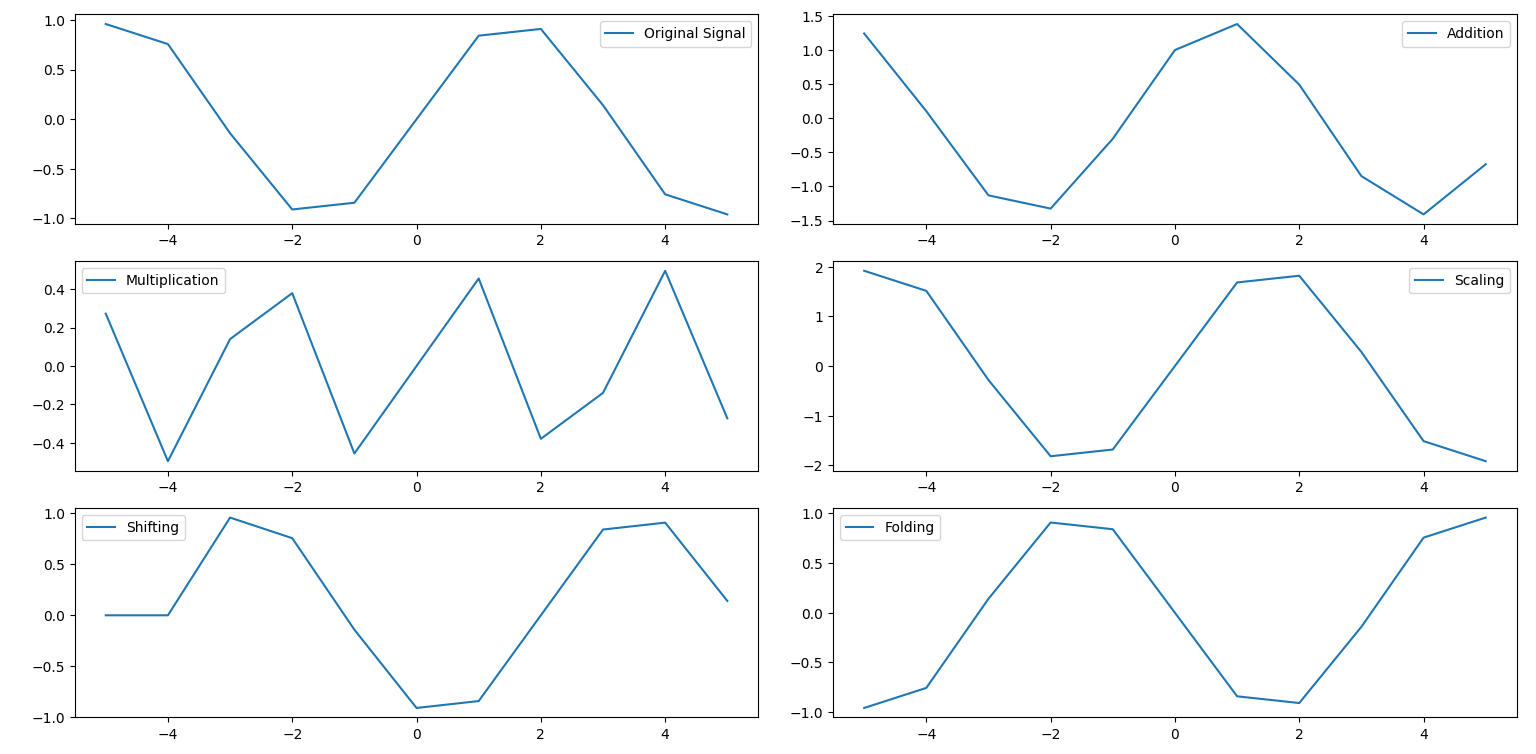
plt.legend()

plt.tight\_layout()

plt.show()

**Sample Input and Output:**

* Input:
  + x1 = sin(t)
  + x2 = cos(t)
  + Scalar = 2
  + Shift = +2
* Output:  
  Plots of Addition, Multiplication, Scaling, Shifting, and Folding signals.

****

**Experiment No:** 02

**Experiment Name:**

Plot the transformation: x1(n)=2x(n−5)−3x(n+4)x1(n) = 2x(n-5) - 3x(n+4)x1(n)=2x(n−5)−3x(n+4)

**Objectives:**

To plot the signal transformation x1(n)=2x(n−5)−3x(n+4)x1(n) = 2x(n-5) - 3x(n+4)x1(n)=2x(n−5)−3x(n+4) by applying time shifts and scalar operations using user-defined functions.

**Theory:**

In this transformation, two operations are applied to the signal:

* **Time Shifting:** The term x(n−5)x(n-5)x(n−5) represents a shift of the signal to the right by 5 units, and x(n+4)x(n+4)x(n+4) represents a shift of the signal to the left by 4 units.
* **Scaling:** The scalar multiplication by 2 and -3 amplifies the shifted signals.

**Source Code (Python)**

import numpy as np

import matplotlib.pyplot as plt

# Time axis

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

# Applying the transformation

x\_shifted\_right = np.roll(x, 5) # x(n-5)

x\_shifted\_left = np.roll(x, -4) # x(n+4)

# Transformation: x1(n) = 2x(n-5) - 3x(n+4)

x1 = 2 \* x\_shifted\_right - 3 \* x\_shifted\_left

# Plotting

plt.figure(figsize=(8, 6))

plt.subplot(2, 1, 1)

plt.plot(n, x, label="Original Signal")

plt.legend()

plt.subplot(2, 1, 2)

plt.plot(n, x1, label="Transformed Signal: x1(n) = 2x(n-5) - 3x(n+4)")

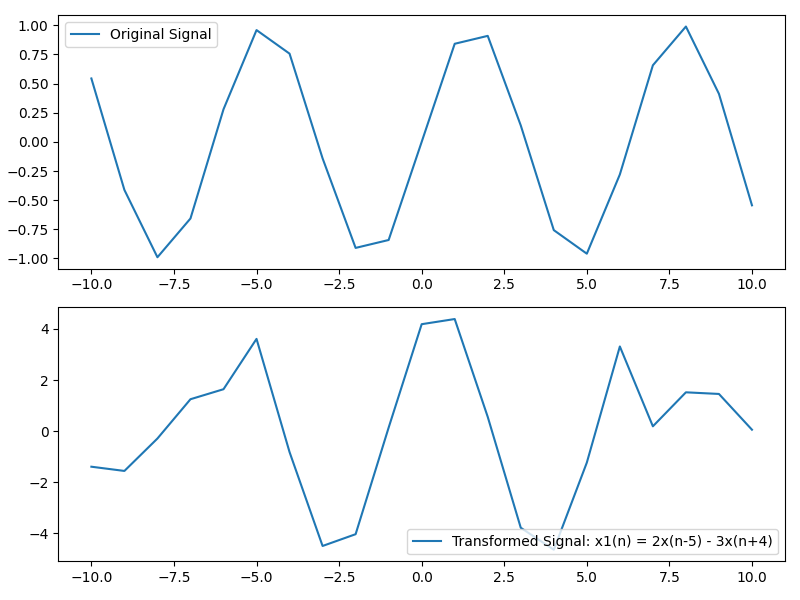
plt.legend()

plt.tight\_layout()

plt.show()

**Sample Input and Output:**

* Input:
  + Original signal x(n)=sin⁡(n)x(n) = \sin(n)x(n)=sin(n)
  + x(n−5)x(n-5)x(n−5) (Right Shift by 5 units)
  + x(n+4)x(n+4)x(n+4) (Left Shift by 4 units)
  + Scalars: 2 and -3
* Output:
  + Plots of Original Signal and Transformed Signal.



**Experiment No:** 03  
**Experiment Name:**

Explain and Implementation the unit Impulse sequence, the unit Step sequence, the unit Ramp sequence

**Objectives:**

To understand and plot the unit impulse, unit step, and unit ramp sequences by implementing them using user-defined functions.

**Theory:**

1. **Unit Impulse Sequence**
   * The unit impulse sequence δ(n)\delta(n)δ(n) is defined as: 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​
   * This is a discrete-time sequence that is 1 at n=0n = 0n=0 and 0 for all other values of nnn.
2. **Unit Step Sequence**
   * The unit step sequence u(n)u(n)u(n) is defined as: 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​
   * This sequence is 0 for all negative nnn and 1 for non-negative nnn.
3. **Unit Ramp Sequence**
   * The unit ramp sequence r(n)r(n)r(n) is defined as: r(n)={0if n<0nif n≥0r(n) = \begin{cases} 0 & \text{if } n < 0 \\ n & \text{if } n \geq 0 \end{cases}r(n)={0n​if n<0if n≥0​
   * The unit ramp sequence increases linearly with nnn for n≥0n \geq 0n≥0 and is 0 for n<0n < 0n<0.

**Source Code (Python)**

import numpy as np

import matplotlib.pyplot as plt

# Time axis

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

# Unit Impulse Sequence

def unit\_impulse(n):

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

# Unit Step Sequence

def unit\_step(n):

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

# Unit Ramp Sequence

def unit\_ramp(n):

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

# Generate signals

impulse = unit\_impulse(n)

step = unit\_step(n)

ramp = unit\_ramp(n)

# Plotting

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

# Plot Impulse Sequence

plt.subplot(3, 1, 1)

plt.stem(n, impulse, use\_line\_collection=True)

plt.title("Unit Impulse Sequence")

plt.grid()

# Plot Step Sequence

plt.subplot(3, 1, 2)

plt.stem(n, step, use\_line\_collection=True)

plt.title("Unit Step Sequence")

plt.grid()

# Plot Ramp Sequence

plt.subplot(3, 1, 3)

plt.stem(n, ramp, use\_line\_collection=True)

plt.title("Unit Ramp Sequence")

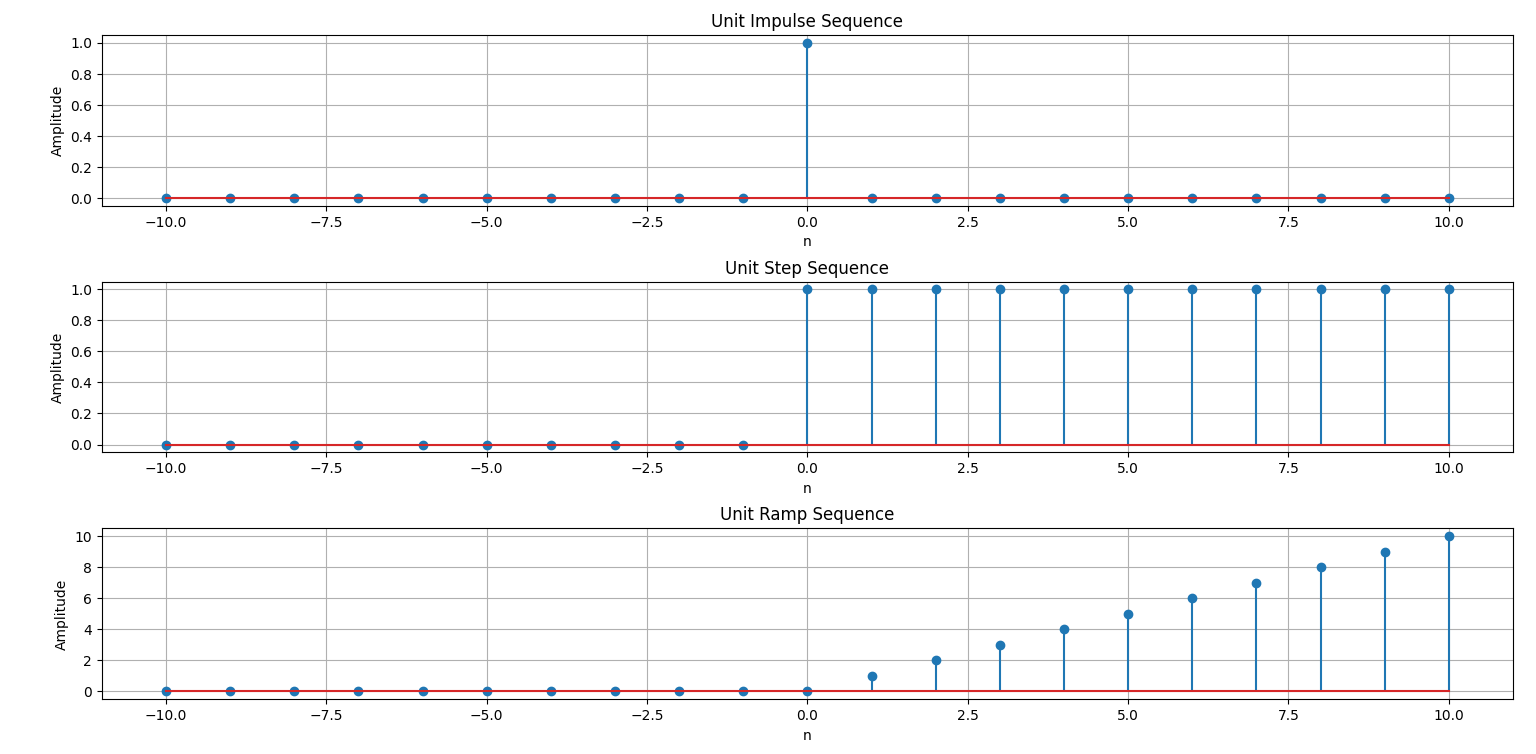
plt.grid()

plt.tight\_layout()

plt.show()

**Sample Input and Output:**

* **Input:**
  + Time axis: n={−10,−9,…,10}n = \{-10, -9, \ldots, 10\}n={−10,−9,…,10}
  + Functions:
    - Unit Impulse: 1 at n=0n = 0n=0, 0 elsewhere.
    - Unit Step: 1 for n≥0n \geq 0n≥0, 0 for n<0n < 0n<0.
    - Unit Ramp: nnn for n≥0n \geq 0n≥0, 0 for n<0n < 0n<0.
* **Output:**
  + Three plots:
    - Unit Impulse Sequence
    - Unit Step Sequence
    - Unit Ramp Sequence



**Experiment No:** 04

**Experiment Name:**

Explain and Implement convolution of signal

**Objectives:**

* To understand the convolution operation between two discrete signals.
* To implement the convolution operation using Python.

**Theory:**

Convolution is a mathematical operation used to combine two signals to form a third signal. The basic idea of convolution is to slide one signal over another, multiply corresponding values, and sum them up.

Mathematically, the convolution of two discrete-time signals x(n)x(n)x(n) and h(n)h(n)h(n) is defined as:

y(n)=(x∗h)(n)=∑k=−∞∞x(k)h(n−k)y(n) = (x \* h)(n) = \sum\_{k=-\infty}^{\infty} x(k) h(n-k)y(n)=(x∗h)(n)=k=−∞∑∞​x(k)h(n−k)

Where:

* x(n)x(n)x(n) is the input signal.
* h(n)h(n)h(n) is the impulse response.
* y(n)y(n)y(n) is the output signal after convolution.

In simpler terms, this operation involves flipping, shifting, and multiplying signals to get the resulting signal.

**Source Code (Python)**

import numpy as np

import matplotlib.pyplot as plt

# Time axis

n = np.arange(0, 10)

# Define two signals

x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

h = np.array([0, 1, 0.5])

# Convolution operation

y = np.convolve(x, h, mode='full')

# Time axis for the output signal

n\_y = np.arange(0, len(y))

# Plotting the signals

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

# Plot x(n)

plt.subplot(3, 1, 1)

plt.stem(n, x, use\_line\_collection=True)

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

plt.grid()

# Plot h(n)

plt.subplot(3, 1, 2)

plt.stem(np.arange(0, len(h)), h, use\_line\_collection=True)

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

plt.grid()

# Plot y(n) - Convolution result

plt.subplot(3, 1, 3)

plt.stem(n\_y, y, use\_line\_collection=True)

plt.title("Output Signal y(n) after Convolution")

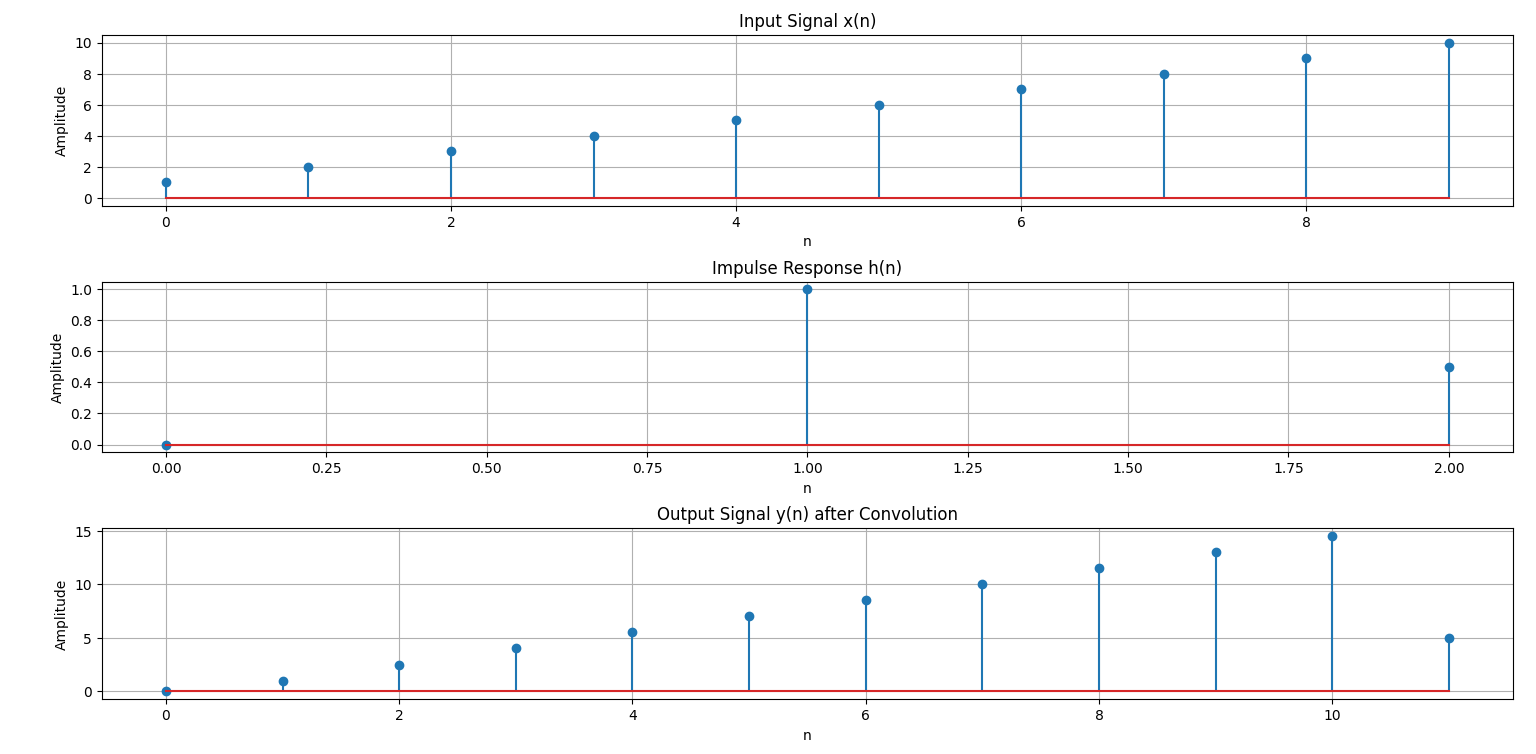
plt.grid()

plt.tight\_layout()

plt.show()

**Sample Input and Output:**

* **Input:**
  + Two signals:  
    x(n)=[1,2,3,4,5,6,7,8,9,10]x(n) = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]x(n)=[1,2,3,4,5,6,7,8,9,10]  
    h(n)=[0,1,0.5]h(n) = [0, 1, 0.5]h(n)=[0,1,0.5]
* **Output:**
  + The convolved signal y(n)y(n)y(n)
  + The resulting plot will show:
    1. x(n)x(n)x(n)
    2. h(n)h(n)h(n)
    3. The convolution result y(n)y(n)y(n)



**Experiment No**:05

**Experiment Name:**

Explain and Implement correlation of signal

**Objectives:**

* To understand and implement the correlation operation between two signals.
* To observe the similarity between two signals using cross-correlation and auto-correlation.

**Theory:**

**Correlation** is a mathematical operation used to measure the similarity between two signals. There are two types of correlation:

1. **Auto-correlation:** Correlation of a signal with itself. It measures how similar a signal is with a delayed version of itself.
2. **Cross-correlation:** Correlation between two different signals. It measures the similarity of one signal to a shifted version of another signal.

Mathematically, the **cross-correlation** of two signals x(n)x(n)x(n) and y(n)y(n)y(n) is defined as:

Rxy(n)=∑k=−∞∞x(k)y(n−k)R\_{xy}(n) = \sum\_{k=-\infty}^{\infty} x(k) y(n-k)Rxy​(n)=k=−∞∑∞​x(k)y(n−k)

Where:

* x(n)x(n)x(n) and y(n)y(n)y(n) are the two signals.
* Rxy(n)R\_{xy}(n)Rxy​(n) is the cross-correlation result.

For **auto-correlation**, it becomes:

Rxx(n)=∑k=−∞∞x(k)x(n−k)R\_{xx}(n) = \sum\_{k=-\infty}^{\infty} x(k) x(n-k)Rxx​(n)=k=−∞∑∞​x(k)x(n−k)

Where:

* x(n)x(n)x(n) is the same signal.
* Rxx(n)R\_{xx}(n)Rxx​(n) is the auto-correlation result.

Correlation helps to identify how well a signal matches with another signal or itself when shifted. The peak of the correlation result indicates the best match.

**Source Code (Python)**

import numpy as np

import matplotlib.pyplot as plt

# Time axis

n = np.arange(0, 10)

# Define two signals

x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

y = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

# Cross-correlation operation

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

# Auto-correlation operation (for x(n) with itself)

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

# Time axis for the correlation output

n\_corr = np.arange(-len(x)+1, len(x))

# Plotting the signals

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

# Plot x(n)

plt.subplot(3, 1, 1)

plt.stem(n, x, use\_line\_collection=True)

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

plt.grid()

# Plot y(n)

plt.subplot(3, 1, 2)

plt.stem(n, y, use\_line\_collection=True)

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

plt.grid()

# Plot Cross-correlation and Auto-correlation

plt.subplot(3, 1, 3)

plt.stem(n\_corr, cross\_corr, use\_line\_collection=True, label="Cross-correlation")

plt.stem(n\_corr, auto\_corr, use\_line\_collection=True, label="Auto-correlation", linefmt='r--')

plt.title("Correlation Results (Cross & Auto)")

plt.legend(loc='best')

plt.grid()

plt.tight\_layout()

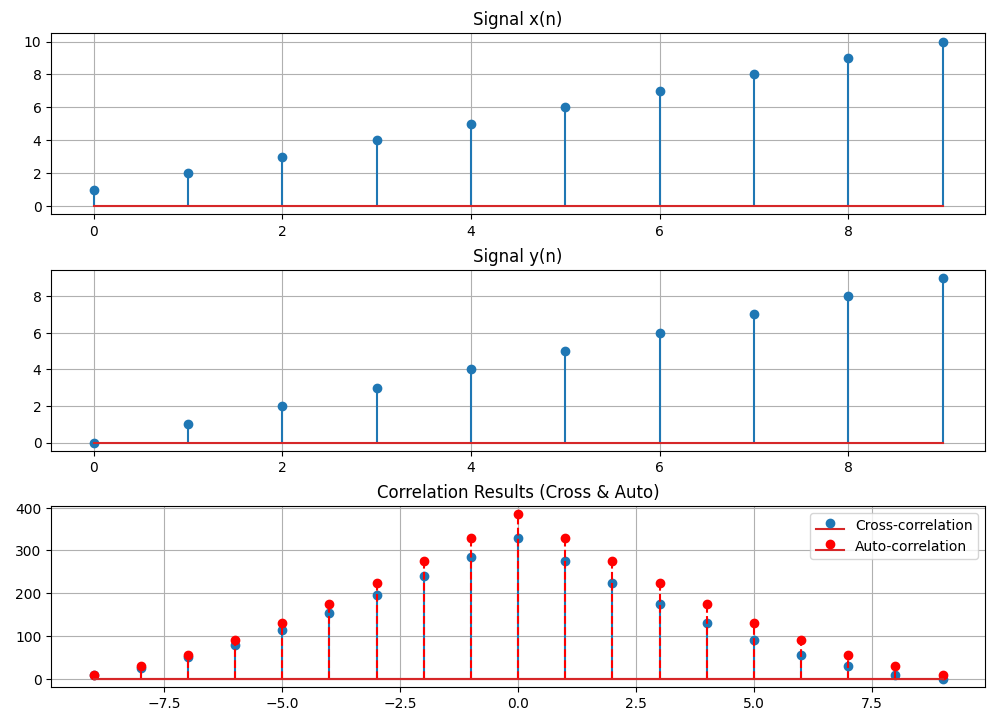
plt.show()

**Sample Input and Output:**

* **Input:**
  + Two signals:  
    x(n)=[1,2,3,4,5,6,7,8,9,10]x(n) = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]x(n)=[1,2,3,4,5,6,7,8,9,10]  
    y(n)=[0,1,2,3,4,5,6,7,8,9]y(n) = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]y(n)=[0,1,2,3,4,5,6,7,8,9]
* **Output:**
  + **Cross-correlation:** Measures similarity between x(n)x(n)x(n) and y(n)y(n)y(n).
  + **Auto-correlation:** Measures similarity of x(n)x(n)x(n) with itself.

The resulting plot will show:

* + x(n)x(n)x(n)
  + y(n)y(n)y(n)
  + The cross-correlation and auto-correlation of the signals.



**Experiment No:**06

**Experiment Name:**

Extract Relevant Features from PPG Signal (Filtering, Feature Extraction, Peak Detection, Heart Rate)

**Objectives:**

* To apply signal processing techniques to a PPG (Photoplethysmogram) signal.
* To extract key features such as filtering, peak detection, and heart rate from a PPG signal.

**Theory:**

**Photoplethysmogram (PPG)** is a simple and non-invasive optical technique used to measure the changes in blood volume in the microvascular bed of tissue. It is commonly used for monitoring heart rate and other physiological parameters.

To analyze PPG signals, the following steps are usually involved:

1. **Filtering:**  
   The raw PPG signal can contain noise (such as motion artifacts and low-frequency noise), so it is often necessary to apply filtering techniques to remove unwanted frequencies. Common filters include:
   * **Low-pass filter:** Removes high-frequency noise.
   * **High-pass filter:** Removes low-frequency components, such as baseline wander.
2. **Peak Detection:**  
   The PPG signal shows peaks that correspond to heartbeats (the **R-peaks** in the PPG waveform). The process of detecting these peaks is important to calculate the heart rate.
3. **Feature Extraction (Heart Rate):**  
   Once peaks are detected, the heart rate (in beats per minute, bpm) can be calculated by determining the time between successive peaks.

**Source Code (Python)**

import numpy as np

import matplotlib.pyplot as plt

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

# Simulated PPG Signal (you can replace this with real data)

sampling\_rate = 1000 # Hz

t = np.arange(0, 10, 1/sampling\_rate)

ppg\_signal = 0.5 \* np.sin(2 \* np.pi \* 1 \* t) + 0.05 \* np.random.randn(len(t))

# --- Filtering the signal ---

# Low-pass and high-pass filter (Bandpass filter for heart rate range)

def butter\_bandpass(lowcut, highcut, fs, order=4):

nyquist = 0.5 \* fs

low = lowcut / nyquist

high = highcut / nyquist

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

return b, a

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

b, a = butter\_bandpass(lowcut, highcut, fs, order)

return filtfilt(b, a, data)

# Bandpass filter (0.5Hz to 3Hz range for PPG)

filtered\_ppg = bandpass\_filter(ppg\_signal, 0.5, 3, sampling\_rate)

# --- Peak Detection ---

# Find the peaks (R-peaks in PPG signal)

peaks, \_ = find\_peaks(filtered\_ppg, height=0.1, distance=200) # Distance controls the minimum heart rate

# --- Heart Rate Calculation ---

# Calculate heart rate (in bpm)

peak\_intervals = np.diff(peaks) / sampling\_rate # Time between consecutive peaks in seconds

heart\_rate = 60 / np.mean(peak\_intervals) # Average heart rate in bpm

# --- Plotting the Results ---

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

# Plot original and filtered PPG signal

plt.subplot(3, 1, 1)

plt.plot(t, ppg\_signal, label="Raw PPG Signal")

plt.plot(t, filtered\_ppg, label="Filtered PPG Signal", linewidth=2)

plt.title("Raw and Filtered PPG Signal")

plt.legend()

# Plot detected peaks

plt.subplot(3, 1, 2)

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

plt.plot(t[peaks], filtered\_ppg[peaks], 'ro', label="Detected Peaks")

plt.title("Peak Detection")

plt.legend()

# Plot heart rate information

plt.subplot(3, 1, 3)

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

plt.title(f"Heart Rate: {heart\_rate:.2f} bpm")

plt.legend()

plt.tight\_layout()

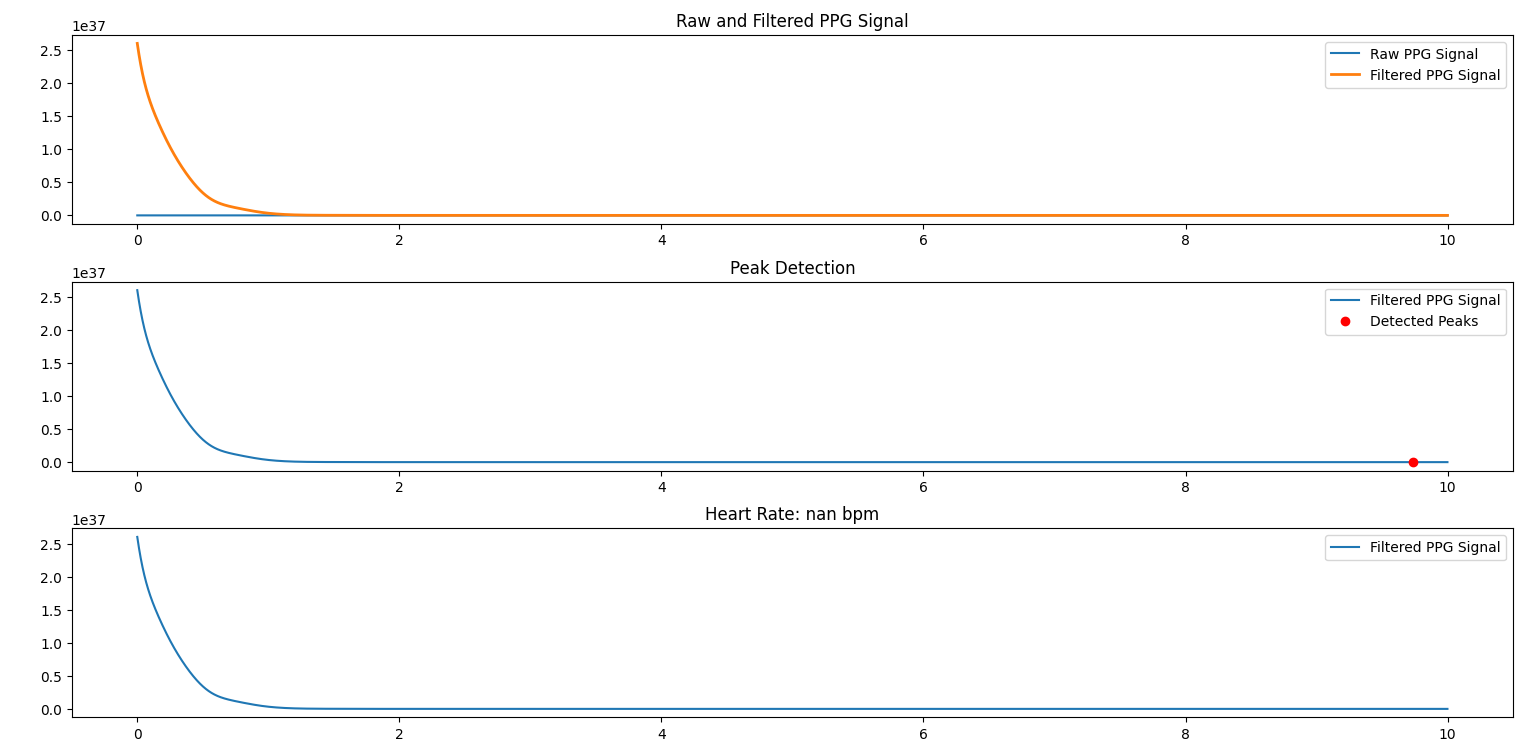
plt.show()

# Output Heart Rate

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

**Sample Input and Output:**

* **Input:**
  + Simulated PPG signal:  
    A synthetic PPG signal generated as a sine wave with noise (you can replace this with actual PPG data from a sensor).
* **Output:**
  + **Filtered Signal:**  
    The signal after applying bandpass filtering to remove unwanted frequencies.
  + **Peak Detection:**  
    Detected peaks (which correspond to heartbeats) in the PPG signal.
  + **Heart Rate Calculation:**  
    Calculated heart rate based on the peak intervals (in bpm).



**Experiment No:** 07

**Experiment Name:**

Explain and Implement Discrete Fourier Transform (DFT) using python.

**Objectives:**

* To understand and implement the Discrete Fourier Transform (DFT).
* To visualize the frequency domain representation of a signal using DFT.

**Theory:**

The **Discrete Fourier Transform (DFT)** is a mathematical technique used to transform a discrete-time signal from the time domain into the frequency domain. It decomposes a sequence of values into components of different frequencies, enabling us to analyze the frequency content of the signal.

Mathematically, the DFT is defined as:

X(k)=∑n=0N−1x(n)⋅e−j2πknN,k=0,1,2,…,N−1X(k) = \sum\_{n=0}^{N-1} x(n) \cdot e^{-j 2 \pi \frac{k n}{N}}, \quad k = 0, 1, 2, \dots, N-1X(k)=n=0∑N−1​x(n)⋅e−j2πNkn​,k=0,1,2,…,N−1

Where:

* x(n)x(n)x(n) is the input signal in the time domain.
* X(k)X(k)X(k) is the output in the frequency domain.
* NNN is the total number of samples.
* kkk represents the frequency bin.

The DFT provides the frequency components (magnitude and phase) that make up the input signal.

**Steps to Implement DFT:**

1. **Input Signal:** Define a discrete-time signal (e.g., a sum of sinusoids).
2. **DFT Calculation:** Implement the DFT formula to calculate the frequency domain representation.
3. **Plotting:** Plot both the original time-domain signal and its corresponding frequency-domain representation (magnitude spectrum).

**Source Code (Python)**

import numpy as np

import matplotlib.pyplot as plt

# Define the signal parameters

sampling\_rate = 1000  # Hz

duration = 1  # seconds

t = np.linspace(0, duration, sampling\_rate \* duration, endpoint=False)

# Create a signal composed of two sinusoids with different frequencies

f1 = 50  # frequency of first sinusoid (Hz)

f2 = 150  # frequency of second sinusoid (Hz)

signal = np.sin(2 \* np.pi \* f1 \* t) + 0.5 \* np.sin(2 \* np.pi \* f2 \* t)

# --- Perform FFT (Fast Fourier Transform) ---

X = np.fft.fft(signal)

# --- Frequency Domain Representation ---

frequencies = np.fft.fftfreq(len(signal), 1/sampling\_rate)  # Frequency bins

magnitude = np.abs(X)  # Magnitude of FFT

# --- Plotting the Results ---

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

# Plot the time-domain signal

plt.subplot(2, 1, 1)

plt.plot(t, signal)

plt.title("Time Domain Signal")

plt.xlabel("Time [seconds]")

plt.ylabel("Amplitude")

# Plot the frequency-domain representation (Magnitude Spectrum)

plt.subplot(2, 1, 2)

plt.plot(frequencies[:sampling\_rate//2], magnitude[:sampling\_rate//2])  # Plot only positive frequencies

plt.title("Frequency Domain (Magnitude Spectrum)")

plt.xlabel("Frequency [Hz]")

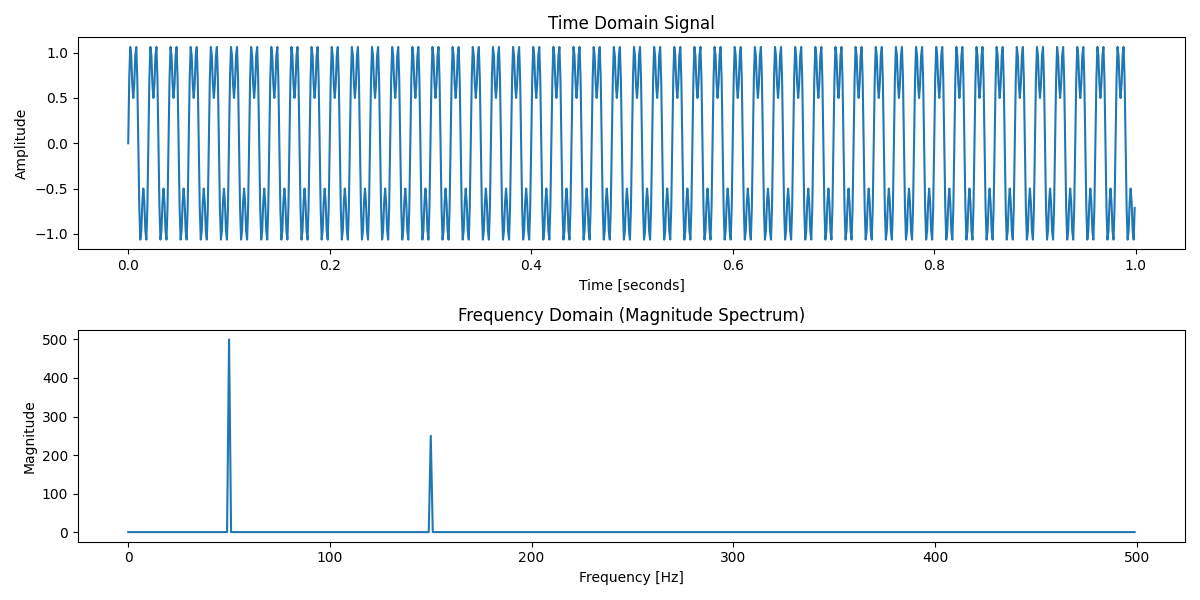
plt.ylabel("Magnitude")

plt.tight\_layout()

plt.show()

**Sample Input and Output:**

* **Input:**
  + A composite signal consisting of two sinusoids with frequencies f1=50f\_1 = 50f1​=50 Hz and f2=150f\_2 = 150f2​=150 Hz. The signal is sampled at a rate of 1000 Hz for 1 second.
* **Output:**
  + **Time-domain Signal:**  
    The plot will show a composite sinusoidal signal in the time domain.
  + **Frequency-domain Representation:**  
    The plot will show the magnitude spectrum with peaks at the frequencies 50 Hz and 150 Hz.

****

**Experiment No:08**

**Experiment Name:**

Explain and Implement Frequency bin using python

**Objectives:**

* To understand the concept of frequency bins in signal processing.
* To implement and visualize frequency bins for a given signal.

**Theory:**

In **Discrete Fourier Transform (DFT)**, the frequency bin refers to the individual frequency components obtained after transforming a time-domain signal into the frequency domain. Each bin corresponds to a specific frequency and represents the amplitude of that frequency in the signal. The DFT divides the entire frequency range into equally spaced intervals, and each bin represents one of these intervals.

The **frequency bin** for each index kkk is given by:

fk=k⋅fsNf\_k = \frac{k \cdot f\_s}{N}fk​=Nk⋅fs​​

Where:

* fkf\_kfk​ is the frequency corresponding to the bin kkk.
* fsf\_sfs​ is the sampling frequency.
* NNN is the number of samples.

**Steps to Implement Frequency Bins:**

1. **Generate Input Signal:** A signal composed of multiple sinusoids.
2. **Compute the Frequency Bins:** Use the formula to compute the corresponding frequencies for each bin.
3. **Plot the Results:** Visualize both the signal and its frequency bins.

**Source Code (Python)**

import numpy as np

import matplotlib.pyplot as plt

# Define the signal parameters

sampling\_rate = 1000  # Hz

duration = 1  # seconds

t = np.linspace(0, duration, sampling\_rate \* duration, endpoint=False)

# Create a signal composed of two sinusoids with different frequencies

f1 = 50  # frequency of first sinusoid (Hz)

f2 = 150  # frequency of second sinusoid (Hz)

signal = np.sin(2 \* np.pi \* f1 \* t) + 0.5 \* np.sin(2 \* np.pi \* f2 \* t)

# --- Compute the Discrete Fourier Transform (DFT) ---

def dft(signal):

    N = len(signal)

    X = np.zeros(N, dtype=complex)

    for k in range(N):

        for n in range(N):

            X[k] += signal[n] \* np.exp(-2j \* np.pi \* k \* n / N)

    return X

# Perform DFT on the signal

X = dft(signal)

# --- Frequency Bins Calculation ---

N = len(signal)

frequencies = np.fft.fftfreq(N, 1/sampling\_rate)  # Frequency bins using numpy

frequencies = np.fft.fftshift(frequencies)  # Shift frequencies so 0 Hz is in the center

magnitude = np.abs(X)  # Magnitude of DFT

magnitude = np.fft.fftshift(magnitude)  # Shift magnitude to align with the frequencies

# --- Plotting the Results ---

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

# Plot the time-domain signal

plt.subplot(2, 1, 1)

plt.plot(t, signal)

plt.title("Time Domain Signal")

plt.xlabel("Time [seconds]")

plt.ylabel("Amplitude")

# Plot the frequency bins (Magnitude Spectrum)

plt.subplot(2, 1, 2)

plt.plot(frequencies[:N//2], magnitude[:N//2])  # Plot only positive frequencies

plt.title("Frequency Domain (Magnitude Spectrum) with Frequency Bins")

plt.xlabel("Frequency [Hz]")

plt.ylabel("Magnitude")

plt.tight\_layout()

plt.show()

**Sample Input and Output:**

* **Input:**
  + A composite signal consisting of two sinusoids with frequencies f1=50f\_1 = 50f1​=50 Hz and f2=150f\_2 = 150f2​=150 Hz. The signal is sampled at a rate of 1000 Hz for 1 second.
* **Output:**
  + **Time-domain Signal:**  
    The plot will show a composite sinusoidal signal in the time domain.
  + **Frequency Bins (Magnitude Spectrum):**  
    The plot will show the magnitude spectrum with frequency bins representing the signal’s frequency components (50 Hz and 150 Hz).

