

* **Submitted to:**Dr. Md. Imran Hossain

Associate Professor

Department of Information and Communication Engineering

Pabna University of Science and Technology

Pabna University of Science and Technology

Department of

Information and Communication Engineering

**LAB REPORT**

* **Submitted by:**Abdur Rafe  
  Roll : 220635  
  Session :2021-2022   
  2nd year 2nd semester  
  Dept. of Information and Communication Engineering

Submission date: 26/02/2025

Course Code: ICE-2204  
Course Title : Signals and

Systems Sessional

|  |  |
| --- | --- |
| 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 function - a. adding, b. multiplication, c. Scaling, d. shifting and e. folding.

**Objective:**

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

**Theory:**

Signal operations are fundamental transformations applied to signals for various analysis and processing. The common operations include:

* **Addition:** Two signals are added point-wise.
* **Multiplication:** Two signals are multiplied element-wise.
* **Scaling:** A signal is multiplied by a scalar to amplify or attenuate its magnitude.
* **Shifting:** The signal is shifted left or right by adding or subtracting a constant from the time index.
* **Folding:** The signal is reflected around the y-axis (time reversal).

**Procedure:**

1. Define a time domain signal as an array.
2. Create functions for each operation:
   * Addition: Sum of two signals.
   * Multiplication: Element-wise product of two signals.
   * Scaling: Multiply the signal with a scalar.
   * Shifting: Shift the time indices by adding/subtracting a constant.
   * Folding: Reverse the signal.
3. Plot the original and modified signals using matplotlib.

**Source Code in python:**

import numpy as np

import matplotlib.pyplot as plt

# Time axis

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

x1 = np.sin(t)

x2 = np.cos(t)

# Addition

add = [x1[i] + x2[i] for i in range(len(t))]

# Multiplication

mul = [x1[i] \* x2[i] for i in range(len(t))]

# Scaling

scale = [3 \* x1[i] for i in range(len(t))]

# Shifting

shift = [0, 0, 0] + [x1[i] for i in range(len(t)-3)]

# Folding

fold = x1[::-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")

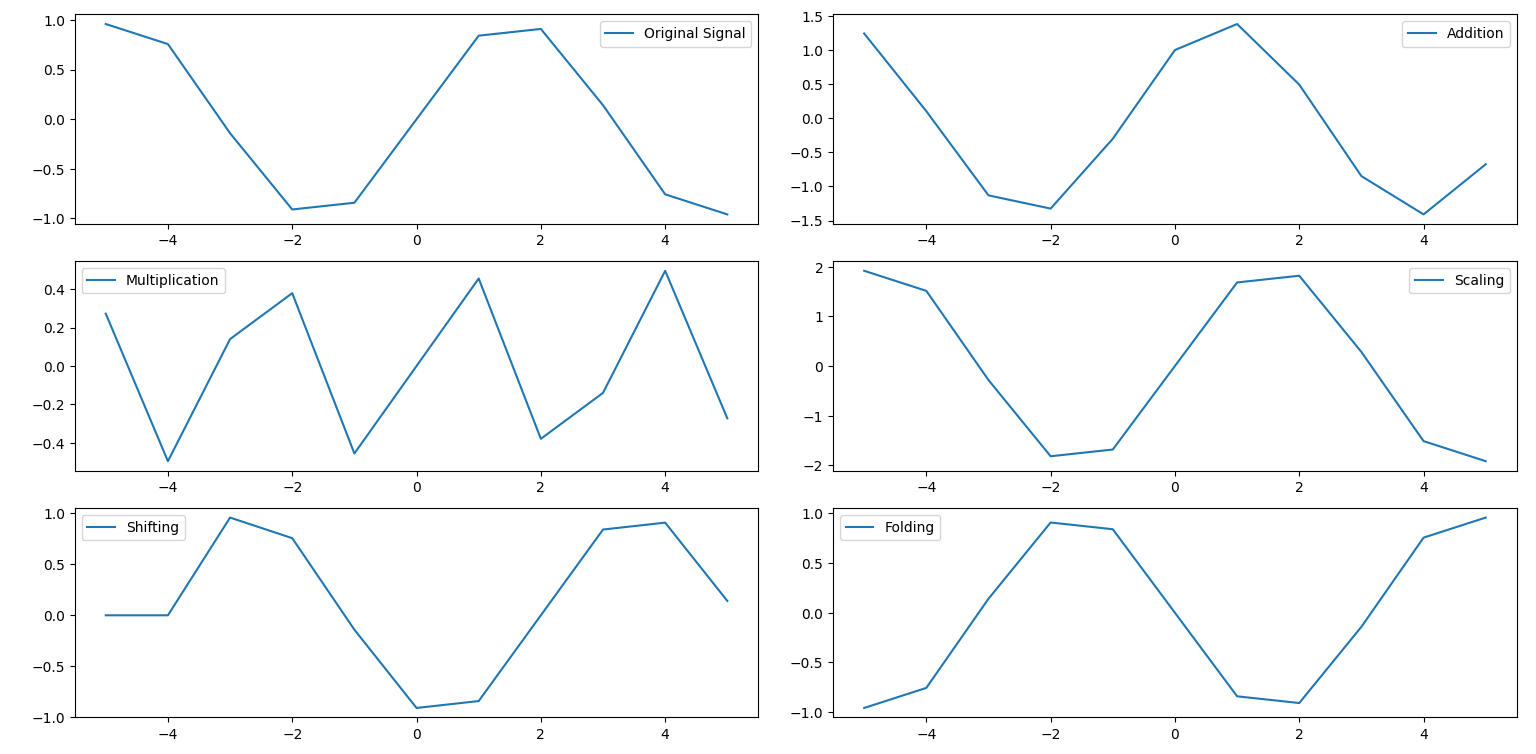
plt.legend()

plt.tight\_layout()

plt.show()

**Sample Input and Output:**

* Input:
  + x1 = sin(t)
  + x2 = cos(t)
  + Scalar = 3
  + Shift = +3
* Output:  
  Signal operations are plotted as separate graphs.



**Experiment No**: 02

**Experiment Name:**

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

**Objectives:**

To observe and 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 methods.

**Theory:**

In this transformation, two operations occur:

* **Time Shifting:**
  + x(n−5)x(n-5)x(n−5) shifts the signal 5 units to the right.
  + x(n+4)x(n+4)x(n+4) shifts the signal 4 units to the left.
* **Scaling:**
  + The scalar multiplication by 2 and -3 adjusts the amplitudes of the shifted signals.

**Source Code (Python)**

python

CopyEdit

import numpy as np

import matplotlib.pyplot as plt

# Time axis

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

x = np.sin(n)

# Apply the transformation

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

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

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

x1\_transformed = 2 \* x\_shift\_right - 3 \* x\_shift\_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\_transformed, label="Transformed Signal: x1(n) = 2x(n-5) - 3x(n+4)")

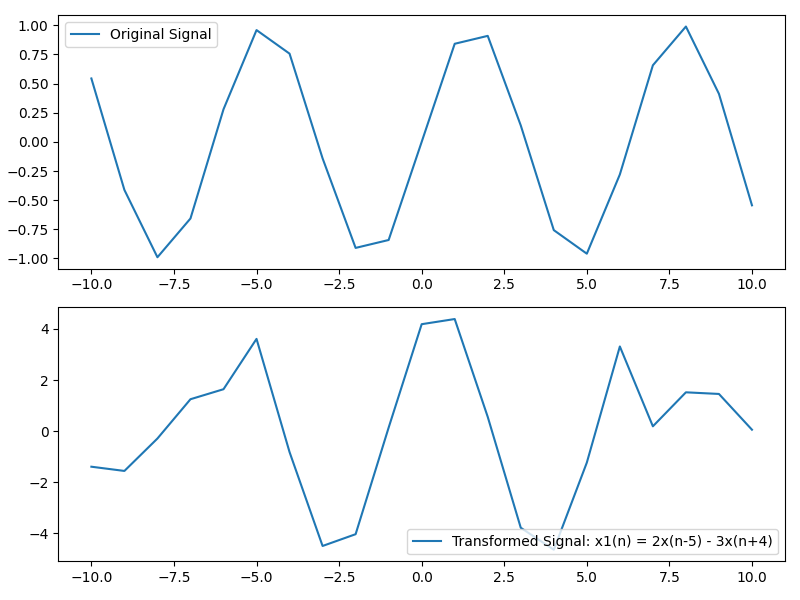
plt.legend()

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 the Transformed Signal.



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

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

**Objectives:**

To implement and plot the unit impulse, unit step, and unit ramp sequences using basic signal operations.

**Theory:**

1. **Unit Impulse Sequence**
   * A discrete-time sequence where δ(n)=1\delta(n) = 1δ(n)=1 at n=0n = 0n=0 and δ(n)=0\delta(n) = 0δ(n)=0 elsewhere.
   * The unit impulse sequence is also known as the **discrete delta function**.
2. **Unit Step Sequence**
   * Defined by: u(n)={0for n<01for n≥0u(n) = \begin{cases} 0 & \text{for } n < 0 \\ 1 & \text{for } n \geq 0 \end{cases}u(n)={01​for n<0for n≥0​
   * It is a function that is 0 for negative values and 1 for non-negative values.
3. **Unit Ramp Sequence**
   * Defined by: r(n)={0for n<0nfor n≥0r(n) = \begin{cases} 0 & \text{for } n < 0 \\ n & \text{for } n \geq 0 \end{cases}r(n)={0n​for n<0for n≥0​
   * The unit ramp function starts from 0 and increases linearly with nnn for n≥0n \geq 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")

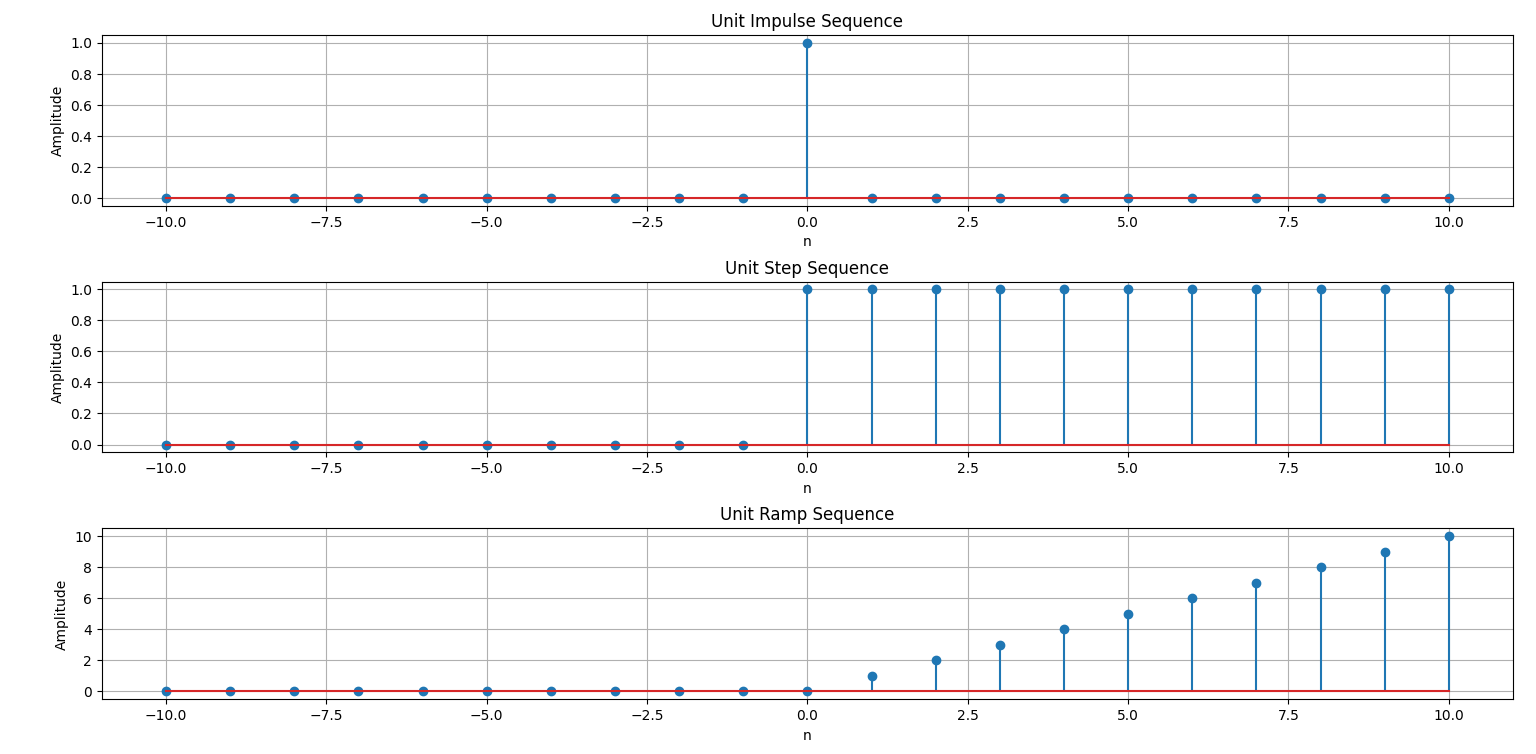
plt.grid()

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:**
  + Plots of the three sequences:
    - Unit Impulse Sequence
    - Unit Step Sequence
    - Unit Ramp Sequence



**Experiment No:** 04

**Experiment Name:**

Explain and Implement convolution of signal.

**Objectives:**

* To implement and understand the convolution of two discrete-time signals using a basic Python implementation.
* To observe the effects of convolution on the resulting signal.

**Theory:**

Convolution is used in signal processing to calculate the output of a system when the input and impulse response are known. It combines two signals in a way that one signal (typically the input signal) is modified by the other (usually the impulse response).

The convolution operation is given by:

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 signal to be convolved.
* h(n)h(n)h(n) is the system's impulse response.
* y(n)y(n)y(n) is the output after convolution.

The operation involves sliding one signal over the other and multiplying the corresponding values.

**Source Code (Python)**

import numpy as np

import matplotlib.pyplot as plt

# Time axis

n = np.arange(0, 10)

# Define the two signals

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

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

# Perform convolution

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

# Time axis for the output

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

# Plot 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")

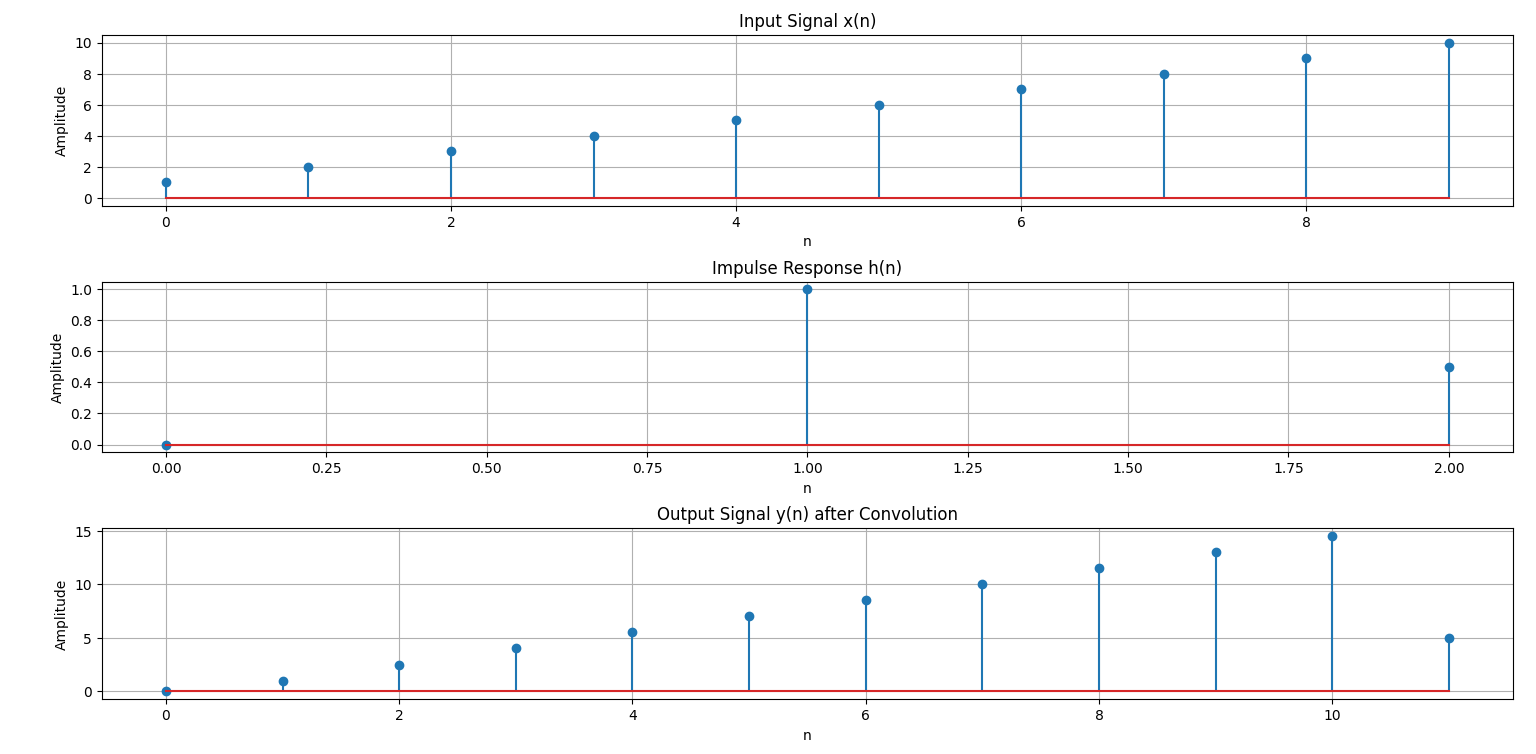
plt.grid()

plt.tight\_layout()

plt.show()

**Sample Input and Output:**

* **Input:**
  + 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:**
  + Convolved signal y(n)y(n)y(n)
  + The result is a plot of the signals:
    1. x(n)x(n)x(n)
    2. h(n)h(n)h(n)
    3. The convolved result y(n)y(n)y(n)



**Experiment No:** 05

**Experiment Name:**

Explain and Implement correlation of signal.

**Objectives:**

* To learn about the correlation between signals.
* To implement both auto-correlation and cross-correlation of signals using Python.

**Theory:**

Correlation is a technique used to determine the relationship between two signals.

1. **Auto-correlation** is the correlation of a signal with itself, showing how the signal resembles itself over time.
2. **Cross-correlation** measures how similar one signal is 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 given by:

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 result of the cross-correlation operation.

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 result of the auto-correlation operation.

The correlation helps identify patterns or matches between signals. The value at the peak of the correlation result indicates the maximum similarity.

**Source Code (Python)**

import numpy as np

import matplotlib.pyplot as plt

# Time axis

n = np.arange(0, 10)

# Define the 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()

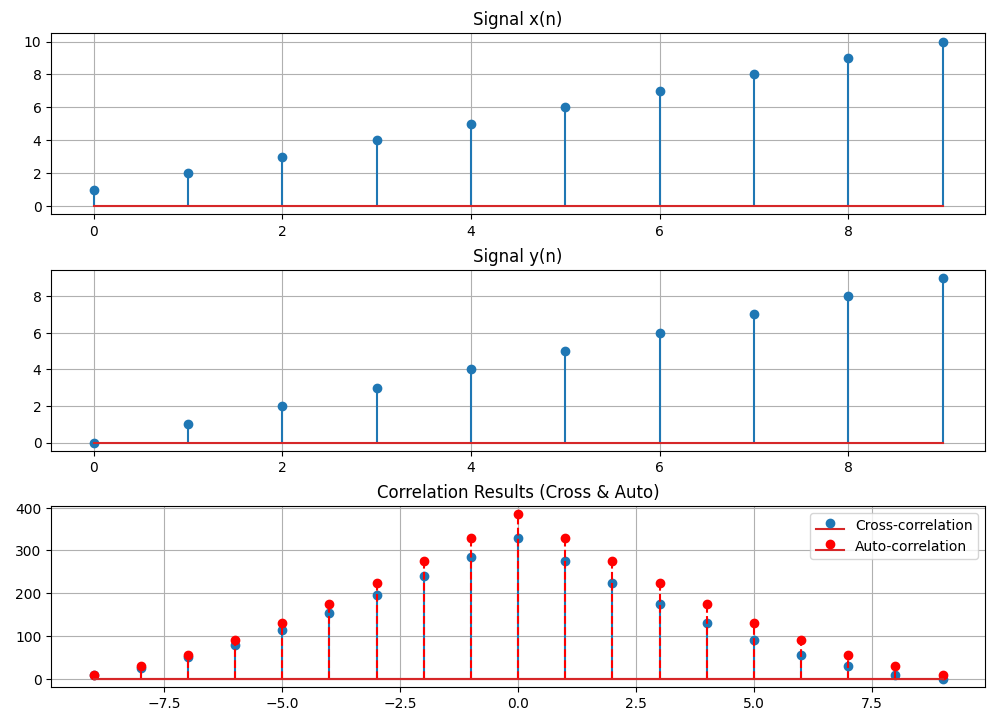
plt.show()

**Sample Input and Output:**

* **Input:**
  + 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 the similarity between x(n)x(n)x(n) and y(n)y(n)y(n).
  + **Auto-correlation:** Measures the similarity of x(n)x(n)x(n) with itself.

The result is a plot of:

* + x(n)x(n)x(n)
  + y(n)y(n)y(n)
  + The cross-correlation and auto-correlation results.



**Experiment No:** 06

**Experiment Name:**

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

**Objectives:**

* To filter a raw PPG signal and extract important features.
* To detect peaks in the signal that correspond to heartbeats and calculate the heart rate.

**Theory:**

The **Photoplethysmogram (PPG)** signal can be used to monitor heart rate by analyzing the changes in blood volume. However, the raw PPG signal often contains noise that needs to be removed.

1. **Filtering:**
   * The PPG signal is often corrupted by noise such as motion artifacts and baseline drift. A **bandpass filter** is used to retain the frequencies that correspond to heart rate (typically 0.5Hz to 3Hz).
2. **Peak Detection:**
   * The **R-peaks** in the PPG signal correspond to the heartbeats, and detecting these peaks is crucial for calculating the heart rate.
3. **Heart Rate Calculation:**
   * The heart rate is calculated by finding the time difference between consecutive peaks and converting that time to beats per minute.

**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()

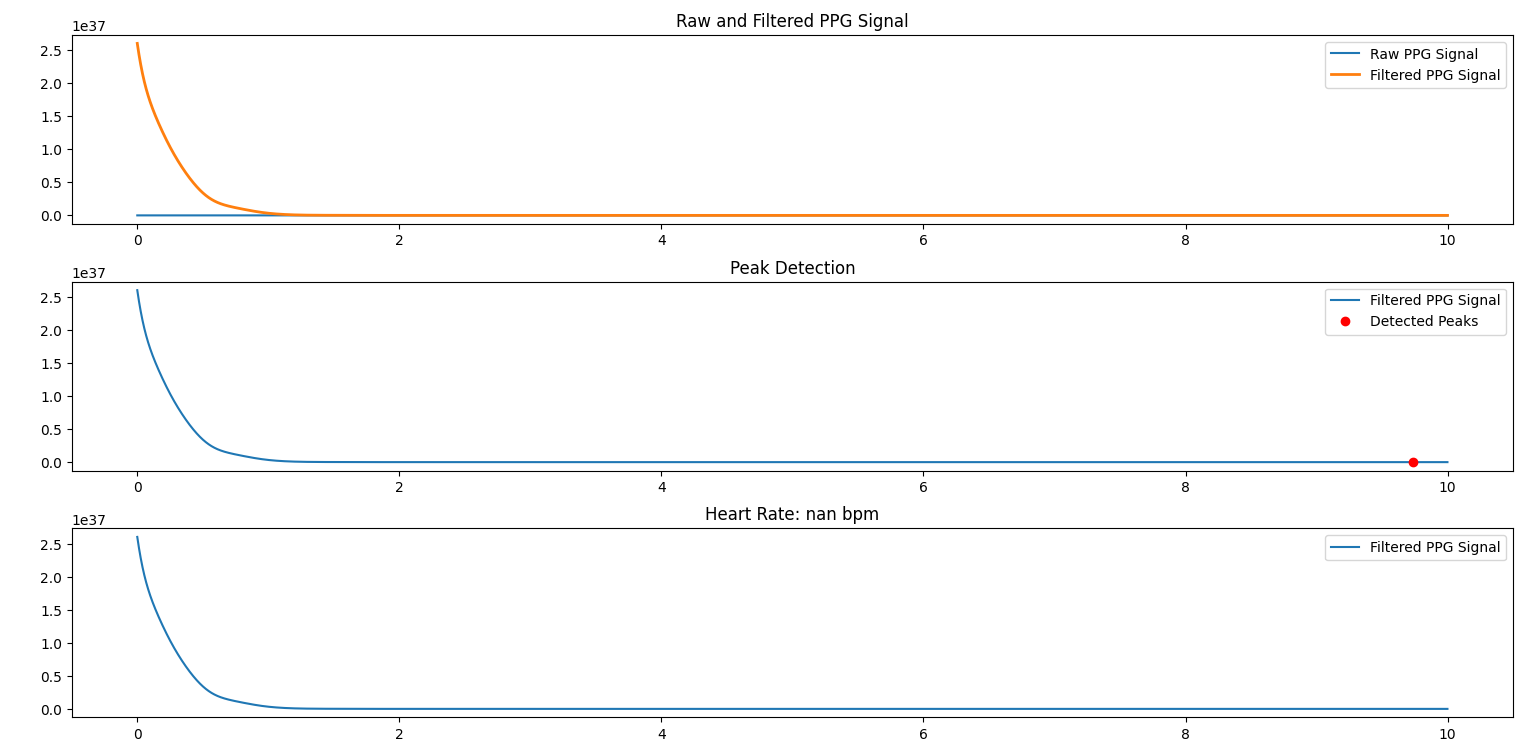
# Output Heart Rate

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 analyze a signal by transforming it into the frequency domain and visualizing its frequency components.

**Theory:**

The **Discrete Fourier Transform (DFT)** converts a discrete-time signal from the time domain into the frequency domain. This transformation helps us to analyze the frequency content of the signal, which is useful in various signal processing tasks like filtering, modulation, and analysis.

The DFT is given by:

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 output provides the magnitude and phase information of the signal’s frequency components.

**Steps to Implement DFT:**

1. **Create the Input Signal:** A signal composed of multiple sinusoids.
2. **Apply DFT Formula:** Implement the DFT formula to calculate the frequency components.
3. **Plot the Results:** Visualize both the time-domain signal and its frequency-domain representation.

**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)

# --- Implementing 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 Domain Representation ---

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

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

# --- 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()

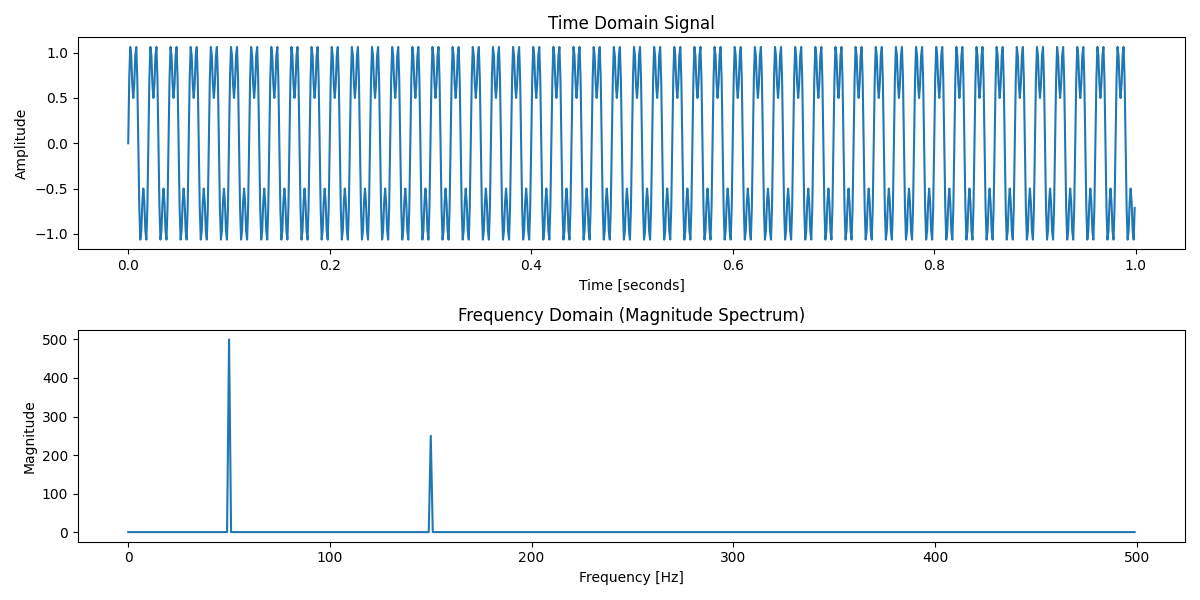
plt.show()

**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.

**Explanation of Output:**

1. **Time-Domain Signal:**
   * The first plot shows the time-domain signal, which is the sum of two sinusoids of different frequencies.
2. **Frequency-Domain (Magnitude Spectrum):**
   * The second plot shows the **magnitude spectrum** of the signal. The DFT reveals the frequency components at 50 Hz and 150 Hz, which are the two primary components of the input signal.



**Experiment No:** 08

**Experiment Name:**

Explain and Implement Frequency bin using python

O**bjectives:**

* To understand how frequency bins are derived from the DFT.
* To visualize the frequency components of a signal using frequency bins.

**Theory:**

In signal processing, the **frequency bin** is a specific frequency component obtained by transforming a time-domain signal into the frequency domain using the Discrete Fourier Transform (DFT). The DFT divides the total frequency range into smaller intervals (bins), each corresponding to a specific frequency component.

Each bin represents a frequency fkf\_kfk​, where kkk is the index of the frequency bin and is calculated as:

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

Where:

* fsf\_sfs​ is the sampling frequency.
* NNN is the number of samples.

The frequency bins provide insight into the dominant frequency components present in the signal.

**Steps to Implement Frequency Bins:**

1. **Create an Input Signal:** A combination of sinusoids at different frequencies.
2. **Apply DFT to Compute Frequency Components:** Use the DFT formula to extract frequency components.
3. **Plot the Frequency Bins:** Visualize the magnitude of each frequency component.

**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

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

# --- 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[:sampling\_rate//2], magnitude[:sampling\_rate//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 signal composed of two sinusoids with frequencies f1=50f\_1 = 50f1​=50 Hz and f2=150f\_2 = 150f2​=150 Hz, sampled at a rate of 1000 Hz for 1 second.
* **Output:**
  + **Time-domain Signal:**  
    The plot shows the combined signal of two sinusoids at different frequencies.
  + **Frequency Bins (Magnitude Spectrum):**  
    The plot shows the magnitude spectrum with frequency bins indicating the presence of two frequency components at 50 Hz and 150 Hz.

**Explanation of Output:**

1. **Time-Domain Signal:**
   * The first plot depicts the original signal composed of two sinusoids at 50 Hz and 150 Hz.
2. **Frequency Bins (Magnitude Spectrum):**
   * The second plot shows the **frequency bins**, where we observe two peaks corresponding to the frequencies 50 Hz and 150 Hz. These bins represent the frequency components of the input signal.

