

|  |  |
| --- | --- |
| INDEX | TITLE |
|  | Plot the following signal operations using user defined function-   1. adding , b. multiplication, c. Scaling, d. shifting and e. folding |
| 2. | Plot the following transformation x1(n) = 2x(n-5) - 3x(n+4). |
| 3. | Explain and Implementation the unit Impulse sequence, the unit Step sequence, the unit Ramp sequence. |
| 4. | Explain and Implement convolution of signal. |
| 5. | Explain and Implement correlation of signal. |
| 6. | Extract relevant features such as filtering, feature extraction, pick detection, heart rate etc. from PPG signal. |
| 7. | Explain and Implement Discrete Fourier Transform (DFT) using python. |
| 8. | Explain and Implement Frequency bin using python. |

**Lab-01:**

**Title: Plot the signal operations using user defined functions**.

**Theory:**

Signal processing is a fundamental aspect of engineering and science that involves manipulating signals for various applications such as communications, audio processing, and control systems. Basic operations on signals include addition, multiplication, shifting, folding, and scaling. These operations allow for the analysis and modification of signals to meet specific requirements.

**2.1 Signal Addition**

Addition of two signals results in a new signal where each sample is the sum of corresponding samples of the original signals.

Mathematically,

**2.2 Signal Multiplication**

Multiplication of two signals results in a new signal where each sample is the product of corresponding samples of the original signals.

**2.3 Signal Shifting**

Shifting involves moving a signal forward (delay) or backward (advance) in time.

* Right shift (delay):
* Left shift (advance):

**2.4 Signal Folding (Reversal)**

Folding or time-reversal of a signal involves flipping it around the vertical axis.

Mathematically,

**2.5 Signal Scaling**

Scaling modifies the amplitude of the signal by multiplying it with a constant factor.

Mathematically, where is the scaling factor.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

def generate\_signal(n, values):

    return np.array(values), np.array(n)

def add\_signals(x1, x2):

    return x1 + x2

def multiply\_signals(x1, x2):

    return x1 \* x2

def shift\_signal(x, n, k):

    return x, n + k

def fold\_signal(x, n):

    return np.flip(x), -np.flip(n)

def scale\_signal(x, a):

    return a \* x

# Example signals

n = np.arange(-5, 6)

x1 = np.sin(n)

x2 = np.cos(n)

# Operations

sum\_signal = add\_signals(x1, x2)

mul\_signal = multiply\_signals(x1, x2)

shifted\_signal, shifted\_n = shift\_signal(x1, n, 2)

folded\_signal, folded\_n = fold\_signal(x1, n)

scaled\_signal = scale\_signal(x1, 2)

# Plot results

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

plt.subplot(3,2,1)

plt.stem(n, x1)

plt.title("Original Signal x1")

plt.subplot(3,2,2)

plt.stem(n, x2)

plt.title("Original Signal x2")

plt.subplot(3,2,3)

plt.stem(n, sum\_signal)

plt.title("Addition of Signals")

plt.subplot(3,2,4)

plt.stem(n, mul\_signal)

plt.title("Multiplication of Signals")

plt.subplot(3,2,5)

plt.stem(shifted\_n, shifted\_signal)

plt.title("Shifted Signal")

plt.subplot(3,2,6)

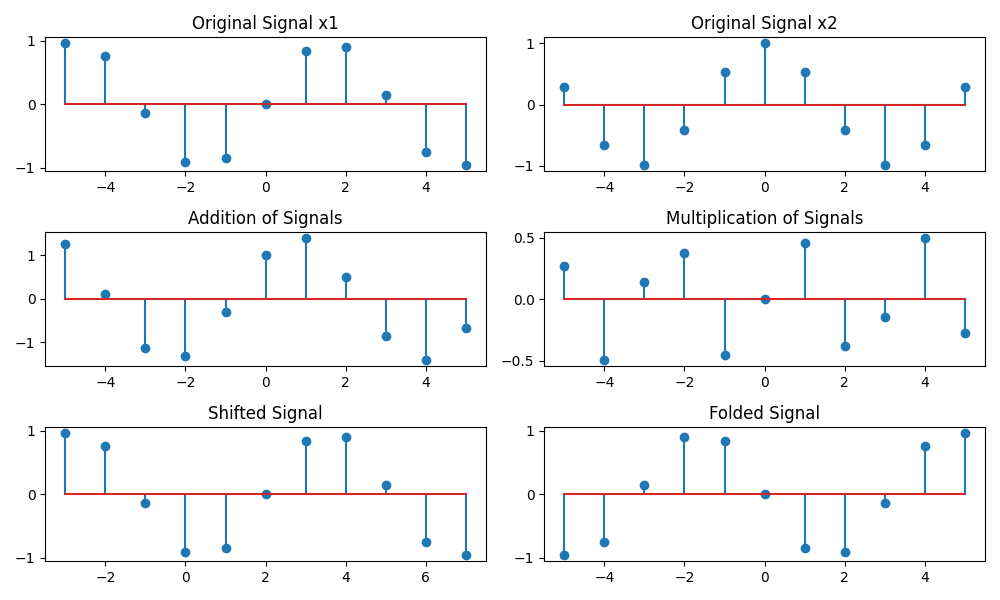
plt.stem(folded\_n, folded\_signal)

plt.title("Folded Signal")

plt.tight\_layout()

plt.show()

**Output:**

****

**Fig-01: basic signal operations**

**Lab-02:**

**Title: Signal Transformation and Plotting of x1(n) = 2x(n-5) - 3x(n+4).**

**Theory:**

A discrete-time signal x(n) can be transformed using operations such as scaling and shifting. The given transformation is:

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

* The term x(n-5) represents a right shift of 5 samples.
* The term x(n+4) represents a left shift of 4 samples.
* The coefficients 2 and -3 indicate amplitude scaling of the shifted signals.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

# Define the original signal x(n)

def x(n):

    return np.where((n >= -5) & (n <= 5), 1, 0)  # Example: rectangular pulse

# Define the range of n values

n = np.arange(-10, 11)

original\_signal = x(n)

# Perform the transformations

x\_shifted\_right = x(n - 5)  # Shift x(n) by +5 (right)

x\_shifted\_left = x(n + 4)   # Shift x(n) by -4 (left)

transformed\_signal = 2 \* x\_shifted\_right - 3 \* x\_shifted\_left

# Plot the signals

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

# Plot original signal

plt.subplot(3, 1, 1)

plt.stem(n, original\_signal, basefmt="b", linefmt='b', markerfmt='bo')

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

plt.xlabel("n")

plt.ylabel("x(n)")

plt.grid()

# Plot transformed signal

plt.subplot(3, 1, 2)

plt.stem(n, 2 \* x\_shifted\_right, basefmt="g", linefmt='g', markerfmt='go')

plt.title("2 \* x(n-5)")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

plt.subplot(3, 1, 3)

plt.stem(n, transformed\_signal, basefmt="r", linefmt='r', markerfmt='ro')

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

plt.xlabel("n")

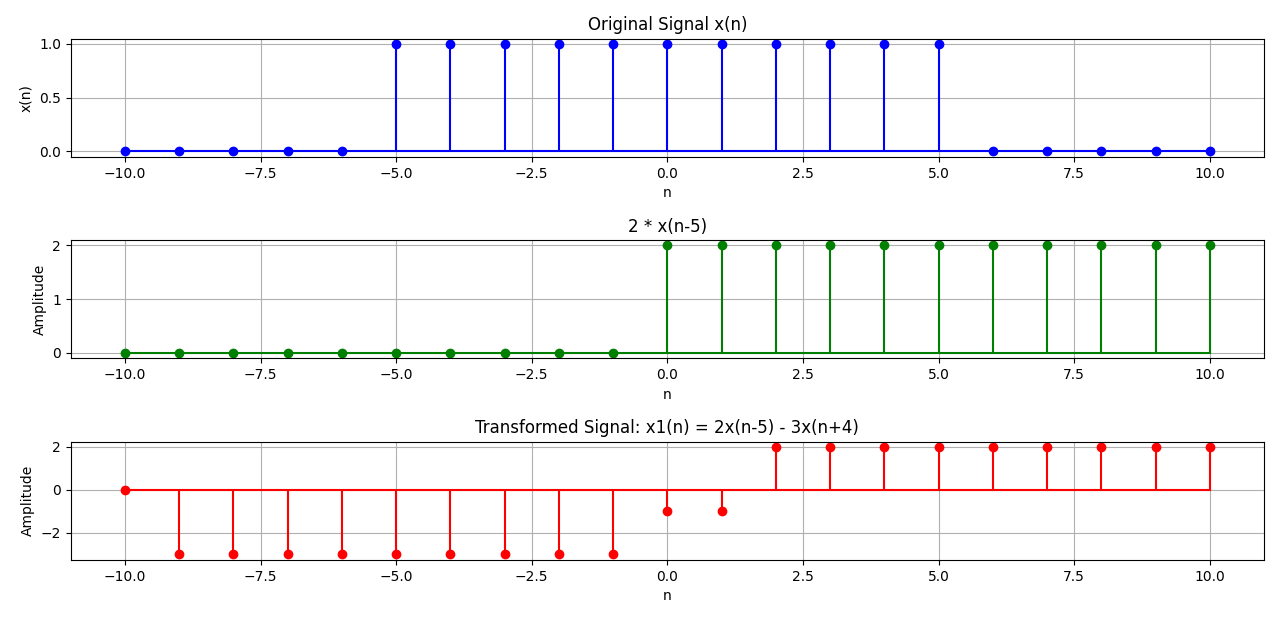
plt.ylabel("Amplitude")

plt.grid()

plt.tight\_layout()

plt.show()

**Output:**

****

**Fig-02: plotting the signal transformation**

**Lab- 03:**

**Title: Explanation and Implementation of Unit Impulse, Unit Step and Unit Ramp Sequences.**

**Theory:**

1. **Unit Impulse Sequence ()**
   * The unit impulse sequence, also known as the Dirac delta function in discrete time, is defined as:
   * It serves as an identity element in convolution and is used to analyze system responses.
2. **Unit Step Sequence ()**
   * The unit step sequence is defined as:
   * It is used to model system inputs that start at a specific point and is fundamental in system analysis.
3. **Unit Ramp Sequence ()**
   * The unit ramp sequence is a discrete-time version of the continuous-time ramp function and is defined as:
   * It represents a signal that linearly increases over time.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

def unit\_impulse(n):

    return np.where(n == 0, 1, 0)

def unit\_step(n):

    return np.where(n >= 0, 1, 0)

def unit\_ramp(n):

    return np.where(n >= 0, n, 0)

# Define range of n values

n = np.arange(-10, 11)

# Generate sequences

impulse = unit\_impulse(n)

step = unit\_step(n)

ramp = unit\_ramp(n)

# Plot the sequences

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

# Unit Impulse Sequence

plt.subplot(3, 1, 1)

plt.stem(n, impulse)

plt.title("Unit Impulse Sequence")

plt.xlabel("n")

plt.ylabel("δ(n)")

plt.grid()

# Unit Step Sequence

plt.subplot(3, 1, 2)

plt.stem(n, step)

plt.title("Unit Step Sequence")

plt.xlabel("n")

plt.ylabel("u(n)")

plt.grid()

# Unit Ramp Sequence

plt.subplot(3, 1, 3)

plt.stem(n, ramp)

plt.title("Unit Ramp Sequence")

plt.xlabel("n")

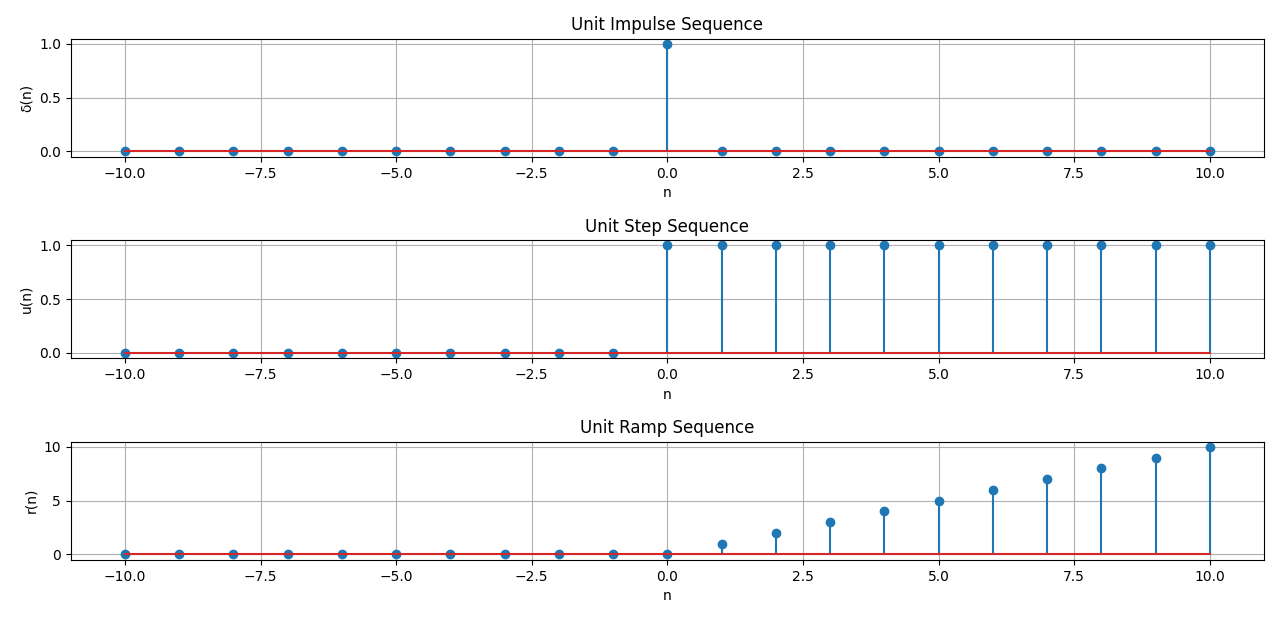
plt.ylabel("r(n)")

plt.grid()

plt.tight\_layout()

plt.show()

**Output:**

****

**Lab-04:**

**Title: Explain and Implement convolution of signal.**

**Theory:**

1. **Definition of Convolution:**
   * Convolution is a mathematical operation used to express the relation between the input signal, the system’s impulse response, and the output signal.
   * The discrete-time convolution sum is given by:
   * Here, is the input signal, is the impulse response, and is the output signal.
2. **Properties of Convolution:**
   * **Commutative Property:**
   * **Associative Property:**
   * **Distributive Property:**
3. **Applications of Convolution:**
   * Convolution is used to analyze LTI systems.
   * It is widely applied in filtering, image processing, and audio signal processing.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

def discrete\_convolution(x, h):

    """Computes the convolution of two discrete-time signals."""

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

    return y

# Define the input signal x(n) and impulse response h(n)

n\_x = np.arange(0, 5)

n\_h = np.arange(0, 4)

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

h = np.array([1, -1, 2, 1])  # Example impulse response

# Perform convolution

y = discrete\_convolution(x, h)

n\_y = np.arange(0, len(y))  # Define the time axis for output

# Plot signals

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

# Input Signal

plt.subplot(3, 1, 1)

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

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

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

# Impulse Response

plt.subplot(3, 1, 2)

plt.stem(n\_h, h, basefmt="g", linefmt='g', markerfmt='go')

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

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

# Output Signal (Convolution Result)

plt.subplot(3, 1, 3)

plt.stem(n\_y, y, basefmt="r", linefmt='r', markerfmt='ro')

plt.title("Output Signal y(n) = x(n) \* h(n)")

plt.xlabel("n")

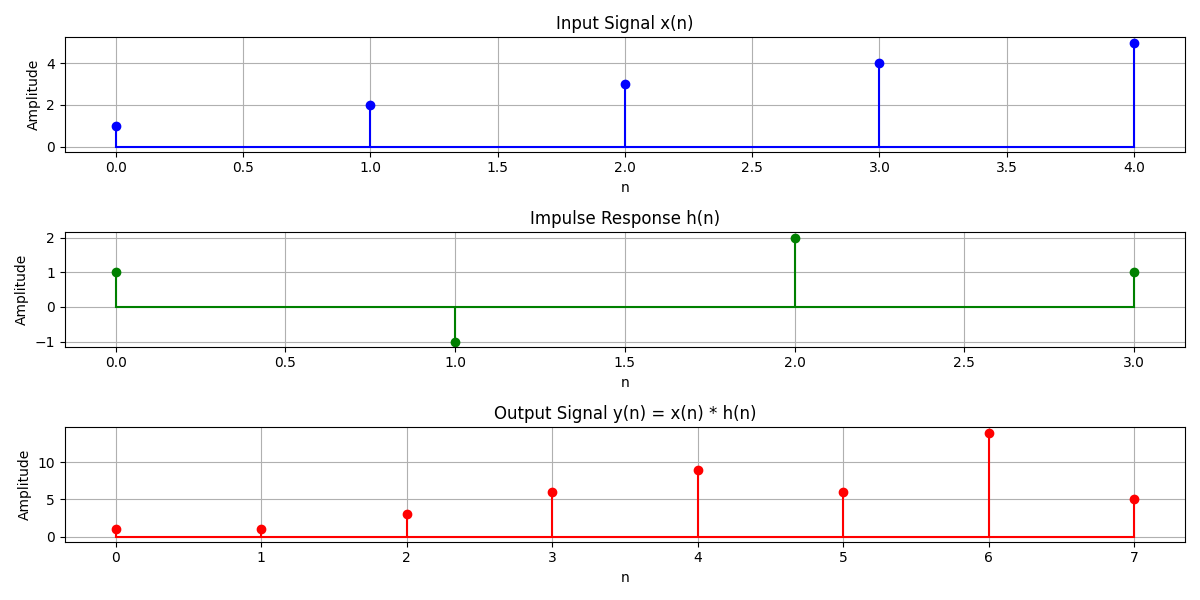
plt.ylabel("Amplitude")

plt.grid()

plt.tight\_layout()

plt.show()

**Output:**

****

**Lab-05:**

**Title: Explanation and Implementation of Correlation in Discrete-Time Signal Processing.**

**Theory:**

1. **Cross-correlation** is used when analyzing the relationship between two signals or time series. It measures how one time series matches with another when one is shifted by some time.

Formula for Cross-correlation:

Rxy​(τ)=t∑​x(t)⋅y(t+τ)

Where τ\tau τ is the time lag, x(t) and y(t) are the two signals or time series.

1. **Auto-correlation** measures the similarity of a signal with a delayed version of itself. It helps identify repeating patterns, periodic signals, or noise.

Formula for Auto-correlation:

Rxx​(τ)=t∑​x(t)⋅x(t+τ)

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

def discrete\_correlation(x, y):

    """Computes the correlation of two discrete-time signals."""

    r\_xy = np.correlate(x, y, mode='full')

    return r\_xy

# Define the signals

n = np.arange(0, 5)

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

y = np.array([2, 1, -1, 3, 2])  # Example signal 2

# Perform correlation

r\_xy = discrete\_correlation(x, y)

lags = np.arange(-len(x) + 1, len(x))

# Plot signals

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

# First Signal

plt.subplot(3, 1, 1)

plt.stem(n, x, basefmt="b", linefmt='b', markerfmt='bo')

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

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

# Second Signal

plt.subplot(3, 1, 2)

plt.stem(n, y, basefmt="g", linefmt='g', markerfmt='go')

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

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

# Correlation Output

plt.subplot(3, 1, 3)

plt.stem(lags, r\_xy, basefmt="r", linefmt='r', markerfmt='ro')

plt.title("Correlation Output r\_{xy}(m)")

plt.xlabel("Lag (m)")

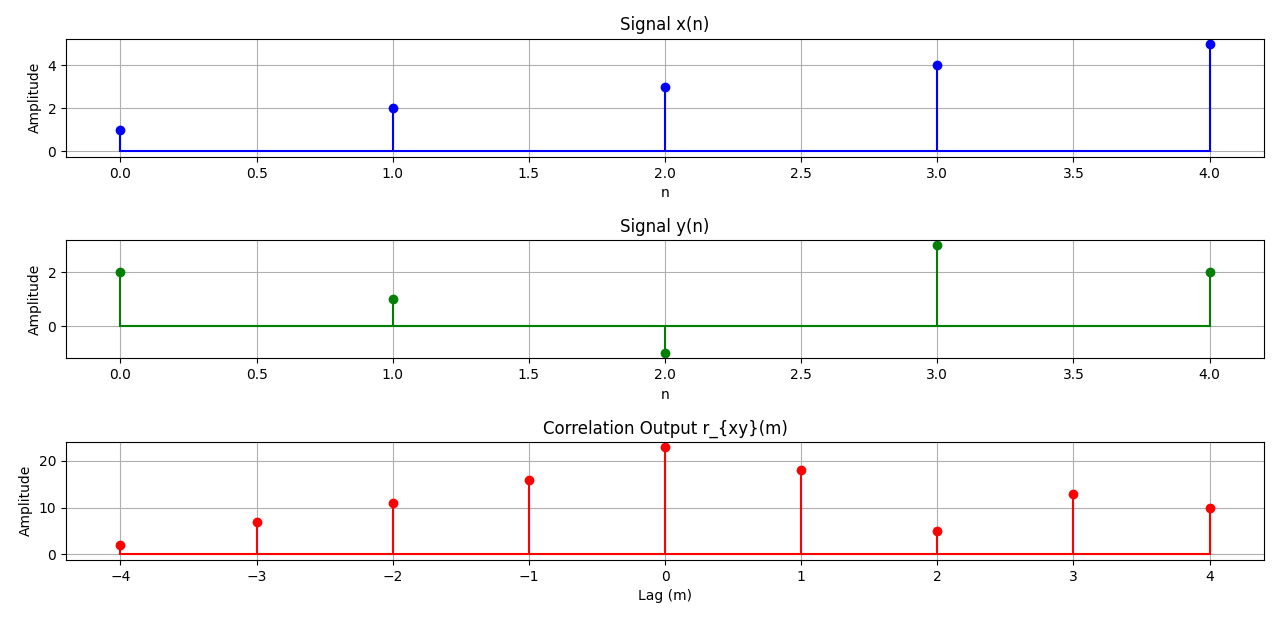
plt.ylabel("Amplitude")

plt.grid()

plt.tight\_layout()

plt.show()

**Output:**

****

**Lab-06:  
Title: Extract relevant features such as filtering, feature extraction, pick detection, heart rate etc. from PPG signal.**

**Theory:**

**1. PPG Signal Basics**

**A Photoplethysmogram (PPG) is an optical technique used to detect blood volume changes in the skin. It consists of:**

* Systolic Peaks (higher values, indicating heartbeat)
* Diastolic Troughs (lower values between beats**)**

**PPG signals are prone to noise from:**

* Motion artifacts
* Power-line interference
* Respiration effects

**2. Noise in Biomedical Signals**

* Gaussian Noise (random fluctuations in the signal)
* Baseline Drift (slow variations in the signal)
* Powerline Interference (50/60 Hz electrical noise**)**

**3. Filtering Techniques**

* Low-pass filters remove high-frequency noise.
* High-pass filters remove baseline drift.
* Band-pass filters retain frequencies within a specific range**.**

**4. Feature Extraction & Peak Detection**

* Find systolic peaks using algorithms like find\_peaks() from scipy.signal.
* Compute heart rate (BPM) from peak intervals.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

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

# Generate a synthetic PPG signal (sine wave with peaks)

fs = 100  # Sampling frequency (Hz)

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

ppg\_clean = 0.6 \* np.sin(2 \* np.pi \* 1.2 \* t) + 1  # Simulated PPG (1.2 Hz ≈ 72 BPM)

# Add noise (Gaussian noise + baseline drift)

noise = 0.2 \* np.random.randn(len(t))  # Random noise

baseline\_drift = 0.5 \* np.sin(2 \* np.pi \* 0.1 \* t)  # Low-frequency drift

ppg\_noisy = ppg\_clean + noise + baseline\_drift  # Noisy PPG

# Bandpass Filter (0.5–5 Hz to remove drift & high-freq noise)

def bandpass\_filter(signal, lowcut=0.5, highcut=5, fs=100, order=4):

    nyquist = 0.5 \* fs

    low = lowcut / nyquist

    high = highcut / nyquist

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

    return filtfilt(b, a, signal)

ppg\_filtered = bandpass\_filter(ppg\_noisy)

# Peak Detection (detect systolic peaks)

peaks, \_ = find\_peaks(ppg\_filtered, distance=fs//2, height=0.5)

# Compute Heart Rate (BPM)

peak\_intervals = np.diff(t[peaks])  # Time differences between peaks

bpm = 60 / np.mean(peak\_intervals)  # Beats per minute

# Plot results

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

plt.subplot(3, 1, 1)

plt.plot(t, ppg\_clean, label="Clean PPG", color='green')

plt.title("Clean PPG Signal")

plt.legend()

plt.subplot(3, 1, 2)

plt.plot(t, ppg\_noisy, label="Noisy PPG", color='red')

plt.title("Noisy PPG Signal with Noise & Baseline Drift")

plt.legend()

plt.subplot(3, 1, 3)

plt.plot(t, ppg\_filtered, label="Filtered PPG", color='blue')

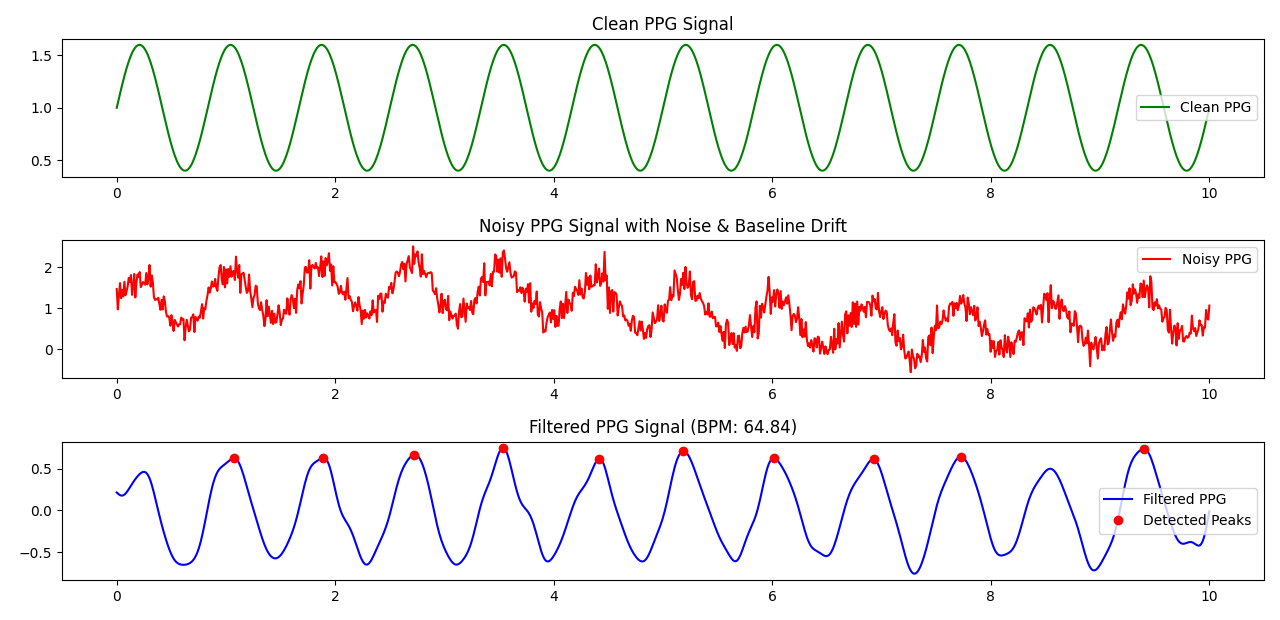
plt.plot(t[peaks], ppg\_filtered[peaks], "ro", label="Detected Peaks")

plt.title(f"Filtered PPG Signal (BPM: {bpm:.2f})")

plt.legend()

plt.tight\_layout()

plt.show()

**Output: **

**Lab 07:**

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

**Theory:**

The **Discrete Fourier Transform (DFT)** is a mathematical technique used to analyze the frequency content of discrete signals. It transforms a sequence of complex numbers from the time domain into the frequency domain, revealing the amplitudes and phases of the constituent sinusoidal components.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

def DFT(x):

    """

    Compute the Discrete Fourier Transform of the 1D array x.

    """

    x = np.asarray(x, dtype=float)

    N = x.shape[0]

    n = np.arange(N)

    k = n.reshape((N, 1))

    exponent = -2j \* np.pi \* k \* n / N

    W = np.exp(exponent)

    return np.dot(W, x)

# Example usage:

# Define a sample signal: a sum of two sine waves

sampling\_rate = 1000  # Hz

T = 1.0 / sampling\_rate  # Sampling interval

t = np.linspace(0.0, 1.0, sampling\_rate, endpoint=False)

freq1 = 50  # Frequency of the first sine wave

freq2 = 120  # Frequency of the second sine wave

amplitude1 = 1.0

amplitude2 = 0.5

signal = amplitude1 \* np.sin(2 \* np.pi \* freq1 \* t) + amplitude2 \* np.sin(2 \* np.pi \* freq2 \* t)

# Compute the DFT of the signal

dft\_result = DFT(signal)

# Compute the frequencies corresponding to the DFT coefficients

N = len(signal)

frequencies = np.fft.fftfreq(N, T)

# Plot the original signal

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

plt.subplot(2, 1, 1)

plt.plot(t, signal)

plt.title('Time Domain Signal')

plt.xlabel('Time [s]')

plt.ylabel('Amplitude')

# Plot the magnitude spectrum

plt.subplot(2, 1, 2)

plt.stem(frequencies[:N // 2], np.abs(dft\_result)[:N // 2] \* 2 / N, 'b', markerfmt=" ", basefmt="-b")

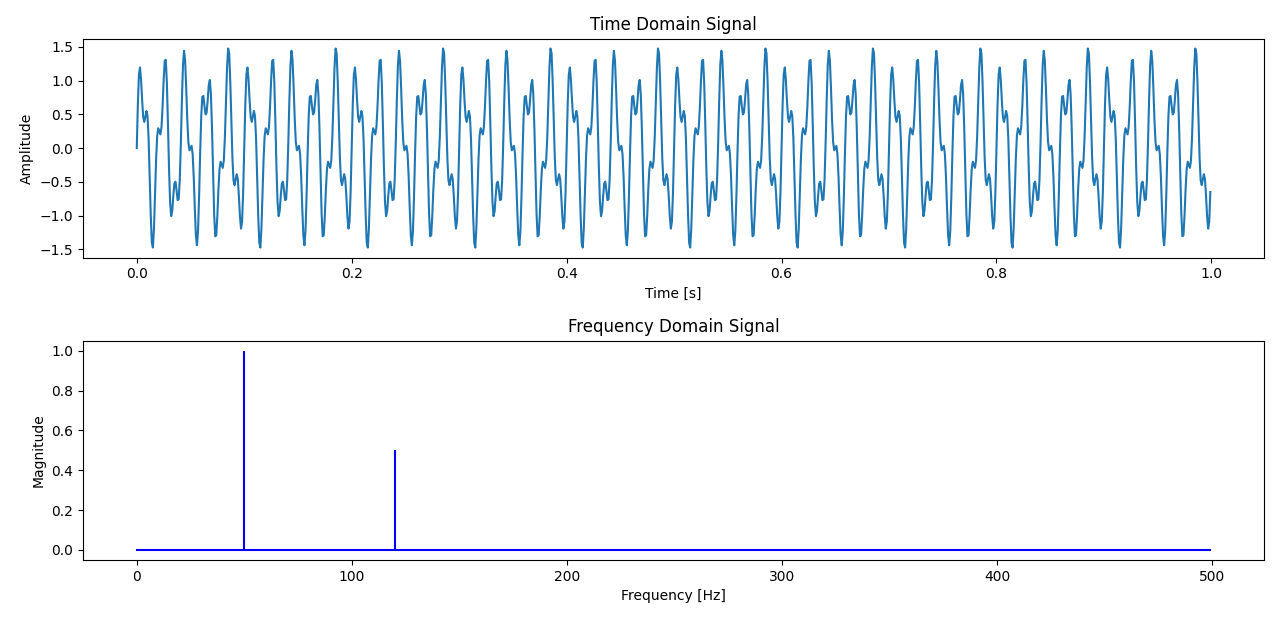
plt.title('Frequency Domain Signal')

plt.xlabel('Frequency [Hz]')

plt.ylabel('Magnitude')

plt.tight\_layout()

plt.show()

**Output:  
**

**Lab-08:**

**Title:** Explain and Implement Frequency bin using python.

**Theory:**

The **Discrete Fourier Transform (DFT)** is a mathematical technique used in digital signal processing to analyze the frequency content of discrete signals. It transforms a finite sequence of equally spaced samples of a function into a sequence of complex numbers representing the function's frequency components.

1. **Linearity:** The DFT is a linear transform, meaning that the DFT of a sum of signals is equal to the sum of their DFTs.

2. **Periodicity:** The DFT assumes that the input signal is periodic. Consequently, both the input signal and its DFT are periodic with period NNN.

3. **Symmetry:** For real-valued input signals, the DFT exhibits conjugate symmetry, which can be utilized to reduce computational complexity.

4. **Convolution Theorem:** The DFT transforms convolution in the time domain into multiplication in the frequency domain, simplifying the analysis and implementation of linear time-invariant systems.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

# Parameters

sampling\_rate = 1000  # Sampling rate in Hz

T = 1.0  # Duration of the signal in seconds

N = int(T \* sampling\_rate)  # Total number of samples

# Generate a sample signal: a sum of two sine waves

t = np.linspace(0.0, T, N, endpoint=False)

freq1 = 50.0  # Frequency of the first sine wave in Hz

freq2 = 120.0  # Frequency of the second sine wave in Hz

signal = 0.7 \* np.sin(2.0 \* np.pi \* freq1 \* t) + 0.3 \* np.sin(2.0 \* np.pi \* freq2 \* t)

# Compute the FFT

fft\_values = np.fft.fft(signal)

# Compute the frequency bins

freq\_bins = np.fft.fftfreq(N, d=1/sampling\_rate)

# Plot the signal

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

plt.subplot(2, 1, 1)

plt.plot(t, signal)

plt.title('Time Domain Signal')

plt.xlabel('Time [s]')

plt.ylabel('Amplitude')

# Plot the magnitude spectrum

plt.subplot(2, 1, 2)

plt.stem(freq\_bins[:N // 2], np.abs(fft\_values)[:N // 2] \* 2 / N, 'b', markerfmt=" ", basefmt="-b")

plt.title('Frequency Domain Signal')

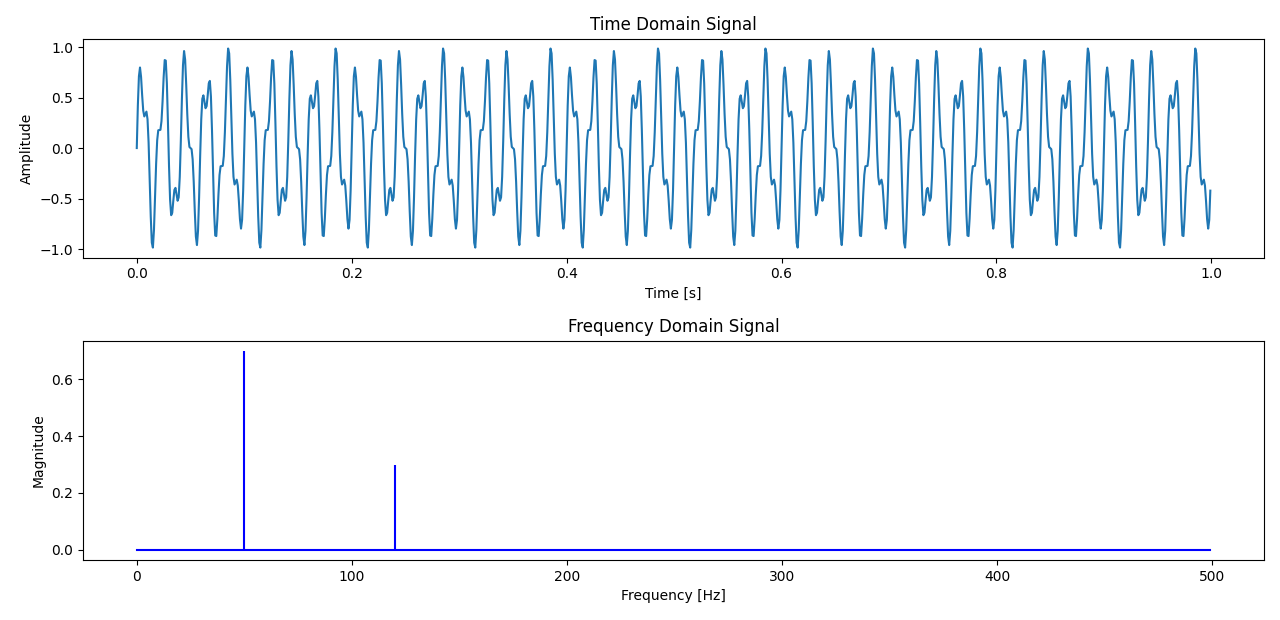
plt.xlabel('Frequency [Hz]')

plt.ylabel('Magnitude')

plt.tight\_layout()

plt.show()

**Output:**

****

**Fig-** Frequency bin using python.