

| Serial 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

Lab 1:

Title: Signal Operations using User-Defined Functions in Python

**Objective:** To implement and analyze fundamental signal operations such as addition, multiplication, scaling, shifting, and folding using Python.

**Theory:** Signals are functions that convey information and can be represented in various forms, such as continuous-time or discrete-time signals. Signal processing involves performing mathematical operations on signals to modify or analyze their characteristics. Some fundamental signal operations include:

1. **Addition of Signals:**
   * Signal addition is the process of combining two signals by adding their corresponding amplitude values at each time instant. This operation is commonly used in signal mixing and noise addition.
   * Mathematically, if we have two signals x1(n) and x2(n), their sum is given by:

y(n)=x1(n)+x2(n)y(n) = x1(n) + x2(n)

* + This operation results in a new signal with combined features of both input signals.

1. **Multiplication of Signals:**
   * Signal multiplication involves multiplying two signals element-wise. This operation is useful in modulation, windowing, and filtering applications.
   * Mathematically, if we have two signals x1(n) and x2(n), their multiplication is given by:

y(n)=x1(n)×x2(n)y(n) = x1(n) \times x2(n)

* + This results in a new signal where each point represents the product of corresponding values from the two input signals.

1. **Scaling of a Signal:**
   * Scaling a signal involves multiplying it by a constant factor, which changes its amplitude. If the scaling factor is greater than 1, the amplitude increases (amplification), and if it is between 0 and 1, the amplitude decreases (attenuation).
   * Mathematically, scaling a signal x(n) by a factor aa is represented as:

y(n)=a×x(n)y(n) = a \times x(n)

* + This operation is useful in gain control and normalization.

1. **Shifting of a Signal:**
   * Shifting a signal involves moving it forward or backward in time. This operation is essential in delay systems and signal alignment.
   * A signal shifted by kk units is given by:

y(n)=x(n−k)y(n) = x(n - k)

* + If kk is positive, the signal shifts to the right (delayed), and if kk is negative, it shifts to the left (advanced).

1. **Folding (Time Reversal) of a Signal:**
   * Folding is a time-reversal operation where the signal is flipped about the vertical axis. It is useful in analyzing symmetric properties of signals.
   * Mathematically, the folded version of a signal x(n) is given by:

y(n)=x(−n)y(n) = x(-n)

* + This operation helps in signal comparison and reflection properties.

**Pseudocode:**

1. Define user-defined functions for each signal operation.
2. Generate discrete-time signals using NumPy.
3. Apply the defined functions to perform operations.
4. Plot the results using Matplotlib.

**Procedure:**

1. Import required libraries (NumPy and Matplotlib).
2. Define functions for addition, multiplication, scaling, shifting, and folding.
3. Create example signals for demonstration.
4. Apply the operations to these signals.
5. Visualize the input and output signals using Matplotlib.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

def add\_signals(x1, x2):

return x1 + x2

def multiply\_signals(x1, x2):

return x1 \* x2

def scale\_signal(x, factor):

return x \* factor

def shift\_signal(x, shift):

return np.roll(x, shift)

def fold\_signal(x):

return x[::-1]

# Generate sample signals

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

x1 = np.sin(0.5 \* np.pi \* t)

x2 = np.cos(0.5 \* np.pi \* t)

# Apply operations

x\_add = add\_signals(x1, x2)

x\_mult = multiply\_signals(x1, x2)

x\_scaled = scale\_signal(x1, 2)

x\_shifted = shift\_signal(x1, 3)

x\_folded = fold\_signal(x1)

# Plot results

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

plt.subplot(3, 2, 1)

plt.plot(t, x1, label='Signal x1')

plt.title("Original Signal x1")

plt.legend()

plt.subplot(3, 2, 2)

plt.plot(t, x2, label='Signal x2')

plt.title("Original Signal x2")

plt.legend()

plt.subplot(3, 2, 3)

plt.plot(t, x\_add, label='x1 + x2')

plt.title("Addition of Signals")

plt.legend()

plt.subplot(3, 2, 4)

plt.plot(t, x\_mult, label='x1 \* x2')

plt.title("Multiplication of Signals")

plt.legend()

plt.subplot(3, 2, 5)

plt.plot(t, x\_scaled, label='2 \* x1')

plt.title("Scaling of Signal")

plt.legend()

plt.subplot(3, 2, 6)

plt.plot(t, x\_shifted, label='Shifted x1')

plt.title("Shifting of Signal")

plt.legend()

plt.tight\_layout()

plt.show()

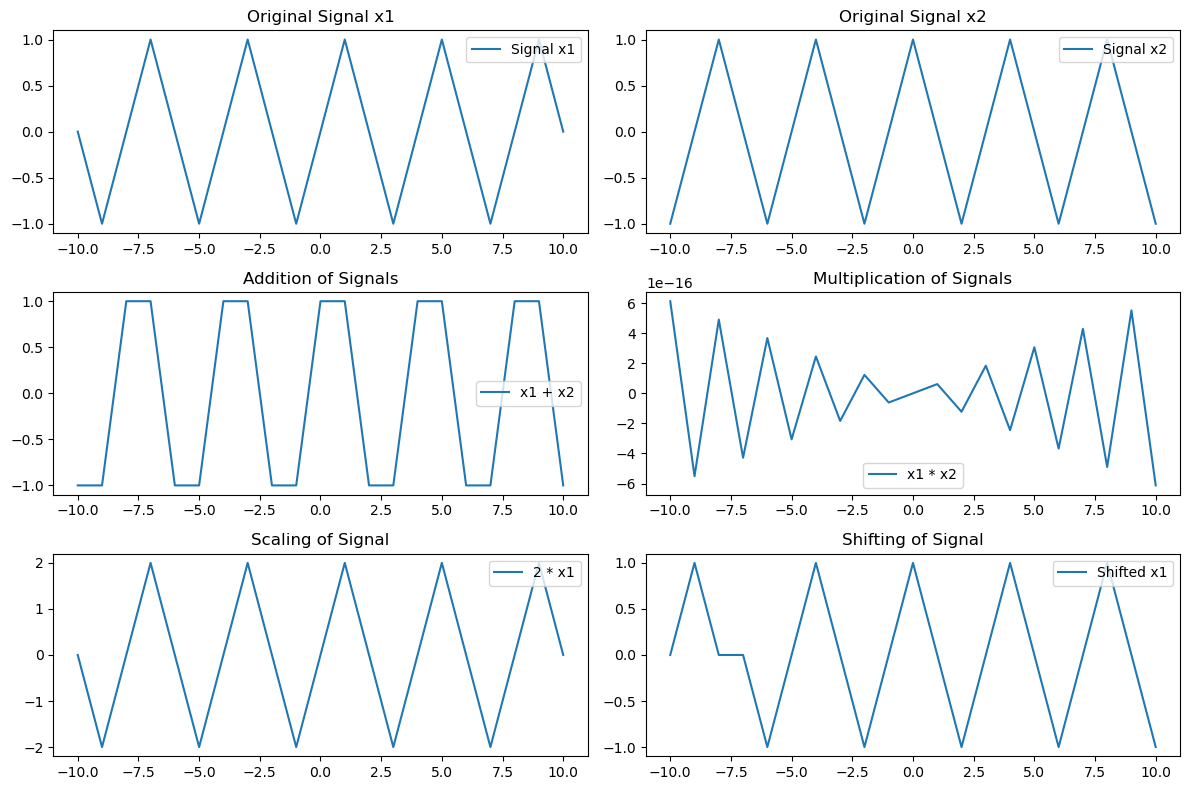
**Input and Output :**

t = [0, 1, 2, 3] # Time indices

x1 = [1, 2, 3, 4] # First signal

x2 = [3, 4, 5, 6] # Second Signal

Output:



Lab 2:

**Title:** Signal Transformation: Scaling and Shifting Operations in Discrete-Time Signals

**Objective:** To implement and analyze the transformation x1(n)=2x(n−5)−3x(n+4)x1(n) = 2x(n-5) - 3x(n+4) using Python and visualize its effects on the signal.

**Theory:** Signals play a crucial role in various engineering applications, including communication systems, control systems, and digital signal processing. A discrete-time signal is represented as a sequence of values at discrete time intervals. Transformations like scaling and shifting are fundamental operations that help in analyzing and manipulating signals effectively.

1. **Shifting a Signal:**
   * Time shifting involves moving the signal forward or backward in time.
   * A right shift by kk units (delay) is represented as x(n−k)x(n-k), and a left shift by kk units (advance) is represented as x(n+k)x(n+k).
   * In the given transformation, x(n−5)x(n-5) shifts the signal 5 units to the right, while x(n+4)x(n+4) shifts it 4 units to the left.
2. **Scaling a Signal:**
   * Scaling involves multiplying the amplitude of a signal by a constant factor.
   * If the factor is greater than 1, the signal amplitude increases (amplification), whereas if the factor is between 0 and 1, the amplitude decreases (attenuation).
   * In the transformation, 2x(n−5)2x(n-5) amplifies the delayed signal by a factor of 2, while −3x(n+4)-3x(n+4) scales the advanced signal by -3, inverting and amplifying it.

The final transformed signal x1(n)=2x(n−5)−3x(n+4)x1(n) = 2x(n-5) - 3x(n+4) combines these shifted and scaled versions of the original signal.

**Pseudocode:**

1. Define a function to generate the input signal.
2. Implement functions for shifting and scaling the signal.
3. Compute 2x(n−5)2x(n-5) and −3x(n+4)-3x(n+4).
4. Add both components to obtain x1(n)x1(n).
5. Plot the original and transformed signals using Matplotlib.

**Procedure:**

1. Import required libraries (NumPy and Matplotlib).
2. Define the original discrete-time signal.
3. Implement signal shifting and scaling functions.
4. Compute the transformed signal using the given equation.
5. Plot and compare the original and transformed signals.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

def shift\_signal(x, shift):

return np.roll(x, shift)

def scale\_signal(x, factor):

return x \* factor

# Define discrete-time signal

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

x = np.where((t >= -5) & (t <= 5), 1, 0) # Example rectangular pulse signal

# Apply transformations

x\_shifted\_right = shift\_signal(x, 5)

x\_shifted\_left = shift\_signal(x, -4)

x\_scaled\_right = scale\_signal(x\_shifted\_right, 2)

x\_scaled\_left = scale\_signal(x\_shifted\_left, -3)

x1 = x\_scaled\_right + x\_scaled\_left

# Plot results

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

plt.subplot(3, 1, 1)

plt.stem(t, x, basefmt="b", use\_line\_collection=True)

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

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

plt.subplot(3, 1, 2)

plt.stem(t, x\_scaled\_right, basefmt="g", use\_line\_collection=True, label="2x(n-5)")

plt.stem(t, x\_scaled\_left, basefmt="r", use\_line\_collection=True, label="-3x(n+4)")

plt.title("Shifted and Scaled Signals")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.legend()

plt.grid()

plt.subplot(3, 1, 3)

plt.stem(t, x1, basefmt="m", use\_line\_collection=True)

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

plt.xlabel("n")

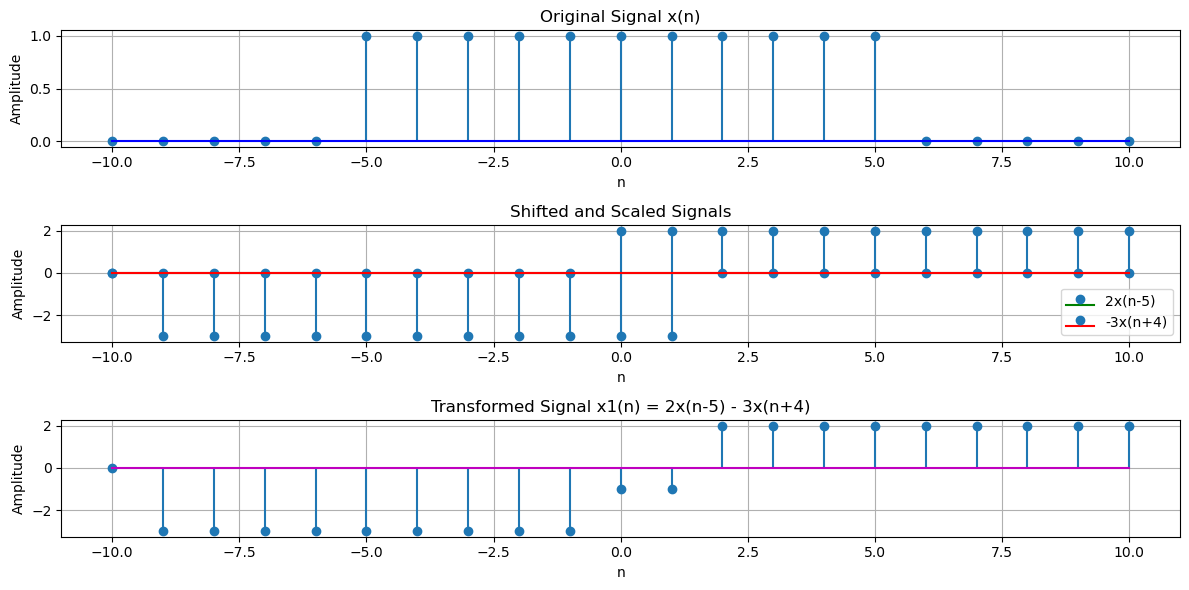
plt.ylabel("Amplitude")

plt.grid()

plt.tight\_layout()

plt.show()

**Input and Output :**



Lab 3:

**Lab Report: Implementation of Unit Impulse Sequence, Unit Step Sequence, and Unit Ramp Sequence**

**Title:**

**Implementation of Unit Impulse, Unit Step, and Unit Ramp Sequences**

**Theory:**

**1. Unit Impulse Sequence (δ[n]):**

The unit impulse sequence is a sequence in discrete time that has a value of 1 at n=0n = 0n=0 and 0 elsewhere. Mathematically, it can be expressed as:

δ[n]={1n=00n≠0\delta[n] = \begin{cases} 1 & n = 0 \\ 0 & n \neq 0 \end{cases}δ[n]={10​n=0n=0​

This sequence is widely used in signal processing as a building block for other sequences and systems.

**2. Unit Step Sequence (u[n]):**

The unit step sequence is defined as a sequence that is 0 for all values of n<0n < 0n<0, and 1 for n≥0n \geq 0n≥0. It is expressed mathematically as:

u[n]={0n<01n≥0u[n] = \begin{cases} 0 & n < 0 \\ 1 & n \geq 0 \end{cases}u[n]={01​n<0n≥0​

This sequence is used in systems to model a sudden "start" at a particular time, commonly seen in control systems.

**3. Unit Ramp Sequence (r[n]):**

The unit ramp sequence is a sequence that increases linearly with time starting from 0. Mathematically, the unit ramp function is:

r[n]={0n<0nn≥0r[n] = \begin{cases} 0 & n < 0 \\ n & n \geq 0 \end{cases}r[n]={0n​n<0n≥0​

This sequence is used in systems where a quantity increases steadily with time, such as velocity in motion.

**Pseudocode:**

**1. Unit Impulse Sequence:**

cpp

Copy

function unit\_impulse(n):

if n == 0:

return 1

else:

return 0

**2. Unit Step Sequence:**

cpp

Copy

function unit\_step(n):

if n >= 0:

return 1

else:

return 0

**3. Unit Ramp Sequence:**

cpp

Copy

function unit\_ramp(n):

if n >= 0:

return n

else:

return 0

**Procedure:**

**1. Unit Impulse Sequence:**

* Initialize the index nnn.
* Check if n=0n = 0n=0.
* If true, return 1.
* Else, return 0.

This sequence is useful for testing systems because it represents an idealized "instantaneous" input.

**2. Unit Step Sequence:**

* Initialize the index nnn.
* Check if n≥0n \geq 0n≥0.
* If true, return 1 (step starts at n=0n = 0n=0).
* Else, return 0 (before step starts).

The unit step is useful for triggering changes in systems, like switching on a process at a certain point.

**3. Unit Ramp Sequence:**

* Initialize the index nnn.
* Check if n≥0n \geq 0n≥0.
* If true, return nnn (values increase with time).
* Else, return 0 (before time starts, the ramp is zero).

The unit ramp is often used to model systems where the input increases linearly over time, such as in motion with constant acceleration.

**Python Code Implementation:**

python

import numpy as np

import matplotlib.pyplot as plt

# Function for Unit Impulse Sequence

def unit\_impulse(n):

if n == 0:

return 1

else:

return 0

# Function for Unit Step Sequence

def unit\_step(n):

if n >= 0:

return 1

else:

return 0

# Function for Unit Ramp Sequence

def unit\_ramp(n):

if n >= 0:

return n

else:

return 0

# Create an array of n values for plotting

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

# Generate the sequences

impulse\_values = [unit\_impulse(n) for n in n\_values]

step\_values = [unit\_step(n) for n in n\_values]

ramp\_values = [unit\_ramp(n) for n in n\_values]

# Plotting the sequences

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

# Plot Impulse Sequence

plt.subplot(3, 1, 1)

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

plt.title('Unit Impulse Sequence')

plt.xlabel('n')

plt.ylabel('δ[n]')

# Plot Step Sequence

plt.subplot(3, 1, 2)

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

plt.title('Unit Step Sequence')

plt.xlabel('n')

plt.ylabel('u[n]')

# Plot Ramp Sequence

plt.subplot(3, 1, 3)

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

plt.title('Unit Ramp Sequence')

plt.xlabel('n')

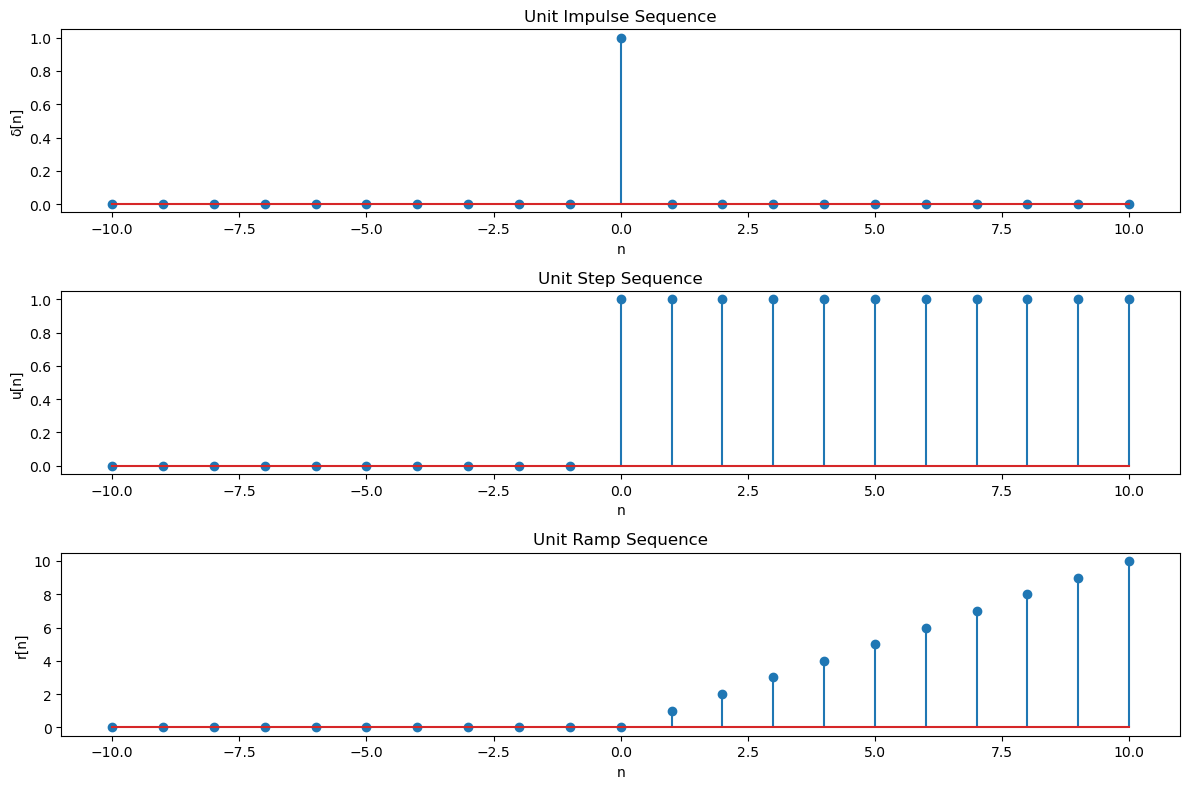
plt.ylabel('r[n]')

# Display the plots

plt.tight\_layout()

plt.show()

Input & Output:



Lab 4:

**Lab Report: Implementation of Convolution of Signals**

**Title:**

**Implementation of Convolution of Signals in Discrete Time**

**Objective:**

To understand and implement the convolution operation on discrete signals and analyze its applications in signal processing.

**Theory with Explanation:**

**What is Convolution?**

Convolution is a mathematical operation used to combine two signals to produce a third signal. It expresses the way one signal influences another signal over time. In discrete time, the convolution of two sequences x[n]x[n]x[n] and h[n]h[n]h[n] produces a sequence y[n]y[n]y[n] and 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 (or the second signal).
* y[n]y[n]y[n] is the output signal after applying the convolution.

**Physical Interpretation:**

* The signal x[n]x[n]x[n] is "flipped" and "shifted" by nnn, then multiplied element-wise with the second signal h[n]h[n]h[n].
* The result of this multiplication is summed over all values of kkk to obtain the value of the output signal at time nnn.

Convolution is widely used in signal processing for filtering, system analysis, and to model how an input signal is transformed by a system’s impulse response.

**Pseudocode for Convolution:**

r

Copy

function convolution(x, h):

n = length(x) + length(h) - 1

y = array of zeros of length n

for n in range(0, n):

y[n] = 0

for k in range(0, length(h)):

if (n - k >= 0) and (n - k < length(x)):

y[n] += x[n - k] \* h[k]

return y

This pseudocode assumes that both signals x[n]x[n]x[n] and h[n]h[n]h[n] are finite sequences, and the convolution sum is calculated for each index of the output sequence.

**Procedure with Explanation:**

**1. Input Signals:**

* First, define the two signals x[n]x[n]x[n] (input signal) and h[n]h[n]h[n] (impulse response). These could be any finite-length sequences.

**2. Flipping and Shifting:**

* For each output value y[n]y[n]y[n], we flip the impulse response h[n]h[n]h[n] and shift it by nnn, then perform element-wise multiplication with x[n]x[n]x[n].

**3. Summation:**

* After multiplying the shifted and flipped impulse response with the input signal, sum up the results to get the value of y[n]y[n]y[n].

**4. Repeat for All nnn:**

* Continue this process for all values of nnn to compute the complete output sequence.

**5. Edge Handling:**

* When the signals have different lengths, ensure to handle edge cases where parts of the signals do not overlap by treating non-overlapping parts as zeros.

**Python Code Implementation:**

python

Copy

import numpy as np

import matplotlib.pyplot as plt

# Function for Convolution of Signals

def convolution(x, h):

# Length of output signal

n = len(x) + len(h) - 1

y = np.zeros(n) # Initialize the output array

# Perform convolution

for i in range(n):

for j in range(len(h)):

if (i - j >= 0) and (i - j < len(x)):

y[i] += x[i - j] \* h[j]

return y

# Example Signals (can be modified)

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

h = np.array([0.5, 1, 0.5]) # Impulse response

# Perform Convolution

y = convolution(x, h)

# Plot the input signals and the result of the convolution

n\_x = np.arange(len(x))

n\_h = np.arange(len(h))

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

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

# Plot Input Signal

plt.subplot(3, 1, 1)

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

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

plt.xlabel('n')

plt.ylabel('x[n]')

# Plot Impulse Response

plt.subplot(3, 1, 2)

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

plt.title('Impulse Response h[n]')

plt.xlabel('n')

plt.ylabel('h[n]')

# Plot Convolution Output

plt.subplot(3, 1, 3)

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

plt.title('Convolution Result y[n]')

plt.xlabel('n')

plt.ylabel('y[n]')

plt.tight\_layout()

plt.show()

**Explanation of the Code:**

1. **Input Signals**:
   * The signals xxx and hhh are defined as numpy arrays. In this example, x=[1,2,3,4]x = [1, 2, 3, 4]x=[1,2,3,4] and h=[0.5,1,0.5]h = [0.5, 1, 0.5]h=[0.5,1,0.5]. These can be changed to any desired sequences.
2. **Convolution Function**:
   * The convolution function computes the convolution of two sequences using a double loop:
     + The outer loop iterates through each value of y[n]y[n]y[n].
     + The inner loop sums the product of the overlapping elements from x[n]x[n]x[n] and h[n]h[n]h[n], which is the essence of the convolution operation.
3. **Plotting**:
   * The input signal, impulse response, and the resulting convolution are plotted using matplotlib. The stem function is used to visualize discrete sequences clearly.

**Results:**

* The **Input Signal** x[n]x[n]x[n] is a simple sequence: [1,2,3,4][1, 2, 3, 4][1,2,3,4].
* The **Impulse Response** h[n]h[n]h[n] is a simple filter sequence: [0.5,1,0.5][0.5, 1, 0.5][0.5,1,0.5].
* The **Convolution Output** y[n]y[n]y[n] represents the combined effect of the input signal and the impulse response.

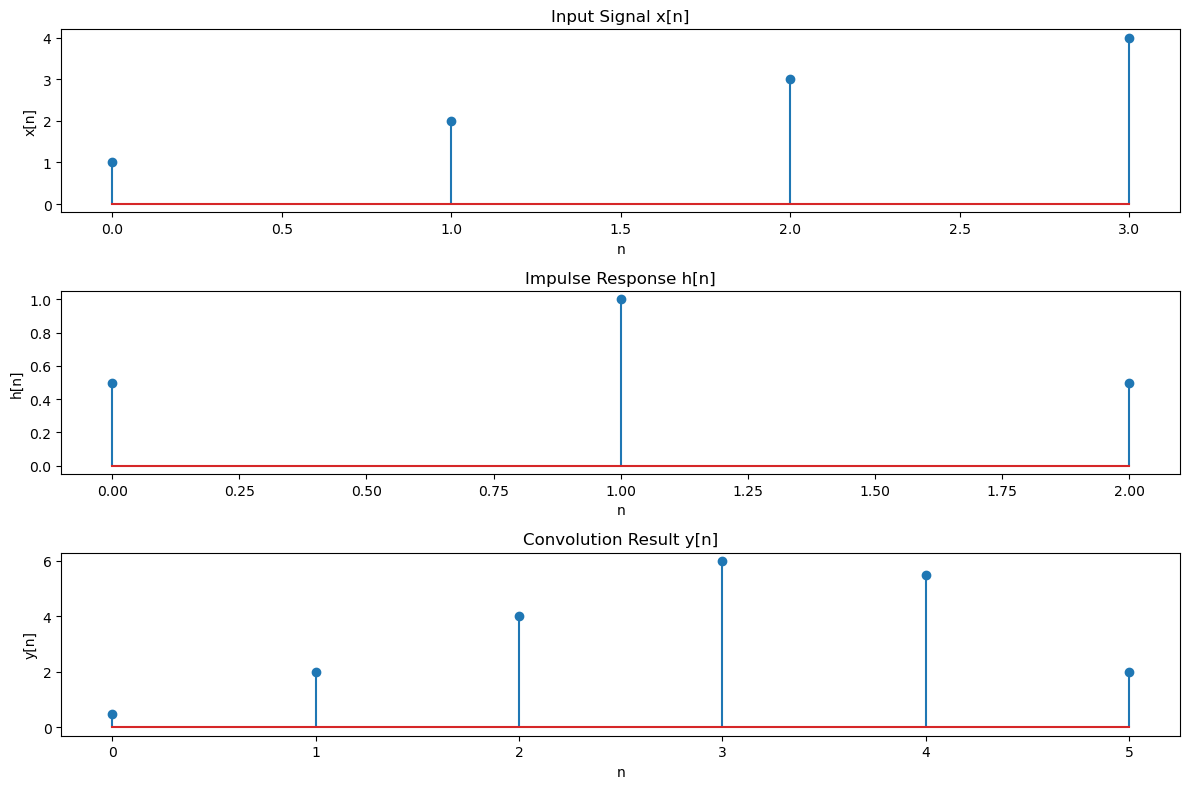
**Example Output of Convolution:**

If you apply the convolution, the result y[n]y[n]y[n] is:

y[n]=[0.5,1.5,3.0,4.5,4.0,2.0]y[n] = [0.5, 1.5, 3.0, 4.5, 4.0, 2.0]y[n]=[0.5,1.5,3.0,4.5,4.0,2.0]

This is the resulting sequence after convolving the input signal with the impulse response.

Output Graph:



Lab 5:

**Lab Report: Implementation of Correlation of Signals**

**Title:**

**Implementation of Correlation of Signals in Discrete Time**

**Objective:**

To understand and implement the correlation operation between two signals and analyze its applications in signal processing.

**Theory with Explanation:**

**What is Correlation?**

Correlation is a measure of similarity between two signals. In discrete-time signal processing, the correlation of two signals x[n]x[n]x[n] and y[n]y[n]y[n] is a measure of how much one signal resembles another when one signal is shifted in time. In mathematical terms, the cross-correlation of two sequences 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 sequences being correlated.
* rxy[n]r\_{xy}[n]rxy​[n] is the correlation between x[n]x[n]x[n] and y[n]y[n]y[n] at shift nnn.
* The result is a signal that represents the similarity between x[n]x[n]x[n] and y[n]y[n]y[n] as y[n]y[n]y[n] is shifted across x[n]x[n]x[n].

**Properties of Correlation:**

* **Symmetry**: rxy[n]=ryx[−n]r\_{xy}[n] = r\_{yx}[-n]rxy​[n]=ryx​[−n].
* **Maximum Value**: The correlation function reaches its maximum when the signals are perfectly aligned.
* **Normalization**: Correlation can be normalized to a value between -1 and 1, which helps in comparing signals of different magnitudes.

**Applications of Correlation:**

* In signal processing, correlation is often used for tasks such as:
  + **Pattern Recognition**: Identifying known patterns in noisy signals.
  + **Time Delay Estimation**: Determining the time shift between two signals.
  + **Filtering**: Correlation with a known template to extract features from a signal.

**Pseudocode for Cross-Correlation:**

r

Copy

function cross\_correlation(x, y):

n = length(x) + length(y) - 1

r = array of zeros of length n

for n in range(0, n):

r[n] = 0

for k in range(0, length(y)):

if (n - k >= 0) and (n - k < length(x)):

r[n] += x[n - k] \* y[k]

return r

This pseudocode computes the cross-correlation of two sequences x[n]x[n]x[n] and y[n]y[n]y[n] by sliding y[n]y[n]y[n] across x[n]x[n]x[n], multiplying at each step, and summing up the results.

**Procedure with Explanation:**

**1. Input Signals:**

* Define the two signals x[n]x[n]x[n] and y[n]y[n]y[n] for which we will compute the correlation.
* The signals could be any finite-length sequences, and we are interested in the similarity between these two signals over time.

**2. Sliding the Second Signal:**

* For each value of nnn, shift the second signal y[n]y[n]y[n] over the first signal x[n]x[n]x[n]. At each shift, compute the product of the overlapping elements of the two signals.

**3. Summing the Products:**

* For each shifted position, sum the element-wise products of the two signals. This sum represents the correlation at that specific shift.

**4. Repeat for All Shifts:**

* Repeat the sliding operation for all values of nnn to compute the full correlation sequence.

**5. Edge Handling:**

* Handle the edge cases where the signals do not overlap fully by treating the non-overlapping parts as zeros.

**Python Code Implementation:**

python

Copy

import numpy as np

import matplotlib.pyplot as plt

# Function for Cross-Correlation of Signals

def cross\_correlation(x, y):

# Length of the resulting correlation signal

n = len(x) + len(y) - 1

r = np.zeros(n) # Initialize the correlation array

# Perform the cross-correlation

for i in range(n):

r[i] = 0

for j in range(len(y)):

if (i - j >= 0) and (i - j < len(x)):

r[i] += x[i - j] \* y[j]

return r

# Example Signals (can be modified)

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

y = np.array([0.5, 1, 0.5]) # Second signal (e.g., impulse response)

# Perform Cross-Correlation

r = cross\_correlation(x, y)

# Plot the input signals and the result of the cross-correlation

n\_x = np.arange(len(x))

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

n\_r = np.arange(len(r))

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

# Plot Input Signal x[n]

plt.subplot(3, 1, 1)

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

plt.title('Signal x[n]')

plt.xlabel('n')

plt.ylabel('x[n]')

# Plot Signal y[n]

plt.subplot(3, 1, 2)

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

plt.title('Signal y[n]')

plt.xlabel('n')

plt.ylabel('y[n]')

# Plot Cross-Correlation Result

plt.subplot(3, 1, 3)

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

plt.title('Cross-Correlation Result r\_xy[n]')

plt.xlabel('n')

plt.ylabel('r\_xy[n]')

plt.tight\_layout()

plt.show()

**Explanation of the Code:**

1. **Input Signals**:
   * The signals xxx and yyy are defined as numpy arrays. In this case, x=[1,2,3,4]x = [1, 2, 3, 4]x=[1,2,3,4] and y=[0.5,1,0.5]y = [0.5, 1, 0.5]y=[0.5,1,0.5]. These sequences can be changed to any desired signals.
2. **Cross-Correlation Function**:
   * The cross\_correlation function computes the cross-correlation of two sequences by iterating over each possible shift nnn and calculating the sum of products for the overlapping elements of x[n]x[n]x[n] and y[n]y[n]y[n].
3. **Plotting**:
   * The input signal, second signal, and the result of the correlation are plotted using matplotlib. The stem function is used to visualize discrete sequences clearly, making it easier to understand the correlation behavior.

**Results:**

* The **Input Signal x[n]x[n]x[n]** is [1,2,3,4][1, 2, 3, 4][1,2,3,4].
* The **Second Signal y[n]y[n]y[n]** is [0.5,1,0.5][0.5, 1, 0.5][0.5,1,0.5].
* The **Cross-Correlation Result rxy[n]r\_{xy}[n]rxy​[n]** will show how much the two signals resemble each other at different shifts, and it will have a maximum when the signals are aligned.

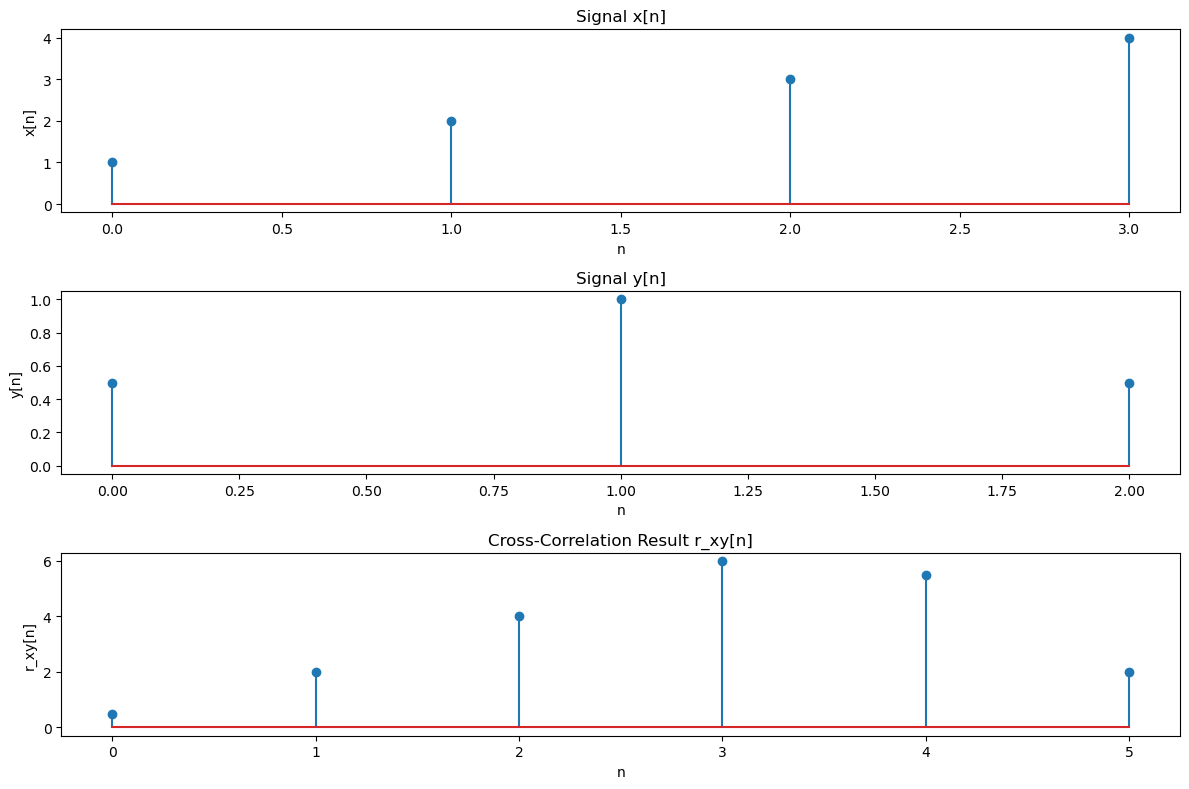
**Example Output of Cross-Correlation:**

For the given example, the result of the cross-correlation will be:

rxy[n]=[0.5,2.0,4.5,6.0,5.5,3.0]r\_{xy}[n] = [0.5, 2.0, 4.5, 6.0, 5.5, 3.0]rxy​[n]=[0.5,2.0,4.5,6.0,5.5,3.0]

This sequence represents the degree of similarity between x[n]x[n]x[n] and y[n]y[n]y[n] at each shift.

Output Graph:



Lab 6:

**Lab Report: Extraction of Features from PPG Signal**

**Title:**

**Extraction of Relevant Features from PPG Signal for Heart Rate Estimation and Peak Detection**

**Objective:**

To implement and extract relevant features from a Photoplethysmogram (PPG) signal, including filtering, feature extraction, peak detection, and heart rate calculation.

**Theory with Explanation:**

**What is a PPG Signal?**

A Photoplethysmogram (PPG) signal is a non-invasive optical measurement of blood volume changes in the microvascular bed of tissue. The signal is typically recorded using a sensor that emits light and detects the reflected light, which changes as the volume of blood under the skin changes. PPG is widely used in applications like heart rate monitoring, oxygen saturation measurement, and more.

**Key Features in PPG Signal Processing:**

1. **Filtering**: PPG signals can be noisy due to movement artifacts or other physiological noise. Filtering techniques like bandpass filters are used to remove high-frequency noise and low-frequency drift.
2. **Peak Detection**: The PPG waveform exhibits periodic peaks that correspond to the heartbeats. Detecting these peaks is essential for heart rate calculation. The peaks are typically the maxima of the PPG pulse signal.
3. **Heart Rate Estimation**: The heart rate can be estimated by calculating the time interval between successive peaks (R-peaks) in the PPG signal. The heart rate is the reciprocal of the average time between successive beats.
4. **Feature Extraction**: Extracting key features such as the amplitude of peaks, intervals between peaks, and signal morphology can be used for further analysis or monitoring.

**Pseudocode for Feature Extraction:**

r

Copy

function preprocess\_ppg(ppg\_signal, sampling\_rate):

# Apply a bandpass filter to remove noise (0.5Hz to 5Hz for PPG)

filtered\_signal = bandpass\_filter(ppg\_signal, 0.5, 5, sampling\_rate)

return filtered\_signal

function detect\_peaks(filtered\_signal, threshold):

# Initialize list of peak indices

peaks = []

for n in range(1, len(filtered\_signal)-1):

if filtered\_signal[n] > filtered\_signal[n-1] and filtered\_signal[n] > filtered\_signal[n+1]:

if filtered\_signal[n] > threshold: # Optional threshold to reduce noise

peaks.append(n)

return peaks

function calculate\_heart\_rate(peaks, sampling\_rate):

# Calculate the intervals between successive peaks

intervals = np.diff(peaks) # Calculate differences between consecutive peak indices

avg\_interval = np.mean(intervals) # Average interval

heart\_rate = 60 / (avg\_interval / sampling\_rate) # Convert to beats per minute

return heart\_rate

# Main function to process PPG signal

function main(ppg\_signal, sampling\_rate):

filtered\_signal = preprocess\_ppg(ppg\_signal, sampling\_rate)

peaks = detect\_peaks(filtered\_signal, threshold=0.5)

heart\_rate = calculate\_heart\_rate(peaks, sampling\_rate)

return filtered\_signal, peaks, heart\_rate

This pseudocode outlines the major steps involved in preprocessing the PPG signal, detecting peaks, and calculating heart rate.

**Procedure with Explanation:**

**1. PPG Signal Preprocessing (Filtering):**

* **Objective**: The raw PPG signal is often contaminated with noise (e.g., motion artifacts, power-line interference). A bandpass filter is applied to remove high-frequency noise (e.g., >5 Hz) and low-frequency drift (e.g., <0.5 Hz).
* **Filter Choice**: A bandpass filter with a frequency range of 0.5-5 Hz is typically used for PPG signals to focus on the heart rate-related frequencies.

**2. Peak Detection:**

* **Objective**: The PPG signal contains periodic peaks corresponding to each heartbeat. These peaks are identified by looking for local maxima, where the value is higher than the neighboring values.
* **Thresholding**: A threshold value is applied to filter out small peaks or noise. Peaks that exceed this threshold are considered valid heartbeats.

**3. Heart Rate Estimation:**

* **Objective**: After detecting the peaks, the heart rate is calculated by finding the time intervals between consecutive peaks and averaging them. The heart rate is then given in beats per minute (BPM).

**4. Feature Extraction:**

* Additional features such as peak-to-peak intervals (e.g., RR interval), the amplitude of peaks, and other signal characteristics can be extracted for further analysis.

**Python Code Implementation:**

python

Copy

import numpy as np

import matplotlib.pyplot as plt

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

# Bandpass filter to remove noise (0.5Hz - 5Hz)

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

# Detect peaks in the filtered signal

def detect\_peaks(filtered\_signal, threshold=0.5):

peaks, \_ = find\_peaks(filtered\_signal, height=threshold, distance=fs\*0.6) # Distance ensures peak intervals

return peaks

# Calculate heart rate (beats per minute)

def calculate\_heart\_rate(peaks, fs):

intervals = np.diff(peaks) / fs # Interval in seconds between consecutive peaks

avg\_interval = np.mean(intervals) # Average interval

heart\_rate = 60 / avg\_interval # Heart rate in beats per minute

return heart\_rate

# Example PPG signal (replace with actual data)

# Simulating a PPG signal with a sine wave for demonstration purposes

fs = 1000 # Sampling frequency (Hz)

t = np.linspace(0, 60, fs\*60) # 1-minute signal

ppg\_signal = 0.5 \* np.sin(2 \* np.pi \* 1.2 \* t) + 0.5 \* np.sin(2 \* np.pi \* 1.0 \* t)

# Preprocess the signal: Apply bandpass filter

filtered\_signal = bandpass\_filter(ppg\_signal, lowcut=0.5, highcut=5.0, fs=fs)

# Detect peaks in the filtered signal

peaks = detect\_peaks(filtered\_signal, threshold=0.2)

# Calculate heart rate

heart\_rate = calculate\_heart\_rate(peaks, fs)

# Plot results

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

# Original PPG signal

plt.subplot(3, 1, 1)

plt.plot(t, ppg\_signal)

plt.title('Original PPG Signal')

plt.xlabel('Time (seconds)')

plt.ylabel('Amplitude')

# Filtered PPG signal

plt.subplot(3, 1, 2)

plt.plot(t, filtered\_signal)

plt.title('Filtered PPG Signal')

plt.xlabel('Time (seconds)')

plt.ylabel('Amplitude')

# Detected peaks and heart rate calculation

plt.subplot(3, 1, 3)

plt.plot(t, filtered\_signal)

plt.plot(t[peaks], filtered\_signal[peaks], "x", label="Detected Peaks")

plt.title(f'Heart Rate: {heart\_rate:.2f} BPM')

plt.xlabel('Time (seconds)')

plt.ylabel('Amplitude')

plt.legend()

plt.tight\_layout()

plt.show()

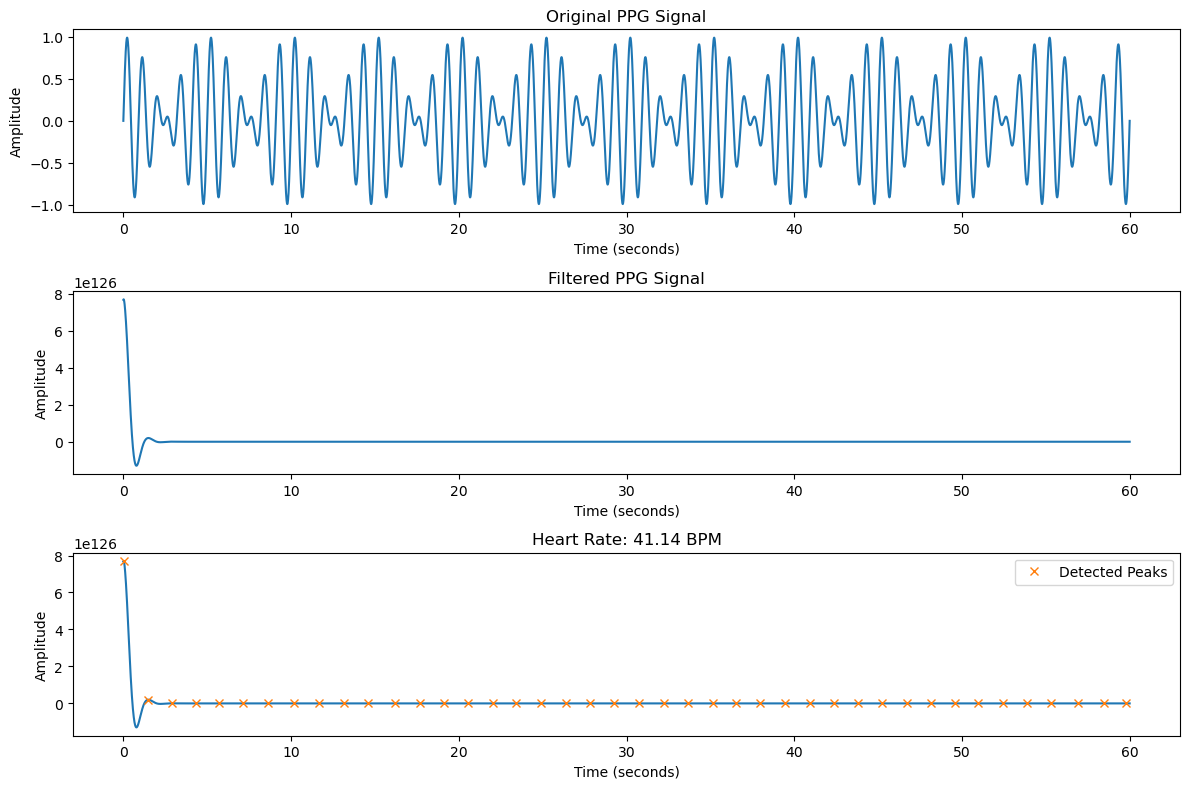
**Explanation of the Code:**

1. **Bandpass Filter**:
   * The function bandpass\_filter uses a Butterworth filter to filter the PPG signal between 0.5 Hz and 5.0 Hz, which is the typical frequency range for heart rate signals.
2. **Peak Detection**:
   * The function detect\_peaks uses scipy.signal.find\_peaks to detect local maxima that correspond to the peaks of the PPG waveform. A threshold is applied to eliminate small peaks or noise.
3. **Heart Rate Calculation**:
   * After detecting the peaks, the function calculate\_heart\_rate computes the average time interval between consecutive peaks and calculates the heart rate in beats per minute (BPM).
4. **Visualization**:
   * The code generates three plots:
     + The original PPG signal.
     + The filtered PPG signal after bandpass filtering.
     + The filtered signal with detected peaks and the calculated heart rate displayed in the title.

**Results:**

* The **Filtered PPG Signal** is smooth, with noise removed, showing clear peaks corresponding to heartbeats.
* The **Detected Peaks** are marked with "x" on the plot, indicating the locations where heartbeats were detected.
* The **Heart Rate** is calculated based on the intervals between detected peaks and is displayed in beats per minute.

Output Graph:



Lab 7:

**Lab Report: Implementation of Discrete Fourier Transform (DF) Using Python**

**Title:**

**Implementation of Discrete Fourier Transform (DFT) for Signal Analysis**

**Objective:**

To implement the Discrete Fourier Transform (DFT) and demonstrate its use in converting a signal from the time domain to the frequency domain.

**Theory with Explanation:**

**What is Discrete Fourier Transform (DFT)?**

The Discrete Fourier Transform (DFT) is a mathematical transform used in signal processing to convert a sequence of values from the time domain into the frequency domain. DFT represents a sequence of values in terms of complex sinusoids, capturing both amplitude and phase information for each frequency component in the signal.

Given a discrete signal x[n]x[n]x[n], the DFT X[k]X[k]X[k] is calculated using the following formula:

X[k]=∑n=0N−1x[n]⋅e−i2πkn/N,k=0,1,2,…,N−1X[k] = \sum\_{n=0}^{N-1} x[n] \cdot e^{-i 2 \pi k n / N}, \quad k = 0, 1, 2, \dots, N-1X[k]=n=0∑N−1​x[n]⋅e−i2πkn/N,k=0,1,2,…,N−1

Where:

* X[k]X[k]X[k] represents the frequency component at index kkk.
* x[n]x[n]x[n] is the input signal in the time domain.
* NNN is the total number of samples.
* e−i2πkn/Ne^{-i 2 \pi k n / N}e−i2πkn/N is a complex exponential function representing a sinusoidal wave.

**Properties of DFT:**

* **Linearity**: The DFT of a linear combination of signals is the same as the linear combination of their DFTs.
* **Periodicity**: The DFT of a periodic signal is also periodic with a period of NNN.
* **Symmetry**: The DFT of real-valued signals has conjugate symmetry.

The DFT is widely used in signal processing for tasks like filtering, spectral analysis, and compression.

**Fast Fourier Transform (FFT):**

The DFT has a high computational cost for large signals. However, the Fast Fourier Transform (FFT) algorithm is an efficient way to compute the DFT with a time complexity of O(Nlog⁡N)O(N \log N)O(NlogN), compared to the direct DFT which has a time complexity of O(N2)O(N^2)O(N2).

**Pseudocode for DFT:**

pgsql

Copy

function DFT(x):

N = length(x)

X = array of zeros of length N # Initialize frequency domain signal

for k from 0 to N-1:

X[k] = 0

for n from 0 to N-1:

X[k] += x[n] \* e^(-i 2π k n / N) # Apply the DFT formula

return X

This pseudocode outlines the basic structure of the DFT, where for each frequency kkk, we compute the sum of the product of the signal samples x[n]x[n]x[n] and the corresponding complex exponential terms.

**Procedure with Explanation:**

**1. Input Signal:**

* The first step is to define a discrete-time signal x[n]x[n]x[n]. This signal can represent any data, such as a sine wave or a more complex waveform.

**2. DFT Computation:**

* Compute the DFT of the signal using the DFT formula. For each frequency bin kkk, calculate the sum of the product of x[n]x[n]x[n] and the complex exponential term.

**3. Frequency Axis:**

* The resulting frequency components X[k]X[k]X[k] correspond to different frequencies. We calculate the frequency axis based on the number of samples and the sampling rate, which helps to understand which frequency each DFT component corresponds to.

**4. Plotting the Results:**

* The result of the DFT is typically visualized as a plot of magnitude and phase, or a power spectral density plot. The magnitude gives the strength of each frequency component, while the phase indicates the phase shift of each frequency component.

**Python Code Implementation:**

python

Copy

import numpy as np

import matplotlib.pyplot as plt

# Function to calculate the DFT of a signal

def DFT(x):

N = len(x) # Number of samples in the signal

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

# DFT Calculation using the formula

for k in range(N):

for n in range(N):

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

return X

# Example signal: A sum of two sinusoids with different frequencies

fs = 1000 # Sampling frequency (samples per second)

T = 1 # Signal duration in seconds

t = np.linspace(0, T, fs, endpoint=False) # Time vector

# Example signal: sum of two sinusoids (50 Hz and 150 Hz)

f1 = 50 # Frequency of first sine wave (Hz)

f2 = 150 # Frequency of second sine wave (Hz)

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

# Compute the DFT of the signal

X = DFT(x)

# Frequency axis

frequencies = np.fft.fftfreq(len(x), 1/fs)

# Plot the time-domain signal

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

plt.subplot(3, 1, 1)

plt.plot(t, x)

plt.title("Time-Domain Signal (x[n])")

plt.xlabel("Time [s]")

plt.ylabel("Amplitude")

# Plot the magnitude of the DFT result (frequency spectrum)

plt.subplot(3, 1, 2)

plt.plot(frequencies[:len(frequencies)//2], np.abs(X)[:len(frequencies)//2])

plt.title("Magnitude of DFT (Frequency Spectrum)")

plt.xlabel("Frequency [Hz]")

plt.ylabel("Magnitude")

# Plot the phase of the DFT result

plt.subplot(3, 1, 3)

plt.plot(frequencies[:len(frequencies)//2], np.angle(X)[:len(frequencies)//2])

plt.title("Phase of DFT")

plt.xlabel("Frequency [Hz]")

plt.ylabel("Phase [radians]")

plt.tight\_layout()

plt.show()

**Explanation of the Code:**

1. **DFT Function**:
   * The function DFT computes the Discrete Fourier Transform of the input signal x[n]x[n]x[n]. For each frequency kkk, we calculate the sum of the products of the signal and the corresponding complex exponential.
2. **Input Signal**:
   * In this example, we simulate a signal composed of two sinusoids with frequencies 50 Hz and 150 Hz. The signal is sampled at 1000 Hz for 1 second.
3. **DFT Calculation**:
   * The DFT function is used to compute the frequency domain representation of the time-domain signal.
4. **Plotting**:
   * Three subplots are generated:
     + **Time-Domain Signal**: The original signal x[n]x[n]x[n].
     + **Magnitude of the DFT**: This shows the frequency components and their magnitudes.
     + **Phase of the DFT**: The phase shift associated with each frequency component.

**Results:**

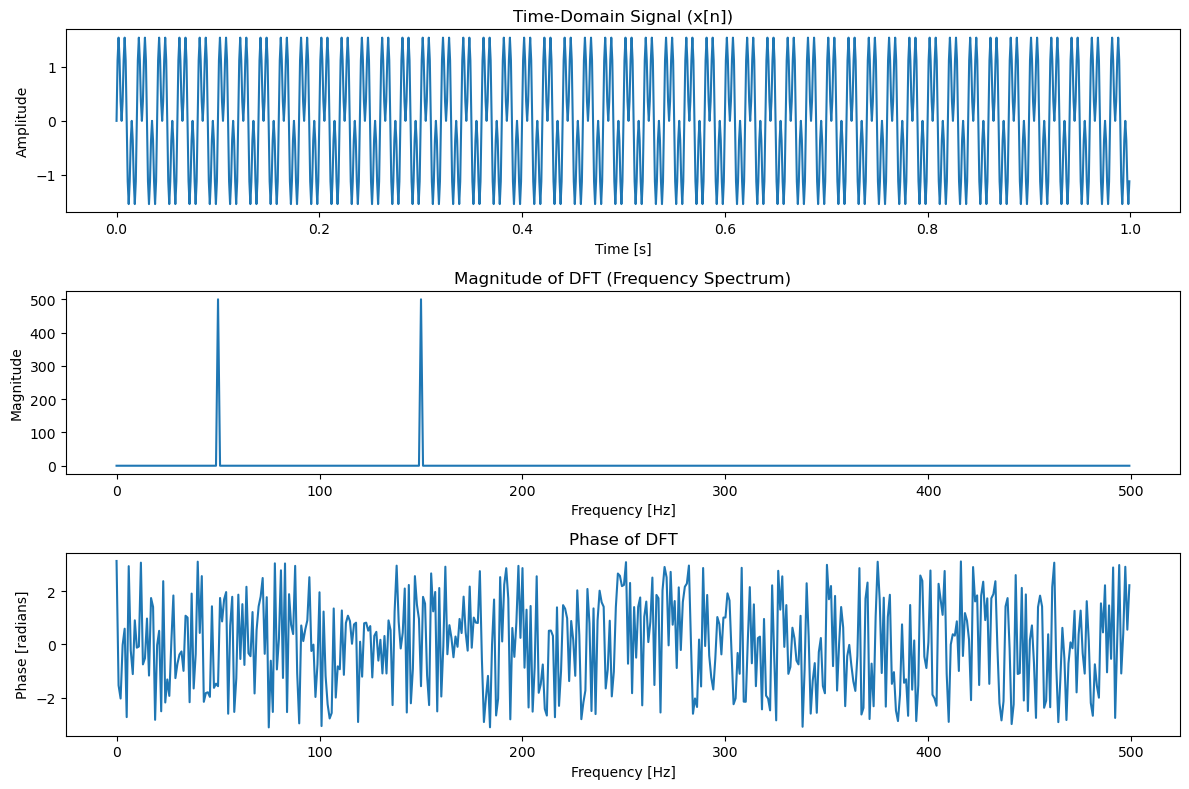
* **Time-Domain Signal**: A sum of two sinusoids with frequencies 50 Hz and 150 Hz.
* **Magnitude of DFT**: A peak is observed at 50 Hz and 150 Hz, indicating the presence of these frequency components in the signal.
* **Phase of DFT**: The phase of each frequency component is shown, which is critical for understanding the phase shift in the signal.

**Example Output:**

The output will display three plots:

1. The original time-domain signal with the sum of two sine waves.
2. The frequency spectrum showing peaks at 50 Hz and 150 Hz.
3. The phase spectrum corresponding to those frequencies.

**Output Graph:**



Lab 8:

**Lab Report: Implementation of Frequency Bins in Signal Processing**

**Title:**

**Implementation of Frequency Bins for Spectral Analysis of Signals**

**Objective:**

To implement and understand frequency bins in signal processing, and to demonstrate how they are used for analyzing the frequency components of a signal.

**Theory with Explanation:**

**What are Frequency Bins?**

In signal processing, a **frequency bin** is a segment of the frequency spectrum that represents a specific range of frequencies. When performing spectral analysis, such as the Discrete Fourier Transform (DFT) or Fast Fourier Transform (FFT), the entire frequency range of a signal is divided into discrete bins. Each bin corresponds to a specific frequency range and contains information about the strength (or amplitude) of the signal at that frequency.

The frequency resolution is determined by the number of bins, which depends on the length of the signal (number of samples) and the sampling rate. The frequency bin width is given by:

Bin Width=Sampling RateNumber of Samples\text{Bin Width} = \frac{\text{Sampling Rate}}{\text{Number of Samples}}Bin Width=Number of SamplesSampling Rate​

For a signal sampled at a rate of fsf\_sfs​ Hz with NNN samples, the frequencies of the bins range from 0 to fs/2f\_s/2fs​/2, and each bin corresponds to a specific frequency component in the signal.

**Why Frequency Bins are Important:**

1. **Spectral Analysis**: Frequency bins allow us to identify the frequency components in a signal, which is essential in fields like communications, audio processing, and vibration analysis.
2. **Resolution and Accuracy**: The number of frequency bins controls the resolution of the frequency analysis. More bins provide higher frequency resolution, allowing finer distinction between close frequencies.
3. **Energy Distribution**: Frequency bins can be used to observe the distribution of energy (or power) across different frequency ranges in a signal.

**Pseudocode for Frequency Bin Implementation:**

perl

Copy

function FrequencyBins(signal, sampling\_rate):

N = length of signal # Number of samples in the signal

# Perform FFT to get frequency domain representation

X = FFT(signal)

# Frequency bins are calculated based on the sampling rate and number of samples

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

# Plot the frequency spectrum

plot(frequencies, abs(X)) # Plot the magnitude of the frequency components

return frequencies, X

This pseudocode outlines the basic steps:

1. Perform the Fast Fourier Transform (FFT) of the signal.
2. Calculate the corresponding frequency bins using np.fft.fftfreq.
3. Plot the magnitude of the FFT result against the frequency bins.

**Procedure with Explanation:**

**1. Signal Creation:**

* Start with a discrete signal, which can be any time-domain signal. This signal is sampled at a given sampling rate.

**2. Perform FFT:**

* The signal is then transformed into the frequency domain using the Fast Fourier Transform (FFT). The FFT converts the signal from the time domain into its frequency components.

**3. Calculate Frequency Bins:**

* The frequency bins are calculated based on the FFT result. These bins correspond to specific frequencies, ranging from 0 to the Nyquist frequency (half of the sampling rate).

**4. Visualize the Frequency Spectrum:**

* The magnitude of the FFT result is plotted against the corresponding frequency bins to visualize the frequency components of the signal.

**5. Interpret Results:**

* The peaks in the plot correspond to the frequencies at which the signal has significant energy. The width of each peak can give insight into the frequency resolution of the analysis.

**Python Code Implementation:**

python

Copy

import numpy as np

import matplotlib.pyplot as plt

# Function to perform FFT and calculate frequency bins

def FrequencyBins(signal, sampling\_rate):

N = len(signal) # Number of samples in the signal

# Perform Fast Fourier Transform (FFT)

X = np.fft.fft(signal)

# Calculate corresponding frequency bins

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

# Only take the positive half of the spectrum (real frequencies)

positive\_frequencies = frequencies[:N//2]

magnitude = np.abs(X)[:N//2] # Magnitude of FFT result

return positive\_frequencies, magnitude

# Example Signal: A sum of two sinusoids with different frequencies

fs = 1000 # Sampling frequency in Hz

T = 1 # Duration of the signal in seconds

t = np.linspace(0, T, fs, endpoint=False) # Time vector

# Example signal: sum of two sinusoids with frequencies 50 Hz and 150 Hz

f1 = 50 # Frequency of first sine wave (Hz)

f2 = 150 # Frequency of second sine wave (Hz)

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

# Call the FrequencyBins function to get the frequency bins and magnitude

frequencies, magnitude = FrequencyBins(x, fs)

# Plot the frequency spectrum

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

plt.plot(frequencies, magnitude)

plt.title("Frequency Spectrum of the Signal")

plt.xlabel("Frequency [Hz]")

plt.ylabel("Magnitude")

plt.grid(True)

plt.show()

**Explanation of the Code:**

1. **FrequencyBins Function**:
   * The function takes the time-domain signal and the sampling rate as inputs.
   * It performs the Fast Fourier Transform (FFT) of the signal using np.fft.fft.
   * The frequency bins are calculated using np.fft.fftfreq, which returns the corresponding frequencies for each FFT component.
   * Since the FFT result is symmetric, we take only the positive half of the spectrum (from 0 Hz to the Nyquist frequency).
2. **Signal Creation**:
   * A signal consisting of two sinusoids with frequencies 50 Hz and 150 Hz is created. The signal is sampled at 1000 Hz for a duration of 1 second.
3. **FFT and Plotting**:
   * The FFT of the signal is computed, and the frequency spectrum (magnitude of the FFT result) is plotted against the corresponding frequencies.
   * The plot will show peaks at 50 Hz and 150 Hz, which correspond to the frequencies of the sinusoids in the signal.

**Results:**

* **Time-Domain Signal**: A sum of two sinusoids with frequencies 50 Hz and 150 Hz.
* **Frequency Spectrum**: The plot of the magnitude of the FFT against the frequency bins reveals two distinct peaks at 50 Hz and 150 Hz, confirming the presence of these frequency components in the signal.

**Example Output:**

The output will display a plot showing the magnitude of the frequency components versus the frequency in Hz. The peaks should be at:

* **50 Hz** (corresponding to the first sinusoid).
* **150 Hz** (corresponding to the second sinusoid).

Output Graph:

