

**Submission Date : 03.03.25**

**Course Title : Signals and Systems Sessional**

**Course Code : ICE - 2204**

**Index**

|  |  |
| --- | --- |
| **SI.** | **Problem Statement** |
| **1** | Write a program on signal operation - addition, shifting, folding, multiplication. |
| **2** | Explain and implementation of Convolution operation of sequences. |
| **3** | Explain and implementation of Correlation operation of sequences. |
| **4** | Explain and implementation of signal sequence. |
| **5** | Write a program on PPG signal - filtering, feature extraction, peak detection. |
| **6** | Explain and implementation of fourier transform. |
| **7** | Write a program on DFT (Discrete Fourier Transform) and IDFT. |
| **8** | Explain and implementation of fourier series decomposition. |

**No. of the experiment : 01**

**Name of the experiment :** Write a program on signal operation - addition, shifting, folding, multiplication.

**Theory :**

**Signal Addition**

Signal addition involves summing two discrete signals element-wise. It is used in applications like combining audio signals or overlaying sensor data.

### Signal Shifting

Shifting a signal involves delaying (right shift) or advancing (left shift) the signal in time. This is crucial in aligning signals in systems and applications like echo generation.

### Signal Folding

Folding, also known as time-reversal, flips the signal around the vertical axis. It is helpful in analyzing signal symmetry and convolution operations.

### Signal Multiplication

Signal multiplication involves multiplying two signals element-wise. It is used in modulation and windowing techniques in DSP.

**Source Code :**

import numpy as np

import matplotlib.pyplot as plt

# Define a sample discrete signal

n = np.arange(-10, 11)

x = np.sin(0.2 \* np.pi \* n)

y = np.cos(0.2 \* np.pi \* n)

# 1. Signal Addition

addition = x + y

# 2. Signal Shifting

shift = 3 # Shift by 3 units to the right

shifted\_signal = np.roll(x, shift)

# 3. Signal Folding

folded\_signal = x[::-1]

# 4. Signal Multiplication

multiplication = x \* y

# Plotting results

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

plt.subplot(2, 2, 1)

plt.stem(n, addition, basefmt=" ")

plt.title("Signal Addition")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.subplot(2, 2, 2)

plt.stem(n, shifted\_signal, basefmt=" ")

plt.title("Signal Shifting (Right by 3)")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.subplot(2, 2, 3)

plt.stem(n, folded\_signal, basefmt=" ")

plt.title("Signal Folding")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.subplot(2, 2, 4)

plt.stem(n, multiplication, basefmt=" ")

plt.title("Signal Multiplication")

plt.xlabel("n")

plt.ylabel("Amplitude")

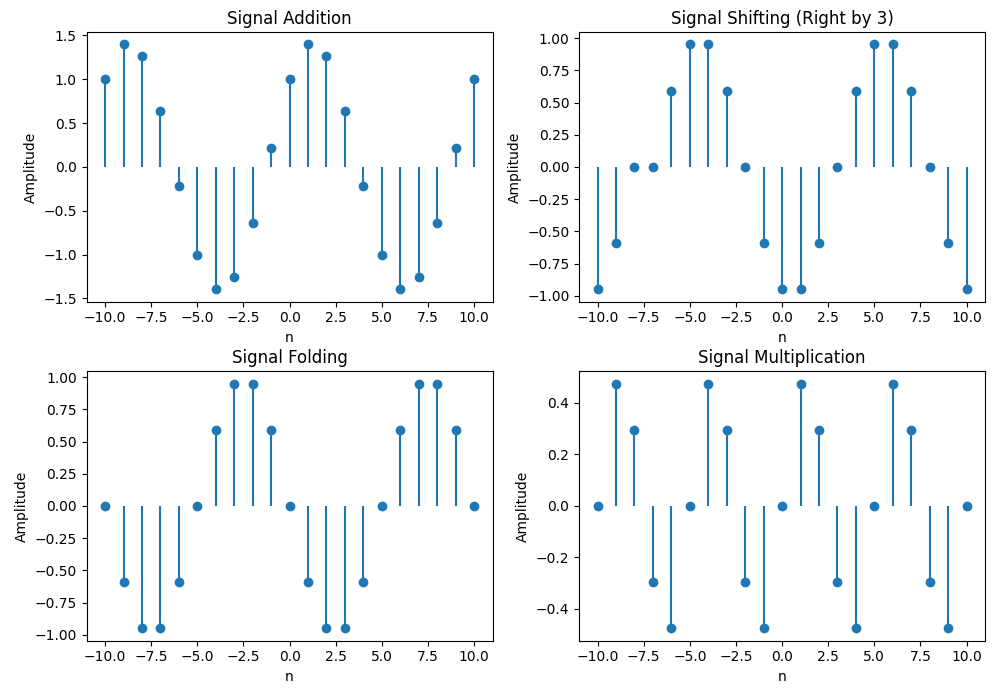
plt.tight\_layout()

plt.show()

## Input :

* Two discrete signals: x = sin(0.2πn) and y = cos(0.2πn)
* Range of n: n = [-10, 10]
* Shift amount: 3

## Output:



**Purpose :**

The purpose of this lab is to understand and implement basic signal operations such as addition, shifting, folding, and multiplication using Python. These operations are fundamental in digital signal processing (DSP) and help manipulate signals for analysis and processing.

**No. of the experiment : 02**

**Name of the experiment :** Explain and implementation of Convolution operation of sequences.

## Theory :

Convolution is a mathematical operation that combines two signals to produce a third signal, representing how the shape of one signal is modified by another. It is widely used in DSP for linear system analysis and image processing.

### Convolution Formula :

For discrete-time signals, the convolution of two signals x[n] and h[n] is defined as:



Where:

* **x[n]** is the input signal.
* **h[n]** is the impulse response of the system.
* **y[n]** is the resulting output signal.
* **k** is the summation index.

### Applications of Convolution :

* **Filtering:** Smoothening or sharpening signals.
* **System Response:** Determining how systems react to input signals.
* **Image Processing:** Blurring, edge detection, and applying various filters.

**Source Code :**

**import numpy as np**

**import matplotlib.pyplot as plt**

**# Define sample discrete signals**

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

**x = np.sin(0.2 \* np.pi \* n) # Input signal**

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

**# Perform convolution**

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

**# Plotting input, impulse response, and convolution result**

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

**plt.subplot(3, 1, 1)**

**plt.stem(n, x, basefmt=" ")**

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

**plt.xlabel("n")**

**plt.ylabel("Amplitude")**

**plt.subplot(3, 1, 2)**

**plt.stem(np.arange(len(h)), h, basefmt=" ")**

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

**plt.xlabel("n")**

**plt.ylabel("Amplitude")**

**plt.subplot(3, 1, 3)**

**plt.stem(n, y, basefmt=" ")**

**plt.title("Convolution Result y[n]")**

**plt.xlabel("n")**

**plt.ylabel("Amplitude")**

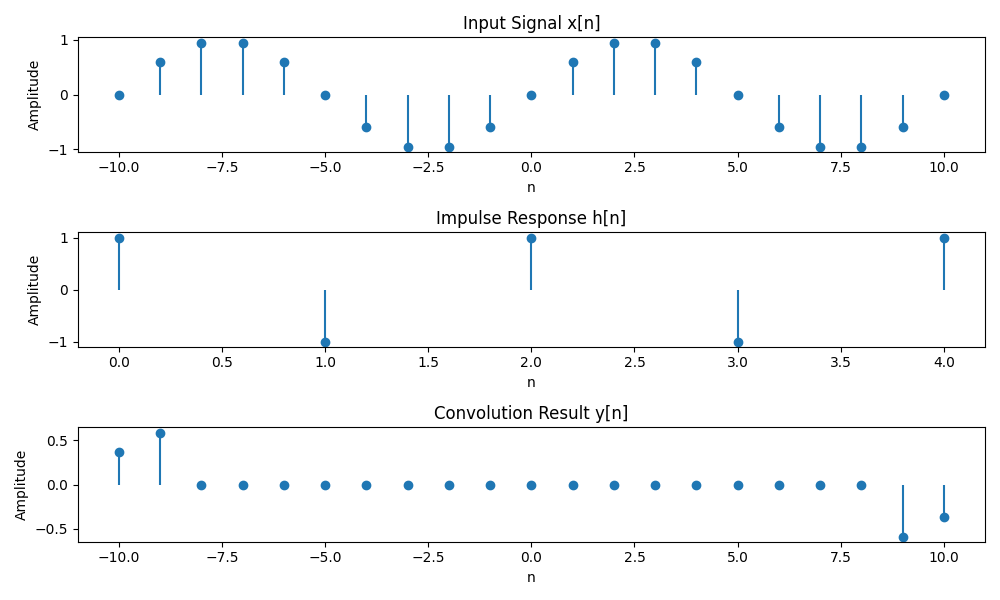
**plt.tight\_layout()**

**plt.show()**

## Input :

* **Input Signal (x[n]):** A sine wave with frequency component 0.2πn.
* **Impulse Response (h[n]):** A simple alternating pattern [1, -1, 1, -1, 1].
* **Range of n:** n = [-10, 10].

**Output :**

****

## 

## Purpose :

The purpose of this lab is to understand and implement the concept of convolution in digital signal processing (DSP) using Python. Convolution is a fundamental operation used in systems analysis, filtering, and feature extraction.

**No. of the experiment : 03**

**Name of the experiment :** Explain and implementation of Correlation operation of sequences.

## Theory :

Correlation is a mathematical operation that quantifies the degree to which two signals are similar. It involves sliding one signal over another and computing the dot product at each position, providing insight into the time delay between signals.

### Correlation Formula :

For discrete-time signals, the correlation of two signals x[n] and y[n] is defined as:



Where:

* **x[n]** is the reference signal.
* **y[n]** is the signal being compared.
* **R\_{xy}[l]** is the correlation as a function of lag **l**.

### Types of Correlation :

* **Cross-Correlation:** Measures the similarity between two different signals.
* **Auto-Correlation:** Measures the similarity of a signal with a delayed version of itself.

### Applications of Correlation :

* **Pattern Recognition:** Matching signals with templates.
* **Time Delay Estimation:** Finding shifts between signals.
* **Signal Detection:** Identifying known signals within noisy data.

**Source Code :**

import numpy as np

import matplotlib.pyplot as plt

# Define sample discrete signals

n = np.arange(-10, 11)

x = np.sin(0.2 \* np.pi \* n) # Signal x[n]

y = np.cos(0.2 \* np.pi \* n) # Signal y[n]

# Perform correlation

correlation = np.correlate(x, y, mode='full')

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

# Plotting the signals and correlation result

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

plt.subplot(3, 1, 1)

plt.stem(n, x, basefmt=" ")

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

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.subplot(3, 1, 2)

plt.stem(n, y, basefmt=" ")

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

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.subplot(3, 1, 3)

plt.stem(lag, correlation, basefmt=" ")

plt.title("Cross-Correlation Rxy[l]")

plt.xlabel("Lag l")

plt.ylabel("Correlation")

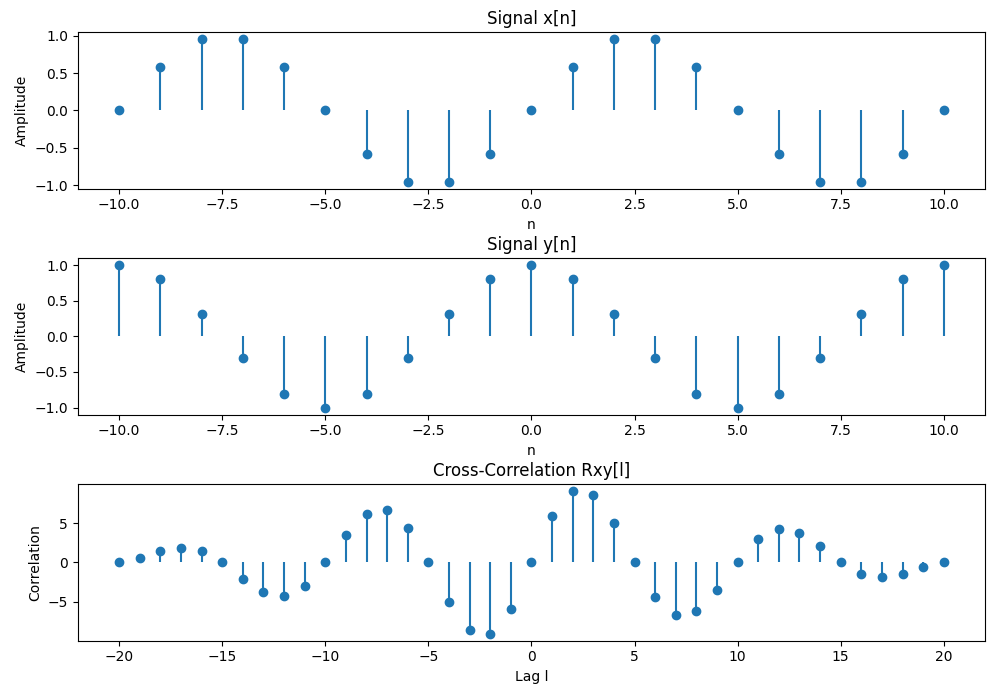
plt.tight\_layout()

plt.show()

## Input:

* Input Signals: x[n] = sin(0.2πn), y[n] = cos(0.2πn)
* Range of n: n = [-10, 10]

## Output :



## Purpose :

The purpose of this lab is to understand and implement the concept of correlation in digital signal processing (DSP) using Python. Correlation measures the similarity between two signals as a function of the time-lag applied to one of them. It is widely used in pattern recognition, signal detection, and feature matching.

**No. of the experiment : 04**

**Name of the experiment :** Explain and implementation of signal sequence.

## Theory :

A signal sequence is a discrete representation of a signal, often represented as a function of time (n) or frequency. In DSP, signals are usually represented as arrays or sequences of numbers that describe their amplitude at each discrete time step.

### Types of Signal Sequences :

1. **Unit Impulse Sequence (Delta Function, δ[n])**
   * A signal that is zero at all points except at n = 0, where it has a value of 1.
   * Useful in system response analysis.
2. **Unit Step Sequence (u[n])**
   * A signal that is zero for n < 0 and one for n ≥ 0.
   * Commonly used in testing system stability and transient response.
3. **Ramp Sequence**
   * A linear increasing sequence starting from zero.
   * Represents continuous growth in a system.
4. **Exponential Sequence**
   * A sequence where each value is an exponential function of its index.
   * Commonly used in modeling growth and decay in systems.
5. **Sinusoidal Sequence**
   * A periodic signal represented by sine or cosine functions.
   * Widely used in communications and signal processing.

**Source Code :**

import numpy as np

import matplotlib.pyplot as plt

# Define the range of n

n = np.arange(-10, 11)

# 1. Unit Impulse Sequence

impulse = np.zeros\_like(n)

impulse[n == 0] = 1

# 2. Unit Step Sequence

step = np.heaviside(n, 1)

# 3. Ramp Sequence

ramp = np.maximum(0, n)

# 4. Exponential Sequence

exponential = np.exp(0.1 \* n)

# 5. Sinusoidal Sequence

sinusoidal = np.sin(0.2 \* np.pi \* n)

# Plotting all signal sequences

plt.figure(figsize=(12, 10))

plt.subplot(3, 2, 1)

plt.stem(n, impulse, basefmt=" ")

plt.title("Unit Impulse Sequence")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.subplot(3, 2, 2)

plt.stem(n, step, basefmt=" ")

plt.title("Unit Step Sequence")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.subplot(3, 2, 3)

plt.stem(n, ramp, basefmt=" ")

plt.title("Ramp Sequence")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.subplot(3, 2, 4)

plt.stem(n, exponential, basefmt=" ")

plt.title("Exponential Sequence")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.subplot(3, 2, 5)

plt.stem(n, sinusoidal, basefmt=" ")

plt.title("Sinusoidal Sequence")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.tight\_layout()

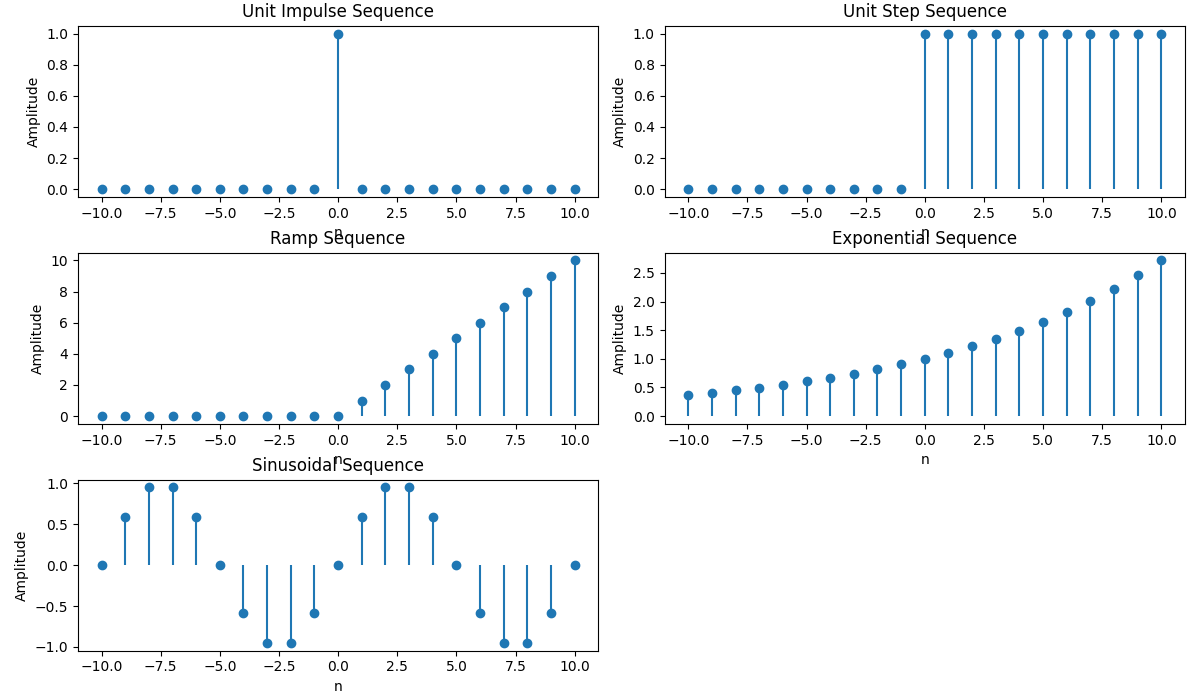
plt.show()

## 

## Input :

* Range of n: n = [-10, 10]
* Different formulas for each signal sequence type.

**Output :**

****

## 

## Purpose :

The purpose of this lab is to explore and analyze different types of signal sequences using Python. By understanding these fundamental sequences—including unit impulse, unit step, ramp, exponential, and sinusoidal signals—students will gain a solid foundation in digital signal processing (DSP). These sequences serve as building blocks for more advanced signal analysis, system response evaluation, and practical applications in communications, control systems, and electronics.

**No. of the experiment : 05**

**Name of the experiment :** Write a program on PPG signal - filtering, feature extraction, peak detection.

## Theory :

A Photoplethysmogram (PPG) signal is a non-invasive optical measurement technique used to detect blood volume changes in microvascular tissue. It involves shining light into the skin and measuring the amount of light either transmitted or reflected. The resulting waveform consists of pulsatile components synchronized with the heartbeat and is widely used in pulse oximetry and wearable health devices.

**Signal Filtering :**

Filtering is essential in preprocessing PPG signals to remove noise and artifacts. Common filters used in PPG processing include:

* **Low-pass Filter:** To remove high-frequency noise.
* **High-pass Filter:** To eliminate baseline wander.
* **Band-pass Filter:** To isolate the frequency range of interest, typically around 0.5 to 5 Hz for heart rate analysis.

### Feature Extraction :

Feature extraction involves deriving meaningful parameters from the PPG signal, such as:

* **Heart Rate:** Calculated from peak intervals.
* **Peak Amplitude:** Indicates pulse strength.
* **Pulse Interval Variability:** Reflects cardiovascular health.

### Peak Detection :

Peak detection is crucial for identifying heartbeats in the PPG signal. Methods include:

* **Thresholding:** Detecting peaks above a certain amplitude.
* **Signal Derivatives:** Highlighting rapid changes to find peak locations.
* **Find Peaks Algorithm:** Using libraries like scipy.signal.find\_peaks for robust detection.

**Source Code :**

**import numpy as np**

**import matplotlib.pyplot as plt**

**from scipy import signal**

**# Simulated PPG Signal with noise**

**fs = 100 # Sampling frequency (100 Hz)**

**t = np.linspace(0, 10, fs \* 10)**

**ppg\_signal = 1.2 \* np.sin(2 \* np.pi \* 1.2 \* t) + 0.5 \* np.random.normal(0, 0.5, len(t))**

**# 1. Filtering (Band-pass filter between 0.5 and 5 Hz)**

**b, a = signal.butter(4, [0.5 / (fs / 2), 5 / (fs / 2)], btype='band')**

**filtered\_signal = signal.filtfilt(b, a, ppg\_signal)**

**# 2. Peak Detection**

**peaks, \_ = signal.find\_peaks(filtered\_signal, distance=fs/2, height=0.5)**

**# 3. Feature Extraction (Heart Rate Calculation)**

**peak\_intervals = np.diff(peaks) / fs**

**heart\_rate = 60 / np.mean(peak\_intervals)**

**# Plotting the PPG signal, filtered signal, and detected peaks**

**plt.figure(figsize=(12, 10))**

**plt.subplot(2, 2, 1)**

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

**plt.title('Raw PPG Signal')**

**plt.xlabel('Time (s)')**

**plt.ylabel('Amplitude')**

**plt.grid()**

**plt.subplot(2, 2, 2)**

**plt.plot(t, filtered\_signal, label='Filtered PPG Signal', color='blue')**

**plt.title('Filtered PPG Signal')**

**plt.xlabel('Time (s)')**

**plt.ylabel('Amplitude')**

**plt.grid()**

**plt.subplot(2, 2, 3)**

**plt.plot(t, filtered\_signal, label='Feature Extraction', color='green')**

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

**plt.title('Feature Extraction')**

**plt.xlabel('Time (s)')**

**plt.ylabel('Amplitude')**

**plt.grid()**

**plt.subplot(2, 2, 4)**

**plt.plot(t, filtered\_signal, label='Peak Detection', color='purple')**

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

**plt.title(f'Peak Detection - Heart Rate: {heart\_rate:.2f} bpm')**

**plt.xlabel('Time (s)')**

**plt.ylabel('Amplitude')**

**plt.legend()**

**plt.grid()**

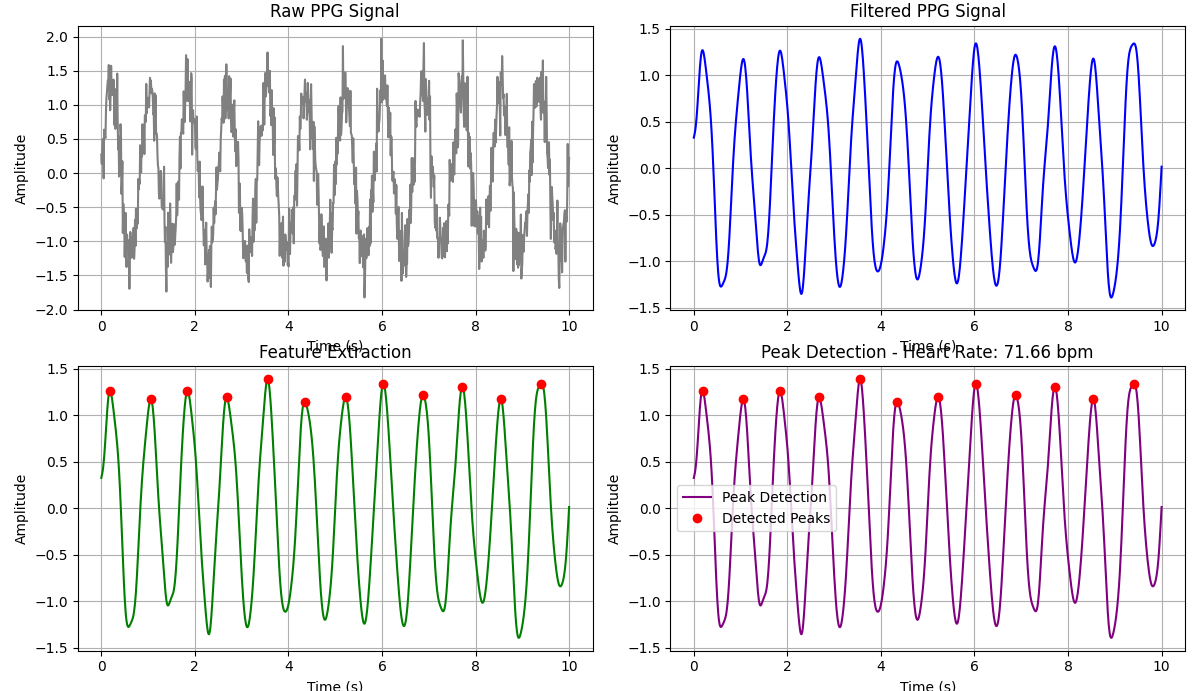
**plt.tight\_layout()**

**plt.show()**

## Input :

* Simulated PPG Signal: Sine wave with noise to mimic PPG characteristics.
* Sampling Frequency: 100 Hz.
* Filter Specifications: Band-pass between 0.5 and 5 Hz.

**Output :**

****

**Purpose :**

The purpose of this lab is to analyze Photoplethysmogram (PPG) signals using Python, focusing on signal filtering, feature extraction, and peak detection. PPG signals are widely used in healthcare for monitoring heart rate, blood oxygen levels, and assessing cardiovascular health. The goal is to demonstrate how to preprocess PPG data, extract relevant features, and accurately detect peaks corresponding to heartbeats.

**No. of the experiment : 06**

**Name of the experiment :** Explain and implementation of fourier transform.

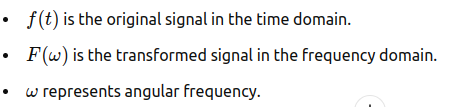
### Theory:

The Fourier Transform (FT) is a mathematical technique that transforms a function of time (or space) into a function of frequency. Named after Jean-Baptiste Joseph Fourier, it allows us to express a time-domain signal as the sum of sinusoidal waves, each with a specific frequency, amplitude, and phase.

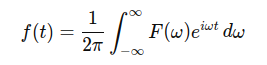
Mathematically, the Fourier Transform of a continuous-time function f(t)f(t)f(t) is defined as:



Where:



The Inverse Fourier Transform is used to convert the frequency-domain representation back to the time-domain signal:



In practical applications, the Discrete Fourier Transform (DFT) is used, which works with discrete signals sampled at regular intervals.

The Fast Fourier Transform (FFT) is an efficient algorithm for computing the DFT, often used in signal processing and data analysis.

**Source Code :**

**import numpy as np**

**import matplotlib.pyplot as plt**

**# Create a time-domain signal (sum of two sine waves)**

**sampling\_rate = 1000 # samples per second**

**T = 1 # duration in seconds**

**t = np.linspace(0, T, sampling\_rate) # time vector**

**# Define two sine waves with different frequencies**

**f1 = 50 # frequency of first sine wave (Hz)**

**f2 = 120 # frequency of second sine wave (Hz)**

**# Time-domain signal (sum of two sine waves)**

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

**# Compute the Fourier Transform using FFT**

**fft\_signal = np.fft.fft(signal)**

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

**# Only take the positive half of the frequencies (real part)**

**positive\_frequencies = frequencies[:len(frequencies) // 2]**

**fft\_signal\_magnitude = np.abs(fft\_signal)[:len(fft\_signal) // 2]**

**# Plot the time-domain 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 frequency-domain representation**

**plt.subplot(2, 1, 2)**

**plt.plot(positive\_frequencies, fft\_signal\_magnitude)**

**plt.title('Frequency-Domain Representation (Fourier Transform)')**

**plt.xlabel('Frequency [Hz]')**

**plt.ylabel('Magnitude')**

**plt.tight\_layout()**

**plt.show()**

### Input:

The input is a time-domain signal, which is a combination of two sine waves with frequencies of 50 Hz and 120 Hz. The signal is sampled at a rate of 1000 samples per second over a duration of 1 second.

**Time-Domain Signal:**

****

### Output:

The output of the Fourier Transform is the frequency-domain representation of the input signal, showing the magnitude of different frequency components. In this case, the two peaks in the frequency domain will appear at 50 Hz and 120 Hz, corresponding to the two sine waves in the original signal.

**Frequency-Domain Output:**

* Peak at 50 Hz
* Peak at 120 Hz

These peaks indicate the presence of the respective frequencies in the time-domain signal.

**Purpose:**

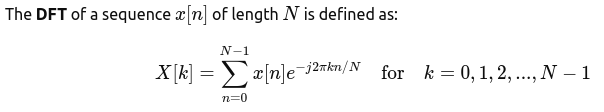
The purpose of this experiment is to understand the concept of the Fourier Transform, explore how it transforms a time-domain signal into its frequency-domain representation, and implement the Fourier Transform using Python. The experiment aims to help visualize how the Fourier Transform breaks down complex signals into simpler sine and cosine components.

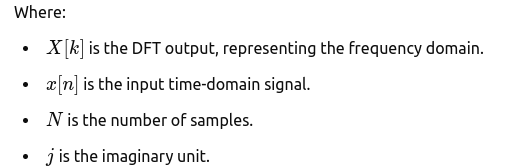
**No. of the experiment : 07**

**Name of the experiment :** Write a program on DFT (Discrete Fourier Transform) and IDFT.

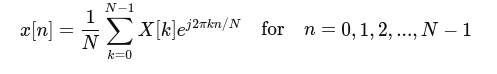
### Theory:

The **Discrete Fourier Transform (DFT)** is a mathematical technique that transforms a discrete-time signal into a discrete frequency spectrum. Unlike the continuous Fourier Transform, the DFT works with discrete signals and is particularly useful in digital signal processing, image processing, and data analysis.





The **Inverse Discrete Fourier Transform (IDFT)** is used to reconstruct the time-domain signal from its frequency-domain representation:

**Applications of DFT :**

* Signal and image processing.
* Spectral analysis.
* Filtering and signal reconstruction.
* Data compression (e.g., JPEG, MP3).

**Source Code :**

**import numpy as np**

**import matplotlib.pyplot as plt**

**# Define a discrete-time signal (sum of two sinusoidal signals)**

**sampling\_rate = 1000 # Samples per second**

**N = 1000 # Number of samples**

**t = np.arange(N) / sampling\_rate # Time vector**

**# Create a signal with two frequencies: 50 Hz and 120 Hz**

**f1 = 50 # Frequency of the first sine wave**

**f2 = 120 # Frequency of the second sine wave**

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

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

**dft\_signal = np.fft.fft(signal)**

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

**# Take only the positive half of the frequency spectrum**

**positive\_freqs = frequencies[:N // 2]**

**dft\_magnitude = np.abs(dft\_signal)[:N // 2]**

**# Plot the time-domain 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 frequency-domain representation using DFT**

**plt.subplot(2, 1, 2)**

**plt.stem(positive\_freqs, dft\_magnitude, 'b', markerfmt=" ", basefmt="-")**

**plt.title('Frequency-Domain Representation (DFT)')**

**plt.xlabel('Frequency [Hz]')**

**plt.ylabel('Magnitude')**

**plt.tight\_layout()**

**plt.show()**

### Input:

The input to the DFT is a discrete time-domain signal, which is a sum of two sinusoidal signals with frequencies of 50 Hz and 120 Hz. The signal is sampled at 1000 samples per second for 1 second.



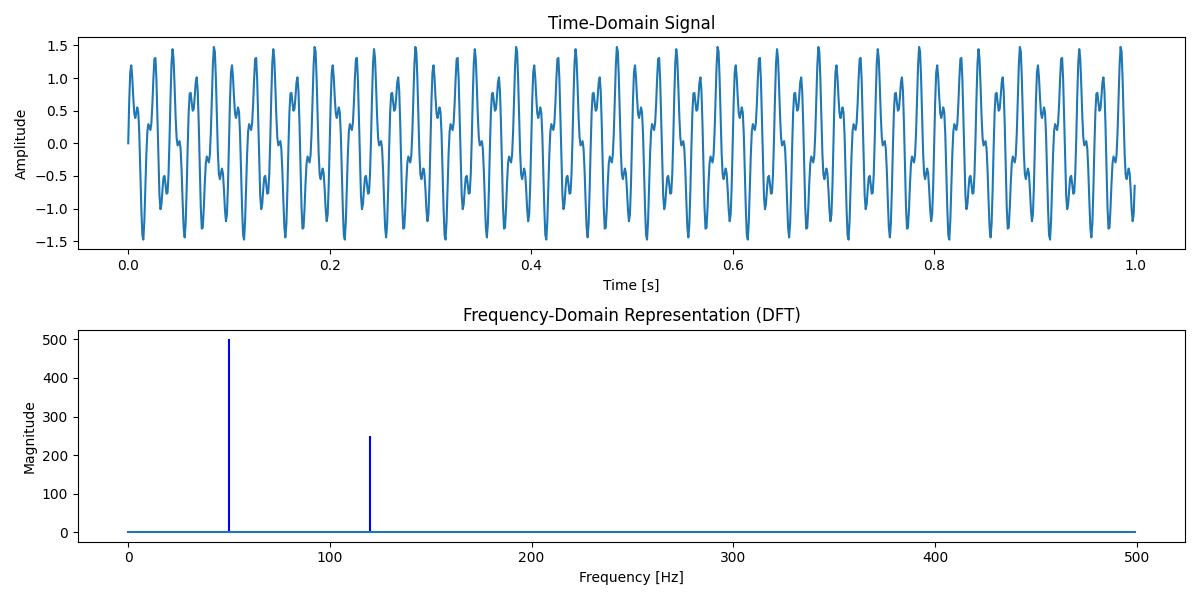
**Output :**

The output of the DFT is a discrete frequency spectrum showing the magnitude of frequency components present in the input signal. The frequency-domain plot will show prominent peaks at 50 Hz and 120 Hz, which correspond to the original frequencies of the input signal.

**Frequency-Domain Output:**

* Peak at 50 Hz (amplitude ~1)
* Peak at 120 Hz (amplitude ~0.5)

These peaks validate that the DFT correctly identified the frequency components of the original signal.



### 

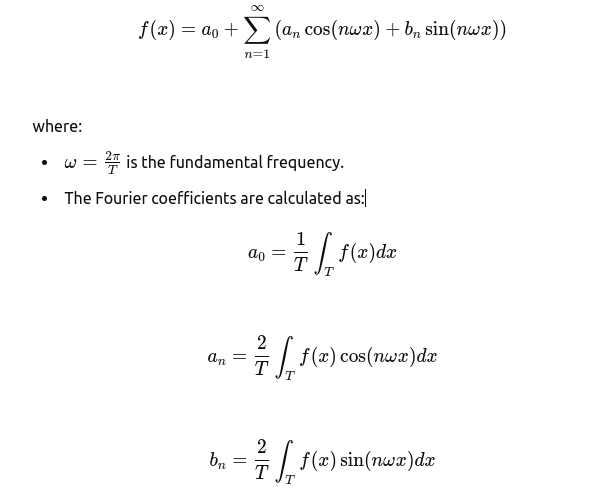
### Purpose:

The purpose of this experiment is to understand the concept of the Discrete Fourier Transform (DFT), its mathematical formulation, and its practical implementation using Python. The experiment aims to analyze how DFT converts a discrete time-domain signal into its frequency-domain representation and visualize the results.

**No. of the experiment : 08**

**Name of the experiment :** Explain and implementation of fourier series decomposition.

**Theory:** A periodic function with period can be represented as an infinite sum of sine and cosine functions:



These coefficients determine the contribution of sine and cosine components in approximating the function.

**Source Code :**

**import numpy as np**

**import matplotlib.pyplot as plt**

**from scipy.integrate import quad**

**# Define the function to approximate (square wave example)**

**def f(x):**

**return 1 if x < np.pi else -1**

**# Compute Fourier coefficients**

**def compute\_coefficients(N, T):**

**omega = 2 \* np.pi / T**

**a0 = (2 / T) \* quad(lambda x: f(x), 0, T)[0]**

**an = []**

**bn = []**

**for n in range(1, N+1):**

**an.append((2 / T) \* quad(lambda x: f(x) \* np.cos(n \* omega \* x), 0, T)[0])**

**bn.append((2 / T) \* quad(lambda x: f(x) \* np.sin(n \* omega \* x), 0, T)[0])**

**return a0, an, bn**

**# Reconstruct the function using Fourier series**

**def fourier\_series(x, a0, an, bn, N, T):**

**omega = 2 \* np.pi / T**

**result = a0 / 2**

**for n in range(1, N+1):**

**result += an[n-1] \* np.cos(n \* omega \* x) + bn[n-1] \* np.sin(n \* omega \* x)**

**return result**

**# Define parameters**

**N = 10 # Number of terms**

**T = 2 \* np.pi # Period**

**x = np.linspace(0, 2\*np.pi, 1000)**

**# Compute coefficients**

**a0, an, bn = compute\_coefficients(N, T)**

**# Compute Fourier approximation**

**y\_approx = [fourier\_series(val, a0, an, bn, N, T) for val in x]**

**# Plot the original function and the Fourier approximation**

**plt.plot(x, [f(val) for val in x], label="Original Function", linestyle='dashed')**

**plt.plot(x, y\_approx, label=f"Fourier Approximation (N={N})")**

**plt.legend()**

**plt.title("Fourier Series Decomposition")**

**plt.xlabel("x")**

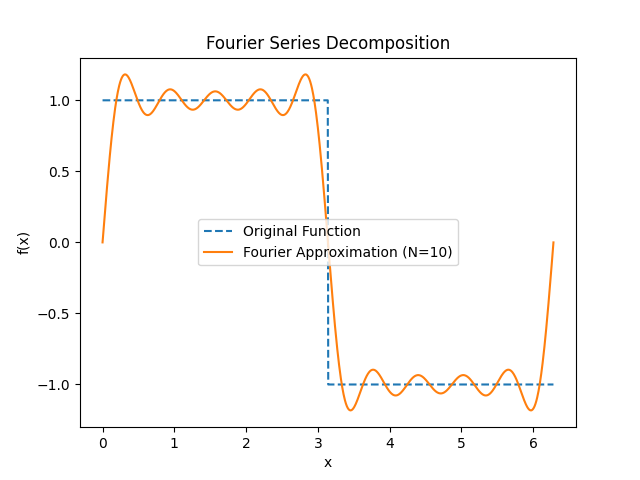
**plt.ylabel("f(x)")**

**plt.show()**

**Input:**

* A periodic function (square wave in this case) defined in the interval .
* Number of terms in the Fourier series approximation.

**Output :**

****

**Purpose:**

The purpose of this lab is to understand and implement Fourier series decomposition, which expresses a periodic function as a sum of sines and cosines. The Fourier series helps analyze and reconstruct signals in various applications, such as signal processing, physics, and engineering.