

|  |  |
| --- | --- |
| **Index** | **Title** |
| 01 | Plot the following signal operations using user defined function -   1. adding ,b. multiplication, c. Scaling, d. shifting and e. folding. |
| 02 | Explain and Implement the unit Impulse sequence, the unit Step sequence, the unit Ramp sequence. |
| 03 | Explain and Implement convolution of signal. |
| 04 | Explain and Implement correlation of signal. |
| 05 | Extract relevant features such as filtering, feature extraction, pick detection, heart rate etc. from the PPG signal. |
| 06 | Explain and Implement Discrete Fourier Transform (DFT) using python. |
| 07 | Explain and Implement Frequency bin using python. |

**Title:** Plot the following signal operations -

1. adding b. multiplication c. Scaling d. shifting and e. folding.

**Theory:**

~Addition: If x1(n) and x2(n) are two discrete-time signals, their sum is given by:

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

This is useful in combining multiple signals or adding noise

~Multiplication:

The point-wise multiplication of two signals is:

y(n)=x1(n)×x2(n)

This operation is used in modulation and filtering

~Scaling: Scaling alters the amplitude of a signal:

y(n)=a⋅x(n)

where a is a constant. Scaling is used in amplification or attenuation

~Shifting: Shifting moves a signal forward or backward along the time axis. It is mathematically represented as:

* **Right shift**: y(n)=x(n−k)y

This delays the signal by k units, which is often used in systems with time delays.

* **Left shift**: y(n)=x(n+k)

This advances the signal by k units, commonly seen in predictive systems.

Time shifting is crucial in synchronization and filtering applications.

~Signal Folding :

Folding or time reversal flips the signal around the vertical axis, reversing its time indices:

y(n)=x(−n)

This operation is widely used in convolution, symmetry analysis, and image processing.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

def signal\_addition(x1, x2):

return x1 + x2

def signal\_multiplication(x1, x2):

return x1 \* x2

def signal\_scaling(x, alpha):

return alpha \* x

def signal\_shifting(n, shift):

return n + shift

def signal\_folding(x):

return np.flip(x)

n = np.array([-2, -1, 0, 1, 2])

x1 = np.array([1, 2, 3, 4, 5])

x2 = np.array([5, 4, 3, 2, 1])

added\_signal = signal\_addition(x1, x2)

multiplied\_signal = signal\_multiplication(x1, x2)

scaled\_signal = signal\_scaling(x1, 2)

shifted\_n1 = signal\_shifting(n, -2)

shifted\_n2 = signal\_shifting(n, 2)

folded\_signal = signal\_folding(x1)

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

plt.subplot(4, 2, 1)

plt.stem(n, x1)

plt.title("Original Signal x1")

plt.subplot(4, 2, 2)

plt.stem(n, x2)

plt.title("Original Signal x2")

plt.subplot(4, 2, 3)

plt.stem(n, added\_signal)

plt.title("Signal Addition")

plt.subplot(4, 2, 4)

plt.stem(n, multiplied\_signal)

plt.title("Signal Multiplication")

plt.subplot(4, 2, 5)

plt.stem(n, scaled\_signal)

plt.title("Signal Scaling")

plt.subplot(4, 2, 6)

plt.stem(shifted\_n1, x1) # Shifted time indices

plt.title("Signal Shifting (-2)")

plt.subplot(4, 2, 7)

plt.stem(shifted\_n2, x1) # Shifted time indices

plt.title("Signal Shifting (+2)")

plt.subplot(4, 2, 8)

plt.stem(n, folded\_signal)

plt.title("Signal Folding")

plt.tight\_layout()

plt.show()

**Input:**

Time index:

n=[−2,−1,0,1,2]

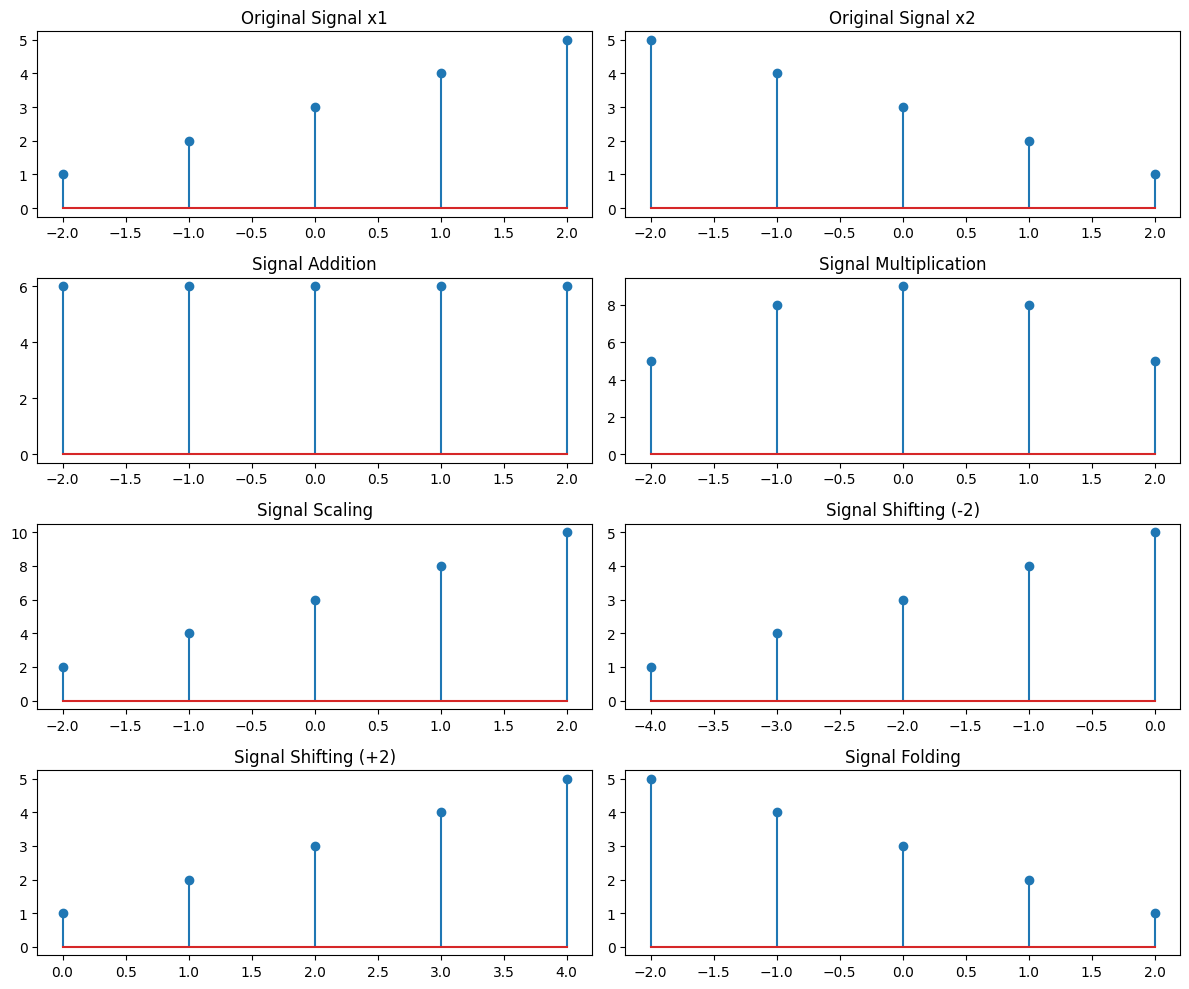
Signal **x1**:

x1=[1,2,3,4,5]

Signal **x2**:

x2=[5,4,3,2,1]

**Output:**



**Purpose:**

The purpose of this experiment is to understand and implement fundamental **signal operations** using Python. These operations—**addition, multiplication, scaling, shifting, and folding**—are essential in **digital signal processing (DSP)** and play a crucial role in various real-world applications, such as:

* **Audio and Speech Processing:** Noise cancellation, equalization, and compression.
* **Image Processing:** Contrast adjustment, edge detection, and filtering.
* **Communication Systems:** Signal modulation, encoding, and transmission.
* **Biomedical Signal Processing:** ECG and EEG analysis.

By visualizing these operations, we gain a deeper understanding of how signals behave under transformations and how they can be manipulated for practical applications.

**Title:** Explain and Implementation the unit Impulse sequence, the unit Step sequence, the unit Ramp sequence.

**Theory:** In signal and system analysis, certain fundamental sequences play a crucial role in representing and analyzing signals. Among these, the **Unit Impulse Sequence, Unit Step Sequence, and Unit Ramp Sequence** are widely used as basic building blocks. These sequences help in understanding system behavior, designing filters, and solving differential and difference equations.

**1. Unit Impulse Sequence**

The unit impulse sequence is a special discrete-time signal that exists only at a single point in time and is zero everywhere else. It acts as a short pulse or an instant energy burst. This sequence is useful in analyzing how systems respond to sudden inputs.

**2. Unit Step Sequence**

The unit step sequence represents a signal that switches on at a particular point in time and stays on indefinitely. It is widely used in DSP and control systems to model signals that begin at a certain time and continue without stopping.

**3. Unit Ramp Sequence**

The unit ramp sequence represents a gradually increasing signal. It starts at zero and grows steadily over time. This sequence is commonly used in modeling physical systems where a value increases with time, such as velocity increasing due to constant acceleration.

**Source code:**

import numpy as np

import matplotlib.pyplot as plt

n = np.arange(-10, 11)

impulse = (n == 0).astype(int)

step = (n >= 0).astype(int)

ramp = np.maximum(n, 0)

plt.figure(figsize=(12, 4))

plt.subplot(1, 3, 1)

plt.stem(n, impulse)

plt.title("Unit Impulse Sequence")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.subplot(1, 3, 2)

plt.stem(n, step)

plt.title("Unit Step Sequence")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.subplot(1, 3, 3)

plt.stem(n, ramp)

plt.title("Unit Ramp Sequence")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.tight\_layout()

plt.show()

Input:

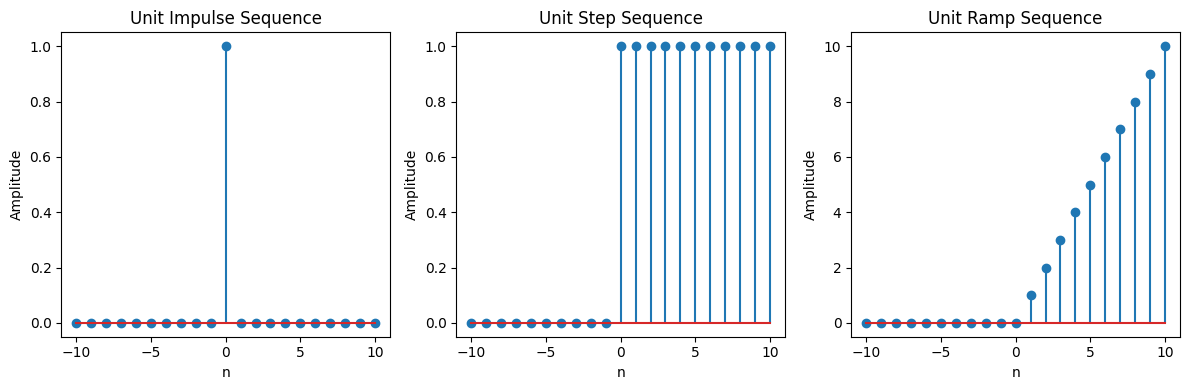
n = np.arange(-10, 11)

impulse = (n == 0).astype(int)

step = (n >= 0).astype(int)

ramp = np.maximum(n, 0)

**Output:**



**Purpose:** The purpose of this lab is to analyze and implement three fundamental discrete-time signals: **Unit Impulse Sequence, Unit Step Sequence, and Unit Ramp Sequence**. These signals are widely used in **Digital Signal Processing (DSP)**, system modeling, and control systems.

1. **Understand the characteristics of basic discrete-time signals:**
2. **Implement and visualize these signals using Python (Matplotlib).**
3. **Analyze how these signals behave and their role in system response analysis.**
4. **Understand real-world applications, such as:**
   * Impulse response in system analysis.
   * Step response in control systems.
   * Ramp signals in modeling gradual changes (e.g., velocity increase)

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

**Theory:** Convolution is a fundamental mathematical operation used in **Signal and System** to determine the output of a system when an input signal is applied. It describes how a signal interacts with a system's impulse response.

Convolution is widely used in **signal processing, image processing, and system analysis** to filter signals, analyze system behavior, and design digital filters.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import convolve

def compute\_convolution(signal1, signal2):

conv\_result = convolve(signal1, signal2, mode='full', method='auto')

return conv\_result

fs = 1000

t = np.linspace(0, 1, fs, endpoint=False)

freq = 5

# Generate a sine wave signal

sin\_signal = np.sin(2 \* np.pi \* freq \* t)

conv\_auto = compute\_convolution(sin\_signal, sin\_signal)

# Convolution between signal and shifted version

signal1 = sin\_signal

signal2 = np.roll(signal1, 100)

conv\_shifted = compute\_convolution(signal1, signal2)

# Convolution with noisy signal

noise = np.random.normal(0, 0.5, fs)

noisy\_signal = signal1 + noise

conv\_noisy = compute\_convolution(signal1, noisy\_signal)

# Plot the results

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

# Plot autoconvolution

plt.subplot(3, 1, 1)

plt.plot(conv\_auto)

plt.title("Autoconvolution of a Sinusoidal Signal")

plt.xlabel("Samples")

plt.ylabel("Convolution Output")

plt.grid()

# Plot convolution with shifted signal

plt.subplot(3, 1, 2)

plt.plot(conv\_shifted)

plt.title("Convolution between Signal and Shifted Version")

plt.xlabel("Samples")

plt.ylabel("Convolution Output")

plt.grid()

# Plot convolution with noisy signal

plt.subplot(3, 1, 3)

plt.plot(conv\_noisy)

plt.title("Convolution with Noisy Signal")

plt.xlabel("Samples")

plt.ylabel("Convolution Output")

plt.grid()

plt.tight\_layout()

plt.show()

Input:

1. **Sinusoidal Signal (sin\_signal):**

* A sine wave with **frequency = 5 Hz**
* Sampled at **fs = 1000 Hz**
* Time range: **0 to 1 second**

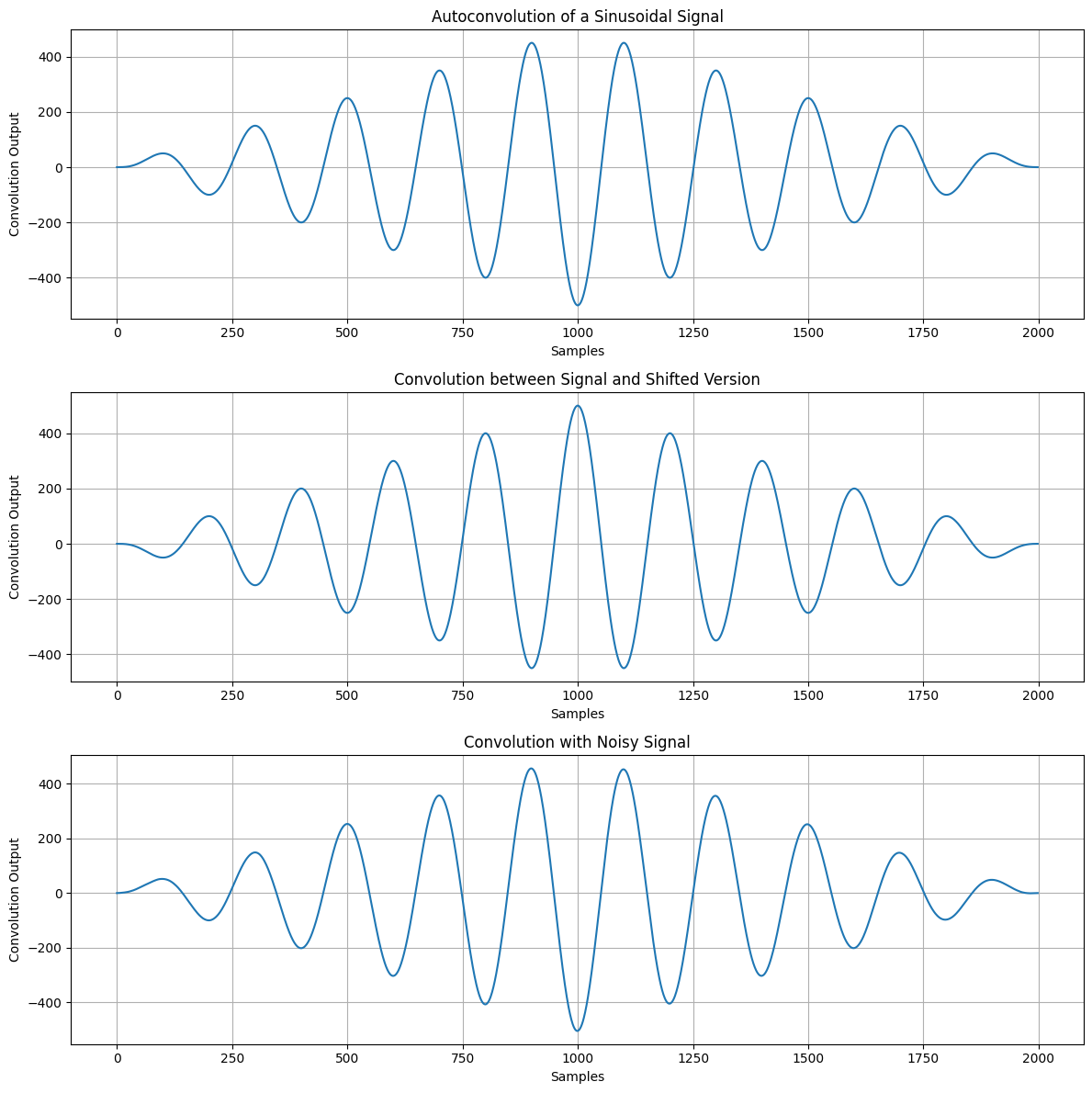
2. **Shifted Signal (signal2):**

* The same sine wave, but **shifted by 100 samples** using np.roll().

3. **Noisy Signal (noisy\_signal):**

* The sine wave with added **random Gaussian noise** (mean = 0, standard deviation = 0.5).

**Output:**



**Purpose:**

The purpose of this lab is to **understand and implement discrete-time convolution** using Python. Through this experiment, students will:

1. **Learn how convolution works** and its role in signal processing.
2. **Implement convolution using Python’s np.convolve() function.**
3. **Visualize input signals, impulse response, and convolution output.**
4. **Understand applications of convolution in filtering, system analysis, and image processing.**

**Title:** Explain and Implement correlation of signal

Theory: Correlation is a mathematical operation used to quantify the relationship between two signals. It is widely used in signal processing to determine how similar two signals are or to detect shifts in one signal relative to another. The primary types of correlation are **cross-correlation** and **auto-correlation**.

**1. Cross-Correlation**

Cross-correlation is used to measure the similarity between two signals as a function of the time-lag applied to one of the signals. It is used to detect if one signal is a delayed version of another and to quantify the time shift at which the signals are most similar.

**2. Auto-Correlation**

Auto-correlation is a special case of cross-correlation, where the signal is correlated with itself. It is used to identify periodicity or repeating patterns within a single signal.

**Source Code:**

import matplotlib.pyplot as plt

from scipy.signal import correlate, correlation\_lags

import numpy as np

# Function to compute autocorrelation of a signal

def compute\_autocorrelation(signal):

auto\_corr = correlate(signal, signal, mode='full', method='auto')

lags = correlation\_lags(len(signal), len(signal), mode='full')

return auto\_corr / np.max(auto\_corr), lags

# Function to compute cross-correlation between two signals

def compute\_crosscorrelation(signal1, signal2):

cross\_corr = correlate(signal1, signal2, mode='full', method='auto')

lags = correlation\_lags(len(signal1), len(signal2), mode='full')

return cross\_corr / np.max(cross\_corr), lags

# Parameters

fs = 1000 # Sampling frequency

t = np.linspace(0, 1, fs, endpoint=False) # Time vector

freq = 5 # Frequency of the sinusoidal signal

# Generate sinusoidal signal

sin\_signal = np.sin(2 \* np.pi \* freq \* t)

# Compute and plot autocorrelation of the sinusoidal signal

auto\_corr, lags = compute\_autocorrelation(sin\_signal)

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

plt.plot(lags, auto\_corr, label="Autocorrelation", color='b')

plt.axvline(0, color='red', linestyle='--', label="Lag=0")

plt.title("Autocorrelation of Sinusoidal Signal")

plt.xlabel("Lags")

plt.ylabel("Normalized Autocorrelation")

plt.grid()

plt.legend()

plt.show()

# Create a shifted version of the sinusoidal signal for cross-correlation

signal1 = np.sin(2 \* np.pi \* freq \* t)

signal2 = np.roll(signal1, 100)

# Compute and plot cross-correlation between the original and shifted signals

cross\_corr, lags = compute\_crosscorrelation(signal1, signal2)

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

plt.plot(lags, cross\_corr, label="Cross-correlation", color='g')

plt.axvline(0, color='red', linestyle='--', label="Lag=0")

plt.title("Cross-Correlation between Two Signals")

plt.xlabel("Lags")

plt.ylabel("Normalized Cross-Correlation")

plt.grid()

plt.legend()

plt.show()

# Add noise to the original signal

noise = np.random.normal(0, 0.5, fs)

noisy\_signal = signal1 + noise

# Compute and plot cross-correlation between the original signal and the noisy signal

cross\_corr\_noise, lags = compute\_crosscorrelation(signal1, noisy\_signal)

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

plt.plot(lags, cross\_corr\_noise, label="Cross-Correlation with Noise", color='purple')

plt.axvline(0, color='red', linestyle='--', label="Lag=0")

plt.title("Cross-Correlation with Noisy Signal")

plt.xlabel("Lag")

plt.ylabel("Noisy Signal")

plt.grid()

plt.legend()

plt.show()

Input:

1. **Signal 1 (Original Signal):**

* **Type:** Sinusoidal wave.
* **Equation:** signal1(t)=sin⁡(2πft)\text{signal1}(t) = \sin(2 \pi f t)signal1(t)=sin(2πft)
* **Frequency:** f=5 Hzf = 5 \, \text{Hz}f=5Hz
* **Sampling Frequency:** fs=1000 Hzf\_s = 1000 \, \text{Hz}fs​=1000Hz (samples per second).
* **Time Duration:** 1 second, resulting in 1000 samples.

The original signal is a continuous sine wave with a frequency of 5 Hz, sampled at 1000 Hz for a total duration of 1 second.

2. **Signal 2 (Shifted Signal):**

* **Type:** A shifted version of Signal 1.
* **Transformation:** signal2(t)=roll(signal1(t),100)\text{signal2}(t) = \text{roll}(\text{signal1}(t), 100)signal2(t)=roll(signal1(t),100)
* Signal 2 is created by shifting **Signal 1** by 100 samples. This represents a time shift in the signal.

3. **Noise:**

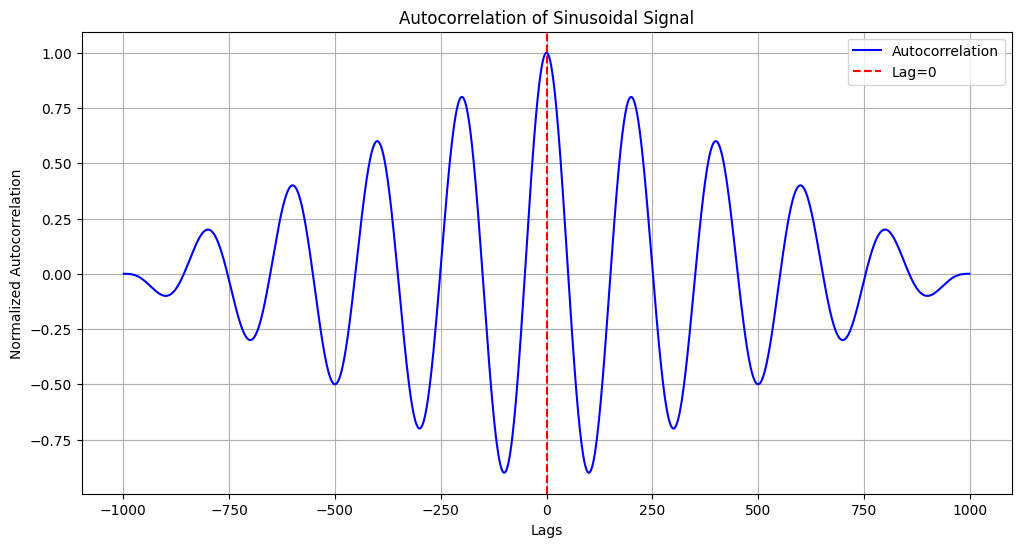
* **Type:** Gaussian (normal) noise.
* **Mean:** 0.
* **Standard Deviation:** σ=0.5\sigma = 0.5σ=0.5.
* **Length:** Same as the length of Signal 1 (1000 samples).

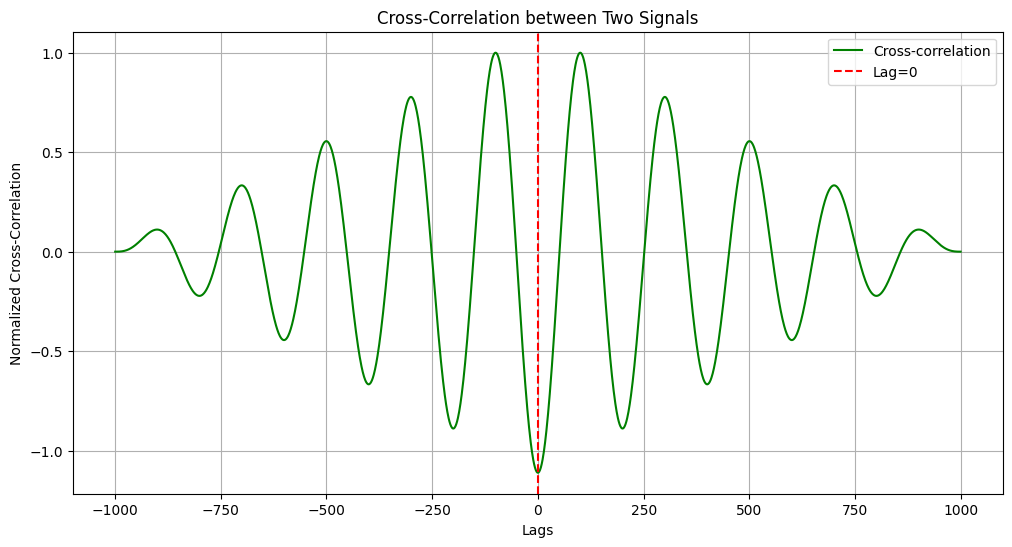
Noise is added to Signal 1 to create a **noisy signal**, representing a real-world scenario where noise corrupts the signal.

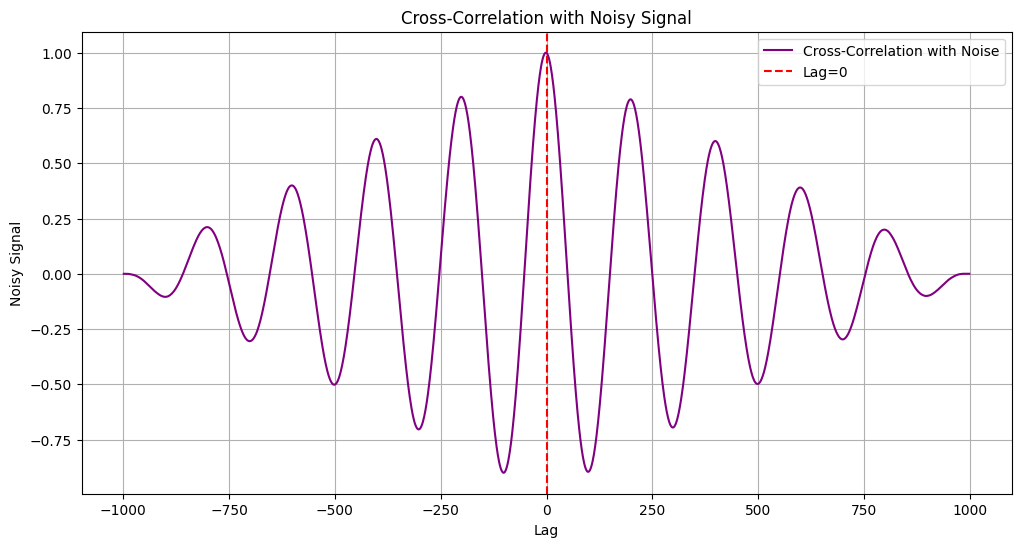
4. **Time Vector:**

* **Range:** t=[0,1)t = [0, 1)t=[0,1) seconds.
* **Samples:** 1000 evenly spaced points in the time interval, generated using t=np.linspace(0,1,1000,endpoint=False)\text{t} = \text{np.linspace}(0, 1, 1000, \text{endpoint=False})t=np.linspace(0,1,1000,endpoint=False).

Output:







In [ ]:

**Purpose**: The primary purpose of this experiment is to analyze the **autocorrelation** and **cross-correlation** of signals and understand their behavior in the presence of noise. Specifically, the objectives of this experiment are:

1. **To calculate and analyze the autocorrelation of a sinusoidal signal**:
   * Autocorrelation is a mathematical tool used to measure the similarity between a signal and a delayed version of itself. By calculating the autocorrelation of a sinusoidal signal, we aim to observe the periodicity and how the signal is related to its time-shifted versions.
2. **To compute the cross-correlation between two signals**:
   * Cross-correlation measures the similarity between two signals as a function of the time shift applied to one of the signals. By analyzing the cross-correlation of a clean and shifted sinusoidal signal, we can understand how well the two signals are correlated at different lags.
3. **To investigate the effect of noise on signal correlation**:
   * Noise is commonly encountered in real-world signals, and this experiment aims to assess how the introduction of Gaussian noise affects the autocorrelation and cross-correlation results. By adding noise to a signal and comparing the cross-correlation with the original, we can study how noise distorts the correlation and reduces the signal's similarity.
4. **To explore the relationship between the lag and the correlation coefficient**:
   * By analyzing the lag values at which the highest correlation occurs, the experiment will demonstrate how time shifts impact the correlation and help establish the significance of lags in both autocorrelation and cross-correlation functions.

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

Theory: **Discrete Fourier Transform (DFT)**

The **Discrete Fourier Transform (DFT)** is a mathematical technique used to convert a discrete-time signal from the time domain to the frequency domain. It analyzes the signal in terms of its frequency components.

**Inverse Discrete Fourier Transform (IDFT)**

The **Inverse Discrete Fourier Transform (IDFT)** is used to convert the signal from the frequency domain back to the time domain.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

# Define the input signal

x = [1, 1, 1, 1]

N = 4

# Zero padding (if necessary)

x = np.pad(x, (0, N - len(x)), mode='constant')

# Calculate DFT

X = np.fft.fft(x, N)

# Calculate IDFT

x\_reconstructed = np.fft.ifft(X).real # Show only the real part of the IDFT result

# Print the DFT and reconstructed signal values

print("DFT values:", X)

print("Reconstructed IDFT values:", x\_reconstructed)

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

plt.subplot(3, 1, 1)

plt.stem(x)

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

plt.xlabel('n')

plt.ylabel('x(n)')

plt.grid()

# Plot the magnitude of the DFT |X(k)|

plt.subplot(3, 1, 2)

plt.stem(np.abs(X))

plt.title('Magnitude of DFT |X(k)|')

plt.xlabel('k')

plt.ylabel('|X(k)|')

plt.grid()

# Plot the reconstructed signal from IDFT

plt.subplot(3, 1, 3)

plt.stem(x\_reconstructed)

plt.title('Reconstructed Signal from IDFT')

plt.xlabel('n')

plt.ylabel('x(n)')

plt.grid()

plt.tight\_layout()

plt.show()

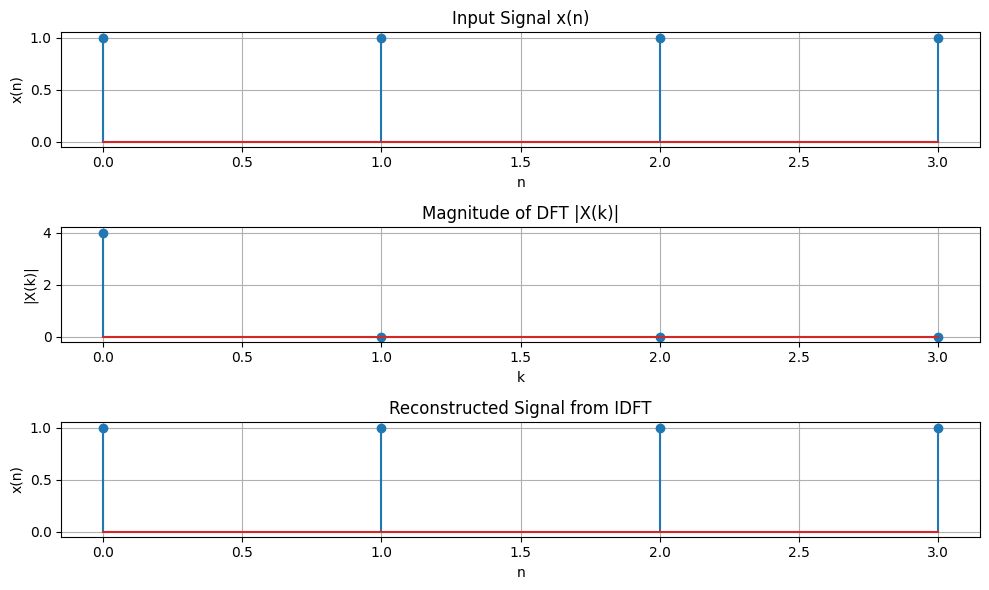
**Input:**

1. **Signal Definition:**
   * **Signal Type:** Discrete-time signal.
   * **Signal Values:** x[n]=[1,1,1,1]x[n] = [1, 1, 1, 1]x[n]=[1,1,1,1].
   * The signal is defined as a simple sequence of ones, representing a constant signal.
   * **Length of Signal:** N=4N = 4N=4, meaning the signal consists of 4 data points.
2. **Zero Padding:**
   * The signal has already the required length of N=4N = 4N=4, so no further zero-padding is applied. If necessary, zero padding would be used to make the signal length equal to NNN.
3. **DFT Calculation:**
   * The **Discrete Fourier Transform (DFT)** is calculated using the Fast Fourier Transform (FFT) algorithm provided by the NumPy library, which transforms the time-domain signal into the frequency-domain representation.
4. **IDFT Calculation:**
   * The **Inverse Discrete Fourier Transform (IDFT)** is calculated using the inverse FFT function. This function reconstructs the original time-domain signal from the frequency-domain representation.
5. **Sampling Frequency and Signal Duration:**
   * The signal consists of 4 discrete samples and is not explicitly associated with a time or frequency domain range, as the signal is represented directly in its discrete form.

**Output:**

DFT values: [4.+0.j 0.+0.j 0.+0.j 0.+0.j]

Reconstructed IDFT values: [1. 1. 1. 1.]



**Purpose:** The purpose of this experiment is to understand and demonstrate the concepts of **Discrete Fourier Transform (DFT)** and **Inverse Discrete Fourier Transform (IDFT)**. The specific objectives of this experiment are:

1. **To Compute the DFT of a Discrete Signal:**
   * Transform a discrete-time signal from the time domain into the frequency domain using DFT. This helps to analyze the frequency components of the signal and understand how each frequency contributes to the signal.
2. **To Compute the IDFT of the DFT Result:**
   * Convert the frequency-domain signal back to the time domain using IDFT. This demonstrates that DFT and IDFT are inverses of each other, as the original signal is reconstructed from its frequency-domain representation.
3. **To Analyze the Relationship Between Time and Frequency Domains:**
   * Explore how a time-domain signal is transformed into its frequency-domain representation, and how the inverse transformation reconstructs the original signal. This process helps in understanding the duality between the time and frequency domains.
4. **To Study the Practical Applications of DFT and IDFT:**
   * Understand the importance of DFT and IDFT in signal processing, communication systems, audio processing, and other applications where the analysis and synthesis of signals are required.

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

**Theory:** A **frequency bin** refers to a range of frequencies that a signal is divided into. When performing a **Fourier Transform** (such as FFT), the result is often a spectrum of values representing the amplitudes of different frequency components. These frequency components are typically grouped into frequency bins, where each bin corresponds to a specific frequency range.

In a Fast Fourier Transform (FFT), the result gives us the frequency components of the signal. These are typically represented in discrete bins. The width of each frequency bin is determined by the sampling rate and the number of points in the FFT.

Source code:

import numpy as np

# Define parameters

N = 1024 # Number of points in DFT

Fs = 1000 # Sampling frequency (Hz)

# Compute frequency bins manually

freq\_bins\_manual = np.array([(k / N) \* Fs for k in range(N)])

# Print first 10 frequency bins

print("Manual Frequency Bins (first 10):")

print(freq\_bins\_manual[:10])

import numpy as np

# Define parameters

N = 1024 # Number of points in DFT

Fs = 1000 # Sampling frequency (Hz)

# Compute frequency bins using NumPy's fftfreq

freq\_bins\_fft = np.fft.fftfreq(N, d=1/Fs)

# Print first 10 frequency bins

print("Frequency Bins using np.fft.fftfreq (first 10):")

print(freq\_bins\_fft[:10])

**Input:**

Enter the number of points in DFT (e.g., 1024): 1024

Enter the sampling frequency (Hz, e.g., 1000): 1000

Output:

Manual Frequency Bins (first 10):

[0. 0.9765625 1.953125 2.9296875 3.90625 4.8828125 5.859375

6.8359375 7.8125 8.7890625]

Frequency Bins using np.fft.fftfreq (first 10):

[0. 0.9765625 1.953125 2.9296875 3.90625 4.8828125 5.859375

6.8359375 7.8125 8.7890625]

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

Theory: The **Photoplethysmogram (PPG)** is a non-invasive optical technique that detects blood volume changes in the microvascular bed of tissue. PPG signals are widely used in wearable devices, clinical monitoring systems, and healthcare applications for measuring physiological parameters such as heart rate, blood oxygen saturation, and respiratory rate.

The raw PPG signal, however, is often noisy due to artifacts such as motion, ambient light changes, and sensor noise. Therefore, filtering, feature extraction, peak detection, and heart rate estimation are critical steps in processing and analyzing the PPG signal.

In this report, we focus on the following tasks:

* **Filtering**: Removing noise and unwanted frequency components.
* **Feature Extraction**: Identifying important characteristics of the PPG signal.
* **Peak Detection**: Detecting the peaks (often corresponding to heartbeats).
* **Heart Rate Calculation**: Estimating the heart rate from the PPG signal.

Source Code:

import numpy as np

import scipy.signal as signal

import matplotlib.pyplot as plt

from scipy.signal import find\_peaks

# Function for Bandpass Filtering the PPG Signal

def bandpass\_filter(ppg\_signal, fs, low\_cutoff=0.5, high\_cutoff=4):

nyquist = 0.5 \* fs

low = low\_cutoff / nyquist

high = high\_cutoff / nyquist

b, a = signal.butter(1, [low, high], btype='band')

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

return filtered\_signal

# Function to Detect Peaks

def detect\_peaks(filtered\_signal, min\_peak\_height=0.2, min\_distance=300):

peaks, \_ = find\_peaks(filtered\_signal, height=min\_peak\_height, distance=min\_distance)

return peaks

# Function to Estimate Heart Rate

def estimate\_heart\_rate(peaks, time\_stamps):

peak\_intervals = np.diff(time\_stamps[peaks])

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

return heart\_rate

# Function to Extract Features from the Signal

def extract\_features(filtered\_signal):

mean\_amplitude = np.mean(filtered\_signal)

rms\_amplitude = np.sqrt(np.mean(filtered\_signal\*\*2))

return mean\_amplitude, rms\_amplitude

# Main Program

def main():

# Simulated PPG signal (replace with actual data)

fs = 1000

t = np.linspace(0, 60, 60\*fs)

ppg\_signal = np.sin(2 \* np.pi \* 1 \* t) + 0.3 \* np.sin(2 \* np.pi \* 0.3 \* t)

# Step 1: Bandpass filter the PPG signal

filtered\_ppg = bandpass\_filter(ppg\_signal, fs)

# Step 2: Detect peaks in the filtered signal

peaks = detect\_peaks(filtered\_ppg)

# Step 3: Estimate Heart Rate (bpm)

heart\_rate = estimate\_heart\_rate(peaks, t)

# Step 4: Extract Features (Mean Amplitude and RMS)

mean\_amplitude, rms\_amplitude = extract\_features(filtered\_ppg)

# Output Results

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

print(f"Mean Amplitude: {mean\_amplitude:.2f}")

print(f"RMS Amplitude: {rms\_amplitude:.2f}")

# Step 5: Plot the results

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

# Plot the original PPG signal

plt.subplot(2, 1, 1)

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

plt.title("Original PPG Signal")

plt.xlabel("Time (s)")

plt.ylabel("Amplitude")

# Plot the filtered PPG signal with detected peaks

plt.subplot(2, 1, 2)

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

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

plt.title("Filtered PPG Signal with Detected Peaks")

plt.xlabel("Time (s)")

plt.ylabel("Amplitude")

plt.legend()

plt.tight\_layout()

plt.show()

# Run the main program

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Input:**

Enter the sampling frequency (Hz, e.g., 1000): 1000

Enter the duration of the PPG signal in seconds (e.g., 60): 60

Enter the low cutoff frequency for bandpass filter (e.g., 0.5 Hz): 0.5

Enter the high cutoff frequency for bandpass filter (e.g., 4 Hz): 4

Enter the minimum peak height for detection (e.g., 0.2): 0.2

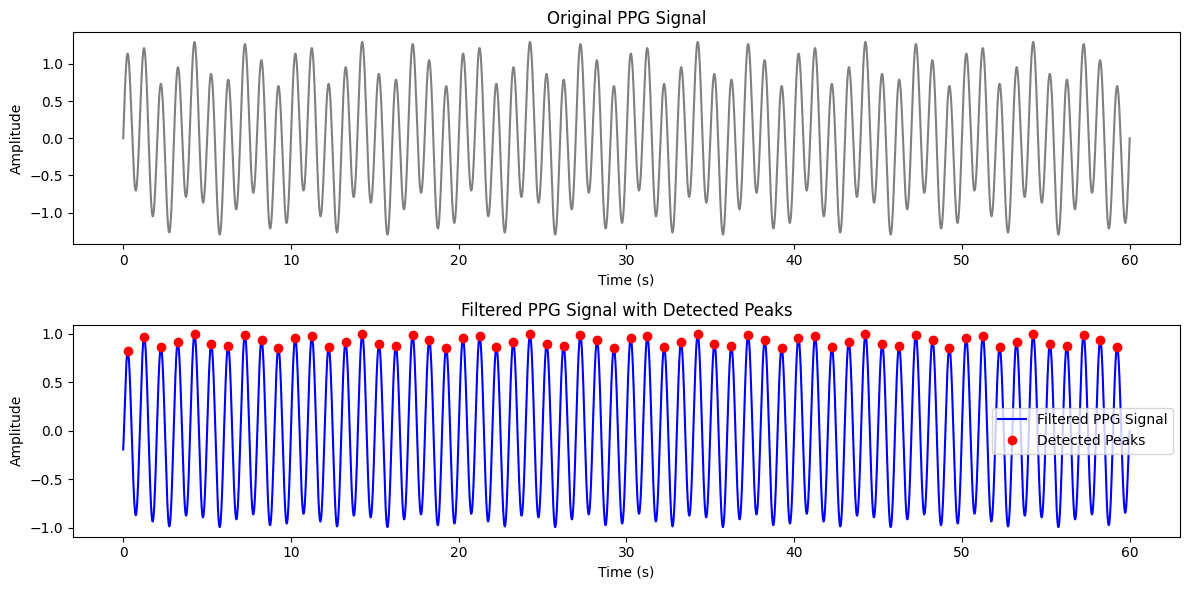
Enter the minimum distance between peaks (e.g., 300): 300

Output:

Estimated Heart Rate: 60.02 bpm

Mean Amplitude: -0.00

RMS Amplitude: 0.65



In [ ]:

**Purpose:** The purpose of this experiment is to demonstrate how to extract relevant features from a **Photoplethysmogram (PPG)** signal, which is commonly used to monitor vital signs like heart rate, blood oxygen saturation, and more. Specifically, the tasks aim to:

1. **Preprocess the PPG signal** by applying **bandpass filtering** to remove noise and unwanted frequency components.
2. **Detect peaks** in the filtered PPG signal, which represent **heartbeats** or systolic blood volume changes.
3. **Estimate the heart rate** by calculating the time intervals between detected peaks, allowing us to derive the beats per minute (bpm) of the subject.
4. **Extract additional features** such as **mean amplitude** and **root mean square (RMS) amplitude** to further characterize the PPG signal and potentially use these features for health monitoring.

The overall aim is to accurately process PPG data, detect heartbeats, and estimate heart rate, which can be useful in various applications, including:

* **Wearable health devices**: Continuous heart rate monitoring.
* **Clinical applications**: Real-time health status tracking.
* **Personal fitness**: Heart rate monitoring during exercise or rest.