

|  |  |
| --- | --- |
| **Lab No.** | **Lab Name** |
| **1** | **Signal Sequence (impuls, step, Ramp)** |
| **2** | **Signal Operation(add, shift, folding)** |
| **3** | **Correlation of signal** |
| **4** | **Convolution of Signal** |
| **5** | **PPG signal (filtering, feature extraction, Peak detection)** |
| **6** | **Implementation of Discrete Fourier Transform(DFT) and Inverse Discrete Fourier Transform(IDFT)** |
| **7** | **Explain and Implement Frequency bin using python.** |

**INDEX**

**Lab 1: Signal Sequence (impuls, step, Ramp)**

**Objectives:**

1. **Understand the basic types of signals**: Learn about impulse, step, and ramp signals and their characteristics.
2. **Analyze the properties of signals**: Investigate how each signal behaves with respect to time and amplitude.
3. **Learn signal generation techniques**: Learn how to generate impulse, step, and ramp signals using simulation software (e.g., MATLAB).
4. **Understand the application of these signals in systems**: Explore how impulse, step, and ramp signals are used to test and analyze dynamic systems in control theory and signal processing.
5. **Visualize the signals**: Graphically represent the impulse, step, and ramp signals and observe their responses.

**Theory:**

**1. Impulse Signal (δ(t)):**

* The impulse signal, denoted as δ(t), represents a sudden, instantaneous change at a specific point in time, typically at **t = 0**. It is a mathematical abstraction used to analyze system behavior under a very rapid change in input. The impulse signal has infinite amplitude at t = 0 but zero amplitude at all other times. It is often used to determine the **impulse response** of a system.

**2. Step Signal (u(t)):**

* The step signal, denoted as u(t), is a signal that starts at 0 and jumps to a constant value (usually 1) at **t = 0**. It represents a sudden shift or change in a system's input. Step signals are widely used to study system responses to a constant input over time and analyze system stability and behavior. It is also used to test how a system responds to a sustained, steady change.

**3. Ramp Signal (r(t)):**

* The ramp signal, denoted as r(t), is a signal that increases linearly with time. It starts at 0 and continuously increases at a constant rate (usually a slope of 1). The ramp signal is used to simulate gradually increasing inputs, such as a steadily growing load. It helps analyze system performance when subjected to linearly increasing input over time.

import numpy as n

import matplotlib.pyplot as plt

# Define the range

n = np.arange(-10, 11)

def impulse\_signal(n):

return np.where(n == 0, 1, 0)

def step\_signal(n):

return np.where(n >= 0, 1, 0)

def ramp\_signal(n):

return np.where(n >= 0, n, 0)

# Generate signals

impulse = impulse\_signal(n) step = step\_signal(n)

ramp = ramp\_signal(n)

# Plot signals plt.figure(figsize=(12, 4))

plt.subplot(1, 3, 1) plt.stem(n, impulse) plt.title("Impulse Signal") plt.xlabel("n") plt.ylabel("Amplitude") plt.grid()

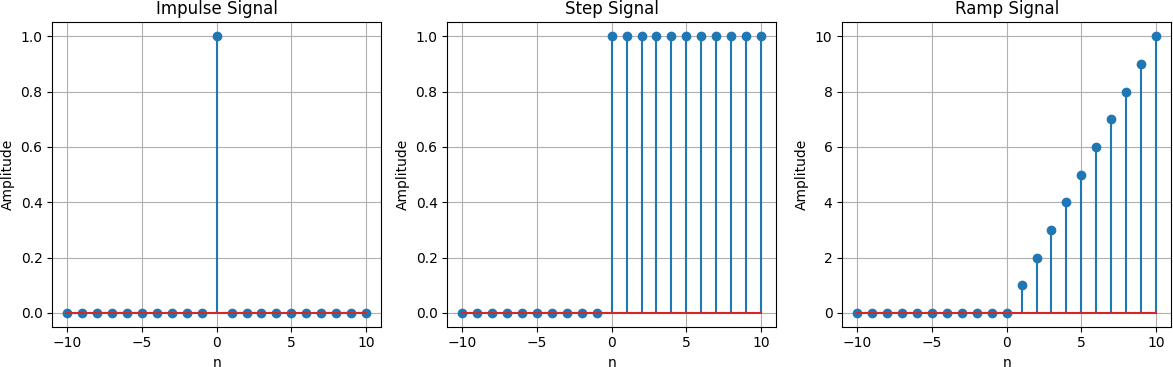
plt.subplot(1, 3, 2) plt.stem(n, step) plt.title("Step Signal") plt.xlabel("n")

plt.ylabel("Amplitude") plt.grid()

plt.subplot(1, 3, 3) plt.stem(n, ramp) plt.title("Ramp Signal") plt.xlabel("n") plt.ylabel("Amplitude") plt.grid()

plt.tight\_layout() plt.show()

Output :



**Lab 2: Signal Operation(add, shift, folding)**

**Objectives:**

1. **Understand signal operations**: Learn how basic signal operations such as addition, shifting, and folding affect signals.
2. **Perform signal addition**: Investigate how the combination of two or more signals is carried out by adding their amplitudes at each point in time.
3. **Understand signal shifting**: Study the effect of shifting signals in time, either to the right or left, and how this affects their behavior.
4. **Analyze signal folding**: Learn how signal folding (or reflection) alters the signal's properties and observe its impact.
5. **Visualize signal operations**: Graphically represent the results of the operations (addition, shifting, folding) and analyze their effects on signals.

**Theory:**

**1. Signal Addition:**

* Signal addition involves combining two or more signals by adding their corresponding values at each point in time. This operation is commonly used in signal processing to combine multiple inputs and analyze the resultant signal's behavior.
* **Mathematical Representation**: If **x(t)** and **y(t)** are two signals, the addition of these signals is given by: **z(t)=x(t)+y(t)**
* This operation is often used in systems where multiple inputs interact, and the resulting signal represents the sum of the effects.

**2. Signal Shifting:**

* Signal shifting refers to moving a signal in time, either to the left or right along the time axis. A positive shift moves the signal to the right (delayed), while a negative shift moves the signal to the left (advanced).
* **Mathematical Representation**:
  + A right shift of T units: x(t−T)
  + A left shift of T units: x(t+T)
* Shifting is often used to model delays in systems or to study the effects of time-varying signals.

**3. Signal Folding (Reflection):**

* Signal folding refers to reflecting the signal across the vertical axis (time axis). The signal is flipped, such that its values are reversed in time.
* **Mathematical Representation**: If x(t) is a signal, its folded version is represented as: x(−t)
* Folding is used in systems where a signal undergoes a time reversal, and it's important for analyzing certain types of waveforms or in system modeling (e.g., convolution).

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\_signal1 = signal\_shifting(n, -2) shifted\_signal2 = signal\_shifting(n, 2) folded\_signal = signal\_folding(x1)

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

plt.subplot(4, 2, 1) plt.stem(n, x1) plt.xlabel("Time") plt.ylabel("Amplitude") plt.title("Original Signal x1") plt.grid()

plt.subplot(4, 2, 2) plt.stem(n, x2) plt.xlabel("Time ") plt.ylabel("Amplitude") plt.title("Original Signal x2") plt.grid()

plt.subplot(4, 2, 3) plt.stem(n, added\_signal)

plt.xlabel("Time") plt.ylabel("Amplitude") plt.title("Signal Addition") plt.grid()

plt.subplot(4, 2, 4) plt.stem(n, multiplied\_signal) plt.xlabel("Time") plt.ylabel("Amplitude")

plt.title("Signal Multiplication") plt.grid()

plt.subplot(4, 2, 5) plt.stem(n, scaled\_signal) plt.xlabel("Time") plt.ylabel("Amplitude")

plt.title("Scaled Signal (x1 \* 2)") plt.grid()

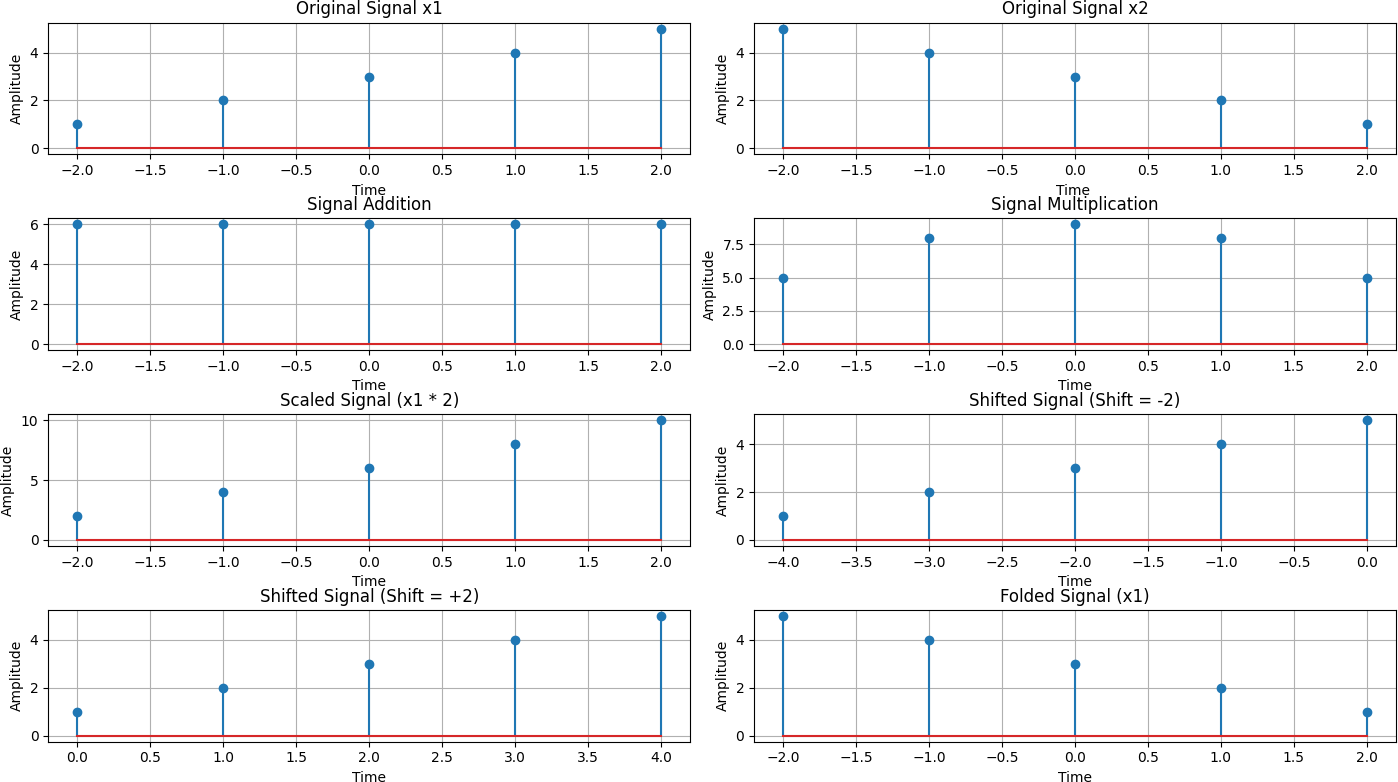
plt.subplot(4, 2, 6) plt.stem(shifted\_signal1, x1) plt.xlabel("Time") plt.ylabel("Amplitude") plt.title("Shifted Signal (Shift = -2)") plt.grid()

plt.subplot(4, 2, 7) plt.stem(shifted\_signal2, x1) plt.xlabel("Time") plt.ylabel("Amplitude") plt.title("Shifted Signal (Shift = +2)") plt.grid()

plt.subplot(4, 2, 8) plt.stem(n, folded\_signal) plt.xlabel("Time") plt.ylabel("Amplitude") plt.title("Folded Signal (x1)") plt.grid()

plt.tight\_layout() plt.show()

Output :



**Lab 3: Correlation of signal**

**Objectives:**

1. **Understand signal correlation**: Learn what correlation is and how it is used to measure the similarity between two signals.
2. **Analyze auto-correlation**: Study how auto-correlation of a signal is used to measure its similarity with itself at different time shifts.
3. **Learn cross-correlation**: Explore how cross-correlation measures the similarity between two different signals as one is shifted in time.
4. **Apply correlation in signal processing**: Understand the practical applications of correlation in signal detection, pattern recognition, and system analysis.
5. **Visualize correlation results**: Graphically represent the correlation between signals and interpret the results.

**Theory:**

**1. Correlation of Signals:**

* **Correlation** is a mathematical tool used to measure the similarity or relationship between two signals. It is used to determine how much one signal resembles another signal as one is shifted over time.
* The correlation of two signals, x(t) and y(t), is often denoted as Rxy(τ)*"*, where is the time shift (lag). It essentially measures how much x(t) and y(t) overlap as y(t) is shifted by τ

**2. Auto-correlation:**

* **Auto-correlation** is a special case of correlation where a signal is correlated with itself. It is used to find repeating patterns or periodicity in a signal, and is often used in signal processing, communications, and time-series analysis.
* Mathematically, auto-correlation is defined as:

* + **x(t)** is the signal, and τ\tau is the time shift.
* The result of auto-correlation can indicate the similarity of a signal to itself as it is shifted in time. Peaks in the auto-correlation function indicate periodicity or repeating patterns in the signal.

**3. Cross-correlation:**

* **Cross-correlation** measures the similarity between two different signals as one is shifted in time relative to the other. It is used to identify patterns, detect signals in noisy environments, or compare the alignment of two signals.
* Mathematically, cross-correlation is defined as:

* + **x(t)** and **y(t)** are two different signals, and τ\tau is the time shift.
* Cross-correlation is useful in signal detection, such as identifying when a known signal pattern appears within a noisy signal, or when comparing two signals with different time alignments.

**4. Practical Applications of Correlation:**

* **Signal Detection**: Correlation is used in detecting specific patterns within a signal. For example, cross-correlation can be used in radar systems to detect reflected signals.
* **Pattern Recognition**: Auto-correlation and cross-correlation are widely used in machine learning and image processing for pattern matching and object recognition.
* **Time-Series Analysis**: Auto-correlation is used to detect periodicity or trends in time-series data, such as stock market trends or sensor data.
* **Noise Reduction**: In noisy signal environments, correlation helps to extract the signal of interest by comparing the known reference signal to the noisy observation.

**5. Visualizing Correlation:**

* **Auto-correlation plot**: By plotting the auto-correlation of a signal, we can visualize how the signal overlaps with itself over various time shifts and detect periodicity or repeating patterns.
* **Cross-correlation plot**: By plotting the cross-correlation between two signals, we can observe the time lag where the two signals are most similar, helping in time alignment or pattern detection.

In summary, correlation is a powerful tool in signal processing, used to measure the relationship between signals, detect patterns, and analyze time-dependent behaviors.

import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import correlate, correlation\_lags

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, lags

def compute\_cross\_correlation(signal1, signal2):

cross\_corr = correlate(signal1, signal2, mode='full', method='auto') lags = correlation\_lags(len(signal1), len(signal2), mode='full') return cross\_corr, lags

fs = 1000 # Sampling frequency in Hz

t = np.linspace(0, 1, fs, endpoint=False) # Time vector freq = 5 # Frequency of the sine wave

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

auto\_corr, lags\_auto = compute\_autocorrelation(sin\_signal) signal1 = sin\_signal

signal2 = np.roll(signal1, 100)

cross\_corr, lags\_cross = compute\_cross\_correlation(signal1, signal2) noise = np.random.normal(0, 0.5, fs)

noisy\_signal = signal1 + noise

cross\_corr\_noise, lags\_noise = compute\_cross\_correlation(signal1, noisy\_signal) plt.figure(figsize=(12, 12))

plt.subplot(3, 1, 1) plt.plot(lags\_auto, auto\_corr)

plt.title("Autocorrelation of a Sinusoidal Signal") plt.xlabel("Lag")

plt.ylabel("Autocorrelation") plt.grid()

plt.subplot(3, 1, 2) plt.plot(lags\_cross, cross\_corr)

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

plt.ylabel("Cross-Correlation") plt.grid()

plt.subplot(3, 1, 3) plt.plot(lags\_noise, cross\_corr\_noise)

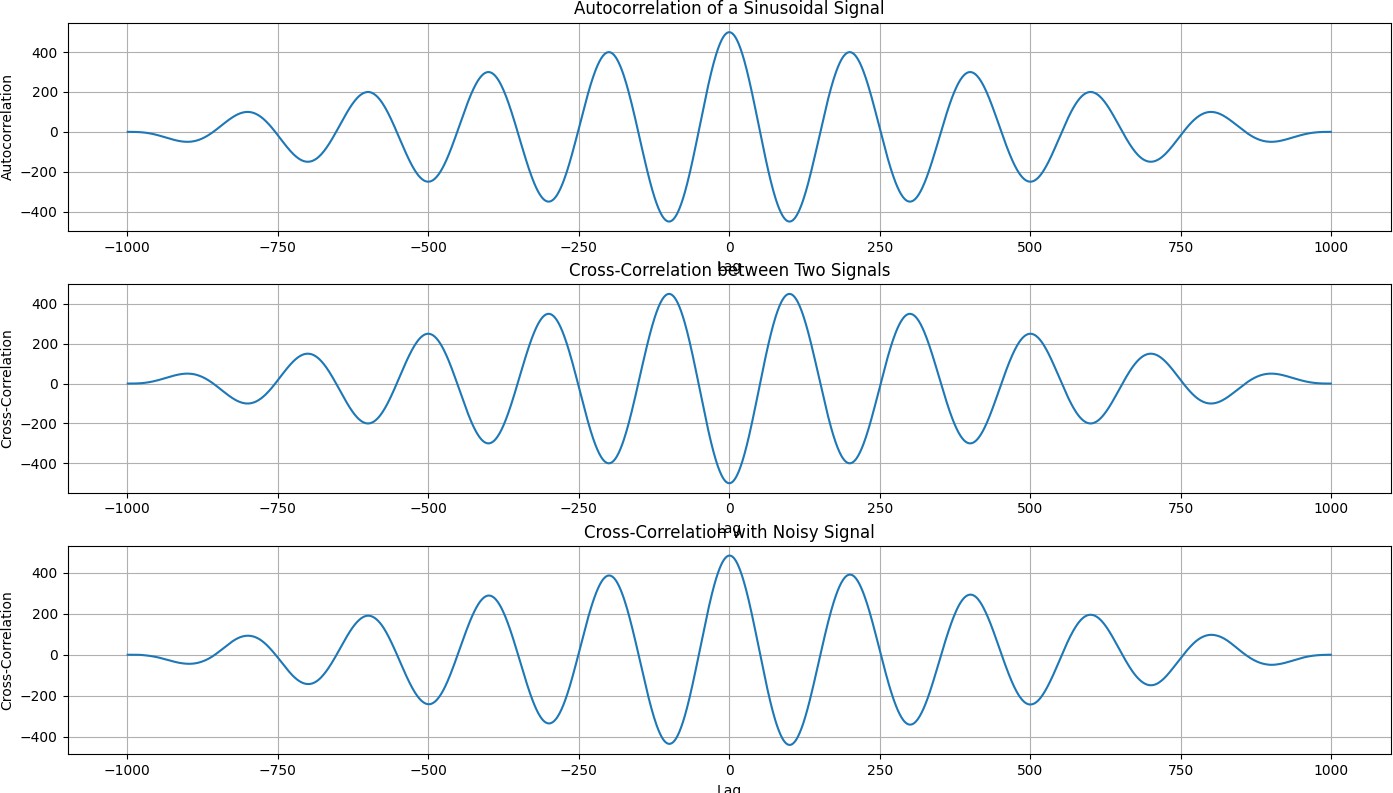
plt.title("Cross-Correlation with Noisy Signal") plt.xlabel("Lag")

plt.ylabel("Cross-Correlation") plt.grid()

plt.tight\_layout()

plt.show()

Output :



**Lab 4: Convolution of Signal**

**Objectives:**

1. **Learn convolution**: Understand the concept and significance of convolution in signal processing.
2. **Perform convolution**: Study how to compute the convolution of two signals (both continuous and discrete).
3. **Analyze properties**: Understand the key properties of convolution such as commutativity, associativity, and distributivity.
4. **Apply convolution in systems**: Understand how convolution is used in analyzing LTI (Linear Time-Invariant) systems.
5. **Visualize results**: Observe the effects of convolution through graphical representations.

**Theory:**

**1. Convolution of Signals:**

* Convolution is the process of combining two signals to produce a third. It is used to analyze the output of a system when an input signal is passed through it.
* For continuous signals, the convolution is:
* For discrete signals, it is:

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 # Sampling frequency in Hz

t = np.linspace(0, 1, fs, endpoint=False) # Time vector freq = 5 # Frequency of the sine wave

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

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

signal1 = sin\_signal

signal2 = np.roll(signal1, 100)

conv\_shifted = compute\_convolution(signal1, signal2) noise = np.random.normal(0, 0.5, fs)

noisy\_signal = signal1 + noise

conv\_noisy = compute\_convolution(signal1, noisy\_signal) plt.figure(figsize=(12, 12))

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()

plt.subplot(3, 1, 2) plt.plot(conv\_shifted)

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

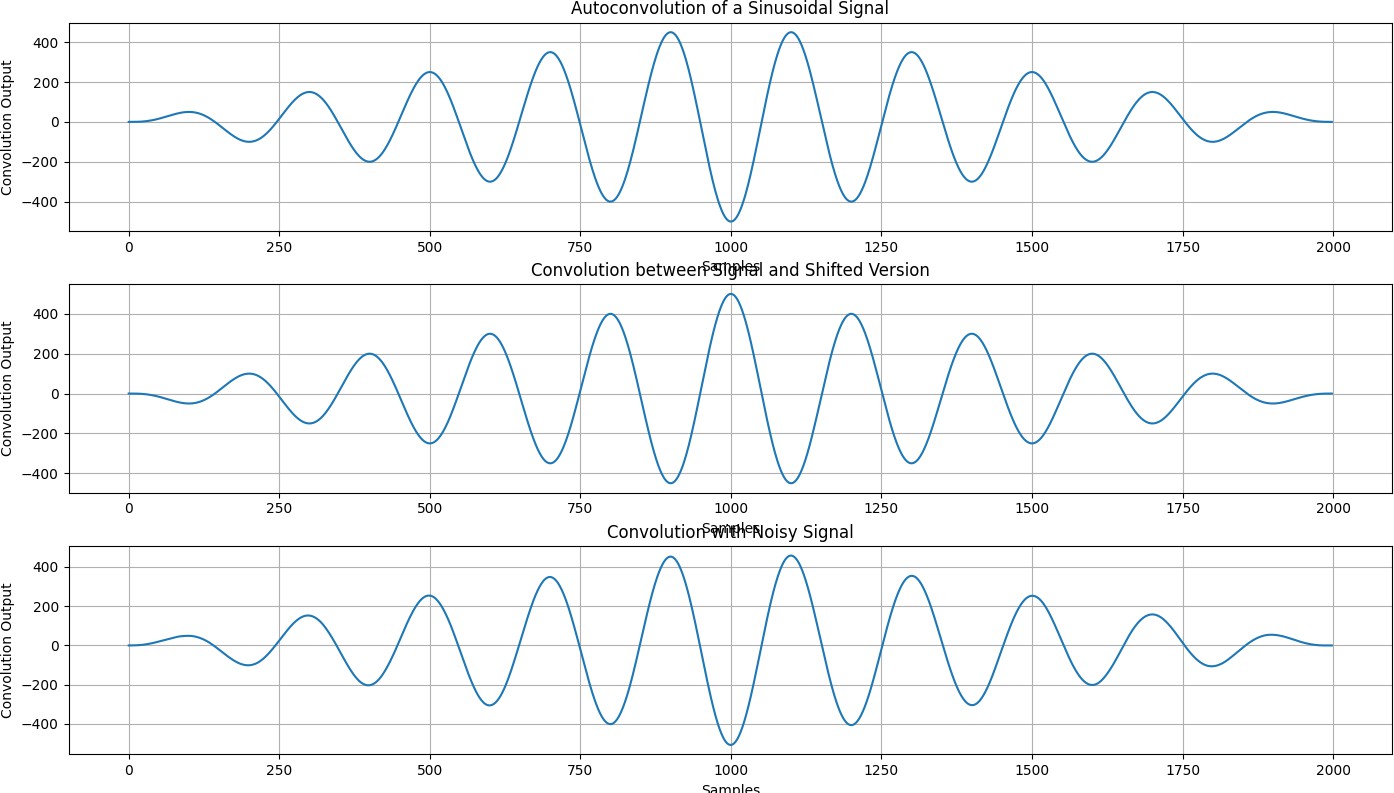
plt.ylabel("Convolution Output") plt.grid()

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()

Output :



**Lab 5: PPG signal (filtering, feature extraction, Peak detection)**

**Objectives:**

1. **Filter the PPG Signal**: Apply bandpass filtering to remove noise and retain the frequency components related to heart rate.
2. **Detect Peaks in the Signal**: Identify the peaks in the filtered PPG signal, which correspond to heartbeats, using peak detection algorithms.
3. **Extract Heart Rate**: Calculate the heart rate by measuring the time intervals between successive peaks (RR intervals) and converting it to beats per minute (BPM).
4. **Analyze Signal Features**: Learn how to extract meaningful features from the PPG signal, such as peak locations and heart rate, for further health monitoring applications.
5. **Understand Signal Processing Techniques**: Gain an understanding of common signal processing techniques used in PPG analysis, including filtering, peak detection, and feature extraction.

**Theory:**

**1. PPG Signal:**

* The **Photoplethysmogram (PPG)** signal is a non-invasive optical measurement of blood volume changes in the microvascular bed of tissue. It is commonly used for monitoring heart rate and blood oxygen levels.

**2. Bandpass Filtering:**

* **Bandpass filtering** removes noise from the PPG signal by allowing only the frequency range of interest (typically between 0.5 Hz to 5.0 Hz) to pass through. This removes low-frequency noise (like motion artifacts) and high-frequency noise (like powerline interference).

**3. Peak Detection:**

* **Peak detection** identifies the **R-peaks** of the PPG signal, which correspond to the heartbeats. This is done by finding the local maxima in the filtered signal. These peaks are crucial for calculating heart rate and other cardiovascular parameters.

**4. Feature Extraction (Heart Rate Calculation):**

* The time intervals between successive peaks, known as **RR intervals**, are calculated. The **heart rate** is determined by averaging the RR intervals and converting it to beats per minute (BPM):
* This process helps estimate the number of heartbeats per minute from the PPG signal.

**5. Applications:**

* **Heart Rate Monitoring**: PPG is widely used in wearable devices to measure heart rate.
* **Health Monitoring**: It can also be used to monitor cardiovascular health by analyzing the shape and amplitude of the PPG signal.

import numpy as np

import scipy.signal as signal import matplotlib.pyplot as plt

def bandpass\_filter(data, fs=100):

b, a = signal.butter(4, [0.5 / (0.5 \* fs), 5.0 / (0.5 \* fs)], btype='band') return signal.filtfilt(b, a, data)

def detect\_peaks(signal\_data):

return signal.find\_peaks(signal\_data, distance=50)[0]

def extract\_heart\_rate(peaks, fs=100): if len(peaks) < 2:

return 0

rr\_intervals = np.diff(peaks) / fs return 60 / np.mean(rr\_intervals)

# Generate synthetic PPG signal fs = 100

t = np.linspace(0, 10, fs \* 10) sine\_signal = np.sin(2 \* np.pi \* 1.2 \* t)

noise\_signal = 0.1 \* np.random.normal(0, 1, len(t)) ppg\_signal = sine\_signal + noise\_signal

# Process PPG signal

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

normalized\_signal = (filtered\_signal - np.min(filtered\_signal)) / (np.max(filtered\_signal) - np.min(filtered\_signal))

peaks = detect\_peaks(normalized\_signal) heart\_rate = extract\_heart\_rate(peaks, fs)

# Print results

print("Filtered Signal (first 10 values):", filtered\_signal[:10]) print("Detected Peaks (first 10 indices):", peaks[:10]) print(f"Estimated Heart Rate: {heart\_rate:.2f} BPM")

# Plot results plt.figure(figsize=(12, 9))

plt.subplot(3, 2, 1)

plt.plot(t, sine\_signal, label='Raw Sine Signal') plt.xlabel("Time")

plt.ylabel("Amplitude") plt.legend()

plt.subplot(3, 2, 2)

plt.plot(t, noise\_signal, label='Raw Noise Signal') plt.xlabel("Time")

plt.ylabel("Amplitude") plt.legend()

plt.subplot(3, 2, 3)

plt.plot(t, ppg\_signal, label='Raw PPG Signal') plt.xlabel("Time")

plt.ylabel("Amplitude") plt.legend()

plt.subplot(3, 2, 4)

plt.plot(t, filtered\_signal, label='Filtered PPG Signal') plt.xlabel("Time")

plt.ylabel("Amplitude") plt.legend()

plt.subplot(3, 2, 5)

plt.plot(t, normalized\_signal, label='Normalized PPG Signal') plt.xlabel("Time")

plt.ylabel("Amplitude") plt.legend()

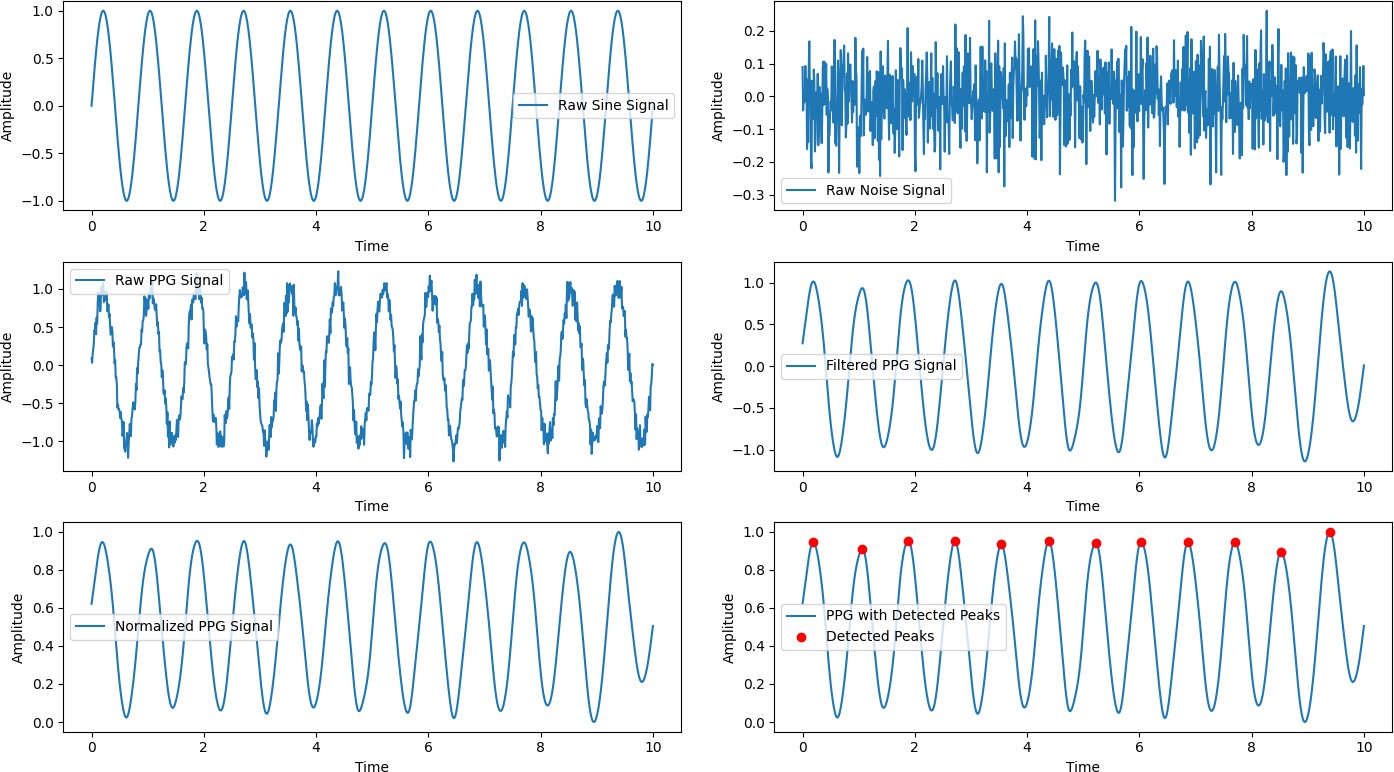
plt.subplot(3, 2, 6)

plt.plot(t, normalized\_signal,label=f'PPG with Detected Peaks') plt.plot(t[peaks], normalized\_signal[peaks],'ro', label='Detected Peaks') plt.xlabel("Time")

plt.ylabel("Amplitude") plt.legend()

plt.tight\_layout() plt.show()

Output :



**Lab 6: Implementation of Discrete Fourier Transform(DFT) and Inverse Discrete Fourier Transform(IDFT)**

**Objectives:**

1. **Understand the Discrete Fourier Transform (DFT)**: Learn the concept and importance of DFT in analyzing signals in the frequency domain.
2. **Implement DFT and IDFT**: Implement the DFT and Inverse DFT algorithms to transform signals between the time and frequency domains.
3. **Analyze Frequency Components**: Extract the frequency components of a signal and understand how DFT reveals the magnitude and phase information of these components.
4. **Implement Efficient Computation**: Learn how to compute DFT and IDFT using both direct methods and optimized techniques, such as FFT (Fast Fourier Transform).
5. **Visualize Results**: Plot the frequency spectrum of the signal after applying DFT and compare it with the reconstructed signal after applying IDFT.

**Theory:**

**1. Discrete Fourier Transform (DFT):**

* The **Discrete Fourier Transform (DFT)** is a mathematical transformation used to analyze the frequency content of discrete signals.
  + The DFT converts a time-domain signal x[n] into a frequency-domain representation X[k] by decomposing it into sinusoids with different frequencies. The formula for DFT is:

Where,

* x[n] is the input signal in the time domain.
* X[k] is the corresponding frequency component at the index kkk.
* N is the number of samples in the signal.
* j is the imaginary unit.

**2. Inverse Discrete Fourier Transform (IDFT):**

* The **Inverse DFT (IDFT)** is used to convert the frequency-domain signal X[k] back into the time-domain signal x[n]. The formula for IDFT is:

* + The IDFT reconstructs the original signal from its frequency components.

import numpy as np

import matplotlib.pyplot as plt

# Input sequence and N x = [1,1,1,1]

N= 4

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

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

# IDFT computation (Inverse DFT) x\_reconstructed = np.fft.ifft(X)

# Print the DFT and IDFT values print("DFT values:", X)

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

# Plot the input signal plt.figure(figsize=(10, 6))

plt.subplot(3, 1, 1) plt.stem(range(len(x)), x) plt.title('Input Signal x(n)') plt.xlabel('n')

plt.ylabel('x(n)') plt.grid()

# Plot the magnitude of DFT plt.subplot(3, 1, 2) plt.stem(range(N), np.abs(X)) plt.title('DFT Magnitude |X(k)|') plt.xlabel('k')

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

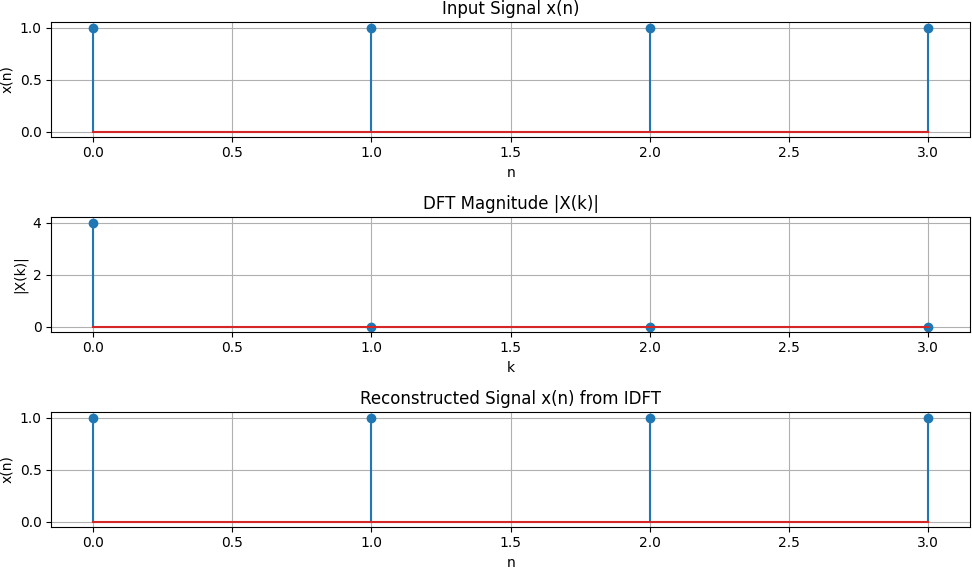
# Plot the IDFT signal plt.subplot(3, 1, 3)

plt.stem(range(N), x\_reconstructed.real) plt.title('Reconstructed Signal x(n) from IDFT') plt.xlabel('n')

plt.ylabel('x(n)') plt.grid()

plt.tight\_layout() plt.show()

Output :



**Lab-7: Explain and Implement Frequency bin using python.**

**Objectives**:

1. **Generate a sine wave**: Create a signal with a frequency of 50 Hz sampled at 1000 Hz.
2. **Compute the FFT**: Apply the Fast Fourier Transform to convert the time-domain signal to the frequency domain.
3. **Normalize the FFT**: Normalize the FFT output to make it independent of the signal length.
4. **Plot the frequency spectrum**: Visualize the one-sided magnitude spectrum, showing the peak at 50 Hz where the sine wave frequency is located.

**Theory:**

A frequency bin refers to the discrete intervals of frequency in the frequency domain when performing a Fourier Transform (such as FFT). In a typical FFT, the resulting frequency spectrum is composed of different frequency bins that represent specific ranges of frequency.

For a signal sampled at a certain frequency (sampling rate), the FFT transforms the time-domain signal into a frequency-domain signal. The resulting frequencies are divided into bins, and each bin contains the magnitude of the frequency components in that range.

The **frequency resolution** is calculated as:

Resolution=

Where:

* fs​ is the sampling frequency.
* N is the number of samples.

**Code:**  
import numpy as np

import matplotlib.pyplot as plt

# Parameters

fs = 1000 # Sampling frequency (Hz)

N = 1024 # Number of points (FFT size)

t = np.arange(N) / fs # Time vector

# Generate a sine wave with frequency of 50 Hz

f\_signal = 50 # Frequency of the sine wave (Hz)

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

# Compute the FFT of the signal

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

fft\_magnitude = np.abs(fft\_output) / N # Normalize

freq\_bins = np.fft.fftfreq(N, 1/fs) # Compute frequency bins

# Plot the FFT magnitude vs frequency bins

plt.figure(figsize=(10, 5))

plt.plot(freq\_bins[:N//2], fft\_magnitude[:N//2]) # One-sided spectrum

plt.xlabel("Frequency (Hz)")

plt.ylabel("Magnitude")

plt.title("Frequency Bins of a Sine Wave")

plt.grid()

plt.show()

Output:

