PABNA UNIVERSITY OF SCIENCE AND TECHNOLOGY

Faculty of Engineering & Technology

Department of Information and Communication Engineering

Course name : Signals and Systems Sessional

Course Code : ICE-2204

Lab Report On Signals and Systems

Submitted By:

Name: Arifur Rahman
Roll: 220612
Session:2021-22
2nd Year 2nd Semester
Department of ICE,PUST

Submitted To:

Dr.Md. Imran Hossain
Associate Professor
Department of Information and Communication Engineering,
Pabna University of Science and Technology, Pabna

INDEX

Experiment no : 1

Experiment name : Analysis of Discrete-Time Signals sequences: Impulse, Step, and Ramp Signals.

Objectives :

1. To understand and analyze fundamental discrete-time signals: impulse, step, and ramp signals.
2. To visualize these signals using Python and Matplotlib.
3. To comprehend the significance of these signals in signal processing and system analysis.

Theory :

In discrete-time signal processing, fundamental signals serve as the building blocks for analyzing and designing complex signals and systems. Three of the most basic discrete-time signals are the impulse signal, step signal, and ramp signal.

1. Impulse Signal (δ[n])

The discrete-time impulse signal, also known as the unit impulse function or Kronecker delta function, is defined as:

$$\delta[n] = \begin{cases} 1, & n = 0 \\ 0, & n \neq 0 \end{cases}$$

This signal is used as an identity element in convolution and plays a crucial role in system analysis, as it helps determine a system’s impulse response.

2. Step Signal (u[n])

The discrete-time unit step signal is defined as:

$$u[n] = \begin{cases} 1, & n \geq 0 \\ 0, & n < 0 \end{cases}$$

It represents a sudden transition from zero to one at n = 0. The step function is widely used in system analysis to represent signals that begin at a certain time and continue indefinitely.

3. Ramp Signal (r[n])

The discrete-time ramp signal is defined as:

$$r[n] = \begin{cases} n, & n \geq 0 \\ 0, & n < 0 \end{cases}$$

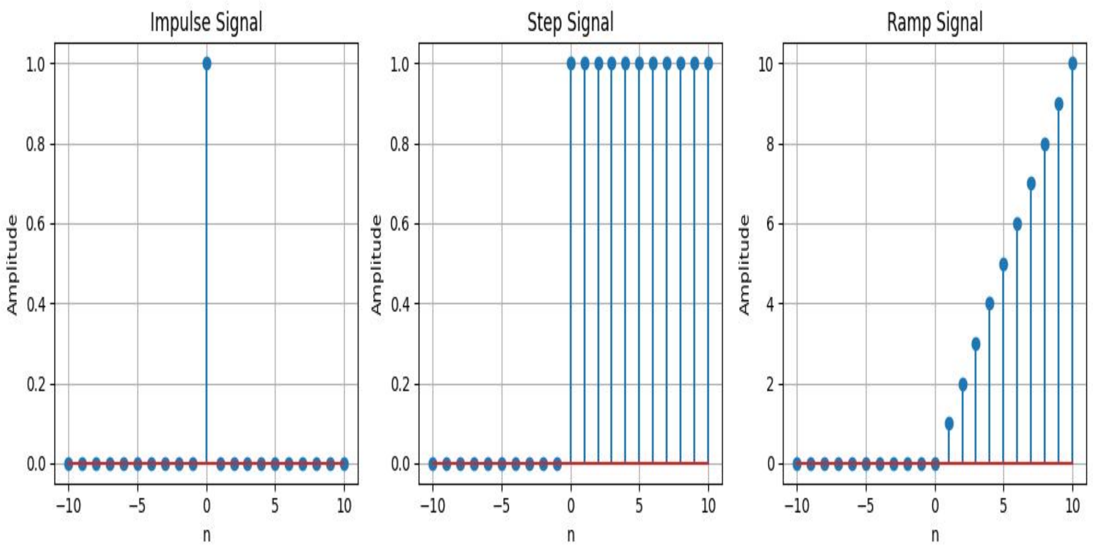
This signal is a linearly increasing function and is often used to model systems with constant acceleration or increasing growth trends.

Source Code :

```
import numpy as np
import matplotlib.pyplot as plt

#defining 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)
```

```
# generating signals
impulse = impulse_signal(n)
step = step_signal(n)
ramp = ramp_signal(n)

# plotting
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 :



Experiment no : 2

Experiment name : Operations on Discrete-Time Signals: Addition, Multiplication, Scaling, Shifting, and Folding.

Objectives :

1. To understand and analyze basic operations on discrete-time signals.
2. To perform signal addition, multiplication, and scaling.
3. To implement time-domain transformations such as shifting and folding.
4. To visualize these operations using Python and Matplotlib.

Theory :

In digital signal processing, various operations on discrete-time signals help in system analysis and modification. Some fundamental operations include:

1. Signal Addition

Signal addition involves summing the corresponding elements of two signals:

$$y[n] = x1[n] + x2[n]$$

This operation is useful in combining signals, such as in mixing audio signals.

2. Signal Multiplication

Signal multiplication involves multiplying the corresponding elements of two signals:

$$y[n] = x1[n] * x2[n]$$

This operation is often used in modulation and feature extraction applications.

3. Signal Scaling

Scaling a signal involves multiplying each value by a constant factor:

$$y[n] = \alpha * x[n]$$

This operation amplifies or attenuates the signal based on the scaling factor α.

4. Signal Shifting (Time Shift)

Shifting a signal involves delaying or advancing its time indices:

- Left Shift (Advance): y[n] = x[n+k](shifts the signal earlier).
- Right Shift (Delay): y[n]=x[n−k] (shifts the signal later).

Shifting is commonly used in time-domain filtering and system response analysis.

5. Signal Folding (Time Reversal)

Folding a signal flips it around the origin:

$$y[n] = x[-n]$$

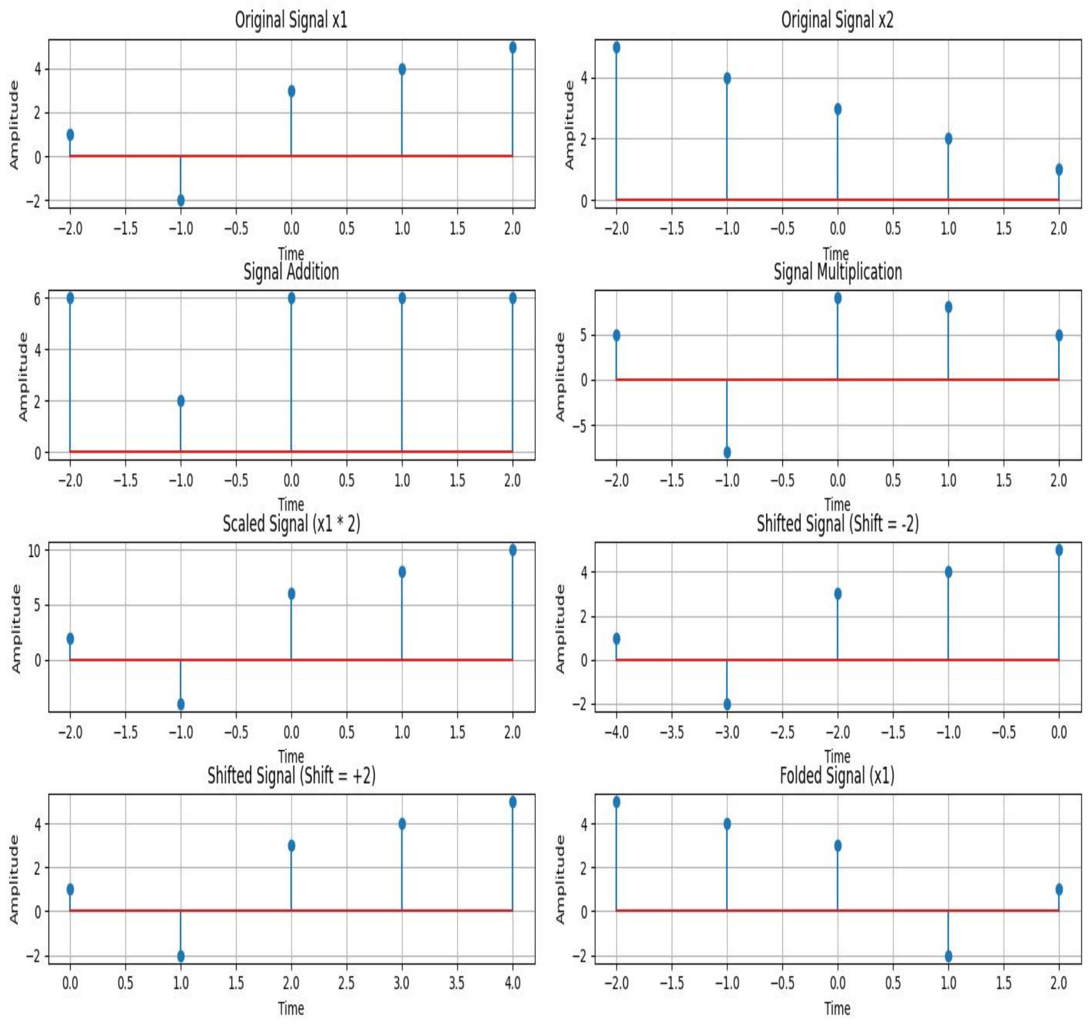
This operation is used in symmetry analysis and convolution operations.

Source Code :

```
import numpy as np
import matplotlib.pyplot as plt

# function for adding
def signal_addition(x1, x2):
    return x1 + x2

# function for multiplying
def signal_multiplication(x1, x2):
    return x1 * x2
```

```
# function for scaling
def signal_scaling(x, alpha):
    return alpha * x

# function for shifting
def signal_shifting(n, shift):
    return n + shift

# Function to fold (reverse) a signal
def signal_folding(x):
    return np.flip(x)

# signals
n = np.array([-2, -1, 0, 1, 2])
x1 = np.array([1, -2, 3, 4, 5])
x2 = np.array([5, 4, 3, 2, 1])

#operations
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 indices only
shifted_signal2 = signal_shifting(n, +2)  # Shifted indices only
folded_signal = signal_folding(x1)  # Folded indices

# plotting
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 :



Experiment no : 3

Experiment name : Convolution of Discrete-Time Signals: Sinusoidal, Shifted, and Noisy Signals.

Objectives :

1. To understand the concept of convolution in discrete-time signal processing.
2. To perform and analyze the autoconvolution of a sinusoidal signal.
3. To study the effect of convolution with a time-shifted version of a signal.
4. To observe the impact of convolution when noise is introduced to a signal.
5. To visualize convolution results using Python and Matplotlib.

Theory :

Convolution combines two functions (e.g., a signal and an impulse response) to produce a new function that represents their interaction.It is a fundamental operation in digital signal processing used for system analysis, filtering, and feature extraction. It is defined as:

$$y[n] = (x1 * x2)[n] = \sum_{k=-\infty}^{\infty} x_1[k]x_2[n-k]$$

where:

- x1[n] and x2[n] are discrete signals.
- y[n]is the convolution result.

1. Autoconvolution of a Sinusoidal Signal

Autoconvolution involves convolving a signal with itself, which helps analyze signal energy distribution and periodicity.

2. **Convolution with a Shifted Version**

When a signal is convolved with its shifted version, the phase relationship between the original and shifted signals affects the output. This process is useful in time-delay estimation and system identification.

3. Convolution with a Noisy Signal

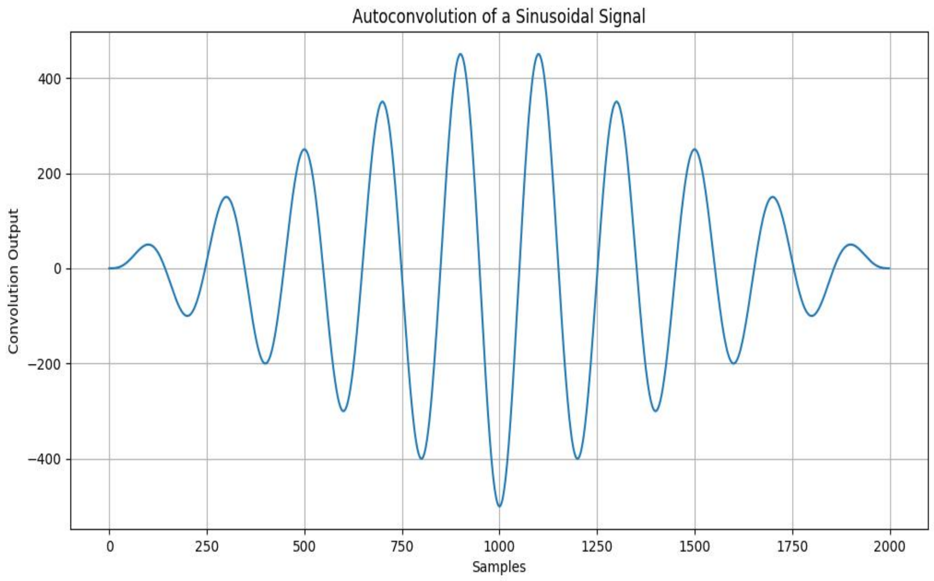
Introducing noise into a signal and performing convolution allows us to study the effects of noise filtering and signal enhancement techniques.

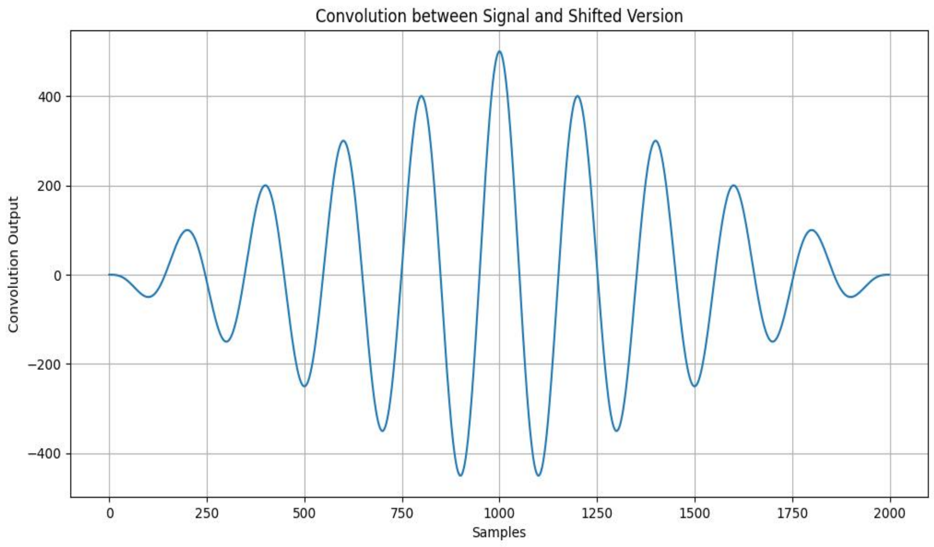
Source Code :

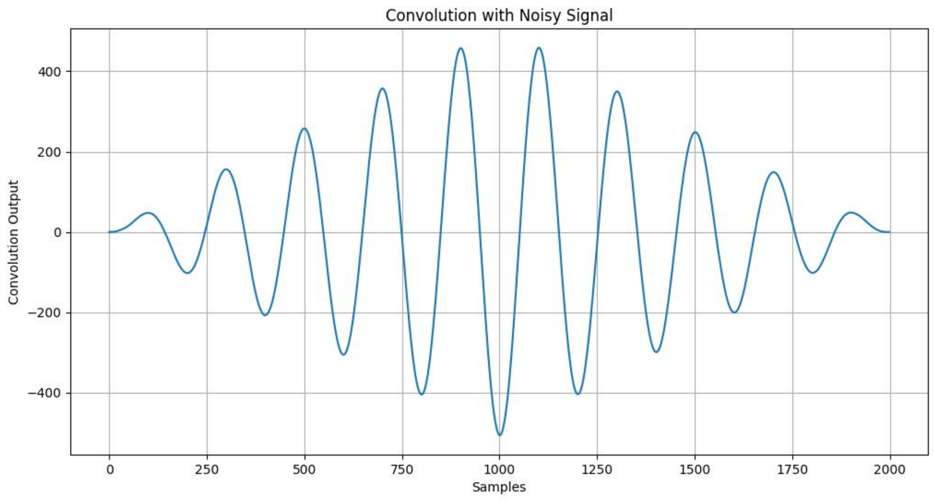
```
import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import convolve


# function for convolution

def compute_convolution(signal1, signal2):

    conv_result = convolve(signal1, signal2, mode='full', method='auto')

    return conv_result


# frequency and time vector

fs = 1000

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

freq = 5  # frequency of the sine wave


#sinusoidal signal

sin_signal = np.sin(2 * np.pi * freq * t)
```

```
# 1: Autoconvolution of a sinusoidal signal
conv_auto = compute_convolution(sin_signal, sin_signal)

plt.figure(figsize=(12, 6))
plt.plot(conv_auto)
plt.title("Autoconvolution of a Sinusoidal Signal")
plt.xlabel("Samples")
plt.ylabel("Convolution Output")
plt.grid()
plt.show()

# 2: Convolution with a shifted version
signal1 = sin_signal  # sinusoidal signal
signal2 = np.roll(signal1, 100)  # shifted version

conv_shifted = compute_convolution(signal1, signal2)

plt.figure(figsize=(12, 6))
plt.plot(conv_shifted)
plt.title("Convolution between Signal and Shifted Version")
plt.xlabel("Samples")
plt.ylabel("Convolution Output")
plt.grid()
plt.show()


# 3: Convolution with a noisy signal
noise = np.random.normal(0, 0.5, fs)  #Gaussian noise
noisy_signal = signal1 + noise


conv_noisy = compute_convolution(signal1, noisy_signal)


plt.figure(figsize=(12, 6))
plt.plot(conv_noisy)
plt.title("Convolution with Noisy Signal")
plt.xlabel("Samples")
plt.ylabel("Convolution Output")
plt.grid()
plt.show()
```

Output :


Autoconvolution of a Sinusoidal Signal
Convolution Output
Samples
400
200
0
−200
−400
0
250
500
750
1000
1250
1500
1750
2000


Convolution between Signal and Shifted Version
Convolution Output
Samples
400
200
0
−200
−400
0
250
500
750
1000
1250
1500
1750
2000


Convolution with Noisy Signal
Convolution Output
Samples
400
200
0
−200
−400
0
250
500
750
1000
1250
1500
1750
2000

Experiment no : 4

Experiment name : Auto-correlation and Cross-Correlation of Discrete-Time Signals.

Objectives :

1. To understand the concepts of auto-correlation and cross-correlation in signal processing.
2. To compute and analyze the auto-correlation of a sinusoidal signal.
3. To examine the cross-correlation between a signal and its time-shifted version.
4. To evaluate the impact of noise on cross-correlation.
5. To visualize these correlation functions using Python and Matplotlib.

Theory :

Correlation measures the similarity between two signals or functions. It quantifies how much one signal resembles another when shifted by a certain amount.

1. Auto-correlation

Auto-correlation measures the similarity of a signal with a delayed version of itself. It helps in detecting periodicity and analyzing signal energy. The auto-correlation function is given by:

$$R_x[k] = \sum_{n=-\infty}^{\infty} x[n]x[n-k]$$

where:

- $R_x[k]$ is the auto-correlation function.
- x[n] is the discrete-time signal.
- k is the lag (shift) value.

Auto-correlation is widely used in applications such as radar, speech analysis, and system identification.

2. Cross-Correlation

Cross-correlation measures the similarity between two different signals as a function of lag. It is useful in detecting time shifts and comparing signals. The cross-correlation function is given by:

$$Rxy[k] = \sum_{n=-\infty}^{\infty} x[n]y[n-k]$$

where:

- $R_{xy}[k]$ is the cross-correlation function between x[n] and y[n].
- k is the lag (shift) value.

Cross-correlation is commonly used in template matching, signal synchronization, and noise filtering.

3. Cross-Correlation with a Noisy Signal

When noise is added to a signal, cross-correlation helps in extracting the desired signal from the noisy environment. This is useful in communication systems and data analysis.

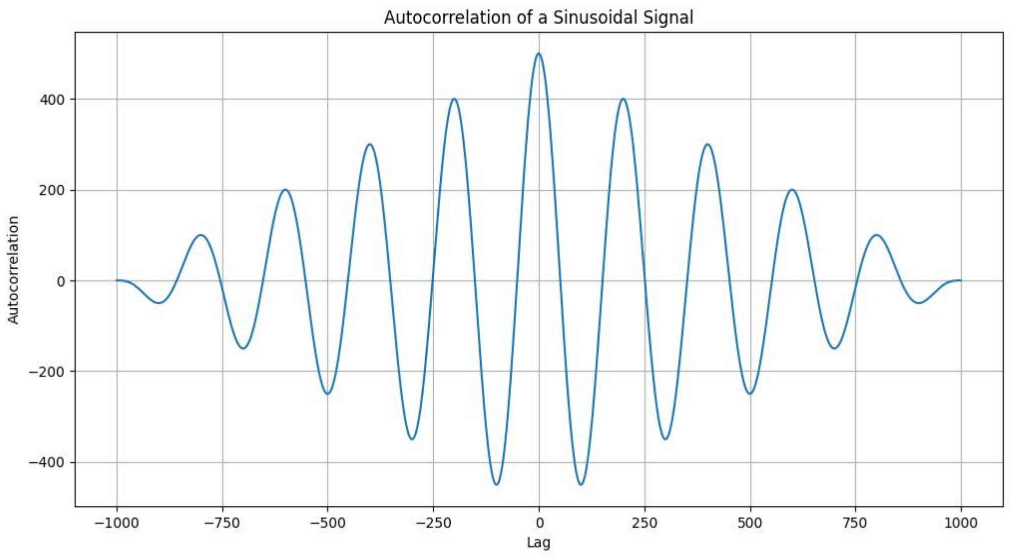
Source code :

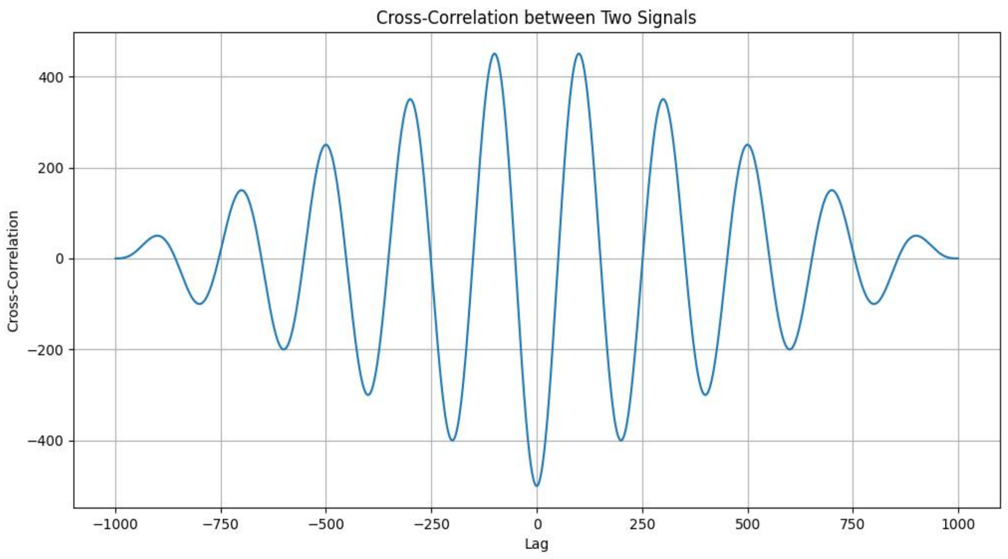
```
import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import correlate, correlation_lags



#function for auto-correlation

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



#function for cross-correlation

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



#frequency and time vector
```

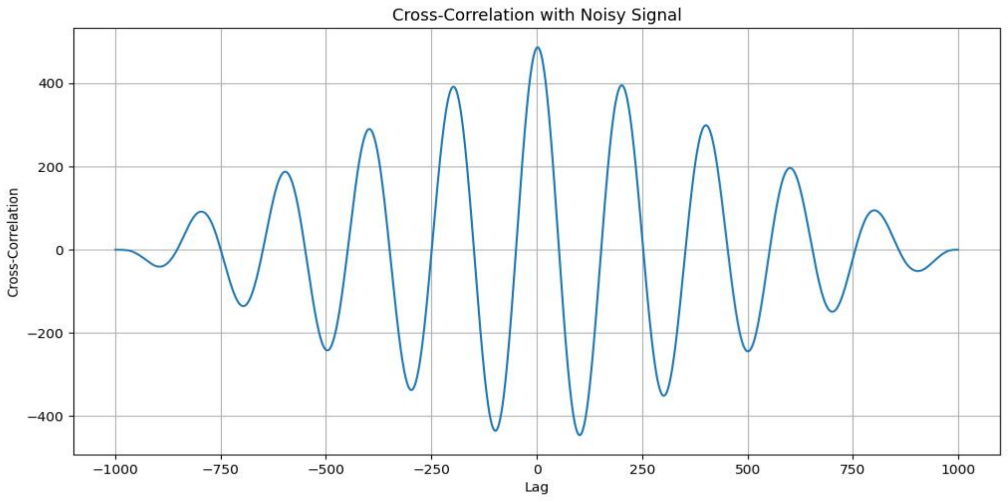
```
fs = 1000

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

freq = 5  #frequency of the sine wave



#sinusoidal signal

sin_signal = np.sin(2 * np.pi * freq * t)



# Compute and plot autocorrelation

auto_corr, lags = compute_autocorrelation(sin_signal)

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

plt.plot(lags, auto_corr)

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

plt.xlabel("Lag")

plt.ylabel("Autocorrelation")

plt.grid()

plt.show()



# cross-correlation between a signal and its shifted version
```

```
signal1 = sin_signal  # sinusoidal signal

signal2 = np.roll(signal1, 100)  # shifted version



cross_corr, lags = compute_cross_correlation(signal1, signal2)

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

plt.plot(lags, cross_corr)

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

plt.xlabel("Lag")

plt.ylabel("Cross-Correlation")

plt.grid()

plt.show()



# cross-correlation with noise

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

noisy_signal = signal1 + noise



cross_corr_noise, lags = compute_cross_correlation(signal1, noisy_signal)

plt.figure(figsize=(12, 6))
plt.plot(lags, cross_corr_noise)

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

plt.xlabel("Lag")

plt.ylabel("Cross-Correlation")

plt.grid()

plt.show()
```

Output :







Experiment no : 5

Experiment name : Detecting peak and extracting feature from a PPG signal.

Objectives :

1. To generate a synthetic Photoplethysmogram (PPG) signal with noise.
2. To apply a bandpass filter for noise reduction in the PPG signal.
3. To detect peaks in the filtered signal corresponding to heartbeats.
4. To estimate heart rate (HR) in beats per minute (BPM) from detected peaks.
5. To visualize raw and processed PPG signals for analysis.

Theory :

1. PPG Signal

Photoplethysmography (PPG) is a non-invasive optical technique used to measure blood volume changes in tissues. The periodic nature of the PPG waveform corresponds to the heartbeat.

2. Bandpass Filtering

A Butterworth bandpass filter is applied to remove unwanted frequency components. The cutoff frequencies are:

- 0.5 Hz (to remove low-frequency drift)
- 5 Hz (to eliminate high-frequency noise)

The bandpass filter equation is:

$$H(f) = \frac{1}{\sqrt{1 + (\frac{f}{f_c})^{2N}}}$$

where f_c is the cutoff frequency and N is the filter order.

3. Peak Detection

Peaks in the filtered PPG signal correspond to heartbeats. The scipy.signal.find_peaks function detects local maxima, ensuring a minimum distance of 50 samples between peaks to avoid false detections.

4. Heart Rate Estimation

Heart rate (HR) is computed using:

$$HR = \frac{60}{\text{Mean RR Interval (s)}}$$

where RR intervals are the time differences between consecutive detected peaks. The HR value is expressed in beats per minute (BPM).

5. Signal Normalization

To improve peak detection, the filtered signal is normalized using:

$$x_{\text{normalized}} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

This ensures the signal values are scaled between 0 and 1.

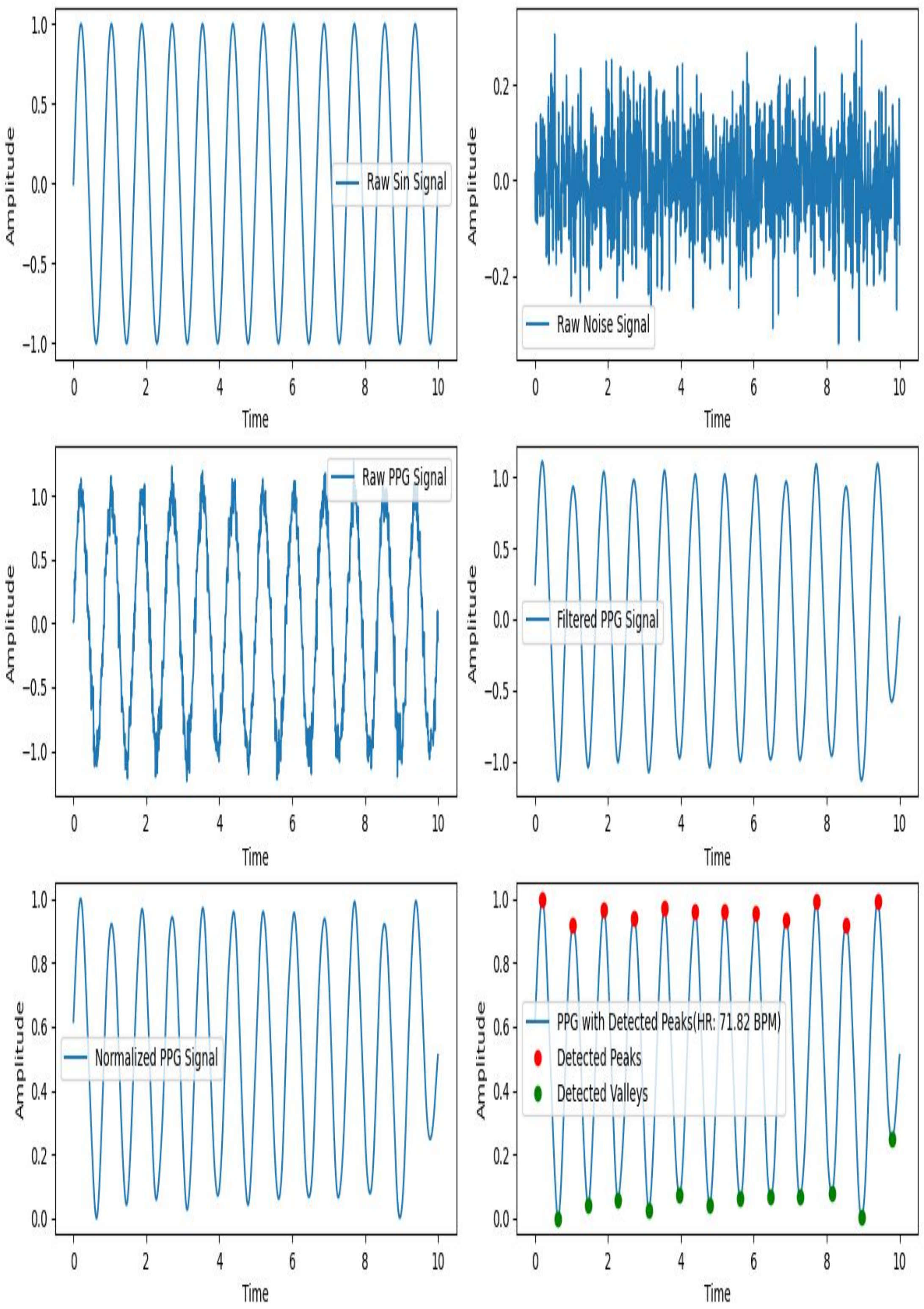
<u>Source code :</u>

```
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 detect_valleys(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)

valleys = detect_valleys(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("Detected Valleys (first 10 indices):", valleys[: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 Sin 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(HR: {heart_rate:.2f} BPM)')
plt.plot(t[peaks], normalized_signal[peaks],'ro', label='Detected Peaks')

plt.plot(t[valleys],normalized_signal[valleys],'go',label='Detected

Valleys')

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.legend()

plt.tight_layout()

plt.show()
```

Output :

```
Filtered Signal (first 10 values): [0.22942572 0.27059672 0.31263922
0.35637466 0.40244699 0.45125591 0.50291041 0.55720714
0.61363509 0.67140468]

Detected Peaks (first 10 indices): [ 20 104 186 271 353 436 518 604 685
771]

Detected Valleys (first 10 indices): [ 61 148 229 312 397 479 563 645
728 813]

Estimated Heart Rate: 71.90 BPM
```


Raw Sin Signal
Raw Noise Signal
Raw PPG Signal
Filtered PPG Signal
Normalized PPG Signal
PPG with Detected Peaks(HR: 71.82 BPM)
Detected Peaks
Detected Valleys
Amplitude
Time

Experiment no : 6

Experiment name : Fourier Series Approximation of a Square Wave.

Objectives :

1. To approximate a square wave using Fourier series expansion.
2. To analyze how the number of terms in the series affects the approximation.
3. To visualize the Gibbs phenomenon in Fourier series representation.

Theory :

1. Fourier Series Representation

A periodic function f(x) can be expressed as a sum of sine and cosine functions using the Fourier series:

$$f(x) = a_0 + \sum_{n=1}^{\infty} (a_n \cos(nx) + b_n \sin(nx))$$

For a square wave (odd function), only the sine terms contribute:

$$f(x) = \sum_{n=1,3,5,\ldots}^{\infty} \frac{4}{n\pi} \sin(nx)$$

where n includes only odd harmonics.

2. Gibbs Phenomenon

The Fourier series approximation of a discontinuous function (like a square wave) exhibits overshooting near the discontinuities. This overshooting does not vanish even with more terms, a behavior known as the Gibbs phenomenon.

3. Approximation with Finite Terms

The accuracy of the approximation improves as more terms are added:

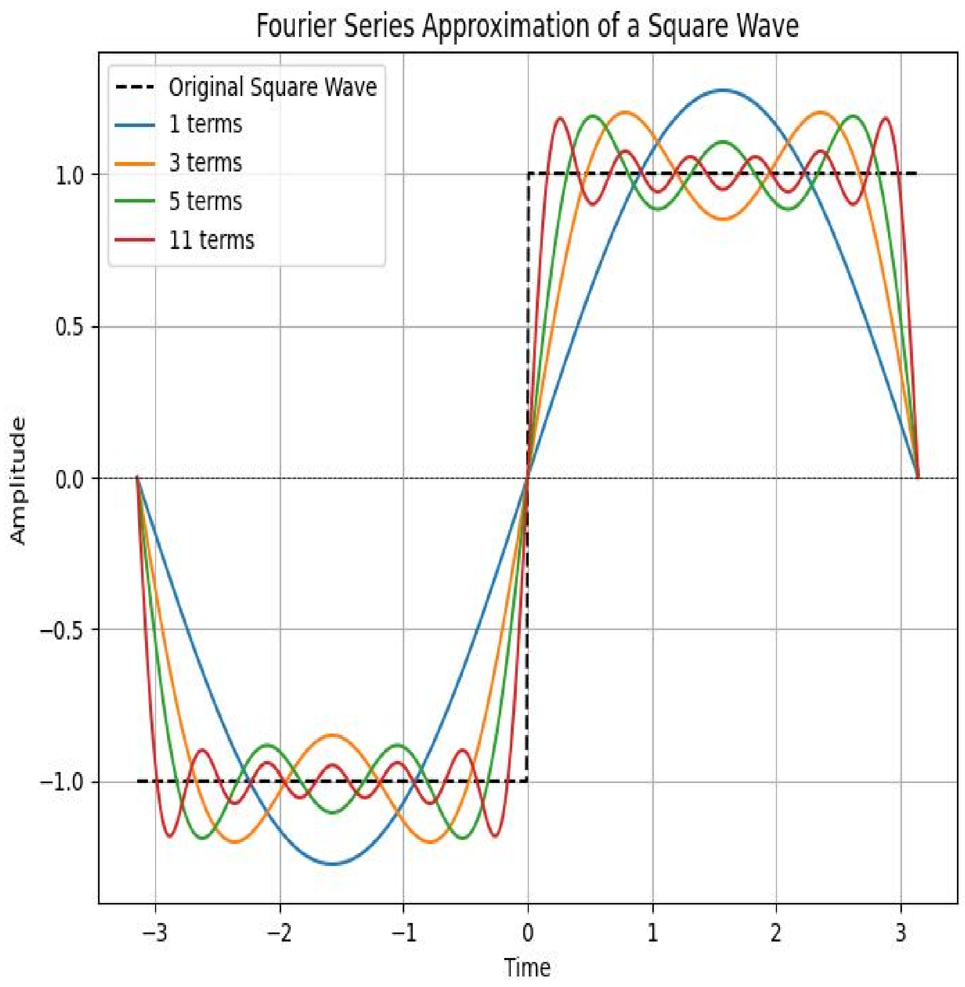
- 1 term: A single sine wave that roughly resembles the square wave.
- 3 terms: A better approximation, but with visible oscillations.
- 5 terms: Closer to the square wave, but still with overshooting.
- 11 terms: Even closer, but Gibbs phenomenon persists.

<u>Source Code :</u>

```
import numpy as np

import matplotlib.pyplot as plt



# Fourier series approximation of a square wave

def fourier_series(x, terms):

    if terms < 1:

        raise ValueError("Number of terms must be at least 1")



    result = x - x

    for n in range(1, terms + 1, 2):  # Use only odd harmonics
```

```
        result += (4 / (np.pi * n)) * np.sin(n * x)  # Sum sine terms
    return result


# Define the original square wave function
def square_wave(x):
    return np.where(np.sin(x) >= 0, 1, -1)  # Standard square wave


# Generate x values
t = np.linspace(-np.pi, np.pi, 400)


# Plot the original square wave
plt.figure(figsize=(8, 6))
plt.plot(t, square_wave(t), label='Original Square Wave', linestyle='--',
color='black')
for terms in [1, 3, 5, 11]:
    plt.plot(t, fourier_series(t, terms), label=f'{terms} terms')
plt.axhline(0, color='black', linewidth=0.5, linestyle='--')
plt.title('Fourier Series Approximation of a Square Wave')
plt.xlabel('Time')
plt.ylabel('Amplitude')

plt.legend()

plt.grid()

plt.show()
```

<u>Output :</u>



Experiment no : 7

Experiment name : Analyzing Fourier transform of a time function, for an aperiodic pulse.

Objectives :

1. To analyze and visualize the real, phase, and magnitude components of a sinc function.
2. To understand the behavior of the sinc function in both time and frequency domains.
3. To explore the significance of the sinc function in signal processing, particularly in filtering and reconstruction.

Theory :

The Fourier Transform is a mathematical operation that transforms a time-domain signal into its frequency-domain representation. It helps analyze how different frequency components contribute to the original signal.

For a continuous-time signal x(t), the Fourier Transform is defined as:

$$X(f) = \int_{-\infty}^{\infty} x(t) e^{-j2\pi ft} dt$$

where:

- X(f) is the Fourier Transform of x(t),

- f is the frequency in Hertz (Hz),
- j = $\sqrt{-1}$ represents the imaginary unit.

Let, an aperiodic pulse *x(t) = 1, −2 < t <2 ,*

Then,the Fourier transform X(f) :

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-j2\pi ft}dt$$

$$X(f) = \int_{-2}^{2} 1.e^{-j2\pi ft}dt$$

$$X(f) = \left[\frac{e^{-j2\pi ft}}{-j2\pi f}\right]_{-2}^{2} \quad .$$

$$X(f) = \frac{e^{-j2\pi f(2)} - e^{-j2\pi f(-2)}}{-j2\pi f}$$

$$X(f) = \frac{-2j\sin(4\pi f)}{-j2\pi f}$$

$$X(f) = \frac{2\sin(4\pi f)}{2\pi f} = \frac{\sin(4\pi f)}{\pi f}$$

By definition :

$$sinc(x) = \frac{\sin(\pi x)}{\pi x}$$

So, if x = 4f :

$$X(f) = 4 \cdot sinc(4\pi f)$$

Now if we plot considering $X(f) = 4 \cdot sinc(4\pi f)$, where t = 4πf

1. Definition of the Sinc Function

The sinc function is defined as:

$$sinc(x) = \frac{\sin(\pi x)}{\pi x}$$

For the given function:

$$x(t) = 4 \cdot sinc(4t)$$

where the factor 4 scales the amplitude, and the argument 4t compresses the function along the time axis.

2. Importance of the Sinc Function

- The sinc function appears in Fourier transforms and is the impulse response of an ideal low-pass filter.
- It is crucial in the Nyquist-Shannon sampling theorem, ensuring perfect signal reconstruction from discrete samples.

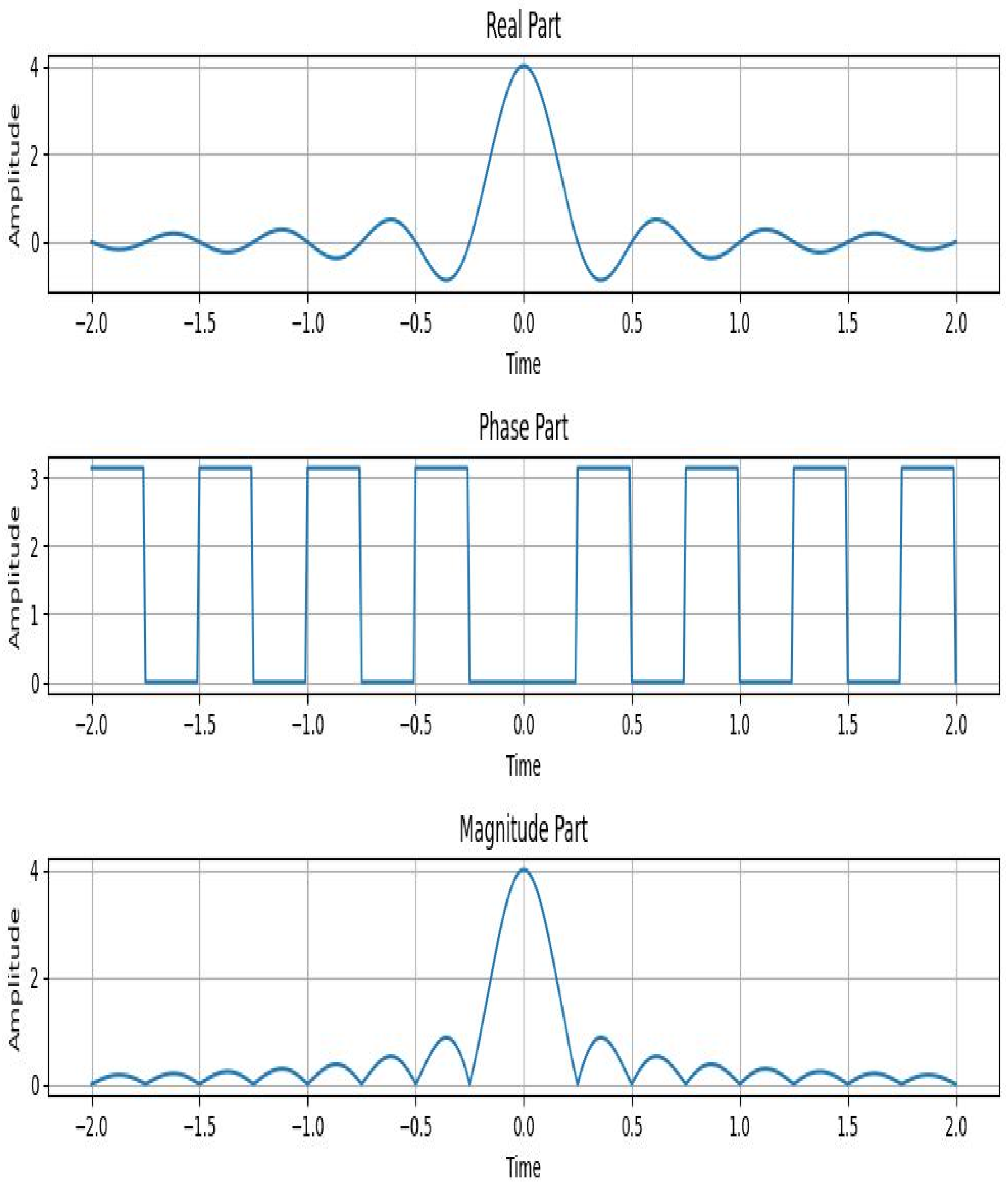
3. Real, Phase, and Magnitude Components

- Real Part: Represents the actual sinc waveform, oscillating and decaying over time.
- Phase Part: Gives the angle (phase) of the function, which remains either 0 or π.
- Magnitude Part: Shows the absolute value of the function, highlighting the envelope of sinc oscillations.

Source Code :

```
import numpy as np

import matplotlib.pyplot as plt



t = np.arange(-2, 2.01, 0.01)

x = 4 * np.sinc(4 * t)



# Plot real part

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

plt.subplot(3, 1, 1)

plt.plot(t, x)

plt.xlabel('Time')

plt.ylabel('Amplitude')

plt.title('Real Part')

plt.grid()



# Plot phase part

plt.subplot(3, 1, 2)
plt.plot(t, np.angle(x))  # Angle (phase) of x

plt.xlabel('Time')

plt.ylabel('Amplitude')

plt.title('Phase Part')

plt.grid()



# Plot magnitude part

plt.subplot(3, 1, 3)

plt.plot(t, np.abs(x))

plt.xlabel('Time')

plt.ylabel('Amplitude')

plt.title('Magnitude Part')

plt.grid()

plt.tight_layout()

plt.show()
```

<u>Output :</u>


Real Part
Amplitude
Time
Phase Part
Amplitude
Time
Magnitude Part
Amplitude
Time

Experiment no : 8

Experiment name : Implementation of Discrete Fourier Transform (DFT) and Inverse Discrete Fourier Transform (IDFT).

Objectives :

1. To compute the Discrete Fourier Transform (DFT) of a given input sequence.
2. To reconstruct the original signal using the Inverse Discrete Fourier Transform (IDFT).
3. To visualize the input sequence, its frequency components, and the reconstructed signal.

Theory :

Discrete Fourier Transform (DFT)

The Discrete Fourier Transform (DFT) is a mathematical tool used to analyze discrete-time signals in the frequency domain. It converts a sequence of N discrete values (time-domain) into another sequence of N complex values (frequency-domain).

For a discrete signal x[n] of length N, the DFT is defined as:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi}{N}kn}, \quad k = 0, 1, \ldots, N-1$$

where:

- X[k] is the DFT coefficient at frequency index k,
- x[n] is the input sequence (time-domain signal),
- N is the total number of samples,
- j = $\sqrt{-1}$ (imaginary unit),
- $e^{-j\frac{2\pi}{N}kn}$ represents the complex exponential basis function.

Inverse Discrete Fourier Transform (IDFT)

The Inverse Discrete Fourier Transform (IDFT) converts the frequency-domain representation back into the time-domain sequence. It is given by:

$$x[n] = \frac{1}{N}\sum_{k=0}^{N-1} X[k]e^{j\frac{2\pi}{N}kn}, \quad n = 0, 1, \ldots, N-1$$

where:

- x[n] is the original time-domain signal,
- X[k] is the DFT coefficient at frequency index k.

Now,the Discrete Fourier Transform (DFT) and the Inverse Discrete Fourier Transform (IDFT) for the sequence:

$$x[n] = \{1, 1, 1, 1\}$$

Compute the DFT :

The DFT formula for an N point sequence is:

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{-j\frac{2\pi}{N}kn}$$

where N=4, and k=0,1,2,3.

For k=0:

$$X[0] = \sum_{n=0}^{3} 1 \cdot e^{-j\frac{2\pi}{4}0n} = 1+1+1+1 = 4$$

For k = 1:

$$X[1] = \sum_{n=0}^{3} 1 \cdot e^{-j\frac{2\pi}{4}(1)n}$$

The exponential terms are:

$$e^{-j0}, e^{-j\frac{\pi}{2}}, e^{-j\pi}, e^{-j\frac{3\pi}{2}}$$

which correspond to:

$$1, -j, -1, j$$

Thus:

$$X[1] = 1 + (-j) + (-1) + j = 0$$

For k = 2 :

$$X[2] = \sum_{n=0}^{3} 1 \cdot e^{-j\frac{2\pi}{4}(2)n}$$

The exponential terms are:

$$e^{-j0}, e^{-j\pi}, e^{-j2\pi}, e^{-j3\pi}$$

which correspond to:

$$\mathbf{1, -1, 1, -1}$$

Thus:

$$X[2] = 1 + (-1) + 1 + (-1) = 0$$

For k = 3:

$$\boldsymbol{X[3] = \sum_{n=0}^{3} 1 \cdot e^{-j\frac{2\pi}{4}(3)n}}$$

The exponential terms are:

$$e^{-j0}, e^{-j\frac{3\pi}{2}}, e^{-j3\pi}, e^{-j\frac{9\pi}{2}}$$

which correspond to:

$$\boldsymbol{1, j, -1, -j}$$

Thus:

$$\boldsymbol{X[3] = 1 + j + (-1) + (-j) = 0}$$

Final DFT Values :

$$X[k] = \{4, 0, 0, 0\}$$

Compute the IDFT :

The Inverse DFT is given by:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{j\frac{2\pi}{N}kn}$$

Since X[K] = {4,0,0,0} , we get:

$$x[n] = \frac{1}{4}\left(4e^{j0} + 0 + 0 + 0\right) = 1$$

for all n = 0,1,2,3.

Final IDFT Values :

$$x[n] = \{1, 1, 1, 1\}$$

For visualize the DFT and IDFT we have to keep mind also :

Zero-Padding

If the length of the sequence is less than N, zero-padding is applied to extend the sequence. This helps in analyzing the frequency characteristics with better resolution.

Magnitude Spectrum

The magnitude spectrum is computed as:

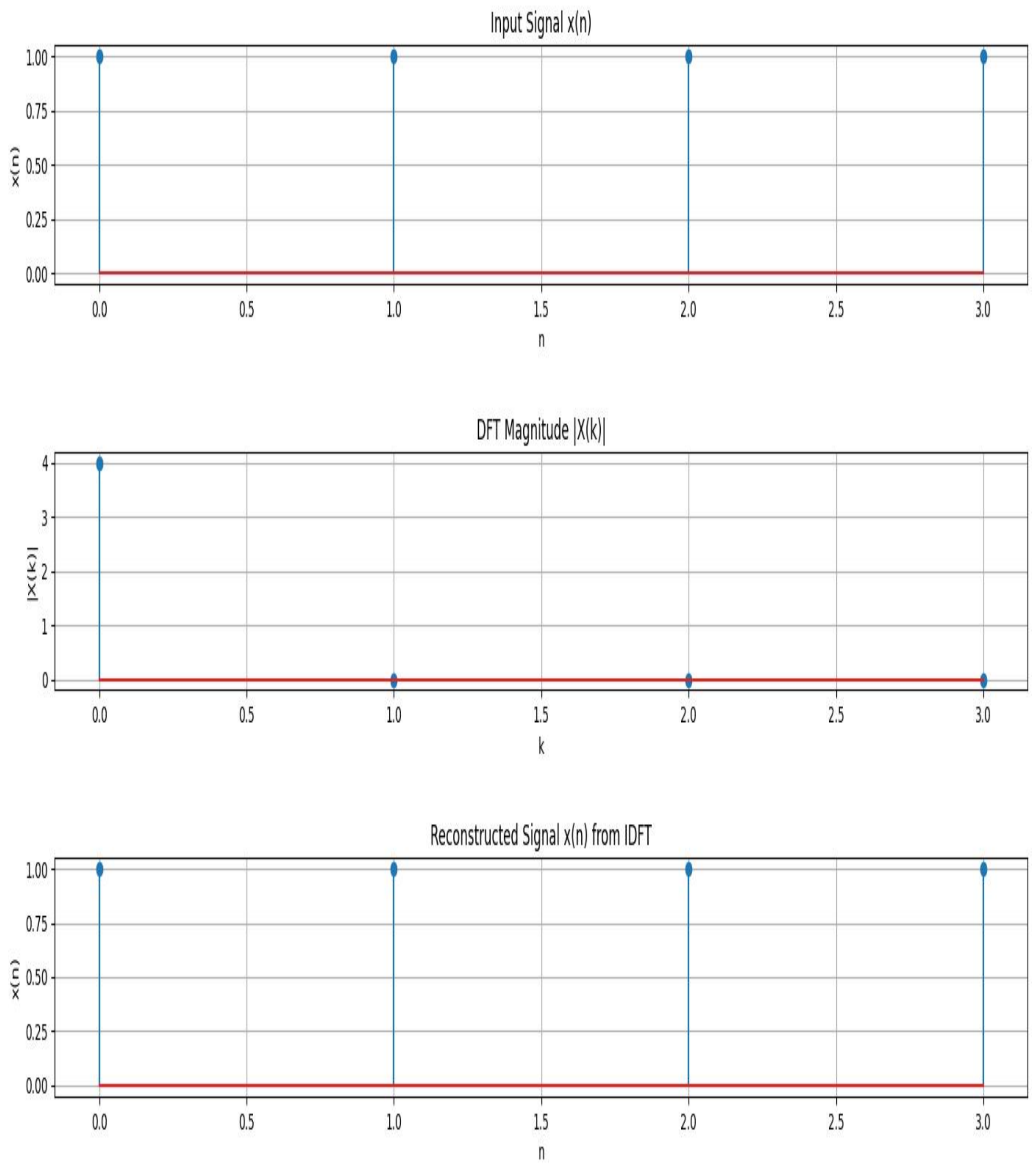
$$| X(k) |= \sqrt{\mathrm{Re}(X(k))^2 + \mathrm{Im}(X(k))^2}$$

<u>Source Code :</u>

```
import numpy as np

import matplotlib.pyplot as plt


# Input sequence and N

x = [1,1,1,1]
```

```
N = 4

# Zero-padding if needed
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 :


Input Signal x(n)
x(n)
n
DFT Magnitude |X(k)|
|X(k)|
k
Reconstructed Signal x(n) from IDFT
x(n)
n

THE END