**Pabna University of Science and Technology**

A logo with a computer and symbols

AI-generated content may be incorrect.

**Faculty of Engineering and Technology**

**Department of Information and Communication Engineering**

**Lab Report**

Course Code: **ICE-2204**

Course title: **Signals and Systems Sessional**

|  |  |
| --- | --- |
| **Submitted By:**  **Name:** Shahriyar  **Roll:** 220616  **Reg:** 1065463  **Session:**2021-2022  2 nd Year 2 nd semester  Department of Information and  Communication Engineering,  Pabna University of Science And Technology, Pabna | **Submitted To:**  **Dr. Md. Imran Hossain**  Associate professor,  Department of  Information and Communication Engineering,  Pabna University of Science And Technology, Pabna. |

**Date of Submission: 02/03/2025**

|  |  |
| --- | --- |
| **Sl.** | **Experiment Name** |
| **1.** | |  | | --- | | Plot following signal operations using user defined function –  1. addition 2.folding 3.signal shifting 4.signal scaling | |
| **2.** | |  | | --- | | implement Discrete-Time Signals: Impulse, Step, and Ramp Signals | |
| **3.** | |  | | --- | | Analysis of Autocorrelation and Cross-Correlation in Signals | |
| **4.** | |  | | --- | | Convolution of Sinusoidal and Noisy Signals | |
| **5.** | |  | | --- | | Heart Rate Estimation from a Synthetic PPG Signal Using Digital Signal Processing | |
| **6.** | |  | | --- | | Discrete Fourier Transform (DFT) and Inverse Discrete Fourier Transform (IDFT) Implementation | |
| **7.** | |  | | --- | | Fourier Series Approximation of a Square Wave | |
| **8.** | |  | | --- | | Analysis of Sinc Function | |

**Index**

**Lab-1: Plot following signal operations using user defined function –**

**1. addition 2.folding 3.signal shifting 4.signal scaling**

**Introduction:**

Signal processing is a fundamental aspect of digital signal processing (DSP), where signals are manipulated for various applications such as communications, control systems, and audio processing. This lab aims to implement basic signal operations using Python, including addition, multiplication, scaling, shifting, and folding. These operations are essential for understanding signal behavior in both time and frequency domains.

**Theory:**

**Signal Addition**

Signal addition involves summing two discrete-time signals element-wise. Mathematically, it is represented as: y[n]=x1[n]+x2[n]

**Signal Multiplication**

Signal multiplication refers to the pointwise product of two signals: This operation is useful in modulation and filtering applications**.**

**Signal Scaling**

Scaling a signal involves multiplying it by a constant factor (): Scaling alters the amplitude of the signal.

**Signal Shifting**

Shifting a signal involves moving it left (negative shift) or right (positive shift) along the time axis: where is the shift amount.

**Signal Folding (Reversal)**

Folding a signal reverses its order along the time axis: This operation is essential in time-reversal and convolution applications.

**Algorithm**

1. Define Discrete-Time Signals:
   * Define the time index array .
   * Define two signals and .
2. Perform Signal Operations:
   * Add and .
   * Multiply and .
   * Scale by a factor of 2.
   * Shift by and .
   * Fold (reverse) .
3. Plot Results:
   * Use matplotlib.pyplot to generate subplots for each operation.
   * Label axes, add titles, and use plt.stem() for discrete-time signals.
4. Display the Plots:
   * Use plt.show() to visualize results.

**4. Observations**

1. Original Signals:
   * The two discrete-time signals are defined with specific amplitude values.
2. Signal Addition:
   * The result is a new signal where each sample is the sum of corresponding samples from and .
3. Signal Multiplication:
   * The result consists of element-wise multiplication of and .
4. Signal Scaling:
   * The signal’s amplitude doubles as expected.
5. Signal Shifting:
   * A left shift () moves the signal earlier in time.
   * A right shift () moves the signal later in time.
6. Signal Folding:
   * The order of samples in is reversed.

**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\_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 :**

A graph of a graph

AI-generated content may be incorrect.

Lab-2: **implement** **Discrete-Time Signals: Impulse, Step, and Ramp Signals**

**Objective:** The primary objective of this experiment is to generate and visualize fundamental discrete-time signals: impulse, step, and ramp signals. These signals are crucial in digital signal processing (DSP) and are widely used in system analysis and control systems.

**Theory:** Discrete-time signals are sequences of values defined at discrete time intervals. Some fundamental discrete-time signals include:

* **Impulse Signal (Unit Impulse Function, )**: Defined as a signal that is 1 at and 0 elsewhere. It is used as a basic building block in system analysis.
* **Step Signal (Unit Step Function, )**: A function that is 0 for negative indices and 1 for zero and positive indices. It is used in system response analysis.
* **Ramp Signal (Unit Ramp Function, )**: A signal that increases linearly with time for non-negative indices. It is a cumulative sum of the unit step function.

These signals play an essential role in DSP as they help in the mathematical modeling of systems.

**Algorithm:**

1. Define the discrete-time range from -10 to 10.
2. Implement functions to generate impulse, step, and ramp signals using NumPy.
3. Compute the signals over the defined range.
4. Plot the signals using Matplotlib's stem function to visualize discrete data points.
5. Label axes, add titles, and configure the layout for better readability.
6. Display the plots.

**Source code:**

import numpy as np

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()

**Results and Observations:**

A graph with blue lines and dots

AI-generated content may be incorrect.

* The impulse signal is observed as a single spike at .
* The step signal remains at 0 for negative and transitions to 1 at .
* The ramp signal starts from 0 and increases linearly for .
* The plots accurately depict the characteristics of each fundamental signal.

Lab-3: **Analysis of Autocorrelation and Cross-Correlation in Signals**

**Objective:** To analyze the properties of autocorrelation and cross-correlation in sinusoidal signals and their variations with noise using computational methods.

**Theory:** Correlation is a mathematical operation used to measure the similarity between signals as a function of time lag. It is widely used in signal processing to identify patterns, detect periodicity, and measure relationships between different signals.

1. **Autocorrelation**: Autocorrelation measures the similarity of a signal with a delayed version of itself. It helps in detecting repeating patterns or periodicity in signals. Mathematically, the autocorrelation of a signal is given by:

where is the time lag.

1. **Cross-Correlation**: Cross-correlation measures the similarity between two different signals as a function of the time lag applied to one of them. It is useful for detecting the time shift between signals. Mathematically, the cross-correlation of signals and is given by:

In this experiment, a sinusoidal signal is analyzed for its autocorrelation properties. Additionally, cross-correlation is computed between the original signal and a shifted version, as well as between the original signal and a noisy version.

**Algorithm:**

1. Define the function compute\_autocorrelation(signal):
   * Compute the autocorrelation using correlate() from the scipy.signal library.
   * Compute the corresponding lags using correlation\_lags().
   * Return the autocorrelation values and lags.
2. Define the function compute\_cross\_correlation(signal1, signal2):
   * Compute the cross-correlation using correlate().
   * Compute the corresponding lags using correlation\_lags().
   * Return the cross-correlation values and lags.
3. Generate a sinusoidal signal:
   * Define the sampling frequency .
   * Create a time vector using np.linspace().
   * Generate a sine wave using .
4. Compute the autocorrelation of the sinusoidal signal.
5. Compute the cross-correlation between the original signal and a shifted version.
6. Add Gaussian noise to the original signal and compute its cross-correlation with the noise-affected signal.
7. Plot the autocorrelation and cross-correlation results using matplotlib.

**Source code:**

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

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

freq = 5

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()

**Results and Discussion:**

A diagram of a wave

AI-generated content may be incorrect.

* The autocorrelation plot shows a periodic pattern, reflecting the periodic nature of the sine wave.
* The cross-correlation between the original and shifted signal indicates a peak at the shift value, showing the time delay.
* The cross-correlation of the noisy signal still retains some characteristics of the original signal but is affected by noise, as seen in the variations.

**Lab-4: Convolution of Sinusoidal and Noisy Signals**

**Objective:** The objective of this experiment is to understand and analyze the convolution of signals using Python. The experiment includes computing the convolution of a sinusoidal signal with itself (autoconvolution), convolution with a time-shifted version of the signal, and convolution with a noisy version of the signal.

**Theory:** Convolution is a fundamental operation in signal processing that combines two signals to produce a third signal, reflecting how one signal modifies the other. Mathematically, convolution of two discrete-time signals and is defined as:

where:

* is the first signal.
* is the second signal, flipped and shifted.

In this experiment, we perform three different types of convolutions:

1. **Autoconvolution**: A sinusoidal signal is convolved with itself.
2. **Convolution with a Shifted Signal**: A time-shifted version of the sinusoidal signal is used.
3. **Convolution with a Noisy Signal**: The effect of noise is analyzed through convolution.

**Algorithm:**

1. Define a function to compute the convolution using the scipy.signal.convolve method.
2. Generate a sinusoidal signal with a given sampling frequency and frequency.
3. Compute and plot the autoconvolution of the sinusoidal signal.
4. Shift the signal in time and compute its convolution with the original signal.
5. Introduce Gaussian noise to the sinusoidal signal and compute the convolution with the original signal.
6. Plot the results for analysis.

**Source code:**

import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import convolve

def compute\_convolution(signal1, signal2):

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

fs = 1000

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

freq = 5

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()

**Results and Discussion:**

A diagram of a wave

AI-generated content may be incorrect.

* The autoconvolution plot demonstrates symmetry and periodic behavior, characteristic of the sinusoidal signal.
* The convolution with a shifted signal shows a phase-shifted pattern.
* The convolution with a noisy signal exhibits distortions due to the added Gaussian noise.

**Lab-5:** **Heart Rate Estimation from a Synthetic PPG Signal Using Digital Signal Processing**

**Introduction**

Photoplethysmography (PPG) is a non-invasive technique for measuring blood volume changes in the microvascular tissue using light absorption. The PPG signal can be used to estimate heart rate by detecting periodic peaks corresponding to heartbeats. This report presents a method for processing a synthetic PPG signal using digital signal processing techniques to estimate heart rate.

**Theory**

A PPG signal is typically composed of a low-frequency baseline, pulsatile heart-related components, and noise. To extract useful information, signal processing techniques such as filtering and peak detection are applied.

* **Bandpass Filtering:** A Butterworth bandpass filter is used to remove noise and retain frequency components relevant to heart rate.
* **Peak Detection:** Peaks in the filtered signal correspond to heartbeats and are detected using a threshold-based method.
* **Heart Rate Calculation:** The time interval between successive peaks (RR intervals) is used to estimate heart rate in beats per minute (BPM).

**Algorithm**

1. **Generate Synthetic PPG Signal:**
   * Create a sine wave with noise to simulate a PPG signal.
   * Sampling frequency (fs) is set to 100 Hz.
2. **Bandpass Filtering:**
   * Apply a Butterworth bandpass filter with cut-off frequencies of 0.5 Hz and 5.0 Hz.
   * Use scipy.signal.butter() and scipy.signal.filtfilt() for filtering.
3. **Signal Normalization:**
   * Scale the filtered signal between 0 and 1.
4. **Peak Detection:**
   * Identify peaks using scipy.signal.find\_peaks() with a minimum distance constraint to avoid false detections.
5. **Heart Rate Calculation:**
   * Compute RR intervals (time differences between successive peaks).
   * Calculate heart rate using the formula:
6. **Visualization:**
   * Plot the raw sine wave, noise, PPG signal, filtered signal, and normalized signal.
   * Highlight detected peaks in the final processed PPG signal.

**Source code:**

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)

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

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("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")

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='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()

**Implementation and Results:**

A group of blue and white graph

AI-generated content may be incorrect.

* The filtered signal is obtained and visually inspected.
* Peaks corresponding to heartbeats are successfully detected.
* The estimated heart rate is printed and verified based on the synthetic signal properties.
* Plots illustrate the transformation of the signal from raw to filtered and processed states.

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

**Objective:** The objective of this lab is to compute the Discrete Fourier Transform (DFT) and its inverse (IDFT) using Python's NumPy library. This implementation demonstrates the transformation of a discrete signal from the time domain to the frequency domain and back to the time domain, verifying the correctness of the IDFT.

**Theory:** The Discrete Fourier Transform (DFT) is a fundamental technique used in signal processing to analyze the frequency content of discrete signals. Given a sequence , the DFT is computed as:

where ranges from to . This transformation converts a time-domain sequence into its frequency-domain representation.

The Inverse Discrete Fourier Transform (IDFT) is used to reconstruct the original signal from its frequency components:

In this experiment, the DFT and IDFT are computed using NumPy's fft.fft() and fft.ifft() functions, respectively.

**Algorithm:**

1. Define the input sequence and specify the value of , the number of points for transformation.
2. Zero-pad the sequence if necessary to ensure it has a length of .
3. Compute the DFT using the numpy.fft.fft() function.
4. Compute the IDFT using the numpy.fft.ifft() function.
5. Print the computed DFT and IDFT values.
6. Plot:
   * The original signal .
   * The magnitude of the DFT coefficients.
   * The reconstructed signal from IDFT.
7. Display the plots for analysis and comparison.

**Source code:**

import numpy as np

import matplotlib.pyplot as plt

x = [1, 1, 1, 1]

N = 4

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

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

x\_reconstructed = np.fft.ifft(X)

print("DFT values:", X)

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

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()

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()

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()

**Results and Discussion:**

A graph with red lines and black text

AI-generated content may be incorrect.

* The DFT values represent the frequency components of the input signal.
* The magnitude plot of DFT shows the strength of different frequency components.
* The reconstructed signal from IDFT matches the original signal, confirming the correctness of the transformation.
* The experiment demonstrates the utility of DFT and IDFT in analyzing and reconstructing signals efficiently.

**Lab-7: Fourier Series Approximation of a Square Wave**

**Objective:** The objective of this lab is to analyze the Fourier series representation of a square wave and observe how increasing the number of terms improves the approximation.

**Theory:** A square wave is a periodic function that alternates between two levels, typically -1 and 1. The Fourier series is a mathematical tool used to represent periodic functions as a sum of sine and cosine functions. The Fourier series representation of a square wave (with period ) is given by:

where only the odd harmonics contribute to the sum. As more terms are included, the approximation of the square wave improves, though some oscillations (Gibbs phenomenon) appear near the discontinuities.

**Algorithm:**

1. Define the Fourier series function that sums the sine terms up to a specified number of terms.
2. Define the original square wave function using a piecewise approach.
3. Generate x-values ranging from to .
4. Compute the Fourier series approximation for different numbers of terms.
5. Plot the original square wave alongside the Fourier approximations.
6. Observe how the approximation improves as the number of terms increases.

**Implementation:**

1. Import necessary libraries (NumPy for computations and Matplotlib for plotting).
2. Define a function fourier\_series(x, terms) that calculates the Fourier series sum for a given number of terms.
3. Define a function square\_wave(x) that returns 1 or -1 based on the sign of sin(x).
4. Generate a range of x-values using np.linspace(-np.pi, np.pi, 400).
5. Use Matplotlib to plot the original square wave and different approximations (e.g., with 1, 3, 5, and 9 terms).
6. Display the plot to visualize the convergence behavior.

**Source code:**

import numpy as np

import matplotlib.pyplot as plt

def fourier\_series(x, terms):

if terms < 1:

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

result = np.zeros\_like(x)

for n in range(1, terms + 1, 2):

result += (4 / (np.pi \* n)) \* np.sin(n \* x)

return result

def square\_wave(x):

return np.where(np.sin(x) >= 0, 1, -1)

t = np.linspace(-np.pi, np.pi, 400)

plt.figure(figsize=(8, 6))

plt.plot(t, square\_wave(t), label='Original Square Wave', linestyle='--', color='black')

for terms in [1, 3, 5, 9]:

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()

**Result and Observations:**

A graph of different colored lines

AI-generated content may be incorrect.

* With fewer terms, the approximation is crude and smooth, missing sharp transitions.
* As more terms are added, the square wave shape becomes more apparent.
* The Gibbs phenomenon is visible near discontinuities, where overshooting occurs.

**Lab-8:** **Analysis of Sinc Function**

**Objective:** To analyze and visualize the sinc function using Python, including its real, phase, and magnitude components.

**Theory:** The sinc function, denoted as , is defined as: It plays a crucial role in signal processing, particularly in Fourier analysis and interpolation of band-limited signals. The function exhibits a central peak at and oscillates with diminishing amplitude as increases.

A scaled version of the sinc function, such as , modifies its frequency characteristics. The function's real part, phase, and magnitude provide valuable insights into its behavior in the time domain.

**Algorithm:**

1. **Initialize Parameters:** Define the time range from -2 to 2 with increments of 0.01.
2. **Compute the Sinc Function:** Evaluate the function using NumPy.
3. **Plot Real Part:** Display the function's real component over time.
4. **Plot Phase Part:** Compute and visualize the phase of the sinc function.
5. **Plot Magnitude Part:** Compute and display the magnitude response of the sinc function.
6. **Enhance Readability:** Label axes, provide titles, and apply grid formatting.
7. **Display Graphs:** Render the plots for analysis.

**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))

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.ylabel('Amplitude')

plt.title('Magnitude Part')

plt.grid()

plt.tight\_layout()

plt.show()

**Results and Discussion:**

A diagram of a graph

AI-generated content may be incorrect.

The output graphs illustrate different properties of the sinc function:

1. The **real part** represents the sinc function oscillations.
2. The **phase plot** depicts the phase shift at different points in time.
3. The **magnitude plot** highlights the amplitude variations of the function.

These plots help in understanding the behavior of the sinc function in signal processing applications, such as filtering and reconstruction of band-limited signals.