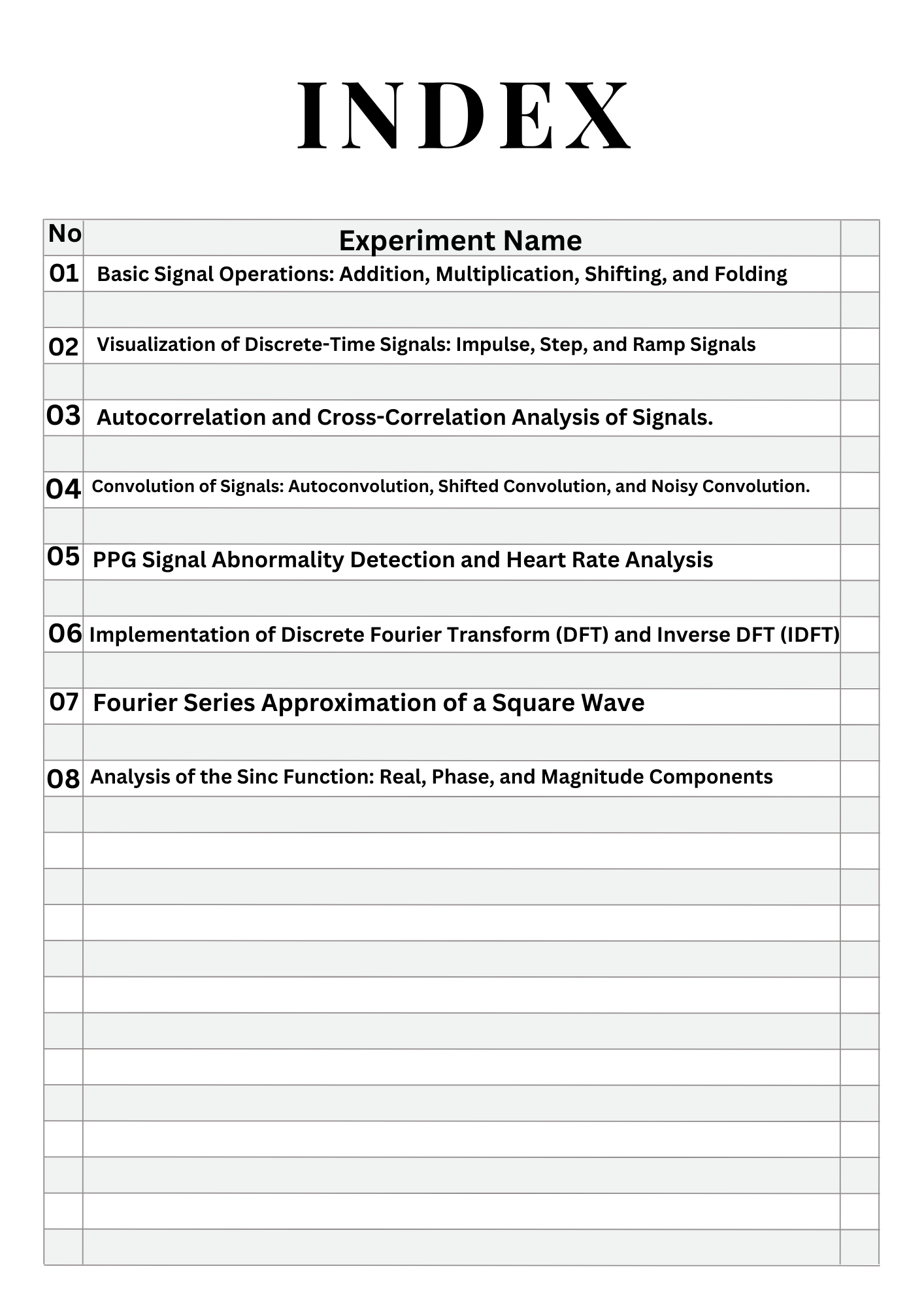
### 



### **Experiment No:01**

### **Title**:Basic Signal Operations: Addition, Multiplication, Shifting, and Folding

### **Objectives:**

The purpose of this program is to perform and visualize fundamental signal processing operations on discrete-time signals, including:

1. Addition of two signals
2. Multiplication of two signals
3. Time Shifting (delaying/advancing a signal)
4. Time Folding (reversing a signal in time)

These operations help in analyzing and manipulating signals in digital signal processing (DSP) applications such as filtering, modulation, and system analysis.

**Theory:**

Digital signals are represented as sequences of numbers (discrete-time signals). The following operations are fundamental in signal processing:

1. Signal Addition
   * When two signals x1[n]x\_1[n]x1​[n] and x2[n]x\_2[n]x2​[n] are added, the result is: y[n]=x1[n]+x2[n]y[n] = x\_1[n] + x\_2[n]y[n]=x1​[n]+x2​[n]
   * This operation is used in applications like signal mixing and interference analysis.
2. Signal Multiplication
   * When two signals x1[n]x\_1[n]x1​[n] and x2[n]x\_2[n]x2​[n] are multiplied element-wise: y[n]=x1[n]×x2[n]y[n] = x\_1[n] \times x\_2[n]y[n]=x1​[n]×x2​[n]
   * This operation is useful in modulation and filtering.
3. Time Shifting
   * A signal x[n]x[n]x[n] can be shifted by adding a constant kkk to the time index: y[n]=x[n−k]y[n] = x[n-k]y[n]=x[n−k]
   * If kkk is positive, the signal shifts to the right (delayed).
   * If kkk is negative, the signal shifts to the left (advanced).
4. Time Folding (Reversal)
   * Time folding flips the signal along the vertical axis, reversing its time indices: y[n]=x[−n]y[n] = x[-n]y[n]=x[−n]
   * This operation is useful in convolution and symmetry analys**is.**

**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\_shifting(n, shift):**

**return n + shift**

**def signal\_folding(n, x):**

**return -n, 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)**

**shifted\_signal1 = signal\_shifting(n, -2)**

**shifted\_signal2 = signal\_shifting(n, 2)**

**folded\_n, folded\_signal = signal\_folding(n, x1)**

**plt.figure(figsize=(12, 8))**

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

**plt.stem(n, x1)**

**plt.xlabel("Time")**

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

**plt.title("Original Signal x1")**

**plt.grid()**

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

**plt.stem(n, x2)**

**plt.xlabel("Time")**

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

**plt.title("Original Signal x2")**

**plt.grid()**

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

**plt.stem(n, added\_signal)**

**plt.xlabel("Time")**

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

**plt.title("Signal Addition (x1 + x2)")**

**plt.grid()**

**plt.subplot(3, 2, 4)**

**plt.stem(n, multiplied\_signal)**

**plt.xlabel("Time")**

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

**plt.title("Signal Multiplication (x1 \* x2)")**

**plt.grid()**

**plt.subplot(3, 2, 5)**

**plt.stem(shifted\_signal1, x1)**

**plt.xlabel("Time")**

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

**plt.title("Shifted Signal (Shift = -2)")**

**plt.grid()**

**plt.subplot(3, 2, 6)**

**plt.stem(folded\_n, folded\_signal)**

**plt.xlabel("Time")**

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

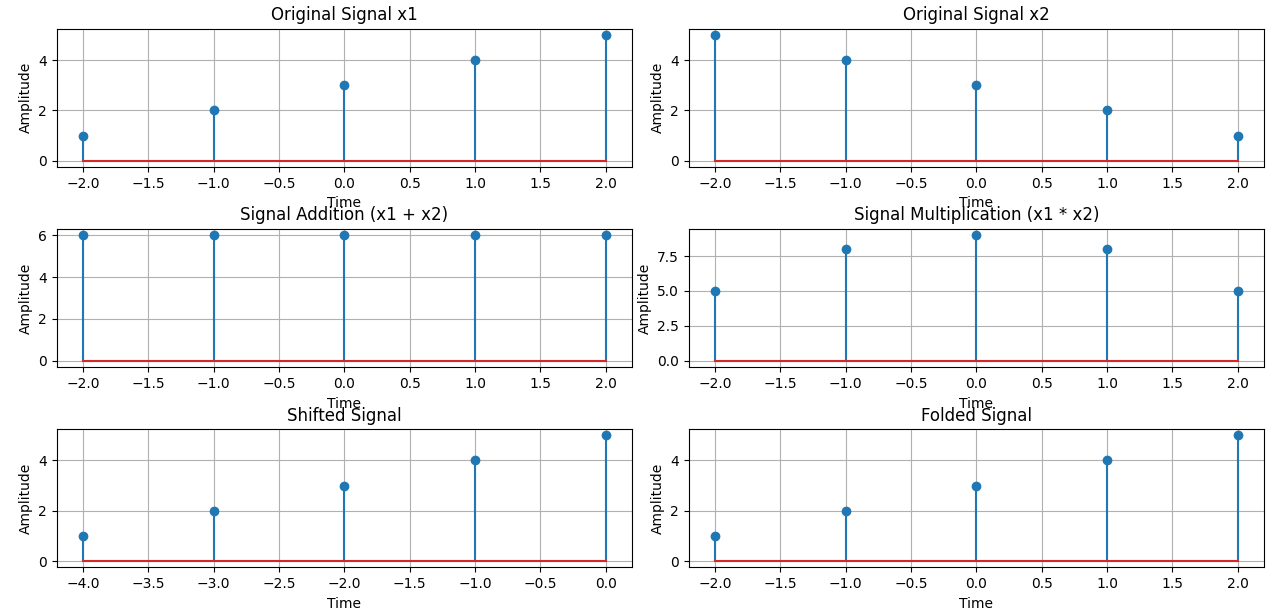
**plt.title("Folded Signal (Time Reversed x1)")**

**plt.grid()**

**plt.tight\_layout()**

**plt.show()**

**Output:**

****

**Purpose:**

The purpose of this Python code is to perform and visualize basic operations on discrete-time signals using NumPy and Matplotlib. The operations include:

1. Signal Addition: Computes the sum of two discrete-time signals x1x1x1 and x2x2x2.
2. Signal Multiplication: Computes the element-wise product of x1x1x1 and x2x2x2.
3. Signal Shifting: Shifts the signal along the time axis (both left and right shifts).
4. Signal Folding (Time Reversal): Reverses the signal in time.

The code then plots:

* The original signals x1x1x1 and x2x2x2.
* The added and multiplied signals.
* A time-shifted version of x1x1x1 (shift = -2).
* A folded (time-reversed) version of x1x1x1.

### Use Case

This type of signal processing is fundamental in digital signal processing (DSP) and is useful in areas such as:

* Audio and image processing
* Communications and control systems
* Understanding basic transformations in signals and systems

**Experiment No:02**

**Title**:Visualization of Discrete-Time Signals: Impulse, Step, and Ramp Signals

**Theory:**

A discrete-time signal is a sequence of values defined at discrete points in time. In this experiment, we explore three fundamental discrete-time signals: Impulse Signal, Step Signal, and Ramp Signal. These signals are essential in digital signal processing (DSP), control systems, and system analysis.

#### **1️.Impulse Signal (δ[n])**

* The Impulse Signal, also called the unit impulse function, is defined as: δ[n]={1,n=00,n≠0\delta[n] = \begin{cases} 1, & n = 0 \\ 0, & n \neq 0 \end{cases}δ[n]={1,0,​n=0n=0​
* It is used to analyze system response and is fundamental in convolution operations.

#### **2️.Step Signal (u[n])**

* The **Step Signal**, or **unit step function**, is defined as: u[n]={1,n≥00,n<0u[n] = \begin{cases} 1, & n \geq 0 \\ 0, & n < 0 \end{cases}u[n]={1,0,​n≥0n<0​
* It represents a signal that switches on at n=0n = 0n=0 and remains constant afterward.
* Used in system stability analysis and digital control systems.

#### **3️.Ramp Signal (r[n])**

* The **Ramp Signal** is a linearly increasing function, defined as: r[n]={n,n≥00,n<0r[n] = \begin{cases} n, & n \geq 0 \\ 0, & n < 0 \end{cases}r[n]={n,0,​n≥0n<0​
* It models increasing trends, such as velocity-to-position conversion in physics.

By plotting these signals, we visualize their behavior and understand their significance in system analysis and signal processing.

### **Objectives:**

The primary objectives of this experiment are:  
1. To understand the concept of **Impulse, Step, and Ramp signals** in discrete time.  
2.To visualize these signals using **Python and Matplotlib**.  
3.To analyze their behavior and mathematical representations.  
4.To explore their significance in **signal processing, system response, and control systems**.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

n = np.arange(-10, 11)

signals = {

"Impulse Signal": lambda n: np.where(n == 0, 1, 0),

"Step Signal": lambda n: np.where(n >= 0, 1, 0),

"Ramp Signal": lambda n: np.where(n >= 0, n, 0),

}

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

for i, (title, func) in enumerate(signals.items(), 1):

plt.subplot(1, 3, i)

plt.stem(n, func(n))

plt.title(title)

plt.xlabel("n")

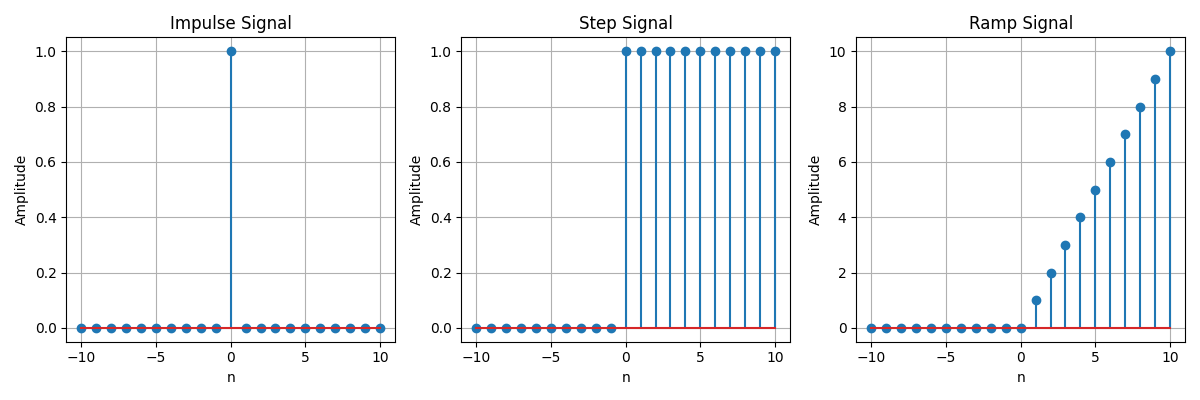
plt.ylabel("Amplitude")

plt.grid()

plt.tight\_layout()

plt.show()

**Output:**

****

# **Purpose:**

The purpose of this Python code is to generate and visualize basic discrete-time signals commonly used in digital signal processing (DSP). The signals plotted include:

1. **Impulse Signal (δ[n]):** A signal that is 1 at n=0n = 0n=0 and 0 elsewhere.
2. **Step Signal (u[n]):** A signal that is 1 for n≥0n \geq 0n≥0 and 0 otherwise.
3. **Ramp Signal (r[n]):** A signal that increases linearly for n≥0n \geq 0n≥0 and is 0 for n<0n < 0n<0.

### **Use Case**

These signals are fundamental in DSP and system analysis:

* The impulse signal is used for system impulse response analysis.
* The step signal is used for system stability and response evaluation.
* The ramp signal helps in understanding system behavior under linear growth inputs.

# 

# **Experiment No:03**

# **Title:**Autocorrelation and Cross-Correlation Analysis of Signals.

## **Objectives:**

The purpose of this program is to analyze signals using autocorrelation and cross-correlation techniques. This helps in:

1. Understanding periodicity in a signal using autocorrelation.
2. Measuring the similarity between two signals using cross-correlation.
3. Examining the effect of noise on correlation.

These techniques are widely used in signal processing, radar, speech recognition, and communications for detecting patterns and extracting useful information from signals.

**Theory:**

### **1. Autocorrelation:**

* Autocorrelation measures how similar a signal is to a time-shifted version of itself.
* It helps identify periodicity in a signal and is useful in applications like pitch detection and system analysis.
* Mathematically, the autocorrelation function is given by:  
  Rxx(τ)=∑nx(n)x(n−τ)R\_{xx}(\tau) = \sum\_{n} x(n) x(n - \tau)Rxx​(τ)=n∑​x(n)x(n−τ)  
  where x(n)x(n)x(n) is the signal, and τ\tauτ is the time lag.

### **2. Cross-Correlation:**

* Cross-correlation measures the similarity between two different signals as one is shifted in time.
* It is useful for time-delay estimation, signal alignment, and pattern recognition.
* The cross-correlation function is given by:  
  Rxy(τ)=∑nx(n)y(n−τ)R\_{xy}(\tau) = \sum\_{n} x(n) y(n - \tau)Rxy​(τ)=n∑​x(n)y(n−τ)  
  where x(n)x(n)x(n) and y(n)y(n)y(n) are two different signals.

### **3. Noise Effect on Cross-Correlation:**

* When noise is added to a signal, its correlation with the original signal decreases.
* This is useful in filtering, denoising, and detecting signals in noisy environments.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import correlate

def autocorrelation(signal):

return correlate(signal, signal, mode='full')

def cross\_correlation(signal1, signal2):

return correlate(signal1, signal2, mode='full')

fs = 100

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

signal1 = np.sin(2 \* np.pi \* 5 \* t)

signal2 = np.roll(signal1, 10)

noise = np.random.normal(0, 0.3, fs)

noisy\_signal = signal1 + noise

auto\_corr = autocorrelation(signal1)

cross\_corr = cross\_correlation(signal1, signal2)

cross\_corr\_noise = cross\_correlation(signal1, noisy\_signal)

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

plt.subplot(3, 1, 1)

plt.plot(auto\_corr)

plt.title("Autocorrelation of Signal")

plt.grid()

plt.subplot(3, 1, 2)

plt.plot(cross\_corr)

plt.title("Cross-Correlation of Shifted Signals")

plt.grid()

plt.subplot(3, 1, 3)

plt.plot(cross\_corr\_noise)

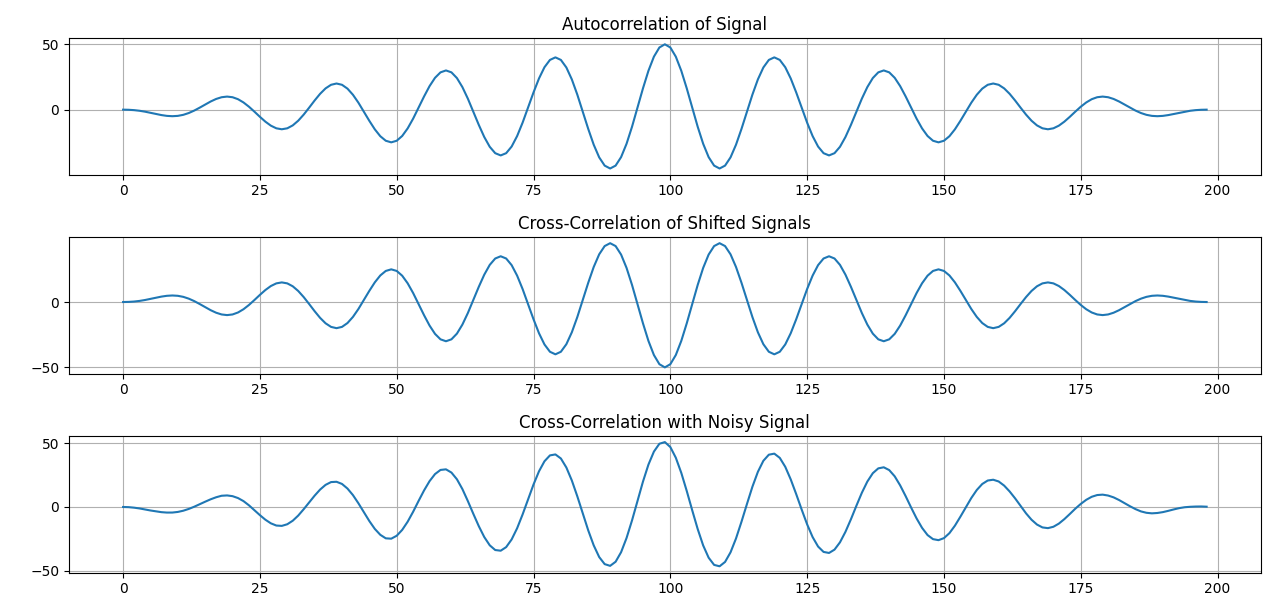
plt.title("Cross-Correlation with Noisy Signal")

plt.grid()

plt.tight\_layout()

plt.show()

**Output:**

****

**Purpose:**

This code computes and visualizes autocorrelation (signal similarity with itself) and cross-correlation (similarity between two signals) using a sinusoidal signal, a shifted version, and a noisy version. It helps analyze signal periodicity, detect time shifts, and assess noise effects, useful in signal processing and communication systems.

# 

# **Experiment No:04**

# **Title:**Convolution of Signals: Autoconvolution, Shifted Convolution, and Noisy Convolution.

# **Theory:**

### **1. Convolution:**

* Convolution is a mathematical operation that combines two signals to produce a third signal.
* It is widely used in signal processing, image processing, and system analysis.
* The convolution formula is:  
  y(n)=∑x(k)h(n−k)y(n) = \sum x(k) h(n - k)y(n)=∑x(k)h(n−k)  
  where x(k)x(k)x(k) is the input signal, and h(k)h(k)h(k) is another signal (like an impulse response).

### 2. Types of Convolution in This Code:

**2.Autoconvolution:**

* Convolution of a signal with itself.
* Helps in identifying periodicity and patterns.

**3.Shifted Convolution:**

* Convolution of a signal with a time-shifted version.
* Used in time-delay estimation.

**4.Noisy Convolution:**

* Convolution of a signal with its noisy version.
* Helps in understanding signal distortion due to noise.

**Source code:**

import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import convolve

def convolution(signal1, signal2):

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

fs = 100

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

signal1 = np.sin(2 \* np.pi \* 5 \* t)

signal2 = np.roll(signal1, 10)

noise = np.random.normal(0, 0.3, fs)

noisy\_signal = signal1 + noise

conv\_auto = convolution(signal1, signal1)

conv\_shifted = convolution(signal1, signal2)

conv\_noisy = convolution(signal1, noisy\_signal)

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

plt.subplot(3, 1, 1)

plt.plot(conv\_auto)

plt.title("Autoconvolution of Signal")

plt.grid()

plt.subplot(3, 1, 2)

plt.plot(conv\_shifted)

plt.title("Convolution with Shifted Signal")

plt.grid()

plt.subplot(3, 1, 3)

plt.plot(conv\_noisy)

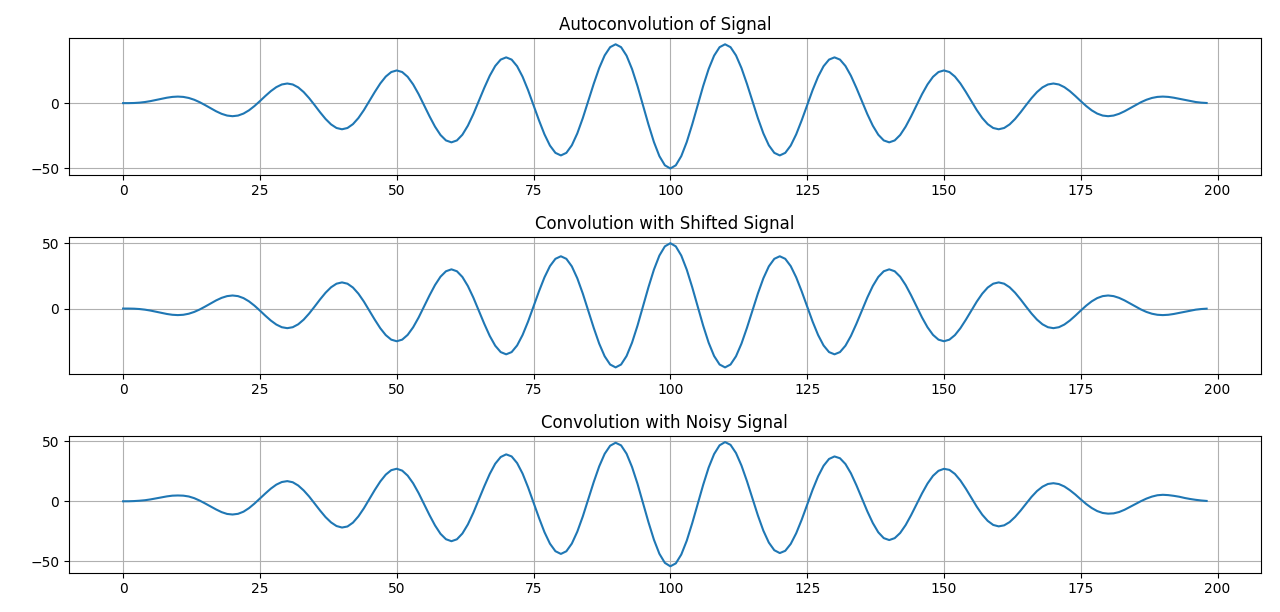
plt.title("Convolution with Noisy Signal")

plt.grid()

plt.tight\_layout()

plt.show()

**Output:**



**Purpose:**

This code computes and visualizes convolution of signals, which is fundamental in signal processing for filtering and system response analysis. It performs:

1. Autoconvolution (signal convolved with itself).
2. Convolution with a shifted version (measuring alignment).
3. Convolution with a noisy signal (analyzing signal impact in noise).

Useful in filtering, system analysis, and signal transformation applications.

### **Experiment no:05**

### **Title:** PPG Signal Abnormality Detection and Heart Rate Analysis

## **1. Theory**

Photoplethysmography (PPG) is a non-invasive optical technique used to measure blood volume changes in microvascular tissue. It works by shining light onto the skin and detecting variations in light absorption caused by blood flow. PPG signals are widely used in medical applications for monitoring heart rate, detecting cardiovascular abnormalities, and analyzing heart rate variability (HRV).

Key concepts used in this project:

* **PPG Signal Processing**: Raw PPG data often contains noise from movement, respiration, and other artifacts, which requires filtering.
* **Peak and Valley Detection**: Heartbeats are identified from peaks and valleys in the PPG waveform.
* **Heart Rate Calculation**: The interval between successive peaks (R-R intervals) is used to determine beats per minute (BPM).
* **Abnormality Detection**: Unexpected high peaks, deep valleys, and irregular intervals indicate arrhythmia, tachycardia, or bradycardia.
* **Fourier Transform Analysis (FFT)**: Helps in noise identification and filtering from the PPG signal.
* **Heart Rate Variability (HRV) Analysis**: HRV metrics such as SDNN (Standard Deviation of NN Intervals), RMSSD (Root Mean Square of Successive Differences), and PNN50 (% of intervals differing by 50ms) are used to assess autonomic nervous system function.

## **2. Objectives**

The main objectives of this project are:  
1. To process raw PPG signals and remove noise using filtering techniques.  
2.To detect normal and abnormal peaks and valleys in the PPG signal.  
3.To calculate heart rate (BPM) from detected peaks and compare raw vs. filtered heart rate.  
 1.To identify abnormal conditions such as tachycardia (high heart rate), bradycardia (low heart rate), and irregular beats.  
 2.To analyze Heart Rate Variability (HRV) to assess cardiovascular health.  
3.To apply Fast Fourier Transform (FFT) for frequency domain analysis of noise.

**Source Code:**

## import numpy as np

## import matplotlib.pyplot as plt

## from scipy.signal import find\_peaks, butter, filtfilt

## from scipy.fft import fft, fftfreq

## np.random.seed(42)

## fs = 100

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

## ppg\_signal = np.sin(2 \* np.pi \* 1.2 \* t) + 0.3 \* np.random.randn(len(t))

## ppg\_signal[300] += 2

## ppg\_signal[700] -= 2

## ppg\_signal[850] += 1.5

## def bandpass\_filter(signal, lowcut=0.5, highcut=5, fs=100, order=3):

## nyquist = 0.5 \* fs

## low = lowcut / nyquist

## high = highcut / nyquist

## b, a = butter(order, [low, high], btype='band')

## return filtfilt(b, a, signal)

## filtered\_ppg = bandpass\_filter(ppg\_signal)

## def detect\_peaks\_and\_heart\_rate(ppg, label):

## peaks, \_ = find\_peaks(ppg, height=0.3, distance=50)

## valleys, \_ = find\_peaks(-ppg, height=0.3, distance=50)

## peak\_threshold = np.mean(ppg[peaks]) + 1.5 \* np.std(ppg[peaks])

## valley\_threshold = np.mean(ppg[valleys]) - 1.5 \* np.std(ppg[valleys])

## 

## abnormal\_peaks = [p for p in peaks if ppg[p] > peak\_threshold]

## abnormal\_valleys = [v for v in valleys if ppg[v] < valley\_threshold]

## if len(peaks) > 1:

## peak\_intervals = np.diff(t[peaks])

## avg\_peak\_interval = np.mean(peak\_intervals)

## heart\_rate = 60 / avg\_peak\_interval

## else:

## heart\_rate = 0

## return peaks, valleys, abnormal\_peaks, abnormal\_valleys, heart\_rate, peak\_interval

## peaks\_raw, valleys\_raw, abnormal\_peaks\_raw, abnormal\_valleys\_raw, heart\_rate\_raw, intervals\_raw = detect\_peaks\_and\_heart\_rate(ppg\_signal, "Raw PPG")

## peaks\_filtered, valleys\_filtered, abnormal\_peaks\_filtered, abnormal\_valleys\_filtered, heart\_rate\_filtered, intervals\_filtered = detect\_peaks\_and\_heart\_rate(filtered\_ppg, "Filtered PPG")

## heart\_rate\_difference = abs(heart\_rate\_raw - heart\_rate\_filtered)

## def calculate\_hrv(intervals):

## if len(intervals) > 1:

## sdnn = np.std(intervals)

## rmssd = np.sqrt(np.mean(np.square(np.diff(intervals))))

## pnn50 = np.sum(np.abs(np.diff(intervals)) > 0.05) / len(intervals) \* 100

## return sdnn, rmssd, pnn50

## return 0, 0, 0

## sdnn\_raw, rmssd\_raw, pnn50\_raw = calculate\_hrv(intervals\_raw)

## sdnn\_filtered, rmssd\_filtered, pnn50\_filtered = calculate\_hrv(intervals\_filtered)

## fft\_vals = np.abs(fft(ppg\_signal))

## fft\_freqs = fftfreq(len(ppg\_signal), 1/fs)

## plt.figure(figsize=(12, 8))

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

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

## plt.scatter(t[peaks\_raw], ppg\_signal[peaks\_raw], color='green', marker='o', label="Normal Peaks")

## plt.scatter(t[valleys\_raw], ppg\_signal[valleys\_raw], color='orange', marker='o', label="Normal Valleys")

## plt.scatter(t[abnormal\_peaks\_raw], ppg\_signal[abnormal\_peaks\_raw], color='red', marker='x', label="Abnormal Peaks")

## plt.scatter(t[abnormal\_valleys\_raw], ppg\_signal[abnormal\_valleys\_raw], color='purple', marker='x', label="Abnormal Valleys")

## plt.xlabel("Time (s)")

## plt.ylabel("Amplitude")

## plt.title(f"Raw PPG Signal (Heart Rate: {heart\_rate\_raw:.1f} BPM)")

## plt.legend()

## plt.grid()

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

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

## plt.scatter(t[peaks\_filtered], filtered\_ppg[peaks\_filtered], color='green', marker='o', label="Normal Peaks")

## plt.scatter(t[valleys\_filtered], filtered\_ppg[valleys\_filtered], color='orange', marker='o', label="Normal Valleys")

## plt.scatter(t[abnormal\_peaks\_filtered], filtered\_ppg[abnormal\_peaks\_filtered], color='red', marker='x', label="Abnormal Peaks")

## plt.scatter(t[abnormal\_valleys\_filtered], filtered\_ppg[abnormal\_valleys\_filtered], color='purple', marker='x', label="Abnormal Valleys")

## plt.xlabel("Time (s)")

## plt.ylabel("Amplitude")

## plt.title(f"Filtered PPG Signal (Heart Rate: {heart\_rate\_filtered:.1f} BPM)")

## plt.legend()

## plt.grid()

## 

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

## plt.plot(fft\_freqs[:len(fft\_freqs)//2], fft\_vals[:len(fft\_vals)//2], color='purple')

## plt.xlabel("Frequency (Hz)")

## plt.ylabel("Magnitude")

## plt.title("FFT Analysis of PPG Signal (Noise Detection)")

## plt.grid()

## plt.tight\_layout()

## plt.show()

## print(f"Heart Rate from Raw PPG: {heart\_rate\_raw:.1f} BPM")

## print(f"Heart Rate from Filtered PPG: {heart\_rate\_filtered:.1f} BPM")

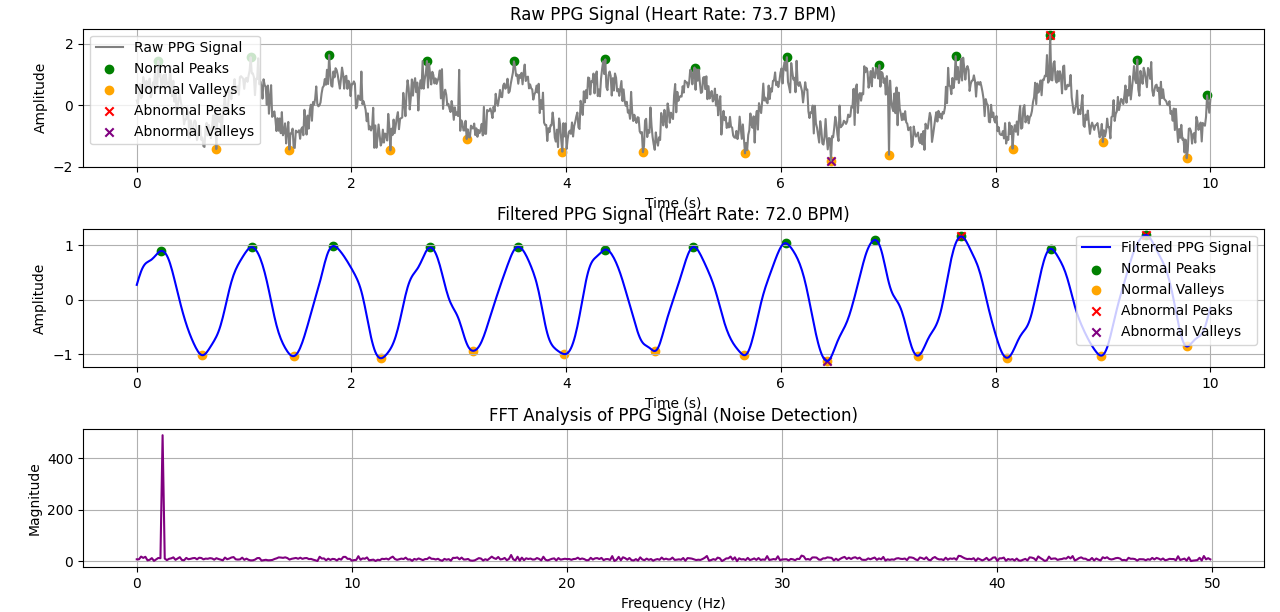
## print(f"Heart Rate Difference due to Noise/Abnormalities: {heart\_rate\_difference:.1f} BPM")

## print(f"HRV (Raw) - SDNN: {sdnn\_raw:.3f}, RMSSD: {rmssd\_raw:.3f}, PNN50: {pnn50\_raw:.3f}%")

## print(f"HRV (Filtered) - SDNN: {sdnn\_filtered:.3f}, RMSSD: {rmssd\_filtered:.3f}, PNN50: {pnn50\_filtered:.3f}%")

## 

## **Output:**



## **3. Purpose**

This project aims to develop an automated PPG signal processing system that can:  
1. Detect abnormalities in heart rate patterns to help diagnose potential cardiac conditions.  
2.Improve heart rate accuracy by comparing raw vs. filtered signals.  
3. Analyze heart rate variability (HRV) for deeper insights into cardiovascular health.  
4. Enhance signal quality by using Fourier Transform (FFT) for noise filtering.  
5. Provide real-time PPG monitoring for wearable health applications.

This system can be applied in medical diagnostics, fitness tracking, and real-time heart monitoring systems to ensure early detection of cardiovascular anomalies and improve patient outcomes.

### **Experiment No:**06

### **Title:**Implementation of Discrete Fourier Transform (DFT) and Inverse DFT (IDFT)

### **Theory:**

The Discrete Fourier Transform (DFT) is a mathematical technique used to analyze the frequency content of discrete-time signals. It transforms a finite sequence of equally spaced samples into a complex-valued frequency representation. The formula for DFT is:

X(k)=∑n=0N−1x(n)e−j(2πkn/N)X(k) = \sum\_{n=0}^{N-1} x(n) e^{-j(2\pi kn/N)}X(k)=n=0∑N−1​x(n)e−j(2πkn/N)

where:

* X(k)X(k)X(k) represents the DFT coefficients (frequency components).
* x(n)x(n)x(n) is the input discrete-time signal.
* NNN is the number of points in the transform.
* kkk is the frequency index.
* jjj is the imaginary unit.

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

x(n)=1N∑k=0N−1X(k)ej(2πkn/N)x(n) = \frac{1}{N} \sum\_{k=0}^{N-1} X(k) e^{j(2\pi kn/N)}x(n)=N1​k=0∑N−1​X(k)ej(2πkn/N)

The Fast Fourier Transform (FFT) is an efficient algorithm used to compute the DFT in O(Nlog⁡N)O(N \log N)O(NlogN) time complexity instead of the direct computation, which has O(N2)O(N^2)O(N2) complexity.

### **Objectives:**

1. To understand and implement DFT and IDFT using NumPy's fft and ifft functions.
2. To visualize the input signal, DFT magnitude spectrum, and reconstructed signal.
3. To verify that IDFT accurately reconstructs the original signal.
4. To demonstrate the use of FFT algorithm for efficient computation of DFT.
5. To observe how Fourier Transform decomposes a time-domain signal into its frequency components.

**Souce 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, 8))

plt.subplot(4, 1, 1)

plt.stem(range(len(x)), x)

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

plt.xlabel('n')

plt.ylabel('x(n)')

plt.grid()

plt.subplot(4, 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(4, 1, 3)

plt.stem(range(N), np.angle(X))

plt.title('DFT Phase ∠X(k)')

plt.xlabel('k')

plt.ylabel('∠X(k) (Radians)')

plt.grid()

plt.subplot(4, 1, 4)

plt.stem(range(N), x\_reconstructed.real)

plt.title('Reconstructed Signal x(n) (IDFT)')

plt.xlabel('n')

plt.ylabel('x(n)')

plt.grid()

plt.tight\_layout()

plt.show()

plt.figure(figsize=(10, 4))

plt.subplot(2, 1, 1)

plt.stem(range(N), X.real)

plt.title('Real Part of DFT')

plt.xlabel('k')

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

plt.grid()

plt.subplot(2, 1, 2)

plt.stem(range(N), X.imag)

plt.title('Imaginary Part of DFT')

plt.xlabel('k')

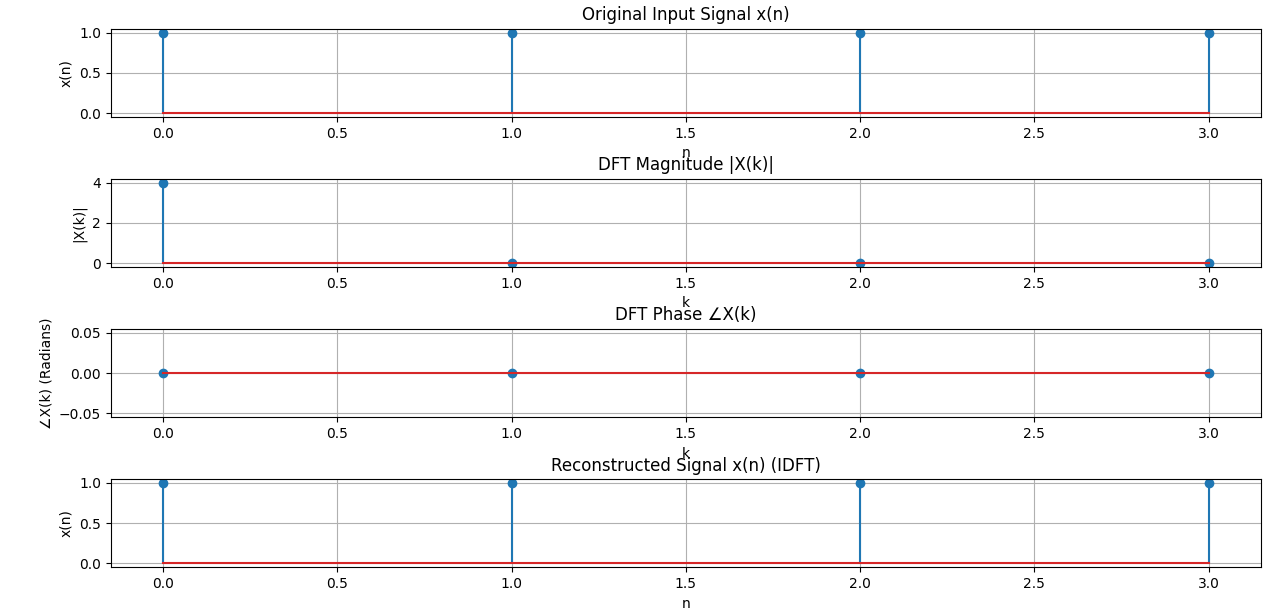
plt.ylabel('Im(X(k))')

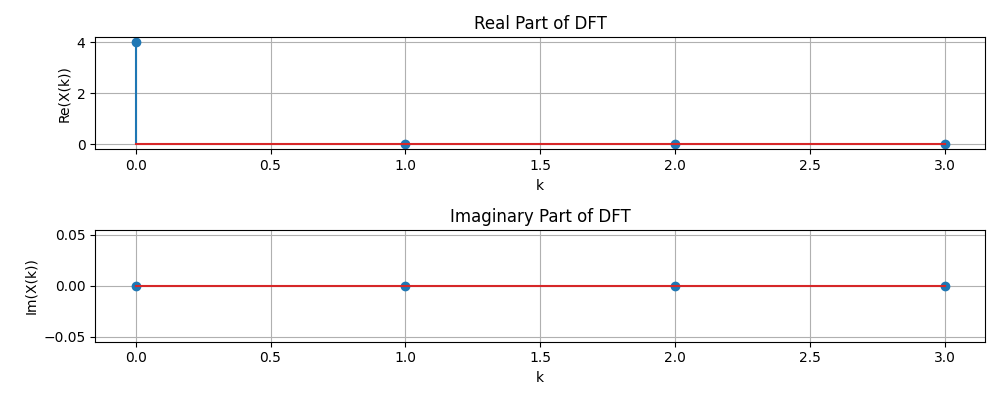
plt.grid()

plt.tight\_layout()

plt.show()

**Output:**

****

****

**Purpose:**

This code demonstrates the Discrete Fourier Transform (DFT) and its inverse (IDFT) using NumPy. It analyzes a simple discrete-time signal and visualizes its transformation in the frequency domain.

### Key Steps:

1. Define Input Signal: A basic sequence [1,1,1,1] is padded to length N=4.
2. Compute DFT: Using np.fft.fft(), transforming the signal into the frequency domain.
3. Compute IDFT: Using np.fft.ifft(), reconstructing the original signal.
4. Visualizations:
   * Original Signal
   * DFT Magnitude Spectrum
   * DFT Phase Spectrum
   * Reconstructed Signal from IDFT
   * Real & Imaginary Parts of DFT

### Purpose & Applications:

* Frequency Analysis: Understand how signals decompose into sinusoids.
* Signal Processing & Filtering: Useful in audio, image, and communication systems.
* Verification: Ensures accurate transformation and reconstruction.

### **Experiment No:07**

**Title:**Fourier Series Approximation of a Square Wave

**Theory:**The Fourier series is a mathematical tool used to represent periodic functions as a sum of sine and cosine functions. It is particularly useful in signal processing, physics, and engineering.

For a square wave function, the Fourier series expansion is given by:

f(x)=∑n=1,3,5,…∞4nπsin⁡(nx)f(x) = \sum\_{n=1,3,5,\dots}^{\infty} \frac{4}{n\pi} \sin(n x)f(x)=n=1,3,5,…∑∞​nπ4​sin(nx)

where nnn includes only odd integers. This expansion allows us to approximate the square wave using a finite number of sinusoidal components.

One important property observed in Fourier series approximations of discontinuous functions is the Gibbs phenomenon, which refers to the overshoot near discontinuities that does not vanish even as more terms are added.

### **Objectives:**

1. To understand the concept of Fourier series and how it is used to approximate periodic functions.
2. To visualize the approximation of a square wave using a finite number of Fourier terms.
3. To observe the effects of increasing the number of terms, particularly how accuracy improves and the Gibbs phenomenon appears.
4. To analyze the convergence behavior of the Fourier series approximation.

**Source Code:**

**import numpy as np**

**import matplotlib.pyplot as plt**

**def fourier\_series(x, terms):**

**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 Square Wave')**

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

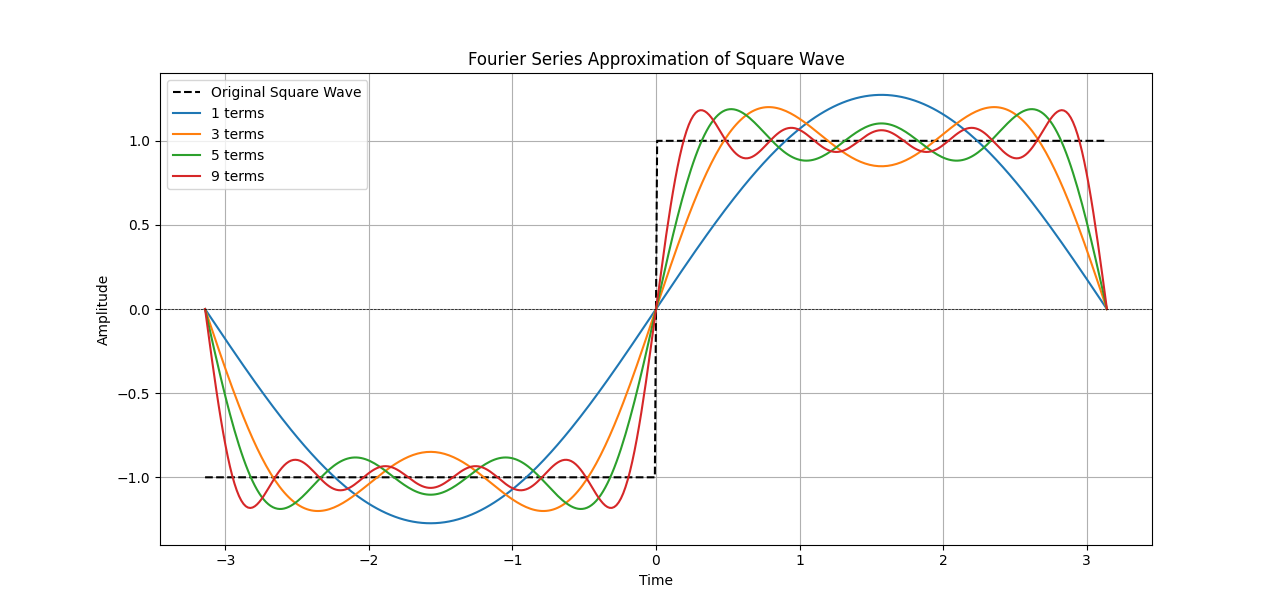
**plt.ylabel('Amplitude')**

**plt.legend()**

**plt.grid()**

**plt.show()**

**Output:**

****

**Purpose:**

This code approximates a square wave using the Fourier series by summing sine wave components. It visualizes the original wave and its approximations with 1, 3, 5, and 9 terms, showing how more terms improve accuracy. Useful for signal processing, Fourier analysis, and wave decomposition.

**Experiment No:**08

# **Title:**Analysis of the Sinc Function: Real, Phase, and Magnitude Components

## **Objectives:**

The main objectives of this program are:

1.Understanding the Sinc Function: Visualizing its properties.  
2.Analyzing the Real Part: Observing the main oscillatory behavior.  
3.Examining the Phase Component: Understanding phase variations.  
4.Studying the Magnitude Component: Exploring the absolute amplitude response.

## **Theory:**

### **1. Sinc Function**

* The sinc function is defined as:  
  sinc(t)=sin⁡(πt)πt\text{sinc}(t) = \frac{\sin(\pi t)}{\pi t}sinc(t)=πtsin(πt)​  
  with the property that sinc(0) = 1 and decays as ttt increases.
* In this case, we use 4×sinc(4t)4 \times \text{sinc}(4t)4×sinc(4t), which means:
  + The function is scaled by 4 in amplitude.
  + The frequency components are stretched by 4, making the function narrower.

### **2. Components of the Sinc Function:**

**Real Part:**

* Since the sinc function is purely **real**, this plot represents the actual waveform.

**Phase Part:**

* Phase is calculated using **np.angle(x)**, but since the sinc function is purely real, its phase is either **0 or π** (depending on the sign).

**Magnitude Part:**

* The magnitude shows the **absolute** values of the sinc function, representing its envelope.

**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)**

**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()**

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

**plt.plot(t, np.angle(x))**

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

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

**plt.title('Phase Part')**

**plt.grid()**

**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()**

**Output:**

