

INDEX

|  |  |
| --- | --- |
| SI No. | Name of Experiment |
| 01. | Signal Operations (Addition, Subtraction, Multiplication, Division, Shifting and Folding. |
| 02. | Signal Sequences (Impulse Signal, Step Signal , Ramp Signal). |
| 03. | Convolution using Python |
| 04. | Correlation (Cross and Auto) Using Python. |
| 05. | PPG signal lab (a sine signal,noise signal ,signal with add noise ,filtered signal, filtering, feature extraction, peak detection). |
| 06. | Fourier Transform using python. |
| 07. | Fourier Series Decomposition with Harmonics using python. |
| 08. | A lab report on DFT using python. |

**Lab: 01**

**Lab name:** Signal Operations (Addition, Subtraction, Multiplication, Division, Shifting and Folding.

**Objectives:**

* Understand the theory behind fundamental signal operations.
* Implement basic signal operations using Python.
* Illustrate how these operations are used in the context of digital signal processing.

**Theory: Signal Operations-**

1. **Addition of Signals:**
   * Signal addition is the process of adding two signals together. It is used in various applications such as combining different signals or adding noise to a signal.
2. **Subtraction of Signals:**

* Signal subtraction is the process of subtracting one signal from another. It can be used for noise reduction or extracting specific components from a composite signal.

1. **Multiplication of Signals:**

* **Theory:** Signal multiplication involves multiplying two signals together. It is used in modulation, filtering, and various other applications.

1. **Division of Signals:**

* **Theory:** Signal division is the process of dividing one signal by another. This is useful in applications like signal normalization or in certain filtering techniques.

1. **Shifting of Signals:**

* **Theory:** Shifting a signal in time is the process of delaying or advancing the signal in time. It can be represented as adding or subtracting a constant from the time index.

1. **Folding of Signals:**

* **Theory:** Folding or time reversal involves reversing the time index of a signal. It is used in applications like time-domain signal analysis or convolution.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

# Define a time vector and two sample signals

t = np.linspace(0, 1, 1000)

signal1 = np.sin(2 \* np.pi \* 5 \* t) # 5 Hz sine wave

signal2 = 0.5 \* np.sin(2 \* np.pi \* 10 \* t) # 10 Hz sine wave with half amplitude

# 1. Addition of Signals

result\_add = signal1 + signal2

# 2. Subtraction of Signals

result\_subtract = signal1 - signal2

# 3. Multiplication of Signals

result\_multiply = signal1 \* signal2

# 4. Division of Signals (Avoiding division by zero)

result\_divide = np.divide(signal1, signal2, where=(signal2 != 0))

# 5. Shifting of Signals

shift\_amount = 200 # number of samples to shift

shifted\_signal = np.roll(signal1, shift\_amount)

# 6. Folding of Signals (Time Reversal)

folded\_signal = np.flip(signal1)

# Plotting all results

plt.figure(figsize=(14, 10))

# Subplot 1: Addition of Signals

plt.subplot(3, 2, 1)

plt.plot(t, result\_add, label="Signal 1 + Signal 2")

plt.title("Addition of Signals")

plt.xlabel("Time [s]")

plt.ylabel("Amplitude")

plt.legend()

# Subplot 2: Subtraction of Signals

plt.subplot(3, 2, 2)

plt.plot(t, result\_subtract, label="Signal 1 - Signal 2")

plt.title("Subtraction of Signals")

plt.xlabel("Time [s]")

plt.ylabel("Amplitude")

plt.legend()

# Subplot 3: Multiplication of Signals

plt.subplot(3, 2, 3)

plt.plot(t, result\_multiply, label="Signal 1 \* Signal 2")

plt.title("Multiplication of Signals")

plt.xlabel("Time [s]")

plt.ylabel("Amplitude")

plt.legend()

# Subplot 4: Division of Signals

plt.subplot(3, 2, 4)

plt.plot(t, result\_divide, label="Signal 1 / Signal 2")

plt.title("Division of Signals")

plt.xlabel("Time [s]")

plt.ylabel("Amplitude")

plt.legend()

# Subplot 5: Shifting of Signals

plt.subplot(3, 2, 5)

plt.plot(t, shifted\_signal, label="Shifted Signal")

plt.title("Shifting of Signal")

plt.xlabel("Time [s]")

plt.ylabel("Amplitude")

plt.legend()

# Subplot 6: Folding of Signals

plt.subplot(3, 2, 6)

plt.plot(t, folded\_signal, label="Folded Signal")

plt.title("Folding of Signal")

plt.xlabel("Time [s]")

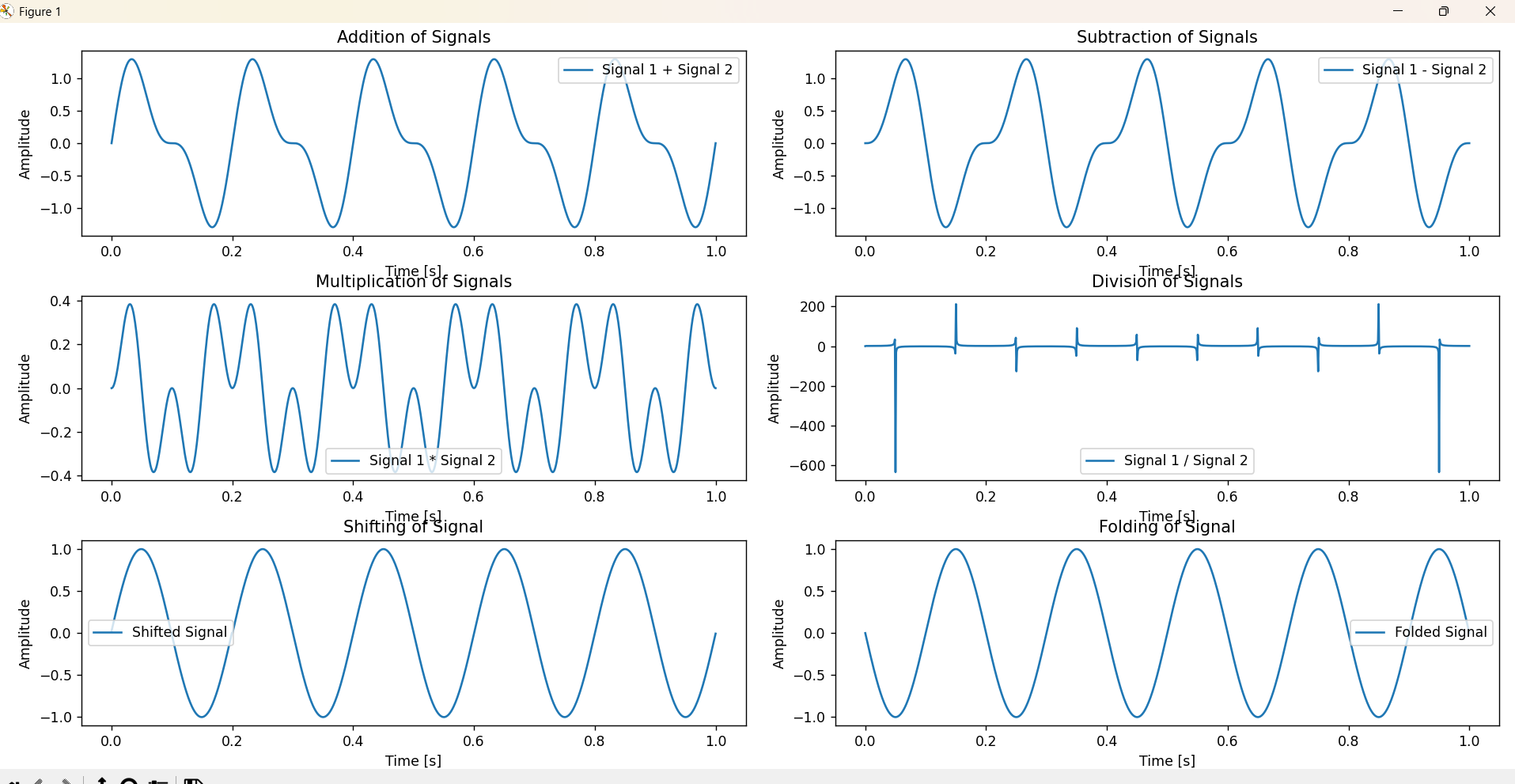
plt.ylabel("Amplitude")

plt.legend()

plt.tight\_layout()

plt.show()

Output:



Lab: 02

Lab name: Signal Sequences (Impulse Signal, Step Signal , Ramp Signal).

**Objectives:**

The objective of this report is to understand and implement three fundamental signal sequences used in signal processing:

1. **Impulse Signal** (Unit Impulse)
2. **Step Signal** (Unit Step)
3. **Ramp Signal**

**Theory:**

**1. Impulse Signal (Unit Impulse)**

The **Impulse Signal**, often represented as **δ(t)** or **δ[n]** in discrete time, is a mathematical function that takes the value of 1 at a specific point (typically at t = 0) and 0 elsewhere. In continuous time, it is a Dirac delta function, and in discrete time.

**2. Step Signal (Unit Step)**

The **Step Signal**, also known as the **Heaviside step function** in continuous time and **u[n]** in discrete time. The step signal is often used to model systems that switch on at a particular time.

**3. Ramp Signal**

The **Ramp Signal** is a continuous-time signal that starts at 0 and increases linearly with time. The ramp signal is often used to simulate systems with linearly increasing inputs.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

# Time settings for discrete and continuous time

n = np.arange(-10, 11)  # Discrete time for -10 to 10

t = np.linspace(-10, 10, 400)  # Continuous time for -10 to 10

# Impulse Signal (Unit Impulse) for Discrete

impulse\_discrete = np.zeros\_like(n)

impulse\_discrete[n == 0] = 1

# Step Signal (Unit Step) for Discrete

step\_discrete = np.zeros\_like(n)

step\_discrete[n >= 0] = 1

# Ramp Signal for Discrete

ramp\_discrete = np.zeros\_like(n)

ramp\_discrete[n >= 0] = n[n >= 0]

# Impulse Signal for Continuous (Dirac-like approximation)

impulse\_continuous = np.zeros\_like(t)

impulse\_continuous[np.abs(t) < 1e-6] = 1  # Dirac impulse approximation

# Step Signal for Continuous

step\_continuous = np.zeros\_like(t)

step\_continuous[t >= 0] = 1

# Ramp Signal for Continuous

ramp\_continuous = np.zeros\_like(t)

ramp\_continuous[t >= 0] = t[t >= 0]

# Plotting

fig, axs = plt.subplots(3, 2, figsize=(12, 12))

# Plot Impulse Signal

axs[0, 0].stem(n, impulse\_discrete, basefmt=" ")  # Removed use\_line\_collection

axs[0, 0].set\_title("Discrete Impulse Signal")

axs[0, 0].set\_xlabel('n')

axs[0, 0].set\_ylabel('Amplitude')

axs[0, 1].plot(t, impulse\_continuous, label='Continuous Impulse')

axs[0, 1].set\_title("Continuous Impulse Signal")

axs[0, 1].set\_xlabel('t')

axs[0, 1].set\_ylabel('Amplitude')

# Plot Step Signal

axs[1, 0].stem(n, step\_discrete, basefmt=" ")  # Removed use\_line\_collection

axs[1, 0].set\_title("Discrete Step Signal")

axs[1, 0].set\_xlabel('n')

axs[1, 0].set\_ylabel('Amplitude')

axs[1, 1].plot(t, step\_continuous, label='Continuous Step')

axs[1, 1].set\_title("Continuous Step Signal")

axs[1, 1].set\_xlabel('t')

axs[1, 1].set\_ylabel('Amplitude')

# Plot Ramp Signal

axs[2, 0].stem(n, ramp\_discrete, basefmt=" ")  # Removed use\_line\_collection

axs[2, 0].set\_title("Discrete Ramp Signal")

axs[2, 0].set\_xlabel('n')

axs[2, 0].set\_ylabel('Amplitude')

axs[2, 1].plot(t, ramp\_continuous, label='Continuous Ramp')

axs[2, 1].set\_title("Continuous Ramp Signal")

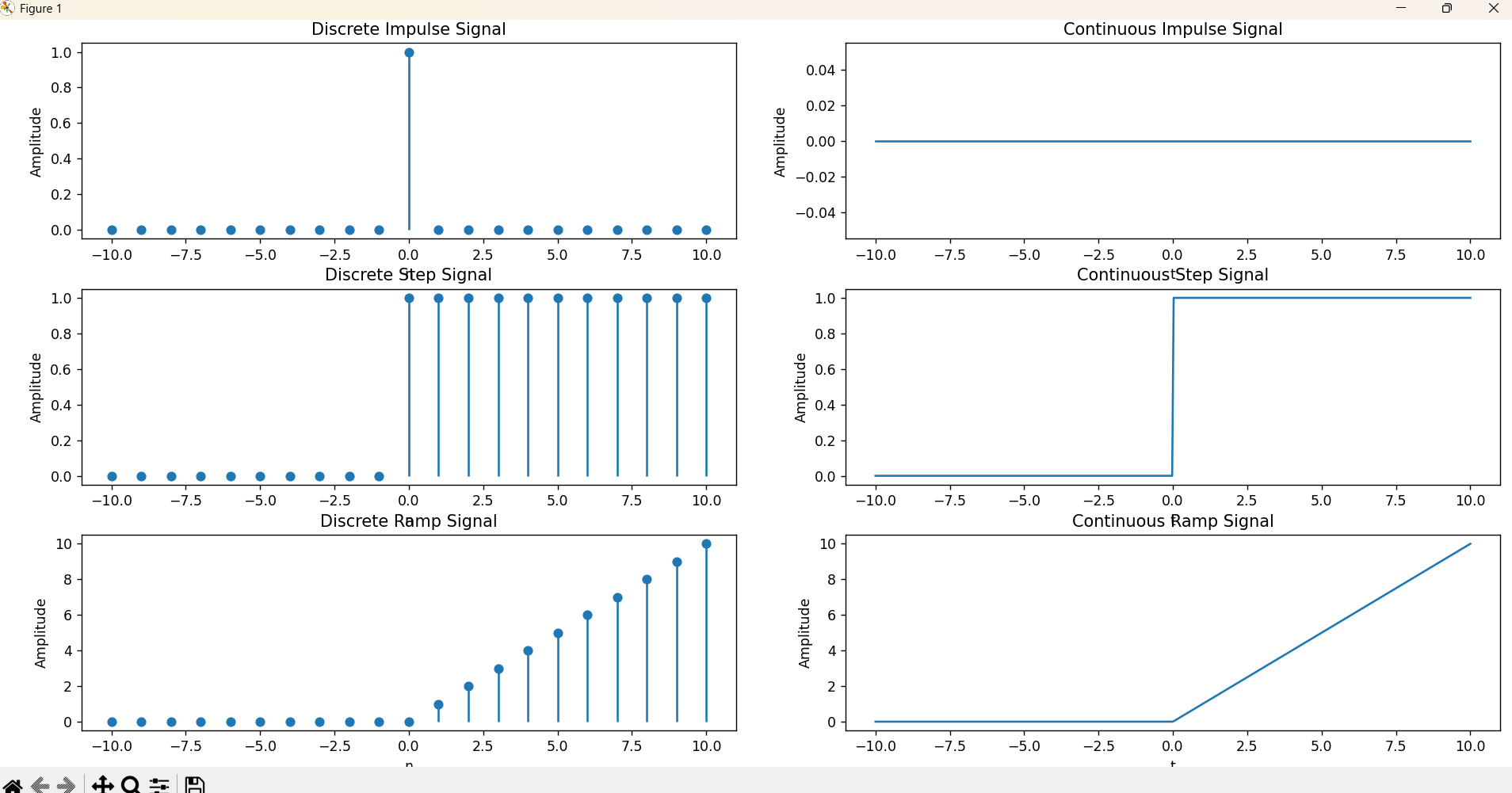
axs[2, 1].set\_xlabel('t')

axs[2, 1].set\_ylabel('Amplitude')

plt.tight\_layout()

plt.show()

**Output:**



**Lab: 03**

**Lan name:** Convolution using Python.

**Objectives:**

1. **Image Filtering**: To extract features like edges, corners, and textures.
2. **Blurring**: To smoothen or reduce noise.
3. **Sharpening**: To enhance edges and improve contrast.
4. **Edge Detection**: To find boundaries between different regions in an image.
5. **Feature extraction**: In neural networks, convolutions help extract low-level features (like edges) that can be used for higher-level pattern recognition.

**Theory:**

Convolution is a mathematical operation that combines two functions to create a third function. It is widely used in signal processing, image processing, and neural networks. In the context of images, convolution involves sliding a filter (or kernel) over an image to produce an output that highlights certain features of the image (like edges, textures, or blurring effects).

For a given image, the convolution operation is performed by applying a kernel (a small matrix) to each local region of the image. The output value is computed by taking the dot product of the kernel and the region of the image it overlaps.

Source code:

import numpy as np

import matplotlib.pyplot as plt

def plot\_signal(n, x, title):

    plt.stem(n, x)

    plt.xlabel('n')

    plt.ylabel('Amplitude')

    plt.title(title)

    plt.grid()

    plt.show()

# Define discrete signals

x = np.array([1, 2, 3, 4])  # Input signal

h = np.array([1, -1, 2])  # Impulse response

# Perform convolution

y = np.convolve(x, h)

n = np.arange(len(y))

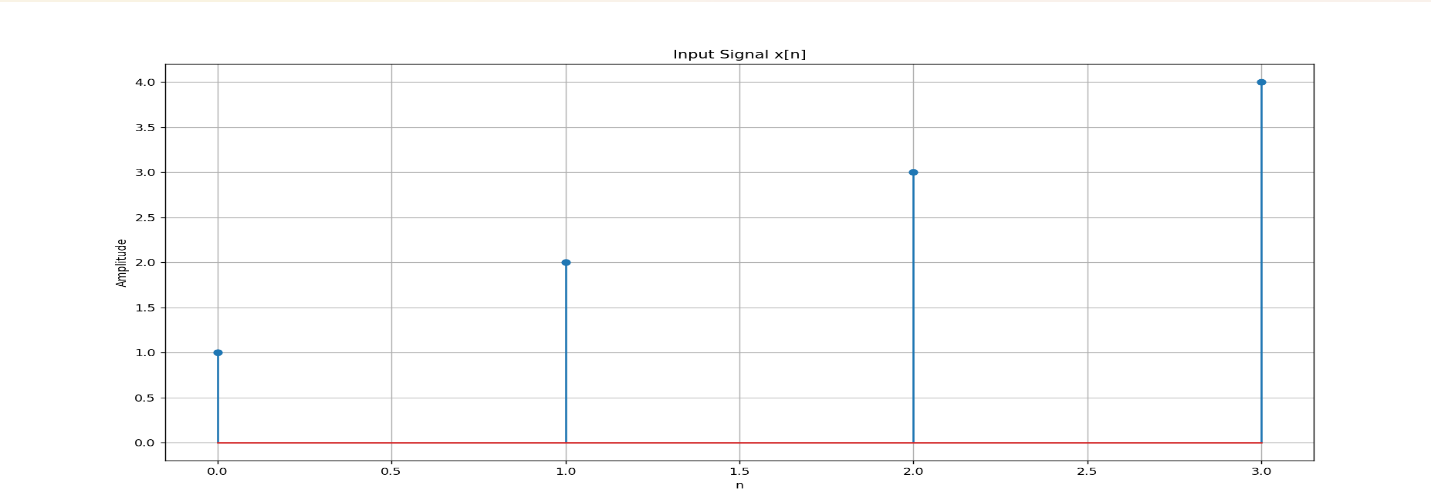
# Plot signals

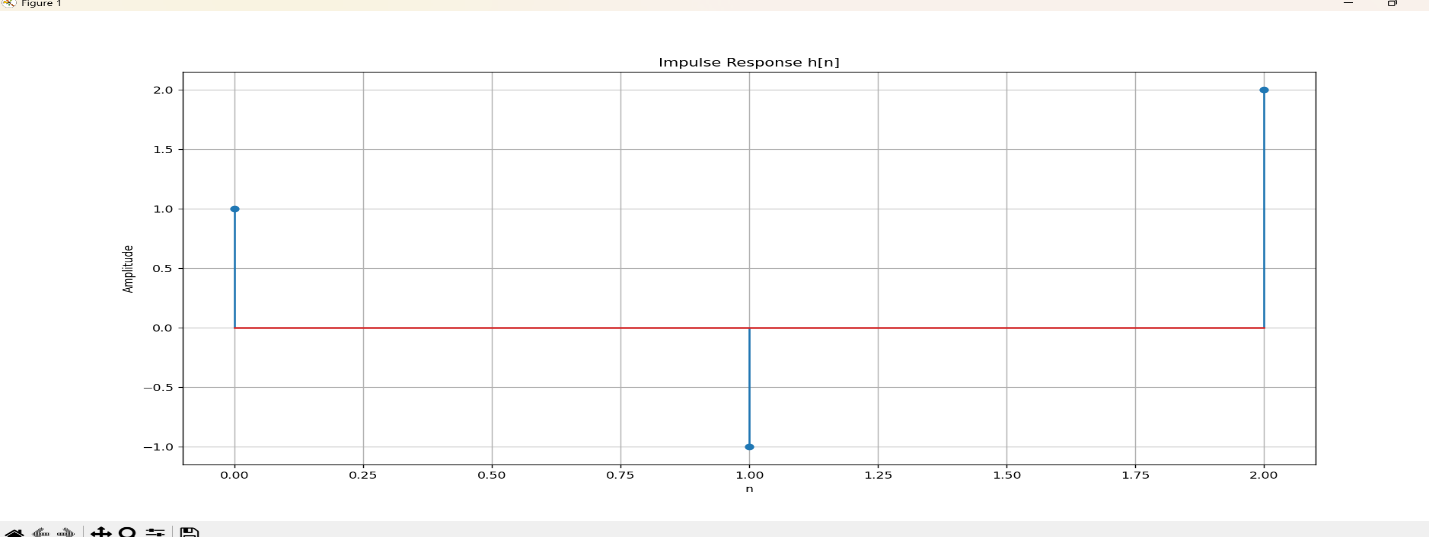
plot\_signal(np.arange(len(x)), x, 'Input Signal x[n]')

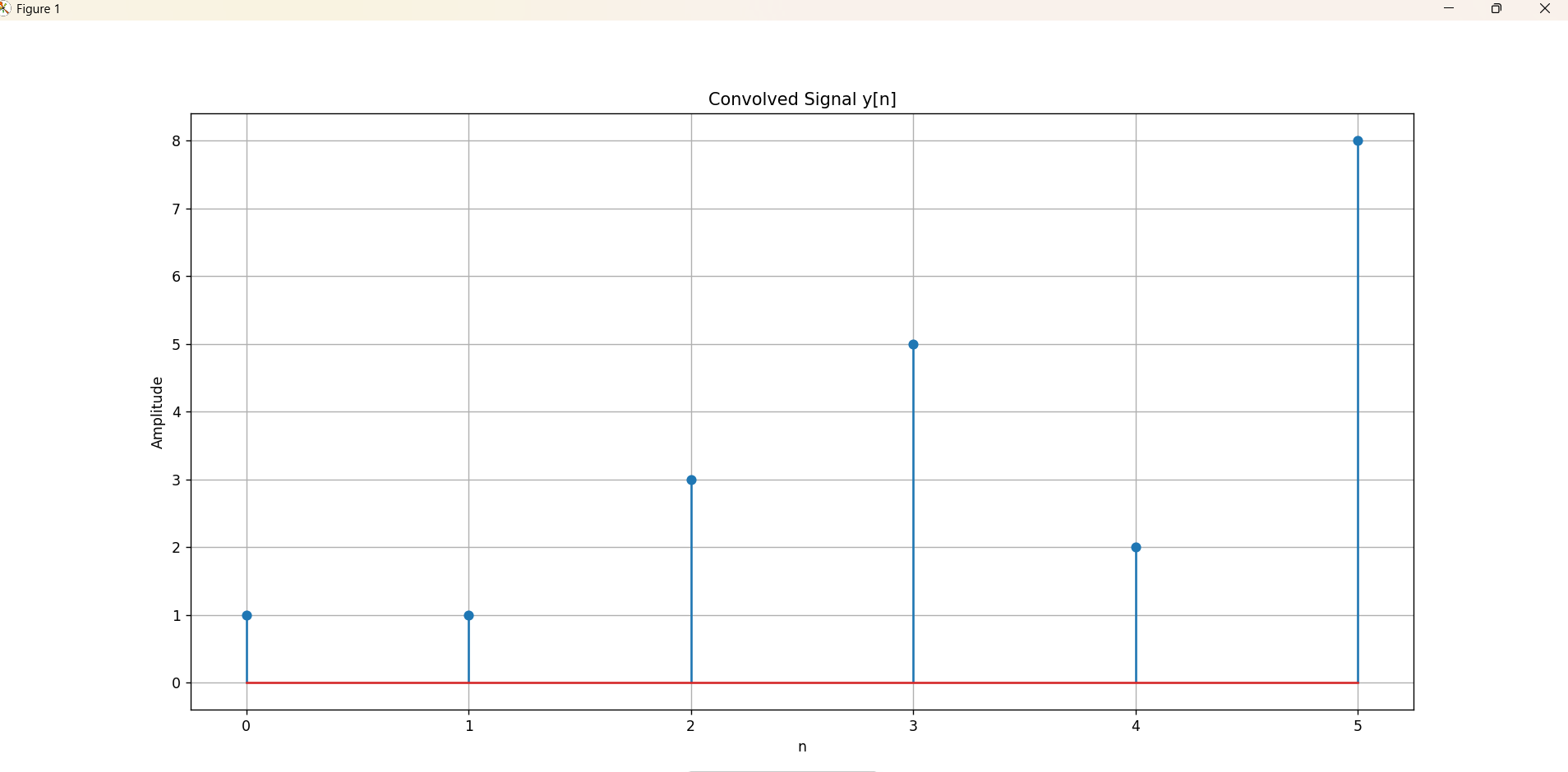
plot\_signal(np.arange(len(h)), h, 'Impulse Response h[n]')

plot\_signal(n, y, 'Convolved Signal y[n]')

**Output:**







**Lab: 04**

**Lab name:** Correlation (Cross and Auto) Using Python.

**Objectives:**

1. Understand Correlation: Correlation measures the statistical relationship between two variables.
2. Types of Correlation:
   * Cross-correlation: Measures the similarity between two signals as a function of time lag.
   * Auto-correlation: Measures the similarity of a signal with a delayed version of itself.
3. Understand How to Calculate Correlation Using Python.

**Theory:**

1. **Cross-correlation** is used when analyzing the relationship between two signals or time series. It measures how one time series matches with another when one is shifted by some time.

Formula for Cross-correlation:

Rxy​(τ)=t∑​x(t)⋅y(t+τ)

Where τ\tau τ is the time lag, x(t) and y(t) are the two signals or time series.

1. **Auto-correlation** measures the similarity of a signal with a delayed version of itself. It helps identify repeating patterns, periodic signals, or noise.

Formula for Auto-correlation:

Rxx​(τ)=t∑​x(t)⋅x(t+τ)

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

# Generating two sample signals for cross-correlation

np.random.seed(0)

t = np.linspace(0, 10, 100)

signal1 = np.sin(t) + np.random.normal(0, 0.1, 100)  # Signal 1: Sinusoidal signal with noise

signal2 = np.sin(t + 0.5) + np.random.normal(0, 0.1, 100)  # Signal 2: Sinusoidal signal with a phase shift and noise

# Cross-correlation using numpy.correlate

cross\_corr = np.correlate(signal1 - np.mean(signal1), signal2 - np.mean(signal2), mode='full')

lags = np.arange(-len(signal1) + 1, len(signal1))

# Plotting Cross-correlation

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

plt.subplot(1, 2, 1)

plt.plot(lags, cross\_corr)

plt.title("Cross-correlation")

plt.xlabel("Lags")

plt.ylabel("Correlation")

# Auto-correlation using numpy.correlate (Auto-correlation is just the cross-correlation of a signal with itself)

auto\_corr = np.correlate(signal1 - np.mean(signal1), signal1 - np.mean(signal1), mode='full')

# Plotting Auto-correlation

plt.subplot(1, 2, 2)

plt.plot(lags, auto\_corr)

plt.title("Auto-correlation")

plt.xlabel("Lags")

plt.ylabel("Correlation")

plt.tight\_layout()

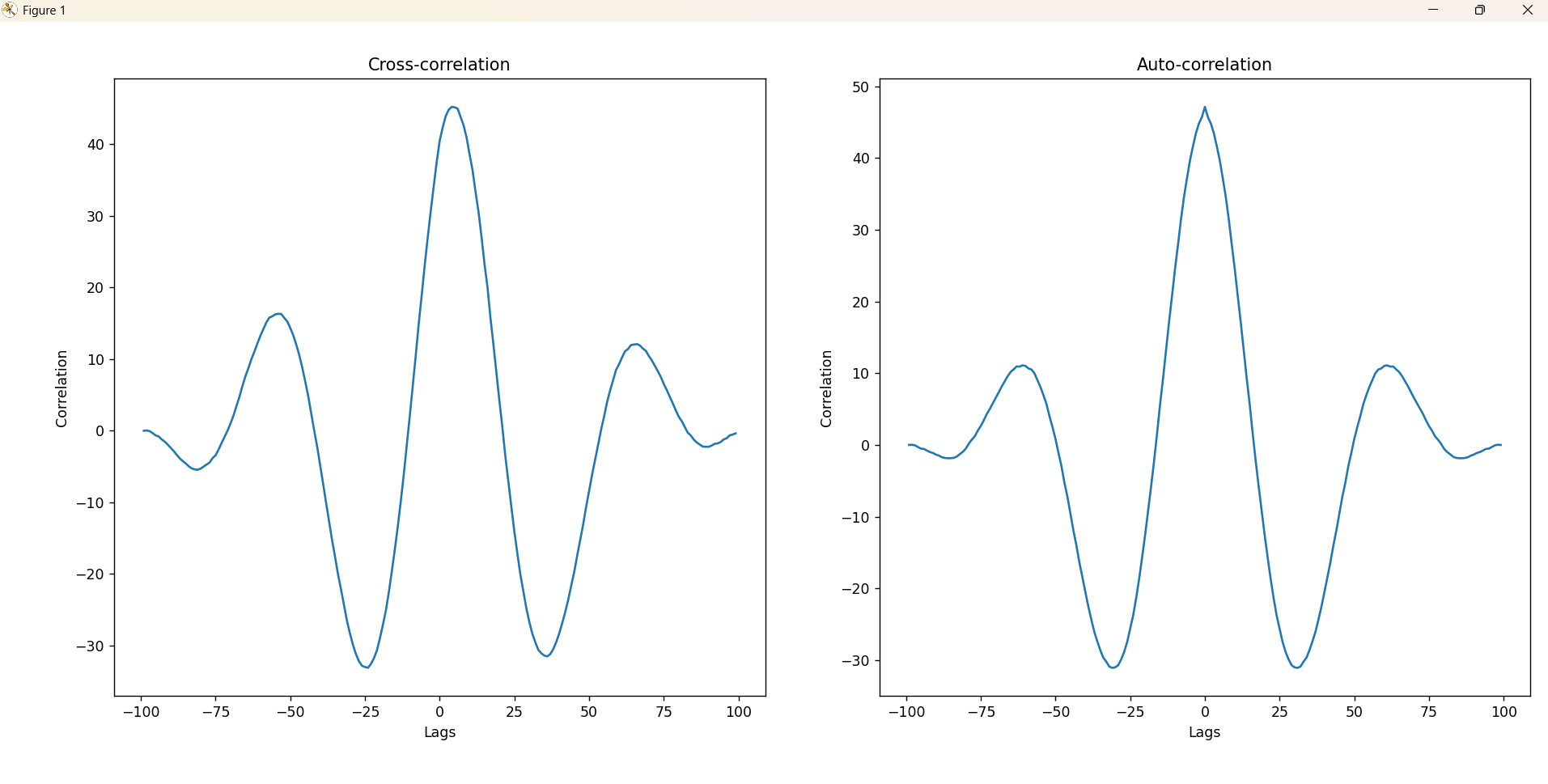
plt.show()

# If you are using pandas time series data:

# Assuming 'data' is a pandas DataFrame containing a time series 'col1' for auto-correlation:

# auto\_corr\_pandas = data['col1'].autocorr(lag=1)  # Auto-correlation with a lag of 1

**Output:**

****

**Lab: 05**

**Lab name:** PPG signal lab (a sine signal, noise signal ,signal with add noise ,filtered signal, filtering, feature extraction, peak detection).

**Objectives**

1. **Understand PPG Signals** – Learn how photoplethysmography (PPG) works and its applications in biomedical signal processing.
2. **Signal Generation** – Create a simulated sine wave to represent a clean PPG signal.
3. **Noise Addition** – Introduce noise to simulate real-world conditions.
4. **Filtering Techniques** – Apply filtering methods to remove noise.
5. **Feature Extraction** – Identify important characteristics such as peaks.
6. **Peak Detection** – Detect heartbeats from the filtered PPG signal.

**Theory:**

**1. PPG Signal Basics**

**A Photoplethysmogram (PPG) is an optical technique used to detect blood volume changes in the skin. It consists of:**

* **Systolic Peaks (higher values, indicating heartbeat)**
* **Diastolic Troughs (lower values between beats)**

**PPG signals are prone to noise from:**

* **Motion artifacts**
* **Power-line interference**
* **Respiration effects**

**2. Noise in Biomedical Signals**

* **Gaussian Noise (random fluctuations in the signal)**
* **Baseline Drift (slow variations in the signal)**
* **Powerline Interference (50/60 Hz electrical noise)**

**3. Filtering Techniques**

* **Low-pass filters remove high-frequency noise.**
* **High-pass filters remove baseline drift.**
* **Band-pass filters retain frequencies within a specific range.**

**4. Feature Extraction & Peak Detection**

* **Find systolic peaks using algorithms like find\_peaks() from scipy.signal.**
* **Compute heart rate (BPM) from peak intervals.**

**Source code:**

**import numpy as np**

**import matplotlib.pyplot as plt**

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

**# Generate a synthetic PPG signal (sine wave with peaks)**

**fs = 100  # Sampling frequency (Hz)**

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

**ppg\_clean = 0.6 \* np.sin(2 \* np.pi \* 1.2 \* t) + 1  # Simulated PPG (1.2 Hz ≈ 72 BPM)**

**# Add noise (Gaussian noise + baseline drift)**

**noise = 0.2 \* np.random.randn(len(t))  # Random noise**

**baseline\_drift = 0.5 \* np.sin(2 \* np.pi \* 0.1 \* t)  # Low-frequency drift**

**ppg\_noisy = ppg\_clean + noise + baseline\_drift  # Noisy PPG**

**# Bandpass Filter (0.5–5 Hz to remove drift & high-freq noise)**

**def bandpass\_filter(signal, lowcut=0.5, highcut=5, fs=100, order=4):**

**nyquist = 0.5 \* fs**

**low = lowcut / nyquist**

**high = highcut / nyquist**

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

**return filtfilt(b, a, signal)**

**ppg\_filtered = bandpass\_filter(ppg\_noisy)**

**# Peak Detection (detect systolic peaks)**

**peaks, \_ = find\_peaks(ppg\_filtered, distance=fs//2, height=0.5)**

**# Compute Heart Rate (BPM)**

**peak\_intervals = np.diff(t[peaks])  # Time differences between peaks**

**bpm = 60 / np.mean(peak\_intervals)  # Beats per minute**

**# Plot results**

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

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

**plt.plot(t, ppg\_clean, label="Clean PPG", color='green')**

**plt.title("Clean PPG Signal")**

**plt.legend()**

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

**plt.plot(t, ppg\_noisy, label="Noisy PPG", color='red')**

**plt.title("Noisy PPG Signal with Noise & Baseline Drift")**

**plt.legend()**

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

**plt.plot(t, ppg\_filtered, label="Filtered PPG", color='blue')**

**plt.plot(t[peaks], ppg\_filtered[peaks], "ro", label="Detected Peaks")**

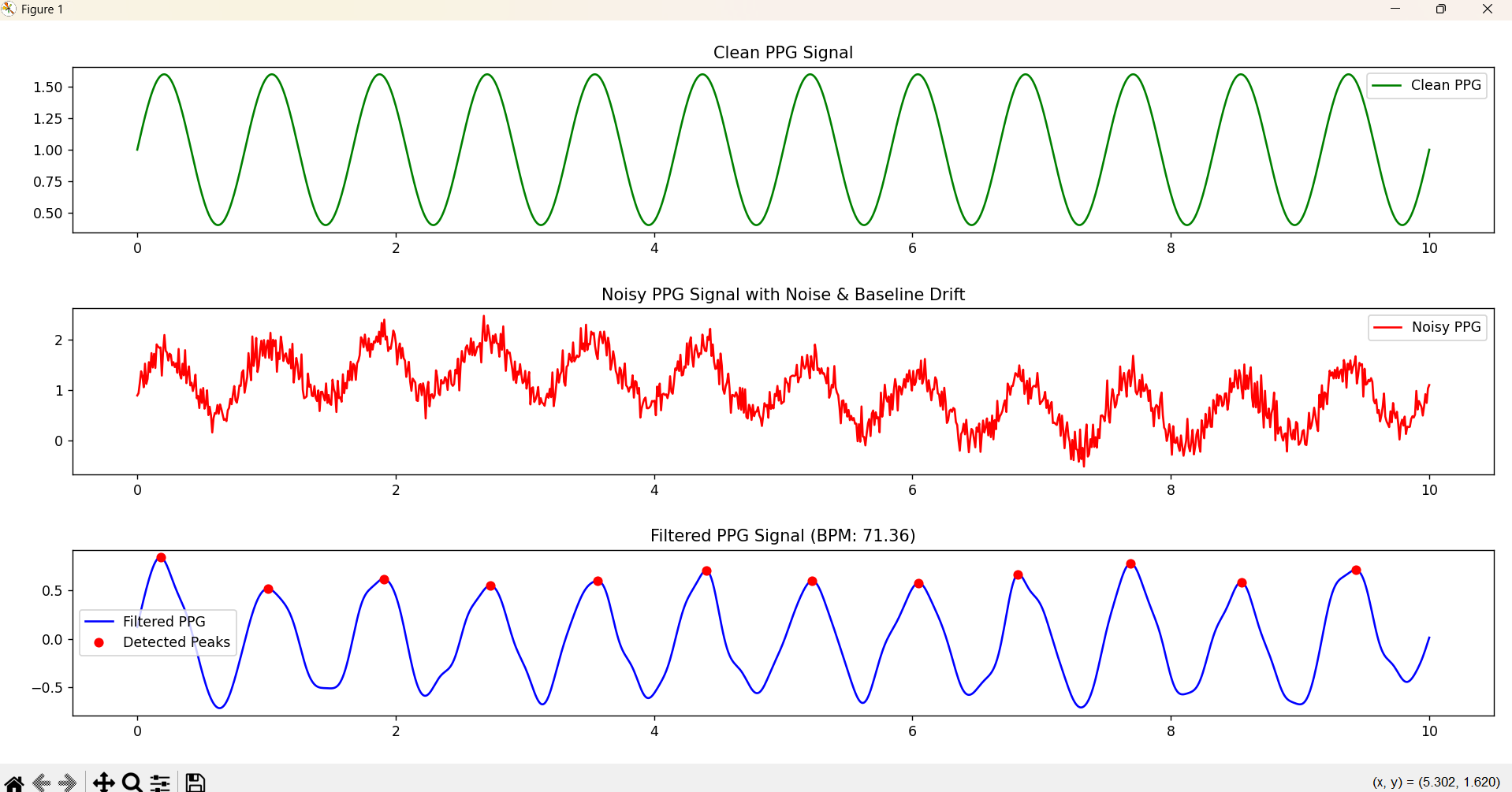
**plt.title(f"Filtered PPG Signal (BPM: {bpm:.2f})")**

**plt.legend()**

**plt.tight\_layout()**

**plt.show()**

**Output:**

****

**Lab: 06**

**Lab name:** Fourier Transform using python.

**Objectives:**

1. Understand the Fourier Transform and its importance in signal processing.
2. Learn how to compute the Fourier Transform using Python.
3. Visualize signals in both time and frequency domains.
4. Implement the Fast Fourier Transform (FFT) using numpy.fft.

**Theory**

The Fourier Transform (FT) is a mathematical technique that transforms a time-domain signal into its frequency components. It is widely used in signal processing, image processing, and physics.

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

**X(f)=**

For discrete signals, we use the **Discrete Fourier Transform (DFT)**, which is given by:

**where:**

* X[k] represents the frequency components.
* x[n] is the discrete input signal.
* N is the total number of samples.
* k is the index for frequency bins.

**Source code:**

import numpy as np

import matplotlib.pyplot as plt

# Generate a sample signal: Sum of two sinusoids

Fs = 500  # Sampling frequency (Hz)

T = 1.0 / Fs  # Sampling interval

N = 1024  # Number of sample points

t = np.linspace(0, N\*T, N, endpoint=False)

# Create a signal with two frequencies (50 Hz and 120 Hz)

f1, f2 = 50, 120

signal = np.sin(2\*np.pi\*f1\*t) + 0.5\*np.sin(2\*np.pi\*f2\*t)

# Compute the Fast Fourier Transform (FFT)

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

frequencies = np.fft.fftfreq(N, T)

# Take magnitude and only positive frequencies

magnitude = np.abs(fft\_output[:N//2])

frequencies = frequencies[:N//2]

# Plot the signal in time domain

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)

plt.plot(t[:500], signal[:500])  # Plot only first 500 points for clarity

plt.title("Time Domain Signal")

plt.xlabel("Time (s)")

plt.ylabel("Amplitude")

# Plot the magnitude spectrum

plt.subplot(1, 2, 2)

plt.plot(frequencies, magnitude)

plt.title("Frequency Domain (FFT)")

plt.xlabel("Frequency (Hz)")

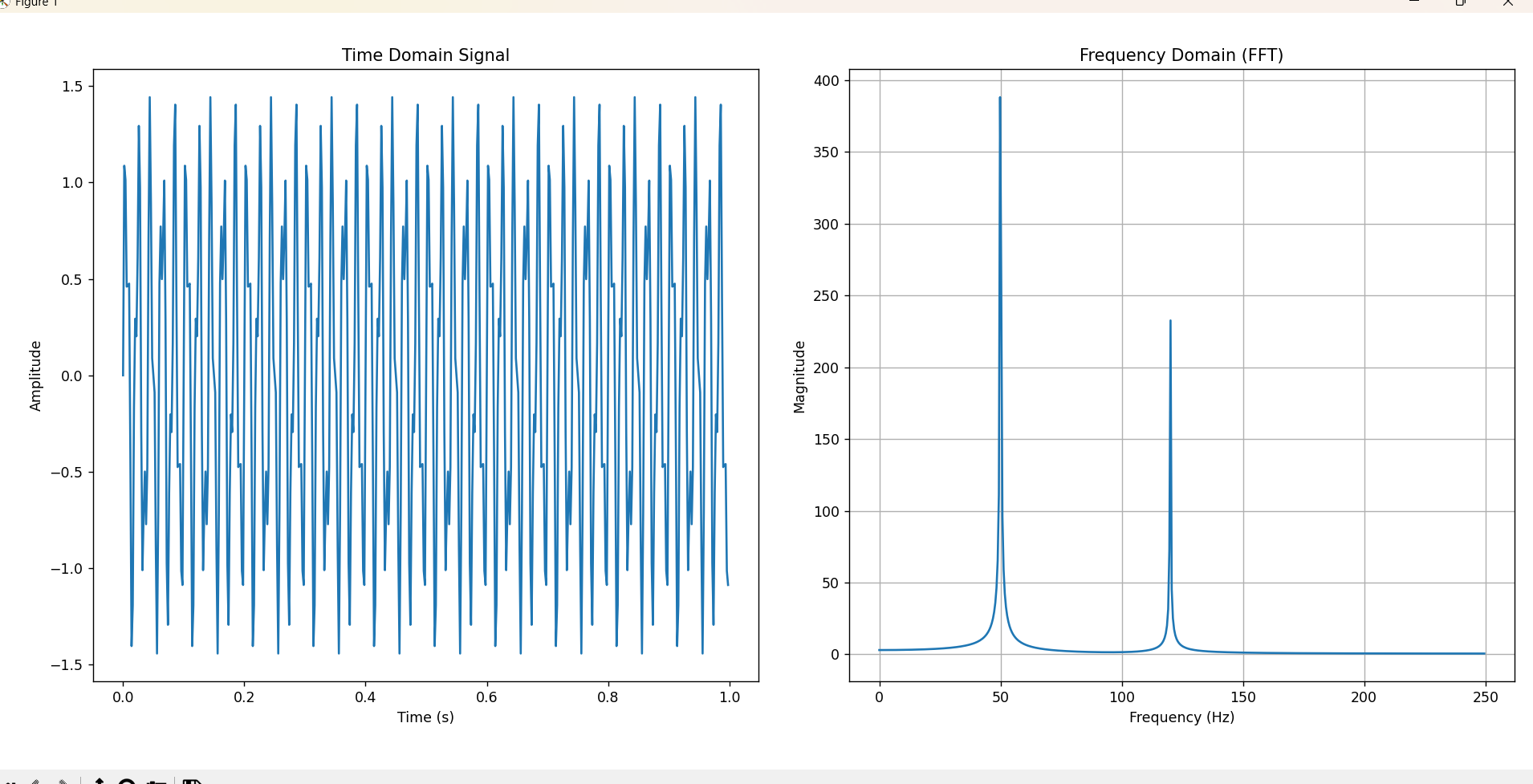
plt.ylabel("Magnitude")

plt.grid()

plt.tight\_layout()

plt.show()

**Output:**

****

**Lab: 07**

**Lab name:** Fourier Series Decomposition with Harmonics using python.

**Objectives:**

1. Understand the Fourier Series representation of a periodic function.
2. Compute Fourier coefficients for a given function.
3. Reconstruct the function using a finite number of harmonics.
4. Visualize the approximation of the function using Python.

**Theory:**

The Fourier Series is a mathematical tool used to break down a periodic function into a sum of simpler trigonometric functions—sines and cosines. This decomposition helps analyze and reconstruct complex waveforms using fundamental frequency components, known as harmonics.

A periodic function can be thought of as a combination of a constant term, a set of sine waves, and a set of cosine waves, each with different amplitudes and frequencies. The fundamental frequency, also called the first harmonic, is the lowest frequency present in the signal. Higher-order harmonics are integer multiples of this fundamental frequency and contribute to the overall shape of the function.

**Source code:**

import numpy as np

import matplotlib.pyplot as plt

from scipy.integrate import quad

# Define the periodic function (Example: Square wave)

def f(x):

    return 1 if (x % (2\*np.pi)) < np.pi else -1  # Square wave function

# Define Fourier Series Coefficients

T = 2 \* np.pi  # Period

omega0 = 2 \* np.pi / T  # Fundamental frequency

# Compute a0

a0 = (1 / T) \* quad(lambda x: f(x), 0, T)[0]

# Compute an and bn

def compute\_coefficients(n):

    an = (2 / T) \* quad(lambda x: f(x) \* np.cos(n \* omega0 \* x), 0, T)[0]

    bn = (2 / T) \* quad(lambda x: f(x) \* np.sin(n \* omega0 \* x), 0, T)[0]

    return an, bn

# Number of harmonics

N = 10

coefficients = [compute\_coefficients(n) for n in range(1, N+1)]

# Reconstruct the function using Fourier series

x\_vals = np.linspace(-2\*np.pi, 2\*np.pi, 400)

y\_vals = np.full\_like(x\_vals, a0 / 2)

for n in range(1, N+1):

    an, bn = coefficients[n-1]

    y\_vals += an \* np.cos(n \* omega0 \* x\_vals) + bn \* np.sin(n \* omega0 \* x\_vals)

# Plot results

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

plt.plot(x\_vals, [f(x) for x in x\_vals], label="Original Function", linestyle="dotted")

plt.plot(x\_vals, y\_vals, label=f"Fourier Series Approximation (N={N})", color="red")

plt.legend()

plt.xlabel("x")

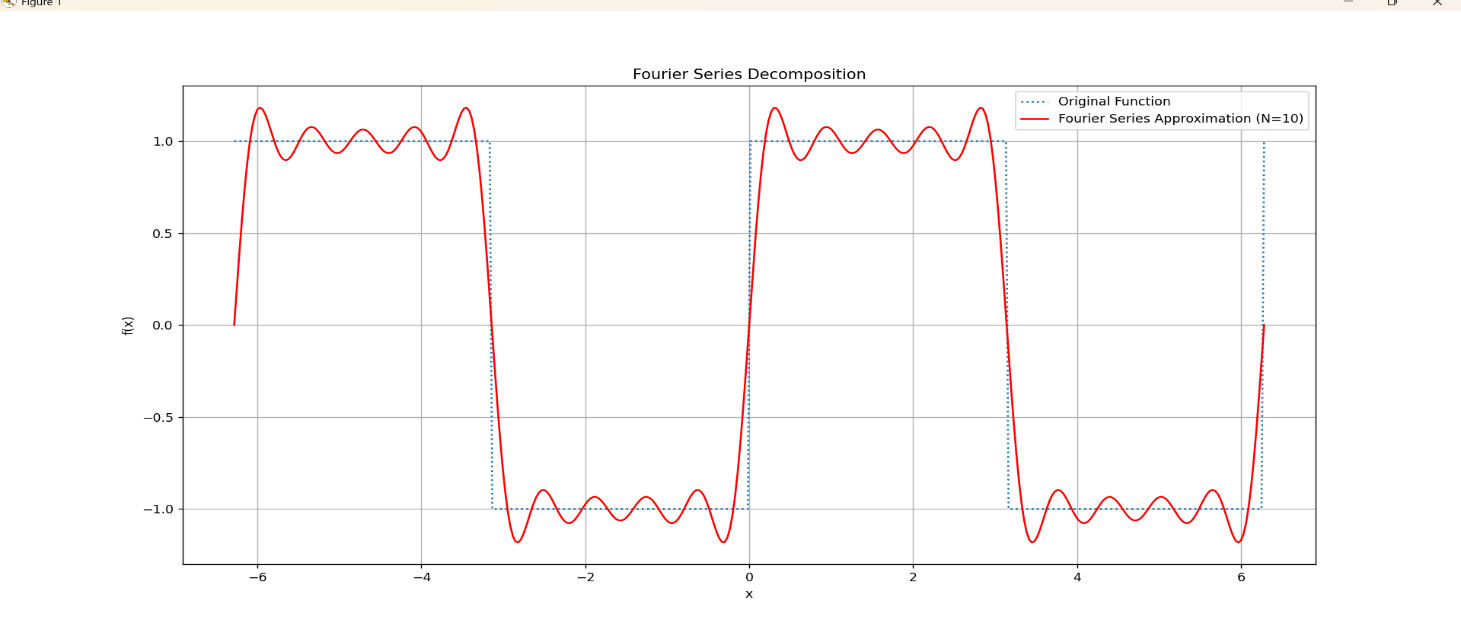
plt.ylabel("f(x)")

plt.title("Fourier Series Decomposition")

plt.grid()

plt.show()

**Output:**

****

**Lab: 08**

**Lab name:** A lab report on DFT using python.

**Objective** The objective of this lab is to implement the Discrete Fourier Transform (DFT) using Python and analyze its application in signal processing. We aim to:

* Understand the mathematical formulation of DFT.
* Implement DFT using Python without using built-in FFT functions.
* Compare the results with NumPy’s Fast Fourier Transform (FFT) implementation.
* Visualize the frequency components of a given signal.

**Theory**:

The **Discrete Fourier Transform (DFT)** is a mathematical technique used in signal processing to analyze the frequency components of a discrete signal. It converts a sequence of values from the time domain into the frequency domain, helping to understand periodic patterns within data.

**Source code:**

import numpy as np

import matplotlib.pyplot as plt

# Define the DFT function

def dft(x):

    N = len(x)

    X = np.zeros(N, dtype=complex)  # Create an array to store the DFT results

    # Perform the DFT computation

    for k in range(N):

        for n in range(N):

            X[k] += x[n] \* np.exp(-2j \* np.pi \* k \* n / N)

    return X

# Define the IDFT function (Inverse Discrete Fourier Transform)

def idft(X):

    N = len(X)

    x = np.zeros(N, dtype=complex)  # Create an array to store the IDFT results

    # Perform the IDFT computation

    for n in range(N):

        for k in range(N):

            x[n] += X[k] \* np.exp(2j \* np.pi \* k \* n / N) / N

    return x

# Example usage

# Generate a sample signal (e.g., a sine wave)

fs = 1000  # Sampling frequency (samples per second)

T = 1 / fs  # Sampling period

t = np.arange(0, 1, T)  # Time vector

f = 50  # Frequency of the signal in Hz

x = np.sin(2 \* np.pi \* f \* t)  # Signal (sine wave)

# Compute the DFT of the signal

X = dft(x)

# Compute the magnitude and phase of the DFT

X\_magnitude = np.abs(X)

X\_phase = np.angle(X)

# Plot the original signal

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

plt.subplot(2, 1, 1)

plt.plot(t, x)

plt.title('Original Signal (Sine Wave)')

plt.xlabel('Time [s]')

plt.ylabel('Amplitude')

# Plot the magnitude of the DFT

frequencies = np.fft.fftfreq(len(x), T)  # Frequency vector

plt.subplot(2, 1, 2)

plt.plot(frequencies[:len(frequencies)//2], X\_magnitude[:len(X\_magnitude)//2])  # Only plot positive frequencies

plt.title('Magnitude of DFT')

plt.xlabel('Frequency [Hz]')

plt.ylabel('Magnitude')

plt.tight\_layout()

plt.show()

# Inverse DFT (to recover the signal)

x\_reconstructed = idft(X)

# Plot the reconstructed signal to verify the IDFT

plt.figure(figsize=(6, 4))

plt.plot(t, x\_reconstructed.real)  # Take real part for the reconstructed signal

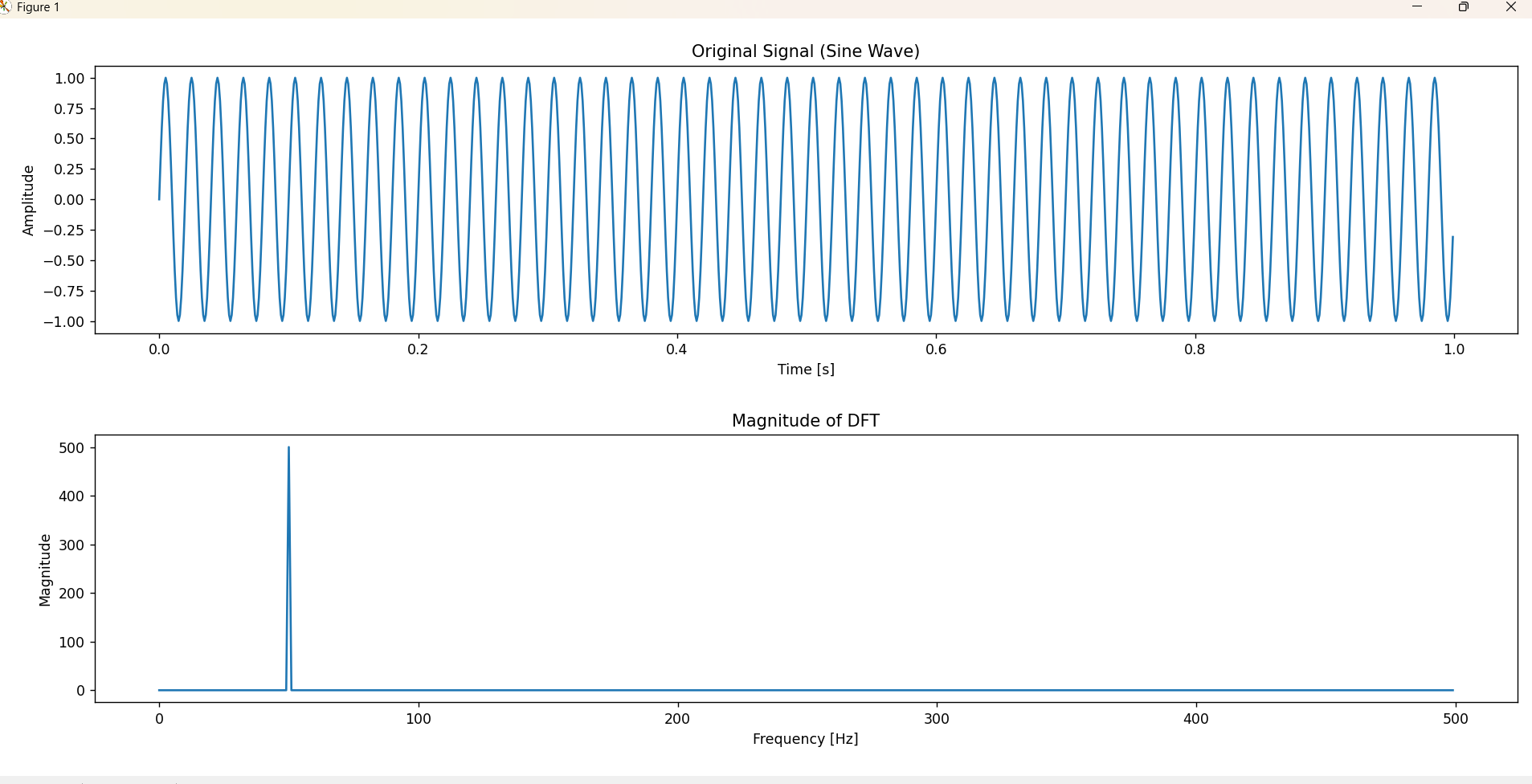
plt.title('Reconstructed Signal (from IDFT)')

plt.xlabel('Time [s]')

plt.ylabel('Amplitude')

plt.show()

**Output:**

****