**LAB REPORT**



Department of

Information and Communication Engineering

ICE - 2204

Signals and Systems Sessional

Submitted By Azmira Akter Simla Roll : 220626

Dept. of Information and Communication Engineering

Submitted To

Dr. Md. Imran Hossain Associate Professor Dept. of Information and

Communication Engineering

Date of Submission : 03-03-25

**INDEX**

|  |  |
| --- | --- |
| **SI.** | **Problem Statement** |
| 01 | Signal Operations Addition, Multiplication, Scaling, Shifting, and Folding. |
| 02 | Analysis of Signal Convolution with Different Signal Types. |
| 03 | Autocorrelation and Cross-Correlation of Signals. |
| 04 | Generation and Visualization of Impulse, Step, and Ramp Signals in Python |
| 05 | Extraction and Analysis of Features from PPG Signal for Pulse and Frequency DomainCharacteristics. |
| 06 | Fourier Transform and Frequency Spectrum Analysis of a Signal. |
| 07 | Implementation of Discrete Fourier Transform (DFT) and Noise Filtering Using Python |

**Problem Number:** 01

**Title:** Signal Operations Addition, Multiplication, Scaling, Shifting, and Folding.

**Theory:**

Signals are mathematical representations of physical quantities that vary over time. Basic signal operations include:

* Addition: Combining two signals by adding their corresponding values. This operation is used in applications such as noise reduction and mixing multiple signals.

Formula: Given two signals x1[n] and x2[n], their addition is defined as:

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

* Multiplication: Producing a new signal by multiplying corresponding values of two signals. This is used in modulation and filtering applications.

Formula: Given two signals x1[n] and x2[n], their multiplication is defined

as:

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

* Scaling: Modifying a signal's amplitude by multiplying it with a constant factor. This operation is useful in amplification and attenuation of signals.

Formula: *y[n]* = *x[n]*

Where is a scaling factor.

* Shifting: Moving a signal forward or backward in time. This is used in time-delay systems and synchronization.

Formula:  *y[n] = x [n - k]*

Where *k* is the shift amount. A positive shifts the signal right, and a

negative shifts it left.

* Folding: Reversing a signal about the vertical axis. This operation is useful in signal symmetry analysis and convolution.

Formula: *y[n] = x [ - n]*

**Source Code (in Python):**

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()

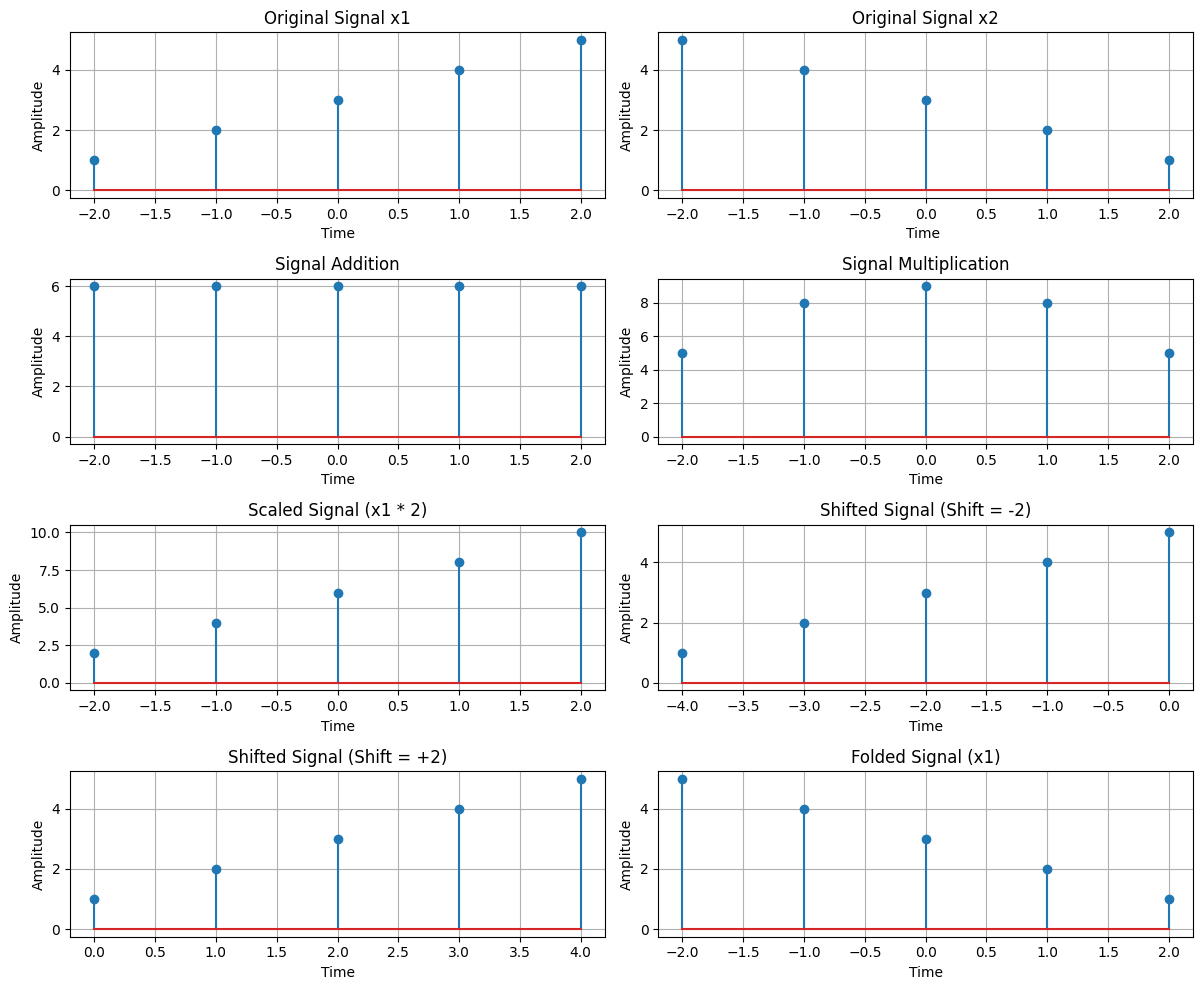
plt.show()

**Input:**

Input Signals:

* Discrete-time signals:
  + *x1* = [1,2,3,4,5]
  + *x2* = [5,4,3,2,1]
  + Time indices: *n* = [-2, -1, 0, 1, 2]

**Output:**



**Conclusion:**

operations are essential for signal analysis and manipulation in digital signal processing. The results show how different mathematical transformations affect signals, which is crucial for applications in audio processing, image processing, and communications.

**Problem Number:** 02

**Title:** Analysis of Signal Convolution with Different Signal Types.

**Theory:**  
Convolution is a mathematical operation that combines two signals to produce a third signal. It can be used in signal processing for various applications like filtering, system analysis, and signal comparison. The convolution operation between two signals is defined as the integral (or sum in discrete-time signals) of the product of the two signals after one is reversed and shifted. Convolution reveals how one signal affects another in terms of their interaction over time.

In this experiment, we analyze three types of convolution:

1. Auto convolution of a sinusoidal signal, where a signal is convolved with itself.
2. Convolution between a signal and a shifted version of itself.
3. Convolution with a noisy version of the signal.

**Source Code (in Python):**

import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import convolve

# Fu nction to compute the convolution

def compute\_convolution(signal1, signal2):

conv\_result = convolve(signal1, signal2, mode='full', method='auto')

return conv\_result

# Parameters

fs = 1000 # Sampling frequency in Hz

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

freq = 5 # Frequency of the sine wave

# Generate a sine wave signal

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

# Compute the auto-convolution of the signal

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

# Compute the convolution between the signal and a shifted version of it

signal1 = sin\_signal

signal2 = np.roll(signal1, 100)

conv\_shifted = compute\_convolution(signal1, signal2)

# Add noise to the signal and compute the convolution

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

noisy\_signal = signal1 + noise

conv\_noisy = compute\_convolution(signal1, noisy\_signal)

# Plot the results

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

# Autoconvolution of the sinusoidal signal

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()

# Convolution between the signal and its shifted version

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()

# Convolution with the noisy signal

plt.subplot(3, 1, 3)

plt.plot(conv\_noisy)

plt.title("Convolution with Noisy Signal")

plt.xlabel("Samples")

plt.ylabel("Convolution Output")

plt.grid()

# Adjust layout for better visualization

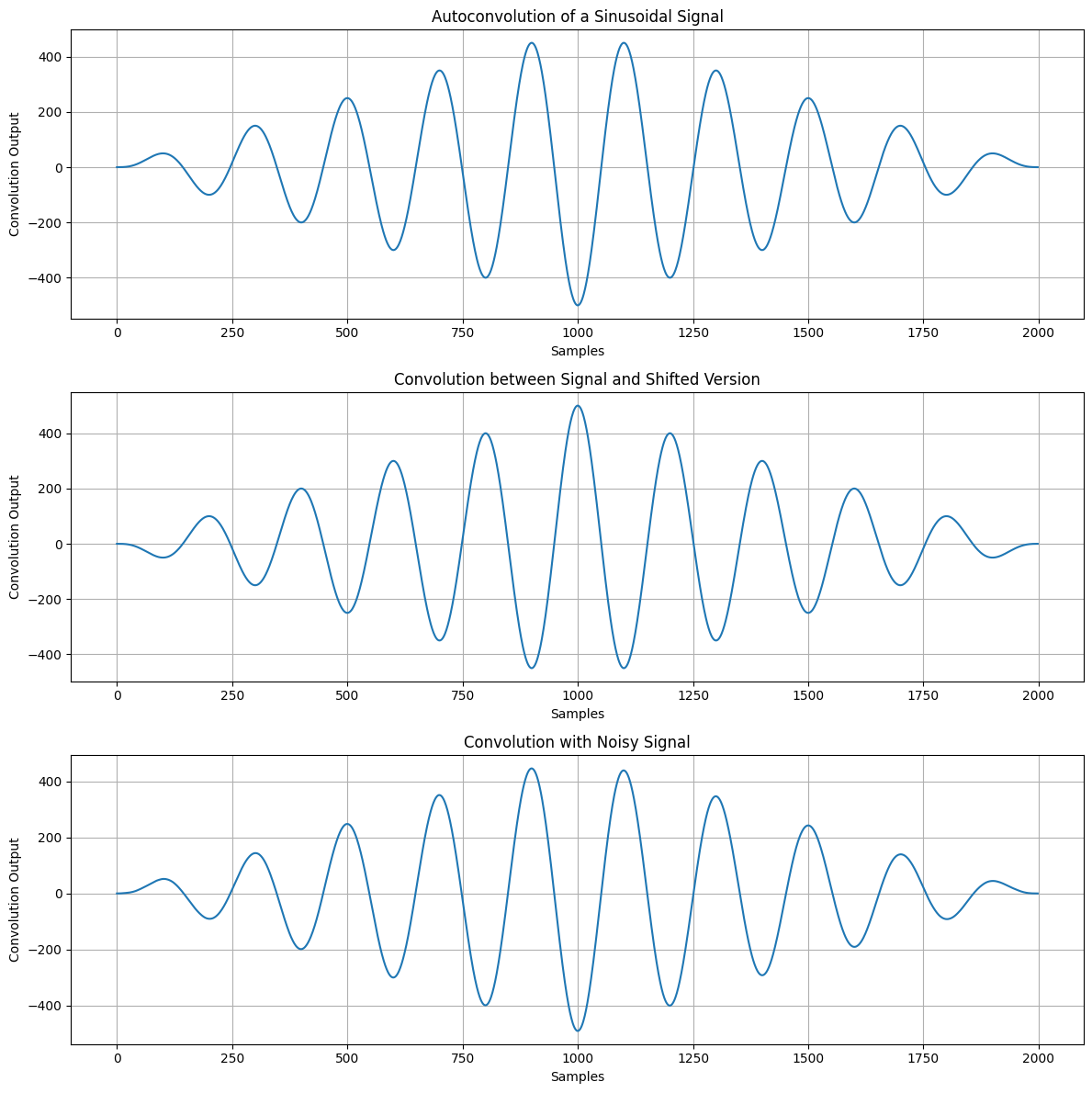
plt.tight\_layout()

plt.show()

**Input:**

* Signal 1: A sinusoidal signal with a frequency of 5 Hz, sampled at 1000 Hz over a time period of 1 second.
* Signal 2: A shifted version of Signal 1.
* Noisy Signal**:** A noisy version of Signal 1, generated by adding random noise.

**Output:**



**Conclusion:**

The purpose of this lab was to demonstrate how convolution interacts with different types of signals:

1. Auto convolution: When convolving the sinusoidal signal with itself, the result shows how the signal correlates with itself over time, highlighting its periodic nature.
2. Convolution with Shifted Signal: Convolution with a shifted signal illustrates how the signal’s shape changes when compared to a delayed version of itself. The shifting effect is evident in the convolution result.
3. Convolution with Noisy Signal: Adding noise to the signal and then convolving it demonstrates how noise can alter the signal’s characteristics. This is useful for understanding how noise affects the behavior of real-world signals.

**Problem Number:** 03

**Title:** Autocorrelation and Cross-Correlation of Signals

**Theory:**

Correlation is a fundamental operation in signal processing used to measure the similarity between signals. The two main types of correlation are:

* Autocorrelation: Measures how a signal correlates with a delayed version of itself. It helps in analyzing periodicity and detecting patterns in signals.

Formula:

where x(n) is the signal, and ‘k’ is the lag.

* Cross-Correlation: Measures the similarity between two different signals as a function of time shift. It is widely used in time-delay estimation and pattern recognition.

Formula:

where x(n) and y(n) are two different signals, and ‘k’ is the lag.

**Source Code (in Python):**

import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import correlate, correlation\_lags

def compute\_autocorrelation(signal):

auto\_corr = correlate(signal, signal, mode='full', method='auto')

lags = correlation\_lags(len(signal), len(signal), mode='full')

return auto\_corr, lags

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

cross\_corr = correlate(signal1, signal2, mode='full', method='auto')

lags = correlation\_lags(len(signal1), len(signal2), mode='full')

return cross\_corr, lags

fs = 1000 # Sampling frequency in Hz

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

freq = 5 # Frequency of the sine wave

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

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

signal1 = sin\_signal

signal2 = np.roll(signal1, 100)

cross\_corr, lags\_cross = compute\_cross\_correlation(signal1, signal2)

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

noisy\_signal = signal1 + noise

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

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

plt.subplot(3, 1, 1)

plt.plot(lags\_auto, auto\_corr)

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

plt.xlabel("Lag")

plt.ylabel("Autocorrelation")

plt.grid()

plt.subplot(3, 1, 2)

plt.plot(lags\_cross, cross\_corr)

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

plt.xlabel("Lag")

plt.ylabel("Cross-Correlation")

plt.grid()

plt.subplot(3, 1, 3)

plt.plot(lags\_noise, cross\_corr\_noise)

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

plt.xlabel("Lag")

plt.ylabel("Cross-Correlation")

plt.grid()

plt.tight\_layout()

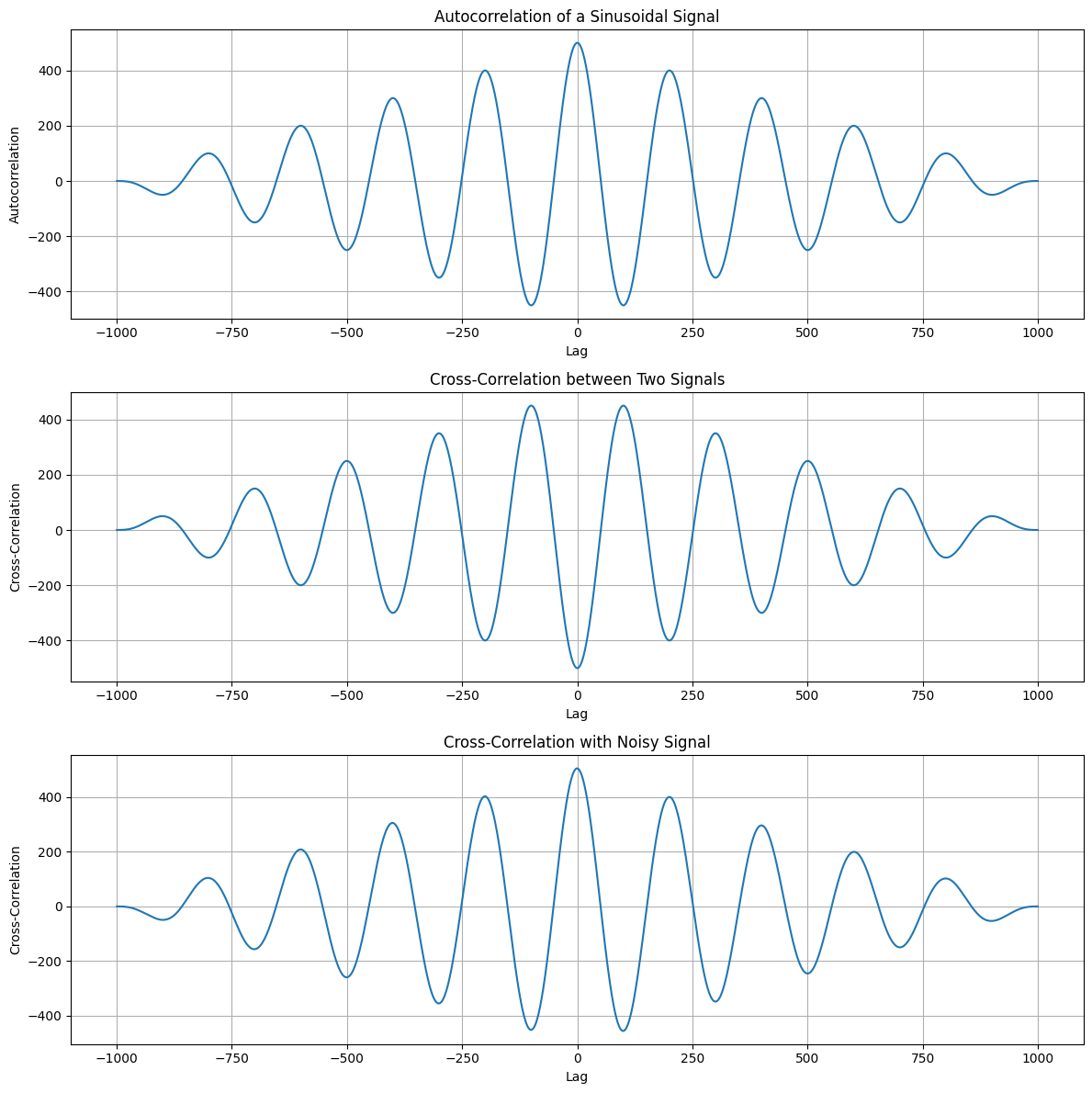
plt.show()

**Input:**

Input Signals:

* A sinusoidal signal with frequency 5 Hz, sampled at 1000 Hz.
* A time-shifted version of the sinusoidal signal (shift = 100 samples).
* A noisy version of the sinusoidal signal (Gaussian noise added).

**Output:**



**Conclusion:** This experiment demonstrates how autocorrelation and cross-correlation help analyze signals. Autocorrelation is useful for detecting repeating patterns and periodicity, while cross-correlation is crucial in measuring similarity between signals, even in noisy conditions. These techniques are widely used in signal processing applications like speech analysis, wireless communications, and

radar systems.

**Problem Number:** 04

**Title:**

Generation and Visualization of Impulse, Step, and Ramp Signals in Python

**Theory:**

Discrete-time signals are sequences of values defined at specific time intervals. These signals are fundamental in digital signal processing (DSP) and are used to analyze and design systems. In this lab, we focus on three basic discrete-time signals:

1. Impulse Signal (δ[n]):
   * The impulse signal is defined as:

​

* + It is used to represent a sudden spike or disturbance in a system.

1. Step Signal (u[n]):
   * The step signal is defined as:

*u*​

* + It represents a signal that "turns on" at n=0 and remains on indefinitely.

1. Ramp Signal (r[n]):
   * The ramp signal is defined as:

*r*​

* + It represents a signal that increases linearly with time.

**Source Code (in python):**

# Import required libraries

import numpy as np

import matplotlib.pyplot as plt

# Define the range

n = np.arange(-10, 11)

# Define signal functions

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))

# Impulse Signal

plt.subplot(1, 3, 1)

plt.stem(n, impulse)

plt.title("Impulse Signal")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

# Step Signal

plt.subplot(1, 3, 2)

plt.stem(n, step)

plt.title("Step Signal")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

# Ramp Signal

plt.subplot(1, 3, 3)

plt.stem(n, ramp)

plt.title("Ramp Signal")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

# Adjust layout and display the plot

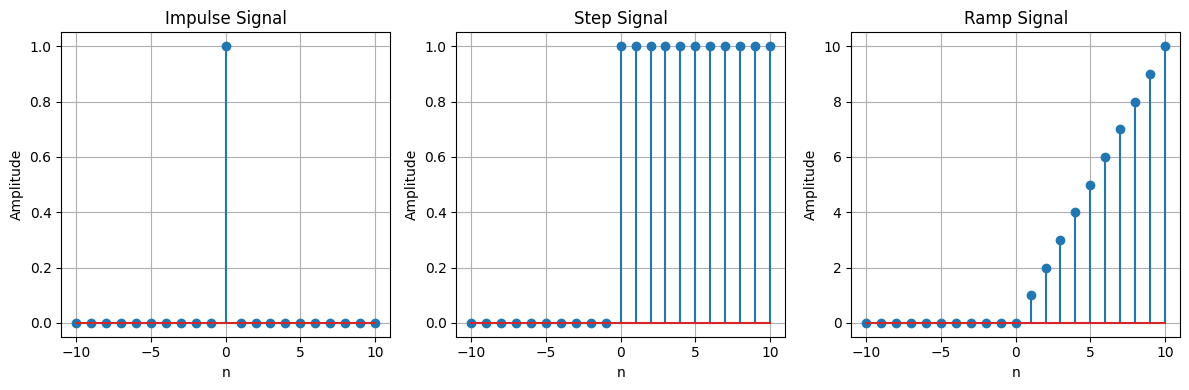
plt.tight\_layout()

plt.show()

**Input:**

* A range of discrete time values n*n* from -10 to 10.
* Functions to generate:
  1. Impulse signal (δ[n]*δ*[*n*]).
  2. Step signal (u[n]*u*[*n*]).
  3. Ramp signal (r[n]*r*[*n*]).

**Output:**



**Conclusion:**

The purpose of this lab was to:

1. Generate and visualize three basic discrete-time signals: impulse, step, and ramp.
2. Understand the mathematical definitions and properties of these signals.
3. Use Python and Matplotlib to create and display the signals.

**Problem Number:**05

**Title:** Extraction and Analysis of Features from PPG Signal for Pulse and Frequency DomainCharacteristics.

**Theory:**

Photoplethysmography (PPG) is an optical technique used to measure variations in blood volume as the heart pumps blood through the body. The PPG signal reflects these changes in light absorption by the skin, providing valuable information about heart rate and circulatory health. The signal consists of a periodic component that correlates with the heartbeat and may include noise due to factors such as motion or environmental interference.

In this lab, we focus on extracting key features from the PPG signal that can be used to analyze heart rate and cardiovascular characteristics:

1. **Lowpass Filtering:**  
   A lowpass filter is used to remove high-frequency noise from the PPG signal while preserving the lower frequency components associated with the heart rate. By attenuating frequencies above a certain threshold, the filter ensures that the signal used for further analysis reflects the physiological rhythms related to blood circulation.
2. **Peak Detection:**  
   The filtered PPG signal is analyzed to detect peaks, which correspond to individual heartbeats. The positions of these peaks provide the timing information needed to calculate important metrics such as pulse amplitude, pulse duration, and the intervals between successive heartbeats.
3. **Pulse Amplitude:**  
   Pulse amplitude refers to the difference in the signal’s value between a peak and its baseline. This feature provides insights into the intensity of the blood flow during each heartbeat, which is influenced by factors such as blood volume and vascular resistance.
4. **Pulse Duration:**  
   Pulse duration measures the time between consecutive peaks in the PPG signal. This feature reflects the speed and efficiency of the cardiovascular system’s response to each heartbeat and can provide information about heart rhythm and cardiac function.
5. **Inter-Beat Interval (IBI):**  
   The inter-beat interval (IBI) is the time interval between two consecutive heartbeats, providing direct information about heart rate. The variability in the IBI is often used as a measure of heart rate variability (HRV), which is associated with autonomic nervous system activity and overall cardiovascular health.
6. **Signal Entropy:**  
   Entropy is a measure of the unpredictability or randomness in the signal. Higher entropy values indicate more irregular or erratic signals, while lower entropy suggests a more predictable pattern. This measure is useful for assessing the regularity of heart rate and detecting abnormal rhythms.
7. **Root Mean Square (RMS):**  
   The RMS value quantifies the overall energy in the signal. It is calculated as the square root of the mean of the squared signal values. This feature is useful for assessing the magnitude of the PPG signal, which can be influenced by both the amplitude of heartbeats and the presence of noise.
8. **Frequency Domain Analysis:**  
   In addition to time-domain features, the PPG signal is analyzed in the frequency domain to identify the frequency components of the signal. The power spectral density (PSD) provides information about how the signal's energy is distributed across different frequencies. By examining the dominant frequency, we can estimate the heart rate and assess the overall power of the signal, which is related to the strength and quality of the PPG signal.

**Python Code:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

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

from scipy.stats import entropy

from scipy.signal import welch

# Dataset

file\_path = 'emd\_1\_imfs.csv'

ppg\_data = pd.read\_csv(file\_path)

ppg\_signal = ppg\_data['Imf\_1\_MEAN'].values

# Lowpass filter design

def butter\_lowpass(cutoff, fs, order=4):

nyquist = 0.5 \* fs

normal\_cutoff = cutoff / nyquist

b, a = butter(order, normal\_cutoff, btype='low', analog=False)

return b, a

def butter\_lowpass\_filter(data, cutoff, fs, order=4):

b, a = butter\_lowpass(cutoff, fs, order)

y = filtfilt(b, a, data)

return y

# Filter PPG signal

fs = 100 # Sampling frequency (Hz)

cutoff = 3 # Cutoff frequency (Hz)

filtered\_ppg = butter\_lowpass\_filter(ppg\_signal, cutoff, fs)

# Feature Extraction

# Peak detection using scipy find\_peaks

peaks, \_ = find\_peaks(filtered\_ppg, distance=fs\*0.6)

# Pulse Amplitude: Peak-to-baseline difference

pulse\_amplitude = filtered\_ppg[peaks] - np.min(filtered\_ppg)

# Pulse Duration: Time between peaks

pulse\_duration = np.diff(peaks) / fs

# Inter-Beat Interval (IBI): Time between consecutive peaks

ibi = np.diff(peaks) / fs

# Signal Variability (IBI Standard Deviation)

ibi\_variability = np.std(ibi)

# Peak-to-Peak Interval (PPI)

ppi = np.diff(peaks) / fs

# Entropy of the Signal (measuring unpredictability)

ppg\_entropy = entropy(filtered\_ppg)

# Root Mean Square (RMS) of the signal

rms\_value = np.sqrt(np.mean(filtered\_ppg\*\*2))

# Frequency Domain Analysis

f, psd = welch(filtered\_ppg, fs, nperseg=1024)

dominant\_frequency = f[np.argmax(psd)]

# Total Power in Signal

total\_power = np.sum(psd)

# Visualize Extracted Features

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

# Plot filtered signal with peaks

plt.subplot(3, 2, 1)

plt.plot(filtered\_ppg, label='Filtered PPG Signal')

plt.plot(peaks, filtered\_ppg[peaks], 'ro', label='Detected Peaks')

plt.title("Filtered PPG Signal with Peaks")

plt.legend()

# Pulse Amplitude

plt.subplot(3, 2, 2)

plt.plot(peaks, pulse\_amplitude, 'bo', label='Pulse Amplitude')

plt.title("Pulse Amplitude")

plt.legend()

# Pulse Duration

plt.subplot(3, 2, 3)

plt.plot(np.arange(len(pulse\_duration)), pulse\_duration, label="Pulse Duration", marker='o')

plt.title("Pulse Duration Between Peaks")

plt.legend()

# Inter-Beat Interval (IBI)

plt.subplot(3, 2, 4)

plt.plot(np.arange(len(ibi)), ibi, label="Inter-Beat Interval (IBI)", marker='o')

plt.title("Inter-Beat Interval (IBI)")

plt.legend()

# RMS of the signal

plt.subplot(3, 2, 5)

plt.axhline(rms\_value, color='g', linestyle='--', label="RMS Value")

plt.title("Root Mean Square (RMS) of the PPG Signal")

plt.legend()

# Frequency Domain (Power Spectral Density)

plt.subplot(3, 2, 6)

plt.semilogy(f, psd, label="Power Spectral Density")

plt.axvline(dominant\_frequency, color='r', linestyle='--', label=f"Dominant Frequency = {dominant\_frequency:.2f} Hz")

plt.title("Power Spectral Density (Frequency Domain)")

plt.legend()

plt.tight\_layout()

plt.show()

# Print Extracted Features

print(f"Pulse Amplitude (mean): {np.mean(pulse\_amplitude):.2f} V")

print(f"Average Pulse Duration: {np.mean(pulse\_duration):.2f} seconds")

print(f"Average Inter-Beat Interval (IBI): {np.mean(ibi):.2f} seconds")

print(f"IBI Variability (Standard Deviation): {ibi\_variability:.2f}")

print(f"Peak-to-Peak Interval (mean): {np.mean(ppi):.2f} seconds")

print(f"Signal Entropy: {ppg\_entropy:.2f}")

print(f"RMS of the Signal: {rms\_value:.2f}")

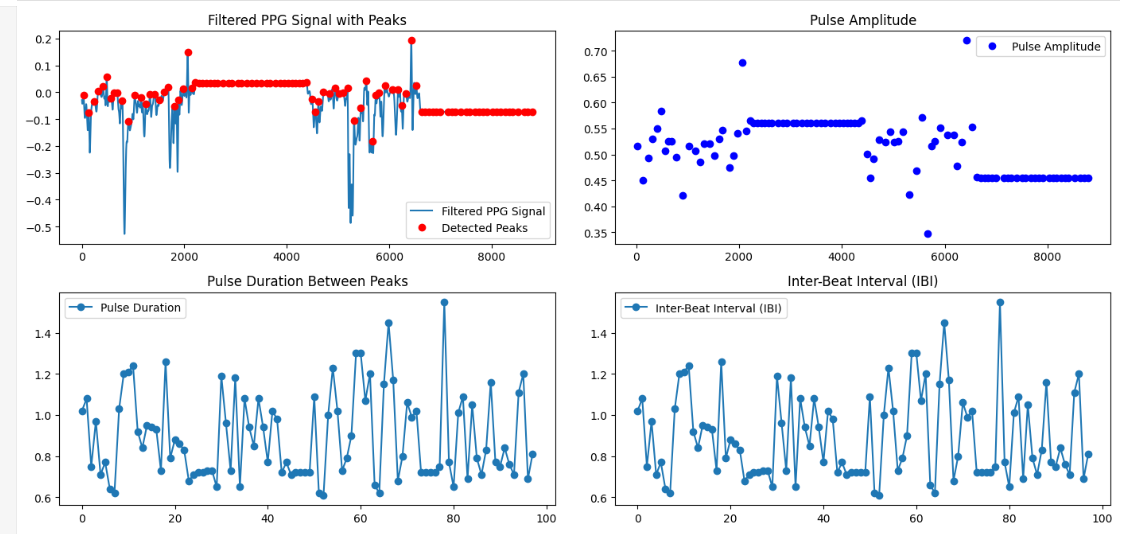
print(f"Dominant Frequency: {dominant\_frequency:.2f} Hz")

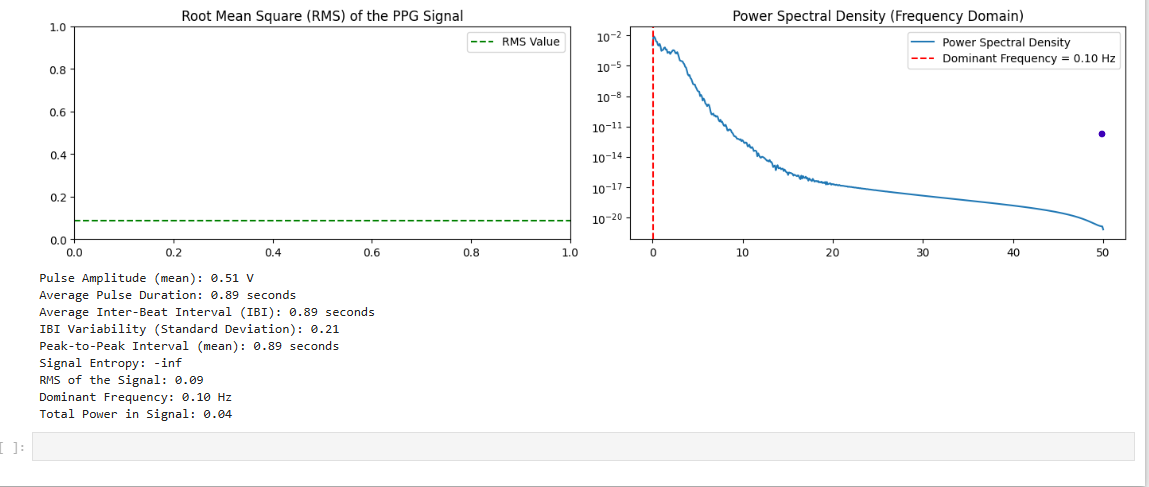
print(f"Total Power in Signal: {total\_power:.2f}")

**Input:**

* PPG Signal: The raw PPG signal is provided in the form of a CSV file containing the "Imf\_1\_MEAN" column.
* Sampling Frequency (fs): 100 Hz, indicating the number of samples per second.
* Lowpass Filter Cutoff Frequency**:** 3 Hz to remove high-frequency noise.

**Output:**





**Conclusion:**

The analysis of the PPG signal through filtering, peak detection, and feature extraction enables effective characterization of cardiovascular health. Key features like pulse amplitude, pulse duration, inter-beat interval, signal entropy, and RMS provide insights into heart rate variability, signal strength, and regularity. Frequency domain analysis through power spectral density and dominant frequency estimation further enhances the understanding of the heart's rhythmic patterns. This approach is valuable for non-invasive heart rate monitoring and can assist in assessing overall cardiovascular function.

**Problem number:** 06

**Title**: Fourier Transform and Frequency Spectrum Analysis of a Signal.

**Theory:**

The Fourier Transform is a fundamental tool in signal processing used to convert a time-domain signal into its frequency-domain representation. This transformation reveals the frequency components that make up a given signal, allowing for a deeper analysis of its behavior. For discrete signals, the Discrete Fourier Transform (DFT) is employed, which computes the frequency components by considering sampled data points over time.

In this experiment, the Fast Fourier Transform (FFT) algorithm is used, which is an efficient method for computing the DFT. The FFT significantly reduces the computational complexity, making it an essential technique for analyzing signals, particularly in real-time applications.

The experiment begins with the generation of a signal composed of two sine waves at 50 Hz and 150 Hz. This signal is sampled at a rate of 1000 Hz. After obtaining the time-domain signal, the FFT is applied to transform the signal into the frequency domain. The resulting spectrum is then analyzed, where the magnitude of the frequency components is plotted to reveal the primary frequencies present in the signal.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

fs = 1000

T = 1 / fs

L = 1000

t = np.arange(0, L) \* T

f1 = 50

f2 = 150

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

Y = np.fft.fft(signal)

P2 = np.abs(Y / L)

P1 = P2[:L // 2]

f = fs \* np.arange(0, (L // 2)) / L

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

plt.subplot(2, 1, 1)

plt.plot(t, signal, 'b')

plt.title('Original Signal')

plt.xlabel('Time (seconds)')

plt.ylabel('Amplitude')

plt.grid(True)

plt.subplot(2, 1, 2)

plt.plot(f, P1, 'r')

plt.title('Frequency Spectrum')

plt.xlabel('Frequency (Hz)')

plt.ylabel('|P1(f)|')

plt.grid(True)

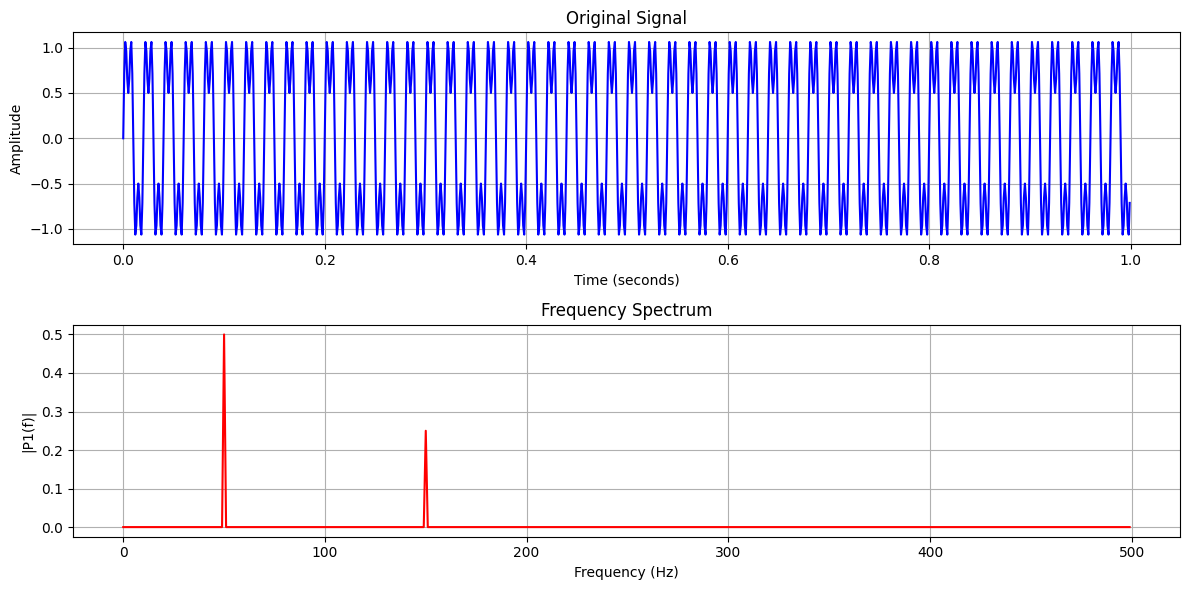
plt.tight\_layout()

plt.show()

**Input:**

* A time-domain signal generated by combining two sine waves with frequencies of 50 Hz and 150 Hz.
* The signal is sampled at a frequency of 1000 Hz for 1000 samples.

**Output:**



**Conclusion:**

In this lab, the Fast Fourier Transform (FFT) was successfully applied to analyze a time-domain signal composed of two sine waves. The frequency-domain representation provided insights into the signal's frequency components. By using FFT, we efficiently computed the signal's spectrum, demonstrating the practical advantages of FFT in signal processing. This lab reinforced the concept of frequency analysis and highlighted the importance of FFT in real-time signal processing applications.

**Problem Number:** 07

**Title:** Implementation of Discrete Fourier Transform (DFT) and Noise Filtering Using Python

**Theory:**

The Discrete Fourier Transform (DFT) is a mathematical technique used to analyze the frequency components of a discrete signal. It transforms a signal from the time domain (amplitude vs. time) to the frequency domain (amplitude vs. frequency). This is particularly useful for identifying the frequencies present in a signal, such as in audio processing, image processing, and communication systems.

The DFT of a signal x[n] with N samples is given by:

X[k]=

where:

* X[k] is the frequency domain representation of the signal.
* *k* is the frequency bin index.
* j is the imaginary unit.

The Inverse DFT (IDFT) is used to reconstruct the time-domain signal from its frequency-domain representation.

**Source Code(python):**

import numpy as np

import matplotlib.pyplot as plt

def DFT(x):

"""

Compute the Discrete Fourier Transform (DFT) of a 1D signal.

"""

N = len(x)

X = np.zeros(N, dtype=complex) # Output array (complex numbers)

for k in range(N): # Loop over frequency bins

for n in range(N): # Loop over time samples

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

return X

# Create a sample signal (two sine waves)

Fs = 1000 # Sampling rate

T = 1 / Fs # Sampling interval

t = np.linspace(0, 1, Fs, endpoint=False) # 1 second duration

# Signal: Combination of 50 Hz and 120 Hz sine waves

f1, f2 = 50, 120

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

# Compute DFT

dft\_output = DFT(signal)

# Compute frequency bins

freqs = np.fft.fftfreq(len(dft\_output), T)

# Plot magnitude spectrum (single-sided)

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

plt.plot(freqs[:Fs//2], np.abs(dft\_output[:Fs//2])) # Single-sided spectrum

plt.title("DFT Frequency Spectrum")

plt.xlabel("Frequency (Hz)")

plt.ylabel("Magnitude")

plt.grid()

plt.show()

# Compute Frequency Bins Manually

N = 1024 # Number of points in DFT

Fs = 1000 # Sampling frequency (Hz)

# Compute frequency bins manually

freq\_bins = np.array([(k / N) \* Fs for k in range(N)

print(freq\_bins[:10]) # Print first 10 frequency bins

# Compute Frequency Bins Using NumPy

N = 1024 # Number of points in DFT

Fs = 1000 # Sampling frequency

# Compute frequency bins using NumPy

freq\_bins = np.fft.fftfreq(N, d=1/Fs)

print(freq\_bins[:10]) # Print first 10 frequency bins

# Example: Removing Noise from an Audio Signal

from scipy.fft import fft, ifft, fftfreq

# Generate a sample audio signal

Fs = 1000 # Sampling rate (1000 Hz)

T = 1 / Fs # Sampling interval

t = np.linspace(0, 1, Fs, endpoint=False) # 1 second time vector

# Generate a pure sine wave (440 Hz, like an "A4" musical note)

freq\_signal = 440

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

# Add random noise

noise = np.random.normal(0, 0.5, pure\_signal.shape)

noisy\_signal = pure\_signal + noise

# Apply FFT

fft\_signal = fft(noisy\_signal)

freqs = fftfreq(len(fft\_signal), T) # Frequency bins

# Filter: Remove frequencies higher than 500 Hz

fft\_filtered = fft\_signal.copy()

fft\_filtered[np.abs(freqs) > 500] = 0 # Zero out high frequencies (noise)

# Apply Inverse FFT to get the cleaned signal

cleaned\_signal = ifft(fft\_filtered).real

# Plot the results

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

plt.subplot(3, 1, 1)

plt.plot(t, pure\_signal, label="Original Signal (440 Hz)")

plt.legend()

plt.title("Original Pure Signal")

plt.subplot(3, 1, 2)

plt.plot(t, noisy\_signal, label="Noisy Signal", color="red")

plt.legend()

plt.title("Noisy Signal")

plt.subplot(3, 1, 3)

plt.plot(t, cleaned\_signal, label="Cleaned Signal (After FFT Filtering)", color="green")

plt.legend()

plt.title("Filtered Signal (Noise Removed)")

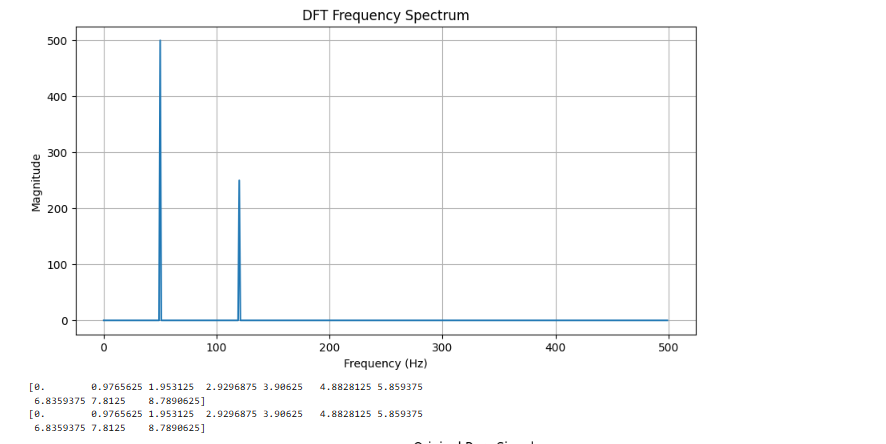
plt.tight\_layout()

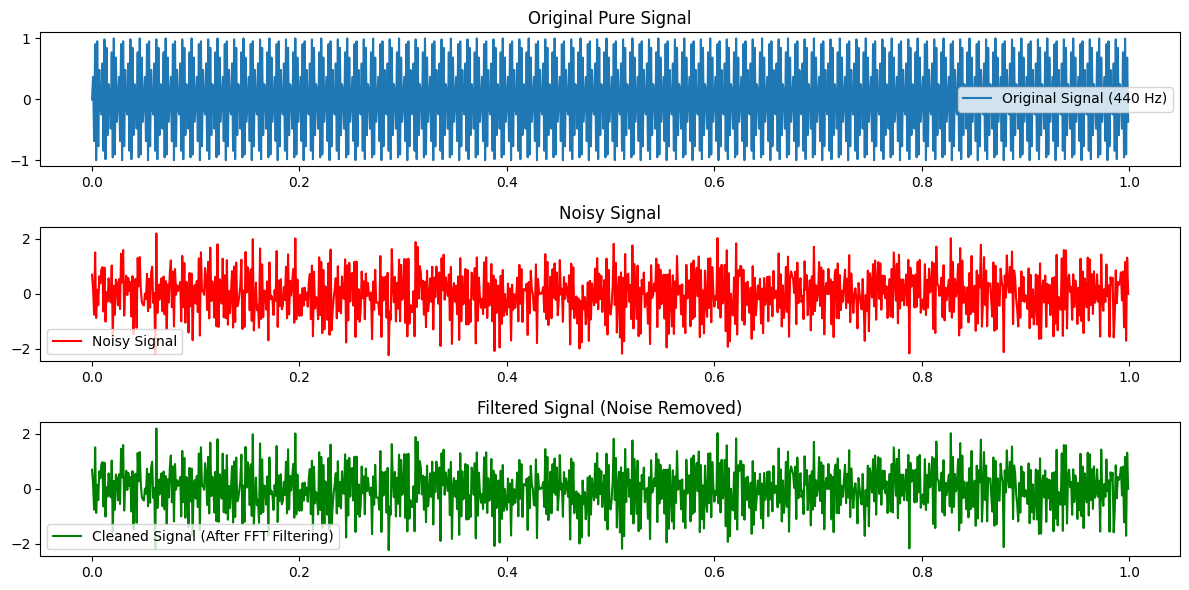
plt.show()

**Input:**

1. Sample Signal: A combination of two sine waves, one at 50 Hz and another at 120 Hz, sampled at a rate of 1000 Hz.
2. Noisy Audio Signal: A pure sine wave at 440 Hz (A4 musical note), with random Gaussian noise added, sampled at 1000 Hz.

**Output:**





**Conclusion:**

The purpose of this lab was to:

1. Understand and implement the Discrete Fourier Transform (DFT) in Python.
2. Analyze the frequency components of a signal using DFT.
3. Use the Fast Fourier Transform (FFT) to filter noise from a noisy signal.
4. Reconstruct the cleaned signal using the Inverse DFT (IDFT).