**PABNA UNIVERSITY OF SCIENCE AND TECHNOLOGY**



**Faculty of Engineering & Technology**

**Department of Information and Communication Engineering**

**Course name: Signal and Systems Sessional**

**Course Code: ICE-2204**

Submitted By:

**Name: Md. Uzzal Mia**

**Roll:** 220605

Reg: 1065452

Session:2021-2022

2nd Year 2nd Semester

Dept. of ICE, PUST

Submitted To:

**Dr. Md. Imran Hossain**

Associate Professor

Department of Information and Communication Engineering

Pabna University of Science and Technology

Pabna, Bangladesh

**Index**

|  |  |
| --- | --- |
| Problem Number | Title |
| 01 | Implementation and Analysis of Impulse, Step, and Ramp Functions in Discrete-Time Signals |
| 02 | Basic Operations on Discrete-Time Signals: Addition, Subtraction, Multiplication, Shifting, and Folding |
| 03 | Convolution, Cross-Correlation, and Autocorrelation of Signals |
| 04 | Processing and Heart Rate Detection from PPG Signal |
| 05 | Discrete Fourier Transform (DFT) and Frequency Spectrum Analysis |
| 06 | Noise Reduction in an Audio Signal Using FFT Filtering |

**Problem Number: 01**

**Title:** Implementation and Analysis of Impulse, Step, and Ramp Functions in Discrete-Time Signals

**Objectives:**

1. To understand and implement fundamental discrete-time signals Unit Impulse (Dirac Delta) Function, Unit Step Function, Unit Ramp Function

2. To visualize these signals using Python (NumPy & Matplotlib).

3. To analyze their mathematical representations and behaviors in discrete-time domains.

4. To establish a foundation for signal processing applications like convolution and system analysis.

**Theory:** In Digital Signal Processing (DSP), three fundamental discrete-time signals are commonly used:

1️. Unit Impulse Function (δ[n]):

* Defined as 1 at n = 0, and 0 elsewhere.
* Used to test system responses and as a building block for other signals.

2. Unit Step Function (u[n]):

* Defined as 1 for n ≥ 0, and 0 for n < 0.
* Represents systems that turn on at a certain time.
* Integral (summation) of the impulse function.

3. Unit Ramp Function (r[n]):

* Defined as n for n ≥ 0, and 0 for n < 0.
* Represents a linearly increasing signal over time.
* Integral (summation) of the step function.

These signals are fundamental for analyzing and designing digital systems, helping in convolution, system response analysis, and control system

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

# Define functions

def impulse\_signal(n):

return np.where(n == 0, 1, 0) # δ[n] = 1 for n=0, else 0

def step\_signal(n):

return np.where(n >= 0, 1, 0) # u[n] = 1 for n >= 0, else 0

def ramp\_signal(n):

return np.where(n >= 0, n, 0) # r[n] = n for n >= 0, else 0

# Generate signals

n1 = np.arange(-10, 11) # For impulse and step

n2 = np.arange(-5, 6) # For ramp

impulse = impulse\_signal(n1)

step = step\_signal(n1)

ramp = ramp\_signal(n2)

# Create a figure with 3 subplots

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

# Plot Unit Impulse Function

plt.subplot(3, 1, 1)

plt.stem(n1, impulse, use\_line\_collection=True)

plt.xlabel('n')

plt.ylabel('δ[n]')

plt.title('Unit Impulse Function')

plt.grid(True)

# Plot Unit Step Function

plt.subplot(3, 1, 2)

plt.stem(n1, step, use\_line\_collection=True)

plt.xlabel('n')

plt.ylabel('u[n]')

plt.title('Unit Step Sequence')

plt.grid(True)

# Plot Unit Ramp Function

plt.subplot(3, 1, 3)

plt.stem(n2, ramp, use\_line\_collection=True)

plt.xlabel('n')

plt.ylabel('r[n]')

plt.title('Ramp Function')

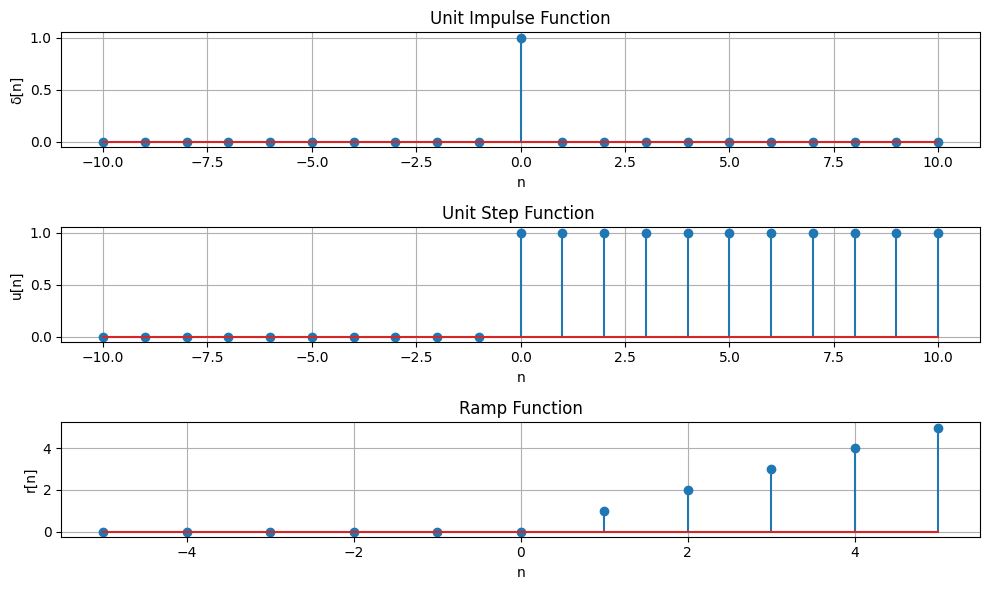
plt.grid(True)

# Adjust layout and show plot

plt.tight\_layout()

plt.show()

**Output:**



**Figure 01: Unit Impulse, Unit Step, and Ramp Function**

**Problem Number: 02**

**Title:** Basic Operations on Discrete-Time Signals: Addition, Subtraction, Multiplication, Shifting, and Folding

**Objectives:**

1. Addition, Subtraction, Multiplication: These operations modify signal amplitude by combining or altering values at each index.
2. Shifting: Moves a signal left (advance) or right (delay), affecting its alignment in time.
3. Folding: Flips the signal horizontally, reversing its time indices.

**Theory:** Discrete-time signals are sequences of values defined at specific time indices. Several fundamental operations can be performed on these signals:

* Addition: Adds two signals element-wise, combining their amplitudes.
* Subtraction: Subtracts one signal from another, showing differences at each index.
* Multiplication: Multiplies two signals element-wise, affecting their amplitudes.
* Shifting: Moves a signal left (advance) or right (delay) in time by a given shift value.
* Folding: Reverses the signal along the time axis, flipping it horizontally.

These operations are essential in signal processing, filtering, and transformation analysis.

**Source Code:**import numpy as np

import matplotlib.pyplot as plt

def signal\_add(x1, n1, x2, n2):

    n\_min = min(min(n1), min(n2))

    n\_max = max(max(n1), max(n2))

    n = np.arange(n\_min, n\_max+1)

    y1 = np.zeros(len(n))

    y2 = np.zeros(len(n))

    y1[np.isin(n, n1)] = x1

    y2[np.isin(n, n2)] = x2

    return y1+y2, n

def signal\_sub(x1, n1, x2, n2):

    n\_min = min(min(n1), min(n2))

    n\_max = max(max(n1), max(n2))

    n = np.arange(n\_min, n\_max+1)

    y1 = np.zeros(len(n))

    y2 = np.zeros(len(n))

    y1[np.isin(n, n1)] = x1

    y2[np.isin(n, n2)] = x2

    return y1-y2, n

def signal\_mul(x1, n1, x2, n2):

    n\_min = min(min(n1), min(n2))

    n\_max = max(max(n1), max(n2))

    n = np.arange(n\_min, n\_max+1)

    y1 = np.zeros(len(n))

    y2 = np.zeros(len(n))

    y1[np.isin(n, n1)] = x1

    y2[np.isin(n, n2)] = x2

    return y1\*y2, n

# *Define two signals*

x1 = np.array([1, 2, 3, 4, 5])  # *First signal*

x2 = np.array([5, 4, 3, 2, 1])  # *Second signal*

# *Define corresponding indices*

n1 = np.array([0, 1, 2, 3, 4])  # *First signal index*

n2 = np.array([1, 2, 3, 4, 5])  # *Second signal index*

x\_add, n\_add = signal\_add(x1, n1, x2, n2)

x\_sub, n\_sub = signal\_sub(x1, n1, x2, n2)

x\_mul, n\_mul = signal\_mul(x1,n1,x2,n2)

# *Create a figure*

plt.figure(figsize=(10, 8))

# *Plot Original Signal x1*

plt.subplot(5, 1, 1)

plt.stem(n1, x1)

plt.title('Original Signal x1')

plt.xlabel('n1')

plt.ylabel('x1(n)')

plt.grid(True)

# *Plot Original Signal x2*

plt.subplot(5, 1, 2)

plt.stem(n2, x2)

plt.title('Original Signal x2')

plt.xlabel('n')

plt.ylabel('x2(n)')

plt.grid(True)

# *Plot Addition*

plt.subplot(5, 1, 3)

plt.stem(n\_add,x\_add)

plt.title('Signal Addition')

plt.xlabel('n')

plt.ylabel('x1(n) + x2(n)')

plt.grid(True)

# *Plot Subtraction*

plt.subplot(5, 1, 4)

plt.stem(n\_sub, x\_sub)

plt.title('Signal Subtraction')

plt.xlabel('n')

plt.ylabel('x1(n) - x2(n)')

plt.grid(True)

# *Plot Multiplication*

plt.subplot(5, 1, 5)

plt.stem(n\_mul, x\_mul)

plt.title('Signal Multiplication')

plt.xlabel('n')

plt.ylabel('x1(n) \* x2(n)')

plt.grid(True)

# *Adjust layout*

plt.tight\_layout()

plt.show()

# ----------------- Signal Shifting and Folding -----------------

import numpy as np

import matplotlib.pyplot as plt

# *Function for signal shifting*

def signal\_shifting(n, shift):

    return n + shift  # *Shift indices by given amount*

# *Function for signal folding (time reversal)*

def signal\_folding(x, n):

    return np.flip(x), -np.flip(n)

n = np.array([-2, -1, 0, 1, 2])

x = np.array([1, 2, 3, 4, 5])     #

folded\_x, folded\_n = signal\_folding(x, n)  # *Signal values*

# *Perform shifting (Right shift by 2)*

n\_shifted = signal\_shifting(n, 2)

# *n\_folded = -n  # Fold indices*

# *Plot the original signal*

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

plt.subplot(3, 1, 1)

plt.stem(n, x1, markerfmt='bo')

plt.title('Original Signal')

plt.xlabel('n')

plt.ylabel('x(n)')

plt.grid(True)

# *Plot shifted signal (Right shift by 2)*

plt.subplot(3, 1, 2)

plt.stem(n\_shifted, x1, markerfmt='ro')

plt.title('Shifted Signal (Right Shift by 2)')

plt.xlabel('n')

plt.ylabel('x(n-2)')

plt.grid(True)

# *Plot folded signal (Time Reversal)*

plt.subplot(3, 1, 3)

plt.stem(folded\_n, folded\_x, markerfmt='go')

plt.title('Folded Signal (Time Reversal)')

plt.xlabel('n')

plt.ylabel('x(-n)')

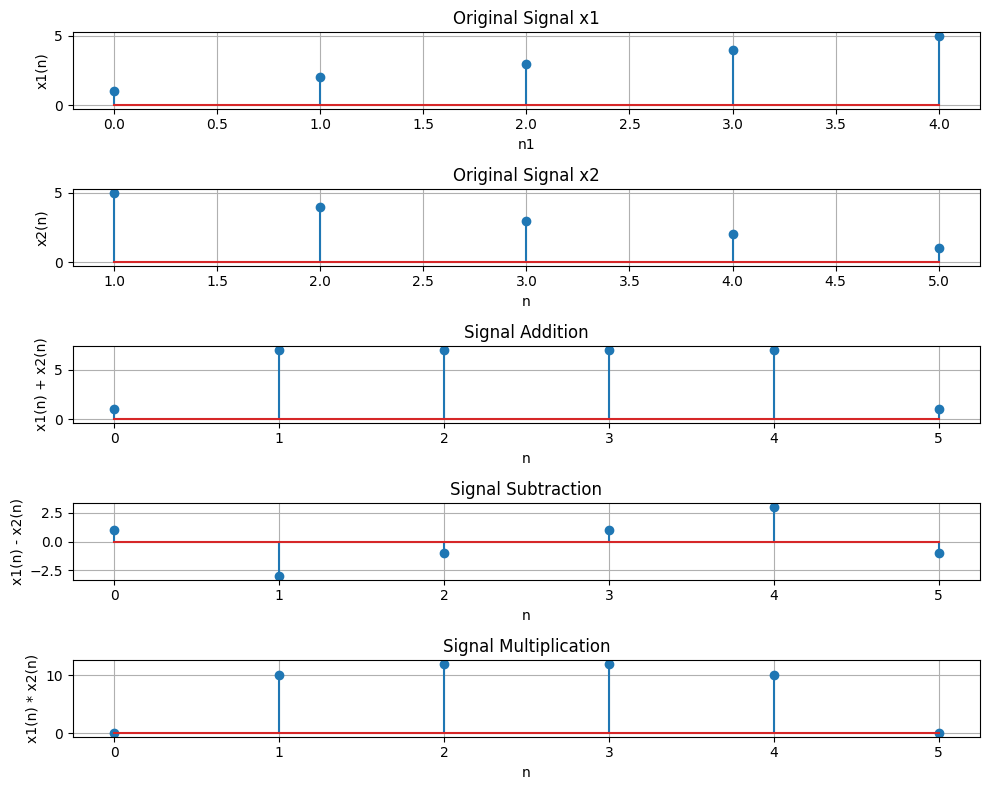
plt.grid(True)

# *Show the plots*

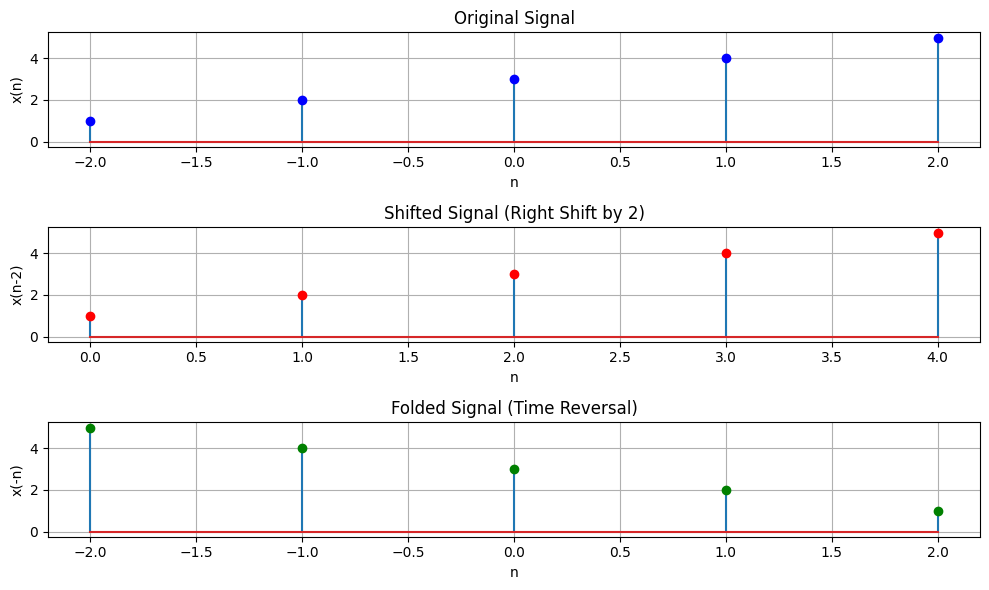
plt.tight\_layout()

plt.show()

**Output:**

****

**Figure 02: Signal operation addition, subtraction, multiplication**

****

**Figure 03: Signal Shifting and folding**

**Problem Number: 03**

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

**Objectives:**

1. To perform and visualize the convolution of a signal with a kernel.
2. To compute and plot the cross-correlation between two signals.
3. To compute and plot the autocorrelation of a signal.
4. To understand how correlation and convolution are related

**Theory:**

* Convolution is a mathematical operation that combines two signals to produce a third. It is commonly used in signal processing and image analysis. In discrete time, convolution is computed by flipping the kernel (or filter) and sliding it across the signal.
* Cross-correlation is similar to convolution but without flipping the kernel. It is used to measure the similarity between two signals.
* Autocorrelation is a special case of cross-correlation where the signal is correlated with itself. It is useful for identifying repeating patterns in a signal

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import correlate, correlation\_lags

# Define signal and kernel

signal = np.array([1, 2, 3, 4, 5])

kernel = np.array([1, 0, -1])

# Perform convolution (same as cross-correlation with flipped kernel)

convolved\_signal = np.convolve(signal, kernel, mode='full')

# Perform cross-correlation by flipping the kernel

correlated\_signal = np.correlate(signal, kernel, mode='full')

# Plot signals

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

# Plot the original signal

plt.subplot(1, 3, 1)

plt.stem(signal)

plt.title('Original Signal')

plt.grid(True)

# Plot the convolved signal

plt.subplot(1, 3, 2)

plt.stem(convolved\_signal, linefmt='r', markerfmt='ro')

plt.title('Convolved Signal')

plt.grid(True)

# Plot the correlated signal

plt.subplot(1, 3, 3)

plt.stem(correlated\_signal, linefmt='g', markerfmt='go')

plt.title('Correlated Signal')

plt.grid(True)

plt.tight\_layout()

plt.show()

# Define autocorrelation function

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

# Define cross-correlation function

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

# Sampling frequency and time vector

fs = 1000

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

freq = 5

# Sinusoidal signal

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

# Autocorrelation of signal1

auto\_corr, lags\_auto = compute\_autocorrelation(signal1)

# Cross-correlation with a shifted version of the signal

signal2 = np.roll(signal1, 100)

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

# Cross-correlation with noisy signal

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

noisy\_signal = signal1 + noise

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

# Plot all results in one figure

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

# Autocorrelation plot

plt.subplot(3, 1, 1)

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

plt.title("Autocorrelation of Sinusoidal Signal")

plt.xlabel("Lag")

plt.ylabel("Autocorrelation")

plt.grid(True)

# Cross-correlation plot (shifted signal)

plt.subplot(3, 1, 2)

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

plt.title("Cross-Correlation between Original and Shifted Signal")

plt.xlabel("Lag")

plt.ylabel("Cross-Correlation")

plt.grid(True)

# Cross-correlation with noisy signal

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")

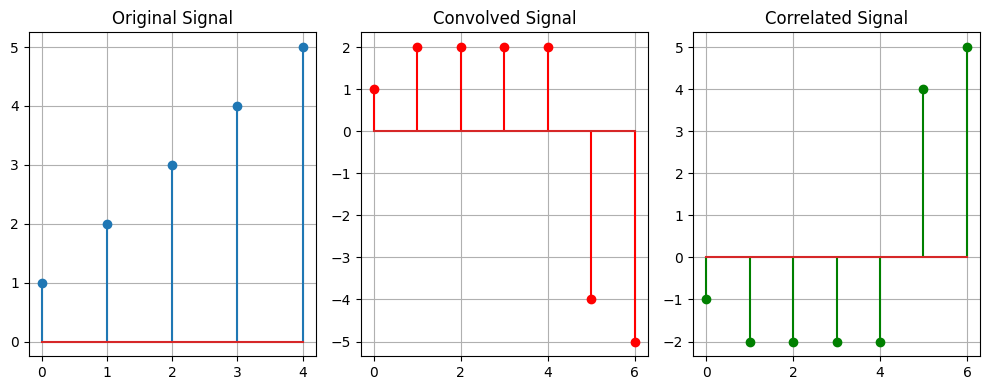
plt.grid(True)

# Adjust layout and show the plot

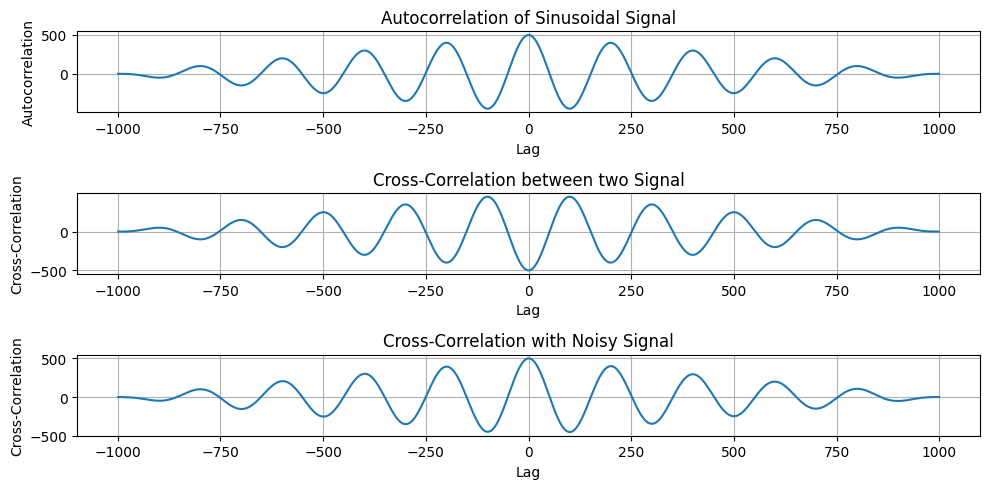
plt.tight\_layout()

plt.show()

**Output:**

****

**Figure 04: Convolved and Correlated Signal**

****

**Figure 05: Autocorrelation of Sinusoidal Signal**

**Problem Number: 04**

**Title:** Processing and Heart Rate Detection from PPG Signal

**Objectives:**

1. To generate and visualize a raw sinusoidal signal, noise signal, and a simulated PPG signal.
2. To apply a bandpass filter to the PPG signal to remove unwanted noise.
3. To normalize the PPG signal to standardize the amplitude.
4. To detect heartbeats using peak detection and compute the heart rate in beats per minute (BPM).

**Theory:**

* PPG (Photoplethysmogram) is a non-invasive optical measurement used to detect blood volume changes, typically for heart rate monitoring.
* A bandpass filter is used to allow frequencies within a certain range to pass through, removing low-frequency noise (e.g., from baseline wander) and high-frequency noise (e.g., from muscle artifacts).
* Peak detection is used to identify the time points of heartbeats in the filtered PPG signal. The Inter-Beat Interval (IBI) is calculated from the time difference between consecutive peaks, and the heart rate is derived by converting the IBI into beats per minute (BPM).

**Source Code:**

*import* numpy *as* np

*import* matplotlib.pyplot *as* plt

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

# *Signal parameters*

fs = 100

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

# *Generate signals*

sin\_signal = 0.6 \* np.sin(2 \* np.pi \* 1.2 \* t)

# *Plotting*

plt.figure(figsize=(12, 10))

# *Subplot 1: Raw Sin Signal*

plt.subplot(3, 2, 1)

plt.plot(t, sin\_signal)

plt.title("Raw Sin Signal")

plt.xlabel("Time (seconds)")

plt.ylabel("Amplitude")

noise\_signal = np.random.normal(0, 0.05, len(t))

# *Subplot 2: Raw Noise Signal*

plt.subplot(3, 2, 2)

plt.plot(t, noise\_signal)

plt.title("Raw Noise Signal")

plt.xlabel("Time (seconds)")

plt.ylabel("Amplitude")

ppg\_signal = sin\_signal + noise\_signal

# *Subplot 3: Raw PPG Signal*

plt.subplot(3, 2, 3)

plt.plot(t, ppg\_signal)

plt.title("Raw PPG Signal")

plt.xlabel("Time (seconds)")

plt.ylabel("Amplitude")

# *Bandpass filter function*

def bandpass\_filter(signal, lowcut, highcut, fs, 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)

# *Filter and normalize PPG signal*

filtered\_ppg = bandpass\_filter(ppg\_signal, 0.5, 5, fs)

# *Subplot 4: Filtered PPG Signal*

plt.subplot(3, 2, 4)

plt.plot(t, filtered\_ppg)

plt.title("Filtered PPG Signal")

plt.xlabel("Time (seconds)")

plt.ylabel("Amplitude")

normalized\_ppg = (filtered\_ppg - np.min(filtered\_ppg)) / (np.max(filtered\_ppg) - np.min(filtered\_ppg))

# *Subplot 5: Normalized PPG Signal*

plt.subplot(3, 2, 5)

plt.plot(t, normalized\_ppg)

plt.title("Normalized PPG Signal")

plt.xlabel("Time (seconds)")

plt.ylabel("Normalized Amplitude")

# *Detect peaks in the PPG signal*

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

ibi = np.diff(peaks) / fs  # *Inter-beat interval in seconds*

heart\_rate = 60 / ibi  # *Heart rate in BPM*

# *Subplot 6: PPG Signal with Detected Peaks*

plt.subplot(3, 2, 6)

plt.plot(t, normalized\_ppg)

plt.plot(t[peaks], normalized\_ppg[peaks], "x", label="Peaks")

plt.title("PPG Signal with Detected Peaks (Heartbeats)")

plt.xlabel("Time (seconds)")

plt.ylabel("Normalized Amplitude")

plt.legend()

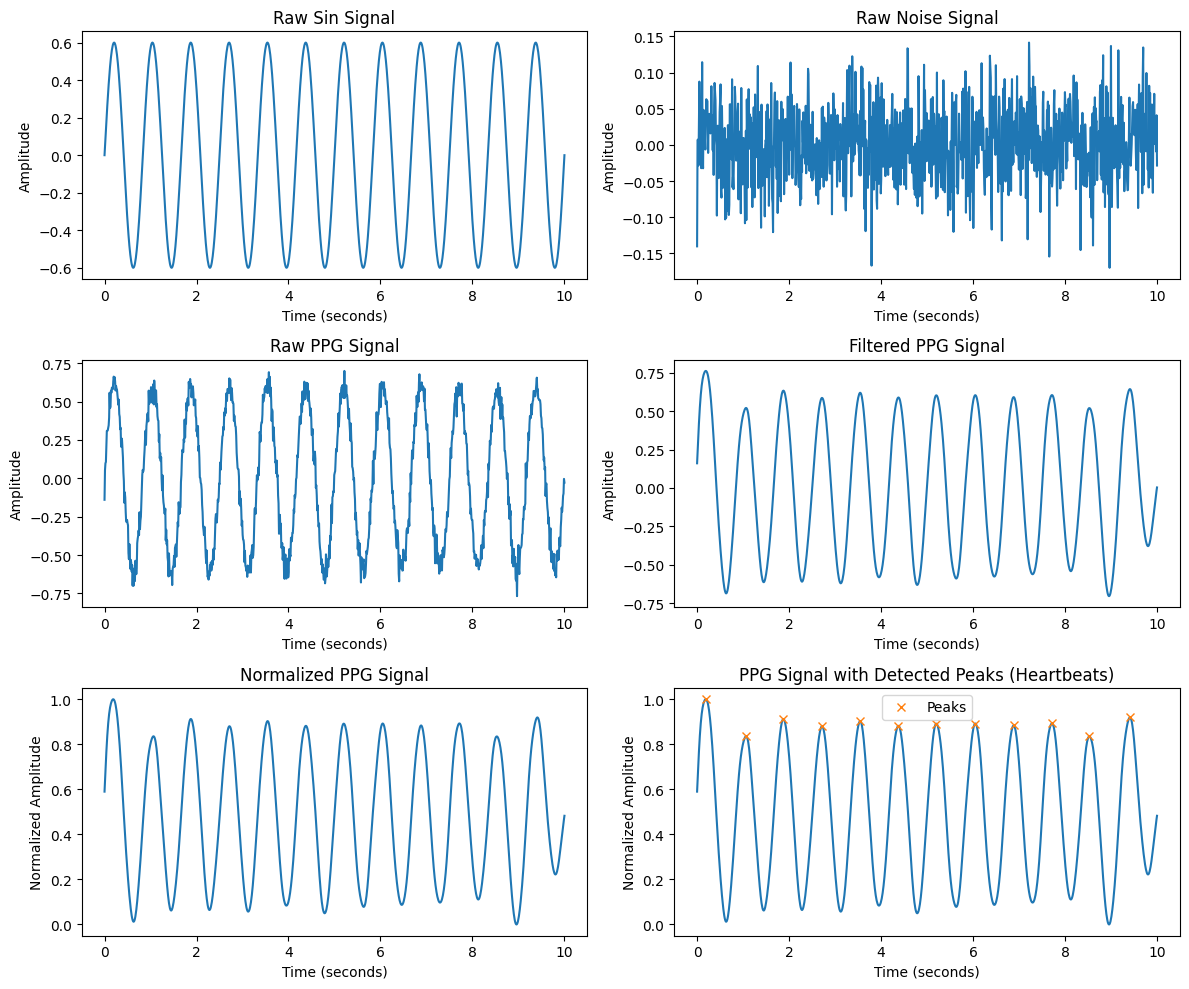
plt.tight\_layout()

plt.show()

# *Print Heart Rate*

print("Heart Rate: ", np.mean(heart\_rate), " BPM")

**Output:**

****

**Figure 06: PPG Signal Processing**

**Problem Number: 05**

**Title:** Discrete Fourier Transform (DFT) and Frequency Spectrum Analysis

**Objectives:**

1. Understand the Discrete Fourier Transform (DFT): Learn how DFT converts a time-domain signal into its frequency-domain representation.
2. Implement DFT in Python: Compute the DFT manually using nested loops to understand it’s working.
3. Analyze Frequency Components: Decompose a signal into its constituent frequencies and observe their magnitudes.
4. Visualize the Spectrum: Plot the frequency spectrum to identify dominant frequency components in a given signal.
5. Compare with FFT: Recognize how manual DFT implementation relates to the efficient Fast Fourier Transform (FFT) algorithm.

**Theory:**

The Discrete Fourier Transform (DFT) is a fundamental mathematical tool used in signal processing to analyze the frequency components of a discrete-time signal. It converts a time-domain signal into its frequency-domain representation.

Implementation in Python:

The provided Python code computes the DFT of a sampled signal containing two sine waves at 50 Hz and 120 Hz. It:

1. Generates a time-domain signal composed of sine waves at different frequencies.
2. Computes the DFT manually using nested loops.
3. Uses np.fft.fftfreq to obtain corresponding frequency bins.
4. Plots the single-sided magnitude spectrum to visualize the frequency components present in the signal.

**Source Code:**

import numpy as np

import matplotlib.pyplot as plt

# Compute the Discrete Fourier Transform (DFT)

def DFT(x):

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()

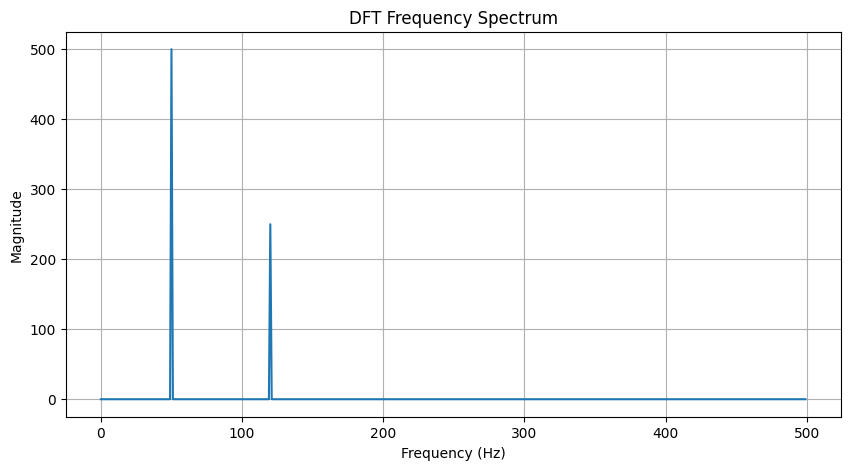
plt.show()

import numpy as np

import matplotlib.pyplot as plt

from scipy.fft import fft, ifft, fftfreq

**Output:**

****

**Figure 07: DFT Frequency Spectrum**

**Problem Number: 06**

**Title:** Noise Reduction in an Audio Signal Using FFT Filtering

**Objectives:**

1. Generate an Audio Signal: Create a pure sine wave signal (440 Hz, "A4" note).
2. Introduce Noise: Add random noise to simulate real-world interference.
3. Apply Fourier Transform: Use the Fast Fourier Transform (FFT) to analyze frequency components.
4. Filter High Frequencies: Remove noise by filtering out frequencies above 500 Hz.
5. Reconstruct the Signal: Use the Inverse FFT (IFFT) to obtain a cleaner signal.
6. Visualize Results: Compare the original, noisy, and filtered signals using plots

**Theory:** Fast Fourier Transform (FFT) is a mathematical algorithm used to convert a signal from the time domain to the frequency domain. The Inverse FFT (IFFT) allows us to reconstruct the time-domain signal after filtering.

A sine wave is a fundamental periodic signal that can be represented as:

x(t)=Asin(2πft)

where A is the amplitude, fff is the frequency, and ttt is time.

Steps in Noise Reduction Using FFT:

1. Transform to Frequency Domain: The noisy signal is converted using FFT.
2. Apply Filtering: Frequencies above 500 Hz are set to zero, preserving the fundamental 440 Hz component.
3. Reconstruct the Signal: The filtered signal is transformed back using IFFT, reducing noise.

This method is commonly used in signal processing to enhance audio quality and remove unwanted noise.

**Source Code:**

# 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")

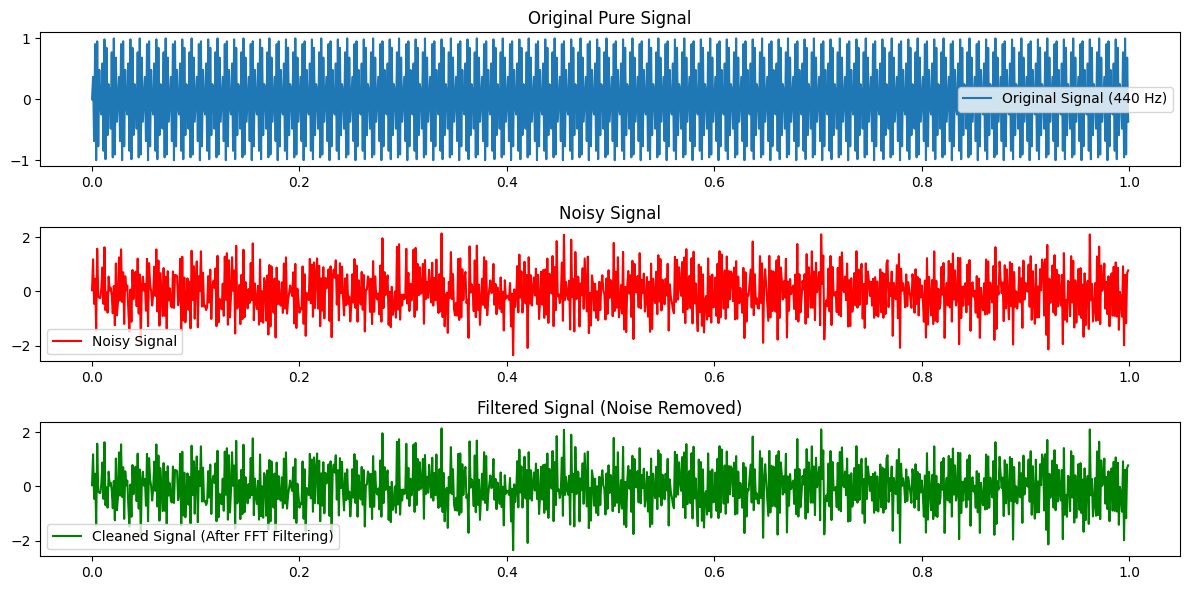
plt.legend()

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

plt.tight\_layout()

plt.show()

**Output:**

****

**Figure 08: Comparison of Original, Noisy, and Filtered Signals (FFT-Based Noise Removal)**