# 

|  |  |
| --- | --- |
| Index | Title |
| 01 | Plot the following signal operations using user defined function -adding ,b. multiplication, c. Scaling, d. shifting and e. folding. |
| 02 | Plot the following transformation x1(n) = 2x(n-5) - 3x(n+4) |
| 03 | Explain and Implementation the unit Impulse sequence, the unit Step sequence, the unit Ramp sequence. |
| 04 | Explain and Implement convolution of signal. |
| 05 | Explain and Implement correlation of signal. |
| 06 | Extract relevant features such as filtering, feature extraction, pick detection, heart rate etc. from PPG signal. |
| 07 | Explain and Implement Discrete Fourier Transform (DFT) using python. |
| 08 | Explain and Implement Frequency bin using python. |

# Problem 01 :

## Title: Plot following signal operations using user defined function –

## adding ,b. multiplication, c. Scaling, d. shifting and e. folding.

# Objective:

To visualize and understand basic signal operations: addition, multiplication, shifting, and folding using Python.

# Theory:

Signal operations are fundamental in digital signal processing (DSP) and allow us to modify signals for various applications. It involves manipulating signals to extract information, enhance features, or analyze behavior. Basic operations like addition, multiplication, scaling, and shifting are fundamental in understanding signal behavior in both time and frequency domains.  
  
1. Signal Addition: Combining two or more signals, often used in overlaying or superimposing information.  
 y(t) = x1(t) + x2(t)  
  
2. Signal Multiplication: Multiplying two signals results in a combined signal with modulated characteristics, often used in amplitude modulation.  
 y(t) = x1(t) \* x2(t)  
  
3. Scaling: Modifying the amplitude or duration of a signal.  
 - Amplitude Scaling: Changes the signal's magnitude by a constant.  
 - Time Scaling: Compresses or expands the signal along the time axis.  
 y(t) = k \* x(at)  
  
4. Shifting: In this operation, each sample of x(n) is shifted by an amount k to obtain a shifted sequence y(n).

y(n)={x(n−k)}

If we let m = n−k, then n = m+k and the above operation is given by

y(m+k) = {x(m)}  
These operations help in signal transformation, modulation, and system analysis.

5. Folding: In this operation each sample of x(n)is flipped around n =0 to obtain a folded sequence y(n).

y(n)={x(−n)}

# Purpose:

These operations help in processing signals for applications such as image and audio processing, communications, and system modeling.

# Source Code:

## # this code is written by APURBO SHARMA (220604)

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

**def** signal\_add(x1,x2):

**return** x1**+**x2

**def** signal\_mul(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)

t **=** np**.**arange(**-**10,10,0.01)

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

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

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

*# define two signal*

p1 **=** np**.**sin(2 **\*** np**.**pi **\*** 1 **\*** t)

p2 **=** np**.**cos(2 **\*** np**.**pi **\*** 0.5 **\*** t)

added\_signal **=** signal\_add(x1,x2)

multiplied\_signal **=** signal\_mul(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**=**(16,20))

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('multiplied\_signal')

plt**.**grid()

plt**.**subplot(4,2,5)

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

plt**.**xlabel("time")

plt**.**ylabel('amplitude')

plt**.**title('Original signal (x1\*2)')

plt**.**grid()

plt**.**subplot(4,2,6)

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

plt**.**xlabel("time")

plt**.**ylabel('amplitude')

plt**.**title('2 shifted signal')

plt**.**grid()

plt**.**subplot(4,2,7)

plt**.**stem(shifted\_signal2,x1)

plt**.**xlabel("time")

plt**.**ylabel('amplitude')

plt**.**title('-2 shifted signal')

plt**.**grid()

plt**.**subplot(4,2,8)

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

plt**.**xlabel("time")

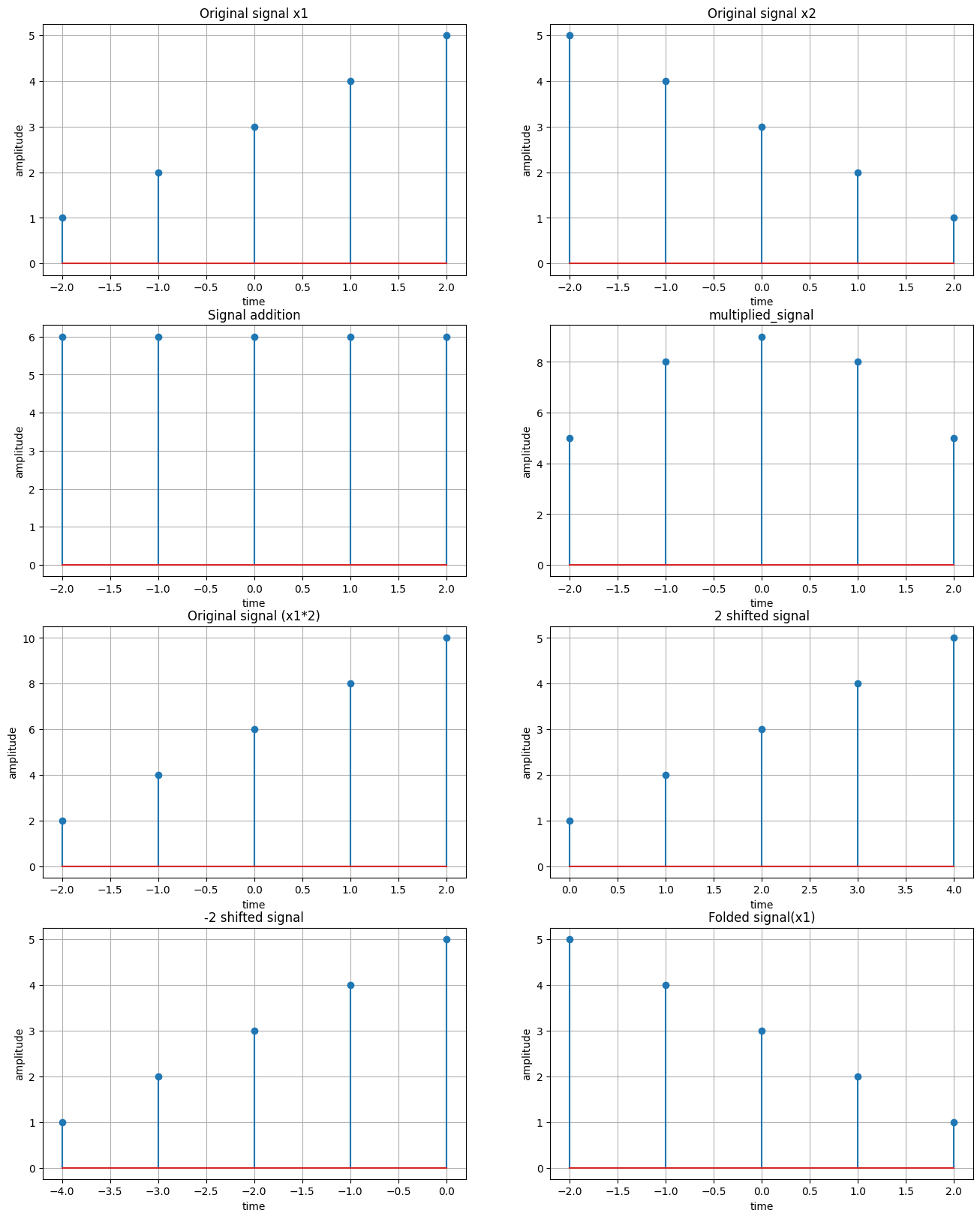
plt**.**ylabel('amplitude')

plt**.**title('Folded signal(x1)')

plt**.**grid()

# Input & Output:

* **Input:** Two discrete-time signals and an operation type (addition, multiplication, shifting, folding).
* **Output:** Graphical representation of the result below.



# Problem 02:

## Title: Plot the following transformation x1(n) = 2x(n-5) - 3x(n+4)

# Objective:

To plot the transformed signal x1(n) given by: x1(n)=2x(n−5)−3x(n+4) using Python.

# Theory:

This transformation consists of **scaling and shifting** operations applied to a discrete-time signal x(n).

1. **Shifting:**
   * x(n−5)) represents a **right shift** of x(n) by 5 units.
   * x(n+4) represents a **left shift** of x(n) by 4 units.
2. **Scaling:**
   * The coefficient **2** scales x(n−5), increasing its amplitude.
   * The coefficient **-3** scales x(n+4), flipping and amplifying it.

The transformation modifies the original signal by shifting and combining scaled versions.

# Purpose:

Understanding signal transformations helps in system modeling, filtering, and time-domain manipulations in DSP.

# Source Code:

## # this code is written by APURBO SHARMA (220604)

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

*# Define the original signal x(n) (e.g., unit impulse or step function)*

n **=** np**.**arange(**-**10, 10, 1) *# Discrete time range*

x\_n **=** np**.**zeros\_like(n) *# Start with a zero array*

*# For simplicity, we will define x(n) as a unit impulse at n=0*

x\_n[n **==** 0] **=** 1 *# Unit impulse at n=0*

*# Apply the transformation x1(n) = 2x(n-5) - 3x(n+4)*

x1\_n **=** 2 **\*** np**.**roll(x\_n, 5) **-** 3 **\*** np**.**roll(x\_n, **-**4)

*# Plot the original signal and the transformed signal*

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

plt**.**stem(n, x\_n)

plt**.**title("Original Signal x(n)")

Text(0.5, 1.0, 'Original Signal x(n)')

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

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

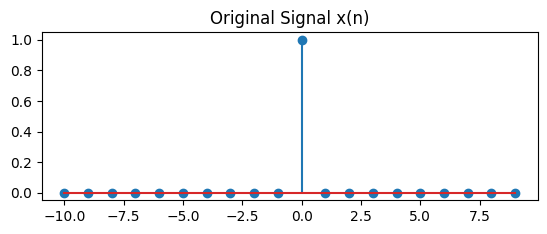
plt**.**title("Transformed Signal x1(n) = 2x(n-5) - 3x(n+4)")

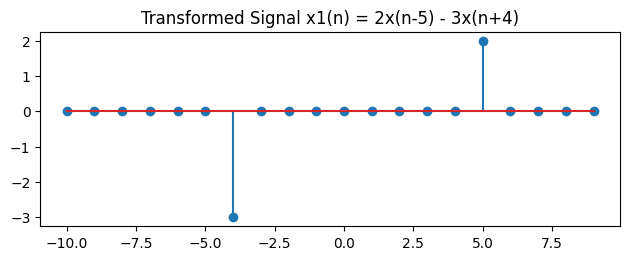
plt**.**tight\_layout()

plt**.**show()

# Input & Output:

* **Input:** A discrete-time signal x(n) over a defined range of n.
* **Output:** A plot of the transformed signal x1(n).





# Problem 03:

## Title: Explain and Implementation the unit Impulse sequence, the unit Step sequence, the unit Ramp sequence.

# Objective:

To generate and understand three fundamental signals: unit impulse, unit step, and unit ramp.

# Theory:

In signal processing and discrete-time systems, the **Unit Impulse Sequence**, **Unit Step Sequence**, and **Unit Ramp Sequence** are fundamental discrete-time signals used for analysis and system characterization. Here’s a detailed explanation of each:

# Purpose:

Understanding these signals helps in analyzing how systems respond to different types of inputs.

# Source Code:

## # this code is written by APURBO SHARMA (220604)

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

*# Generate a discrete-time index*

n **=** np**.**arange(**-**10, 11) *# Range from -10 to 10*

*# Generate a random signal*

random\_signal **=** np**.**random**.**uniform(**-**1, 1, len(n)) *# Random values between -1 and 1*

*# Step Sequence (u[n])*

step\_sequence **=** np**.**heaviside(n, 1) *# 1 for n >= 0, 0 for n < 0*

*# Impulse Sequence (δ[n])*

impulse\_sequence **=** np**.**zeros\_like(n) *# All zeros*

impulse\_sequence[n **==** 0] **=** 1 *# 1 at n = 0*

*# Ramp Sequence (r[n])*

ramp\_sequence **=** np**.**where(n **>=** 0, n, 0) *# n for n >= 0, else 0*

*# Plot the signals*

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

*# Plot Random Signal*

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

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

plt**.**title("Random Signal")

plt**.**xlabel("n")

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

*# Plot Step Sequence*

plt**.**subplot(4, 1, 2)

plt**.**stem(n, step\_sequence)

plt**.**title("Step Sequence u[n]")

plt**.**xlabel("n")

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

*# Plot Impulse Sequence*

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

plt**.**stem(n, impulse\_sequence)

plt**.**title("Impulse Sequence δ[n]")

plt**.**xlabel("n")

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

*# Plot Ramp Sequence*

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

plt**.**stem(n, ramp\_sequence)

plt**.**title("Ramp Sequence r[n]")

plt**.**xlabel("n")

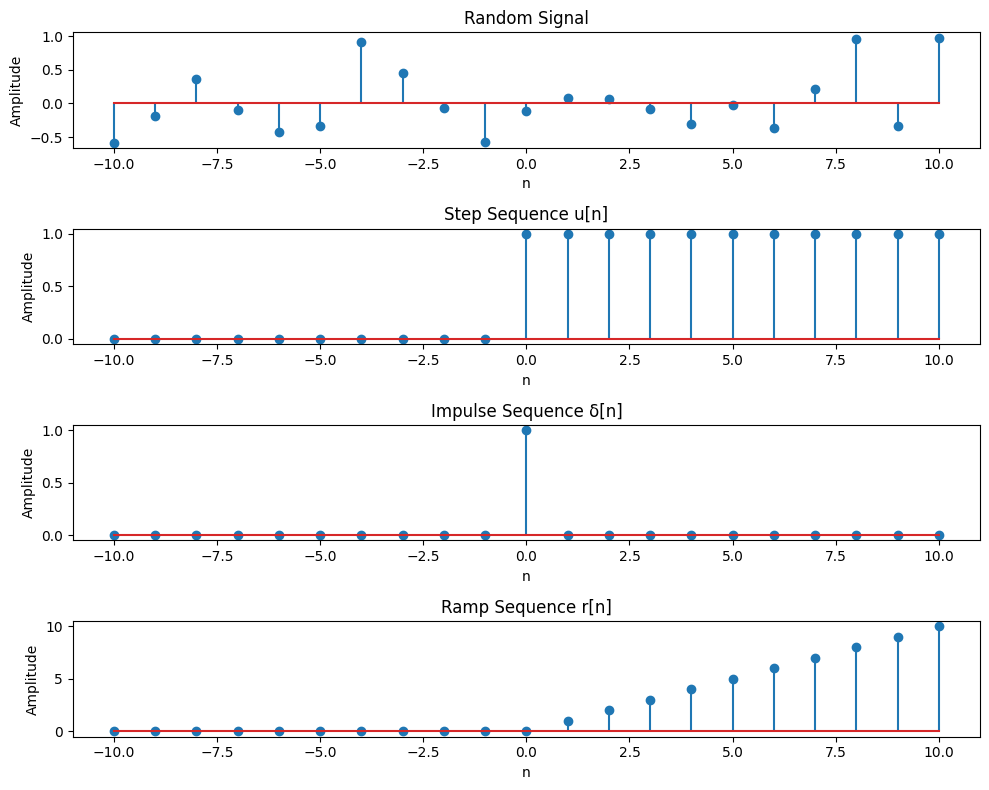
plt**.**ylabel("Amplitude")

plt**.**tight\_layout()

plt**.**show()

# Input & Output:

* **Input:** A range of discrete-time values.
* **Output:** Plotted unit impulse, unit step, and unit ramp sequences



# Problem 04:

## Title: Explain and Implement convolution of signal.

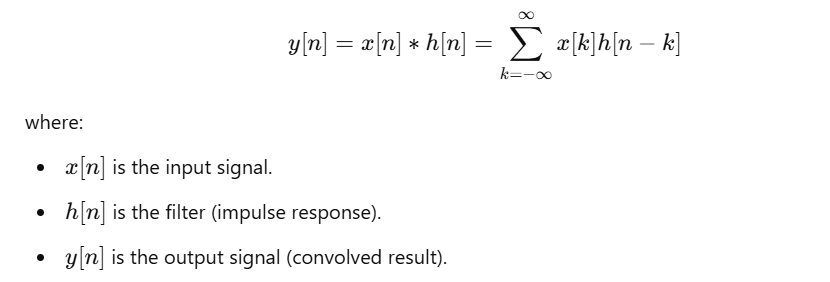
# Objective:

To generate and understand three fundamental signals: unit impulse, unit step, and unit ramp.

# Theory:

**Convolution** is a mathematical operation that combines two signals to produce a third signal. It is widely used in signal processing to determine the system's response to an input signal.

Mathematically, discrete-time convolution is given by:



# Purpose:

>To understand convolution and its effect on signals.

>To demonstrate how filtering affects a signal in DSP.

>To visualize how time-domain signals interact when convolved.

>To apply normalization for better interpretation of results.

>To develop skills in Python-based signal processing.

# Source Code:

## # this code is written by APURBO SHARMA (220604)

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

*# Generate two random signals*

np**.**random**.**seed(42) *# Ensures reproducibility*

x **=** np**.**random**.**randint(0, 10, 10) *# First random signal (length 10)*

h **=** np**.**random**.**randint(0, 10, 5) *# Second random signal (length 5)*

*# Compute convolution (It combines two signals to produce a third signal)*

y **=** np**.**convolve(x, h, mode**=**'full') *# mode='full' returns the complete convolution result of length (N+M-1).*

*# Normalize convolution result*

y\_norm **=** y **/** np**.**max(np**.**abs(y)) *# Normalize by maximum absolute value*

*# Define time indices*

n\_x **=** np**.**arange(len(x)) *# Time indices for x*

n\_h **=** np**.**arange(len(h)) *# Time indices for h*

n\_y **=** np**.**arange(len(y)) *# Time indices for y*

*# Plot original signals and convolution result*

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

*# Plot x[n]*

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

plt**.**stem(n\_x, x, linefmt**=**'b-', markerfmt**=**'bo', basefmt**=**'k')

plt**.**title("Input Signal x[n]")

plt**.**xlabel("n")

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

plt**.**grid()

*# Plot h[n]*

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

plt**.**stem(n\_h, h, linefmt**=**'r-', markerfmt**=**'ro', basefmt**=**'k')

plt**.**title("Filter/Impulse Response h[n]")

plt**.**xlabel("n")

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

plt**.**grid()

*# Plot normalized convolution result y[n]*

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

plt**.**stem(n\_y, y\_norm, linefmt**=**'g-', markerfmt**=**'go', basefmt**=**'k')

plt**.**title("Normalized Convolution Result y[n] = x[n] \* h[n]")

plt**.**xlabel("n")

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

plt**.**grid()

plt**.**tight\_layout()

plt**.**show()

*# Print values*

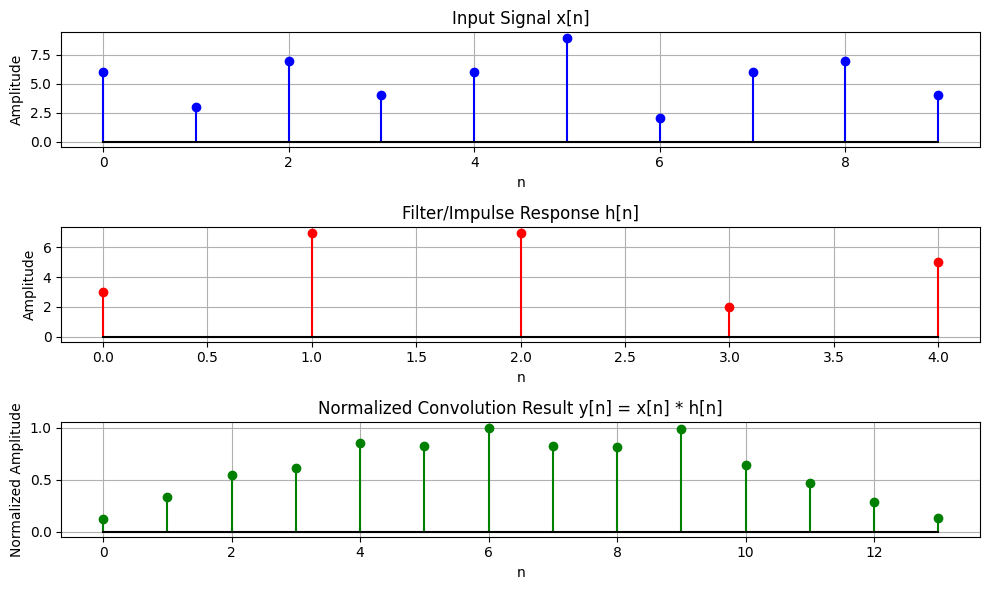
print("Input Signal x[n]:", x)

print("Filter h[n]:", h)

print("Convolution Output y[n]:", y)

print("Normalized Convolution Output y[n]:", y\_norm)

# Output:



Input Signal x[n]: [6 3 7 4 6 9 2 6 7 4]

Filter h[n]: [3 7 7 2 5]

Convolution Output y[n]: [ 18 51 84 94 131 126 154 127 125 152 99 72 43 20]

Normalized Convolution Output y[n]: [0.11688312 0.33116883 0.54545455 0.61038961 0.85064935 0.81818182

1. 0.82467532 0.81168831 0.98701299 0.64285714 0.46753247

0.27922078 0.12987013]

# Problem 05:

## Title: Explain and Implement correlation of signal.

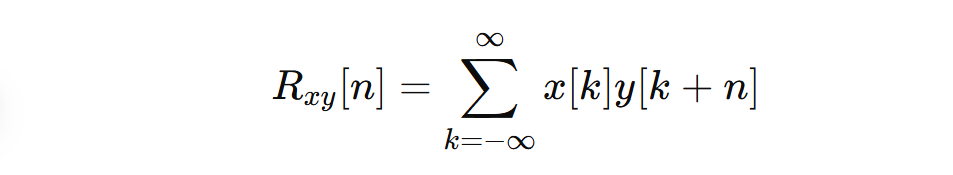
# Objective:

To measure the similarity between two signals using correlation and visualize the **cross-correlation** and **auto-correlation** of two discrete-time signals using Python, and to normalize the results for better interpretation.

# Theory :

**1. Cross-Correlation:**

Cross-correlation measures the similarity between two different signals as one is shifted relative to the other. It helps in detecting patterns and aligning signals. Mathematically, cross-correlation is defined as:

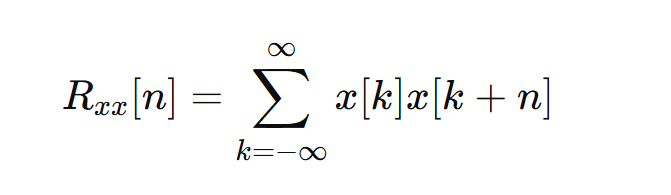


**Applications of Cross-Correlation:**

* **Signal synchronization** in communication systems.
* **Template matching** in image processing.
* **Feature detection** in machine learning.

**2. Auto-Correlation:**

Auto-correlation measures the similarity of a signal with itself over different shifts. It helps in identifying repetitive patterns within a signal. Mathematically, it is given by:



**Applications of Auto-Correlation:**

* **Speech processing:** Identifying periodicity in audio signals.
* **Radar and sonar:** Detecting objects by analyzing echo patterns.
* **DNA sequencing:** Finding repeated gene patterns.

# Purpose:

* To **compare two signals** using cross-correlation.
* To analyze the **repetitive nature** of signals using auto-correlation.
* To **normalize** the correlation values for uniform comparison.
* To **visualize** how signals align at different shifts using plots.

# Source Code:

## # this code is written by APURBO SHARMA (220604)

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

*# Create two signals*

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

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

*# Compute Cross-Correlation (Cross-Correlation: Measures similarity between two different signals like x,y)*

cross\_corr **=** np**.**correlate(x, y, mode**=**'full')

*# Compute Auto-Correlation of x (Auto-Correlation: Measures similarity of a signal with itself )*

auto\_corr\_x **=** np**.**correlate(x, x, mode**=**'full')

*# Compute Auto-Correlation of y*

auto\_corr\_y **=** np**.**correlate(y, y, mode**=**'full')

*# Normalize Cross-Correlation*

norm\_factor **=** np**.**sqrt(np**.**correlate(x, x, mode**=**'full')[len(x) **-** 1] **\*** np**.**correlate(y, y, mode**=**'full')[len(y) **-** 1])

cross\_corr\_norm **=** cross\_corr **/** norm\_factor

*# Normalize Auto-Correlations*

auto\_corr\_x\_norm **=** auto\_corr\_x **/** auto\_corr\_x[len(x) **-** 1] *# Divide by max value*

auto\_corr\_y\_norm **=** auto\_corr\_y **/** auto\_corr\_y[len(y) **-** 1] *# Divide by max value*

*# Plot Cross-Correlation (Shows how x and y align at different shifts)*

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

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

plt**.**stem(range(**-**len(x) **+** 1, len(x)), cross\_corr\_norm)

plt**.**title("Normalized Cross-Correlation of x and y")

plt**.**xlabel("Lag")

plt**.**ylabel("Correlation (Normalized)")

*# Plot Auto-Correlation of x*

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

plt**.**stem(range(**-**len(x) **+** 1, len(x)), auto\_corr\_x\_norm)

plt**.**title("Normalized Auto-Correlation of x")

plt**.**xlabel("Lag")

plt**.**ylabel("Correlation (Normalized)")

*# Plot Auto-Correlation of y*

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

plt**.**stem(range(**-**len(y) **+** 1, len(y)), auto\_corr\_y\_norm)

plt**.**title("Normalized Auto-Correlation of y")

plt**.**xlabel("Lag")

plt**.**ylabel("Correlation (Normalized)")

plt**.**tight\_layout()

plt**.**show()

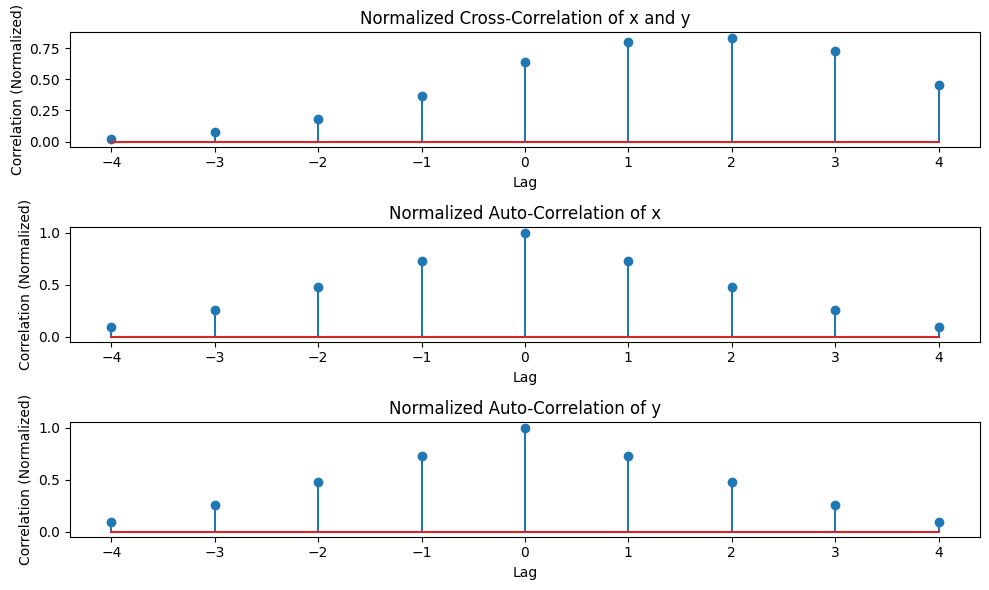
*# Print correlation values*

print("Normalized Cross-Correlation:", cross\_corr\_norm)

print("Normalized Auto-Correlation of x:", auto\_corr\_x\_norm)

print("Normalized Auto-Correlation of y:", auto\_corr\_y\_norm)

# Output:



Normalized Cross-Correlation: [0.01818182 0.07272727 0.18181818 0.36363636 0.63636364 0.8

0.83636364 0.72727273 0.45454545]

Normalized Auto-Correlation of x: [0.09090909 0.25454545 0.47272727 0.72727273 1. 0.72727273

0.47272727 0.25454545 0.09090909]

Normalized Auto-Correlation of y: [0.09090909 0.25454545 0.47272727 0.72727273 1. 0.72727273

0.47272727 0.25454545 0.09090909]

# Problem 06:

## Title: Extract relevant features such as filtering, feature extraction, pick detection, heart rate etc. from PPG signal.

# Objective:

To generate, process, and analyze a **synthetic Photoplethysmogram (PPG) signal** using **Python**, and to detect heart rate and possible abnormalities like **bradycardia, tachycardia, and arrhythmia**.

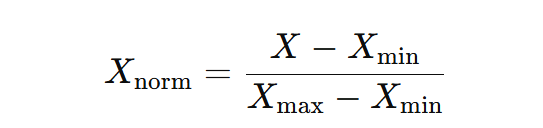
# Theory:

**1. PPG Signal:**

* The **Photoplethysmogram (PPG) signal** is an optical measurement of blood volume changes in tissue.
* It is used to monitor **heart rate (HR), blood oxygen levels (SpO₂), and vascular conditions**.
* The signal consists of a periodic waveform, reflecting heartbeats, and may contain **noise and artifacts**.

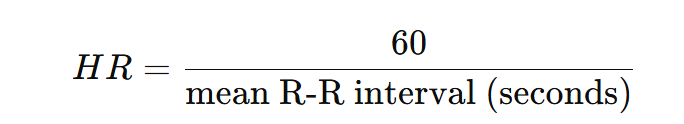
**2. Signal Processing Steps:**

1. **Signal Generation:**
   * A synthetic PPG signal is created using sinusoidal functions with added noise.
   * **Option to introduce abnormalities** such as irregular peaks for arrhythmia simulation.
2. **Normalization:**
   * The signal is normalized using **Min-Max Scaling**:



* + Ensures all values lie between **0 and 1**.

1. **Bandpass Filtering (Noise Removal):**
   * A **Butterworth bandpass filter** is used to retain relevant heart rate frequencies.
   * Removes low-frequency drift and high-frequency noise.
2. **Peak Detection:**
   * Identifies **heartbeats** by detecting peaks in the filtered signal.
   * Uses scipy.signal.find\_peaks() with parameters such as **minimum distance between peaks** and **prominence**.
3. **Feature Extraction:**
   * **R-R Intervals** (time between consecutive heartbeats) are computed.
   * **Heart Rate (HR) is estimated** as:



1. **Abnormality Detection:**
   * **Bradycardia**: HR < 60 BPM (slow heart rate).
   * **Tachycardia**: HR > 100 BPM (fast heart rate).
   * **Arrhythmia**: **Irregular heartbeat** detected if R-R interval standard deviation is high.

**Purpose:**

* To simulate a PPG signal and apply digital signal processing techniques.
* To detect heart rate and abnormalities using peak detection and interval analysis.
* To visualize and interpret PPG data, which is essential for wearable health devices.

# Source Code:

## # this code is written by APURBO SHARMA (220604)

**import** numpy **as** np

**import** scipy.signal **as** signal

**import** matplotlib.pyplot **as** plt

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

*# Step 1:Generate a synthetic PPG signal*

**def** generate\_ppg\_signal(fs**=**100, duration**=**10):

t **=** np**.**linspace(0, duration, fs **\*** duration) *# Time vector*

ppg\_signal **=** 1.5 **\*** np**.**sin(2 **\*** np**.**pi **\*** 1.2 **\*** t) *# Simulated pulse wave*

ppg\_signal **+=** 0.5 **\*** np**.**sin(2 **\*** np**.**pi **\*** 2.5 **\*** t) *# Adding harmonic component*

ppg\_signal **+=** 0.2 **\*** np**.**random**.**randn(len(t)) *# Adding random noise*

**return** t, ppg\_signal

*# Step 2: Normalize the signal*

**def** normalize\_signal(signal):

"""

Normalizes the signal to a range of [0, 1] using Min-Max Scaling.

"""

**return** (signal **-** np**.**min(signal)) **/** (np**.**max(signal) **-** np**.**min(signal))

*# Step 3: Bandpass filtering to remove noise*

**def** bandpass\_filter(data, fs**=**100, lowcut**=**0.5, highcut**=**5.0, order**=**3):

nyquist **=** 0.5 **\*** fs *# Nyquist frequency (half of the sampling rate)*

low **=** lowcut **/** nyquist *# Normalized low cutoff frequency*

high **=** highcut **/** nyquist *# Normalized high cutoff frequency*

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

**return** filtfilt(b, a, data) *# Apply filter using zero-phase filtering*

*# Step 4: Peak Detection*

**def** detect\_peaks(ppg\_signal, fs**=**100):

peaks, \_ **=** find\_peaks(ppg\_signal, distance**=**fs**//**2, prominence**=**0.2) *# Detect peaks*

**return** peaks

*# Step 5: Feature Extraction*

**def** extract\_features(peaks, fs**=**100):

rr\_intervals **=** np**.**diff(peaks) **/** fs *# Compute R-R intervals (time between peaks)*

heart\_rate **=** 60 **/** np**.**mean(rr\_intervals) **if** len(rr\_intervals) **>** 0 **else** 0 *# Compute heart rate*

**return** heart\_rate, rr\_intervals

*# Main execution*

t, raw\_ppg **=** generate\_ppg\_signal() *# Generate synthetic PPG signal*

normalized\_ppg **=** normalize\_signal(raw\_ppg) *# Normalize the signal*

filtered\_ppg **=** bandpass\_filter(normalized\_ppg) *# Apply bandpass filter*

peaks **=** detect\_peaks(filtered\_ppg) *# Detect peaks*

heart\_rate, rr\_intervals **=** extract\_features(peaks) *# Extract heart rate and R-R intervals*

*# Plot the results*

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

plt**.**plot(t, filtered\_ppg, label**=**'Filtered & Normalized PPG Signal')

plt**.**plot(t[peaks], filtered\_ppg[peaks], 'ro', label**=**'Detected Peaks')

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

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

plt**.**title(f'PPG Signal with Detected Peaks (Heart Rate: {heart\_rate:.2f} BPM)')

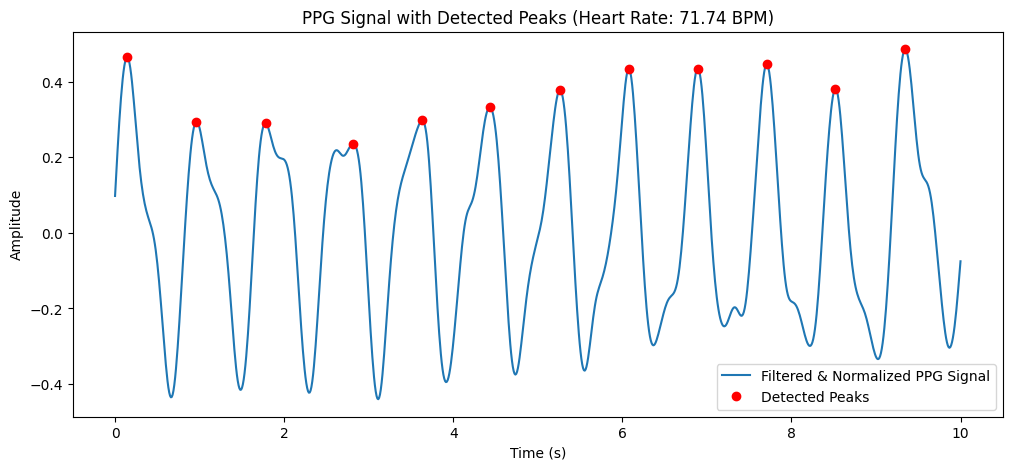
plt**.**legend()

plt**.**show()

# Output:

*# Print estimated heart rate*

print(f'Estimated Heart Rate: {heart\_rate:.2f} BPM')



Estimated Heart Rate: 71.74 BPM

# Problem 07:

## Title: Explain and Implement Discrete Fourier Transform (DFT) using python.

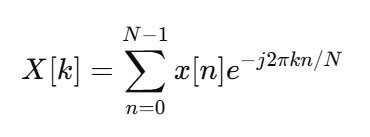
# Objective:

To compute and visualize the **Discrete Fourier Transform (DFT)** of a signal using Python, analyzing its frequency components.

# Theory:

**1. Discrete Fourier Transform (DFT):**

DFT transforms a **time-domain signal** into its **frequency-domain representation**. It is mathematically defined as:

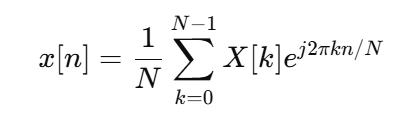


where:

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

**2. Inverse Discrete Fourier Transform (IDFT):**

To recover the time-domain signal from its frequency representation, we use:



DFT is widely used for **signal analysis, filtering, and frequency domain processing**.

**3. Implementation Details:**

* The function DFT(x) computes the **DFT from scratch** using nested loops.
* A signal is generated with two frequency components (70 Hz and 90 Hz).
* The DFT is applied to analyze the frequency spectrum.
* The magnitude spectrum is plotted for interpretation.

# Purpose:

* To **compute the frequency spectrum** of a signal.
* To **visualize how different frequency components** form a signal.
* To **understand how DFT works without relying on fast algorithms (FFT)**.
* To **apply frequency analysis** in DSP applications.

# Source Code:

## # this code is written by APURBO SHARMA (220604)

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

**def** DFT(x):

N **=** len(x)

X **=** np**.**zeros(N, dtype**=**complex)

**for** k **in** range(N):

**for** n **in** range(N):

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

**return** X

*# Fs = input("Sample rate: ")*

Fs **=** 1000

T **=** 1**/**Fs

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

f1, f2 **=** 70, 90

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

plt**.**plot(t,signal) *# Single-sided spectrum*

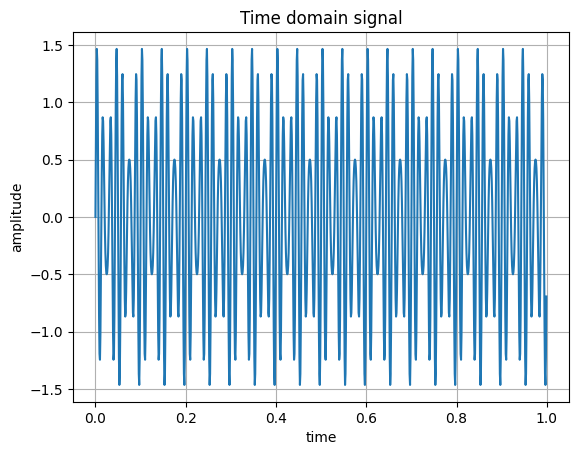
plt**.**title("Time domain signal")

plt**.**xlabel("time")

plt**.**ylabel("amplitude")

plt**.**grid()

plt**.**show()



dft\_output **=** DFT(signal)

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

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

plt**.**plot(freq[:Fs**//**2], np**.**abs(dft\_output[:Fs**//**2]))

plt**.**title("Frequency Spectrum (DFT)")

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

plt**.**ylabel("Magnitude")

plt**.**grid()

plt**.**show()

# Input & Output:

**Input:**

1. **Generated Time-Domain Signal**:
   * **Sampling Frequency (Fs) = 1000 Hz**.
   * **Time Duration = 1 sec**.
   * Two sinusoidal signals:
     + **70 Hz** (primary component).
     + **90 Hz** (weaker component at half amplitude).
2. **DFT Computation** using a manual function (DFT(x)).

**Output:**

1. **Time-Domain Signal Plot:**
   * Displays the sum of two sinusoidal waves.
2. **Frequency Spectrum (DFT Magnitude Plot):**
   * Shows frequency peaks at **70 Hz and 90 Hz**.

# Problem 08:

## Title: Explain and Implement Frequency bin using python.

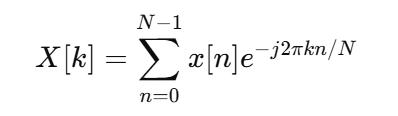
# Objective:

To apply **Fast Fourier Transform (FFT) filtering** to remove high-frequency noise from a **noisy audio signal** and restore the original pure signal.

# Theory:

**1. Fast Fourier Transform (FFT):**

FFT is an efficient algorithm to compute the **Discrete Fourier Transform (DFT)**. It converts a **time-domain signal** into the **frequency domain** using:

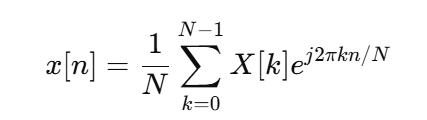


where:

* x[n] is the input signal.
* X[k] represents its frequency components.
* N is the number of samples.

**2. Frequency Filtering in FFT Domain:**

* High-frequency components **above 500 Hz** are removed by setting their magnitude to **zero** in the frequency spectrum.
* The cleaned signal is **reconstructed** using the **Inverse FFT (IFFT)**:



**3. Generating a Noisy Signal:**

* A **pure sinusoidal wave** at **440 Hz** (A4 note in music) is created.
* **Random noise** is added to simulate real-world signal corruption.
* FFT filtering helps **remove noise** and restore the original signal.

# Purpose:

* To **apply FFT for signal processing and noise filtering**.
* To **visualize frequency components and their impact**.
* To **demonstrate real-world applications in audio processing**.

# Source Code:

## # this code is written by APURBO SHARMA (220604)

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

**from** scipy.fft **import** fft, ifft, fftfreq

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*

[0. 0.9765625 1.953125 2.9296875 3.90625 4.8828125 5.859375

6.8359375 7.8125 8.7890625]

*# 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 (back to the time domain)*

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

In [5]:

*# Plot the results*

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

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

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

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**=**"yellow")

plt**.**legend()

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

plt**.**tight\_layout()

plt**.**show()

# Input & Output:

**Input:**

* **Sampling Frequency (Fs) = 1000 Hz**.
* **Signal Frequency = 440 Hz** (pure tone).
* **Random noise added** to simulate distortion.

**Output:**

1. **Original Pure Signal (Time-Domain)** → A clean 440 Hz sine wave.
2. **Noisy Signal (Time-Domain)** → The original signal with added noise.
3. **Filtered Signal (Time-Domain)** → The signal after **FFT-based noise filtering**.

