

1.correlation lab

Theory: Correlation is a statistical measure that describes the degree to which two variables are related. It quantifies the strength and direction of the relationship between variables. In simpler terms, it helps us understand whether and how two variables change in relation to each other.

There are two main types of correlation:

Pearson Correlation:

Definition: The Pearson correlation coefficient (often denoted as

𝑟

r) measures the linear relationship between two variables.

Formula:

𝑟=∑(𝑥𝑖−𝑥ˉ)(𝑦𝑖−𝑦ˉ)∑(𝑥𝑖−𝑥ˉ)2∑(𝑦𝑖−𝑦ˉ)2

r= ∑(x i− xˉ) 2∑(y i− yˉ) 2∑(x i− xˉ)(y i− yˉ)

​

where:

𝑥

𝑖

x

i

​

and

𝑦

𝑖

y

i

​

are individual data points of the variables.

𝑥

ˉ

x

ˉ

and

𝑦

ˉ

y

ˉ

​

are the means of the respective variables.

Range: The Pearson correlation coefficient ranges from -1 to 1:

𝑟

=

1

r=1: Perfect positive correlation (both variables increase or decrease together).

𝑟

=

−

1

r=−1: Perfect negative correlation (one variable increases while the other decreases).

𝑟

=

0

r=0: No linear relationship.

Spearman Rank Correlation:

Definition: The Spearman rank correlation (denoted as

𝜌

ρ or

𝑟

𝑠

r

s

​

) measures the monotonic relationship between two variables, which can be either linear or non-linear. It is based on the ranks of the data points rather than their actual values.

Formula:

𝜌

=

1

−

6

∑

𝑑

𝑖

2

𝑛

(

𝑛

2

−

1

)

ρ=1−

n(n

2

−1)

6∑d

i

2

​

​

where:

𝑑

𝑖

d

i

​

is the difference between the ranks of the corresponding values.

𝑛

n is the number of data points.

Range: Similar to Pearson's correlation, Spearman's correlation also ranges from -1 to 1:

𝜌

=

1

ρ=1: Perfect positive monotonic relationship.

𝜌

=

−

1

ρ=−1: Perfect negative monotonic relationship.

𝜌

=

0

ρ=0: No monotonic relationship.

Objective: 1.Understand and calculate correlation: Learn how to compute correlation coefficients for different types of relationships between variables (Pearson and Spearman).

2.Analyze relationships: Explore the strength and direction of relationships between two variables using real-world or synthetic data.

3.Visualization: Create scatter plots to visually inspect and interpret the correlation between variables.

4.Interpret results: Understand the meaning of correlation coefficients and interpret the degree of relationship between variables.

5.Compare methods: Learn the differences between Pearson and Spearman correlation, and understand when each is appropriate to use.

6.Examine robustness: Understand how correlation methods behave with respect to data # Importing necessary libraries

Source code: # Importing necessary libraries

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from scipy.stats import spearmanr

# Step 1: Creating some sample data

# Let's create two sample datasets, x and y

np.random.seed(42) # for reproducibility

x = np.random.normal(0, 1, 100) # Normally distributed data (mean=0, std=1)

y = 2 \* x + np.random.normal(0, 0.5, 100) # Linear relationship with some noise

# Step 2: Create a pandas DataFrame

data = pd.DataFrame({'X': x, 'Y': y})

# Step 3: Calculate Pearson correlation coefficient

pearson\_corr = data.corr(method='pearson')

print("Pearson Correlation Coefficient:")

print(pearson\_corr)

# Step 4: Calculate Spearman rank correlation coefficient

spearman\_corr, \_ = spearmanr(x, y)

print(f"Spearman Rank Correlation Coefficient: {spearman\_corr}")

# Step 5: Visualize the data using a scatter plot

plt.figure(figsize=(8, 6))

sns.scatterplot(x='X', y='Y', data=data, color='blue', alpha=0.7)

plt.title("Scatter Plot of X vs Y", fontsize=15)

plt.xlabel("X")

plt.ylabel("Y")

plt.show()

# Step 6: Interpret the results

# In this case, Pearson correlation will show a positive correlation (due to the linear relationship).

# Spearman correlation will also show a positive correlation, but it's less sensitive to the exact linearity.

2.implement in convolution .

Theory: Convolution is a fundamental operation used in various fields, particularly in image processing and neural networks. It is used to combine two sets of information. In the context of image processing, convolution typically refers to the operation of applying a filter (also known as a kernel) to an image, where each pixel in the output image is computed by multiplying corresponding values in the filter and a section of the input image, then summing the results.

In this lab, we will:

Understand convolution and its use in image processing.

Implement a simple 2D convolution operation.

Visualize the effect of convolution with different filters (e.g., edge detection, blurring).

Experiment with kernel applications on images.

Objective: Understand convolution: Learn how convolution is used to apply filters to images for tasks such as edge detection, blurring, and sharpening.

Implement convolution: Code a 2D convolution operation and apply it to an image.

Apply different filters: Use various filters (kernels) and observe their effect on an image.

Visualize the results: Show how convolution alters the image with the chosen filter.

Source code: # Import necessary libraries

import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import convolve2d

import cv2

# Step 1: Read and display the original image

image = cv2.imread('your\_image.jpg', cv2.IMREAD\_GRAYSCALE) # Load in grayscale

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

plt.subplot(1, 2, 1)

plt.imshow(image, cmap='gray')

plt.title("Original Image")

plt.axis('off')

# Step 2: Define Convolution Filters (Kernels)

# Identity Filter (no change to image)

identity\_filter = np.array([[0, 0, 0],

[0, 1, 0],

[0, 0, 0]])

# Edge Detection Filter (Sobel Filter)

edge\_filter = np.array([[-1, 0, 1],

[-2, 0, 2],

[-1, 0, 1]])

# Gaussian Blur Filter

blur\_filter = np.array([[1, 2, 1],

[2, 4, 2],

[1, 2, 1]]) / 16 # Normalizing to ensure sum is 1

# Sharpening Filter

sharpen\_filter = np.array([[0, -1, 0],

[-1, 5, -1],

[0, -1, 0]])

# Step 3: Apply Convolution

identity\_image = convolve2d(image, identity\_filter, mode='same', boundary='wrap')

edge\_image = convolve2d(image, edge\_filter, mode='same', boundary='wrap')

blur\_image = convolve2d(image, blur\_filter, mode='same', boundary='wrap')

sharpen\_image = convolve2d(image, sharpen\_filter, mode='same', boundary='wrap')

# Step 4: Display the results of convolution

plt.subplot(1, 2, 2)

plt.imshow(edge\_image, cmap='gray') # Change to visualize other results

plt.title("Edge Detection (Sobel)")

plt.axis('off')

plt.show()

# Additional steps could include saving the processed images or further analysis.

3.implementing DFT in python.

Theory: The Discrete Fourier Transform (DFT) is a mathematical technique used to transform a sequence of complex numbers (discrete-time signal) from the time domain to the frequency domain. It is the basis of many signal processing algorithms and is commonly used in audio, image processing, and communication systems. The DFT is used to analyze the frequency content of a discrete signal.

Formula of DFT:

Given a sequence of N complex numbers:

𝑥

0

,

𝑥

1

,

𝑥

2

,

…

,

𝑥

𝑁

−

1

x

0

​

,x

1

​

,x

2

​

,…,x

N−1

​

The DFT of this sequence,

𝑋

𝑘

X

k

​

, is given by the formula:

𝑋

𝑘

=

∑

𝑛

=

0

𝑁

−

1

𝑥

𝑛

⋅

𝑒

−

𝑖

2

𝜋

𝑘

𝑛

𝑁

for

𝑘

=

0

,

1

,

2

,

…

,

𝑁−1Xk=

n=0∑N−1x n⋅e −i2π Nkn

fork=0,1,2,…,N−1

objective: 1.Understand the mathematical foundation: The DFT is a fundamental concept in signal processing, and implementing it from scratch helps in understanding how frequency-domain transformations work.

2.Real-time frequency analysis: The DFT is used in real-time signal analysis. Implementing it in Python allows you to analyze discrete signals, such as audio signals, images, or any time-domain data, in the frequency domain.

3.Signal Processing Application: Many applications like noise reduction, filtering, and compression rely on DFT. Implementing it helps you develop these processing algorithms.

4.Optimization understanding: Understanding the inefficiency of a direct DFT calculation (which has O(N^2) complexity) and exploring optimizations like Fast Fourier Transform (FFT) for faster computation.

Source code: Implementing DFT in 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()

Computing Frequency Bins in Python

Compute Frequency Bins Manually

import numpy as np

# Define parameters

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

Instead of computing manually, NumPy provides a built-in function np.fft.fftfreq(), which calculates the bins directly.

python

CopyEdit

import numpy as np

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

We'll generate a noisy audio signal, apply FFT, filter out high-frequency noise, and reconstruct the cleaned signal using Inverse FFT.

Steps:

1. Generate an audio signal (a pure sine wave of 440 Hz).

2. Add random noise.

3. Apply FFT to transform the signal to the frequency domain.

4. Remove noise by filtering high frequencies.

5. Apply Inverse FFT to get back the cleaned signal.

import numpy as np

import matplotlib.pyplot as plt

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()

4.implement fourier seriese decomposition.

Theory: The Fourier Series is a way to represent a periodic function as the sum of sines and cosines. In mathematical terms, any periodic function

𝑓

(

𝑡

)

f(t) with a period

𝑇

T can be written as a sum of sines and cosines:

𝑓(𝑡)=𝑎0+∑𝑛=1∞(𝑎𝑛cos(2𝜋𝑛𝑡𝑇)+𝑏𝑛sin⁡(2𝜋𝑛𝑡𝑇))f(t)=a 0+ n=1∑∞(a ncos( T2πnt)+b nsin( T2πnt))

Where:0a 0

is the average value of the function over one period.

𝑎

𝑛

a

n

​

and

𝑏

𝑛

b

n

​

are the Fourier coefficients, which represent the amplitudes of the cosine and sine waves respectively.

Objectives: The goal is to:

Decompose a given periodic function into a sum of sines and cosines (Fourier series).

Plot the original function and its Fourier series approximation using Python.

Source code: import numpy as np

import matplotlib.pyplot as plt

# Function to approximate (square wave)

def square\_wave(t):

return np.sign(np.sin(t))

# Fourier Series Decomposition

def fourier\_series\_approximation(t, N\_terms, T):

a\_0 = 0 # a\_0 is 0 for odd functions like square wave

result = a\_0 / 2 # Starting with the constant term a\_0 / 2

# Add terms for n = 1 to N\_terms

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

# Only odd harmonics contribute for a square wave

if n % 2 == 1:

a\_n = 0 # No cosine terms for square wave (as it's an odd function)

b\_n = 4 / (np.pi \* n) # b\_n for a square wave

# Fourier series component (sine terms only)

result += b\_n \* np.sin(2 \* np.pi \* n \* t / T)

return result

# Parameters

T = 2 \* np.pi # Period of the square wave

t = np.linspace(0, 2\*T, 1000) # Time range (two periods)

N\_terms = 10 # Number of Fourier series terms to consider

# Original square wave

original\_signal = square\_wave(t)

# Fourier Series Approximation (using N\_terms terms)

approx\_signal = fourier\_series\_approximation(t, N\_terms, T)

# Plot the original square wave and its Fourier series approximation

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

# Plot Original Square Wave

plt.subplot(2, 1, 1)

plt.plot(t, original\_signal, label='Original Square Wave', color='b')

plt.title('Original Square Wave')

plt.xlabel('Time')

plt.ylabel('Amplitude')

plt.grid(True)

# Plot Fourier Series Approximation

plt.subplot(2, 1, 2)

plt.plot(t, approx\_signal, label=f'Fourier Series Approximation (N={N\_terms})', color='r')

plt.title(f'Fourier Series Approximation (N={N\_terms})')

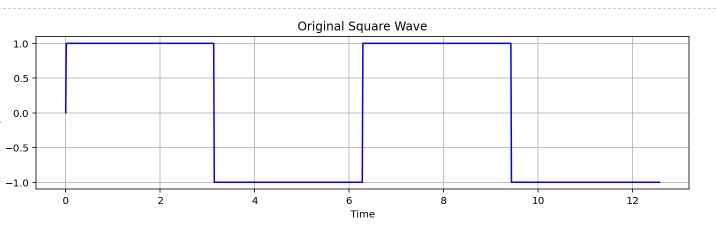
plt.xlabel('Time')

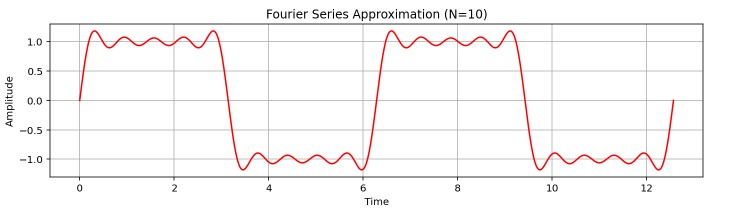
plt.ylabel('Amplitude')

plt.grid(True)

plt.tight\_layout()

plt.show()





5.implement fourier transform?

Theory: The Fourier Transform is a mathematical transformation that converts a function (usually a time-domain signal) into a frequency-domain representation. The Continuous Fourier Transform (CFT) is used for continuous signals, but in practical scenarios with discrete data, we usually apply the Discrete Fourier Transform (DFT).

In Python, the Fourier Transform can be computed using NumPy's fft (Fast Fourier Transform) module, which efficiently calculates the DFT. Here, we will implement both the Discrete Fourier Transform (DFT) from scratch and use NumPy's FFT for comparison.

Objective: To compute the Fourier Transform of a given signal using Python.

To visualize the time-domain signal and its frequency-domain representation.

To understand how the Fourier Transform converts time-domain data into frequency-domain data.

Source code: import numpy as np

import matplotlib.pyplot as plt

# Function to compute the DFT (Discrete Fourier Transform)

def dft(x):

N = len(x)

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

for k in range(N):

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

return X

# Create a sample signal (sum of two sine waves)

N = 256 # Number of samples

t = np.linspace(0, 1, N, endpoint=False) # Time array

f1, f2 = 5, 50 # Frequencies of the sine waves in Hz

x = np.sin(2 \* np.pi \* f1 \* t) + np.sin(2 \* np.pi \* f2 \* t) # Combined signal

# Compute the DFT using our custom function

X\_custom = dft(x)

# Compute the DFT using NumPy's FFT (Fast Fourier Transform)

X\_np = np.fft.fft(x)

# Frequency axis for plotting (in Hz)

frequencies = np.fft.fftfreq(N, t[1] - t[0])

# Plot the original signal in the time domain

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

# Plot Time-Domain Signal

plt.subplot(2, 1, 1)

plt.plot(t, x, label="Time-domain Signal")

plt.title("Time-Domain Signal")

plt.xlabel("Time [s]")

plt.ylabel("Amplitude")

plt.grid(True)

# Plot Frequency-Domain Representation

plt.subplot(2, 1, 2)

plt.plot(frequencies[:N//2], np.abs(X\_np)[:N//2], label="Frequency-domain (FFT)", color='r')

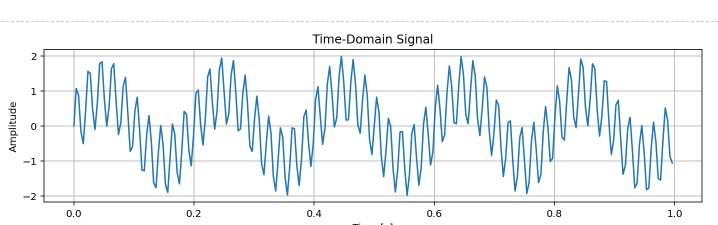
plt.title("Frequency-Domain Representation (Magnitude of DFT)")

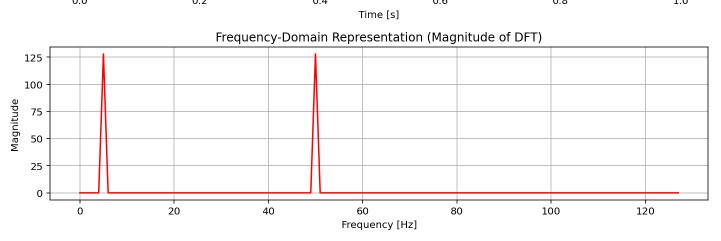
plt.xlabel("Frequency [Hz]")

plt.ylabel("Magnitude")

plt.grid(True)

plt.tight\_layout()

plt.show()



6.detecting heardbeat from ppg signal peacks.

Theory: 1. Preprocessing the PPG Signal:

Filtering:

Apply a low-pass filter to remove high-frequency noise (such as motion artifacts) and a high-pass filter to remove baseline drift.

A typical band-pass filter might be in the range of 0.5 Hz to 5 Hz to focus on the heart rate.

Normalization:

Normalize the signal to ensure that it has a consistent amplitude scale, which can help with peak detection.

2. Peak Detection:

Peak Finding:

Identify the local maxima in the filtered and preprocessed PPG signal. These peaks correspond to the heartbeats.

Use an algorithm like the find\_peaks function in Python (from the scipy.signal library) or other peak detection algorithms to find these peaks.

Example in Python:

python

Copy

from scipy.signal import find\_peaks

# Assume `ppg\_signal` is your preprocessed PPG signal

peaks, \_ = find\_peaks(ppg\_signal, height=threshold, distance=min\_distance)

Threshold: You may need to set a threshold for peak height to eliminate small peaks that are noise.

Minimum Distance: Set a minimum distance between peaks, which is often related to the expected heart rate (e.g., if the heart rate is around 60-100 bpm, you might set a minimum distance corresponding to the minimum time between two beats).

3. Heartbeat Detection and Analysis:

Heart Rate Calculation: Once the peaks are detected, you can calculate the heart rate by measuring the time between successive peaks (i.e., the R-R intervals, if you consider the PPG signal analogous to the ECG signal).

Convert this interval into beats per minute (bpm) by using the formula:

Heart Rate (bpm)

=

60

Time Between Peaks (in seconds)

Heart Rate (bpm)=

Time Between Peaks (in seconds)

60

​

Artifact Removal: In some cases, noise or motion artifacts may cause false peaks. You can refine the peak detection by considering:

Peak amplitude and rejecting smaller, noise-related peaks.

Waveform characteristics, such as the shape of the PPG signal, which should be considered when validating the peaks.

4. Post-Processing (Optional):

Heartbeat Interval Smoothing: Sometimes, smoothing the detected intervals between successive peaks helps to improve the overall stability of the heart rate estimation. Methods like moving average or exponential smoothing can be applied to reduce irregularities due to noise.

Heart Rate Variability: If you need to measure heart rate variability, you can analyze the variability in the time intervals between detected peaks.

Source code: import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import find\_peaks

# Simulated PPG signal (replace with real data)

time = np.linspace(0, 10, 1000) # 10 seconds of data

ppg\_signal = np.sin(2 \* np.pi \* 1.2 \* time) + 0.1 \* np.random.randn(len(time)) # Simulated signal with noise

# Preprocessing: Apply bandpass filter (example)

from scipy.signal import butter, filtfilt

# Bandpass filter (0.5 - 5 Hz) for PPG signal

lowcut = 0.5

highcut = 5.0

fs = 100 # Sampling frequency (Hz)

def butter\_bandpass(lowcut, highcut, fs, order=5):

nyquist = 0.5 \* fs

low = lowcut / nyquist

high = highcut / nyquist

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

return b, a

def butter\_filter(data, lowcut, highcut, fs, order=5):

b, a = butter\_bandpass(lowcut, highcut, fs, order)

return filtfilt(b, a, data)

# Apply filter

filtered\_ppg = butter\_filter(ppg\_signal, lowcut, highcut, fs)

# Peak detection

peaks, \_ = find\_peaks(filtered\_ppg, height=0.1, distance=30)

# Plot PPG signal with detected peaks

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

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

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

plt.xlabel('Time (seconds)')

plt.ylabel('Amplitude')

plt.legend()

plt.title('PPG Signal with Detected Heartbeat Peaks')

plt.show()

# Calculate heart rate (bpm)

peak\_times = time[peaks]

rr\_intervals = np.diff(peak\_times) # R-R intervals in seconds

heart\_rate = 60 / rr\_intervals # Convert to beats per minute

# Display heart rate

print("Heart Rate (bpm):", heart\_rate)

import numpy as np

import matplotlib.pyplot as plt

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

from scipy.fft import fft

from scipy.stats import skew, kurtosis

# Simulate a clean PPG signal (for demonstration)

time = np.linspace(0, 10, 1000) # 10 seconds of data

clean\_ppg\_signal = np.sin(2 \* np.pi \* 1.2 \* time) # Simulated PPG signal (1.2 Hz heart rate)

# Bandpass filter parameters (0.5 - 5 Hz)

lowcut = 0.5

highcut = 5.0

fs = 100 # Sampling frequency (Hz)

# Butterworth bandpass filter

def butter\_bandpass(lowcut, highcut, fs, order=5):

nyquist = 0.5 \* fs

low = lowcut / nyquist

high = highcut / nyquist

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

return b, a

def butter\_filter(data, lowcut, highcut, fs, order=5):

b, a = butter\_bandpass(lowcut, highcut, fs, order)

return filtfilt(b, a, data)

# Apply the bandpass filter to the clean signal

filtered\_ppg\_signal = butter\_filter(clean\_ppg\_signal, lowcut, highcut, fs)

# Peak detection (find heartbeats)

peaks, \_ = find\_peaks(filtered\_ppg\_signal, height=0.1, distance=30)

# Calculate Time-Domain Features

rr\_intervals = np.diff(time[peaks]) # R-R intervals in seconds

mean\_heart\_rate = 60 / np.mean(rr\_intervals) # Mean Heart Rate (bpm)

hrv = np.std(rr\_intervals) # Heart Rate Variability (SDNN)

# Frequency-Domain Features (using FFT)

n = len(filtered\_ppg\_signal)

frequencies = np.fft.fftfreq(n, 1/fs)

fft\_values = np.abs(fft(filtered\_ppg\_signal))

# Extract the heart rate frequency (dominant frequency)

dominant\_frequency = frequencies[np.argmax(fft\_values[1:n//2])] # Exclude DC component

# Morphological Features (Peak Amplitude)

peak\_amplitudes = filtered\_ppg\_signal[peaks]

mean\_peak\_amplitude = np.mean(peak\_amplitudes)

# Statistical Features (Skewness and Kurtosis)

signal\_skewness = skew(filtered\_ppg\_signal)

signal\_kurtosis = kurtosis(filtered\_ppg\_signal)

# Plot the filtered signal and detected peaks

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

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

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

plt.xlabel('Time (seconds)')

plt.ylabel('Amplitude')

plt.legend()

plt.title('Filtered PPG Signal with Detected Heartbeat Peaks')

plt.show()

# Display Extracted Features

print("Mean Heart Rate (bpm):", mean\_heart\_rate)

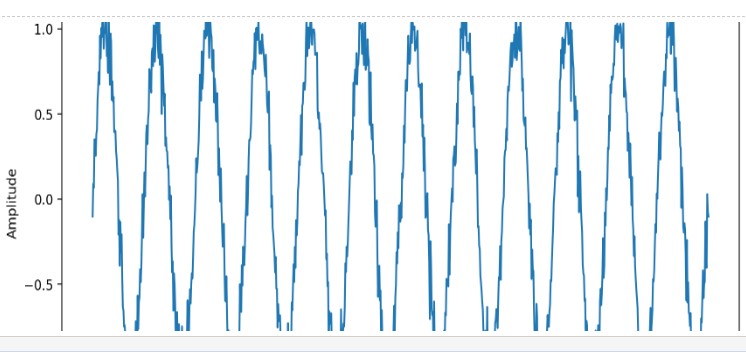
print("Heart Rate Variability (SDNN):", hrv)

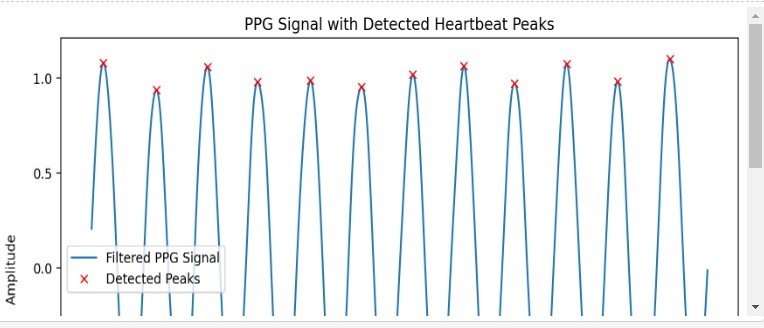
print("Dominant Frequency (Hz):", dominant\_frequency)

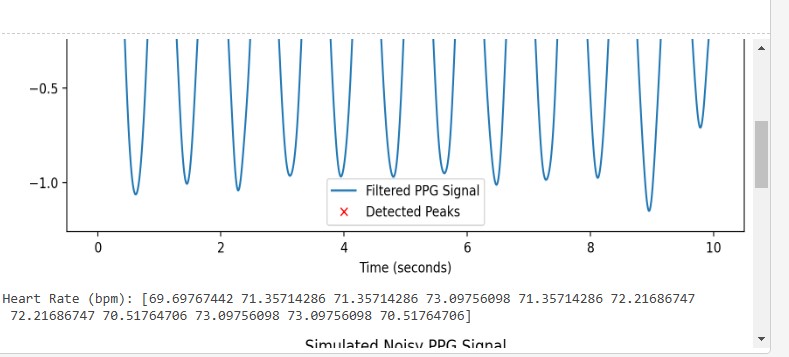
print("Mean Peak Amplitude:", mean\_peak\_amplitude)

print("Signal Skewness:", signal\_skewness)

print("Signal Kurtosis:", signal\_kurtosis)







7.signal operation:add,shifting,folding

Theory: Addition:

Adding two signals corresponds to adding the amplitudes at each time point. This can be useful, for example, when you want to combine two signals or if you have multiple channels (e.g., PPG from two different sensors) and wish to combine them.

Shifting:

Shifting a signal involves displacing the signal along the time axis. This can be done by shifting the entire signal left or right by a given amount, typically in terms of time or samples.

For example, shifting the signal to the right (delaying it) by a number of samples corresponds to introducing a time delay.

Folding (Flipping):

Folding a signal involves reversing its time axis. This is equivalent to flipping the signal around a specified point (e.g., the center of the signal or the origin).

Source code: import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import butter, filtfilt

# Simulate a clean PPG signal (for demonstration)

time = np.linspace(0, 10, 1000) # 10 seconds of data

ppg\_signal = np.sin(2 \* np.pi \* 1.2 \* time) # Simulated PPG signal (1.2 Hz heart rate)

# Signal addition: Adding two PPG signals (with different frequencies for demonstration)

ppg\_signal2 = np.sin(2 \* np.pi \* 1.5 \* time) # Second PPG signal (1.5 Hz)

added\_signal = ppg\_signal + ppg\_signal2

# Signal shifting: Shift the signal by 200 samples (to the right)

shift\_amount = 200 # Shift in samples

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

# Signal folding (flipping): Flip the signal around the center (time axis)

folded\_signal = np.flip(ppg\_signal)

# Plot the original, added, shifted, and folded signals

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

plt.subplot(2, 2, 1)

plt.plot(time, ppg\_signal)

plt.title("Original PPG Signal")

plt.subplot(2, 2, 2)

plt.plot(time, added\_signal)

plt.title("Added Signal (Original + Second PPG)")

plt.subplot(2, 2, 3)

plt.plot(time, shifted\_signal)

plt.title("Shifted Signal (Right by 200 samples)")

plt.subplot(2, 2, 4)

plt.plot(time, folded\_signal)

plt.title("Folded Signal (Flipped)")

plt.tight\_layout()

plt.show()