

Pabna University of Science and

TECHNOlOGY

**INFORMATION AND COMMUNICATION ENGINEERING**

### Course Name: Signals and Systems Sessional

Course Code: ICE-2204

Submitted To,

dr.Md.Imran hossain

Associate PROFESSOR DEpARTMENT OF ICE, PUST

Submitted By, MD.MURSAlIN LEMON Roll: 220606 SESSION:2021-22

2nd Year 2nd Semester

DEpARTWENT OF ICE, PUST

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

**Lab** **Report** **on** **Signal** **Operations**

**Theory**

Signal processing 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)

1. Signal Multiplication: Multiplying two signals results in a combined signal with modulated characteristics, often used in amplitude modulation.

#### y(t) = x1(t) \* x2(t)

1. 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)**

1. 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.

1. 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)}**

## Source Code

import numpy as np

import matplotlib.pyplot as plt

# Parameters

t = np.arange(-10, 10, 0.01) # Time vector # Define two signals

x1 = np.sin(2 \* np.pi \* 1 \* t) # Signal 1: Sine wave

x2 = np.cos(2 \* np.pi \* 0.5 \* t) # Signal 2: Cosine wave # Signal Addition

y\_add = x1 + x2

# Signal Multiplication y\_mult = x1 \* x2

# Amplitude Scaling

k = 2 # Scaling factor y\_scaled = k \* x1

# Time Shifting

shift = 2 # Shift value (in seconds)

y\_shifted = np.sin(2 \* np.pi \* 1 \* (t - shift)) # Delayed signal # Signal Folding

y\_folded = np.sin(2 \* np.pi \* 1 \* (-t)) # Folded sine wave # Plot Results

plt.figure(figsize=(12, 15)) # Original Signals plt.subplot(6, 1, 1)

plt.plot(t, x1, label='x1: Sine Wave', color='b') plt.plot(t, x2, label='x2: Cosine Wave', color='r') plt.title('Original Signals')

plt.legend()

plt.grid()

# Signal Addition plt.subplot(6, 1, 2) plt.plot(t, y\_add, color='m') plt.title('Signal Addition') plt.grid()

# Signal Multiplication plt.subplot(6, 1, 3) plt.plot(t, y\_mult, color='k')

plt.title('Signal Multiplication') plt.grid()

# Amplitude Scaling plt.subplot(6, 1, 4)

plt.plot(t, y\_scaled, color='g') plt.title('Amplitude Scaling') plt.grid()

# Time Shifting plt.subplot(6, 1, 5)

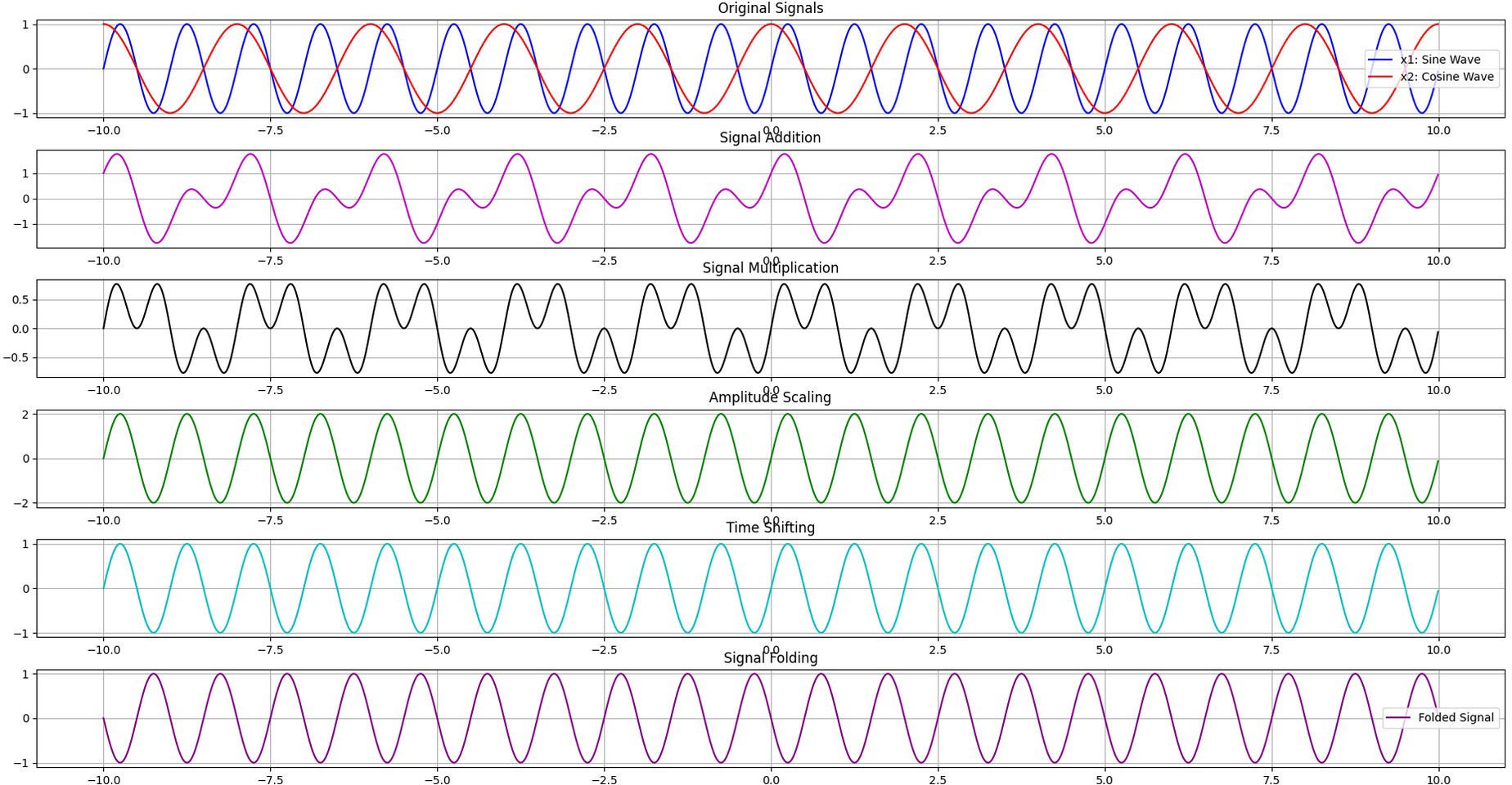
plt.plot(t, y\_shifted, color='c') plt.title('Time Shifting') plt.grid()

# Signal Folding plt.subplot(6, 1, 6)

plt.plot(t, y\_folded, label='Folded Signal', color='purple') plt.title('Signal Folding')

plt.legend() plt.grid()

# Display the plots plt.tight\_layout() plt.show()

**Output**

# Lab Report on Signal Sequence

## Theory

Signal sequences are fundamental building blocks in signal processing and system analysis. They are used to represent basic signals that can be combined or manipulated to model more complex systems. In this lab, we focus on three key signal sequences:

1. Impulse Signal:
   * The impulse signal, also known as the Dirac delta function, is a theoretical signal that is zero everywhere except at *t*=0, where it is infinitely high. In practice, it is approximated as a narrow pulse with unit area.
   * Mathematically, the impulse signal is defined as:

***δ*(*t*)=**{**∞** **if** **t=0**

0 otherwise

* + The impulse signal is used to analyze the response of systems (impulse response) and is a key concept in convolution and filtering.

1. Step Signal:
   * The step signal, also known as the unit step function, represents a signal that transitions from 0 to 1 at *t*=0.
   * Mathematically, the step signal is defined as:

1 **if** *t*≥0

{

***u(t*)=**

0 otherwise

* + The step signal is used to model systems that switch on at a specific time, such as turning on a device or applying a constant input.

1. Ramp Signal:
   * The ramp signal represents a signal that increases linearly with time for *t*≥0.
   * Mathematically, the ramp signal is defined as:

***r(t*)=**{t **if** *t*≥0

0 otherwise

* + The ramp signal is used to model systems with linearly increasing inputs, such as velocity or acceleration.

## Source Code

import numpy as np

import matplotlib.pyplot as plt

# Define the range for continuous-time signals

t = np.linspace(-10, 10, 1000) # Smooth time values

# Define continuous-time signals def impulse\_signal(t):

return np.where(np.abs(t) < 0.1, 1, 0) # Approximate impulse with a narrow pulse

def step\_signal(t):

return np.where(t >= 0, 1, 0)

def ramp\_signal(t):

return np.where(t >= 0, t, 0)

# Generate signals

impulse = impulse\_signal(t) step = step\_signal(t)

ramp = ramp\_signal(t)

# Plot signals plt.figure(figsize=(12, 4))

# Impulse Signal

plt.subplot(1, 3, 1)

plt.plot(t, impulse, 'r', linewidth=2) plt.title("Impulse Signal (Approximate)") plt.xlabel("t")

plt.ylabel("Amplitude") plt.grid()

# Step Signal plt.subplot(1, 3, 2)

plt.plot(t, step, 'g', linewidth=2) plt.title("Step Signal") plt.xlabel("t") plt.ylabel("Amplitude") plt.grid()

# Ramp Signal plt.subplot(1, 3, 3)

plt.plot(t, ramp, 'b', linewidth=2) plt.title("Ramp Signal") plt.xlabel("t") plt.ylabel("Amplitude")

plt.grid()

plt.tight\_layout() plt.show()

**Output**

# Lab Report on Convolution

## Theory

Convolution is a mathematical operation that combines two signals to produce a third signal, which represents how the shape of one signal is modified by the other. It is a fundamental concept in signal processing, used in applications such as filtering, system analysis, and image processing.

The convolution of two continuous-time signals *x*(*t*) and *h*(*t*) is defined as:

***y*(*t*)=*x*(*t*)**∗***h*(*t*)=−∞∫∞*x*(*τ*)*h*(*t*−*τ*)*dτ***

For discrete-time signals, the convolution is given by:

***y*[*n*]=*x*[*n*]**∗***h*[*n*]=k=−∞∑∞*x*[*k*]*h*[*n*−*k*]**

Some properties of convolution:

1. Linear Time-Invariant (LTI) Systems: Convolution is used to describe the output of an LTI system when the input signal and the system's impulse response are known.
2. Commutative Property: Convolution is commutative, meaning *x*(*t*)∗*h*(*t*)=*h*(*t*)∗*x*(*t*).
3. Applications: Convolution is used in filtering (e.g., low-pass, high-pass filters), audio processing, and image blurring or sharpening.

By performing convolution, we can analyze how a system responds to different inputs, design filters, and understand the interaction between signals and systems.

## Source code

import numpy as np

import matplotlib.pyplot as plt

# Define two real signals: sine and cosine waves fs = 1000 # Sampling frequency in Hz

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

# Define the signals

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

# Perform convolution

conv\_result = np.convolve(sine\_wave, cosine\_wave, mode='full') # Display and plot the results

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

# Original signals plt.subplot(3, 1, 1)

plt.plot(t, sine\_wave, label="Sine Wave") plt.plot(t, cosine\_wave, label="Cosine Wave") plt.title("Original Signals")

plt.legend() plt.grid()

# Convolution result

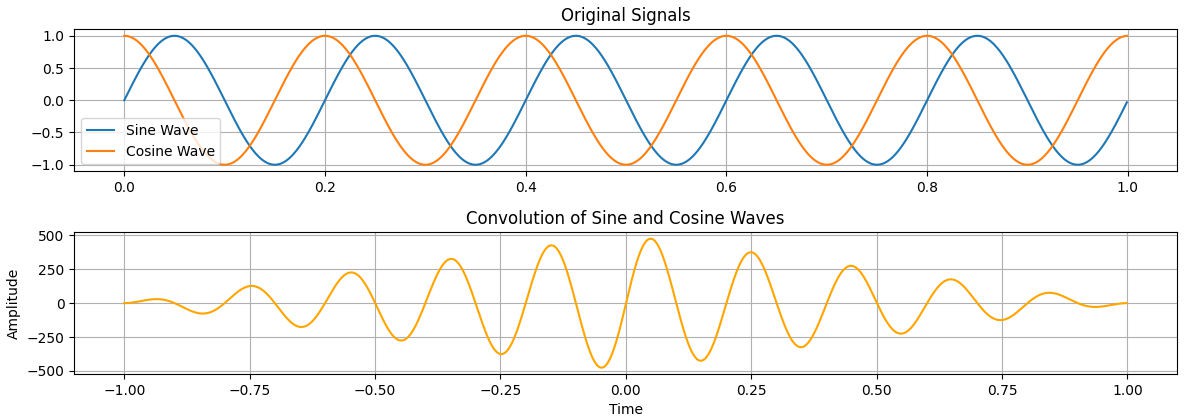
conv\_time = np.linspace(-1, 1, len(conv\_result), endpoint=False) plt.subplot(3, 1, 2)

plt.plot(conv\_time, conv\_result, label="Convolution", color='orange') plt.title("Convolution of Sine and Cosine Waves")

plt.xlabel("Time") plt.ylabel("Amplitude") plt.grid()

plt.tight\_layout() plt.show()

**Output**



# Lab Report on Correlation

## Theory

Correlation is a measure of similarity between two signals, often used to detect patterns or align signals in time. It is widely used in applications such as radar, sonar, and pattern recognition.

The cross-correlation of two continuous-time signals *x*(*t*) and *y*(*t*) is defined as:

***Rxy*(*τ*)=**−∞∫∞***x*(*t*)*y*(*t*+*τ*)*dt***

For discrete-time signals, the cross-correlation is given by:

***Rxy*[*m*]=*n=***−∞∑∞***x*[*n*]*y*[*n*+*m*]**

Some properties of correlation:

1. Autocorrelation: When a signal is correlated with itself, it is called autocorrelation. It measures the similarity of a signal with a time-shifted version of itself.
2. Applications: Correlation is used in signal detection, time-delay estimation, and pattern matching.
3. Peak Detection: The peak of the correlation function indicates the time shift where the signals are most similar.

By using correlation, we can determine the degree of similarity between signals, detect delays, and identify patterns in noisy environments.

## Source code

import numpy as np

import matplotlib.pyplot as plt

# Define two real signals: sine waves with different frequencies fs = 1000 # Sampling frequency in Hz

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

# Define the sine waves with different frequencies sine\_wave1 = np.sin(2 \* np.pi \* 5 \* t) # 5 Hz sine wave

sine\_wave2 = np.sin(2 \* np.pi \* 5 \* t + np.pi / 4) # 5 Hz sine wave with a phase shift of 45 degrees

# Display and plot the results plt.figure(figsize=(12, 8))

# Original signals

plt.subplot(3, 1, 1)

plt.plot(t, sine\_wave1, label="Sine Wave 1 (5 Hz)")

plt.plot(t, sine\_wave2, label="Sine Wave 2 (5 Hz with 45° phase shift)") plt.title("Original Signals (Sine Waves with Phase Shift)") plt.xlabel("Time (s)")

plt.ylabel("Amplitude") plt.legend()

plt.grid()

# Perform cross-correlation

correlation = np.correlate(sine\_wave1, sine\_wave2, mode='full')

lags = np.arange(-len(sine\_wave1) + 1, len(sine\_wave1)) # Compute lag indices

# Cross-correlation plt.subplot(3, 1, 2)

plt.plot(lags, correlation, label="Cross-Correlation", color='green') plt.title("Cross-Correlation of Sine Waves")

plt.xlabel("Lag") plt.ylabel("Correlation") plt.legend()

plt.grid()

# Perform auto-correlation

auto\_correlation = np.correlate(sine\_wave1, sine\_wave1, mode='full')

# Auto-correlation plt.subplot(3, 1, 3)

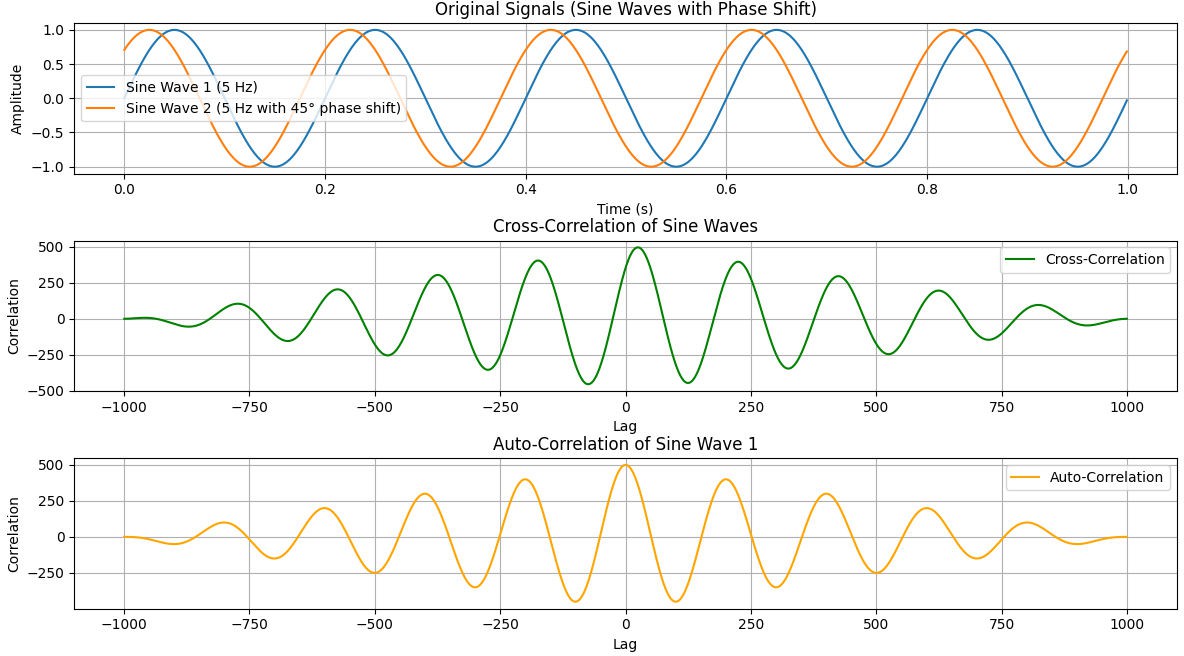
plt.plot(lags, auto\_correlation, label="Auto-Correlation", color='orange') plt.title("Auto-Correlation of Sine Wave 1")

plt.xlabel("Lag") plt.ylabel("Correlation") plt.legend()

plt.grid()

plt.tight\_layout() plt.show()

**Output**

****

# Lab report on Feature Extraction of a PPG (Photoplethysmogram) Signal

**Theory:**

### Photoplethysmography (PPG) is a simple and non-invasive technique used to measure changes in blood volume in the body. It works by shining light onto the skin and detecting variations in the reflected or transmitted light, which correspond to blood flow changes. PPG signals are widely used in heart rate monitoring, oxygen saturation measurement, and cardiovascular health analysis.

Since the PPG signal contains valuable physiological information, we use **feature** **extraction** techniques to analyze it effectively. These techniques help us identify key characteristics such as heart rate, blood circulation patterns, and heart rate variability.

#### Understanding the PPG Signal

The PPG signal consists of repeating waveforms that represent heartbeats. Each cycle of the signal contains:

* + **A** **Systolic** **Peak** **(Highest** **Point):** This corresponds to the maximum blood volume in the arteries when the heart contracts.
  + **A** **Diastolic** **Point** **(Lowest** **Point):** This occurs when the blood volume is at its lowest before the next heartbeat.

The frequency of these peaks is directly related to heart rate, and their shape provides insights into blood circulation. However, the raw PPG signal can be noisy due to motion artifacts, ambient light, and sensor imperfections.

#### Preprocessing the PPG Signal (Filtering)

Before extracting features, the PPG signal must be cleaned to remove unwanted noise. This process involves:

* + **Removing** **High-Frequency** **Noise:** Using a **low-pass** **filter** to eliminate sudden spikes and artifacts.
  + **Removing** **Slow** **Variations** **(Baseline** **Drift):** Using a **high-pass** **filter** to focus on heartbeat-related changes.
  + **Smoothing** **the** **Signal:** Applying signal processing techniques to improve peak detection.

The cleaned signal is now ready for feature extraction.

#### Peak Detection and RR Interval Calculation

After filtering, we detect peaks in the PPG signal. These peaks represent heartbeats, and the time difference between two consecutive peaks is called the **RR** **interval** (measured in seconds).

#### RR Interval=Time between two successive peaks

By analyzing RR intervals, we can calculate:

* + **Heart** **Rate** **(HR):** The number of heartbeats per minute (BPM).

#### Interval (s)HR= 60 / RR Interval (s)

* + **Heart** **Rate** **Variability** **(HRV):** The variation in RR intervals, which provides insights into stress levels and cardiovascular health.

#### Other Important Features in PPG Signals

Apart from heart rate and RR intervals, we can extract additional features from the PPG signal:

* + **Pulse** **Amplitude:** The height difference between peaks and valleys, indicating blood flow strength.
  + **Pulse** **Width:** The time duration of each pulse, giving insights into arterial stiffness.
  + **Pulse** **Slope:** The rate at which blood flow increases or decreases.

These features are useful for diagnosing heart conditions and monitoring overall health.

## Source code & Output:

import neurokit2 as nk import numpy as np

import matplotlib.pyplot as plt

# 1. Generate a synthetic PPG signal (replace this with real PPG data)

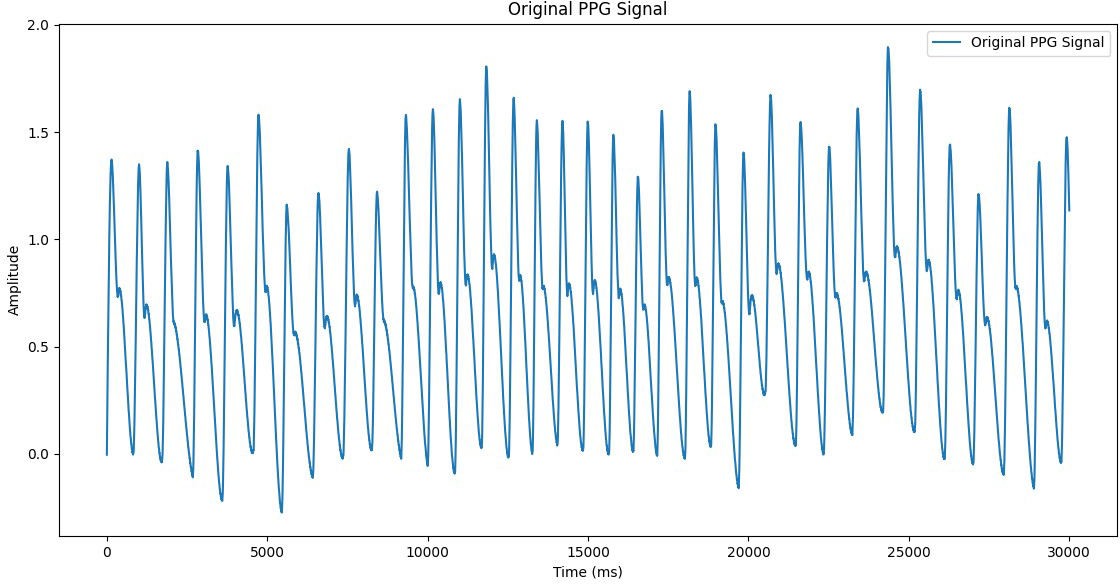
ppg\_signal = nk.ppg\_simulate(duration=30, sampling\_rate=1000)

# Plot the original PPG signal plt.figure(figsize=(10, 4))

plt.plot(ppg\_signal, label="Original PPG Signal") plt.title("Original PPG Signal")

plt.xlabel("Time (ms)") plt.ylabel("Amplitude") plt.legend()

plt.show()



# 2. Filter the signal

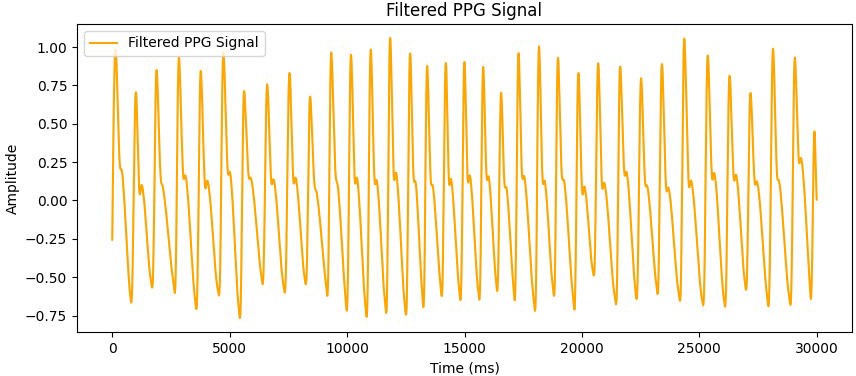
ppg\_filtered = nk.ppg\_clean(ppg\_signal, sampling\_rate=1000)

# Plot the filtered PPG signal plt.figure(figsize=(10, 4))

plt.plot(ppg\_filtered, label="Filtered PPG Signal", color="orange") plt.title("Filtered PPG Signal")

plt.xlabel("Time (ms)") plt.ylabel("Amplitude") plt.legend()

plt.show()



# 3. Detect Peaks

peaks = nk.ppg\_findpeaks(ppg\_filtered, sampling\_rate=1000) peak\_indices = peaks["PPG\_Peaks"]

# Plot the filtered signal with detected peaks plt.figure(figsize=(10, 4))

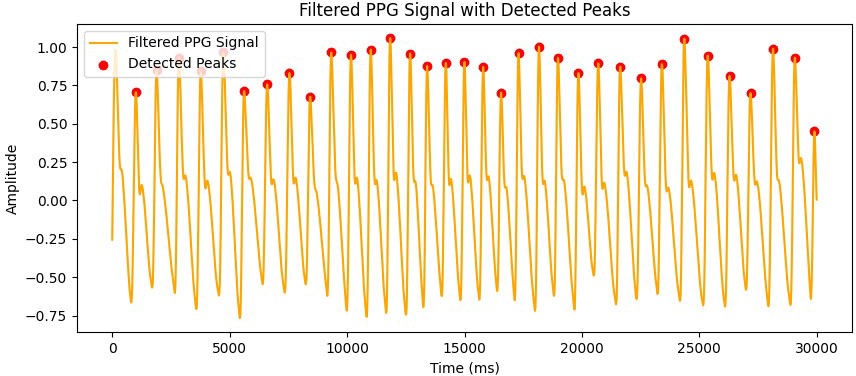
plt.plot(ppg\_filtered, label="Filtered PPG Signal", color="orange")

plt.scatter(peak\_indices, ppg\_filtered[peak\_indices], color="red", label="Detected Peaks")

plt.title("Filtered PPG Signal with Detected Peaks") plt.xlabel("Time (ms)")

plt.ylabel("Amplitude") plt.legend()

plt.show()



# 4. Calculate RR Intervals

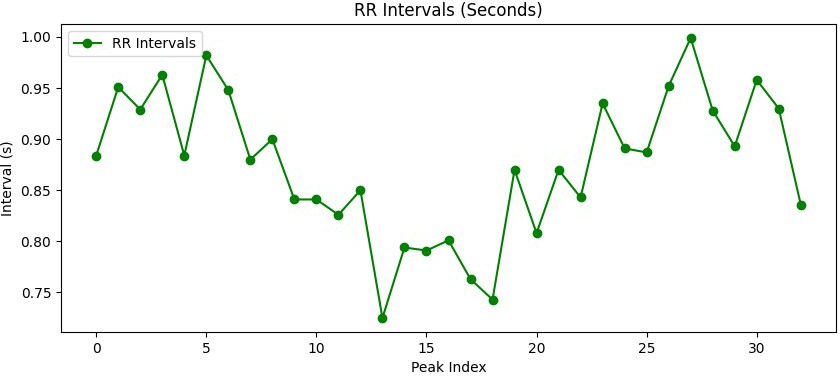
rr\_intervals = np.diff(peak\_indices) / 1000 # Convert to seconds

# Plot RR intervals plt.figure(figsize=(10, 4))

plt.plot(rr\_intervals, label="RR Intervals", color="green", marker="o") plt.title("RR Intervals (Seconds)")

plt.xlabel("Peak Index")

plt.ylabel("Interval (s)") plt.legend()



plt.show()

# 5. Calculate Heart Rate

heart\_rate = 60 / rr\_intervals # BPM

# Plot Heart Rate plt.figure(figsize=(10, 4))

plt.plot(heart\_rate, label="Heart Rate (BPM)", color="purple", marker="o") plt.title("Heart Rate (BPM)")

plt.xlabel("Interval Index")

plt.ylabel("Heart Rate (BPM)") plt.legend()

plt.show()



# Lab Report on Discrete Fourier Transform (DFT)

## Theory

The Discrete Fourier Transform (DFT) is a mathematical technique used to transform a discrete-time signal from the time domain into the frequency

domain. It is a fundamental tool in digital signal processing (DSP) and is widely used for analyzing the frequency content of signals, filtering, and spectral analysis.

The DFT of a discrete-time signal *x*[*n*] of length *N* is defined as:

***X*[*k*]=*n*=0∑** ***N*−1*x*[*n*]*e*−*jN*2*πkn*** **for** ***k*=0,1,2,…,*N*−1**

where:

* *X*[*k*] is the DFT of the signal *x*[*n*], representing the frequency components.
* *k* is the frequency index.
* *N* is the total number of samples in the signal.

The inverse DFT (IDFT) is used to reconstruct the original signal from its frequency components and is given by:

***x*[*n*]=*k*=0∑*N*−1*X*[*k*]*ejN*2*πkn*** **for** ***n*=0,1,2,…,*N*−1**

Some properties of DFT:

1. Frequency Resolution: The frequency resolution of the DFT depends on the number of samples *N* and the sampling rate *fs*. The frequency resolution is given by *Nfs*.
2. Periodicity: The DFT assumes that the signal is periodic with period *N*. This means that the signal repeats itself every *N* samples.
3. Fast Fourier Transform (FFT): The FFT is an efficient algorithm to compute the DFT, reducing the computational complexity

from *O*(*N*2) to *O*(*N*log*N*).

Applications:

* + Spectral Analysis: The DFT is used to analyze the frequency content of signals, such as audio, vibration, and biomedical signals.
  + Filtering: By transforming a signal into the frequency domain, unwanted frequency components can be filtered out.
  + Signal Compression: The DFT is used in compression algorithms like JPEG and MP3 to reduce the size of data by removing redundant frequency components.

## Source code &Output:

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

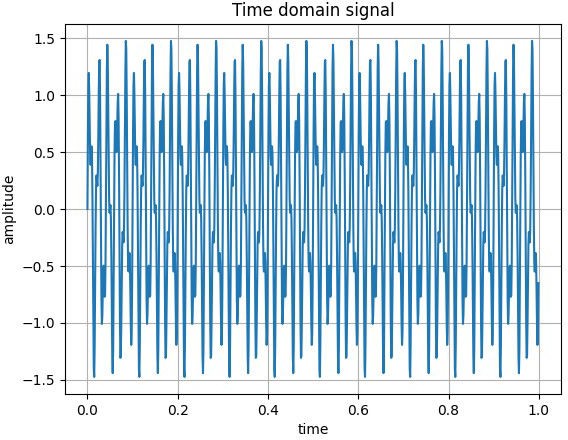
# Signal: Combination of 50 Hz and 120 Hz sine waves f1, f2= 50, 120 #f3=150

signal = np.sin(2 \* np.pi \* f1 \* t) + 0.5 \* np.sin(2 \* np.pi \* f2 \* t) #+ 1.5 \* np.sin(2 \* np.pi \* f3 \* t)

plt.plot(t,signal) # Single-sided spectrum plt.title("Time domain signal") plt.xlabel("time")

plt.ylabel("amplitude") plt.grid()

plt.show()



# 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()

