**Md. Noman Bhuiyan**

**Roll No: 220608**

**Experiment no: 01**

**Experiment name: Signal operation**  **x1(n)=2x(n−5)−3x(n+4) plot in MATLAB,** **where x(n)={1,2,3 ↑,** **4, 5,6,7,6,5,4,3,2,1}**

**Objectives**:

* Analyze a modif singals for system behavior
* Extract or highlight information
* Facilate system design and simultaion

**Theory:**

In discrete-time signal processing, several operations can be performed to study or modify signals. There are some singal operation given below.

* Signal addition: Signal addition involves combining two discrete signals by adding their corresponding samples. we can write it,

y(n) = x1(n) + x2(n)

* Signal multiplication: Element-wise multiplication combines two signals by multiplying their samples. We can represent it like:

y(n) = x1(n) . x2(n)

* Scaling: In operation signal multipy the amplitude signal by constant like

y(n)=k.x(n)

* Shifting: In this operation, the time scale of the signal is changed, which compressess of expand the signal.This operation shifts the signal forward or backword by a specific ammont of time like,

Right shift: x(n-k), where k > 0

Left shift: x(n+k), where K < 0

* Signal Folding: Folding (or time reversal) reflects the signal across the vertical axis, effectively reversing the order of the signal samples. Folding is crucial in analyzing symmetrical properties and determining energy distribution.

y(n) = x(-n)

Signal operations are performed to analyze, modify, and extract meaningful information form signals, enabling effective system design processing tasks in engineering and techology

**Source Code**:

import numpy as np

import matplotlib.pyplot as plt

def sigshift(x, n, k):

    n\_new = n + k

    return x, n\_new

def sigadd(x1, n1, x2, n2):

    min\_n = min(min(n1), min(n2))

    max\_n = max(max(n1), max(n2))

    n = np.arange(min\_n, max\_n + 1)

    y1 = np.zeros(len(n))

    y2 = np.zeros(len(n))

    y1[np.isin(n, n1)] = x1

    y2[np.isin(n, n2)] = x2

    y = y1 + y2

    return y, n

# original signal

n = np.arange(-2, 11)

x = np.concatenate((np.arange(1, 8), np.arange(6, 0, -1)))

# shift signal

x11, n11 = sigshift(x, n, 5)

x12, n12 = sigshift(x, n, -4)

# add signal

x1, n1 = sigadd(2\*x11, n11, -3\*x12, n12)

# graph

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

plt.subplot(2, 2, 1)

plt.stem(n11, x11, 'k', basefmt=" ")

plt.title('Signal Shift')

plt.xlabel('n11')

plt.ylabel('x11')

plt.subplot(2, 2, 2)

plt.stem(n12, x12, 'k', basefmt=" ")

plt.title('Signal Shift 2')

plt.xlabel('n12')

plt.ylabel('x12')

plt.subplot(2, 1, 2)

plt.stem(n1, x1, 'b', basefmt=" ")

plt.title('Sequence for Part (a): x1(n) = 2x(n-5) - 3x(n+4)')

plt.xlabel('n')

plt.ylabel('x1(n)')

plt.tight\_layout()

plt.show()

**Output**:



**Experiment No:** 02

**Experiment name**: using ppg signal detected peaks of signal which is define heartbeats

**Objective**:

1. Detect peaks corresponding to each heartbeat to calculate the heart rate.
2. Assess the number of peaks, their amplitudes, and intervals to determine the quality of the PPG singal.

**Theory**:

Photoplethysmography (PPG) is a simple optical technique used to detect volumetric changes in blood in peripheral circulation. From ppg signal if we want to find out the heartbeats or peaks value we can follow the five step:

* Raw ppg signal
* Add noise
* Filter
* Normalize
* Detect peaks
* Adding noise to the raw ppg signal: In the real world conditions where PPG signals are often corrupted by noise. Like white noise, motion artifacts , baseline wander etc. Add noise to the rawsingal mathmatically:

Noise singal = Raw signal + Noise

Where the noise can be generatde using functions like numpy.random.normal for white noise.

* Recovering the original singal using Butterworth filter: Remove noise and restore the signal components of interest. Design the filter using scipy.signal.butter . Apply it using scipy.signal.filtfilt, which performs forward and backward filtering to eliminate phase distortion. Filter low cutoff frequency and high cutoff frequency
* Normalizing the ppg signal: Scale the signal ampliture to a uniform range for consistency and easier analysis. To normalize we can use:

Normalized signal =(signal – min(signal))/(max(signal) – min(signal))

* Peak detection in ppg signal:Identify the peaks corresponding to heartbeats.

Typically, they occur at regular intervals in a clean signal. Their amplitudes depend on the strength of blood flow. Use scipy.singal.find\_peaks()

**Code**:

1. **Raw ppg signal**:

import numpy as np

import matplotlib.pyplot as plt

fs=100

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

ppg\_signal = 0.6\*np.sin(2\*np.pi\*1,2\*t) + np.random.normal(0,0.05,len(t))

plt.plot(t,ppg\_signal)

plt.title(“raw ppg signal”)

plt.xlabel(“time (seconds)”)

plt.ylabel(“amplitude”)

plt.show()

1. **Original signal**:

import numpy as np

import matplotlib.pyplot as plt

fs = 100

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

original\_signal = 0.6\*np.sin(2\*np.pi\*1,2\*t)

plt.plot(t,original \_signal)

plt.title(“Original”)

plt.xlabel(“time (seconds)”)

plt.ylabel(“amplitude”)

plt.show()

1. **Noise add**:

import numpy as np

import matplotlib.pyplot as plt

fs = 100

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

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

plt.plot(t,noise\_signal)

plt.title(“add noise”)

plt.xlabel(“time (seconds)”)

plt.ylabel(“amplitude”)

plt.show()

1. **Filter signal:**

from scipy.signal import butter, filtfilt

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)

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

plt.plot(t,filtered\_ppg)

plt.title(“filtered ppg signal”)

plt.xlabel(“time (seconds)”)

plt.ylabel(“amplitude”)

plt.show()

1. **Normalized signal**:

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

plt.plot(t,normalized\_ppg)

plt.title(“normalized ppg signal”)

plt.xlabel(“time(seconds)”)

plt.ylabel(“normalized amplitude”)

plt.show()

1. **Peak find**:

from scipy.signal import find\_peaks

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

ibi = np.diff(peaks)/fs

heart\_rate = 60 / ibi

plt.plot(t,normalize\_ppg)

plt.plot(t,[peaks],normalized\_ppg[peaks],”x”)

plt.title(“ppg signal with detected peaks (heartbeats)”)

plt.xlabel(“time(seconds)”)

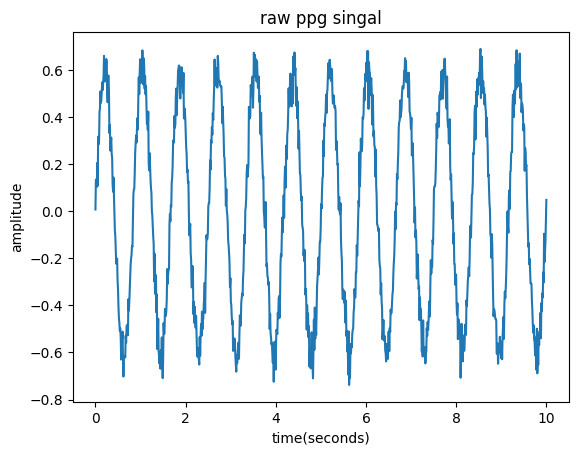
plt.ylabel(“normalized amplitude”)

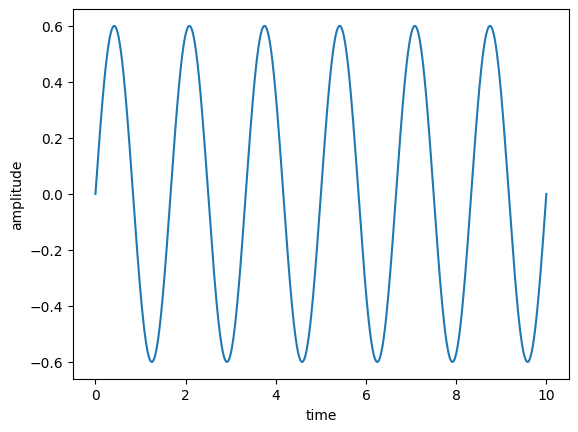
plt.show()

print(“heart rate:”, np.mean(heart\_rate), “BPM”)

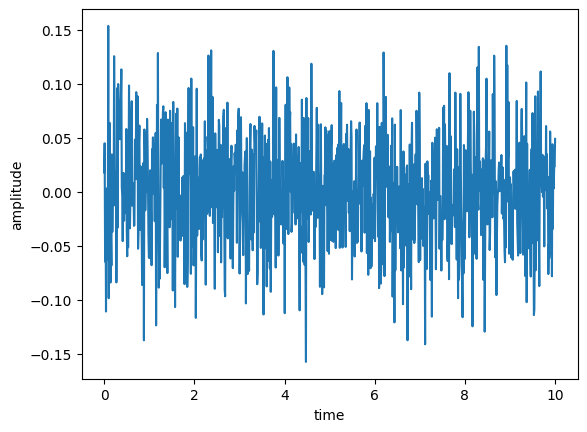
**output:**

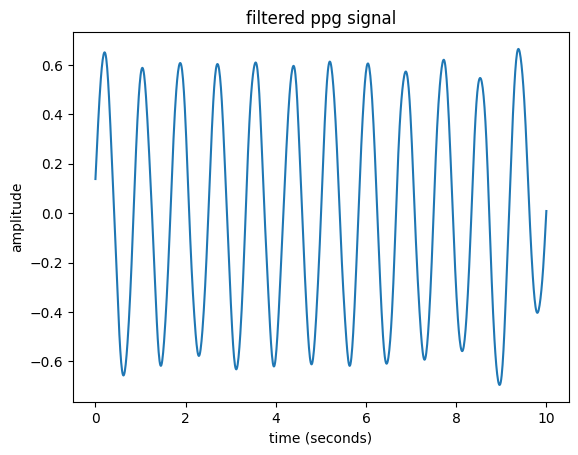
**1.Raw signal:**

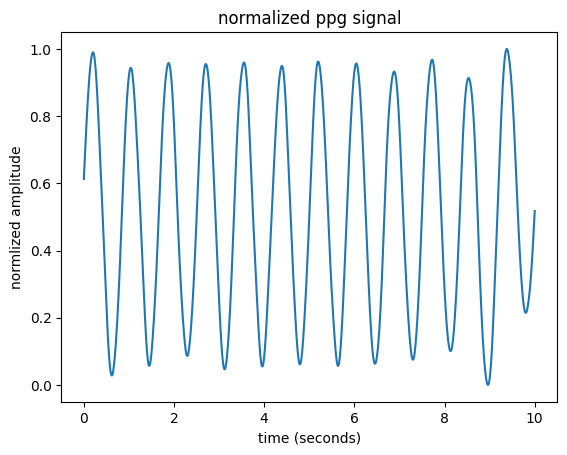


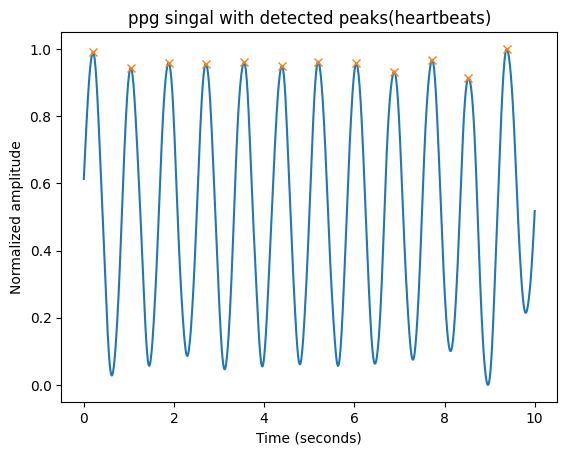
**2.original signal:**

**3.noise signal:**

****

**4.filter signal**:

**5.Normalized ppg signal**:

****6.**ppg signal peaks detect(Heartbeats):**

**Experiment No**: 03

**Experiment name**: Convolution in Signal and Systems

**Objectives:**

1. Understand the concept of convolution and its role in signal processing.
2. Implement convolution mathematically and visualize the effect on signals.
3. Analyze the response of a system to an input signal using convolution.
4. Explore different types of convolution (linear, circular) and their applications in real-world signals.

**Theory:**

Convolution is a fundamental mathematical operation in signal processing that describes how an input signal interacts with a system to produce an output. It is widely used in filtering, system analysis, image processing, and neural networks.

Mathematically, the convolution of two signals x(t)x(t)x(t) and h(t)h(t)h(t) is defined as:

y(t)=(x∗h)(t) =

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

y[n]=(x∗h)[n] =

Consider an input signal x(t)x(t)x(t) and a system’s impulse response h(t)h(t)h(t). The impulse response h(t)h(t)h(t) is flipped and shifted across the time axis. The overlapping regions of x(t)x(t)x(t) and h(t)h(t)h(t) are multiplied and integrated (for continuous signals) or summed (for discrete signals). The convolution process gives the system’s output, which represents how the input has been modified by the system.

**Code:**

import numpy as np

import matplotlib.pyplot as plt

# Define two input sequences

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

h = np.array([1, -1, 2]) # Second sequence

# Perform convolution

y = np.convolve(x, h, mode='full')

# Display results

print('First sequence x[n]:', x)

print('Second sequence h[n]:', h)

print('Convolution result y[n]:', y)

# Plot the signals

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

plt.subplot(3, 1, 1)

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

plt.title('Input Sequence x[n]')

plt.xlabel('n')

plt.ylabel('Amplitude')

plt.grid()

plt.subplot(3, 1, 2)

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

plt.title('Impulse Response h[n]')

plt.xlabel('n')

plt.ylabel('Amplitude')

plt.grid()

plt.subplot(3, 1, 3)

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

plt.title('Convolution Result y[n]')

plt.xlabel('n')

plt.ylabel('Amplitude')

plt.grid()

plt.tight\_layout()

plt.show()

**output:**

**Experiment no: 04**

**Experiment name:** Correlation in Signal and Systems

**Objectives:**

1. Understand the concept of correlation and its significance in signal processing.
2. Compute and analyze the correlation between two signals to measure similarity.
3. Differentiate between auto-correlation and cross-correlation.
4. Apply correlation techniques to detect patterns in signals.

**Theory:**

Correlation is a mathematical operation that measures the similarity between two signals by comparing their shape over time. It helps in detecting patterns, determining signal alignment, and analyzing periodicity.

The correlation between two signals x(t)x(t)x(t) and y(t)y(t)y(t) is defined as:

Rxy​(τ)=

For discrete-time signals, the correlation function is:

Where:

* Rxy(τ)R\_{xy}(\tau)Rxy​(τ) is the cross-correlation function.
* τ\tauτ is the time lag that shifts one signal relative to the other.
* If x=yx = yx=y, it is called auto-correlation, which measures how a signal correlates with itself over time.

**Code:**

import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import correlate

# Generate a sample signal (sine wave with noise)

fs = 100 # Sampling frequency (Hz)

t = np.linspace(0, 2, 2 \* fs, endpoint=False)

signal\_x = np.sin(2 \* np.pi \* 5 \* t) + 0.5 \* np.random.normal(size=len(t)) # Signal with noise

# Create another signal (shifted and noisy version of signal\_x)

signal\_y = np.sin(2 \* np.pi \* 5 \* (t - 0.2)) + 0.5 \* np.random.normal(size=len(t))

# Auto-correlation (signal\_x with itself)

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

lags\_auto = np.arange(-len(signal\_x) + 1, len(signal\_x))

# Cross-correlation (signal\_x with signal\_y)

cross\_corr = correlate(signal\_x, signal\_y, mode='full')

lags\_cross = np.arange(-len(signal\_x) + 1, len(signal\_x))

# Plot the signals

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

plt.subplot(3, 1, 1)

plt.plot(t, signal\_x, label="Signal X")

plt.plot(t, signal\_y, label="Signal Y", linestyle='dashed')

plt.legend()

plt.title("Original Signals")

# Plot Auto-correlation

plt.subplot(3, 1, 2)

plt.plot(lags\_auto, auto\_corr, color='blue')

plt.title("Auto-Correlation of Signal X")

plt.xlabel("Lags")

plt.ylabel("Correlation")

# Plot Cross-correlation

plt.subplot(3, 1, 3)

plt.plot(lags\_cross, cross\_corr, color='red')

plt.title("Cross-Correlation between Signal X and Signal Y")

plt.xlabel("Lags")

plt.ylabel("Correlation")

plt.tight\_layout()

plt.show()



**output:**

**Experiment no:** 05

**Experiment name**: Fourier transform

**Objectives:**

1. Understand Fourier Transform (FT)
2. Visualize Fourier Transform results

**Theory:**

Fourier Transform (Continuous)

The Fourier Transform (FT) converts a time-domain signal into its frequency-domain equivalent. It is mathematically represented as:

X(f)=

where,

* X(f)X(f)X(f) is the frequency-domain representation.
* x(t)x(t)x(t) is the time-domain signal.
* e−j2πfte^{-j 2 \pi f t}e−j2πft is a complex exponential function.

Discrete Fourier Transform (DFT)

* In Digital Signal Processing (DSP), we use the Discrete Fourier Transform (DFT) to analyze discrete signals. The DFT formula is:

where,

* x[n] is the discrete-time input signal.
* X[k] is the frequency-domain representation.
* N is the total number of samples.

**Code:**

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)

Fs = 1000

T = 1 / Fs

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

f1, f2 = 50, 120

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

plt.plot(t,signal)

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]))

plt.title("DFT Frequency Spectrum")

plt.xlabel("Frequency (Hz)")

plt.ylabel("Magnitude")

plt.grid()

plt.show()

import numpy as np

N = 1024

Fs = 1000

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

print(freq\_bins[:10])

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)

freq\_signal = 440

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

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

noisy\_signal = pure\_signal + noise

fft\_signal = fft(noisy\_signal)

freqs = fftfreq(len(fft\_signal), T)

fft\_filtered = fft\_signal.copy()

fft\_filtered[np.abs(freqs) > 500] = 0

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="yellow")

plt.legend()

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

plt.tight\_layout()

plt.show()

**Output:**

