PABNA UNIVERSITY OF SCIENCE AND TECHNOLOGY

Faculty of Engineering & Technology

Department of Information and Communication Engineering

Course Title: Signals and System Sessional

Course Code ICE-2204

Experiment No:

Experiment Name:

Submitted By:	Submitted To:
Roll: Session: 2022-23 2nd Year 2nd Semester Department of ICE, PUST	**Dr. Md. Imran Hossain** Associate Professor Department of ICE Pabna University of Science and Technology, Pabna.

Experiment No: 01

Experiment Name: Explain and Implementation of following Elementary Discrete Signals Using MATLAB/Python i) The unit sample sequence. ii) The unit step signal. iii) The unit ramp signal.

Theory:

1. The unit sample sequence: The unit sample sequence, also known as the discrete-time impulse or Kronecker delta function, is the most fundamental signal in digital signal processing. It is defined such that its value is one at n=0 and zero at all other values of n.
Any discrete-time signal can be represented as a sum of shifted and scaled unit sample sequences.

This signal is very important because it acts like the “building block” for all discrete-time signals. Any arbitrary discrete-time signal can be represented as a combination of shifted and scaled unit sample sequences. One of its most important properties is the sifting property, which allows it to extract the value of another sequence at a specific index. In system analysis, the response of a system to a unit sample sequence is called the impulse response, which completely characterizes the system. Therefore, the unit sample sequence plays a crucial role in representing signals, analyzing systems, and performing operations such as convolution in digital signal processing.

2. **The Unit Step Signal:** The unit step signal is one of the most commonly used elementary discrete-time signals in digital signal processing. It is defined as a sequence that is equal to zero for all negative values of n, and equal to one for n≥n0 .

The unit step signal is important because it represents a sudden change or “switching on” at a specific instant. It is widely used to model input signals that begin at a particular point in time and remain constant afterwards. In system analysis, the unit step is closely related to the unit sample sequence since the step can be written as the cumulative sum of impulses. Similarly, the derivative (or first difference) of the step sequence is the unit sample sequence. The unit step is also essential in defining and constructing more complex signals, such as the unit ramp. Its ability to represent the starting point of signals makes it very useful in both theoretical analysis and practical applications of digital signal processing.

3. The Unit Ramp Signal: The unit ramp signal is another fundamental discrete-time signal that grows linearly with time. It is defined as a sequence that is equal to zero for all negative values of n, and equal to n for n≥n0\geq n0≥0. Mathematically, it can be expressed as

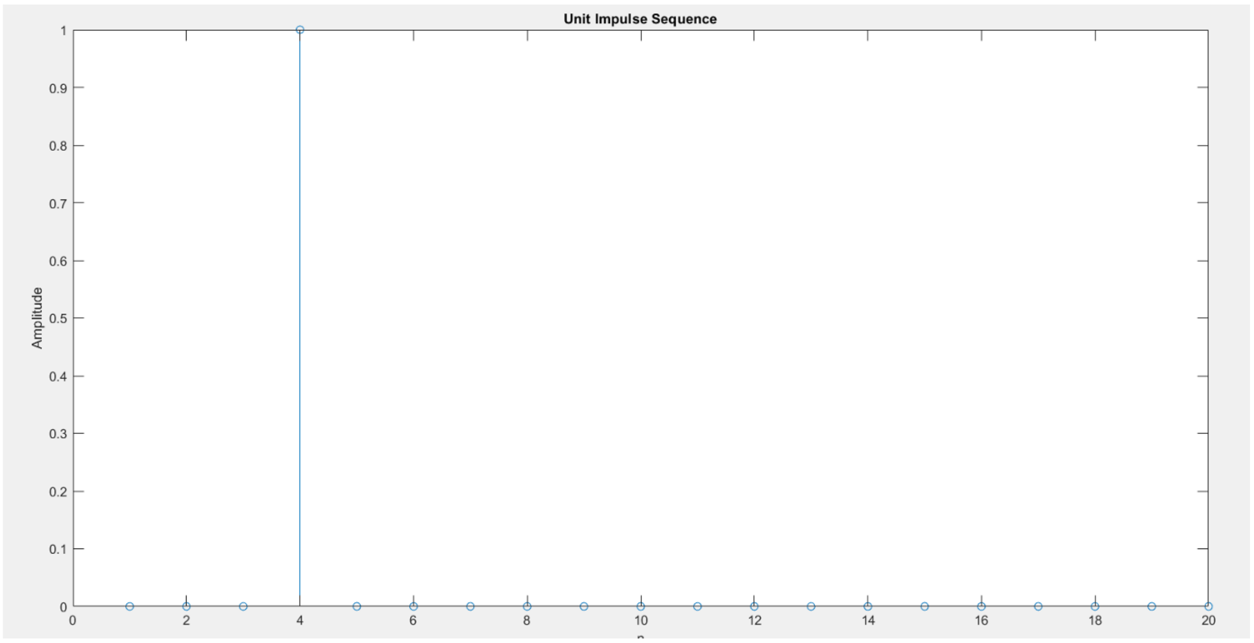
The unit ramp signal is important because it represents a steadily increasing signal that begins at the origin and continues indefinitely in the positive direction. It is closely related to the unit step signal, as the ramp can be obtained by the cumulative sum of the step sequence, while the step sequence itself can be seen as the discrete derivative of the ramp. The unit ramp is useful in the analysis of discrete-time systems, particularly in studying system behavior with signals that increase over time. It also serves as a basis for constructing more complex piecewise linear signals. Overall, the unit ramp sequence, along with the unit sample and unit step signals, forms an essential set of basic signals for understanding and analyzing digital signal processing.

Source Code(The unit sample sequence) :

```
function[x,n]=impulseSequence(n0,n1,n2)

 n=[n1:n2];

x=[(n-n0)==0];
```

```
 end
 ----------------------
n0=4;
 n1=1;
n2=20;
 n=[n1:n2];
x=[n==n0];
stem(n,x);
 xlabel('n');
ylabel('Amplitude');
 title('Unit Impulse Sequence');
```

Output:

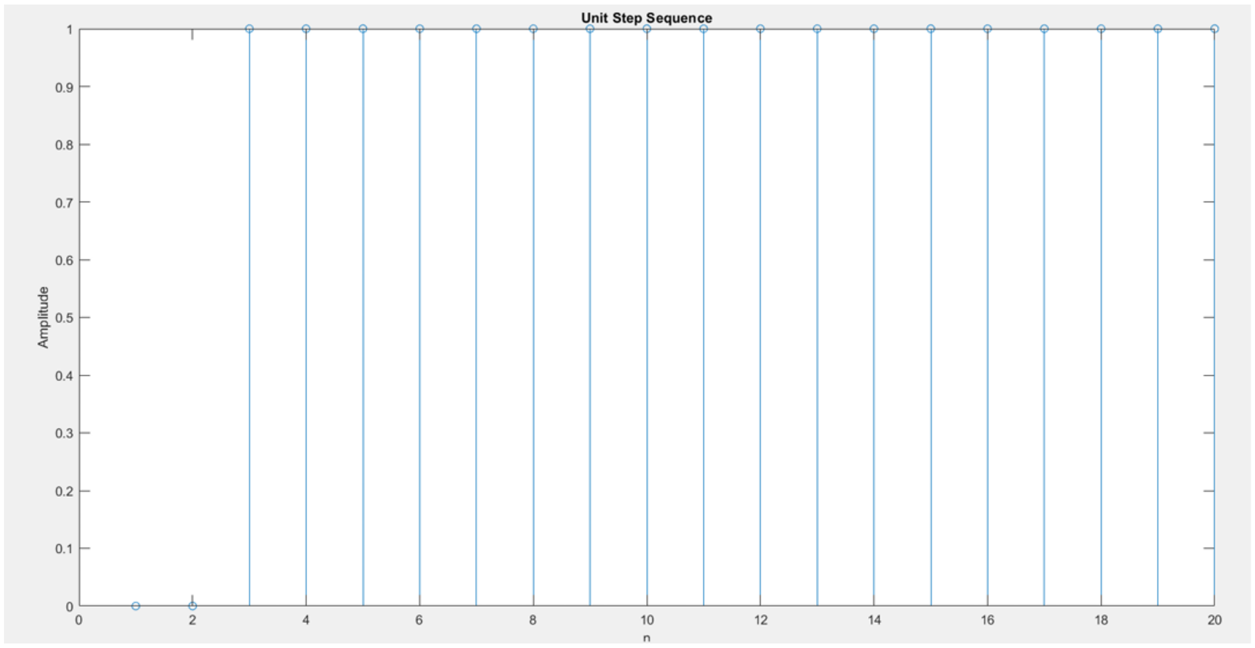


Source Code(The unit Step signal) :

```
function[x,n]=StepSequence(n0,n1,n2)

n=[n1:n2];

x=[(n-n0)>=0];

 end

 ------------------

n0=3;

n1=1;

n2=20;

n=[n1:n2];

x=[(n-n0)>=0];
```

```
stem(n,x);
 xlabel('n');
ylabel('Amplitude');
 title('Unit Step Sequence');
```

Output:

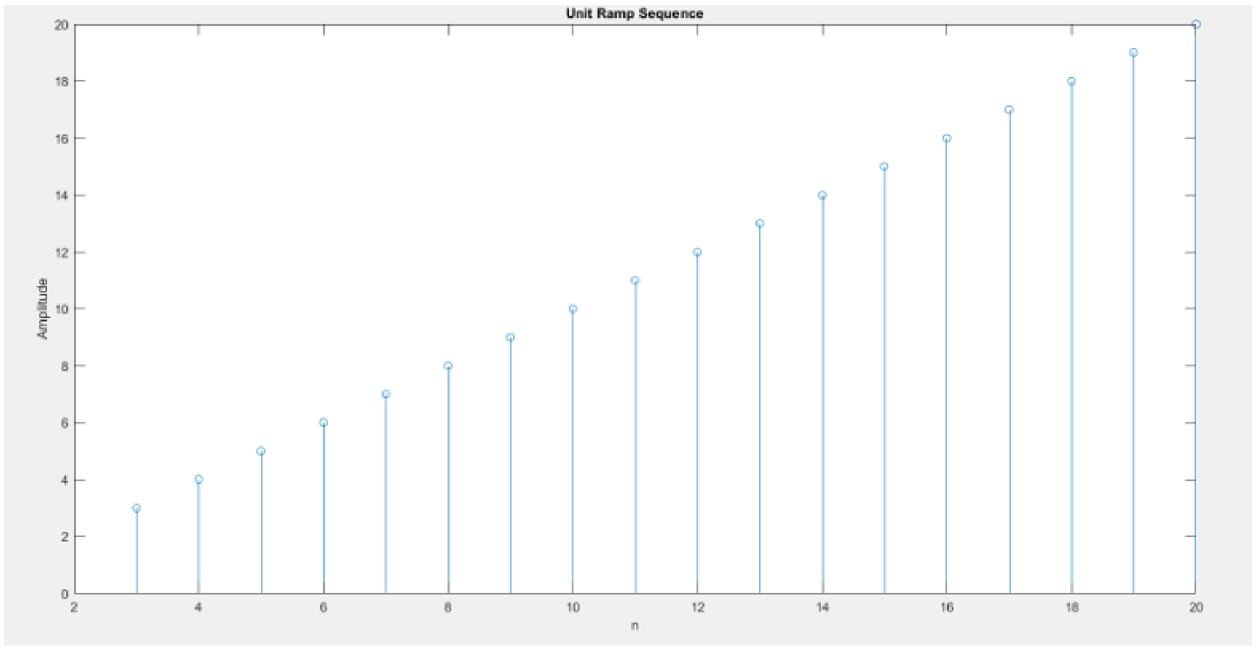


Source Code(The unit Ramp signal) :

```
function[x,n]=UnitRampSequence(n1,n2)
n=[n1:n2];
x=n;
end
```

```
-------------------
 x=n;
 n1=3;
n2=20;
 n=[n1:n2];
x=n;
stem(n,x);
 xlabel('n');
ylabel('Amplitude');
 title('Unit Ramp Sequence');
```

Output:



Experiment No:2

Experiment Name: Let x[n]={1,2,3,4,**5**,6,7,6,5,4,3,2,1}.Here bold number indicates the 0 index. Determine and plot the following sequences.
y[n]=2x[n–5]–3x[n+4].

Theory:

In digital signal processing, signals are often represented as discrete-time sequences**,** written as x[n]. These sequences can be manipulated using different operations such as shifting, scaling, and combination.

Time Shifting: Time shifting changes the position of a signal along the time axis.

Right shift (delay): x[n−k], k>0 ; moves the sequence k units to the right.

Left shift (advance): x[n+k], k<0 ; moves the sequence k units to the left.

Amplitude Scaling: Scaling is multiplying each sample by a constant.

Positive scaling:

$$y[n] = a\cdot x[n],\ a>0$$

This multiplies each sample of the signal by a, increasing or decreasing its amplitude.
Negative scaling:

$$y[n] = -\ a\cdot x[n]$$

This not only scales but also flips the signal vertically.

Combination of Sequences: Multiple sequences can be combined through addition, subtraction, or multiplication.

Addition/Subtraction:

$$y[n]=x1[n] \pm x2[n]$$

Useful in constructing new signals from existing ones.

Weighted Combination:

$$y[n]=a\cdot x1[n] + b\cdot x2[n]$$

Here, each sequence is scaled first and then combined.

Example:

$$y[n]=2x[n-5]-3x[n+4]$$

This shows both shifting (right by 5 and left by 4), scaling (multiply by 2 and -3), and combination (addition).

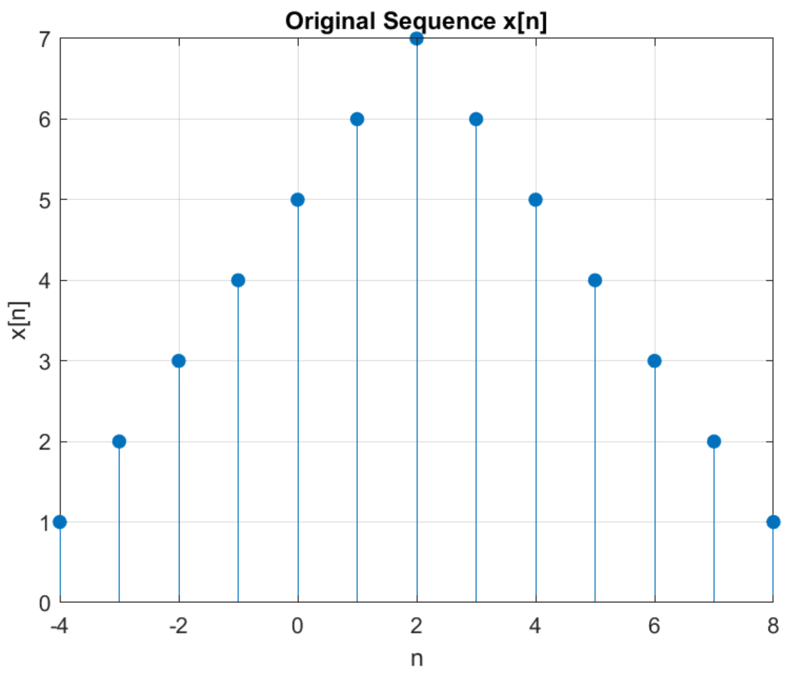
Source Code:

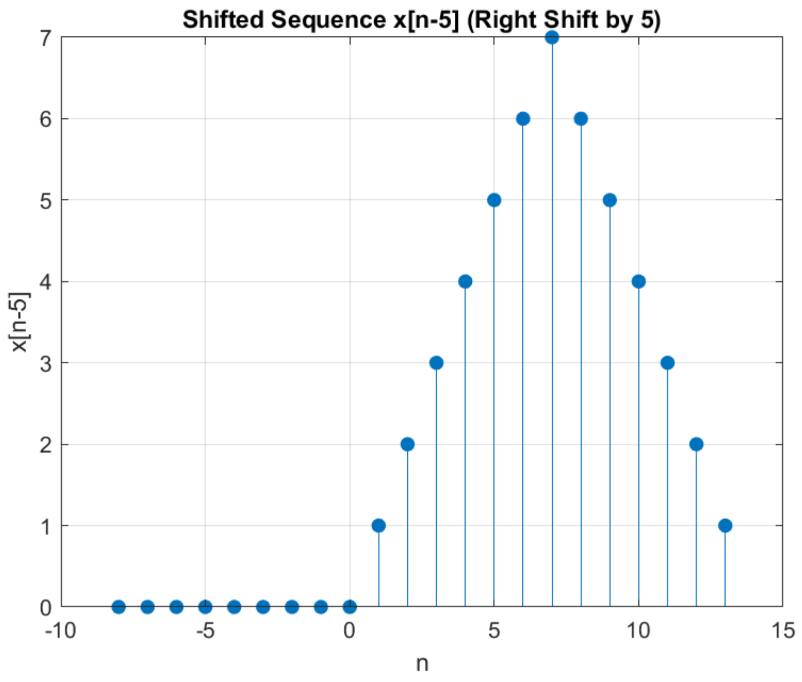
```
clc; clear; close all;

% Original sequence definition

x = [1 2 3 4 5 6 7 6 5 4 3 2 1];  % x[n]
x_n = -4:8;                        % indices for x[n]


% Plot the original sequence x[n]

figure;
stem(x_n, x, 'filled');           % stem plot of x[n]
xlabel('n');
ylabel('x[n]');
title('Original Sequence x[n]');
grid on;


% Prepare for shifting operations
```

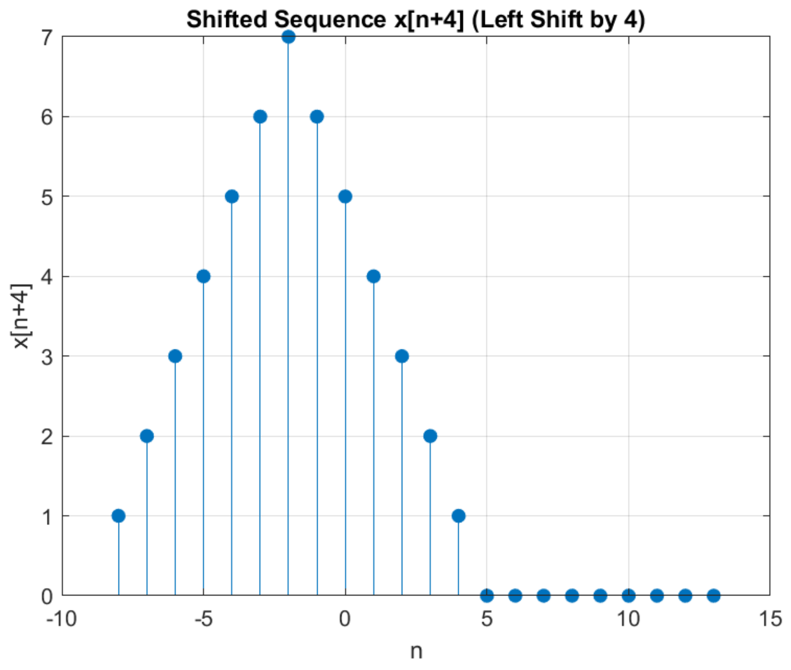
```
y_n = -8:13;                    % union range to cover both shifts
x1 = zeros(1,length(y_n));        % preallocate for x[n-5]
x2 = zeros(1,length(y_n));        % preallocate for x[n+4]


% Compute shifted sequences with zero-padding

for k = 1:length(y_n)
    n = y_n(k);

    % ----- Compute x[n-5] -----
    idx1 = find(x_n == n-5);      % find index where n-5 exists
    if ~isempty(idx1)
        x1(k) = x(idx1);          % assign value if exists
    else
        x1(k) = 0;                % zero-padding if index out of range
    end

    % ----- Compute x[n+4] -----
    idx2 = find(x_n == n+4);      % find index where n+4 exists
    if ~isempty(idx2)
        x2(k) = x(idx2);          % assign value if exists
    else
        x2(k) = 0;                % zero-padding if index out of range
    end
end


% Plot shifted sequence x[n-5]

figure;
stem(y_n, x1, 'filled');
xlabel('n');
ylabel('x[n-5]');
title('Shifted Sequence x[n-5] (Right Shift by 5)');
grid on;

% Plot shifted sequence x[n+4]

figure;
```

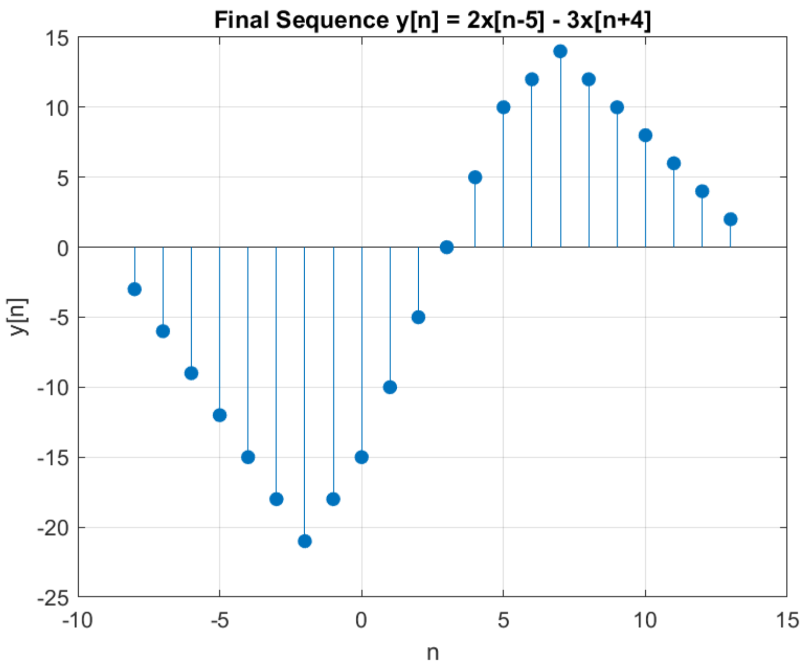
```
stem(y_n, x2, 'filled');
xlabel('n');
ylabel('x[n+4]');
title('Shifted Sequence x[n+4] (Left Shift by 4)');
grid on;

% Compute and plot final output y[n]

y = 2*x1 - 3*x2;          % linear combination
 figure;
stem(y_n, y, 'filled');
xlabel('n');
ylabel('y[n]');
title('Final Sequence y[n] = 2x[n-5] - 3x[n+4]');
grid on;
```

Output:




Shifted Sequence x[n-5] (Right Shift by 5)
x[n-5]
n
0
1
2
3
4
5
6
7
-10
-5
0
5
10
15


Shifted Sequence x[n+4] (Left Shift by 4)
x[n+4]
n
0
1
2
3
4
5
6
7
-10
-5
0
5
10
15


Final Sequence y[n] = 2x[n-5] - 3x[n+4]
y[n]
n
15
10
5
0
-5
-10
-15
-20
-25
-10
-5
0
5
10
15

Experiment No : 03

Experiment Name

Determine and Plot the Discrete-Time Sequence:

$$X(n)=2\delta(n+2)-\delta(n-4) \quad \text{where } -7\leq n\leq 7$$

Theory

A discrete-time signal is defined only at integer values of nn. The unit impulse function δ(n)\delta(n) is defined as:

$$\delta(n) = \begin{cases} 1, & n = 0 \\ 0, & n \neq 0 \end{cases}$$

In this experiment, the sequence is a linear combination of two impulses:

- $2\delta(n+2)$ → impulse of amplitude 2 at n=−2n = -2
- $-\delta(n-4)$ → impulse of amplitude -1 at n=4n = 4

The rest of the values of the sequence are zero. Plotting this sequence helps visualize discrete signals and understand impulse behavior in signal processing.

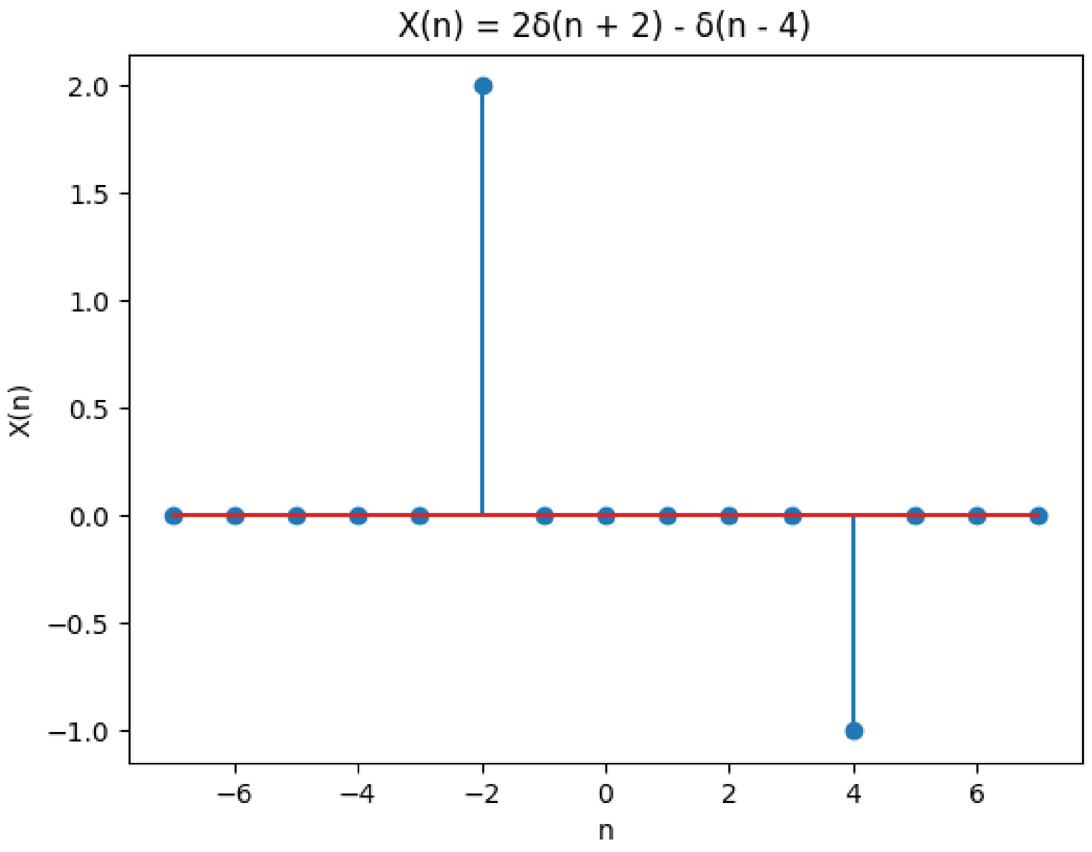
Algorithm (Pseudo Code)

```
BEGIN
1. Define the range of n from -7 to 7
2. Initialize X(n) as a zero array
3. Assign values according to the sequence:
    ▪ X(-2) = 2
    ▪ X(4) = -1
4. Plot the sequence using stem plot
5. Label axes and add title
6. Display the plot
END
```

Source Code (Python)

```python
import numpy as np
import matplotlib.pyplot as plt

# --- Step 1: Define n and initialize X(n) ---
n = np.arange(-7, 8)
x = np.zeros_like(n)

# --- Step 2: Assign values according to the sequence ---
x[n == -2] = 2
x[n == 4] = -1

# --- Step 3: Plot the sequence ---
plt.stem(n, x)
plt.title('X(n) = 2δ(n + 2) - δ(n - 4)')
plt.xlabel('n')
plt.ylabel('X(n)')
plt.show()
```

Output:



Experiment No :04

Title: Implementation of Signal Operations (Addition, Folding, and Shifting) Using User-Defined Functions.

Objectives: To understand and implement basic signal operations such as addition, folding, and shifting using user-defined functions in MATLAB. The aim is to visualize how these operations modify signals in the time domain.

Theory:

1. **Signal Addition:** Signal addition is the process of combining two signals by adding their corresponding values at each time index.

   If two signals $x_1(n)$ and $x_2(n)$ are defined, their addition is: $y(n)= x_1(n) + x_2(n)$

2. **Signal Folding:** Signal folding (or time reversal) is the process of reversing a signal in time, so that x(n)becomes x(−n).
   Folding a signal means reversing it in time. For a discrete signal x(n)x(n)x(n), its folded form is: y(n)=x(−n)

3. **Signal Shifting:** Signal shifting is the process of moving a signal forward or backward in time. A right shift (delay) is x(n−k) and a left shift (advance) is x(n+k).

   Shifting moves the signal forward or backward in time. For a shift by k:
   - Right shift (delay): y(n)=x(n−k)
   - Left shift (advance): y(n)=x(n+k)

Source code:

```
clear; clc; close all;

n = -5:5;

x1 = [0 0 1 2 3 4 5 3 2 1 0];

x2 = [0 0 2 2 2 2 2 2 2 0 0];


% 1) Addition

[y_add, n_add] = add_signals(x1, n, x2, n);


% 2) Folding (time reversal) of x1

[y_fold, n_fold] = fold_signal(x1, n);


% 3) Shifting x1 by k (right shift k>0)
```

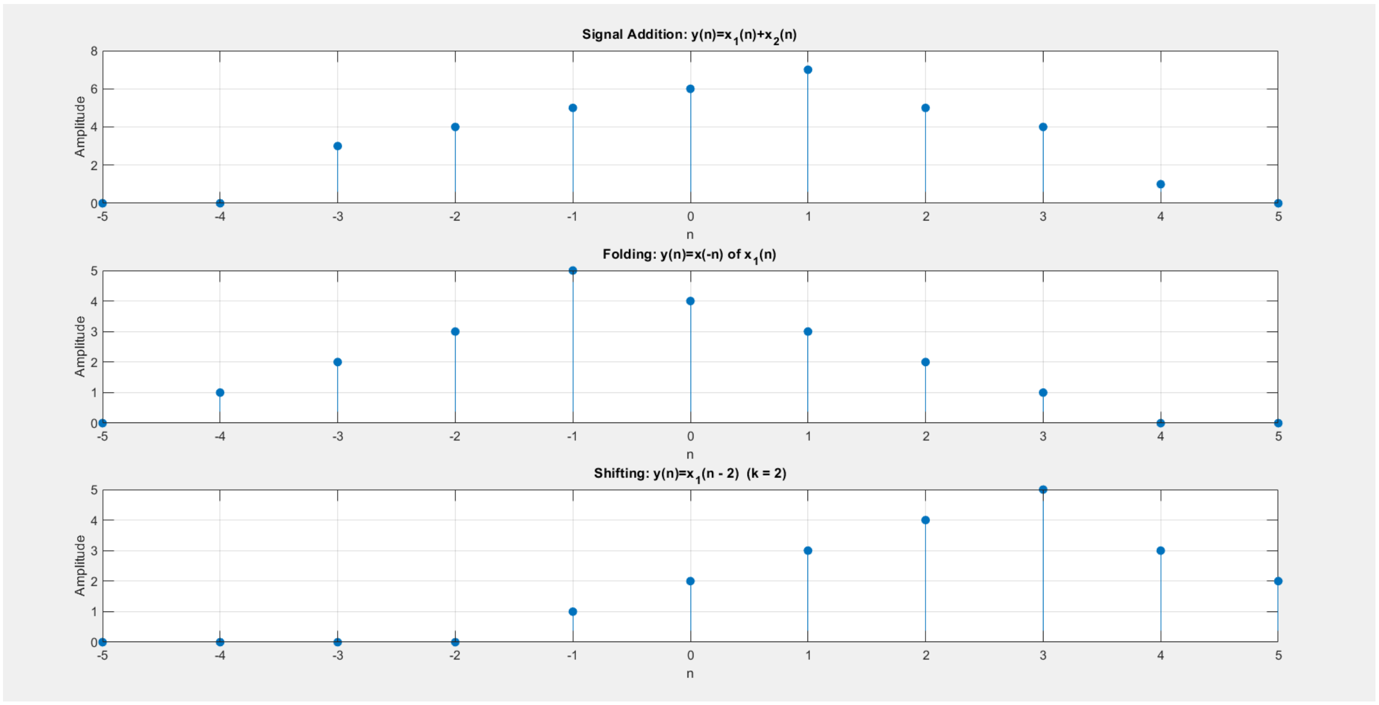
```
k = 2;  % delay by 2
[y_shift, n_shift] = shift_signal(x1, n, k);

% PLOTTING
figure('Name','Signal Operations','NumberTitle','off','Position',[200 200 800 700]);

subplot(3,1,1);
stem(n_add, y_add, 'filled'); grid on;
title('Signal Addition: y(n)=x_1(n)+x_2(n)');
xlabel('n'); ylabel('Amplitude');

subplot(3,1,2);
stem(n_fold, y_fold, 'filled'); grid on;
title('Folding: y(n)=x(-n) of x_1(n)');
xlabel('n'); ylabel('Amplitude');

subplot(3,1,3);
stem(n_shift, y_shift, 'filled'); grid on;
title(['Shifting: y(n)=x_1(n - ' num2str(k) ')  (k = ' num2str(k) ')']);
xlabel('n'); ylabel('Amplitude');

% Print vectors to command window for quick check
disp('n ='); disp(n);
disp('x1 ='); disp(x1);
disp('x2 ='); disp(x2);
disp('y_add = x1 + x2 ='); disp(y_add);
disp('y_fold = x1 folded (x(-n)) ='); disp(y_fold);
disp(['y_shift = x1 shifted by k = ' num2str(k) ' :']); disp(y_shift);

function [y, n_out] = add_signals(x1, n1, x2, n2)
```

```
    n_out = min(min(n1),min(n2)) : max(max(n1),max(n2));
    y1 = zeros(size(n_out)); y2 = zeros(size(n_out));
    offset = n_out(1);
    for i=1:length(n1)
        idx = n1(i) - offset + 1;
        y1(idx) = x1(i);
    end
    for i=1:length(n2)
        idx = n2(i) - offset + 1;
        y2(idx) = x2(i);
    end
    y = y1 + y2;
end

function [y, n_out] = fold_signal(x, n)
    n_out = n;
    y = zeros(size(n_out));
    for i = 1:length(n_out)
        target = -n_out(i);
        j = find(n == target, 1);
        if ~isempty(j)
            y(i) = x(j);
        else
            y(i) = 0;
        end
    end
end

function [y, n_out] = shift_signal(x, n, k)
    n_out = n;
    y = zeros(size(n_out));
    for i = 1:length(n_out)
        target = n_out(i) - k;
        j = find(n == target, 1);
        if ~isempty(j)
            y(i) = x(j);
        else
            y(i) = 0;
        end
    end
end
```

Input:

- n = -5:5
- x1 = [0 0 1 2 3 4 5 3 2 1 0]
- x2 = [0 0 2 2 2 2 2 2 2 0 0]
- k = 2 (shift amount)

Output:


Signal Addition: y(n)=x1(n)+x2(n)
Folding: y(n)=x(-n) of x1(n)
Shifting: y(n)=x1(n - 2) (k = 2)
Amplitude
n

Experiment No:05

Title: Explain and implementation of Convolution operation of sequences.

Objective(s):

- To understand the mathematical concept of convolution and its significance in signal processing and system analysis.
- To implement convolution of discrete sequences using MATLAB programming.
- To compare manual convolution calculations with MATLAB generated results for accuracy verification

Theory: Convolution is a mathematical operation of two functions. The first function is the input signal and the second signal is the impulse response. The output signal is a Linear time invariant system. In the field of engineering convolution is commonly used in digital signal processing, statistics, computer vision, differential equations etc. For discrete-time sequences, convolution of x[n] (input signal) and h[n] (impulse response of the system) is defined as:

$$Y[n]= x[n]*h[n] = \sum_{k=-\infty}^{\infty} x[k].h[n-k]$$

Here,

- x[n]: Input sequence
- h[n]: Impulse response of the system
- Y[n]: Output sequence (response of the system to input)

The extension of the sequences to support finite functions is gives the convolution of finite sequences. This experiment implements the convolution of a general discrete time signal and a general discrete time system. The formula means that each output value y[n] is obtained by shifting, multiplying, and summing the two sequences.

Procedure:

To perform convolution of two finite sequences x[n] and h[n]:

Step 1: Write the given sequences x[n] and h[n] along with their indices.

Step 2: Flip one sequence (usually h[n]) about the vertical axis to form h[−k].

Step 3: Shift the flipped sequen**ce** by n units to obtain h[n−k].

Step 4: Multiply overlapping terms of x[k] and h[n−k] .

Step 5: Sum the products → this gives one value of the output y[n].

Step 6: Repeat for all values of n, shifting h[n−k] step by step.

Step 7: Write down the final output sequence y[n].

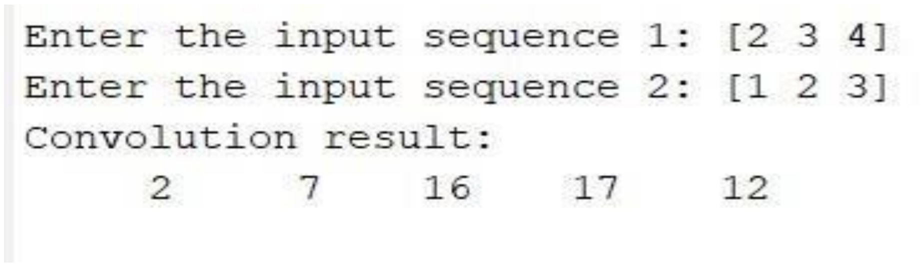
The output sequence will have length $L_x + L_h - 1$.

Code in MATLAB:

```
close all

clear all

clc
x = input('Enter the input sequence 1: ');    % sequence 1
h = input('Enter the input sequence 2: ');    % sequence 2

a = length(x);              % Number of elements in sequence 1
b = length(h);              % Number of elements in sequence 2
N = a + b - 1;             % Length of convolution output

% Zero padding to equal length
X = [x, zeros(1, N-a)];

H = [h, zeros(1, N-b)];

% Initialize output
Y = zeros(1, N);

% Convolution
calculation for i = 1:N
for j = 1:a
if (i-j+1 > 0)
Y(i) = Y(i) + X(j) * H(i-j+1);
end
   end
      end

% Plotting input and output
subplot(2,2,1)
stem(0:a-1, x,'filled');
grid on;
ylabel('x[n]');
xlabel('n');
title('Sequence x[n]');

subplot(2,2,2)
stem(0:b-1, h, 'filled');
```

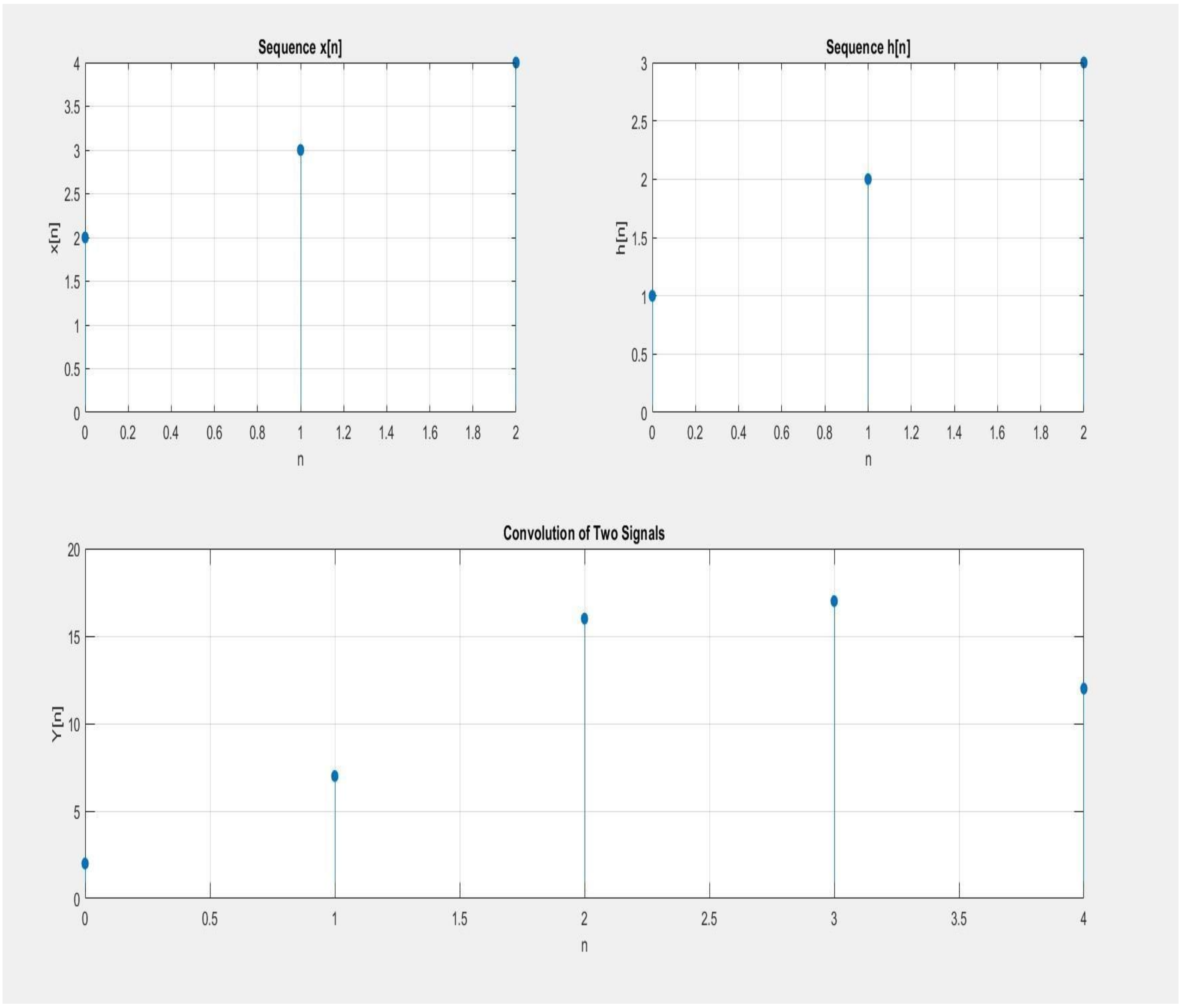
```
grid on;
ylabel('h[n]');
xlabel('n');
title('Sequence h[n]');

subplot(2,2,[3 4])
stem(0:N-1, Y, 'filled');
grid on;
ylabel('Y[n]');
xlabel('n');
title('Convolution of Two Signals');

disp('Convolution result:');
disp(Y);
```

Input:



```
Enter the input sequence 1: [2 3 4]
Enter the input sequence 2: [1 2 3]
Convolution result:
     2     7    16    17    12
```

MATLAB Output:



Experiment No:06

Experiment Name: Generate sinusoidal wave with different frequency using MATLAB.

Theory:

A sinusoidal wave is the most basic and important waveform in science and engineering. It represents a smooth and continuous oscillation which repeats after a fixed interval of time. Because of its simple and periodic nature, it is considered the foundation of signal and system analysis.

The general equation of a sinusoidal wave is:

$x(t) = A * \sin(2 * \pi * f * t + \varphi)$

Here,
A = Amplitude (maximum height of the wave)
f = Frequency in Hertz (number of cycles per second)
t = Time in seconds
φ = Phase (initial shift of the wave)

The frequency of the sinusoid controls how fast the signal oscillates. If the frequency is small, the wave will have fewer cycles within one second. On the other hand, if the frequency is high, the wave will complete more cycles in the same duration. Thus, the shape of the sinusoidal wave becomes compressed as frequency increases.

Sinusoidal signals are widely used in real life. Alternating current (AC) in power systems, audio and speech signals, and wireless communication signals can all be modeled using sinusoids. In fact, any complicated signal can be expressed as a combination of sinusoidal waves, which makes them extremely important in engineering applications.

MATLAB provides a very convenient platform to generate and observe sinusoidal waves. By defining amplitude, frequency, and time, different sinusoidal signals can be plotted. This helps us to understand practically how the frequency affects the nature of the waveform.

Procedure:

1. Clear data and set sampling frequency (fs = 1000).
2. Create time vector from 0 to 1 sec.
3. Choose frequencies 5 Hz, 10 Hz, and 20 Hz.
4. Generate sinusoidal signals using sin(2*pi*f*t).
5. Plot the three waves using subplot with proper titles and labels.
6. Observe that higher frequency produces more cycles in the same time.

Source code in MATLAB:

```
clc;

clear;

close all;



fs = 1000;

t = 0:1/fs:1;



f1 = 5;

f2 = 10;

f3 = 20;



y1 = sin(2*pi*f1*t);

y2 = sin(2*pi*f2*t);

y3 = sin(2*pi*f3*t);



figure;
```

```
subplot(3,1,1);

plot(t, y1, 'b');

title('Sinusoidal Wave with Frequency 5 Hz');

xlabel('Time (s)');

ylabel('Amplitude');


subplot(3,1,2);

plot(t, y2, 'r');

title('Sinusoidal Wave with Frequency 10 Hz');

xlabel('Time (s)');

ylabel('Amplitude');


subplot(3,1,3);

plot(t, y3, 'g');

title('Sinusoidal Wave with Frequency 20 Hz');

xlabel('Time (s)');

ylabel('Amplitude');
```

Output

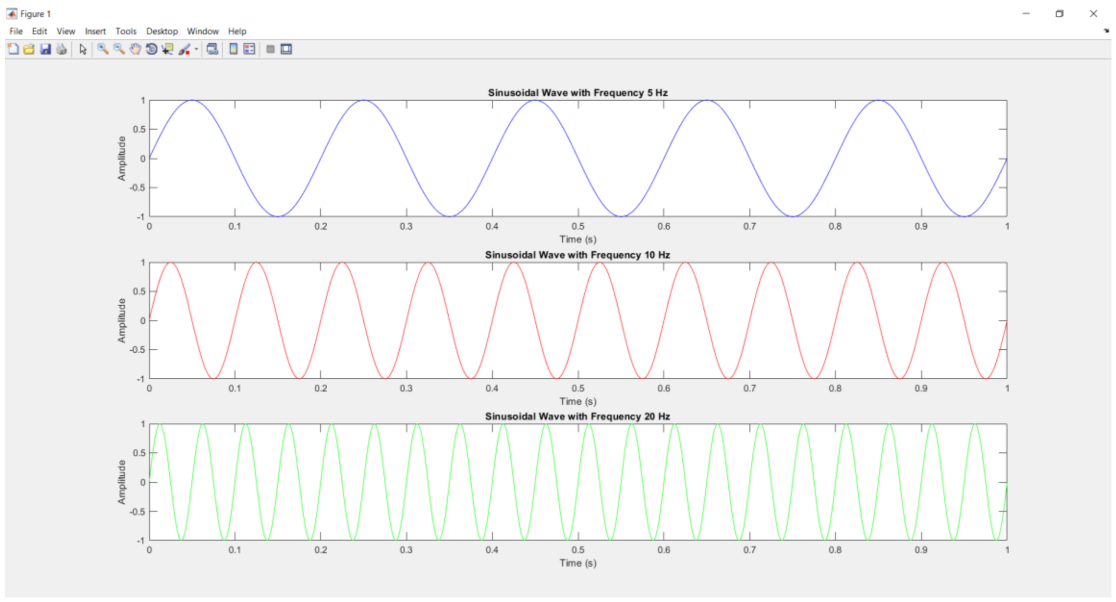


Figure 1
File Edit View Insert Tools Desktop Window Help
Sinusoidal Wave with Frequency 5 Hz
Sinusoidal Wave with Frequency 10 Hz
Sinusoidal Wave with Frequency 20 Hz
Amplitude
Time (s)
1
0.5
0
-0.5
-1
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

Experiement No:07

Experiment name : Explanation and implementation Discrete Fourier Transform(DFT) and Inverse Discrete Fourier Transform(IDFT) using MATLAB.

Theory : In signal processing, signals can be represented either in the time domain or in the frequency domain. In the time domain, a signal is described as it varies with time. Time domain signals are of two types: continuous-time signals and discrete-time signals. A continuous-time signal is defined for every instant of time. A discrete-time signal is defined only at specific intervals of time.

DFT :

The Discrete Fourier Transform (DFT) is a mathematical technique that converts a finite-length discrete-time signal into its equivalent frequency domain representation. In other words, DFT analyzes a signal and shows which frequency components are present and how strong they are in terms of magnitude and phase. This is very important in digital signal processing, because many applications such as filtering, compression, and spectrum analysis require frequency information rather than time domain values.

Mathematically, the DFT of a sequence x[n] of length N is given by

$$X[k] = \sum_{n=0}^{N-1} x(n) * e^{-j2\pi kn/N} \qquad \textit{where } k = 0,1,2,\ldots,N\text{-}1$$

Here,

* x[n] is the discrete-time signal in time domain,

*X[k] is the frequency domain sequence,

* N is the number of samples, and j is the imaginary unit.

Each X[k] represents the contribution of a sinusoidal component of frequency (2πk/N) in the original signal.

IDFT :

The Inverse Discrete Fourier Transform (IDFT) is used to transform the frequency domain data back into the time domain. The IDFT is defined as

$$x([n] = \frac{1}{N}\sum_{k=0}^{N-1} X[k]e^{\frac{j2\pi kn}{N}} \qquad where\ n = 0,1,2 \ldots\ldots N-1$$

Thus, DFT and IDFT are mathematical inverses of each other. DFT converts time domain sequences into frequency domain representation, while IDFT reconstructs the original sequence from its frequency domain samples.

It is important to note that DFT can only be applied to discrete-time signals of finite length. For continuous-time signals, sampling is first performed to obtain a discrete sequence, and then the DFT is applied. In practice, the direct computation of DFT is computationally expensive (order of N^2 operations), so Fast Fourier Transform (FFT) algorithms are used, but in this experiment we focus on the basic DFT and IDFT definitions.

Source code :

DFT :

```
clc
clear all
close all
Fa=input('Enter a frequency Fa : ');
Fs=input('Enter a frequency Fs : ');
T=1/Fa;
t=0:T/99:T;
y=5*sin(2*pi*Fa*t) +2*sin(2*pi*2*Fa*t)+2*sin(2*pi*3*Fa*t);
figure(1)
plot(t,y);
Ts=1/Fs;
N=T/Ts;
n=0:N-1;
yy=5*sin(2*pi*Fa*n*Ts)+ 2*sin(2*pi*2*Fa*n*Ts)+ 2*sin(2*pi*3*Fa*n*Ts);
figure(2)
stem(n,yy)
h=[];
b=[];
for(k=1:1:N)
 for (n=1:1:N)
 ff=yy(n)*exp(-j*2*pi*(k-1-(N/2))*(n-1-(N/2))/N);
 h=[ h ff];
 end
 p=sum(h);
 b=[b p];
 h=0;
end
figure(3)
f=Fs*(-N/2:(N/2)-1)/N;
stem(f,abs(b));
axis([ -30 30 0 160]);
```

Sample input:

```
Enter a frequency Fa : 10
Enter a frequency Fs : 640
```

Output:

figure 1:

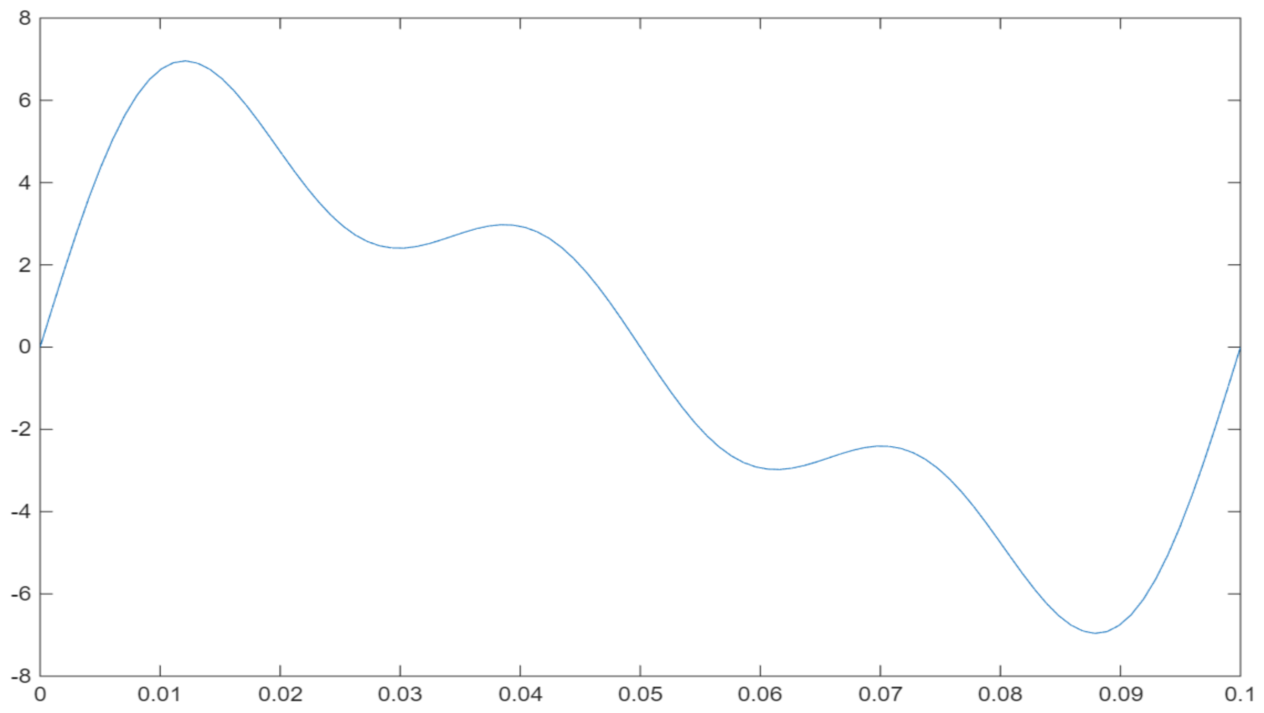


Figure 2:

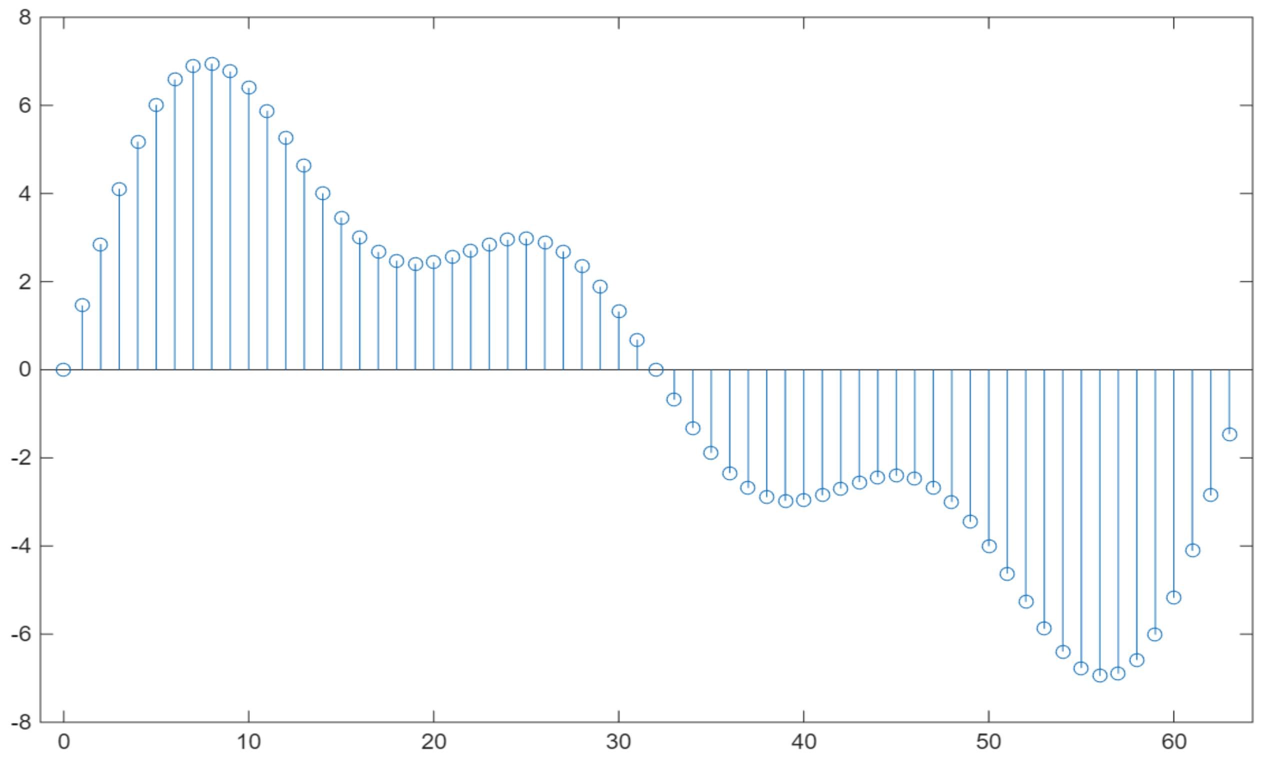
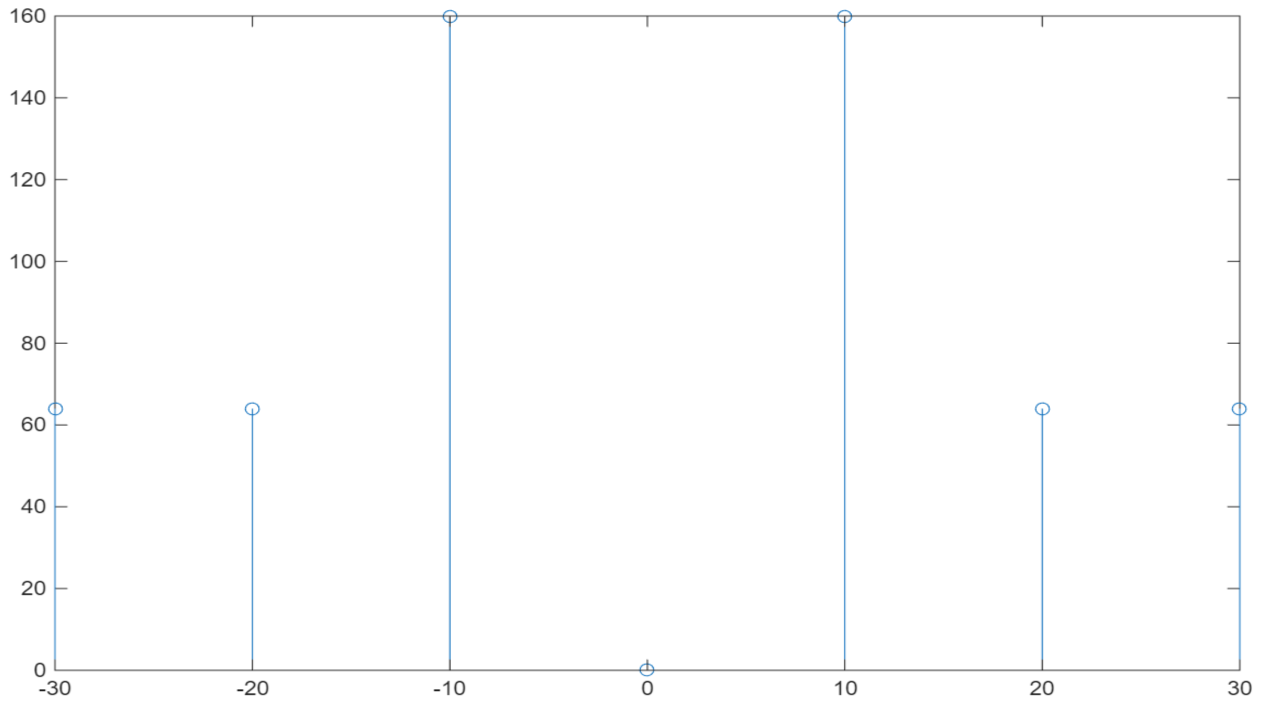


Figure 3:



Source code:

IDFT:

```
clc;
close all;
clear all;
%IDFT of a sequence
Xk=input('Enter X(K)=');
[N,M]=size(Xk);
if M~=1;
 Xk=Xk.';
 N=M;
end;
xn=zeros(N,1);
k=0:N-1;
for n=0:N-1;
xn(n+1)=exp(j*2*pi*k*n/N)*Xk;
end;
xn=xn/N;
disp('x(n)=');
disp(xn);
plot(xn);
grid on;
plot(xn);
stem(k,xn);
xlabel('....>n');
ylabel('........>Magnitude');
title('IDEF OF A SEQUENCE');
```

Sample input:

```
Enter X(K)=3-3j
Enter X(K)=4
```

Sample output:

```
x(n)=
   3.0000 - 3.0000i

x(n)=
     4
```

Figure 1 (when input = 3-3j) :

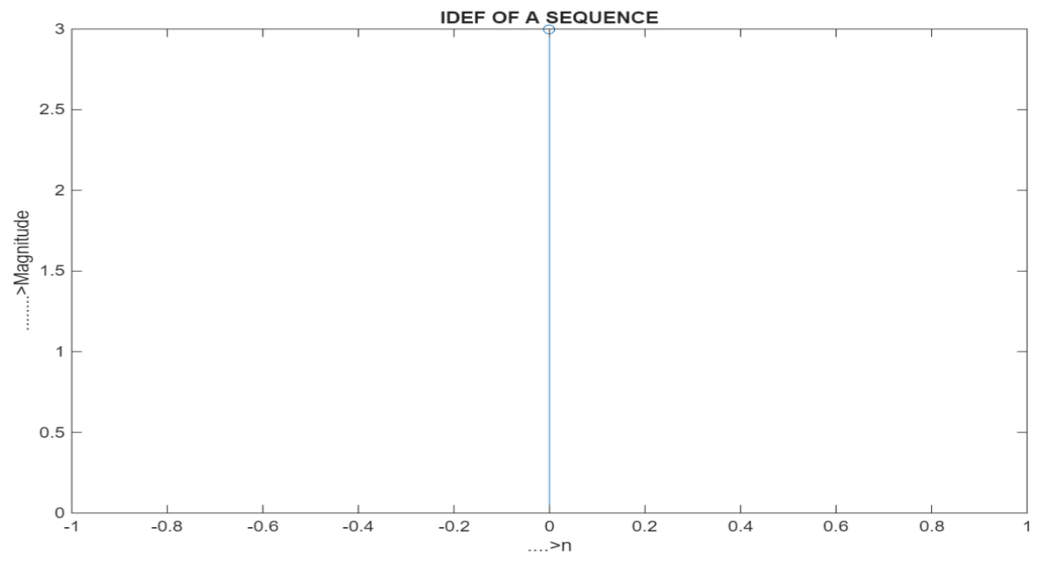
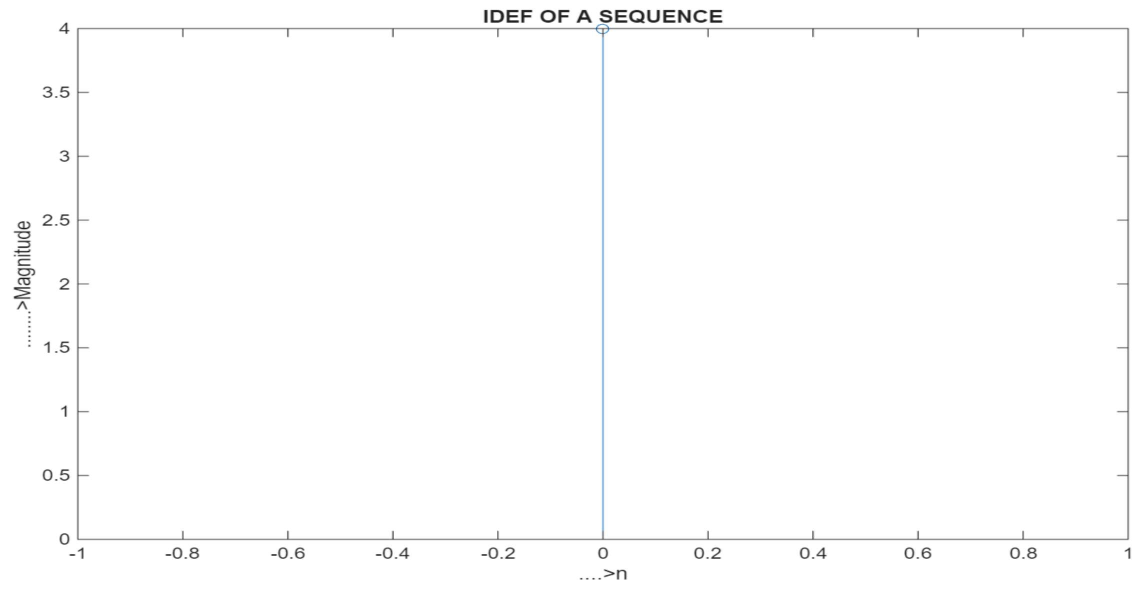


Figure 2 (when input=4):



Experiment No:08

Experiment name : Finding the spectrum of the following signal:
$x(k)=0.25+2\sin(2\pi\cdot5k)+\sin(2\pi\cdot12.5k)+1.5\sin(2\pi\cdot20k)+0.5\sin(2\pi\cdot35k)$

Theory : The given signal is:

$x(k)=0.25+2\sin(2\pi\cdot5k)+\sin(2\pi\cdot12.5k)+1.5\sin(2\pi\cdot20k)+0.5\sin(2\pi\cdot35k)$

Since this is a discrete-time signal, the argument 2πfk of each sine term is always an integer multiple of π\pi for integer k. In such cases, sin(nπ)=0\sin(n\pi) = 0, which means that all sinusoidal terms vanish for integer values of k. As a result, the entire signal reduces to a constant value of x(k)=0.25x(k) = 0.25. In the frequency domain, the Discrete-Time Fourier Transform (DTFT) of a constant sequence produces only a DC component, represented as an impulse at ω=0. Therefore, the spectrum of the given signal consists solely of a single impulse at DC with strength 0.25·2π , and no other frequency components are present.

Property of Sine in Discrete Time

In discrete time, if the argument of sine is a multiple of 2π, then:

$$\sin(2\pi fk)=0, \qquad \forall k\in Z, \quad \text{if } f\in Z.$$

Simplified Signal

So, every sinusoidal term disappears at integer k.
Thus, the discrete-time signal reduces to:

$$x(k) = 0.25, \qquad \forall k.$$

It is just a constant sequence**.**

Spectrum of a Constant Sequence

We know the **Discrete-Time Fourier Transform (DTFT):**

$$X(e^{jw}) \;=\; \sum_{k=-\infty}^{\infty} \mathrm{x(k)} e^{-jwk}$$

If x(k) = C (a constant), then:

$$X(e^{jw}) \;= C.2\pi \sum_{m=-\infty}^{\infty} \delta(\omega - 2\pi m)$$

So the spectrum of a constant is a single impulse at DC (frequency 0), with strength proportional to the constant.

For Our Signal

Since x(k) = 0.25:

$$X(e^{jw}) \;= 0.25\;.2\pi \sum_{m=-\infty}^{\infty} \delta(\omega - 2\pi m)$$

That means:

Only DC component exists.

All other frequency components vanish.

Final Result

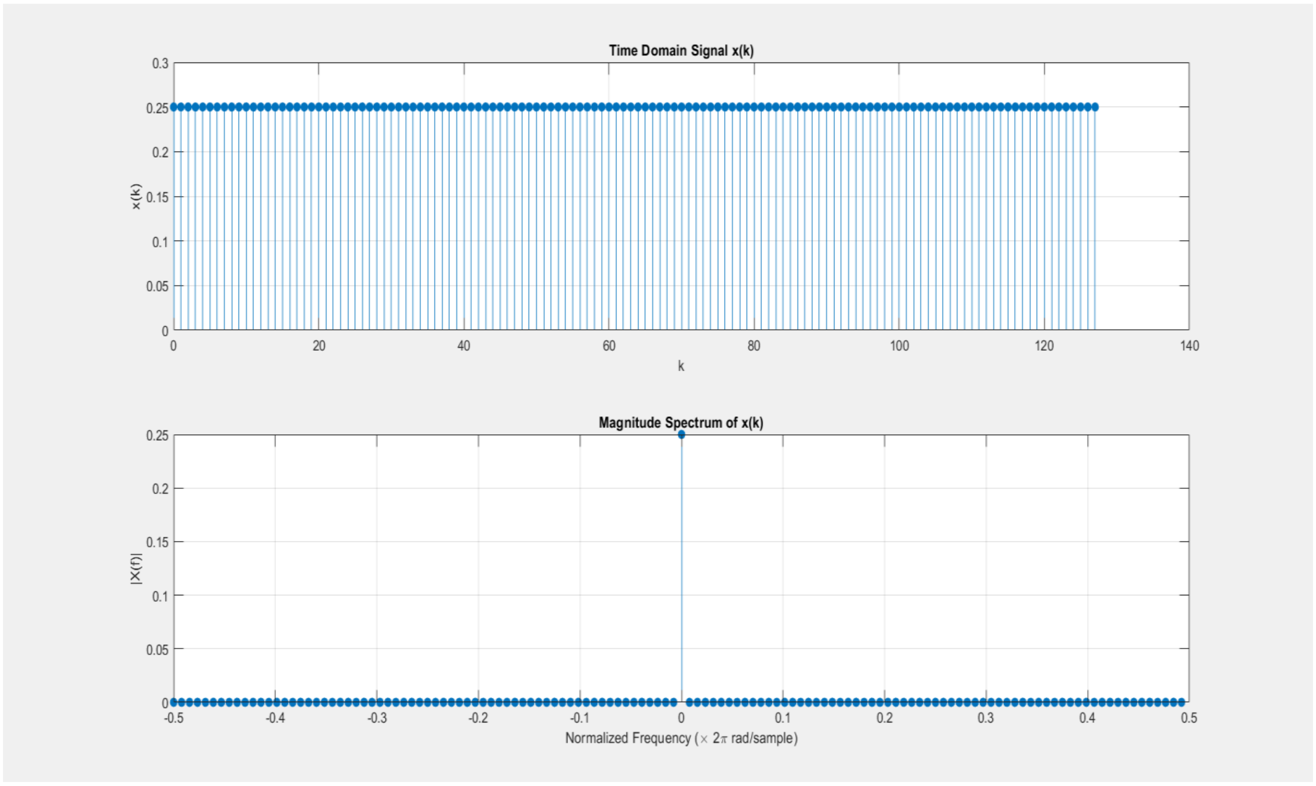
Time-domain signal: constant 0.25.

Spectrum: only DC impulse at **ω=0**, no other frequencies present.

Source code :

```
clc;
clear all;
close all;
N = 128;
k = 0:N-1;
x = 0.25 + 2*sin(2*pi*5*k) + sin(2*pi*12.5*k) +
... 1.5*sin(2*pi*20*k) + 0.5*sin(2*pi*35*k);
X = fft(x, N);
X_shifted = fftshift(X);
X_mag = abs(X_shifted)/N;
f = (-N/2:N/2-1)/N;
figure;
subplot(2,1,1);
stem(k, x, 'filled');
title('Time Domain Signal x(k)');
xlabel('k');
ylabel('x(k)');
grid on;
subplot(2,1,2);
stem(f, X_mag, 'filled');
title('Magnitude Spectrum of x(k)');
xlabel('Normalized Frequency (\times 2\pi
rad/sample)');
ylabel('|X(f)|');
grid on;
```

Output:


Time Domain Signal x(k)
x(k)
k
0.3
0.25
0.2
0.15
0.1
0.05
0
0
20
40
60
80
100
120
140
Magnitude Spectrum of x(k)
|X(f)|
Normalized Frequency (× 2π rad/sample)
0.25
0.2
0.15
0.1
0.05
0
-0.5
-0.4
-0.3
-0.2
-0.1
0
0.1
0.2
0.3
0.4
0.5

Experiment No : 09

Experiment Name : Removing noise from an audio signal.

Theory:

Noise is any unwanted or random disturbance that interferes with a signal, making it less clear or harder to interpret.

The DFT (Discrete Fourier Transform) is a mathematical method used to convert a signal from the time domain (how it changes over time) into the frequency domain (what frequencies it is made up of).

And IDFT (Inverse Discrete Fourier Transform) is the mathematical process that converts a signal from the frequency domain back to the time domain.

This experiment demonstrates how noise can be removed from a signal using frequency-domain filtering. A pure sine wave of 440 Hz is generated as the original signal. Random noise is added to simulate real-world interference. The Discrete Fourier Transform (DFT) is then applied to convert the signal from the time domain to the frequency domain, where the sine wave appears as a sharp peak at 440 Hz while noise spreads across other frequencies. By applying a filter to suppress high-frequency components, most of the noise is removed. Finally, the Inverse DFT (IDFT) is used to reconstruct the cleaned signal back in the time domain. This shows how Fourier analysis can separate useful signals from noise effectively.

Objectives :

1. To generate a pure audio signal.

2. To simulate real-world noisy conditions by adding random noise to the original signal.

3. To apply the Discrete Fourier Transform (DFT) and analyze how the signal and noise appear in the frequency spectrum.

4. To design and implement a filtering technique that removes high-frequency noise components while preserving the fundamental audio signal.

5. To reconstruct the signal using the Inverse DFT (IDFT) and compare the cleaned signal with the original pure signal.

6. To evaluate the effectiveness of frequency-domain filtering in reducing noise while maintaining audio quality

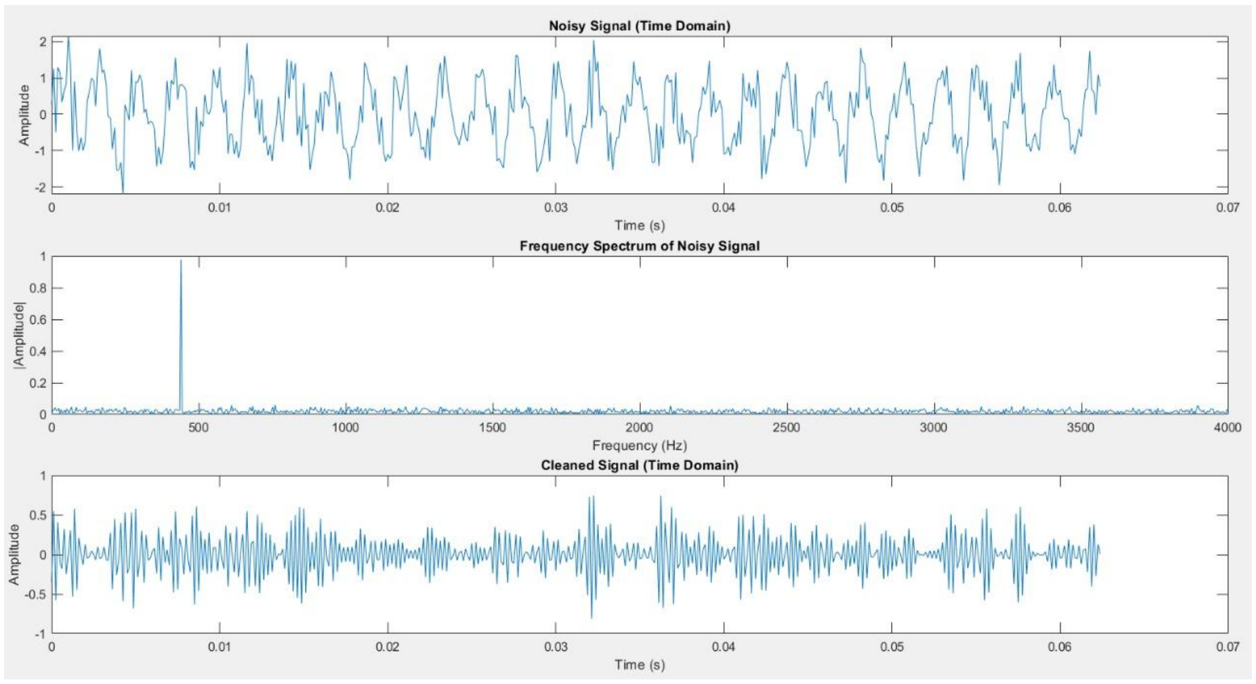
Source Code In MATLAB :

```
clc;

clear all;

close all;


% Step a: Generate an audio signal (pure sine wave, 440 Hz)

Fs = 8000;            % Sampling frequency (Hz)

T = 1/Fs;             % Sampling period

L = 2000;             % Length of signal (number of samples)

t = (0:L-1)*T;        % Time vector

f = 440;              % Frequency of sine wave (Hz)

x = sin(2*pi*f*t);    % Pure sine wave


% Step b: Add random noise

noisy_signal = x + 0.5*randn(size(t));
```

```
% Step c: Apply DFT (FFT in MATLAB)
Y = fft(noisy_signal);

% Frequency axis for plotting
f_axis = Fs*(0:(L/2))/L;

% Step d: Filter high frequencies
% Design a simple low-pass filter by zeroing out frequencies above 1000 Hz
cutoff = 1000;   % Hz
Y_filtered = Y;
Y_filtered(abs((0:L-1)-L/2) > cutoff*L/Fs) = 0;  % crude filtering

% Step e: Apply Inverse DFT
cleaned_signal = ifft(Y_filtered, 'symmetric');

% -------- Plot Results --------
figure;

subplot(3,1,1);
plot(t(1:500), noisy_signal(1:500));
title('Noisy Signal (Time Domain)');
xlabel('Time (s)');
ylabel('Amplitude');
```

```
subplot(3,1,2);
P2 = abs(Y/L);
P1 = P2(1:L/2+1);
P1(2:end-1) = 2*P1(2:end-1);
plot(f_axis, P1);
title('Frequency Spectrum of Noisy Signal');
xlabel('Frequency (Hz)');
ylabel('|Amplitude|');

subplot(3,1,3);
plot(t(1:500), cleaned_signal(1:500));
title('Cleaned Signal (Time Domain)');
xlabel('Time (s)');
ylabel('Amplitude');

% Optional: Play sounds
% sound(x, Fs);            % Original sine wave
% pause(2);
% sound(noisy_signal, Fs);  % Noisy signal
% pause(2);
% sound(cleaned_signal, Fs);% Cleaned signal
```

Output :


Noisy Signal (Time Domain)
Amplitude
Time (s)
Frequency Spectrum of Noisy Signal
|Amplitude|
Frequency (Hz)
Cleaned Signal (Time Domain)
Amplitude
Time (s)

Experiment No. 10

Experiment Name

Extract Relevant Features such as Systolic Peaks, Diastolic Points, and Heart Rate from PPG Signal

---

Theory

Photoplethysmography (PPG) is a non-invasive optical technique to measure blood volume changes in the microvascular bed of tissue. PPG signals are commonly used to monitor heart rate and cardiovascular health.

Key features of a PPG waveform include:

- **Systolic Peak:** Maximum amplitude point, representing blood flow during heart contraction.
- **Diastolic Point:** Minimum amplitude point between systolic peaks, representing relaxation.
- **Heart Rate (HR):** Calculated from time differences between consecutive systolic peaks (RR intervals).

Extracting these features helps in detecting abnormalities and monitoring vital signs.

Algorithm (Pseudo Code)

```
BEGIN

1. Input:
    o Sampling frequency (fs)
    o PPG signal (simulate or load)
2. Detect Systolic Peaks:
    o Use peak detection algorithm with minimum prominence and distance
    o Store systolic peak indices and times
3. Detect Diastolic Points:
    o For each interval between consecutive systolic peaks:
        ▪ Extract the signal segment
        ▪ Invert the segment to detect local minima
        ▪ Record the first minimum as the diastolic point
4. Calculate Heart Rate:
    o Compute RR intervals between consecutive systolic peaks
    o Convert RR intervals to BPM: $HR = 60 / RR$
5. Plot Results:
    o Plot PPG signal
    o Mark systolic peaks in red
```

    - Mark diastolic points in green
    - Annotate heart rate values near systolic peaks in magenta
6. Display Heart Rate:
    - Print BPM values in console

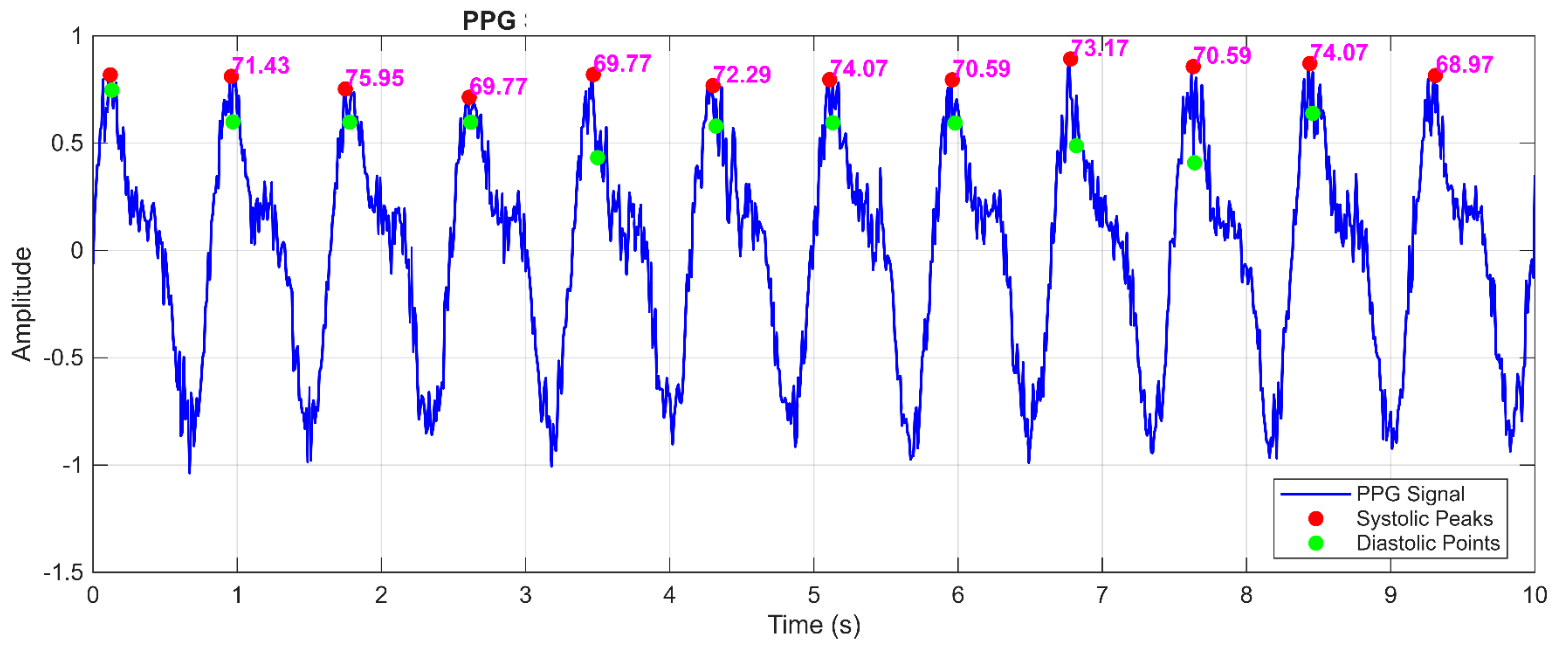
    END

Source Code (MATLAB)

```
% --- Step 1: Simulate or load PPG signal ---
fs = 100; % Sampling frequency in Hz
t = 0:1/fs:10; % 10 seconds of data
ppg = 0.6*sin(2*pi*1.2*t) + 0.3*sin(2*pi*2.4*t + 0.5) + 0.1*randn(size(t)); %
Synthetic PPG

% --- Step 2: Detect systolic peaks ---
[systolic_peaks, systolic_locs] = findpeaks(ppg, ...
    'MinPeakProminence', 0.3, ...
    'MinPeakDistance', round(0.5 * fs));
systolic_times = systolic_locs / fs;

% --- Step 3: Detect diastolic points ---
diastolic_locs = [];
for i = 1:length(systolic_locs)-1
    segment = ppg(systolic_locs(i):systolic_locs(i+1));
    segment_indices = systolic_locs(i):systolic_locs(i+1);

    % Invert segment to find local minima
    [~, local_min_locs] = findpeaks(-segment);

    if ~isempty(local_min_locs)
        diastolic_locs(end+1) = segment_indices(local_min_locs(1));
    end
end
diastolic_times = diastolic_locs / fs;

% --- Step 4: Calculate heart rate ---
rr_intervals = diff(systolic_times); % Time between systolic peaks
heart_rate = 60 ./ rr_intervals;     % BPM

% --- Step 5: Plot results ---
figure;
plot(t, ppg, 'b', 'LineWidth', 1.2);
hold on;
plot(systolic_locs/fs, ppg(systolic_locs), 'ro', 'MarkerFaceColor', 'r');
plot(diastolic_times, ppg(diastolic_locs), 'go', 'MarkerFaceColor', 'g');

% Annotate heart rate near each systolic peak
for i = 1:length(heart_rate)
    text(systolic_times(i+1), systolic_peaks(i+1)+0.05, ...
        num2str(round(heart_rate(i),2)), 'Color','m', 'FontSize',10,
'FontWeight','bold');
```

```
end

title('PPG Signal with Systolic, Diastolic Peaks, and Heart Rate');
xlabel('Time (s)');
ylabel('Amplitude');
legend('PPG Signal', 'Systolic Peaks', 'Diastolic Points', 'Location','best');
grid on;

% --- Step 6: Display heart rate ---
disp('Heart Rate (BPM) between beats:');
disp(round(heart_rate, 2));
```

Output (Figure):



Output (Console):

```
Heart Rate (BPM) between beats:
   59.88   61.54   60.00   62.50   57.14   60.61
```

Note:

Since the PPG signal is generated with random noise, the heart rate values displayed in the console will vary slightly each time the program is run.

Experiment No: 11

Experiment Name: Show Power Density Spectrum of a Square Wave.

Theory:

Power Density Spectrum (PDS)

- The Power Density Spectrum describes how the power of a signal is distributed across different frequencies.
- It is obtained by taking the Fourier Transform (FT) of the autocorrelation function of the signal, or equivalently by computing the squared magnitude of its Fourier Transform:

$$s(f) = |X(f)|^2$$

Where,

- $X(f)$ is the Fourier Transform of the signal $x(t)$,
- $S(f)$ is the Power Spectral Density.

Square Wave Signal

- A square wave is a periodic signal that alternates between two amplitude levels (say +A and -A).
- Its Fourier series expansion shows that it can be expressed as a sum of odd harmonics of the fundamental frequency.

For a square wave of amplitude A and fundamental frequency f_0:

$$x(t) = \frac{4A}{\pi}\left[\sin(2\pi f_0 t) + \frac{1}{3}\sin(3.2\pi f_0 t) + \frac{1}{5}\sin(5.2\pi f_0 t) + \cdots\right]$$

Power Spectrum of a Square Wave

- From the Fourier expansion, only odd harmonics are present in the spectrum.

- The amplitude of the $n\ th$ harmonic (where n =1,3,5, …) is:
  - $A_n = \frac{4A}{n\pi}$
- Therefore, the power contribution of each harmonic is proportional to ${A_n}^2$:

$$P_n \propto (\frac{4A}{n\pi})^2$$

- This shows that higher harmonics have progressively lower power.

Expected Power Density Spectrum

- The PSD of a square wave consists of discrete spikes at odd multiples of the fundamental frequency f_0.
- The magnitude of these spikes decreases as $1/n^2$, meaning the first harmonic carries most of the power while higher harmonics contribute less.

Sketch of PDS (Qualitative):

- A strong line at f_0., weaker lines at $3f_0$, $5f_0$, $7f_0$, ….
- No components at even multiples of f_0.

Objective :

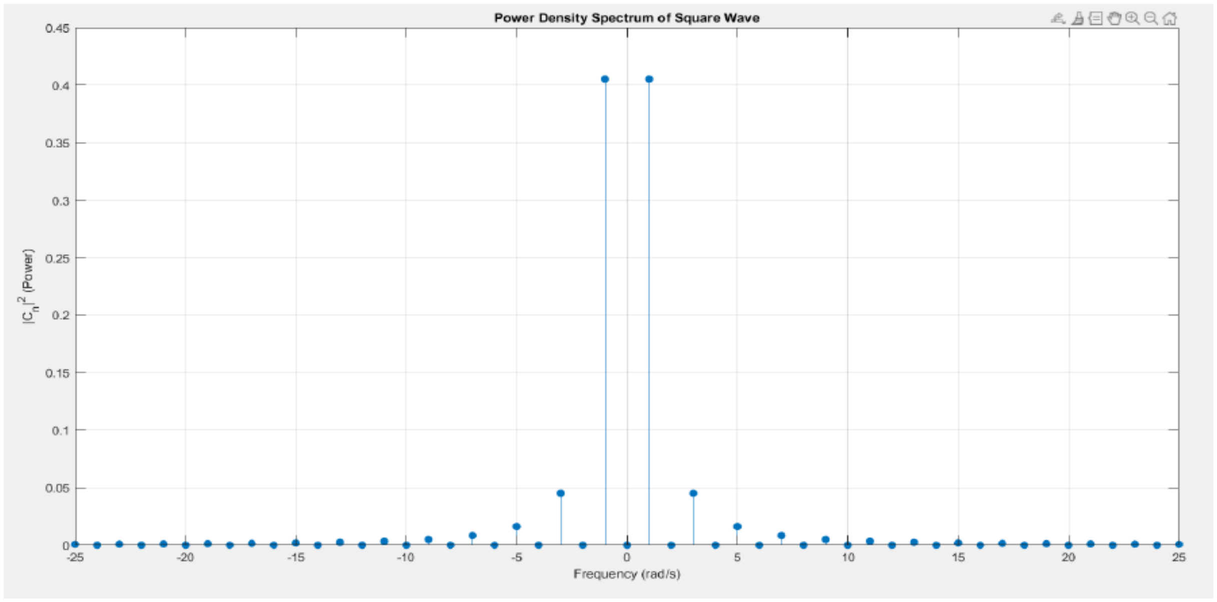
- To generate a square wave signal and analyze its frequency-domain characteristics.
- To study the Power Density Spectrum (PDS) of a square wave and verify that it consists of odd harmonics with decreasing amplitudes.
- To understand how time-domain periodic signals relate to their frequency-domain representation using Fourier analysis.

Power Density Spectrum of a Square Wave :

```
clc;
clear;
close all;
```

```
A = 1;  % Amplitude
T = 2*pi;  % Period
w0 = 2*pi/T;  % Fundamental frequency
N = 25;  % Number of harmonics

#Fourier coefficients
n = -N:N;
Cn = zeros(size(n));
for k = 1:length(n)
if mod(n(k),2) ~= 0  # only odd harmonics
Cn(k) = 2*A/(1j*n(k)*pi);
else
Cn(k) = 0;
end
end

#Power spectrum (|Cn|^2)
P = abs(Cn).^2;

#Plot line spectrum
stem(n*w0, P, 'filled');
xlabel('Frequency (rad/s)');
ylabel('|C_n|^2 (Power)');
title('Power Density Spectrum of Square Wave');
grid on;
```

Output:



Procedure :

1. Open MATLAB and create a new script file.
2. Define the parameters:

    - Amplitude $A = 1$,
    - Period $T = 2\pi$,
    - Fundamental frequency $w_0 = \frac{2\pi}{T}$,
    - Number of harmonics $N = 25$.

3. Generate the set of harmonics by considering $n = -N{:}N$.
4. Compute the Fourier series coefficients C_n:

    - For odd n, use

$$C_n = \frac{2A}{jn\pi}$$

    - For even n, set $C_n = 0$.

5. Calculate the power spectrum as

$$P = |C_n|^2$$

6. Plot the line spectrum using `stem(n*w0, P)` to represent power at discrete harmonic frequencies.
7. Label the x-axis as Frequency (rad/s) and y-axis as power $|C_n|^2$.

Observation :

- The spectrum consists of discrete spikes at odd harmonics of the fundamental frequency w_0.
- No spectral components are present at even multiples of the fundamental frequency.
- The amplitude (and thus power) decreases with increasing harmonic order.
- The first harmonic (fundamental) carries the maximum power, while higher harmonics rapidly diminish in strength.
- The plotted spectrum is symmetric for positive and negative frequencies, which is expected for real signals like the square wave.

Experiment No:12

Experiment Name: Show Fourier series approximation of square wave:

$$x(t) = \frac{4}{\pi}\left(sin(\omega_0 t) + \frac{1}{3}sin(3\omega_0 t) + \frac{1}{5}sin(5\omega_0 t) + \cdots\right)$$

Theory:

Fourier Series

A Fourier Series expresses any periodic signal x(t) as a sum of sine and cosine functions (or complex exponentials) of different frequencies.

For a periodic signal x(t) with period T:

$$x(t) = a_0 + \sum_{n=1}^{\infty}(a_n \cos(n\omega_0 t) + b_n \sin(n\omega_0 t))$$

where:

$\omega_0 = \frac{2\pi}{T}$ is the fundamental angular frequency

a_n *and* b_n are Fourier coefficients.

Square Wave Fourier Series:

A square wave is an odd, periodic function. Its Fourier series contains only odd harmonics (sine terms) because it is odd symmetric:

$$x(t) = \frac{4}{\pi}\left(sin(\omega_0 t) + \frac{1}{3}sin(3\omega_0 t) + \frac{1}{5}sin(5\omega_0 t) + \cdots\right)$$

The amplitude of each harmonic decreases with 1/k, where k= 1, 3, 5,…(odd numbers).

Increasing the number of harmonics N improves the approximation of the square wave.

Objective:

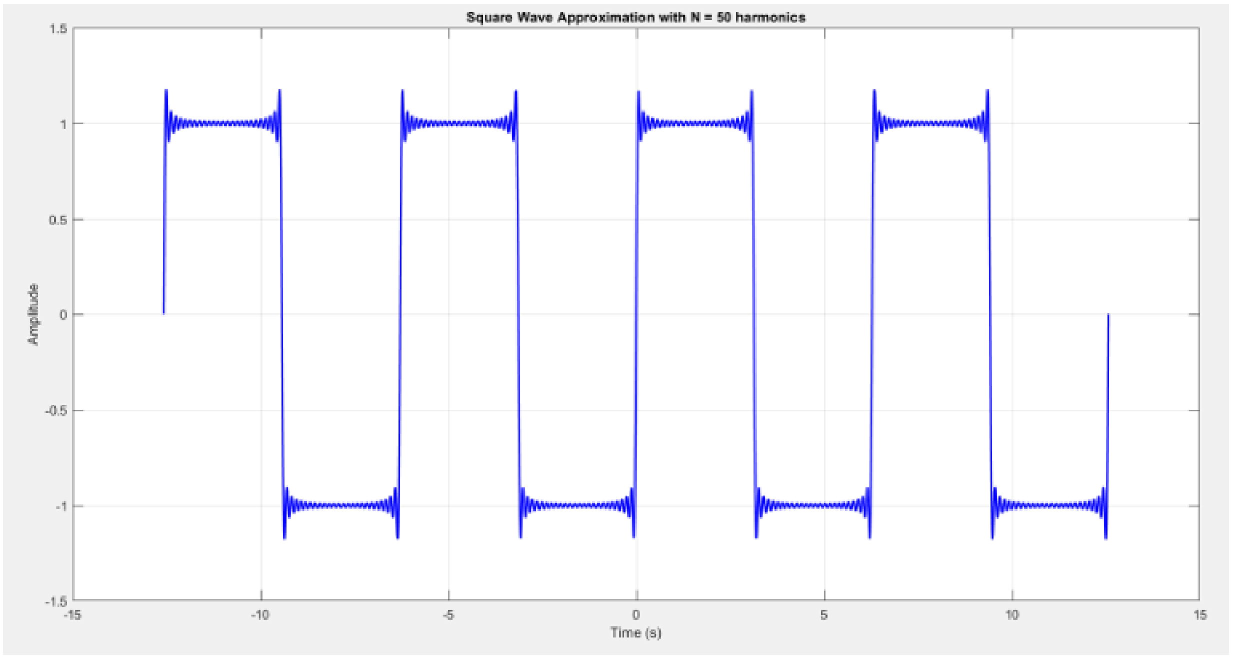
1. To understand the concept of Fourier Series and how it can be used to approximate periodic signals.
2. To approximate a square wave using its Fourier series representation.
3. To implement a MATLAB program to plot the Fourier series approximation of a square wave.
4. To observe the effect of the number of harmonics (N) on the accuracy of the square wave approximation.

Source Code:

```
% Square Wave Approximation using Fourier Series

 clc;

clear;

 close all;

 % Parameters

 T = 2*pi;  % Period %

 w0 = 2*pi/T;    % Fundamental frequency

t = linspace(-2*T, 2*T, 2000);  % Time axis

 % Fourier Series Approximation

N = 50; % Number of harmonics

x_approx = zeros(size(t));
```

```
 for k = 1:2:N % only odd harmonics
 x_approx = x_approx + (4/pi)*(1/k)*sin(k*w0*t);
 end
% Plot
plot(t, x_approx, 'b', 'LineWidth', 1.5);
 xlabel('Time (s)');
ylabel('Amplitude');
t itle(['Square Wave Approximation with N = ' num2str(N) ' harmonics']);
grid on;
 ylim([-1.5 1.5]);
```

Output:



Experiment No: 13

Experiment Name: To find the amplitude of the multi frequency signal:

$$X(t)= \cos(2\pi 100t) + \cos(2\pi 500t) +\cos(2\pi 700t)$$

And also show approximate the Fourier transform integral for $0 \le f \le 900$ Hz.

Objectives:

- To demonstrate how multi -frequency signals are represented in the frequency domain **,** with impulses at each frequency component
- To gener ate a multi -frequency signal composed of several cosine components at different frequencies.
- To restrict the spectrum to the desired frequency range (e.g., 0 –900 Hz) for easier analysis of practical signals.

Theory:

The Fourier Transform (FT**)** is a mathematical tool that converts a time-domain signal into its frequency-domain representation, showing how the signal can be expressed as a sum of sinusoids of different frequencies. For a continuous-time signal x(t)x(t)x(t), the Fourier Transform is defined as

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-j2\pi ft}\, dt$$

With the inverse transform

$$X(t) = \int_{-\infty}^{\infty} X(f)e^{j2\pi ft}\, df$$

When a signal is composed of multiple frequency components (a multi-frequency signal), its spectrum consists of impulses (delta functions) at each frequency where energy is present. For example, a signal such as

$$x(t)=\cos(2\pi f1t) + \cos(2\pi f2t) + \cos(2\pi f3t)$$

has a Fourier Transform that produces impulses at f=±f1, ±f2, ±f3f, each with amplitude 1/2. In general, the Fourier Transform of a single cosine signal

$$x(t)=A \cos(2\pi f0t)$$

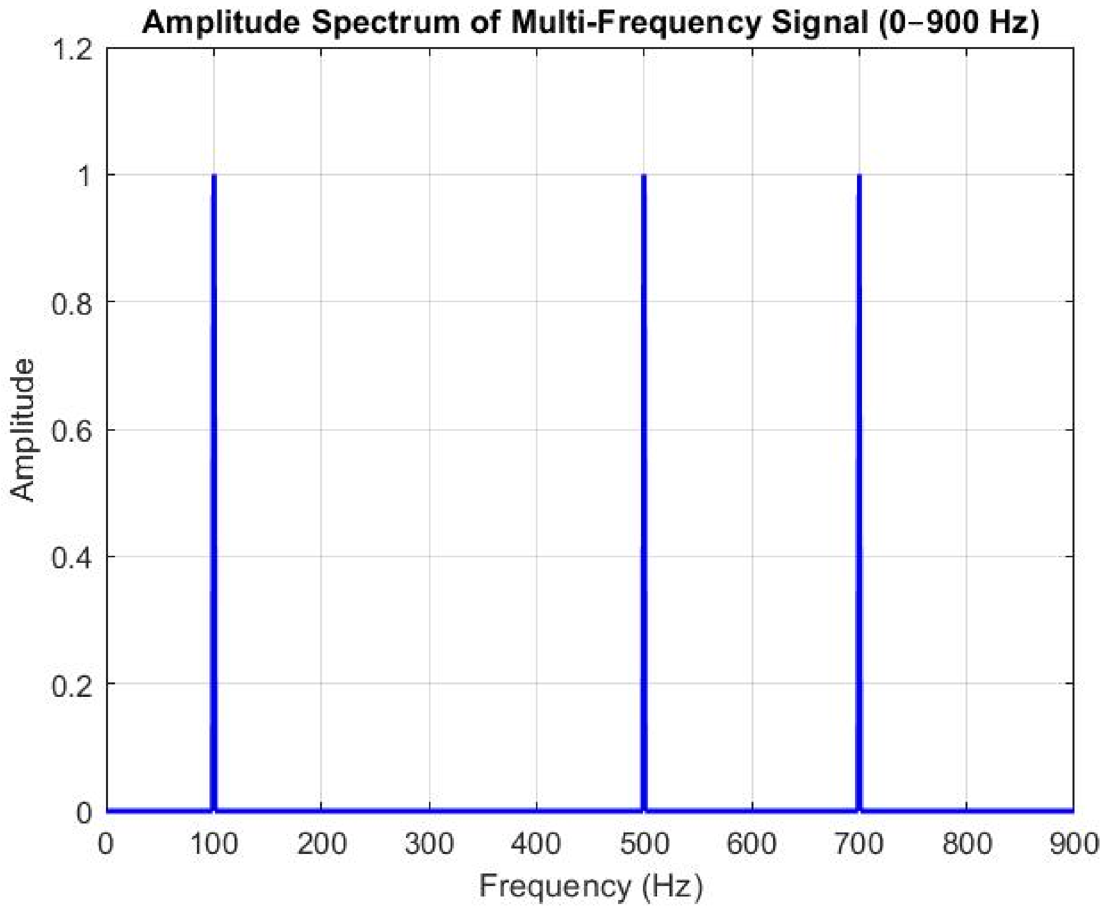
is $x(f)= \frac{A}{2}[\delta(f-f0) + \delta(f+f0)]$,

which shows that a cosine wave is represented in the frequency domain by two impulses of equal strength located at positive and negative frequencies. In practical applications, the Fourier transform integral is approximated numerically using the Fast Fourier Transform (FFT**),** which computes a discrete version of the transform over sampled data. This approximation provides the amplitude spectrum, where sharp peaks appear at the frequencies of the cosine components, indicating their presence and strength. Thus, the Fourier Transform—both in exact theory and in approximate numerical form—is a powerful tool for analyzing and interpreting multi-frequency cosine signals.

Source Code in MATLAB:

```matlab
% Clear workspace
clear; clc; close all;

%% Parameters
N = 250;              % Number of samples
ts = 0.0002;          % Sampling interval (seconds)
t = (0:N-1) * ts;     % Time vector

%% Signal: sum of three cosines
x = cos(2*pi*100*t) + cos(2*pi*500*t) + cos(2*pi*700*t);

%% Plot time-domain signal
figure;
subplot(2,1,1);
plot(t, x, 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('Amplitude');
title('Time-Domain Signal x(t)');
grid on;

%% Compute DTFT using numerical integration
f = 0:1:900;          % Frequency range from 0 to 900 Hz
X = zeros(size(f)); % Initialize amplitude spectrum

for k = 1:length(f)
    X(k) = trapz(t, x .* exp(-1j*2*pi*f(k)*t)); % Numerical
integration
end

%% Plot amplitude spectrum
subplot(2,1,2);
plot(f, abs(X), 'LineWidth', 1.5);
xlabel('Frequency (Hz)');
ylabel('|X(f)|');
title('Amplitude Spectrum of x(t)');
xlim([0 900]); % Show only 0 <= f <= 900 Hz
grid on;
```

OUTPUT:


Amplitude Spectrum of Multi-Frequency Signal (0–900 Hz)
Amplitude
Frequency (Hz)
1.2
1
0.8
0.6
0.4
0.2
0
0
100
200
300
400
500
600
700
800
900

Experiment No: 14

Experiment Name:

Discrete-Time Fourier Transform (DTFT) of a Finite-Duration Sequence.

Theory:

The Discrete-Time Fourier Transform (DTFT) of a sequence $x(n)$ is defined as:

$$X(e^{j\omega}) = \sum_{n=-\infty}^{\infty} x(n)e^{-j\omega n}, \qquad \omega \in [0, \pi]$$

For a finite-duration sequence $x(n) = \{1,2,3,4,5\}$, the DTFT becomes:

$$X(e^{j\omega}) = 1 + 2e^{-j\omega} + 3e^{-j2\omega} + 4e^{-j3\omega} + 5e^{-j4\omega}$$

The DTFT is a continuous function of ω, but we evaluate it at 501 equally spaced frequency samples between 000 and π.

From $X(e^{j\omega})$, we can compute:

- **Magnitude Spectrum:** $|X(e^{j\omega})|$
- **Phase Spectrum (Angle):** $\angle X(e^{j\omega})$
- **Real Part:** $\Re\{X(e^{j\omega})\}$
- **Imaginary Part:** $\Im\{X(e^{j\omega})\}$

Procedure

1. **Launch MATLAB and prepare workspace**

   Open MATLAB → clc; clear; close all;

2. **Enter the input sequence**

   Define the finite-duration sequence and its length:

   Matlab

```
x = [1 2 3 4 5];

N = length(x);

n = 0:N-1;
```

3. **Create frequency grid**

   Generate 501 equally spaced frequencies between 0 and π (rad/sample):

   Matlab

```
w = linspace(0, pi, 501);
```

4. **Compute the DTFT samples**

   Evaluate $X\left(e^{j\omega}\right) = \sum_{n=0}^{N-1} x(n)e^{-j\omega n}$ at every ω in w:

   Matlab

```
X = zeros(size(w));

for k = 1:length(w)

    X(k) = sum(x .* exp(-1j*w(k)*n));

end
```

5. **Derive spectral components**

   Magnitude, phase, real, and imaginary parts:

   Matlab

```
Xmag = abs(X);

Xang = angle(X);

Xre  = real(X);

Xim  = imag(X);
```

6. **Plot results**

   Open a 2×2 figure and plot each component against ω:

   Matlab

```
figure;

subplot(2,2,1); plot(w, Xmag, 'LineWidth',1.2);

xlabel('\omega (rad/sample)'); ylabel('|X(e^{j\omega})|'); title('Magnitude'); grid
on;


subplot(2,2,2); plot(w, Xang, 'LineWidth',1.2);

xlabel('\omega (rad/sample)'); ylabel('Phase (rad)'); title('Phase'); grid on;


subplot(2,2,3); plot(w, Xre,  'LineWidth',1.2);

xlabel('\omega (rad/sample)'); ylabel('Re\{X\}'); title('Real Part'); grid on;
```

```
subplot(2,2,4); plot(w, Xim,  'LineWidth',1.2);
xlabel('\omega (rad/sample)'); ylabel('Im\{X\}'); title('Imaginary Part'); grid on;
```

7. **Verify numerically (optional, recommended)**

   Cross-check with the DFT at dense frequencies using zero-padding and FFT:

   Matlab

   ```
   M = 2048;                      % large FFT size
   Xfft = fft(x, M);
   w_fft = 2*pi*(0:M-1)/M;        % [0, 2π) grid
   % Compare X on [0, π] by interpolating indices k where w_fft ≈ w
   The DTFT plots (continuous in ω) should align with the dense FFT samples on [0,π].
   ```

8. **Record observations**

   Note peak locations, symmetry properties (real/imag parts), and unwrap phase if needed:

   Matlab

   ```
   Xang_unwrapped = unwrap(Xang);
   ```

9. **Save figures (if required)**

   Matlab

   ```
   saveas(gcf, 'DTFT_x_1to5.png');
   ```

MATLAB Input (Code):

clc; clear; close all;

% Input sequence

x = [1 2 3 4 5];

N = length(x);

% Frequency samples (501 points between 0 and pi)

w = linspace(0, pi, 501);

% Compute DTFT manually

X = zeros(size(w));

```
for k = 1:length(w)
    X(k) = sum(x .* exp(-1j*w(k)*(0:N-1)));
end

% Plot results
figure;
subplot(2,2,1);
plot(w, abs(X), 'b','LineWidth',1.2);
xlabel('\omega (rad/sample)'); ylabel('|X(e^{j\omega})|');
title('Magnitude Spectrum'); grid on;

subplot(2,2,2);
plot(w, angle(X), 'r','LineWidth',1.2);
xlabel('\omega (rad/sample)'); ylabel('Phase (radians)');
title('Phase Spectrum'); grid on;

subplot(2,2,3);
plot(w, real(X), 'g','LineWidth',1.2);
xlabel('\omega (rad/sample)'); ylabel('Re\{X\}');
title('Real Part'); grid on;
subplot(2,2,4);
plot(w, imag(X), 'm','LineWidth',1.2);
xlabel('\omega (rad/sample)'); ylabel('Im\{X\}');
title('Imaginary Part'); grid on;
```

Output (Expected Plots):

1. **Magnitude Spectrum** → Shows how the energy of the sequence is distributed over frequency.
2. **Phase Spectrum** → Displays the phase shifts across frequency.

3. **Real Part** → Oscillatory function corresponding to cosine terms.

4. **Imaginary Part** → Oscillatory function corresponding to sine terms.

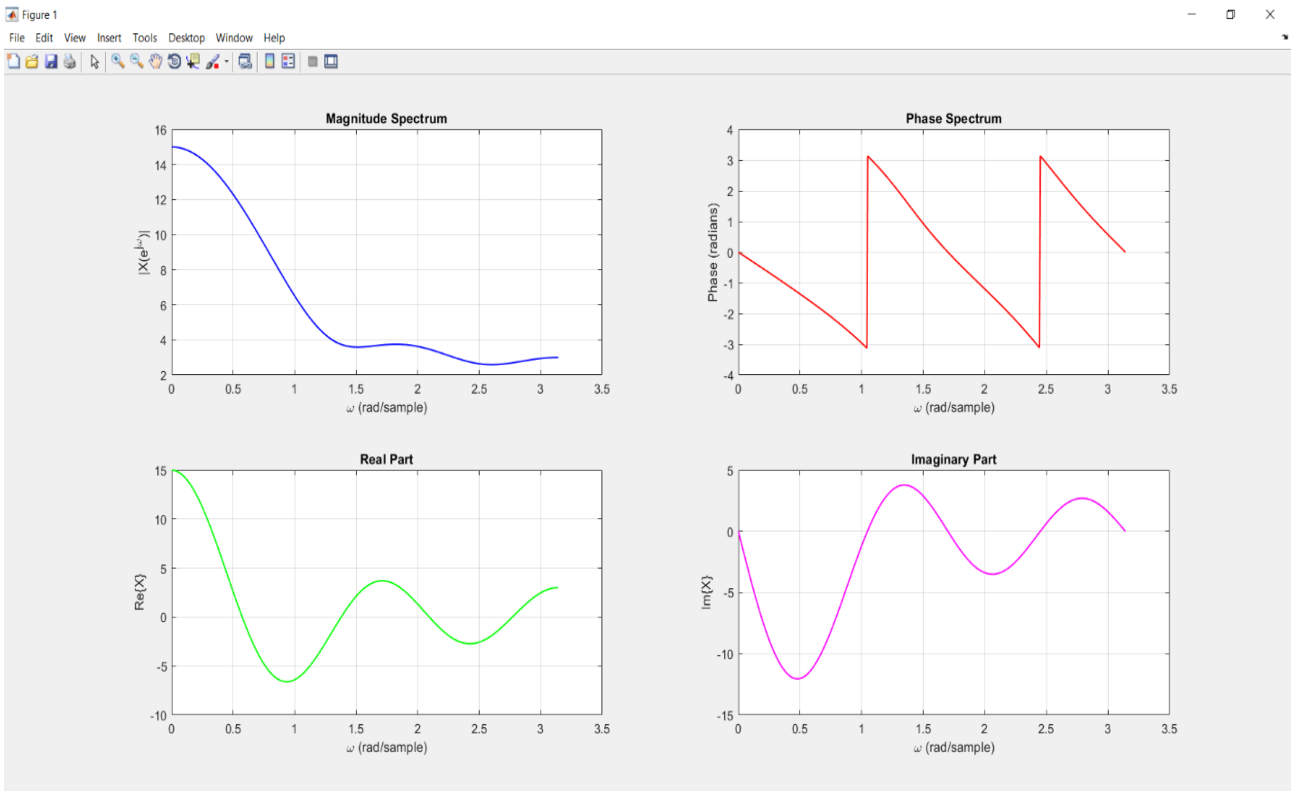


Figure 1
File Edit View Insert Tools Desktop Window Help
Magnitude Spectrum
$|X(e^{j\omega})|$
ω (rad/sample)
Phase Spectrum
Phase (radians)
ω (rad/sample)
Real Part
Re{X}
ω (rad/sample)
Imaginary Part
Im{X}
ω (rad/sample)