Experiment No: 1 Date: 01-08-2024

Generation of Basic Test Signals

Aim: To simulate Basic Test signals in matlab.

Theory: Fundamental signals in Digital Signal Processing (DSP) are crucial for analyzing systems and representing complex signals.

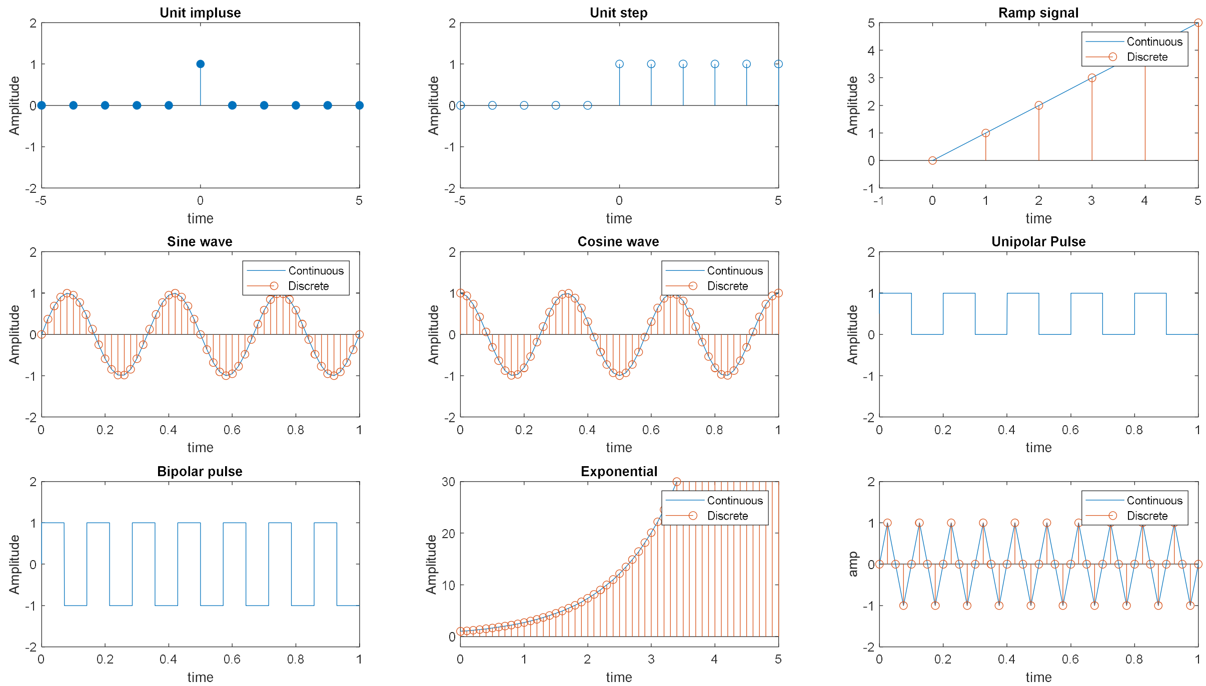
1. Unit Impulse Signal: Zero everywhere except at n=0, where it is 1. Used to test system responses.
2. Unit Step Signal: 0 for n<0 and 1 for n≥0. Analyzes step responses and stability in systems.
3. Ramp Signal: Increases linearly for n≥0. Represents constant growth or acceleration.
4. Sine Wave: A periodic signal oscillating between positive and negative values, fundamental in signal decomposition.
5. Cosine Wave: Similar to a sine wave but starts at its peak; phase-shifted by 90 degrees.
6. Exponential Signal: Grows or decays exponentially, useful for modeling processes like population growth.
7. Unipolar Pulse: A rectangular signal that is positive for a specific period and zero elsewhere.
8. Bipolar Pulse: Rectangular signal that alternates between positive and negative values.
9. Triangular Wave: A periodic signal that rises and falls linearly, forming a triangle shape.

Program:

```
%Simulation of basic test signals
clc;
clear all;
close all;

%Unit impulse
t1=-5:1:5;
y1=[zeros(1,5),ones(1,1),zeros(1,5)];
subplot(3,3,1);
stem(t1,y1,"filled");
xlabel("time");
ylabel("Amplitude");
```

```
title("Unit impluse");
axis([-5 5 -2 2]);

%unit step
y2=[zeros(1,5),ones(1,6)];
subplot(3,3,2);
stem(t1,y2);
xlabel("time");
ylabel("Amplitude");
title("Unit step");
axis([-5 5 -2 2]);

%Unit ramp signal
t3=0:1:5;
y3=t3;
subplot(3,3,3);
plot(t3,y3);
hold on;
stem(t3,y3);
xlabel("time");
ylabel("Amplitude");
title("Ramp signal");
legend("Continuous","Discrete");
axis([-1 5 -1 5]);

%Sine signal
f4=3;
t4=0:0.02:1;
y4=sin(2*pi*f4*t4);
subplot(3,3,4);
plot(t4,y4);
hold on;
stem(t4,y4);
xlabel("time");
ylabel("Amplitude");
title("Sine wave");
legend("Continuous","Discrete");
axis([0 1 -2 2]);

%Cosine signal
t5=0:0.02:1;
y5=cos(2*pi*f4*t5);
subplot(3,3,5);
plot(t5,y5);
hold on;
stem(t5,y5);
xlabel("time");
ylabel("Amplitude");
title("Cosine wave");
legend("Continuous","Discrete");
axis([0 1 -2 2]);

%Unipolar pulse
f6=5;
t6=0:0.0001:1;
y6=0.5* (sign(sin(2*pi*f6*t6))+1);
subplot(3,3,6);
```

Observation:



```
plot(t6,y6);
xlabel("time");
ylabel("Amplitude");
title("Unipolar Pulse");
axis([0 1 -2 2]);

%Bipolar pulse
f7=7;
y7=sign(sin(2*pi*f7*t6));
subplot(3,3,7);
plot(t6,y7);
xlabel("time");
ylabel("Amplitude");
title("Bipolar pulse");
axis([0 1 -2 2]);

%exponential signal
t8=0:0.1:5;
y8=exp(1*t8);
subplot(3,3,8);
plot(t8,y8);
hold on;
stem(t8,y8);
xlabel("time");
ylabel("Amplitude");
title("Exponential");
legend("Continuous","Discrete");
axis([0 5 -2 30]);

%Triangular wave
f9=10;
t9 = 0:0.025:1;
y9 = sin(2 *pi * f9 * t9);
subplot(3,3,9);
plot(t9, y9);
hold on;
stem(t9, y9);
xlabel("time");
ylabel("amp");
legend("Continuous","Discrete");
axis([0 1 -2 2]);
```

Result : Simulated and plotted the Basic Test signals in matlab.

Experiment No: 2 Date: 08-08-2024

Verification of Sampling theorem

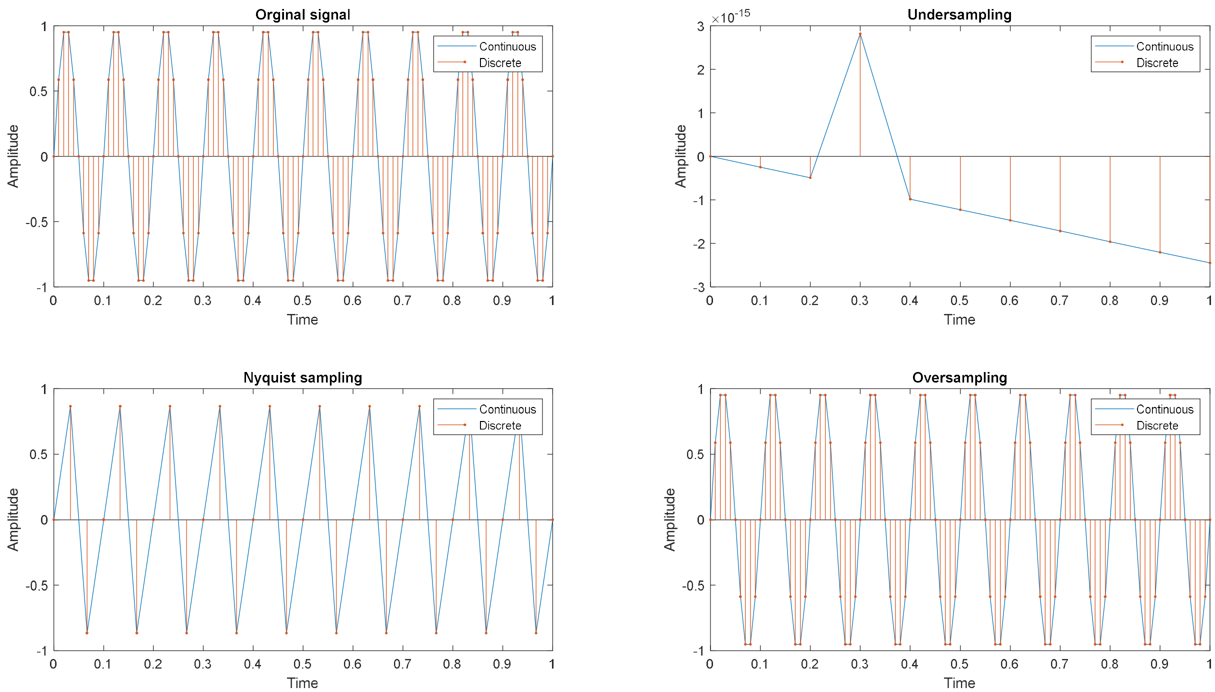
Aim: To verify Sampling theorem.

Theory: The Sampling Theorem states that a continuous signal can be reconstructed from samples if sampled at a rate greater than twice its highest frequency (Nyquist rate). Verification involves sampling a signal at different rates: below (causing aliasing) and at or above (allowing accurate reconstruction), highlighting the importance of proper sampling rates.

Program:

```
%verification of sampling theorem
clc;
clear all;
close all;
%original signal
t=0:0.01:1;
fm=10;
y=sin(2*pi*fm*t);
figure;
subplot(2,2,1);
plot(t,y);
hold on;
stem(t,y,".");
xlabel("Time");
ylabel("Amplitude");
title("Orginal signal");
legend("Continuous","Discrete");
%less than nyquist rate
fs1=fm;
t1=0:1/fs1:1;
y1=sin(2*pi*fm*t1);
subplot(2,2,2);
plot(t1,y1);
hold on;
stem(t1,y1,'.');
xlabel("Time");
ylabel("Amplitude");
title("Undersampling");
legend("Continuous","Discrete");
%equal to nyquist rate
fs2=3*fm;
t2=0:1/fs2:1;
y2=sin(2*pi*fm*t2);
subplot(2,2,3);
plot(t2,y2);
hold on;
stem(t2,y2,'.');
```

Observation:



```
xlabel("Time");
ylabel("Amplitude");
title("Nyquist sampling");
legend("Continuous","Discrete");
%greater than nyquist rate
fs3=10*fm;
t3=0:1/fs3:1;
y3=sin(2*pi*fm*t3);
subplot(2,2,4);
plot(t3,y3);
hold on;
stem(t3,y3,'.');
xlabel("Time");
ylabel("Amplitude");
title("Oversampling");
legend("Continuous","Discrete");
```

Result : Clear distinction between under-sampled, nyquist sampled, and over-sampled signals demonstrating the effects of sampling rate on signal reconstruction.

Experiment No: 3 Date: 08-08-2024

Linear Convolution

Aim: To perform linear convolution of two signals both using built-in MATLAB functions and manual methods.

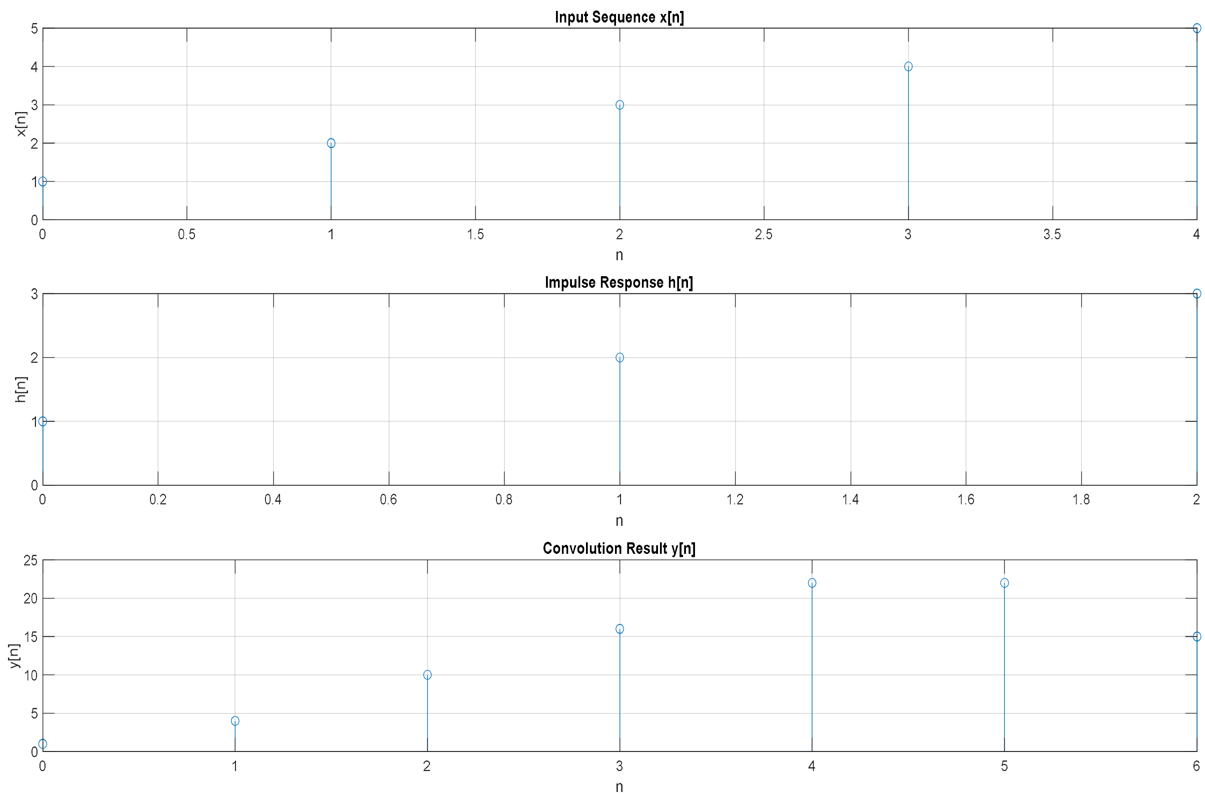
Theory: Linear convolution combines two signals to produce a third signal, representing the output of a linear time-invariant (LTI) system. It involves sliding one signal over another, multiplying overlapping values, and summing them to form the output. This process is crucial in signal processing for analyzing system responses and implementing filtering techniques.

Program:

```
% Linear Convolution using inbuilt function
clc;
clear;
close all;

% Input sequences and their indices
x = input('Enter input sequence x: ');
x_ind = input('Enter index of x: ');
h = input('Enter impulse response h: ');
h_ind = input('Enter index of h: ');

% Linear convolution
y = conv(x, h);

% Determine the time indices for the convolution result
y_ind = min(x_ind) + min(h_ind) : max(x_ind) + max(h_ind);

% Display the convolution result
disp('Convolution result y:');
disp(y);

% Plotting the input sequences and the convolution result

% Create a figure window
figure;

% Plot the first sequence x
subplot(3, 1, 1);
stem(x_ind, x);
title('Input Sequence x[n]');
xlabel('n');
ylabel('x[n]');
grid on;

% Plot the second sequence h
subplot(3, 1, 2);
```

Observation:

Enter input sequence x: [1 2 3 4 5]
Enter index of x: [0:4]
Enter impulse response h: [1 2 3]
Enter index of h: [0:2]
Convolution result y:
    1   4   10   16   22   22   15



```
stem(h_ind, h);
title('Impulse Response h[n]');
xlabel('n');
ylabel('h[n]');
grid on;

% Plot the convolution result y
subplot(3, 1, 3);
stem(y_ind, y);
title('Convolution Result y[n]');
xlabel('n');
ylabel('y[n]');
grid on;
```

Program:

```
%Linear convolution without using inbuilt functions

% Input sequences and their indices
x = input('Enter input sequence x: ');
x_ind = input('Enter index of x: ');
h = input('Enter impulse response h: ');
h_ind = input('Enter index of h: ');

% Get the length of the sequences
len_x = length(x);
len_h = length(h);

% Calculate the length of the convolution result
len_y = len_x + len_h - 1;

% Initialize the result sequence with zeros
y = zeros(1, len_y);

% Perform the convolution
for i = 1:len_x
    for j = 1:len_h
        y(i + j - 1) = y(i + j - 1) + x(i) * h(j);
    end
end

% Determine the time indices for the convolution result
y_ind = min(x_ind) + min(h_ind) : max(x_ind) + max(h_ind);

% Display the result
disp('Linear Convolution Result:');
disp(y);

% Plotting the input sequences and the convolution result

% Create a figure window
figure;

% Plot the first sequence x
subplot(3, 1, 1);
stem(x_ind, x);
title('Input Sequence x[n]');
xlabel('n');
ylabel('x[n]');
grid on;

% Plot the second sequence h
subplot(3, 1, 2);
stem(h_ind, h);
title('Impulse Response h[n]');
xlabel('n');
ylabel('h[n]');
grid on;

% Plot the convolution result y
subplot(3, 1, 3);
stem(y_ind, y);
```

Observation:

Enter input sequence x: [1 2 3 4]

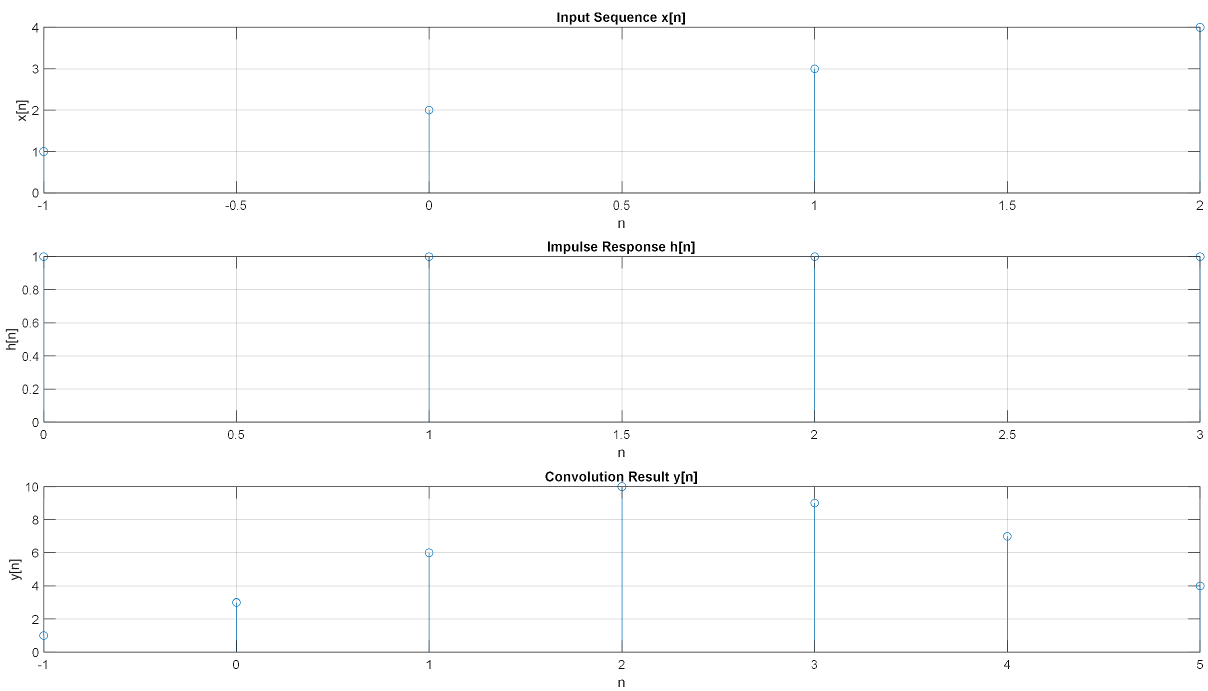
Enter index of x: [-1:2]

Enter impulse response h: [1 1 1 1]

Enter index of h: [0:3]

Linear Convolution Result:

1 3 6 10 9 7 4



```
title('Convolution Result y[n]');
xlabel('n');
ylabel('y[n]');
grid on;
```

<u>Result</u> : Performed linear convolution of two signals both using built-in MATLAB functions and manual methods and verified the outputs.

Observation:

Enter sequence 1:[1 2 3 4]

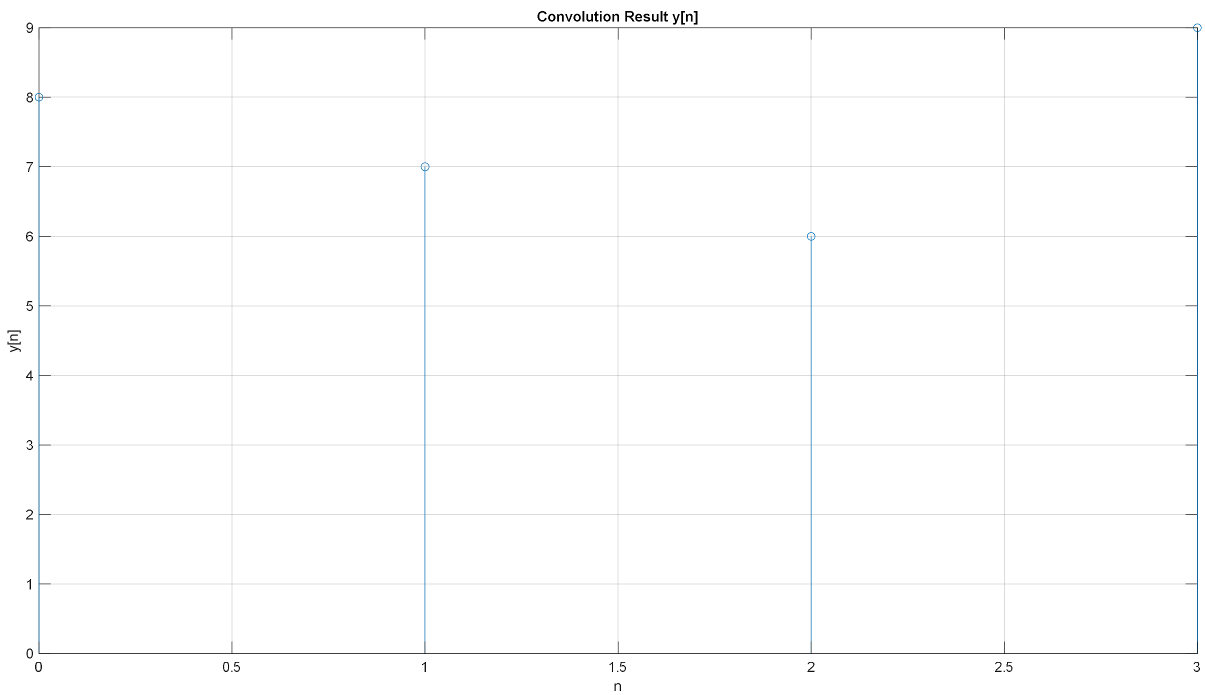
Enter sequence 2:[1 1 1]

Reversed x

    4   3   2   1

Convolution product y:

    8   7   6   9



Experiment No: 4 Date: 22-08-2024

Circular Convolution

Aim: To perform circular convolution of two signals using various methods.

Theory: Circular convolution combines two periodic signals to produce a third periodic signal, wrapping around at the boundaries. Unlike linear convolution, it treats signals as periodic, making it especially useful in the frequency domain with the Discrete Fourier Transform (DFT). This operation is essential in filtering and signal analysis, preserving periodic characteristics in the output.

Program:

```
%Circular convolution using concentric circle method
clc;
close all;
clear all;
x1 = input("Enter sequence 1:");
x2 = input("Enter sequence 2:");
N=max(length(x1),length(x2));
x1new=[x1 zeros(1,N-length(x1))];
x2new=[x2 zeros(1,N-length(x2))];
x1new=x1new(:,end:-1:1);
disp("Reversed x");
disp(x1new);
for i=1:length(x1new)
    x1new=[x1new(end) x1new(1:end-1)];
    y(i)=sum(x1new.*x2new);
end
disp("Convolution product y:");
disp(y);

% Plot the convolution result y
stem(0:length(y)-1, y);
title('Convolution Result y[n]');
xlabel('n');
ylabel('y[n]');
grid on;
```

Observation :

Enter sequence 1:[1 2 3 4]

Enter sequence 2:[1 1 1]

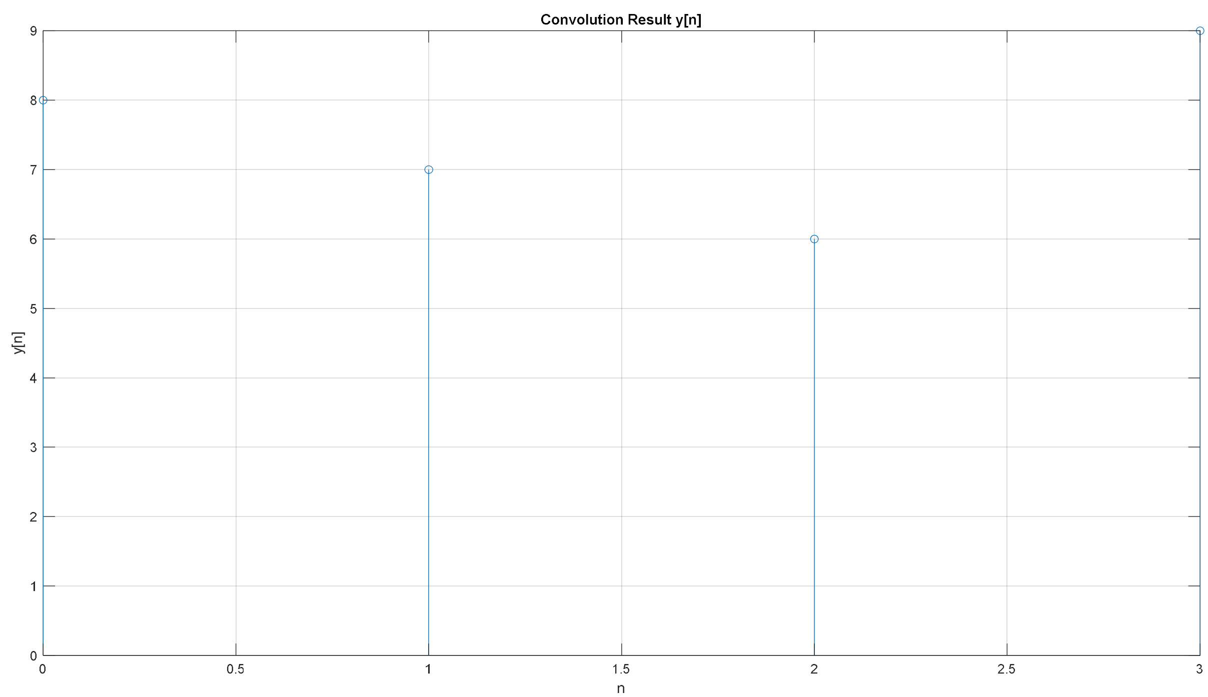
Convolution product y:

8

7

6

9

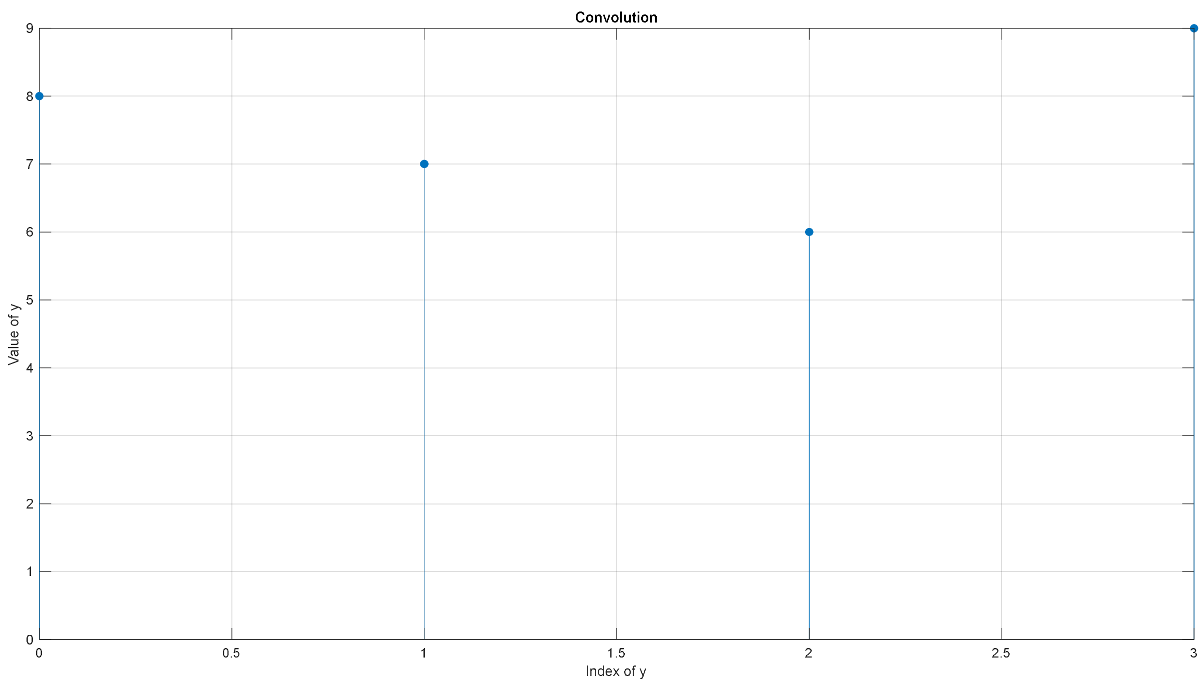


Program:

```
%Circular convolution using matrix multiplication
clc;
close all;
clear all;
x1 = input("Enter sequence 1:");
x2 = input("Enter sequence 2:");
N=max(length(x1),length(x2));
x1new=[x1 zeros(1,N-length(x1))];
x2new=[x2 zeros(1,N-length(x2))];

m=[];
x2new=x2new(:,end:-1:1);
for i=1:length(x2new)
    x2new=[x2new(end) x2new(1:end-1)];
    m=[m;x2new];
end
y=m*x1new';%matrix multiplication
disp("Convolution product y:")
disp(y);
% Plot the convolution result y
stem(0:length(y)-1, y);
title('Convolution Result y[n]');
xlabel('n');
ylabel('y[n]');
grid on;
```

Observation :

Enter Sequence 1:[1 2 3 4]
Enter Sequence 2:[1 1 1]
Convolution product y:
    8   7   6   9



Program:

```
%Circular Convolution using DFT
clc;
close all
clear all;
x=input("Enter Sequence 1:");
h=input("Enter Sequence 2:");
x_len=length(x);
h_len=length(h);
n=max(h_len,x_len);
xnew=[x zeros(1,n-x_len)];
hnew=[h zeros(1,n-h_len)];
x1=fft(xnew);
h1=fft(hnew);
y1=x1.*h1;
y=ifft(y1);
y_ind=0:n-1;
disp("Convolution product y:")
disp(y);
% Plot the convolution result y
stem(y_ind,y,"filled");
title("Convolution");
xlabel("Index of y");
ylabel("Value of y")
grid on;
```

Result : Performed circular convolution of two signals using Concentric circle method,Matrix method and DFT method and verified the outputs.

Experiment No: 5 Date:29-09-2024

Linear Convolution using Circular Convolution and vice-versa.

Aim: To perform Linear convolution of two signals using Circular convolution and vice-versa.

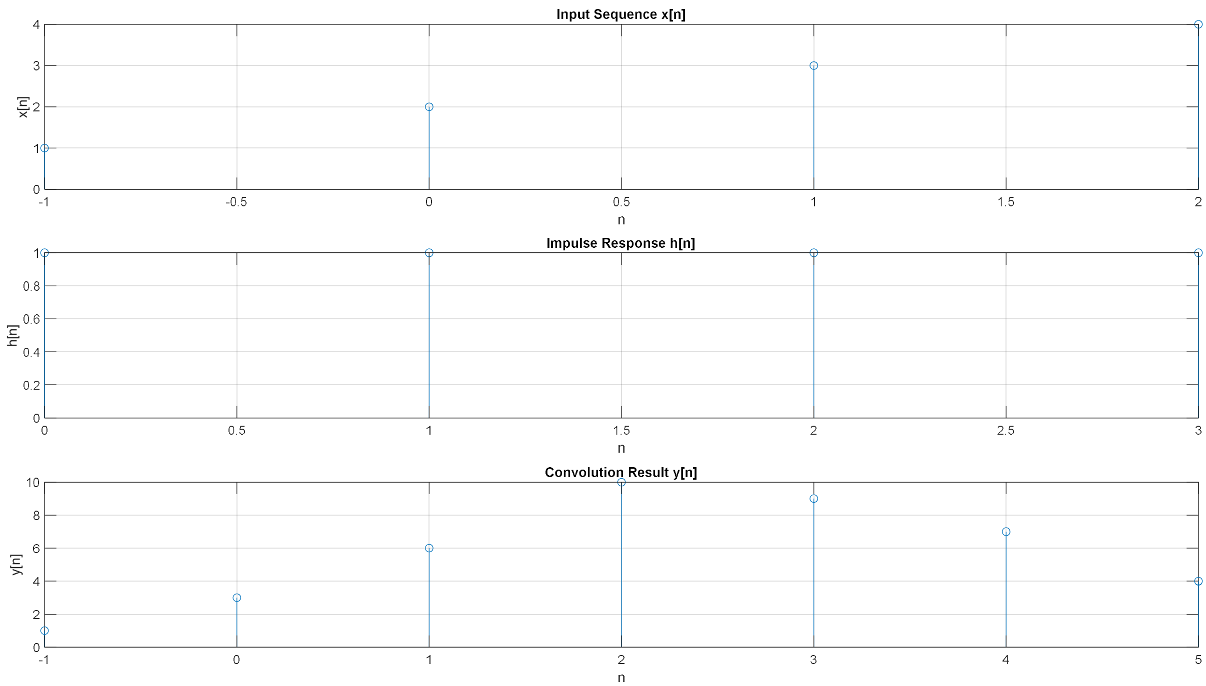
Theory: Linear convolution can be expressed in terms of circular convolution by zero-padding the signals to the same length, allowing for periodic extension. This technique enables the use of efficient algorithms like the Discrete Fourier Transform (DFT) to compute linear convolution in the frequency domain. Conversely, circular convolution can be interpreted as linear convolution when the signals are treated as periodic. This relationship is crucial for efficient processing in digital signal applications, enabling the manipulation of signals without losing important characteristics.

Program:

```
%linear convolution using Circular convolution
clc;
close all;
clear all;
x=input("Enter Sequence 1:");
x_ind=input("Index of sequence 1:");
h=input("Enter Sequence 2:");
h_ind=input("Index of sequence 2:");
x_len=length(x);
h_len=length(h);
y_ind = min(x_ind) + min(h_ind) : max(x_ind) + max(h_ind);
n=x_len+h_len-1;
xnew=[x zeros(1,n-x_len)];
hnew=[h zeros(1,n-h_len)];
x1=fft(xnew);
h1=fft(hnew);
y1=x1.*h1;
y=ifft(y1);
disp("Linear convolution product y:")
disp(y);
% Plotting the input sequences and the convolution result

% Create a figure window
figure;

% Plot the first sequence x
subplot(3, 1, 1);
stem(x_ind, x);
title('Input Sequence x[n]');
xlabel('n');
ylabel('x[n]');
grid on;
```

Observation:

Enter Sequence 1:[1 2 3 4]
Index of sequence 1:[-1:2]
Enter Sequence 2:[1 1 1 1]
Index of sequence 2:[0:3]
Linear convolution product y:
1.0000 3.0000 6.0000 10.0000 9.0000 7.0000 4.0000



```
% Plot the second sequence h
subplot(3, 1, 2);
stem(h_ind, h);
title('Impulse Response h[n]');
xlabel('n');
ylabel('h[n]');
grid on;

% Plot the convolution result y
subplot(3, 1, 3);
stem(y_ind, y);
title('Convolution Result y[n]');
xlabel('n');
ylabel('y[n]');
grid on;
```

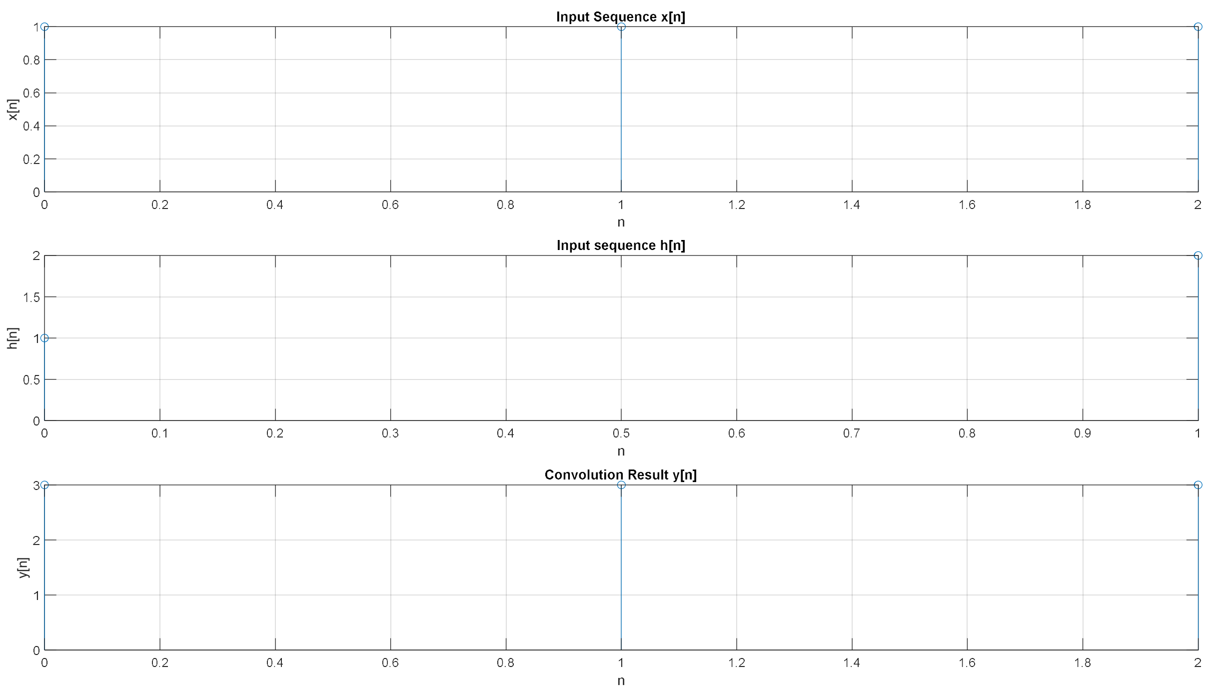
Observation:

Enter Sequence 1:[1 1 1]
Enter Sequence 2:[1 2]
    1   3   3

    2

    2   0   0

Circular convolution product y:
    3   3   3



Program:

```
%Circular convolution using linear convolution
clc;
close all;
clear;
x=input("Enter Sequence 1:");
h=input("Enter Sequence 2:");
y=conv(x,h);
n=max(length(x),length(h));
z=y(1:n);
a=y(n+1:length(y));
disp(z);
disp(a);
a_new=[a zeros(1,n-length(a))];
disp(a_new);
y=z+a_new;
disp("Circular convolution product y:")
disp(y);
% Plotting the input sequences and the convolution result

% Create a figure window
figure;

% Plot the first sequence x
subplot(3, 1, 1);
stem(0:length(x)-1, x);
title('Input Sequence x[n]');
xlabel('n');
ylabel('x[n]');
grid on;

% Plot the second sequence h
subplot(3, 1, 2);
stem(0:length(h)-1, h);
title('Input sequence h[n]');
xlabel('n');
ylabel('h[n]');
grid on;

% Plot the convolution result y
subplot(3, 1, 3);
stem(0:n-1, y);
title('Convolution Result y[n]');
xlabel('n');
ylabel('y[n]');
grid on;
```

Result : Performed Linear convolution using Circular convolution,Circular convolution using Linear convolution and verified the outputs.

Observation:

Enter the sequence: [1 0 1 0]
Enter value of N for N-point DFT :16
DFT without inbuilt function:

```
  Columns 1 through 5

   2.0000 + 0.0000i   1.7071 - 0.7071i   1.0000 - 1.0000i   0.2929 - 0.7071i   0.0000 + 0.0000i

  Columns 6 through 10

   0.2929 + 0.7071i   1.0000 + 1.0000i   1.7071 + 0.7071i   2.0000 + 0.0000i   1.7071 - 0.7071i

  Columns 11 through 15

   1.0000 - 1.0000i   0.2929 - 0.7071i   0.0000 + 0.0000i   0.2929 + 0.7071i   1.0000 + 1.0000i

  Column 16

   1.7071 + 0.7071i
```

DFT using FFT:

```
  Columns 1 through 5

   2.0000 + 0.0000i   1.7071 - 0.7071i   1.0000 - 1.0000i   0.2929 - 0.7071i   0.0000 + 0.0000i

  Columns 6 through 10

   0.2929 + 0.7071i   1.0000 + 1.0000i   1.7071 + 0.7071i   2.0000 + 0.0000i   1.7071 - 0.7071i

  Columns 11 through 15

   1.0000 - 1.0000i   0.2929 - 0.7071i   0.0000 + 0.0000i   0.2929 + 0.7071i   1.0000 + 1.0000i

  Column 16

   1.7071 + 0.7071i
```

Experiment No: 6 Date: 29-08-2024

Discrete Fourier Transform and Inverse Discrete Fourier Transform

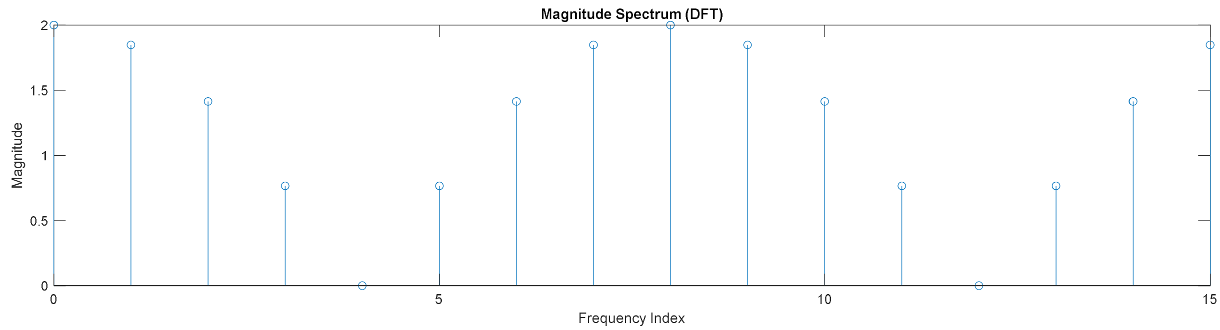
Aim: To compute the DFT and IDFT of a signal using inbuilt functions and manual methods.

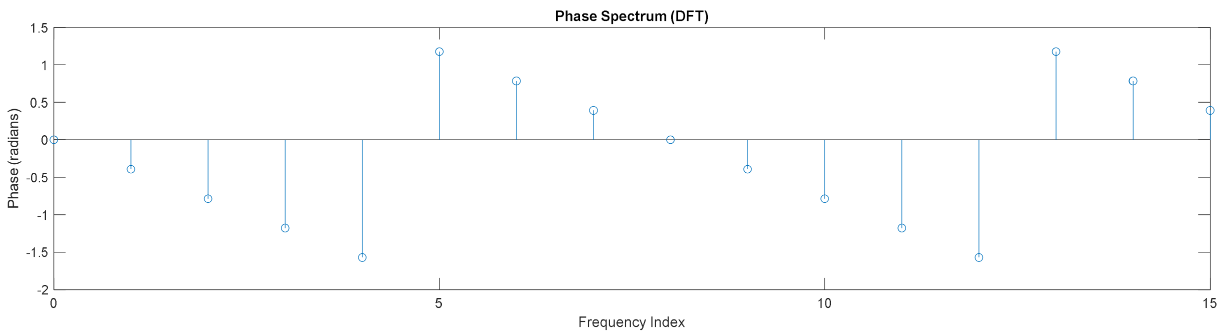
Theory: The Discrete Fourier Transform (DFT) converts a finite sequence of discrete signals from the time domain to the frequency domain, allowing analysis of frequency components. It represents the signal as a sum of complex exponentials, providing insights into its frequency content. The Inverse Discrete Fourier Transform (IDFT) reverses this process, reconstructing the original time-domain signal from its frequency-domain representation. Both DFT and IDFT are essential tools in digital signal processing, enabling efficient signal analysis and manipulation using algorithms like the Fast Fourier Transform (FFT).

Program:

```
%DFT with and without using inbuilt function
clc;
clear all;
close all;

% Input sequence
x = input("Enter the sequence: ");
N=input("Enter value of N for N-point DFT :");
L = length(x);
if N>=L %Checking if N>= length of input sequence
    xn = [x,zeros(1, N-L)];
X=zeros(1,N);
% DFT computation without inbuilt function
for k = 0:N-1
    for n = 0:N-1
        X(k+1) = X(k+1) + xn(n+1) .* exp(-1i * 2 * pi * n * k / N);
    end
end

% Displaying results
disp("DFT without inbuilt function:");
disp(round(X, 5));

disp("DFT using FFT:");
y = fft(xn, N);
disp(round(y,5));

% Magnitude spectrum
mag = abs(X);
subplot(2, 1, 1);
stem(0:N-1, mag);
```

Observation:





```
title('Magnitude Spectrum (DFT)');
xlabel('Frequency Index');
ylabel('Magnitude');

% Phase spectrum
ph = atan2(imag(X),real(X)); % Or use angle(X)
subplot(2, 1, 2);
stem(0:N-1, ph);
title('Phase Spectrum (DFT)');
xlabel('Frequency Index');
ylabel('Phase (radians)');

else  %if N< length of input sequence
    disp("DFT cannot be calculated !")
end
```

Observation:

Enter DFT sequence: [1 2 3 4]
Enter the value of N for N-point IDFT:4
IDFT without using inbuilt function:
  2.5000 + 0.0000i -0.5000 - 0.5000i -0.5000 + 0.0000i -0.5000 + 0.5000i

IDFT using ifft:
  2.5000 + 0.0000i -0.5000 - 0.5000i -0.5000 + 0.0000i -0.5000 + 0.5000i

Program:

```
%IDFT with and without using inbuilt function
clc;
clear all;
close all;
X=input("Enter DFT sequence: ");
L=length(X);
N=input("Enter the value of N for N-point IDFT:");
if N>=L
    Xn=[X zeros(1,N-L)];
x=zeros(1,N);
for n=0:N-1
    for k=0:N-1
        x(n+1)=x(n+1)+((Xn(k+1).*exp(1i*2*pi*n*k/N))/N);
    end
end

disp("IDFT without using inbuilt function:");
disp(round(x,5));
y=round(ifft(Xn,N),5);
disp("IDFT using ifft:");
disp(y);

else
    disp("N-point IDFT cannot be found!")
end
```

Observation:

Enter the sequence: [1 2 3 4]
Enter value of N for N-point DFT: 4
Twiddle Factor Matrix:

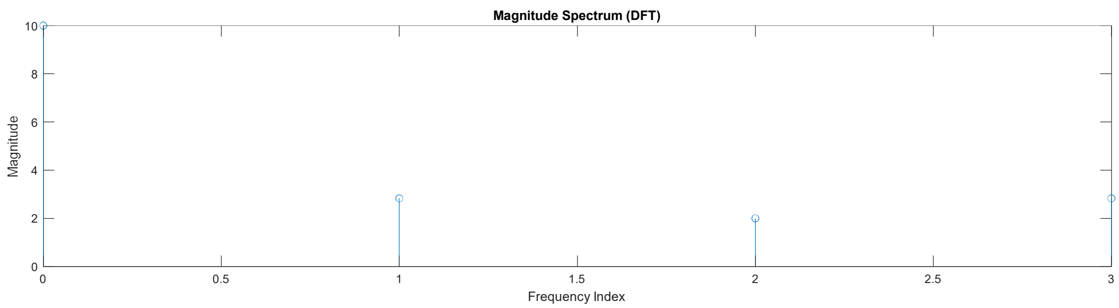
```
   1.0000 + 0.0000i   1.0000 + 0.0000i   1.0000 + 0.0000i   1.0000 + 0.0000i
   1.0000 + 0.0000i   0.0000 - 1.0000i  -1.0000 + 0.0000i   0.0000 + 1.0000i
   1.0000 + 0.0000i  -1.0000 + 0.0000i   1.0000 + 0.0000i  -1.0000 + 0.0000i
   1.0000 + 0.0000i   0.0000 + 1.0000i  -1.0000 + 0.0000i   0.0000 - 1.0000i
```

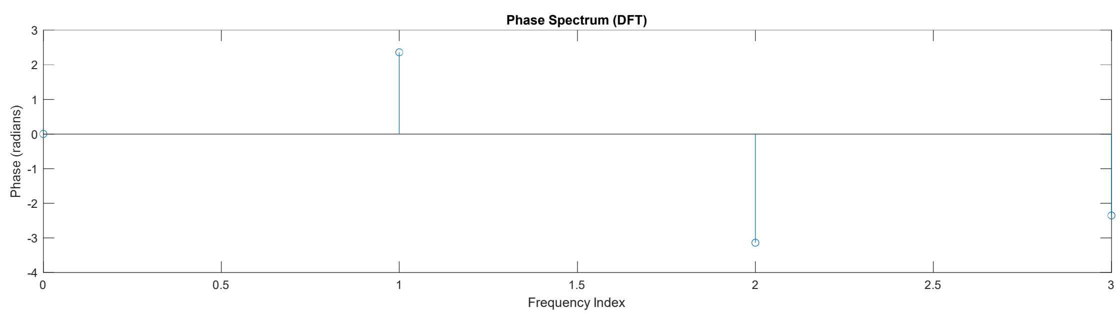
DFT using Twiddle factor matrix multiplication:

```
  10.0000 + 0.0000i  -2.0000 - 2.0000i  -2.0000 + 0.0000i  -2.0000 + 2.0000i
```

DFT using FFT:

```
  10.0000 + 0.0000i  -2.0000 + 2.0000i  -2.0000 + 0.0000i  -2.0000 - 2.0000i
```





Program:

```
%DFT with twiddle factor matrix
clc;
clear all;
close all;

% Input sequence
x = input("Enter the sequence: ");
N = input("Enter value of N for N-point DFT: ");
L = length(x);

if N >= L % Checking if N >= length of input sequence
    xn = [x, zeros(1, N-L)];

    % Create twiddle factor matrix
    k = 0:N-1;
    n = 0:N-1;
    W = exp(-1i * 2 * pi * n' * k / N );

    % Display twiddle factor matrix
    disp("Twiddle Factor Matrix:");
    disp(round(W, 5));

    % DFT computation using matrix multiplication
    X = W * xn';

    % Displaying results
    disp("DFT using Twiddle factor matrix multiplication:");
    disp(round(X', 5));

    disp("DFT using FFT:");
    y = fft(xn, N);
    disp(round(y, 5));

    % Magnitude spectrum
    mag = abs(X);
    subplot(2, 1, 1);
    stem(0:N-1, mag);
    title('Magnitude Spectrum (DFT)');
    xlabel('Frequency Index');
    ylabel('Magnitude');

    % Phase spectrum
    ph = angle(X);
    subplot(2, 1, 2);
    stem(0:N-1, ph);
    title('Phase Spectrum (DFT)');
    xlabel('Frequency Index');
    ylabel('Phase (radians)');
else % if N < length of input sequence
    disp("DFT cannot be calculated!")
end
```

Observation:

Enter DFT sequence: [1 2 3 4]

Enter the value of N for N-point IDFT: 4

Displaying Twiddle Factor Matrix

```
   1.0000 + 0.0000i   1.0000 + 0.0000i   1.0000 + 0.0000i   1.0000 + 0.0000i
   1.0000 + 0.0000i   0.0000 + 1.0000i  -1.0000 + 0.0000i  -0.0000 - 1.0000i
   1.0000 + 0.0000i  -1.0000 + 0.0000i   1.0000 - 0.0000i  -1.0000 + 0.0000i
   1.0000 + 0.0000i  -0.0000 - 1.0000i  -1.0000 + 0.0000i   0.0000 + 1.0000i
```

IDFT without using Twiddle factor matrix multiplication:

```
   2.5000 + 0.0000i  -0.5000 + 0.5000i  -0.5000 + 0.0000i  -0.5000 - 0.5000i
```

IDFT using ifft:

```
   2.5000 + 0.0000i  -0.5000 - 0.5000i  -0.5000 + 0.0000i  -0.5000 + 0.5000i
```

Program:

```
%IDFT using twiddle factor matrix
clc;
clear all;
close all;

X = input("Enter DFT sequence: ");
L = length(X);
N = input("Enter the value of N for N-point IDFT: ");

if N >= L
    Xn = [X zeros(1, N-L)];

    % Create twiddle factor matrix
    n = 0:N-1;
    k = 0:N-1;
    W = exp(1i * 2 * pi * (n' * k) / N);

    disp("Displaying Twiddle Factor Matrix");
    disp(W);

    % Compute IDFT
    x = (W * Xn') / N;

    disp("IDFT without using Twiddle factor matrix multiplication:");
    disp(round(x', 5));

    y = round(ifft(Xn, N), 5);
    disp("IDFT using ifft:");
    disp(y);
else
    disp("N-point IDFT cannot be found!")
end
```

Result :Computed DFT and IDFT using both inbuilt functions and manual methods and verified the outputs.

Experiment No: 7 Date: 29-08-2024

Properties of DFT

Aim: To prove the properties of DFT.

Theory:

1. LINEARITY:

The linear property of DFT states that the DFT of a linear weighted combination of two or more signals is equal to similar linear weighted combinations of the DET of individual signals.

DFT{x1(n)} =X1(k) and DFT{x2(n)} =X2(k)

DFT {a 1 x1(n) + a 2 x2(n)} = a 1 X1(k) + a 2X2 (k)

Where a 1 and a 2 are constants

2. MULTIPLICATION:

The Multiplication property of DFT says that DFT of product of two discrete time sequences is equivalent to the circular convolution of the DFTs of the individual sequences scaled by a factor 1/N.

If DFT{x (n)} =X (k), then

DFT{x1(n) x2(n)} = 1N[X1(k)* X2(k)]

3. CIRCULAR CONVOLUTION:

The Circular Convolution of two N point sequences x1(n) and x2(n) is defined as

$$x_1(n) * x_2(n) = \sum_{m=0}^{N-1} x_1(m) x_2(n-m)_N$$

4. PARSEVALS RELATION:

Let DFT{x1(n)} = X1(k) and DFT{x2(n)} = X2(k) the by Parsevals relation

$$\sum_{n=0}^{N-1} x_1(n)\, x_2^*(n) = \frac{1}{N} \sum_{k=0}^{N-1} X_1(k) X_2^*(k)$$

Observation

Enter sequence 1:[1 2 3 4]
Enter sequence 2:[1 1 1 1]
LHS:
  32.0000 + 0.0000i -4.0000 + 4.0000i -4.0000 + 0.0000i -4.0000 - 4.0000i

RHS:
  32.0000 + 0.0000i -4.0000 + 4.0000i -4.0000 + 0.0000i -4.0000 - 4.0000i

Linearity property verified!

Program:

```
%Linearity property of DFT
clc;
close all;
clear all;

x1 = input("Enter sequence 1:");
x2 = input("Enter sequence 2:");
N=max(length(x1),length(x2));
x1new=[x1 zeros(1,N-length(x1))];
x2new=[x2 zeros(1,N-length(x2))];
a = 2;
b = 3;

X1 = fft(x1new);
X2 = fft(x2new);

LHS = fft(a * x1new + b * x2new); % DFT of linear combination
RHS = a * X1 + b * X2;      % Linear combination of DFTs
disp("LHS:");
disp(round(LHS, 5));
disp("RHS:");
disp(round(RHS, 5));
% Check if the values match
if isequal(round(LHS, 5), round(RHS, 5))
    disp('Linearity property verified!');
else
    disp('Linearity property not verified.');
end
```

Observation

Sequence 1:[1 2 3 4]
Sequence 2:[1 1 0]
DFT{x1(n)*x2(n)}:
   3.0000 + 0.0000i   1.0000 - 2.0000i -1.0000 + 0.0000i   1.0000 + 2.0000i

X1(k)circonvX2(k)/N:
   3.0000 + 0.0000i   1.0000 - 2.0000i -1.0000 + 0.0000i   1.0000 + 2.0000i

Multiplication property verified!

Program:

```
%Multiplication property of DFT
clc;
close all;
clear all;
x1 = input("Sequence 1:");
x2 = input("Sequence 2:");
N=max(length(x1),length(x2));
x1new=[x1 zeros(1,N-length(x1))];
x2new=[x2 zeros(1,N-length(x2))];
product_time = x1new .* x2new;
dft_product_time=fft(product_time);
X1 = fft(x1new);
X2 = fft(x2new);
%Finding circular convolution of X1 and X2 using inbuilt function
Y=cconv(X1,X2,N);
%Display

disp("DFT{x1(n)*x2(n)}:");
disp(dft_product_time);
disp("X1(k)circonvX2(k)/N:");
disp(Y./N);
% Check if the values match
if isequal(round(dft_product_time, 5), round(Y./N, 5))
    disp('Multiplication property verified!');
else
    disp('Multiplication property not verified.');
end
```

Observation:

```
Enter sequence 1:[1 2 3 2]
Enter sequence 2:[1 2 1]
x1(n) cconv x2(n):
    8    6    8   10

IDFT{X1(k)*X2(k)}:
    8    6    8   10

Circular convolution property verified!
```

Program:

```
%Circular convolution property of DFT
clc;
close all;
clear all;

x1 = input("Enter sequence 1:");
x2 = input("Enter sequence 2:");
N=max(length(x1),length(x2));
x1new=[x1 zeros(1,N-length(x1))];
x2new=[x2 zeros(1,N-length(x2))];
X1 = fft(x1new);
X2 = fft(x2new);

circular_conv_time = cconv(x1new, x2new, N);
product_freq = ifft(X1 .* X2);

disp("x1(n) cconv x2(n):");
disp(circular_conv_time);
disp("IDFT{X1(k)*X2(k)}:");
disp(product_freq);
% Check if the values match
if isequal(round(circular_conv_time, 5), round(product_freq, 5))
    disp('Circular convolution property verified!');
else
    disp('Circular convolution property not verified.');
end
```

Observation:

```
Enter sequence 1:[1 9 2 8]
Enter sequence 2:[1 4 5 0]
Sum{n:0->N-1 ;x1(n)*conj(x2(n))}:
   47

Sum{k:0->N-1 ;X1(k)*conj(X2(k))}/N:
   47

Parsevals relation verified!
```

Program:

```
%Parsevals Relation for DFT
clc;
close all;
clear all;
x1 = input("Enter sequence 1:");
x2 = input("Enter sequence 2:");
N =max(length(x1),length(x2));
x1new=[x1 zeros(1,N-length(x1))];
x2new=[x2 zeros(1,N-length(x2))];

time_domain_value = sum(x1new.*conj(x2new));
freq_domain_value = sum(fft(x1new).*conj(fft(x2new)))./ N;

disp("Sum{n:0->N-1 ;x1(n)*conj(x2(n))}:");
disp(time_domain_value);
disp("Sum{k:0->N-1 ;X1(k)*conj(X2(k))}/N:");
disp(freq_domain_value);
% Check if the values match
if isequal(round(time_domain_value, 5), round(freq_domain_value, 5))
    disp('Parsevals relation verified!');
else
    disp('Parsevals relation not verified.');
end
```

Result: Verified the properties of DFT.

Experiment No: 8 Date: 03-10-2024

Overlap Add and Overlap Save methods for Linear Convolution

Aim: To perform linear convolution of two sequences using Overlap Add and Overlap Save methods.

Theory:

In digital signal processing, linear convolution of long sequences is often inefficient when performed directly, especially when the sequences are large. To address this, two popular methods are used: **Overlap-Add** and **Overlap-Save**. Both methods use the **Fast Fourier Transform (FFT)** to speed up the convolution process and are suitable for processing long signals in smaller segments.

1. Overlap-Add Method

The **Overlap-Add** (OLA) method divides the input signal into smaller, non-overlapping segments, performs convolution on each segment, and then combines (adds) the overlapping portions of the results.

2. Overlap-Save Method

The **Overlap-Save** (OLS) method, in contrast, uses overlapping input signal segments to perform the convolution and discards the unwanted portions of the result. This method is particularly useful when performing convolution on a continuous stream of data.

Program:

```
%Overlap Add Method
clc;                    % Clear command window
clear all;              % Clear workspace variables
close all;              % Close all figures

% Input the sequences and the length of each block
x = input("enter x:");  % Input signal
h = input("enter h:");  % Impulse response/filter
N = input("enter length to divide:");  % Input length for block processing

% Check if N is smaller than the length of the filter
if N < length(h)
    disp("not possible");  % If N is too small, display an error message
else
    % Get the lengths of the input sequences
    xl = length(x);    % Length of input signal x
    hl = length(h);    % Length of impulse response h

    % Zero-padding the filter h to make its length N
    hnew = [h, zeros(1, N-hl)];
```

Observation:

```
enter x:[1 2 3 4 5 6 7 8 9]
enter h:[1 2]
enter length to divide:4
Linear convolution using Overlap Add method( :
    1     4     7    10    13    16    19    22    25    18

Linear convolution using inbuilt function:
    1     4     7    10    13    16    19    22    25    18
```

```
    % Calculate how many blocks will be processed
    L = N - hl + 1;  % Length of each block to process
    totalBlocks = ceil(xl / L);  % Total number of blocks

    % Zero-padding the input signal to make it a multiple of the block length
    xnew = [x, zeros(1, totalBlocks*L - xl)];

    % Initialize the result array y, large enough to hold the full result
    y = zeros(1, length(xnew) + hl - 1);

    % Loop through the signal in blocks of length L (without overlap)
    for i = 1:L:length(xnew)
        % Extract the current block from the input signal
        XB = xnew(i:min(i+L-1, length(xnew)));  % Get the current block

        % Zero-padding the current block to length N
        XB = [XB, zeros(1, N - length(XB))];

        % Perform FFT-based convolution: FFT, multiply in frequency domain, then
IFFT
        YB = ifft(fft(XB) .* fft(hnew));

        % Add the result to the output signal at the appropriate location
        y(i:i+N-1) = y(i:i+N-1) + YB;  % Overlap-Add the result
    end

    % Display the final convolution result
    disp("Linear convolution using Overlap Add method( :")
    disp(y(1:xl+hl-1));
    disp("Linear convolution using inbuilt function:")
    disp(conv(x,h));
end
```

Observation:

```
enter x:[1 2 3 4 5 6 7 8 9]
enter h:[1 2]
enter length to divide:4
Linear convolution using Overlap Save Method :
    1     4     7    10    13    16    19    22    25    18

Linear convolution using inbuilt function:
    1     4     7    10    13    16    19    22    25    18
```

Program:

```
%Overlap Save Method
clc;
clear all;
close all;

% Input the sequences and the length of each block
x = input("enter x:");  % Input signal
h = input("enter h:");  % Impulse response
N = input("enter length to divide:");  % Input length for block processing

% Check if N is smaller than the length of the filter
if N < length(h)
    disp("not possible");  % If N is too small, display an error message
else
    % Get the lengths of the input sequences
    xl = length(x);   % Length of input signal x
    hl = length(h);   % Length of impulse response h

    % Calculate the number of elements to process in each block
    L = N - hl + 1;

    % Zero-padding the filter h to make its length N
    hnew = [h, zeros(1, N-hl)];

    % Zero-padding the input signal x with hl-1 zeros at the beginning and
    % N-1 zeros at the end to align with the filter
    xnew = [zeros(1, hl-1), x, zeros(1, N-1)];

    % Initialize the result array y
    y = [];

    % Loop through the signal in blocks of length N
    for i = 1:L:length(xnew) - N + 1
        % Extract the current block from the input signal
        XB = xnew(i:i+N-1);

        % Perform FFT-based convolution: FFT, multiply in frequency domain, then
IFFT
        YB = ifft(fft(XB) .* fft(hnew));

        % Append the useful part of the result (discard the first hl-1 elements)
        y = [y, YB(hl:end)];
    end

    % Display the final convolution result
    disp("Linear convolution using Overlap Save Method :")
    disp(y(1:xl+hl-1));
    disp("Linear convolution using inbuilt function:")
    disp(conv(x,h));
end
```

Result: Performed Linear convolution using Overlap Add and Overlap Save Methods.

Experiment No: 9 Date:15/10/24

IMPLEMENTATION OF FIR FILTERS

Aim:

Implement various FIR filters using different windows

1. Low Pass Filter
2. High Pass Filter
3. Band pass Filter
4. Band stop Filter

Theory:

Design of FIR Filters Using Window Methods

In FIR (Finite Impulse Response) filter design, the goal is to create a filter with specific frequency response characteristics, such as low-pass, high-pass, band-pass, or band-stop. Using window methods, we can shape the filter response by applying a window function to an ideal filter impulse response.

Step 1: Define the Ideal Impulse Response

The ideal impulse response, h_ideal(n), of a low-pass filter with a cutoff frequency f_c is given by:

h_ideal(n) = sin(2 * pi * f_c * (n - (N - 1) / 2)) / (pi * (n - (N - 1) / 2))

Where:

•f_c: Normalized cut off frequency
• N: Filter length
• n: Sample index

Step 2: Select an Appropriate Window Function

The choice of window affects the trade-off between the main lobe width and the sidelobe levels. Common windows include the Rectangular, Hamming, Hanning, Blackman, and Kaiser windows.

Window Type	Formula
Rectangular	w(n) = 1
Triangular	w(n) = 1 - 2*abs(n) / (N - 1)
Hamming	w(n) = 0.54+ 0.46 * cos(2 * pi * n / (N - 1))
Hanning	w(n) = 0.5 * (1 +cos(2 * pi * n / (N - 1)))
Blackman	w(n) = 0.42 + 0.5 * cos(2 * pi * n / (N - 1)) + 0.08 * cos(4 * pi * n / (N - 1))

Kaiser	w(n) = I0(beta * sqrt(1 - (2 * n / (N - 1) - 1)^2)) / I0(beta)

Step 3: Apply the Window to the Ideal Impulse Response
The windowed impulse response is computed as:

h(n) = h_ideal(n) * w(n)

Step 4: Construct the FIR Filter
The final impulse response h(n) defines the FIR filter coefficients that can be used in filtering algorithms.

Filters:

Lowpass: $$h(n) = \begin{cases} \frac{\Omega_c}{\pi} & n = 0 \\ \frac{\sin(\Omega_c n)}{n\pi} \text{ for } n \neq 0 & -M \leq n \leq M \end{cases}$$

Highpass: $$h(n) = \begin{cases} \frac{\pi - \Omega_c}{\pi} & n = 0 \\ -\frac{\sin(\Omega_c n)}{n\pi} \text{ for } n \neq 0 & -M \leq n \leq M \end{cases}$$

Bandpass: $$h(n) = \begin{cases} \frac{\Omega_H - \Omega_L}{\pi} & n = 0 \\ \frac{\sin(\Omega_H n)}{n\pi} - \frac{\sin(\Omega_L n)}{n\pi} \text{ for } n \neq 0 & -M \leq n \leq M \end{cases}$$

Bandstop: $$h(n) = \begin{cases} \frac{\pi - \Omega_H + \Omega_L}{\pi} & n = 0 \\ -\frac{\sin(\Omega_H n)}{n\pi} + \frac{\sin(\Omega_L n)}{n\pi} \text{ for } n \neq 0 & -M \leq n \leq M \end{cases}$$

Advantages and Disadvantages of Window-Based FIR Design
Advantages:
• Simplicity: Windowing is straightforward and does not require iterative optimization.
• Control over Leakage: Different windows provide different control over sidelobes and main lobe width.
Disadvantages:
• Fixed Frequency Response: Once the window is chosen, the frequency response characteristics are determined.
• Trade-Off Limitations: Some applications require specific frequency responses that cannot be perfectly achieved using standard windows.

Program:

1. LOW PASS FILTER

```
clc;
clear all;
close all;
wc=0.5*pi;
N = 50;
alpha = (N-1)/2;
eps = 0.001;
```

```
n = 0:1:N-1;
hd = (sin(wc*(n-alpha+eps)))./(pi*(n-alpha+eps));
wr = boxcar(N);
wt=bartlett(N);
wh=hamming(N);
whn=hanning(N);
hn1 = hd.*wr';
hn2 = hd.*wt';
hn3 = hd.*wh';
hn4 = hd.*whn';
w = 0:0.01:pi;
h1 = freqz(hn1,1,w);
h2 = freqz(hn2,1,w);
h3 = freqz(hn3,1,w);
h4 = freqz(hn4,1,w);
subplot(3,3,1);
plot(w/pi,10*log10(abs(h1)));
title('low pass filter using rectangular window');
xlabel('Normalized frequency');
ylabel('Magnitude in dB');
subplot(3,3,2);
stem(wr);
title('Rectangular window Sequence');
xlabel('No. of Samples');
ylabel('Amplitude');
subplot(3,3,3);
plot(w/pi,10*log10(abs(h2)));
title('low pass filter using triangular window');
xlabel('Normalized frequency');
ylabel('Magnitude in dB');
```

```
subplot(3,3,4);
stem(wt);
title('Triangular window Sequence');
xlabel('No. of Samples');
ylabel('Amplitude');
subplot(3,3,5);
plot(w/pi,10*log10(abs(h3)));
title('low pass filter using hamming window');
xlabel('Normalized frequency');
ylabel('Magnitude in dB');
subplot(3,3,6);
stem(wh);
title('Hanning window Sequence');
xlabel('No. of Samples');
ylabel('Amplitude');
subplot(3,3,7);
plot(w/pi,10*log10(abs(h4)));
title('low pass filter using hanning window');
xlabel('Normalized frequency');
ylabel('Magnitude in dB');
subplot(3,3,8);
stem(whn);
title('Hanning window Sequence');
xlabel('No. of Samples');
ylabel('Amplitude');
```

2.HIGH PASS FILTER

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

```
wc=0.5*pi;

N = 50;

alpha = (N-1)/2;

eps = 0.001;

n = 0:1:N-1;

hd=(sin(pi*(n-alpha+eps))-sin(wc*(n-alpha+eps)))./(pi*(n-
alpha+eps));

wr = boxcar(N);

wt=bartlett(N);

wh=hamming(N);

whn=hanning(N);

hn1 = hd.*wr';

hn2 = hd.*wt';

hn3 = hd.*wh';

hn4 = hd.*whn';

w = 0:0.01:pi;

h1 = freqz(hn1,1,w);

h2 = freqz(hn2,1,w);

h3 = freqz(hn3,1,w);

h4 = freqz(hn4,1,w);

subplot(3,3,1);

plot(w/pi,10*log10(abs(h1)));

title('high pass filter using rectangular window');

xlabel('Normalized frequency');

ylabel('Magnitude in dB');

subplot(3,3,2);

stem(wr);

title('Rectangular window Sequence');

xlabel('No. of Samples');

ylabel('Amplitude');
```

```
subplot(3,3,3);
plot(w/pi,10*log10(abs(h2)));
title('high pass filter using triangular window');
xlabel('Normalized frequency');
ylabel('Magnitude in dB');
subplot(3,3,4);
stem(wt);
title('Triangular window Sequence');
xlabel('No. of Samples');
ylabel('Amplitude');
subplot(3,3,5);
plot(w/pi,10*log10(abs(h3)));
title('high pass filter using hamming window');
xlabel('Normalized frequency');
ylabel('Magnitude in dB');
subplot(3,3,6);
stem(wh);
title('Hanning window Sequence');
xlabel('No. of Samples');
ylabel('Amplitude');
subplot(3,3,7);
plot(w/pi,10*log10(abs(h4)));
title('high pass filter using hanning window');
xlabel('Normalized frequency');
ylabel('Magnitude in dB');
subplot(3,3,8);
stem(whn);
title('Hanning window Sequence');
xlabel('No. of Samples');
ylabel('Amplitude');
```

3.BANDPASS FILTER

```
clc;
clear all;
close all;
wc1=0.5*pi;
wc2=0.9*pi;
N = 50;
alpha = (N-1)/2;
eps = 0.001;
n = 0:1:N-1;
hd    =    (sin(wc2*(n-alpha+eps))-sin(wc1*(n-alpha+eps)))./(pi*(n-
alpha+eps));
wr = boxcar(N);
wt=bartlett(N);
wh=hamming(N);
whn=hanning(N);
hn1 = hd.*wr';
hn2 = hd.*wt';
hn3 = hd.*wh';
hn4 = hd.*whn';
w = 0:0.01:pi;
h1 = freqz(hn1,1,w);
h2 = freqz(hn2,1,w);
h3 = freqz(hn3,1,w);
h4 = freqz(hn4,1,w);
subplot(3,3,1);
plot(w/pi,10*log10(abs(h1)));
title('band pass filter using rectangular window');
xlabel('Normalized frequency');
```

```
ylabel('Magnitude in dB');
subplot(3,3,2);
stem(wr);
title('Rectangular window Sequence');
xlabel('No. of Samples');
ylabel('Amplitude');
subplot(3,3,3);
plot(w/pi,10*log10(abs(h2)));
title('band pass filter using triangular window');
xlabel('Normalized frequency');
ylabel('Magnitude in dB');
subplot(3,3,4);
stem(wt);
title('Triangular window Sequence');
xlabel('No. of Samples');
ylabel('Amplitude');
subplot(3,3,5);
plot(w/pi,10*log10(abs(h3)));
title('band pass filter using hamming window');
xlabel('Normalized frequency');
ylabel('Magnitude in dB');
subplot(3,3,6);
stem(wh);
title('Hanning window Sequence');
xlabel('No. of Samples');
ylabel('Amplitude');
subplot(3,3,7);
plot(w/pi,10*log10(abs(h4)));
title('band pass filter using hanning window');
xlabel('Normalized frequency');
```

```
ylabel('Magnitude in dB');
subplot(3,3,8);
stem(whn);
title('Hanning window Sequence');
xlabel('No. of Samples');
ylabel('Amplitude');
```

4.BANDSTOP FILTER

```
clc;
clear all;
close all;
wc1=0.5*pi;
wc2=0.9*pi;
N = 50;
alpha = (N-1)/2;
eps = 0.001;
n = 0:1:N-1;
hd    =    (sin(wc1*(n-alpha+eps))-sin(wc2*(n-alpha+eps))+sin(pi*(n-
alpha)))./(pi*(n-alpha+eps));
wr = boxcar(N);
wt=bartlett(N);
wh=hamming(N);
whn=hanning(N);
hn1 = hd.*wr';
hn2 = hd.*wt';
hn3 = hd.*wh';
hn4 = hd.*whn';
w = 0:0.01:pi;
h1 = freqz(hn1,1,w);
h2 = freqz(hn2,1,w);
```

```
h3 = freqz(hn3,1,w);
h4 = freqz(hn4,1,w);
subplot(3,3,1);
plot(w/pi,10*log10(abs(h1)));
title('band stop filter using rectangular window');
xlabel('Normalized frequency');
ylabel('Magnitude in dB');
subplot(3,3,2);
stem(wr);
title('Rectangular window Sequence');
xlabel('No. of Samples');
ylabel('Amplitude');
subplot(3,3,3);
plot(w/pi,10*log10(abs(h2)));
title('band stop filter using triangular window');
xlabel('Normalized frequency');
ylabel('Magnitude in dB');
subplot(3,3,4);
stem(wt);
title('Triangular window Sequence');
xlabel('No. of Samples');
ylabel('Amplitude');
subplot(3,3,5);
plot(w/pi,10*log10(abs(h3)));
title('band stop filter using hamming window');
xlabel('Normalized frequency');
ylabel('Magnitude in dB');
subplot(3,3,6);
stem(wh);
title('Hanning window Sequence');
```

```
xlabel('No. of Samples');
ylabel('Amplitude');
subplot(3,3,7);
plot(w/pi,10*log10(abs(h4)));
title('band stop filter using hanning window');
xlabel('Normalized frequency');
ylabel('Magnitude in dB');
subplot(3,3,8);
stem(whn);
title('Hanning window Sequence');
xlabel('No. of Samples');
ylabel('Amplitude');
```

Result:

Implemented various FIR filters using different windows

1Low Pass Filter

2.High Pass Filter

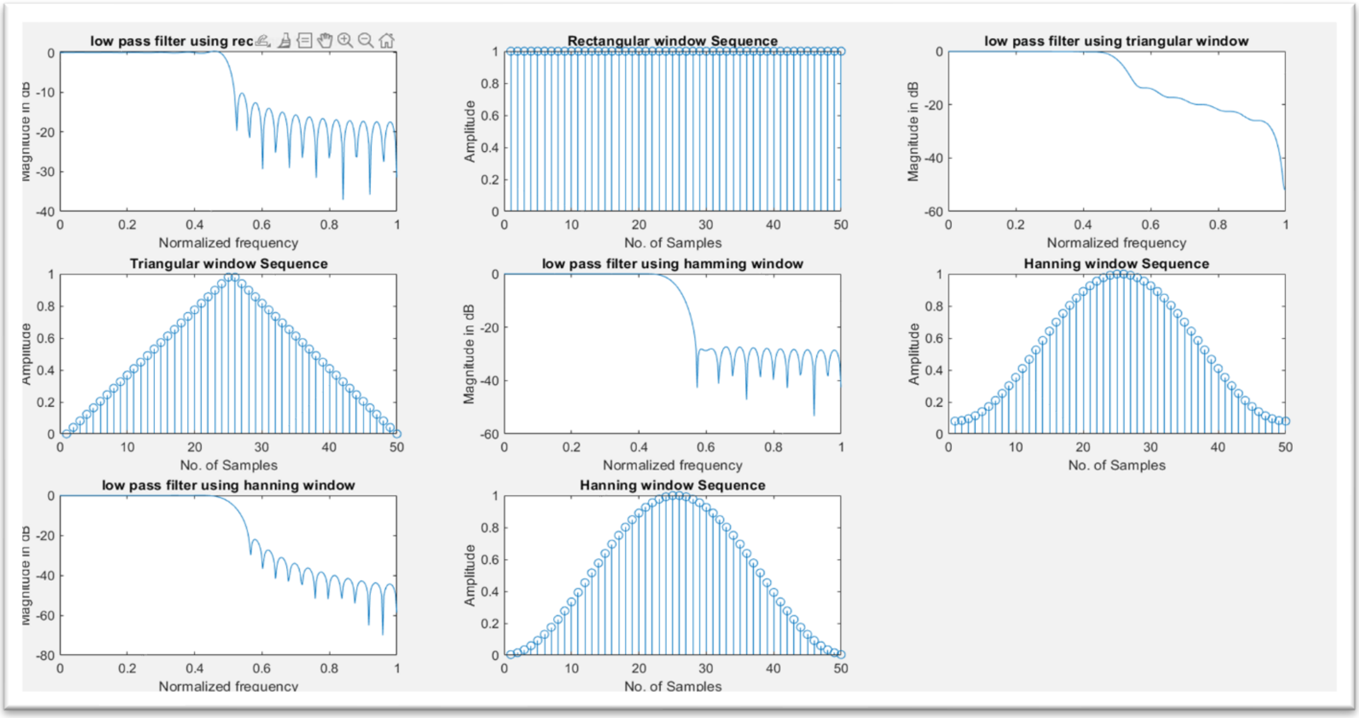
3.Band pass Filter

4.Band stop Filter

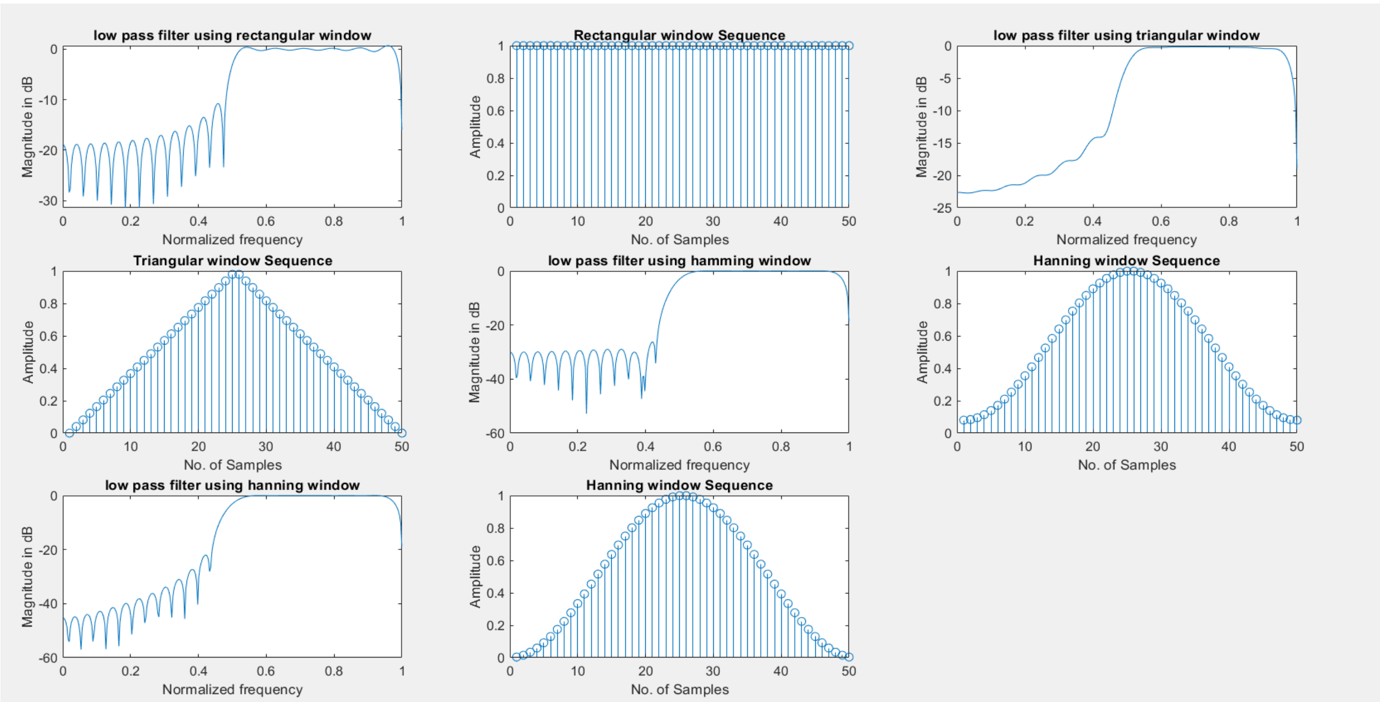
.

Observation:

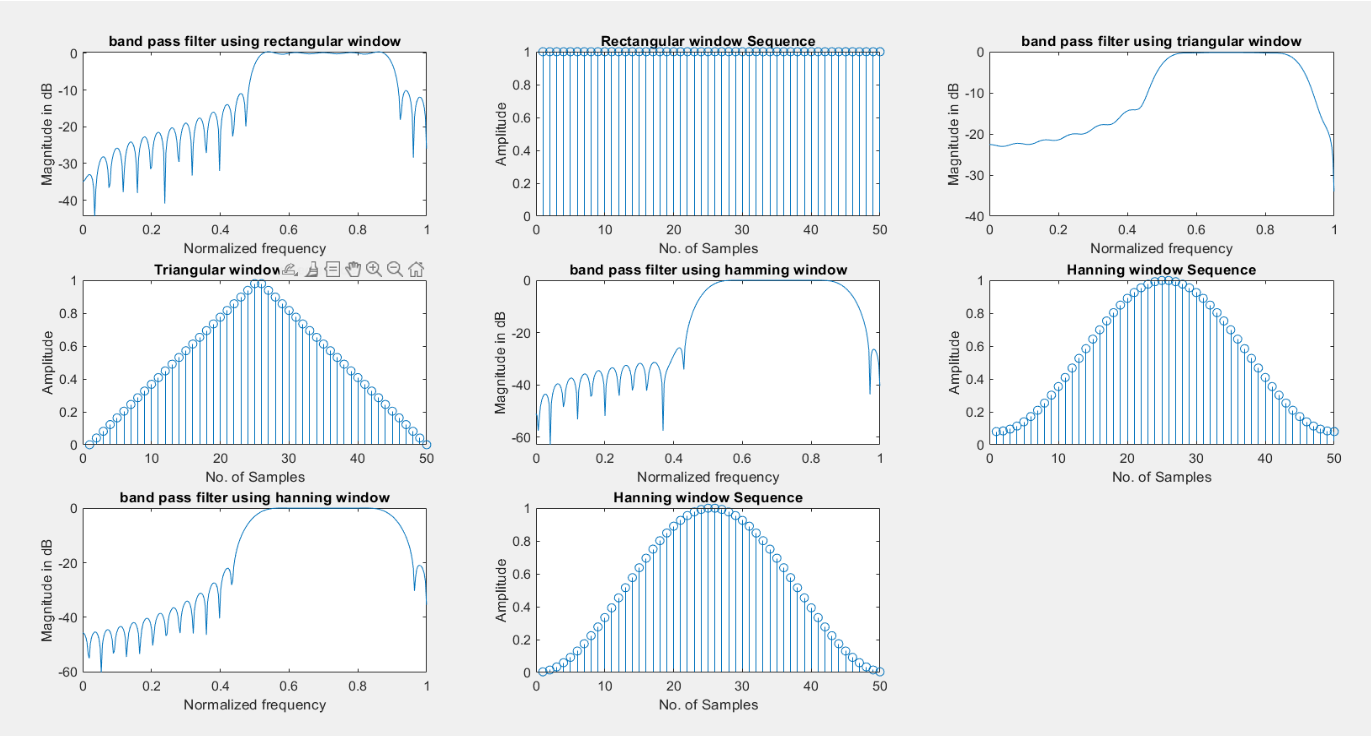
1.LOW PASS FILTER



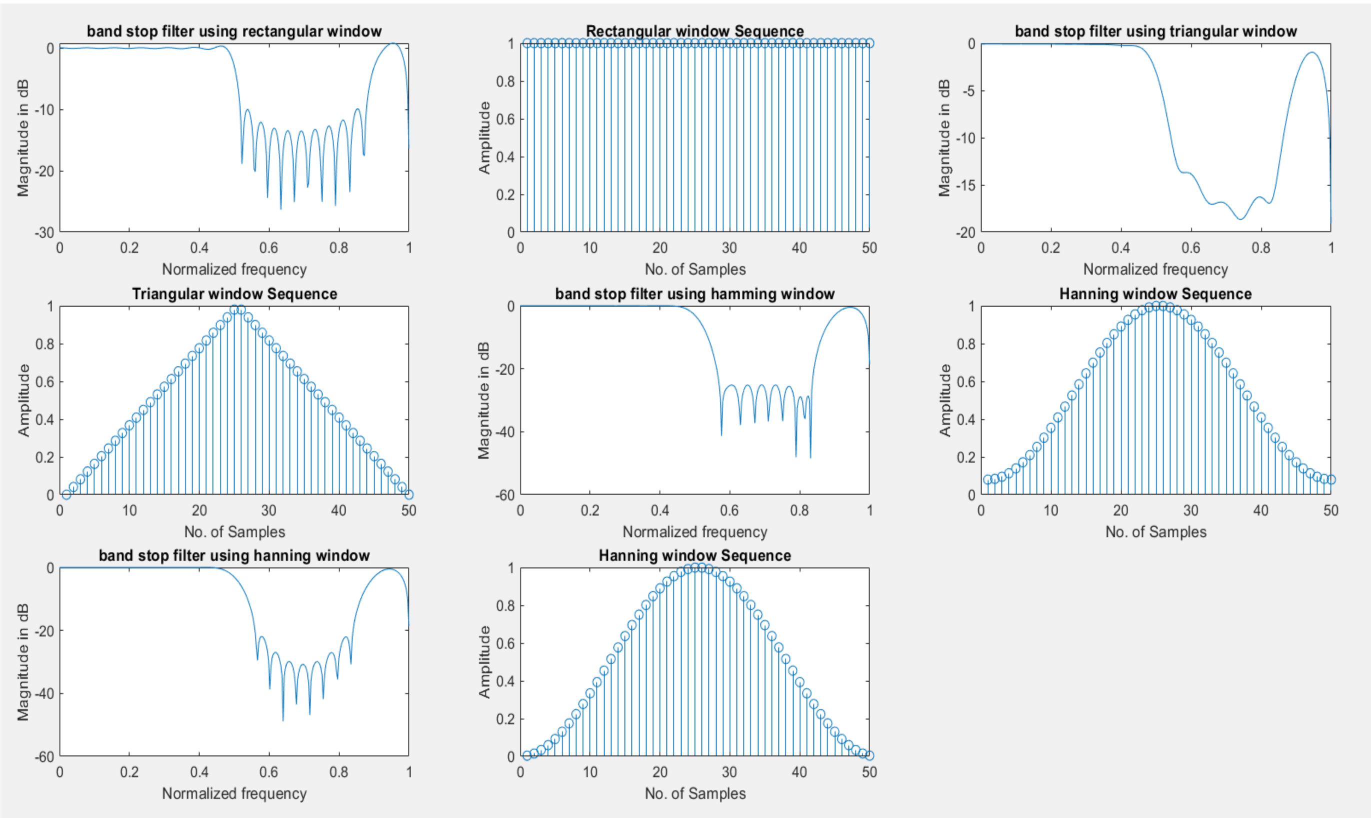
2.HIGH PASS FILTER



3.BAND PASS FILTER



4.BAND STOP FILTER



Experiment No: 10 Date: 22/10/24

FAMILIARIZATION OF THE ANALOG AND DIGITAL INPUT AND OUTPUT PORTS OF DSP BOARD

Aim:

Familiarization of the analog and digital input and output ports of DSP Boards.

Theory:

TMS 320C674x DSP CPU

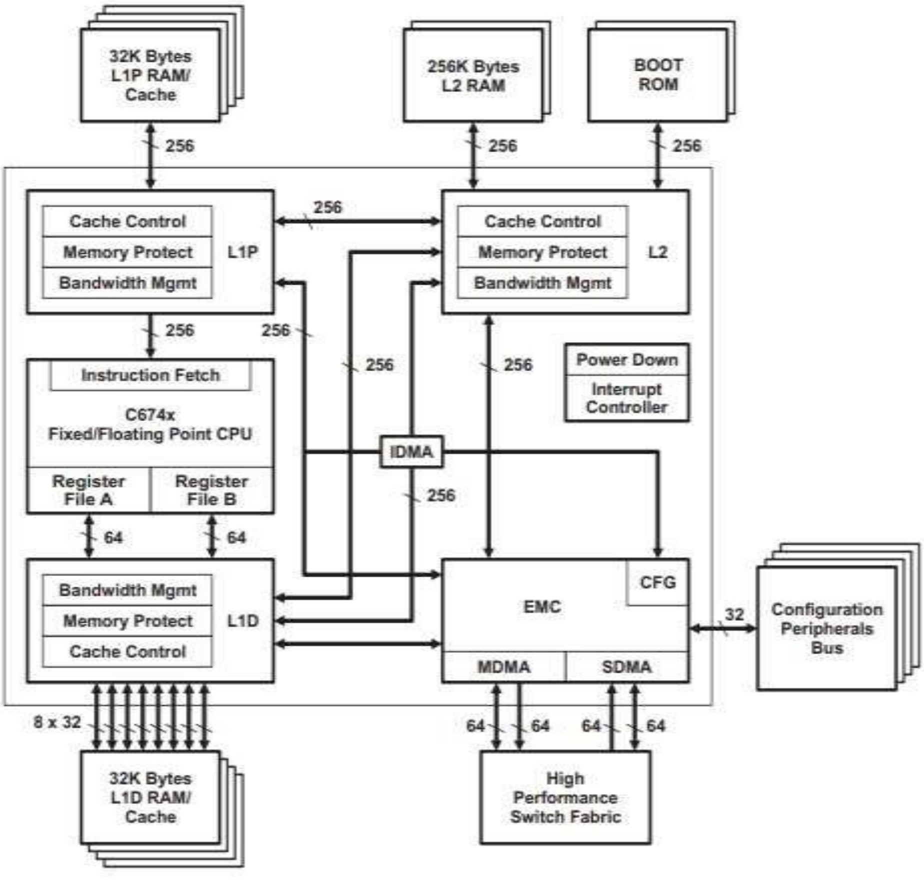


FIGURE: TMS320C 674X DSP CPU BLOCK DIAGRAM

The TMS320C674X DSP CPU consists of eight functional units, two register files, and two data paths as shown in Figure. The two general-purpose register files (A and B) each contain 32 32- bit registers for a total of 64 registers. The general-purpose registers can be used for data or can be data address pointers. The data types supported include packed 8-bit data, packed 16-bit data, 32-bit data, 40- bit data, and 64-bit data. Values larger than 32 bits, such as

40-bit-long or 64-bit-long values are stored in register pairs, with the 32 LSBs of data placed in an even register and the remaining 8 or 32 MSBs in the next upper register (which is always an odd-numbered register). The eight functional units (.M1, .L1, .D1, .S1, .M2, .L2, .D2, and .S2) are each capable of executing one instruction every clock cycle. The .M functional units perform all multiply operations. The .S and .L units perform a general set of arithmetic, logical, and branch functions. The .D units primarily load data from memory to the register file and store results from the register file into memory.

Multichannel Audio Serial Port (McASP):

The McASP serial port is specifically designed for multichannel audio applications.

Its key features are:

• Flexible clock and frame sync generation logic and on-chip dividers

• Up to sixteen transmit or receive data pins and serializers

• Large number of serial data format options, including: – TDM Frames with 2 to 32 time slots per frame (periodic) or 1 slot per frame (burst) – Time slots of 8,12,16, 20, 24, 28, and 32 bits – First bit delay 0, 1, or 2 clocks – MSB or LSB first bit order – Left- or right-aligned datawords within time slots

• DIT Mode with 384-bit Channel Status and 384-bit User Data registers

• Extensive error checking and mute generation logic

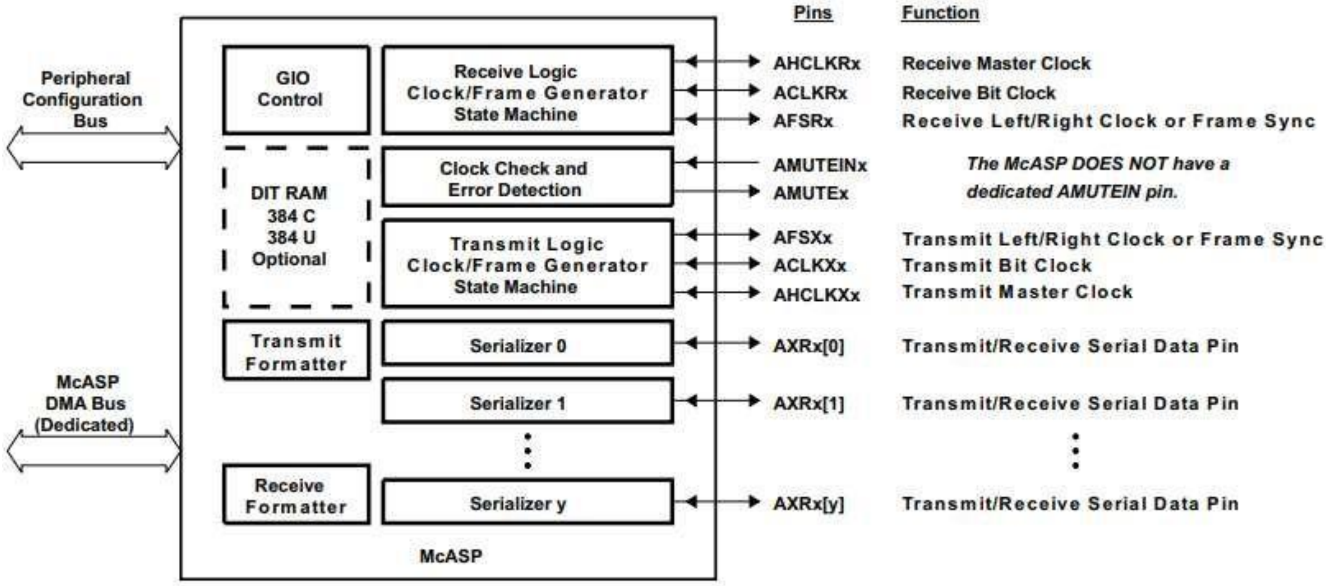
• All unused pins GPIO-capable

• Transmit & Receive FIFO Buffers allow the McASP to operate at a higher sample rate by making it more tolerant to DMA latency.

• Dynamic Adjustment of Clock Dividers – Clock Divider Value may be changed without resetting the McASP. The DSK board includes the TLV320AIC23 (AIC23) codec for input and output.

The ADC circuitry on the codec converts the input analog signal to a digital representation to be processed by the DSP. The maximum level of the input signal to be converted is determined by the specific ADC circuitry on the codec, which is 6 V p-p with the onboard codec. After the captured signal is processed, the result needs.

to be sent to the outside world. DAC, which performs the reverse operation of the ADC. An output filter smooths out or reconstructs the output signal. ADC, DAC, and all required filtering functions are performed by the single-chip codec AIC23 on board the DSK.



Result:

Familiarized the input and output ports of dsp board.

Experiment No: 11 Date: 29/10/24

Generation of Sine Wave using DSP Kit

Aim:

To generate a sine wave using DSP Kit.

Theory:

Sinusoidal are the smoothest signals with no abrupt variation in their amplitude, the amplitude witnesses gradual change with time. Sinusoidal signals can be defined as a periodic signal with waveform as that of a sine wave. The amplitude of sine wave increases from a value of 0 at 0° angle to a maximum value of 1 at 90°, it further reaches its minimum value of -1 at 270° and then returns to 0 at 360°. After any angle greater than 360°, the sinusoidal signal repeats the values so we can say that period of sinusoidal signal is 2π i.e. 360°.If we observe the graph, we can see that the amplitude varying gradually with a maximum value of 1 and a minimum value of -1. We can also observe that the wave begins to repeat its value after a period or angle value of 2π hence periodicity of sinusoidal signal is 2π.

$$\mathbf{y(t)=A\sin(\omega t+\phi)+C}$$

Procedure

1. Open Code Composer Studio, Click on File - New – CCS Project
Select the Target – C674X Floating point DSP , TMS320C6748 , and
Connection – Texas Instruments XDS 100v2 USB Debug Probe and Verify.
Give the project name and select Finish.
2. Type the code program for generating the sine wave and choose
File – Save As and then save the program with a name including ‘main.c’.
Delete the already existing main.c program.
3. Select Debug and once finished, select the Run option.
4. From the Tools Bar, select Graphs – Single Time.
Select the DSP Data Type as 32-bit Floating point and time display unit as second(s).
Change the Start address with the array name used in the program(here,s).
5. Click OK to apply the settings and Run the program or clock Resume in CCS.

Program:

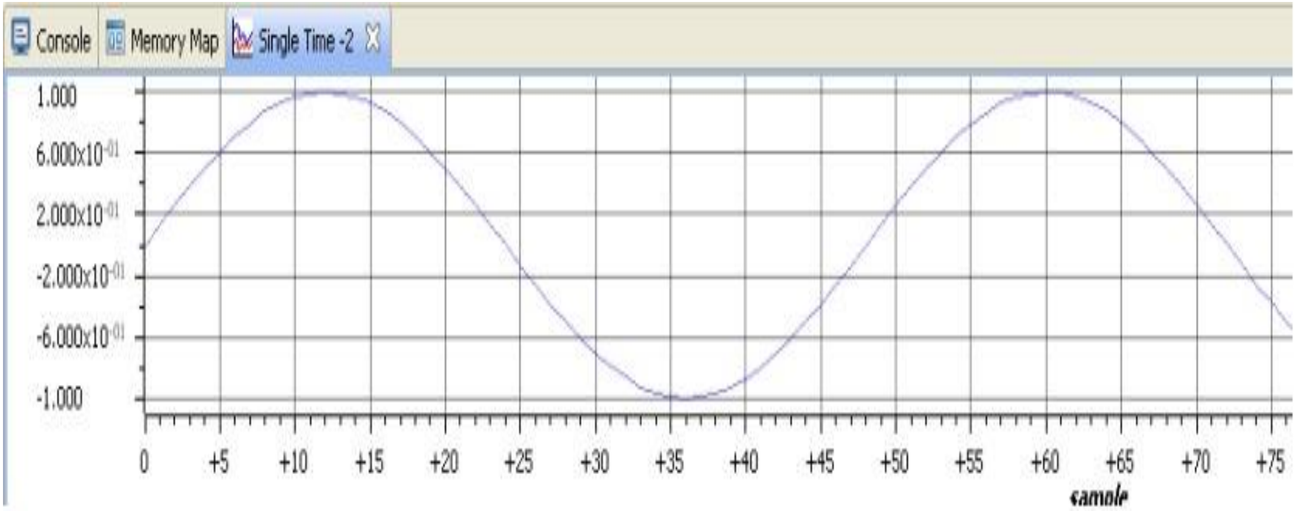
```
#include<stdio.h>
#include<math.h>
#define pi 3.14159
float s[100];
void main()
{
int i;
float f=100, Fs=10000;
for(i=0;i<100;i++)
```

```
s[i]=sin(2*pi*f*i/Fs);
}
```

Result:

Generated sine wave using DSP Kit.

Observation:



Experiment No:12 Date:29/10/24

Linear Convolution using DSP Kit

Aim:

To perform linear convolution of two sequences using DSP Kit.

Theory:

Linear convolution is one of the fundamental operations used extensively in signal and system in electrical engineering. It has applications in areas like audio processing, signal filtering, imaging, communication systems and more. In simple terms, linear convolution is the process of combining two signals or functions to produce a third signal or function.
Formally, the linear convolution of two functions f(t) and g(t) is defined as:
The formula for linear convolution of two discrete signals x[n] and h[n] is given by:

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

In the context of linear convolution in DSP, this operation is applied to digital signals. DSP systems utilize algorithms to perform convolution efficiently, often leveraging Fast Convolution methods to handle large datasets and real-time processing.

Procedure

1. Set Up New CCS Project
Open Code Composer Studio.
Go to File → New → CCS Project.
Target Selection: Choose C674X Floating point DSP, TMS320C6748.
Connection: Select Texas Instruments XDS 100v2 USB Debug Probe.
Name the project and click Finish.

2. Write and Configure the Program
Write the C code for generating and storing a sine wave, configuring it to access data at specified memory locations.
Assign the input Xn and filter Hn values to specified addresses:
Xn: Start at 0x80010000, populate subsequent values at offsets like 0x80010004 for each additional input.
Hn: Start at 0x80011000 with similar offsets for additional values.
Lengths of Xn and Hn should be defined at 0x80012000 and 0x80012004, respectively.

3. Configure Output Location in Code
In the code, configure the output to store convolution results at specific memory addresses starting from 0x80013000, with each result at an offset of 0x04.

4. Save the Program
Go to File → Save As and save the code with a filename like main.c.
Remove any default main.c program that might exist in the project.

5. Build and Debug the Program
Select Debug to build and load the program on the DSP.
Once the build is complete, select Run to execute.

6. Execute and Verify Output
In the Debug perspective, click Resume to run the code.
Use the Memory Browser in Code Composer Studio to verify the output at the memory location 0x80013000:
Check 0x80013000 for the first convolution result, 0x80013004 for the second, and so on.
Cross-check the values with the expected convolution results for accuracy.

Program:

```
#include<fastmath67x.h>
#include<math.h>
void main()
{
int *Xn,*Hn,*Output;
int *XnLength,*HnLength;
int i,k,n,l,m;
Xn=(int *)0x80010000; //input x(n)
Hn=(int *)0x80011000; //input h(n)
XnLength=(int *)0x80012000; //x(n) length
HnLength=(int *)0x80012004; //h(n) length
Output=(int *)0x80013000; // output address
l=*XnLength; // copy x(n) from memory address to variable l
m=*HnLength; // copy h(n) from memory address to variable m
for(i=0;i<(l+m-1);i++) // memory clear
{
Output[i]=0; // o/p array
Xn[l+i]=0; // i/p array
Hn[m+i]=0; // i/p array
}
for(n=0;n<(l+m-1);n++)
{
for(k=0;k<=n;k++)
{
Output[n] =Output[n] + (Xn[k]*Hn[n-k]); // convolution operation.
}
}
}
```

Result:

Performed Linear Convolution using DSP Kit.

.

Observation:

Xn
0x80010000 – 1
0x80010004 – 2
0x80010008 – 3

Hn
0x80011000 – 1
0x80011004 – 2

XnLength
0x80012000 – 3

HnLength
0x80012004 – 2

Output
0x80013000 – 1
0x80013004 – 4
0x80013008 – 7
0x8001300C – 6