

# **THANGAL KUNJU MUSALIAR COLLEGE OF ENGINEERING**

**KOLLAM – 691 005**



## **ELECTRONICS AND COMMUNICATION ENGINEERING**

**22ECL509**

**DIGITAL SIGNAL PROCESSING LAB**

**LABORATORY RECORD**

**YEAR 2024-25**

## INDEX

SL No.	DATE	NAME OF THE EXPERIMENTS	PAGE NO.	REMARKS
1.	01/08/2024	Simulation of Basic Test Signals		
2.	08/08/2024	Verification of Sampling Theorem		
3.	08/08/2024	Linear Convolution		
4.	22/08/2024	Circular Convolution		
5.	12/09/2024	Linear Convolution using circular convolution and vice versa		
6.	29/09/2024	DFT and IDFT		
7.	29/09/2024	Properties of DFT		
8.	03/10/2024	Overlap Add and Overlap Save Method		
9.	17/10/2024	Implementation of FIR Filter		
10.	24/10/2024	Familiarization of DSP Hardware		
11.	24/10/2024	Generation of sine wave using DSP Kit		
12.	24/10/2024	Linear Convolution using DSP Kit		

**Experiment Name: Simulation of Basic Test Signals****Aim:**

To generate continuous and discrete waveforms for the following:

1. Unit Impulse Signal
2. Bipolar Pulse Signal
3. Unipolar Pulse Signal
4. Ramp Signal
5. Triangular Signal
6. Sine Signal
7. Cosine Signal
8. Exponential Signal
9. Unit Step Signal

**Theory:****1. Unit Impulse Signal:**

- A signal that is zero everywhere except at one point, typically at  $t=0$  where its value is 1.
- **Mathematically**  $\delta(t) = \begin{cases} \infty; & t = 0 \\ 0; & t \neq 0 \end{cases}$

**2. Bipolar Pulse Signal:**

- A pulse signal that alternates between positive and negative values, usually rectangular in shape. It switches between two constant levels (e.g., -1 and 1) for a defined duration.
- **Mathematically**  $p(t) = A$  for  $|t| \leq \tau/2$ ,  $p(t) = 0$  otherwise

**3. Unipolar Pulse Signal:**

- A pulse signal that alternates between zero and a positive value. It remains at zero for a specified duration and then jumps to a positive constant level (e.g., 0 and 1).
- **Mathematically**  $p(t) = A$  for  $|t| \leq \tau/2$ ,  $p(t) = 0$  otherwise (assuming A is positive)

#### 4. Ramp Signal:

- A signal that increases linearly with time.
- **Mathematically**  $r(t) = \begin{cases} t; & t \geq 0 \\ 0; & t < 0 \end{cases}$

#### 5. Triangular Signal:

- A periodic signal that forms a triangle shape, linearly increasing and decreasing with time, typically between a positive and negative peak.
- **Mathematically:**  $\Lambda(t) = 1 - |t|$  for  $|t| \leq 1$ ,  $\Lambda(t) = 0$  otherwise

#### 6. Sine Signal:

- A continuous periodic signal. It oscillates smoothly between -1 and 1.
- **Mathematically:**  $y(t) = A \sin(2\pi ft)$

#### 7. Cosine Signal:

- A continuous periodic signal like the sine wave but phase-shifted by  $\pi/2$ .
- **Mathematically:**  $y(t) = A \cos(2\pi ft)$

#### 8. Exponential Signal:

- A signal that increases or decreases exponentially with time. The rate of growth or decay is determined by the constant  $a$ .
- **Mathematically:**  $e^{(at)}$

#### 9. Unit Step Signal:

- A signal that is zero for all negative time values and one for positive time values.
- **Mathematically**  $u(t) = \begin{cases} 1; & t \geq 0 \\ 0; & t < 0 \end{cases}$

#### **Program:**

```
clc; clear
```

```
all; close
```

```
all;
```

```
subplot(3,3,1);  
t = -5:1:5;  
y = [zeros(1,5),ones(1,1),zeros(1,5)];  
stem(t,y);  
xlabel("Time(s)");  
ylabel("Amplitude");  
title("Unit Impulse Signal");
```

```
subplot(3,3,2);  
t2 = 0:0.01:1;  
f = 5;  
y2 = square(2*pi*f*t2);  
stem(t2,y2);  
hold on;  
plot(t2,y2);  
xlabel("Time(s)");  
ylabel("Amplitude");  
title("Bipolar Pulse Signal");  
legend("Discrete","Continuous");
```

```
subplot(3,3,3);  
t3 = 0:0.1:1;  
f = 5;  
y3 = abs(square(2*pi*f*t3));  
stem(t3,y3);  
hold on;  
plot(t3,y3);  
xlabel("Time(s)");  
ylabel("Amplitude");  
title("Unipolar Pulse Signal");  
legend("Discrete","Continuous");
```

```
subplot(3,3,4);  
t4 = -5:1:5;  
y4 = t4 .*(t4>=0);  
stem(t4,y4);  
hold on;  
plot(t4,y4);  
xlabel("Time(s)");  
ylabel("Amplitude");  
title("Unit Ramp Signal");  
legend("Discrete","Continuous");
```

```
subplot(3,3,5);  
t5 = 0:0.025:1;  
f = 10;  
y5 = sawtooth(2*pi*f*t5,0.5);  
stem(t5,y5);  
hold on;  
plot(t5,y5);  
xlabel("Time(s)");  
ylabel("Amplitude");  
title("Triangular Signal");  
legend("Discrete","Continuous");
```

```
subplot(3,3,6);  
t6 = 0:0.001:1;  
f = 10;  
y6 = sin(2*pi*f*t6);  
stem(t6,y6);  
hold on;  
plot(t6,y6);  
xlabel("Time(s)");
```

```
ylabel("Amplitude");  
title("Sine Wave");  
legend("Discrete","Continuous");
```

```
subplot(3,3,7);  
  
t7 = 0:0.001:1;  
f = 10;  
y7 = cos(2*pi*f*t7);  
stem(t7,y7);  
hold on;  
plot(t7,y7);  
xlabel("Time(s)");  
ylabel("Amplitude");  
title("Cosine Wave");  
legend("Discrete","Continuous");
```

```
subplot(3,3,8);  
  
t8 = -5:1:5;  
y8 = exp(t8);  
stem(t8,y8);  
hold on;  
plot(t8,y8);  
xlabel("Time(s)");  
ylabel("Amplitude");  
title("Exponential Signal");  
legend("Discrete","Continuous");
```

```
subplot(3,3,9);  
  
t9 = -5:1:5;  
y9 = [zeros(1,5),ones(1,6)];  
stem(t9,y9);
```

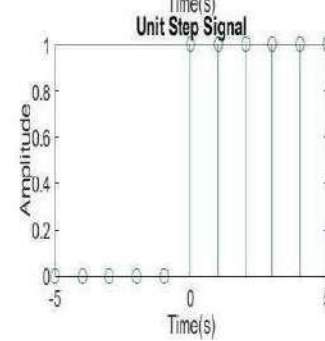
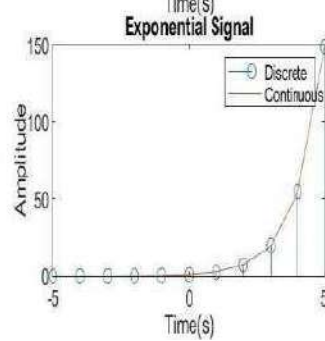
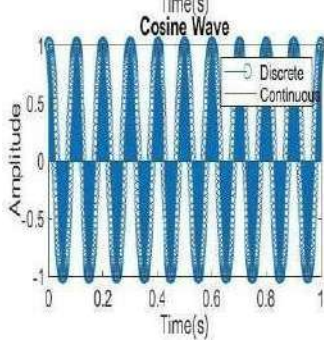
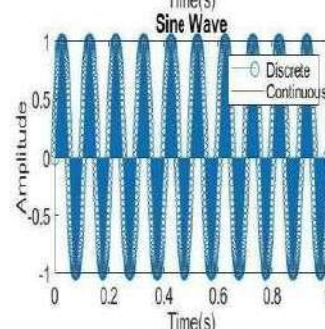
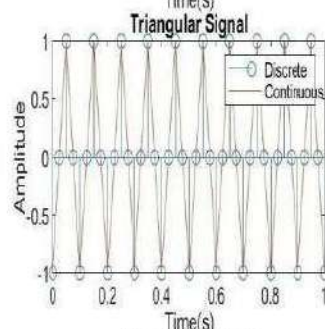
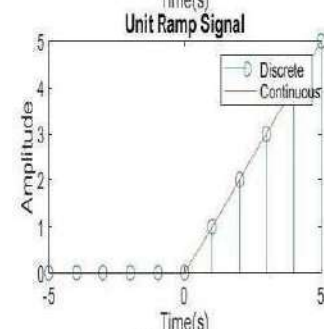
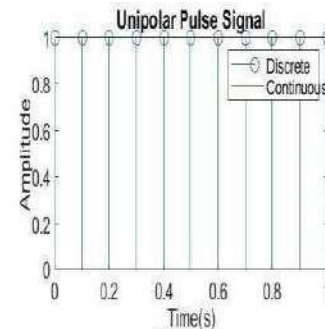
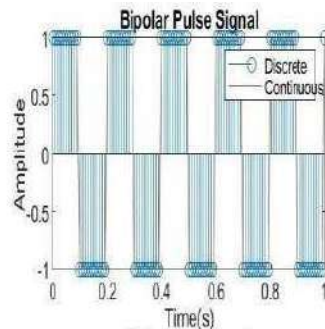
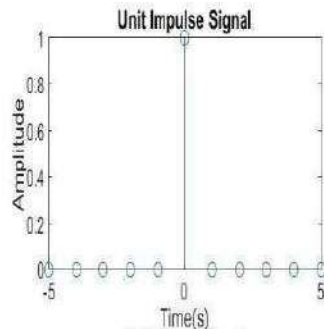
```
xlabel("Time(s)");  
ylabel("Amplitude");  
title("Unit Step Signal");
```

### **Result**

Generated and Verified various Continuous and Discrete waveforms for basic test signals.



**Observation**



Experiment No: 2

Date: 06/08/24

## Experiment Name: Verification of Sampling Theorem

### Aim:

To verify Sampling Theorem.

### Theory:

The Sampling Theorem, also known as the Nyquist-Shannon Sampling Theorem, states that a continuous signal can be completely reconstructed from its samples if the sampling frequency is greater than twice the highest frequency present in the signal. This critical frequency is known as the Nyquist rate.

-

$$\textbf{fs} > 2 \cdot \textbf{fmax}$$

Where:

- fs is the sampling frequency (rate at which the signal is sampled),
- fmax is the highest frequency present in the signal.

### Applications:

- Digital audio and video processing
- Communication systems
- Image processing
- Medical imaging

### Program:

```
clc;  
  
clear all;  
  
close all;  
  
subplot(2,2,1);  
t = 0:0.01:1;  
f=10;
```

```
y = sin(2*pi*f*t);  
plot(t,y);  
grid(true);  
xlabel("Time");  
ylabel("Amplitude");  
title("Continuous Signal");
```

```
subplot(2,2,2);  
fs= 0.5*f; Undersampled
```

```
t1 = 0:1/fs:1;  
y1 = sin(2*pi*f*t1);  
stem(t1,y1);  
hold on;  
plot(t1,y1);  
grid(true);  
xlabel("Time");  
ylabel("Amplitude");  
title("Under Sampled Signal");
```

```
subplot(2,2,3);  
fs2= 3*f; Nyquist sampled
```

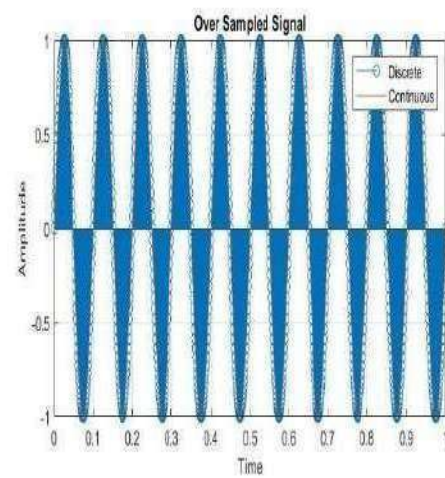
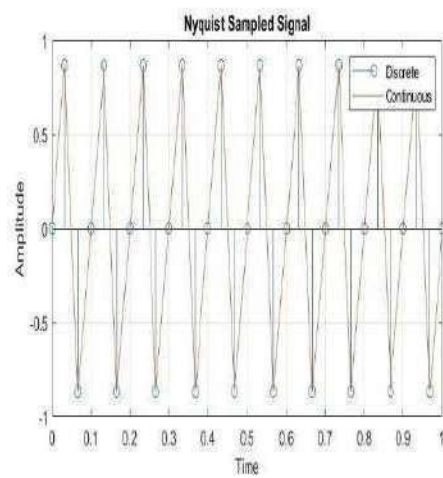
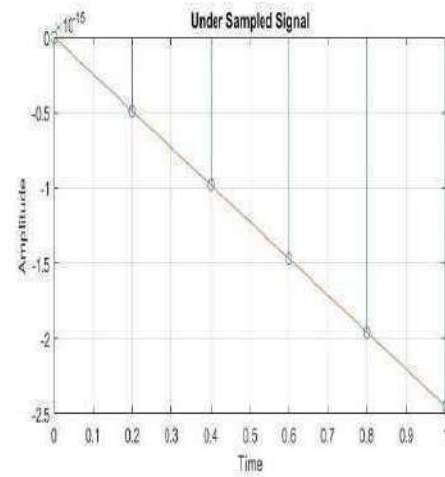
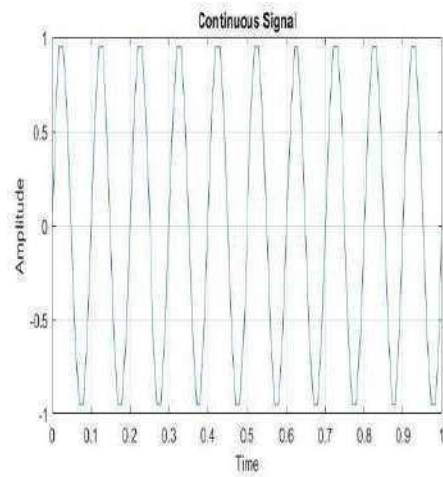
```
t3 = 0:1/fs2:1;  
y2 = sin(2*pi*f*t3);  
stem(t3,y2);  
hold on;  
plot(t3,y2);  
xgrid(true);  
xlabel("Time");  
ylabel("Amplitude");  
legend("Discrete","Continuous")
```

```
title("Nyquist Sampled Signal");  
subplot(2,2,4);  
  
fs2= 100*f; Oversampled  
t3 = 0:1/fs2:1;  
y2 = sin(2*pi*f*t3);  
stem(t3,y2);  
hold on;  
plot(t3,y2);  
grid(true);  
xlabel("Time");  
ylabel("Amplitude");  
legend("Discrete","Continuous")  
title("Over Sampled Signal");
```

### **Result**

Verified Sampling Theorem using MATLAB.

## Observation



**Experiment Name: Linear Convolution****Aim:**

To find linear convolution of following sequences with and without built in function.

- 1.  $x(n) = [1 \ 2 \ 1 \ 1]$   
 $h(n) = [1 \ 1 \ 1 \ 1]$
- 2.  $x(n) = [1 \ 2 \ 1 \ 2]$   
 $h(n) = [3 \ 2 \ 1 \ 2]$

**Theory:**

**Linear convolution** is a mathematical operation used to combine two signals to produce a third signal. It's a fundamental operation in signal processing and systems theory.

**Mathematical Definition:**

Given two signals,  $x(t)$  and  $h(t)$ , their linear convolution is defined as:

$$y(t) = x(t) * h(t) = \int_{-\infty}^{\infty} x(\tau)h(t - \tau) d\tau$$

**Applications:**

**Filtering:** Convolution is used to filter signals, removing unwanted frequencies or noise.

**System Analysis:** The impulse response of a system completely characterizes its behaviour, and convolution can be used to determine the output of the system given a known input.

**Image Processing:** Convolution is used for tasks like edge detection, blurring, and sharpening images.

**Program:****1. with built-in function:**

```
clc;

clear all;

close all;

x1 = input("Enter first Sequence");
h1 = input("Enter second Sequence");
y1 = conv(x1,h1);

disp("The convoluted sequence is: ");
```

```
disp(y1);

l = length(x1);
m = length(h1);
k = l+m-1;
n1 = 0:1:l-1;
n2 = 0:1:m-1;
n3 = 0:1:k-1;

subplot(1,3,1);
stem(n1,x1,"o");
xlabel("n");
ylabel("Amplitude");
title("x(n)");
grid on
xlim([-1 l+1]);
ylim([0 max(x1)+2]);

subplot(1,3,2);
stem(n2,h1,"o");
xlabel("n");
ylabel("Amplitude");
title("h(n)");
grid on
xlim([-1 m+1]);
ylim([0 max(h1)+2]);

subplot(1,3,3);
stem(n3,y1,"o");
xlabel("n");
ylabel("Amplitude");
title("y(n)");
grid on
xlim([-1 k+1]);
```

```
ylim([0 max(y1)+2]);
```

## **2. without built-in function:**

```
clc;
clear all;
close all;
x1 = input("Enter first Sequence");
h1 = input("Enter second Sequence");
l = length(x1);
m = length(h1);
k = l+m-1;
y1 = zeros(1,k);
for i=1:l
    for j=1:m
        y1(i+j-1) = y1(i+j-1) + x1(i)*h1(j);
    end
end
disp("The convoluted sequence is: ");
disp(y1);

n1 = 0:1:l-1;
n2 = 0:1:m-1;
n3 = 0:1:k-1;
subplot(1,3,1);
stem(n1,x1,"o");
xlabel("n");
ylabel("Amplitude");
title("x(n)");
grid on
xlim([-1 l+1]);
```



```
ylim([0 max(x1)+2]);
```

```
subplot(1,3,2);
```

```
stem(n2,h1,"o");
```

```
xlabel("n");
```

```
ylabel("Amplitude");
```

```
title("h(n)");
```

```
grid on
```

```
xlim([-1 m+1]);
```

```
ylim([0 max(h1)+2]);
```

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

```
stem(n3,y1,"o");
```

```
xlabel("n");
```

```
ylabel("Amplitude");
```

```
title("y(n)");
```

```
grid on
```

```
xlim([-1 k+1]);
```

```
ylim([0 max(y1)+2]);
```

## **Result**

Performed Linear Convolution using with and without built-in function.

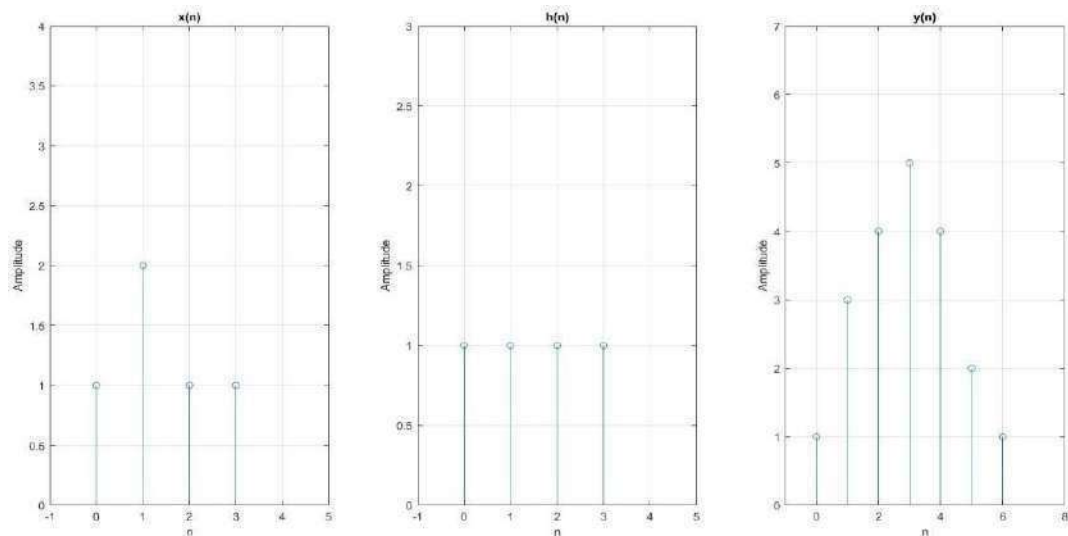
## **Observation**

a) Enter first Sequence [ 1 2 1 1]

Enter second Sequence [1 1 1 1]

The convoluted sequence is:

1 3 4 5 4 2 1

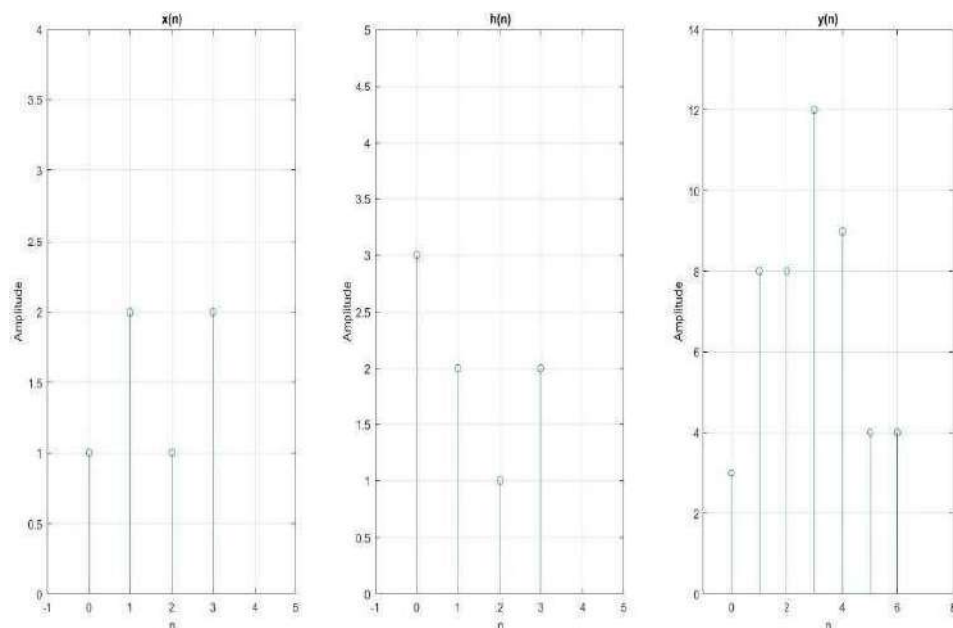


b) Enter first Sequence [1 2 1 2]

Enter second Sequence [3 2 1 2]

The convoluted sequence is:

3 8 8 12 9 4 4



**Experiment Name: Circular Convolution****Aim:**

To find circular convolution

- a. Using FFT and IFFT.
- b. Using Concentric Circle Method.
- c. Using Matrix Method.

**Theory:**

Circular convolution is a mathematical operation that is like linear convolution but is performed in a periodic or circular manner. This is particularly useful in discrete-time signal processing where signals are often represented as periodic sequences.

**Mathematical Definition:**

Given two periodic sequences  $x[n]$  and  $h[n]$ , their circular convolution is defined as:

$$y[n] = (x[n] \circledast h[n]) = \sum_{k=0}^{N-1} x[k]h[(n-k) \bmod N]$$

**Applications:**

- Discrete-Time Filtering: Circular convolution is used for filtering discrete-time signals.
- Digital Signal Processing: It's a fundamental operation in many digital signal processing algorithms.
- Cyclic Convolution: In certain applications, such as cyclic prefix OFDM, circular convolution is used to simplify the implementation of linear convolution.

**Program:****a. Using FFT and IFFT.**

```
clc;
```

```
close all; c
```

```
lear all;
```

```
x1 = [1 2 1 2];
```

```
x2 = [1 2 3 4];
```

```
X1_k = fft(x1);
```

```
X2_k = fft(x2);
```

```
Y1_k = X1_k.*X2_k;  
y1 =ifft(Y1_k);  
disp("Using FFT and IFFT:")  
disp(y1);
```

#### **b. Using Concentric Circle Method.**

```
clc;  
close all;  
clear all;  
  
x = [1 2 1 2];  
h = [1 2 3 4];  
N = max(length(x),length(h));  
y = zeros(1,N);  
for n=1:N  
    h_s = circshift(h,n-1); %shifting h(n) by 1 unit  
  
    y(n) = sum(x.*h_s);  
end  
  
disp("Using Concentric Circle Method:");  
disp(y);
```

#### **c. Using Matrix Method.**

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

```
x = [1 2 1 2];  
h = [1 2 3 4];  
N = max(length(x),length(h));  
h_n = zeros(N,N);  
for n=1:N h_s = circshift(h,n-1); %shifting h(n) by 1 unit  
h_n(:,n) = h_s;  
end  
y = h_n *x';  
disp("Using Concentric Circle Method:")  
disp(y');
```

### **Result**

Performed Circular Convolution using a) FFT and IFFT; b) Concentric Circle method; c) Matrix method and verified result.

## **Observation**

### **a) USING FFT AND IFFT**

*Using FFT and IFFT:*

16 14 16 14

### **b) USING Concentric Circle Method**

Using Concentric Circle Method:

16 14 16 14

### **c) USING Matrix Method**

Using Matrix Method.:

16 14 16 14

**Experiment Name: Linear Convolution using Circular Convolution and Vice versa.**

**Aim:**

1. To perform Linear Convolution using Circular Convolution.
2. To perform Circular Convolution using Linear Convolution.

**Theory:**

**Performing Linear Convolution Using Circular Convolution**

**Method:**

**1. Zero-Padding:**

- Pad both sequences  $x[n]$  and  $h[n]$  with zeros to a length of at least  $2N-1$ , where  $N$  is the maximum length of the two sequences. This ensures that the circular convolution will not wrap around and introduce artificial periodicity.

**2. Circular Convolution:**

- Perform circular convolution on the zero-padded sequences.

**3. Truncation:**

- Truncate the result of the circular convolution to the length  $N1 + N2 - 1$ , where  $N1$  and  $N2$  are the lengths of the original sequences  $x[n]$  and  $h[n]$ , respectively.

**Example:**

Consider the sequences  $x[n] = [1, 2, 3]$  and  $h[n] = [4, 5]$ .

**1. Zero-padding:**

- Pad  $x[n]$  to  $[1, 2, 3, 0, 0]$  and  $h[n]$  to  $[4, 5, 0, 0]$ .

**2. Circular Convolution:**

- Perform circular convolution on the zero-padded sequences. The result will be  $[4, 13, 21, 15, 0]$ .

**3. Truncation:**

- Truncate the result to  $[4, 13, 21, 15]$ .

This result is the same as the linear convolution of  $x[n]$  and  $h[n]$ .

## **Performing Circular Convolution Using Linear Convolution**

### **Method:**

#### **1. Zero-Padding:**

- Pad both sequences  $x[n]$  and  $h[n]$  to a length of at least  $2N-1$ , where  $N$  is the maximum length of the two sequences.

#### **2. Linear Convolution:**

- Perform linear convolution on the zero-padded sequences.

#### **3. Modulus Operation:**

- Apply the modulus operation to the indices of the linear convolution result, using the period  $N$ . This effectively wraps around the ends of the sequence, making it circular.

### **Example:**

Using the same sequences as before,  $x[n] = [1, 2, 3]$  and  $h[n] = [4, 5]$ .

#### **1. Zero-padding:**

- Pad  $x[n]$  to  $[1, 2, 3, 0, 0]$  and  $h[n]$  to  $[4, 5, 0, 0]$ .

#### **2. Linear Convolution:**

- Perform linear convolution. The result will be  $[4, 13, 21, 15, 0]$ .

#### **3. Modulus Operation:**

- Apply the modulus operation to the indices:  $[4, 13, 21, 15, 0]$  becomes  $[4, 13, 2, 15, 0]$ .

### **Program:**

#### **1. Linear Convolution using Circular Convolution**

```
clc;

clear all;

close all;

x = [1 2 3 4];
h = [1 1 1 ];
l = length(x);
m = length(h);
k = l+m-1;

x = [x zeros(1,k-l)];
h = [h zeros(1,k-m)];
```



```

X_k = fft(x);
H_k = fft(h);
Y_k = X_k.*H_k;
y = ifft(Y_k);
disp("Linear Convolution using Circular Convolution :");
disp(y);

```

## **2.Circular convolution using Linear Convolution**

```

clc;

close all;

clear all;

x = [1 2 3 4];
h = [1 1 1 ];
l = length(x);
m = length(h);
Lc = max(l,m);
Ll= l+m-1;
y = conv(x,h);
for i=1:Ll-Lc
    y(i) = y(i) + y(Lc+i);
end
for i=1:Lc
y1(i) = y(i);
end
disp("Circular convolution using Linear Convolution:")
disp(y1);

```

## **Result**

Performed

a) Linear Convolution using Circular Convolution;

b) Circular Convolution using Linear

Convolution and verified result.

## **Observation**

### **1) Linear Convolution using Circular Convolution :**

Linear Convolution using Circular Convolution:

1   3   6   9   7   4

### **2.Circular convolution using Linear Convolution**

Circular convolution using Linear Convolution:

8   7   6   9

## **DFT AND IDFT**

### **Aim:**

1. DFT using inbuilt function and without using inbuilt function. Also plot magnitude and phase plot of DFT
2. IDFT using inbuilt function and without using inbuilt function.

### **Theory:**

#### **Discrete Fourier Transform (DFT)**

The **Discrete Fourier Transform (DFT)** is a mathematical transformation used to analyze the frequency content of discrete signals. For a sequence  $x[n]$  of length  $N$ , the DFT is defined as:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j\frac{2\pi}{N}nk}, \quad k = 0, 1, 2, \dots, N-1$$

- $X[k]$  is the DFT of the sequence  $x[n]$ .
- The exponential factor represents  $e^{-j\frac{2\pi}{N}nk}$  the complex sinusoidal basis functions.
- The DFT maps the time-domain signal into the frequency domain.

#### **Inverse Discrete Fourier Transform (IDFT) Method:**

The **Inverse Discrete Fourier Transform (IDFT)** is used to convert a frequency-domain sequence  $X[k]$  back into its time-domain sequence  $x[n]$ . The IDFT is defined as:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] \cdot e^{j\frac{2\pi}{N}nk}, \quad n = 0, 1, 2, \dots, N-1$$

- The IDFT takes the frequency components  $X[k]$  and reconstructs the original sequence  $x[n]$ .
- The exponential factor  $e^{j\frac{2\pi}{N}nk}$  is the inverse of the DFT's complex sinusoidal basis functions.

### **Application**

- Spectrum (Analysis)
- Filtering
- Compression
- Modulation
- Convolution
- Demodulation

- Equalization
- Restoration
- Detection
- Estimation

### **Program:**

#### **1. Discrete Fourier Transform (DFT)**

```

clc;
clear all;
close all;
x=input("enter sequence:");
N=input("enter the N point:");
l=length(x);
x=[x zeros(1,N-1)];
X=zeros(1,N);
for k=0:N-1
    for n=0:N-1
        X(k+1)=X(k+1)+x(n+1)*exp(-1j*2*pi*n*k/N);
    end
end
disp('X');
disp(X);
disp('round(X)');
disp(round(X));
%verification
disp('fft');
disp(fft(x));

k=0:N-1;
magX=abs(X);

```

```

phaseX=angle(X);
subplot(2,1,1);
stem(k,magX);
title("Magnitude Plot");
hold on;
plot(k,magX);
subplot(2,1,2);
stem(k,phaseX);
hold on;
title("Phase Plot");
plot(k,phaseX);

```

## 2.IDFT

```

clc;
clear all;
close all;
X=input("enter sequence:");
N=input("enter the n point:");
l=length(X);
X=[X zeros(1,N-l)];
x=zeros(N,1);
for k=0:N-1
    for n=0:N-1
        x(n+1)=x(n+1)+X(k+1)*exp(1j*2*pi*n*k/N);
    end
end
x=1/N.*x;
disp('x');
disp(x);
disp('round(x)');

```

```
disp(round(x));  
disp('ifft');  
disp(ifft(X));
```

### **Result:**

Performed

- 1) DFT using inbuilt function and without using inbuilt function. Also plotted magnitude and phase plot of DFT.
  - 2) IDFT using inbuilt function and without using inbuilt function.
- and verified the result.

## **Observation:**

### **1.DFT**

enter sequence:[1 1 1 0]

enter the N point:8

X

Columns 1 through 7

$3.0000 + 0.0000i$   $1.7071 - 1.7071i$   $0.0000 - 1.0000i$   $0.2929 + 0.2929i$   $1.0000 + 0.0000i$   
 $0.2929 - 0.2929i$   $-0.0000 + 1.0000i$

Column 8

$1.7071 + 1.7071i$

round(X)

Columns 1 through 7

$3.0000 + 0.0000i$   $2.0000 - 2.0000i$   $0.0000 - 1.0000i$   $0.0000 + 0.0000i$   $1.0000 + 0.0000i$   
 $0.0000 + 0.0000i$   $0.0000 + 1.0000i$

Column 8

$2.0000 + 2.0000i$

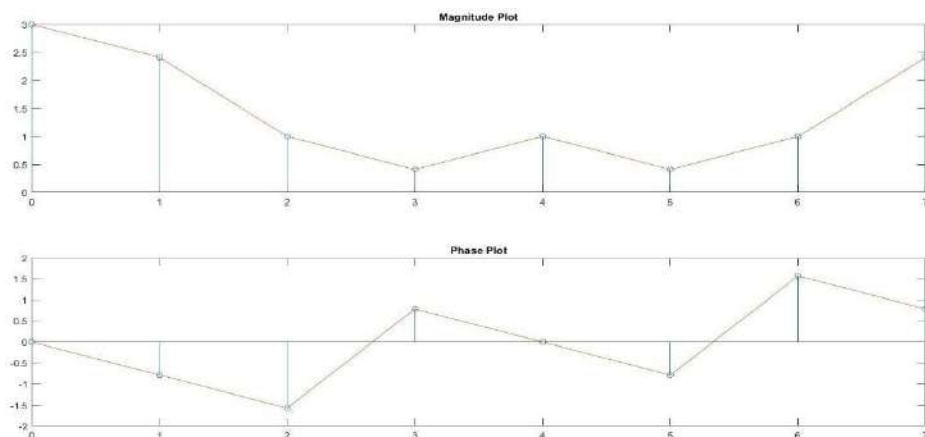
fft

Columns 1 through 7

$3.0000 + 0.0000i$   $1.7071 - 1.7071i$   $0.0000 - 1.0000i$   $0.2929 + 0.2929i$   $1.0000 + 0.0000i$   
 $0.2929 - 0.2929i$   $0.0000 + 1.0000i$

Column 8

$1.7071 + 1.7071i$





## 2.IDFT

enter sequence: [3.0000 + 0.0000i 1.7071 - 1.7071i 0.0000 - 1.0000i 0.2929 + 0.2929i  
1.0000 + 0.0000i 0.2929 - 0.2929i 0.0000 + 1.0000i]

enter the n point:8

x

0.7866 - 0.2134i  
0.6982 - 0.0000i  
0.7866 + 0.2134i  
-0.0000 + 0.3018i  
0.2134 + 0.2134i  
0.3018 - 0.0000i  
0.2134 - 0.2134i  
0.0000 - 0.3018i

round(x)

1  
1  
1  
0  
0  
0  
0  
0

ifft

Columns 1 through 7

0.7866 - 0.2134i 0.6982 + 0.0000i 0.7866 + 0.2134i 0.0000 + 0.3018i 0.2134 +  
0.2134i 0.3018 + 0.0000i 0.2134 - 0.2134i

Column 8

0.0000 - 0.3018i

## **Properties of DFT**

### **Aim:**

Verify following properties of DFT using Matlab/Scilab.

1. Linearity Property
2. Parseval's Theorem
3. Convolution Property
4. Multiplication Property

### **Theory:**

#### **1. Linearity Property**

The linearity property of the DFT states that if you have two sequences  $x_1[n]$  and  $x_2[n]$ , and their corresponding DFTs are  $X_1[k]$  and  $X_2[k]$ , then for any scalar  $a$  and  $b$ :

$$\text{DFT}\{a \cdot x_1[n] + b \cdot x_2[n]\} = a \cdot \text{DFT}\{x_1[n]\} + b \cdot \text{DFT}\{x_2[n]\}$$

#### **2. Parseval's Theorem**

Parseval's theorem states that the total energy of a signal in the time domain is equal to the total energy in the frequency domain. For a sequence  $x[n]$  and its DFT  $X[k]$ :

$$\sum_{n=0}^{N-1} |x[n]|^2 = \frac{1}{N} \sum_{k=0}^{N-1} |X[k]|^2$$

#### **3. Convolution Property**

The convolution property of the DFT states that the circular convolution of two sequences in the time domain is equivalent to the element-wise multiplication of their DFTs in the frequency domain:

$$\text{DFT}\{x_1[n] \otimes x_2[n]\} = \text{DFT}\{x_1[n]\} \cdot \text{DFT}\{x_2[n]\}$$

#### **4. Multiplication Property**

The multiplication property of DFT states that pointwise multiplication in the time domain corresponds to circular convolution in the frequency domain:

$$\text{DFT}\{x_1[n] \cdot x_2[n]\} = \frac{1}{N} \text{DFT}\{x_1[n]\} \otimes \text{DFT}\{x_2[n]\}$$

### **Program:**

#### **1. Linearity Property**

```
clc;
```

```

clear all;
close all;
x=input("enter first sequence");
h=input("enter sequence sequence:");
lx=length(x);
lh=length(h);
if lx>lh
    h=[h zeros(1,lx-lh)]
else
    x=[x zeros(1,lh-lx)]
end
a=input("enter value of 'a':");
b=input("enter value of 'b':");
lhs=fft((a.*x)+(b.*h));
rhs=a.*fft(x)+b.*fft(h);
disp('LHS');
disp(lhs);
disp('RHS');
disp(rhs);
if lhs==rhs
    disp('Linearity property verified');
else
    disp('Linearity property not verified');
end

```

## 2. Parseval's Theorem

```

clc;
clear all;
close all;
x=input("enter first sequence:");

```

```

h=input("enter second sequence:");
N=max(length(x),length(h));
xn=[x zeros(1,N-length(x))];
hn=[h zeros(1,N-length(h))];
lhs=sum(xn.*conj(hn));
rhs=sum(fft(xn).*conj(fft(hn)))/N;
disp('LHS');
disp(lhs);
disp('RHS');
disp(rhs);
if lhs==rhs
    disp("Parseval's Theorem verified");
else
    disp("Parseval's Theorem not verified");
end

```

### **3.Convolution Property**

```

clc;
clear all;
close all;
x=input("enter first sequence");
h=input("enter sequence sequence:");
N=max(length(x), length(h));
xn=[x zeros(N-length(x))];
hn=[h zeros(N-length(h))];
Xn=fft(xn);
Hn=fft(hn);
lhs=cconv(xn,hn,N);
rhs=ifft(Xn.*Hn);
disp('LHS');

```

```

disp(lhs);
disp('RHS');
disp(rhs);
if lhs==rhs
    disp('Circular Convolution verified')
else
    disp('Circular Convolution not verified');
end

```

#### **4. Multiplication Property**

```

clc;
clear all;
close all;
x=input("enter first sequence");
h=input("enter sequence sequence:");
N=max(length(x), length(h));
xn=[x zeros(N-length(x))];
hn=[h zeros(N-length(h))];
lhs=fft(xn.*hn);
Xn=fft(xn);
Hn=fft(hn);
rhs=(cconv(Xn,Hn,N))/N;
disp('LHS');
disp(lhs);
disp('RHS');
disp(rhs);
if lhs==rhs
    disp('Multiplication property verified');
else
    disp('Multiplication property not verified');
end

```

**Result:**

Performed and verified the following properties of DFT:

- 1.Linear Property
- 2.Parseval's Theorem
- 3.Convolution Property
- 4.Multiplication Property

## **Observation:**

### **1. Linearity Property**

enter first sequence[1 2 3 4]

enter sequence sequence:[1 1 1 1]

x =

1    2    3    4

enter value of 'a':2

enter value of 'b':3

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

### **2. Parseval's Theorem**

enter first sequence:[1 2 3 4]

enter second sequence:[1 1 1 1]

LHS

$$10$$

RHS

$$10$$

Parseval's Theorem verified

### **3.Convolution Property**

enter first sequence[1 2 3 4]

enter sequence sequence:[1 1 1 1]

LHS

$$10 \quad 10 \quad 10 \quad 10$$

RHS

$$10 \quad 10 \quad 10 \quad 10$$

Circular Convolution verified

#### **4.Multiplication Property**

enter first sequence[1 2 3 4]

enter sequence sequence:[1 1 1 1]

LHS

Columns 1 through 3

$$10.0000 + 0.0000i -2.0000 + 2.0000i -2.0000 + 0.0000i$$

Column 4

$$-2.0000 - 2.0000i$$

RHS

Columns 1 through 3

$$10.0000 + 0.0000i -2.0000 + 2.0000i -2.0000 + 0.0000i$$

Column 4

$$-2.0000 - 2.0000i$$

Multiplication property verified



## **OVERLAP ADD AND OVERLAP SAVE METHOD**

### **Aim:**

Implement overlap add and overlap save method using Matlab/Scilab.

### **Theory:**

Both the Overlap-Save and Overlap-Add methods are techniques used to compute the convolution of long signals using the Fast Fourier Transform (FFT). The direct convolution of two signals, especially when they are long, can be computationally expensive. These methods allow us to break the signals into smaller blocks and use the FFT to perform the convolution more efficiently.

#### **Overlap-Save Method**

The Overlap-Save method deals with circular convolution by discarding the parts of the signal that are corrupted by wrap-around effects. Here's how it works:

1. **Block Decomposition:** The input signal is divided into overlapping blocks. If the filter has length  $M$  and we use blocks of length  $N$ , the overlap is  $M$  samples, so each block has  $N - M + 1$  new samples and  $M$  samples from the previous block.
2. **FFT and Convolution:** Each block is convolved with the filter using FFT. However, because of circular convolution, the result contains artifacts due to the overlap.
3. **Discard and Save:** We discard the first  $M$  samples from each block (the part affected by the wrap-around) and save the remaining samples. This gives us the correct linear convolution.

#### **Overlap-Add Method**

The Overlap-Add method, on the other hand, handles circular convolution by adding overlapping sections of the convolved blocks. Here's how it works:

1. **Block Decomposition:** The input signal is split into non-overlapping blocks of size  $N$ . Each block is then zero-padded to a size of  $2N - M$ , where  $M$  is the length of the filter.
2. **FFT and Convolution:** Each block is convolved with the filter using FFT. Since the blocks are zero-padded, the convolution produces valid linear results, but the output blocks overlap.
3. **Overlap and Add:** After convolution, the results of each block overlap by  $M$  samples. These overlapping regions are added together to form the final output.

## **Program:**

### **1. Overlap Add**

```
clc;
clear all;
close all;

x = input('Enter the input sequence x : ');
h = input('Enter the impulse response h : ');
L = length(h); % Length of impulse response
N = length(x);
M = length(h);
x_padded = [x, zeros(1, L - 1)];
y = zeros(1, N + M + 1);

num_sections = (N + L - 1) / L; % Calculate number of sections
for n = 0:num_sections-1
    start_idx = n * L + 1;
    end_idx = start_idx + L - 1;
    x_section = x_padded(start_idx:min(end_idx, end));

    conv_result = conv(x_section, h);
```

```

        y(start_idx:start_idx + length(conv_result) - 1)
    =y(start_idx:start_idx + length(conv_result) - 1) + conv_result;
end
y = y(1:N + M - 1);
y_builtin = conv(x, h);
% Display results
disp('Overlap-add convolution result:');
disp(y);
disp('Built-in convolution result:');
disp(y_builtin);
figure;
subplot(2, 1, 1);
stem(y, 'filled');
title('Overlap-add Convolution Result');
grid on;
subplot(2, 1, 2);
stem(y_builtin, 'filled');
title('Built-in Convolution Result');
grid on;

```

## 2.Overlap Save

```

clc;
clear all;
close all;
x = input("Enter 1st sequence: ");
h = input("Enter 2nd sequence: ");

```

```

N = input("Fragmented block size: ");
y = overlav(x, h, N);
disp("Using Overlap and Save method");
disp(y);
disp("Verification");
disp(cconv(x,h,length(x)+length(h)-1));
function y = overlav(x, h, N)
    if (N < length(h))
        error("N must be greater than the length of h");
    end
    Nx = length(x); % Length of input sequence x
    M = length(h); % Length of filter sequence h
    M1 = M - 1; % Length of overlap
    L = N - M1; % Length of non-overlapping part
    x = [zeros(1, M1), x, zeros(1, N-1)];
    h = [h, zeros(1, N - M)];
    K = floor((Nx + M1 - 1) / L);
    Y = zeros(K + 1, N);

    for k = 0:K
        xk = x(k*L + 1 : k*L + N);
        Y(k+1, :) = cconv(xk, h, N);
    end
end

```

```
Y = Y(:, M:N)';  
y = (Y(:))';  
end
```

**Result:**

Performed Overlap Save and Overlap Add method and verified the result.

## **Observation:**

### **1. Overlap Add**

Enter the input sequence x : [3 -1 0 1 3 2 0 1 2 1]

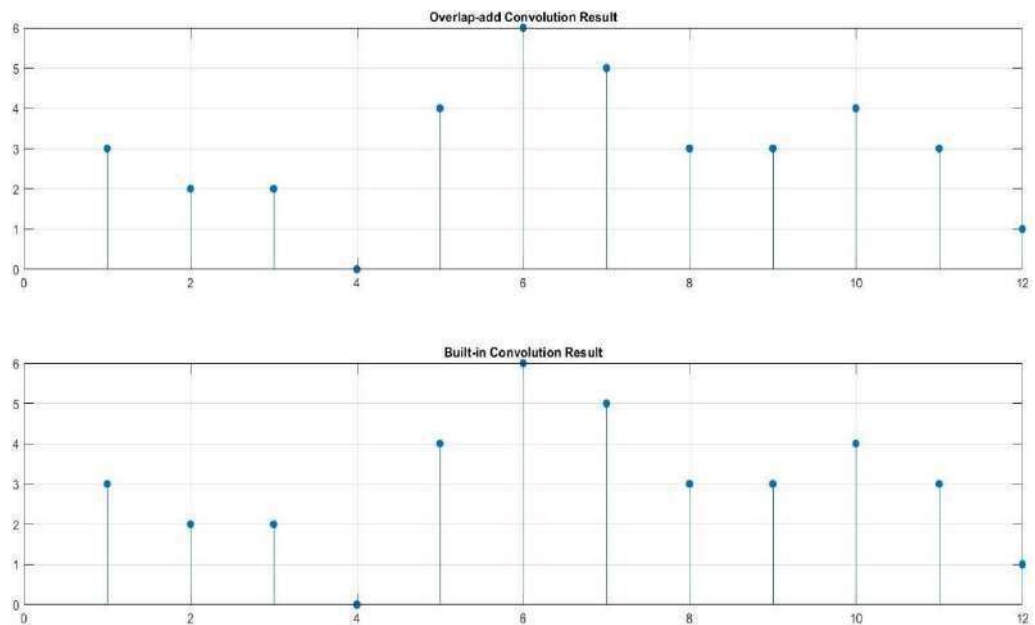
Enter the impulse response h : [1 1 1]

Overlap-add convolution result:

3 2 2 0 4 6 5 3 3 4 3 1

Built-in convolution result:

3 2 2 0 4 6 5 3 3 4 3 1



### **2.Overlap Save**

Enter 1st sequence: [3 -1 0 1 3 2 0 1 2 1]

Enter 2nd sequence: [1 1 1]

Fragmented block size: 3

Using Overlap and Save method

3 2 2 0 4 6 5 3 3 4 3 1

Verification

3.0000 2.0000 2.0000 0 4.0000 6.0000 5.0000 3.0000 3.0000 4.0000  
3.0000 1.0000

## Implementation of FIR Filters

**Aim:** Write a MATLAB program to implement the following FIR filters using Hanning, Hamming, Rectangular and Triangular windows.

- a) Low Pass Filter
- b) High Pass Filter
- c) Band Pass Filter
- d) Band Stop Filter

### Theory

Finite Impulse Response (FIR) filters are a type of digital filter characterized by a finite duration of the impulse response. The window method is a common technique used to design FIR filters. This method involves multiplying an ideal (infinite) impulse response by a window function to create a realizable FIR filter. The FIR filter coefficients  $h[n]$  are obtained by multiplying the ideal impulse response by the chosen window function. The frequency response of the FIR filter can be analyzed using the Discrete Fourier Transform (DFT). The window function influences the main lobe width and side lobe levels in the frequency response. A narrower main lobe provides better frequency resolution, while lower side lobes reduce spectral leakage.

Windows:

1. Rectangular window:

$$w_{rec}(n) = 1, \quad -M \leq n \leq M.$$

2. Triangular (Bartlett) window:

$$w_{tri}(n) = 1 - \frac{|n|}{M}, \quad -M \leq n \leq M.$$

3. Hanning window:

$$w_{han}(n) = 0.5 + 0.5 \cos\left(\frac{n\pi}{M}\right), \quad -M \leq n \leq M.$$

4. Hamming window:

$$w_{ham}(n) = 0.54 + 0.46 \cos\left(\frac{n\pi}{M}\right), \quad -M \leq n \leq M.$$

Filters:

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

## Program

a) Low Pass Filter

```

wc = 0.5*pi;

N=50;

alpha = (N-1)/2;

n=0:1:N-1;

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

%LPFhamming

w1=hamming(N);

hn=hd.*w1';

w=0:0.01:pi;

h1=freqz(hn,1,w);

subplot(4,2,1);

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

title('LPF using hamming window');

xlabel('normalized frequency');

ylabel('magnitude in db');

%LPFhanning

w2=hanning(N);

```



```

hn=hd.*w2';
w=0:0.01:pi;
h2=freqz(hn,1,w);
subplot(4,2,3);
plot(w/pi,10*log10(abs(h2)));
title('LPF using hanning window');
xlabel('normalized frequency');
ylabel('magnitude in db');

%LPFrect
w3=boxcar(N);
hn=hd.*w3';
w=0:0.01:pi;
h3=freqz(hn,1,w);
subplot(4,2,5);
plot(w/pi,10*log10(abs(h3)));
title('LPF using rectangular window');
xlabel('normalized frequency');
ylabel('magnitude in db');

%LPFtri
w4=bartlett(N);
hn=hd.*w4';
w=0:0.01:pi;
h4=freqz(hn,1,w);
subplot(4,2,7);
plot(w/pi,10*log10(abs(h4)));
title('LPF using triangular window');

```

```
xlabel('normalized frequency');  
ylabel('magnitude in db');  
  
%hamming  
subplot(4,2,2);  
stem(w1);  
title('hamming window sequence');  
xlabel('no of samples');  
ylabel('amplitude');  
  
%hanning  
subplot(4,2,4);  
stem(w2);  
title('hanning window sequence');  
xlabel('no of samples');  
ylabel('amplitude');  
  
%rectangular  
subplot(4,2,6);  
stem(w3);  
title('rectangular window sequence');  
xlabel('no of samples');  
ylabel('amplitude');  
  
%triangular  
subplot(4,2,8);  
stem(w4);  
title('tirangular window sequence');  
xlabel('no of samples');  
ylabel('amplitude');
```

b) High Pass Filter

```
clc;

clear all;

close all;

wc = 0.5*pi;

N=50;

alpha = (N-1)/2;

n=0:1:N-1;

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

%HPFhamming

w1=hamming(N);

hn=hd.*w1';

w=0:0.01:pi;

h1=freqz(hn,1,w);

subplot(4,2,1);

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

title('HPF using hamming window');

xlabel('normalized frequency');

ylabel('magnitude in db');

%HPFhanning

w2=hanning(N);

hn=hd.*w2';

w=0:0.01:pi;

h2=freqz(hn,1,w);

subplot(4,2,3);

plot(w/pi,10*log10(abs(h2)));
```

```

title('HPF using hanning window');
xlabel('normalized frequency');
ylabel('magnitude in db');

%HPFrect

w3=boxcar(N);
hn=hd.*w3';
w=0:0.01:pi;
h3=freqz(hn,1,w);
subplot(4,2,5);
plot(w/pi,10*log10(abs(h3)));
title('HPF using rectangular window');
xlabel('normalized frequency');
ylabel('magnitude in db');

%HPFtri

w4=bartlett(N);
hn=hd.*w4';
w=0:0.01:pi;
h4=freqz(hn,1,w);
subplot(4,2,7);
plot(w/pi,10*log10(abs(h4)));
title('HPF using triangular window');
xlabel('normalized frequency');
ylabel('magnitude in db');

%WINDOWS

%hamming
subplot(4,2,2);

```

```
stem(w1);  
title('hamming window sequence');  
xlabel('no of samples');  
ylabel('amplitude');  
  
%hanning  
subplot(4,2,4);  
stem(w2);  
title('hanning window sequence');  
xlabel('no of samples');  
ylabel('amplitude');  
  
%rectangular  
subplot(4,2,6);  
stem(w3);  
title('rectangular window sequence');  
xlabel('no of samples');  
ylabel('amplitude');  
  
%triangular  
subplot(4,2,8);  
stem(w4);  
title('tirangular window sequence');  
xlabel('no of samples');  
ylabel('amplitude');
```

c) Band Pass Filter

```
clc;

clear all;

close all;

wc1= 0.25*pi;

wc2 = 0.75*pi;

N=50;

%N = input('enter the value of N');

alpha = (N-1)/2;

n=0:1:N-1;

hd=(sin(wc1*(n-alpha+eps))-sin(wc2*(n-alpha+eps)))./(pi*(n-
alpha+eps));

%BPFFhamming

w1=hamming(N);

hn=hd.*w1';

w=0:0.01:pi;

h1=freqz(hn,1,w);

subplot(4,2,1);

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

title('BPF using hamming window');

xlabel('normalized frequency');

ylabel('magnitude in db');

%BPFFhanning

w2=hanning(N);

hn=hd.*w2';

w=0:0.01:pi;
```

```

h2=freqz(hn,1,w);
subplot(4,2,3);
plot(w/pi,10*log10(abs(h2)));
title('BPF using hanning window');
xlabel('normalized frequency');
ylabel('magnitude in db');

%BPFFrect
w3=boxcar(N);
hn=hd.*w3';
w=0:0.01:pi;
h3=freqz(hn,1,w);
subplot(4,2,5);
plot(w/pi,10*log10(abs(h3)));
title('BPF using rectangular window');
xlabel('normalized frequency');
ylabel('magnitude in db');

%BPFFtri
w4=bartlett(N);
hn=hd.*w4';
w=0:0.01:pi;
h4=freqz(hn,1,w);
subplot(4,2,7);
plot(w/pi,10*log10(abs(h4)));
title('BPF using triangular window');
xlabel('normalized frequency');
ylabel('magnitude in db');

```

```
%WINDOWS

%hamming

subplot(4,2,2);

stem(w1);

title('hamming window sequence');

xlabel('no of samples');

ylabel('amplitude');

%hanning

subplot(4,2,4);

stem(w2);

title('hanning window sequence');

xlabel('no of samples');

ylabel('amplitude');

%rectangular

subplot(4,2,6);

stem(w3);

title('rectangular window sequence');

xlabel('no of samples');

ylabel('amplitude');

%triangular

subplot(4,2,8);

stem(w4);

title('tirangular window sequence');

xlabel('no of samples');

ylabel('amplitude');
```



d) Band Stop Filter

```
wc1= 0.25*pi;
wc2 = 0.75*pi;
N=50;
%N = input('enter the value of N');
alpha = (N-1)/2;
n=0:1:N-1;
hd=(sin(wc1*(n-alpha+eps))-sin(wc2*(n-alpha+eps))+sin(pi*(n-
alpha+eps)))./(pi*(n-alpha+eps));
%BSFhamming
w1=hamming(N);
hn=hd.*w1';
w=0:0.01:pi;
h1=freqz(hn,1,w);
subplot(4,2,1);
plot(w/pi,10*log10(abs(h1)));
title('BSF using hamming window');
xlabel('normalized frequency');
ylabel('magnitude in db');
%BSFhanning
w2=hanning(N);
hn=hd.*w2';
w=0:0.01:pi;
h2=freqz(hn,1,w);
subplot(4,2,3);
plot(w/pi,10*log10(abs(h2)));
```

```

title('BSF using hanning window');
xlabel('normalized frequency');
ylabel('magnitude in db');

%BSFrect

w3=boxcar(N);
hn=hd.*w3';
w=0:0.01:pi;
h3=freqz(hn,1,w);
subplot(4,2,5);
plot(w/pi,10*log10(abs(h3)));
title('BSF using rectangular window');
xlabel('normalized frequency');
ylabel('magnitude in db');

%BSFtri

w4=bartlett(N);
hn=hd.*w4';
w=0:0.01:pi;
h4=freqz(hn,1,w);
subplot(4,2,7);
plot(w/pi,10*log10(abs(h4)));
title('BSF using triangular window');
xlabel('normalized frequency');
ylabel('magnitude in db');

%WINDOWS

%hamming
subplot(4,2,2);

```

```

stem(w1);
title('hamming window sequence');
xlabel('no of samples');
ylabel('amplitude');

%hanning
subplot(4,2,4);
stem(w2);
title('hanning window sequence');
xlabel('no of samples');
ylabel('amplitude');

%rectangular
subplot(4,2,6);
stem(w3);
title('rectangular window sequence');
xlabel('no of samples');
ylabel('amplitude');

%triangular
subplot(4,2,8);
stem(w4);
title('tirangular window sequence');
xlabel('no of samples');
ylabel('amplitude');

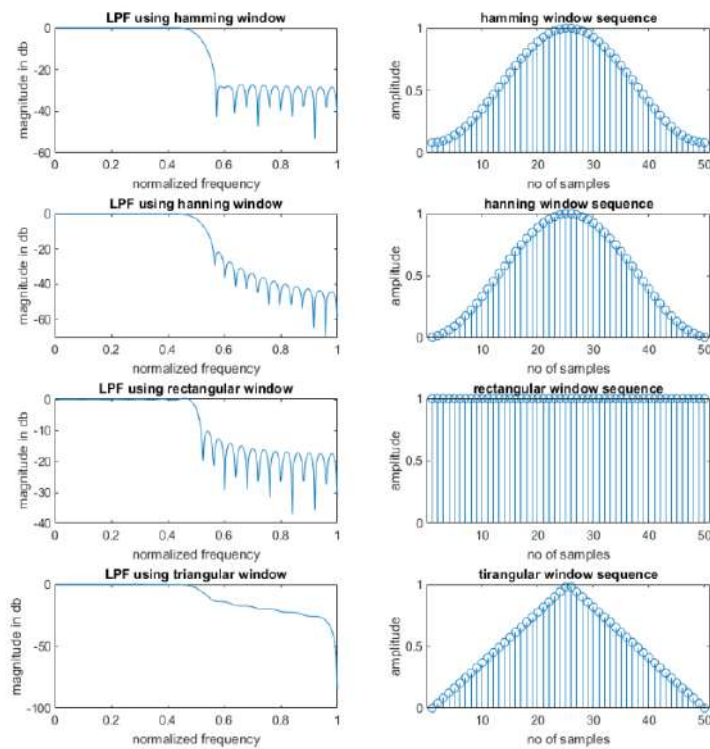
```

## Result

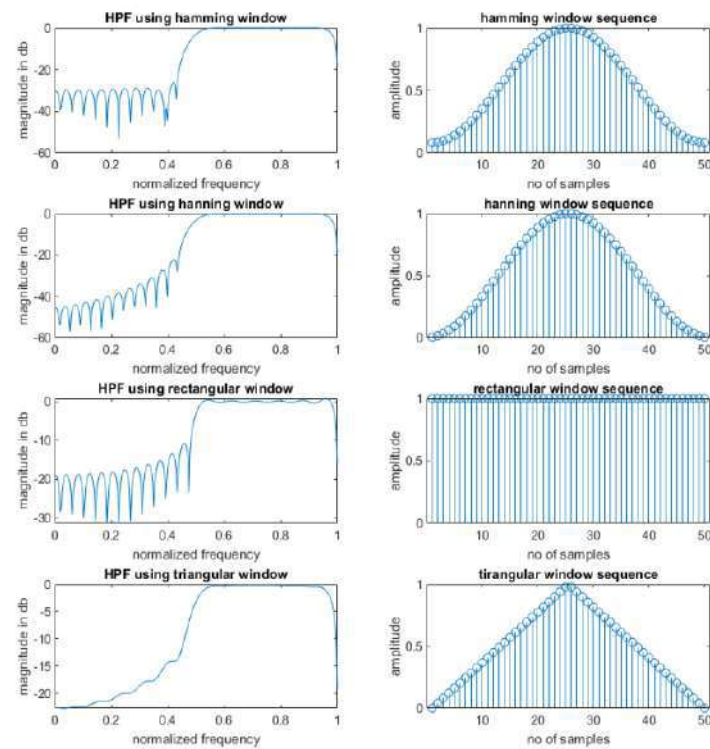
Implemented FIR filters using Window method.

## Observation

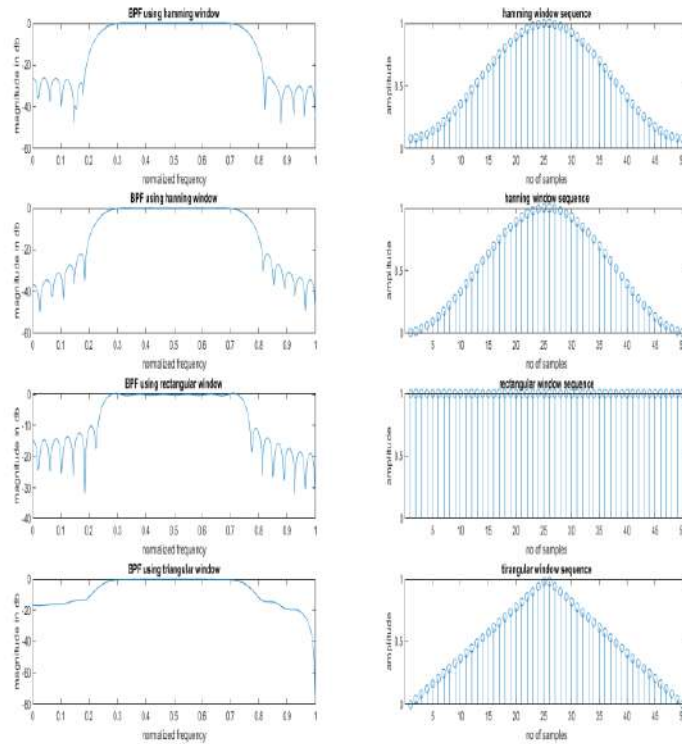
### a) Low Pass Filter



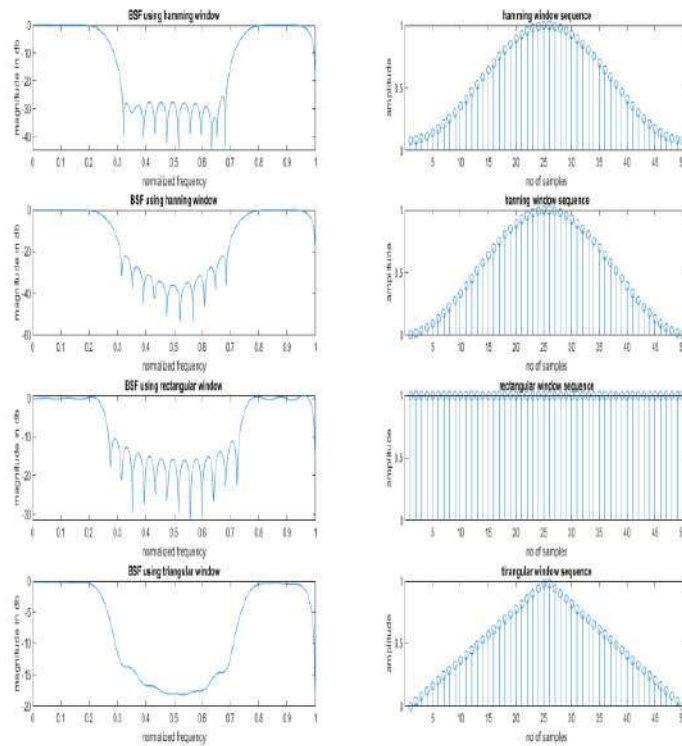
### b) High Pass Filter



### c) Band Pass Filter



### d) Band Stop Filter



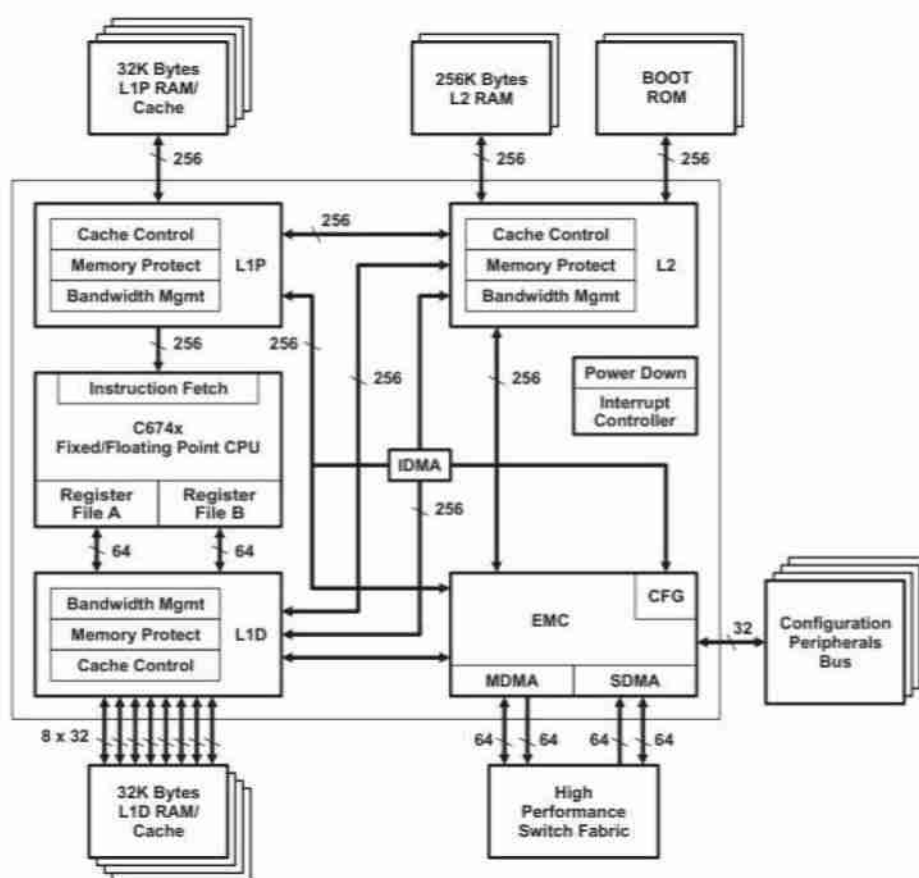
## FAMILIARIZATION OF THE ANALOG AND DIGITAL INPUT AND OUTPUT PORTS OF DSP BOARD

### AIM

To familiarize with the input and output ports of dsp board.

### 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):**

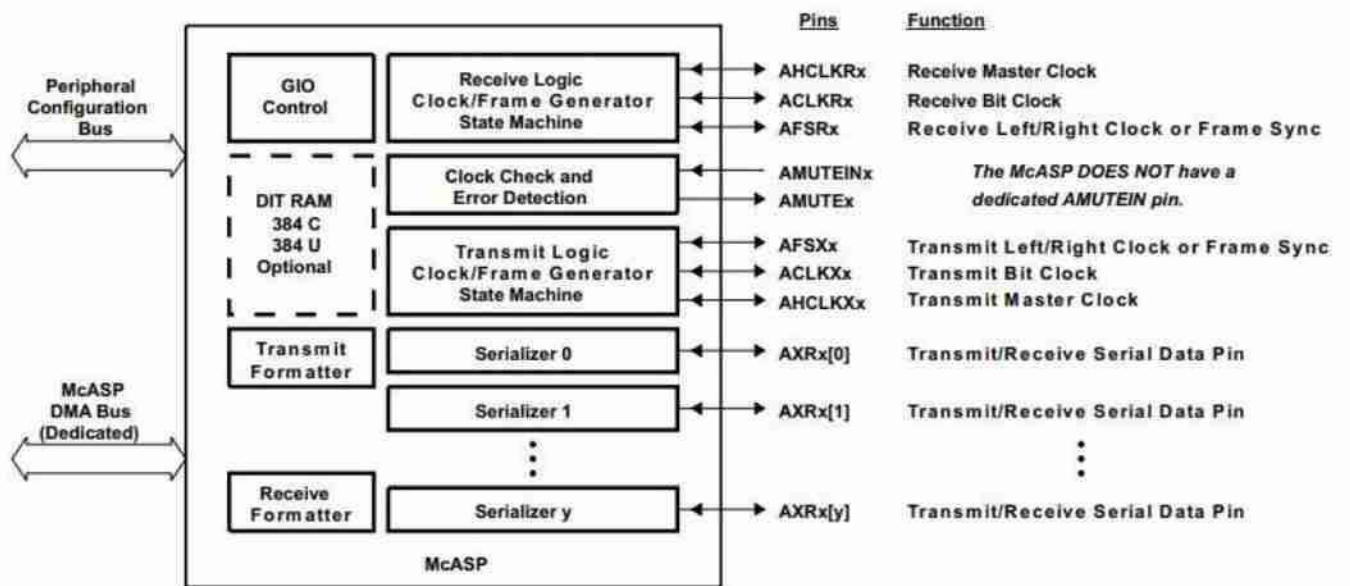
The McASP serial port is specifically designed for multichannel audio applications.

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 data words 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.



## 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  $0^\circ$  angle to a maximum value of 1 at  $90^\circ$ , it further reaches its minimum value of -1 at  $270^\circ$  and then return to 0 at  $360^\circ$ . After any angle greater than  $360^\circ$ , the sinusoidal signal repeats the values so we can say that period of sinusoidal signal is  $2\pi$  i.e.  $360^\circ$ . 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  $2\pi$  hence periodicity of sinusoidal signal is  $2\pi$ .

### 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 click Resume in CCS.

## **Program**

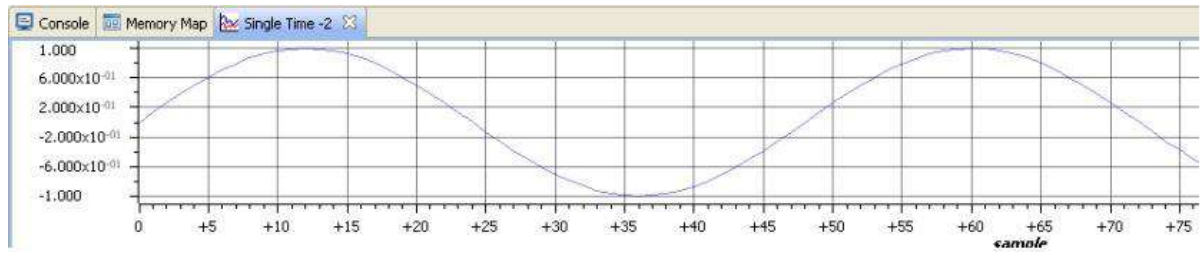
Sine wave

```
#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



## 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:

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

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  $X_n$  and filter  $H_n$  values to specified addresses:

$X_n$ : Start at 0x80010000, populate subsequent values at offsets like 0x80010004 for each additional input.

$H_n$ : Start at 0x80011000 with similar offsets for additional values.

Lengths of  $X_n$  and  $H_n$  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