THANGAL KUNJU MUSALIAR COLLEGE OF ENGINEERING

KOLLAM – 691 005

ELECTRONICS AND COMMUNICATION ENGINEERING

LABORATORY RECORD

YEAR 2024-25

Certified that this is a Bonafide Record of the work done by Sri ***MARIYA MANOJ P 5th*** *Semester class B22ECB71* ***Electronics and Communication*** *in the* ***Digital Signal Processing*** *Laboratory during the year* ***2024-25***

Name of the Examination: ***Fifth Semester B.Tech Degree Examination 2024***

Register Number : ***TKM22EC080***

Staff Member in-charge *External Examiner*

Date

INDEX

Experiment No:1 Date:29-7-24

SIMULATION OF BASIC TEST SIGNALS

Aim

Simulation of basic test signals using Matlab, Signals are:

1) Impulse signal
2) Unit step signal
3) Ramp signal
4) Sine signal
5) Cosine signal
6) Square Wave-Bipolar
7) Square Wave-Unipolar
8) Triangular signal
9) Exponential signal

Theory

1. Impulse signal

An **impulse signal** is an idealized, infinitesimally narrow pulse that occurs at a single instant in time, typically at t=0. It is represented mathematically by the Dirac delta function, denoted as

$$\delta(t) = \begin{cases} 0, & \text{if } t \neq 0 \\ \infty, & \text{if } t = 0 \end{cases}$$

2.**Unit step signal** is a function that jumps from 0 to 1 at a specified time, typically at t=0. The unit step function is denoted as

$$u(t) = \begin{cases} 0, & if\ t < 0 \\ 1, & if\ t \geq 0 \end{cases}$$

3. Ramp signal

A **ramp signal** is a signal that increases linearly with time, starting from zero. The ramp function is denoted as

$$r(t) = \begin{cases} t, & if\ t \geq 0 \\ 0, & if\ t < 0 \end{cases}$$

4. Sine signal

A **sine signal** is a continuous wave that oscillates smoothly and periodically over time, following the shape of a sine or cosine function. The general form of a sinusoidal signal is given by:

$$x(t) = Asin(\omega t + \phi)$$

Where,

- Ais the amplitude of the signal (the peak value),
- ω is the angular frequency in radians per second, where $\omega = 2\pi f$, f is the frequency in Hertz,
- t is the time variable
- ϕ is the phase shift, which determines the initial angle at t=0.

5.Cosine signal

A **cosine signal** is a type of sinusoidal signal that oscillates in a smooth, periodic manner over time, following the shape of a cosine function. The general form of a cosine signal is given by:

$$x(t) = Acos(\omega t + \phi)$$

Where:

- A is the amplitude of the signal (the peak value),
- ω is the angular frequency in radians per second, where $\omega = 2\pi f$ and f is the frequency in Hertz.
- t is the time variable
- ϕ is the phase shift, which determines the initial angle at $t = 0$.

6. Square Wave-Bipolar

A square wave is a type of periodic waveform that alternates between two distinct levels, typically +A and -A in a bipolar signal. It has a 50% duty cycle, meaning the signal spends equal time at both levels. The equation for a bipolar square wave can be written as:

$V(t)=A\cdot sgn(\sin(2\pi ft))$ where A is the amplitude, f is the frequency, and

sgn is the sign function.

7. Square Wave-Unipolar A unipolar square wave is a periodic signal that alternates between 0 and a positive voltage level (e.g., V_max) with abrupt transitions. It has no negative amplitude. The signal is typically represented as:

$f(t)=V_{max}$, for $0 \leq t < T/2$ $f(t)=0$, for $T/2 \leq t < T$ Where T

is the period of the waveform.

8. Triangular signal

A **triangular signal** is a type of periodic waveform that linearly rises and falls between a maximum and minimum value, forming a triangular shape. The transition between the high and low levels in a triangular wave is gradual, creating a linear slope.

9. Exponential signal

An **exponential signal** is a signal whose amplitude varies exponentially with time. It can either grow or decay depending on the sign of the exponent. The exponential signal is generally expressed as

$$x(t) = Ae^{\alpha t}$$

Where:

- A is the amplitude of the signal,
- α is the exponent that determines the rate of growth or decay,
- t is the time variable.
- If $\alpha > 0$, the signal represents exponential growth.
- If $\alpha < 0$, the signal represents exponential decay.

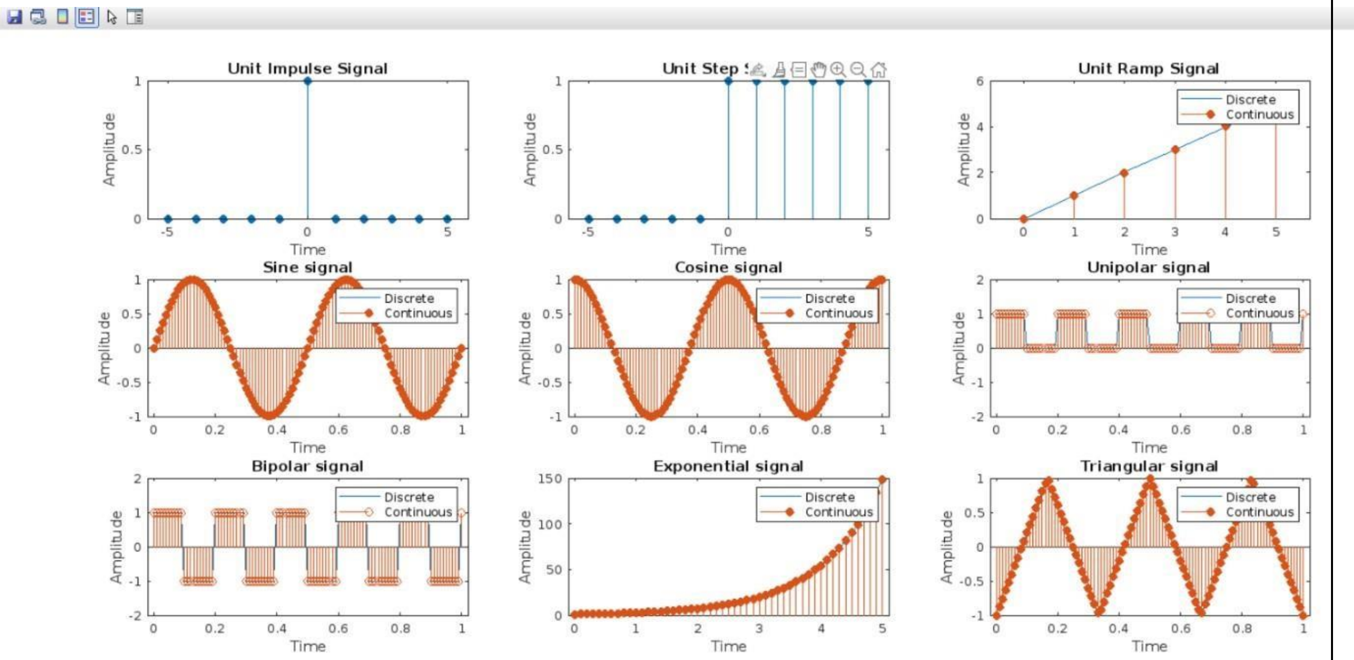
PROGRAM

```
%unit impulse signal clc; close all;
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 Impulse Signal"); %unit
step        signal        t2=-5:1:5;
y2=[zeros(1,5),ones(1,6)];
subplot(3,3,2);
stem(t2,y2,'filled');
xlabel("Time"); ylabel("Amplitude");
title("Unit Step Signal"); %unit ramp
signal       t3=0:1:5;        y3=t3;
subplot(3,3,3); plot(t3,y3); hold on;
stem(t3,y3,'filled');
xlabel("Time");
ylabel("Amplitude");
title(“Unit Ramp Signal");
legend("Discrete","Continuous")
;
%sine     t4=0:0.01:1;     f4=2;
y4=sin(2*pi*f4*t4);
subplot(3,3,4);    plot(t4,y4);
hold  on;  stem(t4,y4,'filled');
xlabel("Time");
ylabel("Amplitude");
```

```
title("Sine            signal");
legend("Discrete","Continuous")
;  %cosine  signal  t5=0:0.01:1;
f5=2;        y5=cos(2*pi*f5*t5);
subplot(3,3,5);     plot(t5,y5);
hold  on;  stem(t5,y5,'filled');
xlabel("Time");
ylabel("Amplitude");
title("Cosine          signal");
legend("Discrete","Continuous")
;
%unipolar signal
f6=5;               t6=0:0.01:1;
y6=sqrt(square(2*pi*f6*t6));
subplot(3,3,6);     plot(t6,y6);
hold      on;       stem(t6,y6);
legend("Discrete","Continuous")
;                xlabel("Time");
ylabel("Amplitude");
title("Unipolar        signal");
ylim([-2,2]);  %bipolar  signal
f7=5;               t7=0:0.01:1;
y7=square(2*pi*f7*t7);
subplot(3,3,7);     plot(t7,y7);
hold      on;       stem(t7,y7);
legend("Discrete","Continuous")
;                xlabel("Time");
ylabel("Amplitude");
title("Bipolar         signal");
ylim([-2,2]);       %exponential
```

```
signal  t8=0:0.1:5;  y8=exp(t8);
subplot(3,3,8); plot(t8,y8);
```

OBSERVATION



```
hold  on;  stem(t8,y8,'filled');
xlabel("Time");
ylabel("Amplitude");
title("Exponential    signal");
legend("Discrete","Continuous")
;       %triangular       signal
t9=0:0.01:1;              f9=3;
y9=sawtooth(2*pi*f9*t9,0.5);
subplot(3,3,9);    plot(t9,y9);
hold  on;  stem(t9,y9,'filled');
xlabel("Time");
ylabel("Amplitude");
title("Triangular     signal");
legend("Discrete","Continuous")
;
```

Result

Simulated and plotted basic test signal in Matlab.

1) Impulse signal
2) Unit step signal
3) Ramp signal
4) Sine signal
5) Cosine signal
6) Square Wave-Bipolar
7) Square Wave-Unipolar
8) Triangular signal
9) Exponential signal

VERIFICATION OF SAMPLING THEOREM

Aim

To verify sampling theorem using Matlab.

Theory

The Sampling Theorem, or Nyquist-Shannon theorem, states that a continuous signal can be accurately reconstructed from its samples if it is sampled at a rate at least twice the highest frequency present in the signal. This minimum sampling rate is called the Nyquist rate and is given by:

$$f_s \geq 2f_{max}$$

where fs is the sampling frequency and fmax is the maximum frequency in the signal.

Undersampling occurs when fs < 2fmax, leading to aliasing, where high-frequency components appear as lower frequencies.

Nyquist sampling is when fs=2fmax, ensuring perfect reconstruction.

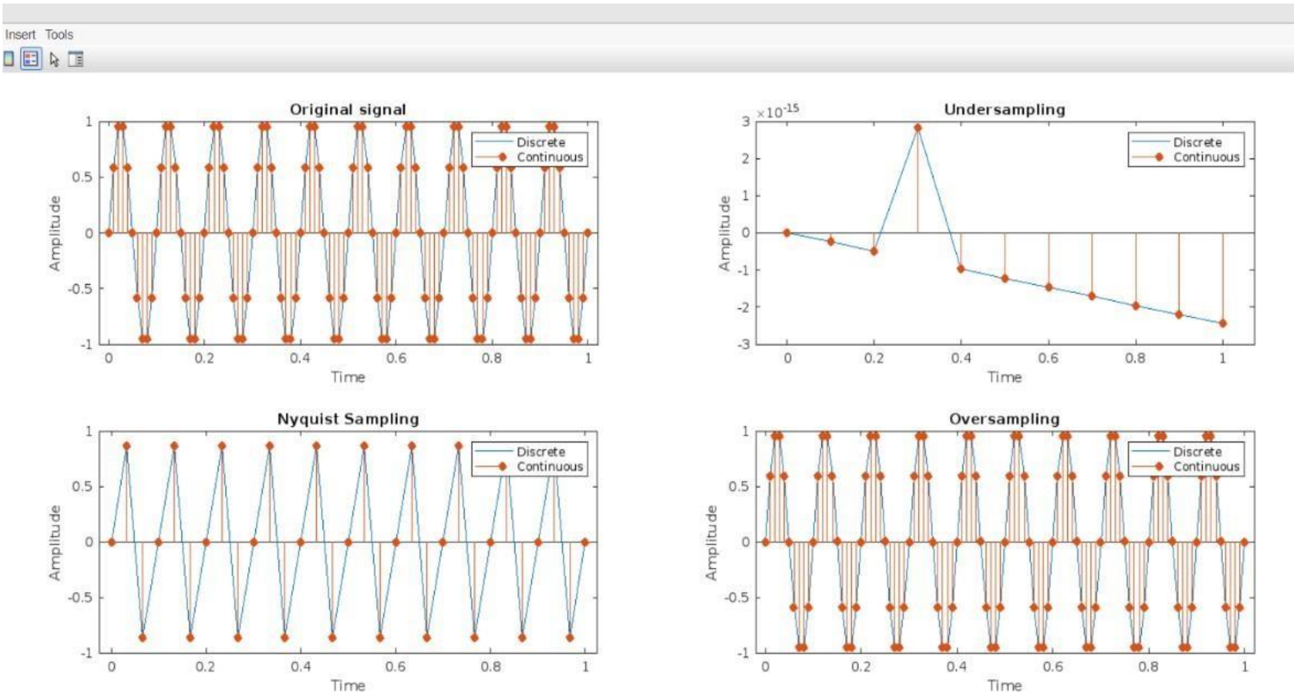
Oversampling is when fs > 2fmax, which increases redundancy without aliasing, improving signal quality and noise reduction.

PROGRAM

```
clc;   clear   all;   close   all;
t=0:0.01:1;               fm=10;
y=sin(2*pi*fm*t);
subplot(2,2,1);  plot(t,y);  hold
on;           stem(t,y,"filled");
xlabel("Time");
ylabel("Amplitude");
title("Original         signal");
legend("Discrete",
"Continuous");
```

```
%undersampling 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,"filled");
xlabel("Time");
ylabel("Amplitude");
title("Undersampling")
;
```

OBSERVATION



```
legend("Discrete","Continuous");
%nyquist    sampling    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,    "filled");
xlabel("Time");
ylabel("Amplitude");
title("Nyquist        Sampling");
legend("Discrete",
"Continuous");      %oversampling
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,"filled");
xlabel("Time");
ylabel("Amplitude");
title("Oversampling");
legend("Discrete",
"Continuous");
```

RESULT

Verified sampling theorem using Matlab.

Experiment no-3 Date:10-8-24

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 in signal processing to combine two signals, often to understand how one signal modifies another. It operates by sliding one function over another, multiplying corresponding values, and summing the products at each step. For two signals x(n)and h(n), their convolution y(n) is given by:

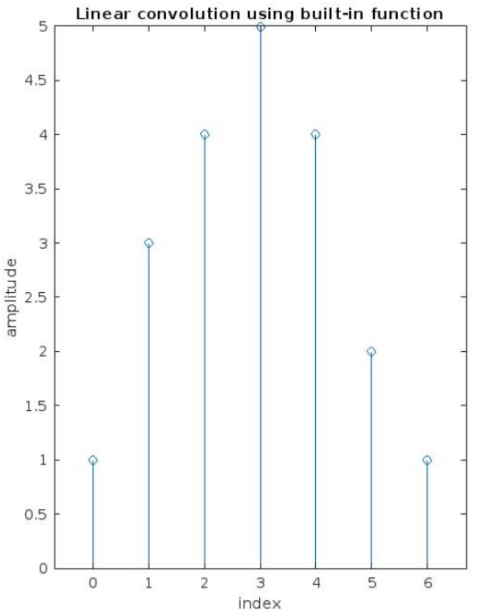
$$y(n) = \sum_{k=-\infty}^{\infty} x(k)h(n-k)$$

This process evaluates how much the signal h(n), often called the impulse response, overlaps with the input signal x(n). Linear convolution is used for filtering, smoothing, and analyzing system responses in digital and analog signal processing. The result of the convolution is usually a signal whose length is the sum of the lengths of the two input signals minus one.

OBSERVATION

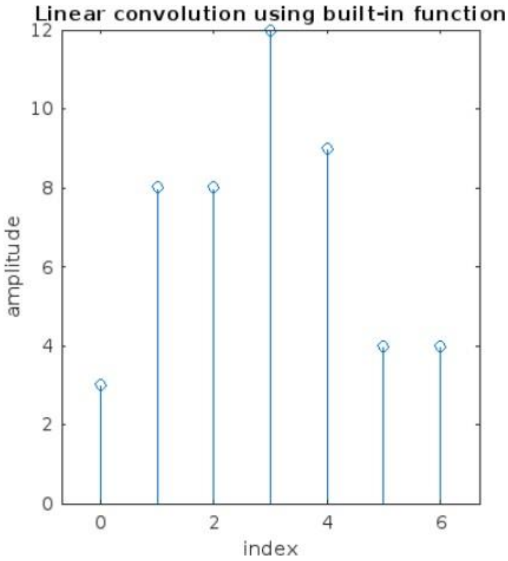
1) x(n) = [1 2 1 1] h(n)

= [1 1 1 1]



2) x(n) = [1 2 1 2] h(n)

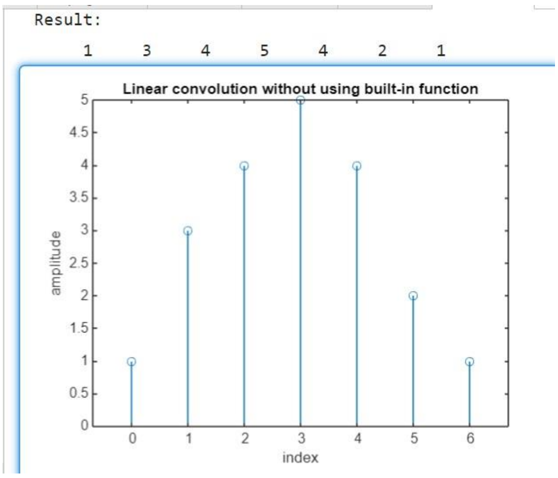
= [3 2 1 2]



PROGRAM

a) LINEAR CONVOLUTION USING BUILT IN FUNCTION `clc; close all; x=input("enter input"); x_index=input("enter index of x"); h=input("enter impulse response"); h_index=input("enter index of h"); y_index=min(x_index)+min(h_index):max(x_index)+max(h_index); y=conv(x,h); disp(y); subplot(1,2,1); stem(y_index,y); xlabel("index"); ylabel("amplitude"); title("Linear convolution using built-in function");` **OBSERVATION**

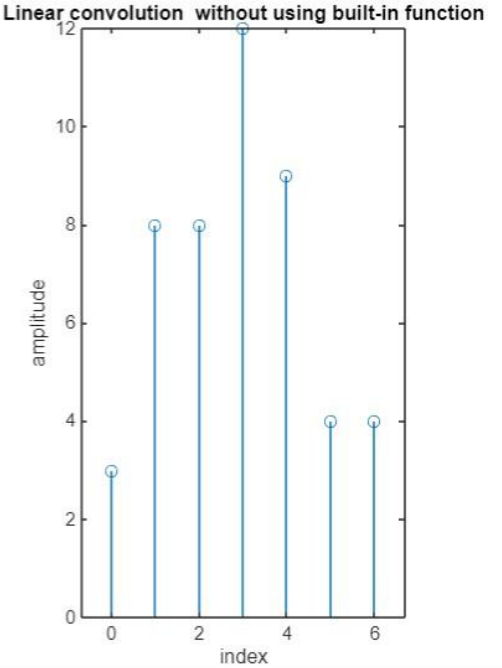
1) x(n) = [1 2 1 1] h(n) = [1 1 1 1]



2) x(n) = [1 2 1 2] h(n)

= [3 2 1 2]

3 8 8 12 9 4 4



b)LINEAR CONVOLUTION WITHOUT USING BUILT IN FUNCTION

```
% without using built in clc; close all; x=input("enter
input"); x_index=input("enter index of x"); h=input("enter
impulse response"); h_index=input("enter index of h");
y_index=min(x_index)+min(h_index):max(x_index)+max(h_index)
; n=length(x); m=length(h); len_y=length(y_index);
y=zeros(1,len_y);
for i=1:n for
    j=1:m
        y(i+j-1)=y(i+j-1)+x(i)*h(i);
    end
```

```
end disp("Result:")
disp(y)
stem(y_index,y);
xlabel("index");
ylabel("amplitude")
;
title("Linear convolution without using built-in function");
```

RESULT

Performed linear convolution with and without using built in function in Matlab.

Experiment No:4 Date:3-9-24

CIRCULAR CONVOLUTION

AIM

To find circular convolution using FFT, concentric circle method and matrix method using Matlab.

THEORY

Circular convolution is a mathematical operation used primarily in signal processing. It involves wrapping one signal around a circular buffer and performing the convolution operation on it, often used when signals are periodic or when working with discrete Fourier transforms (DFT). This technique ensures that the result maintains periodicity by aligning the endpoints of signals. It is computationally efficient and widely applied in fast algorithms like the Fast Fourier Transform (FFT). It can be performed by 3 methods:

Using FFT (Fast Fourier Transform):

- Circular convolution is performed by transforming the sequences to the frequency domain using FFT, multiplying them element-wise, and transforming them back using the inverse FFT.

$$y[n] = \mathrm{IFFT}(\mathrm{FFT}(x[n]) \cdot \mathrm{FFT}(h[n]))$$

Concentric Circle Method:

- This is a graphical method where one sequence is placed in a circular pattern, and the other sequence is rotated around it. The inner product of corresponding values after each rotation gives the result.

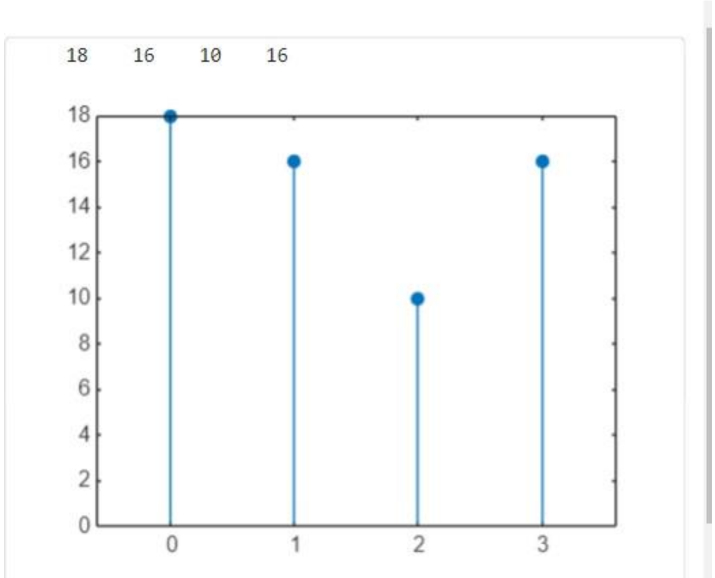
Matrix Method:

- Circular convolution can be represented as matrix multiplication, where one sequence is arranged in a circulant matrix, and the other is a column vector. y=C.h

  where C is the circulant matrix formed from x, and h is the vecto

OBSERVATION

CIRCULAR CONVOLUTION USING FFT



PROGRAM

a) Circular convolution using FFT

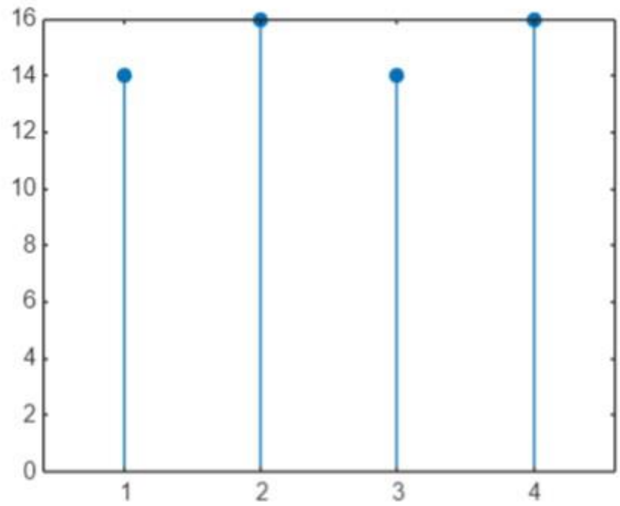
```
clc;  clear;  close  all;
x=input("Enter the seq1:");
h=input("Enter the seq2:");
x_len=length(x);
h_len=length(h);
n=max(x_len,h_len);
xnew=[x  zeros(1,n-x_len)];
hnew=[h  zeros(1,n-h_len)];
x1=fft(xnew);
h1=fft(hnew);    y1=x1.*h1;
y=ifft(y1);    y_ind=0:n-1;
disp(y);      stem(y_ind,y,
"filled");
```

OBSERVATION

b) CIRCULAR CONVOLUTION USING CONCENTRIC CIRCLE METHOD

```
Reversed x
     1     2     1     2
```

```
Convolution product
    14    16    14    16
```



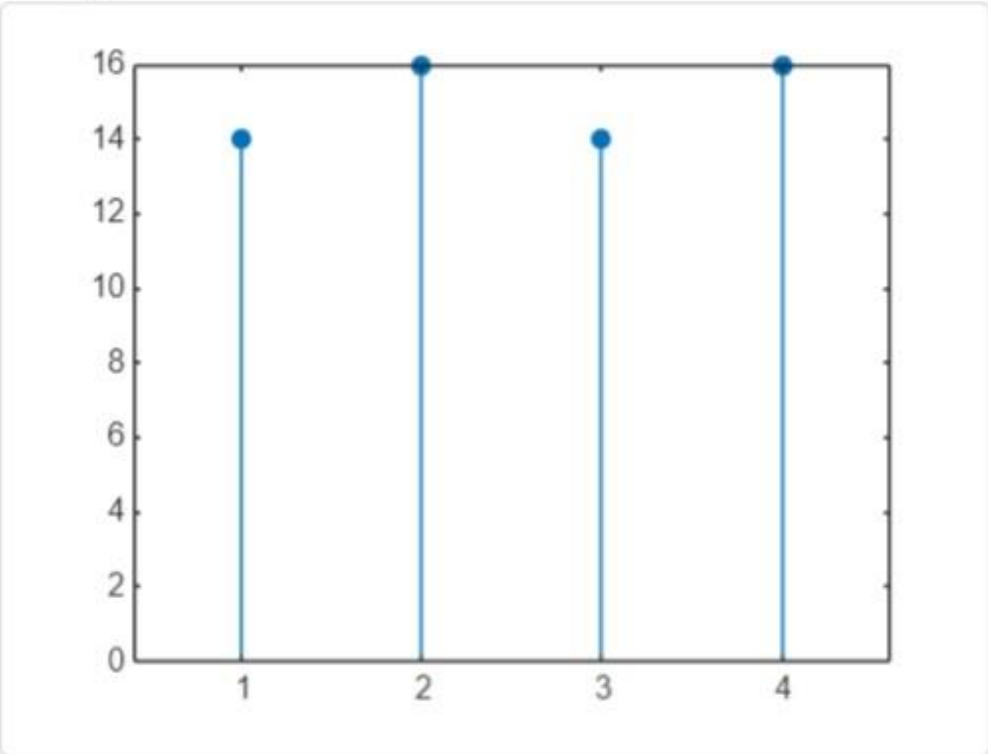
b) Circular convolution using Concentric Circle Method

```
clc; clear; close; x=[2 1 2 1]; X=x(:, end-1:1); h=[1 2 3 4]; for i=1:length(x) x=[x(end) x(1:end-1)]; h1=h; y(i)=sum(x.*h1); end disp(y);
```

OBSERVATION

c) CIRCULAR CONVOLUTION USING MATRIX METHOD

```
14
16
14
16
```



c)Circular convolution using Matrix Method

```
%Circular convolution using matrix multiplication
clc; close all; clear all; xn=[2 1 2 1]; %defining
the matrix hn=[1 2 3 4]; h=[ ]; hn=hn(:,end:-
1:1);%accesing the elements of hn
for i=1:length(hn) hn=[hn(end)
    hn(1:end-1)];
    h=[h;hn];
end y=h*xn';
disp(y);
```

```
stem(y,'filled')

;
```

RESULT

Obtained circular convolution using FFT, concentric circle method and matrix method in Matlab

Experiment No:5 Date:10-9-24

LINEAR CONVOLUTION USING CIRCULAR CONVOLUTION AND VICE VERSA

AIM

To perform linear convolution using circular convolution and vice versa using Matlab.

THEORY Performing Linear Convolution Using Circular Convolution

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.

Performing Circular Convolution Using Linear Convolution

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. E

OBSERVATION

```
1     3     6     9     7     4

1     3     6     9     7     4
```

PROGRAM a) Linear convolution using circular convolution

```
%linear convolution using circular convolution
clc; clear; close all; x=[1,2,3,4]; y=[1,1,1];
xl=length(x);   yl=length(y);   zl=(xl+yl)-1;
xn=[x zeros(1,zl-xl)]; yn=[y zeros(1,zl-yl)];
xa=fft(xn);      ya=fft(yn);      ans=xa.*ya;
anss=ifft(ans);  disp(anss);  answ=conv(x,y);
disp(answ);
```

OBSERVATION

```
8     7     6     9
```

b) Circular convolution using linear convolution

```
%circular convolution using linear convolution
clc; clear; close all; x = [1, 2, 3, 4]; h = [1,
1,    1];    y=conv(x,h);    z=max(length(x),
length(h));    r    =    [y(1:z)];    new    =
[y(z+1:length(y)) zeros(1, length(y)-z)];
for k = 1:z-1
    r(k)=r(k)+new(k);
end disp(r);
```

RESULT

Performed linear convolution using circular convolution and vice versa using Matlab.

Experiment No:6 Date: 24/09/24

DFT and IDFT

Aim

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

Theory:

The Discrete Fourier Transform (DFT) is a fundamental mathematical tool used in signal processing, communication systems, and many areas of engineering and science. It converts a discrete sequence (signal) from the time domain into its representation in the frequency domain. The DFT transforms a finite sequence of equally spaced samples of a function into a sequence of coefficients of complex sinusoids, ordered by their frequencies.

The Inverse Discrete Fourier Transform (IDFT) is a mathematical process that converts a sequence of complex numbers in the frequency domain back into the time domain. It is the inverse operation of the Discrete Fourier Transform (DFT) and is used to recover the original time-domain sequence from its DFT.

Program:

```
%dft using inbuilt and manual methods
clc;   clear   all;   close   all;
x=input('Enter   the   sequence:');
N=input('enter the N point DFT: ');
l=length(x);   x=[x   zeros(1,N-1)];
X=zeros(N,1);
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);
```

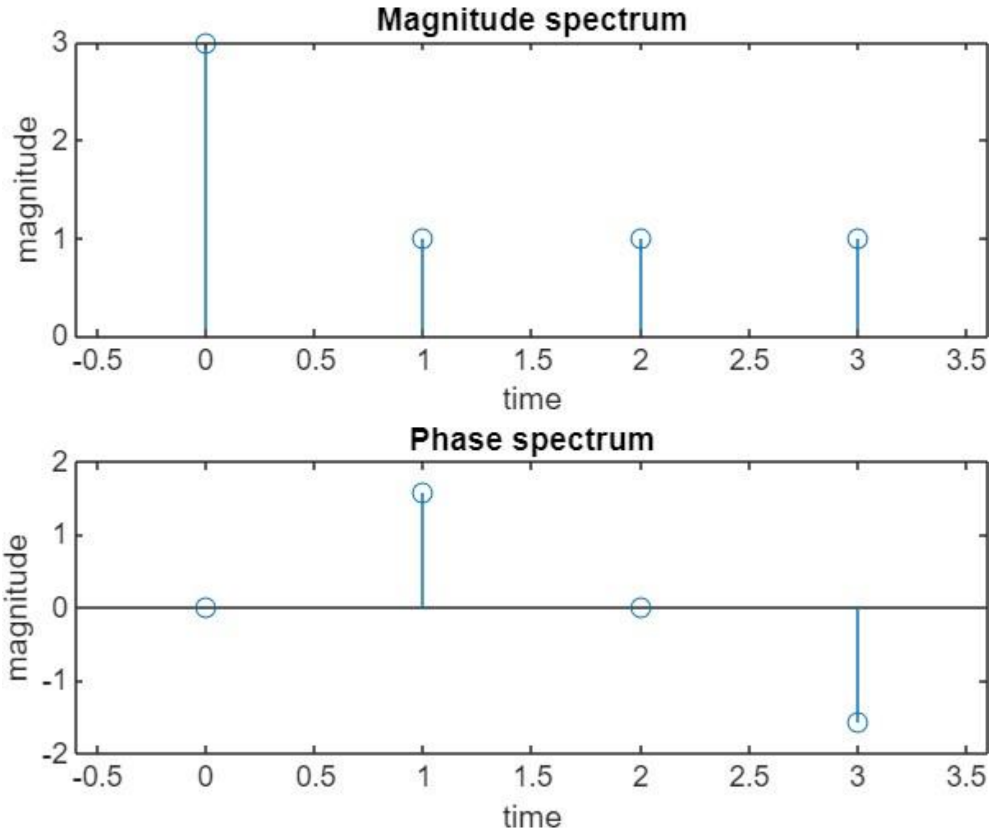
Observation:

Enter the sequence:[1 0 1 1] enter

the N point DFT: 4

X

3.0000 + 0.0000i
-0.0000 + 1.0000i
1.0000 - 0.0000i
0.0000 - 1.0000i
DFT
3.0000 + 0.0000i 0.0000 + 1.0000i 1.0000 + 0.0000i 0.0000 - 1.0000i




```
disp("DFT"); disp(fft(x,N));
%magnitude spectrum k=0:N-1;
mag=abs(X);  subplot(2,1,1);
stem(k,mag); xlabel('time');
ylabel('magnitude');
title('Magnitude spectrum');
%phase              spectrum
phase=angle(X);
subplot(2,1,2);
stem(k,phase);
xlabel('time');
```

```
ylabel('magnitude');
title('Phase spectrum');
```

Enter the sequence:[1 0 1 1]

Enter the N point DFT: 8

X

   3.0000 + 0.0000i

   0.2929 - 1.7071i

  -0.0000 + 1.0000i

   1.7071 + 0.2929i

   1.0000 - 0.0000i

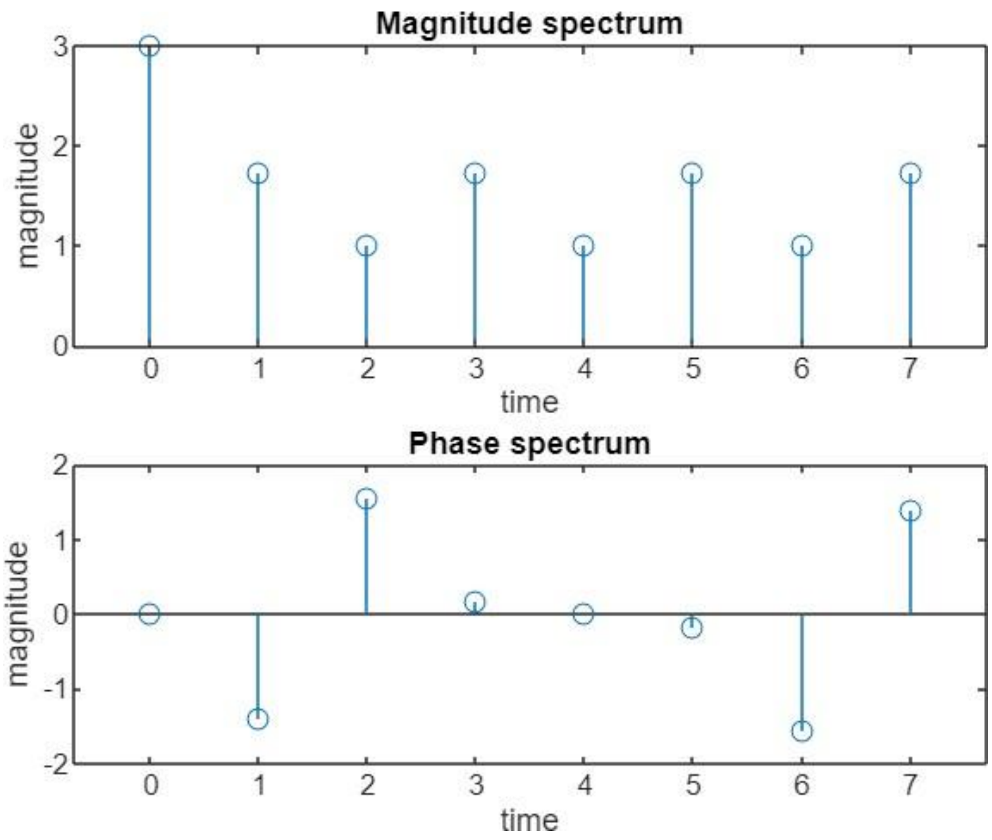
   1.7071 - 0.2929i

   0.0000 - 1.0000i

   0.2929 + 1.7071i

DFT

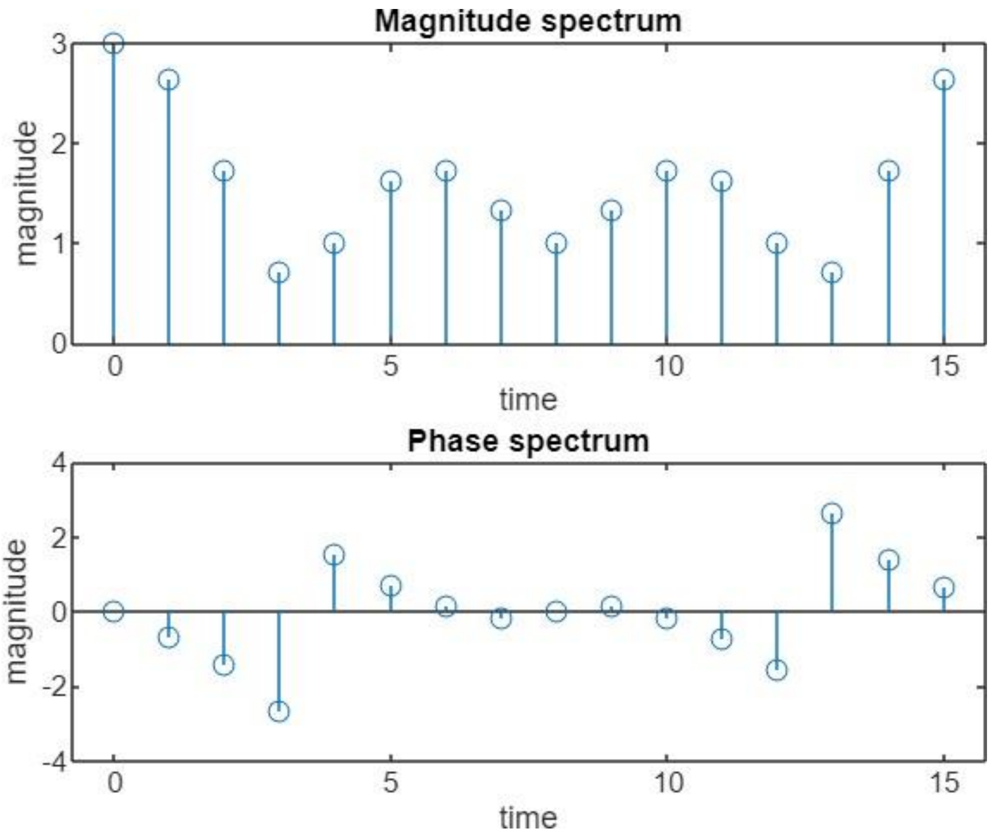
   3.0000 + 0.0000i 0.2929 - 1.7071i 0.0000 + 1.0000i 1.7071 + 0.2929i 1.0000 + 0.0000i 1.7071 - 0.2929i 0.0000 - 1.0000i 0.2929 + 1.7071i



```
Enter the sequence:[1 0 1 1] enter

the N point DFT: 16

X
   3.0000 + 0.0000i
   2.0898 - 1.6310i
   0.2929 - 1.7071i
  -0.6310 - 0.3244i
  -0.0000 + 1.0000i
   1.2168 + 1.0898i
   1.7071 + 0.2929i
   1.3244 - 0.2168i
   1.0000 - 0.0000i
   1.3244 + 0.2168i
   1.7071 - 0.2929i
   1.2168 - 1.0898i
   0.0000 - 1.0000i
  -0.6310 + 0.3244i
   0.2929 + 1.7071i
   2.0898 + 1.6310i
DFT
        3.0000 + 0.0000i 2.0898 - 1.6310i  0.2929 - 1.7071i -0.6310 - 0.3244i  0.0000 + 1.0000i
     1.2168 + 1.0898i 1.7071 + 0.2929i  1.3244 - 0.2168i  1.0000 + 0.0000i  1.3244 + 0.2168i
1.7071 - 0.2929i 1.2168 - 1.0898i  0.0000 - 1.0000i -0.6310 + 0.3244i  0.2929 + 1.7071i 2.0898
+ 1.6310i
```



Observation:

Enter the sequence:[1 1 1 0]

Enter the N point of DFT:4

x

0.7500 + 0.0000i
0.0000 + 0.2500i
0.2500 - 0.0000i
-0.0000 - 0.2500i

IDFT

0.7500 + 0.0000i 0.0000 + 0.2500i 0.2500 + 0.0000i 0.0000 - 0.2500i

```
%idft using inbuilt and manual functions
clc; clear all; close all;
X=input('Enter the sequence:'); N=input('Enter
the N point of DFT:');
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('IDFT');
disp(ifft(X,N))
;
```

Observation:

Enter the input sequence: [1 0 1 1]

DFT of the input sequence (using Twiddle Factor Matrix):

```
   3.0000 + 0.0000i
  -0.0000 + 1.0000i
   1.0000 - 0.0000i
   0.0000 - 1.0000i
```

```
% DFT using twiddle factor matrix x =
input('Enter the input sequence: ');
N = length(x);
W    = exp(-1i*2*pi*(0:N-1)'*(0:N-1)/N);
X    = W * x(:);
disp('DFT of the input sequence (using Twiddle Factor Matrix):');
disp(X);
```

Observation:

Enter the input sequence: [1 0 1 1]

IDFT of the input sequence (using Twiddle Factor Matrix):

```
   0.7500 + 0.0000i
  -0.0000 - 0.2500i
   0.2500 + 0.0000i
   0.0000 + 0.2500i
```

```
% IDFT using twiddle factor matrix x =
input('Enter the input sequence: ');
N = length(x);
W = exp(1i*2*pi*(0:N-1)'*(0:N-1)/N); X_idft
= (1/N) * (W * x(:));
disp('IDFT of the input sequence (using Twiddle Factor Matrix):');
disp(X_idft);
```

Result:

Computed DFT and IDFT using inbuilt and manual methods and Twiddle factor matrix and verified the output.

Experiment No:7 Date: 01/10/24

PROPERTIES OF DFT

Aim

To prove the following properties of DFT

- Linearity
- Convolution
- Multiplication
- Parseval's Theorem **Theory:**

Linearity:

The DFT is a linear transformation, meaning that the DFT of the sum of two signals is equal to the sum of their individual DFTs, and multiplying a signal by a constant in the time domain results in the DFT being multiplied by the same constant.If x1(n) and x2(n) are two sequences and a and b are constants then:

DFT(ax1(n)+bx2(n))= a.DFT(x1(n))+b.DFT(x2(n))

Multiplication:

The DFT of a pointwise multiplication (element-wise product) of two signals in the time domain corresponds to the circular convolution of their DFTs in the frequency domain. If x1(n) and x2(n) are two signals then:

DFT{x1(n).x2(n)}= 1/N DFT{x(n)}⊛DFT{h(n)} Convolution:

The DFT of the convolution of two sequences in the time domain is the element-wise multiplication of their DFTs in the frequency domain. If x1(n) and x2(n) are two signals, then: DFT{x1(n)∗x2(n)}=DFT{x1(n)}·DFT{x2(n)}

Parseval's Theorem:

Parseval's theorem states that the total energy of a discrete-time signal (the sum of the squared magnitudes of the signal in the time domain) is equal to the total energy of its DFT (the sum of the squared magnitudes of the DFT coefficients).

Program:

```
%linearity property clc; clear all;
close all; x1=input('Enter the first
sequence:');
```

Observation:

Enter the first sequence:[1 2 3 4]

Enter the second sequence:[1 1 1]

LHS:

29.0000 + 0.0000i -4.0000 + 1.0000i -1.0000 + 0.0000i -4.0000 – 1.0000i RHS:

29.0000 + 0.0000i -4.0000 + 1.0000i -1.0000 + 0.0000i -4.0000 – 1.0000i

LHS=RHS

Linearity Property Verified

```
x2=input('Enter the second sequence:');
a=2; b=3; l1=length(x1); l2=length(x2);
if  l1>l2  x2=[x2  zeros(1,l1-l2)]  else
x1=[x1 zeros(1,l2-l1)];
end
LHS=fft((a.*x1)+(b.*x2));
RHS=[a.*fft(x1)+b.*fft(x2)]
; disp('LHS:'); disp(LHS);
disp('RHS:'); disp(RHS);
disp(['LHS=RHS')_
```

Observation:

LHS
8 7 6 9
RHS
8 7 6 9
Convolution property verified

```
%Convolution  property  clc;
clear   all;   close   all;
x=input('enter sequence 1');
h=input('enter sequence 2');
N=max(length(x),length(h));
X=[x zeros(1,N-length(x))];
H=[h zeros(1,N-length(h))];
X1=fft(X);
H1=fft(H);
LHS= cconv(X,H,N);
RHS=ifft(X1.*H1);
disp(LHS);
disp(RHS);
if LHS==RHS disp('Convolution property
    verified');
else disp('Convolution property
verified'); end
```

Observation:

enter the first sequence:
[1 2 3 4] enter the second

sequence:

[1 1 1]

6.0000 + 0.0000i -2.0000 - 2.0000i 2.0000 + 0.0000i -2.0000 + 2.0000i

6.0000 + 0.0000i -2.0000 - 2.0000i 2.0000 + 0.0000i -2.0000 + 2.0000i

```
%multiplication property clc; clear
all; close all; x1=input('enter the
first sequence:'); x2=input('enter the
second sequence:'); l1=length(x1);
l2=length(x2); n=max(l1,l2); x1=[x1
zeros(1,n-l1)]; x2=[x2 zeros(1,n-l2)];
lhs=fft(x1.*x2); X1=fft(x1);
X2=fft(x2); rhs=cconv(X1,X2,n)/n;
disp(lhs); disp(rhs);
```

Observation: enter the first sequence: [1 2 3 4]
enter the second sequence:
[1 1 1]
LHS
6
RHS
6

```
%parseval's  theorem  clc;  clear  all;
close  all;  x1=input('enter  the  first
sequence:'); x2=input('enter the second
sequence:');             l1=length(x1);
l2=length(x2);   n=max(l1,l2);   x1=[x1
zeros(1,n-l1)];  x2=[x2  zeros(1,n-l2)];
lhs=sum(x1.*conj(x2));
rhs=sum(fft(x1).*conj(fft(x2)))/n;
disp('LHS');  disp(lhs);  disp('RHS');
disp(rhs);
```

Result:

Verified linearity, convolution, multiplication,and parseval’s properties of DFT

Experiment No:8 Date: 08/10/24

OVERLAP SAVE AND ADD METHOD

Aim

To perform linear convolution of two sequences using overlap save and add method

Theory:

The Overlap-Add and Overlap-Save methods are efficient techniques used to perform linear convolution of long signals with finite impulse response (FIR) filters using the Fast Fourier Transform (FFT). Both methods help reduce computational complexity by breaking a long signal into smaller chunks, processing them independently in the frequency domain, and then combining the results.

The Overlap Add method splits the input signal into overlapping segments, performs convolution on each segment using the FFT, and then adds the overlapping parts to reconstruct the final output. This method is efficient for filtering long signals by using FFT-based convolution.

The Overlap Save method also divides the input signal into segments, but unlike OLA, it saves the non-overlapping parts and discards the overlapping parts of the convolution output.

Program:

```
%overlap  save  method  clc;  clear
all;  close  all;  x=input("Enter
sequence    1");    h=input("Enter
sequence    2");    N=input('Enter
length to divide'); if N<length(h)
disp('not possible');
else
    xl=length(x);
    hl=length(h);
    L=N-hl+1;
    hnew=[h zeros(1,N-hl)]; xnew=[zeros(1,hl-
    1),x,zeros(1,N-1)]; y=[];
for i=1:L:length(xnew)-N+1
    XB=xnew(i:i+N-1);
    YB=ifft(fft(XB).*fft(hnew));
    y=[y,YB(hl:end)];
end disp(y(1:xl+hl-1));
```

```
end
```

4 3 1

Observation:

Enter sequence 1[3 -1 0 1 3 2 0 1 2 1]
Enter sequence 2[1 1 1]
Enter length to divide3
final convoluted sequence
3 2 2 0 4 6 5 3 3

Observation: 8 9

Enter the input sequence: [0 1 2 3 4 5 6 7 8 9]

Enter the filter sequence: [1 0 1]

Enter the segment length (choose N >= Lh): 3

final convoluted sequence:

```
    0     1     2     4     6     8    10    12    14    16
```

```
%overlap add method clc; clear all; close
all; x = input('Enter the input sequence:
'); h = input('Enter the filter sequence:
');
Lx = length(x);
Lh = length(h);
N = input('Enter the segment length (choose N >= Lh): '); if N < Lh
error('Segment length N must be greater than or equal to filter
length'); end
x =  [x,  zeros(1,  N  -  mod(Lx,  N))];  Lx_padded  =
  length(x); y = zeros(1, Lx_padded + Lh - 1); for i
  = 1:N:Lx_padded x_segment = x(i:i+N-1); y_segment
  = conv(x_segment, h);
           y(i:i+length(y_segment)-1) = y(i:i+length(y_segment)-1) +
y_segment; end
y = y(1:Lx + Lh - 1);
disp('final convoluted sequence:'); disp(y);
```

Result:

Implemented overlap add and overlap save method using MATLAB and verified the output

Experiment No: 9 Date:14/10/24

IMPLEMENTATION OF FIR FILTER

Aim:

Design FIR Filters Using Window Methods **Theory:**

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.

Steps for FIR Filter Design Using Windows 1. **Define the Ideal Impulse Response**
First, compute the ideal impulse response, hideal(n)h_ideal(n)hideal(n), of the desired filter in the time domain. For example, for a low-pass filter with a cutoff frequency fcf_cfc, the ideal impulse response is:

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

where:

- f_c is the cutoff frequency in normalized units,
- N is the filter length, ○ n is the sample index.

This ideal response is typically non-causal, so it is shifted to make it causal by adding (N - 1) / 2 to the sample index.

2. **Select an Appropriate Window Function**
To achieve a practical FIR filter, select a window function, w(n)w(n)w(n), that will shape the frequency response. The choice of window affects the trade-off between the main lobe width (frequency resolution) and the sidelobe levels (leakage). Common windows include the **Hamming**, **Hann**, **Blackman**, **Kaiser**, and **rectangular** windows, each defined by specific equations:

- **Rectangular Window**: w(n) = 1 ○ **Triangular (Bartlett) Window**: w(n) = 1 – 2*abs(n) / (N - 1) ○ **Hamming Window**: w(n) = 0.54 - 0.46 * cos(2 * pi * n / (N - 1)) ○ **Hanning Window**: w(n) = 0.5 * (1 - cos(2 * pi * n / (N - 1))) ○ **Blackman Window**: w(n) = 0.42 - 0.5 * cos(2 * pi * n / (N - 1)) + 0.08 * cos(4 * pi * n / (N - 1)
- **Kaiser Window**: w(n) = I0(beta * sqrt(1 - (2 * n / (N - 1) - 1)^2)) / I0(beta)

where I0 is the modified zero-th order Bessel function, and beta is a parameter controlling the trade-off between the main lobe width and sidelobe levels.

3. **Apply the Window to the Ideal Impulse Response**
   Multiply each point in the ideal impulse response h_ideal(n) by the corresponding point in the window function w(n) to get the windowed impulse response h(n):

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

The result is a practical, finite impulse response that approximates the ideal response with controlled sidelobes.

4. **Construct the FIR Filter**
   The final impulse response h(n) defines the coefficients of the FIR filter. These coefficients can now be used in a filtering algorithm (e.g., convolution with input data) to perform the desired filtering operation.

Example: Designing a Low-Pass FIR Filter Using a Hamming Window

1. **Specify the Filter Requirements**:

   ○ Cutoff frequency f_c: 0.2 (normalized frequency) ○

   Filter length N: 51 (odd number for symmetry)

2. **Compute the Ideal Impulse Response**:

   h_ideal(n) = sin(2 * pi * 0.2 * (n - (51 - 1) / 2)) / (pi * (n - (51 - 1) / 2))

3. **Apply the Hamming Window**:

   w(n) = 0.54 - 0.46 * cos(2 * pi * n / 50)

   Then, compute h(n) = h_ideal(n) * w(n).

4. **Use h(n) as FIR Filter Coefficients**: The resulting h(n) values form the coefficients of the FIR filter, which can be used in a filtering algorithm. **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, allowing design flexibility.

Disadvantages:

- **Fixed Frequency Response**: Once the window is chosen, the frequency response characteristics are determined, limiting customization.
- **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 = input('Enter the value of N=');
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);
wh=hamming(N);
wn=hanning(N);
wt=bartlett(N);     hn     =
hd.*wr';        hn1=hd.*wh';
hn2=hd.*wn'; hn3=hd.*wt';  w
=     0:0.01:pi;     h     =
freqz(hn,1,w);     h1     =
freqz(hn1,1,w);     h2     =
freqz(hn2,1,w);
h3=freqz(hn3,1,w);
subplot(4,2,1);
plot(w/pi,10*log10(abs(h)))
;
title('low pass filter using rectangular window'); xlabel('Normalized
frequency');
ylabel('Magnitude   in   dB');   subplot(4,2,2);
stem(wr); title('Rectangular window Sequence');
xlabel('No. of Samples'); ylabel('Amplitude');
subplot(4,2,3);   plot(w/pi,10*log10(abs(h1)));
title('low pass filter using hamming window');
xlabel('Normalized                  frequency');
ylabel('Magnitude   in   dB');   subplot(4,2,4);
```

```
stem(wh);  title('Hamming  window  Sequence');
xlabel('No. of Samples'); ylabel('Amplitude');
subplot(4,2,5);  plot(w/pi,10*log10(abs(h2)));
title('low pass filter using hanning window');
xlabel('Normalized                frequency');
ylabel('Magnitude  in  dB');  subplot(4,2,6);
stem(wn);  title('Hanning  window  Sequence');
xlabel('No. of Samples'); ylabel('Amplitude');
subplot(4,2,7);  plot(w/pi,10*log10(abs(h2)));
title('low pass filter using bartlett window');
xlabel('Normalized                frequency');
ylabel('Magnitude  in  dB');  subplot(4,2,8);
stem(wt);
title('bartlett window Sequence');
xlabel('No.     of     Samples');
ylabel('Amplitude');
```

2. HIGH PASS FILTER

```
clc;   clear   all;
close          all;
wc=0.9*pi;
eps=0.001;
N = input('Enter the value of N='); alpha
= (N-1)/2;

n = 0:1:N-1;
hd=(sin(pi*(n-alpha+eps))-sin(wc*(n-
alpha+eps)))./(pi*(n-  alpha+eps));  wr  =  boxcar(N);
wh=hamming(N);  wn=hanning(N);  wt=bartlett(N);  hn  =
hd.*wr';  hn1=hd.*wh';  hn2=hd.*wn';  hn3=hd.*wt';  w  =
0:0.01:pi; h = freqz(hn,1,w);
```

```
h1 = freqz(hn1,1,w); h2 =
freqz(hn2,1,w);
h3=freqz(hn3,1,w);
subplot(4,2,1);
plot(w/pi,10*log10(abs(h)))
;
title('high pass filter using rectangular window');
xlabel('Normalized  frequency');  ylabel('Magnitude
in dB'); subplot(4,2,2); stem(wr);
title('Rectangular      window      Sequence');
xlabel('No. of Samples'); ylabel('Amplitude');
subplot(4,2,3);   plot(w/pi,10*log10(abs(h1)));
title('high pass filter using hamming window');
xlabel('Normalized                 frequency');
ylabel('Magnitude   in   dB');   subplot(4,2,4);
stem(wh);  title('Hamming  window  Sequence');
xlabel('No. of Samples'); ylabel('Amplitude');
subplot(4,2,5);   plot(w/pi,10*log10(abs(h2)));
title('high pass filter using hanning window');
xlabel('Normalized                 frequency');
ylabel('Magnitude   in   dB');   subplot(4,2,6);
stem(wn);
title('Hanning window Sequence'); xlabel('No. of
Samples');  ylabel('Amplitude');  subplot(4,2,7);
plot(w/pi,10*log10(abs(h2)));  title('high  pass
filter      using      bartlett      window');
xlabel('Normalized                  frequency');
ylabel('Magnitude   in   dB');   subplot(4,2,8);
stem(wt);  title('bartlett  window  Sequence');
xlabel('No. of Samples'); ylabel('Amplitude');
```

.

3. Band pass filter

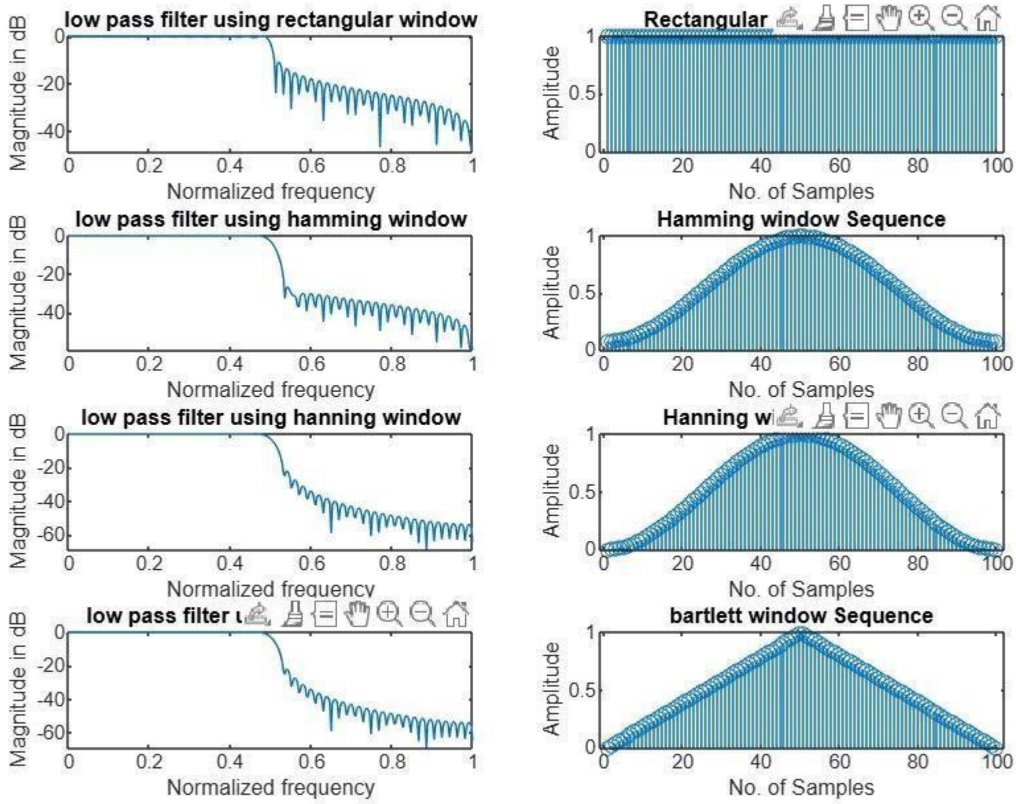
```
clc;  clear
all;  close
all;
wc1=0.5*pi
;
wc2=0.9*pi
;
eps=0.001;
N = input('Enter the value of N='); alpha
= (N-1)/2;
n = 0:1:N-1;
hd   =    (sin(wc2*(n-alpha+eps))-sin(wc1*(n-alpha+eps)))./(pi*(n-
alpha+eps)); wr = boxcar(N); wh=hamming(N); wn=hanning(N);
wt=bartlett(N);     hn     =
hd.*wr';        hn1=hd.*wh';
hn2=hd.*wn'; hn3=hd.*wt';  w
=     0:0.01:pi;     h     =
freqz(hn,1,w);     h1     =
freqz(hn1,1,w);     h2     =
freqz(hn2,1,w);
h3=freqz(hn3,1,w);
subplot(4,2,1);
plot(w/pi,10*log10(abs(h)))
;
title('band pass filter using rectangular window');
xlabel('Normalized  frequency');  ylabel('Magnitude
in dB'); subplot(4,2,2); stem(wr);
```

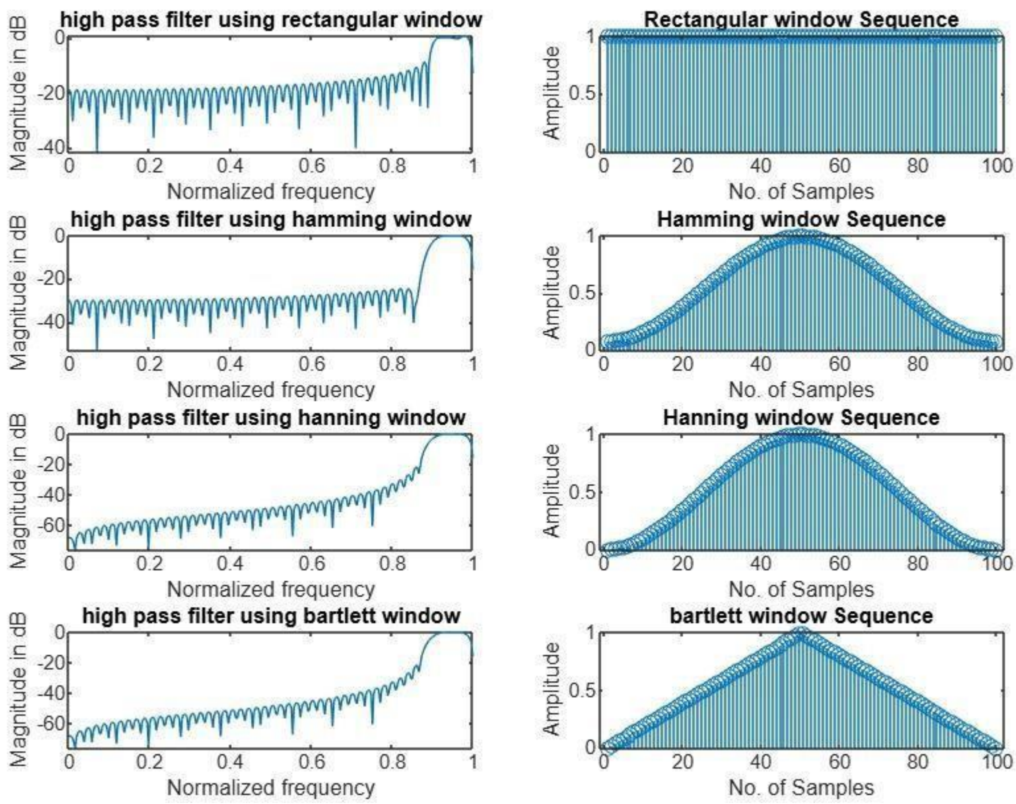
```
title('Rectangular      window      Sequence');
xlabel('No. of Samples'); ylabel('Amplitude');
subplot(4,2,3);   plot(w/pi,10*log10(abs(h1)));
title('band pass filter using hamming window');
xlabel('Normalized                    frequency');
ylabel('Magnitude   in   dB');   subplot(4,2,4);
stem(wh);  title('Hamming  window  Sequence');
xlabel('No. of Samples'); ylabel('Amplitude');
subplot(4,2,5);    plot(w/pi,10*log10(abs(h2)));
title('band pass filter using hanning window');
xlabel('Normalized                    frequency');
ylabel('Magnitude   in   dB');   subplot(4,2,6);
stem(wn);   title('Hanning   window   Sequence');
xlabel('No.  of  Samples');  ylabel('Amplitude');
subplot(4,2,7);    plot(w/pi,10*log10(abs(h2)));
title('band pass filter using bartlett window');
xlabel('Normalized                    frequency');
ylabel('Magnitude   in   dB');   subplot(4,2,8);
stem(wt);  title('bartlett  window  Sequence');
xlabel('No. of Samples'); ylabel('Amplitude');
```

2. <u>Band stop filter</u>

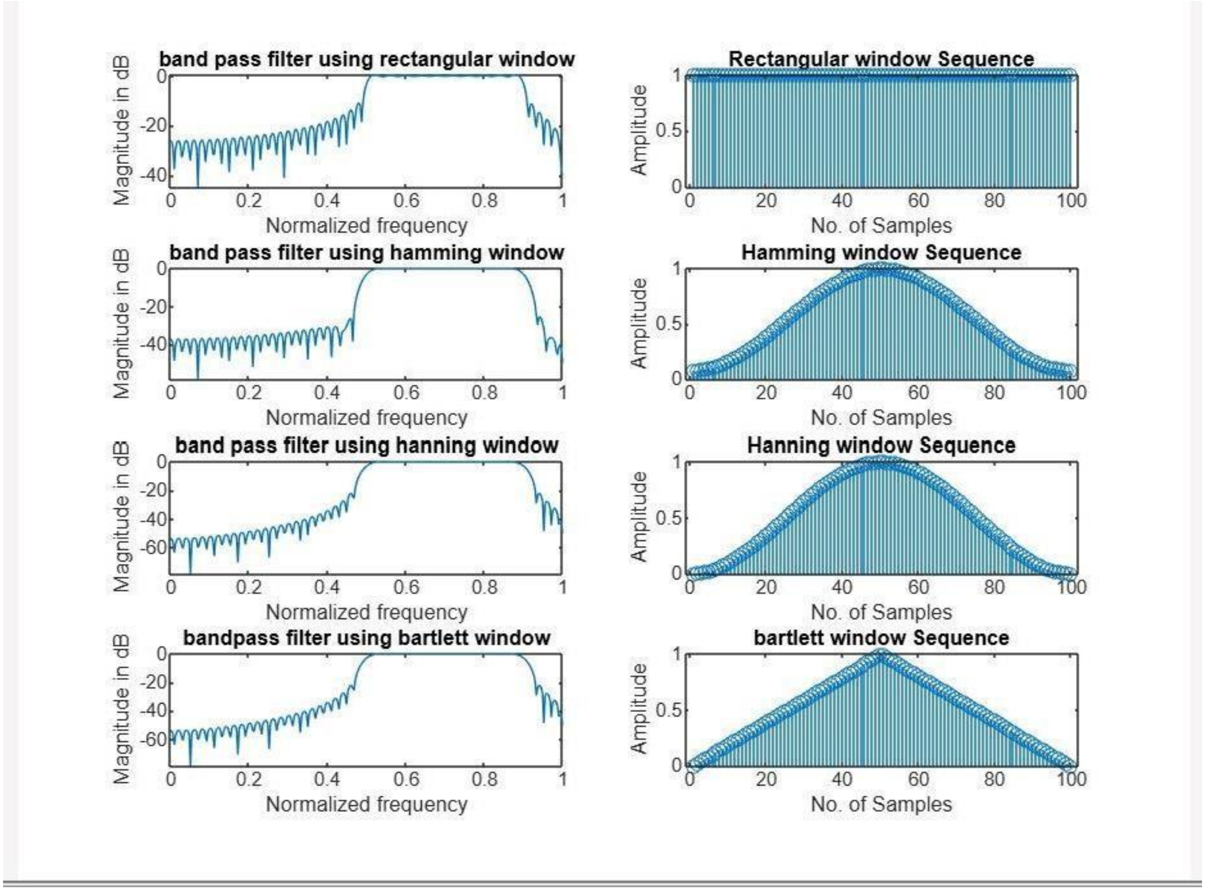
```
clc;  clear
all;  close
all;
wc1=0.5*pi
;
wc2=0.9*pi
;
eps=0.001;
```

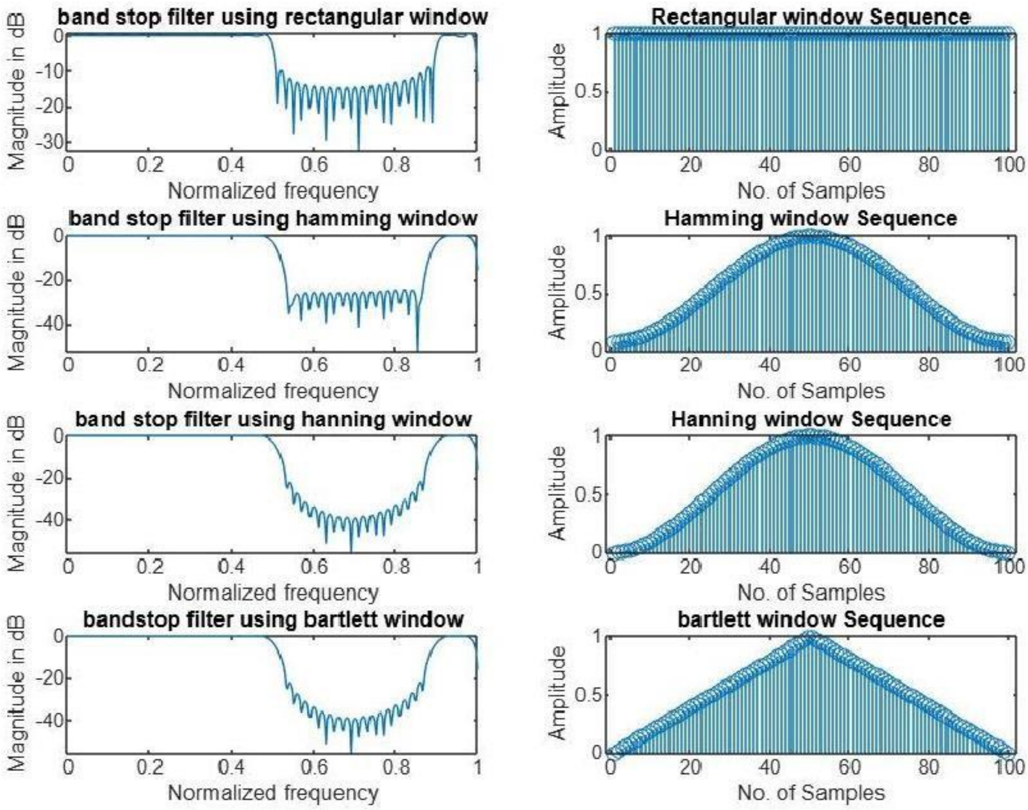
```
N = input('Enter the value of N='); alpha
= (N-1)/2;
n = 0:1:N-1;
hd = (sin(wc1*(n-alpha+eps))-sin(wc2*(n-alpha+eps))+sin(pi*(n-
alpha)))./(pi*(n-alpha+eps)); wr = boxcar(N); wh=hamming(N);
wn=hanning(N); wt=bartlett(N); hn = hd.*wr'; hn1=hd.*wh';
hn2=hd.*wn'; hn3=hd.*wt'; w = 0:0.01:pi; h = freqz(hn,1,w); h1
= freqz(hn1,1,w); h2 = freqz(hn2,1,w); h3=freqz(hn3,1,w);
subplot(4,2,1); plot(w/pi,10*log10(abs(h)));
title('band stop filter using rectangular window');
xlabel('Normalized  frequency');  ylabel('Magnitude
in dB'); subplot(4,2,2); stem(wr);
title('Rectangular      window      Sequence');
xlabel('No. of Samples'); ylabel('Amplitude');
subplot(4,2,3);   plot(w/pi,10*log10(abs(h1)));
title('band stop filter using hamming window');
xlabel('Normalized frequency');
```


low pass filter using rectangular window
Rectangular
low pass filter using hamming window
Hamming window Sequence
low pass filter using hanning window
Hanning w
low pass filter u
bartlett window Sequence
Magnitude in dB
Normalized frequency
Amplitude
No. of Samples


high pass filter using rectangular window
Rectangular window Sequence
high pass filter using hamming window
Hamming window Sequence
high pass filter using hanning window
Hanning window Sequence
high pass filter using bartlett window
bartlett window Sequence
Magnitude in dB
Normalized frequency
Amplitude
No. of Samples

```
ylabel('Magnitude in
dB'); subplot(4,2,4);
stem(wh);
title('Hamming window Sequence'); xlabel('No.
of Samples'); ylabel('Amplitude');
subplot(4,2,5); plot(w/pi,10*log10(abs(h2)));
title('band stop filter using hanning window');
xlabel('Normalized frequency');
ylabel('Magnitude in dB'); subplot(4,2,6);
stem(wn); title('Hanning window Sequence');
xlabel('No. of Samples'); ylabel('Amplitude');
subplot(4,2,7); plot(w/pi,10*log10(abs(h2)));
title('bandstop filter using bartlett window');
xlabel('Normalized frequency');
ylabel('Magnitude in dB'); subplot(4,2,8);
stem(wt); title('bartlett window Sequence');
xlabel('No. of Samples'); ylabel('Amplitude');
```


band pass filter using rectangular window
Rectangular window Sequence
band pass filter using hamming window
Hamming window Sequence
band pass filter using hanning window
Hanning window Sequence
bandpass filter using bartlett window
bartlett window Sequence
Magnitude in dB
Normalized frequency
Amplitude
No. of Samples


band stop filter using rectangular window
Rectangular window Sequence
band stop filter using hamming window
Hamming window Sequence
band stop filter using hanning window
Hanning window Sequence
bandstop filter using bartlett window
bartlett window Sequence
Magnitude in dB
Normalized frequency
Amplitude
No. of Samples

Result:

Performed Lowpass, highpass, bandpass, bandstop filters using windows method.

Experiment No:10 Date:14/10/2024

Familiarization of TMS 320C6748

Aim

To explore the architectural and functional capabilities of the TMS320C6748 DSP processor.

TMS 320C6748

Overview of the TMS320C6748 DSP Processor: The TMS320C6748 DSP processor is designed to handle intensive digital signal processing tasks efficiently. At its core is the powerful C674x™ DSP CPU, optimized for real-time embedded applications, multimediaprocessing, and other high-computation requirements.

DSP Subsystem and Memory Components: The DSP subsystem includes several memory components for efficient storage and data handling. A 32 KB L1 Program Cache and 32 KBL1 RAM ensure quick access to frequently used data, while the 256 KB L2 RAM provides storage for larger datasets. Additionally, a BOOT ROM is available to facilitate the processor's startup sequence.

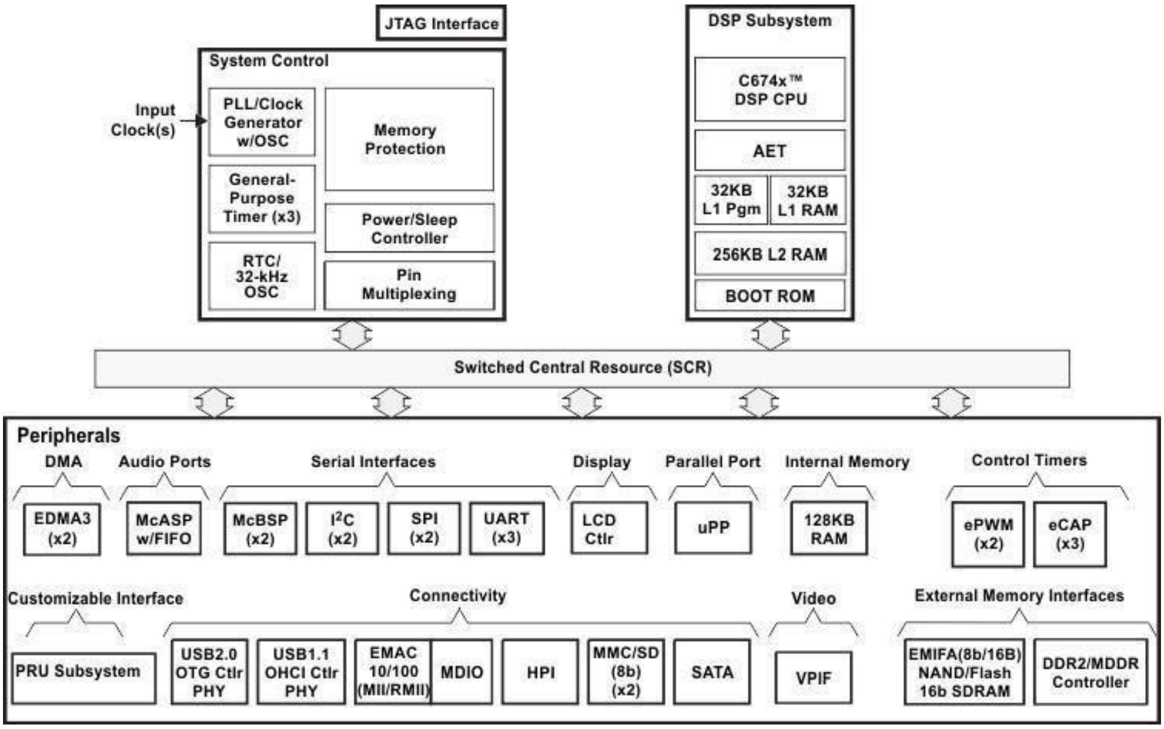
System Control Features: Essential to managing the processor's operation, the SystemControl section includes a PLL/Clock Generator with Oscillator (OSC) to supply clock signals, a General-Purpose Timer, and an RTC/32-kHz Oscillator for precise timing

functions. This section also provides Memory Protection for secure data handling, aPower/Sleep Controller to optimize power consumption, and Pin Multiplexing for I/O configuration flexibility.

Debugging and System Testing: For system testing and troubleshooting, the TMS320C6748includes a JTAG Interface. This interface supports connection to external debugging tools, allowing developers to perform in-depth analysis and testing.

Switched Central Resource (SCR): At the heart of data management, the SCR is a high- speed interconnect linking the DSP subsystem, system control, and various peripherals. TheSCR efficiently manages data flow, facilitating smooth communication across the processoreven when multiple subsystems are active.

Peripherals and Interfaces: The TMS320C6748 offers a versatile range of peripherals.
Direct Memory Access (DMA) through EDMA3 with two channels enables rapid data transfers without CPU involvement. Audio Ports, including McASP and McBSP, support audio data I/O, making the processor suitable for audio applications. Multiple Serial Interfaces (I2C, SPI, and UART) enable communication with devices like sensors and storageunits, while the LCD Controller supports direct display interfacing. Additionally, the 128 K Internal RAM provides further data storage.


JTAG Interface
DSP Subsystem
System Control
Input Clock(s)
PLL/Clock Generator w/OSC
Memory Protection
General-Purpose Timer (x3)
Power/Sleep Controller
RTC/ 32-kHz OSC
Pin Multiplexing
C674x™ DSP CPU
AET
32KB L1 Pgm
32KB L1 RAM
256KB L2 RAM
BOOT ROM
Switched Central Resource (SCR)
Peripherals
DMA
Audio Ports
Serial Interfaces
Display
Parallel Port
Internal Memory
Control Timers
EDMA3 (x2)
McASP w/FIFO
McBSP (x2)
I2C (x2)
SPI (x2)
UART (x3)
LCD Ctlr
uPP
128KB RAM
ePWM (x2)
eCAP (x3)
Customizable Interface
Connectivity
Video
External Memory Interfaces
PRU Subsystem
USB2.0 OTG Ctlr PHY
USB1.1 OHCI Ctlr PHY
EMAC 10/100 (MII/RMII)
MDIO
HPI
MMC/SD (8b) (x2)
SATA
VPIF
EMIFA(8b/16B) NAND/Flash 16b SDRAM
DDR2/MDDR Controller

PRU Subsystem for Real-Time Control: The Programmable Real-time Unit (PRU) subsystem adds customization and flexibility, enabling the processor to handle specializedtasks in real-time.

Connectivity Options: For connectivity, the processor supports USB 2.0 OTG, USB 1.1, Ethernet (EMAC 10/100) for network connections, as well as MDIO and HPI interfaces. Italso supports MMC/SD and SATA ports, enhancing its capability to interface with various storage devices.

Video Port Interface (VPIF): The VPIF feature makes the TMS320C6748 suitable for video applications by supporting video input and output functions.

Control Timers: The Control Timers section includes ePWM and eCAP timers, providing precise control over pulse-width modulation and capture events. These features are useful formotor control, sensor data acquisition, and other control-based applications.

External Memory Interfaces: To support additional memory, the processor includes interfaces for EMIFA (8b/16b), NAND/Flash 16b, and a DDR2/MDDR Controller, allowingthe connection of external DRAM and flash memory for data-intensive applications.

Application

The TMS320C6748 DSP processor is versatile, supporting applications across industries dueto its powerful DSP core, real-time control, and connectivity options. It is ideal for audio processing in digital audio workstations and industrial automation tasks like motor control and robotics. In the medical field, it handles real-time bio-signal processing for imaging and monitoring, while in video and image processing, it’s used for surveillance and computer vision. The processor also serves well in communication systems (e.g., modems and wirelessnetworks), automotive ADAS, energy management, test and measurement equipment, and IoT applications, enabling smart devices and automation in home environments.

Result

Studied and obtained a comprehensive overview of the TMS320C6748 DSP processor, highlighting its robust DSP CPU core, high-speed data handling through the Switched CentralResource (SCR), and its extensive set of peripherals and connectivity option

Experiment : 11 Date:19/10/2024

Generation of Sine Wave Using DSP Processor

Aim

To generate a sine wave using DSP processor

Theory

Sinusoidal are the most smooth 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 increase 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 return to 0 at 360° . After any angle greater than 360° , the sinusoidal signal repeats the values so we can say that time 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 time period or angle value of 2π hence periodicity of sinusoidal signal is 2π.

These are sinusoidal signal parameters:

- **Graph:** It is a plot used to depict the relation between quantities. Depending upon the number of variables, we can decide to number of axes each perpendicular to the other.
- **Time period: The** period for a signal can be defined as the time taken by a periodic signal to complete one cycle.
- **Amplitude:** Amplitude can be defined as the maximum distance between the horizontal axis and the vertical position of any signal.
- **Frequency:** This can be defined as the number of times a signal oscillates in one second. It can be mathematically defined as the reciprocal of a period.
- **Phase:** It can be defined as the horizontal position of a waveform in one oscillation. The symbol θ is used to indicate the phase.

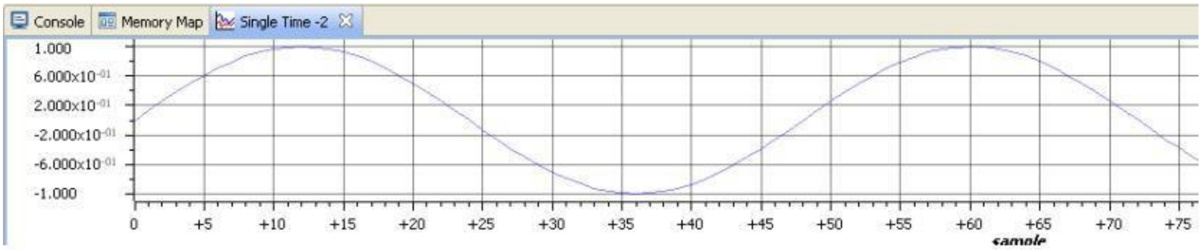
If we consider a sinusoidal signal y(t) having an amplitude A, frequency f, and phase of quantity then we can represent the signal as

$$y(t) = A\, sin(2\pi f t + \theta)$$

If we denote 2πf as an angular frequency ω the we can re-write the signal as

$$y(t) = A\, sin(\omega t + \theta\ )$$

Output


Console
Memory Map
Single Time -2
1.000
6.000x10-01
2.000x10-01
-2.000x10-01
-6.000x10-01
-1.000
0
+5
+10
+15
+20
+25
+30
+35
+40
+45
+50
+55
+60
+65
+70
+75
sample

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,a).

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
3.1415625         float
a[200]; main()
{ int i=0; for(i=0;i<200;i++)
    a[i]=sin(2*pi*5*i/200);
}
```

RESULT

Generated a sine wave using DSP processor

Experiment : 12 Date:19/10/2024

Linear Convolution using DSP Processor

Aim

To perform linear convolution of two sequences using DSP processor.

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:

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

where:

- x[n] is the input signal.
- h[n] is the impulse response of the system.
- y[n] is the output signal.

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.

Applications of Linear Convolution :

- Filtering: Used in digital filters to process signals.
- Image Processing: Applied for edge detection and blurring.
- System Analysis: Helps in analyzing LTI systems in response to inputs.

Output

Linear Convolution Output

```
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
```

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. In the Debug perspective, click Resume to run the code on DSP. Observe the console output to verify the convolution result.

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 of two sequences using DSP processor.