THANGAL KUNJU MUSALIAR COLLEGE OF ENGINEERING

KOLLAM – 691 005

ELECTRONICS AND COMMUNICATION ENGINEERING

22ECL509

DIGITAL SIGNAL PROCESSING LAB

LABORATORY RECORD

YEAR 2024-25

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. **SALMANUL FARIS PV** *of* **5th** *Semester class (Roll No.* **B22ECB60** ***Electronics and Communication*** *Branch) in the* ***Digital Signal Processing*** *Laboratory during the year* ***2024-25.***

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

Register Number : **TKM22EC116**

Staff Member in-charge *External Examiner*

Date:

INDEX

SL No.	DATE	NAME OF THE EXPERIMENTS	PAGE NO.	REMARKS
1.	30/07/2024	Simulation of Basic Test Signals		
2.	30/07/2024	Verification of Sampling Theorem		
3.	06/08/2024	Linear Convolution		
4.	06/08/2024	Circular Convolution		
5.	13/08/2024	Linear Convolution using circular convolution and vice versa		
6.	01/09/2024	DFT and IDFT		
7.	15/09/2024	Properties of DFT		
8.	08/10/2024	Overlap Add and Overlap Save Method		
9.	15/10/2024	Implementation of FIR Filter		
10.	22/10/2024	Familiarization of DSP Hardware		
11.	29/10/2024	Generation of sine wave using DSP Kit		
12.	29/10/2024	Linear Convolution using DSP Kit		

Experiment No: 1 Date: 30-07-2024

Simulation of Basic Test Signals

Aim : To generate basic test signals

a) unit impulse

b) unit step

c) ramp signal

d) sine wave

e) cosine wave

f) unipolar pulse

g) bipolar pulse

h) exponential

i) triangular

Theory

In signal processing experiments, we need the help of some fundamental signals such as sine wave, square wave, ramp wave, unit step, unit impulse etc. A digital signal can be either a deterministic signal that can be predicted with certainty, or a random signal that is unpredictable. Due to ease in signal generation and need for predictability, deterministic signals can be used for system simulation studies. Standard forms of some deterministic signals that are frequently used in DSP are discussed below

1. Impulse: The simplest signal is the unit impulse signal which is defined as,

$\delta(n) = 1$ for $n = 0$

$= 0$ for $n \neq 0$

2. Step: The general form of step function is,

$u(n) = 1$ for $n \geq 0$

$= 0$ for $n < 0$

3. Exponential: The decaying exponential is a basic signal in DSP whose general form is,

$x(n) = a^n$ for all n.

4. Ramp: This signal is given by,

$r(n) = n$ for $n \geq 0$

$= 0$ for $n < 0$

5. Sine: A sinusoidal sequence is defined as,

x(n) = sin(n)

A continuous time signal is defined for all values of time t. Standard forms of some basic continuous time signals frequently used in DSP are discussed below:

1. Impulse: An impulse signal is given by,

$\int_{-\infty}^{\infty} \delta(t) = 1$ for t = 0

= 0 for t ≠ 0

2. Sine: A sinusoidal sequence is defined as,

x(t) = sin(t)

3. Unit step signal: The general form of step signal is,

u (t) = A for t ≥ 0

= 0 for t< 0

Where, A is the amplitude. If A=1, then it is a unit step signal.

4. Ramp: Ramp signal is given by,

r(t) = t for t ≥ 0

= 0 for t< 0

5. Exponential : Exponential signal is given by

x(t) = e^at

a>0 for growing exponential, a<0 for decaying exponential

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);
```

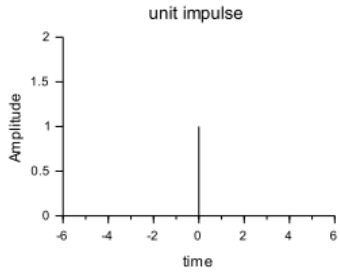
```
xlabel("Time(s)");
ylabel("Amplitude");
title("Sine Wave");
legend("Discrete","Continuous");
subplot(3,3,7);
t7 = 0:0.001:1;
f = 10;
y7 = cos(2*pi*f*t7);
stem(t7,y7);
hold on;
plot(t7,y7);
xlabel("Time(s)");
ylabel("Amplitude");
title("Cosine Wave");
legend("Discrete","Continuous");
subplot(3,3,8);
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);
```

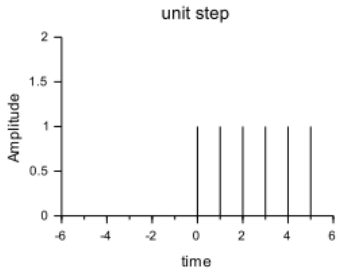
```
t9 = -5:1:5;

y9 = [zeros(1,5),ones(1,6)];

stem(t9,y9);

xlabel("Time(s)");

ylabel("Amplitude");

title("Unit Step Signal");
```

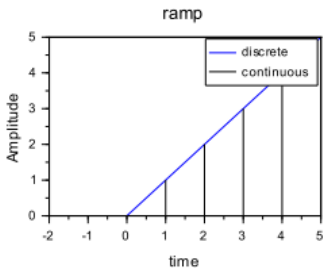
Result

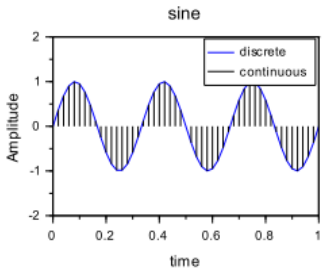
Simulated basic test signals in MATLAB.

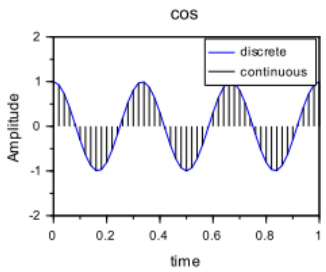
Observation

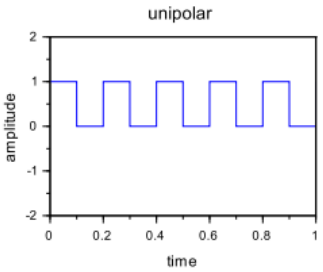


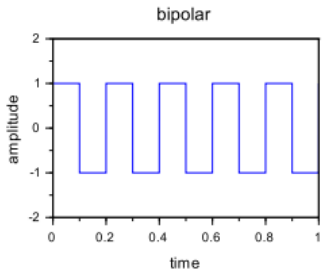


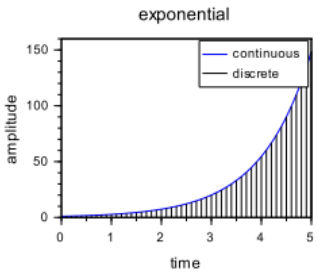


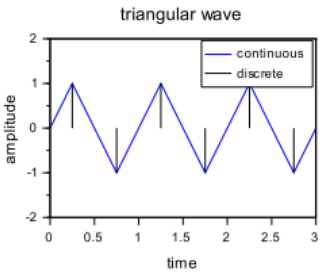












Experiment No: 2 Date: 30-07-2024

Verification of Sampling Theorem

Aim : To verify sampling theorem.

Theory

Sampling Theorem: A band limited signal can be reconstructed exactly if it is sampled at a rate at least twice the maximum frequency component in it. Figure 1 shows the spectrum of a signal g(t) that is band limited. The maximum frequency component of g(t) is fm. To recover the signal g(t) exactly from its samples it has to be sampled at a rate fs=2fm. The minimum required sampling rate fs = 2fm is called Nyquist rate.

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);
```

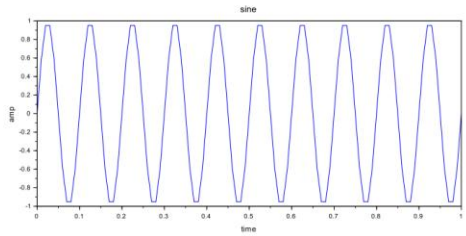
```
stem(t1,y1);
hold on;
plot(t1,y1);
grid(true);
xlabel("Time");
ylabel("Amplitude");
title("Under Sampled Signal");
subplot(2,2,3);
fs2= 3*f; %undersampled
t3 = 0:1/fs2:1;
y2 = sin(2*pi*f*t3);
stem(t3,y2);
hold on;
plot(t3,y2);
xgrid(true);
xlabel("Time");
ylabel("Amplitude");
legend("Discrete","Continuous")
title("Nyquist Sampled Signal");
subplot(2,2,4);
fs2= 100*f; %undersampled
t3 = 0:1/fs2:1;
y2 = sin(2*pi*f*t3);
stem(t3,y2);
hold on;
plot(t3,y2);
```

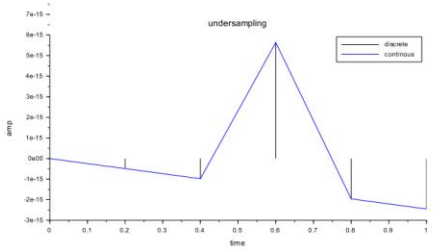
```
grid(true);
xlabel("Time");
ylabel("Amplitude");
legend("Discrete","Continuous")
title("Over Sampled Signal");
```

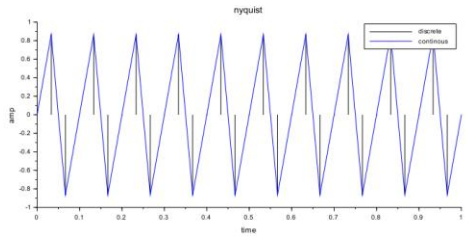
Result

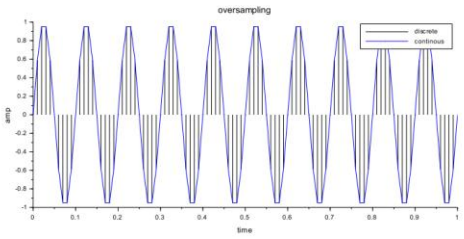
Verified sampling theorem.

Observation









Experiment No: 3 Date: 06-08-2024

Linear Convolution

Aim : a) Write a MATLAB program to perform linear convolution using inbuilt function

b) Write a MATLAB program to perform linear convolution without using inbuilt function

Theory

Convolution is used to find the output response of a digital system. The linear convolution of two continuous time signals x[n] and h[n] is defined by

y[n] = x[n] * h[n].

The length of the output sequence y[n]= length of x[n]+length of h[n]-1.

Circular Convolution of two sequences x[n] and y[n], each of length N is given by ,

p[n] = x[n]©y[n].

If the length of sequence is not equal, zero padding is done to get the maximum length among the two. The resulting convolved signal would be zero outside the range n= 0,1,...,N-1.

In order to find the circular convolution of two given sequences without using inbuilt function, we make use of the circular convolution property of DFT.

If X(k) = DFT [x(n)], Y(k) = DFT[y(n)], then convolution property states that,

DFT[x (n)©y (n)] = X (k)Y(k)

Program

```
//with conv()

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");
```

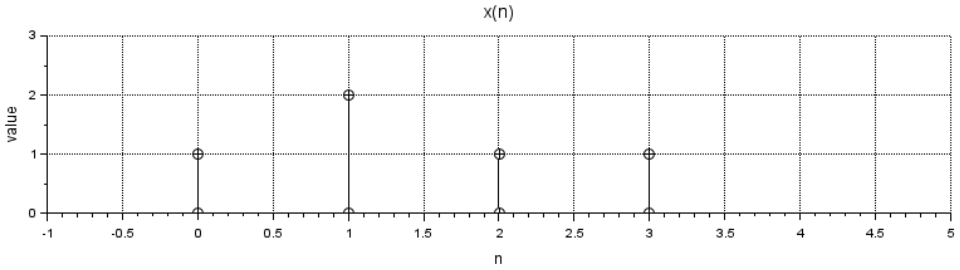
```
title("y(n)");
grid on
xlim([-1 k+1]);
ylim([0 max(y1)+2]);
//without using conv()
clc;
clear all;
close all;
x1 = input("Enter first Sequence");
h1 = input("Enter second Sequence");
l = length(x1);
m = length(h1);
k = l+m-1;
y1 = zeros(1,k);
for i=1:l
for j=1:m
y1(i+j-1) = y1(i+j-1) + x1(i)*h1(j);
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");
```

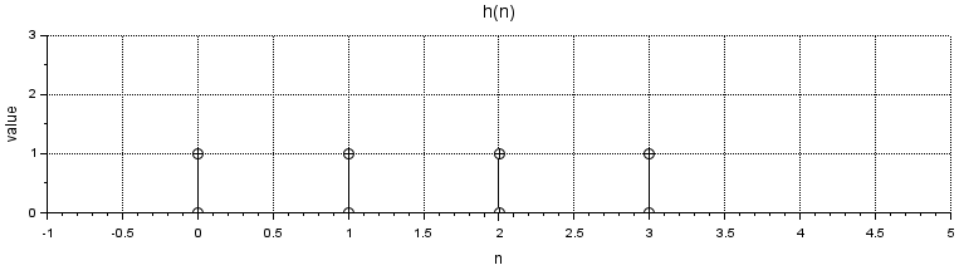
```
xlabel("n");
ylabel("Amplitude");
title("x(n)");
grid on
xlim([-1 l+1]);
ylim([0 max(x1)+2]);
subplot(1,3,2);
stem(n2,h1,"o");
xlabel("n");
ylabel("Amplitude");
title("h(n)");
grid on
xlim([-1 m+1]);
ylim([0 max(h1)+2]);
subplot(1,3,3);
stem(n3,y1,"o");
xlabel("n");
ylabel("Amplitude");
title("y(n)");
grid on
xlim([-1 k+1]);
ylim([0 max(y1)+2]);
```

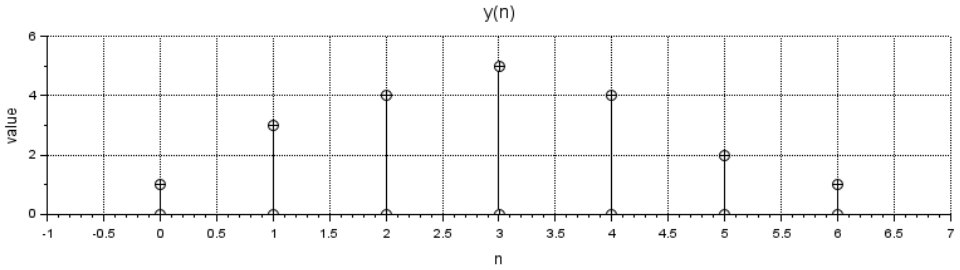
Result

Performed linear convolution with and without using inbuilt function.

Observation







Experiment No: 4 Date: 06-08-2024

Circular Convolution

Aim : Write a MATLAB program to perform circular convolution using

a) concentric circle method
b) matrix method
c) discrete fourier transform

Theory

Convolution is used to find the output response of a digital system. The linear convolution of two continuous time signals x[n] and h[n] is defined by

y[n] = x[n] * h[n].

The length of the output sequence y[n]= length of x[n]+length of h[n]-1.

Circular Convolution of two sequences x[n] and y[n], each of length N is given by ,

p[n] = x[n]©y[n].

If the length of sequence is not equal, zero padding is done to get the maximum length among the two. The resulting convolved signal would be zero outside the range n= 0,1,...,N-1.

In order to find the circular convolution of two given sequences without using inbuilt function, we make use of the circular convolution property of DFT.

If X(k) = DFT [x(n)], Y(k) = DFT[y(n)], then convolution property states that,

DFT[x (n)©y (n)] = X (k)Y(k)

Program

a) 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(x);
disp(h);
disp(y);
```

b) matrix method

```
clc;
clear all;
close all;
% Define the two sequences
x = [1 2 1 2]; % First sequence
h = [1 2 3 4]; % Second sequence
% Length of the sequences
N = length(x);
% Construct the circulant matrix for sequence 'x'
% The first column is [x(1), x(N), x(N-1), ..., x(2)]
% The first row is [x(1), x(2), ..., x(N)]
X_circulant = toeplitz([x(1), fliplr(x(2:end))], x);
% Perform circular convolution by matrix multiplication
y = X_circulant * h';
% Display the result
disp(x);
disp(h);
disp(y);
```

c) using dft

```
clc;

close all;

clear 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(x);

disp(h);

disp(y1);
```

Result

Performed circular convolution using concentric method, matrix method and dft.

Observation

a) concentric circle method

x = [1, 2, 3]

h = [4, 5, 6, 7, 8]

y = [40, 41, 37, 28, 34]

b) matrix method

x = [1, 2, 3]

h = [4, 5, 6, 7, 8]

y = [41, 37, 28, 34, 40]

c) discrete fourier transform

x = [1, 2, 3]

h = [4, 5, 6, 7, 8]

y = [41, 37, 28, 34, 40]

Experiment No: 5 Date: 13-08-2024

Linear Convolution using Circular Convolution and vice versa

Aim : Write a MATLAB program to perform

a) linear using circular
b) circular using linear

Theory

a) Linear Convolution Using Circular Convolution

Linear convolution and circular convolution differ in how they handle boundary conditions. To perform linear convolution using circular convolution, you need to modify the circular convolution process to accommodate for the boundary effects.

Steps to Perform Linear Convolution Using Circular Convolution:

Zero-Padding:

Let the input sequences be x and h with lengths Lx and Lh, respectively.

Zero-pad both sequences to a length N = Lx + Lh - 1, which is the expected length of the linear convolution output.

Circular Convolution:

Compute the circular convolution of the zero-padded sequences using methods such as the Discrete Fourier Transform (DFT) or matrix methods.

Result:

The result of this circular convolution will now be equivalent to the linear convolution of the original sequences because of the padding.

b) Circular Convolution Using Linear Convolution

To compute circular convolution using linear convolution, we must ensure that the result fits within the size of the original sequences, without any wrapping of the output.

Steps to Perform Circular Convolution Using Linear Convolution:

Truncate or Wrap the Output:

Perform linear convolution on the input sequences x and h.

Truncate or wrap the output of the linear convolution to the length of the shorter sequence, typically N = max(Lx, Lh).

Modular Summation for Wrapping:

If the result exceeds the desired length, sum the overlapping values to get the final circular convolution output. This process wraps the linear convolution result back onto itself, as circular convolution would do.

Program

a) linear using circular

```
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(x);

disp(h);

disp(y);
```

b) circular using linear

```
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(x);
disp(h);
disp(y1);
```

Result

Performed linear convolution using circular convolution and vice versa.

Observation

a) linear using circular

x = [1 2 3 4];

h = [1 1 1];

y = [1 3 6 9 7 4];

b) circular using linear

x = [1 2 3]

h = [4 5 6 7 8]

y = [41 37 28 34 40]

Experiment No: 6 Date: 01-10-2024

Discrete Fourier transform and Inverse Discrete Fourier transform

Aim : Write a MATLAB program to perform

a) DFT and IDFT without using built-in function
b) DFT and IDFT using twiddle factor
c) DFT – magnitude and phase plots

Theory

a) DFT and IDFT without using built-in function

Discrete Fourier Transform is the transformation used to represent the finite duration frequencies. The DFT of a discrete sequence x(n) is obtained by performing sampling operations in both the time domain and the frequency domain. It is the frequency domain representation of a discrete digital signal.

The DFT of a sequence x(n) of length N is given by the following equation:

$$X(k) = \left\{ \sum_{n=0}^{N-1} x(n)\, e^{\frac{-i2\pi kn}{N}} \;;\; 0 \le k \le N-1 \right\}$$

The Inverse Discrete Fourier Transform (IDFT) performs the reverse operation of the DFT to obtain the time-domain sequence x(n) from the frequency domain sequence X(k). The IDFT of the sequence is given as:

$$X(k) = \left\{ \frac{1}{N} \sum_{k=0}^{N-1} X(k)\, e^{\frac{i2\pi kn}{N}} \right\}$$

b) DFT and IDFT using twiddle factor

Discrete Fourier Transform (DFT**)** can be represented using the twiddle factor. The twiddle factor is denoted as:

$$W_N = e^{-j\frac{2\pi}{N}}$$

where W_N is a complex exponential that represents the rotation factor in the frequency domain. The DFT of a sequence x(n) of length N can be written as:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn},\ 0 \le k \le N-1$$

Inverse Discrete Fourier Transform (IDFT) performs the reverse operation of the DFT, transforming the frequency-domain sequence X(k) back into the time-domain sequence x(n). The IDFT uses the conjugate twiddle factor:

$$W_N^{-1} = e^{j\frac{2\pi}{N}}$$

The IDFT is given by:

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k)W_N^{-kn},\ 0 \le n \le N-1$$

In this expression, W_N^{-kn} is the conjugate of the twiddle factor, which transforms the frequency components back to the time domain.

Program

a) DFT and IDFT without using built-in function

i) DFT

```
clc;
clear all;
close all;
N=4;
x = [1 1 1];
L = length(x);
x = [x,zeros(1,N-L)];
X = zeros(N,1);
for k=0:N-1
    for n=0:N-1
            X(k+1)= X(k+1)+ x(n+1)*exp(-1i*2*pi*n*(k/N));
    end
end
%disp(X);
disp(round(X));
%disp(fft(x));
```

ii) IDFT

```
clc;
```

```
clear all;
close all;
N=4;
X = [3 -1i 1 1i];
%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(1i*2*pi*n*(k/N));
    end
end
x=(1/N).*x;
%disp(x);
disp(round(x));
%disp(ifft(X));
```

b) DFT and IDFT using twiddle factor

i) DFT

```
clc;
clear all;
close all;
N=4;
x = [1 1 1];
L = length(x);
x = [x,zeros(1,N-L)];
X = zeros(N,1);

T = zeros(N, N);
for k = 0:N-1
```

```
    for n = 0:N-1
        T(k+1, n+1) = exp(-1i * 2 * pi * k * n / N);
    end
end
X=T*x';
%disp(X);
disp(round(X));
%disp(fft(x));
```

ii) IDFT

```
clc;
clear all;
close all;
N=4;
X = [3 -1i 1 1i];
%L = length(X);
%X = [X,zeros(1,N-L)];
x = zeros(N,1);
T = zeros(N, N);
for k = 0:N-1
    for n = 0:N-1
        T(k+1, n+1) = exp(1i * 2 * pi * k * n / N);
    end
end
x=T*X';
x=(1/N).*x;
%disp(x);
disp(round(x));
%disp(ifft(X));
```

c) DFT – magnitude and phase plots

```
clc;
clear all;
close all;
N=4;
x = [1 1 1];
L = length(x);
x = [x,zeros(1,N-L)];
X = zeros(N,1);
for k=0:N-1
    for n=0:N-1
            X(k+1)= X(k+1)+ x(n+1)*exp(-1i*2*pi*n*(k/N));
    end
end
%disp(X);
%disp(round(X));
%disp(fft(x));
mg = abs(X);
ph = angle(X);
subplot(2,1,1);
stem(mg);
hold on;
plot(mg);
subplot(2,1,2);
stem(ph);
hold on;
plot(ph);
```

Result

Performed Discrete Fourier transform and Inverse Discrete Fourier transform without built-in function and using twiddle factor.

Observation

a) DFT and IDFT without using built-in function

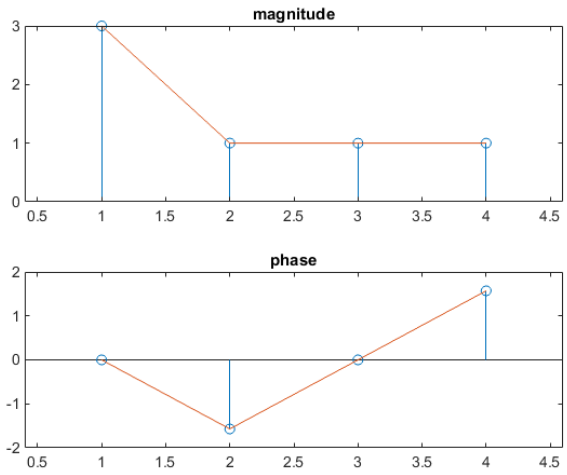
```
3.0000 + 0.0000i
0.0000 - 1.0000i
1.0000 + 0.0000i
0.0000 + 1.0000i
```

b) DFT and IDFT using twiddle factor

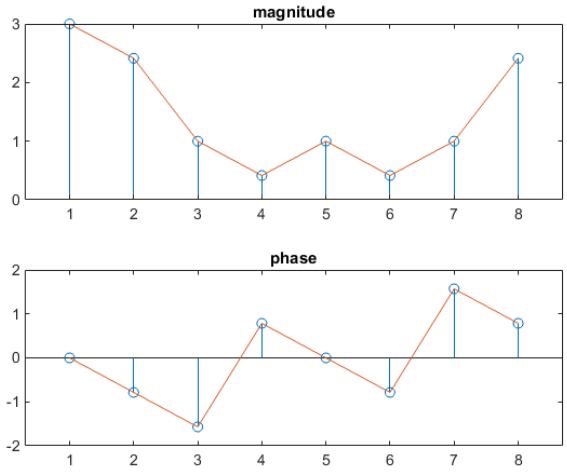
```
3.0000 + 0.0000i
0.0000 - 1.0000i
1.0000 + 0.0000i
0.0000 + 1.0000i
```

c) DFT – magnitude and phase plots

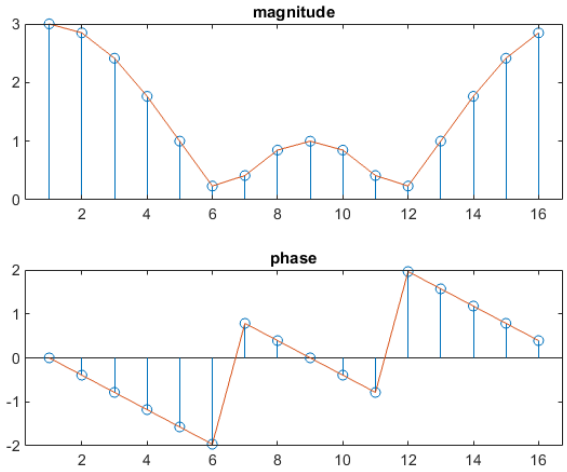
N=4



N=8



N=16



Experiment No: 7 Date: 01-10-2024

Properties Discrete Fourier transform

Aim : Write a MATLAB program to verify the properties of DFT

a) Linearity property
b) Parseval's theorem
c) Convolution property
d) Multiplication property

Theory

a) Linearity property

The DFT is a linear transformation, meaning that the DFT of a linear combination of signals is equal to the linear combination of their DFTs. Mathematically, if x1(n) and x2(n) are two sequences, and a and b are constants, then:

$$DFTax1(n) + bx2(n) = a \cdot DFTx1(n) + b \cdot DFTx2(n)$$

This means DFT can be distributed over the sum of two signals.

b) Parseval's theorem

Parseval's theorem relates the total energy of a sequence in the time domain to the total energy in the frequency domain. If x(n) is a sequence of length N, then the total energy is conserved, and it holds that:

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

c) Convolution property

The DFT converts the convolution of two sequences in the time domain into the pointwise multiplication of their DFTs in the frequency domain. This makes it useful for fast convolution calculations.

In time domain:

$$y(n) = (x * h)(n) = \sum_{m=0}^{N-1} x(m)h(n-m)$$

In frequency domain:

$$\text{DFT}\{y(n)\} = X(k) \cdot H(k)$$

d) Multiplication property

In contrast to the convolution theorem, multiplication in the time domain corresponds to convolution in the frequency domain.

If two sequences x(n) and h(n) are multiplied in the time domain, their DFTs are convolved in the frequency domain

$$y(n) = x(n) \cdot h(n)$$

Then, in the frequency domain:

$$\text{DFT}\{y(n)\} = \frac{1}{N}\big(X(k) * H(k)\big)$$

Program

a) Linearity property

```
x1 = [1 2 3];
x2 = [1 2 1 2];
L = max(length(x1),length(x2));
x1 = [x1, zeros(1,L-length(x1))];
x2 = [x2, zeros(1,L-length(x2))];
a=2;
b=3;
x3=(a).*x1 + (b).*x2;
lhs=fft(x3);
X1 = fft(x1);
X2 = fft(x2);
rhs = a.*X1 + b.*X2;
disp(lhs);
disp(rhs);
```

b) Parseval's theorem

```
x = [-1 2 -3];
X = fft(x);
lhs = sum(abs(x.*x));
N=length(x);
rhs = sum(abs(X.*X));
rhs = (1/N)*rhs;
disp(lhs);
disp(rhs);
```

c) Convolution property

```
x1 = [1 2 3];
x2 = [1 2 1 2];
L = max(length(x1),length(x2));
x1 = [x1, zeros(1,L-length(x1))];
x2 = [x2, zeros(1,L-length(x2))];
lhs = cconv(x1,x2,L);
lhs = fft(lhs);
X1=fft(x1);
X2=fft(x2);
rhs = X1.*X2;
disp(lhs);
disp(rhs);
```

d) Multiplication property

```
x1 = [1 2 3];
x2 = [1 2 1 2];
L = max(length(x1),length(x2));
x1 = [x1, zeros(1,L-length(x1))];
x2 = [x2, zeros(1,L-length(x2))];
```

```
lhs=fft(x1.*x2);
X1=fft(x1);
X2=fft(x2);
rhs=cconv(X1,X2,L);
rhs=(1/L)*rhs;
disp(lhs);
disp(rhs);
```

Result

Verified the properties of DFT

Observation

a) Linearity property

```
30.0000 + 0.0000i  -4.0000 - 4.0000i  -2.0000 + 0.0000i  -4.0000
+ 4.0000i

30.0000 + 0.0000i  -4.0000 - 4.0000i  -2.0000 + 0.0000i  -4.0000
+ 4.0000i
```

b) Parseval's theorem

```
14

14.0000
```

c) Convolution property

```
36     0    -4     0

36     0    -4     0
```

d) Multiplication property

```
8.0000 + 0.0000i  -2.0000 - 4.0000i   0.0000 + 0.0000i  -2.0000 +
4.0000i
8.0000 + 0.0000i  -2.0000 - 4.0000i   0.0000 + 0.0000i  -2.0000 +
4.0000i
```

Experiment No: 8 Date: 08-10-2024

Overlap-Add and Overlap-Save

Aim : Write a MATLAB program to perform overlap-add and overlap-save methods

a) Overlap-add
b) Overlap-save

Theory

Generally, in discrete-time processing we talk about linearly convolving a sequence with a FIR filter. In contrast, the FFT performs circular convolution with a filter of equal or lesser length. In computing the DFT or FFT we often have to make linear convolution behave like circular convolution or vice versa. Two methods that make linear convolution look like circular convolution are overlap-add and overlap-save.

a) Overlap-add

This procedure cuts the signal up into equal length segments with no overlap. Then it zero-pads the segments and takes the DFT of the segments. Part of the convolution result corresponds to the circular convolution. The tails that do not correspond to the circular convolution are added to the adjoining tail of the previous and subsequent sequence. This addition results in the aliasing that occurs in circular convolution.

b) Overlap-save

This procedure cuts the signal up into equal length segments with some overlap. Then it takes the DFT of the segments and saves the parts of the convolution that correspond to the circular convolution. Because there are overlapping sections, it is like the input is copied therefore there is not lost information in throwing away parts of the linear convolution.

Program

a) Overlap-add

```
x=[1 2 3 4 5 6 7 8 9];
p=length(x);
h=[1 2 1];
q=length(h);
cc=cconv(x,h);
r=0;
```

```
v=ceil(p/q);
while r ~= 0
    p = p + 1;
    r = mod(p,q);
    if r==0
        break
    end
end
x=[x zeros(1,p)];
Lx=length(x);
Lh=length(h);
Lb = Lh; %Lb=blockLength
a=1;
b=Lb;
x=[x zeros(1,Lb)];
h=[h zeros(1,Lb-1)];
y1=[zeros(1,2*Lb-1)];
y=[];
for i =1:v+1
    x1=x(a:b);
    a=a+Lb;
    b=b+Lb;
    x1=[x1 zeros(1,Lb-1)];
    y2=cconv(x1,h,2*Lb-1);
    y3=y1+y2;
    y=[y y3(1:Lb)];
    y1=[y2(Lb+1:end) zeros(1,Lb)];
end
disp(cc);
disp(y);
```

b) Overlap-save

```
x=[1 2 3 4 5 6 7 8 9];
p=length(x);
h=[1 2 1];
q=length(h);
cc=cconv(x,h);
r=0;
v=ceil(p/q);
while r ~= 0
    p = p + 1;
    r = mod(p,q);
    if r==0
        break
    end
end
x=[x zeros(1,p)];
Lx=length(x);
Lh=length(h);
Lb = Lh; %Lb=blockLength
a=1;
b=Lb;
x=[x zeros(1,Lb)];
h=[h zeros(1,Lb-1)];
y1=[zeros(1,2*Lb-1)];
y=[];
for i =1:v+1
    x1=x(a:b);
    a=a+Lb;
```

```
    b=b+Lb;
    y2=[y1(Lb+1:end) x1];
    y3=cconv(y2,h,2*Lb-1);
    y=[y y3(Lb:end)];
    y1=y2;
end
disp(cc);
disp(y);
```

Result

Performed overlap-add and-overlap save methods.

Observation

a) Overlap-add

```
1.0000    4.0000    8.0000   12.0000   16.0000   20.0000   24.0000
28.0000   32.0000   26.0000    9.0000
```

```
1.0000    4.0000    8.0000   12.0000   16.0000   20.0000   24.0000
28.0000   32.0000   26.0000    9.0000         0
```

b) Overlap-save

```
1.0000    4.0000    8.0000   12.0000   16.0000   20.0000   24.0000
28.0000   32.0000   26.0000    9.0000
```

```
 1.0000    4.0000    8.0000   12.0000   16.0000   20.0000   24.0000
28.0000   32.0000   26.0000    9.0000    0.0000
```

Experiment No: 9 Date: 15-10-2024

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:

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

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

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

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

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;

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

%BPFhamming

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');

%BPFhanning

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');
%BPFrect
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');
%BPFtri
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

```
wcl= 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(wcl*(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);
```

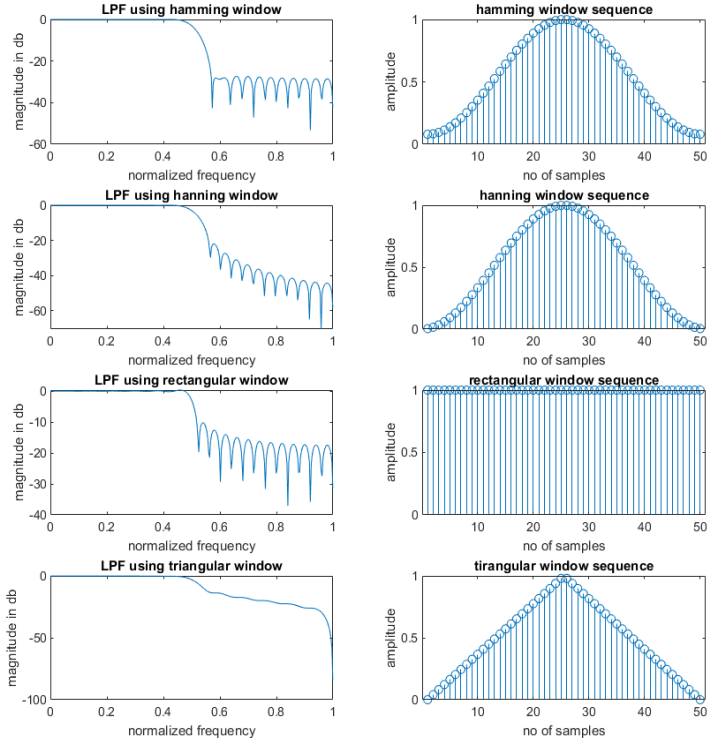
```
stem(w1);
title('hamming window sequence');
xlabel('no of samples');
ylabel('amplitude');
%hanning
subplot(4,2,4);
stem(w2);
title('hanning window sequence');
xlabel('no of samples');
ylabel('amplitude');
%rectangular
subplot(4,2,6);
stem(w3);
title('rectangular window sequence');
xlabel('no of samples');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

Result

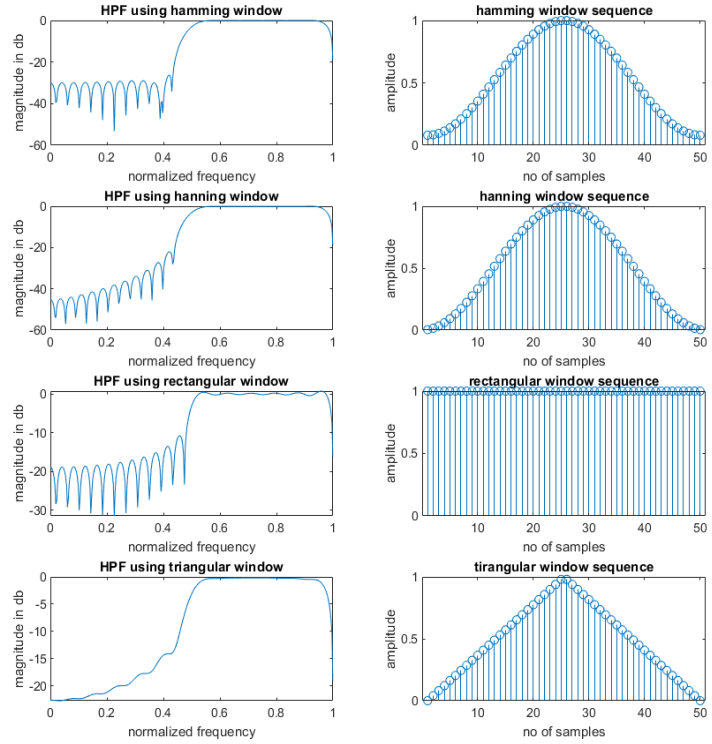
Implemented FIR filters using Window method.

Observation

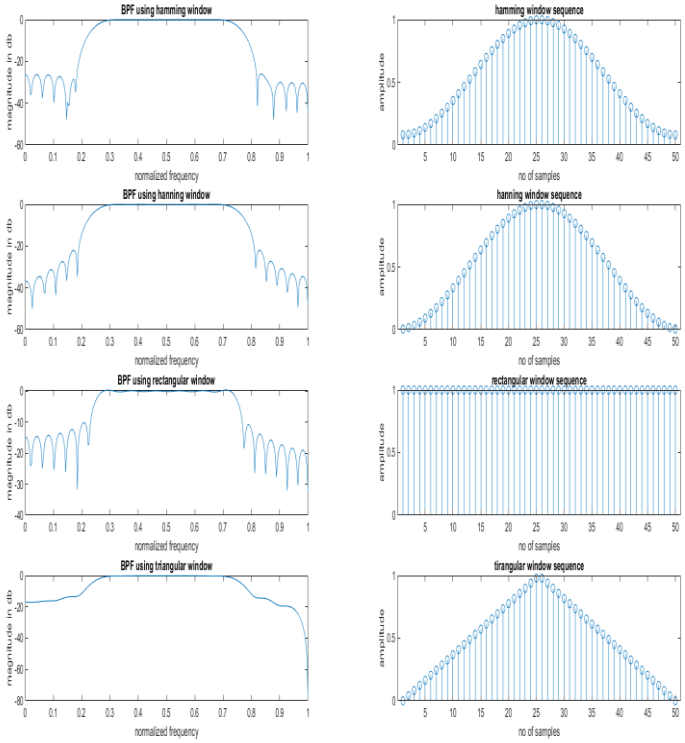
a) Low Pass Filter



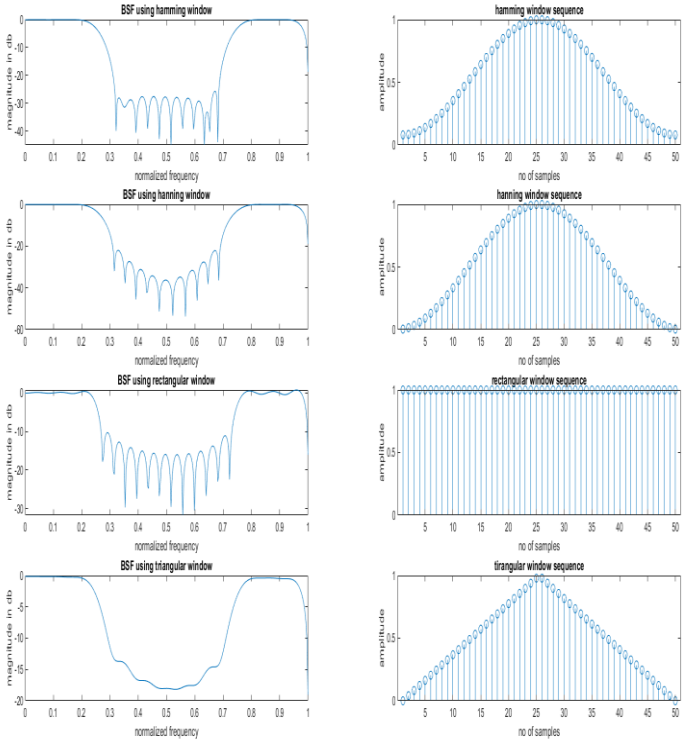
b) High Pass Filter



c) Band Pass Filter



d) Band Stop Filter



Experiment No: 10 Date: 22/09/2024

Familiarization of DSP Hardware

Aim

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

Theory

TMS 320C674x DSP CPU

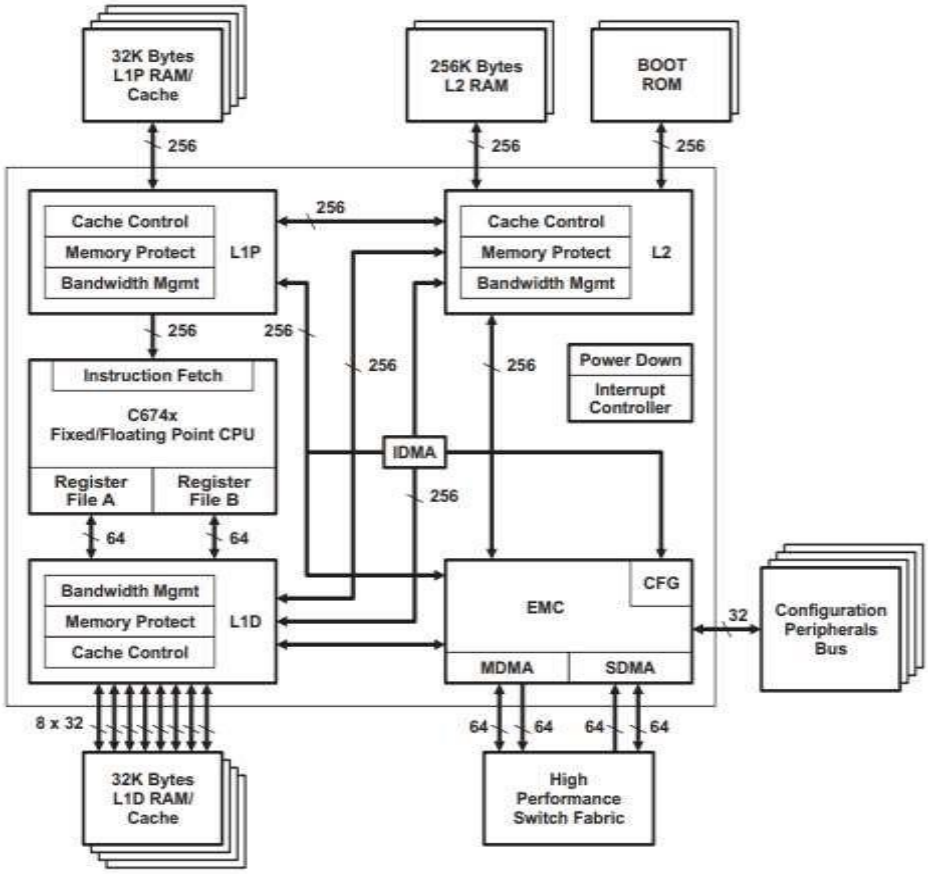


FIGURE: TMS320C 674X DSP CPU BLOCK DIAGRAM

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.

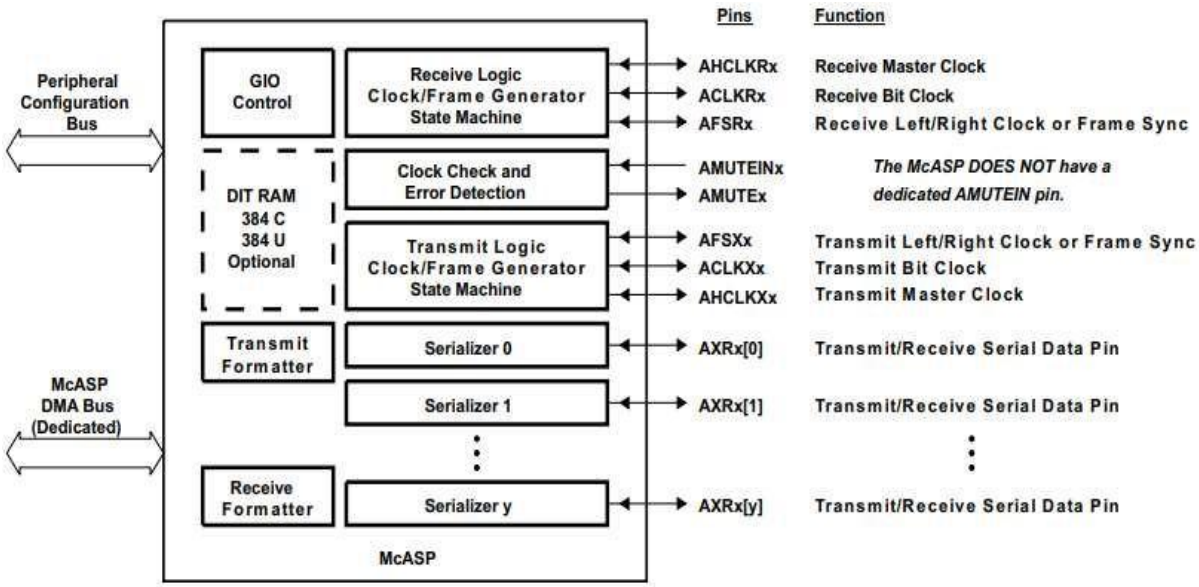
Multichannel Audio Serial Port (McASP):

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.

Experiment No: 11 Date: 29-10-2024

Generation of Sine Wave using DSP Kit

Aim: To generate a sine wave using DSP Kit

Theory

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

Procedure

1. Open Code Composer Studio, Click on File - New – CCS Project
Select the Target – C674X Floating point DSP , TMS320C6748 , and
Connection – Texas Instruments XDS 100v2 USB Debug Probe and Verify.
Give the project name and select Finish.

2. Type the code program for generating the sine wave and choose
File – Save As and then save the program with a name including ‘main.c’.
Delete the already existing main.c program.

3. Select Debug and once finished, select the Run option.

4. From the Tools Bar, select Graphs – Single Time.
Select the DSP Data Type as 32-bit Floating point and time display unit as second(s).
Change the Start address with the array name used in the program(here,s).

5. Click OK to apply the settings and Run the program or clock Resume in CCS.

Program

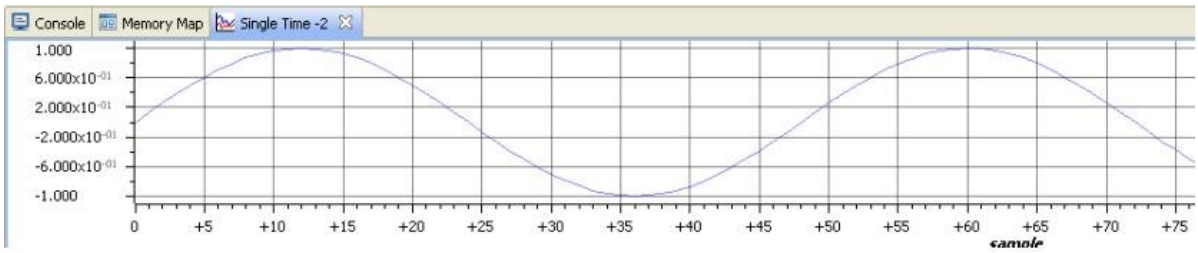
Sine wave

```
#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



Experiment No: 12 Date: 29-10-2024

Linear Convolution using DSP Kit

Aim: To perform linear convolution of two sequences using DSP Kit.

Theory

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

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

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

Procedure

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

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

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

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

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

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

Program

```
//#include<fastmath67x.h>

#include<math.h>

void main()

{

int *Xn,*Hn,*Output;

int *XnLength,*HnLength;

int i,k,n,l,m;

Xn=(int *)0x80010000; //input x(n)

Hn=(int *)0x80011000; //input h(n)

XnLength=(int *)0x80012000; //x(n) length

HnLength=(int *)0x80012004; //h(n) length

Output=(int *)0x80013000; // output address

l=*XnLength; // copy x(n) from memory address to variable l

m=*HnLength; // copy h(n) from memory address to variable m
```

```
for(i=0;i<(l+m-1);i++) // memory clear
{
Output[i]=0; // o/p array
Xn[l+i]=0; // i/p array
Hn[m+i]=0; // i/p array
}
for(n=0;n<(l+m-1);n++)
{
for(k=0;k<=n;k++)
{
Output[n] =Output[n] + (Xn[k]*Hn[n-k]); // convolution operation.
}
}
}
```

Result

Performed Linear Convolution using DSP Kit.

Observation

Xn

0x80010000 - 1

0x80010004 - 2

0x80010008 - 3

Hn

0x80011000 - 1

0x80011004 - 2

XnLength

0x80012000 - 3

HnLength

0x80012004 - 2

Output

0x80013000 - 1

0x80013004 - 4

0x80013008 - 7

0x8001300C - 6