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. Dixon Mathews of ***5th*** *Semester class (Roll No.* ***B22ECB29 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 : ***TKM22EC046***

Staff Member in-charge *External Examiner*

Date:

INDEX

Experiment No: 1 Date: 25/07/2024

SIMULATION OF BASIC TEST SIGNALS

AIM: To generate continuous and discrete waveforms of the signals

1. Unit step signal
2. Unit impulse signal
3. Ramp signal
4. Sine wave
5. Square wave - Unipolar
6. Square wave - Bipolar
7. Sawtooth wave
8. Exponential signal

THEORY

1. Unit Step Signal

A signal that changes from 0 to 1 at a specified time (usually at time t = 0).

Mathematical representation:

$$u(t) = \{0;\ t < 0\ \&\ 1;\ t \geq 0\}$$

2. Unit Impulse Signal

A signal that is zero everywhere except at t = 0, where it has an infinite value such that its integral over time is 1.

Mathematical representation:

$$\delta(t) = \{\infty;\ t = 0\ \&\ 0;\ t \neq 0\}$$

3. Ramp Signal

A signal that increases linearly over time.

Mathematical representation:

$$r(t) = \{t;\ t \geq 0\ \&\ 0;\ t < 0\}$$

4. Sine Wave

A periodic continuous signal that oscillates smoothly between positive and negative values.

Mathematical representation:

$$x(t) = A\sin(\omega t + \phi)$$

where A = Amplitude of the sine wave, $\omega = Angular\ frequency\ of\ the\ sine\ wave, \phi = Phase\ of\ the\ sine\ wave$

5. **Square wave - Unipolar**

A square wave that oscillates between 0 and a positive value, typically 0 and +A.

Mathematical representation:

$$x(t) = \{A;\ 0 < t < T/2\ \&\ 0;\ T/2 <\ t < T\ \}$$

6. **Square wave - Bipolar**

A periodic signal that alternates between two levels (e.g., +1 and -1) at a constant frequency.

Mathematical representation:

$$x(t)=\{+A/2;\ 0 < t < T/2\ \&\ -A/2;\ T/2 < t < T\}$$

7. **Sawtooth wave**

A periodic waveform that ramps upward linearly and then sharply drops, resembling the teeth of a saw.

Mathematical representation:

$$x(t) = (A/T)t,\ t\ \epsilon[0,T]$$

8. **Exponential signal**

A signal that grows or decays exponentially over time.

Mathematical representation:

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

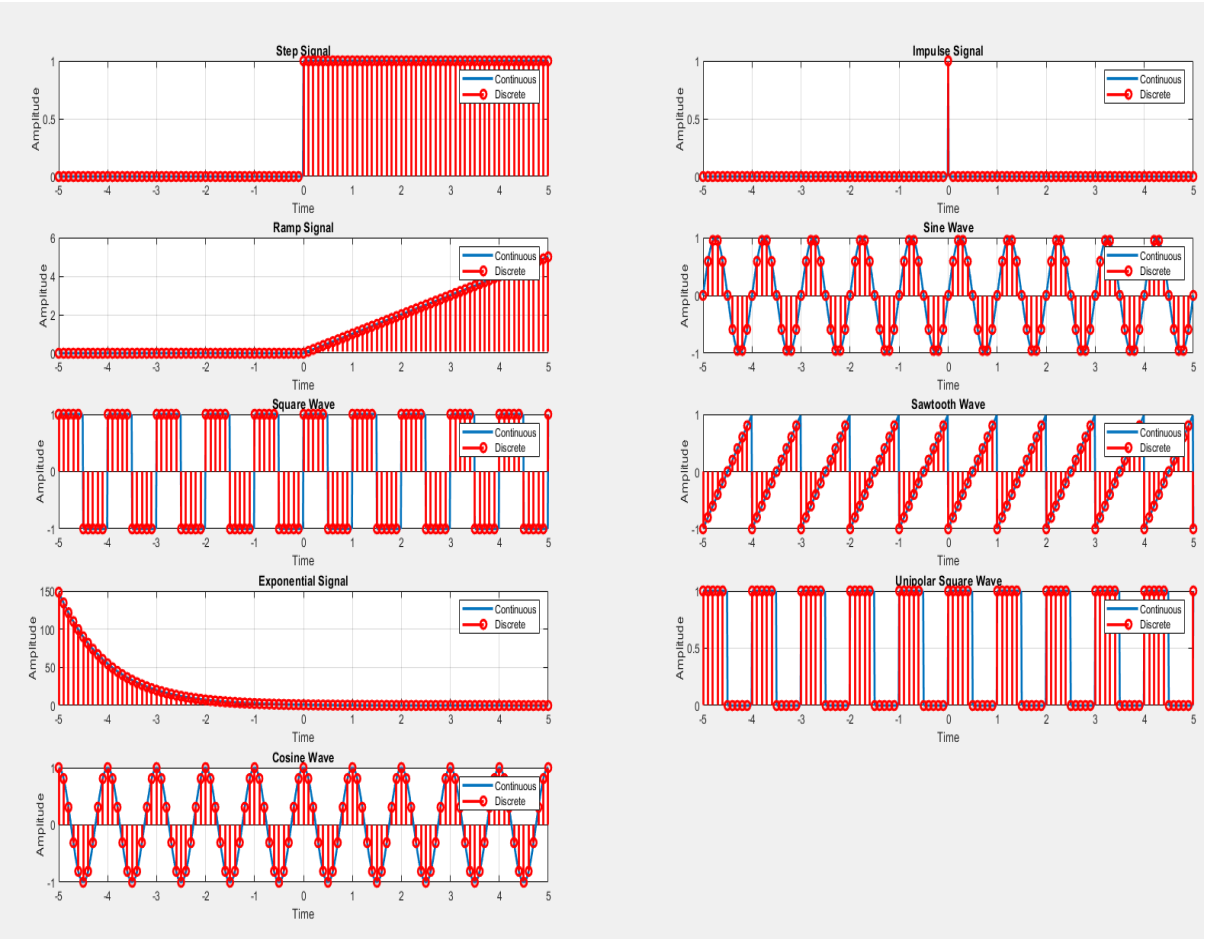
where A = initial amplitude of the signal, $\alpha = Decay\ rate\ of\ the\ signal$

PROGRAM

```
clc;clear; t = -5:0.01:5;    % Continuous time vector
n = -5:0.1:5;       % Discrete time vector
figure;
%% 1. Step Signal
subplot(4,2,1);
step_signal_continuous = t >= 0;
step_signal_discrete = n >= 0;
plot(t, step_signal_continuous, 'LineWidth', 2);
hold on;
stem(n, step_signal_discrete, 'r', 'LineWidth', 2);
title('Step Signal');
xlabel('Time');
ylabel('Amplitude');
legend('Continuous', 'Discrete');
grid on;
hold off;
```

```
%% 2. Impulse Signal
subplot(4,2,2);
impulse_signal_continuous = t == 0; % Approximation for continuous
impulse
impulse_signal_discrete = n == 0;
plot(t, impulse_signal_continuous, 'LineWidth', 2);
hold on;
stem(n, impulse_signal_discrete, 'r', 'LineWidth', 2);
title('Impulse Signal');
xlabel('Time');
ylabel('Amplitude');
legend('Continuous', 'Discrete');
grid on;
hold off;
%% 3. Ramp Signal
subplot(4,2,3);
ramp_signal_continuous = t .* (t >= 0);
ramp_signal_discrete = n .* (n >= 0);
plot(t, ramp_signal_continuous, 'LineWidth', 2);
hold on;
stem(n, ramp_signal_discrete, 'r', 'LineWidth', 2);
title('Ramp Signal');
xlabel('Time');
ylabel('Amplitude');
legend('Continuous', 'Discrete');
grid on;
hold off;
%% 4. Sine Wave
subplot(4,2,4);
sine_wave_continuous = sin(2*pi*t);
sine_wave_discrete = sin(2*pi*n);
plot(t, sine_wave_continuous, 'LineWidth', 2);
hold on;
stem(n, sine_wave_discrete, 'r', 'LineWidth', 2);
title('Sine Wave');
xlabel('Time');
ylabel('Amplitude');
legend('Continuous', 'Discrete');
grid on;
hold off;
%% 5. Square Wave
subplot(4,2,5);
square_wave_continuous = square(2*pi*t);
square_wave_discrete = square(2*pi*n);
plot(t, square_wave_continuous, 'LineWidth', 2);
```

OUTPUT


Step Signal
Impulse Signal
Ramp Signal
Sine Wave
Square Wave
Sawtooth Wave
Exponential Signal
Unipolar Square Wave
Cosine Wave
Continuous
Discrete
Amplitude
Time

```
hold on;
stem(n, square_wave_discrete, 'r', 'LineWidth', 2);
title('Square Wave');
xlabel('Time');
ylabel('Amplitude');
legend('Continuous', 'Discrete');
grid on;
hold off;
%% 6. Sawtooth Wave
subplot(4,2,6);
sawtooth_wave_continuous = sawtooth(2*pi*t);
sawtooth_wave_discrete = sawtooth(2*pi*n);
plot(t, sawtooth_wave_continuous, 'LineWidth', 2);
hold on;
stem(n, sawtooth_wave_discrete, 'r', 'LineWidth', 2);
title('Sawtooth Wave');
xlabel('Time');
ylabel('Amplitude');
legend('Continuous', 'Discrete');
grid on;
hold off;
%% 7. Exponential Signal
subplot(4,2,7);
exp_signal_continuous = exp(-t);  % Decaying exponential
exp_signal_discrete = exp(-n);    % Decaying discrete exponential
plot(t, exp_signal_continuous, 'LineWidth', 2);
hold on;
stem(n, exp_signal_discrete, 'r', 'LineWidth', 2);
title('Exponential Signal');
xlabel('Time');
ylabel('Amplitude');
legend('Continuous', 'Discrete');
grid on;
hold off;
%% 8. Unipolar Square Wave
subplot(5,2,8);
unipolar_wave_continuous = 0.5*(square_wave_continuous + 1);
unipolar_wave_discrete = 0.5*(square_wave_discrete + 1);
plot(t, unipolar_wave_continuous, 'LineWidth', 2);
hold on;
stem(n, unipolar_wave_discrete, 'r', 'LineWidth', 2);
title('Unipolar Square Wave');
xlabel('Time');
ylabel('Amplitude');
legend('Continuous', 'Discrete');
grid on;
```

```
hold off;

%% 9. Cosine Wave
subplot(5,2,9);
cosine_wave_continuous = cos(2*pi*t);
cosine_wave_discrete = cos(2*pi*n);
plot(t, cosine_wave_continuous, 'LineWidth', 2);
hold on;
stem(n, cosine_wave_discrete, 'r', 'LineWidth', 2);
title('Cosine Wave');
xlabel('Time');
ylabel('Amplitude');
legend('Continuous', 'Discrete');
grid on;
hold off;
```

RESULT

Basic test signals are generated and are verified for continuous and discrete waveforms.

Experiment No: 2 Date: 01/08/2024

VERIFICATION OF SAMPLING THEOREM

AIM: To sample a sinusoidal signal and verify Nyquist sampling criteria (Sampling Theorem) for that signal

THEORY

Sampling is a process of converting a continuous time continuous amplitude signal to discrete time continuous amplitude signal. Value of the signal is taken only at discrete points. These values are then used to reconstruct the signal. Higher sampling rates provide a more accurate reconstruction of the signal but increase data consumption. Conversely, lower sampling rates reduce data usage but result in a less accurate reconstructed signal. Hence there should be trade-off between sampling rate and memory consumption.

Nyquist theorem or Nyquist criteria states that to accurately reconstruct a signal, the sampling rate, fs should be **at least twice the maximum frequency component** of the signal fm. That is

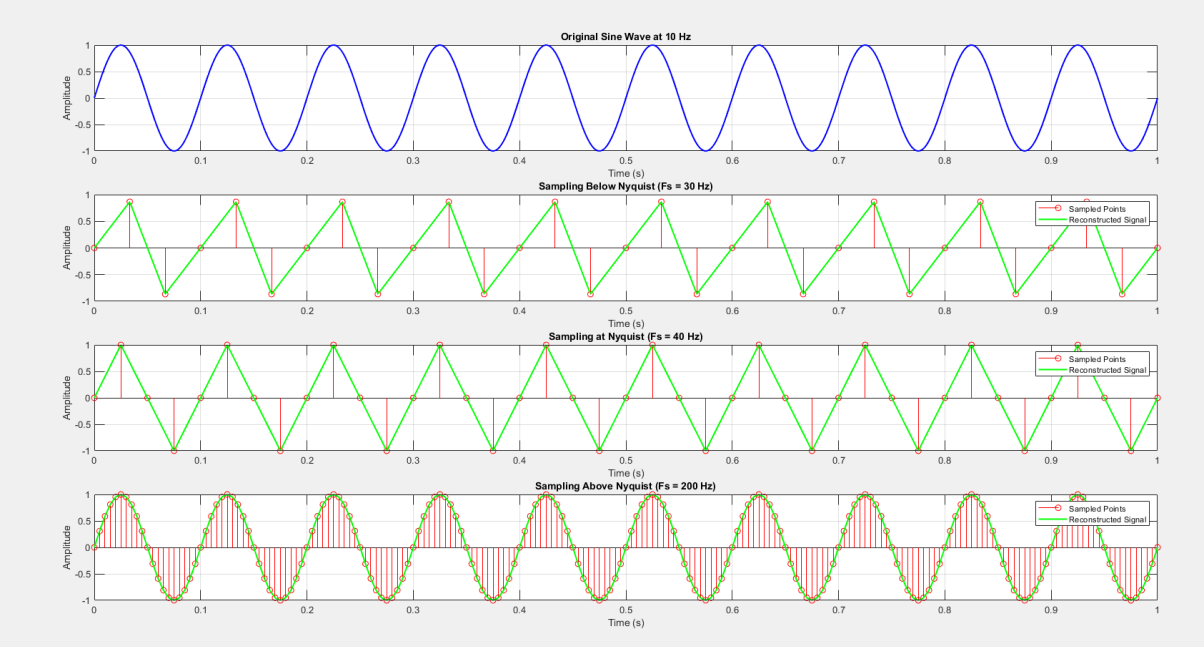
$$f_s = 2f_m$$

In practice, signals are typically sampled at or above the Nyquist rate to minimize information loss while keeping file size manageable. Sampling slightly above the Nyquist rate helps ensure accurate signal reconstruction without significantly increasing data storage requirements.

PROGRAM

```
f = input('Enter the frequency of the sine wave (Hz): ');
Fs_nyquist =4 * f;
Fs_below = 0.75 * Fs_nyquist;
Fs_above = 5 * Fs_nyquist;
duration = 1;
t_original = 0:1/1000:duration;
x_cont = sin(2*pi*f*t_original);
t_below = 0:1/Fs_below:duration;
t_nyquist = 0:1/Fs_nyquist:duration;
t_above = 0:1/Fs_above:duration;
x_below = sin(2*pi*f*t_below);
x_nyquist = sin(2*pi*f*t_nyquist);
x_above = sin(2*pi*f*t_above);
x_below_interp = interp1(t_below, x_below, t_original, 'linear');
x_nyquist_interp = interp1(t_nyquist, x_nyquist, t_original,
'linear');
```

OUTPUT


Original Sine Wave at 10 Hz
Amplitude
Time (s)
Sampling Below Nyquist (Fs = 30 Hz)
Sampled Points
Reconstructed Signal
Sampling at Nyquist (Fs = 40 Hz)
Sampling Above Nyquist (Fs = 200 Hz)

```
x_above_interp = interp1(t_above, x_above, t_original, 'linear');
figure;
subplot(4, 1, 1);
plot(t_original, x_cont, 'b', 'LineWidth', 1.5);
title(['Original Sine Wave at ', num2str(f), ' Hz']);
xlabel('Time (s)');
ylabel('Amplitude');
grid on;
subplot(4, 1, 2);
stem(t_below, x_below, 'r', 'DisplayName', 'Sampled Points');
hold on;
plot(t_original,    x_below_interp,    'g',    'LineWidth',    1.5,
'DisplayName', 'Reconstructed Signal');
title(['Sampling Below Nyquist (Fs = ', num2str(Fs_below), ' Hz)']);
xlabel('Time (s)');
ylabel('Amplitude');
legend;
grid on;
subplot(4, 1, 3);
stem(t_nyquist, x_nyquist, 'r', 'DisplayName', 'Sampled Points');
hold on;
plot(t_original,    x_nyquist_interp,    'g',    'LineWidth',    1.5,
'DisplayName', 'Reconstructed Signal');
title(['Sampling at Nyquist (Fs = ', num2str(Fs_nyquist), ' Hz)']);
xlabel('Time (s)');
ylabel('Amplitude');
legend;
grid on;
subplot(4, 1, 4);
stem(t_above, x_above, 'r', 'DisplayName', 'Sampled Points');
hold on;
plot(t_original,    x_above_interp,    'g',    'LineWidth',    1.5,
'DisplayName', 'Reconstructed Signal');
title(['Sampling Above Nyquist (Fs = ', num2str(Fs_above), ' Hz)']);
xlabel('Time (s)');
ylabel('Amplitude');
legend;
grid on;
```

RESULT

Sinusoidal signal of frequency 10 Hz was sampled and reconstructed

1. Below Nyquist rate
2. In Nyquist rate
3. Above Nyquist rate

Experiment No: 3 Date: 08/08/2024

LINEAR CONVOLUTION

AIM: To find the linear convolution of two signals with and without using built in MATLAB function. (conv( ))

THEORY:

Linear Convolution is a mathematical operation that combines two discrete-time signals to produce a third signal, representing how the shape of one signal modifies the other. It is defined as the sum of the product of the two signals after one is flipped and shifted. In the context of discrete-time signals, if (x[n]) and (h[n]) are two sequences, their linear convolution is denoted as (**y[n] = x[n] * h[n]**), and is calculated by the following equation:

$$\boldsymbol{y[n]} = \sum \boldsymbol{x(m).h(n-m)}$$

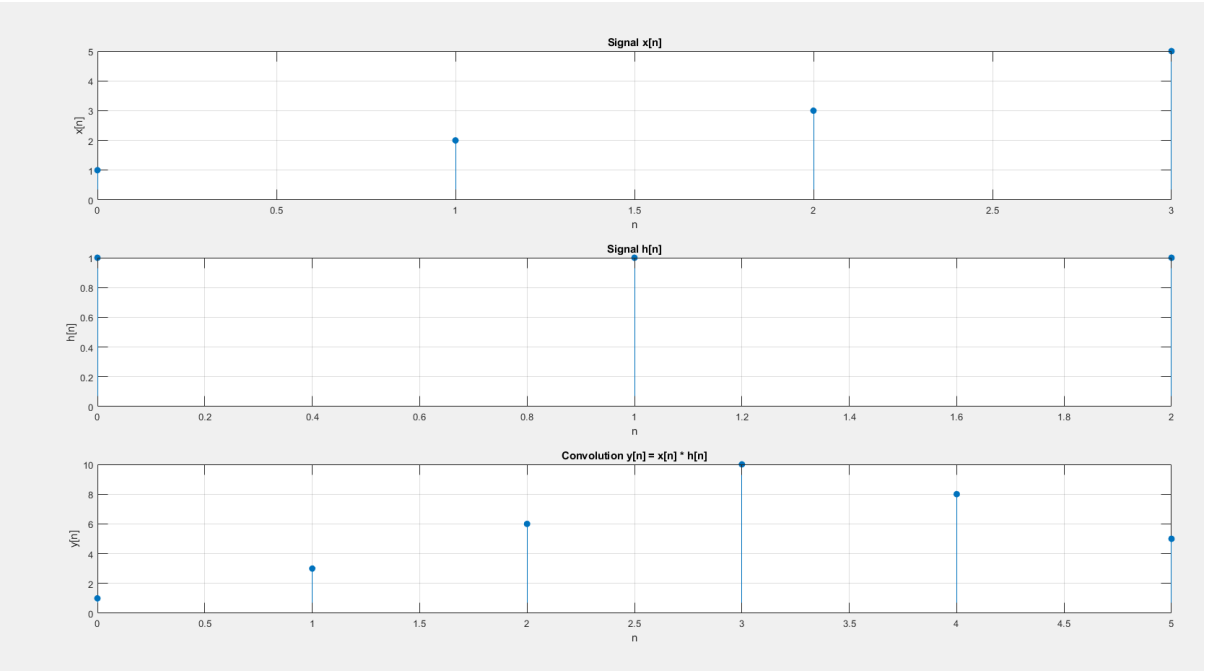
This operation is fundamental in signal processing, as it describes how a signal (input) is transformed by a system (impulse response). For finite-length signals, the convolution has a length of N+M−1, where N and M are the lengths of the two input sequences.

Program

1.Using Built in Function

```
clc;clear;
x=[1,2,3,5];
h=[1,1,1]
disp("Convolution of X and H")
y=conv(x,h)
% Plot the sequences
n_x = 0:length(x)-1;  % Time indices for x
n_h = 0:length(h)-1;  % Time indices for h
n_y = 0:length(y)-1;  % Time indices for convolution result
figure;
% Plot x[n]
subplot(3,1,1);
stem(n_x, x, 'filled');
title('Signal x[n]');
xlabel('n');
ylabel('x[n]');
grid on;
% Plot h[n]
subplot(3,1,2);
stem(n_h, h, 'filled');
title('Signal h[n]');
xlabel('n');
ylabel('h[n]');
grid on;
```

OUTPUT




```
>> run direct_method.m

h =

     1     1     1

Convolution of X and H

y =

     1     3     6    10     8     5

>>
```

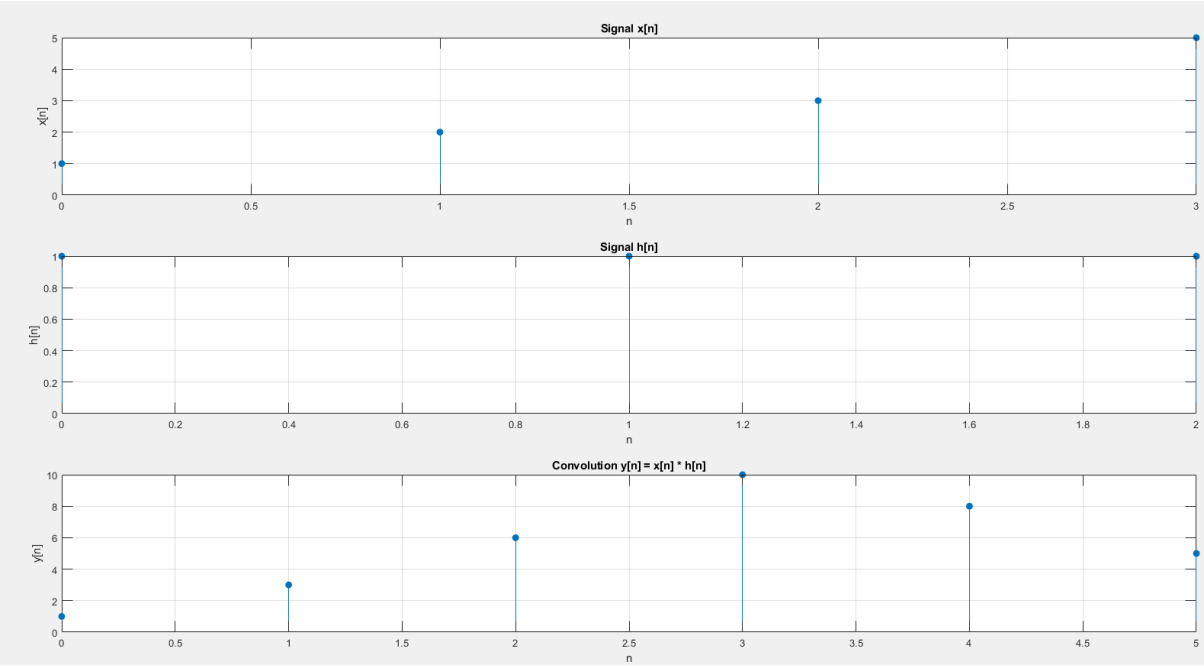
```
% Plot the convolution result y[n]
subplot(3,1,3);
stem(n_y, y, 'filled');
title('Convolution y[n] = x[n] * h[n]');
xlabel('n');
ylabel('y[n]');
grid on;
```

2. Without using built in function

```
clc;clear;close all;
x=[1,2,3,5];
h=[1,1,1];
x_len = length(x);
h_len = length(h);
y_len = x_len + h_len - 1;
x_padded = [x, zeros(1, y_len - x_len)];
h_padded = [h, zeros(1, y_len - h_len)]
h_mat = zeros(y_len, y_len);
for i = 1:y_len
    h_mat(:, i) = circshift(h_padded, [0, i-1]);  % Circularly shift
the padded h
end
y_matrix = h_mat * x_padded';
disp('Shifted h matrix:');
disp(h_mat);
disp('Convolution result using matrix method:');
disp(y_matrix');
% Plot the sequences
n_x = 0:length(x)-1;  % Time indices for x
n_h = 0:length(h)-1;  % Time indices for h
n_y = 0:length(y_matrix)-1;  % Time indices for convolution result
figure;
% Plot x[n]
subplot(3,1,1);
stem(n_x, x, 'filled');
title('Signal x[n]');
xlabel('n');
ylabel('x[n]');
grid on;
% Plot h[n]
subplot(3,1,2);
stem(n_h, h, 'filled');
title('Signal h[n]');
xlabel('n');
ylabel('h[n]');
grid on;
subplot(3,1,3);
```

Output

```
>> run matrix.m

h_padded =

     1     1     1     0     0     0

Shifted h matrix:
     1     0     0     0     1     1
     1     1     0     0     0     1
     1     1     1     0     0     0
     0     1     1     1     0     0
     0     0     1     1     1     0
     0     0     0     1     1     1

Convolution result using matrix method:
     1     3     6    10     8     5
```



```
stem(n_y, y_matrix, 'filled');
title('Convolution y[n] = x[n] * h[n]');
xlabel('n');
ylabel('y[n]');
grid on;
```

RESULT

Linear convolution was performed and visualized on a given input sequences using built in function (.conv()) and using matrix method.

Experiment No: 4 Date: 15/08/2024

CIRCULAR CONVOLUTION

AIM: To find the circular convolution of two signals with and without using built in MATLAB function. (cconv( ))

THEORY

Circular convolution is a type of convolution where the signals are assumed to be periodic, meaning the signal wraps around after a certain length. Unlike linear convolution, which considers the entire duration of the signals, circular convolution treats the signals as though they repeat indefinitely, and the result is limited to the length of the longest sequence.

Mathematically, for two discrete sequences x[n] and h[n], both of length N, their circular convolution y[n] is defined as:

$$y[n] = x[n] \textcircled{c} h[n] = \sum_{k=0}^{N-1} x[k].\, h[(n-k) mod\ N]$$

where, y[n] is the result of circular convolution, x[n] and h[n] are input sequences of length N, © represents circular convolution and (n-k) mod N ensures that the indices wrap around when they exceed the length of the sequence.

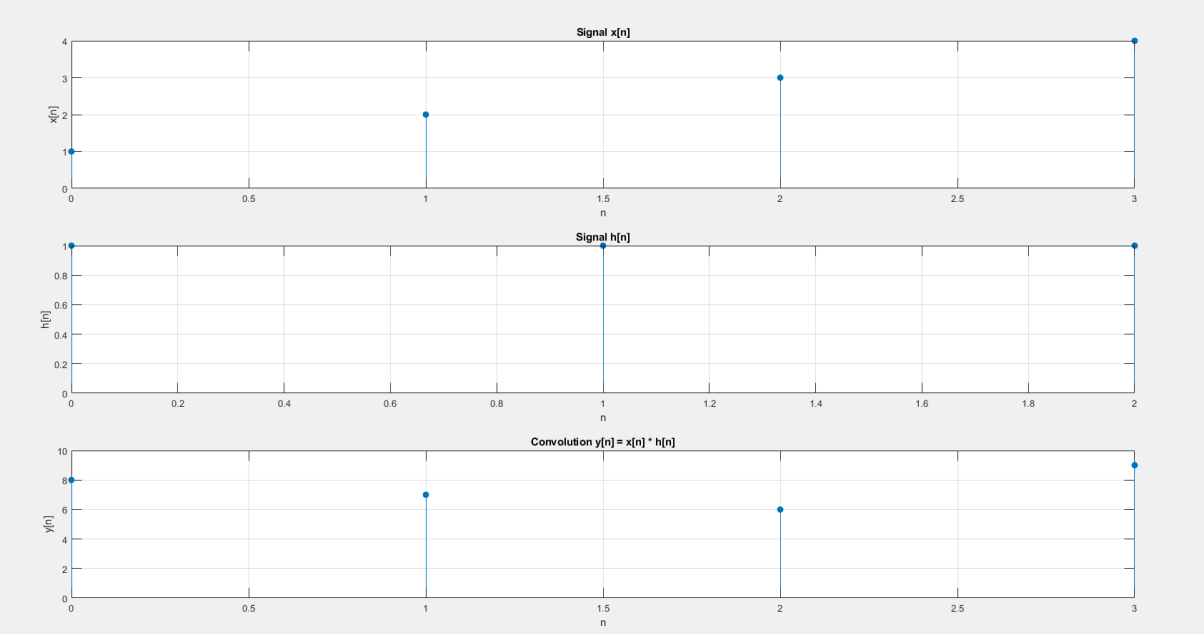
PROGRAM

1. Direct Method

```
clc;clear;close all;
x = [1, 2, 3, 4];
h = [1,1,1];
N=max(length(x),length(h));
disp('Convolution result using matrix method:');
y=cconv(x,h,N)
```

OUTPUT

```
Convolution result using matrix method:

y =

     8     7     6     9
```



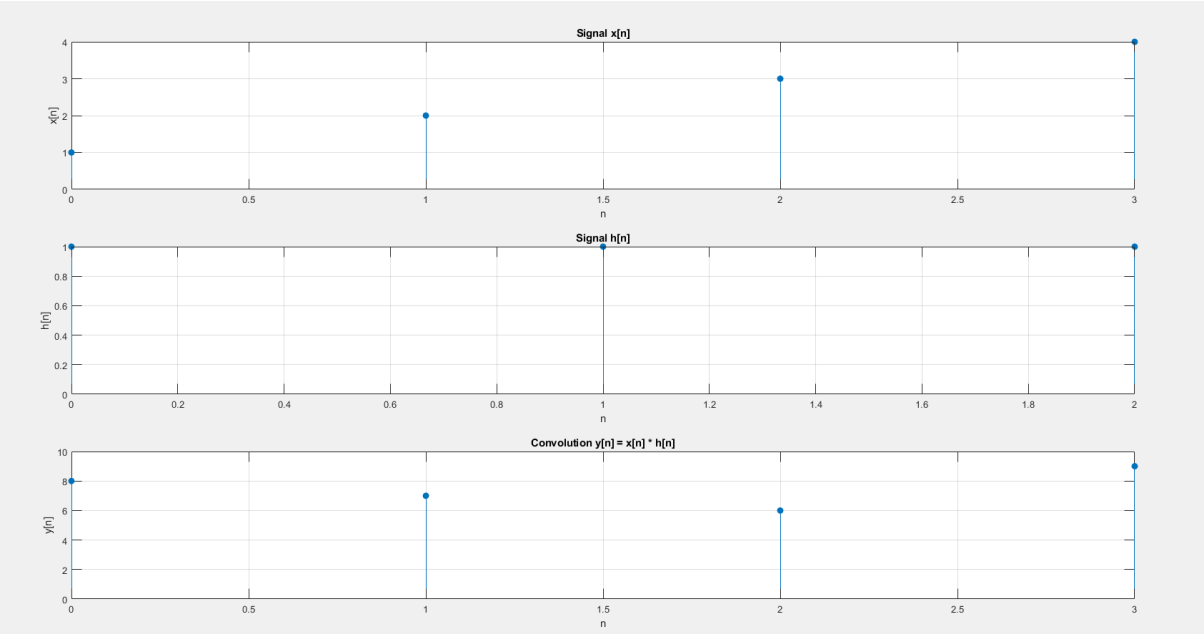
```
% Plot the sequences
n_x = 0:length(x)-1;  % Time indices for x
n_h = 0:length(h)-1;  % Time indices for h
n_y = 0:length(y)-1;  % Time indices for convolution result
figure;
% Plot x[n]
subplot(3,1,1);
stem(n_x, x, 'filled');
title('Signal x[n]');
xlabel('n');
ylabel('x[n]');
grid on;
% Plot h[n]
subplot(3,1,2);
stem(n_h, h, 'filled');
title('Signal h[n]');
xlabel('n');
ylabel('h[n]');
grid on;
% Plot the convolution result y[n]
subplot(3,1,3);
stem(n_y, y, 'filled');
title('Convolution y[n] = x[n] * h[n]');
xlabel('n');
ylabel('y[n]');
grid on;
```

2. Using Matrix Method

```
clc;clear;close all;
x = [1, 2, 3, 4];
h = [1, 1, 1];
x_len = length(x);
h_len = length(h);
y_len = max(x_len,h_len);
x_padded = [x, zeros(1, y_len - x_len)];
h_padded = [h, zeros(1, y_len - h_len)]
h_mat = zeros(y_len, y_len);
for i = 1:y_len
    h_mat(:, i) = circshift(h_padded, [0, i-1]);% Circularly shift h
end
y_matrix = h_mat * x_padded';
disp('Shifted h matrix:');
disp(h_mat);
disp('Convolution result using matrix method:');
disp(y_matrix');
% Plot the sequences
n_x = 0:length(x)-1;  % Time indices for x
```

OUTPUT

```
h_padded =

     1     1     1     0

Shifted h matrix:
     1     0     1     1
     1     1     0     1
     1     1     1     0
     0     1     1     1

Convolution result using matrix method:
     8     7     6     9
```



```
n_h = 0:length(h)-1;  % Time indices for h
n_y = 0:length(y)-1;  % Time indices for convolution result
figure;
% Plot x[n]
subplot(3,1,1);
stem(n_x, x, 'filled');
title('Signal x[n]');
xlabel('n');
ylabel('x[n]');
grid on;
% Plot h[n]
subplot(3,1,2);
stem(n_h, h, 'filled');
title('Signal h[n]');
xlabel('n');
ylabel('h[n]');
grid on;
% Plot the convolution result y[n]
subplot(3,1,3);
stem(n_y, y, 'filled');
title('Convolution y[n] = x[n] * h[n]');
xlabel('n');
ylabel('y[n]');
grid on;
```

RESULT

Circular convolution was performed and visualized using built in function (cconv) and using matrix method for a given input sequence.

Experiment No: 5 Date: 22/08/2024

LINEAR CONVOLUTION USING CIRCULAR CONVOLUTION AND CIRCULAR CONVOLUTION USING LINEAR CONVOLUTION

AIM: To perform Linear Convolution using Circular Convolution and to perform Circular Convolution using Linear Convolution.

THEORY

Linear convolution can be performed using circular convolution performing the following algorithm

1. **Calculate the length** of the resultant convolved matrix by using the formula, L+M-1 where L, M are the matrix are the length of the input sequences.
2. **Zero pad** the input matrices to the length of the resultant convolved matrix
3. **Perform the circular convolution** either using the built in function (cconv) or without using the matrix method.

Circular convolution can be performed using linear convolution by performing the following algorithm

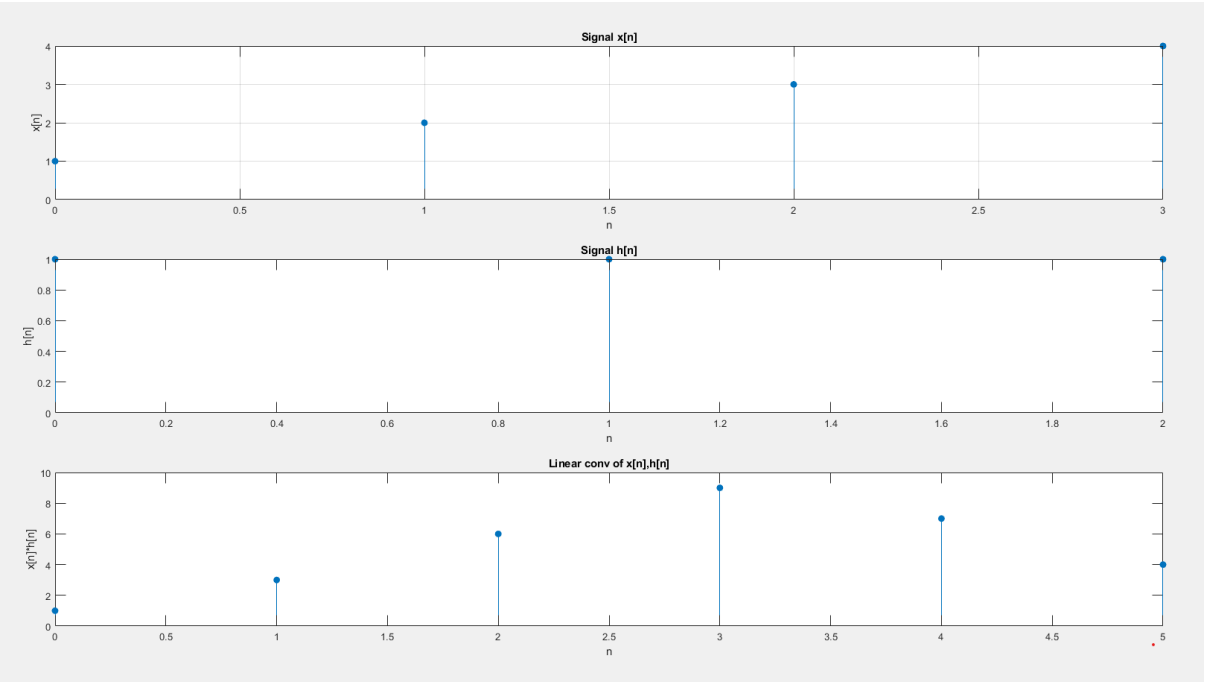
1. **Zero pad** both the input sequences to the length 2N-1 where N is the maximum length of the sequence
2. Perform **linear convolution** on both the sequences
3. Perform the **modulus operation** to the indices of the linear convolution result, using the period N. This effectively wraps around the ends of the sequence, making it circular.

PROGRAM

1.Linear Convolution using Circular convolution

```
clc; clear; close all;
x = [1, 2, 3, 4];
h = [1, 1, 1];
disp('Linear convolved sequence using built-in function:');
x_conv = conv(x, h);
disp(x_conv);
conv_length = length(x) + length(h) - 1;
x_new = [x, zeros(1, conv_length - length(x))];
h_new = [h, zeros(1, conv_length - length(h))];
X = fft(x_new);
H = fft(h_new);
Y = ifft(X .* H);
disp('Circular convolution result using FFT:');
disp(Y);
n_x=0:length(x)-1;
```

OUTPUT




```
Linear convolved sequence using built-in function:
     1     3     6     9     7     4

Circular convolution result using FFT:
     1     3     6     9     7     4
```

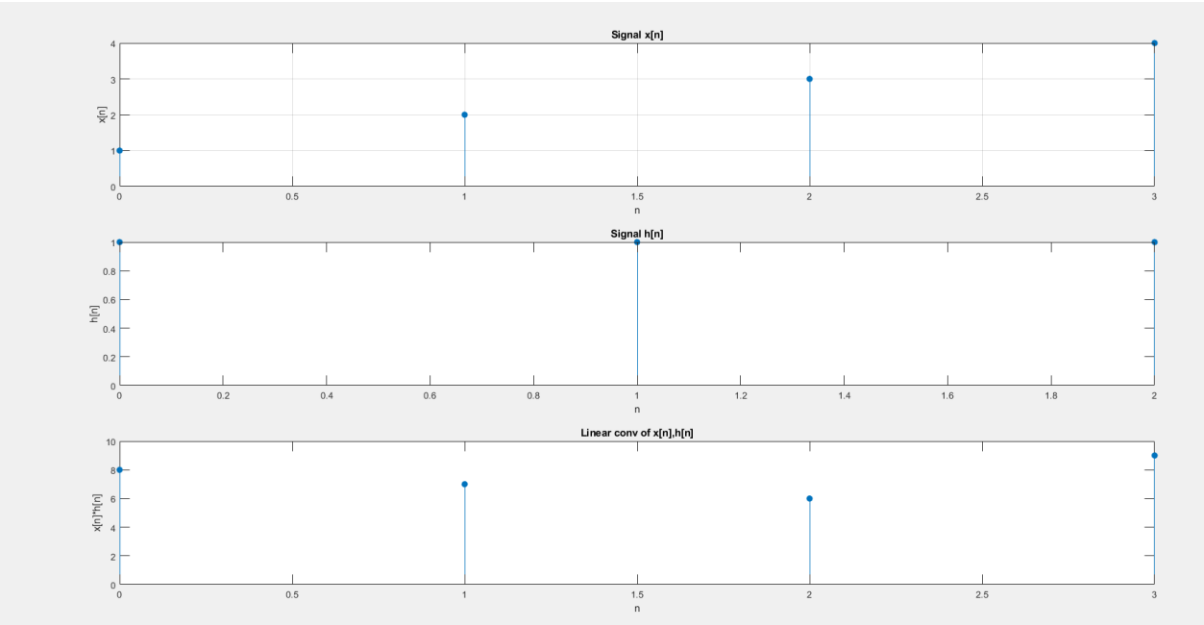
```
n_h=0:length(h)-1;
n_y=0:length(Y)-1;
figure;
subplot(3,1,1);
stem(n_x,x,'filled');
title('Signal x[n]');
xlabel('n');
ylabel('x[n]');
grid on;
subplot(3,1,2);
stem(n_h,h,"filled");
title('Signal h[n]');
xlabel('n');
ylabel('h[n]');
subplot(3,1,3);
stem(n_y,Y,"filled");
title('Linear conv of x[n],h[n]');
xlabel('n');
ylabel('x[n]*h[n]');
```

2. Circular Convolution using linear convolution

```
x = [1, 2, 3, 4];
h = [1, 1, 1];
disp('linear convolved sequence')
x_conv = conv(x, h)
new_len = length(x);
if length(h) > length(x)
    new_len = length(h);
end
diff=length(x_conv)-new_len;
x_new=x_conv(1:new_len);

for i=1:diff
    x_new(i)=x_conv(i)+x_conv(i+new_len);
end
disp('circular Convolved sequence')
x_new
n_x=0:length(x)-1;
n_h=0:length(h)-1;
n_y=0:length(x_new)-1;
figure;
subplot(3,1,1);
stem(n_x,x,'filled');
title('Signal x[n]');
xlabel('n');
```

OUTPUT

```
>> run code.m
linear convolved sequence

x_conv =

     1     3     6     9     7     4

circular Convolved sequence

x_new =

     8     7     6     9
```



```
ylabel('x[n]');
grid on;
subplot(3,1,2);
stem(n_h,h,"filled");
title('Signal h[n]');
xlabel('n');
ylabel('h[n]');
subplot(3,1,3);
stem(n_y,x_new,"filled");
title('Linear conv of x[n],h[n]');
xlabel('n');
ylabel('x[n]*h[n]');
```

RESULT

Linear convolution was performed using circular convolution and circular convolution was performed using linear convolution.

Experiment No: 6 Date: 1/10/2024

DFT and IDFT

Aim:

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

2.IDFT using inbuilt function and without using inbuilt function.

Theory:

Discrete Fourier Transform (DFT

The Discrete Fourier Transform (DFT) is a mathematical operation that decomposes a finite sequence of numbers into a sum of complex sinusoids. It's a fundamental tool in digital signal processing, used for analyzing and manipulating signals in the frequency domain.

Mathematical Definition

Given a sequence of N complex numbers, x[n], where n = 0, 1, ..., N-1, the DFT is defined as:

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

where:

- X[k] is the DFT coefficient at frequency k.
- j is the imaginary unit (√-1).
- n and k are indices.

Fast Fourier Transform (FFT)

The Fast Fourier Transform (FFT) is an efficient algorithm for computing the DFT. It reduces the computational complexity from O(N^2) to O(N log N), making it practical for large sequences.

Applications of DFT

- **Frequency Analysis:** Identifying the frequency components of a signal.
- **Filtering:** Removing unwanted frequency components from a signal.
- **Spectrum Analysis:** Analyzing the power spectrum of a signal.
- **Image Processing:** Image filtering, compression, and reconstruction.
- **Communications:** Modulation and demodulation of signals.

Inverse Discrete Fourier Transform (IDFT)Method:

The **Inverse Discrete Fourier Transform (IDFT)** is a mathematical operation used to reconstruct a finite sequence of numbers from its Discrete Fourier Transform (DFT) coefficients. It's essentially the reverse process of the DFT.

Mathematical Definition

Given a sequence of N complex DFT coefficients, X[k], where k = 0, 1, ..., N-1, the IDFT is defined as:

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

where:

- x[n] is the reconstructed sequence at index n.
- j is the imaginary unit (√-1).
- n and k are indices.

Applications of DFT

- **Frequency Analysis:** Identifying the frequency components of a signal.
- **Filtering:** Removing unwanted frequency components from a signal.
- **Spectrum Analysis:** Analyzing the power spectrum of a signal.
- **Image Processing:** Image filtering, compression, and reconstruction.
- **Communications:** Modulation and demodulation of signals.

Applications of IDFT

- **Signal Reconstruction:** Recovering a time-domain signal from its frequency-domain representation.
- **Filtering:** Applying filters to a signal in the frequency domain and then using the IDFT to obtain the filtered time-domain signal.
- **Image Processing:** Image reconstruction from frequency-domain data.
- **Communications:** Demodulation of signals.

Code

Discrete Time Fourier Transform

1. Direct Method

```
clc;clear;
x=[1,2,3,4]
disp('dft of x[n] is');
X_fft=fft(x);
disp(X_fft);
figure;
subplot(2,1,1);
stem(0:length(X_fft)-1,abs(X_fft));
xlabel('frequency');
ylabel('Magnitude');
title('DFT of x');
subplot(2,1,2);
stem(0:length(X_fft)-1,angle(X_fft),"filled");
xlabel('frequency');
ylabel('phase (rads)');
title('phase spectrum')
```

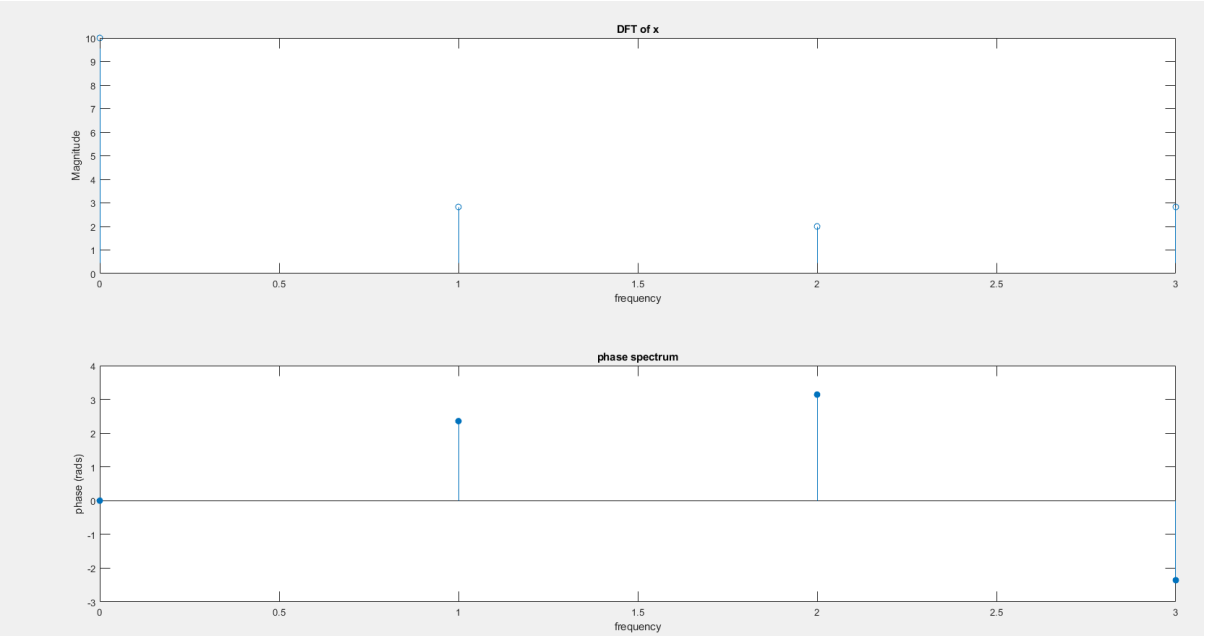
2. Without using built in method

```
clc; clear
x=[1,2,3,4];
N=length(x);
X_fft=zeros(1,N);
for k=0:N-1
    for n=0:N-1
        X_fft(k+1)=X_fft(k+1)+x(n+1)*exp(-1i*2*pi*k*n/N);
    end
end
disp("DFT of x[n] is");
```

Observation

DFT

```
>> with_builtin

x =

     1     2     3     4

dft of x[n] is
  10.0000 + 0.0000i  -2.0000 + 2.0000i  -2.0000 + 0.0000i  -2.0000 - 2.0000i

>>
```



```
disp(abs(X_fft));

figure;

stem(abs(X_fft));

xlabel('Time scale');

ylabel('Magnitude');

title('DFT of x[n]');
```

Inverse Discrete Fourier Transform

1. Direct Method

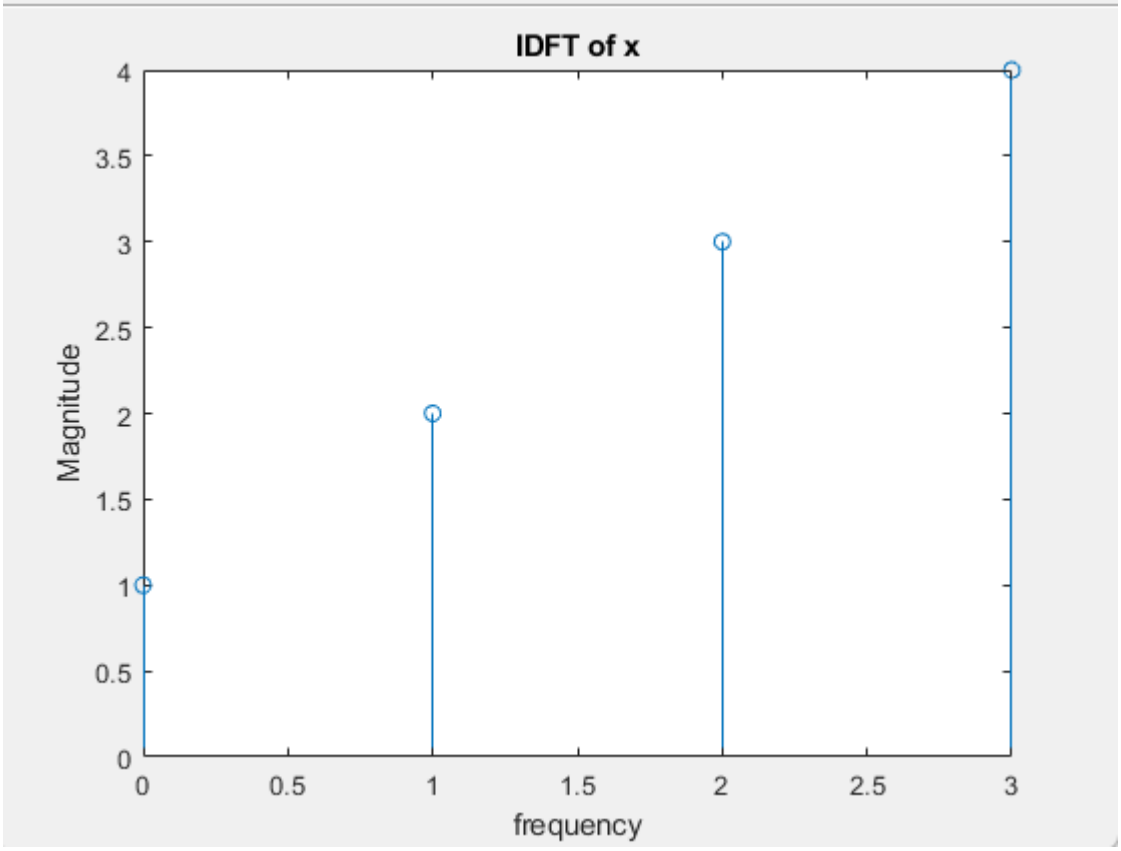
```
disp('Inverse fourier transform of the sequence');

x_fft=[10.0000 + 0.0000i,  -2.0000 + 2.0000i,  -2.0000 + 0.0000i,  -
2.0000 - 2.0000i]

disp('is');

x=ifft(x_fft)

stem(0:length(x)-1,abs(x));

xlabel('frequency');

ylabel('Magnitude');

title('IDFT of x');
```

2. Without using built in method

```
disp('Inverse fourier transform of the sequence');
x_fft=[10.0000 + 0.0000i,  -2.0000 + 2.0000i,  -2.0000 + 0.0000i,  -
2.0000 - 2.0000i]
disp('is');
x=zeros(1,length(x_fft));
for k=0:length(x_fft)-1
    for m=0:length(x_fft)-1
        x(k+1)=x(k+1)+x_fft(m+1)*exp(1i*2*pi*m*k/length(x_fft));
    end
end
x=(1/length(x_fft))*x
abs(x)
stem(0:length(x)-1,abs(x));
xlabel('frequency');
ylabel('Magnitude');
title('IDFT of x');
```

IDFT

```
>> code
Inverse fourier transform of the sequence

x_fft =

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

is

x =

     1     2     3     4
```



Result:

Performed

1)DFT using inbuilt function and without using inbuilt function. Also plotted magnitude and phase plot of DFT.

2)IDFT using inbuilt function and without using inbuilt function and verified the result.

Experiment No: 7 Date: 01/10/24

Properties of DFT

Aim:

Verify following properties of DFT using Matlab.

1.Linearity Property

2.Parsevals Theorem

3.Convolution Property

4.Multiplication Property

Theory:

1. Linearity Property

The linearity property of the DFT states that if you have two sequences x1[n] and x2[n], and their corresponding DFTs are X1[k] and X2[k], then for any scalar a and b:

$$\mathbf{DFT\{a\cdot x1[n]+b\cdot x2[n]\} = a\cdot DFT\{x1[n]\}+b\cdot DFT\{x2[n]\}}$$

2. Parseval's Theorem

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

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

3.Convolution Property

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

$$\mathbf{DFT\{x1[n]\circledast x2[n]\}=DFT\{x1[n]\}\cdot DFT\{x2[n]\}}$$

4. Multiplication Property

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

$$\mathbf{DFT\{x1[n]\cdot x2[n]\} = \frac{1}{N}DFT\{x1[n]\}\circledast DFT\{x2[n]\}}$$

Program:

1. Linearity Property

```
clc; clear;
```

Observation:

1. Linearity Property

enter first sequence[1 2 3 4]

enter sequence sequence:[1 1 1 1]

x =

1 2 3 4

enter value of 'a':2

enter value of 'b':3

LHS

32.0000 + 0.0000i -4.0000 + 4.0000i -4.0000 + 0.0000i -4.0000 - 4.0000i

RHS

32.0000 + 0.0000i -4.0000 + 4.0000i -4.0000 + 0.0000i -4.0000 - 4.0000i

Linearity property verified

2. Parseval's Theorem

enter first sequence:[1 2 3 4]

enter second sequence:[1 1 1 1]

LHS

10

RHS

10

Parseval's Theorem verified

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

2. Parseval's Theorem

```
clc;
clear all;
close all;
x=input("enter first sequence:");
h=input("enter second sequence:");
N=max(length(x),length(h));
```

3.Convolution Property

enter first sequence[1 2 3 4]

enter sequence sequence:[1 1 1 1]

LHS

10 10 10 10

RHS

10 10 10 10

Circular Convolution verified

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

3.Convolution Property

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

4.Multiplication Property

enter first sequence[1 2 3 4]

enter sequence sequence:[1 1 1 1]

LHS

Columns 1 through 3

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

Column 4

-2.0000 - 2.0000i

RHS

Columns 1 through 3

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

Column 4

-2.0000 - 2.0000i

Multiplication property verified

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

4. Multiplication Property

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

```
rhs=(cconv(Xn,Hn,N))/N;
disp('LHS');
disp(lhs);
disp('RHS');
disp(rhs);
if lhs==rhs
    disp('Multiplication property verified');
else
    disp('Multiplication property not verified');
end
```

Result:

Performed and verified the following properties of DFT:

1.Linearity Property

2.Parsevals Theorem

3.Convolution Property

4.Multiplication Property

Experiment No: 8 Date: 08/10/24

OVERLAP ADD AND OVERLAP SAVE METHOD

Aim:

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

Theory:

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

Overlap-Save Method

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

1. Block Decomposition: The input signal is divided into overlapping blocks. If the filter has length and we use blocks of length, the overlap is samples, so each block has new samples and samples from the previous block.

2. FFT and Convolution: Each block is convolved with the filter using FFT. However, because of circular convolution, the result contains artifacts due to the overlap.

3. Discard and Save: We discard the first samples from each block (the part affected by the wrap-around) and save the remaining samples. This gives us the correct linear convolution.

Overlap-Add Method

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

1. Block Decomposition: The input signal is split into non-overlapping blocks of size. Each block is then zero-padded to a size of , where is the length of the filter.

2. FFT and Convolution: Each block is convolved with the filter using FFT. Since the blocks are zero-padded, the convolution produces valid linear results, but the output blocks overlap.

3. Overlap and Add: After convolution, the results of each block overlap by samples. These overlapping regions are added together to form the final output.

Program:

1. Overlap Add

```
clc;
clear all;
close all;
% User input for the input sequence
x = input('Enter the input sequence x : ');
% User input for the impulse response
h = input('Enter the impulse response h : ');
% Section length for overlap-save
L = length(h);  % Length of impulse response
% Initialization
N = length(x);
M = length(h);
% Pad input x with zeros
x_padded = [x, zeros(1, L - 1)];
% Prepare the output array
y = zeros(1, N + M +1);
% Calculate the number of sections
num_sections = (N + L - 1) / L;  % Calculate number of sections
% Process sections
for n = 0:num_sections-1
    % Determine the current section
    start_idx = n * L + 1;
    end_idx = start_idx + L - 1;
    % Ensure the section does not exceed the bounds
    x_section = x_padded(start_idx:min(end_idx, end));
    % Convolution
    conv_result = conv(x_section, h);
    % Save the results to the output
```

```
    y(start_idx:start_idx     +     length(conv_result)     -     1)
=y(start_idx:start_idx + length(conv_result) - 1) + conv_result;
end
% Trim the output to the valid part
y = y(1:N + M - 1);
% Compare with built-in convolution
y_builtin = conv(x, h);
% Display results
disp('Overlap-add convolution result:');
disp(y);
disp('Built-in convolution result:');
disp(y_builtin);
% Plotting results
figure;
subplot(2, 1, 1);
stem(y, 'filled');
title('Overlap-add Convolution Result');
grid on;
subplot(2, 1, 2);
stem(y_builtin, 'filled');
title('Built-in Convolution Result');
grid on;
```

2.Overlap Save

```
clc;
clear all;
close all;
% Input the sequences and block size
x = input("Enter 1st sequence: ");
```

Observation:

1. Overlap Add

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

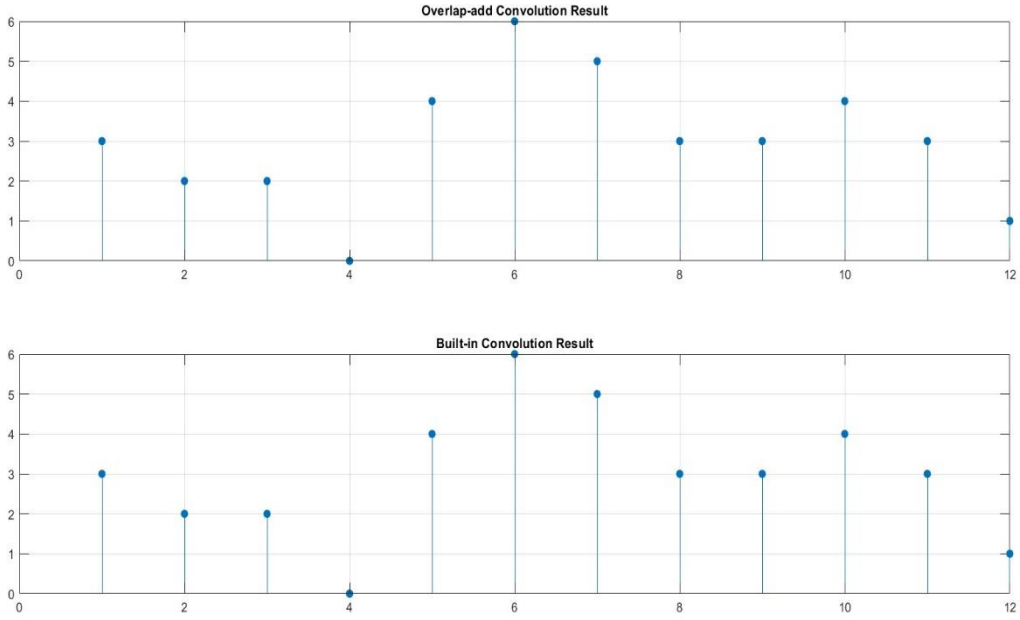
Enter the impulse response h : [1 1 1]

Overlap-add convolution result:

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

Built-in convolution result:

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



2.Overlap Save

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

Enter 2nd sequence: [1 1 1]

Fragmented block size: 3

Using Overlap and Save method

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

```
h = input("Enter 2nd sequence: ");
N = input("Fragmented block size: ");
% Call the overlap-save function
y = ovrlsav(x, h, N);
disp("Using Overlap and Save method");
disp(y);
disp("Verification");
disp(cconv(x,h,length(x)+length(h)-1));
% Define the overlap-save method function
function y = ovrlsav(x, h, N)
    if (N < length(h))
        error("N must be greater than the length of h");
    end
    Nx = length(x);  % Length of input sequence x
    M = length(h);   % Length of filter sequence h
    M1 = M - 1;      % Length of overlap
    L = N - M1;      % Length of non-overlapping part
    % Zero-padding for input and filter sequences
    x = [zeros(1, M1), x, zeros(1, N-1)];
    h = [h, zeros(1, N - M)];
    % Number of blocks
    K = floor((Nx + M1 - 1) / L);
    % Initialize the output matrix Y
    Y = zeros(K + 1, N);

    % Perform block convolution using circular convolution
    for k = 0:K
        xk = x(k*L + 1 : k*L + N);  % Extract block of input sequence
        Y(k+1, :) = cconv(xk, h, N);  % Circular convolution
```

Verification

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

```
end
    % Extract valid part from the result and concatenate
    Y = Y(:, M:N)';
    y = (Y(:))';
end
```

Result:

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

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

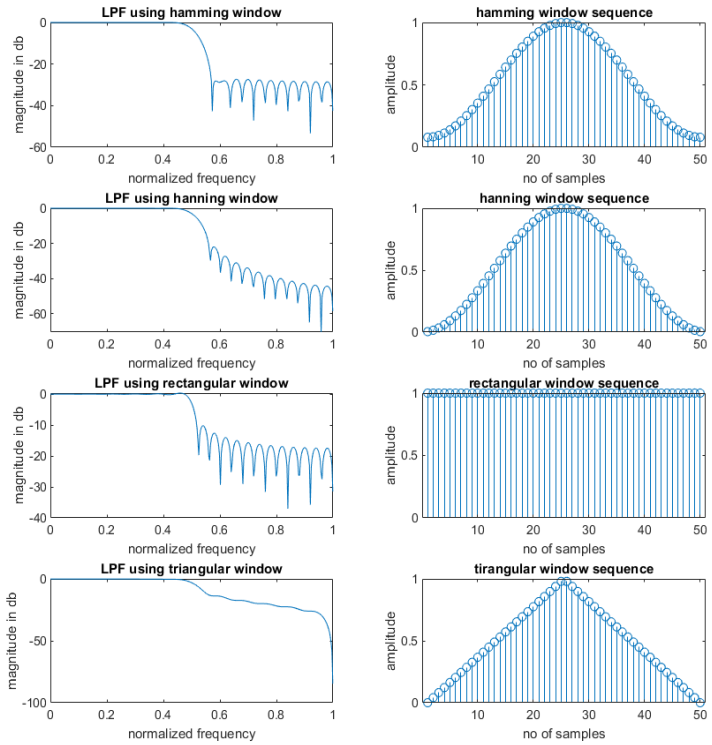
Program

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);
```

Observation

a) Low Pass Filter



```
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');
```

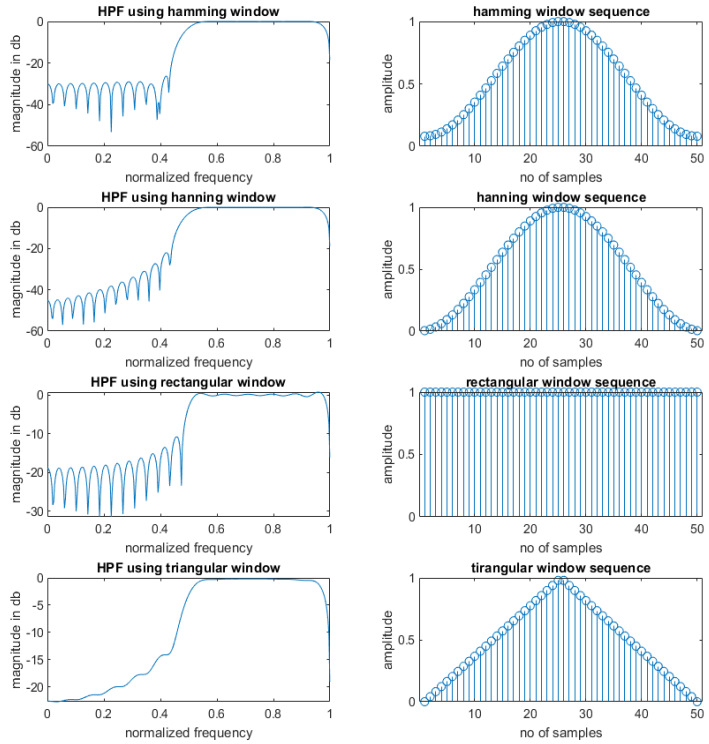
```
xlabel('normalized frequency');
ylabel('magnitude in db');
%hamming
subplot(4,2,2);
stem(w1);
title('hamming window sequence');
xlabel('no of samples');
ylabel('amplitude');
%hanning
subplot(4,2,4);
stem(w2);
title('hanning window sequence');
xlabel('no of samples');
ylabel('amplitude');
%rectangular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

Observation

b. High Pass Filter



```
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
```

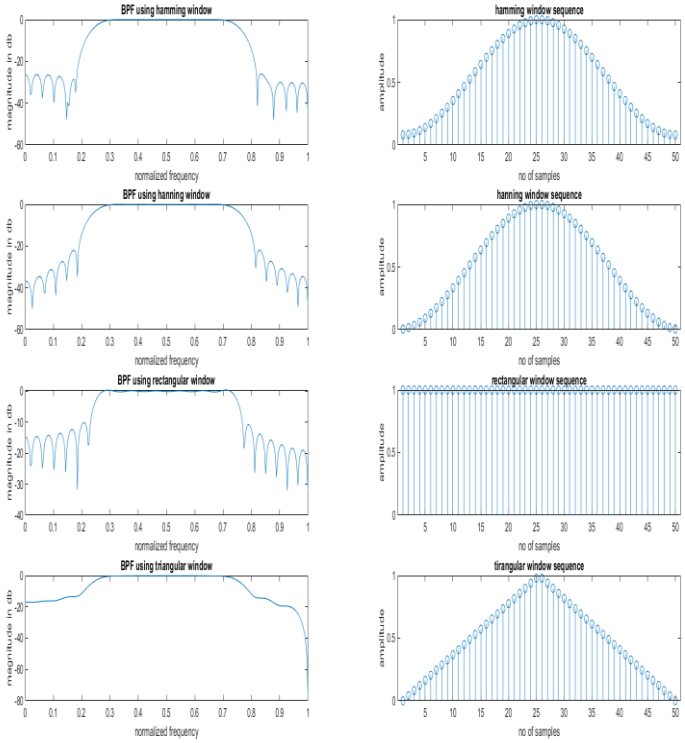
```
subplot(4,2,2);
stem(w1);
title('hamming window sequence');
xlabel('no of samples');
ylabel('amplitude');
%hanning
subplot(4,2,4);
stem(w2);
title('hanning window sequence');
xlabel('no of samples');
ylabel('amplitude');
%rectangular
subplot(4,2,6);
stem(w3);
title('rectangular window sequence');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

Observation

Band Pass Filter



```
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');
```

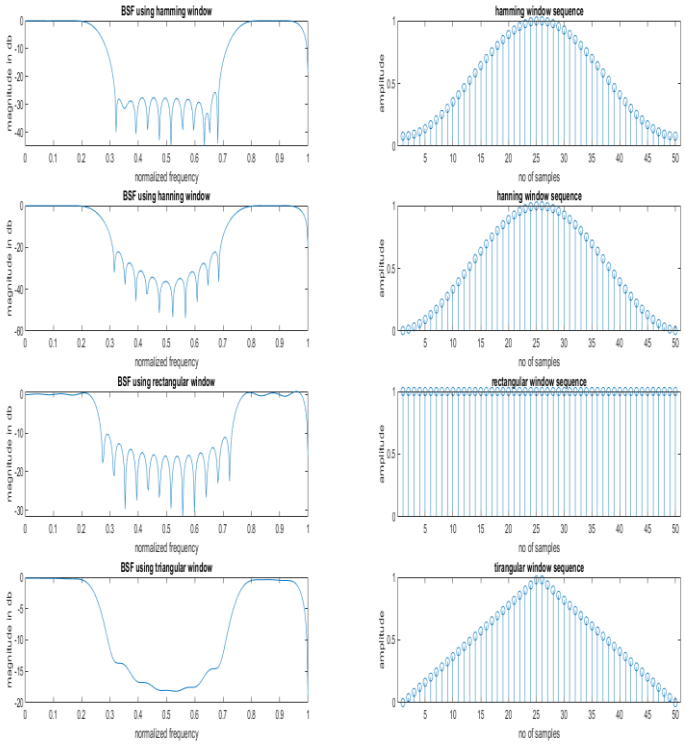
```
ylabel('magnitude in db');
%WINDOWS
%hamming
subplot(4,2,2);
stem(w1);
title('hamming window sequence');
xlabel('no of samples');
ylabel('amplitude');
%hanning
subplot(4,2,4);
stem(w2);
title('hanning window sequence');
xlabel('no of samples');
ylabel('amplitude');
%rectangular
subplot(4,2,6);
stem(w3);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

Observation

Band Pass Filter



```
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.

Experiment No:10 Date: 24-10-2024

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

AIM

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

THEORY

TMS 320C674x DSP CPU

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.

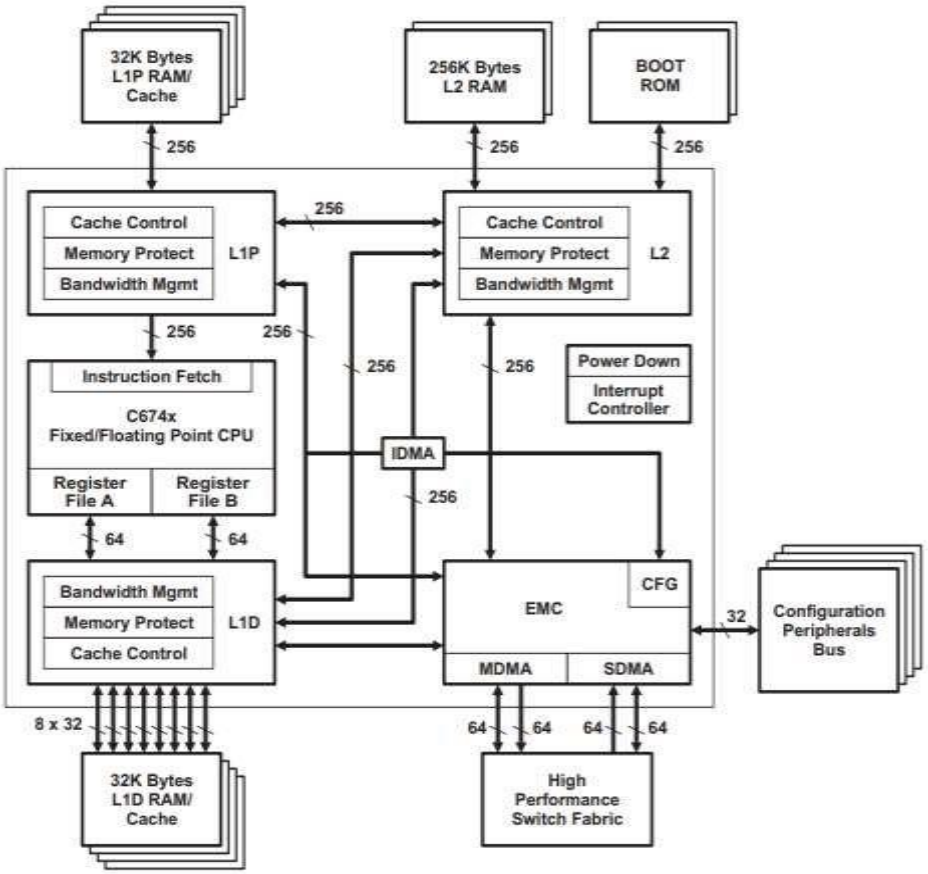


FIGURE: TMS320C 674X DSP CPU BLOCK DIAGRAM

Multichannel Audio Serial Port (McASP):

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

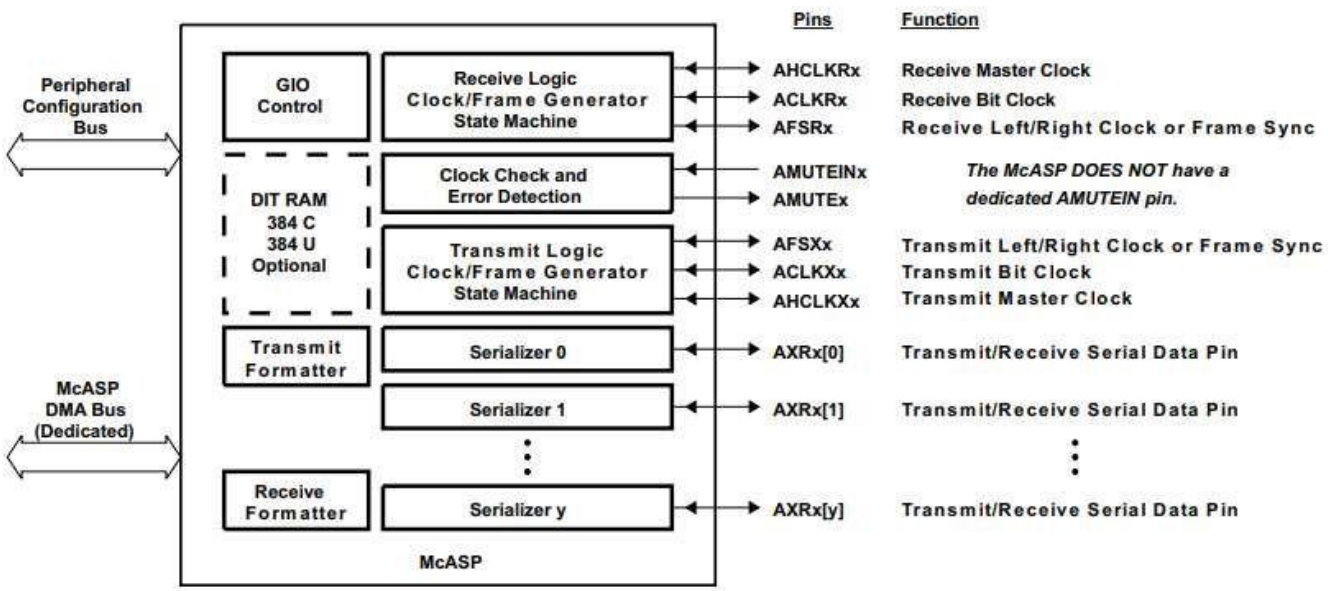
Its key features are:

- 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.


Pins
Function
Peripheral Configuration Bus
GIO Control
Receive Logic Clock/Frame Generator State Machine
AHCLKRx
Receive Master Clock
ACLKRx
Receive Bit Clock
AFSRx
Receive Left/Right Clock or Frame Sync
DIT RAM 384 C 384 U Optional
Clock Check and Error Detection
AMUTEINx
AMUTEx
The McASP DOES NOT have a dedicated AMUTEIN pin.
Transmit Logic Clock/Frame Generator State Machine
AFSXx
Transmit Left/Right Clock or Frame Sync
ACLKXx
Transmit Bit Clock
AHCLKXx
Transmit Master Clock
McASP DMA Bus (Dedicated)
Transmit Formatter
Serializer 0
AXRx[0]
Transmit/Receive Serial Data Pin
Serializer 1
AXRx[1]
Transmit/Receive Serial Data Pin
Receive Formatter
Serializer y
AXRx[y]
Transmit/Receive Serial Data Pin
McASP

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.

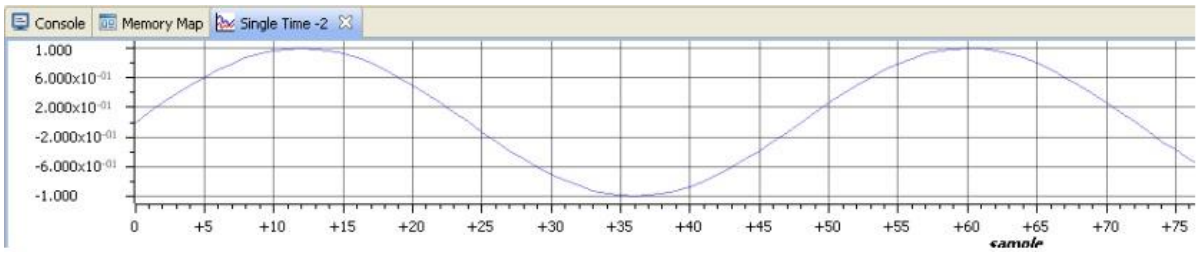
2. Type the code program for generating the sine wave and choose
File – Save As and then save the program with a name including ‘main.c’.
Delete the already existing main.c program.

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.

Observation



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.

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)
```

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

```
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.