Digital Signal Processing Lab

Laboratory report submitted for the partial fulfillment
of the requirements for the degree of

Bachelor of Technology
in
Electronics and Communication Engineering

by

Mohit Akhouri - 19ucc023

Course Coordinator
Dr. Divyang Rawal

Department of Electronics and Communication Engineering
The LNM Institute of Information Technology, Jaipur

September 2021

Contents

List of Figures

Chapter *1*

Experiment - 1

1.1 Aim of the Experiment

- Signal Sampling and Reconstruction
- Introduction to Simulink

1.2 Software Used

- MATLAB
- Simulink

1.3 Theory

1.3.1 About Sampling :

In signal processing, Sampling is the reduction of a continuous-time signal to a discrete-time signal. A common example could be the conversion of a sound wave (a continuous signal) to a sequence of samples (a discrete-time signal). A **sample** is a value or set of values at a point in time and/or space. A **sampler** is a subsystem or operation that extracts samples from a continuous signal. For functions that vary with time, let s(t) be a continuous function (or "signal") to be sampled, and let sampling be performed by measuring the value of the continuous function every T seconds, which is called the sampling interval or the sampling period.Then the sampled function is given by the sequence:

$$S(nT) \tag{1.1}$$

where **n** can be any integer. The **sampling frequency** or **sampling rate**, f_s, is the average number of samples obtained in one second (samples per second), thus $f_s = 1/T_s$. The original signal is retrievable from a sequence of samples, up to the **Nyquist limit**, by passing the sequence of samples through a type of low pass filter called a **reconstruction filter**.

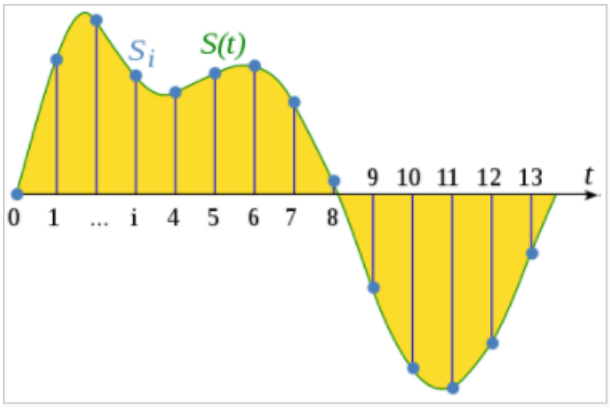


Figure 1.1 Signal Sampling

In the above figure, **green lines** represent the continuous signal and **blue vertical lines** represent the sampled signal.

1.3.2 About Signal Reconstruction :

In signal processing, reconstruction usually means the determination of an original continuous signal from a sequence of equally spaced samples. The process of reconstruction, also commonly known as **interpolation**, produces a continuous time signal that would sample to a given discrete time signal at a specific sampling rate. Reconstruction can be mathematically understood by first generating a continuous time impulse train

$$x_{imp}(t) = \sum_{n=-\infty}^{\infty} x_s(n)\delta(t - nT_s) \tag{1.2}$$

from the sampled signal x_s with sampling period T_s and then applying a lowpass filter G that satisfies certain conditions to produce an output signal $\overline{x}$. Here $\overline{x}$ is given as :

$$\overline{x} = (x_{imp} * g)(t) = \sum_{n=-\infty}^{\infty} x_s(n)g(t - nT_s) \tag{1.3}$$

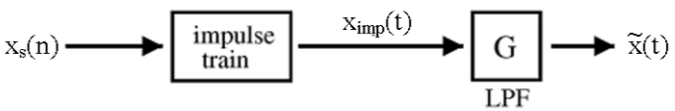


Figure 1.2 Block diagram for Signal Reconstruction

1.3.3 About Interpolation :

In the domain of digital signal processing, the term interpolation refers to the process of converting a sampled digital signal (such as a sampled audio signal) to that of a higher sampling rate (**Upsampling**) using various digital filtering techniques (for example, convolution with a frequency-limited impulse signal). In this application there is a specific requirement that the harmonic content of the original signal be preserved without creating aliased harmonic content of the original signal above the original **Nyquist** limit of the signal (that is, above $f_s/2$ of the original signal sample rate). There are different types of Interpolation techniques which are as follows :

- Nearest-neighbour interpolation
- Linear Interpolation
- Polynomial Interpolation
- Spline Interpolation

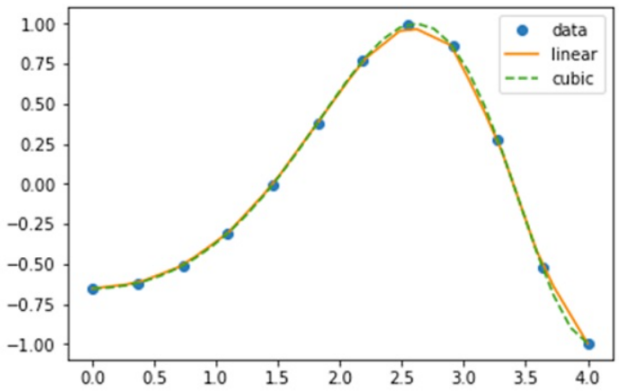


Figure 1.3 Linear and Cubic Interpolation

1.3.4 About Simulink :

Simulink is a MATLAB-based **graphical programming** environment for modeling, simulating and analyzing **multidomain** dynamical systems. Its primary interface is a graphical block diagramming tool and a customizable set of block libraries. It offers tight integration with the rest of the MATLAB environment and can either drive MATLAB or be scripted from it. Simulink is widely used in **automatic control** and **digital signal processing** for **multidomain** simulation and model-based design.

1.4 Code and results

1.4.1 Plot 5 cycles of input signal for different sampling frequencies :

```
% 19ucc023
% Mohit Akhouri
% Experiment 1 - Observation 1

clc;
clear all;
close all;

A = 1; % defining amplitude
n_cycles = 5; % defining number of cycles
fs_ideal = 100000; % defining ideal frequency
f = 3000; % defining message signal frequency

% generating ideal signal

n_ideal = 0:1:floor(n_cycles*(fs_ideal/f))-1;
x_ideal = A*cos(2*pi*f*n_ideal*(1/fs_ideal));

figure;
stem(n_ideal,x_ideal);
xlabel('samples(n) ->');
ylabel('amplitude ->');
title('19ucc023 - Mohit Akhouri','Generation of Ideal Signal for F_{s}
 = 100000 Hz');
grid on;

% generating sampled signals
fs1 = 10000;
ns1 = 0:1:floor(n_cycles*(fs1/f))-1;
x_sampled_1 = A*cos(2*pi*f*ns1*(1/fs1)); % sampled signal 1

fs2 = 6000;
ns2 = 0:1:floor(n_cycles*(fs2/f))-1;
x_sampled_2 = A*cos(2*pi*f*ns2*(1/fs2)); % sampled signal 2

fs3 = 12000;
ns3 = 0:1:floor(n_cycles*(fs3/f))-1;
x_sampled_3 = A*cos(2*pi*f*ns3*(1/fs3)); % sampled signal 3

fs4 = 4000;
ns4 = 0:1:floor(n_cycles*(fs4/f))-1;
x_sampled_4 = A*cos(2*pi*f*ns4*(1/fs4)); % sampled signal 4

fs5 = 5000;
ns5 = 0:1:floor(n_cycles*(fs5/f))-1;
x_sampled_5 = A*cos(2*pi*f*ns5*(1/fs5)); % sampled signal 5

% plotting the sampled signals
figure;
subplot(2,1,1);
stem(ns1,x_sampled_1);
xlabel('samples(n)->');
```

Figure 1.4 Part 1 of the code for observation 1

```
ylabel('amplitude->');
title('sampling for f_{s}=10000Hz');
grid on;

subplot(2,1,2);
stem(ns2,x_sampled_2);
xlabel('samples(n)->');
ylabel('amplitude->');
title('sampling for f_{s}=6000Hz');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');

figure;
subplot(2,1,1);
stem(ns3,x_sampled_3);
xlabel('samples(n)->');
ylabel('amplitude->');
title('sampling for f_{s}=12000Hz');
grid on;

subplot(2,1,2);
stem(ns4,x_sampled_4);
xlabel('samples(n)->');
ylabel('amplitude->');
title('sampling for f_{s}=4000Hz');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');

figure;
stem(ns5,x_sampled_5);
xlabel('samples(n)->');
ylabel('amplitude->');
title('19ucc023 - Mohit Akhouri','sampling for f_{s}=5000Hz');
grid on;
```

Figure 1.5 Part 2 of the code for observation 1

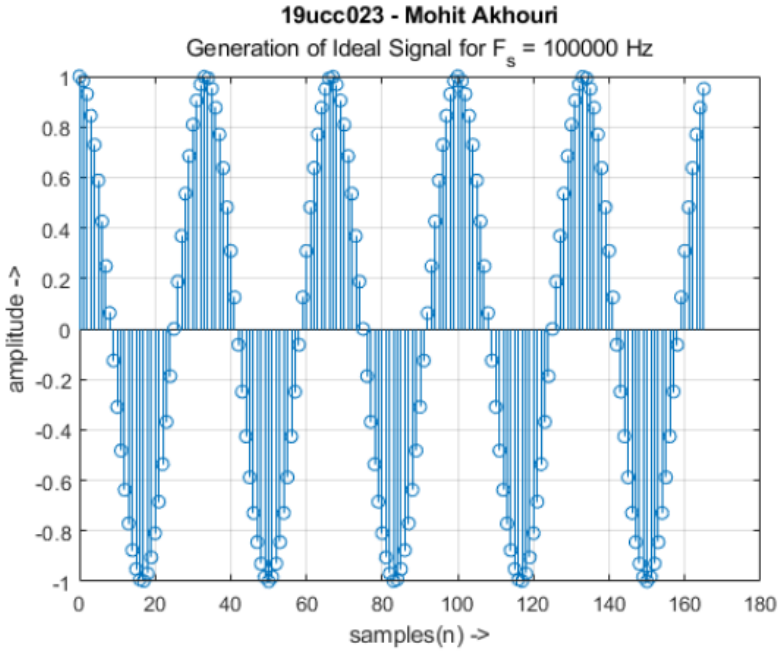


Figure 1.6 Ideal Signal For Sampling frequency = 100000 Hz

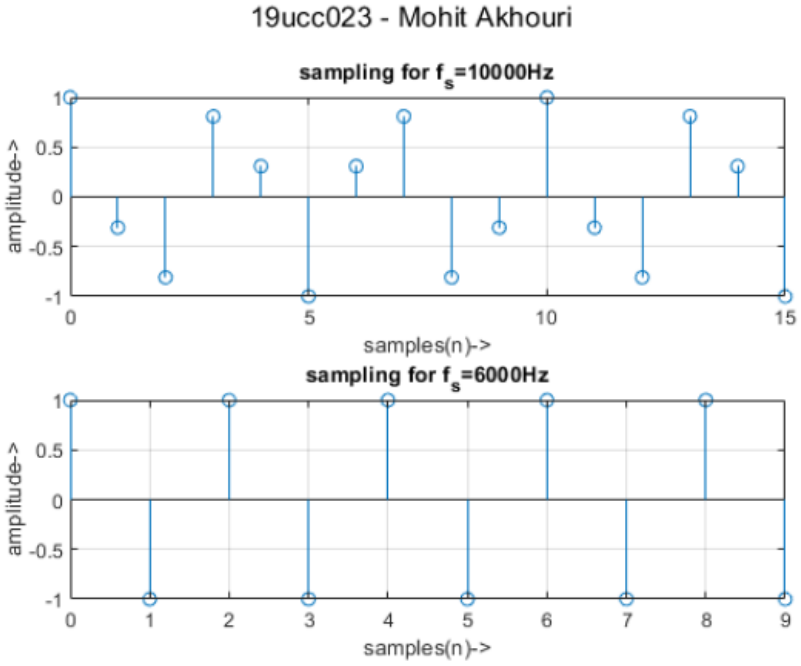


Figure 1.7 Plot of the sampled signals for F_s = 10000 Hz and 6000 Hz

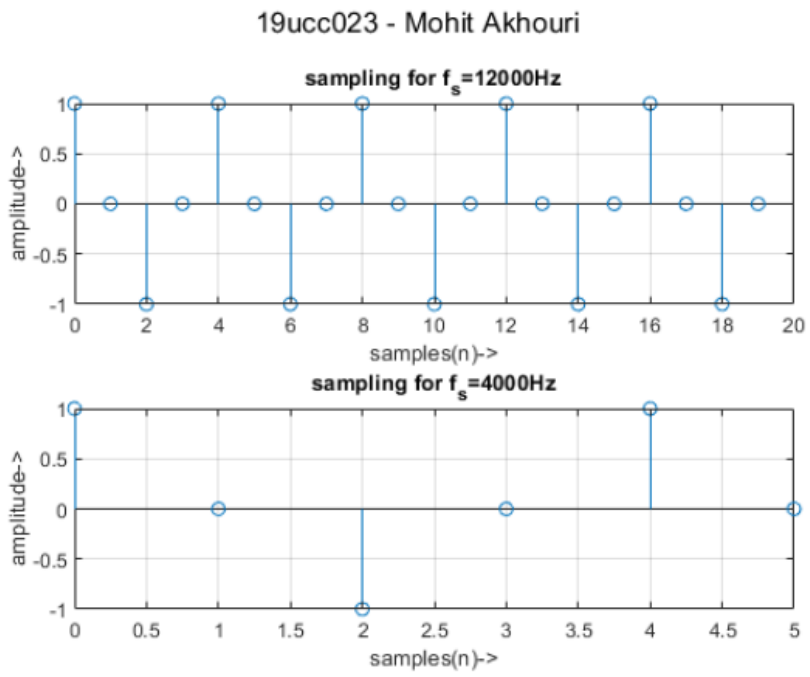


Figure 1.8 Plot of the sampled signals for F_s = 12000 Hz and 4000 Hz

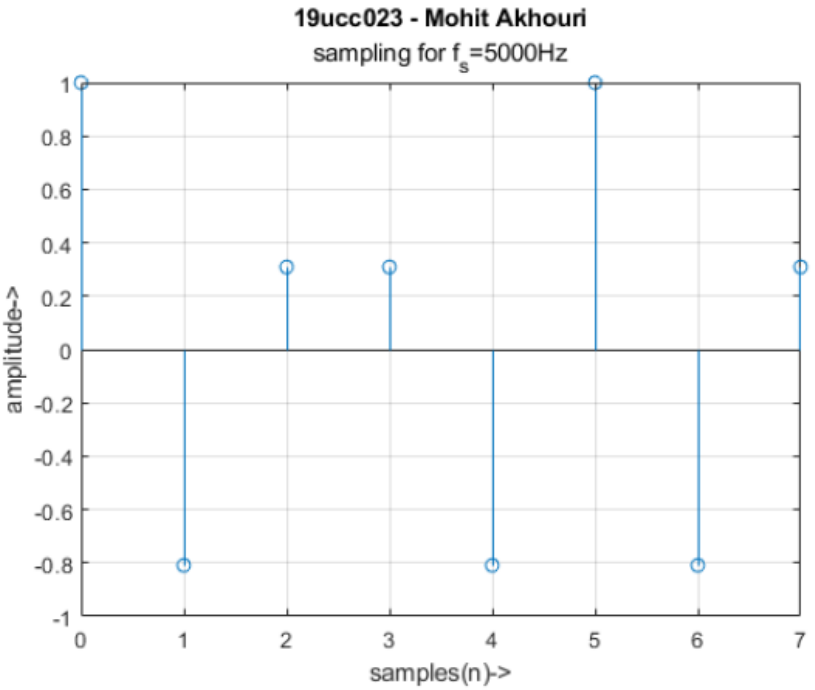


Figure 1.9 Plot of the sampled signal for F_s = 5000 Hz

1.4.2 Plot Fourier Spectra and determine the aliasing effect :

```
% 19ucc023
% Mohit Akhouri
% Experiment 1 - Observation 2

clc;
clear all;
close all;

A = 1; % defining amplitude
n_cycles = 5; % defining number of cycles
fs_ideal = 100000; % defining ideal frequency
f = 3000; % defining message signal frequency

N = 64; % defining number of samples
fs = N; % defining nyquist frequency
sf=linspace(-fs,fs,N); % defining frequency range

% generating ideal signal

n_ideal = 0:1:floor(n_cycles*(fs_ideal/f))-1;
x_ideal = A*cos(2*pi*f*n_ideal*(1/fs_ideal));

figure;
subplot(2,1,1);
stem(n_ideal,x_ideal);
xlabel('samples(n) ->');
ylabel('amplitude ->');
title('Generation of Ideal Signal for F_{s} = 100000 Hz');
grid on;

ft_ideal = abs(fftshift(fft(x_ideal,N))); % fourier transform of ideal
 signal

subplot(2,1,2);
stem(sf,ft_ideal);
xlabel('frequency(Hz)->');
ylabel('Amplitude->');
title('Fourier transform for the ideal signal with F_{s}=100000 Hz');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');


% generating sampled signals
fs1 = 10000;
ns1 = 0:1:floor(n_cycles*(fs1/f))-1;
x_sampled_1 = A*cos(2*pi*f*ns1*(1/fs1));

fs2 = 6000;
ns2 = 0:1:floor(n_cycles*(fs2/f))-1;
x_sampled_2 = A*cos(2*pi*f*ns2*(1/fs2));

fs3 = 12000;
```

Figure 1.10 Part 1 of the code for observation 2

```
ns3 = 0:1:floor(n_cycles*(fs3/f))-1;
x_sampled_3 = A*cos(2*pi*f*ns3*(1/fs3));

fs4 = 4000;
ns4 = 0:1:floor(n_cycles*(fs4/f))-1;
x_sampled_4 = A*cos(2*pi*f*ns4*(1/fs4));

fs5 = 5000;
ns5 = 0:1:floor(n_cycles*(fs5/f))-1;
x_sampled_5 = A*cos(2*pi*f*ns5*(1/fs5));

% generating the fourier transform of the sampled signals
ft_sample1 = abs(fftshift(fft(x_sampled_1,N))); % fourier transform of
 sampled signal with 10000Hz
ft_sample2 = abs(fftshift(fft(x_sampled_2,N))); % fourier transform of
 sampled signal with 6000Hz
ft_sample3 = abs(fftshift(fft(x_sampled_3,N))); % fourier transform of
 sampled signal with 12000Hz
ft_sample4 = abs(fftshift(fft(x_sampled_4,N))); % fourier transform of
 sampled signal with 4000Hz
ft_sample5 = abs(fftshift(fft(x_sampled_5,N))); % fourier transform of
 sampled signal with 5000Hz

% plotting the sampled signals and their fourier transforms
figure;
subplot(2,1,1);
stem(ns1,x_sampled_1);
xlabel('samples(n)->');
ylabel('amplitude->');
title('sampling for f_{s}=10000Hz');
grid on;

subplot(2,1,2);
stem(sf,ft_sample1);
xlabel('frequency(Hz)->');
ylabel('Amplitude->');
title('Fourier transform for the sampled signal 1 with F_{s}=10000
 Hz');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');

figure;
subplot(2,1,1);
stem(ns2,x_sampled_2);
xlabel('samples(n)->');
ylabel('amplitude->');
title('sampling for f_{s}=6000Hz');
grid on;

subplot(2,1,2);
stem(sf,ft_sample2);
xlabel('frequency(Hz)->');
ylabel('Amplitude->');
```

Figure 1.11 Part 2 of the code for observation 2

```
title('Fourier transform for the sampled signal 2 with F_{s}=6000
 Hz');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');

figure;
subplot(2,1,1);
stem(ns3,x_sampled_3);
xlabel('samples(n)->');
ylabel('amplitude->');
title('sampling for f_{s}=12000Hz');
grid on;

subplot(2,1,2);
stem(sf,ft_sample3);
xlabel('frequency(Hz)->');
ylabel('Amplitude->');
title('Fourier transform for the sampled signal 3 with F_{s}=12000
 Hz');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');

figure;
subplot(2,1,1);
stem(ns4,x_sampled_4);
xlabel('samples(n)->');
ylabel('amplitude->');
title('sampling for f_{s}=4000Hz');
grid on;

subplot(2,1,2);
stem(sf,ft_sample4);
xlabel('frequency(Hz)->');
ylabel('Amplitude->');
title('Fourier transform for the sampled signal 4 with F_{s}=4000
 Hz');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');

figure;
subplot(2,1,1);
stem(ns5,x_sampled_5);
xlabel('samples(n)->');
ylabel('amplitude->');
title('sampling for f_{s}=5000Hz');
grid on;

subplot(2,1,2);
stem(sf,ft_sample5);
xlabel('frequency(Hz)->');
ylabel('Amplitude->');
title('Fourier transform for the sampled signal 5 with F_{s}=5000
 Hz');
grid on;
```

Figure 1.12 Part 3 of the code for observation 2

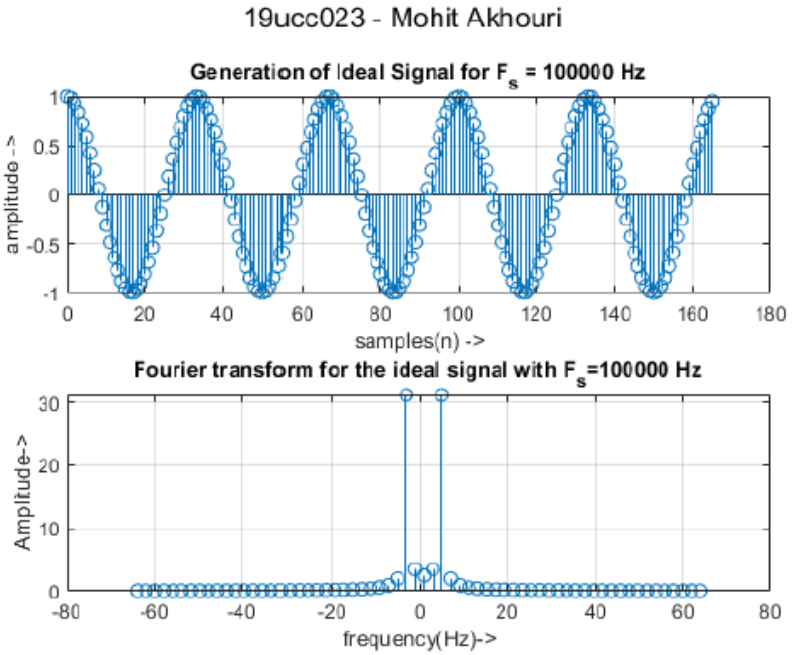


Figure 1.13 Fourier Transform for the Ideal signal with F_s = 100000 Hz

Figure 1.14 Fourier Transform for the Sampled signal with F_s = 10000 Hz

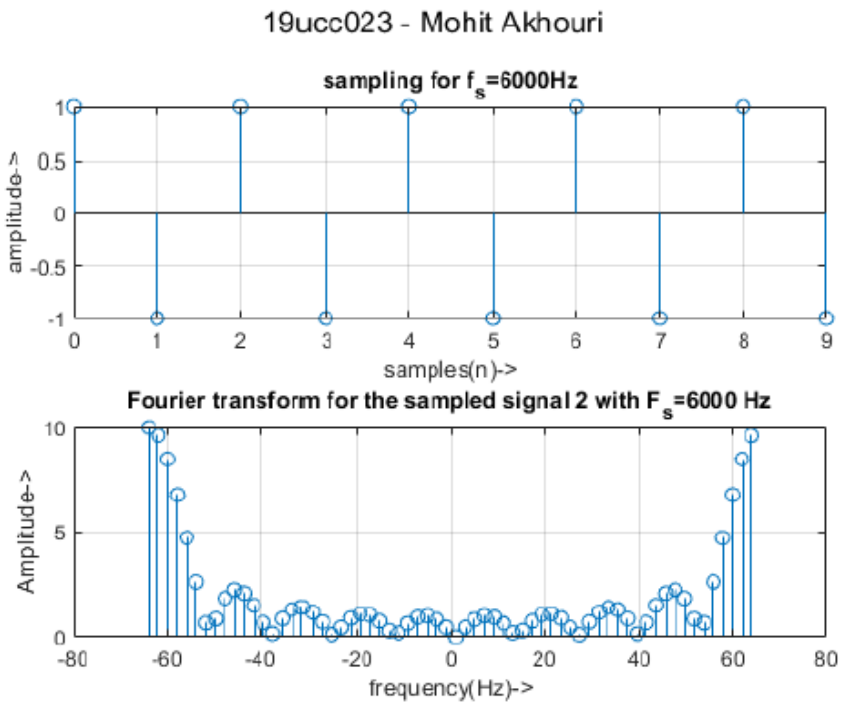


Figure 1.15 Fourier Transform for the Sampled signal with F_s = 6000 Hz

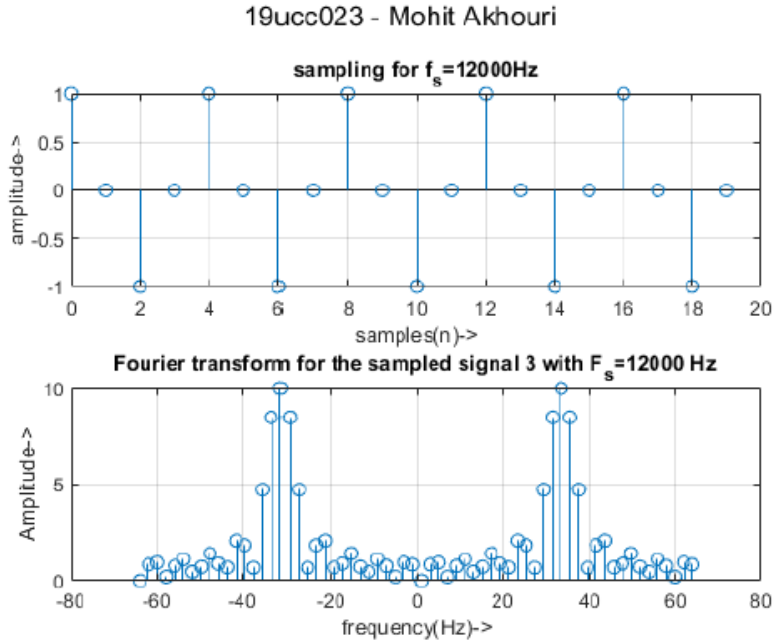


Figure 1.16 Fourier Transform for the Sampled signal with F_s = 12000 Hz

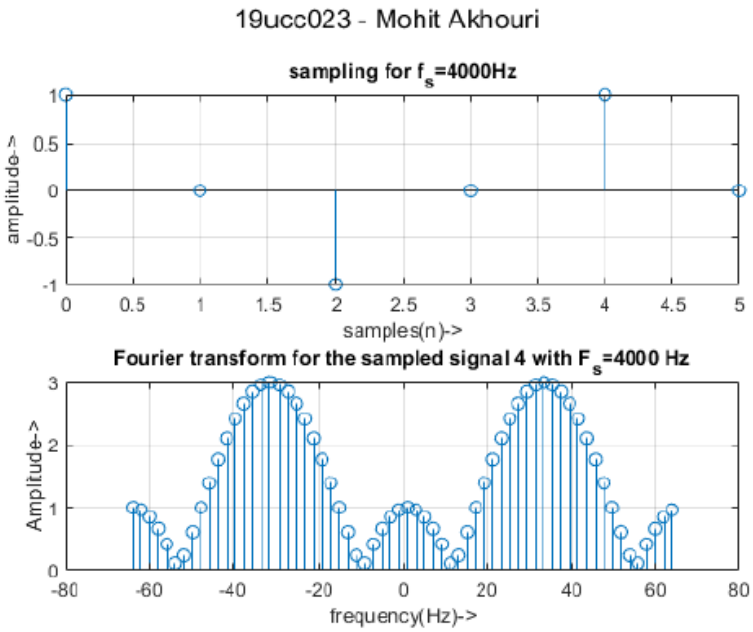


Figure 1.17 Fourier Transform for the Sampled signal with F_s = 4000 Hz

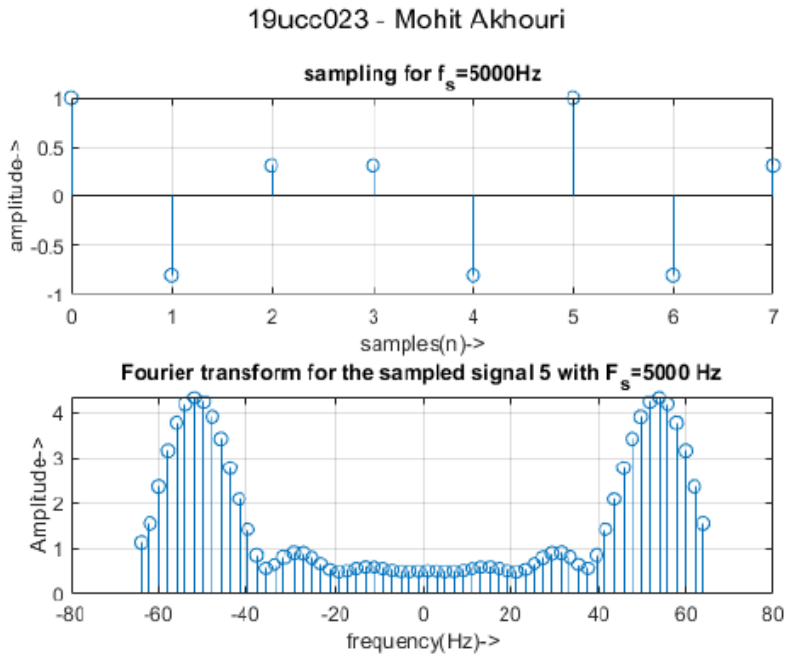


Figure 1.18 Fourier Transform for the Sampled signal with F_s = 5000 Hz

1.4.3 Find MSE for linear interpolation with different sampling rates (Plot MSE vs. F_s) :

```
% 19ucc023
% Mohit Akhouri
% Experiment 1 - Observation 3

clc;
clear all;
close all;

A = 1; % defining amplitude
n_cycles = 5; % defining number of cycles
fs_ideal = 100000; % defining ideal frequency
f = 3000; % defining message signal frequency

% generating ideal signal

n_ideal = 0:1:floor(n_cycles*(fs_ideal/f))-1;
x_ideal = A*cos(2*pi*f*n_ideal*(1/fs_ideal));

% generating sampled signals
fs1 = 10000;
ns1 = 0:1:floor(n_cycles*(fs1/f))-1;
x_sampled_1 = A*cos(2*pi*f*ns1*(1/fs1));

fs2 = 6000;
ns2 = 0:1:floor(n_cycles*(fs2/f))-1;
x_sampled_2 = A*cos(2*pi*f*ns2*(1/fs2));

fs3 = 12000;
ns3 = 0:1:floor(n_cycles*(fs3/f))-1;
x_sampled_3 = A*cos(2*pi*f*ns3*(1/fs3));

fs4 = 4000;
ns4 = 0:1:floor(n_cycles*(fs4/f))-1;
x_sampled_4 = A*cos(2*pi*f*ns4*(1/fs4));

fs5 = 5000;
ns5 = 0:1:floor(n_cycles*(fs5/f))-1;
x_sampled_5 = A*cos(2*pi*f*ns5*(1/fs5));

% generating the parameters x and xq for use in interp1 function for
% different sampling rates
x_s1 = 0:1:max(ns1);
xq_s1 = 0:max(ns1)/max(n_ideal):max(ns1);

x_s2 = 0:1:max(ns2);
xq_s2 = 0:max(ns2)/max(n_ideal):max(ns2);

x_s3 = 0:1:max(ns3);
xq_s3 = 0:max(ns3)/max(n_ideal):max(ns3);

x_s4 = 0:1:max(ns4);
xq_s4 = 0:max(ns4)/max(n_ideal):max(ns4);
```

Figure 1.19 Part 1 of the code for observation 3

```
x_s5 = 0:1:max(ns5);
xq_s5 = 0:max(ns5)/max(n_ideal):max(ns5);

inter_method = 'linear'; % using linear interpolation method

inter_s1 = interp1(x_s1,x_sampled_1,xq_s1,inter_method); %
 interpolated signal for Fs = 10000 Hz
inter_s2 = interp1(x_s2,x_sampled_2,xq_s2,inter_method); %
 interpolated signal for Fs = 6000 Hz
inter_s3 = interp1(x_s3,x_sampled_3,xq_s3,inter_method); %
 interpolated signal for Fs = 12000 Hz
inter_s4 = interp1(x_s4,x_sampled_4,xq_s4,inter_method); %
 interpolated signal for Fs = 4000 Hz
inter_s5 = interp1(x_s5,x_sampled_5,xq_s5,inter_method); %
 interpolated signal for Fs = 5000 Hz

mse_s1 = mean((x_ideal - inter_s1).^2); % MSE for Fs = 10000 Hz
mse_s2 = mean((x_ideal - inter_s2).^2); % MSE for Fs = 6000 Hz
mse_s3 = mean((x_ideal - inter_s3).^2); % MSE for Fs = 12000 Hz
mse_s4 = mean((x_ideal - inter_s4).^2); % MSE for Fs = 4000 Hz
mse_s5 = mean((x_ideal - inter_s5).^2); % MSE for Fs = 5000 Hz

% plotting the interpolated signals for different sampling frequencies
figure;
plot(inter_s1);
xlabel('samples(n)->');
ylabel('amplitude->');
title('19ucc023 - Mohit Akhouri','interpolated signal for F_{s}=10000
 Hz');
grid on;

figure;
plot(inter_s2);
xlabel('samples(n)->');
ylabel('amplitude->');
title('19ucc023 - Mohit Akhouri','interpolated signal for F_{s}=6000
 Hz');
grid on;

figure;
plot(inter_s3);
xlabel('samples(n)->');
ylabel('amplitude->');
title('19ucc023 - Mohit Akhouri','interpolated signal for F_{s}=12000
 Hz');
grid on;

figure;
plot(inter_s4);
xlabel('samples(n)->');
ylabel('amplitude->');
title('19ucc023 - Mohit Akhouri','interpolated signal for F_{s}=4000
 Hz');
```

Figure 1.20 Part 2 of the code for observation 3

```
grid on;

figure;
plot(inter_s5);
xlabel('samples(n)->');
ylabel('amplitude->');
title('19ucc023 - Mohit Akhouri','interpolated signal for F_{s}=5000
 Hz');
grid on;

% plotting the MSE vs. Fs graph for LINEAR INTERPOLATION METHOD
figure;
stem(fs1,mse_s1,'Linewidth',1.5);
hold on;
stem(fs2,mse_s2,'Linewidth',1.5);
hold on;
stem(fs3,mse_s3,'Linewidth',1.5);
hold on;
stem(fs4,mse_s4,'Linewidth',1.5);
hold on;
stem(fs5,mse_s5,'Linewidth',1.5);

xlabel('F_{s} ->');
ylabel('MSE->');
title('19ucc023 - Mohit Akhouri','MSE vs. F_{s} for different sampling
 rates');
grid on;
```

Figure 1.21 Part 3 of the code for observation 3

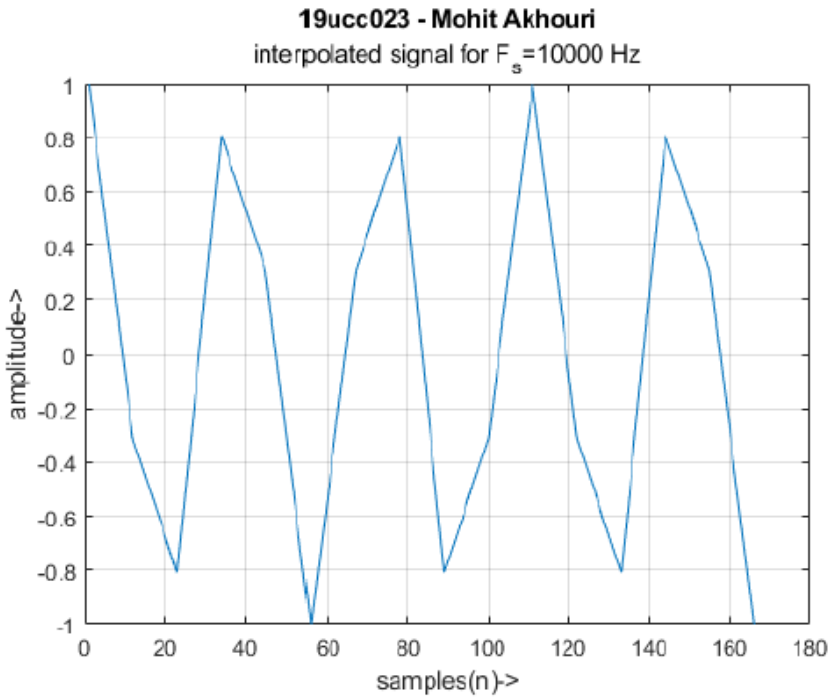


Figure 1.22 Linear Interpolation for the sampled signal with F_s = 10000 Hz

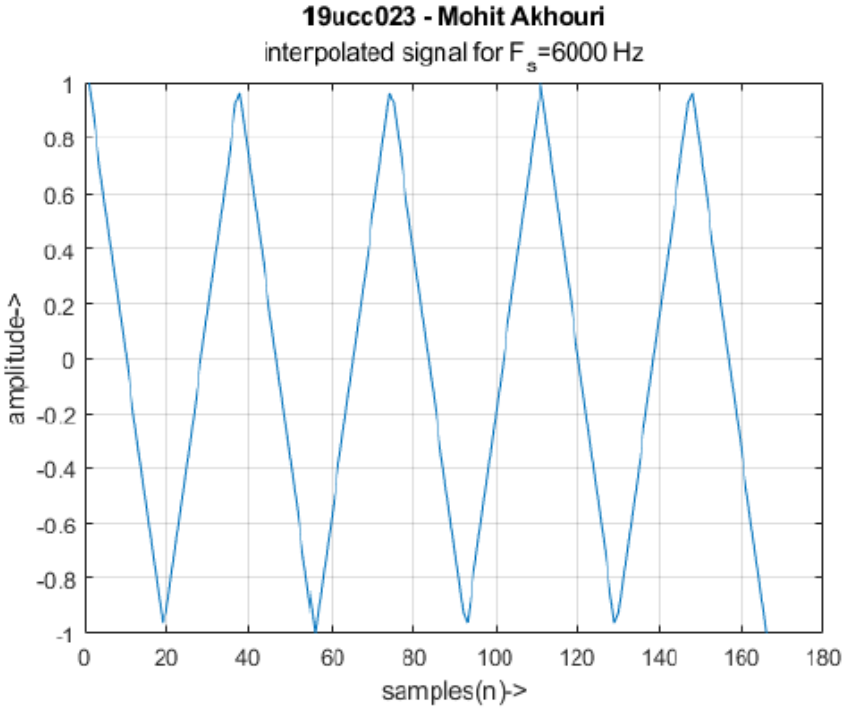


Figure 1.23 Linear Interpolation for the sampled signal with F_s = 6000 Hz

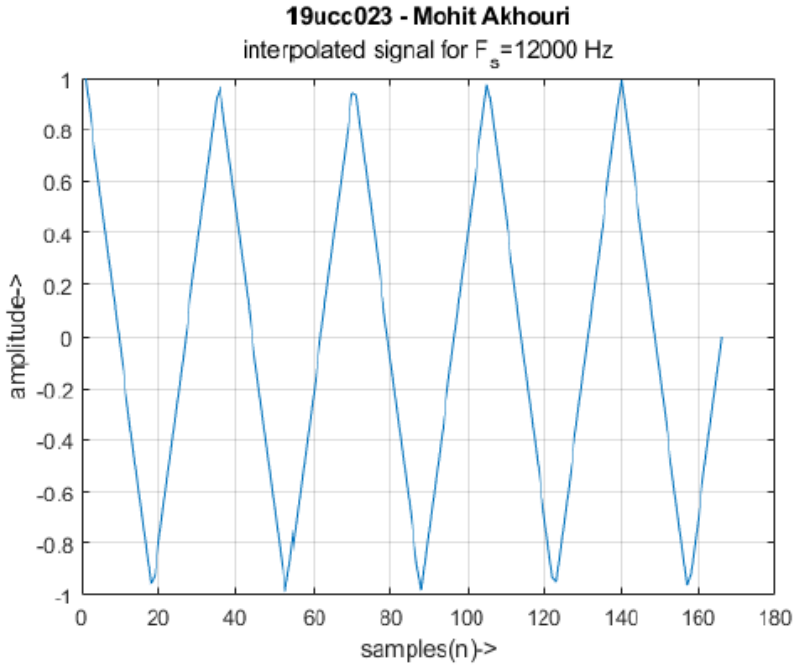


Figure 1.24 Linear Interpolation for the sampled signal with F_s = 12000 Hz

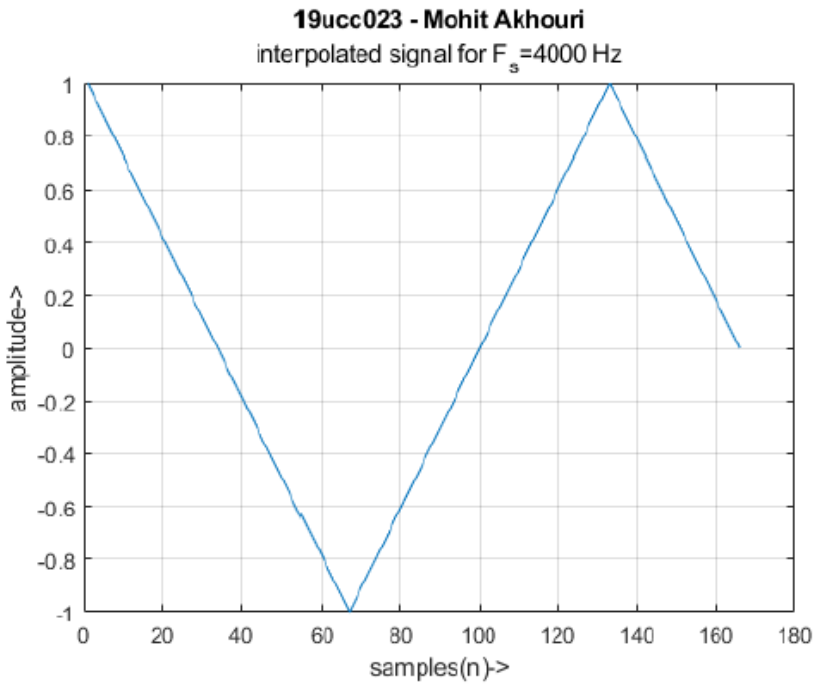


Figure 1.25 Linear Interpolation for the sampled signal with F_s = 4000 Hz

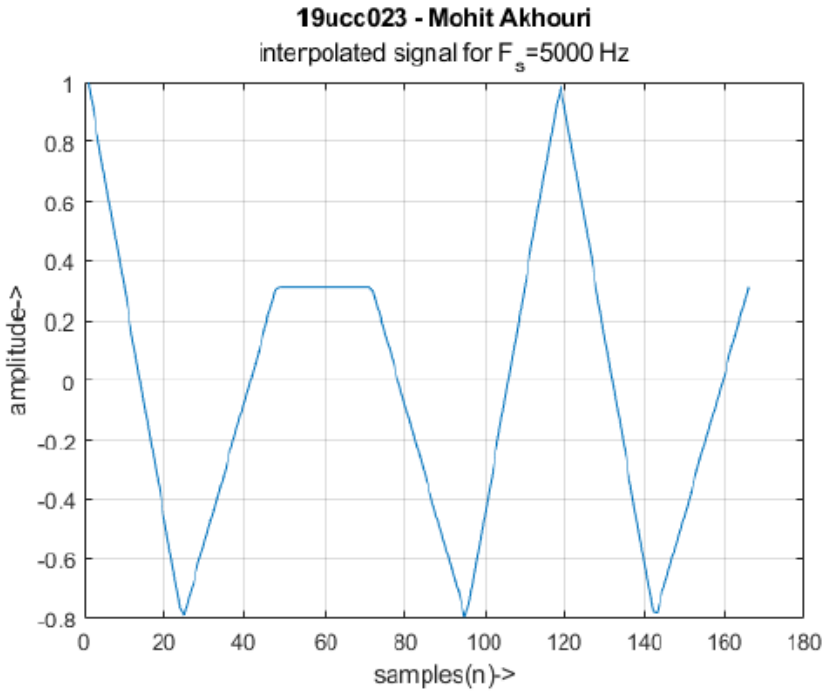


Figure 1.26 Linear Interpolation for the sampled signal with F_s = 5000 Hz

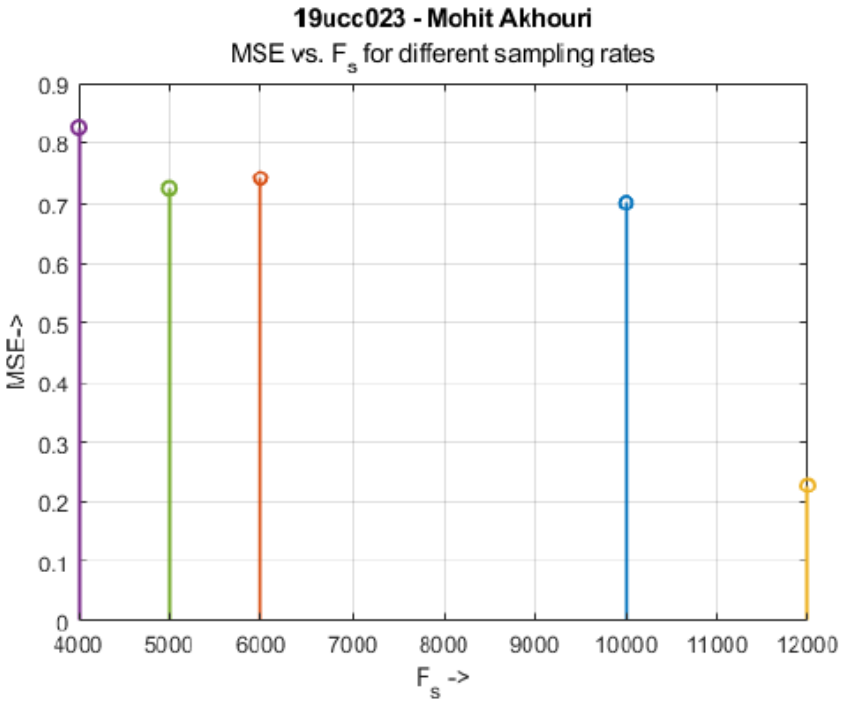


Figure 1.27 MSE vs. F_s for Linear Interpolation

1.4.4 Repeat MSE vs. F_s for different interpolation techniques :

```
% 19ucc023
% Mohit Akhouri
% Experiment 1 - Observation 4

clc;
clear all;
close all;

A = 1; % defining amplitude
n_cycles = 5; % defining number of cycles
fs_ideal = 100000; % defining ideal frequency
f = 3000; % defining message signal frequency

% generating ideal signal

n_ideal = 0:1:floor(n_cycles*(fs_ideal/f))-1;
x_ideal = A*cos(2*pi*f*n_ideal*(1/fs_ideal));

% generating sampled signals
fs1 = 10000;
ns1 = 0:1:floor(n_cycles*(fs1/f))-1;
x_sampled_1 = A*cos(2*pi*f*ns1*(1/fs1));

fs2 = 6000;
ns2 = 0:1:floor(n_cycles*(fs2/f))-1;
x_sampled_2 = A*cos(2*pi*f*ns2*(1/fs2));

fs3 = 12000;
ns3 = 0:1:floor(n_cycles*(fs3/f))-1;
x_sampled_3 = A*cos(2*pi*f*ns3*(1/fs3));

fs4 = 4000;
ns4 = 0:1:floor(n_cycles*(fs4/f))-1;
x_sampled_4 = A*cos(2*pi*f*ns4*(1/fs4));

fs5 = 5000;
ns5 = 0:1:floor(n_cycles*(fs5/f))-1;
x_sampled_5 = A*cos(2*pi*f*ns5*(1/fs5));

% generating the parameters x and xq for use in interp1 function for
% different sampling rates
x_s1 = 0:1:max(ns1);
xq_s1 = 0:max(ns1)/max(n_ideal):max(ns1);

x_s2 = 0:1:max(ns2);
xq_s2 = 0:max(ns2)/max(n_ideal):max(ns2);

x_s3 = 0:1:max(ns3);
xq_s3 = 0:max(ns3)/max(n_ideal):max(ns3);

x_s4 = 0:1:max(ns4);
xq_s4 = 0:max(ns4)/max(n_ideal):max(ns4);
```

Figure 1.28 Part 1 of the code for observation 4

```
x_s5 = 0:1:max(ns5);
xq_s5 = 0:max(ns5)/max(n_ideal):max(ns5);

inter_method1 = 'linear';
inter_method2 = 'spline';
inter_method3 = 'cubic';

% calculating MSE vs. Fs for linear interpolation
inter_s1_m1 = interp1(x_s1,x_sampled_1,xq_s1,inter_method1);
inter_s2_m1 = interp1(x_s2,x_sampled_2,xq_s2,inter_method1);
inter_s3_m1 = interp1(x_s3,x_sampled_3,xq_s3,inter_method1);
inter_s4_m1 = interp1(x_s4,x_sampled_4,xq_s4,inter_method1);
inter_s5_m1 = interp1(x_s5,x_sampled_5,xq_s5,inter_method1);

mse_s1_m1 = mean((x_ideal - inter_s1_m1).^2);
mse_s2_m1 = mean((x_ideal - inter_s2_m1).^2);
mse_s3_m1 = mean((x_ideal - inter_s3_m1).^2);
mse_s4_m1 = mean((x_ideal - inter_s4_m1).^2);
mse_s5_m1 = mean((x_ideal - inter_s5_m1).^2);

figure;
stem(fs1,mse_s1_m1,'Linewidth',1.5);
hold on;
stem(fs2,mse_s2_m1,'Linewidth',1.5);
hold on;
stem(fs3,mse_s3_m1,'Linewidth',1.5);
hold on;
stem(fs4,mse_s4_m1,'Linewidth',1.5);
hold on;
stem(fs5,mse_s5_m1,'Linewidth',1.5);

xlabel('F_{s} ->');
ylabel('MSE->');
title('19ucc023 - Mohit Akhouri','MSE vs. F_{s} for different sampling
 rates for LINEAR INTERPOLATION');
grid on;


% calculating MSE vs. Fs for spline interpolation
inter_s1_m2 = interp1(x_s1,x_sampled_1,xq_s1,inter_method2);
inter_s2_m2 = interp1(x_s2,x_sampled_2,xq_s2,inter_method2);
inter_s3_m2 = interp1(x_s3,x_sampled_3,xq_s3,inter_method2);
inter_s4_m2 = interp1(x_s4,x_sampled_4,xq_s4,inter_method2);
inter_s5_m2 = interp1(x_s5,x_sampled_5,xq_s5,inter_method2);

mse_s1_m2 = mean((x_ideal - inter_s1_m2).^2);
mse_s2_m2 = mean((x_ideal - inter_s2_m2).^2);
mse_s3_m2 = mean((x_ideal - inter_s3_m2).^2);
mse_s4_m2 = mean((x_ideal - inter_s4_m2).^2);
mse_s5_m2 = mean((x_ideal - inter_s5_m2).^2);

figure;
stem(fs1,mse_s1_m2,'Linewidth',1.5);
```

Figure 1.29 Part 2 of the code for observation 4

```
hold on;
stem(fs2,mse_s2_m2,'Linewidth',1.5);
hold on;
stem(fs3,mse_s3_m2,'Linewidth',1.5);
hold on;
stem(fs4,mse_s4_m2,'Linewidth',1.5);
hold on;
stem(fs5,mse_s5_m2,'Linewidth',1.5);

xlabel('F_{s} ->');
ylabel('MSE->');
title('19ucc023 - Mohit Akhouri','MSE vs. F_{s} for different sampling
 rates for SPLINE INTERPOLATION');
grid on;


% calculating MSE vs. Fs for cubic interpolation
inter_s1_m3 = interp1(x_s1,x_sampled_1,xq_s1,inter_method3);
inter_s2_m3 = interp1(x_s2,x_sampled_2,xq_s2,inter_method3);
inter_s3_m3 = interp1(x_s3,x_sampled_3,xq_s3,inter_method3);
inter_s4_m3 = interp1(x_s4,x_sampled_4,xq_s4,inter_method3);
inter_s5_m3 = interp1(x_s5,x_sampled_5,xq_s5,inter_method3);

mse_s1_m3 = mean((x_ideal - inter_s1_m3).^2);
mse_s2_m3 = mean((x_ideal - inter_s2_m3).^2);
mse_s3_m3 = mean((x_ideal - inter_s3_m3).^2);
mse_s4_m3 = mean((x_ideal - inter_s4_m3).^2);
mse_s5_m3 = mean((x_ideal - inter_s5_m3).^2);

figure;
stem(fs1,mse_s1_m3,'Linewidth',1.5);
hold on;
stem(fs2,mse_s2_m3,'Linewidth',1.5);
hold on;
stem(fs3,mse_s3_m3,'Linewidth',1.5);
hold on;
stem(fs4,mse_s4_m3,'Linewidth',1.5);
hold on;
stem(fs5,mse_s5_m3,'Linewidth',1.5);

xlabel('F_{s} ->');
ylabel('MSE->');
title('19ucc023 - Mohit Akhouri','MSE vs. F_{s} for different sampling
 rates for CUBIC INTERPOLATION');
grid on;
```

Figure 1.30 Part 3 of the code for observation 4

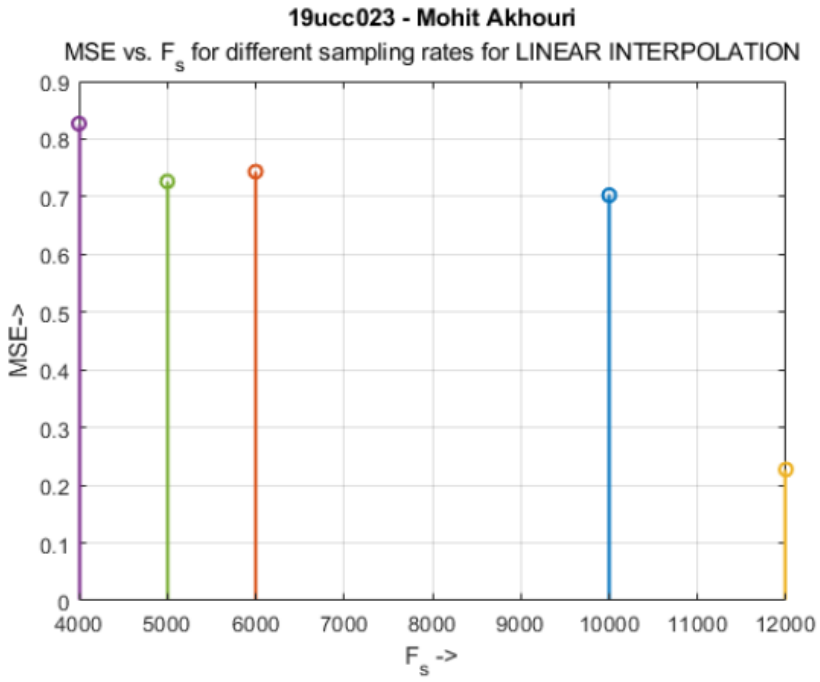


Figure 1.31 MSE vs. F_s for Linear interpolation

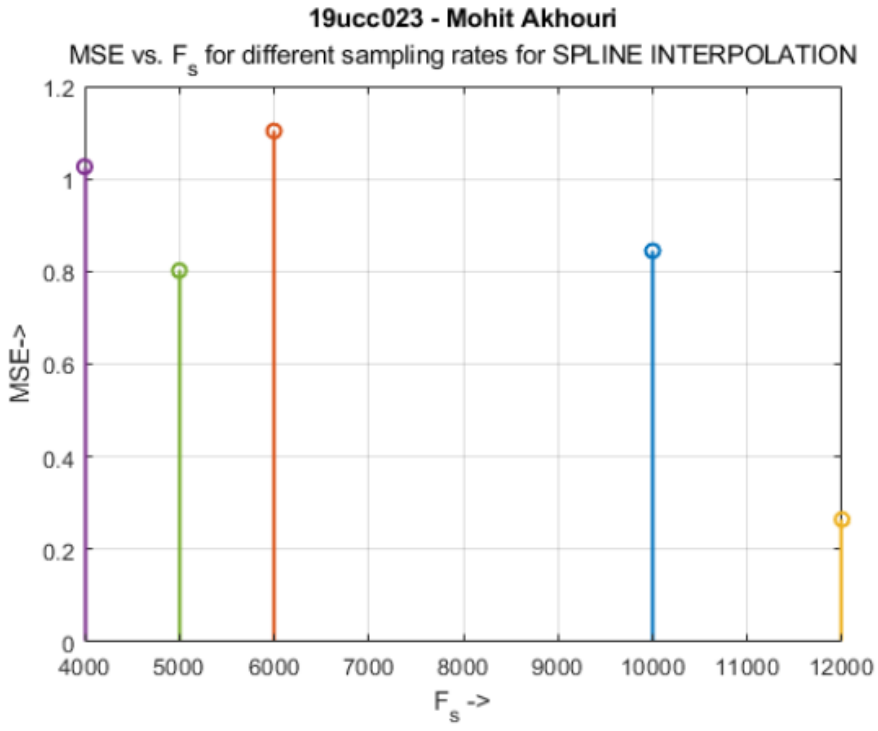
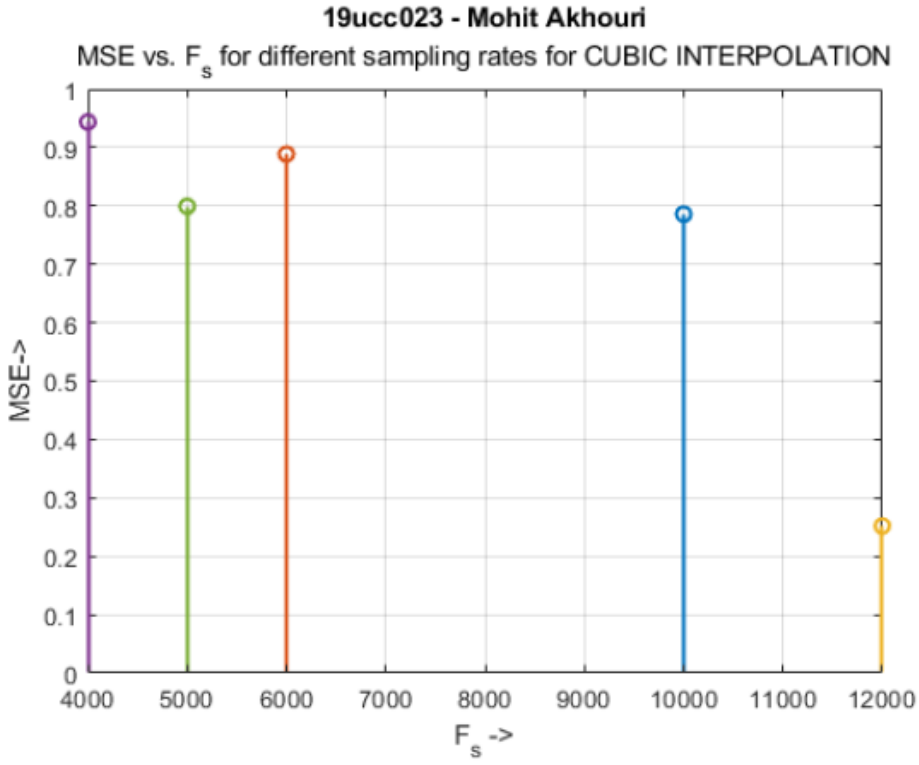


Figure 1.32 MSE vs. F_s for Spline interpolation



Published with MATLAB® R2020b

Figure 1.33 MSE vs. F_s for Cubic interpolation

1.4.5 Perform sampling and interpolation in simulink :

```
% 19ucc023
% Mohit Akhouri
% Experiment 1 - Observation 5

sim('observation1_simulink'); % calling the simulink model

figure;
subplot(2,3,1);
stem(out.Ideal_signal.data);
xlabel('samples(n)->');
ylabel('amplitude->');
title('Ideal signal');
grid on;

subplot(2,3,2);
stem(out.x_sampled_1.data);
xlabel('samples(n)->');
ylabel('amplitude->');
title('sampled signal for fs = 10000 Hz');
grid on;

subplot(2,3,3);
stem(out.x_sampled_2.data);
xlabel('samples(n)->');
ylabel('amplitude->');
title('sampled signal for fs = 6000 Hz');
grid on;

subplot(2,3,4);
stem(out.x_sampled_3.data);
xlabel('samples(n)->');
ylabel('amplitude->');
title('sampled signal for fs = 12000 Hz');
grid on;

subplot(2,3,5);
stem(out.x_sampled_4.data);
xlabel('samples(n)->');
ylabel('amplitude->');
title('sampled signal for fs = 4000 Hz');
grid on;

subplot(2,3,6);
stem(out.x_sampled_5.data);
xlabel('samples(n)->');
ylabel('amplitude->');
title('sampled signal for fs = 5000 Hz');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');

% plotting MSE vs. Fs for different sampling rates through linear
% interpolation techniques
```

Figure 1.34 Part 1 of the code for observation 5

```
fs1 = 10000;
fs2 = 6000;
fs3 = 12000;
fs4 = 4000;
fs5 = 5000;
figure;
stem(fs1,out.mse_s1.data,'Linewidth',1.5);
hold on;
stem(fs2,out.mse_s2.data,'Linewidth',1.5);
hold on;
stem(fs3,out.mse_s3.data,'Linewidth',1.5);
hold on;
stem(fs4,out.mse_s4.data,'Linewidth',1.5);
hold on;
stem(fs5,out.mse_s5.data,'Linewidth',1.5);

xlabel('F_{s} ->');
ylabel('MSE->');
title('19ucc023 - Mohit Akhouri','MSE vs. F_{s} for different sampling
 rates using LINEAR INTERPOLATION');
grid on;
```

Figure 1.35 Part 2 of the code for observation 5

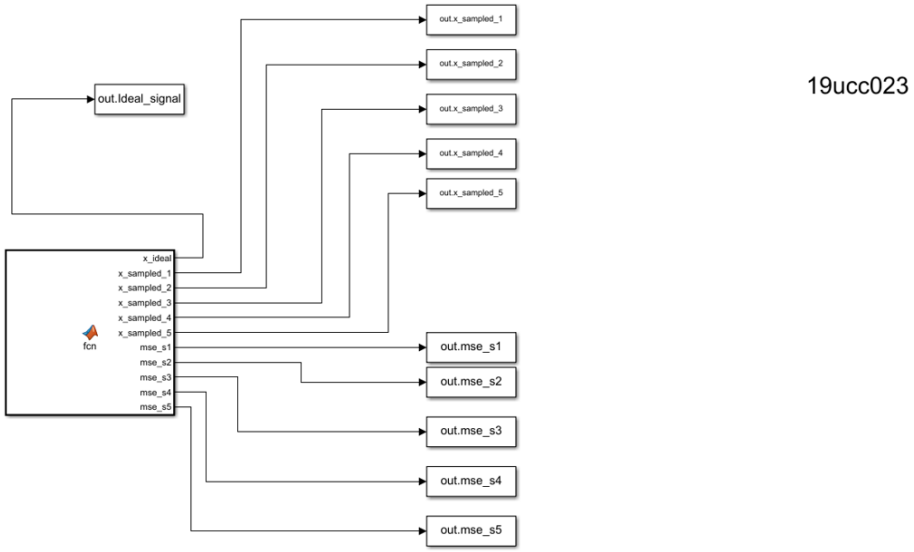


Figure 1.36 Simulink Block Diagram for observation 5 for sampling and interpolation

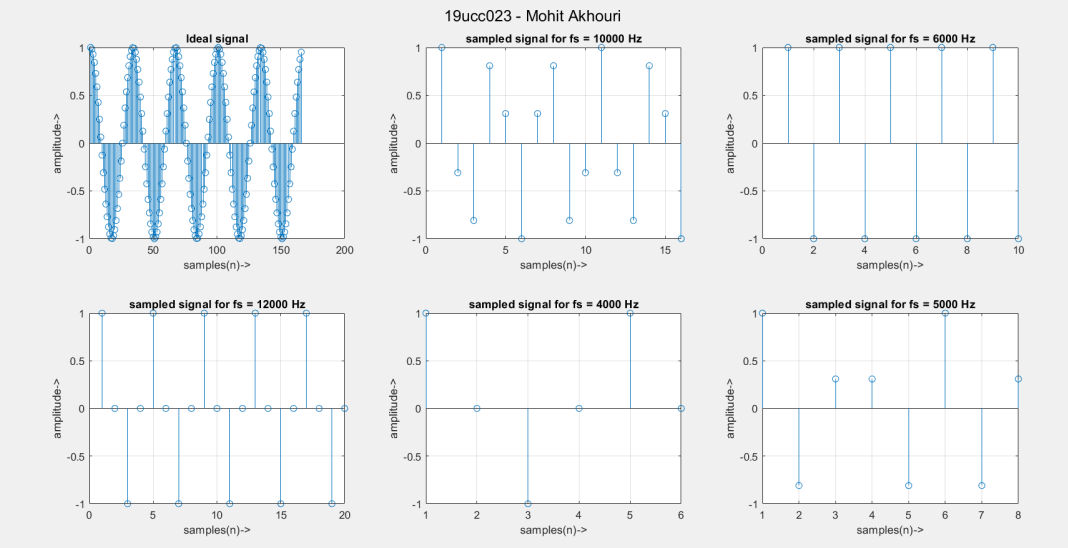
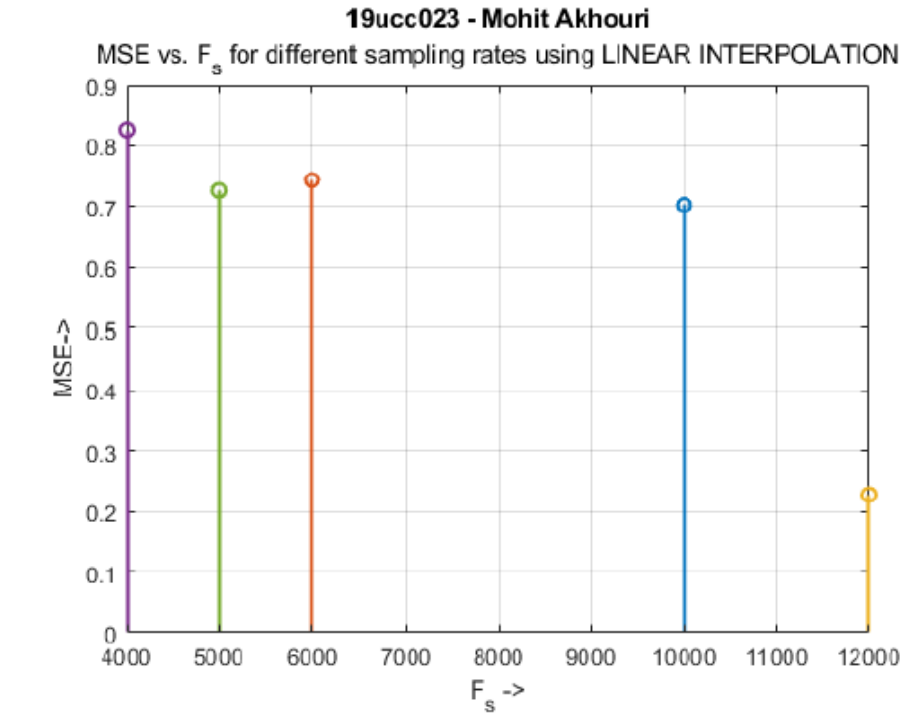


Figure 1.37 Plot of the sampling done via Simulink Model



Published with MATLAB® R2020b

Figure 1.38 MSE vs. F_s for linear interpolation done via Simulink Model

1.5 Conclusion

In this experiment, we learnt about the two concepts of Digital Signal Processing which are **Signal Sampling** and **Signal Reconstruction via Interpolation** . We also learnt how to write code for the above methods in MATLAB and how to create models for the same using **Simulink**. In simulink , we learnt about two blocks - **MATLAB function** and **To workspace**. We plotted the MSE vs. F_s curves for different sampling rates. We also learnt about various Interpolation techniques such as **Linear** , **Spline** and **Cubic** Interpolation. At Last , we concluded that **Cubic interpolation** is **better** than other interpolation techniques due to the **smoothness** of the function and **higher accuracy** in achieving the original function.

Chapter *2*

Experiment - 2

2.1 Aim of the Experiment

- Quantization and Encoding
- Simulink based Quantization

2.2 Software Used

- MATLAB
- Simulink

2.3 Theory

2.3.1 About Quantization :

Quantization, in mathematics and digital signal processing, is the process of **mapping** input values from a large set (often a continuous set) to output values in a **countable smaller set**, often with a finite number of elements. **Rounding** and **truncation** are typical examples of quantization processes.Quantization is involved to some degree in nearly all digital signal processing, as the process of representing a signal in digital form ordinarily involves rounding. Quantization also forms the core of essentially all **lossy compression** algorithms. There are different types of Quantizer which are as follows :

- Analog-to-digital converter
- Rate–distortion optimization
- Mid-riser and mid-tread uniform quantizers
- Dead-zone quantizer

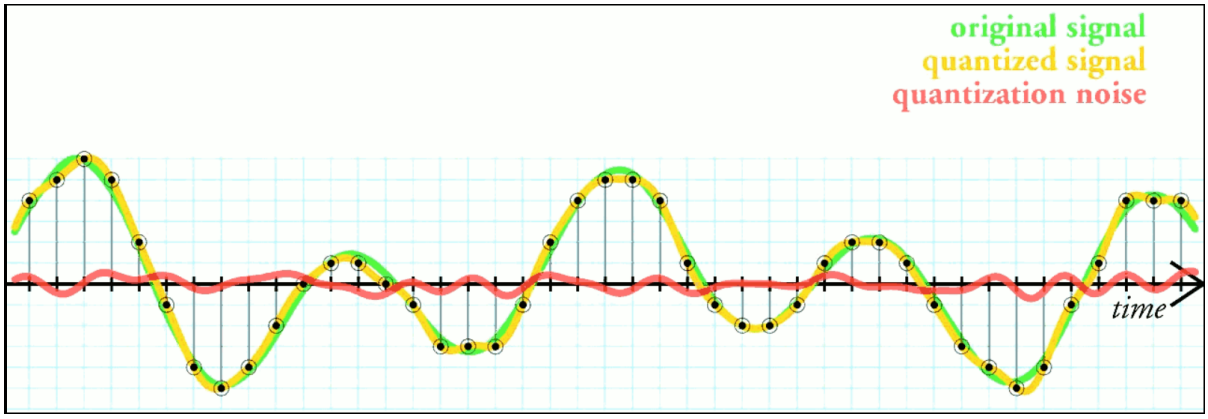


Figure 2.1 Original signal, Quantized signal and Quantization Noise

Quantized value can be calculated by first calculating the **index** value (i) which can be done as follows :

$$i = round(\frac{x - x_{min}}{\Delta}) \tag{2.1}$$

Now the **Quantization level** (x_q) of a particular sample can be calculated as:

$$x_q = x_{min} + i\Delta \tag{2.2}$$

$$\Delta = \frac{x_{max} - x_{min}}{L} \tag{2.3}$$

In the above equations , **x** is the original signal , x_{max} is the maximum value of the original signal, x_{min} is the minimum value of the original signal, Δ is the **step size** and **L** is the **number of levels** provided to the quantizer.

2.3.1.1 About Encoding :

The process of representing quantized values digitally is called **encoding**. The quantized value is coded into binary form. Typically, each binary digit is assigned an output line. At read-out time, these lines carry a 0 or 1. The number of digits used to represent each quantized value can be given as:

$$b >= log(L) \tag{2.4}$$

where **b** is the number of digits used to represent the quantized value and **L** is the number of levels.

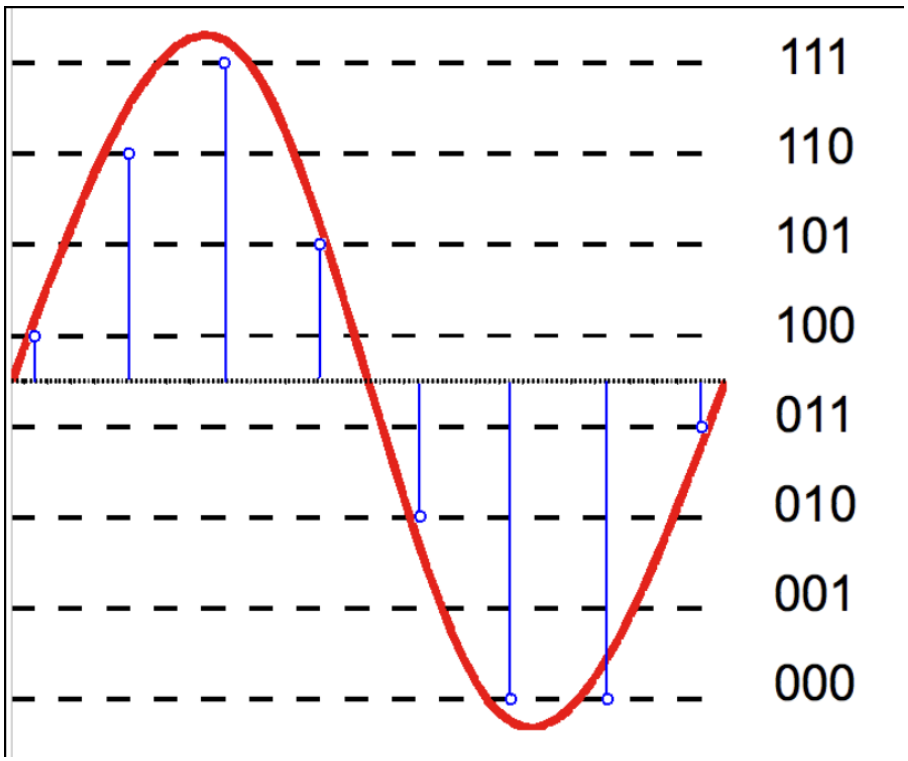


Figure 2.2 Quantized and Encoded signal for L=8

2.3.2 About Quantization Noise Error :

When an Analog-Digital Converter (ADC) converts a continuous signal into a discrete digital representation, there is a range of input values that produces the same output. That range is called quantum (Q) and is equivalent to the Least Significant Bit (LSB). The difference between input and output is called the quantization error. Therefore, the quantization error can be between $\pm\frac{Q}{2}$.

2.3.3 About Signal to Quantization Noise ratio (SQNR) :

SQNR, short for signal to quantization noise ratio, is a measure of the quality of the quantization, or digital conversion of an analog signal. Defined as **normalized signal power** divided by **normalized quantization noise power**. The SQNR in dB is approximately equal to **6 times** the number of bits of the analog-to-digital converter (ADC). For example, the maximum SQNR for 16 bits is approximately 96dB. The Formula for calculating SQNR is given as :

$$SQNR = \frac{3}{2}(2^{2b}) = 1.76 + 6.02(b) \tag{2.5}$$

In the above equation , SQNR is in decibel (**dB**) and **b** represents the number of bits used to represent the Quantized signal samples. Relationship between number of levels(L) ans number of bits(b) is given as $L = 2^b$.

2.4 Code and results

2.4.1 Plot 5 cycles of sampled and quantized signal for various values of L=8,16,32,64 :

```
% 19ucc023
% Mohit Akhouri
% Experiment 2 - Observation 1

clc;
clear all;
close all;

% generating Ideal Signal for Fs = 100000 Hz

A = 1; % defining Amplitude
n_cycles = 5; % defining number of cycles
fs_ideal = 100000; % defining ideal frequency
f = 3000; % defining message signal frequency

n_ideal = 0:1:floor(n_cycles*(fs_ideal/f))-1; % defining range of n
x_ideal = A*cos(2*pi*f*n_ideal*(1/fs_ideal)); % generating Ideal
 signal

figure; % plotting Ideal signal
stem(n_ideal,x_ideal);
ylabel('Amplitude ->');
xlabel('Samples(n) ->');
title('19ucc023 - Mohit Akhouri','Generation of Ideal Signal for F_{s}
 = 100000 Hz');
grid on;

% generating Sampled signal for Fs = 8000 Hz
fs_sampled = 8000;
n_sampled = 0:1:floor(n_cycles*(fs_sampled/f))-1;
x_sampled = A*cos(2*pi*f*n_sampled*(1/fs_sampled));

figure;
stem(n_sampled,x_sampled,'Linewidth',1.5);
ylabel('Amplitude ->');
xlabel('Samples(n) ->');
title('19ucc023 - Mohit Akhouri','Sampled Signal for F_{s} = 8000
 Hz');
grid on;

y_quant_8 = myquantizer(x_sampled,8); % calculating quantized signal
 for L=8
y_quant_16 = myquantizer(x_sampled,16); % calculating quantized signal
 for L=16
y_quant_32 = myquantizer(x_sampled,32); % calculating quantized signal
 for L=32
y_quant_64 = myquantizer(x_sampled,64); % calculating quantized signal
 for L=64

% plotting Sampled signal and Quantized signal in 2 different plots
% for L = 8 and L = 16
figure;
```

Figure 2.3 Part 1 of the code for observation 1

```
subplot(2,2,1);
stem(n_sampled,x_sampled,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('Amplitude ->');
title('Sampled signal for F_{s} = 8000 Hz');
grid on;
subplot(2,2,2);
stem(n_sampled,y_quant_8,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('Amplitude ->');
title('Quantized signal for L = 8');
grid on;

subplot(2,2,3);
stem(n_sampled,x_sampled,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('Amplitude ->');
title('Sampled signal for F_{s} = 8000 Hz');
grid on;
subplot(2,2,4);
stem(n_sampled,y_quant_16,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('Amplitude ->');
title('Quantized signal for L = 16');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');

% plotting Sampled signal and Quantized signal in 2 different plots
% for L = 32 and L = 64
figure;
subplot(2,2,1);
stem(n_sampled,x_sampled,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('Amplitude ->');
title('Sampled signal for F_{s} = 8000 Hz');
grid on;
subplot(2,2,2);
stem(n_sampled,y_quant_32,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('Amplitude ->');
title('Quantized signal for L = 32');
grid on;

subplot(2,2,3);
stem(n_sampled,x_sampled,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('Amplitude ->');
title('Sampled signal for F_{s} = 8000 Hz');
grid on;
subplot(2,2,4);
stem(n_sampled,y_quant_64,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('Amplitude ->');
title('Quantized signal for L = 64');
```

Figure 2.4 Part 2 of the code for observation 1

```
grid on;
sgtitle('19ucc023 - Mohit Akhouri');

% plotting sampled signal and Quantized signal together in one plot
% for L = 8,16,32 and 64
figure;
stem(n_sampled,x_sampled,'Linewidth',1.2);
hold on;
stem(n_sampled,y_quant_8,'Linewidth',1.2);
xlabel('Samples(n) ->');
ylabel('Amplitude ->');
title('19ucc023 - Mohit Akhouri','Sampled Signal and Quantized signal
 for L = 8');
grid on;
legend('Sampled Signal','Quantized Signal');
hold off;

figure;
stem(n_sampled,x_sampled,'Linewidth',1.2);
hold on;
stem(n_sampled,y_quant_16,'Linewidth',1.2);
xlabel('Samples(n) ->');
ylabel('Amplitude ->');
title('19ucc023 - Mohit Akhouri','Sampled Signal and Quantized signal
 for L = 16');
grid on;
legend('Sampled Signal','Quantized Signal');
hold off;

figure;
stem(n_sampled,x_sampled,'Linewidth',1.2);
hold on;
stem(n_sampled,y_quant_32,'Linewidth',1.2);
xlabel('Samples(n) ->');
ylabel('Amplitude ->');
title('19ucc023 - Mohit Akhouri','Sampled Signal and Quantized signal
 for L = 32');
grid on;
legend('Sampled Signal','Quantized Signal');
hold off;

figure;
stem(n_sampled,x_sampled,'Linewidth',1.2);
hold on;
stem(n_sampled,y_quant_64,'Linewidth',1.2);
xlabel('Samples(n) ->');
ylabel('Amplitude ->');
title('19ucc023 - Mohit Akhouri','Sampled Signal and Quantized signal
 for L = 64');
grid on;
legend('Sampled Signal','Quantized Signal');
hold off;
```

Figure 2.5 Part 3 of the code for observation 1

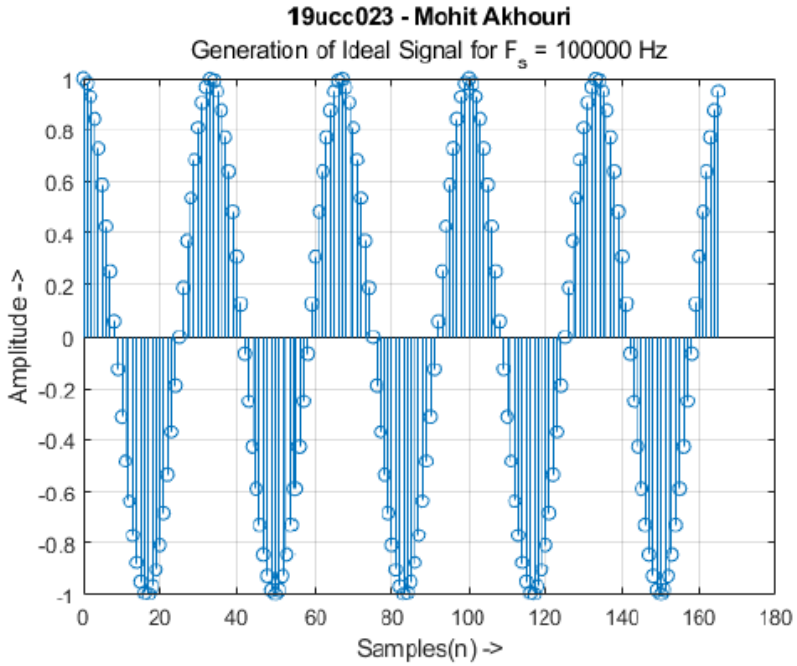


Figure 2.6 Ideal Signal For Sampling frequency = 100000 Hz

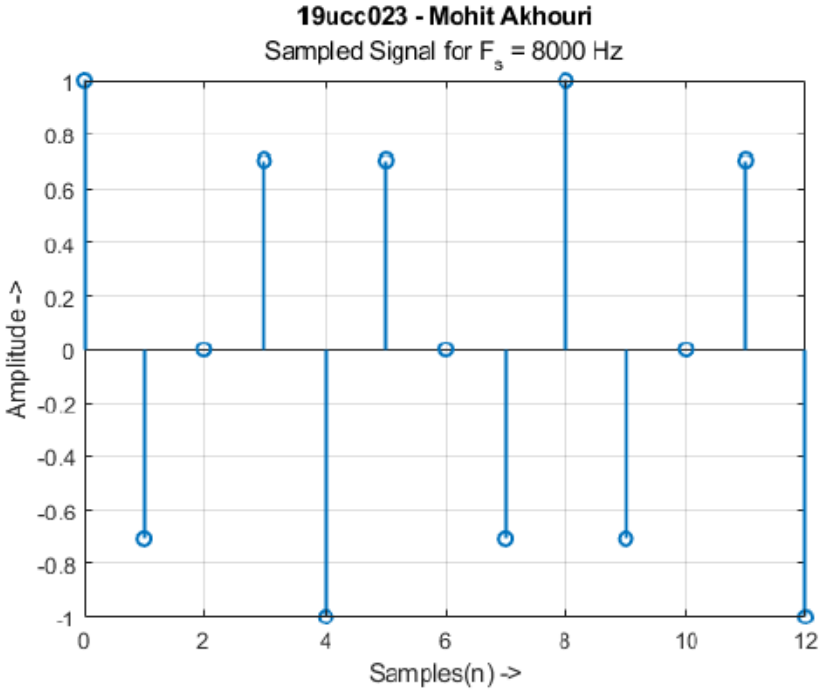


Figure 2.7 Sampled Signal For Sampling frequency = 8000 Hz

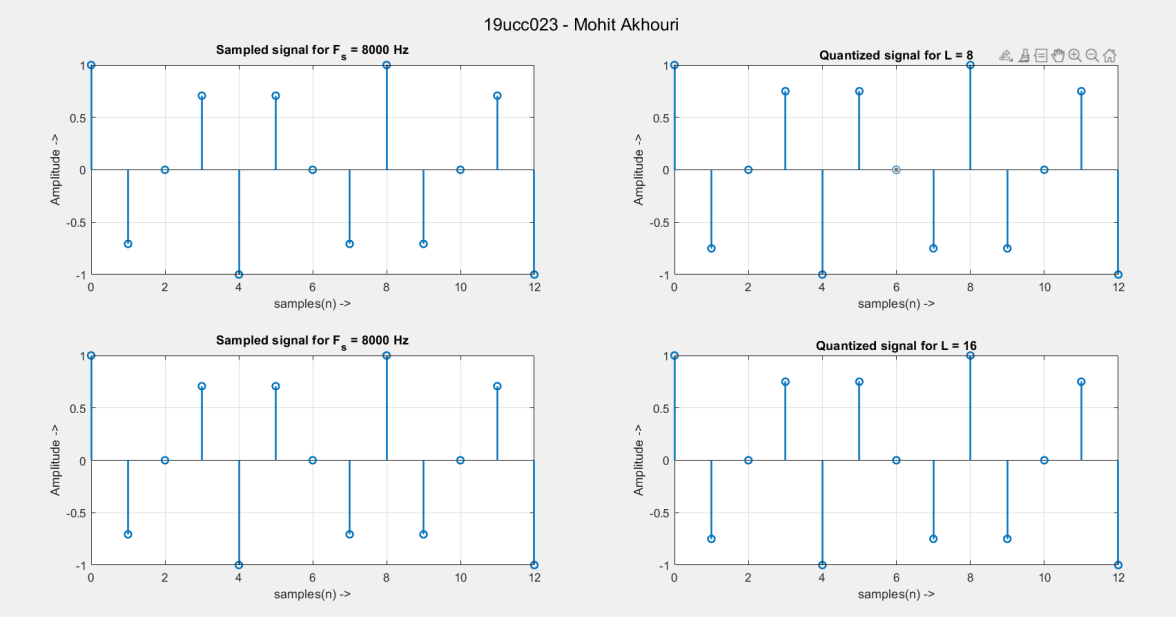


Figure 2.8 Sampled and Quantized signal (Different Plots) for number of levels(L) = 8 and 16

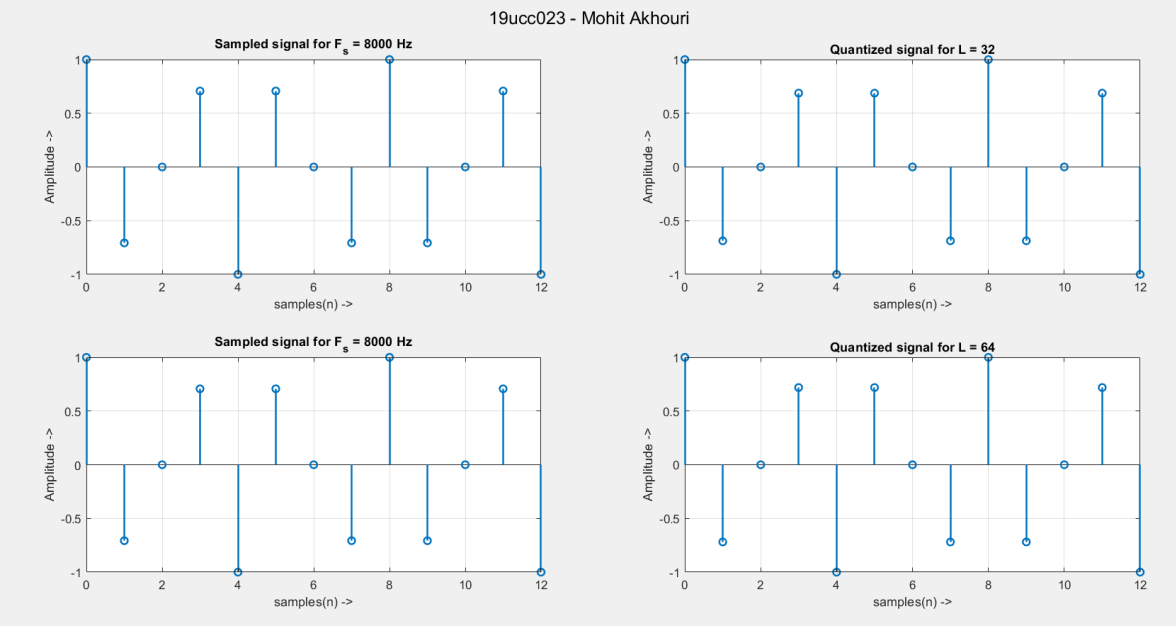


Figure 2.9 Sampled and Quantized signal (Different Plots) for number of levels(L) = 32 and 64

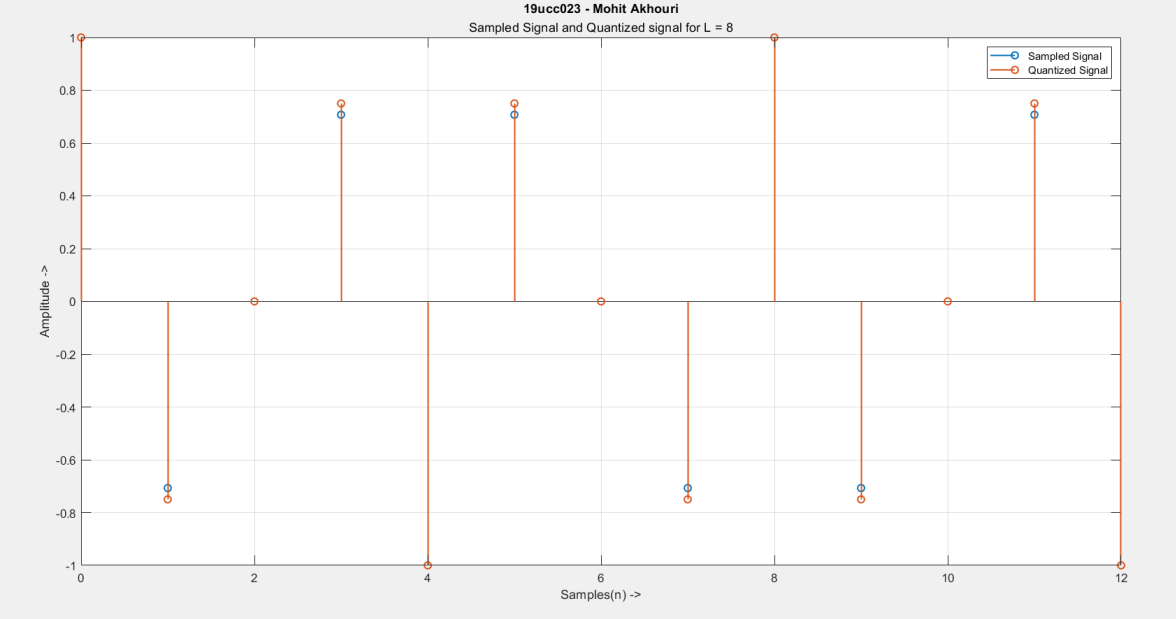


Figure 2.10 Sampled and Quantized signal (Same plot using **hold on**) for number of levels(L) = 8

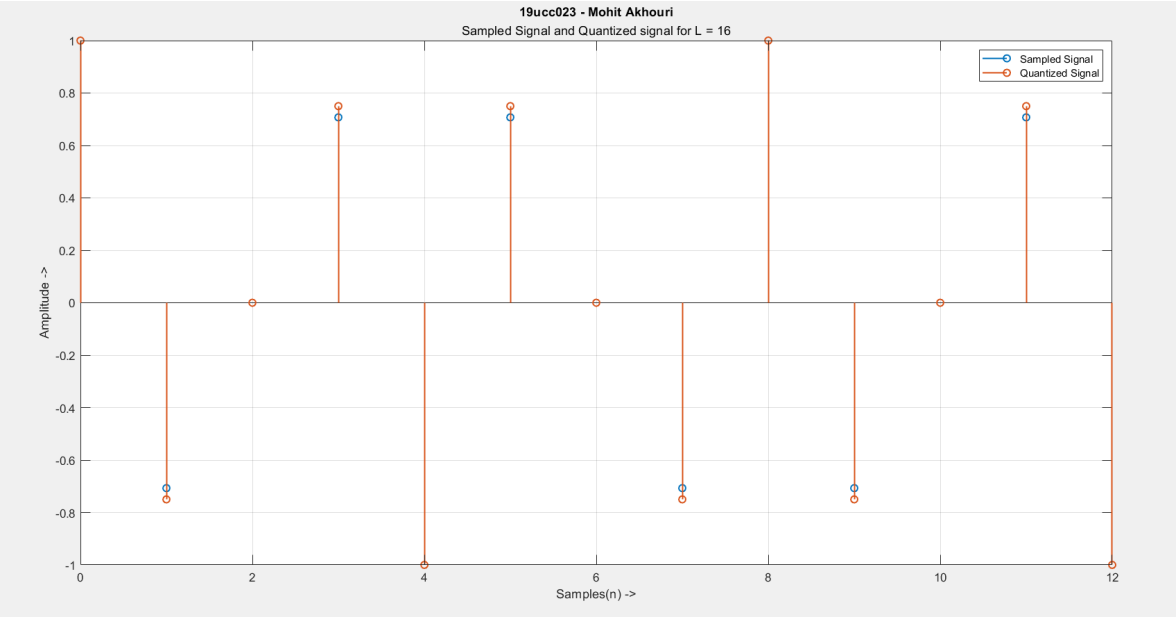


Figure 2.11 Sampled and Quantized signal (Same plot using **hold on**) for number of levels(L) = 16

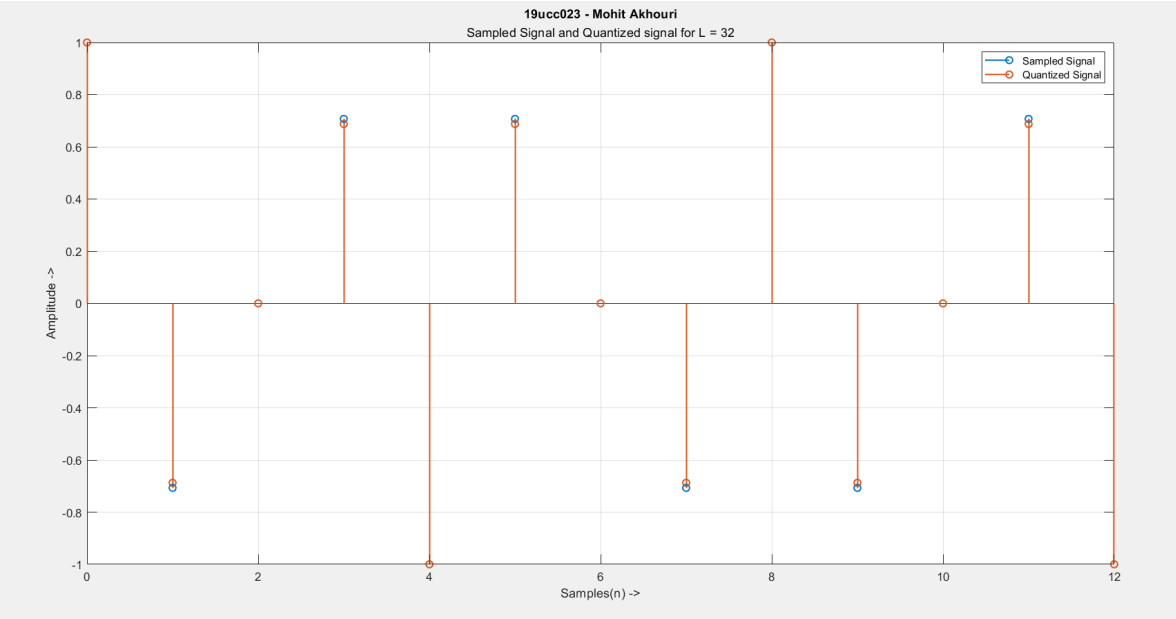


Figure 2.12 Sampled and Quantized signal (Same plot using **hold on**) for number of levels(L) = 32

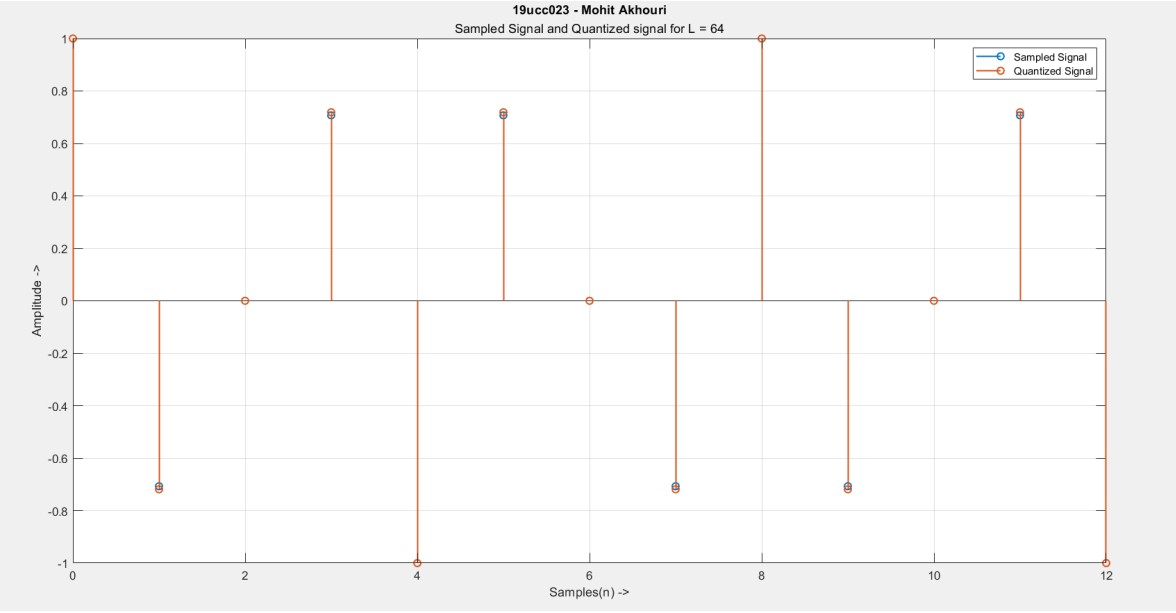


Figure 2.13 Sampled and Quantized signal (Same plot using **hold on**) for number of levels(L) = 64

2.4.2 Plot the graph b/w Quantization Noise Power (Q_e) and quantization levels L=16,32,64:

```
% 19ucc023
% Mohit Akhouri
% Experiment 2 - Observation 2

clc;
clear all;
close all;

A = 1; % defining Amplitude
n_cycles = 5; % defining number of cycles
f = 3000; % defining message signal frequency

fs_sampled = 8000; % defining sampling frequency
n_sampled = 0:1:floor(n_cycles*(fs_sampled/f))-1; % defining range of
 n
x_sampled = A*cos(2*pi*f*n_sampled*(1/fs_sampled)); % defining sampled
 signal x_sampled

L = [16 32 64]; % defining array for number of levels
Quantization_Noise_Power = zeros(1,3); % to store the SQNR practical
for i=1:length(L)
    y = myquantizer(x_sampled,L(i));
    Quantization_Noise_Power(i) = mean((y-x_sampled).*(y-
x_sampled)); % SQNR practical calculation
end

% plotting SQNR practical vs. number of levels
figure;
subplot(2,1,1);
stem(L,Quantization_Noise_Power,'Linewidth',1.5);
xlabel('Number of levels (L) ->');
ylabel('Q_{e} (MSE) Practical ->');
title('PRACTICAL Quantization Noise power ( Q_{e} ) vs. Number of
 Levels (L)');
grid on;

% calculating theoretical SQNR in dB
SQNR_db = zeros(1,3);
b16 = 4; % number of bits for L=16
b32 = 5; % number of bits for L=32
b64 = 6; % number of bits for L=64
SQNR_db(1) = 1.76 + 6.02*b16;
SQNR_db(2) = 1.76 + 6.02*b32;
SQNR_db(3) = 1.76 + 6.02*b64;

% plotting SQNR (theoretical) vs. Number of levels
subplot(2,1,2);
stem(L,SQNR_db,'Linewidth',1.5);
xlabel('Number of levels (L) ->');
ylabel('SQNR(dB) Theoretical ->');
title('THEORETICAL value of SQNR (dB) vs. Number of levels(L)');
grid on;
```

Figure 2.14 Code for observation 2

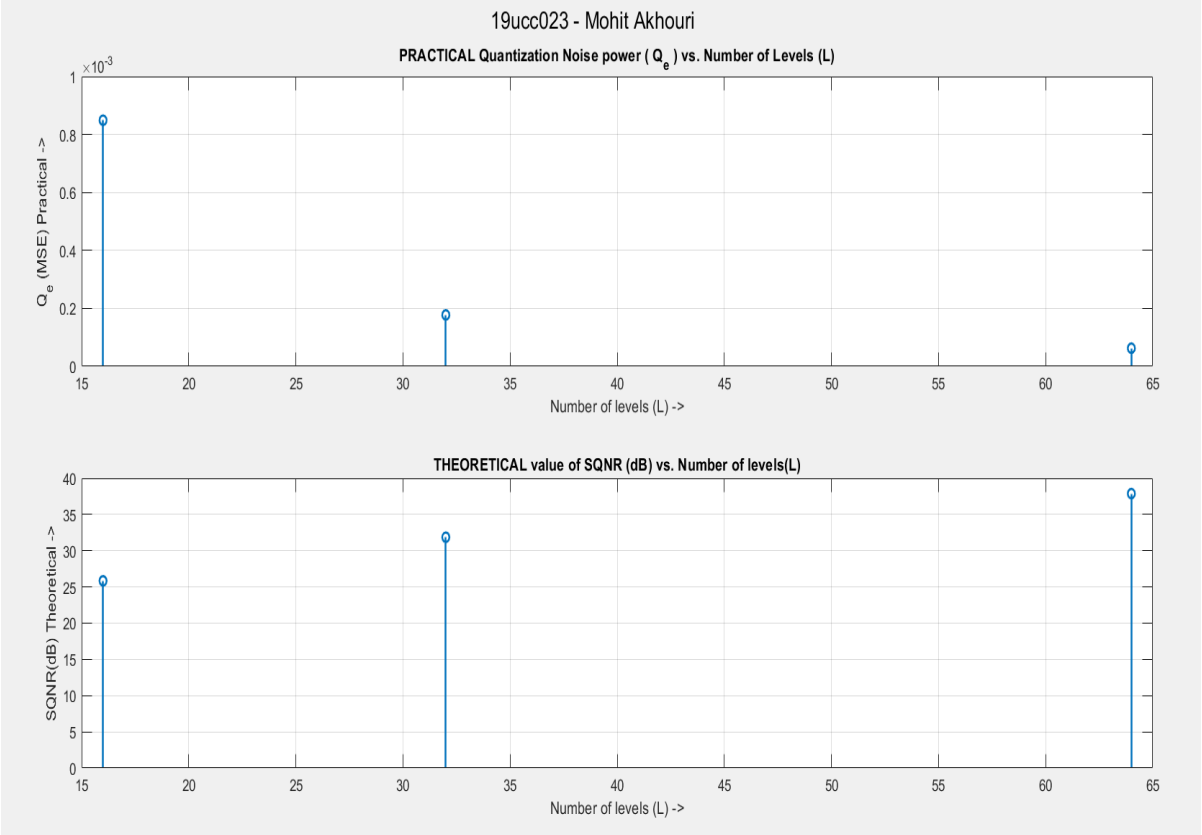


Figure 2.15 Plots of Practical and Theoretical (Q_e) vs. Number of levels

2.4.3 Plot SQNR vs. Input voltage for voltage levels in step size 0.1V :

```
% 19ucc023
% Mohit Akhouri
% Experiment 2 - Observation 3
clc;
clear all;
close all;

n_cycles = 5; % defining number of cycles
f = 3000; % defining message signal frequency
fs_sampled = 8000; % defining Sampling frequency
L = 64; % defining number of levels for the quantizer
b = 6; % defining the number of bits for calculation of SQNR

voltage_array = 0.1:0.1:1; % defining the voltage array at 0.1
 increments
len = length(voltage_array); % calculating length of voltage array

SQNR_practical = zeros(1,len); % initializing SQNR_practical array
SQNR_Theoretical = zeros(1,len); % initializing SQNR_theoretical array

for i=1:len
    A = voltage_array(i); % defining amplitude of x_sampled
    n_sampled = 0:1:floor(n_cycles*(fs_sampled/f))-1; % defining range
 of n
    x_sampled = A*cos(2*pi*f*n_sampled*(1/fs_sampled)); % creating
 sampled signal

    y = myquantizer(x_sampled,L); % quantized signal of sampled signal
 for L=64
    noise_mse = mean((y-x_sampled).*(y-x_sampled)); % MSE for the
 quantized and sampled signal
    signal_noise_ratio = (A*A/2)/(noise_mse); % SQNR practical

    % storing corresponding values in arrays
    SQNR_practical(i) = 10*log10(signal_noise_ratio);
    SQNR_Theoretical(i) = 1.76 + 6.02*b;
end
% plotting SQNR practical and SQNR Theoretical vs. Voltage
figure;
subplot(2,1,1);
stem(voltage_array,SQNR_practical,'Linewidth',1.2);
xlabel('Voltage(V) ->');
ylabel('SQNR(dB) practical ->');
title('SQNR(dB) PRACTICAL vs. Voltage(V)');
grid on;
subplot(2,1,2);
stem(voltage_array,SQNR_Theoretical,'Linewidth',1.2);
xlabel('Voltage(V) ->');
ylabel('SQNR(dB) theoretical ->');
title('SQNR(dB) THEORETICAL vs. Voltage(V)');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');
```

Figure 2.16 Code for observation 3

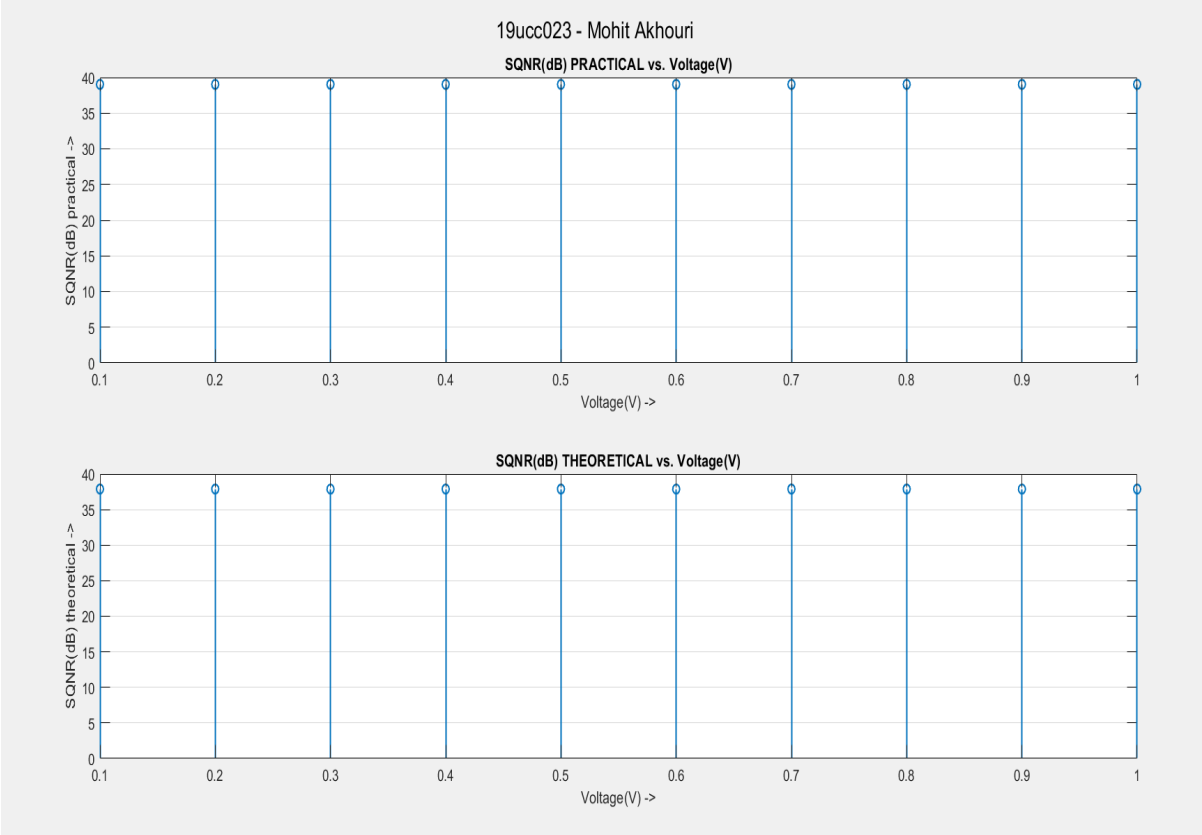


Figure 2.17 Plots of SQNR(dB) vs. Voltage levels in 0.1V step size

2.4.4 Plot the sampled, quantized and encoded signal for L=8 :

```
% 19ucc023
% Mohit Akhouri
% Experiment 2 - Observation 4

clc;
clear all;
close all;

n_cycles = 5; % defining number of cycles
f = 3000; % defining message signal frequency
fs_sampled = 8000; % defining Sampling frequency
A = 1; % defining Amplitude
L = 8; % defining number of levels for the quantizer

n_sampled = 0:1:floor(n_cycles*(fs_sampled/f))-1; % defining the range
 of "n"
x_sampled = A*cos(2*pi*f*n_sampled*(1/fs_sampled)); % defining the
 sampled signal

y = myquantizer(x_sampled,L); % calculating the quantized value of
 sampled signal

% plotting sampled and quantized signal separately
figure;
subplot(2,1,1);
stem(n_sampled,x_sampled,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('Amplitude ->');
title('Sampled signal for F_{s} = 8000 Hz');
grid on;
subplot(2,1,2);
stem(n_sampled,y,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('Amplitude ->');
title('Quantized signal for L = 8');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');

% plotting sampled and quantized signal together
figure;
stem(n_sampled,x_sampled,'Linewidth',1.2);
hold on;
stem(n_sampled,y,'Linewidth',1.2);
xlabel('Samples(n) ->');
ylabel('Amplitude ->');
title('19ucc023 - Mohit Akhouri','Sampled Signal and Quantized signal
 for L = 8');
grid on;
legend('Sampled Signal','Quantized Signal');
hold off;

% doing encoding of quantized signal
```

Figure 2.18 Part 1 of the Code for observation 4

```
y_encoded = myencoder(y,L); % calling myencoder function for encoding
 of quantized signal
display('The encoded signal is :');
for i=1:length(y)
    display(sprintf('%-10f = %s',y(i),y_encoded(i))); % displaying the
 encoded values
end
```

Figure 2.19 Part 2 of the Code for observation 4

```
The encoded signal is :
1.000000   = 111
-0.750000  = 000
0.000000   = 011
0.750000   = 110
-1.000000  = 000
0.750000   = 110
0.000000   = 011
-0.750000  = 000
1.000000   = 111
-0.750000  = 000
0.000000   = 011
0.750000   = 110
-1.000000  = 000
```

Figure 2.20 Displaying **encoded values** for different quantized values

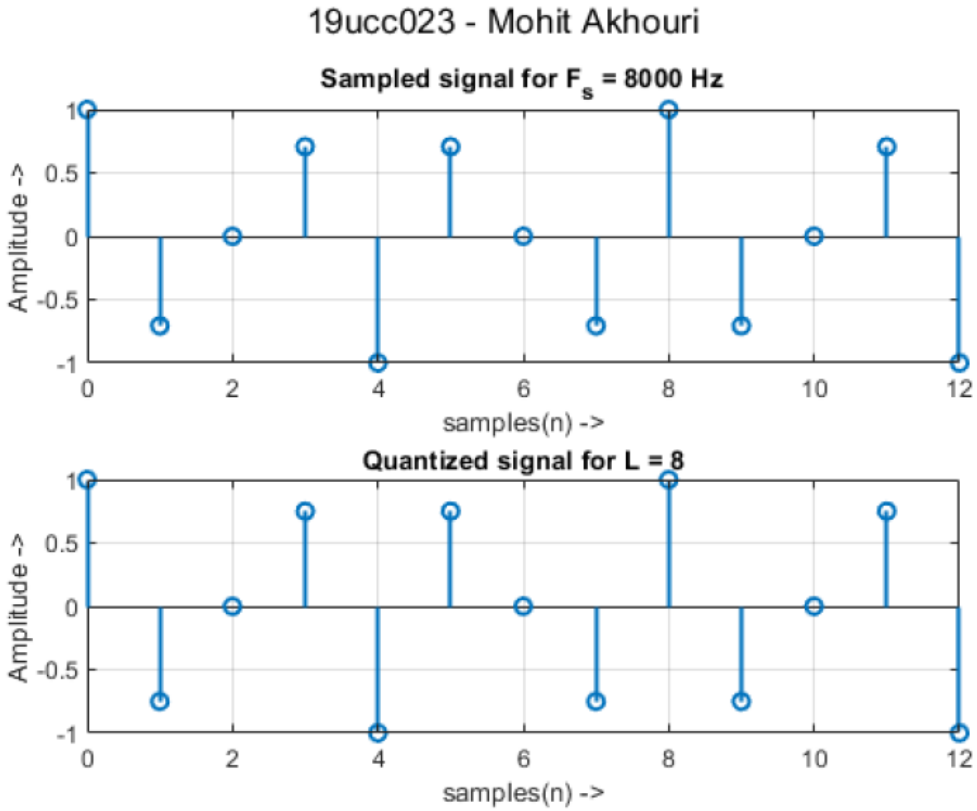


Figure 2.21 Sampled and Quantized signal (Different Plots) for number of levels(L) = 8

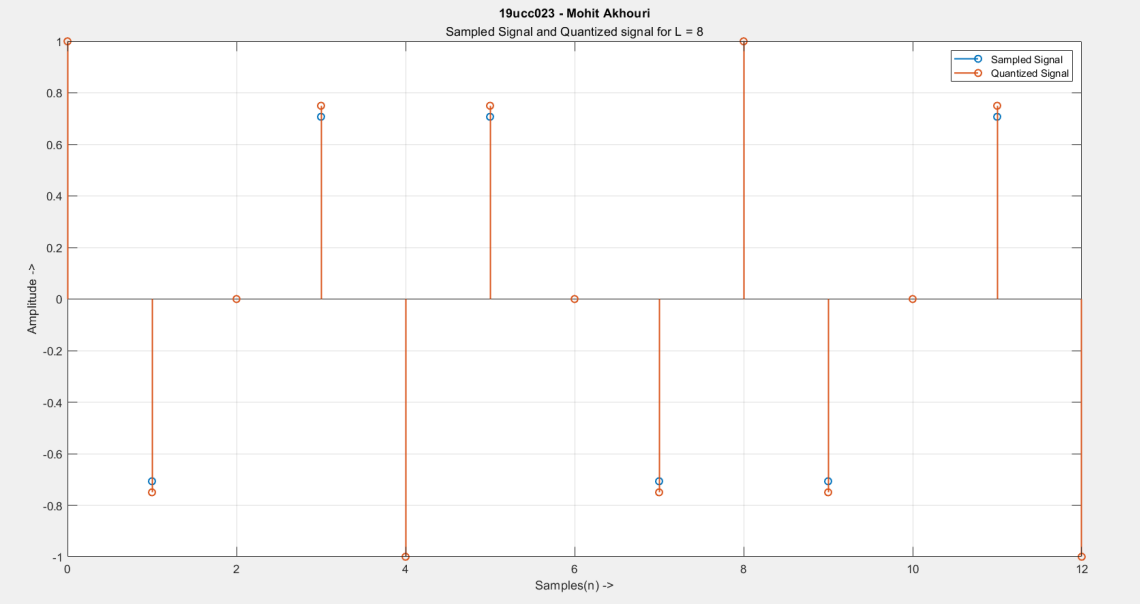


Figure 2.22 Sampled and Quantized signal (Same plots using **hold on**) for number of levels(L) = 8

2.4.5 Quantization and encoding in Simulink :

```
% 19ucc023
% Mohit Akhouri
% Experiment 2 - Observation 5

% doing quantization and encoding for L=8 via Simulink model

sim('Simulink_Observation_5'); % calling the Simulink model

n_cycles = 5; % defining number of cycles
f = 3000; % defining message signal frequency
fs_sampled = 8000; % defining Sampling frequency
A = 1; % defining Amplitude
L = 8; % defining number of levels for the quantizer

n_sampled = 0:1:floor(n_cycles*(fs_sampled/f))-1; % defining the range
 of "n"
x_sampled = A*cos(2*pi*f*n_sampled*(1/fs_sampled)); % defining the
 sampled signal

% plotting sampled and quantized signal separately
figure;
subplot(2,1,1);
stem(n_sampled,x_sampled,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('Amplitude ->');
title('Sampled signal for F_{s} = 8000 Hz');
grid on;
subplot(2,1,2);
stem(n_sampled,out.y_sampled.data,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('Amplitude ->');
title('Quantized signal for L = 8');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');

% plotting sampled and quantized signal together
figure;
stem(n_sampled,x_sampled,'Linewidth',1.2);
hold on;
stem(n_sampled,out.y_sampled.data,'Linewidth',1.2);
xlabel('Samples(n) ->');
ylabel('Amplitude ->');
title('19ucc023 - Mohit Akhouri','Sampled Signal and Quantized signal
 for L = 8');
grid on;
legend('Sampled Signal','Quantized Signal');
hold off;

% doing encoding of quantized signal via Simulink
y = out.y_sampled.data;
y_encoded = out.y_encoded.data;
% doing encoding of quantized signal
```

Figure 2.23 Part 1 of the Code for observation 5

```
display('The encoded signal is :');
for i=1:length(y)
    display(sprintf('%-10f = %s',y(i),dec2bin(y_encoded(i),3))); %
 displaying the encoded values
end
```

Figure 2.24 Part 2 of the Code for observation 5

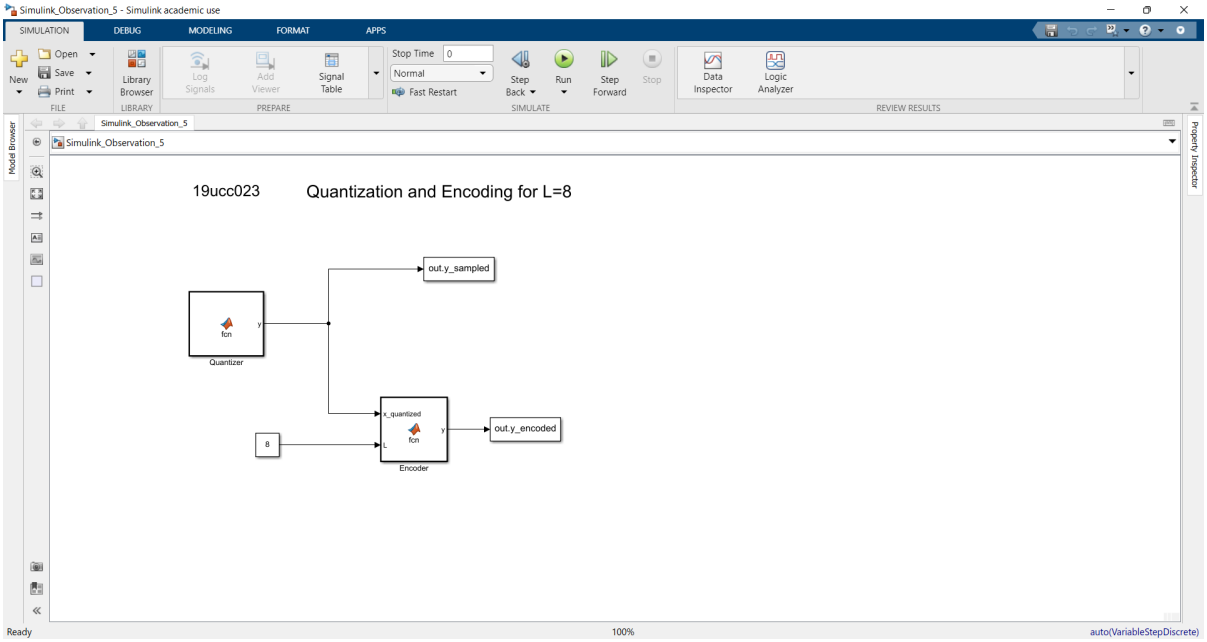


Figure 2.25 Simulink model for Quantization and Encoding

```
The encoded signal is :
1.000000    = 111
-0.750000   = 001
0.000000    = 011
0.750000    = 111
-1.000000   = 000
0.750000    = 111
0.000000    = 011
-0.750000   = 001
1.000000    = 111
-0.750000   = 001
0.000000    = 011
0.750000    = 111
-1.000000   = 000
```

Figure 2.26 Encoded values for the quantized signal for L=8

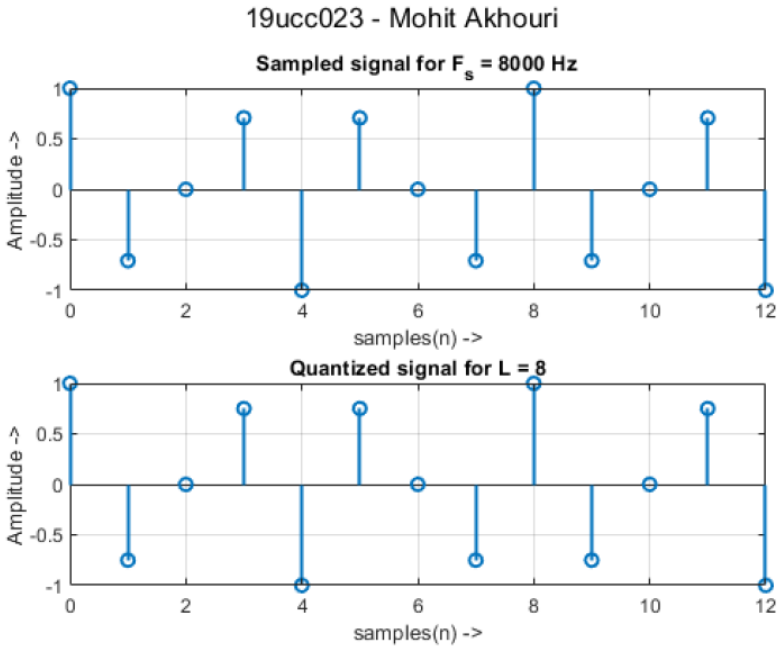


Figure 2.27 Sampled and Quantized signal (Different Plots) for L = 8 using Simulink model

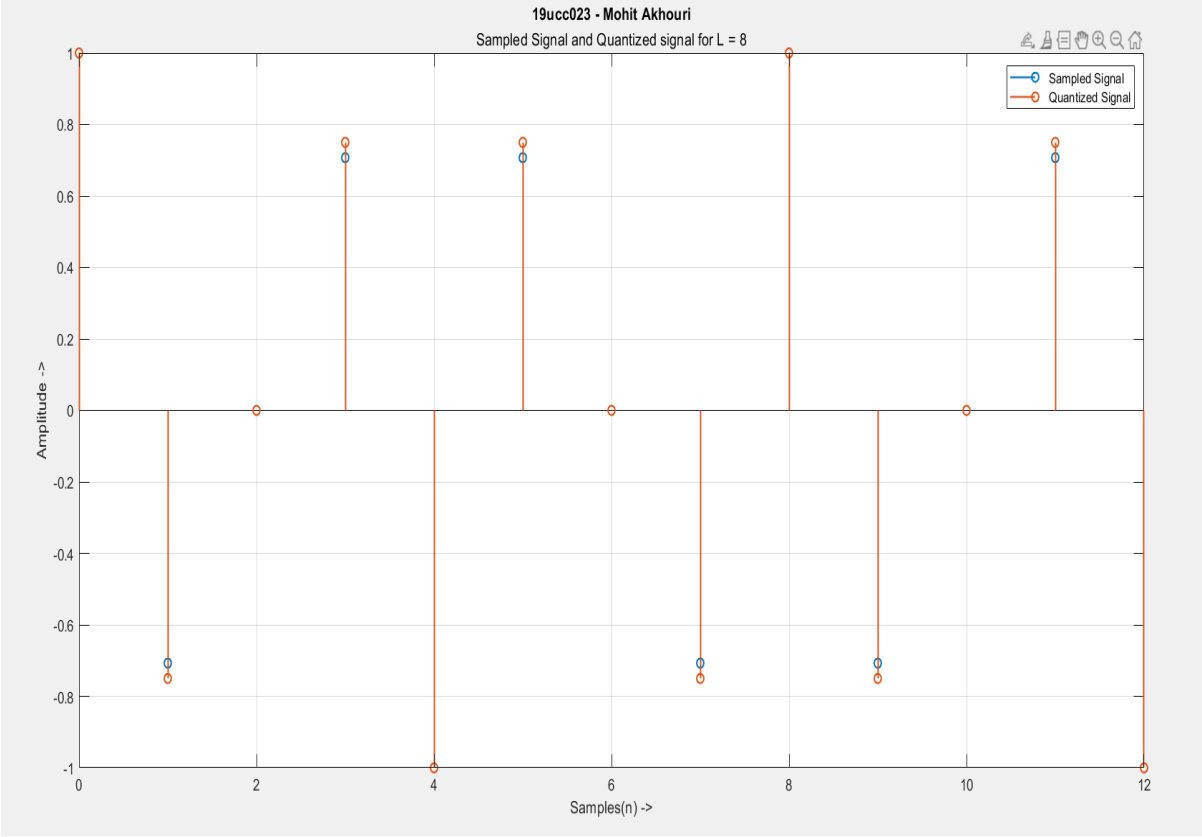


Figure 2.28 Sampled and Quantized signal (Same Plots) for L = 8 using Simulink model

2.4.6 Functions used in main codes for Quantization and encoding :

2.4.6.1 myquantizer.m function code :

```
function [y] = myquantizer(x_sampled,L)
% This function does quantization of sampled signal x_sampled for L
 number
% of levels

% 19ucc023
% Mohit Akhouri

max_value = max(x_sampled); % maximum value of sampled signal
min_value = min(x_sampled); % minimum value of sampled signal
delta = (max_value - min_value)/L; % Step size ( delta ) calculation

len = length(x_sampled); % defining length of sampled signal

y = zeros(1,len);
for ind = 1:len
    i = round((x_sampled(ind)-min_value)/delta); % calculating value
 of i
    y(ind) = min_value + i*delta; % calculating and storing quantized
 value of ith sample of x_sampled
end

end
```

Published with MATLAB® R2020b

Figure 2.29 myquantizer function for Quantization computation

2.4.6.2 myencoder.m function code :

```
function [y_encoded] = myencoder(x_quantized,L)
% This function encodes the Quantized value by binary numbers

% 19ucc023
% Mohit Akhouri

max_value = max(x_quantized); % maximum value of sampled signal
min_value = min(x_quantized); % minimum value of sampled signal
delta = (max_value - min_value)/(L-1); % Step size ( delta )
 calculation

len = length(x_quantized); % calculating length of x_quantized
y_encoded = strings(1,len); % initializing the y_encoded array

for ind=1:len
    i = (x_quantized(ind)-min_value)/delta; % calculating the decimal
 value for encoding
    y_encoded(ind) = dec2bin(i,3); % converting decimal number i to
 binary number
end

end
```

Published with MATLAB® R2020b

Figure 2.30 myencoder function for Encoding of Quantized signal

2.5 Conclusion

In this experiment , we learnt the concepts of **Quantization** and **encoding** of Digital Signal Processing. We learnt how do we do quantization by dividing the voltage range into number of levels and then rounding each of sample values. We also learnt the process of encoding different quantized values to different binary numbers. We also learnt the concepts of **Quantization Noise error** and **Signal to Noise ratio (SQNR)** and how to compute them both practically and theoretically. We concluded that **Quantization noise** is **lower** for **more number of levels**. We also concluded the effect of increasing voltage levels on SQNR. We performed the same experiment in Simulink and cross-checked the results obtained with the MATLAB code.

Chapter *3*

Experiment - 3

3.1 Aim of the Experiment

- Linear Convolution and DFT matrix Generation
- Simulink based convolution

3.2 Software Used

- MATLAB
- Simulink

3.3 Theory

3.3.1 About Linear Time invariant (LTI) system :

In system analysis, among other fields of study, a **linear time-invariant system** (LTI system) is a system that produces an output signal from any input signal subject to the constraints of **linearity** and **time-invariance**. These properties apply (exactly or approximately) to many important physical systems, in which case the response **y(t)** of the system to an arbitrary input **x(t)** can be found directly using convolution: **y(t) = x(t) h(t)** where **h(t)** is called the system's impulse response and represents convolution. A good example of an LTI system is any **electrical circuit** consisting of resistors, capacitors, inductors and linear amplifiers. Linear time-invariant system theory is also used in **image processing**.

LTI systems can also be characterized in the frequency domain by the system's transfer function, which is the **Laplace transform** of the system's impulse response (or Z transform in the case of discrete-time systems). As a result of the properties of these transforms, the output of the system in the frequency domain is the product of the transfer function and the transform of the input. In other words, **convolution** in the **time domain** is equivalent to **multiplication** in the frequency domain.

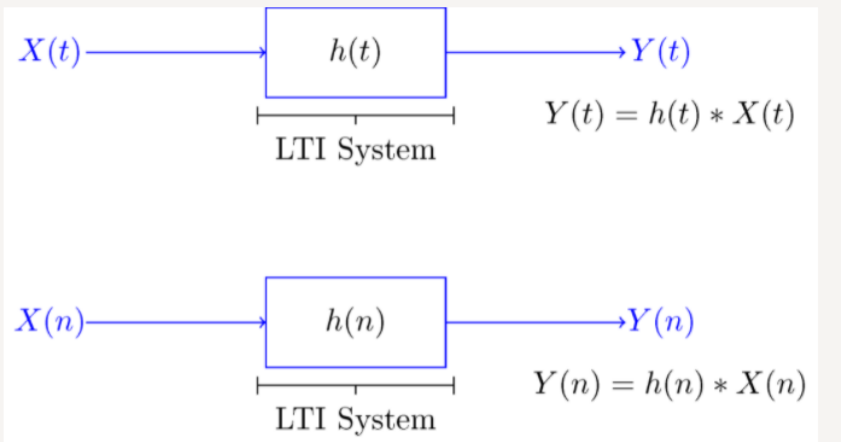


Figure 3.1 LTI system characterization

3.3.2 About Linear Convolution :

Linear convolution is a mathematical operation done to calculate the output of any Linear-Time Invariant (LTI) system given its **input** and **impulse response**. In linear convolution, both the sequences (input and impulse response) may or may not be of equal sizes. It is possible to find the response of a filter using linear convolution. The equation for Linear Convolution of **Discrete Time Sequence** is given as :

$$y[n] = x[n] * h[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k] \tag{3.1}$$

In the above equation , **x[n]** is the **input sequence** and **h[n]** is the **Impulse response**.

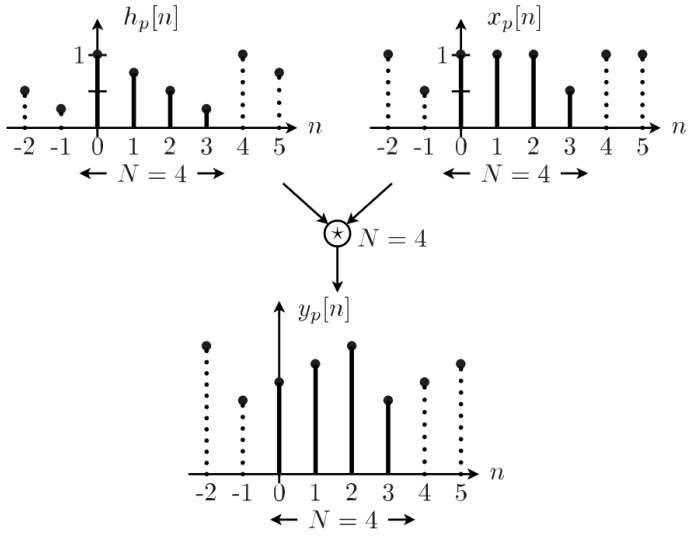


Figure 3.2 Linear Convolution of two finite length sequences

3.3.3 About Discrete Fourier Transform (DFT) :

The **discrete Fourier transform** (DFT) is one of the most important tools in digital signal processing. The DFT can calculate a signal's **frequency spectrum**. This is a direct examination of information encoded in the frequency, phase, and amplitude of the component sinusoids. For example, **human speech** and hearing use signals with this type of encoding. Second, the DFT can find a system's frequency response from the system's impulse response, and vice versa.

The DFT of a discrete-time signal x[n] with total **N samples** is given as :

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{\frac{-j2\pi nk}{N}} \tag{3.2}$$

3.3.3.1 About DFT matrix :

A **DFT matrix** is an expression of a discrete Fourier transform (DFT) as a **transformation matrix**, which can be applied to a signal through matrix multiplication.

DFT matrix can be constructed from **Twiddle factor** which is given as :

$$W_N = e^{\frac{-j2\pi}{N}} \tag{3.3}$$

The DFT can be calculated from DFT matrix and the equation for the same is given as:

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn} \tag{3.4}$$

In the above equation , k ranges from 0 to N-1.

$$W_N^{kn} \equiv \begin{array}{c} \\ k=0 \\ k=1 \\ k=2 \\ \\ \\ \\ k=N-1 \end{array} \begin{array}{c} \begin{array}{cccccc} n=0 & n=1 & n=2 & \cdots\cdots & & n=N-1 \end{array} \\ \begin{bmatrix} W_N^0 & W_N^0 & W_N^0 & \cdots\cdots & & W_N^0 \\ W_N^0 & W_N^1 & W_N^2 & \cdots\cdots & & W_N^{N-1} \\ W_N^0 & W_N^2 & W_N^4 & \cdots\cdots & & W_N^{2(N-1)} \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ W_N^0 & W_N^{N-1} & W_N^{2(N-1)} & \cdots\cdots & & W_N^{(N-1)^2} \end{bmatrix} \end{array}$$

Figure 3.3 Twiddle Factor Matrix

3.4 Code and results

3.4.1 Linear Convolution using Convolution matrix M :

```
% 19ucc023
% Mohit Akhouri
% Experiment 3 - Observation 1 and Observation 2

% This code will calculate the convolution of x[n] and h[n]
% for different frequencies and impulse responses
% both by calling the user defined functions myLinConvMat and myConv
% and using inbuilt function 'conv'

% ALGORITHM : First we calculate the convolution matrix M and
% calculate the convolution using M and input sequence x[n]

clc;
clear all;
close all;

n = 0:1:99; % initializing the number of samples
h1 = [1 1]; % initializing impulse response h1[n]
h2 = [1 -1]; % initializing impulse response h2[n]
h3 = (1/3)*[1 1 1]; % initializing impulse response h3[n]
h4 = (1/4)*[1 1 -4 1 1]; % initializing impulse response h4[n]

freq = [0 1/10 1/5 1/4 1/2]; % initializing the frequency array

% Convolution for x and h1

h = h1; % temporary variable to store the different impulse responses
ind = 1; % index for correct printing of 'number of impulse response'
for i = 1:length(freq)
    xi = cos(2*pi*freq(i)*n); % initializing x[n] for different
 frequencies

    M = myLinConvMat(h,length(xi)); % storing convolution matrix in M
    y_conv = myConv(xi,M); % convolution of x[n] and h[n] using myConv
 function
    y_conv_inbuilt = conv(xi,h); % convolution of x[n] and h[n] using
 INBUILT function conv

    % plotting x[n], h[n]
    figure;
    subplot(2,2,1);
    stem(xi,'Linewidth',1.3);
    xlabel('samples(n)->');
    ylabel(sprintf('x_{%d}[n]->',i));
    title(sprintf('x_{%d}[n] = cos(2*%s*(%0.2f)*n)',i,'\pi',freq(i)));
    grid on;
    subplot(2,2,2);
    stem(h,'Linewidth',1.3);
    xlabel('samples(n)->');
    ylabel(sprintf('h_{%d}[n]->',ind));
    title(sprintf('Impulse response h_{%d}[n]',ind));
    grid on;
```

Figure 3.4 Part 1 of the code for observation 1 and 2

```
    % plotting convolution using myConv and INBUILT FUNCTION conv
    subplot(2,2,3);
    stem(y_conv,'Linewidth',1.3);
    xlabel('samples(n)->');
    ylabel(sprintf('x_{%d}[n]*h_{%d}[n]->',i,ind));
    title(sprintf('convolution of x_{%d}[n] and h_{%d}[n] for
 frequency = %0.2f Hz using myConv',i,ind,freq(i)));
    grid on;
    subplot(2,2,4);
    stem(y_conv_inbuilt,'Linewidth',1.3);
    xlabel('samples(n)->');
    ylabel(sprintf('x_{%d}[n]*h_{%d}[n]->',i,ind));
    title(sprintf('convolution of x_{%d}[n] and h_{%d}[n] for
 frequency = %0.2f Hz using INBUILT function conv',i,ind,freq(i)));
    grid on;
    sgtitle('19ucc023 - Mohit Akhouri');
end

% Convolution for x and h2

h = h2; % temporary variable to store the different impulse responses
ind = 2; % index for correct printing of 'number of impulse response'
for i = 1:length(freq)
    xi = cos(2*pi*freq(i)*n); % initializing x[n] for different
 frequencies

    M = myLinConvMat(h,length(xi)); % storing convolution matrix in M
    y_conv = myConv(xi,M); % convolution of x[n] and h[n] using myConv
 function
    y_conv_inbuilt = conv(xi,h); % convolution of x[n] and h[n] using
 INBUILT function conv

    % plotting x[n], h[n]
    figure;
    subplot(2,2,1);
    stem(xi,'Linewidth',1.3);
    xlabel('samples(n)->');
    ylabel(sprintf('x_{%d}[n]->',i));
    title(sprintf('x_{%d}[n] = cos(2*%s*(%0.2f)*n)',i,'\pi',freq(i)));
    grid on;
    subplot(2,2,2);
    stem(h,'Linewidth',1.3);
    xlabel('samples(n)->');
    ylabel(sprintf('h_{%d}[n]->',ind));
    title(sprintf('Impulse response h_{%d}[n]',ind));
    grid on;
    % plotting convolution using myConv and INBUILT FUNCTION conv
    subplot(2,2,3);
    stem(y_conv,'Linewidth',1.3);
    xlabel('samples(n)->');
    ylabel(sprintf('x_{%d}[n]*h_{%d}[n]->',i,ind));
    title(sprintf('convolution of x_{%d}[n] and h_{%d}[n] for
 frequency = %0.2f Hz using myConv',i,ind,freq(i)));
    grid on;
```

Figure 3.5 Part 2 of the code for observation 1 and 2

```
    subplot(2,2,4);
    stem(y_conv_inbuilt,'Linewidth',1.3);
    xlabel('samples(n)->');
    ylabel(sprintf('x_{%d}[n]*h_{%d}[n]->',i,ind));
    title(sprintf('convolution of x_{%d}[n] and h_{%d}[n] for
 frequency = %0.2f Hz using INBUILT function conv',i,ind,freq(i)));
    grid on;
    sgtitle('19ucc023 - Mohit Akhouri');
end

% Convolution for x and h3

h = h3; % temporary variable to store the different impulse responses
ind = 3; % index for correct printing of 'number of impulse response'
for i = 1:length(freq)
    xi = cos(2*pi*freq(i)*n); % initializing x[n] for different
 frequencies

    M = myLinConvMat(h,length(xi)); % storing convolution matrix in M
    y_conv = myConv(xi,M); % convolution of x[n] and h[n] using myConv
 function
    y_conv_inbuilt = conv(xi,h); % convolution of x[n] and h[n] using
 INBUILT function conv

    % plotting x[n], h[n]
    figure;
    subplot(2,2,1);
    stem(xi,'Linewidth',1.3);
    xlabel('samples(n)->');
    ylabel(sprintf('x_{%d}[n]->',i));
    title(sprintf('x_{%d}[n] = cos(2*%s*(%0.2f)*n)',i,'\pi',freq(i)));
    grid on;
    subplot(2,2,2);
    stem(h,'Linewidth',1.3);
    xlabel('samples(n)->');
    ylabel(sprintf('h_{%d}[n]->',ind));
    title(sprintf('Impulse response h_{%d}[n]',ind));
    grid on;
    % plotting convolution using myConv and INBUILT FUNCTION conv
    subplot(2,2,3);
    stem(y_conv,'Linewidth',1.3);
    xlabel('samples(n)->');
    ylabel(sprintf('x_{%d}[n]*h_{%d}[n]->',i,ind));
    title(sprintf('convolution of x_{%d}[n] and h_{%d}[n] for
 frequency = %0.2f Hz using myConv',i,ind,freq(i)));
    grid on;
    subplot(2,2,4);
    stem(y_conv_inbuilt,'Linewidth',1.3);
    xlabel('samples(n)->');
    ylabel(sprintf('x_{%d}[n]*h_{%d}[n]->',i,ind));
    title(sprintf('convolution of x_{%d}[n] and h_{%d}[n] for
 frequency = %0.2f Hz using INBUILT function conv',i,ind,freq(i)));
    grid on;
    sgtitle('19ucc023 - Mohit Akhouri');
```

Figure 3.6 Part 3 of the code for observation 1 and 2

```
end

% Convolution for x and h4

h = h4; % temporary variable to store the different impulse responses
ind = 4; % index for correct printing of 'number of impulse response'
for i = 1:length(freq)
    xi = cos(2*pi*freq(i)*n); % initializing x[n] for different
 frequencies

    M = myLinConvMat(h,length(xi)); % storing convolution matrix in M
    y_conv = myConv(xi,M); % convolution of x[n] and h[n] using myConv
 function
    y_conv_inbuilt = conv(xi,h); % convolution of x[n] and h[n] using
 INBUILT function conv

    % plotting x[n], h[n]
    figure;
    subplot(2,2,1);
    stem(xi,'Linewidth',1.3);
    xlabel('samples(n)->');
    ylabel(sprintf('x_{%d}[n]->',i));
    title(sprintf('x_{%d}[n] = cos(2*%s*(%0.2f)*n)',i,'\pi',freq(i)));
    grid on;
    subplot(2,2,2);
    stem(h,'Linewidth',1.3);
    xlabel('samples(n)->');
    ylabel(sprintf('h_{%d}[n]->',ind));
    title(sprintf('Impulse response h_{%d}[n]',ind));
    grid on;
    % plotting convolution using myConv and INBUILT FUNCTION conv
    subplot(2,2,3);
    stem(y_conv,'Linewidth',1.3);
    xlabel('samples(n)->');
    ylabel(sprintf('x_{%d}[n]*h_{%d}[n]->',i,ind));
    title(sprintf('convolution of x_{%d}[n] and h_{%d}[n] for
 frequency = %0.2f Hz using myConv',i,ind,freq(i)));
    grid on;
    subplot(2,2,4);
    stem(y_conv_inbuilt,'Linewidth',1.3);
    xlabel('samples(n)->');
    ylabel(sprintf('x_{%d}[n]*h_{%d}[n]->',i,ind));
    title(sprintf('convolution of x_{%d}[n] and h_{%d}[n] for
 frequency = %0.2f Hz using INBUILT function conv',i,ind,freq(i)));
    grid on;
    sgtitle('19ucc023 - Mohit Akhouri');
end
```

Published with MATLAB® R2020b

Figure 3.7 Part 4 of the code for observation 1 and 2

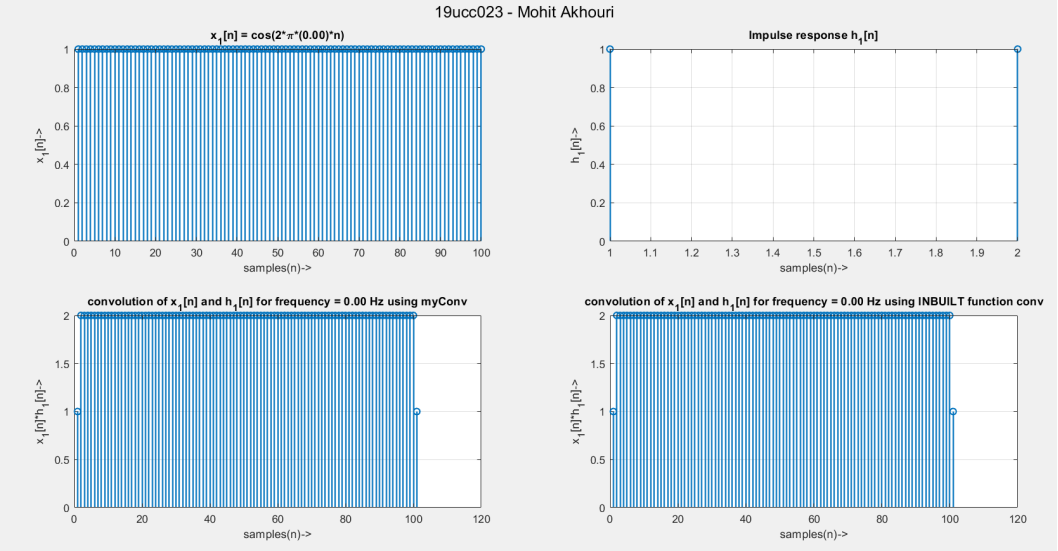


Figure 3.8 Plot of the convolution between $x_1[n]$ and $h_1[n]$

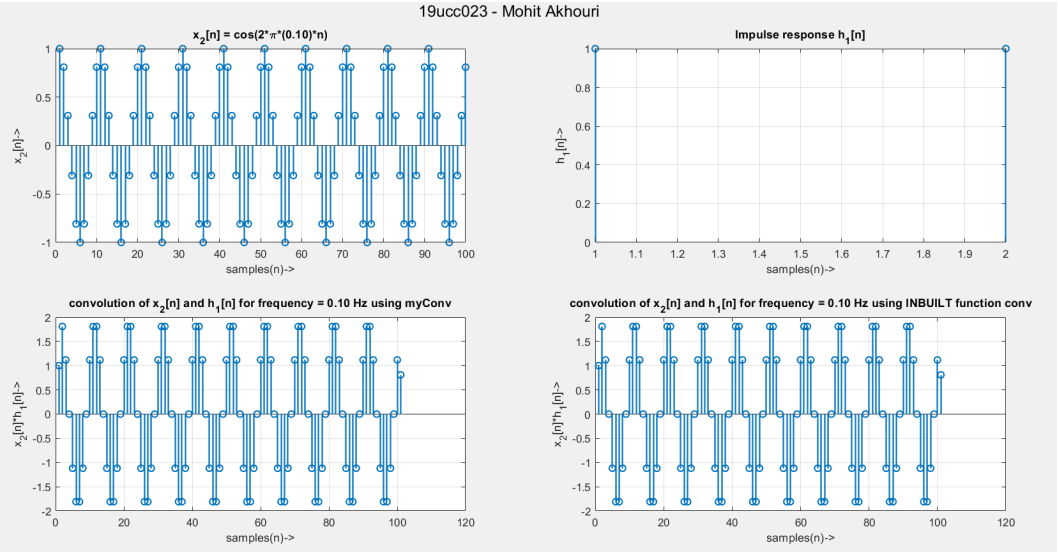


Figure 3.9 Plot of the convolution between $x_2[n]$ and $h_1[n]$

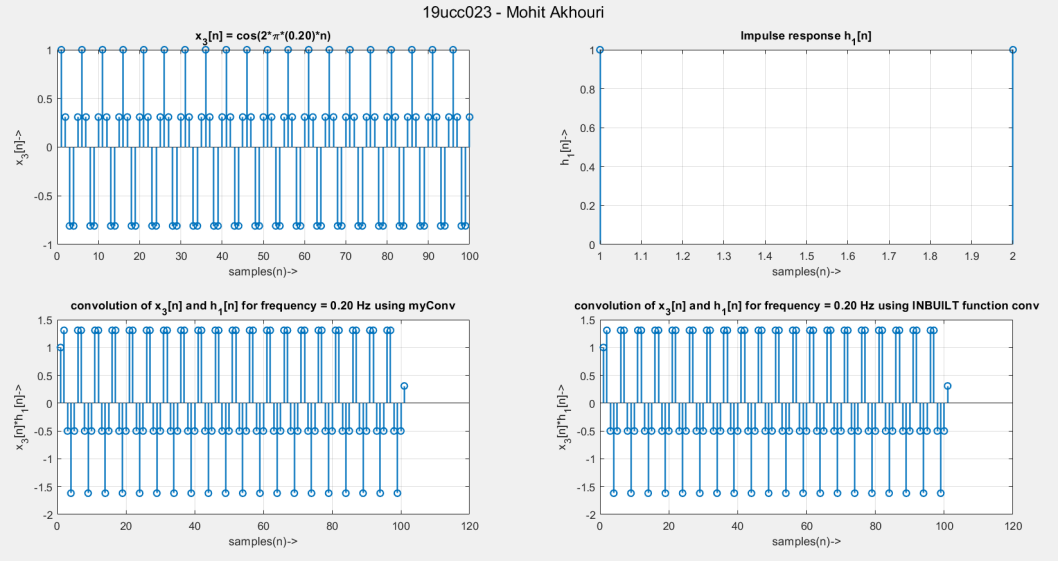


Figure 3.10 Plot of the convolution between $x_3[n]$ and $h_1[n]$

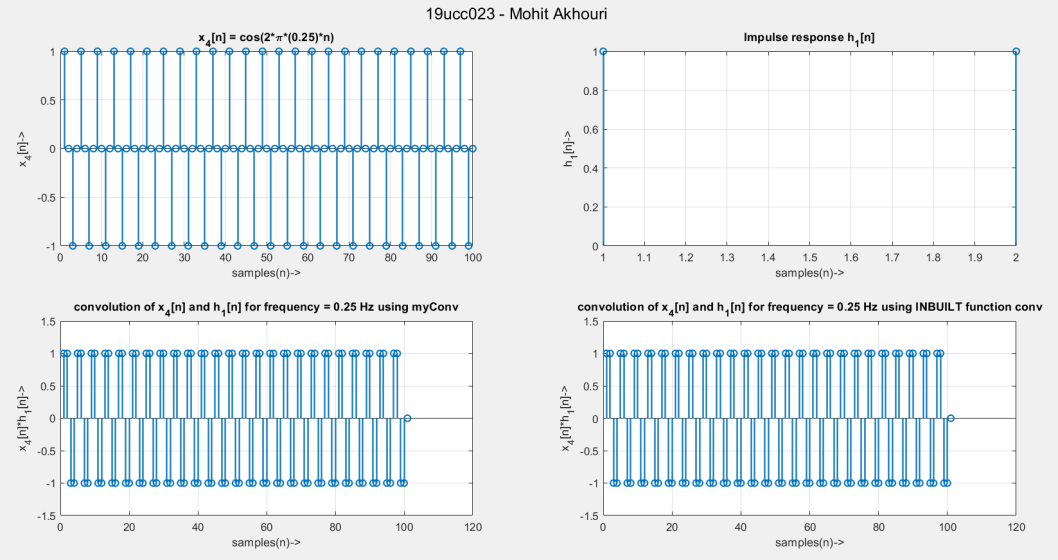


Figure 3.11 Plot of the convolution between $x_4[n]$ and $h_1[n]$

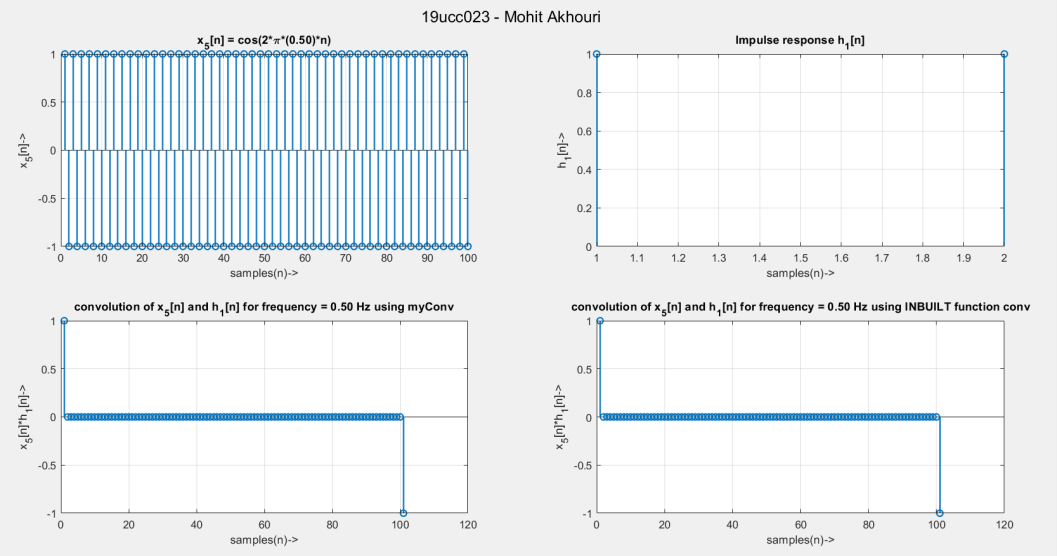


Figure 3.12 Plot of the convolution between $x_5[n]$ and $h_1[n]$

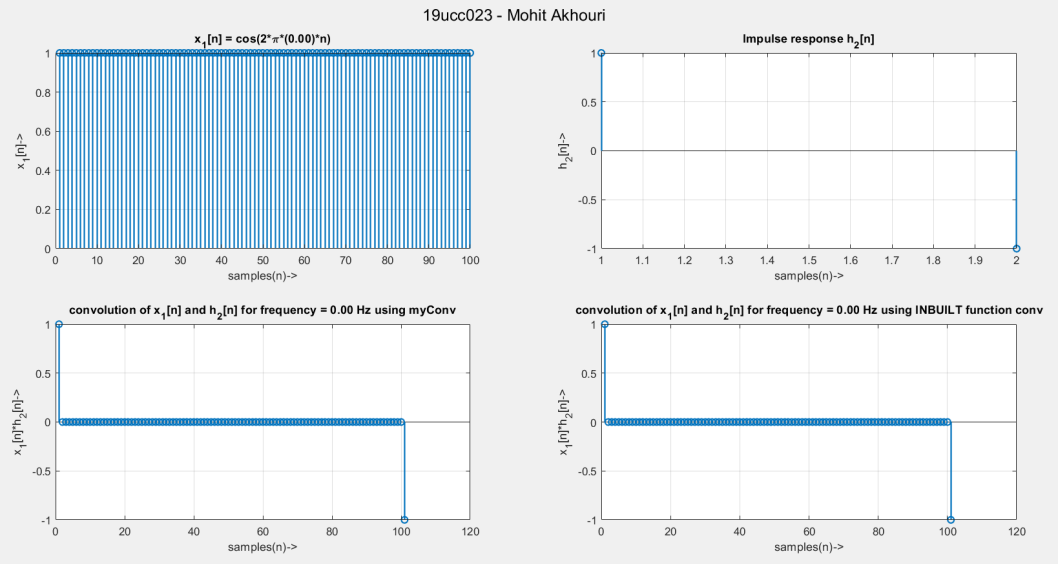


Figure 3.13 Plot of the convolution between $x_1[n]$ and $h_2[n]$

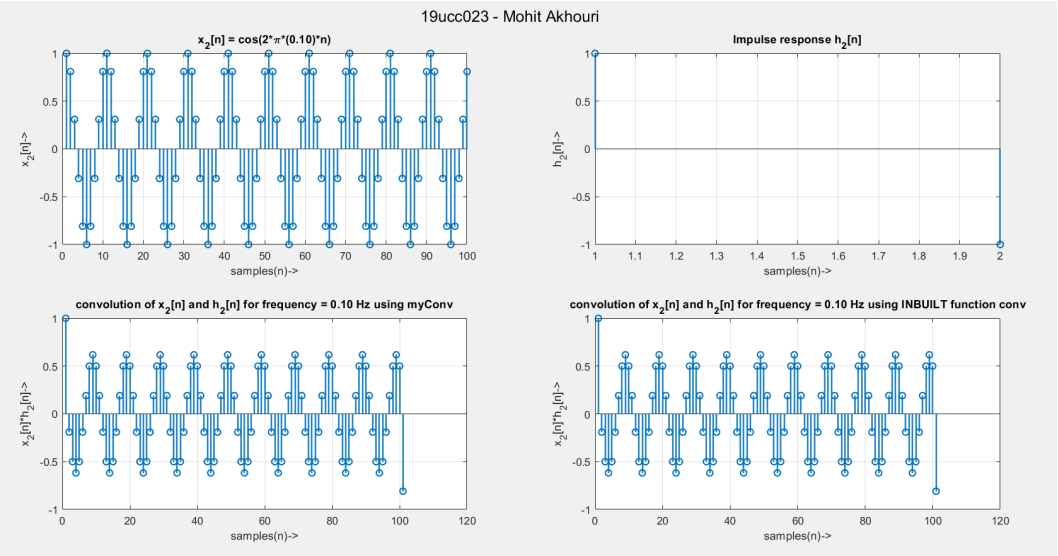


Figure 3.14 Plot of the convolution between $x_2[n]$ and $h_2[n]$

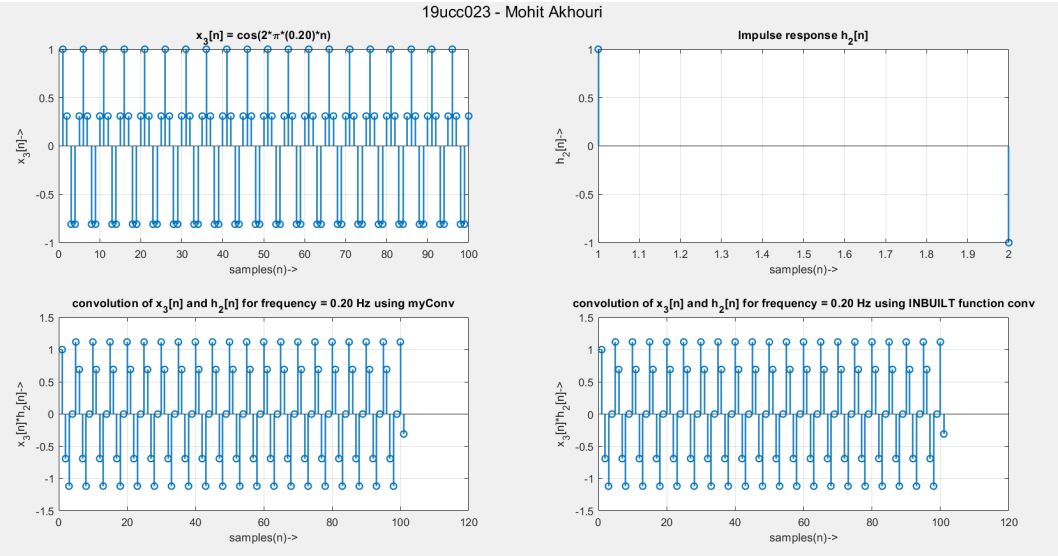


Figure 3.15 Plot of the convolution between $x_3[n]$ and $h_2[n]$

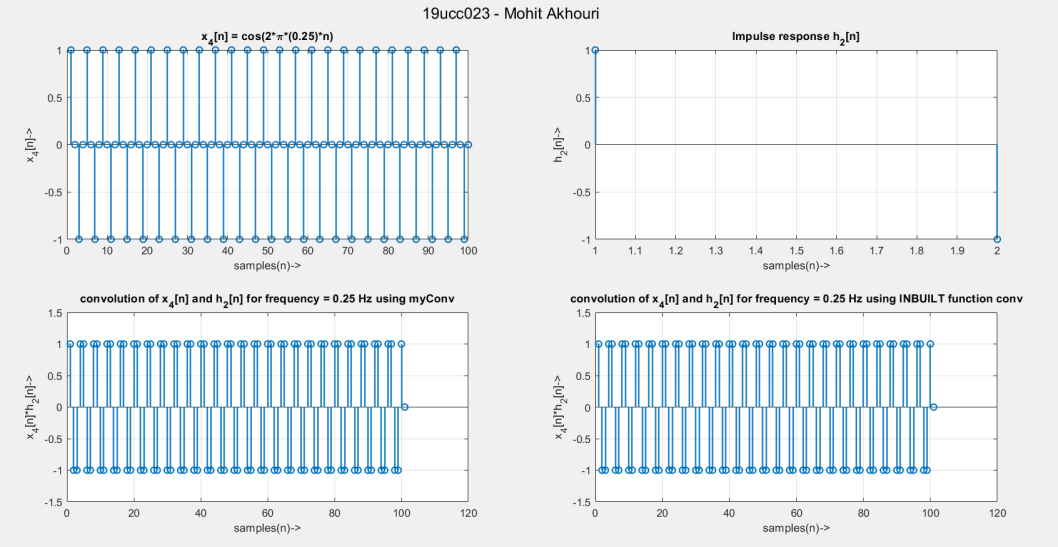


Figure 3.16 Plot of the convolution between $x_4[n]$ and $h_2[n]$

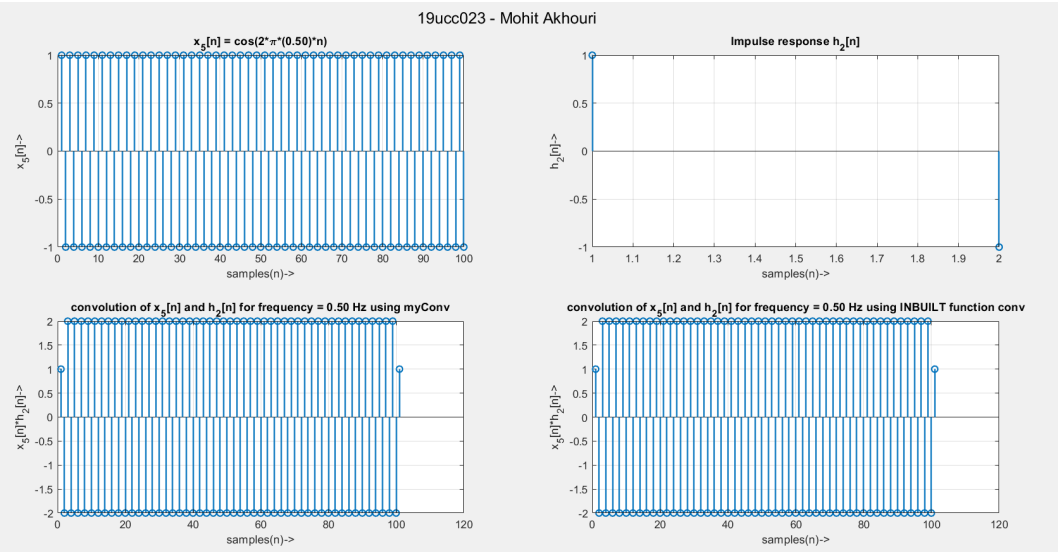


Figure 3.17 Plot of the convolution between $x_5[n]$ and $h_2[n]$

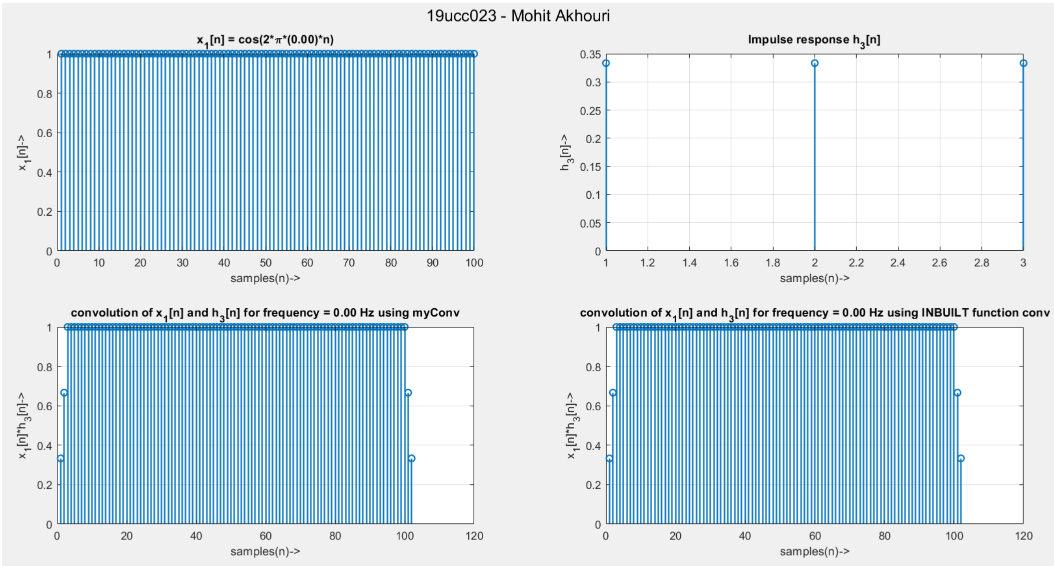


Figure 3.18 Plot of the convolution between $x_1[n]$ and $h_3[n]$

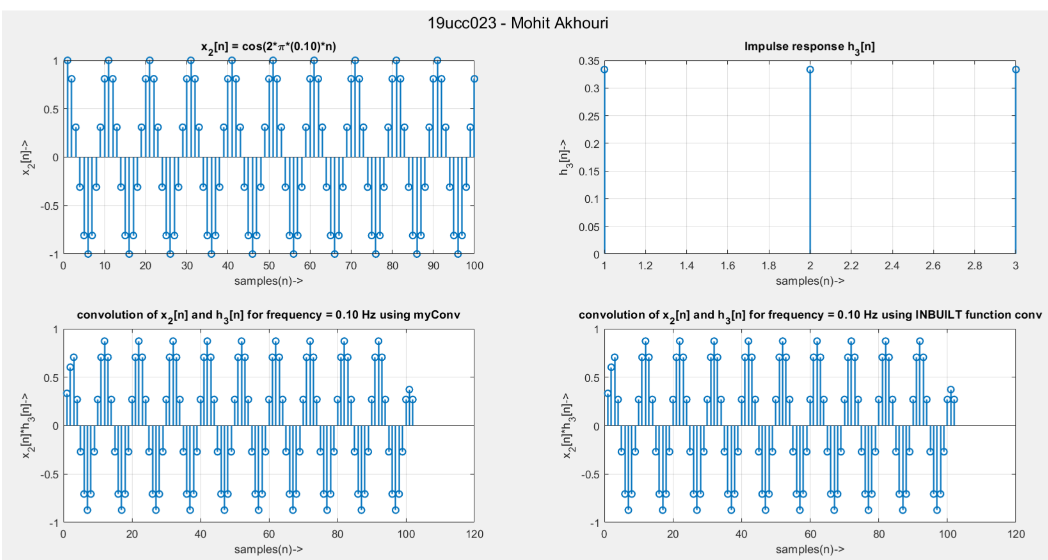


Figure 3.19 Plot of the convolution between $x_2[n]$ and $h_3[n]$

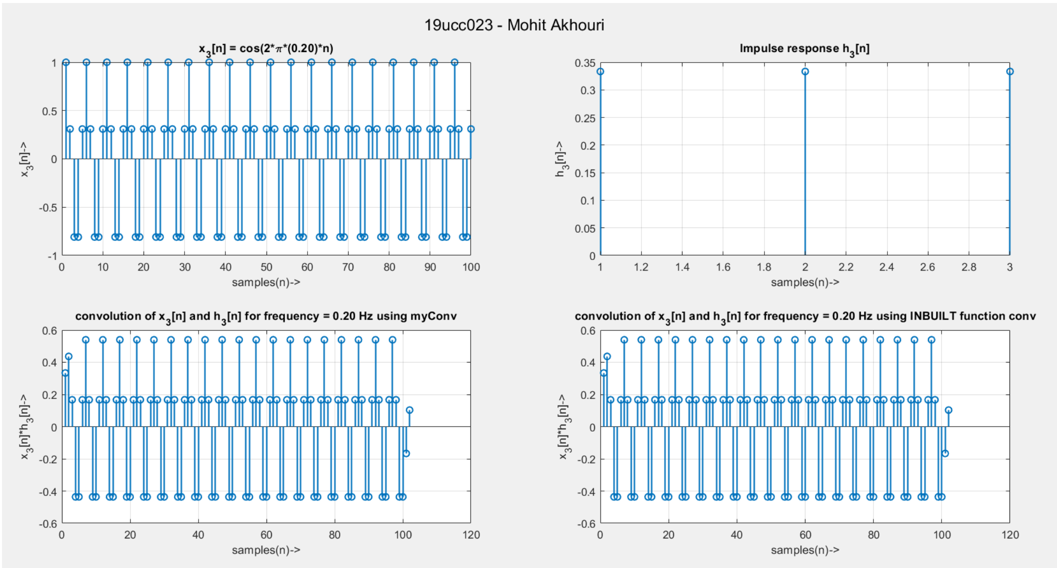


Figure 3.20 Plot of the convolution between $x_3[n]$ and $h_3[n]$

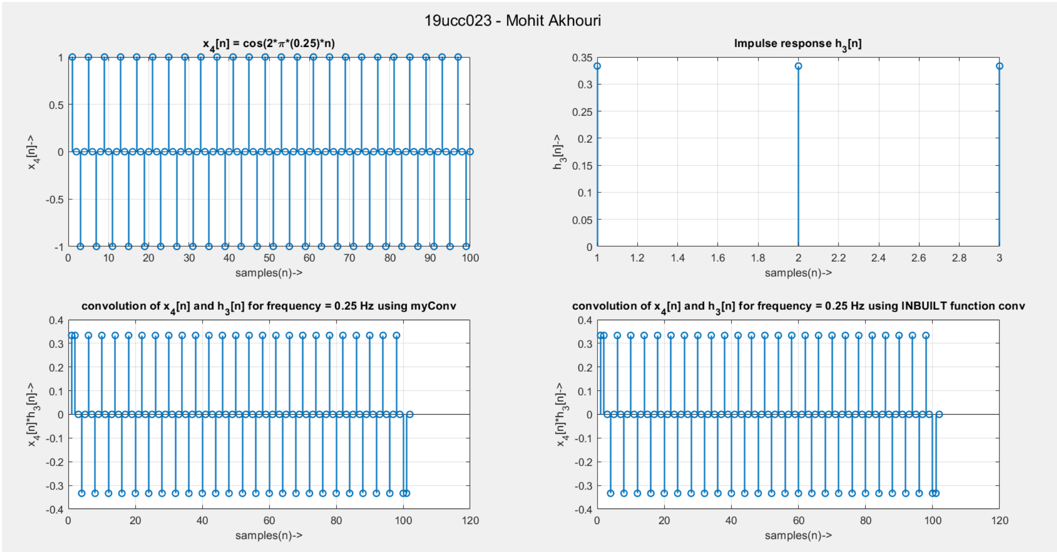


Figure 3.21 Plot of the convolution between $x_4[n]$ and $h_3[n]$

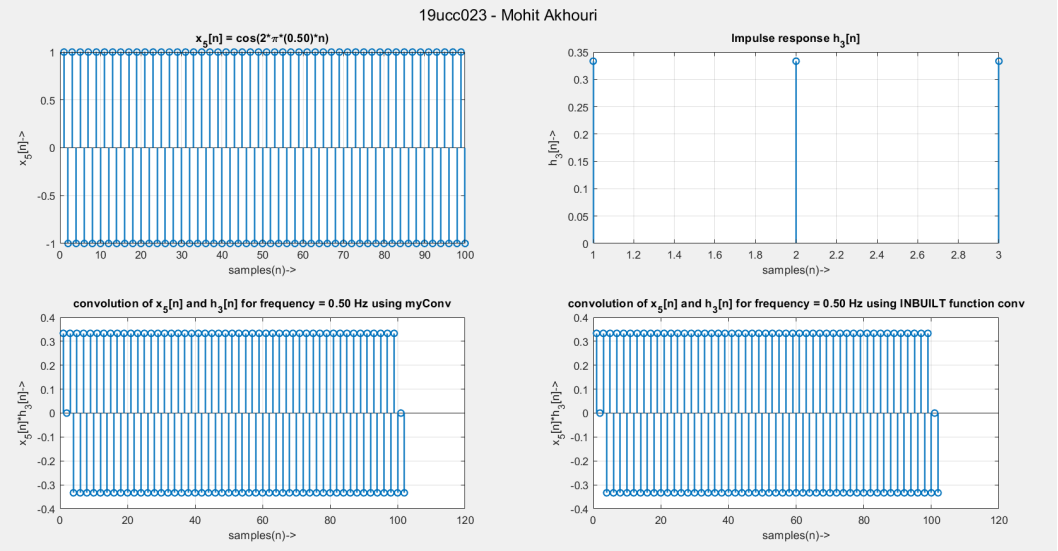


Figure 3.22 Plot of the convolution between $x_5[n]$ and $h_3[n]$

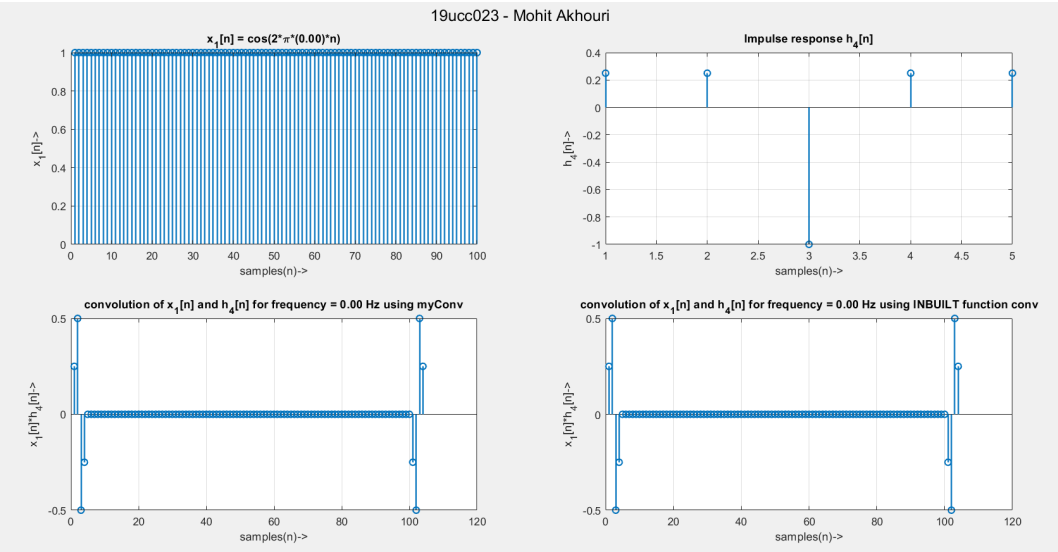


Figure 3.23 Plot of the convolution between $x_1[n]$ and $h_4[n]$

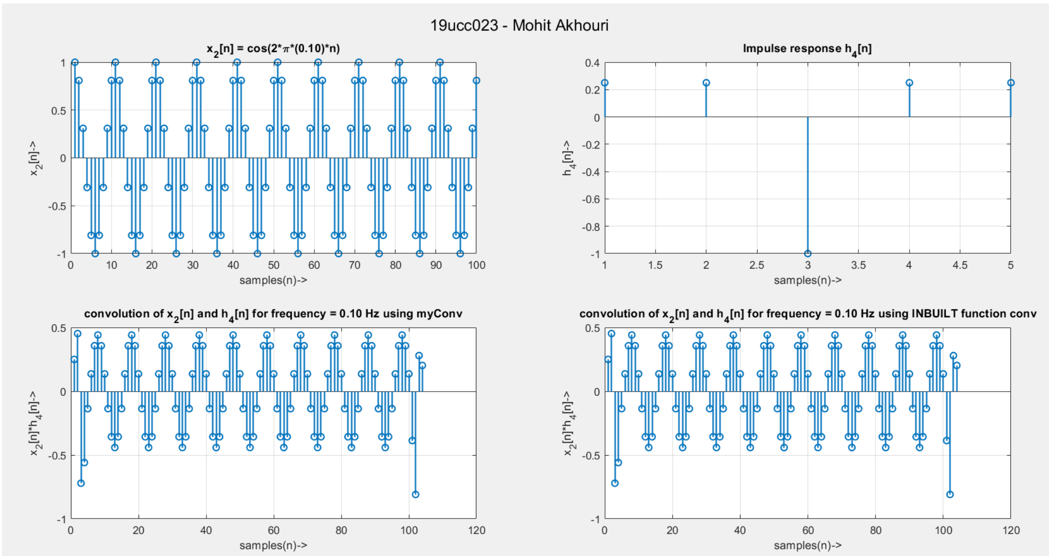


Figure 3.24 Plot of the convolution between $x_2[n]$ and $h_4[n]$

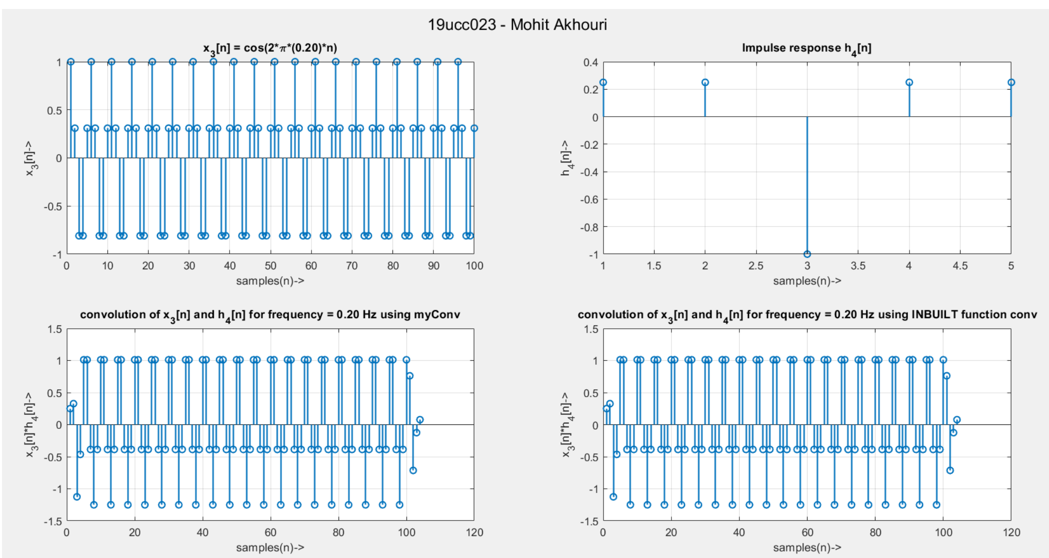


Figure 3.25 Plot of the convolution between $x_3[n]$ and $h_4[n]$

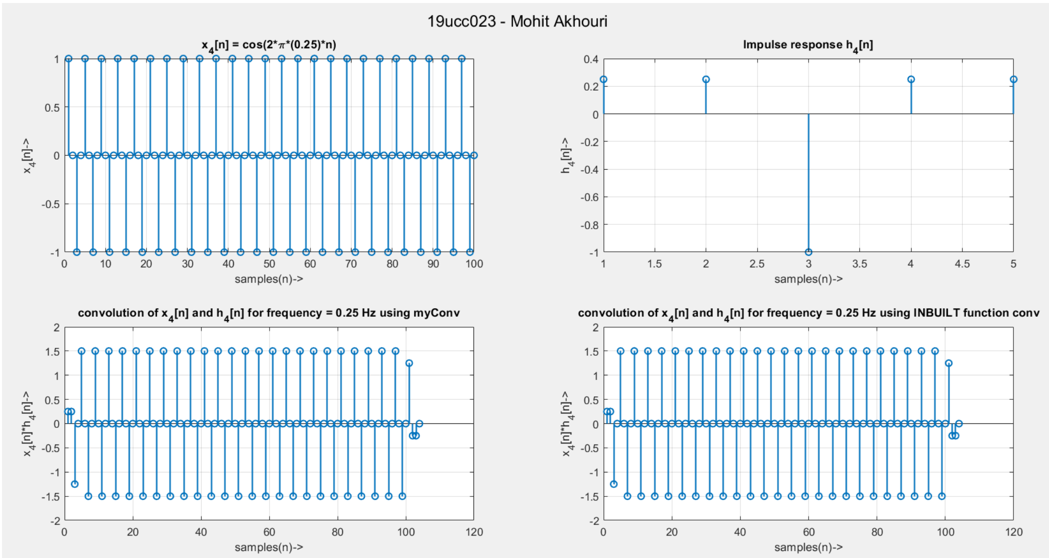


Figure 3.26 Plot of the convolution between $x_4[n]$ and $h_4[n]$

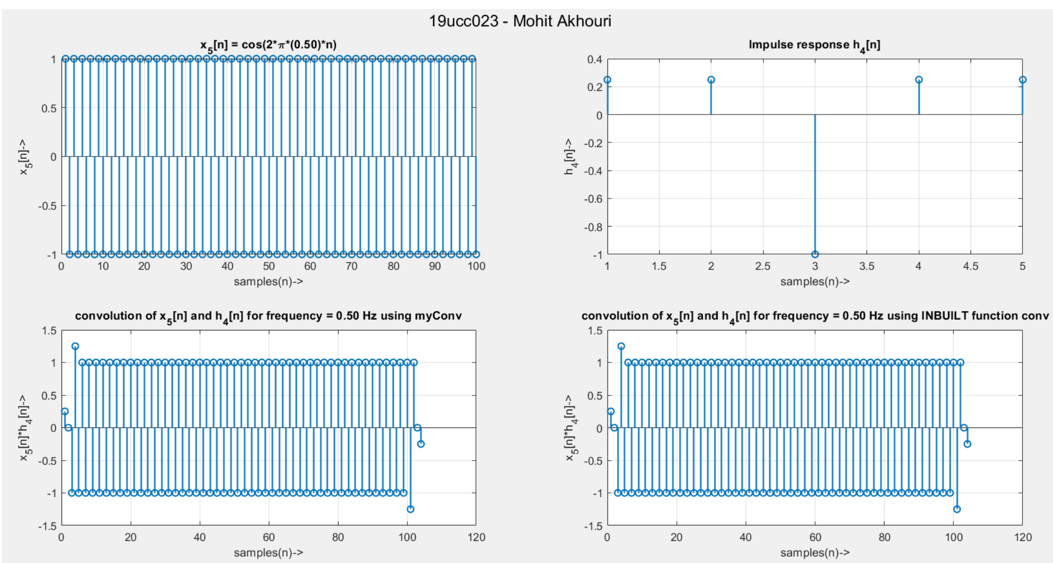


Figure 3.27 Plot of the convolution between $x_5[n]$ and $h_4[n]$

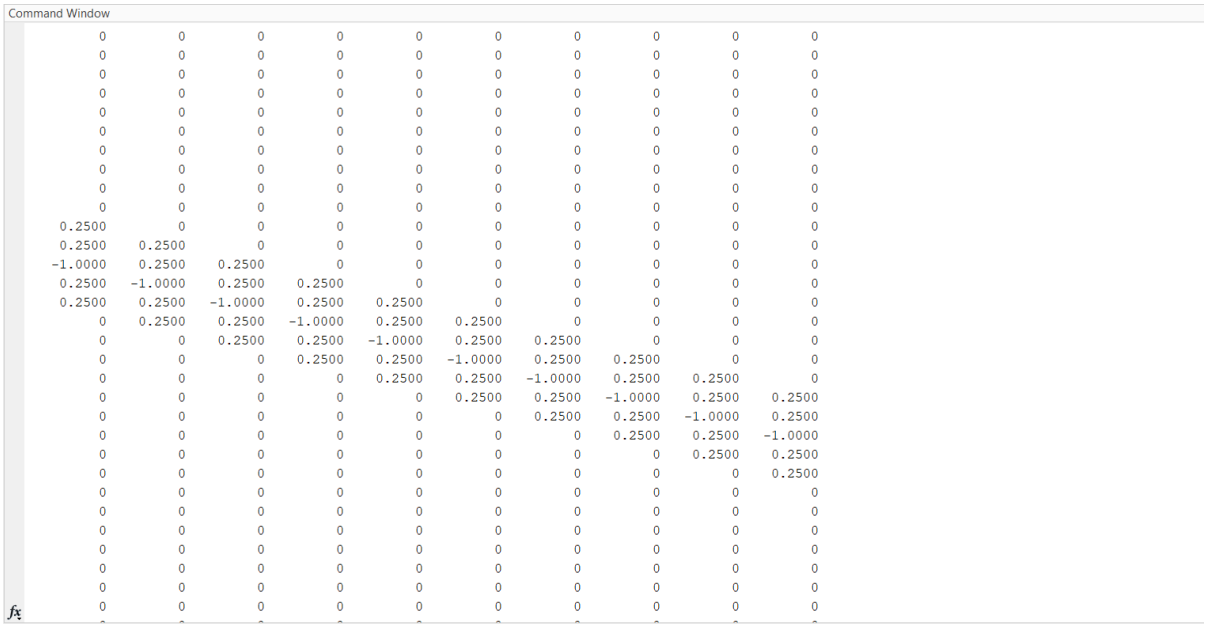


Figure 3.28 Screenshot of Convolution matrix generated by **myLinConvMat** function

3.4.2 DFT matrix generation and N-point DFT of x[n] for N=8,16,32,64:

```
% 19ucc023
% Mohit Akhouri
% Experiment 3 - Observation 3 and Observation 4

% This code will utilize the myDft function to calculate N-point DFT
% of any random sequence and compare the results with in-built fft
 function

% ALGORITHM : First we calculate the N-point DFT matrix and multiply
 the
% DFT matrix with input sequence x[n] to obtain N-point DFT

clc;
clear all;
close all;

x_8_point = rand(1,8); % A random 1 row * 8 columns array
x_16_point = rand(1,16); % A random 1 row * 16 columns array
x_32_point = rand(1,32); % A random 1 row * 32 columns array
x_64_point = rand(1,64); % A random 1 row * 64 columns array

% DFT of sequence x_8_point
dft_8_user_defined = myDft(x_8_point,8); % DFT through function myDft
dft_8_inbuilt = fft(x_8_point,8); % DFT through INBUILT function fft

% DFT of sequence x_16_point
dft_16_user_defined = myDft(x_16_point,16); % DFT through function
 myDft
dft_16_inbuilt = fft(x_16_point,16); % DFT through INBUILT function
 fft

% DFT of sequence x_32_point
dft_32_user_defined = myDft(x_32_point,32); % DFT through function
 myDft
dft_32_inbuilt = fft(x_32_point,32); % DFT through INBUILT function
 fft

% DFT of sequence x_64_point
dft_64_user_defined = myDft(x_64_point,64); % DFT through function
 myDft
dft_64_inbuilt = fft(x_64_point,64); % DFT through INBUILT function
 fft

% plotting of various signal x[n] and X(w) for different N-point
figure;
subplot(3,1,1);
stem(x_8_point,'Linewidth',1.5);
xlabel('samples(n)->');
ylabel('x[n]->');
title('RANDOM x[n] sequence for 8-point DFT');
grid on;
subplot(3,1,2);
```

Figure 3.29 Part 1 of the code for observation 3 and 4

```
stem(dft_8_user_defined,'Linewidth',1.5);
xlabel('samples(n)->');
ylabel('X[\omega]->');
title('Discrete fourier transform (DFT) of x[n] using USER-DEFINED
 myDft function');
grid on;
subplot(3,1,3);
stem(dft_8_inbuilt,'Linewidth',1.5);
xlabel('samples(n)->');
ylabel('X[\omega]->');
title('Discrete fourier transform (DFT) of x[n] using INBUILT fft
 function');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');

figure;
subplot(3,1,1);
stem(x_16_point,'Linewidth',1.5);
xlabel('samples(n)->');
ylabel('x[n]->');
title('RANDOM x[n] sequence for 16-point DFT');
grid on;
subplot(3,1,2);
stem(dft_16_user_defined,'Linewidth',1.5);
xlabel('samples(n)->');
ylabel('X[\omega]->');
title('Discrete fourier transform (DFT) of x[n] using USER-DEFINED
 myDft function');
grid on;
subplot(3,1,3);
stem(dft_16_inbuilt,'Linewidth',1.5);
xlabel('samples(n)->');
ylabel('X[\omega]->');
title('Discrete fourier transform (DFT) of x[n] using INBUILT fft
 function');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');

figure;
subplot(3,1,1);
stem(x_32_point,'Linewidth',1.5);
xlabel('samples(n)->');
ylabel('x[n]->');
title('RANDOM x[n] sequence for 32-point DFT');
grid on;
subplot(3,1,2);
stem(dft_32_user_defined,'Linewidth',1.5);
xlabel('samples(n)->');
ylabel('X[\omega]->');
title('Discrete fourier transform (DFT) of x[n] using USER-DEFINED
 myDft function');
grid on;
subplot(3,1,3);
stem(dft_32_inbuilt,'Linewidth',1.5);
```

Figure 3.30 Part 2 of the code for observation 3 and 4

```
xlabel('samples(n)->');
ylabel('X[\omega]->');
title('Discrete fourier transform (DFT) of x[n] using INBUILT fft
 function');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');

figure;
subplot(3,1,1);
stem(x_64_point,'Linewidth',1.5);
xlabel('samples(n)->');
ylabel('x[n]->');
title('RANDOM x[n] sequence for 64-point DFT');
grid on;
subplot(3,1,2);
stem(dft_64_user_defined,'Linewidth',1.5);
xlabel('samples(n)->');
ylabel('X[\omega]->');
title('Discrete fourier transform (DFT) of x[n] using USER-DEFINED
 myDft function');
grid on;
subplot(3,1,3);
stem(dft_64_inbuilt,'Linewidth',1.5);
xlabel('samples(n)->');
ylabel('X[\omega]->');
title('Discrete fourier transform (DFT) of x[n] using INBUILT fft
 function');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');
```

Figure 3.31 Part 3 of the code for observation 3 and 4

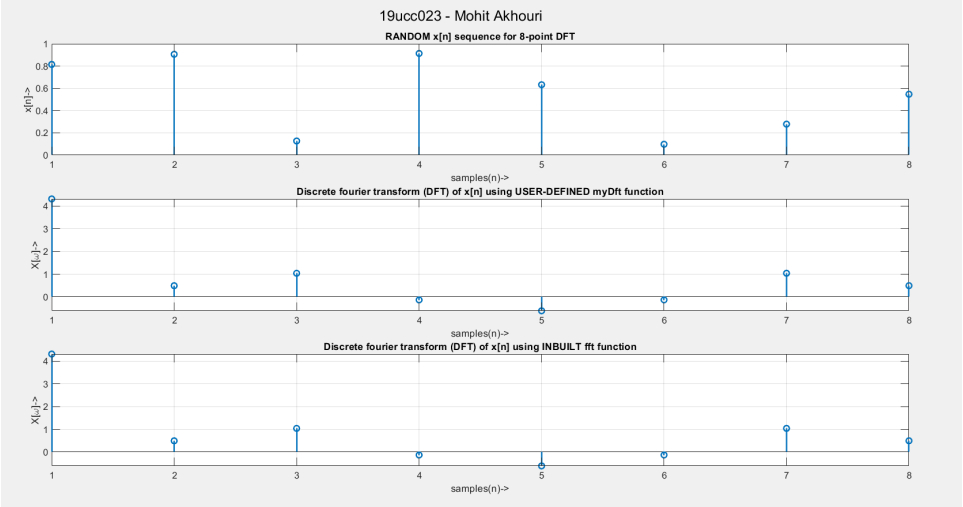


Figure 3.32 Plot of the 8-point DFT of random sequence x[n]

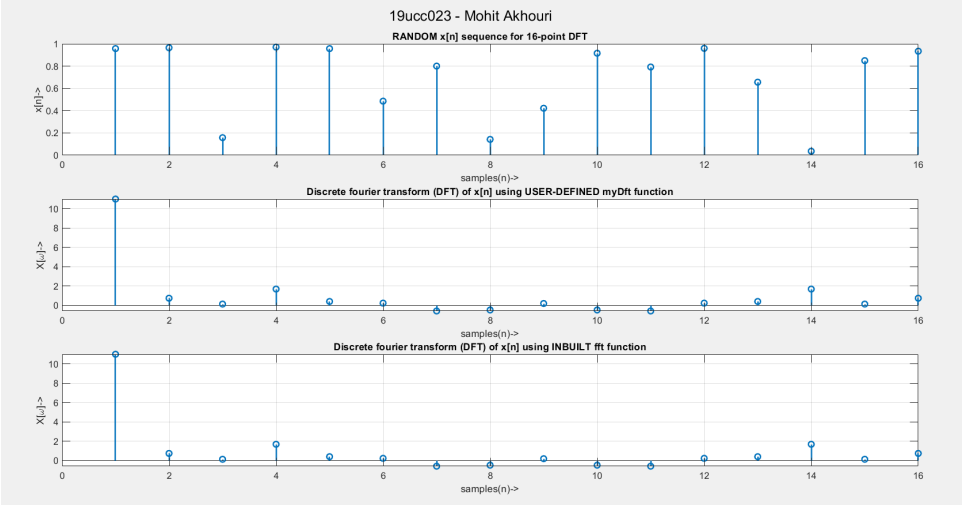


Figure 3.33 Plot of the 16-point DFT of random sequence x[n]

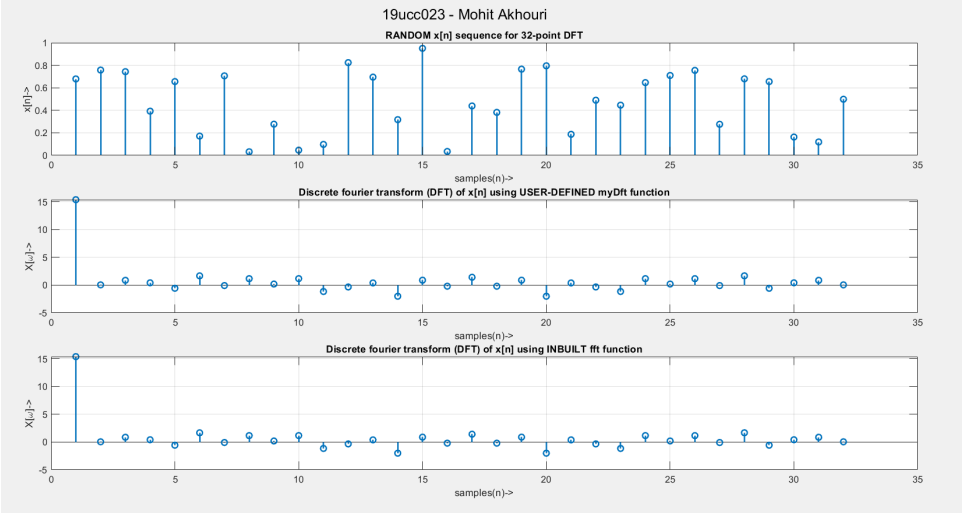


Figure 3.34 Plot of the 32-point DFT of random sequence x[n]

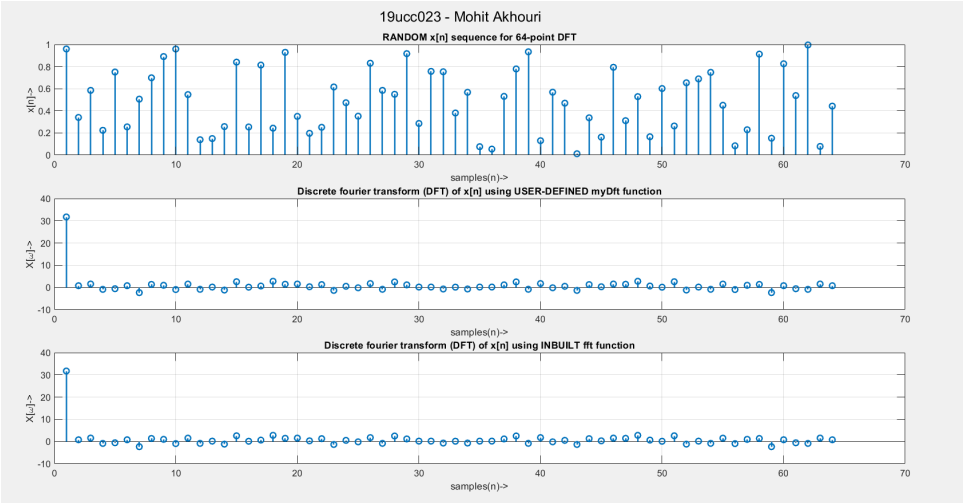


Figure 3.35 Plot of the 64-point DFT of random sequence x[n]

```
The DFT matrix is given as :
  Columns 1 through 5

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

  Columns 6 through 8

   1.0000 + 0.0000i   1.0000 + 0.0000i   1.0000 + 0.0000i
  -0.7071 + 0.7071i  -0.0000 + 1.0000i   0.7071 + 0.7071i
   0.0000 - 1.0000i  -1.0000 - 0.0000i  -0.0000 + 1.0000i
   0.7071 + 0.7071i   0.0000 - 1.0000i  -0.7071 + 0.7071i
  -1.0000 - 0.0000i   1.0000 + 0.0000i  -1.0000 - 0.0000i
   0.7071 - 0.7071i  -0.0000 + 1.0000i  -0.7071 - 0.7071i
  -0.0000 + 1.0000i  -1.0000 - 0.0000i  -0.0000 - 1.0000i
  -0.7071 - 0.7071i  -0.0000 - 1.0000i   0.7071 - 0.7071i
```

Figure 3.36 Screenshot of 8-point DFT matrix generated by the function **myDft**

3.4.3 Convolution in Simulink :

```
% 19ucc023
% Mohit Akhouri
% Experiment 3 - Observation 5

% This code will deploy the simulink model to calculate convolution of
 two
% random sequences x[n] and h[n]

sim('Simulink_Observation_5'); % calling the simulink model
y_conv_inbuilt = conv(out.xn.data,out.hn.data); % Convolution from
 INBUILT function for cross-checking

% plotting of signals x[n],h[n] and convolved signal y[n]

figure;
subplot(2,2,1);
stem(out.xn.data,'Linewidth',1.5);
xlabel('samples(n)->');
ylabel('x[n]->');
title('random input signal x[n] from Simulink Model');
grid on;
subplot(2,2,2);
stem(out.hn.data,'Linewidth',1.5);
xlabel('samples(n)->');
ylabel('h[n]->');
title('random impulse response h[n] from Simulink Model');
grid on;
subplot(2,2,3);
stem(out.yn.data,'Linewidth',1.5);
xlabel('samples(n)->');
ylabel('y[n] = x[n] * h[n]');
title('Convolved signal y[n] obtained from Simulink Model');
grid on;
subplot(2,2,4);
stem(y_conv_inbuilt,'Linewidth',1.5);
xlabel('samples(n)->');
ylabel('y[n] = x[n] * h[n]');
title('Convolved signal y[n] obtained from INBUILT function conv');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');
```

Figure 3.37 Code for the observation 5

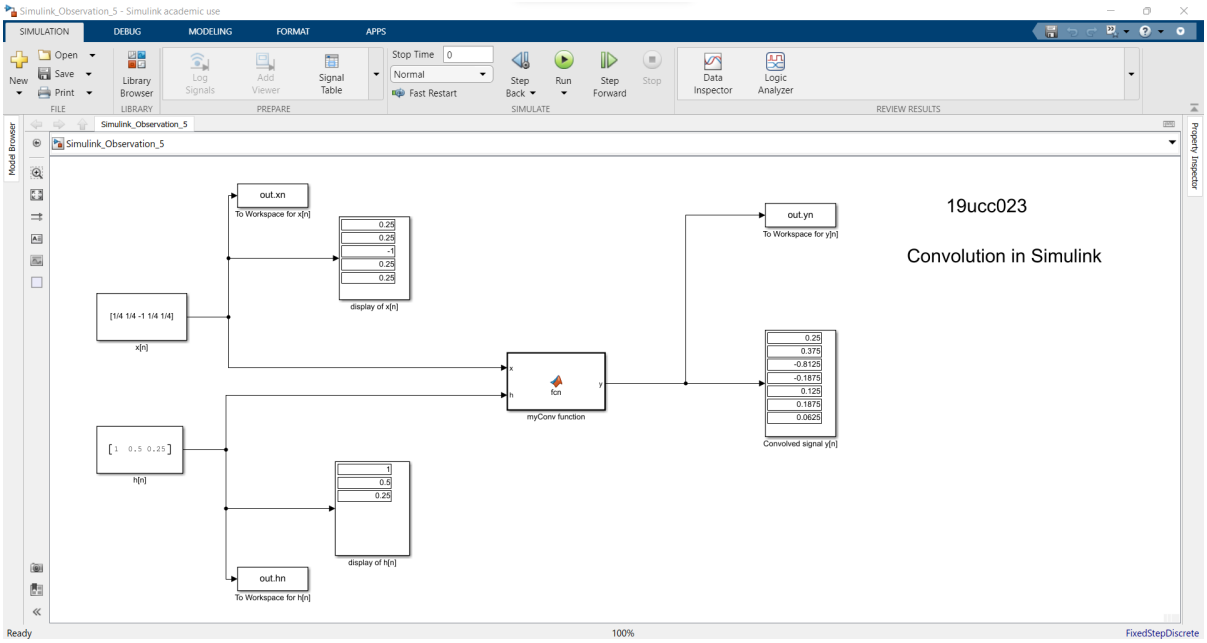


Figure 3.38 Simulink Model used for Convolution

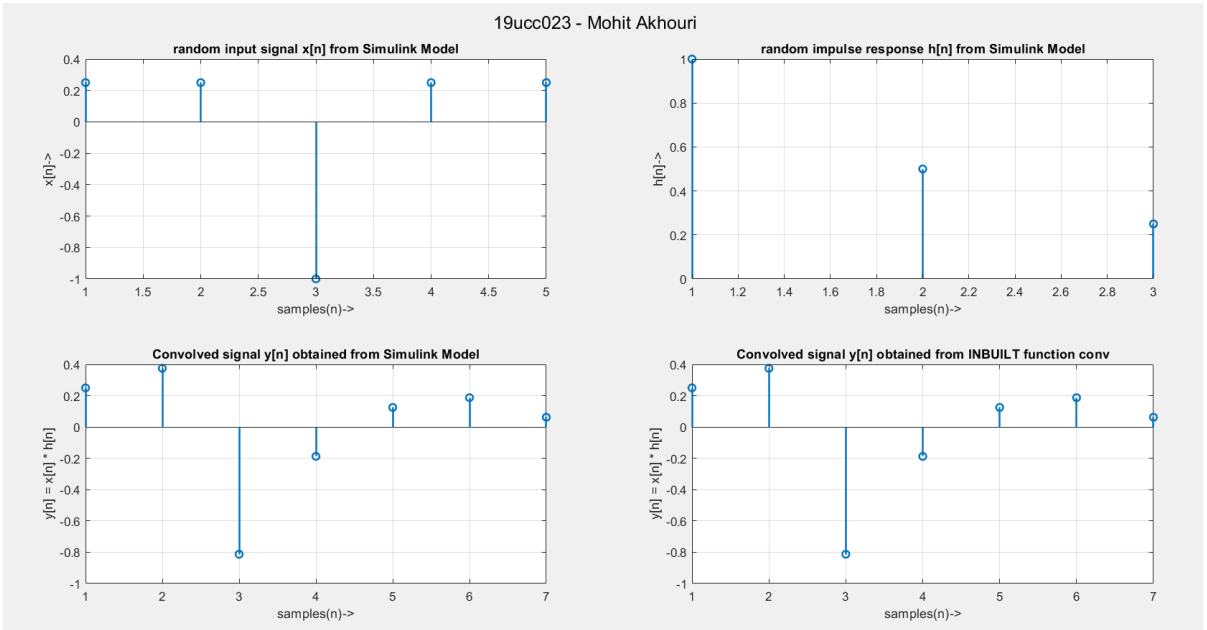


Figure 3.39 Plot of the convolution obtained between x[n] and h[n] from Simulink Model

3.4.4 Functions used in main codes for Convolution and DFT :

3.4.4.1 myLinConvMat.m function code :

```
function [M] = myLinConvMat(h,n)
% This function computes the convolution matrix M for impulse response
 h(n)
% with 'n' being the length of the input sequence 'x'

% 19ucc023
% Mohit Akhouri

rows = n; % initializing the number of rows of Convolution matrix M
length_h = length(h); % declaring the length of 'h'
columns = length(h) + n - 1; % declaring the number of columns of
 matrix M
M = zeros(rows,columns); % initializing the convolution matrix with
 zeros

% generating the convolution matrix (ALGORITHM)

for i=1:rows
    offset = i; % offset in the matrix from where 'h' needs to be
 stored
    for j=1:length_h
        M(i,offset) = h(j);
        offset = offset + 1; % offset incremented
    end
end
disp(M);
end
```

Published with MATLAB® R2020b

Figure 3.40 myLinConvMat function for computation of Convolution matrix

3.4.4.2 myConv function code :

```
function [y] = myConv(x,M)
% This function computes the convolution of 'x' and 'h'
% using convolution matrix M where x = input sequence

% 19ucc023
% Mohit Akhouri

rows = size(M,1); % storing number of rows of matrix M
columns = size(M,2); % storing number of columns of matrix M

y = zeros(1,columns); % initializing the variable to store convolution
 of sequences
sum = 0; % temporary variable to store sum

% ALGORITHM for calculation of convolution of sequences is as follows
for i=1:columns
    sum = 0;
    for j=1:rows
       sum = sum + (M(j,i)*x(j));
    end
    y(i) = sum;
end
end
```

Published with MATLAB® R2020b

Figure 3.41 myConv function for computation of Convolution from Convolution matrix M

3.4.4.3 myDft function code :

```
function [X] = myDft(x,N)
% This function will calculate the discrete fourier transform
% of input sequence x[n] , this function calculates N-point DFT

% 19ucc023
% Mohit Akhouri

% calculating the DFT matrix 'D'
D = zeros(N,N); % DFT matrix to store the values of twiddle factor
twd_factor = 0; % to store the value of twiddle factor

for n=1:N
    for k=1:N
        twd_factor = exp(-1j*2*pi*(k-1)*(n-1)/N);
        D(n,k) = twd_factor;
    end
end

disp('The DFT matrix is given as :');
disp(D);

% The ALGORITHM for calculation of DFT is as follows
X = zeros(1,N);
for i=1:N
    sum = 0;
    for j=1:N
        sum = sum+(D(i,j)*x(j));
    end
    X(i)=sum;
end

end
```

Published with MATLAB® R2020b

Figure 3.42 myDft function for computation of DFT matrix and N-point DFT

3.5 Conclusion

In this experiment , we learnt the concepts of **Linear Convolution** and **Discrete Fourier Transform (DFT)** of Digital Signal Processing. We learnt about convolution matrix and DFT matrix and how to utilize them to compute convolution and N-point DFT. We also compared the results with inbuilt functions like **fft** and **conv**. We also learnt about **Twiddle factor** utilization in the construction of the DFT matrix. We also performed the convolution in Simulink through various blocks and compared the results obtained from the Simulink model with results obtained from MATLAB code.

Chapter *4*

Experiment - 4

4.1 Aim of the Experiment

- Circular Convolution and DFT Multiplication for two sequences
- Simulink based circular convolution

4.2 Software Used

- MATLAB
- Simulink

4.3 Theory

4.3.1 About Circular Convolution :

Circular convolution, also known as **cyclic convolution**, is a special case of **periodic convolution**, which is the convolution of two periodic functions that have the same period. Periodic convolution arises, for example, in the context of the discrete-time Fourier transform (DTFT). In particular, the DTFT of the product of two discrete sequences is the periodic convolution of the DTFTs of the individual sequences. And each DTFT is a periodic summation of a continuous Fourier transform function. Although DTFTs are usually continuous functions of frequency, the concepts of periodic and circular convolution are also directly applicable to discrete sequences of data. In that context, **circular convolution** plays an **important role** in **maximizing the efficiency of a certain kind of common filtering operation**.

Circular convolution can also be obtained through **change of basis**. In the first method, by multiplying **DFT of two sequences** we obtain their circular convolution if we take **Inverse DFT** of the product. In the second method , We can obtain Circular Convolution by product of the **Convolution matrix** , **DFT matrix** and **inverse DFT matrix**.

The circular convolution for a system with **system response** h[n] and **input sequence** x[n] (both are padded with zeros to make them equal in size) is as follows :

$$y(n) = \sum_{i=0}^{N-1} h(i).x((n-i))_N \tag{4.1}$$

In the above equation , **N** is the **larger sequence length** and it is taken as a multiple of 2.N = 2^i where i can be 2,3,4 etc. In the equation the **term** $x((n-i))_N$ represents the **index** $N - i + n^{'}$.

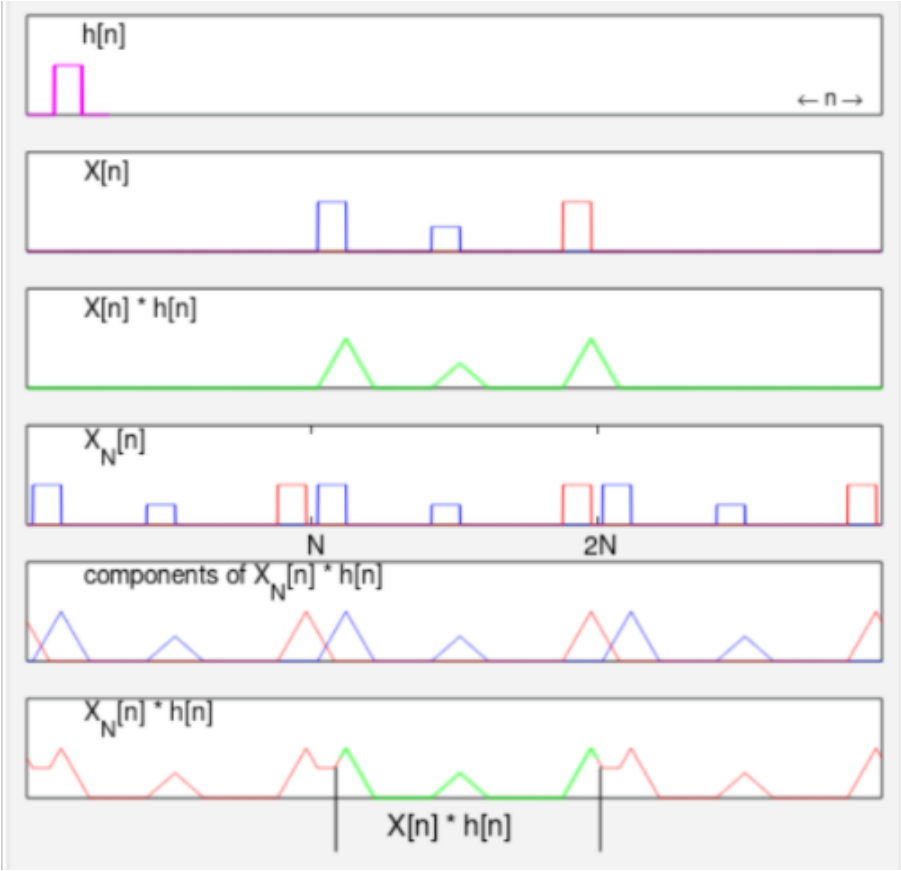


Figure 4.1 Example of Circular Convolution

4.3.1.1 Change of Basis

Circular Convolution can also be obtained for two sequences x[n] and h[n] by :

- Taking Inverse DFT of product of DFT of x[n] and h[n]
- Through Matrix Multiplication method

4.4 Code and results

4.4.1 Circular Convolution of two pair of sequences using Circular Convolution matrix :

```
% 19ucc023
% Mohit Akhouri
% Experiment 4 - Observation 1 and Observation 2

% This code will calculate circular convolution between two sets
% of x[n] and h[n] and compare them with inbuilt command
% for circular convolution 'cconv(x,y)'

% ALGORITHM : call the my_Circular_Convolution function

clc;
clear all;
close all;

h = [2 1 2 1]; % initializing h1
x1 = [1 2 3 4]; % initializing x1 (first input sequence)

% initializing second input sequence x2
n = 0:1:9; % initializing samples for x2[n]
x2 = zeros(1,length(n)); % initializing x2[n] with zeros

for i=1:length(n)
    x2(i) = (0.5)^(n(i)); % calculating the sample values of x2[n]
end

length_h = length(h); % length of impulse response h[n]
length_x1 = length(x1); % length of input sequence x1[n]
length_x2 = length(x2); % length of input sequence x2[n]

% calling my_Circular_Convolution to calculate circular convolution
% of x1[n] and h[n] , x2[n] and h[n] and compare the results obtained
% with inbuilt command for circular convolution 'cconv(x,y,n)'

% circular convolution from USER_DEFINED function
 my_Circular_Convolution
circ_conv_x1h = my_Circular_Convolution(x1,h);
circ_conv_x2h = my_Circular_Convolution(x2,h);

% circular convolution from INBUILT function cconv(x,y,n)
circ_conv_x1h_inbuilt = cconv(x1,h,max(length_x1,length_h));
circ_conv_x2h_inbuilt = cconv(x2,h,max(length_x2,length_h));

% plotting the results obtained for circular convolution

% circular convolution between x1[n] and h[n]
figure;
subplot(2,2,1);
stem(x1,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('x_{1}[n] ->');
title('plot of input sequence x_{1}[n]');
grid on;
```

Figure 4.2 Part 1 of the code for observation 1 and 2

```
subplot(2,2,2);
stem(h,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('h[n] ->');
title('plot of impulse response h[n]');
grid on;
subplot(2,2,3);
stem(circ_conv_x1h,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel(' x_{1}[n] \otimes h[n]->');
title('Circular convolution of x_{1}[n] and h[n] using my\_Circular
\_Convolution');
grid on;
subplot(2,2,4);
stem(circ_conv_x1h_inbuilt,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel(' x_{1}[n] \otimes h[n]->');
title('Circular convolution of x_{1}[n] and h[n] using INBUILT
 function cconv');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');

% circular convolution between x2[n] and h[n]
figure;
subplot(2,2,1);
stem(x2,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('x_{2}[n] ->');
title('plot of input sequence x_{2}[n]');
grid on;
subplot(2,2,2);
stem(h,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('h[n] ->');
title('plot of impulse response h[n]');
grid on;
subplot(2,2,3);
stem(circ_conv_x2h,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel(' x_{2}[n] \otimes h[n]->');
title('Circular convolution of x_{2}[n] and h[n] using my\_Circular
\_Convolution');
grid on;
subplot(2,2,4);
stem(circ_conv_x2h_inbuilt,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel(' x_{2}[n] \otimes h[n]->');
title('Circular convolution of x_{2}[n] and h[n] using INBUILT
 function cconv');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');
```

Figure 4.3 Part 2 of the code for observation 1 and 2

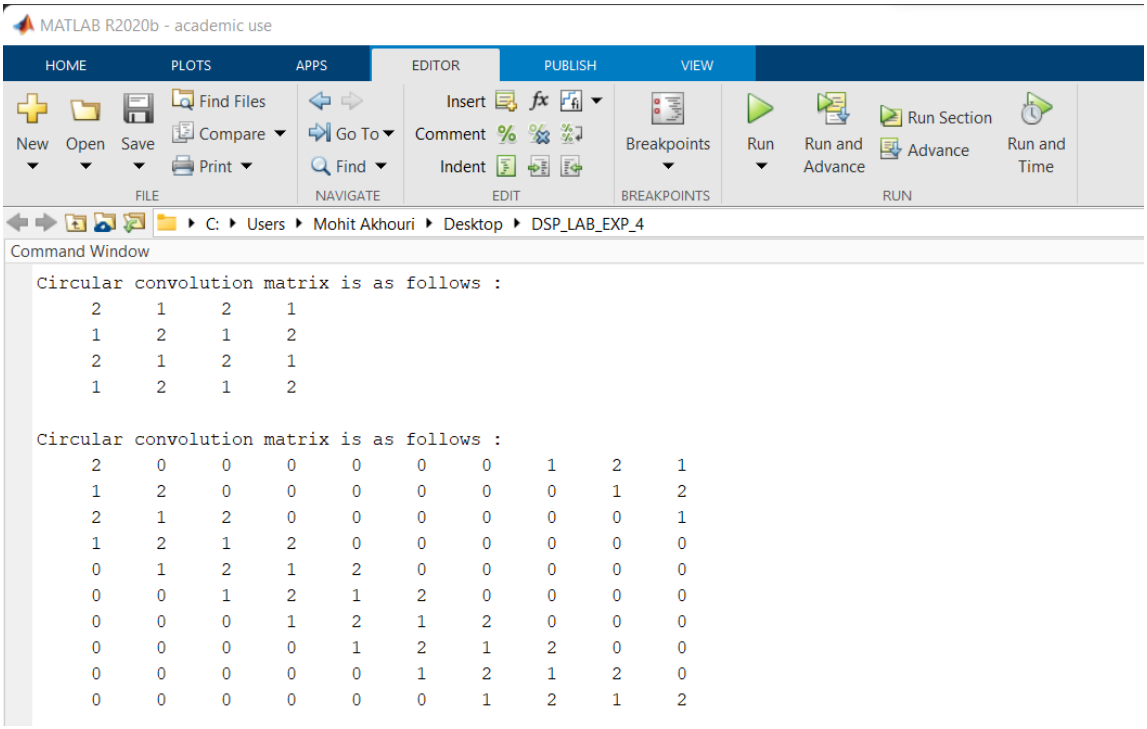


Figure 4.4 Displaying Circular Convolution matrices of **h[n]** for $x_1[n]$ and $x_2[n]$

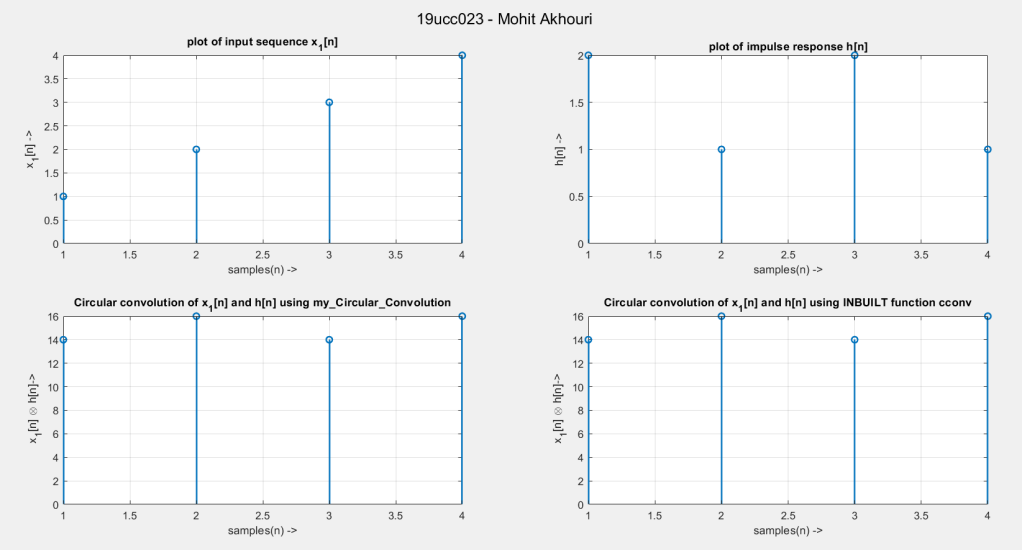


Figure 4.5 Plot of Circular Convolution between $x_1[n]$ and h[n]

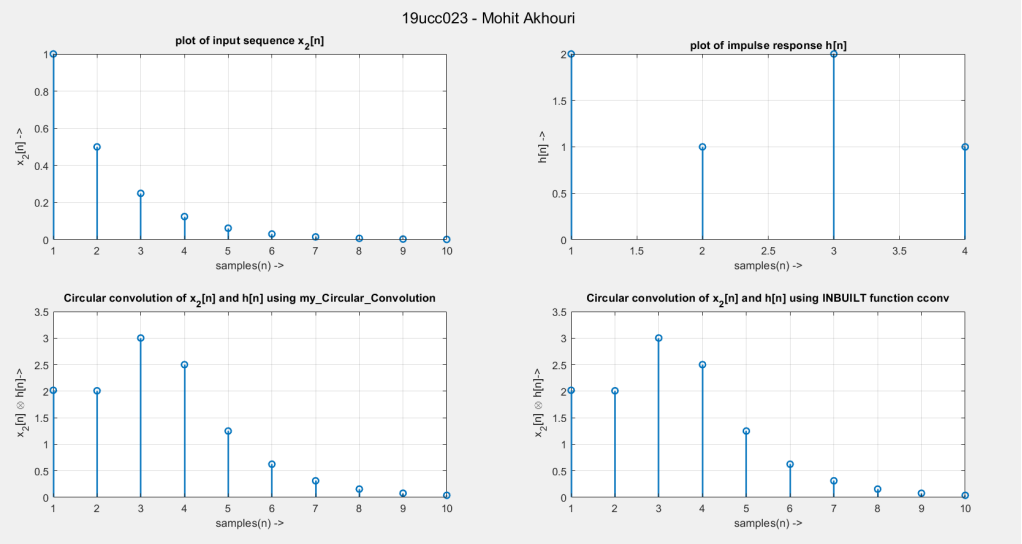


Figure 4.6 Plot of Circular Convolution between $x_2[n]$ and h[n]

4.4.2 Change of Basis :

4.4.2.1 Verify Circular Convolution $\Longleftrightarrow$ DFT multiplication of Fourier Transform Pair :

```
% 19ucc023
% Mohit Akhouri
% Experiment 4 - Observation 3

% This code will verify the property that :
% if x[n] and h[n] are sequences then CIRCULAR CONVOLUTION
% of x[n] and h[n] is equal to IDFT(X(k)*H(k))

clc;
clear all;
close all;

% ALGORITHM : Initialize x[n] and h[n]
% First calculate DFT of x[n] and h[n] , let it be X(k) and H(k)
% Now , multiply X(k) and H(k) point by point and finally pass them
% thtough myIDFT function to calculate IDFT. IDFT obtained can now be
% compared with the circular convolution of x[n] and h[n]

% randperm(N,K) : returns a random permutation of K integers from 1 to
 N

x = randperm(8,8); % taking a random sequence x[n]
h = randperm(8,8); % taking a random sequence h[n]

% calculating the circular convolution of x[n] and h[n]

circ_conv_xh = my_Circular_Convolution(x,h);

% Finding the 8-point DFT sequence of x[n] and h[n]

[X,Dx] = myDFT(x,8); % DFT of input sequence x[n] , Dx is DFT matrix
 of x[n]
[H,Dh] = myDFT(h,8); % DFT of impulse response h[n] , Dh is DFT matrix
 of h[n]

% Multiplying X(k) and H(k) sample by sample ( point-by-point )

Y = X .* H; % point-by-point multiplication of X(k) and H(k)

% Finding the IDFT of Y(k)

y = myIDFT(Y,8); % IDFT of Y(k) to be compared with circ_conv_xh

% Plotting the results and comparing the IDFT y[n] with circular
% convolution circ_conv_xh

% plotting x[n],h[n],X(k) and H(k)
figure;
subplot(2,2,1);
stem(x,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('x[n] ->');
```

Figure 4.7 Part 1 of the code for observation 3

```
title('plot of input sequence x[n]');
grid on;
subplot(2,2,2);
stem(h,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('h[n] ->');
title('plot of impulse response h[n]');
grid on;
subplot(2,2,3);
stem(X,'Linewidth',1.5);
xlabel('samples(k) ->');
ylabel('X(k) ->');
title('DFT of input sequence x[n]');
grid on;
subplot(2,2,4);
stem(H,'Linewidth',1.5);
xlabel('samples(k) ->');
ylabel('H(k) ->');
title('DFT of impulse response h[n]');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');

% plotting Y(k) = X(k) .* H(k)
figure;
stem(Y,'Linewidth',1.5);
xlabel('samples(k) ->');
ylabel('Y(k) ->');
title('19ucc023 - Mohit Akhouri','Y(k) = X(k)*H(k) : MULTIPLICATION of
 X(k) and H(k) point-by-point in frequency domain');
grid on;

% plotting y[n] = IDFT(Y(k)) and circ_conv_xh for comparison of plots
figure;
subplot(2,1,1);
stem(y,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('y[n] ->');
title('IDFT of seqeunce Y(k)');
grid on;
subplot(2,1,2);
stem(circ_conv_xh,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('x[n] \otimes h[n] ->');
title('Circular Convolution of x[n] and h[n] using my\_Circular
\_Convolution');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');
```

Figure 4.8 Part 2 of the code for observation 3

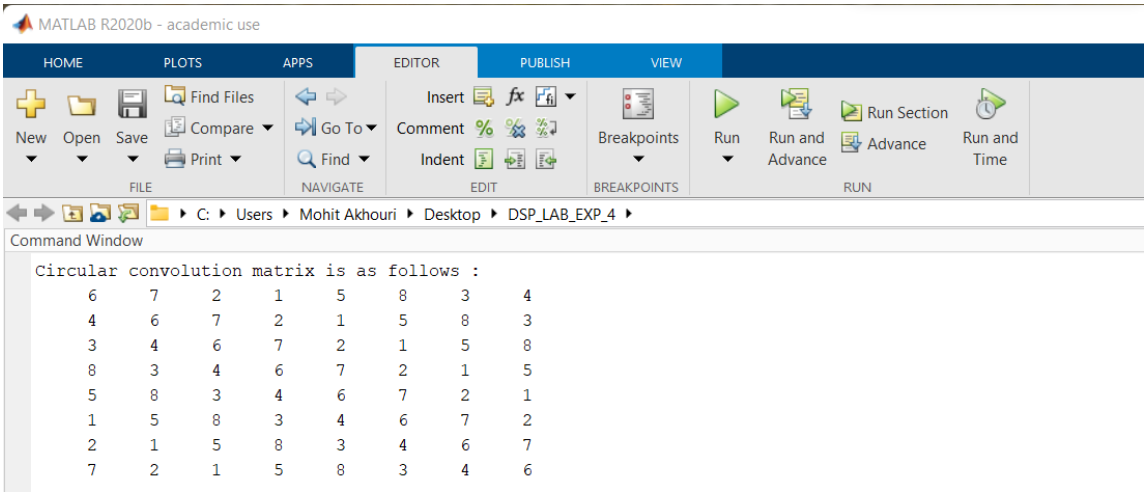


Figure 4.9 Display of Circular Convolution matrix for impulse response h[n]

```
The DFT matrix is given as :
  Columns 1 through 5

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

  Columns 6 through 8

   1.0000 + 0.0000i   1.0000 + 0.0000i   1.0000 + 0.0000i
  -0.7071 + 0.7071i  -0.0000 + 1.0000i   0.7071 + 0.7071i
   0.0000 - 1.0000i  -1.0000 - 0.0000i  -0.0000 + 1.0000i
   0.7071 + 0.7071i   0.0000 - 1.0000i  -0.7071 + 0.7071i
  -1.0000 - 0.0000i   1.0000 + 0.0000i  -1.0000 - 0.0000i
   0.7071 - 0.7071i  -0.0000 + 1.0000i  -0.7071 - 0.7071i
  -0.0000 + 1.0000i  -1.0000 - 0.0000i  -0.0000 - 1.0000i
  -0.7071 - 0.7071i  -0.0000 - 1.0000i   0.7071 - 0.7071i
```

Figure 4.10 Display of DFT matrix for Impulse response h[n]

```
The DFT matrix is given as :
  Columns 1 through 5

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

  Columns 6 through 8

   1.0000 + 0.0000i   1.0000 + 0.0000i   1.0000 + 0.0000i
  -0.7071 + 0.7071i  -0.0000 + 1.0000i   0.7071 + 0.7071i
   0.0000 - 1.0000i  -1.0000 - 0.0000i  -0.0000 + 1.0000i
   0.7071 + 0.7071i   0.0000 - 1.0000i  -0.7071 + 0.7071i
  -1.0000 - 0.0000i   1.0000 + 0.0000i  -1.0000 - 0.0000i
   0.7071 - 0.7071i  -0.0000 + 1.0000i  -0.7071 - 0.7071i
  -0.0000 + 1.0000i  -1.0000 - 0.0000i  -0.0000 - 1.0000i
  -0.7071 - 0.7071i  -0.0000 - 1.0000i   0.7071 - 0.7071i
```

Figure 4.11 Display of DFT matrix for Input sequence x[n]

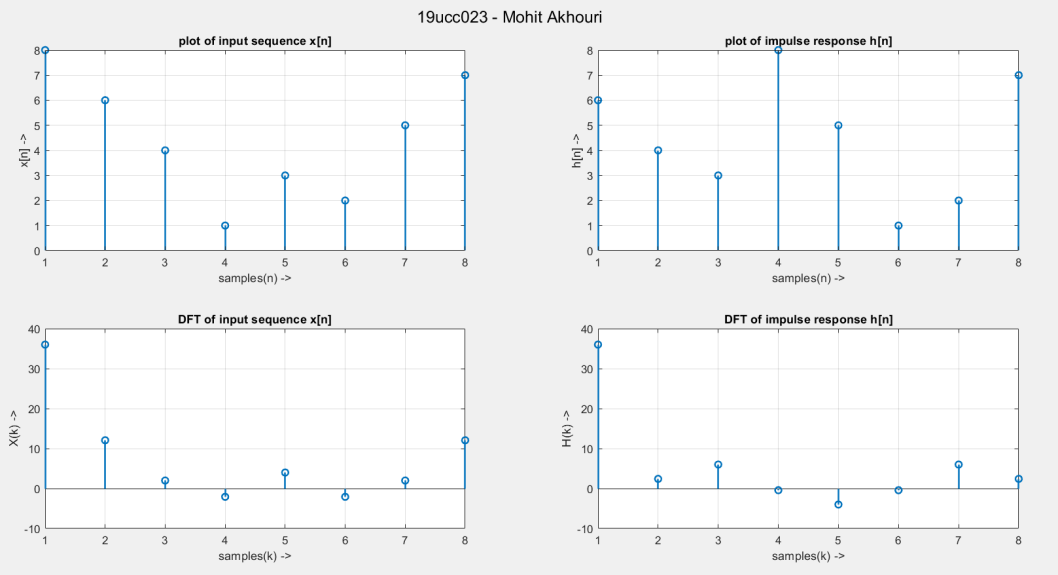


Figure 4.12 Plots of x[n], h[n], X(k) and H(k)

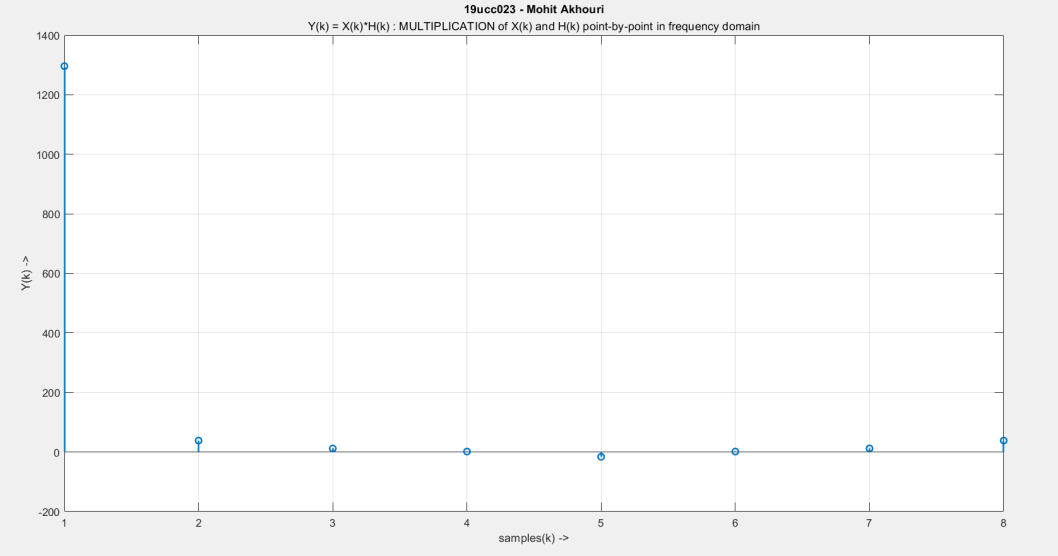


Figure 4.13 Plot of multiplication of X(k) and H(k) in frequency domain

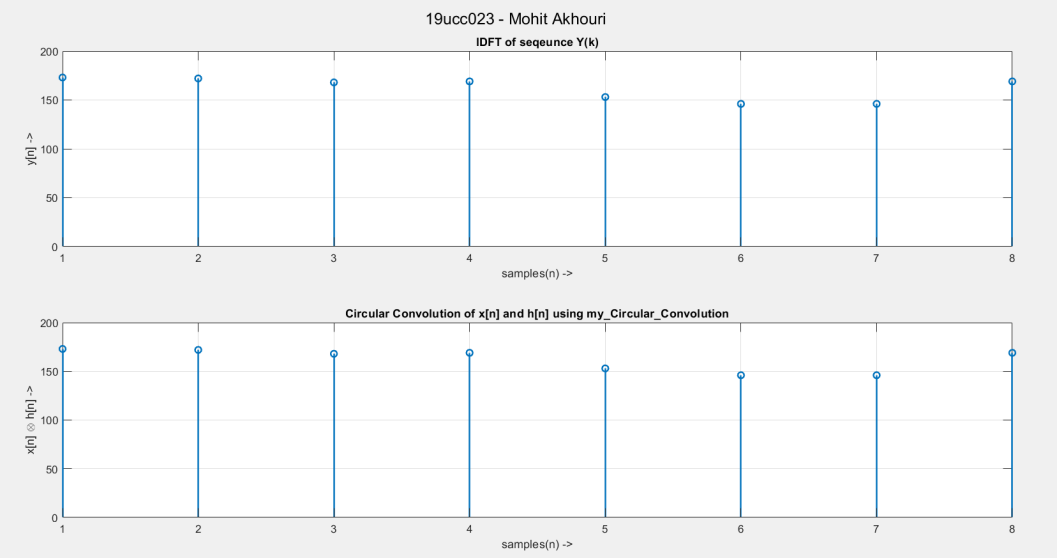


Figure 4.14 Comparison of Plots of IDFT sequence and Circular Convolution (USER-DEFINED)

4.4.2.2 Verify Circular Convolution through MATRIX Multiplication :

```
% 19ucc023
% Mohit Akhouri
% Experiment 4 - Observation 4

% In this code , we will perform matrix multiplication of Convolution
% matrix, DFT matrix and inverse of DFT matrix, next we will compare
 the
% results obtained with circular convolution of x[n] and h[n]

clc;
clear all;
close all;

% ALGORITHM : Initialize 8-point x[n] and h[n]
% First calculate the convolution matrix H and DFT matrix D
% Now calculate 8-point DFT of x[n]
% Then we perform the matrix multiplication and compare the results
% with the circular convolution of x[n] and h[n]

x = randperm(8,8); % input sequence x[n]
h = randperm(8,8); % impulse response h[n]
circ_conv_xh = my_Circular_Convolution(x,h); % Circular convolution of
 x[n] and h[n]

length_x = length(x); % length of input sequence x[n]
length_h = length(h); % length of impulse response h[n]

rows = 8; % rows of matrix ( both convolution matrix and DFT matrix )
columns = 8; % columns of matrix ( both convolution matrix and DFT
 matrix )
n = 8; % parameter for passing to myDFT function

H = myCirConvMat(h,length_x); % Circular convolution matrix H
[Xf , D_mat_x] = myDFT(x,8); % Calculating DFT matrix D_mat_x and DFT
 (Xf) of input sequence x[n]

D_mat_x_inverse = inv(D_mat_x); % Inverse of DFT matrix D_mat_x of
 x[n]

Hf = (D_mat_x * H)*(D_mat_x_inverse); % calculating Hf matrix

% calculating Yf = Hf * Xf
Yf = zeros(1,n); % initializing Yf vector

for i=1:rows
    sum = 0;
    for j=1:columns
        sum = sum + (Hf(i,j)*Xf(j));
    end
    Yf(i) = sum;
end
```

Figure 4.15 Part 1 of the code for observation 4

```
% calculating y = D8(-1) . Yf to be compared with circ_conv_xh
y = zeros(1,n); % initializing vector 'y'
for i=1:rows
    sum = 0;
    for j=1:columns
        sum = sum + (D_mat_x_inverse(i,j)*Yf(j));
    end
    y(i) = sum;
end

% plotting x[n] and h[n]
figure;
subplot(2,1,1);
stem(x,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('x[n] ->');
title('plot of input sequence x[n]');
grid on;
subplot(2,1,2);
stem(h,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('h[n] ->');
title('plot of impulse response h[n]');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');

% plotting y[n] and circ_conv_xh for comparison of Circular
 convolution
figure;
subplot(2,1,1);
stem(y,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('y[n] ->');
title('y[n] = D_{8}^{-1}.Y_{F} : circular convolution obtained via
 MATRIX MULTIPLICATION');
grid on;
subplot(2,1,2);
stem(circ_conv_xh,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('x[n] \otimes h[n] ->');
title('Circular Convolution of x[n] and h[n] using my\_Circular
\_Convolution');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');
```

Figure 4.16 Part 2 of the code for observation 4

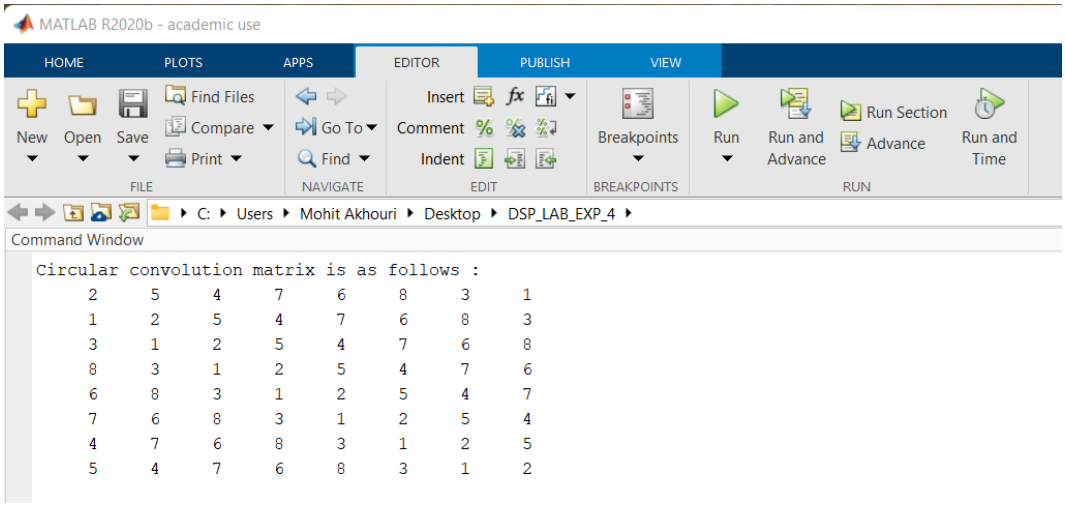


Figure 4.17 Display of Circular Convolution matrix of impulse response h[n]

```
The DFT matrix is given as :
  Columns 1 through 5

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

  Columns 6 through 8

   1.0000 + 0.0000i   1.0000 + 0.0000i   1.0000 + 0.0000i
  -0.7071 + 0.7071i  -0.0000 + 1.0000i   0.7071 + 0.7071i
   0.0000 - 1.0000i  -1.0000 - 0.0000i  -0.0000 + 1.0000i
   0.7071 + 0.7071i   0.0000 - 1.0000i  -0.7071 + 0.7071i
  -1.0000 - 0.0000i   1.0000 + 0.0000i  -1.0000 - 0.0000i
   0.7071 - 0.7071i  -0.0000 + 1.0000i  -0.7071 - 0.7071i
  -0.0000 + 1.0000i  -1.0000 - 0.0000i  -0.0000 - 1.0000i
  -0.7071 - 0.7071i  -0.0000 - 1.0000i   0.7071 - 0.7071i
```

Figure 4.18 Display of DFT matrix of Input sequence x[n]

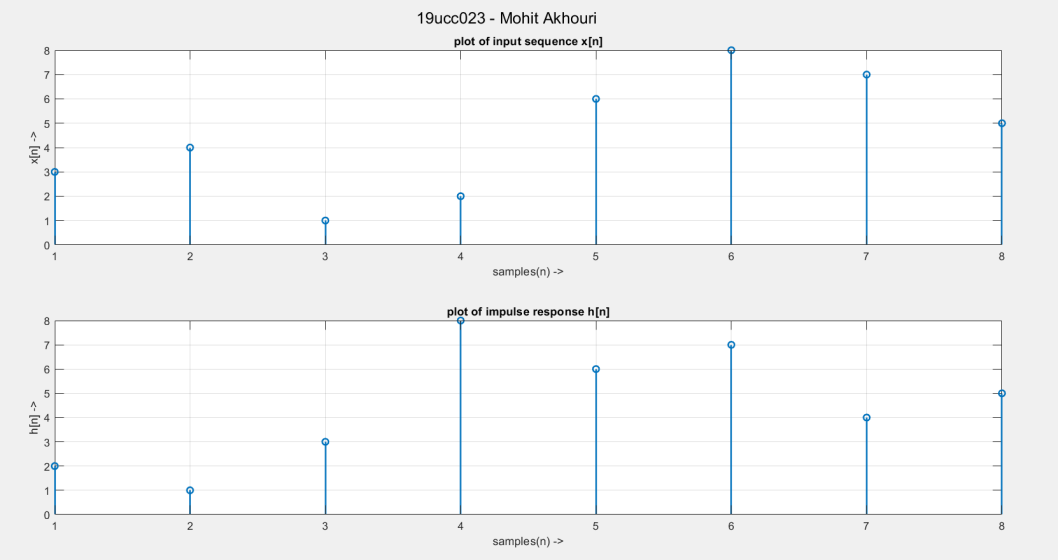


Figure 4.19 Plot of input sequence x[n] and impulse response h[n]

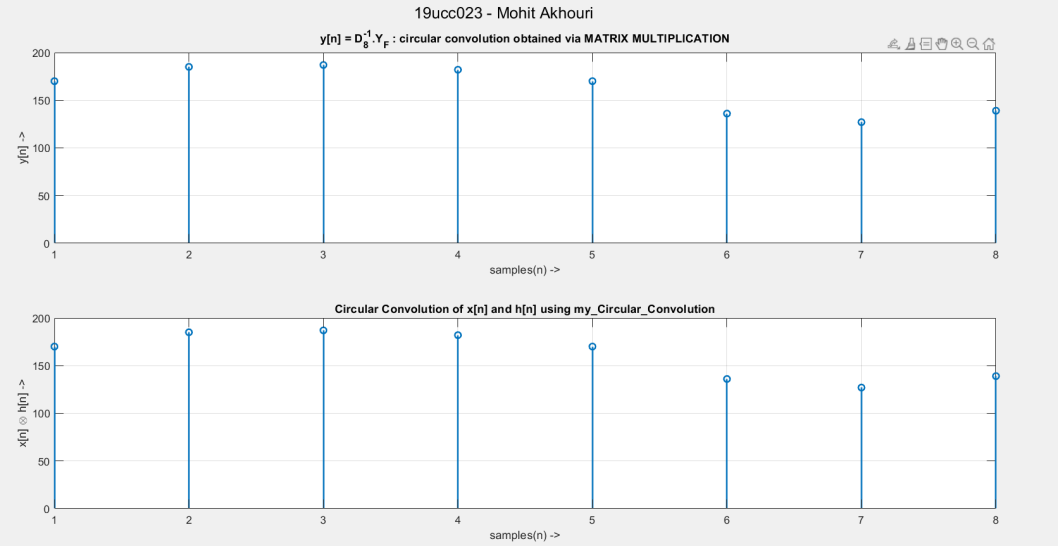


Figure 4.20 Comparison of Plots of MATRIX MULTIPLICATION OUTPUT and circular Convolution obtained (USER-DEFINED)

4.4.3 Circular Convolution in Simulink :

```
% 19ucc023
% Mohit Akhouri
% Experiment 4 - Observation 5

% This code will call the simulink model 'Simulink_Observation_5' for
% calculation of circular convolution between x[n] and h[n]

sim('Simulink_Observation_5'); % calling the simulink model

% plotting the two circular convolutions obtained via Simulink Model

figure;
subplot(2,1,1);
stem(out.circ_conv_xh_1.data,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('x[n] \otimes h[n] ->');
title('Circular Convolution between x[n] and h[n] obtained via
 SIMULINK MODEL through my\_Circular\_Convolution function');
grid on;
subplot(2,1,2);
stem(out.circ_conv_xh_2.data,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('x[n] \otimes h[n] ->');
title('Circular Convolution between x[n] and h[n] obtained via
 SIMULINK MODEL through DFT multiplication');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');
```

Figure 4.21 Code for the observation 5

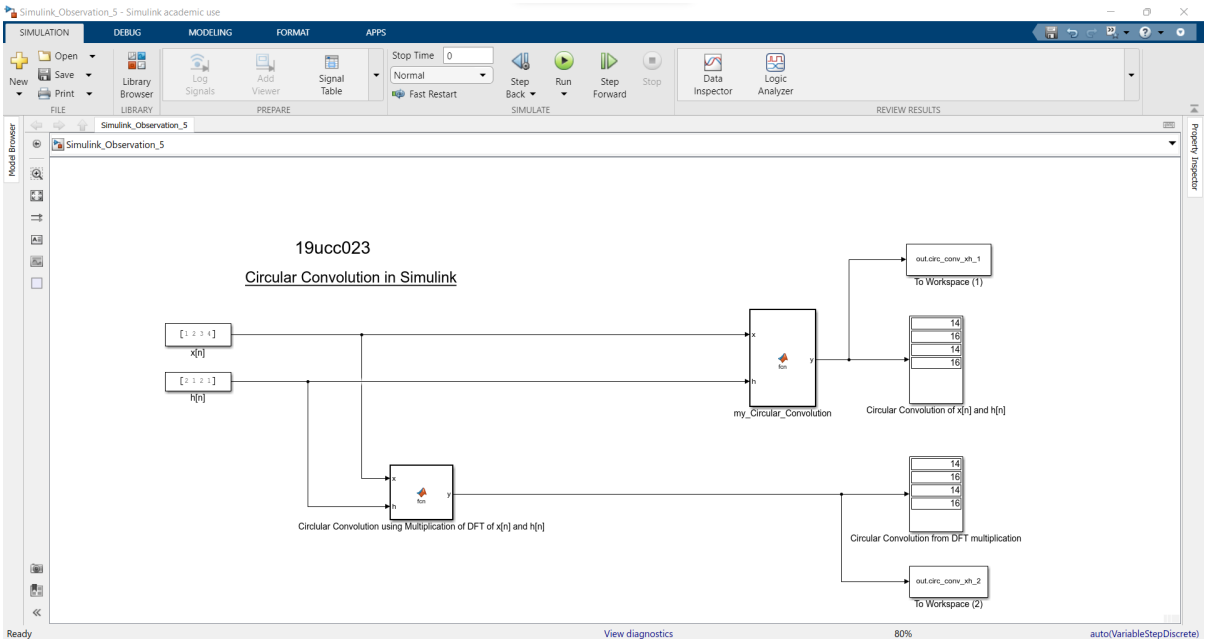


Figure 4.22 Simulink Model used for Circular Convolution

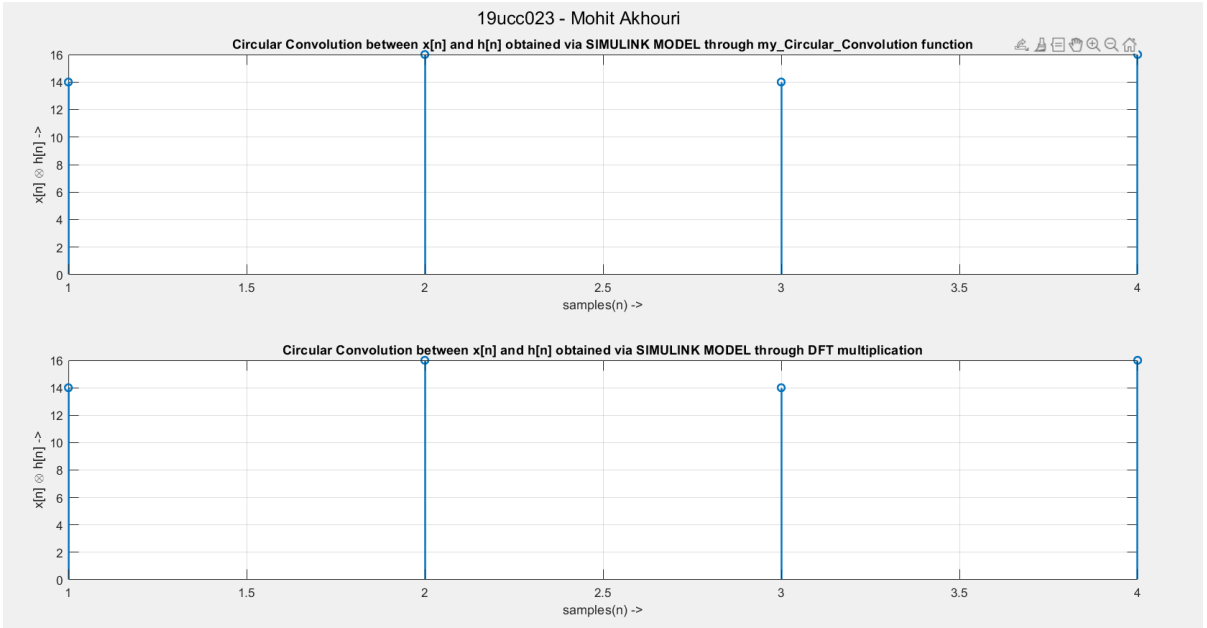


Figure 4.23 Plot of Circular Convolution (through 2 methods) obtained via Simulink Model

4.4.4 Functions used in main codes for Circular Convolution , DFT and IDFT :

4.4.4.1 myCirConvMat.m function code :

```
function [H] = myCirConvMat(h,n)
% This function calculates the circular convolution matrix
% here 'h' is the impulse response and 'n' is the length of the input
% sequence x[n]

% 19ucc023
% Mohit Akhouri

h = flip(h); % flipping the impulse response h[n]
rows = n; % rows of Circular convolution matrix
columns = n; % columns of Circular convolution matrix
H = zeros(rows,columns); % initializing the Circular convolution
 matrix with zeros

length_h = length(h); % length of impulse response h[n]
row_ind = 1; % row index for storing in matrix

% ALGORITHM : for calculation of circular convolution matrix
% Flip the h[n]
% run a loop from i to last index of h[n] and store the values in H
% run a loop from 1 to i-1 ( this is the cyclic part ) of h[n] and
 store
% values in H

for i=length_h:-1:1
    offset = 1; % for column index ( for storage of values in H )

    % loop from 'i' to 'last index' of h[n]
    for j=i:length_h
        H(row_ind,offset) = h(j);
        offset = offset + 1;
    end

    % loop from 1 to 'i-1' of h[n]
    for j=1:i-1
        H(row_ind,offset) = h(j);
        offset = offset + 1;
    end

    row_ind = row_ind + 1; % incrementing row index

end

disp('Circular convolution matrix is as follows :');
disp(H); % displaying the matrix

end
```

Published with MATLAB® R2020b

Figure 4.24 myCirConvMat function to calculate **Circular Convolution matrix** of input given to it

4.4.4.2 my_Circular_Convolution.m function code :

```
function [y] = my_Circular_Convolution(x,h)
% This function will calculate the circular convolution
% of input sequence x[n] and impulse response h[n]

% 19ucc023
% Mohit Akhouri

% ALGORITHM : First make the length of both sequences equal , then
 call
% myCirConvMat function to calculate the circular convolution matrix.
% Now multiply the Matrix with 'column-vector' x[n] to get the final
 result

% Making the length of the two sequences equal by 'PADDING WITH ZEROS'

length_x = length(x); % length of input sequence x[n]
length_h = length(h); % length of impulse response h[n]

if(length_x > length_h)
    h = [h zeros(1,length_x - length_h)]; % padding 'h[n]' with zeros
    length_h = length(h);
else
    x = [x zeros(1,length_h - length_x)]; % padding 'x[n]' with zeros
    length_x = length(x);
end

n = length_x; % length of input sequence x[n] to be passed to function

% Calling the myCirConvMat function to calculate the circular
 convolution
% matrix and storing it in variable H

H = myCirConvMat(h,n); % calling the function for Circular Conv.
 matrix

% Algorithm for calculation of circular convolution is as follows

y = zeros(1,n); % initializing the output vector

for i=1:n
    sum = 0;
    for j=1:n
       sum = sum + (H(i,j)*x(j));
    end
    y(i) = sum;
end

end
```

Published with MATLAB® R2020b

Figure 4.25 my_Circular_Convolution function to calculate the **Circular Convolution** of x[n] and h[n]

4.4.4.3 myDFT.m function code :

```
function [X,D] = myDFT(x,N)
% This function will calculate the discrete fourier transform
% of input sequence x[n] , this function calculates N-point DFT

% 19ucc023
% Mohit Akhouri

% calculating the DFT matrix 'D'
D = zeros(N,N); % DFT matrix to store the values of twiddle factor
twd_factor = 0; % to store the value of twiddle factor

for n=1:N
    for k=1:N
        twd_factor = exp(-1j*2*pi*(k-1)*(n-1)/N);
        D(n,k) = twd_factor;
    end
end

disp('The DFT matrix is given as :');
disp(D);

% The ALGORITHM for calculation of DFT is as follows
X = zeros(1,N);
for i=1:N
    sum = 0;
    for j=1:N
        sum = sum+(D(i,j)*x(j));
    end
    X(i)=sum;
end

end
```

Published with MATLAB® R2020b

Figure 4.26 myDFT function to calculate the **Discrete Fourier Transform** of input given to it

4.4.4.4 myIDFT.m function code :

```
function [x] = myIDFT(X,N)
% This function will calculate the inverse discrete fourier transform
% (IDFT) of N-point DFT sequence X(k)

% 19ucc023
% Mohit Akhouri

x=zeros(1,N); % initializing output

% main loop algorithm to calculate IDFT
for n=1:N
    sum=0;
    for k=1:N
        sum=sum+(X(k).*(exp(1j*2*pi*(k-1)*(n-1)/N)));
    end
    sum=sum/N;
    x(n)=sum;
end

x=abs(x);

end
```

Published with MATLAB® R2020b

Figure 4.27 myIDFT function to calculate the **Inverse Discrete Fourier Transform** of input given to it

4.5 Conclusion

In this experiment , we learnt the concepts of **Circular Convolution** and **change of Basis** of Digital Signal Processing. We learnt about **circular convolution matrix** and how it can be used to computer circular convolution of two sequences. We also learnt about the concept of **Change of basis**. In change of basis , we learnt about the relationship between Circular Convolution and **DFT multiplication**. We learnt that after taking IDFT of product of DFT of two sequences x[n] and h[n] , we get their circular convolution. We also learnt about the relationship between Circular Convolution and **Matrix Multiplication** . In Matrix multiplication , we generated **Circular Convolution matrix** , **DFT matrix** and **inverse of DFT matrix** and through their multiplication , we obtained the circular convolution . We implemented the techniques in MATLAB . We also built model for Circular Convolution in Simulink and compared the results obtained from MATLAB coding.

Chapter *6*

Experiment - 6

6.1 Aim of the Experiment

- Discrete Cosine Transform and it's energy compaction property
- Simulink based image compression

6.2 Software Used

- MATLAB
- Simulink

6.3 Theory

6.3.1 About Image Compression :

Image compression is a type of data compression applied to digital images, to reduce their cost for storage or transmission. Algorithms may take advantage of visual perception and the statistical properties of image data to provide superior results compared with generic data compression methods which are used for other digital data.

6.3.1.1 Lossy and Lossless Image Compression :

Image compression may be **lossy** or **lossless**. **Lossless compression** is preferred for **archival purposes** and often for medical imaging, technical drawings, clip art, or comics. Lossy compression methods, especially when used at **low bit rates**, introduce **compression artifacts**. **Lossy methods** are especially suitable for **natural images** such as photographs in applications where minor (sometimes imperceptible) loss of fidelity is acceptable to achieve a substantial reduction in bit rate. Lossy compression that produces negligible differences may be called visually lossless.

6.3.2 About Discrete Cosine Transform (DCT) :

A **discrete cosine transform (DCT)** expresses a finite sequence of data points in terms of a **sum of cosine functions oscillating at different frequencies**. The DCT, first proposed by **Nasir Ahmed** in 1972, is a widely used transformation technique in **signal processing** and **data compression**. It is used in most digital media, including **digital images** (such as JPEG and HEIF, where small high-frequency components can be discarded), **digital video** (such as MPEG and H.26x), **digital audio** (such as Dolby Digital, MP3 and AAC), **digital television** (such as SDTV, HDTV and VOD), **digital radio** (such as AAC+ and DAB+), and **speech coding** (such as AAC-LD, Siren and Opus). DCTs are also important to numerous other applications in science and engineering, such as spectral methods for the numerical solution of **partial differential equations**.

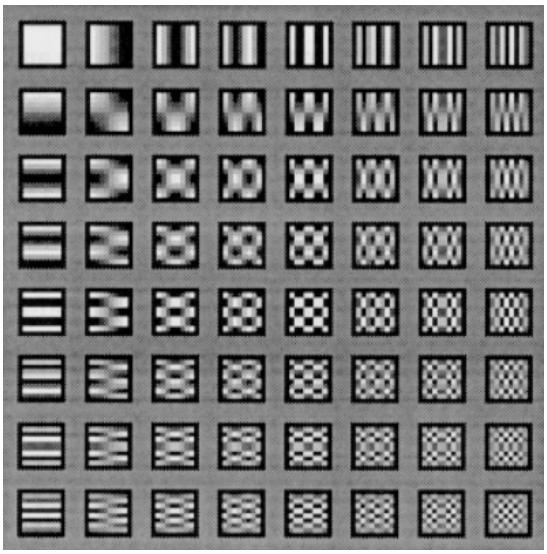


Figure 6.1 plot of the 64 (8 x 8) DCT basis functions

The **2D-Discrete Cosine Transform** (ImF) of a image **Im** of size NxN is given as :

$$ImF(k,l) = \frac{1}{\sqrt{2N}}\beta(k)\beta(l)\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} Im(i,j)cos\left(\frac{\pi(2j+1)k}{2N}\right)cos\left(\frac{\pi(2i+1)l}{2N}\right) \tag{6.1}$$

In the above equation , **Beta function** (β) is defined as :

$$\beta(u) = \frac{1}{\sqrt{2}} \qquad u = 0$$

$$\beta(u) = 1 \qquad u > 0$$

DCT compression, also known as **block compression**, compresses data in sets of discrete DCT blocks. DCT blocks can have a number of sizes, including 8x8 pixels for the standard DCT, and varied integer DCT sizes between 4x4 and 32x32 pixels. The DCT has a strong **energy compaction property**, capable of achieving high quality at high data compression ratios. However, **blocky compression artifacts** can appear when heavy DCT compression is applied.

6.4 Code and results

6.4.1 Using Inbuilt DCT and IDCT to compress and reconstruct any input image :

```
% 19ucc023
% Mohit Akhouri
% Experiment 6 - Observation 1

% This code will apply inbuilt functions "dct2" and "idct2" on the
 image
% "cameraman.tif" and observe the compressed image
% Lastly , after knocking off half the pixels , we will observe the
% artifact effects in the image

clc;
clear all;
close all;

img = imread('cameraman.tif'); % Reading of image file in variable
 'img'

figure;
subplot(1,2,1);
imshow(img);
title('Original Image - cameraman.tif'); % Plot of Original Image

dct_img = dct2(img); % computing Discrete Cosine Transform ( DCT ) of
 img

subplot(1,2,2);
imshow(dct_img);
title('DCT of image - cameraman.tif'); % Plot of DCT of camerman.tif
sgtitle('19ucc023 - Mohit Akhouri');

% Compression Algorithm for the image starts from here
% ALGORITHM : knocking off half the pixels in the compressed image
for i=1:256
    for j=1:256
        if j<=128
            dct_img(i,j)=dct_img(i,j); % keeping half the pixels
        else
            dct_img(i,j)=0; % knocking off other half of pixels
        end
    end
end

figure;
subplot(1,2,1);
imshow(dct_img); % plot of DCT of image after knocking off half pixels
title('DCT of image - cameraman.tif after KNOCKING OFF half pixels');

idct_img = uint8(idct2(dct_img)); % computing IDCT of compressed image
 (dct_img) after knocking off half pixels

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

Figure 6.2 Part 1 of the code for observation 1

```
imshow(idct_img); % plot of IDCT of compressed image demonstrating
 ARTIFACT EFFECTS
title('IDCT of compressed image - cameraman.tif after KNOCKING OFF
 half pixels (ARTIFACT EFFECTS)');
sgtitle('19ucc023 - Mohit Akhouri');
```

Figure 6.3 Part 2 of the code for observation 1

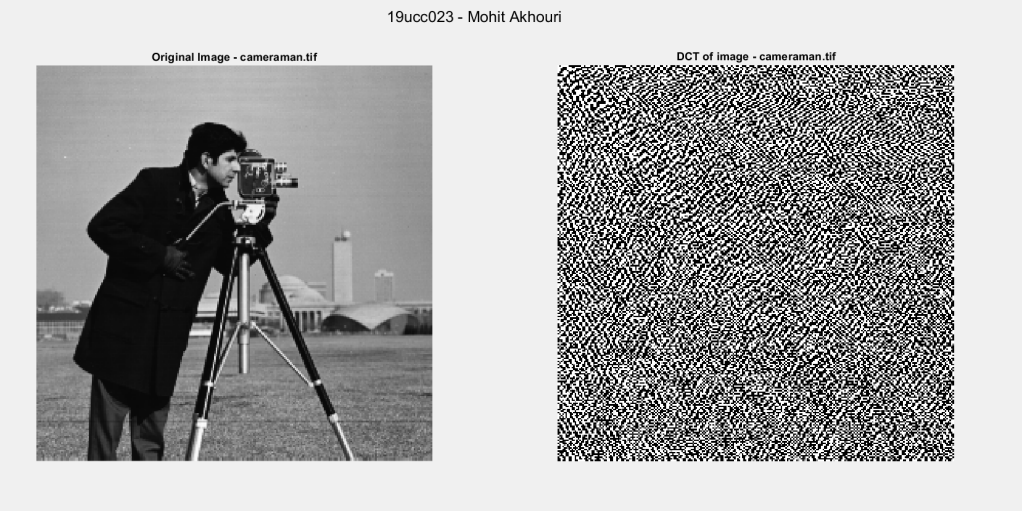


Figure 6.4 Plot of Original Image and DCT of the Image

Figure 6.5 Plot of DCT after **knocking half pixels** and reconstructed Image using IDCT

6.4.2 Using User-Defined function myCompression.m to verify the results of Observation 1 :

```
% 19ucc023
% Mohit Akhouri
% Experiment 6 - Observation 2

% This code will use the function "myCompression.m" function to
 calculate
% the DCT of image "cameraman.tif" and compare the results with
 inbuilt
% function dct2 and idct2

clc;
clear all;
close all;

img = imread('cameraman.tif'); % Reading of image file in variable
 'img'

dct_inbuilt = dct2(img); % INBUILT DCT of image - cameraman.tif
dct_myCompression = myCompression(img); % USER-DEFINED DCT of image -
 cameraman.tif

figure;
imshow(img); % Plot of image - cameraman.tif
title('Original Image - cameraman.tif');
sgtitle('19ucc023 - Mohit Akhouri');

figure;
subplot(1,2,1);
imshow(dct_inbuilt); % Plot of INBUILT DCT of image - cameraman.tif
title('DCT of image - cameraman.tif from INBUILT FUNCTION dct2(im)');

subplot(1,2,2);
imshow(dct_myCompression); % Plot of DCT calculated from USER-DEFINED
 Function 'myCompression'
title('DCT of image - cameraman.tif from USER DEFINED FUNCTION
 myCompression.m');
sgtitle('19ucc023 - Mohit Akhouri');
```

Published with MATLAB® R2020b

Figure 6.6 Code for the observation 2

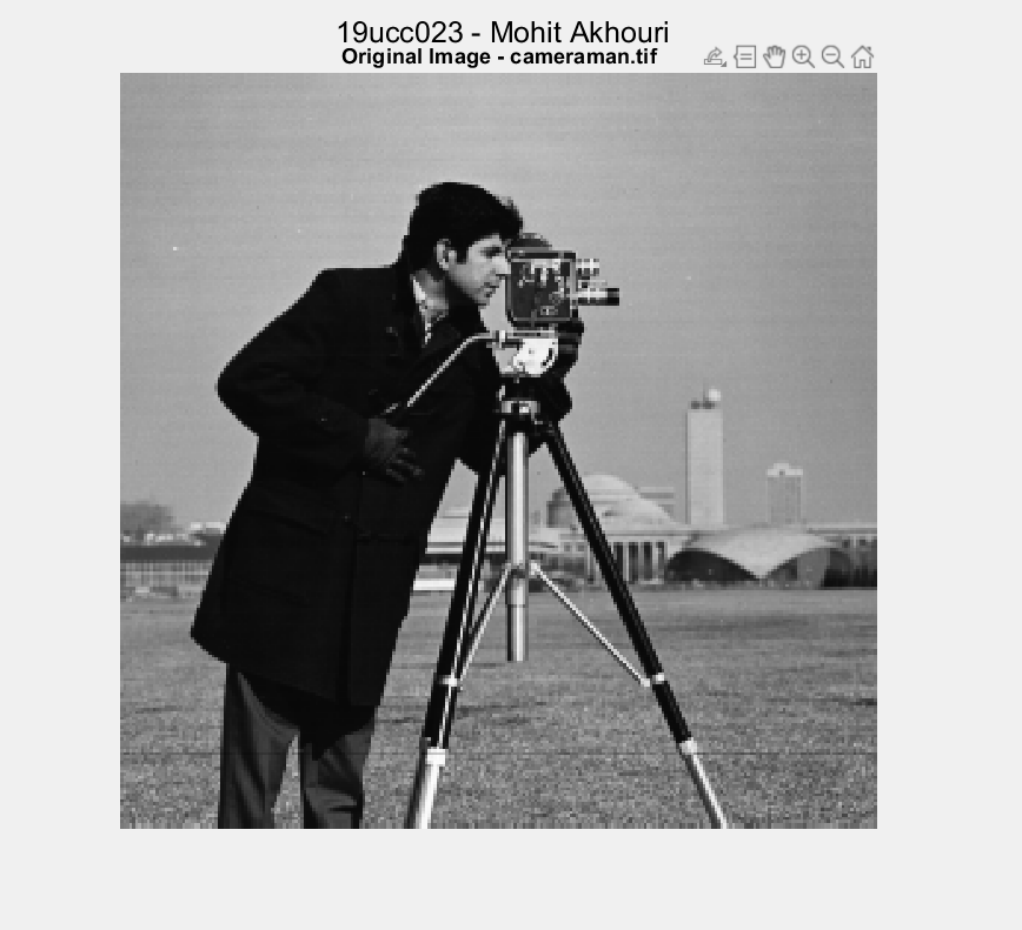


Figure 6.7 Plot of Original Image cameraman.tif

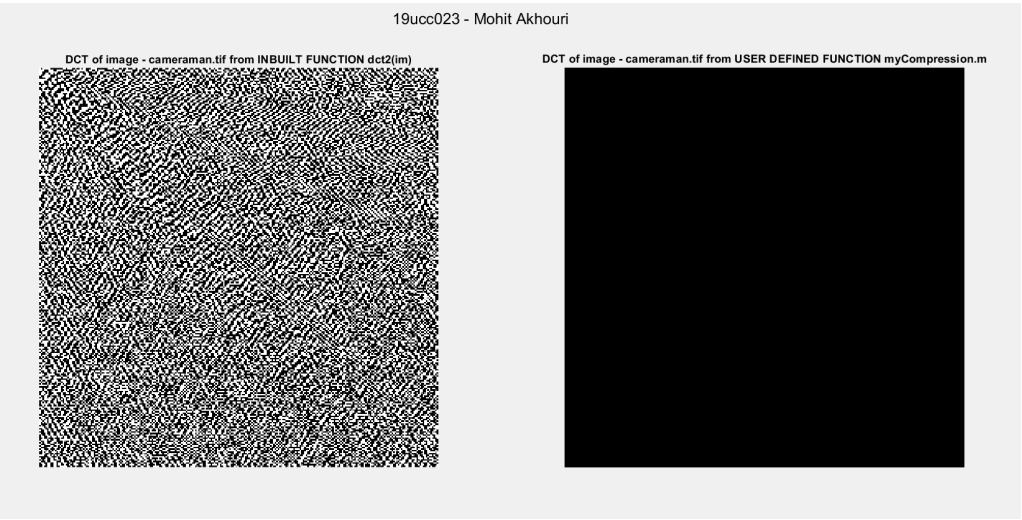


Figure 6.8 Plots of DCT obtained via inbuilt and user-defined functions

6.4.3 Observing Artfifact effects for different values of compression ratio (ρ) :

```
% 19ucc023
% Mohit Akhouri
% Experiment 6 - Observation 3 and Observation 4

% ALGORITHM :
% This code will apply the compression algorithm for 4 different
 cases :
% top 8 coefficients out of 64 , top 16 coefficients out of 64 ,
% top 32 coefficients out of 64 , top 48 coefficients out of 64
% and observe the artifact effects

% This code will also plot the graph between the mean square error and
% compression ratio

clc;
clear all;
close all;

img = imread('cameraman.tif'); % Reading of image file in variable
 'img'
N = size(img,1); % storing the size of the image (256x256) in variable
 'N'

% CASE 1 : Keeping top 48 coefficients out of 64

recon_img = zeros(N,N); % to store the reconstructed image

for i=1:8:N
    for j=1:8:N
        block_8x8 = img(i:i+7,j:j+7); % selecting a 8x8 block
        block_dct = dct2(block_8x8); % Finding the DCT of the selected
 block
        block_dct(7:8,1:8)=zeros(2,8); % knocking off the remaining 16
 coefficients
        recon_img(i:i+7,j:j+7)=idct2(block_dct); % taking IDCT of the
 remaining coefficients
    end
end

% Plot of original image and reconstructed image
figure;
subplot(1,2,1);
imshow(img);
title('Original Image - cameraman.tif');
subplot(1,2,2);
imshow(uint8(recon_img));
title('Reconstructed Image after keeping TOP 48 coefficients out of
 64');
sgtitle('19ucc023 - Mohit Akhouri');

% calculating compression ratio and mean square error
```

Figure 6.9 Part 1 of the code for observation 3 and 4

```
mse_case_1 = mse(img,uint8(recon_img)); % mean square error
 calculation
knocked_off_coeff_1 = 1024*(64-48); % calculating and storing the
 knocked off coefficients
p_case_1 = ((N*N - knocked_off_coeff_1)/(N*N)); % calculating
 compression ratio


% CASE 2 : Keeping top 32 coefficients out of 64

recon_img = zeros(N,N); % to store the reconstructed image

for i=1:8:N
    for j=1:8:N
        block_8x8 = img(i:i+7,j:j+7); % selecting a 8x8 block
        block_dct = dct2(block_8x8); % Finding the DCT of the selected
 block
        block_dct(5:8,1:8)=zeros(4,8); % knocking off the remaining 32
 coefficients
        recon_img(i:i+7,j:j+7)=idct2(block_dct); % taking IDCT of the
 remaining coefficients
    end
end

% Plot of original image and reconstructed image
figure;
subplot(1,2,1);
imshow(img);
title('Original Image - cameraman.tif');
subplot(1,2,2);
imshow(uint8(recon_img));
title('Reconstructed Image after keeping TOP 32 coefficients out of
 64');
sgtitle('19ucc023 - Mohit Akhouri');

% calculating compression ratio and mean square error
mse_case_2 = mse(img,uint8(recon_img)); % mean square error
 calculation
knocked_off_coeff_2 = 1024*(64-32); % calculating and storing the
 knocked off coefficients
p_case_2 = ((N*N - knocked_off_coeff_2)/(N*N)); % calculating
 compression ratio


% CASE 3 : Keeping top 16 coefficients out of 64

recon_img = zeros(N,N); % to store the reconstructed image

for i=1:8:N
    for j=1:8:N
        block_8x8 = img(i:i+7,j:j+7); % selecting a 8x8 block
        block_dct = dct2(block_8x8); % Finding the DCT of the selected
 block
```

Figure 6.10 Part 2 of the code for observation 3 and 4

```
        block_dct(3:8,1:8)=zeros(6,8); % knocking off the remaining 32
 coefficients
        recon_img(i:i+7,j:j+7)=idct2(block_dct); % taking IDCT of the
 remaining coefficients
    end
end

% Plot of original image and reconstructed image
figure;
subplot(1,2,1);
imshow(img);
title('Original Image - cameraman.tif');
subplot(1,2,2);
imshow(uint8(recon_img));
title('Reconstructed Image after keeping TOP 16 coefficients out of
 64');
sgtitle('19ucc023 - Mohit Akhouri');

% calculating compression ratio and mean square error
mse_case_3 = mse(img,uint8(recon_img)); % mean square error
 calculation
knocked_off_coeff_3 = 1024*(64-16); % calculating and storing the
 knocked off coefficients
p_case_3 = ((N*N - knocked_off_coeff_3)/(N*N)); % calculating
 compression ratio


% CASE 4 : Keeping top 8 coefficients out of 64

recon_img = zeros(N,N);

for i=1:8:N
    for j=1:8:N
        block_8x8 = img(i:i+7,j:j+7); % selecting a 8x8 block
        block_dct = dct2(block_8x8);  % Finding the DCT of the
 selected block
        block_dct(2:8,1:8)=zeros(7,8); % knocking off the remaining 56
 coefficients
        recon_img(i:i+7,j:j+7)=idct2(block_dct); % taking IDCT of the
 remaining coefficients
    end
end

% Plot of original image and reconstructed image
figure;
subplot(1,2,1);
imshow(img);
title('Original Image - cameraman.tif');
subplot(1,2,2);
imshow(uint8(recon_img));
title('Reconstructed Image after keeping TOP 8 coefficients out of
 64');
sgtitle('19ucc023 - Mohit Akhouri');
```

Figure 6.11 Part 3 of the code for observation 3 and 4

```
% calculating compression ratio and mean square error
mse_case_4 = mse(img,uint8(recon_img)); % mean square error
 calculation
knocked_off_coeff_4 = 1024*(64-8); % calculating and storing the
 knocked off coefficients
p_case_4 = ((N*N - knocked_off_coeff_4)/(N*N)); % calculating
 compression ratio

% Plot of Mean Square error vs. Compression Ratio

mse_array = zeros(1,4); % For storing different values of MSE
p_array = zeros(1,4); % For storing different values of compression
 ratio (p)

% storage of different values of MSE
mse_array(1) = mse_case_1;
mse_array(2) = mse_case_2;
mse_array(3) = mse_case_3;
mse_array(4) = mse_case_4;

% storage of different values of compression ratio (p)
p_array(1) = p_case_1;
p_array(2) = p_case_2;
p_array(3) = p_case_3;
p_array(4) = p_case_4;

% Plot of MSE vs. Compression ratio (p)
figure;
stem(p_array,mse_array,'Linewidth',1.5);
xlabel('Compression ratio (\rho) ->');
ylabel('Mean Square Error (\epsilon) ->');
title('19ucc023 - Mohit Akhouri','Plot of Mean Square Error (\epsilon)
 vs. Compression ratio (\rho)');
grid on;
```

Figure 6.12 Part 4 of the code for observation 3 and 4

Figure 6.13 Plot of Original Image and **Compressed Image** obtained by keeping **top 48 / 64 coefficients**

Figure 6.14 Plot of Original Image and **Compressed Image** obtained by keeping **top 32 / 64 coefficients**

Figure 6.15 Plot of Original Image and **Compressed Image** obtained by keeping **top 16 / 64 coefficients**

Figure 6.16 Plot of Original Image and **Compressed Image** obtained by keeping **top 8 / 64 coefficients**

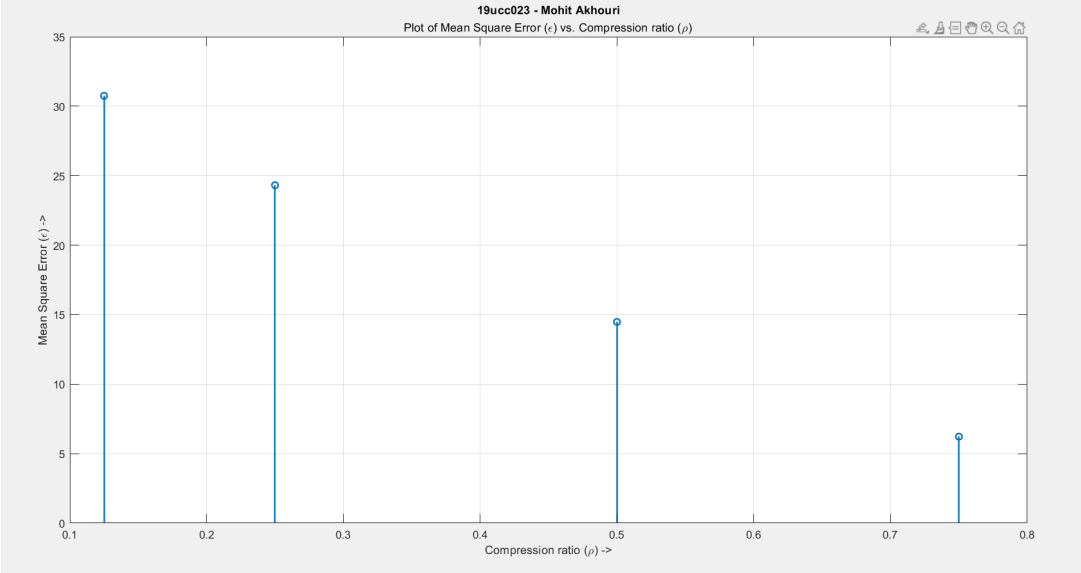


Figure 6.17 Graph of Mean Square Error (ϵ) vs. Compression Ratio (ρ)

6.4.4 Simulink based Image Compression :

```
% 19ucc023
% Mohit Akhouri
% Experiment 6 - Observation 5

% ALGORITHM :
% This code will use the model "Simulink_Observation_5" and calculate
 the
% DCT of the received image signal for four different cases by
 knocking off
% 16,32,48 and 56 coeffcients

sim('Simulink_Observation_5'); % calling the simulink model

N = 256; % Total size of image ( 256 rows with 256 columns )
output_case_1 = zeros(N,N); % Output for case 1 = top 48/64
 coefficients
output_case_2 = zeros(N,N); % Output for case 2 = top 32/64
 coefficients
output_case_3 = zeros(N,N); % Output for case 3 = top 16/64
 coefficients
output_case_4 = zeros(N,N); % Output for case 4 = top 8/64
 coefficients

[output_case_1,output_case_2,output_case_3,output_case_4] =
 DCT_2D(out.y1.data,out.y2.data,out.y3.data,out.y4.data);

% Plot of Figures for case 1 and case 2
figure;
subplot(1,2,1);
imshow(uint8(output_case_1));
title('Compressed Image obtained via SIMULINK MODEL for Case 1 - top
 48/64 coefficients kept');
subplot(1,2,2);
imshow(uint8(output_case_2));
title('Compressed Image obtained via SIMULINK MODEL for Case 2 - top
 32/64 coefficients kept');
sgtitle('19ucc023 - Mohit Akhouri');

% Plot of Figures for case 3 and case 4
figure;
subplot(1,2,1);
imshow(uint8(output_case_3));
title('Compressed Image obtained via SIMULINK MODEL for Case 3 - top
 16/64 coefficients kept');
subplot(1,2,2);
imshow(uint8(output_case_4));
title('Compressed Image obtained via SIMULINK MODEL for Case 4 - top
 8/64 coefficients kept');
sgtitle('19ucc023 - Mohit Akhouri');
```

Figure 6.18 Code for the observation 5

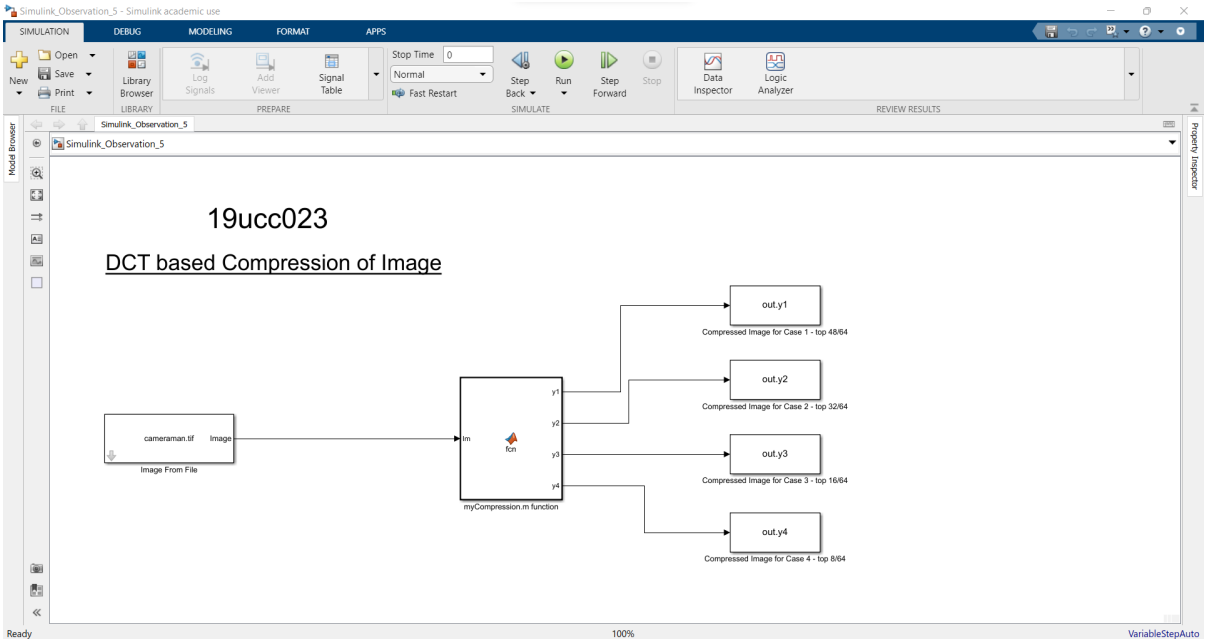


Figure 6.19 Simulink Model used for Image Compression

Figure 6.20 Plots of **Compressed Images** obtained via Simulink Model (for Case 1 and Case 2)

Figure 6.21 Plots of **Compressed Images** obtained via Simulink Model (for Case 3 and Case 4)

6.4.5 Functions used in main codes for DCT and Image Compression :

6.4.5.1 myCompression.m function code :

```
function [ImF] = myCompression(Im)

% 19ucc023
% Mohit Akhouri

% ALGORITHM :
% This function will compute the DCT of given image signal "Im"
% It uses the expression of "sum of cosines" to calculate the DCT

N = size(Im,1); % Size of the image (256x256)

ImF = zeros(N,N); % to store the final computed DCT

Beta = zeros(1,N); % variable that takes value 1/root(2) if i=1 else
 takes value 1

% algorithm for calculation of different values of variable "beta" is
 given
% belo , here a loop is run and correspondingly the value is set.
for i=1:N
    if i==1
        Beta(i)=1/sqrt(2);
    else
        Beta(i)=1.0;
    end
end

% Main ALGORITHM for the calculation of DCT is as follows :
for k=1:N
    for l=1:N
        sum = 0.0;
        for i=1:N
            for j=1:N
                sum = sum + (Im(i,j)*cos((pi*((2.0*(j-1)+1))*(k-1))/
(2.0*N))*cos((pi*((2.0*(i-1))+1)*(l-1))/(2.0*N)));
            end
        end
        sum = sum * (1.0/sqrt(2.0*N)) * Beta(k) * Beta(l) ;
    end
end

end
```

Published with MATLAB® R2020b

Figure 6.22 myCompression.m function used to calculate the **DCT** of the given input image

6.4.5.2 DCT_2D.m function code :

```
function [y1,y2,y3,y4] = DCT_2D(Im1,Im2,Im3,Im4)

% 19ucc023
% Mohit Akhouri

% ALGORITHM : This function will calculate the DCT of image img for
 various
% cases : knocking off 16,32,48 and 56 coefficients ( higher frequency
% terms )


N = size(Im1,1); % size of image (256x256)
img = Im1; % temporary variable to store the image

y1 = zeros(N,N); % output variable for case 1
y2 = zeros(N,N); % output variable for case 2
y3 = zeros(N,N); % output variable for case 3
y4 = zeros(N,N); % output variable for case 4

% ALGORITHM for compression is as follows :

recon_img = zeros(N,N); % to store the reconstructed image

for i=1:8:N
    for j=1:8:N
        block_8x8 = img(i:i+7,j:j+7); % selecting a 8x8 block
        block_dct = dct2(block_8x8); % Finding the DCT of the selected
 block
        block_dct(7:8,1:8)=zeros(2,8); % knocking off the remaining 16
 coefficients
        recon_img(i:i+7,j:j+7)=idct2(block_dct); % taking IDCT of the
 remaining coefficients
    end
end

y1=recon_img;

% ALGORITHM for compression is as follows :

recon_img = zeros(N,N); % to store the reconstructed image

for i=1:8:N
    for j=1:8:N
        block_8x8 = img(i:i+7,j:j+7); % selecting a 8x8 block
        block_dct = dct2(block_8x8); % Finding the DCT of the selected
 block
        block_dct(5:8,1:8)=zeros(4,8); % knocking off the remaining 32
 coefficients
        recon_img(i:i+7,j:j+7)=idct2(block_dct); % taking IDCT of the
 remaining coefficients
    end
end
```

Figure 6.23 Part 1 of the code for the function DCT_2D.m used for **Image Compression**

```
y2 = recon_img;

% ALGORITHM for compression is as follows :

recon_img = zeros(N,N); % to store the reconstructed image

for i=1:8:N
    for j=1:8:N
        block_8x8 = img(i:i+7,j:j+7); % selecting a 8x8 block
        block_dct = dct2(block_8x8); % Finding the DCT of the selected
 block
        block_dct(3:8,1:8)=zeros(6,8); % knocking off the remaining 32
 coefficients
        recon_img(i:i+7,j:j+7)=idct2(block_dct); % taking IDCT of the
 remaining coefficients
    end
end

y3 = recon_img;

% ALGORITHM for compression is as follows :

recon_img = zeros(N,N);

for i=1:8:N
    for j=1:8:N
        block_8x8 = img(i:i+7,j:j+7); % selecting a 8x8 block
        block_dct = dct2(block_8x8);  % Finding the DCT of the
 selected block
        block_dct(2:8,1:8)=zeros(7,8); % knocking off the remaining 56
 coefficients
        recon_img(i:i+7,j:j+7)=idct2(block_dct); % taking IDCT of the
 remaining coefficients
    end
end

y4 = recon_img;

end
```

Published with MATLAB® R2020b

Figure 6.24 Part 2 of the code for the function DCT_2D.m used for **Image Compression**

6.5 Conclusion

In this experiment , we learnt the concepts of **Discrete Cosine Transform** , **Inverse DCT** and **Transform Based Lossy Image Compression** of Digital Signal Processing. We learnt about DCT matrix and how to compute the DCT of any given image signal. We learnt the significance of DCT in **Image Compression**. We also observed the compression of image for various cases - by keeping top 48,32,16 and 8 coeffcients out of 64 and rejecting the rest. We observed the **Artifact effects** due to lossy compression. We also learnt about the relation between **Compression Ratio** (ρ) and **Mean square error** (ϵ). We also plotted the graph of Mean square error vs. Compression ratio for different values of Compression ratio. We also implemented the Image Compression in Simulink and compared the results obtained from MATLAB coding.

Chapter *7*

Experiment - 7

7.1 Aim of the Experiment

- Discrete Cosine Transform and it's energy compaction property
- Simulink based Audio compression

7.2 Software Used

- MATLAB
- Simulink

7.3 Theory

7.3.1 About Audio Compression :

Audio data compression has the potential to reduce the transmission bandwidth and storage requirements of audio data. Audio compression algorithms are implemented in software as **audio codecs**. In both lossy and lossless compression, information redundancy is reduced, using methods such as coding, quantization, **discrete cosine transform** and **linear prediction** to reduce the amount of information used to represent the uncompressed data. Lossy audio compression algorithms provide higher compression and are used in numerous audio applications including **Vorbis** and **MP3**. These algorithms almost all rely on **psychoacoustics** to eliminate or **reduce fidelity** of less audible sounds, thereby reducing the space required to store or transmit them.

Lossless audio compression produces a representation of digital data that can be decoded to an exact digital duplicate of the original. Compression ratios are around 50–60 % of the original size, which is similar to those for generic lossless data compression. **Lossless codecs** use curve fitting or linear prediction as a basis for estimating the signal. Parameters describing the estimation and the difference between the estimation and the actual signal are coded separately.

7.3.2 About Discrete Cosine Transform (DCT) :

A **discrete cosine transform (DCT)** expresses a finite sequence of data points in terms of a **sum of cosine functions oscillating at different frequencies**. The DCT, first proposed by **Nasir Ahmed** in 1972, is a widely used transformation technique in **signal processing** and **data compression**. It is used in most digital media, including **digital audio** (such as Dolby Digital, MP3 and AAC), **digital radio** (such as AAC+ and DAB+), and **speech coding** (such as AAC-LD, Siren and Opus). DCTs are also important to numerous other applications in science and engineering like **Partial Differential Equations**.

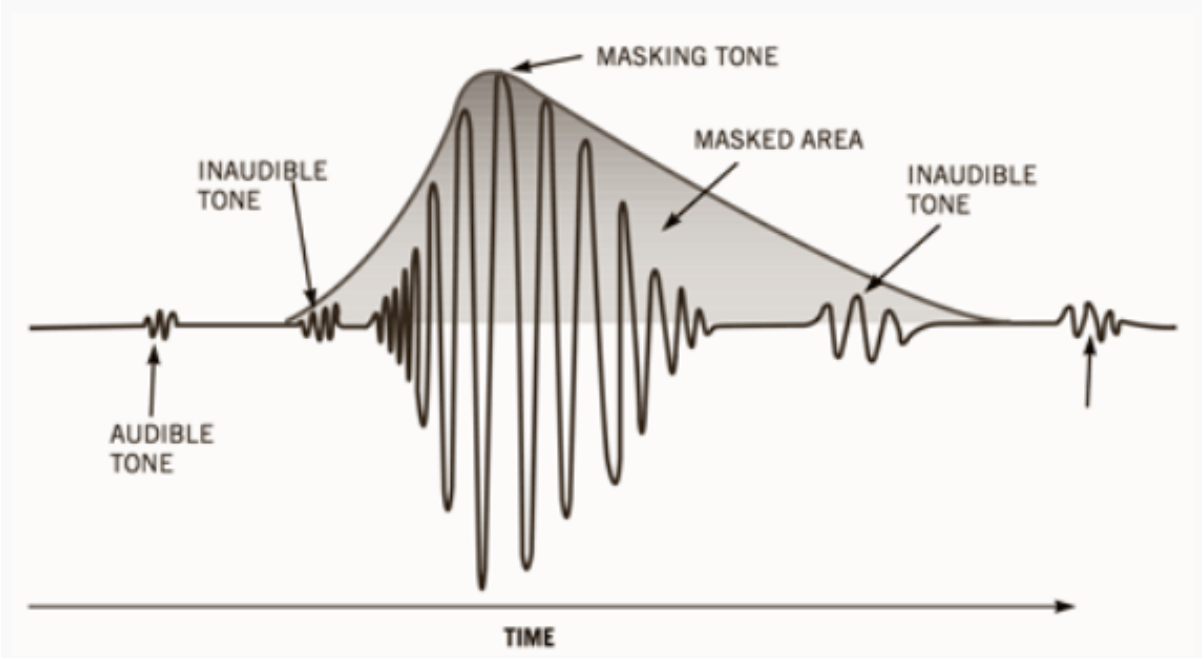


Figure 7.1 Lossy Compression of mp3 audio signal

7.3.2.1 Expression used for calculation of DCT of audio file :

Let **x** be the audio file of size 1 x N. The corresponding 1D-Discrete Cosine Transform **y** is given as:

$$y(k) = w(k) \sum_{n=1}^{N} x(n) cos\left(\frac{\pi(2n-1)(k-1)}{2N}\right) \tag{7.1}$$

In the above equation , **w** function is defined as :

$$w(k) = \frac{1}{\sqrt{N}} \qquad k = 1$$

$$w(k) = \sqrt{\frac{2}{N}} \qquad k > 1$$

7.3.2.2 Expression used for calculation of 1D-IDCT for reconstruction of audio signal :

The **1D-IDCT** expression is given as :

$$x_{idc}(n) = \sum_{k=1}^{N} w(k)y(k)cos\left(\frac{\pi(2n-1)(k-1)}{2N}\right) \tag{7.2}$$

In the above equation , **w** function is defined as :

$$w(k) = \frac{1}{\sqrt{N}} \qquad k = 1$$

$$w(k) = \sqrt{\frac{2}{N}} \qquad k > 1$$

7.3.3 How Compression of Audio helps :

Compression can be used to subtly **massage** a track to make it more **natural sounding** and **intelligible** without adding distortion, resulting in a song that's more "comfortable" to listen to. Additionally, many compressors — both hardware and software — will have a signature sound that can be used to inject wonderful coloration and tone into otherwise lifeless tracks.

Alternately, over-compressing your music can really squeeze the life out of it. Having a good grasp of the basics will go a long way toward understanding how compression works, and confidently using it to your advantage.

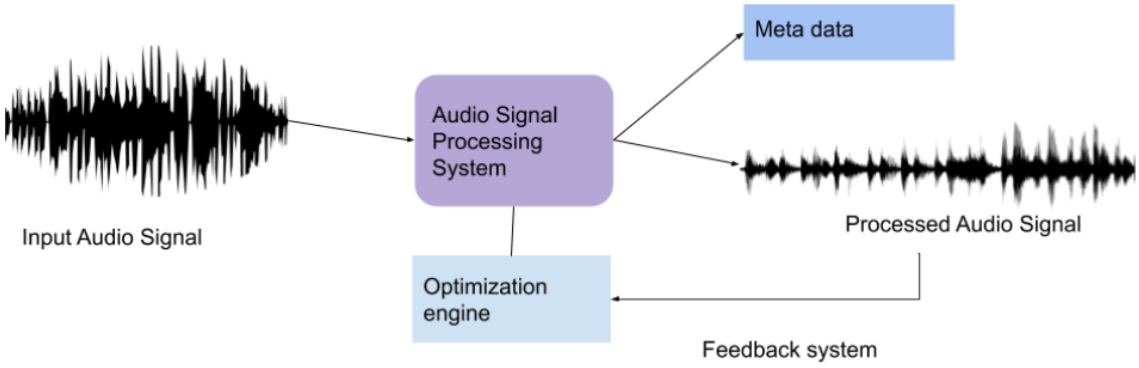


Figure 7.2 Block Diagram of Audio Processing

7.4 Code and results

7.4.1 Using Inbuilt DCT and IDCT to compress and reconstruct any input audio signal :

```
% 19ucc023
% Mohit Akhouri
% Experiment 7 - Observation 1

% In this code , we take the input of an audio signal
% We calculate the 1D-DCT ( via inbuilt function dct ) to obtain the
% compressed audio wave
% Later on , we calculate the inbuilt IDCT of the compressed wave to
 get
% the reconstructed approx. Original Sound signal

clc;
clear all;
close all;

[x,fs] = audioread('input.wav'); % Reading of audio file 'input.wav'

% Plot of Original Sound wave 'input.wav'
figure;
plot(x);
xlabel('time(t) ->');
ylabel('x(t) ->');
title('19ucc023 - Mohit Akhouri','Plot of Original Sound wave
 input.wav');
grid on;

dct_audio = dct(x); % Calculation of DCT of input.wav via INBUILT DCT
 - Compression of audio file

% Plot of INBUILT DCT of input.wav
figure;
plot(dct_audio);
xlabel('frequency (Hz) ->');
ylabel('x_{DCT}(f) ->');
title('19ucc023 - Mohit Akhouri','Plot of DCT of Sound Wave input.wav
 obtained via INBUILT FUNCTION');
grid on;

audiowrite('Obs1_DCT.wav',dct_audio,fs); % Writing dct_audio to audio
 file

idct_audio = idct(dct_audio); % Reconstructed Approx. Original audio
 wave via INBUILT IDCT

% Plot of Reconstructed Audio wave
figure;
plot(idct_audio);
xlabel('time(t) ->');
ylabel('x_{IDCT}(t) ->');
title('19ucc023 - Mohit Akhouri','Plot of Reconstructed Original Sound
 wave obtained via INBUILT IDCT');
grid on;
```

Figure 7.3 Part 1 of the code for observation 1

```
audiowrite('Obs1_IDCT.wav',idct_audio,fs); % Writing Reconstructed
 audio wave to a audio file
```

Figure 7.4 Part 2 of the code for observation 1

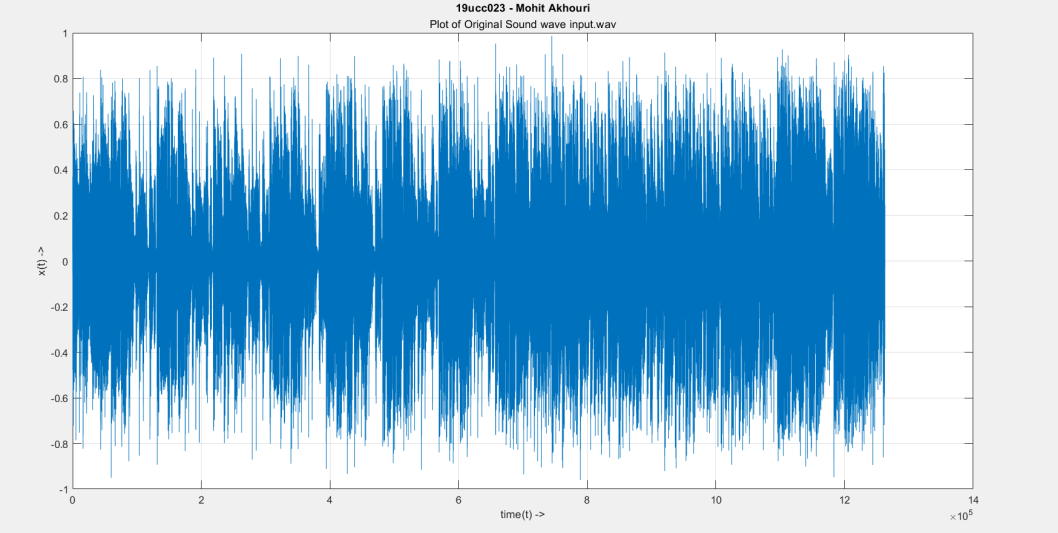


Figure 7.5 Plot of Original Audio signal

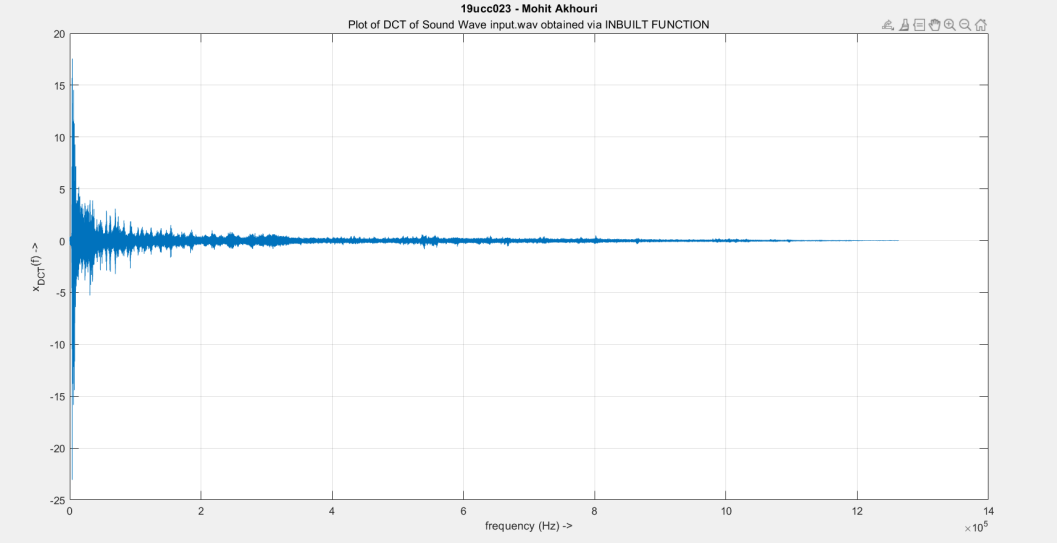


Figure 7.6 Plot of the 1D-DCT of the Audio Signal using INBUILT function

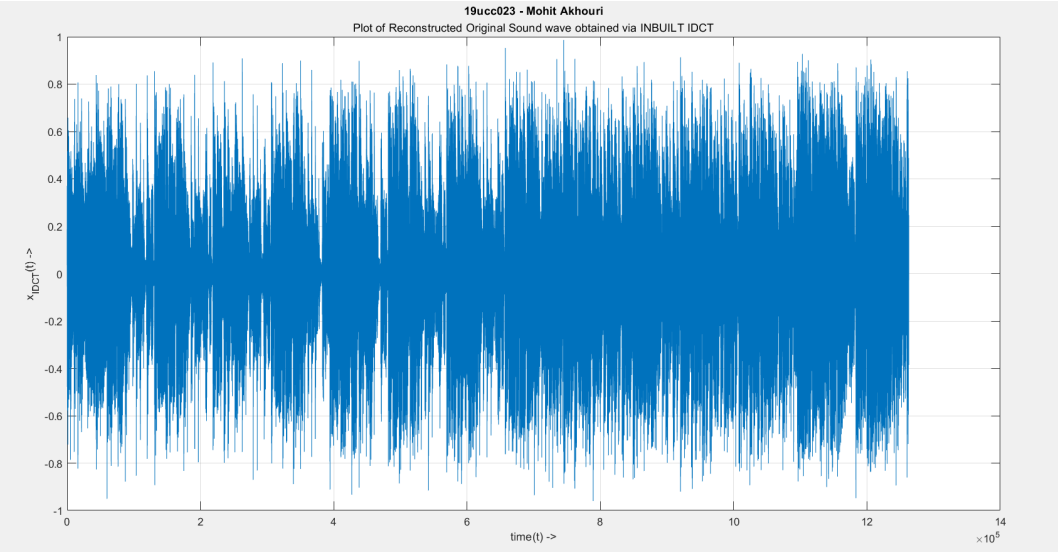


Figure 7.7 Plot of the Reconstructed Audio signal using Inbuilt 1D-IDCT

7.4.2 Using User-Defined functions for calculation of DCT and IDCT of audio signal :

```
% 19ucc023
% Mohit Akhouri
% Experiment 7 - Observation 2

% In this code , we take the input of an audio signal
% We calculate the 1D-DCT ( via user-defined function myCompression )
 to obtain the
% compressed audio wave
% Later on , we calculate the user-defined IDCT ( myDeCompression ) of
 the compressed wave to get
% the reconstructed approx. Original Sound signal

clc;
clear all;
close all;

[x,fs] = audioread('input.wav'); % Reading of audio file 'input.wav'

% Plot of Original Sound wave 'input.wav'
figure;
plot(x);
xlabel('time(t) ->');
ylabel('x(t) ->');
title('19ucc023 - Mohit Akhouri','Plot of Original Sound wave
 input.wav');
grid on;

dct_audio_userdefined = myCompression(x); % Calculation of DCT of
 input.wav via USER-DEFINED DCT (myCompression.m) - Compression of
 audio file
dct_audio_inbuilt = dct(x); % Calculation of DCT of input.wav via
 INBUILT DCT - Compression of audio file

% Plot of USER-DEFINED DCT of input.wav
figure;
subplot(2,1,1);
plot(dct_audio_userdefined);
xlabel('frequency (Hz) ->');
ylabel('x_{DCT}(f) ->');
title('Plot of DCT of Sound Wave input.wav obtained via USER-DEFINED
 FUNCTION');
grid on;

% Plot of INBUILT DCT of input.wav
subplot(2,1,2);
plot(dct_audio_inbuilt);
xlabel('frequency (Hz) ->');
ylabel('x_{DCT}(f) ->');
title('Plot of DCT of Sound Wave input.wav obtained via INBUILT
 FUNCTION');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');
```

Figure 7.8 Part 1 of the Code for the observation 2

```
audiowrite('Obs2_DCT.wav',dct_audio_userdefined,fs); % Writing
 dct_audio to audio file

idct_audio_userdefined = myDeCompression(dct_audio_userdefined); %
 Reconstructed Approx. Original audio wave via USER-DEFINED IDCT
idct_audio_inbuilt = idct(dct_audio_inbuilt); % Reconstructed Approx.
 Original audio wave via INBUILT IDCT

% Plot of Reconstructed Audio wave via USER-DEFINED IDCT
figure;
subplot(2,1,1);
plot(idct_audio_userdefined);
xlabel('time(t) ->');
ylabel('x_{IDCT}(t) ->');
title('Plot of Reconstructed Original Sound wave obtained via USER-
DEFINED IDCT');
grid on;

% Plot of Reconstructed Audio wave via INBUILT IDCT
subplot(2,1,2);
plot(idct_audio_inbuilt);
xlabel('time(t) ->');
ylabel('x_{IDCT}(t) ->');
title('Plot of Reconstructed Original Sound wave obtained via INBUILT
 IDCT');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');

audiowrite('Obs2_IDCT.wav',idct_audio_userdefined,fs);
```

Figure 7.9 Part 2 of the Code for the observation 2

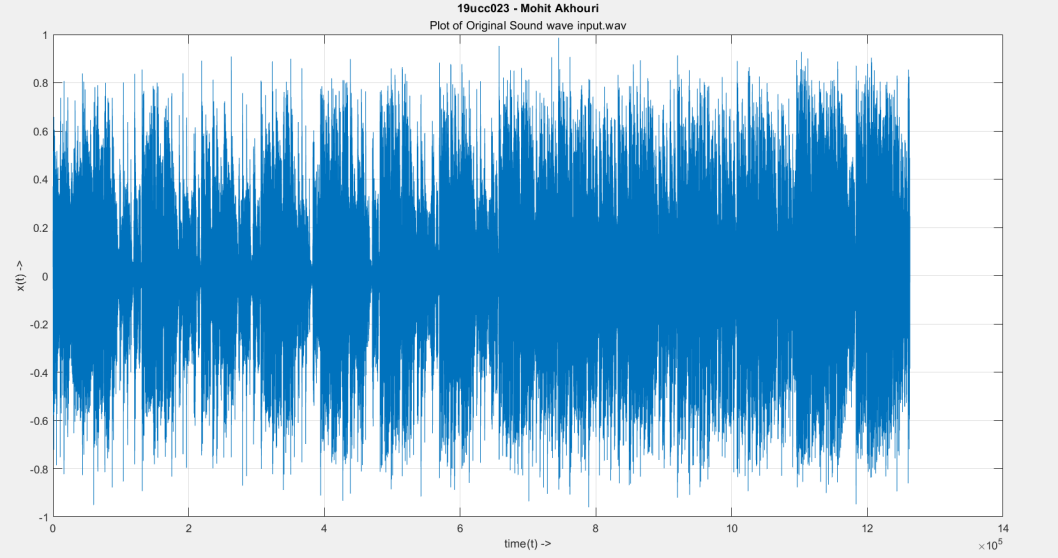


Figure 7.10 Plot of Original Audio signal

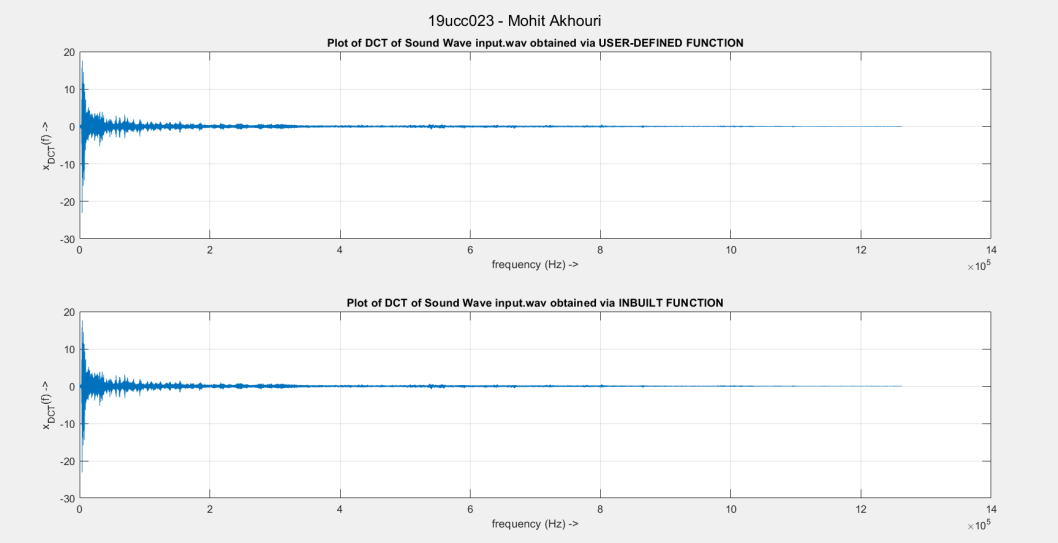


Figure 7.11 Plots of the USER-DEFINED DCT and INBUILT DCT of audio signal

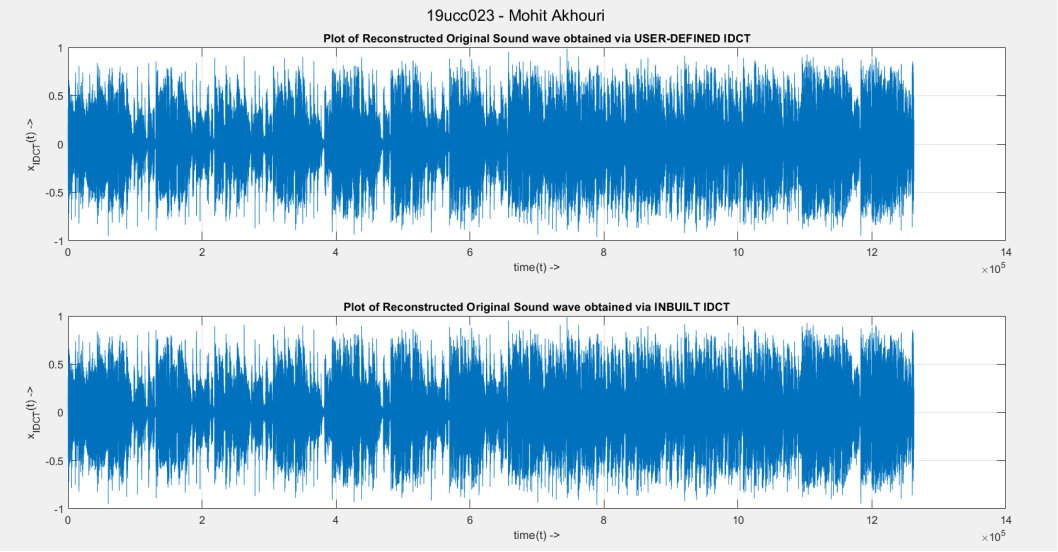


Figure 7.12 Plots of the USER-DEFINED IDCT and INBUILT IDCT

7.4.3 Division of Audio file into blocks of size 1x256 and applying threshold :

```
% 19ucc023
% Mohit Akhouri
% Experiment 7 - Observation 3

% In this code , we take the input of audio file and divide the whole
% audio file into blocks of size 1x256 . Now we apply DCT on each
% individual block and Apply the COMPRESSION ALGORITHM

% COMPRESSION ALGORITHM : In this we decide the threshold ( minimum
% threshold and maximum threshold ) , we reject the coefficient values
% which are between the threshold values and accept the rest. This is
 how
% we achieve compression.

% Finally we plot the reconstructed wave and also write it to an audio
 file

clc;
clear all;
close all;

[x,fs] = audioread('input.wav'); % Reading of audio file 'input.wav'

% Plot of Original Sound wave 'input.wav'
figure;
plot(x);
xlabel('time(t) ->');
ylabel('x(t) ->');
title('19ucc023 - Mohit Akhouri','Plot of Original Sound wave
 input.wav');
grid on;

sound(x,fs); % To hear the original sound wave via speakers

N = size(x,1); % Row size of the input audio signal

index = ceil(N/256); % calculating the index for division of blocks

% We adjust the input audio signal 'x(t)' so as it is DIVISIBLE by 256

% We pad the extra spaces by zeros
x = [x ; zeros(256*index - N,1) ];

N = size(x,1); % Re-calculation of size after adjustment of audio
 signal x(t)

threshold1 = 0.09; % Maximum threshold
threshold2 = -0.09; % Minimum threshold

% Main loop algorithm for the compression of audio signal starts here
for i=1:256:N
```

Figure 7.13 Part 1 of the code for observation 3

```
    x_block = x(i:i+255); % Dividing input audio signal into 1x256
 blocks
    x_block_dct = myCompression(x_block); % Taking DCT of 1x256 block

    % Loop for the checking of coefficients
    % Those coefficients are rejected , which are between the
 threshold
    % values , rest are accepted
    for j=1:256
        if (x_block_dct(j) >= threshold2 && x_block_dct(j) <=
 threshold1)
            x_block_dct(j) = 0;
        end
    end

    x_block_idct = myDeCompression(x_block_dct); % Inverse DCT of the
 Compressed Audio block
    x(i:i+255) = x_block_idct; % Storing the IDCT into reconstructed
 wave variable x
end

% Plot of the reconstructed Compressed wave after applying threshold
 values
% for compression
figure;
plot(x);
xlabel('time(t) ->');
ylabel('x_{reconstructed}(t) ->');
title('19ucc023 - Mohit Akhouri','Plot of Reconstructed Sound Wave
 when - removing coefficients between -0.09 and 0.09');
grid on;

audiowrite('Obs3_outputcompression.wav',x,fs); % Writing Compressed
 audio signal to a file
sound(x,fs); % To hear the compressed audio signal from speakers
```

Figure 7.14 Part 2 of the Code for the observation 3

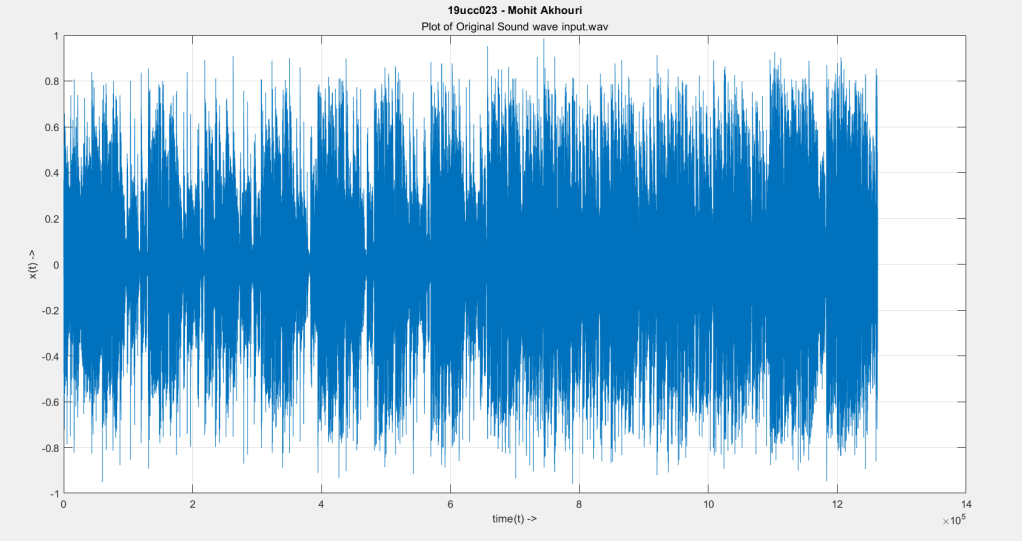


Figure 7.15 Plot of the Original Audio Signal

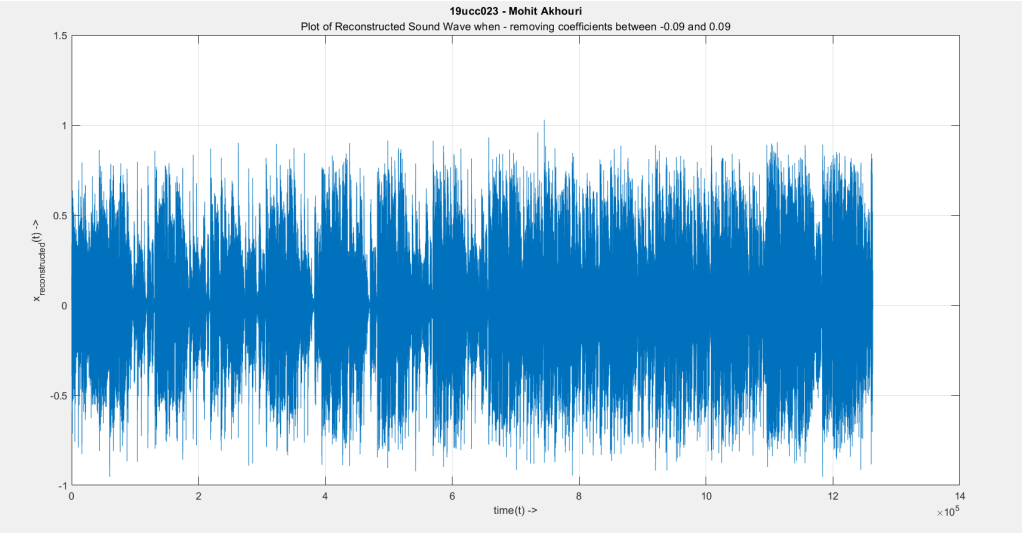


Figure 7.16 Plot of the **Compressed** Audio Signal after applying threshold

7.4.4 Applying Compression Algorithm for different threshold values :

```
% 19ucc023
% Mohit Akhouri
% Experiment 7 - Observation 4

% In this code , we take the input of audio file and divide the whole
% audio file into blocks of size 1x256 . Now we apply DCT on each
% individual block and Apply the COMPRESSION ALGORITHM

% COMPRESSION ALGORITHM : In this we decide the threshold ( minimum
% threshold and maximum threshold ) , we reject the coefficient values
% which are between the threshold values and accept the rest. This is
 how
% we achieve compression.

% We repeat the above compression algorithm for different threshold
 values
% and plot the graph between Mean square error (MSE) and Compression
 ratio

clc;
clear all;
close all;

[x,fs] = audioread('input.wav'); % Reading of audio file 'input.wav'

N = size(x,1); % Row size of the input audio signal

index = ceil(N/256); % calculating the index for division of blocks
x_orig = [x ; zeros(256*index-N,1)]; % storing the "padded with zeros"
 original signal in new variable
N = size(x_orig,1); % Re-calculation of size after adjustment of audio
 signal x(t)

% Plot of the original signal input.wav
figure;
plot(x_orig);
xlabel('time(t) ->');
ylabel('x(t) ->');
title('19ucc023 - Mohit Akhouri','Plot of Original Sound Wave
 input.wav ');
grid on;

% sound(x_orig,fs);

% Case 1 = removing coefficients for values between -0.09 and 0.09

x_recon = zeros(N,1); % Initializing variable to store reconstructed
 Audio signal

threshold_1a = 0.09; % Maximum threshold
threshold_1b = -0.09; % Minimum threshold
cnt = 0; % To hold the count of discarded coefficients
```

Figure 7.17 Part 1 of the code for observation 4

```
% Main loop algorithm for the compression of audio signal starts here
for i=1:256:N
    x_block = x_orig(i:i+255); % Dividing input audio signal into
 1x256 blocks
    x_block_dct = myCompression(x_block); % Taking DCT of 1x256 block

    % Loop for the checking of coefficients
    % Those coefficients are rejected , which are between the
 threshold
    % values , rest are accepted
    for j=1:256
        if (x_block_dct(j) >= threshold_1b && x_block_dct(j) <=
 threshold_1a)
            x_block_dct(j) = 0;
            cnt = cnt + 1;
        end
    end

    x_block_idct = myDeCompression(x_block_dct);  % Inverse DCT of the
 Compressed Audio block
    x_recon(i:i+255) = x_block_idct; % Storing the IDCT into
 reconstructed wave variable x_recon
end

% Plot of the reconstructed Compressed wave after applying threshold
 values
% for compression
figure;
plot(x_recon);
xlabel('time(t) ->');
ylabel('x_{reconstructed}(t) ->');
title('19ucc023 - Mohit Akhouri','Plot of Reconstructed Sound Wave for
 Case 1 - removing coefficients between -0.09 and 0.09');
grid on;

sound(x_recon,fs); % To hear the compressed audio signal from speakers
audiowrite('Obs4_outputcompression_case1.wav',x_recon,fs); % Writing
 Compressed audio signal to a file

mse_case1 = mse(x_orig,x_recon); % calculation of MSE between Original
 and Reconstructed audio signal for case 1
p_case1 = (N-cnt)/N; % Calculation of Compression ratio for case 1


% Case 2 = removing coefficients for values between -0.5 and 0.5

x_recon = zeros(N,1); % Initializing variable to store reconstructed
 Audio signal

threshold_2a = 0.5; % Maximum threshold
threshold_2b = -0.5; % Minimum threshold
cnt = 0; % To hold the count of discarded coefficients
```

Figure 7.18 Part 2 of the code for observation 4

```
% Main loop algorithm for the compression of audio signal starts here
for i=1:256:N
    x_block = x_orig(i:i+255); % Dividing input audio signal into
 1x256 blocks
    x_block_dct = myCompression(x_block); % Taking DCT of 1x256 block

    % Loop for the checking of coefficients
    % Those coefficients are rejected , which are between the
 threshold
    % values , rest are accepted
    for j=1:256
        if (x_block_dct(j) >= threshold_2b && x_block_dct(j) <=
 threshold_2a)
            x_block_dct(j) = 0;
            cnt = cnt + 1;
        end
    end

    x_block_idct = myDeCompression(x_block_dct); % Inverse DCT of the
 Compressed Audio block
    x_recon(i:i+255) = x_block_idct; % Storing the IDCT into
 reconstructed wave variable x_recon
end

% Plot of the reconstructed Compressed wave after applying threshold
 values
% for compression
figure;
plot(x_recon);
xlabel('time(t) ->');
ylabel('x_{reconstructed}(t) ->');
title('19ucc023 - Mohit Akhouri','Plot of Reconstructed Sound Wave for
 Case 2 - removing coefficients between -0.5 and 0.5');
grid on;

sound(x_recon,fs); % To hear the compressed audio signal from speakers
audiowrite('Obs4_outputcompression_case2.wav',x_recon,fs); % Writing
 Compressed audio signal to a file

mse_case2 = mse(x_orig,x_recon); % calculation of MSE between Original
 and Reconstructed audio signal for case 2
p_case2 = (N-cnt)/N; % Calculation of Compression ratio for case 2


% Case 3 = removing coefficients for values between -0.01 and 0.01

x_recon = zeros(N,1); % Initializing variable to store reconstructed
 Audio signal

threshold_3a = 0.01; % Maximum threshold
threshold_3b = -0.01; % Minimum threshold
cnt = 0; % To hold the count of discarded coefficients

% Main loop algorithm for the compression of audio signal starts here
```

Figure 7.19 Part 3 of the code for observation 4

```
for i=1:256:N
    x_block = x_orig(i:i+255); % Dividing input audio signal into
 1x256 blocks
    x_block_dct = myCompression(x_block); % Taking DCT of 1x256 block

    % Loop for the checking of coefficients
    % Those coefficients are rejected , which are between the
 threshold
    % values , rest are accepted
    for j=1:256
        if (x_block_dct(j) >= threshold_3b && x_block_dct(j) <=
 threshold_3a)
            x_block_dct(j) = 0;
            cnt = cnt + 1;
        end
    end

    x_block_idct = myDeCompression(x_block_dct); % Inverse DCT of the
 Compressed Audio block
    x_recon(i:i+255) = x_block_idct; % Storing the IDCT into
 reconstructed wave variable x_recon
end

% Plot of the reconstructed Compressed wave after applying threshold
 values
% for compression
figure;
plot(x_recon);
xlabel('time(t) ->');
ylabel('x_{reconstructed}(t) ->');
title('19ucc023 - Mohit Akhouri','Plot of Reconstructed Sound Wave for
 case 3 - removing coefficients between -0.01 and 0.01');
grid on;

sound(x_recon,fs); % To hear the compressed audio signal from speakers
audiowrite('Obs4_outputcompression_case3.wav',x_recon,fs); % Writing
 Compressed audio signal to a file

mse_case3 = mse(x_orig,x_recon); % calculation of MSE between Original
 and Reconstructed audio signal for case 3
p_case3 = (N-cnt)/N; % Calculation of Compression ratio for case 3

% Plotting Graph between MSE and Compression Ratio for different cases

mse_array = zeros(1,3); % Initializing MSE Array
p_array = zeros(1,3); % Initializing Compression Ratio Array

% Storing MSE for different cases in array
mse_array(1) = mse_case1;
mse_array(2) = mse_case2;
mse_array(3) = mse_case3;

% Storing Compression Ratio for different cases in array
p_array(1) = p_case1;
```

Figure 7.20 Part 4 of the code for observation 4

```
p_array(2) = p_case2;
p_array(3) = p_case3;

% Plot of MSE vs. Compression Ratio
figure;
stem(mse_array,p_array,'Linewidth',1.5);
xlabel('Compression ratio (\rho) ->');
ylabel('Mean square error (\epsilon) ->');
title('19ucc023 - Mohit Akhouri','Plot of Mean square error (\epsilon)
 vs. Compression Ratio (\rho)');
grid on;
```

Figure 7.21 Part 5 of the code for observation 4

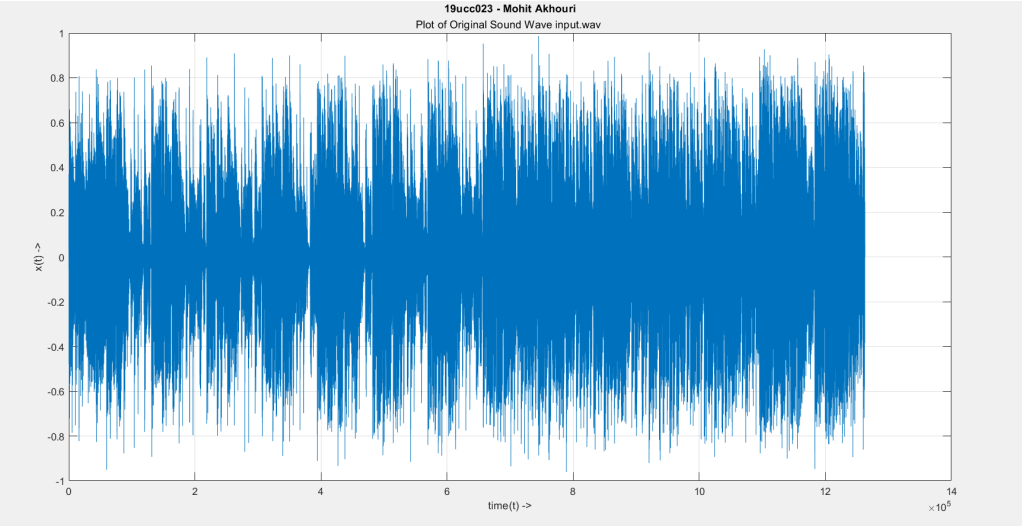


Figure 7.22 Plot of the Original Audio Signal

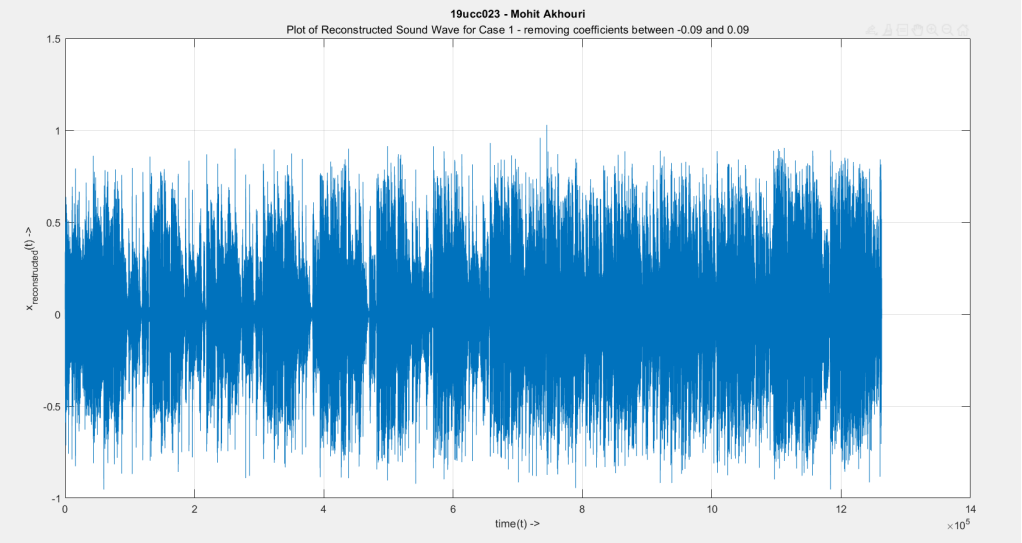


Figure 7.23 Plot of the **Compressed** Audio signal for threshold **case 1**

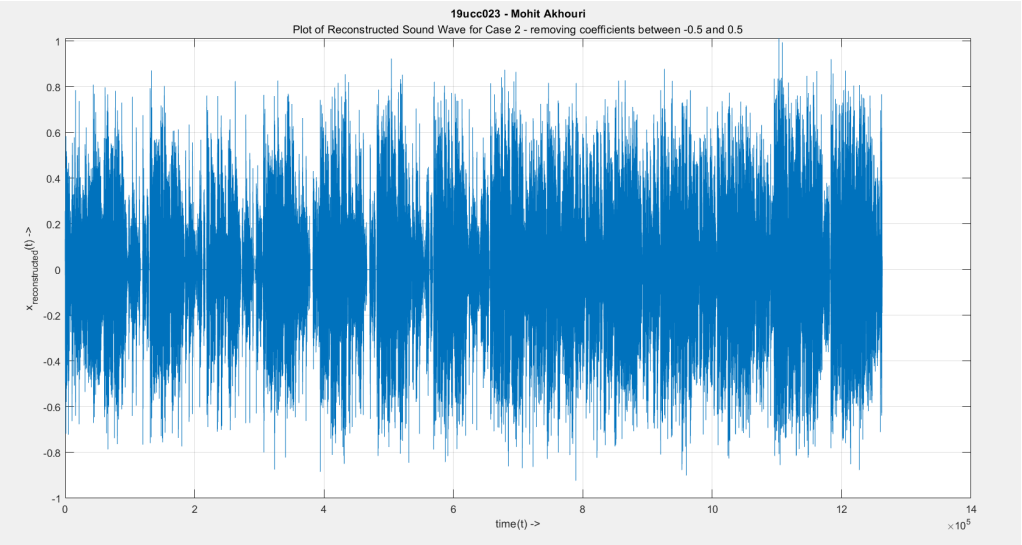


Figure 7.24 Plot of the **Compressed** Audio signal for threshold **case 2**

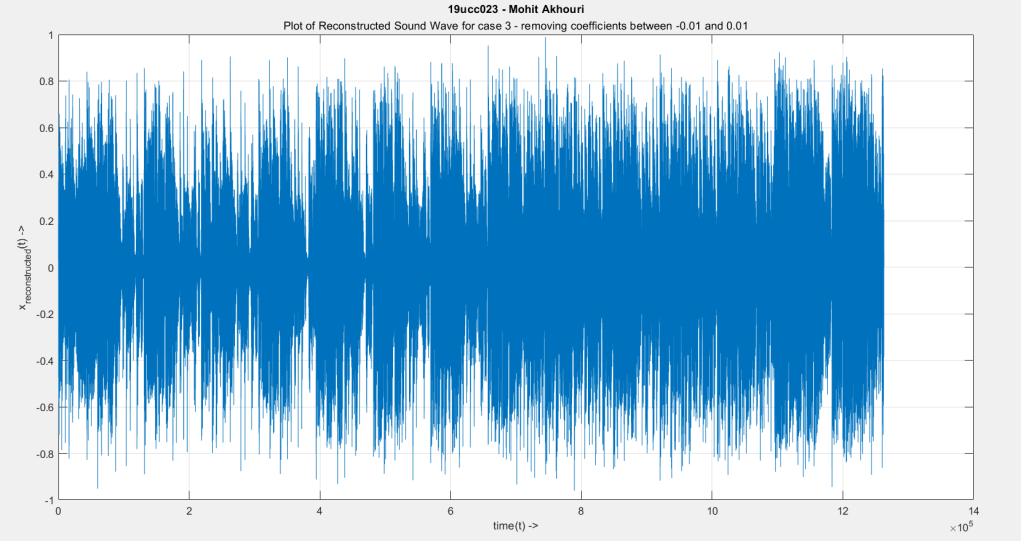


Figure 7.25 Plot of the **Compressed** Audio signal for threshold **case 3**

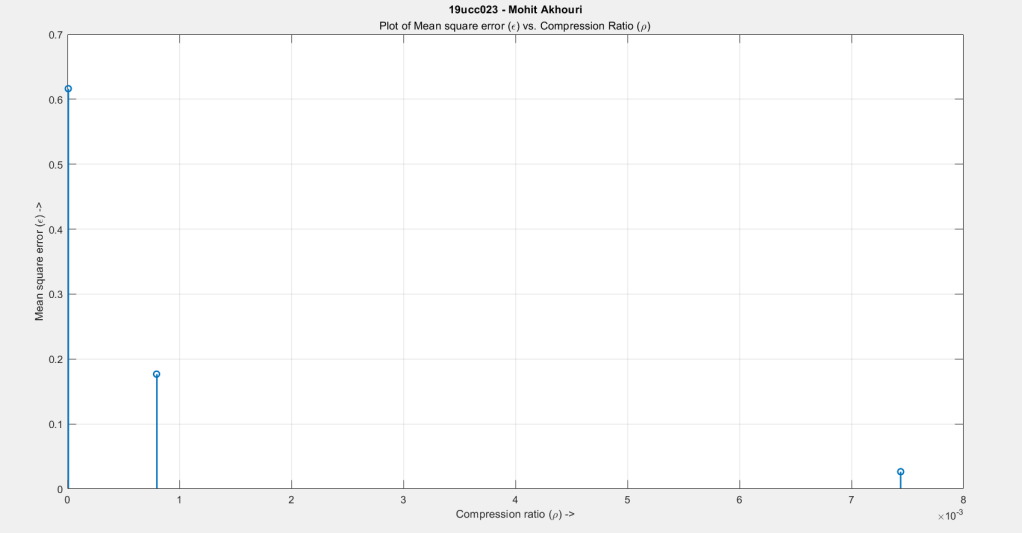


Figure 7.26 Graph of Mean Square Error (ϵ) vs. Compression Ratio (ρ)

7.4.5 Simulink based Audio Compression :

```
% 19ucc023
% Mohit Akhouri
% Experiment 7 - Observation 5

% This code will call the Simulink Model 'Simulink_Observation_5' and
% perform DCT based compression of given audio input file 'input.wav'

% It also performs the IDCT and we get back the reconstructed audio
 wave

sim('Simulink_Observation_5'); % Calling the Simulink Model
[x,fs] = audioread('input.wav'); % For calculation of sampling
 frequency of input.wav

sound(x,fs);

% Plot of Original Sound signal input.wav
figure;
plot(out.Orig_Audio.data);
xlabel('time(t) ->');
ylabel('x(t) ->');
title('19ucc023 - Mohit Akhouri','Plot of Original Sound wave
 input.wav');
grid on;

% Plot of DCT of input.wav obtained via Simulink Model
figure;
plot(out.DCT_Audio.data);
xlabel('frequency (Hz) ->');
ylabel('x_{DCT}(f) ->');
title('19ucc023 - Mohit Akhouri','Plot of DCT of Sound Wave input.wav
 obtained via Simulink Model');
grid on;

audiowrite('Obs5_DCT_AudioWave.wav',out.DCT_Audio.data,fs); % Writing
 DCT_AudioWave to audio file

% Plot of compressed reconstructed audio signal
figure;
plot(out.Compressed_Audio.data);
xlabel('time(t) ->');
ylabel('x_{IDCT}(t) ->');
title('19ucc023 - Mohit Akhouri','Plot of Reconstructed Original Sound
 wave obtained via Simulink Model');
grid on;

audiowrite('Obs5_Compressed_Audio.wav',out.Compressed_Audio.data,fs); %
 Writing Compressed audio signal to audio file
sound(out.Compressed_Audio.data,fs); % Hearing Compressed sound from
 speakers
```

Figure 7.27 Code for the observation 5

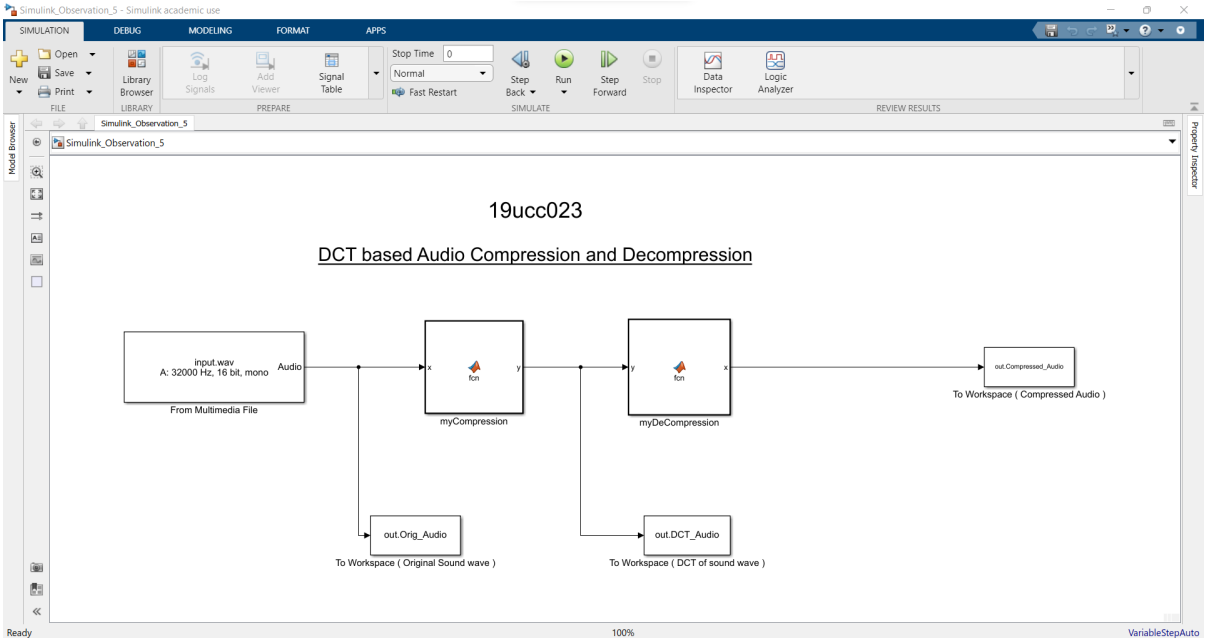


Figure 7.28 Simulink Model used for Audio Compression

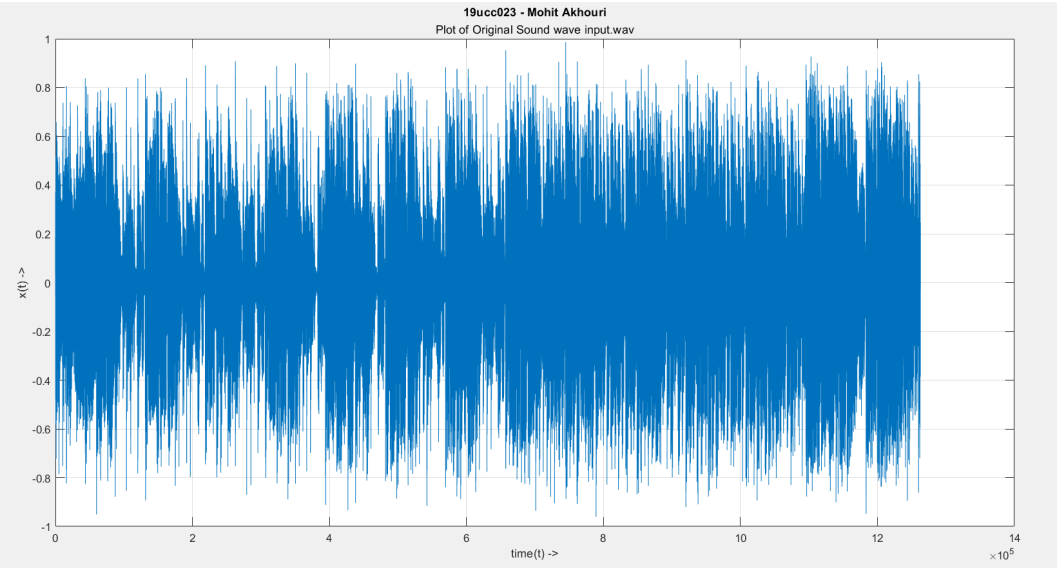


Figure 7.29 Plot of the Original Audio Signal obtained via Simulink Model

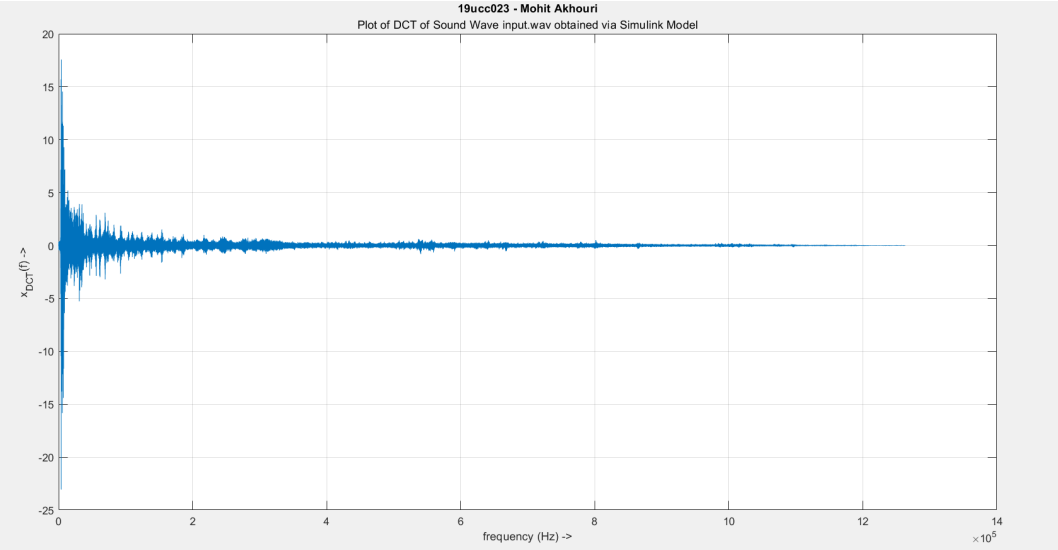


Figure 7.30 Plot of the DCT of audio signal obtained via Simulink Model

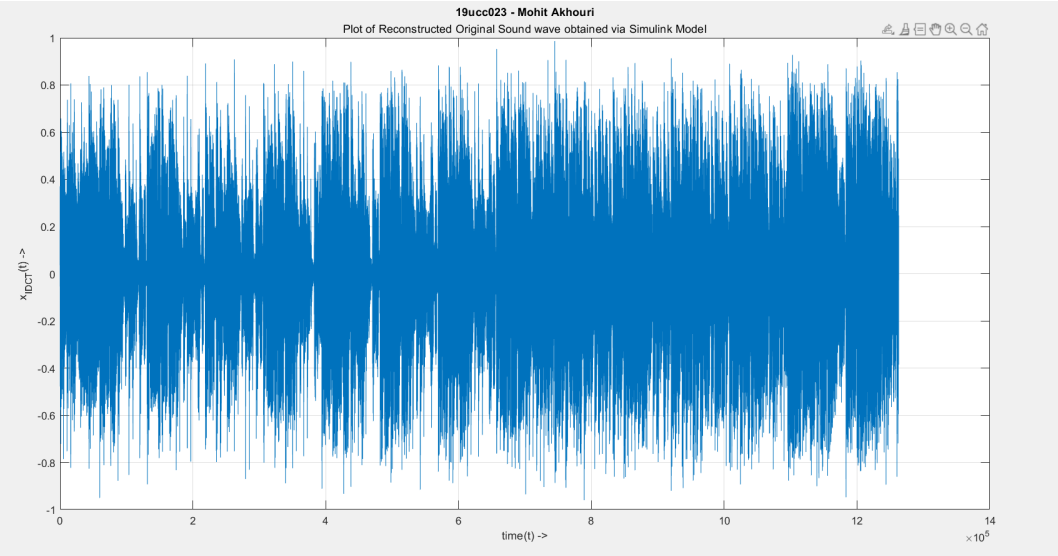


Figure 7.31 Plot of the Reconstructed Audio signal obtained via Simulink Model

7.4.6 Functions used in main codes for DCT and IDCT for Audio file compression :

7.4.6.1 myCompression.m function code :

```
function [y] = myCompression(x)

% 19ucc023
% Mohit Akhouri

% ALGORITHM :
% This function will compute the 1D-DCT of given input audio signal
% Basically this function performs the compression of the audio signal
% through 1D-DCT

N = size(x,1); % row size of the audio signal x(t)

y = zeros(N,1); % Initializing output variable to store the DCT
w = zeros(N,1); % factor 'w' used in the expression of DCT calculation

% Loop algorithm for the calculation of the different values of factor
 'w'
for i=1:N
    if i==1
        w(i) = 1/sqrt(N);
    else
        w(i) = sqrt(2/N);
    end
end

% Main Loop algorithm for the calculation of DCT is as follows
for k=1:N
    sum = 0;
    for n=1:N
        sum = sum + ( x(n)*cos((pi*(2*n-1)*(k-1))/(2*N)) );
    end
    y(k) = w(k) * sum;
end
```

Published with MATLAB® R2020b

Figure 7.32 myCompression.m function used to calculate the **DCT** of the given input audio signal

7.4.6.2 myDeCompression.m function code :

```
function [x] = myDeCompression(y)

% 19ucc023
% Mohit Akhouri

% ALGORITHM :
% This function will compute the 1D-IDCT of given compressed audio
 signal
% Basically this function reconstructs back the approximated
 Compressed
% audio signal obtained after DCT

N = size(y,1); % row size of the audio signal y(t)
x = zeros(N,1); % Initializing output variable to store the IDCT

w = zeros(N,1); % factor 'w' used in the expression of IDCT
 calculation

% Loop algorithm for the calculation of the different values of factor
 'w'
for i=1:N
    if i==1
        w(i) = 1/sqrt(N);
    else
        w(1) = sqrt(2/N);
    end
end

% Main Loop algorithm for the calculation of IDCT is as follows
for n=1:N
    sum = 0;
    for k=1:N
        sum = sum + ( w(k)*y(k)*cos((pi*(2*n-1)*(k-1))/(2*N)) );
    end
    x(n) = sum;
end

end
```

Published with MATLAB® R2020b

Figure 7.33 myDeCompression.m function for calculation of **IDCT** for reconstruction of audio signal

7.5 Conclusion

In this experiment , we learnt the concepts of **Discrete Cosine Transform** , **Inverse DCT** and **Transform Based Audio Compression** of Digital Signal Processing. We learnt about DCT matrix and how to compute the DCT of any given audio signal. We learnt the significance of DCT in **Audio Compression**. We also observed the compression of audio for various cases - different threshold values like -0.09 to 0.09 , -0.5 to 0.5 and -0.01 to 0.01. We **observed** the difference in compressed audio signals using the **sound** function of MATLAB. We also learnt about the relation between **Compression Ratio** (ρ) and **Mean square error** (ϵ). We also plotted the graph of Mean square error vs. Compression ratio for different values of Compression ratio. We also learnt new MATLAB functions like **sound** and **audiowrite**. We also implemented the Audio Compression in Simulink and compared the results obtained from MATLAB coding.

Chapter *8*

Experiment - 8

8.1 Aim of the Experiment

- Radix-2 FFT Algorithm

8.2 Software Used

- MATLAB
- Simulink

8.3 Theory

8.3.1 About Discrete Fourier Transform (DFT) :

The **discrete Fourier transform (DFT)** converts a finite sequence of **equally-spaced samples** of a function into a same-length sequence of equally-spaced samples of the discrete-time Fourier transform (DTFT), which is a complex-valued function of frequency. The interval at which the DTFT is sampled is the reciprocal of the duration of the input sequence. An **inverse DFT** is a **Fourier series**, using the DTFT samples as coefficients of complex sinusoids at the corresponding DTFT frequencies. It has the same sample-values as the original input sequence. The **DFT** is therefore said to be a **frequency domain representation of the original input sequence**. If the original sequence spans all the non-zero values of a function, its DTFT is continuous (and periodic), and the DFT provides discrete samples of one cycle. If the original sequence is one cycle of a periodic function, the DFT provides all the non-zero values of one DTFT cycle.

The DFT is the most important discrete transform, used **to perform Fourier analysis** in many practical applications. It is used in **digital signal processing**. The DFT is also used to efficiently **solve partial differential equations**, and to perform other operations such as **convolutions** or **multiplying large integers**.

8.3.2 About Fast Fourier Transform (FFT) :

A **fast Fourier transform (FFT)** is an **algorithm** that computes the discrete Fourier transform (DFT) of a sequence, or its inverse (IDFT). Fourier analysis converts a signal from its original domain (often time or space) to a representation in the frequency domain and vice versa. The DFT is obtained by decomposing a sequence of values into components of different frequencies. An FFT rapidly computes such transformations by **factorizing the DFT matrix into a product of sparse (mostly zero) factors**. As a result, it manages to reduce the complexity of computing the DFT from $O(N^2)$ which arises if one simply applies the definition of DFT, to $O(NlogN)$, where N is the **data size**.

Fast Fourier transforms are widely used for applications in engineering, music, science, and mathematics. The basic ideas were popularized in 1965, but some algorithms had been derived as early as 1805.

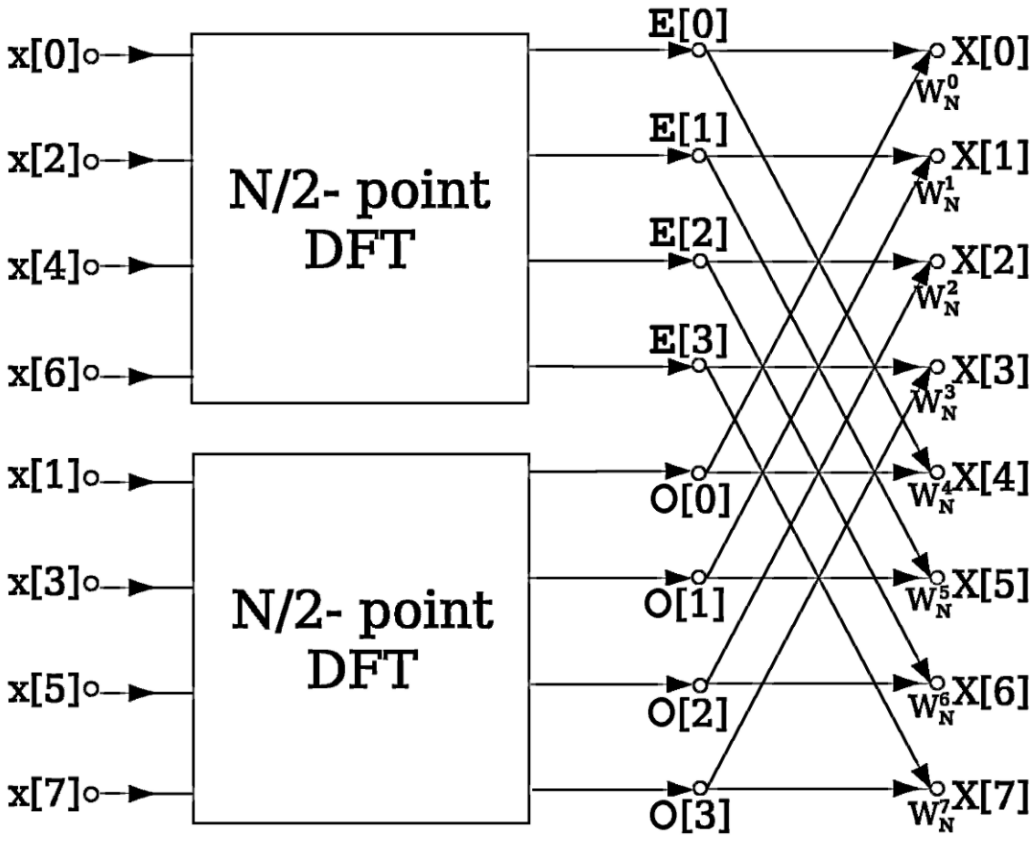


Figure 8.1 Butterfly structure of DIT-FFT Algorithm

The **FFT** is used in **digital recording**, **sampling**, **additive synthesis** and **pitch correction software**. Some of the important applications of the FFT include : fast large-integer and polynomial multiplication, efficient matrix–vector multiplication for Toeplitz, circulant and other structured matrices, filtering algorithms , solving difference equations and computation of isotopic distributions.

8.3.2.1 About DIT-FFT Algorithm :

The steps to be followed to find the **N-point DFT** of a sequence x[n] of length N is as follows :

- The N length sequence can be divided into two $\frac{N}{2}$ point data sequence f1[n] and f2[n], corresponding to the even numbered and odd numbered samples of x[n].
- The DFT of x[n] is given by :

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn} \tag{8.1}$$

  where k = 0,1,2...N-1 and W_N = exp(-j*2*pi/N)

- We now break x[n] into **even** and **odd** sequence :

$$X[k] = \sum_{n=even} x[n] W_N^{kn} + \sum_{n=odd} x[n] W_N^{kn} \tag{8.2}$$

$$X[k] = \sum_{m=0}^{\frac{N}{2}-1} x[2m] W_N^{2mk} + \sum_{m=0}^{\frac{N}{2}-1} x[2m+1] W_N^{(2m+1)k} \tag{8.3}$$

  The **final expression** for X[k] now becomes :

$$X[k] = \sum_{m=0}^{\frac{N}{2}-1} f1[m] W_{N/2}^{mk} + \sum_{m=0}^{\frac{N}{2}-1} f2[m] W_{N/2}^{mk} \tag{8.4}$$

- The above equation now becomes after simplifying :

$$X[k] = F_1[k] + F_2[k].W_N^k \tag{8.5}$$

  In the above equation , k=0,1,2...N-1 where $F_1[k]$ and $F_2[k]$ are **N/2 point DFT** sequence of $f_1[m]$ and $f_2[m]$.

- Due to **periodicity** of $F_1[k]$ and $F_2[k]$ (which is $F_i[k + \frac{N}{2}] = F_i[k]$), the equations for the both halves are as follows :

$$X[k] = F_1[k] + F_2[k].W_N^k \tag{8.6}$$

$$X[k + \frac{N}{2}] = F_1[k] - F_2[k].W_N^k \tag{8.7}$$

  In the above equation , k=0,1,2.... $\frac{N}{2}$ - 1

8.4 Code and results

8.4.1 Simulating 2-point fft using radix2-fft method and verifying with inbuilt function :

```
% 19ucc023
% Mohit Akhouri
% Experiment 8 - Observation 1

% In this code, we will take a random 2-length sequence x[n]
% Then we will find its FFT using both INBUILT function fft and
% USER-DEFINED function my_dit_fft and also plot the graphs
 respectively

clc;
clear all;
close all;

N = 2; % Size of the seqeunce x[n]
x = randperm(4,N); % using randperm(k,N) to select RANDOM N integers
 from range 1-k , here it selects N=2 integers from range 1-4

% Plot of random seqeunce x[n]
figure;
stem(x,'Linewidth',1.8);
xlabel('samples(n) ->');
ylabel('x[n] ->');
title('19ucc023 - Mohit Akhouri','Plot of Input Sequence x[n] of
 length = 2');
grid on;

fft_inbuilt = fft(x,N); % Calculation of N-point DFT via INBUILT
 function fft
fft_user_defined = my_dit_fft(x,N); % Calculation of N-point DFT using
 RADIX-2 fft algorithm via USER-DEFINED function my_dit_fft

% Plot of N-point DFT obtained via INBUILT function fft(x,N)
figure;
subplot(2,1,1);
stem(fft_inbuilt,'Linewidth',1.8);
xlabel('samples(k) ->');
ylabel('X(k) ->');
title('2-point DFT of sequence x[n] using INBUILT function fft(x,N)');
grid on;

% Plot of N-point DFT obtained via USER-DEFINED function
 my_dit_fft(x,N)
subplot(2,1,2);
stem(fft_user_defined,'Linewidth',1.8);
xlabel('samples(k) ->');
ylabel('X(k) ->');
title('2-point DFT of sequence x[n] using USER-DEFINED function my
\_dit\_fft(x,N)');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');
```

Figure 8.2 Code for the observation 1

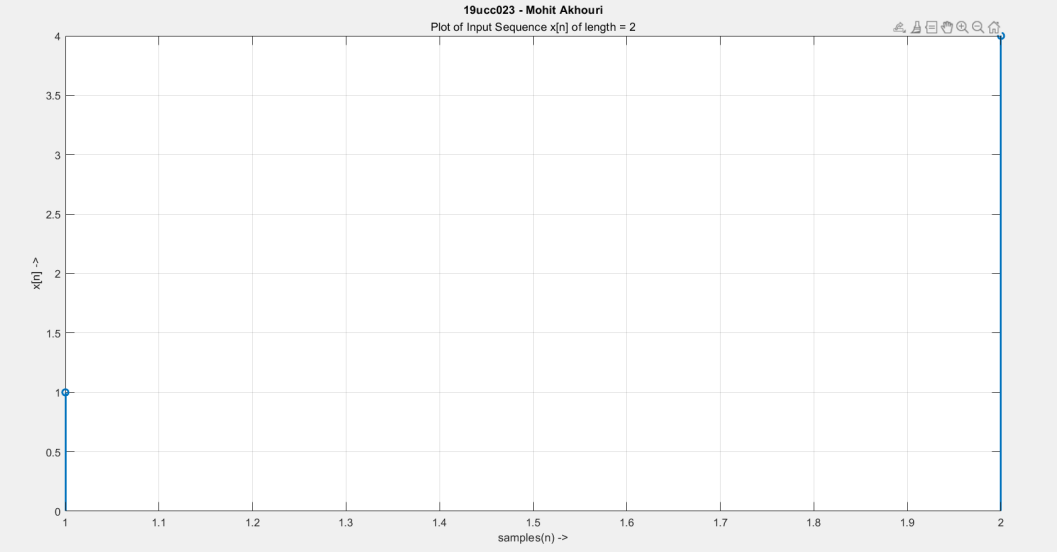


Figure 8.3 Plot of RANDOM Input Sequence of length = 2

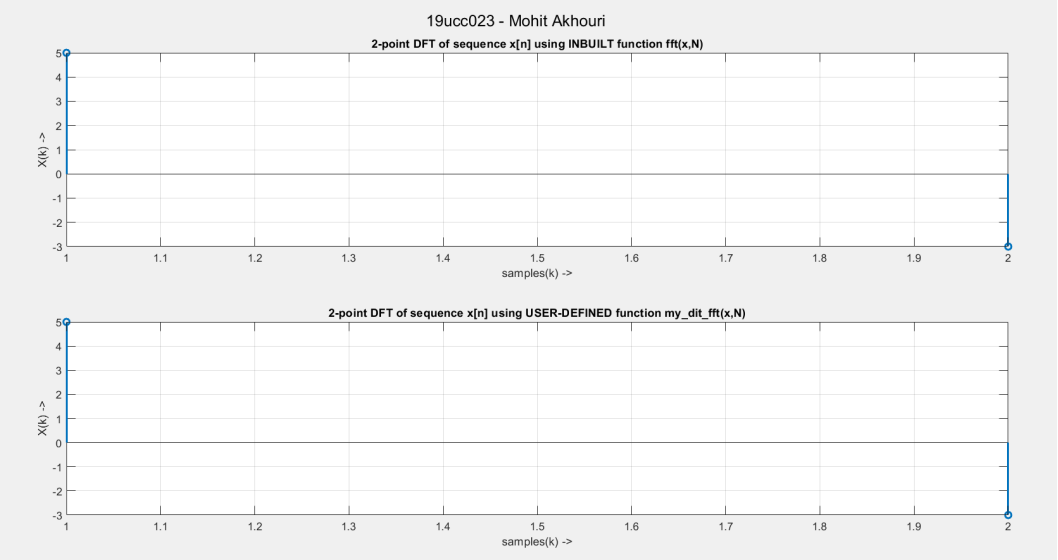


Figure 8.4 Plots of the DFT of input sequence x[n] via both INBUILT and USER-DEFINED functions

8.4.2 Simulating 4-point fft using radix2-fft method and verifying with inbuilt function :

```
% 19ucc023
% Mohit Akhouri
% Experiment 8 - Observation 2

% In this code, we will take a random 4-length sequence x[n]
% Then we will find its FFT using both INBUILT function fft and
% USER-DEFINED function my_dit_fft and also plot the graphs
 respectively

clc;
clear all;
close all;

N = 4; % Size of the seqeunce x[n]
x = randperm(6,N); % using randperm(k,N) to select RANDOM N integers
 from range 1-k , here it selects N=4 integers from range 1-6

% Plot of random seqeunce x[n]
figure;
stem(x,'Linewidth',1.8);
xlabel('samples(n) ->');
ylabel('x[n] ->');
title('19ucc023 - Mohit Akhouri','Plot of Input Sequence x[n] of
 length = 4');
grid on;

fft_inbuilt = fft(x,N); % Calculation of N-point DFT via INBUILT
 function fft
fft_user_defined = my_dit_fft(x,N); % Calculation of N-point DFT using
 RADIX-2 fft algorithm via USER-DEFINED function my_dit_fft

% Plot of N-point DFT obtained via INBUILT function fft(x,N)
figure;
subplot(2,1,1);
stem(fft_inbuilt,'Linewidth',1.8);
xlabel('samples(k) ->');
ylabel('X(k) ->');
title('4-point DFT of sequence x[n] using INBUILT function fft(x,N)');
grid on;

% Plot of N-point DFT obtained via USER-DEFINED function
 my_dit_fft(x,N)
subplot(2,1,2);
stem(fft_user_defined,'Linewidth',1.8);
xlabel('samples(k) ->');
ylabel('X(k) ->');
title('4-point DFT of sequence x[n] using USER-DEFINED function my
\_dit\_fft(x,N)');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');
```

Figure 8.5 Code for the observation 2

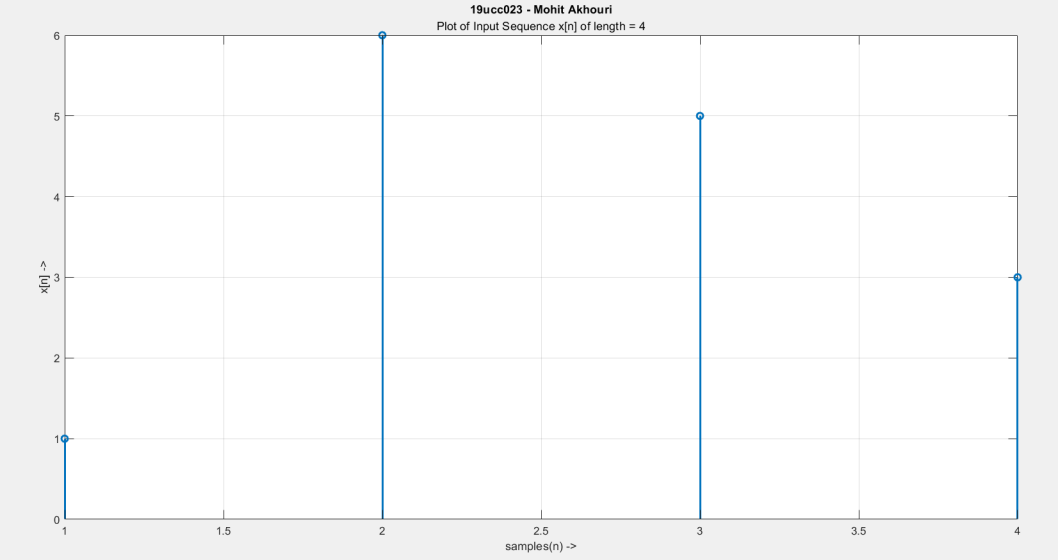


Figure 8.6 Plot of RANDOM Input Sequence of length = 4

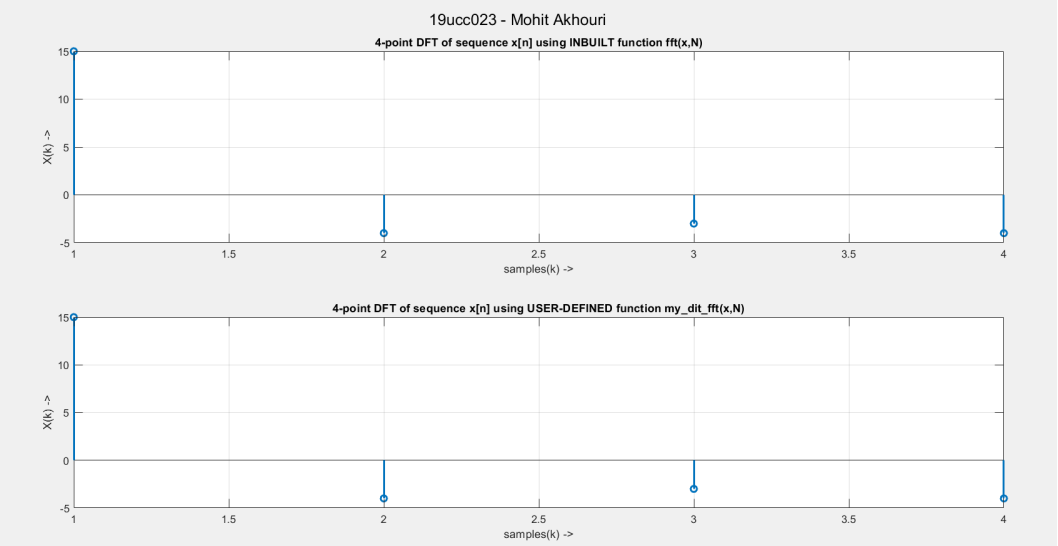


Figure 8.7 Plots of the DFT of input sequence x[n] via both INBUILT and USER-DEFINED functions

8.4.3 Simulating 8-point fft using radix2-fft method and verifying with inbuilt function :

```
% 19ucc023
% Mohit Akhouri
% Experiment 8 - Observation 3

% In this code, we will take a random 8-length sequence x[n]
% Then we will find its FFT using both INBUILT function fft and
% USER-DEFINED function my_dit_fft and also plot the graphs
 respectively

clc;
clear all;
close all;

N = 8; % Size of the seqeunce x[n]
x = randperm(10,N); % using randperm(k,N) to select RANDOM N integers
 from range 1-k , here it selects N=8 integers from range 1-10

% Plot of random seqeunce x[n]
figure;
stem(x,'Linewidth',1.8);
xlabel('samples(n) ->');
ylabel('x[n] ->');
title('19ucc023 - Mohit Akhouri','Plot of Input Sequence x[n] of
 length = 8');
grid on;

fft_inbuilt = fft(x,N); % Calculation of N-point DFT via INBUILT
 function fft
fft_user_defined = my_dit_fft(x,N); % Calculation of N-point DFT using
 RADIX-2 fft algorithm via USER-DEFINED function my_dit_fft

% Plot of N-point DFT obtained via INBUILT function fft(x,N)
figure;
subplot(2,1,1);
stem(fft_inbuilt,'Linewidth',1.8);
xlabel('samples(k) ->');
ylabel('X(k) ->');
title('8-point DFT of sequence x[n] using INBUILT function fft(x,N)');
grid on;

% Plot of N-point DFT obtained via USER-DEFINED function
 my_dit_fft(x,N)
subplot(2,1,2);
stem(fft_user_defined,'Linewidth',1.8);
xlabel('samples(k) ->');
ylabel('X(k) ->');
title('8-point DFT of sequence x[n] using USER-DEFINED function my
\_dit\_fft(x,N)');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');
```

Figure 8.8 Code for the observation 3

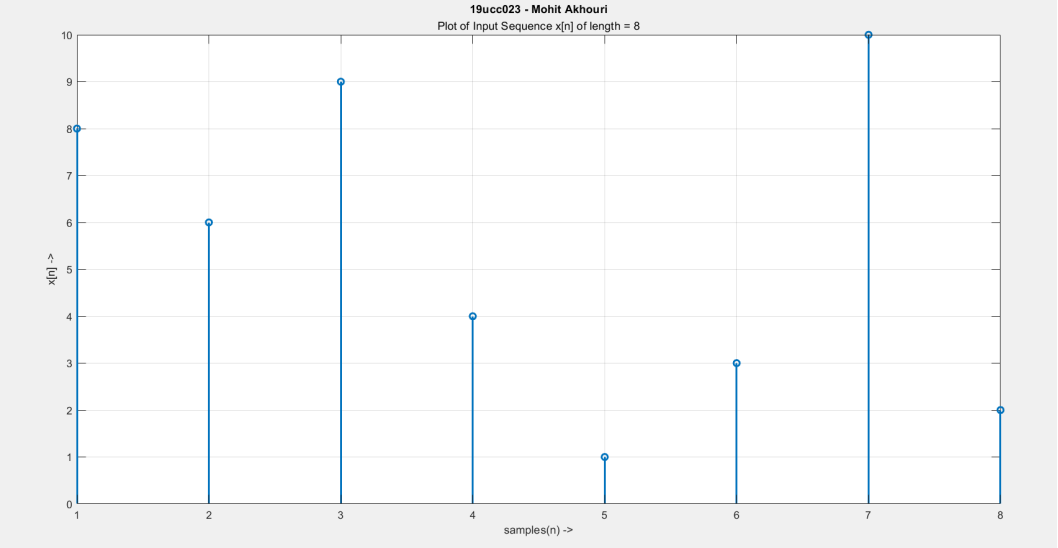


Figure 8.9 Plot of RANDOM Input Sequence of length = 8

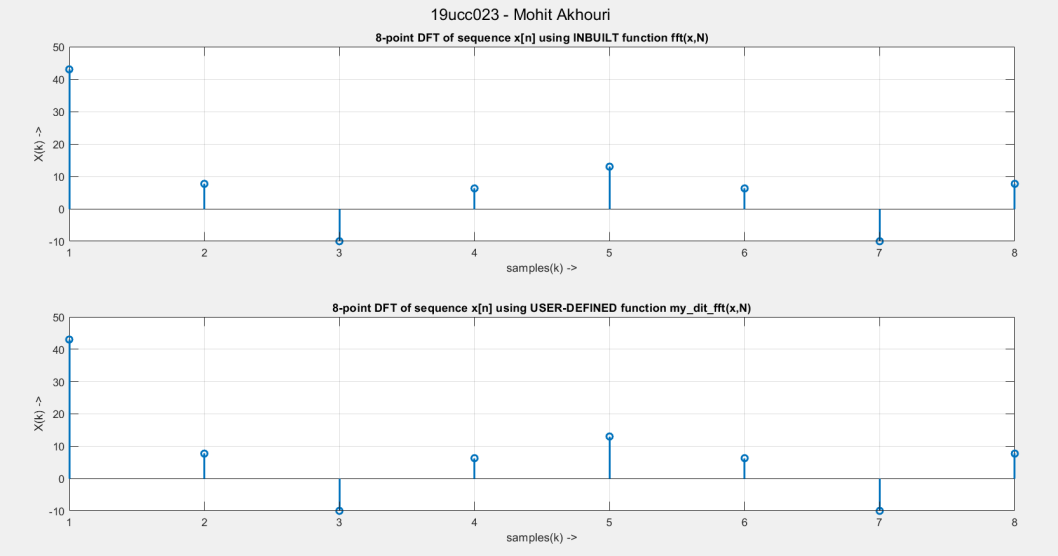


Figure 8.10 Plots of the DFT of input sequence x[n] via both INBUILT and USER-DEFINED functions

8.4.4 Calculation and Plot of Speed Factor vs. N for N=2,4,8,16 and 128 :

```
% 19ucc023
% Mohit Akhouri
% Experiment 8 - Observation 4

% In this code , we will calculate the number of complex
 multiplications in
% both DIRECT METHOD and RADIX-2 FFT METHOD , then we will plot the
 graph
% between speed factor vs. N ( for various values of N = 2,4,8...128 )

clc;
clear all;
close all;

% ALGORITHM : First we will calculate number of complex
 multiplications in
% DIRECT and RADIX-2 FFT METHOD which are as follows :

% Complex Multiplications in Direct Method      = N^2              -
 eqn 1
% Complex Multiplications in Radix-2 fft Method = (N/2)*(log2(N))  -
 eqn 2

% In the above equations N is the length of the input sequence x[n]
% Lastly we will calculate the ratio of eqn 1 and eqn 2 , which is
 defined
% as speed factor and then plot the graph between speed factor and N

N_array = [2 4 8 16 128]; % Array to store the values of N ( size of
 sequence x[n] )

Speed_Factor = zeros(1,5); % Array to store the calculated Speed
 factor for various values of N

comp_mult_direct = 0; % To store the number of complex multiplications
 obtained via Direct method
comp_mult_radix2 = 0; % To store the number of complex multiplications
 obtained via Radix-2 fft method

% Main loop algorithm for calculation of speed factor for various N
 values
for i=1:5

    N = N_array(i); % size stored in N variable

    comp_mult_direct = N^2; % Calculation of complex mult. in Direct
 Method
    comp_mult_radix2 = (N/2)*(log2(N)); % Calculation of complex mult.
 in Radix-2 fft Method

    Speed_Factor(i) = comp_mult_direct / comp_mult_radix2; %
 Calculation of speed factor
```

Figure 8.11 Part 1 of the code for observation 4

```
end

% Plot of Speed factor vs. Different values of N
figure;
stem(N_array,Speed_Factor,'Linewidth',1.8);
xlabel('N ->');
ylabel('Speed Factor ->');
title('19ucc023 - Mohit Akhouri','Plot of Speed Factor vs. N ( length
 of sequence x[n] ) for N = 2,4,8,16,128');
grid on;
```

Figure 8.12 Part 2 of the code for observation 4

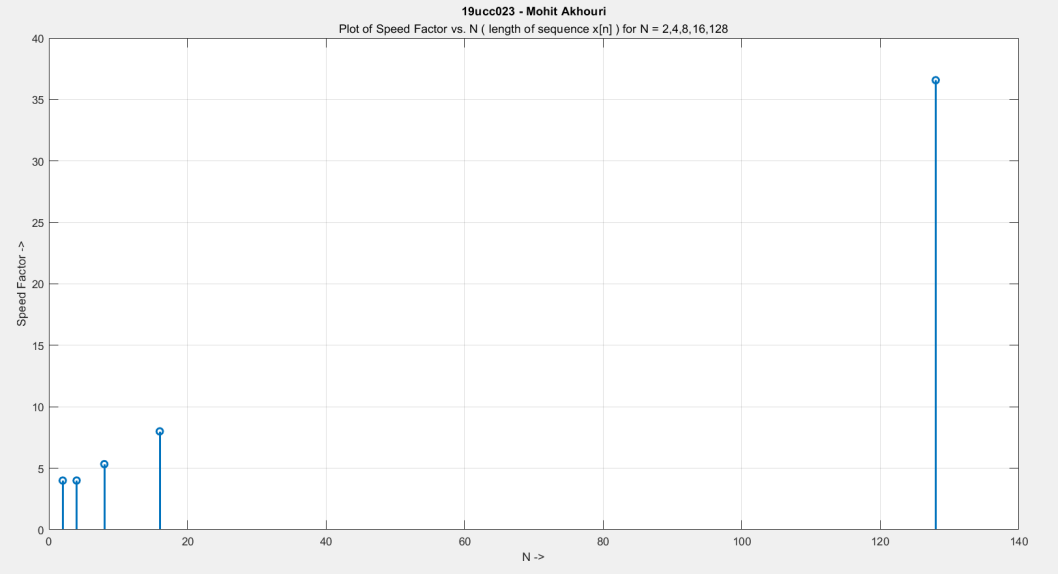


Figure 8.13 Plot of Speed Factor vs. N

8.4.5 Simulink based radix-2 dit fft algorithm :

```
% 19ucc023
% Mohit Akhouri
% Experiment 8 - Observation 5

% This code will make use of Simulink Model for calculation of radix-2
 fft
% of random sequences of length = 2,4 and 8. This code will also
 compare
% the fft calculated via inbuilt function fft with that calculated via
% Simulink Model

sim('Simulink_Observation_5'); % calling the simulink model for
 calculation of N-point fft

xn_2 = [1 2]; % random seqeunce x[n] of length = 2
xn_4 = [3 2 1 4]; % random seqeunce x[n] of length = 4
xn_8 = [5 1 8 7 6 2 3 4]; % random seqeunce x[n] of length = 8

fft_inb_2 = fft(xn_2,2); % 2-point fft of x[n] using INBUILT function
 fft
fft_inb_4 = fft(xn_4,4); % 4-point fft of x[n] using INBUILT function
 fft
fft_inb_8 = fft(xn_8,8); % 8-point fft of x[n] using INBUILT function
 fft

% Plot of input seqeunce x[n] of length = 2
figure;
stem(xn_2,'Linewidth',1.8);
xlabel('samples(n) ->');
ylabel('x[n] ->');
title('19ucc023 - Mohit Akhouri','Plot of Input sequence x[n] of
 length=2');
grid on;

% Plots of 2-point DFT (via SIMULINK MODEL and via INBUILT FUNCTION
 fft)
figure;
subplot(2,1,1);
stem(out.fft_2point.data,'Linewidth',1.8);
xlabel('samples(k) ->');
ylabel('X(k) ->');
title('2-point DFT of sequence x[n] using SIMULINK MODEL');
grid on;
subplot(2,1,2);
stem(fft_inb_2,'Linewidth',1.8);
xlabel('samples(k) ->');
ylabel('X(k) ->');
title('2-point DFT of sequence x[n] using INBUILT FUNCTION fft');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');

% Plot of input seqeunce x[n] of length = 4
```

Figure 8.14 Part 1 of the code for observation 5

```
figure;
stem(xn_4,'Linewidth',1.8);
xlabel('samples(n) ->');
ylabel('x[n] ->');
title('19ucc023 - Mohit Akhouri','Plot of Input sequence x[n] of
 length=4');
grid on;

% Plots of 4-point DFT (via SIMULINK MODEL and via INBUILT FUNCTION
 fft)
figure;
subplot(2,1,1);
stem(out.fft_4point.data,'Linewidth',1.8);
xlabel('samples(k) ->');
ylabel('X(k) ->');
title('4-point DFT of sequence x[n] using SIMULINK MODEL');
grid on;
subplot(2,1,2);
stem(fft_inb_4,'Linewidth',1.8);
xlabel('samples(k) ->');
ylabel('X(k) ->');
title('4-point DFT of sequence x[n] using INBUILT FUNCTION fft');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');

% Plot of input seqeunce x[n] of length = 8
figure;
stem(xn_8,'Linewidth',1.8);
xlabel('samples(n) ->');
ylabel('x[n] ->');
title('19ucc023 - Mohit Akhouri','Plot of Input sequence x[n] of
 length=8');
grid on;

% Plots of 8-point DFT (via SIMULINK MODEL and via INBUILT FUNCTION
 fft)
figure;
subplot(2,1,1);
stem(out.fft_8point.data,'Linewidth',1.8);
xlabel('samples(k) ->');
ylabel('X(k) ->');
title('8-point DFT of sequence x[n] using SIMULINK MODEL');
grid on;
subplot(2,1,2);
stem(fft_inb_8,'Linewidth',1.8);
xlabel('samples(k) ->');
ylabel('X(k) ->');
title('8-point DFT of sequence x[n] using INBUILT FUNCTION fft');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');
```

Figure 8.15 Part 2 of the code for observation 5

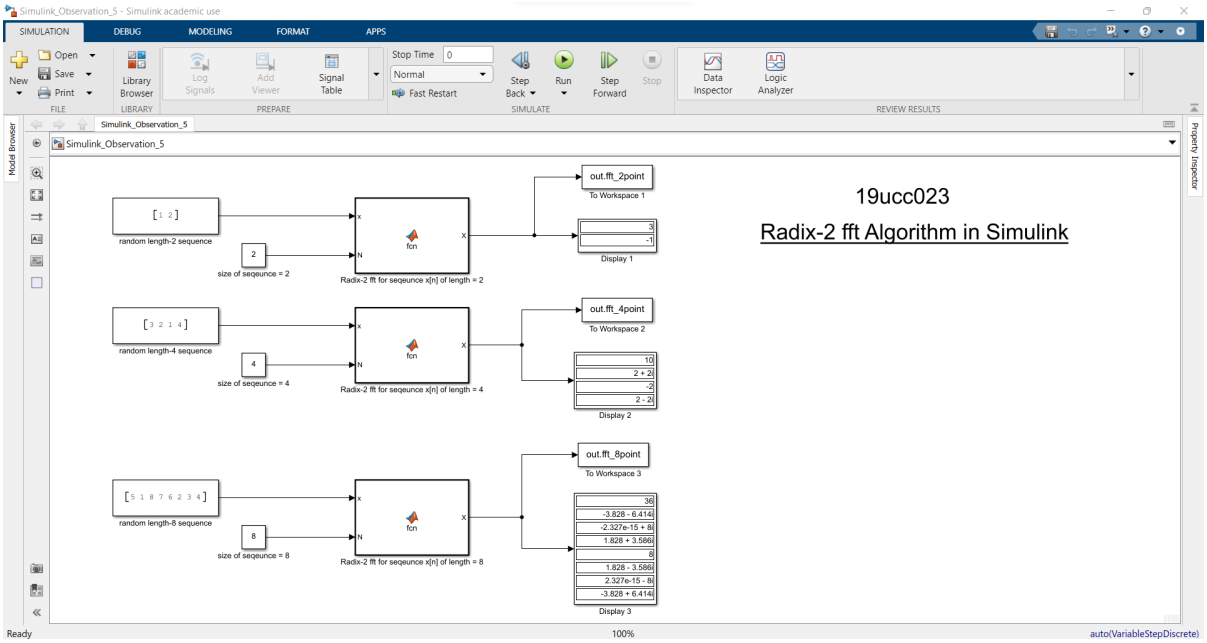


Figure 8.16 Simulink model used for the implementation of radix-2 dit fft algorithm

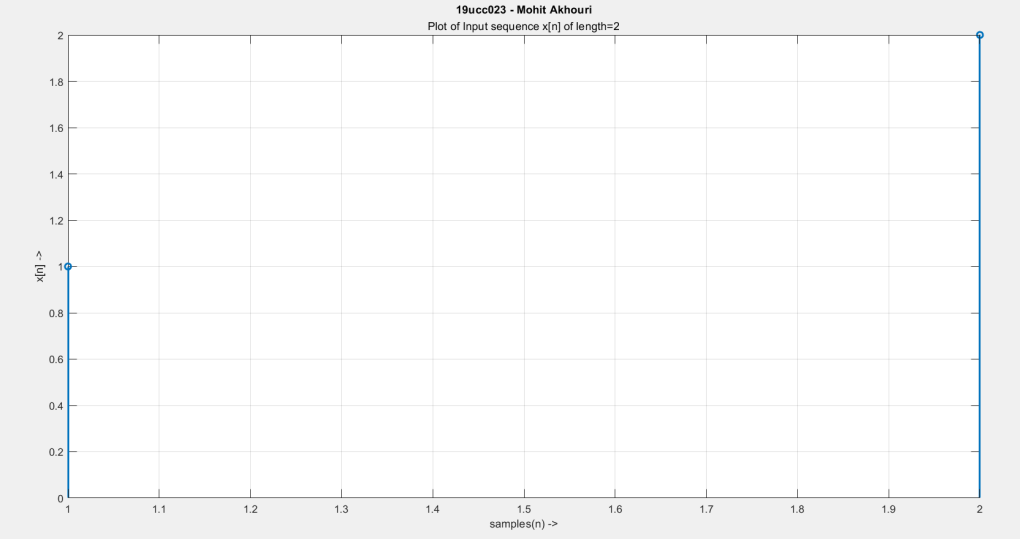


Figure 8.17 Plot of RANDOM Input Sequence of length = 2

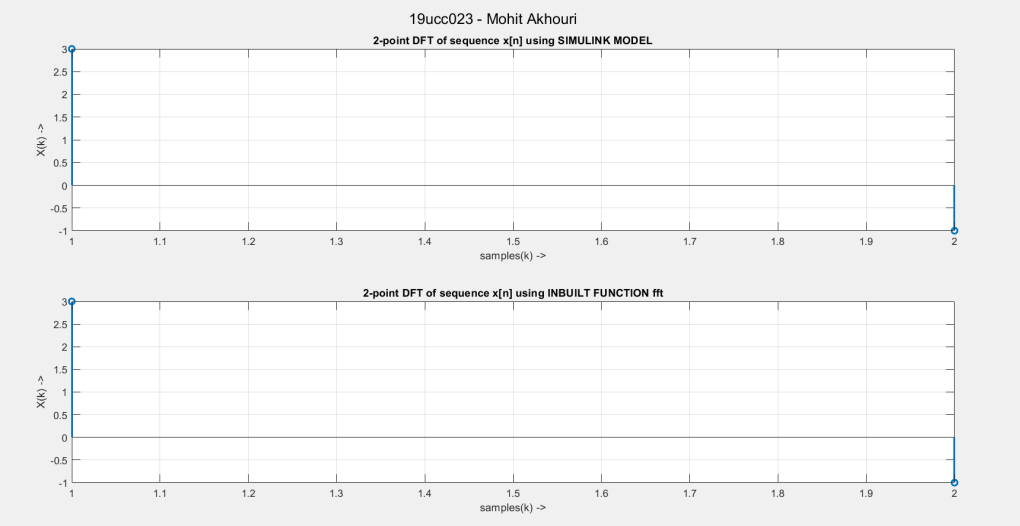


Figure 8.18 Plots of the DFT of input x[n] via both SIMULINK MODEL and INBUILT function

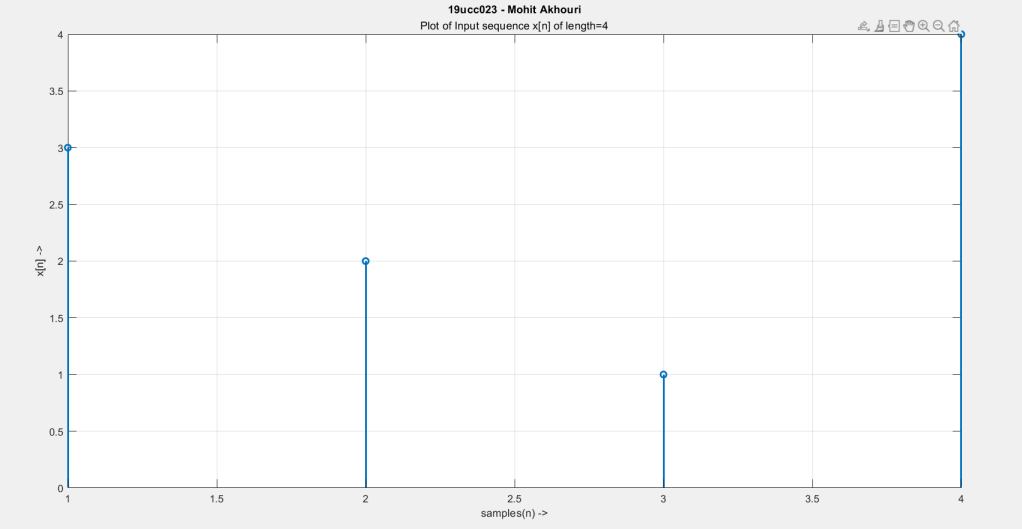


Figure 8.19 Plot of RANDOM Input Sequence of length = 4

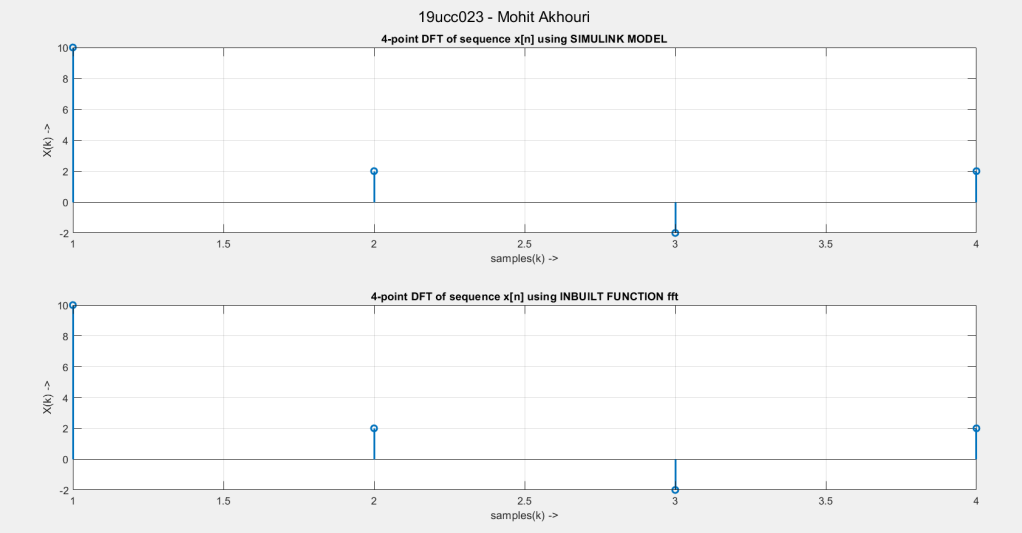


Figure 8.20 Plots of the DFT of input x[n] via both SIMULINK MODEL and INBUILT function

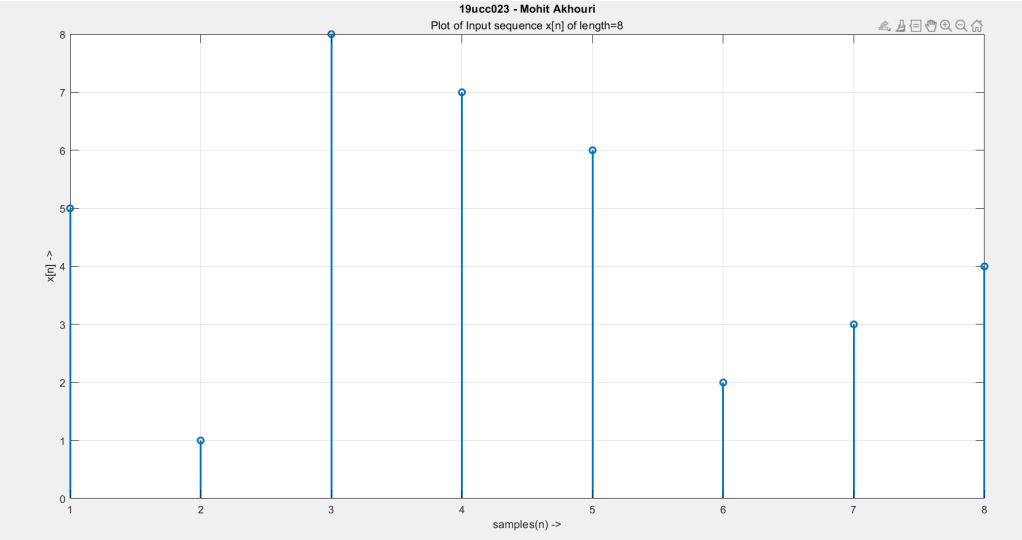


Figure 8.21 Plot of RANDOM Input Sequence of length = 8

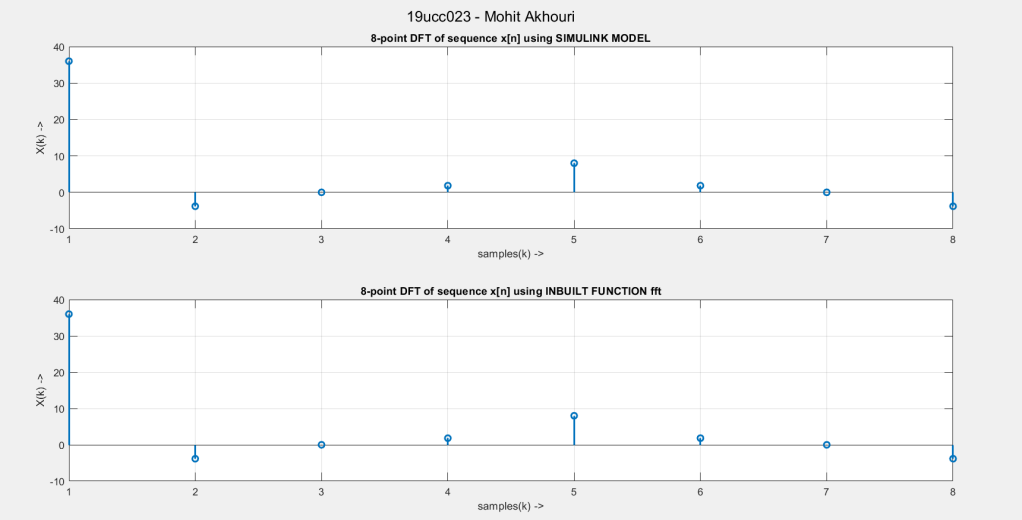


Figure 8.22 Plots of the DFT of input x[n] via both SIMULINK MODEL and INBUILT function

8.4.6 Function used in main codes implementation of radix-2 dit fft algorithm :

8.4.6.1 my_dit_fft.m function code :

```
function [X] = my_dit_fft(x,N)

% 19ucc023
% Mohit Akhouri

% ALGORITHM : This function will calculate the Radix-2 dit
 ( decimation in
% time ) fft for a sequence x[n]
% It divides the x[n] into seqeunce of EVEN and ODD seqeunces and
 calculate
% their N/2 point DFT separately. Lastly , it combines the calculated
 DFT
% to get the final answer of DFT.

X = zeros(1,N); % Output variable to store the calculated DFT
f1 = zeros(1,N/2); % To store the ODD parts of sequence x[n]
f2 = zeros(1,N/2); % To store the EVEN parts of sequence x[n]
Wn = exp(-1j*2*pi/N); % Twiddle factor used in the calculation of DFT

% Main loop algorithm for calculation of DFT via Radix-2 FFT algorithm
for k=1:N/2
    for m=1:N/2
        f1(k) = f1(k) + (x(2*(m-1)+1)*(Wn^(2*(m-1)*(k-1)))); %
 Calculation of N/2 point-DFT of ODD parts of seqeunce x[n]
        f2(k) = f2(k) + (x(2*(m-1)+2)*(Wn^(2*(m-1)*(k-1)))); %
 Calculation of N/2 point-DFT of EVEN parts of seqeunce x[n]
    end
    X(k)     = f1(k) + (f2(k)*(Wn^(k-1))); % Combination of f1(k) and
 f2(k) for N-point DFT of the first half of seqeunce x[n]
    X(k+N/2) = f1(k) - (f2(k)*(Wn^(k-1))); % Combination of f1(k) and
 f2(k) for N-point DFT of the second half of seqeunce x[n]
end

end
```

Figure 8.23 my_dit_fft.m function code to calculate the N-point DFT of a input sequence x[n]

8.5 Conclusion

In this experiment , we learnt the concepts of **Discrete Fourier Transform (DFT)** , **Fast fourier transform (FFT)** and **Radix-2 dit fft algorithm** of Digital Signal Processing. We learnt about the dit-fft algorithm to calculate the DFT of a input sequence x[n] faster. We learnt about two types of fft algorithm - **decimation in frequency (dif-fft)** and **decimation in time (dit-fft)**. We implemented the dit-fft algorithm in MATLAB by splitting the input sequence into **even** and **odd** parts. We also learnt about the **speed factor** for different values of length of input sequence (**N**) and also plotted the graph between them. We observed that speed factor **increases** as we move towards **higher N**. We also learnt about the **butterfly diagram** to efficiently calculate the FFT. We also implemented the algorithm in Simulink and compared the results obtained via MATLAB coding.

Chapter *9*

Experiment - 9

9.1 Aim of the Experiment

- Digital Filter Design

9.2 Software Used

- MATLAB
- Simulink

9.3 Theory

9.3.1 About Window function :

In signal processing and statistics, a window function (also known as an **apodization function** or **tapering function**) is a mathematical function that is zero-valued outside of some chosen interval, normally symmetric around the middle of the interval, usually near a maximum in the middle, and usually tapering away from the middle. Mathematically, when another function or waveform/data-sequence is ”multiplied” by a window function, the product is also zero-valued outside the interval: all that is left is the part where they overlap, the ”view through the window”. Equivalently, and in actual practice, the segment of data within the window is first isolated, and then only that data is multiplied by the window function values. Thus, tapering, not segmentation, is the main purpose of window functions.

In typical applications, the window functions used are non-negative, smooth, **bell-shaped curves**. Rectangle, triangle, and other functions can also be used. A more general definition of window functions does not require them to be identically zero outside an interval, as long as the product of the window multiplied by its argument is square integrable, and, more specifically, that the function goes sufficiently rapidly toward zero.

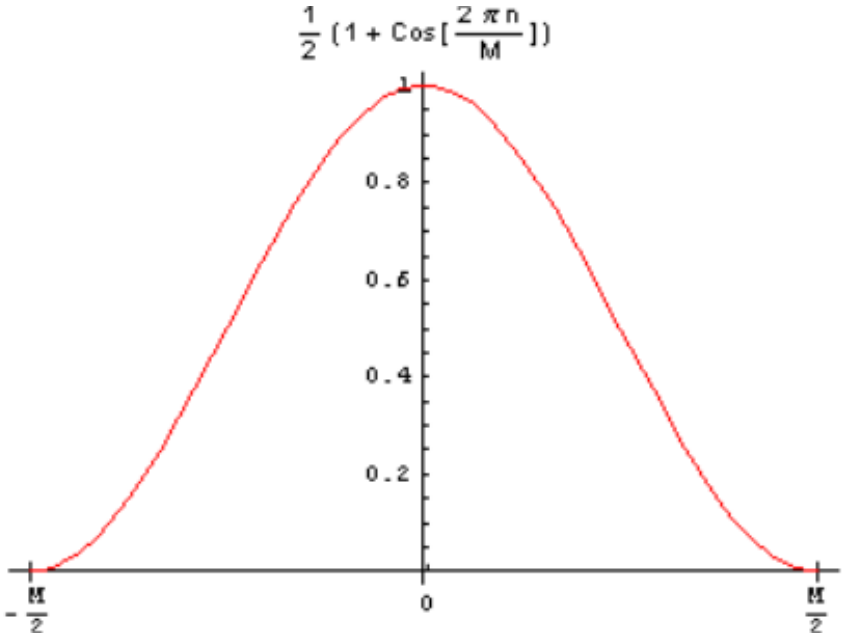


Figure 9.1 Hanning Window function

Windows are sometimes used in the **design of digital filters**, in particular to convert an "ideal" impulse response of infinite duration, such as a sinc function, to a finite impulse response (FIR) filter design. That is called the **window method**.

9.3.2 Types of Window functions :

9.3.2.1 About Rectangular Window :

The rectangular window (sometimes known as the **boxcar** or **Dirichlet window**) is the simplest window, equivalent to replacing all but N values of a data sequence by zeros, making it appear as though the waveform suddenly turns on and off. The window function $w(n)$ is given as :

$$w(n) = 1 \tag{9.1}$$

The rectangular window is the **1st order B-spline window** as well as the **0th power power-of-sine window**.

9.3.2.2 About B-spline Window :

B-spline windows can be obtained as k-fold convolutions of the rectangular window. They include the **rectangular window** itself (k = 1), the **Triangular window** (k = 2) and the **Parzen window** (k = 4). Alternative definitions sample the appropriate normalized B-spline basis functions instead of convolving discrete-time windows. A k^{th}-order B-spline basis function is a piece-wise polynomial function of degree k-1 that is obtained by **k-fold self-convolution** of the rectangular function.

9.3.2.3 About Hamming Window :

The Hamming window was proposed by **Richard W. Hamming**. That choice places a zero-crossing at frequency $5\pi/(N-1)$, which cancels the first sidelobe of the Hann window, giving it a height of about one-fifth that of the Hann window. The Hamming window is often called the **Hamming blip** when used for **pulse shaping**. The window function $w(n)$ is given as :

$$w(n) = 0.42 - 0.5 * cos\left(\frac{2\pi n}{M-1}\right) + 0.08 * cos\left(\frac{4\pi n}{M-1}\right) \tag{9.2}$$

9.3.2.4 About Blackman Window :

The Blackman window is a **taper** formed by using the first three terms of a summation of cosines. It was designed to have close to the minimal leakage possible. It is close to optimal, only slightly worse than a **Kaiser window**. The window function $w(n)$ is given as :

$$w(n) = 0.54 - 0.46 * cos\left(\frac{2\pi n}{M-1}\right) \tag{9.3}$$

9.3.3 About Notch Filter :

A notch filter is a type of **band-stop filter**, which is a filter that attenuates frequencies within a specific range while passing all other frequencies unaltered. For a notch filter, **this range of frequencies is very narrow**.

The range of frequencies that a band-stop filter attenuates is called the **stopband**. The narrow stopband in a notch filter makes the frequency response resemble a deep notch, which gives the filter its name. It also means that notch filters have a **high Q factor**, which is the ratio of center frequency to bandwidth.

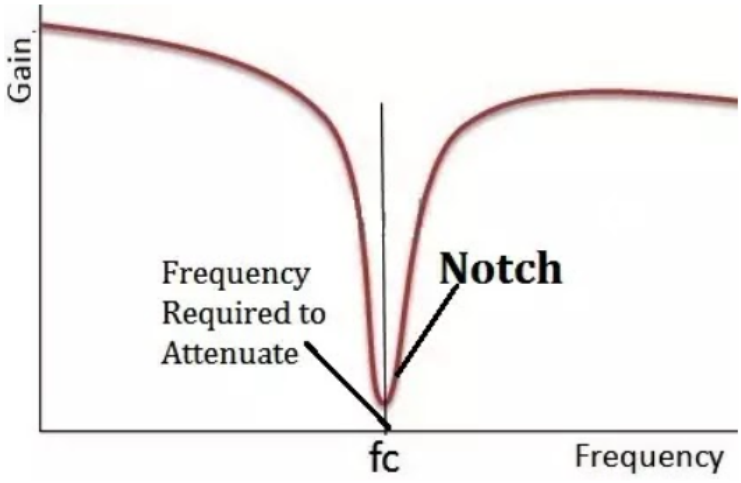


Figure 9.2 Frequency Response of Notch Filter

9.4 Code and results

9.4.1 Simulating various window based low pass FIR filter for different cutoff frequencies :

```
% 19ucc023
% Mohit Akhouri
% Experiment 9 - Observation 1a

% In this code , we will implement a low pass filter based on various
% windowing techniques - like rectangular , hamming and blackman. We
 design
% the low pass filter for various cutoff frequencies like pi/2,pi/4
 and
% pi/6. We plot the amplitude and phase response of the various low
 pass
% filter.

clc;
clear all;
close all;

M = 31; % delay in time domain

wc = pi/2; % cutoff frequency
hd = zeros(1,M); % to store the filter impulse response
w1 = zeros(1,M); % to store the window function for rectangular window
w2 = zeros(1,M); % to store the window function for hamming window
w3 = zeros(1,M); % to store the window function for blackman window

% main loop algorithm for calculation of window functions and filter
% impulse response for various cases
for n=0:M-1
    H = @(w) exp(-1j*w*(n-((M-1)/2))); % response of the low pass
 filter
    hd(n+1) = (integral(H,-wc,wc))/(2*pi); % filter impulse response
    w1(n+1) = 1; % window function for rectangular window
    w2(n+1) = 0.42 - 0.5*cos((2*pi*n)/(M-1)) + 0.08*cos((4*pi*n)/
(M-1)); % window function for hamming window
    w3(n+1) = 0.54 - 0.46*cos((2*pi*n)/(M-1)); % window function for
 blackman window
end

h1 = hd .* w1; % finite impulse response of rectangular window based
 low pass filter
h2 = hd .* w2; % finite impulse response of hamming window based low
 pass filter
h3 = hd .* w3; % finite impulse response of blackman window based low
 pass filter

% Plotting the magnitude and phase responses of various low pass
 filter
figure;
freqz(h1);
title('19ucc023 - Mohit Akhouri','Rectangular window for low pass
 filter for cutoff frequency = \pi/2');
grid on;
```

Figure 9.3 Part 1 of the Code for the observation 1(a)

```
figure;
freqz(h2);
title('19ucc023 - Mohit Akhouri','Hamming window for low pass filter
 for cutoff frequency = \pi/2');
grid on;

figure;
freqz(h3);
title('19ucc023 - Mohit Akhouri','Blackman window for low pass filter
 for cutoff frequency = \pi/2');
grid on;

wc = pi/4; % cutoff frequency
hd = zeros(1,M); % to store the filter impulse response
w1 = zeros(1,M); % to store the window function for rectangular window
w2 = zeros(1,M); % to store the window function for hamming window
w3 = zeros(1,M); % to store the window function for blackman window

% main loop algorithm for calculation of window functions and filter
% impulse response for various cases
for n=0:M-1
    H = @(w) exp(-1j*w*(n-((M-1)/2))); % response of the low pass
 filter
    hd(n+1) = (integral(H,-wc,wc))/(2*pi); % filter impulse response
    w1(n+1) = 1; % window function for rectangular window
    w2(n+1) = 0.42 - 0.5*cos((2*pi*n)/(M-1)) + 0.08*cos((4*pi*n)/
(M-1)); % window function for hamming window
    w3(n+1) = 0.54 - 0.46*cos((2*pi*n)/(M-1)); % window function for
 blackman window
end

h1 = hd .* w1; % finite impulse response of rectangular window based
 low pass filter
h2 = hd .* w2; % finite impulse response of hamming window based low
 pass filter
h3 = hd .* w3; % finite impulse response of blackman window based low
 pass filter

% Plotting the magnitude and phase responses of various low pass
 filter
figure;
freqz(h1);
title('19ucc023 - Mohit Akhouri','Rectangular window for low pass
 filter for cutoff frequency = \pi/4');
grid on;

figure;
freqz(h2);
title('19ucc023 - Mohit Akhouri','Hamming window for low pass filter
 for cutoff frequency = \pi/4');
grid on;

figure;
```

Figure 9.4 Part 2 of the Code for the observation 1(a)

```
freqz(h3);
title('19ucc023 - Mohit Akhouri','Blackman window for low pass filter
 for cutoff frequency = \pi/4');
grid on;

wc = pi/6; % cutoff frequency
hd = zeros(1,M); % to store the filter impulse response
w1 = zeros(1,M); % to store the window function for rectangular window
w2 = zeros(1,M); % to store the window function for hamming window
w3 = zeros(1,M); % to store the window function for blackman window

% main loop algorithm for calculation of window functions and filter
% impulse response for various cases
for n=0:M-1
    H = @(w) exp(-1j*w*(n-((M-1)/2))); % response of the low pass
 filter
    hd(n+1) = (integral(H,-wc,wc))/(2*pi); % filter impulse response
    w1(n+1) = 1; % window function for rectangular window
    w2(n+1) = 0.42 - 0.5*cos((2*pi*n)/(M-1)) + 0.08*cos((4*pi*n)/
(M-1)); % window function for hamming window
    w3(n+1) = 0.54 - 0.46*cos((2*pi*n)/(M-1)); % window function for
 blackman window
end

h1 = hd .* w1; % finite impulse response of rectangular window based
 low pass filter
h2 = hd .* w2; % finite impulse response of hamming window based low
 pass filter
h3 = hd .* w3; % finite impulse response of blackman window based low
 pass filter

% Plotting the magnitude and phase responses of various low pass
 filter
figure;
freqz(h1);
title('19ucc023 - Mohit Akhouri','Rectangular window for low pass
 filter for cutoff frequency = \pi/6');
grid on;

figure;
freqz(h2);
title('19ucc023 - Mohit Akhouri','Hamming window for low pass filter
 for cutoff frequency = \pi/6');
grid on;

figure;
freqz(h3);
title('19ucc023 - Mohit Akhouri','Blackman window for low pass filter
 for cutoff frequency = \pi/6');
grid on;
```

Figure 9.5 Part 3 of the Code for the observation 1(a)

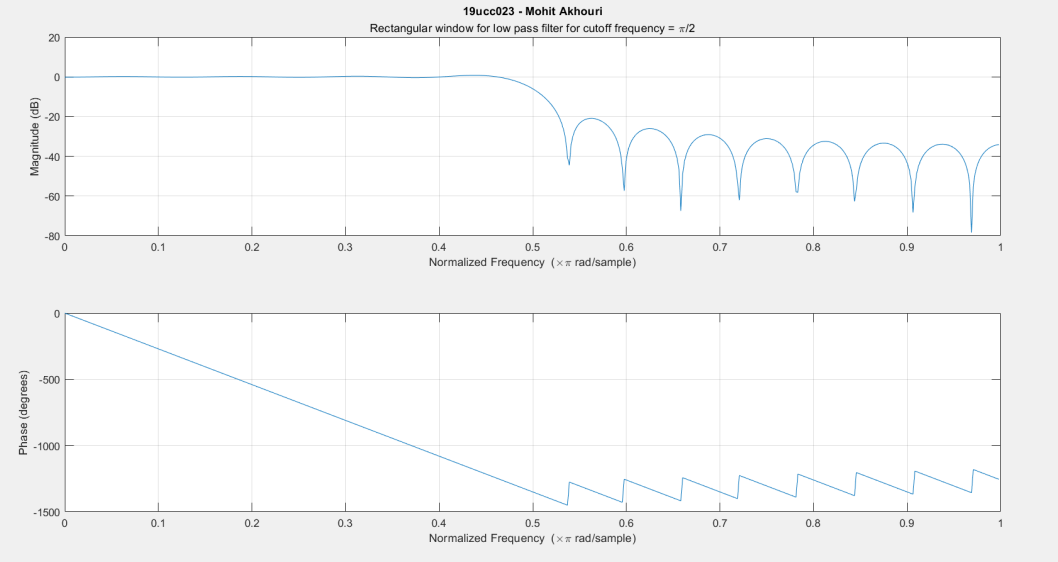


Figure 9.6 Plot of Rectangular window based Low pass filter for $w_c = \frac{\pi}{2}$

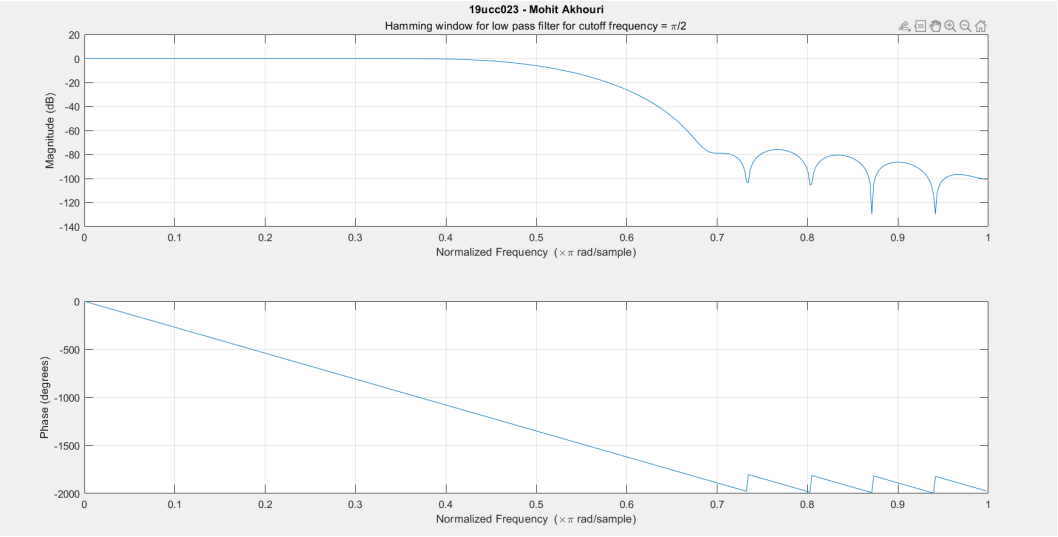


Figure 9.7 Plot of Hamming window based Low pass filter for $w_c = \frac{\pi}{2}$

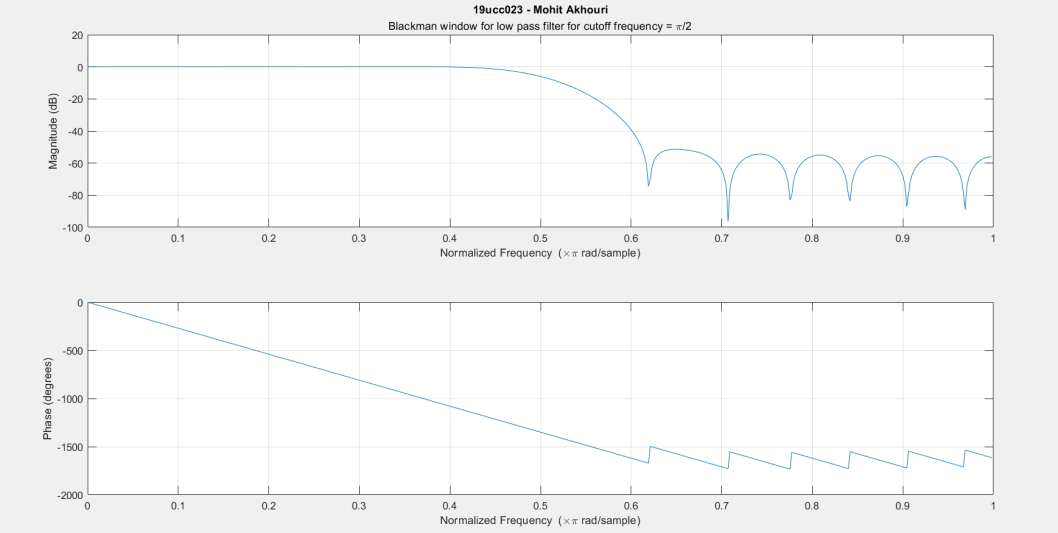


Figure 9.8 Plot of Blackman window based Low pass filter for $w_c = \frac{\pi}{2}$

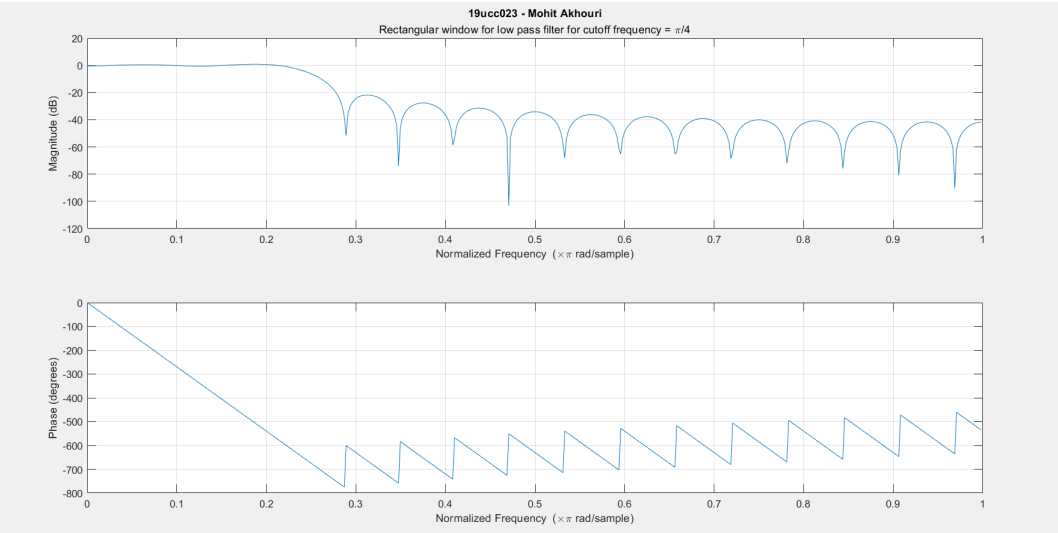


Figure 9.9 Plot of Rectangular window based Low pass filter for $w_c = \frac{\pi}{4}$

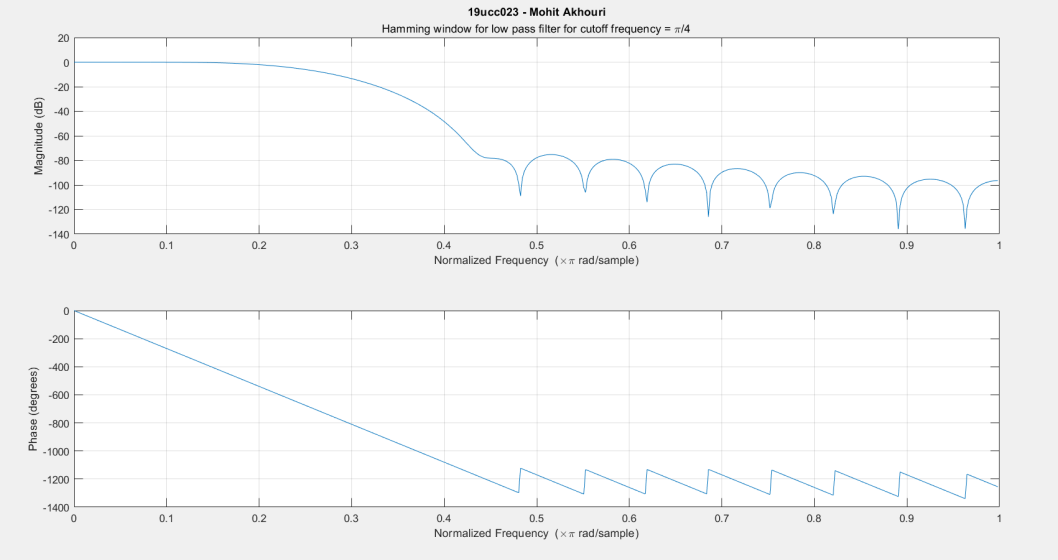


Figure 9.10 Plot of Hamming window based Low pass filter for $w_c = \frac{\pi}{4}$

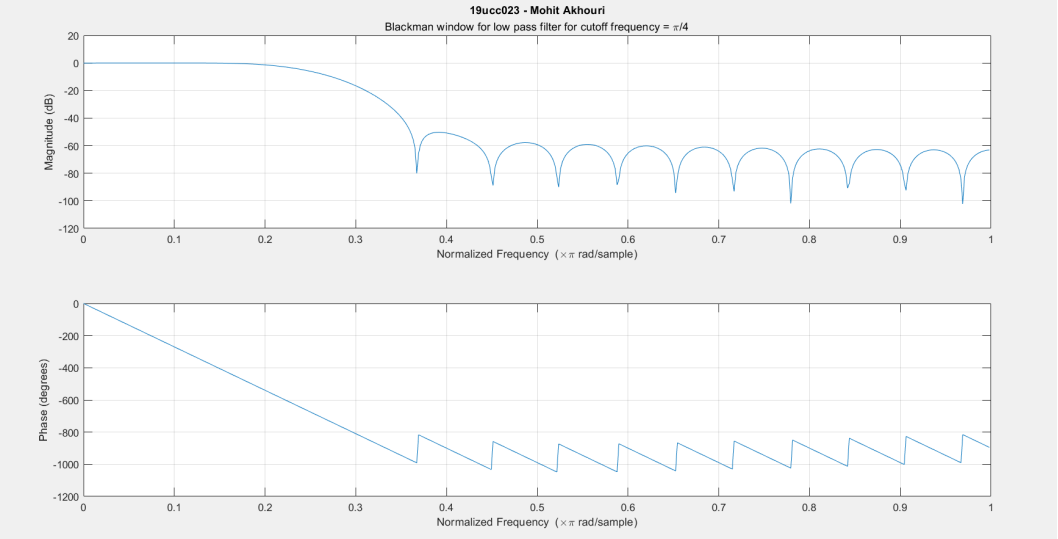


Figure 9.11 Plot of Blackman window based Low pass filter for $w_c = \frac{\pi}{4}$

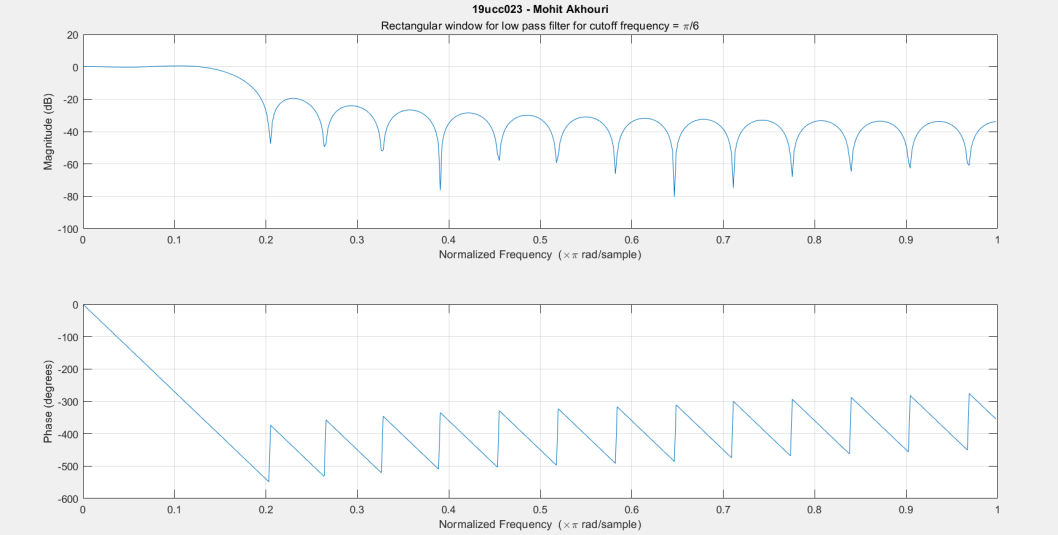


Figure 9.12 Plot of Rectangular window based Low pass filter for $w_c = \frac{\pi}{6}$

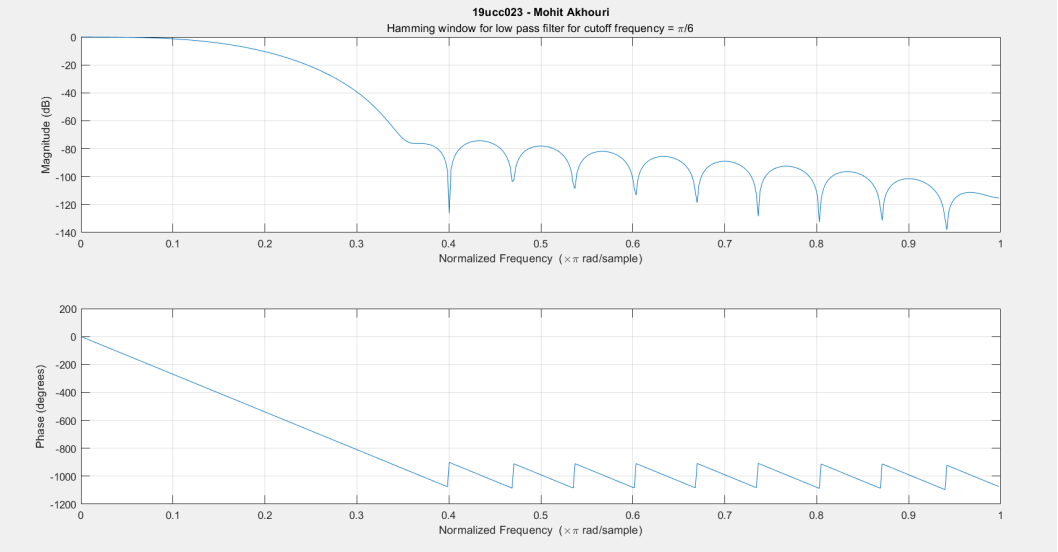


Figure 9.13 Plot of Hamming window based Low pass filter for $w_c = \frac{\pi}{6}$

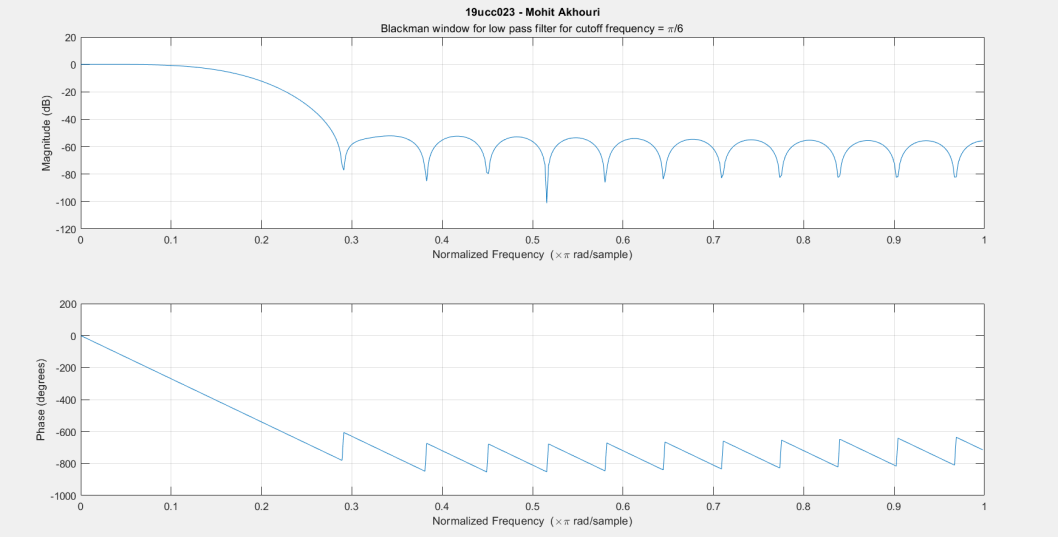


Figure 9.14 Plot of Blackman window based Low pass filter for $w_c = \frac{\pi}{6}$

9.4.2 Simulating various window based high pass FIR filter for different cutoff frequencies :

```
% 19ucc023
% Mohit Akhouri
% Experiment 9 - Observation 1b

% In this code , we will implement a high pass filter based on various
% windowing techniques - like rectangular , hamming and blackman. We
 design
% the high pass filter for various cutoff frequencies like pi/2,pi/4
 and
% pi/6. We plot the amplitude and phase response of the various high
 pass
% filter.

clc;
clear all;
close all;

M = 31; % delay in time domain

wc = pi/2; % cutoff frequency
hd = zeros(1,M); % to store the filter impulse response
w1 = zeros(1,M); % to store the window function for rectangular window
w2 = zeros(1,M); % to store the window function for hamming window
w3 = zeros(1,M); % to store the window function for blackman window

% main loop algorithm for calculation of window functions and filter
% impulse response for various cases
for n=0:M-1
    H = @(w) exp(-1j*w*(n-((M-1)/2))); % response of the high pass
 filter
    hd(n+1) = (integral(H,-pi,-wc) + integral(H,wc,pi))/(2*pi); %
 filter impulse response
    w1(n+1) = 1; % window function for rectangular window
    w2(n+1) = 0.42 - 0.5*cos((2*pi*n)/(M-1)) + 0.08*cos((4*pi*n)/
(M-1)); % window function for hamming window
    w3(n+1) = 0.54 - 0.46*cos((2*pi*n)/(M-1)); % window function for
 blackman window
end

h1 = hd .* w1; % finite impulse response of rectangular window based
 high pass filter
h2 = hd .* w2; % finite impulse response of hamming window based high
 pass filter
h3 = hd .* w3; % finite impulse response of blackman window based high
 pass filter

% Plotting the magnitude and phase responses of various high pass
 filter
figure;
freqz(h1);
title('19ucc023 - Mohit Akhouri','Rectangular window for high pass
 filter for cutoff frequency = \pi/2');
```

Figure 9.15 Part 1 of the Code for the observation 1(b)

```
grid on;

figure;
freqz(h2);
title('19ucc023 - Mohit Akhouri','Hamming window for high pass filter
 for cutoff frequency = \pi/2');
grid on;

figure;
freqz(h3);
title('19ucc023 - Mohit Akhouri','Blackman window for high pass filter
 for cutoff frequency = \pi/2');
grid on;

wc = pi/4; % cutoff frequency
hd = zeros(1,M); % to store the filter impulse response
w1 = zeros(1,M); % to store the window function for rectangular window
w2 = zeros(1,M); % to store the window function for hamming window
w3 = zeros(1,M); % to store the window function for blackman window

% main loop algorithm for calculation of window functions and filter
% impulse response for various cases
for n=0:M-1
    H = @(w) exp(-1j*w*(n-((M-1)/2))); % response of the high pass
 filter
    hd(n+1) = (integral(H,-pi,-wc) + integral(H,wc,pi))/(2*pi); %
 filter impulse response
    w1(n+1) = 1; % window function for rectangular window
    w2(n+1) = 0.42 - 0.5*cos((2*pi*n)/(M-1)) + 0.08*cos((4*pi*n)/
(M-1)); % window function for hamming window
    w3(n+1) = 0.54 - 0.46*cos((2*pi*n)/(M-1)); % window function for
 blackman window
end

h1 = hd .* w1; % finite impulse response of rectangular window based
 high pass filter
h2 = hd .* w2; % finite impulse response of hamming window based high
 pass filter
h3 = hd .* w3; % finite impulse response of blackman window based high
 pass filter

% Plotting the magnitude and phase responses of various high pass
 filter
figure;
freqz(h1);
title('19ucc023 - Mohit Akhouri','Rectangular window for high pass
 filter for cutoff frequency = \pi/4');
grid on;

figure;
freqz(h2);
title('19ucc023 - Mohit Akhouri','Hamming window for high pass filter
 for cutoff frequency = \pi/4');
grid on;
```

Figure 9.16 Part 2 of the Code for the observation 1(b)

```
figure;
freqz(h3);
title('19ucc023 - Mohit Akhouri','Blackman window for high pass filter
 for cutoff frequency = \pi/4');
grid on;

wc = pi/6; % cutoff frequency
hd = zeros(1,M); % to store the filter impulse response
w1 = zeros(1,M); % to store the window function for rectangular window
w2 = zeros(1,M); % to store the window function for hamming window
w3 = zeros(1,M); % to store the window function for blackman window

% main loop algorithm for calculation of window functions and filter
% impulse response for various cases
for n=0:M-1
    H = @(w) exp(-1j*w*(n-((M-1)/2))); % response of the high pass
 filter
    hd(n+1) = (integral(H,-pi,-wc) + integral(H,wc,pi))/(2*pi); %
 filter impulse response
    w1(n+1) = 1; % window function for rectangular window
    w2(n+1) = 0.42 - 0.5*cos((2*pi*n)/(M-1)) + 0.08*cos((4*pi*n)/
(M-1)); % window function for hamming window
    w3(n+1) = 0.54 - 0.46*cos((2*pi*n)/(M-1)); % window function for
 blackman window
end

h1 = hd .* w1; % finite impulse response of rectangular window based
 high pass filter
h2 = hd .* w2; % finite impulse response of hamming window based high
 pass filter
h3 = hd .* w3; % finite impulse response of blackman window based high
 pass filter

% Plotting the magnitude and phase responses of various high pass
 filter
figure;
freqz(h1);
title('19ucc023 - Mohit Akhouri','Rectangular window for high pass
 filter for cutoff frequency = \pi/6');
grid on;

figure;
freqz(h2);
title('19ucc023 - Mohit Akhouri','Hamming window for high pass filter
 for cutoff frequency = \pi/6');
grid on;

figure;
freqz(h3);
title('19ucc023 - Mohit Akhouri','Blackman window for high pass filter
 for cutoff frequency = \pi/6');
grid on;
```

Figure 9.17 Part 3 of the Code for the observation 1(b)

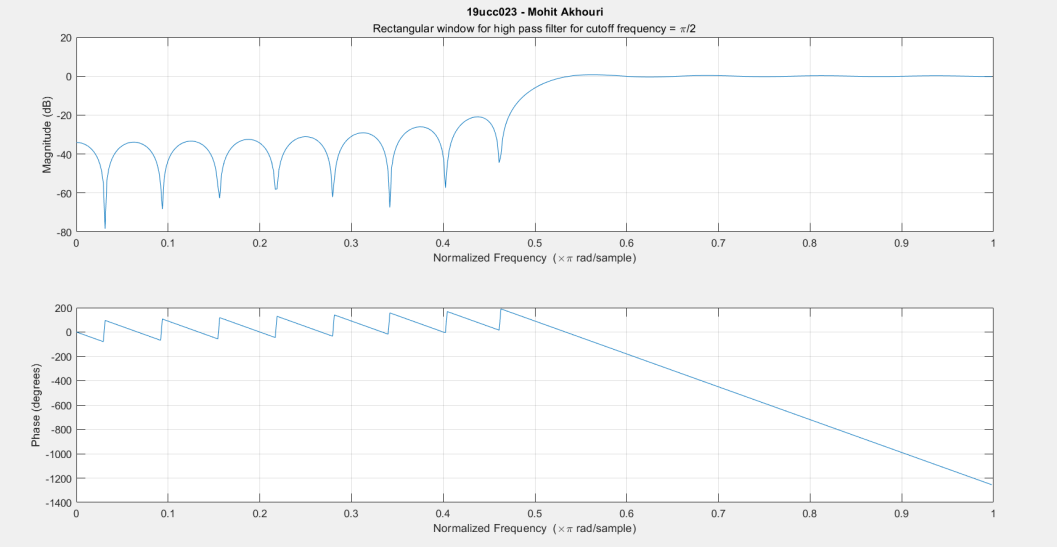


Figure 9.18 Plot of Rectangular window based High pass filter for $w_c = \frac{\pi}{2}$

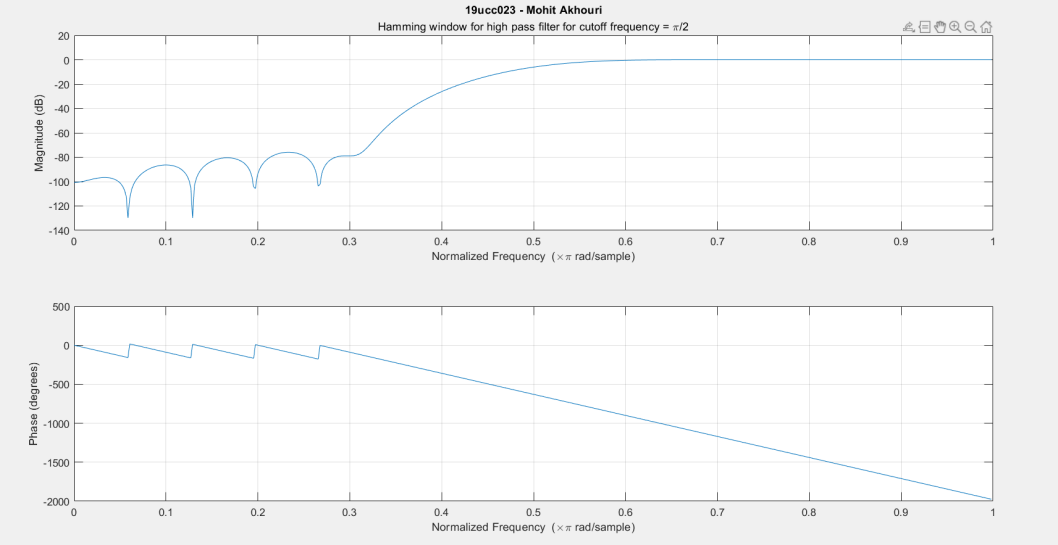


Figure 9.19 Plot of Hamming window based High pass filter for $w_c = \frac{\pi}{2}$

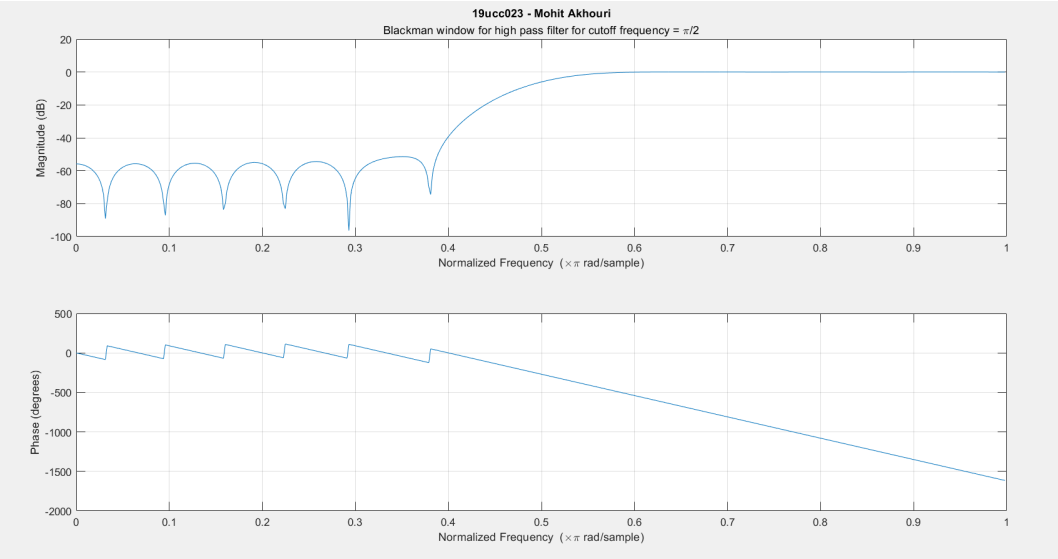


Figure 9.20 Plot of Blackman window based High pass filter for $w_c = \frac{\pi}{2}$

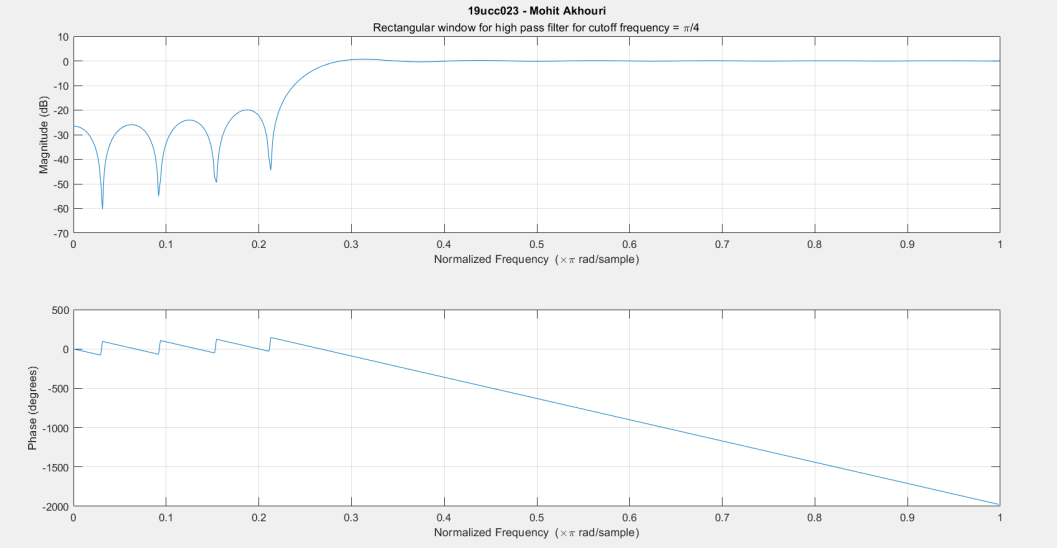


Figure 9.21 Plot of Rectangular window based High pass filter for $w_c = \frac{\pi}{4}$

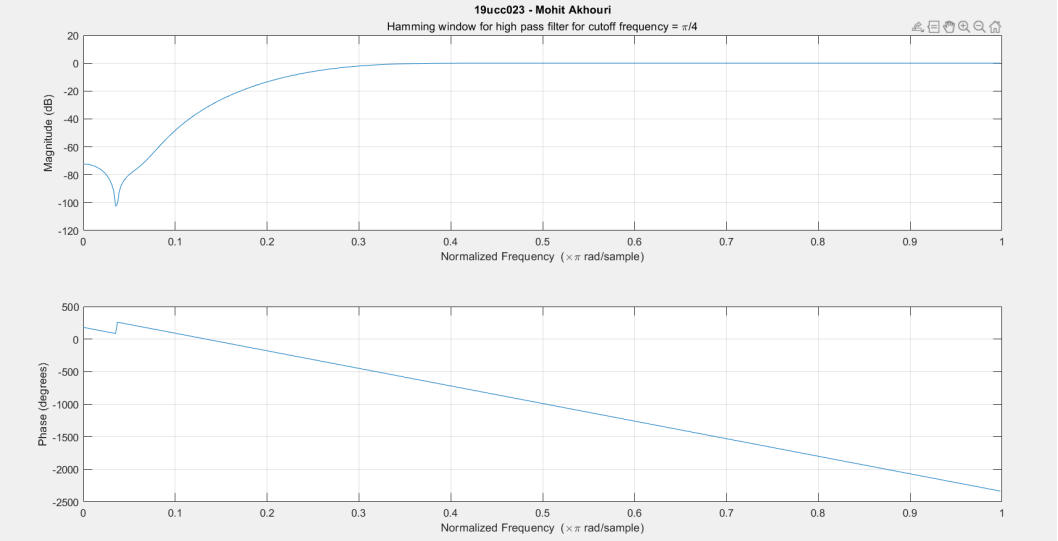


Figure 9.22 Plot of Hamming window based High pass filter for $w_c = \frac{\pi}{4}$

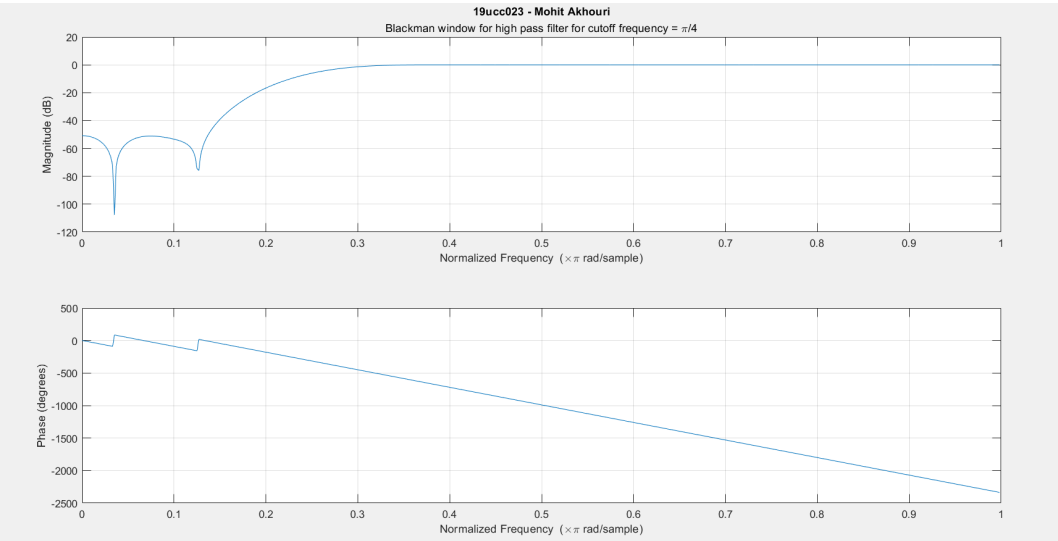


Figure 9.23 Plot of Blackman window based High pass filter for $w_c = \frac{\pi}{4}$

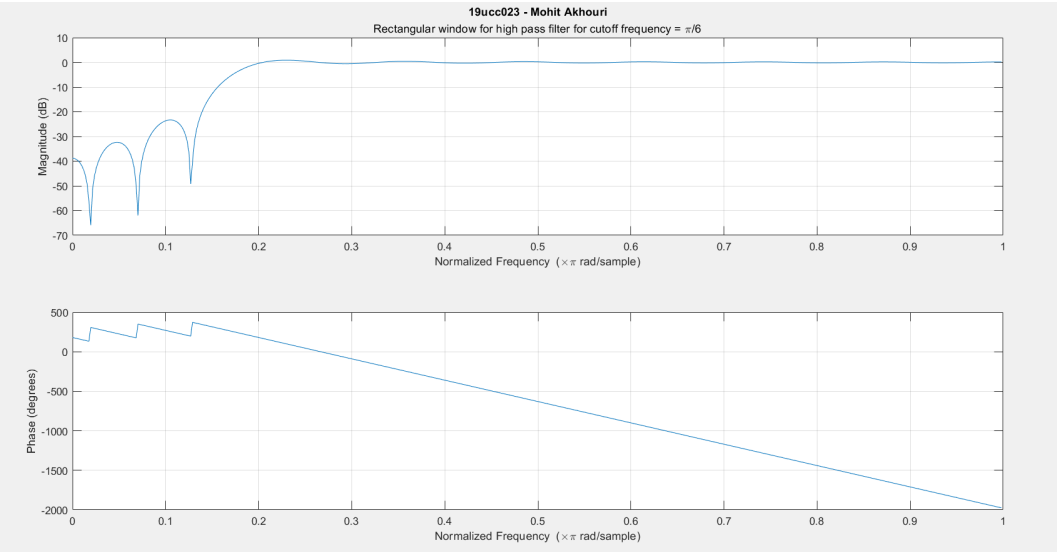


Figure 9.24 Plot of Rectangular window based High pass filter for $w_c = \frac{\pi}{6}$

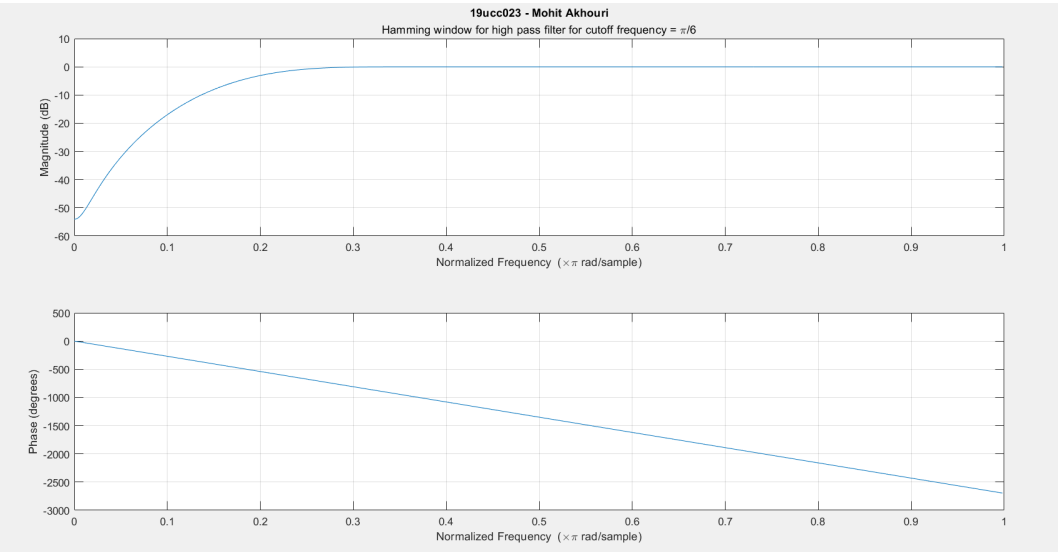


Figure 9.25 Plot of Hamming window based High pass filter for $w_c = \frac{\pi}{6}$

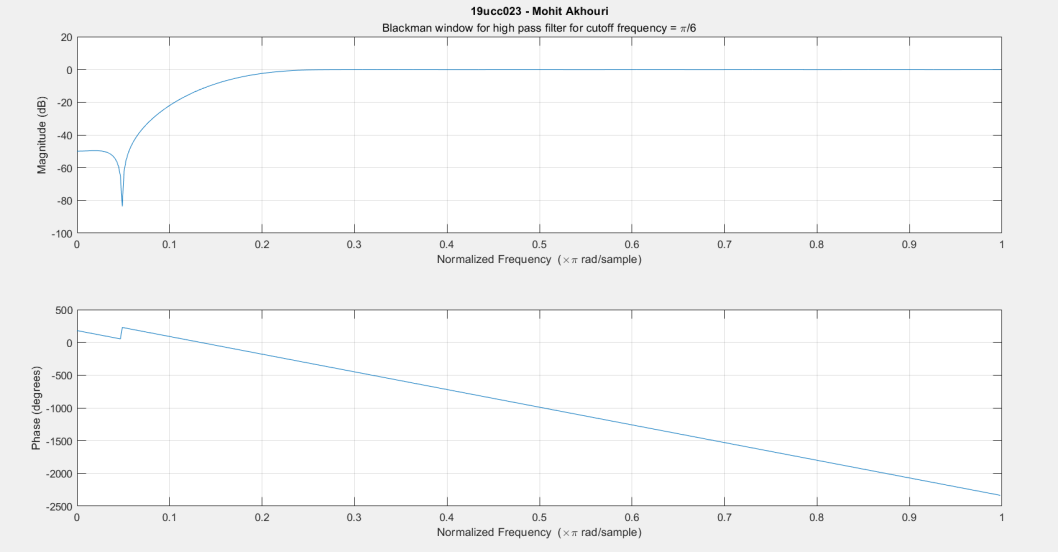


Figure 9.26 Plot of Blackman window based High pass filter for $w_c = \frac{\pi}{6}$

9.4.3 Simulating various window based band pass FIR filter for different cutoff frequencies :

```
% 19ucc023
% Mohit Akhouri
% Experiment 9 - Observation 1c

% In this code , we will implement a band pass filter based on various
% windowing techniques - like rectangular , hamming and blackman. We
 design
% band pass filter for cutoff frequencies - wc(low) = pi/4 and
 wc(high) =
% pi/2. We also plot the magnitude and phase spectrum of the designed
% filter.

clc;
clear all;
close all;

M = 31; % delay in time domain

wc_low = pi/4; % low frequency cutoff for band pass filter
wc_high = pi/2; % high frequency cutoff for band pass filter

hd = zeros(1,M); % to store the filter impulse response
w1 = zeros(1,M); % to store the window function for rectangular window
w2 = zeros(1,M); % to store the window function for hamming window
w3 = zeros(1,M); % to store the window function for blackman window

% main loop algorithm for calculation of window functions and filter
% impulse response for various cases
for n=0:M-1
    H = @(w) exp(-1j*w*(n-((M-1)/2))); % response of the band pass
 filter
    hd(n+1) = (integral(H,-wc_high,-wc_low) +
 integral(H,wc_low,wc_high))/(2*pi); % filter impulse response
    w1(n+1) = 1; % window function for rectangular window
    w2(n+1) = 0.42 - 0.5*cos((2*pi*n)/(M-1)) + 0.08*cos((4*pi*n)/
(M-1)); % window function for hamming window
    w3(n+1) = 0.54 - 0.46*cos((2*pi*n)/(M-1)); % window function for
 blackman window
end

h1 = hd .* w1; % finite impulse response of rectangular window based
 band pass filter
h2 = hd .* w2; % finite impulse response of hamming window based band
 pass filter
h3 = hd .* w3; % finite impulse response of blackman window based band
 pass filter

% Plotting the magnitude and phase responses of various band pass
 filter
figure;
freqz(h1);
```

Figure 9.27 Part 1 of the Code for the observation 1(c)

```
title('19ucc023 - Mohit Akhouri','Rectangular window for band pass
 filter for cutoff frequency as w_{c}(low) = \pi/4 and w_{c}(high) =
 \pi/2');
grid on;

figure;
freqz(h2);
title('19ucc023 - Mohit Akhouri','Hamming window for band pass filter
 for cutoff frequency as w_{c}(low) = \pi/4 and w_{c}(high) = \pi/2');
grid on;

figure;
freqz(h3);
title('19ucc023 - Mohit Akhouri','Blackman window for band pass filter
 for cutoff frequency as w_{c}(low) = \pi/4 and w_{c}(high) = \pi/2');
grid on;
```

Published with MATLAB® R2020b

Figure 9.28 Part 2 of the Code for the observation 1(c)

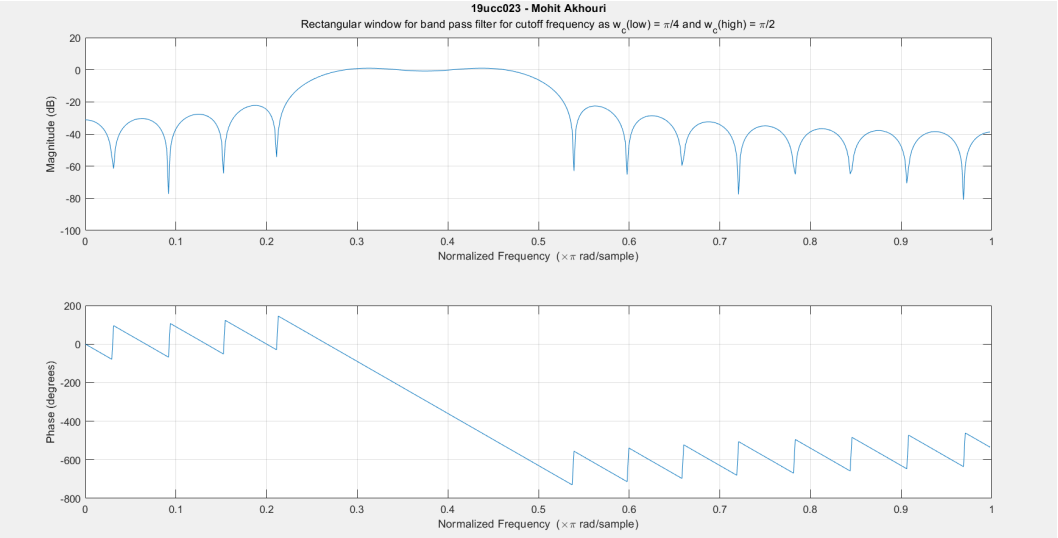


Figure 9.29 Plot of Rectangular window based Band pass filter for w_c(low) = $\frac{\pi}{4}$ and w_c(high) = $\frac{\pi}{2}$

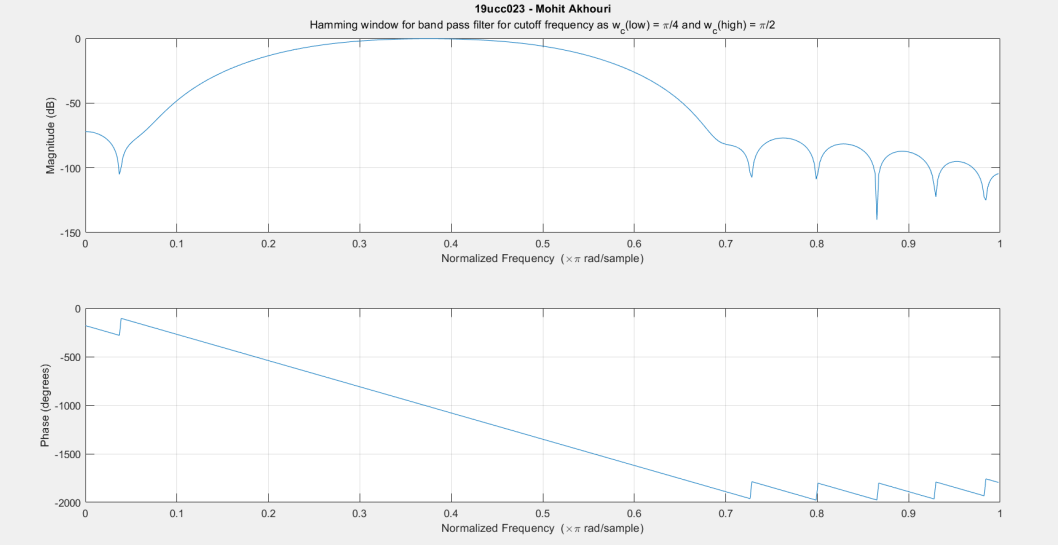


Figure 9.30 Plot of Hamming window based Band pass filter for w_c(low) = $\frac{\pi}{4}$ and w_c(high) = $\frac{\pi}{2}$

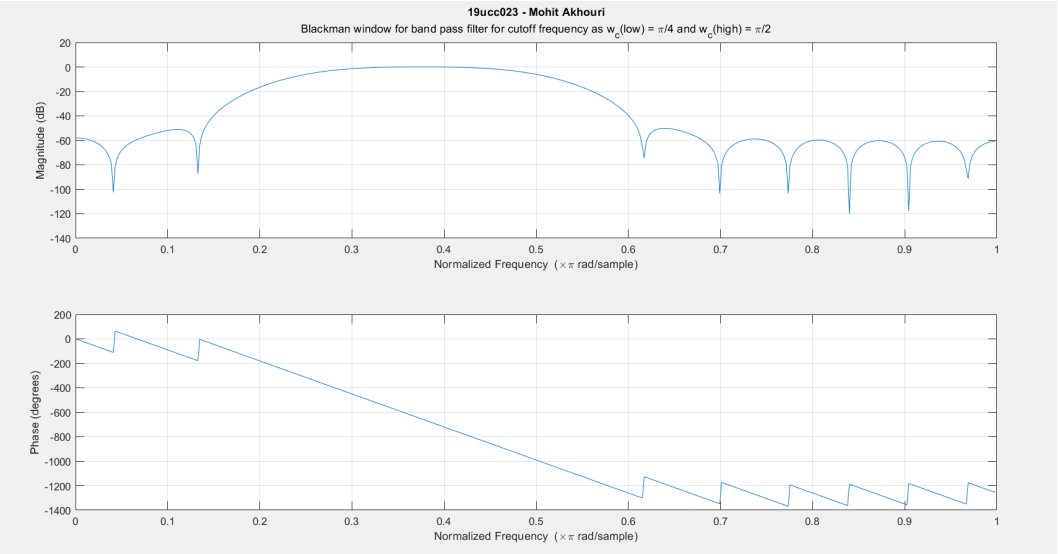


Figure 9.31 Plot of Blackman window based Band pass filter for w_c(low) = $\frac{\pi}{4}$ and w_c(high) = $\frac{\pi}{2}$

9.4.4 Removing the hissing sound using ideal low pass filter :

```
% 19ucc023
% Mohit Akhouri
% Experiment 9 - Observation 2

% This code will take the input of an audio signal with hissing
 component
% frequency greater than pi/2. Next it will design a low pass filter
 with
% frequency spectrum as : value '1' for all samples except for 66th
 and
% 192th samples , i.e. we design a IDEAL LOW PASS FILTER

% Finally we will divide the audio file into chunks of 1x256 and find
 the
% DFT for each of the chunk. Let it be X(w) . Finally we multiply the
% filter impulse response H(w) with X(w) to get the smoothened version
 of
% audio signal with hissing sound removed.

[x,fs] = audioread('inputwithhissgreaterthanpiby2.wav'); % reading an
 audio file
x = x'; % Taking the transpose to convert from Nx1 to 1xN for easy
 multiplication later on
x_length = length(x); % Finding length of audio signal x[n]

% plot of the original audio signal x[n] + unwanted components (hiss
 sound)
figure;
plot(x);
xlabel('samples(n) ->');
ylabel('x[n] ->');
title('19ucc023 - Mohit Akhouri','Plot of original audio signal x[n] +
 hissing frequency components');
grid on;

samples = 256; % variable to store the sample number for partition of
 audio file
hw = ones(1,256); % ideal low pass filter impulse response

% making the 66th and 192th sample number = 0
hw(66) = 0;
hw(192) = 0;

% plot of the impulse response of the IDEAL LOW PASS FILTER
figure;
plot(hw,'Linewidth',1.5);
xlabel('frequency (radians/sec) ->');
ylabel('H(\omega) ->');
title('19ucc023 - Mohit Akhouri','Impulse response H(\omega) of an
 IDEAL LOW PASS FILTER for removing hissing frequency components');
grid on;
```

Figure 9.32 Part 1 of the Code for the observation 2

```
% main loop algorithm for dividing the audio file into chunks and
% individually applying fft on each chunk and multiplying with filter
% impulse response to get the smoothened version of audio signal
for i=1:samples:x_length
    x_sampled = x(i:i+255); % taking 256 samples chunk
    x_sampled_fft = fft(x_sampled,256); % taking DFT of the 256
 samples
    yw = x_sampled_fft .* hw; % filtering process takes place
    x(i:i+255) = ifft(yw); % rewriting the original audio file with
 smoothened version of audio
end

x = x'; % taking transpose back again to convert to original form
 ( Nx1 )

% Plot of the smoothened version of audio signal x[n]
figure;
plot(x);
xlabel('samples(n) ->');
ylabel('x[n] ->');
title('19ucc023 - Mohit Akhouri','Plot of smoothened audio signal
 after removing the hissing frequency components');
grid on;

sound(x,fs); % listening to the smoothened version of the audio signal
 after passing through IDEAL LOW PASS FILTER
audiowrite('Exp9_obs_2_low_pass_filter_output.wav',x,fs); % Writing
 the final audio signal to an audio file
```

Published with MATLAB® R2020b

Figure 9.33 Part 2 of the Code for the observation 2

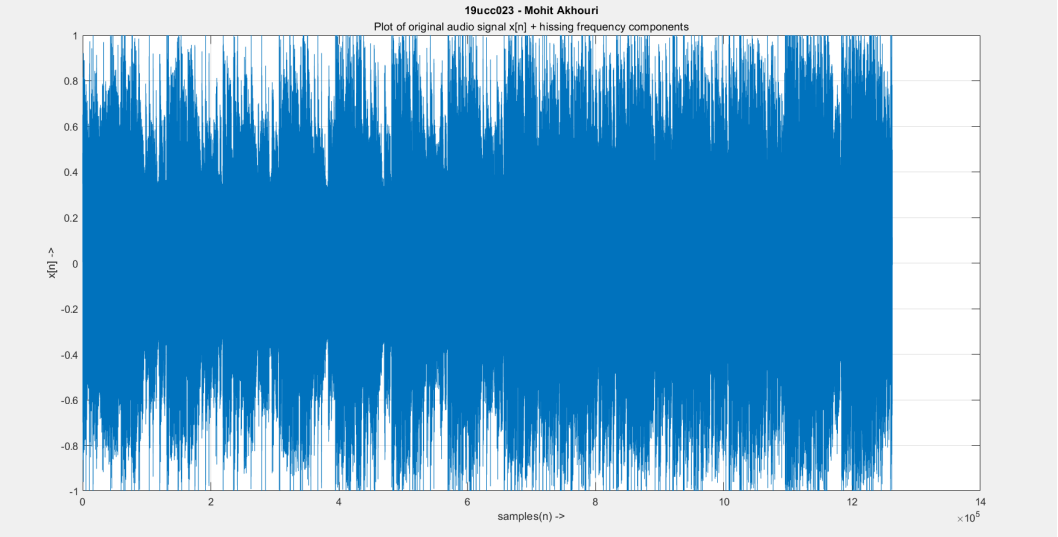


Figure 9.34 Plot of Original Audio Signal + hissing sound

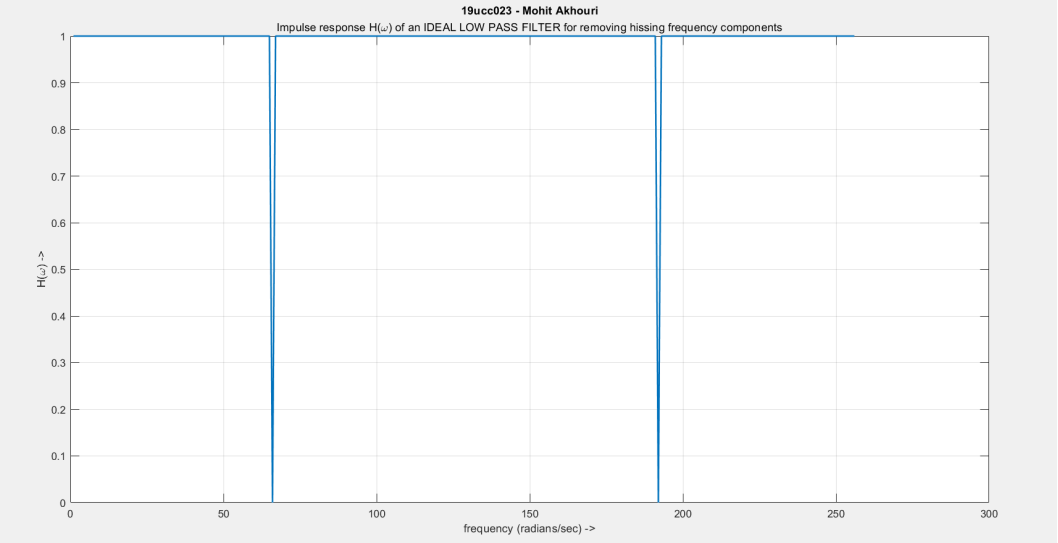


Figure 9.35 Plot of Frequency response of Ideal Low Pass Filter

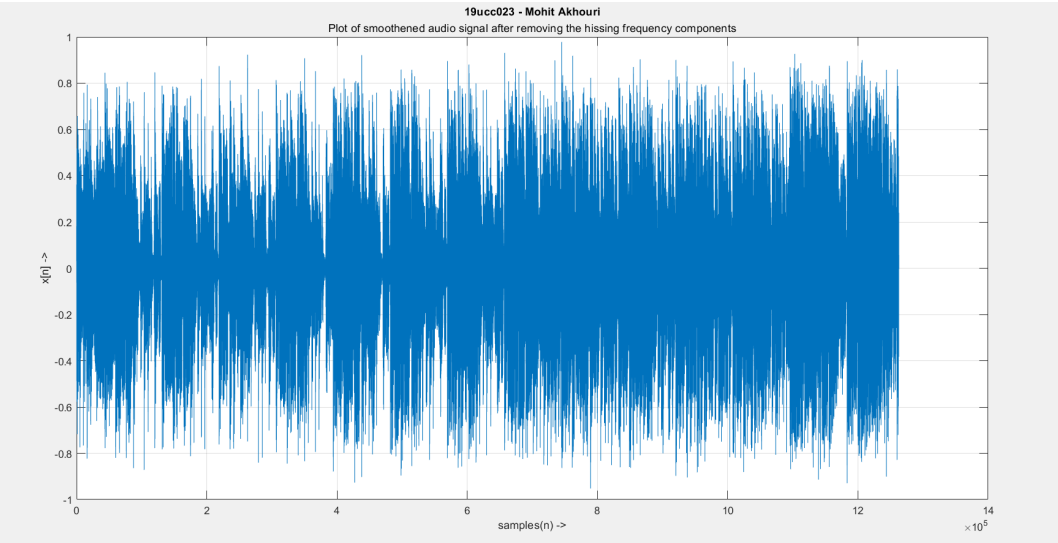


Figure 9.36 Plot of Smoothened audio signal after removing hissing sound

9.4.5 Removing the hissing sound using notch filter :

```
% 19ucc023
% Mohit Akhouri
% Experiment 9 - Observation 3

% This code will read an audio signal 'inputwithhissmadelaptop.wav'
 and
% sample it using 32 Khz. We will take the first 8 samples of this
 audio
% signal and try to locate the hissing frequencies ( approximately at
 3rd
% and 7th samples ).

% Next we will design a NOTCH FILTER and convolve it with distorted
 signal.
% What we finally get is a smoothened version of the audio signal with
% hissing removed and some distortion still left.

clc;
clear all;
close all;

[x,fs] = audioread('inputwithhissmadelaptop.wav'); % reading an audio
 file

samples_x = x(1:8); % Taking first 8 samples of audio signal x[n]
spectrum_sample = abs(fftshift(fft(samples_x))); % Finding the
 frequency spectrum of the first 8 samples for locating hissing
 frequencies

% Plot of the frequency spectrum of the first 8 samples
figure;
stem(spectrum_sample,'Linewidth',1.5);
xlabel('frequency (radians/sec) ->');
ylabel('X(\omega) ->');
title('19ucc023 - Mohit Akhouri','Frequency Spectrum of the first
 8 samples of input x[n] - hissing frequencies at 3^{rd} and 7^{th}
 samples');
grid on;

hn = [1 -2*cos(pi/2) 1]; % Filter impulse response for NOTCH FILTER
x = conv(x,hn); % convolution of distorted signal x[n] and impulse
 response of NOTCH FILTER h[n]

% Plots of input signal x[n] for first 8 samples , impulse response
 h[n]
% and convolved signal y[n].
figure;
subplot(3,1,1);
stem(samples_x,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('x[n] ->');
title('Plot of the first 8 samples of input signal x[n]');
```

Figure 9.37 Part 1 of the Code for the observation 3

```
grid on;
subplot(3,1,2);
stem(hn,'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('h[n] ->');
title('impulse response h[n] of the NOTCH FILTER used for
 convolution');
grid on;
subplot(3,1,3);
stem(x(1:8),'Linewidth',1.5);
xlabel('samples(n) ->');
ylabel('x[n] ->');
title('Plot of the first 8 samples of input signal x[n] after
 convolution with NOTCH FILTER');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');

% Plot of the smoothened version of audio signal after removing the
 hissing
% frequencies
figure;
plot(x);
xlabel('samples(n) ->');
ylabel('x[n] ->');
title('19ucc023 - Mohit Akhouri','smoothened version of the audio
 signal x[n] after passing through NOTCH FILTER');
grid on;

sound(x,fs); % listening to the smoothened version of the audio signal
 after passing through NOTCH FILTER
audiowrite('Exp9_obs_3_notch_filter_output.wav',x,fs); % Writing the
 final audio signal to an audio file
```

Published with MATLAB® R2020b

Figure 9.38 Part 2 of the Code for the observation 3

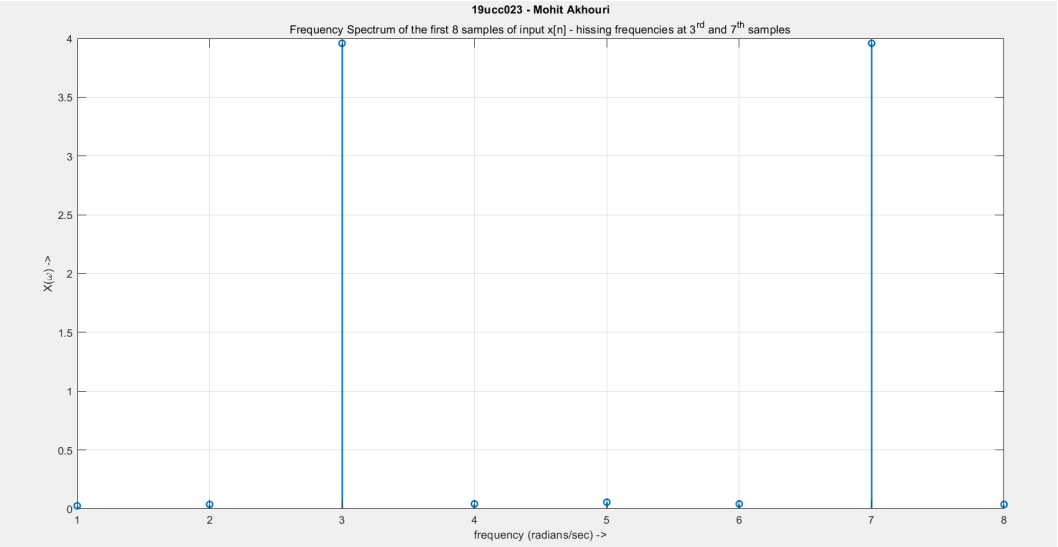


Figure 9.39 Frequency spectrum of the first 8 samples of input audio signal x[n]

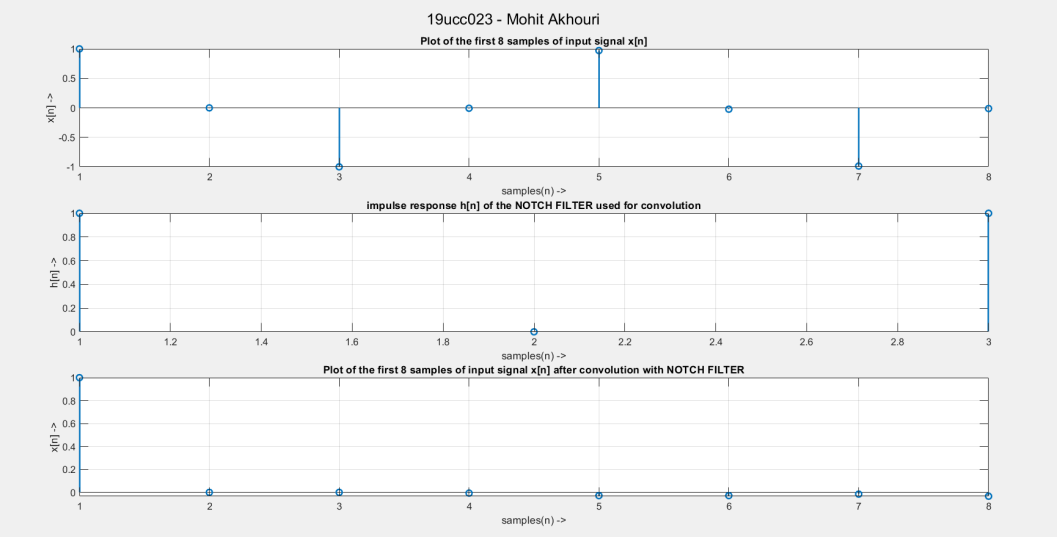


Figure 9.40 Plots for the filtering process via convolution with notch filter

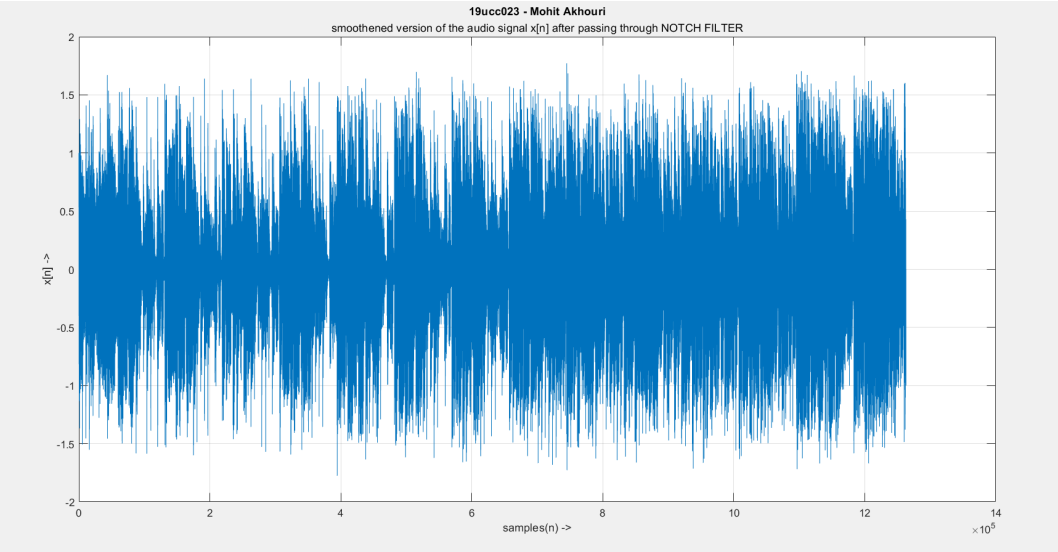


Figure 9.41 Plot of Smoothened audio signal after removing hissing sound

9.4.6 Ideal Low pass filter design and Notch filter design in Simulink :

```
% 19ucc023
% Mohit Akhouri
% Experiment 9 - Observation 4

% This code will call the simulink model and perform the filtering
% operations on the given audio signal with the help of IDEAL LOW PASS
% FILTER and NOTCH FILTER

sim('Simulink_Observation_5'); % calling the simulink model
fs = 32000; % Sampling frequency of audio signal

% Uncomment the below lines if you want to hear the audio signals
% sound(out.input_lowpass_filter.data,fs);
% sound(out.output_lowpass_filter.data,fs);

% Plots of input audio signal to LOWPASS FILTER and corresponding
 output
% obtained via SIMULINK MODEL
figure;
subplot(2,1,1);
plot(abs(fftshift(fft(out.input_lowpass_filter.data))));
xlabel('frequency (radians/sec) ->');
ylabel('X(\omega) ->');
title('Frequency spectrum of original input audio signal with hiss
 frequency greater than \pi/2');
grid on;
subplot(2,1,2);
plot(abs(fftshift(fft(out.output_lowpass_filter.data))));
xlabel('frequency (radians/sec) ->');
ylabel('Y(\omega) ->');
title('Frequency spectrum of smoothened output audio signal after
 passing through LOW PASS FILTER');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');

% Uncomment the below lines if you want to hear the audio signals
% sound(out.input_notch_filter.data,fs);
% sound(out.output_notch_filter.data,fs);

% Plots of input audio signal to NOTCH FILTER and corresponding output
% obtained via SIMULINK MODEL
figure;
subplot(2,1,1);
plot(abs(fftshift(fft(out.input_notch_filter.data))));
xlabel('frequency (radians/sec) ->');
ylabel('X(\omega) ->');
title('Frequency spectrum of original input audio signal with hiss
 frequency present at 3^{rd} and 7^{th} samples');
grid on;
subplot(2,1,2);
plot(abs(fftshift(fft(out.output_notch_filter.data))));
xlabel('frequency (radians/sec) ->');
```

Figure 9.42 Part 1 of the Code for the observation 4

```
ylabel('Y(\omega) ->');
title('Frequency spectrum of smoothened output audio signal after
 passing through NOTCH FILTER');
grid on;
sgtitle('19ucc023 - Mohit Akhouri');
```

Published with MATLAB® R2020b

Figure 9.43 Part 2 of the Code for the observation 4

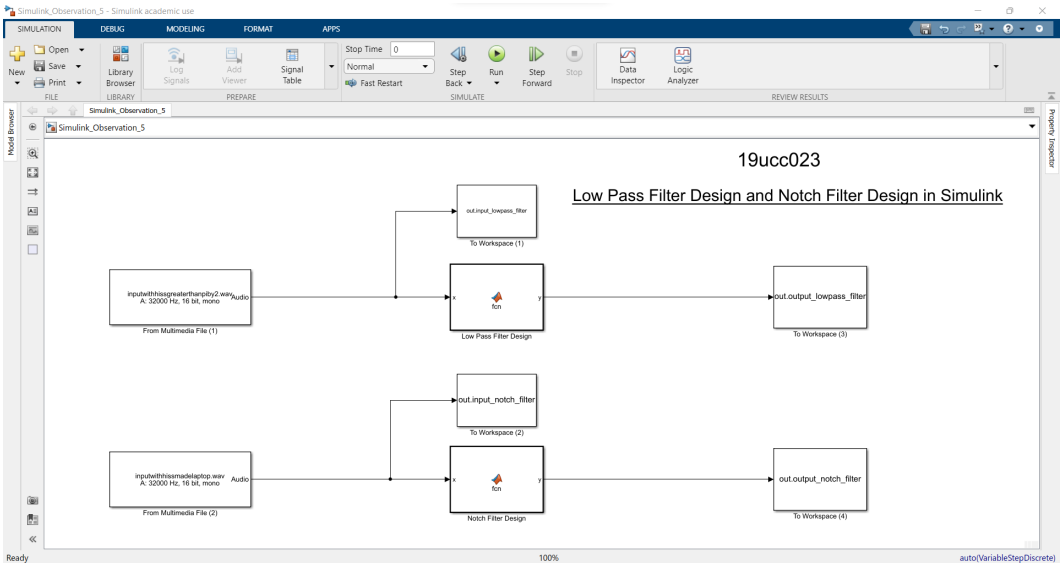


Figure 9.44 Simulink Model used for Filter Design

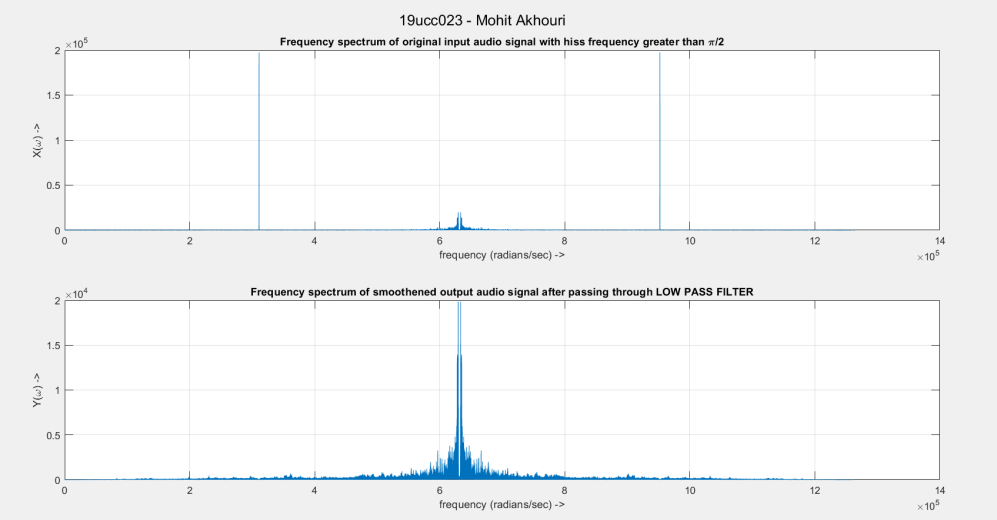


Figure 9.45 Frequency spectrums of original and smoothened audio signal for ideal low pass filter

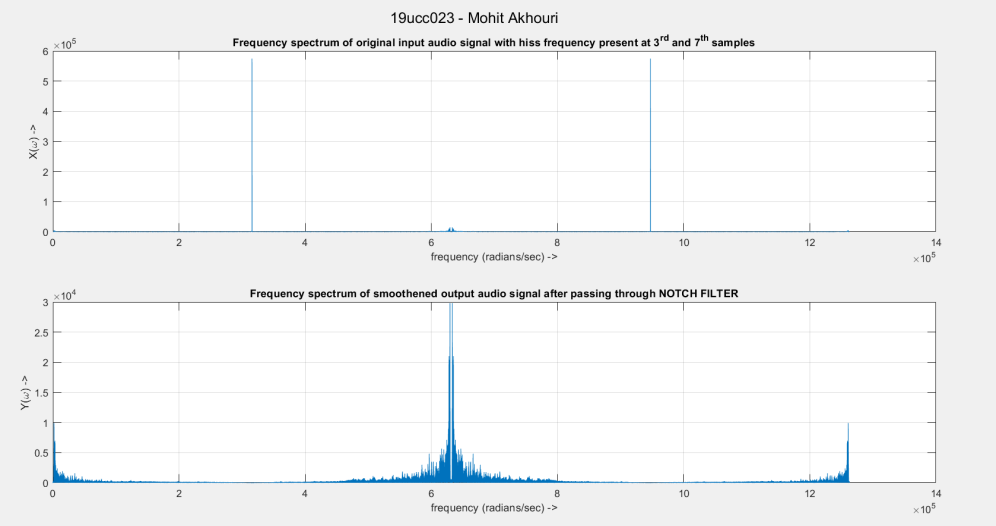


Figure 9.46 Frequency spectrums of original and smoothened audio signal for notch filter

9.5 Conclusion

In this experiment , we learnt the concepts of **Windowing based filter design** , **Low pass filter design** and **Notch filter design** of Digital Signal Processing. We learnt about different types of Window functions like - **rectangular window**,**hamming window** and **Blackman window**. We learnt how to design high,low and band pass filters using **window method**. We also learnt about the concept of **hissing sound** in audio signal and how to remove it. We design two types of filters - **Ideal low pass filter** and **Notch filter** for removal of hissing sound. We learnt about new MATLAB functions like **integral**, **audioread**, **sound** and **audiowrite**. We designed different types of filters for different cutoff frequencies like - $\pi/2$, $\pi/4$ and $\pi/6$. We also implemented the filter design concept in Simulink and compared the results obtained via MATLAB coding.