In the name of God

# CHW1 Report - DSP

MohammadParsa Dini -- 400101204

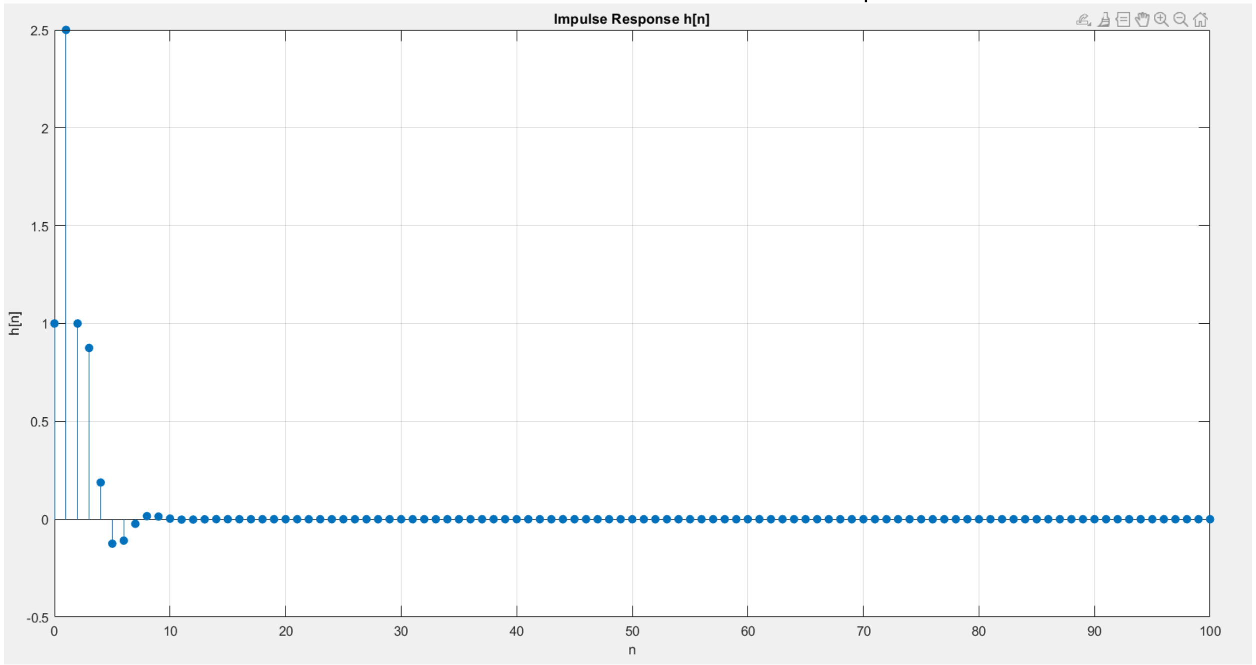
---

## *Q1*

***PART A:*** Here is the implementation of h[n] using recursive method:

```
%% Part (a): Impulse response using recursive method
num_samples = 100; % Number of samples
input_x = zeros(1, num_samples + 1); % Initialize input
output_y = zeros(1, num_samples + 1); % Initialize impulse response
input_x(1) = 1; % Impulse at n=0

% Recursive calculation based on difference equation
for index_n = 1:num_samples
    if index_n == 1
        output_y(index_n) = input_x(index_n);
    elseif index_n == 2
        output_y(index_n) = 2*input_x(index_n-1) + 0.5*output_y(index_n-1);
    elseif index_n == 3
        output_y(index_n) = 0.5*output_y(index_n-1) - 0.25*output_y(index_n-2);
    elseif index_n == 4
        output_y(index_n) = input_x(index_n-3) + 0.5*output_y(index_n-1) -
0.25*output_y(index_n-2);
    else
        output_y(index_n) = 0.5*output_y(index_n-1) - 0.25*output_y(index_n-2);
    end
end

% Plot impulse response
figure;
stem(0:num_samples, output_y, 'filled');
title('Impulse Response h[n]');
xlabel('n');
ylabel('h[n]');
```

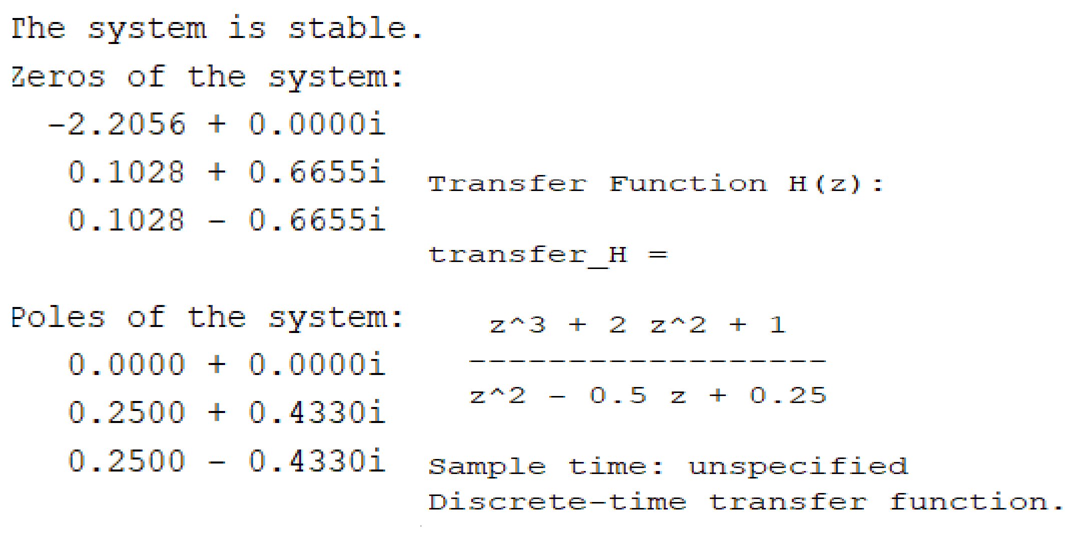
Here is the result of the above code for this difference equation:



***PART B:*** Here is the implementation of h[n] using Z-Transform:

```
%% Part (b): Z-transform and stability check
% Transfer function H(z)
transfer_H = tf(coeff_b, coeff_a, -1); % Define transfer function with discrete time
disp('Transfer Function H(z):');
transfer_H

% Display the impulse response using impz
impulse_response_len = 100; % Length of impulse response
impulse_response_vals = impz(coeff_b , coeff_a, impulse_response_len);
% Plot the impulse response
index_n = 0:impulse_response_len-1;
figure;
stem(index_n, impulse_response_vals , 'filled');
xlabel('n');
ylabel('h[n]');
title('Impulse Response h[n] of the System');
grid on;

% Check poles for stability
poles_H = pole(transfer_H);
disp('Poles of the system:');
disp(poles_H);
if all(abs(poles_H) < 1)
    disp('The system is stable.');
else
    disp('The system is unstable.');
end
```

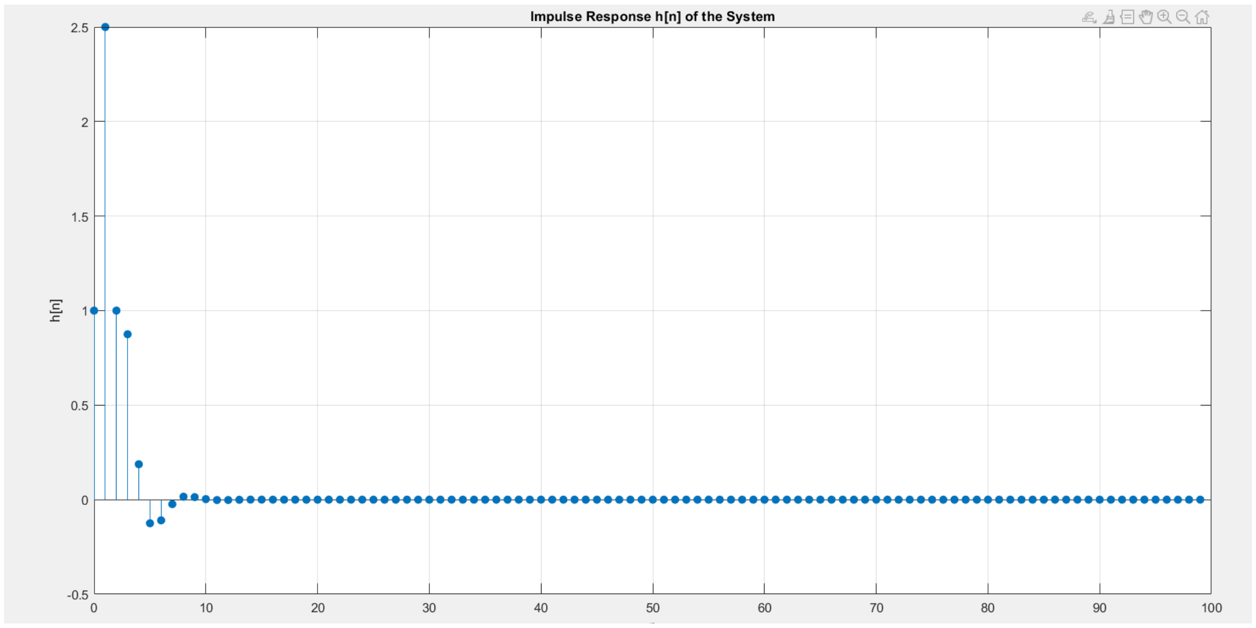
--🡒 Clearly the system is stable since its ROC contain the unit circle in Complex plain.



```
The system is stable.
Zeros of the system:
  -2.2056 + 0.0000i
   0.1028 + 0.6655i
   0.1028 - 0.6655i

Poles of the system:
   0.0000 + 0.0000i
   0.2500 + 0.4330i
   0.2500 - 0.4330i
```

```
Transfer Function H(z):

transfer_H =

   z^3 + 2 z^2 + 1
  -----------------
  z^2 - 0.5 z + 0.25

Sample time: unspecified
Discrete-time transfer function.
```

And this is the result of the code above which is identical to the recursive method's result:



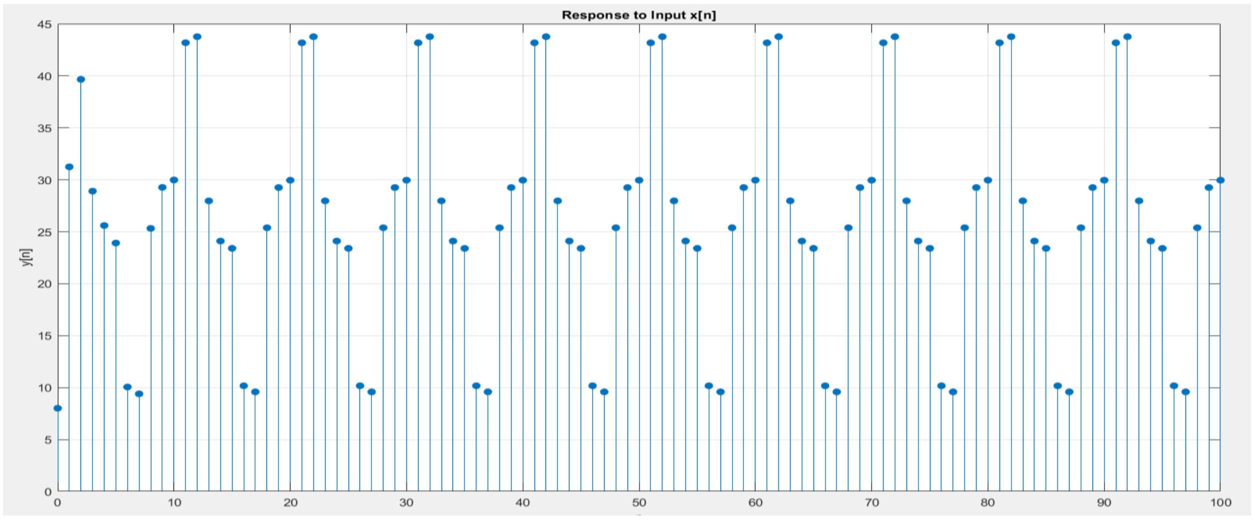
***PART C:*** Here is the code for getting the response of the system with impulse response h[n] to the input

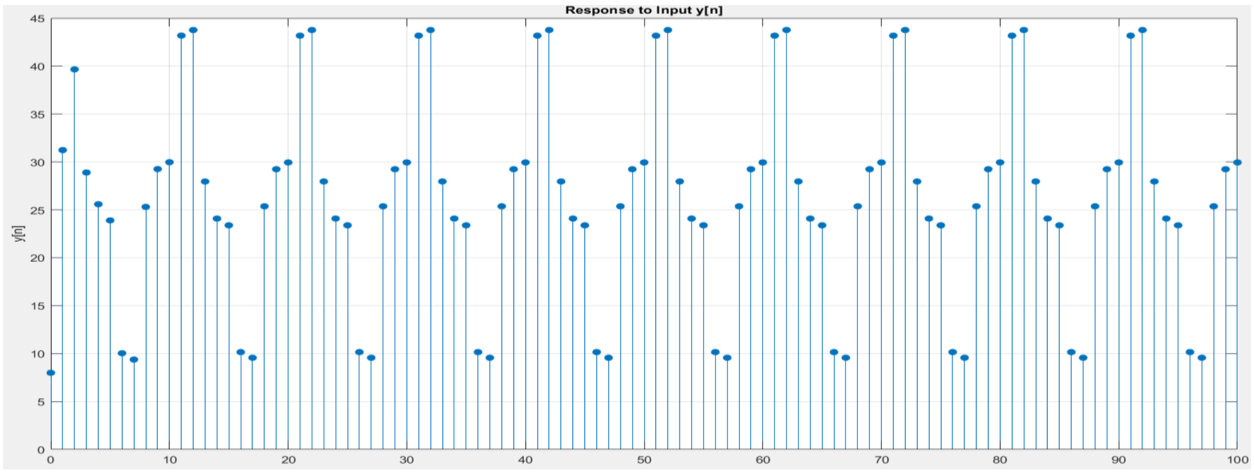
$x[n] = (5 + 3 * \cos(0.2\pi n) + 4\sin(0.6\pi n))u[n]$:

```
%% Part (c): Input response
index_n = 0:100; % Define the range for n
input_x = 5 + 3*cos(0.2*pi*index_n) + 4*sin(0.6*pi*index_n); % Define the input
signal

% Compute the output using the filter function
output_y = filter(coeff_b, coeff_a, input_x); % Apply the transfer function to the
input signal
```

```
% Plot the input
figure;
stem(index_n, input_x, 'filled');
title('Input x[n]');
xlabel('n');
ylabel('x[n]');
grid on;

% Plot the response to the specific input
figure;
stem(index_n, output_y, 'filled');
title('Response to Input x[n]');
xlabel('n');
ylabel('y[n]');
grid on;
```

And this is the result for the code above( The first image is the input versus time, while the second image is the output versus time

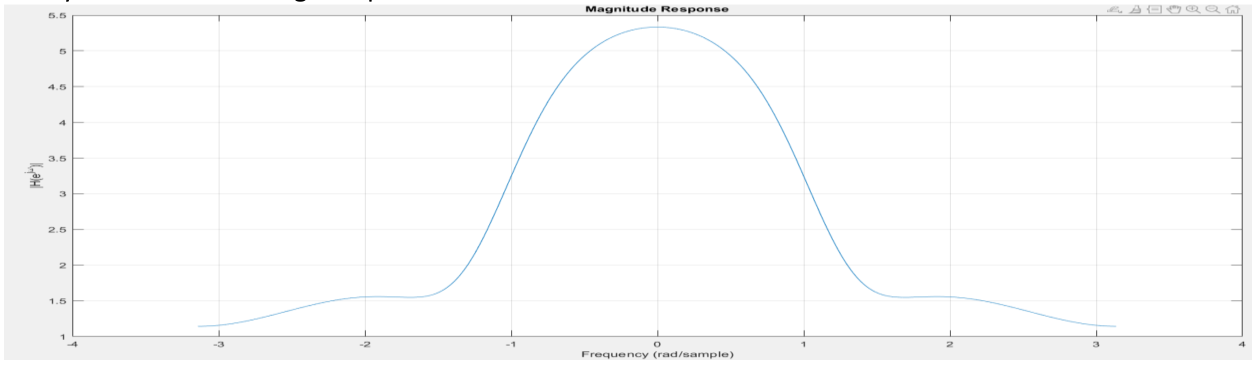




***PART D:*** Here is the code for getting the frequency response of the system:

```
%% Part (d): Amplitude of frequency response
% Frequency response plot
[response_H, freq_w] = freqz(coeff_b, coeff_a, 'whole', 1024);
freq_w = freq_w - pi;  % Shift to range [-pi, pi]
response_H = fftshift(response_H); % Center the zero-frequency component

% Plot magnitude
figure;
plot(freq_w, abs(response_H));
title('Magnitude Response');
xlabel('Frequency (rad/sample)');
ylabel('|H(e^{j\omega})|');
grid on
```

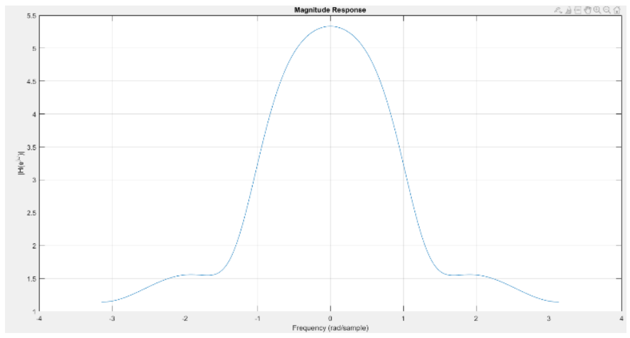
as you can see, the system appears to be a lowpass system as signals of low frequency will pass through the system without being damped.

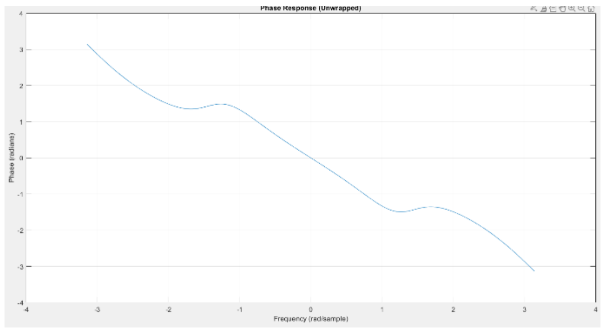


***PART E:*** Here is the code for plotting the freq response, once wrapped, once unrapped

```
%%  Part (e): Phase of frequency response
% Plot phase (wrapped)
figure;
plot(freq_w, angle(response_H));
title('Phase Response (Wrapped)');
xlabel('Frequency (rad/sample)');
ylabel('Phase (radians)');
grid on

% Plot phase (unwrapped)
figure;
plot(freq_w, unwrap(angle(response_H)));
title('Phase Response (Unwrapped)');
xlabel('Frequency (rad/sample)');
ylabel('Phase (radians)');
grid on
```

and here is the result from code above(the left image is wrapped, the right one isn't):

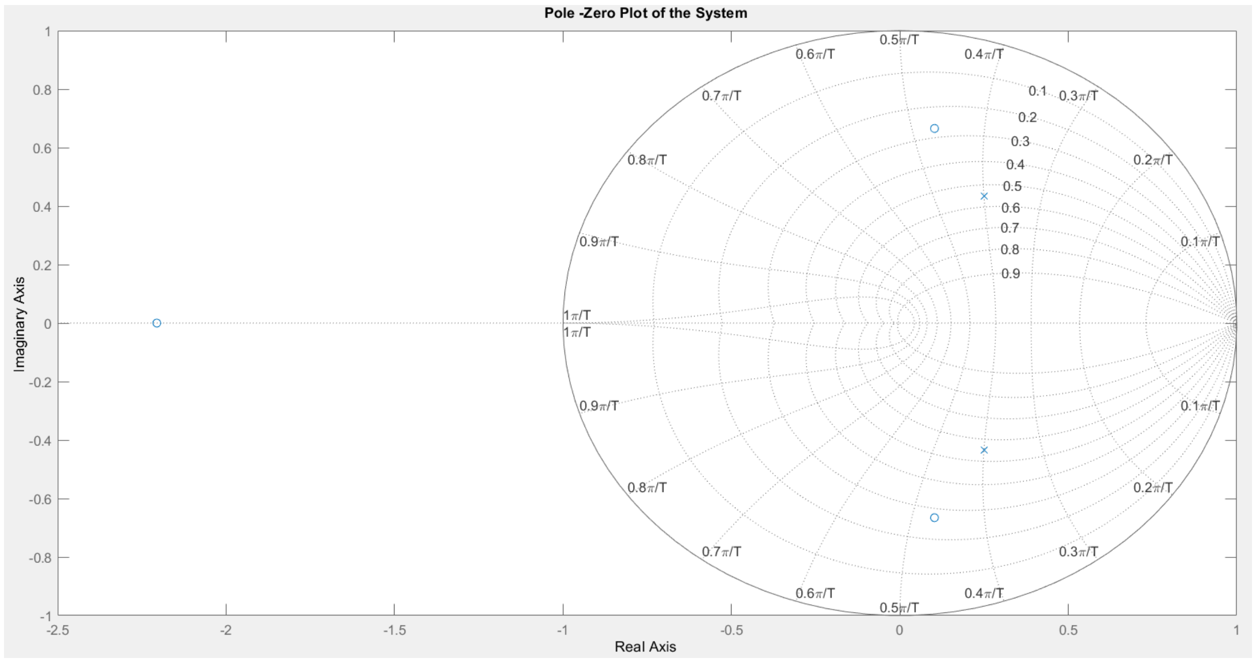




***PART E:*** Here is the code for plotting the zeros and poles of the system:

```
%% Part (f): Pole-Zero plot
% Compute the zeros , poles , and gain using tf2zp
[zeros_H, poles_H, gain_H] = tf2zpk(coeff_b, coeff_a);

% Display the zeros , poles
disp('Zeros of the system:');
disp(zeros_H);
disp('Poles of the system:');
disp(poles_H);

% Plot the zeros and poles using pzplot
figure;
pzplot(transfer_H); % -1 indicates Z-domain
title('Pole -Zero Plot of the System');
grid on;
```

and here is the result from above code:

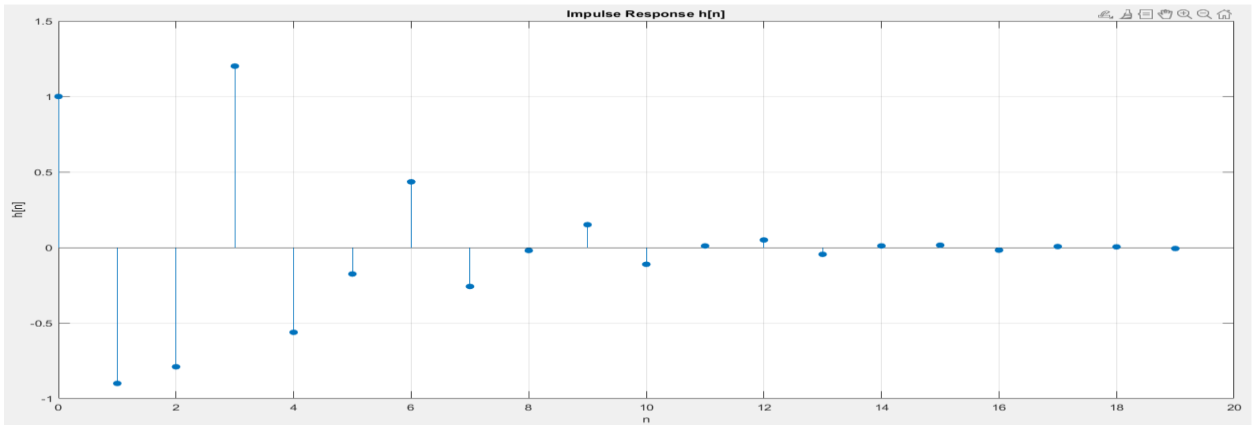


## Q2

Here is the code for fraction decomposition of the transfer function in Z-Domain:

```
clc, clear, close all

% Define numerator and denominator coefficients
num = [1 0 -1];
den = [1 0.9 0.6 0.05];

% Use residue function to get the partial fraction expansion
[r, p, k] = residue(num, den);

% Define the number of samples for impulse response
n_samples = 20; % you can adjust this as needed
n = 0:n_samples-1;

% Initialize the impulse response array
h = zeros(1, n_samples);

% Sum the terms from each residue and pole
for i = 1:length(r)
    h = h + r(i) * (p(i) .^ n); % Accumulate each term in the impulse response
end

% Display the impulse response
stem(n, h, 'filled');
title('Impulse Response h[n]');
xlabel('n');
ylabel('h[n]');
grid on;
```

The impulse response h[n] of the system, obtained through partial fraction decomposition, reveals the system's behavior, stability, and transient characteristics by showing how it reacts to an impulse input. This method breaks down the response into individual components, highlighting each pole's contribution.



## Q3

Here is the code for the whole section:

```
clc, clear, close all

% Parameters
f1 = 4e3;      % Initial frequency in Hz
mu = 600e3;    % Sweep rate in Hz/s
phi = 0;       % Initial phase
fs = 8e3;      % Sampling frequency in Hz
t_duration = 0.05;  % Duration of the signal in seconds

% Time vector for continuous signal
t = linspace(0, t_duration, 1000);

% Continuous chirp signal
x_t = cos(pi * mu * t.^2 + 2 * pi * f1 * t + phi);

% Instantaneous frequency calculation
syms t_sym
phase = pi * mu * t_sym^2 + 2 * pi * f1 * t_sym + phi;  % Phase of the signal
instantaneous_freq = diff(phase, t_sym) / (2 * pi);     % Instantaneous frequency in
Hz
instantaneous_freq_values = double(subs(instantaneous_freq, t_sym, t)); % Substitute
t values for plotting

% Plot continuous signal
figure;
plot(t, x_t);
title('Continuous Chirp Signal');
xlabel('Time (s)');
ylabel('Amplitude');

% Plot instantaneous frequency
figure
plot(t, instantaneous_freq_values);
title('Instantaneous Frequency');
xlabel('Time (s)');
ylabel('Frequency (Hz)');
grid on

% Sampling the signal
n = 0:1/fs:t_duration;
x_sampled = cos(pi * mu * n.^2 + 2 * pi * f1 * n + phi);

% Plot sampled signal
figure
stem(n, x_sampled);
title('Sampled Chirp Signal');
```

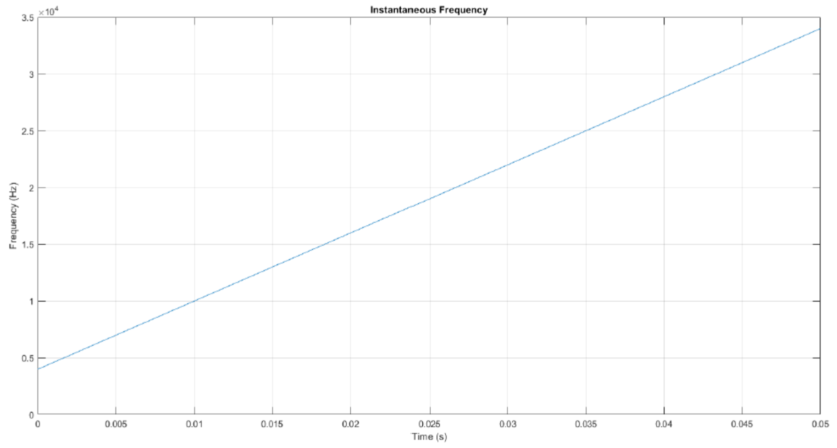
```
xlabel('Time (s)');
ylabel('Amplitude');

% Aliasing check
max_instantaneous_freq = max(instantaneous_freq_values);
if max_instantaneous_freq > fs / 2
    disp('Aliasing might occur since the maximum instantaneous frequency exceeds
Nyquist rate.');
else
    disp('No aliasing issue detected.');
end
```

***Part A:***

For the signal $x(t) = \cos(\pi\mu t^2 + 2\pi f_1 t + \phi)$ $x(t) = \cos(\pi\, \mu\, t^2 + 2\pi\, f_1 t + \phi)$

, the instantaneous frequency is determined by taking the derivative of the argument of the cosine function with respect to time:

$$f(t) = \frac{1}{2}\pi\frac{d}{dt}(\pi\mu t^2 + 2\pi f_1 t + \phi) = \mu t + f_1$$



This result shows that the frequency varies linearly over time, which confirms the chirp nature of the signal. This means the frequency increases or decreases at a constant rate as time progresses.

***Part B:***

The instantaneous frequency f(t) of a linear chirp signal is given by $f(t) = f_1 + \mu t f(t) = f_1 + \mu\, t$. At t=0, the starting frequency is $f(0) = f_1 = 4\,\text{kHz}$. At t=T=0.05 such tha t = T = 0.05s , the ending frequency is $f(T) = f1 + \mu T$. Substituting the values, we get

$$f(T) = 4\,\text{kHz} + (600\,\text{kHz/s}) \cdot (0.05\,\text{s})$$

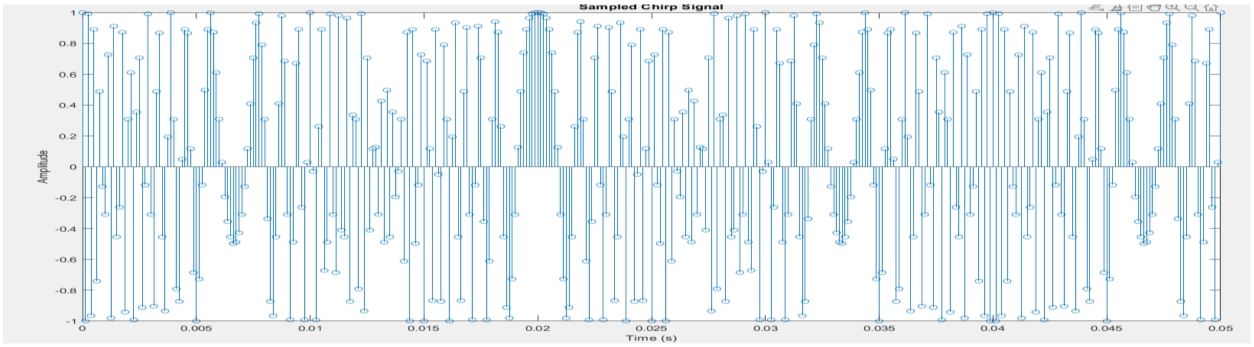
$$f(T) = 4\,\text{kHz} + 30\,\text{kHz} = 34\,\text{kHz}$$

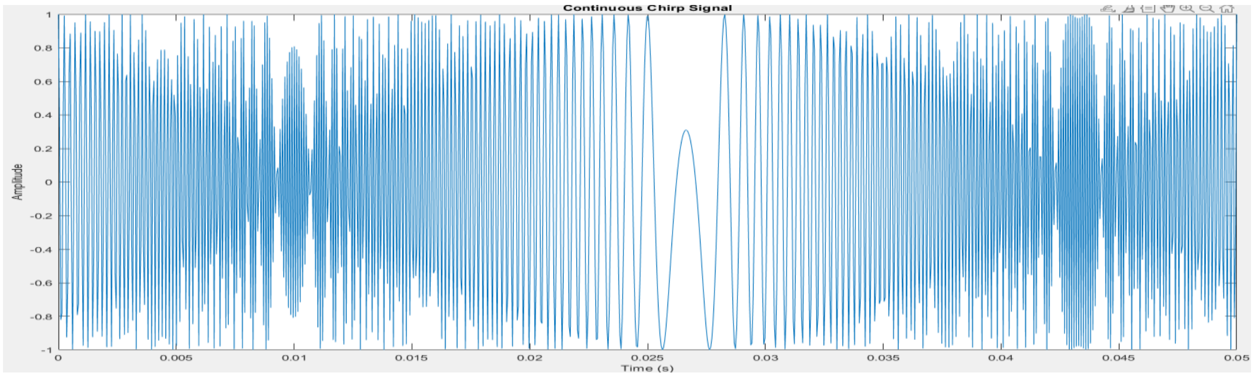
Therefore, the frequency sweep range for the chirp signal is from 4 kHz to 34 kHz over the 0.05-second interval, demonstrating the increase in frequency due to the positive sweep rate $\mu$.

***Part C, D:***

Here is the signal in time domain, as you can see as the times passes the frequency and the phase will change so we will end up with a chaotic signal as depicted down below. The code to this is above.

The signal above is sampled with 8khz whereas the signal below it is sampled with much more frequency such that it looks continuous to us:





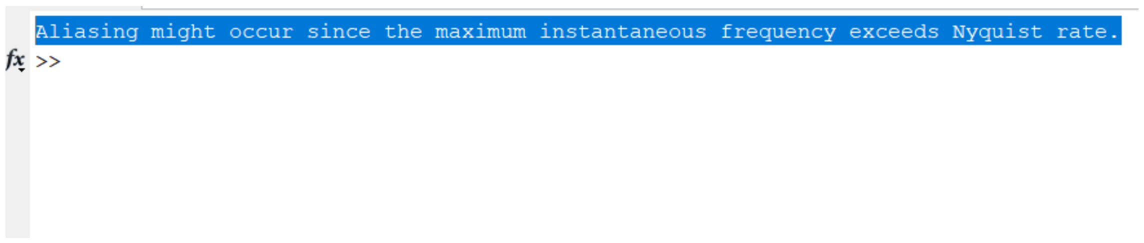
Aliasing can happen when the instantaneous frequency in the sampled signal exceeds the Nyquist frequency of 4 kHz. This results in frequency folding, as observed in the figure, when the frequency surpasses this limit. The phenomenon occurs due to the sampling frequency being too low to accurately capture the higher frequencies in the chirp signal.

***Part E:***

This is the code for aliasing check:

```
% Aliasing check
max_instantaneous_freq = max(instantaneous_freq_values);
if max_instantaneous_freq > fs / 2
    disp('Aliasing might occur since the maximum instantaneous frequency exceeds Nyquist rate.');
else
    disp('No aliasing issue detected.');
end
```

And this is the result:



```
Aliasing might occur since the maximum instantaneous frequency exceeds Nyquist rate.
>>
```

---

## *Q4*

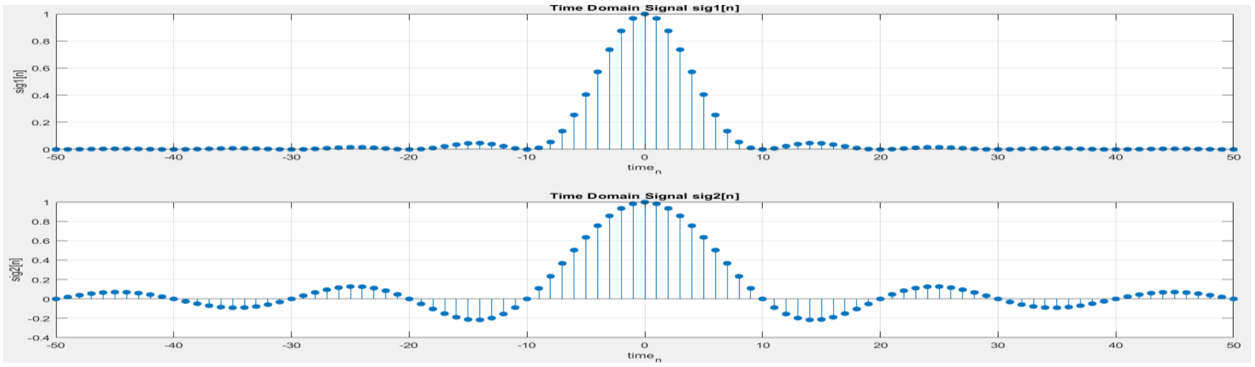
Here is the code for this section:

```
clc; clear; close all;

% Define the range of n for time domain signals
time_n = -50:50;  % Adjust range as needed for accurate FFT

% Define signals sig1[n] and sig2[n]
sig1 = (sin(pi/10 * time_n).^2) ./ ((pi/10 * time_n).^2);
sig2 = sin(pi/10 * time_n) ./ (pi/10 * time_n);

% Handle the time_n = 0 case to avoid division by zero
sig1(time_n == 0) = 1; % lim sig1 as time_n -> 0
sig2(time_n == 0) = 1; % lim sig2 as time_n -> 0

% Plot sig1[n] and sig2[n] in the time domain
figure;
subplot(2,1,1);
stem(time_n, sig1, 'filled');
title('Time Domain Signal sig1[n]');
xlabel('time_n');
ylabel('sig1[n]');
grid on;

subplot(2,1,2);
stem(time_n, sig2, 'filled');
title('Time Domain Signal sig2[n]');
xlabel('time_n');
ylabel('sig2[n]');
```

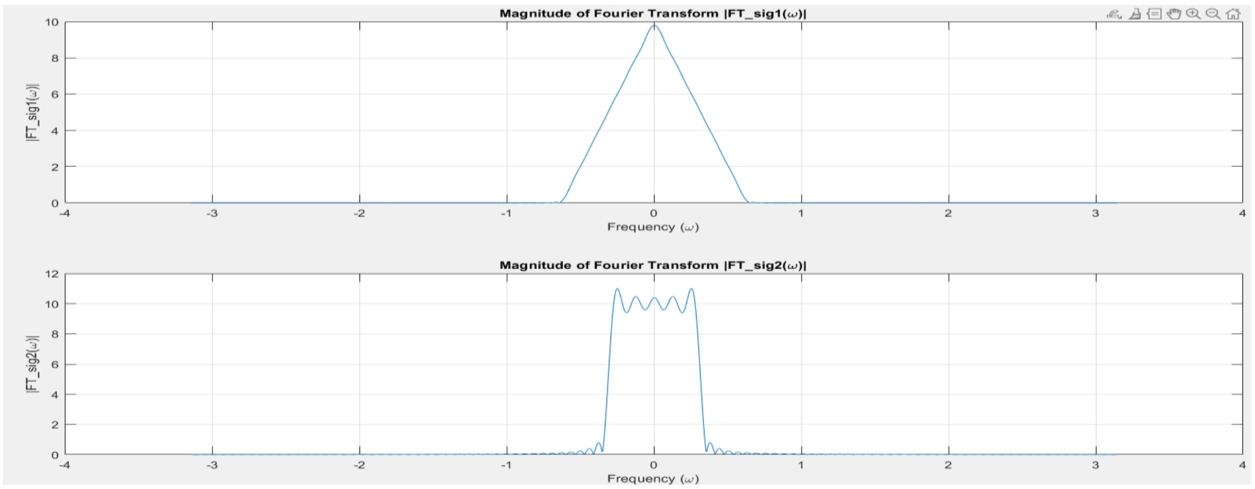
```
grid on;

% Compute Fourier Transforms using fft and fftshift for centered frequency range
FT_sig1 = fftshift(fft(sig1, 1024));  % Use 1024-point FFT for better resolution
FT_sig2 = fftshift(fft(sig2, 1024));
freq_f = linspace(-pi, pi, 1024);  % Frequency vector in [-pi, pi]

% Plot Fourier Transforms of sig1[n] and sig2[n]
figure;
subplot(2,1,1);
plot(freq_f, abs(FT_sig1));
title('Magnitude of Fourier Transform |FT\_sig1(\omega)|');
xlabel('Frequency (\omega)');
ylabel('|FT\_sig1(\omega)|');
grid on;

subplot(2,1,2);
plot(freq_f, abs(FT_sig2));
title('Magnitude of Fourier Transform |FT\_sig2(\omega)|');
xlabel('Frequency (\omega)');
ylabel('|FT\_sig2(\omega)|');
grid on;

% Part (b): Define modified signals mod_sig1[n], mod_sig2[n], mod_sig3[n] based on
sig2[n]
% mod_sig1[n] = sig2[2n]
mod_sig1 = sig2(1:2:end);  % Direct downsampling, resulting in half the length of
sig2
mod_sig1 = mod_sig1(mod_sig1 ~= 0);

% mod_sig2[n] as described in the question
mod_sig2 = zeros(size(time_n));
mod_sig2(mod(time_n, 2) == 0) = sig2((time_n(mod(time_n, 2) == 0) / 2) +
(length(time_n) + 1) / 2);

% mod_sig3[n] = sig2[n] * sin(2π * 0.3 * n)
mod_sig3 = sig2 .* sin(2 * pi * 0.3 * time_n);

% Plot sig2[n], mod_sig1[n], mod_sig2[n], mod_sig3[n] in the time domain
figure
subplot(4,1,1);
stem(time_n, sig2, 'filled');
title('Time Domain Signal sig2[n]');
xlabel('time_n');
ylabel('sig2[n]');
grid on;

subplot(4,1,2);
stem(-25:25, mod_sig1, 'filled');  % Plot mod_sig1 with correct indices
title('Time Domain Signal mod\_sig1[n] = sig2[2n]');
xlabel('time_n');
ylabel('mod\_sig1[n]');
grid on;

subplot(4,1,3);
```

```
stem(time_n, mod_sig2, 'filled');
title('Time Domain Signal mod\_sig2[n]');
xlabel('time_n');
ylabel('mod\_sig2[n]');
grid on;

subplot(4,1,4);
stem(time_n, mod_sig3, 'filled');
title('Time Domain Signal mod\_sig3[n] = sig2[n] \cdot \sin(2 \pi \cdot 0.3 \cdot
time_n)');
xlabel('time_n');
ylabel('mod\_sig3[n]');
grid on;

% Compute Fourier Transforms of mod_sig1[n], mod_sig2[n], mod_sig3[n]
FT_mod_sig1 = fftshift(fft(mod_sig1, 1024));
FT_mod_sig2 = fftshift(fft(mod_sig2, 1024));
FT_mod_sig3 = fftshift(fft(mod_sig3, 1024));

% Plot Fourier Transforms of mod_sig1[n], mod_sig2[n], mod_sig3[n]
figure;
subplot(4,1,1);
plot(freq_f, abs(FT_sig2));
title('Magnitude of Fourier Transform |FT\_sig2(\omega)|');
xlabel('Frequency (\omega)');
ylabel('|FT\_sig2(\omega)|');
grid on;

subplot(4,1,2);
plot(freq_f, abs(FT_mod_sig1));
title('Magnitude of Fourier Transform |FT\_mod\_sig1(\omega)| for mod\_sig1[n] =
sig2[2n]');
xlabel('Frequency (\omega)');
ylabel('|FT\_mod\_sig1(\omega)|');
grid on;

subplot(4,1,3);
plot(freq_f, abs(FT_mod_sig2));
title('Magnitude of Fourier Transform |FT\_mod\_sig2(\omega)| for mod\_sig2[n]');
xlabel('Frequency (\omega)');
ylabel('|FT\_mod\_sig2(\omega)|');
grid on;

subplot(4,1,4);
plot(freq_f, abs(FT_mod_sig3));
title('Magnitude of Fourier Transform |FT\_mod\_sig3(\omega)| for mod\_sig3[n] =
sig2[n] \cdot \sin(2 \pi \cdot 0.3 \cdot time_n)');
xlabel('Frequency (\omega)');
ylabel('|FT\_mod\_sig3(\omega)|');
grid on;
```

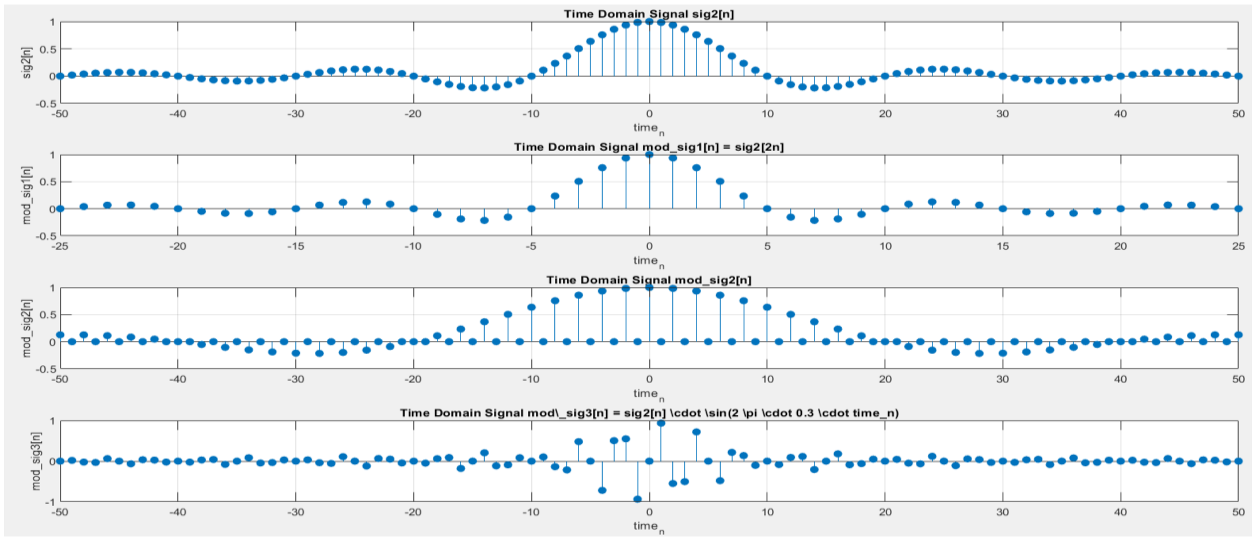
***PART A:*** Here are the signals x1[n] and x2[n]:

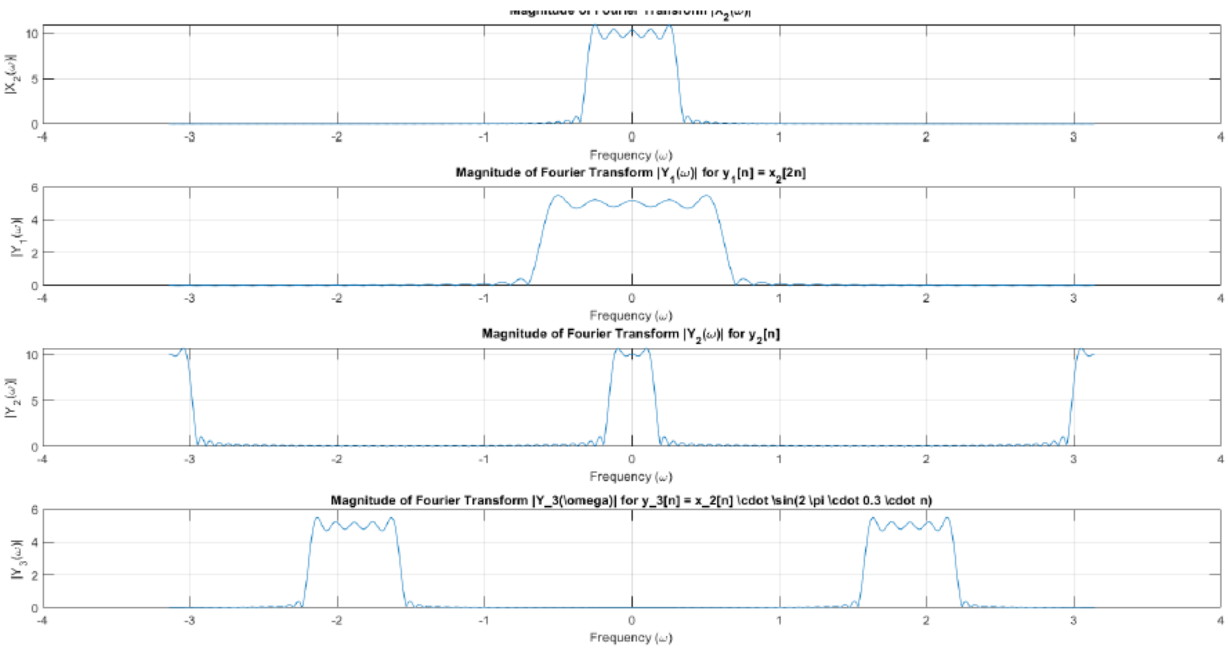


Here are the magnitude of frequency response of x1[n] and x2[n]:



***PART B:***





Figures 1 and 2 show the time-domain signals $x_1[n]$ and $x_2[n]$ and their Fourier transforms. Figures 3 and 4 display the modified signals $y_1[n], y_2[n]$, and $y_3[n]$, based on $x_2[n]$, and their Fourier transforms. This analysis allows us to compare how each transformation affects the frequency spectrum and signal shape.

The original signals $x_1[n]$ and $x_2[n]$ are sinc-like functions that decay in amplitude as nn moves away from zero. Their Fourier transforms reveal narrow-band frequency responses, typical of low-pass signals with energy concentrated near zero frequency.

In the first transformation, $y_1[n] \ = \ x_2[2n]$, downsampling by a factor of 2 creates a sparser signal and causes aliasing in the frequency domain, introducing overlapping frequency components.

The second transformation, $y_2[n]$, involves selective sampling with zero padding, leading to an interleaved structure where every other index is zero. This introduces additional frequency components, or "ghost" frequencies, in the spectrum.

In the third transformation, $y_3[n] = x_2[n] \cdot sin(2\pi \cdot 0.3 \cdot n)$, modulation by a sine wave results in a frequency shift. The frequency spectrum of $x_2[n]$ is shifted to new frequency bands, demonstrating frequency translation.

In summary:

- Downsampling introduces aliasing, causing frequency overlap.
- Selective sampling with zero padding adds high-frequency components due to periodic gaps.
- Modulation shifts the frequency spectrum, illustrating frequency translation.

Understanding these effects is essential in applications like digital communications, audio processing, and image compression, where controlling frequency content is crucial.

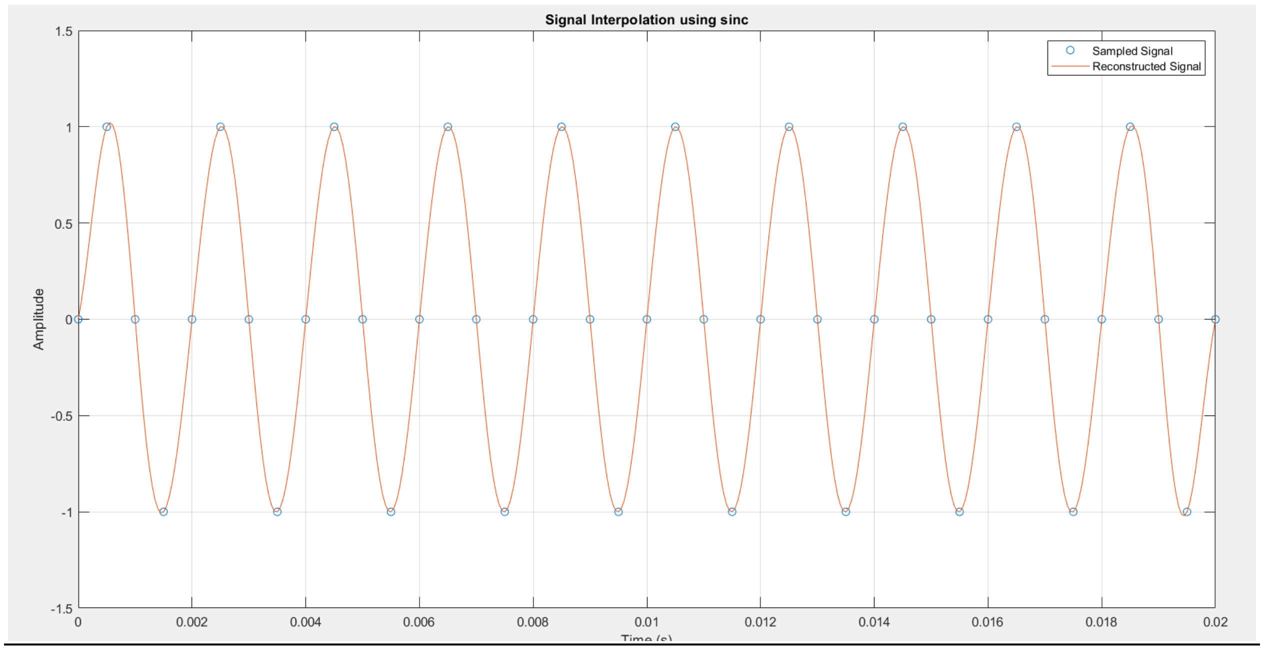
## *Q5*

PART A: Using the script down below we interpolated the signal having its samples in time domain and reconstructed the signal using sinc interpolation.

```
% part a
% Signal parameters
Ts = 0.0005; % 500 microseconds
t = 0:Ts:0.02; % Time range from 0 to 0.02 seconds

% Original signal
x = sin(1000*pi*t) + sin(2000*pi*t);

% Desired times for interpolation
delta = 0.00005; % 50 microseconds
t_recon = 0:delta:0.02;

% Use the interpolation function
y_recon = sinc_interpolation(x, t, t_recon);

% Plot the original and interpolated signals
figure;
plot(t, x, 'o', 'DisplayName', 'Sampled Signal'); % Sampled signal
hold on;
plot(t_recon, y_recon, '-', 'DisplayName', 'Reconstructed Signal'); % Interpolated
signal
title('Signal Interpolation using sinc');
xlabel('Time (s)');
ylabel('Amplitude');
legend;
```

```
grid on;


% sinc_interpolation function for reconstructing signal using sinc interpolation
function y_reconstructed = sinc_interpolation(sampled_signal, sample_times,
desired_times)
    % Number of samples in the original signal
    num_samples = length(sampled_signal);

    % Initialize the output vector
    y_reconstructed = zeros(1, length(desired_times));

    % Interpolation using the sinc function
    for n = 1:num_samples
        y_reconstructed = y_reconstructed + sampled_signal(n) * sinc((desired_times -
sample_times(n)) / (sample_times(2) - sample_times(1)));
    end
end
```

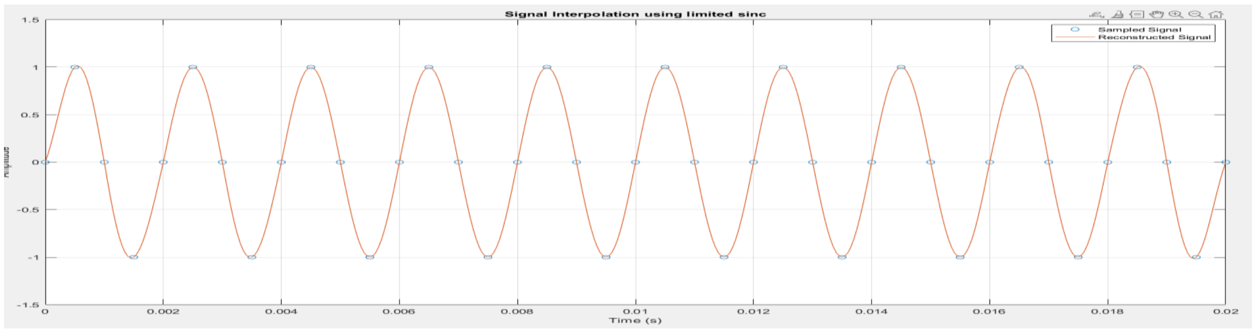
here is the reconstructed signal and its samples in time domain in one look, as it seems, we have done a great job.



Part B:

```
% part b
clc; clear; close all;

% Signal parameters
Ts = 0.0005; % 500 microseconds
t = 0:Ts:0.02; % Time range from 0 to 0.02 seconds

% Original signal
```

```
x = sin(1000*pi*t) + sin(2000*pi*t);

% Desired times for interpolation
delta = 0.00005; % 50 microseconds
t_recon = 0:delta:0.02;

% Use the interpolation function with limited sinc
y_recon = limited_sinc_interpolation(x, t, t_recon);

% Plot the original and interpolated signals
figure;
plot(t, x, 'o', 'DisplayName', 'Sampled Signal'); % Sampled signal
hold on;
plot(t_recon, y_recon, '-', 'DisplayName', 'Reconstructed Signal'); % Interpolated
signal
title('Signal Interpolation using limited sinc');
xlabel('Time (s)');
ylabel('Amplitude');
grid on;
legend

function y_reconstructed = limited_sinc_interpolation(sampled_signal, sample_times,
desired_times)
    % Number of samples in the original signal
    num_samples = length(sampled_signal);

    % Initialize the output vector
    y_reconstructed = zeros(1, length(desired_times));

    % Define the sinc function with limited lobes (9 lobes)
    sinc_lobe_limit = 9;

    % Interpolation using the limited sinc function
    for n = 1:num_samples
        % Calculate the sinc function value only within the lobe limit
        sinc_values = (desired_times - sample_times(n)) / (sample_times(2) -
sample_times(1));
        limited_sinc = sinc(sinc_values) .* (abs(sinc_values) <= sinc_lobe_limit);
        y_reconstructed = y_reconstructed + sampled_signal(n) * limited_sinc;
    end
end
```



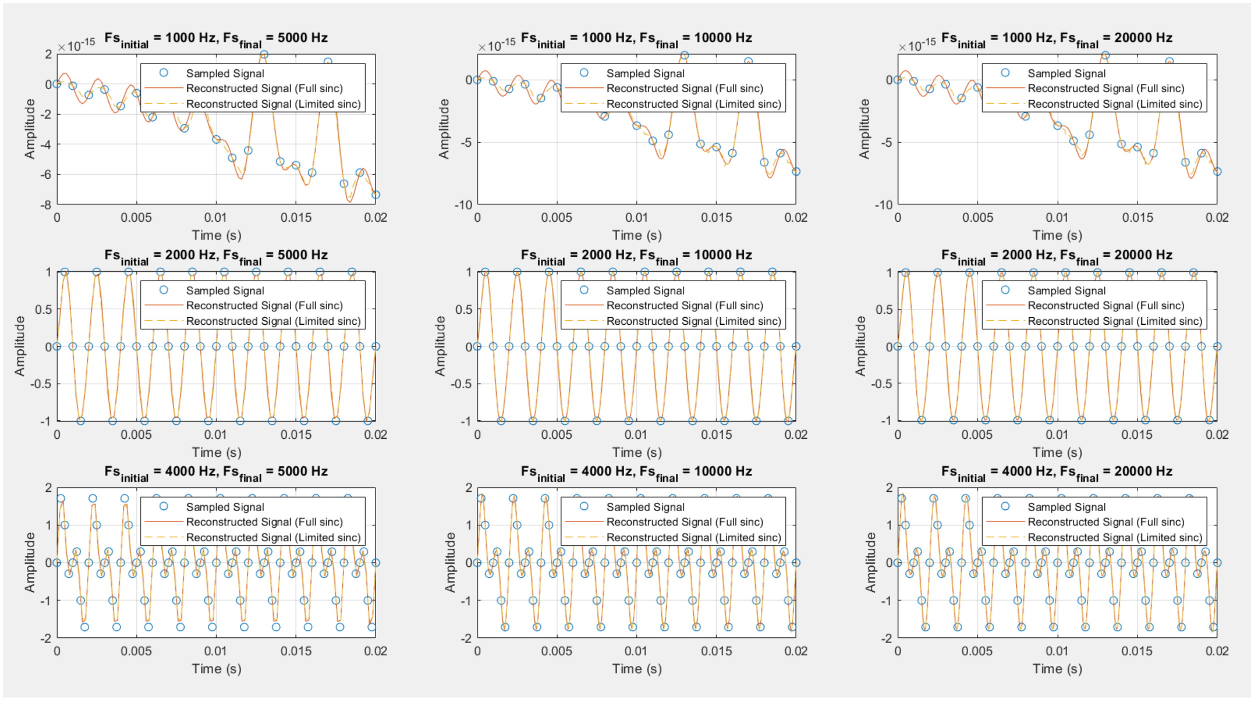
The reconstructed signal still fits the sampled data points.

PART c:

Using less sinc terms we will get less accurate signals that could have resembled our original signal if we used the whole terms.

PART E:

Here is the result for part E



```
% part a
% Signal parameters
Ts = 0.0005; % 500 microseconds
t = 0:Ts:0.02; % Time range from 0 to 0.02 seconds

% Original signal
x = sin(1000*pi*t) + sin(2000*pi*t);

% Desired times for interpolation
delta = 0.00005; % 50 microseconds
t_recon = 0:delta:0.02;

% Use the interpolation function
y_recon = sinc_interpolation(x, t, t_recon);

% Plot the original and interpolated signals
figure;
plot(t, x, 'o', 'DisplayName', 'Sampled Signal'); % Sampled signal
hold on;
plot(t_recon, y_recon, '-', 'DisplayName', 'Reconstructed Signal'); % Interpolated
signal
```

```
title('Signal Interpolation using sinc');
xlabel('Time (s)');
ylabel('Amplitude');
legend;
grid on;


%%
% part b
clc; clear; close all;

% Signal parameters
Ts = 0.0005; % 500 microseconds
t = 0:Ts:0.02; % Time range from 0 to 0.02 seconds

% Original signal
x = sin(1000*pi*t) + sin(2000*pi*t);

% Desired times for interpolation
delta = 0.00005; % 50 microseconds
t_recon = 0:delta:0.02;

% Use the interpolation function with limited sinc
y_recon = limited_sinc_interpolation(x, t, t_recon);

% Plot the original and interpolated signals
figure;
plot(t, x, 'o', 'DisplayName', 'Sampled Signal'); % Sampled signal
hold on;
plot(t_recon, y_recon, '-', 'DisplayName', 'Reconstructed Signal'); % Interpolated
signal
title('Signal Interpolation using limited sinc');
xlabel('Time (s)');
ylabel('Amplitude');
grid on;
legend

%%
clc; clear; close all;



clc; clear; close all;

% Signal parameters
Fs_initial = [1000, 2000, 4000]; % Initial sampling frequencies (Hz)
Fs_final = [5000, 10000, 20000]; % Final sampling frequencies (Hz)
t_duration = 0.02; % Duration of signal (seconds)

% Original signal
original_signal = @(t) sin(1000*pi*t) + sin(2000*pi*t);

figure;

subplot_idx = 1;
```

```
for i = 1:length(Fs_initial)
    for j = 1:length(Fs_final)
        Ts_initial = 1 / Fs_initial(i); % Initial sampling period
        t_initial = 0:Ts_initial:t_duration; % Initial sample times
        x = original_signal(t_initial); % Sampled signal

        Ts_final = 1 / Fs_final(j); % Final sampling period
        t_final = 0:Ts_final:t_duration; % Desired sample times for interpolation

        % Use the sinc interpolation function (full and limited)
        y_recon_full = sinc_interpolation(x, t_initial, t_final);
        y_recon_limited = limited_sinc_interpolation(x, t_initial, t_final);

        % Plot the sampled and reconstructed signals in subplots
        subplot(length(Fs_initial), length(Fs_final), subplot_idx);
        plot(t_initial, x, 'o', 'DisplayName', 'Sampled Signal'); % Sampled signal
        hold on;
        plot(t_final, y_recon_full, '-', 'DisplayName', 'Reconstructed Signal (Full
sinc)'); % Interpolated signal (full sinc)
        plot(t_final, y_recon_limited, '--', 'DisplayName', 'Reconstructed Signal
(Limited sinc)'); % Interpolated signal (limited sinc)
        title(sprintf('Fs_{initial} = %d Hz, Fs_{final} = %d Hz', Fs_initial(i),
Fs_final(j)));
        xlabel('Time (s)');
        ylabel('Amplitude');
        legend;
        grid on;

        subplot_idx = subplot_idx + 1;
    end
end



% Define the sinc_interpolation function
function y_reconstructed = sinc_interpolation(sampled_signal, sample_times,
desired_times)
    num_samples = length(sampled_signal);
    y_reconstructed = zeros(1, length(desired_times));
    for n = 1:num_samples
        y_reconstructed = y_reconstructed + sampled_signal(n) * sinc((desired_times -
sample_times(n)) / (sample_times(2) - sample_times(1)));
    end
end

% Define the limited_sinc_interpolation function
function y_reconstructed = limited_sinc_interpolation(sampled_signal, sample_times,
desired_times)
    num_samples = length(sampled_signal);
    y_reconstructed = zeros(1, length(desired_times));
    sinc_lobe_limit = 4.5;
    for n = 1:num_samples
        sinc_values = (desired_times - sample_times(n)) / (sample_times(2) -
sample_times(1));
        limited_sinc = sinc(sinc_values) .* (abs(sinc_values) <= sinc_lobe_limit);
        y_reconstructed = y_reconstructed + sampled_signal(n) * limited_sinc;
    end
end
```