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;

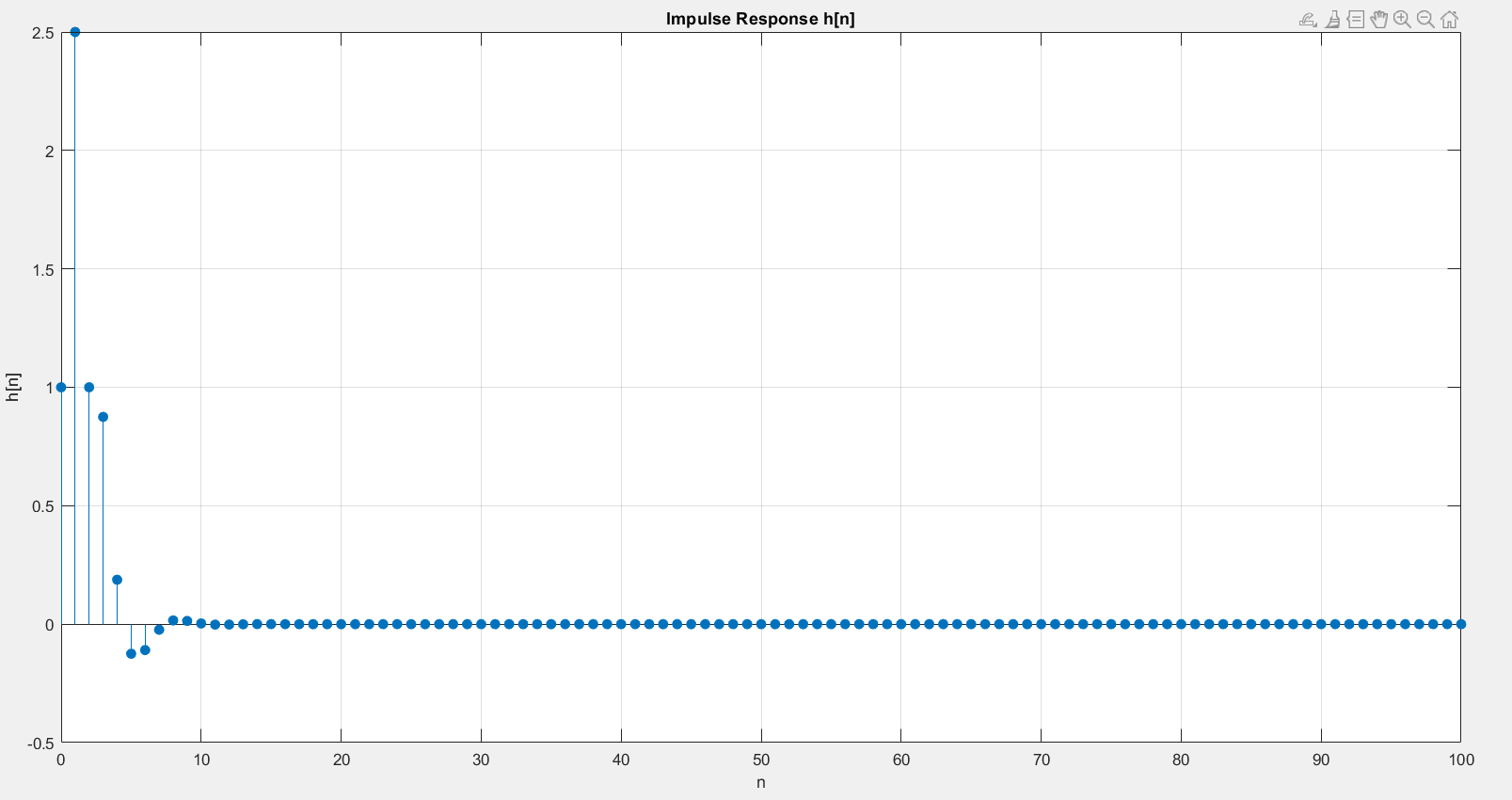
stem(0:num\_samples, output\_y, 'filled');

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)

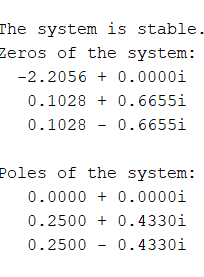
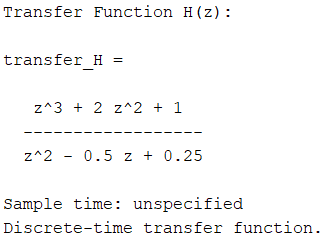
disp('The system is stable.');

else

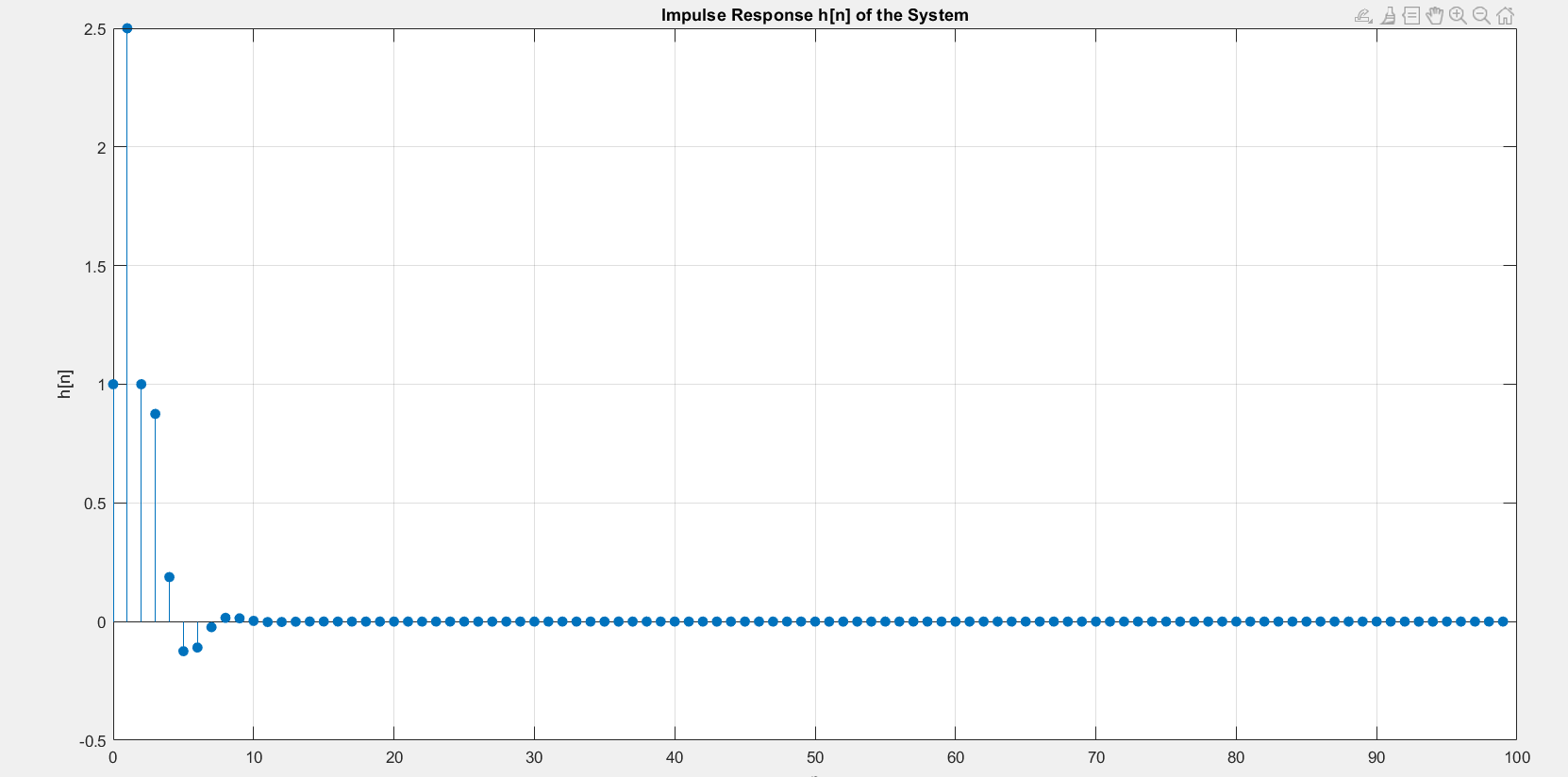
disp('The system is unstable.');

end

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

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(0.2𝜋𝑛) + 4 sin(0.6𝜋𝑛))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');

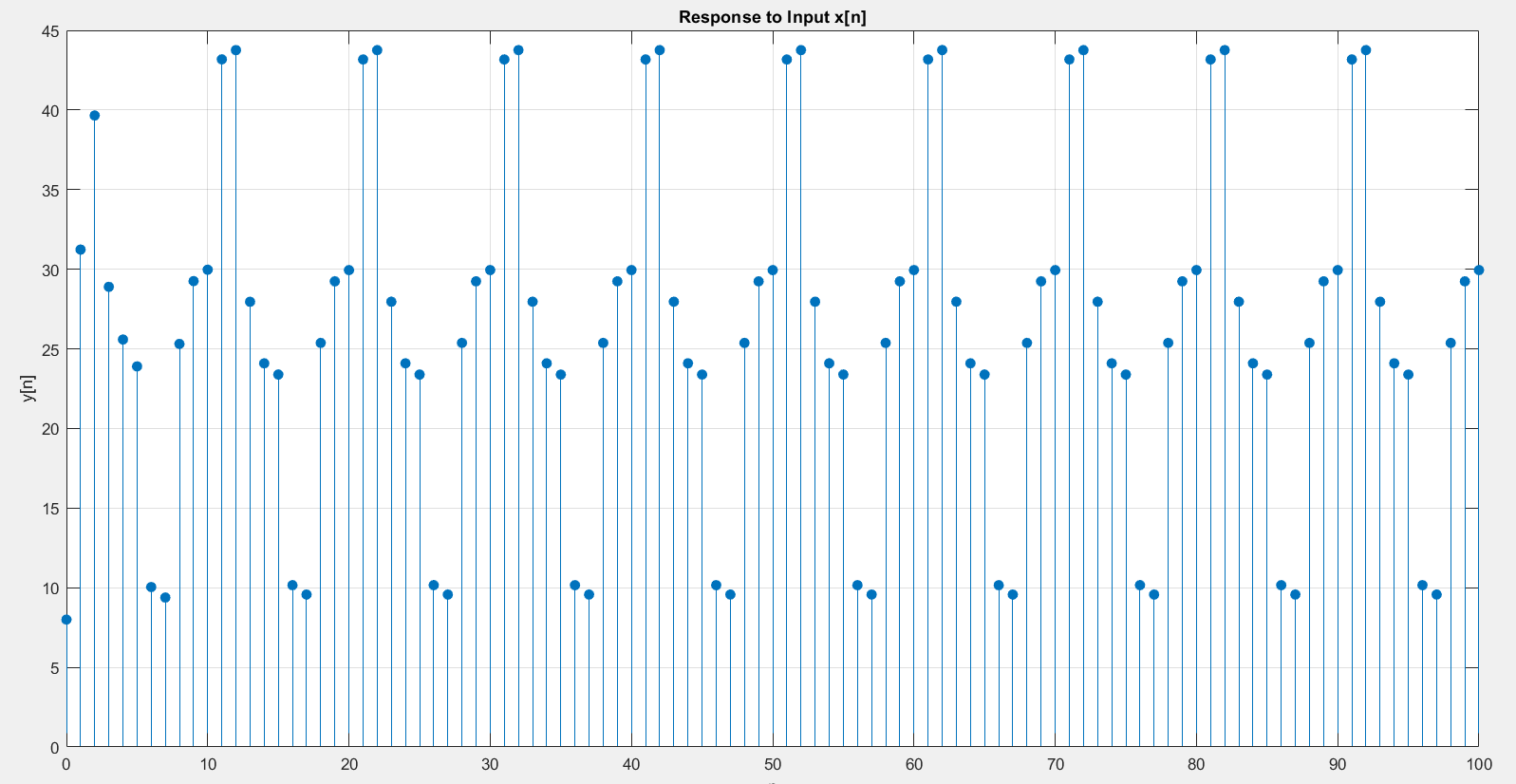
title('Response to Input x[n]');

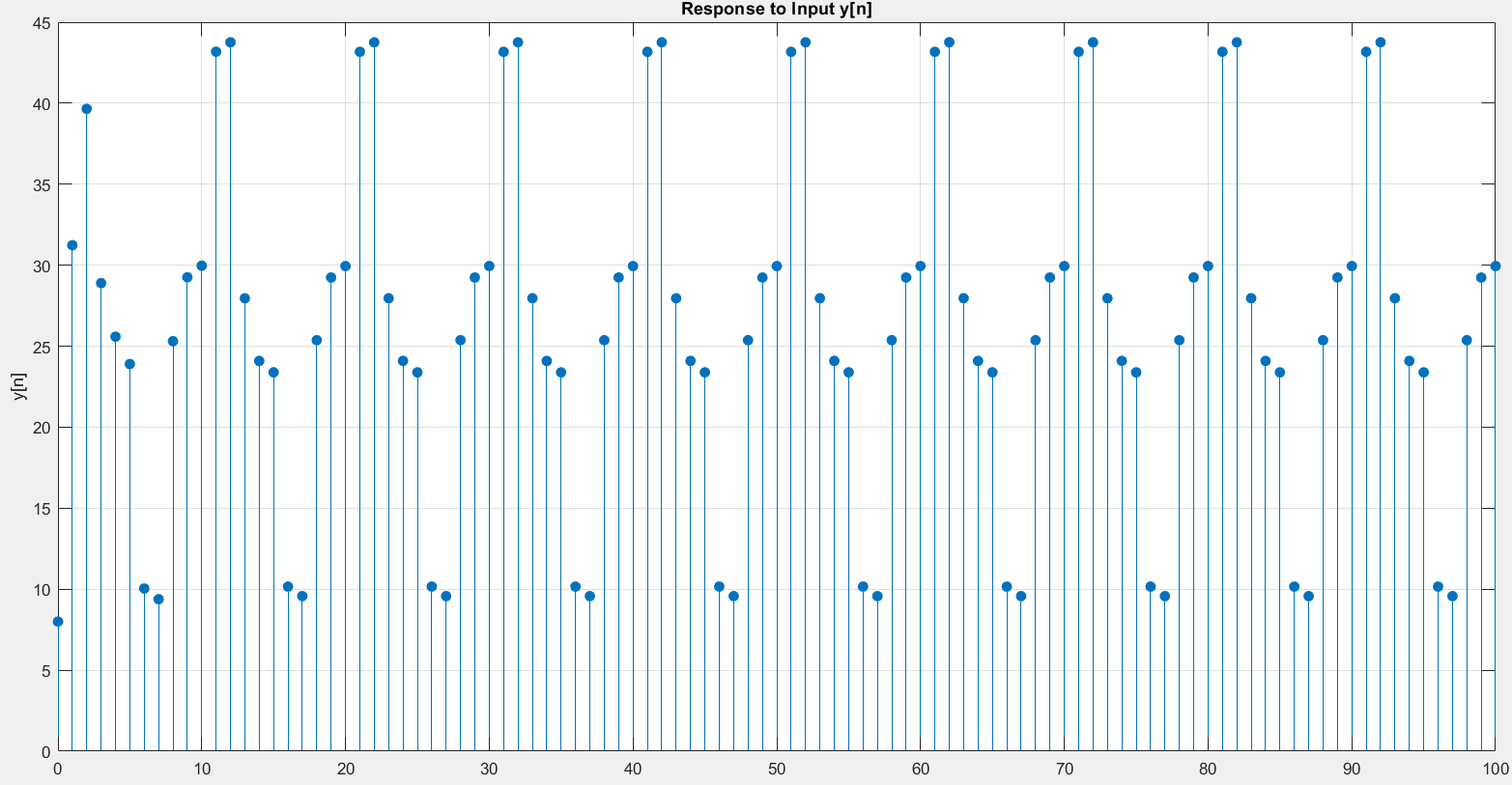
xlabel('n');

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));

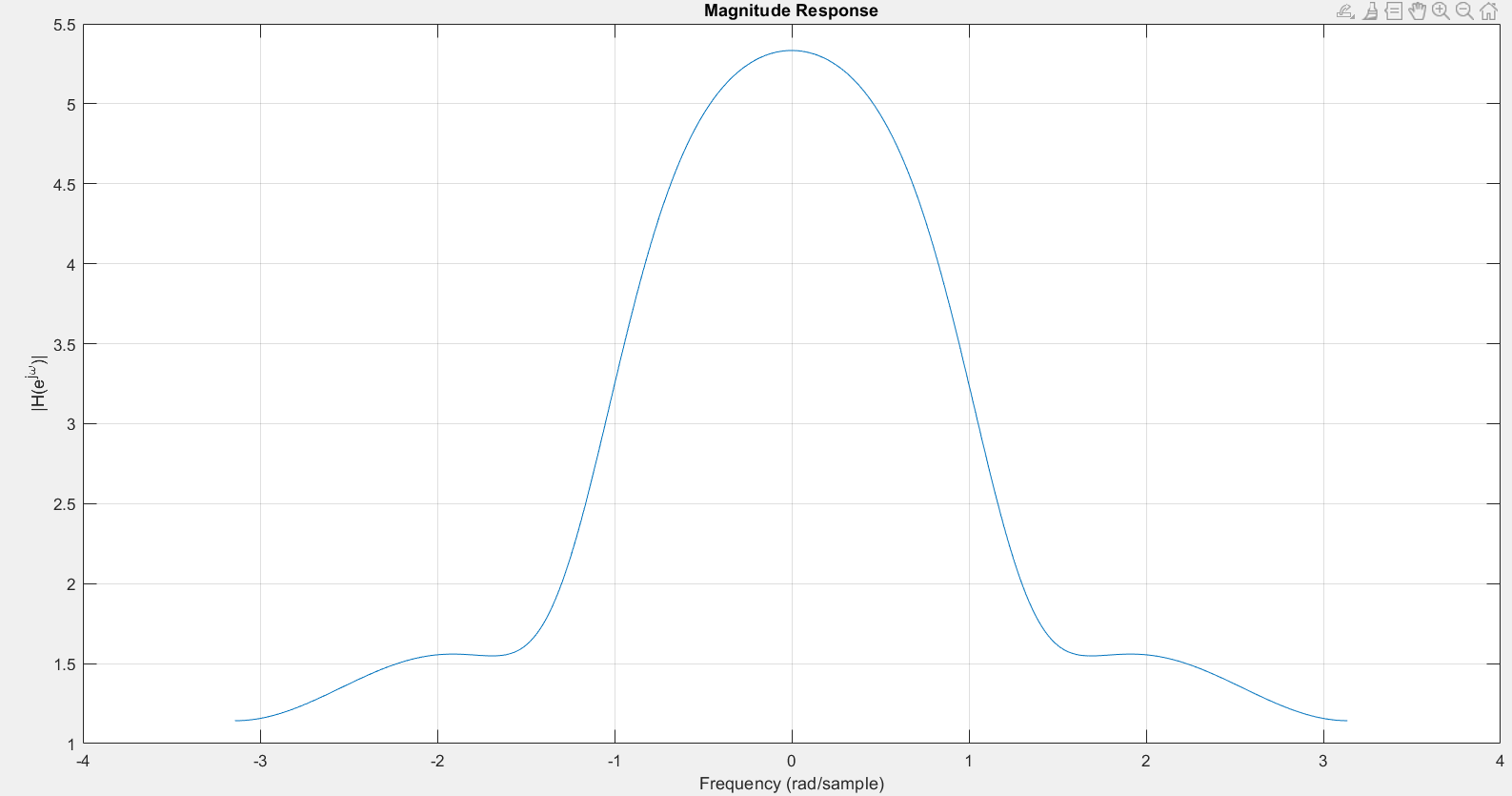
title('Magnitude Response');

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)));

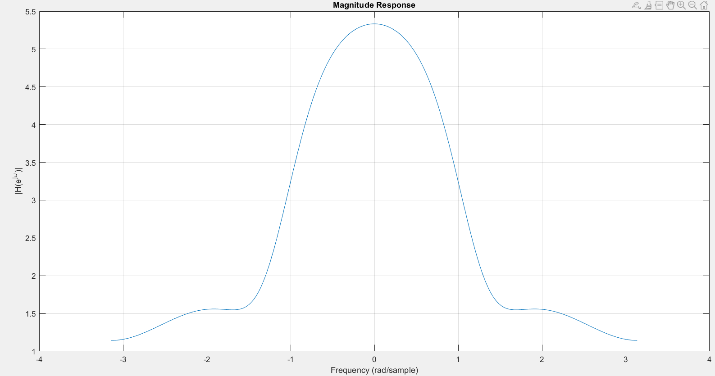
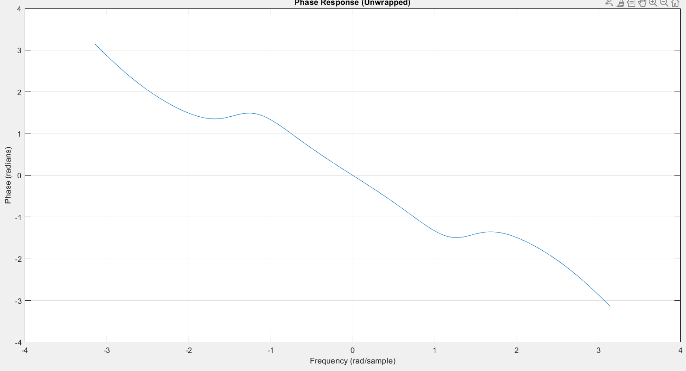
title('Phase Response (Unwrapped)');

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

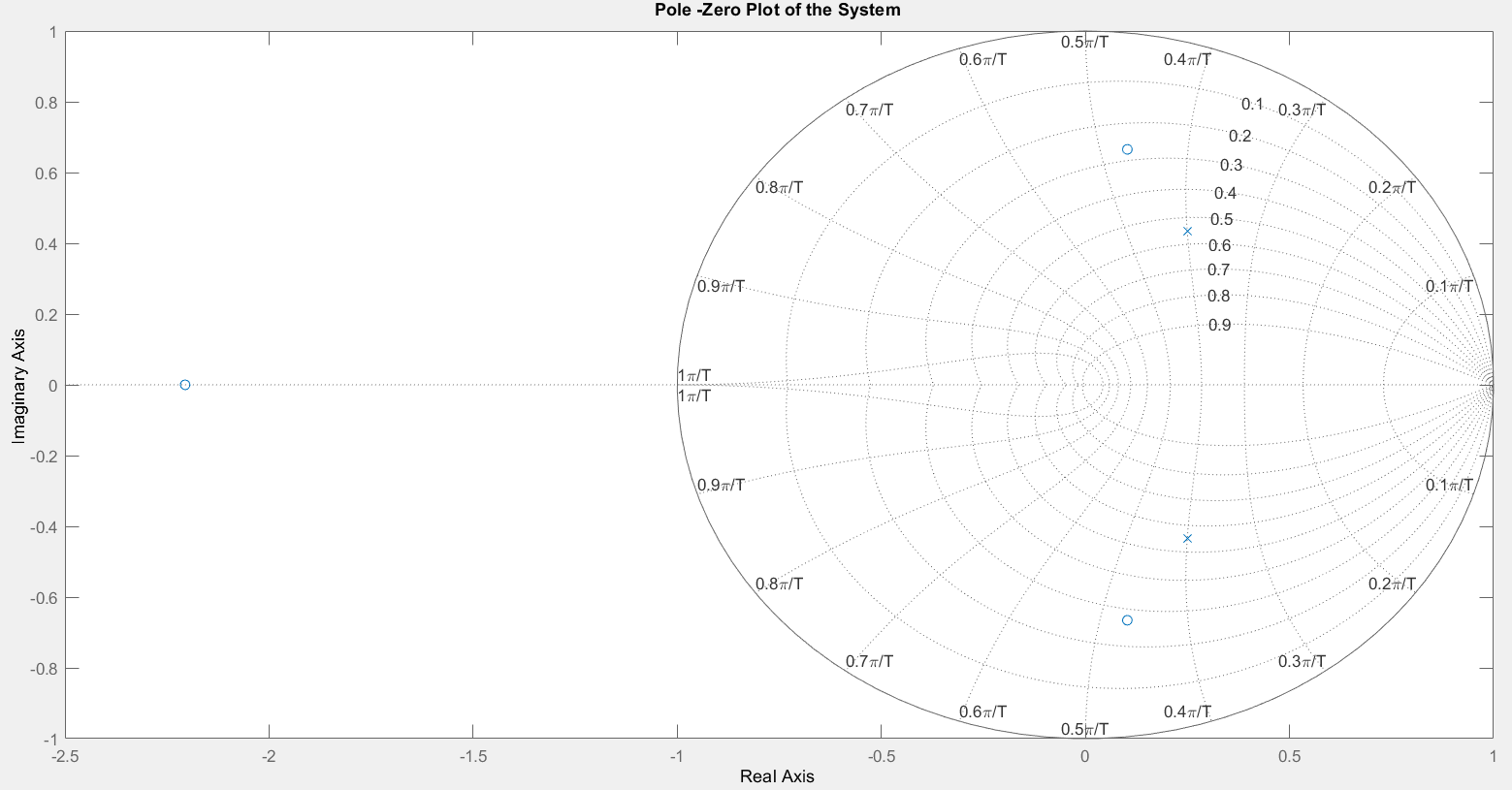
figure;

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');

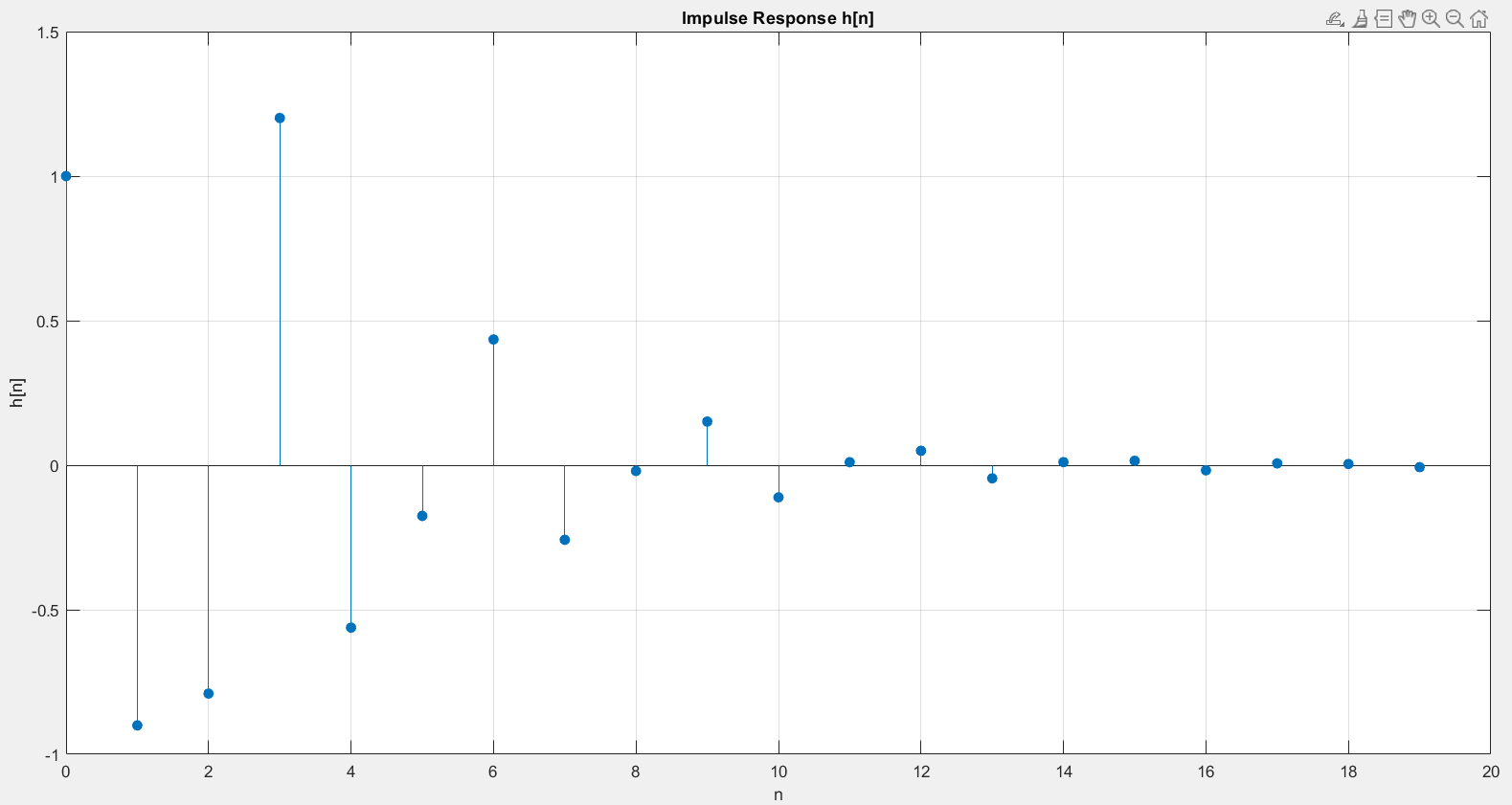
title('Impulse Response h[n]');

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

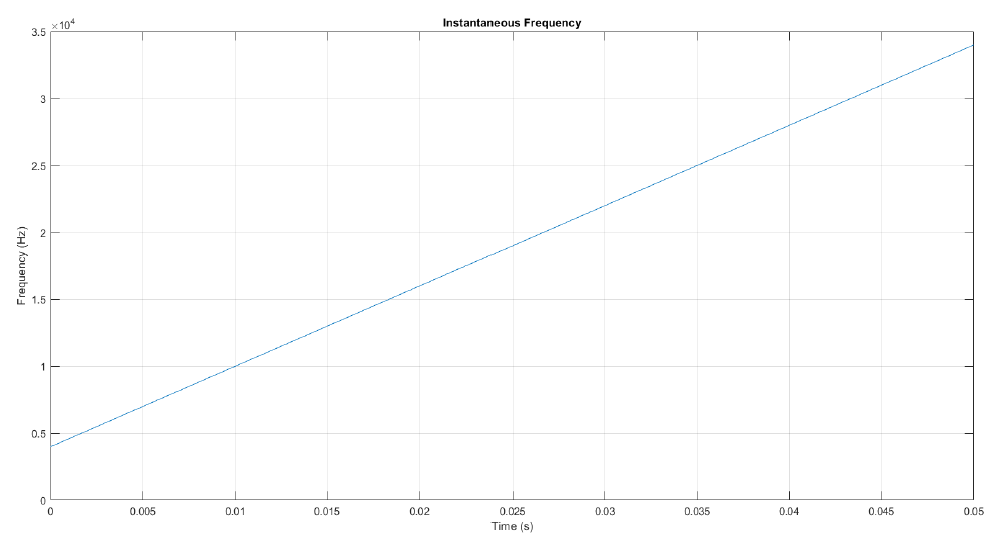
disp('No aliasing issue detected.');

end

***Part A:***

For the

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



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:***

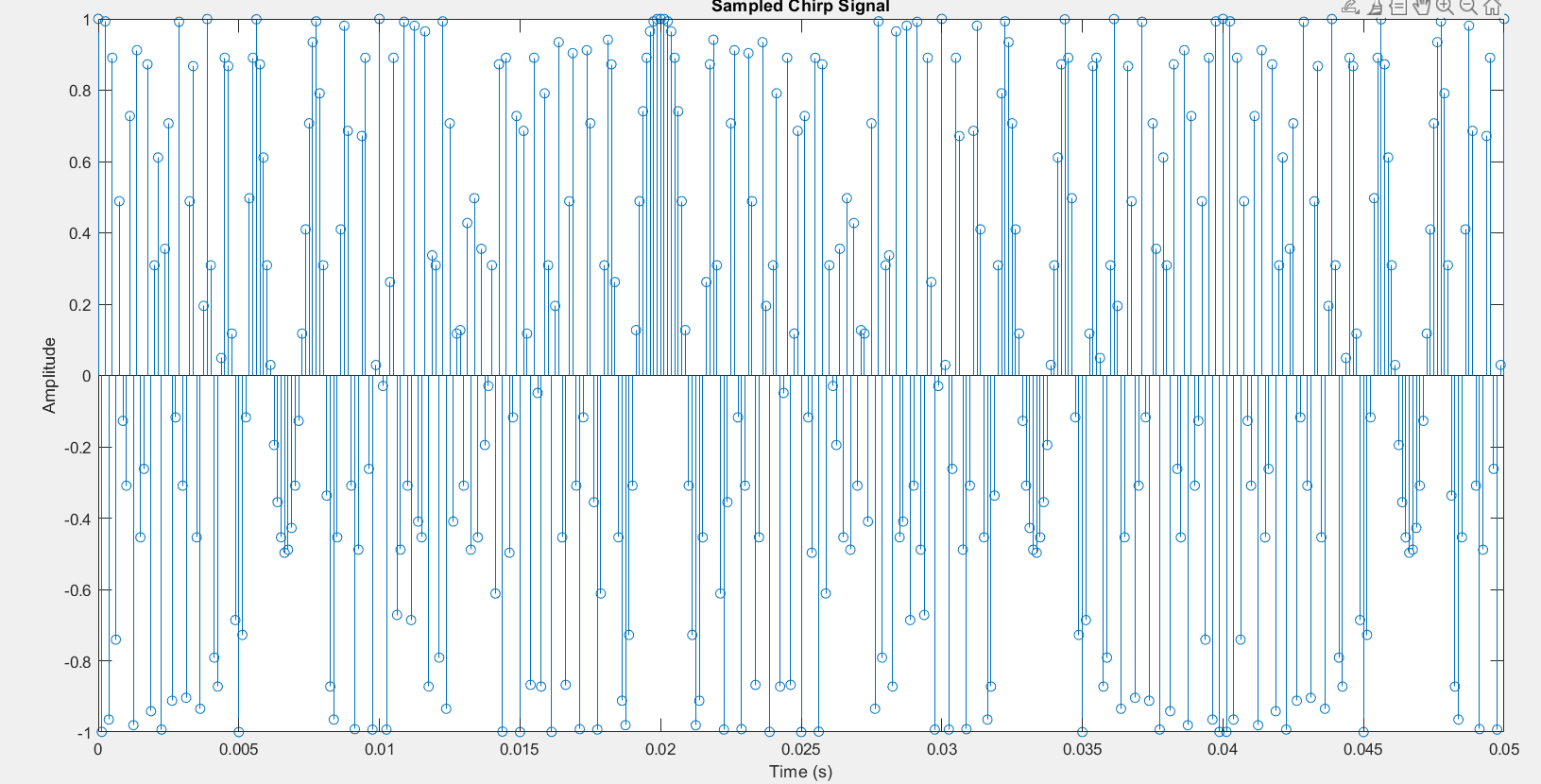
The instantaneous frequency f(t) of a linear chirp signal is given by . At t=0, the starting frequency is . At t=T=0.05 such tha t = T = 0.05s , the ending frequency is . Substituting the values, we get

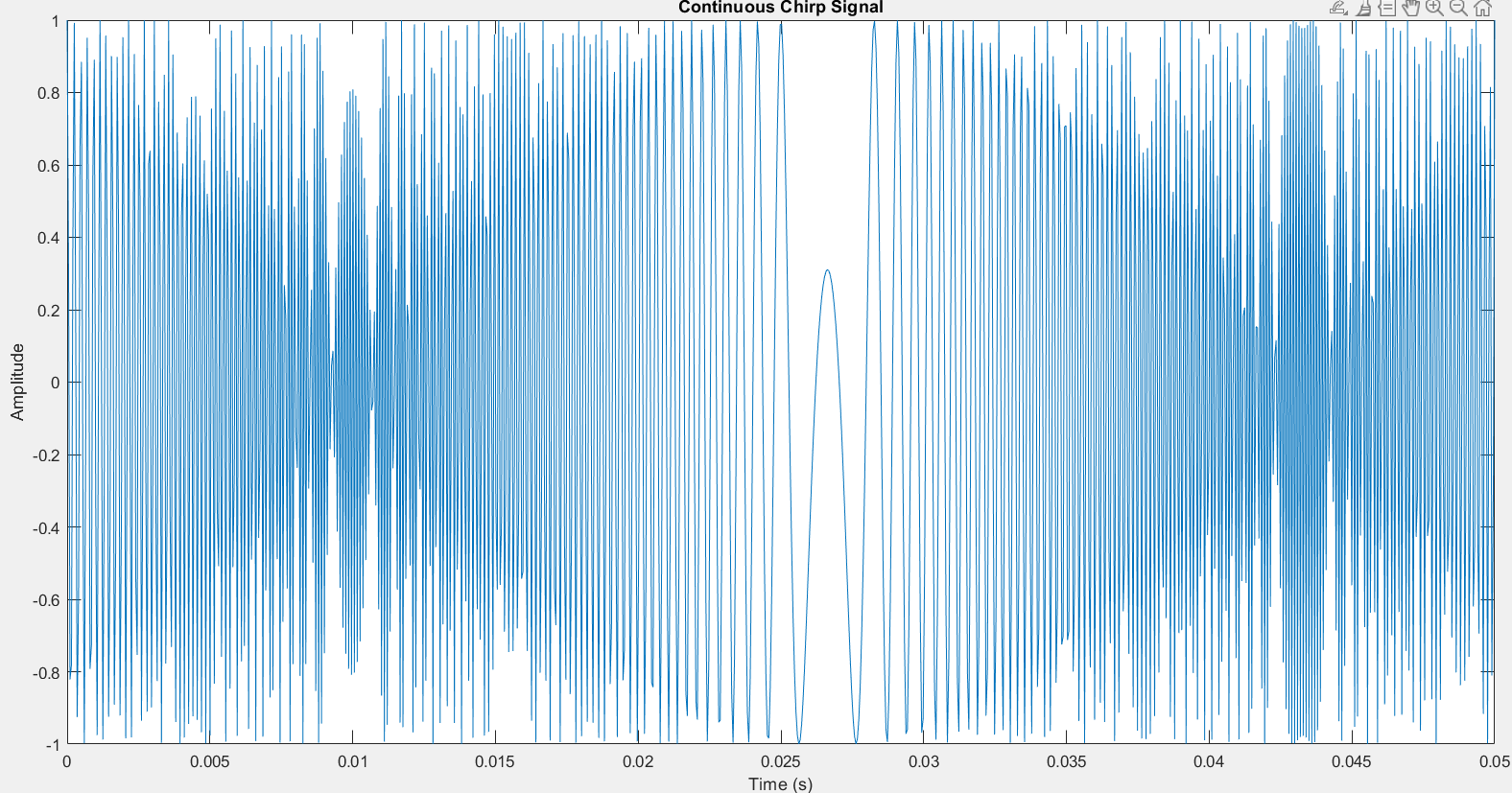
Therefore, the frequency sweep range for the chirp signal is from 4 kHz to 34 kHz over the 0.05-second interval, demonstrating the increase in frequency due to the positive sweep rate .

***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:

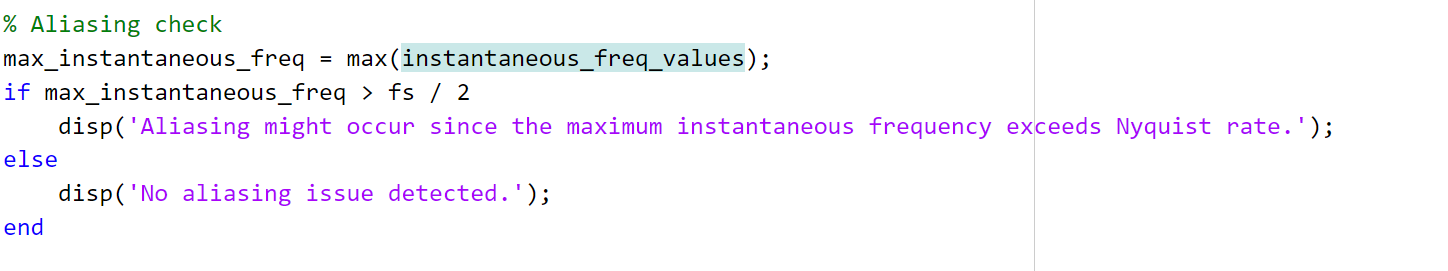




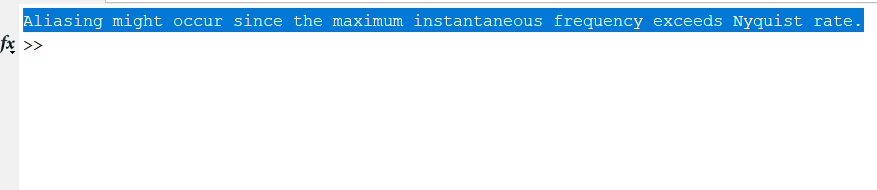
Aliasing can happen when the instantaneous frequency in the sampled signal exceeds the Nyquist frequency of 4 kHz. This results in frequency folding, as observed in the figure, when the frequency surpasses this limit. The phenomenon occurs due to the sampling frequency being too low to accurately capture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

***Part E:***

This is the code for aliasing check:



And this is the result:



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));

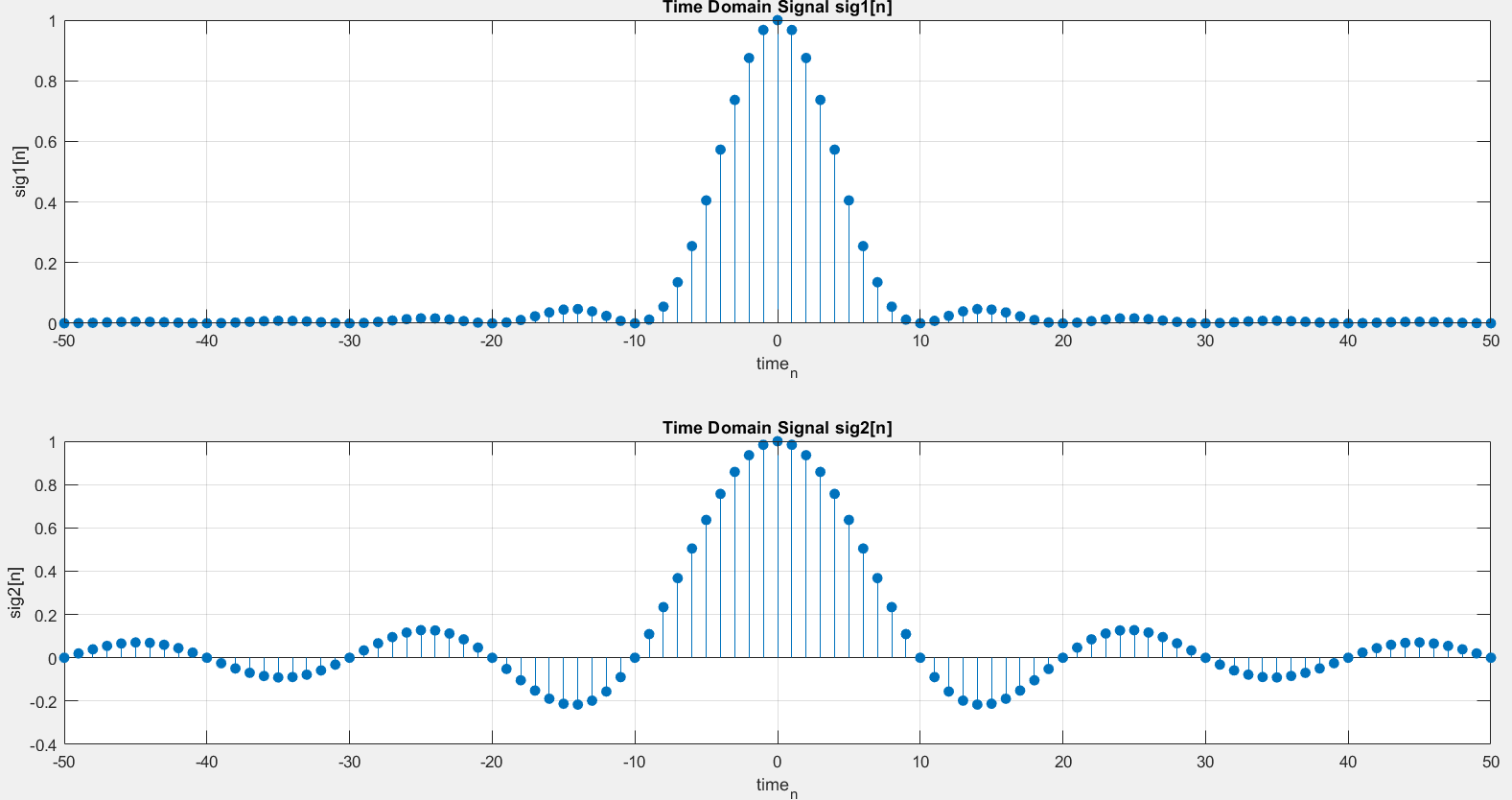
title('Magnitude of Fourier Transform |FT\\_mod\\_sig3(\omega)| for mod\\_sig3[n] = sig2[n] \cdot \sin(2 \pi \cdot 0.3 \cdot time\_n)');

xlabel('Frequency (\omega)');

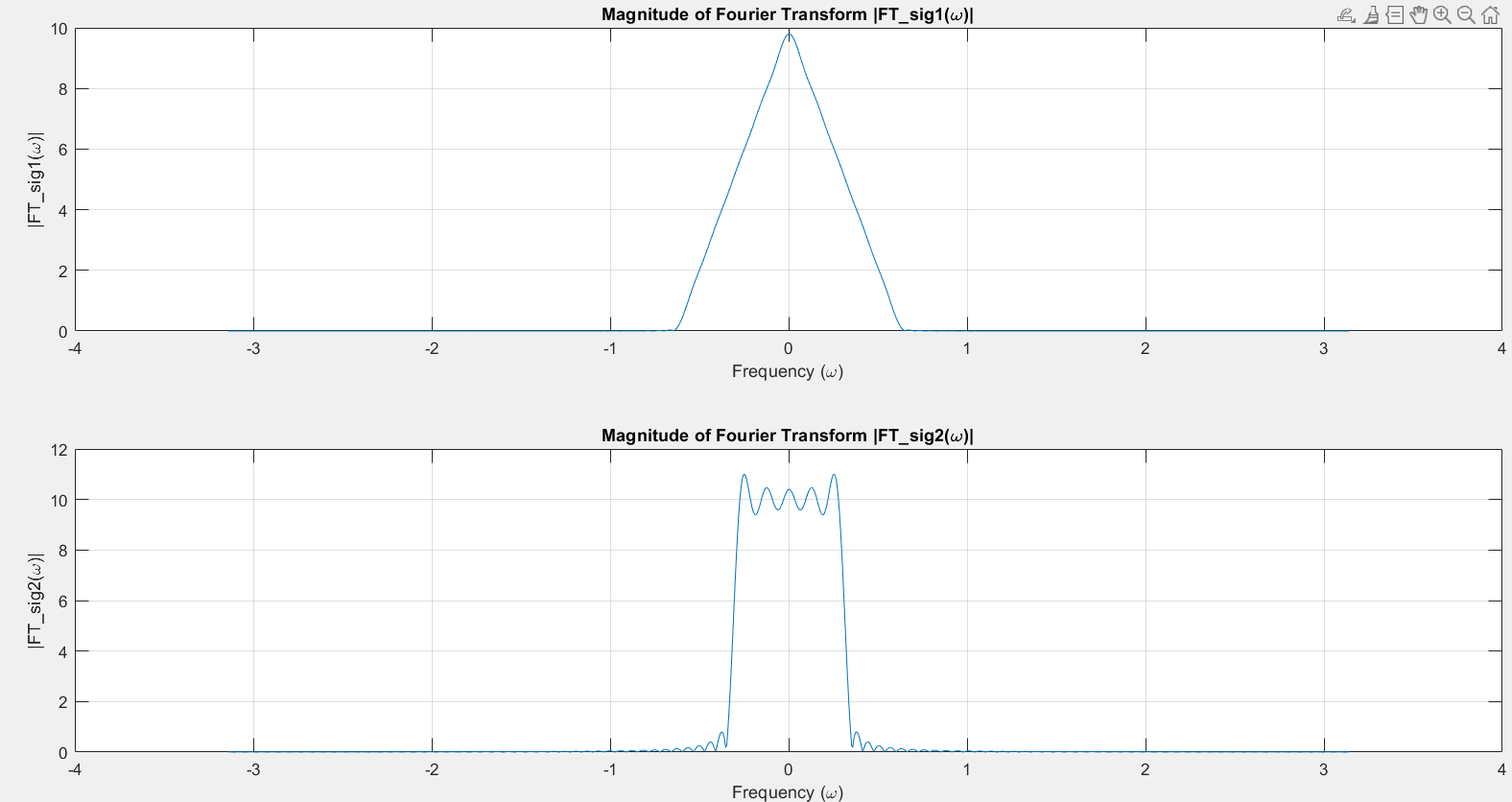
ylabel('|FT\\_mod\\_sig3(\omega)|');

grid on;

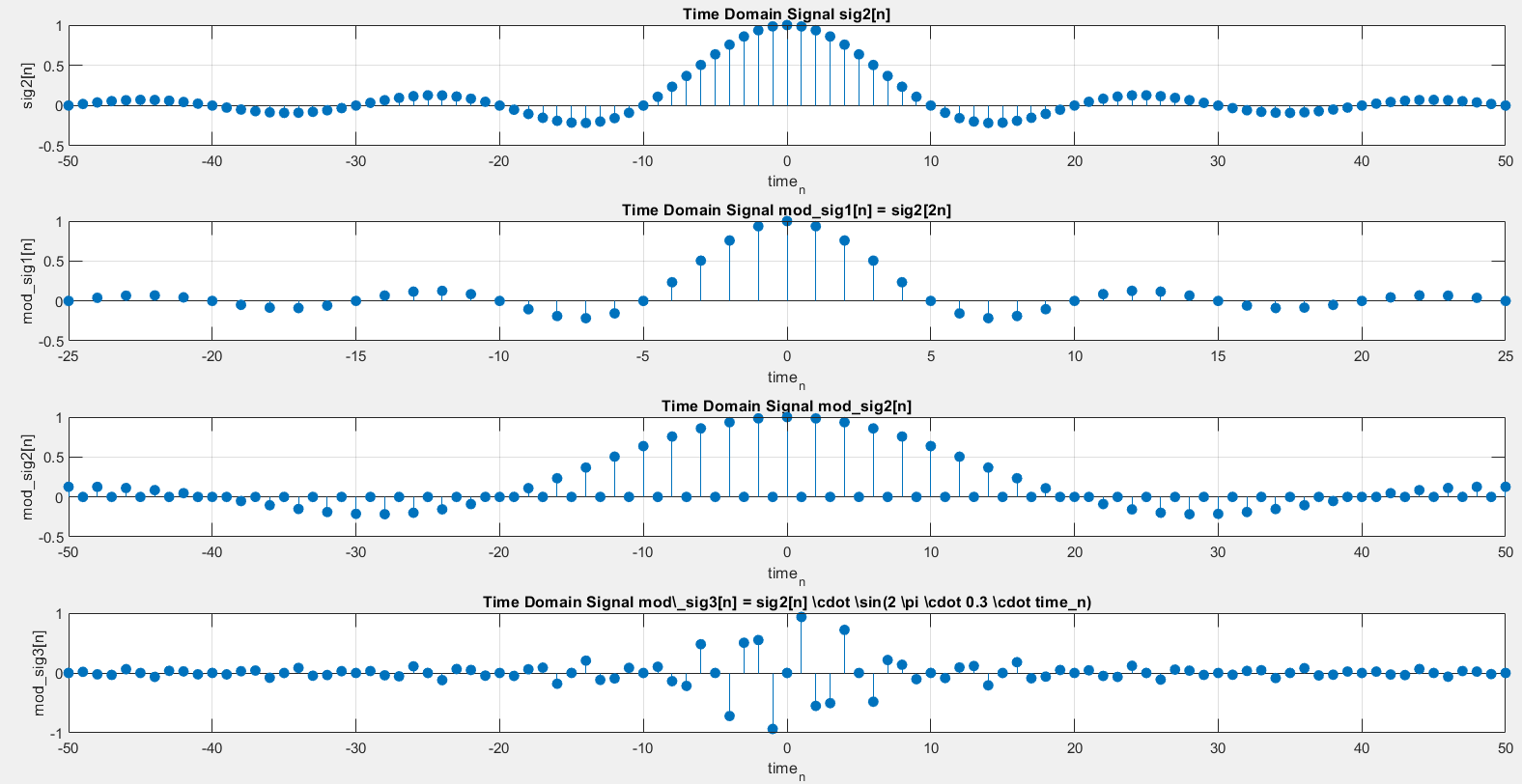
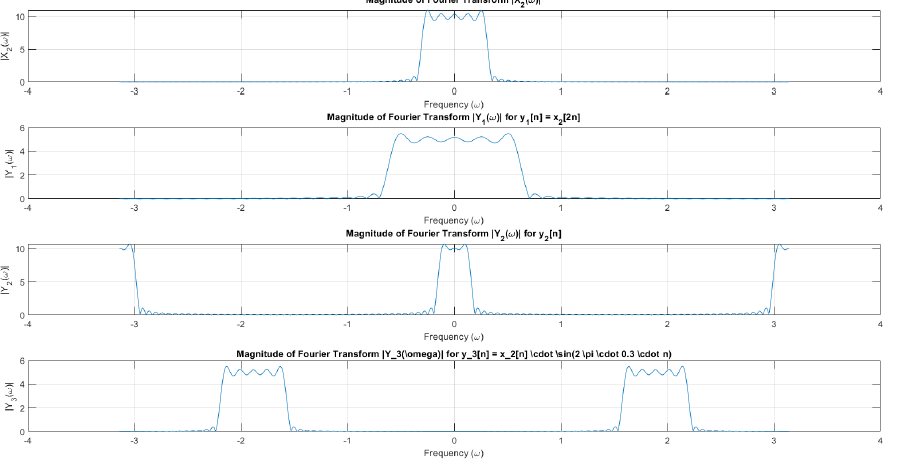
***PART A:*** Here are the signals x1[n] and x2[n]:



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



***PART B:***

Figures 1 and 2 show the time-domain signals and and their Fourier transforms. Figures 3 and 4 display the modified signals , and based on , and their Fourier transforms. This analysis allows us to compare how each transformation affects the frequency spectrum and signal shape.

The original signals and 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, , downsampling by a factor of 2 creates a sparser signal and causes aliasing in the frequency domain, introducing overlapping frequency components.

The second transformation, , 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, modulation by a sine wave results in a frequency shift. The frequency spectrum of 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

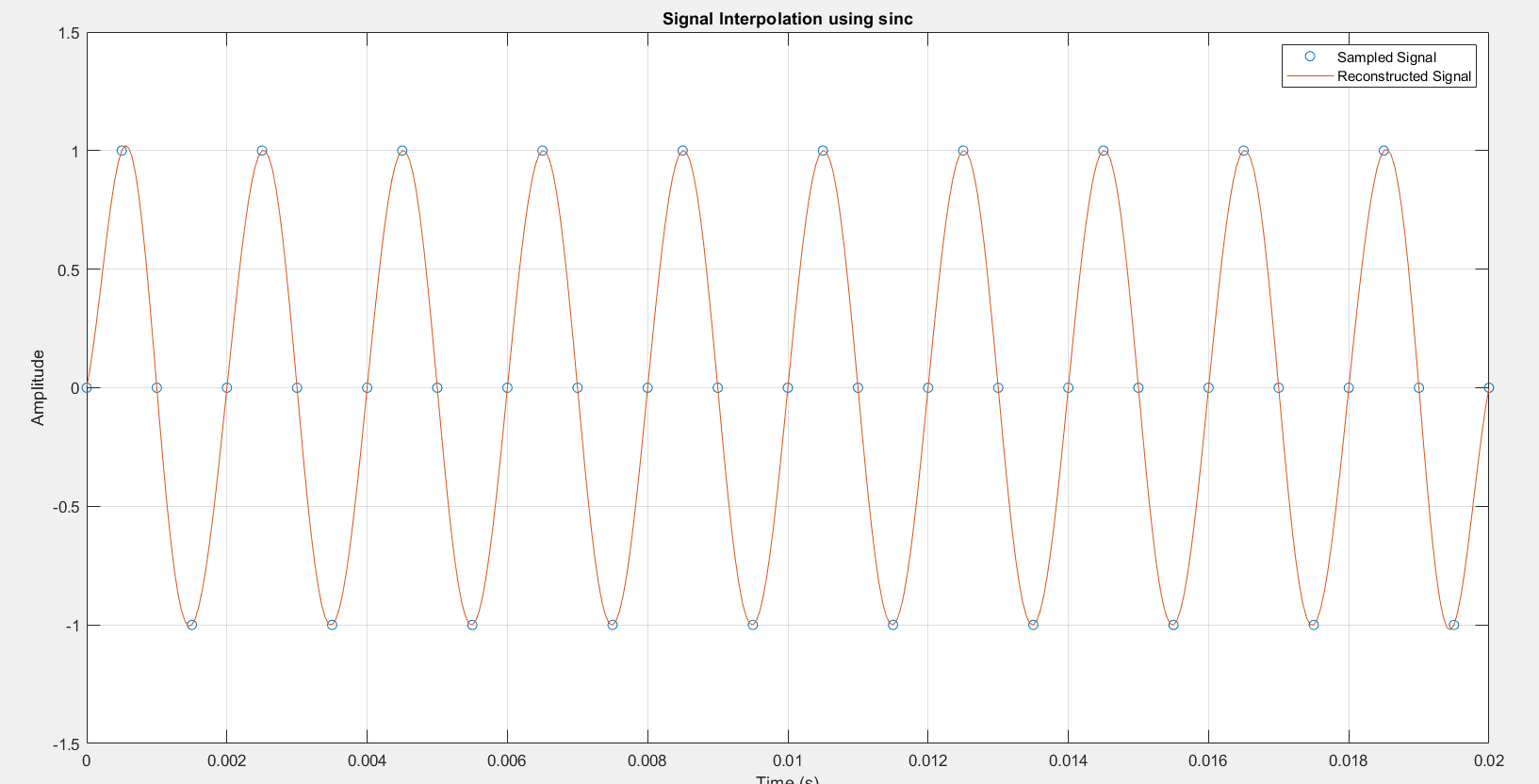
for n = 1:num\_samples

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

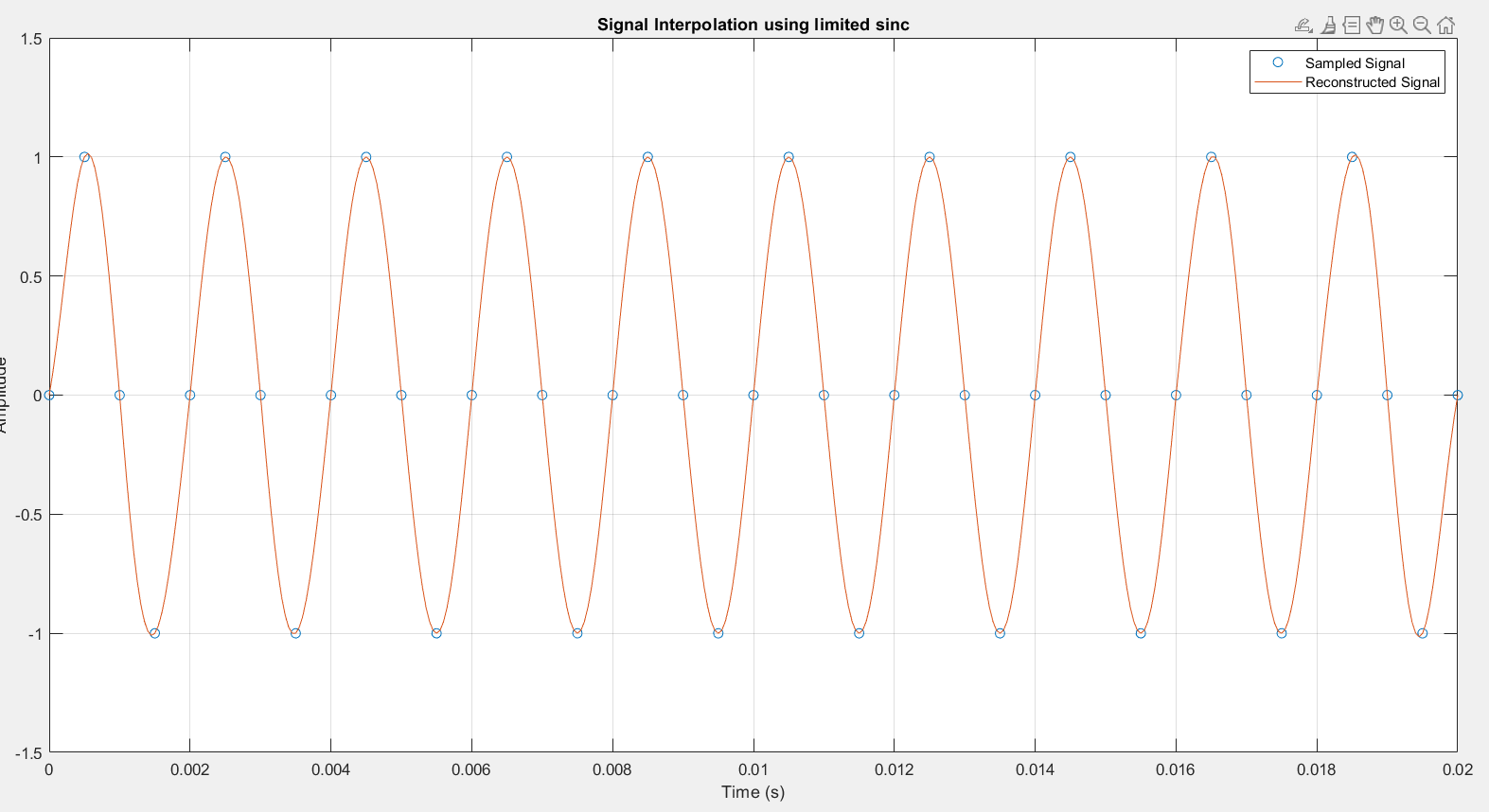
sinc\_values = (desired\_times - sample\_times(n)) / (sample\_times(2) - sample\_times(1));

limited\_sinc = sinc(sinc\_values) .\* (abs(sinc\_values) <= sinc\_lobe\_limit);

y\_reconstructed = y\_reconstructed + sampled\_signal(n) \* limited\_sinc;

end

end



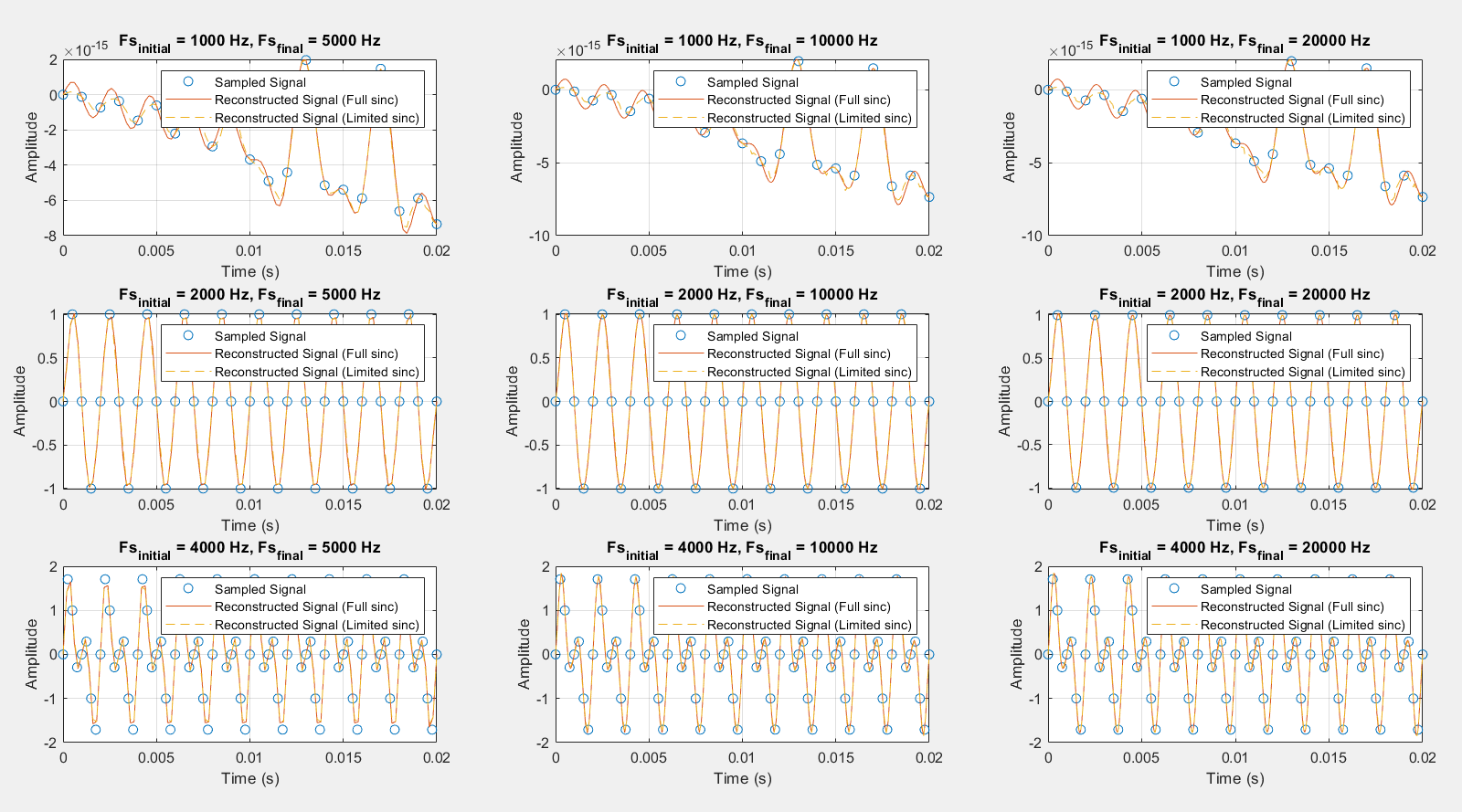
The reconstructed signal still fits the sampled data points.

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