AMERICAN UNIVERSITY OF BEIRUT

MAROUN SEMAAN FACULTY OF ENGINEERING & ARCHITECTURE

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

EECE 340 – Systems and signals

Project

Prof. Hadi Sarieddine

Nour Shammaa **Nour Fawaz**

Spring 2022-2023

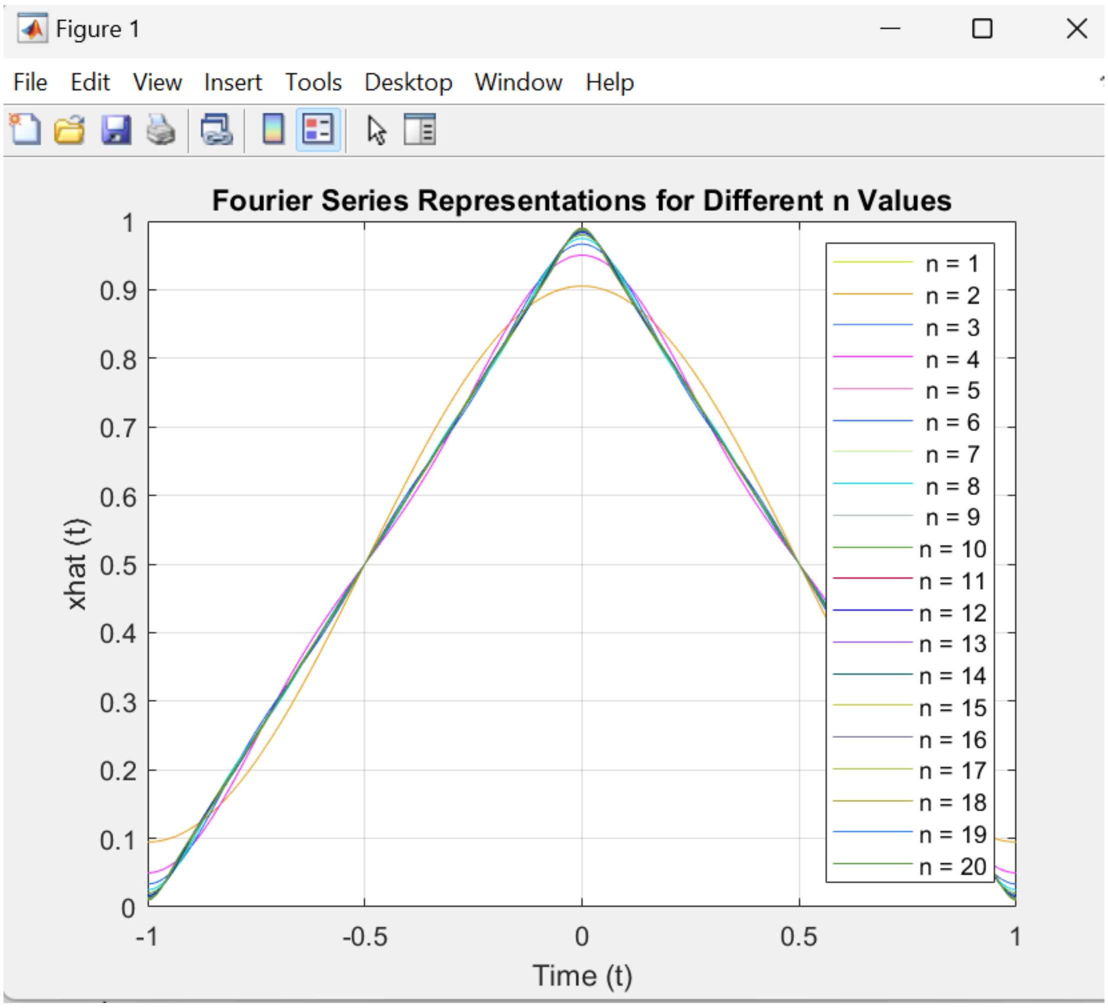
Part I:

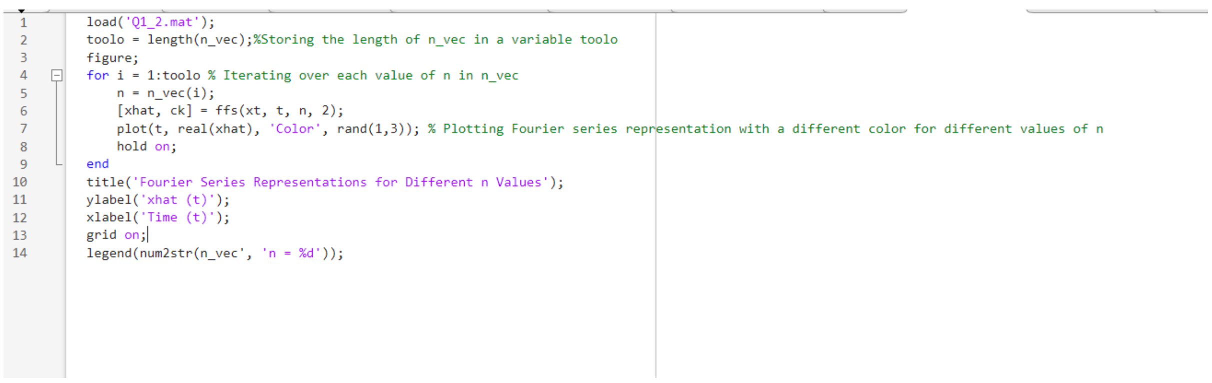
1.1

The second code that appears below is the implementation of the function that creates a finite Fourier time series of a time-limited signal. It takes four input parameters: xt (the time-limited signal), t (the time vector), n (the number of Fourier coefficients), and T (the period). The function calculates the Fourier coefficients ck and the reconstructed signal xhat using numerical integration based on the trapezoidal rule.

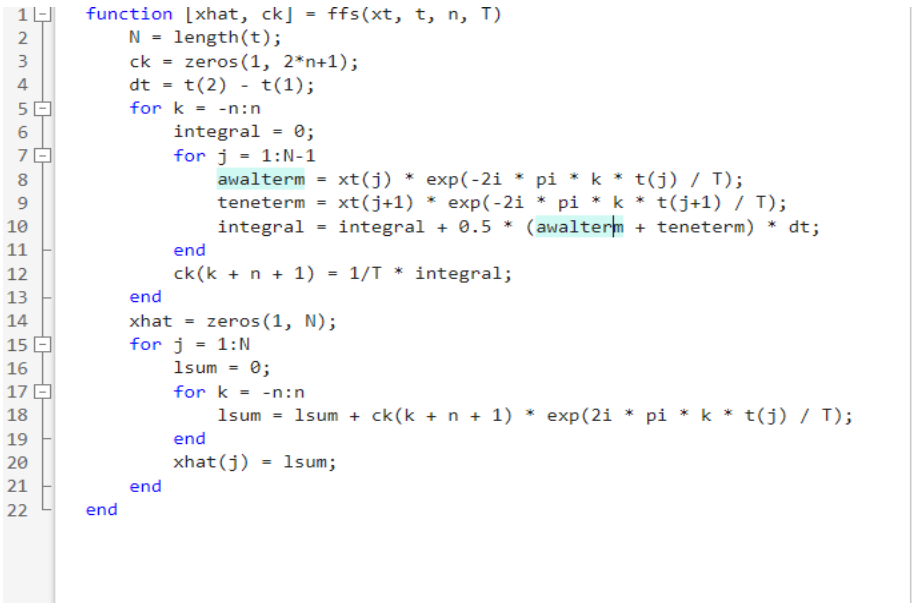
While the first one utilizes the function ffs to generate Fourier series representations for different values of n. It iterates over each value of n in n_vec and calls ffs to obtain xhat and ck. It then plots the real part of xhat in function of t, with each representation shown in a different color. The plot is labeled with appropriate titles, axes labels, and a legend indicating the corresponding values of n.

These functions returns both x^(t) and the ck's in the vectors xhat and ck, respectively.





```
load('Q1_2.mat');
toolo = length(n_vec);%Storing the length of n_vec in a variable toolo
figure;
for i = 1:toolo % Iterating over each value of n in n_vec
    n = n_vec(i);
    [xhat, ck] = ffs(xt, t, n, 2);
    plot(t, real(xhat), 'Color', rand(1,3)); % Plotting Fourier series representation with a different color for different values of n
    hold on;
end
title('Fourier Series Representations for Different n Values');
ylabel('xhat (t)');
xlabel('Time (t)');
grid on;
legend(num2str(n_vec', 'n = %d'));
```



```
function [xhat, ck] = ffs(xt, t, n, T)
    N = length(t);
    ck = zeros(1, 2*n+1);
    dt = t(2) - t(1);
    for k = -n:n
        integral = 0;
        for j = 1:N-1
            awalterm = xt(j) * exp(-2i * pi * k * t(j) / T);
            teneterm = xt(j+1) * exp(-2i * pi * k * t(j+1) / T);
            integral = integral + 0.5 * (awalterm + teneterm) * dt;
        end
        ck(k + n + 1) = 1/T * integral;
    end
    xhat = zeros(1, N);
    for j = 1:N
        lsum = 0;
        for k = -n:n
            lsum = lsum + ck(k + n + 1) * exp(2i * pi * k * t(j) / T);
        end
        xhat(j) = lsum;
    end
end
```

1.2

The objective of this part is to analyse the effects of changing the number of coefficients ‘n’ and the period ‘T’ on the Fourier series representation of a triangle signal ‘x(t)'.

A. Varying the Number of Coefficients ‘n’:

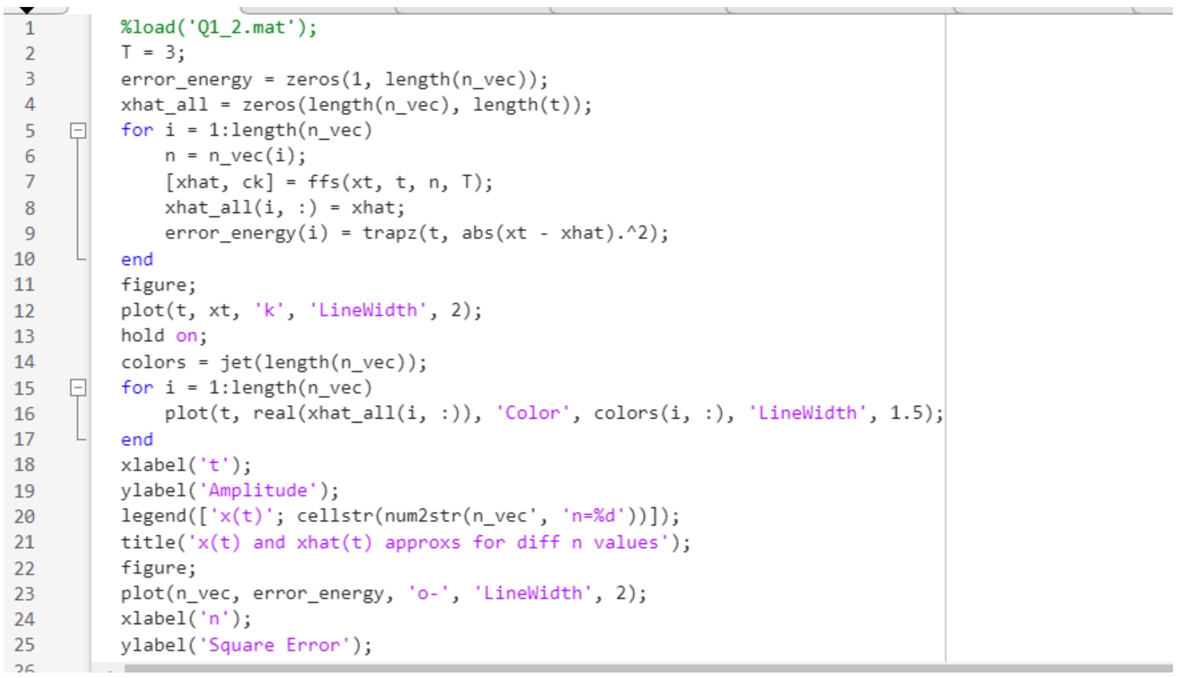
The Fourier series approximations ‘xhat(t)’ and coefficients ‘ck’ are calculated for different ‘n’ values. As ‘n’ increases, the approximations converge closer to the original signal ‘x(t)’. The square error of the representation decreases with increasing ‘n’, indicating better accuracy. However, the reduction in error becomes less significant for higher ‘n’ values.

B. Varying the Period ‘T’:

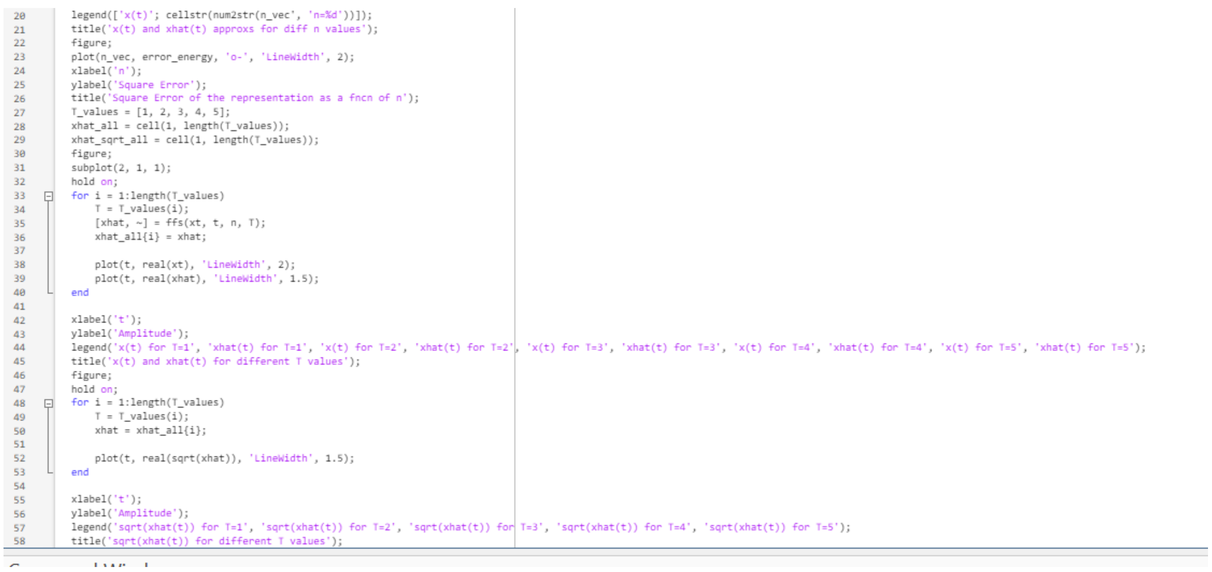
The Fourier series approximations ‘xhat(t)’ and coefficients ‘ck’ are calculated for different ‘T’ values. As ‘T’ increases, the approximations capture the broader features of the triangle signal. The square error of the representation decreases as ‘T’ increases, indicating improved accuracy. Extremely large ‘T’ values may introduce oversampling and unnecessary complexity.

Comment:
Concerning the square error as n increases, the error energy decreases, indicating a better representation of the original signal. Similarly, changing T affects the error energy, showing how different periods capture the signal's components differently.
The choice of n and T depends on the specific signal and the desired accuracy. Increasing n can lead to a more accurate representation, but it also increases the computational complexity. Selecting an appropriate T value involves considering the signal's frequency content and ensuring that it captures the relevant components.

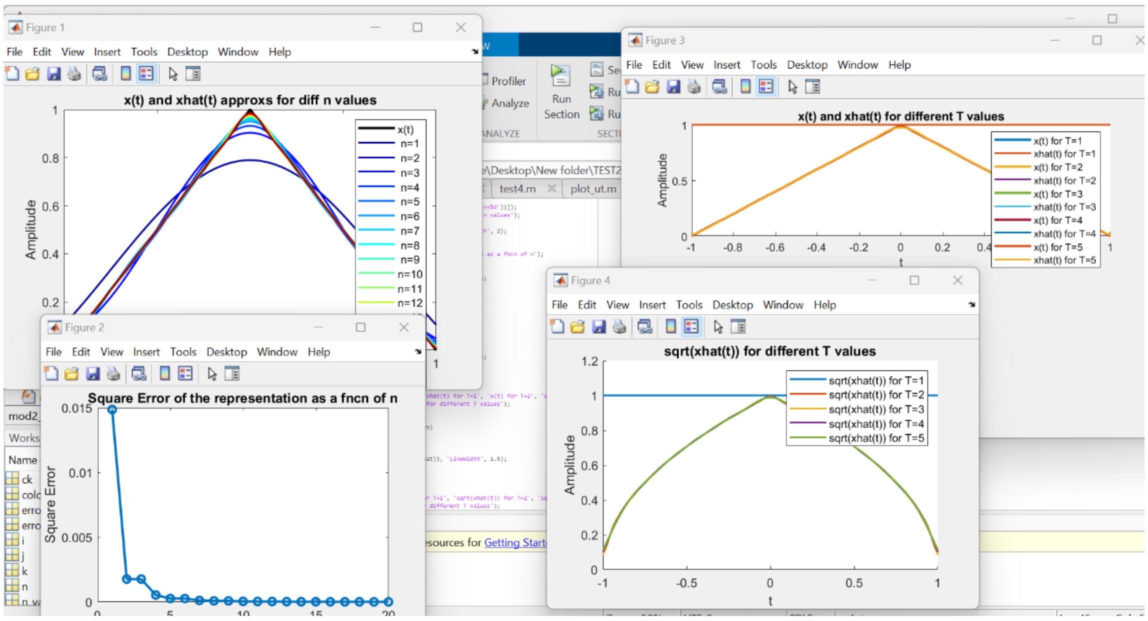
By analyzing the square error as a function of n and T, it is possible to determine the trade-off between accuracy and computational complexity. This analysis helps in selecting suitable values for n and T to achieve a desired level of approximation while managing computational resources effectively.



```matlab
%load('Q1_2.mat');
T = 3;
error_energy = zeros(1, length(n_vec));
xhat_all = zeros(length(n_vec), length(t));
for i = 1:length(n_vec)
    n = n_vec(i);
    [xhat, ck] = ffs(xt, t, n, T);
    xhat_all(i, :) = xhat;
    error_energy(i) = trapz(t, abs(xt - xhat).^2);
end
figure;
plot(t, xt, 'k', 'LineWidth', 2);
hold on;
colors = jet(length(n_vec));
for i = 1:length(n_vec)
    plot(t, real(xhat_all(i, :)), 'Color', colors(i, :), 'LineWidth', 1.5);
end
xlabel('t');
ylabel('Amplitude');
legend(['x(t)'; cellstr(num2str(n_vec', 'n=%d'))]);
title('x(t) and xhat(t) approxs for diff n values');
figure;
plot(n_vec, error_energy, 'o-', 'LineWidth', 2);
xlabel('n');
ylabel('Square Error');
```



```matlab
legend(['x(t)'; cellstr(num2str(n_vec', 'n=%d'))]);
title('x(t) and xhat(t) approxs for diff n values');
figure;
plot(n_vec, error_energy, 'o-', 'LineWidth', 2);
xlabel('n');
ylabel('Square Error');
title('Square Error of the representation as a fncn of n');
T_values = [1, 2, 3, 4, 5];
xhat_all = cell(1, length(T_values));
xhat_sqrt_all = cell(1, length(T_values));
figure;
subplot(2, 1, 1);
hold on;
for i = 1:length(T_values)
    T = T_values(i);
    [xhat, ~] = ffs(xt, t, n, T);
    xhat_all{i} = xhat;

    plot(t, real(xt), 'LineWidth', 2);
    plot(t, real(xhat), 'LineWidth', 1.5);
end

xlabel('t');
ylabel('Amplitude');
legend('x(t) for T=1', 'xhat(t) for T=1', 'x(t) for T=2', 'xhat(t) for T=2', 'x(t) for T=3', 'xhat(t) for T=3', 'x(t) for T=4', 'xhat(t) for T=4', 'x(t) for T=5', 'xhat(t) for T=5');
title('x(t) and xhat(t) for different T values');
figure;
hold on;
for i = 1:length(T_values)
    T = T_values(i);
    xhat = xhat_all{i};

    plot(t, real(sqrt(xhat)), 'LineWidth', 1.5);
end

xlabel('t');
ylabel('Amplitude');
legend('sqrt(xhat(t)) for T=1', 'sqrt(xhat(t)) for T=2', 'sqrt(xhat(t)) for T=3', 'sqrt(xhat(t)) for T=4', 'sqrt(xhat(t)) for T=5');
title('sqrt(xhat(t)) for different T values');
```



1.3

Fourier Transform (ft.m): The ft.m script defines a function for computing the Fourier Transform (FT) of a time domain signal. Input parameters: - - - xt: Time-domain signal values (row vector). t: Corresponding time vector (row vector). T: Total duration for scaling the frequency domain. Core Computation:

Frequency Vector Calculation: f = linspace(-1/(2*dt), 1/(2*dt), N); calculates the frequency vector which is evenly spaced and symmetric about zero, based on the sampling interval dt. Discrete Fourier Transform (DFT): The DFT is computed using a loop over each frequency component: Xf(k) = sum(xt .* exp(-1j * 2 * pi * f(k) * t)) * dt; This equation multiplies the signal with a complex exponential corresponding to each frequency and integrates (sums) over the time interval dt. Inverse Fourier Transform (ift.m): The ift.m script is used to compute the Inverse Fourier Transform (IFT) from the frequency domain back to the time domain. Input Parameters:

xf: Frequency-domain signal values (row vector). f: Corresponding frequency vector (row vector). W: Total bandwidth for scaling the time domain. Core Computation: - Time Vector Calculation: t = linspace(0, W, N); creates a time vector from 0 to the bandwidth W, evenly spaced. - Inverse Discrete Fourier Transform (IDFT): The IDFT is computed similarly to the DFT but uses a conjugate complex exponential: xt(k) = sum(xf .* exp(1j * 2 * pi * f * t(k))) * df; This calculates the inverse transformation by summing over the frequency components weighted by their respective exponential terms.

Detailed Analysis of MATLAB Code for Rect/Sinc Duality Test Case: This MATLAB script demonstrates the Fourier Transform (FT) and Inverse Fourier Transform (IFT) on a rectangular pulse signal. It illustrates the rect/sinc duality, showing how a time-domain rectangular pulse transforms into a sinc function in the frequency domain, and how it can be reconstructed back into the time domain.

Testing Parameters Definition:

Sampling Frequency and Duration: fs = 1000; % Sampling frequency T = 2; % Total duration in seconds These parameters set up the sampling frequency (fs) at 1000 Hz and the total duration (T) of the signal for 2 seconds. Time Vector and Bandwidth: t = linspace(0, T, fs*T); % Time vector W = 1/T; % Bandwidth The time vector t is generated using linspace, creating a linearly spaced vector from 0 to T, with fs*T points. The bandwidth W is calculated as the inverse of the duration T.

Pulse Parameters and Signal Definition: rect_width = 0.5; % Width of the rectangular pulse xt = double(abs(t) <= rect_width/2); A rectangular pulse of width 0.5 seconds is defined by setting values to 1 (or True) where the absolute value of t is less than or equal to half the pulse width, and to 0 otherwise.

Computation of Fourier Transform: [Xf, f] = ft(xt, t, T); The Fourier Transform of the rectangular pulse xt is calculated using the ft function defined previously. This function returns the frequency-domain representation Xf and the corresponding frequency vector f.

Reconstruction Using IFT: [xt_reconstructed, t_reconstructed] = ift(Xf, f, W); Using the ift function, the script reconstructs the signal from its Fourier Transform. This step verifies the accuracy of the Fourier and Inverse Fourier transforms.

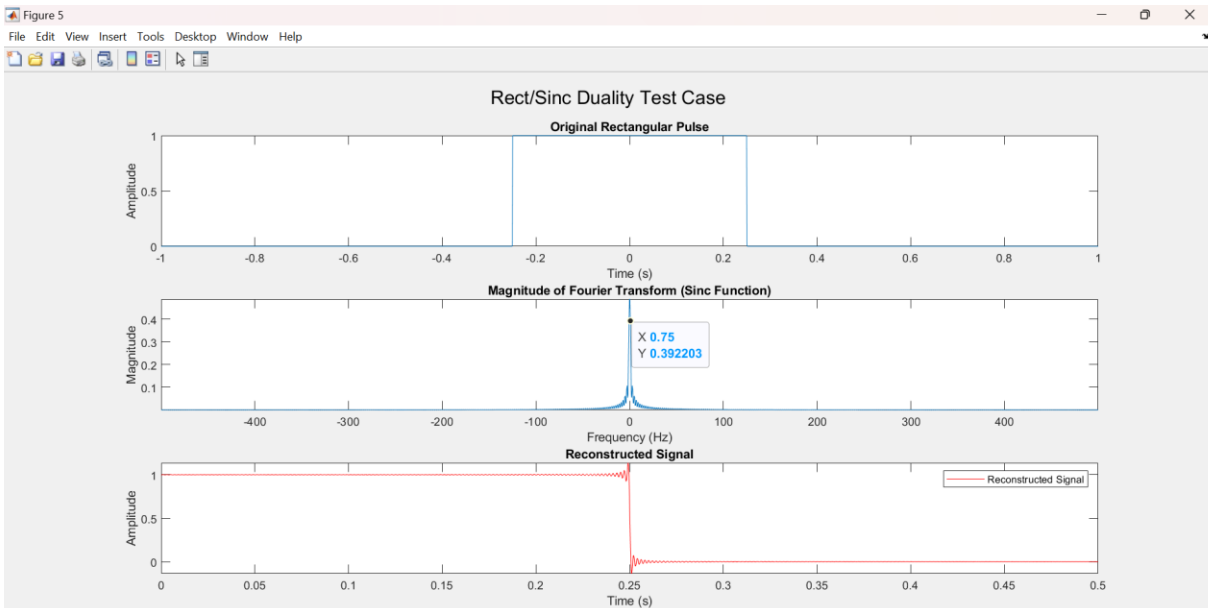
Visualization:

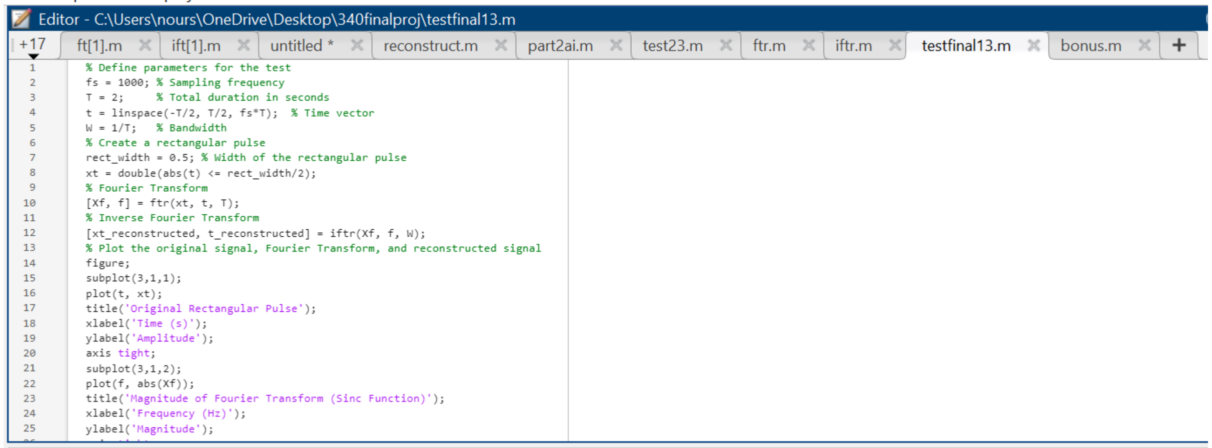
Plotting the Results:

Original Rectangular Pulse: Displays the time-domain signal.

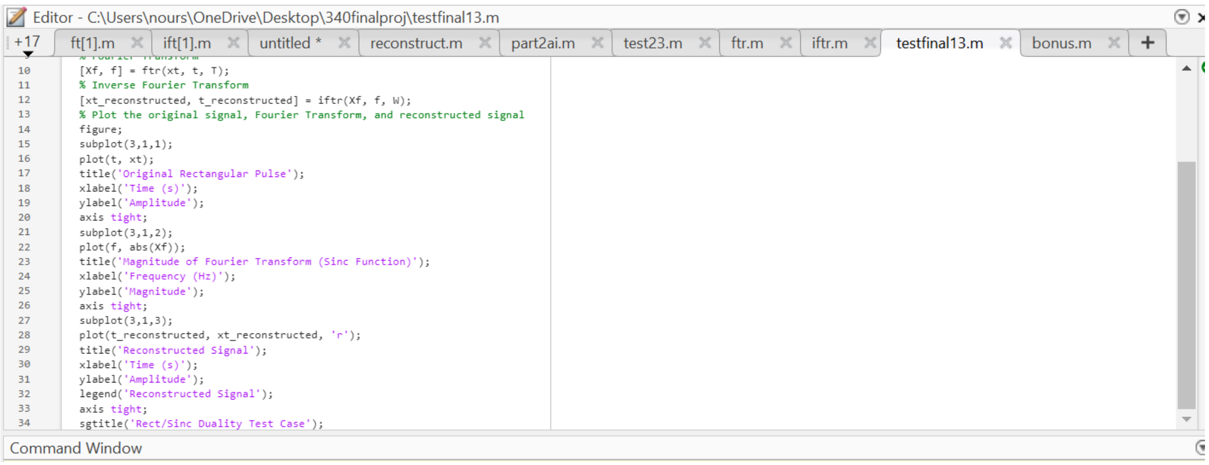
Magnitude of Fourier Transform: Shows the magnitude of the Fourier Transform, depicting a sinc function due to the duality between the rectangular pulse in the time domain and the sinc function in the frequency domain.

Reconstructed Signal: Displays the signal reconstructed from the Fourier Transform to validate the inverse operation. Each subplot is properly labeled with titles, axes labels, and legends, providing a clear and detailed visualization of the transformations

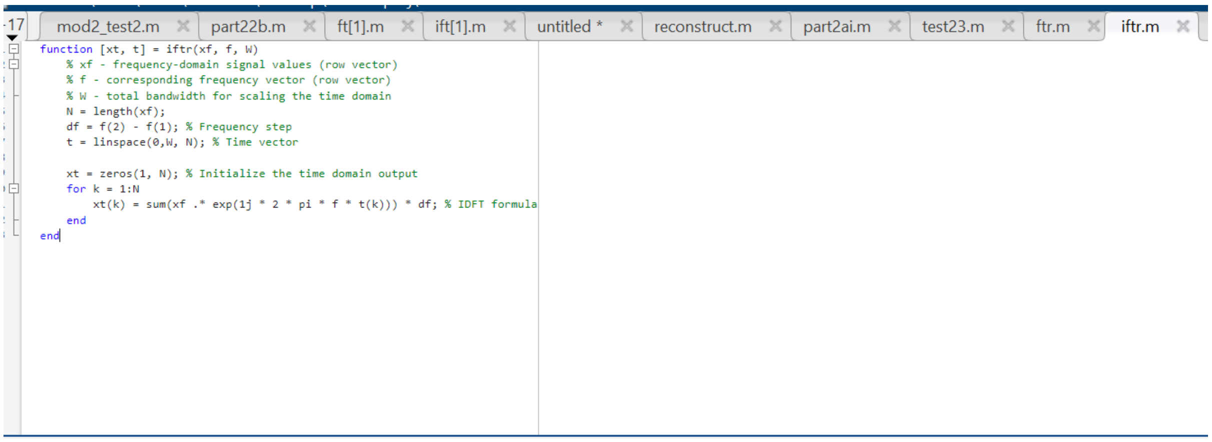



Editor - C:\Users\nours\OneDrive\Desktop\340finalproj\testfinal13.m
+17 ft[1].m ift[1].m untitled * reconstruct.m part2ai.m test23.m ftr.m iftr.m testfinal13.m bonus.m

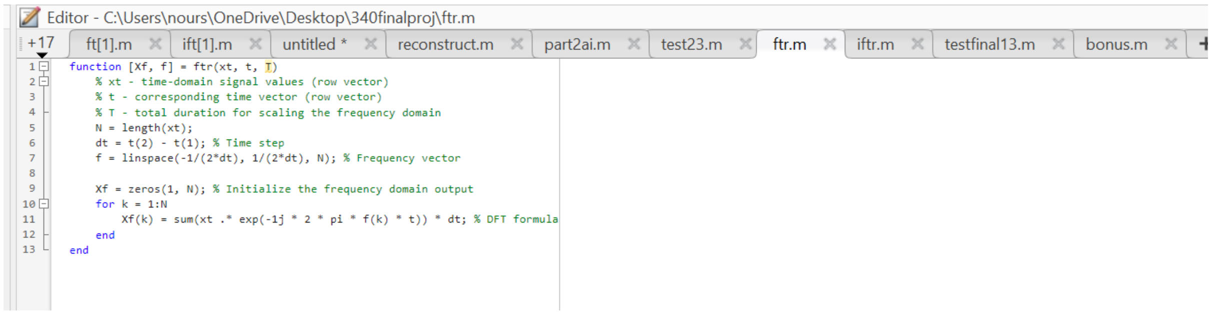
```
% Define parameters for the test
fs = 1000; % Sampling frequency
T = 2;     % Total duration in seconds
t = linspace(-T/2, T/2, fs*T);  % Time vector
W = 1/T;   % Bandwidth
% Create a rectangular pulse
rect_width = 0.5; % Width of the rectangular pulse
xt = double(abs(t) <= rect_width/2);
% Fourier Transform
[Xf, f] = ftr(xt, t, T);
% Inverse Fourier Transform
[xt_reconstructed, t_reconstructed] = iftr(Xf, f, W);
% Plot the original signal, Fourier Transform, and reconstructed signal
figure;
subplot(3,1,1);
plot(t, xt);
title('Original Rectangular Pulse');
xlabel('Time (s)');
ylabel('Amplitude');
axis tight;
subplot(3,1,2);
plot(f, abs(Xf));
title('Magnitude of Fourier Transform (Sinc Function)');
xlabel('Frequency (Hz)');
ylabel('Magnitude');
```


Editor - C:\Users\nours\OneDrive\Desktop\340finalproj\testfinal13.m
+17 ft[1].m ift[1].m untitled * reconstruct.m part2ai.m test23.m ftr.m iftr.m testfinal13.m bonus.m
Command Window

```
[Xf, f] = ftr(xt, t, T);
% Inverse Fourier Transform
[xt_reconstructed, t_reconstructed] = iftr(Xf, f, W);
% Plot the original signal, Fourier Transform, and reconstructed signal
figure;
subplot(3,1,1);
plot(t, xt);
title('Original Rectangular Pulse');
xlabel('Time (s)');
ylabel('Amplitude');
axis tight;
subplot(3,1,2);
plot(f, abs(Xf));
title('Magnitude of Fourier Transform (Sinc Function)');
xlabel('Frequency (Hz)');
ylabel('Magnitude');
axis tight;
subplot(3,1,3);
plot(t_reconstructed, xt_reconstructed, 'r');
title('Reconstructed Signal');
xlabel('Time (s)');
ylabel('Amplitude');
legend('Reconstructed Signal');
axis tight;
sgtitle('Rect/Sinc Duality Test Case');
```



```
function [xt, t] = iftr(xf, f, W)
    % xf - frequency-domain signal values (row vector)
    % f - corresponding frequency vector (row vector)
    % W - total bandwidth for scaling the time domain
    N = length(xf);
    df = f(2) - f(1); % Frequency step
    t = linspace(0,W, N); % Time vector

    xt = zeros(1, N); % Initialize the time domain output
    for k = 1:N
        xt(k) = sum(xf .* exp(1j * 2 * pi * f * t(k))) * df; % IDFT formula
    end
end
```



```
function [Xf, f] = ftr(xt, t, T)
    % xt - time-domain signal values (row vector)
    % t - corresponding time vector (row vector)
    % T - total duration for scaling the frequency domain
    N = length(xt);
    dt = t(2) - t(1); % Time step
    f = linspace(-1/(2*dt), 1/(2*dt), N); % Frequency vector

    Xf = zeros(1, N); % Initialize the frequency domain output
    for k = 1:N
        Xf(k) = sum(xt .* exp(-1j * 2 * pi * f(k) * t)) * dt; % DFT formula
    end
end
```

Part 2:

2.1

1. my_sinc Function:

The my_sinc function calculates the sinc function for a given input vector ‘x’. It handles the case where `x` is zero to avoid division by zero. The function returns a vector 'y’ with the sinc values for the corresponding elements of 'x’. This function is used in the reconstruction process.

2. sample Function:

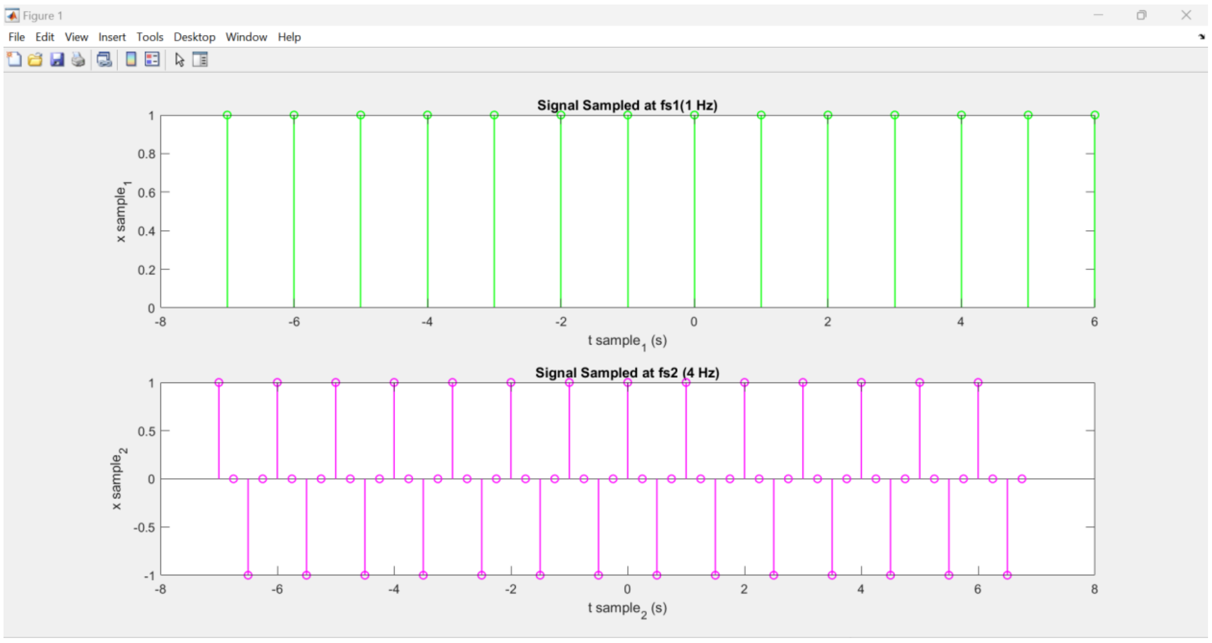
The sample   function samples the continuous-time signal xt at the specified sampling rate ‘fs’. It calculates the sampling period ‘Ts’ based on the inverse of the sampling rate. It generates a vector sample_indices that contains the indices of the sampled data points. These indices are calculated based on the sampling period and the time spacing between consecutive elements in 't'. It samples the signal 'xt’ by selecting the elements of ‘xt’ corresponding to the indices in ‘sample_indices’. It samples the time vector ‘t’ in the same way, generating the sampled time vector ‘t_sample’. The function returns the sampled time vector ‘t_sample’ and the sampled signal ‘x_sample’.

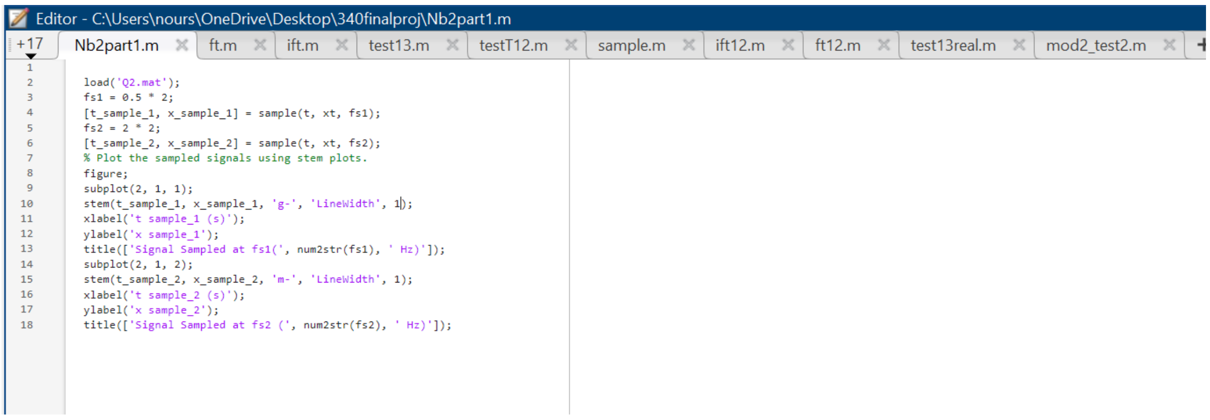
3. ‘reconstruct’ Function:

The ‘reconstruct’ function takes three inputs: ’t’_sample`, and ‘fs’. It reconstructs the continuous-time signal from the sampled data using the sinc interpolation method. It calculates the length of the sampled signal ‘x_sample’ and the sampling period ‘Ts’ based on the sampling rate ‘fs’. It initializes the ‘recon’ variable as a vector of zeros with the same size as the time vector `t`. This variable will store the reconstructed signal. It iterates over a range that spans from `-n` to `n`, where `n` is the length of ‘x_sample’. Inside the loop, it checks if the current index is within the valid range of the sampled signal. If the condition is satisfied, it adds to the ‘recon’ variable the product of the corresponding sample ‘x_sample(k+n+1)’ and the sinc function evaluated at ‘(t - k*Ts) / Ts’. The function returns the reconstructed signal.

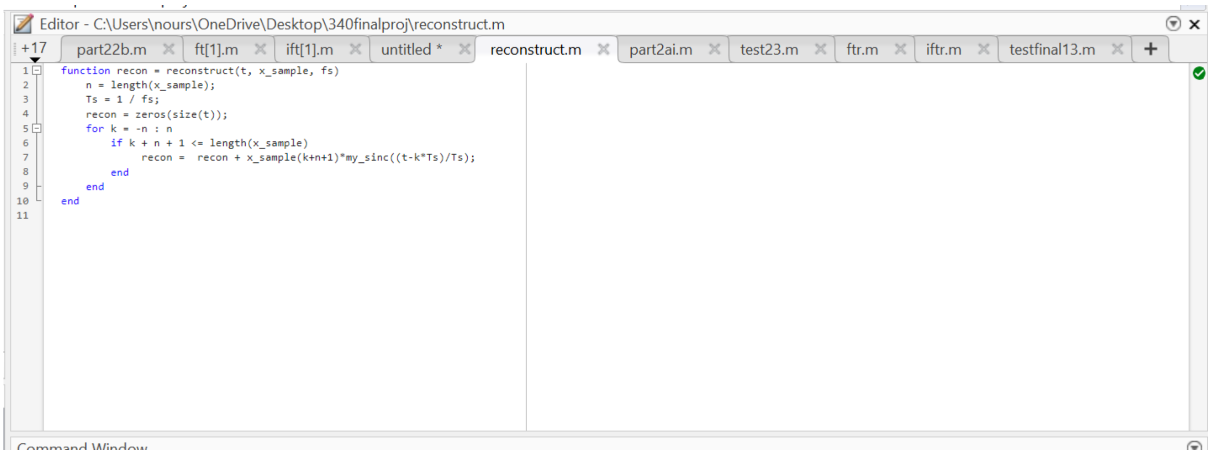
4. Main Script:

The main script demonstrates the sampling process by calling the `sample` function twice, with sampling rates of ‘0.5 * fs’ and ‘2 * fs’. It then calls the ‘reconstruct’ function to reconstruct the original signal from the sampled data using the sinc interpolation method. Finally, it plots the original signal, the sampled signals, and the reconstructed signals for both sampling rates.

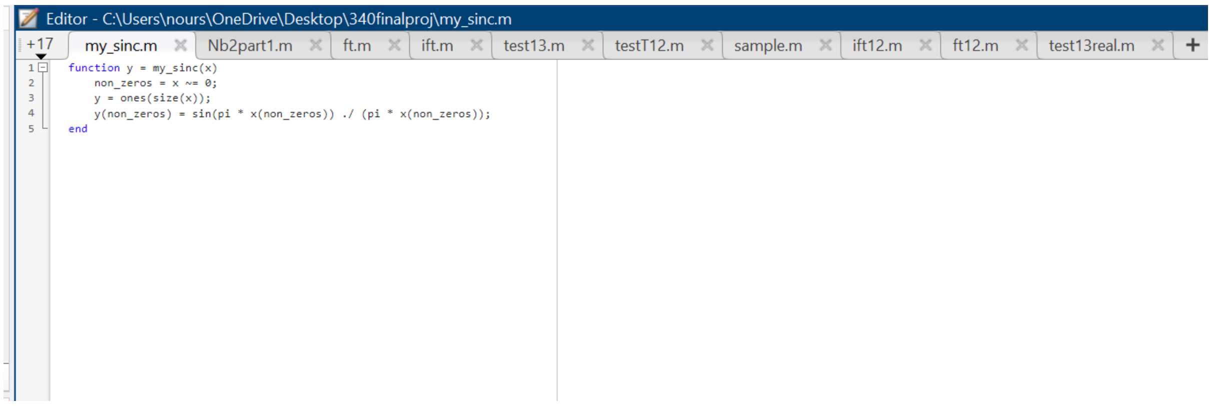




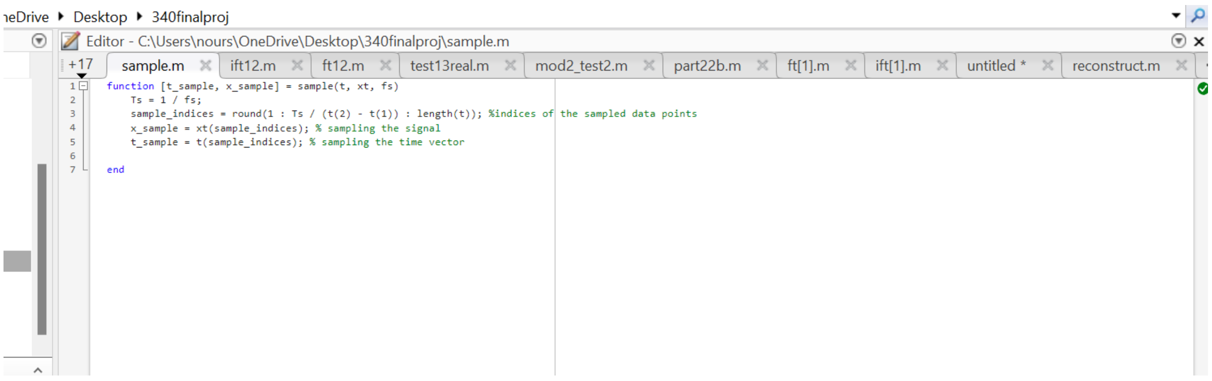
```matlab

load('Q2.mat');
fs1 = 0.5 * 2;
[t_sample_1, x_sample_1] = sample(t, xt, fs1);
fs2 = 2 * 2;
[t_sample_2, x_sample_2] = sample(t, xt, fs2);
% Plot the sampled signals using stem plots.
figure;
subplot(2, 1, 1);
stem(t_sample_1, x_sample_1, 'g-', 'LineWidth', 1);
xlabel('t sample_1 (s)');
ylabel('x sample_1');
title(['Signal Sampled at fs1(', num2str(fs1), ' Hz)']);
subplot(2, 1, 2);
stem(t_sample_2, x_sample_2, 'm-', 'LineWidth', 1);
xlabel('t sample_2 (s)');
ylabel('x sample_2');
title(['Signal Sampled at fs2 (', num2str(fs2), ' Hz)']);
```



```matlab
function recon = reconstruct(t, x_sample, fs)
    n = length(x_sample);
    Ts = 1 / fs;
    recon = zeros(size(t));
    for k = -n : n
        if k + n + 1 <= length(x_sample)
            recon =  recon + x_sample(k+n+1)*my_sinc((t-k*Ts)/Ts);
        end
    end
end

```



```matlab
function y = my_sinc(x)
    non_zeros = x ~= 0;
    y = ones(size(x));
    y(non_zeros) = sin(pi * x(non_zeros)) ./ (pi * x(non_zeros));
end
```

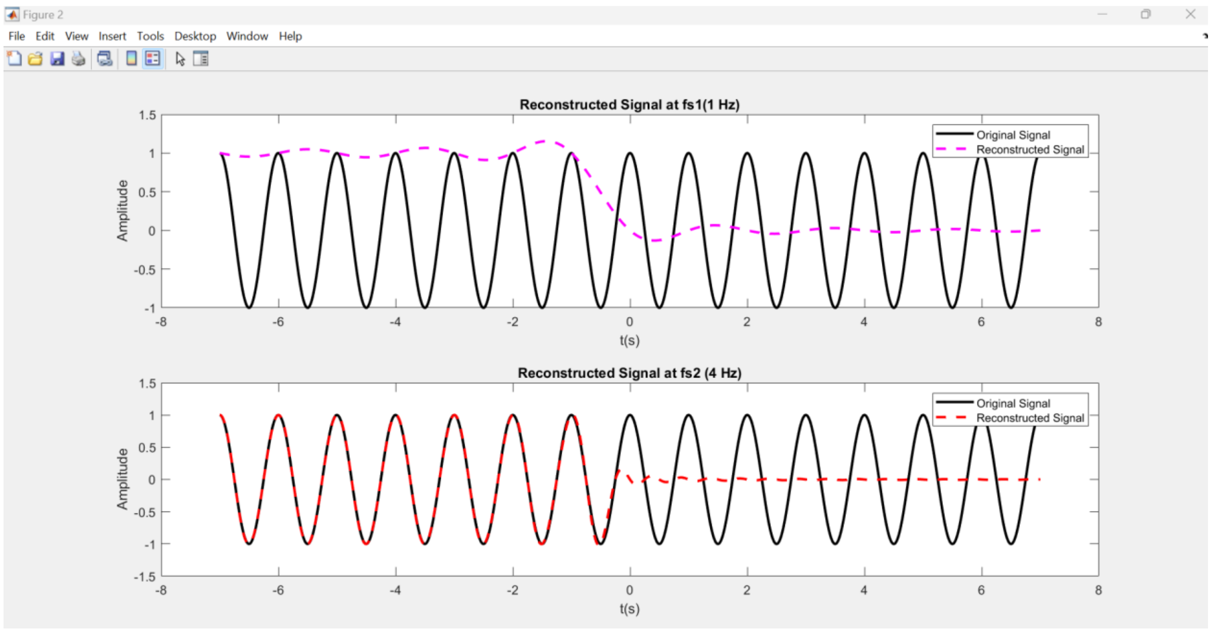


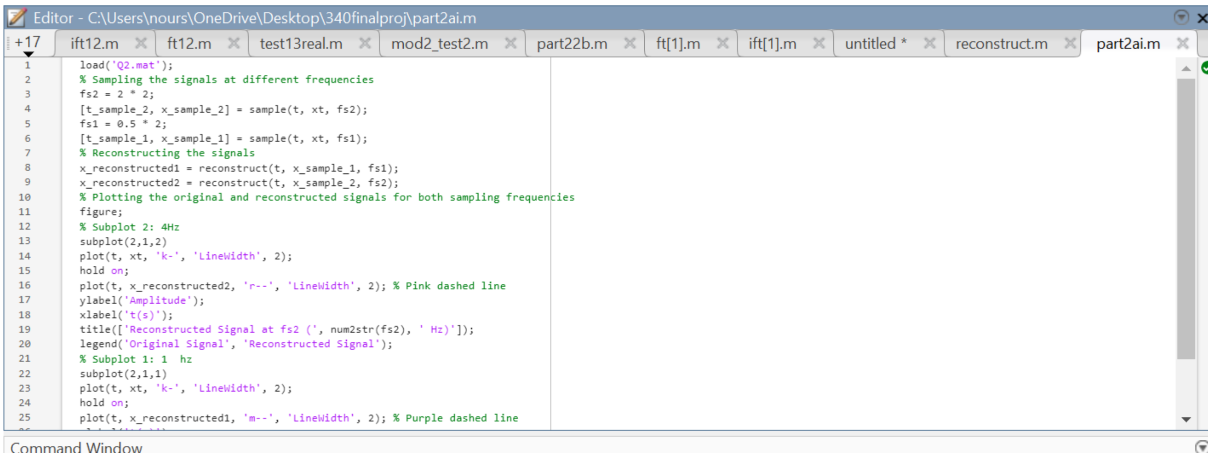
```matlab
function [t_sample, x_sample] = sample(t, xt, fs)
    Ts = 1 / fs;
    sample_indices = round(1 : Ts / (t(2) - t(1)) : length(t)); %indices of the sampled data points
    x_sample = xt(sample_indices); % sampling the signal
    t_sample = t(sample_indices); % sampling the time vector

end
```

2.2

The original signal is sampled at two different frequencies, fs2 and fs1, using the sample function.

The signals are then reconstructed using the reconstruct function, resulting in x_reconstructed1 and x_reconstructed2.
Finally, the original and reconstructed signals are plotted in two subplots, one for each sampling frequency.
Besides, the sampling frequency fs3 is calculated as 4 times the frequency f0. The original signal is sampled at fs3 using the sample function. The original signal and the sampled signal are plotted on the same figure using plot and stem functions, respectively. For more details, please check the video for further explanation.

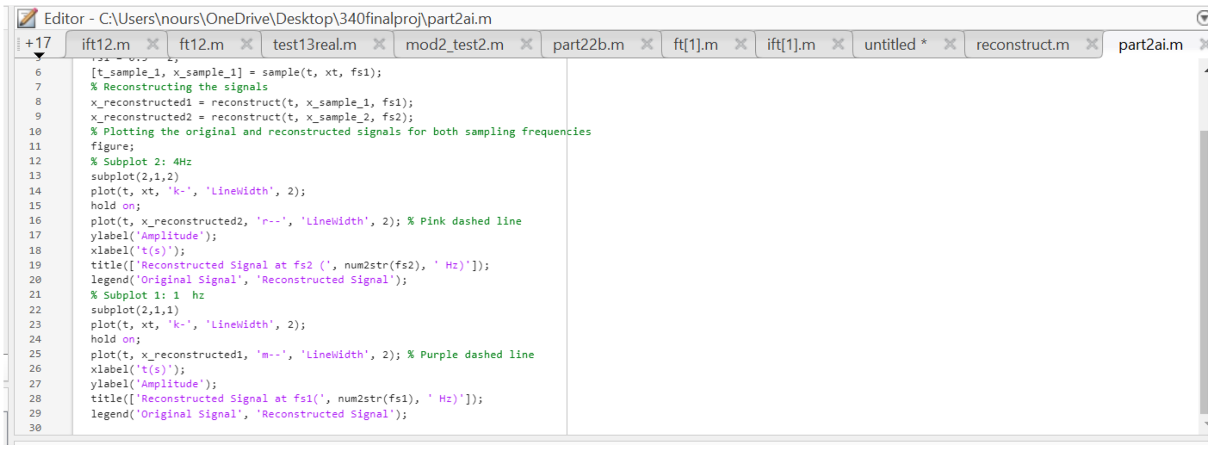



Editor - C:\Users\nours\OneDrive\Desktop\340finalproj\part2ai.m

+17 | ift12.m | ft12.m | test13real.m | mod2_test2.m | part22b.m | ft[1].m | ift[1].m | untitled * | reconstruct.m | part2ai.m

```
load('Q2.mat');
% Sampling the signals at different frequencies
fs2 = 2 * 2;
[t_sample_2, x_sample_2] = sample(t, xt, fs2);
fs1 = 0.5 * 2;
[t_sample_1, x_sample_1] = sample(t, xt, fs1);
% Reconstructing the signals
x_reconstructed1 = reconstruct(t, x_sample_1, fs1);
x_reconstructed2 = reconstruct(t, x_sample_2, fs2);
% Plotting the original and reconstructed signals for both sampling frequencies
figure;
% Subplot 2: 4Hz
subplot(2,1,2)
plot(t, xt, 'k-', 'LineWidth', 2);
hold on;
plot(t, x_reconstructed2, 'r--', 'LineWidth', 2); % Pink dashed line
ylabel('Amplitude');
xlabel('t(s)');
title(['Reconstructed Signal at fs2 (', num2str(fs2), ' Hz)']);
legend('Original Signal', 'Reconstructed Signal');
% Subplot 1: 1  hz
subplot(2,1,1)
plot(t, xt, 'k-', 'LineWidth', 2);
hold on;
plot(t, x_reconstructed1, 'm--', 'LineWidth', 2); % Purple dashed line
```

Command Window

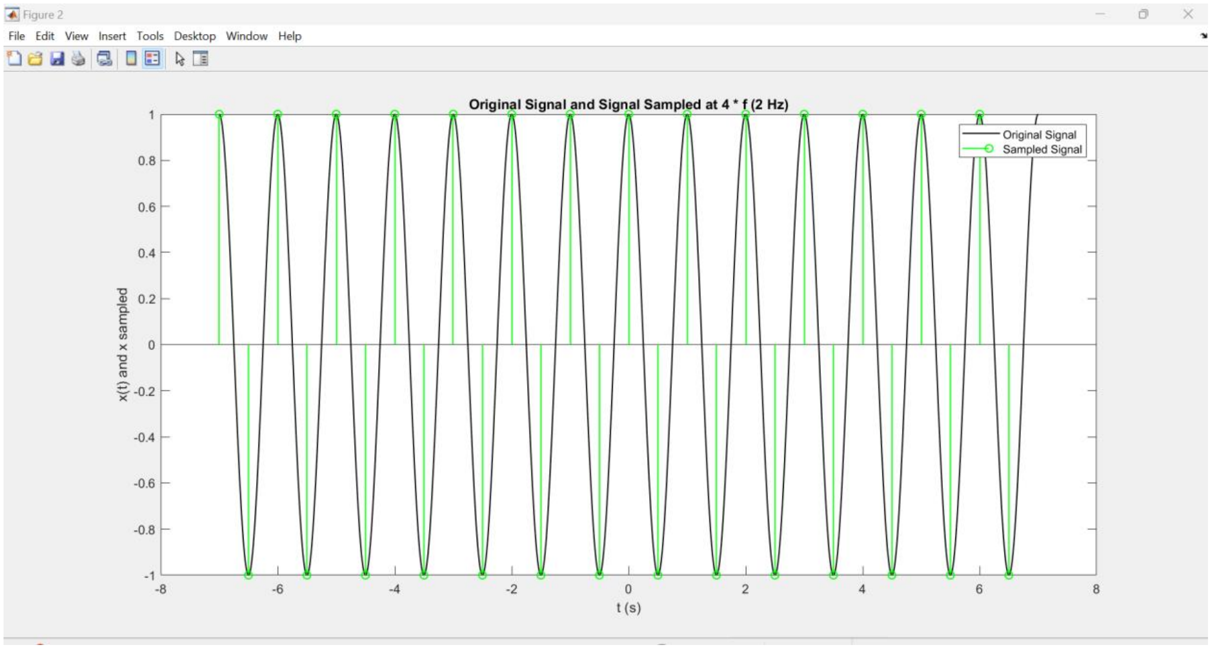


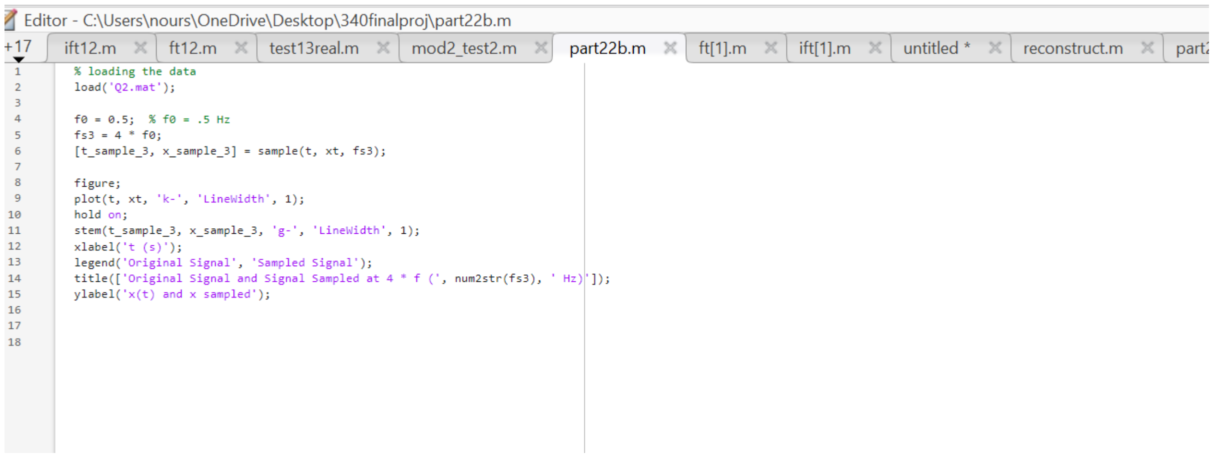
```matlab
[t_sample_1, x_sample_1] = sample(t, xt, fs1);
% Reconstructing the signals
x_reconstructed1 = reconstruct(t, x_sample_1, fs1);
x_reconstructed2 = reconstruct(t, x_sample_2, fs2);
% Plotting the original and reconstructed signals for both sampling frequencies
figure;
% Subplot 2: 4Hz
subplot(2,1,2)
plot(t, xt, 'k-', 'LineWidth', 2);
hold on;
plot(t, x_reconstructed2, 'r--', 'LineWidth', 2); % Pink dashed line
ylabel('Amplitude');
xlabel('t(s)');
title(['Reconstructed Signal at fs2 (', num2str(fs2), ' Hz)']);
legend('Original Signal', 'Reconstructed Signal');
% Subplot 1: 1  hz
subplot(2,1,1)
plot(t, xt, 'k-', 'LineWidth', 2);
hold on;
plot(t, x_reconstructed1, 'm--', 'LineWidth', 2); % Purple dashed line
xlabel('t(s)');
ylabel('Amplitude');
title(['Reconstructed Signal at fs1(', num2str(fs1), ' Hz)']);
legend('Original Signal', 'Reconstructed Signal');
```

The second reconstruction is better because it utilizes a higher sampling rate resulting in improved signal reconstruction. In MATLAB, the provided signal is time-limited, which means it has a finite duration. However, the time-bandwidth product concept (Δt * Δf ≥ 1/(2π)) states that it is not possible to have both a time-limited and a band-limited signal simultaneously.

If we attempt to limit a signal in time (by having a finite duration), its frequency content will spread out, making it impossible to achieve perfect band-limited characteristics. Therefore, it is not possible to achieve perfect reconstruction at any rate.

In this specific scenario, assuming a band-limited signal x(t) = cos(2πft) with f = 1 Hz, the signal was sampled at a rate of 4f, resulting in samples of cos(0), cos(±2π1/4), and cos(±2π1/2).





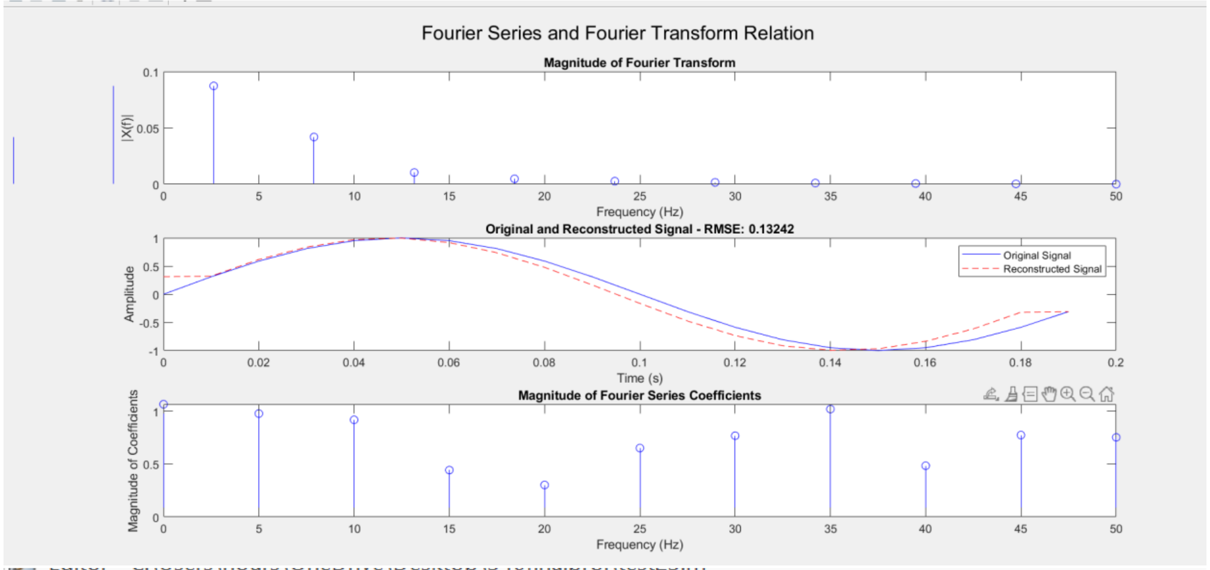
```matlab
% loading the data
load('Q2.mat');

f0 = 0.5;  % f0 = .5 Hz
fs3 = 4 * f0;
[t_sample_3, x_sample_3] = sample(t, xt, fs3);

figure;
plot(t, xt, 'k-', 'LineWidth', 1);
hold on;
stem(t_sample_3, x_sample_3, 'g-', 'LineWidth', 1);
xlabel('t (s)');
legend('Original Signal', 'Sampled Signal');
title(['Original Signal and Signal Sampled at 4 * f (', num2str(fs3), ' Hz)']);
ylabel('x(t) and x sampled');
```

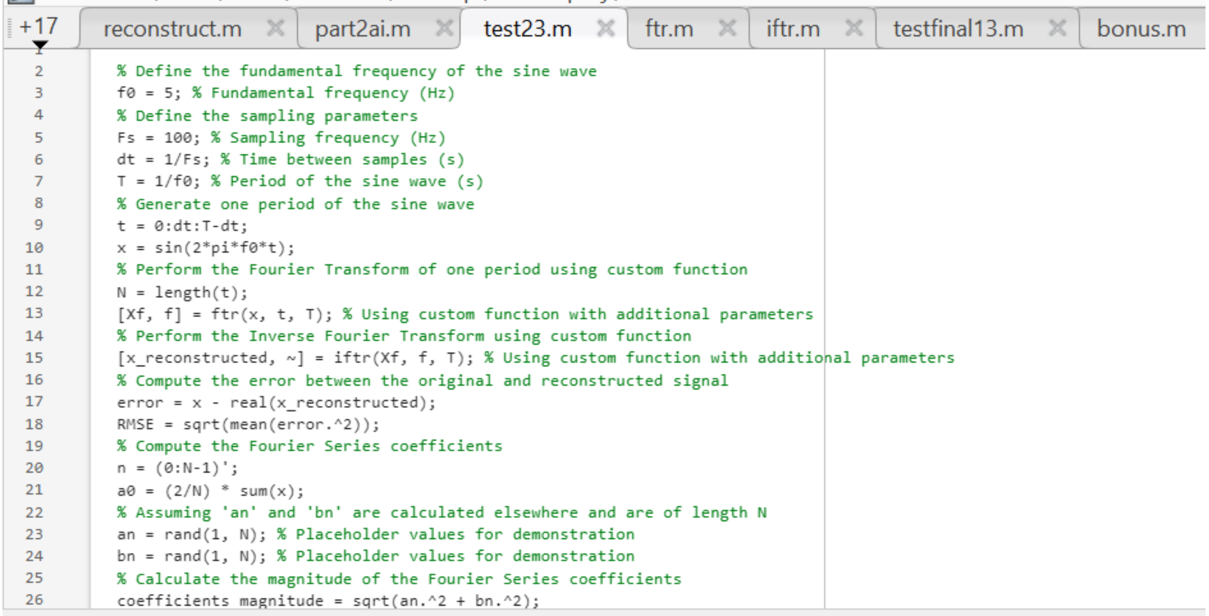
the function takes three inputs: t (time vector), x_sample (sampled signal), and fs (sampling frequency). It calculates the sampling period and initializes an array to store the reconstructed signal.

The code then iterates over a range of values and checks if the indices are within the bounds of the x_sample array. If they are, it multiplies the corresponding sample with a normalized sinc function evaluated at specific time differences. The result is added to the reconstructed signal array.

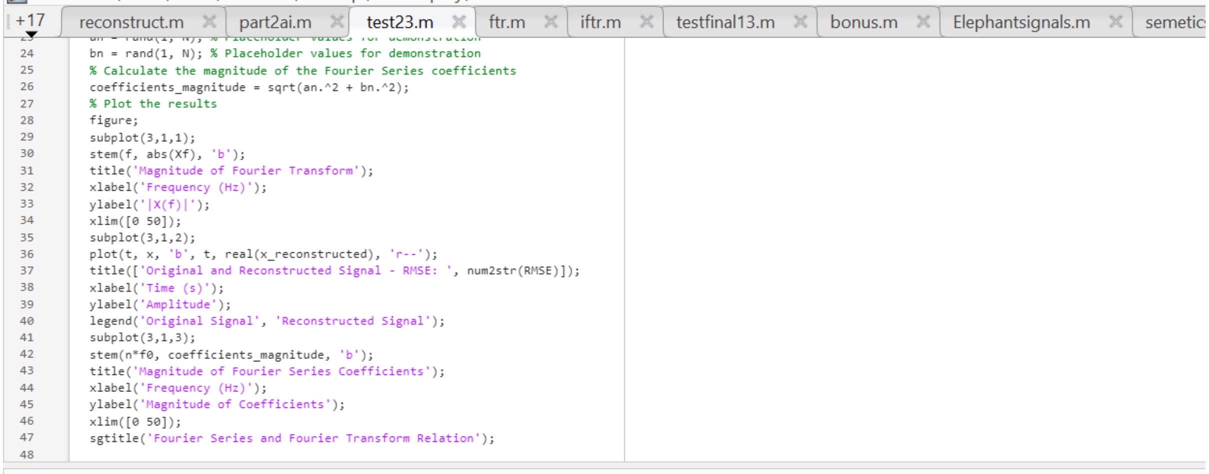
After the loop completes, the reconstructed signal is returned as the output of the function.

2.3





```
% Define the fundamental frequency of the sine wave
f0 = 5; % Fundamental frequency (Hz)
% Define the sampling parameters
Fs = 100; % Sampling frequency (Hz)
dt = 1/Fs; % Time between samples (s)
T = 1/f0; % Period of the sine wave (s)
% Generate one period of the sine wave
t = 0:dt:T-dt;
x = sin(2*pi*f0*t);
% Perform the Fourier Transform of one period using custom function
N = length(t);
[Xf, f] = ftr(x, t, T); % Using custom function with additional parameters
% Perform the Inverse Fourier Transform using custom function
[x_reconstructed, ~] = iftr(Xf, f, T); % Using custom function with additional parameters
% Compute the error between the original and reconstructed signal
error = x - real(x_reconstructed);
RMSE = sqrt(mean(error.^2));
% Compute the Fourier Series coefficients
n = (0:N-1)';
a0 = (2/N) * sum(x);
% Assuming 'an' and 'bn' are calculated elsewhere and are of length N
an = rand(1, N); % Placeholder values for demonstration
bn = rand(1, N); % Placeholder values for demonstration
% Calculate the magnitude of the Fourier Series coefficients
coefficients_magnitude = sqrt(an.^2 + bn.^2);
```

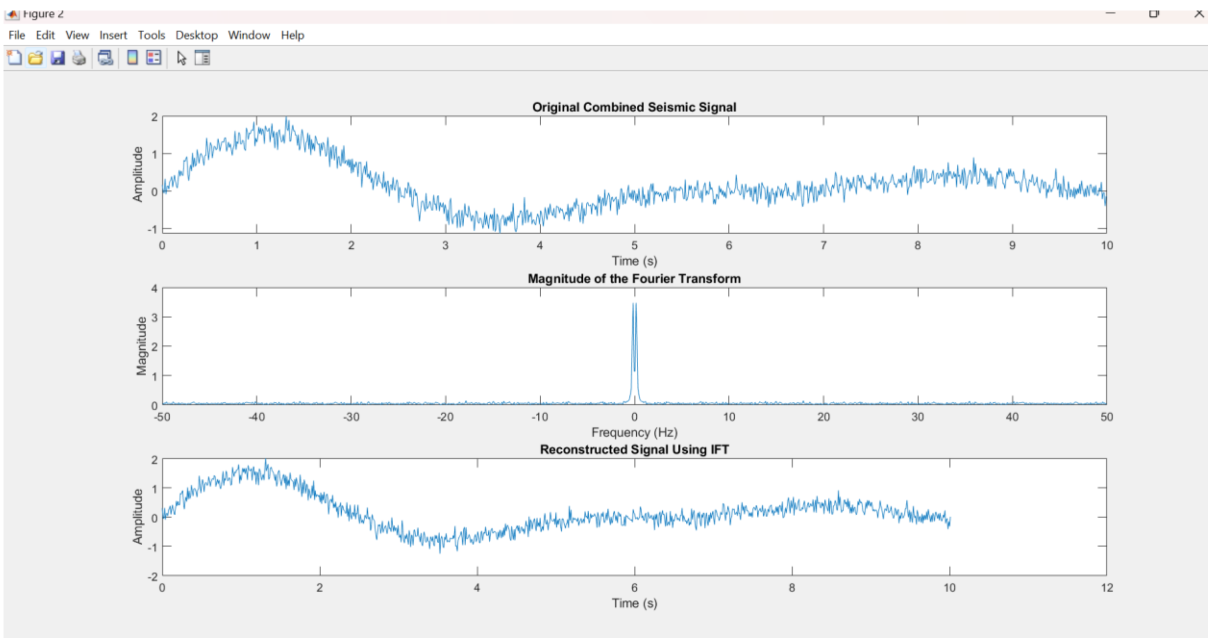


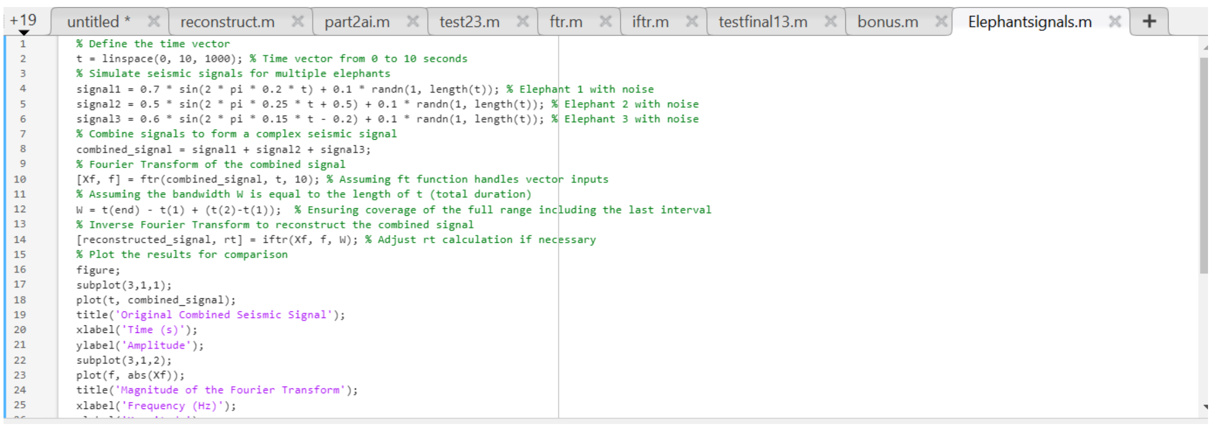
```
bn = rand(1, N); % Placeholder values for demonstration
% Calculate the magnitude of the Fourier Series coefficients
coefficients_magnitude = sqrt(an.^2 + bn.^2);
% Plot the results
figure;
subplot(3,1,1);
stem(f, abs(Xf), 'b');
title('Magnitude of Fourier Transform');
xlabel('Frequency (Hz)');
ylabel('|X(f)|');
xlim([0 50]);
subplot(3,1,2);
plot(t, x, 'b', t, real(x_reconstructed), 'r--');
title(['Original and Reconstructed Signal - RMSE: ', num2str(RMSE)]);
xlabel('Time (s)');
ylabel('Amplitude');
legend('Original Signal', 'Reconstructed Signal');
subplot(3,1,3);
stem(n*f0, coefficients_magnitude, 'b');
title('Magnitude of Fourier Series Coefficients');
xlabel('Frequency (Hz)');
ylabel('Magnitude of Coefficients');
xlim([0 50]);
sgtitle('Fourier Series and Fourier Transform Relation');

```

The code begins by setting up the parameters for a sine wave, such as its frequency and how often it's sampled. It then generates the actual waveform based on these settings.
Next, the code uses ftr to transform it. And then iftr to reconstruct it.

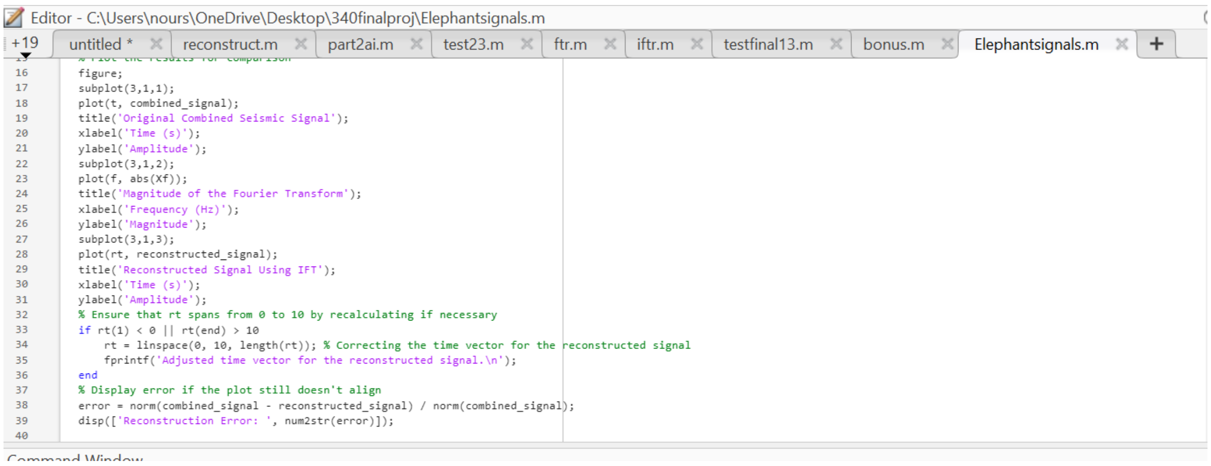
The code also measures the difference between the original and reconstructed waveforms, calculating a value called Root Mean Square Error (RMSE) to quantify the accuracy of the reconstruction.
The code then calculates the coefficients for representing the waveform using a Fourier Series. Random values are used as placeholders for these coefficients, just for demonstration purposes. The magnitudes of these coefficients are plotted in a stem plot.
we get a graph that compares the original waveform with the reconstructed waveform. The graph also includes a title indicating the calculated RMSE, which tells us how closely the reconstructed waveform matches the original.
The graph is organized into three parts: one shows the magnitudes of the frequency components, another shows the original and reconstructed waveforms, and the last one shows the magnitudes of the Fourier Series coefficients.

Bonus: Sematic activity done by 3 elephants:



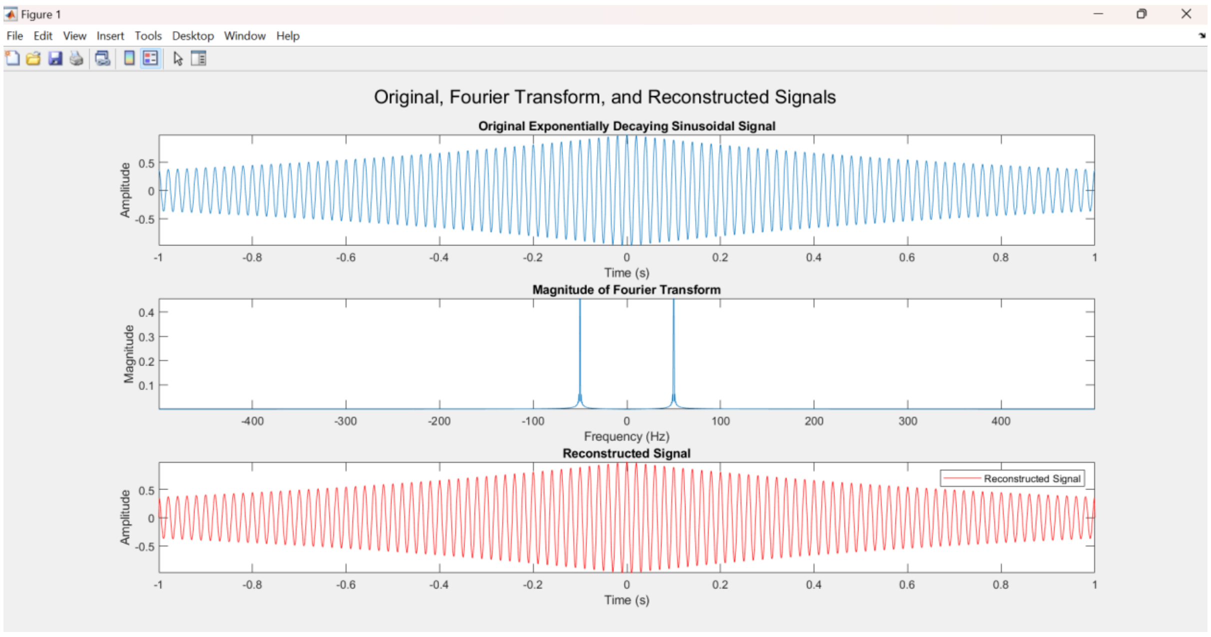


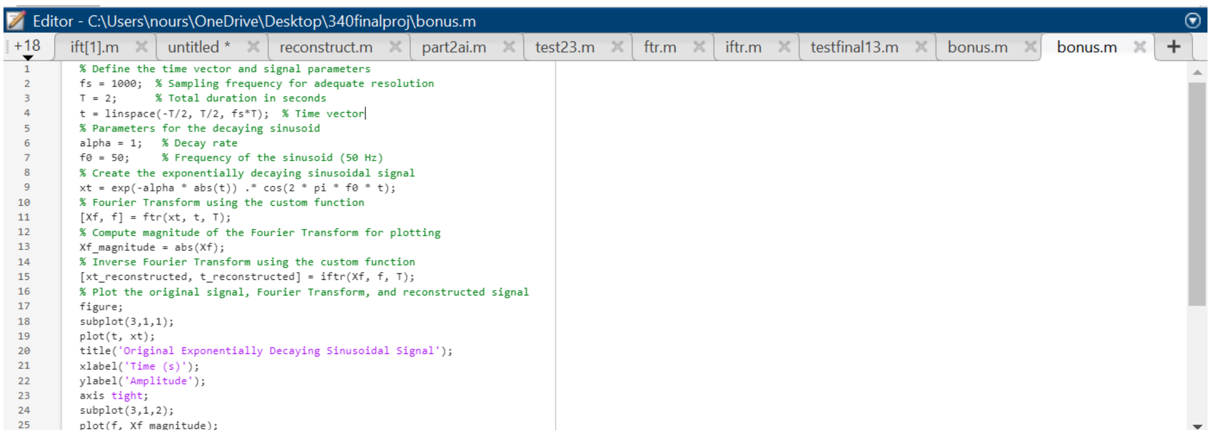
```matlab
% Define the time vector
t = linspace(0, 10, 1000); % Time vector from 0 to 10 seconds
% Simulate seismic signals for multiple elephants
signal1 = 0.7 * sin(2 * pi * 0.2 * t) + 0.1 * randn(1, length(t)); % Elephant 1 with noise
signal2 = 0.5 * sin(2 * pi * 0.25 * t + 0.5) + 0.1 * randn(1, length(t)); % Elephant 2 with noise
signal3 = 0.6 * sin(2 * pi * 0.15 * t - 0.2) + 0.1 * randn(1, length(t)); % Elephant 3 with noise
% Combine signals to form a complex seismic signal
combined_signal = signal1 + signal2 + signal3;
% Fourier Transform of the combined signal
[Xf, f] = ftr(combined_signal, t, 10); % Assuming ft function handles vector inputs
% Assuming the bandwidth W is equal to the length of t (total duration)
W = t(end) - t(1) + (t(2)-t(1));  % Ensuring coverage of the full range including the last interval
% Inverse Fourier Transform to reconstruct the combined signal
[reconstructed_signal, rt] = iftr(Xf, f, W); % Adjust rt calculation if necessary
% Plot the results for comparison
figure;
subplot(3,1,1);
plot(t, combined_signal);
title('Original Combined Seismic Signal');
xlabel('Time (s)');
ylabel('Amplitude');
subplot(3,1,2);
plot(f, abs(Xf));
title('Magnitude of the Fourier Transform');
xlabel('Frequency (Hz)');
```



```
% Plot the results for comparison
figure;
subplot(3,1,1);
plot(t, combined_signal);
title('Original Combined Seismic Signal');
xlabel('Time (s)');
ylabel('Amplitude');
subplot(3,1,2);
plot(f, abs(Xf));
title('Magnitude of the Fourier Transform');
xlabel('Frequency (Hz)');
ylabel('Magnitude');
subplot(3,1,3);
plot(rt, reconstructed_signal);
title('Reconstructed Signal Using IFT');
xlabel('Time (s)');
ylabel('Amplitude');
% Ensure that rt spans from 0 to 10 by recalculating if necessary
if rt(1) < 0 || rt(end) > 10
    rt = linspace(0, 10, length(rt)); % Correcting the time vector for the reconstructed signal
    fprintf('Adjusted time vector for the reconstructed signal.\n');
end
% Display error if the plot still doesn't align
error = norm(combined_signal - reconstructed_signal) / norm(combined_signal);
disp(['Reconstruction Error: ', num2str(error)]);
```

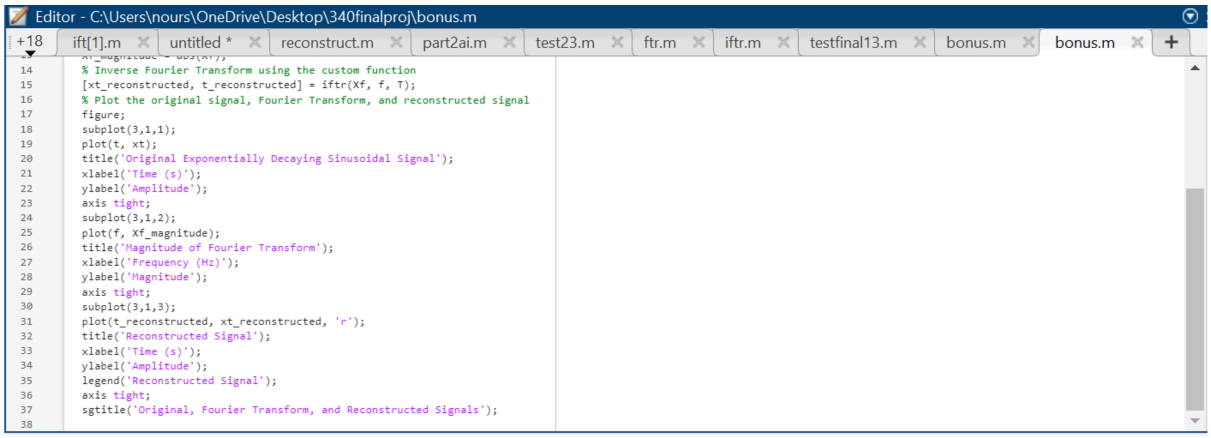
Bonus: Bong sound or sudden thud.





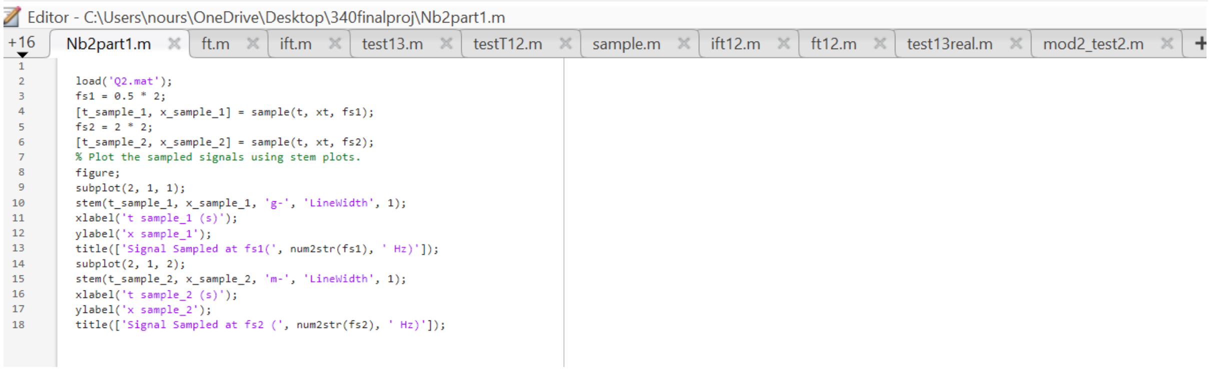
Editor - C:\Users\nours\OneDrive\Desktop\340finalproj\bonus.m

```
% Define the time vector and signal parameters
fs = 1000;  % Sampling frequency for adequate resolution
T = 2;      % Total duration in seconds
t = linspace(-T/2, T/2, fs*T);  % Time vector
% Parameters for the decaying sinusoid
alpha = 1;   % Decay rate
f0 = 50;     % Frequency of the sinusoid (50 Hz)
% Create the exponentially decaying sinusoidal signal
xt = exp(-alpha * abs(t)) .* cos(2 * pi * f0 * t);
% Fourier Transform using the custom function
[Xf, f] = ftr(xt, t, T);
% Compute magnitude of the Fourier Transform for plotting
Xf_magnitude = abs(Xf);
% Inverse Fourier Transform using the custom function
[xt_reconstructed, t_reconstructed] = iftr(Xf, f, T);
% Plot the original signal, Fourier Transform, and reconstructed signal
figure;
subplot(3,1,1);
plot(t, xt);
title('Original Exponentially Decaying Sinusoidal Signal');
xlabel('Time (s)');
ylabel('Amplitude');
axis tight;
subplot(3,1,2);
plot(f, Xf_magnitude);
```



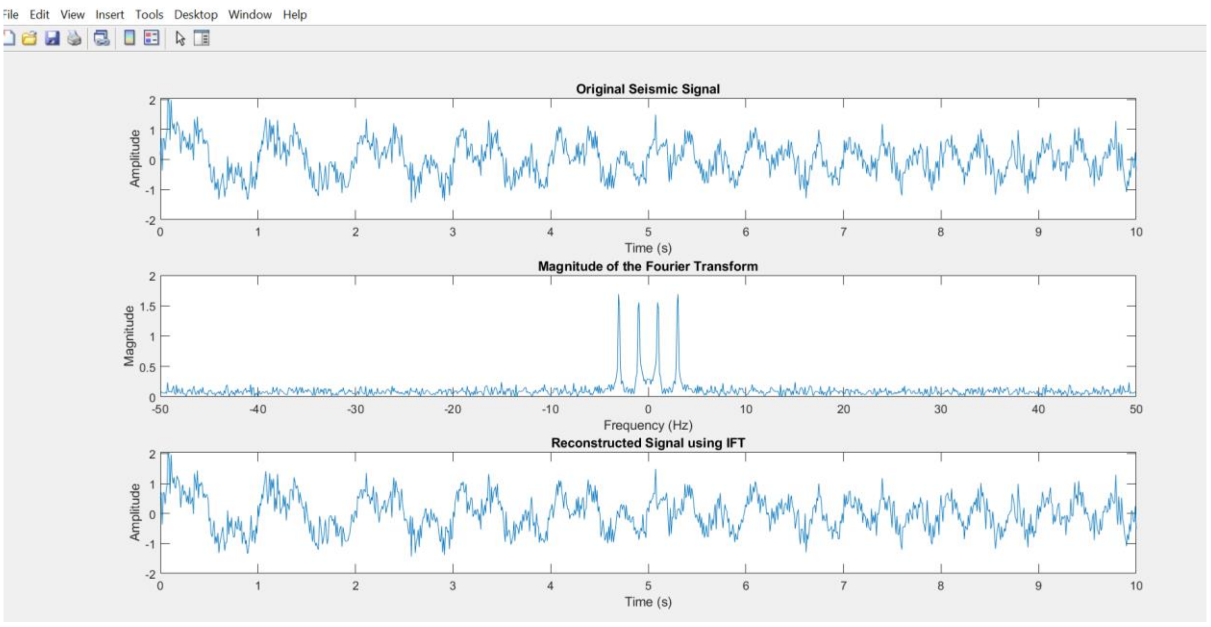
Editor - C:\Users\nours\OneDrive\Desktop\340finalproj\bonus.m

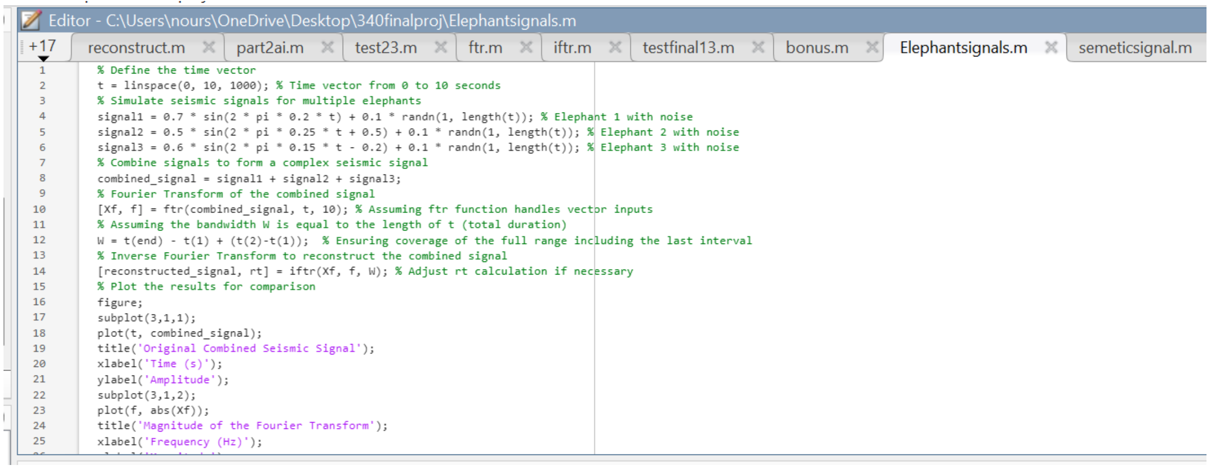
```
% Inverse Fourier Transform using the custom function
[xt_reconstructed, t_reconstructed] = iftr(Xf, f, T);
% Plot the original signal, Fourier Transform, and reconstructed signal
figure;
subplot(3,1,1);
plot(t, xt);
title('Original Exponentially Decaying Sinusoidal Signal');
xlabel('Time (s)');
ylabel('Amplitude');
axis tight;
subplot(3,1,2);
plot(f, Xf_magnitude);
title('Magnitude of Fourier Transform');
xlabel('Frequency (Hz)');
ylabel('Magnitude');
axis tight;
subplot(3,1,3);
plot(t_reconstructed, xt_reconstructed, 'r');
title('Reconstructed Signal');
xlabel('Time (s)');
ylabel('Amplitude');
legend('Reconstructed Signal');
axis tight;
sgtitle('Original, Fourier Transform, and Reconstructed Signals');
```



Editor - C:\Users\nours\OneDrive\Desktop\340finalproj\Nb2part1.m

```

load('Q2.mat');
fs1 = 0.5 * 2;
[t_sample_1, x_sample_1] = sample(t, xt, fs1);
fs2 = 2 * 2;
[t_sample_2, x_sample_2] = sample(t, xt, fs2);
% Plot the sampled signals using stem plots.
figure;
subplot(2, 1, 1);
stem(t_sample_1, x_sample_1, 'g-', 'LineWidth', 1);
xlabel('t sample_1 (s)');
ylabel('x sample_1');
title(['Signal Sampled at fs1(', num2str(fs1), ' Hz)']);
subplot(2, 1, 2);
stem(t_sample_2, x_sample_2, 'm-', 'LineWidth', 1);
xlabel('t sample_2 (s)');
ylabel('x sample_2');
title(['Signal Sampled at fs2 (', num2str(fs2), ' Hz)']);
```

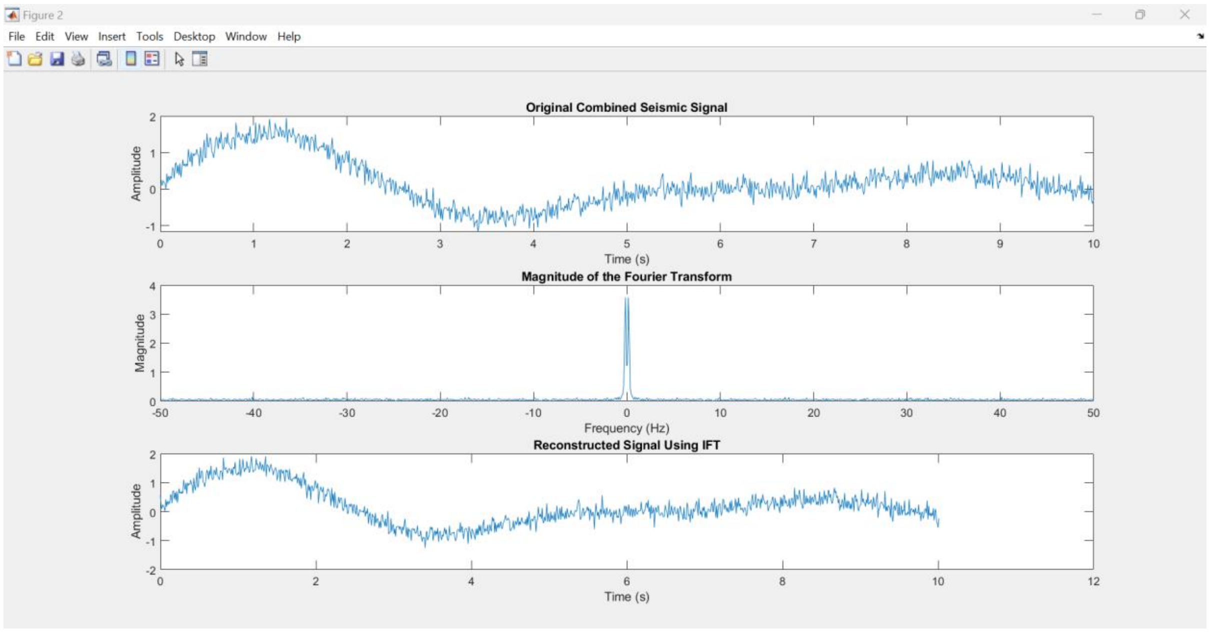


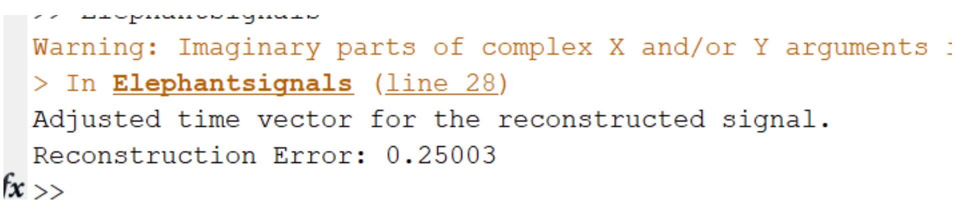


Editor - C:\Users\nours\OneDrive\Desktop\340finalproj\Elephantsignals.m

+17 | reconstruct.m | part2ai.m | test23.m | ftr.m | iftr.m | testfinal13.m | bonus.m | Elephantsignals.m | semeticsignal.m

```matlab
% Define the time vector
t = linspace(0, 10, 1000); % Time vector from 0 to 10 seconds
% Simulate seismic signals for multiple elephants
signal1 = 0.7 * sin(2 * pi * 0.2 * t) + 0.1 * randn(1, length(t)); % Elephant 1 with noise
signal2 = 0.5 * sin(2 * pi * 0.25 * t + 0.5) + 0.1 * randn(1, length(t)); % Elephant 2 with noise
signal3 = 0.6 * sin(2 * pi * 0.15 * t - 0.2) + 0.1 * randn(1, length(t)); % Elephant 3 with noise
% Combine signals to form a complex seismic signal
combined_signal = signal1 + signal2 + signal3;
% Fourier Transform of the combined signal
[Xf, f] = ftr(combined_signal, t, 10); % Assuming ftr function handles vector inputs
% Assuming the bandwidth W is equal to the length of t (total duration)
W = t(end) - t(1) + (t(2)-t(1));  % Ensuring coverage of the full range including the last interval
% Inverse Fourier Transform to reconstruct the combined signal
[reconstructed_signal, rt] = iftr(Xf, f, W); % Adjust rt calculation if necessary
% Plot the results for comparison
figure;
subplot(3,1,1);
plot(t, combined_signal);
title('Original Combined Seismic Signal');
xlabel('Time (s)');
ylabel('Amplitude');
subplot(3,1,2);
plot(f, abs(Xf));
title('Magnitude of the Fourier Transform');
xlabel('Frequency (Hz)');
```





```
Warning: Imaginary parts of complex X and/or Y arguments :
> In Elephantsignals (line 28)
Adjusted time vector for the reconstructed signal.
Reconstruction Error: 0.25003
fx >>
```

As for the real signals: "Real-signal.txt" and "Elephant-signals.txt".

Real Signal Processing: A complex seismic signal is simulated over 10 seconds at 1000 points: seismic_signal = exp(-0.2*t) .* sin(2 * pi * 1 * t) + 0.5 * sin(2 * pi * 3 * t) + 0.3 * randn(size(t));

The Fourier Transform is applied using ft.m, and then the signal is reconstructed using ift.m.

Visualization:

The script plots three graphs:

Original Signal:

Displays the original time-domain seismic signal. Magnitude of the Fourier Transform: Shows the magnitude of the transformed signal in the frequency domain.

Reconstructed Signal using IFT: Plots the signal reconstructed from the Fourier Transform to validate the inverse operation.

Analyzing Seismic Signals of Multiple Elephants This MATLAB script is designed to simulate seismic signals from multiple elephants, combine them into a complex signal, and then analyze this complex signal using Fourier Transform (FT) and Inverse Fourier Transform (IFT). The purpose is to demonstrate the handling of composite signals and validate the reconstruction capabilities of the transforms.

Simulated Seismic Signals:

Each elephant's seismic activity is represented by a sinusoidal function modified by random noise to simulate real-world data inconsistencies: signal1 = 0.7 * sin(2 * pi * 0.2 * t) + 0.1 * randn(1, length(t)); % Elephant 1 signal2 = 0.5 * sin(2 * pi * 0.25 * t + 0.5) + 0.1 * randn(1, length(t)); % Elephant 2 signal3 = 0.6 * sin(2 * pi * 0.15 * t - 0.2) + 0.1 * randn(1, length(t)); % Elephant 3

These equations create three distinct seismic signals by varying amplitude, frequency, phase, and noise level.

Combined Signal: combined_signal = signal1 + signal2 + signal3; The individual signals are summed to form a complex seismic signal, which is then used for further analysis.

Visualization:

The script plots three figures:

Original Combined Seismic Signal:

Displays the time-domain representation of the combined seismic signal.

Magnitude of the Fourier Transform: Shows the magnitude spectrum of the Fourier Transform, highlighting dominant frequencies.

Reconstructed Signal Using IFT: Plots the reconstructed time-domain signal for visual comparison with the original.

Reconstruction Error Calculation: error = norm(combined_signal - reconstructed_signal) / norm(combined_signal); disp(['Reconstruction Error: ', num2str(error)]); This computes the norm of the error between the original and reconstructed signals relative to the original signal's norm, providing a quantitative measure of reconstruction accuracy.