Question 1. Comparing OLS/LMS and LTS estimators on MSE/RB/MAD metrics

```
% Parameters
p = 2; % Number of predictors
N_values = [20, 100]; % Number of observations
R = 200; % Number of realizations

% Generate synthetic data
data = struct(); % Store synthetic data
for i = 1:length(N_values)
    N = N_values(i);
    X = randn(N, p, R); % Predictor variables for R realizations
    beta_true = [3; 5]; % True coefficients
    epsilon = randn(N, R); % Residuals for R realizations
    Y = zeros(N, R); % Response variable
    for r = 1:R
        Y(:, r) = X(:,1,r) * beta_true(1) + X(:,2,r) * beta_true(2) + epsilon(:, r); % Response variable for each realization
    end
    data(i).X = X;
    data(i).Y = Y;
end

% Perform parameter estimation and compute metrics
results = struct(); % Store results

metrics_OLS = zeros(length(N_values),2,3);
metrics_LMS = zeros(length(N_values),2,3);
metrics_LTS = zeros(length(N_values),2,3);

for i = 1:length(N_values)
    N = N_values(i);
    X_all = data(i).X;
    Y_all = data(i).Y;
    beta_estimates_all = zeros(p, R, 3); % Store beta estimates for each realization and each estimator

    for r = 1:R
        X = X_all(:,:,r);
        Y = Y_all(:,r);

        % OLS estimation
```

```
        beta_ols = OLS_GD(X, Y, 0.001, 800, 1e-4);

        % LMS estimation
        beta_lms = LMS_GD(X, Y, 0.001, 800, 1e-4);

        % LTS estimation
        beta_lts = LTS_GD(X, Y, 0.001, 800, 1e-4);

        % Store beta estimates
        beta_estimates_all(:, r, 1) = beta_ols; % Store OLS estimates
        beta_estimates_all(:, r, 2) = beta_lms; % Store LMS estimates
        beta_estimates_all(:, r, 3) = beta_lts; % Store LTS estimates

    end

    % Compute metrics

    beta_ols_mean = mean(beta_estimates_all(:, :, 1), 2);
    beta_lms_mean = mean(beta_estimates_all(:, :, 2), 2);
    beta_lts_mean = mean(beta_estimates_all(:, :, 3), 2);


    MSE_OLS = (beta_ols_mean - beta_true).^2;
    RB_OLS = median(beta_estimates_all(:, :, 1),2)-beta_true;
    MAD_OLS = median(abs(beta_estimates_all(:, :, 1) - beta_true),2);


    MSE_LMS = (beta_lms_mean - beta_true).^2;
    RB_LMS= median(beta_estimates_all(:, :, 2),2)-beta_true;
    MAD_LMS = median(abs(beta_estimates_all(:, :, 2) - beta_true),2);


    MSE_LTS = (beta_lts_mean - beta_true).^2;
    RB_LTS= median(beta_estimates_all(:, :, 3),2)-beta_true;
    MAD_LTS = median(abs(beta_estimates_all(:, :, 3) - beta_true),2);

%     Store the Metrics in an Array

    metrics_OLS(i, :, :) = [MSE_OLS, RB_OLS, MAD_OLS];
    metrics_LMS(i, :,:) = [MSE_LMS, RB_LMS, MAD_LMS];
    metrics_LTS(i, :,:) = [MSE_LTS, RB_LTS, MAD_LTS];
```

```
end
```

Displaying the Metrics

```
% Display metrics for all estimators
disp('Metrics for OLS, LMS, LTS:');
```

```
Metrics for OLS, LMS, LTS:
```

```
disp('-----------------------------------------------');
```

```
-----------------------------------------------
```

```
disp('N       |   MSE                  |   RB                   |   MAD');
```

```
N       |   MSE                  |   RB                   |   MAD
```

```
disp('-----------------------------------------------');
```

```
-----------------------------------------------
```

```
for i = 1:length(N_values)
    disp(['N = ', num2str(N_values(i))]);
    disp('OLS:');
    disp(squeeze(metrics_OLS(i, :, :)));
    disp('LMS:');
    disp(squeeze(metrics_LMS(i, :, :)));
    disp('LTS:');
    disp(squeeze(metrics_LTS(i, :, :)));
    disp('-----------------------------------------------');
    disp('N       |   MSE                  |   RB                   |   MAD');
    disp('-----------------------------------------------');
end
```

```
N = 20
OLS:
    0.0011   -0.0491    0.1634
    0.0001   -0.0032    0.1646
LMS:
```

```
    0.0049   -0.0408    0.1972
    0.0059   -0.0773    0.1896
LTS:
    0.2913   -0.4695    0.5058
    0.9188   -0.8057    0.8057
-----------------------------------------------
N       |   MSE                    |   RB                     |   MAD
-----------------------------------------------
N = 100
OLS:
    0.0001   -0.0160    0.0647
    0.0000   -0.0008    0.0690
LMS:
    0.0003   -0.0245    0.0870
    0.0000   -0.0016    0.0865
LTS:
    0.0327   -0.1599    0.2045
    0.0844   -0.2508    0.2508
-----------------------------------------------
N       |   MSE                    |   RB                     |   MAD
-----------------------------------------------
```

```
disp('-----------------------------------------------');
```

```
-----------------------------------------------
```

Gradient Descent Functions

```
function beta_hat = OLS_GD(X, Y, alpha, max_iter, tol)

    % alpha: Learning rate
    % max_iter: Maximum number of iterations
    % tol: Tolerance for convergence

    % Initialize beta with zeros
    beta_hat = zeros(size(X, 2), 1);

    % Gradient descent loop
    for iter = 1:max_iter
        % Compute gradient of the loss function
        grad = 2 * X' * (X * beta_hat - Y);
```

```
        % Update beta using gradient descent
        beta_hat = beta_hat - alpha * grad;

        % Check convergence
        if norm(grad) < tol
%             fprintf('Converged at iteration %d.\n', iter);
            break;
        end
    end

    if iter == max_iter
%         fprintf('Maximum number of iterations reached.\n');
    end
end

function beta_hat = LMS_GD(X, Y, alpha, max_iter, tol)
    % alpha: Learning rate
    % max_iter: Maximum number of iterations
    % tol: Tolerance for convergence
    % Initialize beta with zeros
    beta_hat = zeros(size(X, 2), 1);

    % Gradient descent loop
    for iter = 1:max_iter

        % Compute Gradient
        grad = X'* sign(X * beta_hat - Y);

        % Update beta using gradient descent
        beta_hat = beta_hat - alpha * grad;

        % Check convergence
        if norm(grad) < tol
%             fprintf('Converged at iteration %d.\n', iter);
            break;
        end
    end

    if iter == max_iter
%         fprintf('Maximum number of iterations reached.\n');
    end
end
```

```
function beta_hat = LTS_GD(X, Y, alpha, max_iter, tol)
    % X: Design matrix (n x p)
    % Y: Response vector (n x 1)
    % alpha: Learning rate
    % max_iter: Maximum number of iterations
    % tol: Tolerance for convergence

    n = size(X, 1); % Number of samples
    q = floor(n/2) + 1; % Number of samples in the subset

    % Initialize beta with zeros
    beta_hat = zeros(size(X, 2), 1);

    % Gradient descent loop
    for iter = 1:max_iter
        % Compute squared residuals
        residuals = (X * beta_hat - Y).^2;

        % Order squared residuals
        ordered_residuals = sort(residuals);

        % Select a subset of squared residuals
        subset_residuals = ordered_residuals(1:q);

        % Find indices of selected squared residuals
        [~, idx] = ismember(subset_residuals, residuals);

        % Extract corresponding rows from X and Y
        X_subset = X(idx, :);
        Y_subset = Y(idx);

        % Compute gradient of the OLS cost function
        grad = 2 * X_subset' * (X_subset * beta_hat - Y_subset);

        % Update beta using gradient descent
        beta_hat = beta_hat - alpha * grad;

        % Check convergence
        if norm(grad) < tol
%             fprintf('Converged at iteration %d.\n', iter);
```

```
            break;
        end
    end

    if iter == max_iter
%         fprintf('Maximum number of iterations reached.\n');
    end
end
```

1. **Ordinary Least Squares (OLS) Estimator:**

- MSE: OLS minimizes the sum of squared errors, assuming the errors are Gaussian white noise. Since the errors are sampled from a Gaussian white noise, OLS is expected to perform well and provide the least MSE which is verifed experimentally.
- RB and MAD: OLS also tends to have low relative bias and MAD when the underlying assumptions of the linear model are met. In the case of Gaussian white noise, OLS is efficient and unbiased, leading to low RB and MAD.

2. Least Mean Squares (LMS) Estimator:

- MSE: LMS is an adaptive estimator that updates the estimate iteratively based on the observed data. While it may converge to the true parameter value over time, it might not reach the theoretical minimum MSE achieved by OLS.
- RB and MAD: LMS can be biased, especially during the transient phase of adaptation, where it may take time to converge to the true parameter value. This transient bias can result in higher RB and MAD compared to OLS.

3. Least Trimmed Squares (LTS) Estimator:

- MSE: LTS is robust to outliers as it minimizes the sum of squared errors after discarding a certain percentage of the most extreme residuals. However, in this scenario where the errors are Gaussian white noise without outliers, LTS might discard useful information, leading to a higher MSE.
- RB and MAD: LTS might introduce a larger bias and MAD compared to OLS and LMS, especially if the proportion of outliers removed is not appropriately chosen. Without outliers in the data, LTS might discard valid data points, resulting in a biased estimate and higher MAD.

Question 1- Part iii) Real Dataset

```
clear
data = readtable('medical_insurance.csv');

% Convert categorical variables to dummy variables (one-hot encoding)
[sex_categories, ~, sex_dummy] = unique(data.sex);
sex_dummy;
[region_categories, ~, region_dummy] = unique(data.region);
region_dummy;
[smoker_categories, ~, smoker_dummy] = unique(data.smoker);
smoker_dummy;


% Extract numerical features
numerical_features = data{:, {'age', 'bmi', 'children'}};

% Concatenate numerical features with dummy variables
X = [numerical_features, sex_dummy, smoker_dummy, region_dummy];
X=normalize(X);

% Extract target variable (Y)
Y = data.charges;
Y=normalize(Y);
```

```
% Split the data into training and test sets (test data = 20%)
rng(42); % Set random seed for reproducibility
test_ratio = 0.2;
N = size(X, 1);
test_size = round(test_ratio * N);
idx = randperm(N);
X_train = X(idx(1:end-test_size), :);
Y_train = Y(idx(1:end-test_size));
X_test = X(idx(end-test_size+1:end), :);
Y_test = Y(idx(end-test_size+1:end));
```

```
beta_OLS = OLS_GD(X_train, Y_train, 0.0001, 2000, 1e-6);
```

```
% LMS estimation
beta_LMS = LMS_GD(X_train, Y_train, 0.0001, 2000, 1e-6);

% LTS estimation
beta_LTS = LTS_GD(X_train, Y_train, 0.0001, 2000, 1e-6);
```

Reporting the metrics on training data

```
Y_pred_OLS_train=X_train*beta_OLS;
MSE_OLS_train=1/N*sum((Y_train-Y_pred_OLS_train).^2)
```

```
MSE_OLS_train = 0.1957
```

```
Y_pred_LMS_train=X_train*beta_LMS;
MSE_LMS_train=1/N*sum((Y_train-Y_pred_LMS_train).^2)
```

```
MSE_LMS_train = 0.2490
```

```
Y_pred_LTS_train=X_train*beta_LTS;
MSE_LTS_train=1/N*sum((Y_train-Y_pred_LTS_train).^2)
```

```
MSE_LTS_train = 0.3060
```

Reporting the metrics on test data

```
Y_pred_OLS=X_test*beta_OLS;
MSE_OLS=1/N*sum((Y_test-Y_pred_OLS).^2)
```

```
MSE_OLS = 0.0536
```

```
Y_pred_LMS=X_test*beta_LMS;
MSE_LMS=1/N*sum((Y_test-Y_pred_LMS).^2)
```

```
MSE_LMS = 0.0616
```

```
Y_pred_LTS=X_test*beta_LTS;
MSE_LTS=1/N*sum((Y_test-Y_pred_LTS).^2)
```

```
MSE_LTS = 0.0728
```

Gradient Descent Function

```
function beta_hat = OLS_GD(X, Y, alpha, max_iter, tol)
```

```
    % alpha: Learning rate
    % max_iter: Maximum number of iterations
    % tol: Tolerance for convergence

    % Initialize beta with zeros
    beta_hat = zeros(size(X, 2), 1);

    % Gradient descent loop
    for iter = 1:max_iter
        % Compute gradient of the loss function
        grad = 2 * X' * (X * beta_hat - Y);

        % Update beta using gradient descent
        beta_hat = beta_hat - alpha * grad;

        % Check convergence
        if norm(grad) < tol
%             fprintf('Converged at iteration %d.\n', iter);
            break;
        end
    end

    if iter == max_iter
%         fprintf('Maximum number of iterations reached.\n');
    end
end


function beta_hat = LMS_GD(X, Y, alpha, max_iter, tol)
    % alpha: Learning rate
    % max_iter: Maximum number of iterations
    % tol: Tolerance for convergence
    % Initialize beta with zeros
    beta_hat = zeros(size(X, 2), 1);

    % Gradient descent loop
    for iter = 1:max_iter

        % Compute Gradient
        grad = X'* sign(X * beta_hat - Y);
```

```
        % Update beta using gradient descent
        beta_hat = beta_hat - alpha * grad;

        % Check convergence
        if norm(grad) < tol
%             fprintf('Converged at iteration %d.\n', iter);
            break;
        end
    end

    if iter == max_iter
%         fprintf('Maximum number of iterations reached.\n');
    end
end



function beta_hat = LTS_GD(X, Y, alpha, max_iter, tol)
    % X: Design matrix (n x p)
    % Y: Response vector (n x 1)
    % alpha: Learning rate
    % max_iter: Maximum number of iterations
    % tol: Tolerance for convergence

    n = size(X, 1); % Number of samples
    q = floor(n/2) + 1; % Number of samples in the subset

    % Initialize beta with zeros
    beta_hat = zeros(size(X, 2), 1);

    % Gradient descent loop
    for iter = 1:max_iter
        % Compute squared residuals
        residuals = (X * beta_hat - Y).^2;

        % Order squared residuals
        ordered_residuals = sort(residuals);

        % Select a subset of squared residuals
        subset_residuals = ordered_residuals(1:q);
```

```
        % Find indices of selected squared residuals
        [~, idx] = ismember(subset_residuals, residuals);

        % Extract corresponding rows from X and Y
        X_subset = X(idx, :);
        Y_subset = Y(idx);

        % Compute gradient of the OLS cost function
        grad = 2 * X_subset' * (X_subset * beta_hat - Y_subset);

        % Update beta using gradient descent
        beta_hat = beta_hat - alpha * grad;

        % Check convergence
        if norm(grad) < tol
%             fprintf('Converged at iteration %d.\n', iter);
            break;
        end
    end

    if iter == max_iter
%         fprintf('Maximum number of iterations reached.\n');
    end
end
```

It can be seen that OLS estimator gives the least MSE on both the training and test data. Hence I would choose OLS estimator.

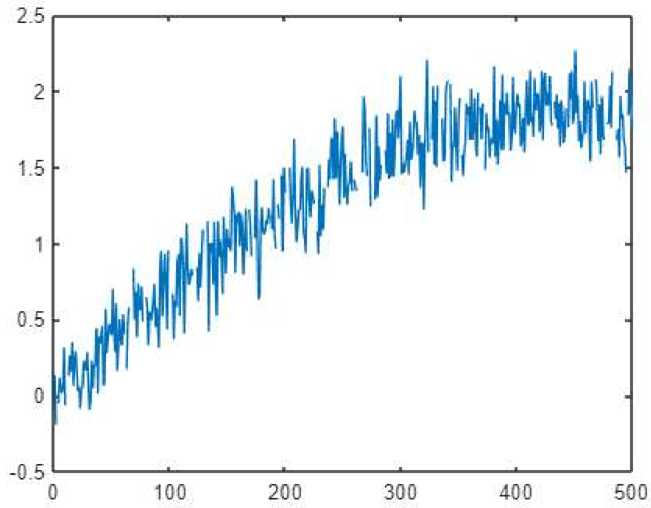
Q2 A)

```
N = 500;
Fs = 50;
t = linspace(0, 1, N);

frequencies = [10, 17];


M=20;
indices = 1:Fs* M/2;
frequencies_modelling= indices * (1/M);
```

We have chosen the upper bound of the frequency as the Nyquist frequency which is 1/2* (Sampling Frequency)

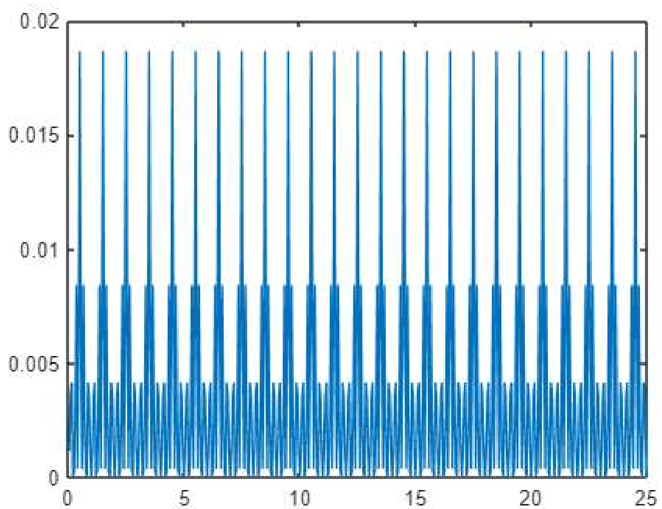
Constructing and Plotting the signal according to the specifications

```
SNR=10;

signal_woNoise = sin(2*pi*frequencies(1) * t/50) + sin(2*pi*frequencies(2) * t/50);
missing_percentage = 0.1;
missing_indices = randperm(N, floor(N * missing_percentage));

% Remove the indices corresponding to missing data
signal_woNoise(missing_indices) = NaN;
signal_woNoise;

signal_power = mean(signal_woNoise.^2);
% sigma_e = sqrt(signal_power/SNR);

sigma_e= nanvar(signal_woNoise) / (10^(SNR/10));
% Add noise to the signal
e_k = sqrt(sigma_e) * randn(size(signal_woNoise));

N_new=length(signal_woNoise);
% ek = normrnd(0,sigma_e,N_new,1)';

% Plotting the signal
signal= signal_woNoise + e_k;
plot(signal)
```



Training

```
max_iterations = 10000;
learning_rate = 0.000001;
tolerance=0.01;
[a_estimated, b_estimated,y_predicted,NMSE,MAPE,periodogram] = lsp_gradient_descent(signal, frequencies_modelling, max_iterations, learning_rate,tolerance);

plot(frequencies_modelling,periodogram)
```




```
% MAPE= mean(abs((y_predicted-signal)./(signal)))*100
```

Displaying the Metrics

```
disp('Estimated parameters (a_i):');
```

```
Estimated parameters (a_i):
```

```
disp(a_estimated);
```

```
    0.0159
    0.0197
   -0.0371
    0.0035
   -0.0164
   -0.0230
    0.0301
    0.0034
    0.0776
    0.0000
   -0.0360
    0.0174
    0.0746
    0.0507
    0.0013
```

```
disp('Estimated parameters (b_i):');
```

```
Estimated parameters (b_i):
```

```
disp(b_estimated);
```

```
   -0.0301
   -0.0564
   -0.0527
   -0.0067
    0.0035
   -0.0461
   -0.0868
   -0.0191
    0.0145
    0.1367
   -0.0703
   -0.0088
   -0.0536
    0.0089
    0.0167
```

```
% disp('The predicted signal is:')
% disp(y_predicted)
disp('NMSE')
```

```
NMSE
```

```
disp(NMSE)
```

```
    0.4551
```

```
disp('MAPE')
```

```
MAPE
```

```
disp(MAPE)
```

```
   101.0862
```

Function definition

```
function [a_estimated, b_estimated,y_predicted, error_final,MAPE,periodogram] = lsp_gradient_descent(signal, frequencies, max_iterations, learning_rate, tol)

     nan_indices = isnan(signal);
     signal(nan_indices) = 0;


    % Initialize parameters (a_i and b_i)
    num_frequencies = length(frequencies);
    a = zeros(num_frequencies, 1);
    b = zeros(num_frequencies, 1);

    % Mean-center the signal
    signal_mean = mean(signal);
    centered_signal = signal - signal_mean;
    N = length(signal);

    % Gradient descent
    for iter = 1:max_iterations

        gradient_a = zeros(num_frequencies, 1);
        gradient_b = zeros(num_frequencies, 1);
        cos_term_f=zeros(num_frequencies,N, 1);
        sin_term_f=zeros(num_frequencies,N, 1);

        t_k = 1:N; % Time index

        for i = 1:num_frequencies
            cos_term = cos(2 * pi * frequencies(i) * t_k);
            sin_term = sin(2 * pi * frequencies(i) * t_k);
            residual = centered_signal - a(i) * cos_term - b(i) * sin_term;
            gradient_a(i) = 2 * sum(residual .* sin_term);
            gradient_b(i) = -2 * sum(residual .* cos_term);
            cos_term_f(i,:)=cos_term;
            sin_term_f(i,:)=sin_term;


        end

        % Update parameters
        a = a - learning_rate * gradient_a;
        b = b - learning_rate * gradient_b;
```

```
        periodogram = (a.^2+ b.^2)';
        % Check termination condition
        if norm(gradient_a) + norm(gradient_b) < tol
            fprintf('Termination Reached\n');
            break;
        end
    end


    a_estimated=a;
    b_estimated=b;

    y_predicted=  a .* cos_term_f + b .* sin_term_f;
    y_predicted= sum(y_predicted,1)/num_frequencies;

    y_error= (centered_signal-y_predicted).^2;
    error_final=mean(y_error);

    %To avoid 0/0 form

    non_zero_indices = signal ~= 0;
    signal_non_zero = signal(non_zero_indices);
    y_predicted_non_zero = y_predicted(non_zero_indices);

    % Calculate MAPE
    MAPE = mean(abs((y_predicted_non_zero - signal_non_zero) ./ signal_non_zero)) * 100;


end
```

The plots for the synthetic signal and the periodogram have been plotted above. The values of NMSE and MAPE have been reported for the lomb scale periodogram estimator. Signal has been reconstructed by taking the sum of the signals obatained for each (a,b) pair for the chosen modelling frequencies.

Q2 B)

Loading the data

```
% Step 1: Load the data
data = readtable('Tesla Stock Price.csv');
dates = datenum(data.Date, 'dd-mm-yyyy'); % Convert dates to numeric values

% Explicitly set the first date
first_date = dates(1);

% Step 2: Split the data into training and testing sets (e.g., 80% training, 20% testing)
train_percentage = 0.8;
train_idx = 1:round(train_percentage * length(dates));
test_idx = (length(train_idx)+1):length(dates);

train_dates = dates(train_idx);
train_data = normalize(data.Close(train_idx));
test_dates = dates(test_idx);
test_data = normalize(data.Close(test_idx));

% Convert dates to units on the time scale starting from the first date
train_time_scale = train_dates - first_date;
test_time_scale = test_dates - first_date;


% Convert the time series data into equally spaced time intervals
t_train = train_time_scale;
t_test = test_time_scale;

% k = 1:N; % Time index
% t_k = t_train(k);
```

Setting Modelling Frequencies

The sample frequency chosen is 50 Hz and the frequency range is 1- Fs/2.

```
N = length(t_train);
Fs = 50;
t = linspace(0, 1, N);

M=10
```

```
M = 10
```

```
indices = 1:Fs* M/2;
frequencies_modelling= indices * (1/M);
```

Training

```
max_iterations = 10000; % Set the maximum number of iterations
learning_rate = 0.000001; % Set the learning rate
tol = 0.001;% Set the tolerance

% train_data'
[a_estimated, b_estimated,y_predicted,error] = lsp_gradient_descent(train_data',t_train', frequencies_modelling, max_iterations, learning_rate,tol);

y_predicted;
% NMSE_train=error/var(train_data)

[NMSE_train,MAPE_train]=findMetrics(train_data', t_train', frequencies_modelling,a_estimated,b_estimated)
```

```
NMSE_train = 1.0002
MAPE_train = 100.2039
```

```
[NMSE_test,MAPE_test]=findMetrics(test_data', t_test', frequencies_modelling,a_estimated,b_estimated)
```

```
NMSE_test = 1.0010
MAPE_test = 110.1481
```

Displaying the metrics

```
disp(['Normalized Mean Square Error (NMSE) for training data: ', num2str(NMSE_train)]);
```

```
Normalized Mean Square Error (NMSE) for training data: 1.0002
```

```
disp(['Mean Absolute Percentage Error (MAPE) for training data: ', num2str(MAPE_train), '%']);
```

```
Mean Absolute Percentage Error (MAPE) for training data: 100.2039%
```

```
disp(['Normalized Mean Square Error (NMSE) for test data: ', num2str(NMSE_test)]);
```

```
Normalized Mean Square Error (NMSE) for test data: 1.001
```

```
disp(['Mean Absolute Percentage Error (MAPE) for test data: ', num2str(MAPE_test), '%']);
```

```
Mean Absolute Percentage Error (MAPE) for test data: 110.1481%
```

ARIMA modelling

```
% Step 4: Fit the ARIMA model to training data
model = estimate(arima(2,1,2), train_data);
```

```
    ARIMA(2,1,2) Model (Gaussian Distribution):

                   Value      StandardError    TStatistic      PValue
```

Constant	0.0034991	0.0025664	1.3634	0.17275
AR{1}	-0.66121	0.12096	-5.4666	4.5875e-08
AR{2}	-0.78504	0.11267	-6.9677	3.2212e-12
MA{1}	0.69327	0.11405	6.0788	1.2105e-09
MA{2}	0.81203	0.10598	7.662	1.8299e-14
Variance	0.0019667	3.2496e-05	60.519	0

```
% Step 5: Forecast on training data
train_forecast = forecast(model, length(train_data));

% Step 6: Evaluate on training data

train_data = data.Close(train_idx);
train_actual_values = train_data;
train_forecast_values = train_forecast;

% Calculate NMSE on training data
train_NMSE = mean((train_actual_values - train_forecast_values).^2) / var(train_actual_values.^2);

% Calculate MAPE on training data
train_MAPE = mean(abs((train_actual_values - train_forecast_values) ./ train_actual_values)) * 100;

disp(['Training Normalized Mean Square Error (NMSE) using ARIMA: ' num2str(train_NMSE)]);
```

```
Training Normalized Mean Square Error (NMSE) using ARMIA: 2.6651e-05
```

```
disp(['Training Mean Absolute Percentage Error (MAPE) using ARIMA: ' num2str(train_MAPE)]);
```

```
Training Mean Absolute Percentage Error (MAPE) using ARMIA: 98.8582
```

```
% Step 7: Forecast on testing data
test_forecast = forecast(model, length(test_data));

% Step 8: Evaluate on testing data

test_data = data.Close(test_idx);
test_actual_values = test_data;
test_forecast_values = test_forecast;

% Calculate NMSE on testing data
test_NMSE = mean((test_actual_values - test_forecast_values).^2) / var(test_actual_values.^2);

% Calculate MAPE on testing data
test_MAPE = mean(abs((test_actual_values - test_forecast_values) ./ test_actual_values)) * 100;

disp(['Testing Normalized Mean Square Error (NMSE) using ARMIA: ' num2str(test_NMSE)]);
```

```
Testing Normalized Mean Square Error (NMSE) using ARMIA: 3.1458e-05
```

```
disp(['Testing Mean Absolute Percentage Error (MAPE) using ARMIA: ' num2str(test_MAPE)]);
```

```
Testing Mean Absolute Percentage Error (MAPE) using ARMIA: 99.8787
```

The performance of the ARIMA model largely depends on whether the dataset is normalised or not.It performs better when the data is unnormalised unlike the Lomb Scale Periodogram.When normalised it performs worse, giving high values of NMSE.

Gradient Descent Function

```
function [a_estimated, b_estimated,y_predicted, error_final] = lsp_gradient_descent(signal,times, frequencies, max_iterations, learning_rate, tol)
    % Initialize parameters (a_i and b_i)
    num_frequencies = length(frequencies);
    a = zeros(num_frequencies, 1);
    b = zeros(num_frequencies, 1);


    % Mean-center the signal
    signal_mean = mean(signal);
    centered_signal = signal - signal_mean;
    N = length(signal);

    % Gradient descent
    for iter = 1:max_iterations
        gradient_a = zeros(num_frequencies, 1);
        gradient_b = zeros(num_frequencies, 1);
        cos_term_f=zeros(num_frequencies,N, 1);
        sin_term_f=zeros(num_frequencies,N, 1);

        k = 1:N; % Time index
        for i = 1:num_frequencies

            cos_term = cos(2 * pi * frequencies(i) * times(k));
            sin_term = sin(2 * pi * frequencies(i) * times(k));
            residual = centered_signal - a(i) * cos_term - b(i) * sin_term;
            gradient_a(i) = 2 * sum(residual .* sin_term);
            gradient_b(i) = -2 * sum(residual .* cos_term);
            cos_term_f(i,:)=cos_term;
            sin_term_f(i,:)=sin_term;
        end

        % Update parameters
        a = a - learning_rate * gradient_a;
        b = b - learning_rate * gradient_b;

        % Check termination condition
```

```
        if norm(gradient_a) + norm(gradient_b) < tol
            fprintf('Termination Reached\n');
            break;
        end
    end

    a_estimated=a;
    b_estimated=b;

    y_predicted=  a .* cos_term_f + b .* sin_term_f;
    y_predicted= sum(y_predicted,1);

    y_error= (centered_signal-y_predicted).^2;
    error_final=mean(y_error);

end



function [NMSE,MAPE] = findMetrics(signal, times, frequencies,a_estimated,b_estimated)
    % Initialize parameters (a_i and b_i)

    num_frequencies = length(frequencies);

    % Mean-center the signal
    signal_mean = mean(signal);
    centered_signal = signal - signal_mean;
    N = length(signal);

    % Generating the signal

        cos_term_f=zeros(num_frequencies,N, 1);
        sin_term_f=zeros(num_frequencies,N, 1);

        k = 1:N; % Time index

        for i = 1:num_frequencies
            cos_term = cos(2 * pi * frequencies(i) * times(k));
            sin_term = sin(2 * pi * frequencies(i) * times(k));
            cos_term_f(i,:)=cos_term;
            sin_term_f(i,:)=sin_term;
        end


    y_predicted=  a_estimated .* cos_term_f + b_estimated .* sin_term_f;
    y_predicted= sum(y_predicted,1);

    y_error= (centered_signal-y_predicted).^2;
    NMSE=mean(y_error)/var(centered_signal);
```

```
    MAPE= mean(abs((centered_signal-y_predicted)./(centered_signal)))*100;
end
```

The NMSE values obtained using lomb scale periodogram are higher than ARIMA. It is typically due to the fact that we are considering all the frequencies in the range to construct the signal. Using only the dominant frequencies that give low values of residues would make the Lomb Scale fit better. Also identifying the suitable frequency range plays a very important role.