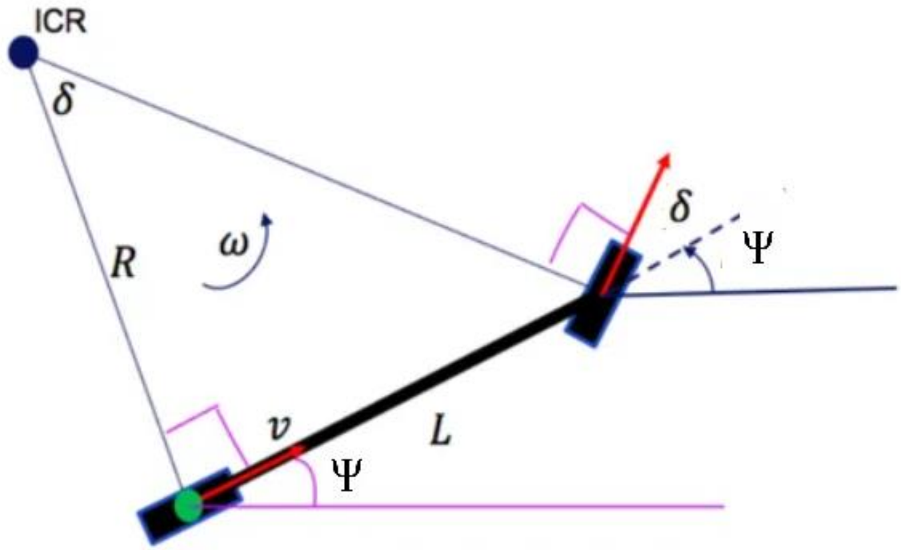
Q1.1:

Ans:



Using the above diagram [1] adopted from the Rajamani's figure provided in the writeup (and in the classroom). The assumptions considered here are that:

- The vehicle has no rear wheel steering, δ_r=0
- Vehicle slip angle "β" is ignored or zero
- Position of vehicle is measured at the rear wheel of the vehicle, i.e. point B

Using vector decomposition,

$$\boxed{X_{dot} = V\cos\Psi}$$
$$\boxed{Y_{dot} = V\sin\Psi}$$

Additionally we know,

$$\omega = V/R$$

Since, referencing the diagram above:

$$\tan\delta_f = \frac{l_f + l_r}{R}$$
$$\frac{1}{R} = \frac{\tan\delta_f}{l_f + l_r}$$

So,

$$\boxed{\Psi_{dot} = \omega = V\,\frac{\tan\delta_f}{l_f + l_r}}$$

Q1.2:

Ans: Referencing Rajamani's equations for Kong's simplified model. The assumption over here are:

- The vehicle has no rear wheel steering, δ_r=0
- Position of vehicle is measured at the center of gravity of the vehicle, i.e. point C

$$\boxed{X_{dot} = V\cos(\Psi + \beta)}$$
$$\boxed{Y_{dot} = V\sin(\Psi + \beta)}$$

$$\Psi_{dot} = \frac{V\cos\beta}{l_f + l_r}\left(\tan\delta_f - \tan\delta_r\right)$$

Putting δ_r=0

$$\Psi_{dot} = \frac{V\cos\beta}{l_f + l_r}\left(\tan\delta_f\right)$$

Assuming, δ_f and β to be negligible, and $l_f + l_r \sim l_r$

$$\Psi_{dot} = \frac{V\cos\beta}{l_r}\left(\frac{\sin\delta_f}{\cos\delta_f}\right)$$

For small enough angles, $\cos\beta = \cos\delta_f = 1$, and $\sin\beta = \sin\delta_f$:

$$\boxed{\Psi_{dot} = \frac{V\sin\beta}{l_r}}$$

Q 1.3:

(a):

Ans: Vehicle slip angle "β" is defined as the angle subtended by the velocity vector and the center of gravity along the longitudinal axis of the vehicle. In the Pepy's model it is indeed assumed to be zero, but in reality it does not mean that vehicle slip angle β is zero, since this is a convenient oversimplification of the generalized Kinematic Bicycle Model. Mathematically, slip angle can be determined using the following formula:

$$\beta = tan^{-1}\left(\frac{Y_{dot}}{X_{dot}}\right)$$

$$\beta = tan^{-1}\left(\frac{V\sin\Psi}{V\cos\Psi}\right)$$

(b):

Ans: "Vehicle Slip" is defined as the measure of deviation of the vehicle's actual path from its heading (i.e., where it is pointed), and is determined by calculating the angle between velocity vector and center of gravity through the longitudinal axis of the vehicle. While "tire slip" is the slip occurring during breaking/acceleration or corner banking experienced by the tire interfaces with the surface.

The relationship between tire slip and vehicle slip is codependent but distinct and does not necessarily share a causal relationship. It is possible to have tire slip without vehicle slip when the vehicle is following a straight line, where the vehicle's actual path and orientation are aligned. Consequently, it is possible to have a vehicle slip without tire slip when the coefficient of friction between the contact of the wheel and the surface is reduced which causes skid.

1.4:

(a):

Ans: For Pepy's simplified Kinematic Bicycle Model, the following relationship holds true:

$$\boxed{tan\delta_f = \frac{L}{R}}$$

Where, δ_f is the steering angle of the front wheel, L= $l_f + l_r$ and R is the turning radius or path radius from the ICR to the rear wheel.

Rearranging the above equation we get:

$$R = \frac{L}{tan\delta_f}$$

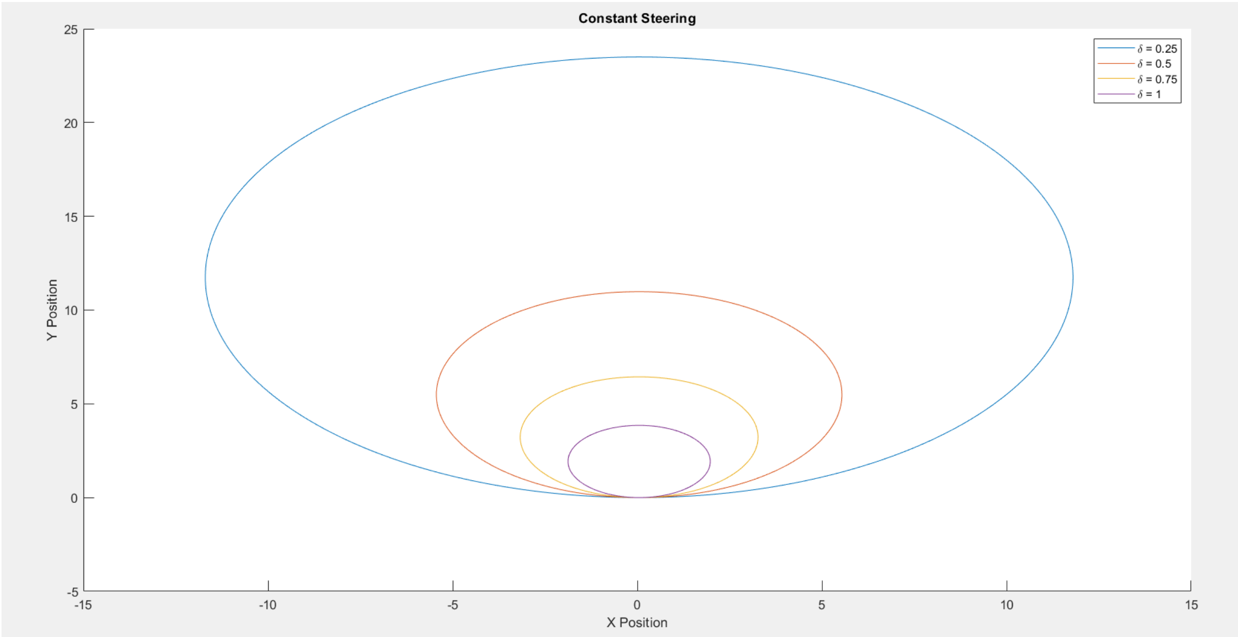
Since, curvature "K" has an inverse relationship with path radius "R", then:

$$K = \frac{1}{R}$$

i.e.

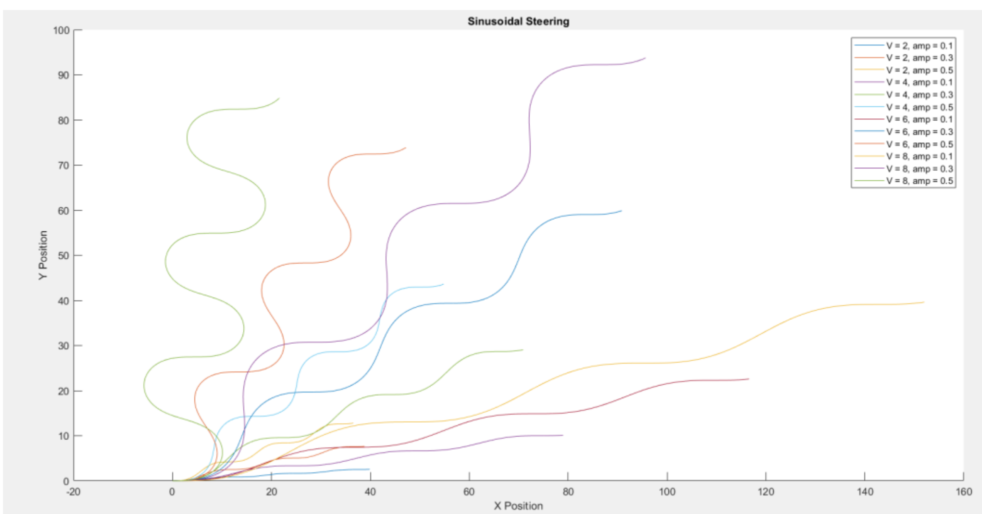
$$\boxed{K = \frac{tan\delta_f}{L}}$$

Which is the relationship between the path curvature "K" and steering angle δ_f



(b):

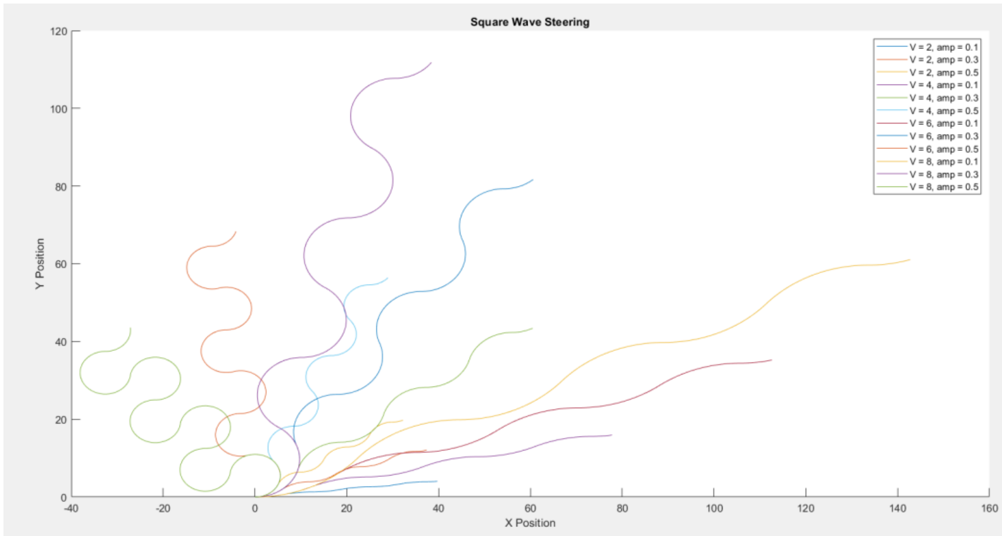
Ans: For this case, steering angle $\delta = A\sin(\omega t)$. Intuitively, I expected the trajectory to look like sinusoid itself, oscillating left and right along the forward heading, varying the amplitude for the input steering signal would make the vehicle oscillate even move, and varying the velocity would have a direct relationship with the peak-to-peak distance of the trough and crest of the sinusoidal trajectory of the vehicle. This is exactly what I observed when I simulated the trajectory with varying amplitudes of 0.1, 0.3 and, 0.5, and velocities of 2, 4, 6, and 8 as shown below:



(c):

Ans: For this case, steering angle δ is considered to be a square wave. Physically this is unrealistic since it would mean instantaneous changes in steering angle of the vehicle, then staying constant for a period, and then instantly changing direction again. There are physical limitations on the car like inertia of the mechanical steering mechanism, propensity of the vehicle to skid and loose control, actuation transition, and wear & tear of mechanical parts within the vehicle making this sort of a steering signal implausible.

Such kind of a maneuver could be handled by using a smoothing lag between such rash transitions, which would introduce continuous relaxation (something similar to a sigmoid waveform) instead of rigid discrete extreme transitions (like a square wave).



2.1:

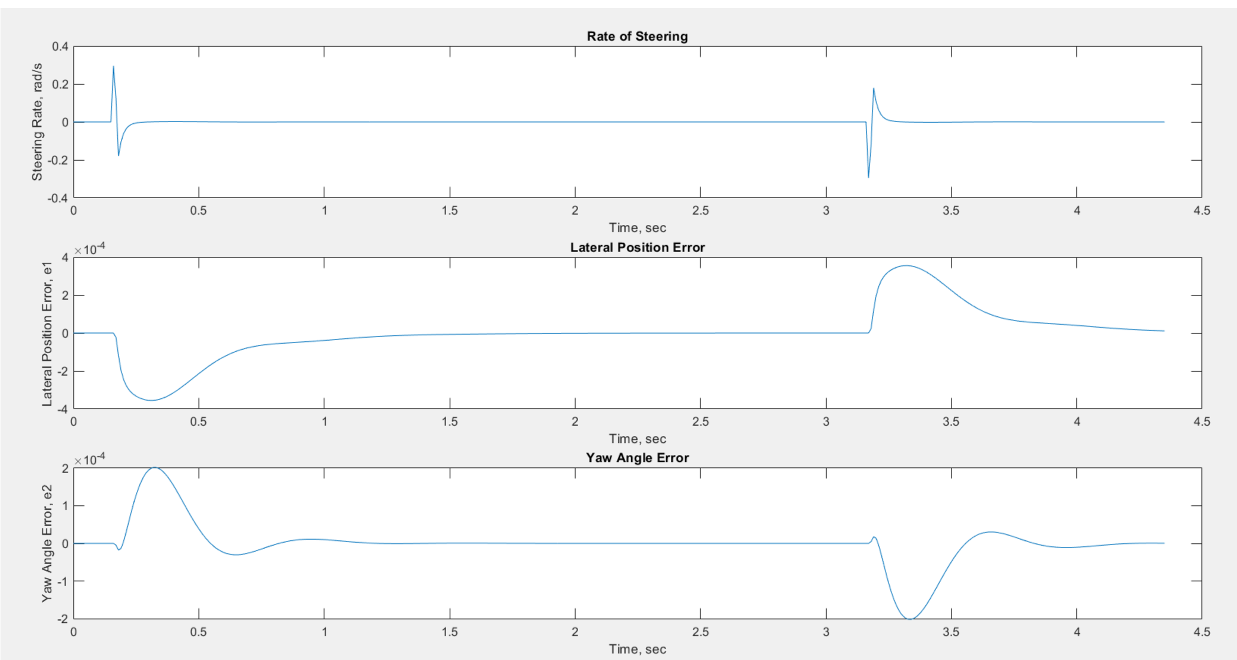
```
>> init_dynamic_model
Eigenvalues of A:
   0.0000 + 0.0000i
  -6.8308 + 5.0278i
  -6.8308 - 5.0278i
  -0.0000 + 0.0000i
```

2.2:

Ans: The controller design method I used for following the desired path (as described in the assignment), is LQR rather than pole placement just because of ease of use in MATLAB. MATLAB has a convenient function '*lqr'* with tuning parameters of Q and R as opposed to the iterative and non-intuitive design of the pole placement method. The poles I got are:

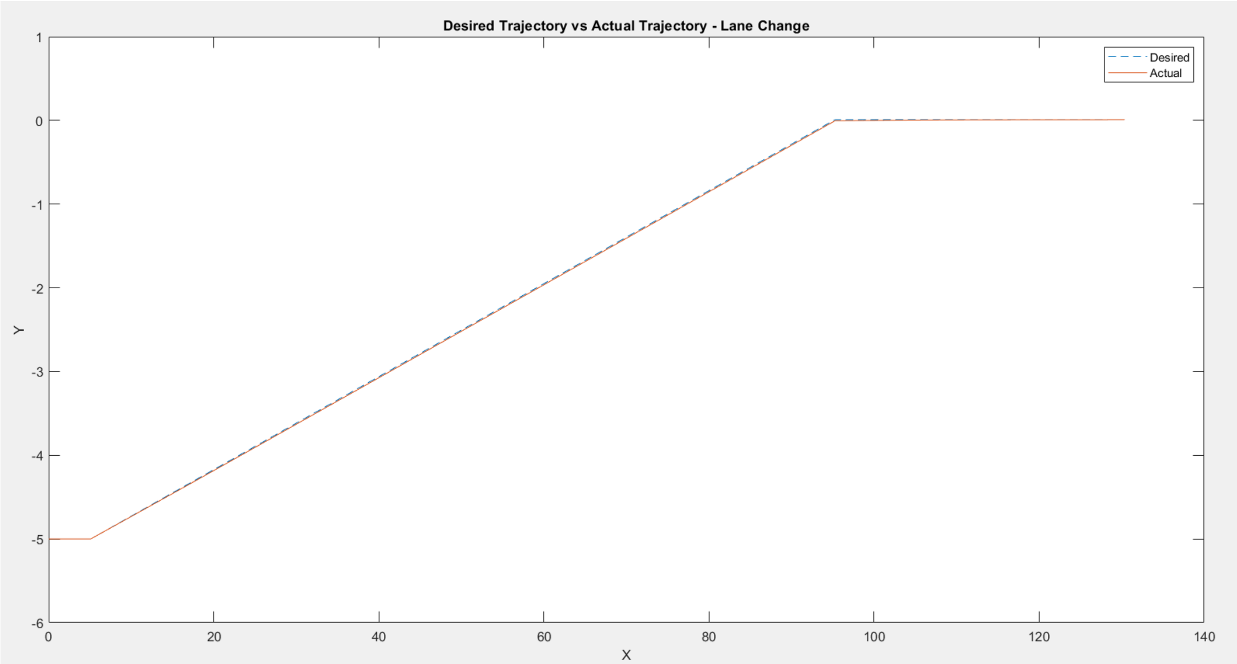
```
>> lane_change_task(vx,A, B1, B2, C)
Closed-loop poles are:
 -54.8026 + 0.0000i
  -3.1706 + 0.0000i
  -4.9069 + 9.9741i
  -4.9069 - 9.9741i
```

As observed in figure below, the steering rate, lateral position error (e1), and yaw angle error (e2) are all within the bounds as described in the writeup.

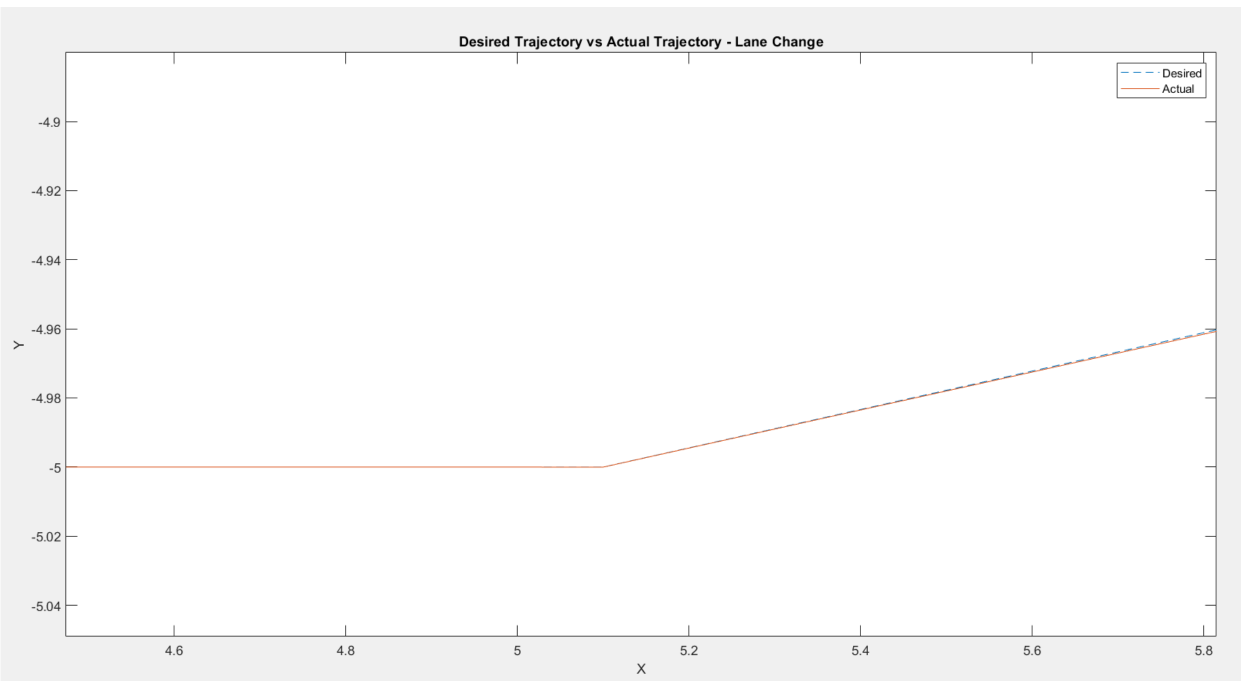


2.3

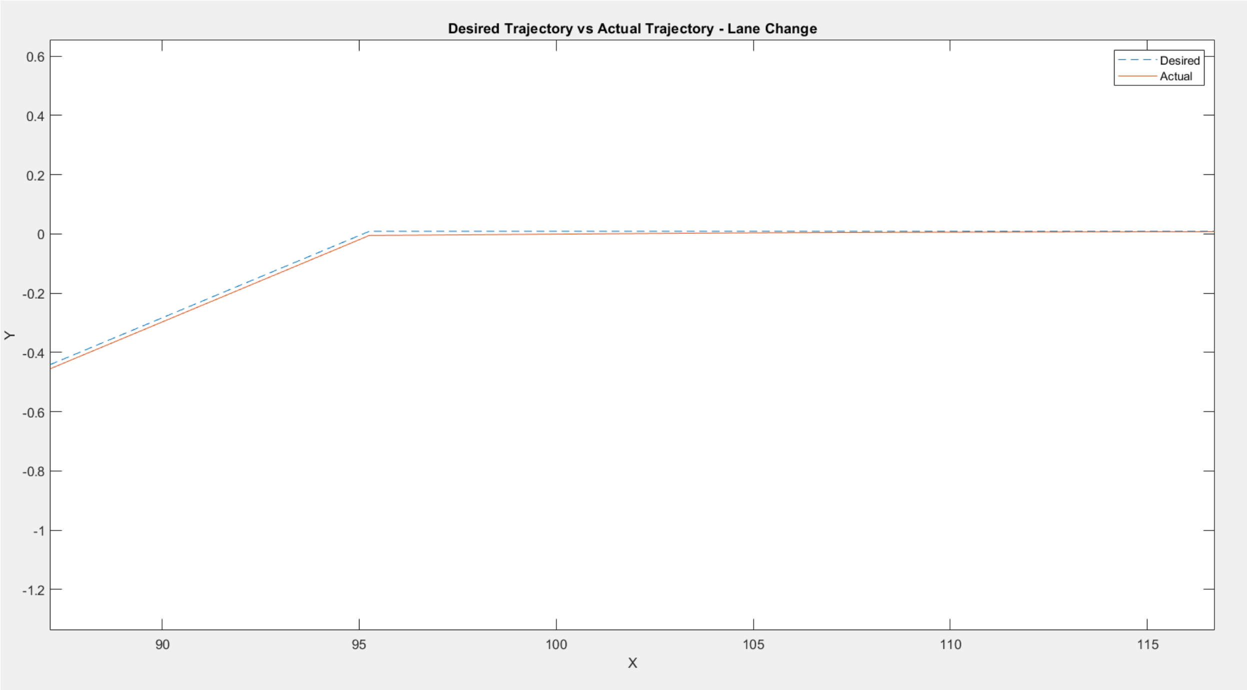
Ans: The following is the desired path vs actual path:



Following is the zoomed in graph when the vehicle is leaving the lane:



Following is the zoomed in graph when the vehicle is entering or converging into the lane:

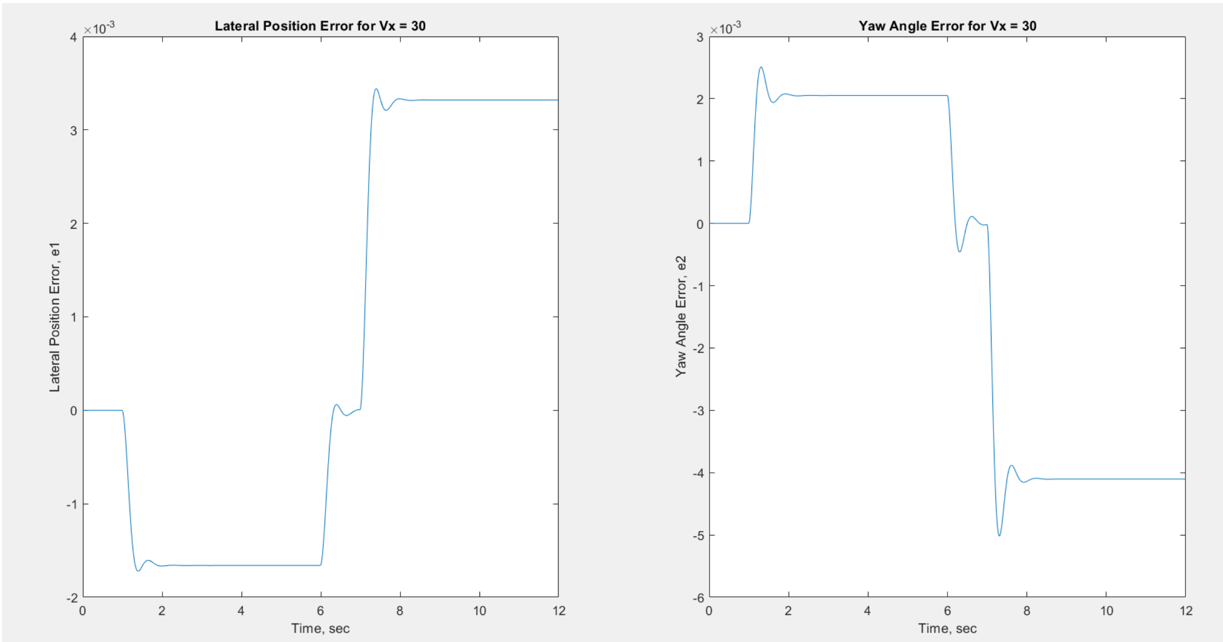


2.4:

Ans: I used the LQR over pole placement method for controller design. The poles (eigenvalues) I found were:

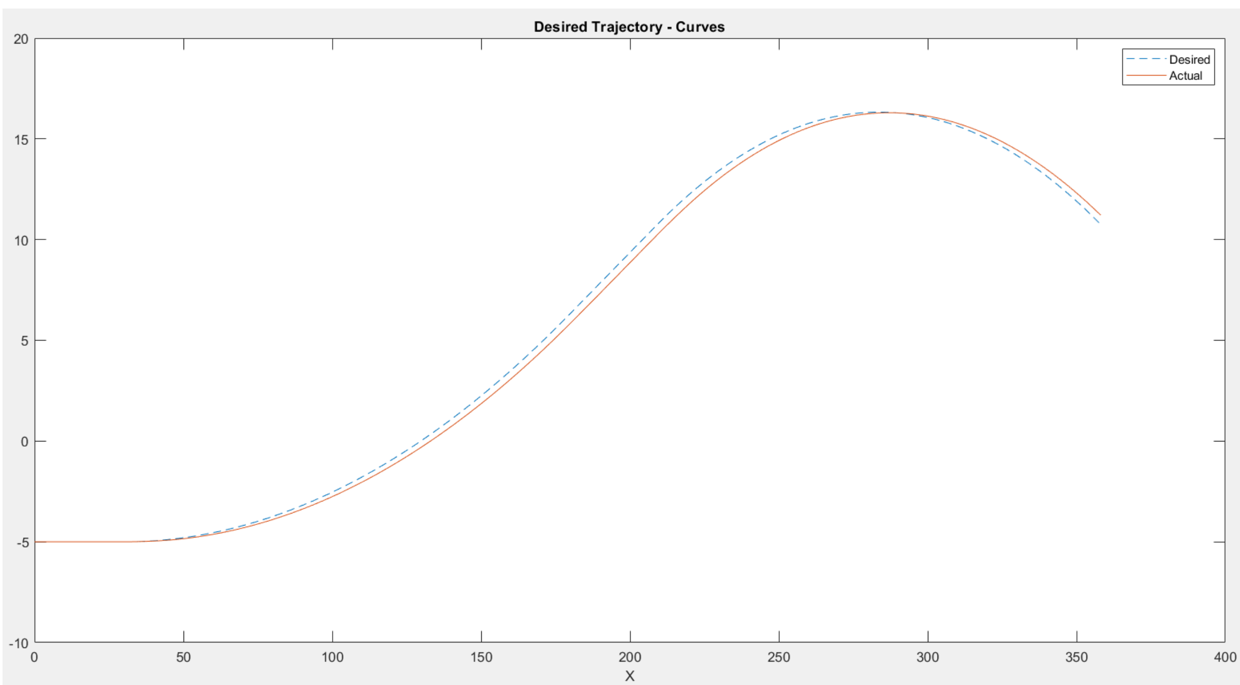
```
>> bend_curve_task(vx, A, B1, B2, C);
Closed-loop poles are:
   1.0e+03 *

  -1.1875 + 0.0000i
  -0.0064 + 0.0000i
  -0.0047 + 0.0102i
  -0.0047 - 0.0102i
```

The plot for lateral error “e1” and yaw angle error “e2” is:


Lateral Position Error for Vx = 30
Lateral Position Error, e1
Time, sec
Yaw Angle Error for Vx = 30
Yaw Angle Error, e2
Time, sec

2.5:

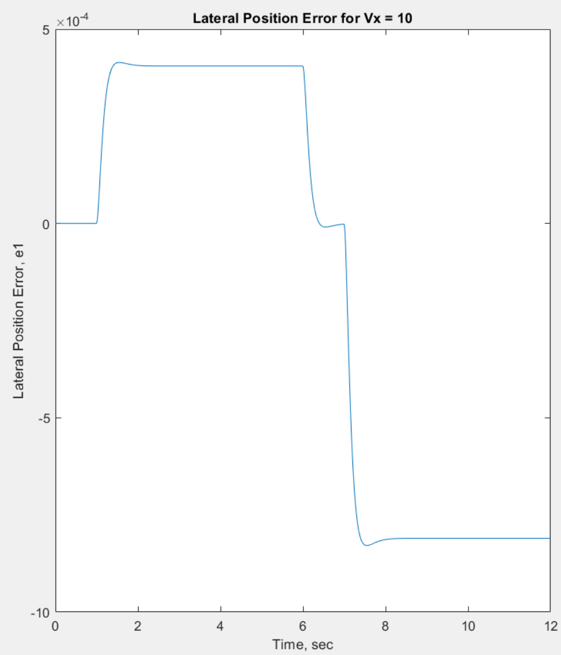
Ans:

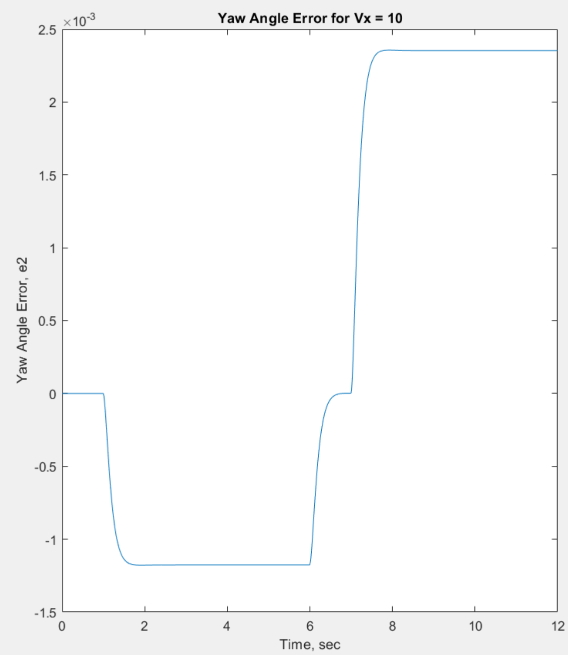

Desired Trajectory - Curves
Desired
Actual
X

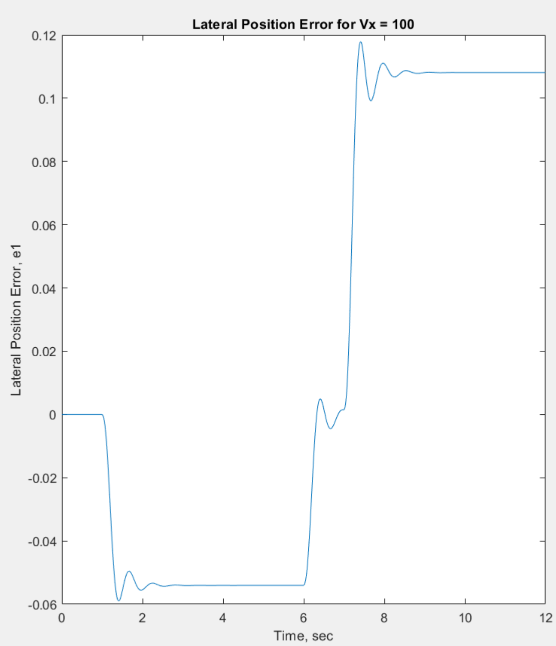
2.6:

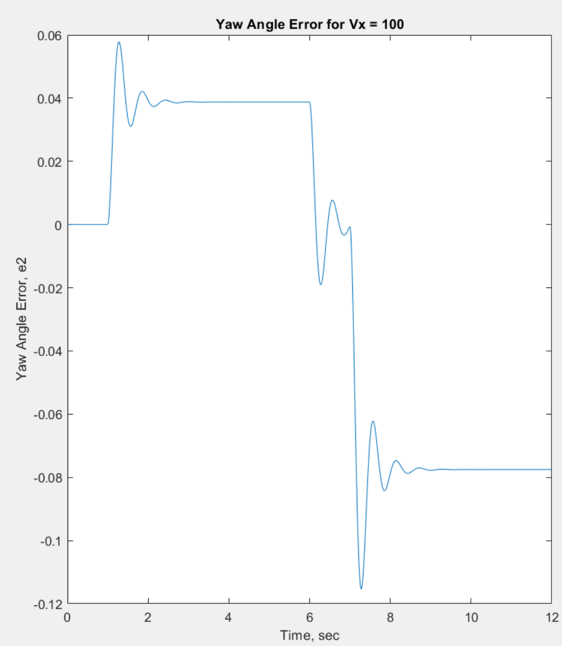
Submitted by: Shahram Najam Syed Dated: 21st September, 2023

Ans: For Vx=10 vs Vx=100, the lateral error e1 and yaw error e2 remains within the bounds of the constraints provided within question 2.2 for Vx=10, with lesser oscillations. This suggests that at lower velocities the lateral position and yaw orientation is better tracked as compared to higher velocities.









3.1:

Ans: The equations used to implement the pure pursuit algorithm are as follows:

$$yaw_{target} = atan2\left(\frac{dy}{dx}\right) = atan2\left(\frac{target_y - current_y}{target_x - current_x}\right)$$

$$yaw_{error} = yaw_{target} - yaw_{current}$$

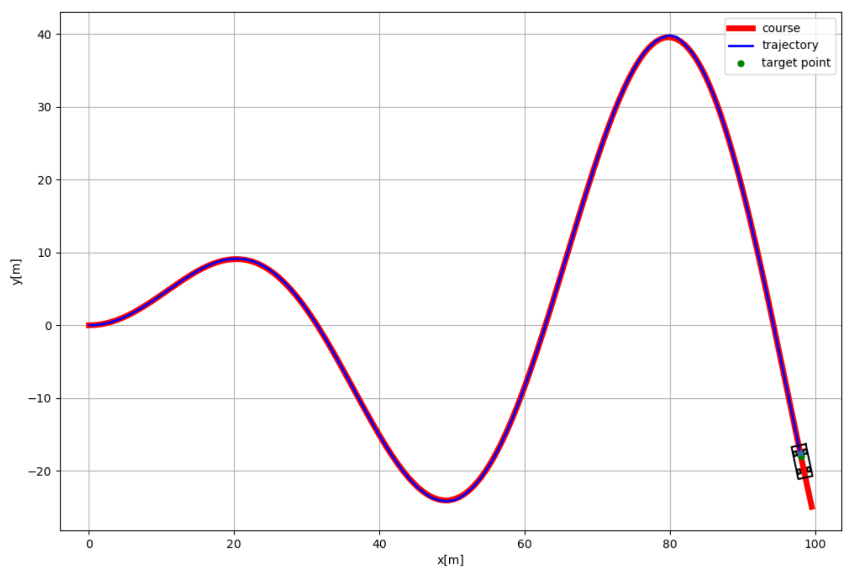
$$\delta = atan2\left(\frac{2\ WB\ \sin(yaw_{error})}{L_d}\right)$$

Where,

WB = wheelbase length
L_d = Lookahead distance

3.2:

Ans:



3.3:

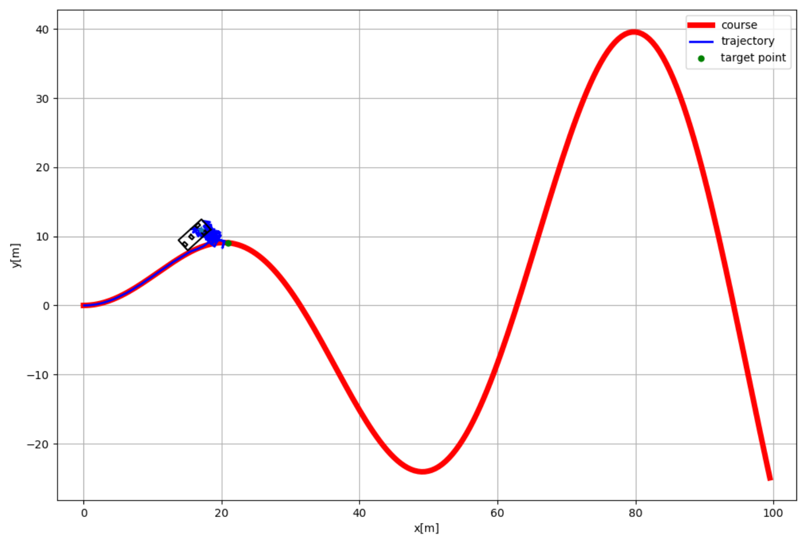
Ans: For this question I varied the value of lookahead distance L_d, and the particular select I used was

L_d = 0.5, 5, 50

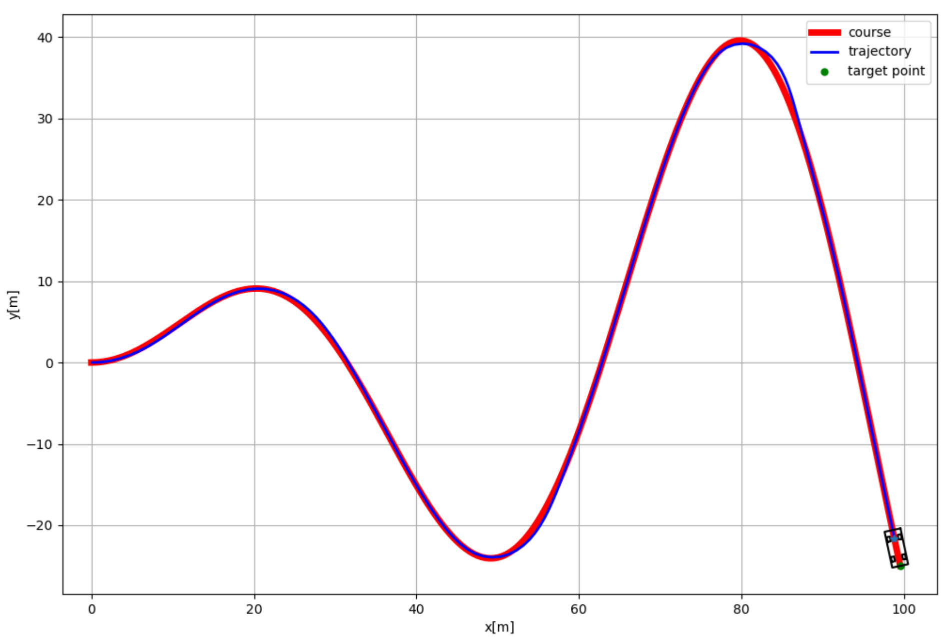
For 0.5 the target distance is way to small to properly follow the waypoint, hence looses the direction fairly quickly early on for fairly small corner banking angles at slower speeds or along a straight line this value would work. For 5.0, the vehicle does reach the end goal but exhibits overshoots during corner turning.

Finally for the absurd value of lookahead distance of 50, the vehicle completely looses the actual course, and tries to reach the final location (as the target point swiftly converges to the end of the course).

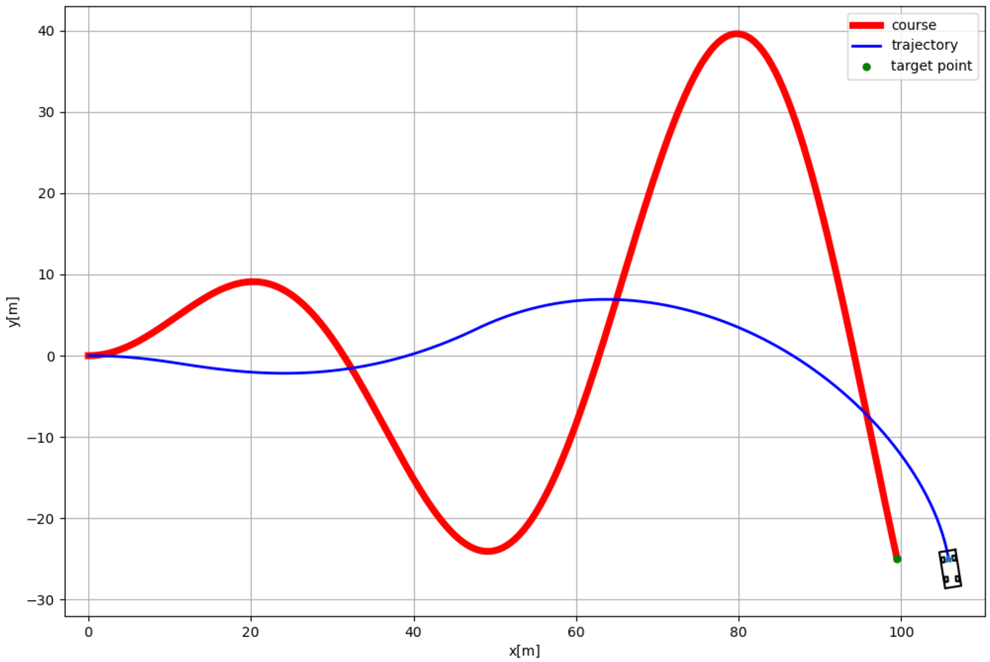
Lookahead distance = 0.5:



Lookahead distance = 5:



Lookahead distance = 50:


course
trajectory
target point
40
30
20
10
0
−10
−20
−30
y[m]
0
20
40
60
80
100
x[m]

Submitted by: Shahram Najam Syed Dated: 21st September, 2023

Code:

Q1.4(a): constant_steering.m

```
% Initialization
V = 10; % Constant velocity
lf = 1.5; lr = 1.5; % Parameters
sampling_time = 0.01;
t_end = 20;
time = 0:sampling_time:t_end;

% Initial conditions
X = zeros(1, length(time));
Y = zeros(1, length(time));
psi = zeros(1, length(time));

% Case A: Constant steering
delta_values = [0.25, 0.5, 0.75, 1.0];
figure(1);
title('Constant Steering');
hold on;
for delta = delta_values
    for i = 2:length(time)
        X(i) = X(i-1) + V*cos(psi(i-1))*sampling_time;
        Y(i) = Y(i-1) + V*sin(psi(i-1))*sampling_time;
        psi(i) = psi(i-1) + (V*tan(delta)/(lf + lr))*sampling_time;
    end
    plot(X,Y, 'DisplayName', ['\delta = ', num2str(delta)]);
end
legend;
xlabel('X Position');
ylabel('Y Position');
```

Q1.4(b): sinusoidal_steering.m

```
% Initialization
V = 10; % Constant velocity
lf = 1.5; lr = 1.5; % Parameters
sampling_time = 0.01;
t_end = 20;
time = 0:sampling_time:t_end;

% Initial conditions
X = zeros(1, length(time));
Y = zeros(1, length(time));
psi = zeros(1, length(time));

% Case B: Sinusoidal steering
V_values = [2, 4, 6, 8];
amplitude_values = [0.1, 0.3, 0.5];
figure(2);
title('Sinusoidal Steering');
hold on;

for V = V_values
    for amplitude = amplitude_values
```

```
        X(1) = 0; Y(1) = 0; psi(1) = 0; % Reset initial conditions
        for i = 2:length(time)
            delta = amplitude * sin(time(i));
            X(i) = X(i-1) + V*cos(psi(i-1))*sampling_time;
            Y(i) = Y(i-1) + V*sin(psi(i-1))*sampling_time;
            psi(i) = psi(i-1) + (V*tan(delta)/(lf + lr))*sampling_time;
        end
        plot(X,Y, 'DisplayName', ['V = ', num2str(V), ', amp = ',
num2str(amplitude)]);
    end
end

legend;
xlabel('X Position');
ylabel('Y Position');
```

Q1.4(c): square_steering.m

```
% Initialization
V = 10; % Constant velocity
lf = 1.5; lr = 1.5; % Parameters
sampling_time = 0.01;
t_end = 20;
time = 0:sampling_time:t_end;

% Initial conditions
X = zeros(1, length(time));
Y = zeros(1, length(time));
psi = zeros(1, length(time));

% Case C: Square wave steering
V_values = [2, 4, 6, 8];
amplitude_values = [0.1, 0.3, 0.5];
figure(3);
title('Square Wave Steering');
hold on;

for V = V_values
    for amplitude = amplitude_values
        X(1) = 0; Y(1) = 0; psi(1) = 0; % Reset initial conditions
        for i = 2:length(time)
            delta = amplitude * sign(sin(time(i)));
            X(i) = X(i-1) + V*cos(psi(i-1))*sampling_time;
            Y(i) = Y(i-1) + V*sin(psi(i-1))*sampling_time;
            psi(i) = psi(i-1) + (V*tan(delta)/(lf + lr))*sampling_time;
        end
        plot(X,Y, 'DisplayName', ['V = ', num2str(V), ', amp = ',
num2str(amplitude)]);
    end
end

legend;
xlabel('X Position');
ylabel('Y Position');
```

Q2.1: init_DBM.m

```
% DBM Model Setup
% ------------------------

% Vehicle Parameters
vx = 30;          % Vehicle longitudinal speed (m/s)
m  = 1573;        % Vehicle mass (kg)
Iz = 2873;        % Vehicle yaw moment of inertia (kg-m^2)
lf = 1.10;        % Distance from CoG to front axle (m)
lr = 1.58;        % Distance from CoG to rear axle (m)
Caf = 80000;      % Cornering stiffness at front tires (N/rad)
Car = 80000;      % Cornering stiffness at rear tires (N/rad)

% Initialize System Dynamics Matrices
A = zeros(4, 4);
B1 = zeros(4, 1);
B2 = zeros(4, 1);
C = [1 0 0 0;
     0 0 1 0];   % For lateral error e1 and yaw error e2
D = [0;
     0];

A(1, 2) = 1;
A(2, :) = [0, -(2 * Caf + 2 * Car) / (m * vx),  (2 * Caf + 2 * Car) / m, -(2
* Caf * lf - 2 * Car * lr) / (m * vx)];
A(3, 4) = 1;
A(4, :) = [0, -(2 * Caf * lf - 2 * Car * lr) / (Iz * vx),  (2 * Caf * lf - 2
* Car * lr) / Iz, -(2 * Caf * lf^2 + 2 * Car * lr^2) / (Iz * vx)];

B1(2) = 2 * Caf / m;
B1(4) = 2 * Caf * lf / Iz;

B2(2) = -(2 * Caf * lf - 2 * Car * lr) / (m * vx) - vx;
B2(4) = -(2 * Caf * lf^2 + 2 * Car * lr^2) / (Iz * vx);

% Open Loop System
open_system = ss(A, B1, C, D);

% Display Eigenvalues and controllability
fprintf('Eigenvalues of A: \n');
disp(eig(A));
fprintf('Rank of controllability matrix: %d\n', rank(ctrb(open_system)));
```

Q2.2 & 2.3: lane_change_task.m

```
function lane_change_task(vehicleVelocity, A, B1, B2, C)
    timeInterval = 0.01;
    totalSimulationSteps = determineTotalSteps(vehicleVelocity,
timeInterval);

    [desiredLaneOrientation, rateOfLaneOrientationChange] =
determineLaneChangeParameters(vehicleVelocity, timeInterval,
totalSimulationSteps);
    controlGain = determineLQRControlGains(A, B1, C);
```

```
    closedLoopSystem = defineClosedLoopSystem(A, B1, B2, C, controlGain);
    [response, simulationTimestamp, systemState, lateralPositionError, 
yawAngleError] = generateSystemResponse(closedLoopSystem, 
rateOfLaneOrientationChange, timeInterval, totalSimulationSteps);

    [steeringAngleChange, rateOfSteeringChange] = 
computeControlInputs(systemState, controlGain, timeInterval);

    displaySimulationResults(simulationTimestamp, rateOfSteeringChange, 
lateralPositionError, yawAngleError);
    assessSimulationPerformance(vehicleVelocity, timeInterval, 
rateOfSteeringChange, lateralPositionError, yawAngleError);

    visualizePathComparison(vehicleVelocity, timeInterval, 
totalSimulationSteps, desiredLaneOrientation, lateralPositionError, 
yawAngleError);
end

function totalSteps = determineTotalSteps(vehicleVelocity, timeInterval)
    travelDistances = [5, 90, 5 + vehicleVelocity];
    stepsForEachDistance = arrayfun(@(distance) round(distance / 
(vehicleVelocity * timeInterval)), travelDistances);
    totalSteps = sum(stepsForEachDistance) + 2;
end

function [desiredOrientation, rateOfOrientationChange] = 
determineLaneChangeParameters(vehicleVelocity, timeInterval, totalSteps)
    determineStepsRequired = @(distance) round(distance / (vehicleVelocity * 
timeInterval)) + 1;

    initialSteps = determineStepsRequired(5);
    middleSteps = determineStepsRequired(90);
    finalSteps = determineStepsRequired(5 + vehicleVelocity) - 1;

    initialOrientation = zeros(1, initialSteps);
    middleOrientation = ones(1, middleSteps) * atan(5 / 90);
    finalOrientation = zeros(1, finalSteps);

    desiredOrientation = [initialOrientation, middleOrientation, 
finalOrientation, 0];
    rateOfOrientationChange = diff(desiredOrientation);
end

function controlGain = determineLQRControlGains(systemMatrixA, inputMatrixB1, 
outputMatrixC)
    weightMatrixQ = diag([10, 1, 5, 1]);
    weightMatrixR = 5;
    controlGain = lqr(systemMatrixA, inputMatrixB1, weightMatrixQ, 
weightMatrixR);
end

function closedLoop = defineClosedLoopSystem(systemMatrixA, inputMatrixB1, 
inputMatrixB2, outputMatrixC, controlGain)
    adjustedSystemMatrix = systemMatrixA - inputMatrixB1 * controlGain;
    closedLoop = ss(adjustedSystemMatrix, inputMatrixB2, outputMatrixC, 0);
end
```

```
function [response, simulationTimestamp, systemState, lateralError, yawError]
= generateSystemResponse(closedLoop, rateOfOrientationChange, timeInterval,
totalSteps)
    simulationTimestamp = 0:timeInterval:(totalSteps - 1) * timeInterval;
    [response, ~, systemState] = lsim(closedLoop, rateOfOrientationChange,
simulationTimestamp);

    lateralError = systemState(:, 1);
    yawError = systemState(:, 3);
end
function [steeringChange, rateOfChange] = computeControlInputs(systemState,
controlGain, timeInterval)
    steeringChange = -controlGain * systemState';
    rateOfChange = diff([steeringChange, 0]) / timeInterval;
end

function displaySimulationResults(simulationTimestamp, rateOfSteeringChange,
lateralError, yawError)
    figure();

    subplot(3, 1, 1);
    plot(simulationTimestamp, rateOfSteeringChange);
    title('Rate of Steering');
    xlabel('Time (s)');
    ylabel('Steering Rate (rad/s)');

    subplot(3, 1, 2);
    plot(simulationTimestamp, lateralError);
    title('Lateral Position Error');
    xlabel('Time (s)');
    ylabel('Lateral Error (m)');

    subplot(3, 1, 3);
    plot(simulationTimestamp, yawError);
    title('Yaw Angle Error');
    xlabel('Time (s)');
    ylabel('Yaw Error (rad)');
end

function assessSimulationPerformance(vehicleVelocity, timeInterval,
rateOfSteeringChange, lateralError, yawError)
    timeForInitialPosition = round(5 / (vehicleVelocity * timeInterval));
    timeForMiddlePosition = round(90 / (vehicleVelocity * timeInterval));

    thresholdsForInitial = [0.002, 0.0007];
    thresholdsForMiddle = [0.002, 0.0007];

    checkConstraints(rateOfSteeringChange, 25.0, 'Steering exceed set
threshold.');
    checkConstraints(lateralError, 0.01, 'Max threshold for lateral error
exceeded.');
    checkConstraintsDuringTransition(lateralError, yawError,
timeForInitialPosition, thresholdsForInitial, 'Transition 1 Conditions NOT
Satisfied');
    checkConstraintsDuringTransition(lateralError, yawError,
timeForMiddlePosition, thresholdsForMiddle, 'Transition 2 Conditions NOT
Satisfied');
```

```
end

function checkConstraints(value, threshold, errorMessage)
    if max(abs(value)) >= threshold
        fprintf('[Failure]: %s %f!\n', errorMessage, max(abs(value)));
    end
end

function checkConstraintsDuringTransition(lateralError, yawError,
transitionTime, thresholds, errorMessage)
    if max(abs(lateralError(transitionTime))) > thresholds(1) &&
max(abs(yawError(transitionTime))) > thresholds(2)
        fprintf('[Failure] %s (%f, %f)\n', errorMessage,
max(abs(lateralError(transitionTime))), max(abs(yawError(transitionTime))));
    end
end

function visualizePathComparison(vehicleVelocity, timeInterval, totalSteps,
desiredOrientation, lateralError, yawError)
    [desiredX, desiredY] = computeDesiredTrajectory(vehicleVelocity,
timeInterval, totalSteps, desiredOrientation);
    [actualX, actualY] = computeActualTrajectory(vehicleVelocity,
timeInterval, totalSteps, desiredOrientation, lateralError, yawError);

    plotTrajectories(desiredX, desiredY, actualX, actualY);
end

function [desiredX, desiredY] = computeDesiredTrajectory(vehicleVelocity,
timeInterval, totalSteps, desiredOrientation)
    desiredX = zeros(1, totalSteps);
    desiredY = zeros(1, totalSteps);

    for k = 2:totalSteps
        desiredX(k) = desiredX(k-1) + vehicleVelocity * timeInterval *
cos(desiredOrientation(k));
        desiredY(k) = desiredY(k-1) + vehicleVelocity * timeInterval *
sin(desiredOrientation(k));
    end

    return;
end

function [actualX, actualY] = computeActualTrajectory(vehicleVelocity,
timeInterval, totalSteps, desiredOrientation, lateralError, yawError)
    actualX = zeros(1, totalSteps);
    actualY = zeros(1, totalSteps);

    for i = 2:totalSteps
        wpsi = desiredOrientation(i) + yawError(i);
        actualX(i) = actualX(i-1) + vehicleVelocity * timeInterval *
cos(wpsi) + lateralError(i) * sin(wpsi);
        actualY(i) = actualY(i-1) + vehicleVelocity * timeInterval *
sin(wpsi) + lateralError(i) * cos(wpsi);
    end

    return;
end
```

```
function plotTrajectories(desiredX, desiredY, actualX, actualY)
    figure();
    plot(desiredX, desiredY, '--');
    hold on;
    plot(actualX, actualY);
    title('Desired vs Actual Trajectory');
    legend('Desired', 'Actual');
    xlabel('X Position (m)');
    ylabel('Y Position (m)');
end
```

Q 2.4, 2.5 & 2.6: bend_curve_task.m

```
function bend_curve_task(vx, A, B1, B2, C)
    % Define time step and total durations
    timeStep = 0.01;
    maxTime = 12;
    totalSteps = round(maxTime / timeStep);
    straightDurationSteps = round(1 / timeStep);
    curveDurationSteps = round(5 / timeStep);

    radiusOuter = 1000;
    radiusInner = 500;

    % Calculate yaw rates during maneuver
    yawRates = [zeros(1, straightDurationSteps), ...
                ones(1, curveDurationSteps) * vx / radiusOuter, ...
                zeros(1, straightDurationSteps), ...
                ones(1, curveDurationSteps) * vx * (-1) / radiusInner];

    % Calculate orientations over time
    orientations = [0, cumsum(yawRates) * timeStep];

    % LQR Controller Configuration
    Q = [400 0    0    0; 0    10    0    0; 0    0    50   0; 0    0    0    10];
    R = 0.1;
    controller = lqr(A, B1, Q, R);

    % Closed Loop System Dynamics
    closedLoopSystemMatrix = (A - B1 * controller);
    systemOutputMatrix = B2;

    % Closed Loop System
    closedLoopSystem = ss(closedLoopSystemMatrix, systemOutputMatrix, C, 0);

    % Simulation Response
    timeInterval = 0:timeStep:(totalSteps - 1) * timeStep;
    [response, timeOut, stateValues] = lsim(closedLoopSystem, yawRates,
timeInterval);
    lateralError = stateValues(:, 1);
    yawError = stateValues(:, 3);

    % Plotting Errors
    plotErrors(timeOut, lateralError, yawError);
```

```
    % Check for constraint violations
    verifyConstraints(lateralError, yawError);

    % Calculate Desired and Actual Trajectories
    [desiredX, desiredY] = calculateTrajectory(vx, timeStep, orientations,
totalSteps, [0, -5]);
    [actualX, actualY] = calculateTrajectory(vx, timeStep, orientations,
totalSteps, [0, -5], lateralError, yawError);

    % Plot Trajectories
    plotTrajectories(desiredX, desiredY, actualX, actualY);
end

function plotErrors(time, lateralErr, yawErr)
    figure();
    subplot(1, 2, 1);
    plot(time, lateralErr);
    title("Lateral Position Error for Vx = 30");
    xlabel("Time, sec");
    ylabel("Lateral Position Error, e1");

    subplot(1, 2, 2);
    plot(time, yawErr);
    title("Yaw Angle Error for Vx = 30");
    xlabel("Time, sec");
    ylabel("Yaw Angle Error, e2");
end

function verifyConstraints(lateralErr, yawErr)
    maxLateralError = 0.01;
    maxYawError = 0.0007;

    if max(abs(lateralErr)) > maxLateralError && max(abs(yawErr)) >
maxYawError
        fprintf("[Failure]Threshold(s) not satisfied (%f, %f)!\n",
max(abs(lateralErr)), max(abs(yawErr)));
    end
end

function plotTrajectories(desiredX, desiredY, actualX, actualY)
    figure();
    plot(desiredX, desiredY, '--');
    hold on;
    plot(actualX, actualY);
    title("Desired vs Actual Trajectory");
    legend('Desired', 'Actual');
    xlabel("X");
    ylabel("Y");
end

function [xPath, yPath] = calculateTrajectory(velocity, timeStep,
orientations, totalSteps, startPos, lateralErr, yawErr)
    if nargin < 6
        lateralErr = zeros(1, totalSteps);
        yawErr = zeros(1, totalSteps);
    end
```

```
    xPath = zeros(1, totalSteps);
    yPath = zeros(1, totalSteps);
    xPath(1) = startPos(1);
    yPath(1) = startPos(2);

    for k = 2:totalSteps
        effectiveOrientation = orientations(k) + yawErr(k);
        xPath(k) = xPath(k-1) + velocity * timeStep *
cos(effectiveOrientation) + lateralErr(k) * sin(effectiveOrientation);
        yPath(k) = yPath(k-1) + velocity * timeStep *
sin(effectiveOrientation) + lateralErr(k) * cos(effectiveOrientation);
    end
end
-----------------------------------------------------------------------------
```

Q3.2 & 3.3: purepursuit 16665.py

```
"""

Path tracking simulation with pure pursuit steering control and PID speed
control.

"""


import math
import matplotlib.pyplot as plt
import numpy as np

# Pure Pursuit parameters
L = 1.0  # look ahead distance
dt = 0.1  # discrete time

# Vehicle parameters (m)
LENGTH = 4.5         #length of the vehicle (for the plot)
WIDTH = 2.0          #length of the vehicle (for the plot)
BACKTOWHEEL = 1.0    #length of the vehicle (for the plot)
WHEEL_LEN = 0.3      #length of the vehicle (for the plot)
WHEEL_WIDTH = 0.2    #length of the vehicle (for the plot)
TREAD = 0.7          #length of the vehicle (for the plot)
WB = 2.5             # wheel-base


def plotVehicle(x, y, yaw, steer=0.0, cabcolor="-r", truckcolor="-k"):

    outline = np.array(
        [
            [
```

```
                -BACKTOWHEEL,
                (LENGTH - BACKTOWHEEL),
                (LENGTH - BACKTOWHEEL),
                -BACKTOWHEEL,
                -BACKTOWHEEL,
            ],
            [WIDTH / 2, WIDTH / 2, -WIDTH / 2, -WIDTH / 2, WIDTH / 2],
        ]
    )

    fr_wheel = np.array(
        [
            [WHEEL_LEN, -WHEEL_LEN, -WHEEL_LEN, WHEEL_LEN, WHEEL_LEN],
            [
                -WHEEL_WIDTH - TREAD,
                -WHEEL_WIDTH - TREAD,
                WHEEL_WIDTH - TREAD,
                WHEEL_WIDTH - TREAD,
                -WHEEL_WIDTH - TREAD,
            ],
        ]
    )

    rr_wheel = np.copy(fr_wheel)

    fl_wheel = np.copy(fr_wheel)
    fl_wheel[1, :] *= -1
    rl_wheel = np.copy(rr_wheel)
    rl_wheel[1, :] *= -1

    Rot1 = np.array([[math.cos(yaw), math.sin(yaw)], [-math.sin(yaw),
math.cos(yaw)]])
    Rot2 = np.array(
        [[math.cos(steer), math.sin(steer)], [-math.sin(steer), math.cos(steer)]]
    )

    fr_wheel = (fr_wheel.T.dot(Rot2)).T
    fl_wheel = (fl_wheel.T.dot(Rot2)).T
    fr_wheel[0, :] += WB
    fl_wheel[0, :] += WB

    fr_wheel = (fr_wheel.T.dot(Rot1)).T
    fl_wheel = (fl_wheel.T.dot(Rot1)).T

    outline = (outline.T.dot(Rot1)).T
```

```
    rr_wheel = (rr_wheel.T.dot(Rot1)).T
    rl_wheel = (rl_wheel.T.dot(Rot1)).T

    outline[0, :] += x
    outline[1, :] += y
    fr_wheel[0, :] += x
    fr_wheel[1, :] += y
    rr_wheel[0, :] += x
    rr_wheel[1, :] += y
    fl_wheel[0, :] += x
    fl_wheel[1, :] += y
    rl_wheel[0, :] += x
    rl_wheel[1, :] += y

    plt.plot(
        np.array(outline[0, :]).flatten(), np.array(outline[1, :]).flatten(),
truckcolor
    )
    plt.plot(
        np.array(fr_wheel[0, :]).flatten(),
        np.array(fr_wheel[1, :]).flatten(),
        truckcolor,
    )
    plt.plot(
        np.array(rr_wheel[0, :]).flatten(),
        np.array(rr_wheel[1, :]).flatten(),
        truckcolor,
    )
    plt.plot(
        np.array(fl_wheel[0, :]).flatten(),
        np.array(fl_wheel[1, :]).flatten(),
        truckcolor,
    )
    plt.plot(
        np.array(rl_wheel[0, :]).flatten(),
        np.array(rl_wheel[1, :]).flatten(),
        truckcolor,
    )
    plt.plot(x, y, "*")


def getDistance(p1, p2):
    """
    Calculate distance
    :param p1: list, point1
```

```
    :param p2: list, point2
    :return: float, distance
    """
    dx = p1[0] - p2[0]
    dy = p1[1] - p2[1]
    return math.hypot(dx, dy)


class Vehicle:
    def __init__(self, x, y, yaw, vel=0):
        """
        Define a vehicle class (state of the vehicle)
        :param x: float, x position
        :param y: float, y position
        :param yaw: float, vehicle heading
        :param vel: float, velocity

        """
        # State of the vehicle

        self.x = x #x coordinate of the vehicle
        self.y = y #y coordinate of the vehicle
        self.yaw = yaw  #yaw of the vehicle
        self.vel = vel  #velocity of the vehicle

    def update(self, acc, delta):
        """
        Vehicle motion model, here we are using simple bycicle model
        :param acc: float, acceleration
        :param delta: float, heading control
        """

        # TODO- update the state of the vehicle (x,y,yaw,vel) based on simple
bicycle model
        self.vel += acc * dt
        self.x += self.vel * math.cos(self.yaw) * dt
        self.y += self.vel * math.sin(self.yaw) * dt
        self.yaw += self.vel / WB * math.tan(delta) * dt

class Trajectory:
    def __init__(self, traj_x, traj_y):
        """
        Define a trajectory class
        :param traj_x: list, list of x position
        :param traj_y: list, list of y position
```

```
        """
        self.traj_x = traj_x
        self.traj_y = traj_y
        self.last_idx = 0

    def getPoint(self, idx):
        return [self.traj_x[idx], self.traj_y[idx]]

    def getTargetPoint(self, pos):
        """
        Get the next look ahead point
        :param pos: list, vehicle position
        :return: list, target point
        """
        target_idx = self.last_idx
        target_point = self.getPoint(target_idx)
        curr_dist = getDistance(pos, target_point)

        while curr_dist < L and target_idx < len(self.traj_x) - 1:
            target_idx += 1
            target_point = self.getPoint(target_idx)
            curr_dist = getDistance(pos, target_point)

        self.last_idx = target_idx
        return self.getPoint(target_idx)


class Controller:
    def __init__(self, kp=1.0, ki=0.1):
        """
        Define a PID controller class
        :param kp: float, kp coeff
        :param ki: float, ki coeff
        :param kd: float, kd coeff
        """
        self.kp = kp
        self.ki = ki
        self.Pterm = 0.0
        self.Iterm = 0.0
        self.last_error = 0.0

    def Longitudinalcontrol(self, error):
        """
        PID main function, given an input, this function will output a
acceleration for longitudinal error
```

```
        :param error: float, error term
        :return: float, output control
        """
        self.Pterm = self.kp * error
        self.Iterm += error * dt

        self.last_error = error
        output = self.Pterm + self.ki * self.Iterm
        return output

    def PurePursuitcontrol(self, error):
        #TODO- find delta

        delta = math.atan2((2 * WB * math.sin(error)), L)

        return delta

def main():
    # create vehicle
    ego = Vehicle(0, 0, 0)
    plotVehicle(ego.x, ego.y, ego.yaw)

    # target velocity
    target_vel = 10

    # target course
    traj_x = np.arange(0, 100, 0.5)
    traj_y = [math.sin(x / 10.0) * x / 2.0 for x in traj_x]
    traj = Trajectory(traj_x, traj_y)
    goal = traj.getPoint(len(traj_x) - 1)

    # create longitudinal and pure pursuit controller
    PI_acc = Controller()
    PI_yaw = Controller()

    # real trajectory
    traj_ego_x = []
    traj_ego_y = []

    plt.figure(figsize=(12, 8))

    while getDistance([ego.x, ego.y], goal) > 1:
        target_point = traj.getTargetPoint([ego.x, ego.y])

        # use PID to control the speed vehicle
```

```
        vel_err = target_vel - ego.vel
        acc = PI_acc.Longitudinalcontrol(vel_err)

        # use pure pursuit to control the heading of the vehicle
        # TODO- Calculate the yaw error
        dx = target_point[0] - ego.x
        dy = target_point[1] - ego.y
        target_yaw = math.atan2(dy, dx)
        yaw_err = target_yaw - ego.yaw   #TODO- Update the equation
        delta = PI_yaw.PurePursuitcontrol(yaw_err)  #TODO- update thr Pure
pursuit controller

        # move the vehicle
        ego.update(acc, delta)

        # store the trajectory
        traj_ego_x.append(ego.x)
        traj_ego_y.append(ego.y)

        # plots
        plt.cla()
        plt.plot(traj_x, traj_y, "-r", linewidth=5, label="course")
        plt.plot(traj_ego_x, traj_ego_y, "-b", linewidth=2, label="trajectory")
        plt.plot(target_point[0], target_point[1], "og", ms=5, label="target
point")
        plotVehicle(ego.x, ego.y, ego.yaw, delta)
        plt.xlabel("x[m]")
        plt.ylabel("y[m]")
        plt.axis("equal")
        plt.legend()
        plt.grid(True)
        plt.pause(0.1)


if __name__ == "__main__":
    main()
```

References:

[1] https://dingyan89.medium.com/simple-understanding-of-kinematic-bicycle-model-81cac6420357
[2] Ogata, K. (2010). Modern control engineering (5th ed.). Prentice Hall.
[3] MathWorks Documentation. (n.d.). Control System Toolbox User's Guide. Retrieved from https://www.mathworks.com/help/control/index.htm
[4] Palm, W. J. (2014). Introduction to MATLAB for engineers (3rd ed.). McGraw-Hill.
[5] Rajamani, R. (2011). Vehicle dynamics and control (2nd ed.). Springer Science & Business Media.
[6] Gillespie, T. D. (1992). Fundamentals of vehicle dynamics. Society of Automotive Engineers.
[7] Thrun, S., Burgard, W., & Fox, D. (2005). Probabilistic Robotics. MIT Press.
[8] Correll, N., Hayes, B., & MacDonald, B. A. (2019). Introduction to Autonomous Robots.