16-665
HW1
Boxiang Fu
boxiangf
09/15/2024

Q1.1.

Pepy Model:

$$\dot{X} = V\cos\psi$$
$$\dot{Y} = V\sin\psi$$
$$\dot{\psi} = \frac{V}{l_f + l_r}\tan\delta_f$$

where V is the magnitude of $\vec{v}$

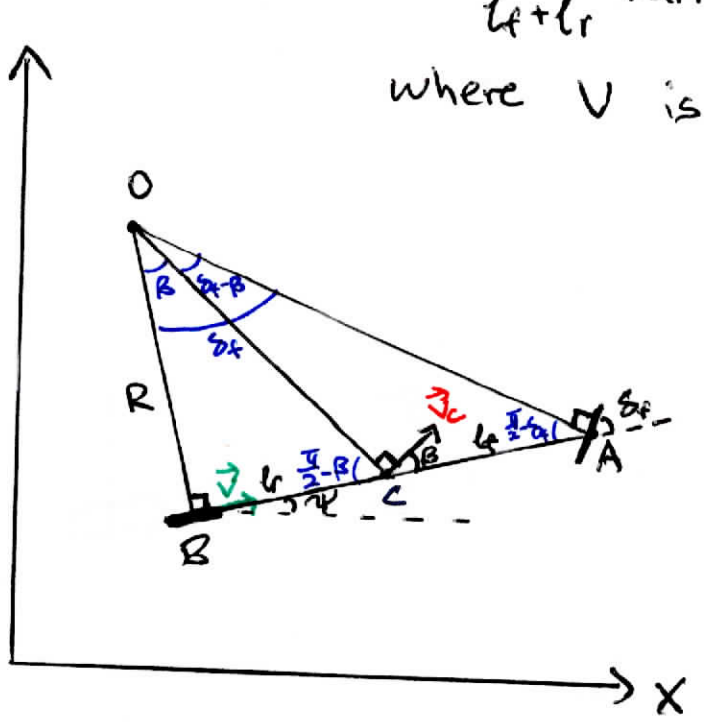


Fig 1: Kinematics of Pepy Model

Referring to Fig 1 with the positions measured relative to point B, we can obtain the angles in blue using basic trigonometry.

Using SOH CAH TOA on triangle OAB:

$$\tan(\angle AOB) = \frac{\text{opposite}(\angle AOB)}{\text{adjacent}(\angle AOB)}$$

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

$$\Rightarrow \frac{1}{R} = \frac{\tan\delta_f}{l_r + l_f} \qquad (\text{Eq. } 1.1)$$

Recall from lectures, for a slow changing turning radius,

$$\dot{\psi} = \frac{V}{R} \quad (\text{Lecture 2 Slide 17}) \qquad (\text{Eq. } 1.2)$$

Substitute Eq. 1.1 into Eq. 1.2, we obtain

$$\dot{\psi} = V \cdot \frac{1}{R}$$

$$= \frac{V}{l_r + l_f}\tan\delta_f \qquad (\text{Eq. } 1.3)$$

$\dot{x}$ and $\dot{y}$ are simply the x- and y-components of $\vec{v}$, thus $\dot{x} = \vec{v}_x$

$$= V\cos\psi \qquad (\text{Eq. } 1.4)$$

$$\dot{y} = \vec{v}_y$$

$$= V\sin\psi \qquad (\text{Eq. } 1.5)$$

Eq. 1.2-1.5 gives the Pepy model as required. □

Q1.2.

Kong Model: $\dot{X} = V\cos(\psi+\beta)$

$\dot{Y} = V\sin(\psi+\beta)$

$\dot{\psi} = \frac{V}{l_r}\sin\beta$

where V is the magnitude of $\vec{v}$

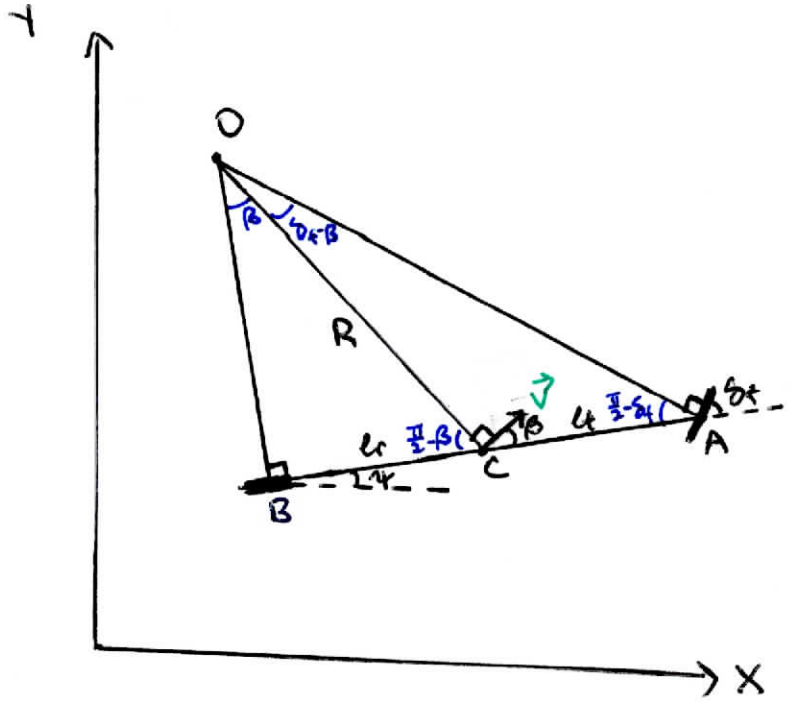


Fig 2: Kinematics of Kong Model

Referring to Fig 2 with positions measured relative to point C, we can obtain the angles in blue using basic trigonometry.

Using SOH CAH TOA on triangle OBC:

$$\sin(\angle BOC) = \frac{\text{opposite}(\angle BOC)}{\text{hypotenuse}(\angle BOC)}$$

$$\Rightarrow \sin\beta = \frac{l_r}{R}$$

$$\Rightarrow \frac{1}{R} = \frac{\sin\beta}{l_r} \quad (\text{Eq. } 1.6)$$

Recall from lectures, for a slow changing turning radius,

$\dot{\psi} = \frac{V}{R}$ (Lecture 2 Slide 17) (Eq. 1.7)

Substitute Eq. 1.6 into Eq. 1.7, we obtain

$$\dot{\psi} = V \cdot \frac{1}{R}$$

$$= \frac{V}{l_r}\sin\beta \quad (\text{Eq. } 1.8)$$

$\dot{X}$ and $\dot{Y}$ are simply the x- and y-components of $\vec{V}$, thus $\dot{X} = \vec{V}_x$

$$= V\cos(\psi+\beta) \quad (\text{Eq. } 1.9)$$

$$\dot{Y} = \vec{V}_y$$

$$= V\sin(\psi+\beta) \quad (\text{Eq. } 1.10)$$

Eq. 1.8-1.10 gives the Kong Model as required.

□

Q1.3.

A. β no longer appears in the Pepy model because $\vec{V}$ is defined at the rear wheel (point B in Fig 1). This does not mean $\beta = 0$.

For the vehicle velocity vector at the center of mass, we can define β relative to $\vec{V}_c$ in Fig 1.

Using SOH CAH TOA on triangle OBC:

$$\tan(\angle BOC) = \frac{\text{opposite}(\angle BOC)}{\text{adjacent}(\angle BOC)}$$

$$\Rightarrow \quad \tan\beta = \frac{l_r}{R}$$

Recall $\frac{1}{R} = \frac{\tan\delta_f}{l_r + l_f}$ (from Eq. 1.1), we get

$$\tan\beta = \frac{l_r \tan\delta_f}{l_r + l_f}$$

$$\therefore \beta = \arctan\left(\frac{l_r \tan\delta_f}{l_r + l_f}\right) \quad \text{(Eq. 1.11)}$$

B.

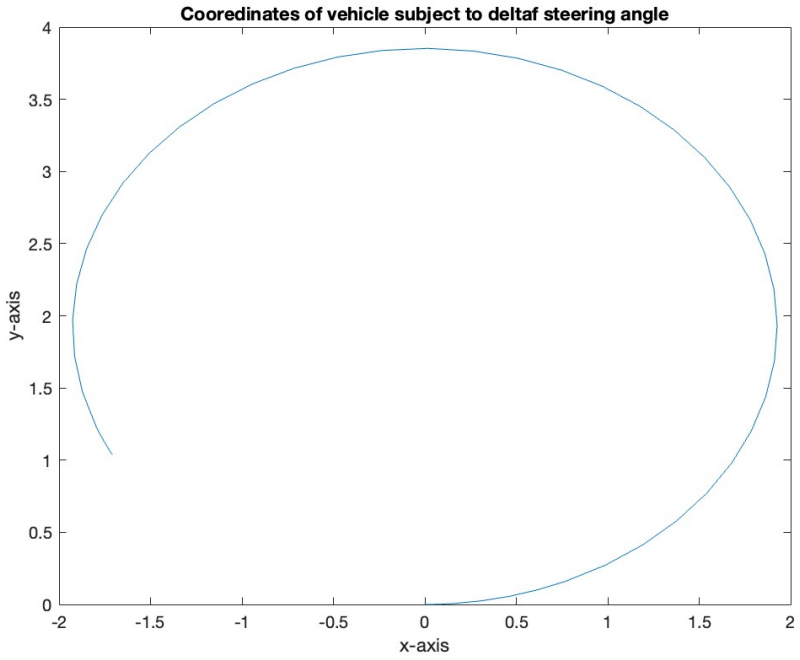
Vehicle slip is the difference between the velocity direction of a vehicle's center of mass and the direction the body of the vehicle is pointing. It is denoted by β in Figs 1 and 2. Tire slip is the difference between the direction a wheel is pointing and the direction the wheel's velocity vector is pointing. They differ as vehicle slip considers the vehicle's slip as a whole (i.e. at center of mass) whereas tire slip only considers slip at each tire.

For the Kinematic Bicycle model, they are independent of each other. In fact, the tire slip angle is assumed to be zero for both front and rear wheels. I.e., the tires move in the same direction they are rolling. Note that this is not true for more complicated models such as the Dynamic Bicycle model where they are dependent on each other.

For the Kinematic Bicycle model, you can have one without the other. Tire slip is always assumed to be zero, whereas vehicle slip is given by Eq. 1.11 and can differ from zero.

Q1.4.

A. Parameters used are: $V=1$, $l_f=1.5$, $l_r=1.5$, $X_0=0$, $Y_0=0$, $\psi_0=0$, $t\in[0,10]$, $\delta_f=1$



Note: Plot starts at (0,0) at $t=0$

For various δ_f constants, the path radius changes. From Eq. 1.1, we have

$$K=\frac{1}{R}=\frac{\tan\delta_f}{l_r+l_f} \quad \text{(curvature)}$$

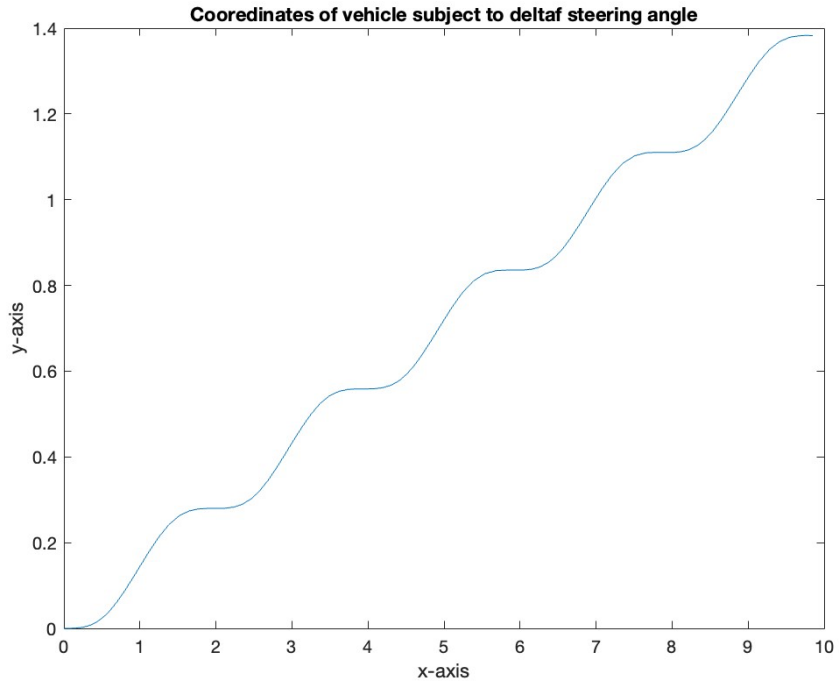
$$\Rightarrow R=\frac{l_r+l_f}{\tan\delta_f} \quad \text{(path radius)}$$

With $l_r=l_f=1.5$, we have

$$K=\frac{\tan\delta_f}{3}, \quad R=\frac{3}{\tan\delta_f}$$

Generally speaking, as steering angle increases, path radius decreases.

B. Parameters used are: $V=1$, $l_f=1.5$, $l_r=1.5$, $X_0=0$, $Y_0=0$, $\psi_0=0$, $t\in[0,10]$, $\delta_f=\sin(\pi t)$



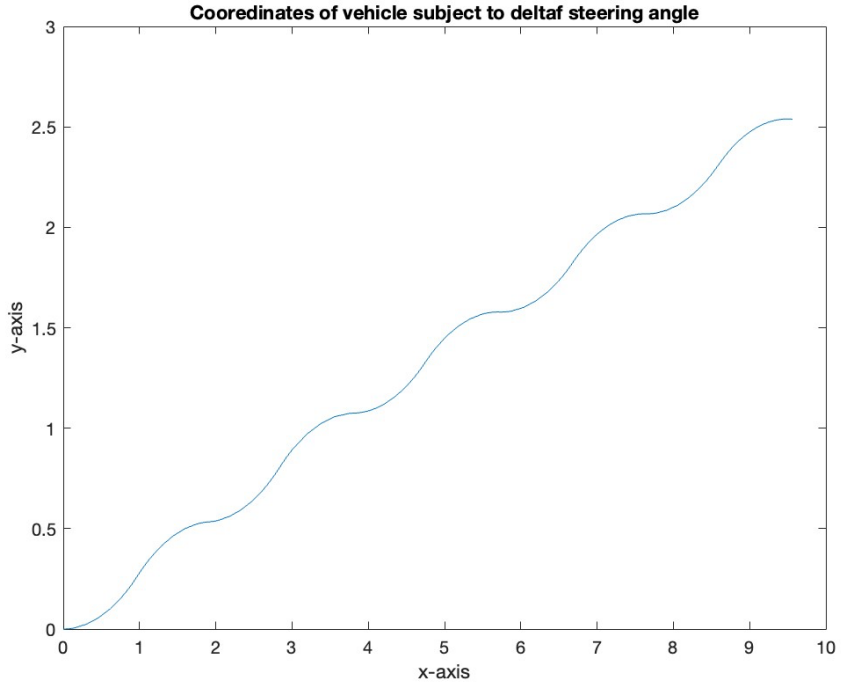
Note: Plot starts at (0,0) at $t=0$

Since we are using a sine function with a period of 2 seconds, we expect the trajectory to move in the positive x, positive y direction. After half a period, the wheels are to the right of the vehicle's body frame, so we expect the y-axis increase to taper off. After each period ($t=2,4,6,\ldots$), we expect the vehicle to be only travelling in the positive x-direction as sine is symmetric about the x-axis relative to a half-period offset. So the anticlockwise rotation exactly cancels with the clockwise rotation after each period. The cycle repeats after each period. This is exactly what we see in the figure above.

As we increase V, we see both an increase in the x and y values obtained after each period. The proportional increase in y is more than the proportional increase in x for a set increase in V. For fast enough velocities, the trajectory wraps around in a curly $\mathcal{L}$ shape and actually starts travelling in the negative y-direction. This was something that I did not expect.

As we increase the amplitude of the sine curve, we see an increase in the y values obtained after each period. The increment in the x values obtained decreases. This is because we turn faster during each period and more of the velocity is "used" to travel in the y-direction instead of the x-direction.

c. Parameters used are: $V=1$, $l_f=1.5$, $l_f=1.5$, $X_0=0$, $Y_0=0$, $\psi_0=0$, $t\in[0,10]$, $\delta_f=\text{square}(\pi t)$



Note: Plot starts at (0,0) at $t=0$

A square wave steering angle is unrealistic as this means we instantaneously change the direction of the wheel discontinuously. We can handle this lack of realism by modelling the change continuously. Some examples that could be used include the sine wave and/or a slanted straight line at the changes in δ_f. This is so the wheel changes direction in a smooth manner and is not discontinuous.

Q2.2.

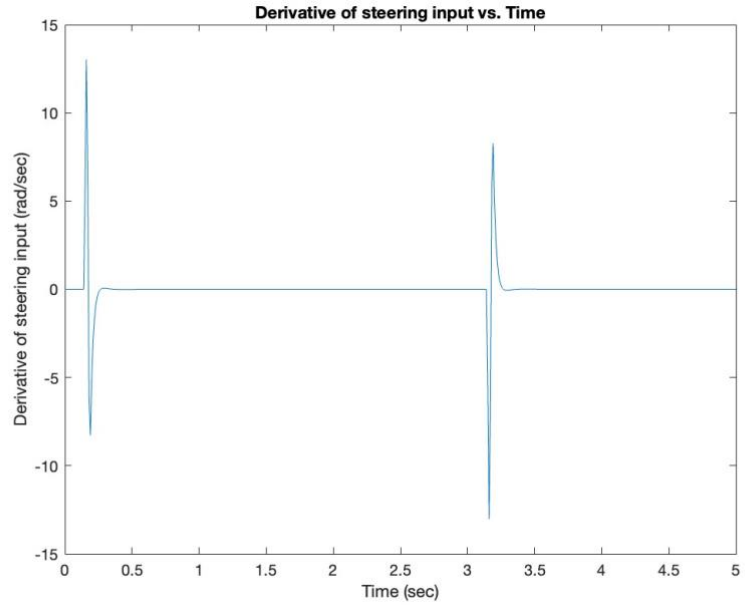
Pole placement is used to solve for the state feedback matrix. Using poles at

$$p = \{-4 - 3i, -4 + 3i, -30, -40\}$$

We obtain the state feedback matrix to be

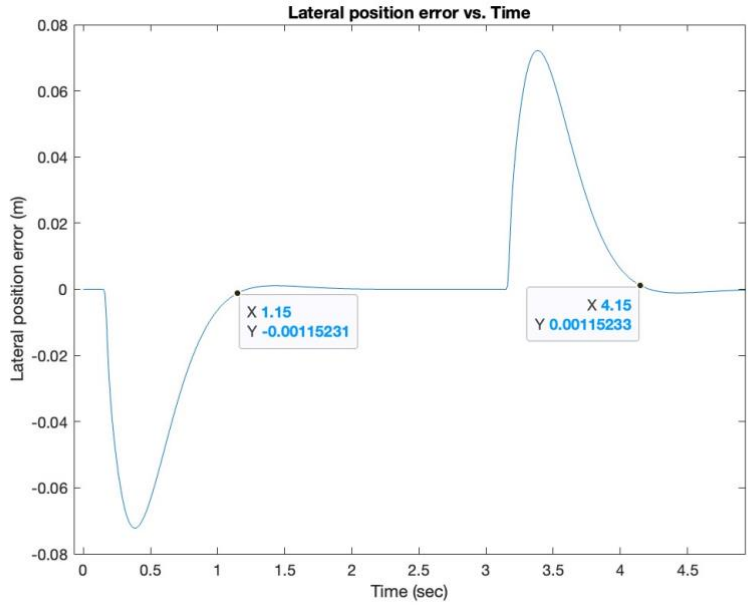
$$K = [1.9761 \quad 0.1076 \quad 16.0784 \quad 0.8716]$$

Plot of the derivative of steering input:



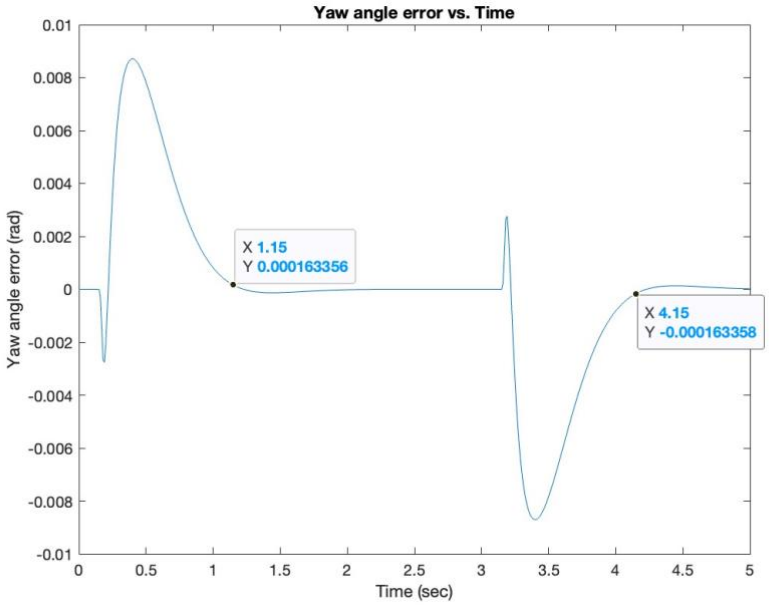
Note that the maximum derivative of the steering input does not exceed 25 rad/sec.

Plot of the resultant lateral position error (e_1):



Note that abs(e_1) falls below 0.002 m within 1 second at the transition points.

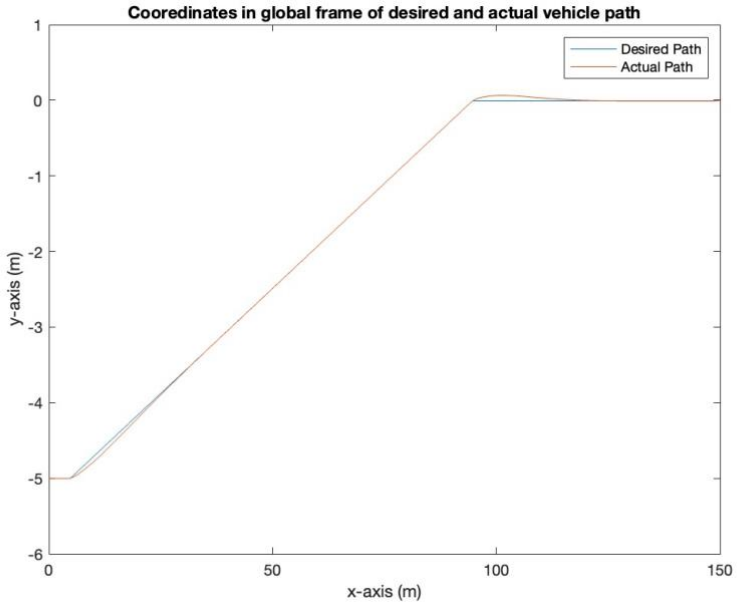
Plot of the yaw angle error (e_2):



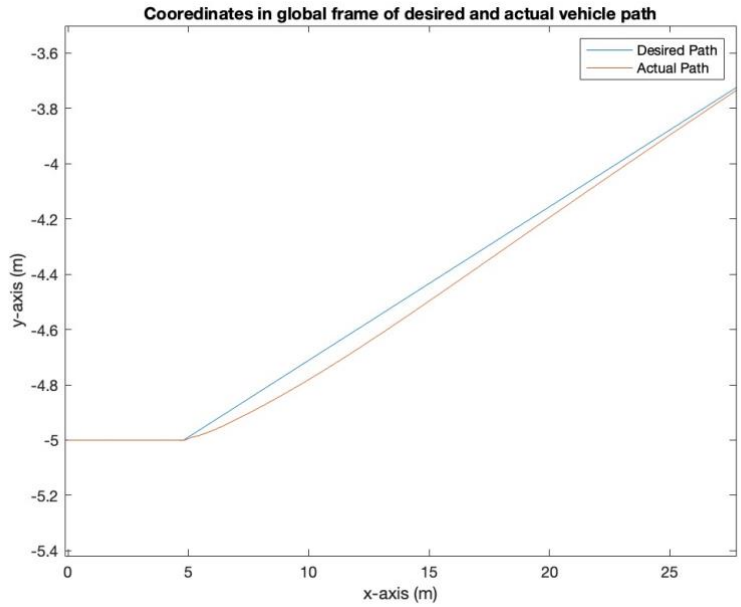
Note that abs(e_2) falls below 0.0007 rad within 1 second at the transition points. The maximum abs(e_2) obtained over the period is 0.0087 rad, below the maximum specified e_2 error of 0.01 rad.

Q2.3.

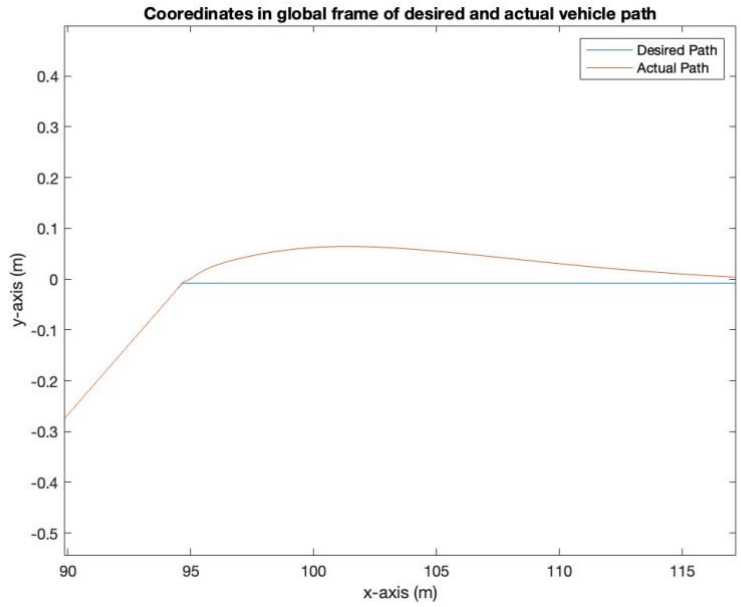
Plot of desired and actual vehicle path in global frame:



Close-up at transition point 1 (leaving initial lane):



Close-up at transition point 2 (entering new lane):



Q2.4.

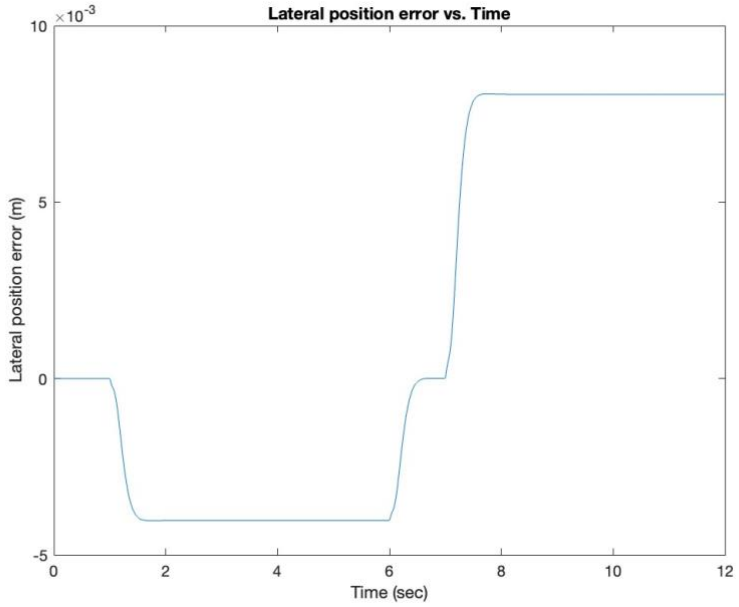
Pole placement is used to solve for the state feedback matrix. Using poles at

$$p = \{-10 - 5i, -10 + 5i, -30, -40\}$$

We obtain the state feedback matrix to be

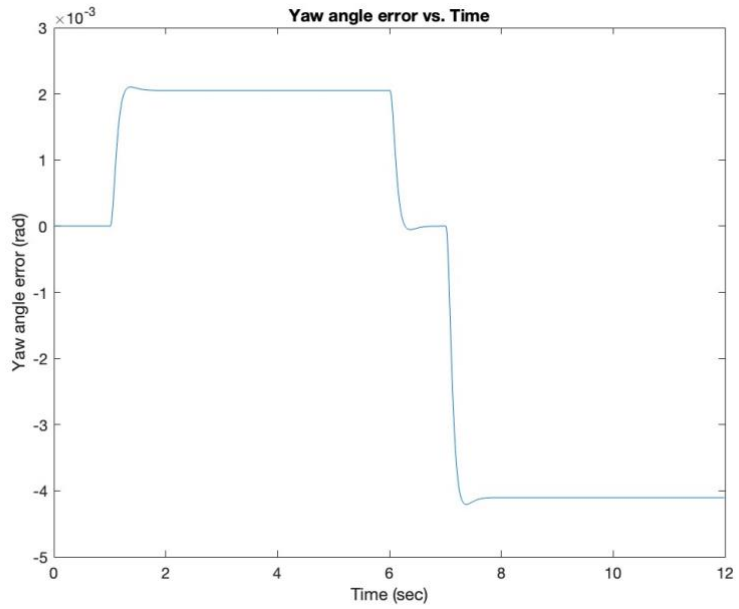
$$K = [9.8805 \quad 1.0598 \quad 17.3126 \quad -0.5135]$$

Plot of the resultant lateral position error (e_1):



The maximum abs(e_1) obtained is 0.0081 m, below the maximum specified e_1 error of 0.01 m.

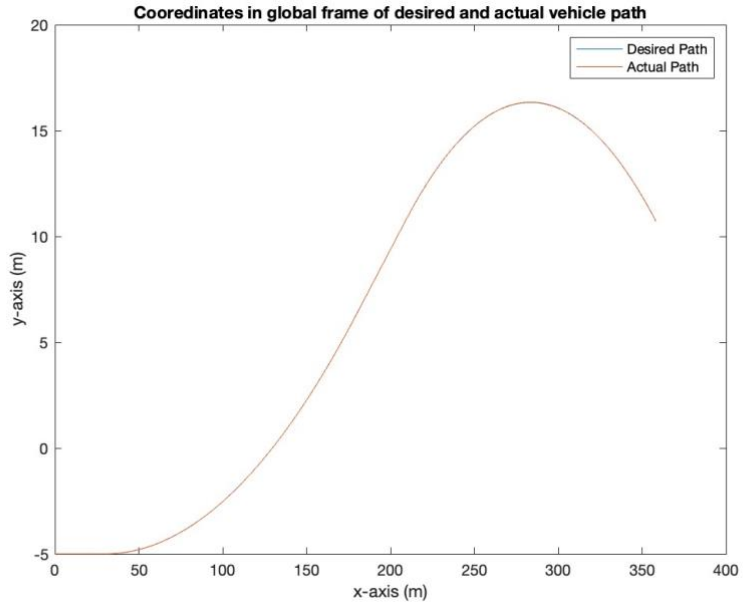
Plot of the yaw angle error (e_2):



The maximum abs(e_2) obtained is 0.0042 rad, below the maximum specified e_2 error of 0.01 rad.

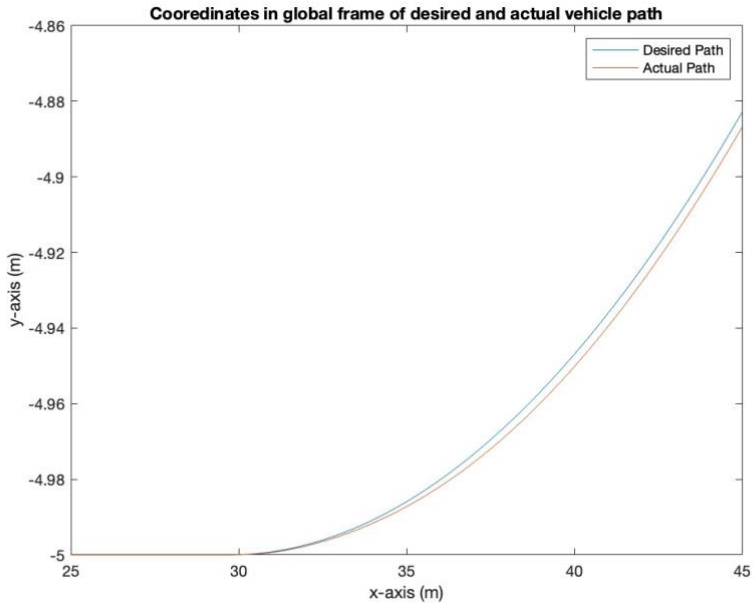
Q2.5.

Plot of desired and actual vehicle path in global frame:

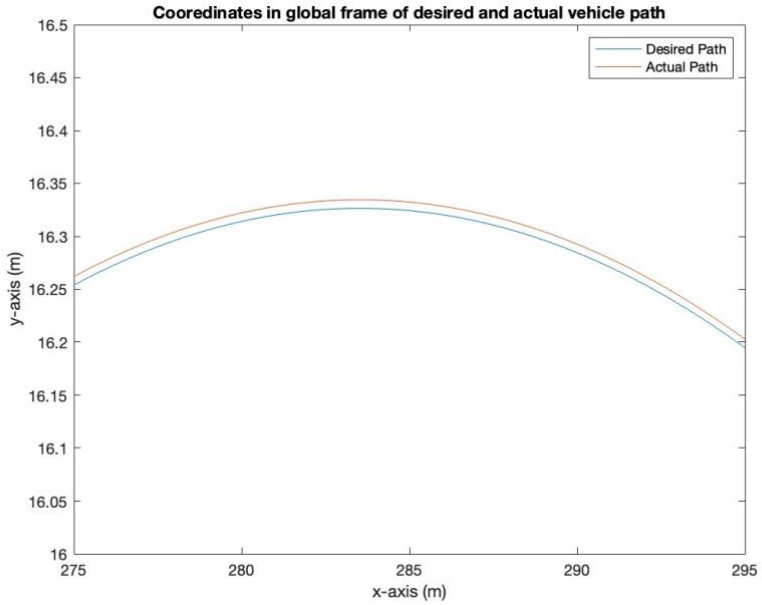


Note that the actual and desired trajectories are very close together and hard to distinguish. Some close-ups are provided below.

Close-up at transition between straight path to positive-curvature circular arc:



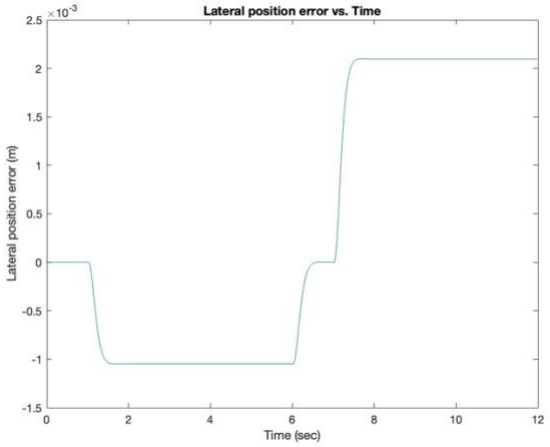
Close-up of negative-curvature circular arc:

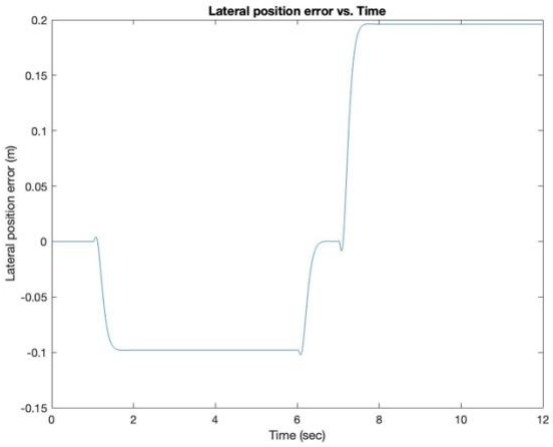


Q2.6.

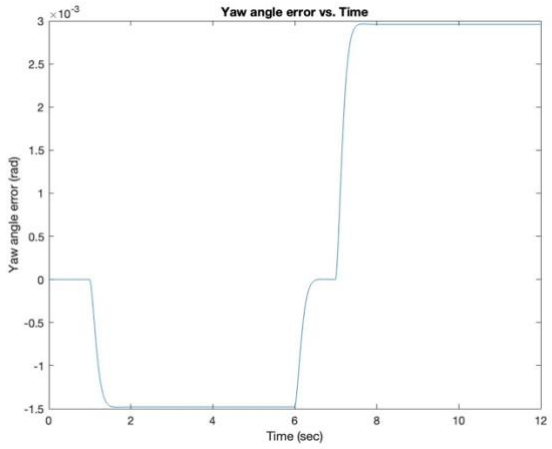
Keeping the poles chosen in Q2.4, we choose velocities to be $V_x = 5$ m/sec (plotted on the left-hand side) and $V_x = 100$ m/sec (plotted on the right-hand side)

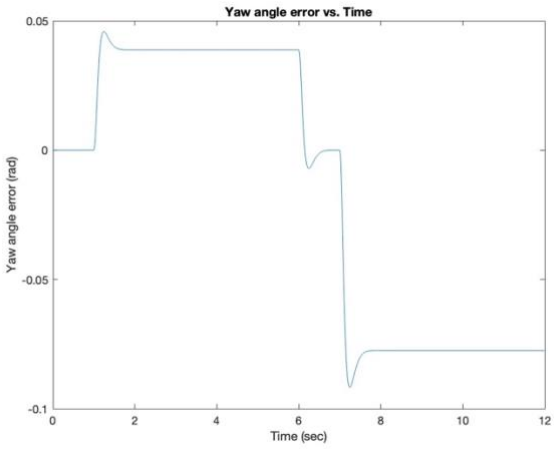
Plot of the resultant lateral position error (e_1):





Plot of the yaw angle error (e_2):





In general, as we increase velocity V_x, the errors e_1 and e_2 tends to increase as well. For each 100% increase in V_x, we expect to approximately see a 60% increase in e_1, whereas the relationship between V_x and e_2 appears to be non-linear. It is also interesting to note that while turning, the vehicle at lower velocities tend to understeer while the vehicle at higher velocities tend to oversteer.

Q3.1.

The following equations were used to implement the pure pursuit controller:

Equations used to update the state of the vehicle:

$$\dot{X} = V\cos\psi$$

$$\dot{Y} = V\sin\psi$$

$$\dot{\psi} = \frac{V}{WB}\tan\delta_f$$

Where $WB = 2.5$ m is the wheelbase length, and the other parameters are defined and derived in the Pepy model in Q1.1. Note that for the implementation, a discrete timestep of $dt = 0.1$ sec was used. Hence, we used the updated V and ψ values at the end of each timestep to calculate the above parameters. The above values are then numerically integrated over dt and added to their respective initial states to give the ending state of the vehicle.

Equations used to calculate yaw error (ψ_{error}):

$$\psi_{desired} = \tan^{-1}\left(\frac{y_{target} - y_{ego}}{x_{target} - x_{ego}}\right)$$

$$\psi_{error} = \psi_{desired} - \psi_{ego}$$

Where the target coordinates and ego (vehicle) coordinates can be calculated by calling functions and methods provided by the starter code.

Equations used to calculate δ_f:

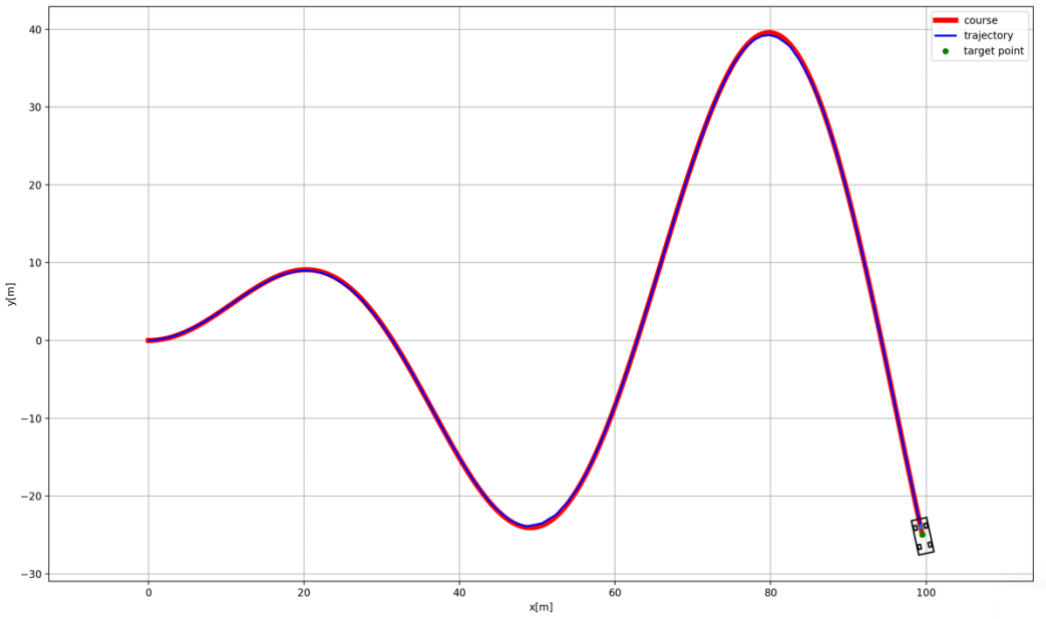
$$\delta_f = \tan^{-1}\left(\frac{2 \times WB \times \sin(\psi_{error})}{L}\right)$$

Where L is the look ahead distance and $WB = 2.5$ m is the wheelbase length. The derivation for the above equation can be found in (Theers & Singh, n.d.)[1].

[1] Theers, M., & Singh, M. (n.d.). *Pure pursuit*. Algorithms for Automated Driving. https://thomasfermi.github.io/Algorithms-for-Automated-Driving/Control/PurePursuit.html

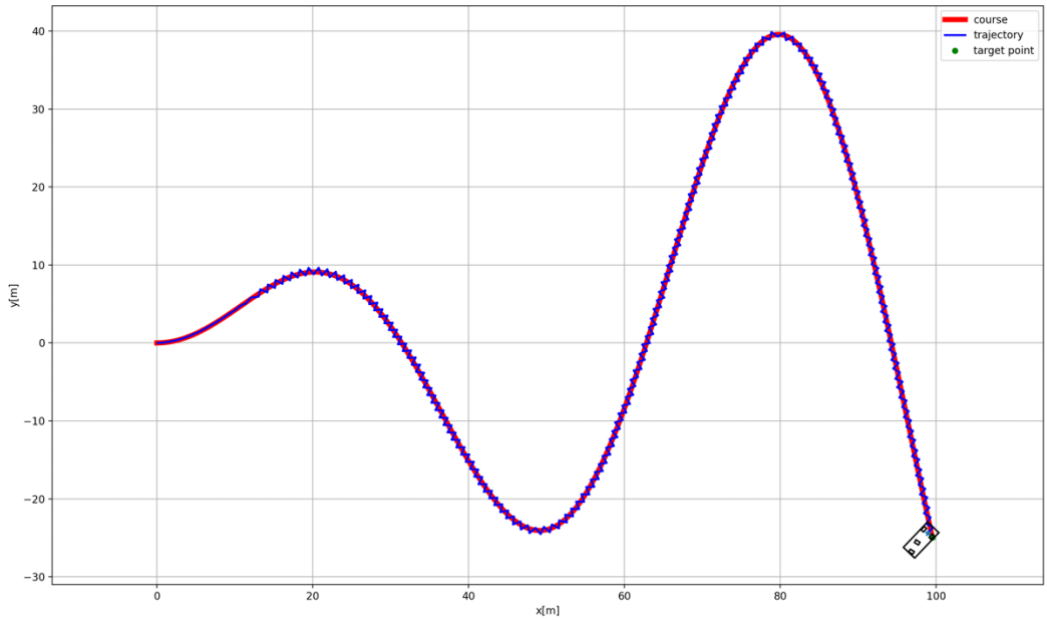
Q3.2.

Using a look ahead distance of $L = 1.5$ m, we obtain the following vehicle trajectory:

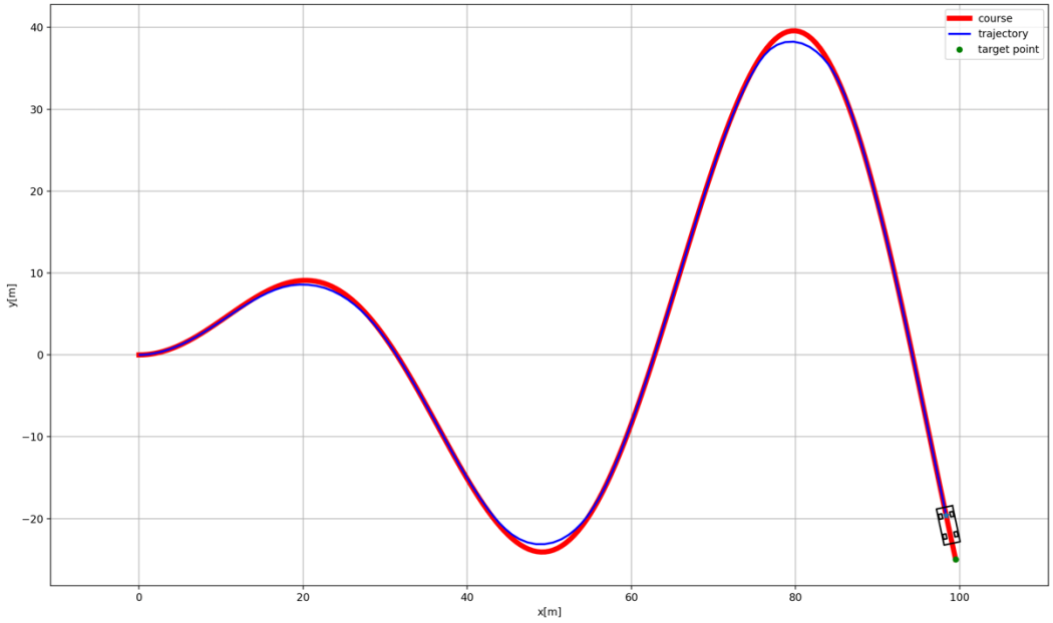


Q3.3.

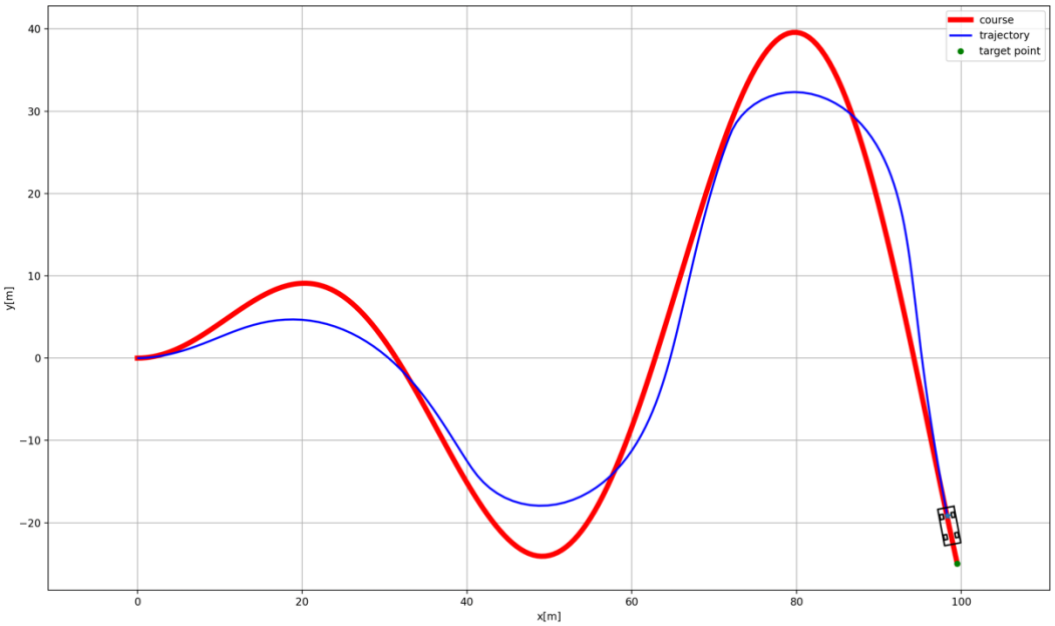
Using $L = 1$ m, we obtain the following vehicle trajectory:



Using $L = 5$ m, we obtain the following vehicle trajectory:



Using $L = 15$ m, we obtain the following vehicle trajectory:



For excessively small look ahead distances (i.e. $L = 1$ m), the vehicle follows the reference trajectory closely but has an undesired zigzagging behavior due to overshooting and undershooting the trajectory at each time step. The time taken to reach the end is also slower due to the longer distance travelled from zigzagging. As we increase to $L = 5$ m, the zigzagging disappears, and the trajectory is smoother. The vehicle also reaches the end faster, but the vehicle does not follow the

path as closely. Finally, increasing to $L = 15$ m causes the vehicle to significantly deviate from the reference trajectory, especially at corners. This is caused by the larger look ahead distance drawing large radii tangent arc trajectories that do not well approximate the reference path. In summary, an excessively small or excessively large look ahead distance is both undesirable, and a look ahead distance between 1.5 m to 5 m appears to best follow the reference trajectory without any undesirable side effects.

Appendices:

A.1. Code for Q1.4

```
% Use Runge-Kutta method to solve the ODE
clear;

% Parameters
V = 1;
lf = 1.5;
lr = 1.5;

% Time span
tspan = [0, 10];

% Initial conditions
y0 = [0, 0, 0];

% Run ode45
[t, y] = ode45(@(t, y) odefun(t, y, V, lf, lr), tspan, y0);

% Plot figure
figure;

plot(y(:,1), y(:,2));
title("Cooredinates of vehicle subject to deltaf steering angle")
xlabel("x-axis");
ylabel("y-axis");

% Create function for ODE
% y(t) = [X(t), Y(t), psi(t)]

function dydt = odefun(t, y, V, lf, lr)

    % Front steering angle in radians
    % Part A: Constant
    % deltaf = 1;

    % Part B: Sinusoid normalized with period of 2 seconds
    % deltaf = sin(pi * t);

    % Part C: Square wave normalized with period of 2 seconds
    % deltaf = square(pi * t);

    dydt = zeros(3, 1);
    dydt(1) = V * cos(y(3));
    dydt(2) = V * sin(y(3));
    dydt(3) = (V * tan(deltaf) ) / (lf + lr);
end
```

A.2. Code for Q2.2 and Q2.3

```
% DBM with lane change
clear;

% Parameters
Vx = 30;
m = 1573;
Iz = 2873;
lf = 1.1;
lr = 1.58;
Calphaf = 80000;
Calphar = 80000;

% Populate linear state space system of the form xdot = A * x +
% B1 * deltaf + B2 * psidot_des and y = C * x + D * deltaf
% Populate A matrix
A = zeros(4,4);
```

```
A(1,2) = 1;
A(2,2) = -(2 * Calphaf + 2 * Calphar)/(m * Vx);
A(2,3) = (2 * Calphaf + 2 * Calphar)/m;
A(2,4) = (-2 * Calphaf * lf + 2 * Calphar * lr)/(m * Vx);
A(3,4) = 1;
A(4,2) = -(2 * Calphaf * lf - 2 * Calphar * lr)/(Iz * Vx);
A(4,3) = (2 * Calphaf * lf - 2 * Calphar * lr)/Iz;
A(4,4) = -(2 * Calphaf * lf * lf + 2 * Calphar * lr * lr)/(Iz * Vx);

% Populate B1 matrix
B1 = zeros(4,1);
B1(2,1) = (2 * Calphaf)/m;
B1(4,1) = (2 * Calphaf * lf)/Iz;

% Populate B2 matrix
B2 = zeros(4,1);
B2(2,1) = -(2 * Calphaf * lf - 2 * Calphar * lr)/(m * Vx) - Vx;
B2(4,1) = -(2 * Calphaf * lf * lf + 2 * Calphar * lr * lr)/(Iz * Vx);

% Populate C matrix
C = eye(4);

% Populate D matrix
D = zeros(4,1);

% Create state space object for open-loop system
open_loop_sys = ss(A, B1, C, D);

% Check open-loop system eigenvalues
Eopen = eig(A);

% Desired eigenvalues
p = [complex(-4,-3);
    complex(-4,3);
    -30;
    -40];

% Solve for K using pole placement
K = place(A,B1,p);

% Check for closed-loop system eigenvalues
Aclosed = A - B1 * K;
Eclosed = eig(Aclosed);

% Create state space object for closed-loop system
closed_loop_sys = ss(Aclosed, B2, C, D);

% Create inputs for psidot_des
tstep = 0.01;

u1 = zeros(1, round(5/(tstep * 30)));
u2 = atan(5/90) * ones(1, round(90/(tstep * 30)));
u3 = zeros(1, round(5/(tstep * 30)));

% Additional tradjectory to examine error and settling time
u4 = zeros(1, round(50/(tstep * 30)));

psi_des = [u1 u2 u3 u4];
psidot_des = gradient(psi_des, tstep);

% Create linear simulation object
tIn = 0:tstep:((length(psidot_des) - 1) * tstep);

[y, tOut, x] = lsim(closed_loop_sys, psidot_des, tIn);

% Errors
e1 = x(:,1);
e1dot = x(:,2);
e2 = x(:,3);
e2dot = x(:,4);

% Calculate derivative of steering input deltaf
deltaf = -K * transpose(x);
```

```
deltafdot = gradient(deltaf, tstep);

% Plot of derivative of steering input (deltafdot)
figure();
plot(tOut, deltafdot);
title("Derivative of steering input vs. Time")
xlabel("Time (sec)");
ylabel("Derivative of steering input (rad/sec)");

% Plot lateral position error e1
figure();
plot(tOut, e1);
title("Lateral position error vs. Time")
xlabel("Time (sec)");
ylabel("Lateral position error (m)");

% Plot yaw angle error e2
figure();
plot(tOut, e2);
title("Yaw angle error vs. Time")
xlabel("Time (sec)");
ylabel("Yaw angle error (rad)");

% Plot desired and actual vehicle path in global frame
num_tsteps = length(tOut);

% Desired path
X_des = zeros(1,num_tsteps);
Y_des = zeros(1,num_tsteps);
X_des(1) = 0;
Y_des(1) = -5;

% Actual path
X_act = zeros(1,num_tsteps);
Y_act = zeros(1,num_tsteps);
X_act(1) = 0;
Y_act(1) = -5;

% Equations from page 40 of Rajamani (2012)
for step = 2:num_tsteps
    X_des(step) = X_des(step-1) + Vx * tstep * cos(psi_des(step));
    Y_des(step) = Y_des(step-1) + Vx * tstep * sin(psi_des(step));
    X_act(step) = X_des(step) - e1(step) * sin(e2(step) + psi_des(step));
    Y_act(step) = Y_des(step) + e1(step) * cos(e2(step) + psi_des(step));
end

figure();
plot(X_des,Y_des);
hold on
plot(X_act, Y_act);
title("Cooredinates in global frame of desired and actual vehicle path");
legend("Desired Path","Actual Path");
xlabel("x-axis (m)");
ylabel("y-axis (m)");
hold off
```

A.3. Code for Q2.4 and Q2.5

```
% DBM with lane change
clear;

% Parameters
Vx = 30;
m = 1573;
Iz = 2873;
lf = 1.1;
lr = 1.58;
Calphaf = 80000;
Calphar = 80000;

```

```
% Populate linear state space system of the form xdot = A * x +
% B1 * deltaf + B2 * psidot_des and y = C * x + D * deltaf
% Populate A matrix
A = zeros(4,4);
A(1,2) = 1;
A(2,2) = -(2 * Calphaf + 2 * Calphar)/(m * Vx);
A(2,3) = (2 * Calphaf + 2 * Calphar)/m;
A(2,4) = (-2 * Calphaf * lf + 2 * Calphar * lr)/(m * Vx);
A(3,4) = 1;
A(4,2) = -(2 * Calphaf * lf - 2 * Calphar * lr)/(Iz * Vx);
A(4,3) = (2 * Calphaf * lf - 2 * Calphar * lr)/Iz;
A(4,4) = -(2 * Calphaf * lf * lf + 2 * Calphar * lr * lr)/(Iz * Vx);

% Populate B1 matrix
B1 = zeros(4,1);
B1(2,1) = (2 * Calphaf)/m;
B1(4,1) = (2 * Calphaf * lf)/Iz;

% Populate B2 matrix
B2 = zeros(4,1);
B2(2,1) = -(2 * Calphaf * lf - 2 * Calphar * lr)/(m * Vx) - Vx;
B2(4,1) = -(2 * Calphaf * lf * lf + 2 * Calphar * lr * lr)/(Iz * Vx);

% Populate C matrix
C = eye(4);

% Populate D matrix
D = zeros(4,1);

% Create state space object for open-loop system
open_loop_sys = ss(A, B1, C, D);

% Check open-loop system eigenvalues
Eopen = eig(A);

% Desired eigenvalues
p = [complex(-10,-5);
    complex(-10,5);
    -30;
    -40];

% Solve for K using pole placement
K = place(A,B1,p);

% Check for closed-loop system eigenvalues
Aclosed = A - B1 * K;
Eclosed = eig(Aclosed);

% Create state space object for closed-loop system
closed_loop_sys = ss(Aclosed, B2, C, D);

% Create inputs for psidot_des
tstep = 0.01;

u1dot = zeros(1, 1/tstep);
u2dot = ones(1, 5/tstep) * (Vx/1000);
u3dot = zeros(1, 1/tstep);
u4dot = ones(1, 5/tstep) * (-Vx/500);

psidot_des = [u1dot u2dot u3dot u4dot];

% Numerically integrate to get psi_des
psi_des = cumtrapz(tstep, psidot_des);

% Create linear simulation object
tIn = 0:tstep:((length(psidot_des) - 1) * tstep);

[y, tOut, x] = lsim(closed_loop_sys, psidot_des, tIn);

% Errors
e1 = x(:,1);
e1dot = x(:,2);
e2 = x(:,3);
```

```
e2dot = x(:,4);

% Calculate derivative of steering input deltaf
deltaf = -K * transpose(x);
deltafdot = gradient(deltaf, tstep);

% Plot of derivative of steering input (deltafdot)
figure();
plot(tOut, deltafdot);
title("Derivative of steering input vs. Time")
xlabel("Time (sec)");
ylabel("Derivative of steering input (rad/sec)");

% Plot lateral position error e1
figure();
plot(tOut, e1);
title("Lateral position error vs. Time")
xlabel("Time (sec)");
ylabel("Lateral position error (m)");

% Plot yaw angle error e2
figure();
plot(tOut, e2);
title("Yaw angle error vs. Time")
xlabel("Time (sec)");
ylabel("Yaw angle error (rad)");

% Plot desired and actual vehicle path in global frame
num_tsteps = length(tOut);

% Desired path
X_des = zeros(1,num_tsteps);
Y_des = zeros(1,num_tsteps);
X_des(1) = 0;
Y_des(1) = -5;

% Actual path
X_act = zeros(1,num_tsteps);
Y_act = zeros(1,num_tsteps);
X_act(1) = 0;
Y_act(1) = -5;

% Equations from page 40 of Rajamani (2012)
for step = 2:num_tsteps
    X_des(step) = X_des(step-1) + Vx * tstep * cos(psi_des(step));
    Y_des(step) = Y_des(step-1) + Vx * tstep * sin(psi_des(step));
    X_act(step) = X_des(step) - e1(step) * sin(e2(step) + psi_des(step));
    Y_act(step) = Y_des(step) + e1(step) * cos(e2(step) + psi_des(step));
end

figure();
plot(X_des,Y_des);
hold on
plot(X_act, Y_act);
title("Cooredinates in global frame of desired and actual vehicle path");
legend("Desired Path","Actual Path");
xlabel("x-axis (m)");
ylabel("y-axis (m)");
hold off
```

A.4. Code for Q3

```
"""

Path tracking simulation with pure pursuit steering control and PID speed control for 16-665 .

author: Rathin Shah(rsshah), Shruti Gangopadhyay (sgangopa)

"""

```

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

# Pure Pursuit parameters
L = 15  # 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

        yawdot = (self.vel/WB) * math.tan(delta)
        self.yaw += yawdot * dt

```

```
155.         xdot = self.vel * np.cos(self.yaw)
156.         ydot = self.vel * np.sin(self.yaw)
157.         self.x += xdot * dt
158.         self.y += ydot * dt
159.
160. class Trajectory:
161.     def __init__(self, traj_x, traj_y):
162.         """
163.         Define a trajectory class
164.         :param traj_x: list, list of x position
165.         :param traj_y: list, list of y position
166.         """
167.         self.traj_x = traj_x
168.         self.traj_y = traj_y
169.         self.last_idx = 0
170.
171.     def getPoint(self, idx):
172.         return [self.traj_x[idx], self.traj_y[idx]]
173.
174.     def getTargetPoint(self, pos):
175.         """
176.         Get the next look ahead point
177.         :param pos: list, vehicle position
178.         :return: list, target point
179.         """
180.         target_idx = self.last_idx
181.         target_point = self.getPoint(target_idx)
182.         curr_dist = getDistance(pos, target_point)
183.
184.         while curr_dist < L and target_idx < len(self.traj_x) - 1:
185.             target_idx += 1
186.             target_point = self.getPoint(target_idx)
187.             curr_dist = getDistance(pos, target_point)
188.
189.         self.last_idx = target_idx
190.         return self.getPoint(target_idx)
191.
192. class Controller:
193.     def __init__(self, kp=1.0, ki=0.1):
194.         """
195.         Define a PID controller class
196.         :param kp: float, kp coeff
197.         :param ki: float, ki coeff
198.         :param kd: float, kd coeff
199.         """
200.         self.kp = kp
201.         self.ki = ki
202.         self.Pterm = 0.0
203.         self.Iterm = 0.0
204.         self.last_error = 0.0
205.
206.     def Longitudinalcontrol(self, error):
207.         """
208.         PID main function, given an input, this function will output a acceleration for
longitudinal error
209.         :param error: float, error term
210.         :return: float, output control
211.         """
212.         self.Pterm = self.kp * error
213.         self.Iterm += error * dt
214.
215.         self.last_error = error
216.         output = self.Pterm + self.ki * self.Iterm
217.         return output
218.
219.     def PurePursuitcontrol(self, error):
220.         #TODO- find delta
221.         delta = np.arctan((2 * np.sin(error) * WB)/L)
222.
223.         return delta
224.
225. def main():
226.     # create vehicle
```

```
227.     ego = Vehicle(0, 0, 0)
228.     plotVehicle(ego.x, ego.y, ego.yaw)
229.
230.     # target velocity
231.     target_vel = 10
232.
233.     # target course
234.     traj_x = np.arange(0, 100, 0.5)
235.     traj_y = [math.sin(x / 10.0) * x / 2.0 for x in traj_x]
236.     traj = Trajectory(traj_x, traj_y)
237.     goal = traj.getPoint(len(traj_x) - 1)
238.
239.     # create longitudinal and pure pursuit controller
240.     PI_acc = Controller()
241.     PI_yaw = Controller()
242.
243.     # real trajectory
244.     traj_ego_x = []
245.     traj_ego_y = []
246.
247.     plt.figure(figsize=(12, 8))
248.
249.     while getDistance([ego.x, ego.y], goal) > 1:
250.         target_point = traj.getTargetPoint([ego.x, ego.y])
251.
252.         # use PID to control the speed vehicle
253.         vel_err = target_vel - ego.vel
254.         acc = PI_acc.Longitudinalcontrol(vel_err)
255.
256.         # use pure pursuit to control the heading of the vehicle
257.         # TODO- Calculate the yaw error
258.         x_diff = target_point[0] - ego.x
259.         y_diff = target_point[1] - ego.y
260.         yaw_des = np.arctan2(y_diff, x_diff)
261.
262.         yaw_act = ego.yaw
263.
264.         yaw_err = yaw_des - yaw_act #TODO- Update the equation
265.
266.         delta = PI_yaw.PurePursuitcontrol(yaw_err)  #TODO- update the Pure pursuit controller
267.
268.         # move the vehicle
269.         ego.update(acc, delta)
270.
271.         # store the trajectory
272.         traj_ego_x.append(ego.x)
273.         traj_ego_y.append(ego.y)
274.
275.         # # plots
276.         plt.cla()
277.         plt.plot(traj_x, traj_y, "-r", linewidth=5, label="course")
278.         plt.plot(traj_ego_x, traj_ego_y, "-b", linewidth=2, label="trajectory")
279.         plt.plot(target_point[0], target_point[1], "og", ms=5, label="target point")
280.         plotVehicle(ego.x, ego.y, ego.yaw, delta)
281.         plt.xlabel("x[m]")
282.         plt.ylabel("y[m]")
283.         plt.axis("equal")
284.         plt.legend()
285.         plt.grid(True)
286.         plt.pause(0.1)
287.
288. if __name__ == "__main__":
289.     main()
```