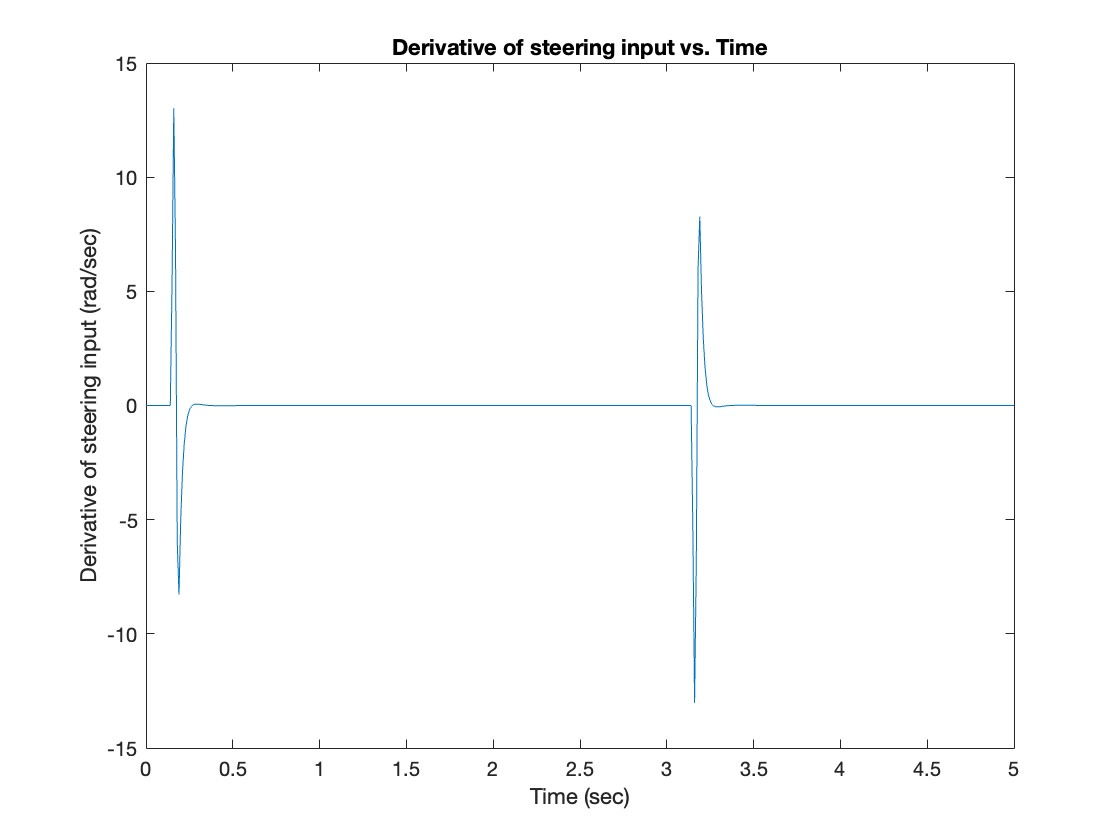
**Q2.2.**

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

We obtain the state feedback matrix to be

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 ():

A graph showing a normal position

Description automatically generated with medium confidence

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

Plot of the yaw angle error ():

A graph showing a number of numbers and a line

Description automatically generated with medium confidence

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

**Q2.3.**

Plot of desired and actual vehicle path in global frame:

A graph of a path

Description automatically generated

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

A graph of a path

Description automatically generated

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

A graph of a path

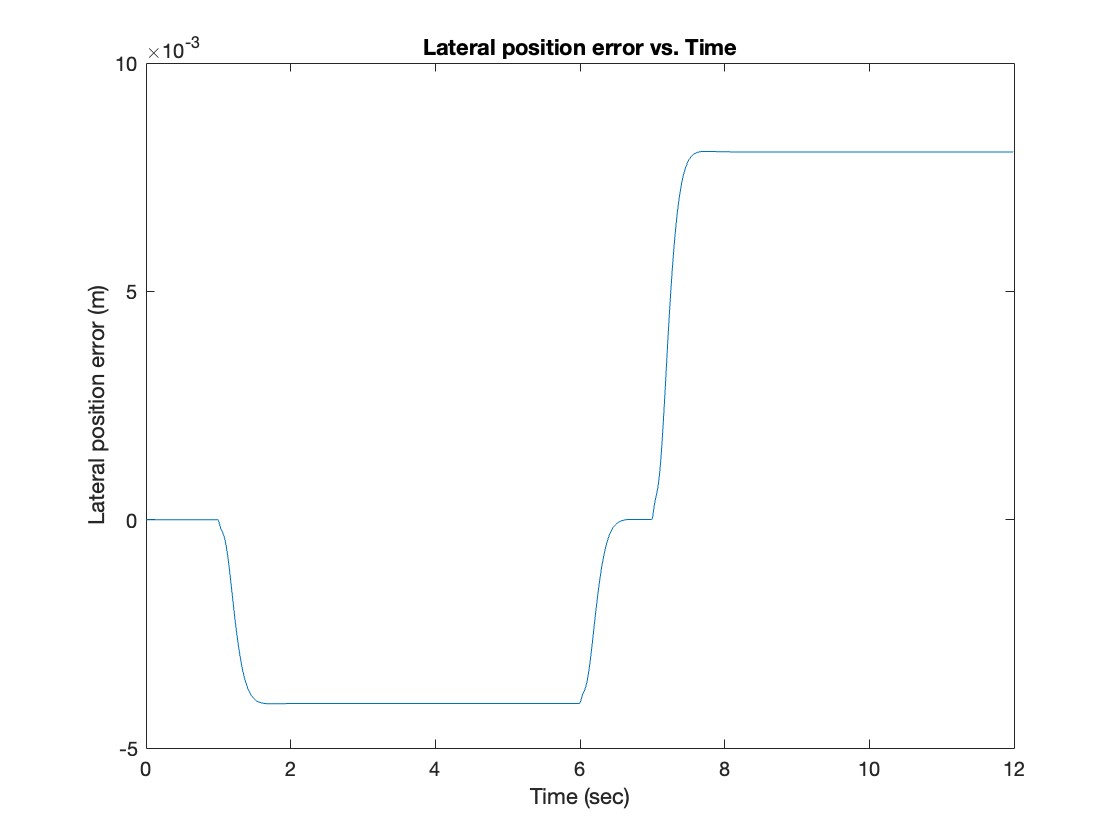
Description automatically generated

**Q2.4.**

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

We obtain the state feedback matrix to be

Plot of the resultant lateral position error ():



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

Plot of the yaw angle error ():

A graph with a line

Description automatically generated

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

**Q2.5.**

Plot of desired and actual vehicle path in global frame:

A graph of a path

Description automatically generated

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:

A graph of a path

Description automatically generated

Close-up of negative-curvature circular arc:

A graph of a curve

Description automatically generated

**Q2.6.**

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

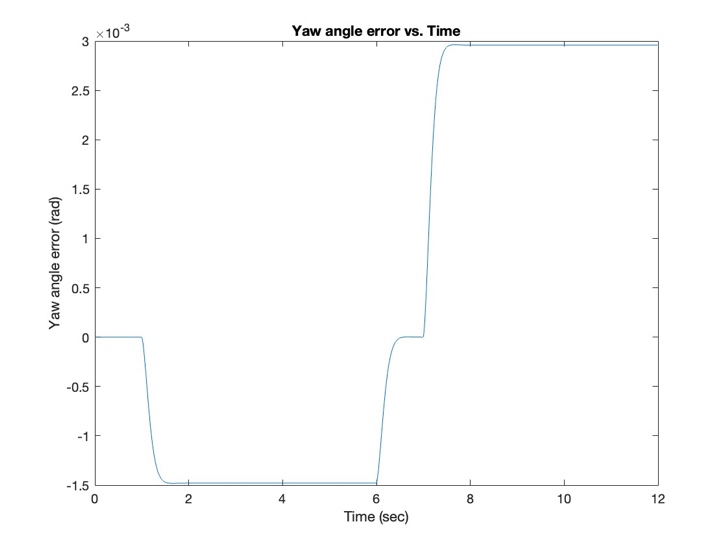
Plot of the resultant lateral position error ():

A graph of a function

Description automatically generatedA graph of a function

Description automatically generated

Plot of the yaw angle error ():

A graph of a graph

Description automatically generated

In general, as we increase velocity , the errors and tends to increase as well. For each 100% increase in , we expect to approximately see a 60% increase in , whereas the relationship between and 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:

Where 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 sec was used. Hence, we used the updated and values at the end of each timestep to calculate the above parameters. The above values are then numerically integrated over and added to their respective initial states to give the ending state of the vehicle.

Equations used to calculate yaw error ():

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 :

Where is the look ahead distance and m is the wheelbase length. The derivation for the above equation can be found in (Theers & Singh, n.d.)[[1]](#footnote-1).

**Q3.2.**

Using a look ahead distance of m, we obtain the following vehicle trajectory:

A graph with a red line

Description automatically generated

**Q3.3.**

Using m, we obtain the following vehicle trajectory:

A graph with a line graph

Description automatically generated

Using m, we obtain the following vehicle trajectory:

A graph with red and blue lines

Description automatically generated

Using m, we obtain the following vehicle trajectory:

A graph with red and blue lines

Description automatically generated

For excessively small look ahead distances (i.e. 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 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 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**

1. % Use Runge-Kutta method to solve the ODE

2. clear;

3.

4. % Parameters

5. V = 1;

6. lf = 1.5;

7. lr = 1.5;

8.

9. % Time span

10. tspan = [0, 10];

11.

12. % Initial conditions

13. y0 = [0, 0, 0];

14.

15. % Run ode45

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

17.

18. % Plot figure

19. figure;

20.

21. plot(y(:,1), y(:,2));

22. title("Cooredinates of vehicle subject to deltaf steering angle")

23. xlabel("x-axis");

24. ylabel("y-axis");

25.

26. % Create function for ODE

27. % y(t) = [X(t), Y(t), psi(t)]

28.

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

30.

31. % Front steering angle in radians

32. % Part A: Constant

33. % deltaf = 1;

34.

35. % Part B: Sinusoid normalized with period of 2 seconds

36. % deltaf = sin(pi \* t);

37.

38. % Part C: Square wave normalized with period of 2 seconds

39. % deltaf = square(pi \* t);

40.

41. dydt = zeros(3, 1);

42. dydt(1) = V \* cos(y(3));

43. dydt(2) = V \* sin(y(3));

44. dydt(3) = (V \* tan(deltaf) ) / (lf + lr);

45. end

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

1. % DBM with lane change

2. clear;

3.

4. % Parameters

5. Vx = 30;

6. m = 1573;

7. Iz = 2873;

8. lf = 1.1;

9. lr = 1.58;

10. Calphaf = 80000;

11. Calphar = 80000;

12.

13. % Populate linear state space system of the form xdot = A \* x +

14. % B1 \* deltaf + B2 \* psidot\_des and y = C \* x + D \* deltaf

15. % Populate A matrix

16. A = zeros(4,4);

17. A(1,2) = 1;

18. A(2,2) = -(2 \* Calphaf + 2 \* Calphar)/(m \* Vx);

19. A(2,3) = (2 \* Calphaf + 2 \* Calphar)/m;

20. A(2,4) = (-2 \* Calphaf \* lf + 2 \* Calphar \* lr)/(m \* Vx);

21. A(3,4) = 1;

22. A(4,2) = -(2 \* Calphaf \* lf - 2 \* Calphar \* lr)/(Iz \* Vx);

23. A(4,3) = (2 \* Calphaf \* lf - 2 \* Calphar \* lr)/Iz;

24. A(4,4) = -(2 \* Calphaf \* lf \* lf + 2 \* Calphar \* lr \* lr)/(Iz \* Vx);

25.

26. % Populate B1 matrix

27. B1 = zeros(4,1);

28. B1(2,1) = (2 \* Calphaf)/m;

29. B1(4,1) = (2 \* Calphaf \* lf)/Iz;

30.

31. % Populate B2 matrix

32. B2 = zeros(4,1);

33. B2(2,1) = -(2 \* Calphaf \* lf - 2 \* Calphar \* lr)/(m \* Vx) - Vx;

34. B2(4,1) = -(2 \* Calphaf \* lf \* lf + 2 \* Calphar \* lr \* lr)/(Iz \* Vx);

35.

36. % Populate C matrix

37. C = eye(4);

38.

39. % Populate D matrix

40. D = zeros(4,1);

41.

42. % Create state space object for open-loop system

43. open\_loop\_sys = ss(A, B1, C, D);

44.

45. % Check open-loop system eigenvalues

46. Eopen = eig(A);

47.

48. % Desired eigenvalues

49. p = [complex(-4,-3);

50. complex(-4,3);

51. -30;

52. -40];

53.

54. % Solve for K using pole placement

55. K = place(A,B1,p);

56.

57. % Check for closed-loop system eigenvalues

58. Aclosed = A - B1 \* K;

59. Eclosed = eig(Aclosed);

60.

61. % Create state space object for closed-loop system

62. closed\_loop\_sys = ss(Aclosed, B2, C, D);

63.

64. % Create inputs for psidot\_des

65. tstep = 0.01;

66.

67. u1 = zeros(1, round(5/(tstep \* 30)));

68. u2 = atan(5/90) \* ones(1, round(90/(tstep \* 30)));

69. u3 = zeros(1, round(5/(tstep \* 30)));

70.

71. % Additional tradjectory to examine error and settling time

72. u4 = zeros(1, round(50/(tstep \* 30)));

73.

74. psi\_des = [u1 u2 u3 u4];

75. psidot\_des = gradient(psi\_des, tstep);

76.

77. % Create linear simulation object

78. tIn = 0:tstep:((length(psidot\_des) - 1) \* tstep);

79.

80. [y, tOut, x] = lsim(closed\_loop\_sys, psidot\_des, tIn);

81.

82. % Errors

83. e1 = x(:,1);

84. e1dot = x(:,2);

85. e2 = x(:,3);

86. e2dot = x(:,4);

87.

88. % Calculate derivative of steering input deltaf

89. deltaf = -K \* transpose(x);

90. deltafdot = gradient(deltaf, tstep);

91.

92. % Plot of derivative of steering input (deltafdot)

93. figure();

94. plot(tOut, deltafdot);

95. title("Derivative of steering input vs. Time")

96. xlabel("Time (sec)");

97. ylabel("Derivative of steering input (rad/sec)");

98.

99. % Plot lateral position error e1

100. figure();

101. plot(tOut, e1);

102. title("Lateral position error vs. Time")

103. xlabel("Time (sec)");

104. ylabel("Lateral position error (m)");

105.

106. % Plot yaw angle error e2

107. figure();

108. plot(tOut, e2);

109. title("Yaw angle error vs. Time")

110. xlabel("Time (sec)");

111. ylabel("Yaw angle error (rad)");

112.

113. % Plot desired and actual vehicle path in global frame

114. num\_tsteps = length(tOut);

115.

116. % Desired path

117. X\_des = zeros(1,num\_tsteps);

118. Y\_des = zeros(1,num\_tsteps);

119. X\_des(1) = 0;

120. Y\_des(1) = -5;

121.

122. % Actual path

123. X\_act = zeros(1,num\_tsteps);

124. Y\_act = zeros(1,num\_tsteps);

125. X\_act(1) = 0;

126. Y\_act(1) = -5;

127.

128. % Equations from page 40 of Rajamani (2012)

129. for step = 2:num\_tsteps

130. X\_des(step) = X\_des(step-1) + Vx \* tstep \* cos(psi\_des(step));

131. Y\_des(step) = Y\_des(step-1) + Vx \* tstep \* sin(psi\_des(step));

132. X\_act(step) = X\_des(step) - e1(step) \* sin(e2(step) + psi\_des(step));

133. Y\_act(step) = Y\_des(step) + e1(step) \* cos(e2(step) + psi\_des(step));

134. end

135.

136. figure();

137. plot(X\_des,Y\_des);

138. hold on

139. plot(X\_act, Y\_act);

140. title("Cooredinates in global frame of desired and actual vehicle path");

141. legend("Desired Path","Actual Path");

142. xlabel("x-axis (m)");

143. ylabel("y-axis (m)");

144. hold off

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

1. % DBM with lane change

2. clear;

3.

4. % Parameters

5. Vx = 30;

6. m = 1573;

7. Iz = 2873;

8. lf = 1.1;

9. lr = 1.58;

10. Calphaf = 80000;

11. Calphar = 80000;

12.

13. % Populate linear state space system of the form xdot = A \* x +

14. % B1 \* deltaf + B2 \* psidot\_des and y = C \* x + D \* deltaf

15. % Populate A matrix

16. A = zeros(4,4);

17. A(1,2) = 1;

18. A(2,2) = -(2 \* Calphaf + 2 \* Calphar)/(m \* Vx);

19. A(2,3) = (2 \* Calphaf + 2 \* Calphar)/m;

20. A(2,4) = (-2 \* Calphaf \* lf + 2 \* Calphar \* lr)/(m \* Vx);

21. A(3,4) = 1;

22. A(4,2) = -(2 \* Calphaf \* lf - 2 \* Calphar \* lr)/(Iz \* Vx);

23. A(4,3) = (2 \* Calphaf \* lf - 2 \* Calphar \* lr)/Iz;

24. A(4,4) = -(2 \* Calphaf \* lf \* lf + 2 \* Calphar \* lr \* lr)/(Iz \* Vx);

25.

26. % Populate B1 matrix

27. B1 = zeros(4,1);

28. B1(2,1) = (2 \* Calphaf)/m;

29. B1(4,1) = (2 \* Calphaf \* lf)/Iz;

30.

31. % Populate B2 matrix

32. B2 = zeros(4,1);

33. B2(2,1) = -(2 \* Calphaf \* lf - 2 \* Calphar \* lr)/(m \* Vx) - Vx;

34. B2(4,1) = -(2 \* Calphaf \* lf \* lf + 2 \* Calphar \* lr \* lr)/(Iz \* Vx);

35.

36. % Populate C matrix

37. C = eye(4);

38.

39. % Populate D matrix

40. D = zeros(4,1);

41.

42. % Create state space object for open-loop system

43. open\_loop\_sys = ss(A, B1, C, D);

44.

45. % Check open-loop system eigenvalues

46. Eopen = eig(A);

47.

48. % Desired eigenvalues

49. p = [complex(-10,-5);

50. complex(-10,5);

51. -30;

52. -40];

53.

54. % Solve for K using pole placement

55. K = place(A,B1,p);

56.

57. % Check for closed-loop system eigenvalues

58. Aclosed = A - B1 \* K;

59. Eclosed = eig(Aclosed);

60.

61. % Create state space object for closed-loop system

62. closed\_loop\_sys = ss(Aclosed, B2, C, D);

63.

64. % Create inputs for psidot\_des

65. tstep = 0.01;

66.

67. u1dot = zeros(1, 1/tstep);

68. u2dot = ones(1, 5/tstep) \* (Vx/1000);

69. u3dot = zeros(1, 1/tstep);

70. u4dot = ones(1, 5/tstep) \* (-Vx/500);

71.

72. psidot\_des = [u1dot u2dot u3dot u4dot];

73.

74. % Numerically integrate to get psi\_des

75. psi\_des = cumtrapz(tstep, psidot\_des);

76.

77. % Create linear simulation object

78. tIn = 0:tstep:((length(psidot\_des) - 1) \* tstep);

79.

80. [y, tOut, x] = lsim(closed\_loop\_sys, psidot\_des, tIn);

81.

82. % Errors

83. e1 = x(:,1);

84. e1dot = x(:,2);

85. e2 = x(:,3);

86. e2dot = x(:,4);

87.

88. % Calculate derivative of steering input deltaf

89. deltaf = -K \* transpose(x);

90. deltafdot = gradient(deltaf, tstep);

91.

92. % Plot of derivative of steering input (deltafdot)

93. figure();

94. plot(tOut, deltafdot);

95. title("Derivative of steering input vs. Time")

96. xlabel("Time (sec)");

97. ylabel("Derivative of steering input (rad/sec)");

98.

99. % Plot lateral position error e1

100. figure();

101. plot(tOut, e1);

102. title("Lateral position error vs. Time")

103. xlabel("Time (sec)");

104. ylabel("Lateral position error (m)");

105.

106. % Plot yaw angle error e2

107. figure();

108. plot(tOut, e2);

109. title("Yaw angle error vs. Time")

110. xlabel("Time (sec)");

111. ylabel("Yaw angle error (rad)");

112.

113. % Plot desired and actual vehicle path in global frame

114. num\_tsteps = length(tOut);

115.

116. % Desired path

117. X\_des = zeros(1,num\_tsteps);

118. Y\_des = zeros(1,num\_tsteps);

119. X\_des(1) = 0;

120. Y\_des(1) = -5;

121.

122. % Actual path

123. X\_act = zeros(1,num\_tsteps);

124. Y\_act = zeros(1,num\_tsteps);

125. X\_act(1) = 0;

126. Y\_act(1) = -5;

127.

128. % Equations from page 40 of Rajamani (2012)

129. for step = 2:num\_tsteps

130. X\_des(step) = X\_des(step-1) + Vx \* tstep \* cos(psi\_des(step));

131. Y\_des(step) = Y\_des(step-1) + Vx \* tstep \* sin(psi\_des(step));

132. X\_act(step) = X\_des(step) - e1(step) \* sin(e2(step) + psi\_des(step));

133. Y\_act(step) = Y\_des(step) + e1(step) \* cos(e2(step) + psi\_des(step));

134. end

135.

136. figure();

137. plot(X\_des,Y\_des);

138. hold on

139. plot(X\_act, Y\_act);

140. title("Cooredinates in global frame of desired and actual vehicle path");

141. legend("Desired Path","Actual Path");

142. xlabel("x-axis (m)");

143. ylabel("y-axis (m)");

144. hold off

**A.4. Code for Q3**

1. """

2.

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

4.

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

6.

7. """

8.

9. import math

10. import matplotlib.pyplot as plt

11. import numpy as np

12.

13. # Pure Pursuit parameters

14. L = 15 # look ahead distance

15. dt = 0.1 # discrete time

16.

17. # Vehicle parameters (m)

18. LENGTH = 4.5 #length of the vehicle (for the plot)

19. WIDTH = 2.0 #length of the vehicle (for the plot)

20. BACKTOWHEEL = 1.0 #length of the vehicle (for the plot)

21. WHEEL\_LEN = 0.3 #length of the vehicle (for the plot)

22. WHEEL\_WIDTH = 0.2 #length of the vehicle (for the plot)

23. TREAD = 0.7 #length of the vehicle (for the plot)

24. WB = 2.5 # wheel-base

25.

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

27.

28. outline = np.array(

29. [

30. [

31. -BACKTOWHEEL,

32. (LENGTH - BACKTOWHEEL),

33. (LENGTH - BACKTOWHEEL),

34. -BACKTOWHEEL,

35. -BACKTOWHEEL,

36. ],

37. [WIDTH / 2, WIDTH / 2, -WIDTH / 2, -WIDTH / 2, WIDTH / 2],

38. ]

39. )

40.

41. fr\_wheel = np.array(

42. [

43. [WHEEL\_LEN, -WHEEL\_LEN, -WHEEL\_LEN, WHEEL\_LEN, WHEEL\_LEN],

44. [

45. -WHEEL\_WIDTH - TREAD,

46. -WHEEL\_WIDTH - TREAD,

47. WHEEL\_WIDTH - TREAD,

48. WHEEL\_WIDTH - TREAD,

49. -WHEEL\_WIDTH - TREAD,

50. ],

51. ]

52. )

53.

54. rr\_wheel = np.copy(fr\_wheel)

55.

56. fl\_wheel = np.copy(fr\_wheel)

57. fl\_wheel[1, :] \*= -1

58. rl\_wheel = np.copy(rr\_wheel)

59. rl\_wheel[1, :] \*= -1

60.

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

62. Rot2 = np.array(

63. [[math.cos(steer), math.sin(steer)], [-math.sin(steer), math.cos(steer)]]

64. )

65.

66. fr\_wheel = (fr\_wheel.T.dot(Rot2)).T

67. fl\_wheel = (fl\_wheel.T.dot(Rot2)).T

68. fr\_wheel[0, :] += WB

69. fl\_wheel[0, :] += WB

70.

71. fr\_wheel = (fr\_wheel.T.dot(Rot1)).T

72. fl\_wheel = (fl\_wheel.T.dot(Rot1)).T

73.

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

75. rr\_wheel = (rr\_wheel.T.dot(Rot1)).T

76. rl\_wheel = (rl\_wheel.T.dot(Rot1)).T

77.

78. outline[0, :] += x

79. outline[1, :] += y

80. fr\_wheel[0, :] += x

81. fr\_wheel[1, :] += y

82. rr\_wheel[0, :] += x

83. rr\_wheel[1, :] += y

84. fl\_wheel[0, :] += x

85. fl\_wheel[1, :] += y

86. rl\_wheel[0, :] += x

87. rl\_wheel[1, :] += y

88.

89. plt.plot(

90. np.array(outline[0, :]).flatten(), np.array(outline[1, :]).flatten(), truckcolor

91. )

92. plt.plot(

93. np.array(fr\_wheel[0, :]).flatten(),

94. np.array(fr\_wheel[1, :]).flatten(),

95. truckcolor,

96. )

97. plt.plot(

98. np.array(rr\_wheel[0, :]).flatten(),

99. np.array(rr\_wheel[1, :]).flatten(),

100. truckcolor,

101. )

102. plt.plot(

103. np.array(fl\_wheel[0, :]).flatten(),

104. np.array(fl\_wheel[1, :]).flatten(),

105. truckcolor,

106. )

107. plt.plot(

108. np.array(rl\_wheel[0, :]).flatten(),

109. np.array(rl\_wheel[1, :]).flatten(),

110. truckcolor,

111. )

112. plt.plot(x, y, "\*")

113.

114. def getDistance(p1, p2):

115. """

116. Calculate distance

117. :param p1: list, point1

118. :param p2: list, point2

119. :return: float, distance

120. """

121. dx = p1[0] - p2[0]

122. dy = p1[1] - p2[1]

123. return math.hypot(dx, dy)

124.

125. class Vehicle:

126. def \_\_init\_\_(self, x, y, yaw, vel=0):

127. """

128. Define a vehicle class (state of the vehicle)

129. :param x: float, x position

130. :param y: float, y position

131. :param yaw: float, vehicle heading

132. :param vel: float, velocity

133.

134. """

135. # State of the vehicle

136.

137. self.x = x #x coordinate of the vehicle

138. self.y = y #y coordinate of the vehicle

139. self.yaw = yaw #yaw of the vehicle

140. self.vel = vel #velocity of the vehicle

141.

142. def update(self, acc, delta):

143. """

144. Vehicle motion model, here we are using simple bycicle model

145. :param acc: float, acceleration

146. :param delta: float, heading control

147. """

148.

149. # TODO- update the state of the vehicle (x,y,yaw,vel) based on simple bicycle model

150. self.vel += acc \* dt

151.

152. yawdot = (self.vel/WB) \* math.tan(delta)

153. self.yaw += yawdot \* dt

154.

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()

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