Project: Part 3

24-677 Special Topics: Modern Control - Theory and Design

Ryan Wu (ID: weihuanw)

Due: Nov 16, 2023, 11:59 pm.

P3: Problems

Exercise 1

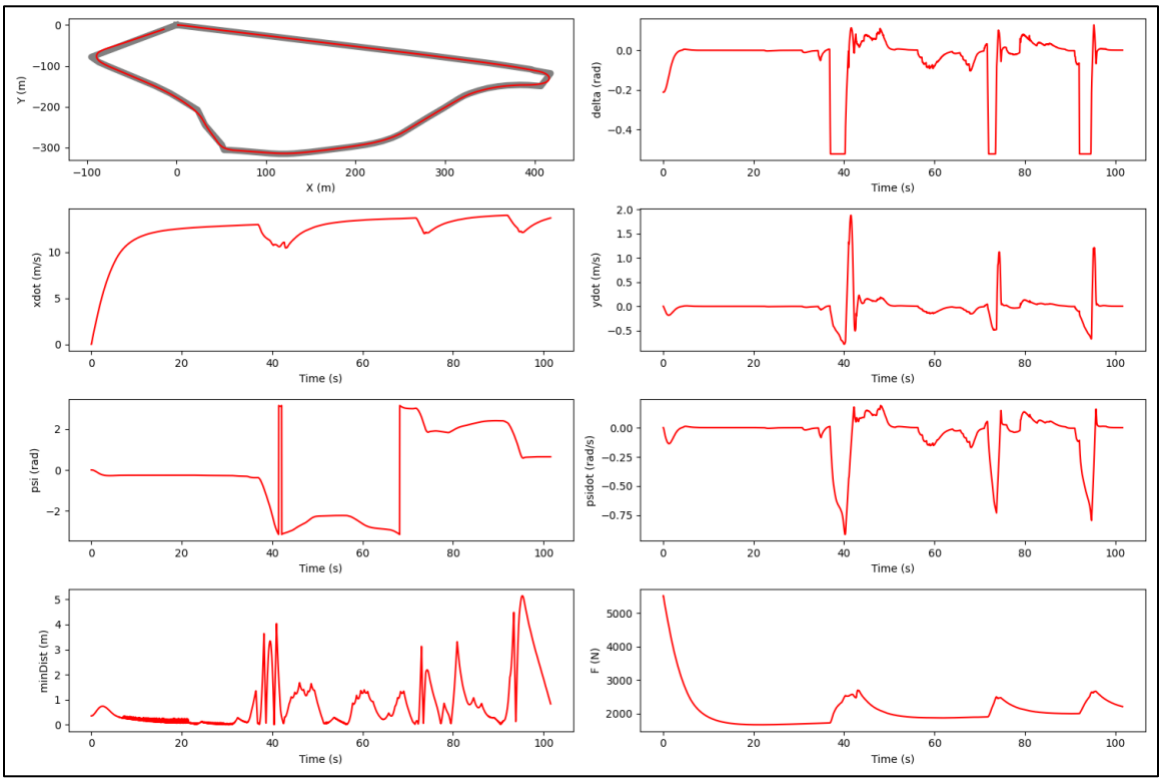

Figure 1. The final completion plot using my LQR controller.

```
Evaluating...
Score for completing the loop: 30.0/30.0
Score for average distance: 30.0/30.0
Score for maximum distance: 30.0/30.0
Your time is  101.536
Your total score is : 100.0/100.0
total steps:  101536
maxMinDist:  5.140777021230266
avgMinDist:  0.8281291182830609
INFO: 'main' controller exited successfully.
```

Figure 2. The final score message from Webots simulation using my LQR controller.

Exercise 2

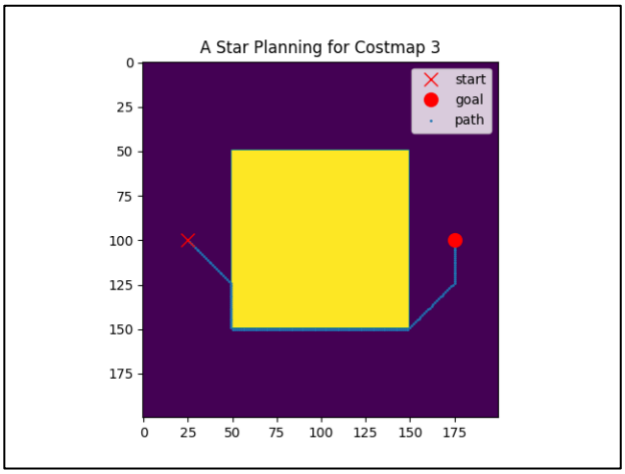

Figure 3. The path planning results from my Astar_script.py.

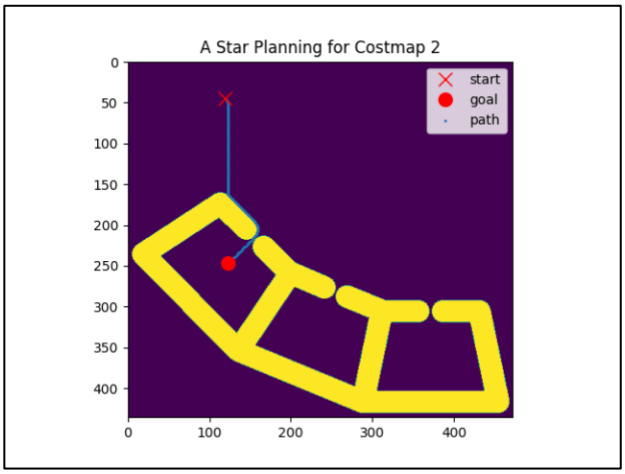

Figure 4. The path planning results from my Astar_script.py.

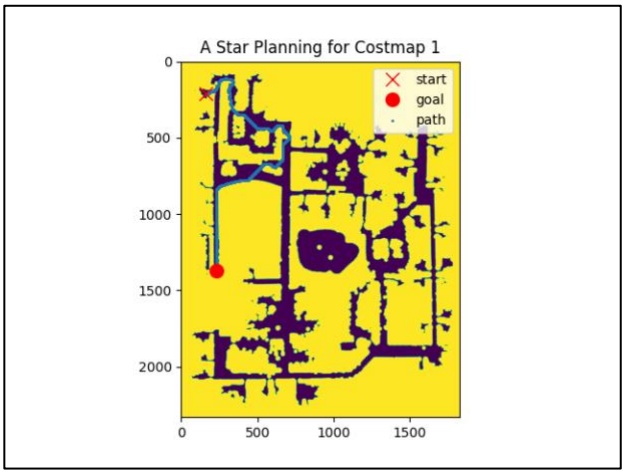

Figure 5. The path planning results from my Astar_script.py.

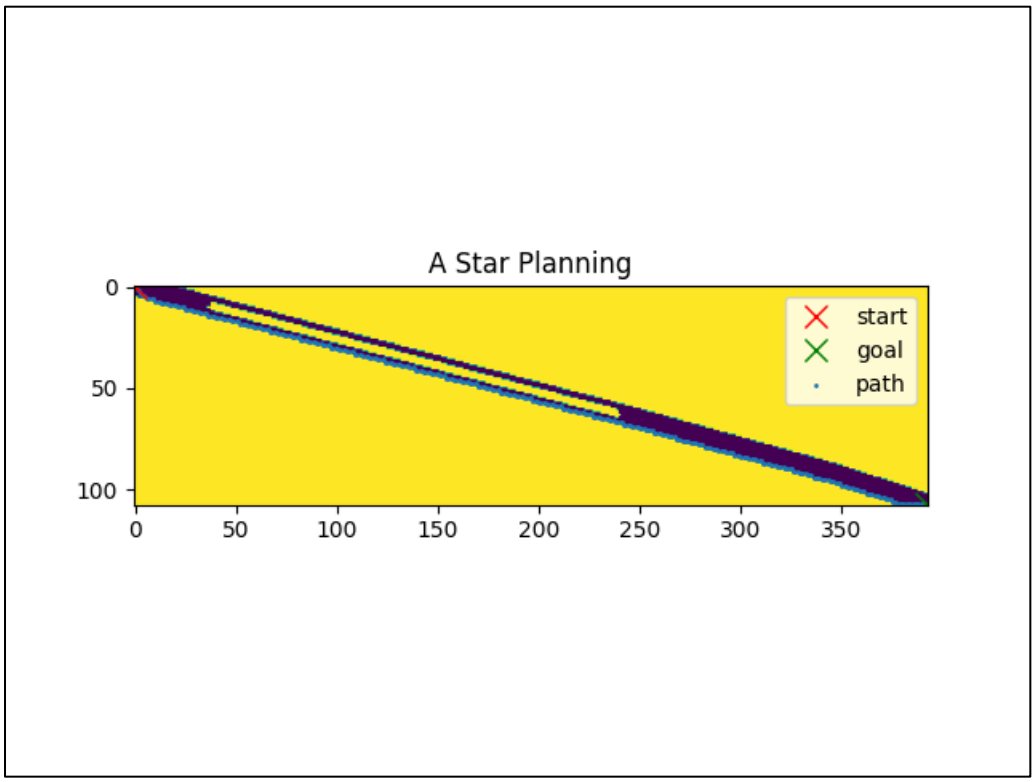

Figure 6. The path planning results from my Astar.py.

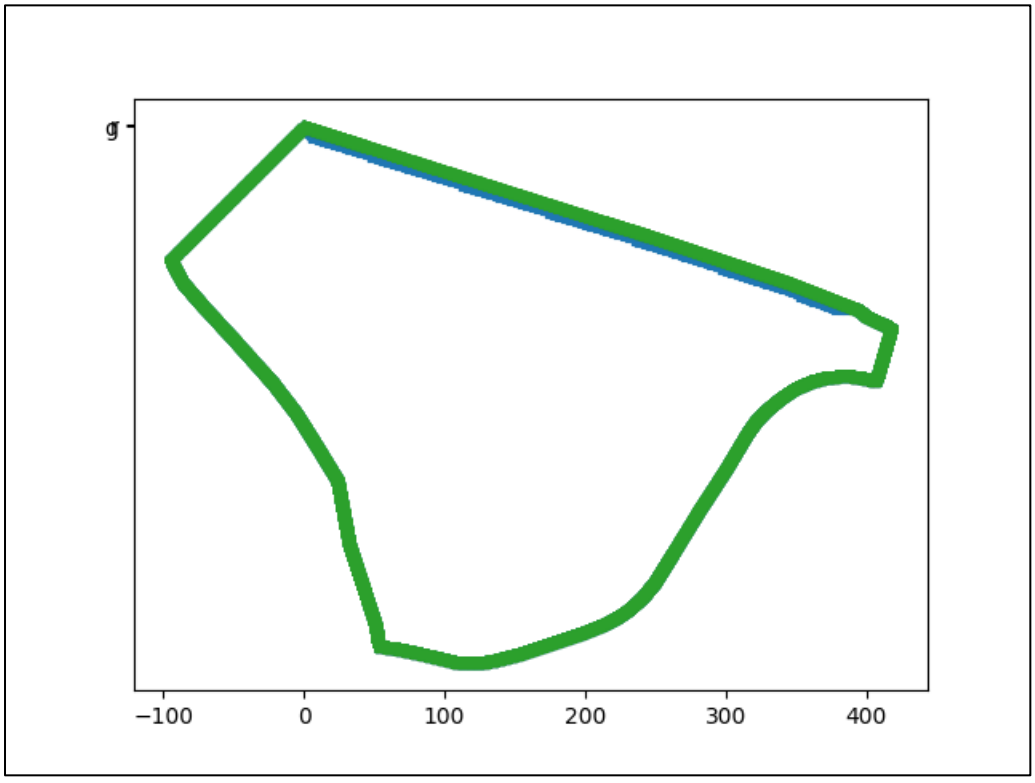

Figure 7. The path planning results from my Astar.py.

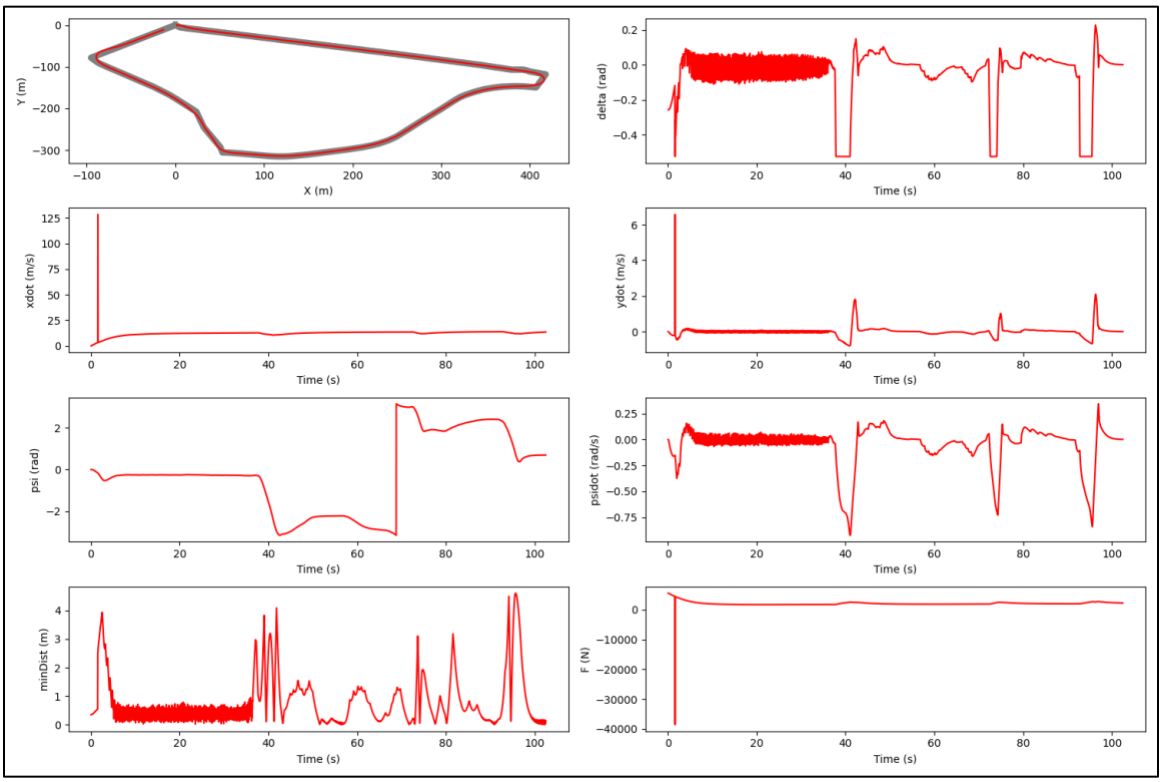

Figure 8. The final completion plot using my A algorithm & my LQR controller.*

```
[[0 0 0 ... 1 1 1]
 [0 0 0 ... 1 1 1]
 [0 0 0 ... 1 1 1]
 ...
 [1 1 1 ... 0 0 0]
 [1 1 1 ... 0 0 0]
 [1 1 1 ... 0 0 0]]
map size (108, 393)
reach goal
path length 393
total steps:  102464
maxMinDist:  4.606402184992135
avgMinDist:  0.8091769190754369
INFO: 'main' controller exited successfully.
```

Figure 9. The final score message from Webots simulation using my A algorithm & my LQR controller*

```python
# Fill in the respective functions to implement the controller

# Import libraries
import numpy as np
from base_controller import BaseController
from scipy import signal, linalg
from util import wrapToPi, closestNode

# CustomController class (inherits from BaseController)
class CustomController(BaseController):

    def __init__(self, trajectory, look_ahead_distance=190):

        super().__init__(trajectory)

        # Define constants
        # These can be ignored in P1
        self.lr = 1.39
        self.lf = 1.55
        self.Ca = 20000
        self.Iz = 25854
        self.m = 1888.6
        self.g = 9.81

        # Add additional member variables according to your need here.
        self.look_ahead_distance = look_ahead_distance
        self.previous_psi = 0
        self.velocity_start = 58
        self.velocity_integral_error = 0
        self.velocity_previous_step_error = 0

    def update(self, timestep):

        trajectory = self.trajectory

        lr = self.lr
```

```
37          lf = self.lf
38          Ca = self.Ca
39          Iz = self.Iz
40          m = self.m
41          g = self.g
42
43          # Fetch the states from the BaseController
    method
44          delT, X, Y, xdot, ydot, psi, psidot = super
    ().getStates(timestep)
45
46          #  Set the look-ahead distance and find the
     closest index to the current position
47          look_ahead_distance = 190
48          _, closest_index = closestNode(X,Y,
    trajectory)
49
50          # stop look-ahead distance from going out
    of bounds
51          max_allowed_look_ahead = min(
    look_ahead_distance, len(trajectory) -
    closest_index - 1)
52          look_ahead_distance = max(0,
    max_allowed_look_ahead)
53
54          # Design your controllers in the spaces
    below.
55          # Remember, your controllers will need to
    use the states
56          # to calculate control inputs (F, delta).
57
58          # ---------------|Lateral Controller
    |-------------------------
59
60          # Please design your lateral controller
    below.
61
62          # state space model for lateral control
63          A = np.array([[0, 1, 0, 0], [0, -4 * Ca / (
    m * xdot), 4 * Ca / m, (-2 * Ca * (lf - lr)) / (m
     * xdot)], [0, 0, 0, 1], [0, (-2 * Ca * (lf - lr
```

```
63 )) / (Iz * xdot), (2 * Ca * (lf - lr)) / Iz, (-2
    * Ca * (lf ** 2 + lr ** 2)) / (Iz * xdot)]])
64         B = np.array([[0], [2 * Ca / m], [0], [2
    * Ca * lf / Iz]])
65         C = np.eye(4)
66         D = np.zeros((4, 1))
67
68         # discretize the state space model
69         sys_continuous = signal.StateSpace(A, B, C
   , D)
70         sys_discretize = sys_continuous.
   to_discrete(delT)
71         A_discretize = sys_discretize.A
72         B_discretize = sys_discretize.B
73
74         # calculate the desired heading angle (
   psi_desired) (referencing Project 2 solution)
75         psi_desired = np.arctan2(trajectory[
   closest_index + look_ahead_distance, 1] - Y,
   trajectory[closest_index + look_ahead_distance, 0
   ] - X)
76
77         # error calculation (referencing Project 2
    solution)
78         e1 = (Y - trajectory[closest_index +
   look_ahead_distance,1])*np.cos(psi_desired) - (X
    - trajectory[closest_index+look_ahead_distance,0
   ])*np.sin(psi_desired)
79         e2 = wrapToPi(psi - psi_desired)
80         e1_dot = ydot + xdot * e2
81         e2_dot = psidot
82
83         # LQR controller design
84         Q = np.eye(4)
85         R = 40
86
87         # solve for P and gain matrix K
88         P = linalg.solve_discrete_are(A_discretize
   , B_discretize, Q, R)
89         K = linalg.inv(R + B_discretize.T @ P @
   B_discretize) @ (B_discretize.T @ P @ A_discretize
```

```
 89 )
 90
 91         # control delta calculation
 92         delta = (-K @ np.array([[e1], [e1_dot], [
    e2], [e2_dot]]))[0, 0]
 93         delta = np.clip(delta, -np.pi / 6, np.pi
     / 6)
 94
 95         # ---------------|Longitudinal Controller
    |-------------------------
 96
 97         #Please design your longitudinal
    controller below.
 98
 99         # declaring PID variables
100         Kp_velocity = 95
101         Ki_velocity = 1
102         Kd_velocity = 0.005
103
104         # velocity error calculation
105         velocity = np.sqrt(xdot ** 2 + ydot ** 2
    ) * 3.6
106         velocity_error = self.velocity_start -
    velocity
107         self.velocity_integral_error +=
    velocity_error * delT
108         velocity_derivative_error = (
    velocity_error - self.velocity_previous_step_error
    ) / delT
109
110         # F with PID feedback control
111         F = (velocity_error * Kp_velocity) + (self
    .velocity_integral_error * Ki_velocity) + (
112                         velocity_derivative_error *
    Kd_velocity)
113
114         # Return all states and calculated control
     inputs (F, delta)
115         return X, Y, xdot, ydot, psi, psidot, F,
    delta
116
```

```python
from os import close
import numpy as np
from heapq import heappop, heappush
import matplotlib.pyplot as plt
import sys

class Node(object):
    def __init__(self, pose):
        self.pose = np.array(pose)
        self.x = pose[0]
        self.y = pose[1]
        self.g_value = 0
        self.h_value = 0
        self.f_value = 0
        self.parent = None

    def __lt__(self, other):
        return self.f_value < other.f_value

    def __eq__(self, other):
        return (self.pose == other.pose).all()

class AStar(object):
    def __init__(self, map_path):
        self.map_path = map_path
        self.map = self.load_map(self.map_path).astype(int)
        #print(self.map)
        self.resolution = 0.05
        self.y_dim = self.map.shape[0]
        self.x_dim =self.map.shape[1]
        print(f'map size ({self.x_dim}, {self.y_dim})')

    def load_map(self, path):
        #return np.load(path)
        return np.genfromtxt(path, delimiter = ",")

    def reset_map(self):
        self.map = self.load_map(self.map_path)

```

```
40      def heuristic(self, current, goal):
41          """
42          TODO:
43          Euclidean distance
44          """
45          # Euclidean distance calculation
46          dx = current.x - goal.x
47          dy = current.y - goal.y
48          Euclidean_distance = np.sqrt(dx**2 + dy**2)
49
50          return Euclidean_distance
51
52      def get_successor(self, node):
53          successor_list = []
54          x,y = node.pose
55          pose_list = [[x+1, y+1], [x, y+1], [x-1, y+
    1], [x-1, y],
56                               [x-1, y-1], [x, y-1], [x+1
    , y-1], [x+1, y]]
57
58          for pose_ in pose_list:
59              x_, y_ = pose_
60              if 0 <= x_ < self.y_dim and 0 <= y_ <
    self.x_dim and self.map[x_, y_] == 0:
61                  self.map[x_, y_] = -1
62                  successor_list.append(Node(pose_))
63
64          return successor_list
65
66      def calculate_path(self, node):
67          path_ind = []
68          path_ind.append(node.pose.tolist())
69          current = node
70          while current.parent:
71              current = current.parent
72              path_ind.append(current.pose.tolist())
73          path_ind.reverse()
74          print(f'path length {len(path_ind)}')
75          path = list(path_ind)
76
77          return path
```

```
 78
 79      def plan(self, start_ind, goal_ind):
 80          """
 81          TODO:
 82          Fill in the missing lines in the plan
    function
 83          @param start_ind : [x, y] represents
    coordinates in webots world
 84          @param goal_ind : [x, y] represents
    coordinates in webots world
 85          @return path : a list with shape (n, 2)
    containing n path point
 86          """
 87
 88          # initialize start node and goal node
    class
 89          start_node = Node(start_ind)
 90          goal_node = Node(goal_ind)
 91          """
 92          TODO:
 93          calculate h and f value of start_node
 94          (1) h can be computed by calling the
    heuristic method
 95          (2) f = g + h
 96          """
 97          # calculate h and f value of start_node
 98          start_node.g_value = 0
 99          start_node.h_value = self.heuristic(
    start_node, goal_node)
100          start_node.f_value = start_node.g_value +
    start_node.h_value
101
102          """
103          END TODO
104          """
105
106          # Reset map
107          self.reset_map()
108
109          # Initially, only the start node is known.
110          # This is usually implemented as a min-
```

```
110 heap or priority queue rather than a hash-set.
111         # Please refer to https://docs.python.org/
    3/library/heapq.html for more details about heap
    data structure
112         open_list = []
113         closed_list = np.array([])
114         heappush(open_list, start_node)
115 
116         # while open_list is not empty
117         while len(open_list):
118 
119             """
120             TODO:
121             get the current node and add it to the
     closed list
122             """
123             # Current is the node in open_list
    that has the lowest f value
124             # This operation can occur in O(1)
    time if open_list is a min-heap or a priority
    queue
125 
126             # get and add current node to the
    closed list
127             current = heappop(open_list)
128             """
129             END TODO
130             """
131             closed_list = np.append(closed_list,
    current)
132 
133             self.map[current.x, current.y] = -1
134 
135             # if current is goal_node: calculate
    the path by passing through the current node
136             # exit the loop by returning the path
137             if current == goal_node:
138                 print('reach goal')
139                 return self.calculate_path(current
    )
140 
```

```
141             for successor in self.get_successor(
    current):
142                 """
143                 TODO:
144                 1. pass current node as parent of
    successor node
145                 2. calculate g, h, and f value of
    successor node
146                     (1) d(current, successor) is
    the weight of the edge from current to successor
147                     (2) g(successor) = g(current
    ) + d(current, successor)
148                     (3) h(successor) can be
    computed by calling the heuristic method
149                     (4) f(successor) = g(successor
    ) + h(successor)
150                 """
151                 successor.parent = current
152                 successor.g_value = current.
    g_value + 1
153                 successor.h_value = self.heuristic
    (successor, goal_node)
154                 successor.f_value = successor.
    g_value + successor.h_value
155
156                 if tuple(successor.pose) in
    closed_list:
157                     continue
158
159                 in_open_list = any(successor ==
    node for node in open_list)
160
161                 if not in_open_list or successor.
    g_value < current.g_value:
162                     if in_open_list:
163                         open_list.remove(successor
    )
164                     heappush(open_list, successor)
165                 """
166                 END TODO
167                 """
```

```
168
169         # If the loop is exited without return any
     path
170         # Path is not found
171         print('path not found')
172         return None
173
174     def run(self, cost_map, start_ind, goal_ind):
175         '''
176         Change the original main function to a
    method "run" inside the AStar class
177         '''
178
179         if cost_map[start_ind[0], start_ind[1
    ]] == 0 and cost_map[goal_ind[0], goal_ind[1]] ==
    0:
180             return self.plan(start_ind, goal_ind)
181
182         else:
183             print('already occupied')
184
185
186 def visualize_path(cost_map, path, title):
187     x = [item[0] for item in path]
188     x = x[1:-1]
189     y = [item[1] for item in path]
190     y = y[1:-1]
191
192     plt.imshow(np.transpose(cost_map))
193     plt.plot(path[0][0], path[0][1], 'x', color =
    'r', label = 'start', markersize = 10)
194     plt.plot(path[-1][0], path[-1][1], 'o', color
     = 'r', label = 'goal', markersize = 10)
195     plt.scatter(x, y, label = 'path', s = 1)
196     plt.legend()
197     plt.title(title)
198     plt.show()
199
200 if __name__ == "__main__":
201     costmap1 = np.genfromtxt('map1.csv', delimiter
     = ',')
```

```
202     costmap2 = np.genfromtxt('map2.csv', delimiter
    = ',')
203     costmap3 = np.genfromtxt('map3.csv', delimiter
    = ',')
204
205     # plt.imshow(np.transpose(costmap3))
206     # plt.show()
207
208     start_ind1 = [159, 208]
209     goal_ind1 = [231, 1369]
210     start_ind2 = [119, 45]
211     goal_ind2 = [123, 247]
212     start_ind3 = [25, 100]
213     goal_ind3 = [175, 100]
214
215     Planner1 = AStar('map1.csv')
216     Planner2 = AStar('map2.csv')
217     Planner3 = AStar('map3.csv')
218
219     path_ind1 = Planner1.run(costmap1, start_ind1
    , goal_ind1)
220     path_ind2 = Planner2.run(costmap2, start_ind2
    , goal_ind2)
221     path_ind3 = Planner3.run(costmap3, start_ind3
    , goal_ind3)
222
223     visualize_path(costmap1, path_ind1, 'A Star
    Planning for Costmap 1')
224     visualize_path(costmap2, path_ind2, 'A Star
    Planning for Costmap 2')
225     visualize_path(costmap3, path_ind3, 'A Star
    Planning for Costmap 3')
226
```

```python
from os import close
import numpy as np
from heapq import heappop, heappush
import matplotlib.pyplot as plt

class Node(object):
    """
    Class Node: a data structure that help process calculation of AStar
    """
    def __init__(self, pose):
        """
        param self.pose: [x, y] index position of node
        """
        self.pose = np.array(pose)
        self.x = pose[0]
        self.y = pose[1]
        self.g_value = 0
        self.h_value = 0
        self.f_value = 0
        self.parent = None

    def __lt__(self, other):
        """
        less than function for heap comparison
        """
        return self.f_value < other.f_value

    def __eq__(self, other):
        return (self.pose == other.pose).all()

class AStar(object):
    def __init__(self, map_path):
        self.map_path = map_path
        self.map = self.load_map(self.map_path).astype(int)
        print(self.map)
        self.resolution = 0.05
        self.y_dim = self.map.shape[0]
        self.x_dim =self.map.shape[1]
```

```
39          print(f'map size ({self.x_dim}, {self.y_dim
    })')
40
41      def load_map(self, path):
42          return np.load(path)
43
44      def reset_map(self):
45          self.map = self.load_map(self.map_path)
46
47      def heuristic(self, current, goal):
48          """
49          TODO:
50          Euclidean distance
51          """
52          # Euclidean distance calculation
53          dx = current.x - goal.x
54          dy = current.y - goal.y
55          Euclidean_distance = np.sqrt(dx**2 + dy**2)
56
57          return Euclidean_distance
58
59      def get_successor(self, node):
60          """
61          :param node: A Node data structure
62          :return: a list of Nodes containing
    successors of current Node
63          """
64          successor_list = []
65          x,y = node.pose  # Get x, y coordinates of
    the current node
66          pose_list = [[x+1, y+1], [x, y+1], [x-1, y+
    1], [x-1, y],
67                               [x-1, y-1], [x, y-1], [x+1
    , y-1], [x+1, y]]  # Pose list contains 8 neighbors
     of the current node
68
69          for pose_ in pose_list:
70              x_, y_ = pose_
71              if 0 <= x_ < self.y_dim and 0 <= y_ <
    self.x_dim and self.map[x_, y_] == 0: # Eliminate
    nodes that are out of bound, and nodes that are
```

```
71 obstacles
72                     self.map[x_, y_] = -1
73                     successor_list.append(Node(pose_))
74 
75         return successor_list
76 
77     def calculate_path(self, node):
78         """
79         :param node: A Node data structure
80         :return: a list with shape (n, 2)
   containing n path point
81         """
82         path_ind = []
83         path_ind.append(node.pose.tolist())
84         current = node
85         while current.parent:
86             current = current.parent
87             path_ind.append(current.pose.tolist())
88         path_ind.reverse()
89         print(f'path length {len(path_ind)}')
90         path = list(path_ind)
91 
92         return path
93 
94     def plan(self, start_ind, goal_ind):
95         """
96         TODO:
97         Fill in the missing lines in the plan
   function
98         @param start_ind : [x, y] represents
   coordinates in webots world
99         @param goal_ind : [x, y] represents
   coordinates in webots world
100        @return path : a list with shape (n, 2)
   containing n path point
101        """
102 
103        # initialize start node and goal node
   class
104        start_node = Node(start_ind)
105        goal_node = Node(goal_ind)
```

```
106         """
107         TODO:
108         calculate h and f value of start_node
109         (1) h can be computed by calling the heuristic method
110         (2) f = g + h
111         """
112         # calculate h and f value of start_node
113         start_node.g_value = 0
114         start_node.h_value = self.heuristic(start_node, goal_node)
115         start_node.f_value = start_node.g_value + start_node.h_value
116 
117         """
118         END TODO
119         """
120 
121         # Reset map
122         self.reset_map()
123 
124         # Initially, only the start node is known.
125         # This is usually implemented as a min-heap or priority queue rather than a hash-set.
126         # Please refer to https://docs.python.org/3/library/heapq.html for more details about heap data structure
127         open_list = []
128         closed_list = np.array([])
129         heappush(open_list, start_node)
130 
131         # while open_list is not empty
132         while len(open_list):
133 
134             """
135             TODO:
136             get the current node and add it to the closed list
137             """
138             # Current is the node in open_list that has the lowest f value
```

```
139             # This operation can occur in O(1)
    time if open_list is a min-heap or a priority
    queue
140
141             # get and add current node to the
    closed list
142             current = heappop(open_list)
143             """
144             END TODO
145             """
146             closed_list = np.append(closed_list,
    current)
147
148             self.map[current.x, current.y] = -1
149
150             # if current is goal_node: calculate
    the path by passing through the current node
151             # exit the loop by returning the path
152             if current == goal_node:
153                 print('reach goal')
154                 return self.calculate_path(current
    )
155
156             for successor in self.get_successor(
    current):
157                 """
158                 TODO:
159                 1. pass current node as parent of
    successor node
160                 2. calculate g, h, and f value of
    successor node
161                     (1) d(current, successor) is
    the weight of the edge from current to successor
162                     (2) g(successor) = g(current
    ) + d(current, successor)
163                     (3) h(successor) can be
    computed by calling the heuristic method
164                     (4) f(successor) = g(successor
    ) + h(successor)
165                 """
166                 successor.parent = current
```

```
167                 successor.g_value = current.
    g_value + 1
168                 successor.h_value = self.heuristic
    (successor, goal_node)
169                 successor.f_value = successor.
    g_value + successor.h_value
170 
171                 if tuple(successor.pose) in
    closed_list:
172                     continue
173 
174                 in_open_list = any(successor ==
    node for node in open_list)
175 
176                 if not in_open_list or successor.
    g_value < current.g_value:
177                     if in_open_list:
178                         open_list.remove(successor
    )
179                     heappush(open_list, successor)
180                 """
181                 END TODO
182                 """
183 
184         # If the loop is exited without return any
     path
185         # Path is not found
186         print('path not found')
187         return None
188 
189     def run(self, cost_map, start_ind, goal_ind):
190         if cost_map[start_ind[0], start_ind[1
    ]] == 0 and cost_map[goal_ind[0], goal_ind[1]] ==
    0:
191             return self.plan(start_ind, goal_ind)
192 
193         else:
194             print('already occupied')
195 
```

```python
# Fill in the respective functions to implement the LQR optimal controller

# Import libraries
import numpy as np
from base_controller import BaseController
from scipy import signal, linalg
from util import wrapToPi, closestNode

class CustomController(BaseController):

    def __init__(self, trajectory, look_ahead_distance=190):

        super().__init__(trajectory)

        # Define constants
        # These can be ignored in P1
        self.lr = 1.39
        self.lf = 1.55
        self.Ca = 20000
        self.Iz = 25854
        self.m = 1888.6
        self.g = 9.81

        # Add additional member variables according to your need here.
        self.look_ahead_distance = look_ahead_distance
        self.previous_psi = 0
        self.velocity_start = 58
        self.velocity_integral_error = 0
        self.velocity_previous_step_error = 0
    def update(self, timestep):

        trajectory = self.trajectory

        lr = self.lr
        lf = self.lf
        Ca = self.Ca
        Iz = self.Iz
```

```
38          m = self.m
39          g = self.g
40
41          # Fetch the states from the BaseController
    method
42          delT, X, Y, xdot, ydot, psi, psidot,
    obstacleX, obstacleY = super().getStates(timestep)
43
44          #  Set the look-ahead distance and find the
     closest index to the current position
45          look_ahead_distance = 190
46          _, closest_index = closestNode(X, Y,
    trajectory)
47
48          # stop look-ahead distance from going out
    of bounds
49          max_allowed_look_ahead = min(
    look_ahead_distance, len(trajectory) -
    closest_index - 1)
50          look_ahead_distance = max(0,
    max_allowed_look_ahead)
51
52          # Design your controllers in the spaces
    below.
53          # Remember, your controllers will need to
    use the states
54          # to calculate control inputs (F, delta).
55
56          # ---------------|Lateral Controller
    |-------------------------
57
58          # Please design your lateral controller
    below.
59
60          # state space model for lateral control
61          A = np.array(
62              [[0, 1, 0, 0], [0, -4 * Ca / (m * xdot
    ), 4 * Ca / m, (-2 * Ca * (lf - lr)) / (m * xdot
    )], [0, 0, 0, 1],
63               [0, (-2 * Ca * (lf - lr)) / (Iz * xdot
    ), (2 * Ca * (lf - lr)) / Iz,
```

```
64                 (-2 * Ca * (lf ** 2 + lr ** 2)) / (
   Iz * xdot)]])
65         B = np.array([[0], [2 * Ca / m], [0], [2
    * Ca * lf / Iz]])
66         C = np.eye(4)
67         D = np.zeros((4, 1))
68
69         # discretize the state space model
70         sys_continuous = signal.StateSpace(A, B, C
   , D)
71         sys_discretize = sys_continuous.
   to_discrete(delT)
72         A_discretize = sys_discretize.A
73         B_discretize = sys_discretize.B
74
75         # calculate the desired heading angle (
   psi_desired) (referencing Project 2 solution)
76         psi_desired = np.arctan2(trajectory[
   closest_index + look_ahead_distance, 1] -
   trajectory[closest_index, 1],
77                                  trajectory[
   closest_index + look_ahead_distance, 0] -
   trajectory[closest_index, 0])
78
79         # error calculation (referencing Project 2
    solution)
80         e1 = (Y - trajectory[closest_index +
   look_ahead_distance, 1]) * np.cos(psi_desired) - (
81                     X - trajectory[closest_index
    + look_ahead_distance, 0]) * np.sin(psi_desired)
82         e2 = wrapToPi(psi - psi_desired)
83         e1_dot = ydot + xdot * e2
84         e2_dot = psidot
85
86         # LQR controller design
87         Q = np.eye(4)
88         R = 40
89
90         # solve for P and gain matrix K
91         P = linalg.solve_discrete_are(A_discretize
   , B_discretize, Q, R)
```

```
 92          K = linalg.inv(R + B_discretize.T @ P @ B_discretize) @ (B_discretize.T @ P @ A_discretize)
 93
 94          # control delta calculation
 95          delta = (-K @ np.array([[e1], [e1_dot], [e2], [e2_dot]]))[0, 0]
 96          delta = np.clip(delta, -np.pi / 6, np.pi / 6)
 97
 98          # ---------------|Longitudinal Controller|-------------------------
 99
100          # Please design your longitudinal controller below.
101
102          # declaring PID variables
103          Kp_velocity = 95
104          Ki_velocity = 1
105          Kd_velocity = 0.005
106
107          # velocity error calculation
108          velocity = np.sqrt(xdot ** 2 + ydot ** 2) * 3.6
109          velocity_error = self.velocity_start - velocity
110          self.velocity_integral_error += velocity_error * delT
111          velocity_derivative_error = (velocity_error - self.velocity_previous_step_error) / delT
112
113          # F with PID feedback control
114          F = (velocity_error * Kp_velocity) + (self.velocity_integral_error * Ki_velocity) + (
115                  velocity_derivative_error * Kd_velocity)
116          # Return all states and calculated control inputs (F, delta) and obstacle position
117          return X, Y, xdot, ydot, psi, psidot, F, delta, obstacleX, obstacleY
```