```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
```

```
        m = self.m
        g = self.g

        # Fetch the states from the BaseController method
        delT, X, Y, xdot, ydot, psi, psidot, obstacleX, obstacleY = super().getStates(timestep)

        #  Set the look-ahead distance and find the closest index to the current position
        look_ahead_distance = 190
        _, closest_index = closestNode(X, Y, trajectory)

        # stop look-ahead distance from going out of bounds
        max_allowed_look_ahead = min(look_ahead_distance, len(trajectory) - closest_index - 1)
        look_ahead_distance = max(0, max_allowed_look_ahead)

        # Design your controllers in the spaces below.
        # Remember, your controllers will need to use the states
        # to calculate control inputs (F, delta).

        # ---------------|Lateral Controller |-------------------------

        # Please design your lateral controller below.

        # state space model for lateral control
        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)) / (Iz * xdot), (2 * Ca * (lf - lr)) / Iz,
```

```
              (-2 * Ca * (lf ** 2 + lr ** 2)) / (Iz * xdot)]])
        B = np.array([[0], [2 * Ca / m], [0], [2 * Ca * lf / Iz]])
        C = np.eye(4)
        D = np.zeros((4, 1))

        # discretize the state space model
        sys_continuous = signal.StateSpace(A, B, C, D)
        sys_discretize = sys_continuous.to_discrete(delT)
        A_discretize = sys_discretize.A
        B_discretize = sys_discretize.B

        # calculate the desired heading angle (psi_desired) (referencing Project 2 solution)
        psi_desired = np.arctan2(trajectory[closest_index + look_ahead_distance, 1] - trajectory[closest_index, 1],
                                 trajectory[closest_index + look_ahead_distance, 0] - trajectory[closest_index, 0])

        # error calculation (referencing Project 2 solution)
        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)
        e2 = wrapToPi(psi - psi_desired)
        e1_dot = ydot + xdot * e2
        e2_dot = psidot

        # LQR controller design
        Q = np.eye(4)
        R = 40

        # solve for P and gain matrix K
        P = linalg.solve_discrete_are(A_discretize, B_discretize, Q, R)
```

```
        K = linalg.inv(R + B_discretize.T @ P @ B_discretize) @ (B_discretize.T @ P @ A_discretize)

        # control delta calculation
        delta = (-K @ np.array([[e1], [e1_dot], [e2], [e2_dot]]))[0, 0]
        delta = np.clip(delta, -np.pi / 6, np.pi / 6)

        # ---------------|Longitudinal Controller|-------------------------

        # Please design your longitudinal controller below.

        # declaring PID variables
        Kp_velocity = 95
        Ki_velocity = 1
        Kd_velocity = 0.005

        # velocity error calculation
        velocity = np.sqrt(xdot ** 2 + ydot ** 2) * 3.6
        velocity_error = self.velocity_start - velocity
        self.velocity_integral_error += velocity_error * delT
        velocity_derivative_error = (velocity_error - self.velocity_previous_step_error) / delT

        # F with PID feedback control
        F = (velocity_error * Kp_velocity) + (self.velocity_integral_error * Ki_velocity) + (
                velocity_derivative_error * Kd_velocity)
        # Return all states and calculated control inputs (F, delta) and obstacle position
        return X, Y, xdot, ydot, psi, psidot, F, delta, obstacleX, obstacleY
```