```python
# Fill in the respective function to implement the LQR/EKF SLAM controller

# Import libraries
import numpy as np
from base_controller import BaseController
from scipy import signal, linalg
from scipy.spatial.transform import Rotation
from util import *
from ekf_slam import EKF_SLAM

# 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

        self.counter = 0
        np.random.seed(99)

        # 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

```

```python
    def getStates(self, timestep, use_slam=False):

        delT, X, Y, xdot, ydot, psi, psidot = super().getStates(timestep)

        # Initialize the EKF SLAM estimation
        if self.counter == 0:
            # Load the map
            minX, maxX, minY, maxY = -120., 450., -500., 50.
            map_x = np.linspace(minX, maxX, 7)
            map_y = np.linspace(minY, maxY, 7)
            map_X, map_Y = np.meshgrid(map_x, map_y)
            map_X = map_X.reshape(-1,1)
            map_Y = map_Y.reshape(-1,1)
            self.map = np.hstack((map_X, map_Y)).reshape((-1))

            # Parameters for EKF SLAM
            self.n = int(len(self.map)/2)
            X_est = X + 0.5
            Y_est = Y - 0.5
            psi_est = psi - 0.02
            mu_est = np.zeros(3+2*self.n)
            mu_est[0:3] = np.array([X_est, Y_est, psi_est])
            mu_est[3:] = np.array(self.map)
            init_P = 1*np.eye(3+2*self.n)
            W = np.zeros((3+2*self.n, 3+2*self.n))
            W[0:3, 0:3] = delT**2 * 0.1 * np.eye(3)
            V = 0.1*np.eye(2*self.n)
            V[self.n:, self.n:] = 0.01*np.eye(self.n)
            # V[self.n:] = 0.01
            print(V)

            # Create a SLAM
            self.slam = EKF_SLAM(mu_est, init_P, delT, W, V, self.n)
```

```
            self.counter += 1
        else:
            mu = np.zeros(3+2*self.n)
            mu[0:3] = np.array([X,
                                Y,
                                psi])
            mu[3:] = self.map
            y = self._compute_measurements(X, Y, psi)
            mu_est, _ = self.slam.predict_and_correct(y, self.previous_u)

        self.previous_u = np.array([xdot, ydot, psidot])

        print("True      X, Y, psi:", X, Y, psi)
        print("Estimated X, Y, psi:", mu_est[0], mu_est[1], mu_est[2])
        print("-------------------------------------------------------")

        if use_slam == True:
            return delT, mu_est[0], mu_est[1], xdot, ydot, mu_est[2], psidot
        else:
            return delT, X, Y, xdot, ydot, psi, psidot

    def _compute_measurements(self, X, Y, psi):
        x = np.zeros(3+2*self.n)
        x[0:3] = np.array([X, Y, psi])
        x[3:] = self.map

        p = x[0:2]
        psi = x[2]
        m = x[3:].reshape((-1,2))

        y = np.zeros(2*self.n)

        for i in range(self.n):
```

```
            y[i] = np.linalg.norm(m[i, :] - p)
            y[self.n+i] = wrapToPi(np.arctan2(m[i, 1]-p[1], m[i,0]-p[0]) - psi)

        y = y + np.random.multivariate_normal(np.zeros(2*self.n), self.slam.V)
        # print(np.random.multivariate_normal(np.zeros(2*self.n), self.slam.V))
        return y

    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 newly defined getStates method
        delT, X, Y, xdot, ydot, psi, psidot = self.getStates(timestep, use_slam=True)
        # You must not use true_X, true_Y and true_psi since they are for plotting purpose
        # _, true_X, true_Y, _, _, true_psi, _ = self.getStates(timestep, use_slam=False)

        # You are free to reuse or refine your code from P3 in the spaces below.
        # 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

        #  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)

        # 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)
        return X, Y, xdot, ydot, psi, psidot, F, delta
```