```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 closestNode, wrapToPi
from scipy.signal import place_poles

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

    def __init__(self, trajectory, look_ahead_distance=50):

        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 = 30
        self.velocity_integral_error = 0
        self.velocity_previous_step_error = 0

    def update(self, timestep):

        trajectory = self.trajectory
        lr = self.lr
```

```python
        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 = super().getStates(timestep)

        #  Set the look-ahead distance
        look_ahead_distance = 100
        _, closest_index = closestNode(X,Y, trajectory)

        if look_ahead_distance + closest_index >= 8203:
            look_ahead_distance = 0

        # Calculate the look-ahead distance
        closest_index = np.argmin(np.sqrt((trajectory[:, 0] - X) ** 2 + (trajectory[:, 1] - Y) ** 2))
        look_ahead_distance = min(self.look_ahead_distance, len(trajectory) - closest_index - 1)
        # look_ahead_X, look_ahead_Y = trajectory[closest_index + look_ahead_distance]

        # Calculate the desired heading angle
        X_desired = trajectory[closest_index + look_ahead_distance][0]
        Y_desired = trajectory[closest_index + look_ahead_distance][1]
        psi_desired = np.arctan2(Y_desired - Y, X_desired - X)

        # 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]])

        # desired poles
        P = np.array([-4, -1, -3, -2])

        # calculate the gain matrix K using pole placement
        K = place_poles(A, B, P).gain_matrix

        # calculate lateral control error vector E
        e1 = 0
        e2 = wrapToPi(psi - psi_desired)
        e1dot = ydot + xdot * e2
        e2dot = psidot
        E = np.array([e1, e2, e1dot, e2dot])

        # control delta using the gain matrix K and error vector E
        delta = -np.dot(K, E)[0]
        delta = np.clip(delta, -np.pi/6, np.pi/6)

        # update the previous psi
        self.previous_psi = psi

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

```python

        # Please design your longitudinal controller below.

        # declaring PID variables
        Kp_velocity = 90
        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

```