```
# 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 *

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

    def __init__(self, trajectory):


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.velocity_start = 30
        self.velocity_integral_error = 0

    self.velocity_previous_step_error = 0
        self.radian_integral_error = 0

self.radian_previous_step_error = 0


        """

self.max_deceleration = self.velocity_start
        self.previous_psi = 0

    def
detect_corner(self, psi, previous_psi, threshold_radian):


        delta_psi =
wrapToPi(psi - previous_psi)
        velocity_decrease = 0.2


        return
abs(delta_psi) > threshold_radian
        """

    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

        #
declaring PID variables
```

```
        Kp_velocity= 90
        Ki_velocity = 1
        Kd_velocity=
0.005

        Kp_radian = 0.2
        Ki_radian = 0.05
        Kd_radian = 0.02


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

        # 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.

        # 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)

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

        # Please
design your longitudinal controller below.

        min_distance, min_index =
closestNode(X, Y, trajectory)
        index = min_index + 30
        X_distance =
trajectory[index][0] - X
        Y_distance = trajectory[index][1] - Y
        alpha =
np.arctan2(Y_distance, X_distance)
        alpha = wrapToPi(alpha)


radian_error = alpha - psi
        radian_error = wrapToPi(radian_error)

self.radian_integral_error += radian_error * delT
        radian_derivative_error =
radian_error / delT

        # delta with PID feedback control and steering limit

      delta_max = np.radians(40)
        delta = (radian_error * Kp_radian) +
(self.radian_integral_error * Ki_radian) + (radian_derivative_error * Kd_radian)
        delta
= np.clip(delta, -delta_max, delta_max)


        # Update previous error terms
for next iteration
```

```
        self.velocity_previous_step_error = velocity_error

self.radian_previous_step_error = radian_error

        """
        #
---------------|Corner Braking|------------------------
        threshold_radian = 0.01

 max_brake_force = 10000

        is_corner = self.detect_corner(psi,
self.previous_psi, threshold_radian)

        if is_corner:


velocity_corner = velocity_decrease * self.velocity_start
            velocity_error =
velocity_corner - velocity

            brake_distance = (velocity ** 2) / (2 *
self.max_deceleration)
            deceleration = (velocity ** 2) / (2 * brake_distance)


            brake_command = deceleration / max_brake_force

            delta =
self.change_longitudinal_controller(brake_command, delT)


print("Velocity: ", velocity)
            print("Brake Distance: ",
brake_distance)
            print("Deceleration: ", deceleration)

print("Brake Command: ", brake_command)

        else:


    delta = np.clip(delta, -delta_max, delta_max)
        #

        # Update previous
error terms for next iteration
        self.velocity_previous_step_error = velocity_error

  self.radian_previous_step_error = radian_error

        self.previous_psi = psi

  """

        # Return all states and calculated control inputs (F,
delta)
        return X, Y, xdot, ydot, psi, psidot, F, delta
```