Deadline: ⅗
Summarizing progress and findings,

Contribution of each member

Planning and Control Task 1: Longitudinal and Lateral Control of vehicles
Summary:

- Important Concept:
  - Waypoints:
    - The waypoints variable is a Python list of waypoints to track where each row denotes a waypoint of format [x, y, v], which are the x and y positions as well as the desired speed at that position, respectively.
  - Kinematic bicycle model:
    - The kinematic bicycle is a simplified car model, which could capture vehicle motion in normal driving conditions.
  - Proportional Integral and Derivative (PID) controller:
    - A PID controller is a control loop mechanism employing feedback that is widely used in industrial control systems and a variety of other applications requiring continuously modulated control.
    - PID control is mathematically formulated by adding three terms dependent on the error function:
      - A proportional term directly proportional to the error e
      - An integral term proportional to the integral of the error e
      - A derivative term proportional to the derivative of the error e
  - Stanley controller
    - The stanley controller is a geometric path tracking controller which is simple but useful for autonomous robotics and autonomous cars
  - Longitudinal control:
    - For the longitudinal control of a vehicle, a PID controller will be deployed. The PID controller will take the desired speed as the reference and output the throttle and the brake.
  - Lateral control:
    - For lateral control, the Stanley controller will be implemented. The final steering input is the summation of the steering derived from the cross track error and the steering derived from the heading error is the total steering input, which completes the lateral controller.

- Implementing longitudinal control and lateral control in the controller2d.py
  Code:

```
#!/usr/bin/env python3

"""
2D Controller Class to be used for the CARLA waypoint follower demo.
```

```
"""

import cutils
import numpy as np
import math

class Controller2D(object):
    def __init__(self, waypoints):
        self.vars                = cutils.CUtils()
        self._current_x          = 0
        self._current_y          = 0
        self._current_yaw        = 0
        self._current_speed      = 0
        self._desired_speed      = 0
        self._current_frame      = 0
        self._current_timestamp  = 0
        self._start_control_loop = False
        self._set_throttle       = 0
        self._set_brake          = 0
        self._set_steer          = 0
        self._waypoints          = waypoints
        self._conv_rad_to_steer  = 180.0 / 70.0 / np.pi
        self._pi                 = np.pi
        self._2pi                = 2.0 * np.pi

    def update_values(self, x, y, yaw, speed, timestamp, frame):
        self._current_x         = x
        self._current_y         = y
        self._current_yaw       = yaw
        self._current_speed     = speed
        self._current_timestamp = timestamp
        self._current_frame     = frame
        if self._current_frame:
            self._start_control_loop = True

    def update_desired_speed(self):
        min_idx       = 0
        min_dist      = float("inf")
        desired_speed = 0
        for i in range(len(self._waypoints)):
            dist = np.linalg.norm(np.array([
                    self._waypoints[i][0] - self._current_x,
                    self._waypoints[i][1] - self._current_y]))
            if dist < min_dist:
```

```
                min_dist = dist
                min_idx = i
        if min_idx < len(self._waypoints)-1:
            desired_speed = self._waypoints[min_idx][2]
        else:
            desired_speed = self._waypoints[-1][2]
        self._desired_speed = desired_speed

    def update_waypoints(self, new_waypoints):
        self._waypoints = new_waypoints

    def get_commands(self):
        return self._set_throttle, self._set_steer, self._set_brake

    def set_throttle(self, input_throttle):
        # Clamp the throttle command to valid bounds
        throttle           = np.fmax(np.fmin(input_throttle, 1.0), 0.0)
        self._set_throttle = throttle

    def set_steer(self, input_steer_in_rad):
        # Covnert radians to [-1, 1]
        input_steer = self._conv_rad_to_steer * input_steer_in_rad

        # Clamp the steering command to valid bounds
        steer           = np.fmax(np.fmin(input_steer, 1.0), -1.0)
        self._set_steer = steer

    def set_brake(self, input_brake):
        # Clamp the steering command to valid bounds
        brake           = np.fmax(np.fmin(input_brake, 1.0), 0.0)
        self._set_brake = brake

    def update_controls(self):
        ######################################################
        # RETRIEVE SIMULATOR FEEDBACK
        ######################################################
        x               = self._current_x
        y               = self._current_y
        yaw             = self._current_yaw
        v               = self._current_speed
        self.update_desired_speed()
        v_desired       = self._desired_speed
        t               = self._current_timestamp
        waypoints       = self._waypoints
```

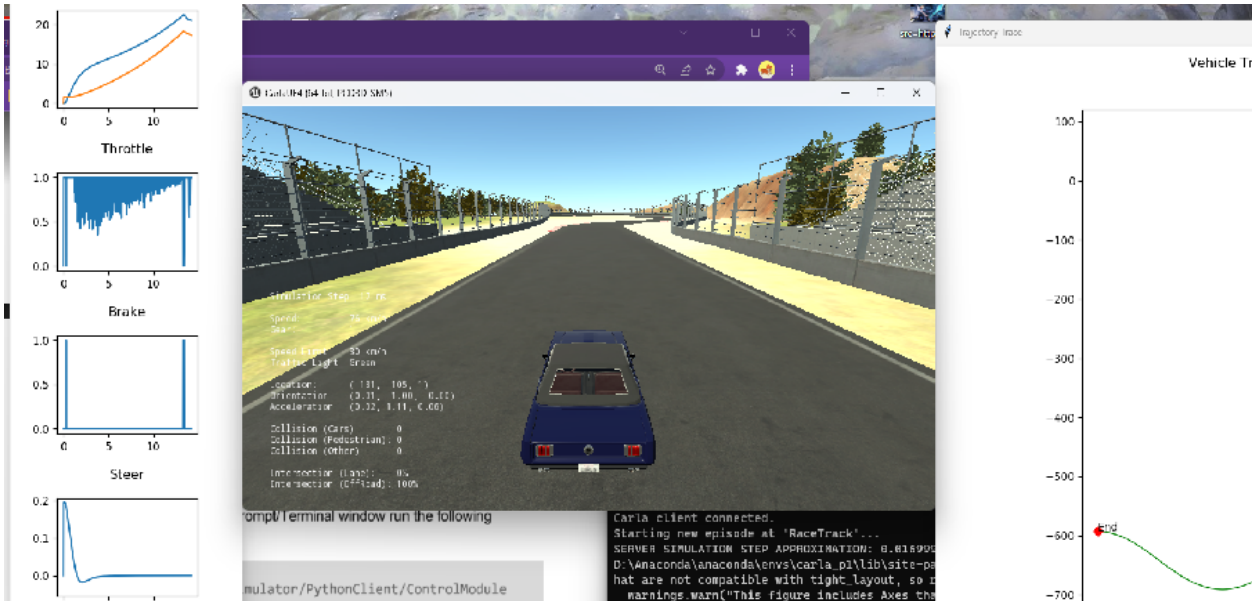
```
throttle_output = 0
steer_output    = 0
brake_output    = 0

######################################################
######################################################
# MODULE 7: DECLARE USAGE VARIABLES HERE
######################################################
######################################################
"""
   Use 'self.vars.create_var(<variable name>, <default value>)'
   to create a persistent variable (not destroyed at each iteration).
   This means that the value can be stored for use in the next
   iteration of the control loop.

   Example: Creation of 'v_previous', default value to be 0
   self.vars.create_var('v_previous', 0.0)

   Example: Setting 'v_previous' to be 1.0
   self.vars.v_previous = 1.0

   Example: Accessing the value from 'v_previous' to be used
   throttle_output = 0.5 * self.vars.v_previous
"""
self.vars.create_var('v_previous', 0.0)
self.vars.create_var('t_previous', -100)
self.vars.create_var('e_previous', 0.0)
self.vars.create_var('I', 0.0)

# Skip the first frame to store previous values properly
if self._start_control_loop:
   """
      Controller iteration code block.

      Controller Feedback Variables:
         x              : Current X position (meters)
         y              : Current Y position (meters)
         yaw            : Current yaw pose (radians)
         v              : Current forward speed (meters per second)
         t              : Current time (seconds)
         v_desired      : Current desired speed (meters per second)
                          (Computed as the speed to track at the
                          closest waypoint to the vehicle.)
         waypoints      : Current waypoints to track
```

```
                    (Includes speed to track at each x,y
                    location.)
                    Format: [[x0, y0, v0],
                             [x1, y1, v1],
                             ...
                             [xn, yn, vn]]
                    Example:
                        waypoints[2][1]:
                        Returns the 3rd waypoint's y position

                        waypoints[5]:
                        Returns [x5, y5, v5] (6th waypoint)

    Controller Output Variables:
        throttle_output : Throttle output (0 to 1)
        steer_output    : Steer output (-1.22 rad to 1.22 rad)
        brake_output    : Brake output (0 to 1)
"""

######################################################
######################################################
# MODULE 7: IMPLEMENTATION OF LONGITUDINAL CONTROLLER HERE
######################################################
######################################################
"""
    Implement a longitudinal controller here. Remember that you can
    access the persistent variables declared above here. For
    example, can treat self.vars.v_previous like a "global variable".
"""


# Change these outputs with the longitudinal controller. Note that
# brake_output is optional and is not required to pass the
# assignment, as the car will naturally slow down over time.
Kp = 2.0
Ki = 0.1
Kd = 2.0
e = v_desired - v

P = Kp * e
I = self.vars.I + Ki * e * (t - self.vars.t_previous)
D = Kd * (e - self.vars.e_previous)/(t-self.vars.t_previous)

u = P + I + D
```

```
        if u >= 0:
            throttle_output = u
            brake_output = 0
        else:
            throttle_output = 0
            brake_output    = -u

        ######################################################
        ######################################################
        # MODULE 7: IMPLEMENTATION OF LATERAL CONTROLLER HERE
        ######################################################
        ######################################################
        """
            Implement a lateral controller here. Remember that you can
            access the persistent variables declared above here. For
            example, can treat self.vars.v_previous like a "global variable".
        """
        # Stanley params
        Ke = 0.4
        Kv = 10

        # Compute the heading error
        x_prev = waypoints[0][0]
        y_prev = waypoints[0][1]
        x_next = waypoints[-1][0]
        y_next = waypoints[-1][0]
        yaw_path = np.arctan2(y_next-y_prev, x_next-x_prev)
        heading_error = yaw_path - yaw

        ## make sure the difference is between [-PI, PI]
        if heading_error > np.pi:
            heading_error -= 2 * np.pi
        if heading_error < - np.pi:
            heading_error += 2 * np.pi

        # Compute the crosstrack error
        current_loc = np.array([x, y])
        crosstrack_error = np.min(np.sum((current_loc - np.array(waypoints)[:, :2])**2,
axis=1))
        yaw_cross_track = np.arctan2(y-y_prev, x-x_prev)
        yaw_diff2 = yaw_path - yaw_cross_track
        if yaw_diff2 > np.pi:
            yaw_diff2 -= 2 * np.pi
        if yaw_diff2 < - np.pi:
```

```
            yaw_diff2 += 2 * np.pi
        if yaw_diff2 > 0:
            crosstrack_error = abs(crosstrack_error)
        else:
            crosstrack_error = - abs(crosstrack_error)
        cross_track_steering = np.arctan(Ke * crosstrack_error / (Kv + v))

        # 3. control low
        steer_output = heading_error + cross_track_steering
        if steer_output > np.pi:
            steer_output -= 2 * np.pi
        if steer_output < - np.pi:
            steer_output += 2 * np.pi

        if steer_output < -1.22:
            steer_output = -1.22
        if steer_output > 1.22:
            steer_output = 1.22


        ######################################################
        # SET CONTROLS OUTPUT
        ######################################################
        self.set_throttle(throttle_output)  # in percent (0 to 1)
        self.set_steer(steer_output)        # in rad (-1.22 to 1.22)
        self.set_brake(brake_output)        # in percent (0 to 1)

    ######################################################
    ######################################################
    # MODULE 7: STORE OLD VALUES HERE (ADD MORE IF NECESSARY)
    ######################################################
    ######################################################
    """
        Use this block to store old values (for example, we can store the
        current x, y, and yaw values here using persistent variables for use
        in the next iteration)
    """
    self.vars.v_previous = v  # Store forward speed to be used in next step
    self.vars.e_previous = e
    self.vars.t_previous = t
    self.vars.I = I
```

Result:



- Contribution:

All members got involved in the topic discussion and debugging.