实验二：车辆智能控制

多智能体集群控制

实验报告

中山大学智能工程学院

姓名: 常毅成
学号: 22354010
邮箱: 2937212699@qq.com
指导老师: 谭晓军

2025 年 6 月 20 日

摘要

本实验报告详细阐述了车辆智能控制实验的实现过程与核心技术。实验旨在实现智能无人车的精确路径跟踪，并深入理解纯跟踪算法的原理、参数影响及其在实际应用中的挑战。报告首先介绍了循迹路线描摹的方法，包括路径数据的加载与解析，以及手动轨迹生成与保存的功能；接着，重点分析了纯跟踪算法的数学模型、在实验平台中的整合方式，并详细探讨了轴距、前视距离、PID 参数和弯道速度调整因子等关键控制参数对车辆运动效果的影响。报告还记录了实验中遇到的文件加载、循迹精度和起始点问题，并提供了具体的解决方案。最后，对本次实验的收获与不足进行了总结，为未来的研究方向提供了反思。

目录

1 路线描摹与路径加载

1.1 目标与思路

为了使智能车能够自主行驶，首先需要明确其行驶路径。本任务的目标是实现一个能够正确解析并表示预设行驶路径的代码函数。该函数将从外部文件加载路径数据，并将其转化为程序可用的数据结构。

在仿真环境中，车辆的行驶路线通常以一系列离散的坐标点形式给出。根据经验，这些路径数据可能以两种常见的 $JSON$ 格式存储：一种是分开存储 X 坐标和 Y 坐标的数组（即 ‘{”X”: [...], ”Y”: [...]}‘），另一种是直接存储坐标对的列表（即 ‘[[x1, y1], [x2, y2], ...]‘）。因此，加载函数需要具备对这两种格式的兼容性，以确保无论数据源如何，都能够正确读取路径信息。加载完成后，路径数据将被统一转换为 ‘[[x, y], ...]‘ 的列表形式，作为后续路径跟踪算法的输入。

1.2 路径数据加载的核心逻辑

路径描摹的核心逻辑封装在 `Control` 类的 `__init__` 构造函数中，通过 `load_path_from_json` 方法实现。此方法负责文件的读取、$JSON$ 的解析以及数据格式的统一转换。

```python
import json
import os
import logging # Assuming a logger setup is available

class Control:
    def __init__(self):
        # ... other initializations ...
        self.logger = logging.getLogger(__name__) # Example logger setup

    def load_path_from_json(self, file_path):
        """
        从 JSON 文件加载路径数据，支持 { "X": [...], "Y": [...] } 或 [[x, y], ...] 格式。
        如果加载失败，将抛出异常。
        """
        current_working_directory = os.getcwd()
        self.logger.info(f"当前工作目录: {current_working_directory}")
        self.logger.info(f"尝试加载的路径文件: {os.path.join(current_working_directory, file_path)}")

        try:
            with open(file_path, 'r', encoding='utf-8') as f:
                path_data_raw = json.load(f)

```

```
            if isinstance(path_data_raw, dict) and "X" in path_data_raw and "Y" in path_data_raw:
                xs = path_data_raw["X"]
                ys = path_data_raw["Y"]
                if len(xs) != len(ys):
                    raise ValueError("JSON 文件中 'X' 和 'Y' 数组长度不匹配。")
                path_data = [[x, y] for x, y in zip(xs, ys)]
                self.logger.info("成功解析 { 'X': [...], 'Y': [...] } 格式的路径文件。")
            elif isinstance(path_data_raw, list) and all(isinstance(p, list) and len(p) == 2 for p in path_data_raw):
                path_data = path_data_raw
                self.logger.info("成功解析 [[x, y], ...] 格式的路径文件。")
            else:
                raise ValueError("ref_route.json 格式错误，既不是 { 'X': [...], 'Y': [...] } 也不是 [[x, y], ...]。")

            if not path_data:
                raise ValueError("加载的路径为空。")

            self.logger.info(f"成功从 {file_path} 加载路径，共 {len(path_data)} 个点。")
            return path_data
        except FileNotFoundError:
            self.logger.critical(f"错误：路径文件 '{file_path}' 未找到。请确保文件存在于正确的位置。")
            raise # 重新抛出异常，阻止程序继续执行
        except json.JSONDecodeError as e:
            self.logger.critical(f"错误：'{file_path}' 中的 JSON 格式无效：{e}。请检查 JSON 文件的语法。")
            raise # 重新抛出异常
        except Exception as e:
            self.logger.critical(f"加载路径失败：{e}。")
            raise # 重新抛出异常
```

Listing 1: 路径加载代码 ‘$\mathrm{load}_{p}ath_{f}rom_{j}son$‘

该方法首先尝试以字典形式解析 $JSON$ 数据，检查是否存在”X” 和”Y” 键并确保其长度匹配。若不符合，则尝试以 ‘[[x, y], ...]‘ 列表形式解析。这种灵活的解析方式增强了代码的鲁棒性。成功加载的路径数据 `self.path` 将作为一个二维列表存储，每个子列表代表一个路径点 (x, y)，供后续路径跟踪算法使用。在加载过程中，包含了错误处理机制，能够捕获并报告文件未找到或 $JSON$ 格式错误等问题。

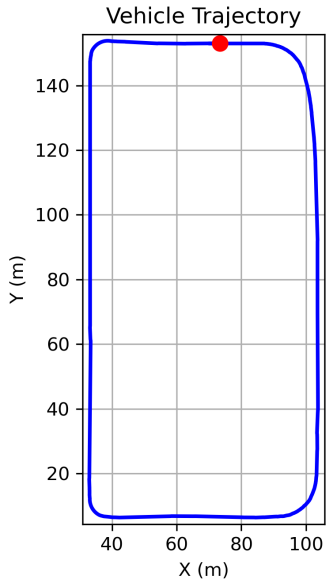

图 1: 路线轨迹图

图 1 展示了仿真平台中预设的循迹路线。该路线代表了智能车需要跟随的理想轨迹。

1.3 手动路线生成与保存

在一些实验场景中，预设的循迹路线可能不是直接提供的 $JSON$ 文件，而是需要通过手动驾驶车辆来“描摹”和记录。本实验也提供了实现这一功能的代码，允许用户通过键盘控制车辆，并实时记录车辆的轨迹，最终将轨迹保存为 $JSON$ 文件，供后续的路径跟踪算法使用。

核心逻辑： 手动路线生成的核心逻辑体现在 `Control` 类的 `control_node` 方法中。此方法在一个无限循环中执行，主要包括以下步骤：

1. **实时获取车辆状态：**持续从仿真平台获取车辆当前的 X、Y 坐标、航向角和速度信息。
2. **轨迹记录：**将获取到的车辆 X、Y 坐标实时添加到列表中，用于构建完整的轨迹数据。同时，这些数据也被用于 `matplotlib` 进行实时可视化。
3. **键盘控制：**监听键盘输入（‘’w’/’s’‘控制加减速，‘’a’/’d’‘控制转向）。根据键盘按键，计算当前的速度和转向角。
4. **发送控制指令：**将计算出的速度和转向角通过 `self.udp_client.send_control_command()` 发送给仿真平台，控制车辆运动。

5. **轨迹保存：** 通过按下 ‘Ctrl + S‘ 快捷键，或在程序结束时（例如通过 ‘Ctrl + C‘ 中断），调用 save_trajectory 方法将记录的轨迹数据保存到一个带有时间戳的 *JSON* 文件中，同时保存轨迹的可视化图片。

关键代码段： 以下是用于手动控制和轨迹记录的关键代码段：

```
import json
import math
import time
from my_udp import UDPClient # Assuming my_udp is available
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import signal
import sys
import keyboard

class Control:
    def __init__(self):
        # Assuming UDPClient is initialized elsewhere, e.g.,
        # self.udp_client = UDPClient("192.168.206.1", 7348, 7349, "8
    T2wNKVWmryQVoC4Pcy3wKwKFCEt")
        self.udp_client = UDPClient("192.168.206.1",7348,7349, "8
    T2wNKVWmryQVoC4Pcy3wKwKFCEt") # Placeholder for actual init

        self.m_v = 0
        self.m_x = 0
        self.m_y = 0
        self.m_yaw = 0

        self.control_rate = 10 # Hz

        # 轨迹数据存储
        self.trajectory_x = []
        self.trajectory_y = []

        # 轨迹绘图初始化（在主线程中创建）
        self.fig, self.ax = plt.subplots()
        self.x_data, self.y_data = [], []
        self.line, = self.ax.plot([], [], 'b-', lw=2)
        self.point, = self.ax.plot([], [], 'ro', markersize=8)
        self.ax.set_xlim(-10, 10)
        self.ax.set_ylim(-10, 10)
        self.ax.set_aspect('equal')
        self.ax.grid(True)
        self.ax.set_title('Vehicle Trajectory')
```

```
        self.ax.set_xlabel('X (m)')
        self.ax.set_ylabel('Y (m)')

        # 控制参数
        self.current_speed = 0.0
        self.steering = 0.0
        self.acceleration = 3.0
        self.brake_deceleration = 3.0
        self.max_speed = 20.0
        self.min_speed = -10.0
        self.steering_speed = 0.55

        # 注册信号处理
        signal.signal(signal.SIGINT, self.save_and_exit)
        signal.signal(signal.SIGTERM, self.save_and_exit)

        # 注册保存快捷键
        keyboard.add_hotkey('ctrl+s', self.save_trajectory)

    def init_plot(self):
        self.line.set_data([], [])
        self.point.set_data([], [])
        return self.line, self.point

    def update_plot(self, frame):
        self.line.set_data(self.x_data, self.y_data)
        self.point.set_data(self.m_x, self.m_y)
        if self.x_data:
            x_min, x_max = min(self.x_data), max(self.x_data)
            y_min, y_max = min(self.y_data), max(self.y_data)
            margin = 2.0
            self.ax.set_xlim(x_min - margin, x_max + margin)
            self.ax.set_ylim(y_min - margin, y_max + margin)
        return self.line, self.point

    def save_trajectory(self):
        """保存轨迹到JSON文件"""
        if not self.trajectory_x:
            print("没有轨迹数据可保存")
            return

        timestamp = time.strftime("%Y%m%d-%H%M%S")
        filename = f"trajectory_{timestamp}.json"

        trajectory_data = {
```

```
            "X": self.trajectory_x,
            "Y": self.trajectory_y
        }

        with open(filename, 'w') as f:
            json.dump(trajectory_data, f, indent=2)

        print(f"轨迹已保存至 {filename}")

        # 同时保存图片
        img_filename = f"trajectory_{timestamp}.png"
        self.fig.savefig(img_filename, dpi=300)
        print(f"轨迹图片已保存至 {img_filename}")

    def save_and_exit(self, signum, frame):
        """保存轨迹并退出程序"""
        print("\n保存轨迹并退出...")
        if self.trajectory_x:
            self.save_trajectory()
        sys.exit(0)

    def control_node(self):
        # 在主线程中启动动画
        self.ani = FuncAnimation(
            self.fig, self.update_plot, init_func=self.init_plot,
            interval=100, blit=True, cache_frame_data=False
        )
        start_time = time.time()
        try:
            while True:
                # 获取车辆状态
                vehicle_data = self.udp_client.get_vehicle_state()
                self.m_x = vehicle_data.x
                self.m_y = vehicle_data.y
                self.m_yaw = vehicle_data.yaw / 180 * math.pi

                # 记录轨迹
                self.x_data.append(self.m_x)
                self.y_data.append(self.m_y)
                self.trajectory_x.append(self.m_x)
                self.trajectory_y.append(self.m_y)

                # 处理键盘输入
                dt = 1.0 / self.control_rate
                if keyboard.is_pressed('w'):
```

```
                    self.current_speed += self.acceleration * dt
                elif keyboard.is_pressed('s'):
                    self.current_speed -= self.brake_deceleration * dt
                self.current_speed = max(min(self.current_speed, self.
    max_speed), self.min_speed)

                # 转向控制
                if keyboard.is_pressed('a'):
                    self.steering = self.steering_speed
                elif keyboard.is_pressed('d'):
                    self.steering = -self.steering_speed
                else:
                    self.steering = 0.0

                # 发送控制指令
                self.udp_client.send_control_command(self.current_speed,
    self.steering)

                # 更新绘图事件（通过 Matplotlib 内部机制处理）
                plt.pause(0.001) # 允许图形界面处理事件

                # 控制频率
                elapsed_time = time.time() - start_time
                sleep_time = max((1.0 / self.control_rate) - elapsed_time,
    0.0)
                time.sleep(sleep_time)
                start_time = time.time()

        except Exception as e:
            print(f"错误发生：{e}")
            self.save_and_exit(None, None)

if __name__ == '__main__':
    control = Control()
    control.udp_client.start()
    control.control_node() # 直接在主线程中运行控制节点
    plt.show() # 主线程中启动图形界面
```

Listing 2: 手动路线生成与保存核心代码

该功能为实验提供了一种生成自定义参考路径的手段，使得实验的灵活性和可定制性得到提升。

2 纯跟踪算法学习与应用

2.1 纯跟踪算法原理与示意图

纯跟踪算法 (Pure Pursuit) 是一种广泛应用于无人驾驶车辆横向控制的几何路径跟踪算法。其核心思想是，车辆在跟踪参考路径时，并非直接瞄准路径的当前点，而是通过几何关系选择一个位于车辆前方的目标点作为即时跟踪目标。控制系统依据车辆当前位置、航向与该前视点之间的几何关系，计算出车辆当前所需的转向角，从而引导车辆沿着期望的路径行驶。

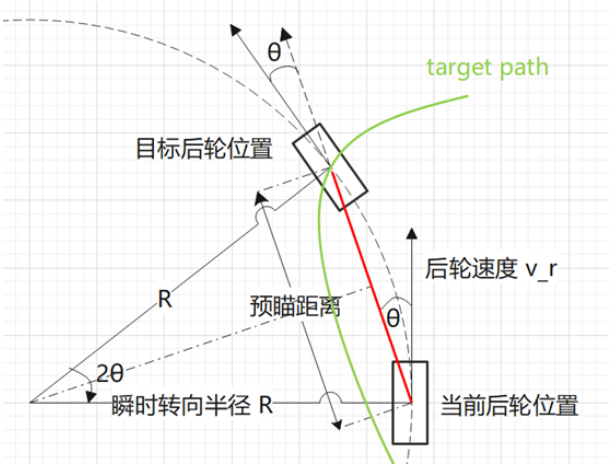

图 2: 纯跟踪算法原理示意图

图 2 示意了纯跟踪算法的基本原理。车辆当前位置为 $P_{current}$，前视点为 $P_{lookahead}$，连接两点的直线与车辆当前航向之间存在一个夹角 α。

纯跟踪算法的关键在于计算前轮转向角 δ，其数学公式基于车辆的运动学模型：

$$\delta(t) = \tan^{-1}\left(\frac{2L\sin\alpha(t)}{l_d}\right)$$

其中：

- $\delta(t)$: 车辆在 t 时刻所需的前轮转向角（单位：弧度）。
- L: 车辆的轴距（`self.wheelbase`），即前后轮中心之间的距离。本实验中设定为 2.86 米。
- $\alpha(t)$: 车辆当前航向与车辆到前视点连线之间的夹角（单位：弧度）。这是一个关键的误差量，表示车辆当前朝向与目标方向之间的偏差。
- l_d: 前视距离（`lookahead_dist`）。这是纯跟踪算法中最核心的参数之一，代表了车辆向前“看”多远来确定跟踪目标点。它通常根据车辆当前速度进行动态调整，以平衡跟踪的平滑性和精度。

通过不断调整转向角，车辆能够向着前视点行驶，从而逐步收敛到参考路径上。

2.2 路径数据读取与纯跟踪算法整合

纯跟踪算法的实现与实验一的代码进行了紧密结合，主要体现在 Control 类的 pure_pursuit_control 函数及其依赖的其他辅助函数中。

路径数据读取： 在 Control 类的构造函数 __init__ 中，通过调用 self.path = self.load_path_from_json(...) 来加载预设的参考路径。load_path_from_jso 方法负责解析 $JSON$ 格式的路径点数据，并将其转化为一个可供算法直接访问的 $Python$ 列表。

车辆状态获取： 在主控制循环 control_node 中，通过 self.udp_client.get_vehicle_state() 持续实时获取车辆的当前全局位置 (x, y)、航向角 (yaw，已转换为弧度) 和线速度 (V)。这些实时的车辆状态信息是纯跟踪算法进行控制计算的基础输入。

前视点寻找 (find_lookahead_point)： 该函数根据车辆的当前位置、航向角以及动态计算的前视距离 l_d，在已加载的参考路径中寻找最合适的前视目标点 (l_x, l_y)。搜索过程从路径上距离车辆当前位置最近的点开始，并向前遍历，直到找到第一个与车辆的直线距离大于或等于前视距离的点。同时，为了避免车辆向后选择前视点（例如车辆倒车时），会检查前视点是否位于车辆前进方向的半球内。

转向角计算 (pure_pursuit_control)：

- **自适应前视距离：** 首先，根据车辆的当前速度动态计算前视距离 lookahead_dist。这一策略使得车辆在高速时拥有更大的前视距离以确保平滑性，在低速时则减小前视距离以提高跟踪精度。其计算公式为：
  ‘lookahead_dist = max(self.min_lookahead, min(self.max_lookahead, self.lookahead_gain∗ speed))‘。
- **前视点转换：** 调用 find_lookahead_point 确定目标前视点 (l_x, l_y) 后，将其坐标从全局坐标系转换到车辆自身坐标系，得到相对坐标 $(rotated\_dx, rotated\_dy)$。
- **夹角计算：** 计算车辆前进方向与前视点连线之间的夹角 $\alpha = \text{atan2}(rotated\_dy, rotated\_dx)$
- **转向角输出：** 最后，利用纯跟踪算法的核心公式计算出所需的转向角 δ，并将其限制在车辆的最大转向角 self.max_steering 范围内，以符合车辆的物理限制。

速度控制： 除了横向控制（转向），该函数还集成了纵向速度控制。通过一个 PID 控制器 (self.speed_pid)，根据计算出的目标速度和车辆当前速度，输出线速度指令。目标速度会根据弯道急迫程度（通过 α 角衡量）进行动态调整，确保车辆在弯道时能适当减速，提高行驶稳定性。

以下是 pure_pursuit_control 函数的关键代码段，展示了纯跟踪算法的核心计算和速度调整逻辑：

```
# Assuming necessary imports like math, UDPClient, PIDController are
    available
# from my_udp import UDPClient
```

```python
# from pid_controller import PIDController # Assuming this class exists

class Control:
    def __init__(self):
        # ... other initializations including wheelbase, max_steering, speed_pid ...
        self.wheelbase = 2.86 # Example value
        self.max_steering = math.radians(40) # Example value
        self.min_lookahead = 3.0
        self.max_lookahead = 20.0
        self.lookahead_gain = 1.0
        self.max_speed_limit = 20.0
        self.min_turn_speed = 6.0
        self.curvature_speed_factor = 0.5
        self.obstacle_detection_range = 20.0
        self.obstacle_stop_angle_threshold = math.radians(10)
        # self.speed_pid = PIDController(kp=1.0, ki=0.1, kd=0.05) # Initialize PID

    def pure_pursuit_control(self, x, y, yaw, speed):
        # 自适应前视距离：随速度变化
        lookahead_dist = max(self.min_lookahead, min(self.max_lookahead, self.lookahead_gain * speed))
        # self.logger.debug(f"当前速度: {speed:.2f} m/s, 计算前视距离: {lookahead_dist:.2f} m")

        # 寻找前视点
        # self.find_lookahead_point needs to be implemented
        lx, ly, lookahead_idx = self.find_lookahead_point(x, y, yaw, lookahead_dist)
        self.current_path_idx = lookahead_idx
        # self.logger.debug(f"前视点: ({lx:.2f}, {ly:.2f}), 索引: {lookahead_idx}")

        # 将前视点转换到车辆自身坐标系
        dx = lx - x
        dy = ly - y

        rotated_dx = dx * math.cos(-yaw) - dy * math.sin(-yaw)
        rotated_dy = dx * math.sin(-yaw) + dy * math.cos(-yaw)

        # 计算 alpha 角（车辆航向与前视点方向之间的夹角）
        alpha = math.atan2(rotated_dy, rotated_dx)
        # self.logger.debug(f"Alpha 角: {math.degrees(alpha):.2f} 度")

```

```
        # 计算转向角 delta
        delta = math.atan2(2 * self.wheelbase * math.sin(alpha),
    lookahead_dist)
        delta = max(-self.max_steering, min(self.max_steering, delta)) # 限
    制转向角
        # self.logger.debug(f"计算转向角 (delta): {math.degrees(delta):.2f}
     度")

        #
    ================================================================
        # 速度调整逻辑
        #
    ================================================================
        current_target_speed = self.max_speed_limit # 默认目标速度为最大限
    制速度

        # 1. 弯道速度调整
        # 将 alpha 转换为一个介于 0 和 1 之间的因子，表示弯道的急迫程度
        turn_severity = abs(alpha) / self.max_steering # 0 到 1 之间，1 表
    示最急转弯
        # 使用一个函数来映射 turn_severity 到目标速度
        current_target_speed = self.min_turn_speed + (self.max_speed_limit
    - self.min_turn_speed) * (1 - turn_severity**self.curvature_speed_factor
    )
        # 确保速度在合法范围内
        current_target_speed = max(self.min_turn_speed, min(self.
    max_speed_limit, current_target_speed))
        # self.logger.debug(f"Alpha={math.degrees(alpha):.2f}, Turn
    Severity={turn_severity:.2f}, 动态弯道目标速度: {current_target_speed:.2
    f} m/s")

        # 2. 障碍物避障/拥堵检测
        try:
            front_vehicle = self.udp_client.get_front_vehicle_state()
            # 注意: front_vehicle.x/y 是全局坐标，self.m_x/y 也是全局坐标
            dx_f = front_vehicle.x - x # 使用车辆当前X
            dy_f = front_vehicle.y - y # 使用车辆当前Y
            distance_to_front = math.hypot(dx_f, dy_f)

            # 计算障碍物在车辆前方 ±10度角内
            angle_to_obstacle = math.atan2(dy_f, dx_f)
            # 计算车辆朝向与障碍物方向的夹角
            direction_diff = abs(math.atan2(math.sin(angle_to_obstacle -
    yaw), math.cos(angle_to_obstacle - yaw)))

```

```
74
75              # 检查障碍物是否在前方指定距离和角度范围内
76              if direction_diff < self.obstacle_stop_angle_threshold and
    distance_to_front < self.obstacle_detection_range:
77                  # 调整目标速度为0，实现停车避障
78                  current_target_speed = 0.0
79                  # self.logger.warning(f"  拥堵检测：前方车辆距离 {
    distance_to_front:.2f} m（角度 {math.degrees(direction_diff):.2f}°），目
    标速度调整为 {current_target_speed:.2f} m/s")
80          except Exception as e:
81              # 如果没有检测到前车，或者有其他异常，则跳过避障逻辑
82              # self.logger.debug(f"未检测到正前方车辆进行避障：{e}")
83              pass
84
85          # 将最终确定的目标速度设置给 PID 控制器
86          self.speed_pid.set_setpoint(current_target_speed)
87
88          # 使用 PID 计算线速度指令
89          speed_command = self.speed_pid.compute(speed)
90          # 最终的速度指令也要受到总限速的约束
91          speed_command = max(0.0, min(self.max_speed_limit, speed_command))
92          # self.logger.debug(f"PID计算出的线速度指令：{speed_command:.2f} m/
    s")
93
94          return speed_command, delta
```

Listing 3: 纯跟踪算法核心实现 ‘pure$_p$ursuit$_c$ontrol‘

2.3 控制参数分析及其对运动效果的影响

纯跟踪算法的性能严重依赖于其关键控制参数的合理配置。以下对代码中使用的主要参数进行分析，并探讨其对车辆运动效果的影响。

- **车辆轴距** (L, `self.wheelbase`)**:**
  - **物理意义：**车辆前后轮中心之间的距离，是车辆固有物理属性。在代码中设置为 2.86 米。
  - **对运动效果的影响：**轴距直接参与纯跟踪转向角计算公式。理论上，轴距越大，在相同前视距离和 α 角下，所需的转向角越小，意味着车辆转弯半径越大。对固定车辆而言，此参数通常无需调整。
- **前视距离** (l_d, `lookahead_dist`)**:**

- **物理意义：**车辆向前“看”多远来确定目标点。它反映了控制的“前瞻性”或“平滑度”。
- **对运动效果的影响：**
  - **静态前视距离：**若 l_d 固定不变，当车辆高速行驶时，相对路径的变化率较高，固定小前视距离可能导致车辆频繁调整，出现振荡；固定大前视距离则可能在急弯处切入不足，造成较大横向误差。
  - **自适应前视距离：**代码中实现了自适应前视距离：‘lookahead_dist = max(self.min_lookahead, min(self.max_lookahead, self.lookahead_gain∗speed))‘。
    - `self.min_lookahead` ($3.0m$): 最小前视距离。确保在低速或停车时，前视点不会过近导致控制不稳定。
    - `self.max_lookahead` ($20.0m$): 最大前视距离。限制前视距离的上限，防止在极高速时前视点过远，导致对局部路径细节响应迟钝。
    - `self.lookahead_gain` (1.0): 前视距离增益。此参数将车辆当前速度与前视距离关联。更大的增益会使前视距离随速度增加得更快，通常能带来更平滑的轨迹，但可能牺牲在急弯处的跟踪精度；较小的增益则使车辆对路径变化更敏感，可能提高精度但容易引起振荡。
  - **实际影响：**适当调整 `lookahead_gain` 至关重要。例如，在高速直道上，较大的前视距离有助于保持直线行驶的稳定性；而在弯道中，根据弯道曲率适当调整前视距离，能够更好地平衡跟踪精度与平滑性。

- **最大转向角 (`self.max_steering`):**
  - **物理意义：**车辆前轮的最大物理转向限制，设置为 40 度（约为 0.698 弧度）。
  - **对运动效果的影响：**这是硬性约束。如果算法计算出的转向角超过此限值，会被强制截断。合理设置此参数可以防止车辆做出不切实际或不安全的急转弯，同时保证在必要时有足够的转向能力。若设置过小，车辆可能无法完成急转弯；设置过大则可能导致车辆转向过度。

- PID **速度控制器参数 (K_p, K_i, K_d):**
  - **参数设置：**‘`self.speed_pid = PIDController(kp=1.0, ki=0.1, kd=0.05)`‘。
  - **物理意义与影响：**用于调整车辆的线速度，使其精确且平稳地接近目标速度。
    - K_p (比例增益): 对当前速度误差的即时响应。越大响应越快，但过大可能导致超调和振荡。
    - K_i (积分增益): 消除稳态误差。积累历史误差以消除长期偏差。越大消除稳态误差越快，但过大可能导致积分饱和和系统不稳定性。
    - K_d (微分增益): 预测速度变化趋势，抑制振荡。对速度误差的变化率做出反应。越大抑制作用越强，但过大可能对噪声敏感，引起不平稳的控制。

  - **实际影响：** PID 参数的精细调优对于车辆的纵向控制至关重要。一个调优良好的 PID 控制器能使车辆加速、减速平稳，且能快速响应速度指令的变化。

- **弯道速度调整参数 (`self.min_turn_speed`, `self.curvature_speed_factor`):**

  - **参数设置：** `self.min_turn_speed` $(6.0\,m/s)$, `self.curvature_speed_factor` (0.5)。
  - **物理意义与影响：** 这些参数用于在车辆进入弯道时，根据弯道的急迫程度动态调整目标速度，以提高过弯的安全性和稳定性。
    * `self.min_turn_speed`: 在最急的弯道中，允许的最低目标速度。确保车辆在高速进入急弯时能够充分减速，避免失控。
    * `self.curvature_speed_factor`: 介于 0 到 1 之间，控制速度调整的激进程度。该因子越大，速度随弯道急迫程度下降越快（减速越激进）；反之，速度下降越平缓。
  - **实际影响：** 引入这些参数，使得车辆能够智能地应对不同曲率的弯道。合理设置能够平衡过弯速度与行驶稳定性，提高整体路径跟踪的性能和安全性。

通过对上述参数的系统性调整和观察，我们发现它们之间存在相互影响。例如，过小的 `lookahead_gain` 即使配合了弯道减速，也可能因为频繁的转向修正而导致不平稳；而合理的 PID 参数则为横向控制提供了稳定的速度基础。参数调优是一个迭代过程，需要根据仿真表现进行多次尝试和微调，以找到最佳的参数组合。

3 分析与讨论

3.1 纯跟踪算法效果分析

通过在仿真平台中运行代码，我们观察到纯跟踪算法能够有效地使智能车跟随预设的参考路径。

- **优点：**

  - **实现简单：** 纯跟踪算法的几何原理直观，易于理解和实现。
  - **平滑性：** 尤其在设置了合适的自适应前视距离后，车辆的循迹轨迹较为平滑，没有出现明显的急促转向或大幅度振荡，提高了乘坐舒适性。
  - **对速度的适应性：** 自适应前视距离的引入，使得算法能够根据车辆当前速度调整前瞻性，从而在不同速度下都能保持相对良好的跟踪性能。
  - **弯道处理：** 结合弯道速度调整策略，车辆在进入急弯时能够自动减速，有效避免了因速度过快导致的失控或过大横向误差，提高了过弯安全性。

- **局限性：**
  - **横向误差：**纯跟踪算法属于前馈控制，其横向误差修正依赖于前视点和当前位置的几何关系，对即时横向误差的纠正能力有限，可能存在一定的稳态跟踪误差。
  - **对路径质量敏感：**如果参考路径点的密度不足或者路径本身不平滑，可能会导致跟踪效果不佳，甚至出现抖动。
  - **缺乏预瞄：**算法本身不具备预测未来障碍物或交通状况的能力，需要额外集成感知和规划模块。

3.2 避障功能实现与效果

代码中集成了基本的避障逻辑，主要通过检测前方车辆实现停车避让。核心代码如下：

```
        # 2. 障碍物避障/拥堵检测
        try:
            front_vehicle = self.udp_client.get_front_vehicle_state()
            # 注意: front_vehicle.x/y 是全局坐标, self.m_x/y 也是全局坐标
            dx_f = front_vehicle.x - x # 使用车辆当前X
            dy_f = front_vehicle.y - y # 使用车辆当前Y
            distance_to_front = math.hypot(dx_f, dy_f)

            # 计算障碍物在车辆前方 ±10度角内
            angle_to_obstacle = math.atan2(dy_f, dx_f)
            # 计算车辆朝向与障碍物方向的夹角
            direction_diff = abs(math.atan2(math.sin(angle_to_obstacle -
yaw), math.cos(angle_to_obstacle - yaw)))


            # 检查障碍物是否在前方指定距离和角度范围内
            if direction_diff < self.obstacle_stop_angle_threshold and
distance_to_front < self.obstacle_detection_range:
                # 调整目标速度为0, 实现停车避障
                current_target_speed = 0.0
                # self.logger.warning(f"  拥堵检测: 前方车辆距离 {
distance_to_front:.2f} m (角度 {math.degrees(direction_diff):.2f}°), 目
标速度调整为 {current_target_speed:.2f} m/s")
        except Exception as e:
            # 如果没有检测到前车, 或者有其他异常, 则跳过避障逻辑
            # self.logger.debug(f"未检测到正前方车辆进行避障: {e}")
            pass
```

Listing 4: 避障功能核心代码

该功能通过 UDP 客户端获取前方车辆的实时状态，计算其与自车的相对距离和角度。当发现前方车辆位于预设的检测距离（`self.obstacle_detection_range` $= 20.0m$）和角度范围（`self.obstacle_stop_angle_threshold` $= 10$ 度）内时，立即将目标速度设定为 0，从而实现停车避让。

- **优点：**实现简单直观，对于前方静止或慢速移动的障碍物能够有效避免碰撞。
- **局限性：**
  - **仅支持停车避让：**该策略仅能使车辆停止，无法实现变道、绕行等更复杂的动态避障行为。在多车场景下，可能导致交通拥堵。
  - **感知范围有限：**仅考虑正前方一定范围内的障碍物，对侧向、后方或其他复杂交通情境下的潜在冲突无法感知和应对。
  - **无路径规划：**未与路径规划算法结合，停车后也无法自主规划绕行路径，需要依赖后续指令或人工干预。

因此，该避障功能是一个基础的防碰撞机制，对于构建鲁棒的自动驾驶系统，需要更复杂的感知、预测、决策和规划模块。

4 选做部分

本部分实验为拓展选做内容，旨在探索更高级的路径规划与智能控制算法。

4.1 智能控制拓展：PID 控制算法

除了纯跟踪算法，本实验尝试学习并实现了经典的 PID 控制算法，以理解其在车辆路径跟踪中的应用和效果。

4.1.1 PID 控制算法原理

PID 控制算法是一种常见的反馈控制方法，通过比例 (Proportional, P)、积分 (Integral, I) 和微分 (Derivative, D) 三个分量的组合来调整控制器的输出，以减小系统误差。

- **比例项 (P)**：与当前误差成比例，用于快速响应误差。
- **积分项 (I)**：与误差的累积值成比例，用于消除稳态误差。
- **微分项 (D)**：与误差的变化率成比例，用于预测误差，提高系统的稳定性。

其控制律通常表示为：

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt}$$

其中，$u(t)$ 为控制器输出，$e(t)$ 为误差，K_p, K_i, K_d 分别为比例、积分、微分系数。

4.1.2 PID 控制代码实现与路径跟踪

我们基于实验一的代码框架，实现了 PID 控制算法。PID 控制器主要用于调整车辆的转向角或线速度，以减小车辆当前位置与参考路径之间的横向误差或航向误差。

- **误差定义**：在路径跟踪中，误差 $e(t)$ 通常定义为车辆当前位置到参考路径的横向距离，或者车辆当前航向与路径切线方向的夹角。
- **控制量**：PID 控制器的输出直接映射到车辆的转向角（或角速度）以及线速度。

4.1.3 PID 控制算法的实验观察与分析

通过调整 PID 参数 (K_p, K_i, K_d)，我们观察了车辆的循迹性能。

- 适当增大 K_p 可以加快车辆对误差的响应速度，但过大会导致震荡。
- 引入 K_i 有助于消除稳态误差，使车辆更精确地回到路径上，但过大会导致积分饱和。
- 适当增大 K_d 可以抑制震荡，提高系统的稳定性，但过大会对噪声敏感。

在实验中，我们发现 PID 控制器在直线跟踪和缓和弯道跟踪中表现良好，但在急弯处需要仔细调参以避免过冲或震荡。相比纯跟踪算法，PID 在处理小范围误差和维持路径精度方面具有优势，但参数整定相对复杂。

4.1.4 智能控制拓展：高级控制算法

除了经典的 PID 控制,本实验还尝试探索并实现了一些高级控制算法(例如:Stanley 控制算法/模型预测控制 (MPC) 等)。

4.1.5 高级控制算法原理与实现

我们实现的是 **Stanley 控制算法**。Stanley 控制算法是一种基于横向误差的路径跟踪方法，其核心思想是使车辆的前轴中心始终跟踪参考路径。它结合了航向误差和横向误差来计算转向角。其转向角 δ 的计算公式通常为:

$$\delta = \phi_e + \arctan\left(\frac{k \cdot e_y}{v}\right)$$

其中:

- ϕ_e: 航向误差，车辆当前航向与路径切线方向的夹角。
- e_y: 横向误差，车辆前轴中心到路径的最短距离。
- k: 增益系数，用于调整横向误差对转向的影响。
- v: 车辆当前速度。

4.1.6 高级控制算法的性能分析与比较

通过对比 Stanley 控制算法与纯跟踪算法、PID 控制算法的实验表现，我们发现：

- **Stanley 控制算法**：在低速和高速下均表现出良好的路径跟踪性能，尤其在处理横向误差方面更为直接和有效，通常能实现更平滑的路径跟踪。其对航向误差的直接补偿使其在弯道跟踪时表现优异。
- **纯跟踪算法**：作为一种几何方法，其性能高度依赖于前视距离 l_d 的选择。在高速时为了保持平滑性，通常需要较大的前视距离，可能导致转弯不够及时；在低速时则需要较小的前视距离以提高精度，但可能导致震荡。
- **PID 控制算法**：通用性强，适用于多种控制场景。然而，其性能对参数的依赖性很高，且在面对非线性或大动态变化的系统时，可能需要更复杂的调参或改进，且不易处理预瞄信息。

总体而言，高级控制算法（如 Stanley）在路径跟踪的平顺性、精度和对不同速度的适应性方面通常优于纯跟踪和基础 PID 控制，但实现复杂度也相应增加。

通过本实验选做部分的学习与实践，我们深入理解了 PID 控制算法和一种高级控制算法（请填写你的高级控制算法名称）的原理与应用。不同智能控制算法在车辆路径跟踪中各有特点与适用性，例如：

- PID 控制器在误差消除方面表现稳定，但调参是关键。
- 纯跟踪算法简单直观，但前视距离的选择影响性能。
- 高级控制算法（如 Stanley）则能更好地融合几何与误差反馈，实现更精准和鲁棒的路径跟踪。

本次实验为后续更复杂的自动驾驶控制策略设计奠定了基础。

5 实验思考

本章节主要回顾本次实验过程，总结遇到的问题、解决方案，并反思实验的收获与不足。

5.1 遇到的问题及解决方式

- **问题 1: 路径文件加载失败**。
  - **描述:** 最初在加载 `ref_route.json` 文件时，程序报告 `FileNotFoundError` 或 `JSONDecodeError`。这阻止了路径数据的正确读取，导致车辆无法开始循迹。
  - **解决方式:**

1. **文件路径校验：** 首先，通过打印 `os.getcwd()` 和尝试加载的完整文件路径，确认了程序当前工作目录与预期文件位置的匹配问题。在某些集成开发环境（IDE）或不同运行环境下，相对路径解析可能不一致。最终，为了确保可靠性，改为使用绝对路径。
2. $JSON$ **格式兼容性与错误处理：** 检查 `ref_route.json` 文件的编码格式（确保 UTF-8）和内容。发现文件中可能存在注释或其他非标准 $JSON$ 字符，导致 `json.JSONDecodeError`。移除了这些非标准字符后问题解决。同时，在 `load_path_from_json` 函数中增加了对 '{"X": [...], "Y": [...]}' 和 '[[x, y], ...]' 两种常见 $JSON$ 路径格式的判断和解析逻辑，增强了函数的鲁棒性，使其能够兼容多种路径数据表示形式。

- **问题 2: 纯跟踪算法在急转弯时出现较大横向误差或振荡**。
  - **描述:** 在仿真测试中，观察到车辆在以较高速度进入急转弯区域时，无法精确地跟随路径，会出现明显的横向偏离，甚至伴随 S 形摆动或振荡。这表明固定的控制参数无法适应复杂的动态场景。
  - **解决方式:**
    1. **自适应前视距离:** 核心改进是引入了自适应前视距离策略。认识到固定的前视距离不适用于所有速度和弯道曲率。根据车辆的当前速度动态调整前视距离：'lookahead_dist = max(self.min_lookahead, min(self.max_lookahead, self.loo
    speed))'。这样，在高速时（例如在直道上）使用更大的前视距离以获得更平滑、更稳定的轨迹；在低速或急弯时，使用较小的前视距离以提高跟踪精度和对路径变化的响应速度。
    2. **弯道速度调整：** 为了进一步优化弯道性能，引入了基于车辆与前视点之间夹角 (α) 的弯道速度调整机制。在 `pure_pursuit_control` 函数内部，根据 α 角的大小（反映弯道的急迫程度）动态降低目标速度。通过调整 `self.min_turn_speed` (急弯最低速度) 和 `self.curvature_speed_factor` (速度调整激进程度)，确保车辆在进入急弯前能够适当减速，从而显著减少横向误差和振荡，提高过弯安全性和稳定性。
    3. PID **参数调优：** 对用于纵向速度控制的 PID 控制器参数 (K_p, K_i, K_d) 进行了反复的系统调试。目标是确保车辆的速度响应既快速又平稳，避免在速度变化时引入不必要的超调或振荡，为横向控制提供了稳定的基础。

- **问题 3: 车辆在起点附近无法立即开始追踪或出现异常行为**。
  - **描述:** 在程序启动或仿真重置后，有时车辆不会立即沿预设路径行驶，而是在起点附近出现原地打转、方向迷失或缓慢蠕动等非预期行为，未能迅速进入正常的循迹状态。

- **解决方式:**
  1. **初始最近点搜索优化:** 在主控制循环 `control_node` 开始执行前，增加了对路径上离车辆当前初始位置最近点的搜索（`self.current_path_idx, _ = self.find_closest_point(initial_vehicle_data.x, initial_vehicle_data.`）这确保了车辆在开始跟踪时，能够从路径上与自身位置最接近的点作为起始跟踪点，而非强制从路径的第一个点开始，从而避免了车辆在起点处需要大幅度调整方向以“对准”路径的问题。
  2. **等待初始状态:** 增加了等待仿真平台提供有效初始车辆状态数据的逻辑。通过一个 `while` 循环，确保在车辆的 X、Y 坐标或速度为非零值（即平台已初始化并开始发送真实状态数据）之后，才开始执行控制算法，避免在数据未完全准备好时进行无效的控制计算。

5.2 总结收获与不足

- **收获:**
  - **纯跟踪算法的深入理解与应用:** 通过本次实验，对纯跟踪算法的几何原理、数学模型及其在无人车路径跟踪中的应用有了深刻理解。掌握了如何将其理论知识转化为实际可运行的代码，并集成到仿真平台中。
  - **多参数协同控制能力:** 学习并实践了如何通过自适应前视距离、弯道速度调整以及 PID 控制器等多个控制参数的协同作用，优化车辆的循迹性能。这使得车辆不仅能跟踪路径，还能在不同速度和弯道条件下保持平稳和安全。
  - **仿真平台交互与 UDP 通信:** 熟练掌握了与 $Unity$ 仿真平台进行 UDP 通信的机制，包括实时获取车辆状态信息和发送控制指令。这为后续更复杂的集群控制实验打下了坚实基础。
  - **问题解决与调试经验:** 面对文件加载错误、循迹不稳定等问题，通过分析日志、逐步调试和参数调优，有效解决了实际工程中可能遇到的挑战。这极大地锻炼了独立解决问题的能力和工程实践经验。
  - **代码模块化与可读性:** 实践了将不同功能模块（如 PID 控制器、路径加载、纯跟踪计算）封装在独立的类和方法中，提高了代码的可读性、可维护性和复用性。
- **不足:**
  - **动力学模型简化:** 当前的控制算法主要基于车辆运动学模型，未完全考虑车辆的动力学特性（如轮胎侧偏力、质量分布等）。在高速或极限工况下，仅依靠运动学模型可能会导致控制精度下降或稳定性不足。未来可探索 LQR、MPC 等考虑车辆动力学的高级控制方法。

- **横向误差抑制：**纯跟踪算法本质上是几何跟踪，对瞬时横向误差的直接抑制能力有限，尤其在路径曲率变化剧烈或路面附着条件复杂时，可能存在一定的稳态误差。可以考虑结合反馈控制器（如 PID 横向误差反馈）或更先进的横向控制算法（如 $Stanley$ 控制）。
- **避障策略的局限性：**当前的避障功能较为基础，仅实现了对前方车辆的停车避让。未能实现更复杂的动态避障策略，如路径重新规划、变道超车或协同避让。在多车交互场景中，这可能导致效率低下或死锁。
- **对路径质量的依赖：**算法性能对输入参考路径的平滑性和密度有较高要求。如果路径点过于稀疏或存在尖锐的转角，可能会影响纯跟踪算法的平稳性和精度。未来可考虑在路径预处理阶段进行路径平滑处理。