实验三：联机多车交互与协同运动控制

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多智能体集群控制中车辆交互与冲突解决的实现过程与核心技术。实验旨在理解和解决交通场景中可能出现的多车冲突情况，并探索不同的车辆应对策略。报告首先描述了实验目标与背景，强调了多智能体系统在智能交通中的重要性。接着，深入分析了常见的交通冲突场景，如十字路口冲突、车道汇入冲突和编队行驶冲突，并说明了这些冲突在实验平台中如何通过多辆智能车的协作与对抗进行仿真。报告重点介绍了针对这些冲突场景的车辆应对策略，包括基于规则的优先级策略、协作式决策方法以及动态避障路径规划，并给出了相应的实现思路。此外，报告还详细阐述了用于获取全局车辆信息的函数结构及其数据组织方式，并提供了实时数据输出示例。最后，总结了实验过程中遇到的问题及解决方式，并反思了实验的收获与不足，为未来复杂交通场景下的多智能体控制研究提供了实践经验。

目录

1 实验目标与背景

1.1 实验目标

本次实验的核心目标是深入理解多智能体环境下车辆之间的交互行为，并针对交通场景中可能出现的冲突情况，设计并实现有效的冲突解决策略。具体目标包括：

1. 掌握多智能体系统的基本概念，理解车辆间信息交互的重要性。
2. 识别和分析交通场景中常见的多车冲突类型，例如路口交叉、车道汇入、超车、编队行驶等。
3. 探索在仿真平台中实现多车冲突仿真的方法，包括多辆智能车的建模与控制。
4. 设计并实现针对不同冲突场景的车辆应对策略，包括但不限于基于优先级的避让、协作式决策、以及动态路径调整等。
5. 评估所设计策略在保证交通流畅性与安全性的有效性。
6. 记录实验过程中遇到的问题与挑战，并提出相应的解决方案。

通过本实验，旨在提升学生在复杂交通环境下多智能体协同控制算法的设计与实现能力。

1.2 实验背景

随着自动驾驶技术的快速发展，(Autonomous Driving)，未来的交通系统将越来越多地由多个智能体，(Multi-Agent System)，即智能车辆组成。这些智能车辆在共享道路资源时，不可避免地会发生交互，甚至产生冲突。例如，在十字路口，多辆车辆可能同时到达交叉点；在高速公路上，车辆需要进行车道变换或汇入主路；在编队行驶中，(Platooning)，车辆之间需要保持安全距离并协同运动。

传统的人工驾驶环境下，司机通过观察、预测和经验来解决冲突。然而，在智能交通系统，(Intelligent Traffic System, ITS)，中，需要设计一套智能、高效、安全的机制来自动化地处理这些冲突。这不仅涉及到单车智能，(Single-Vehicle Intelligence)，如路径跟踪和避障，更关键的是多车之间的协同决策和控制。如何使智能车辆在保证自身安全的同时，提高整体交通效率，是当前智能交通领域面临的重要挑战。本实验旨在通过仿真平台，模拟真实交通场景，为解决这些问题提供实践经验和技术基础。

2 全局状态获取函数 get_global_state() 的实现与数据结构

为了实现多智能体间的协作与冲突解决，每辆智能车需要实时获取整个仿真环境中所有车辆的全局状态信息。这通常通过一个专门的函数来实现，该函数负责从仿真平台接收并解析全局数据包。

函数结构与调用方式： 在实验框架中，通常会有一个 *UDP* 客户端 ‘*UDPClient*‘ 负责与仿真服务器进行通信。‘get$_g$*lobal*$_s$*tate*()‘ ‘UDPClient‘

```python
import json
import socket
from collections import namedtuple

# 定义车辆状态数据结构，便于访问
VehicleState = namedtuple('VehicleState', ['id', 'x', 'y', 'yaw', 'speed',
    'timestamp'])
# 定义全局状态数据结构，包含所有车辆信息
GlobalState = namedtuple('GlobalState', ['timestamp', 'vehicles']) #
    vehicles 是一个字典，键为id，值为VehicleState

class UDPClient:
    def __init__(self, server_ip, send_port, receive_port, api_key):
        self.server_ip = server_ip
        self.send_port = send_port
        self.receive_port = receive_port
        self.api_key = api_key
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sock.bind(('', self.receive_port)) # 绑定接收端口
        self.sock.settimeout(0.01) # 非阻塞读取

        self.global_data = GlobalState(timestamp=0, vehicles={}) # 存储最新
    全局数据

    def start(self):
        # 启动一个独立的线程来持续接收数据
        import threading
        self.receive_thread = threading.Thread(target=self.
    _receive_data_loop)
        self.receive_thread.daemon = True # 设置为守护线程，主程序退出时自
    动关闭
        self.receive_thread.start()

```

```
    def _receive_data_loop(self):
        while True:
            try:
                data, _ = self.sock.recvfrom(4096) # 接收数据
                decoded_data = data.decode('utf-8')
                parsed_json = json.loads(decoded_data)
                self._process_global_state_data(parsed_json)
            except socket.timeout:
                pass # 没有数据可读
            except json.JSONDecodeError:
                print("Error decoding JSON from UDP.")
            except Exception as e:
                print(f"UDP receive error: {e}")

    def _process_global_state_data(self, data):
        # 假设服务器发送的全局状态包含一个 'timestamp' 和 'vehicles' 列表
        current_timestamp = data.get('timestamp', 0)
        vehicles_list = data.get('vehicles', [])

        new_vehicles_dict = {}
        for v_data in vehicles_list:
            try:
                vehicle_id = str(v_data.get('id'))
                x = float(v_data.get('x'))
                y = float(v_data.get('y'))
                yaw = float(v_data.get('yaw')) # 假设为度数
                speed = float(v_data.get('speed'))
                new_vehicles_dict[vehicle_id] = VehicleState(vehicle_id, x,
    y, yaw, speed, current_timestamp)
            except (ValueError, TypeError) as e:
                print(f"Malformed vehicle data received: {v_data}, error: {
    e}")
                continue

        # 更新全局数据
        self.global_data = GlobalState(timestamp=current_timestamp,
    vehicles=new_vehicles_dict)

    def get_global_state(self):
        """
        返回最新的全局车辆状态数据。
        """
        return self.global_data

# 调用方式示例:
```

```
# udp_client = UDPClient("192.168.206.1", 7348, 7349, "YOUR_API_KEY")
# udp_client.start()
# while True:
#     global_state = udp_client.get_global_state()
#     # 遍历所有车辆信息
#     for vehicle_id, vehicle_data in global_state.vehicles.items():
#         print(f"Vehicle {vehicle_id}: X={vehicle_data.x:.2f}, Y={
    vehicle_data.y:.2f}, Yaw={vehicle_data.yaw:.2f}, Speed={vehicle_data.
    speed:.2f}")
#     time.sleep(0.1) # 控制查询频率
```

Listing 1: get$_g lobal_s tate()$

变量 *global__data* **中存储的数据结构：** ‘global$_d ata$‘

类型：通常是一个自定义的数据结构（例如，‘namedtuple‘ 或一个 ‘dataclass‘），它包含两部分关键信息：

1. ‘*timestamp*‘：一个浮点数，表示当前全局状态数据的时间戳，用于判断数据的实时性。
2. ‘*vehicles*‘：一个字典，其键是字符串形式的车辆 ID（例如：”vehicle_0”, ”vehicle_1”），其值是另一个数据结构，代表单辆车的详细状态。

单辆车状态结构：字典中每个键值对的值（即每辆车的数据）通常包含以下信息：

- ‘*id*‘：车辆的唯一标识符（字符串）。
- ‘x‘：车辆在全局坐标系中的 X 坐标（浮点数，单位：米）。
- ‘y‘：车辆在全局坐标系中的 Y 坐标（浮点数，单位：米）。
- ‘*yaw*‘：车辆的航向角（浮点数，单位：度或弧度）。
- ‘*speed*‘：车辆的当前线速度（浮点数，单位：米/秒）。
- ‘*timestamp*‘：（可选）该辆车数据的时间戳，可以与全局时间戳保持一致。

这种嵌套的字典结构允许通过车辆 ID 快速检索任何特定车辆的最新状态，为多智能体间的感知和决策提供了便利。

终端中打印的实时车辆信息示例输出： 以下是模拟在终端中持续打印实时车辆状态的输出示例。实际输出的格式和精度可能因具体实现和仿真环境而异。

```
--- Global State Update ---
Timestamp: 1678886400.123
Vehicle vehicle_0: X=10.50, Y=5.20, Yaw=45.00, Speed=8.25
Vehicle vehicle_1: X=12.10, Y=4.80, Yaw=42.00, Speed=7.90
```

```
Vehicle vehicle_2: X=15.00, Y=3.50, Yaw=38.00, Speed=9.10
--- Global State Update ---
Timestamp: 1678886400.223
Vehicle vehicle_0: X=11.30, Y=5.35, Yaw=44.50, Speed=8.30
Vehicle vehicle_1: X=13.00, Y=4.95, Yaw=41.50, Speed=7.95
Vehicle vehicle_2: X=16.10, Y=3.65, Yaw=37.50, Speed=9.15
--- Global State Update ---
Timestamp: 1678886400.323
Vehicle vehicle_0: X=12.10, Y=5.50, Yaw=44.00, Speed=8.35
Vehicle vehicle_1: X=13.90, Y=5.10, Yaw=41.00, Speed=8.00
Vehicle vehicle_2: X=17.20, Y=3.80, Yaw=37.00, Speed=9.20
... (continues)
```

这样的输出可以帮助开发者实时监控所有智能体的运动状态，从而进行调试和性能分析。

3 交通场景中多车冲突的描述与仿真实现

3.1 多车冲突情况描述与分析

在复杂的交通环境中，多辆智能车在有限的道路资源上行驶时，可能会出现多种形式的冲突。这些冲突可以根据发生位置、参与车辆数量和冲突性质进行分类。以下是几种典型的多车冲突情况及其分析：

1. **十字路口冲突** (**Intersection Conflict**)

   - **描述：** 当多辆智能车同时接近或进入无交通信号灯控制的十字路口时，它们的预期路径可能在路口中心区域发生交叉。常见的冲突场景包括：
     - 直行车辆之间的冲突：多辆直行车从不同方向同时进入路口。
     - 直行与转弯车辆的冲突：直行车与左转或右转车辆在路口发生路径交叉。
     - 左转车辆之间的冲突：两辆对向左转车辆在路口中心点发生冲突。
   - **分析：** 路口冲突的核心是共享空间的竞争。需要确定明确的优先级规则或进行实时协商，以避免碰撞并保证通行效率。车辆需要感知其他车辆的位置、速度、预期路径，并据此做出停车、减速或加速通过的决策。

2. **车道汇入冲突** (**Lane Merging Conflict**)

- **描述：**在高速公路入口匝道或多车道汇合点，一辆或多辆车辆需要从一个车道汇入另一个车道。冲突发生在汇入车辆与目标车道上的车辆之间。
- **分析：**汇入冲突要求汇入车辆找到合适的间隙 (Gap)，并调整速度和路径以安全地插入车流。目标车道上的车辆可能需要适当地减速或加速，甚至微调横向位置，以配合汇入车辆。这是一种典型的协作与博弈混合的场景。

3. **超车冲突 (Overtaking Conflict)**

   - **描述：**当一辆智能车希望超越前方同向行驶的较慢车辆时，会发生超车冲突。这通常涉及变道、加速、完成超车后再变回原车道的过程。
   - **分析：**超车冲突不仅需要考虑超车车辆与被超车辆之间的距离和速度，还需要感知相邻车道上是否有足够的空间进行变道，以及是否存在对向来车（双向车道）。成功的超车要求精细的速度和横向控制，以及对周围车辆行为的预测。

4. **编队行驶冲突 (Platooning Conflict)**

   - **描述：**在车辆编队行驶中，多辆车以很小的间距紧密跟随。当编队需要解散、有新车辆加入或前方出现障碍物时，编队内部以及编队与其他车辆之间可能会发生冲突。
   - **分析：**编队行驶冲突的挑战在于维持编队的稳定性同时处理外部干扰。这需要高精度的通信和协同控制，以确保所有车辆在速度和距离调整上保持一致，避免追尾或不必要的脱离编队。

3.2 在实验平台中实现多车冲突仿真

在当前的实验平台中，实现上述多车冲突的仿真通常依赖于以下几个关键方面：

1. **多车辆实例化：**实验平台允许同时加载和控制多辆智能车模型。每辆车都拥有独立的物理模型、传感器信息（例如：自身位置、速度、航向角、以及其他车辆信息）和控制接口。

2. **通信机制：**智能车辆之间需要建立实时的通信机制，(Inter-Vehicle Communication)。这通常通过 UDP 协议实现，允许车辆广播自身状态信息（如 ID、位置、X、Y、速度、V、航向角 Yaw）并接收其他车辆的状态。这种通信是实现协同感知和决策的基础。

3. **环境配置：**仿真环境需要预设或动态生成复杂的交通场景。

- **十字路口：**通过构建包含交叉道路的模型来实现。可以通过设置车辆的起始点和目标点，使其路径在路口发生交叉。
- **车道汇入/超车：**通过设置多车道的高速公路场景，并规划车辆的路径，使一些车辆从匝道进入主路，或使某些车辆需要进行车道变换。
- **编队：**设置多辆车在同一车道上沿相同方向行驶，并初始设定较小的车间距。

4. **传感器模拟：**除了自身的定位信息，车辆还需要模拟对周围环境的感知能力，例如通过 UDP 接口获取其他车辆的全局状态信息。在更高级的仿真中，可能还会模拟雷达、(Lidar) 或相机传感器数据来检测障碍物和车道线。

5. **独立控制逻辑：**每辆智能车都运行独立的控制算法。这意味着每辆车都有自己的决策模块和执行器，可以根据自身状态和感知到的环境信息，独立计算速度和转向角指令，并发送给仿真平台。

6. **时间同步：**为了确保所有车辆的运动和交互是同步的，仿真平台需要维护一个统一的时间步长，并且所有车辆的控制指令和状态更新都应基于此时间步长进行。

通过上述机制，实验平台能够构建出逼真的多车交互场景，为智能车辆的冲突解决算法提供一个可控且可重复的测试环境。

4 针对不同交互冲突场景的车辆应对策略与实现方法

针对上一节描述的多车冲突场景，智能车辆需要设计相应的应对策略以确保安全性和效率。以下将分别说明针对不同交互冲突场景的车辆应对策略及其实现方法：

4.1 十字路口冲突应对策略与实现

应对策略：基于优先级的避让与协商 在无信号灯的十字路口，可以采用基于优先级的避让策略，并结合简单的协商机制。

- **基本规则：**
  1. **先到先行：**优先让先到达交叉点安全距离内的车辆通过。
  2. **右侧先行：**如果同时到达，则右侧来车拥有优先通过权。
  3. **直行优先：**直行车辆优先于转弯车辆。
- **协商机制：**当优先级无法明确或存在模糊情况时，车辆之间可以进行简单的信息交换（例如，通过 UDP 广播意图：”我将通过“或”我将停车 “），以达成共识。

实现方法：

```
class VehicleControl:
    def __init__(self, vehicle_id, ...):
        self.vehicle_id = vehicle_id
        self.current_state = {} # x, y, yaw, speed
        self.target_path = []
        self.intersection_point = [0, 0] # Assume a known intersection point
        self.conflict_zone_radius = 5.0 # meters
        self.decision_threshold_distance = 15.0 # meters to intersection
        self.has_priority = False
        self.negotiation_status = "idle" # "request", "yield", "proceed"

    def update_state(self, new_state):
        self.current_state = new_state

    def detect_intersection_conflict(self, other_vehicles_states):
        # Check if own vehicle is approaching intersection
        dist_to_intersection = self.calculate_distance(self.current_state, self.intersection_point)
        if dist_to_intersection < self.decision_threshold_distance:
            for other_id, other_state in other_vehicles_states.items():
                if other_id == self.vehicle_id:
                    continue
                other_dist_to_intersection = self.calculate_distance(other_state, self.intersection_point)
                # Check if other vehicle is also approaching conflict zone
                if other_dist_to_intersection < self.decision_threshold_distance:
                    # Check for potential path intersection (simplified for example)
                    # This would involve more sophisticated trajectory prediction
                    if self.will_paths_intersect(other_state):
                        return True, other_id
        return False, None

    def resolve_intersection_conflict(self, other_vehicle_state):
        # Determine priority based on rules (simplified)
        if self.is_first_to_arrive(other_vehicle_state):
            self.has_priority = True
```

```
        elif self.is_right_of_way(other_vehicle_state):
            self.has_priority = True
        else:
            self.has_priority = False

        if self.has_priority:
            # Broadcast "proceed" intent, maintain speed
            self.negotiation_status = "proceed"
            target_speed = self.max_speed
        else:
            # Broadcast "yield" intent, reduce speed or stop
            self.negotiation_status = "yield"
            target_speed = 0.0 # Or slow down gradually

        # Further logic for handling negotiation responses
        # If conflicting "proceed" intents, use tie-breaking rule or random backoff

        return target_speed

    def control_loop(self, other_vehicles_states):
        is_conflict, other_id = self.detect_intersection_conflict(other_vehicles_states)
        if is_conflict:
            target_speed = self.resolve_intersection_conflict(other_vehicles_states[other_id])
            # Apply target_speed to PID controller
        else:
            target_speed = self.pure_pursuit_speed_control() # Normal path following
        # Calculate steering based on pure pursuit
        # Send control commands
```

Listing 2: 十字路口冲突应对策略伪代码

- **感知：**每辆智能车通过 UDP 获取其他车辆的实时位置、速度、航向等信息。
- **冲突检测：**根据自身路径和其他车辆的路径预测，判断是否存在在路口区域的路径交叉。例如，通过计算车辆到达交叉点的时间来预测冲突。
- **优先级判断：**实现 ‘is$_f irst_t o_a rrive$‘ ‘$is_r ight_o f_w ay$‘ **决策与控制：**
-
  - 如果判断自己拥有优先级，则保持或加速通过路口，同时广播自己的意图。
  - 如果判断需要避让，则根据与路口的距离和冲突车辆的相对位置，平稳地减速甚至停车，直到冲突解除。

- 结合纵向 PID 控制器实现平滑的速度调整，横向使用纯跟踪算法维持车道。

信息交互： 利用 UDP 客户端和服务器进行车辆意图的广播和接收。例如，车辆可以发送“我将通过优先级路口”或“我将让行”等消息。

4.2 车道汇入冲突应对策略与实现

应对策略：协作式汇入与间隙选择 汇入车辆需要找到主车道上的合适间隙，而主车道上的车辆则需要提供必要的协助。

- **汇入车辆：** 寻找目标车道上前方和后方车辆之间的安全间隙，计算进入该间隙所需的加减速和转向轨迹。
- **主车道车辆：** 当检测到有汇入车辆靠近时，如果当前位置没有合适的间隙，可以适度调整速度（例如，轻微减速以扩大后方间隙，或轻微加速以让出前方间隙），或者短暂调整横向位置以提供更多空间，从而协作汇入。

实现方法：

```
class MergingVehicle:
    def __init__(self, ...):
        self.is_merging = False
        self.target_lane_id = 1
        self.merging_speed_profile = [] # Pre-calculated speed profile
        self.gap_threshold = 10.0 # meters

    def find_suitable_gap(self, main_lane_vehicles):
        # Iterate through main lane vehicles to find a gap
        # Consider relative distances and speeds
        for front_car, rear_car in self.get_gaps(main_lane_vehicles):
            if self.calculate_gap_size(front_car, rear_car) > self.
    gap_threshold:
                # Calculate if self can safely merge into this gap
                if self.can_safely_insert(front_car, rear_car):
                    return True, front_car, rear_car
        return False, None, None

    def execute_merge(self, front_car, rear_car):
        # Adjust longitudinal control (speed PID setpoint) to match
    gap
        # Adjust lateral control (pure pursuit to new lane path)
        self.is_merging = True
        # Send merge intent to other vehicles (optional)

```

```
class MainLaneVehicle:
    def __init__(self, ...):
        self.is_assisting_merge = False

    def detect_merging_vehicle(self, merging_vehicle_state):
        # Check if a merging vehicle is within proximity and intends to merge
        pass

    def assist_merge(self, merging_vehicle_state):
        # If appropriate, adjust speed or lateral position slightly
        if self.current_speed > merging_vehicle_state.speed:
            self.target_speed = max(self.current_speed * 0.9, merging_vehicle_state.speed + 1.0) # Slow down slightly
        # Or, if space allows, slightly shift laterally within own lane
        self.is_assisting_merge = True
        # Send assistance intent (optional)
```

Listing 3: 车道汇入冲突应对策略伪代码

- **汇入车辆：**
  - **间隙检测：** 感知目标车道上车辆的位置和速度，计算出可用的间隙大小。这需要复杂的几何和时间预测。
  - **路径规划：** 基于选定的间隙，动态规划一条从当前车道到目标车道的平滑汇入路径，并生成相应的速度剖面。可以使用样条插值，(Spline Interpolation)，或贝塞尔曲线等方法。
  - **控制：** 利用纯跟踪算法跟踪新规划的汇入路径，并使用 PID 控制器精确调整速度，以匹配目标间隙车辆的速度。
- **主车道车辆：**
  - **汇入请求感知：** 监听周围环境，检测是否有车辆表现出汇入意图（例如，转向灯信号，或直接通过 UDP 发送汇入请求）。
  - **协作决策：** 如果条件允许（例如，不会导致急刹车或危及自身安全），主动调整自己的速度，为汇入车辆提供一个合适的间隙。这可能涉及纵向的微调速，甚至横向的微调以拉开空间。

4.3 编队行驶冲突应对策略与实现

应对策略：分布式协同控制与异常处理 编队行驶中的冲突主要围绕编队稳定性、成员加入/退出和突发事件处理。

- **分布式控制：**每辆编队车辆都独立运行控制算法，但通过车间通信共享信息（如前车速度、距离），以实现协同控制。
- **间距保持：**基于恒定时间间距策略 (Constant Time Headway Policy)，动态调整自身速度以保持与前车的安全距离。
- **异常处理：**当编队中某车辆出现异常（如急刹车、失控）或外部障碍物进入编队路径时，所有编队车辆需要快速响应，进行协同减速、避障或解散编队。

实现方法：

```
class PlatooningVehicle:
    def __init__(self, vehicle_id, ...):
        self.platoon_id = "A"
        self.leader_id = None
        self.predecessor_id = None
        self.target_gap = 1.5 # seconds, for constant time headway
        self.k_p_platoon = 0.5 # Proportional gain for distance control

    def update_platoon_state(self, all_vehicles_states):
        # Get leader and predecessor states from communication
        leader_state = all_vehicles_states.get(self.leader_id)
        predecessor_state = all_vehicles_states.get(self.predecessor_id)

        if leader_state and predecessor_state:
            # Calculate desired speed based on predecessor's speed and distance
            current_distance = self.calculate_distance(self.current_state, predecessor_state)
            desired_distance = predecessor_state.speed * self.target_gap

            # Simple proportional control for distance
            distance_error = current_distance - desired_distance
            target_speed = predecessor_state.speed + self.k_p_platoon * distance_error

            # Ensure target speed is within limits
            target_speed = max(0.0, min(self.max_speed, target_speed))

            # Apply target_speed to PID controller
            self.speed_pid.set_setpoint(target_speed)

        # Emergency braking/evasion for obstacles
        if self.detect_obstacle_in_path():
```

```
            self.speed_pid.set_setpoint(0.0) # Emergency stop
            # Potentially broadcast emergency braking signal to other
platoon members
```

Listing 4: 编队行驶冲突应对策略伪代码

- **车间通信：**车辆持续广播自身的速度和位置，并接收前车和后车的信息。
- **纵向控制：**
  - **定间距控制：**使用 PID 或其他控制器（如：线性二次调节器，(LQR)）来保持与前车的预设距离或时间间距。目标速度根据前车的速度和当前车间距动态调整。
  - **队列管理：**实现车辆加入和退出编队的协议，例如通过加速追上前车或减速脱离编队。
- **横向控制：**继续使用纯跟踪算法或 $Stanley$ 控制算法来保持在车道中心线行驶。
- **异常与突发事件：**
  - **紧急制动：**当前方检测到紧急障碍物或前车急刹车时，迅速触发紧急制动，并同时向后方车辆广播紧急情况，触发连锁反应。
  - **编队解散：**在无法通过减速避让时，协同执行编队解散操作，各车辆自主寻找新的路径或停车。

上述策略和实现方法为智能车辆在多车交互环境下的冲突解决提供了基础框架。实际应用中，还需要结合更精确的感知、预测模型以及更复杂的决策算法，如强化学习，(Reinforcement Learning)，或模型预测控制，(Model Predictive Control, MPC)，以实现更鲁棒和高效的解决方案。

5 实验过程中的问题与收获

5.1 遇到的问题及解决方式

问题 1：多车通信延迟与数据不同步。 **描述：**在仿真初期，多辆车通过 UDP 进行状态信息交换时，发现接收到的其他车辆状态存在明显延迟，有时甚至数据跳变，导致车辆决策基于过时的信息，从而引发不必要的冲突或不协调的运动。特别是在高控制频率下，问题更为突出。 **解决方式：**

- 1. **优化 UDP 接收逻辑：**确保 UDP 接收线程能够以尽可能高的频率轮询和处理传入的数据包，减少数据在缓冲区中的停留时间。

  2. **时间戳同步与数据验证：**在每个数据包中加入发送时间戳。接收方在处理数据时，可以根据时间戳判断数据的新鲜度，丢弃过旧的数据包。同时，对接收到的数据进行基本校验，例如速度和位置的合理性，过滤掉明显异常的数据。
  3. **状态预测：**在等待最新数据或数据偶尔丢失时，利用简单的运动学模型对其他车辆的状态进行短期预测，以弥补通信延迟。例如，假设其他车辆在短时间内保持当前速度和航向，估算其未来位置。

- **问题 2: 十字路口冲突解决中的“死锁”或效率低下。**

  描述: 在十字路口场景中，当多辆车同时到达路口且优先级规则不足以明确区分时，车辆可能陷入相互等待的“死锁”状态，或者因为过于保守的避让而导致通行效率极低。例如，两辆对向左转车辆同时到达，都等待对方先通过。 **解决方式:**

  引入随机性或超时机制：在优先级无法明确时，引入一个小的随机等待时间，或者设置一个超时机制。当车辆等待超过一定时间后，随机选择一辆车优先通过，打破僵局。

  增加协商维度：引入更复杂的协商机制，不仅仅是简单的意图广播，而是可以交换“期望通过时间”、“当前速度”等信息，并通过简单的拍卖机制或投票机制来确定通过顺序。 **中央协调 ((Centralized Coordination)，选做)：**对于固定路口，可以引入一个中央协调器来管理路口通行权，通过向每辆车发送通行许可来避免冲突。但这会增加系统复杂度。

3. **问题 3: 动态路径规划与跟踪的平滑性与实时性平衡。**

   描述: 在实现动态避障或车道汇入时，如果实时生成的路径过于激进或频繁更新，可能导致车辆运动不平滑，出现抖动或突然转向。而如果过于追求平滑，又可能牺牲了对突发情况的实时响应能力。 **解决方式:** **多项式拟合或样条曲线：**在生成避障或汇入路径时，使用高阶多项式、(Polynomial

Fitting)，或贝塞尔/B-样条曲线，(B-Spline)，来确保生成路径的曲率连续性，从而保证车辆运动的平滑性。 **频率控制与平滑滤波：**限制路径更新的频率，避免过于频繁的路径重规划。同时，对控制指令（如转向角、加速度）应用低通滤波器，(Low-Pass Filter)，或滑动平均滤波，(Moving Average Filter)，以消除高频噪声，平滑输出。 **渐进式调整：**当需要大幅度调整路径时，不是立即跳到新路径，而是逐步向新路径过渡，通过速度和转向角的渐进式变化来实现平稳过渡。

5.2 总结收获与不足

收获：

多智能体系统理解：通过实践，对多智能体系统中的交互、感知、决策和控制有了更直观和深入的理解。认识到单个智能体的性能不足以解决复杂交通问题，协同才是关键。 **冲突场景分析能力：**掌握了识别和分析不同交通场景下多车冲突的类型和特点，为后续设计解决方案奠定了基础。 **策略设计与实现经验：**锻炼了针对特定冲突场景设计应对策略的能力，并将其转化为可执行的代码。特别是对基于规则、协商和协作的策略有了初步的实践经验。 **系统集成与调试：**熟悉了在仿真平台中集成多车控制模块、通信模块和感知模块的方法，提升了系统调试和问题解决的能力。 **对安全与效率的权衡：**在解决冲突时，深刻体会到安全 ((Safety)) 与效率 ((Efficiency)) 之间的权衡。过于保守可能导致低效率，过于激进则可能增加碰撞风险。

不足：

预测能力有限：当前的冲突解决策略主要基于实时感知和简单规则，对其他车辆的意图和未来轨迹的预测能力较弱。在动态变化或高不确定性环境中，这可能导致决策的局限性。 **缺乏复杂决策算法：**多数策略仍停留在基于规则或简单协作层面，未能深入应用模型预测控制 ((MPC))、强化学习 ((Reinforcement Learning)) 或博弈论 ((Game Theory)) 等更高级的决策算法来处理复杂多变的交互情境。 **鲁棒性不足：**在极端情况（如通信中断、传感器故障、恶意行为）下，当前的系统鲁棒性较差，可能无法有效处理。 **通用性欠缺：**针对不同冲突场景设计的策略往往是独立的，未能形成一个统一、可扩展的通用冲突解决框架，未来需要考虑如何将这些策略进行模块化和整合。 **实验评估指标单一：**实验评估主要关注是否避免碰撞，对交通流量、通行时间、乘客舒适度等更全面的评估指标考虑不足。

本次实验为多智能体集群控制奠定了实践基础，但仍有广阔的探索空间，尤其是在提升智能体预测能力、引入高级决策算法和增强系统鲁棒性方面。未来的研究将致力于解决这些不足，构建更智能、更高效、更安全的交通系统。