POLITECNICO DI TORINO

MASTER's Degree in COMPUTER ENGINEERING

MASTER's Degree Thesis

Development of autonomous driving control algorithms based on ASIL scenarios.

Supervisors

Prof. Massimo VIOLANTE

Candidate

Cihan YURTSEVER

06 2024

Summary

The advancement of autonomous systems has catalyzed significant transformations across various industries, notably the automotive sector. This disruptive technology has become a cornerstone of innovation, fundamentally reshaping conventional paradigms of transportation and mobility. The incorporation of autonomous features into vehicles holds immense potential for enhancing efficiency, comfort, and, most importantly, safety on our roadways.

This study aims to implement autonomous driving functionalities through a systematic approach encompassing perception, motion planning, control, and actuation. The primary objective is to evaluate the efficacy and outcomes of different control and motion planning algorithms concerning the critical facet of autonomous driving: safety.

For safety assessment, the study will adhere to the Automotive Safety Integrity Level (ASIL) standards outlined in ISO 26262. Hazardous scenarios identified through Hazard Analysis and Risk Assessment (HARA) will be leveraged to compare the most and least complex control algorithms (PID and MPC). Real-world scenarios, representing both urban and rural settings, will be simulated within the Carla Simulation environment due to its capability in scenario modeling. Path planning will utilize the Carla Simulator API, while both the camera and obstacle sensors of the simulator will be employed for perception tasks. The car model will be based on the bicycle model, and motion planning will utilize adaptive cruise control.

The project will be developed in Python programming, adhering to PEP8 standards, and executed on the Ubuntu operating system. Code quality and readability will be evaluated using Pylint linting analysis, with logging and output tracking in log and CSV file formats to store operational data.

The study will culminate in a practical exploration of real-life scenarios within the Carla simulator, followed by a comparative analysis of outcomes, including cross-track and heading errors. These parameters will be used for ISO 26262 safety assurance by employing HARA to evaluate the safety parameters of the control algorithm.

Acknowledgements

"Disce quasi semper victurus vive quasi cras moriturus"

First and foremost, I would like to express my deepest gratitude to my family. In particular, I am profoundly thankful to my mother for her guidance and inspiration, which have shaped my educational pursuits.

I am also deeply grateful to my close friend Ataberk from my Istanbul days, whose intellectual insights and stimulating conversations have been invaluable.

A heartfelt thank you goes to my dear friend Andrea. Your steadfast loyalty and constant presence have been a source of strength and comfort.

I am also indebted to my company tutors, Mauro, Nicola, and Elena, for providing me with this incredible opportunity. Your mentorship and guidance have been pivotal in my professional growth. Additionally, I would like to thank Professor Violante for his enlightening courses that have laid a strong foundation for my academic and professional endeavors.

Special thanks to my aunt Noli, and her son Emir for their unwavering love, support, and encouragement.

Lastly, I would like to express my deep appreciation to Sanna and the Gavotto family for their generous hospitality and unwavering support. Your kindness and encouragement have meant the world to me, and I am forever grateful for your friendship and care.

Table of Contents

List of Tables VI

List of Figures VII

Acronyms IX

1 Introduction 1

1.1 Autonomous Driving . . . 3

1.1.1 Autonomous Driving Software Stack . . . 4

1.1.2 Automotive Safety Integrity Level (ASIL) . . . 7

1.2 State of the Art . . . 8

1.3 Thesis Motivation . . . 11

2 Background 13

2.1 Vehicle Dynamics . . . 14

2.1.1 Tire Forces and Moments . . . 14

2.1.2 Bicycle Model . . . 15

2.1.3 Longitudinal and Lateral Motion . . . 16

2.1.4 Vehicle Stability . . . 18

2.2 Control Dynamics . . . 20

2.2.1 Control theory in autonomous vehicles . . . 20

2.2.2 Reactive Control in Autonomous Vehicles . . . 21

2.2.3 Proportional-Integral-Derivative (PID) Control . . . 21

2.2.4 Optimal Control . . . 23

2.2.5 Model Predictive Control (MPC) . . . 24

2.2.6 Control Algorithms . . . 26

2.3 Basics of Motion Planning . . . 28

2.3.1 Adaptive Cruise Control (ACC) . . . 29

2.4 ASIL and HARA Standards . . . 33

3 Architecture and Methodology 37
3.0.1 Environment Management with Conda 39
3.1 Python language and libraries 39
3.1.1 Python 40
3.1.2 Code Linting and Formatting 40
3.2 Carla Simulator 41
3.2.1 Client to Carla Environment Localhost Communication for Efficient Interaction 45
3.3 Linux Ubuntu Operating System and Shell Script 46
3.3.1 Project File Directory; Modules and Submodules 51

4 Implementation 54
4.1 Client Implementation 54
4.2 Setup Environment and Objects 56
4.2.1 World Creation 56
4.2.2 Synchronized Settings 57
4.2.3 Waypoint Visualization 57
4.2.4 Vehicle Spawning 59
4.2.5 Sensors Spawning 61
4.2.6 Cleanup and Resource Management 64
4.3 Controller; Driving Car Through Path. 64
4.3.1 PID Controller Implementation 65
4.3.2 MPC Controller Implementation 69

5 Conclution and Final Assessment 81
5.1 Benchmark on Control Operations and Algorithms 81
5.2 Hazard and Risk Assessment: PID vs MPC 86
5.3 Final Considerations 87
.0.1 Setting Up the Environment 91

Bibliography 93

List of Tables

1.1 HARA analysis concerns scenarios (1-9) with regards to the probability of severity, exposure, and controllability. 7

5.1 HARA analysis for scenarios (1-6) with respect to PID Algorithm (A1) and MPC Algorithm (A2). 86

List of Figures

1.1 SAE levels of driving automation. 2
1.2 Autonomous driving pipeline. 3
1.3 Software Stack Pipeline of Autonomous Driving. 4
1.4 Perception detection. 4
1.5 Localization and Mapping representation. 5
1.6 (a) Path planning, (b) maneuver planning, and (c) trajectory planning. 6
1.7 Closed loop PID controller. 6

2.1 Tire Forces and Moments 14
2.2 Bicycle Model 15
2.3 Lateral and Longitudinal distances. 17
2.4 Heading Error. 18
2.5 Cross Track Error. 19
2.6 Example of Reactive Control. 21
2.7 Example of Optimal Control. 23
2.8 Example of Autonomous Vehicle Flowchart. 26
2.9 Adaptive Cruise Control. 29
2.10 ISO development cycle. 35

3.1 Project System Architecture. 38
3.2 Carla Simulator. 41
3.3 Low rendering quality of Carla Simulator. 44
3.4 Epic rendering quality of Carla Simulator. 45
3.5 The CARLA (UE4) Server Interface. 50

4.1 Waypoints drawn on the path. 59
4.2 Microlino car that will be used. 60
4.3 Real life camera and lidar. 61

5.1 Operation view in a rural scenario. 82
5.2 PID controller errors in a rural scenario. 83
5.3 MPC controller errors in a rural scenario. 83

5.4 Complex urban scenario. 84
5.5 MPC algorithm operation in an urban scenario. 85
5.6 PID algorithm operation in an urban scenario. 85

Acronyms

ACC

Adaptive Cruise Control

ADAS

Advanced Driver Assistance System

AI

Artificial Intelligence

ASIL

Automotive Safety Integrity Level

AUTOSAR

Automotive Open System Architecture

CARLA

Car Learning to Act

CTE

Cross Track Error

HARA

Hazard and Risk Assessment

LIDAR

Light Detection and Ranging

MPC

Model Predictive Control

PEP8	Python Enhancement Proposal 8
PID	Proportional Integral Derivative
SAE	Society of Automotive Engineers

Chapter 1

Introduction

The concept of self-driving vehicles may seem futuristic, but its roots can be traced back to the curious minds of the Middle Ages. Throughout history, humans have harbored a curiosity for autonomous transportation. However, it wasn't until the early 20th century that the first steps towards self-driving cars were taken, albeit rudimentary due to limited technological understanding of physical laws.

In 1920, an early experiment with a crude ratio controller car was conducted, aiming to control a 1926 model Chandler equipped with a transmitter antenna[1]. This vehicle, operated by a person in another car, sent out radio impulses to control its movements.

Fast forward to 1948, and the dawn of advanced driver assistance systems emerged with the invention of Cruise Control by Ralph Teetor, then president of the Society of Automotive Engineers (SAE). This pioneering technology was first commercially applied in Chrysler's luxury model "Auto Pilot" in 1958, later known as Cruise Control.

Japan's Tsukuba Mechanical Engineering Laboratory made significant strides in 1977 with the development of the first semi-autonomous self-driving car[2], albeit with support from an elevated rail. By 1995, Navlab5 in the USA had developed neural networks to control steering, marking a milestone in autonomous vehicle technology even though throttle and brakes were human-controlled, chiefly for safety reasons**??**.

In the early 2000s, the integration of obstacle avoidance systems began to emerge within autonomous car projects. By 2005, incorporating Light Detection and Ranging(LiDAR) sensors for motion planning became prevalent. In 2007, the DARPA challenge unfolded in an urban setting, witnessing the introduction of various control methods, each contributing to a significant milestone in control algorithms. The rapid pace of technological advancement and innovation fueled car companies to intensify their efforts in autonomous car projects. However, the proliferation of projects using the term "autonomous" confused, necessitating the establishment of standardized definitions. Consequently, in 2014, the Society of Automotive Engineers (SAE) introduced Autonomy standards encompassing six levels[3] as shown in Figure 1.1.

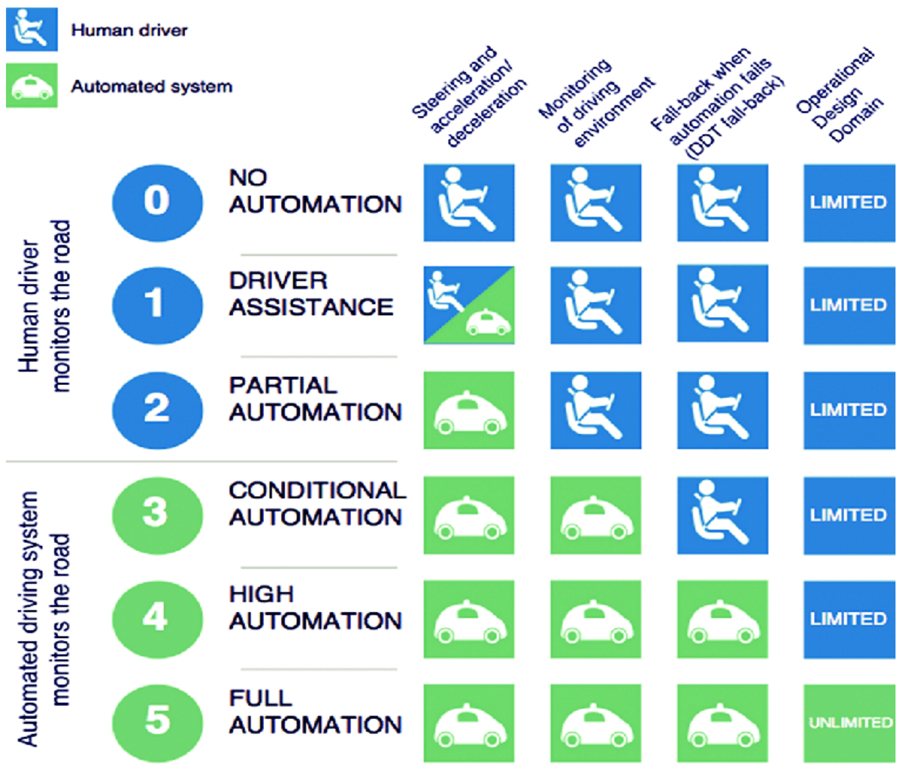

Human driver
Automated system
Steering and acceleration/ deceleration
Monitoring of driving environment
Fall-back when automation fails (DDT fall-back)
Operational Design Domain
Human driver monitors the road
Automated driving system monitors the road
0 NO AUTOMATION
1 DRIVER ASSISTANCE
2 PARTIAL AUTOMATION
3 CONDITIONAL AUTOMATION
4 HIGH AUTOMATION
5 FULL AUTOMATION
LIMITED
UNLIMITED

Figure 1.1: SAE levels of driving automation.

In the following years, the development of autonomous vehicle projects began to adhere to these standardized definitions. In 2017, major automotive companies opted to release open-source software modules and interface Automotive Open System Architecture (AUTOSAR)[4]. This initiative gained momentum in 2019 when legal adjustments commenced in Europe, the USA, the United Kingdom, Japan, and various other countries. Presently, the deployment and standardization of autonomous driving technologies are regulated according to specific deployment models. The autonomous driving mission is typically divided into localization and mapping, perception, motion planning, and control. In the subsequent pages, we will delve into these steps along with applications developed in the Carla Simulator.

1.1 Autonomous Driving

Autonomous driving represents the endeavor to empower a vehicle with the ability to navigate diverse scenarios independently. That expectation comes with challenges and limitations: safety concerns, and regulatory hurdles. Additionally, there are technological limitations, such as the reliability of sensors in adverse weather conditions. Those challenges entail encountering hazardous situations where the vehicle must make decisions promptly. Consequently, the vehicle in every situation must be:

- Self-aware of its location;
- Understand of dynamic and static objects in its environment;
- Adapt to location and environment, determine real-time paths to be reached;
- Implement the planned path for vehicle operation.

As illustrated in Figure 1.2, the vehicle is outfitted with hardware and software elements to ensure its safety and efficiency to satisfy the expected conditions as previously listed. The hardware includes Electrical Control Units (ECUs), cameras, LiDAR, and a variety of other sensors. Meanwhile, the software consists of Localization, Perception, Motion Planning, and Control algorithms. These four main software components contribute to the vehicle's attainment of different SAE autonomy levels.

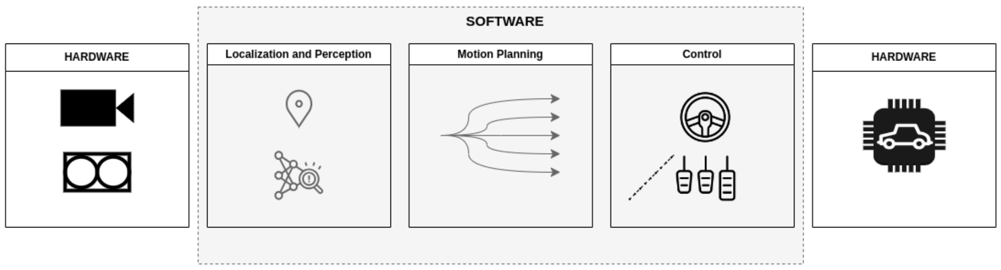


Figure 1.2: Autonomous driving pipeline.

As the level of SAE autonomy increases, the complexity of hazard situations and risks also rises. In this context, Hazard and Risk Assessment (HARA) analysis is employed to verify whether the expectations are met. HARA assesses different scenarios and expects the vehicle to behave appropriately in each one to avoid hazardous and risky situations. Therefore, the tasks of Localization, Perception, Motion Planning, and Control software gain significance in the vehicle's operation.

1.1.1 Autonomous Driving Software Stack

The dream of self-driving vehicles hinges on the power of software. This software serves as the vehicle's "brain," transforming raw data into intelligent decision-making and ultimately enabling autonomous operation. The Autonomous Driving Software Stack (AD Stack) encompasses four critical functions working in concert: perception, localization, motion planning, and control.

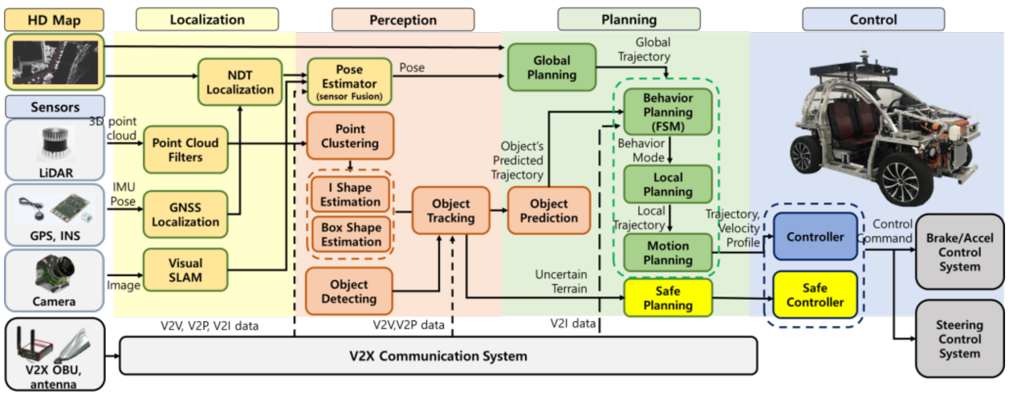


Figure 1.3: Software Stack Pipeline of Autonomous Driving.

Perception

Perception is the software component responsible for interpreting data from various sensors to understand the vehicle's surroundings. It involves processing information from cameras, LiDAR, radar, and other sensors to identify objects such as vehicles, pedestrians, and road signs. Perception algorithms analyze sensor data to detect

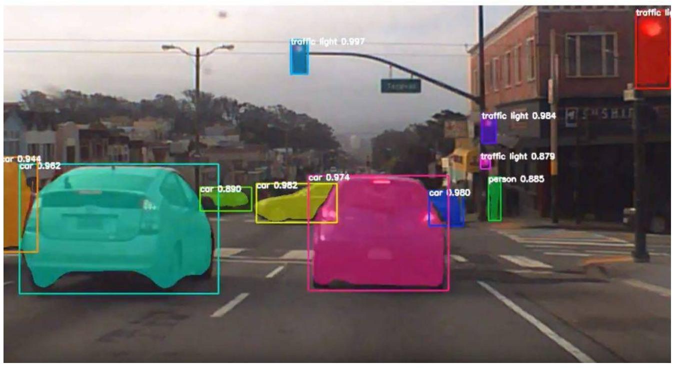


Figure 1.4: Perception detection.

and classify objects, estimate their positions and velocities, and predict their future movements as in Figure 1.4. This information forms the basis for higher-level decision-making in autonomous driving systems.

Localization and Mapping

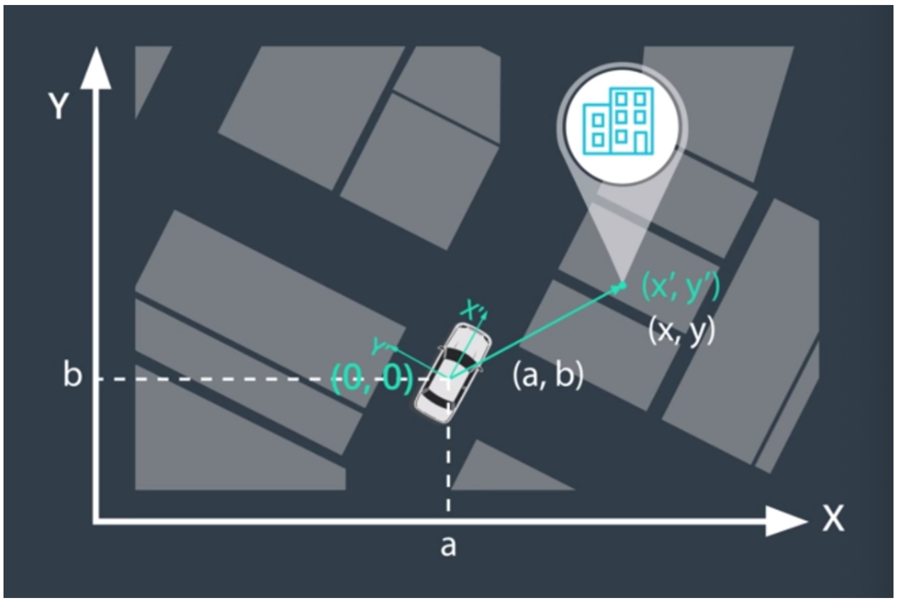


Figure 1.5: Localization and Mapping representation.

Localization and mapping software enables the vehicle to accurately determine its position and orientation within its environment, often referred to as the "pose." Localization algorithms use sensor data, such as GPS (Global Positioning System) and inertial measurement units (IMUs), to estimate the vehicle's pose relative to a known map of the environment. Simultaneously, mapping algorithms create and update detailed maps of the surroundings, incorporating information from sensors to represent features like roads, lanes, and landmarks. By continuously comparing sensor data to the map, localization, and mapping software ensure precise navigation and enable the vehicle to follow planned trajectories safely.

Motion Planning

Motion planning is a critical aspect of autonomous driving software responsible for generating feasible and safe trajectories for the vehicle to follow. This process involves computing optimal paths through the environment while considering various factors such as the vehicle's current position, destination, surrounding obstacles, and dynamic constraints. Motion planning algorithms aim to find paths that minimize travel time, maintain safe distances from other objects, adhere to traffic regulations,

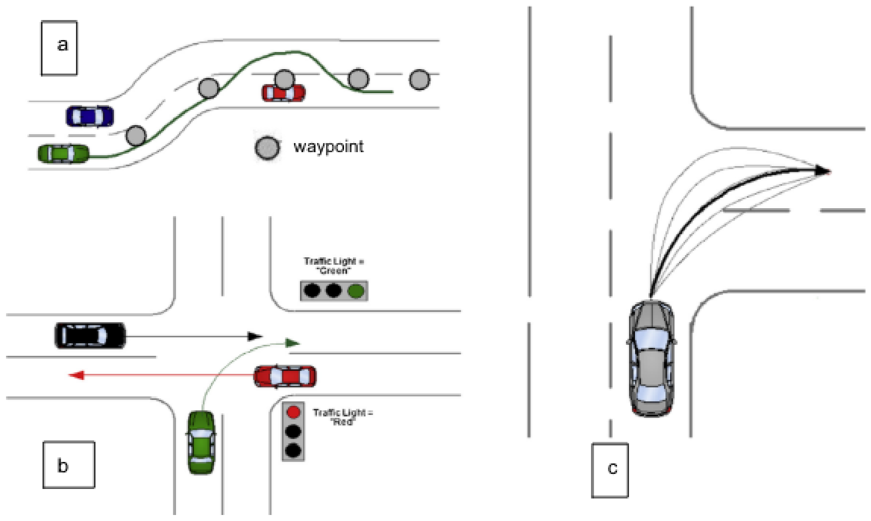


Figure 1.6: (a) Path planning, (b) maneuver planning, and (c) trajectory planning.

and optimize overall efficiency. These algorithms utilize advanced techniques such as probabilistic roadmaps, potential fields, and optimization methods to navigate complex environments and ensure reliable autonomous driving.

Control

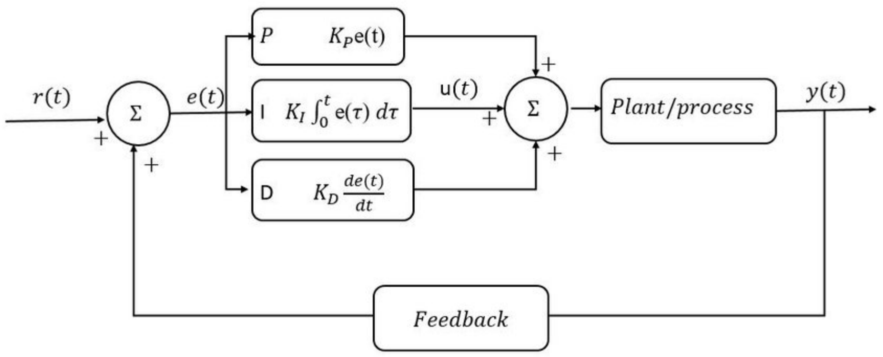


Figure 1.7: Closed loop PID controller.

Motion planning is a critical aspect of autonomous driving software responsible for generating feasible and safe trajectories for the vehicle to follow. This process involves computing optimal paths through the environment while considering various factors such as the vehicle's current position, destination, surrounding obstacles, and dynamic constraints. Motion planning algorithms aim to find paths that minimize travel time, maintain safe distances from other objects, adhere to traffic regulations, and optimize overall efficiency. These algorithms utilize advanced techniques such as probabilistic roadmaps, potential fields, and optimization methods to navigate complex environments and ensure reliable autonomous driving.

1.1.2 Automotive Safety Integrity Level (ASIL)

The paramount concern in the development of autonomous vehicles is safety. The automotive industry employs the Automotive Safety Integrity Level (ASIL), a foundational principle within the ISO 26262 standard, to gauge safety levels. ASIL categorizes the potential severity of hazards that could arise from malfunctions in a vehicle's electrical and electronic systems. These classifications directly influence the development process, dictating the level of rigor required to achieve the necessary safety objectives. ASIL functions as a tiered system, ranging from A (least severe) to D (most severe), with each level mandating progressively stricter safety requirements.

	S1	S2	S3	S4	S5	S6	S7	S8	S9	ASIL
H1	S:0 E:4 C:2	S:3 E:4 C:2	S:2 E:3 C:0	S:1 E:4 C:1	S:3 E:4 C:2	S:1 E:4 C:0	S:3 E:4 C:3	S:0 E:4 C:2	S:0 E:4 C:2	D
H2	S:3 E:4 C:2	S:1 E:4 C:0	S:3 E:4 C:2	S:1 E:4 C:1	S:3 E:4 C:2	S:3 E:4 C:2	S:0 E:4 C:2	S:0 E:2 C:2	S:1 E:2 C:3	C
H3	S:1 E:4 C:0	S:0 E:4 C:0	S:0 E:3 C:0	S:0 E:4 C:0	S:0 E:3 C:0	S:2 E:4 C:2	S:1 E:4 C:1	S:0 E:4 C:1	S:0 E:2 C:2	A

Table 1.1: HARA analysis concerns scenarios (1-9) with regards to the probability of severity, exposure, and controllability.

Working in tandem with ASIL is Hazard Analysis and Risk Assessment. HARA establishes the groundwork for ASIL classification by systematically identifying and appraising potential hazards that could stem from system malfunctions. This process involves defining safety goals and assigning corresponding ASIL levels specific to different operational scenarios. In essence, HARA guides the risk assessment process by identifying potential hazards, assessing their severity, and defining appropriate safety objectives and ASIL classifications. This collaborative approach between ASIL and HARA forms the foundation of safety protocols for autonomous vehicles, ensuring reliable and secure operation across various driving conditions. An example of HARA analysis is provided in Table 1.1. In this table, scenarios and hazards are analyzed with respect to severity, exposure, and controllability. The results of the analysis are presented in the last column, indicating their ASIL score.

Both ISO 26262 standards further emphasize a systematic approach to safety throughout the entire development lifecycle, encompassing not only the design and production phases but also operation, service, and decommissioning. In this study will leverage HARA to create specific operating scenarios in the Carla Sim environment, to facilitate the testing and comparison of various autonomous vehicle control algorithms.

1.2 State of the Art

The landscape of autonomous vehicles is experiencing a profound transformation, fueled by relentless advancements across a spectrum of technological domains. In this comprehensive exploration, we delve into the current state-of-the-art for each pivotal component of the autonomous vehicle software stack, namely perception, localization and mapping, motion planning, and control algorithms.

Perception

Modern autonomous vehicles employ an array of sensors, including cameras, LiDAR, radar, and ultrasonic sensors, to gain a comprehensive understanding of their surroundings. Sensor fusion integrates data from these diverse sources, enabling Autonomous vehicles to perceive their environment holistically. Deep learning algorithms are instrumental in this process, facilitating accurate object detection, classification, and tracking, thereby enabling Autonomous vehicles to recognize vehicles, pedestrians, traffic signs, and other relevant objects with high precision [5], [6].

- **LiDAR for High-Resolution Perception:** LiDAR sensors provide Autonomous vehicles with precise 3D point cloud data, allowing for meticulous object detection and localization. This capability is particularly valuable in navigating complex urban environments and challenging weather conditions where traditional sensors may struggle. By leveraging LiDAR technology, Autonomous vehicles can achieve enhanced precision and confidence in their operations [7].

- **Camera-Based Perception with Deep Learning:** Cameras are ubiquitous sensors in Autonomous vehicles due to their affordability and ability to capture rich visual data. Deep learning algorithms have revolutionized camera-based perception tasks, including object detection, lane recognition, and traffic sign identification. By harnessing deep neural networks, Autonomous vehicles can extract crucial information from visual scenes, facilitating real-time decision-making with unparalleled accuracy [8].

Localization and Mapping

- **Simultaneous Localization and Mapping (SLAM):** SLAM algorithms are essential for autonomous navigation systems, enabling Autonomous vehicles to construct real-time maps of their surroundings while simultaneously determining their precise location within these maps. By synthesizing data from LiDAR and visual odometry, SLAM algorithms play a crucial role in ensuring reliable path planning and obstacle avoidance capabilities in Autonomous vehicles [9] [10].

- **High-Definition (HD) Maps:** HD maps provide detailed representations of the environment, including lane markings, traffic signs, and landmarks. These meticulously crafted maps are used in conjunction with SLAM algorithms to enhance localization accuracy and overcome potential limitations of on-board sensors. By leveraging HD maps, Autonomous vehicles can navigate complex environments with greater precision and confidence [11].

Motion Planning

- **Probabilistic Roadmaps (PRM):** PRM is a fundamental motion planning technique for Autonomous vehicles, offering a robust framework for generating feasible paths in complex environments. By probabilistically sampling the environment, PRM algorithms construct a roadmap of potential paths, enabling Autonomous vehicles to navigate towards their destination efficiently while avoiding obstacles and adhering to safety constraints [12].

- **Rapidly-exploring Random Trees (RRT):** RRT algorithms provide an efficient approach to motion planning in dynamic environments, incrementally growing a tree-like structure within the environment. By iteratively exploring the configuration space, RRT algorithms enable Autonomous vehicles to adapt to changing surroundings while ensuring collision-free trajectories [13].

- **Learning-based Motion Planning:** Reinforcement learning algorithms are emerging as promising tools for developing adaptive motion planning strategies in Autonomous vehicles. By learning from simulated driving experiences, these algorithms can iteratively refine their policies, leading to agile and efficient motion planning strategies that can handle diverse driving scenarios with ease [14].

Control Algorithms

- **Proportional-Integral-Derivative (PID) Control:** PID control remains a popular choice for regulating vehicle motion due to its simplicity and effectiveness. By adjusting proportional, integral, and derivative terms, PID controllers can maintain desired vehicle dynamics, such as speed and trajectory. However, PID control may struggle in highly dynamic environments or when faced with complex vehicle dynamics [15].

- **Model Predictive Control (MPC):** MPC algorithms leverage mathematical models of the vehicle to predict its future behavior and optimize control inputs over a finite time horizon. This proactive control strategy enables Autonomous vehicles to anticipate and mitigate potential risks, making them well-suited for handling complex dynamics and uncertain environments with confidence [16].

- **Learning-based Control:** Deep reinforcement learning techniques are gaining traction in the realm of autonomous vehicle control algorithms, offering a data-driven approach to developing robust and adaptable control strategies. By learning directly from simulated driving experiences, these algorithms can autonomously acquire optimal control policies, leading to enhanced performance and safety in diverse driving scenarios [17].

This comprehensive overview highlights the significant progress and ongoing innovation in perception, localization and mapping, motion planning, and control algorithms for autonomous vehicles.

ASIL analysis on autonomous vehicle technologies

ASIL analysis is an iterative process that evolves alongside advancements in autonomous vehicle technologies. As new features are introduced and algorithms are refined, continuous ASIL assessment is essential to adapt to emerging risks and maintain the highest standards of safety.

Through meticulous risk assessment and classification, ASIL analysis categorizes potential hazards into four levels: ASIL A, B, C, and D, with ASIL D representing the highest level of risk. By assigning ASIL levels to different components and functions within autonomous driving systems, engineers can prioritize safety-critical elements and allocate resources accordingly to ensure robustness and reliability.

The ASIL standard HARA analysis is crucial for estimating and assessing the risk posed by electronics in road vehicles. With the capabilities of autonomous vehicles expanding, there's a need to broaden the analysis scope to encompass more complex scenarios [18]. As the development of autonomous vehicles progresses, there is a growing demand for different and more specific scenarios, prompting research efforts to focus on function-specific impacts on operations like braking, acceleration and some other operations[19][20]. Some studies suggest that Real-time decision-making in autonomous driving necessitates tailoring using a quantitative risk norm (QRN) with consequence classes, each with defined limits for occurrence frequency. Incident types are then categorized and assigned to these consequence classes, with requirements serving as safety goals for implementation. The QRN approach ensures the completeness of safety goals and prevents limitations imposed by poorly formulated safety goals for an Autonomous Driving System (ADS)[21].

1.3 Thesis Motivation

The transportation landscape is undergoing a profound transformation, driven by the increasing potential of autonomous vehicles. These advanced vehicles hold the promise of a future marked by heightened safety, efficiency, and convenience. However, ensuring the safety of passengers and others on the road remains of utmost importance. To achieve this objective, rigorous development processes and robust control systems are indispensable.

The ISO 26262 standard, with its HARA methodology, plays a crucial role in ensuring safety in autonomous vehicle development. Traditionally utilized for assessing the electronics of autonomous vehicles, we aim to leverage HARA analysis in this thesis to benchmark the safety performance of different control algorithms. Given our constraints of limited time and resources, our focus will be on comparing the safety aspects of the most and least complex widely used control algorithms: Proportional-Integral-Derivative (PID) and Model Predictive Control (MPC).

Within the AD software stack, two key components significantly impact safety: control algorithms and motion planning. Control algorithms are responsible for translating high-level motion plans into real-time steering, acceleration, and braking commands for the vehicle. Meanwhile, motion planning determines the optimal path for the vehicle to navigate while adhering to traffic rules and avoiding obstacles.

This thesis undertakes a comparative analysis of PID and MPC algorithms within the context of Adaptive Cruise Control (ACC), a fundamental technology for maintaining safe distances from preceding vehicles. Additionally, we will explore the implementation of these algorithms for obstacle avoidance, a critical function for ensuring safety in dynamic driving scenarios.

To facilitate this comparative analysis, we will utilize the CARLA Simulator, which provides a realistic and adaptable virtual environment for testing and evaluating autonomous vehicle control systems. Python, a versatile and widely used programming language, will serve as the primary development platform within the Ubuntu operating system.

By thoroughly evaluating the performance of PID and MPC algorithms in both

ACC and obstacle avoidance scenarios within a HARA framework, our thesis aims to highlight the differences between these algorithms in the context of autonomous vehicle control. The outcomes of our study will unveil distinct strengths and limitations of each algorithm across various driving scenarios, providing valuable insights that set this research apart from existing efforts. These insights have the potential to inform more effective selection and optimization of control strategies aimed at enhancing safety in autonomous vehicles, thereby contributing to advancements in the field.

The forthcoming chapters in this thesis are meticulously structured to provide a comprehensive exploration of the research topic. The Background chapter will offer a thorough examination of relevant theories, concepts, and technological advancements pertinent to autonomous vehicles, control algorithms, and safety standards. Following this, the Architecture and Methodology chapter will delineate the overarching framework and methodological approach employed in the research, offering insights into the system architecture and research methodology. Subsequently, the Implementation chapter will delve into the practical aspects of executing the research, detailing the development process, software tools utilized, and challenges overcome. Finally, the Conclusion chapter will synthesize the key findings, implications, and future research directions, ensuring a cohesive culmination of the research endeavor.

Chapter 2

Background

In this chapter, we lay the foundation for understanding the intricacies of control and motion planning algorithms in the context of autonomous vehicles operating within ASIL HARA scenarios. Autonomous vehicles represent a paradigm shift in transportation, promising safer and more efficient mobility solutions. Central to their operation are sophisticated control algorithms and motion planning strategies that enable vehicles to perceive their environment, make decisions, and navigate autonomously.

Firstly, we delve into the fundamentals of vehicle dynamics, exploring the principles governing the movement and behavior of vehicles. Understanding vehicle dynamics is crucial for designing effective control systems that can stabilize the vehicle and optimize its performance in diverse driving conditions.

Next, we provide an overview of control theory, which forms the theoretical framework for designing control algorithms in autonomous vehicles. Control theory encompasses a range of methodologies for regulating the behavior of dynamical systems, offering insights into how to design controllers that can achieve desired objectives while accounting for uncertainties and disturbances.

Furthermore, we examine the basics of motion planning algorithms, which play a vital role in determining the trajectory of autonomous vehicles as they navigate through their environment. Motion planning algorithms generate feasible and safe paths for vehicles to follow, considering factors such as obstacles, traffic rules, and vehicle dynamics constraints.

Lastly, we introduce the ASIL (Automotive Safety Integrity Level) and HARA (Hazard Analysis and Risk Assessment) standards, which provide a systematic approach to ensuring the safety and reliability of autonomous vehicles. ASIL and HARA standards guide the development process, helping engineers identify and mitigate potential hazards arising from system malfunctions and external factors.

By exploring these foundational concepts, this chapter sets the stage for the subsequent discussion on control and motion planning algorithms in autonomous vehicles within ASIL HARA scenarios.

2.1 Vehicle Dynamics

In this section, we delve into the intricate dynamics governing the motion and behavior of vehicles. Understanding vehicle dynamics is paramount for the design, control, and optimization of autonomous vehicles, as it directly impacts their stability, maneuverability, and overall performance.

2.1.1 Tire Forces and Moments

Tires play a crucial role in vehicle dynamics by providing traction and transmitting forces and moments between the vehicle and the road surface. The interaction between the tire and the road can be modeled using various tire models, such as the Pacejka magic formula.

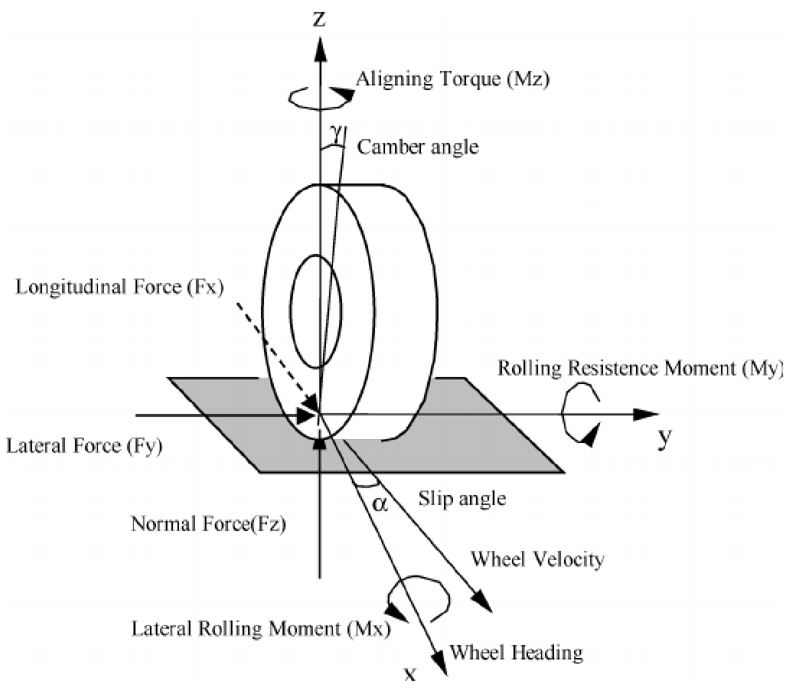


Figure 2.1: Tire Forces and Moments

One of the key relationships in vehicle dynamics is the relationship between tire forces and moments. For example, the lateral force F_y generated by the tire can be related to the slip angle α and the vertical load F_z as follows:

$$F_y = C_\alpha \alpha \cdot F_z \tag{2.1}$$

where C_α is the cornering stiffness of the tire.

Similarly, the longitudinal force F_x generated by the tire can be related to the longitudinal slip β and the vertical load F_z as follows:

$$F_x = C_\beta \beta \cdot F_z \tag{2.2}$$

where C_β is the longitudinal stiffness of the tire.

2.1.2 Bicycle Model

The bicycle model is a simplified representation of vehicle dynamics that is commonly used in control and motion planning algorithms for autonomous vehicles. It does work in a consideration where the left and right sides of the tires behave perfectly in the same way. In the bicycle model, the vehicle is represented as a single track with two wheels, allowing for modeling of both longitudinal and lateral dynamics.

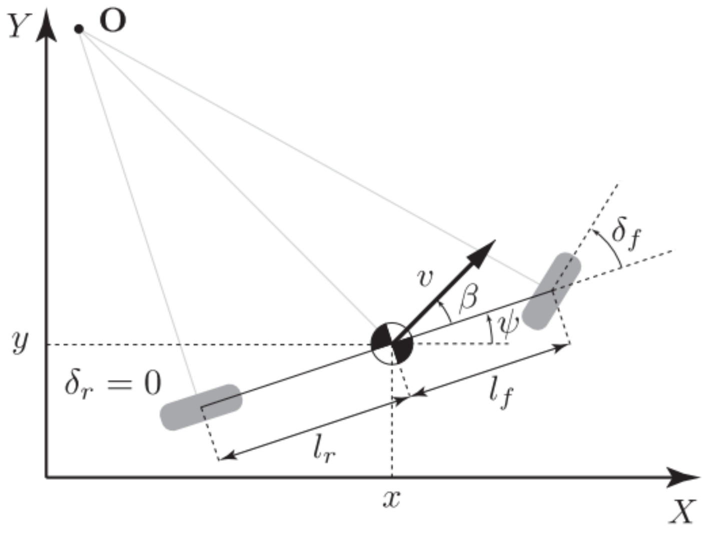


Figure 2.2: Bicycle Model

The equations of motion for the bicycle model can be expressed as follows:

$$\dot{x} = v\cos(\theta + \beta) \tag{2.3}$$
$$\dot{y} = v\sin(\theta + \beta) \tag{2.4}$$
$$\dot{\theta} = \frac{v\cos(\beta)\tan(\delta)}{L} \tag{2.5}$$
$$\dot{v} = a \tag{2.6}$$

where

- x and y: coordinates of the vehicle's center of mass
- θ: yaw angle (heading angle) of the vehicle
- v: velocity
- β: sideslip angle
- δ: steering angle
- L: distance between the front and rear axles (wheelbase)
- a: longitudinal acceleration

The bicycle model provides a simplified yet effective representation of vehicle dynamics for use in control and motion planning algorithms, allowing for efficient simulation and optimization of vehicle behavior in various driving scenarios.

2.1.3 Longitudinal and Lateral Motion

Longitudinal motion refers to the forward or backward movement of a vehicle along its longitudinal axis, which runs parallel to its direction of travel. This type of motion involves changes in speed or velocity, commonly referred to as acceleration or deceleration. It is influenced by various factors such as engine propulsion, aerodynamic drag, rolling resistance, road grade, and braking. Precise modeling of these forces is crucial for autonomous control systems to ensure smooth and efficient vehicle operation.

The longitudinal movement governed by the following equation:

$$m \cdot \dot{v} = F_{\text{prop}} - (F_{\text{drag}} + F_{\text{rolling}} + F_{\text{grade}}) - F_{\text{brake}} \tag{2.7}$$

where:

- m is the vehicle mass (kg)
- $\dot{v}$ is the longitudinal acceleration (m/s^2)
- F_{prop} is the engine propulsion force (N)
- F_{drag} is the aerodynamic drag force (N)
- F_{rolling} is the rolling resistance force (N)
- F_{grade} is the force due to road grade (N) (positive uphill, negative downhill)

- F_{brake} is the braking force applied to the wheels (N)

For autonomous control, precise modeling of these forces is essential. It involves incorporating drag coefficients, rolling resistance parameters, and accurate engine torque characteristics for reliable performance. Additionally, factors like wind gusts and varying road inclines must be considered to ensure safe operation.

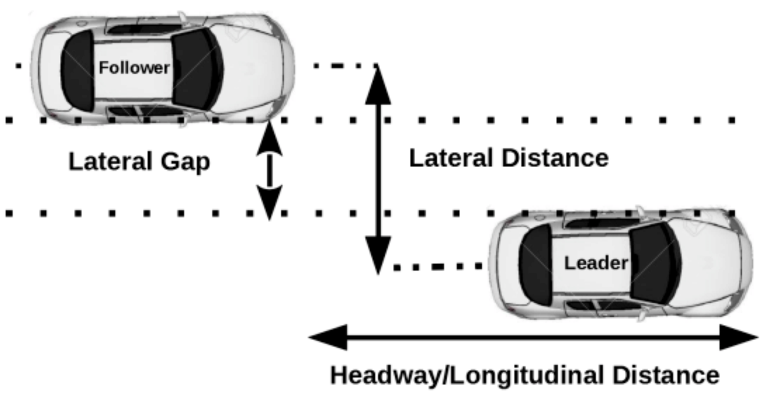


Figure 2.3: Lateral and Longitudinal distances.

Lateral movement is crucial for executing maneuvers such as turning corners, changing lanes, and avoiding obstacles on the road. Factors influencing lateral movement include steering inputs, tire forces, road conditions, and vehicle dynamics. Precise control of lateral forces is paramount, especially for ASIL HARA safety levels in autonomous driving. This necessitates accurate tire modeling considering factors like tire slip angle, lateral force saturation, and camber thrust. The control system must manage steering inputs to maintain stability during maneuvers and ensure the vehicle adheres to the planned trajectory.

The equation governing lateral dynamics is:

$$I_z \cdot \ddot{\psi} = M_z \tag{2.8}$$

where:

- I_z is the vehicle's yaw moment of inertia (kg·m^2)
- $\ddot{\psi}$ is the yaw acceleration (rad/s^2)
- M_z is the total moment acting around the vehicle's yaw axis (N·m)

Understanding and controlling both longitudinal and lateral motions are critical for ensuring the stability, maneuverability, and safety of vehicles, particularly in autonomous driving systems where precise control is indispensable.

2.1.4 Vehicle Stability

Vehicle stability is a critical aspect of vehicle dynamics, especially for autonomous vehicles where maintaining control is essential for safe operation. Even when an autonomous vehicle follows a planned trajectory, external factors and inherent system limitations can impact its stability. These factors include; mass distribution, suspension geometry, and the sophistication of control systems. To mitigate the effects of those factors it is necessary to minimize the path track and heading errors.

Even it's important to keep the vehicle within the planned trajectory through a safe operation. When the vehicle deviates from the path due to unforeseen circumstances, the control system should make informed decisions to ensure the safety of occupants and surrounding vehicles. Achieving this requires a deep understanding of the errors that can occur. Two key errors impacting stability are cross-track error and heading(yaw) error.

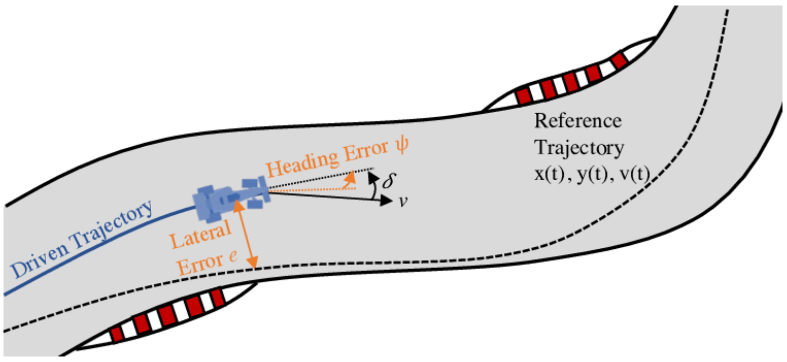


Figure 2.4: Heading Error.

The heading errors' main aspect is the yaw moment, which indicates the vehicle's tendency to rotate around its vertical axis. Yaw stability, closely linked to factors such as the positioning of the center of gravity and the distribution of mass within the vehicle, is crucial for maintaining control. It is quantified by the measurement of yaw heading error, which indicates the deviation of the vehicle's heading from its intended direction of travel. Minimizing yaw heading error is crucial for precise navigation and trajectory tracking, ensuring that the vehicle stays on course.

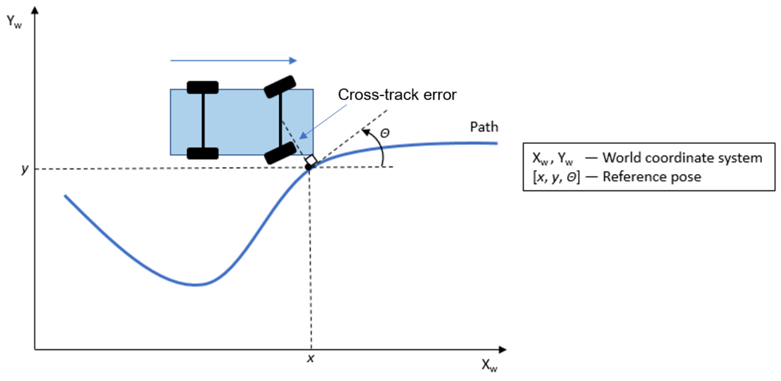
The heading error, typically denoted as ψ, represents the deviation between the vehicle's actual heading angle δ and its desired or reference heading angle δ_{ref}. Mathematically, the heading error formula can be expressed as:

$$\psi = \delta_{\text{ref}} - \delta \tag{2.9}$$

where:

- ψ is the heading error;

- δref is the desired or reference yaw angle;
- δ is the actual yaw angle (heading angle) of the vehicle.



Y_w
X_w
y
x
Cross-track error
Θ
Path
X_w, Y_w — World coordinate system
$[x, y, \Theta]$ — Reference pose

Figure 2.5: Cross Track Error.

Furthermore, the cross-track error (CTE) plays a pivotal role in assessing stability. CTE measures the lateral deviation of the vehicle from its desired trajectory or path. Minimizing CTE is essential for maintaining effective lane-keeping and trajectory tracking performance, ensuring that the vehicle remains aligned within its lane or adheres to a specified path accurately.

The cross-track error (CTE), denoted as e_{CTE}, quantifies the lateral deviation between the vehicle's actual position and its desired trajectory or path. Mathematically, the cross-track error formula can be expressed as:

$$e_{\text{CTE}} = y_{\text{actual}} - (y_{\text{desired}} + d \cdot \sin(\psi_{\text{desired}} - \psi_{\text{actual}})) \tag{2.10}$$

where;

- e_{CTE} is the cross-track error.
- y_{actual} is the actual lateral position of the vehicle.
- y_{desired} is the desired lateral position along the planned trajectory or path.
- d is the perpendicular distance between the vehicle's position and the desired trajectory.
- ψ_{desired} is the desired yaw angle (heading angle) of the vehicle along the planned trajectory, and
- ψ_{actual} is the actual yaw angle (heading angle) of the vehicle.

In summary, the field of vehicle dynamics encompasses a diverse array of phenomena and principles that govern the motion and behavior of vehicles. A thorough understanding of these dynamics, coupled with knowledge of their underlying equations, empowers engineers to develop control systems for autonomous vehicles that not only enhance efficiency but also elevate performance and safety standards on the road.

2.2 Control Dynamics

Control theory is a field of engineering and mathematics that deals with the behavior of dynamical systems. A dynamical system is one that evolves over time according to a set of rules or equations. These systems can be described using mathematical models, such as differential equations or difference equations. The main goal of control theory is to design controllers that manipulate inputs to the system in order to regulate its outputs.

2.2.1 Control theory in autonomous vehicles

As in other areas, control theory plays a critical role in the development and operation of autonomous vehicles, representing a vital component of their functionality and safety. In the context of autonomous vehicles, control theory is indispensable for orchestrating the intricate interactions between various subsystems, sensors, actuators, and environmental factors to ensure safe and efficient operation.

Autonomous vehicles, also known as self-driving cars, rely heavily on control theory to navigate complex and dynamic environments autonomously. These vehicles must perceive their surroundings, interpret sensory data, plan optimal trajectories, and execute control actions in real-time to safely transport passengers from one location to another. Control theory provides the mathematical foundation and algorithms necessary to achieve these objectives.

For instance, in autonomous vehicle navigation, control theory is employed to design controllers that regulate steering, acceleration, and braking inputs based on sensor feedback and high-level commands. These controllers must account for factors such as vehicle dynamics, road conditions, traffic patterns, and obstacle avoidance to ensure smooth and safe operation.

Moreover, control theory facilitates the integration of various control algorithms and decision-making processes within autonomous vehicle systems. This includes reactive control strategies for immediate responses to unforeseen events, as well as optimal control techniques for long-term planning and trajectory optimization. By leveraging control theory principles, autonomous vehicles can adapt to diverse driving scenarios, minimize energy consumption, optimize passenger comfort, and enhance overall performance and safety

2.2.2 Reactive Control in Autonomous Vehicles

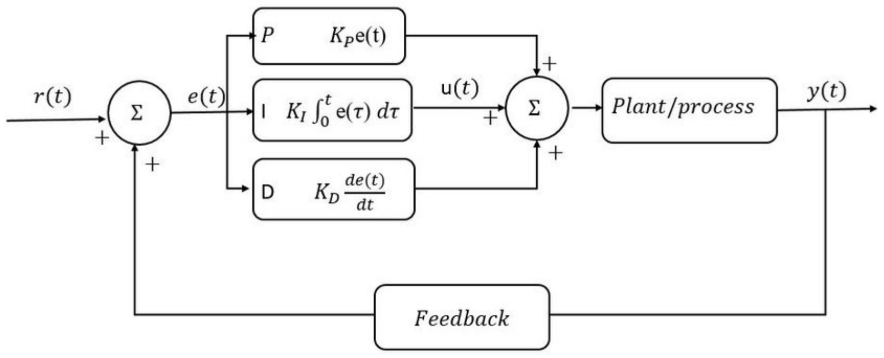


Figure 2.6: Example of Reactive Control.

Reactive control in the context of autonomous vehicles refers to a control strategy that enables real-time response to immediate sensory input without explicit reference to a pre-defined global plan. Instead of relying on complex predictive models of the environment, reactive control algorithms make decisions based solely on the vehicle's current sensor readings and local perceptions. This approach allows autonomous vehicles to react quickly to changing road conditions, dynamic obstacles, and unexpected events without the need for extensive planning or deliberation.

In autonomous vehicles, reactive control plays a crucial role in various aspects of driving, including obstacle avoidance, lane keeping, adaptive cruise control, and collision mitigation. By continuously monitoring sensor data such as lidar, radar, and cameras, reactive control algorithms can adjust vehicle trajectory, speed, and behavior to ensure safe and efficient navigation through complex environments.

2.2.3 Proportional-Integral-Derivative (PID) Control

PID control is one of the most widely used feedback control algorithms in various engineering applications due to its simplicity and effectiveness in regulating system behavior. The PID controller computes the control signal based on three terms: proportional, integral, and derivative, each contributing to the overall control action.

Proportional (P) Term

The proportional term in a PID controller generates a control signal that is proportional to the current error between the desired setpoint and the actual output. Mathematically, it is represented as:

$$P(t) = K_p \cdot e(t) \tag{2.11}$$

where;

- $P(t)$ is the proportional term of the control signal at time t.
- K_p is the proportional gain.
- $e(t)$ is the error at time t, calculated as the difference between the desired setpoint and the actual output.

The proportional term provides immediate response to changes in the error signal, ensuring that the system quickly approaches the desired setpoint. However, it alone may result in steady-state error, where the system settles at a non-zero error value.

Integral (I) Term

The integral term accounts for the cumulative sum of past errors over time and aims to eliminate steady-state error. It integrates the error signal over time and applies a control action to reduce the accumulated error. Mathematically, the integral term is given by:

$$I(t) = K_i \int_0^t e(\tau)\, d\tau \tag{2.12}$$

where;

- $I(t)$ is the integral term of the control signal at time t.
- K_i is the integral gain.
- $e(\tau)$ is the error at time τ.
- The integral is evaluated over the interval from 0 to t.

The integral term becomes increasingly significant for sustained errors over time, effectively driving the system towards the desired setpoint. However, it can also introduce instability if not properly tuned, leading to overshoot or oscillations.

Derivative (D) Term

The derivative term anticipates the future trend of the error signal by measuring its rate of change. It dampens the control action based on the rate of change, thus improving system stability and reducing overshoot. Mathematically, the derivative term is expressed as:

$$D(t) = K_d \frac{de(t)}{dt} \tag{2.13}$$

where:

- $D(t)$ is the derivative term of the control signal at time t,
- K_d is the derivative gain,
- $\frac{de(t)}{dt}$ represents the derivative of the error with respect to time.

The derivative term helps in smoothing out rapid changes in the error signal, thereby enhancing the transient response of the system while minimizing overshoot and oscillations.

PID Control Signal

The overall control signal $u(t)$ generated by the PID controller is the sum of the proportional, integral, and derivative terms:

$$u(t) = P(t) + I(t) + D(t) \tag{2.14}$$

Tuning the PID controller involves adjusting the proportional, integral, and derivative gains (K_p, K_i, and K_d) to achieve desired system performance, such as stability, responsiveness, and minimal steady-state error. Proper tuning is essential to optimize the controller's performance under various operating conditions and disturbances.

PID control finds widespread application in systems requiring precise control, such as temperature regulation, speed control of motors, robotics, process control, and automotive control systems. Its versatility, simplicity, and effectiveness make it a fundamental tool in control engineering.

2.2.4 Optimal Control

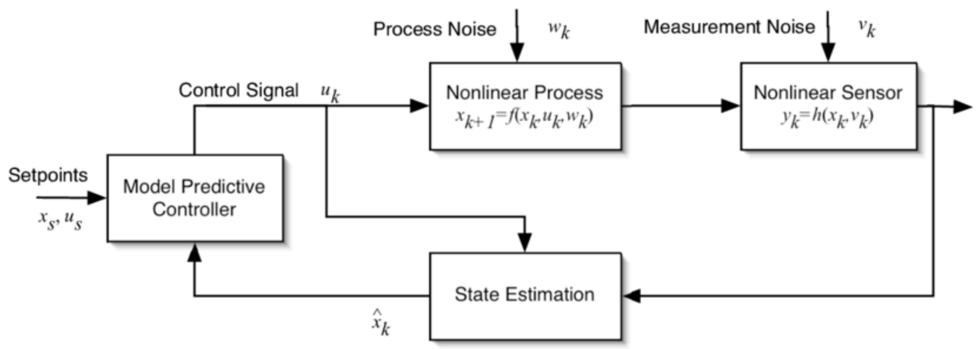


Figure 2.7: Example of Optimal Control.

Optimal control is a branch of control theory that focuses on finding control inputs that optimize a certain performance criterion. Unlike reactive control strategies, which respond directly to current sensory inputs, optimal control algorithms consider future states and system dynamics to determine the most suitable control action. By formulating and solving optimization problems, optimal control aims to minimize or maximize a defined objective function while satisfying system constraints.

In the context of autonomous vehicles, optimal control plays a crucial role in trajectory planning, path optimization, and energy management. By optimizing control inputs over a finite or infinite time horizon, optimal control algorithms can improve vehicle efficiency, reduce fuel consumption, and enhance safety.

2.2.5 Model Predictive Control (MPC)

Model Predictive Control (MPC) is an advanced control strategy that utilizes a dynamic model of the system to predict future behavior and optimize control actions over a finite time horizon. Unlike traditional control methods that compute control inputs in real-time, MPC solves an optimization problem at each time step to determine the best sequence of control inputs that minimize a cost function while satisfying system constraints.

In the context of autonomous vehicles, MPC enables proactive decision-making by considering the vehicle's dynamics, environmental conditions, and mission objectives. By incorporating predictive models of the vehicle's motion and external factors such as traffic and road conditions, MPC can generate optimal control commands that anticipate future events and optimize vehicle behavior accordingly.

Nonlinear Model Predictive Control (NMPC)

Nonlinear Model Predictive Control (NMPC) extends the principles of MPC to systems with nonlinear dynamics and constraints. Unlike linear MPC, which relies on linearized models of the system, NMPC directly handles nonlinearities in the system dynamics and constraints, allowing for more accurate predictions and control actions.

NMPC formulates an optimization problem with nonlinear objective functions and constraints, which are typically solved using nonlinear optimization techniques such as gradient-based methods or numerical optimization algorithms. By considering the full nonlinear dynamics of the system, NMPC can achieve superior performance and robustness compared to linear MPC, especially in highly nonlinear systems like autonomous vehicles.

NMPC finds applications in various autonomous vehicle tasks, including trajectory tracking, obstacle avoidance, and vehicle stabilization. By leveraging accurate nonlinear models and sophisticated optimization algorithms, NMPC enables precise and agile control of autonomous vehicles in complex and dynamic environments.

Formulas and Explanations

The general formulation of an optimal control problem involves minimizing or maximizing an objective function J subject to system dynamics and constraints. Mathematically, it can be represented as:

$$\min_{u} J(x, u) \tag{2.15}$$

subject to:

$$x(k+1) = f(x(k), u(k)) \tag{2.16}$$

$$u_{\min} \leq u(k) \leq u_{\max} \tag{2.17}$$

Where:

- x is the state vector,
- u is the control input vector,
- J is the cost function,
- f is the system dynamics model,
- $u_{\min}$ and $u_{\max}$ are the lower and upper bounds on the control inputs, respectively.

For Model Predictive Control (MPC), the optimization problem is solved at each time step within a receding horizon framework. The control input sequence u^* that minimizes the cost function while satisfying constraints is calculated as:

$$u^* = \arg\min_{u} J(x, u) \tag{2.18}$$

subject to:

$$x(k+1) = f(x(k), u(k)) \tag{2.19}$$

$$u_{\min} \leq u(k) \leq u_{\max} \tag{2.20}$$

NMPC extends this framework to handle nonlinear dynamics and constraints directly, allowing for more accurate predictions and control actions in highly nonlinear systems.

These optimization problems are typically solved using numerical optimization techniques such as gradient-based methods, nonlinear programming, or convex optimization algorithms, depending on the complexity of the system dynamics and constraints.

2.2.6 Control Algorithms

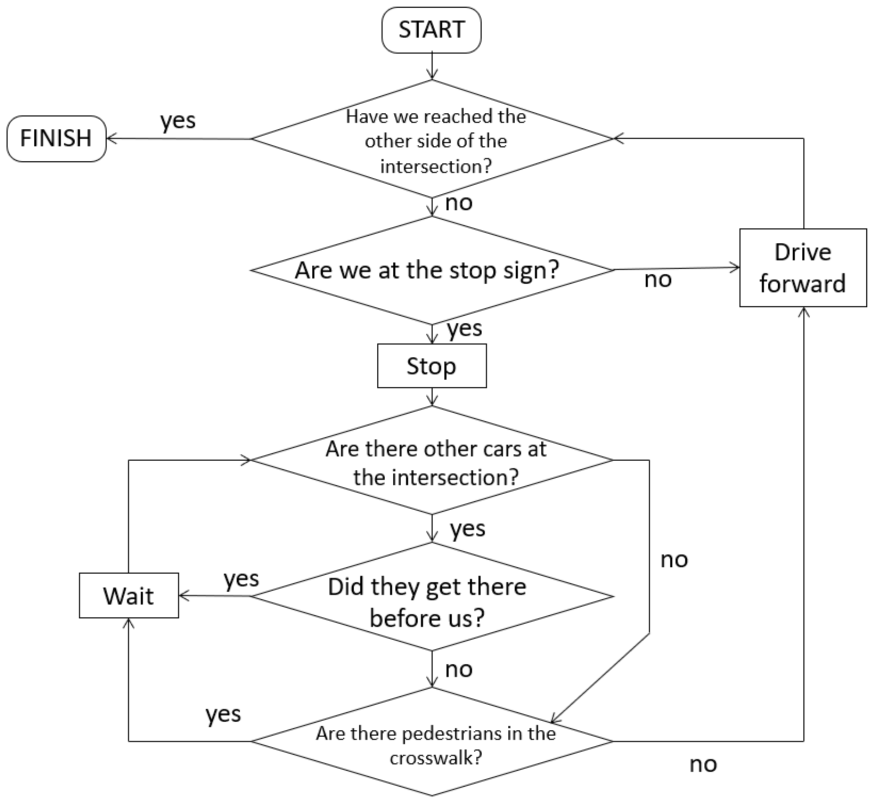

START
Have we reached the other side of the intersection?
yes
FINISH
no
Are we at the stop sign?
no
Drive forward
yes
Stop
Are there other cars at the intersection?
yes
no
Wait
yes
Did they get there before us?
no
yes
Are there pedestrians in the crosswalk?
no

Figure 2.8: Example of Autonomous Vehicle Flowchart.

Control algorithms are computational procedures used to implement control strategies in practice. These algorithms calculate control signals based on feedback from sensors and desired setpoints, enabling real-time adjustment of system inputs to achieve desired outcomes. There are various types of control algorithms, each suited to different types of systems and control objectives.

One of the key challenges in autonomous vehicle control is the need to balance competing objectives, such as safety, comfort, and efficiency, while navigating through complex and dynamic environments. Control algorithms must be robust to uncertainties in sensor measurements, road conditions, and the behavior of other road users. They should also be capable of adapting to rapidly changing situations and making decisions quickly and reliably.

Control algorithms in autonomous vehicles often employ a combination of reactive and predictive approaches. Reactive control strategies enable immediate responses to local sensory input, allowing the vehicle to react quickly to obstacles and unexpected events. On the other hand, predictive control techniques use models of the vehicle dynamics and environment to anticipate future states and plan optimal trajectories.

Advanced control algorithms like model predictive control (MPC) are increasingly being deployed in autonomous vehicles to address these challenges. MPC takes into account the vehicle's dynamics, environmental constraints, and desired objectives to optimize control actions over a finite time horizon. By considering future states and system constraints, MPC enables smoother and more efficient vehicle operation while ensuring safety and comfort.

Development steps of Control Algorithm

Control algorithms are typically developed through a systematic process that involves several key steps. Initially, engineers analyze the requirements and objectives of the control system, considering factors such as system dynamics, performance criteria, and operating constraints. Based on these requirements, they design mathematical models that describe the behavior of the system and its interactions with the environment.

Once the models are established, engineers proceed to design control strategies that achieve the desired system behavior. This often involves selecting appropriate control architectures, such as feedback or feedforward control, and choosing control algorithms that best suit the application requirements. For example, in the context of autonomous vehicles, control algorithms must be able to handle the complexities of real-world driving scenarios while ensuring safety and efficiency.

After selecting candidate algorithms, engineers conduct simulations and experiments to evaluate their performance under various conditions. This iterative process allows them to refine the algorithms and tune their parameters to optimize system performance. Additionally, engineers may employ techniques such as model-based design, where control algorithms are developed and validated using computer-aided design tools before implementation.

Once the control algorithms have been thoroughly tested and validated, they are implemented in software or hardware components of the autonomous vehicle's control system. During integration, engineers ensure that the algorithms interact seamlessly with other vehicle subsystems, such as perception, planning, and actuation, to enable coordinated and effective control.

Throughout the development process, engineers must also consider factors such as computational efficiency, real-time performance, and scalability to ensure that the control algorithms meet the requirements of the autonomous vehicle platform. Continuous testing, validation, and refinement are essential to ensure that the control algorithms operate reliably and safely in real-world driving conditions.

Overall, the development of control algorithms for autonomous vehicles requires a multidisciplinary approach, combining expertise in control theory, robotics, computer science, and automotive engineering. By leveraging advances in technology and research, engineers can continue to improve the capabilities of autonomous vehicles and pave the way for a future where safe and efficient autonomous transportation is a reality.

2.3 Basics of Motion Planning

Motion planning and algorithms are fundamental components of autonomous systems, enabling them to navigate through complex environments and reach desired goals safely and efficiently. These algorithms are designed to generate trajectories or paths for robots or vehicles by considering various factors such as obstacles, dynamic constraints, and environmental conditions.

One of the primary objectives of motion planning is to find a feasible path from a start configuration to a goal configuration while avoiding collisions with obstacles in the environment. These obstacles can be static, such as walls or obstacles, or dynamic, such as moving vehicles or pedestrians. Motion planning algorithms must account for uncertainties and disturbances in the environment to ensure robustness and reliability in real-world scenarios.

There are several types of motion planning algorithms, each with its own advantages and limitations. Some of the commonly used types include:

1. **Sampling-Based Algorithms**: These algorithms, such as Probabilistic Roadmap (PRM) and Rapidly-exploring Random Tree (RRT), construct a roadmap or tree structure by randomly sampling the configuration space of the robot. They then connect these samples to form a graph that represents feasible paths. Sampling-based algorithms are particularly suitable for high-dimensional configuration spaces and environments with complex obstacles.

2. **Optimization-Based Algorithms**: Optimization-based algorithms formulate motion planning as an optimization problem, where the objective is to minimize a cost function subject to constraints. Examples include trajectory optimization and optimal control techniques. These algorithms are often used for generating smooth and dynamically feasible trajectories, especially in continuous and time-varying environments.

3. **Grid-Based Algorithms**: Grid-based algorithms discretize the environment into a grid or lattice structure and search for collision-free paths using techniques like Dijkstra's algorithm or A* search. While these algorithms are computationally efficient and easy to implement, they may suffer from the curse of dimensionality in high-dimensional configuration spaces.

4. **Hybrid Approaches**: Hybrid approaches combine elements of different types of motion planning algorithms to leverage their respective strengths. For example, a hybrid algorithm may use sampling-based techniques to explore the configuration space and optimization-based techniques to refine the generated paths.

Implementing motion planning algorithms and control systems involves several steps, including representation of the environment, selection of appropriate algorithms, and integration with perception and control systems. Engineers use various data structures and algorithms to represent the environment, such as occupancy grids, voxel grids, or point clouds. They then choose suitable motion planning algorithms based on factors like computational complexity, real-time performance, and environmental characteristics.

Once selected, the chosen algorithms are implemented in software or embedded systems, often using programming languages like C++ or Python. Engineers conduct extensive testing and validation to ensure that the motion planning system operates reliably and safely in diverse scenarios, accounting for factors like sensor noise, localization errors, and uncertainties in the environment.

Overall, motion planning algorithms and control systems play critical roles in the autonomy and intelligence of robotic and autonomous systems, enabling them to navigate, interact with their surroundings effectively, and maintain safe and efficient operation in various domains.

2.3.1 Adaptive Cruise Control (ACC)

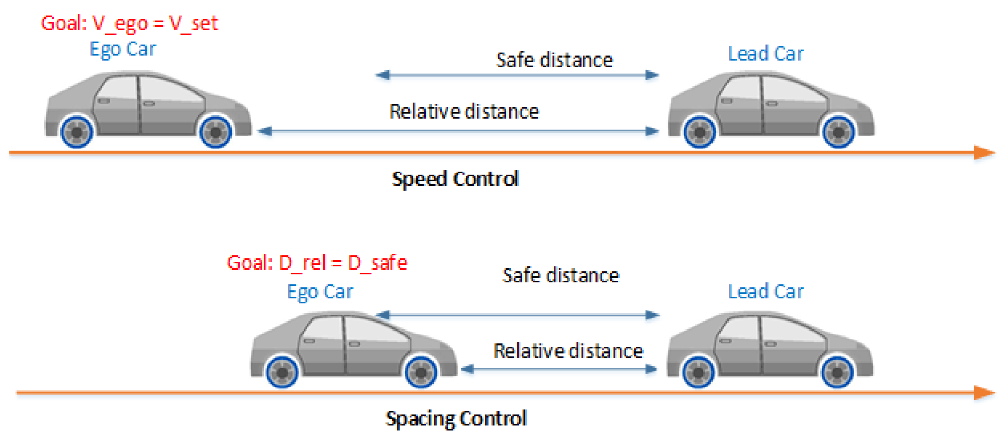


Figure 2.9: Adaptive Cruise Control.

Adaptive Cruise Control (ACC) is a sophisticated form of cruise control that automatically adjusts a vehicle's speed to maintain a safe following distance from the vehicle ahead. This system is pivotal in enhancing driving comfort and safety, especially in varying traffic conditions. ACC leverages control theory principles to dynamically adjust the vehicle's speed based on real-time sensor data.

Principles of ACC

ACC systems rely on sensors such as radar, lidar, and cameras to detect the distance and relative speed of the vehicle ahead. The primary goal of ACC is to maintain a predefined safe distance while ensuring smooth acceleration and deceleration. The system continuously monitors the traffic conditions and adjusts the throttle and brakes to maintain the desired gap.

Control Strategy of ACC

The control strategy of Adaptive Cruise Control (ACC) can be broadly categorized into three modes: speed control, distance control, and autonomous emergency braking.

- **Speed Control(SC)**: When there is no vehicle ahead within the ACC's range, the system functions like conventional cruise control, maintaining the set speed.
- **Distance Control(DC)**: When a vehicle is detected ahead, the ACC adjusts the speed to maintain a safe following distance, which is often set by the driver or predefined by the system.

The mode selection for the adaptive cruise control system is given by:

$$\text{mode} = \begin{cases} \text{VC} & \text{if } d > d_{\text{tar}} \text{ and } a_{\text{tar,VC}} \leq a_{\text{tar,SC}} \\ \text{SC} & \text{if } d > d_{\text{tar}} \text{ and } a_{\text{tar,VC}} > a_{\text{tar,SC}} \\ \text{SC} & \text{if } d \leq d_{\text{tar}} \end{cases}$$

where:

- mode represents the current mode of the adaptive cruise control system.
- d is the current distance to the vehicle ahead.
- d_{tar} is the target distance to the vehicle ahead.
- $a_{\text{tar,VC}}$ is the target acceleration in vehicle control mode (VC).
- $a_{\text{tar,SC}}$ is the target acceleration in speed control mode (SC).

For safety reasons, the arrangement of Emergency Braking is an important aspect of ACC:

- **Emergency Braking**: In case a pedestrian crosses between the host and lead vehicle, the control algorithm must react safely and brake to avoid a collision.

The minimum deceleration distance for safety is given by:

$$x_{\text{dect,min}} = -\frac{v_0^2}{2a_{h,\text{min}}} + x_{\text{safe}} \tag{2.21}$$

where:

- $x_{\text{dect,min}}$ is the minimum deceleration distance.
- v_0 is the initial velocity.
- $a_{h,\text{min}}$ is the minimum deceleration (should be a maximum of 2 m/s^2 for safety).
- x_{safe} is the safety distance.

Control Algorithms in ACC

The control algorithms used in ACC typically involve Proportional-Integral-Derivative (PID) control or Model Predictive Control (MPC), depending on the system's complexity and desired performance. These algorithms help regulate the vehicle's speed and maintain a safe distance.

PID Control in ACC

In ACC, PID control is used to adjust the vehicle's speed based on the error between the desired distance and the actual distance to the vehicle ahead. The control signal $u(t)$ is computed as follows:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt} \tag{2.22}$$

where:

- $e(t) = d_{\text{desired}} - d_{\text{actual}}$ is the distance error at time t.
- d_{desired} is the desired following distance.
- d_{actual} is the actual distance to the vehicle ahead.
- K_p, K_i, and K_d are the proportional, integral, and derivative gains, respectively.

The proportional term $K_p e(t)$ provides an immediate response to the current error. The integral term $K_i \int_0^t e(\tau)d\tau$ addresses accumulated past errors to eliminate steady-state error. The derivative term $K_d \frac{de(t)}{dt}$ anticipates future errors by considering the rate of change of the error.

Model Predictive Control (MPC) in ACC

MPC is a more advanced control strategy used in ACC systems to handle complex and dynamic driving environments. MPC predicts future vehicle states and optimizes control inputs over a finite time horizon.

The MPC problem for ACC can be formulated as:

$$\begin{aligned} & \min_{u} \sum_{k=0}^{N-1} J(x(k), u(k)) \\ & \text{subject to:} \\ & x(k+1) = f(x(k), u(k)), \\ & u_{\min} \leq u(k) \leq u_{\max}, \\ & d_{\min} \leq d(k) \leq d_{\max}, \end{aligned} \tag{2.23}$$

where:

- $x(k)$ is the state vector at time step k.
- $u(k)$ is the control input vector at time step k.
- $J(x(k), u(k))$ is the cost function to be minimized, which typically includes terms for speed error, distance error, and control effort.
- $f(x(k), u(k))$ represents the system dynamics.
- $u_{\min}$ and $u_{\max}$ are the lower and upper bounds on the control inputs, respectively.
- $d_{\min}$ and $d_{\max}$ are the minimum and maximum allowable distances to the vehicle ahead, ensuring safety and comfort.

MPC iteratively solves this optimization problem at each time step, using the current state and predicted future states to determine the optimal sequence of control inputs. This approach allows the ACC system to proactively adjust the vehicle's speed, accounting for changes in traffic conditions and vehicle dynamics.

Formulas and Explanations

The objective function $J(x, u)$ in MPC is designed to balance multiple criteria, such as minimizing speed deviation, maintaining a safe distance, and reducing control effort. A typical cost function might be:

$$J(x,u) = \sum_{k=0}^{N-1} \left(Q_d(d_{\text{desired}} - d(k))^2 + Q_v(v_{\text{desired}} - v(k))^2 + Ru(k)^2 \right) \tag{2.24}$$

where:

- Q_d and Q_v are weights for the distance and speed errors, respectively.
- R is the weight for the control effort.
- $d(k)$ and $v(k)$ are the distance to the vehicle ahead and the vehicle's speed at time step k, respectively.
- v_{desired} is the desired speed.

By solving this optimization problem, MPC provides control inputs that ensure the vehicle maintains a safe following distance, adapts to traffic conditions, and achieves smooth and efficient driving behavior.

In summary, Adaptive Cruise Control (ACC) is a critical feature in autonomous vehicles, utilizing advanced control algorithms like PID and MPC to maintain safe and comfortable driving conditions. These algorithms enable real-time adjustments to the vehicle's speed, ensuring optimal performance in diverse driving scenarios.

2.4 ASIL and HARA Standards

Automotive Safety Integrity Level (ASIL) and Hazard Analysis and Risk Assessment (HARA) are fundamental concepts in automotive safety engineering, particularly in the development of advanced driver assistance systems (ADAS) and autonomous vehicles. These standards provide a systematic framework for identifying, analyzing, and mitigating potential hazards associated with automotive systems, thereby ensuring the safety and reliability of vehicles on the road.

ASIL Standards

The Automotive Safety Integrity Level (ASIL) classification, as defined by the ISO 26262 standard, serves as a crucial framework for assessing and managing the safety of automotive systems. The four ASIL levels—A, B, C, and D—categorize safety functions based on their potential risk to vehicle occupants and other road users.

ASIL A represents the lowest level of risk among the classifications, indicating safety functions where the risk of severe injury or harm to occupants and other road users is relatively low. These functions typically involve basic vehicle controls and operations, such as turn signal activation or interior lighting.

ASIL B signifies a moderate level of risk, where safety functions may involve more critical vehicle systems that could lead to minor injuries or accidents if they were to malfunction. Examples of ASIL B functions include adaptive cruise control and lane departure warning systems.

ASIL C denotes a higher level of risk, indicating safety functions that, if compromised, could result in significant injuries or fatalities. These functions often involve advanced driver assistance systems (ADAS) such as automatic emergency braking or blind-spot detection.

ASIL D represents the highest level of risk and requires the most stringent safety measures. Safety functions classified as ASIL D are those that, if they were to fail or malfunction, could lead to catastrophic consequences, including multiple fatalities or widespread property damage. Examples of ASIL D functions include autonomous driving systems and vehicle-to-vehicle communication systems.

HARA Standards

Hazard Analysis and Risk Assessment (HARA) is a systematic process for identifying and evaluating potential hazards associated with the operation of automotive systems. It involves analyzing various system components, functions, and failure modes to identify potential hazards, assessing their severity, and estimating the associated risk. HARA aims to ensure that safety goals are defined and met throughout the development lifecycle of automotive systems.

Implementation

The implementation of ASIL and HARA standards involves several key steps. First, the automotive system under consideration is thoroughly analyzed to identify potential hazards and failure modes. This analysis may involve techniques such as Failure Mode and Effects Analysis (FMEA) and Fault Tree Analysis (FTA) to systematically identify and evaluate potential risks.

The determination of ASIL for a particular safety function involves the assessment of three HARA key factors:

1. **Severity of Potential Injury:** This factor considers the potential harm or injury that could occur if the safety function were to fail. It evaluates the severity of injuries to vehicle occupants, pedestrians, and other road users in the event of a malfunction.

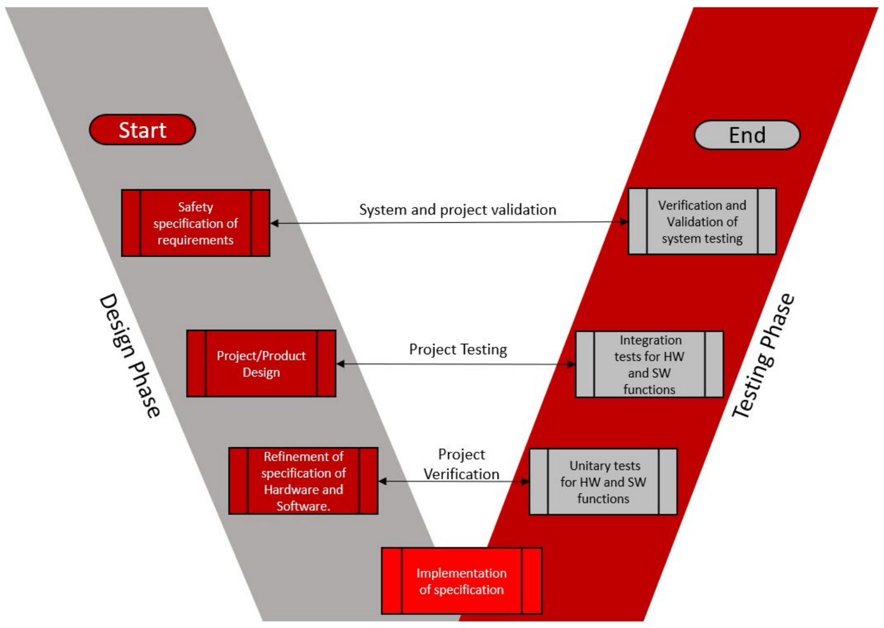

Start
End
Safety specification of requirements
System and project validation
Verification and Validation of system testing
Design Phase
Testing Phase
Project/Product Design
Project Testing
Integration tests for HW and SW functions
Refinement of specification of Hardware and Software.
Project Verification
Unitary tests for HW and SW functions
Implementation of specification

Figure 2.10: ISO development cycle.

2. **Probability of Exposure to the Hazard:** This factor assesses the likelihood of the safety function encountering hazardous situations during normal operation. It considers factors such as driving conditions, traffic density, and environmental factors that could increase the probability of exposure to potential hazards.

3. **Controllability of the Driver or System:** This factor examines the ability of the driver or the automated system to mitigate or control the hazard once it is detected. It evaluates the effectiveness of safety measures, such as warning signals, emergency braking, or vehicle maneuvering, in preventing or minimizing the consequences of a hazardous event.

By considering these factors, automotive researchers can assign appropriate ASIL levels to safety functions and implement the necessary safety measures to mitigate risks and ensure the overall safety and reliability of vehicles on the road.

Throughout the development process, rigorous verification and validation activities are conducted to ensure that the safety goals are effectively implemented and verified. This may involve testing the system under various operating conditions, simulating potential failure scenarios, and assessing the system's response to safety-critical events.

Overall, ASIL and HARA standards are essential components of the automotive safety engineering process, providing a systematic approach to identifying, analyzing, and mitigating potential hazards to ensure the safety and reliability of vehicles on the road.

Chapter 3

Architecture and Methodology

In this section, we transition from the theoretical aspects of the research to their practical implementation, detailing both the methodology and the architecture. Our approach is built upon the creation and testing of Hazard Analysis and Risk Assessment (HARA) scenarios within the CARLA simulator, executed on an Ubuntu Linux operating system using Python programming. We emphasize clean code practices throughout the development process, with code quality rigorously evaluated using Pylint to ensure adherence to Python PEP8 standards.

The methodology encompasses several key components. We begin with the setup and configuration of the CARLA simulation environment, which is meticulously detailed to provide clarity on the initial conditions and system requirements. Following this, we delve into the design and implementation of HARA scenarios, which serve as the foundation for our risk assessment and hazard analysis processes. Various control and motion planning algorithms are then applied within these scenarios, with each step of the process documented to ensure a comprehensive understanding of the experimental setup, data collection methods, and the criteria used for algorithm evaluation.

Our approach is iterative and structured, following Agile methodology. This allows for continuous improvement and refinement of the implemented solutions, ensuring that each iteration brings us closer to achieving our research objectives. The Agile framework facilitates adaptive planning, evolutionary development, early delivery, and continual improvement, encouraging flexible responses to change.

In addition to the methodology, we also outline the architecture that underpins our research framework (as shown in Figure 3.1). The architecture section details the overall system design, including the hardware and software components utilized in our experiments. To address compatibility issues, we run the application within a Conda environment, ensuring a consistent and isolated setup that avoids conflicts between different software dependencies. The CARLA environment and client are invoked from a shell script, streamlining the setup process and ensuring that all necessary components are correctly initialized.

We discuss the integration of the CARLA simulator with other tools and libraries, the configuration of the Ubuntu Linux operating system, and the use of Python for scripting and automation. The architectural design ensures a robust and scalable platform capable of supporting complex simulation scenarios and extensive data analysis.

Furthermore, we provide an in-depth examination of the software architecture, highlighting the modular design that allows for easy extension and maintenance. Each module is designed to perform specific tasks, from data acquisition and preprocessing to algorithm execution and result analysis. The interconnections between these modules are clearly defined, ensuring smooth data flow and efficient processing.

By combining a detailed methodology with a well-defined architecture, we establish a solid foundation for our research. This comprehensive approach not only facilitates the effective implementation of HARA scenarios within the CARLA simulator but also ensures that our system is scalable, maintainable, and capable of adapting to future research needs. Each aspect of the methodology and architecture is introduced in this section, providing a clear roadmap for replicating and building upon our work.

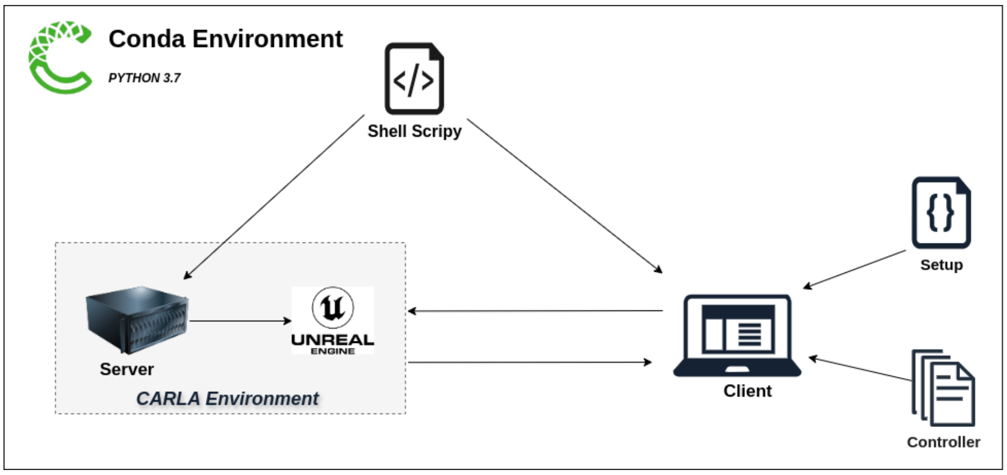


Figure 3.1: Project System Architecture.

3.0.1 Environment Management with Conda

Conda is a versatile package and environment management tool that streamlines the installation, organization, and utilization of software packages and dependencies across various environments. It is particularly useful for managing the Python environment and dependencies in this project due to CARLA's compatibility limitations with Python versions beyond 3.7.

Creating and Managing Environments

The primary advantage of Conda lies in its capability to create isolated environments. This feature allows for the creation of multiple Python environments with distinct package versions, enabling the simultaneous execution of diverse projects with unique requirements without encountering conflicts.

In our project, Conda is employed to manage the Python environment and dependencies due to CARLA's compatibility limitation with Python versions beyond 3.7. By utilizing Conda, we can effortlessly establish a Python 3.7 environment and install the requisite packages, ensuring seamless compatibility with CARLA.

The primary advantage of Conda lies in its capability to create isolated environments. This feature enables the creation of multiple Python environments with distinct package versions, facilitating the simultaneous execution of diverse projects with distinct requirements without encountering conflicts. Furthermore, Conda provides a straightforward command-line interface for environment management, simplifying tasks such as environment creation, activation, deactivation, and deletion.

For this study, Ubuntu is used as the operating system, and a Python environment version 3.7 or earlier is required for compatibility with CARLA. By utilizing Conda, we ensure that the development environment remains stable and consistent, preventing potential conflicts and issues related to package dependencies.

3.1 Python language and libraries

In this thesis, the development of control algorithms is accompanied by a strong emphasis on clean coding practices and effective environment management. This section outlines the tools and methodologies employed to ensure code quality, readability, and compatibility, focusing on Python as the primary programming language.

3.1.1 Python

Python's versatility and extensive library support make it an ideal choice for implementing and testing control algorithms. The following subsections detail the specific tools and libraries used to maintain code quality and manage the development environment.

3.1.2 Code Linting and Formatting

To maintain high code quality, standardization, and readability, several tools were integrated into the development workflow.

PEP8 Compliance

PEP8 is the Python Enhancement Proposal that outlines the conventions for Python code style. Adhering to PEP8 ensures that the code is consistent and readable, which is crucial for collaborative development and long-term maintenance. In this thesis, all Python scripts follow PEP8 guidelines, enhancing the overall code quality.

Pylint

Pylint is a static code analysis tool used to enforce coding standards and detect programming errors. By integrating Pylint into the development process, the code is continuously analyzed for potential issues such as coding standard violations, potential bugs, and other errors. This proactive approach helps maintain a high standard of code quality and prevents common programming mistakes, ensuring the robustness of the control algorithms developed in this work.

Black

Black is an uncompromising code formatter for Python that automatically formats code to comply with PEP8 standards. Using Black enhances code readability and consistency, making it easier for team members to understand and modify the code. By automating the formatting process, Black saves time and reduces the likelihood of formatting-related errors, allowing the focus to remain on the development and refinement of control algorithms.

Plotting

To visualize and compare the performance of PID and MPC controllers, plotting functions are utilized. These functions generate graphs that depict key performance metrics such as heading and cross track error. Visualizations are critical in this thesis as they provide an intuitive way to assess the effectiveness of different control strategies, facilitating the analysis and comparison of various approaches.

3.2 Carla Simulator

CARLA is an open-source, photorealistic simulator developed to facilitate the training, validation, and testing of autonomous driving algorithms. Built using C++ and Unreal Engine, CARLA provides a rich simulation environment with extensive features, including comprehensive control over actors, environmental condition adjustments, a versatile sensor suite, map generation capabilities, a flexible API, and server-client based communication.

The core simulation, including all control logic, rendering, physics, and actor properties, is handled by the CARLA Simulator. Additionally, CARLA offers a Python API module that allows users to interface with the simulator. In this architecture, the simulator acts as the server, while the client-server communication is managed through the Python API. Through the Python API, users can access

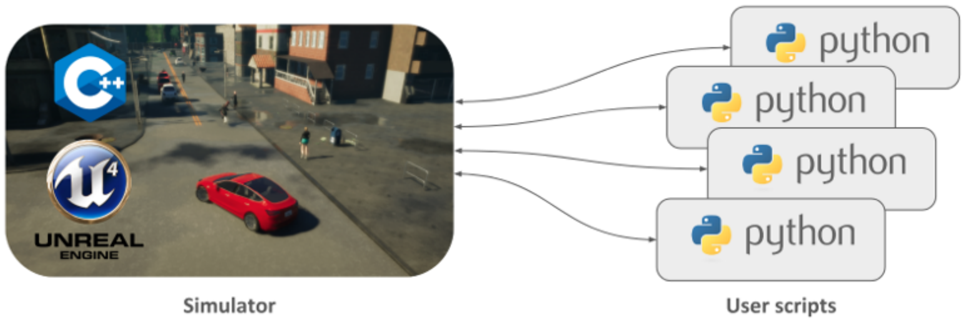


Figure 3.2: Carla Simulator.

most aspects of the simulation. Python scripts enable the retrieval of raw data from CARLA sensors attached to the ego vehicle, the processing of this data, the computation of control parameters, and the transmission of throttle, brake, and steering commands back to the simulator.

Client-side Python scripts are typically divided into two main components:

1. **World and client:** The client in CARLA is a crucial module that users run to request information or make changes within the simulation. Each client operates with a designated IP address and port number, communicating with the server through the terminal. Multiple clients can run simultaneously, but advanced management of these clients requires a deep understanding of CARLA and synchronization techniques.

To set up a client, you initiate it to communicate with the CARLA server. This setup allows the client to interact with the server, whether it's running locally or on a different network machine. The client's ability to load new maps, record simulations, and initialize the traffic manager showcases its versatility and control over the simulation environment.

The client-server link is pivotal because it enables dynamic and flexible control of the simulation world. Through the client, users can load different maps, adjust simulation settings, and manage various actors within the environment. This interaction is essential for real-time testing and validation of autonomous driving algorithms, as it allows for immediate adjustments and data retrieval.

In CARLA, the world is an object that represents the entire simulation. It serves as an abstract layer encompassing the primary methods needed to spawn actors, change weather conditions, retrieve the current state of the simulation, and more. There is only one world per simulation, and it is recreated whenever the map changes.

The client retrieves the world object, which can then be used to access and manipulate various elements within the simulation. This includes modifying weather, controlling vehicles, managing traffic lights, interacting with buildings, and navigating the map.

By linking the client and server, users can efficiently manage the simulation's environment and actors, making it possible to create realistic and varied scenarios for testing autonomous driving systems. This interaction is fundamental to the iterative development and validation process, ensuring that the algorithms perform reliably under diverse conditions.

2. **Synchronous and asynchronous mode**: CARLA operates on a client-server architecture where the server runs the simulation, and the client retrieves information and requests changes within the simulation. This section addresses the communication dynamics between the client and server, focusing on the modes of operation and their implications for simulation control and data integrity.

   By default, CARLA operates in asynchronous mode, where the server runs as fast as possible, handling client requests on the fly. Asynchronous mode is suitable for experimental setups or initial simulations where users can freely navigate the map and place actors. However, when generating training data or deploying an agent within the simulation, synchronous mode is recommended for greater control and predictability.

   In synchronous mode, the client, running Python code, dictates when the server updates. This mode is particularly important for maintaining synchrony between different elements, such as sensors, ensuring all sensors use data from the same simulation moment. If the client application is slow and the server does not wait, information overflow can occur, leading to data loss or mixing. Synchronous mode helps prevent this by making the server wait for a client tick, ensuring the client can manage the incoming data effectively.

In a multiclient architecture, only one client should control the ticks. The server treats each tick received as if it came from the same client, and multiple client ticks can create inconsistencies between the server and clients.

Switching between synchronous and asynchronous modes involves changing a boolean state in the simulation settings. Enabling synchronous mode ensures that the server waits for client ticks, providing better control over the simulation. However, if synchronous mode is enabled and a Traffic Manager is running, the Traffic Manager must also be set to synchronous mode to maintain consistency.

Using synchronous mode is particularly relevant for applications requiring synchronization between various sensors. For instance, GPU-based sensors like cameras often generate data with a slight delay, making synchrony crucial to ensure data accuracy. The world object in CARLA provides methods to make the client wait for a server tick or perform specific actions upon receiving a tick, ensuring that all simulation elements remain coordinated.

By maintaining synchrony in client-server communication, users can produce more reliable and consistent simulation results, crucial for developing and testing autonomous driving algorithms.

In addition to those two aspects, CARLA's rendering capabilities are critical to its simulation environment, contributing to the realism and immersion of virtual scenarios. Leveraging the power of Unreal Engine, CARLA offers a range of rendering options that can vary from epic quality to low quality. Rendering quality plays a vital role in simulating real-world environments accurately. High-quality graphics enhance the fidelity of the simulation, allowing for more detailed and visually rich scenes. This level of realism is essential for tasks such as perception training, where objects need to be accurately identified based on visual cues.

However, rendering quality can also be adjusted to meet specific requirements. Lower-quality rendering settings can reduce computational demands, making it possible to run simulations on less powerful hardware or at higher speeds. This flexibility allows users to optimize their simulation setup according to their computational resources and performance needs.

As a result, CARLA, as a simulation platform, offers unparalleled freedom and the capability to generate highly realistic scenarios compared to other simulators. Its advanced features and flexibility make it an ideal choice for implementing HARA scenarios in our study. By leveraging CARLA, we can create diverse and detailed

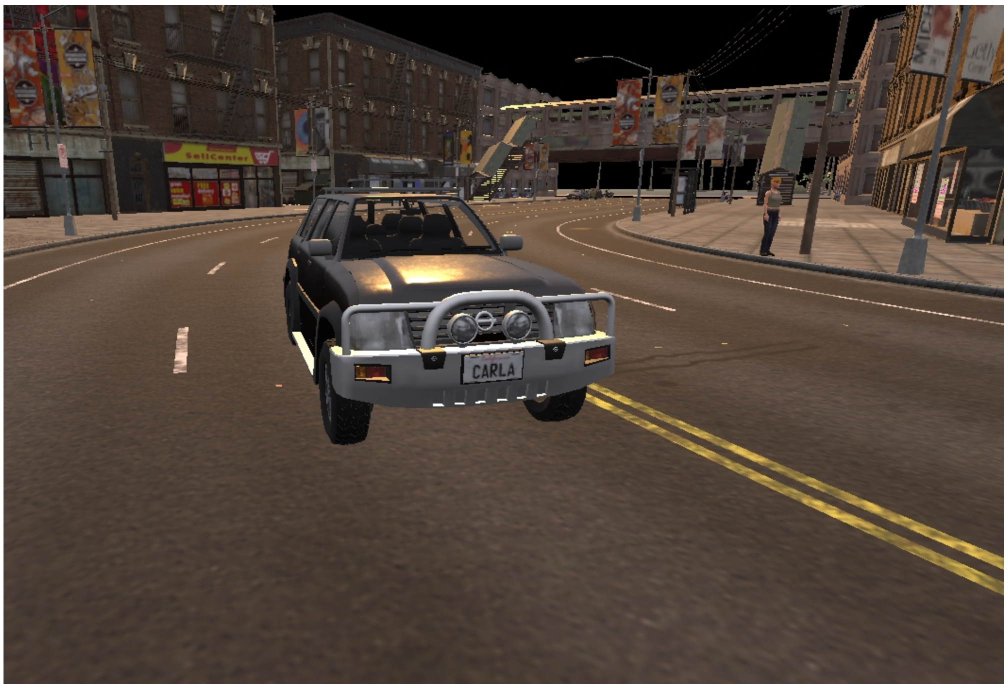


Figure 3.3: Low rendering quality of Carla Simulator.

driving environments that closely mimic real-world conditions, thus enabling a more accurate and robust evaluation of the safety performance of various control algorithms. This capability is crucial for our research, as it ensures that the scenarios tested are not only varied but also true to life, providing meaningful insights into the efficacy of PID and MPC algorithms under different driving conditions. Consequently, CARLA's comprehensive simulation environment is instrumental in our quest to enhance the safety protocols of autonomous vehicles through meticulous HARA analysis.

Lastly, in our study, we will use Synchronous mode to ensure consistent results in each iteration, as it is also suitable for the computer running the CARLA simulation. Rendering will be set to Low due to the computer's graphics capabilities.

Figure 3.4: Epic rendering quality of Carla Simulator.

3.2.1 Client to Carla Environment Localhost Communication for Efficient Interaction

This project adopts a modular design philosophy, separating the core controller logic from the simulation environment (e.g., CARLA). This approach offers several advantages:

Clear Separation of Concerns: Modular design promotes cleaner code by isolating the controller logic from the complexities of the simulator. This facilitates independent development and testing of the controller, reducing the risk of unintended consequences. Enhanced Reusability: The isolated controller module can be potentially reused with different simulators or even real hardware in the future, enhancing project flexibility.

To further enhance these benefits and facilitate collaboration, the project utilizes Git, a version control system, as described elsewhere in this document.

CARLA, the chosen simulation platform, leverages localhost communication for interaction between the controller and the simulation environment during development and testing. This communication facilitates several functionalities:

- Sensor Data Visualization: A separate application running on the same machine as CARLA acts as the client. The CARLA simulator acts as the server, continuously generating sensor data (e.g., camera images, LiDAR point clouds) from the simulated environment. The client connects to the CARLA server on localhost and receives a stream of sensor data in real-time, allowing for visualization and analysis of the vehicle's perception of the environment. Communication protocols like TCP or UDP can be employed depending on data format and real-time requirements.

- External Control of the Simulator: An external control application running on localhost can send control commands (e.g., steering commands, acceleration/deceleration) over localhost to the CARLA server. The server interprets these commands and modifies the behavior of the simulated vehicle accordingly. This setup facilitates the development and testing of external applications controlling the simulation. Depending on the desired level of control and real-time responsiveness, TCP or a custom binary protocol could be suitable options.

Localhost communication offers several benefits for development and testing:

Simplified Development and Testing: It allows for rapid development and testing of applications interacting with CARLA's data or controlling the simulation. Fast Data Transfer: Communication within the same machine ensures faster data transfer compared to using external networks. Secure Environment: Data exchange remains confined to the local machine, offering a secure testing environment. However, limitations also exist:

- Limited Accessibility: The client application can only access the CARLA server running on the same machine, limiting collaboration across different devices.
- Deployment Considerations: Localhost communication is primarily for development and testing. Real-world deployments might necessitate communication with external systems over a network, requiring adjustments to the communication setup.

In conclusion, the project's modular design and utilization of localhost communication with CARLA promote efficient development, testing, and collaboration. The modular design fosters clean, maintainable code, while localhost communication allows for rapid interaction with the simulation environment. These strategies pave the way for a well-structured and adaptable autonomous vehicle control system.

3.3 Linux Ubuntu Operating System and Shell Script

In this work, the Linux Ubuntu operating system was utilized due to its robust support for development environments and extensive library support. Ubuntu provides a stable and secure platform, making it an ideal choice for running simulations and managing software dependencies efficiently.

Ubuntu is particularly well-suited for the CARLA simulation environment, as it offers enhanced compatibility and performance. The open-source nature of Ubuntu provides more freedom to customize and optimize the system for specific simulation needs, ensuring a seamless and efficient workflow.

To streamline the workflow and enhance productivity, various shell commands were employed. Shell commands offer powerful capabilities for automating tasks, managing files, and executing scripts, all of which are crucial for handling complex simulation environments and data processing tasks.

Moreover, an interface was created to facilitate the use of shorter and more intuitive commands. This interface acts as a wrapper around commonly used shell commands, enabling quicker execution and reducing the potential for errors. By simplifying command syntax, the interface significantly improves efficiency and accessibility, especially for repetitive tasks.

This approach not only optimizes the interaction with the Linux operating system but also ensures that the development process remains smooth and user-friendly, thereby enhancing overall project management and execution.

The Project Shell Command Toolkit

As previously mentioned, the CARLA Simulator can be somewhat user-unfriendly, requiring numerous shell commands to execute even simple tasks. This becomes particularly tedious when writing long scripts or repeatedly testing functionality and partial code segments. To address this issue, a project toolkit named `luxad_toolkit.sh` was developed.

The `luxad_toolkit.sh` script simplifies and automates many of the repetitive and complex shell commands needed to run CARLA simulations. This toolkit enhances productivity by reducing the amount of manual command entry, thereby minimizing errors and saving valuable development time.

To activate the toolkit, it must be sourced into the Linux system each time it is used. This is achieved with the following command:

```

source luxad_toolkit.sh
```

By sourcing this script, the Linux operating system treats the commands within the script as if they were typed directly into the terminal. This makes the toolkit's functions immediately available for use, streamlining the workflow significantly.

Key benefits of using `luxad_toolkit.sh` include:

- **Efficiency**: Automates repetitive commands, reducing the time and effort required for each simulation run.
- **Consistency**: Ensures that the same commands are used each time, reducing the likelihood of human error.
- **Convenience**: Simplifies the process of running and testing CARLA simulations, making it more user-friendly and accessible.

Overall, the `luxad_toolkit.sh` script is a valuable addition to the development environment, enhancing the usability and efficiency of the CARLA simulation process.

`luxad_toolkit.sh` is implemented as follows:

```bash

#!/bin/bash

export LUXAD_TOOLKIT_VERSION="0.1.0"
export CARLA_ROOT_FOLDER=lux_ad_carla
export CARLA_BINARIES=$CARLA_ROOT_FOLDER/CarlaUE4/Binaries
export CARLA_PYTHON_API=$CARLA_ROOT_FOLDER/PythonAPI
export PYTHONPATH=$CARLA_PYTHON_API/carla/dist/carla-0.9.15-py3.7-linux-x86_64.egg:$CARLA_PYTHON_API/carla:$CARLA_PYTHON_API/carla/agents
```

The export commands are crucial for setting up path and versioning issues. They ensure the validity of the CARLA Simulator directory and the correct Python version. When the `source luxad_toolkit.sh` command is executed, the script first verifies the Python version, which should be 3.7 or an older version. The verification script is as follows:

```bash

function luxad_check_python
{
    python_version=$(python --version 2>&1)
    major_version=$(echo $python_version | cut -d ' ' -f 2 | cut -d '.' -f 1)
    minor_version=$(echo $python_version | cut -d ' ' -f 2 | cut -d '.' -f 2)

    if [ "$major_version" -eq 3 ] && [ "$minor_version" -eq 7 ]; then
        echo "Python version 3.7 is compatible with CARLA"
    else
        echo "Wrong Python version $python_version, please use 3.7"
    fi
}
```

In this script, the variables are:

- `python_version`: Checks the version of Python being used.
- `major_version`: Extracts the major version number.
- `minor_version`: Extracts the minor version number.
- These variables are then compared to the required 3.7 version. If the version matches, the toolkit can be activated.

If the version is correct, the script activates the toolkit as follows:

```

function luxad_activate_toolkit
{
    echo "Luxoft AD toolkit activated"
    chmod +x "$CARLA_BINARIES/Linux/CarlaUE4-Linux-Shipping"
}
```

As mentioned before, CARLA has issues due to `.egg` files. The export command aims to set this directory as the `PYTHON_PATH` for CARLA.

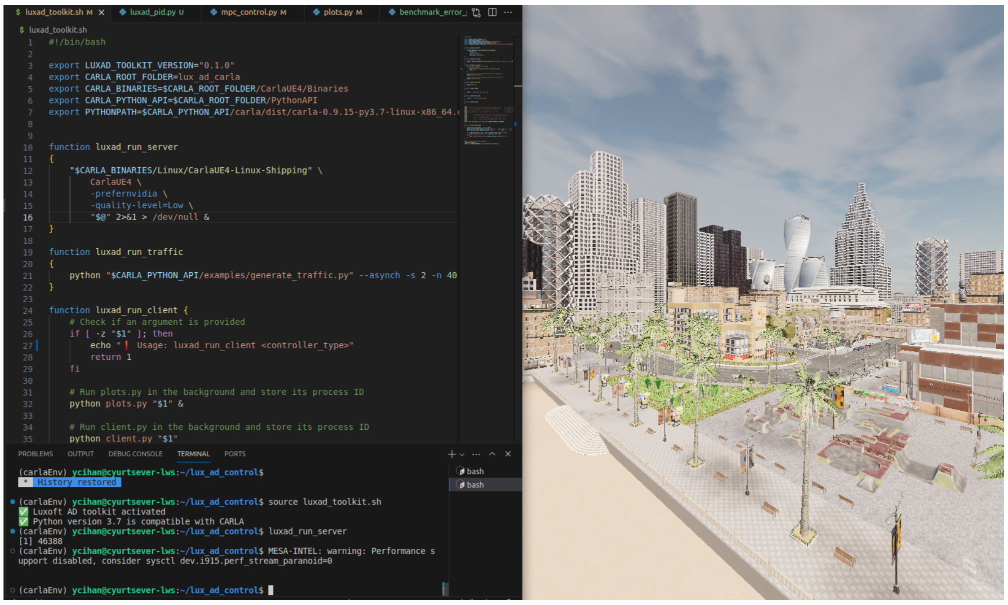
```

function luxad_run_server {
    "$CARLA_BINARIES/Linux/CarlaUE4-Linux-Shipping" \
        CarlaUE4 \
        -prefernvidia \
        -quality-level=Low \
        "$@" 2>&1 > /dev/null &
}

function luxad_run_client {
    # Check if an argument is provided
    if [ -z "$1" ]; then
        echo "Usage: luxad_run_client <controller_type>"
        return 1
    fi

    # Run plots.py in the background and store its process ID
    python plots.py "$1" &

    # Run client.py in the background and store its process ID
    python client.py "$1"
}
```

As previously mentioned, CARLA operates using a Server-Client communication model.

In the server command:

- `CARLA_BINARIES`: This is the root directory.
- The `CarlaUE4` is activated with:
  - `-prefernvidia`: Indicates that an Nvidia GPU is preferred.
  - `-quality-level=Low`: Sets the rendering quality to low to ensure performance.
  - The last part of the command ensures that the server process runs as a child process in the background.

These commands ensure that the server-side runs correctly with the specified options. After executing these commands, the server screen will open as shown in Figure 3.5.



```
#!/bin/bash

export LUXAD_TOOLKIT_VERSION="0.1.0"
export CARLA_ROOT_FOLDER=lux_ad_carla
export CARLA_BINARIES=$CARLA_ROOT_FOLDER/CarlaUE4/Binaries
export CARLA_PYTHON_API=$CARLA_ROOT_FOLDER/PythonAPI
export PYTHONPATH=$CARLA_PYTHON_API/carla/dist/carla-0.9.15-py3.7-linux-x86_64.e


function luxad_run_server
{
    "$CARLA_BINARIES/Linux/CarlaUE4-Linux-Shipping" \
        CarlaUE4 \
        -prefernvidia \
        -quality-level=Low \
        "$@" 2>&1 > /dev/null &
}

function luxad_run_traffic
{
    python "$CARLA_PYTHON_API/examples/generate_traffic.py" --asynch -s 2 -n 40
}

function luxad_run_client {
    # Check if an argument is provided
    if [ -z "$1" ]; then
        echo "❗ Usage: luxad_run_client <controller_type>"
        return 1
    fi

    # Run plots.py in the background and store its process ID
    python plots.py "$1" &

    # Run client.py in the background and store its process ID
    python client.py "$1"
```

```
(carlaEnv) ycihan@cyurtsever-lws:~/lux_ad_control$
 *  History restored

● (carlaEnv) ycihan@cyurtsever-lws:~/lux_ad_control$ source luxad_toolkit.sh
✅ Luxoft AD toolkit activated
✅ Python version 3.7 is compatible with CARLA
● (carlaEnv) ycihan@cyurtsever-lws:~/lux_ad_control$ luxad_run_server
[1] 46388
○ (carlaEnv) ycihan@cyurtsever-lws:~/lux_ad_control$ MESA-INTEL: warning: Performance s
upport disabled, consider sysctl dev.i915.perf_stream_paranoid=0


○ (carlaEnv) ycihan@cyurtsever-lws:~/lux_ad_control$ 
```

Figure 3.5: The CARLA (UE4) Server Interface.

The client part needs to be activated, which is responsible for sending commands via local communication. To activate the client application, the function `luxad_run_client` includes:

- First, the function checks if an argument is provided.
- If no additional command is provided after `luxad_run_client`, it sends the message: `"Usage: luxad_run_client <controller_type>"`, indicating that a controller type is required.
- By indicating PID, we run a PID controller.
- Alternatively, using the MPC command, we run a Model Predictive Controller.

These commands enable the client side to function correctly, ensuring effective communication with the server.

The CARLA Simulator API allows the creation of city traffic scenarios. The command for generating traffic can be seen as follows:

```

function luxad_run_traffic
{
    python "$CARLA_PYTHON_API/examples/generate_traffic.py" --asynch
   -s 2 -n 40 -w 0 &
}
```

This command means:

- `-asynch`: Runs the traffic API in asynchronous mode since only one client can run in synchronous mode at a time in CARLA.
- `-s 2`: Sets the seed number to produce the same scenario each time CARLA runs.
- `-n 40 -w 0`: Specifies that 40 vehicles will be produced for traffic and no pedestrians will be on the road.

The values and arrangements can change depending on the scenarios being produced. Thus, versioning of the code becomes important to keep track of different setups, which is managed with luxad_version function.Where the different scenarios and plot types can be differentiated.

3.3.1 Project File Directory; Modules and Submodules

```
Lux_ad_control
└─Lux_ad_carla
```

```
│
├── luxad_docs
│   ├── controller_luxoft.plantuml
│   ├── Error.csv
│   ├── MPC_Errors.csv
│   ├── PID_Errors.csv
│   └── records.log
└── src
    ├── benchmark_error_plot.py
    ├── client.py
    ├── constant.py
    ├── cutils.py
    ├── luxad_mpc.py
    ├── luxad_pid.py
    ├── luxad_toolkit.sh
    ├── mpc_control.py
    ├── mpc_utils.py
    ├── plots.py
    └── setup_luxoft.py
```

This project adopts a modular design philosophy, separating the core controller logic from the simulation environment (e.g., CARLA). This approach offers several advantages. It promotes cleaner code by fostering a clear separation of concerns. The controller module's development and testing become independent of the simulator, reducing the risk of unintended consequences. Furthermore, the isolated controller module can be potentially reused with different simulators or even real hardware in the future, enhancing project flexibility.

To further enhance these benefits and facilitate collaboration, the project utilizes Git, a version control system. Git offers several key functionalities:

- Tracking Changes: Git meticulously tracks all modifications made to the codebase, enabling developers to revert to previous versions if necessary. This ensures a safety net during development and experimentation.

- Branching and Merging: Git facilitates the creation of branches, allowing developers to work on independent features or bug fixes without affecting the main codebase. Once satisfied, these branches can be merged back into the main codebase, promoting efficient collaboration.

- Version Control History: Git maintains a comprehensive history of all changes made to the codebase. This history provides valuable insights into the project's evolution and can be crucial for debugging or identifying the origin of specific code sections.

The project's modular design translates into a well-organized Git repository structure. The controller code resides in a separate directory within the repository, potentially named "controller." The simulator submodule (e.g., CARLA integration code) might be located in a subdirectory within the controller directory, or even as a separate Git submodule altogether. This structure reflects the modular design and simplifies code management.

Utilizing Git alongside the modular design fosters several additional benefits:

- Collaboration: Git streamlines collaboration among developers by enabling them to work on different parts of the codebase simultaneously while maintaining a synchronized code history.
- Reproducibility: With Git, it's easy to recreate the project's state at any point in time, facilitating the reproducibility of results and debugging efforts.
- Backup and Disaster Recovery: Git serves as a robust backup solution, allowing for the recovery of lost or corrupted code.
- While adopting a modular design and Git might introduce some initial overhead regarding setup and learning curve, the long-term benefits in maintainability, collaboration, and code organization outweigh this overhead. This combination lays the groundwork for a well-structured, maintainable, and future-proof autonomous vehicle control system.

Chapter 4

Implementation

4.1 Client Implementation

The `Client` class is designed to manage the lifecycle of a simulation run using either a Model Predictive Controller (MPC) or a Proportional-Integral-Derivative (PID) controller. This class ensures that the simulation is initialized, executed, and cleaned up correctly, providing a robust framework for running various control algorithms within the CARLA simulation environment.

The `Client` class is implemented as a singleton to avoid multiple initializations, which could lead to inconsistent states or resource conflicts. The class structure is as follows:

- **___new___**: Ensures that only one instance of the `Client` class is created.
- **___init___**: Initializes the simulation setup and selects the appropriate controller based on the input argument.
- **_get_controller**: A class method that returns the controller class (either MPC or PID) based on the specified type.
- **run_simulation**: Manages the entire simulation process, including vehicle and sensor setup, waypoint navigation, and cleanup.

The singleton pattern is implemented using the ___new___ method, which ensures that only one instance of the `Client` class is created:

```python
class Client:
    _instance = None

    def __new__(cls, controller_type, *args, **kwargs):
        if not cls._instance:
            cls._instance = super().__new__(cls, *args, **kwargs)
            cls._instance._controller = cls._get_controller(
    controller_type)
        return cls._instance
```

In the __init__ method, the `SetupLuxad` class is instantiated to handle the environment setup, and the controller type is set:

```python
def __init__(self, controller_type):
    self.setup = SetupLuxad()
    self.control = self._controller()
```

The _get_controller method selects the appropriate controller class based on the input argument:

```python
@classmethod
def _get_controller(cls, controller_type):
    if controller_type == "MPC":
        return ControllerMPCLuxad
    elif controller_type == "PID":
        return ControllerLuxad
    else:
        raise ValueError("Invalid controller type. Please specify '
    MPC' or 'PID'.")
```

The `run_simulation` method manages the entire simulation process. Initially, it checks if a dummy vehicle is needed, which is used for emergency braking purposes. If the scenario includes the dummy vehicle, it will be spawned; otherwise, it will be skipped to avoid unnecessary resource allocation and potential simulation regeneration issues.

Then, the method handles vehicle spawning, sensor setup, and waypoint navigation, using the selected controller to drive the vehicle through the planned route. It also includes error handling to manage interruptions and ensure that resources are cleaned up properly:

```python
if constant.HAS_DUMMY_CAR:
    dummy_vehicle = self.setup.vehicle_spawn(constant.DUMMY_SPAWN_POS
   )
else:
    dummy_vehicle = None

try:
    vehicle = self.setup.vehicle_spawn(constant.CAR_SPAWN_POS)
    camera, spectator = self.setup.camera_spawn(vehicle)
    obstacle_sensor = self.setup.obs_sensor_spawn(vehicle)
    obstacle_sensor.listen(self.setup.on_obstacle_callback)
    sensor_objects = [spectator, camera, self.setup.world]
    self.control.drive_through_plan(self.setup.waypoint_list, vehicle
   , sensor_objects)
except (IndexError, KeyboardInterrupt):
    logging.warning("KEYBOARD INTERRUPT")
except RuntimeError:
    logging.warning("SPAWN ERROR, retry :)")
finally:
    self.setup.delete_objects_and_settings(vehicle, dummy_vehicle)
    logging.warning("done.\n")
```

4.2 Setup Environment and Objects

The `Setup` class is designed to configure the simulation environment, including the creation of the world, setting up waypoints, and spawning vehicles and sensors. This class ensures that the environment is correctly initialized and provides the necessary infrastructure for running the simulation.

4.2.1 World Creation

The `create_world` method establishes a connection between the client and server, ensuring communication is properly set up. It initializes the world to a specific map, "Town01", and sets the weather conditions as defined in the constants. This setup is crucial for maintaining a consistent simulation environment.

```python
def create_world(self):
    client = carla.Client("localhost", 2000)
    client.set_timeout(15.0)

    world = client.get_world()

    if "Town01" not in world.get_map().name:
        world = client.load_world("Town01")

    weather = constant.WEATHER
    world.set_weather(weather)
    logging.debug("The worlds is created")
    return world
```

4.2.2 Synchronized Settings

To ensure deterministic behavior and avoid asynchronous issues between the server and client, the `create_synch_settings` method configures the simulation to run in synchronous mode. This synchronization ensures that the server waits for the client to complete its tasks in each iteration. The delta seconds are the discrete frequency of the simulation.

```
def create_synch_settings(self):
    settings = self.world.get_settings()
    settings.synchronous_mode = True
    settings.fixed_delta_seconds = constant.SERVER_DELTA_SECOND
    self.world.apply_settings(settings)
```

4.2.3 Waypoint Visualization

The `draw_waypoints` method is used to visually represent the waypoints in the simulation environment. By drawing waypoints, we can visually track the vehicle's adherence to the planned route. The waypoints are drawn with different markers to indicate their position and are essential for debugging and ensuring correct vehicle navigation.

```
def draw_waypoints(self):
    spawn_points = self.world.get_map().get_spawn_points()

    start_p = carla.Location(spawn_points[0].location)
    finish_p = carla.Location(spawn_points[100].location)

    self.world.debug.draw_point(
        start_p, color=carla.Color(r=0, g=255, b=0), size=1.6,
   life_time=120.0
    )
    self.world.debug.draw_point(
        finish_p, color=carla.Color(r=0, g=255, b=0), size=1.6,
   life_time=120.0
```

```python
    )

    amap = self.world.get_map()
    sample_resolution = 2
    wps = []
    grp = GlobalRoutePlanner(amap, sample_resolution)

    waypoints = grp.trace_route(start_p, finish_p)

    for iter_i, way in enumerate(waypoints):
        if iter_i % 10 == 0:
            self.world.debug.draw_string(
                way[0].transform.location,
                "O",
                draw_shadow=False,
                color=carla.Color(r=255, g=0, b=0),
                life_time=120.0,
                persistent_lines=True,
            )
        else:
            self.world.debug.draw_string(
                way[0].transform.location,
                "^",
                draw_shadow=False,
                color=carla.Color(r=0, g=0, b=255),
                life_time=1000,
                persistent_lines=True,
            )

    for waypoint in waypoints:
        wps.append(waypoint[0])

    logging.debug("The waypoints are drawn")
    return wps
```

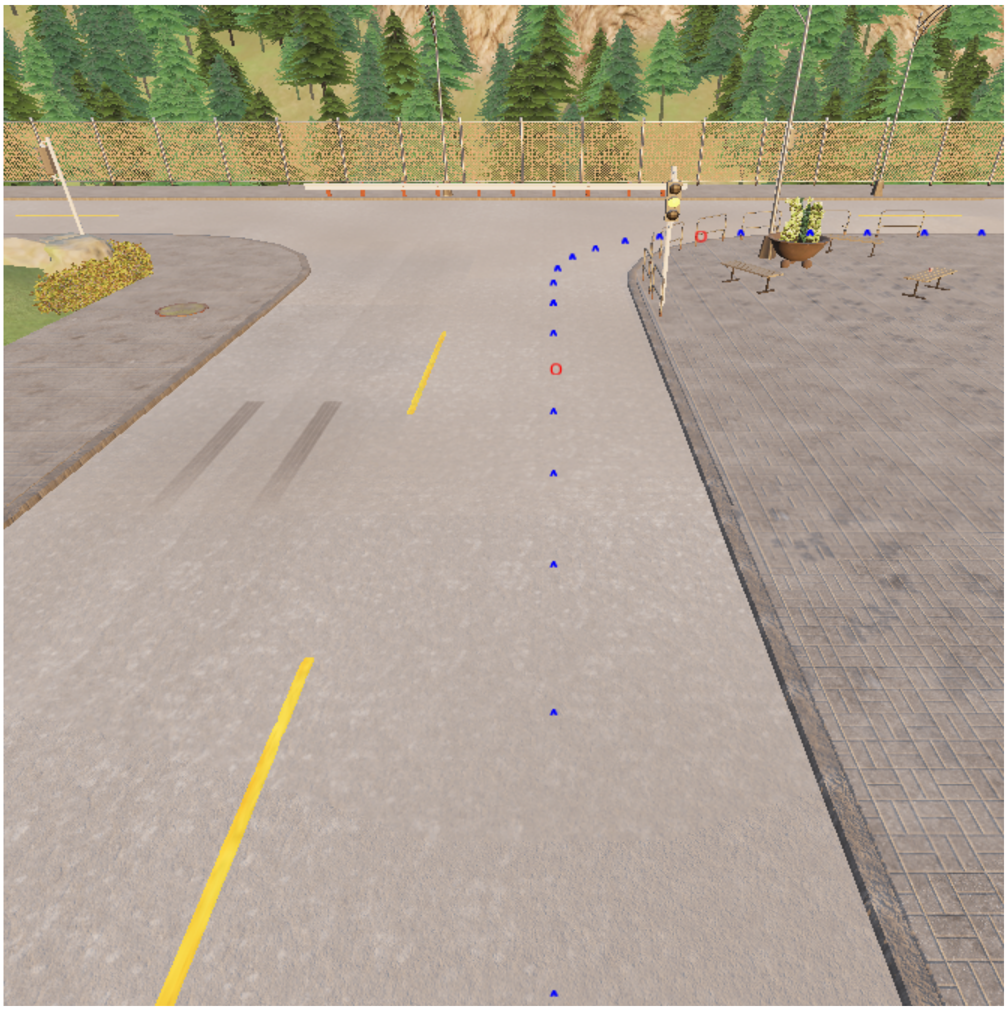


Figure 4.1: Waypoints drawn on the path.

4.2.4 Vehicle Spawning

Within the intricate framework of CARLA (Car Learning to Act), the process of vehicle spawning emerges as a foundational element, facilitating the creation and integration of vehicular entities within the virtual environment. At the core of this functionality lies the vehicle_spawn function, a meticulously crafted piece of code designed to orchestrate the seamless instantiation of vehicles. In the context of this discussion, our focus turns to the instantiation of vehicles modeled after the Microlino – a compact and innovative electric vehicle designed for urban mobility. By leveraging the capabilities of vehicle_spawn, we delve into the process of bringing the Microlino to life within the CARLA simulation environment.

Figure 4.2: Microlino car that will be used.

```python
def vehicle_spawn(self, offset):
    spawn_points = self.world.get_map().get_spawn_points()
    new_pose = carla.Transform(
        carla.Location(
            x=spawn_points[0].location.x + offset[0],
            y=spawn_points[0].location.y + offset[1],
            z=spawn_points[0].location.z,
        ),
        carla.Rotation(
            pitch=spawn_points[0].rotation.pitch,
            yaw=spawn_points[0].rotation.yaw,
            roll=spawn_points[0].rotation.roll,
        ),
    )

    blueprint_library = self.world.get_blueprint_library()
    vehicle_bp = blueprint_library.filter("vehicle.micro.microlino")
    [0]

    logging.debug("The vehicle is created")
    return self.world.spawn_actor(vehicle_bp, new_pose)
```

4.2.5 Sensors Spawning

In autonomous vehicles, the roles of camera and LiDAR sensors are pivotal, as they form the backbone of perception systems, enabling the vehicle to sense and interpret its surrounding environment. These sensors gather crucial data about the vehicle's surroundings, including objects, obstacles, pedestrians, and road conditions. The algorithms responsible for navigation, obstacle avoidance, and decision-making heavily rely on the information provided by these sensors.

It's important to note that, in our project, the perception and sensor fusion aspects are simulated using the Carla Simulator(by `obstacle_sensor`). While the data generated by the simulator may not fully replicate real-world conditions, it provides a valuable testing environment for developing and fine-tuning our algorithms. By leveraging the mock functions provided by the Carla Simulator, we can simulate various scenarios and assess the performance of our perception algorithms under different conditions.

Figure 4.3: Real life camera and lidar.

The `camera_spawn` method attaches a camera to the vehicle, providing a visual feed of the vehicle's surroundings. This is crucial for monitoring the vehicle's behavior and navigation during the simulation.

```python
def camera_spawn(self, vehicle_obj):
    camera_bp = self.world.get_blueprint_library().find("sensor.camera.rgb")
    camera_transform = carla.Transform(
        carla.Location(x=-6, z=5), carla.Rotation(pitch=330)
    )
    cam = self.world.spawn_actor(camera_bp, camera_transform, attach_to=vehicle_obj)

    spec = self.world.get_spectator()
    spec.set_transform(cam.get_transform())
    cam_spec = [cam, spec]
    logging.debug("The camera is created")
    return cam_spec
```

The `obs_sensor_spawn` method spawns an obstacle sensor on the vehicle. This sensor detects obstacles within a specified radius and triggers callbacks for obstacle detection events. Its working principle combines lidar and camera sensors to perform image classification through segmentation. This enables us to detect objects such as pedestrians or cars. By utilizing this information, we can ensure safer navigation for our vehicle in various scenarios. The detection distance of the sensor is selected for worst-case scenarios, which is 30 meters. Lastly, the sensor is attached to the vehicle.

```python
def obs_sensor_spawn(self, vehicle_obj):
    obs_bp = self.world.get_blueprint_library().find("sensor.other.obstacle")
    obs_bp.set_attribute("distance", str(30))
    obs_bp.set_attribute("hit_radius", str(1))
    obs_bp.set_attribute("only_dynamics", str(True))
    obs_bp.set_attribute("sensor_tick", str(0.02))
    obs_location = carla.Location(0, 0, 0.4)
    obs_rotation = carla.Rotation(0, 0, 0)
    obs_transform = carla.Transform(obs_location, obs_rotation)

    logging.debug("The Obstacle sensor is created")
    return self.world.spawn_actor(obs_bp, obs_transform, attach_to=vehicle_obj)
```

Detected obstacle callback

In Carla UE4 (server), the sensor operates at an assigned frequency of `constant.SERVER_DELTA_S`
At each discrete step, the sensor produces results. However, if the callback function is invoked each time, it might miss detected objects between the loop iteration

seconds. As the loop interval increases, the number of missed values will also increase. To avoid this, we use `Queues`, a synchronization threading library. The implemented callback function can be seen below:

```python
def on_obstacle_callback(self, event):
        """Access the obstacle distance information
        when the object detected

        Args:
            event (object): informations on object
        """
        obstacle_distance = event.distance
        actor_type_str = str(event.other_actor)

        actor_loc_rot = event.other_actor.get_transform()

        ObstacleInfo = namedtuple(
            "ObstacleInfo", ["actor_position", "obstacle_distance"]
        )

        actor_type = actor_type_str.split(", ")[1].split("=")[1].
    split(".")[0]
        logging.debug("Yes OBS detected its name is: %s",
    actor_type_str)
        if actor_type == "vehicle" and not actor_type == "static":
            obs_sensor_out = ObstacleInfo(actor_loc_rot,
    obstacle_distance)
            self.obs_queue.put_nowait(obs_sensor_out)
            logging.debug("Yes OBS detected: %s", obs_sensor_out.
    actor_position)
        else:
            logging.debug("No OBS detected..")
```

Braking Distance Calculation

The `braking_distance` method calculates the minimum detection distance for the obstacle sensor, ensuring that the vehicle can safely decelerate to avoid collisions.

```python
def braking_distance(self, speed, decceleration, cr_dist):
    speed_ms = speed * 1000 / 3600
    dist = speed_ms * speed_ms / (2 * decceleration) + cr_dist
    logging.info("The emergency breaking distance is: %s", dist)
    return dist
```

4.2.6 Cleanup and Resource Management

Finally, the `delete_objects_and_settings` method ensures that all created objects are destroyed, and the simulation settings are reverted to their defaults. This cleanup step is essential to maintain a stable simulation environment for subsequent runs.

```
def delete_objects_and_settings(self, vehicle_obj, dummy_obj):
    self.world.apply_settings(carla.WorldSettings(False, False, 0))
    vehicle_obj.destroy()
    if dummy_obj is not None:
        dummy_obj.destroy()
```

4.3 Controller; Driving Car Through Path.

As explained before, in this study, the Adaptive Cruise Controller (ACC) is implemented using both Model Predictive Control (MPC) and Proportional-Integral-Derivative (PID) controllers. The ACC functions as a motion planner, controlling the car through its path using these two different controllers. In the client file, the function is called as shown below:

```
try:
        vehicle = self.setup.vehicle_spawn(constant.CAR_SPAWN_POS)

        # Get the camera actor and attach it to vehicle
        camera, spectator = self.setup.camera_spawn(vehicle)

        obstacle_sensor = self.setup.obs_sensor_spawn(vehicle)

        # Spawn and arrange the Obstacle sensor and listen
        obstacle_sensor.listen(self.setup.on_obstacle_callback)

        # Fetch and draw the waypoints
        sensor_objects = [spectator, camera, self.setup.world]

        # Control the car through the waypoints
        self.control.drive_through_plan(
            self.setup.waypoint_list, vehicle, sensor_objects
        )
```

As can be seen above, the camera, spectator (simulator view), obstacle sensor, and vehicle are passed as an array to the `drive_through_plan` function. These sensors are created by calling their setup functions, where their properties are set as previously explained.

The `drive_through_plan` function, located on the `self.control` side, determines which controller to use based on the command given in the client file. When the client file is started with the bash command "`run_client <controller_name>`", the specified controller name dictates whether the PID or MPC controller will be used. This mechanism allows the system to dynamically choose and initialize the appropriate controller based on the provided controller name.

4.3.1 PID Controller Implementation

The Proportional-Integral-Derivative (PID) controller is a fundamental control mechanism widely used in various engineering applications, including vehicle control systems. In this section, we delve into the specifics of the PID controller implementation within the context of the Adaptive Cruise Controller (ACC) system in CARLA.

As explained in the Background section, the PID controller is designed to maintain a desired setpoint by adjusting the control inputs based on the difference (error) between the desired and actual system states. The PID controller combines three control strategies:

- **Proportional (P) Control:** Directly proportional to the current error.
- **Integral (I) Control:** Based on the accumulation of past errors.
- **Derivative (D) Control:** Based on the prediction of future errors by considering the rate of change of the error.

The PID controller implementation in our system is encapsulated within the `ControllerLuxad` class, specifically in the `create_pid` method. The PID controller is tailored for the ego vehicle, ensuring precise control over its movements.

```python
def create_pid(self, vehicle_ego):
    """Function that defines PID controller

    Args:
        vehicle_ego (object): The car parameters

    Returns:
        function: PID that created specifically for the passed car
    """

    args_lateral = {"K_P": 1.5, "K_D": 0.35, "K_I": 0.4, "dt": 1.0 /
    5.0}

    args_long = {"K_P": 0.9, "K_D": 0.18, "K_I": 0.3, "dt": 1.0 / 5.0}

    return controller.VehiclePIDController(vehicle_ego, args_lateral, args_long)
```

In this function, two sets of PID parameters are defined:

- **Lateral Control Parameters:** These parameters (`K_P`, `K_D`, `K_I`, `dt`) are tuned to control the vehicle's lateral position, ensuring it follows the planned route accurately.
- **Longitudinal Control Parameters:** These parameters are tuned to control the vehicle's speed, ensuring it maintains the desired velocity.

Integration with the Control Loop

The `drive_through_plan` method integrates the PID controller into the vehicle's control loop. The method orchestrates the vehicle's movement through a series of waypoints while continuously adjusting its speed and direction based on sensor inputs. The control of the car is done by the car_control object.

```
def drive_through_plan(self, planned_route, ego_vehicle, sense_obj):
    """Controls the vehicle on the waypoints

    Args:
        planned_route (matrix): waypoints that needed to be passed
        ego_vehicle (object): a car that needed to be controlled on path
        sense_obj (matrix): Sensors that will be used
    """

    spec, cam, world_obj = sense_obj

    pid_value = self.create_pid(ego_vehicle)
    logging.debug("The PID controller created")

    self.target_waypoint = planned_route[0]
    time_first = float(time.time() / 1000.0)
    while self.waypoint_index != (len(planned_route) - 1):
        world_obj.tick()
        # get the location of the car in each iteration
        ego_vehicle_pose = ego_vehicle.get_location()
        ego_vehicle_yaw = ego_vehicle.get_transform().rotation.yaw
        ego_vehicle_velocity = 3.6 * math.sqrt(
            ego_vehicle.get_velocity().x ** 2 + ego_vehicle.get_velocity().y ** 2
        )
        ego_vehicle_velocity_error = constant.SPEED - ego_vehicle_velocity

        # make spectator go with the camera of the car
        spec.set_transform(cam.get_transform())
        fra_angle = 90
        cross_track_error, heading_error = self.check_vehicle_to_waypoint_distance(
            ego_vehicle_pose, planned_route, ego_vehicle_yaw, fra_angle
        )
        current_time = float(time.time() / 1000.0) - time_first
        self.write_to_document(current_time, cross_track_error, heading_error, ego_vehicle_velocity_error)
        if not self.following_obstacle_queue.empty():
            # dequeue obstacle distance information from the queue
            obs = self.following_obstacle_queue.get(timeout=0.1)
            front_obstacle_distance = obs.obstacle_distance
            logging.warning("The distance to OBS: %s", front_obstacle_distance)

            car_control = self.upper_level_control_acc(
                front_obstacle_distance, pid_value
            )

        else:
            self.controller_speed = constant.SPEED
            car_control = pid_value.run_step(
                self.controller_speed, self.target_waypoint
            )

        self.speed_history.append((current_time, self.controller_speed))
        logging.debug("The applied control is: %s", car_control)
        ego_vehicle.apply_control(car_control)

    logging.debug("The last waypoint is reached!")
    car_control = pid_value.run_step(0, planned_route[len(planned_route) - 1])
    ego_vehicle.apply_control(car_control)
```

In this method:

1. The PID controller is created using the `create_pid` method.

2. The vehicle's position, velocity, and orientation are continuously monitored.

3. The `check_vehicle_to_waypoint_distance` function calculates the cross-track and heading errors.

4. If an obstacle is detected within a critical distance, the `upper_level_control_acc` method adjusts the vehicle's speed to maintain a safe distance. Otherwise, the vehicle's speed is controlled based on the PID controller's outputs to maintain a constant speed.

5. The vehicle's control inputs are applied using the `apply_control` method. Which is assigned by the car_control which is a CARLA object that has Steering, throttle, brake and gear box information.

In summary, the implementation of the PID controller within the Adaptive Cruise Controller system provides precise control over the ego vehicle's movements. By continuously monitoring its position, velocity, and orientation, and adjusting its speed and direction based on sensor inputs and controller outputs, the system ensures safe and efficient navigation through planned routes while dynamically responding to obstacles in real-time.

Logging and Documentation

Throughout the control loop, key variables such as timestamp, cross-track error, heading error, and velocity error are logged for analysis. This is done using the `write_to_document` method, which writes these values to a CSV file for later review.

```python
def write_to_document(self, timestamp, cross_track_error, error_theta, vehicle_velocity_error):
    """Function is used to log the values of cross track error and heading error (error_theta) with respect to timestamp.

    Args:
        timestamp (float): time in milliseconds
        cross_track_error (float): lateral distance to waypoint
        error_theta (float): heading error
    """
    fieldnames = ["timestamp", "cross_track_error", "error_theta", "velocity_error"]
    with open(
        "./luxad_docs/PID_Errors.csv", "a", newline="", encoding="utf-8"
    ) as file:
        writer = csv.DictWriter(file, fieldnames=fieldnames)

        # Check if file is empty, if so write header
        file.seek(0, 2)
        if file.tell() == 0:
            writer.writeheader()

        writer.writerow(
            {
                "timestamp": timestamp,
                "cross_track_error": cross_track_error,
                "error_theta": error_theta,
                "velocity_error": vehicle_velocity_error
            }
        )
```

This method ensures that vital performance metrics are recorded for analysis and system refinement.

4.3.2 MPC Controller Implementation

In this section, we delve into the implementation of the Model Predictive Control (MPC) algorithm used to control the autonomous vehicle within the Carla Simulator environment. This subsection will cover the setup, functionality, and key components of the `ControllerMPCLuxad` class, which is responsible for the MPC-based path following and obstacle avoidance.

Data Classes for Vehicle Parameters

We define data classes to store current and historical vehicle parameters and waypoints for the MPC algorithm logging.

Listing 4.1: Data Classes for Vehicle Parameters

```
@dataclasses.dataclass
class CurrentVehicleParameters:
    speed: int
    time: int
    pose_x: int
    pose_y: int
    yaw: int
    speed_limit: int

@dataclasses.dataclass
class HistoryVehicleParameters:
    speed: np.ndarray
    time: np.ndarray
    pose_x: np.ndarray
    pose_y: np.ndarray
    yaw: np.ndarray

```

```
@dataclasses.dataclass
class Waypoint:
    mpc: np.ndarray
    index: int
    distance: np.ndarray
    interpol: np.ndarray  # interpolated values -> (rows = waypoints,
     columns = [x, y, v])
    hash_interpol: np.ndarray  # self.waypoints_mpc to the index of
    the waypoint in interpol
```

These data classes encapsulate the necessary attributes for managing the vehicle's state and the waypoints it must follow.

ControllerMPCLuxad Class

The Model Predictive Control (MPC) is an advanced control strategy that leverages a model of the system to predict future behavior and optimize control actions over a specified prediction horizon. In the realm of autonomous driving and adaptive cruise control (ACC) systems, MPC offers significant advantages in terms of handling constraints, optimizing performance, and ensuring safety.

In this section, we delve into the specifics of the MPC implementation within the ACC system in CARLA. The MPC controller aims to follow a predefined path while dynamically adjusting the vehicle's speed and steering to maintain optimal performance and safety. Unlike traditional controllers, MPC takes into account future predictions and constraints, making it well-suited for complex, real-time applications in autonomous driving.

As explained in the Background section, the MPC algorithm involves solving an optimization problem at each control step. This problem formulation incorporates:

A cost function that captures the deviation from the desired trajectory and control effort. Constraints that ensure the vehicle operates within safe and feasible limits, such as maximum speed, acceleration, and steering angles. A predictive model that simulates the vehicle's future states based on current conditions and control inputs. The MPC implementation in our system is encapsulated within the `ControllerMPCLuxad` class, specifically in the `drive_through_plan` method. This implementation is tailored for the ego vehicle, ensuring precise control over its movements, path following, and obstacle avoidance in a dynamic environment.

Initialization

```
class ControllerMPCLuxad:
    def __init__(self):
        self.controller_speed = constant.SPEED
        self.following_obstacle_queue = constant.obstacle_queue
        self.history_vehicle = HistoryVehicleParameters(
            speed=[0], time=[0], pose_x=[0], pose_y=[0], yaw=[0]
        )
        self.closest_index = 0
        self.closest_distance = 0
        self.current_vehicle = CurrentVehicleParameters(
            speed=0, time=0, pose_x=0, pose_y=0, yaw=0, speed_limit
    =-1
        )
        self.mpc_waypoints = Waypoint(
            mpc=[], index=0, distance=[], interpol=[], hash_interpol
    =[]
        )
```

The constructor initializes various parameters, including the controller speed, obstacle queue, historical vehicle parameters, and waypoints.

Drive Through Plan

As in the PID implementation `drive_through_plan` method controls the vehicle along the planned route, using the MPC algorithm to update control commands dynamically. The main difference is in Model Predictive controller we are trying to predict the future paths and iteratively correct the behavior of the vehicle and keep it in the planned path. Due to that, the the x and y location and heading of the vehicle data needed to be collected in the each iterationn. The loop is executed until the last point of the path is achieved.

```
def drive_through_plan(self, planned_route, ego_vehicle, sense_obj):
    spec, cam, world_obj = sense_obj
    controller = self.controller_defining(planned_route)
    self.waypoint_interpolation(planned_route)

    self.history_vehicle.pose_x[0] = ego_vehicle.get_location().x
    self.history_vehicle.pose_y[0] = ego_vehicle.get_location().y
    self.history_vehicle.yaw[0] = ego_vehicle.get_transform().
    rotation.yaw * 0.0174533

    logging.debug("The MPC controller created")

    while self.mpc_waypoints.index != (len(planned_route) - 1):
        world_obj.tick()
        spec.set_transform(cam.get_transform())
        ego_vehicle_pose = ego_vehicle.get_location()
        self.current_vehicle.pose_x = ego_vehicle_pose.x
        self.current_vehicle.pose_y = ego_vehicle_pose.y
        self.current_vehicle.yaw = ego_vehicle.get_transform().rotation.yaw * constant.ANGLE_TO_RADIAN
        self.current_vehicle.speed = math.sqrt(ego_vehicle.get_velocity().x ** 2 + ego_vehicle.get_velocity().y ** 2)
        self.current_vehicle.time = float(time.time() / 1000.0)

        self.history_vehicle.pose_x.append(self.current_vehicle.pose_x)
        self.history_vehicle.pose_y.append(self.current_vehicle.pose_y)
        self.history_vehicle.yaw.append(self.current_vehicle.yaw)
        self.history_vehicle.speed.append(self.current_vehicle.speed)
        self.history_vehicle.time.append(self.current_vehicle.time)

        if not self.following_obstacle_queue.empty():
            obs = self.following_obstacle_queue.get(timeout=0.1)
            front_obstacle_distance = obs.obstacle_distance
            logging.warning("The distance to OBS: %s", front_obstacle_distance)

            if front_obstacle_distance < constant.CRITICAL_DISTANCE:
                control = constant.control_emergency_br
            else:
                self.current_vehicle.speed_limit = self.distance_braking_speed(
                    front_obstacle_distance,
                    constant.CRITICAL_DISTANCE,
                    constant.DECELERATION,
                )
                self.update_controller(controller)
                control = self.controller_assignment(controller)
        else:
            self.current_vehicle.speed_limit = -1
            self.update_controller(controller)
            control = self.controller_assignment(controller)

        logging.debug("The applied control is: %s", control)
        ego_vehicle.apply_control(control)
        self.mpc_waypoints.index = +1

    logging.debug("The last waypoint is reached!")
    ego_vehicle.apply_control(control)
```

Again was the same as the previously implemented control loop the obstacle list is always checked from the queue. If the obstacle detected in a safe distance the vehicle adopts the front vehicle speed by revoking the "`distance_breaking_speed`"

Waypoint Interpolation

The `waypoint_interpolation` method in the MPC controller plays a crucial role in creating smooth navigation paths for the vehicle by generating interpolated waypoints between the given planned waypoints. This finer resolution of waypoints helps the controller to track the path more precisely, ensuring smoother transitions and more accurate following of the desired trajectory.

```python
def waypoint_interpolation(self, planned_waypoints):
    self.mpc_waypoints.mpc = [
        [
            waypoint.transform.location.x,
            waypoint.transform.location.y,
            self.controller_speed / constant.MS_TO_KMH,
        ]
        for waypoint in planned_waypoints
    ]

    self.mpc_waypoints.mpc = np.array(self.mpc_waypoints.mpc)

    for i in range(1, self.mpc_waypoints.mpc.shape[0]):
        self.mpc_waypoints.distance.append(
            np.sqrt(
                (self.mpc_waypoints.mpc[i, 0] - self.mpc_waypoints.
mpc[i - 1, 0]) ** 2
                + (self.mpc_waypoints.mpc[i, 1] - self.mpc_waypoints.
mpc[i - 1, 1]) ** 2
            )
        )
    self.mpc_waypoints.distance.append(0)

    interp_counter = 0

    for i in range(self.mpc_waypoints.mpc.shape[0] - 1):
        self.mpc_waypoints.interpol.append(list(self.mpc_waypoints.
mpc[i]))
        self.mpc_waypoints.hash_interpol.append(interp_counter)
        interp_counter += 1

        np.seterr(invalid="ignore")
        num_pts_to_interp = int(
            np.floor(
                self.mpc_waypoints.distance[i]
                / float(constant.INTERPOL_INCREMENT_ITERATION)
            )
            - 1
        )
        wp_vector = self.mpc_waypoints.mpc[i + 1] - self.mpc_waypoints.mpc[i]
        wp_uvector = wp_vector / np.linalg.norm(wp_vector)
        for j in range(num_pts_to_interp):
            next_wp_vector = (
                constant.INTERPOL_INCREMENT_ITERATION * float(j + 1) * wp_uvector
            )
            self.mpc_waypoints.interpol.append(
                list(self.mpc_waypoints.mpc[i] + next_wp_vector)
            )
            interp_counter += 1

    self.mpc_waypoints.interpol.append(list(self.mpc_waypoints.mpc[-1]))
    self.mpc_waypoints.hash_interpol.append(interp_counter)
    interp_counter += 1
```

This method ensures that the waypoints are interpolated at a finer resolution for smoother path tracking.

The `waypoint_interpolation` method starts by taking a list of `planned_wayponts` as input. These planned waypoints are processed to extract their x and y coordinates and normalize the speed component using a constant conversion factor from km/h to m/s (`constant.MS_TO_KMH`). These values are then stored in the `mpc_waypoints.mpc` attribute as a NumPy array for efficient computation.

Next, the method calculates the Euclidean distance between each consecutive waypoint. This is achieved by iterating over the waypoints starting from the second one and computing the distance using the Pythagorean theorem. These distances are stored in the `mpc_waypoints.distance` list. An additional 0 is appended to the distance list to account for the distance from the last waypoint to itself, which is inherently zero.

The method then initializes an interpolation counter, `interp_counter`, to keep track of the indices of the interpolated waypoints. For each pair of consecutive waypoints, the method first adds the original waypoint to the `interpol` list and maps this index in the `hash_interpol` list.

To determine the number of points to interpolate between each pair of waypoints, the method calculates the number of interpolation points needed based on the desired resolution defined by `constant.INTERPOL_INCREMENT_ITERATION`. It computes the vector between the current and next waypoint and normalizes it to create unit vectors for interpolation. Using these unit vectors, the method generates intermediate waypoints and appends them to the `interpol` list, incrementing the `interp_counter` for each new interpolated point.

Finally, the last waypoint is appended to the `interpol` list to ensure the end of the path is included, and the corresponding index is recorded in the `hash_interpol` list.

The usage of `np.seterr(invalid="ignore")` helps suppress potential warnings during calculations, ensuring the method runs smoothly without interruption.

Overall, this interpolation process ensures a finer resolution of waypoints, allowing for smoother path tracking and more precise control of the vehicle.

Controller Assignment and Update

The `controller_assignment` and `update_controller` methods are essential components of the MPC controller, responsible for generating control commands and updating the controller with new waypoints and vehicle parameters, respectively.

```python
def controller_assignment(self, controller):
    control = carla.VehicleControl()
    control.throttle, control.steer, control.brake = controller.get_commands()
    control.hand_brake = False
    control.manual_gear_shift = False
    control.reverse = False

    return control

def update_controller(self, controller_mpc):
    new_waypoints = self.new_propper_waypoint_iteration(
        self.current_vehicle.pose_x, self.current_vehicle.pose_y
    )

    controller_mpc.update_waypoints(new_waypoints)
    current_vehicle_parameters = [
        self.current_vehicle.pose_x,
        self.current_vehicle.pose_y,
        self.current_vehicle.yaw,
        self.current_vehicle.speed,
    ]
    print(self.current_vehicle.speed * constant.MS_TO_KMH)
    controller_mpc.update_values(
        current_vehicle_parameters,
        self.current_vehicle.time,
        True,
    )
    controller_mpc.update_controls(self.current_vehicle.speed_limit)
```

The `controller_assignment` method takes a controller object as input and creates a `VehicleControl` object containing throttle, steer, and brake commands obtained from the controller using the `get_commands` method. Additional attributes such as hand brake, manual gear shift, and reverse are set to their default values.

Finally, the method returns the control object.

The `update_controller` method updates the MPC controller with new waypoints and vehicle parameters. It first generates new waypoints using the `new_propper_waypoint_` method based on the current vehicle position. Then, it updates the controller with these waypoints and the current vehicle parameters, including position, yaw angle, speed, and time. Additionally, it sets a flag to indicate if the vehicle is in motion. Finally, the method updates the controller with the current speed limit.

These methods dynamically adjust the control commands based on the vehicle's current state and the updated waypoints, ensuring accurate and responsive control of the vehicle.

New Waypoint Iteration

The `new_propper_waypoint_iteration` method is responsible for generating a subset of waypoints that the vehicle will follow, based on the current vehicle position and the lookahead distance. This method ensures that the vehicle receives a manageable number of waypoints from the MPC controller for smoother navigation.

```python
def new_propper_waypoint_iteration(self, current_x_pose, current_y_pose):
     self.closest_distance = np.linalg.norm(
         np.array(
             [
                 self.mpc_waypoints.mpc[self.closest_index, 0] - current_x_pose,
                 self.mpc_waypoints.mpc[self.closest_index, 1] - current_y_pose,
             ]
         )
     )
     new_distance = self.closest_distance
     new_index = self.closest_index
     while new_distance <= self.closest_distance:
         self.closest_distance = new_distance
         self.closest_index = new_index
         new_index += 1
         if new_index >= self.mpc_waypoints.mpc.shape[0]:  # End of path
             break
         new_distance = np.linalg.norm(
             np.array(
                 [
                     self.mpc_waypoints.mpc[new_index, 0] - current_x_pose,
                     self.mpc_waypoints.mpc[new_index, 1] - current_y_pose,
                 ]
             )
         )
     new_distance = self.closest_distance
     new_index = self.closest_index
     while new_distance <= self.closest_distance:
         self.closest_distance = new_distance
         self.closest_index = new_index
         new_index -= 1
         if new_index < 0:  # Beginning of path
             break
         new_distance = np.linalg.norm(
             np.array(
                 [
                     self.mpc_waypoints.mpc[new_index, 0] - current_x_pose,
                     self.mpc_waypoints.mpc[new_index, 1] - current_y_pose,
                 ]
             )
         )

     waypoint_subset_first_index = self.closest_index - 1
     waypoint_subset_first_index = max(waypoint_subset_first_index, 0)

     waypoint_subset_last_index = self.closest_index
     total_distance_ahead = 0
     while total_distance_ahead < constant.TOTAL_DISTANCE_AHEAD:
         total_distance_ahead += self.mpc_waypoints.distance[
             waypoint_subset_last_index
         ]
         waypoint_subset_last_index += 1
         if waypoint_subset_last_index >= self.mpc_waypoints.mpc.shape[0]:
             waypoint_subset_last_index = self.mpc_waypoints.mpc.shape[0] - 1
             break

     new_waypoints = self.mpc_waypoints.interpol[
         self.mpc_waypoints.hash_interpol[
             waypoint_subset_first_index
         ] : self.mpc_waypoints.hash_interpol[waypoint_subset_last_index]
         + 1
     ]

     return new_waypoints
```

The `new_propper_waypoint_iteration` method is crucial for determining the subset of waypoints that the vehicle will follow at any given moment during navigation. It operates by dynamically selecting waypoints based on the vehicle's current position and the lookahead distance specified by the system.

To begin, the method calculates the closest waypoint to the vehicle's current position. This is done by computing the Euclidean distance between the vehicle's coordinates and the coordinates of each waypoint along the planned route. The waypoint with the minimum distance to the vehicle is identified as the closest waypoint.

Once the closest waypoint is determined, the method selects a subset of waypoints to be followed by the vehicle. This subset includes waypoints both ahead of and behind the vehicle, providing a buffer for smooth navigation. The number of waypoints included in the subset is determined based on a specified lookahead distance, which represents the distance ahead of the vehicle that the controller should consider for planning its trajectory.

The method iterates over the planned route, incrementing and decrementing the waypoint index from the closest waypoint to identify the subset of waypoints. It ensures that the subset includes at least one waypoint behind the vehicle and extends sufficiently ahead of the vehicle to cover the specified lookahead distance.

Once the subset of waypoints is determined, it is returned for use by the MPC controller. These waypoints serve as key reference points for trajectory planning and control, enabling the vehicle to navigate safely and effectively along its planned route.

Overall, the `new_propper_waypoint_iteration` method plays a critical role in ensuring that the vehicle receives a manageable number of waypoints for navigation, thereby facilitating smooth and efficient path tracking.

Controller Definition

The `controller_defining` method is responsible for creating an instance of the Model Predictive Control (MPC) controller used for trajectory planning and control. This method plays a crucial role in initializing the controller with the appropriate set of waypoints extracted from the planned route.

The method takes a list of route points (waypoints) as input, typically represented as a matrix containing the (x, y, z) coordinates of each waypoint along the planned route. These waypoints are essential for guiding the vehicle's trajectory and ensuring it follows the desired path accurately.

To create the MPC controller, the method iterates over the list of route points and converts each waypoint into a format suitable for MPC trajectory planning. This involves extracting the x and y coordinates of each waypoint and converting the speed limit from meters per second (m/s) to kilometers per hour (km/h), as required by the MPC controller.

Once the list of converted waypoints is prepared, it is passed as input to the constructor of the MPC controller class (`mpc_control.Controller2D`). This initializes the MPC controller with the necessary waypoints and other parameters required for trajectory planning and control.

The initialized MPC controller object is then returned by the method, ready to be used for guiding the vehicle along its planned route. By encapsulating the creation of the MPC controller within this method, the system maintains modularity and flexibility, allowing for easy integration of different controller implementations or modifications to the trajectory planning algorithm.

Overall, the `controller_defining` method serves as a vital component in the initialization of the MPC controller, ensuring that it is properly configured with the relevant waypoints for effective trajectory planning and control.

```python
def controller_defining(self, routes):
    waypoints_mpc_list = list(
        (
            waypoint.transform.location.x,
            waypoint.transform.location.y,
            self.controller_speed / 3.6,
        )
        for waypoint in routes
    )

    return mpc_control.Controller2D(waypoints_mpc_list)
```

Distance Braking Speed

The `distance_braking_speed` method that ensuring the safety of the vehicle by calculating the appropriate speed based on the distance to obstacles in the environment.

When called, this method takes three arguments: `obstacle_dist`, which represents the distance to the nearest obstacle, `critical_dist`, the threshold distance beyond which emergency braking is required, and `decelleration`, the rate at which the vehicle should decelerate.

The method calculates the safe braking speed (`speed_br`) by first determining the difference between the obstacle distance (`obstacle_dist`) and the critical distance (`critical_dist`). This difference is then multiplied by two times the deceleration rate (`decelleration`).

Subsequently, the square root of the resulting value is taken to obtain the final safe braking speed. Finally, the speed is converted from meters per second (m/s) to kilometers per hour (km/h) using the conversion factor provided by the constant

`MS_TO_KMH`.

The calculated safe braking speed is returned by the method, providing the vehicle with the necessary information to adjust its speed dynamically based on the proximity of obstacles in its path.

```python
def distance_braking_speed(self, obstacle_dist, critical_dist,
    deceleration):
    speed_br = (obstacle_dist - critical_dist) * 2 * deceleration
    speed_br = math.sqrt(speed_br)
    print(speed_br * constant.MS_TO_KMH)
    return speed_br  # conversion from m/s
```

Chapter 5

Conclution and Final Assessment

5.1 Benchmark on Control Operations and Algorithms

As previously described, the autonomous driving control algorithms implemented in this study are PID (Proportional-Integral-Derivative) and MPC (Model Predictive Control) for Adaptive Cruise Control (ACC) in motion planning. These algorithms are critical for maintaining the vehicle's desired speed and trajectory while adapting to changing road conditions and obstacles.

To comprehensively evaluate the performance of these control algorithms, scenarios were created and tested in both rural and urban environments. The rural scenarios typically feature fewer cars and simpler road layouts, which provide a controlled environment to observe the basic performance and tuning of the control algorithms. On the other hand, the urban scenarios include a higher density of pedestrians, bicycles, and complex traffic patterns, offering a more challenging environment to test the robustness and adaptability of the algorithms.

The comparison between PID and MPC algorithms is conducted using two primary metrics: heading error and cross-track error. Heading error measures the difference between the vehicle's actual heading and the desired heading, reflecting the accuracy of the vehicle's steering control. Cross-track error, on the other hand, measures the lateral distance between the vehicle's current position and the desired trajectory, indicating how well the vehicle maintains its path.

Figures depicting the created rural scenario, which features fewer cars, and the urban scenario, which includes more pedestrians and bicycles, are shown below. These scenarios were carefully designed to test the control algorithms under various conditions and complexities:

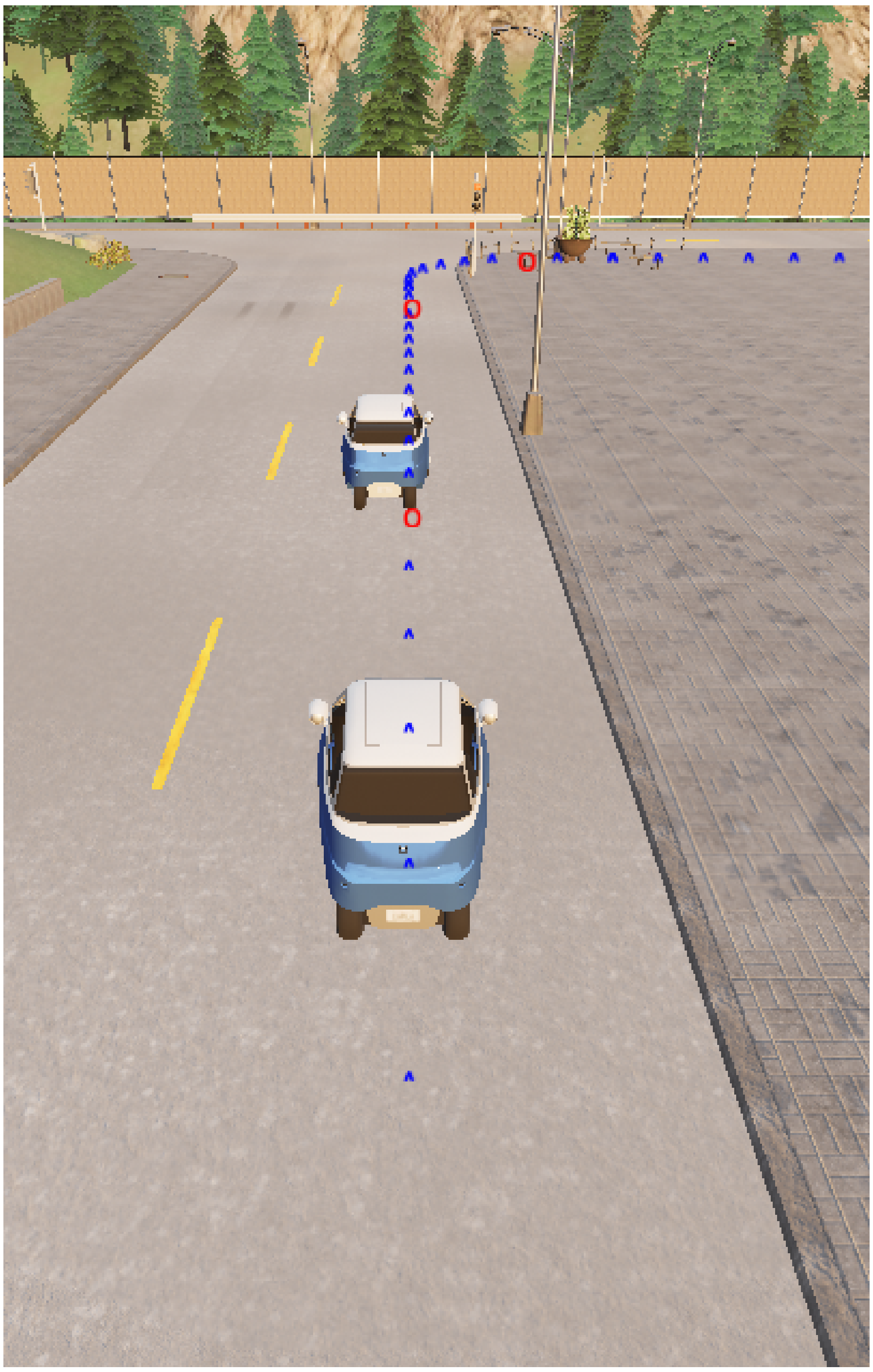


Figure 5.1: Operation view in a rural scenario.

The analysis of the PID implementation in the rural scenario can be seen below:

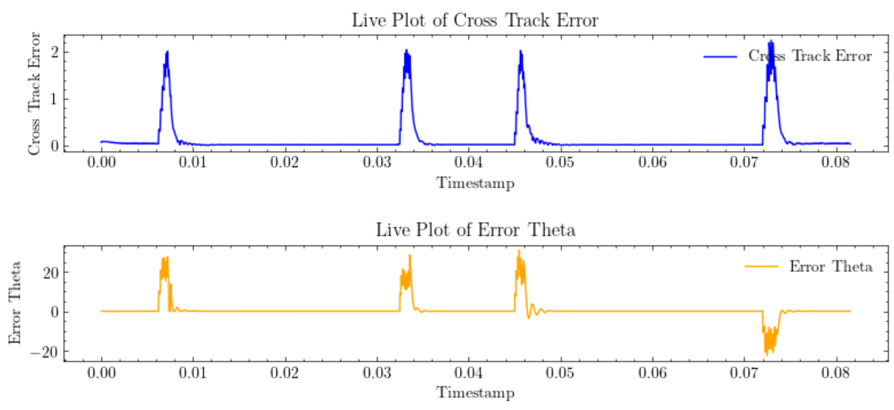


Figure 5.2: PID controller errors in a rural scenario.

Similarly, the MPC implementation in the rural scenario is shown below:

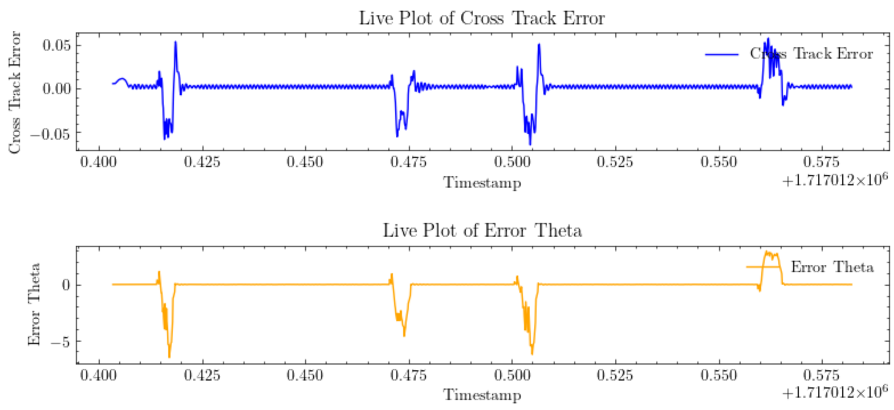


Figure 5.3: MPC controller errors in a rural scenario.

As seen in these graphs, the peak cross-track error for the MPC is within 5 cm, whereas for the PID controller, it exceeds 2 meters. Another critical aspect is the heading error, which measures the correctness of the vehicle's steering. For the PID controller, the heading error is around 25 degrees, while for the MPC, it is only 5 degrees.

In rural scenarios, it is evident that the MPC outperforms the PID controller significantly. Next, we will examine the scenarios where the controllers are tested in urban environments with more traffic and pedestrians.

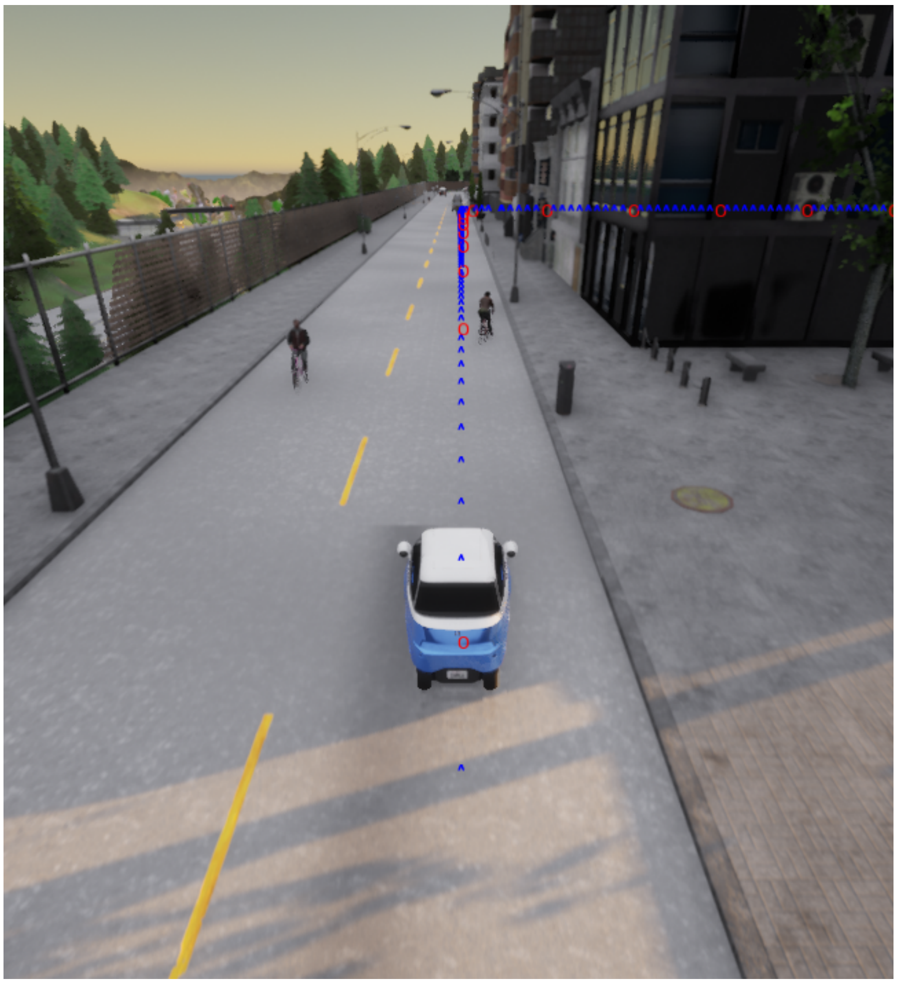


Figure 5.4: Complex urban scenario.

As depicted in the image, the urban scenario is more complex, featuring two bicyclists and a car ahead. In such a scenario, the vehicle must detect the front obstacle and maintain the speed of the front vehicle according to Adaptive Cruise Control.

The performance of the two different control algorithms with ACC in the urban scenario is shown below:

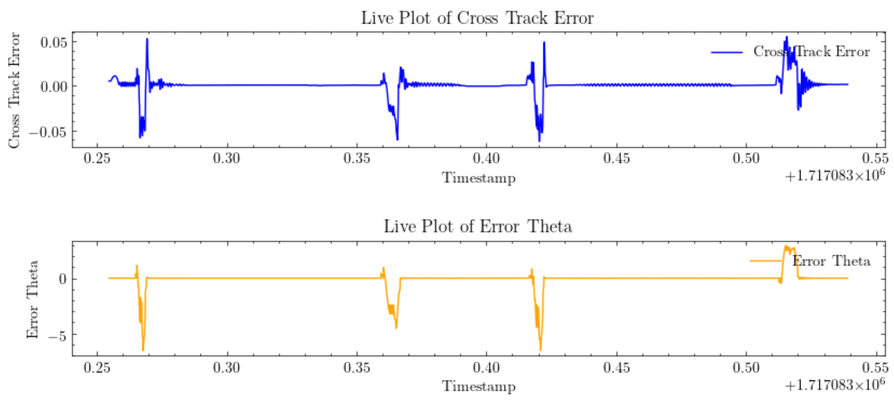


Figure 5.5: MPC algorithm operation in an urban scenario.

The analysis of the PID implementation in the urban scenario is shown below:

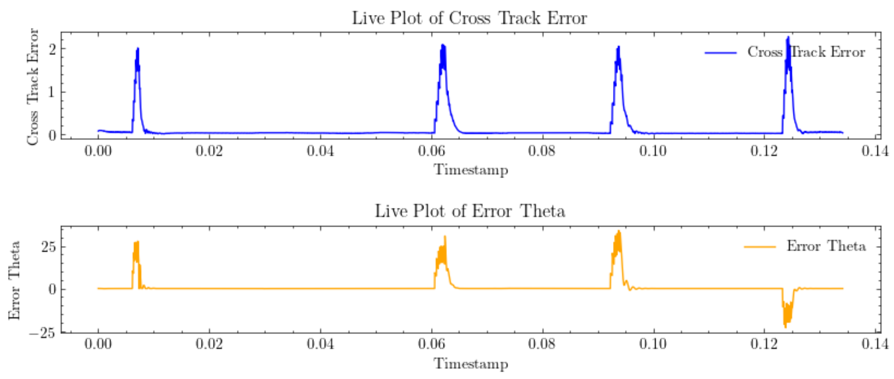


Figure 5.6: PID algorithm operation in an urban scenario.

From the error analysis, it is clear that both algorithms respond in a similar manner, but the time taken to correct errors varies. The PID controller shows more peaks in response, particularly on curves where centripetal force is applied, indicating less smooth performance.

In conclusion, the more complex control algorithm methodology, Model Predictive Control, performs better as it evaluates hazardous scenarios and provides responses that also consider driving safety in autonomous models. Therefore, MPC is more suitable for adaptive cruise control in complex environments, ensuring a smoother and safer driving experience.

5.2 Hazard and Risk Assessment: PID vs MPC

As one of the most important parts of the study, the ISO analysis will be used for the software algorithm application analysis. In the previous sections, scenarios were created in different environments where various operations are implemented.

The scenarios include the most and least dangerous situations: complex traffic in the city and a more comfortable rural area. These will be evaluated in terms of emergency braking, adaptive cruise control driving, and lateral movements.

The scenarios can be classified as follows:

1. Scenario 1 (S1): In `city` the *adaptive cruise control*.
2. Scenario 2 (S2): In a `city` the *emergency braking*.
3. Scenario 3 (S3): In a `city` the *lateral* (curve) movement.
4. Scenario 4 (S4): In a `rural` area the *adaptive cruise control* movement.
5. Scenario 5 (S5): In a `rural` area the *emergency braking* movement.
6. Scenario 6 (S6): In a `rural` area the *lateral* (curve) movement.

The two algorithms considered are:

- Algorithm 1 (A1): PID control algorithm
- Algorithm 2 (A2): MPC control algorithm

The parameters for hazard analysis are Severity (S), Exposure (E), and Controllability (C).

	S1	S2	S3	S4	S5	S6	ASIL
A1	S:2 E:1 C:1	S:2 E:2 C:2	S:3 E:4 C:3	S:1 E:1 C:1	S:1 E:2 C:2	S:1 E:4 C:3	D
A2	S:1 E:1 C:1	S:1 E:2 C:1	S:3 E:2 C:1	S:1 E:1 C:1	S:1 E:1 C:1	S:1 E:2 C:1	B

Table 5.1: HARA analysis for scenarios (1-6) with respect to PID Algorithm (A1) and MPC Algorithm (A2).

The analysis is based on cross-track and heading errors (Figures 5.2, 5.3, 5.6, and 5.5). The main differences occur during lateral movements, which become more severe in city scenarios. Therefore, Scenario 3 is the riskiest scenario for both algorithms. For the PID controller, it has a 25-degree heading error and a 2-meter cross-track error in city scenarios during lateral movements, which results in an ASIL D rating. In contrast, thanks to its predictive behavior, the MPC controller performs with a 5-degree heading error and a peak cross-track error of 20 cm, resulting in an ASIL B rating.

5.3 Final Considerations

In conclusion, both PID and MPC controllers are integral components of the ACC system, yet they exhibit distinct characteristics and origins. The PID controller library, provided by CARLA, offers a well-established control mechanism for maintaining desired setpoints. Its implementation is relatively straightforward, relying on proportional, integral, and derivative control strategies to adjust vehicle behavior based on error signals. While PID controllers are effective in many scenarios and are easy to implement, they may struggle in complex environments with nonlinear dynamics or stringent constraints.

Conversely, the MPC library utilized in this project is sourced from the University of Toronto's Autonomous Driving course. MPC offers significant advantages over PID in terms of handling constraints, optimizing performance, and ensuring safety. Its predictive nature allows it to anticipate future states and optimize control actions over a specified horizon, making it well-suited for complex, real-time applications in autonomous driving. However, MPC controllers tend to be more complex to implement and require detailed modeling of the system dynamics.

Our exploration has revealed that the MPC controller exhibits greater flexibility and adaptability compared to PID, particularly in scenarios with dynamic obstacles, complex road geometries, or stringent safety requirements. By testing various scenarios, such as highway merging, urban navigation, and emergency braking, the strengths and weaknesses of each controller were assessed in different contexts.

Overall, the MPC algorithm demonstrates superior performance compared to the PID algorithm, particularly in more complex urban scenarios. The predictive nature of the MPC allows it to handle dynamic environments and maintain better control of the vehicle's trajectory and speed. This results in safer and more reliable performance, making MPC a preferable choice for adaptive cruise control and other advanced driver assistance systems in autonomous driving applications.

As our maiden results in Table 5.1 show, the HARA analysis resulted in an ASIL B for the MPC controller and an ASIL D for the PID controller, indicating that MPC offers two levels higher safety assurance compared to PID.

In this study, we did not have the time to implement highway scenarios due to time constraints. Instead, we focused on the most and least dangerous scenarios for the HARA analysis. For future work, it is recommended to create highway scenarios and implement AI-based nonlinear MPC for predicting the cost function variables of the MPC. Reinforcement learning algorithms can be employed in conjunction with Rapidly-exploring Random Tree (RRT) methods for enhanced motion planning. This approach will potentially improve the adaptability and performance of control systems in more complex and dynamic environments, further advancing the safety and reliability of autonomous driving technologies.

Appendix

User Handbook for Project Setup and Execution

This section provides a detailed guide on setting up and running the LUXOFT AD Control repository, which includes the control algorithm and the CARLA setup submodule (`lux_ad_carla`). Follow these steps to ensure a proper setup and execution environment for CARLA.

SSH Key Setup

To interact with the repository, ensure that you have an SSH key set up on your local machine. If you do not have an SSH key, follow the instructions here to create one. This step is crucial for secure access to the repository.

Anaconda Installation

The project requires Anaconda with Python 3.7. Anaconda is a widely used distribution that simplifies package management and deployment. You can download Anaconda here. Make sure to choose the version that includes Python 3.7.

Cloning the Repository

Follow these steps to clone the LUXOFT AD Control repository and initialize the necessary submodules:

1. **Clone the Repository with SSH**: Open a terminal and run the following command to clone the repository:

   ```
   git clone git@bitbucket.org:your_username/lux_ad_control.git
   ```

2. **Navigate to the Directory**: After cloning, navigate to the repository directory:

```
cd lux_ad_control
```

3. **Initialize Submodules**: Initialize and update the submodules to ensure all components are correctly set up:

   ```
   git submodule update --init --recursive --progress
   ```

.0.1 Setting Up the Environment

With the repository cloned and submodules initialized, set up the Python environment using Anaconda:

1. **Configure Anaconda with Python 3.7**: Create a new Anaconda environment with Python 3.7 and activate it:

   ```
   conda create -n lux_ad_env python=3.7
   conda activate lux_ad_env
   ```

2. **Run the Setup Script**: Source the setup script to configure the environment with necessary dependencies and settings:

   ```
   source luxad_toolkit.sh
   ```

Running CARLA

Once the environment is set up, follow these steps to run CARLA:

1. **Start CARLA Server**: Launch the CARLA server, which is required for the simulation environment:

   ```
   luxad_run_server
   ```

   After starting the server, a screen will open. Press ENTER to proceed.

2. **Start CARLA Client**: With the server running, start the CARLA client to interact with the simulation and observe the control algorithms in action:

```
luxad_run_client
```

This command initiates the client interface, allowing you to visualize and test the implemented code.

By following these steps, you can successfully set up and run the LUXOFT AD Control system with CARLA, enabling you to evaluate and analyze the performance of the implemented control algorithms in various driving scenarios.

Bibliography

[1] Farzeen Munir, Shoaib Azam, Muhammad Ishfaq Hussain, Ahmed Muqeem Sheri, and Moongu Jeon. «Autonomous Vehicle: The Architecture Aspect of Self Driving Car». In: *Proceedings of the 2018 International Conference on Sensors, Signal and Image Processing.* SSIP '18. Prague, Czech Republic: Association for Computing Machinery, 2018, pp. 1–5. ISBN: 9781450366205. DOI: `10.1145/3290589.3290599`. URL: `https://doi.org/10.1145/3290589.3290599` (cit. on p. 1).

[2] Srinivas P, Rohan Gudla, Vijay Telidevulapalli, Jayasree Kota, and Gayathri Mandha. «Review on self-driving cars using neural network architectures». In: *World Journal of Advanced Research and Reviews* 16 (Nov. 2022), pp. 736–746. DOI: `10.30574/wjarr.2022.16.2.1240` (cit. on p. 1).

[3] On-Road Automated Driving (ORAD) Committee. «Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles». In: (Apr. 2021). DOI: `https://doi.org/10.4271/J3016_202104`. URL: `https://doi.org/10.4271/J3016_202104` (cit. on p. 2).

[4] Thor Axel Achim Heinrich José Struck. «Application Software Components in AUTOSAR». AAI28992423. PhD thesis. 2020. ISBN: 9798209811756 (cit. on p. 2).

[5] Pei Sun et al. «Scalability in Perception for Autonomous Driving: Waymo Open Dataset». In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).* June 2020 (cit. on p. 8).

[6] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. «DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving». In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV).* Dec. 2015 (cit. on p. 8).

[7] You Li and Javier Ibanez-Guzman. «Lidar for autonomous driving: The principles, challenges, and trends for automotive lidar and perception systems». In: *IEEE Signal Processing Magazine* 37.4 (2020), pp. 50–61 (cit. on p. 8).

[8] Li-Hua Wen and Kang-Hyun Jo. «Deep learning-based perception systems for autonomous driving: A comprehensive survey». In: *Neurocomputing* 489 (2022), pp. 255–270 (cit. on p. 9).

[9] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, José Neira, Ian Reid, and John J Leonard. «Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age». In: *IEEE Transactions on robotics* 32.6 (2016), pp. 1309–1332 (cit. on p. 9).

[10] Hugh Durrant-Whyte and Tim Bailey. «Simultaneous localization and mapping: part I». In: *IEEE robotics & automation magazine* 13.2 (2006), pp. 99–110 (cit. on p. 9).

[11] Rong Liu, Jinling Wang, and Bingqi Zhang. «High definition map for automated driving: Overview and analysis». In: *The Journal of Navigation* 73.2 (2020), pp. 324–341 (cit. on p. 9).

[12] Lydia E Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. «Probabilistic roadmaps for path planning in high-dimensional configuration spaces». In: *IEEE transactions on Robotics and Automation* 12.4 (1996), pp. 566–580 (cit. on p. 9).

[13] Steven M LaValle and James J Kuffner. «Rapidly-exploring random trees: Progress and prospects: Steven m. lavalle, iowa state university, a james j. kuffner, jr., university of tokyo, tokyo, japan». In: *Algorithmic and computational robotics* (2001), pp. 303–307 (cit. on p. 9).

[14] Jiankun Wang, Tianyi Zhang, Nachuan Ma, Zhaoting Li, Han Ma, Fei Meng, and Max Q-H Meng. «A survey of learning-based robot motion planning». In: *IET Cyber-Systems and Robotics* 3.4 (2021), pp. 302–314 (cit. on p. 9).

[15] Riccardo Marino, Stefano Scalzi, and Mariana Netto. «Nested PID steering control for lane keeping in autonomous vehicles». In: *Control Engineering Practice* 19.12 (2011), pp. 1459–1467 (cit. on p. 10).

[16] Muhammad Awais Abbas, Ruth Milman, and J Mikael Eklund. «Obstacle avoidance in real time with nonlinear model predictive control of autonomous vehicles». In: *Canadian journal of electrical and computer engineering* 40.1 (2017), pp. 12–22 (cit. on p. 10).

[17] Stéphanie Lefevre, Ashwin Carvalho, and Francesco Borrelli. «A learning-based framework for velocity control in autonomous driving». In: *IEEE Transactions on Automation Science and Engineering* 13.1 (2015), pp. 32–42 (cit. on p. 10).

[18] Siddartha Khastgir, Stewart Birrell, Gunwant Dhadyalla, Håkan Sivencrona, and Paul Jennings. «Towards increased reliability by objectification of Hazard Analysis and Risk Assessment (HARA) of automated automotive systems». In: *Safety Science* 99 (2017). Risk Analysis Validation and Trust in Risk management, pp. 166–177. ISSN: 0925-7535. DOI: https://doi.org/10.1016/j.ssci.2017.03.024. URL: https://www.sciencedirect.com/science/article/pii/S0925753517305763 (cit. on p. 10).

[19] Kuen-Long Leu, Hsiang Huang, Yung-Yuan Chen, Li-Ren Huang, and Kung-Ming Ji. «An intelligent brake-by-wire system design and analysis in accordance with ISO-26262 functional safety standard». In: *2015 International Conference on Connected Vehicles and Expo (ICCVE)*. 2015, pp. 150–156. DOI: 10.1109/ICCVE.2015.20 (cit. on p. 11).

[20] Thomas Drage, Kai Li Lim, Joey En Hai Koh, David Gregory, Craig Brogle, and Thomas Bräunl. «Integrated Modular Safety System Design for Intelligent Autonomous Vehicles». In: *2021 IEEE Intelligent Vehicles Symposium (IV)*. 2021, pp. 258–265. DOI: 10.1109/IV48863.2021.9575662 (cit. on p. 11).

[21] Fredrik Warg, Martin Skoglund, Anders Thorsén, Rolf Johansson, Mattias Brännström, Magnus Gyllenhammar, and Martin Sanfridson. «The Quantitative Risk Norm - A Proposed Tailoring of HARA for ADS». In: *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 2020, pp. 86–93. DOI: 10.1109/DSN-W50199.2020.00026 (cit. on p. 11).