



Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelor-Thesis

Thanh Nguyen

Design and Test of a Controller Approach for Adaptive Cruise Control in CARLA

Thanh Nguyen

Matriculation number: [REDACTED]

Design and Test of a Controller Approach for Adaptive Cruise Control in CARLA

**Bachelor thesis submitted as part of the Bachelor examination,
in the course of study Vehicle Technology
at the Department of Automotive and Aeronautical Engineering
of the Faculty of Engineering and Computer Science
of the University of Applied Sciences Hamburg**

First examiner: Prof. Dr. Dirk Adamski

Second examiner: Prof. Dr. Dirk Engel

Submission date: 01.09.2023

Abstract

Thanh Nguyen

Title of the paper

Design and Test of a Controller Approach for Adaptive Cruise Control in CARLA

Keywords

Adaptive Cruise Control, ACC, CARLA, Python, PID Controller, Lidar sensors, Automated driving, Simulation, Traffic scenarios, Environmental conditions, Testing

Abstract

This paper discusses developing and testing an Adaptive Cruise Control (ACC) controller approach for automated driving using lidar sensors in CARLA, an open-source simulation environment. ACC systems enable drivers to set a desired speed and minimum distance from the vehicle in front, using radar or lidar sensors to detect the speed and distance of other vehicles. Due to the complexity involved in developing such control systems, simulation environments are necessary, and CARLA provides a realistic environment for testing and evaluating automated driving control systems. The article describes the design of an ACC controller with lidar sensors in CARLA and emphasizes the importance of testing the functionality of the controller approach in a wide variety of traffic and environmental scenarios.

Contents

Abstract.....	1
List of Figures	4
List of Tables.....	6
List of Abbreviations.....	7
List of Symbols.....	8
List of Code Snippets.....	9
1 Introduction	10
1.1 Background	10
1.2 Motivation.....	10
1.3 Objectives.....	11
1.4 Scope	11
1.5 Overview	11
2 Related work	12
2.1 Definitions	12
2.1.1 ADAS [1]	12
2.1.2 ACC	12
2.1.3 Radar vs. Lidar	16
2.1.4 Proportional-Integral-Derivative (PID) Controller [10]	19
2.2 The diversity of development strategies for ACC	21
2.3 CARLA Simulation Environment [17].....	22
2.3.1 Introduction	22
2.3.2 Simulation Engine.....	23
2.3.3 CARLA coordinates system.....	24
2.3.4 Lidar model in CARLA [19].....	25
2.3.5 Vehicle model in CARLA	27
2.3.6 CARLA maps [25]	30
3 Methodology.....	31
3.1 ACC Controller Design and its implementation in Python	31
3.1.1 ACC structure	32
3.1.2 ACC principles and algorithms	32
3.1.3 PID Controller design [40]	38
3.2 Testing methodology	44
3.2.1 Test setup.....	44
3.2.2 Output data retrieval method.....	46
3.2.3 Assessment criteria	46

3.2.4 Pre-simulation setup	47
3.2.5 Main test scenarios and prediction	65
3.2.6 Advanced scenarios.....	70
3.3 Implemented control logics	70
4 Execution and results	71
4.1 Executing the defined scenarios and results.....	71
4.1.1 Approaching target vehicle	71
4.2.2 Alternating headway values.....	78
4.2.3 Target vehicle decelerating	89
4.2.4 Target vehicle accelerating	92
4.2.5 Target vehicle speeding off	96
4.2 Executing advanced scenarios and results.....	100
5 Discussion.....	102
6 Conclusion	103
Works Cited	105
The use of generative AI in this paper	109
Final words	110

List of Figures

Figure 1 ACC functionality [2]	13
Figure 2 SAE Classification of autonomous driving levels [3]	15
Figure 3 Working principle of a Radar system [6]	16
Figure 4 Working principle of a lidar system [9]	18
Figure 5 PID controller scheme [11]	20
Figure 6 CARLA client-server architecture [63]	23
Figure 7 CARLA Coordinate System [18]	24
Figure 8 Lidar's point cloud [20]	25
Figure 9 Point cloud imported to Meshlab [21]	26
Figure 10 Tesla model 3 in CARLA [57]	28
Figure 11 Dodge charger police 2020 in CARLA [23]	29
Figure 12 Comparing acceleration curve with/without a set mass. Above: without set mass, below: with set mass	30
Figure 13 Town 6 in CARLA [60]	31
Figure 14 Overall ACC system structure [26]	32
Figure 15 Euclidean distance formula two-dimensional [33]	33
Figure 16 Euclidean distance three-dimensional [33]	34
Figure 17 Two 100% overlapping vehicles	35
Figure 18 Two partial overlapping vehicles	36
Figure 19 Control system block diagram [42]	40
Figure 20 PID controller response types [43]	44
Figure 21 Lidar sensor placement	48
Figure 22 Color intensity indicating different distances	49
Figure 23 3D point cloud in Meshlab	50
Figure 24 Settings to clarify the 3D point cloud	51
Figure 25 Identifying target vehicle	51
Figure 26 Measuring tool in Meshlab	52
Figure 27 Reference distance from the CARLA client in command prompt	52
Figure 28 Measurement #1	53
Figure 29 Measurement #2	53
Figure 30 Measurement #3	54
Figure 31 Measurement #4	54
Figure 32 Example for the Ziegler-Nichols method [53]	57
Figure 33 Tuning simulation result #1 ($Kp = 1.0$)	58
Figure 34 Plateau section	59
Figure 35 First spike in speed curve	60
Figure 36 Sustained oscillation ($Kp = 4.7$)	61
Figure 37 Response utilizing the classic PID gains parameters	62
Figure 38 Classic PID controller at set speed=15ms	63
Figure 39 Classic PID controller at set speed=20ms	64
Figure 40 Classic PID controller at set speed=25ms	65
Figure 41 ACC regulating cycle	66
Figure 42 Headway alternating regulating cycle	67
Figure 43 Target vehicle decelerating process	68
Figure 44 Target vehicle accelerating process	69
Figure 45 Target vehicle speeding off process	69
Figure 46 Result from the approaching scenario #1	73

Figure 47 Result from the approaching scenario #2	76
Figure 48 Closer inspection in the speed curve	77
Figure 49 Jagged curve behavior in deceleration	77
Figure 50 Result from alternating headway from long to medium distance	80
Figure 51 Headway change after 200s	81
Figure 52 Distance curves after changing headway.....	81
Figure 53 Acceleration curve after changing headway to 2s	82
Figure 54 Result from alternating from long to short distance	84
Figure 55 Headway dropped to 1,5s	85
Figure 56 Distance curves behavior after headway decreased	85
Figure 57 Acceleration curve after changing the headway to 1,5s.....	86
Figure 58 Result from alternating headway from short to long	87
Figure 59 Headway increased to 2,5s	88
Figure 60 Distance curves behavior after headway increased	88
Figure 61 Speed curve after the headway increase.....	88
Figure 62 Acceleration curve after the headway increase.....	89
Figure 63 Result of the target vehicle decelerating from steady following state test.....	90
Figure 64 Headway curve after target vehicle decelerated	91
Figure 65 Distance curves after the target vehicle decelerated	91
Figure 66 Speed curve after target vehicle decelerated	91
Figure 67 Acceleration curve after the target vehicle decelerated	92
Figure 68 Result of the target vehicle accelerating from steady following state test	94
Figure 69 Headway bump after target vehicle finished accelerating	95
Figure 70 Distance curves after the acceleration of target vehicle	95
Figure 71 Speed curve after target vehicle accelerated	96
Figure 72 Acceleration curve after target vehicle accelerated	96
Figure 73 Scoring of the target vehicle accelerating test.....	96
Figure 74 Result of the target vehicle speeding off test	98
Figure 75 Headway curve before and after target vehicle speeding off	99
Figure 76 Distance curves before and after target vehicle speeding off	99
Figure 77 Speed curve after set speed is reached but target vehicle keeps accelerating	99
Figure 78 Acceleration curve during and after target vehicle speeding off.....	100
Figure 79 Result of the following target vehicle in curve test	101

List of Tables

Table 1 List of available maps in CARLA	30
Table 2 Simplified scenarios and corresponding responses from ACC system.....	38
Table 3 Criteria for assessing the ACC system	47
Table 4 Distance measurements in Meshlab and mean value	54
Table 5 Ziegler–Nichols tuning method table [52].....	56
Table 6 Scoring for the approaching scenario #1.....	74
Table 7 Scoring for the approaching scenario #2.....	78
Table 8 Scoring of the headway alternating test from long to medium distance	82
Table 9 Scoring of the headway alternating test from long to short distance	86
Table 10 Scoring of the headway alternating test from short to long distance	89
Table 11 Scoring of the target vehicle decelerating test	92
Table 12 Scoring of the target vehicle speeding off test	100

List of Abbreviations

ACC	Adaptive Cruise Control
PID	Proportional-Integral-Derivative
Radar	Radio Detection and Ranging
Lidar	Light Detection and Ranging
CARLA	Cars Learning to Act
CC	Cruise Control
ADAS	Advanced Driver Assistance Systems
SAE	Society of Automobile Engineers
UE4	Unreal Engine 4
NPC	Non-playable Character
GIF	Graphics Interchange Format
FPS	Frames per Second
RAM	Random Access Memory
AI	Artificial Intelligence
TTC	Time to Collision
RPM	Rotation per Minute

List of Symbols

t, τ	Time related variables
Δt	Time frame
d	Distance
c	Speed of light
I	<i>Actual light intensity</i>
I_0	<i>Ideal light intensity</i>
a	<i>Attenuation coefficient</i>
v_{target}	Current speed of the target vehicle
v_{ego}	Current speed of the ego vehicle
v_{set_speed}	Predefined speed set by the driver
H	Headway (time)
t_i	The time for an “i” vehicle to pass a point
K_p	Proportional gain of the PID controller
K_i	Integral gain of the PID controller
K_d	Derivative gain of the PID controller
P_{out}	Output of the proportional term
I_{out}	Output of the integral term
D_{out}	Output of the derivative term
$u(t)$	Output of the PID controller (ideal)
$e(t)$	Error function with respect to time
K_u	Ultimate gain
T_u	Oscillation period at the ultimate gain

List of Code Snippets

Code snippet 1 Euclidean distance three-dimensional.....	34
Code snippet 2 Retrieval of vehicles speeds.....	36
Code snippet 3 Class “PIDLongitudinalControl” initial arguments	41
Code snippet 4 Low-pass filter in Python.....	42
Code snippet 5 Run_step function.....	42
Code snippet 6 Building the PID controller in Python.....	43
Code snippet 7 Applying the PID controller to the vehicle.....	55

1 Introduction

1.1 Background

Adaptive Cruise Control (ACC) is an advanced driver assistance system that has been developed to enhance driver comfort and safety by reducing the drivers' fatigue. The development of ACC has been a significant milestone in the evolution of automotive technology, with the first commercial ACC systems being introduced in luxury vehicles in the late 1990s. Since then, ACC technology has evolved and is now offered as a standard or optional feature in many vehicles. The use of ACC technology has significant implications for road safety. Rear-end collisions are one of the most common types of accidents on the road, and ACC technology has been developed to prevent these types of collisions. ACC technology uses radar or lidar sensors to detect the distance and speed of other vehicles on the road, and if it senses that the vehicle in front is too close, it automatically applies the brakes to maintain a safe following distance. This technology has been shown to be effective in reducing the incidence of said collisions.

In recent years, there has been growing interest in autonomous driving technology, and ACC is a crucial component of such vehicles. ACC technology is expected to play an essential role in enabling fully autonomous driving, as it can help vehicles navigate complex traffic situations and maintain a safe distance from other vehicles on the road. The development of ACC technology has also paved the way for the development of more advanced driver assistance systems, such as lane-keeping assist and automatic emergency braking, etc. The development of ACC technology has been made possible through the use of advanced sensors and computer control systems. These systems must be able to detect and process substantial amounts of data in real-time to ensure the safe operation of the vehicle. To test and evaluate the effectiveness of ACC technology, simulation environments such as CARLA (Cars Learning to Act) have been developed to provide a realistic environment for testing and evaluating automated driving control systems.

1.2 Motivation

Designing and assessing an ACC controller in a simulation environment such as CARLA is important for several reasons. First, the design of ACC controllers requires the evaluation of the controller's performance under a wide variety of traffic situations and environmental conditions. Testing such controllers in a real-world setting would be time-consuming and expensive. However, using a simulation environment such as CARLA can provide a cost-effective and efficient way to evaluate and refine the performance of ACC controllers. Second, simulation environments such as CARLA can provide a controlled and repeatable environment for testing the performance of ACC controllers. This means that the same traffic scenarios and environmental conditions can be tested repeatedly, providing a reliable way to evaluate and compare the performance of different ACC controllers. Moreover, the use of simulation environments can also help to evaluate the performance of ACC controllers in extreme or hazardous conditions, which would be difficult or unsafe to test in a real-world setting. Third, the use of simulation environments such as CARLA can help to accelerate the development of ACC controllers. By testing and refining the performance of ACC controllers in a simulation environment, the time and costs associated with real-world testing can be reduced. This can enable developers to iterate on the design of ACC controllers more quickly, leading to faster development and deployment of the technology. Finally, the use of simulation environments such as CARLA can help to ensure the safety of ACC controllers before they are deployed in real-world settings. By testing the performance of ACC controllers in a simulation environment, potential safety issues can be identified and addressed in an early stage. This can help to minimize the risks associated with the deployment of new technologies and ensure the safety of all road users.

1.3 Objectives

The main objectives of the thesis are twofold. Firstly, the thesis aims to design an ACC system with a Proportional-Integral-Derivative (PID) controller approach for a lidar system in CARLA. The PID controller approach is a widely used control strategy that can be used to achieve stable and accurate control of a vehicle's speed in relation to the distance to the vehicle ahead. Secondly, the thesis aims to test the functionality of the designed ACC PID controller approach in CARLA under various traffic and environmental conditions. The testing of the ACC PID controller approach will involve creating various traffic scenarios in CARLA, ranging from low to high traffic density, and evaluating the performance of the ACC system under these scenarios. Additionally, the testing will also evaluate the performance of the ACC system under different environmental conditions, such as varying weather and lighting conditions. Overall, the thesis aims to contribute to the development of automated driving control systems by providing a comprehensive evaluation of the performance of an ACC PID controller approach with a lidar system in a simulation environment. The results of the testing will provide valuable insights into the behavior and limitations of the ACC system under different traffic and environmental conditions and can help to guide the development of more efficient and reliable ACC systems in the future.

1.4 Scope

The scope of the study is focused on designing and testing an ACC PID controller approach for a lidar-based system in CARLA. The study will not cover other sensor modalities, such as radar or camera-based systems, as the focus will be exclusively on the use of lidar sensors for distance measurement and speed control. The results of the study may not be directly applicable to other ACC systems that use different sensor modalities or control strategies. Other versions or some other approaches/strategies for developing ACC systems will be briefly mentioned, but not in depth. Additionally, the study will not cover other advanced driver assistance systems that may be integrated with the ACC system, such as lane departure warning or automatic emergency braking, etc. One limitation of the study is that it will be conducted entirely in a simulated environment, and as such, it may not fully capture the complexities and unpredictability of real-world traffic scenarios. However, the environment provided in CARLA is one of the most realistic, with advanced graphics and physics engines that can emulate real-world driving scenarios the closest to reality.

1.5 Overview

Chapter 2: Providing a comprehensive understanding of ACC, its different sensor usages and approaches, and their applicability in various situations, as well as going over the fundamentals of CARLA, including its capabilities and limitations.

Chapter 3: Explaining the methodology for designing and implementing an ACC controller approach, along with an evaluation of the Lidar sensor in CARLA. Also going over PID controllers comprehensively and defining a set of test scenarios to evaluate the approach's effectiveness under various traffic and environmental conditions. The goal is to ensure the approach's applicability and functionality in different scenarios.

Chapter 4: Evaluating test results gathered through the simulations, analyzing, and evaluating using a predetermine point system.

Chapter 5: Examining the entire process entails addressing the encountered challenges and obstacles. Identifying specific areas for enhancement, along with acknowledging limitations, is pivotal in this evaluation.

Chapter 6: Summarizing the study's findings, its significance in exploring the possibilities in the realm of autonomous driving.

2 Related work

2.1 Definitions

To develop a functional ACC system and successfully implement it in real-world scenarios, it is crucial to first determine the specific category in which the ACC system falls under. Precise categorization and comprehension of ACC involves analyzing its technical aspects, operational characteristics, and intended purpose. This entails thoroughly studying the relevant literature, research papers, industry standards, and existing implementations to acquire a solid foundation of knowledge regarding ACC systems. By establishing a clear understanding of ACC's classification and defining its core principles, we can effectively design, build, and deploy ACC systems that align with the desired specifications and objectives. This comprehensive understanding serves as a crucial steppingstone in the development process, facilitating the creation of accurate models, algorithms, and control mechanisms required for a reliable and efficient ACC system implementation.

2.1.1 ADAS [1]

ADAS, or Advanced Driver Assistance systems, refers to a set of automotive safety features that utilize advanced technology to enhance driver safety and improve overall driving experience. ADAS incorporates various sensors, cameras, and intelligent algorithms to detect and analyze the surrounding environment, providing real-time information and assistance to the driver. These systems are designed to assist drivers in avoiding accidents, reducing the severity of collisions, and enhancing overall road safety. ADAS technologies can provide safety warnings, automate certain driving functions, and offer additional support to the driver in critical situations.

Some of the most prominent systems included in ADAS are:

1. Adaptive Cruise Control (ACC): Automatically adjusts the vehicle's speed to maintain a safe distance from the vehicle ahead.
2. Lane Departure Warning (LDW): Alerts the driver if the vehicle deviates from its lane without signaling.
3. Forward Collision Warning (FCW): Warns the driver if a potential collision with a vehicle or obstacle ahead is detected.
4. Automatic Emergency Braking (AEB): Automatically applies the brakes to prevent or mitigate a collision if the driver does not respond to a warning.
5. Blind Spot Detection (BSD): Monitors the vehicle's blind spots and provides warnings if another vehicle is detected in those areas.
6. Parking Assistance Systems: Assists in parking maneuvers by providing guidance, automated steering, or warning of obstacles.
7. Pedestrian Detection: Identifies and alerts the driver if pedestrians are detected in the vicinity of the vehicle.

In this paper we are only going to dig deeper into the ACC system, all other systems might be mentioned in conjunction with the ACC system but will not be thoroughly explained or analyzed acutely.

2.1.2 ACC

ACC stands for Adaptive Cruise Control. It is an advanced driver assistance system that uses sensors and intelligent algorithms. Unlike traditional Cruise Control (CC) systems that only maintain a set speed, ACC takes into account the speed and distance of the preceding (or target - interchangeable) vehicle and adjusts the acceleration (throttle) or deceleration (brakes) of the ego vehicle accordingly. ACC relies on sensors such as radar, lidar, or cameras to continuously monitor the distance and relative speed of the vehicle in front (see Figure 1). It enables the driver to set a desired following distance,

and the system automatically adjusts the vehicle's speed to keep that distance from the preceding vehicle. If the vehicle ahead slows down, ACC will reduce the speed of the host vehicle to maintain the desired distance. Conversely, if the vehicle ahead accelerates or moves away, ACC can increase the speed of the host vehicle up to the set limit. Additionally, by integrating ACC with steering assistance, the vehicle gains the ability to not only regulate its speed and distance but also actively steer within its lane. This enhances the vehicle's autonomy and further reduces the driver's direct involvement in the driving process.

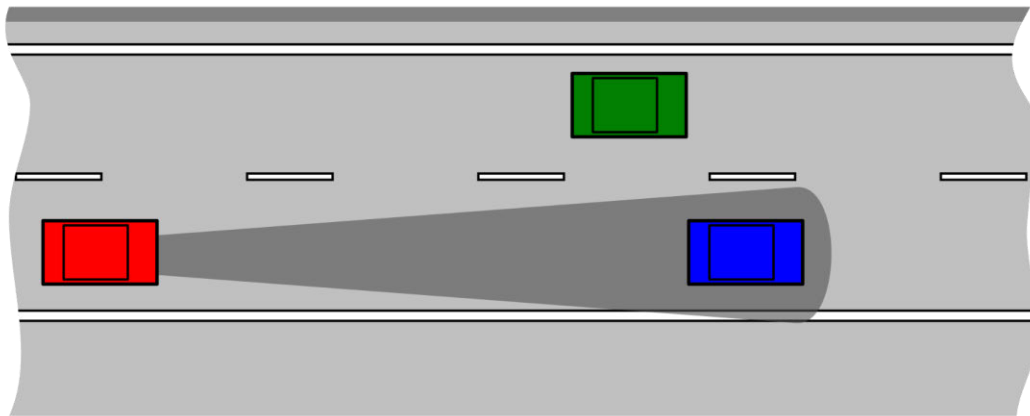


Figure 1 ACC functionality [2]

The primary purpose of ACC is to enhance driving comfort and safety on highways or in congested traffic conditions. It helps reduce the driver's workload by automating speed adjustments and maintaining a safe following distance. However, it's important for drivers to remain attentive and ready to take control of the vehicle when needed, as ACC is not designed to replace the driver's responsibility for safe operation.

2.1.2.1 Brief history

1970s: The concept of ACC began to emerge in the 1970s as researchers and engineers explored ways to enhance vehicle safety and driver comfort. Early studies focused on using radar-based sensors to measure the distance between vehicles and automatically adjust the speed of the host vehicle.

1980s: Experimental ACC systems started to be developed and tested in the 1980s. These systems utilized radar sensors mounted on the front of the vehicle to detect the distance and relative speed of the preceding vehicle. However, the technology was still in its infancy, and these early prototypes were primarily limited to research institutions and experimental vehicles.

1990s: In the 1990s, ACC technology began to make its way into commercial vehicles. Luxury automakers, such as Mercedes-Benz and BMW, introduced ACC systems as optional features in their high-end models. These systems used radar sensors to monitor the distance to the vehicle ahead and automatically adjusted the vehicle's speed by controlling the throttle and brakes.

2000s: Throughout the 2000s, ACC systems became more prevalent and accessible to a wider range of vehicles. Automakers began offering ACC as an optional feature in mid-range cars, making the technology more accessible to consumers. ACC systems evolved to include additional functionalities, such as the ability to maintain a desired time gap between vehicles and operate in stop-and-go traffic. These advancements improved convenience and safety for drivers in various traffic conditions.

2010s: The 2010s saw significant advancements in ACC technology. Manufacturers integrated ACC with other driver assistance features to create more comprehensive systems. For example, ACC was combined with lane-keeping assist to enable the vehicle to automatically steer within its lane while maintaining a set distance from the preceding vehicle. This integration marked a shift toward higher levels of automation and paved the way for the development of semi-autonomous driving technologies.

Present: ACC has become a standard feature in many modern vehicles across different price ranges. The technology continues to evolve rapidly, with the integration of advanced sensors such as lidar and cameras. These sensors provide improved object detection and enable more precise control of the vehicle's acceleration and braking. ACC is now often part of larger ADAS, which include features like automatic emergency braking, blind spot detection, and lane-keeping assist. These advancements contribute to the ongoing development of semi-autonomous and autonomous driving capabilities.

2.1.2.2 ACC and autonomy

ACC plays a significant role as a foundational component in the progression toward fully autonomous vehicles. According to the Society of Automotive Engineers (SAE) classification, driver support features at Level 1 of Driving Automation provide either steering or brake/acceleration assistance to the driver. However, reaching Level 2 requires the integration of features that offer both steering and brake/acceleration assistance. As an individual system, ACC operates at Level 1, providing the driver with assistance in maintaining a set speed and following distance. However, when combined with another driver assist feature that includes steering functionality, the vehicle can advance to Level 2 on the Driving Automation scale. This combination represents a significant step closer to achieving fully autonomous driving capabilities. It is important to note that while ACC combined with steering assistance represents Level 2 automation, achieving higher levels of autonomy requires additional advancements in technology and functionality, as defined by the SAE classification system (see Figure 2).

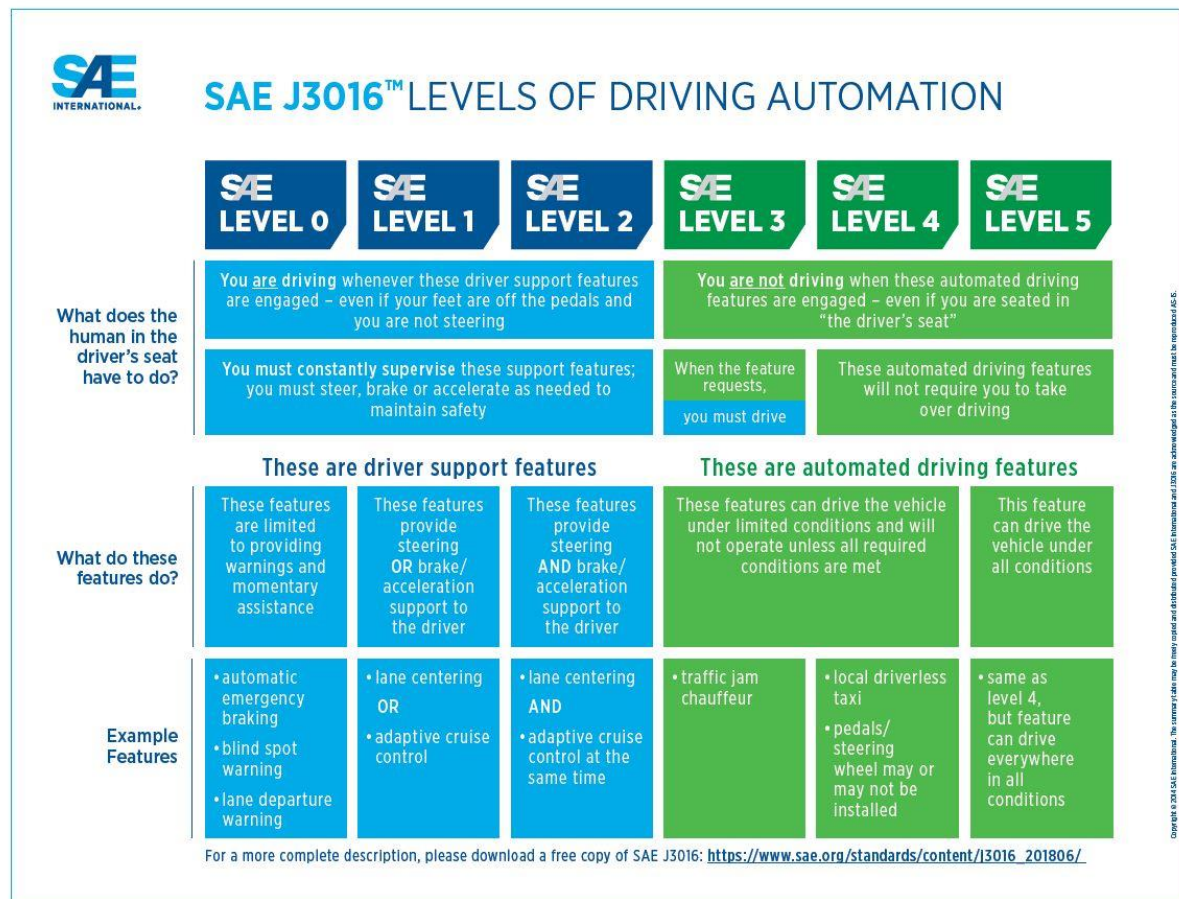


Figure 2 SAE Classification of autonomous driving levels [3]

2.1.2.3 ACC'S influence on Collisions (statistics)

Base on the research papers “Impact of adaptive cruise control (ACC) system on fatality and injury reduction in China” [4], a number of factors, for example limitations such as the applicable weather conditions, applicable light conditions, applicable road conditions, and applicable speed conditions, were considered alongside with a national-level safety impact evaluation model. The goal is to assess the safety benefits of ACC in China, including potential maximum impact as well as realistic impact. According to the research findings, a market penetration rate of 100% for ACC could potentially lead to a 5,48% reduction in fatalities and a 4,91% reduction in injuries. However, by significantly increasing the market penetration rate of ACC in the near future, the estimated reductions in fatalities and severe injuries could range from 324 to 957 and 1,035 to 2,737 respectively in 2025, and 531 to 1,579 and 1,604 to 4,242 respectively in 2030. It is important to consider the activation rate of ACC and its four main limitations when assessing these potential benefits. Therefore, after accounting for these factors, the adjusted realistic result is projected to be approximately one-third of the potential maximum result.

However, there are a surprising number of articles, take for example one from Samuel S. Monfort called “Speeding behavior while using adaptive cruise control and lane centering in free flow traffic” [5], found that ACC can actually negatively impacted the safety of the drivers on the road. In this study, an analysis of 40 drivers in the Boston area who drove either a 2016 Land Rover Range Rover Evoque or a 2017 Volvo S90 for a period of four weeks, was made. Keep in mind that the assessment of crash risk was determined through statistical analysis rather than relying on real-world crash incidents. Nonetheless, the study observed a significant increase in speeding behavior among drivers using ACC

compared to periods of manual control. Approximately 95% of drivers exhibited speeding tendencies while using ACC, whereas 77% did so when driving manually. A similar pattern emerged when drivers utilized ACC with lane centering, with 96% of them speeding compared to 77% during manual control. Moreover, drivers using ACC systems exceeded the posted speed limit by a slightly higher margin compared to those in manual control. Notably, these differences were most pronounced on limited-access roads with lower speed limits, particularly those set at 55 mph (around 90 kph). A possible explanation for this is the perception of safety. The study leans heavily on the behavioral side of the drivers and not the technical side of the ACC system. After all, ACC is still a tool, and how we use it will greatly affect the outcome, regardless of the original design purpose.

2.1.3 Radar vs. Lidar

In this sub-section, we will cover the fundamental principles of radar and lidar, as well as compare the two technologies. Additionally, we will delve into the factors that influence the selection of either sensor for this particular application.

2.1.3.1 Radar (Radio Detection and Ranging)

Radar is a radiolocation system that utilizes radio waves to determine the distance, angle, and velocity of objects in relation to the radar site. It has a wide range of applications, including detecting and tracking aircraft, ships, missiles, vehicles, as well as mapping weather patterns and terrain. Radar systems operate on the principle of transmitting radio waves, known as radar signals, and then detecting the reflected waves that bounce back from objects in the system's range. While certain aspects of radar technology can be intricate and involve advanced mathematics, the fundamental concept of radar is relatively simple. A radar system includes a transmitter that emits radar signals in a specific direction chosen by the operator. These electromagnetic waves travel outwards and subsequently return to the transmitter, where they are interpreted for further analysis.

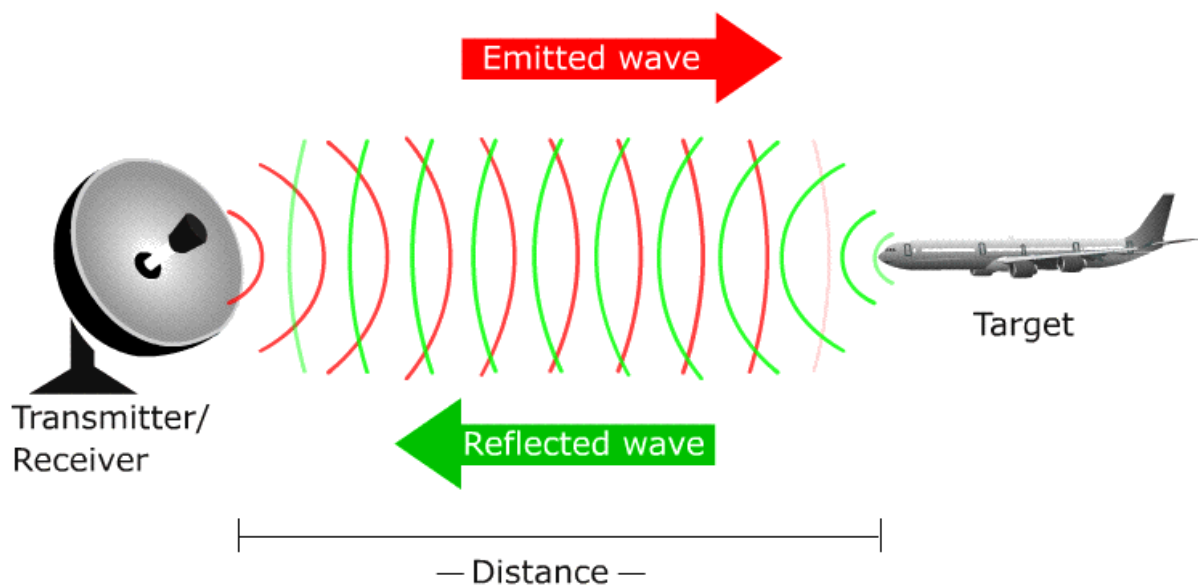


Figure 3 Working principle of a Radar system [6]

As depicted in the provided image, the fundamental idea involves transmitting radio waves from a sender or transmitter across a certain distance. These waves interact with an object, causing them to be reflected (illustrated in green) and return back along the same "Distance" towards the receiver. This concept is relatively straightforward, and with the aid of basic high school mathematics, one can

measure the time it takes for the wave to travel from the transmitter to the object and back. The formula for calculating the distance in a radar system is based on the time it takes for the radio waves to travel to the target and back. The formula can be expressed as:

$$d = \frac{c * t}{2}$$

Equation 1 Radar's equation for determining distance

where d represents the distance between the radar transmitter/receiver and the target, c is the speed of light, and t is the time taken for the radar signals to travel to the target and return to the receiver. This formula is similar to the one used in lidar, as both rely on measuring the time of travel to determine the distance. Radio waves are a form of electromagnetic radiation and they do travel at the speed of light in a vacuum. The speed of light in a vacuum is approximately 299,792,458 meters per second [7]. In practical applications, such as radar systems, the speed of radio waves is very close to the speed of light and is typically considered to be equal to the speed of light for most calculations. Radar technology emerged during World War II, initially developed for military purposes. The term "RADAR," derived from "radio detection and ranging," was coined by the United States Navy. Since then, radar has become a commonly used term in various languages, becoming a lowercase noun in general usage [8]. Radar sensors are currently being used widely in the automotive industry, since a lot of functions from ADAS require detecting and determining velocities, trajectories, of other vehicles or objects in the surroundings.

2.1.3.2 Lidar (Light Detection and Ranging)

Lidar is a technology that utilizes lasers to measure distances by targeting objects or surfaces and measuring the time it takes for the laser light to return to a receiver. The distance of an object or surface is determined by a lidar using the formula similar to radar's as mentioned above:

$$d = \frac{c * t}{2}$$

Equation 2 Lidar's equation for determining distance

where c represents the speed of light, d is the distance between the detector and the object or surface being detected, and t is the time taken for the laser light to travel from the lidar device to the object or surface and then back to the detector.

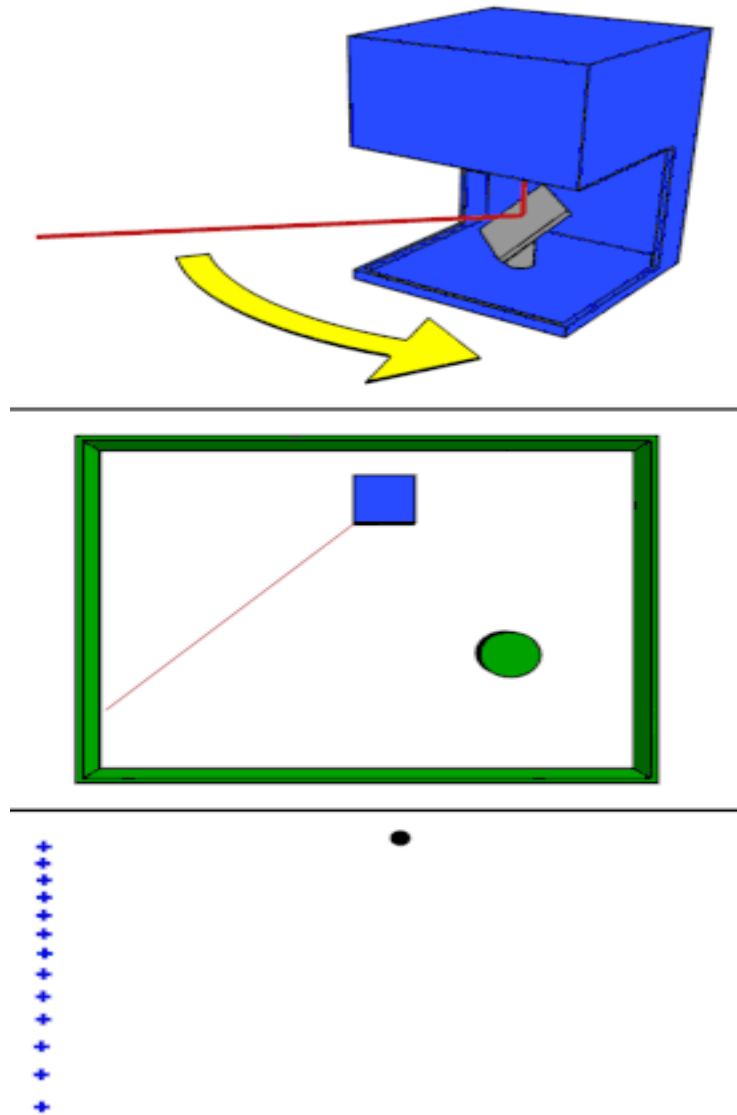


Figure 4 Working principle of a lidar system [9]

In the Figure 4 shown above, when a laser beam encounters an obstruction, a data point is recorded. This process enables the creation of precise representations of the surrounding environment in the form of point clouds. This technique allows for accurate spatial mapping by capturing and analyzing the reflected laser beams from various surfaces and objects. It is often referred to as 3D laser scanning and finds applications in various fields. Lidar is widely used for creating high-resolution maps and has applications in surveying, geodesy, archaeology, geography, geology, forestry, and more. It enables the generation of digital 3D representations of Earth's surface and ocean bottoms, particularly in coastal areas. Notably, it played a crucial role in the record-setting flights of the Mars helicopter Ingenuity, providing terrain data for guidance and safety.

In recent years, lidar has gained significant importance in autonomous vehicles for control and navigation purposes thanks to its precision, capable of detecting objects at great range, typically several hundred meters. Lidar sensors provide accurate and detailed 3D point cloud data of the vehicle's surroundings. This enables the vehicle's perception system to detect and recognize various objects such as pedestrians, vehicles, cyclists, and obstacles in real-time. Lidar helps create a

comprehensive and reliable understanding of the environment, enhancing the vehicle's ability to make informed decisions. Several factors need to be taken into account when choosing between the two sensors:

1. **Object Detection and Range:** Radar is particularly effective in detecting and tracking objects over long distances, including in adverse weather conditions such as rain or fog. Lidar, on the other hand, provides high-resolution 3D mapping and is well-suited for detecting and identifying smaller objects with greater precision, but its range is typically shorter than radar.
2. **Environmental Conditions:** Radar performs well in various environmental conditions, including low light and poor visibility. Lidar, however, relies on laser beams that can be affected by factors such as heavy rain, snow, or dust, which may reduce its effectiveness.
3. **Cost:** Radar systems tend to be more affordable compared to lidar systems, which typically involve more complex technology and higher manufacturing costs.
4. **Size and Integration:** Radar sensors can be more compact and easily integrated into vehicles due to their simpler design and fewer components. Lidar systems, with their multiple lasers and rotating mechanisms, are generally larger and require more space.
5. **Applications:** Radar is widely used in automotive applications such as adaptive cruise control, collision warning systems, and blind spot detection due to its long-range capabilities and ability to detect larger objects. Lidar is commonly used in autonomous driving systems, mapping, and advanced robotics, where precise 3D mapping and object recognition are essential.

In this research paper, lidar is chosen as the preferred sensing technology due to its remarkable precision. The selection stems from the requirement of achieving complete autonomy for vehicles, even in complex urban environments. To accomplish this goal, the sensor must possess the capability to accurately detect and identify various intricate elements in the surroundings. These elements encompass pedestrians, buildings, diverse vehicles, traffic signs, traffic lights, streetlights, and other significant details that contribute to the urban landscape. Plus, with the flexibility provided in a simulated environment, researchers are allowed to utilize the any desired tools effectively. Furthermore, the downsides of lidar are anticipated to be improved in the near future, which can offset any concerns about affordability, as well as functioning conditions such as in non-ideal weather. Considering the paper's focus on universality rather than solely commercial vehicles, the adoption of lidar technology is seen as advantageous. It is recommended to embrace lidar technology as soon as possible to leverage its benefits.

2.1.4 Proportional-Integral-Derivative (PID) Controller [10]

A PID controller, which stands for proportional integral derivative controller, is a device used to control various process variables such as pressure, flow, temperature, and speed in industrial applications. It is widely used in temperature control, scientific processes, automation, and chemical industries. The PID controller operates within a closed-loop feedback system to regulate the process variables. It continuously compares the actual output of the system with the desired setpoint and generates an error signal. Based on this error signal, the controller adjusts the system output to minimize the error and bring the process variable closer to the setpoint. The output then will be fed back into the controller as input, and the process repeats until the error is 0.

The PID controller consists of three main control modes: proportional (P), integral (I), and derivative (D). Each mode contributes to the overall control action of the controller:

- **Proportional (P) Control:** The P-controller produces an output that is proportional to the current error between the setpoint and the actual value. The output is obtained by multiplying

the error with a proportional constant. The P-controller provides stable operation but may have a steady-state error.

- **Integral (I) Control:** The I-controller integrates the error over time and continuously adjusts the output based on the accumulated error. It helps eliminate steady-state error and brings the system to the desired setpoint. However, excessive integral gain can affect system stability, and the response may be slower.
- **Derivative (D) Control:** The D-controller predicts the future behavior of the error by considering the rate of change of the error with respect to time. It provides a kick-start to the output and improves the system's response time. The D-controller compensates for the phase lag caused by the I-controller, also reduces overshoot and enhances overall system stability.

By combining these three control modes, the PID controller can achieve the desired response for the system (see Figure 5). Different manufacturers may employ different PID algorithms in their controllers.

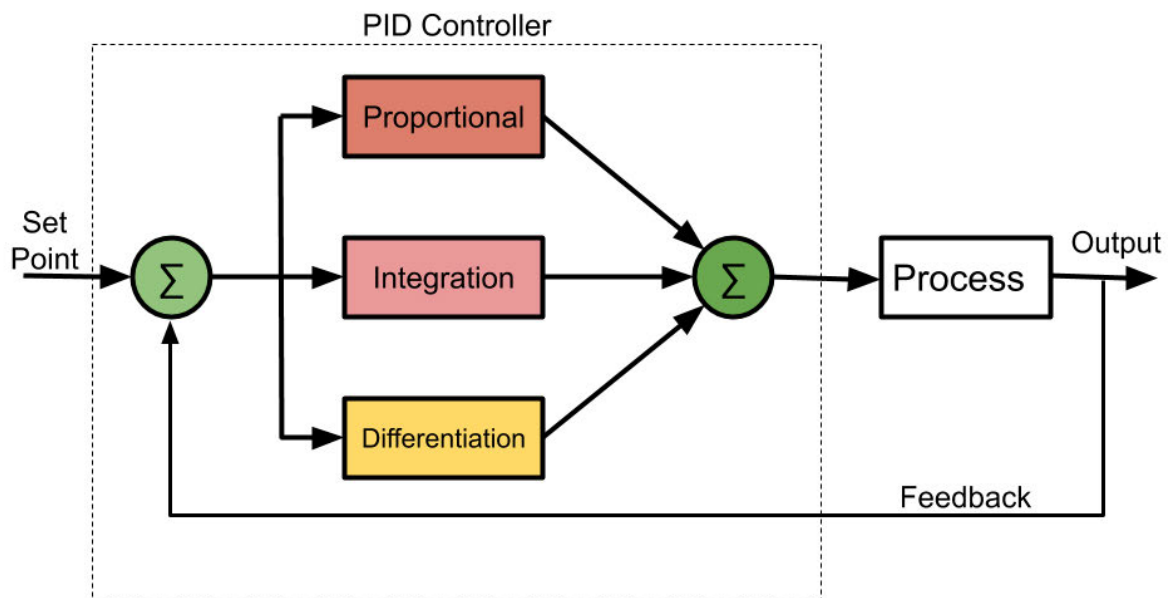


Figure 5 PID controller scheme [11]

2.1.4.1 Brief history

The PID controller has a rich history of development and refinement. In 1911, Elmer Sperry created the first PID controller. This early version laid the foundation for future advancements in control systems. In 1933, the Taylor Instrumental Company introduced a pneumatic controller that was fully tunable. This marked a significant step forward in control technology, as it allowed for precise adjustments to be made to the controller's parameters. Control engineers recognized a persistent issue with proportional controllers, namely the presence of steady-state error. To address this problem, they introduced the proportional-integral (PI) controller. By including an integral component, the PI controller was able to eliminate the steady-state error by continuously integrating the error signal over time. In the 1940s, the first pneumatic PID controller was developed, incorporating derivative action to mitigate overshooting issues. The derivative component anticipated the future behavior of the error signal by measuring its rate of change, thereby improving the system's response. In 1942, Ziegler and Nichols introduced tuning rules to facilitate the determination of suitable parameters for PID controllers. These rules provided a systematic approach for engineers to

optimize the performance of PID controllers in different applications. During the mid-1950s, automatic PID controllers became widely utilized in various industries. Their effectiveness in regulating process variables and their ease of implementation contributed to their popularity. The historical progression of the PID controller demonstrates the iterative development and refinement of control systems over time, leading to their widespread adoption and application in industrial processes.

2.1.4.2 Tuning methods

Before the PID controller can effectively function, it needs to be tuned to match the dynamics of the process being controlled. Default values for the proportional, integral, and derivative terms provided by the designers may not always deliver the desired performance and can sometimes result in instability or slow control response. Therefore, different tuning methods have been developed to find the optimal values for the PID gains, requiring careful consideration from the operator. There are various techniques available for tuning PID controllers, which include the following main methods [12]:

1. Trial and Error Method (heuristic tuning): This is a straightforward approach to tuning a PID controller. The controller is adjusted while the system is operational. In this method, the integral (K_i) and derivative (K_d) terms are initially set to zero, and the proportional (K_p) term is gradually increased until the system exhibits oscillatory behavior. Once the system is oscillating, the integral term is adjusted to eliminate the oscillations, followed by fine-tuning of the derivative term to achieve a fast response.
2. Process Reaction Curve Technique: This is an open-loop tuning technique where a step input is applied to the system, and the resulting response curve is recorded. By analyzing the curve and determining parameters such as slope, dead time, and rise time, the values of the PID gains can be calculated and adjusted accordingly.
3. Zeigler-Nichols Method: Zeigler-Nichols proposed closed-loop methods for tuning PID controllers, namely the continuous cycling method and the damped oscillation method. The procedures for both methods are similar, but the oscillation behavior differs. In this method, the proportional gain (K_p) is gradually increased while the integral (K_i) and derivative (K_d) terms are set to zero. The gain at which the system exhibits constant oscillations is called the ultimate gain (K_u), and the period of oscillations is known as the ultimate period (P_c). Based on these values, the PID gains can be determined using Zeigler-Nichols tables specific to the controller type (P, PI, or PID).

The tuning process, as well as specific design of the PID controller will be discussed in detail further in the later sections of this paper.

2.2 The diversity of development strategies for ACC

There are various approaches and technologies employed in the development of ACC systems, each with its own methodology and level of complexity. These approaches can be ranked from simple to complicated based on their control strategies. From simpler method like conventional PID controller, to more complex approaches such as Fuzzy-Logic Controller [13], Model Predictive Control [14], and Neural Networks and Machine Learning [15] [16]. Although artificial intelligence (AI) and other similar machine learning methods are often seen as the future of control systems, there are still valid reasons to test and consider simpler approaches like PID control. First, PID control has a long-standing track record of successful implementation in various industries and applications, providing reliable and robust control. Its simplicity allows for straightforward tuning and understanding of the control parameters. Second, PID control requires fewer computational resources, making it suitable for

systems with limited processing capabilities. This reduces the complexity and cost of implementation. Third, PID control is more interpretable and transparent compared to AI algorithms, enabling engineers to analyze and diagnose system behavior effectively. Lastly, PID control can serve as a baseline or benchmark for evaluating the performance of more advanced control strategies. By comparing the results obtained from PID control with those of more complex approaches, it becomes easier to assess the effectiveness and potential benefits of AI-based methods. Therefore, despite the growing popularity of AI, exploring and testing simpler approaches like PID control remains valuable for practical control system design and development.

2.3 CARLA Simulation Environment [17]

2.3.1 Introduction

According to the CARLA paper [17], the challenges of sensorimotor control in three-dimensional environments are first discussed, particularly in the context of autonomous ground vehicles navigating densely populated urban environments. It highlights the complexities involved, such as multi-agent dynamics at traffic intersections, tracking and responding to the motion of numerous actors, recognizing street signs and traffic rules, dealing with rare events, and reconciling conflicting objectives. The research in autonomous urban driving is hindered by the high infrastructure costs and logistical difficulties of training and testing systems in the physical world. Operating robotic cars requires significant funds and manpower, and collecting sufficient data to cover various corner cases is challenging. The paper suggests using simulation as an alternative for training and validating driving strategies. Simulation can democratize research in autonomous urban driving and allows for system verification in dangerous scenarios that cannot be replicated in the physical world. While simulation has been used in autonomous driving research, existing simulation platforms have limitations. Open-source racing simulators lack the complexity of urban driving, while commercial games (e.g., Grand Theft Auto franchise) with high-fidelity urban environments lack customization, control, and detailed benchmarking capabilities required for autonomous driving research.

Thus, introducing a solution, CARLA, an open simulator specifically designed for urban driving. CARLA supports training, prototyping, and validation of autonomous driving models, including perception and control. It provides urban environments with customizable layouts, various vehicle models, pedestrians, street signs, and more. CARLA allows flexible setup of sensor suites and provides signals for training driving strategies. It offers a wide range of environmental conditions, such as weather and time of day. CARLA can and will be used to evaluate the performance of three approaches to autonomous driving: a classic modular pipeline, a deep network trained via imitation learning, and a deep network trained via reinforcement learning. Experiments are conducted with controlled navigation scenarios of increasing difficulty, manipulating the complexity of the route, the presence of traffic, and environmental conditions. The results provide insights into the performance characteristics of the three approaches.

2.3.2 Simulation Engine

CARLA is built for flexibility and realism in rendering and physics simulation. It is implemented as an open-source layer over Unreal Engine 4 (UE4), allowing for future extensions by the community. UE4 provides state-of-the-art rendering quality, realistic physics, basic NPC logic, and an ecosystem of interoperable plugins. The engine itself is free for non-commercial use. CARLA functions as a server-client system (see Figure 6), where the server runs the simulation and renders the scene, while the client API, implemented in Python, facilitates interaction between the autonomous agent and the server through sockets. The client sends commands and meta-commands to the server and receives sensor readings in return. Commands control the vehicle's steering, acceleration, and braking, while meta-commands control the behavior of the server, such as resetting the simulation, changing environmental properties, and modifying the sensor suite. Environmental properties include weather conditions, illumination, and the density of cars and pedestrians. When the server is reset, the agent is re-initialized at a new location specified by the client.

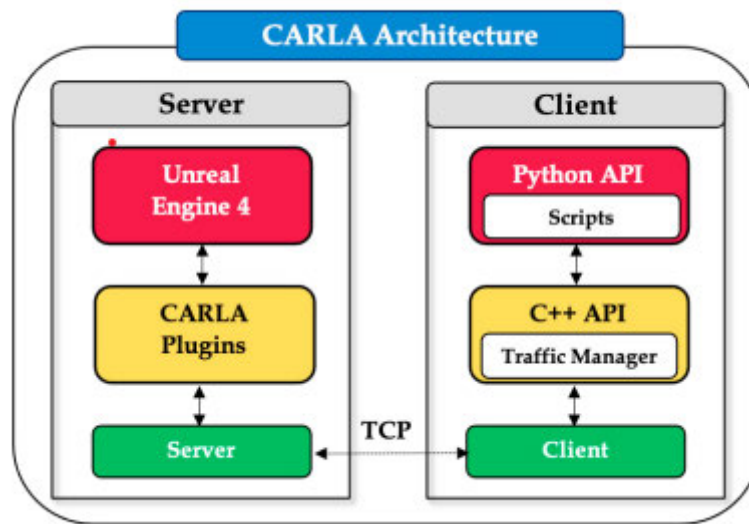


Figure 6 CARLA client-server architecture [63]

The environment in CARLA consists of 3D models of static objects like buildings, vegetation, traffic signs, infrastructure, as well as dynamic objects like vehicles and pedestrians. The models are designed to balance visual quality and rendering speed, utilizing low-weight geometric models and textures while maintaining visual realism through material craftsmanship and variable level of detail. The 3D models share a common scale and reflect real-world object sizes. The simulator includes two towns: Town 1 for training with 2.9 km of drivable roads and Town 2 for testing with 1.4 km of drivable roads. CARLA incorporates realistic non-player character (NPC) behavior. The non-player vehicles are based on the standard UE4 vehicle model (PhysXVehicles), with adjusted kinematic parameters for realism. A basic controller governs non-player vehicle behavior, including lane following, respecting traffic lights, speed limits, and decision-making at intersections. Vehicles and pedestrians can detect and avoid collisions. Advanced NPC controllers can be integrated in the future. Pedestrians navigate the streets based on a town-specific navigation map, encouraging them to use sidewalks and marked road crossings while allowing them to cross roads at any point. If a pedestrian collides with a car, the pedestrian is removed from the simulation and replaced with a new one after a brief interval.

In terms of sensors, CARLA allows for flexible configuration of the agent's sensor suite. At present, sensors are limited to RGB (Red-Green-Blue) cameras and pseudo-sensors providing ground-truth depth and semantic segmentation. The number, type, and position of cameras can be specified by the

client, including parameters such as 3D location, orientation, field of view, and depth of field. The semantic segmentation pseudo-sensor provides 12 semantic classes, including road, lane-marking, traffic sign, sidewalk, fence, pole, wall, building, vegetation, vehicle, pedestrian, and others. Additionally, CARLA provides measurements related to the agent's state, such as location, orientation, speed, acceleration, and impact from collisions. It also provides information on compliance with traffic rules, such as the percentage of the vehicle's footprint impinging on wrong-way lanes or sidewalks, and the states of traffic lights and speed limits. Exact locations and bounding boxes of all dynamic objects in the environment are accessible as well, which are crucial for training and evaluating driving policies.

2.3.3 CARLA coordinates system

CARLA is built on the Unreal Engine 4, which employs a Z-up left-handed coordinate system for its world coordinates (Figure 7 – disregard the KITTI, is another system and not relevant). However, in the local coordinate system of actors' LIDAR, there is a distinction. In this particular coordinate system, the x-axis (roll) indicates the forward direction, the y-axis (pitch) indicates the right direction, and the z-axis (yaw) indicates the upward direction. The origin of the vehicle (0,0,0) lies in the middle of the vehicle.

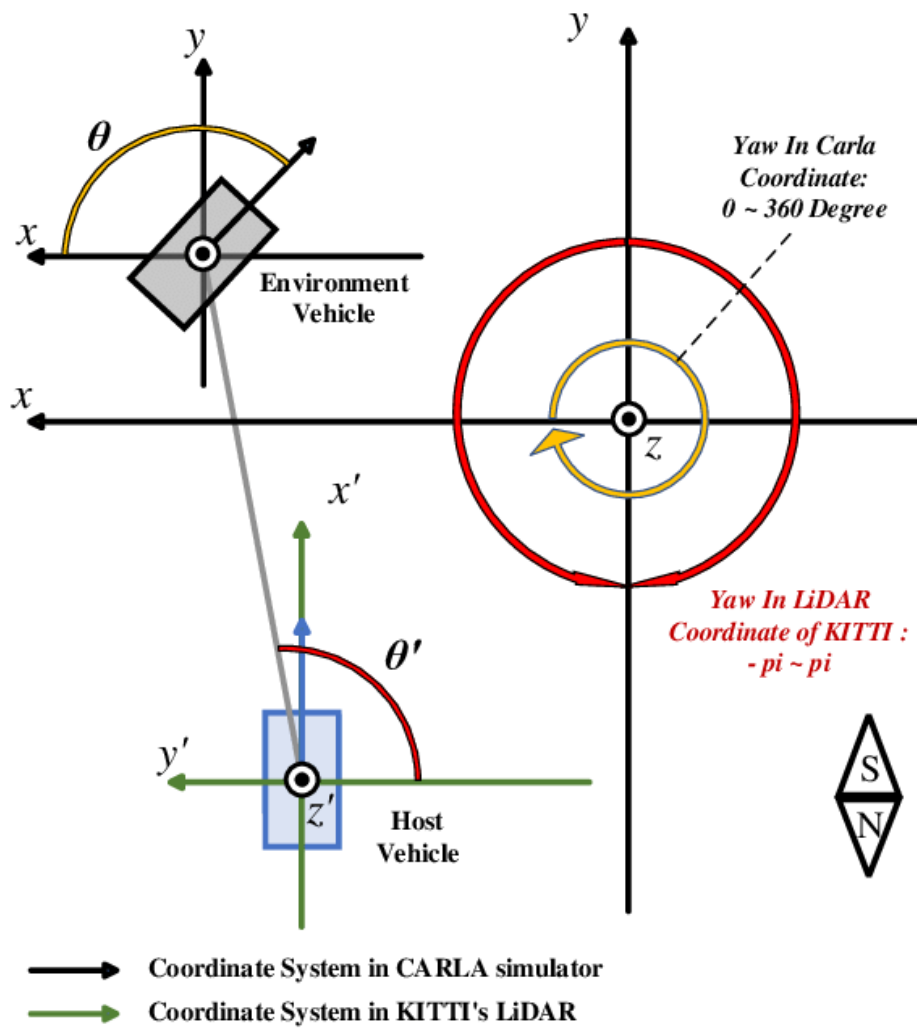


Figure 7 CARLA Coordinate System [18]

2.3.4 Lidar model in CARLA [19]

The model sensor emulates a rotating lidar system by using ray-casting techniques. To create the point cloud, lasers are positioned across the vertical field of view and their output is calculated for each channel. The rotation of the LIDAR is simulated by determining the horizontal angle it has rotated within a frame. The point cloud is generated by performing ray-casting for each laser during each iteration. The following equation is used to calculate the number of points per channel for each step:

$$\text{points_per_channel_each_step} = \frac{\text{points_per_second}}{\text{FPS} * \text{channels}}$$

Equation 3 Lidar points per channel per step equation

A single LIDAR measurement consists of a package containing all the points generated within a specific interval of time, which is typically 1 divided by the frames per second ($\frac{1}{\text{FPS}}$). It's important to note that during this interval, the physical state of the scene being observed remains unchanged. As a result, all the points within a measurement reflect the same "static picture" of the scene captured at that particular moment in time.

The output of the LIDAR sensor in CARLA simulation consists of a cloud of simulation points. This point cloud can be iterated through to retrieve a list of `carla.Location` objects, which represent the locations of the points in the simulated environment (see Figure 8).



Figure 8 Lidar's point cloud [20]

These points can be output as a .ply¹ file and imported in Meshlab² (see Figure 9) for further analyzing, specifically for the lidar sensor evaluation section in methodology.

¹ A type of data that Meshlab uses.

² A program with the function of importing 3D point clouds and numerous tools for the purpose of analyzing these point clouds.

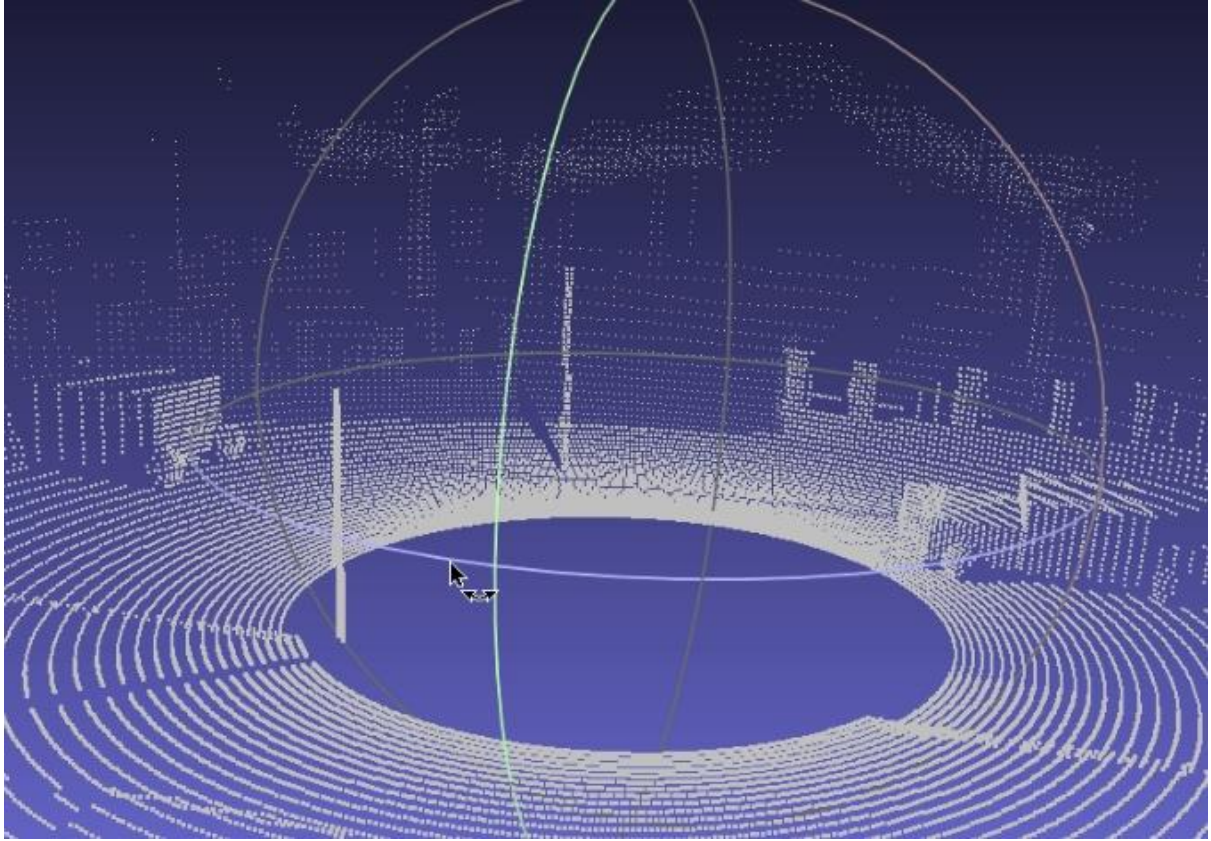


Figure 9 Point cloud imported to Meshlab [21]

The information contained in a LIDAR measurement is encoded as 4D points. The first three dimensions represent the spatial coordinates (x, y, z) of the points in XYZ coordinates, while the fourth dimension represents the intensity loss of the LIDAR signal during travel. This intensity loss is computed using the following formula:

$$\frac{I}{I_0} = e^{-a \cdot d}$$

Equation 4 Lidar intensity loss equation

a: The attenuation coefficient, which determines the intensity loss of the LIDAR signal during travel, can depend on factors such as the sensor's wavelength and the atmospheric conditions. In CARLA, this attenuation coefficient can be modified using the LIDAR attribute "atmosphere_attenuation_rate". Additionally, the distance (d) from the hit point to the sensor is another factor that influences the intensity loss. To enhance realism, it is possible to introduce dropped points in the point cloud. This simple technique simulates the loss of points in the LIDAR data due to external disturbances or perturbations. This can be achieved in 2 ways:

- **General Drop-off:** The general drop-off refers to the proportion of points that are randomly dropped from the point cloud. This drop-off process occurs before the actual tracing, which means that the dropped points are not calculated, resulting in improved performance. For example, if the drop-off rate is set to 0.5, half of the points will be dropped randomly from the point cloud.
- **Intensity-based Drop-off:** In addition to the general drop-off, an extra drop-off process can be applied based on the intensity of each detected point. This drop-off is determined by two parameters: `dropoff_zero_intensity` and `dropoff_intensity_limit`. The drop-off probability for points with zero intensity is controlled by the `dropoff_zero_intensity` parameter. On the other

hand, the drop-off intensity limit sets a threshold above which no points will be dropped. For points within the intensity range between zero and the intensity limit, the drop-off probability is calculated as a linear proportion based on these two parameters.

The “noise_stddev” attribute introduces a noise model that simulates unexpected deviations commonly encountered in real-life sensors. When “noise_stddev” is set to a positive value, each point in the point cloud is randomly perturbed along the vector of the laser ray. This means that the angular positioning of the LIDAR sensor remains perfect, but the distance measurements become noisy due to the added perturbations. The noise_stddev value controls the standard deviation of the noise distribution, influencing the magnitude of the perturbations applied to the distance measurements.

The rotation of the LIDAR can be adjusted to cover a specific angle during each simulation step, assuming a fixed time-step. For instance, to achieve a full circle output where the LIDAR rotates once per step, the rotation frequency and the simulated FPS should be equal. This synchronization ensures that the LIDAR completes a full rotation within the duration of a single simulation step, resulting in a complete circle of coverage in the generated output.

2.3.5 Vehicle model in CARLA

The selection of the model for our ego vehicle is crucial as different vehicles have distinct properties, including speed limits and handling characteristics. Although the specific information about the chosen model is currently lacking, extensive testing with various vehicles has shown consistent results. In CARLA, the virtual models are designed to closely resemble their real-life counterparts. Therefore, we can assume that any information obtained from the real-life models can be extrapolated and applied to those in the simulations. Some other notable vehicle physics that can be tweaked in the simulation [22], allowing for more freedom in varying the scenarios:

- “max_rpm”: The maximum revolutions per minute (RPM) of the vehicle's engine.
- “moi”: The moment of inertia of the vehicle's engine, measured in kilogram square meters.
- “use_gear_autobox”: If set to True, the vehicle will have an automatic transmission.
- “gear_switch_time”: The time taken to switch between gears, measured in seconds.
- “clutch_strength”: The clutch strength of the vehicle, measured in kilogram meter squared per second.
- “mass”: The mass of the vehicle, measured in kilograms.
- “drag_coefficient”: The drag coefficient of the vehicle's chassis.

2.3.3.1 Tesla Model 3

One of the choice for the target vehicle will be the Tesla Model 3. Below is the picture taken from the official CARLA website (Figure 10), along with some additional information:



Figure 10 Tesla model 3 in CARLA [57]

- Manufacturer: Tesla
- Model: Model 3
- Class: Standard
- Generation: 1
- Blueprint ID: vehicle.tesla.model3
- Base type: car
- Special type: electric
- Has lights: False
- Has opening doors: False

2.3.3.2 Dodge Charger Police 2020

As for our ego vehicle, the Police Dodge Charger 2020 is selected (see Figure 11). Note that some scenarios will also use this vehicle as the target vehicle.



Figure 11 Dodge charger police 2020 in CARLA [23]

Below is the picture and some additional information:

- Manufacturer: Dodge
- Model: Police Charger 2020
- Class: Standard
- Generation: 2
- Blueprint ID: vehicle.dodge.charger_police_2020
- Base type: car
- Special type: emergency
- Has lights: True
- Has opening doors: True

The properties can also be tweaked with the parameters mentioned above to enhance the realism of the simulations. Specifically, these were tweaked as follows [24]:

- max_rpm = 6400 (performance document)
- moi = 1,0
- use_gear_autobox = True
- gear_switch_time = 0,2 (experimental)
- clutch_strength = 10 (default value)
- mass = 2400 (maximum load)
- drag_coefficient = 0,25 (default common value for a sedan)

The parameters were extracted from the real-world version of the vehicle. Any parameters that have not been explicitly mentioned will be assigned default values provided by CARLA. Consider an instance where the torque curve is inaccurately configured, impeding the vehicle's ability to shift gears. In cases where the torque curve is not explicitly defined in the script, the vehicle functions in the expected manner, seamlessly shifting gears according to specific scenarios. This suggests that CARLA might inherently incorporate the accurate torque curve for the vehicle by default. Another scenario pertains to configuring the mass using the aforementioned command. When the mass parameter is absent

from the script (other parameters stay the same), CARLA noticeably employs a significantly smaller value, leading to a prolonged oscillation period towards the end. Conversely, in the scenario where the mass is explicitly set, the equilibrium state is already attained within the same time frame without the extended oscillations (see Figure 12). Additionally, the acceleration changing frequency is also noticeably higher. However, the specific default values are not accessible.

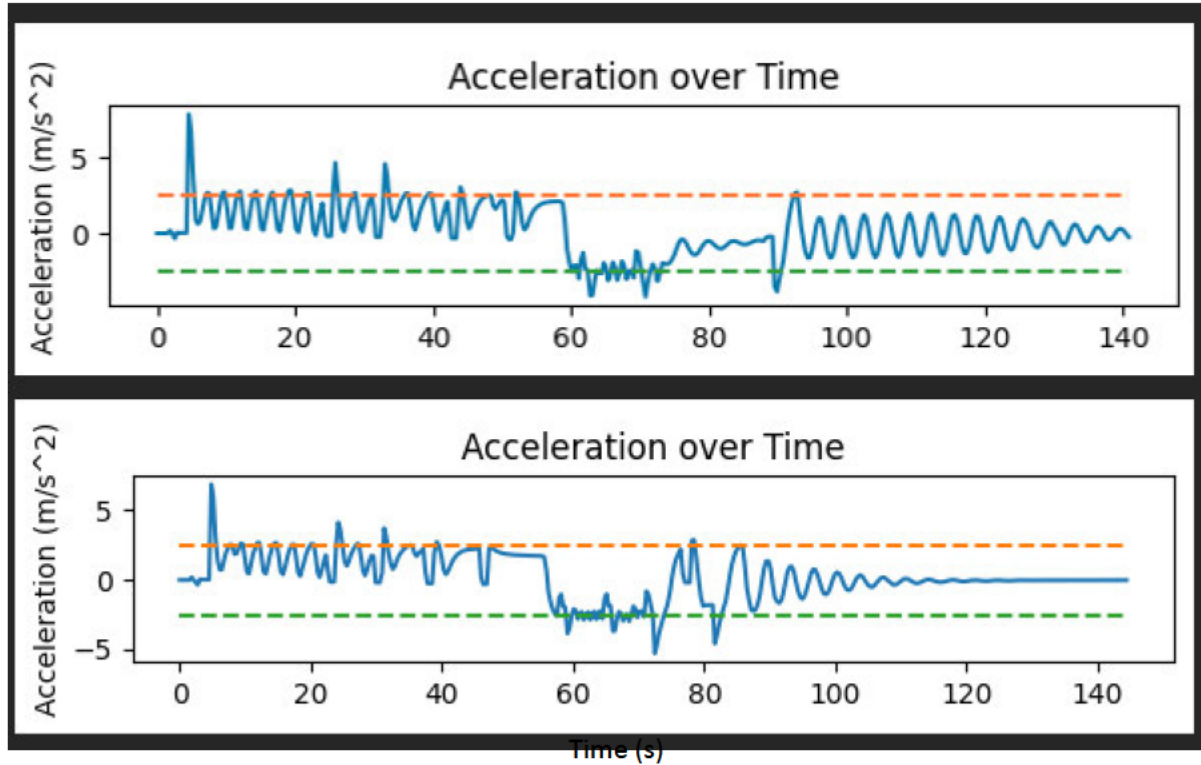


Figure 12 Comparing acceleration curve with/without a set mass. Above: without set mass, below: with set mass

2.3.6 CARLA maps [25]

A total of 8 non-layered maps are featured in CARLA basic map package as shown in Table 1:

Towns	Characteristics
01	A basic town layout with "T" junctions
02	Similar to Town01, but smaller in size
03	The most complex town, having a 5-lane junction, a roundabout, unevenness, a tunnel, and more
04	An infinite loop with a highway and a small town
05	A squared-grid town with cross junctions and a bridge and multiple lanes per direction
06	Long highways with many highway entrances and exits
07	A rural environment with narrow roads, barns and hardly any traffic lights
10	A city environment with different environments and more realistic textures

Table 1 List of available maps in CARLA

In order to evaluate the basic functionality of the ACC system, town number 6 (Figure 13) is recommended as the testing environment. This town offers longer multi-lane roads, which makes it ideal for testing the ACC system's ability to follow a preceding vehicle in a one-on-one scenario. The more distance we have, the better, since it takes a decent distance for the vehicles to reach the desired states we wanted. The availability of such roads in town number 6 simplifies the evaluation process and provides a convenient scenario to assess the performance of the ACC system.



Figure 13 Town 6 in CARLA [60]

To test the more advanced functions of the ACC system or increase the complexity of the experiment, towns such as town 3 or town 10 can be chosen. These towns offer environments with urban areas, which present different challenges compared to straight roads. By transitioning from simulations on straight roads to towns with more curves, it becomes possible to further examine the capability of the ACC system to handle complex urban driving scenarios. This allows for a more comprehensive evaluation of the ACC system's performance in a variety of real-world driving conditions.

3 Methodology

The purpose of the methodology section is to provide a comprehensive description of the approach, techniques, and procedures employed in designing and testing the ACC controller in CARLA. It serves to ensure replicability by documenting the methods used, promoting transparency, and enabling the validation of findings. Additionally, it establishes the validity and reliability of the study by outlining the experimental design and addressing potential biases or limitations. The methodology section also serves as a guide for readers interested in applying similar techniques or conducting similar experiments, providing insights into decision-making, tools used, and processes followed. It facilitates the evaluation of the ACC controller's performance by explaining the metrics, measurements, and test scenarios employed. Furthermore, it identifies challenges, limitations, and areas for improvement, offering inspiration for future research and development in the field of adaptive cruise control and autonomous driving systems.

3.1 ACC Controller Design and its implementation in Python

The goal of the ACC controller is to achieve both cruise control and adaptive control by maintaining a desired distance from the preceding vehicle. Different traffic situations can be classified into several scenarios, including steady following, preceding vehicle emergency braking, preceding vehicle emergency accelerating, vehicle cutting in, vehicle cutting out, etc. Furthermore, the transition between different scenarios should be smooth, ensuring driver comfort and safety. The ACC system should strive to provide a seamless and comfortable driving experience while prioritizing driver safety throughout the switching process.

3.1.1 ACC structure

The control system consists of three main components: an observer, an upper-level controller, and lower-level controllers. The observer utilizes measured feedback signals, including wheel angular velocities and acceleration, to estimate unmeasurable parameters such as vehicle mass, aerodynamic coefficient, and road slope. Additionally, information about the lead vehicle is obtained through sensors like radar/lidar and vision sensors. The upper-level controller utilizes all the measured feedback signals and information to analyze the data and determine the desired states and acceleration for the host vehicle. The desired acceleration commands generated by the upper-level controller are then used to control the lower-level controllers, including the engine control unit, transmission controller, and brake controller. These lower-level controllers execute the desired acceleration commands to manage the respective systems in the vehicle.

Figure 14 illustrates the overall structure of the ACC system, encompassing the driver, the upper-level controller, the lower-level controllers, the host vehicle dynamics, and the environment. The driver plays a crucial role in deciding whether to activate or deactivate the ACC system. They also have control over setting the desired velocity and the desired time headway. The desired velocity represents the target speed of the host vehicle in the absence of any vehicles or obstacles ahead. On the other hand, the desired time headway denotes the duration it would take for the lead vehicle and the host vehicle to collide if the lead vehicle were to suddenly come to a halt while the ego vehicle maintains its initial speed.

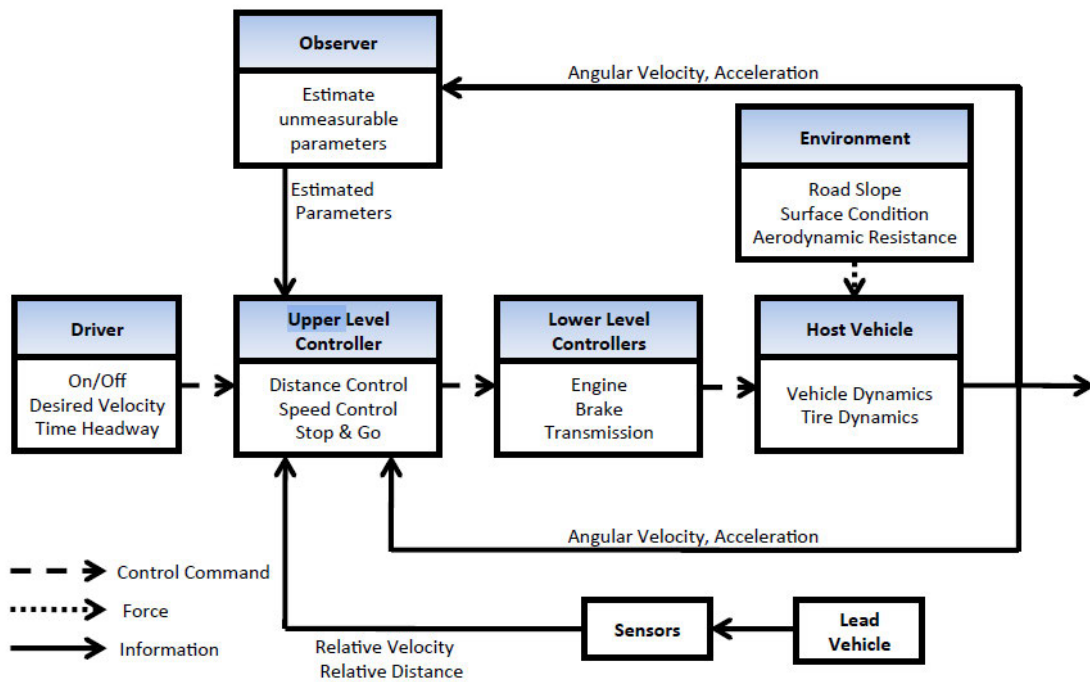


Figure 14 Overall ACC system structure [26]

3.1.2 ACC principles and algorithms

The basics of the ACC system were already discussed in former sections. This section, however, delves deeply into how ACC systems maintain safe following distances and adapt to changing traffic conditions, providing readers with a foundational understanding of this critical aspect of this modern driver assistance systems.

3.1.2.1 Environmental detection

In practical implementations of ACC systems, a forward-facing camera is typically essential for the system to accurately detect and recognize preceding vehicles. The camera captures real-time visual information that is processed using various image processing algorithms, such as machine learning, deep learning, neural networks, or template matching, ... In this specific scenario however, an RGB camera is placed on the windshield. Rather than utilizing algorithms for detecting target vehicles, the camera serves the purpose of providing a clear and detailed visibility from the perspective of the driver, helping the task of observing the experiments. Regarding distances, in a number of studies, millimeter-wave radar is used for detecting preceding vehicles [27]. To overcome its limitation in detecting static objects, researchers have explored data fusion techniques, combining millimeter-wave radar with smart cameras [28]. Tracking algorithms for curve driving scenarios have been a focal point in [29] and [30], while [31] proposed a multi-vehicle environment detection method using millimeter-wave radar, smart camera, and lidar. Additionally, [32] focused on selecting the primary target from multiple vehicle situations. The primary sensor for perceiving the surrounding environment in this scenario is a lidar sensor, which measures the distance to preceding vehicles.

It is, however, in the 3D environment, very time consuming to use the data from the lidar sensor in real time to compute the distance between the two vehicles since the lidar's outputs consist of thousands of points in a point cloud, which constantly will be updated in a relatively small timeframe (e.g., once every 0.033s). One proposed way to achieve this is to use the Euclidean distance formula (2 dimensional - Equation 5) (see also Figure 15):

$$d(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2}$$

Equation 5 Euclidean distance in Cartesian coordinates of two-dimensional space

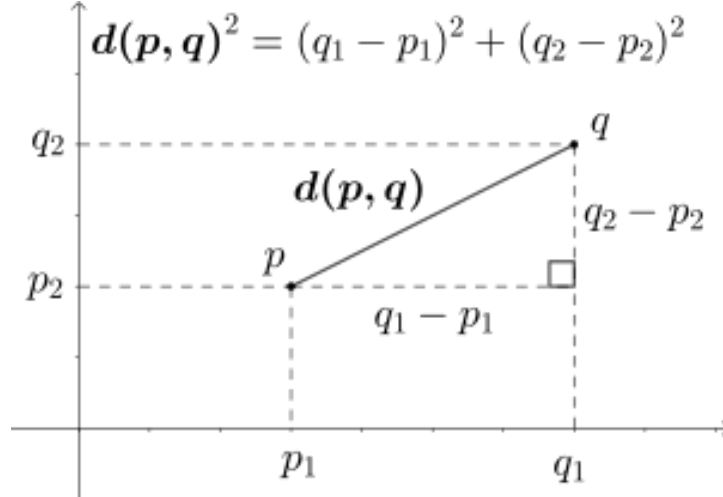


Figure 15 Euclidean distance formula two-dimensional [33]

The Euclidean distance, also known as the Pythagorean distance, is a mathematical measure of the length between two points in Euclidean space. It is determined by the Cartesian coordinates of the points and can be calculated using the Pythagorean theorem. In CARLA we have 3 dimensions, so the equation became:

$$d(P, Q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + (p_3 - q_3)^2}$$

Equation 6 Euclidean distance in Cartesian coordinates of three-dimensional space

where $p_1, q_1, p_2, q_2, p_3, q_3$ are the corresponding coordinates of the two points P and Q in 3D space (see Figure 16).

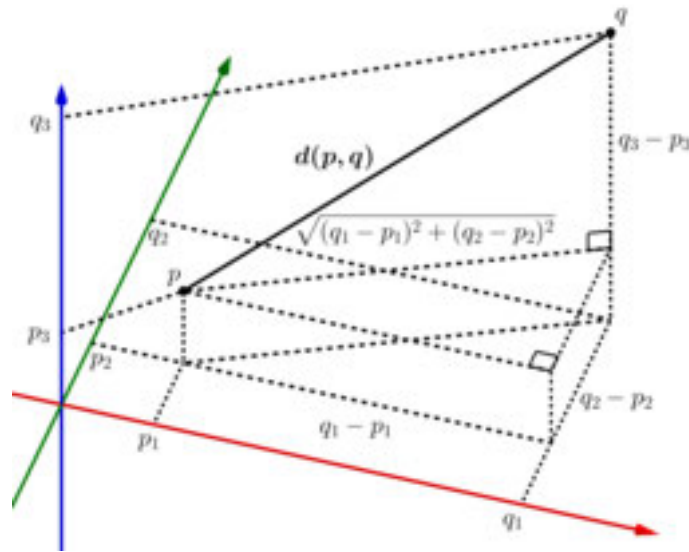


Figure 16 Euclidean distance three-dimensional [33]

Determining the appropriate points for applying the Euclidean distance equation poses another challenging problem. Some efforts have been made to define a specific set of points, which are the intersection points between the casted rays of the lidar and the bounding box [34] of the target vehicle. Subsequently, the Euclidean distances are computed between the origin coordinates of the LIDAR and these intersecting points. Finally, considering the existence of multiple intersecting points, the average distance from the previous step is calculated and designated as the distance, with a slight adjustment of subtracting approximately half of the car's length since the lidar is positioned on top of the ego vehicle. The real-time distance calculation method is expected to provide highly accurate results. However, it is anticipated that this approach may be time-consuming. As an alternative solution, another approach is proposed: involves utilizing the same Euclidean distance equation as depicted in Code snippet 1, but instead of calculating via the point cloud, the distance will be determined from the coordinates of the center of the ego vehicle, to the center of the target vehicle. Both coordinates are taken directly from the CARLA client.

```
distance = math.sqrt((location2.x - location1.x)**2 + (location2.y -
location1.y)**2 + (location2.z - location1.z)**2)
```

Code snippet 1 Euclidean distance three-dimensional

The code provided calculates the Euclidean distance between two points ("location1" and "location2") in three-dimensional space, where location 1 is the ego vehicle's origin, location 2 is the target vehicle's origin. It uses the formula for Euclidean distance mentioned above, which involves taking the square root of the sum of the squared differences between the corresponding coordinates of the two points along each axis (x, y, and z). By applying this formula, the code calculates the straight-line distance between the two points, giving a measure of their separation in three-dimensional space. It is important to note that when calculating this distance, the measurement is taken from the center points of the two vehicles. To determine the actual distance from bumper to bumper, it is necessary to subtract approximately one car-length from the measured distance.

As for whether this distance can/should be used for the simulations going forward, an evaluation of the lidar sensor is set, purposely to see if the two distance values are within acceptable range of another. It involves importing the point cloud data into Meshlab [35]. Within Meshlab, the mean distance values can be calculated for a single frame. This alternative method will be further investigated and explored in the testing methodology section. Additionally, note that not all distances are from two perfectly overlapping vehicles (see Figure 17) but will be calculated as so.

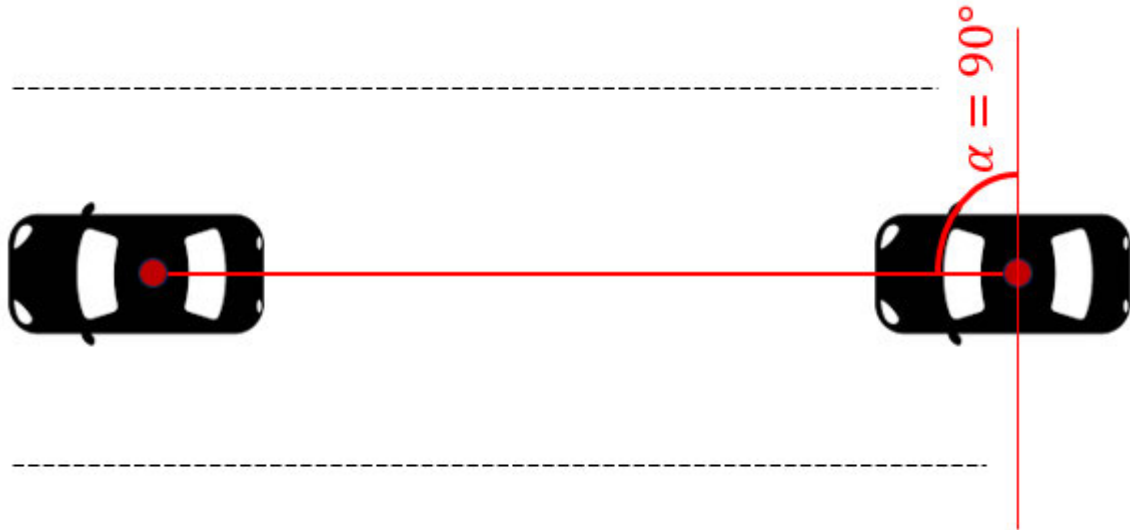


Figure 17 Two 100% overlapping vehicles

It is more than likely that most of the distances calculated in the simulation will be in some form of a diagonal line (see Figure 18). In most cases, other than purposely testing in curves, these inconsistencies amount to a very small percentage, and lies within the range of acceptable tolerance (α is reasonably small). If not, the result of that particular simulation will be marked as non-valid. This can be tested by calculating the Euler angle [36] within the CARLA simulation, using the transformation vectors of the two vehicles. However, it is observable with our eyes and the view from the front camera when the ego vehicle is deviating too far from the path of the target vehicle. Therefore, carefully observing the simulation will be sufficient for this evaluation.

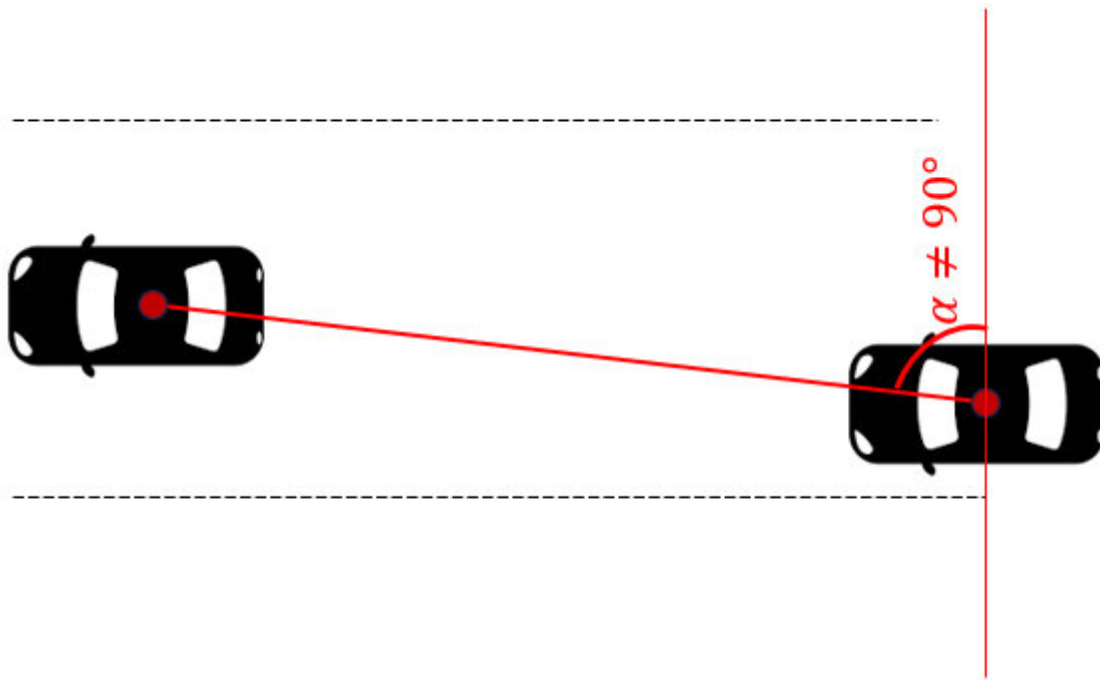


Figure 18 Two partial overlapping vehicles

Normally, the velocity values would be determined by the lidar sensor, based on the travel time of the lasers bouncing back to the receiver (2.1.3 Radar vs. Lidar). But again, to follow this method seems to be a tedious task.: calculating the distance with the complex method mentioned above for a single frame, repeat for the next frame, then determining the time delay Δt between 2 frames, finally with Equation 7:

$$v = \frac{d}{\Delta t}$$

Equation 7 Speed equation

where d is the distance, the speed v is calculated.

Additionally, it can also negatively impact the simulation, making it slower and more likely to crash, since an immense number of calculations are happening in the background. Granted, the codes used for this paper are not perfectly optimized, mostly due to the limited knowledge in coding in general, plus the short time frame given, but choosing to use a simpler method still can provide valuable results when approaching a problem. So instead, another proposed alternative is to take the velocities directly once again, from the CARLA client with function shown in Code snippet 2.

```
def get_speed(vehicle):
    vel=vehicle.get_velocity()
    return math.sqrt(vel.x**2 + vel.y**2 + vel.z**2) #m/s
```

Code snippet 2 Retrieval of vehicles speeds

`“vel = vehicle.get_velocity()”`: This line retrieves the velocity of the “vehicle” object. The `“get_velocity()”` method returns a vector that represents the vehicle's velocity in three-dimensional space. The velocity vector contains three components: “x”, “y”, and “z”, which represent the velocity along each axis.

`“vel.x**2 + vel.y**2 + vel.z**2”`: This part of the code calculates the square of the magnitude of the velocity vector. To find the magnitude, we need to calculate the sum of the squares of the individual components. The `“**”` operator represents exponentiation, so `“vel.x**2”` calculates the square of the “x” component of the velocity, `“vel.y**2”` calculates the square of the “y” component, and `“vel.z**2”` calculates the square of the “z” component.

`“math.sqrt(vel.x**2 + vel.y**2 + vel.z**2)”`: Finally, the `“math.sqrt()”` function from the Python `“math”` module is used to calculate the square root of the sum of the squared components. This gives us the magnitude of the velocity vector, which represents the speed of the vehicle.

The resulting value represents the speed of the vehicles in meters per second ($\frac{m}{s}$).

3.1.2.2 Function wise

Once the distance and velocities are known, the next step for the ego vehicle is to maintain an appropriate distance from the vehicle ahead to ensure both traffic efficiency and safety. This implies that the distance should neither be excessively long nor excessively short, striking a balance between the two factors. The key component lays in the term “Time headway” also known as just “Headway” or “Headway time”, any of these terms are used interchangeably within this paper. The time headway H [37] is determined by calculating the time difference between two vehicles reaching a specific location (as shown in Equation 8). Or in this case for ACC specifically, headway is the time it would be necessary for the ego vehicle to catch up to the target vehicle at set speed.

$$H = t_i - t_{i-1}$$

Equation 8 Headway equation

More specifically, the headway is calculated by comparing the time at which a vehicle “i” passes a particular location (t_i) with the time at which the preceding vehicle passes the same location (t_{i-1}). Different countries have varying regulations and guidelines regarding the legal or recommended safety distance between vehicles. In the United States, for example, several driver training programs [38] assert that maintaining a headway of less than 2 seconds behind another vehicle is considered unsafe.

To initiate the ACC system, a predefined set speed is required from the driver. Once the set speed is specified, the ego vehicle will try and maintain that set velocity until it detects a target vehicle ahead with the following conditions: as soon as the target vehicle lies within detecting range, the velocity of the target vehicle is smaller than the current set speed of the ego vehicle, and the desired distance is greater than the actual distance between the two vehicles, the controller will command the ego vehicle to decelerate. The ego vehicle will continue to decelerate until the desired distance, which is calculated with the Equation 9, is achieved.

$$\text{desired distance} = H * v_{ego}$$

Equation 9 Desired distance equation

Once the desired distance is reached, assuming that the target vehicle is moving at the same constant speed, our ego vehicle will also continue to keep the same speed. However, if the target vehicle starts to brake or decelerate, resulting in the actual distance between the two vehicles being smaller than

the desired distance, the ego vehicle will also decelerate. In situations where the maximum deceleration of $2,5 \frac{m}{s^2}$ is insufficient to prevent a potential collision, a warning is issued to alert the driver to manually apply the brakes. The choice of a maximum deceleration value of $2,5 \frac{m}{s^2}$ is based on comfort considerations, as studies have shown that the maximum comfortable acceleration/deceleration for drivers typically falls within the range of 2,0 to $2,5 \frac{m}{s^2}$, as highlighted in Van Arem's study [39].

When the target vehicle speeds up, and ego's speed at that moment is smaller than the set speed, then the acceleration of $2,5 \frac{m}{s^2}$ will be initiated and kept until the actual distance is equal to the desired distance ($v_{target} < v_{set_speed}$). Or until the actual speed of the ego vehicle is equal to the set speed ($v_{target} > v_{set_speed}$), in which case the actual distance can be equal or greater than the desired distance. To summarize, Table 2 is created.

Scenario	Action
Actual distance < desired distance	Deceleration, if insufficient, warning = issued
Actual distance > desired distance and actual speed < set speed	Acceleration
Actual distance $\geq$ desired distance and actual speed = set speed	Maintain speed

Table 2 Simplified scenarios and corresponding responses from ACC system

To ensure the comfort of the passengers, the maximum acceleration/deceleration of $2,5 \frac{m}{s^2}$ will not be overstepped. One way to easily achieve this is to use the "if" statement. If the acceleration value of the ego vehicle is greater than $2,5 \frac{m}{s^2}$, ACC will command the ego to release the gas, and apply a certain amount of brake, if needed. In case of deceleration, a warning is simply issued and a command for full/partial brakes is warranted, simulating the driver manually applying the brakes. Specifically, if the maximum deceleration of $2,5 \frac{m}{s^2}$ proves insufficient and the time to collision (TTC) (Equation 10) drops below 1 second, a warning is triggered to simulate a driver response. In the absence of actual drivers, the simulation executes an "if" statement to simulate the driver taking over and applying maximum braking force.

$$TTC = \frac{distance}{|v_{ego} - v_{target}|}$$

Equation 10 Time to Collision calculating

3.1.3 PID Controller design [40]

In this section, we will begin by presenting the generic implementation of a PID controller. Following that, we will delve into the specific application of the PID controller, tailored for an ACC system.

3.1.3.1 PID controller main principles

In this section, we will present a focused and concise pseudo-code version along with equations specifically related to the PID controller. We will avoid including other variables unrelated to the PID controller to maintain clarity and relevance. A detailed and comprehensive version will be provided in subsequent sections, covering all aspects when integrating the controller into the ACC system. The controller parameters consist of K_p , K_i , K_d [41]— the three controller gains. The following parameters are initially set to 0: the previous error (prev_error) term, integral term. Next up is the PID controller function to compute the control output. Starting out with the error term.

$$\text{error} = \text{target_value} - \text{current_value}$$

Equation 11 Error equation

The setpoint represents the desired value that we want our system to achieve, while the measured value corresponds to the actual value that is being observed or recorded from the system.

Output of the proportional term is calculated as follows:

$$P_{\text{out}} = K_p * e(t)$$

Equation 12 Output of the proportional term

where $e(t)$ is the error function with respect to time. Equation 12 can also be described in pseudo code:

$$P_{\text{out}} = K_p * \text{error}$$

Equation 13 Output of the proportional term in pseudo code

Output from the integral term:

$$I_{\text{out}} = K_i * \int_0^t e(\tau) * d\tau$$

Equation 14 Output of the integral term

where $e(\tau)$ is the error at the time τ , and the integral is taken from the start of the control action ($t = 0$) until the current time (t).

In pseudo code Equation 14 can be written as:

$$I_{\text{out}} = K_i * \text{integral}$$

Equation 15 Output of the integral term in pseudo code

In which the Integral term is:

$$\text{integral} = \text{integral} + \text{error} * dt$$

Equation 16 Integral term equation in pseudo code

Derivative term:

$$\text{derivative} = \frac{\text{error} - \text{prev_error}}{dt}$$

Equation 17 Derivative term equation (pseudo)

means the term depends on the rate of change of the error with respect to time.

Output of the derivative term:

$$D_{\text{out}} = K_d \frac{de(t)}{dt}$$

Equation 18 Output of the derivative term

In pseudo code this can be expressed as:

$$D_{\text{out}} = K_d * \text{derivative}$$

Equation 19 Output of the derivative term in pseudo code

Gathering all the outputs from all the terms above, the output without time lag (ideal PID) can be expressed as:

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

Equation 20 PID controller output

Equation 20 can also be written as:

$$PID_{output} = K_p * error + K_i * integral + K_d * derivative$$

Equation 21 Ouput of the PID controller in pseudo code

3.1.3.2 PID controller's integration with ACC system

Figure 19 shows a simple control system block diagram, each one of the blocks and variables is represented by a function in time or s domains. This block diagram and variables can be applied to any type of system, wherever it's hydraulic, electronic, thermal, pneumatic, etc.

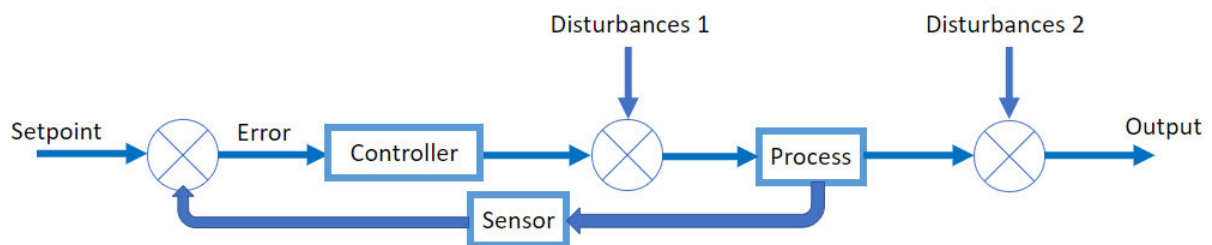


Figure 19 Control system block diagram [42]

Integrating this with an ACC system, we have the following:

- **Controlled Variable:** The controlled variable is the vehicle's speed.
- **Control Variables:** The control variables are the throttle and brake inputs to the vehicle.
- **Reference Level (Setpoint):** The reference level is the desired speed set by the driver or the ACC system's user.
- **Controlled System (Process):** The controlled system, or process, is the vehicle itself, including the engine, transmission, and brakes.
- **Controlled System Input:** The inputs to the controlled system are the throttle and brake signals, which regulate the vehicle's speed.

Below are the steps to translate and build a PID controller in Python.

First, a helper function called "get_speed" is defined. The goal of this helper function is to take a vehicle object as input, then retrieves and calculates its speed in meters per second ($\frac{m}{s}$) using its velocity components in three axes. Next, a class called "VehiclePIDController" is defined, using "init"³ method. The class "VehiclePIDController" represents the combination of two PID controllers, lateral

³ The "init" method, short for "initialization", is a special method in object-oriented programming languages like Python. It is also known as a constructor in some other programming languages. The purpose of the "init" method is to initialize the attributes or properties of an object when it is created from a class. In Python, the "init" method is defined within a class and is automatically called when you create a new instance (object) of that class using the class name followed by parentheses. It is always the first method that gets called when an object is created, allowing you to set initial values for the object's attributes or perform any other setup that needs to be done.

and longitudinal. Within it, a “run_step” function is defined, which when invoked, will execute one step of control, invoking both the lateral and longitudinal PID controllers to reach a target waypoint at a given target speed (input of the target value for the PID controller). The PID controller is configured to accept a speed input exclusively as the setpoint. In our dynamic system, when engaging with a target vehicle, the appropriate input speed to consider would be the target speed calculated as the ratio of “distance” to “desired_headway”.

Furthermore, we have no actual driver, so using the same PID controller to control the steering wheel is a good replacement, although not perfect. This provided class can be utilized to keep the ego vehicle on track, or to further test the limits of our ACC PID controller system. Specifically, the PID controller for lateral movement can serve as the driver's steering input, enabling us to assess the longitudinal control performance while driving through curves. This approach allows for a comprehensive evaluation of the system's capabilities in complex driving scenarios. Additionally, within the “VehiclePIDController” class a limit of 75% is set for both longitudinal and lateral control, meaning that the ACC system cannot exceed that maximum value (max_throttle=max_steering=0.75). The brake limit is set at 30%. But for the moment, our interest lies in longitudinal control, which is defined in the class “PIDLongitudinalControl”. Note that the class “VehiclePIDController” also uses the class “PIDLongitudinalControl”, alongside with another class “PIDLateralControl”, in which the longitudinal/lateral class is called ‘within the “VehiclePIDController” class. Explained here is only the longitudinal control since it's the focus of the ACC system. In addition to the code provided in the “PIDLongitudinalControl” class, there are crucial segments of code implemented within the “VehiclePIDController” class to achieve smooth acceleration. These code segments are responsible for fine-tuning the control behavior and ensuring a seamless acceleration experience for the vehicle.

The code checks whether the desired acceleration is positive or zero to determine if the vehicle needs to accelerate or maintain its current speed. For positive or zero acceleration, the throttle input is set to the minimum of the absolute value of the desired acceleration and the maximum brake value, ensuring the throttle is not excessively high. The brake input is set to zero in this scenario, deactivating the brakes. On the other hand, if the desired acceleration is negative (for deceleration or stopping), the throttle input is set to zero, and the brake input is adjusted to the minimum of the absolute value of the desired acceleration and the maximum brake value, ensuring smooth deceleration. Overall, this control logic ensures that the vehicle accelerates and decelerates smoothly, preventing abrupt changes in speed.

Using the “init” method, the “PIDLongitudinalControl” class is initialized. The constructor takes the following arguments as input (see Code snippet 3):

```
class PIDLongitudinalControl():
    def __init__(self, vehicle, K_P=1.0, K_I=0.0, K_D=0.0, dt =
0.03, error_window_size=10, cutoff_frequency=10):
```

Code snippet 3 Class “PIDLongitudinalControl” initial arguments

- “vehicle”: The vehicle object, whose speed will be controlled with the controller, in this case our ego vehicle.
- “K_P”, “K_I”, “K_D”: The gains parameter of the PID controller
- “dt”: The time interval at which the PID controller is executed [s].
- “error_window_size”: number of errors allowed to be stored, if exceeded, the oldest error will be removed.

- “cutoff_frequency”: implementing a low-pass filter to smooth out the control signal, by reducing high frequency noise or rapid changes in the error signal, leading to more stable and controlled responses [Hz].

The low-pass filter can be designed as follows (see Code snippet 4):

```
def apply_low_pass_filter(self, error):
    alpha = self.dt / (self.cutoff_frequency + self.dt)
    self.filtered_error = alpha * error + (1 - alpha) *
self.filtered_error
    return self.filtered_error
```

Code snippet 4 Low-pass filter in Python

The provided code snippet defines a function, “apply_low_pass_filter”, which applies a simple low-pass filter to an input error signal using exponential moving average principles. The filter's goal is to smooth out rapid fluctuations in the error signal while retaining responsiveness to changes. The process involves calculating a filtered error value by blending the current error value with the previous filtered error value, weighted by an “alpha” factor. This factor balances the contribution of the current and past values based on the time step (self.dt) and the cutoff frequency (self.cutoff_frequency). The resulting filtered error value is stored for subsequent iterations. This approach effectively reduces high-frequency noise in the error signal, enhancing stability in control systems.

The default values of these parameters can later be changed when the function is called, otherwise if no arguments were given, the controller will use these default values. In Python, when defining a class, instance variables (attributes) and methods can be created, and these are associated with the instances of that class. The “self” keyword is used to differentiate instance variables from local variables within methods. By using “self”, variables become accessible throughout the class as instance variables, and they can be accessed and modified using the “self.variable_name” syntax. self.errorBuffer is used to store a history of errors for the controller (for the usage of D and I elements). As new errors are generated during the control process, they are added to the right end of the “deque”. If the length of the deque exceeds 10 (the specified “maxlen”), the oldest error at the left end will be removed automatically, ensuring that the buffer contains only the most recent 10 errors. self.integral = 0.0, self.prev_error = 0.0, self.last_output = 0.0 means that these parameters are initialized with the value of 0. Initializing it to 0.0 in the constructor guarantees that it is available and avoids potential runtime errors in the code that uses this attribute.

Following that, a “run_step” function as shown in Code snippet 5 is defined:

```
def run_step(self, target_speed):
    current_speed=get_speed(self.vehicle)
    return self.pid_controller(target_speed, current_speed)
```

Code snippet 5 Run_step function

This function takes in two arguments, itself, and target_speed – the desired speed ego vehicle should reach. The command current_speed=get_speed(self.vehicle) uses the helper function earlier to retrieve ego vehicle’s current speed. The last line returns the result of the pid_controller() method, passing the target_speed and current_speed as arguments. Subsequently pid_controller() is defined as follows (see Code snippet 6):

```
def pid_controller(self, target_speed, current_speed):
    error = target_speed - current_speed
    filtered_error = self.apply_low_pass_filter(error)
    self.error_buffer.append(filtered_error)
    self.error_sum += filtered_error * self.dt
    de = (filtered_error - self.prev_error) / self.dt
    if len(self.error_buffer) > self.error_window_size:
        self.error_sum -= self.error_buffer.pop(0) * self.dt

    output = self.K_P * error + self.K_I * self.error_sum +
self.K_D * de
    self.prev_error = filtered_error
    return np.clip(output, -1.0, 1.0)
```

Code snippet 6 Building the PID controller in Python

The core process begins with the calculation of the speed error, achieved by subtracting the current speed from the target speed. This error value is then subjected to a low-pass filtering technique, which serves to smooth out any high-frequency fluctuations or noise in the signal. The filtered error is subsequently integrated over time, aiding in addressing steady-state errors through the accumulation of historical errors. Furthermore, the derivative of the filtered error is determined to capture the rate of change of the error signal. The calculated proportional, integral, and derivative terms are then combined to compute the PID control output (transformed from Equation 21). This output acts as the correctional signal that guides the system towards the desired target speed. However, to prevent abrupt and erratic control actions, the output is bounded within a predefined range using the `np.clip`⁴ function. Integral windup, a potential issue where the integral term leads to overshooting or oscillations, is mitigated by maintaining an error buffer of limited size. Old error values are pruned from the buffer to ensure that the integral term remains meaningful and effective. Finally, the controller retains the previous filtered error for use in calculating the derivative term during the next iteration. By orchestrating these intricate processes, the PID controller orchestrates a nuanced and balanced approach to managing the system's speed. The outcome is a control output that effectively minimizes the deviation between the desired target speed and the actual speed of the system. This culminates in a more accurate and stable regulation of the system's behavior, aligning it with the desired performance objectives.

⁴ Is a function from NumPy library, used to ensure that the control output generated by the PID controller remains within a certain range (-1,1). If output is less than -1.0, it will be set to -1.0. If output is greater than 1.0, it will be set to 1.0, any value between (-1,1) stays the same.

3.1.3.3 PID controller response types

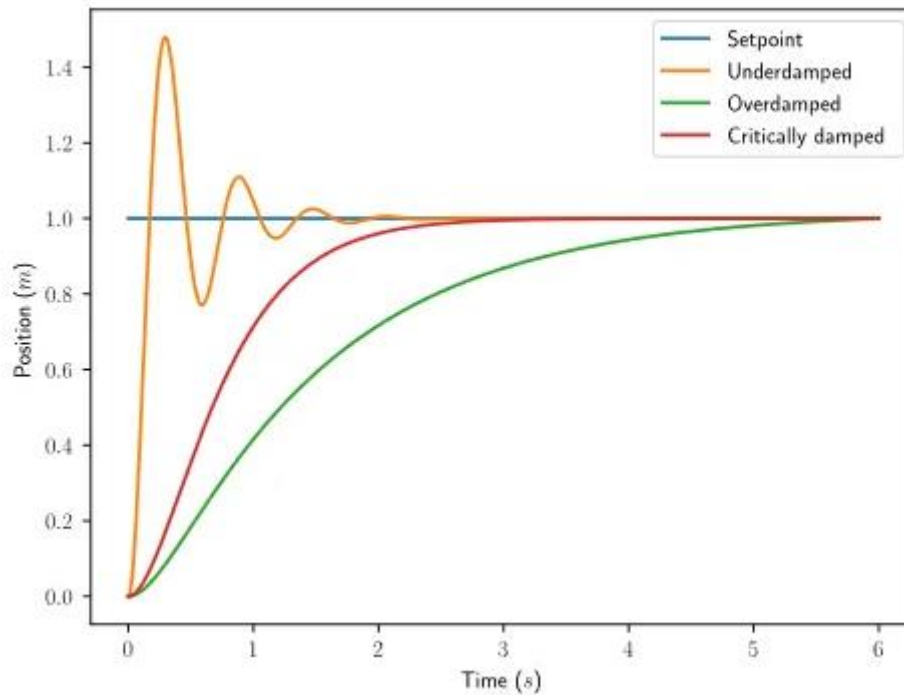


Figure 20 PID controller response types [43]

As seen in Figure 20, there are three main types of response, underdamped, overdamped, and critically damped. When the system exhibits an underdamped response, it initially oscillates around the reference before eventually settling down. This means that it takes some time for the system to reach the desired reference value, and during this process, there are oscillations or fluctuations observed. On the other hand, in an overdamped response, the system rises very slowly without overshooting the reference. It does not exhibit any oscillations, but it may take longer to reach the desired reference value compared to other response types. A critically damped response represents the fastest rise time among the three response types, without any overshooting of the reference. This means that the system quickly approaches the desired reference value without oscillating around it. Our objective is to fine-tune the controller to attain this specific type of response. Nevertheless, a quicker rise time and a slight degree of overshoot could potentially yield the optimal outcome. The determination of the optimal response, including whether to incorporate these factors, will be made following the execution of the simulations.

3.2 Testing methodology

3.2.1 Test setup

3.2.1.1 Hardware

Since CARLA being a sophisticated and realistic simulator for autonomous driving research, it places considerable demands on hardware resources, requiring substantial processing power, RAM, and a capable graphics card to run smoothly and deliver accurate results. However, if decent hardware is not readily available, there are several options to consider. First, reducing the graphics settings in CARLA can alleviate some strain on the GPU and allow it to run on lower-end systems. CARLA has an option called Low mode for this purpose [44]. Another approach is to scale down the simulation complexity, such as reducing the number of dynamic objects or decreasing the simulation area, while

still preserving essential aspects of the study. Which means when choosing the map, the size does matter. Additionally, layered maps can be used, where different layers can be toggle on/off [45].

The computer used for this paper is however considerably powerful. The specifications are as follows:

- GPU (most important): NVIDIA RTX 3070 Ti, 16GB VRAM,
- CPU: AMD Ryzen 5 5500
- RAM: 16Gb, DDR4
- Operating System: Windows 11 Pro 64-bit (10.0 Build 22621)

3.2.1.2 Software, libraries, and dependencies

Software needed are Python 3.7, CARLA, Meshlab, installed with the instruction found on their official website [46] [47] [48]. Dependencies and libraries in Python are essential components that extend the functionality of the language by providing pre-written code and additional functionalities. Dependencies refer to external modules or packages that a Python program relies on to perform specific tasks or operations. These dependencies are typically listed in a file, such as "requirements.txt" or "setup.py," and can be installed using package managers like "pip"⁵ [49]. Libraries, on the other hand, are collections of code and functions that serve a particular purpose and can be imported into Python programs. These dependencies and libraries are immensely useful for several reasons. Firstly, they save time and effort for developers as they don't need to reinvent the wheel for common tasks, but rather leverage the work done by others. Secondly, they enhance the capabilities of Python, allowing programmers to perform a wide array of tasks ranging from scientific calculations and data analysis to web development and machine learning. Additionally, dependencies and libraries ensure code reliability, as they are often developed, tested, and maintained by large communities, making them less prone to errors and security vulnerabilities.

In this section, we will outline the essential libraries and dependencies required for the simulated tests. Additionally, we'll provide a brief and concise explanation of each. Among these modules, some are considered "universal dependencies," meaning they must be present in every script. Therefore, if a script mentions the use of universal dependencies, it implies that all the dependencies listed below will be utilized:

1. "glob": The glob module is used for file and directory name pattern matching. It enables searching for files that match a specific pattern using wildcard characters such as "*" and "?". This module is particularly useful when you need to work with multiple files that follow a certain naming pattern in a directory.
2. "os": The os module provides a way to interact with the operating system, allowing you to perform various OS-related tasks. It offers functions to manipulate files and directories, access environment variables, and execute system commands. With os, you can create, delete, rename, and move files and directories, among other functionalities.
3. "sys": The sys module provides access to some variables and functions that are specific to the Python interpreter and the system it is running on. It allows you to interact with the Python runtime environment, access command-line arguments passed to a script, and control the Python interpreter itself. The sys module is commonly used when you need to handle command-line arguments or modify the default behavior of the interpreter.
4. "argparse": The argparse module in Python is used for parsing command-line arguments and options. It provides an easy and efficient way to create user-friendly command-line interfaces for your scripts or applications. With argparse, you can define the arguments your script

⁵ Packet installer for Python, providing a simple method of library installation.

expects, and it automatically generates help messages and handles parsing of command-line arguments according to the defined rules.

5. “time”: The time module in Python provides various functions that allow you to work with time-related tasks and measure the performance of code execution, is focused on dealing with time as a simple numeric value in seconds.
6. “datetime”: provides higher-level abstractions to work with both dates and times, allowing for more complex date and time operations.
7. “random”: provides functions for generating random numbers and random selections.
8. “numpy”: is a powerful third-party library for numerical computing in Python. It stands for "Numerical Python." NumPy provides support for large, multi-dimensional arrays and matrices, along with a vast collection of high-level mathematical functions to operate on these arrays efficiently.
9. “matplotlib”: for creating static, interactive, and animated visualizations. It provides a simple and flexible way to generate various types of plots, charts, and graphs, making it an essential tool for data visualization and analysis in Python.
10. “carla”: provides a realistic and flexible environment to simulate and evaluate various autonomous driving algorithms and systems.

Other case specific dependencies/libraries

1. “open3d”: is an open-source library for 3D data processing and visualization. It provides a set of tools and functionalities to work with 3D point clouds, meshes, and other geometric data.

3.2.2 Output data retrieval method

3.2.2.1 Diagram

To facilitate easier visualization, we utilize the “Matplotlib” library to generate diagram outputs for each scenario. To begin, we initialize empty lists to store data that will be used in the plots. Once a value becomes available, we employ the `.append()` function to add this value to the aforementioned empty list. This process is repeated for each new data point obtained, continuously updating the list with the relevant information for generating the plot later. After populating the lists with the required values, we utilize the `plt.plot()` command to open a new window displaying the diagram. As the data in the lists is continuously updated, the plot in the window will also update in real-time, providing a dynamic visualization of the changing data.

3.2.2.2 Vehicle telemetry

The CARLA specific command `vehicle.show_debug_telemetry(True)` activates the display of debug telemetry information for a specific vehicle. This information includes real-time data about the vehicle's dynamics, control inputs, and sensor readings. Enabling debugging telemetry can be useful for monitoring and analyzing the vehicle's behavior and interactions within the simulation environment, aiding in debugging, testing, and performance evaluation of control algorithms and systems. Precisely, this data provides insights into the vehicle's speed, steering, throttle and braking inputs, current gear, drag, RPM, load on each tire, friction, and slip angle, along with individual RPM values. For this simulation, the throttle and brake inputs, along with the ego vehicle's current gear, are of particular significance. This information greatly facilitates troubleshooting and comprehension of the behavior exhibited in the subsequent acceleration and speed curves.

3.2.3 Assessment criteria

When assessing the performance of an ACC system within a simulated environment, several key criteria come into focus. The evaluation encompasses factors such as the system's ability to consistently follow a lead vehicle while maintaining the desired distance, its capacity to smoothly

transition between acceleration and deceleration. Additionally, the ACC system's responses to the lead vehicle's behavior, its adaptability to varying traffic conditions play integral roles in gauging its effectiveness. The assessment also delves into parameters related to stability and safety margins, all of which contribute to an ACC system's ability to ensure secure, comfortable, and efficient driving experiences. For the purpose of assigning scores to the respective aspects, a table is formulated (see Table 3). The scoring scale ranges from 1 to 10, with 1 denoting a complete failure and 10 indicating a flawless success.

Aspect	Score
Steady following state	
Desired distance/Headway keeping	
Overshoot	
Oscillation	
Response to target vehicle	
Smoothness in braking and throttle control	

Table 3 Criteria for assessing the ACC system

3.2.4 Pre-simulation setup

3.2.4.1 Lidar setup

The purpose of this evaluation is to assess whether the distance value obtained from the LIDAR sensor significantly differs from the distance obtained from the CARLA client. If the deviation falls within an acceptable tolerance range, the distance value utilized for the PID controller will be the one obtained directly from the CARLA client, ensuring a consistent feed of data.

The lidar sensor will be placed on top of our ego vehicle, as shown in Figure 21. The shape of the ego vehicle can vaguely be recognized, and the black area in the middle indicates the position of the lidar sensor. Which is logical since the lidar sensor can't scan itself although numerous yellow dots (the further the distance the more the color will differ from yellow) can be seen. These dots are generated by the lidar's cast rays intersecting with objects, primarily with the ground in this particular scenario in Figure 21.

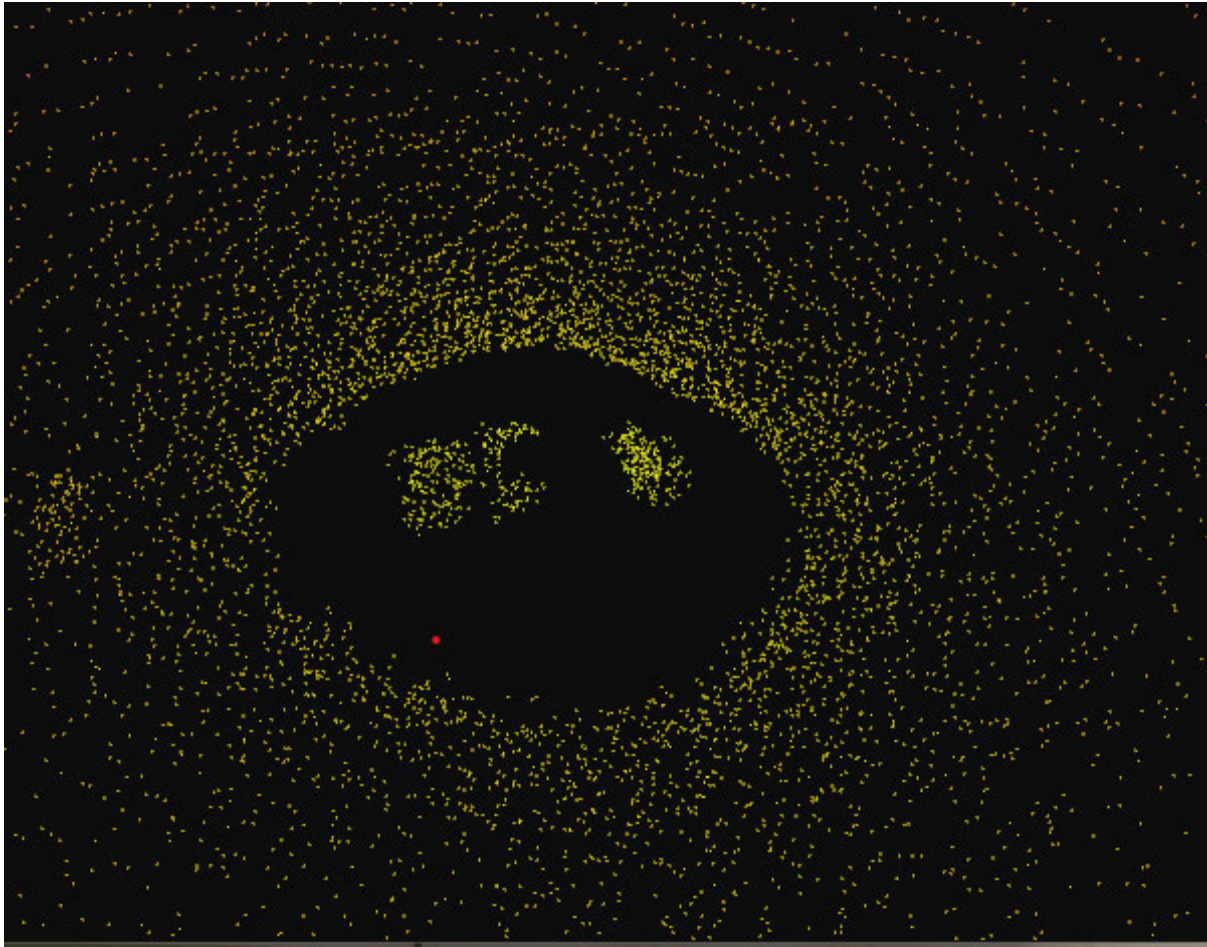


Figure 21 Lidar sensor placement

The evaluation method employed here is a stationary test, where both vehicles are positioned at a predetermined distance from each other. This ensures the lidar sensor will operate at maximum accuracy, since in dynamic situations its performance will surely worsen. The reference distance can be directly obtained from the CARLA client, representing the “true” distance. The comparison involves measuring the distance from the output file in Meshlab and then assessing whether the deviation between the measured distance and the referenced distance is sufficiently insignificant to proceed with further testing. Before placing a lidar sensor on our ego vehicle in CARLA, we must first spawn the ego vehicle inside the map. To achieve this, we need to establish a connection to the CARLA client. Once connected, we import the necessary dependencies, including the universal dependencies plus the “open3d” library. This step is crucial to ensure that the ego vehicle is properly integrated into the CARLA simulation environment, allowing us to proceed with adding the lidar sensor and performing the evaluation process. In the following steps, we will create a “plasma” color map. Based on this color map, the points that are farther away from the lidar sensor will have different colors compared to those that are closer (see Figure 22). The “plasma” colormap will be used to represent the intensity of each point in the point cloud, providing a visual distinction between points at varying distances from the lidar sensor.

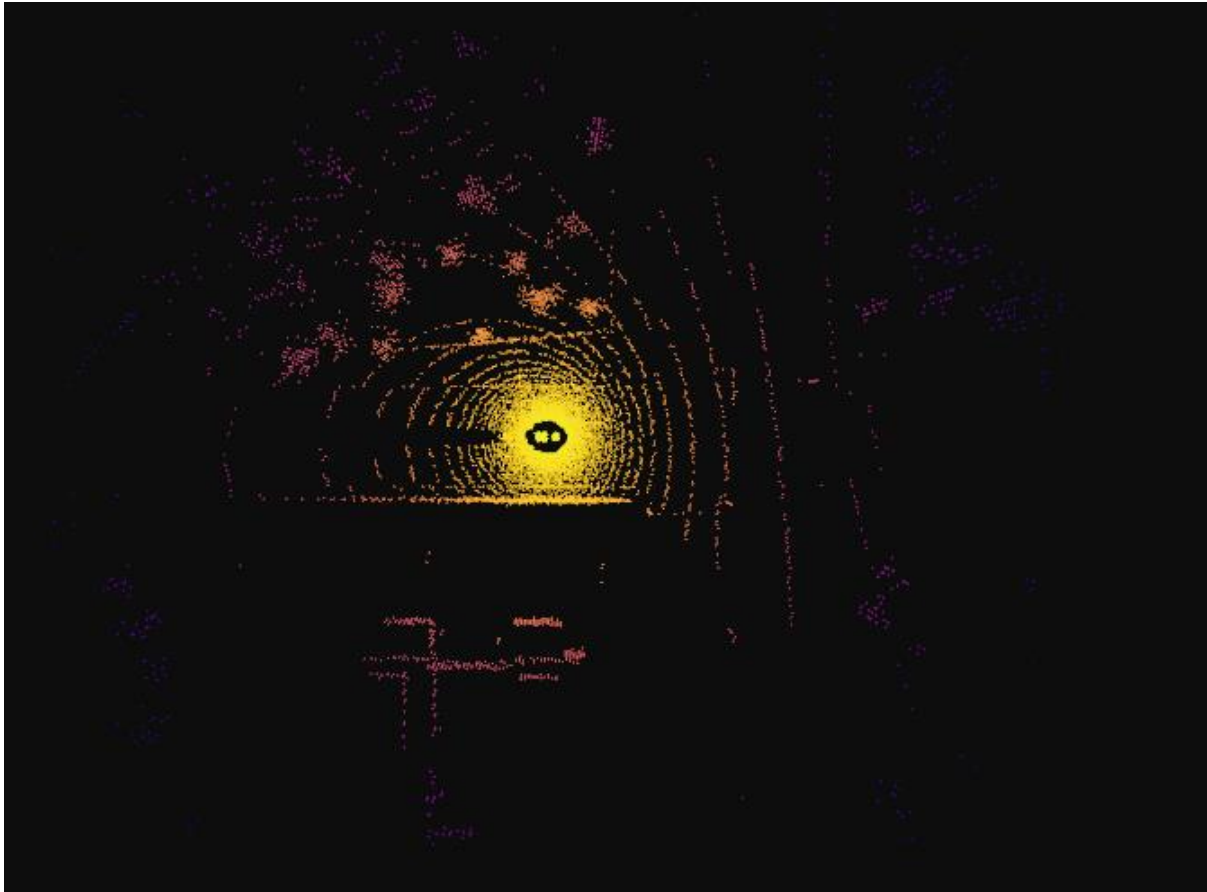


Figure 22 Color intensity indicating different distances

Afterwards, the `lidar_callback` function is defined, is necessary for the `listen()`⁶ method [50]. The `lidar_callback` function takes a point cloud and prepares it with intensity-based colors for visualization in Open3D. It extracts the intensity values from the point cloud, computes corresponding RGB colors based on a “plasma” colormap, and assigns these colors to each point in the cloud. The 3D data points are adjusted for correct visualization in Open3D's right-handed coordinate system, which is different from CARLA, therefore, the point's coordinates must be converted. In Figure 22, the points further away appear in a purple color, while the inner points are yellow. Additionally, the function includes commented-out code that demonstrates an example of converting the points from the sensor's coordinate space to the vehicle's coordinate space using a `carla.Transform`⁷ variable named “`tran`”. However, this part is currently commented out and not affecting the visualization. This becomes essential if we opt for the more intricate approach of calculating real-time intersecting points mentioned in 3.1.2.1 Environmental detection.

The main function is defined starting with the client, then the world, in which the “actors” [51] will be placed inside of. Now we need to have the blueprint for the lidar sensor. The `generate_lidar_bp` function means that the lidar blueprint is taken from the CARLA library, with the name `sensor.lidar.ray_cast`. The function sets various attributes such as field of view, number of channels, range, rotation frequency, and points per second based on the given arguments. It also

⁶ Every sensor has a `listen()` method. This is called every time the sensor retrieves data. The argument `callback` is a lambda function. It describes what should the sensor do when data is retrieved.

⁷ a combination of location and rotation, without scaling. Location (`carla.Location`): describes a point in the coordinate system. Rotation (`carla.Rotation` - degrees (pitch, yaw, roll)): describes a rotation for an object according to Unreal Engine's axis system.

handles the configuration for noise in the lidar sensor if required. Some of the default key properties are:

- upper FOV=15°
- lower FOV=-25°
- channel count=64
- max range=100m
- points per second=500000

For our ego vehicle, the blueprint will be taken from CARLA library (vehicle model = tesla model 3). It will be spawned in randomly with the “random” module, but with the command `random.seed(1000)`, every time we run the script after the first, the spawn position of the ego vehicle will stay the same. As for the target vehicle, it will be spawned within 20m in front of the ego vehicle (vehicle model = dodge charger police). Both vehicles are stationary. Next step will be to spawn and place the lidar on top of the ego vehicle. The proper placement of the lidar sensor on top of the ego vehicle is crucial for obtaining clear images. Setting the height too low could result in the lidar being positioned inside the vehicle, obstructing the emitted rays, and leading to distorted and confusing data. Moreover, this height adjustment proves beneficial when testing the lidar on various ego vehicles, as each vehicle has a different height that must be accommodated. `attach_to=vehicle` is a function that ensures, as the vehicle moves and rotates, the lidar sensor will move and rotate with it, always maintaining its relative position to the vehicle's origin.

3.2.4.2 Lidar evaluating

Subsequently, when running the script (CARLA client must be ran first), utilizing the `listen()` method, a number of .ply files will be saved to the hard drive, then exported to Meshlab, where they can be further analyzed. Upon opening one of these .ply file, a 3D point cloud map is presented (see Figure 23).

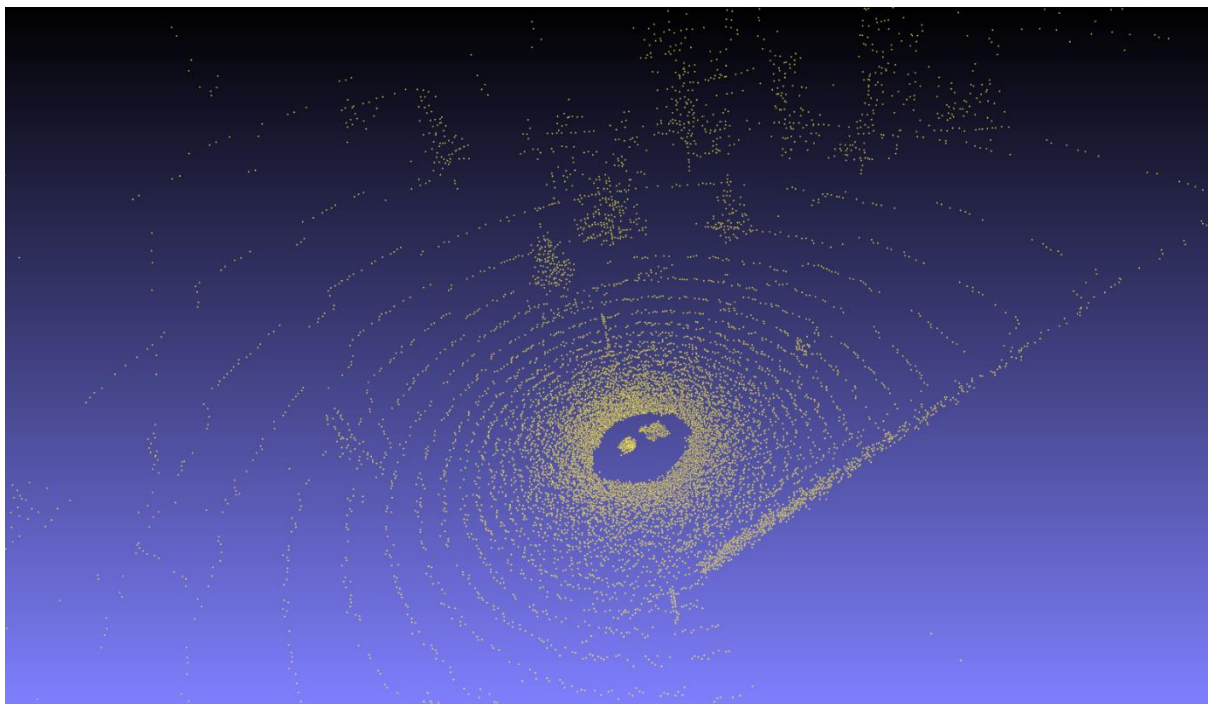


Figure 23 3D point cloud in Meshlab

This snapshot captures a specific moment in time, showing all the points formed by obstructions of the rays emitted from the lidar sensor. In the visual representation, our ego vehicle is located at the

center, surrounded by rings of yellow dots. By adjusting the settings in Figure 24 (specifically, the “Dot Decorator” and minimizing the point size on the scale), we can enhance the clarity of the image. Notably, there is a distinct cluster of yellow dots, followed by an empty space ahead of the ego vehicle. This vacant region indicates the presence of a target vehicle, where the rays are blocked, causing the clustering of dots.

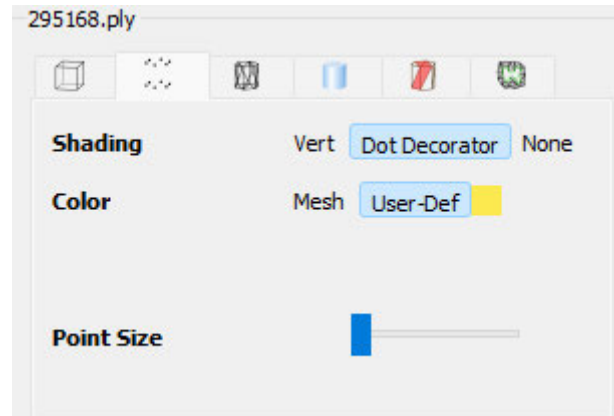


Figure 24 Settings to clarify the 3D point cloud

Figure 25 was created to facilitate easier observation. The area demarcated by red cross lines signifies the space behind the target vehicle (including the vehicle itself), as observed from the lidar sensor's origin (depicted as the big red dot). Just preceding this area lies the back of the target vehicle, represented by the cluster of dots, primarily capturing the bumper and a small portion of the rear windshield. The two straight red lines represent the rays emanating from the lidar sensor, which, in actuality, extend throughout a complete 360-degree circumference around the ego vehicle.

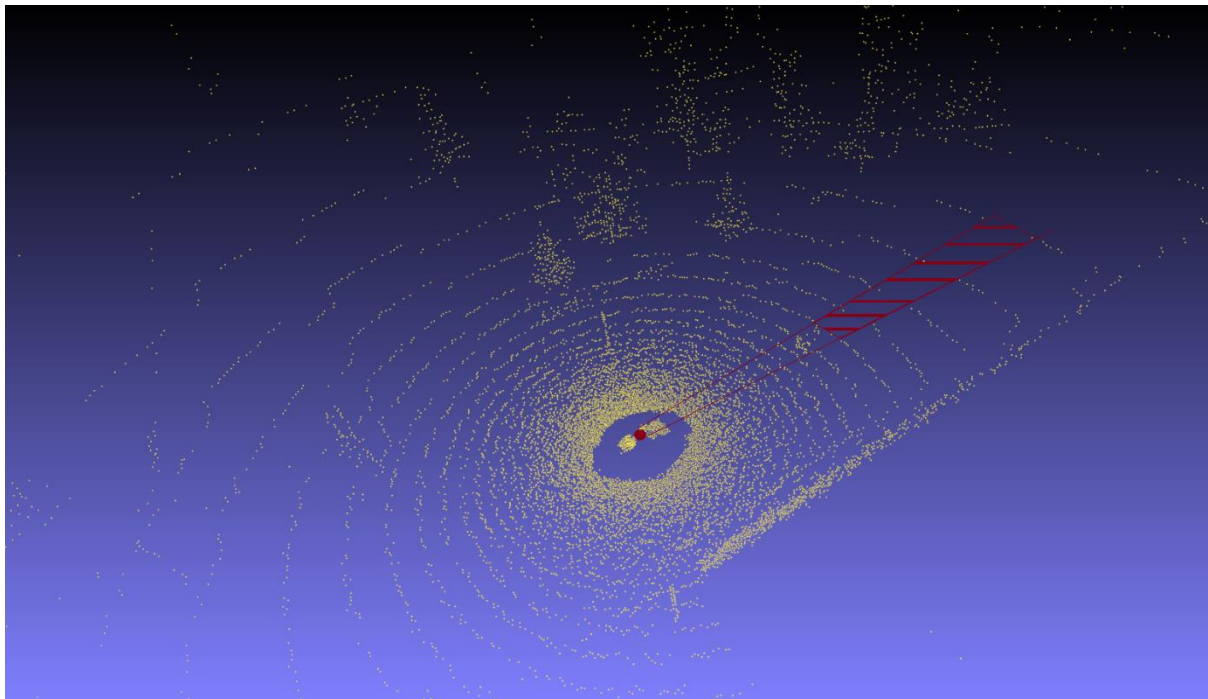


Figure 25 Identifying target vehicle

Following we have the function “Measuring tool” (Figure 26). With this function the distance between the two vehicles will be manually determined.

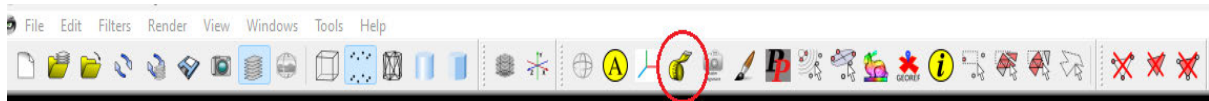


Figure 26 Measuring tool in Meshlab

There will be four distances measured in total, to ensure the accuracy of the measurement, from there a mean value will be calculated. In total four points from the ego vehicle, plus four points from the target vehicle will be taken in measurement. The points in consideration are typically near the edge of each vehicle, meaning from the front bumper of our ego vehicle to the back bumper of the target vehicle. What's also important is that after having a mean value, we must subtract approximately one car-length from it since the distance retrieved from the client is measured from middle point of both vehicles, as mentioned in (3.1.2.1 Environmental detection). Reference distance retrieval can be done with the Code snippet 1 (using Euclidean distance equation 3-dimensional). After letting the simulation running for at least five to ten seconds (to ensure stability since the client spawns vehicles by dropping them onto the road). The distance can vary despite the spawn command explicitly stating a set distance (although extremely insignificant), the result can be seen in the windows command prompt (Figure 27):

```
[2023-07-31 20:40:30]Distance = 19.999738287469153 m
```

Figure 27 Reference distance from the CARLA client in command prompt

Subtracting one car-length distance, which is 5 meters (typical light vehicle length) and adding 0.3m (spawn position of the lidar sensor relative to vehicle's origin, in x-axis) we have the true reference distance:

$$\text{reference distance} = 19,9997 - 5 + 0,5 = 15,4997 \text{ (m)}$$

Equation 22 True reference distance

The argument regarding potential errors is valid as we are using two different vehicles, which means that a car-length distance may not precisely correspond to 5 meters, technically should be half the length of ego plus half the length of target vehicle. However, these minor discrepancies will be disregarded as they are insignificant for the purposes of our analysis. All four measurements will be done in four different perspectives (see Figure 28, Figure 29, Figure 30, Figure 31)

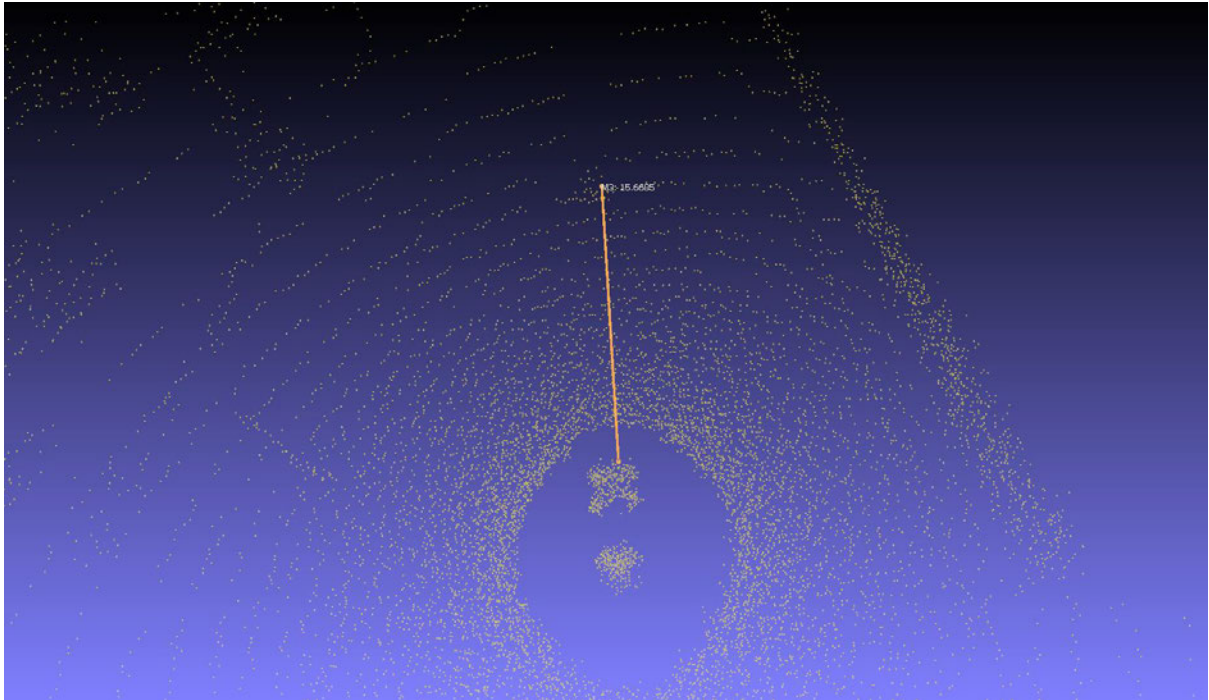


Figure 28 Measurement #1

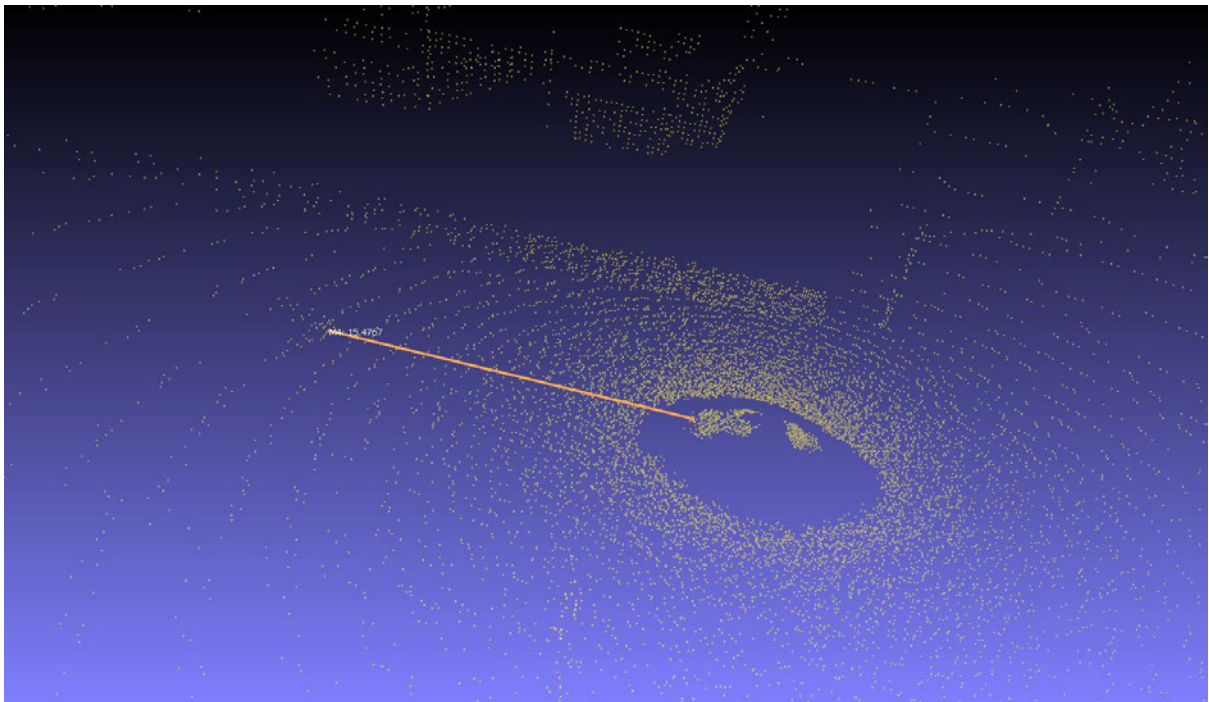


Figure 29 Measurement #2

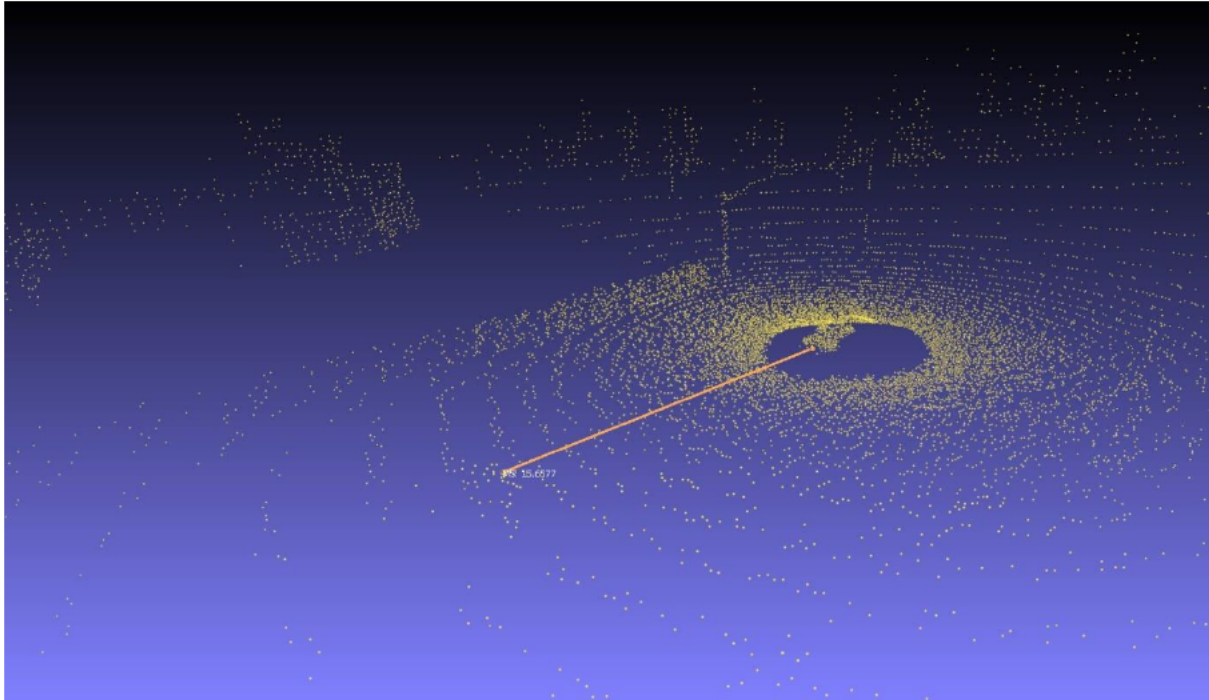


Figure 30 Measurement #3

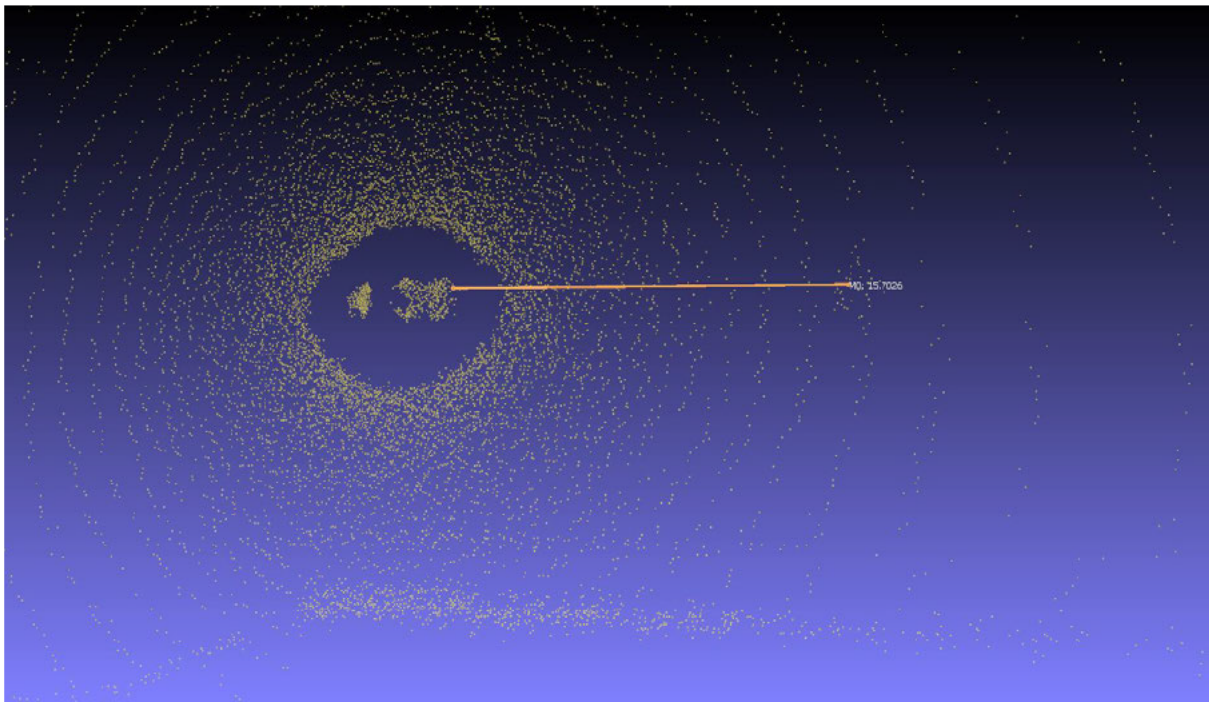


Figure 31 Measurement #4

Input the measurements in Table 4 and calculate the mean value:

	Measurement #1	Measurement #2	Measurement #3	Measurement #4
Distance (m)	15,6686	15,4767	15,6577	15,7026
Mean value (m)	15,6264			

Table 4 Distance measurements in Meshlab and mean value

Comparing with the true reference distance (Equation 22), we have Equation 23:

$$x = \frac{15,6264 - 15,4997}{15,4997} * 100\% = 0,8174\%$$

Equation 23 Tolerance range vs. true distance

where x is the error percentage. Keep in mind that the 0,8174% error rate could also be influenced by the method used to select points. Since the points are chosen manually by visual inspection, potentially it could result in an overestimation of the distance due to their diagonal arrangement. Consequently, the actual error rate might be even lower than reported. All in all, considering an error percentage of less than 1%, we can confidently state that it falls within an acceptable range. From this point onwards, we will use the distance directly obtained from the client as data for the controller.

3.2.4.3 PID controller pre-tuning setup

The first step in tuning a PID controller is using a simple scenario, in which the objective is for the ego vehicle to smoothly reach the desired speed set by the driver. The PID controller then regulates the throttle or brakes of the ego vehicle to reach and maintain that set speed. By closely observing the system's output, the tuning process can begin, allowing adjustments to the PID controller's parameters to achieve the desired control performance. Starting off by running the same script used in section 3.2.4.1 Lidar, with some modifications. First is removing the target vehicle since we don't need our ego to follow any other vehicle. Second is removing the output of the .ply files as we don't need to analyze the lidar distance anymore. Third is adding a command specifically in CARLA to retrieve the ego vehicle's speed in real-time: `vehicle.get_velocity().length()`⁸. Note that the ego vehicle's initial speed is still $0 \frac{m}{s}$.

When using the "VehiclePIDControl" class, a "waypoint" will be needed as argument. To accomplish this task, a few CARLA commands were utilized, one to retrieve a list of waypoints in the CARLA map, one to specify the vicinity of the waypoints, and last is to randomly pick a waypoint in the filtered list retrieved. Then selected waypoints are added to a waypoint list. Finally, the code iterates over each waypoint in said list and draws a "0" in the client, so that it's easier to visualize. The main function of these waypoints is to provide crucial information about the path the vehicle should follow, in the sense that it has reference points to steer towards, which helps in determining the appropriate control actions based on the vehicle's current state.

To run the "VehiclePIDControl" class, Code snippet 7 is examined:

```
control_vehicle = VehiclePIDController(vehicle, args_lateral={'K_P':1.0,
'K_D':0.0, 'K_I': 0.0}, args_longitudinal={'K_P': 1.0, 'K_D': 0.0,
'K_I': 0.0, 'K_l': 0.0})
control_signal = control_vehicle.run_step(ego_target_speed, waypoint
vehicle.apply_control(control_signal))
```

Code snippet 7 Applying the PID controller to the vehicle

The first line creates an instance of the "VehiclePIDController" class and assigns it to the variable `control_vehicle`. The instance takes the vehicle object (ego vehicle) as an input, which represents the controlled system. It also takes two dictionaries as inputs, which specify the parameters for the lateral and longitudinal control. The longitudinal and lateral parameters are K_P , K_D , and K_I , which are the proportional, derivative, and integral gains, respectively. Currently, the lateral

⁸ is essentially the Euclidean calculation of the three velocity vectors in three axis x, y, z (similar to Code snippet 2), compacted into a single command.

parameters have been configured to a value of 0, reflecting the fact that lateral movement is not currently within our scope of consideration.

Then the `run_step` method of the `VehiclePIDController` instance is called and the result is assigned to the variable `control_signal`. The `run_step` method takes two inputs: `ego_target_speed` and `waypoint`, which are the desired values for the longitudinal and lateral control, respectively. The method computes the control signal for the vehicle based on these inputs and the current state of the vehicle. The control signal is a tuple of (throttle, brake, steer), which are the outputs of the PID controller. Finally, the `apply_control` method of the vehicle object is called and the `control_signal` is passed as an input. The `apply_control` method changes the speed and direction of the vehicle according to the `control_signal`. Note that the gains values shown are the default value and will be tweaked in the section below. At last, the ACC system is still a system focused on comfort, which means a maximum value of absolute acceleration of $2,5 \frac{m}{s^2}$ is not to be exceeded. An “if” statement is used to assure that whenever the acceleration goes over the limit, the ACC system will take action accordingly.

3.2.4.4 PID controller tuning

3.2.4.4.1 Ziegler-Nichols method

In our approach to tuning the PID controller in CARLA, we will employ the well-known Ziegler-Nichols method, a widely used heuristic technique for optimizing PID controller parameters. The primary objective is to find suitable proportional (K_p), integral (K_i), and derivative (K_d) gains that yield stable and responsive control for the ego vehicle. The Ziegler-Nichols method involves systematically adjusting the gains based on the system's response to step inputs. We'll start by increasing the proportional gain K_p from 0 until the system exhibits sustained oscillations (i.e., reaches the ultimate gain K_u). We'll then note the critical oscillation period (T_u). From these observations, we'll calculate the ultimate gain (K_u) and use it to determine the appropriate gains for our PID controller according to Table 5:

Control Type	K_p	T_i	T_d	K_i	K_d
P	$0.5K_u$	–	–	–	–
PI	$0.45K_u$	$0.83T_u$	–	$0.54K_u/T_u$	–
PD	$0.8K_u$	–	$0.125T_u$	–	$0.10K_uT_u$
classic PID^[2]	$0.6K_u$	$0.5T_u$	$0.125T_u$	$1.2K_u/T_u$	$0.075K_uT_u$
Pessen Integral Rule^[2]	$0.7K_u$	$0.4T_u$	$0.15T_u$	$1.75K_u/T_u$	$0.105K_uT_u$
some overshoot^[2]	$0.33K_u$	$0.50T_u$	$0.33T_u$	$0.66K_u/T_u$	$0.11K_uT_u$
no overshoot^[2]	$0.20K_u$	$0.50T_u$	$0.33T_u$	$0.40K_u/T_u$	$0.066K_uT_u$

Table 5 Ziegler–Nichols tuning method table [52]

An example can be seen in Figure 32, showing how to measure the ultimate period T_u and some other guidelines regarding this tuning method.

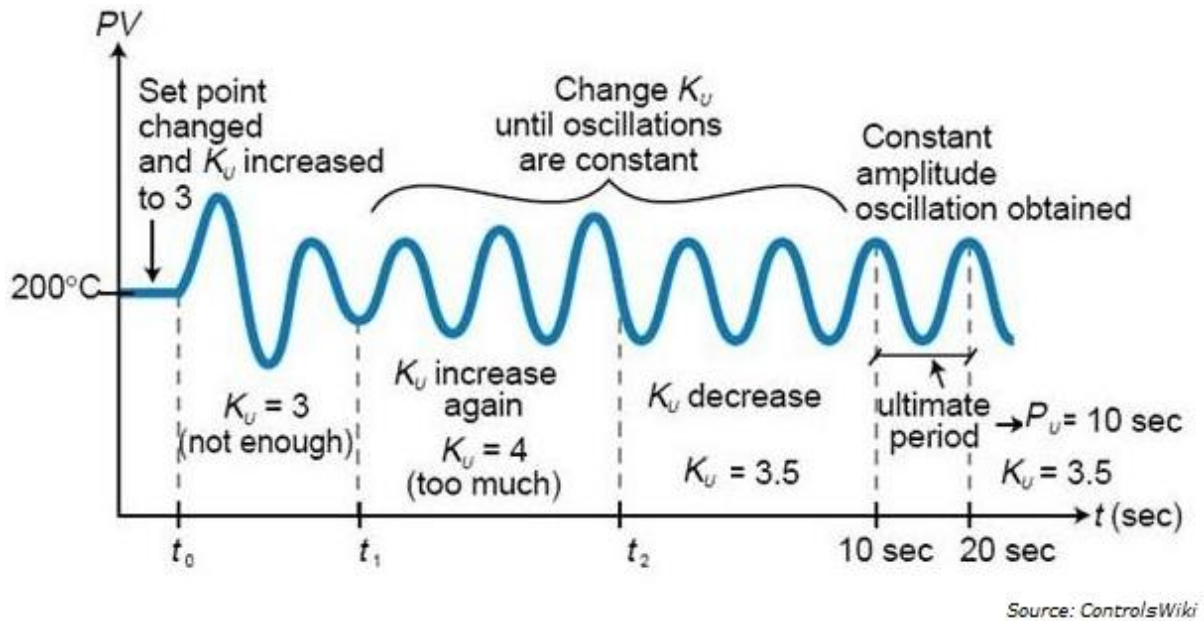


Figure 32 Example for the Ziegler-Nichols method [53]

Choosing the classic PID, we have the following equations:

$$K_p = 0,6 * K_u$$

$$K_i = 1,2 * \frac{K_u}{T_u}$$

$$K_d = 0,075 * K_u * T_u$$

Equation 24 Equations for determining PID gains according to Ziegler–Nichols tuning method

This tuning method, as depicted by the “Classic PID” equations in the provided Table 5, leads to a response known as “quarter wave decay”⁹. While this response is suitable for certain applications, it may not be optimal for all scenarios. The primary objective of this tuning rule is to provide PID loops with excellent disturbance rejection capabilities. However, it results in an aggressive gain setting, which can lead to overshooting. Some applications prioritize minimizing or entirely eliminating overshoot, making the Ziegler-Nichols method unsuitable for those cases. It's essential to avoid excessive gains that could lead to overshooting, instability, or rapid control oscillations, as well as to consider the specific dynamics of the CARLA simulation environment when applying these tuning parameters. Should the classic PID tuning result in excessive overshoot, we will switch to the no overshoot tuning option and assess whether this adjustment improves our system's response. While this approach targets overshoot issues and strives for a controlled and stable system behavior, it's essential to acknowledge that it may introduce the drawback of slower rise times.

3.2.4.4.2 Executing the tuning simulation

Starting the simulation with the $K_p = 1, K_i = K_d = 0$, along with a set speed of $10 \frac{m}{s}$ ($36 \frac{km}{h}$), we have the following system response:

⁹ Response in a loop is tuned so the oscillation decays with each wave being one quarter of the previous wave, it produces satisfactory, if not optimum, set point response and disturbance rejection.

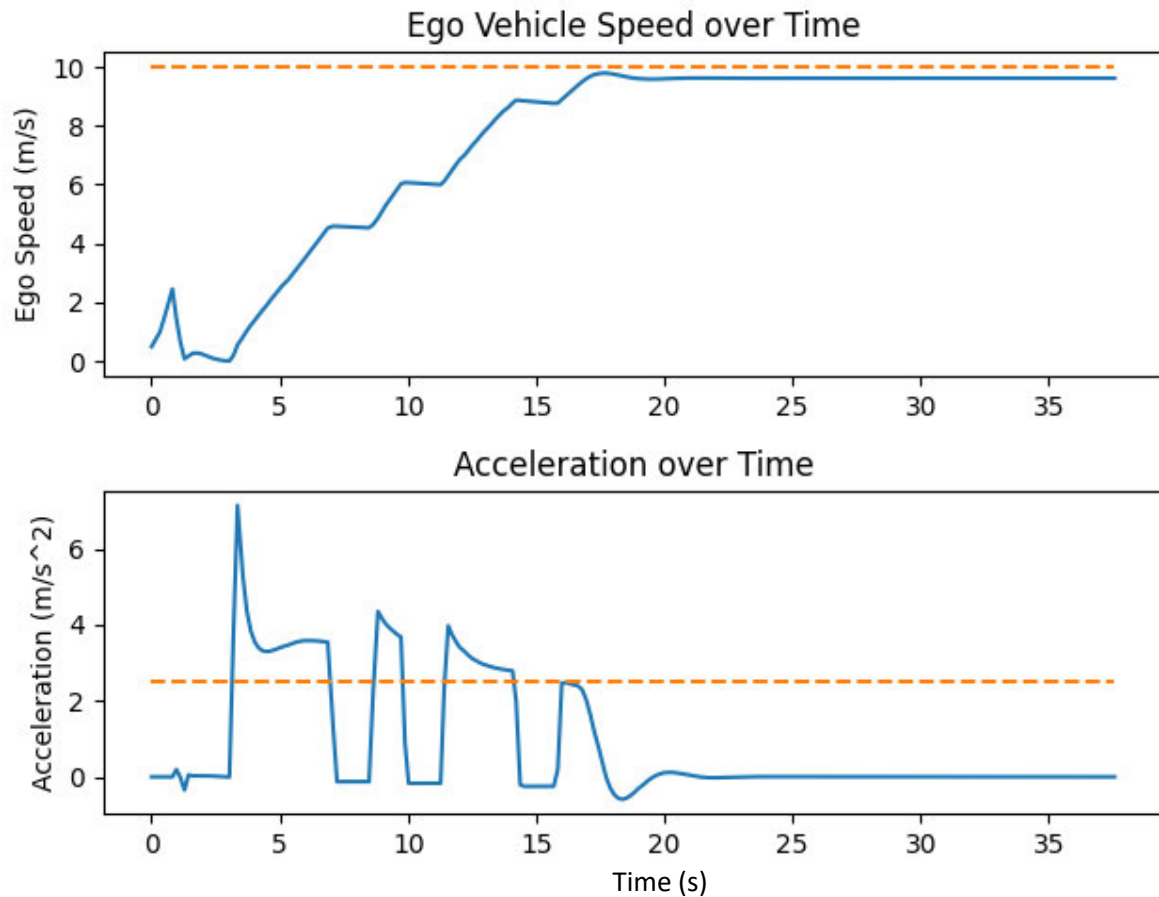


Figure 33 Tuning simulation result #1 ($K_p = 1.0$)

As depicted in Figure 33, it is observed that it took about 22 seconds for the ego to reach a steady state. Using the Matplotlib library to analyze further, we can see that the speed reached as close as $9,8 \frac{m}{s}$, then decayed for a few seconds and settled at $9,62 \frac{m}{s}$. That means we have a steady-state error of $0,38 \frac{m}{s}$, or 3,8%. The emergence of steady-state error aligns with expectations, given the current utilization of solely the proportional component within the controller. This phenomenon could also be attributed to the possibility that the assigned value of $K_p = 1$ might be comparatively low. Moreover, the instances of abrupt spikes within the acceleration curve appear to be largely attributed to the gear shifting process. The acceleration value surpasses the maximum comfort threshold, yet this isn't a concern as our focus lies on the ego speed curve as it approaches the set speed. Each occurrence of a "Plateau section" (as depicted in Figure 34) in the speed curve corresponds to a gear shift by the vehicle. During this phase, the speed remains constant momentarily, followed by an upward surge. Simultaneously, the acceleration rate during this interval declines to zero before experiencing a sudden ascent to a higher level. However, it's worth noting that the acceleration curve retains an overall downward trend.

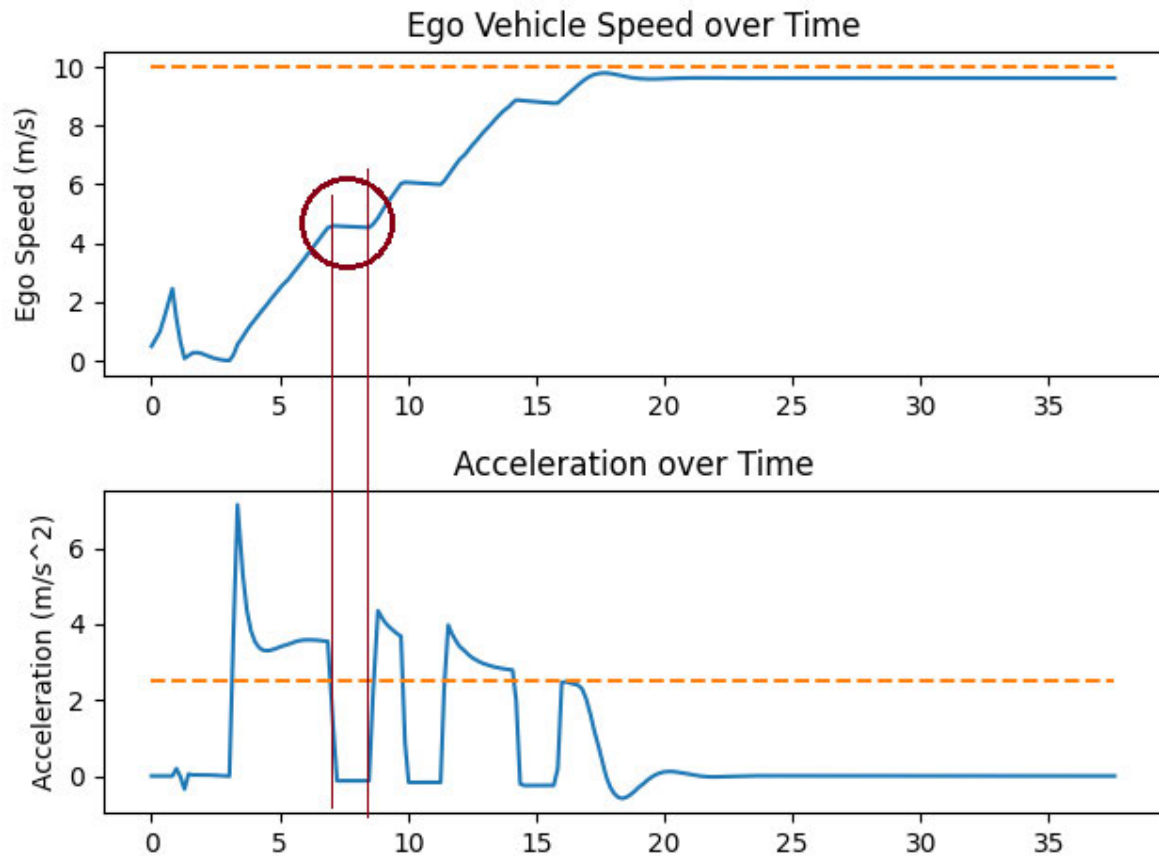


Figure 34 Plateau section

Additionally, the first spike in the speed curve is believed to be caused by the CARLA client dropping the ego vehicle onto the road (see Figure 35).

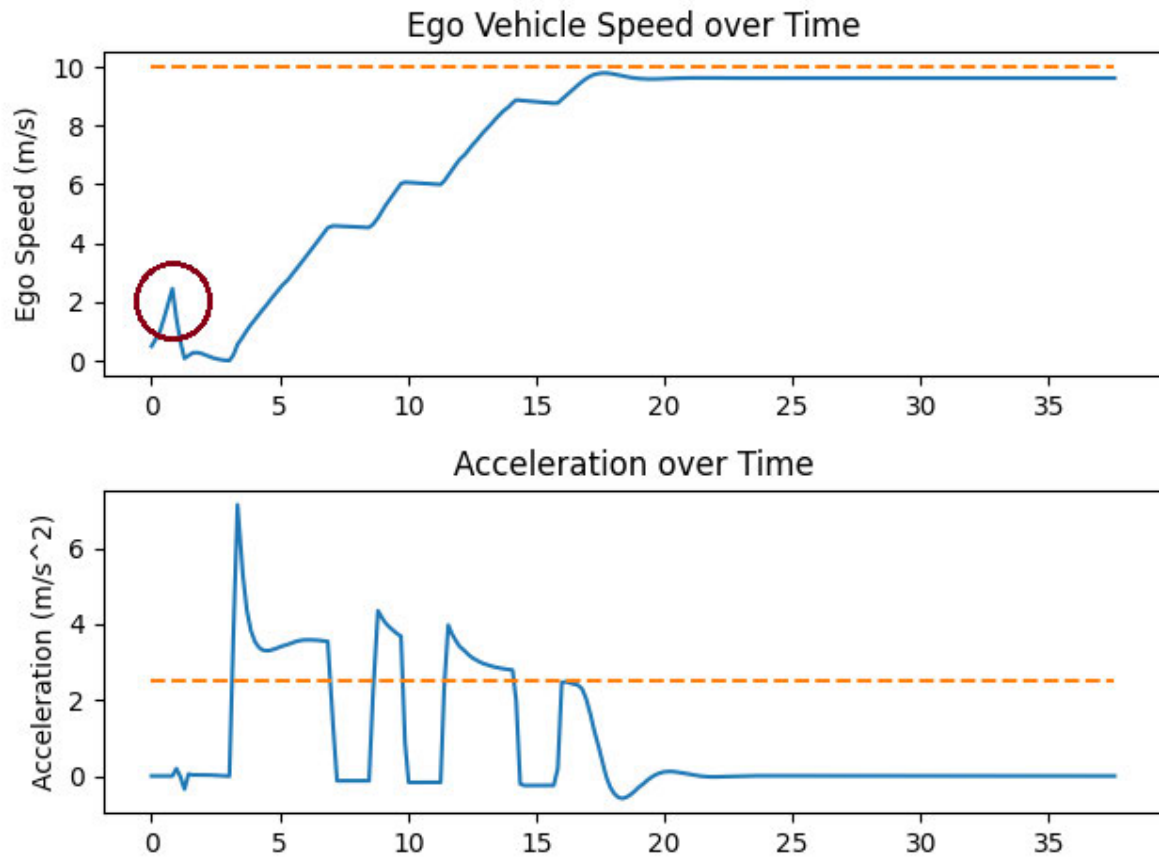


Figure 35 First spike in speed curve

Continuously raise the value to $K_p = 4,7$, a first time consistent sustained oscillation pattern is achieved (Figure 36).

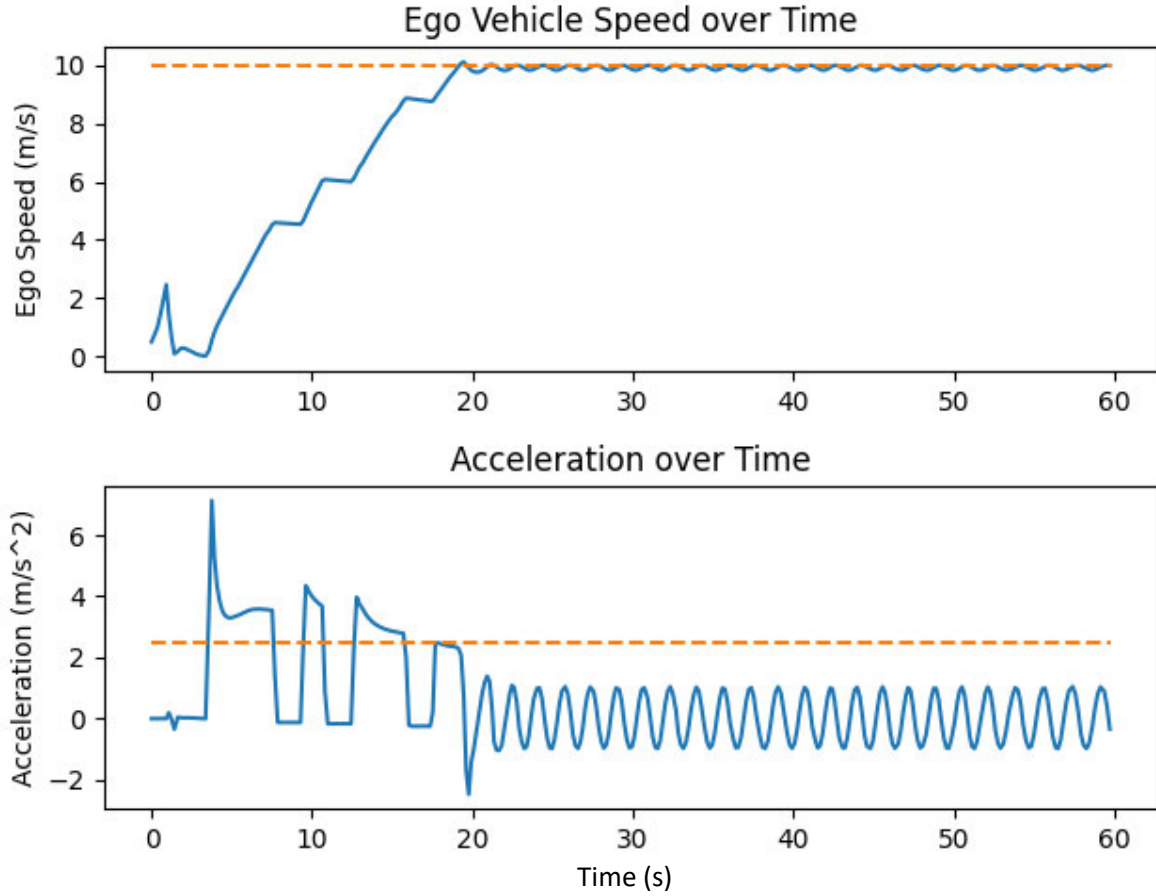


Figure 36 Sustained oscillation ($K_p = 4.7$)

Analyzing the graph, a measurement between two constant oscillation amplitude peaks shows that the oscillation period $T_u = 1.7s$. This outcome is achieved by identifying the timestamps of four peaks and subsequently calculating the average value to enhance precision. With this information, plus $K_u = 4.7$, we can calculate the optimal gains parameters of our classic PID controller:

$$K_p = 0.6 * K_u = 0.6 * 4.7 = 2.82$$

$$K_i = 1.2 * \frac{K_u}{T_u} = 1.2 * \frac{4.7}{1.7} = 3.32$$

$$K_d = 0.075 * K_u * T_u = 0.075 * 4.7 * 1.7 = 0.59$$

Equation 25 Calculating optimal classic PID gains parameters

Plugging these parameters into our PID controller, a set speed from $10 \frac{m}{s}$ will be the input, we have the following response (Figure 37):

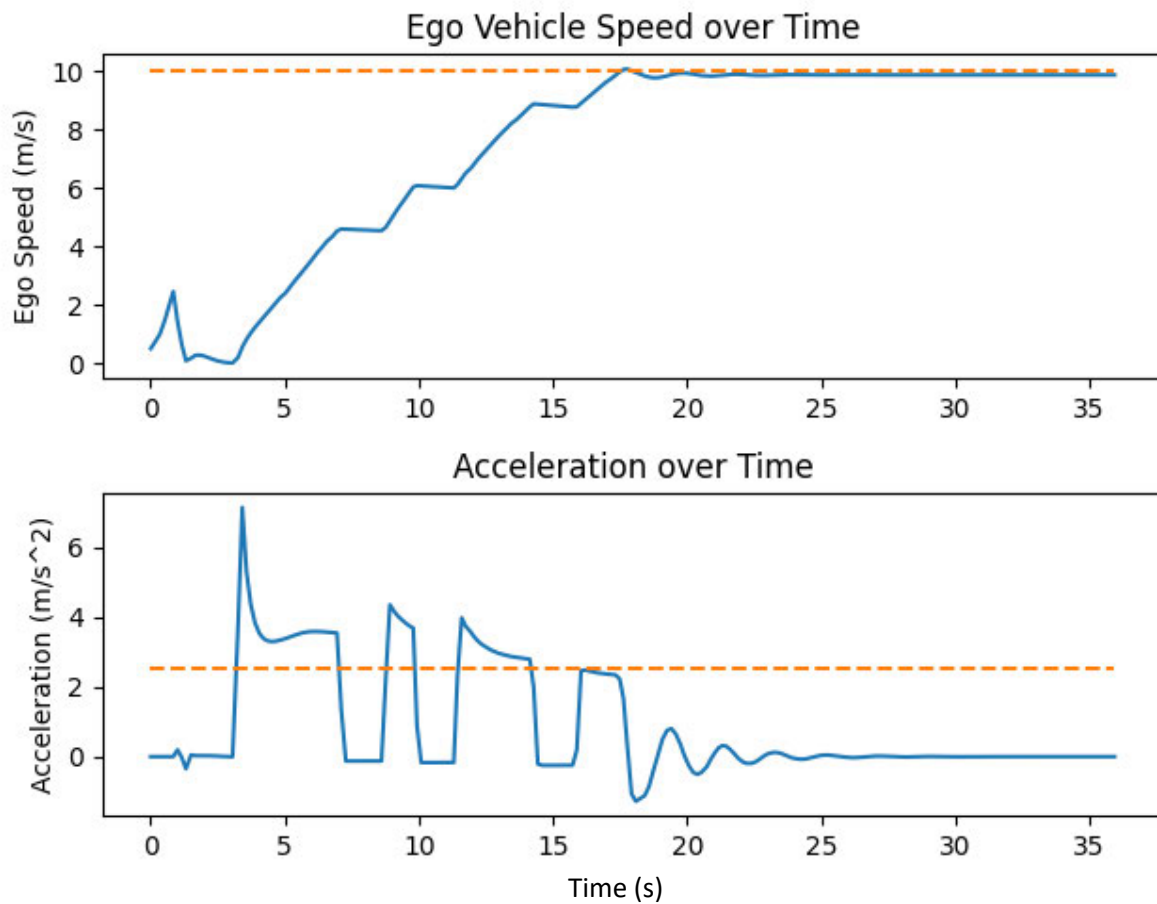


Figure 37 Response utilizing the classic PID gains parameters

The calculated rise time from the graph is approximately 18.6 seconds. While this might seem slow, it's important to take into account that the ego vehicle's initial speed was $0 \frac{m}{s}$, alongside with a maximum throttle allowed of 75%, which makes this rise time reasonable. Notably, there is an overshoot observed at $10,075 \frac{m}{s}$, equating to a 0.75% overshoot, which falls comfortably within an acceptable range. Following the overshoot, the waveform gradually subsided and stabilized at around $9,87 \frac{m}{s}$, exhibiting no sustained oscillations, meaning a steady-state error of 1,3%. The acceleration pattern appears erratic until approximately $t = 20.6 \text{ seconds}$. At this juncture, there was a swift decay in the waveform over the ensuing 5 seconds, ultimately "settling" close to $0 \frac{m}{s}$. A minor continuous oscillation, ranging from -0.01 to 0.01, was present. It's worth noting that each plateau in speed and acceleration attributed to gear shifting lasted for about 1.36 seconds. This duration is due to the clutch disengaging and is slightly longer than the desired timeframe (gear switch time is set to 0.5s). The elongated duration is believed to stem from the clutch strength parameter (set at 10 for this simulation) and the inherent gear shifting mechanism of the vehicle itself. Additionally, through the vehicle telemetry command, it is known that the ego vehicle is operating at fourth gear. In summary, the initial 15 seconds of the process might induce some discomfort, but once the targeted speed is attained, the absence of oscillations and consistent speed adjustments should ensure a smooth ride thereafter.

Testing with a different set speed, we have the following result:

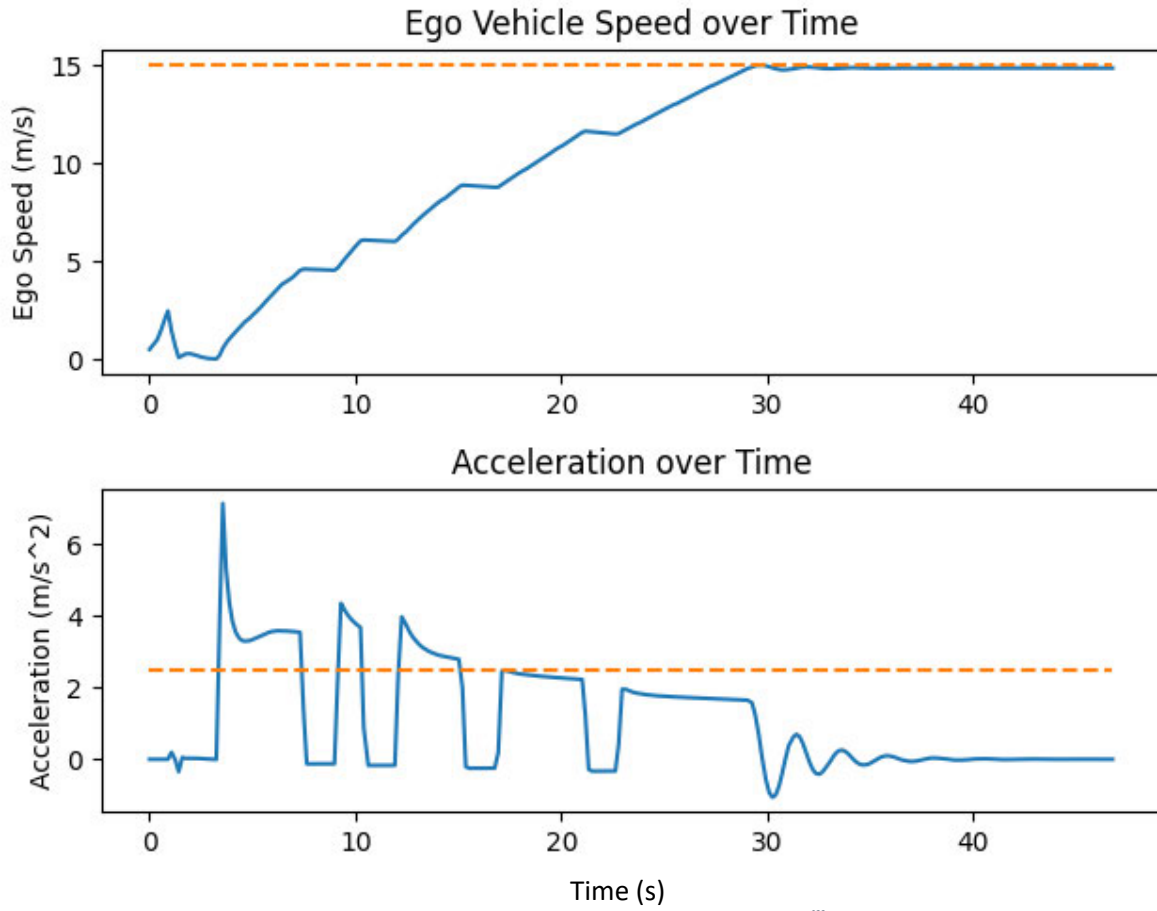


Figure 38 Classic PID controller at set speed= $15 \frac{m}{s}$

Figure 38 exhibits a behavior akin to that of Figure 37, albeit with some minor distinctions. Notably, an absence of overshoot is observed; the speed attains a maximum of $14,99 \frac{m}{s}$ before stabilizing at $14,85 \frac{m}{s}$. This results in a slightly smaller steady-state error of 1%. Another distinction is the presence of four plateau segments, signifying an additional gear shift in the ego vehicle as compared to the previous simulation, in fifth gear. Apart from these differences, the initial discomfort period spanning 15 seconds remains consistent. Likewise, the attenuated wave patterns within the acceleration curve exhibit similarity to those seen previously.

Increasing the set speed to $20 \frac{m}{s}$ and $25 \frac{m}{s}$, we have the responses in Figure 39 and Figure 40, respectively:

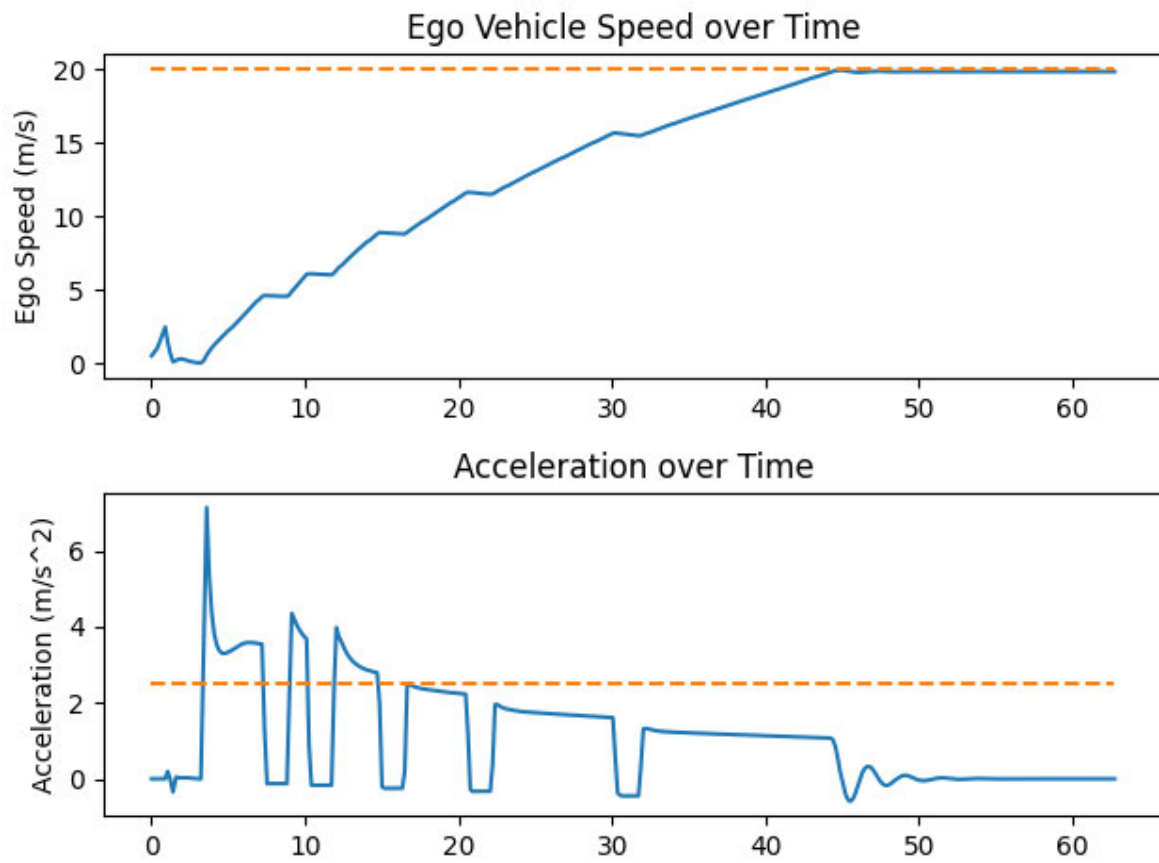


Figure 39 Classic PID controller at set speed= $20 \frac{m}{s}$

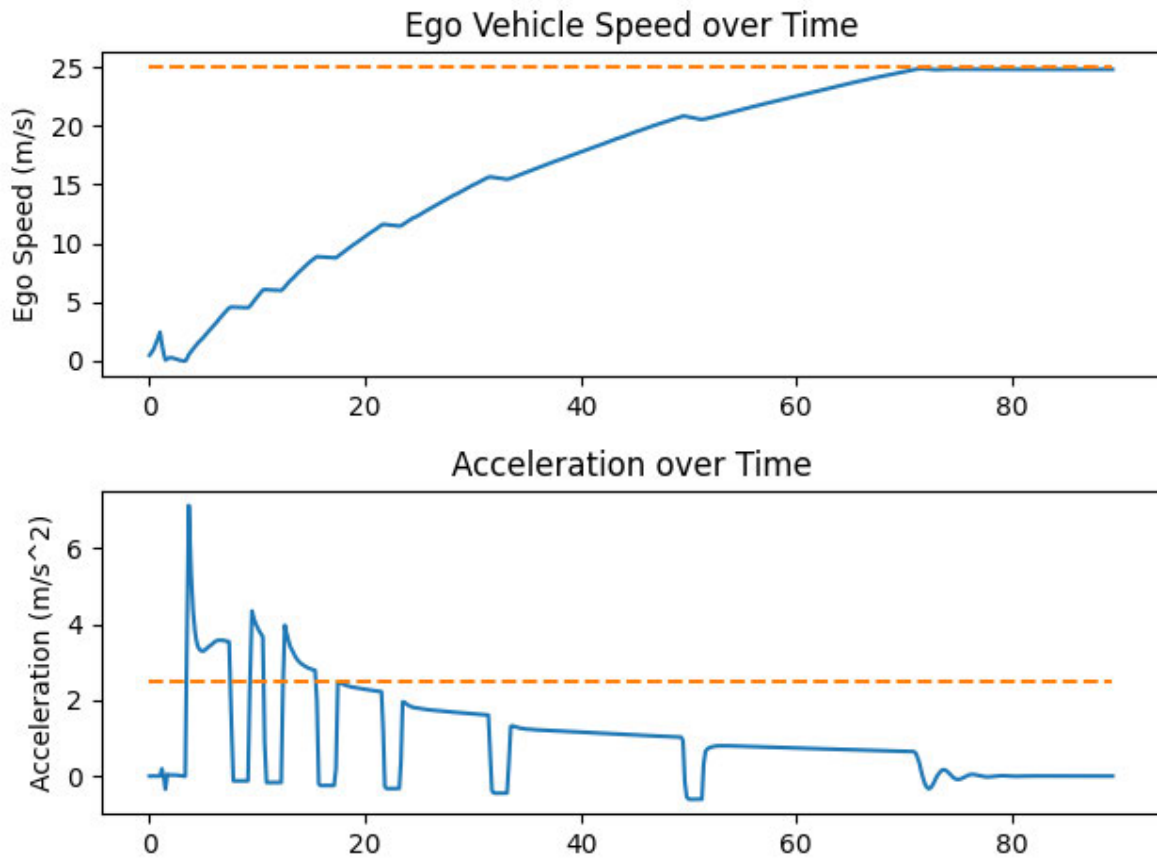


Figure 40 Classic PID controller at set speed= $25 \frac{m}{s}$

By examining the two graphs, it becomes evident that there is an absence of overshoot in both cases. The steady-state error remains unchanged in Figure 39, and notably decreases to 0,8% in Figure 40. The sole distinction lies in the increased frequency of gear shifts. From these observations, the deduction can be made that our PID controller exhibits enhanced performance at higher speeds. This outcome is highly satisfactory, signifying our readiness to implement the PID controller with the newfound gain parameters in real-world, intricate scenarios.

3.2.5 Main test scenarios and prediction

Once the PID controller is accurately tuned, our system becomes ready to be tested in basic, real-life scenarios. These scenarios will encompass various fundamental cases in which the ACC system will be employed. They will be thoroughly described and analyzed before being implemented within the CARLA simulation environment. The anticipated performance standard generally involves achieving minimal to no overshooting, ensuring stability, minimizing oscillations, all while successfully maintaining the desired distance.

3.2.5.1 Approaching target vehicle

The objective of this simulation is to assess the performance of the ACC system as it attempts to reach the set speed but is interrupted by the presence of a target vehicle, resulting in a deceleration command from the system. First step is setting up the CARLA client and world (similar to 3.2.4.1 Lidar), with the map “Town 6”. In this world ego and target vehicle will be spawned in, with a distance of x meter(s) relative to one another. There are no other vehicles in the vicinity. The Dodge charger is assigned as the ego vehicle, the Tesla will act as target vehicle. Both vehicles’ initial velocities are at

$0 \frac{m}{s}$. Then applying the PID controller to the ego vehicle (similar to 3.2.4.4 PID controller tuning). Once ACC is activated through in the script, ego's speed and steering will be regulated by the PID controller. A predetermined set speed is assigned to the ego vehicle, ensuring it is greater than the target vehicle's current speed to facilitate catching up. The CARLA client is used to retrieve and calculate the distance between the two vehicles. The ego vehicle is driven at the set speed until the desired distance is smaller than the actual distance to the target vehicle. At this point, the ACC system intervenes and commands the ego vehicle to decelerate by managing the throttle and brakes. The cycle can be described as seen in Figure 41, however, the assessment occurs exclusively once the state of steady following has been established and does not extend beyond this point.

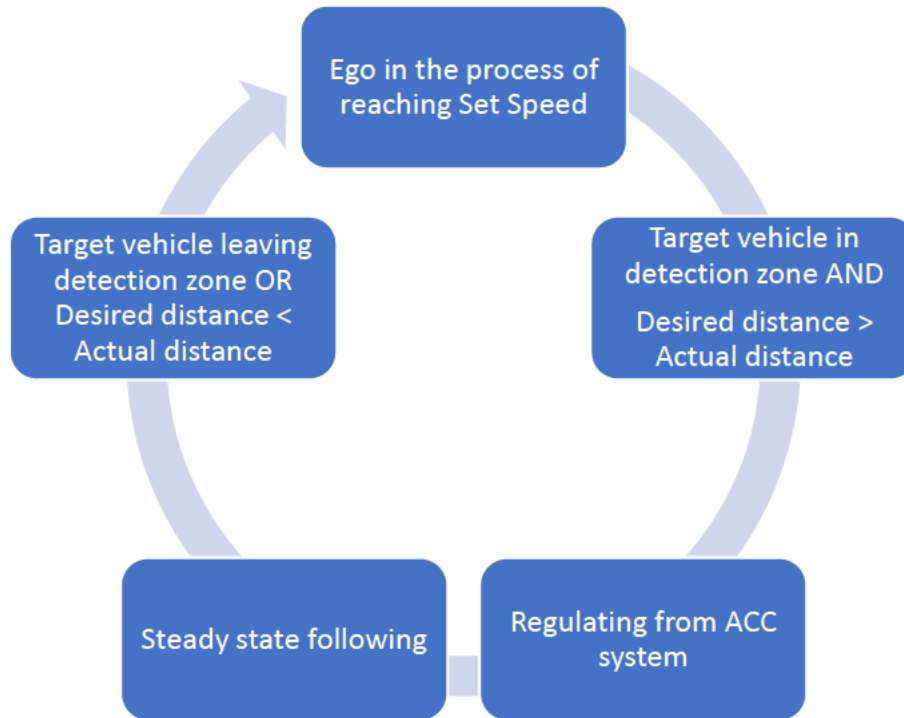


Figure 41 ACC regulating cycle

However, with the target vehicle, the autopilot function [54] will be turned on. The use of autopilot for the target vehicle introduces an element of unpredictability to the simulation, as the vehicle's behavior is controlled by the built-in artificial intelligence (AI) system. This unpredictability means that the exact desired scenarios might not always be guaranteed. Therefore, the data and graphs collected during the simulations provide valuable insights. With this information, we can analyze and classify different scenarios, enabling us to fine-tune the control script. By iteratively adjusting and tweaking the script based on the gathered data, we can create specific scenarios that closely match the desired conditions for evaluation. This iterative approach allows us to optimize according to the autopilot behavior, ensuring that the target vehicle exhibits more consistent and predictable responses in the scenarios we want to evaluate. If the autopilot system exhibits erratic behavior, proving desired scenarios difficult to obtain, we have the option to rely on the tuned PID controller and apply it to the target vehicle. This approach ensures that we can create our desired scenarios more consistently. However, it's essential to recognize that the accuracy of these simulations heavily affected by the tuning performed ourselves in the previous sections. For example, a poorly tuned PID controller will cause the target vehicle to fail to maintain constant speed (system unstable), and so on. Thus, this limitation should be taken into account when evaluating the simulation results. For now, the autopilot method will be utilized, meaning three sub-scenarios are defined, namely approaching accelerating

and decelerating, along with target vehicles travelling at constant speed. The simulations are to be repeated until at least one of each sub-scenario is correctly categorized.

During the approaching process, the performance of the ACC system will be assessed at different initial distances between the ego vehicle and the target vehicle, as well as various set speeds. Additionally, the behavior of the ACC system after the approaching process will be monitored. The goal of the ACC system is to effectively manage the speed of the vehicle and maintain a safe distance from the target vehicle. The distance between the two should be equal to or less than the desired distance, giving the driver enough time to react to sudden stops or unexpected behavior. For the passengers' comfort, the maximum acceleration and deceleration of the ACC system should not exceed $2.5 \frac{m}{s^2}$. This ensures a smooth and comfortable ride by avoiding sudden changes in speed. Additionally, the ACC system should avoid excessive oscillations and overshooting while matching the target vehicle's speed. Precise control is necessary to achieve this without making aggressive throttle inputs.

3.2.5.2 Alternating headway values

The goal of this test is to simulate the driver choosing different desired distances relative to the target vehicle. Similar to the setup used in the two scenarios mentioned before, we'll have the two vehicles spawning at $0 \frac{m}{s}$, set speed will be set so that its always greater than target vehicle's speed to facilitate catching up. The PID controller applied to ego will also be in charge of maintaining various constant speeds for the target vehicle. However, the ego vehicle's set speed will remain unchanged throughout the test. Initial distance should be minimized in order to expedite the process of catching up, thus reducing testing time. The "Town 6" map will be our environment, which provides us with sufficient road length for our tests. Once the catching up process is complete and the following process is successful, the two vehicles will travel at relatively constant speeds while maintaining the desired distance. Only then, the headway value will then be changed, which will alter the desired distance. There will be three options for the headway value, ranging from $2.5s$ to $1.5s$, corresponding to high, medium, and low levels. A pattern is also defined, for example: from low to medium, from medium to high, from high to low, from low to high, etc., each for a certain amount of time. After a stable following state is reached, another alternation can be executed (see Figure 42).

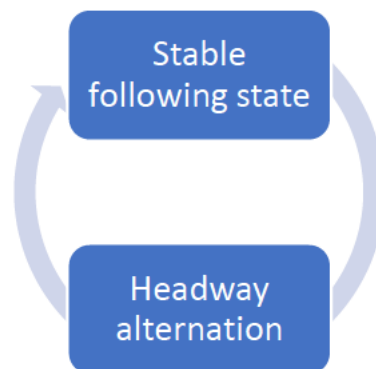


Figure 42 Headway alternating regulating cycle

Expected from this test is that the ACC system will maintain the desired headway values, thus keeping the ego vehicle within a safe distance relative to the target vehicle. The performance in the period after an alternation of headway is also to be observed, so that any overshooting, oscillation, or instability in the system is detected. The smoothness of acceleration and deceleration when the ACC system adjusts ego vehicle's speed is also taken being evaluated, to ensure comfort for the passengers.

3.2.5.3 Target vehicle decelerating

The goal here is to evaluate how well the ACC system's response to gradual all the way to sudden deceleration from the target vehicle from a steady following state. The two vehicles are spawned in with a relatively small initial distance, and an initial speed of $0 \frac{m}{s}$. PID will be responsible for both vehicles' speeds adjustments. By employing varying PID gains for the target vehicle, we can replicate unforeseen behaviors akin to those exhibited by human drivers on the road. After achieving a steady following state, the target vehicle will be instructed to decelerate. Various levels of deceleration will be tested, and the response of the ego vehicle will be monitored. The evaluation criteria applied in the previous two simulations will be utilized to assess the results. A simple chart of the process can be seen in Figure 43. Ego vehicle should decelerate as well and maintain the desired distance relative to the target vehicle.

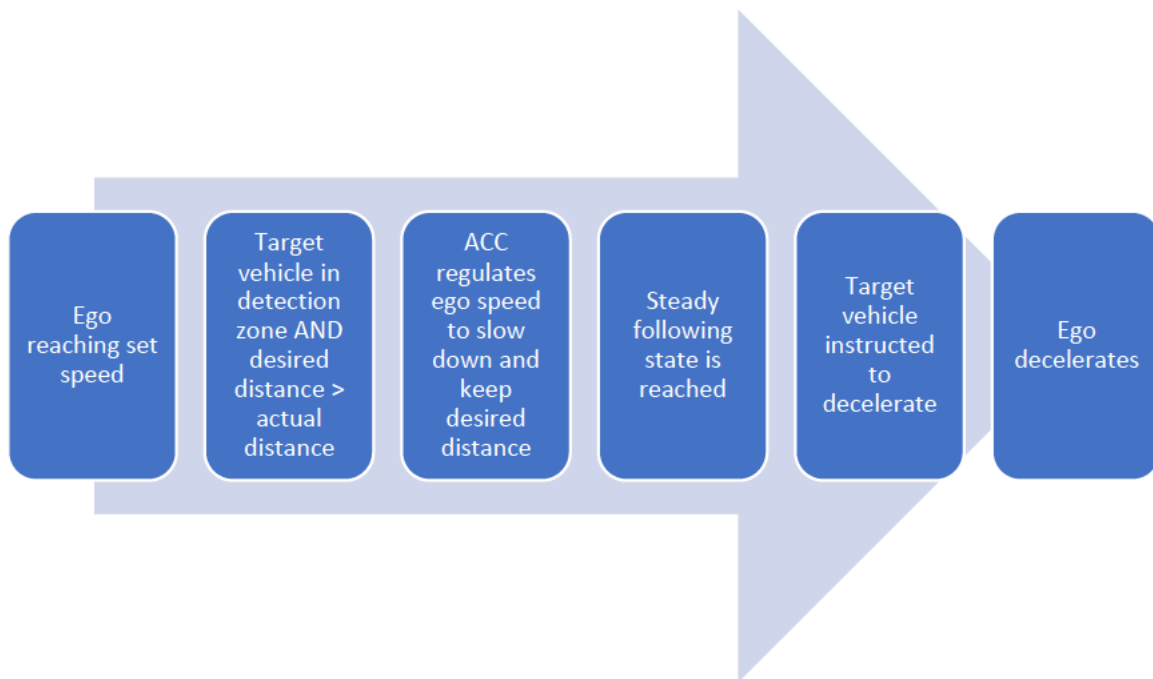


Figure 43 Target vehicle decelerating process

3.2.5.4 Target vehicle accelerating

The goal here is to evaluate how well the ACC system's response to an accelerating target vehicle from a steady following state. Both vehicles are spawned in at $0 \frac{m}{s}$ initial speed and a set distance from one another. To facilitate catching up, we will set a higher set speed than the target vehicle's current speed. Once a stable following state is established, the target vehicle will receive an acceleration command (see Figure 44). Various acceleration rates will be tested, with the constraint that the acceleration should remain within a range that prevents the actual distance between the two being greater than the desired distance. The evaluation criteria applied in the previously mentioned simulations will be used for assessment in this scenario as well. Ego should be accelerating when the target vehicle is accelerating, maintaining the desired distance.

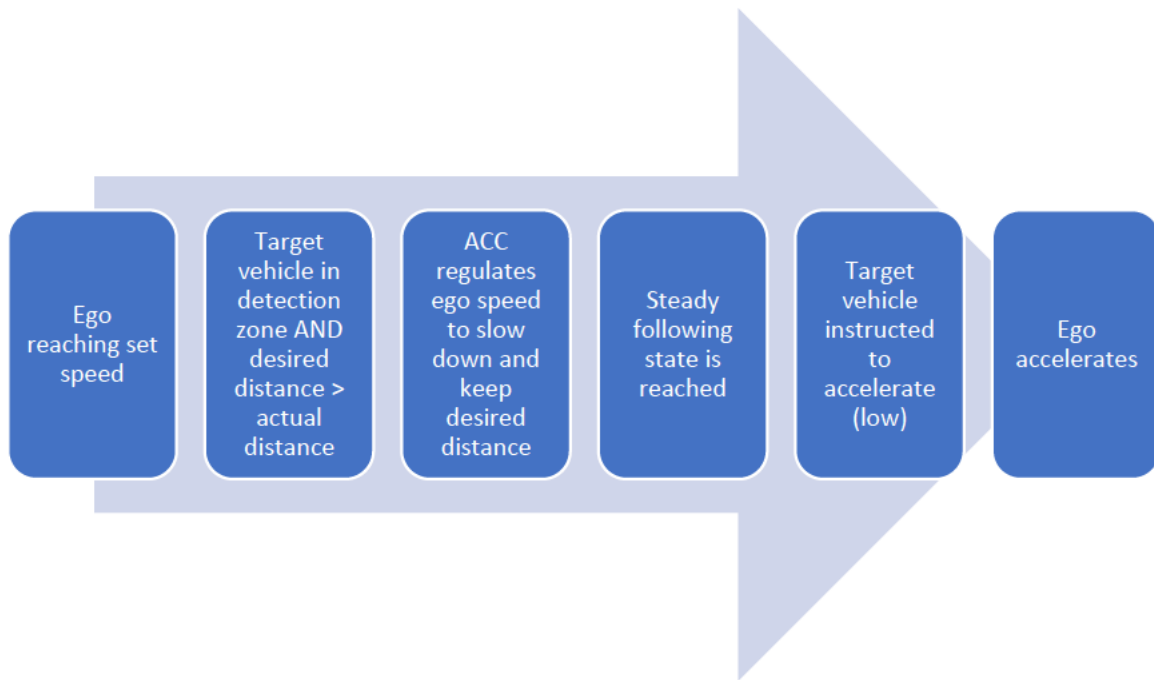


Figure 44 Target vehicle accelerating process

3.2.5.5 Target vehicle speeding off

The premise stays the same as 3.2.5.4 Target vehicle accelerating. In the final phase however, when commanding the target vehicle to accelerate, a significantly higher acceleration rate will be employed. This intentional approach is aimed at ensuring that the actual distance surpasses the desired distance. Anticipated behavior involves the ego vehicle initially accelerating along with the target vehicle. Subsequently, when the actual distance becomes less than or equal to the desired distance – indicating that the ego has reached the set speed – the ego vehicle's speed will remain consistent, even if the target vehicle continues to accelerate (see Figure 45).

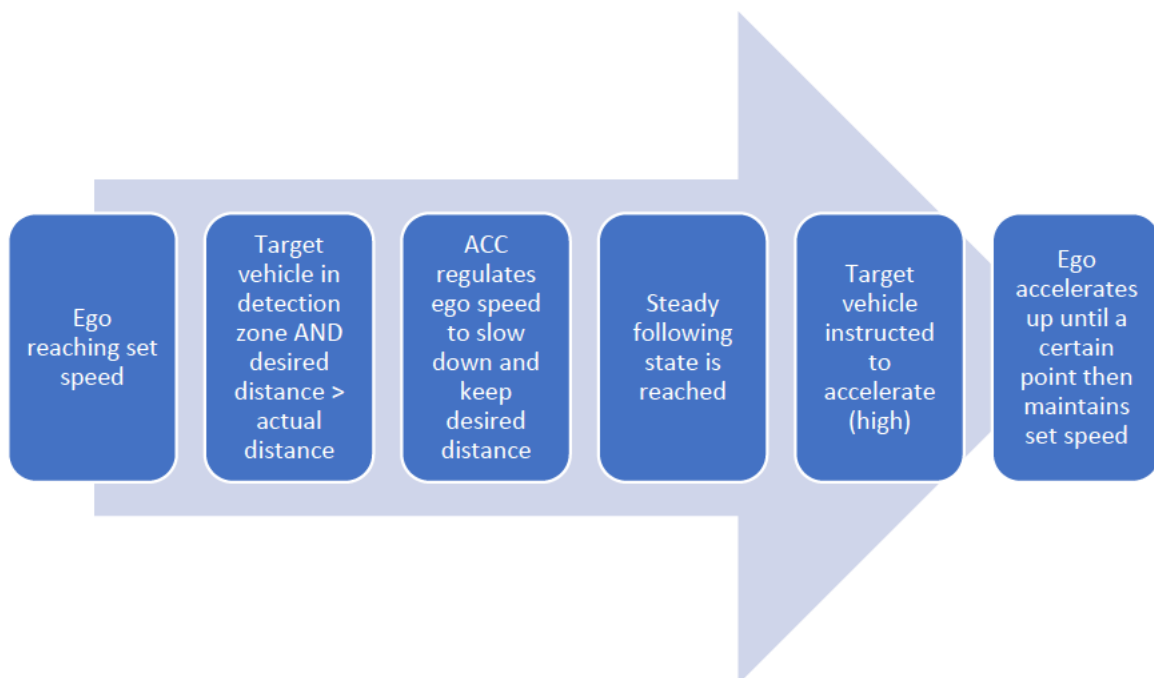


Figure 45 Target vehicle speeding off process

3.2.6 Advanced scenarios

Considering the limitations in time, the advanced scenarios will focus on a single chosen scenario, specifically the “driving in curve” scenario. The enhanced ACC system encompasses both longitudinal and lateral control capabilities. With the PID controller already in place for lateral control, the remaining steps involve placing our ego vehicle within a map featuring curved roads, setting the target vehicle to autopilot, and observing the system's response to alterations in speed and lateral positioning.

3.3 Implemented control logics

This sub-section describes the control logic and actions taken by the vehicle based on different conditions, including managing time headway, collision prevention, control using a PID controller, and applying throttle or brake values to adhere to acceleration and deceleration limits. The following points encompass the various logics employed when executing the simulation scripts:

- Time Headway Calculation:
 - If the ego vehicle's speed (`ego_speed`) is greater than 1 m/s (to avoid divided by 0):
 - Calculate the time headway as the absolute value of distance divided by `ego_speed`.
 - Otherwise (when `ego_speed` is 1 m/s or lower):
 - If distance is greater than 0 (to prevent division by zero when distance is not zero):
 - Set the time headway to positive infinity (to better illustrate in diagrams as infinity in y-axis will not be shown)
 - Else (when distance is also zero):
 - Set the time headway to 0 (to avoid divided by 0).
- Collision Imminent Check:
 - If the calculated `time_headway` is less than 1:
 - Apply maximum braking (`brake=1`) to the vehicle to prevent a collision (simulate the driver applying full brake).
 - Display a message indicating an imminent collision.
- Scaling Factor Definition:
 - Define a scaling factor of 0,2 for smoothed brake responses.
- PID Controller Logic:
 - If distance is less than the `desired_distance`:
 - Use a PID controller with lateral and longitudinal arguments to control the vehicle's behavior based on the relative distance to desired headway.
 - Else if the difference between distance and `desired_distance` is less than 5% of `desired_distance` and ego's current speed is smaller than the set speed:
 - Use the same PID controller to control the vehicle's behavior based on the relative distance to desired headway. Target speed input for the controller = `distance/desired_headway`.
 - Otherwise (when neither of the above conditions are met):
 - Use the same PID controller to control the vehicle's behavior, targeting a set speed slightly higher than the desired speed at +0,1 (compensating for steady state error).
 - Maximum Acceleration Exceeded Check:
 - If the acceleration (`acceleration`) is greater than the maximum allowed acceleration (`a_max`):

- Calculate the difference between acceleration and $a_{\max}$ (accel_difference).
- The formula $(\text{accel_difference}^2)$ is used to create a parabolic curve. This is done to ensure that the braking response is proportional to the degree of acceleration beyond the limit.
- By squaring the accel_difference, the resulting value increases rapidly as the difference grows larger, promoting a gradual increase in brake response.
- The calculated parabolic value is then multiplied by a scaling factor (scaling_factor) to control the magnitude of the brake response.
- A smaller scaling factor like 0,2 ensures that the braking is not overly aggressive, allowing for a smoother and more controlled deceleration.
- To avoid excessively strong braking, the calculated brake value is capped at a maximum value, in this case, 0,2. This means that even if the parabolic calculation yields a higher value, the brake value applied will not exceed 0,2.
- Set throttle to 0 and apply the calculated brake value.
- Maximum Deceleration Exceeded Check:
 - If the acceleration is less than the negative of the maximum allowed deceleration ($-a_{\max}$):
 - Apply a throttle control value of 0,3 to the vehicle to prevent exceeding the maximum deceleration.
 - Release the brake.

4 Execution and results

Within this section, we will establish the precise parameters before initiating the simulation execution. This involves configuring the specific values that govern the vehicle's behavior and interactions. By meticulously setting these parameters, we aim to replicate real-world scenarios as accurately as possible, thus enabling us to comprehensively evaluate the performance of the ACC system. Note that if the PID gains are referenced without additional context, it signifies their utilization for the PID controller of the ego vehicle. Note that if the PID gains are referenced without additional context, it signifies their utilization for the PID controller of the ego vehicle.

4.1 Executing the defined scenarios and results

4.1.1 Approaching target vehicle

4.1.1.1 Approaching scenario #1

The initial parameters are as follows:

- Initial distance between two vehicles :15m
- Target vehicle speed = autopilot
- Set speed = $20 \frac{m}{s}$ (target value input = 20,1)
- Desired headway = 2,5s
- Map=Town 6, only straight roads
- PID lateral gains $K_p = K_i = K_d = 0$
- PID longitudinal gains $K_p = 2,82; K_i = 3,32; K_d = 0,59$

- Ego = Dodge Charger, target vehicle = Tesla Model 3

After the simulation is executed, this is the result as shown in Figure 46:

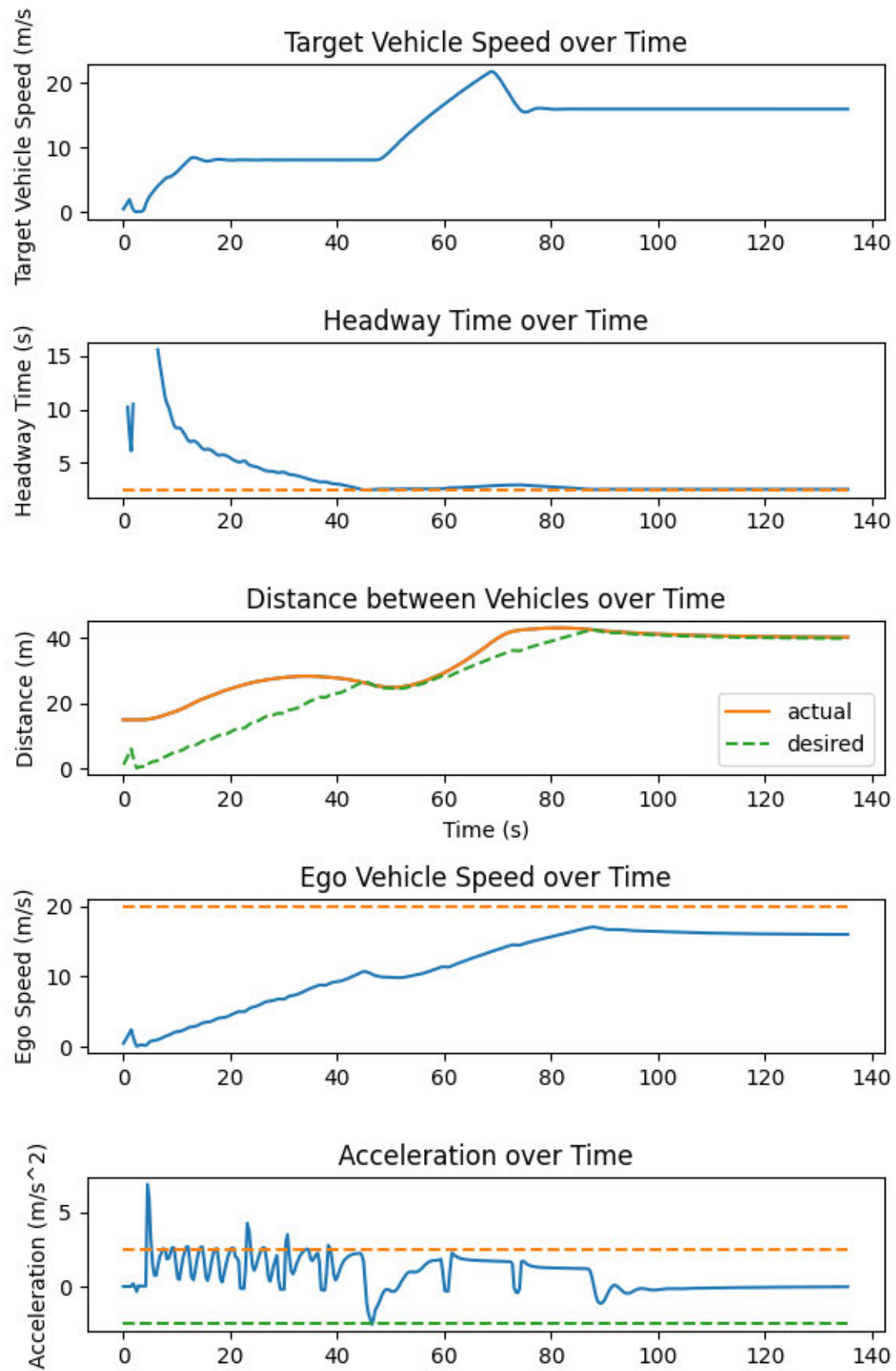


Figure 46 Result from the approaching scenario #1

The initial spike observed in all graphs can be attributed to the CARLA client placing vehicles onto the road, which is not a concern for analysis. Subsequently, the target vehicle exhibited an acceleration phase for approximately 13 seconds, followed by a sustained speed of about $8 \frac{m}{s}$ for approximately 35 seconds. During this period, the ego vehicle aimed to reach its designated speed. The speed curve displays slight irregularities, attributed to braking commands triggered when acceleration surpassed its maximum allowed value. Though instances of acceleration exceeding $2,5 \frac{m}{s^2}$ persisted, such as at seconds 24 and 32, around $4 \frac{m}{s^2}$, this can be mitigated by more pronounced braking and constraining throttle usage within the ACC system. However, such an approach might lead to extended rise times and frequent high-level braking, making it less advisable.

In the initial 50 seconds, the frequency of acceleration changes stood at 0.33 Hz, stemming from the process of reaching 2.5 m/s^2 and subsequently releasing the throttle. At approximately second 47, a speed plateau was evident, momentarily dipping to signify a gear shift. By second 46, the ACC system initiated speed regulation for the ego vehicle to maintain the desired distance. A marginal overshoot at 27,05 meters occurred, whereas the intended distance was calculated to be 26,61 meters (resulting in a 1.65% overshoot). This can also be observed in the actual headway curve, with a bump at the same time. The desired distance curve closely tracked the actual distance curve for the subsequent two seconds. Following this, the target vehicle's acceleration led to the actual distance curve outpacing the desired distance curve. At nearly $22 \frac{m}{s}$, the target vehicle promptly decelerated, stabilizing at approximately $16 \frac{m}{s}$ until the simulation's conclusion.

Throughout this interval, the ego vehicle persisted in pursuing its set speed. However, around $17 \frac{m}{s}$, the two distance curves converged, compelling the ego vehicle to decelerate. Subsequently, the ego speed was regulated to align the two distance curves. The simulation culminated with the ego vehicle's speed at $16 \frac{m}{s}$, successfully matched the target vehicle's speed while maintaining the desired distance. The acceleration curve exhibited oscillations around 0 and commenced decay around the 89-second mark, eventually stabilizing at $0 \frac{m}{s^2}$. This signified the establishment of a steady following state. The set speed was however never attained, primarily attributed to the initial short distance between vehicles. Assessing with the pre-determine aspects, the following scores were given in Table 6:

Aspect	Score
Steady following state	10
Desired distance/Headway keeping	8
Overshoot	9
Oscillation	8
Steady state error	-
Response	9
Smoothness in braking and throttle control	8
Total score	52/60

Table 6 Scoring for the approaching scenario #1

Minor overshoots and deviations in maintaining the desired distance were observed, necessitating potential adjustments in throttle and brake interventions. The system exhibited diligent efforts in matching the target vehicle's fluctuations, despite occasional instances of mismatched distance curves. Towards the simulation's conclusion, the ACC system adeptly regulated the ego vehicle's speed to harmonize with the target's pace. While some deviations and oscillations were noted, the system managed to settle into a steady following state. Overall, the ACC system displayed competence in

maintaining safe distances and managing speed variations, while pinpointing areas for refinement in achieving a more consistent and seamless performance.

4.1.1.2 Approaching scenario #2

The initial parameters are as follows:

- Initial distance between two vehicles :40m
- Target vehicle speed = autopilot
- Set speed = $20 \frac{m}{s}$ (target value input = 20,1)
- Desired headway = 2,5s
- Map=Town 6, only straight roads
- PID lateral gains $K_p = K_i = K_d = 0$
- PID longitudinal gains $K_p = 2,82; K_i = 3,32; K_d = 0,59$
- Ego = Dodge Charger, target vehicle = Tesla Model 3

Leaving the simulation run for a while longer, the result received is as shown in Figure 47:

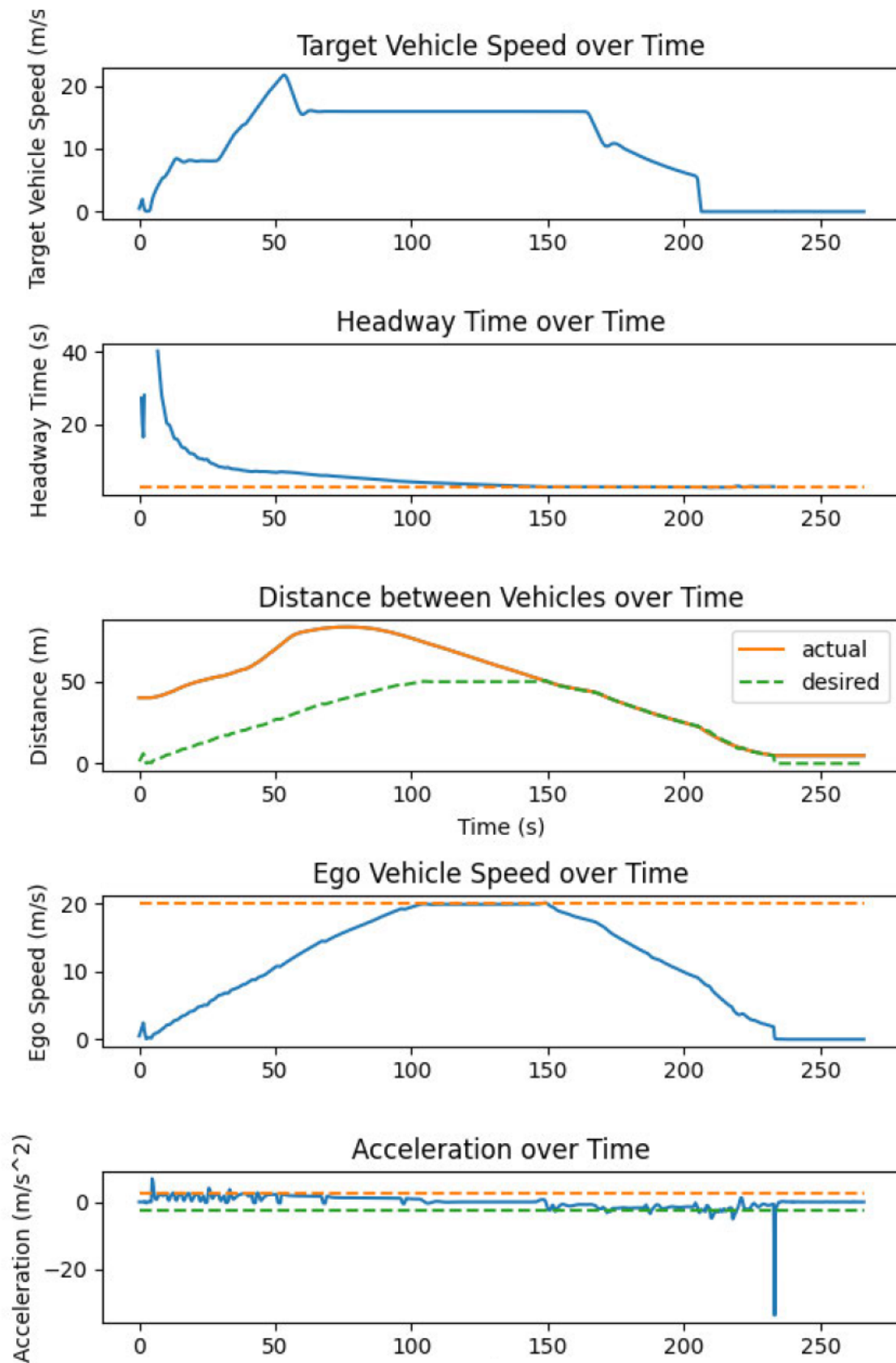


Figure 47 Result from the approaching scenario #2

Initially, the speed and acceleration curves exhibit resemblances to the prior simulation, marked by oscillations while aiming to attain the set speed. Notably, a distinctive aspect emerges as the set speed is accomplished at approximately the 103-second mark, characterized by a marginal overshoot at 20.01 m/s, yielding nearly negligible overshoot (0.05%). However, a subsequent drop in ego speed transpired, followed by the gradual decay of oscillations, ultimately leading to speed stabilization at 19.92 m/s. This outcome translates to a minimal steady-state error of 0.04%. This behavior is elucidated in Figure 48.

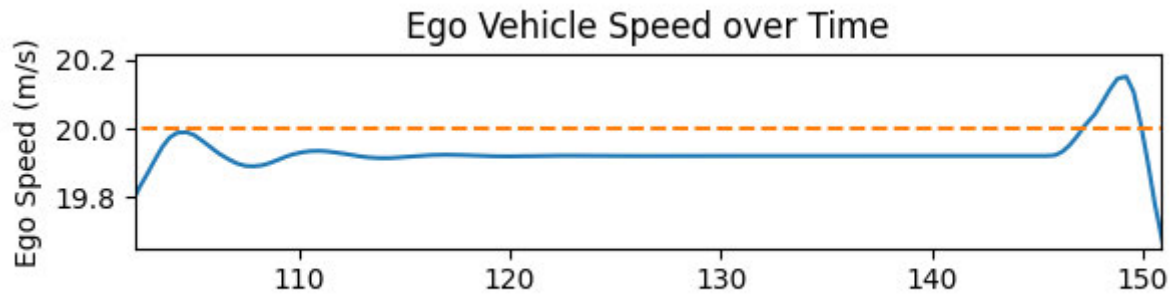


Figure 48 Closer inspection in the speed curve

Figure 48 further illustrates a discernible spike in ego speed just before the convergence of the two distance curves, reaching a peak at $20.16 \frac{m}{s}$ before rapidly descending. This transition seamlessly aligns the two curves, characterized by minimal oscillations ($<0.1\%$). Remarkably, the ACC system adeptly regulated ego speed to ensure synchronization with the target vehicle's deceleration, preserving curve conformity even amid abrupt changes. This behavior persisted during the target vehicle's sudden deceleration around the 205-second mark. As the target vehicle decelerated, the acceleration curve displayed a notably uneven pattern characterized by oscillations around 1 Hz. These oscillations exhibited varying amplitudes, some of which surpassed the maximum comfort threshold. (deceleration at $3,1 \frac{m}{s^2}$; $4,7 \frac{m}{s^2}$; and $5 \frac{m}{s^2}$) (see Figure 49)

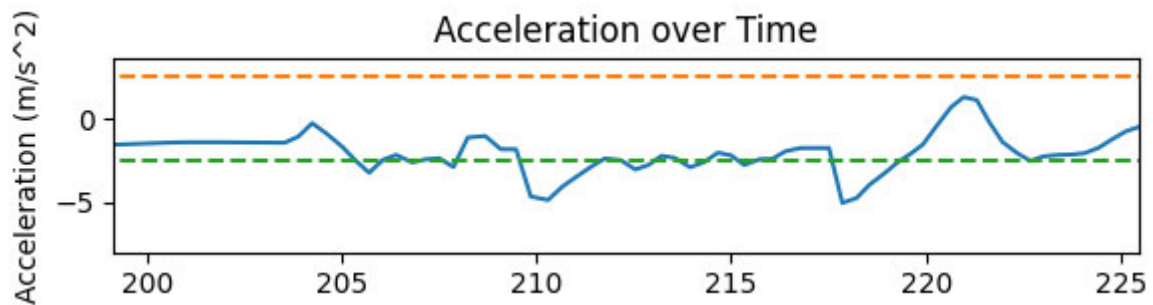


Figure 49 Jagged curve behavior in deceleration

Although a minor enlargement in the error between the curves became apparent, its significance remained marginal. Approximately at the 233-second juncture, the distance curves diverged, resulting in a notable gap between them. Concurrently, the ego's acceleration rate plummeted to approximately $-32 \frac{m}{s^2}$ (!). Subsequently, the target vehicle decelerated to a halt, inducing the ego to also come to a standstill approximately 30 seconds later, maintaining a final distance between them of 4,7m. Note that the emergency simulated brake from the driver was used when the distance between two vehicles was at 10m.

The scores were given as seen in Table 7:

Aspect	Score
Steady following state	10
Desired distance/Headway keeping	8
Overshoot	8
Oscillation	7
Steady state error	8
Response	9
Smoothness in braking and throttle control	6
Total score	56/70

Table 7 Scoring for the approaching scenario #2

Notably, the ACC system exhibits a commendable ability to minimize overshooting, achieving the desired set speed with remarkable precision and only slight deviations. However, subsequent speed variations indicate room for improvement in maintaining constant speeds post-achievement. The minor oscillations and steady-state errors are noteworthy and could potentially be addressed through fine-tuning of control parameters, e.g., increasing integral gain, increasing error window size. One other possible solution is to increase the target input value to e.g., $+0,2 \frac{m}{s}$, however, a higher rate of overshooting might occur. The peak in speed just before the two distance curves is yet to be explained. However, the successful synchronization of speed changes with the target vehicle's deceleration is a strong testament to the ACC system's effectiveness in real-time adjustments. Contrary to expectations, the divergence observed in the distance curve around the 233-second mark doesn't signify a system fault; rather, it reveals a flaw attributed to the method used for calculating the desired headway. The headway calculation, as presented in Equation 8, yields an erroneous outcome whereby the desired distance becomes 0 meters when the ego vehicle's speed reaches 0 m/s. This incongruence is evidently inaccurate. One proposed remedy involves incorporating an "if" statement: when the ego vehicle's speed equals 0, set the desired distance to infinity. This corrective measure effectively eliminates the desired distance during that time interval, ensuring that only the actual distance curve remains depicted in the graph. The significant decrease in acceleration is attributed to the activation of a safety measure designed to simulate the driver's response to an imminent collision. This measure initiates when the headway between vehicles falls below 1 second, triggering a simulation of maximum brake application. The synchronized standstill of both the ego and target vehicles signifies the ACC system's capability to manage abrupt deceleration and come to a halt while maintaining a predetermined distance.

4.2.2 Alternating headway values

4.2.2.1 Long to medium distance

The initial parameters are as follows:

- Initial distance between two vehicles :50m
- Target vehicle speed = $10,1 \frac{m}{s}$, controlled by PID controller, gains $K_p = 2,82$; $K_i = 3,32$; $K_d = 0,59$
- Set speed = $15 \frac{m}{s}$ (target value input = 15,1)
- Desired headway = 2,5s
- Map = Town 6, only straight roads
- PID lateral gains $K_p = K_i = K_d = 0$
- ACC PID longitudinal gains $K_p = 2,82$; $K_i = 3,32$; $K_d = 0,59$

- Both vehicles = Dodge Charger

After a steady following state is achieved, new parameters are defined:

- Desired headway = 2,0s only when $t > 200s$ to ensure a steady following state was achieved.

The results can be seen in Figure 50:

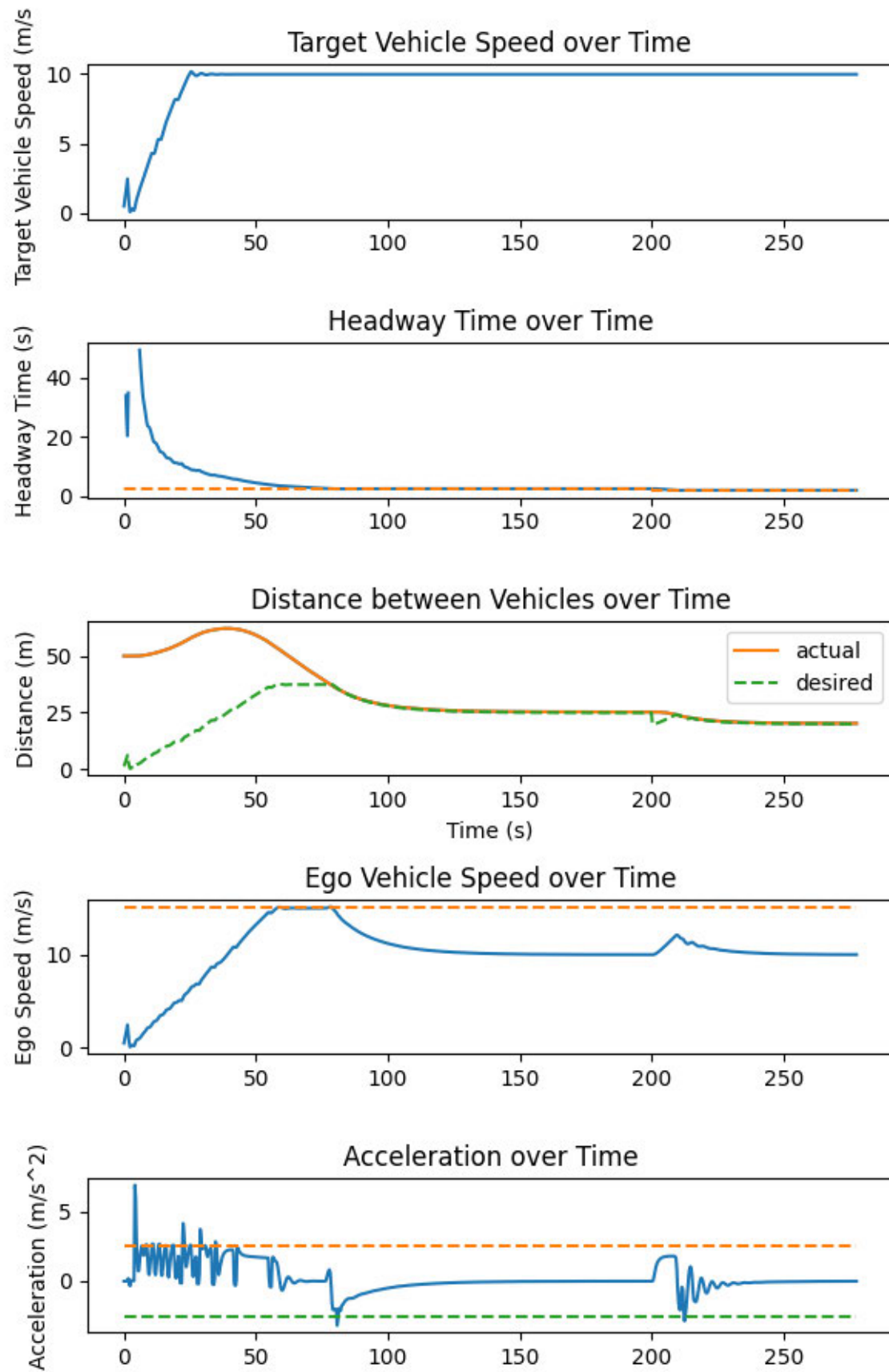


Figure 50 Result from alternating headway from long to medium distance

In the initial 60 seconds, the speed and acceleration followed a pattern similar to past simulations. However, upon reaching the target speed, a slight overshoot of approximately 0,4% occurred at $15,06 \frac{m}{s}$. Subsequently, the speed graph exhibited oscillations for the ensuing 6 seconds before stabilizing at $14,95 \frac{m}{s}$, resulting in a steady-state error of 0,3%. Just before the two distance curves converged, a similar speed bump was observed, briefly peaking at $15,1 \frac{m}{s}$ before rapidly decreasing. Following this event, the two curves ran in parallel with minimal overshooting and oscillation, both staying within a range of less than 0,1%. Around the 200-second mark, the two distance graphs merged completely, the same for the headway graphs. The ego vehicle's acceleration reached 0, indicating a consistent trailing state. Only at this point was the decision made to reduce the headway to 2,0 seconds. It's worth noting the distinct bump in the desired headway graph, as depicted in Figure 51.

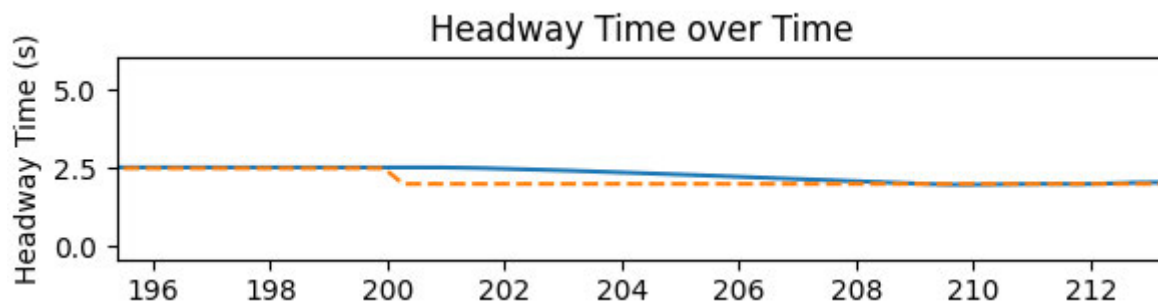


Figure 51 Headway change after 200s

Over the subsequent 10 seconds, the actual headway curve gradually aligned with the desired headway curve. Eventually, it stabilized with a slight overshoot of 3% and no oscillation. The alteration in headway also induced a noticeable bump in the desired distance curve. This adjustment prompted a swift decrease in the desired distance to 20,2 meters, triggering a corresponding shift in the actual distance curve. Approximately nine seconds later, the two distance curves converged, displaying a 3% overshoot. The oscillation during this phase was marginal, registering at 0,3 Hz, as indicated in Figure 52. Shortly thereafter, a stable equilibrium was attained, with a steady state error of 1,3%.

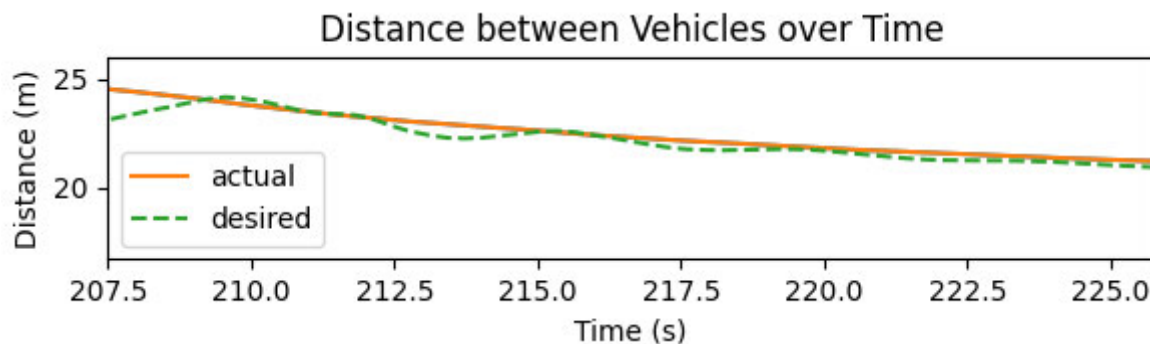


Figure 52 Distance curves after changing headway

A bump in the ego speed curve can also be observed after the headway change, signifying the ego is gaining speed to match the lower headway value. The speed was as high as $12,2 \frac{m}{s}$, then decreased with a small oscillation (0,5Hz) to slowly match the target vehicle speed again. In the acceleration curve (see Figure 53), a big initial boost is observed, following by a sharp drop into a minor oscillation (0,24Hz), exceeding the maximum deceleration value at $2,8 \frac{m}{s^2}$. The oscillation then decayed in the next 30 seconds, then the curve is settled at $0 \frac{m}{s^2}$, signifying another steady state.

Following the adjustment in headway, a notable elevation can be observed in the ego speed curve. This elevation indicates that the ego vehicle is accelerating to align with the revised lower headway value. The speed surged to a peak of $12,2 \frac{m}{s}$, subsequently decreasing with a minor oscillation of 0,5 Hz, gradually realigning with the target vehicle's speed. In the acceleration curve (refer to Figure 53), a substantial initial boost is evident, succeeded by a rapid decline into a minor oscillation (0,24 Hz), surpassing the maximum deceleration value at $2,8 \frac{m}{s^2}$. The oscillation then attenuated over the subsequent 30 seconds, with the curve eventually stabilizing at $0 \frac{m}{s^2}$, indicating another instance of steady-state behavior.

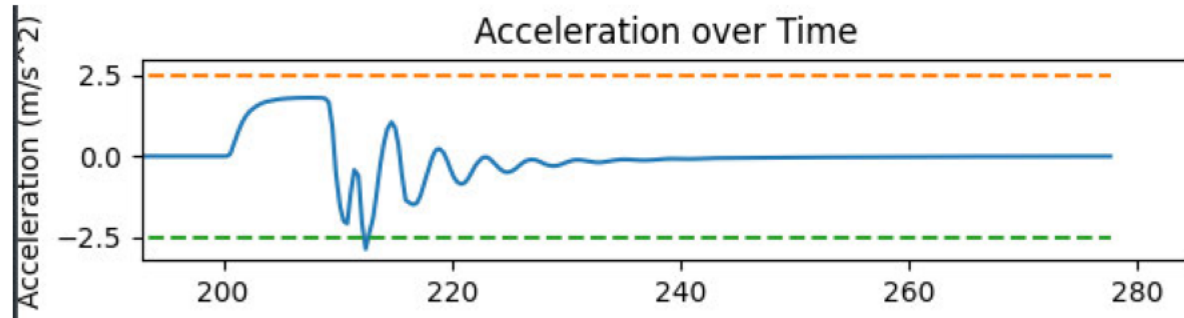


Figure 53 Acceleration curve after changing headway to 2s

The scores were given as seen in Table 8:

Aspect	Score
Steady following state	10
Desired distance/Headway keeping	8
Overshoot	7
Oscillation	8
Steady state error	8
Response	9
Smoothness in braking and throttle control	8
Total score	58/70

Table 8 Scoring of the headway alternating test from long to medium distance

In general, the system exhibited slightly more overshooting when attempting to reach the set speed compared to previous scenarios. However, aside from this, the curve behaviors in the beginning remained largely consistent with past simulations. Following the change in headway, the system responded exceptionally well to the change. It adeptly and seamlessly regulated the ego vehicle's speed to uphold the desired distance, with just one acceleration peak surpassing the comfort threshold, and very briefly. Oscillation behavior was well within acceptable limits, occurring at a very low frequency. Ultimately, the system achieved a steady state by the end of the scenario, with an extremely minor steady-state error.

4.2.2.2 Long to short distance

The initial parameters are as follows:

- Initial distance between two vehicles :50m
- Target vehicle speed = $10,1 \frac{m}{s}$, controlled by PID controller, gains $K_p = 2,82$; $K_i = 3,32$; $K_d = 0,59$
- Set speed = $15 \frac{m}{s}$ (target value input = 15,1)
- Desired headway = 2,5s

- Map = Town 6, only straight roads
- PID lateral gains $K_p = K_i = K_d = 0$
- ACC PID longitudinal gains $K_p = 2,82; K_i = 3,32; K_d = 0,59$
- Both vehicles = Dodge Charger

After a steady following state is achieved, new parameters are defined:

- Desired headway = 1,5s only when $t > 200s$ to ensure a steady following state was achieved.

The results can be seen in Figure 54:

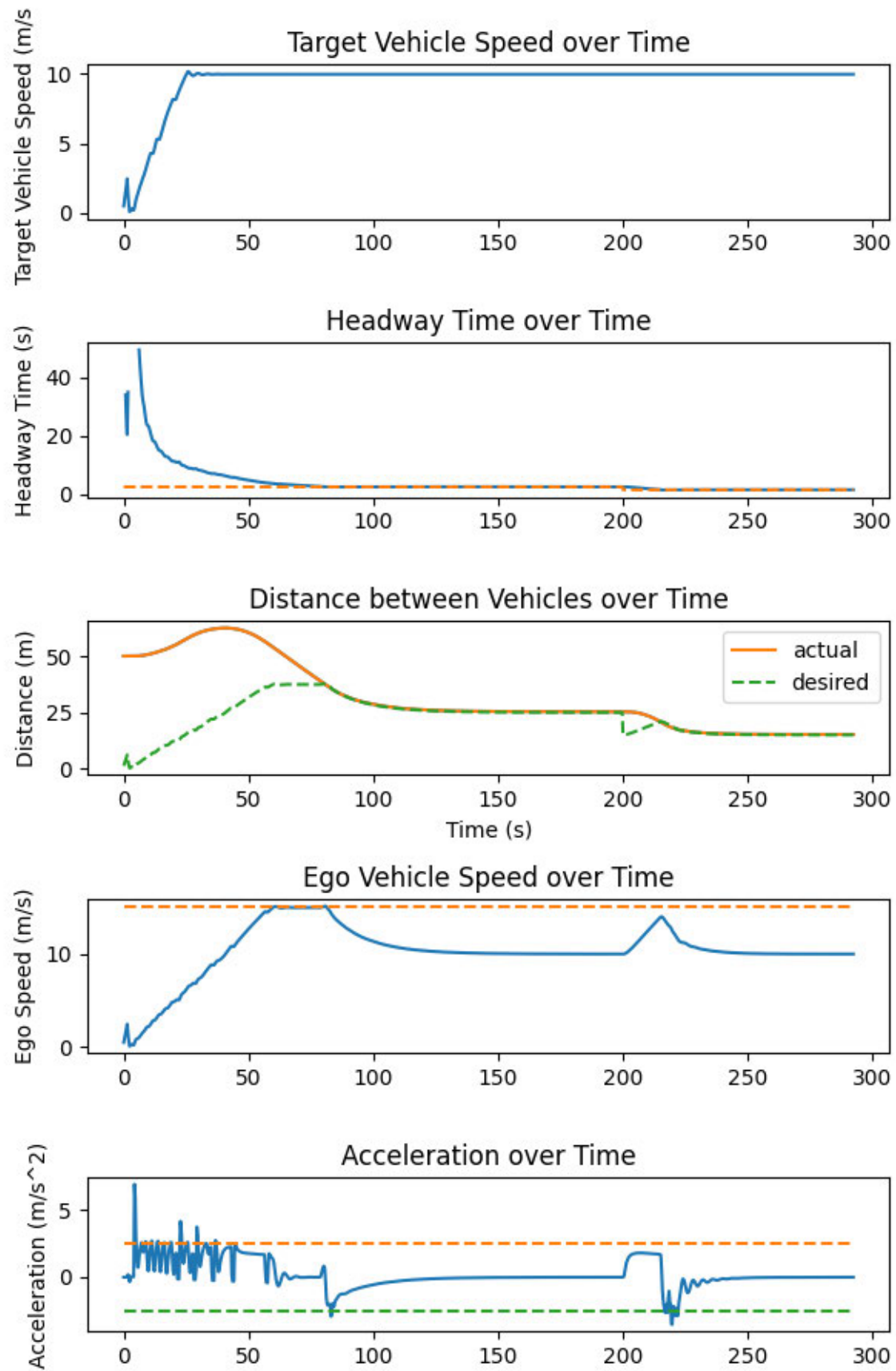


Figure 54 Result from alternating from long to short distance

Until the point of changing the headway, there isn't much difference compared to the previous scenario. Once the headway is changed to 1,5s, over the next ten seconds, the actual headway curve gradually matches the desired one. Ultimately, a steady state is achieved with minimal overshoot and very insignificant steady state error (see Figure 55) at 1,435s (4,3%) and 1,519s (1,3%) respectively. Some oscillation did occur but was very minimal.

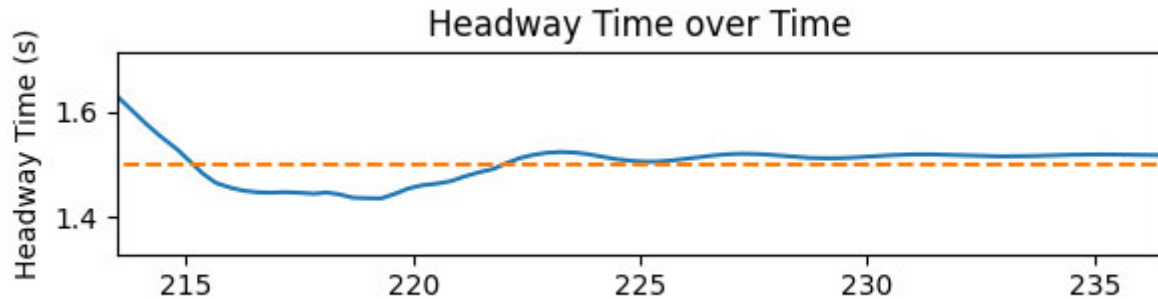


Figure 55 Headway dropped to 1,5s

Moving on to the distance curves as shown in Figure 56, it is observed that a sharper drop occurred at the 200s mark, comparing to the previous scenario. Eventually, the two curves aligned with minimal overshoot (4%), oscillation (<1 Hz), and a steady-state error of (1,2%).

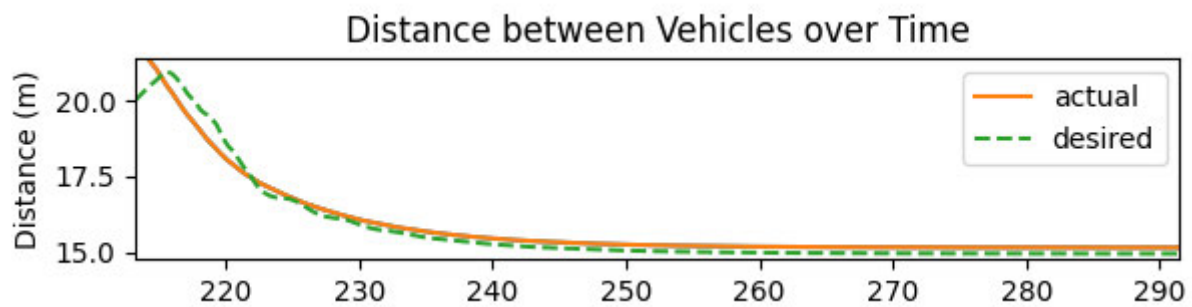


Figure 56 Distance curves behavior after headway decreased

The ego vehicle can also be observed accelerating to compensate for the change in headway, reaching a peak of $14 \frac{m}{s}$ and then gradually decreasing to match the target vehicle's speed, while displaying very minimal oscillation. Towards the end of the simulation, a steady-state error of 1% is noted.

Shifting the focus to the acceleration curve (refer to Figure 57), more intriguing patterns emerge. There's a peak that exceeds the maximum comfortable deceleration at $-3,51 \frac{m}{s^2}$. Furthermore, this exceeded deceleration lasts notably longer compared to the previous scenario, extending for approximately one second. Additionally, there are four additional peaks that exceed the maximum acceleration, though they are of shorter duration and lower amplitude compared to the main peak. Subsequently, the acceleration experiences oscillations for roughly 20 seconds before settling at $0 \frac{m}{s^2}$, indicating another stable following state.

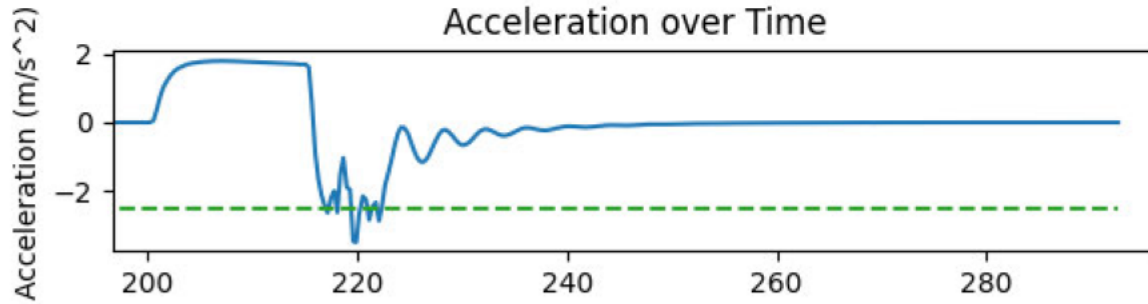


Figure 57 Acceleration curve after changing the headway to 1,5s

The scores are given as follows (see Table 9):

Aspect	Score
Steady following state	10
Desired distance/Headway keeping	7
Overshoot	7
Oscillation	8
Steady state error	8
Response	9
Smoothness in braking and throttle control	6
Total score	55/70

Table 9 Scoring of the headway alternating test from long to short distance

As anticipated, the system's instability exacerbated in response to abrupt alterations. To maintain the shorter headway, the ego vehicle needs to accelerate more swiftly. Therefore, the rates of overshooting were generally elevated, accompanied by prolonged oscillations, thereby leading to an extended timeframe required to achieve stability. Following the modification of the headway, the acceleration curve no longer presents a passenger-friendly profile, since a longer period of limit exceeding acceleration occurred. Nonetheless, despite this change, the desired distance was maintained, and the overall system's response remained satisfactory. The minor steady state error is also believed to be solved if the simulation were run for a longer period.

4.2.2.3 Short to long distance

The initial parameters are as follows:

- Initial distance between two vehicles :35m
- Target vehicle speed = $10,1 \frac{m}{s}$, controlled by PID controller, gains $K_p = 2,82$; $K_i = 3,32$; $K_d = 0,59$
- Set speed = $15 \frac{m}{s}$ (target value input = 15,1)
- Desired headway = 1,5s
- Map = Town 6, only straight roads
- PID lateral gains $K_p = K_i = K_d = 0$
- ACC PID longitudinal gains $K_p = 2,82$; $K_i = 3,32$; $K_d = 0,59$
- Both vehicles = Dodge Charger

After a steady following state is achieved, new parameters are defined:

- Desired headway = 2,5s only when $t > 200s$ to ensure a steady following state was achieved.

The results are depicted in Figure 58:

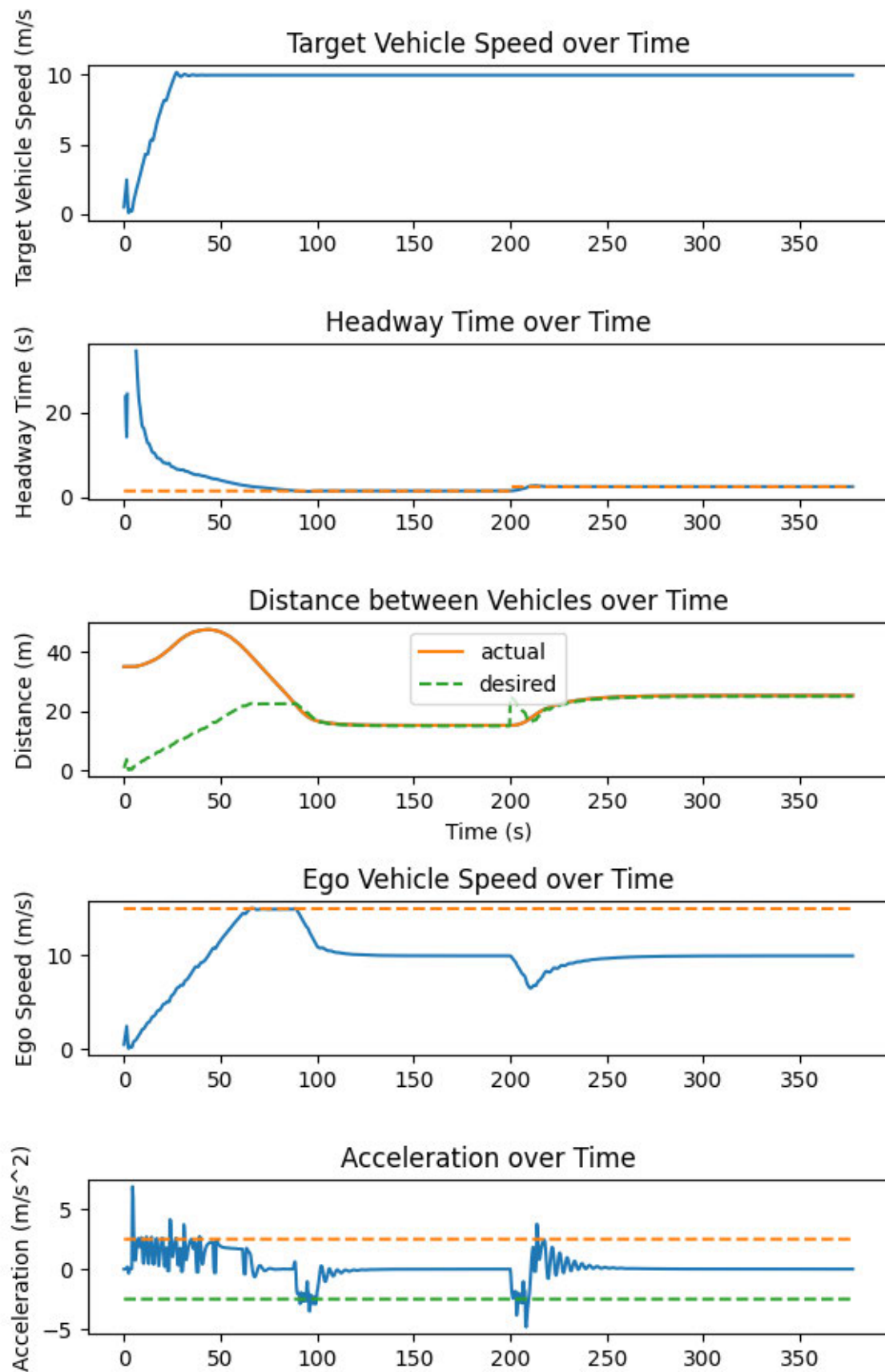


Figure 58 Result from alternating headway from short to long

There are no significant changes in the initial phase, so let's jump straight into the moment when the headway is changed, as shown in Figure 59. Upon increasing the headway, the actual headway curve gradually falls behind, only catching up after 10 seconds, with an overshoot of 2,79s (11,6%). A minor oscillation follows, and then the two curves overlap, yielding a steady-state error of 1,7%.

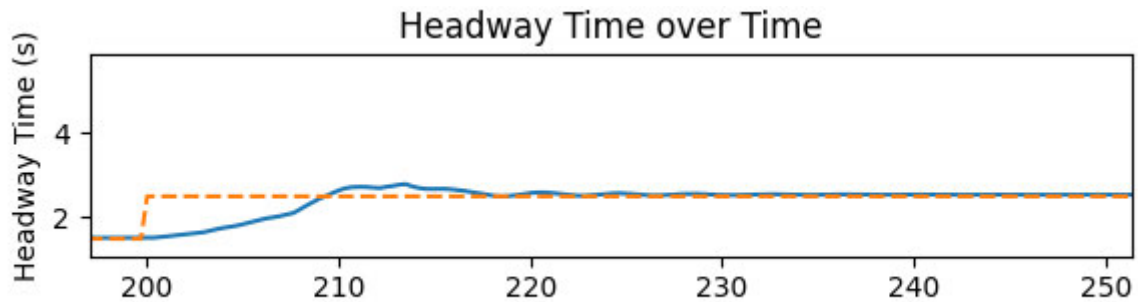


Figure 59 Headway increased to 2,5s

As seen in Figure 60, after the sharp jump in the desired distance curve, there was an overshoot of 11,5%, resulted in a distance discrepancy of 1,89m. Eventually the two curves aligned, with a steady state error of 1,4%.

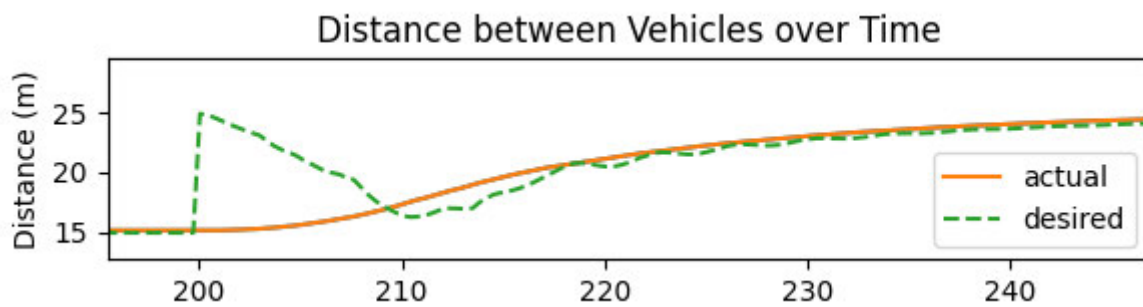


Figure 60 Distance curves behavior after headway increased

Regarding the speed curve, depicted in Figure 61, a notable decrease is evident after the headway was extended, indicating a deceleration of the ego vehicle, to $6,46 \frac{m}{s}$. Subsequently, the speed gradually rebounds, accompanied by slight oscillation (0,25Hz) throughout this transition. Ultimately, the speeds of the two vehicles become aligned.

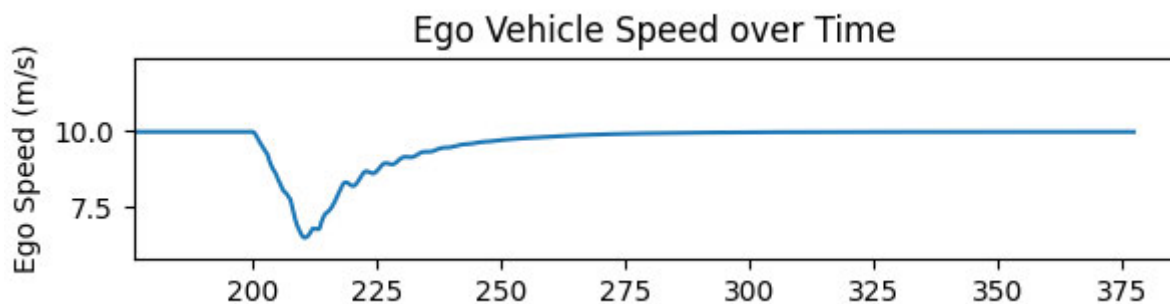


Figure 61 Speed curve after the headway increase

Progressing to the acceleration curve, illustrated in Figure 62, there are instances of surpassing the limits. Notably, during deceleration, there are brief intervals ($<1s$) where deceleration drops to $-3,76$ and $-4,82 \frac{m}{s^2}$. Immediately thereafter, when accelerating at $3,74 \frac{m}{s^2}$, the acceleration surpasses the

limit once more. Oscillations ensue and gradually diminish, ultimately leading to a stable state of the system.

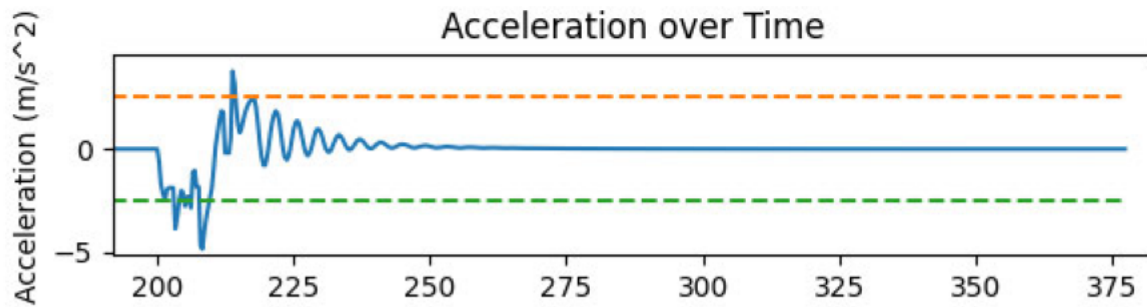


Figure 62 Acceleration curve after the headway increase

Scores were given as shown in Table 10

Aspect	Score
Steady following state	10
Desired distance/Headway keeping	6
Overshoot	6
Oscillation	7
Steady state error	9
Response	9
Smoothness in braking and throttle control	6
Total score	53/70

Table 10 Scoring of the headway alternating test from short to long distance

In general, the system exhibited a satisfactory response to the alteration in the headway value. While there were instances of slight greater overshooting, prolonged oscillations, and acceleration exceeding, notable overshooting occurred mainly in the headway value and distance. This isn't as critical as it might appear, given that the actual distance surpassed the desired value. Slight deviations of a few meters less than the desired distance are acceptable; safety concerns only arise when the actual distance falls below the desired threshold.

4.2.3 Target vehicle decelerating

The initial parameters are as follows:

- Initial distance between two vehicles :35m
- Target vehicle speed = $10,1 \frac{m}{s}$, controlled by PID controller, gains $K_p = 2,82$; $K_i = 3,32$; $K_d = 0,59$
- Set speed = $15 \frac{m}{s}$ (target value input = 15,1)
- Desired headway = 2,0s
- Map = Town 6, only straight roads
- PID lateral gains $K_p = K_i = K_d = 0$
- ACC PID longitudinal gains $K_p = 2,82$; $K_i = 3,32$; $K_d = 0,59$
- Both vehicles = Dodge Charger

After a steady following state is achieved, new parameters are defined:

- Target vehicle speed = $5,1 \frac{m}{s}$, controlled by PID controller, gains $K_p = 2,82$; $K_i = 3,32$; $K_d = 0,59$ only when $t > 120s$

The result can be seen in Figure 71:

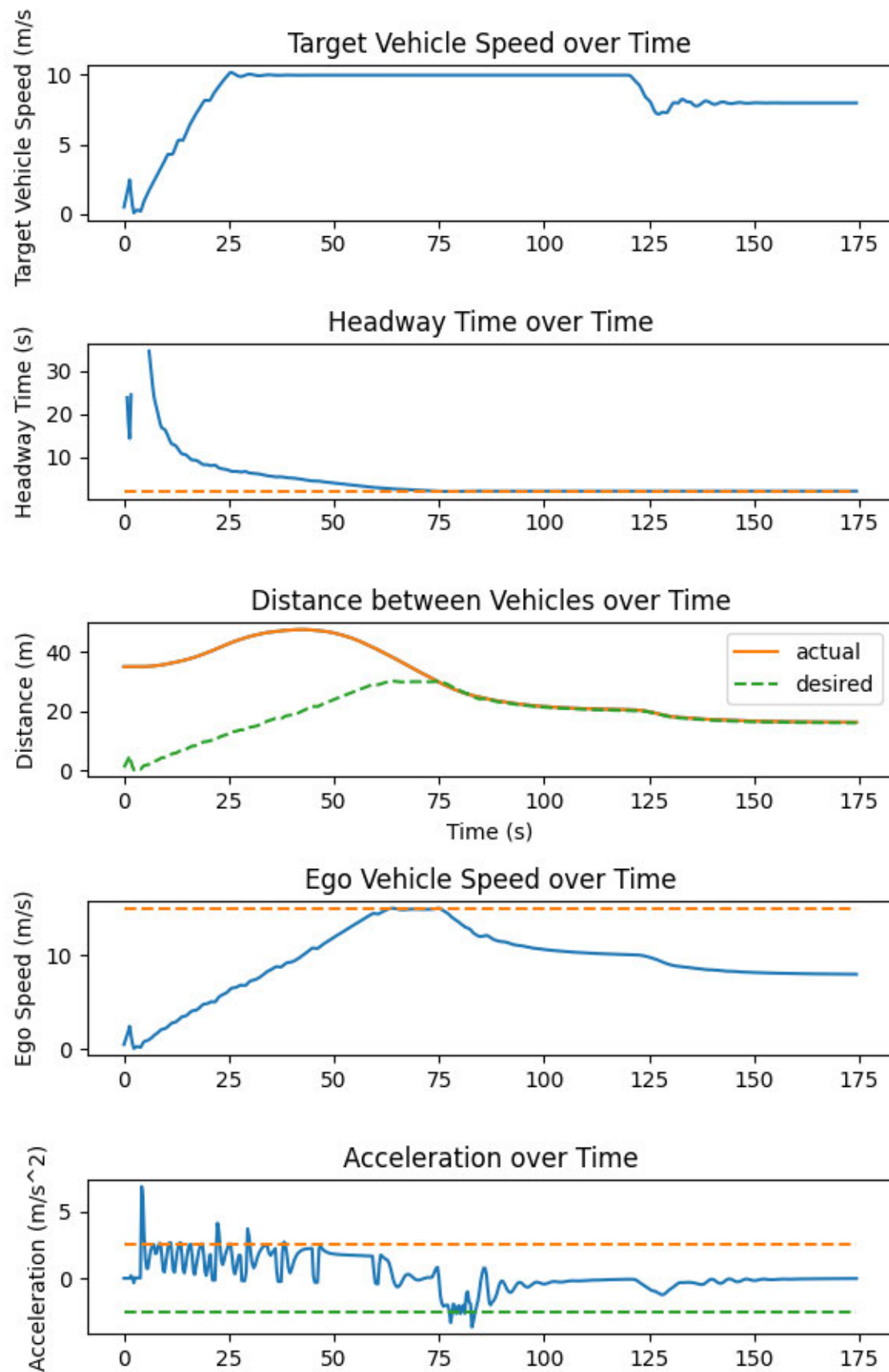


Figure 63 Result of the target vehicle decelerating from steady following state test

The speed curve of the target vehicle demonstrates that our PID controller's performance worsens at lower speeds, with a much higher rate of overshoot and a longer period of oscillation. However, our primary focus remains on the ego vehicle's response. So, let's shift our attention to the headway curve following the target vehicle's deceleration (see Figure 64). The two curves largely mirror each other over time. No overshoot is observed, only a minor steady-state error of 1,5%, accompanied by a period of very insignificant oscillation.

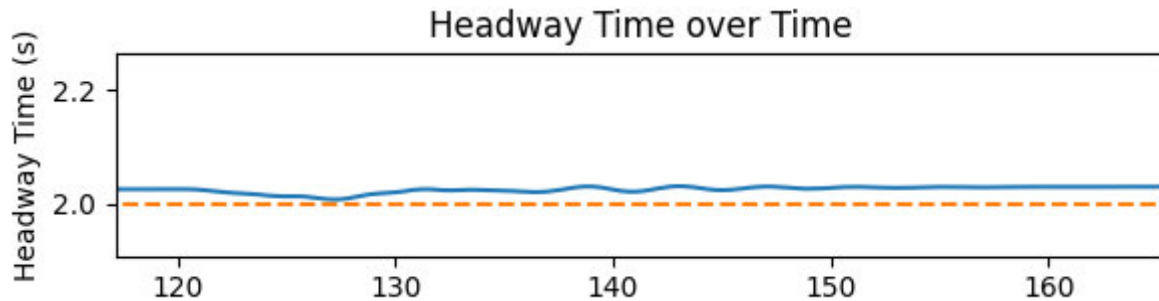


Figure 64 Headway curve after target vehicle decelerated

Past the 120-second mark, the two distance curves remained almost perfectly aligned despite the target vehicle's sudden change in speed, as illustrated in Figure 65. There was no overshoot observed, only a steady-state error of 1,3%. This pattern persisted until the conclusion of the simulation.

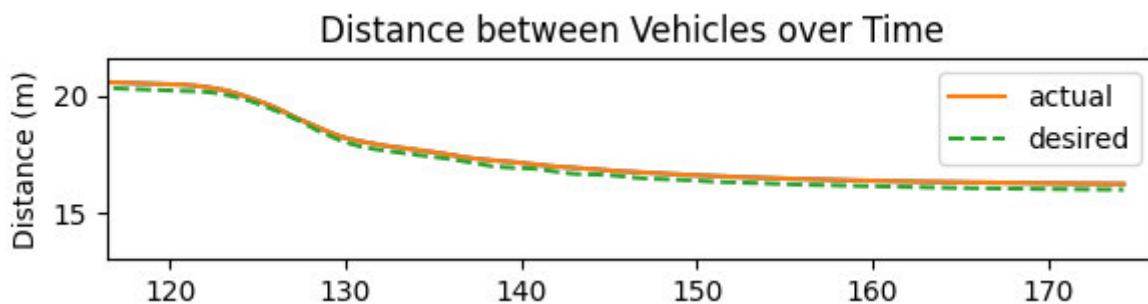


Figure 65 Distance curves after the target vehicle decelerated

The ego's speed curve after the deceleration was exceptionally smooth as seen in Figure 66. The ego responded and decelerated seamlessly, reducing its speed from 10,1 to $8,1 \frac{m}{s}$ by the end of the simulation, resulting in a steady state error of 1,3%. However, it's expected that the error will decrease as the simulation runs for a longer period.

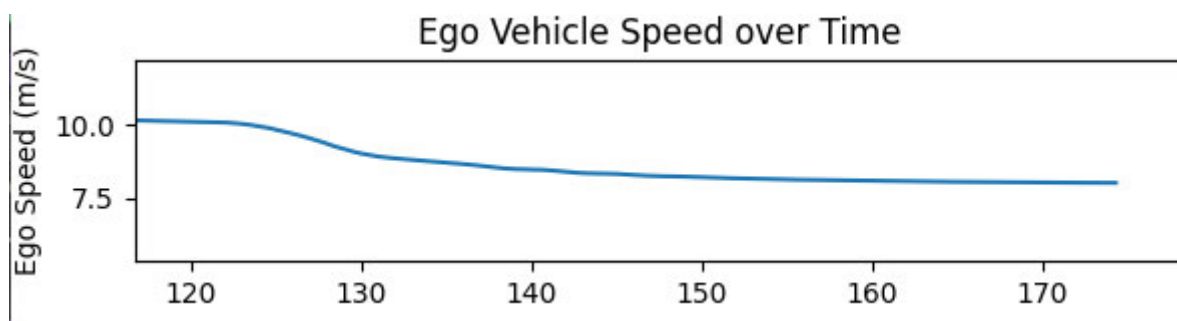


Figure 66 Speed curve after target vehicle decelerated

The acceleration curve is equally characterized by its smoothness (Figure 67). There are no peaks that exceed the maximum comfort value. While there were minimal oscillations, they dissipated swiftly, leading to a steady state of $0 \frac{m}{s^2}$.

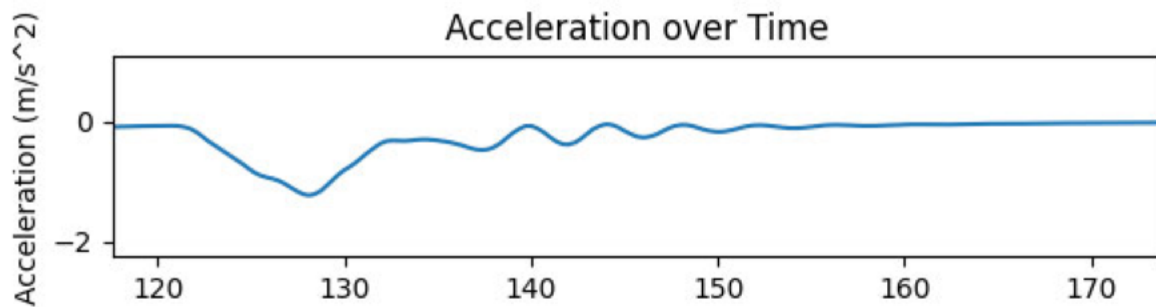


Figure 67 Acceleration curve after the target vehicle decelerated

The scores were given as follows in Table 11:

Aspect	Score
Steady following state	10
Desired distance/Headway keeping	9
Overshoot	10
Oscillation	9
Steady state error	9
Response	9
Smoothness in braking and throttle control	9
Total score	65/70

Table 11 Scoring of the target vehicle decelerating test

Overall, the performance was remarkable. The response was timely, maintaining the desired distance with no overshoot, minimal oscillation, and steady-state error, even in the face of a relatively abrupt speed change in the target vehicle. It's important to recognize that this success is due to the modest reduction in the target vehicle's speed, which was only by $2 \frac{m}{s}$. It's reasonable to anticipate that the performance could deteriorate with more substantial speed changes.

4.2.4 Target vehicle accelerating

The initial parameters are as follows:

- Initial distance between two vehicles :35m
- Target vehicle speed = $10,1 \frac{m}{s}$, controlled by PID controller, gains $K_p = 2,82$; $K_i = 3,32$; $K_d = 0,59$
- Set speed = $15 \frac{m}{s}$ (target value input = 15,1)
- Desired headway = 2,0s
- Map = Town 6, only straight roads
- PID lateral gains $K_p = K_i = K_d = 0$
- ACC PID longitudinal gains $K_p = 2,82$; $K_i = 3,32$; $K_d = 0,59$
- Both vehicles = Dodge Charger

After a steady following state is achieved, new parameters are defined:

- Target vehicle speed = $15,1 \frac{m}{s}$, controlled by PID controller, gains $K_p = 2,82$; $K_i = 3,32$; $K_d = 0,59$ only when $t > 120s$

The results are shown in Figure 68 below:

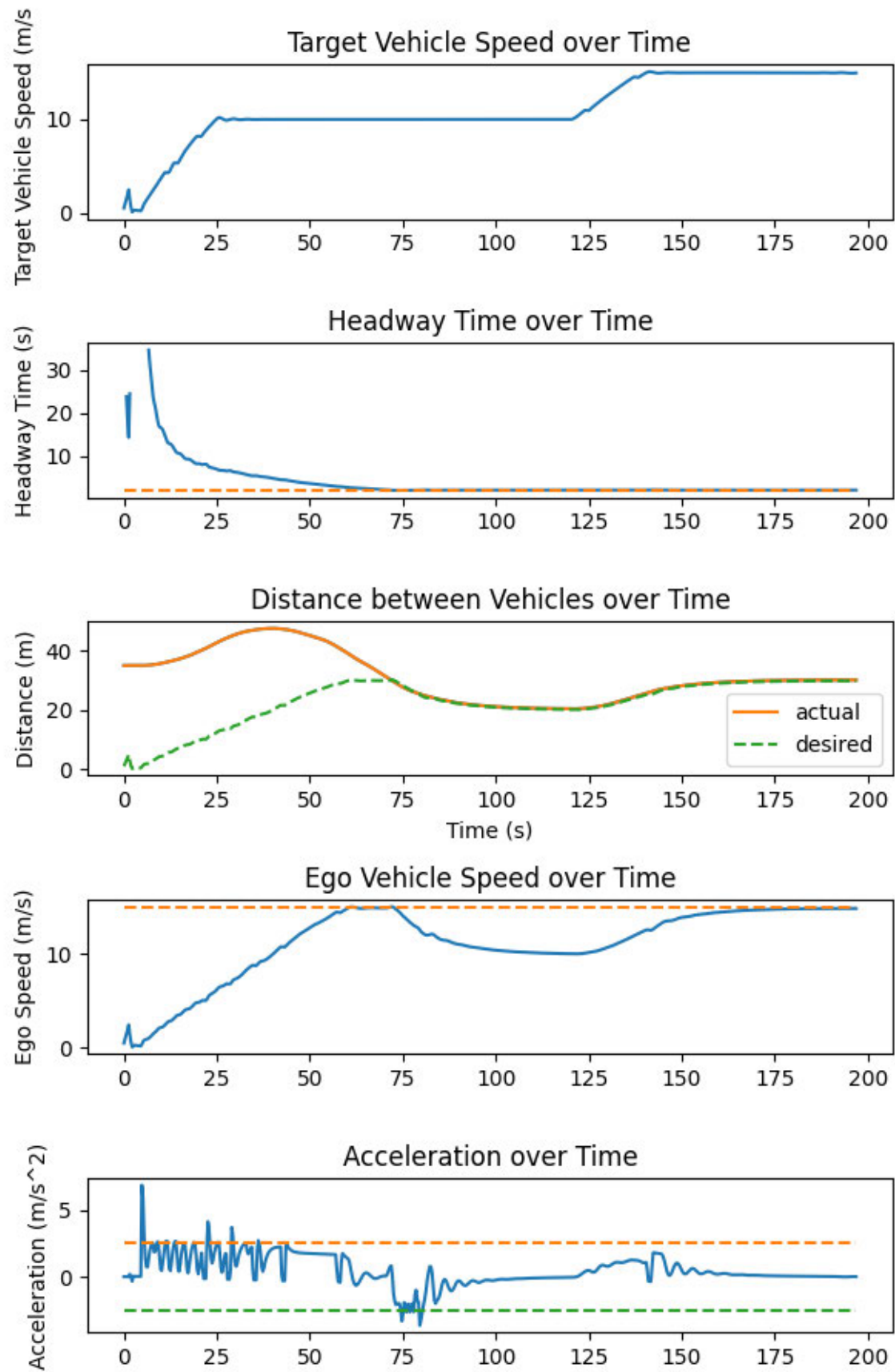


Figure 68 Result of the target vehicle accelerating from steady following state test

Until the point of convergence of the two distance curves, there isn't much difference compared to other scenarios. The ego managed to reach the predetermined set speed with a minor overshoot, minimal oscillations, as well as an insignificant steady-state error of 0,4%. However, the stable state of traveling at the set speed didn't last very long, as only 12 seconds later, the ego was commanded to slow down to maintain the desired distance. It's observable once more that right before decelerating, a small speed bump at $15,1 \frac{m}{s}$ was reached. Ego's velocity continued to decline in order to maintain the desired distance.

Directing our focus beyond the 120-second mark, it becomes apparent that in response to the target vehicle accelerating to $15 \frac{m}{s}$, the ego vehicle adjusted its speed accordingly by increasing. In the headway diagram (see Figure 69), there is a tiny bump at the 142-second mark, at 2,1 s. This happened shortly after the target vehicle overshoot in speed. Otherwise, there is only a minor steady-state error of 1,1%.

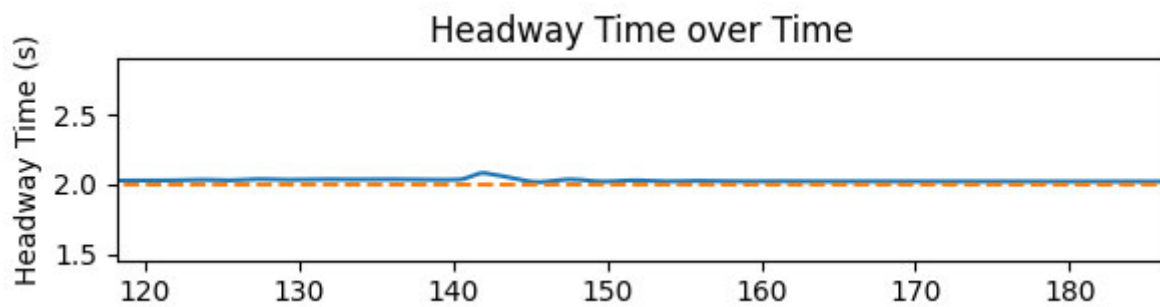


Figure 69 Headway bump after target vehicle finished accelerating

In relation to the distance curves, the two trajectories closely mirrored each other quite effectively subsequent to the target vehicle's acceleration (refer to Figure 70). There was merely one minor overshoot, occurring at the 142-second mark, measuring at 4,3%. Although oscillation did transpire, its duration was very brief, followed by the attainment of a steady state with an inconsequential steady-state error of 1,1%.

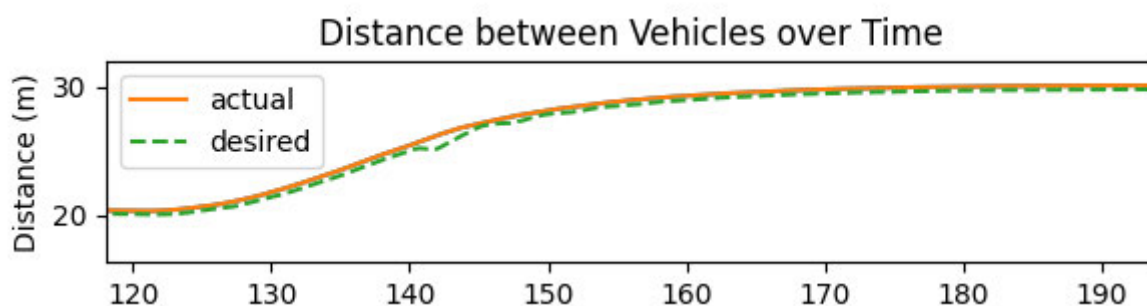


Figure 70 Distance curves after the acceleration of target vehicle

Regarding the speed curve, the ego vehicle adeptly accelerated and smoothly aligned with the target vehicle's updated speed, as evidenced in both Figure 71 and Figure 72. There was a brief period of steadiness, succeeded by a minor episode of oscillation attributed to gear shifting. However, there was no overshoot observed, and a mere small steady-state error of 1,3% was present, which is expected to diminish over time. In terms of acceleration, there were no instances of exceeding, only some minor oscillations followed by settling into a steady state of $0 \frac{m}{s^2}$.

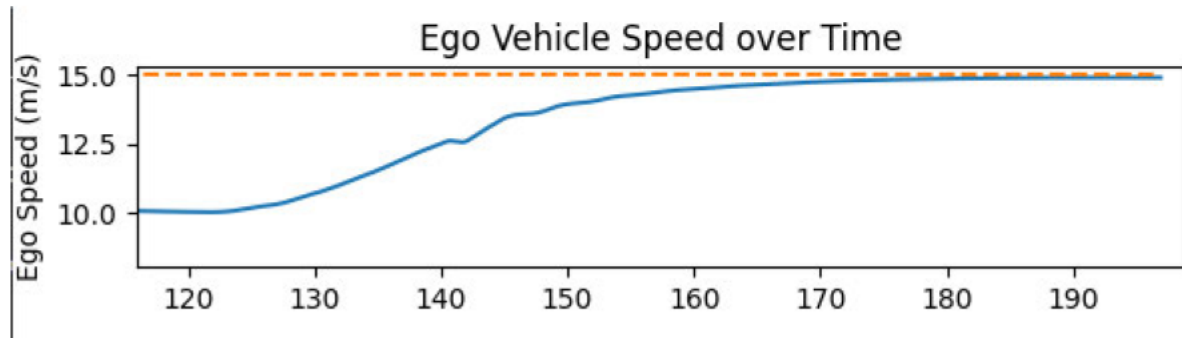


Figure 71 Speed curve after target vehicle accelerated

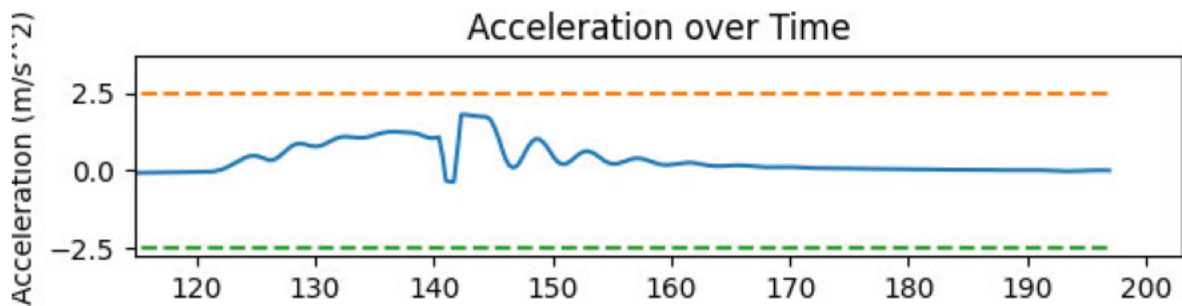


Figure 72 Acceleration curve after target vehicle accelerated

The scores were given as shown in

Aspect	Score
Steady following state	10
Desired distance/Headway keeping	9
Overshoot	8
Oscillation	9
Steady state error	9
Response	9
Smoothness in braking and throttle control	9
Total score	63/70

Figure 73 Scoring of the target vehicle accelerating test

In general, this can be regarded as the most favorable system response observed thus far. It effectively accomplishes all tasks with minimal overshoot, oscillation, and no instances of acceleration exceeding. Nevertheless, there remains some uncertainty about the slight acceleration of the ego vehicle just before the convergence of the two distance curves.

4.2.5 Target vehicle speeding off

The initial parameters are as follows:

- Initial distance between two vehicles :35m
- Target vehicle speed = $10,1 \frac{m}{s}$, controlled by PID controller, gains $K_p = 2,82$; $K_i = 3,32$; $K_d = 0,59$
- Set speed = $15 \frac{m}{s}$ (target value input = 15,1)
- Desired headway = 2,0s
- Map = Town 6, only straight roads
- PID lateral gains $K_p = K_i = K_d = 0$

- ACC PID longitudinal gains $K_p = 2,82$; $K_i = 3,32$; $K_d = 0,59$
- Both vehicles = Dodge Charger

After a steady following state is achieved, new parameters are defined:

- Target vehicle speed = $30,1 \frac{m}{s}$, controlled by PID controller, gains $K_p = 2,82$; $K_i = 3,32$; $K_d = 0,59$ only when $t > 120s$

The results are shown in Figure 74:

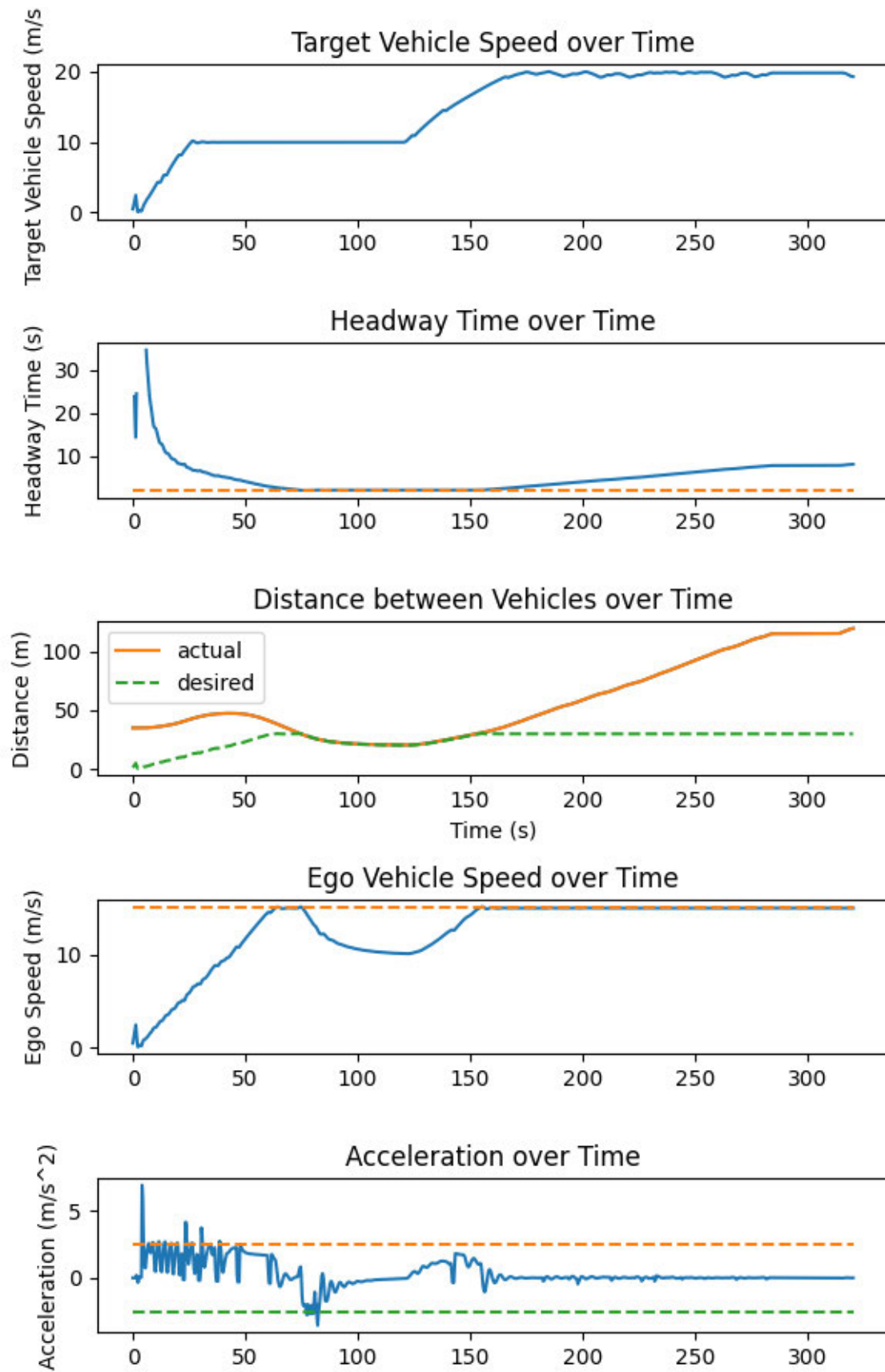


Figure 74 Result of the target vehicle speeding off test

When examining the headway curve, as shown in Figure 75, a slight overshoot of 2% is noticeable the first time the distance between the two vehicles equals the desired distance. This is succeeded by a minor oscillation, followed by a steady-state error of 1,6%. This condition endured for around 60 seconds, after which the curve started to rise and continued its upward trend over time.

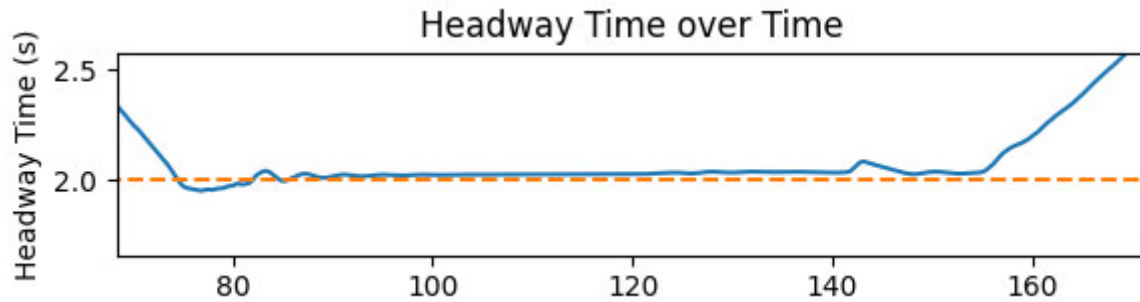


Figure 75 Headway curve before and after target vehicle speeding off

Shifting our focus to the distance curves (Figure 76), it becomes evident that the initial mirroring behavior is quite satisfactory, characterized by minimal overshoot and oscillation. However, once the ego vehicle attains the set speed, the ACC system prevents any acceleration, causing the two curves to diverge. While the desired distance curve remains constant, the actual distance continues to increase.

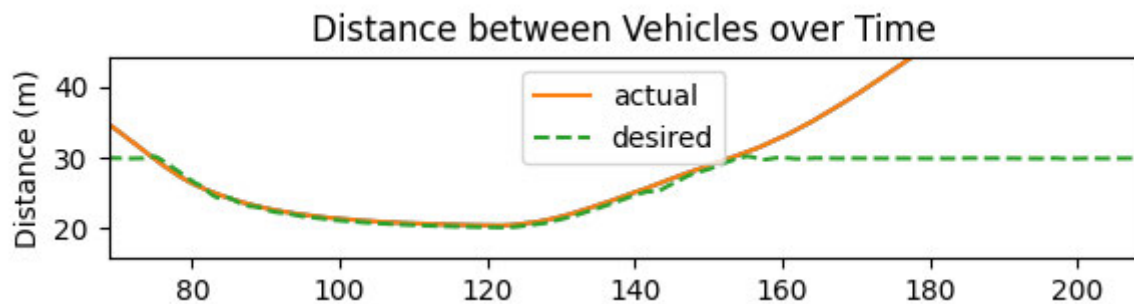


Figure 76 Distance curves before and after target vehicle speeding off

Figure 77 presents the exact moment when the set speed is reestablished, approximately at the 154-second mark. This instance saw a modest overshoot at 15,1 m/s, measuring around 0,6%. Following this, a brief period of oscillation occurred, leading to a steady-state error of 0,3%.

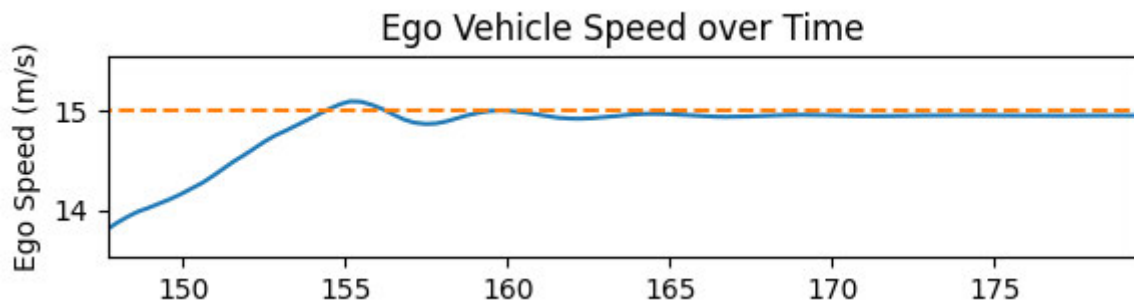


Figure 77 Speed curve after set speed is reached but target vehicle keeps accelerating

Examining the acceleration curve (Figure 78), it's notable that there were no instances of exceeding the comfort limit once the target vehicle accelerated. However, there were several minor fluctuations throughout the process. Ultimately, a steady state was achieved by the end.

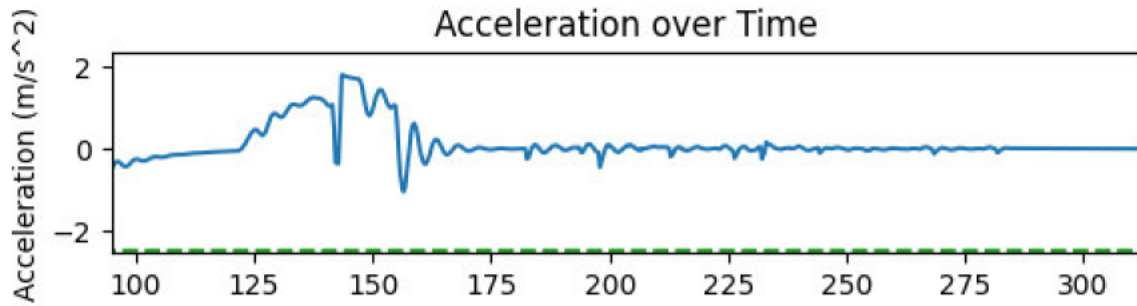


Figure 78 Acceleration curve during and after target vehicle speeding off

The scores were given as shown in Table 12:

Aspect	Score
Steady following state	10
Desired distance/Headway keeping	9
Overshoot	8
Oscillation	8
Steady state error	9
Response	9
Smoothness in braking and throttle control	9
Total score	63/70

Table 12 Scoring of the target vehicle speeding off test

Overall, the ACC system demonstrates impressive performance, adeptly managing speed changes. The desired distance is effectively maintained with negligible overshoot and oscillation. Furthermore, successful prevention of acceleration when the desired distance becomes equal to or smaller than the actual distance between the two vehicles is evident. The subsequent maintenance of the set speed also proves successful, with only minor overshooting and oscillation. In all cases, the steady-state error is exceedingly minor and can be disregarded.

4.2 Executing advanced scenarios and results

The initial parameters are as follows:

- Initial distance between two vehicles :10m
- Target vehicle speed = autopilot
- Set speed = $15 \frac{m}{s}$ (target value input = 15,1)
- Desired headway = 2,0s
- Map=Town 2, mentioned in Table 1
- PID lateral gains $K_p = 3$; $K_i = K_d = 0$
- PID longitudinal gains $K_p = 2,82$; $K_i = 3,32$; $K_d = 0,59$
- Ego = Dodge Charger, target vehicle = Tesla Model 3
- Maximum steering value = 100%

Employing the random seed number 1002, both vehicles are positioned directly ahead of a three-way intersection. The target vehicle consistently executes right turns (as determined through multiple iterations). On the other hand, the ego vehicle's objectives encompass preserving the desired distance and aligning its steering with the trajectory of the target vehicle. Results can be seen in Figure 79:

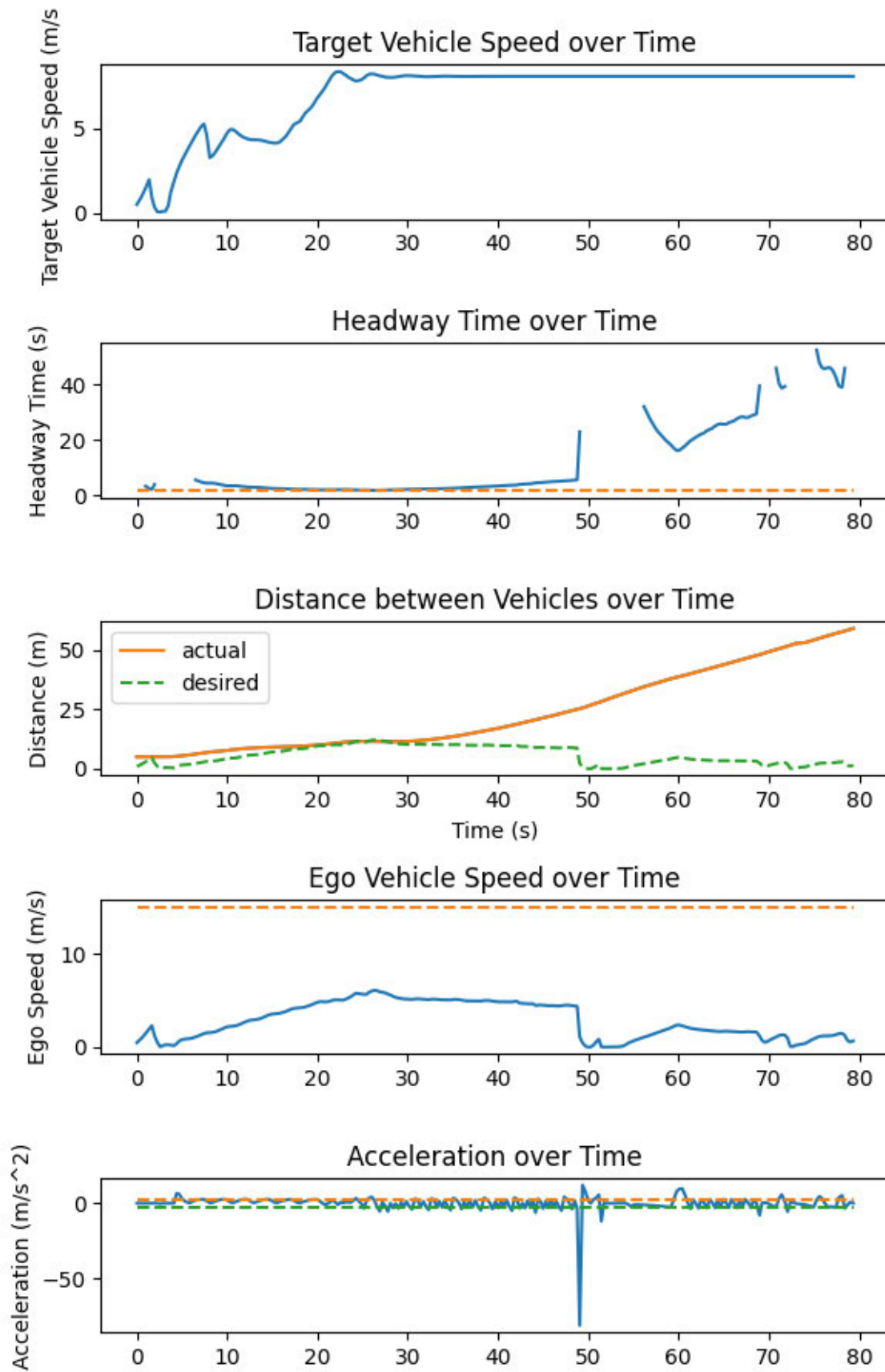


Figure 79 Result of the following target vehicle in curve test

The overall behavior of the curve displays unevenness. The headway was maintained fleetingly; however, when the ego vehicle attempted to emulate the target vehicle into the curve, it unintentionally navigated onto the sidewalk, evident around the 49-second mark. There's a likelihood that some form of obstruction is impeding its complete steering range, despite configuring the maximum steering allowance at 100%. The resultant curve traced by the ego's steering was notably broad, causing an instance of oversteering and contact with the sidewalk. This test can be marked as unsuccessful, and there can be various factors contributing to this outcome. It's plausible that there's an incorrect configuration that is inhibiting the ego's steering capability, particularly evident when applying realistic physics to the vehicle. Even attempting to raise the K_p value does not appear to yield any changes. The ego vehicle still veers onto the sidewalk, regardless of whether the value is set above 3,0.

5 Discussion

In the quest for enhanced realism, the integration of lidar data and the introduction of noise and environmental disturbances aimed to simulate the intricate interplay of laser rays with real-world environmental factors. However, this endeavor, while more realistic, introduced complexity that consumed substantial time. In hindsight, a more straightforward approach from the outset might have been preferable. One prominent challenge that emerged was the optimization of code, which lagged notably once distance, speed, and headway calculations were added. Seeking ways to bolster simulation performance without compromising accuracy becomes paramount. Expanding the PID controller configuration presents an avenue for advanced scenario testing. Incorporating features like anti-windup and automatic parameter tuning, both for longitudinal and lateral control, can enable the ACC system to adapt more dynamically to a broader range of driving conditions. Configuring vehicle-specific parameters, such as torque curves, becomes a delicate balancing act. Any missteps in this realm can potentially disrupt the vehicle's gear-changing behavior, underscoring the need for meticulous attention. The painstaking process of parameter definition, often stemming from manual iterations, calls for a more systematic approach to ensure consistency and efficiency in test setup. Leveraging tools like Scenic¹⁰ streamlines scenario testing, eliminating the need for extensive code rewrites for each variation. This simplifies the iterative process and enhances realism. Fine-tuning parameters specific to different speed ranges becomes crucial to eliminate instabilities, particularly at lower speeds. When evaluating results, comparing them with a pre-established baseline, such as the Euro NCAP test, is recommended. The opacity surrounding certain parameters within the CARLA simulation presents challenges, as it's unclear which values are in use without explicit configuration. This lack of transparency can affect the system's behavior and results, as well as our understanding in evaluation of conducted tests. Furthermore, the mysterious disappearance of diagrams post-simulation adds an additional layer of complexity, extending testing timelines.

In the realm of ACC controllers, a critical avenue for exploration involves a comprehensive comparison between the performance of the devised ACC controller approach and the existing array of ACC controllers. This comparative assessment should encompass a meticulous evaluation across dimensions of safety, adaptability, and efficiency. By scrutinizing how the designed ACC controller fares against its counterparts in real-world scenarios, a more nuanced understanding of its strengths and areas for refinement can emerge. Moving forward, an in-depth analysis of its ability to handle dynamic environments, adapt to diverse driving conditions, and maintain a high degree of vehicle and occupant safety is paramount.

¹⁰ is a domain-specific probabilistic programming language for modeling the environments of cyber-physical systems like robots and autonomous cars [71]

Amidst the pursuit of advancing autonomous driving technologies, it's essential to acknowledge the inherent limitations that arise within the study's simulation environment. The CARLA simulation framework, while a powerful tool for testing and refining autonomous systems, is not without its constraints. One notable limitation lies in the complexity of replicating the vast and intricate dynamics of real-world traffic scenarios within the simulation. While CARLA does an admirable job of emulating various driving conditions, it is challenging to capture every nuanced interaction, unforeseen road conditions, and the unpredictable behavior of other road users that a real-world setting presents. This limitation can potentially impact the comprehensive evaluation of the ACC controller's performance, as it might not fully encompass the entire spectrum of driving challenges that vehicles encounter in actual traffic environments. Moreover, CARLA's reliance on predefined maps and environments might restrict the simulation's adaptability to new or rapidly evolving road layouts. The extent to which these predefined environments accurately reflect the complexities of diverse real-world road networks can be variable. Additionally, discrepancies between the simulated environment and the real-world terrain could introduce biases or inaccuracies that affect the evaluation of the ACC controller's effectiveness. Another limitation stems from the simplified physics and sensor models employed in CARLA. While these models serve the purpose of providing a simulated representation, they might not encompass the full intricacies of real-world physics and sensor behaviors. This discrepancy could potentially impact the accuracy of the results, particularly when evaluating the ACC controller's responses to complex and dynamic scenarios.

While the current research represents a significant advancement in ACC controller design, charting future directions for refinement and innovation remains imperative. Potential trajectories for enhancing the ACC controller approach could include a deeper investigation into parameter tuning strategies, exploring the integration of machine learning techniques for adaptive control, and refining the lateral control aspects of the PID controller. Additionally, the application of sensor fusion to enhance accuracy and robustness in varied conditions presents a promising avenue. As the field of autonomous driving and vehicle control continues to evolve, potential avenues for further research could encompass real-world testing and validation to bridge the gap between simulation and practical implementation.

6 Conclusion

In conclusion, the implementation and evaluation of the ACC system with a PID controller in the CARLA simulation environment showcased promising outcomes in basic scenario tests. The ACC-PID combination effectively achieved its fundamental objectives, maintaining desired distances, exhibiting smooth speed transitions, and minimizing oscillations, scoring around 84% across all tests (excluding the failed experiment for advanced scenario). However, as the system ventured into more advanced scenarios, the need for comprehensive tuning and parameter configuration became evident. Fine-tuning, especially in intricate areas like vehicle physics, proved challenging due to the limited accessibility to essential parameters within CARLA. This enigma around parameter access reflected a situation of utilizing a tool without a complete understanding of its inner workings. Despite these challenges, this project offers an engrossing exploration into the realm of autonomous vehicle control, with ample opportunities for enhancement.

Looking ahead, this paper holds a pivotal role in shaping the trajectory of autonomy in the automotive industry. By addressing the intricacies of ACC controller design and real-world simulation, it bridges a crucial gap in understanding the dynamic interplay between control algorithms and complex driving scenarios. As the industry moves toward greater autonomy, the insights gleaned from this study provide valuable steppingstones for refining ACC systems and enhancing their adaptability in diverse environments. Moreover, the challenges encountered underscore the necessity of comprehensive

parameter access for accurate simulation, driving further dialogue on the development of simulation tools that mirror real-world dynamics. In this context, this paper assumes the role of a catalyst, contributing to the ongoing conversation on autonomy's evolution and fostering innovation in the pursuit of safer and more efficient autonomous vehicles.

Works Cited

- [1] R. C. D. G. L. C. Adam Ziebinskia. [Online]. Available: https://www.researchgate.net/publication/321364551_Review_of_advanced_driver_assistance_systems_ADAS.
- [2] [Online]. Available: https://de.wikipedia.org/wiki/Abstandsregeltempomat#/media/Datei:Schema_ICC.svg.
- [3] [Online]. Available: <https://www.sae.org/news/2019/01/sae-updates-j3016-automated-driving-graphic>.
- [4] F. Z. a. Z. L. Hong Tana, "Impact of adaptive cruise control (ACC) system on fatality and injury reduction in China," 4/2021.
- [5] S. S. Monfort. [Online]. Available: <https://www.tandfonline.com/doi/abs/10.1080/15389588.2021.2013476>. [Accessed 2022].
- [6] [Online]. Available: <https://www.skyradar.com/blog/radar-principle>.
- [7] [Online]. Available: <https://ieeexplore.ieee.org/document/1701081>.
- [8] [Online]. Available: <https://en.wikipedia.org/wiki/Radar>.
- [9] [Online]. Available: <https://commons.wikimedia.org/wiki/File:LIDAR-scanned-SICK-LMS-animation.gif>.
- [10] [Online]. Available: <https://www.elprocus.com/the-working-of-a-pid-controller/>.
- [11] [Online]. Available: <https://www.smlease.com/entries/automation/what-is-pid-proportional-integral-derivative-control/>.
- [12] [Online]. Available: <https://www.incatools.com/pid-tuning/pid-tuning-methods/#:~:text=The%20trial%2Dand%2Derror%20method%20is%20a%20relatively%20easy%20method,tweaks%20to%20improve%20the%20response..>
- [13] R. C. M. S. S. D. G. Khaled Alomari. [Online]. Available: <https://www.scitepress.org/Papers/2020/101751/101751.pdf>.
- [14] N. S. H. & A.-h. S. A. Mostafa Al-Gabalawy. [Online]. Available: <https://link.springer.com/article/10.1007/s40435-020-00732-w>.
- [15] P. S. Merry Cherian. [Online]. Available: https://www.researchgate.net/publication/258651214_Neural_Network_based_ACC_for_Optimized_safety_and_comfort.
- [16] W. YIN. [Online]. Available: <https://kth.diva-portal.org/smash/get/diva2:1375828/FULLTEXT01.pdf>.
- [17] A. D. a. G. R. a. F. C. a. A. L. a. V. Koltun, "carla.org," 2017. [Online]. Available: <http://proceedings.mlr.press/v78/dosovitskiy17a/dosovitskiy17a.pdf>.

- [18] [Online]. Available: https://www.researchgate.net/figure/Visualization-of-data-transformation-from-the-CARLA-environment-to-KITTI-format_fig1_359228060.
- [19] [Online]. Available: https://carla.readthedocs.io/en/latest/ref_sensors/#lidar-sensor.
- [20] [Online]. Available: https://carla.readthedocs.io/en/latest/img/lidar_point_cloud.jpg.
- [21] [Online]. Available: https://carla.readthedocs.io/en/latest/tuto_G_retrieve_data/#lidar-raycast-sensor.
- [22] [Online]. Available: https://carla.readthedocs.io/en/latest/python_api/#carla.VehiclePhysicsControl.
- [23] [Online]. Available: https://carla.readthedocs.io/en/latest/catalogue_vehicles/#dodge-police-charger.
- [24] [Online]. Available: https://www.automobile-catalog.com/auta_details1.php#gsc.tab=0.
- [25] [Online]. Available: <https://carla.readthedocs.io/en/latest/catalogue/#maps>.
- [26] S. Kim, "Design of the Adaptive Cruise Control Systems: An Optimal Control Approach," p. 4, 2012.
- [27] K. K. A. K. K. F. H. F. S. Tokoro, "Electronically scanned millimeter-wave radar for pre-crash safety and adaptive cruise control system," in *Intelligent Vehicles Symposium, 2003. Proceedings. IEEE*, 2003.
- [28] J. S. F. I. Shunji Miyahara, "Radar-based Target Tracking Method: Application to Real Road," pp. 421-430, 2005.
- [29] M. B. ., M. M. Rainer Möbus, "Multi-object adaptive cruise control," p. 359–374, 2003.
- [30] H.-J. K. K. Y. Seungwuk Moon, "Multi-vehicle target selection for adaptive cruise control," pp. 1325-1343, 2010.
- [31] G. D. Y.-w. J. Tao Ren, "ABR traffic control over ATM network using fuzzy immune-PID controller," in *American Control Conference*, 2006.
- [32] L. L. H. W. H. L. Lilong Gong, "Adaptive Cruise Control Design Based on Fuzzy-PID," in *International Conference*, 2010.
- [33] [Online]. Available: https://en.wikipedia.org/wiki/Euclidean_distance.
- [34] [Online]. Available: https://carla.readthedocs.io/en/latest/tuto_G_bounding_boxes/#bounding-boxes_1.
- [35] [Online]. Available: <https://www.meshlab.net/>.
- [36] [Online]. Available: [https://en.wikipedia.org/wiki/Euler_angles#:~:text=Euler%20angles%20can%20be%20defined%20by%20intrinsic%20rotations,0%2Dz%20\(initial\)](https://en.wikipedia.org/wiki/Euler_angles#:~:text=Euler%20angles%20can%20be%20defined%20by%20intrinsic%20rotations,0%2Dz%20(initial)).

- [37] K. Vogel, "A comparison of headway and time to collision as safety indicators," *Accident Analysis and Prevention* 35, p. 427–433, 2003.
- [38] F. C. L. W. O. D. Paul G. Michael, "Headway on urban streets: observational data and an intervention to decrease tailgating," *Transportation Research Part F* 3, pp. 55-64, 2000.
- [39] B. v. D. C. J. G. & V. R. Van Arem, "The Impact of Cooperative Adaptive Cruise Control on Traffic-Flow Characteristics," *IEEE TRANSACTIONS ON INTELLIGENT TRANSPORTATION SYSTEMS*, vol. 7, no. 4, pp. 429-436, 2006.
- [40] [Online]. Available: https://github.com/pirate-lofy/CARLA-PID-controller/blob/main/carla_controller.py.
- [41] [Online]. Available: https://en.wikipedia.org/wiki/PID_controller.
- [42] [Online]. Available: <https://www.electricalibrary.com/en/2020/09/01/classical-control-theoretical-introduction/>.
- [43] [Online]. Available: <https://docs.wpilib.org/en/stable/docs/software/advanced-controls/introduction/introduction-to-pid.html>.
- [44] [Online]. Available: <https://carla.readthedocs.io/en/latest/foundations/#rendering>.
- [45] [Online]. Available: https://carla.readthedocs.io/en/latest/core_map/#layered-maps.
- [46] [Online]. Available: <https://www.python.org/downloads/release/python-370/>.
- [47] [Online]. Available: https://carla.readthedocs.io/en/latest/start_quickstart/.
- [48] [Online]. Available: <https://www.meshlab.net/#download>.
- [49] [Online]. Available: <https://pypi.org/project/pip/>.
- [50] [Online]. Available: https://carla.readthedocs.io/en/latest/core_sensors/#listening.
- [51] [Online]. Available: https://carla.readthedocs.io/en/latest/core_actors/.
- [52] [Online]. Available: https://en.wikipedia.org/wiki/Ziegler%E2%80%93Nichols_method.
- [53] [Online]. Available: <https://shawnmklee.wordpress.com/2017/03/31/pid-controller-and-ziegler-nichols-method-how-to-get-oscillation-period/>.
- [54] [Online]. Available: https://carla.readthedocs.io/en/latest/adv_traffic_manager/#configuring-autopilot-behavior.
- [55] [Online]. Available: https://carla.readthedocs.io/en/latest/start_quickstart/.
- [56] [Online]. Available: https://msc.berkeley.edu/old4/past_research/automotive/acc.html.
- [57] [Online]. Available: https://carla.readthedocs.io/en/latest/catalogue_vehicles/#tesla-model-3.
- [58] [Online]. Available: https://carla.readthedocs.io/en/latest/catalogue_vehicles/#audi-tt.

- [59] [Online]. Available: <https://cdn.euroncap.com/media/58813/euro-ncap-ad-test-and-assessment-protocol-v10.pdf>.
- [60] [Online]. Available: https://carla.readthedocs.io/en/latest/map_town06/.
- [61] Z. Y. E. C. J. F. C. Wang, "Space headway calculation and analysis at turn movement trajectories using hybrid model," pp. 2669-2679.
- [62] H. Deza, "Study on Vehicle Forward Collision Avoidance System," 2004.
- [63] M. A. K. H. E.-S. Sumbal Malika, "CARLA: Car Learning to Act — An Inside Out," p. 745, 2021.
- [64] [Online]. Available: <https://routetogermany.com/drivingin germany/city-driving#tailgating>.
- [65] [Online]. Available: https://en.wikipedia.org/wiki/Bilinear_transform.
- [66] [Online]. Available: <https://de.wikipedia.org/wiki/PT1-Glied>.
- [67] [Online]. Available: https://www.google.com/search?q=laplace+transformation+table&oq=laplace+transformation+table&gs_lcrp=EgZjaHJvbWUyBggAEEUYOTIICA EQABgWGB4yCAgCEAAYFhgeMggIAx AAGBYYHjIICAQQABgWGB4yCAgFEAAYFhgeMggIBhAAGBYHjIICAcQABgWGB4yCAgIEAAYFhgeMggICRAAGBYHtIBCDUzMDRqMG.
- [68] [Online]. Available: <https://www.youtube.com/watch?v=mLYHrTdFfZU>.
- [69] [Online]. Available: https://en.wikipedia.org/wiki/Final_value_theorem.
- [70] [Online]. Available: https://en.wikipedia.org/wiki/Convolution_theorem.
- [71] [Online]. Available: <https://rocketloop.de/en/blog/creating-scenarios-carla-scenic/#:~:text=What%20is%20Scenic%3F,like%20robots%20and%20autonomous%20cars..>

The use of generative AI in this paper

In addition to Python with the CARLA client and Unreal Engine 4, the project also leveraged the assistance of ChatGPT as a valuable tool. Its primary role was in refining sentence structures and aiding in code troubleshooting when needed. It's important to clarify that the generative AI was solely used to rephrase existing sentences crafted by the author. On the coding front, ChatGPT provided initial insights through basic prompts, albeit often requiring multiple iterations of troubleshooting before yielding effective results. Ultimately, ChatGPT emerged as a potent asset, offering substantial power when harnessed adeptly, despite the iterative process it often entails.

Final words

Embarking on this subject was a departure from my comfort zone, as I ventured into the intricacies of adaptive cruise control and autonomous driving technology, all to be meticulously replicated within the CARLA environment using Python. While initially daunting, the journey turned out to be an exhilarating learning experience. Tackling challenges that had no predefined solutions was both exciting and enlightening. This endeavor encouraged me to think creatively, experiment with different approaches, and step beyond the boundaries of conventional problem-solving. The absence of a fixed roadmap required me to rely on resourcefulness, tapping into various tools and insights to navigate uncharted territories. Ultimately, this venture not only expanded my knowledge but also instilled in me the value of embracing challenges as opportunities for growth.

