PG iTraffic with TurtleBots: Preliminary Project Report

Debkowski, Julia
Eilers, Nellson
Gerber, Stefan
Grave, Malte
Heckelmann, Lasse
Kowalska, Paulina
Marken, Marie
Monenschein, Jan-Magnus
Schneiders, Carl
Struck, Simon
Wojciak, Filip

January 19, 2024

Contents

This documentation serves as the second submission, providing information of a project still in development. It has been refined to its current state, representing the progress achieved thus far. Further iterations and enhancements to this documentation are expected and planned, but for now, this submission stands as a reflection of our ongoing efforts.

Chapter 1

Product Vision

The goal of the project group iTraffic with TurtleBots is to develop autonomous driving functions with TurtleBots to solve scenarios of varying complexity based on a German Autobahn. To assure that these driving functions meet their specifications, test-based validation will be used.

The project group will use the TurtleBot platform as a basis. This makes it possible to experimentally validate autonomous driving functions in different simulated traffic scenarios with the use of mostly low-cost sensor technology. The goal is to present and tackle the challenges of autonomous driving in a way that is cost-effective and low-risk.

In order to achieve this, a platform for creating autonomous driving functions based on the TurtleBot will be developed. It will enable members of the project group as well as future developers to implement controllers for vehicles on different autonomy levels and simulating those vehicles on a TurtleBot. This platform is called TurtleCar.

Additionally, TurtleCar will provide capabilities to validate the controllers in a simulated as well as a real life environment. For this, a Domain Specific Language (DSL) to define test cases for the controllers will be developed. The simulation suite „Gazebo“ will be used for testing in a simulated environment. Using the simulation, it will be possible to automatically execute test cases, gather the results and determine whether the test conditions where met. TurtleCar will ensure that both environments behave similarly with regard to the inputs and outputs of the controller.

Using the TurtleCar platform, several scenarios of various complexities will be developed and provided for the creation of test cases for the testbed. This will enable a developing cycle that is closely related to the DevOps method: The development of the testbed follows the requirements posed by the scenarios, and can be adjusted as needed.

1.1 Autonomous Driving Functions

The scenarios developed as part of this project group will all be based on a highway with the properties of a German Autobahn. Controllers with three different levels of autonomy will be built:

- no autonomy

- partially automated
- highly automated

The functions will be implemented using suitable, robust control strategies. They will be based on the current state of the art in the control engineering domain.

1.2 Functions of the Non-Autonomous Vehicle

The non-autonomous vehicle has no assistance systems. In the non-autonomous vehicle, a human driver controls the vehicle completely. They can define the speed and the steering angle, and the bot moves according to the vehicle's dynamics.

Functions of the Partially Automated Vehicle The partially automated vehicle can perform certain functions autonomously within bounded conditions. It may call for the driver's intervention if needed.

Lane Keeping Assistant The driver will determine the speed of the vehicle. As long as the Lane Keeping Assistant is activated, the vehicle will keep to the center of its current lane without the need of the driver to control the steering angle.

Adaptive Cruise Control The speed of the vehicle will be partially determined by the driver. When Adaptive Cruise Control is activated, and another, slower vehicle is driving in the front, the speed will be adjusted so that a safety margin will be kept.

Lane Changing When the Lane Changing function is engaged, if conditions permit, the vehicle will execute lane changes, while maintaining appropriate spacing from neighboring vehicles.

Collision Avoidance System The vehicle will avoid static obstacles like road works by changing lanes or stopping safely before the obstacle until a safe lane changing is possible.

Overtaking The vehicle will avoid obstacles moving in the same lane, like a slower car ahead, by changing lanes or reducing speed until a safe lane changing is possible.

Functions of the Highly Automated Vehicle The highly automated vehicle is able to drive on the highway without needing intervention from the driver. All actions will be self-initiated. Using only the aforementioned driving functions, it will move the vehicle forward as safely as possible without any input from the driver while adhering to the German traffic regulations in terms of safety margins. Also, it will adhere to speed limits and „no overtaking“ road signs.

Malicious Agent Avoidance The vehicle will avoid collisions with cars which are moving in defiance of traffic rules by choosing a safe driving strategy.‘

Platooning In platooning mode, the vehicle will join a closely coordinated group of vehicles traveling in a convoy-like formation. The system will automatically control the vehicle's speed, following distance, and positioning within the platoon. The platooning system will continuously communicate with other vehicles in the group, ensuring safe and efficient travel.

Chapter 2

State of the Art

This section introduces the autonomy levels according to the Society of Automotive Engineers (SAE), shows the group's ongoing research on driving functions, and outlines past projects working on similar topics.

2.1 Levels of Automotive Autonomy

The SAE defines levels of autonomy in on-road automated driving vehicles. The SAE standard J3016_202104 [47] outlines the six levels of driving automation, ranging from Level 0 (no automation) to Level 5 (full automation) depicted in Table 2.1 as follows.

Table 2.1: Levels of driving automation according to the SAE standard J3016_202104 [47]

Level	Description
Level 0	No Driving Automation
Level 1	Driver Assistance
Level 2	Partial Driving Automation
Level 3	Conditional Driving Automation
Level 4	High Driving Automation
Level 5	Full Driving Automation

While level 1–2 use „driver support“ features, level 3–5 use „automated driving“ features. The level of driving automation of a vehicle is determined by a combination of factors: the extent of required human involvement in driving tasks, the vehicle's capability to perform driving functions, and the operational design domain under which a feature is designed to function (i. e. environmental restrictions). The standard also differentiates between three types of actors: the (human) user, the driving automation system, and other vehicle systems and components.

Because of this, systems that provide alerts about driving hazards are excluded from this classification as they neither automate driving tasks nor change the driver's role in performing them. Additionally, the lane keeping assistant, the electronic stability control or other certain types of driver assistance systems are not covered by this driving automation classification. This is because

it provides momentary intervention rather than sustained automation of driving tasks.

2.2 Assisted Driving Functions

The following contains the initial research done before implementing the driving functionalities.

2.2.1 Lane Departure Warning System [30]

The Lane Departure Warning System is a feature designed to alert the driver when their vehicle unintentionally drifts from its lane without using a turn signal. There are different types of such a system:

- Lane departure warning (LDW)
- Lane keeping assist (LKA)
- Lane centering assist (LCA)
- Automated lane keeping systems (ALKS).

While the LDW only warns the driver, the LKA ensures that the vehicle stays in its lane. Furthermore, the LKA makes sure that the car stays centered in its lane. The ALKS is a combination of LKA and ACC.

There are several vehicles in which a LDWS is integrated dating back to 2001. Generally, they are based on video sensors mounted behind the windshield, laser sensors and infrared sensors.

The LDW observes the TurtleBot's movements and its position within the lane. It can recognize the TurtleBot leaving its lane without using a turn signal and gives an alert. The LDW can be implemented with the LIDAR Sensor by orienting the lanes along a wall and / or with the camera

When the lanes can be perceived, the TurtleBot leaving its lane or starting to leave its lane has to be recognized. This can be achieved by observing the displacement of the TurtleBot in its lane. To later control the TurtleBot to stay in its lane, the direction in which the TurtleBot deflects should be identified as well.

2.3 Related Projects

In the past, there were several projects from the Carl von Ossietzky University of Oldenburg who dealt with implementing driving functions on hardware representing a vehicle. In the following, these will be described and distinguished from the project group.

„Realtime Controlled Cooperative Autonomous Racing System“ (RCCARS) has undertaken the task to develop a safety-critical system using the racetrack Mini-Z Grand Prix Circuit 30 and RC-Cars from Kyosho. This system is responsible for observing and controlling autonomously operating vehicles on a racetrack. In their „collision-free“ scenario, a single car is supposed to autonomously complete five laps on the racetrack at a minimum average speed of 1.5 m/s without colliding with the track's boundaries. [9]

„Realtime Controlled Cooperative Autonomous Racing System Next Generation“ (RCCARSng) builds upon the work of RCCARS. It extends the project by adding a second car and several static obstacles. Both cars are supposed to complete a minimum of ten collision-free laps. During this, both vehicles have the opportunity to overtake each other and should avoid obstacles while doing so. This group divides their scenario „collision-free overtaking“ in three variants:

- One vehicle following the other.
- One vehicle overtaking the other.
- Following and overtaking while avoiding obstacles. [7]

RCCARS and RCCARSng both use global knowledge and external calculations. A camera situated above the racetrack perceives the track and the vehicles on the track. There exists an external component responsible for location determination and for controlling the vehicles. For the overtaking function, they use a preceding trajectory calculation implemented in Matlab.

„Emergency Braking Assistant for fully Autonomous Cars“ (EmBrAAC) has undertaken the task to develop a real-time vehicle assistant. Depending on the situation, it should be capable of calculating an evasive strategy or performing emergency braking. They use a remotely-controlled vehicle from Traxxas in combination with a predefined and self-build course. Their focus lies on real-time capabilities and contract-based design. [17]

Within the context of the university course „Forschendes Lernen - Mobiles Multiagenten-Robotersystem“ eight students investigated and practically implemented method-oriented topics in the field of mobile robotic systems using a TurtleBot. They familiarized themselves with the simulation software Gazebo and used it to validate initial prototypes before transferring them into real hardware. After doing some fundamental work with the TurtleBot and Gazebo software, the students were split into two groups.

One group focused on using Simulink to address the question „How can an autonomous driving function for obstacle avoidance be developed?“. As part of this, they developed control algorithms that enable the robot to follow the desired path, navigate around obstacles, and perform precise navigation.

The other group, using Python, explored the question „How is realistic driving behavior simulated?“. In doing so, they researched vehicle models and implemented a suitable one. This included considering factors such as friction, inertia, road conditions, and other physical properties.

During the course, Simulink and Python were compared for the implementation of driving functions on a TurtleBot. The course was meant as a preliminary project for the „iTraffic with TurtleBots“ project group. The project group adopted the vehicle model and knowledge about the differences between reality and Gazebo simulation.

The project group „iTraffic with TurtleBots“ enables the utilization and implementation of driving functions on a TurtleBot based on local knowledge. The implemented functions use a camera und a LIDAR sensor on the TurtleBot. These sensors can be combined freely. The environment in which the TurtleBot operates and the TurtleBot itself closely resembles reality: The TurtleBot is located on a three-lane highway and behaves like a specific car. The goal is to

develop a modular development platform. That means vehicle models, environments and driving functions can be added and are interchangeable. Alongside the creation of the development platform, an automated testing platform is created. This allows experimentally validating the driving functions.

Chapter 3

Reflecting Reality

The driving functions to be developed and the used environment should be realistic to allow accurate emulation of vehicles. To develop a testing framework based on the German highway, a representative environment needs to be created. This environment has to be defined in a way that makes it usable in reality and in a Gazebo simulation. The definition is done by scaling down the real environment to TurtleBot dimensions, which is described in detail in this section. Furthermore, the safety measures employed should follow those used on a German highway.

3.1 Terminology

Before discussing the topic of scaling down the real highway model to an environment that is usable by the TurtleBot, some terminology is defined.

Environment The actual scaled down environment used by TurtleCar in reality, or in Gazebo. When prefixed with 'real', specifically the real environment made of paper and cardboard is referenced. Respectively, the 'simulated environment' targets the Gazebo environment.

Road Model The actual, real road parameters that are the origin of the scaled down environment, i.e. the German highway.

Scaling Factor The factor by which a road model distance unit is scaled down to the environment distance. i. e. , if the real road specifies a width of 2 m, and the scaling factor is 0.25, the distance would be 0.5 m in the scaled environment.

3.2 Technical Parameters of Subjects Involved in the Environment

In this section, the technical parameters used to create the scaled down environment are described. These parameters are referenced in later sections, where their values are used to construct the scaled down environment. The involved subjects are the TurtleBot 3 Burger model, a Golf VIII car model, and a standard German autobahn.

3.2.1 TurtleBot 3 Burger Model

To emulate a car on the TurtleBot the technical details of the TurtleBot model are needed. Even though a TurtleBot 3 Specifications Guide exists, the technical parameters were determined empirically, to make the comparison between reality and specifications possible. The experimental data is depicted in Table 3.1. The data from the specifications is depicted in Table 3.2. Please note that this information does not apply when using the TurtleBot in the Gazebo environment.

The value column corresponds with the value which is provided through the */cmd_vel* topic. Anything below or above the Min/Max values will be ignored by the TurtleBot. Position/Speed accuracy for the odometer was not collected because the */odom* topic already returns a covariance matrix which corresponds with the accuracy of the measurement.

Table 3.1: Experimentally determined TurtleBot 3 Burger parameters

Parameter	**Value**	**Notes**
Min. Velocity (inclusive)	0.01	
Max. Velocity (exclusive)	0.22	
Min. Velocity Increment	0.01	
Min. Turn Velocity (inclusive)	0.01	Very irregular speed, almost a stutter
Min. Turn Velocity (reliable)	0.1	
Max. Turn Velocity (exclusive)	2.64	
Min. Turn Velocity Increment	0.01	Not 100% certain

The specification defines the following technical parameters for the TurtleBot 3 Burger model (see [41]):

Table 3.2: TurtleBot 3 parameters from specification

Parameter	**Value**
Max. Velocity	$0.22\,^{\mathrm{m}}/_{\mathrm{s}}$
Max. Turn Velocity	$2.84\,^{\mathrm{rad}}/_{\mathrm{s}}$ ($162.72\,^{\mathrm{deg}}/_{\mathrm{s}}$)
Size (Length, Width, Height)	138 mm, 178 mm, 192 mm

The minimal speed increment of 0.01 m/s poses a problem for the velocity calculations inside the Transposer component of TurtleCar Core. The problem is that the transposer is calculating a new TurtleBot velocity every time step to simulate a correct acceleration. If the calculated velocity for a given time step is smaller than 0.01 m/s then the TurtleBot doesn't change its velocity for the current time step. The current velocity v_k is used to calculate the velocity for the next time step v_{k+1} using the formula:

$$v_{k+1} = v_k + a_k * T$$

Where a_k is the current acceleration and T is the length of the time step. For a constant a_k and T, this would conclude that the TurtleBot would not increase in speed. To counteract this issue, the transposer is saving the remaining

velocity part v_r which is not handled by the TurtleBot until the next time step and adds this part to the velocity calculation.

$$v_{k+1} = v_k + v_r + a_k * T$$

This ensures that after enough time steps the TurtleBot will reach a velcoity greater than 0.01 m/s.

3.2.2 Golf VIII

When scaling the actual environment down to the representative model, the x-axis is adjusted using a scaling factor derived from the size comparison between the TurtleBot and a real VW Golf VIII. This car model has the specifications as depicted in table Table 3.3 (see [1]):

Table 3.3: Golf VIII parameters

Parameter	**Value**
Max. Velocity	69,44 $^{m}/_{s}$, 250 $^{km}/_{h}$ respectively
Size (Length, Width, Height)	4287 mm, 1789 mm, 1478 mm

3.2.3 Road Model

The used road model is based on a standard German highway. In general, the measurements given in the table Table 3.4 are used (see [46]):

Table 3.4: German highway dimensions

Parameter	**Value**
Lane Width	2.75 m - 3.75 m
Dash Mark Width	normal 15 cm, broader 30 cm
Dash Mark Length	6 m
Dash Mark Spacing	12 m

3.3 Strategy for Creating the Scaled Environment

A specific scaling strategy is used to adapt the environment, which is based on two factors - one for the x-axis, one for the y-axis. The following explains the strategy of how the scaling factors are calculated (see section 3.4 for the actual calculations).

The first factor is based on the ratio of the width of a Golf VIII to that of a TurtleBot. For more information on why the Golf VIII has been chosen as the reference point, see section 3.6. The width of a Golf VIII, which is approximately 2.073 m, is set in relation to the width of the TurtleBot, which is 0.178 m (see section 3.2). This results in a ratio that is used as the scaling factor

from road model to environment when displayed as a floating point number. This factor is used to scale the x-axis of the environment. see Table 3.5 for the factor value.

Table 3.5: X-axis scale factor

Parameter	**Value**	**Unit**
Width TB	0,178	m
Width Golf VIII (incl. Mirror)	2,073	m
Width Scale (X-Axis)	**0,086**	

The second factor is based on the ratio of the Golf's maximum speed and the TurtleBot's maximum speed. For this, it's pretended that the real car can drive with a maximum speed of 100 $^{km}/_{h}$. Using the real maximum speed of the Golf VIII model would result in an impractical environment y-axis scaling. For example, the scaling would be so small, that every millimeter traveled in the environment would equal 0.333 m traveled in the real environment, making the driving functions hard to comprehend. Therefore, a custom maximum speed was established as the factor used to scale the y-axis of the environment. see Table 3.6 for the factor value.

Table 3.6: Y-axis scale factor

Parameter	**Value**	**Unit**
Pretend Speed	100,000	$^{km}/_{h}$
Pretend Speed	27,778	$^{m}/_{s}$
Real TB Speed	0,200	$^{m}/_{s}$
Speed Scale (Y-Axis)	**0,007**	

In order to use the same factor on Speed and Length, the regular TB length is scaled down. see Table 3.7 for the calculation.

Table 3.7: Scaling the TurtleBot length

Parameter	**Value**	**Unit**
Actual Length TB	0,138	m
Length Golf	4,284	m
Length Scale (same as Speed)	0,007	
Presumed Length TB	**0,031**	m

The summary of this approach, i.e. the scaling strategy, will be called Width-Speed-Scaling in following sections.

3.4 Scaling Down the Road Model

The table Table 3.8 shows the specifications of both the scaled and the original environment, with the respective scale factors used for the parameters. Based on the given values, the lane width of a German highway is 3.750 m in reality,

Table 3.8: Scaled environment parameters

	Actual	Axis	Scale Factor	**Scaled**	
Lane Width	3,750	X-Axis	0,086	0,322	m
Car Width	2,073	X-Axis	0,086	0,178	m
Roadway Width	11,250	X-Axis	0,086	0,966	m
Dashes Width	0,300	X-Axis	0,086	0,026	m
Speed	27,778	Y-Axis	0,007	0,200	$^{m}/_{s}$
Roadway Length	694,000	Y-Axis	0,007	4,997	m
Dashes Length	6,000	Y-Axis	0,007	0,043	m
Dashes Gap	12,000	Y-Axis	0,007	0,086	m
Car Length	4,284	Y-Axis	0,007	0,031	m

whereas it is 0.32 m in the scaled down environment, making the environment constructible.

3.5 Resulting Environment

In order to represent a real car in a smaller environment, the real models and Gazebo models of a three-lane highway use the dimensions depicted in figures Figure 3.1, which are based on the scaled environment parameters from table Table 3.8.

The resulting width of the scaled environment also fits a TurtleBot 3 Waffle model, since that model has a width of 30.6 cm. That means, the project group is not limited to using TurtleBot 3 Burger models only. E.g., Waffles could be used as trucks in the environment.

3.5.1 Graphical Representation of the Resulting Environment

Figure Figure 3.1 displays a graphical representation of the scaled environment using the calculated values from table Table 3.8

3.5.2 Applying the Results to the Environment Used in Reality and Simulation

The environments are depicted as in Figure 3.2 and Figure 3.3.

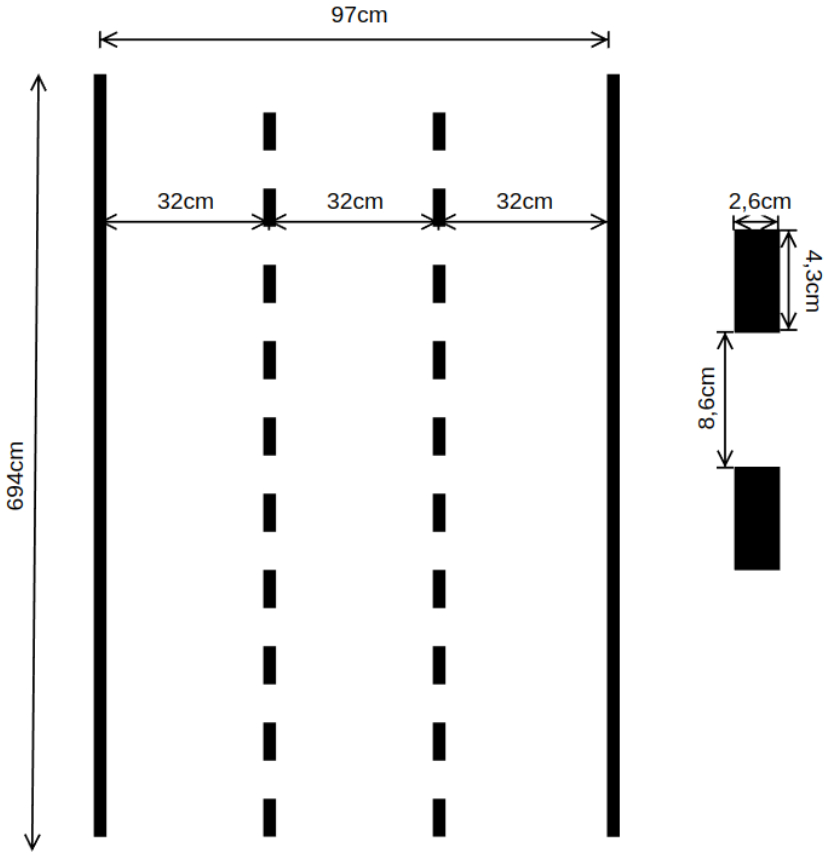


Figure 3.1: Specifications of the straight highway environment

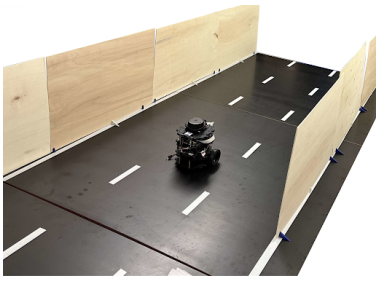


Figure 3.2: Road in reality

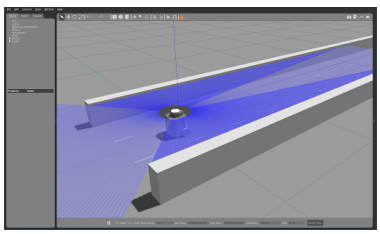


Figure 3.3: Road in the simulated environment (Gazebo)

3.6 Disadvantages and other Approaches

There are disadvantages to the chosen approach. Six other scaling strategies were contemplated, but ultimately it has been settled to use the described Width-Speed-Scaling approach, because the disadvantages of it seemed manageable - more so than those of the other strategies.

An example alternative approach would be using a length based ratio instead of the speed based ratio for the y-Axis. That approach would have resulted in three different scaling factors:

- Length: 0.032
- Width: 0.087
- Speed: 0.007

To mitigate this, Length and Speed could be brought together to one scaling factor (0.007) by changing the pretended maximum speed. However, the resulting pretended speed in the simulated environment would then equal 22.35 kmh at maximum. That means, it would have to be pretended that the Golf VIII can only reach that speed, which would be unfeasible for implementing driving functions based on reality.

The selected Width-Speed-Scaling strategy has its limitations and results in inconsistencies when applied to vehicle models that deviate from the dimensions of the Golf VIII, such as the Jaguar. Comparing such different sized vehicles to the TurtleBot's dimensions leads to discrepancies in the resulting scaling ratios. However, the Golf VIII is a good reference point for modeling the environment, since it is one of the most popular cars in Germany (see [27]). The error resulting from using other vehicle models in an environment that is scaled to the dimensions of the TurtleBot is expected to be neglectable. This is supported by the fact that only the y-axis scaling factor is derived from the ratio of the TurtleBot to a standard car model. That means, swapping in another vehicle configuration would only affect the scaled y-axis parameters. However, it's worth noting that, in the real world, vehicles of various sizes are commonly used, even though the typical German highway may be designed with a standard vehicle size in mind. This issue is not addressed further in the ongoing work of this project group.

The Width-Speed-Scaling additionally results in a very short TurtleBot 3 cm, which is shorter than in reality (13.8 cm). However, this allows the usage of only two scaling factors for the three dimensions: x-axis, y-axis, and speed, making further calculations easier. Using only two scaling factors is the main benefit of the Speed-Width-Scaling approach.

Since it is not possible to shorten the TurtleBot's dimensions to the resulting 3 cm in reality, this limitation is addressed by using the TurtleBot's actual length in critical components such as the Advanced Cruise Control. Furthermore, e.g. the vehicle model currently uses the scaled length of 3 cm, to make various calculations, such as the air resistance.

In summary, each analyzed scaling strategy presented its own set of disadvantages. The Width-Speed-Scaling strategy was chosen because its disadvantages were deemed to be the most manageable.

3.7 Differences between Gazebo and Reality

Various aspects of the Gazebo simulation lead to inevitable differences between it and the actual real-world setup. The following differences between the two have been identified:

- Gazebo has a continuous guardrail. In the real environment, this is approximated with smaller straight segments, which can lead to inaccuracies especially in curves. If the segments are placed too far apart, it can happen that a hole in the wall is detected, resulting in calculating wrong lanes.
- The small stands for the guardrail segments in the real environment currently reach into the lanes. This provides a potential hazard to the TurtleBot at the moment.
- The obstacle in Gazebo is 33 cm x 100 cm while in the real environment it is 32 cm x 44 cm. This means that the real environment obstacle doesn't completely fill a lane.
- The positions of lanes and lane markings in Gazebo are according to the measurements in Figure 3.1. In the real environment, they might be different depending on how precisely they are set up. The ground segments in real life are not perfectly flat, resulting in small inaccuracies in the lane widths.

3.8 Terms of Safety

The terms of safety for the TurtleBot follow those of the StVO.

3.8.1 Front Back Clearance

The distance to the vehicle in front or behind is not precisely defined by law. According to §4 Abs. 1 StVO, „The distance to a vehicle driving in front must generally be so large that it can be maintained behind it even if it suddenly brakes."

However, there are some general rules of thumb for orientation: One rule suggests that the distance should be at least half the value shown on the speedometer, which is based on §2 Abs. 3a StVO. The second rule suggests that within urban areas, the distance should be 1 second, which at a speed of 50 km/h is approximately 15 meters. Outside urban areas, the distance should be 2 seconds, which at a speed of 100 km/h is approximately 50 meters.

3.8.2 Lateral Clearance

Regarding the lateral distance during overtaking, the legal guidelines are vague. According to §5 Abs. 4 StVO, „When overtaking, a sufficient lateral distance to other road users must be maintained." For overtaking „pedestrians, cyclists, and operators of small electric vehicles" with motor vehicles, the guidelines are more specific: As per §5 Abs. 4 StVO, the sufficient lateral distance within urban areas must be at least 1.5 meters, and outside urban areas, it must be at least 2 meters.

3.8.3 Guidelines for the TurtleBot

In Table 3.9 are the clearance guidelines for the TurtleBot defined, which consider the regulations of the StVO. These are not complete (yet) and only are considering the scenarios which will be implemented.

Table 3.9: Clearances

Direction	**Clearance**	**Notes**
Longitudinal	$\frac{<\text{speed [km/h]}>}{2\ \text{[m]}}$	Rule of thumb: „half speedometer“ (§2 Abs. 3a StVO)
Lateral (overtaking car)	1 [m]	No concrete source could be found. There is a consensus among the internet on the value of 1 meter.
Lateral (overtaking bicycle, in town)	1.5 [m]	§5 Abs. 4 StVO
Lateral (overtaking bicycle, out of town)	2 [m]	§5 Abs. 4 StVO

Chapter 4

Vehicle Emulation

In order to be able to develop controllers for real cars and test them on the TurtleBot, the behavior of a real vehicle needs to be emulated on the bot. This section describes the vehicle model used for emulation, its dynamics and its control inputs. First, an idealized vehicle model is defined, which is an abstraction of the actual emulation that can be used for controller development. Afterward, the differences between the idealized model and the actual implementation are discussed.

4.1 Idealized Vehicle Model

The vehicle emulation is based on the kinematic bicycle model. The bicycle model is a simplified representation of a car's dynamics that can be used in the field of vehicle dynamics and autonomous driving for motion planning and is often used for model-predictive control [38]. It is called the „bicycle model“ as it consolidates the dynamics of a car into a two-wheeled model, where the two front wheels and the two rear wheels are each represented as a single wheel. The model can be defined from different points of view along the vehicle. For this project group, a model with a viewpoint from the center of the rear axle was chosen for reasons of simplicity. For controlling the lateral movement, the bicycle model uses steering angle as input. The model used here is based on the bicycle model definition given in [38]. It contains the following variables:

- X and Y: The longitudinal and lateral positions of the vehicle, respectively
- θ: The heading angle of the vehicle
- v: The speed of the vehicle
- a: The acceleration of the vehicle
- α: The steering angle
- l: The wheelbase length, which is the distance between the front and rear axles
- r_1: Tire friction value

- r_2: Air resistance value
- d_1: Linear approximation of the disturbance on the acceleration due to motor friction and transmission

l, α, r_1 and r_2 are all parts of the vehicle configuration. For model simplicity, the acceleration in this model is only affected by friction forces. Motor resistance is only considered as a linear disturbance parameter d_1 acting on the acceleration input in this idealized model. These errors were chosen to only affect acceleration, not steering.

The control inputs to the model are defined as follows:

$$u_1 = a$$
$$u_2 = \alpha$$

The dynamics are defined as follows:

$$\dot{x_1} = \dot{X} = x_3 \cdot \cos(x_4)$$
$$\dot{x_2} = \dot{Y} = x_3 \cdot \sin(x_4)$$
$$\dot{x_3} = \dot{v} = r_1 \cdot x_3 + r_2 \cdot x_3^2 + d_1 \cdot u_1$$
$$\dot{x_4} = \dot{\theta} = \frac{v}{l} \cdot \tan(u_2)$$

In the beginning of the project group, a simplified model with three instead of four states had been used. The velocity had been modeled as a control input and the implementation was based on using always the maximum acceleration and deceleration to achieve that velocity as quickly as possible. Because this only allowed for very sharp and inconvenient driving maneuvers, the model was reworked to use the acceleration as an input, as it gives the controllers more freedom and represents more closely the input of a real car.

4.2 Emulated Vehicle Model

The emulation extends the idealized model described in section 4.1 in order to make it more realistic. In addition to friction and wind resistance, it contains an approximation of a car's transmission and gear shift, which affects the actual acceleration of the car. This is not part of the idealized model. Controllers need to be designed in a way to be robust against these errors and model discrepancies.

In order to provide these realistic disturbances, a motor and friction model is used to calculate the highest acceleration of the vehicle possible given the current gear, vehicle weight, velocity and other relevant variables. If the acceleration given by the controller is higher, the highest possible acceleration of the vehicle is used instead. Conversely, if the vehicle is braking (the controller is giving a negative acceleration), the maximum possible deceleration is calculated and the minimum is used.

The model and its implementation are described in the following.

4.2.1 Motor Model

The motor model is implemented through a series of calculations and functions which account for various factors including the engine's revolutions per minute (RPM), torque and gear ratios. The motor model takes into account car-specific factors which can be configured to emulate different types of cars.

The following calculations are based on the formulas given in [45].

Engine RPM Calculation: The engine's RPM is computed based on the vehicle's current speed, the wheel circumference, and the current gear and final drive ratios.

1. Convert the speed from meters per second (`speed_m_s`) to meters per minute by multiplying with 60:

$$\text{speed\_m\_min} = \text{speed\_m\_s} \cdot 60$$

2. Calculate the wheel revolutions per minute (RPM) by dividing the speed in meters per minute by the wheel circumference (`wheel_circumference_m`):

$$\text{wheel\_rpm} = \frac{\text{speed\_m\_min}}{\text{wheel\_circumference\_m}}$$

3. Finally, calculate the engine RPM by multiplying the wheel RPM with the current gear ratio (`current_gear_ratio`) and the final drive ratio (`final_drive_ratio`):

$$\text{engine\_rpm} = \text{wheel\_rpm} \cdot \text{current\_gear\_ratio} \cdot \text{final\_drive\_ratio}$$

Torque Calculation: The current torque is calculated based on the engine's RPM. A linear interpolation function, `interp1d`, is employed to interpolate the torque values from a predefined set of engine speed and torque points.

Gear Shift Handling: The motor model checks whether a gear shift is available or necessary based on the current RPM and the specified RPM ranges for each gear. If a gear shift is required, the current gear is updated, and the time of the last gear switch is recorded.

Engine Acceleration Force Calculation: The engine acceleration force is the total force provided by the engine and is calculated using the current torque, gear ratio, final drive ratio, and the wheel radius. This calculation accounts for transmission losses.

1. Compute the engine torque after transmission by multiplying the average engine torque (`avg_engine_torque_nm`) with the gear ratio (`gear_ratio`) and the final drive ratio (`final_drive_ratio`):

$$\begin{aligned}\text{engine\_torque\_after\_transmission} = {} & \text{avg\_engine\_torque\_nm} \\ & \cdot \text{gear\_ratio} \cdot \text{final\_drive\_ratio}\end{aligned}$$

2. Compute the engine torque after accounting for engine losses by multiplying the engine torque after transmission with the engine loss factor (`engine_loss`):

   $$\text{engine\_torque\_after\_losses} = \text{engine\_torque\_after\_transmission} \cdot \text{engine\_loss}$$

3. Finally, calculate the engine acceleration force by dividing the engine torque after losses by the wheel radius (`wheel_radius_m`):

   $$\text{engine\_acceleration\_force} = \frac{\text{engine\_torque\_after\_losses}}{\text{wheel\_radius\_m}}$$

The TurtleBot's linear and angular velocities can be controlled to emulate the motion of a vehicle as described by the Bicycle Model. The model's state variables are mapped to TurtleBot controls as follows:

- The longitudinal velocity v of the model corresponds to the linear velocity of the TurtleBot.
- The heading θ of the model is used to control the angular velocity of the TurtleBot

4.2.2 Steering Angle Limiting

To ensure that the simulated vehicle behaves realistically at high speeds, its steering angle needs to be constrained depending on its driven speed. The steering angle directly affects the turn rate of the vehicle; a high steering angle combined with high speed results in high lateral acceleration and consequently could lead to a loss of control over the vehicle. To avoid this and ensure that the vehicle's behavior is predictable, the allowed lateral acceleration of the vehicle needs to be restricted in the emulation model.

The authors of [8] present a formula for the accepted lateral acceleration as a function of vehicle speed. The formula uses a criterion taken from [31] using data from a driving behavior study to model the acceptable lateral acceleration for an average and a 85th percentile driver. The term „85th percentile driver“ here refers to someone driving more dynamically than 85% of the population. This solution is suitable for setting a velocity-based limit on the steering angle in the used vehicle model, as it is slip free and relatively simple. For more complex vehicle models, additional factors such as tire dynamics and road conditions could be considered in order to limit the steering angle in a way that ensures safety. This alternative approach would be independent of the drivers comfort and instead based on vehicle configuration and road conditions.

The calculation of the maximum acceptable steering angle in the model is as follows:

1. **Radius**: The turning radius R of the vehicle given a steering angle δ and the vehicle's wheelbase L is calculated as

   $$R = \frac{L}{\tan(\delta)}$$

   The turning radius is inversely proportional to the tangent of the steering angle, which follows from the Bicycle Model.

2. **Current Lateral Acceleration**: Given the vehicle's velocity v and turning radius R, the lateral acceleration a_{lat} experienced by the vehicle can be calculated using the centripetal acceleration formula:

$$a_{lat} = \frac{v^2}{R}$$

3. **Accepted Lateral Acceleration**: The accepted lateral acceleration for a given velocity is calculated using the criterion K from [31]. For an average driver, the K criterions has been estimated to be 36.0 and for the 85th percentile driver 42.0. The formula from [31] is as follows:

$$a_{lat\_accepted} = \left(\frac{K}{v}\right)^2$$

If the lateral acceleration for a given speed remains below this value, that means the vehicle's behavior is within safe limits.

4. **Steering Angle**: If the current lateral acceleration exceeds the accepted value, the steering angle δ must be reduced. This is done by first calculating the maximum allowed turning radius R_{max} using the accepted lateral acceleration (formula derived from step 2):

$$R_{max} = \frac{v^2}{a_{lat\_accepted}}$$

Subsequently, the required steering angle to achieve this turning radius is found using:

$$\delta_{max} = \arctan\left(\frac{L}{R_{max}}\right)$$

which gives the maximum permissible steering angle.

Additionally, the model incorporates a static maximum lateral acceleration value of $5\,\mathrm{m/s^2}$. This limit was derived based on the graph in Figure 4.1 and was imposed to improve the low speed behavior of the simulated TurtleBot.

4.3 Vehicle Configuration

Vehicle configuration files, which contain all parameters necessary to simulate a realistic vehicle, are used. These can be switched out depending on the simulated scenario. Each configuration file is written in the YAML language and contains the parameters for a specific vehicle model. Furthermore, the configuration of each vehicle includes models for single vehicle parts such as the Motor or the Transmission. This modularity enables constructing configurations using various vehicle part models that have already been defined. Currently two different configurations are used, one for simulating a sports car (Jaguar F-Type) and one for a more casual car (VW Golf VII). The modules describing vehicle parameters are divided into five categories.

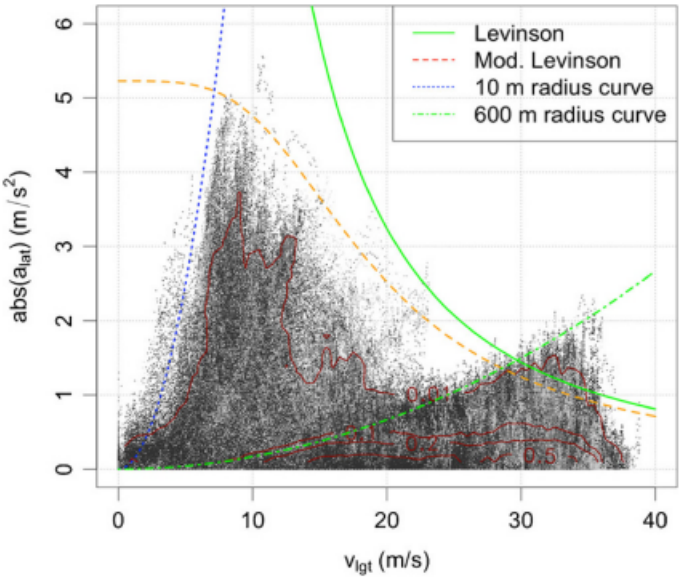


Figure 4.1: Lateral acceleration in relation to speed for study data, taken from [8]. The green curve is the limit calculated using Levinson's criterion for an average driver [31].

4.4 Engine

The engine is a component that is used by all vehicles and usually varies from vehicle to vehicle, therefore, its parameters need to be specified separately. Performance diagrams might need to be evaluated to acquire some of the engine parameters. The engine specification can be found in Table 4.1.

Table 4.1: Engine Specifications

Parameter	**Unit**
Max torque	Newton Meter (Nm)
Speed at maximum torque	Revolutions per minute (RPM)
Maximum power	HP
Speed at maximum power	RPM
Average torque	Nm
Average loss	Percentage
Speed points full load	RPM
Static torque points full load	Nm

4.5 Transmission

Similar to the engine, transmissions are usually unique across different vehicle models. The transmission specification can be found in Table 4.2.

Table 4.2: Transmission Specifications

Parameter	Unit
Gear switch time	Seconds
Start speed	Revolutions per minute (RPM)
End speed	Revolutions per minute (RPM)
Gear Ratio	Multiplier value
Final drive ratio	Multiplier value

4.6 Wheels

The wheels are a highly variable component when comparing different vehicles. The configuration of wheels uses one of the basic wheel size specified by the manufacturer. The wheel characteristics can be found in Table 4.3.

Table 4.3: Wheel Characteristics

Parameter	Unit
Wheel radius	Meter
Wheel circumference	Meter
Friction	Newton

4.7 Vehicle Dynamics

The remaining parameters are dependent on the whole car. They can be found in Table 4.4.

Table 4.4: Vehicle Dynamics and Performance

Parameter	Unit
Maximum steering angle	Radians
Wheelbase	Meters
Braking force	Newton
Mass	Kilogram
Air resistance	Newton
Aerodynamic drag	Newton
Frontal area	Square meter
Maximum speed	KPH
Acceleration time 0 to 100 KPH	Seconds

4.8 VW Golf VII

The VW Golf VII was selected to represent a casual everyday car compared to the rather sporty Jaguar F-Type. The file `VW-Golf-7_2-0-TDI_DSG.yml` de-

scribes a Volkswagen Golf MK7 with an 2.0 litre diesel engine which provides 150 HP. The used parameters from Table 4.5 correspond to models built from 12/2016 to 05/2020, and mainly influence the motor and transmission type. There has been a facelift in 2017, which slightly changed the exterior und integer design, but has no impact on technical parameters.

The transmission type is a DSG, which is an automatic transmission in the Volkswagen Group, and has seven gears. The specific transmission type is called „DQ381“. The default wheel and tires suggested by the manufacturer are of the dimensions 205/55 R16.

Some of those specifications are not strictly bound to the car model itself, i. e. the transmission „DQ381“ is used in many other vehicles. The information regarding the specifications of this vehicle was gathered via several internet sites and is linked in the subsection below.

Important note:
In comparison to the Jaguar F-Type configuration file, the golf has two different `final gear ratio` for different sets of gears. Due to this fact, the Jaguar F-Type configuration was adapted to represent this structure.

Table 4.5: VW Golf VII Model Details

Parameter	**Details**
Model	VW Golf VII 2.0 TDI with DSG
Build Duration	12/2016 - 05/2020
Remarks	The Golf VII had a facelift in 2017 (no impact on specifications)
Engine Type	Diesel
Engine Series	VW EA288
Engine Code Letters	CRMB, DCYA, DEJA, CRLB
Displacement	1968 cm^3
Max. HP @ RPM	150 @ 3500 - 4000
Max. Torque @ RPM	340 @ 1750 - 3000
Used Wheel Size	205/55 R16
Transmission Type	DQ381
Remarks on Transmission	DSG with 7 gears (From 12/2026, 6 gears previously)
Drive Type	Front wheel drive

4.8.1 Notes for parameters

Mass

- The mass is calculated by adding 100 kg to the curb weight (Leergewicht) of the car.
- Curb weight is 1316 kg.
- 100 kg is split into:

  - avg. of 80 kg for one person.
  - roughly 20 kg of fuel (diesel mass = 0.820g per litre times half tank volumes = 25 litres).

Final Drive Ratio

- The used transmission has two values instead of a single global one for each gear.
- The final drive ratio is assigned to each corresponding gear.

Steering Angle

- *Auto Motor und Sport* specifies a „40°steering angle for conventional vehicles“ [4].
- 40°converts to 0.6981 radians.

Braking Force

- In Newton, given by weight times deceleration.
- Deceleration depends on how hard the brake is applied.
  - Emergency braking equals about $10.6m/s^2$ for the Golf MK7 [11].
  - The Minimum required by law is $2.5m/s^2$ [10].
  - For calculation $7m/s^2$ is used, which represents medium braking.
- 1416 kg times $7m/s^2$ equals 9912N.

Aerodynamic drag, Frontal area and Air resistance

- The values were taken form the collection of *Rüdiger Cordes* [14].

Gear Switch Time

- The values were taken from *VWVortex* [16].

Gears

- The following links contain information about the gear ratios:
- `https://www.golfmk7.com/forums/index.php?threads/dq381-dsg-gear-ratios.360005/`
- `https://forums.tdiclub.com/index.php?threads/shift-points-on-mk7-tdi-manual.431653/`
  - Even though internally manual gears are used in the model, the shift points should be the same as in the acquired data.

Engine

- The dyno chart was taken from *More BHP* [5].
  - It shows two graphs, the important one is the thick line representing the stock engine.
  - The chart was manually evaluated using *Engauge Digitizer* [32].

4.8.2 Further Sources

- Car:
  - `https://de.wikipedia.org/wiki/VW_Golf_VII#Dieselmotoren`
  - `https://carwiki.de/vw-golf-7-technische-daten/(mustbemanuallysettoDiesel/150PS/2.0TDI(150PS)DSG)`
  - `https://www.auto-data.net/de/volkswagen-golf-vii-facelift-2017-2.0-tdi-150hp-dsg-27831`
- Wheels:
  - `https://www.1010tires.com/Tools/Tire-Size-Calculator/205-55R16?active=0&ismetric=true`

4.9 Jaguar F-Type

The configuration of this vehicle model originates from the pre-project and contains the specifications of a Jaguar F-Type. These specification are depicted in Table 4.6 The Jaguar F-Type was selected to represent a sports car amongst the vehicles that will be simulated.

Table 4.6: Jaguar F-Type Specifications

Parameter	**Specification**
Model	Jaguar F-Type
Engine type	3-litre V6 DOHC V6
Max. HP @ RPM	340 @ 6500
Max. torque @ RPM	450 @ 3500
Used wheel size	295/30 R20
Transmission type	Automatic, ZF8HP, RWD
Drive type	Rear-wheel drive

4.9.1 Notes for parameters

Mass

- The mass is calculated by adding the driver's weight to the curb weight of the car.
- Curb weight is 1741 kg.
- Driver's weight is 80 kg.

Engine

- Engine speed for maximum torque: 3500 rpm
- Engine speed for maximum power: 6500 rpm
- Maximum engine speed: 6500 rpm
- Minimum engine speed: 1000 rpm

Transmission

- Highest gear: 8
- Final drive ratio (differential): 3.31
- Driveline efficiency: 0.85

Tires

- Tire width: 0.295 m
- Rim diameter (converted to meters): 0.508 m
- Wheel (tire) friction coefficient: 1.1
- Rear axle load coefficient: 0.65

Vehicle

- Drag coefficient: 0.36
- Frontal area: 2.42 m^2

The parameters were taken from *X-engineer.org* [48]. Additionally, this site provides a single `final_drive_ratio` parameter that applies to all gears and therefore was initially being set only once. Due to the introduction of the VW Golf configuration (section 4.8) which has two different `final_drive_ratio`, this parameter was copied and is the same for both final drive ratios.

Chapter 5

Sensor Augmentations

This section describes various sensor augmentations made during the project group. The augmentations are ranging from hardware modifications to additional ROS topics.

5.1 Camera

The camera of the TurtleBot streams the image data in front of the TurtleBot into the ROS network. The camera data can then be used to perform lane and object detection in the frames sent by the camera. Lane boundaries and road participants are examples of objects to be detected.

5.1.1 Camera Service

The camera service is used to stream image data into the ROS network. It can be used in two ways:

- With a web server
- Headless

With a web server This variant uses a web server to show the image stream sent by the TurtleBot camera on a webpage. The server is hosted on the TurtleBot's network address and is running on port 5000. To see the images, the webpage has to be refreshed once after starting the camera service. This variant is more suitable for troubleshooting.

Headless The second variant needs no user interaction for sending messages. If you call the ROS service

```
/camera_serice
```

it will either start or stop sending images.

Parameters In the service request, different values are used for parameters as „Opcodes“ to customize how the node should behave. These are listed in Table 5.1, Table 5.2, Table 5.3, Table 5.4:

Table 5.1: Parameters for the camera service request

Parameter	Description	Default Value
request	The request opcode	0
frame_width	The requested frame width of images	640
frame_height	The requested frame height of images	480
frame_rate	The framerate to capture images with	10

Table 5.2: Table for Opcodes

Opcode	Description
0	Toggels camera server

Table 5.3: Parameters for the camera service response

Parameter	Description	Default value
status	The response Code	0
message	A message with status information	n.a

Table 5.4: Table for status codes

Opcode	Description
0	Success

5.1.2 Camera Mount

The TurtleBot already had a static mount for the camera attached. To be able to dynamically change the viewport of the camera, the simple mount was replaced by a more advanced mount. This new mount uses a pan-tilt design to make the camera angle adjustable on two axles. The new mount was designed for and printed using a 3D-printer. For the assembly, small screws were used and the servos were put in place, even though they are not connected or controlled yet. It can be seen in Figure 5.1.

5.2 Odometry

The TurtleBot uses the Odometry topic `/odom` to publish information about the TurtleBot's position and movement. This data is however quite nosiy. In order to acquire smoother data, an Extended Kalman Filter is employed to combine Odometry and IMU data. The IMU sensor yields data about the TurtleBot's orientation. The topic `/odometry/filtered` is used to publish the filtered data.

The filter was implemented using an online guide by „Automatic Addison“ [3]. The following dependency is required for the filter node:
`ros-humble-robot-localization`

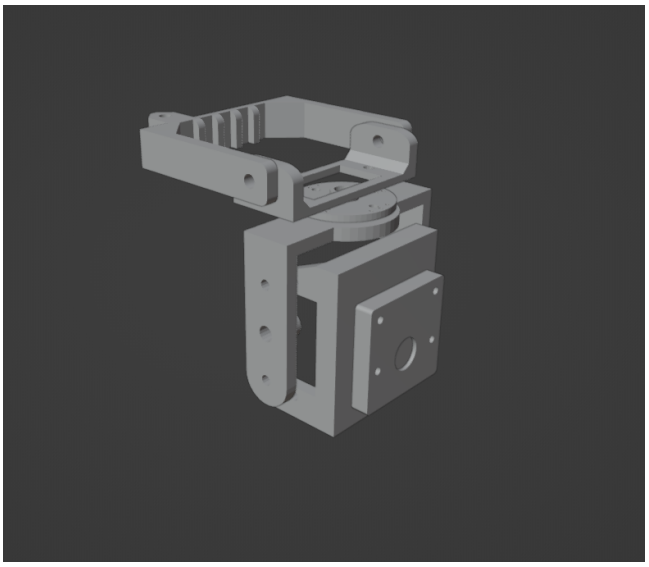


Figure 5.1: New camera mount with movable joints to control the camera with servo motors.

Chapter 6

TurtleCar-Core

There are multiple ROS Nodes running on the TurtleBot which together form TurtleCar-Core. The architecture of each node is described here. In order to run these nodes, a specific setup of the image running on the TurtleBot is required, which is explained here as well.

6.1 TurtleCar Node

In the module called `TurtleCar-Core`, the main parts of the software controlling the TurtleBot are implemented. Its tasks are to gather sensor data, define a control action according to its current scenario and goal, and publish that action to the relevant actuators.

6.1.1 Architecture

The diagram in Figure 6.1 shows the basic building blocks of the code. It is simplified in the way that the *TurtleCarNode* class is the root class and consists of all other classes. In order not to clutter the diagram, these compositions are not drawn.

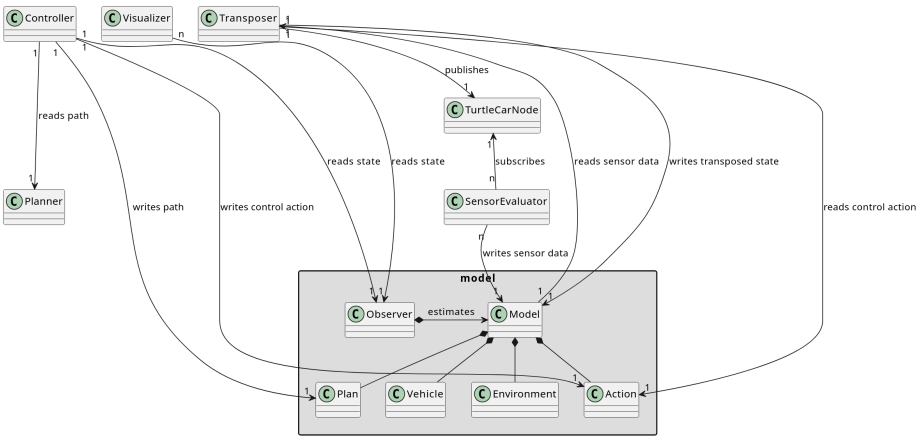


Figure 6.1: Static view of the architecture of the *TurtleCarNode*

The architecture is modular and can be separated into these classes:

- **TurtleCarNode:** The *TurtleCarNode* class is the root class. It provides the ROS interface which is used by other parts of the software to subscribe or publish to ROS topics. It is also the root for the tree of dependent classes.
- **Model:** The model represents the observable state of the robot. It contains information on the state of the vehicle as well as the environment and the control actions taken. It is filled by the *SensorEvaluator* classes and read out by the *Observer*.
- **Sensors Evaluators:** These classes read out sensor values by subscribing to their ROS topics and processing the information gathered to create meaningful information from them, i. e. detecting obstacles or lanes. The processed information is added to the *Model*. To gain information from a sensor and put it into the model, this class needs to be inherited from.
- **Observer:** This class acts as an Observer in the context of control design. The data in the model only represents the observable state, which may not be the complete state information needed to control the system. The observer estimates the actual state from the observable model. The *Controller* and the *Visualizer* read from this observer instead from the *Model* directly. If the model is amended, the observer probably has to be altered as well.
- **Controller:** Reads the state information provided by the *Observer* and decides on a control action depending on that state. Writes the control action back into the *Model*. Adaptation of the robots actions is done here. Third parties are able to write their own controllers, in order to implement driving functions.
- **Visualizer:** Reads the state information provided by the *Observer* and visualizes it through a GUI. You may add additional visualizers.
- **Transposer:** The goal is to simulate a car which has a different behaviour than the TurtleBot. The Transposer reads the control actions from the *Model* and maps them to the behaviour of the car model. It then publishes messages via the *TurtleCarNode* to the bot's actuators so that the robot shows that behaviour. Since it simulates the car, it also writes the information about the car's new state - like the current gear - back into the *Model*.

6.1.2 TurtleCarNode Core Loop

The core loop of the *TurtleCarNode* on a high level is shown in figure Figure 6.2.

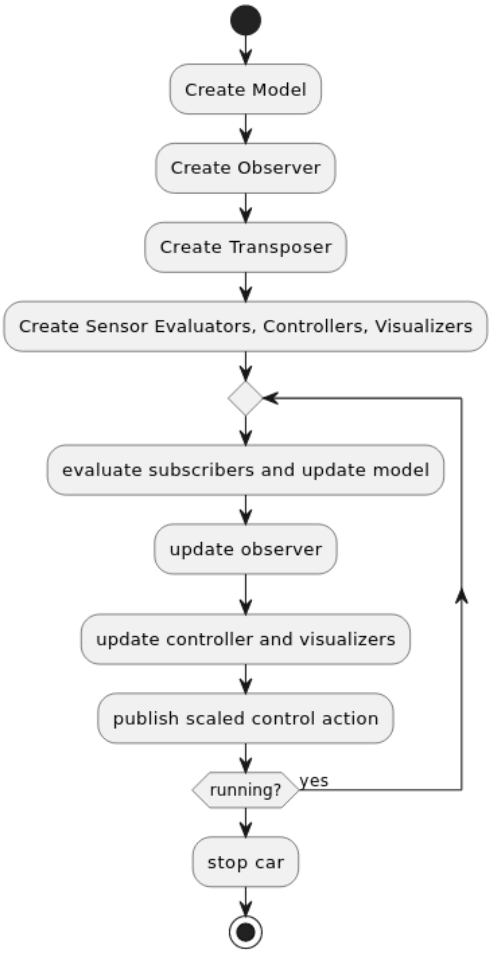


Figure 6.2: Start and core loop of *TurtleCarNode*

6.1.3 Filtering Sensor Values

The information gathered from the sensors via the ROS interface may need to be filtered to be usable by the modules interpreting that sensor data. In the context of the project group, two variants of filtering are defined, which are reflected in different aspects of the architecture:

Technically Necessary Filtering There exist technical reasons for filtering values directly when they are retrieved from a ROS message. One example is the LIDAR: It has a varying resolution which must be upscaled to a fixed resolution by interpolating missing values. This is done directly when retrieving the values. Sensor Evaluators using the standard `lidar.subscribe_lidar()` function implicitly receive the fixed-resolution values. When necessary, a similar standard filtering behaviour may be implemented for other sensors as well.

Task Specific Filtering Some Sensor Evaluators may have requirements for filtering the sensor values that are not necessary for processing the values, but are functional requirements related to their task. These filters are implemented in the context of the Sensor Evaluator and only used to fulfill its task, but do not influence the input to other Sensor Evaluators. Each Sensor Evaluator has to explicitly implement the filters it needs or explicitly use a filter function shared between evaluators.

6.1.4 Unit Testing

pytest [28] is used to perform unit tests. For mocking, *mockito-python* [34] is used. When writing unit tests, the following criteria should be met:

- Test one specific aspect of the code under test
- Mock the complete environment of the function. Everything that is not part of the code under test should not be executed.
- If the tests or the mocking effort is high, consider refactoring the tested code to enable smaller tests.

All tests are located in the `tests` directory.

6.2 TurtleBot ROS2 Image

It is possible to automatically build a minimal, customized image, which is real-time capable, for the TurtleBot. The repository which can be used for this can be found in the project groups GitLab [24].

It is important to note that the image cannot be built using the docker image, due to limitations of *systemd-nspawn*.

Dependencies are customizable by opening

```
image_builder/data/jammy-rt-hubmle/scripts/
```

and adapting the file `phase1-target`. Under the comment „user-specific dependencies“ it is possible to add desired dependencies via apt.

To build the image, change to the top folder and run

```
make jammy-rt-ros2
```

After that, a fully bundled `.img` file is generated, which can be burned to a sd card. Detailed documentation can be found in the repository. Please note this is an updated version of another public project called *Raspberry Pi image with ROS 2 and the real-time kernel* [20].

Chapter 7

Code Quality

In this part of the documentation, elaboration on the decisions regarding Coding Style and Static Code Analysis (SCA) can be found. In the context of the project group, Coding Style and SCA are differentiated. Coding Style includes the ruleset and principles which influence what code is produced. SCA consists of development tools and CI/ CD methods, which allows maintaining parity to the set Code Quality. CI stands for Continuous Integration, and is a typically automatically triggered process that performs tasks such as checking the source code, running tests and ensuring that the source code is compilable [19]. With this process, the goal of having a stable code repository in terms of Code Quality is supported, since automatic Code Quality checks are possible. This is explained in more detail in section 7.3.

7.1 SCA

There are two main parts of SCA used in the project group: formatting and linting. The coding style is provided by the tools used.

Formatting is the way how the code is formatted: which indentation size is used, how long lines should be and where newlines are located. Formatting ensures that each line of code that is written is in a format that is comprehensible by each member of the project group. The code formats automatically and no further manual intervention is required.

Linting on the other hand makes sure that the code that is written is error-free and adheres to a certain code style: Here, checks against unused variables, long lines and unnecessary complexity are employed. Some linting errors are also fixed by formatting, i. e. long lines. But because fixing most linting errors is a non-trivial task, oftentimes manual intervention is required.

7.2 Development Tools

In order to ensure that the code is in the correct format and to reduce its errors, tools are employed. For formatting, use *black* [6] is used. For linting, use *ruff* [42] is used.

Both tools were chosen for the following reasons:

- Opinionated
  - Being opinionated allows for adhering to community rules forged by years of development time.
  - At the beginning, there is no desire to employ custom, project group specific rules. Everything is changing constantly - there is no need for complex configurations, but for quick usage.
- Modern
  - Modern tools allow for staying cutting-edge.
  - They improve the readability of the codebase.
- Fast
  - Being fast means every machine can run the tools, even if one developer happens to have a slow machine (by modern standards).
  - There is no need to worry about the code base growing so large that the SCA tools will take an unreasonable amount of time to run.
  - Limited numbers of job runners are available in GitLab, as the instance is self-hosted. Therefore, being fast reduces the occupation on those limited resources.

7.3 Continuous Integration

In this section, the ways CI is used in the project group are described. Also, the configuration of the employed pipelines is explained. Pipelines are essentially a set of steps the source code has to pass in order to be valid.

7.3.1 Integration in the workflow with CI

In order to allow constant integration, *black* and *ruff* are used not only locally, but also in the GitLab projects pipelines. This ensures that every commit and merge request is checked.

If *black* detects that the format is not correct or *ruff* finds any linting errors, merging the respecting merge request is disallowed. Also, reviewers will immediately take notice of this and will ask the developer to fix this.

This ensures that the code in the stable branches of the projects remains protected and in a valid state. Additionally, this provides fast feedback for developers whether their code contains errors. This makes locating and fixing errors faster.

7.3.2 Pipeline

In the pipeline, ruff and black are executed. In the following, the mechanics of the pipeline are documented. This part explains the following:

- Elaboration on the pipeline concepts, not the details
- Explanation of the most important caveats, like caching and sometimes allowing pipes to fail
- Starting point for getting to know the pipeline

How the pipeline works The following will explain the structure & concepts of the pipeline in use. For a better understanding, please take a look at the *.gitlab-ci.yml*. It is included at the root of the *turtlebot* project.

In the implementation of the pipeline configuration, the official Python docker image is used, so that some configurations for the executing runner are already present.

Pipeline Building Blocks

- `before_script` Block
  - Ensures that a virtualenv is used
  - Debugs the Python version
  - Executes before each job
- `build-job`
  - Currently only a stub
  - Might be used later, when actual building of the ros packages is required
- `format-test-job`
  - Runs black and checks for formatting errors
  - Prints encountered errors to ‘stdout‘ for debugging purposes
- `lint-test-job`
  - Runs ruff
  - Looks at the .pyproject.toml file in order to configure ruff
  - Generates a codequality artifact .json, which is used by GitLab to measure code quality
  - Also prints all encountered errors and warnings to `stdout`
  - By using *dependencies*, this job only runs after `format-test-job`

Important note: The `test` in the jobs name refers to the task of testing if the source code is in a conforming state. This does not mean that the jobs are only 'test' versions.

Caching the installed pip packages The `cache` is used in order to let the runner cache installed packages, so that ruff und black are not reinstalled in every run of the pipeline.

By configuring `PIP_CACHE_DIR`, pip is told to cache its dependencies and installed packages in the directory provided - which are defined as a pipeline cache directory as well. Therefore, the cache directory gets cached in between job runs and reused.

Conclusion: Working with the pipeline Now that the pipeline is configured, it is possible to review Merge Requests based on their generated code quality report. Also, this makes sure that every line of code is formatted in a consistent way. When committing to a custom branch, or merging to ‘main‘, the pipeline is evaluated and run. Project members are required to provide conforming source code, and get hints to why their changes might not be of the desired quality.

Chapter 8

Lane Detection

For a TurtleBot used in the context of autonomous driving, the ability to perceive and understand its road environment is of high importance. One crucial aspect of this perception is the lane detection, which involves identifying and tracking the lanes on the road. Accurate lane detection is a fundamental building block for many autonomous driving functions, from simple lane-keeping assistance to complex path planning and decision-making algorithms. This section introduces two approaches to lane detection, one based on LIDAR and on based on camera data.

8.1 LIDAR-Based Lane Detection

LIDAR technology plays an important role in the project's implementation of lane detection, as the LIDAR provides essential data about the robot's surrounding. The concept here is to utilize this data to calculate and represent lane boundaries accurately. Visual lanes as indicated by lane markings are therefore not directly detected but are rather projected based on a given road configuration and a rightmost boundary that is detectable by the LIDAR.

8.1.1 Preconditions

The calculation of the lane boundaries using LIDAR assumes that a certain structure for the lanes is always present. One particular assumption is that there always exists a wall that is detectable by the LIDAR sensor on the right side of the road. Furthermore, the first lane always has a distance of w_s to this wall, forming a road shoulder with constant width. Additionally, every lane has the exact same constant width, noted as w_l in the following.

8.1.2 Coordinate Transformation

The process of the LIDAR-based lane detection begins with transforming polar coordinates into the Cartesian coordinate system. This conversion simplifies subsequent processing steps and provides a clear representation of the environment. Based on a respective angle α_i and a distance value d_i of each LIDAR measuring point i, Cartesian coordinates x_i and y_i for such point can be created using the common formulas $x = d * \cos(\alpha)$ and $y = d * \sin(\alpha)$.

8.1.3 Boundary Detection and Lane Projection

Once in Cartesian coordinates, the system calculates the lane boundaries based on the distance of the robot from the right wall. In particular, this calculation uses the coordinates of the wall that is detected by the LIDAR between 230° and 300°. This is effectively any wall that is to the right of the TurtleBot's facing direction. A B-spline is then fitted to these data points, ensuring smooth and continuous representation of the lane-defining wall. Using B-splines offers the possibility to control the degree of the lane boundaries, which is useful to extend the lane projections from straight to curved roads. For individual points of the given B-spline, normalized orthogonal vectors are then calculated. This is done by first calculating a tangential vector of a given point on the B-spline, normalizing that vector and then rotating it by 90° into the correct direction. For a given lane $n \geq 0$, these normalized vectors are used to determine the position of the lane's right ($j = n$) and left ($j = n + 1$) boundary, if multiplied with the factor $(w_s + j * w_l)$.

8.1.4 Current Lane Determination

Identifying the current lane is the next critical aspect of the lane detection. This is accomplished by evaluating the closest measured distance to the wall that is used as a basis for the lane projection, as introduced above. For example, if the robot's facing direction is parallel to the wall, the angle for the closest distance is typically at 270°. Given that this minimum distance to the boundary wall is d_w, the current lane number n_{TB} can then be calculated as follows:

$$n_{TB} = \left\lfloor \frac{d_w - w_s}{w_l} \right\rfloor$$

8.2 Camera-Based Lane Detection

Limitations of the LIDAR-based lane detection approach, such as the requirement of a wall being present on the lane boundary, have motivated the implementation of a camera-based approach. Camera-based lane detection allows the TurtleBot to detect lane markings using visual data from its onboard camera. The project group has explored two distinct methods for lane detection in camera images: classical computer vision techniques and an AI-driven approach. The latter, overcoming limitations of the former, has been adopted.

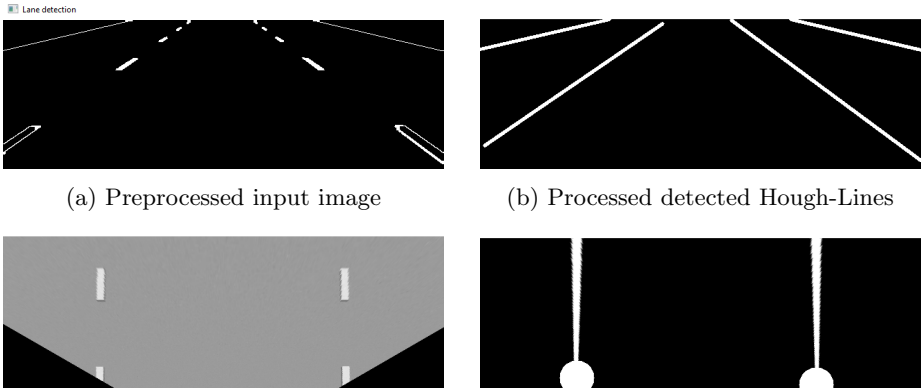
8.2.1 Classical Computer Vision Approach

The initial approach for the detection of lanes was based on the Hough Line Transform, which is a technique for detection of straight lines within images. The overall concept of this method was inspired by [25] and followed several stages Figure 8.2 (outputs are visualized in Figure 8.1):

1. Region of Interest (ROI) Segmentation: Isolation of the road segment from the camera's field of view.

2. Preprocessing: Conversion to grayscale, blurring, edge detection via the Canny algorithm and morphological closing to prepare the image for the Hough Transform.

3. Hough Line Transform: Detection of potential lane line candidates in the preprocessed image.
4. Postprocessing: Filtering and clustering of the lane line candidates.
5. Bird's-eye View Transformation: Transformation of the coordinates for further use.
6. Lane Line Extension: Extrapolation of the detected lines.
7. Lane Data Calculation: Estimation of the TurtleBot's current lane and the positions of adjacent lane boundaries, followed by scaling of the detected coordinates.



(a) Preprocessed input image (b) Processed detected Hough-Lines

(c) Bird's eye view of the input image (d) Bird's eye view of the Hough image

Figure 8.1: Outputs of the stages of the initial approach.

The segmentation ensures a focus on the road, while the preprocessing steps reduces noise and filtered unnecessary information from the image to ensure better performance of the Hough Line Transform. Since the Hough Line Transformation has a tendency to detect multiple lines where there should only be one, clustering and filtering of outliers allows increases the precision of the detection.

While this approach performs well in predefined simulation scenarios, it isn't sensible in the less refined real-environment setup. Additionally, a large caveat of it is the inability of detecting curved lines, which is a crucial aspect for the project group, and thus necessities the development of another approach.

8.2.2 AI Enhanced Implementation

In contrast to the first approach, the AI-driven approach solves the problem of lane detection using a pre-trained AI model, specialized in the detection of lane lines on highways. The followed approach, called "Ultra Fast Lane Detection" (UFLD) uses a ResNet-18 based model trained on the TUSimple dataset [44], which consists out of 6408 road images on US highways with the resolution

1280×720 pixels. UFLD delivers a high-performance solution to lane detection suitable for the computational constraints of this project. The models' architecture is described in the paper [39] and the authors have provided code for it in [13].

An important consideration when researching the feasibility of the usage of an AI model for this task was the limited computational power accessible. Since the project requires for the computation to run on a Laptop CPU or on the TurtleBot itself, most available AI models are not suitable. A solution for this issue is provided in [22], which offers an implementation of UFLD in the ONNX format, while the ONNX version of the model itself can be downloaded from a pretrained model collection [37]. ONNX is an open source library, which amongst other things provides a hardware optimized format for AI models, allowing performant inference on CPUs. The lane detection process in this approach works as follows (Figure 8.3):

1. Cropping: Cropping of the captured images to a required aspect ratio.
2. Inference: Passing of cropped images to the AI model, which detects the lane lines and returns their coordinates Figure 8.4.
3. Bird's-eye View Transformation: Transformation of the coordinates for further use Figure 8.5.
4. Lane Line Extension: Extrapolation of the detected lines.
5. Lane Data Calculation: Estimation of the TurtleBot's current lane and the positions of adjacent lane boundaries, followed by scaling of the detected coordinates.

The last three steps remained the same as in the initial approach, as they are independent of the actual lane detection method.

8.2.3 Preconditions

For optimal performance, the road texture used for the Gazebo simulation of a highway environment has been updated with a higher resolution one. Additionally, the camera height and angle have been adjusted to match the perspective of the images in the TuSimple dataset for optimal detection. Furthermore, the captured images with the size of 640x480 pixels (height, width) are cropped to the resolution 640x360, which is a multiple of the resolution of the images in the TuSimple dataset (1280x720). These aspects influence the model's ability to discern lane lines and present a challenge for detection when using the actual TurtleBot, as the real-world camera setup and environment are not as easily adjustable as the simulated counterparts.

8.2.4 Bird's-eye Perspective Transformation

In order for the detected lane line coordinates detected in images to be compatible with the internal coordinate system, a Bird's eye view perspective transform needs to be applied to them. This is done by using predefined source points and destination points for the transformation, which have been manually established

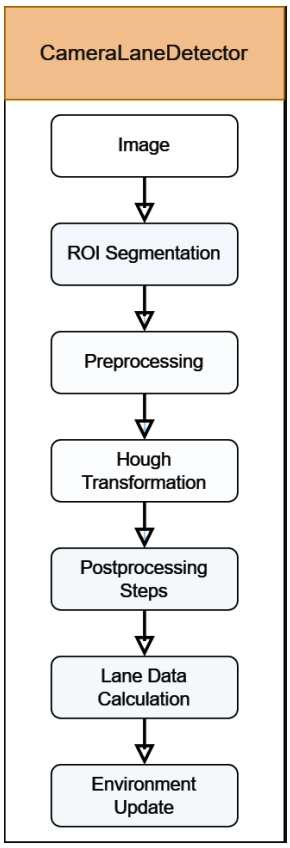


Figure 8.2: Stages of the lane detection in the original approach.

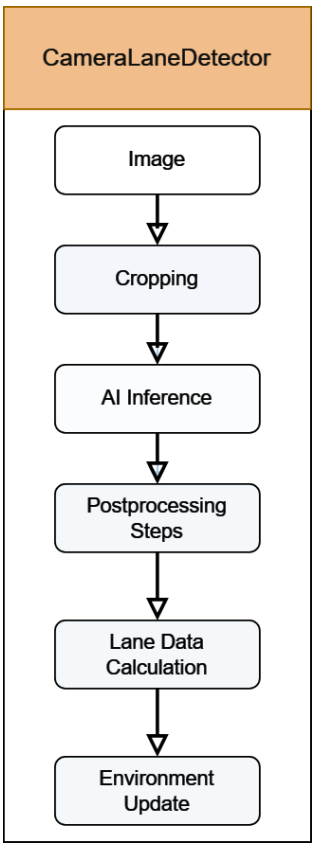


Figure 8.3: Stages of the lane detection in the AI enhanced approach.

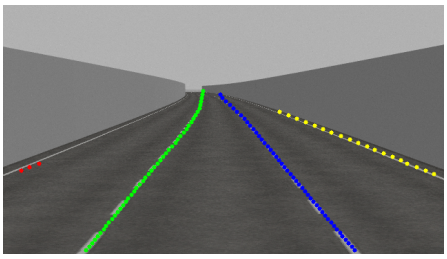


Figure 8.4: The lane lines detected by the AI model.

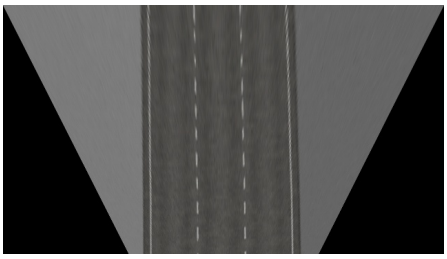


Figure 8.5: The image after the perspective transformation.

using the images captured by the camera. As the real camera and simulated camera differ, the adequate camera lane detection configuration must be loaded. Using the set points, a transformation matrix is computed and applied to the detected lane line coordinates to acquire the Bird's eye view coordinates. The implementation of this functionality is done based on [18].

8.2.5 Lane Data Processing

Lane lines below a certain length are filtered, as short lane lines detected by the model tend to be more inaccurate, based on performed experiments. Since the camera perceives certain distance in front of the TurtleBot (varying on the camera's field of view), the lane lines are extrapolated "backwards". This is done because the lane coordinates at the location of the TurtleBot are required for further calculations. Finally, the lane line coordinates are rescaled to match the dimensions of the internal coordinate system.

8.2.6 Current Lane and Boundary Calculation

The current lane of the TurtleBot is determined by subtracting the amount of lane lines found left of the TurtleBot from the defined lane count. The TurtleBot's position within the lane is similarly calculated using the defined lane width and the distance of the TurtleBot to the closest left and right lane lines. Both of these calculation use the processed lane coordinates.

8.2.7 Advantages and Limitations

The camera-based approach is an alternative to the LIDAR-based lane perception and has advantages such as not requiring a border wall next to the road for it to work. Additionally, it is able to predict coordinates of lane lines which are partially obstructed by objects such as other vehicles on the road, which the LIDAR approach cannot.
However, this approach is not perfect as the nature of the AI model makes using it to acquire reliable lane information difficult. The detection is affected by various factors such as lighting conditions or reflectivity of the road. For instance, as depicted in figure Figure 8.3 the left lane line is barely detected, even though it is visible. This inconsistency may originate from the pretrained models training data, which consists of images captured using a camera with a different field of view and resolution. Additionally, the environments used in the project deviate from the environments in the models training data, these factors likely contribute to the observed discrepancies as the model has limited generalization capabilities. It was found through testing that the lane lines making up the boundaries of the current lane the TurtleBot is on, are detected most consistently, whereas detection of other lane lines is less reliant.
A consideration to be made is the retraining of the AI model using the same dataset (TUSimple), but adjusted for the available camera. This improvement should allow the model to provide better predictions for the images captured by the camera used in this project and thus increase the quality of the lane detection.

Chapter 9

Object Detection

This section presents the methodologies employed for object detection in the project, showcasing camera-based and LIDAR-based techniques. The implementations of these methods in both simulated and real-world scenarios are described and their setup, advantages, and limitations within the context of this project are explored.

9.1 Camera Based Object Detection

Various ways of camera-data-based object detection have been researched and evaluated as a part of the project. Considerations included training an object detection AI model (such as YOLOv7) or using traditional computer vision algorithms. The AI based approach, often involving convolutional neural networks (CNNs), is known for achieving high accuracy in object detection. However, its complexity and (usually) resource-intensive nature makes it less ideal for the scope of this project. Traditional computer vision algorithms offer a less computationally demanding alternative, but they might require additional conditions to be met.

This project uses a classical approach, enabled by the fact that full control over the environment and the TurtleBot is available: Fixed synthetic markers with a known layout. Strategic placement of such markers on to-be-detected object ensures optimal visibility and ease of detection. The simplicity of this approach translates into faster processing on the TurtleBot, making it an efficient solution. Moreover, the required preparation of the markers on the detectable objects leads to the natural elimination of possible false-positive object detections.

9.1.1 Marker Systems

During research of object detection using fixed markers, two prominent state-of-the-art marker systems came into focus: AprilTag [2] and ArUco [35]. Both methods employ square markers, based on a visual bit representation of unique ID patterns. They facilitate rapid and accurate identification, making them suitable choices for applications with limited resources.

AprilTag markers are slightly more complicated to generate than ArUco markers, however, pre-existing repositories that contain different AprilTag-families eliminate the need for individual marker generation. Additionally, the detection of AprilTag markers is implemented in a ROS package, which made it seem like a preferable choice considering that the TurtleCar application already uses ROS2. However, the AprilTag package is not fully migrated from ROS to ROS2, which complicates its actual utilization and would require additional debugging. On the other hand, the ArUco marker system is included in the OpenCV python package, which is already used by the TurtleCar application. Integrating the generation and detection of ArUco markers was therefore straightforward for the existing architecture and was the preferred choice.

9.1.2 ArUco Marker

The ArUco library allows the detection of ArUco markers along with their distances. The requirement of making the marker detection as reliable as possible, rendered the ArUco marker family `DICT_4X4_50` the most promising. 4×4 indicates the bit size, whereas 50 is the number of available markers in that family. Choosing the minimum in both regards ensures a maximum Hamming Distance between the marker IDs and better overall detectability.

9.1.3 Environment Preparation

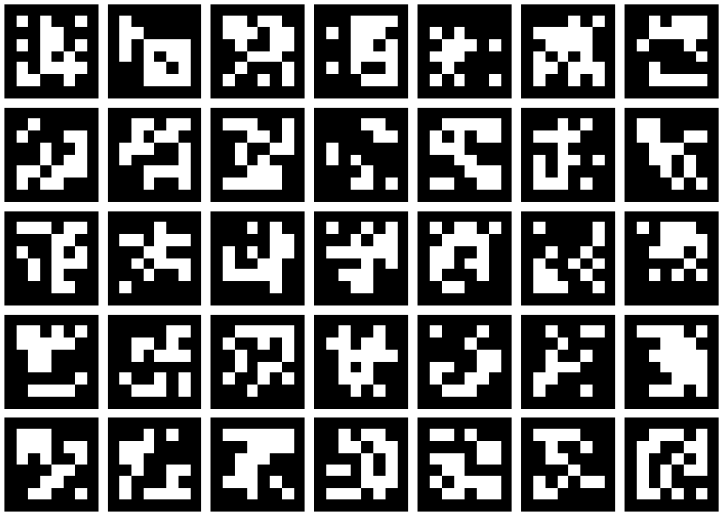
For the successful usage of fixed markers, preparation of the controlled environment is necessary in both simulation and reality. For that, all marker IDs available in the chosen marker family are mapped to an object type:

- IDs 0-9: other TurtleBots
- IDs 10-49: cuboid obstacles

The placement of markers follows a respective strategy for the two different object types: (1) TurtleBots are equipped with a single marker positioned on their backs, serving as a distinctive „license plate“, (2) cuboidal obstacles require four distinct markers, placed on each corner of their detectable faces. Using four markers instead of one enables the accurate calculation of the obstacle's size and rotation along with the distance.

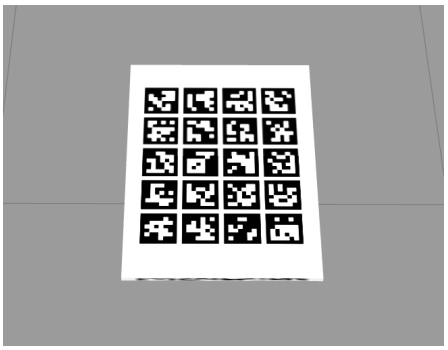
To achieve optimal precision in both the simulated and the real environment, an initial one-time camera calibration process needs to be conducted. This process obtains camera parameters, which can then be utilized for camera based detection tasks. For the calibration, a special board with ArUco markers that can be seen in Figure 9.1a is used. With a set of roughly 50 images that capture the calibration board from different angles, the ArUco library is used to acquire parameters of the camera, which are then saved to a config file for further usage. Examples of such images can be seen in Figure 9.1b and Figure 9.1c.

Additionally, to allow the object detection to accurately calculate distances to the markers, cohesive marker sizes withing reality and the simulation are required. To determine the optimal marker sizes, markers were experimentally printed and attached to the respective objects in reality. TurtleBots in the real environment impose some restrictions, as their license plate should not restrict any other sensors or the overall mobility of the TurtleBot. After the optimal



(a) Aruco board used for camera calibration.

(b) Example image used for the camera calibration in reality.



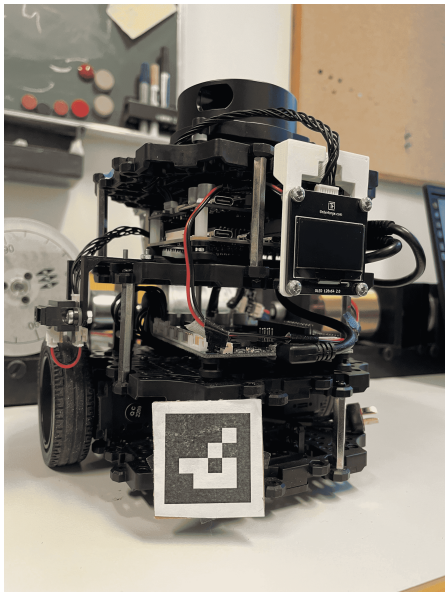
(c) Example image used for the camera calibration in simulation.

Figure 9.1: Camera calibration for ArUco detection.

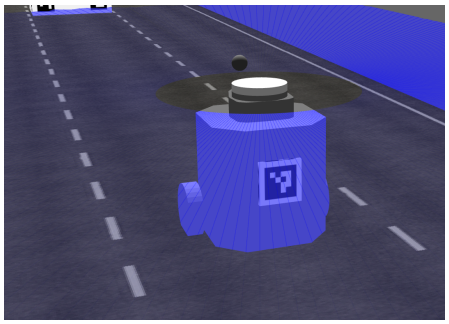
marker sizes were established, fitting 3D models could be created. The resulting layout for the TurtleBot markers can be seen in Figure 9.2a and Figure 9.2b and those for the obstacles can be seen in Figure 9.3a and Figure 9.3b.

9.1.4 Marker Detection

The actual marker detection process is mostly independent of whether the environment is simulated or real, apart from the correct camera calibration that has to be loaded. The detection simply uses the available camera images and leverages the ArUco library to detect all markers in the image. After that, the same library is used to calculate the poses and distances of the individual markers. For each one detected, the object type and the transform/rotation vector are recorded in the TurtleCar-Core Model.



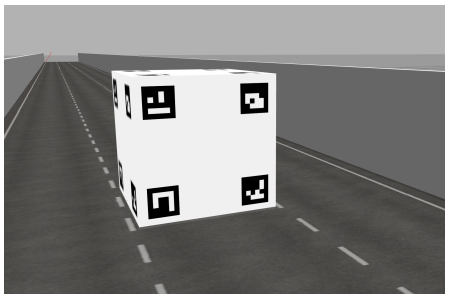
(a) TurtleBot marker in reality.



(b) TurtleBot marker in simulation.

Figure 9.2: TurtleBot marker layout in both environments.

(a) Obstacle marker in reality.



(b) Obstacle marker in simulation.

Figure 9.3: Obstacle marker layout in both environments.

Chapter 10

Path Planning

In this section, the planning of paths for a vehicle is described. First, Path Planning and Trajectory Planning are defined, and context to these topics within the project group is given. Next, the implementation of Path Planning is explained. Also, the end of the section contains guidance for building and integrating a custom path planning module.

10.1 Definitions and Context

First, Path Planning is differentiated from Trajectory Planning, and an introduction to what both of these terms mean in the context of the project group is given. For further references and mathematical function definitions, see *Trajectory Planning* [36].

Path Planning A path P is a continuous function which connects a start q_{start} and a goal q_{goal} in a coordinate system. Therefore, the domain of P is $[0, 1]$ and its co-domain is $\mathcal{C}$, i.e. the coordinate space that is used. P is devoid of any time information, and only resembles the geometric component. When enriching it with time information, it becomes a trajectory [36]. For us, planning a path means to plan out a geometric ordered list of points that the robot should follow, disregarding any time information.

Trajectory Planning A trajectory Π is a path P endowed with a time parameterization s. s is a strictly increasing function, which gives the position on the path for each time instant t. Thus, the same path P can give rise to many different trajectories Π [36]. For us, planning a trajectory means to take into account time information to the planned path.

At the current state of the project group, trajectories are not planned, only paths. Planning trajectories would involve many more considerations, which have not been prioritized as of now.

10.2 Implementation

The architectural overview of the path planning implementation can be seen in Figure 10.1. It closely resembles Figure 6.1, but elaborates more on the path

planning part.

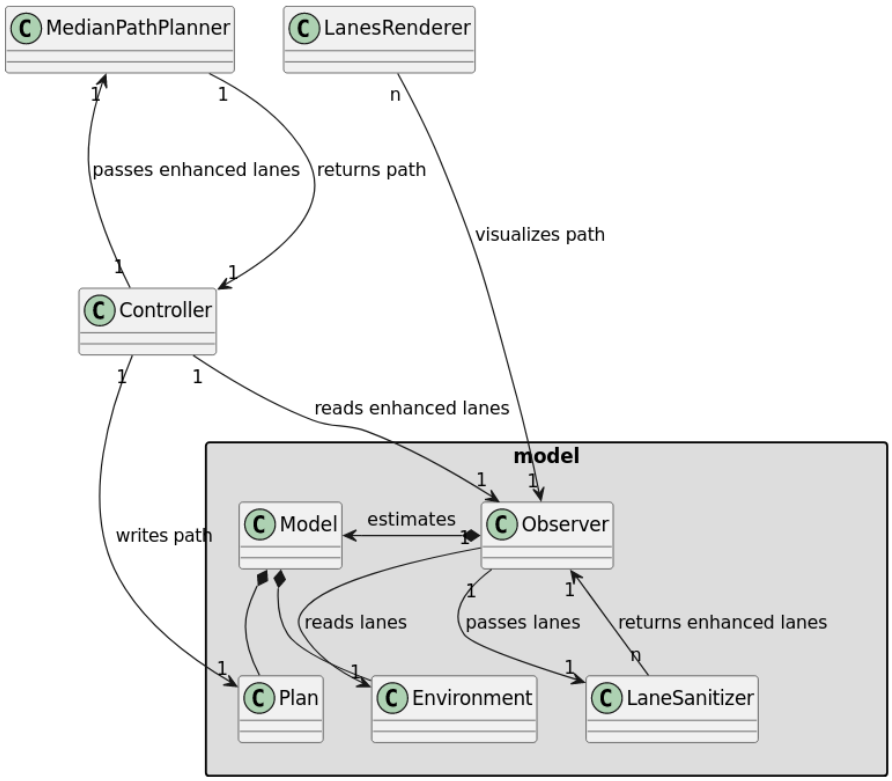


Figure 10.1: Architectural overview with the path planner.

10.2.1 Processing the Lane Data

Since the original sensor readings contain incomplete and not ready-to-use data, the data is processed before usage. *Enhancing* the lanes refers to enhancing the lane data provided by the sensors with i. e. interpolation, and is done by the `Observer`.

10.2.2 Lane Format

Each lane is represented by two borders (left and right), each of of the lane borders consists out of multiple detected lane points.

10.2.3 Enhancing the Lanes

During processing of the lane data the border points are interpolated using a Euclidean distance formula. The formula is the following, using points p and q:

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

When the distance of two consecutive points in the border list exceeds a parameterized threshold, new points are interpolated in between them, also

using this distance formula. There is no formalized threshold defined, the results of different parameters are tested empirically.

10.2.4 Planning the Path

Now that the lanes contain enough data points to use, a plan for the path for the robot to take can be created. In order to do that, the middle of the border points from the enhanced lane data is calculated and thus creates a path along the center of a lane. The path is visualized via the `LanesRenderer`.

10.2.5 Example images

In this section, example images for the path planning module are demonstrated. The snapshots are taken directly from the debugging tool, where orange points visualize paths and yellow points indicate lanes.

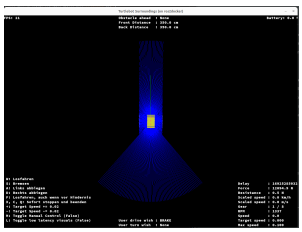


Figure 10.2: An ordinary path

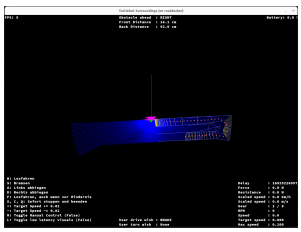


Figure 10.3: A more complex path

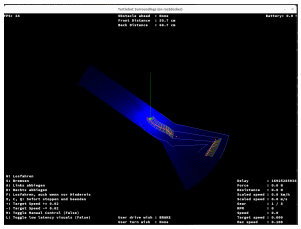


Figure 10.4: Lower border interpolation resolution

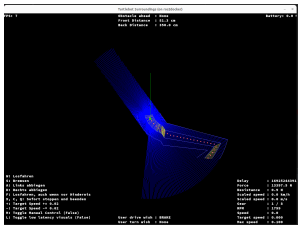


Figure 10.5: Higher interpolation resolution for the border

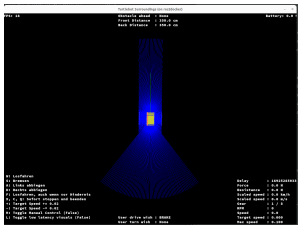


Figure 10.6: Higher sample rate and offset to the left of the border

Chapter 11

TurtleCar-Test

TurtleCar-Test is a test framework developed by the project group to test the implemented driving functions on a TurtleBot. In collaboration with Gazebo, it allows interactive, headless, scenario-based testing. In this section, related work is presented and then the architecture of TurtleCar-Test based on requirement analysis explained. Furthermore, the creation of tests is explained.

11.1 Traffic Sequence Charts

Traffic Sequence Charts (TSCs) proposed by Damm et al. [15] provide a specification language for defining test scenarios for autonomous vehicles. It allows capturing of their behavior in all possible traffic situations. A TSC specification consists of a world model that defines classes of objects (e. g. cars) together with their attributes (e. g. position and velocity) and the dynamics of moving objects, represented by a set of snapshot charts. A symbol dictionary links graphic symbols to their respective objects in the world model. These charts can then be translated to formula in first-order multi-sorted real-time logic.

Snapshot charts describe the evolution over time (e. g. via a snapshot sequence) and are used to visually depict potential traffic situations. They may contain

- present objects,
- relative placements of objects,
- defined absolute distances between objects,
- timing constraints, and
- so-called somewhere- and nowhere-boxes to specify the presence of an object somewhere inside an area or the absence of an object inside a certain area.

Also, they can be composed of premises and consequences. An example snapshot sequence is given in Figure 11.1.

The first two snapshots in the dashed hexagon on the left define the premise, the consequence is specified via the snapshots right of the hexagon. The premise denotes that there is an obstacle in front of a car. Both are on the same lane

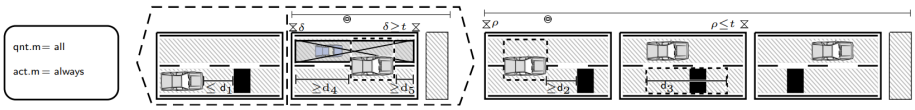


Figure 11.1: Change lane to avoid collision, if the next lane is free [15]

and less than $d1$ meters apart. From $d4$ to $d5$ there is no car to the lane left of the car (second snapshot). If the premise is fulfilled, the car has to change lanes to avoid collision (last three snapshots).

The TSCs exhibit possibilities to (relatively) place objects, define their attributes (e. g. velocity), define areas and denote premises and expected consequences. These possibilities were also made available in TurtleCar-Test. However, the TSC specification language was not adopted for Turtlecar-Test specifically, since the effort is considered too high while offering no significant advantage in terms of comprehensibility. Even so, in the future, there could be an extension to Turtlecar-Test that allows for the generation of test cases for the testbed from logical formulas generated by TSCs.

11.2 Architecture

Figure 11.2 depicts the simplified architecture of TurtleCar-Test. TurtleCar-Test introduces two main features for writing and executing tests: the Trigger-System (see subsection 11.2.1) and a Simulated Driver (see subsection 11.2.2). The Trigger-System receives information from the Gazebo simulation and executes specified actions when specified conditions are met. The Simulated Driver acts as a human driver would and has control over steering, throttle, and toggling automated driving functions. The Trigger-System and the Simulated Driver in collaboration ensure that the dynamic behavior can be defined in a way that the test requires. Those two features are further explained in the following sections.

11.2.1 Trigger-System

The Trigger-System evaluates data coming from the Gazebo simulation system. It contains an internal state for formulating complex testing conditions. The system's structure is visualized in Figure 11.3

The Trigger-System evaluates Gazebo data based on user-generated Triggers. When writing the testing specification, the programmer can formulate testing conditions and their resulting actions in the form of those Triggers. A Trigger is made up of three components:

1. state (optional)
2. condition
3. action

The Trigger-System first checks if the Trigger has a required state. If so, it is compared to the current state of the system and only if it matches will the following condition be evaluated. Checking the condition is based solely on the

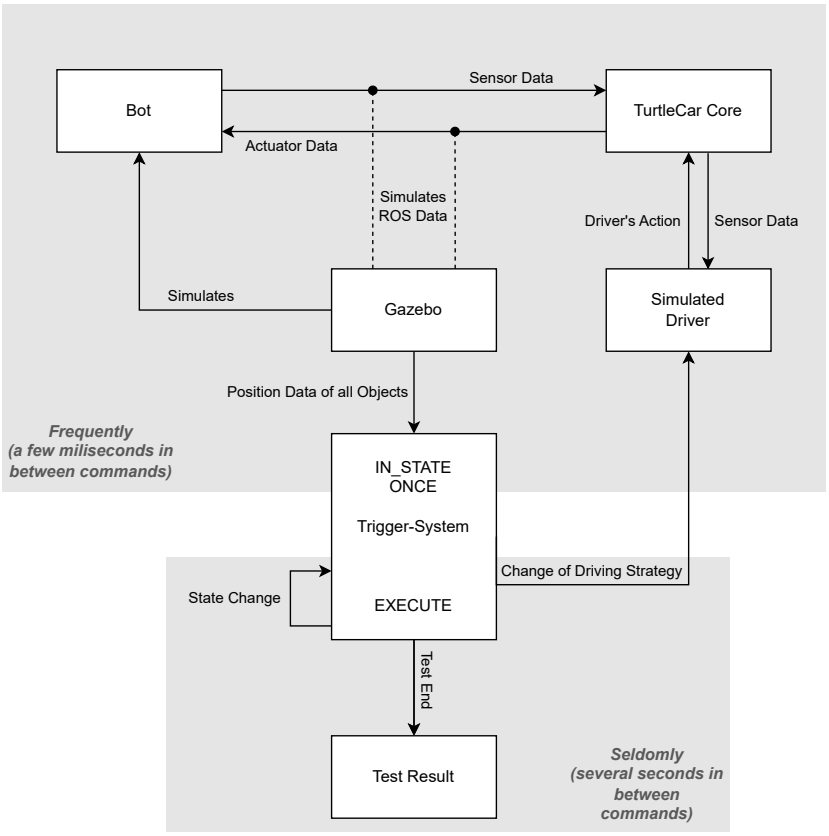


Figure 11.2: Architecture of TurtleCar-Test

data given to the Trigger-System by Gazebo. This way, it is decoupled from TurtleCar-Core's internal state. If TurtleCar-Core behaves in an unexpected way - e. g. by outputting the wrong velocity values - the test-outcome is still based on the observed behavior of the TurtleBot.

The condition is a boolean condition that can be formulated with the helper methods provided by TurtleCar-Test. Conditions allow checking positional and temporal constraints, as well as other aspects of the robot under test. That includes but is not limited to conditions over

- the absolute position,
- the velocity,
- the steering angle,
- the activated driving functions,
- the time elapsed between certain points in the test procedure,
- the distance to other objects or actors, and
- the position in relation to other objects or actors.

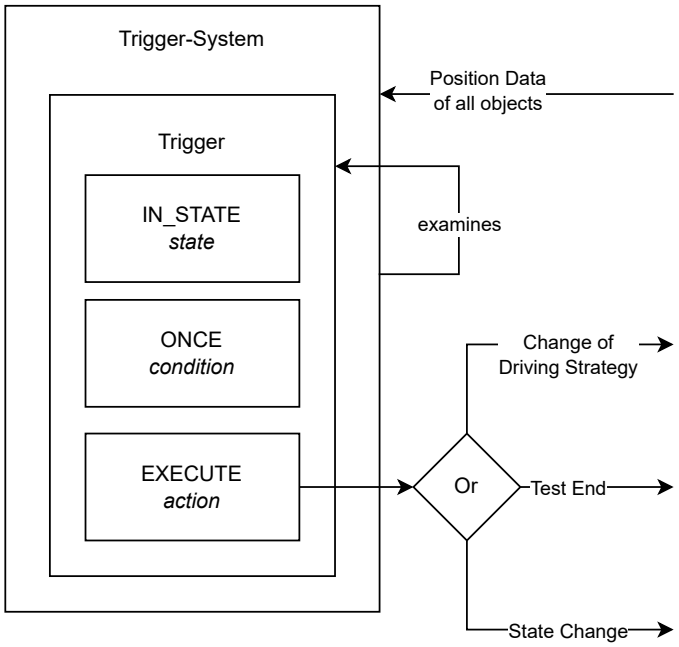


Figure 11.3: Structure of the Trigger-System

If the condition holds, the Trigger's action is executed. This action either changes the state, outputs a test result or changes the driving strategy of a bot. The Triggers can be grouped by the type of action they Trigger.

11.2.2 Simulated Driver

The Simulated Driver has access to the same controls that a human behind the steering wheel of a car has. It can control acceleration, steering, and toggle autonomous driving functions.

As part of the Testbed, this module takes care of keeping a set speed when no cruise-control system is active. It can be adjusted by the Trigger-System by way of a `DrivingFunctionChange` or a `SimulatedDriverSettingsChange`.

11.2.3 Gazebo integration

The testbed relies heavily on Gazebo for providing the simulation environment and the positional data of each object within the simulation to evaluate conditions against. The Gazebo simulation system is started at the start of each test and shut down after its completion. During the test, it provides the positional data via the ROS topic `/tf`. Data provided by this topic is absolute and has no inaccuracies, unlike the simulated sensors that the simulated robots possess.

11.3 Defining Test Cases

A test describes a scenario with its desired outcome. Optionally, a series of in-between states can be included for more complex tests. To define a test case, one

must be able to define an environment, an actor that acts on the environment and at least one criterion to be checked for.

TurtleCar-Test defines the environment via a gazebo map representing an empty road in the form of a .world-file and a custom .areas-file holding coordinates and ids of areas occupied by lanes, obstacles, road signs, and other relevant locations within the world.

As an actor, the robot model with its available hardware needs to be defined. This is done via the model's .sdf-file. Furthermore, every actor needs a unique ID and namespace to communicate with its instance of TurtleCar-Core. It also needs a starting state, including its pose (x, y, z and rotation), velocity, and active driving functions. An agent's state can change over time; It can accelerate or (de-)activate a driving function, for instance. TurtleCar-Test allows this behavior by defining a driving strategy for an actor through a Trigger.

A criterion to be checked for can be defined by formulating a Trigger. This criterion-Trigger defines when to give what test result. To simplify, this can state that if two cars crash, then the test results in an error.

To fully define and execute a test case, the following steps have to be taken:

1. Description of the environment.
2. Description of robot(s).
3. Declaration of relevant areas.
4. Definition of Trigger.
5. Start of the test.

11.4 State Machines

State machines can be used to separate different parts of a test scenario. They can be used to limit which conditions are tested for in different parts of the test. A state machine can be specified in the test scenario specification file. It requires a list of states and a starting state as parameters. Every Trigger can include an optional check for the current state of a state machine. Such a Trigger's condition will not be checked until the expected state is reached. The state machine can be put into the following state or a specific state in a Trigger-Action.

11.5 Timers

A timer can be specified in the test scenario specification file. It can be started, stopped, and reset as the action of a Trigger. The current amount of time passed can also be used as a condition in a Trigger.

11.6 Implementation of TurtleCar-Test

The implementation is straightforward and aligns with the description of the functions given above. The startup procedure is shown as an activity diagram in Figure 11.4. It begins by starting its own ROS node for publishing and

subscribing to topics. This is used for the communication between robot, simulation, and TurtleCar-Test itself. It additionally starts the `TransformListener`, which specifically subscribes to the messages sent by Gazebo via the aforementioned `tf` topic containing information about the pose of the objects in the simulation, to obtain the position data of all simulated objects. Afterwards, the processes necessary to conduct the tests are started in the order given in the diagram. The visualization components are only started if the user sets the corresponding flag when starting the testbed.

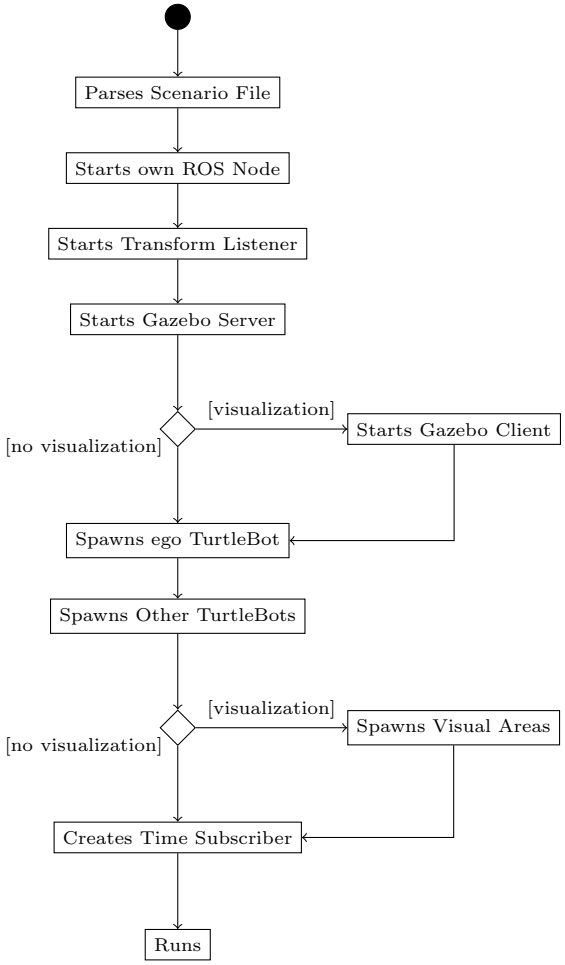


Figure 11.4: Activity diagram of the startup of TurtleCar-Test

After startup, TurtleCar-Test loops through every Trigger and checks their state and the condition. If both are true, the associated action is executed. The action can range anywhere from a small adjustment in the SimulatedDriver to ending the test with a success or a failure outcome. The state associated with a Trigger is only re-checked when said state changes, while the Trigger-condition is checked every simulation tick. Once a Trigger's action is executed, the Trigger will be discarded. If the action of a Trigger is a `TestOutcome`, then the testbed will stop the Gazebo and TurtleCar-Core processes, and exit with

a return message and a corresponding exit code.

11.6.1 Calculating Velocities from Gazebo Pose Data

The Gazebo simulation only sends information about poses. To obtain velocities, the `TransformListener` calculates the velocity of a bot from the distance the bot traveled between the last two simulation steps. When the time between the steps is too small, this can result in wrongly calculating very high velocities. To avoid this problem, the velocity is only recalculated after at least one second.

Chapter 12

Architectural Concepts

In order to implement the various driving assistance function in such a way that they can be combined as would be the case in a real car, there is a need for an architecture that supports this. In this section, several architectural concepts are described that form the basis of implementing the individual driving functions. First, a structure is introduced that binds individual driving functions together to form the autonomy levels described in section 1.1. Afterwards, the usage of model predictive control (MPC) within this project is described, which is used to implement certain driving functions.

12.1 Autonomy Level Architecture

This section describes two use cases which are derived from the vision (see section 1.1). The first use case combines the autonomy levels "no autonomy" and "partially automated", while the second organizes the driving functions for highly autonomous driving.

12.1.1 Manual Driving and Partial Autonomy

This section describes the usage of the first two autonomy levels, Manual driving and partial autonomy, since they are closely related and implemented in one consistent architecture. The general principle is that the vehicle always starts in manual driving mode. Afterwards the human driver has the ability to manually activate assistance functions or deactivate them again. The inputs that can be given by the driver are shown in Table 12.1. The table shows the input mappings for keyboard and gamepad. Additionally, the inputs can be given via ROS parameters for automatic testing.

The interplay of the manual and assisted driving functions can be modeled as a pipeline. The architecture is described in Figure 12.1. The supervisor module contains an ordered list of control modules for driving functions which can be turned on or off. Each iteration of the control loop, it first asks the manual control function (described in section 13.1) for its input. Afterwards, each controller in the ordered list that is switched on is asked for its input. Each controller writes its output directly to the `Action` interface. The next controller therefore has the possibility of overriding or altering input from the

Table 12.1: The keyboard and gamepad inputs for a human driver.

Function	**Key**	**XBOX Gamepad**	ROS Parameter
Increase Velocity	W	Press Right Trigger	`user_input_target_speed`
Decrease Velocity	S	Release Right Trigger	`user_input_target_speed`
Steer Left	A	B	`user_input_steering_angle`
Steer Right	D	X	`user_input_steering_angle`
Toggle Lane Keeping	K	Y	`user_input_toggle_lka`
Emergency Stop	Space	Left/Right Shoulder Button	No parameter, send a message of type Twist with all zero values to topic `\cmd_vel`

previous controller. Some controllers may contain a planner which writes its planned path to the `Plan` interface. This may also be overridden by a subsequent controller. To avoid confusion, only one controller should therefore be allowed to develop a plan. In the future, this may be replaced by a using global planner, which develops a plan for all driving functions or by integrating multiple driving functions into one unit. Therefore, the pipeline architecture described here is subject to change. Currently, the pipeline consists of the following controllers in this order:

1. Manual Control
2. Lane Keeping Assistant

These controllers are described in detail in the following sections.

12.2 Model Predictive Control

In this project group, the aim is to build driving functions which enable solving different scenarios. In order to achieve this in a way that consistently provides good results, a two-layered approach for the control strategy was chosen. A planner calculates a trajectory and boundary conditions, which are then used by a model predictive control to derive the best possible solution. This approach is heavily based on the work of Wunderli [49] and of Kröger [29], who used model predictive control to achieve optimal tracking of a given trajectory with a model racing car. The approach is almost directly transferable as the model it uses is the same bicycle model used in the project group, as defined in section 4.1. This section aims to give an overview and guidelines on how new driving functions can be implemented by designing a problem consisting of a trajectory and boundary conditions which can then be solved by the model predictive control, resulting in optimal control signals. First, an overview over the Model Predictive Control approach used in this project group is given. This also includes information on where the project group deviated from the approaches described by Wunderli

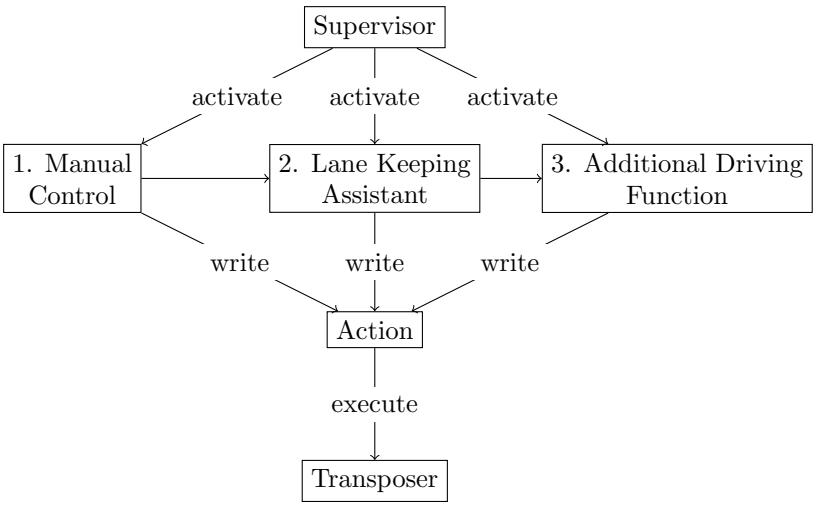


Figure 12.1: Interplay of supervisor and controllers for the individual driving functions.

and Kröger. Afterwards, an architecture is described which incorporates the model predictive control and implementation guidelines are given.

12.2.1 The Model Predictive Control Algorithm

Wunderli describes how to model the trajectory tracking problem as a quadratic problem which can be solved by existing solvers. Kröger deviates from Wunderli's original approach in some ways. These deviations were incorporated in this project group. It is assumed that a trajectory is given which can be tracked. The trajectory has the form

$$\vec{x}_n(t) = \begin{pmatrix} v_n(t) \\ \psi_n(t) \\ x_n(t) \\ y_n(t) \end{pmatrix}$$

where for each point in time t the nominal state $x_n(t)$ is completely defined. Additionally, for each point in time, the current trajectory curvature κ_n is defined. Instructions on how to derive such a trajectory are given below. Using this nominal trajectory and the dynamics of the bicycle model, a model of the error dynamics, i.e., of the deviations between the nominal and the actual trajectory can be derived. This model is linearized around $\psi_e = 0$, $v_e = 0$ while approximating $\cos(v_e) = 1$ and $v_e \sin(v_e) = 0$. The linearized error dynamics are then described by

$$\dot{x}_e = \begin{pmatrix} \dot{v}_e \\ \dot{\psi}_e \\ \dot{x}_e \\ \dot{y}_e \end{pmatrix} = \begin{pmatrix} a - a_n \\ \omega - \omega_n \\ v_e + \omega_n y_e \\ v_n \psi_e - \omega_n x_e \end{pmatrix}$$

with $\omega = \frac{v}{L}\delta$ and $\omega_n = v_n \kappa_n$. When discretized with time constant T, the error dynamics are defined by

$$x_{e,k+1} = A_k \cdot x_{e,k} + B \cdot \begin{pmatrix} a_k \\ \omega_k \end{pmatrix} + C_k \tag{12.1}$$

where

$$A_k = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ T & 0 & 1 & T\omega_{n,k} \\ 0 & Tv_{n,k} & -T\omega_{n,k} & 1 \end{pmatrix},\ B_k = \begin{pmatrix} T & 0 \\ 0 & T \\ 0 & 0 \\ 0 & 0 \end{pmatrix},\ C_k = \begin{pmatrix} Ta_{n,k} \\ T\omega_{n,k} \\ 0 \\ 0 \end{pmatrix} \tag{12.2}$$

In order to obtain the optimal control signals which minimize the error between the actual and nominal state for N time steps, the following optimization problem is defined:

$$\min_u \sum_{k=0}^{N} \begin{pmatrix} v_{e,k} \\ \psi_{e,k} \\ x_{e,k} \\ y_{e,k} \end{pmatrix}^T Q_k \begin{pmatrix} v_{e,k} \\ \psi_{e,k} \\ x_{e,k} \\ y_{e,k} \end{pmatrix} + \sum_{k=0}^{N-1} \begin{pmatrix} a_k \\ \omega_k \end{pmatrix}^T R_k \begin{pmatrix} a_k \\ \omega_k \end{pmatrix} \tag{12.3}$$

$Q_k \in \mathbb{R}^{4\times4}$ is the weighing matrix for the error states and $R_k \in \mathbb{R}^{2\times2}$ is the weighting matrix for the inputs. They contain constants on their diagonal which weigh the respective input and state variables. They can change each step, but in the simplest case they stay the same. Note that the weights for the velocity and the position should not be chosen to be the same, since the focus for the cost function needs to be set on either one. This optimization problem needs to be subject to the following constraints:

Error Dynamics The optimization must be conducted along the dynamics of the system, so per Equation 12.1 and the following condition must be met:

$$\forall k \in [0, N-1] : \begin{pmatrix} v_{e,k+1} \\ \psi_{e,k+1} \\ x_{e,k+1} \\ y_{e,k+1} \end{pmatrix} = A_k \begin{pmatrix} v_{e,k} \\ \psi_{e,k} \\ x_{e,k} \\ y_{e,k} \end{pmatrix} + B_k \begin{pmatrix} a_k \\ \omega_k \end{pmatrix} + C_k \tag{12.4}$$

with A_k, B_k and C_k defined by Equation 12.2.

Maximum Steering Angle The steering angle of a car is limited. Since not the steering angle itself, but the input $\omega = \frac{v}{L}\delta$ is used, the constraint on the steering angle is

$$\forall k \in [0, N-1] : \frac{v}{L}\delta_{min} \leq \omega \leq \frac{v}{L}\delta_{max} \tag{12.5}$$

Maximum Longitudinal Acceleration Because of restrictions on the vehicle's abilities, the acceleration must be constrained:

$$\forall k \in [0, N-1] : a_{min} \leq a_k \leq a_{max} \tag{12.6}$$

Maximum Lateral Acceleration For passenger convenience, the lateral acceleration $q = v\omega$ should not exceed a certain value. Because $v\omega$ can not be expressed in a linear MPC, it is assumed that $v \approx v_n$. Therefore the constraint is formulated as

$$\forall k \in [0, N-1] : v_n \omega_k \leq q_{max} \tag{12.7}$$

.

Position Constraints The space on which the car may drive is limited by several factors, i. e. the lane and road boundaries or obstacles. For now only lateral position constraints based on y are considered, but in principle obstacles could also be modeled by using constraints on x:

$$\forall k \in [0, N] : y_{k,min} \leq y_k \leq y_{k,max} \tag{12.8}$$

. For reasons of simplicity, the relaxation strategy implemented by Kroeger was not adopted in this project group. This may be subject to change if the MPC regularly fails to find a solution under the given constraints.

12.2.2 Implementation

The model predictive control is implemented by building the optimization problem from a given trajectory and given position constraints and solving it using a suitable solver for quadratic problems. In this project group, the Gurobi solver is used as it has been successfully used for similar problems previously (see [29], [7]). In contrast to Matlab, which also provides functions for solving optimization problems (see [33]), Gurobi can be used via a Python interface (see [21]) which makes it possible to quickly integrate it into the existing architecture.

Driving functions need to provide a plan and position constraints in a standardized format to the component building the optimization. The architecture is depicted in Figure 12.2. It shows how a driving function can obtain optimal control signals through a standardized interface by providing a trajectory and constraints to a problem builder component. The problem builder generates the optimization problem as described above and communicates with the Gurobi solver to generate a series of optimal control signals, the first of which is handed back to the driving function.

12.2.3 Implementation Roadmap

In order to implement this system, the following steps must be taken:

1. Obtain a Gurobi License and install Gurobi on the developer's computer
2. Create a minimal working example with a simple trajectory and boundary conditions and the optimization problem described above
3. Test the minimal example with the simulated TurtleBot
4. Install Gurobi on the TurtleBot
5. Test the minimal example on the real TurtleBot

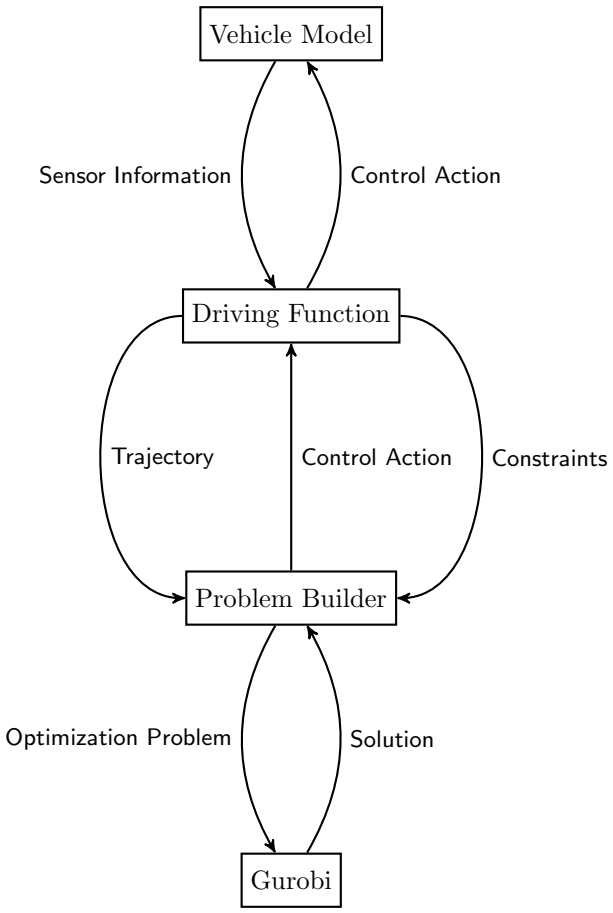


Figure 12.2: The architecture for intergrating model predictive control in TurtleCar by providing a problem builder as interface between driving function and optimizer.

6. Create an interface for the controller as described in Figure 12.2
7. Create a minimal controller using the interface to obtain signals from the MPC and test it in simulation and reality

Afterward, the resulting MPC interface can be used by various driving functions, such as the Lane Change Assistant or Obstacle Avoidance. In order to implement these, driving strategies and decision algorithms for choosing the correct strategies for the current situation must be implemented which create trajectories and constraints to use with the MPC.

Chapter 13

Driving Functions

In this section, the driving functions implemented by the project group are documented. These are used within the architecture described in chapter 12. For each driving function, the definition of the requirements, the controllers used to implement the driving function, and the validation are described. Some of these driving functions can't be active at the same time, as the controllers may provide conflicting control inputs. These conflicts are covered in section 13.8

13.1 Manual Driving

The manual driving controller controls the vehicle conforming solely to the driver's inputs. Since the focus of the project group is the development of autonomous driving function and not a realistic mapping of controls of a real vehicle, the user input is simplified. It consists of two inputs: A target velocity and a steering wheel angle. The manual driving controller gives these inputs directly to the `Action` interface. This means that the steering angle is set to be exactly the steering wheel angle given by the driver and the target velocity of the vehicle is set to be exactly the target velocity given by the driver. These values are then used by the `Transposer` to drive the vehicle accordingly. It is necessary to mention that the `Transposer` acts as a very aggressive cruise control with maximum acceleration and deceleration. In order to enable more realistic and smoother driving, the input of the driver or the vehicle model needs to be changed.

13.2 Lane Keeping Assistant

The Lane Keeping Assistant driving function should assure that the vehicle keeps in its lane. In Figure 13.1 the ego vehicle is located on the middle lane and driving. In a scenario with the LKA activated the vehicle should keep the same distance to the lane markings on each side - so it should drive centered in the lane it starts in.

The requirements of the Lane Keeping Assistant are based on the ISO standard 11270 [23], but do not yet cover all aspects. These requirements are subject to rework.

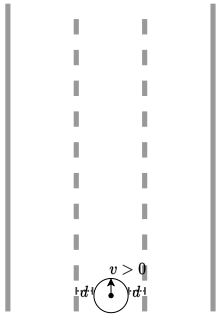


Figure 13.1: Lane Keeping Assistant scenario

13.2.1 General Requirements

- The LKA must be able to be switched on or off
  - The LKA can be toggled by user input
  - The LKA can be switched on by startup flag
- The robot must identify the lane its on and its center
- The LKA must be disabled when the lane change disabling condition according to the ALKS regulation is fulfilled
- The LKA enabled vehicle should never cross lane borders
- The robot should follow the lane's center
- The controller should be based on model prediction

13.2.2 Functional Requirements

Requirement LKA.1

GIVEN

- The vehicle is running

WHEN

- The input to enable the Lane Keeping Assistant is given

THEN

- The Lane Keeping Assistant is enabled

Requirement LKA.2

GIVEN

- The robot starts inside of lane boundaries

- The initial velocity is 0, the initial acceleration is 0, the initial steering angle is 0
- The robot heading fulfills the following criteria:
  - If the robot is left of the center of the lane, it faces in the direction it will drive, oriented within 0 and 40.2° toward the right lane boundary.
  - If the robot is right of the center of the lane, it faces in the direction it will drive, oriented within 0 and 40.2° toward the left lane boundary.

WHEN

- A target velocity greater than 0 is given by a human driver
- A steering wheel angle smaller than a threshold ω is given by the driver
- The Lane Keeping Assistant is enabled by the driver

THEN

- The LKA engages
- The robot identifies the lane it is on
- The robot accelerates to the speed defined by the human driver and maintains this speed
- The robot follows the center of the lane
- The robot never crosses lane borders
- The steering angle is always within the vehicle's specifications

Requirement LKA.3

GIVEN

- The robot is driving with arbitrary speed, arbitrary acceleration
- The Lane Keeping Assistant is active
- The steering angle is arbitrary within the vehicle's specification
- Initially there is no steering input from the user

WHEN

- A steering wheel angle greater than a threshold ω is given by the driver

THEN

- The LKA disengages
- The steering angle of the vehicle is the same as the steering wheel angle given by the driver

Requirement LKA.4

GIVEN

- The robot is driving with arbitrary speed, arbitrary acceleration
- The Lane Keeping Assistant is enabled
- The steering wheel angle given by the user is greater than a threshold ω
- The LKA is disengaged

WHEN

- A steering wheel angle smaller or equal to a threshold ω is given by the driver

THEN

- The LKA engages

Requirement LKA.5

GIVEN

- The LKA is enabled

WHEN

- The user input to disable the LKA is given

THEN

- The Lane Keeping Assistant is disabled

13.2.3 Non-Functional Requirements

LKA.A

The controller for the Lane Keeping Assistant is based on model prediction.

13.2.4 Additional Information

Overruling the LKA How an LKA can be overruled by the user differs depending on car manufacturer, model and available sensors and actuators in the car. In the manual for the EV6, KIA Motors describe that turning the steering wheel over a certain degree deactivates the Lane Keeping Assistant [26]. The deactivation criteria of the Lane Keeping Assist systems developed by Bosch depend on the availability of power steering: When available, the Lane Keeping Assistant actively turns the steering wheel and can therefore be overruled by the driver using enough force. [40]). Since the TurtleCar system does not contain force feedback inputs and adding support for these is out of scope for this project group, this is not possible to implement. In order to demonstrate temporary overruling for a lane change, the steering wheel angle threshold ω is defined. It was determined experimentally that setting ω to 1.7 degrees yields suitable results.

Determining the suitable initial conditions The maximum possible orientation is based on the most narrow curve that a car with the wheel base length and the maximum steering angle of a VW Golf. When in the center of the lane, the greatest angle it can recover from is 40.2°. This can be computed as follows:

- a: maximum steering angle (40°for VW Golf)
- w: wheelbase length (2.6365m for VW Golf)
- r: radius of turning circle
- $r = \frac{w}{\tan(a)}$

Since the outer set of wheels on the car is of interest when determining the size of the circle driven by the vehicle, half of the golf's width is added to the to the radius:

$$r_{golf} = r + 0.9$$

The maximum recovery angle based on that radius was determined graphically as shown in Figure 13.2. When placing a circle with radius of r_{golf} so that it touches the lane boundary, it represents the path that a vehicle would take in the most extreme, still manageable case. The tangent of the circle at the intersection of the center of the lane shows the orientation of the car in this extreme case when placed at the lane center. When at the left of the lane center, the car is therefore able to recover from orientations of 53.6° to the right or lower. The same goes for the situation that the car is right of the center and faces left. With a safety margin of 25%, this results in a maximum allowed orientation of 40.2°.

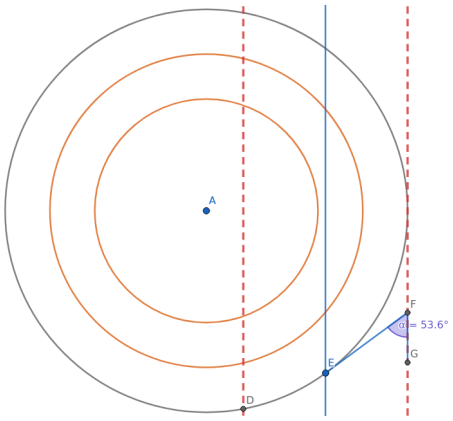


Figure 13.2: Graphical representation of the maximum heading pointing out of the lane that a car can recover from

13.2.5 Implementation

For the implementation, a controller based on the bicycle model is used. Since the bicycle model is a nonlinear differential equation, it is linearized in order to obtain a linear controller.

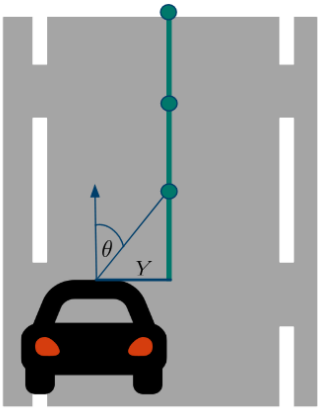


Figure 13.3: A graphical representation of the state variables used for the controller of the lane keeping assistant. Y and θ are always relative to the next point provided by the path planner.

Controller Model Since the Lane Keeping Assistant does not control the acceleration but only the steering angle, the velocity of the vehicle can be regarded as a parameter of the system. Therefore the bicycle model defined in section 4.1 can be reduced to the simplified version

$$\begin{aligned} \dot{x_1} &= \dot{Y} = v \cdot \sin(x_2) \\ \dot{x_2} &= \dot{\theta} = \frac{v}{l} \cdot \tan(u_2) \end{aligned}$$

where X is the position in the linear direction of the car, Y the lateral position, and θ the heading. These are all relative to the next point that the path planner provides. A depiction of the meaning of Y and θ can be seen in Figure 13.3.

This allows for a simpler linearization. The operating point to linearize around is $x = 0$ and $u = 0$. This represents the state where the vehicle is exactly on the line that has to be followed, and assumes that the controller only needs to make small corrections.

With that the system is linearized:

$$\begin{aligned} \dot{x}(t) = f(x,u) = Ax + Bu &\approx f(0,0) + \frac{\partial f}{\partial x}\Big|_{\substack{u=0\\x=0}} \cdot x + \frac{\partial f}{\partial u}\Big|_{\substack{u=0\\x=0}} \cdot u \\ &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2 s} \end{bmatrix}\Big|_{\substack{u=0\\x=0}} \cdot x + \begin{bmatrix} \frac{\partial f_1}{\partial u_1} \\ \frac{\partial f_2}{\partial u_1} \end{bmatrix}\Big|_{\substack{u=0\\x=0}} \cdot u \\ &= \begin{bmatrix} 0 & v\cos(x_2) \\ 0 & 0 \end{bmatrix}\Big|_{x=0} \cdot x + \begin{bmatrix} 0 \\ \frac{v}{l \cdot \cos^2(u)} \end{bmatrix}\Big|_{u=0} \cdot u \\ &= \begin{bmatrix} 0 & v \\ 0 & 0 \end{bmatrix} \cdot x + \begin{bmatrix} 0 \\ \frac{v}{l} \end{bmatrix} \cdot u \end{aligned}$$

The state feedback control law $u = -[k_1\ k_2] \cdot x$ is used to design the controller.

This results in the closed-loop function:

$$f_{cl} = \begin{bmatrix} 0 & v \\ 0 & 0 \end{bmatrix} \cdot x + \begin{bmatrix} 0 \\ \frac{v}{l} \end{bmatrix} \cdot (-[k_1\ k_2] \cdot x) = \begin{bmatrix} 0 & v \\ 0 & -\frac{v}{l} * k_2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

From the closed loop function it can be seen that k_1 can remain undetermined, as the lateral position has no direct influence on the control input. Now the operating domain for the velocity the Lane Keeping Assistant should be stable in has to be chosen. For this $v \in (0, 34]$ $^{\mathrm{m}}/_{\mathrm{s}}$ was chosen, which is $(0, 120]$ $^{\mathrm{km}}/_{\mathrm{h}}$.

Using Matlab the characteristic polynomial for which all eigenvalues have real parts strictly less than 2 was determined. With the coefficients, it is possible to solve for values of k_2 which hold the closed loop system in a stable domain. For this, the parameter space was sampled with a step size of 0.5. Since the system is uncontrollable for $v = 0$, sampling started at 0.5. This resulted in values for $k_2 \in [0.05, 1.4]$.

This enabled building a stable controller for the Lane Keeping Assistant. From the domain of stable values for k_2, choosing $k_2 = \frac{l}{v}$ has been determined experimentally to yield the best results for any speed v.

13.3 Adaptive Cruise Control

The Adaptive Cruise Control function should assure that a vehicle keeps a safety margin to a vehicle in front of it. In Figure 13.4 the ego vehicle is located on the middle lane and driving. In front of the ego vehicle is another vehicle, driving at least as fast as the ego vehicle. The distance between both vehicles is at least the safety margin distance at all times.

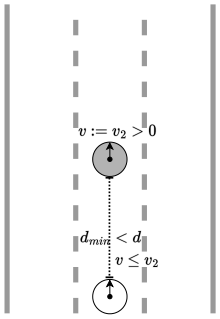


Figure 13.4: Adaptive Cruise Control scenario

13.3.1 General Requirements

- The vehicle must be able to activate/deactivate Adaptive Cruise control depending on the drivers wishes
  - The ACC must be deactivated if the driver takes manual control
- The vehicle must keep at least a minimum safe distance including a suitable error margin

  - The vehicle should keep the same speed as the front vehicle
  - The vehicle must be able to reduce its speed to keep the minimum safe distance
  - The vehicle should be able to increase its speed to keep the distance to front vehicle
- The driver can issue a command to drive at a certain speed overriding the ACC
- The ACC must be disabled if the driver issues a brake command
- The ACC controller should be build using model prediction

Definition: Minimum Safe Distance To be defined according to relevant regulations: Minimum safe distance which a following vehicle needs to maintain in order to be able to decelerate if the leading vehicle brakes (with bounds for deceleration)

In the German traffic regulations, a rule of thumb to determine the distance between two vehicles considered safe is described: The vehicle needs to keep a distance of at least half of the current speed (kilometers per hour) in meters [12, §2 Abs. 3a S. 2a].

13.3.2 Functional Requirements

Requirement ACC.1

GIVEN

- The ego vehicle is driving behind another vehicle. Both vehicles have arbitrary speed and the ego vehicle maintains at least minimum safe distance to the leading vehicle.

WHEN

- The driver triggers the switch for the adaptive cruise control

THEN

- The Adaptive Cruise Control is enabled

Requirement ACC.2

GIVEN

- The vehicle is driving behind another vehicle with arbitrary speed v_i with at least minimum safe distance including error margin

WHEN

- The adaptive cruise control is enabled

THEN

- The ego vehicle drives at most with velocity v_i
- The ego vehicle accelerates or brakes within the velocity bounds so that it maintains at least a minimum safe distance including error margin to the leading vehicle

Requirement ACC.3

GIVEN

- The adaptive cruise control is enabled
- The vehicle is driving behind another vehicle with arbitrary speed v_i and has at least minimum safe distance including error margin

WHEN

- The other vehicle brakes to velocity v_b

THEN

- The ego vehicle drives at most with velocity v_b
- The ego vehicle decelerates to velocity v_b and maintains at least a minimum safe distance without error margin at all times

Requirement ACC.4

GIVEN

- The adaptive cruise control is enabled
- The leading vehicle is driving with velocity v_i
- The vehicle is driving behind another vehicle with arbitrary speed v_i and has a a distance to the leading vehicle that is smaller than the minimum safe distance including error margin
- The driver does not give a command to accelerate to a speed greater than v_i

WHEN

- No Action

THEN

- The ego vehicle drives at most with velocity v_i
- The ego vehicle decelerates to until it maintains at least a minimum safe distance including error margin

Requirement ACC.5

GIVEN

- The adaptive cruise control is enabled
- The vehicle is driving behind another vehicle with arbitrary speed v_i and has at least minimum safe distance including error margin

WHEN

- The other vehicle accelerates to velocity v_a

THEN

- The ego vehicle drives at most with the minimum v_m of velocities v_i and v_a
- The ego vehicle accelerates to velocity v_m and maintains at least a minimum safe distance with error margin at all times

Requirement ACC.6

GIVEN

- The adaptive cruise control is enabled
- The vehicle is driving behind another vehicle with arbitrary speed v_i and has at least minimum safe distance including error margin

WHEN

- The driver continuously issues a command to drive with velocity v_t

THEN

- The ego vehicle accelerates to velocity v_t without regard for the minimal safe distance

Requirement ACC.7

GIVEN

- The adaptive cruise control is enabled

WHEN

- The relevant user input is received

THEN

- The adaptive cruise control is disabled
- The vehicle drives according to the driver's commands only

Requirement ACC.8

GIVEN

- The adaptive cruise control is enabled

WHEN

- The driver issues a braking command

THEN

- The adaptive cruise control is disabled
- The vehicle drives according to the driver's commands only

13.3.3 Non-Functional Requirements

ACC.A

The controller for the lane keeping assistant is based on model prediction.

13.3.4 Implementation

For the implementation, the approach described by Zhenhai et al. [50] is used. The function of the Adaptive Cruise Control is based on a switching strategy between two modes: In Cruise mode the vehicle simply keeps a given speed. In Follow mode the vehicle maintains a constant distance to the preceding vehicle and matches its speed if it is lower than the speed defined by the driver. The control law in each mode defines the acceleration a. For the description of the controllers, the following variables are used:

- v is the speed of the ego vehicle
- v_d is the target speed for the cruise control
- v_p is the speed of the preceding vehicle
- $\Delta v = v_p - v$ is the relative speed of the two vehicles
- $\Delta d = d - d_{\min(\Delta v)}$ is the distance of the ego vehicle to the closest safe point behind the preceding vehicle based on their relative speed
- a_{follow} is the target acceleration calculated by the follow mode algorithm
- a_{cruise} is the target acceleration calculated by the cruise mode algorithm
- d_{offset} is a parameter to define the offset between the zones where Follow mode and Cruise mode are applied

The control laws in each mode are described in Table 13.1. The constants k_p, k_i, k_v and k_d are selected so that the vehicle is able to achieve the desired state quickly, but without creating too much jerk in its movements. The chosen values are shown in table XYZ.

Table 13.1: The control laws of the Adaptive Cruise Control.

	Control Law
Cruise Mode	$a_{cruise} = k_p(v_d - v) + k_i \int (v_d - v)dt$
Follow Mode	$a_{follow} = k_v \Delta v + k_d \Delta d$

Table 13.2: The zone-based switching strategy from [50]

Distance	Velocity	Acceleration	Resulting Control Mode
$\Delta d \leq 0$	$\Delta v \leq 0$	-	Follow Mode
$\Delta d > 0$	$\Delta v < 0$	$a_{follow} \leq a_{cruise}$	Follow Mode
$\Delta d > 0$	$\Delta v < 0$	$a_{follow} > a_{cruise}$	Cruise Mode
$\Delta d < d_{offset}$	$\Delta v > 0$	$a_{follow} \leq a_{cruise}$	Follow Mode
$\Delta d < d_{offset}$	$\Delta v > 0$	$a_{follow} > a_{cruise}$	Cruise Mode
$\Delta d \geq d_{offset}$	$\Delta v \geq 0$	-	Cruise Mode

Each time a control signal needs to be generated, the controller first checks which mode should currently be applied. It then chooses the corresponding control law. This switching strategy is taken from [50] and is based on dividing the parameter space into zones in which the different control laws apply. This division is given by distance, velocity and acceleration. This method ensures that switching between modes is conducted smoothly. In favor of brevity, only the switching table is documented here without reasoning about the parameter zones. It is shown in Table 13.2. The table describes the conditions on distance, velocity and acceleration and defines the control mode that should be applied in each case. A „-“ means that the respective parameter does not factor in the decision in this case.

13.4 Lane Change Assistant

The Lane Change Assistant driving function should assure that the vehicle can safely change lanes. In Figure 13.5 the ego vehicle is located on the middle lane and driving. In a scenario with the LCA activated the vehicle should at some point in time drive on the left lane — having performed a lane change.

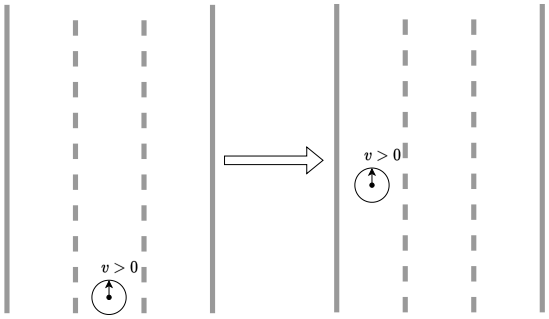


Figure 13.5: Lane Change Assistant scenario

13.5 Obstacle Avoidance

The Obstacle Avoidance driving function should assure that the vehicle avoids obstacles by stopping or changing lanes. In Figure 13.6 the ego vehicle is located on the middle lane and driving. In front of it — in a safe distance — there is an obstacle. In a scenario with the Obstacle Avoidance driving function activated, this distance is being kept and a lane change performed. The ego vehicle changes back to the middle lane at some point so that a safe distance from the obstacle is assured.

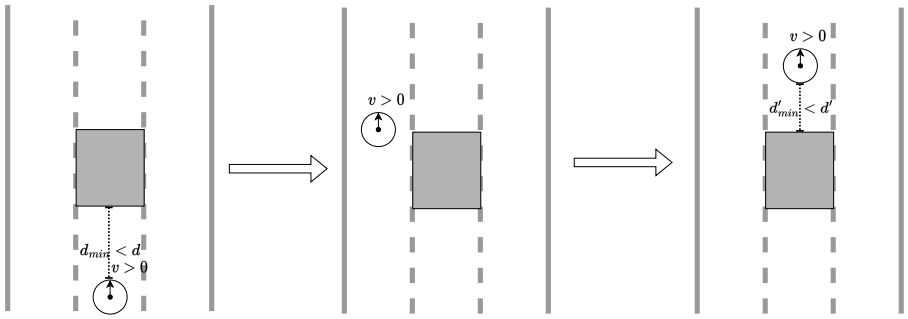


Figure 13.6: Obstacle Avoidance scenario

13.6 Overtaking

The Overtaking driving function should assure a safe overtaking maneuver when a moving vehicle is ahead. In Figure 13.7 the ego vehicle is located on the middle lane and driving. In front of the ego vehicle is another vehicle driving. The distance between both vehicles is at least the safety margin distance. In a scenario with the Overtaking driving function activated, this distance is being kept and a lane change performed. The ego vehicle changes back to the middle lane at some point so that a safe distance from the other vehicle is assured.

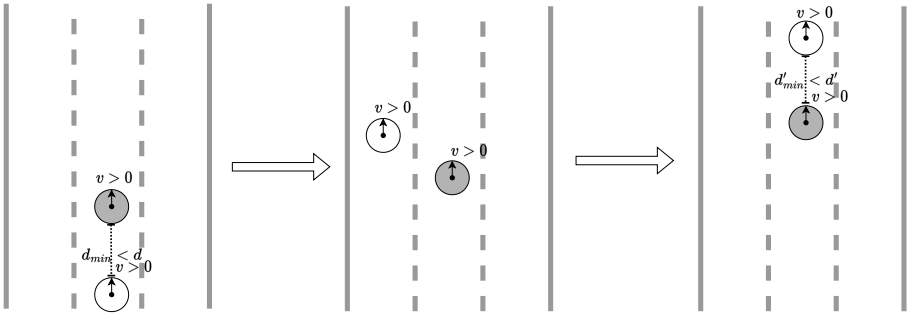


Figure 13.7: Overtaking scenario

13.7 Platooning

The Platooning driving function should assure that a convoy-like formation is created and kept. In Figure 13.8 the ego vehicle is located on the middle lane and driving. Behind it is another vehicle and behind that another. All the vehicle keep the same distance from each other that is within a predefined range.

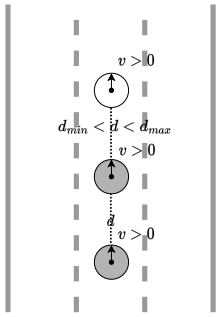


Figure 13.8: Platooning scenario

13.8 Constraints on Driving Function

Multiple driving functions are generally unable to be activated at the same time, because of conflicting control inputs. For example, the LKA wants to keep to the center of its current lane, while a LCA wants to move from the current lane to another. This section describes constraints on the simultaneous activation of driving functions. Since there are conceptual differences between an approach based on model predictive control and approaches that aren't based on MPC a distinction between these is made.

13.8.1 Classic Approach

Based on the implementations for different driving function, it is apparent which control inputs a given controller calculates. These inputs are then used to control the TurtleBot. This information provides a way to establish which driving functions can't be active simultaneously. Using this approach first results can be seen in Table 13.3. The + symbolizes that the functions can be used together, while the - symbolizes that they cannot. The black boxes means that they are the same function.

It is apparent, that most driving functions can't be used at the same time. An exception to this is the ACC which can be used in conjunction with one of the other functions that don't generate an input for accelaration.

However, this does not show the whole picture, as some driving functions behave similarly to others, like Overtaking and Obstacle Avoidance. At their core, these functions make the vehicle change the lane to drive past another object. In the case of Overtaking this object is also moving, while for Obstacle Avoidance it remains stationary. Additionally, both functions make the vehicle change back to its original lane after avoiding an obstacle. This behavior can be compared to using the LCA twice.

Table 13.3: Driving functions that can be active simultaneously

Driving Function	LKA	ACC	LCA	CAS	Overtaking
LKA		+	-	-	-
ACC	+		+	+	-
LCA	-	+		-	-
CAS	-	+	-		-
Overtaking	-	-	-	-	

The fully autonomous TurtleBot has to drive in a safe way, which implicitly establishes a hierarchy for the driving functions. The TurtleBot should use the Obstacle Avoidance or Overtaking driving function if there is an impending collision with another object, rather than using the ACC or LKA.

Platooning behaves a bit differently here because the leader of the platoon (the ego vehicle) is set to act using possibly all other driving functions, while the platoon members should just mimic the ego vehicle's actions and not use the functions themselves. This way the members aren't involved in planning actions, but just in the perception of fellow members and looking out for possible collisions with non-platoon objects.

13.8.2 MPC Approach

The MPC consists of two layers. First, there is a path planner and second, there is a path follower. Every function can be implemented using either layer, with different degrees of complexity. This changes how driving functions interact with each other. For example, the Obstacle Avoidance function can define the constraints for a trajectory to forbid „driving through“ an obstacle. This forces the path follower to drive around the obstacle automatically if the trajectory is still feasible. The other option is to replan the trajectory around the obstacle. This would mean that the main function of the Obstacle Avoidance is done in the planning layer. These ideas are the same for Overtaking.

The LKA and ACC only generate one control input, either steering angle or acceleration. This is different from what the MPC approach expects. It needs both inputs at all times. This leaves two options: Keep the LKA and ACC in their existing implementation and not use them in the MPC context. The other option is to rework both functions such that they also generate the missing input parameter in some way. This consideration has not yet been made; The status quo is the former option.

Another caveat for the ACC if approached in an MPC context, is that it requires constant monitoring of the lead vehicle. The acceleration of the ego vehicle based on the lead's vehicle speed and acceleration as well as the distance between the two should be able to change often. This results in a constant updates for the MPC, which is not something that should be done in the MPC approach.

If the Obstacle Avoidance and Overtaking function are realized in the follower layer then it can't be done using the LCA function to change the lane once to avoid the obstacle and once to change back to the original lane. The

LCA has to be implemented in the planner layer, as it is not just a temporary lane change like for avoiding obstacles, but a permanent change in the trajectory. This is different from the classic approach, as allowing the Overtaking and Obstacle Avoidance functions to use the LCA to implement their behaviour is an applicable solution. If the Obstacle Avoidance and Overtaking features are implemented in the planner and LCA, it is impossible to use them in conjunction. They would plan different trajectories resulting in a conflict. Generally it can be said that driving functions themselves can't be used simultaneously for planning a trajectory. This has to be done by a single component, however it is possible to represent another driving function with constraints for the MPC. This specific edge case can be seen as multiple driving functions working simultaneously. Otherwise it is benefitial to work on a strategy to find the optimal driving function for a given situation. Based on this the chosen driving function would be able to plan its trajectory. Replanning would have to be done if the developed strategy decides that the current state of the vehicle and its environment demand it.

Chapter 14

Organization

This section describes everything related to the internal organization of the project group. A more detailed description on how the product vision will be achieved is provided here.

14.1 Milestones and Timeline

In order to reach the project group's goal, the following four milestones as listed in Table 14.1 were defined in the beginning.

Table 14.1: Planned milestones

Milestone	**Start date**	**End date**
MS 1: Lane keeping assistant and fundamental architecture	05.05.2023	08.09.2023
MS 2: Adaptive cruise control and basic testbed features	09.09.2023	28.09.2023
MS 3: Autonomy Features, Robot Vision	29.09.2023	22.12.2023
MS 4: Rogue actor and platooning	23.12.2023	07.03.2023

Furthermore, a more detailed time schedule depicted in Figure 14.1 was offered initially. The thick vertical lines depict the end of a milestone. Additionally, the epic can be grouped together as follows: driving functions (green), test framework (orange), obstacle avoidance (purple), platooning (blue), documentation (yellow), and higher-level (grey).

14.2 Sprint-flow

The previously defined milestones will be archived in an agile way using the scrum process [43]. A sprint lasts three weeks and consists of the following aspects:

- **Feature-Planning (FP)**
  In this phase Product Owner (PO) and Business Engineer (BE) and all interested parties consider which features should be developed in the future

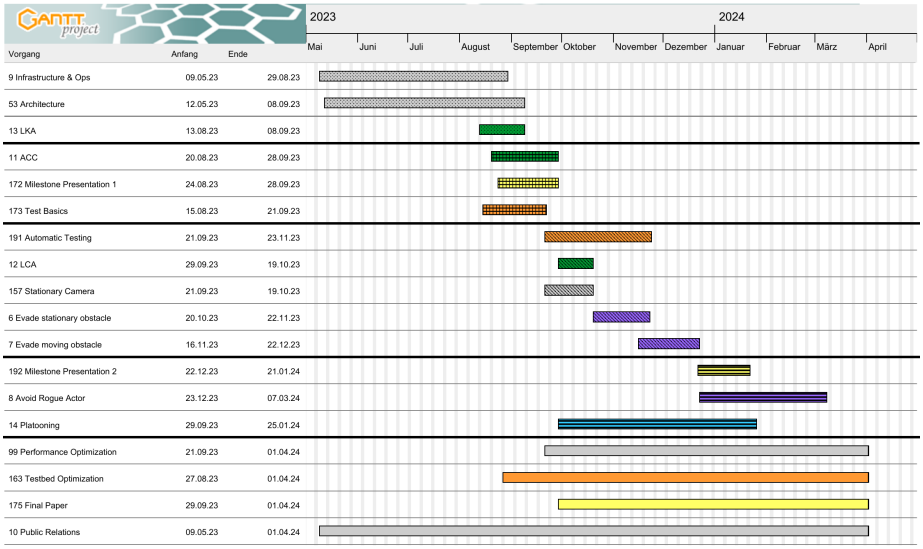


Figure 14.1: Gantt chart of epics

to reach the milestones. The features are recorded in Jira. Tickets are created that contain the needed requirements.

- **Implementation**
  During this period, the tickets are processed, documentation is written, and reviews are performed by others so that they can finally be merged.

- **Review**
  The goal of the review is to bring all stakeholders up to date. It should be mentioned which goals have been achieved and the progress should be presented.

- **Retrospective**
  The team sits down internally at the retrospective at the end of the sprint and draws a summary. The focus is on filtering out problems, exploring possible solutions and citing positive aspects.

During the sprint, a weekly serves as an exchange with the stakeholders by giving a quick presentation of last week's progress. Internally, meetings are scheduled twice a week. Once every sprint, a refinement of the backlog is planned which is done to facilitate the feature planning and refine Jira tickets to enable faster sprint plannings.

14.3 Sprint Workflow

To assure that all members follow the same workflow regarding the arising tasks during a sprint, the following well-defined workflows have been agreed upon. For example, every sprint follows a specific workflow. An overview is given in Figure 14.2, where every colored step (except the „Sprint planning“) resembles one column in a Jira Sprint Board, as it is shown in Figure 14.3. First, the sprint

has to be planned. Every ticket in the sprint is then assigned to one or more people. When they have finished processing the ticket, it goes into review and finally into acceptance by the PO or BE.

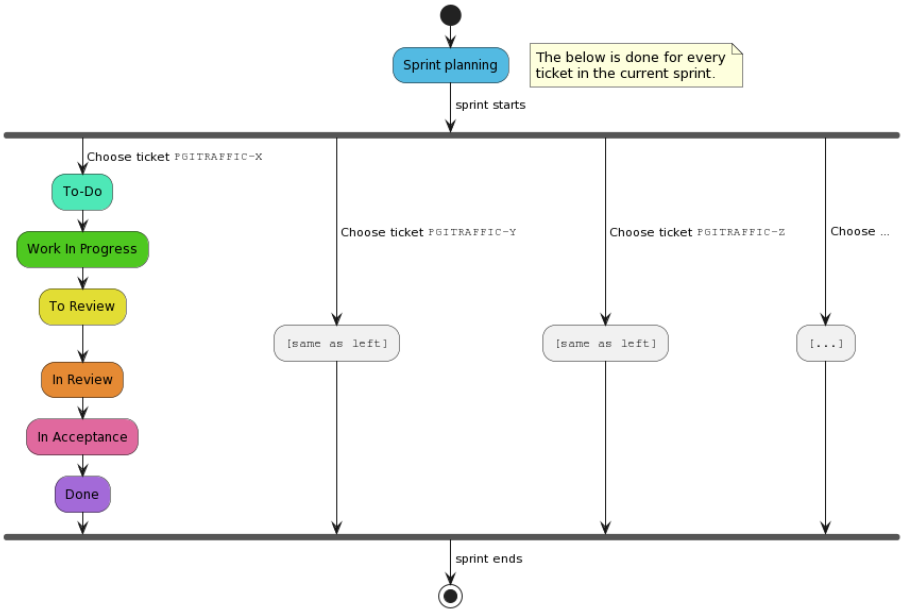


Figure 14.2: Overview of how the work on items in a sprint is done.

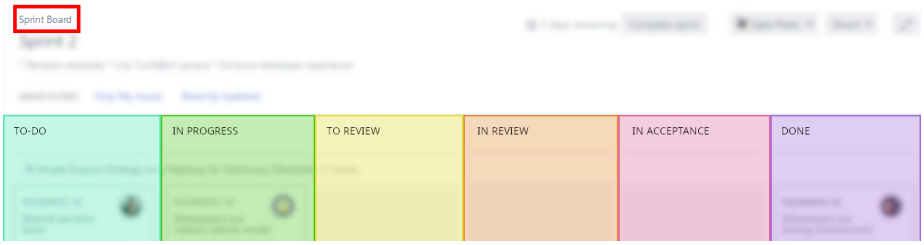


Figure 14.3: The states of an issue as represented in the Jira board.

Since some of these steps are complex in nature, it is important to clearly define their respective workflows. This is done by the following diagrams, with continued usage of the color coding as seen above. The „Sprint planning“ workflow is described in Figure 14.4. The activity „To-Do“ is empty, as this step only consists of waiting for any ticket-related work to start, thus requiring no well-defined workflow. The workflow described in Figure 14.5 shows how tickets that are in progress should be worked on. The workflow for „To Review“ and „In Review“ is shown in Figure 14.6 and the workflow for finalizing a ticket is described in Figure 14.7.

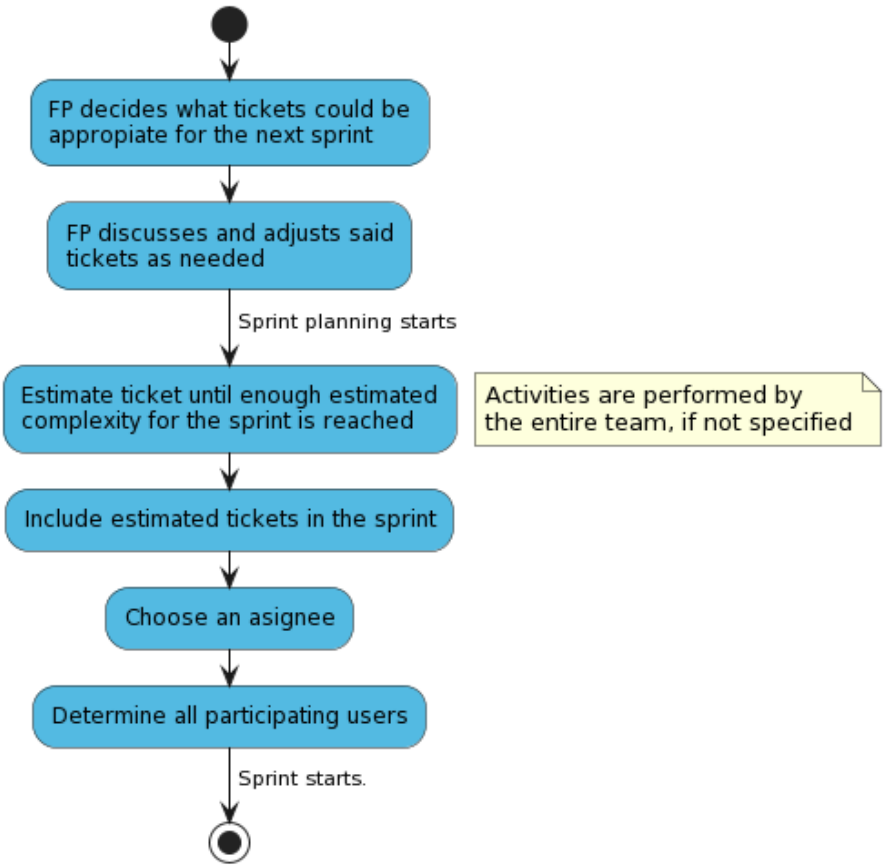


Figure 14.4: Sprint planning workflow

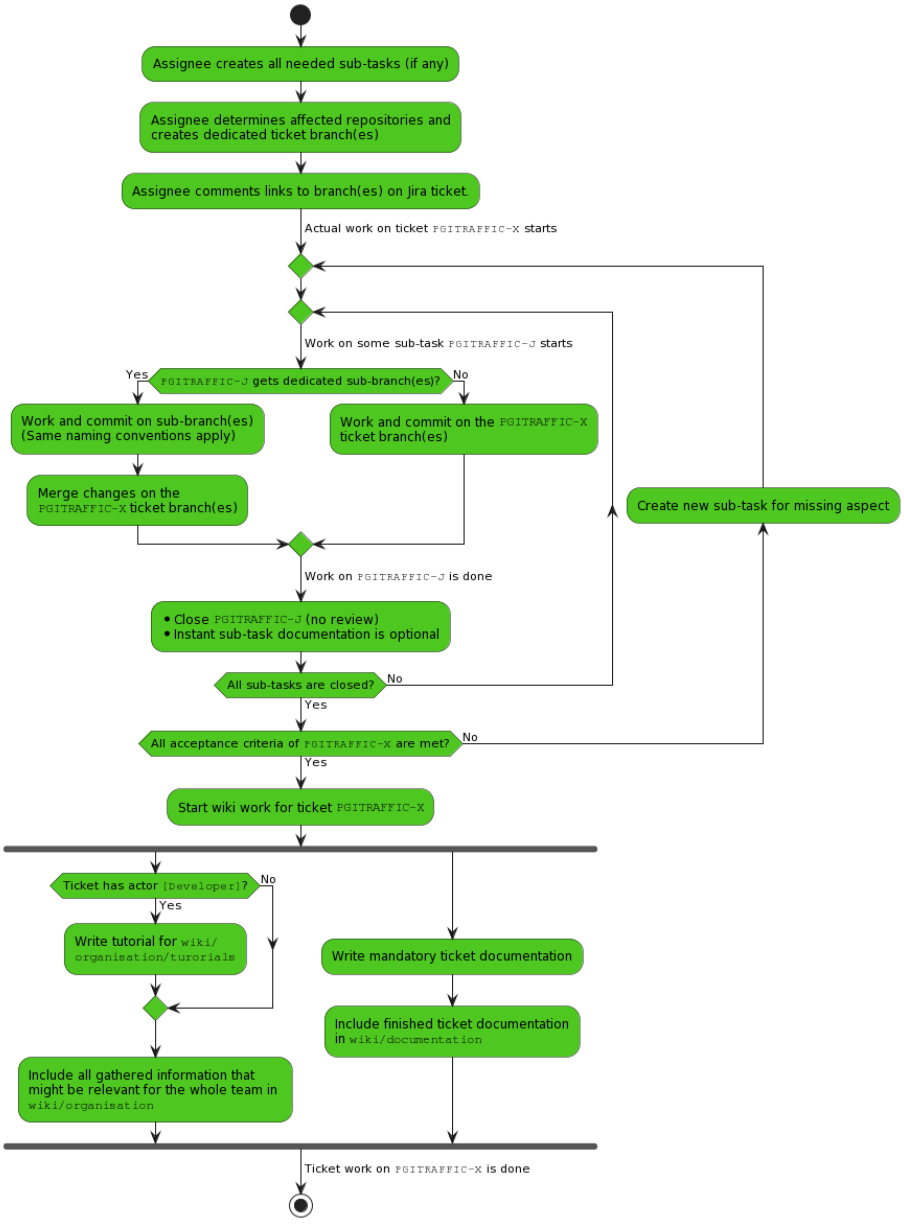


Figure 14.5: Work in Progress workflow

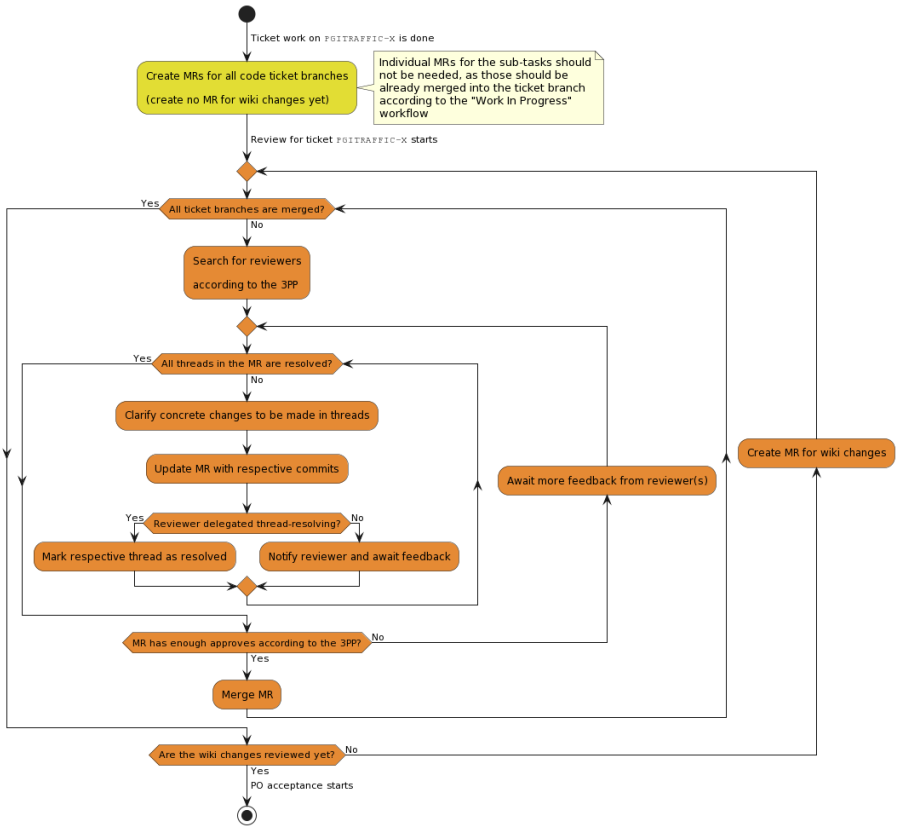


Figure 14.6: To Review and In Review workflow

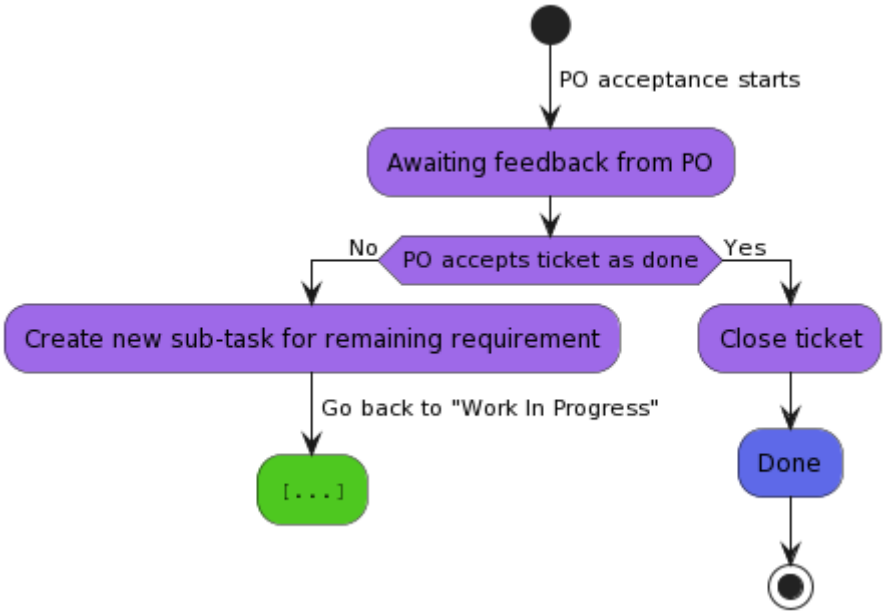


Figure 14.7: In Acceptance workflow

14.4 Defintion of Done

A ticket is considered done if the following requirements are fulfilled:

- Functionality implemented
- Reproducibly tested
- Documented
- At least three persons were involved in implementation and review, at least one of them is only a reviewer
- All acceptance criteria are met
- Accepted by PO or BE

14.5 Roles

The project group consists of eleven students. Each member is a developer, but some also fulfill different roles or focus on certain topics. These roles with the member's names inside this project are listed below.

- Scrum Master (Carl Schneiders)
  - Ensures conformity to scrum practices
  - Maintains the team's processes and takes care of removing obstacles in the process

  - Organizes retrospectives.

- Product Owner (Marie Marken)
  - Creates and maintains the product vision in consultation with the group and other interested parties.
  - Communicates with BTC ES, Foundations and Applications of Systems of Cyber-Physical-Systems and Distributed Control in Interconnected Systems
  - Maintains the backlog and organizes feature planning
  - Leads sprint planning and sprint review

- Business Engineer (Lasse Heckelmann)
  - Supports the Product Owner in her tasks
  - Keeps track of the project
  - Provides a point of contact for specialized questions

- Documentation Steward (Nellson Eilers)
  - Keeps track of the internal wiki
  - Makes sure that everyone documents their work
  - Ensures that conventions regarding the documentation are adhered to

- Infrastructure (Malte Grave)
  - Maintains the server infrastructure
  - Makes sure that everyone can work
  - Also offers technical support

- Code Steward (Jan-Magnus Monenschein)
  - Ensures that the code quality is of the desired level
  - Specifies rules and principles for working on the code base
  - Helps with all things CI/CD
  - Helps with configuring and working with development tools

- Technical Lead (Simon Struck)
  - Has an oversight over the whole system
  - Responsible for creation/management of datatransfer protocols
  - Evaluates technical feasibility

- Quality Analysis (Filip Wojciak)
  - Ensures product functionality
  - Tasked with testing of the product
  - Ensures fulfillment of requirement

- Developer (Julia Debkowski)
  - Assists with software development
  - Keeps track of the project's progress
- Software Architect (Stefan Gerber)
  - Maintains the architecture
  - Is a contact for architectural questions
- PR work (Paulina Kowalska)
  - Responsible for public work
  - Responsible for planning events

14.6 Tools

For easier collaboration, using a few tools proved to be essential. Below, some of these tools are presented.

14.6.1 Jira

Jira is a popular project management and issue tracking tool developed by Atlassian. Jira helps to manage tasks efficiently, maintain transparency, adapt to different project methodologies and collaborate effectively.
Jira allows teams to create, track, and manage issues, tasks, bugs, and user stories. This helps in maintaining a clear and organized list of work items, making it easier to prioritize and address them. Also, Jira supports agile methodologies like Scrum. It provides features such as sprint planning, backlog management, and burndown charts to facilitate agile processes.
Furthermore, Jira is highly customizable. This enables the ability to have custom workflows, issue types, and fields to tailor it to a project's specific needs.
Another important aspect is that Jira can integrate with a wide range of tools, including source code repositories (i. e. Gitlab), CI/CD pipelines and more.

14.6.2 Discord

Discord is used for communication within the team. A custom bot called Hugo is used, which partially automates processes. Particularly, he reminds the group of the internal weekly deadline, helps with the estimation process of user stories and can be used to list current merge requests and their review status.

14.6.3 Gitlab

Versioning is essential. Gitlab is used for this purpose. Repositories for the following projects exist.

- the internal wiki
- the website
- everything related to public relations

- the project report
- the server configuration
- and of course the TurtleCar implementation itself

14.6.4 Google Calendar

To keep track of important dates Google Calendar is used. Here all appointments as well as vacations are entered.

14.6.5 Etherpad

Etherpad is used to share notes and to keep the agenda for meetings.

Chapter 15

Public Relations

In this section, the presentation to the public will be addressed. This will cover tasks performed during participation in events like the FleiWa, as well as the management of the project group's online presence, including a website and Instagram account.

15.1 Quartierstag

Figure 15.1: Presentation at the Quartierstag

At the „Alte Fleiwa“ neighborhood, as part of its 100th-anniversary celebration the „Quartierstag“ was held. Here a first major milestone, the Lane Keeping

Assistant, was presented. This can be seen in Figure 15.1. During this event, local businesses, research institutions, organizations, and municipal offices provided insights into their work. More information can be found on the following link: `https://quartierstag.de/` On behalf of the University and BTC-ES, current findings were presented, a live demonstration was offered, a poster as seen in Figure 15.2 was created and the opportunity to examine hardware and software, including the Visualizer was given.

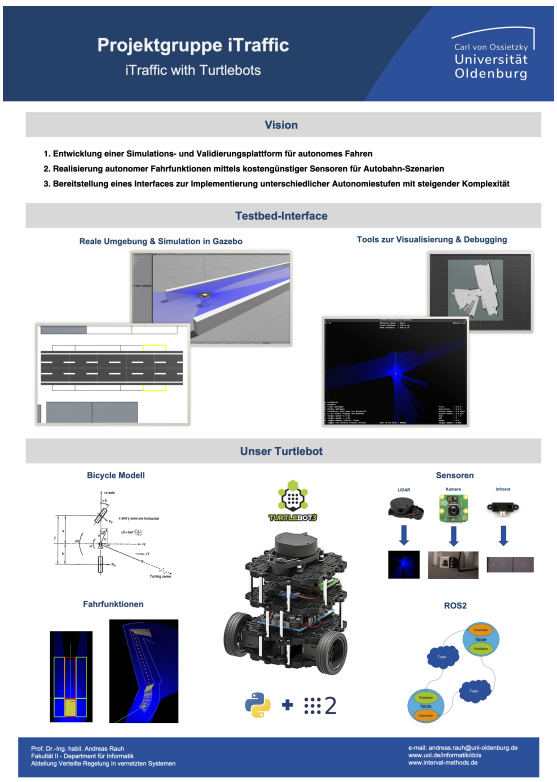


Figure 15.2: Overview of the poster for the Quartierstag.

15.2 Website

In today's world, it is of paramount importance to establish an online presence. To this end, a digital presence was created. First, an Instagram account exists, which will be actively curated in the near future. What is already accessible by the public is the website. The project's website displays the most important information for the public. Its public domain is `https://itraffic-uol.de/`.

The website is a good way to document the progress being made over time and to also show it to stakeholders. The content should primarily address the goals of the project group. Progress should be documented as well as challenges to avoid or not to repeat possible mistakes. The team represents the basic building block of the project group and is therefore presented. This way, even strangers who have nothing to do with the project group can build a good understanding of it.

15.2.1 Dependencies

The following tools are used to create the website:

- Jekyll (Static Site Generator)
- Minimal Mistakes Theme for Jekyll
- Ruby's bundler gem in order to manage the projects dependencies
- Gitlab CI/CD for building and deploying the site automatically

15.2.2 Content Review Policy

Since the content affects everyone and appears online, changes should be approved by everyone in advance. Joint reviews are mandatory.

15.3 Email

The teams public email address is: team@itraffic-uol.de

15.4 Instagram

The project group's Instagram channel can be found here: `https://www.instagram.com/pg_itraffic/`

The Instagram channel is a bit more informal and is intended to represent the project group away from the achievement of goals. For this purpose, insights into meetings but also social events can be shared. Regularity to post is secondary.

Glossary

ROS This refers to ROS2 (Robot Operating System 2). In version 2, specifications have been changed, which also include concrete implementation changes. The ROS2 version used is ROS2 Humble, which is marked as Long Term Support (LTS). 31

TurtleBot This refers to used TurtleBot's within the project group. 31

Bibliography

[1] ADAC. *Testfahrt VW Golf 8 GTI: Der Golf aus dem Fitnessstudio*. Online. Accessed on 2023-11-10. Aug. 2022. URL: https://www.adac.de/rund-ums-fahrzeug/autokatalog/marken-modelle/vw/vw-golf-8-gti/.

[2] AprilRobotics. *AprilTag: a visual fiducial system popular for robotics research*. 2023. URL: https://github.com/AprilRobotics/apriltag (visited on 12/15/2023).

[3] Author automaticaddison. *Sensor fusion using the robot localization package - Ros 2*. Dec. 2021. URL: https://automaticaddison.com/sensor-fusion-using-the-robot-localization-package-ros-2/ (visited on 10/08/2023).

[4] Uli Baumann. *Die Räder stehen fast quer*. July 2022. URL: https://www.auto-motor-und-sport.de/tech-zukunft/zf-easyturn-achse-extrem-lenkung/ (visited on 10/08/2023).

[5] More BHP. *VW MK7 Golf GT 2.0TDI 150 ECU Remap*. URL: https://www.more-bhp.com/volkswagen-golf-remapping/vw-mk7-golf-gt-20tdi-150-ecu-remap.html (visited on 10/08/2023).

[6] *Black Python Formatter GitHub Repository*. Oct. 2023. URL: https://github.com/psf/black (visited on 10/07/2023).

[7] Philipp Borchers et al. *Realtime Controlled Cooperative Autonomous Racing System next generation*. Checked 2023-10-05. Apr. 2018. URL: https://uol.de/f/2/dept/informatik/download/lehre/PGs/PG-RCCARS.pdf (visited on 10/08/2023).

[8] Paolo Bosetti, Mauro Lio, and Andrea Saroldi. "On the human control of vehicles: An experimental study of acceleration". In: *European Transport Research Review* 6 (Sept. 2013). DOI: 10.1007/s12544-013-0120-2.

[9] Nikolai Bräuer et al. *Realtime Controlled Cooperative Autonomous Racing System*. Nov. 2016. URL: https://uol.de/f/2/dept/informatik/download/studium/pg/PG_RCCARS.pdf (visited on 10/05/2023).

[10] *Bremsen*. URL: https://vorschriften.bgn-branchenwissen.de/daten/dguv/70/19.htm (visited on 10/08/2023).

[11] *Bremswege im Vergleich*. Oct. 2019. URL: https://www.adac.de/rund-ums-fahrzeug/autokatalog/autotest/bremswege-vergleich/ (visited on 10/08/2023).

[12] Bundesrepublik Deutschland. *Straßenverkehrsordnung*. 2013. URL: https://www.gesetze-im-internet.de/stvo_2013/ (visited on 10/08/2023).

[13] cfzd. *Ultra-Fast-Lane-Detection*. 2022. URL: https://github.com/cfzd/Ultra-Fast-Lane-Detection.

[14] Rüdiger Cordes. *cw-Werte*. 2022. URL: http://rc.opelgt.org/indexcw.php (visited on 10/08/2023).

[15] Werner Damm et al. *Traffic Sequence Charts - From Visualization to Semantics*. Tech. rep. Oct. 2017. URL: http://www.avacs.org/fileadmin/Publikationen/Open/avacs_technical_report_117.pdf.

[16] *DSG Shift Time*. June 2007. URL: https://www.vwvortex.com/threads/dsg-shift-time.3311040/ (visited on 10/08/2023).

[17] PG EmBrAAC. *Projektgruppe Emergency Braking Assistant for fully Autonomous Cars*. Sept. 2019.

[18] Nikolay Falaleev. *Bird's Eye View Transformation*. URL: https://nikolasent.github.io/opencv/2017/05/07/Bird's-Eye-View-Transformation.html (visited on 05/07/2017).

[19] Brian Fitzgerald and Klaas-Jan Stol. "Continuous software engineering: A roadmap and agenda". en. In: *Journal of Systems and Software* 123 (Jan. 2017), pp. 176–189. ISSN: 01641212. DOI: 10.1016/j.jss.2015.06.063. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121215001430 (visited on 10/07/2023).

[20] ROS 2 Real-Time Working Group. *Raspberry Pi image with ROS 2 and the real-time kernel*. 2023. URL: https://github.com/ros-realtime/ros-realtime-rpi4-image (visited on 10/04/2023).

[21] Gurobi Optimization LLC. *Python API Overview - Gurobi Optimization*. 2022. URL: https://www.gurobi.com/documentation/current/refman/py_python_api_overview.html (visited on 11/23/2023).

[22] ibaiGorordo. *onnx-Ultra-Fast-Lane-Detection-Inference*. 2022. URL: https://github.com/ibaiGorordo/onnx-Ultra-Fast-Lane-Detection-Inference.

[23] International Organization for Standardization. *Intelligent transport systems — Lanekeeping assistance systems (LKAS) —Performance requirements and testprocedures*. Tech. rep. 2014. URL: https://www.iso.org/obp/ui/en/#iso:std:iso:11270:ed-1:v1:en (visited on 10/08/2023).

[24] PG iTraffic. *TurtleBot 3 Image Builder*. 2023. URL: https://gitlab.itraffic-uol.de/itraffic/TurtleBot3-image-builder (visited on 10/04/2023).

[25] kemfic. *Curved Lane Detection*. URL: https://www.hackster.io/kemfic/curved-lane-detection-34f771 (visited on 05/24/2018).

[26] KIA Motors. *Kia EV6 Manual*. Oct. 2023. URL: https://www.kia.com/content/dam/kia2/in/en/content/ev6-manual/index.html (visited on 10/28/2023).

[27] Kraftfahrtbundesamt. *Monatliche Neuzulassungen September 2022*. URL: https://www.kba.de/DE/Statistik/Fahrzeuge/Neuzulassungen/MonatlicheNeuzulassungen/monatl_neuzulassungen_node.html?yearFilter=2022&monthFilter=09_september (visited on 05/11/2023).

[28] Holger Krekel. *pytest Documentation*. docs.pytest.org, Oct. 2023. URL: https://buildmedia.readthedocs.org/media/pdf/pytest/latest/pytest.pdf (visited on 10/04/2023).

[29] Janis Kröger. "Optimierung und Erweiterung einer bestehenden modellprädiktiven Regelung zur Durchführung dynamischer Überholmanöver in einem autonomen Rennbetrieb". MA thesis. Carl von Ossietzky Universität Oldenburg, 2019.

[30] *Lane departure warning system*. URL: https://en.wikipedia.org/wiki/Lane_departure_warning_system (visited on 10/05/2023).

[31] William Levison et al. *Development of a Driver Vehicle Module (DVM) for the Interactive Highway Safety Design Model (IHSDM)*. Nov. 2007. DOI: 10.13140/RG.2.2.35982.05446.

[32] Baurzhan Muftakhidinov Mark Mitchell and Tobias Winchen et al. *Engauge Digitizer Software*. URL: http://markummitchell.github.io/engauge-digitizer (visited on 10/08/2023).

[33] MathWorks. *Quadratic Programming - MATLAB quadprog*. 2023. URL: https://de.mathworks.com/help/optim/ug/quadprog.html (visited on 12/06/2023).

[34] *mockito-python GitHub Repository*. Oct. 2023. URL: https://github.com/kaste/mockito-python (visited on 10/04/2023).

[35] OpenCV. *ArUco marker detection*. 2023. URL: https://docs.opencv.org/4.x/d9/d6d/tutorial_table_of_content_aruco.html (visited on 12/15/2023).

[36] Quang-Cuong Pham. "Trajectory Planning". en. In: *Handbook of Manufacturing Engineering and Technology*. Ed. by Andrew Y. C. Nee. London: Springer London, 2015, pp. 1873–1887. ISBN: 978-1-4471-4669-8 978-1-4471-4670-4. DOI: 10.1007/978-1-4471-4670-4_92. URL: https://link.springer.com/10.1007/978-1-4471-4670-4_92 (visited on 10/04/2023).

[37] PINTO0309. *PINTO Model Zoo*. 2023. URL: https://github.com/PINTO0309/PINTO_model_zoo/blob/main/140_Ultra-Fast-Lane-Detection/download_tusimple.sh.

[38] Philip Polack et al. "The kinematic bicycle model: A consistent model for planning feasible trajectories for autonomous vehicles?" In: *2017 IEEE Intelligent Vehicles Symposium (IV)*. 2017, pp. 812–818. DOI: 10.1109/IVS.2017.7995816.

[39] Zequn Qin, Huanyu Wang, and Xi Li. *Ultra Fast Structure-aware Deep Lane Detection*. 2020.

[40] Robert Bosch GmbH. *Lane Keeping Assist*. 2023. URL: https://www.bosch-mobility.com/en/solutions/assistance-systems/lane-keeping-assist/ (visited on 10/31/2023).

[41] Robotis. *Robotis TurtleBot 3 e-Manual Chapter Two*. Online. Nov. 2023. URL: https://emanual.robotis.com/docs/en/platform/turtlebot3/features/.

[42] *Ruff Python Linter GitHub Repository*. Oct. 2023. URL: https://github.com/astral-sh/ruff (visited on 10/07/2023).

[43] Scrum.org. *What is Scrum?* 2023. URL: https://www.scrum.org/learning-series/what-is-scrum (visited on 10/08/2023).

[44] Manideep Sridhara. *TuSimple - Ace the Lane Detection Challenge*. 2021. URL: https://www.kaggle.com/datasets/manideep1108/tusimple.

[45] Anthony Stark. *Vehicle acceleration and maximum speed modeling and simulation*. 2022. URL: https://x-engineer.org/vehicle-acceleration-maximum-speed-modeling-simulation/ (visited on 11/16/2023).

[46] Forschungsgesellschaft für Straßen- und Verkehrswesen, ed. *Richtlinien für die Markierung von Straßen. Teil A: Autobahnen*. ger. Ausgabe 2019. FGSV 330A. Cologne: Forschungsgesellschaft für Straßen- und Verkehrswesen e.V, 2019. ISBN: 978-3-86446-251-1.

[47] *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*. Apr. 2021. URL: https://www.sae.org/standards/content/j3016_202104/ (visited on 10/05/2023).

[48] *Vehicle acceleration and maximum speed modeling and simulation*. URL: https://x-engineer.org/vehicle-acceleration-maximum-speed-modeling-simulation/ (visited on 10/08/2023).

[49] Lukas Wunderli. *MPC based Trajectory Tracking for 1:43 scale Race Cars*. Tech. rep. Automatic Control Laboratory (IfA), Swiss Federal Institute of Technology (ETH) Zurich, Apr. 2011.

[50] Gao Zhenhai et al. "Multi-argument Control Mode Switching Strategy for Adaptive Cruise Control System". In: *Procedia Engineering* 137 (2016). Green Intelligent Transportation System and Safety, pp. 581–589. ISSN: 1877-7058. DOI: https://doi.org/10.1016/j.proeng.2016.01.295. URL: https://www.sciencedirect.com/science/article/pii/S1877705816003222.