technische universität dortmund

Autonomous Driving Using Neural Networks

Project group
„Intelligente Autonome Taxis“

March 31, 2019

Participants:
Viktor Brack
Maximilian Diergardt
Björn Engelmann
Sebastian Gerard
Matthias Jakobs
Leonard Kleinhans
Arthur Matei
Lorenzo Perez Veenstra
Tobias Rickhoff
Christopher Riesner
Finn Thieme
Oxana Warkentin

Technical University of Dortmund
Department of Computer Science

Contents

Chapter 1

Introduction

1.1 Motivation and Background

Recently, autonomous driving systems have attracted a significant amount of interest and in the last decade, noteworthy progress has been made in autonomous driving. Development is pushed by companies like Alphabet (Google), Tesla, and Uber while being chased by conventional automotive companies. The ever-increasing computing power and the progress that has been made in image recognition and understanding enables even further growth in the area.
An autonomous driving system enables a vehicle to drive by itself without human input. It perceives and analyzes the environment and its own position, follows rules like traffic regulations, and is able to reach any specified location safely. Making human drivers obsolete will eliminate mistakes made by human drivers, yet might introduce mistakes made by autonomous driving systems. Expectations are that driving will become safer as a result. Such a system requires perception, localization, planning, and control, and must include a vast variety of sensors, actuators, and computing units.
However, making the "correct" decision based on sensor input at all times is still considered very difficult. Fully autonomous vehicles are therefore still in an experimental phase and the topic of current research.
With image processing, understanding and sensor interpretation being an important part of an autonomous driving system, companies like Alphabet that already deploy algorithms for data analysis at scale can use their advantage and experience in areas such as advanced machine learning and computer vision to develop autonomous driving systems.

In the project group "Intelligente Autonome Taxis" at TU Dortmund we aim to develop such an autonomous driving system to work not only in a simulated environment but also with a physical model car. This task connects multiple different fields such as system architecture, machine learning, computer vision, control theory and validation and also requires us to apply project management techniques. The connection of different fields

and to work on something that is part of current research motivates our group the most. But we are also excited to be able to lay the groundwork and make architectural decisions that will be used by future project groups working with the physical cars.
This report describes the development and the architecture of the autonomous driving system and summarizes the results and experience of the project group's work in the course of a year.

1.2 Project Goals

The goals of the project group can be divided into three phases:

Phase 1: Proof of concept Proving the ability of all the previously mentioned tasks on the given hardware, including the implementation of a first version of a control loop.

Phase 2: Minimal goals Using the experience from the previous phase to achieve measurable minimal goals.

Phase 3: Optional goals Fulfilling further optional goals.

During the whole project, the following definitions and limitations are considered:

Definitions and Limitations:

1. A route is a sequence of two-lane road sections with lane markers defined by navigation signs (similar to diversion signs).

2. A route can contain right- and left-hand bends / curves as well as T and X-intersections.

3. The start and end of the route are marked by road signs.

4. A route in the physical test environment is composed by 20 AADC A.2 floor tiles.

5. A route in the virtual test environment is a setup of VTD 3.3 street segments of equal length.

6. The route does not contain moving obstacles or other traffic participants.

7. An obstacle is an object with at least 35cm width and 20cm height (in the physical simulation) with a continuous and ultrasonic reflecting surface. Width is measured from the vehicles perspective.

8. A route is not known by the vehicle; there can be different routes.

An obstacle fills one lane completely and is as tall as a vehicle.

1.2.1 Proof of Concept

The actual goal of the proof of concept phase is to implement a first version of a working control loop running, however that goal is measured by the following proof of concept goals:

Goals for the Proof of Concept:

1. The autonomous car must be able to drive straight.
2. The autonomous car must be able to keep a lane on a straight road.
3. If there is a static obstacle in front of the autonomous car the autonomous car must be able to stop prior to collision.

1.2.2 Minimal Goals

1. The autonomous car must be able to follow a previously unknown route inside the virtual simulation.
2. The autonomous car must be able to follow a previously unknown route in the physical simulation.
3. The autonomous car must set light signals when turning and braking.
4. The autonomous car must prevent a collision with static obstacles by stopping.

1.2.3 Optional Goals

1. The vehicle can drive around static obstacles.
2. Moving obstacles can be recognized.
3. The vehicle can drive around moving obstacles.
4. The vehicle is driving smoothly.
5. The vehicle can recognize signs indicating heights of bridges and use this information to detect a bridge as an obstacle if it is too low to pass below.
6. The vehicle can recognize the height of bridges to detect a bridge as an obstacle if it is too low to pass below.
7. The vehicle creates a live stream giving an overview about its current status, mainly in form of telemetry data.
8. The vehicle can park in vertical and parallel parking space.
9. The vehicle can use city signs to navigate through the test environment.

10. Multiple vehicles can drive within the environment without impeding each other.
11. The vehicle respects the priority in traffic according to German law.
12. Multiple routes can be distinguished by numbers on the signs.
13. Usage of neural networks in other environments than the one it was trained in.

1.3 Contents and Structure

This report is divided into six chapters: It starts with defining the requirements for an autonomous system based on the project goals in chapter (2), followed by chapter 3 with a description of the preconditions, the physical car, and the simulation environment.
The next chapter addresses architectural aspects and decisions of the developed system, including its functional and logical views as well as safety aspects of the system. This includes an abstraction layer for the physical car and the simulation and safety mechanisms to avoid physical damage.
Chapter 5 presents the chosen approaches and their evaluation to a semantic understanding of the environment, that is the interpretation of sensor data and camera images. This includes lane detection, road detection and street sign detection.
The sixth chapter addresses the lane keeping algorithms and the control loop within the autonomous driving system and the evaluation thereof.
In the seventh chapter the implementation and documentation approach is outlined.
The conclusion of this report is the summary and the outlook.

Chapter 2

Requirements of the Autonomous Driving System

To model the requirements both for the minimal, and the optional goals, the requirements were split in safety requirements and general functional requirements. First, the general requirements were defined, later the safety requirements were used to explicitly prevent unwanted behavior.
These requirements are specified in the following, and are used in future to clarify the high-level goals which are pursued during the development of the single components of the autonomous driving system.
As the project contains several different task areas, safety requirements and requirements for the neural network are shown separately.

2.1 General Requirements

In this section the general requirements which the system should satisfy are stated. They are grouped by

- Simulation and Execution (SNX) - execution of the vehicles commands in physical and virtual simulations
- Semantic Understanding (SEM) - creation of an abstract world model
- Image Tagging (IMT) - preparation of training data for the SEM requirements
- Controlling (CON) - generate commands from the abstract world model
- Visualization (VIS) - reporting to the user
- Benchmarking Requirements (BEN) - requirements that can be used to evaluate the quality of driving

If not explicitly mentioned, the requirements relate to the physical and the virtual test environment as explained in 1.2.

ID	**Description**
GEN-SNX-1	The autonomous car WILL drive with the same behavior in physical and virtual simulations.
GEN-SNX-2	The autonomous car MUST be able to be controlled by a remote controller.
GEN-SNX-3	The remote control MUST be able to enable autonomous driving.
GEN-SNX-4	The remote control MUST be able to disable autonomous driving.
GEN-SNX-5	The remote control SHOULD be able to delegate a turning intention without street signs.
GEN-SNX-6	The remote control MUST overwrite the autonomously generated steering commands and disable the autonomous driving mode.
GEN-SNX-7	The remote control MUST overwrite the autonomously generated acceleration and deceleration commands and disable the autonomous driving mode.
GEN-SNX-8	The remote control MUST be able to trigger the emergency brake mode.
GEN-SNX-9	The remote control SHOULD be able to disable the emergency brake mode.
GEN-SNX-10	When decelerating the autonomous car MUST enable its brake lights.
GEN-SNX-11	When turning left the autonomous car MUST enable its turn signal for left turning intention.
GEN-SNX-12	When turning right the autonomous car MUST enable its turn signal for right turning intention.
GEN-SNX-13	When driving, the car MUST drive with at least 0.4 m/s.

ID	**Description**
GEN-SEM-1	The autonomous car MUST be able to recognize drivable and non-drivable parts of the floor on straight streets.
GEN-SEM-2	The autonomous car MUST be able to recognize drivable and non-drivable parts of the floor on bendings.
GEN-SEM-3	The autonomous car MUST be able to recognize drivable and non-drivable parts of the floor on intersections.
GEN-SEM-4	The autonomous car SHOULD detect obstacles as non-drivable area.
GEN-SEM-5	The autonomous car MUST be able to recognize signs indicating left turning intention.
GEN-SEM-6	The autonomous car MUST be able to recognize signs indicating right turning intention.
GEN-SEM-7	The autonomous car MUST be able to recognize signs indicating stopping intention.

ID	**Description**
GEN-IMT-1	The image tagging tool MUST be able to use images that were shot in real life.
GEN-IMT-2	The image tagging tool MUST be able to use images that were shot in VTD.
GEN-IMT-3	The image tagging tool MUST be able to export annotations as XML file.
GEN-IMT-4	The image tagging tool MUST allow team work on more than one image.

ID	**Description**
GEN-CON-1	The autonomous car MUST be able to steer autonomously.
GEN-CON-2	The autonomous car MUST be able to accelerate and decelerate autonomously.
GEN-CON-3	When driving on an X-intersection the autonomous car MUST be able to head straight.
GEN-CON-4	When driving on an X-intersection in the autonomous car MUST be able to turn left.
GEN-CON-5	When driving on an X-intersection the autonomous car MUST be able to turn right.
GEN-CON-6	When driving on an T-intersection the autonomous car MUST be able to head straight.
GEN-CON-7	When driving on an T-intersection the autonomous car MUST be able to turn left.
GEN-CON-8	When driving on an T-intersection the autonomous car MUST be able to turn right.
GEN-CON-9	After recognizing street signs indicating left turning intention within 3 seconds prior an intersection the autonomous car MUST turn left.
GEN-CON-10	After recognizing street signs indicating right turning intention within 3 seconds prior an intersection the autonomous car MUST turn right.
GEN-CON-11	After recognizing street signs indicating stopping intention the autonomous car MUST stop before passing the stop sign.
GEN-CON-12	When no street sign was detected within 3 seconds prior an intersection the autonomous car WILL head straight.
GEN-CON-13	The autonomous car SHOULD be able to avoid obstacles by driving around them.
GEN-CON-14	When driving on an intersection the autonomous car SHOULD reduce its speed.

ID	**Description**
GEN-VIS-1	The autonomous car MUST be able to display relevant information to the user by an attached monitor or using screen streaming over Wifi.
GEN-VIS-2	The autonomous car MUST display recognized drivable parts of the floor as drivable area to a user by an attached monitor or using screen streaming over Wifi.
GEN-VIS-3	The autonomous car MUST overlay drivable area over the real recognized camera image.
GEN-VIS-4	The autonomous car MUST generate a birds eye view as abstract world model containing the recognized drivable area.
GEN-VIS-5	The autonomous car MUST generate a birds eye view as abstract world model containing its controllers calculation basis.
GEN-VIS-6	The autonomous car MUST be able to show the bird’s eye view.

ID	**Description**
GEN-BEN-1	When driving on a straight street the autonomous car MUST not leave drivable areas for more than 1 second at a time.
GEN-BEN-2	When driving on a straight street the autonomous car SHOULD not leave the right lane for more than 2 seconds at a time.
GEN-BEN-3	When driving on a non-straight street the autonomous car MUST not leave drivable areas for more than 3 seconds at a time.
GEN-BEN-4	When driving on a non-straight street the autonomous car SHOULD not leave the right lane for more than 3 seconds at a time.
GEN-BEN-5	When driving on a straight street the autonomous car MUST drive at least 0,4 m/s.

2.2 Safety Requirements

Safety is one of the most important issues when developing an autonomous driving system. Therefore, requirements concerning the safety of the system are stated in this section.

ID	**Description**
SAF-1	The autonomous car MUST NOT hit static obstacles on the street.
SAF-2	The autonomous steering functions MUST be overwritable by user input.
SAF-3	The autonomous acceleration functions MUST be overwritable by user input.
SAF-4	The autonomous car MUST offer an emergency stop function to stop the vehicle by user input.
SAF-5	When in emergency brake mode the autonomous car MUST get to zero speed within under one second.
SAF-6	When closer than 100mm to an obstacle the emergency brake MUST have triggered the emergency brake mode.

Chapter 3

Simulation and Execution

The project group's experiments in autonomous driving were fundamentally split into working with two different environments. The real world, in which a physical car is supposed to drive autonomously in an environment that has to be physically adapted to individual driving scenarios, and the simulated world, in which a simulated car drives autonomously in an environment of which many more aspects (such as weather, traffic or entire cities) can be controlled. The role of the simulation is to provide a test ground for the implemented algorithms. So, it is possible to fail, and iterate quickly, without any damage that might be caused when using a real car for the tests. However, in the end, all of the algorithms must work with the physical hardware of the real car in a real-world environment with all its irregularities, intricacies and challenges.

In this chapter, the software and hardware the project group's software is built upon is examined. Also the interaction between these components and how it helps to achieve the main goals of the project group are explained. First, the development car that will eventually be used to test the algorithms in a physical test environment is introduced. Next, the framework that allows to execute code on the car is presented: ADTF. Then, the simulation aspect of the project is presented. After explaining the general makeup of a simulation scene, the core aspects of VTD, the software used to simulate all of the virtual testing environment and the virtual car, are illustrated. Followed by the explanation how the physical and the virtual cars are connected to the control code in ADTF. Afterwards, it is explained how the physical car is modeled in the simulation environment, and details about the encountered difficulties are elaborated. It concludes with the information on test routes used and how training- and test data for the lane detection task were generated from the simulation.

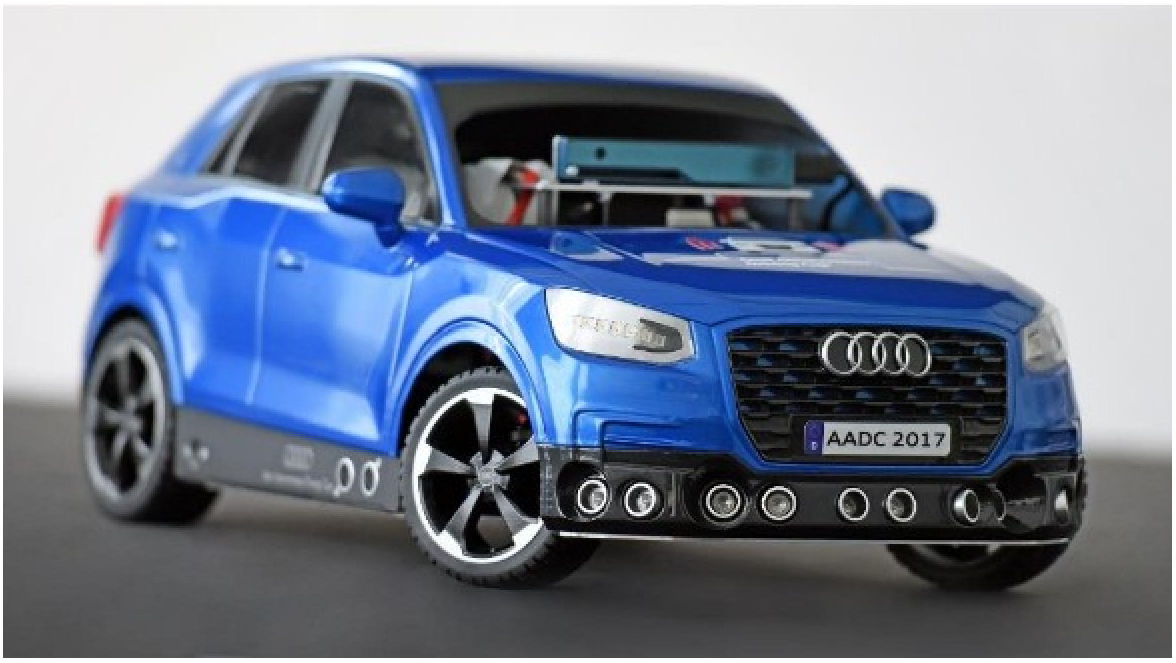


Figure 3.1: 1:8 Model car used in this group project car .

3.1 Model Car

To be able to face real world problems there is a model car. It enables to test the autonomous driving capabilities in a complex and non-ideal environment. Additionally it represents what an autonomous car is doing in this project.

The given model car is equipped with numerous sensors A.2.1, cameras and a general purpose PC 3.1.1 on board to make it suitable for autonomous operation. In addition to the usual RC car features (steering, acceleration, and braking), it has all the lights and signals that are necessary for real life traffic. It is electrically powered and can operate up to two hours without charging.

The car is equipped with different types of sensors to allow differentiated environment perception: There are 10 ultrasonic sensors to continuously measure the distance of objects they are pointed towards. Wheel encoders detect the rotation speed of the back wheels and therefore wheel speed can be calculated using the diameter of the back wheels. The car has two voltmeters for monitoring its batteries output voltage. The front and back cameras give a wide angle view of the surrounding. A detailed description of the sensors used and the ones installed but unused can be found in the appendix A. To get an overview of the available hardware infrastructure various components are introduced in this chapter. This clarifies, how the car is capable of real-time environment perception and maneuver planning. The individual hardware components are connected can be reviewed in figure 3.2.

3.1.1 PC

The core of the car's infrastructure is its general purpose (desktop-like) PC. There is a 128GB SSD, 8GB DDR4 RAM and an Intel i3 processor with 2 cores at 3.2GHz. Wifi and Bluetooth are integer part of the mainboard.
The mainboard is a GIGABYTE GA-Z170N-WIFI in the miniITX configuration (space related reasons). It provides the base for the system exactly like in any normal PC. The cameras are directly connected with USB 3.0. Further, a USB hub with its Arduinos is connected to the mainboard. Mouse, keyboard and display can be connected to work directly on the installed XUbuntu (16.04.2 LTS).
The graphics processing is achieved through the NVIDIA GeForce GTX 1050Ti (connected to the PCIe slot on the mainboard). It is equipped with 4GB GDDR5 memory and clocks in at 1290MHz. This makes it capable to host the neuronal network for segmentation and to process a picture 480 x 360 four times a second. For information about the network see chapter 5.3.
The CPU's workload is generally over-dimensioned for solely driving with ADTF, yet is nearly fully occupied with both driving and visualization of all relevant sensor data. More computing power would have increased certainty throughout the development process.
The direct handling is conducted by the five connected Arduinos (which can be seen in figure 3.2) rather than the CPU itself.

3.1.2 Arduinos

Arduino is a family of microcontroller boards. They are versatile and small, therefore often used for fast and easy prototyping. The Arduino Micro has 20 digital I/O pins, seven of which can be used for PWM[1], clock speed is at 16MHz and the flash memory is 32KB.
The car has five Arduino Micros fitted making communication with the sensors and actuators easier, faster and overall more convenient. Communication between PC and Arduinos works via USB, the Arduino boards themselves sit on Printed Circuit Boards (PCBs) (see 3.1.3), where their in- and output pins are connected through the circuit boards with the sensors, lights and actuators. All of them are programmed for their specific task by the manufacturer and ready for us to use with ADTF.
There is one Arduino handling the five ultrasonic (US) sensors in the front bumper of the car and another one manages the rest in the rear bumper and the sides. They trigger the sensor's measurements and calculate the distance, so that the Arduino Communication Filter (see 3.2.2) can return this value into ADTF.
One Arduino is used as a handler for the position and motion tracking sensor. Again the desired values can be received directly from the Arduino Communication Filter.

[1]Pulse-width modulation - varying a boolean signal at a high frequency to get virtually a intermediate value

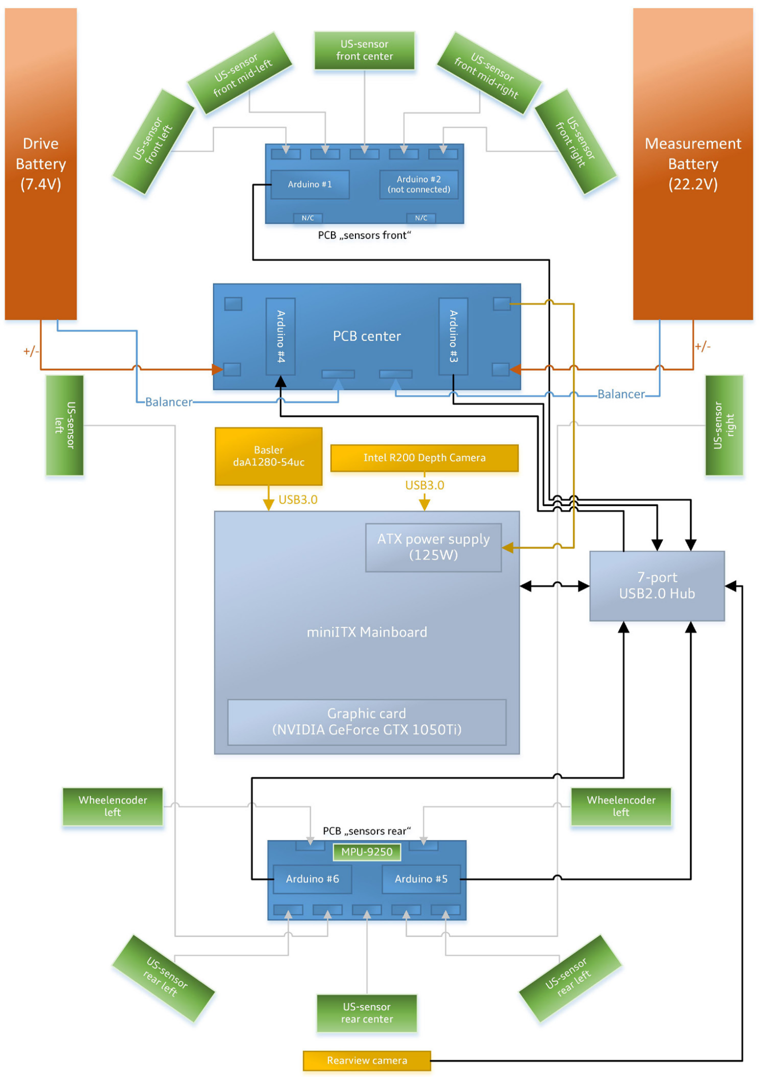


Figure 3.2: Overview of hardware architecture of the model car. The main elements are the mainboard with the connected Arduinos and sensors. As they need a bigger bandwidth, the cameras are directly connected to the PC. The printed circuit boards (PCB) provide an organized connection between the Arduinos and the respective hardware. bff

The actuators and lights are controlled with yet another Arduino, this makes accessing speeds, steering angles and lights very easy to the programmer. Desired values are sent directly to the Arduino Communication Filter. The final Arduino is used to control the battery (power balancing, monitoring the voltage).

To provide a fixed connection between Arduinos and sensors, car control etc. there are three printed circuit boards.

3.1.3 Printed Circuit Boards

The front and back PCBs are simple connection circuits between Arduinos and sensors. They provide a socket for the Arduino Micros and enable the ultrasonic sensors to be

plugged in. The position and motion tracking sensor is placed directly on the rear PCB and connected via conductive paths.
The center PCB is far more complex: It houses the units for checking the batteries and the one for actuator-/ light control, fuses for ATX power supply and motor current, here the user also switches between autonomous- and RC mode. At this board the lights, RC receiver, speed controller, batteries, the respective balancers and the external power supply are connected. The potentiometers on it are to fine tune the initial position of the steering servo and the idle torque of the motor.
The chassis is connected at the center PCB, to operate the lights.

3.1.4 Lights

The car's body contains indicators, headlights, rear lights, brake lights and reverse lights. They are controlled by a small circuit board embedded in the bodywork. This board is to be connected to the center PCB via RJ45.
Indicator- and brake lights are operated according to the car's maneuvers, as specified in GEN-SNX-11.

3.1.5 Handling Noisy Ultrasonic Signals

Test measurements revealed that the ultrasonic signals of the front sensors contained a lot of noise. The setup consisted of a cardboard box being positioned at different distances in front of the car. While the front center sensor reported a signal of expected shape, the more angled sensors recorded signals that were very noisy.
Analyzing the signal graphs, three types of noise were identified. The first type of noise consists of singular blips that strongly differ from the trend of the surrounding signal points (see Figure 3.3). These were easily smoothed by implementing a moving average approach, replacing the current signal value by the unweighted mean of the last k signal values. Testing approved $k = 5$ to be able to correct such jumps. Higher values resulted in smoothing out the signal too much, thereby risking to recognize trend changes too late. Lower values sometimes where not able to average out the noise enough. For simplicity the emergency brake application was connected and its stopping capability observed. This is adequate, as the ultrasonic values are mainly used for this task.
The second type of noise consists of maximum/minimum values. The sensors report a value of -1 in case of errors, and 400 to report that there is no object within reach. While this is generally a desirable behavior, it causes problems when the correct signal is surrounded by these values (see Figure 3.4). Especially when using the moving average approach outlined above, this noise would strongly distort the true signal values. Therefore the moving average approach is modified in the following way: When computing the moving average over the last k frames, only those values that lie within the interval of $[0, 399]$ are used. If no valid signals are left in that window, -1 is returned as the correct error value.

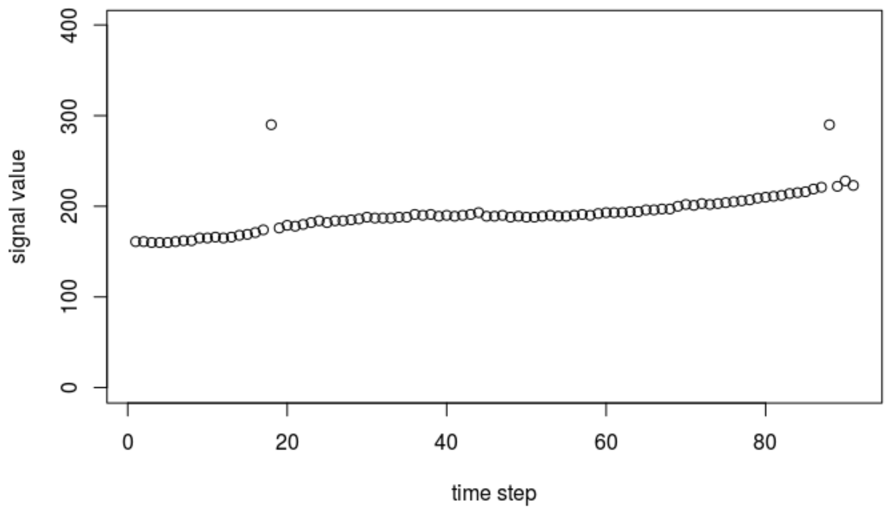


Figure 3.3: Type 1 noise in the ultrasonic sensors: Singular values that differ from the longtime trend. An object was placed at 1600mm and moved to 2200mm during the test.

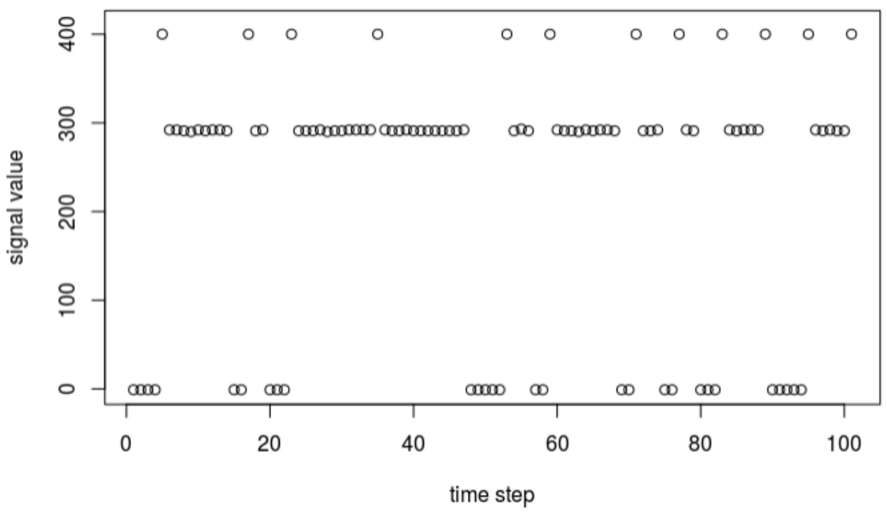


Figure 3.4: Type 2 noise in the ultrasonic sensors: Extreme values indicating error states. -1 indicates an error, 400 indicates that no obstacle was detected. An object was stationary at 3000mm away from the sensor during the test.

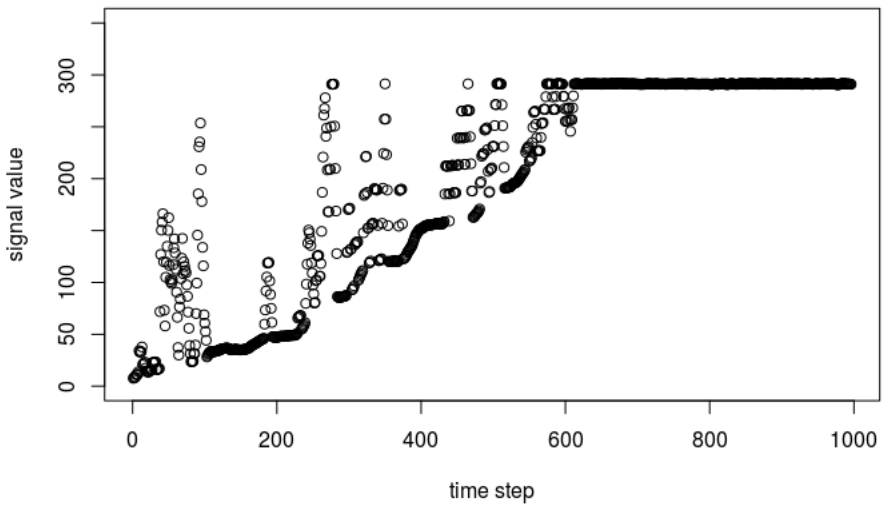


Figure 3.5: Type 3 noise in the ultrasonic sensors: Several contiguous values that differ from the longtime trend. An object was placed at 20mm in front of the sensor and pulled back to be placed at 3000mm.

The third type of noise in the ultrasonic sensors is a prolonged version of the first type. While it is easy to smooth out singular wrong values in the signal, this does not work anymore when it randomly fluctuates over a prolonged period of time (see Figure 3.5). Smoothing these random signals just results in other random values. Detecting this kind of noise is not trivial, since it requires separating which trend in the data represents the real world, and which one is fluctuation. This is especially hard, since the noise does not seem to follow any specific patterns, unlike in the first two cases.
As the outliers where exclusively higher than the correct values, the approach was to widen the window of the mentioned moving average. The resulting value is now equal or higher than the real distance. So if for example the measured distance is 500mm, the using routine (for example the emergency brake) can expect the real distance to be *much* less and therefore react (*much*) earlier. (This is precisely, what the safety factor is doing in the emergency brake, see 4.2.2.) To find the appropriate window size and what "early" reaction means, again the practical stopping capabilities were consulted.

To be able to control aspects like triggering the sensors via the Arduinos, a framework is used. ADTF is a platform for (particularly, but not exclusively) controlling the car's actuators and sensors. It provides diagnostic tools and serves as a testing suite.

3.2 ADTF

Lately, the number of autonomous cars and cars with an advanced driver-assistance system (ADAS) have significantly increased. To be able to assist the driver or to control the car as a whole, the system needs to have access to actuators and the drive train. To make

decisions about manipulating the car's state by executing driving maneuvers, systems need authoritative data about the current environment. Therefore these cars are equipped with numerous sensors. It is vital to receive their information within a reasonable time.
To solve this task not only quickly, but also reliably and therefore safely, the industry uses excessively tested frameworks. One of them is the Automotive Data and Time Triggered Framework (ADTF). This is also the framework used in this project to deal with the car's (see section 3.1) sensors and actuators. Made by a large car manufacturer for internal purposes it was later made accessible to other industry competitors.The main strategy behind this decision was to make the general development of autonomous driving systems faster and to facilitate interchanging and already implemented components. Having a standardized structure for components means the safety of components can be verified more efficiently and more independent because of easy exchangeability. Today ADTF is licensed by the company Elektrobit and is available for Linux and Windows. It is used by big automotive companies, both car manufacturers and component suppliers.

3.2.1 Basics

Functionality in ADTF is achieved by connecting so-called filters: A filter is a component that can examine and/or manipulate data. It might receive input data or might send output data (both optional). The transfer of data is achieved via the filters' pins (dedicated input- or output pins). They send simple or complex data types like numbers or pictures. For a detailed explanation of the inner functionality of a filter see section 3.2.2.
The central element of ADTF is the ADTF configuration. It describes which filters will be executed and how the data flow between them is configured. The configuration can be represented in XML to make it easily interchangeable among developers. It can also be saved in binary format to protect knowledge of implementation details and prevent customers from making changes.
To manipulate the configuration ADTF comes with a *what-you-see-is-what-you-get-* interface: the programmer can drag-and-drop filters into the configuration, change their properties, create connections between pins and has a visual feedback of what the data and control flow looks like. This component of ADTF is called the ADTF Development Environment and is used for setting up the configuration. The other major component of ADTF is ADTF Runtime, which enables the developer to execute the configuration in question. Here, the developer can observe camera streams, sensor values, and filter activity. ADTF Runtime can also be started via console. This is helpful when the configuration is finished and can operate autonomously without a GUI.

3.2.2 Filters

Filters are data processing elements, which need at least one input or output pin. Otherwise, they could not be part of the filtergraph and therefore of the configuration. Their

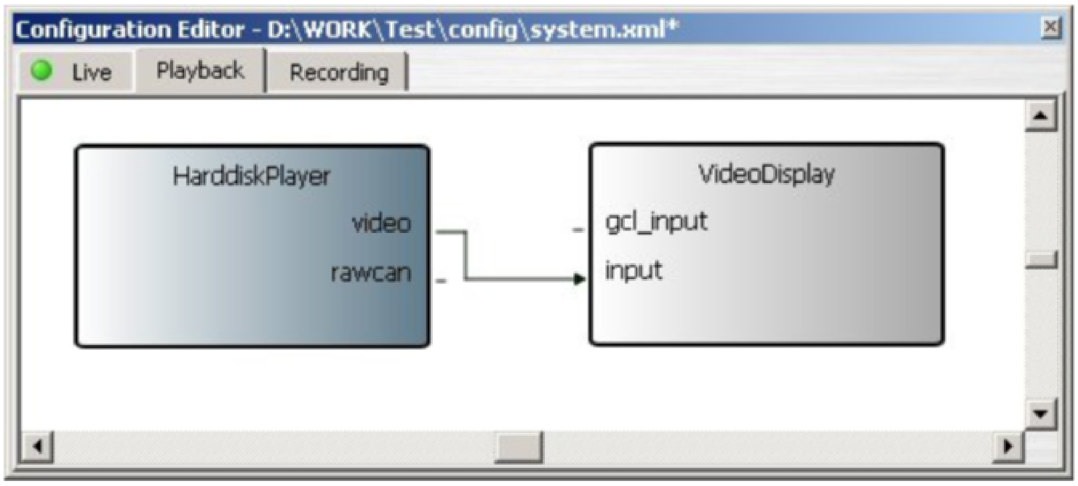
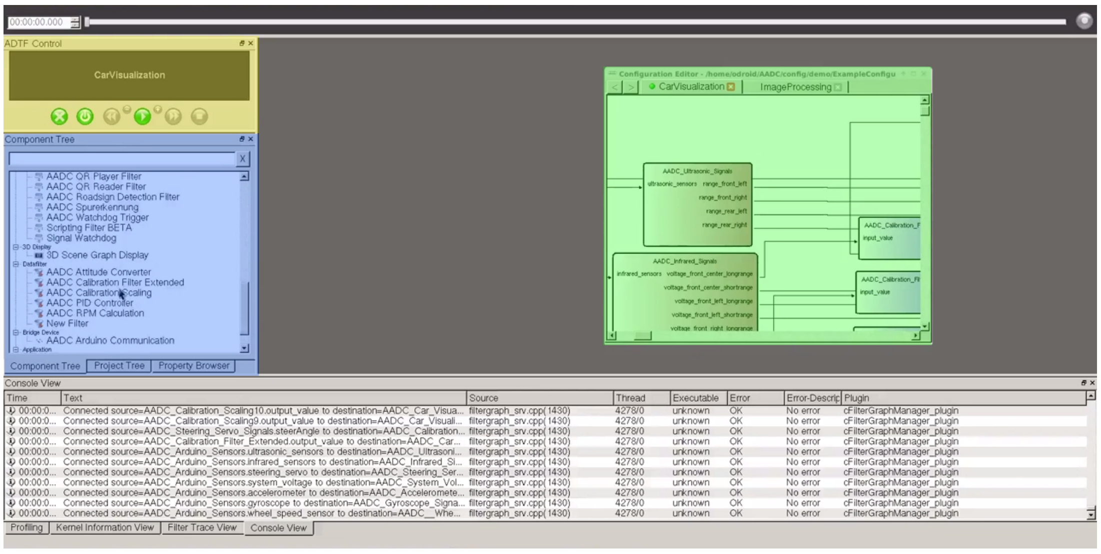


Figure 3.6: Filtergraph with two filters. The video output pin of the HarddiskPlayer is connected to the input pin of the Video display.



```
Time         Text                                                                              Source                  Thread  Executable  Error  Error-Descrip  Plugin
00:00:0...   Connected source=AADC_Calibration_Scaling10.output_value to destination=AADC_Car_Visua...   filtergraph_srv.cpp(1430)   4278/0  unknown  OK  No error  cFilterGraphManager_plugin
00:00:0...   Connected source=AADC_Calibration_Scaling9.output_value to destination=AADC_Car_Visuali...   filtergraph_srv.cpp(1430)   4278/0  unknown  OK  No error  cFilterGraphManager_plugin
00:00:0...   Connected source=AADC_Steering_Servo_Signals.steerAngle to destination=AADC_Calibration...   filtergraph_srv.cpp(1430)   4278/0  unknown  OK  No error  cFilterGraphManager_plugin
00:00:0...   Connected source=AADC_Calibration_Filter_Extended.output_value to destination=AADC_Car...    filtergraph_srv.cpp(1430)   4278/0  unknown  OK  No error  cFilterGraphManager_plugin
00:00:0...   Connected source=AADC_Arduino_Sensors.ultrasonic_sensors to destination=AADC_Ultrasoni...    filtergraph_srv.cpp(1430)   4278/0  unknown  OK  No error  cFilterGraphManager_plugin
00:00:0...   Connected source=AADC_Arduino_Sensors.infrared_sensors to destination=AADC_Infrared_Si...    filtergraph_srv.cpp(1430)   4278/0  unknown  OK  No error  cFilterGraphManager_plugin
00:00:0...   Connected source=AADC_Arduino_Sensors.steering_servo to destination=AADC_Steering_Ser...     filtergraph_srv.cpp(1430)   4278/0  unknown  OK  No error  cFilterGraphManager_plugin
00:00:0...   Connected source=AADC_Arduino_Sensors.system_voltage to destination=AADC_System_Vol...       filtergraph_srv.cpp(1430)   4278/0  unknown  OK  No error  cFilterGraphManager_plugin
00:00:0...   Connected source=AADC_Arduino_Sensors.accelerometer to destination=AADC_Acceleromete...      filtergraph_srv.cpp(1430)   4278/0  unknown  OK  No error  cFilterGraphManager_plugin
00:00:0...   Connected source=AADC_Arduino_Sensors.gyroscope to destination=AADC_Gyroscope_Signa...       filtergraph_srv.cpp(1430)   4278/0  unknown  OK  No error  cFilterGraphManager_plugin
00:00:0...   Connected source=AADC_Arduino_Sensors.wheel_speed_sensor to destination=AADC__Whe...         filtergraph_srv.cpp(1430)   4278/0  unknown  OK  No error  cFilterGraphManager_plugin
```

Figure 3.7: Example of the ADTF gui. In green: The filtergraph editor; in blue: Component tree (all available components); in yellow: Control panel. Source: https://www.youtube.com/watch?v=6qALb70FQ10

tasks range from communication with devices like cameras and actuators, writing to/reading from hard disk, and exchanging data over the network for handling and converting application related data (steering information, sensor data, etc.).
Communication between filters is dealt with by so-called *pins*. Pins are constructs to transmit data from a sending filter, creating a **pin-event** on the side of the receiving filter. This has the advantage of being able to react only to specific events on a pin. Events can differ between the pin the event came from and the transmitted data the receiving filter can take the programmed action.
While the implementation of a filter cannot be changed in ADTF, the filter can be customized by the user via *properties*. Should the application need multiple filters of the same kind (i.e. TCP network filters), the user can set different properties for different filters (i.e. different ports or network addresses). The user uses the ADTF GUI to set the properties directly for the displayed filters. The current values of the properties are also saved in the configuration file (XML or binary).

3.2.3 Services

ADTF has two categories of services: system services like the console output or the control of the Runtime, and GUI services like the configuration editor and the component tree. GUI services enhance the capabilities of the Development Environment with new functionality. Such a service could be an oscilloscope, a different view of the filtergraph or any other kind of development aid.

3.2.4 Implementation of Filters

A filter itself is a C++ class with a corresponding header. By extending the *cFilter*-class, each filter can be initialized, updated and equipped with pins in the (ADTF-) framework. ADTF uses CMake to manage dependencies, so if necessary, there should be a CMakeList file in addition to the header and class.
The header includes the specification of the filter's name, category, and declarations of methods and attributes, including any pins. The name of the filter does not have to be the class name. It should be descriptive and recognizable to the developer working on the filtergraph. The filter's category is there to make the list of components more clear and make it a component *tree*. Hereby the ADTF user not only finds the filter more easily but also has a better idea of what the filter is supposed to do. Examples of filter categories are DataFilters, LoggingFilters, CameraAdapters and so on.
Every filter goes through a life cycle, which is broken up into *stages*. By accessing the eStage object in the filter its current state can be checked. The life cycle consists of three stages: StageFirst, StageNormal, and StageGraphReady. With the methods *init* and *shutdown* the filter transitions into and out of a stage. As the name suggests the first stage is StageFirst. Here, static pins are initialized and assigned their data types. In

StageNormal the properties are checked and embedded into the filter. Also, dynamic pins are registered if any are being used. StageGraphReady indicates to the programmer that the filter is ready to use, the pins are initialized and will transmit or receive data.
When data is received, the method *onPinEvent()* is automatically called. Here the developer is able to react to specific data and data sources (i.e. different pins), by calling specific methods or running adequate code.
As for data types, ADTF uses a System, where common data types are wrapped in *Mediatypes*, which always contain some load data and a time stamp. This makes it easy for each *Mediasample* to get its initial date of transmission in addition to the usual data. ADTF allows the developer to use its standard types for video streams, numeric data, etc., and also to create own Mediatypes as required. If a data type is needed that contains just the front center ultrasonic sensor value and the current wheel speed, it can be created and used in filters.

3.3 Constructing a Simulation Scene

To let a car drive autonomously in a virtually simulated environment, the simulated scene can be divided into the following three different layers: First it is necessary to describe the surface of the road that the car is going to drive on. Having defined a single road, multiple roads need to be connected to a network of roads to drive on, possibly including obstacles and other static objects. Lastly, dynamic behavior happening on the road can be added, mostly to test how other road users behave and how they interact with and react to each other.
Since this is a rather universal approach to constructing simulation scenes in the automotive field, VIRES Simulationstechnologie GmbH has started three corresponding projects to define open data types for the scene layers. They would facilitate interoperability and make it possible to easily switch between different simulation environments, without transforming the descriptions and data each time. However, at the point of writing this, the format describing the dynamic behavior (OpenScenario) is still in development.
In the following subsections each of the three projects will be introduced.

3.3.1 OpenCRG - The Road Surface

OpenCRGope [a] is a project that aims to define a set of open file types for the description of road surfaces. Additionally, they provide tools to interact with the respective data and files in MATLAB and ANSI-C. The format originated in the CRG file format used by Daimler, which directly implies the practical relevance of it.
CRG is an acronym for ’curved regular grid’, which is the method with which data about the road surface is recorded. A curved grid is superimposed on the represented road and for each crossing point on the grid the values of interest can be recorded. The recorded data typically pertains to the elevation and friction coefficients of the road. However, the

```xml
<OpenDRIVE>
  <header revMajor="1" revMinor="1" name="" version="1.00" date="Thu Dec 10 10:35:57 2009"
north="0.0000000000000000e+00" south="0.0000000000000000e+00" east="0.0000000000000000e+00"
west="0.0000000000000000e+00" maxRoad="517" maxJunc="2" maxPrg="0">
  </header>
  <road name="" length="1.6517824248160636e+01" id="500" junction="2">
    <link>
      <predecessor elementType="road" elementId="502" contactPoint="start"/>
      <successor elementType="road" elementId="514" contactPoint="start"/>
    </link>
    <type s="0.0000000000000000e+00" type="town"/>

    <planView>
      <geometry s="0.0000000000000000e+00" x="-7.0710678117841717e+00"
       y="7.0710678119660715e+00" hdg="5.4977871437752235e+00" length="4.8660000002386400e-01">
        <line/>
      </geometry>
      <geometry s="4.8660000002386400e-01" x="-6.7269896520425938e+00"
       y="6.7269896522231525e+00" hdg="5.4977871437736381e+00" length="3.1746031746031744e+00">
        <spiral curvStart="-0.0000000000000000e+00" curvEnd="-1.2698412698412698e-01"/>
      </geometry>
```

Figure 3.8: Excerpt from OpenDrive file. Source: ope [b]

definition of the file type allows any scalar data to be recorded at each reference point on the grid. Several different file types exist, both binary and in human-readable ASCII format, although all of them contain a human-readable header portion, describing the data contained.

3.3.2 OpenDrive - Networks of Roads

Having described the road surface of a single road, the combination of several roads to a network is necessary. The OpenDrive project defines a file format to do just that. According to the project page "it is considered a de-facto standard in the simulation industry" (ope [b]), being used by Audi, Daimler and BMW, among others.
OpenDrive is a hierarchical XML-format with a wide variety of features. While the properties of the road surface can simply be described by including OpenCRG files, the XML-format can be used to describe all the other static elements of the simulation scene. This includes the logical features of the network, like which streets and lanes are connected, and by which type of intersection. It also allows for semantical features, for example whether the simulated road is representing a road within a town or on a highway. Furthermore street signs can be added, as well as signals and interdependencies between them. Lastly, it is also possible to add static objects, like houses next to the road, obstacles on the street or trees on the green belt between two lanes.
Figure 3.8 shows an excerpt of an OpenDrive file to illustrate the file format.

3.3.3 OpenScenario - Dynamic Behavior

After defining the static components of a simulation scene with the OpenCRG and OpenDrive file, any dynamic behaviors that should occur in the scene can be defined. This

```
storyboard
    story: owner =Ego
        act 1: condition =simTime >2s
            sequence 1.1: actor =$owner
                maneuver 1.1.1: name =start driving
                    event 1.1.1.1: condition =upon start of act
                        action 1.1.1.1.1: apply throttle at 0.4
            sequence 1.2: actor =$owner
                maneuver 1.2.1: name =apply steering torque
                    event 1.2.1.1: condition =10s after start of act
                        action 1.2.1.1.1: set throttle to 0.0
                        action 1.2.1.1.2: apply 2.0Nm torque to steering wheel
```

Figure 3.9: Structure of an OpenScenario file. Source: ope [c]

mainly includes describing the behavior of different drivers. For example one driver might keep driving a steady pace, while another accelerates until they come too close to another car, and then starts an overtake maneuver. While such definitions seem simple at first, the complexity increases significantly if the behavior of several drivers is combined. With a growing number of such drivers in a simulation it becomes harder more difficult to predict the exact behavior emerging from the interactions of those rules.

OpenScenario aims to capture these different behaviors in XML-formatted data types. However, the format is still in the early stages of its development. The current state of the project is that a formal definition of the structure of the XML files exists in the form of XML schema files. The stated goal for the end of 2017 includes a style guide and a tool to validate OpenScenario documents. However, this goal seems to have not been met yet.

Figure 3.9 outlines the exemplary structure of an OpenScenario file. After two seconds, the driver accelerates for ten seconds. Then they stop accelerating and rotate the steering wheel by applying a specified torque to it.

3.3.4 Assembling the Scene

The interaction of the different file formats to create the simulation scene is shown in figure 3.10. The OpenScenario file describes the whole scene from a logical perspective. Additional files referenced in this file are then used to infer the physical features of the simulated objects. From this, the data received from the sensors (e.g. simple cameras or ultrasound sensors) can be simulated, as well as the visualization for the users of the simulation software. Using this information, the systems that are supposed to be tested via the simulation (e.g. autonomous driving systems) can make decisions which are then evaluated.

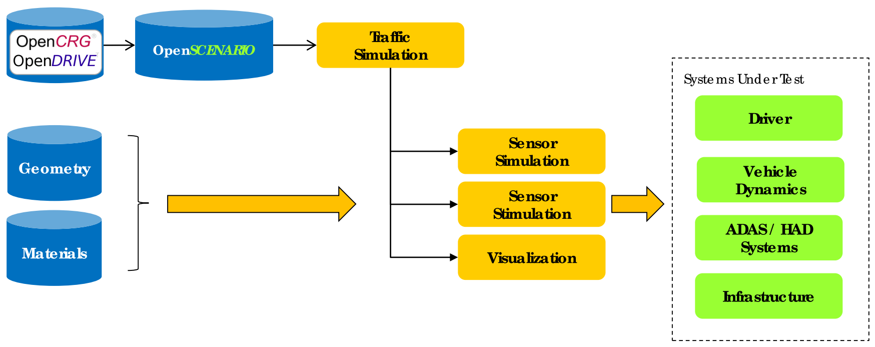


Figure 3.10: Integration of the different file formats in the creation of the simulation scene. Source: vir

3.4 Virtual Test Drive (VTD)

Virtual Test Drive (VTD) is a simulation software used in the automotive industry, developed by VIRES Simulationstechnologie GmbH (from hereon referred to only as VIRES). It "is a complete tool-chain for driving simulation applications." (vir) The software offers a wide range of modular components for different tasks involved in automotive simulations. While it can also be used for aerospace and railroad simulations. This will not be discussed any further, since the main focus of the project is on autonomous driving cars.

Since VIRES is also the driving force behind the previously mentioned open file formats OpenCRG, OpenDrive and OpenScenario, the software modules included in VTD align well with the separation that the file formats follow. It is assumed that roads in the form of OpenCRG files are present, while the data represented in OpenDrive and OpenScenario files can be actively manipulated in VTD. In the following sections the different modules included in VTD will be introduced.

3.4.1 Road Designer (ROD)

The Road Designer (ROD) enables the definition of single streets, in the OpenCRG format, as well as assembling whole networks of roads. The system works similar to an assembly kit. It is possible to choose different building blocks of OpenCRG-roads from the library and combine them into one road, saved in an OpenDrive file. Figure 3.12 illustrates this principle, while figure 3.11 shows what the software interface for the road designer looks like.

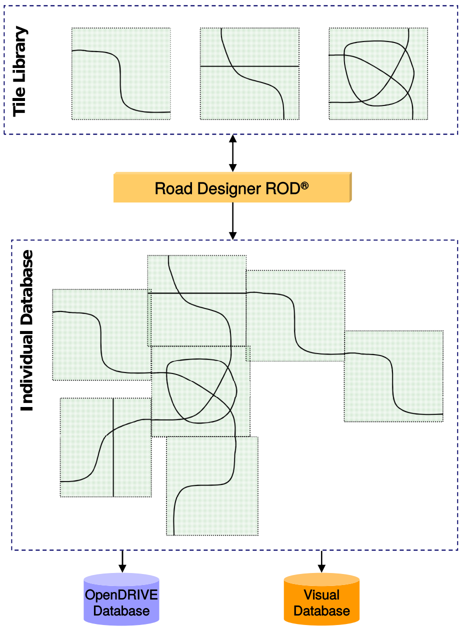


Figure 3.11: Road Designer working principle. Several roads represented as OpenCRG files are combined into a single network of roads and saved in an OpenDrive file. Source: vir

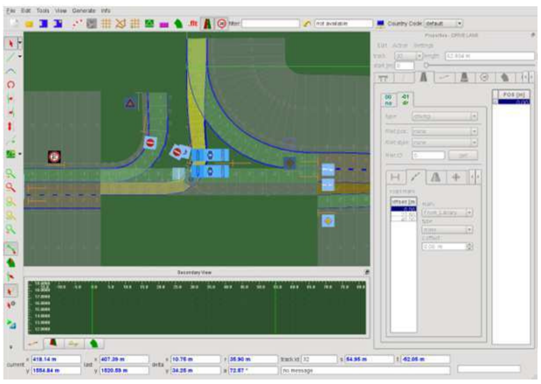
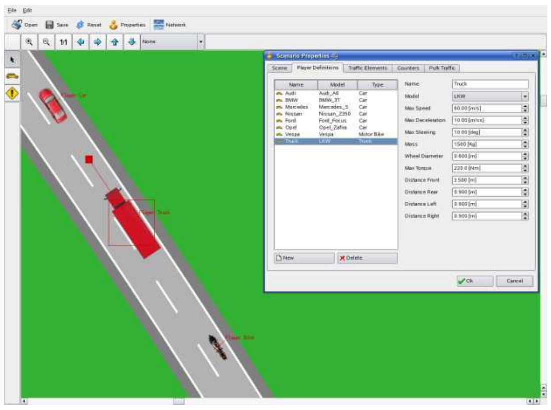


Figure 3.12: Road Designer interface. Source: vir

3.4.2 Scenario Editor: v-Scenario & v-Traffic

The scenario editor v-Scenario is the core component of the whole VTD toolkit. As the name suggests, it is used to implement the data represented by the OpenScenario file format. It is both an offline editor to manipulate the scenario, as well as a monitoring tool during the running simulation. v-Traffic is the name of the engine module that actually runs the simulation in the background. The user interface of the v-Scenario editor can be seen in figure 3.13.



Name	Model	Type
Audi	Audi_A6	Car
BMW	BMW_3T	Car
Mercedes	Mercedes_S	Car
Nissan	Nissan_Z350	Car
Ford	Ford_Focus	Car
Opel	Opel_Zafira	Car
Vespa	Vespa	Motor Bike
Truck	LKW	Truck

Name	Truck
Model	LKW
Max Speed	60.00 [m/s]
Max Deceleration	10.00 [m/ss]
Max Steering	10.00 [deg]
Mass	1500 [Kg]
Wheel Diameter	0.600 [m]
Max Torque	220.0 [Nm]
Distance Front	3.500 [m]
Distance Rear	0.900 [m]
Distance Left	0.800 [m]
Distance Right	0.900 [m]

Figure 3.13: User interface of the v-Scenario editor, which allows the manipulation of the dynamic contents of a scene. Source: vir

Dynamical Scene Elements

As indicated in the introduction of the file formats, v-Scenario works based on a OpenDrive data file that defines all the static components of the simulation scene, and adds dynamical elements on top of it. Typical elements that can be added are the following:

- vehicles
- pedestrians
- control programs for traffic signals
- triggers and resulting actions

It is also possible to add static objects like obstacles or way-side objects. While this does break the strict separation indicated in the data type section, it makes sense to include this in the scenario editor for a better user experience.

Controlling Vehicles

Vehicles can be controlled in the following ways:

- autonomously
- controlled by actions
- controlled externally during the simulation
- combinations of the above

The software offers several different types of autonomous drivers already. Additionally, they can be set to simply follow a lane or a given path. Pedestrians are simulated by an external 3rd party software. They can either follow given paths or perform different tasks like switching between running and walking or perform gestures.

Vehicle Actions

The most important actions that can be performed by vehicles are the following:

- speed change
- lane change
- switching to autonomous driving

Additionally, it is possible to perform general simulation actions, like stopping the simulation or changing the weather.

Trigger Conditions

Actions are started when certain trigger conditions are met. The following conditions can be used to define triggers:

- absolute position (on the whole map, or on the road)
- relative position (spacial distance or time until collision)
- time based (e.g. 10 seconds after the start of the simulation)
- external trigger (e.g. by user during the live simulation)

As mentioned when discussing OpenScenario, these trigger-action-driven behaviors seem simple, but the complexity emerges when several vehicles with their own behaviors interact.

3.4.3 Image Generator: v-IG

v-IG is the image generator module. It is responsible for generating both the video stream for the visualization of the scene to the user, as well as all the sensors that the simulated car might have, for example simple optical cameras, or even infrared sensors.
Notable weather features that the image generator is able to handle include the following:

- precipitation by particles (rain, snow)
- impaired visibility by fog
- continuous time-of-day
- real-time shadows
- sun glare
- reflective road surfaces, based on humidity and clouds
- reflective traffic signs

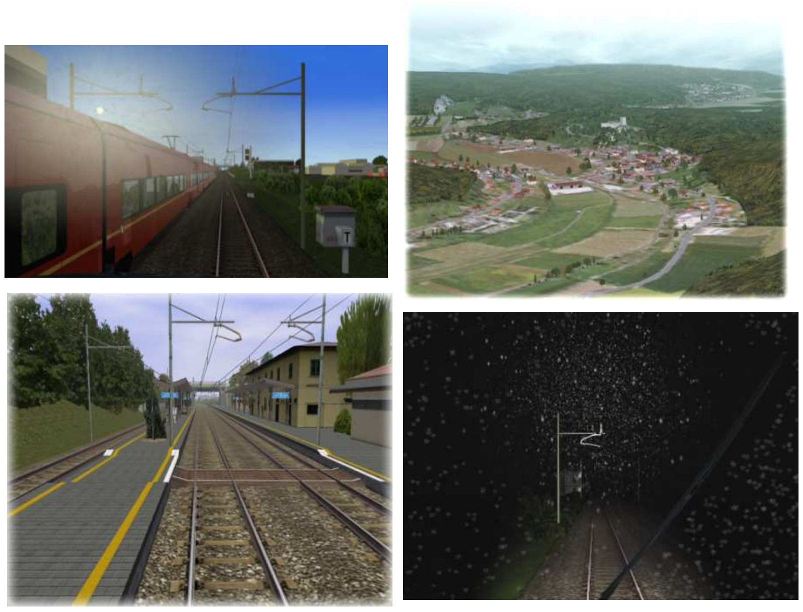


Figure 3.14: Sample images from VTD simulations, illustrating different capabilities of the v-IG module. Source: vir

- additional light sources (street lamps etc.)

A few sample images illustrating the different aspects of v-IG's capabilities are shown in Figure 3.14.

3.4.4 External Communication: RDB

VTD is able to send simulation data to (and receive instructions from) ADTF during the simulation. For this a TCP connection between the two can be opened and can then receive simulated camera images, positions, speed, acceleration of vehicles, or sensor data in ADTF. Since the image generator and each ultrasonic sensor uses their own ports several TCP connections are opened to receive all of the relevant data.
The format of the data being transmitted is VTD's own RDB (Runtime Data Bus) format. Here the focus is on the important aspects of the format, the full detail can be reviewed in the corresponding documentation and definition files. Each message contains a header `RDB_MSG_HDR_t`, indicating the size of the message etc. Following this is a list of variable size, consisting of entry headers `RDB_MSG_ENTRY_HDR_t`. Each of those introduces a corresponding list of same-type entries that hold the actual data. The type of those entries is indicated by their package id. For example, to retrieve images contained in RDB entries, one can filter for `pkgId == RDB_PKG_ID_IMAGE`. The image is then included in the message as binary data in an RDB format, the name of which is also indicated in the `RDB_MSG_ENTRY_HDR_t` by an enum type. Unfortunately this format does not necessarily correspond directly to ADTF data types. Additionally, it is necessary to manually com-

pute the address and length of the payload data, like the image, based on the data given in the header.

```
RDB_MSG_HDR_t
        RDB_MSG_ENTRY_HDR_t (pkgId = TypeA)
                entryOfTypeA
                entryOfTypeA
                entryOfTypeA
        RDB_MSG_ENTRY_HDR_t (pkgId = TypeB)
                entryOfTypeB
        RDB_MSG_ENTRY_HDR_t (pkgId = TypeC)
                entryOfTypeC
                entryOfTypeC
```

Figure 3.15: Example structure of an RDB message. Source: VTD documentation

3.5 Providing a Unified Interface to the Simulation and Car Data

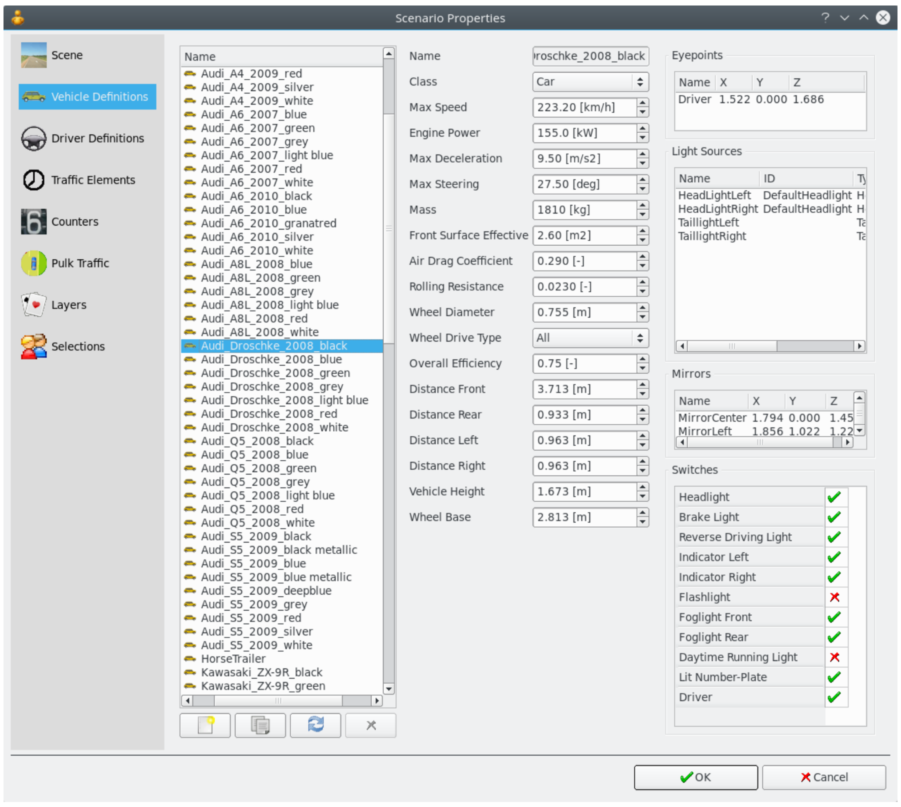
During the project two distinct environments are used: The simulation in VTD with a simulated car, and the real world with a physical car. It is likely that there will be differences in terms of parameters, e.g. in the controllers and the interpretation of images. However, it is preferable to provide a unified interface to the components processing the data. This way only specific parameters have to be adjusted for the two environments, instead of changing whole parts of the pipeline.
Since the goal is to achieve the best possible performance in the physical world, the simulation data is preferred to be transformed to fit the format of the car data. The data received from the car is not altered and the group's models and computations operate on the information-rich raw data instead. The group assumes that this will improve the performance, since the training and testing data are as close as possible to the setting experienced in the target environment.

3.5.1 Recreating the Car in VTD

In order to prepare the transition from working on the simulation in VTD to the model car from AADC it is necessary to recreate the car described in chapter 3.1 in VTD as accurately as possible. In order to do that a custom car in the scenario editor of VTD can be created, which is assigned the appropriate vehicle dynamics, camera types and camera positions of the model car, described in the following subsections.

3.5.2 Vehicle Dynamics

In VTD a car is defined by a number of parameters shown in figure 3.16 as well as boolean values for lights and 3D points for mirrors and eyepoints. During the recreation of the used model car there were no accurate descriptions of the actual AADC car. Since it



Name
Audi_A4_2009_red
Audi_A4_2009_silver
Audi_A4_2009_white
Audi_A6_2007_blue
Audi_A6_2007_green
Audi_A6_2007_grey
Audi_A6_2007_light blue
Audi_A6_2007_red
Audi_A6_2007_white
Audi_A6_2010_black
Audi_A6_2010_blue
Audi_A6_2010_granatred
Audi_A6_2010_silver
Audi_A6_2010_white
Audi_A8L_2008_blue
Audi_A8L_2008_green
Audi_A8L_2008_grey
Audi_A8L_2008_light blue
Audi_A8L_2008_red
Audi_A8L_2008_white
Audi_Droschke_2008_black
Audi_Droschke_2008_blue
Audi_Droschke_2008_green
Audi_Droschke_2008_grey
Audi_Droschke_2008_light blue
Audi_Droschke_2008_red
Audi_Droschke_2008_white
Audi_Q5_2008_black
Audi_Q5_2008_blue
Audi_Q5_2008_green
Audi_Q5_2008_grey
Audi_Q5_2008_light blue
Audi_Q5_2008_red
Audi_Q5_2008_white
Audi_S5_2009_black
Audi_S5_2009_black metallic
Audi_S5_2009_blue
Audi_S5_2009_blue metallic
Audi_S5_2009_deepblue
Audi_S5_2009_grey
Audi_S5_2009_red
Audi_S5_2009_silver
Audi_S5_2009_white
HorseTrailer
Kawasaki_ZX-9R_black
Kawasaki_ZX-9R_green

Name	roschke_2008_black
Class	Car
Max Speed	223.20 [km/h]
Engine Power	155.0 [kW]
Max Deceleration	9.50 [m/s2]
Max Steering	27.50 [deg]
Mass	1810 [kg]
Front Surface Effective	2.60 [m2]
Air Drag Coefficient	0.290 [-]
Rolling Resistance	0.0230 [-]
Wheel Diameter	0.755 [m]
Wheel Drive Type	All
Overall Efficiency	0.75 [-]
Distance Front	3.713 [m]
Distance Rear	0.933 [m]
Distance Left	0.963 [m]
Distance Right	0.963 [m]
Vehicle Height	1.673 [m]
Wheel Base	2.813 [m]

Eyepoints

Name	X	Y	Z
Driver	1.522	0.000	1.686

Light Sources

Name	ID	Ty
HeadLightLeft	DefaultHeadlight	H
HeadLightRight	DefaultHeadlight	H
TaillightLeft		Ta
TaillightRight		Ta

Mirrors

Name	X	Y	Z
MirrorCenter	1.794	0.000	1.45
MirrorLeft	1.856	1.022	1.22

Switches

Headlight	✓
Brake Light	✓
Reverse Driving Light	✓
Indicator Left	✓
Indicator Right	✓
Flashlight	✗
Foglight Front	✓
Foglight Rear	✓
Daytime Running Light	✗
Lit Number-Plate	✓
Driver	✓

Figure 3.16: Example of the vehicle definitions of a car in the scenario editor of VTD including the vehicle dynamics and eye points, which is used as main camera position. Other optional properties include light sources, mirror positions and various boolean indicators (not used by the VTD car replica)

is supposed to be a model car for the Audi Q2, the vehicle dynamics of that car were researched and entered into VTD.
For the eyepoints the model car was measured to figure out where the main camera is located in relation to the reference point, which in VTD lays centered in the back axle. In the real world the eyepoint is 29.5 cm to the front and 21.5 cm higher compared to the reference point. To transform it to real world specifications the distances were scaled, assuming an accurate 1:8 scale of the AADC car. Those coordinates are used in VTD as the driver spot, which will later be used as the main eyepoint.
The other properties namely light sources, mirrors and switches are irrelevant for the driving properties and were largely ignored. The values for those properties seen in figure 3.16 are copied from the VTD preexisting Audi Q5 2008 representation, which was used to create an initial design. The VTD user interface of the scenario editor does not support all the options mentioned above, so some of the options like the 3D points had to be manually set in the according xml files.

3.5.3 Cameras and Sensors

VTD is able to create depth images of the same point of view as the RGB image, which is also (approximately) an ability of the model car's cameras. The depth camera and Basler camera are almost in the same position, so setting up an additional eyepoint for the depth camera was not necessary, since using the main eyepoint already provides acceptable results.
For now only the five front ultrasonic sensors were set up in VTD. Following the same procedure as setting up the camera position, the distance of the ultrasonic sensors from the vehicle's reference point as well as their angles were measured, scaled to real world dimensions and entered into the simulation environment. VTD has its own section for sensors, where the perfect single ray sensors can be used to emulate the ultrasonic sensors. Those sensors return a value describing the distance of an object to the car measured with a single ray and can return it within an RDB package on a separate port for each sensor. Of course the accuracy of the real ultrasonic sensors are nowhere near the accuracy of those perfect sensors. Therefore working with the real data is harder than with the simulation data. Only the center ultrasonic-sensor was used for tests of the emergency brake.

3.6 Complications and Differences

The replication of the AADC car in VTD did not work exactly as desired. Due to shortcomings of documentation of the simulation software it took a while to properly set up the depth camera of the vehicle. A live stream of the RGB and depth image at the same time would be necessary for real time calculations, which is also the default option of VTD video streaming. However, streaming both image types slows down the simulation time significantly so that real time tests are unusable slow. After contacting the VTD

development team they provided help setting up the streaming first to shared memory on the VTD machine and then streaming that data to the image handling machine, which required a few specific project settings and external tools. This method lead to an image delay of about 10 seconds on the receiving machine, which makes this method unfeasible. Thus settling with the still suboptimal former option to slow down the simulation speed turned out to be the better choice. Later the streaming of the depth image was disabled entirely, since it would only be used for obstacle avoidance as an optional goal in the future.
In addition to the camera issues, the simulated car might have considerable differences to the real car as the actual vehicle dynamics of the AADC car were unavailable. VTD does not have any way of setting the engine type to an electric one, so in order to simulate the driving properties of the real model car the maximum possible acceleration and deceleration should be used, following a tip by the developers on the Vires support board. Thus for accurate control of the VTD car a different set of parameters than the ones used for the AADC car have to be found and used.

3.7 Test Routes

Before using the implemented system in the physical world, it needs to be ensured that the complete architecture is able to execute all the subtasks accurately and coherently in the simulation. For this, tracks in VTD to drive on are necessary. While it is possible to use premade tracks, those generally seem rather complex for simple test purposes. Therefore custom routes have been created.

3.7.1 A First Test Route in VTD

In the beginning simple routes were needed to concentrate on the task of lane keeping. After several iterations it was decided to perform the initial tests for lane detection and steering control on a road that starts with a long straight route and then bends into a long right curve (see Figure 3.17), so the group can test both the lane keeping on a straight road as well as slowly turning the wheels in a curve to stay on track (requirement GEN-BEN-X chapter 2.1).
This track already allowed testing the following important situations:

- Straight road
- Straight road with curve ahead
- Curved road

While these are not all situations (specifically crossings are missing here) it provided a very simple testing environment. Observing the first tries of following the road and adjusting parameters was used, to make sure that the basic elements of the system work.

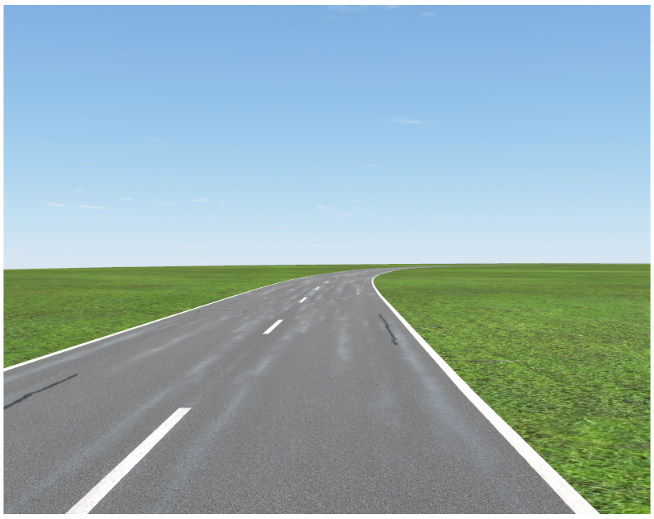


Figure 3.17: First simple track to test lane detection and steering control for lane keeping on.

3.7.2 Advanced Test Routes

To test more complex driving situations fitting test routes were necessary. Since the first test route covered only detection and driving on a simple and barely curved road, other test tracks needed to be created which were supposed to cover:

- Sharp curves
- Junctions
- Distractions next/on the road

For this reason, several more test routes were created in VTD. The first ones were simple circuit courses with various big and small curves, but without junctions. The advanced lane keeping algorithm, which used the output of the neural network (see Chapter 5.3), could be evaluated and tested on these tracks. The two lanes of the road were enclosed by grass and a few trees, thus the lane detection was kept quite simple, since it only needs to filter out the greenery to find the road.
Other tracks later created varied from a big rural track to a small city circuit. These tracks contained T- and X-junctions, as well as street signs on the sides of the road, to prepare for the sign detection. Two of those tracks can be seen in Figure 3.18.
The town road was enclosed by various buildings and other city structures like phone booths and bus stops. The road also occasionally was covered with pot holes, manhole covers and crosswalks. Another created test route was a scaled replica of the physical test route built from the tiles provided by AADC (see also Chapter 3.7.4). Training on this route with black roads on the same black background turned out to be difficult, so it was used less for tests in VTD. The problem later turned out to be less challenging in the real world scenario with properly trained networks. An overview of all created test routes in the simulation can be seen in Chapter A.3.

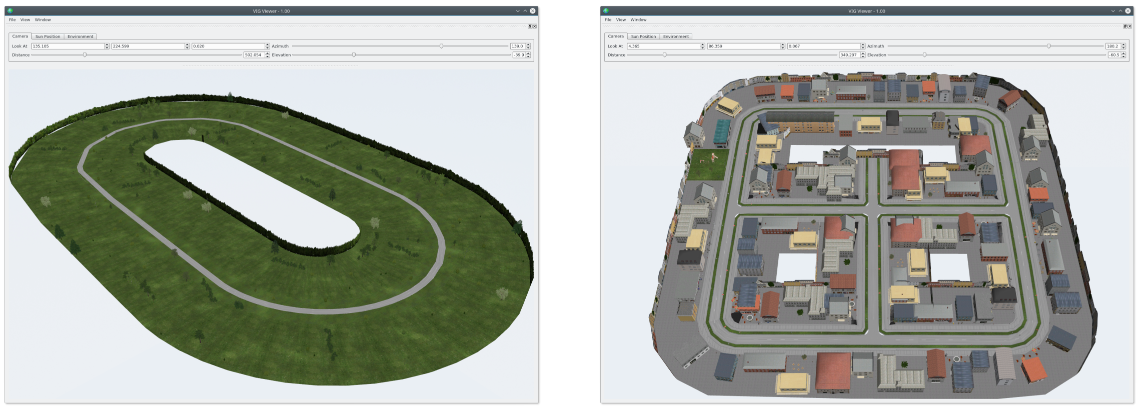


Figure 3.18: Overview over two advanced test routes as seen from the road designer. On the left the circuit and on the right the simple town track

3.7.3 Generating Test Images for Segmentation

To be able to assess the quality of the lane detection algorithm in an offline setting, the project group generated images from the VTD simulation.
VTD claims to be able to generate videos from the simulation. However, during testing it became clear that it required a myriad of different video and image conversion libraries to be installed, based on the errors printed by the video conversion script. After installing about seven of those, partially outdated, this attempt was aborted.
Instead, the single images were used, which the program generates when setting the 'save to file' property of the project's video parameters to true. This produced one 24bit 3-channel RGB bitmap image per simulated frame.
After generating some test data with the simple track mentioned above, the more complex routes were used to generate training data for the neural-network-based lane detection approach (see Chapter 5.3). Some sample images generated this way can be seen in Figure 3.19.
Since the network should be able to generalize well, more noise (details and object irrelevant to the street detection) and different situations were included in this set of images. The more complex track used for this consisted of a route starting outside of town, going through a tunnel, and running along streets with varying exteriors in a small town.
The noise in this scenario was introduced via the following scene variations:

- varying street marks (outside/inside town, in the tunnel)
- varying outer borders of the roads
- varying traffic and street signs
- guide rails outside of town, but not inside
- varying environment beside the road:

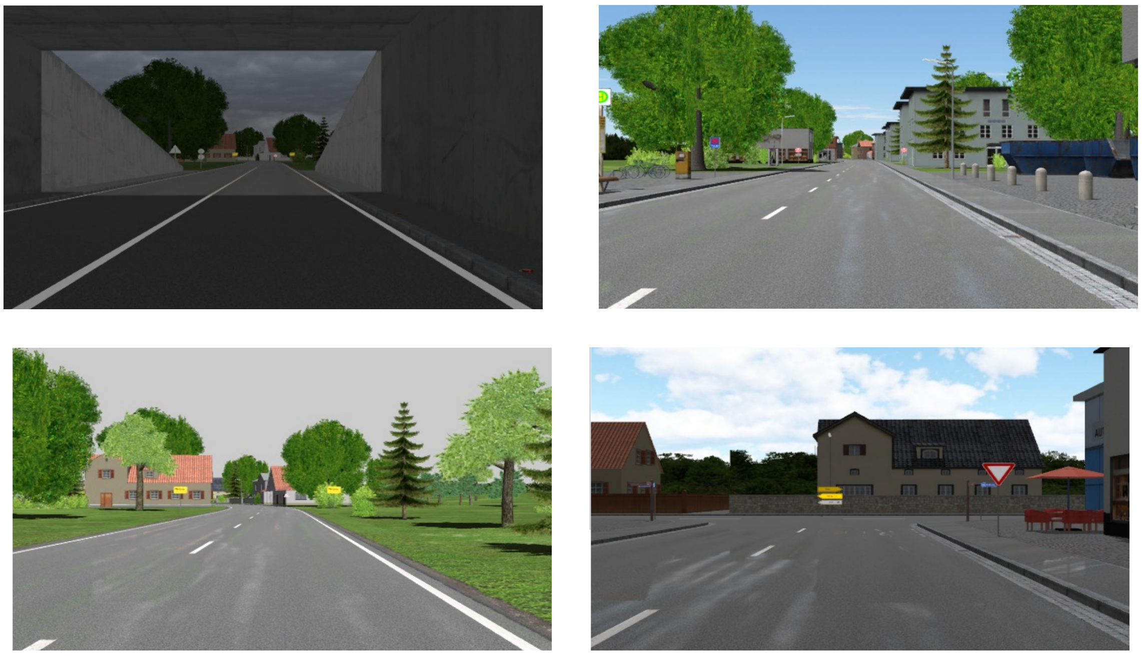


Figure 3.19: Sample images of the complex test route used as training data for the neural-network-based lane detection

  - different houses
  - bikes
  - bus stop
  - zebra crossing
  - lamp posts
  - trees

Furthermore the drive along the track was recorded five times, simulating different weather conditions:

- no sky (replaced with a completely gray texture)
- blue sky
- cloudy
- overcast
- rainy

These images were then manually annotated and used to train a neural network for lane detection (see Chapter 5.3), but later also automatically annotated using advanced features of VTD.

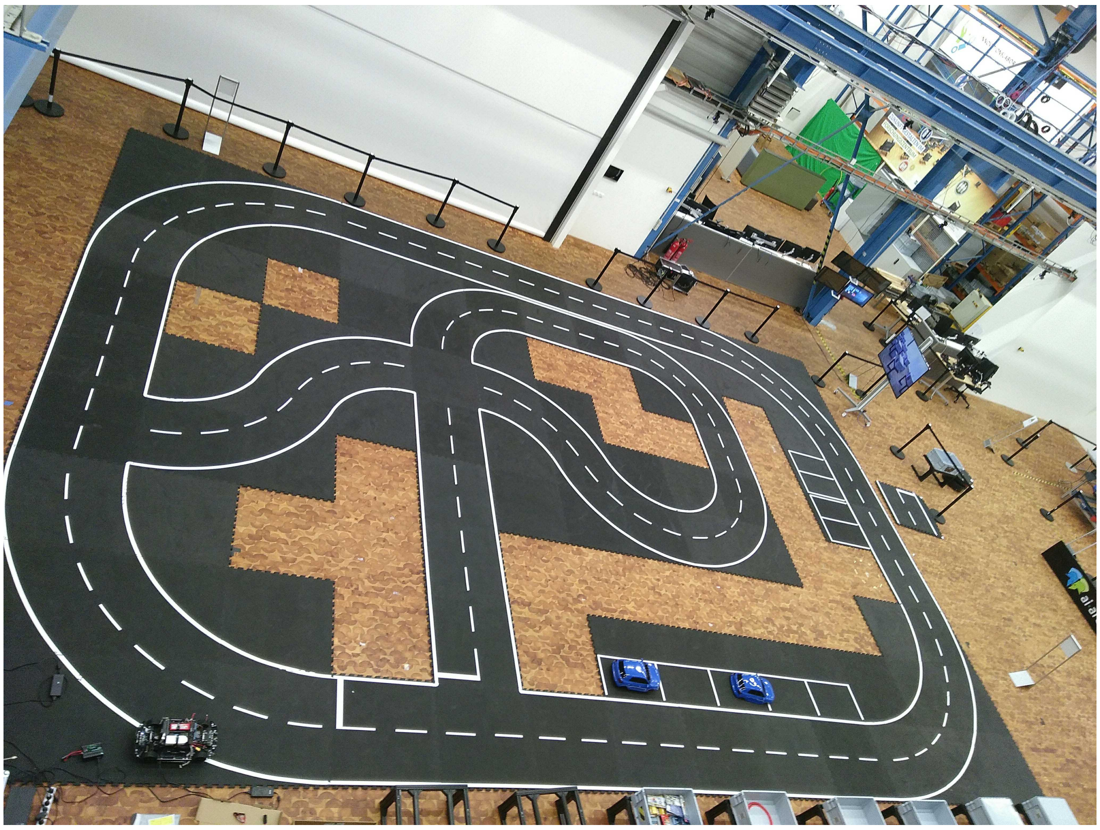


Figure 3.20: Overview of the real test route build with floor mats

3.7.4 Physical Test Routes

For testing the car's performance in a real physical environment there are floor mats with lines printed on them. These mats are modular to allow great variety of road situations and to lay out a course to the tester's preference. Such a course can include sharp and shallow curves, T-junctions, an X-junction, two S-curves and parking spots (both parallel and orthogonal to the street).

To test the lane keeping on the real car, it was first driven on straights only. Later test tracks also included curves to build a simple circuit, to see if the car could stay on track after a few rounds on the circuit. For complex testing a large test course was built with all different parts of the floor mats, to cover all possible scenarios of a road as seen in Figure 3.20. This route included following straight and non-straight roads, X- and T-intersections and some parking spots next to the road. To include the sign recognition hand-crafted sign posts were added to the physical test course at various places covering signs indicating a left- and right turn, as well as a stop sign.

For traffic and static obstacles to react to there are four remote controlled Jeeps. They can either stand in the way of the autonomous car, limit the range of motion in a parking situation or can be driven by (conventional) radio remote control to simulate other traffic. This way the behavior of the autonomous car in unexpected situations can be tested and it can be investigated how it might react to insecure drivers up to lunatics, how accident prevention can be trained by the car and so on.

Chapter 4

Architectural Aspects

The development of an autonomous driving system requires some architectural decisions concerning the system's functionalities and their implementation.

The architecture of the system is a mixture of bottom-up and top-down design. Because of the use of agile project management, a bottom-up draft is a reasonable choice. However, some components concerning safety of the autonomous driving system had to be planned in advance.

This chapter shows different views of the software architecture for a better understanding of the different aspects of the project group's work.

A description about the relevant aspects of the components of the system and about ADTF, the framework used for the implementation of the system's components, is given. Later it is explained, how the implemented components interact with each other.

Further on, the architectural styles used in the software architecture are presented and their impact on the project's work is explained.

Additionally, the support of non-functional properties of a system by the software architecture is discussed.

Subsequently, the implemented approach to increase safety in the system is illustrated. This especially includes the emergency brake.

The conclusion of this chapter is the discussion about which aspects can be considered to evaluate the software architecture and an evaluation of the software architecture developed during the project.

4.1 Architectural Views

The following sections introduce the functional and logical views of the developed autonomous driving system which describe the system's functionalities. Additionally, it is explained how these functionalities are implemented as a chain of filters.

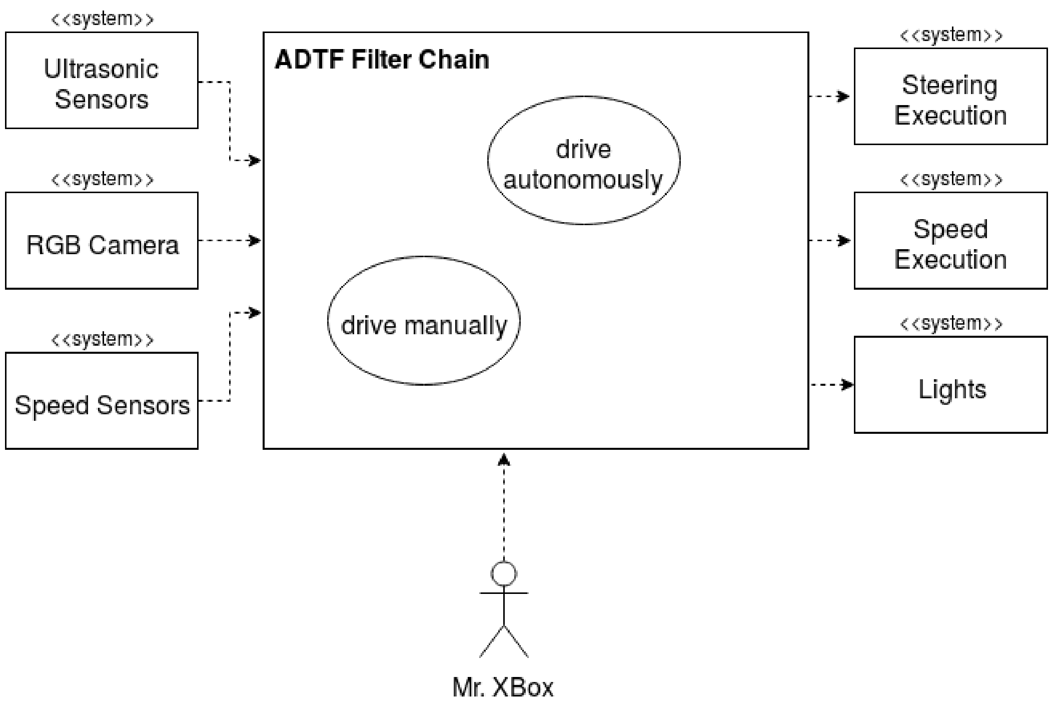


Figure 4.1: Functional view on the autonomous driving system

4.1.1 Functional View

To gain a functional overview of a system, the required functionalities have to be determined first. The main functionality which the autonomous driving system has to fulfill is to drive autonomously. For safety and testing purposes it is important, that the functionality of driving manually is implemented into the system.
The requirements presented in section 2 specify these functionalities. In doing so, different requirements are relevant for each of the functionalities.
Figure 4.1 shows a functional view of the system. The functionalities *drive autonomously* and *drive manually* are delivered through an ADTF filter chain (see section 3.2), which is described in detail in section 4.1.2.
In the following, both of the functionalities are described.

Drive Autonomously

The relevant requirements for the functionality *drive autonomously* are the requirements from the groups `SEM`, `CON`, `BEN`, `SAF`, and requirements GEN-SNX-11 to GEN-SNX-13 from the group `SNX` (see section 2.1 and section 2.2).
The functionality *drive autonomously* uses inputs from the ultrasonic sensors, the RGB Camera and the speed sensors.

Given the information about the vehicle's speed and the image of the environment including the distance to possible obstacles in front of the vehicle, the functionality *drive autonomously* enables the car to accelerate and drive forward. Furthermore, this functionality includes steering to stay on the street and to turn on intersections following the street if the RGB camera detects a turning sign. Another task which belongs to this functionality is the ability to control the turning lights and the brake light. The functionality *drive autonomously* includes the ability of the vehicle to perform an emergency brake if an obstacle is detected by the ultrasonic sensors and to stop if the RGB camera detects a stop sign.

To provide these functionalities, it outputs an angle as a steering value, a percentage of engine power as a speed value, and a signal to turn on and off the lights. The requested commands are then executed by the Arduinos in the car.

Drive Manually

The relevant requirements for the functionality *drive manually* are the requirements GEN-SNX-1 to GEN-SNX-9 (see section 2.2).

In contrast to the functionality *drive autonomously*, the functionality *drive manually* uses just the information given by *Mr. XBox*, representing the XBox controller, as input. See section 6.4.3 for more information about the XBox controller.

So, this functionality includes the ability to control the vehicle via the XBox controller. With the controller it is possible to turn on the functionality *drive autonomously*. Furthermore, the XBox controller can be used to steer, accelerate, and brake manually. Also, the turning and brake lights can be turned on and off with the XBox controller. However, the most important task of the functionality *drive manually* is to override the commands of the functionality *drive autonomously* with its own commands. This ensures, that *Mr. XBox* is always able to regain control over the vehicle in case of danger for humans or the driving system itself.

To fulfill the included functions, the functionality *drive manually* has the same outputs as the functionality *drive autonomously*, namely an angle as a steering value, a percentage of engine power as a speed value, and a signal to turn on and off the lights. The requested commands are executed by the Arduinos in the car and are preferred over the commands from the *drive autonomously* functionality as it is required by GEN-SNX-6 and GEN-SNX-7.

The next section describes how the functionalities *drive manually* and *drive autonomously* as well as the included function of the emergency brake are utilized within the implemented filter chain representing the logical view of the system.

4.1.2 Logical View

The focus of the logical view is the structure of the software. In contrast to the functional view, the logical view describes the actual decomposition of the functionality into detailed components.
Furthermore, depending on the intended use, different architectural styles are therefore reasonable for the design of the logical view of the system. Since the development of an autonomous vehicle requires a variety of functions, such as image recognition, car movement or path planning, a software architecture for autonomous driving requires a mix of different architectural styles.
The single components of the autonomous driving system are described in the corresponding read-mes, which is explained in section 7. So, this section focuses on the description of the whole implemented filter chain as well as the architectural styles that are used to segment this filter chain. Furthermore it is explained, which requirements should be fulfilled by the different filters.

Overview of the System

The architectural style that is used in the implemented autonomous driving system for a rough subdivision is a kind of layered architecture. The special feature of the layered architecture is that the individual components of the system are arranged in different layers, which handle different functionalities. In this way, the layer architecture allows a division of responsibilities in a system.
The division into these layers represents a kind of separation of concerns, which helps to gain a better understanding of the system, to counteract a fast growing system complexity and to reasonably divide the tasks. This makes the system more manageable and allows a better division of the project group in smaller teams.
Figure 4.2 shows an overview over the logical view of the implemented system. The inputs *ultrasonic data*, *image* from the Basler camera and *speed* are passed to the *input normalization* layer on the left. Then they are transmitted to the *action planning* layer, which passes the data on to the *control consolidation* layer. The processed signal is then sent to the *output normalization* layer which is shown on the right in Figure 4.2.
Another pattern used in the system is the adapter pattern. As the developed system has to be able to process data from the simulation environment VTD as well as from the real world, the filters which preprocess the input data (*input normalization* layer) and postprocess the output data (*output normalization* layer) are designed as adapters.
To allow the system to run independently of the chosen platform (simulation, physical car) an abstraction for both sensing and control instructions was defined and built. Sensing represents the input to the system and control instructions are the system's output. The abstraction mostly transforms the input data into a mutual data format and maps and interprets the output data for the respective output platform.

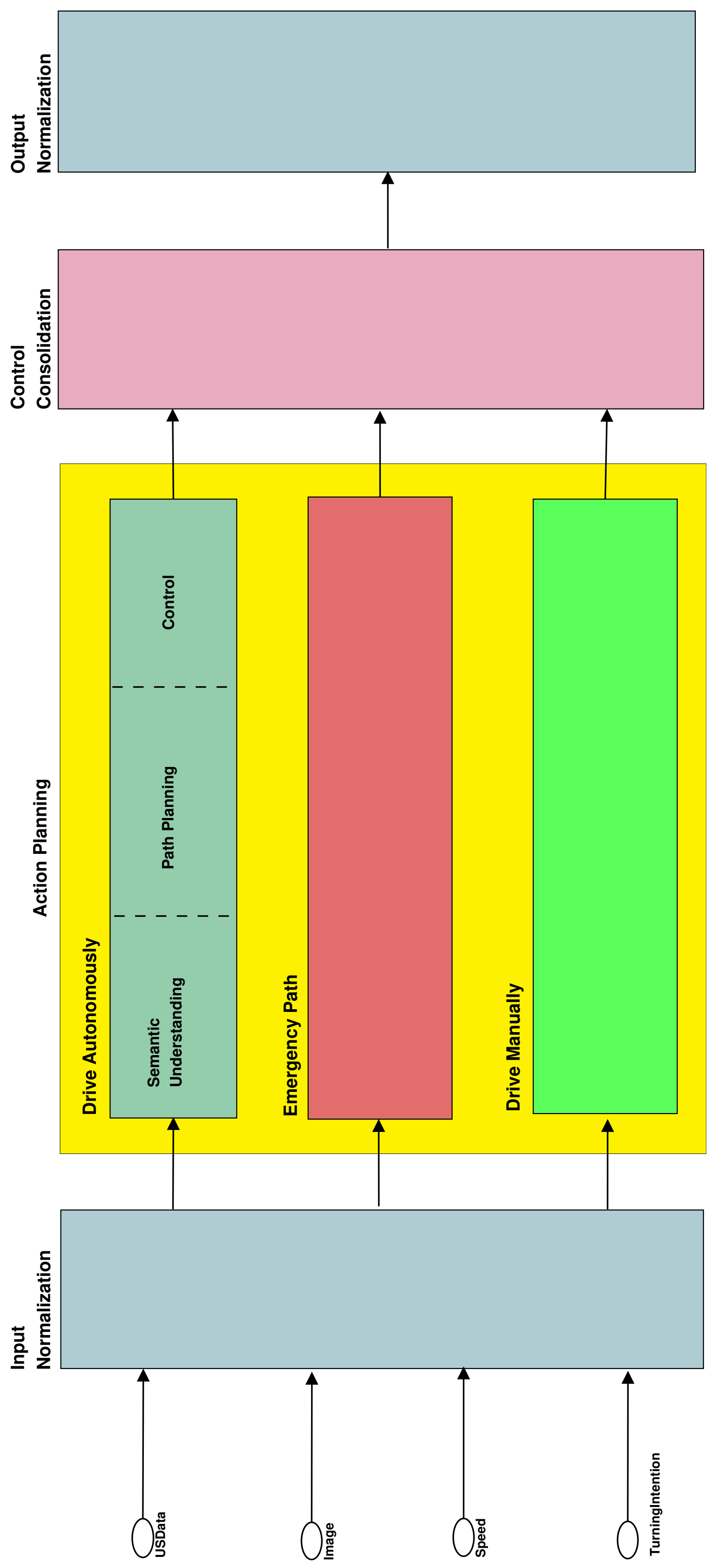


Figure 4.2: Layers of the implemented system

So, the normalization of the input data given from the sensors, the camera, and the XBox controller happens in the adapter-like *input normalization* segment shown in Figure 4.2.

Input normalization includes functionalities concerning the sensors and cameras of the autonomous vehicle.

The *input normalization* layer passes the data on to the *action planning* layer.

As explained in section 4.1.1, the functionalities *drive manually* and *drive autonomously* as well as the emergency brake, which is necessary for safety, are separated into three different sublayers of the *action planning* layer as it is visible in Figure 4.2.

The sublayer *drive autonomously* is additionally subdivided into three different partitions as it contains functions from different domains which are needed for the vehicle to drive autonomously.

In this layer, the normalized input is processed in the partition *semantic understanding* first, which contains functionalities concerning the semantic processing of the information perceived by the sensors and the cameras, like the recognition of a drivable lane.

The partition *path planning* uses the information given by the *semantic understanding* for deciding which destination the autonomous vehicle should head to.

The partition *control* in the *drive autonomously* layer includes functionalities for the computation of the maneuvers, which have to be executed for reaching the destination computed by *path planning.*

The second sublayer shown in Figure 4.2 is the *emergency path.* This layer includes just the function that the vehicle stops if there is an obstacle in front of it.

The third layer includes the implementation of the functionality *drive manually.* So, in this layer the commands from the XBox controller are received. The slimness of this layer enables it to react fast in case there is a need to control the driving system manually.

The commands for speed and steering from all three sublayers of the *action planning* layer are then transmitted to the *control consolidation* segment as it is shown in Figure 4.2. The main function of this partition is to decide, which of the commands should be executed. The commands from the *emergency path* are considered first to ensure the safety of the system. Moreover, the commands from the layer *drive manually* overwrite the commands from the layer *drive autonomously* as is already described in section 4.1.1.

The chosen commands for speed and steering are passed to *output normalization*, which is an adapter-like segment just as *input normalization.* The *output normalization* segment is responsible for the hardware abstraction of the speed and steering commands.

However, the layered architecture is used only for a rough division of the system, while the fine subdivision is dominated by the pipes and filters architecture.

As explained in section 3.2, the chosen framework for the project is ADTF, where all functionalities are realized by filters. So, the pipes and filters architecture style is the natural choice for designing the system. In this architectural style several filters are connected by pipes. The input of a filter is processed by this and the result is passed

on to the next filter. This way, sequential processing of data streams is possible, as it is especially suitable in the processing of video data.
In the following, the implemented filter chain as well as the communication and the data flow between the components are described in more detail.

The Filter Chain

The filter chain shown in Figure 4.3 can be divided into the layers and partitions presented in section 4.1.2. The implemented filters are either data-triggered or time-triggered. For a better overview of the system the filters in the figures in this section are marked with a T if they are time-triggered and with a D if they are data-triggered. For more information about this aspect of the filter chain, see section 4.3.2.
The filters in the *input normalization* layer are needed to prepare the input signals for their further processing to enable the autonomous driving system to fulfill the requirements stated in sections 2.1 and 2.2.
The *input normalization* layer contains the `PG618_VTDSensing` filter which is used to normalize the input data for VTD. Figure 4.4 shows the `PG618_VTDSensing` filter which outputs the *speed*, the *image* and the *ultrasonic sensor data*. These outputs are passed on through the filter chain shown in Figure 4.3.
Furthermore, the `PG618_USSSmoothing` filter for smoothing the ultrasonic signal to reduce the noise can be classified into the segment *input normalization*, too. It is also possible to use two different `PG618_USSSmoothing` filters if different parameter values should be used to smooth the input for the `PG618_HistogramBasedPathPlanning` filter and the `PG618_EmergencyBrake` filter.

The normalized and smoothed signals are then passed on to the *action planning* layer.
The *action planning* layer contains three sublayers. One of these is the *drive autonomously* layer, which contains the most filters because of the complexity of its functionality. The *drive autonomously* layer itself is subdivided in three different partitions again. These partitions are *semantic understanding*, *path planning* and *control*.
To the partition *semantic understanding* belongs the `PG618_Segmentation` filter used for the segmentation of the received image. The filter `PG618_ChannelExtractor` uses the segmented image and decides which channel is used for the transformation into a bird's-eye view by the `PG618_BirdsEyeView` filter. So, the filters `PG618_Segmentation` and `PG618_ChannelExtractor` are used to fulfill the requirements GEN-SEM-1 to GEN-SEM-4. The `PG618_BirdsEyeView` filter is implemented to fulfill the requirements GEN-VIS-4 to GEN-VIS-6.
The `PG618_StreetSignExtractor` filter uses the segmented image for the identification of the perceived street signs. These street signs are cut out from the original Basler camera image. The `PG618_StreetSignExtractor` filter sends the resulting cutouts to the `PG618_CutoutSignClassifier` filter for the classification of the identified signs. The

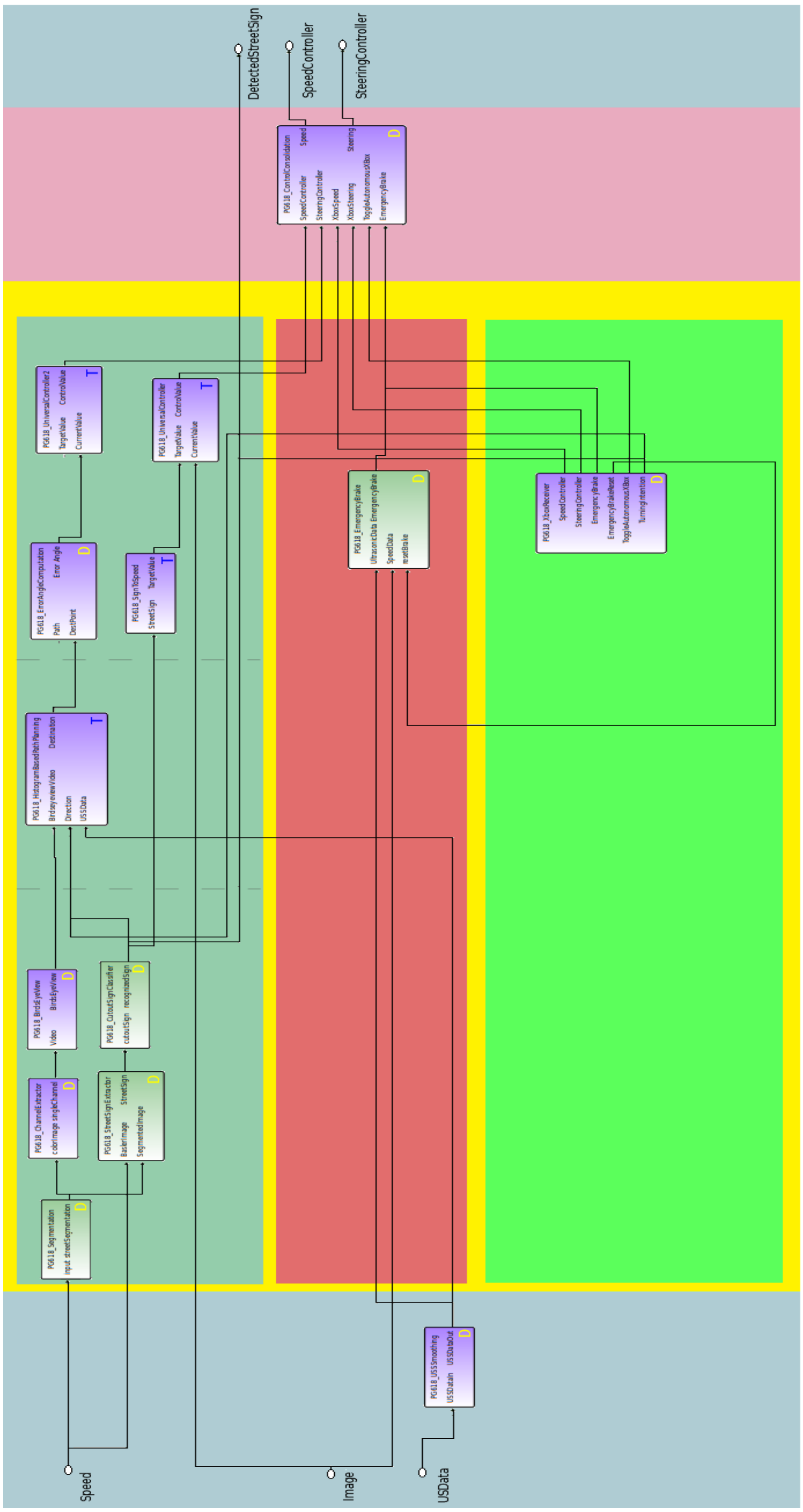


Figure 4.3: Implemented filter chain for the autonomous driving system

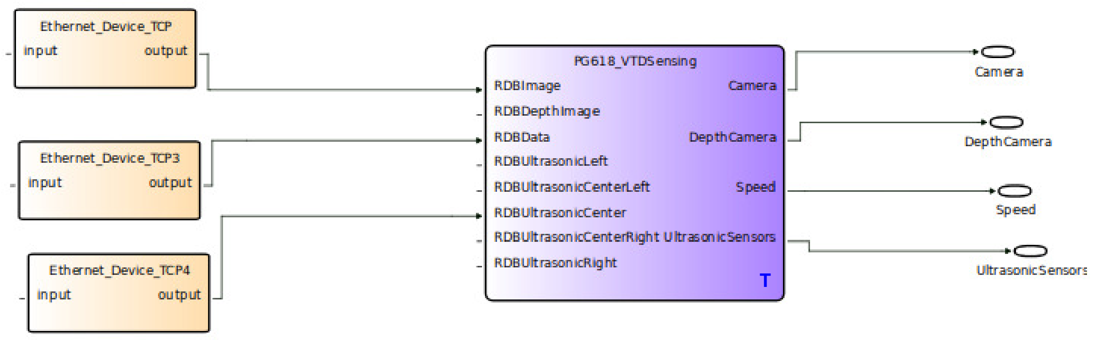


Figure 4.4: Sensing filter for VTD

functions of these filters are explained in chapter 5.4. These two filters should realize the requirements GEN-SEM-5 to GEN-SEM-7.

In the *path planning* partition of the *drive autonomously* layer, the bird's-eye view and the information about the direction, which the autonomous vehicle has to choose according to the classification of the identified street sign, as well as the smoothed signal from the *input normalization* segment are used by the `PG618_HistogramBasedPathPlanning` filter for the computation of a destination point. Alternatively, it is also possible to delegate a turning intention to the `PG618_HistogramBasedPathPlanning` filter by using the XBox controller, which fulfills requirement GEN-SNX-5.

As explained in section 6.2.2, this destination point is used in the *control* partition of the *drive autonomously* layer for the computation of the orientation error of the vehicle by the `PG618_ErrorAngleComputation` filter. The `PG618_SigntoSpeed` filter adapts the desired target speed for the universal controller with regard to the detected street sign. So, the desired target speed is set to 0 if a stop sign is recognized and to a low speed if a turning sign is recognized. The `PG618_SigntoSpeed` filter should realize the requirement GEN-CON-14.

The both `PG618_UniversalController` filters in this partition are responsible for the computation of the control values for steering and speed. The `PG618_UniversalController` filter is implemented to fulfill especially the requirements GEN-CON-1, GEN-CON-2 and GEN-SNX-13.

The filters `PG618_HistogramBasedPathPlanning`, `PG618_ErrorAngleComputation` and `PG618_UniversalController` are as a whole responsible for the fulfillment of the requirements GEN-CON-1 to GEN-CON-8 as well as the requirement GEN-CON-13.

The filters from the partitions *path planning* and *control* in the layer *drive autonomously* shown in Figure 4.3 should also be able to fulfill the requirements GEN-CON-9 to GEN-CON-12 and the requirements GEN-BEN-1 to GEN-BEN-4.

As safety is a paramount aspect of the project group's work, the filter chain in Figure 4.3 includes an *emergency path* sublayer, too. The *emergency path* layer contains the `PG618_EmergencyBrake` filter. The emergency brake filter receives the smoothed ultrasonic sensor data and the speed data from the *input normalization* segment and decides whether the autonomous car has to stop driving. The requirements fulfilled by the `PG618_EmergencyBrake` filter are SAF-1, SAF-5 and SAF-6.

The *drive manually* sub layer of the *action planning* layer contains the `PG618_XboxReceiver` filter. This filter enables an intervention in the filter chain through an XBox controller. So, the `PG618_XboxReceiver` filter implements the requirements GEN-SNX-2 to GEN-SNX-9 and the requirements SAF-2 to SAF-4.
Figure 4.3 shows that the *emergency path* and the *drive manually* layers have a much shorter pipeline than the *drive autonomously* layer. This is reasonable as it enables the system to react quickly if there is an obstacle or if there is a need to intervene with the XBox controller.

The segment *control consolidation* contains the `PG618_ControlConsolidation` filter, which is responsible for the last filtering of the received data from the three layers of the system *drive manually*, *drive autonomously* and *emergency path*. Its outputs are steering and speed values. The `PG618_ControlConsolidation` filter works according to the principle of prioritization. If the emergency brake is triggered, the filter outputs a negative speed value which lets the vehicle brake. As long as the emergency brake is not triggered, the output values for speed and steering come either from the `PG618_XboxReceiver` filter if the vehicle should be controlled manually or from the `PG618_UniversalController` filters if the autonomous driving function is activated and there is no further command from the XBox controller after the activation of the autonomous driving function. In case there is another command from the XBox controller, the steering and speed values from the `PG618_UniversalController` filters are overwritten by the XBox controller commands. So, the function of the `PG618_ControlConsolidation` filter is the fulfillment of the requirements GEN-SNX-1 to GEN-SNX-5.

The speed and steering values are passed to the *output normalization* segment, where they are abstracted and used to address the corresponding Arduinos in the vehicle or to control the vehicle in VTD. The filters in the *output normalization* layer are responsible for fulfilling the requirement GEN-SNX-1. See section 3.5 for more information about the creation of an unified interface for the simulation and real car data.
Additionally, Figure 4.5 shows a detailed overview over the implemented filters for the *output normalization* of the real car and their interaction with the AADC filters. The speed and steering values transmitted from the `ControlConsolidation` filter are normalized by the `PG618_NormalizeSpeed` and the `PG618_NormalizeSteering` filters and sent to the

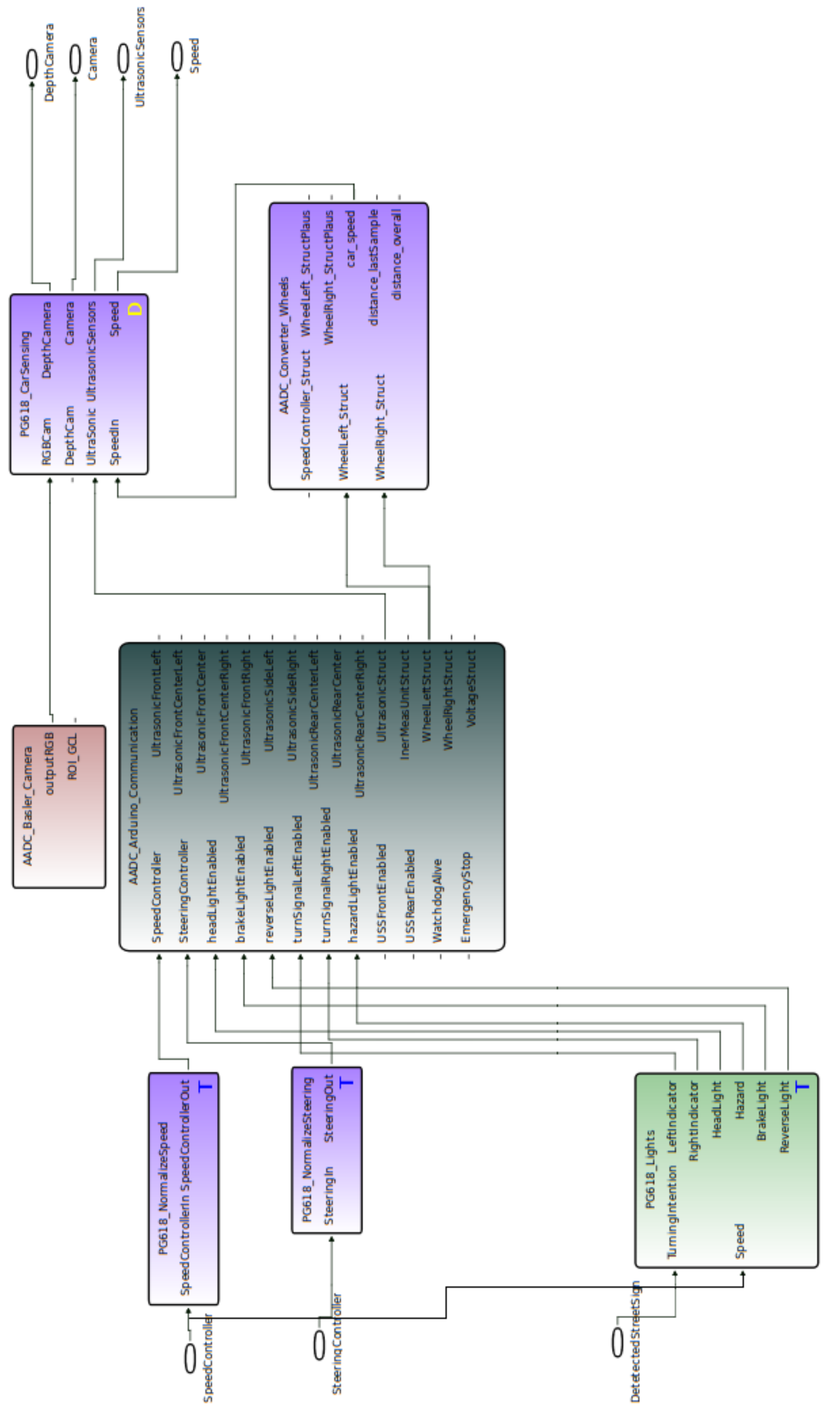


Figure 4.5: Implemented filters chain for output normalization for the real car

AADC filters for further handling. Meanwhile, the `PG618_Lights` filter gets the speed value from the `PG618_ControlConsolidation` filter to control the braking lights and handles the turning intention given either by the detected street sign or by the XBox controller to control the turning lights of the vehicle. So, the `PG618_Lights` filter fulfills the requirements GEN-SNX-10 to GEN-SNX-12.

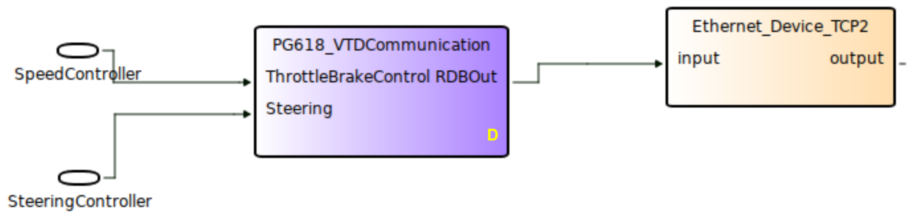


Figure 4.6: Implemented filters chain for the output normalization for VTD

Analogical, Figure 4.6 shows the transmission from the speed and steering values to the `PG618_VTDCommunication` filter which connects the filter chain of the implemented system to VTD.

As the project group has to be able to test, debug, and present the results of the implemented filter chain, the computations of the filters are visualized inside a simple web application: Each filter sends data for visualization to a message queue on a message broker (RabbitMQ) using AMQP. The web application then consumes those messages using the STOMP protocol via websockets and displays the data. This web based approach enables much better visualization possibilities then implementing visualization inside ADTF: It allows faster iteration and eases the building of the interface enormously. By this visualization, the requirements GEN-VIS-1 and GEN-VIS-2 as well as the requirements GEN-VIS-4to GEN-VIS-6 are fulfilled. An example for the visualization of the system's functionalities is shown in Figure 6.10.

4.2 Safety

The software architecture has an important influence on the non-functional quality features of a system. In systems of automobiles, especially in autonomous driving, safety and also reliability are particularly important quality features. Reliability includes the availability, the recoverability and the fault tolerance of the system.

This means that the autonomous driving software architecture must focus on the safety and reliability of the autonomous driving system. These quality characteristics can be ensured by different mechanisms.

Probably the most important safety mechanism used in the autonomous driving system is the emergency brake presented in section 4.2.2.

Another frequently used mechanism is the self-adaptation. Self-adaptive systems may induce their own gain or slowdown or adapt their steering angle using feedback loops. This mechanism is used in the implemented autonomous driving system for the regulation of the steering angle as is explained in section 6.3. The self-adaptation supports safety by including the current state of the system in the computation of e.g. the current target speed or the steering angle, which is needed to approach the next destination point. So, the system can work more stable and the safety is less endangered.

Furthermore, a kind of a timeout mechanism is included in the path planning filter to ensure, that the autonomous system reacts reasonably according to safety when there is no new image input for a period of time, as it is described in section 6.2.4. In this way the timeouts can help to reduce the probability of damaging the system in a critical situation.

4.2.1 Ensuring System Safety

The focus of this project group is achieving autonomous driving capabilities. Therefore, a pre-built vehicle (see section 3.1) is used that already offers a high-level API. Any safety considerations below this level, for example anything concerning the Arduinos directly, is not considered here. Instead, the safety risks all pertain to the process of registering sensor inputs, processing them, and generating new commands that are sent to the Arduino control filter. The commands relevant to safety are those governing the steering and driving speed of the car. For these respective values, a control functionality has been implemented which enforces the changes in steering angle and speed do not exceed a certain threshold. This ensures smoother transitions, reducing the physical stress on the system, thereby also reducing the risk of any structural failures in the vehicle or the environment, which could lead to accidents. Furthermore, the movement of the car becomes more predictable. This should make it easier to plan future steps by increasing planning accuracy, thereby indirectly ensuring safety. Additionally, any other road users will also be able to predict the system's behavior more accurately, enabling them to plan better, and decrease the likelihood of accidents.

Lastly, the system's safety is ensured by implementing an emergency brake functionality based on ultrasonic sensor values (see section 4.2.2). For control personnel interventions, a remote control via XBox controller is implemented, including braking, to override the autonomous decisions of the car (see section 6.4.3).
A functionality that could be implemented in future projects is checking the plausibility of one sensor's measurements by juxtaposing it with the measurements recorded by other sensors. For example, the ultrasonic sensors report rather noisy signals (see section 3.1.5). Therefore, if an object is registered at a distance in front the car, one might cross-examine this fact with regard to the images of the depth- and color images.

4.2.2 Emergency Brake

To establish a basic level of safety, a simple emergency brake is implemented. It is based solely on the measured distance of the distance between the front-center ultra sonic sensor, and an obstacle. This is to keep it simple and have less false emergency braking due to objects near the road. If this distance is calculated to be within the critical stopping distance (see below), the car applies braking action (setting the target speed to 0 acts like an electrical brake, the hardware lacks mechanical brakes). The space the car requires to come to a complete stop is highly dependent on its velocity v. Other factors like road surface, tire wear and specific characteristics of the (virtual) brake mechanism are neglected for simplicity. If the measured distance in meters would fall below v^2, a safe stop could not be guaranteed. This formula is not found in scientific literature, but used in driving schools to calculate the braking distance without reaction time. Cho et al. [2006] can be used to confirm this as an upper threshold. Due to the noisy sensor data (see type 3 in section 3.1.5) the car needs to brake earlier. Therefore a fixed safety factor f is introduced. The critical distance in centimeters (the ultrasonic sensors output centimeters) is now below $(v^2/100) * f$.
The simplicity of this emergency brake implies strong limitations: obstacles are detected only in direct vicinity straight ahead of the vehicle, regardless of whether its course would clear the obstacle. So in fast cornering with close-proximity objects, the brake might be activated. Vice versa the car might drive into an obstacle without braking at all, because it was not directly in sight of the front-center sensor. While these faults can be improved with the consideration of more sensors and the orientation of the front wheels, there are more systematic limitations.
Using the ultra sonic sensors to detect obstacles is not very reliable. The data is noisy (see section 3.1.5) and therefore must be smoothed. The braking algorithm uses an average of the last k measured distances; a larger k means that it takes longer for measurements to have an impact on the average, and therefore on the brake algorithm. A smaller window means outliers could set off the emergency brake erroneously. Experiments showed that the bigger reaction time due to smoothing was successfully compensated for by a larger safety factor (earlier braking).

4.3 Evaluation

The selection of a software architecture that supports the requirements of an autonomous vehicle system stated in section 2 always requires a balancing of different quality criteria as it is not possible to achieve an overall optimal solution. Accordingly, the evaluation of a software architecture is difficult, because there is a lack of standardized metrics to evaluate the overall quality of a software architecture.
However, the chosen filter structure shown in section 4.1.2 can be evaluated on the basis of the following aspects.
In this section, the created filter structure is evaluated according to the data and information flow in the system. Furthermore, the system is evaluated under the aspect of reusability and it is explained, why this structure was preferred over possible alternatives. The last section discusses the advantages and disadvantages of the usage of time-triggered or data-triggered filters.

4.3.1 Evaluating the Flow of Data and Communication

As always when considering software architectures, the data flow and the communication flow are important analytic criteria.
When examining different architecture alternatives with regard to interesting quality characteristics, the data flows and communication flows in the system are observed. These allow to examine if there is a single-point-of-failure in the architecture of the system.
Furthermore, insight to data flows and communication flows help to decide how to build the system safely. These analysis criteria are essential for the decision, where to implement components like the emergency brake to achieve higher safety. Because the data flows and communication flows in this case can be used to estimate the system's response time to an obstacle.
To evaluate the software architecture, the broad high-level separation into three functional areas is explained and evaluated first. Afterwards, the decisions made during the construction of the low-level architecture within those areas are discussed.

Evaluating the High-Level Structure

The shown filter chain in Figure 4.3 is the result of a continuous deployment. At the beginning of the project, the developed filters included a lot of different functionalities as they grew with the increase of assessed knowledge of the project group's members. It soon became visible, that the powerful filters had many disadvantages. The filters were difficult to understand and to maintain, which is the reason why they were refactored into smaller components during the project.
Especially the filters that belong in the domains *path planning* and *control* were in focus of the refactoring as the functionalities of path planning and the vehicle control were part of

a single filter first. So the refactoring was done in order to make the filters more readable, understandable, maintainable and universal.
The less powerful filters with the decreased functionalities allow an easier integration of more complex algorithms. This is important for functionalities in different domains like path planning and vehicle control, with progression of the project more complex algorithms were developed for handling the complexity of the autonomous driving functionality. The newly developed algorithms could be easily integrated in the complete filter chain.
On a broad level, the chosen architecture is separated into six parts: execution (or simulation thereof), sensing, semantic understanding, path planning, control, and safety. The execution deals directly with the car, implementing the driving directives, and the sensors. The semantic understanding interprets the sensor data from the sensing layer to gain knowledge about the environment. The path planning layer computes a destination and the control uses the knowledge from the semantic understanding layer and the destination to generate driving directives for the executing components, while the safety layer ensures that there is no collision with an obstacle.
This separation directly supports the data flow requirements given by the task. Furthermore, failures in one of the modules can be treated separately from failures in the others. Therefore this architecture is to be rated very positively in terms of data flow and also in terms of separation of responsibility. The alternative would have been a more integrated approach in which this separation is less clear. Such an approach would have suffered in both of those criteria. However, it might be faster since the transmission of the data from one submodule to the other might not be necessary.
The separation is also desirable from the perspectives of project management and software development. Since the separation of tasks is clearly integrated in this architecture, the development team can be separated into different teams as well. After defining the interfaces between the submodules, each team can then use their full productivity to work at least partly independently of each other towards the common goal.
Furthermore, the physical car which is used, is meant to be operated via the ADTF environment (see section 3.2). This framework naturally promotes a separation into smaller submodules (or filters, in ADTF terms) and handles the asynchronous transmission of data between those different modules. Therefore, any approach that would not utilize these structures would either not be able to take advantage of ADTF's full capabilities, or require taking a completely different approach. Such an approach would probably require starting at the Arduino level of computation, requiring a lot of extra work.

Evaluating the Low-Level Architecture

One problem that the project group had to solve is that the physical car in a physical environment as well as the simulated car in a simulated environment should be handled in a similar fashion. Following the separation of tasks approach mentioned in the previous section, the goal was to transmit the exact same type and format of data to the semantic understanding module, no matter which environment and which car were currently used. The two tasks were separated again, creating completely distinct sets of ADTF filters to deal with either the physical or virtual environment. The alternative would have been to create an adapter in the form of one filter to deal with both cases and have an internal switch mechanism to choose which kind of routines to execute, based on the environment the vehicle should drive in.

The chosen approach has several benefits. Again, any errors in one of the submodules do not affect the functionality of the other. Regarding data flow, there is a clear separation of tasks, which is conceptually desirable. Another benefit is that when driving in one environment, there is no need to load the code dealing with the other environment into the memory. This reduces the needed memory and thereby improves and potentially speeds up the performance, depending on the size of the code and the available memory.

The components of the control layer need to use knowledge gained by the semantic understanding and generate valid driving instructions based on this knowledge. In addition to the autonomously computed driving instructions, a direct control via an XBox controller should be enabled. This direct control is prioritized over the autonomously computed instructions to allow for intervention (see section 6.4.3).

While this was not desirable in the execution layer, the control layer implements a switching mechanism that forwards the XBox control signal, if there is one, and the internally computed one otherwise. In terms of separation, this is less elegant than the approach taken in the simulation module. However, it has to be this way, considering the fact that the requirements are very different in this case. The integration of the XBox controller enables spontaneous interventions by the user during operation. This has very strong positive effects on the system safety since the user can anticipate unsafe situations and initiate early countermeasures.

The second big decision in terms of control architecture concerns the integration of the emergency brake (see section 4.2.2). Most driving control signals are computed within one system of filters that include the described XBox controls, switching depending on whether it is available, and checking whether the generated signals lie within a permitted range. The automatic emergency brake has deliberately been moved out of this system. It uses a simple heuristic based on the ultrasonic sensors to determine whether an emergency brake should be initiated or not. Since this functionality is highly critical for system safety, it has been moved out of the usual loop to ensure that the signal to brake gets transmitted as quickly as possible. This architectural decision seems less desirable since two different parts of the system generate control signals, instead of one. However, this trade-off can be accepted for the big gains in safety.

4.3.2 Evaluation of Data- or Time-Triggered Decisions

When implementing filters in ADTF, there are two different kinds that can be considered for this project. The data- or event-triggered filters perform actions only when they receive input data. Time-triggered filters, on the other hand, have a set time interval, after which their action is performed.

An advantage of the event-triggered approach is that the architecture can perform computations immediately when new information is available. This means that it can react quickly to new events, thereby potentially increasing safety. In addition, there is no need to take care of any queues that buffer incoming data and signals. This simplifies the programming process and removes opportunities to accidentally introduce errors.

The big disadvantage of the event-triggered approach is that there is no direct way to detect a sensor failure. If no data is sent by the defective sensor, the filter does not perform any actions, so it also cannot check whether the sensor has failed. In the worst case, this means that the car does not receive any more steering or speed control values, follows the past orders, and crashes. Additionally, the computations might be performed much more frequently than required, thereby wasting resources. On the other hand, it is possible to save resources if the speed and steering values are computed less frequently than a time-triggered filter might.

Time-triggered filters guarantee for example a steering command to be given every 10th of a second. However, they require queue-handling, and predetermining a timing interval, which might be hard in certain cases. Underestimating the interval can lead to problems by not processing incoming data quickly enough, while overestimation wastes computational resources.

Considering these arguments, data-triggered filters were chosen to be the default for the project, mostly for their ease of usage. However, in the safety-critical area of sending steering and speed commands to the Arduino interface, the guarantees provided by time-triggered filters are used to ensure that the car is always being controlled. Additionally, the used controller architecture requires a time-triggered structure to limit the number of computations and thereby enable proper planning of the system's behavior.

This results in the filters `HistogramBasedPathPlanning`, `NormalizeSpeed`, `Normalize-Steering`, `UniversalController`, `SignToSpeed` and `Lights` being time-triggered. To mirror the functionality for the simulation, the filter `VTDSensing` is also implemented as time- triggered. The guarantees given by time-triggered filters could also increase the safety of the `EmergencyBrake` filter, this filter on the other hand could be data-triggered as it relies on other time-triggered filters and therefore is robust to sensor failure.

Chapter 5

Semantic Understanding

In order for the car to know what to do in the world it resides in, it needs to have an understanding of its environment. This can be achieved by capturing and interpreting sensor data. One of these sensors is the RGB camera found on top of the car or, in case of VTD, a simulation thereof. The main goal of the *Perception group* in the first semester was to detect the part of the road the car is currently driving on (from now on called the *drivable lane* or simply the *lane*). First, the decision was made to use classical computer vision methods in order to build a custom algorithm for detecting the lane. As there are a lot of well-documented solutions to this problem already, this quickly allowed getting decent results. After that, a custom convolutional neural network was built in hopes of achieving more robust and accurate results.

5.1 Simple Lane Detection

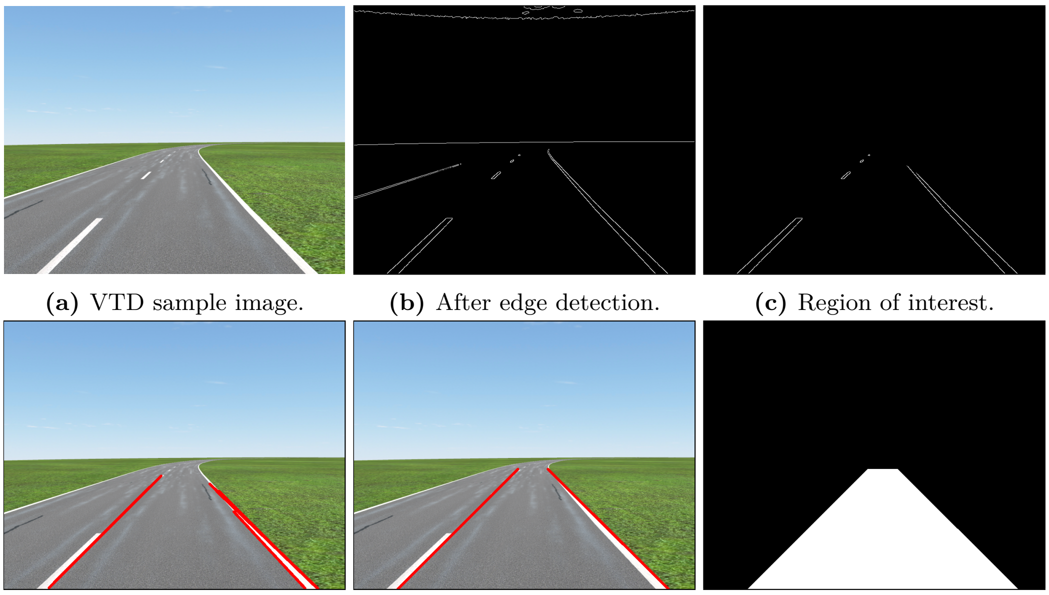
The first approach for detecting the drivable lane was a simple OpenCV-based algorithm that uses *Hough transformation* to detect the lane markings in an image. As this algorithm can only be used to extract straight lines, it is impractical for use on curved roads but was enough to get some first results to start working on a simple controller and to use as a basis for a more sophisticated approach later on (see section 5.2).

5.1.1 Algorithm Pipeline

The algorithm pipeline is based on Sqalli [2016]. It works by first extracting only the lane markings from the image, detecting all straight line segments and some cleanup afterwards. Figure 5.1 shows most of the stages an image passes through during the algorithm.

Gaussian Blur

In the first step a simple Gaussian blur was applied onto the image. This reduces noise and prevents single white pixels from being detected as part of the lane markings.



(a) VTD sample image. **(b)** After edge detection. **(c)** Region of interest.

(d) Extracted lines. **(e)** Computed lane lines. **(f)** Resulting binary image.

Figure 5.1: Stages of the simple lane detection algorithm from input to output image.

Color Filter

Next, a color filter is applied. The approach described here only uses the lane markings to detect the current line. Since these are usually white or yellow, every pixel that does not fit this criterion was removed. For white pixels the image was converted to grayscale and a threshold for light gray pixels was defined. For yellow pixels it was converted into the HSV format, which made it easier to define a range for yellow colors, since the hue is controlled by a separate parameter, while RGB images share the same values for a pixel's color and brightness.

Edge Detection

Then, a *Canny edge detection filter* was applied, which uses rapid changes in pixel values to find edges in an image. This is done because lane markings have sharp edges and, since all that is looked for are straight lines, this removes a lot of unnecessary information (see Figure 5.1b).

Region of Interest

The lane markings of the current lane are usually not the only white or yellow pixels in an image. For example, the sky tends to be bright as well and there could be parts of other lanes visible. These pixels were removed by applying a region of interest, which assumes that the lane is inside a trapezoid shape in the bottom half of the image (see Figure 5.1c).

Line Extraction

A *probabilistic Hough transformation* was used to extract straight lines from the remaining pixels. This algorithm tries to find line segments that follow edges in an image. The result of its OpenCV implementation are a set of endpoints of these segments (see Figure 5.1d).

Rotation Filter

The Hough transformation returns all candidates of possible straight lines it finds. Since some lane markings also have an upper and lower edge, these might be returned as well. There is also always the possibility of it detecting lines between the right and left edge of a line or just on some random pixels that did not get filtered correctly. It can be assumed though, that the lines which are looked for will be mostly vertical, so every line segment that is approximately horizontal is removed.

Right and Left Line Extraction

To find the left and right marking of the street, the set of edges need to get separated, which are left from filtering. So all edges in the left part of the image are ordered to the left lane edge set and symmetrical all the edges in the right part are ordered to the right lane edge set.

Median Line Computation

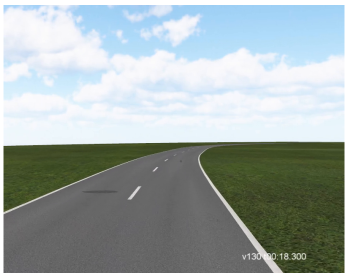
After filtering of the horizontal edges, only vertical edges are left. These are not necessarily edges, which are affiliated to one of the two lane markings. The edges need to get extended until the border of the image and the limit of a mask, in which the lane is to be sought in. Then the mean edge will be taken out of two sets, one for the left and one for the right lane, while mean is defined as:

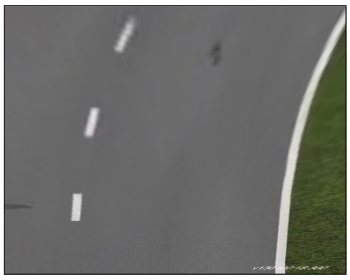
$$x_{mean} = x_{n/2} \in \{x_1, x_2, ..., x_n\}. \tag{5.1}$$

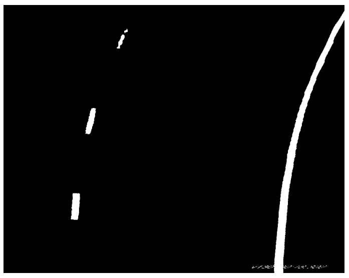
So the mean is the middle object out of an ordered set (see Figure 5.1e).

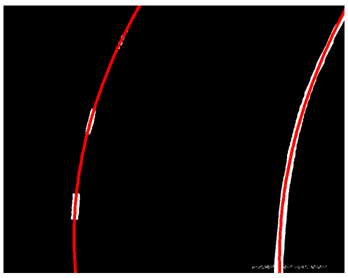
Binary Image

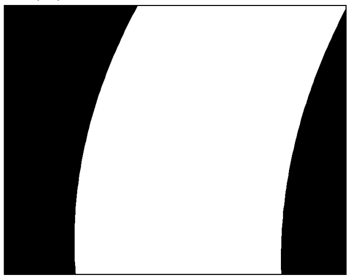
The result are the left and right lane markings, and limits given by a mask, in which a lane is sought, the polygon can be filled with the vertices of the endpoints by the lane markings to get a binary image with the drivable space on the current image. It can be interpreted as the region of interest for the steering control (see Figure 5.1f).


(a) VTD sample image.


(b) Birds-eye transformed.


(c) Extracted lane pixels.


(d) Fitted curves.


(e) Detected lane.

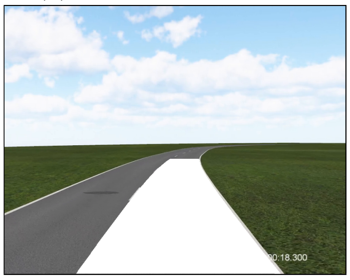

(f) Retransformed to original perspective.

Figure 5.2: Stages of the advanced lane detection algorithm.

5.2 Advanced Lane Detection

The simple lane detection algorithm worked well enough to get started but it had some obvious flaws. First of all it only worked on straight roads. Whenever it tried to detect a curve in the road, it either resulted in a very inaccurate approximation of an average angle for the whole curve or it just did not detect anything at all. It also had problems with the gaps between lane markings in VTD, which were sometimes big enough to only have very small parts of two markings inside the region of interest and thus were not detected as straight lines. There also was a more general problem with the images perspective distortion, because for the controller it would be better to have equidistant waypoints.

All of these problems were addressed with the *advanced lane detection* algorithm, which is based on Palazzi [2017] and uses a polynomial curve fitting algorithm to estimate two functions for both lane boundaries.

5.2.1 Algorithm Pipeline

While the name may suggest otherwise, the advanced lane detection algorithm actually uses less steps than the simple lane detection. Figure 5.2 shows these steps. Note that the retransformation step shown in 5.2f is just for demonstration purposes and is not actually part of the algorithm.

Perspective Transformation

In the first step the image gets transformed to what is called the *birds-eye view.* This uses a similar trapezoid region as the region of interest from before (see section 5.1.1) but instead of just removing the pixels outside it, it stretches the region to fill the whole image. To get the correct perspective a trapezoid was used that starts at the bottom two corners and follows the direction of the images principal point.

Note that, while the distances on each of the axes are equal, the distances between both axes are not. A correct representation of the road would be a lot longer, which makes the curves look steeper than they actually are.

Lane Pixel Extraction

The relevant pixels were extracted with the same color filter, which was used for the simple lane detection (see section 5.1.1). As this algorithm does not need straight edges or lines and just uses the raw pixel coordinates, no other operations were necessary.

Definition of Left and Right Lane Pixels

Since it is required to find two curves, one for each boundary line, it is necessary to distinguish between the pixels that correspond to each of those lines. To achieve this, two different approaches were used depending on whether or not the lane was already detected in the previous frame, which differ a little bit from the used reference algorithm (see Palazzi [2017]).

While the reference algorithm uses a sliding window that slides upwards starting from the points where the histogram reaches its maximum values when detecting without knowledge of the previous frame, it was decided to just define every pixel in the left half of the image as part of the left line and vice versa. This means that the algorithm is not as stable as the reference and has to start on a fairly straight road but it also made the computation a bit faster and reduced development time.

In cases where the lane was already found in the previous frame, it can be assumed that it will not drastically change from one frame to the next. This way it can use the previously calculated lines to assign pixels near one of them to that line in the next frame. This generally works pretty well but does have one unwanted side effect: lines which where detected incorrectly can get very close to each other or even overlap and thus see the same pixels as part of its own. This in turn can lead to both lines getting so close to each other that they essentially become the same line. In this case the algorithm assumes it detected two lines correctly and never even tries to find another one. This problem was solved by checking whether points of both lines get too close to, or too far apart from, each other, to represent a plausible lane.

Curve Fitting

With the pixels classified as being either part of the left or right line, a curve fitting algorithm was used to estimate a quadratic function through each of the set of pixel coordinates. The algorithm is based on the *polyfit* function found in *MATLAB*, which uses a *Vandermonde matrix* to calculate the polynomial coefficients (see MathWorks).

Binary Image

As with the simple lane detection algorithm before, the advanced algorithm also returns a binary image as the output, with the lane in white and the rest black. The image is not reprojected to the original perspective, as the birds-eye view is actually better to use for path planning.

5.3 Road detection using neural networks

To further improve the robustness of the lane detection, a switch was made from the advanced lane detection algorithm (see section 5.2) towards using neural networks to segment the drivable area. To make the task easier for the neural network, the decision was made to not only segment the right lane as drivable, but the whole road.
The software tool used to implement the neural networks is the Keras deep learning library with Tensorflow as the backend.
In this section, the following subjects will be discussed. First, the generation of the basic training data is described, both for simulated and real world environments. Then, the augmentation operations are described that increase the variation in the data and thereby should increase the robustness of the segmentation, as well as preprocessing steps like resizing or greyscale-conversion that are applied after the augmentation has taken place. Several architecture variants that were tested are introduced next. Finally, the range of experiments that assessed the usefulness of all these different parameters is described.

5.3.1 Training data

To be able to drive both in the simulated VTD environment, and in the physical environment consisting of the assembled road panels, it is necessary to generate training data for both. Ideally, the training data should contain straight roads, curved roads, T-crossings, and regular crossings. In this section, the generation of all training data that is used in the experiments is described more closely.

VTD training data

Since VTD is a simulation environment, it is possible to create a lot of training data at very little cost. While at the beginning of the project those training images were labeled

by hand, later on, a transition was made to create automatic annotations from within VTD.
For the first set of training images, a data set of 1058 (reasonably distinct) labeled images were generated. The route from which the data was generated contains straight roads, curves, T-crossings, regular crossings, and S-curves. An example can be seen in Figure 5.3. his data set is going to get referred as `vtd_crossings`. Figure 5.3 shows an example image from this data set.

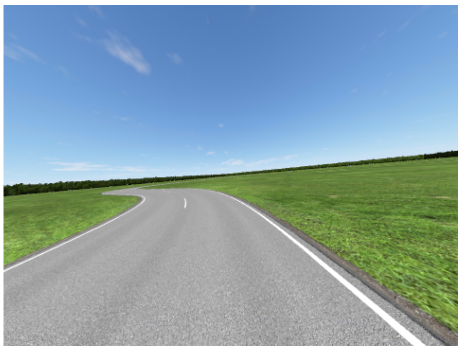
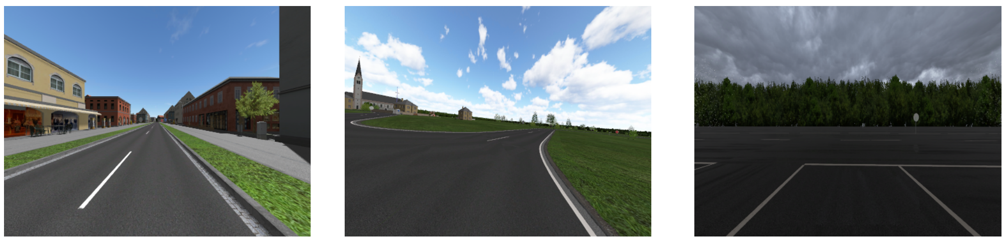


Figure 5.3: Example image in the `vtd_crossings` data set.

Since creating simulated data is comparatively cheap, three further routes were created from which training data was generated. `vtd_rural` consists of roads in a rural area, with a small town in the background and mostly green side stripes. `vtd_town` is a route through a town, with roads mostly following a Manhattan-style grid. `vtd_evaluation` is constructed like the real-life road panel track that is used to evaluate the whole system at the end of the project (see figure 6.23). From each of these tracks three different data sets were generated by varying the simulated weather conditions between `blue sky`, `rainy`, and `cloudy`. This way, it was possible to not only create a lot of variation in the data set by varying the tracks, but also by varying the sky in the background and especially the lighting condition. Training on this data set should therefore make the segmentation model more robust to these influences. Figure 5.4 shows example images of each of the data tracks and weather conditions. While the exact number of images generated from each track and weather condition varies, a minimum of 438 labeled images were created per configuration.

Automatic Generation of Test Images

In order to automatically create labels for road and signs, the same route was driven inside the simulation with three different visual databases: one containing the original route, one with the road colored red, and a final one with the signs colored red. Afterwards the images were converted to labels by converting the red colored parts of the image to white color and everything else to black color with custom written imagemagick scripts as seen in Figure 5.5.



(a) `vtd_town - blue sky` (b) `vtd_rural - cloudy` (c) `vtd_evaluation - rainy`

Figure 5.4: Example images from three data sets. The first word indicates the name of the data set, the second word indicates the weather condition.

The neural network was only trained on images created by perfectly driving on the road. In an attempt to learn to find the way back to the road while off track, a series of images was created with the car driving completely off the road. This was done by creating an invisible road the simulated car drives along, while keeping the same visual database as the original track. Later, instead of driving completely offroad, data was created by shifting the camera slightly to the left and right so the images are just slightly off the road.

Those images were by no means perfect, since coloring the road itself in a different color wasn't actually possible in the road designer. Colored versions of lane marks were available though, so instead of coloring the road directly, a red colored lane mark replaced the middle marks on the road and was stretched to the width of both lanes. Even after smoothing the resulting labels were imperfect especially on the left and right edges of the road, because the texture of the road mark was not a perfect rectangle, which was then amplified by stretching the texture to abnormal widths. Nevertheless, automatically creating thousands of training images simplified the training of neural networks for VTD and was much quicker than using the labeling tool (see Chapter 5.3.1), which still had to be used to create labels of the real world data.

Training data for road panels

As a first test case for the neural networks, a road network was assembled consisting only of a curved road. Assuming that crossings of any kind pose a more difficult problem, this data can be used to determine whether the network is able to segment the lane at all. Since the camera used by the car has a fish eye effect, an ADTF filter was used to reverse this distortion. The resulting images had a resolution of 472×312 pixels. There are 684 images all together, 615 used for training and validation, 69 used for testing. This data set is going to get referred as `rl_straight`.

Since manually annotating images takes a lot of time, a subset of 64 annotated images was used, augmented by the first augmentation techniques described in section 5.3.1 to artificially increase the number of training images.

Increasing the complexity of the test environment, a new route containing curves, a T-crossing, and a regular crossing was constructed. It was decided to not remove the fish-

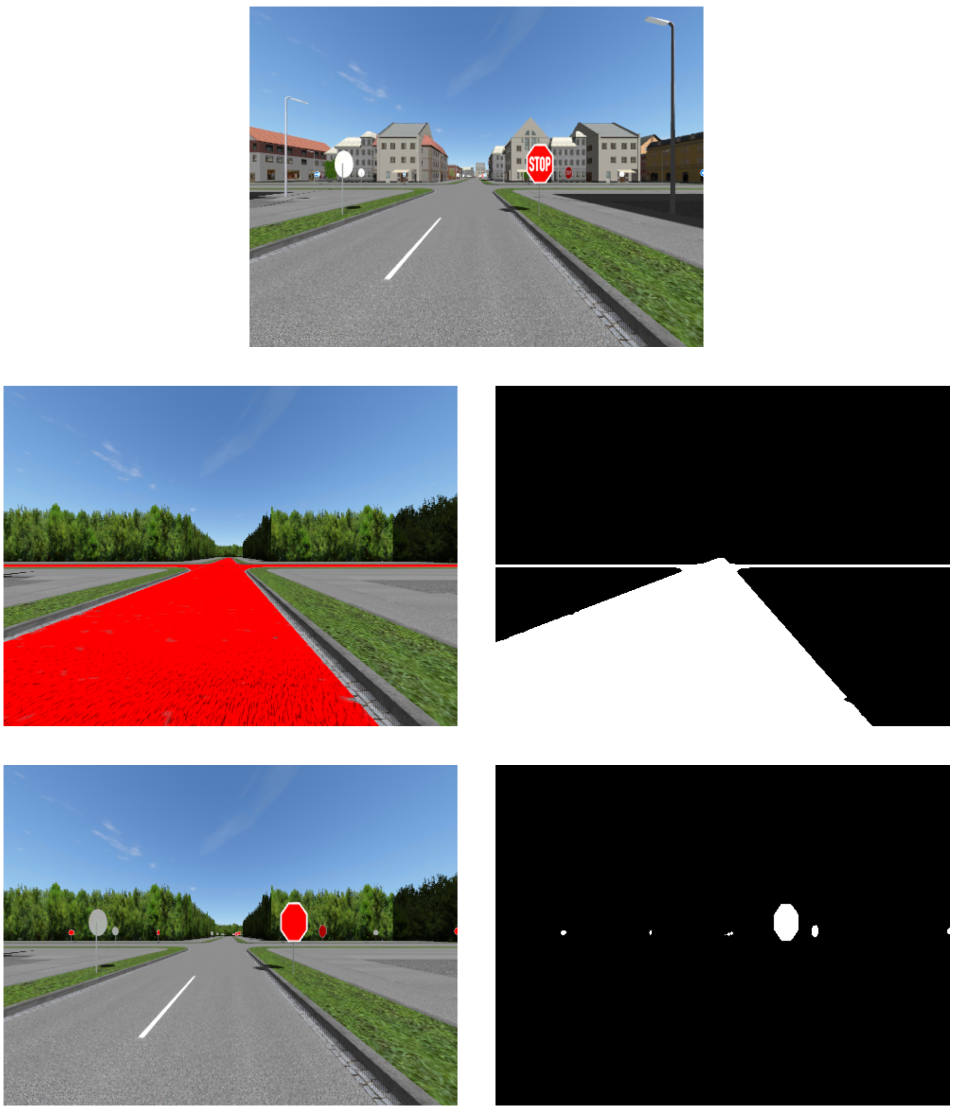
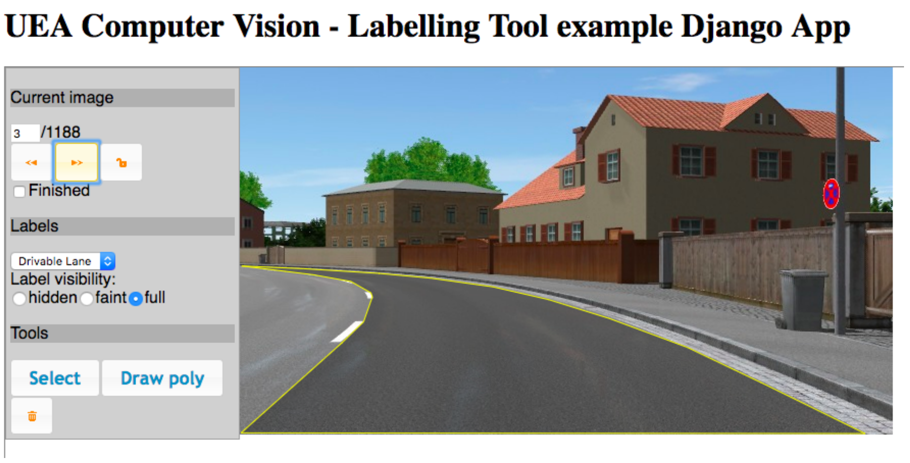


Figure 5.5: Image of the `vtd_town` test route with red colored road and signs and the generated labels

eye distortion for this data set, since removing the distortion also greatly reduces the field of view, making any control-type decisions more difficult. The images were resized to a resolution of 480×360. Using 403 manually annotated images, the data was split into 382 training images and 41 testing images. Then, the same augmentation methods were used to receive a set of 1444 images for training and validation. Test images are not augmented, as this would weaken the meaningfulness of the metrics on the test set. This data set is referred to as `rl_crossings`.
Furthermore, both data sets were combined, naming the result `rl_combined`. The images were resized to fit the smaller format, therefore some of the images have a fish-eye perspective, while others don't. For the `rl_crossings` part, the non-augmented images were used.
All images were manually labeled using the software ImageTagger (see section 5.3.1).

Labeling tool


UEA Computer Vision - Labelling Tool example Django App

Current image

3 /1188

Finished

Labels

Drivable Lane

Label visibility: hidden faint [✓ full]

Tools

Select Draw poly

Figure 5.6: The drivable lane annotated onto an VTD image.

In order to create a custom training data set, first, frames were extracted from a video that was taken in VTD. The video was long enough to create 1200 frames. The goal was to annotate the drivable area for each frame. In order to segment the drivable lane, first, it was evaluated whether or not writing an own tool would be necessary, or if appropriate tools were already available. One feature that was needed was to draw polygons onto the frames, which then get annotated with a pre-defined label. This seemed to be the fastest way to annotate the drivable lane. Also, the process of drawing the polygon as well as changing the image to the next once it was drawn in the lane needs to be as fast as possible so that the labeling time gets minimized. Most importantly, though, it was required to spread the work across the whole group so that everyone gets their own stack of images to label them. Also, the group members had to be able to label the images from home, which meant that a client-server architecture was needed.

The initial findings suggested that no tool fit the requirements. However, the closest tool available was the Image labeling tool from the computer vision department at the University of East Anglia[1]. The decision was mode to modify this tool, which was published under the MIT open source license, instead of devising a self-made solution.
The annotation results were then exported as a single JSON object which contains the coordinates of each of the the polygon vertices. Then a simple program was written which takes the original image and the corresponding JSON object and creates a black and white image where the lane is painted in white.
When it came time to start labeling drivable area and street signs, a decision was made to update the labeling tool to BitBots Imagetagger by Fiedler et al. [2018], because it allows to switch the selection of label using keyboard shortcuts, thus speeding up the labeling process immensely. Also, it was not necessary to adapt the code of the tool to the project group needs because of its extensive configuration options inside the tool itself. It is still able to export the annotation information as a JSON object, albeit in a slightly different format.

Augmentation

The Python library imgaug[2] by Alexander Jung was used to augment the group's training data. First, the images were flipped horizontally, as this does not change the semantics of what is and is not a street. Afterwards, a random rotation between -10 and +10 degrees was applied, both on the flipped and unflipped images. These steps were mostly done to increase the size of the data set, while still being very similar to the original images. With the original 64 images this resulted in 256 images, 229 of which were used for training the model, the rest for testing.
Later on, the augmentation method was extended to further increase the robustness and generalization of the network. After applying the aforementioned transformations one of three noise filters was applied to 50% of the images. These filters were:

- Dropout: Completely erases some pixels, either from all channels or separately per channel.
- Gaussian Noise: Adds values sampled from a normal distribution to each pixel.
- Gaussian Blur: Blurs the image.

Additionally between one and three of the following color modifications were applied to each image:

- Normalize the contrast by a factor between 0.5 and 1.5.

[1]`https://bitbucket.org/ueacomputervision/image-labelling-tool`
[2]`https://github.com/aleju/imgaug`

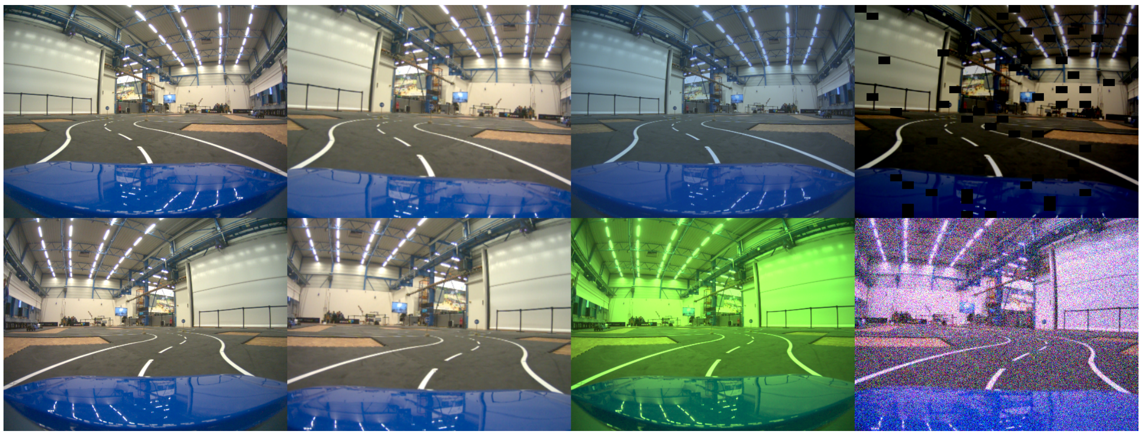


Figure 5.7: Example augmentation of a single image with applied rotation, noise and color modifications.

- Add a random value between -100 and 100 to all pixels of the image.
- Multiply each layer with a value between 0.5 and 1.5.

These modifications did not change the performance on the test set much but proved very important in actually driving on streets, as it increased performance on images, which are less similar to the training data. With all of these augmentations applied, this increased the size of the training data by a factor of 8. Figure 5.7 shows an example of the results of augmenting a single image.

5.3.2 Preprocessing

Several different preprocessing steps were tested to determine their influence on the segmentation performance of the neural network. Independently of which preprocessing steps (described below) were performed, the pixel values were scaled to [-1,1] before the images were fed into the neural network, excluding the very first experiments that were performed on the reduced `rl_straight` data set.
Since a big part of the segmentation task seemed to be to distinguish between bright road marks and dark road, a first approach was to convert the images to greyscale. Furthermore, the width and height were also reduced by 50%. Both of these transformations reduce the size of the image data, thereby resulting in faster inference. In addition to using each transformation on its own, both were also combined to receive greyscale images that were 50% less wide and high. The idea is that the greyscale transformation might strip away information not essential to the task and thereby improve predictive performance. However, it did not prove to be useful (see Figure 5.14).
A problem that might occur, is that the dark road has pixel values near zero, but should be classified as 1. Therefore multiplication by a large number is necessary. To improve this problem, the image colors can be inverted, as another preprocessing step.

5.3.3 Network architectures

Several different architectures were tested to determine which one would be best fit for the task. This section first describes the base architecture and then the variations that were created from it.

Initial design

The initial design for the neural network uses an architecture inspired by Unet[3]. Unet was used to perform image segmentation on a data set of images of biological cells via an encoder-decoder architecture. The computational performance available on the AADC car (see section 3.1) is limited. This was the reason to use a smaller architecture than SegNet Badrinarayanan et al. [2017], which would be the go-to choice for semantic segmentation in the automobile environment. As it only needs to distinguish the classes of 'road' and 'no road', the same way that Unet performs a binary segmentation, Unet seems to be a good first approach for the group's problem.
Figure 5.8 shows the architecture of the initially used network. It takes an RGB-image with 3 channels of 8-bit color information as input, and outputs a greyscale 8-bit image of the same size. For each pixel the output represents the probability of being drivable. Therefore, the network technically computes a saliency map, not a segmentation. For distinguishing between road and not road, the additional step of applying an argmax to convert it to a binary segmentation map seemed unnecessary, since the results were close to binary already. When adding the task of segmenting signs as an additional class (see section 5.4), an argmax operation was added before passing on the result to the control architecture.

[3]`https://github.com/zhixuhao/unet`

The network can be split into two parts, where the first part is the downsampling part, while the second is the upsampling part. The downsampling part consists of convolution and pooling operations. The convolutions increase the number of features at each pixel, starting at 32, then doubling with each convolution, until there are 512 features per pixel. The max-pooling operation meanwhile reduces the number of pixels by only propagating the maximum values per feature in a 2×2 window, thereby halving width and height with every max-pooling step. The upsampling part of the network consists of two operations. The first kind are convolutions over the image features at the current step, merged with the features at the symmetric downsampling step. This again decreases the number of features per pixel, and incorporates the therein contained information into the now lower-dimensional representation. The second kind are upconvolutions (not transpose convolutions!). These are simply convolutions applied after a simple 2×2 upsampling operation, which only duplicates one pixel value into four. With these upsampling operations the resolution gets increased. Combining the merged-convolutions and upconvolutions yields a one-dimensional image of the original image size, containing the segmentation information.

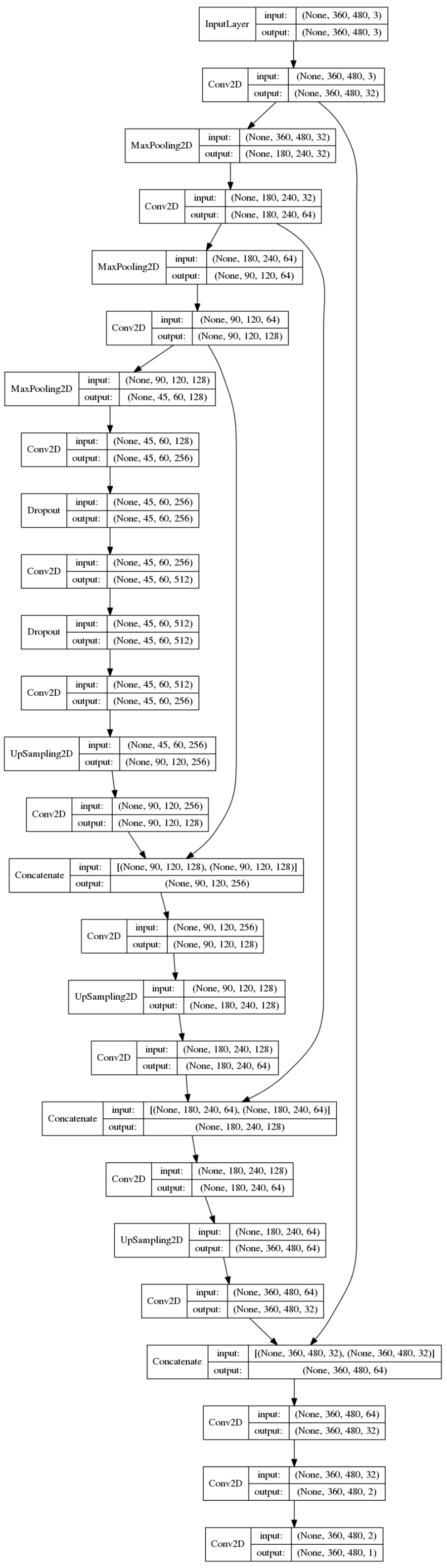

InputLayer input: (None, 360, 480, 3) output: (None, 360, 480, 3)
Conv2D input: (None, 360, 480, 3) output: (None, 360, 480, 32)
MaxPooling2D input: (None, 360, 480, 32) output: (None, 180, 240, 32)
Conv2D input: (None, 180, 240, 32) output: (None, 180, 240, 64)
MaxPooling2D input: (None, 180, 240, 64) output: (None, 90, 120, 64)
Conv2D input: (None, 90, 120, 64) output: (None, 90, 120, 128)
MaxPooling2D input: (None, 90, 120, 128) output: (None, 45, 60, 128)
Conv2D input: (None, 45, 60, 128) output: (None, 45, 60, 256)
Dropout input: (None, 45, 60, 256) output: (None, 45, 60, 256)
Conv2D input: (None, 45, 60, 256) output: (None, 45, 60, 512)
Dropout input: (None, 45, 60, 512) output: (None, 45, 60, 512)
Conv2D input: (None, 45, 60, 512) output: (None, 45, 60, 256)
UpSampling2D input: (None, 45, 60, 256) output: (None, 90, 120, 256)
Conv2D input: (None, 90, 120, 256) output: (None, 90, 120, 128)
Concatenate input: [(None, 90, 120, 128), (None, 90, 120, 128)] output: (None, 90, 120, 256)
Conv2D input: (None, 90, 120, 256) output: (None, 90, 120, 128)
UpSampling2D input: (None, 90, 120, 128) output: (None, 180, 240, 128)
Conv2D input: (None, 180, 240, 128) output: (None, 180, 240, 64)
Concatenate input: [(None, 180, 240, 64), (None, 180, 240, 64)] output: (None, 180, 240, 128)
Conv2D input: (None, 180, 240, 128) output: (None, 180, 240, 64)
UpSampling2D input: (None, 180, 240, 64) output: (None, 360, 480, 64)
Conv2D input: (None, 360, 480, 64) output: (None, 360, 480, 32)
Concatenate input: [(None, 360, 480, 32), (None, 360, 480, 32)] output: (None, 360, 480, 64)
Conv2D input: (None, 360, 480, 64) output: (None, 360, 480, 32)
Conv2D input: (None, 360, 480, 32) output: (None, 360, 480, 2)
Conv2D input: (None, 360, 480, 2) output: (None, 360, 480, 1)

Figure 5.8: Network architecture `regular`

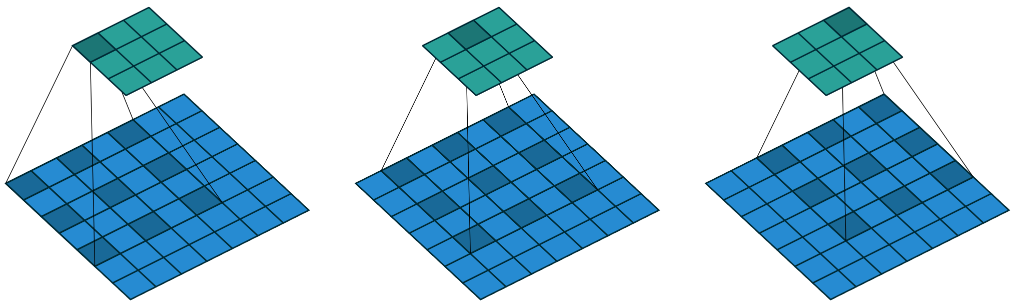


Figure 5.9: Dilated convolution with $d = 2$. The 3×3 kernel is spread out over a 5×5 receptive field. The shown dimensionality reduction is not a product of the dilation but of the missing padding Dumoulin and Visin [2016].

Architecture variations

To test the effect of different configurations, the following variations from the original architecture were designed. In experiments, the original architecture will be referred to as `regular`.

A lot of the image of a road will be a rather monotonous grey or black shade. Therefore all the local image features generated by the convolutional layers should be the same. Distinguishing black areas that belong to the road from those that do not seems difficult based on such local features. A wider receptive field for the convolutions would be needed. Therefore dilated convolutions were introduced at the end of the downsampling process. They take the same number of parameters as regular convolutions, but instead of working on a dense 3×3 window, they move d pixels from the center pixel to determine the other pixels to compute the convolution over. A dilated convolution with $d = 1$ corresponds to a regular convolution, while one with $d = 2$ regards the center and eight outer pixels of a 5×5 window, leaving one pixel horizontally and vertically between each included pixel. Figure 5.9 visualizes this idea. As figure 5.14 shows, the dilated convolutions did not necessarily improve the performance much but allowed for a reduction in the architecture's depth, while still keeping performance at the same level.

Three different architectures were created, testing varying dilation rates. Architecture `dil2` changes the last downsampling convolution to a dilated convolution with $d = 2$. For `dil4` this gets changed to $d = 4$, while the previous layer has $d = 2$. With `dil8` $d = 8$ is used in the last downsampling convolution, 2 and 4 in the respective previous layers. To further reduce the size of the network, and thereby increase the computation speed, also an architecture with fewer layers was designed. At this point of the experiments it could not have been tested the inference speed on the car architecture, therefore this was a preventative step. The hardware resources have been limited on the car, therefore speeding up the inference is desirable. By leaving out the three middle layers, the convolutions produce a maximum depth of 128 features. The resulting architecture is referred to as `flat`. Adding the idea of dilated convolutions in a similar way results in the architectures `flat_dil2` and `flat_dil8`, which use a maximum of 256 features. Since the Unet architecture uses upconvolutions, instead of transpose convolutions (sometimes referred to as 'deconvolutions'), also an architecture named `deconv` was tested. It arises from the `regular` architecture by replacing the upconvolution with transpose evolution operations.

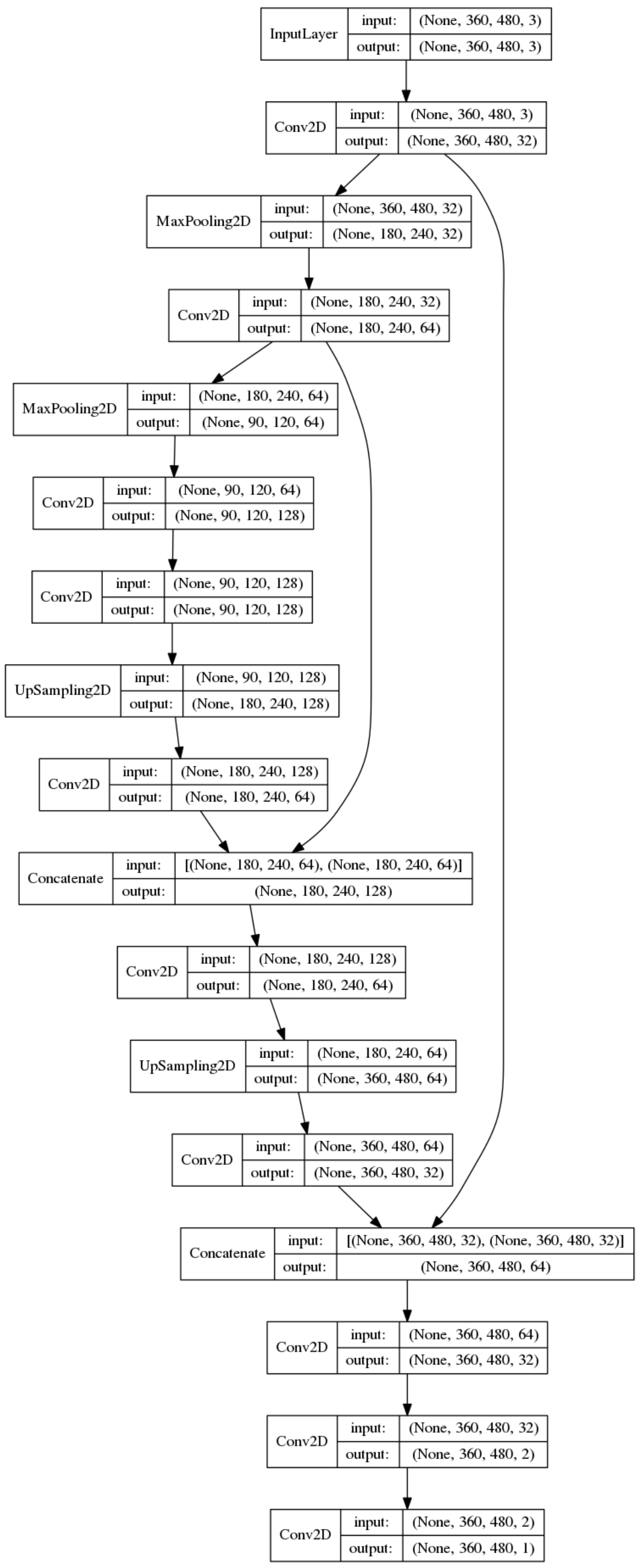

InputLayer input: (None, 360, 480, 3) output: (None, 360, 480, 3)
Conv2D input: (None, 360, 480, 3) output: (None, 360, 480, 32)
MaxPooling2D input: (None, 360, 480, 32) output: (None, 180, 240, 32)
Conv2D input: (None, 180, 240, 32) output: (None, 180, 240, 64)
MaxPooling2D input: (None, 180, 240, 64) output: (None, 90, 120, 64)
Conv2D input: (None, 90, 120, 64) output: (None, 90, 120, 128)
Conv2D input: (None, 90, 120, 128) output: (None, 90, 120, 128)
UpSampling2D input: (None, 90, 120, 128) output: (None, 180, 240, 128)
Conv2D input: (None, 180, 240, 128) output: (None, 180, 240, 64)
Concatenate input: [(None, 180, 240, 64), (None, 180, 240, 64)] output: (None, 180, 240, 128)
Conv2D input: (None, 180, 240, 128) output: (None, 180, 240, 64)
UpSampling2D input: (None, 180, 240, 64) output: (None, 360, 480, 64)
Conv2D input: (None, 360, 480, 64) output: (None, 360, 480, 32)
Concatenate input: [(None, 360, 480, 32), (None, 360, 480, 32)] output: (None, 360, 480, 64)
Conv2D input: (None, 360, 480, 64) output: (None, 360, 480, 32)
Conv2D input: (None, 360, 480, 32) output: (None, 360, 480, 2)
Conv2D input: (None, 360, 480, 2) output: (None, 360, 480, 1)

Figure 5.10: Network architecture `flat`

Finally, using the gained knowledge from the different experiments, an architecture was designed that deviates more strongly from the original Unet, called `droschkinator`.

The overall architecture of this network mostly follows that of `flat_dil8`, using two down- and upsampling steps respectively and dilation factors of 2, 4 and 8 in the fourth to sixth convolutional layer. It differs from the previous architechtues in that it uses strided convolutions, to combine down-sampling and feature extraction into a single layer. Strided convolutions work similarly to regular convolutions, though instead of moving the kernel over every single input pixel, it uses only every s pixel, s being the stride parameter. A stride of $s = 2$ moves the kernel by two pixels, thus reducing the output size by 50%, achieving the same downsampling result as the pooling layers before. It also uses transposed convolutions for upsampling, which in previous experiments did not have any measurable impact on the quality of the results but helped in reducing the size of the architectures.

Additionally batch normalization layers were added after each convolution. These normalize the activations of the previous layer in the training batch to have a mean of $\mu = 0$ and a standard deviation of $\sigma = 1$, which improves the convergence and overall stability of the training process. This was done due to the extreme variance between the results of different training runs visible in figure 5.14 and, as figure 5.16 shows, helped in alleviating this issue.

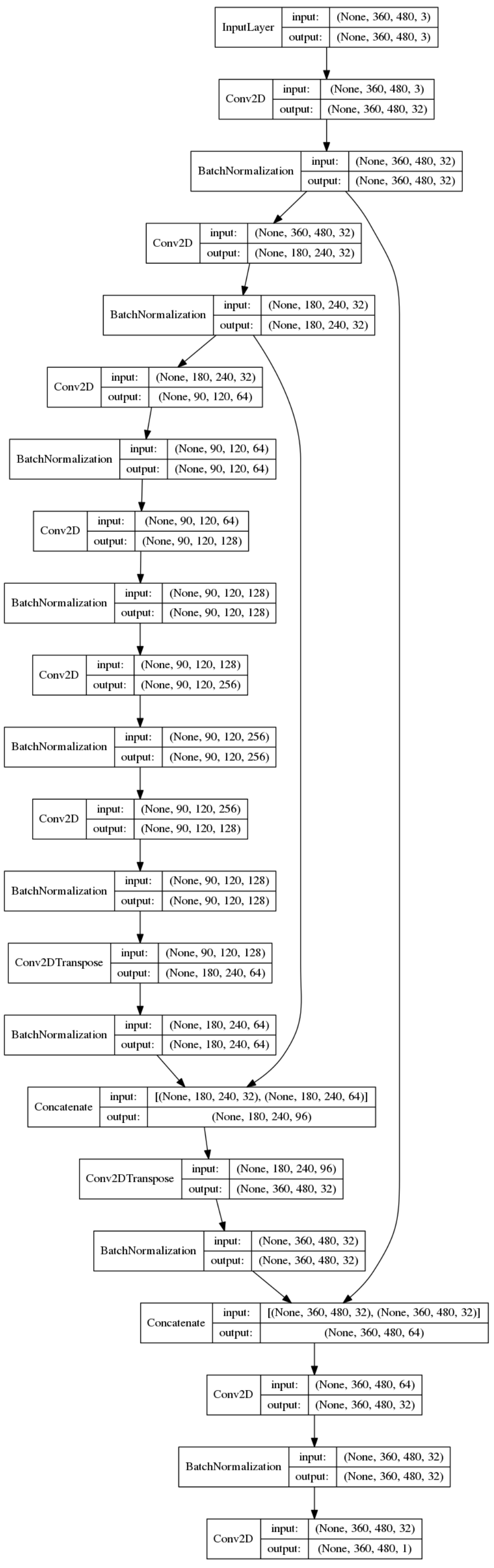

InputLayer | input: (None, 360, 480, 3) | output: (None, 360, 480, 3)
Conv2D | input: (None, 360, 480, 3) | output: (None, 360, 480, 32)
BatchNormalization | input: (None, 360, 480, 32) | output: (None, 360, 480, 32)
Conv2D | input: (None, 360, 480, 32) | output: (None, 180, 240, 32)
BatchNormalization | input: (None, 180, 240, 32) | output: (None, 180, 240, 32)
Conv2D | input: (None, 180, 240, 32) | output: (None, 90, 120, 64)
BatchNormalization | input: (None, 90, 120, 64) | output: (None, 90, 120, 64)
Conv2D | input: (None, 90, 120, 64) | output: (None, 90, 120, 128)
BatchNormalization | input: (None, 90, 120, 128) | output: (None, 90, 120, 128)
Conv2D | input: (None, 90, 120, 128) | output: (None, 90, 120, 256)
BatchNormalization | input: (None, 90, 120, 256) | output: (None, 90, 120, 256)
Conv2D | input: (None, 90, 120, 256) | output: (None, 90, 120, 128)
BatchNormalization | input: (None, 90, 120, 128) | output: (None, 90, 120, 128)
Conv2DTranspose | input: (None, 90, 120, 128) | output: (None, 180, 240, 64)
BatchNormalization | input: (None, 180, 240, 64) | output: (None, 180, 240, 64)
Concatenate | input: [(None, 180, 240, 32), (None, 180, 240, 64)] | output: (None, 180, 240, 96)
Conv2DTranspose | input: (None, 180, 240, 96) | output: (None, 360, 480, 32)
BatchNormalization | input: (None, 360, 480, 32) | output: (None, 360, 480, 32)
Concatenate | input: [(None, 360, 480, 32), (None, 360, 480, 32)] | output: (None, 360, 480, 64)
Conv2D | input: (None, 360, 480, 64) | output: (None, 360, 480, 32)
BatchNormalization | input: (None, 360, 480, 32) | output: (None, 360, 480, 32)
Conv2D | input: (None, 360, 480, 32) | output: (None, 360, 480, 1)

Figure 5.11: Network architecture `droschkinator`

5.3.4 Experiments and results

This section outlines the experiments and analysis of the results to determine how to best segment the drivable area using neural networks.

All experiments in this section are run using a 90/10 split between training and validation data. A test set of roughly 10% of the original size of the data set has been split off by random selection before any training began, so the training/validation split is only performed on the non-test data.

The performance of the models is measured by $IOU = 1 - IOU_{loss}$, where IOU_{loss} is the intersection over union loss, as described by Rahman and Wang Rahman and Wang [2016]. Note that this definition of IOU is different from the classical definition. This variant does not require a hard classification decision, but can also evaluate probabilities per class. Since it is defined via the negative IOU_{loss}, higher values indicate better performance.

Therefore, the metric is computed as follows:

$$IOU = \frac{\sum_{v \in V} X_v \times Y_v}{\sum_{v \in V}(X_v + Y_v - X_v \times Y_v)},$$

where X, Y are the prediction and ground truth respectively, X_v, Y_v indicate prediction or ground truth for a single pixel and $v \in V$ iterates over all pixels of an image.

The models are trained using the Adam solver with a learning rate of $1e-4$, binary crossentropy as loss function and a batch size of 8. Before splitting the data into training and validation data, it is shuffled. After each epoch of training, the training, validation and test loss and IOU are logged.

Comparing architectures

As an initial experiment, it needs to be determined whether the network architectures are able to recognize the road on the road panels at all. Therefore, a small subset of `rl_straight` consisting of 64 images was manually annotated. To increase the number of training images, these images and corresponding labels were then augmented (see the first paragraph of section 5.3.1).

Figure 5.12 illustrates the development of the IOU on the validation set, as measured over 100 epochs of training, for the different architectures. It can be seen that after about 15 epochs, most of the models already achieve very good performance, indicating that the architectures are well-suited to solve the problem. Also visible is, that a reduction in architecture depth does decrease its performance but, using dilated convolutions, this effect can be negated, with the `dil8` architecture even slightly surpassing the performance of the `regular` architecture.

However, two graphs show unexpected behavior. Architecture `dil2` stays far below the rest of the other architectures. The performance of `flat_dil2` stays on the same level as that of `dil2` for the first few epochs, but between epoch 10 and 15 it drastically improves.

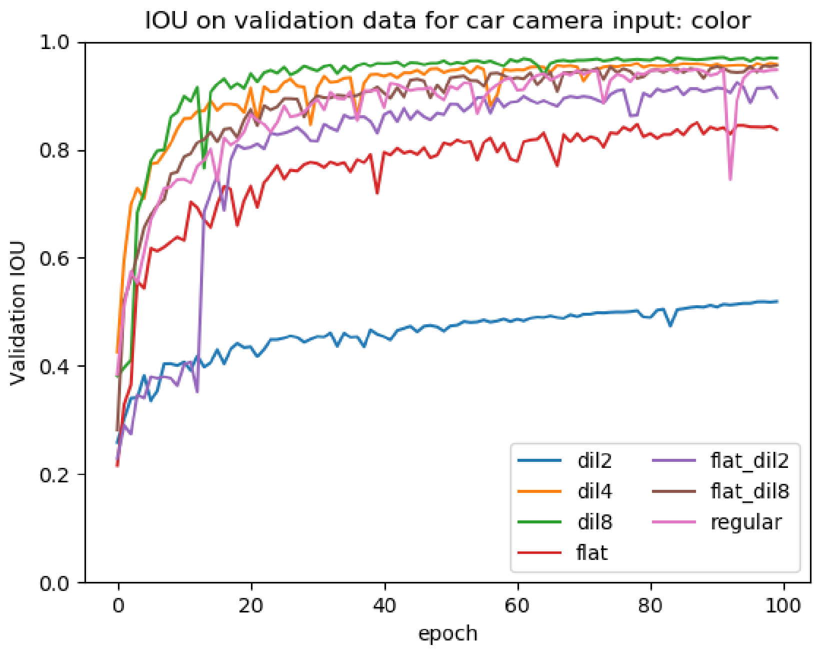
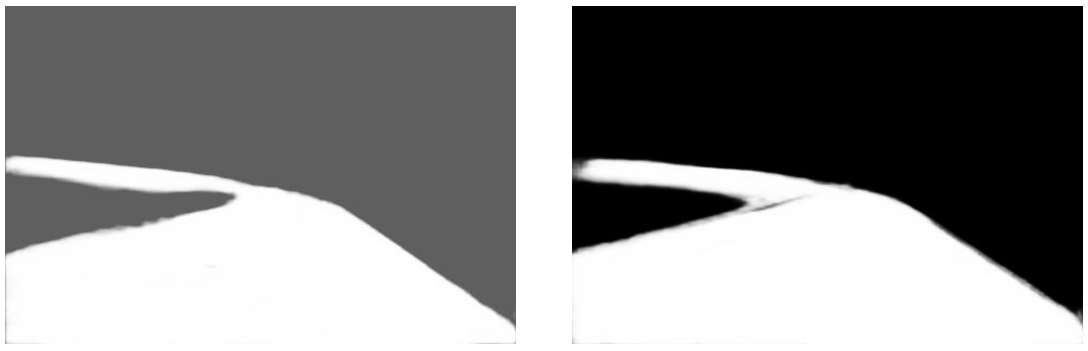


Figure 5.12: Development of the IOU measure on the validation set for different architectures. Higher values represent better performance. The training was performed for 100 epochs on a data set of 229 images, an early subset of `rl_straight`.

Looking at the other graphs, a similarly strong movement can be seen during the first few epochs.

Since `dil2` is not very different from either `regular`, or `dil4`, its singular behavior is highly conspicuous. Looking at the images generated by the final model, a big difference between it and the other models becomes apparent. While it is able to correctly determine where the road is, it does not label the background as black (not drivable), but in a grey tone (see Figure 5.13). Considering the IOU measure, this leads to a small error at every background pixel, summing to a large error for every image. Looking back at the big jump in performance for `flat_dil2`, it might be assumed that this jump was caused by switching from grey background to black background, since this seems to be what separates `dil2` from the other architectures.

Because the architecture of `dil2` is not very different from the other ones, it was assumed that the difference in performance is not an architectural problem, but one of initialization. Therefore, in the following experiments, each training run was repeated five times with different random seeds, to be able to also judge the variance introduced by initialization. Additionally, the IOU measure was evaluated on the test set, instead of the validation set. However, both values turned out to develop qualitatively in the same way.



(a) Inference using `dil2` architecture (b) Inference using `dil8` architecture

Figure 5.13: Images annotated by the neural network after training on the reduced `rl_straight` data set. The left image shows the problem of uncertain classification of the background area, thereby accruing a large error over the whole image, even though the shape of the segmentation is actually very good.

Comparing preprocessing steps

To find the best way to segment road images without intersections, each of the previously outlined preprocessing steps was combined with each of the architectures, repeating each of those training runs five times to account for the assumed initialization variance. The results are shown in Figure 5.14. The main results are that there is a strong variance in the results of most of the configurations. The `flat` architecture seems stable for most input configurations. However, for color input, it varies just as strongly as the others. Additionally, other architectures have higher maximum performances, even if they vary more. Furthermore, it became evident that greyscale images lead to significantly worse performance than the ones containing color information. The different versions of images containing color (no preprocessing, inverted, half size, inverted + half size) do not seem to show any significant differences. Therefore, further experiments focused on regular color images.
Building on the experience from these experiments, and ideas gained from further research, the `droschkinator` architecture was added. In the results of another test run on the same data in Figure 5.15, the new architecture is much more stable than the other architectures, while still delivering a high performance.
Since the performance of the neural networks is already very good, although only 64 annotated images were used, the problem of segmenting images of a curved road without intersections was considered solved at this point.

Understanding intersections

After being able to recognize roads without intersections, different kinds of intersections should now be added to determine whether a new architecture is needed to solve this segmentation problem. The `rl_crossings` data set was used and the existing architectures were tested. The results in Figure 5.16 show that the existing architectures already perform

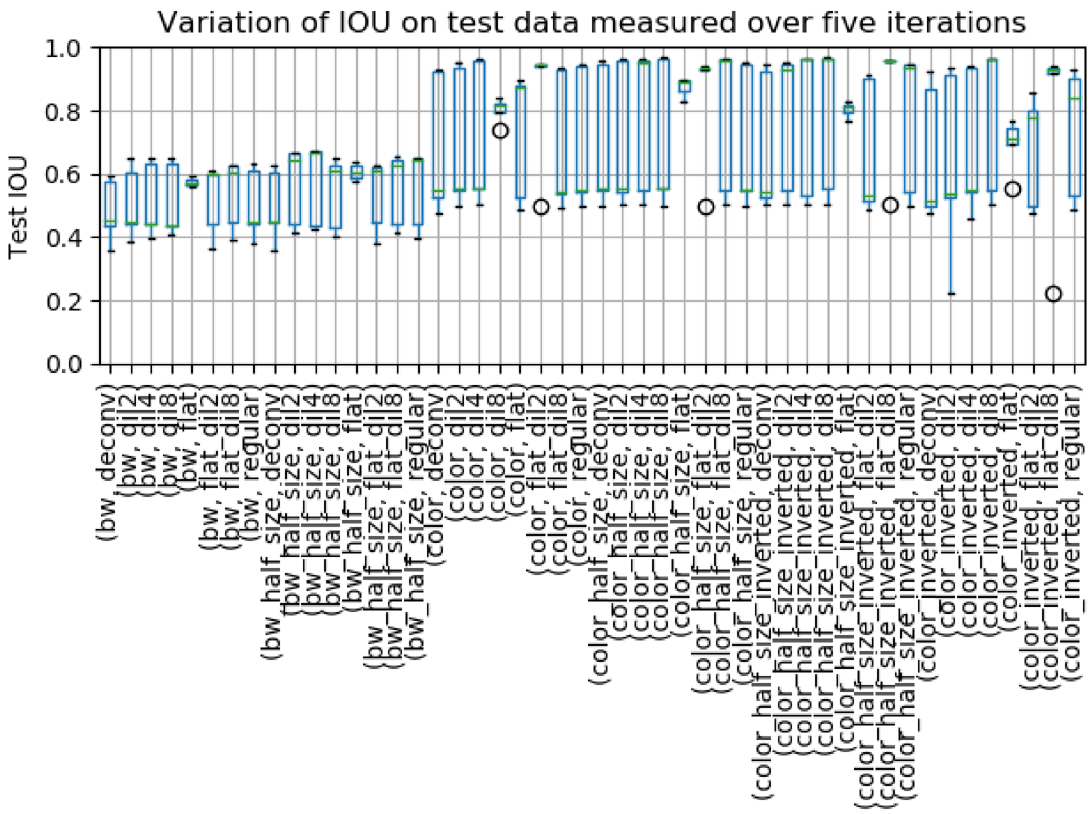


Figure 5.14: Performance of different combinations of preprocessing and architectures, as measured on the test set, after 100 epochs of training. Each training run was repeated five times with differing random seeds to account for variance introduced by initialization. Higher values represent better performance. The training was performed for 100 epochs on a data set of 229 images, an early subset of `rl_straight`. BW represents the transformation to greyscale images (Black and White).

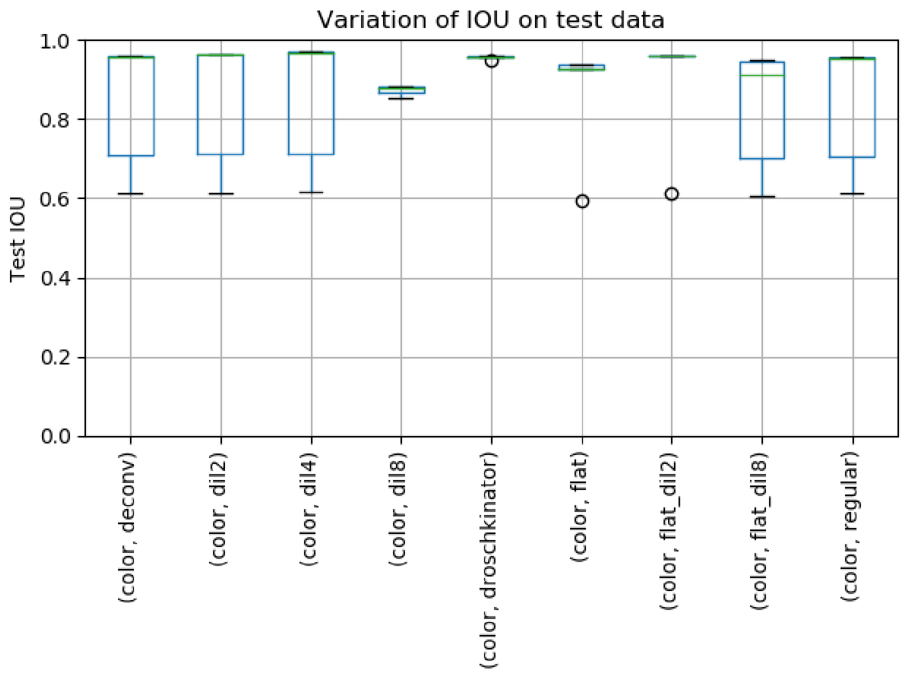


Figure 5.15: Performance of different architectures, as measured on the test set, after 100 epochs of training. Each training run was repeated five times with differing random seeds to account for variance introduced by initialization. Higher values represent better performance. The training was performed for 100 epochs on a data set of 229 images, an early subset of `rl_straight`. This experiment added the new `droschkinator` architecture.

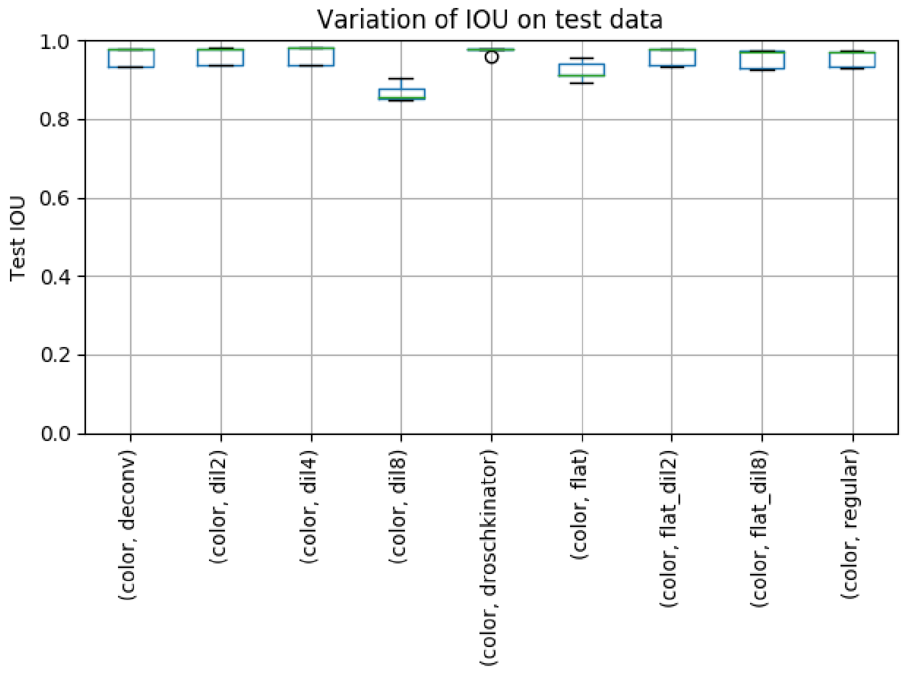
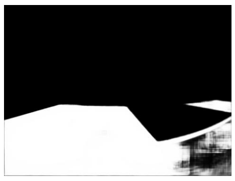


Figure 5.16: Performance of different architectures, as measured on the test set, after 100 epochs of training. Each training run was repeated five times with differing random seeds to account for variance introduced by initialization. Higher values represent better performance. The training was performed for 100 epochs on the `rl_crossings` data set.


(a) Noisy segmentation

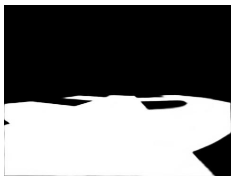

(b) Consistent segmentation

Figure 5.17: Exemplary test images, infered by the neural network after training on `rl_crossings`. The overall segmentation works extremely well, though some jitter-errors exist.

very well on the new data set. The reduced variance in results can probably be assigned to the significantly increased amount of training data. Figure 5.17 shows the surprisingly good predictions. While there are some errors, those should not pose a problem when feeding the segmentation into the control logic.
Since this experiment did not contain any long curvy routes, as in `rl_straight`, now the two data sets were combined to `rl_combined`. If the resulting network performs well, it can be assumed that it will perform well on the road panels in real life (given the same lighting conditions etc. as in the training data). The results are shown in 5.18.

Recognizing roads in VTD

To test different configurations of the control logic, controlling the movement of the car, it was desirable to be able to drive within the VTD simulation. Therefore, the network needs to be able to segment roads in VTD, too. The used data set was `vtd_crossings`.
The results shown in figure 5.19 indicate that all of the tested architectures perform sufficiently on this data set. The `flat` architecture performs worse than the rest, while the `droschkinator` architecture shows almost no variation in the IOU test set.
Based hereon, it was assumed that the tested architectures are able to solve the road segmentation task.

Learning transferable knowledge

An interesting question at the point of merging the simulated and real world applications is whether it is possible to ascertain knowledge that is applicable to both worlds. I.e.: Can one neural network be used to drive well enough in both the simulated, as well as the real world?
To test this, `vtd_crossings` and `rl_combined` were combined. Since a lot of the architectures seemed to perform similarly well in previous experiments, some of them were removed from the comparison.
Furthermore, the `EarlyStopping-Callback` was activated in Keras. Since past experiments seemed to converge to a good solution quickly, and not deteriorate in performance after a lot of further training iterations, this should speed up experiments. From here on out the EarlyStopping mechanism stops the training if the validation loss has not im-

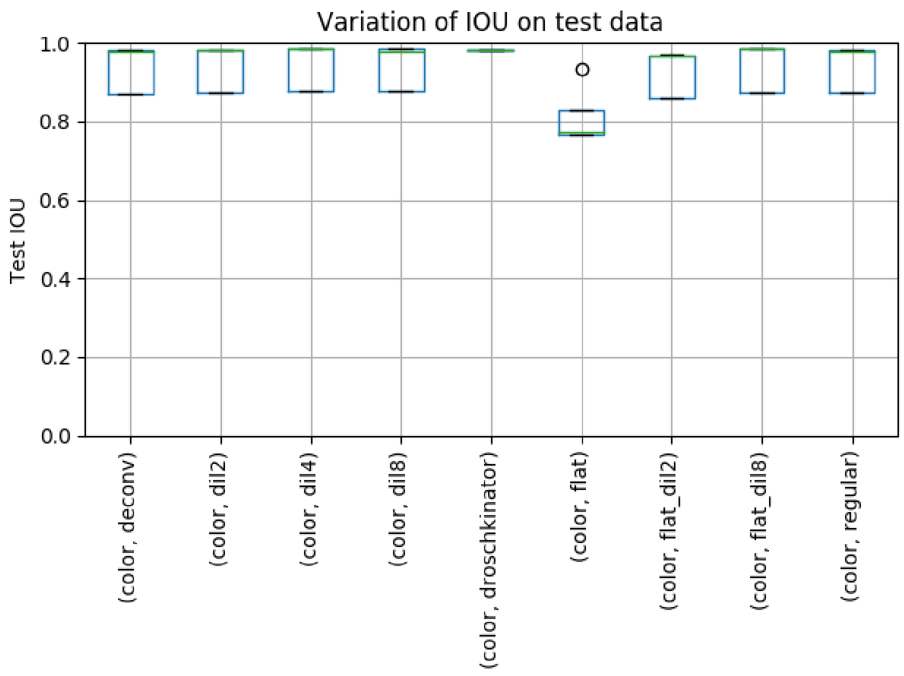


Figure 5.18: Performance of different architectures, as measured on the test set, after 100 epochs of training. Each training run was repeated five times with differing random seeds to account for variance introduced by initialization. Higher values represent better performance. The training was performed for 100 epochs on the `rl_combined` data set. High values indicate that both curvy roads and crossings can be recognized well.

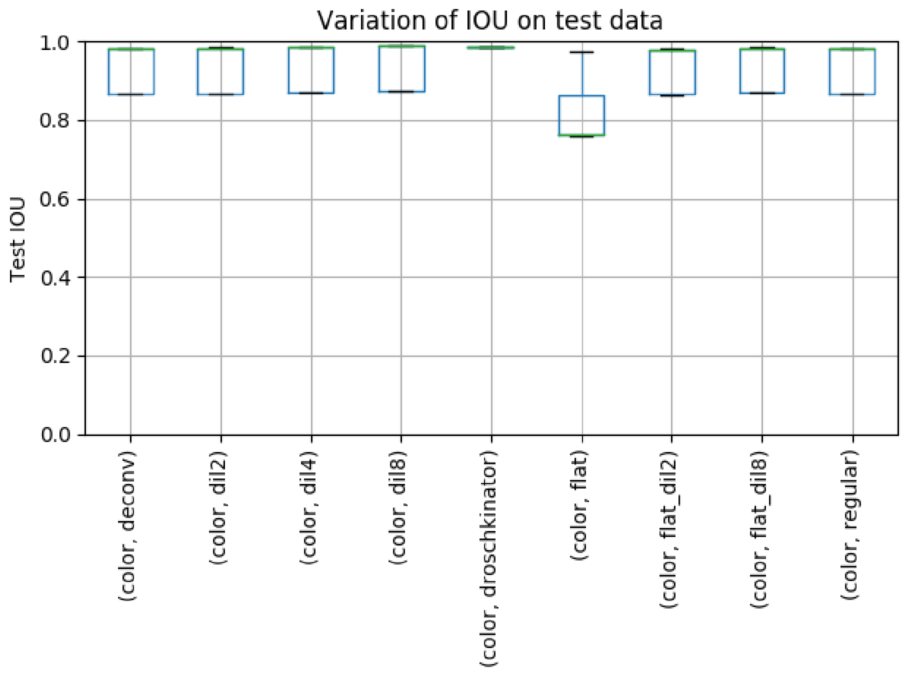


Figure 5.19: Performance of different architectures, as measured on the test set, after 100 epochs of training. Each training run was repeated five times with differing random seeds to account for variance introduced by initialization. Higher values represent better performance. The training was performed for 100 epochs on the `vtd_crossings` data set. High values indicate that both curvy roads and crossings can be recognized well.

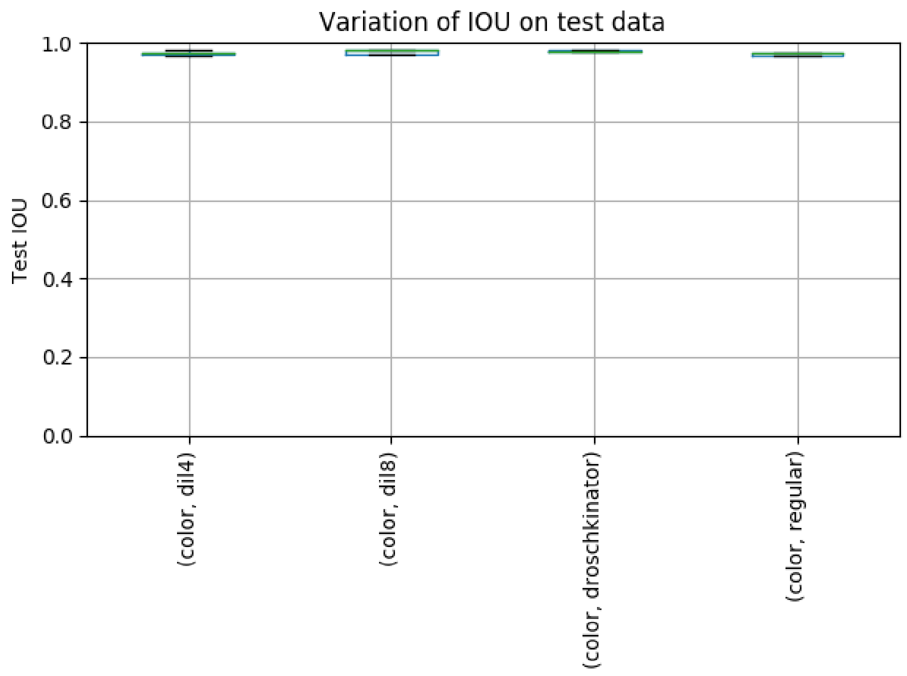


Figure 5.20: Performance of different architectures, as measured on the test set, after a maximum of 100 epochs of training. Each training run was repeated five times with differing random seeds to account for variance introduced by initialization. Higher values represent better performance. The training was performed on the combined `vtd_crossings` and `rl_combined` data set. High performance indicates that the neural network is able to recognize streets both in the real world, and in the simulation.

proved over the last three epochs of training. The results in Figure 5.20 indicate that the network is able to generalize well enough to recognize roads in both simulated and real world. While the test case is of course limited, these conclusions can be used as the starting point for further research.

Enriching real life training with simulated scenarios

Another important question in context to the challenges posed by the real-world application of autonomous driving methods is whether simulated data can be used to improve real-world driving capabilities. Creating annotated real-world data is very costly and time consuming. Therefore, it would be desirable to create a wide range of cheap simulated data and add it to a lower amount of costly real-world data to then create a strong model for driving in the real world.

To test this, several data sets were created, varying the proportions of VTD and real life data. After initial experiments that tested 0, 25, 50, 75, and 100% of VTD data respectively, the two consistently outperformed architectures `regular` and `dil4`, were dropped to explore the space between 75 and 100% in 5% steps.

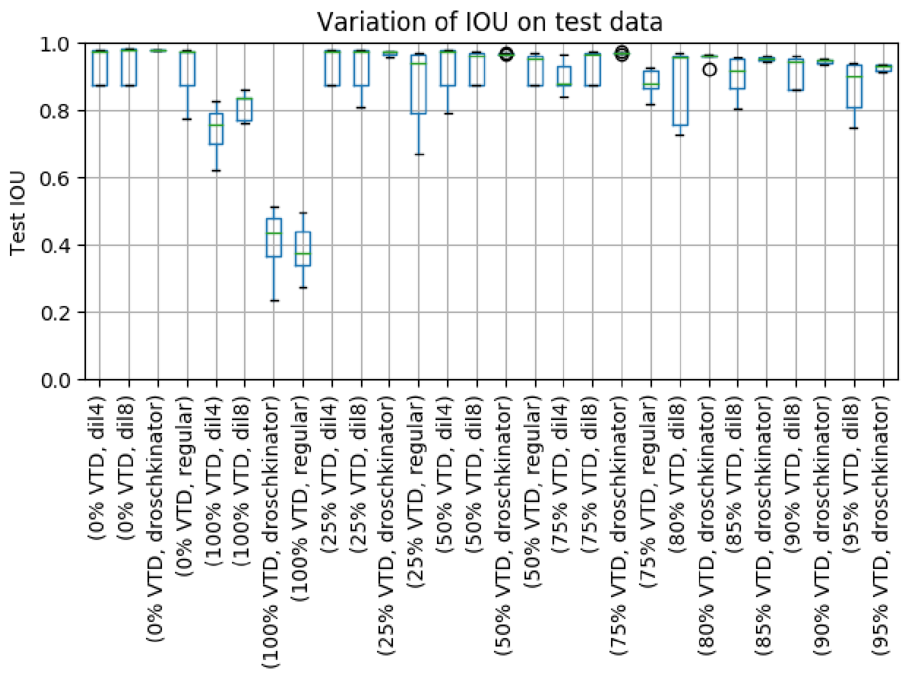


Figure 5.21: Performance of different architectures, as measured on the test set, after a maximum of 100 epochs of training. Each training run was repeated five times with differing random seeds to account for variance introduced by initialization. Higher values represent better performance. The training was performed on different proportions of VTD and real life data. Higher values indicate higher performance on segmenting real life street pictures.

The real life data used was the `rl_combined` data set. Since this data set was the limiting factor, it was decided to always use a total of 977 images, no matter the percentage chosen. This is the total number of images in the training set of `rl_combined`, and therefore the maximum number of real life images that can be used. Fixing the number of images allows to exclude any influence that the size of the training set might have. Since the influence on predictions in the real world should be judged, the test set consists solely of the test data from `rl_combined`.

The VTD data was supposed to reflect a variety of different circumstances to enrich the training set with a lot of variation in the data and allow for robust features to be learned. Therefore three different test routes were used in VTD, and images from each of them for the weather settings `blue_sky`, `cloudy`, and `rainy` were extracted. The routes used are `vtd_rural`, `vtd_town`, and `vtd_evaluation` (see section 5.3.1). An equal numbers of images from each of the data sets and weather conditions was chosen, based on the given percentage of VTD data. Any remaining image slots to reach the total number of 977 images are filled up with images from `rl_combined`.

To increase the computation speed, and because previous experiments had not shown any drop in performance for this, all images were downscaled to a resolution of 240x160.

The results of the experiments are shown in Figure 5.21. As expected, the worst results are achieved when the model gets tested on real life data without ever seeing any during training (100% VTD). The performance, as measured via the IOU, slowly declines when increasing the proportion of VTD data. Interestingly, the IOU on the test set is still above 0.9, even for 95% VTD data indicating a solid recognition performance. It has to be kept in mind that this happens for a total number of less than 50 real life images, since it was kept the total number of images fixed. Therefore a decline of performance is to be expected. Furthermore, experiments were run in which images were augmented, increasing the size of the training data set eightfold. However, the results were qualitatively the same, so this won't be reported here.

Based on these results, it can be assumed that simulation data can be effectively used to increase the size of and variation within the training data set. A clear limitation of the experiment is the limited variation within the test set. Further experiments should be repeated with a wider variety within the test set to better judge the quality of the resulting models. Additionally, a larger total number of images should be used, so that using 95% of VTD images still leaves a reasonable number of real life images for the training. Augmentation might help with this, but in this case did not improve performance.

5.3.5 Performing inference directly on the car

This section outlines the technical details of how inference was performed on the car.

The car should be able to recognize the ongoing street and detect the drivable area to make proper control decisions. To achieve this, the model needs to be fed with live recordings and to obtain a 2D-Matrix, which indicates the drivable area as fast as possible.

Like Tensorflow was used to define and train the neural network it also has been used for making the live predictions, but without using Keras. Tensorflow is a well suited framework, because it is well documented, completely free, broad utilizable and there are many state of the art examples available on github which have overlapping objects. One special reason to use Tensorflow for this project was the requirement to work with C++ and the possibility to define and train the neural network in python. Another advantage is the well integrated GPU-utilization.

The team discussed many implementation methods for the inference with Tensorflow, because it was not clear how it is possible to utilize the already trained segmentation model. As a first approach, there was an idea to use a python environment within the C++ environment. It is possible to execute python code within C++ but it is not recommended to do so, if the python code needs many libraries. This approach was discarded, because a stable and fast solution for the problem was needed, which is scalable to include post processing in further development states.

Another consideration was to call a Python server within the C++-environment and transfer the input/output data over TCP. This solution was judged as not optimal, because it would always get long response time due connection and conversion overhead.

Tensorflow itself brings an approach to deploy models for productive environments, which is called Tensorflow Serving. The Introduction from the official page is: Ten [c]

TensorFlow Serving is a flexible, high-performance serving system for machine learning models, designed for production environments. TensorFlow Serving makes it easy to deploy new algorithms and experiments, while keeping the same server architecture and APIs. TensorFlow Serving provides out-of-the-box integration with TensorFlow models, but can be easily extended to serve other types of models and data.
This approach was also discarded because the effort appeared too high for integration and understanding the relevant features, and Tensorflow Serving provides a great amount of functionality.
Finally it was decided to do the prediction routine completely in C++ with the provided C-API Ten [a]. By using the C-API, the possibility was given to use C++ code, which would be the native solution in the ADTF environment. Another advantage is the flexibility, which is necessary for the post processing implementation and the expected speed for executing the inference routine.
Because Docker is used for the complete car setup it's obvious to use and integrate an already available dockerfile provided by Tensorflow.
Dockerfile Setup: (sources: Tensorflow Docker: Ten [b], NVIDIA Docker: NVI):

- Tensorflow image: development version with GPU support
  - Ubuntu 16.04
  - Bazel 0.15.0
  - Tensorflow 1.10
- NVIDIA image
  - CUDA 9.0
  - cuDNN 7.2.1.38

It is necessary to use the NVIDIA-Dockerfile as well, because it contains the relevant packages required for Tensorflow. All dependencies will be integrated into one dockerfile. Since the main dockerfile runs with Ubuntu 18.04, it is required to modify the dockerfile for working with the actual Version. Many commands could not be used with Ubuntu 18.04 because it was not available at this moment at the classical repositories, therefore it was necessary to find suitable replacements.
Because the official dockerfiles were prepared for using a Python environment, but C++ was required for Tensorflow yet another modification was required. It was also necessary to integrate the C-API into the dockerfile. Further it turned out to be desirable to be able to use Tensorflow with or without GPU support, which made it necessary to create one image with GPU functionality and another with only CPU usage as different dependencies

were required. In the beginning, the non-development version was chosen, which installs the Tensorflow package via the Ubuntu package management. But it turns out that only the development version could use the complete functionality which is necessary for the C++ environment.

To use a neural network trained within the C++ environment, it first has to be converted. Since the definition and training is done within the Keras library, the trained model was a HDF5-file, which does not work with the C-API. It was necessary to convert this model into a format which could be used in an environment without Keras. Therefore the script provided by Amir Abdi[4] was used to convert the saved Keras model, including graph definition and weights, into a single Protobuf file. This file could then be loaded in the C++ environment.

5.4 Sign Segmentation

In order for the car to find its destination, it needs some kind of navigation system. In the beginning of the project the decision was made not to create a world map that saves the position of the car, as synchronizing the real world with the map and the cars position on the map would in itself have been a difficult problem. Instead, it was decided to only use local navigation based on street signs indicating where to turn and when to stop. With a stop sign, turn right sign and turn left sign, that results in the car needing to be able to distinguish three different street signs. This problem was divided into two parts: segmentation and classification. For the segmentation task the team's previous segmentation network had to be adapted to not only segment roads but also street signs. The segmented street sign would then be given to a simple classification network, described in section 5.6.1, which distinguishes between the three types of signs plus an additional `No Sign` class, in case something other than an actual sign was segmented. Between the segmentation and classification it is also necessary to remove possible noise from the segmentation map and crop the closest street sign to use as input for the classifier. This step is described in section 5.5.

As it proved to be the best architecture for the previous street segmentation task, we chose to use the `droschkinator` architecture to adapt to the new sign segmentation task. This architecture used a *Sigmoid* activation in the last layer, where a 1 corresponds to the street class and a 0 to the background class. Adding a third class meant we had to change this activation function to now be a *Softmax* activation. Given a prediction vector y of size J the *Softmax* function is defined as

$$\text{Softmax}(y_i) = \frac{\exp(y_i)}{\sum_{j=1}^{J} \exp(y_j)}$$

[4]https://github.com/amir-abdi/keras_to_tensorflow

and produces a probability vector containing a probability for each class. As this is done per pixel, this results in a $h \times w \times c$ matrix, h and w being the image's width and height and c being the number of classes, in this case 3. As a result of this, the labels also had to be adapted to this matrix format, containing a one-hot vector for each pixel.

As all previous datasets do not contain signs and manual labeling takes a lot of time, in addition to capturing new images with signs, the augmentation is also modified as described in 5.4.2, to artificially increase the amount of street signs in the data and produce a better balance between classes.

But even with the added augmentation, achieving acceptable results for the street sign segmentation turned out to be more difficult then initially anticipated. Testing the trained network in the real world scenario revealed two main problems: Firstly, the network relies very heavily on color, thus segmenting almost every red or blue area as a street sign and not correctly segmenting actual street signs if their color is slightly off. Secondly, street signs are only correctly segmented when they are less than a meter away from the camera, which in most cases is too late to react to them while driving. To remedy these issues modifications to both augmentation and network architecture have to be made, described in sections 5.4.2 and 5.4.3 respectively.

5.4.1 Evaluation Metrics and Training

The metrics to evaluate the three class segmentation are now `meanIOU` and `per-class IOU`, referring to the canonical IOU definition this time. This allows for more detailed evaluation of the model performance, which becomes desirable when distinguishing between more than two classes. The metrics are computed in the following way.

$$\texttt{meanIOU} = \text{mean}_{c \in C} \frac{TP_c}{FP_c + FN_c - TP_c},$$

where $c \in C$ iterates over all classes, and the other terms indicate the true positive, false positive and false negative counts for the respective class c.Garcia-Garcia et al. [2017] The `per-class IOU` for a given class is simply the central term, parametrized to the according class. To report the performance on a set of images, instead of a single image, the values computed for each of the single images are averaged. However, this is where a problem occurs that is specific to the use case of this project. If a test image does not contain any pixels that are labeled as `street sign`, then this class does not have any `true positives`, therefore the `per-class IOU` is zero, and the whole `meanIOU` value is skewed towards zero. For this reason, in the aggregation of the `meanIOU`, any contributions by images without street signs to the `per-class IOU` of the street sign class are ignored. The respective `meanIOU` and the `per-class IOU` for the street sign class over all images are only computed over the remaining values.

Training is done using a 50/50 combination of augmented VTD and real life data. For VTD, images from both `vtd_rural` and `vtd_town` with automatically labeled roads and

signs, as described in section 5.3.1, are used. For real life images, the newly created `rl_complete` dataset is used, which consists of manually labeled images from a small testing route, as well as the full route described in 3.7.4. In total the resulting dataset includes 4720 training images and 248 test images.
The models are trained using the `Adam` solver with a learning rate of $1e-4$, `binary crossentropy` as loss function and a batch size of 8. Before splitting the data into training and validation data, it is shuffled. After each epoch of training, training, validation, and test loss, `meanIOU` and `per-class IOUs` are logged. Since Keras applies metrics per batch and averages over the batch-wise metrics, the problem described above appears again. In batches without any street signs on the images, Keras still computes a (meaningless) `per-class IOU` for the street sign class. It then automatically averages those wrong `per-class IOUs` over all batches and reports it as values for the training and validation set. Because of the faulty values included, these measures will generally by skewed towards zero. A custom Callback was written to manually correct this error for the measures on the test set. Adding even more manual computations for the training and validation set, or changing code deep within the Keras libraries for this single use case was deemed too costly.

5.4.2 Street Sign Augmentation

As learning of the street sign class turned out to be more difficult than just streets, especially with the much lower representation of that class in the training data, an additional step was added between the transformation and noise filter steps described in section 5.3.1. In this step up to five images of street signs are randomly placed in each image, each of which is individually randomly rotated between -45 and +45 degrees and sheared between -60 and 60 degrees to simulate three-dimensional rotation. To simulate distance to the signs, they are additionally randomly scaled to be between 10×10 and 85×85 pixels. The labels are also automatically modified to include the added signs. This helps to balance the classes and prevents the network from outright ignoring that class but it does not prevent it from just relying on the color of the signs.
This issue is addressed in two steps. Firstly the network has to be encouraged to learn the signs' structural features rather than their color, so instead of placing the signs in their original color into the images, they are first converted to the HSV color space and their hue value shifted by up to 90 degrees in either positive or negative direction. Additionally the pixel values of the signs are multiplied by a factor between 0.5 and 1.5 and blended with the original sign image using a frequency noise map to simulate light and shadow effects.
Secondly the network has to be discouraged from just segmenting single colored areas or other structures similar to signs. To achieve this, in addition to the signs other image parts are inserted into the images, just without labeling them as signs and in this case only one per image. Some of these were created by us, while others were taken from parts of the

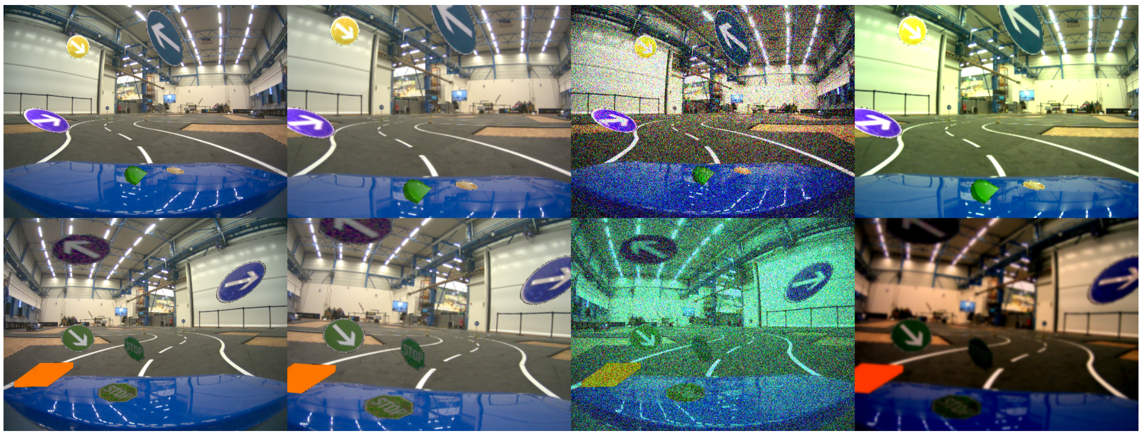


Figure 5.22: Example augmentation of a single image with added signs and the modifications described in section 5.3.1.

scenery, which the network actually segmented as signs. Figure 5.22 shows an example of this modified augmentation.

5.4.3 Finding the Correct Receptive Field

As described in section 5.3.3, one measure to optimize street segmentation, was to increase the receptive field of the neural network, because the local features of parts inside the street are very similar to those outside it. Additionally, as a means to increase computational speed, the resolution of the input images was reduced to 240×160, as described in 5.3.4. In most cases the street covers a large portion of the entire image, so more global features suffice to detect it. Street signs on the other hand tend to only cover very small parts of the image, especially when they are a few meters away from the camera. As this makes them very hard to even see on such a low resolution, we decided to increase the resolution back to the original 480×360 pixels. This change alone did not lead to satisfying results though, as can be seen in figure 5.24b, so we assumed that, even on the larger image resolution, the receptive field used to predict a pixel's class might still be too large.

The receptive field size R of each layer i in a convolutional neural network can be calculated as

$$R_i = R_{i-1} + (k_i - 1)d_i \prod_{j=1}^{i-1} s_j,$$

where k_i and d_i are the kernel size and dilation factor of layer i respectively, s_j is the stride of layer j and $R_0 = 1$, meaning a receptive field of 1×1 Le and Borji [2017]. Transpose convolutions, also sometimes called *fractionally strided convolutions*, are considered to have a fractional stride, in this case of $s_j = 1/2$.

Using this equation for the `droschkinator` architecture reveals its receptive field to have a size of 129×129 pixels. To see whether changing the receptive field does actually have

Architecture	d_4	d_5	d_6	R_{Last}	$\text{IOU}_{\text{street}}$	IOU_{sign}	$\text{IOU}_{\text{other}}$	meanIOU
`droschkinator`	2	4	8	129	**0.94**	0.45	**0.97**	0.79
`dil244`	2	4	4	97				
`dil224`	2	2	4	81	0.92	0.53	0.96	0.80
`dil124`	1	2	4	73	0.93	**0.56**	0.96	**0.82**
`nodil`	1	1	1	41	0.89	0.54	0.94	0.79

Table 5.1: Comparison between the different architectures with their dilation factors in layers 4, 5 and 6, receptive fields at the last layer, per-class IOUs and meanIOUs.

an impact on sign segmentation performance, a modified version of the `droschkinator` architecture was created, with all of the dilation factors set to 1. This model has a reduced receptive field of 41×41 pixels and will be called `nodil`.

The results of this new architecture show an extreme increase in sign segmentation performance but in turn the performance for both the street and the background class declined, as table 5.1 shows. As this showed a clear connection between receptive field size and segmentation performance, multiple variations, with different dilation factors in layers 4 to 6, were trained, to find the best compromise between the different classes. Table 5.1 shows the dilation factors and the receptive field at the last layer of each of these new architectures and figure 5.23 shows a visual representation of their receptive fields on an example image.

Table 5.1 also shows the `per-class IOUs` and `meanIOUs` achieved on the test set. As expected, the `droschkinator` architecture provides the best results for both the street and the background class but performs the worst on the street sign class. Considering the `meanIOU`, the `dil124` architecture seems to yield the best compromise between the classes. Due to low resource availability and technical failures towards the end of the project, metrics for the `dil244` architecture will not be provided but figure 5.24 suggests, that they should be very similar to those of the `dil224` architecture.

Figure 5.24 shows an example segmentation using each of the architectures, with `dil124` clearly providing the best overall results.

Using one of these models to perform the inference on the car makes the whole image processing loop take up to 200ms. Since a street sign has to be correctly segmented and classified in multiple frames before triggering the turning intention, as described in section 5.6.1, this means the car has to drive very slowly. Tests show that decreasing the input resolution to 240×160 pixels, would decrease this time to about 100ms, thus effectively doubling the frames per second and the possible speed of the car. Unfortunately, none of the architectures are able to reliably segment street signs on such a low resolution, so it is kept at 480×360 pixels.

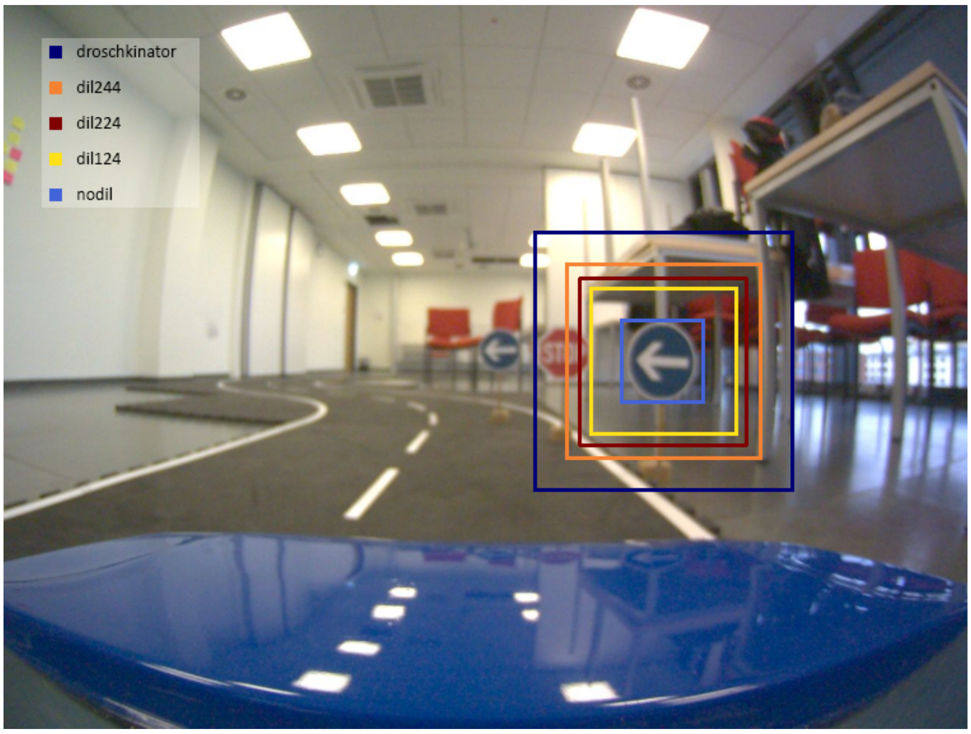
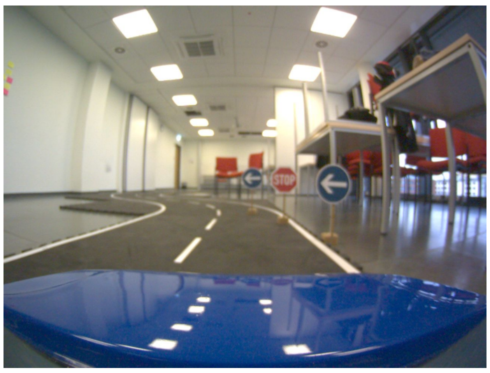


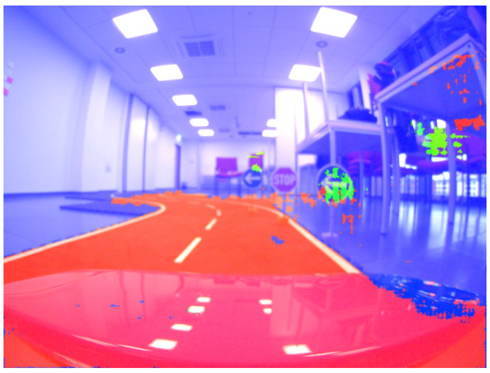
Figure 5.23: Visual comparison between receptive fields of the different architectures.

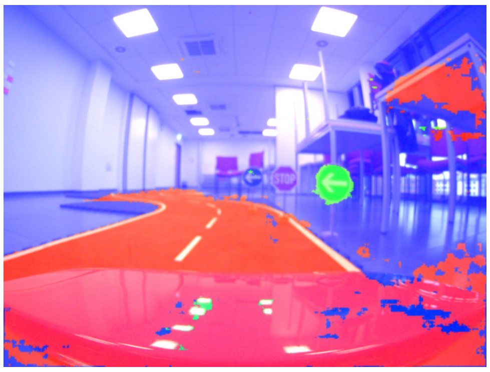
5.4.4 SegNet

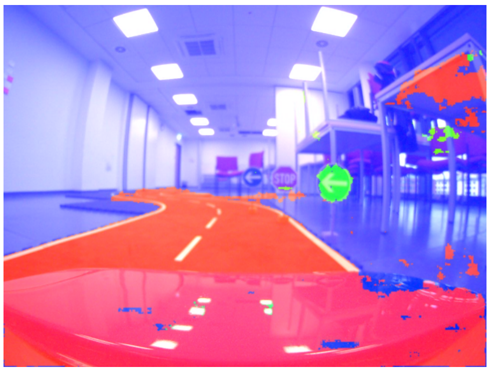
While the network architecture was modified to distinguish between three classes per pixel instead of two, the classification error increases to an unacceptable value. In this state it seems useful to establish a second network with a different architecture to compare the group's results.
Another advantage for this approach is the possibility to use this alternative network when the primary network will not work well enough for the multiclass segmentation. The SegNet is a deep convolutional encoder-decoder architecture for image segmentation which was presented in 2016 (seg). It outperforms several state of the art segmentation networks and solves the pixel-wise classification task for 11 different classes.

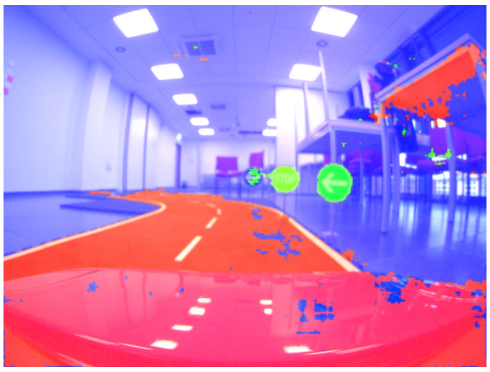
One difference between SegNet and the architectures used in this project is the amount of downsample/upsample layers, where the SegNet has one more for each. This results in a more complex architecture which requires more GPU-memory due to the increasing amount of parameters.
To use the SegNet for the three class segmentation, it was necessary to adapt the model and distinguish only between street, background and signs.
Another difference in the SegNet implementation is the possibility to weight each class with a real value for the loss function to balance the training process. This weighting approach is not integrated in Keras per default but could get implemented if the experiments on the SegNet show reasonable advantages.


(a) Original image


(b) `droschkinator`


(c) `dil244`


(d) `dil224`


(e) `dil124`

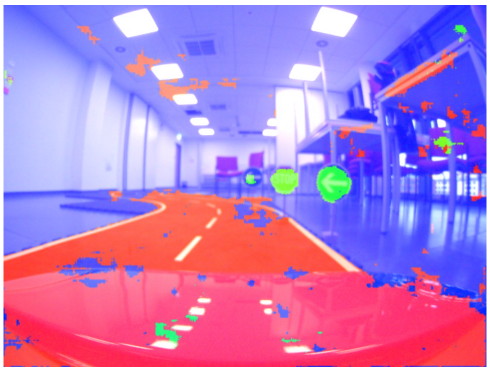

(f) `nodil`

Figure 5.24: Comparison between segmentation results using different dilation factors.

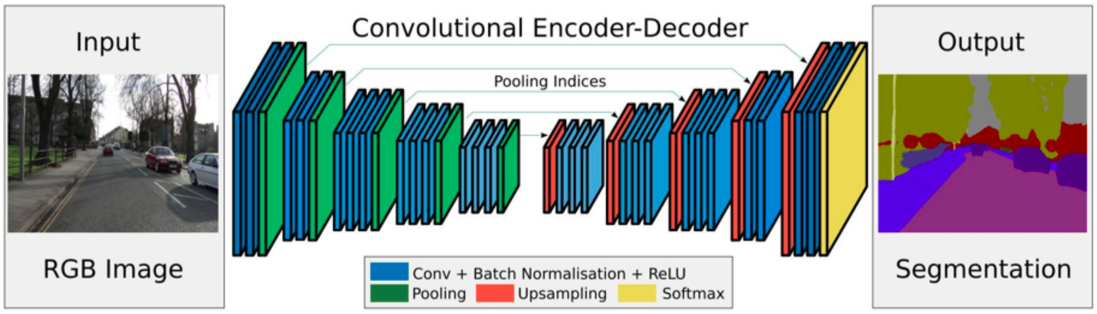
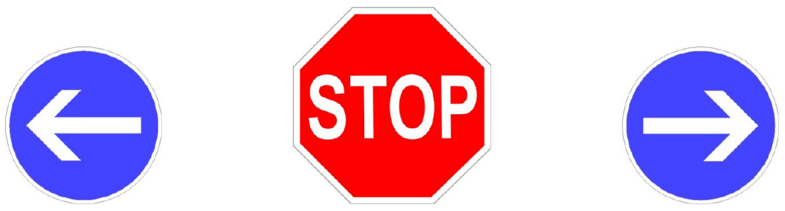


Figure 5.25: Network architecture of `SegNet`

Architecture	$\text{IOU}_{\text{street}}$	IOU_{sign}	$\text{IOU}_{\text{other}}$	meanIOU
`dil124`	0.93	0.56	0.96	0.82
`SegNet`	0.88	0.40	0.94	0.74

Table 5.2: Comparison between the SegNet architecture and dil124



(a) Turn left **(b)** Stop **(c)** Turn right

Figure 5.26: Used street signs for navigation. Signs for turning left, or right at crossings and a stop sign for stopping at the destination.

One further adaption was the expected shape of the labels where the primary model expects three channels per sample with a 1 in the predicted channel and 0 for the rest (one hot encoding). The net expects only one channel with the predicted class per pixel. Therefore it was necessary to convert the complete training data.
Having the SegNet as an alternative network architecture makes the process of experimenting with different architectures for the primary network easier, because the outputs of both networks can be compared to eliminate possible sources of failure and errors.
The table 5.2 shows the different results for segmentation quality. The `dil124` architecture achieved better scores for every segmentation class. It is to be noted that the SegNet architecture was developed for a different problem and is not tweaked to the maximum performance for the presented challenge with three classes. Since the SegNet is supposed to be a reference for providing state of the art results in segmentation, it turns out `dil124` achieved respectable performance.

5.5 Sign Extraction

Three kinds of street signs are going to be distinguished, which are shown in figure 5.26. The segmentation network outputs a segmented image, which contains information about occurrence and position of street signs. In this section, it will be discussed how to use this information to crop the street sign out of the frame, which was captured by the on-board camera. The algorithm is given the inference by the segmentation network and the corresponding camera image.

5.5.1 Filter Artifacts

Because of imperfections in the segmentation process, the sign detection needs to be denoised before the actual sign image can be extracted. In order to do that, the resolution

of the inference produced by the segmentation network gets reduced by a factor of 4 and after that *morphological transformations* are applied.
First *opening* is applied, which consists of *erosion* and *dilation* in that order. *Erosion* removes a one pixel layer from each group of pixels and *Dilation* adds a one pixel layer to each group of pixels. This removes single pixels and noise from the image, while keeping large structures intact. Afterwards *closing* is applied, which is the exact opposite of *closing*, *dilation* first and *erosion* second. This closes gaps inside structures to prevent interpreting one structure as multiple different ones. The image is then rescaled to the original size in order to use the positional information given by the segmentation process.

5.5.2 Align Contours around Street Signs

After filtering the artifacts there should mainly be actual objects of class street sign left in the inference. In order to cut these objects out of the camera image, contours need to be aligned around these objects. These contours get computed similarly to a convex hull and are needed to infer a bounding box. For this, the OpenCV implementation of Suzuki and Abe [1985] was used. The bounding box is then cut out from the camera image, extracting the street sign.

Without reduction of the resolution, eroding and dilating the objects, even little noise is detected as an actual street sign, as in figure 5.27. Even after preprocessing, not all bounding boxes contain actual street signs, because the filtering of the artifacts does not guarantee to remove all artifacts. As figure 5.28 shows, there might be very large artifacts, which can probably survive the reduction of resolution and the erosion. To maximize the probability of detecting the nearest street sign, which, for the purpose of turning intentions, is the most important street sign, a heuristic approach is used where only the largest bounding box is chosen. This also means that, even if there are multiple street signs on a camera image, at most one street sign will be classified. Figure 5.28 shows the found bounding boxes of the street signs after preprocessing of the inference. After the bounding box gets cut out of the camera images, it is resized to a fixed resolution of 32 by 32 pixels as this is the size of the GTSRB street sign data set which will be incorporated in the street sign classification training process (see 5.6.1). As shown in figure 5.29 these images are sometimes not as easy to distinguish as the street signs of figure 5.26, although most can be identified by humans.

The preprocessing steps described above are implemented as an ADTF filter which takes the segmented image from the segmentation filter as well as the corresponding, original camera image. As an output, it sends the 32 by 32 pixel street sign, if found. In order to filter out additional noise a minimum width and height was chosen for a detected sign to actually get sent out. Because the largest bounding box is always chosen, this adds another

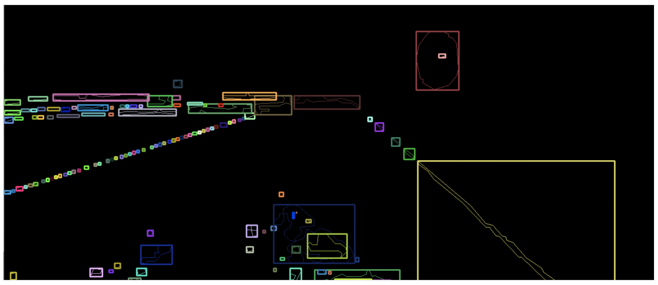


Figure 5.27: Found objects without preprocessing steps (reduction of resolution, opening and closing). Notice that, due to the imperfection of the segmentation, many artifacts were identified to be part of the street sign.

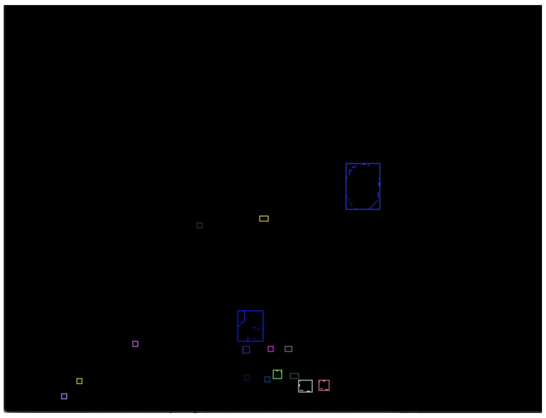


Figure 5.28: Found objects with preprocessing steps (reduction of resolution, opening and closing). Notice that the largest bounding box in this example represents the actual street sign.

layer of filtering as small size noise will not mistakenly be sent out as a recognized sign. The exact values for these parameters where set to **(50,50)** after manual experiments.

5.6 Sign Classification

5.6.1 Street Sign Classifier

After cutting out the nearest street sign, a small classifier is needed to classify the type of street sign detected. Similar to the image segmentation network, the sign classification task is handled by a CNN but, as shown in figure 5.30, there are much fewer layers, since the classification task is much simpler. Also, the classifier outputs a prediction vector with confidence values for each type of street sign. Categorical crossentropy was used as the training loss and the Adam was used as the optimizer, both of which were provided by Keras.

As shown in 5.26 the classifier needs to be able to distinguish three different signs. However, since the approach for cutting out street signs is heuristic and thus not perfect, the classifier also needs to be able to output *No sign*. Thus, a fourth class was added,

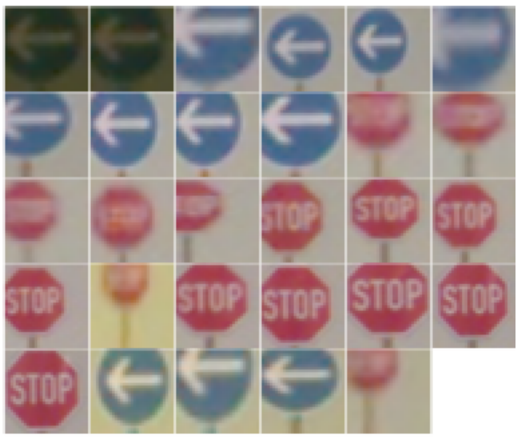


Figure 5.29: Multiple Sequences of consecutive frames with detected cropped street signs. These are the input features for the street sign classifier

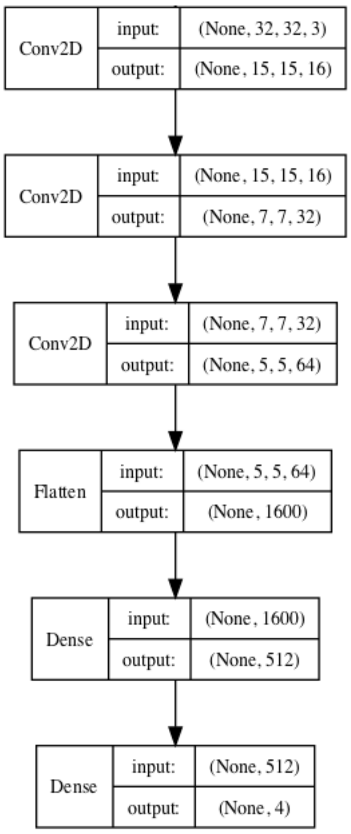


Figure 5.30: Architecture of the proposed street sign classifier by Shustanov and Yakimov [2017].

which is trained by providing random excerpts from existing, real life images captured in *rl_complete* which could be mistaken for a street sign.

Street sign data

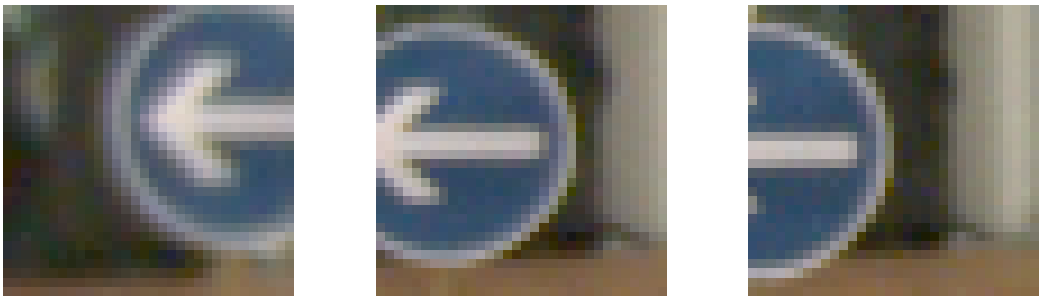
Two sources of training data were used to train the street sign classifier. First, 150 training images (per class) were sampled from the GTSRB street sign dataset provided by Stallkamp et al. [2011]. Since the dataset does not contain the preferred left and right signs a decision was made to take images from the *straight ahead* arrow class and rotate them to be either left or right arrow signs. For the *No sign* class, images were created in an automated fashion using already segmented images from the *rl_complete* dataset. Parts of segmented images which do not belong to the class *street sign* were automatically cut out and resized to 32 by 32 pixels.
Additional training data was manually created from *rl_complete*, using the *street sign* class. For this, the cutout algorithm described above was used on images with exactly one street sign, automatically extracting them. However, this resulted in relatively few images since not every image in *rl_complete* contains street signs. In order to increase the number of these images to the desired 150, subimages were cut out. A subimage was defined as a 22 by 22 square positioned randomly inside the original image. Three subimages per original were created, thus not only quadrupling the data but also making the classifier more robust to partial images. Overall, this resulted in 300 training images per class.
The same approach was used to create 100 test images per class, although no images from GTSRB were used in order to test the classifier on data as close to the actual use case as possible. Also, automatic augmentation of the training images was applied during training in order to make training more robust. The augmentation consists of a 10 degree rotation to either side, a random change in contrast and the possibility for added Gaussian blur. Overall, after augmentation, the training set contained roughly 7800 images.

Experiments and results

Based on a proposed street sign classification model by Shustanov and Yakimov [2017], which can be seen in 5.30, three architectures were trained and evaluated:

1. The original architecture *(Arch1)*
2. Same as 1, but with added dropout after fully-connected layer *(Arch2)*
3. Same as 2, but with batch normalization after each convolutional layer *(Arch3)*

Dropout is a method for regularization in neural network, originally proposed by Srivastava et al. [2014]. While training, it disables each neuron of a specified layer with a probability of p. Thus, the network is forced to chose different routes and thus is less likely to overfit.



(a) Right boundary not visible **(b)** Left boundary not visible **(c)** Only arrow body visible

Figure 5.31: Three example cases in which the classifier would get confused. Notice that, for all three cases, the left arrow is only partially shown.

Architecture	Sign	Precision	Recall	F1-Score
Arch1	NoSign	0.98	0.93	0.95
	Stop	0.99	0.99	0.99
	Right	0.85	0.79	0.82
	Left	0.81	0.91	0.85
Arch2	NoSign	0.96	0.96	0.96
	Stop	0.98	0.98	0.98
	Right	0.90	**0.89**	0.89
	Left	**0.90**	0.91	**0.90**
Arch3	NoSign	**0.99**	**0.98**	**0.98**
	Stop	**1.00**	**1.00**	**1.00**
	Right	**0.92**	0.88	**0.90**
	Left	0.88	**0.92**	**0.90**

Table 5.3: Results on test data after training each classifier architecture for 150 epochs. Best values are shown in bold for convenience.

Naturally, dropout has no effect when using the network for inference. p was set to 0.9, which drops each neuron with a probability of 90%.

For evaluation, the F1-Score was used since it gives a better insight into how and why the model performs as opposed to raw accuracy while evaluating. Precision measures the ratio of true positive decisions to the sum of true positives and false positives. In other words, it measures (per class) how many of the signs classified were actually of that class. On the other hand, recall is given by the ratio of true positives to the sum of true positives and false negatives. That way, one measures (again, per class) how many of the test images were actually classified with that class. The F1-Score is then simply computed by taking the harmonic mean of precision and recall.

$$\text{Precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}$$

$$\text{Recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$$

$$\text{F1} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

As can be seen in 5.3, the third architecture with batch normalization and dropout resulted in the best overall model, achieving F1-Scores of at least 0.9 on all classes. This came as no surprise since the original architecture has no form of regularization and thus should not generalize as good as the alternative architectures. Because of this, a decision was made to use *Arch3* as the street sign classifier.

After evaluating the model on the actual course it became clear that one particular scenario was fooling the classifier. This case takes many forms, some of which are presented in 5.31. As one can see, the arrow is not fully present in the images, causing the classifier to decide on the opposite sign (in this case *Right sign* instead of *Left sign*). Additional training data for this specific scenario was captured and added to the training data, resulting in far fewer errors of this kind. Most likely, this error was caused by the GTSRB training images, because they only feature street signs centered in the middle of the image with background around them and no partial signs.

Implementation of classification filter

When implementing the network described above into an ADTF filter, extra steps were taken to ensure that a turning intention is correct, since a wrong turning intention can cause the car to leave the track completely and may thus prevent the car from reaching its desired destination.

First, the results of the classifier are aggregated into a buffer of size 4. The buffer is reset every two seconds. Once the buffer is filled, the filter checks if the buffer contains one class with more than 70%. If so, this class of sign is send out as *detected*. By choosing the prevailing class of frames detected the confidence that the turning direction is correct was increased. Additionally, a classifier result is only added to the aggregation buffer if the confidence of the highest class is bigger than 0.8. These parameters were determined by manual experimentation.

5.7 Evaluation

In this chapter, the success of the semantic understanding aspect of the project group will be made, based on the requirements that were set in section 2. Additionally, proposals for improvements on further project work will be given, based on the experiences during the project.

5.7.1 Fulfilling requirements

The first three requirements GEN-SEM-1, GEN-SEM-2, and GEN-SEM-3 demand that the car distinguishes between drivable and non-drivable areas of the floor mats on straight streets, bendings, and intersections. In the setting where only those two classes exist, the neural network was able to distinguish well between them, in all three settings. When adding the task of recognizing street signs, the quality of the segmentation became clearly worse for all road settings, although still remaining good enough for the control logic to properly steer the car. In some cases, the middle of the intersection became hard to recognize. This is, however, an inherently difficult problem, since the local features at this spot all simply indicate dark spots without any guiding white lines around them to indicate that they belong to the street.

Requirement GEN-SEM-4 asks for the car to detect obstacles as non-drivable areas. This is the case for obstacles that are similar to those included in the training data, for example other cars. As is generally the case for neural networks, it can not be guaranteed that this holds for all possible obstacles. Since the requirement only asks that it 'should' classify obstacles as non-drivable, it can be regarded as fulfilled.

The last three requirements GEN-SEM-5, GEN-SEM-6, and GEN-SEM-7 ask that the three different street signs be recognized. This task consists of two parts. First, the segmentation network must segment the sign within the camera image. Then, this segmented area is cut out and classified into one of the sign classes, or the 'NoSign' class. Under the right conditions, i.e. similar to those in the training data, this process works very well. The classification achieves f- scores above 0.8 on all classes. Combined with an aggregation over several frames, this leads to very few misclassifications in practical test runs. The segmentation network, however, depends on the test conditions being very similar to those in the training data. Different lighting and different environments surrounding the road panels, e.g. the difference between the test hall and the development area, were detrimental to the performance of the segmentation, as compared with the strong results shown on pre-recorded test data. Segmenting the signs from a large distances was therefore not possible in the physical test environment. While this task has been solved in principle (indicated by high mean IOU on the test set), the performance varies with changing environment conditions. This is of course true for all segmentation efforts in the group's experiments, but mostly proved a problem with regard to segmenting signs.

5.7.2 Lessons learned

In this section, experiences will be shared on how the project work could be improved if it were repeated.

First off, the initial network to work with should have been the Segnet, as the standard approach to solving automotive segmentation tasks. While it is bigger than the UNet, it can be used as a benchmark to compare other approaches to, and see how different

data augmentation or preparation techniques impact the performance. Furthermore, a pruning approach could have been used on the trained Segnet to decrease the size, thereby accounting for the limited computational resources available.
Regarding the self-developed network architectures, an approach that was not explored would have been to use Residual BlocksHe et al. [2016], which could enable precise reconstructions of the borders of the road and detected signs.
With regard to choosing a course of action, it might have been better to work on street sign detection and street detection at the same time, either with two teams in parallel, or starting out with three classes right away. The sign detection proved more difficult than anticipated, leading to bottle-neck problems towards the end of the project, since other components relied on a working street sign detection. Additionally, especially the beginning of the project was marked by trial&error experimentation and visual inspection instead of systematic experiments, based on factual metrics. However, while working on the sign detection, it also became clear that those metrics would often show promising results during training on testing, but the network would perform a lot more poorly when in the physical testing environment. Evaluation by visual inspection therefore can not be eliminated completely from the process.

Chapter 6

Lane Keeping

Programming a car to be autonomous is a modern topic which is studied profoundly today. Research topics like pattern recognition (see chapter chapter 5) and car-specific embedded systems (see chapter 3) have to be considered to make a car drive reactively to its surroundings. For an autonomous driving style method of motion planning and control theory are required. The former makes use of the segmented image as described in chapter chapter 5 to compute a drivable path for the car. In other words, the question *where* the car drives to is answered, namely the *path planning.* The latter focuses on *how* to drive, namely the *motion control.* This chapter describes the approach of lane keeping from the perspective of the path planning and the motion control. At first, a motivation for this issue is given. Thereafter, the chosen handling of the orientation error of the vehicle is discussed. Two different path planning algorithms are presented and explained, whereby both the computation of the desired destination and the computation of the error angle are explained. Furthermore, this chapter deals with controlling the car to follow a lane, using the previously computed error angle.

6.1 Introduction and Basics of Motion Control

In this section, an overview over the basics of path planning and motion control are given.

Path Planning

In chapter 5 methods were introduced to segment the car's camera images. For both the trapezoid front-view as well as the bird's eye view the result is a segmented black-white image, which is represented as a matrix containing the values 0 (black pixel) and 255 (white pixel) in each cell, where white pixels represent the drivable lane. The aim of path planning is to find a destination point for the car to head for, based on the given segmentation. Different heuristics to find the best destination point were tested in section 6.2, based on different assumptions. In subsection 6.2.1, it is assumed that the car is only driving straightforward and there are no obstacles on its route. In subsection 6.2.3, the premise of driving on a straight road only is loosened. Therefore, a path planning heuristic is described that enables the car to keep the lane in bendings.

In the second half of the project, the input from the lane detection module was changed, so that the white pixels indicate the whole street and not only the right lane. This change supports a clearer separation between the semantic understanding and the decision and control layers in the implemented architecture and creates a basis for an advanced path planning algorithm. So, a new algorithm for path planning based on a histogram of drivable pixels was developed, which is described in subsection 6.2.4. The main aim of this development was the ability to turn on crossings as it is stated in the requirements GEN-CON-3 to GEN-CON-8, which was the primary target of the second half of the project.

Motion Control

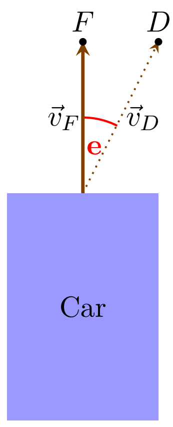


Figure 6.1: Birdseye-view of a car facing towards point F. A new destination point D is computed with a path planning algorithm. The angle between the vectors $\vec{v}_F$ and $\vec{v}_D$ is called the *error angle* **e**.

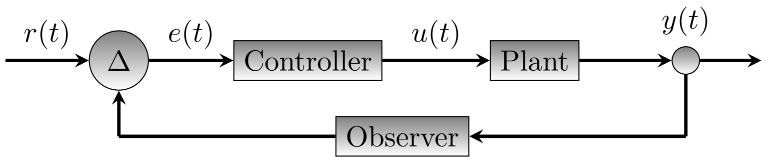


Figure 6.2: A closed-loop system. $y(t)$ is the current state of the controlled parameter, $r(t)$ is the desired state of the controlled parameter, $e(t) = r(t) - y(t)$ is the error, $u(t)$ is the control output.

Once a new destination point **D** is decided using a path planning algorithm, the angle between the vector $\vec{v}_F$ of the car's current facing direction and the vector $\vec{v}_D$ from the car's front to **D** can be computed (see Figure 6.1). This angle is called the *error angle* **e**. It is necessary to quickly reduce **e** while the car is driving to reach the destination point

D, as **e** = 0 implies that the car is facing towards the destination point **D**. However, it is also necessary to reduce **e** in such a way that abrupt and non-static behavior of the car is prevented. Therefore, methods of Control Theory are required, namely *dynamic control-loop systems*. Dynamic control-loop systems describe real-time scenarios where the change of certain system parameters is controlled over time. They consist of a *plant*, a desired parameter state as a *control input* (also called *setpoint*) and a *controller*. The former is a mathematical representation of a system whereas the latter is a heuristic that uses the control input to compute a control output which will be passed to the plant.

There are two kinds of control-loop systems, namely *open-loop* and *closed-loop* systems. While an open-loop system has no more components than those previously described, a closed-loop system also contains an *observer* (also called *sensor*) that monitors the controlled parameter and creates a feedback loop (see Figure 6.2). By using an observer, the dynamic closed-loop system can compute the difference between the desired and the current parameter state (denoted as $r(t)$ and $y(t)$), which is called the error *error* $e(t) = r(t) - y(t)$. This error is passed to the controller (as the controller input) to compute a control output $u(t)$, which is further passed to the plant. By describing the car's driving style as a dynamic closed-loop system, it is possible to reduce $e(t)$ both quickly and periodically over time and to prevent a high overshoot of the steering angle.

Altogether, a dynamic closed-loop system for the car has the following specifications:

- *Parameter to control* is the steering angle (also called the *yaw angle*).
- The *observer* is the segmented image. Strictly speaking, it is the Basler Camera (see subsection 3.1.2), as the it supplies the natural images for the segmentation (see chapter 5). However, it is assumed that the car is located at the bottom center of the segmentation and facing straight ahead at all times. In the segmentation, the vector $\vec{v}_F$ of the car's current facing direction always points to the center of the segmentation (also see Figure 6.5).
- The *control input* is the error angle (see Figure 6.1).
- Because of the car's actuators, the frequency for the computation is bounded to approximately $\frac{1}{30}s$, thus there will be about 30 iterations of the closed-loop system per second. Ideally, every iteration reduces $e(t)$ slightly to guarantee a smooth driving style.

The *controller* heuristics will be described in section 6.3 and the *plant* is a state space model of the car, which will be explained in the remainder of this chapter.

To describe the car mathematically as a state space model, *kinematic equations* are required, which describe the motion of objects in space and time (see Theorem 6.1.1). As the car is a rigid body that is assumed to perform only planar motion, the kinematic equations can be described using three coordinates.

6.1.1 Definition. (kinematic equations, Zhao et al. [2012])
Let x and y be the car's coordinates, r the car's angular velocity around the center of gravity, u, v the longitudinal and the angular velocity. Then basic kinematic equations are defined as follows:

$$m(\dot{u} - vr) = \sum F_x \tag{6.1}$$

$$m(\dot{v} - ur) = \sum F_y \tag{6.2}$$

$$I\dot{r} = \sum M_z \tag{6.3}$$

where m is the vehicle inertia mass, I the vehicle yaw moment inertia, $\sum F_x$ and $\sum F_y$ are the net force components in the x and y direction and $\sum M_z$ is the external torque around the z axis.

Following the approach in Zhao et al. [2012], a constant longitudinal velocity $u = u_c$ is assumed. Therefore, Equation 6.1 can be omitted, resulting in a *dynamic two degree-of-freedom model.* A vector $(v, r)^T$ with arbitrary lateral and angular velocities v and r is therefore called a *state.* Equation 6.2 and Equation 6.3 describe the motion of an object using net force components in the y and the z direction. These equations - or rather the net force components - need to be further specified using the vehicle parameters described in Table 6.1 to compute a car-specific state space model.

The lateral forces of the left and right tires are assumed to be equal, however, the lateral forces of the front and rear tires (denoted as F_{yf} and F_{yr}) need to be distinguished as the front wheel is the steering wheel. Equation 6.2 and Equation 6.3 can be expressed as in Equation 6.4 and Equation 6.5.

$$m(\dot{v} - u_c r) = F_{yf} + F_{yr} \tag{6.4}$$

$$I\dot{r} = aF_{yf} - bF_{yr} \tag{6.5}$$

The lateral forces F_{yf} and F_{yf} can be approximated as

$$F_{yf} = -C_f \cdot (\frac{v - ar}{u_c} - \delta_f), \tag{6.6}$$

for an arbitrary front wheel steer angle δ_f, and

$$F_{yr} = -C_r \cdot (\frac{v - br}{u_c}). \tag{6.7}$$

Equation 6.6 and Equation 6.7 can be substituted into Equation 6.4 and Equation 6.5 to form vehicle-specific kinematic equations using the vehicle parameters described in Table 6.1.

$$m(\dot{v} - u_c r) = C_f \delta_f - \frac{(C_f + C_r)}{u_c} v - \frac{(aC_f - bC_r)}{u_c} r \tag{6.8}$$

$$I\dot{r} = aC_f \delta_f - \frac{(aC_f + bC_r)}{u_c} v - \frac{(a^2 C_f - b^2 C_r)}{u_c} r \tag{6.9}$$

Parameter	Unit	Description
m	kg	vehicle mass
I	$kg \cdot m^2$	vehicle yaw moment inertia
w	m	wheelbase
a	m	longitudinal position of front wheel from vehicle center of gravity
b	m	longitudinal position of rear wheel from vehicle center of gravity
h_g	m	height of vehicle center of gravity
C_f	$N \cdot rad^{-1}$	cornering stiffness of front tire
C_r	$N \cdot rad^{-1}$	cornering stiffness of rear tire
u_c	$m \cdot s^{-1}$	longitudinal velocity

Table 6.1: pertinent vehicle parameters for space state model, see Zhao et al. [2012]

The resulting state space system $\mathcal{M}$ is shown in Equation 6.10.

$$\begin{bmatrix} \dot{v} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} v \\ r \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \delta_f(t) \tag{6.10}$$

With:

$$a_{11} = -\frac{C_f + C_r}{u_c m} \qquad a_{12} = -u_c - \frac{aC_f - bC_r}{u_c m}$$

$$a_{21} = -\frac{aC_f - bC_r}{u_c I} \qquad a_{22} = -\frac{a^2 C_f - b^2 C_r}{u_c I}$$

$$b_1 = \frac{C_f}{m} \qquad b_2 = \frac{aC_f}{I}$$

The vehicle used in Zhao et al. [2012] is a 1.6L Tiggo3 SUV with the parameters listed in Table 6.2. The resulting state space system $\mathcal{M}_{Tiggo}$ is shown in Equation 6.11.

$$\begin{bmatrix} \dot{v} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} -3.785 & -19.167 \\ 0.469 & 0.976 \end{bmatrix} + \begin{bmatrix} v \\ r \end{bmatrix} + \begin{bmatrix} 34.409 \\ 27.686 \end{bmatrix} \delta_f(t) \tag{6.11}$$

The car used in the virtual environment as well as in the physical world both are an Audi Q2 (see Figure 3.1), however, their parameters differ vastly as the physical car is a miniature model of the original car. To some degree the model-car's parameters may be

Parameter	Value
m	2325
I	4132
w	3.025
a	1.430
b	1.595
h_g	0.5
C_f	80000
C_r	96000

Table 6.2: Car parameters of Tiggo3 SUV

scaled down in proportion to the original car, but especially weight is a factor that cannot be scaled authentically with all other parameters. In Table 6.3 the parameters for the car in the simulation are shown. As a number of approximations were made, they might be not precise.

Parameter	Value
m	1810
I	3815[a]
w	2.813
a	1.407[b]
b	1.407
h_g	0.5[c]
C_f	62280[d]
C_r	74735[e]

[a]As in McHenry and McHenry [2008], a brief approximation for the Yaw Inertia is $I = M \cdot \frac{x^2+y^2}{12}$, where M is the vehicle's mass and x and y are the vehicle's length and width. For this car, the values are $x = 4.646m$ and $y = 1.926m$.

[b]The center of gravity is assumed to be in the exact center of the wheelbase.

[c]As both the Tiggo and the Audi Q2 are SUV's, the height of the center of gravity is assumed to be h_g as in Table 6.2.

[d]As mass affects this value the most, this value was taken proportionally from the mass of the Tiggo to its front tire stiffness. The mass-to-front-tire-stiffness ratio is 34.41, so this value is the product of the mass of the car and this ratio.

[e]The mass-to-rear-tire-stiffness ratio is 41.29.

Table 6.3: Car parameters of VTD car

The resulting state space system $\mathcal{M}_{VTD}$ is shown in Equation 6.12.

$$\begin{bmatrix} \dot{v} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} -3.785 & -19.516 \\ 0.2297 & 0.3231 \end{bmatrix} \begin{bmatrix} v \\ r \end{bmatrix} + \begin{bmatrix} 34.409 \\ 22.968 \end{bmatrix} \delta_f(t) \tag{6.12}$$

The parameters for the model car are shown in Table 6.4. Again, a number of approximations were made. Also, for the model car, a constant velocity is assumed.

Parameter	Value
m	7.5
I	0.235[a]
w	0.37
a	0.185[b]
b	0.185
h_g	0.1
C_f	258[c]
C_r	310

[a]The same formula as in Table 6.3 was used. The model car's length is $0.54m$, the width is $0.29m$.
[b]The center of gravity is assumed to be in the exact middle of the wheelbase.
[c]The same assumption as in Table 6.3 was made.

Table 6.4: Car parameters of model car

The resulting state space system is $\mathcal{M}_{Q2}$ is as follows:

$$\begin{bmatrix} \dot{v} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} -18.17 & -3.861 \\ 9.759 & 1.805 \end{bmatrix} \begin{bmatrix} v \\ r \end{bmatrix} + \begin{bmatrix} 34.409 \\ 203.32 \end{bmatrix} \delta_f(t). \tag{6.13}$$

As a specific model is used during the project, a precise model for motion control is needed. That is the reason for the use of different controllers, namely a P-controller and a PID-controller. For more information about controllers, see section 6.3.

6.2 The Orientation Error of the Vehicle

In the upcoming sections, first a desired path that follows the road and has enough clearance for the car to drive along is defined and following the computation for such a desired path is presented. Afterwards the calculation for the resulting yaw error (see Figure 6.1), that the car tries to minimize to follow a desired path, is described. Lastly, a naive path planning approach and a more sophisticated histogram-based path planning method are introduced to improve the quality of driving autonomously.
An overview of all implemented components and how they interact with each other is shown in Figure 6.3.

6.2.1 Computation of the Desired Path

In the first half of the project the basis for the computation of the desired path was based on the Lane Detection algorithm described in section 5.2, which was implemented for a proof of concept. A segmentation with a CNN was not yet implemented at this stage. The lane detection method was able to detect simple lanes without any obstacles or crossroads, thus the first priority was to gain experience on driving on such simple lanes. To do so, the middle of it is assumed to be the desired path, as it follows the road and has always enough clearance for the car to navigate.

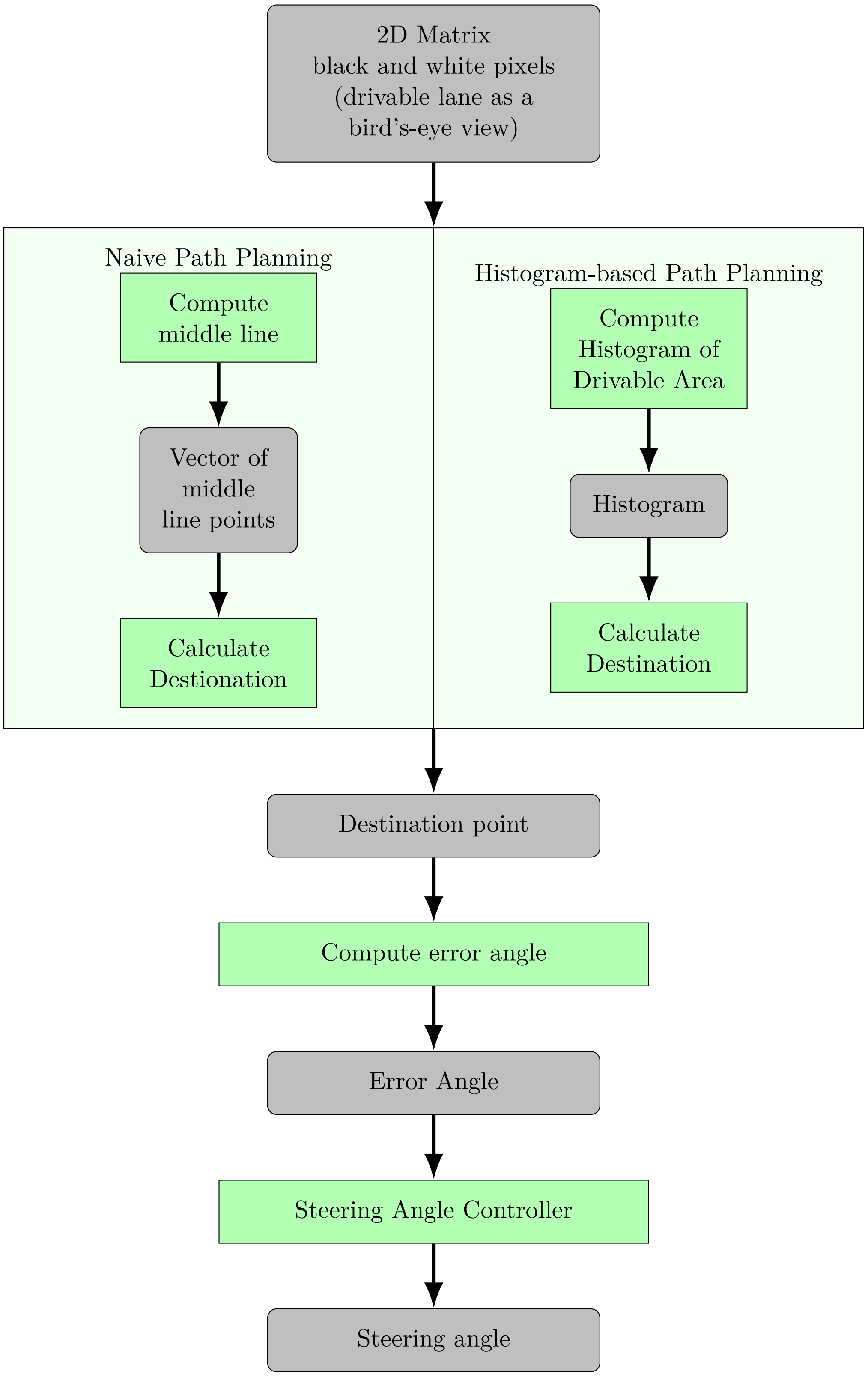


Figure 6.3: Overview of the implemented components for lane keeping.

```
Input: matrix of pixels
Output: middle line of the drivable lane
  initialize middle line as empty list
  for each row do
    if white pixels exist then
      add the middle pixel between the first and last white pixel to middle line
    end if
  end for
  return middle line
```

Algorithm 6.1: Computation of the middle line

The computation of this middle line is achieved through algorithm 6.1. It was developed with simplicity in mind to accomplish the proof of concept stated in subsection 1.2.1 quickly. This algorithm is described in the following.

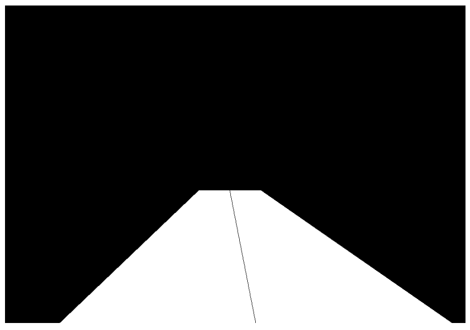
The input for the algorithm is a matrix containing a binary image of black and white pixels. White pixels represent the drivable lane from a bird's-eye view as detected by the semantic understanding module, described in section 5.2.

To compute the middle of the lane, the positions of the borders of the lane have to be found. The algorithm searches for white pixels in each row of the delivered matrix and labels the x-coordinates as x_{first} and x_{last} for the first and last found white pixel respectively. These pixels form the borders of the lane, so their x-coordinates can be used for the computation of the middle of the lane for each row.

The x-coordinate of each row's middle pixel is computed as the arithmetic mean of both borders pixels x-coordinates, i.e.:

$$\frac{x_{first} + x_{last}}{2}$$

At the end of the algorithm, the points are returned as a list of points, which is used as the desired path for path planing.


(a) Middle line of the lane shown as a trapezoid produced by an early version of the lane detection algorithm.

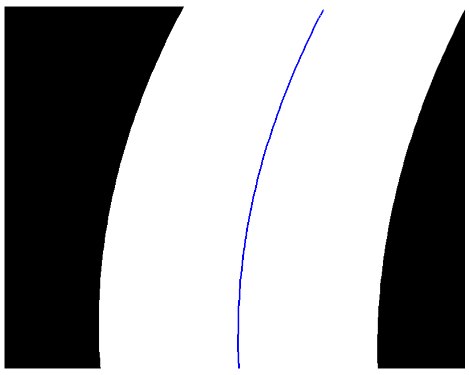
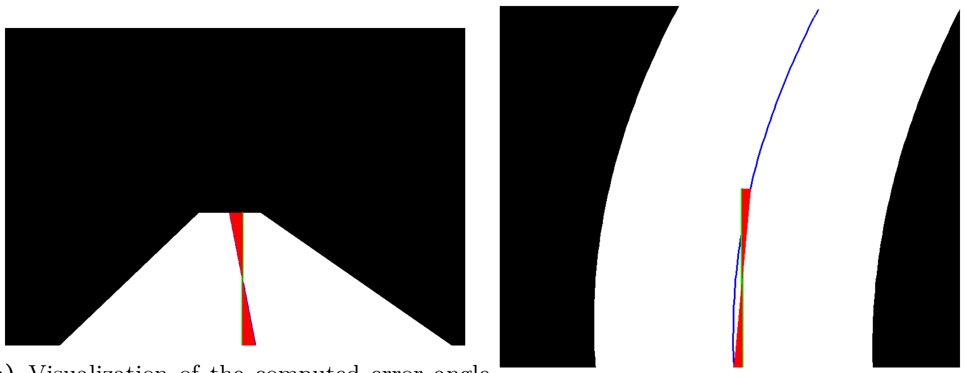

(b) Middle line of the lane shown from a bird's-eye view produced by the advanced lane detection algorithm.

Figure 6.4: Examples for calculated desired paths.



(a) Visualization of the computed error angle using the whole middle line.

(b) Visualization of the computed error angle using half of the middle line.

Figure 6.5: Two examples of calculated error angles. The blue line shows the calculated middle line, the green line describes the current heading direction and the red area describes the error that has to be minimized in order to keep the car in the lane.

In an earlier state of the project, the output from the semantic understanding module was an image showing a detected lane from a frontal view from the car's hood. The image was therefore always a simple trapezoid, as shown in figure 6.4a.

When an advanced lane detection algorithm was implemented (see section 5.2), the input changed to a bird's-eye view of the road ahead of the car. The algorithm to obtain the middle line can be used with both image types without the need of any adaptations. Figure 6.4b shows the result of a calculated middle line given a bird's-eye view image.

6.2.2 Computation of the Error Angle

The current heading direction of the vehicle is abstractly seen as the y-axis of the car's view. The error angle ϕ_{error} between the y-axis and the computed middle line of the lane, see Figure 6.1. It is calculated as shown in equation (6.14).

At first, the points used for the calculation were the first point (x_{first}, y_{first}) and the last point (x_{last}, y_{last}) of the recognized middle line in the whole trapezoid sighting, as seen in figure 6.5a.

$$\phi_{error} = \arccos\left(\frac{|x_{last} - x_{first}|}{\sqrt{(x_{last} - x_{first})^2 + (y_{last} - y_{first})^2}}\right) \tag{6.14}$$

When the input for the middle line computation was switched to the Advanced Lane Detection algorithm, which was able to also detect bends, the computed error angle was observably too high in bends of the road, as the first and the last point on the middle line were used.

As a consequence was that the vehicle oversteered even in the slightest bends. The chosen solution for this problem was using only a fraction of the recognized middle line for the computation of the error angle, because this way it could be ensured that the vehicle adapts

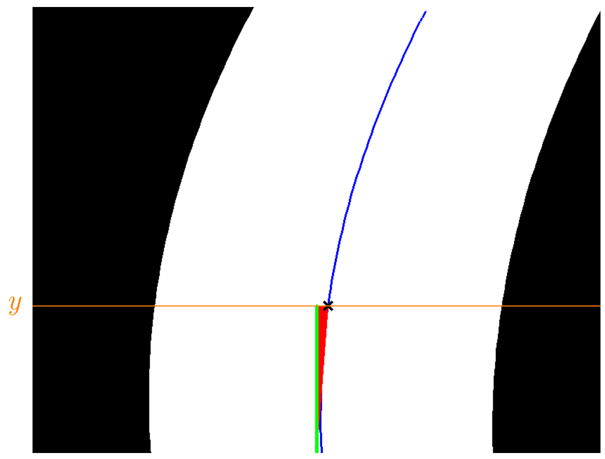


Figure 6.6: Example of an error angle when using naive path planning.

its motion to its direct vicinity and not to the part of the lane which is not urgently relevant yet. Figure 6.5b shows a new error angle, now based on the bird's-eye view and only using half of the middle line.
For following the line, the computed error angle has to be minimized, so that the heading direction of the vehicle, corresponds to the desired path. This minimization is achieved by using a controller, which is explained in detail in section 6.3.

6.2.3 Naive Path Planning

Using the trapezoid images the vehicle was able to correct its yaw error with the introduced methods. The introduction of the bird's-eye view removed the vanishing point from the image. Thus, only the yaw error, but not the distance to the center of the road are corrected, which lead to the car drifting away slowly from the detected road. A naive path planning was implemented to cope with this new behavior.
To incorporate the cars distance to the center of the road a more suitable error angle is necessary. A naive approach is to use the angle to a destination point on the calculated middle line. Choosing a destination point from the middle line is achieved by intersecting the middle line with a vertical scanline, as shown in Figure 6.6. The y-value of the scanline is set by a hyper parameter. The error is small when the car is heading directly to a destination point and large when the car is not facing the destination directly, thus not driving directly to the center of the lane.
The simple method to estimate a desired path described in section 6.2.1 and section 6.2.2 has proven to be sufficient in simple scenarios where the whole street is inside the field of view. In situations where the car is off-center on the road and an edge of the road is not visible, the middle line algorithm cannot compute a continuous middle line. Figure 6.7a shows an example in VTD where the vehicle is off-centered to the left and the lane is not completely in the field of view. The resulting bird's-eye view with the calculated middle line is pictured in figure 6.7b. Because the lane exits the picture to the sides in the top and

bottom parts, algorithm 6.1 is not able to estimate center points for these rows. If this happens on the scanline mentioned in Figure 6.6 a destination point cannot be calculated. To solve this problem, the missing parts of the desired path are approximated using a Lagrange polynomial. Defining a polynomial of degree k in Lagrange form requires $k+1$ support points. Therefore, $k+1$ equidistant points from the calculated middle line are used as supports. From the calculated polynomial n new points are evaluated and used as target points.

Let $(x_0, y_0), \dots (x_k, y_k)$ be points from the calculated middle line. These points will be used as pairs of arguments and values for the interpolation polynomial. A Lagrange base polynomial is generated for every point

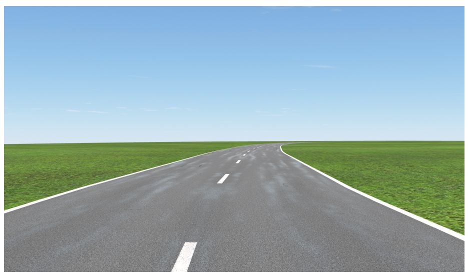
$$l_j(x) := \prod_{\substack{m=0 \\ m \neq j}}^{k} \frac{x - x_m}{x_j - x_m}$$

with $0 \leq j \leq k$. The interpolation polynomial is then given as a linear combination of base polynomials and the corresponding values

$$L(x) := \sum_{j=0}^{k} y_j l_j(x)$$

As an example, Figure 6.8a shows five points on the calculated middle, which are used to calculate five base polynomials. Using the Lagrange polynomial defined by the five base polynomials eight new points in figure 6.8b are interpolated.

Similar to the error angle described at the beginning of the chapter, an interpolated destination point will be chosen even if parts of the drivable lane are cut away in the bird's-eye view. Figure 6.9 shows an example where the third point from eight newly interpolated points is chosen as the destination point.


(a) VTD image with vehicle off center to the left.

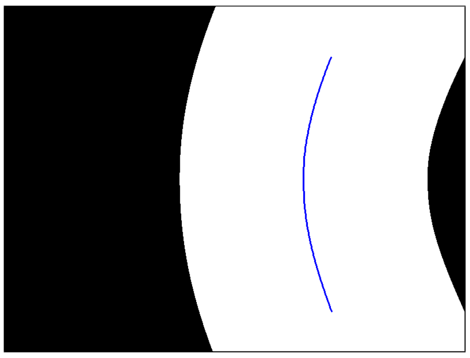
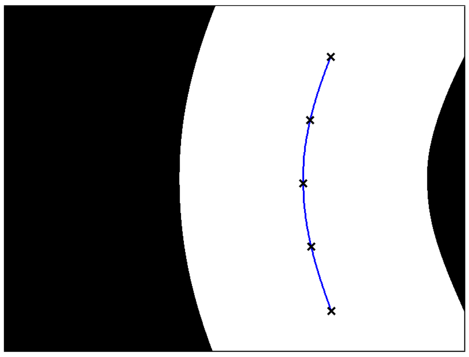

(b) Resulting bird's-eye view.

Figure 6.7: Example for a problematic scenario for the simple method of finding a desired path.


(a) Five points taken from the simple middle line from the same bird's-eye view as in figure 6.7b.

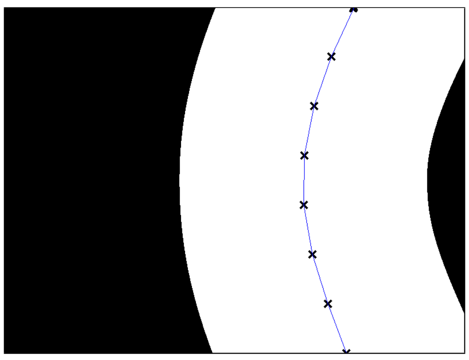

(b) Resulting interpolated line segments.

Figure 6.8: Example for an interpolated polynomial using Lagrange form. Left shows the points used for the base polynomials, right shows the new interpolated points.

Additional points from the interpolation can be used in future works if different controllers, for example the Model Predictive Controller, should be tried out to correct the vehicles orientation.
The described method to obtain new points for missing segments is a heuristic that does not guarantee an optimal controlling process of the vehicle. For an optimized controlling process of the vehicle a histogram-based algorithm was developed. This is described in the next section.

6.2.4 Histogram-based Path Planning

While the proof of concept could already be achieved by using the simple middle line algorithm (see 6.1), the group's advanced goal to use a neural network to segment the drivable street on a pixel basis required a different approach.
The neural network does not define borders of the drivable area as a polynomial but classifies every pixel instead. Therefore, the resulting segmentation sometimes has fuzzy

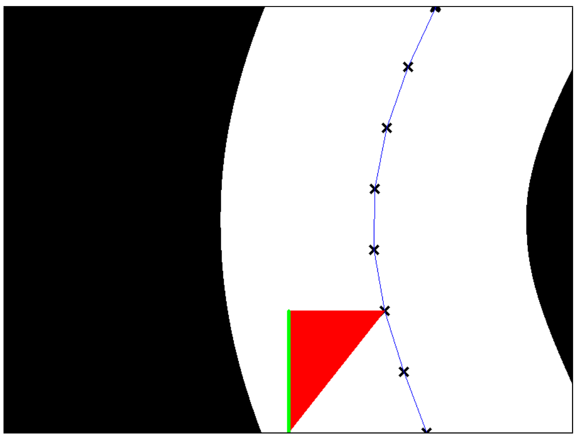


Figure 6.9: New error angle to use for the controller

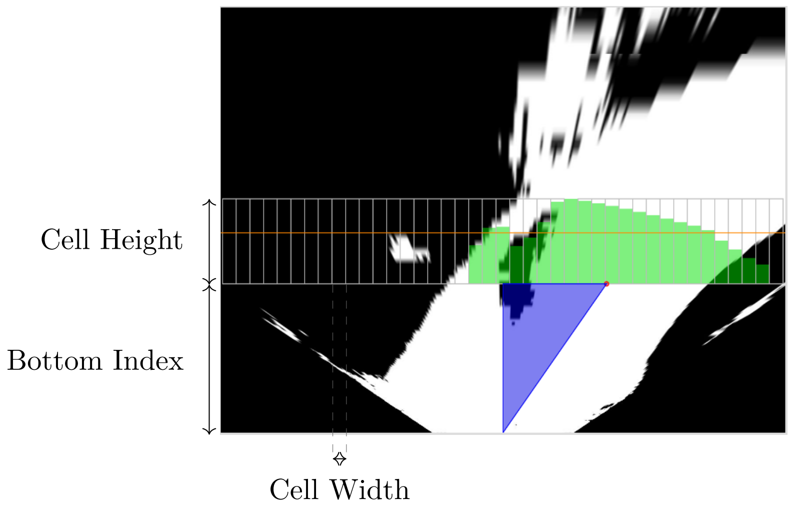


Figure 6.10: Visualization of the effect of hyper parameters *bottom index*, *cell height* and *cell width*.

borders or contained holes (see Figure 5.2e and 5.17a for comparison). These small errors make calculating a suitable middle line for the naive path planning hard if not impossible. Thus, a new path planning approach based on the obstacle detection technique described in [Siciliano and Khatib, 2016, Chap. 35.9.2] was implemented. The technique divides the field of view in evenly sized sections and assigns a value to each section based on the amount of sensed obstacles inside that part of the field of view. This yields to a histogram where each bin represents an angular region of the field of view and every bins value corresponds to the clearance in that direction. Choosing the bin with the lowest value leads to the direction the car is least likely to collide with an obstacle.
To use this idea for the task of lane keeping the meaning of the histogram values is inverted. A bin is assigned a value based on the drivable area it contains. Choosing a cell with the highest value therefore yields to the direction with the most drivable space. This technique does not depend on clear borders for calculating a destination point and thus can be used with the new segmentation. The whole approach is described in detail as follows.
Similarly to the middle line algorithm described in subsection 6.2.1 his algorithm receives a binary image of the street from an aerial perspective as input, where white and black pixels represent drivable and non-drivable areas respectively. Given three hyper parameters a region of interest and the size for each cell are defined in the input image. The available parameters are:

- *bottom index* - the bottom index for each cell
- *cell height* - the height for each cell
- *cell width* - the width for each cell

Figure 6.10 shows the effect of each parameter. Given this setup each cell represents an angular range in front of the car.

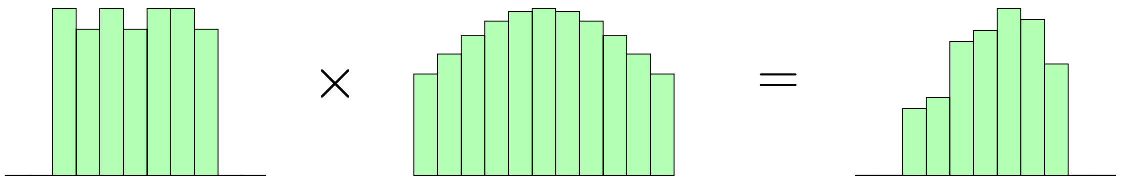


Figure 6.11: Visualization for weighting the histogram of drivable area with a gaussian.

In the next step, each cell is rated by the amount of drivable area it contains. For this all pixel values inside a cell are added and subsequently divided by the sum of all cells to get a relative rating. This leads to a histogram where each bin represents an angular section and the value of a bin the corresponding rating. Since pixels that are classified as drivable area are white (255) or non drivable area are black (0), cells with a higher rating contain more pixels that are classified as drivable area. Therefore, destination points chosen from the highest rated cell are the least likely to lead away from the road.
To give the car the tendency to drive forward the histogram is multiplied with the probability density function of a normal distribution. The mean for the distribution is given as the center of the histogram, while the variance can be set as another hyper parameter. Figure 6.11 gives a visual explanation of this calculation. Additionally, when the car has to make a turn, the mean of the normal distribution will be moved to the corresponding edge of the histogram, favoring cells in the turning direction. This is reset to default after a predefined time, which can be set via a hyper parameter.
Choosing a cell gives a rough direction in which the car can drive. To calculate an error angle a concrete destination point has to be calculated. The bottom index is used as y-coordinate. To calculate the x-coordinate two different methods were implemented, which are explained in the following.

Interpolation of destination point The first method assumes the center of the cell with maximum value to be the target direction. This results in as many discrete destinations as there are cells in the histogram. Changes to the histogram maximum that naturally occur while driving therefore lead to jumps in the target calculation, which results in a jumpy steering behavior. To conquer this and provide smooth changes to the destination the values of the cells adjacent to the maximum are used to interpolate the destinations x-coordinate. First the minimum of both adjacent cells is identified and labeled as *min*. The respective other cell is therefore the in between cell and is labeled as *inb*. Based on the ratio between the values a interpolation coefficient

$$\alpha := \frac{inb - min}{max - min}$$

is calculated. This α is then used to enforce the same ratio between one half of a cell width and the interpolated x-coordinate. An explaining diagram for this method is shown in Figure 6.12.

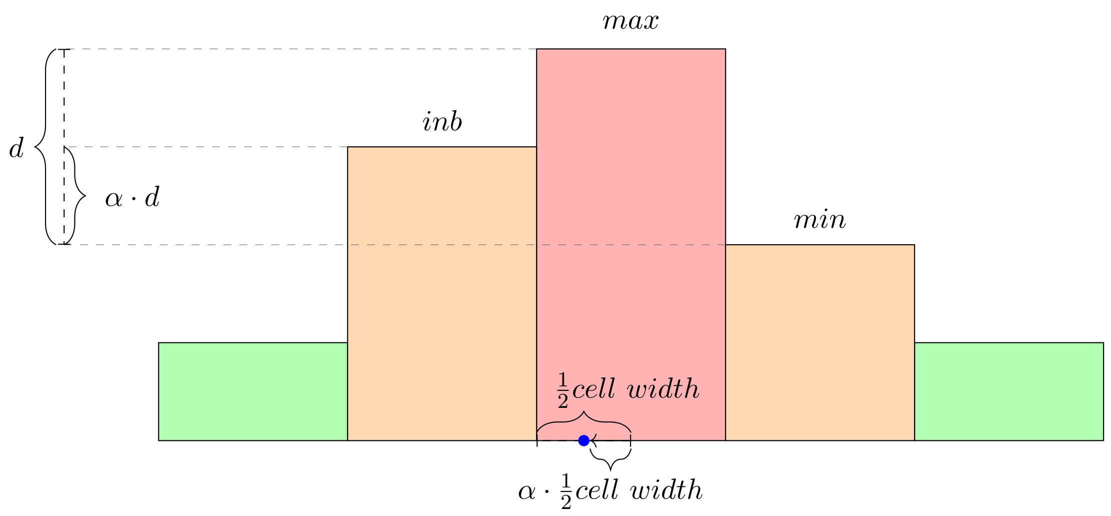


Figure 6.12: Visualization of the interpolation method to calculate the destination x-coordinate.

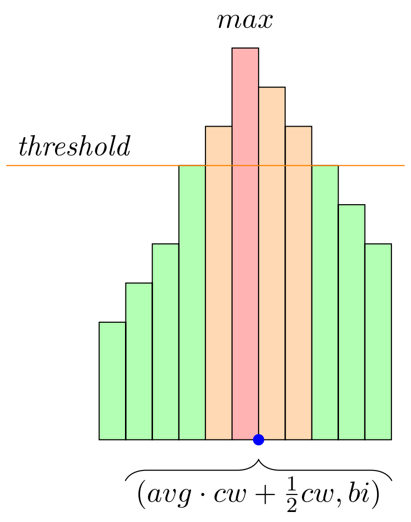


Figure 6.13: Visualization of the averaging method to calculate the destination x-coordinate. Significant cells, i.e., cells with values above the threshold, are colored orange. *avg* is the average of the indices of all significant cells; *cw* and *bi* are abbreviations for the earlier mentioned hyper parameters *cell width* and *bottom index*.

Averaging of destination point The second method differentiates between significant and insignificant cells. A significant cell is a cell whose value is higher than a given threshold, which is derived from a proportion of the maximum value. The proportion is set through a hyper parameter by the user. After that, the average index over all significant cells is taken and the x-coordinate is then calculated as

$$x := avg \cdot cell\ width + \frac{1}{2} cell\ width.$$

One half of a cell width is added to the coordinate to guarantee a centered destination in a balanced histogram. Figure 6.13 shows a diagram of this method.

As the segmentation is sometimes jittery, the histogram values and the directly derived destination point are jittery as well. To smooth out the steering the current destination point is interpolated with the previously calculated destination. This dampens changes to the destination and prevents sending jumpy changes to the steering controller.

6.3 Control the Vehicle

section 6.1 explains that the car (or rather its driving style) needs to be described as a dynamic closed-loop system where the controlled variable is the steering angle (also see Figure 6.2). Also, the *observer* and the *plant* were specified. In section 6.2, different approaches to determine the error angle were introduced. In this chapter, the focus is set on the *controller* of the dynamic closed-loop system.

Recall the following values:

- the *current* state, $y(t)$
- the *desired* state, $r(t)$
- the resulting *error*, $e(t) = r(t) - y(t)$
- the *control output*, $u(t)$

As mentioned in section 6.1, a controller is a mathematical heuristic that takes the error $e(t)$ as a parameter to compute a control output $u(t)$. As there are many different kinds of controllers, the focus was set on two prominent kinds of controllers, namely a *P-Controller* and a *PID-Controller*. The PID controller is an advanced controller, which consists of 3 components: the *proportional*-, the *integrative*- and the *derivative* gain. Each component computes a value based on $e(t)$, which will then be summed up to form the control output $u(t)$. Briefly speaking, the proportional gain influences the steering by a ratio of $e(t)$. The integrative component integrates the sum of all $u(\tau)$ with $0 \leq \tau < t$ that were computed before the current computation of $u(t)$. The derivative component 'predicts' future error terms, so that they can be treated beforehand. For each component there is a constant: K_P for the proportional gain, K_I for the integrative gain and K_D for the derivative gain. Let t be the time unit. Then the control output $u(t)$ is calculated as follows (also see Figure 6.14):

$$u(t) = K_P * e(t) + K_I * \int_0^t e(\tau) d\tau + K_D * \frac{de(t)}{dt} \tag{6.15}$$

Proper values for K_P, K_I and K_D have to be found to control the vehicle, as there exist many combinations of values leading to uncontrolled behavior of the vehicle. It is possible

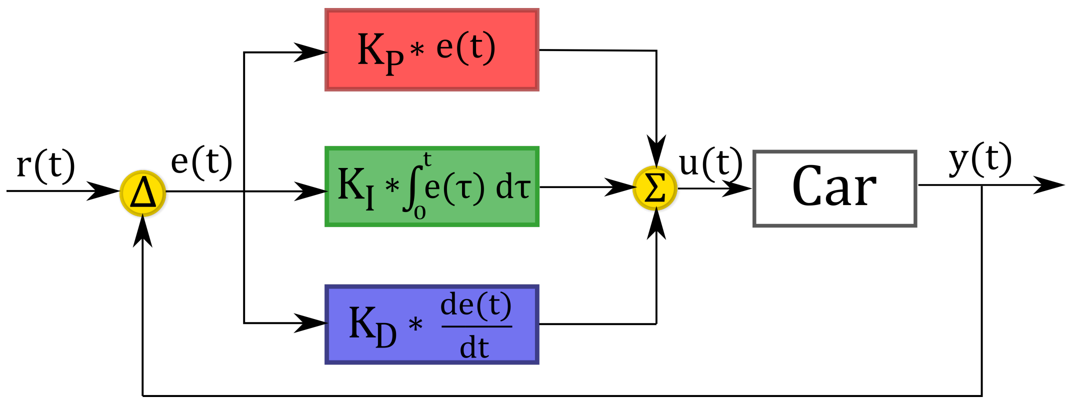


Figure 6.14: Closed loop system using a PID Controller

to choose $K_x = 0$ for some $x \in \{P, I, D\}$, to get a different type of controller.

Different approaches for the path planning were tested (see section 6.2), just like differently trained neural networks for the lane detection (see chapter 5). As the choice of the controller's constants depend strongly on the chosen path planning as well as the lane detection heuristic, using the same chosen PID-values led to different results within each scenario. In the following chapters, the results of different PID-constants are presented. These PID-constants were found either empirically or heuristically by making use of the mathematical state space systems Equation 6.12 and Equation 6.13 in the program *Octave*[1], especially using the package *control*[2], which contains the necessary functions for control theory.
However, since the heuristics evolved during the project, not every combination of PID-constants, lane detection algorithm and path planning heuristic was tested. The following chapters describe the progress chronologically with respect to the project.

[1]https://www.gnu.org/software/octave/
[2]https://octave.sourceforge.io/control/index.html

6.3.1 The Proportional Controller

Choosing $K_I = K_D = 0$ leads to a *proportional controller*, which multiplies the resulting error angle only with a proportional gain. The calculation will look like:

$$u(t) = K_P * e(t) \tag{6.16}$$

In the first attempt to let the VTD car drive autonomously, a proportional controller was used. The simple lane detection heuristic described in section 5.1 as well as the naive route planning heuristic described in subsection 6.2.1 were used. Several values for K_P were tested empirically. The first attempt was $K_P = 100$, which led to progressive oscillation. Trying different values emerged that the vehicle's proportional gain for the steering controller should depend on its velocity.

Using $K_P = 1.75$ enabled the car to drive most of the curves steadily in our VTD simulation with an approximate velocity of $22\frac{m}{s}$ (see Figure 6.15). The figure shows the step response of the state space model $\mathcal{M}_{VTD}$ with the given K_P as time passes. The setpoint is 1, thus the step response ideally has to approach 1 over time.

On the first glance, it might look like it is rather steady and approaching to 1, which indicates controlled behavior. However, having $K_P = 1.75$, the step response is actually diverging as the overshoot increases in each step. Since no severe oscillation was noticeable in the simulation, no further investigations were made.

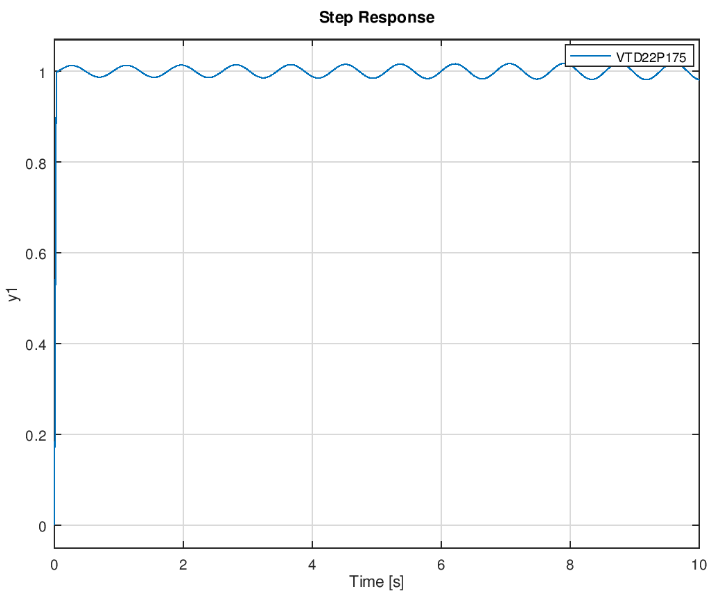


Figure 6.15: Model $\mathcal{M}_{VTD}$ (Equation 6.12), P-controlled with $K_P = (1.75, 0, 0)$ with constant velocitiy of 22 $\frac{m}{s}$ and setpoint 1. The explanation follows the following name pattern: 'VTD' (induces $\mathcal{M}_{VTD}$) + [*longitudal velocity*] + 'P' + [*P-constant with decimal point after the first digit from right to left*]

The vehicle would only leave the lane on peaky curves appearing near the horizon of the segmentation (see Figure 6.7a). This problem could be solved by changing the length of the desired path as described in section 6.2. For a controlled driving style it was premised that the vehicle started driving in the middle of the street. Also, there is no correction of lateral offset. To solve this problem prototypically, a middle line position error $e_{pos}(t)$ was added, so that the steering algorithm worked as follows:

$$u(t) = K_P * e(t) - e_{pos}(t) * 0.0005. \tag{6.17}$$

Using Equation 6.17, the car was able to drive steadily through peaky curves and get back to the center of the lane. However, the smaller the constant longitudinal velocity was, the more abrupt the movement induced by this controller became.

In the second attempt, the Advanced Lane Detection (see section 5.2) as well as the route planning heuristic in subsection 6.2.1 were used. Several values were empirically tested within the VTD simulation and values that seemed to cause a controlled driving style were additionally tested in Octave using the mathematical state space model Equation 6.12. For $K_P = 0.4$, the step response is shown in Figure 6.16.

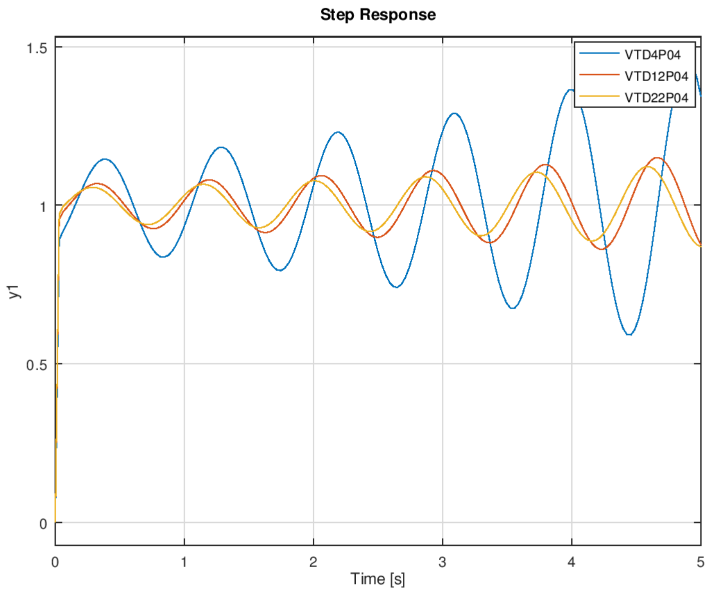


Figure 6.16: Model $\mathcal{M}_{\mathcal{VTD}}$ (Equation 6.12), P-controlled with $K_P = 0.4$ at constant velocities of 4, 12 and 22 $\frac{m}{s}$.

It is clear to see that for $K_P = 0.4$ the amplitude of the step response increases over time the slower the car drives. This is in contrast to what was observed empirically in VTD. The differences may have occurred because of inaccuracies of the state space models Equation 6.12 and Equation 6.13. Another possible cause for this mismatch is that the velocity in the empirical tests was not completely constant, so that the replication of this situation was not fully matching the state space model.

The histogram-based route planning heuristic (see subsection 6.2.4) as well as a neuronal network (see section 5.3) were implemented and used for the following test cases:

- $K_P = 0.4$, performed well
- $K_P = 0.5$, performed best
- $K_P = 0.25$, performed badly
- $K_P \geq 1$, performed okay

$K_P = 0.5$ led to the most promising results. Therefore, this value was tested with the state space model $\mathcal{M}_{Q2}$ (see Equation 6.13). Different constant speeds were tested and can be seen in Figure 6.17. Having lower velocities of 2 or 4 $\frac{m}{s}$, the overshoot initially is high (at a step response of 1.24, so 24%), whereas the setpoint is reached at about 0.15 seconds. At higher velocities, the overshoot is lesser, but the setpoint is reached after 0.4 to 0.6 seconds. However, having a very high velocity of 20 $\frac{m}{s}$, the the step response results in progressive oscillation, inducing an uncontrolled driving style. This may be happening because at very high velocities, even small control inputs can lead to changes that cause an even greater error $e(t)$.

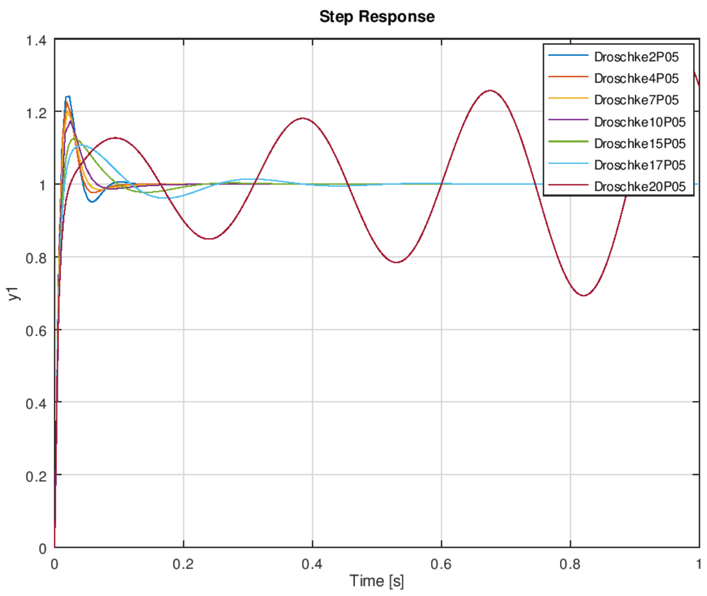


Figure 6.17: Model $\mathcal{M}_{Q2}$ (Equation 6.13), P-controlled with $K_P = 0.5$ with constant velocities of 2, 4, 7, 10, 15, 17 and 20 $\frac{m}{s}$. The explanation follows the following name pattern: 'Droschke' (induces $\mathcal{M}_{Q2}$) + [*longitudinal velocity*] + 'P' + [*P-constant with decimal point after the first digit from right to left*]

6.3.2 The PID Controller

Choosing a non-zero value for K_P, K_I and K_D leads to a PID-Controller described at the end of section 6.3. Equation 6.15 describes the calculation of the control output.

By following the the scheme in Table 6.5, the most promising result found using the state space model $\mathcal{M}_{\mathcal{VTD}}$ (see Equation 6.12) was $K_{PID} = (0.0005, 0.00001, 0.0007)$ at a constant velocity of $12\frac{m}{s}$. The histogram-based route planning heuristic (see subsection 6.2.4) as well as a neuronal network (see section 5.3) were used. The step response can be seen in Figure 6.18.

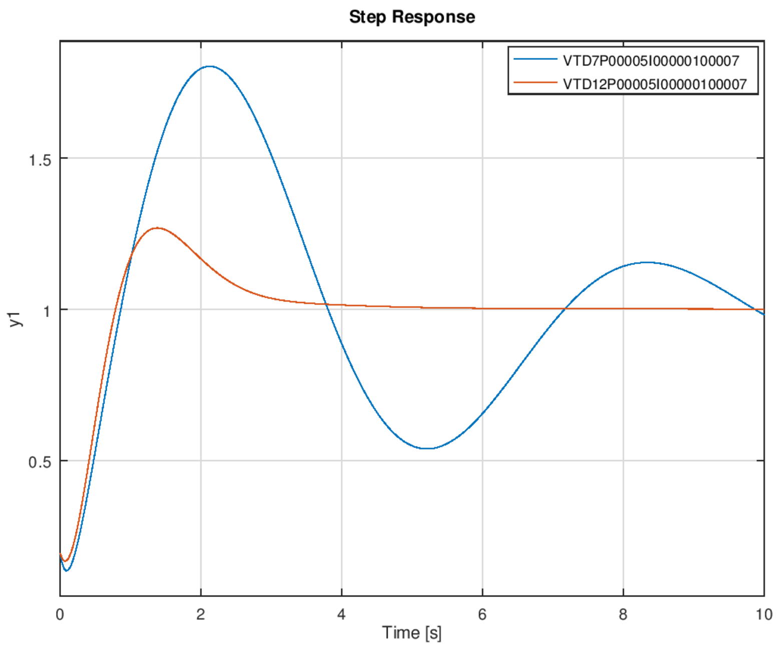


Figure 6.18: Model $\mathcal{M}_{VTD}$, PID-controlled with $K_{PID} = (0.0005, 0.00001, 0.0007)$ with constant velocities of 7 and 12 $\frac{m}{s}$. The explanation follows the following name pattern: 'VTD' (induces $\mathcal{M}_{VTD}$) + [*longitudinal velocity*] + 'P' + [*P-constant with decimal point after the first digit from right to left*] + 'I' + [*I-constant with decimal point after the first digit from right to left*] + 'D' + [*D-constant with decimal point after the first digit from right to left*]

However, these values do not cause a controlled steering behavior if the velocity is below $7\frac{m}{s}$. The resulting step response of using these values with a constant velocity of 5 $\frac{m}{s}$ can be seen in Figure 6.19.

Testing these values in VTD showed that the car did not steer, which led to uncontrolled behavior. This might have happened because of the low PID-values that cause the control output to be so low that it becomes unnoticeable.
By empirically testing PID-values on the actual car, the most promising values found were $K_{PID} = (0.4, 0.005, 0.05)$, although the values $K_{PID} = (0.5, 0, 0)$, shown in Figure 6.17,

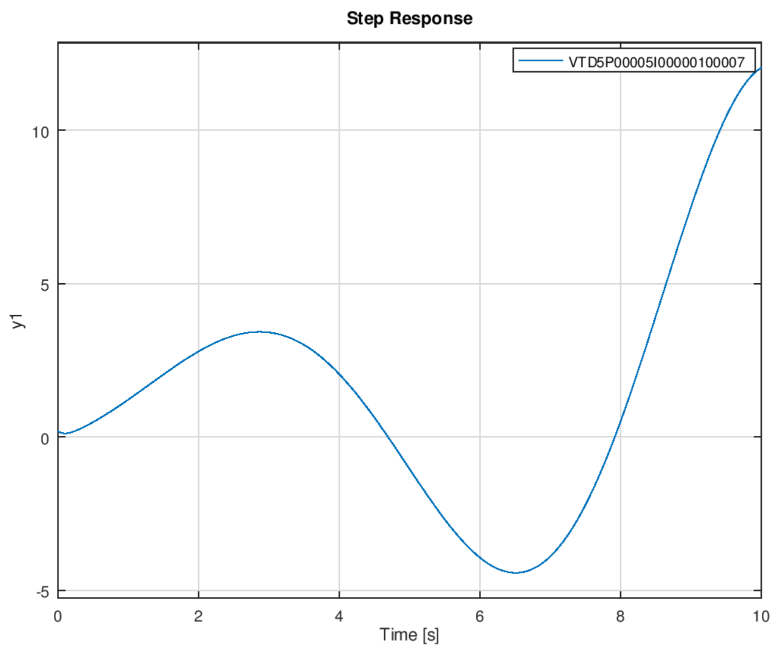


Figure 6.19: Model $\mathcal{M}_{VTD}$, PID-controlled with $K_{PID} = (0.0005, 0.00001, 0.0007)$ with constant velocity of 5 $\frac{m}{s}$

showed a better performance when steering in practical experiments. The corresponding step response of the state space model $\mathcal{M}_{Q2}$ using these values can be seen in Figure 6.20.

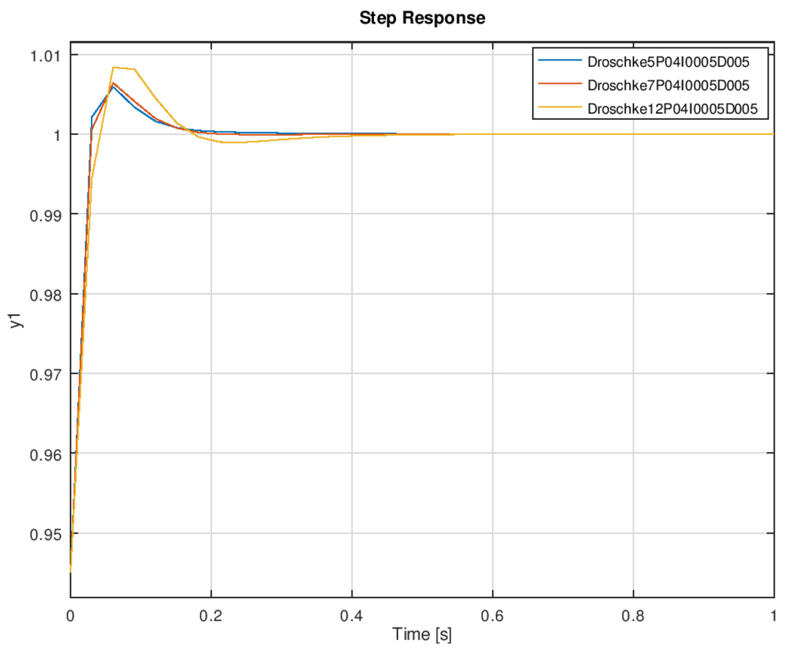


Figure 6.20: Model $\mathcal{M}_{Q2}$, PID-controlled with $K_{PID} = (0.4, 0.005, 0.05)$ with constant velocities of 5, 7 and 12 $\frac{m}{s}$. Note that the time span is between 0 and 1.

Analogous to Figure 6.17, the PID-constants cause uncontrolled behavior at high velocity (here: $20\frac{m}{s}$) in the state space model, as shown in Figure 6.21.

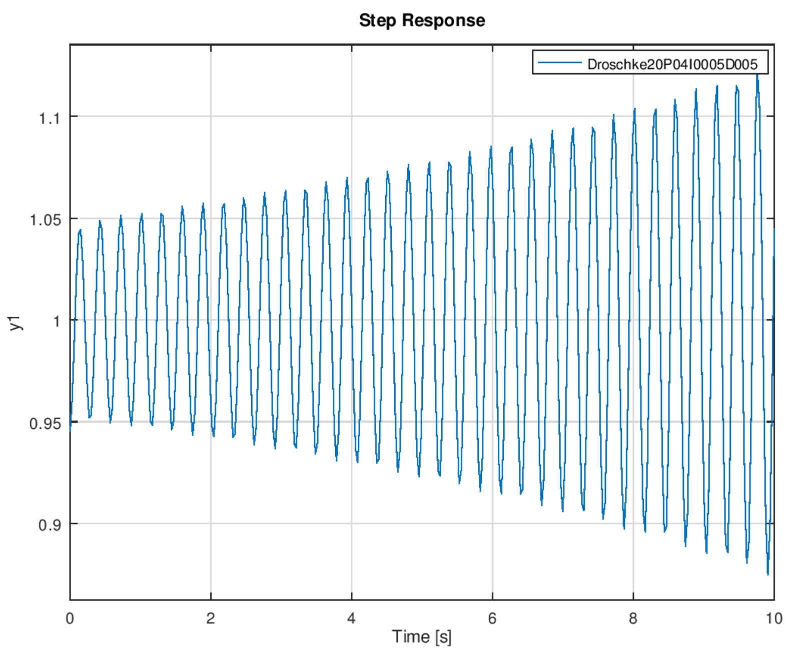


Figure 6.21: Model $\mathcal{M}_{VTD}$, PID-controlled with $K_{PID} = (0.4, 0.005, 0.05)$ with constant velocity of 20 $\frac{m}{s}$

6.4 Evaluation of Lanekeeping

6.4.1 Fulfilling Requirements

GEN-CON-1 requires autonomous steering by the car. This is achieved by the Universal Controller which receives an error angle based on the destination point that is computed by the path planning filters.

GEN-CON-3 to GEN-CON-5 and GEN-CON-6 to GEN-CON-8 require proper handling of crossings and T-crossings respectively. This is implemented by moving the mean of the multiplied normal probability density function. When multiplied to the edge of the histogram, this edge is therefore favored for the computation of the destination point. Thus the car plans a path to steer in the desired direction.

GEN-CON-9, GEN-CON-10 and GEN-CON-12 define the car's behavior when detecting a turn sign and the upcoming actions to take at an intersection. When the path planning filter receives a turn signal the mean is moved to the histograms edge as mentioned previously, thus taking a turn. After three seconds, the path planning filter resets the mean position back to the center, which lets the car drive straightforward again.

Driving around an obstacle is required by GEN-CON-13. This is achieved by multiplying the array of ultrasonic sensors to the histogram. A sensed obstacle will create small weights in the array and thus extenuate cells in the histogram that contain the obstacle. Therefore, a destination point not targeting the obstacle is chosen.

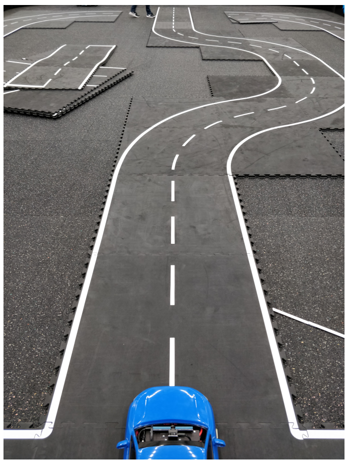
6.4.2 Motivation and Setup

To test assumptions about the car's behavior, mainly two different setups were used. Firstly, a small configuration (max. $6m \cdot 5m$) of printed street mats was laid out in the corridor to quickly test minor aspects (getting used to the interaction with controls and drive train, basic lane keeping and integration tests of manual control, see subsection 6.4.3). This setup is fast to build and remove and therefore convenient for spontaneous testing.

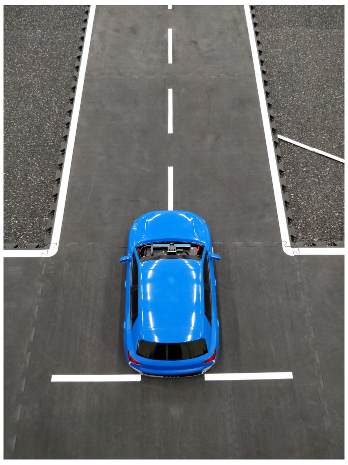
For more in-depth experiments, a bigger configuration was built in a research hall (see Figure 6.23 and Figure 3.20). This allowed for tests with higher velocities, greater distances and in general a more spacious world environment for the sensors.

To evaluate the quality of each configuration, the car drove a simple course autonomously and logged every calculated error angle about every fifth of a second. The logs were evaluated in Table 6.8 afterwards using the metrics described in Table 6.7. Starting with a default configuration that seemed to cause a good driving style of the car (for instance, staying on straight streets as well as sharp bends while driving at low velocities or reaching the steering setpoint quickly without much disturbance), certain parameter values were configured while the other parameter values were fixed. To enable comparability between the experiments, every experiment had to be performed with minimum changes to the surrounding environment. Therefore, a simple course and a protocol for all experiments is defined. The course is shown in Figure 6.22a. It starts with three straight mats, followed by a right-hand S-bend, one straight mat, a left-hand S-bend and finally three straight mats. The starting straight section is used to give the car enough space to accelerate until the target speed is reached. Both S-bends are used to test the behavior of the path planning and the controller. The remaining three straight mats are used to check if and how fast the car is able to stabilize itself after following a bend. Each execution starts at the same position. Figure 6.22b shows the starting position for an execution where the car drives in the center of the road. For experiments where the car had to follow a specific side of the road, the car was positioned respectively. To minimize variances in the starting position the axle position was marked with small pieces of tape, as shown in Figure 6.22c.

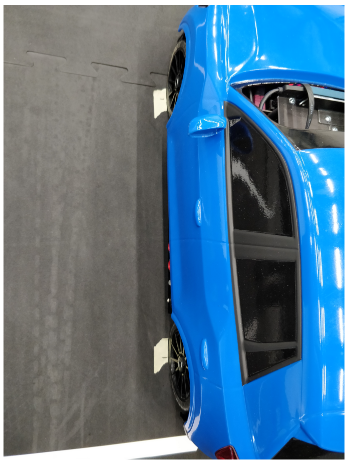
Every experiment follows the same protocol. The car is first aligned given the markers. Triggering the autonomous mode starts the experiment and the logging of error angles. When the car reaches the end of the course, the autonomous mode is paused and the car is aligned again at the end of the course to pass it once again this time from end to beginning.



(a) Experiment route consisting of three straight mats, a right-hand S-bend, one straight mat, a left-hand S-bend and finally again three straight mats.



(b) Starting position for every execution. After passing the course once the car is positioned the same way at the end again to pass it the other way around.



(c) Markers for the tire positions. To minimize the differences between executions the axle positions are also marked with a small stroke on the tape.

Figure 6.22: Setup for the experiments. During every experiment the car has to pass the same course twice, always starting from the same position.

6.4.3 Manual Controls

The car has two operation modes. There is a RC mode that enables the user to control the vehicle by using a common remote control from model motor sport. In this mode, one can accelerate and steer the car manually and none of the sensors data or software decisions have an impact on the car's behavior. Vice versa, the autonomous mode disables all of the RC functions and the car is operated only via ADTF. This means all orders from setting lights to operating the actuators originate in ADTF.

To be able to steer and accelerate but most importantly brake remotely, a XBox remote controller was used. It proved to be highly useful to set the car back on track in cases where it failed to follow the track. The operator is able to control basic operations of the car by hand while ADTF can still override the commands and trigger important actions like emergency braking in the autonomous mode. Also, for testing steering related features, the possibility to control the car's speed helped immensely in finding the right velocity when the track led to corners or junctions. In this case the software to test would take over the steering and the operator using the XBox remote controller would be in control of the longitudinal acceleration.

6.4.4 Experiments

Experiment: Finding Controller Parameter Values with State Space Model

As explained in subsection 6.3.2 the PID-constants in Figure 6.18 were found heuristically using the scheme in Table 6.5. Initially, the values were found using a constant velocity of

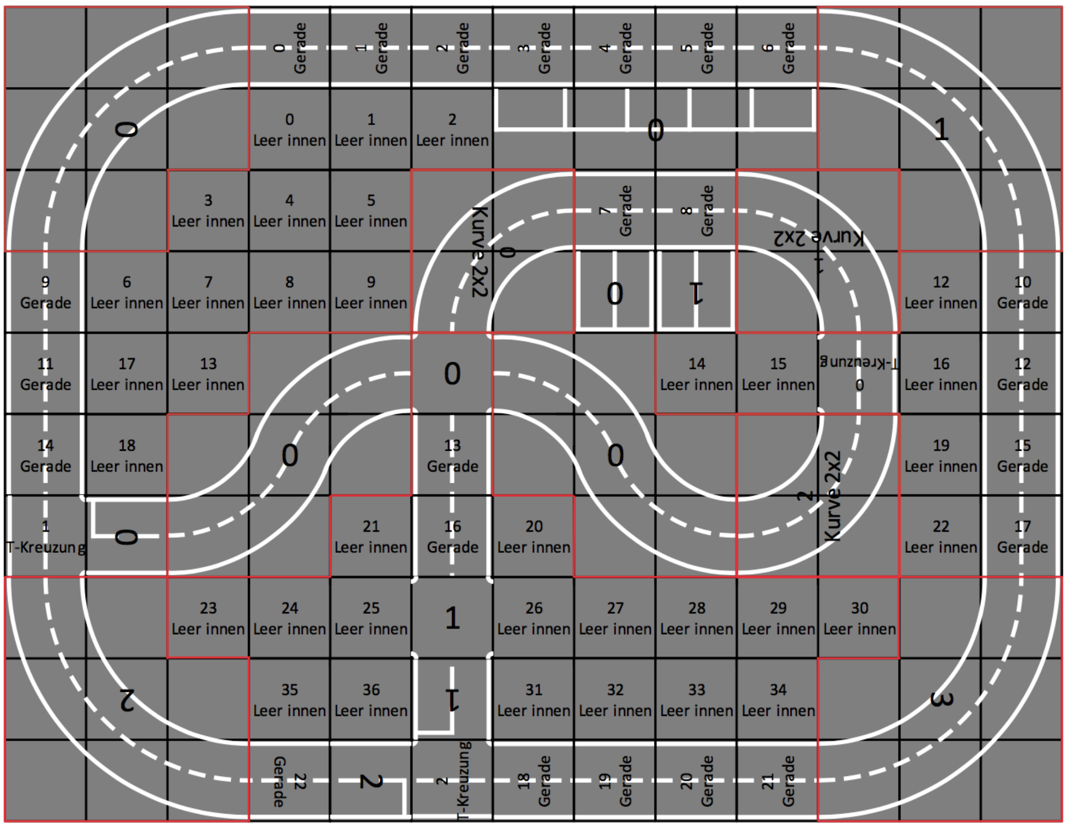


Figure 6.23: To have a variety of use cases, this track was designed with tighter and shallower curves, different junctions, parking spaces and straights

$20\frac{m}{s}$, however re-checking the values using the velocity $12\frac{m}{s}$ appeared to behave similar. In Table 6.6 a table of tested parameter values is shown using the state space system Equation 6.12. The table shows 5 columns, one for each the P-, I- and D-constant as well as one column that shows either the required time for the step response to converge to 1 or ∞ if it diverges. In the fifth column, individual comments were made for further explanation of the step response's behavior. The terms *disturbance* and *overshoot* are used here often. The first refers to the frequency or rather the oscillation of the step response, the latter refers to its amplitude.

Result: Testing these values led to two main observations: first a lurching vehicle, changing the steering rapidly and secondly a far to conservative steering, not being able to stay on the street. The conclusion is that both the path planning systems and the controller are regulating against the measured error and adding their actions. The car was then tested with values that seemed to be the least wrong to the naked eye. Then, also based on intuition, the values were increased or decreased with constant check of the affected steering performance. As described in section 6.3, it turned out that $K_P = 0.4$ resulted

	RT	**OS**	**ST**	**SSE**	**S**
Increasing K_P	Decrease	Increase	Small Increase	Decrease	Degrade
Increasing K_I	Small Decrease	Increase	Increase	Large Decrease	Degrade
Increasing K_D	Small Decrease	Decrease	Decrease	Minor Change	Improve

Table 6.5: Effects of independent P, I and D tuning on closed-loop response. See [Li et al., 2006, p. 33, Table 1]. RT = Rise Time, OS = Overshoot, ST = Settling Time, SSE = Steady-State Error, S = Stability. For example, while K_I and K_D are fixed, increasing K_P alone can decrease rise time, increase overshoot, slightly increase settling time, decrease the steady-state error and decrease stability margins.

K_P	K_I	K_D	**Converging?**	**Comment**
1	0	0	∞	divergence visible at 20sec
5	0	0	∞	divergence visible at 400sec
5	0	1	∞	divergence visible at 100sec
5	0	2.5	>1000sec	very high disturbance, low overshoot
5	0	3	>1000sec	very high disturbance, low overshoot
5	0	1.4	∞	
5	0	10	>1000sec	extreme disturbance, very low overshoot
0.5	0	0	∞	divergence visible at 10sec
0.3	0	0.9	>1000s	
0.3	0	0.15	400sec	
0.002	0	0.15	<200sec	
0.002	0	0.075	100sec	
0.0002	0	0.01	20sec	overshoot 20 percent
0.0006	0	0.002	6sec	overshoot about 25 percent, clear convergance
0.0006	0.0063	0.002	∞	
0.001	0.00001	0.0007	4-6sec	clear convergance
0.0005	0.00001	0.0007	4sec	clear convergance, overshoot 20 percent

Table 6.6: Tested PID-constants using scheme in Table 6.5

in the VTD car being able to drive on a straight lane and on streets with slight curves. This observation is in contrast to the results in Table 6.6.

Experiment: Finding Controller Parameter Values in Practice

The goal of testing the PID-constants using the state space models described in Equation 6.12 and Equation 6.13 was to reduce the general risk of using wrong PID-constants that could lead to the car getting damaged in an accident. However, because certain required parameters for the state space model of the car were missing, they had to be approximated. This inaccuracy may have been the reason that the allegedly fitting PID-constants within the state space models actually did not work in practice (see subsection 6.3.1 and subsection 6.3.2). To solve this problem, the PID-constants were tested empirically within VTD. The PID-constants on the model-car were also tested empirically, but under strict supervision, at low velocities only and with the use of an emergency brake system. It seemed like the VTD car and the real car behaved approximately similar using the same PID-constants.

Result: Using the trial-and-error-method showed that a P-Controller with $K_P = 0.7$ was the overall best solution to the naked eye for the model car with a constant longitudinal velocity of $0.4\frac{m}{s}$. In the experiments 1 and 21 - 24 in Table 6.8, $K_P \in \{0.4, 0.5, 0.7, 1, 1.5\}$ were tested. Seemingly, $K_P = 1.5$ leads to the best driving style according to the data. This is true if the car's longitudinal velocity is very low. If the velocity is greater, the quality of the driving style in general decreases as the steering becomes too strong which leads to uncontrolled driving behavior. Using $K_P = 0.7$ enables the use of greater longitudinal velocities and therefore offers more flexibility. Also, it causes no significantly worse result than $K_P = 1.5$ at a longitudinal velocity of $0.4\frac{m}{s}$.

Experiment: Ultrasonic Distances for Histogram

For a proof of concept of a prototype obstacle avoidance the results of the histogram were enhanced with distance data from the ultrasonic sensors. This experiment used the ultrasonic values of all five front sensors to weight the corresponding cells of the histogram (see subsection 6.2.4) such that the cells with small measured distances are less likely to be chosen. The car is keeping the road based on the histogram. On the right lane a car stands motionless, serving as an obstacle. A car is used as an obstacle, because cars are usually labeled drivable.

Result: In some cases (curves, corners) the Ego vehicle just crashes into the parked car. When the Ego vehicle dodges the parked one, velocity plays a critical role for when the Ego car turns back on the right lane: If the longitudal velocity is too low, the car turns back too early and crashes into the parking vehicle. On a straight road with the right speed the autonomous car switches to the left lane just before the parked car and turns back after it.

Experiment: Changing the Bottom Index

Experiments 3 to 6 show the impact of the cell's bottom index on the autonomous driving. The value controls the distance of a calculated destination point from the car. Lower values result in closer destination points. Every value from 0.2 to 0.5 in steps of 0.1 was tested.

Result: The evaluation shows the best results in the mean and integral with a value of 0.2. These seemingly good results are tainted, as the bird's-eye view has blind triangles in the bottom corners that result from stretching the segmented image. Thus, the path planning cannot compute destination points on the edges and therefore results in bigger error angles. In the real world, values around 0.2 result in following the middle line accurately on tracks with wide bends. However, because of the mentioned blind triangles in the bottom corners, indices of this magnitude make it hard to navigate on tracks with sharp corners. The absolute maximum and absolute integral are best when setting the parameter to 0.5, because calculating a more foresighted destination point gives the car more time to react on bends in the track. Values above 0.5 result in the car steering to early in bends as it chooses a destination point too far away to navigate, thus sometimes cutting the road edges.

Experiment: Changing the Cell Height

Experiments 7 to 10 evaluate the effect of different cell heights on the path planning. The value controls the size of the area that is used for the analysis of the drivable road to calculate the histogram. Every value from 0.1 to 0.5 in steps of 0.1 were tested.
Result: Using a value of 0.1 leads to the best results in every metric except for the absolute integral. Higher values cause the car to steer early, similar to the experiment of changing the bottom index. The value of choice is still 0.2 as this is slightly more foresighted, which helps steering in sharp bends.

Experiment: Changing the Standard Deviation

Experiments 11 to 15 show the effect of different values for the standard deviation of the multiplied normal probability density function. The tested values were 5 and 10 to 30 in steps of 10. Additionally, experiment 15 shows the effect of not multiplying a normal probability density function to the histogram.
Result: Optimal results for all metrics except the absolute integral were achieved with a value of 10 or 20, which reflects the observations in the real world and the decision to use a value of 15. Experiment 5 failed due to the car driving off the road. Low values lead to a narrow histogram. Therefore, more cells on the edge are not considered in the averaging as they generally are below the necessary threshold. As a result, the car drives really close to the roads edges before it countersteers.

Experiment: Changing the Averaging Threshold

Experiments 16 to 20 evaluate the impact of the averaging threshold. Values from 0.2 to 1 in steps of 0.2 were tested for this parameter.
Result: Based on the chosen metrics, high values around 0.9 lead to the best results with the exception of the mean difference. A reason for this is that a lower threshold leads to more considered cells in the averaging which makes the calculated destination point more reliable. Higher values on the other hand lead to fewer considered cells, making the calculated destination point prone to frequent changes. Observations in the real world show that for high values (above 0.7) the car steers not enough in sharp bends due to the ignoring of cells at the edge that are weighted by the normal probablity density function.

Path Planning	**NC**	Number of Cells
	BI	Bottom Index
	H	Height
	CO	Cell Offset
	GM	Gauss Mult
	SD	Standard Deviation
	AT	Average Threshold
	IT	IgnThresh
Metrics	**M**	The mean of all logged error angles. A value close to zero indicates a balanced driving behavior, not tending to certain lateral direction.
	AMax	The greatest absolute error angle that occurred. A high value indicates a meaningful steering maneuver during the experiment.
	TI	The trapezoidal integral. As the logged error angles are discrete, this integral approximates a continuous integral. The same route is driven forwards and backwards, so ideally, this should be close to zero as the driving style should be equivalent.
	ATI	The absolute trapezoidal integral, so all integrals are treated as positives. A small value indicates that the logged error angles during the experiment also were small.
	MD	The mean difference between all consecutive steps. A value close to zero indicates a smooth driving style.

Table 6.7: Description of the used parameters and metrics in Table 6.8

	Path Planning								Control	Metrics					
Nr	NC	BI	H	CO	GM	SD	AT	IT	K_P	M	AMax	TI	ATI	MD	Plot
1	41	0.35	0.2	0	1	15	0.6	0.4	0.7	$2.23e^{-2}$	$6.57e^{-1}$	1.57	20.287	$-1.51e^{-3}$	Figure B.2
2	41	0.25	0.2	0	1	10	0.6	0.4	0.7	$-6.57e^{-5}$	$6.44e^{-1}$	$6.75e^{-3}$	19.718	$-3.14e^{-4}$	Figure B.1
3	41	0.2	0.2	0	1	15	0.6	0.4	0.7	$\mathbf{-4.58e^{-3}}$	$8.15e^{-1}$	$\mathbf{-2.99e^{-1}}$	21.017	$-2.40e^{-3}$	Figure B.3
4	41	0.3	0.2	0	1	15	0.6	0.4	0.7	$-7.16e^{-3}$	$6.83e^{-1}$	$-4.46e^{-1}$	21.454	$\mathbf{-8.25e^{-4}}$	Figure B.4
5	41	0.4	0.2	0	1	15	0.6	0.4	0.7	$3.43e^{-2}$	$6.53e^{-1}$	1.97	20.039	$-1.18e^{-3}$	Figure B.5
6	41	0.5	0.2	0	1	15	0.6	0.4	0.7	$4.08e^{-2}$	$\mathbf{6.27e^{-1}}$	2.54	**17.461**	$-1.94e^{-3}$	Figure B.6
7	41	0.35	0.1	0	1	15	0.6	0.4	0.7	$\mathbf{1.50e^{-2}}$	$\mathbf{6.36e^{-1}}$	**1.07**	21.733	$\mathbf{-1.11e^{-3}}$	Figure B.7
8	41	0.35	0.3	0	1	15	0.6	0.4	0.7	$3.21e^{-2}$	$6.91e^{-1}$	2.01	20.611	$-1.93e^{-3}$	Figure B.8
9	41	0.35	0.4	0	1	15	0.6	0.4	0.7	$2.58e^{-2}$	$7.38e^{-1}$	1.94	19.705	$-1.95e^{-3}$	Figure B.9
10	41	0.35	0.5	0	1	15	0.6	0.4	0.7	$4.63e^{-2}$	$7.49e^{-1}$	2.87	**18.792**	$-2.08e^{-3}$	Figure B.10
11	41	0.35	0.2	0	1	5	0.6	0.4	0.7	$2.97e^{-2}$	$9.46e^{-1}$	1.96	19.424	$-2.22e^{-3}$	Figure B.11
12	41	0.35	0.2	0	1	10	0.6	0.4	0.7	$5.15e^{-2}$	$\mathbf{6.65e^{-1}}$	3.39	21.469	$-1.94e^{-3}$	Figure B.12
13	41	0.35	0.2	0	1	20	0.6	0.4	0.7	$\mathbf{4.70e^{-2}}$	$7.46e^{-1}$	**2.94**	22.657	$\mathbf{-4.08e^{-4}}$	Figure B.13
14	41	0.35	0.2	0	1	30	0.6	0.4	0.7	$5.23e^{-2}$	$7.28e^{-1}$	3.04	20.912	$-1.43e^{-3}$	Figure B.14
15	41	0.35	0.2	0	-	-	0.6	0.4	0.7	$5.76e^{-2}$	$7.49e^{-1}$	3.71	**20.493**	$-1.51e^{-3}$	Figure B.15
16	41	0.35	0.2	0	1	15	0.2	0.4	0.7	$4.86e^{-2}$	$6.72e^{-1}$	2.84	20.101	$\mathbf{-1.35e^{-4}}$	Figure B.16
17	41	0.35	0.2	0	1	15	0.4	0.4	0.7	$6.04e^{-2}$	1.05	3.59	21.266	$2.39e^{-4}$	Figure B.22
18	41	0.35	0.2	0	1	15	0.8	0.4	0.7	$5.10e^{-2}$	$7.20e^{-1}$	2.81	20.138	$9.38e^{-4}$	Figure B.17
19	41	0.35	0.2	0	1	15	0.9	0.4	0.7	$\mathbf{3.84e^{-2}}$	$\mathbf{6.13e^{-1}}$	**2.79**	**17.854**	$6.74e^{-4}$	Figure B.18
20	41	0.35	0.2	0	1	15	1	0.4	0.7	$5.26e^{-2}$	$7.02e^{-1}$	3.35	16.699	$-1.82e^{-3}$	Figure B.19
21	41	0.35	0.2	0	1	15	0.6	0.4	0.5	$6.01e^{-1}$	$8.48e^{-1}$	4.05	29.655	$-1.84e^{-4}$	Figure B.20
22	41	0.35	0.2	0	1	15	0.6	0.4	0.4	$1.13e^{-1}$	1.05	7.60	35.354	$-2.80e^{-3}$	Figure B.21
23	41	0.35	0.2	0	1	15	0.6	0.4	1	$2.70e^{-2}$	$5.76e^{-1}$	1.70	14.251	$3.71e^{-4}$	Figure B.23
24	41	0.35	0.2	0	1	15	0.6	0.4	1.5	$\mathbf{1.84e^{-2}}$	$\mathbf{5.21e^{-1}}$	**1.15**	**9.388**	$\mathbf{-1.66e^{-5}}$	Figure B.24

Table 6.8: NC = Number of Cells, BI = BotIdx, H = Height, CO = CellOffset, GM = GaussMult, SD = StdDev, AT = AvgThresh, IT = IgnThresh, BS = BuffSize, T = target, M = Mean, AMax = Absolute Maximum, TI = Trapezoidal Integral, ATI = Absolute TI, MD = MeanDiff. The bold faced values represent the best results of the local set of experiments. In the red highlighted experiments the car drove off the street, so these values should not be considered.

Chapter 7

Implementation and Documentation

Gitlab was chosen to be the documentation and implementation platform. It provided a git repository, wiki and an issue board at once. Seven projects were created by the group:

- *main* The main code, wiki and the issues are placed here
- *doc* Documentation and related artifacts
- *docker-droschke* The place where the docker images are hosted
- *accplus-adtf* Code from another project group with a related topic as reference
- *image-labelling-tool* Code for a labeling tool the group used for labeling images to train the neural network
- *ref* Reference project containing example source code from VTD and ADTF
- *keras-to-tensorflow* Code for transforming Keras models to Tensorflow models. Forked from github.

Project Main The main project has a protected master branch. From the master branch we created feature branches that usually refer to a certain issue on the board that was handled by a single team member. If helpful, the developer added the issue number to a commit so that the reviewer immediately knew what was developed. Only code that was reviewed and approved by the other team member was merged into the master branch.
On a high level, the repository contains the directories `description`, `projects` and `src`. The directory `projects` contains a version of an ADTF project and the directory `de-scription` holds the media type description. There is also the directory `src` containing the code for the neural network in the sub-directory `Droschken-Net`, the code for the label extraction in the sub-directory `labelutils` and the code for the visualization of the path planning algorithm in the directory `web-visualization` including the visualization's

code. Most recently, a small reporting application `error-logging` was added, containing a python script that logs debugging values from the path planning.
Development usually led to individual filters of which each had their own folder, which can be found in the directory `filters`. For each filter, there is a single directory with the name `PG618_FilterName`.
Since there was no further documentation about the filters than comments in the code, a short README file explaining each filter, its input, output parameters, its properties and a description of what it is doing were introduced. This might look like:

> **Local Path Planning Filter**
>
> **Inputs**
>
> Birdseyeview Video : cVideoPin
>
> The video that is received from Lane Detection containing BEV (may include abstract world model later)
>
> Current Speed : tSpeedData
>
> Current Speed of the vehicle received from VTDAbstractionFilter
>
> **Outputs**
>
> Local Path : tPath
>
> 10 Points of the calculated path
>
> **Properties**
>
> Minimum Distance : tFloat32
>
> The minimum distance that is used for path planning
>
> **What is the purpose of the filter?**
>
> This filter takes the bev to calculate a path to drive through the abstract world model.

All README files are attached in section A.4 of the appendix and can be reviewed on the repository in the most up-to-date version.
For the attribute types there is a structs file that is the basis for all communication between the filters using ADTF media types.
For important standards and agreements, wiki pages were created and spread between team members. Also, for the weekly review with the stakeholders, there was an individual wiki page containing all information about tasks we worked on, problems that occurred and what would be done next. This was used as the basis for presentations, for documenting results from the meeting, and for other stakeholders to keep up-to-date when not being able to join the appointments.

Project Doc This project keeps the slides from the seminar phase that happened prior the project. Also, it contains information about the architecture, as well as the code for this report.

A continuous integration pipeline was created for this, so that the group both knows that the LaTeX files are compilable and there is an up-to-date PDF file whenever it is needed.

Project Docker-Droschke In the beginning of the project, there were problems for team members to be able to use the development environment the group agreed on due to issues with their operating systems and incompatibilities. Therefore, the project group created a Docker image that enabled any machine to run the proposed IDE and compile the code successfully using the libraries provided by ADTF.
Due to the success and good experience with Docker, the team decided to run ADTF on the vehicle using Docker, too. This way, ADTF could be run on other machines in the office which made parallel development easier and faster.
Besides the images, there is a documentation about how to use them in the repository as a README file, too.

Project Accplus-Adtf Another project group already implemented an adaptive cruise control software that is able to keep a configurable distance to the next vehicle given a maximum speed. During their development, they implemented an XBox controller, too, that we wanted to adjust and make usable for us as a safety mechanism in physical simulations. The project contains the project groups code they provided to us.
Additional information was spread using Slack as instant messenger and a Google Calendar for appointments.

Chapter 8

Summary

Enrolling in one of the many offered project groups is part of the (Applied) Computer Science Masters degree at TU Dortmund University. Project groups are therefore constrained to a one year period and while that may at first seem very long, it is in fact not. Getting a quite heterogeneous team (heterogeneous because previous knowledge and expertise differed vastly) up to speed took more time than anticipated. Time was spent on gaining domain specific knowledge (e.g. control theory, path planning), on learning to work and integrate code with already existing frameworks (e.g. ADTF, Tensorflow), on learning programming tools (e.g. git, docker) and programming languages (e.g. C++) and last but not least on organizing a rather big group of twelve students. So, the group had to acquire a lot of knowledge and learn to coordinate the teamwork first.

In the following the project group's results are stated to summarize twelve months of development of an autonomous driving system.

During the initial phase, the first goals to achieve were the goals for the proof of concept stated in section 1.2.1. The proof of concept contains the goal 1 *The autonomous car must be able to drive straight*, goal 2 *The autonomous car must be able to keep a lane on a straight road* and goal 3 *If there is a static obstacle in front of the autonomous car the autonomous car must be able to stop prior to collision.* These proof of concept goals were fulfilled approximately by the end of the first half of the project. The process to fulfill these goals is mostly described in section 5.1, section 6.2.1, and section 4.2.2.

The second half was used to extend and improve the system's functionalities to realize at least the minimal goals stated in subsection 1.2.2.

With the fulfillment of the proof of concept goal 3 the minimal goal 4 *The autonomous car must prevent a collision with static obstacles by stopping* was already achieved sufficiently to enable further work on the other minimal goals. So, requirements SAF-1 and SAF-6 belonging to goal 4 are met in that the autonomous vehicle stops in front of an obstacle, as long as the obstacle is standing in front of the vehicle.

The minimal goal 1 *The autonomous car must be able to follow a previously unknown route inside the virtual simulation* and goal 2 *The autonomous car must be able to follow a previously unknown route in the physical simulation* required a lot of work as not only the feature-rich *drive autonomously* functionality but also the *drive manually* functionality and the *emergency brake* path had to be developed to ensure the system's safety, as it is explained in chapter 4. The successful accomplishment of these both goals included the fulfillment of most of the general requirements from section 2.1 and safety requirements from section 2.2. As the project group aimed to meet the goals by using neural networks as it is stated in section 5.3, it was necessary not only to design a suitable convolutional neural network architecture but also to develop an advanced path planning algorithm which enables the autonomous vehicle to turn on various intersections. Especially the detection of signs meeting requirements GEN-SEM-5 to GEN-SEM-7 demanded more work from the group than initially expected. An additional difficulty was the different behavior of the vehicle in the virtual and the physical simulation, which is stated in section 3.6.
In summary, it can be stated that the minimal goals 1 and 2 were achieved sufficiently as the developed autonomous driving system is able to follow different previously unknown routes inside the virtual simulation as well as inside the physical simulation. Indeed, the vehicle does not always stay reliably on the street during the whole route and does not always recognize the correct street sign at the first try, but in most cases it is able to get back on the street and to react correctly to street signs.
The last minimal goal 3 *The autonomous car must set light signals when turning and braking* and the associated requirements GEN-SNX-10 to GEN-SNX-12 were achieved successfully with the implementation of the `PG618_Lights` filter as it is described in section 4.1.2.

While working on the achievement of the minimal goals, several of the optional goals defined in subsection 1.2.3 were additionally accomplished.
The vehicle can drive around static obstacles under certain limitations. This is possible because of the usage of an histogram-based path planning algorithm which is described in section 6.2.4. However, the emergency brake must be deactivated for this purpose as this function of the autonomous vehicle prevents its ability to drive around static obstacles. So, a trade-off between the different goals of the project group has to be made.
Furthermore, the optional goal *The vehicle is driving smoothly* was achieved for the most part as suitable controller values for this goal were found during the evaluation described in section 6.4.
Also, the optional goal *The vehicle creates a live stream giving an overview about its current status* was fulfilled during the development. As it is briefly explained in section 4.1.2 most of the requirements concerning visualization stated in section 2.1 were met with the development of a visualization showing the current status of the vehicle for testing and presentation purposes.

Another optional goal achieved by the project group is the goal *Usage of neural nets in other environments than the one it was trained with.* As explained in section 5.3.4, the group tried out the usage of a neural network trained with data from the virtual simulation in the physical simulation, yet this usage turned out to be unfit for autonomous driving on the used street tiles.

During development the project group had to experience that several trade-offs concerning the goals and the requirements had to be made. The improvement of one function often led to reduction of quality of another function, e.g. the creation of a live stream giving a full overview about its current status made an adaptation of requirement GEN-SNX-13 necessary, as it was not possible for the system to handle new camera information arriving too quickly.

Overall, the project group considers its goals as being sufficiently achieved in the given context and at the same time sees many opportunities to expand and improve the system.

Chapter 9

Outlook

While many visions and ideas were formulated in the first weeks of the course, in the second half year it became apparent that the group would not be able to fulfill much more than the minimal goals. But at the same time the team gained enough knowledge in all areas and is able to outline next steps and future possibilities.
One big area the group was not able to pay attention to is (automated) testing: Unit tests combined with continuous integration would not only enable finding programming errors earlier, it would also increase the general stability of the system. Furthermore, the project would profit from integration tests: An example would be the (automated) testing of the segmentation filter, with a pretrained network model and fixed test data to test not only correctness but also inference speed. While testing the system as a whole might seem hard, there are parts that still can be tested and thus integrating system tests is possible. One could think about a predefined course inside VTD and the usage of VTD capabilities to detect if the vehicle passes a specific point on the road for testing and benchmarking lane keeping abilities of the system.

The previous point addresses another area that needs to be improved: It would be of great usage if more and better metrics could be defined that measure the quality of driving. Those metrics would consider e.g. lane keeping abilities and obstacle avoidance. The absence of those metrics makes testing especially in real world scenarios harder and less scientific because it is not possible to quantify the performance of the system.

After working with ADTF filters, the project group now recommends an even further refinement of the filter chain to improve the non-functional requirements like reusability and maintainability. Looking back at the experience with ADTF 2 the group can conclude that programming filters for ADTF 2 requires to write a big amount of boilerplate code. While an effort to reduce the required boilerplate was started by introducing own filter and pin classes, working with the ADTF 2 SDK was still frustrating for other reasons: An inconvenient API and not very modern C++ are some. Many of those shortcomings are addressed in the newest version of ADTF 3 and upgrading seems a very logical step.

ADTF 3 also contains much better debugging and data flow visualization capabilities.

After outlining these general improvement ideas the project group also wants to address some possible functional improvements of the implemented system: To improve safety, more sophisticated logic for emergency braking with better obstacle detection using more input sensors (e.g. depth camera) or a combination thereof can be thought of. Running different emergency brakes independently might also be beneficial.
The motion control area also leaves a big room for improvements: It would be helpful to obtain the unknown state space parameters to improve the current controllers. Furthermore, trying different controller models provides even more potential.
The current path planning algorithm is mostly based on heuristics and operates on a very local understanding of the situation. That could be vastly improved by making it aware of the global situation, e.g. adding localization to the system. That can be done in various ways, e.g. by including a map of the scenario that enables the estimation of the position or even by a GPS-like system. Such changes would directly improve the turning behavior of the car, since the system could estimate its relative position to an interception.
But all those components would already be improved by increasing the semantic understanding of the world: The localization problem could also be solved by using SLAM (Simultaneous Localization and Mapping) where a map is built of an unknown scenario while at the same time keeping track of the agents location in this map. But also in the current architecture there is room for improvements regarding semantic understanding: One could think of a more automated and replicable system to run and evaluate network models on different training data sets, e.g. by using CI tools like GitLab pipelines. Many other network architectures remain unexplored and integrating obstacle and sign type detection into the segmentation network are logical next steps. Thinking about hybrid solutions between neural networks and other reasoning systems might improve some of those objectives even more.

A completely different approach that would change most of the system's architecture is to train the neural network to take over path planning or even controlling. Given the complexity of this task, it might be the starting point for a new project group.

Appendix A

Additional Information

A.1 Tutorial: How to Create a Track in VTD

This guide is meant as a starting point to quickly create your first track in the road designer. It has a tutorial / documentation that you can reach by clicking `Info>ROD Tutorial`, which is much better than any of the other VTD documentation. Using this guide should still give you a much quicker head start.

A.1.1 How to Create a Route in VTD

1. Open VTD via `/opt/VTD.2.1/bin/vtdStart.sh`

2. Open the road designer from VTD via `Tools>RoadDesigner`

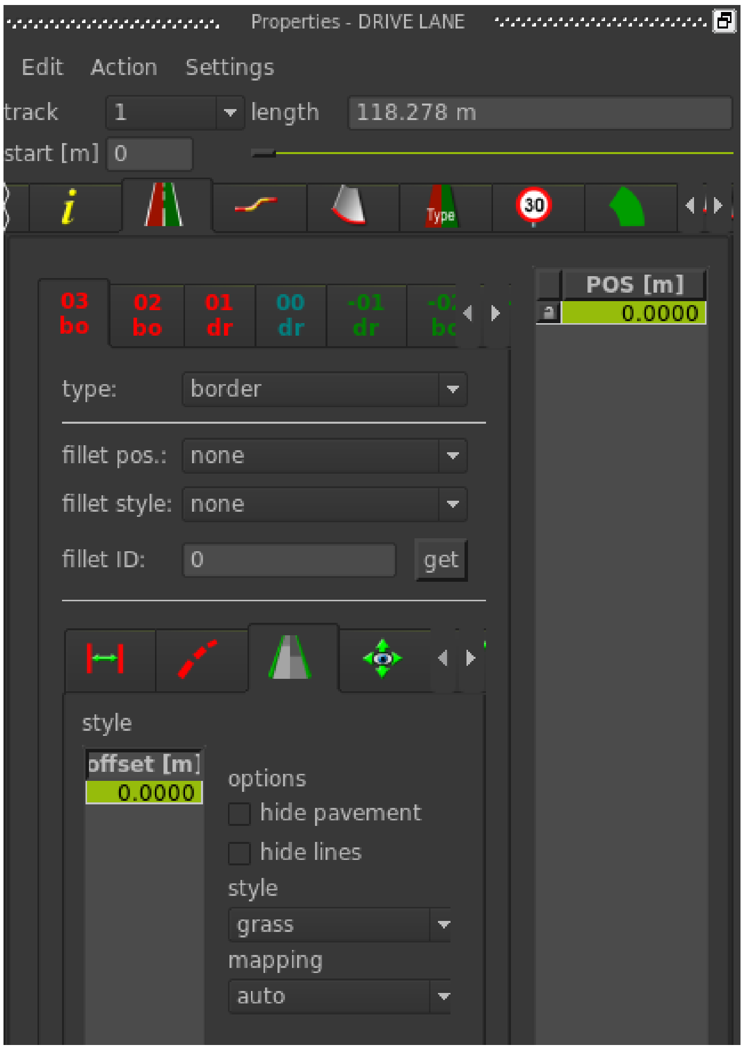
Preparation

Create a new project with its own project folder. The single routes created will be stored as `overlays`, but you still seem to need a project within which those are created.

Turn on the grid by clicking the grid icon in the top bar. This will give you a sense of distance and size, since each square on the grid is one square meter.

Creating a Simple Track

To start drawing your track, switch to `line mode` by clicking . After drawing a line, switch to `pick pointer` and choose the line you just drew. Click on the left toolbar to turn the selected line into a track.
Select the line again to open the `track properties window`:


Properties - DRIVE LANE

Edit Action Settings

track 1 length 118.278 m

start [m] 0

POS [m]
0.0000

type: border

fillet pos.: none

fillet style: none

fillet ID: 0 get

style

offset [m]
0.0000

options
hide pavement
hide lines

style
grass

mapping
auto

In that window, switch to the `Drive Lane` tab and click `Edit>New` to create a new lane. Now you can use a macro to create a track in a pre-defined style, e.g. a country road with two lanes, via Action>Macro.

To view the lane you just created, you have to save your `overlay` first. Then you click the `generate database` icon . This will open a simulation of the road that you can navigate.

Decoration

To make the surrounding landscape look nice, we can use style macros. In the track properties window, choose the `style` tab . Then open the style macro window via `Action>Execute macro`. Choose a style, click `Execute macro` and you should have nice surroundings for your roads.

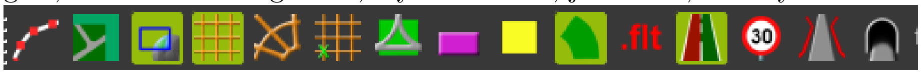
A.1.2 Miscellaneous

Road Architecture

Roads are constructed out of different lanes. Each lane has a width, a style (e.g. standard (for a normal road), or grass), road marks (with their own style). The middle road marks are created as their own lane. To have a wide grass environment, you should choose the outermost lanes (probably labeled 03 and -03 in the lane properties window), set the width high, set `style` on `grass` and uncheck `hide pavement` to make it visible.

Viewing More Info in the Editor

Toggle the buttons in the top bar to have the editor show more visual details, like the grid, different existing lanes, style elements, junctions, or very useful sticky points.



Adding 3D Objects

To add 3D obejct anywhere on the map, do: `Rightclick>3D Model>Add single`
You could, for example, place a chicken on the road.
If you want to repeat 3D objects along the lane, you have to click the respective object first. Then you enlarge the properties window appearing on the bottom right of the editor window. If you click `repeat` and choose `aligned on path`, you can choose a distance within which the object is repeated along the track.

Creating Curves

Choose `draw spline` on the second icon in the left bar, instead of `line mode`.
Alternatively: Draw two lines (they may intersect), choose both with the pick pointer while holding shift, then click the `link with curve` symbol (third/fourth in left bar). This one makes really smooth curves.

Adding Roadmarks

In the track properties window, choose the lane you want to add roadmarks to (likely the middle 00 lane or one of the border lanes), then choose the `road marks` tab , right click under the `offset[m]`, choose `new`. `Offset` indicates the offset in along the direction of the road, `z offset` indicates the offset from the middle of the road.

Creating a Junction

Open the documentation by `Info>ROD Tutorial` and open chapter 10 on PDF-page 60. It is very well-written and copying it here does not make sense.

Getting proper road marks on the junction seems problematic. If you create a standard T-crossing, there will be six ways that a car could take into and out of the crossing. Therefore, you will have six, partially overlapping, tracks. Therefore, if you add road marks to each track, they will overlap. You can try to fix this by manually adjusting the `offset[m]` property of one of those until it looks the least awful.
Attention: All roads to be used in the junctions need to have been drawn as coming from the outside into the junction area. Else there are problems with right- and left-hand-side traffic. Alternative, you might be able to use the `revert track` feature to change the direction of an existing track.

Unexpected Behavior

Sometimes ROD does weird things. E.g. not being able to select tracks anymore. Turn it off and on again.

A.1.3 Importing Constructed Roads into VTD

In ROD you only create the static environment that your car is supposed to drive in. You still have to create a scenario in the Scenario Editor, which adds the car and its control logic. The important file containing the information about your created road is a `.xodr` file (OpenDRIVE format). In the scenario editor, you add it under properties>layout file. In case you can not find the `.xodr` file, try `Generate>OpenDRIVE` in ROD, or `File>Export>OpenDRIVE` to see where the file might already be stored.

A.2 Hardware and Sensors of the Model Car

This model car is designed and built to participate in the "Audi Autonomous Driving Cup"(AADC). This is by now, a yearly challenge by a large German car manufacturer, where interested groups of students and researchers can compete against each other by mastering a course with one of these autonomous model cars.
The 2018 driving tasks included dealing with traffic, reverse parking (both parallel and orthogonally to the road), turning, reacting to pedestrians on a zebra crossing, dealing with a tight road due to a construction side, overtaking a broke-down car and so on. The competition demonstrates how capable such a model car can be with regard to practical approaches on (scaled down) real world problems.
To offer the competing teams many possible approaches the car has very different sensor types available. This is also to make teams able to compensate weaknesses in the perception of one sensor with other sensor data from a different kind of sensor. This project group used some but not all of these features.

A.2.1 Sensors

As said, the model car has numerous different sensor types. They can be basic: Voltmeters check the voltage of both the driving battery and the sensors/PC battery, this is to monitor the car's ability to operate with enough current.

Or they might be more complex. And especially when they are used to scan the environment they deserve a closer look on their own:

Ultrasonic Sensors

Our car is fitted with HC-SR04 ultrasonic sensors; one behind each front wheel, looking sideways (pointing 9 and 3 o'clock); three in the back bumper, pointing 7, 6 and 5 o'clock and also a front bumper with five ultrasonic sensors. To get a better coverage of the scene in front of the car, they are oriented as such that they can detect objects in a wide field of view and have as few blind spots as possible. In the bumpers the more sideways pointing sensors are tilted up a bit to allow for better detection of close objects and collision avoidance in manoeuvres like lane changing.

The sensor's manual says the object to detect should have a surface area of at least 0.5m^2 to have best performance. Objects can be detected in a range of 20mm up to 4000mm with a resolution of 3mm (It might be that these are values for the best case scenarios). The sensor operates at 40kHz and in a 30 degrees field of view (fov), however the effectual usable fov is stated to be 15 degrees uss. Unfortunately however we found the front ultrasonic sensors to be very noisy in their signal quality. See section 3.1.5 for further details on how we tried to handle these problems.

The ultrasonic sensors were useful for an indipendent Emergency brake algorithm, as they give fast results and are oriented in a way that they see areas, the cameras cannot cover.

At the front ultrasonic sensors are good for measuring distances, but for having visual data cameras are needed.

Basler Camera

For the main view upfront there is the Basler daA1280-54uc camera. This industrial camera with its small form factor is typically used for in process monitoring and quality checks but also appropriate for many other fixed focus use cases. Its wide angle (130 degrees) and small distortion make it suitable for object-, road sign- and lane detection. The global shutter makes it suited for (fast-) moving applications.

The resolution is 1280 x 960 pixels (1.2 MP) at 45 frames per second, but was scaled down to 480 x 360; this is to make the Segmentation real-time compatible 5. Over the lens the focus can be set manually.

Rear Camera

The rear camera, a Delock 96368, has with 2592 x 1944 pixels a much higher resolution. It can output up to 30fps. The (horizontal) field of view (fov) is with 80 degrees a lot smaller than the Basler's fov. With this in mind it is clear that its overall capabilities are very different from those of the front cameras. Furthermore, it is mounted short on top of the rear bumper under the bootlid, so that the point of view (pov) is far down.

Nevertheless its use can help with parking situations, traffic detection and reversing manoeuvres in general. As this was not part of the scope of this work, the rear camera was not used here.

3D Camera

Also for future works, the Intel R200 depth camera might prove useful. It delivers four video streams looking in the direction of travel. One (up to) full HD RGB stream with 15, 30 or 60fps. Also two infrared streams and most important a depth stream with various settings of resolution and framerate. This depth stream can, with its distance values per pixel, aid object detection. Combining the results of that with this cameras RGB stream it could make for example road sign detection faster and easier. One possibly neat feature of this camera is to have the RGB- and depth streams have (roughly) the same resolution, point of view (pov). Data of one stream could help analysing the other one, visualisation of important parts in the depth picture could be shown in the RGB picture and so on.

The depth camera works by having an infrared projector laying out a grid over the fov of the two infrared cameras, this grid shows up differently in the infrared cameras, depending on the scene. As the IR cameras are a known distance apart, the module can calculate an object-to-camera-distance for each pixel.

By implementing all algorithms only based on the Basler camera, the project was more manageable and less focused on data migration. To get to better and more reliable solutions, future developers should look into migrating the camera data. Another example of sensors measuring in the same thematic field are the speedometer and accelerometer.

Speedometer

For speed calculation the car has encoder sensors fitted. A slotted disc in each rear wheel mount rides in the encoders, which then can determine the absolute speed and rotational direction of the wheel. In this work the speed was simply used for updating the speed control. However by comparing the measured speed with the other sensor data and the set target speed, future works might be able to learn about delays in the control chain or maybe draw conclusions about the road surface (i.e. how slippery it might be). Other sensor data, in this case, comes from the accelerometer.

Position and Motion Tracking Sensor

The MPU-9250 Motion Tracking Device is a 3-axis angular rate sensor (gyroscope), which can detect changes in its orientation, a 3-axis accelerometer, which can measure the actual accelerations in all three directions and a 3-axis magnetic compass, which can orientate itself with respect to the magnetic field of the earth.
For future work this can be relevant in a situation, where the wheels lose traction, so the real speed and accelerations of the car can determined (for example in an emergency braking situation or in the case the car has to dodge a fast moving danger). The compass and gyroscope can help with uncertainties about the car's localisation and/or orientation (for example in a turning action to aid unsatisfying camera data).
Should future project groups decide to incorporate a sophisticated world model, for example with mapping the environment, this sensor will provide important aid.

A.2.2 Actuators

For steering the carhas the Absima ACS1615SG Combat Series servo motor, it operates at 6V and can pull with a force up to 150N.
The drive system is based on a Hacker SKALAR 10 21.5 brushless motor from model racing. The power electronics is dealt with by a Robitronic Speedstar brushless speed controller.
The two components enable future projects to add exceptional behavior like fast accelerating or rapid cornering if necessary.
The power train relies on a 5200Ah, 7.4V battery, which should power the drive train for at least two hours. The fuses for the drive components blows at 20A to save the batteries and the motor with its electrical controller from overload.

A.3 VTD Test Routes

Route name	**Description**	**Challenges**
Race Track (in Figure 3.18)	Big "race"-like roundcourse with big curves	- No distractions on/next to road - Simple track to test lane following algorithm
Reagenzglas	Roundcourse with differently sided curves	- Minor distractions next to road (trees) - Track to test ability to steer in curves - Used to long-time test lane keeping ability
Rural	Rural track with big junctions and signs	- Some distractions next to road (trees, small town) - Track to test turning on junctions and following signs
Town (in Figure 3.18)	Small city track with junctions and a lot of signs	- Lots of distractions on and next to road (houses, lights, potholes) - Track to test how the segmentation deals with distractions - Lots of signs next to the road to test the sign recognition
Evaluation	Replica of the physical evaluation track	- Black streets with black surroundings like the physical course - Track to estimate behaviour of the car in the real world scenario

A.4 Filter READMEs

`Bird's Eye View`

Inputs:

- Video : cVideoPin

Outputs

- BirdsEyeVideo : cVideoPin
  Output format is the same as the input format.

Properties:

- scale_horizontal : float
  Proportion of the input image's width used to perspective transform; 1 = full image width; Minimum value: 0
- scale_vertical : float
  Proportion of the input image's height used to perspective transform; 0.25 = bottom quarter of the image; Minimum value: 0; Maximum value: 0.49

Purpose:

Perspective transforms a trapezoid region of the input image to produce a top-down perspective on that region.

`Car Sensing`

Inputs:

- RGBCam : ImageFormat::getCarRealsenseColor()
  (Outputs a camera image)

- DepthCam : ImageFormat::getCarDepth()
  Outputs a depth image.
- UltraSonic : tUltrasonicStruct
  Outputs a struct of ultra sonic sensor data (value and timestamp per sensor).
- Speed : tSpeedData
  Outputs the current speed of the car. Calculted by prefixed ADTF filter 'AADC Converter Wheels', since car doesn't provide it directly.

Outputs:

- Camera : ImageFormat::getCarRealsenseColor()
- DepthCamera : ImageFormat::getCarDepth()
- UltrasonicSensors : tUltrasonicData
- Speed : tSpeedData

Purpose:
This filter is the interface between the car on one side, and the perception and control group on the other side. It ensures that the other groups receive consistent data and can concentrate on their specific tasks. The VTDAbstraction filter offers the same interface, but sources its data from VTD. The images being output by the car sensors and the simulation have different sizes and formats. Therefore we convert the VTD image data to the car image format (both color and depth). Since we want to drive the car as well as possible, this direction seems to make the most sense. It means we don't up- or downscale the car sensor data, so our performance when using the car should be better than if we had done so. Since this filter receives car data already, it does not perform any transformation.

`Channel Extractor`
Inputs:

- Color Video : cVideoPin

Outputs:

- One-channel-Video : cVideoPin

Properties:

- Channel : int

Purpose:
The filter gets, as an input, an RGB (three channel) video and, according to what channel was specified using the property, sends out an greyscale video of the specified channel. As an example: If channel == 0, then the Red channel will be send out. This is useful because the segmented image, which is also an RGB image, contains the segments of street, signs and background in the respective R, G and B channels.

`Control Consolidation`

Inputs:

- SpeedController : tSpeedController
  Speed command from the autonomous controller
- SteeringController : tSteeringController
  Steering command from the autonomous controller
- XboxSpeed : tSpeedController
  Speed command from the Xbox controller
- XboxSteering : tSteeringController
  Steering command from the Xbox controller

Outputs:

- Speed : tSpeedController (Consolidated speed command)
- Steering : tSteeringController (Consolidated steering command)

Purpose:
The filter consolidates the control signals of the Xbox controller and the autonomous driving controller. The Xbox controller is always given precedence, enabling manual emergency interventions. As soon as we have used the Xbox controller once, all signals by the autonomous system will be ignored.

`Correction Angle`

Inputs:

- Vanilla Video : cVideoPin
- Trapezoid Video : cVideoPin
- Speed Data : tSpeedData

Outputs:

- Output Video : cVideoPin
- Control Data : tControlData

Purpose:
This filter was intended to be the filter that is delivered for the control part in the vehicle. By using the new architecture it is now deprecated and will only stay until all the code was moved to the new architecture. Originally the filter would take the trapVideo to calculate a middleLine, overlay it on the vanillaVideo to forward outputVideo for operators view

and uses the calculated middleLine to compute the correction angle that is used to get the vehicle back on track.

`Emergency Brake`

Inputs:

- Speed Input : tSpeedData
- Ultrasonic Input : tUltrasonicData

Outputs:

- Needs Emergency Stop : tJuryEmergencyStop

Purpose:
This filter uses speed and ultrasonic information to compute, if emergency stop must be applied. The algorithm is very simple and may raise emergency to early.

`Emergency Tester`

Inputs:

- Display There is a window in the ADTF screen where the operator can select whether an emergency stop should be raised in the running system or not.

Outputs:

- Needs Emergency Stop : tJuryEmergencyStop

Purpose:
This filter is used to test the behavior of the vehicle on sudden shut-down by emergency brake functionality. The shut-down can be executed by the operator in the UI.

`Error Angle Computation`

Inputs:

- Path : tPath
  This is the path that is used for error angle computation.

Outputs:

- Angle : tAngle
  The angle that is computed.

Purpose:
Calculate the error angle that results from the calculated path. The error angle can be used by a controller filter to set the steering of the vehicle.

`Histogram Based Path Planning`

Inputs:

- Birdseyeview Video : cVideoPin
  The video that is received from Lane Detection containing BEV (may include abstract world model later).

- Direction : tDirection
  Struct that contains a turning intention. Possible values are 'Left', 'Right' or 'NoSign'. Based on the input the path is planned favoring the corresponding side of the histogram.

- USS : tUSSStruct
  Array of ultrasonic sensor values. These values are multiplied to the histogram to achieve planning a path around obstacles sensed via ultrasonic sensors.

Outputs:

- Destination Point : tPoint
  The Destination determined by the cell containing the most non-zero pixels. The x-coordinate is given by the center of the cell, the y-coordinate is given by the Bottom Index set via the property.

Properties:

- Number of Cells per Side:
  The number off cells the histogram has to have on each side. If set to n, the total number of cells will be $2n + 1$. Twice for each side and one for the center.

- Proportional Cell Width:
  The width each cell has to have. The value is interpreted as a proportional width to the image width. Ignored if "Calculate Cell Width" is set to true.

- Calculate Cell Width:
  Toggles weather the cell width is calculated automatically based on the number of cells and the image width or "Proportional Cell Width" should be used.

- Cell Bottom Index:
  Factor to calculate the bottom row of the cells. The bottom is calculated proportionally to the input image height.

- Cell Height:
  Factor to calculate the height of the cells. The height is calculated proportionally to the input image height.

- Standard Deviation:
  Standard Deviation for the Gaussian PDF that will be multiplied to the histogram, the mean defaults to the center cell, and can be shifted with the "Cell Offset" parameter.
- Cell Offset:
  Shifts the mean of the Gaussian PDF over the histogram by n cells. Negative Values shift the mean to the left, positive values to the right. This achieves driving offset to the center. Only use this with "Shift Method" set to "Averaging".
- Gauss Factor:
  Factor to multiply on the Gauss before multiplying it to the histogram. Can be used to compensate to small values.
- Relative Offset:
  Similar to "Cell Offset" but with a value relative to the input image size. Use this with "Shift Method" set to "Interpolation".
- Print Histogram Log:
  Toggles if the histogram values should be printed to the log.
- Interpolation Factor to Previous Destination:
  Every calculated destination point is interpolated to the previous destination to smooth out the steering. This parameter defines the weight of interpolation.
- Maximum Cells per Side to use for Interpolation:
  Defines how many cell "Shift Method = Interpolation" uses to interpolate the x-coordinate.
- Enable Ultrasonic Cell Elimination:
  Enables multiplying the array of ultrasonic sensors to the histogram to plan a path around obstacles.
- Shift Method:
  Chooses between two implemented methods to calculate a destination point from the histogram. "Interpolation" interpolates the x-coordinate from the center of the maximum cell in the direction of it's neighbors based on the value of neighboring cells. "Averaging" considers the index of every cell that has a value higher than "Averaging Threshold" and calculates the average index over these indices.
- Averaging Threshold:
  Threshold for "Shift Method = Averaging".
- Ignoring Threshold:
  Cell values below this threshold are set to zero. This helps with fuzzy segmented road edges.

- Turning Timer in ms:
  Timer for turning. When a Turning Intention is recieved over the Direction Pin the mean of the Gaussian PDF is moved to the corresponding edge of the histogram. After the set ms the mean is moved back to the center.

Purpose:
This filter receives a birds-eye view image of the segmented road and calculates a destination to drive toward.
`Image Cropper`
Inputs:

- Input Image : cVideoPin
  The image that we want to crop.

Outputs:

- Output Image : cVideoPin
  The cropped image (different dimensions as the input image).

Purpose:
This filter takes an input image, crops it (according to fixed values at the moment) and then sends the cropped image out at the other end. Can be used for many use cases like stripping away borders around an image.

`Lane Detection`
Inputs:

- Camera Video : cVideoPin
  The input is either the camera image from the car or the virtual camera from VTD. Note that both images are RGB images.

Outputs:

- Birdseyeview Video : cVideoPin
  The output is a birdseyeview video in grayscale.

Purpose:
The filter creates the birdseyeview by applying OpenCV filters and functions onto the camera image. The birdseyeview simulates (by transforming the input image) a top-down view of the street.

`Lights`
Inputs:

- TurningIntention : tSign
- ThrottleBrakeControl : tSpeedController

Outputs:

- Left Indicator : tBoolSignalValue
- Right Indicator : tBoolSignalValue
- Head Lights : tBoolSignalValue
- Hazard Lights : tBoolSignalValue
- Brake Lights : tBoolSignalValue

Purpose:
This filter uses the turning intention and the target speed information to send out the information to the arduino communication filter.

`Local Path Planning`
Inputs:

- Birdseyeview Video : cVideoPin
  The video that is received from Lane Detection containing BEV (may include abstract world model later).
- Current Speed : tSpeedData
  Current Speed of the vehicle received from VTDSensing

Outputs:

- Local Path : tPath
  10 Points of the calculated path

Properties:

- Minimum Distance : tFloat32
  The minimum distance that is used for path planning.

Purpose:
This filter takes the bev to calculate a path to drive through the abstract world model.

`Normalize Speed`

Inputs:

- SteeringIn : tSpeedController
  Speed command from ControlConsolidation, to be normalized

Outputs:

- SteeringOut : tSpeedController
  Normalized speed command

Purpose:
Converts the speed command from $[-1, 1]$ to the range of values that the ArduinoCommunication filter accepts. Since the Arduino filter accepts values in $[-100, 100]$, but the actual speed of the car only changes within $[-tbd_1, tbd_2]$, we linearly map onto the latter range.
Furthermore, we check for large jumps between the last two commands. If the difference is too big, we limit the change to a pre-set constant into the desired direction.

`Normalize Steering`

Inputs:

- SteeringIn : tSteeringController
  Steering command from ControlConsolidation, to be normalized

Outputs:

- SteeringOut : tSteeringController
  Normalized steering command

Purpose:
Converts the steering command from $[-1, 1]$ to the range of values that the ArduinoCommunication filter accepts. Since the Arduino filter accepts values in $[-100, 100]$, but the actual steering angle of the car only changes within $[-85, 85]$, we linearly map onto the latter range.
Furthermore, we check for large jumps between the last two commands. If the difference is too big, we limit the change to a pre-set constant into the desired direction.

`Record USS`

Inputs:

- USSStruct : tUltrasonicStruct
  Ultrasonic signals coming from the car

Purpose:
The filter records the ultrasonic signal values coming from the car in a semicolon-separated CSV file '/tmp/ultrasonic.csv'. The columns represent the values defined in tUltrasonic-Struct: frontLeft, frontCenterLeft, frontCenter, frontCenterRight, frontRight, sideLeft, sideRight, rearLeft, rearCenter, rearRight.
Since C++ formats the float values with a comma, you'll want to replace those commas with simple dots, e.g. using the sed command:

```
sed s/,/./g ultrasonic.csv > ultrasonic_cleaned.csv
```

Opening and visualizing the file can then for example be done with the following R code:

```
colnames = c("left", "center-left", "center", "center-right", "right",
"side-left", "side-right", "rear-left", "rear-center", "rear-right")
data = read.csv("ultrasonic_rear_clean.csv", sep = ';', col.names = colnames)
for(i in 1:10){
  plot(data[,i], main = colnames[i], ylim = c(0,500))
}
```

`Save Images`

Inputs:

- Input Video : cVideoPin

Purpose:
This filter takes an input video stream and saves each single frame as a .bmp image. The standard path to save the images is /tmp, but it can be configured via the filter property. Please note that in the directory choice dialogue the 'force absolute path' checkbox needs to be checked for this to work.

`Segmentation`

Inputs:

- Basler camera video : cVideoPin

Outputs:

- Segmented image : cVideoPin
- Copy of original Basler camera image : cVideoPin

Properties:

- Path to neural network model : string
- Neural network input width : int
- Neural network input height : int

Purpose:
The filter takes each image of the Basler camera stream and sends it through the segmentation neural network provided. The segmentation contains, as an RGB image, binary segments on each channel. The R channel represents street, G is street signs and B is background (i.e. everything else). In combination to sending out the segmented image, a copy of the image used for segmentation is also send out so that other filters (especially CutoutSignClassifier) can compute on both images without a lag between the frames.

`Sign to Speed`

Inputs:

- tSign
  The recognized street sign

Outputs:

- tSignalvalue
  The desired target speed for the universal controller with regard to the detected street sign.

Purpose:
This filter adapts the speed for turning and stopping.

`Street Sign Extractor`

Inputs:

- Basler camera image : cVideoPin
- Segmented image : cVideoPin

Outputs:

- Cut out street sign : cVideoPin

Properties:

- Minimum side length of sign to cut out : int

Purpose:
The filter takes a camera image and its corresponding segmentation and, if there is a street sign present with side lengths bigger than specified, cuts it out, resizes it to 32 x 32 pixels and sends it out through the output pin.

`Test Controller`

Outputs:

- SpeedController : tSpeedController
- SteeringController : tSteeringController

Purpose:
This filter serves as a plattform to send hardcoded values to the controll consolidation filter. Its purpose is to have an input for the car, so developers can see it responding or not.

`Universal Controller`
Inputs:

- Target Value : tControlInputOutput
- Actual Value : tControlInputOutput

Outputs:

- Control Value : tControlInputOutput

Purpose:
This filter takes a target and an actual value of any kind and can use the control value to get the actual value near to the target value. This filter will be used to contain different kinds of controllers, e.g. PID for steering or other for throttle.

`US Smoothing`
Inputs:

- USSDataIn : tUltrasonicData
  struct with float value and timestamp for each of the five front sensors

Outputs:

- USSDataOut : tUltrasonicData
  The respective float values become -1.0 if we have no valid values to report

Purpose:
The filter implements filtering and moving-average smoothing of the input ultrasonic sensors.
Since the front sensors exhibited a very worrying amount and type of noise, we investigated the problem further. There are three distinct types of noise we found. One is constantly showing very high or very low values (possibly caused by echo), the second is strongly trend-breaking variation of the values, varying strongly over several consecutive time steps, the last is a short error noise, also breaking the trend, but only showing a small blip instead of a prolonged phenomenon. An example for the second type of noise would be that we have an object at a 30cm distance for several seconds before and after the break, but within that trend break window, the sensors report values between 25 and 300, without any apparant reason in the physical world.
To combat the first type of error, we simply filter out any values outside of [1,295]. The second error is handled by smoothing the signal via a 5-frame moving window. To that effect the filter utilizes five ring buffers of size 5, each recording the last five values received by the respective sensor. The filtering of too big or too small values happens during the

computation of the moving average. Only values inside this range are used to compute the smoothed value.

`VTD Communication`

Inputs:

- ThrottleBrakeControl : tSpeedController
- Steering : tSteeringController

Outputs:

- RDBOut : RDB Package

Purpose:
This filter is the interface between the control filters and VTD (via RDB) to steer and control the simulated vehicle.

`VTD Sensing`

Inputs:

- RDBImage : RDBImage
- RDBDepthImage : RDBDepthImage
- DRBData : RDBData
- RDBUltrasonicLeft : RDBUltrasonicLeft
- RDBUltrasonicCenterLeft : RDBUltrasonicCenterLeft
- RDBUltrasonicCenter : RDBUltrasonicCenter
- RDBUltrasonicCenterRight : RDBUltrasonicCenterRight
- RDBUltrasonicRight : RDBUltrasonicRight

Outputs:

- Camera : ImageFormat::getCarRealsenseColor()
- DepthCamera : ImageFormat::getCarDepth()
- UltrasonicSensors : tUltrasonicData
- Speed : tSpeedData

Purpose:
This filter is the interface between VTD (via RDB) on one side, and the perception and control group on the other side. It ensures that the other groups receive consistent data and can concentrate on their specific tasks.

`XBox Filter`
Outputs:

- SpeedController : tSpeedController
- SteeringController : tSteeringController
- EmergencyBrake : tBoolData
- EmergencyBrakeReset : tBoolData
- ToggleAutonomousXBox : tBoolData
- TurningIntention : tSign

Properties:

- XBoxController in use : int
  Specify which of the two game controller is used.
- Type of connection : int
  Specify which connection is used, bluetooth or usb. The buttons are mapped differently.

Purpose:
To have influence on the cars' behavior while driving, there is the XBox game controller. This filter is its ADTF interface.
The operator can steer via the left thumb stick (button 9), accelerate with the RT axis and brake through the LT axis. The speed is set by the RT and LT buttons and is held by the car until another input is recieved. Sending one of those inputs the car escapes the autonomous mode.
It is possible to send other information to the car: B triggers the emergency brake routine. Y resets any connected emergency brakes. A enables the autonomous mode. X enables the manual driving mode.
For test and demo purposes the direction cross can send sign information: UP is no sign detected, DOWN is the stop sign, LEFT and RIGHT are the respective turn signs.

Appendix B

Lanekeeping Experiments

This appendix solely contains plots concerning real test scenarios with the model car. The plots show the passing time in the x-axis and the error angle as radians in the y-axis. The green dashed horizontal line marks the error angle value 0, the red dashed vertical line denotes the end of the car's driving route and the start of the same route backwards. Therefore, the graph approximatively looks point-symmetric at the point of intersection of the green and red dashed line.

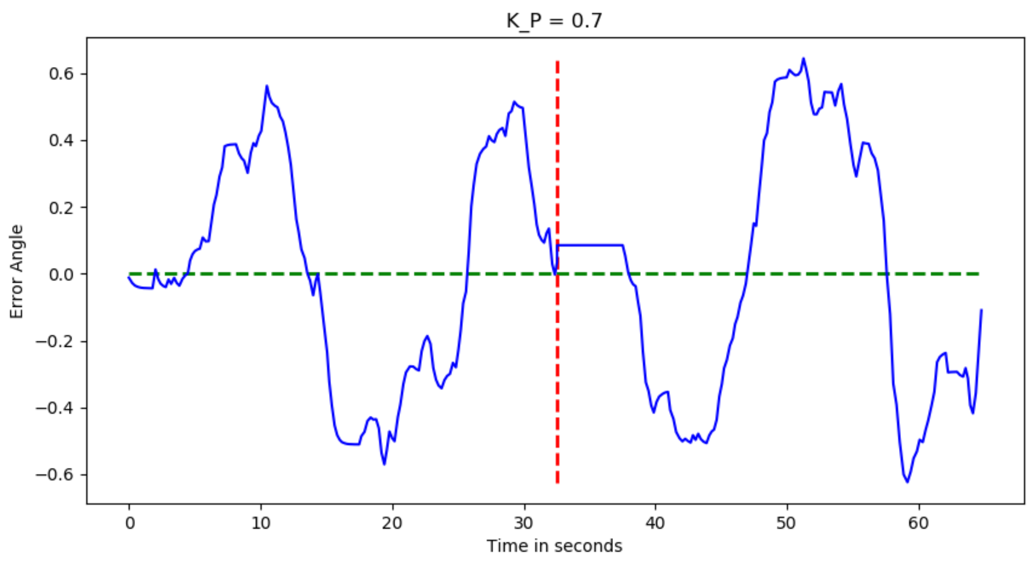


Figure B.1: 20190320-164800 - 20190320-164856, objectively the best configuration

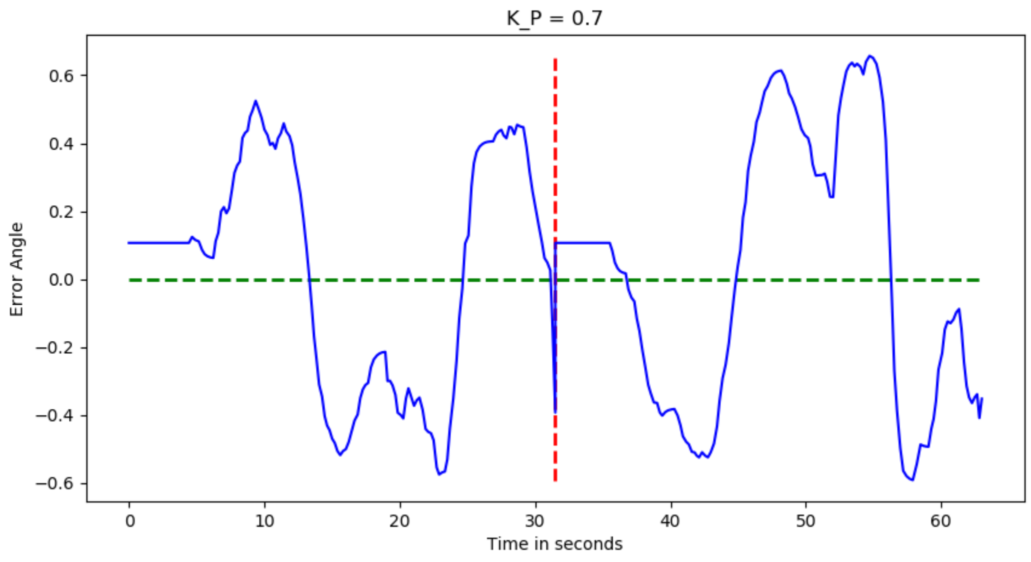


Figure B.2: 20190320-170428 - 20190320-170531

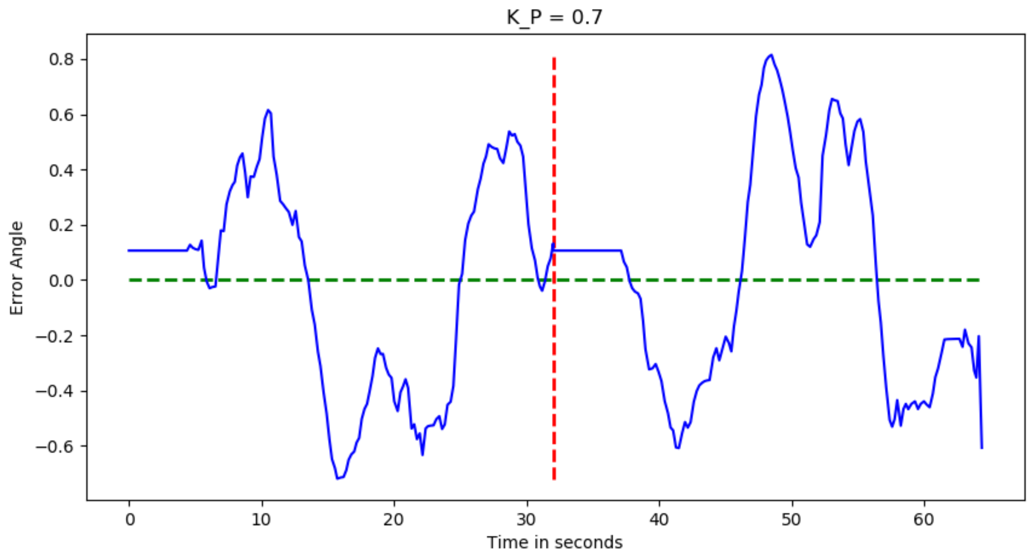


Figure B.3: 20190320-174351 - 20190320-174558, low bot idx → car stays closer to the middle line but does not steer as much in long turns

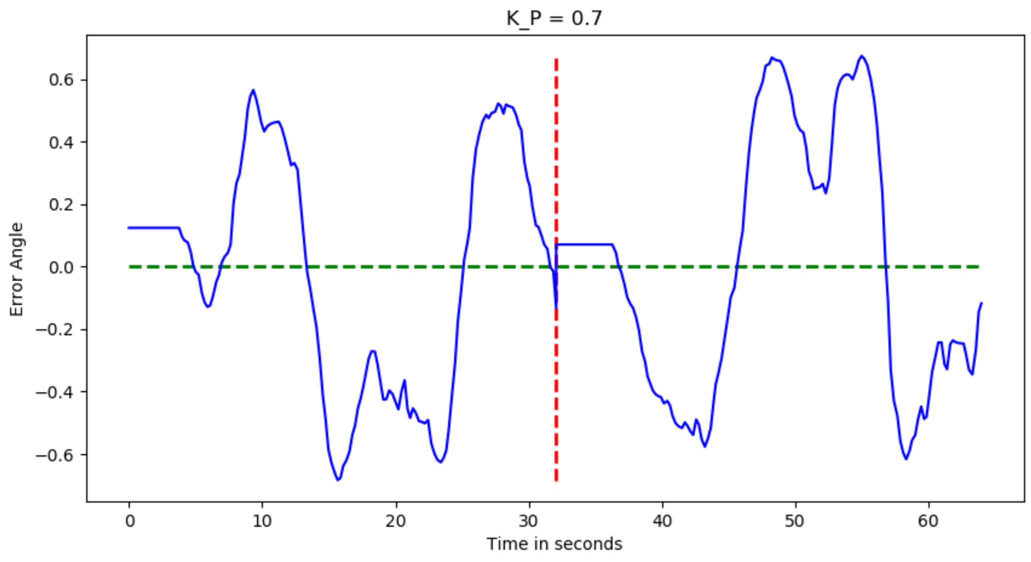


Figure B.4: 20190320-174758 - 20190320-174910

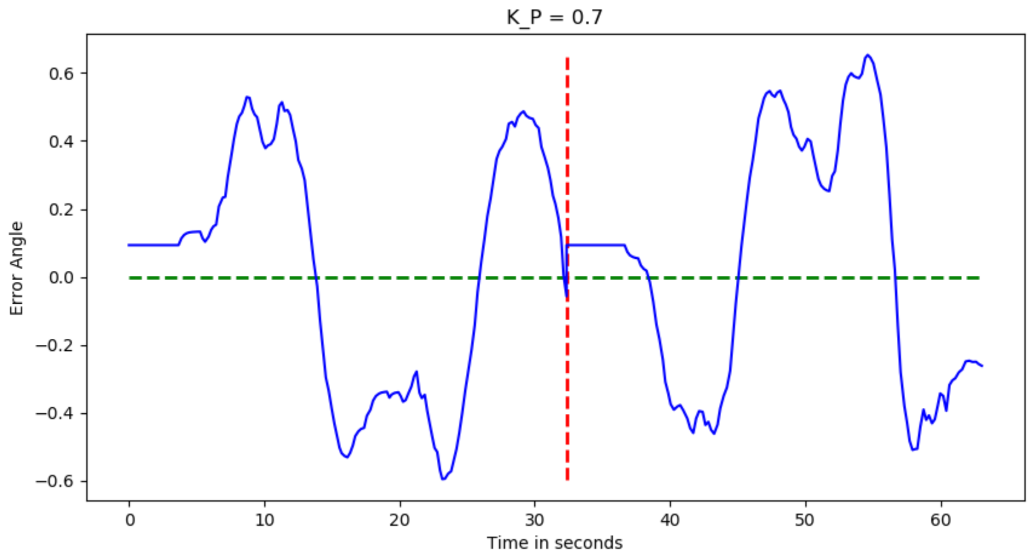


Figure B.5: 20190320-175411 - 20190320-175455

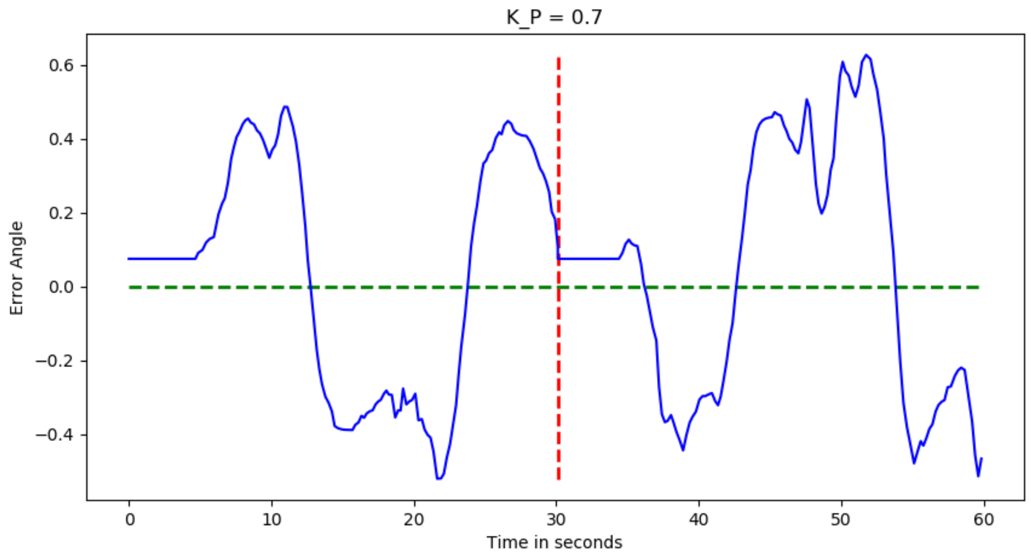


Figure B.6: 20190320- 175627 - 20190320-175742

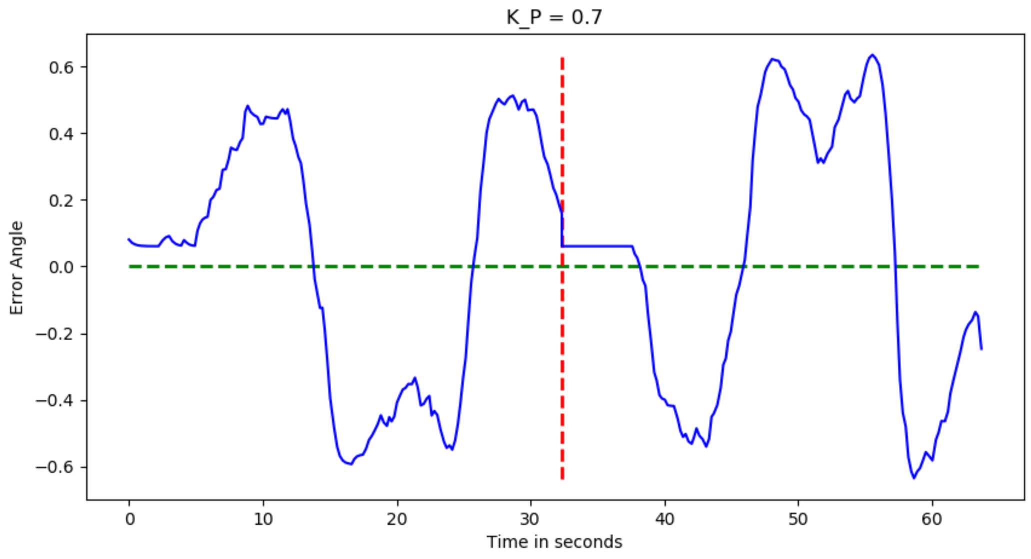


Figure B.7: 20190320- 175903 - 20190320-175950

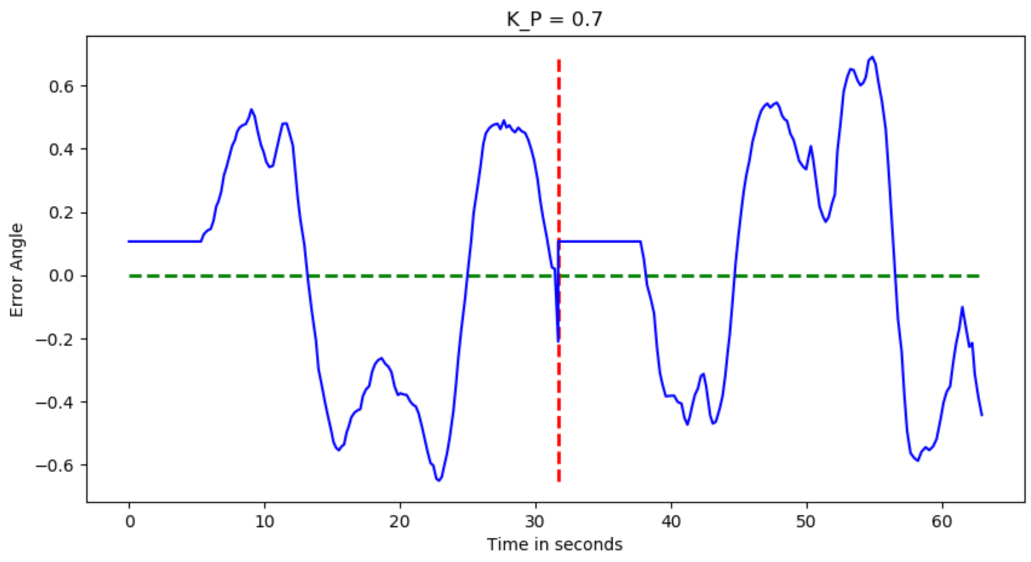


Figure B.8: 20190320-180154 - 20190320-180241

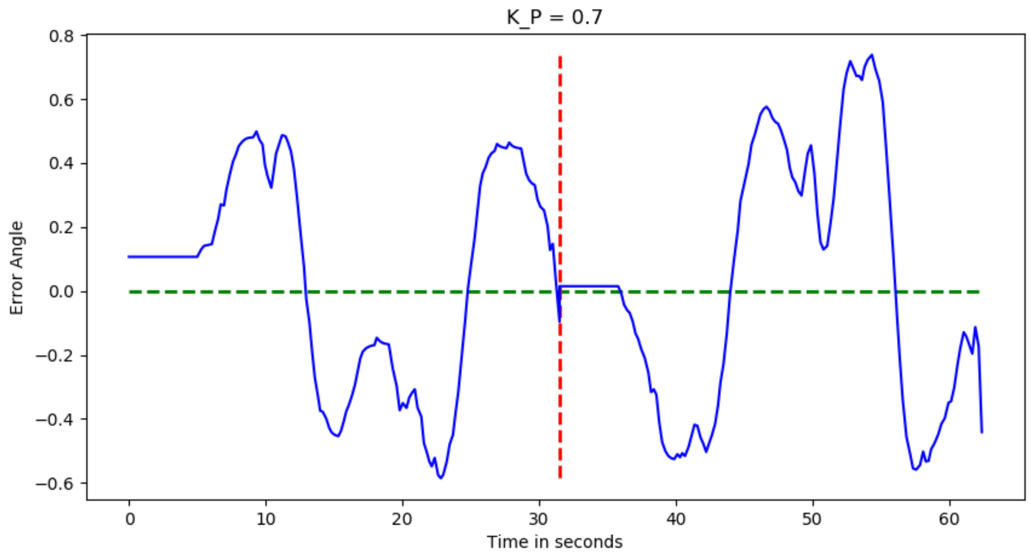


Figure B.9: 20190320-180350 - 20190320-180457

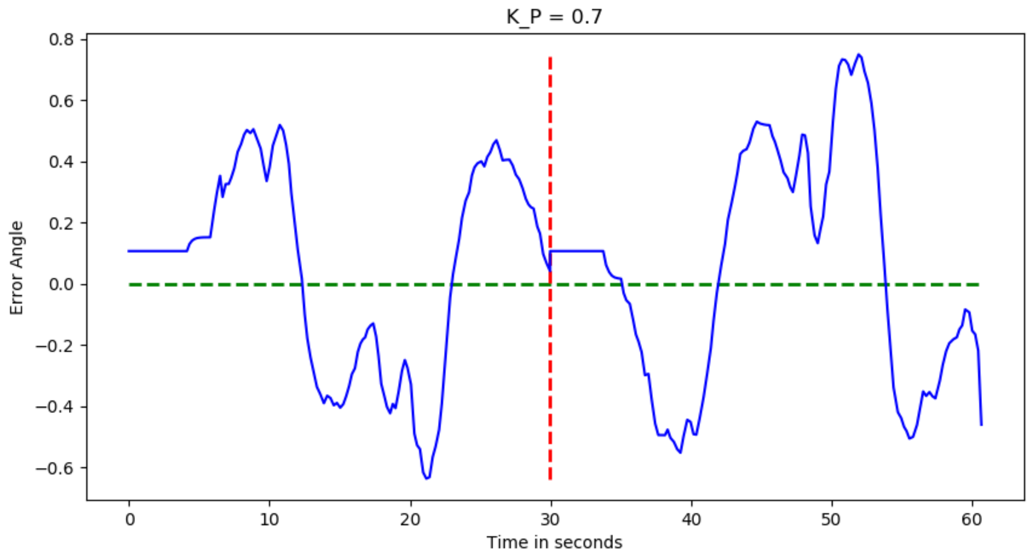


Figure B.10: 20190320-180607 - 20190320-180701

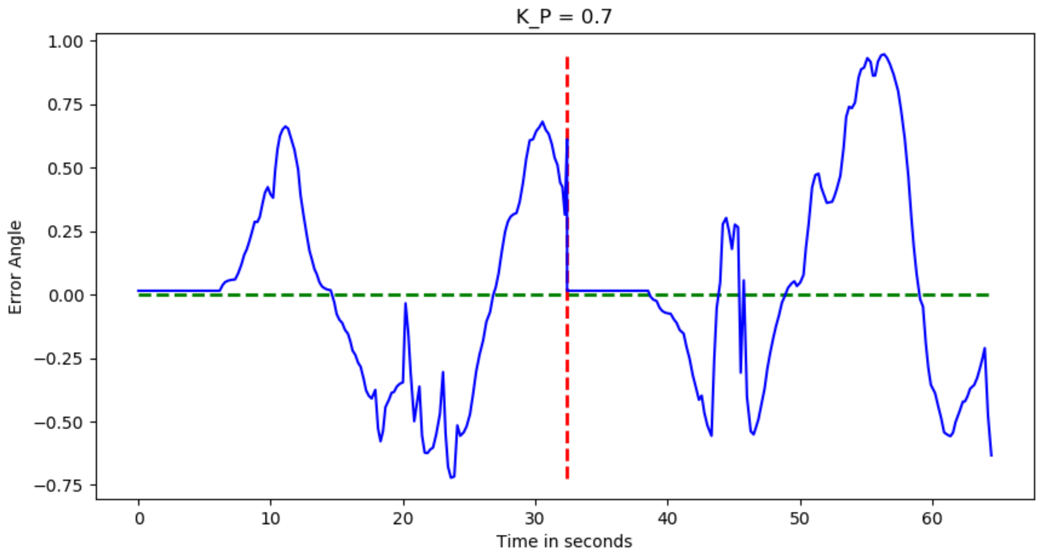


Figure B.11: 20190320-180835 - 20190320-180949, car drove off the road, low gauss allows only minor steering as only the top is chosen because of the threshold

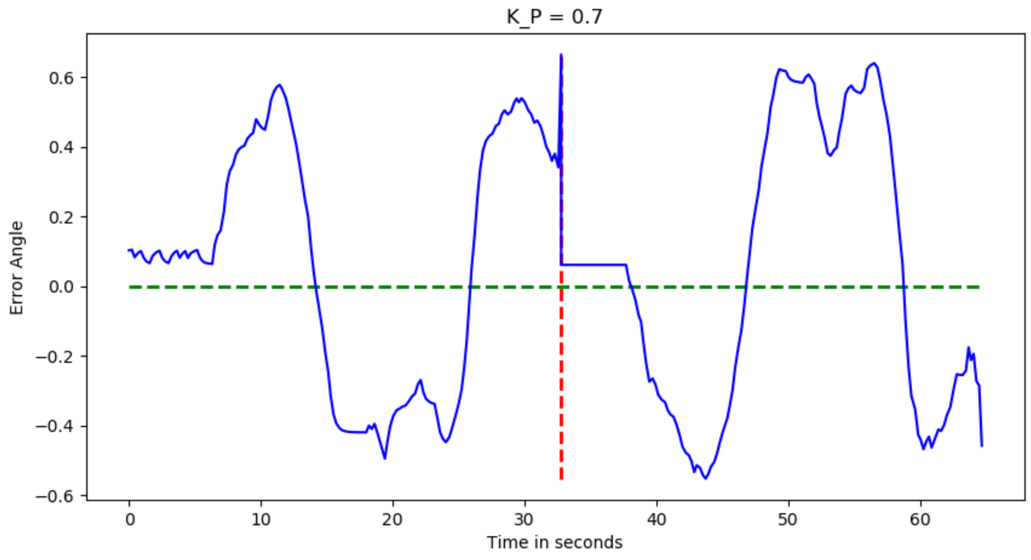


Figure B.12: 20190320-181143 - 20190320-181250, car drove close to the border of the street before steering

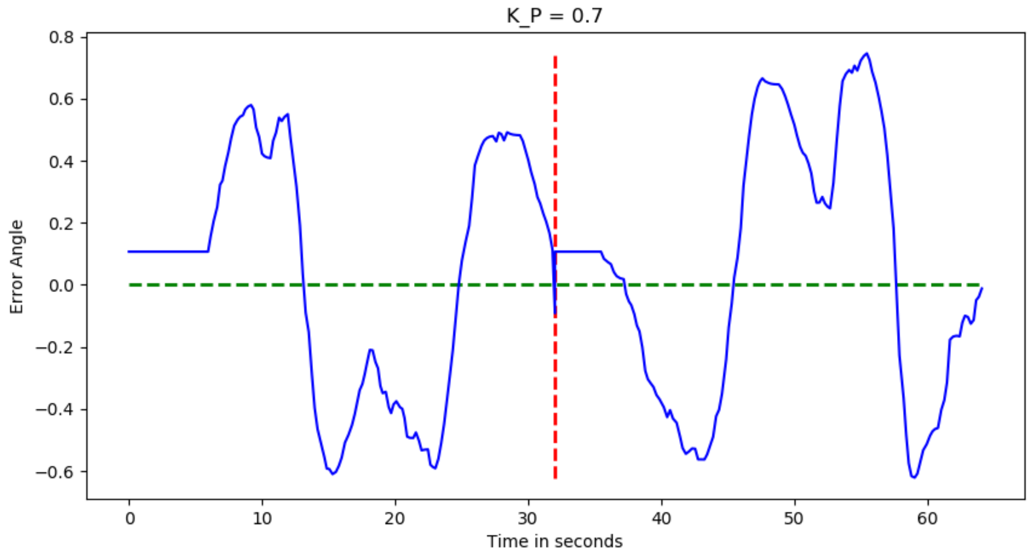


Figure B.13: 20190320-181546 - 20190320-181642

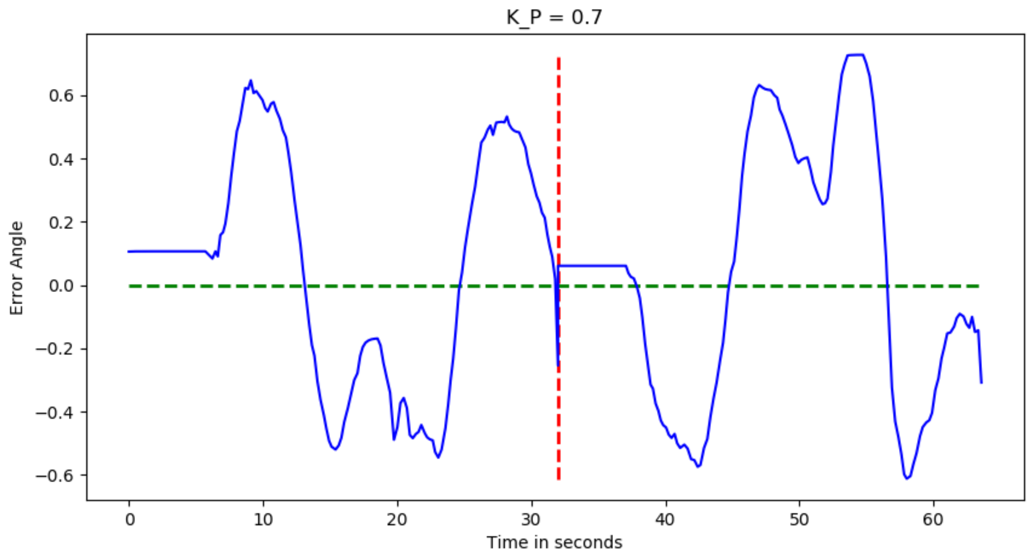


Figure B.14: 20190320-181800 - 20190320-181911

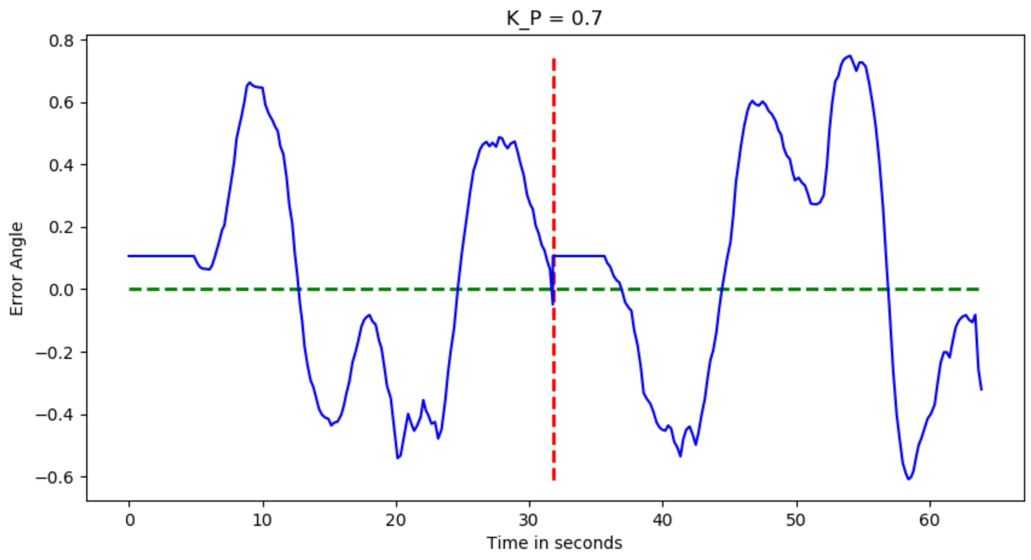


Figure B.15: 20190320-182101 - 20190320-182203

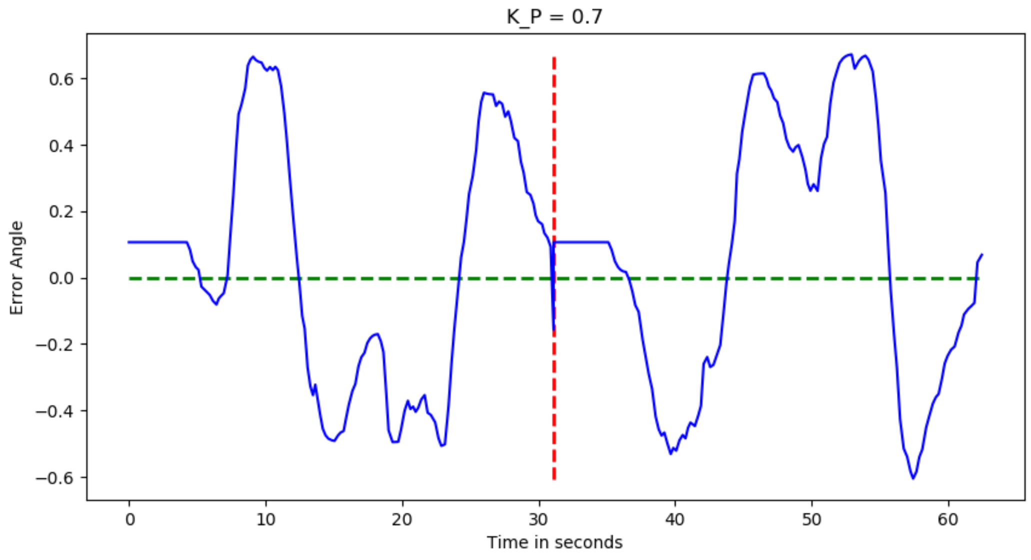


Figure B.16: 20190320-182433 - 20190320-182531

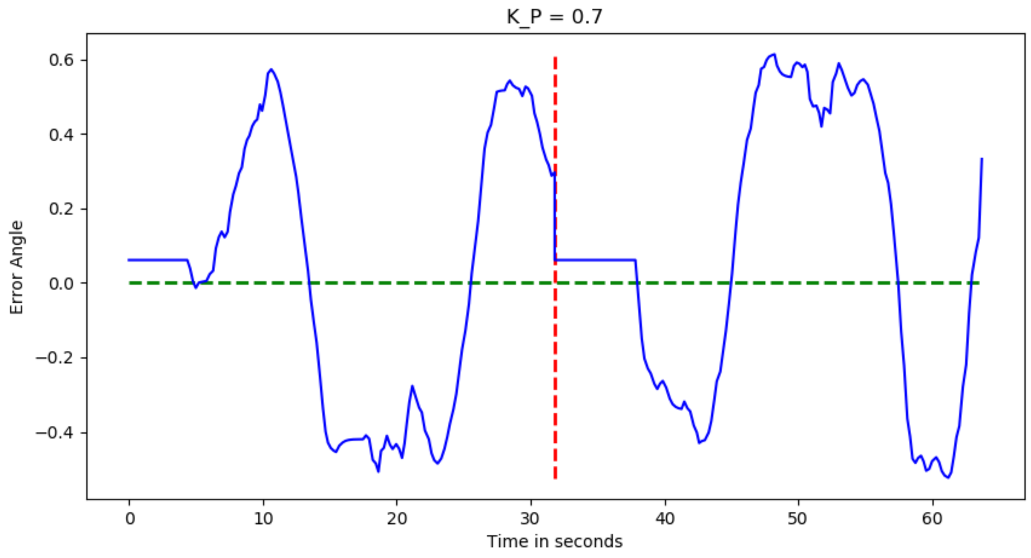


Figure B.17: 20190320-182753 - 20190320-182854, low threshold makes the car drive close to the border of the road

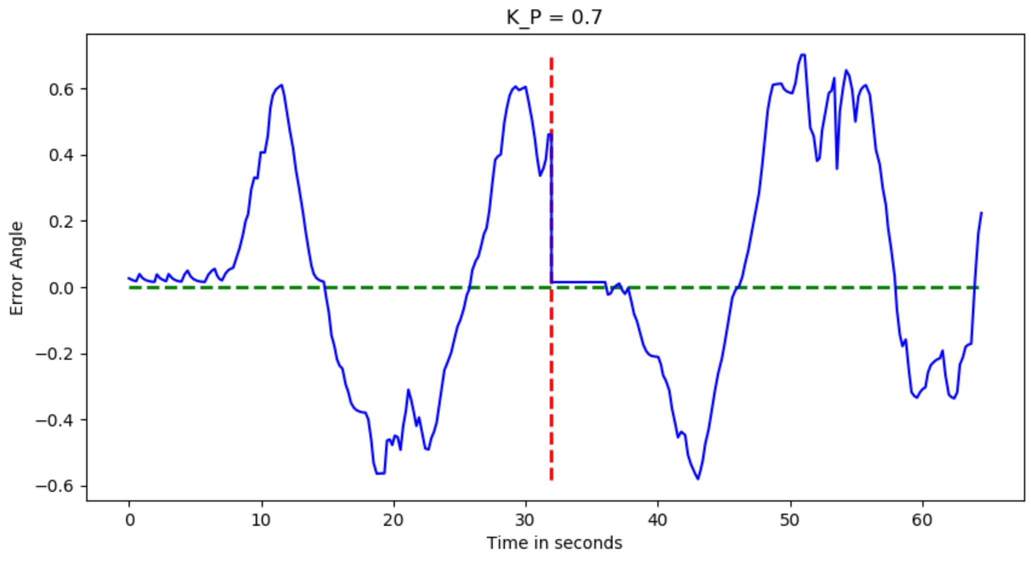


Figure B.18: 20190320-183040 - 20190320-183137

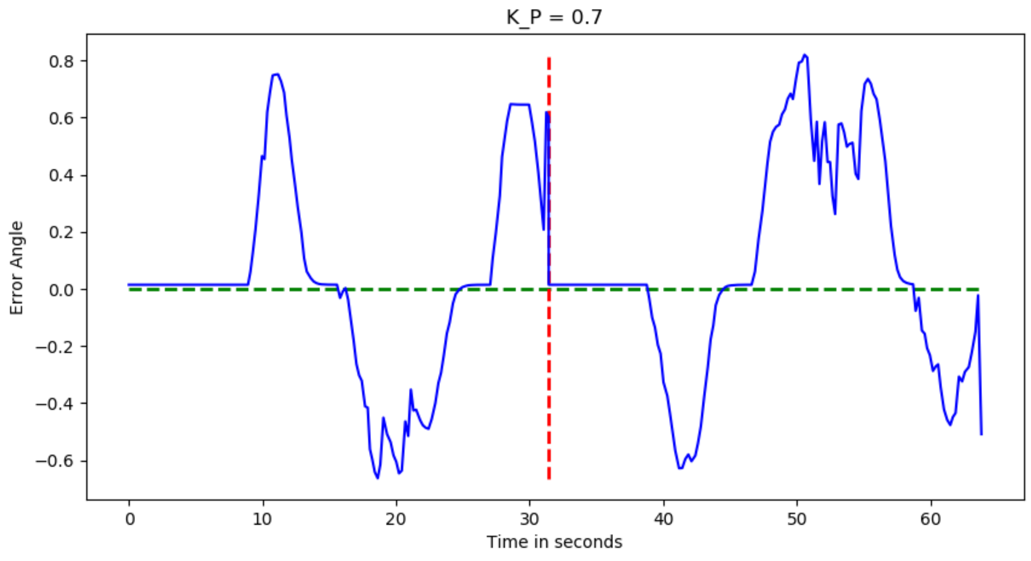


Figure B.19: 20190320-183321 - 20190320-183422

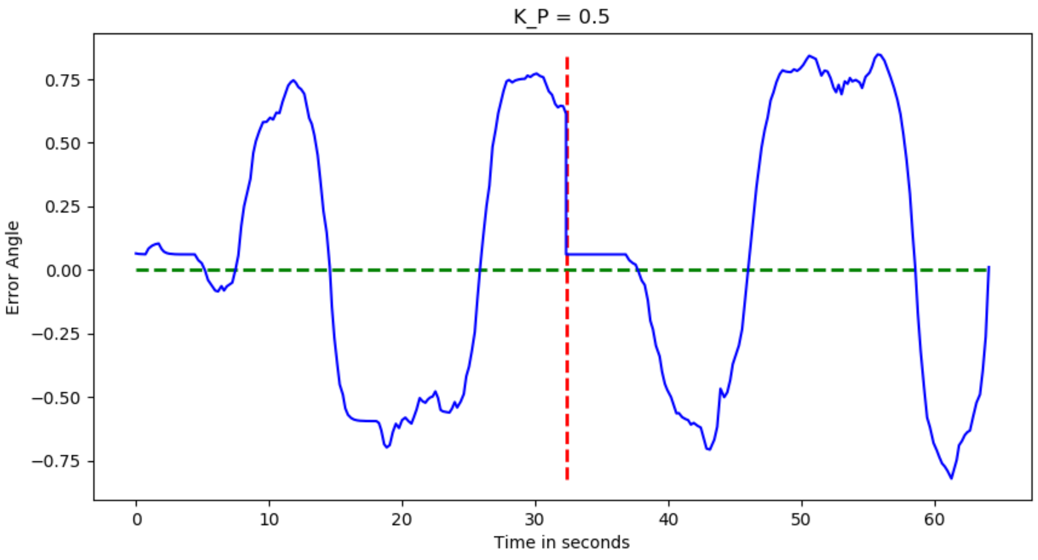


Figure B.20: 20190320-183533 - 20190320-183645, car drove off the road because all cells are considered - even noise close to the border of the street influences the choice of the destination

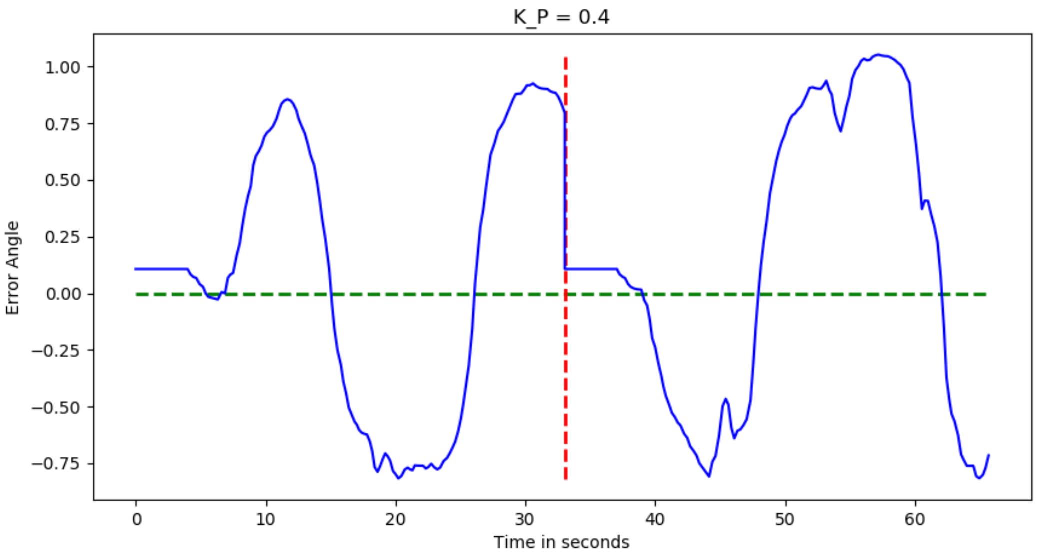


Figure B.21: 20190320-184027 - 20190320-184127, the steering is not enough

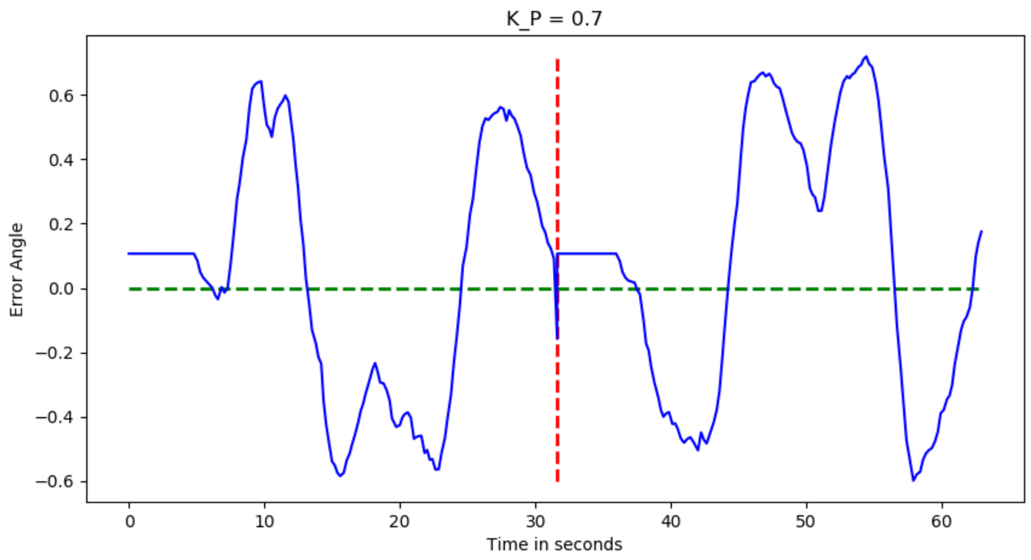


Figure B.22: 20190320-184257 - 20190320-184359, drove off the street at the end of the first half of the experiment, drove off the street on the second half as well, however, the car coincidentally drove on the street again at some point

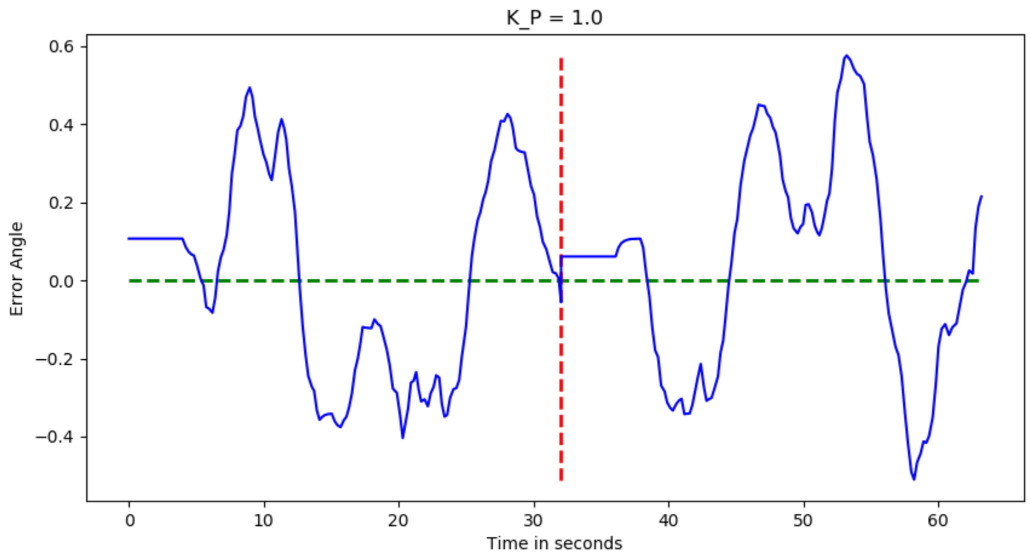


Figure B.23: 20190320-184623 - 20190320-184708

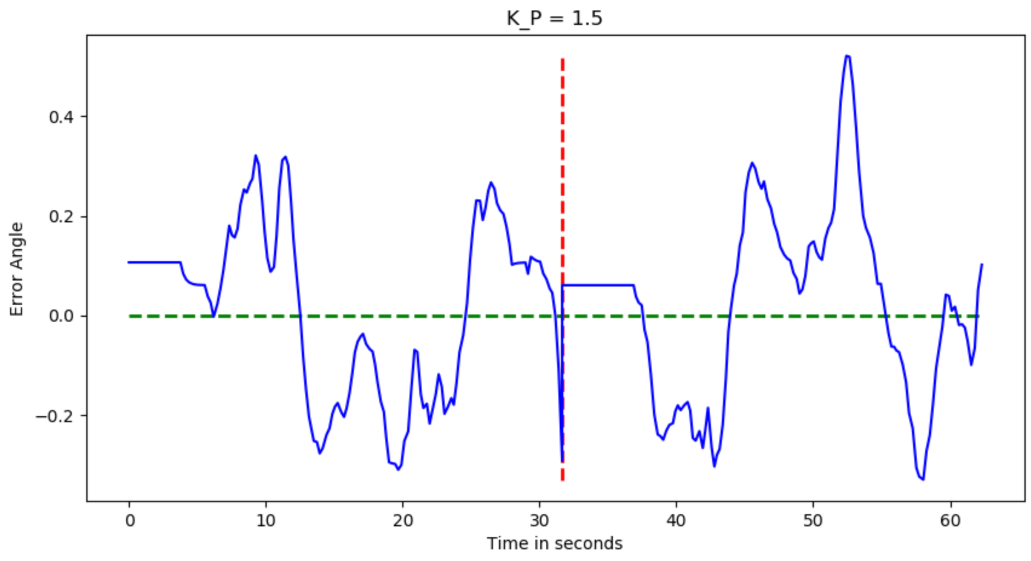


Figure B.24: 20190320-184931 - 20190320-185019

List of Figures

List of Algorithms

Bibliography

Nvidia docker. https://gitlab.com/nvidia/cuda/blob/ubuntu16.04/9.0/devel/cudnn7/Dockerfile, zuletzt abgerufen am 14.08.2018.

Tensorflow c api details, a. https://www.tensorflow.org/install/install_c, zuletzt abgerufen am 14.08.2018.

Tensorflow docker, b. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/tools/docker/Dockerfile.devel-gpu-cuda9-cudnn7, zuletzt abgerufen am 14.08.2018.

Tensorflow serving, c. https://www.tensorflow.org/serving, zuletzt abgerufen am 14.08.2018.

BFFT Produktreferenz ADAS-Modellfahrzeug.

WANT. https://www.want.nl/audi-autonomous-driving-cup-wedstrijd Last visited: 04.03.2019.

OpenCRG, a. http://opencrg.org Last visited: 30.04.2018.

OpenDRIVE, b. http://opendrive.org Last visited: 30.04.2018.

OpenSCENARIO, c. http://openscenario.org Last visited: 30.04.2018.

Segnet. https://arxiv.org/abs/1511.00561, zuletzt abgerufen am 15.02.2019.

WANT. http://cdn-reichelt.de/documents/datenblatt/A300/DATENBLATTULTRASCHALLSENSOR.pdf Last visited: 04.03.2019.

VIRES Simulationstechnologie GmbH. http://vires.com Last visited: 30.04.2018.

V. Badrinarayanan, A. Kendall, and R. Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.

J. Cho, J. Choi, W. Yoo, G. Kim, and J. Woo. Estimation of dry road braking distance considering frictional energy of patterned tires. *Finite Elements in Analysis and Design*, 42(14-15):1248–1257, 2006.

V. Dumoulin and F. Visin. A guide to convolution arithmetic for deep learning. *ArXiv e-prints*, Mar. 2016.

N. Fiedler, M. Bestmann, and N. Hendrich. Imagetagger: An open source online platform for collaborative image labeling. In *RoboCup 2018: Robot World Cup XXII*. Springer, 2018.

A. Garcia-Garcia, S. Orts-Escolano, S. Oprea, V. Villena-Martinez, and J. G. Rodríguez. A review on deep learning techniques applied to semantic segmentation. *CoRR*, abs/1704.06857, 2017. URL `http://arxiv.org/abs/1704.06857`.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

H. Le and A. Borji. What are the receptive, effective receptive, and projective fields of neurons in convolutional neural networks? *arXiv preprint arXiv:1705.07049*, 2017.

Y. Li, K. H. Ang, and G. C. Chong. Pid control system analysis and design. In *IEEE Control System Magazine*, volume 26. 2006.

MathWorks. polyfit. `https://www.mathworks.com/help/matlab/ref/polyfit.html`. Last visited: 03.09.2018.

R. R. McHenry and B. G. McHenry. *Accident Reconstruction*. McHenry Software, 2008. URL `http://www.mchenrysoftware.com/forum/SNAGoption.pdf`.

A. Palazzi. Project 4 - advanced lane finding. `https://github.com/ndrplz/self-driving-car/tree/master/project_4_advanced_lane_finding`, 2017. Last visited: 01.09.2018.

M. A. Rahman and Y. Wang. Optimizing intersection-over-union in deep neural networks for image segmentation. In *ISVC*, 2016.

A. Shustanov and P. Yakimov. Cnn design for real-time traffic sign recognition. *Procedia Engineering*, 201:718–725, 2017.

B. Siciliano and O. Khatib. *Springer handbook of robotics*. Springer, 2016.

M. Sqalli. Lane detection. `https://medium.com/@MSqalli/lane-detection-446986c44021`, Nov. 2016. Last visited: 20.08.2018.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014. URL `http://jmlr.org/papers/v15/srivastava14a.html`.

J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel. The German Traffic Sign Recognition Benchmark: A multi-class classification competition. In *IEEE International Joint Conference on Neural Networks*, pages 1453–1460, 2011.

S. Suzuki and K. Abe. Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing*, 30(1):32–46, apr 1985.

P. Zhao, J. Chen, Y. Song, X. Tao, T. Xu, and T. Mei. Design of a control system for an autonomous vehicle based on adaptive-pid. In *International Journal of Advanced Robotic Systems*, volume 9. InTech, 2012.