RWTH Aachen University
Software Engineering Group

Model-based design and simulation of autonomous vehicle controllers

Bachelor Thesis

presented by
Lorang, Mike

1st Examiner: Prof. Dr. B. Rumpe

2nd Examiner: Prof. Dr.-Ing. S. Kowalewski

Advisor: Dipl.-Ing E. Kusmenko

The present work was submitted to the Chair of Software Engineering

Aachen, 13th September 2017

Eidesstattliche Versicherung

Name, Vorname

Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum

Unterschrift

Abstract

Cyber-physical systems often rely on complex software performing tasks like control, planning, environment understanding and others. Thus an efficient way to design and model these systems is needed. The component and connector paradigm has proven to be an appropriate methodology to develop cyber-physical systems in a hierarchical manner and is widely used, e.g., in the automotive domain. However, modern component and connector C&C languages often lack elements indispensable for the domain. This thesis will present a model driven approach to develop a controller for an autonomous vehicle controller with the (C&C) language MontiCAR. The controller is self-adaptive, modular and extendable. Since the code generator for the MontiCAR language is not completely finished yet, the controller is also developed in C++. This way the MontiCAR language can be evaluated and compared to an existing approach and the system can be validated in a simulator.

Contents

Chapter 1

Introduction

From January to May 2017, there were 1197 deaths due to car accidents in Germany and 325 in the month of May alone. That is an increase of 0.9% of death compared to May 2016. In the first five months of the year 2017 the police recorded 1.05 million accidents on the roads of Germany, which is an increase of 2.5% compared to these same five months previous year. According to World Health Organization, each year, approximately 1.2 million lives are lost in traffic accidents. There are about 50 million people who suffer car accidents every year.

The main reasons for the enormously high number of accidents are speeding, lack of concentration, slow reaction time, miscalculation of the situation, distraction, tiredness, drugs and alcohol.
An autonomous vehicle does not have any of these negative attributes as slow reaction time or lack of concentration. Thus self-driving cars could make the roads safer than ever. Other benefits of autonomous driving are the higher safe speeds, improved transportation of goods and time saving [SG15]. The development of self-driving cars includes many disciplines like computer-vision, sensor fusion, inter and intra car communication, regulation of the actuators and many more.

Automating vehicles has been a research topic for decades. In 2004, DARPA offered a one million dollar prize to challenge participants to get a self-driving vehicle to complete a 150-mile trek through the Mojave Desert. None of the participants could finish the race and claim the prize.
A year later, DARPA presented the challenge again, this time offering 2 million dollars to challenge teams to complete a 132-mile track trough the Mojave Desert with autonomous cars. Five teams completed the race. Stanford University, which turned a Volkswagen Touareg into a self-driving vehicle called Stanley, won.
In 2007 the challenge started to become about urban driving and passenger cars. The event required the teams to build an autonomous vehicle capable of driving in traffic and performing complex maneuvers such as parking, merging, passing and negotiating intersections. This was the first time that autonomous vehicles were interacting with both manned and unmanned vehicle traffic in an urban environment. Since then research quickly move forward [dar07].

1.1 State of the Art

There are already numerous active safety systems, e.g. adaptive cruise control, implemented in vehicles. In October 2014 Tesla released Tesla Autopilot. Tesla Autopilot is a driver assistance system that supports the driver in the driving process. Hardware to support fully autonomous driving is already built in the Tesla cars since the *Model S*. To reach further levels of autonomy software updates can be downloaded to the car.
Since 2009 Google also started developing an autonomous car. In 2015 their fully self-driving car nicknamed *Firefly* hit the public road for the first time. The car has no steering wheel or pedals anymore.
Other companies such as Apple, Volkswagen, Volvo, Audi, Ford and BMW are also developing autonomous cars [Sel]. Thus within the next few years customers can already buy fully autonomous vehicles [New17].

1.2 Requirements

One of the main differences between a manned and a unmanned vehicle is the number of sensors needed. An unmanned autonomous vehicle needs detailed information about its environment as accurate as possible to come to the best decision. It is, however not possible for the sensor to reflect the real world with an infinite accuracy because disturbances of like heat, magnetic fields or the imperfection of the quantified conversion from the analog sensor value to the digital value. Apart from the sensors who measure the state of the vehicle and its environment, the actuators influence the state of the vehicle through manipulating e.g. the brake, the acceleration of the steering angle. A efficient vehicle controller is modeled in order to map the sensor data and the vehicle goal to actuator commands. The vehicle controller is not only defined by its model of algorithm but also by its set of parameters. This set of parameters often changes from one vehicle to another. That is the reason why a method is needed which takes care of that task. Ideally all the sensor values and the actuating values can be read or written from one common interface. The developer should not deal with the complex functionality of sensor or actuators but only focus on the heterogeneous interface providing simple access to the sensors and actuators.

Based on the previous considerations, we derive the following requirements towards a simple, yet extensible self-driving racing car system: (R1) The vehicle must be able to follow a given trajectory on the test track. (R2) The vehicle must be able to adjust its velocity according to a reference value with respect to soft acceleration constraints. (R3) The vehicle controller must be able to compensate for sensor imperfections as well as be able to assess the limits of maximum Signal to Noise Ratio (SNR) it can mitigate. (R4) The vehicle must be able to improve its performance in terms of the Mean Squared Error (MSE) of the driven trajectory as well as its velocity and acceleration (R5). The architecture must be modular and easily adaptable, allowing for an easy interoperability, interchangeability and testability of the components (R6). The complexity of the underlying heterogeneous sensor and actuator technology has to be abstracted away, i.e., the software developer should only need to deal with a homogeneous interface providing access to all sensor signals as well as all possible actuator inputs.

In the following sections a model-driven development approach leading to a solution fulfilling our requirements will be presented and deployed and tested in *TORCS*, **T**he **O**pen **R**acing **C**ar **S**imulator.

1.3 Outline

First, the simulator used to validate the controller is introduced. Therewith it is outlined how the sensors are simulated. Subsequently the preliminaries for the understanding of connector and component languages and PID controllers are given. Then the controller architecture in the C&C language MontiCAR is elucidated. Thereby is is explained how the different components communicate with each other, and how the sensor data is read/written from/to the simulation. It is outlined how the noisy sensor data is being filtered before being processed. It is explained how the vehicle is able to adapt its controller parameters in an automated way. Additionally an alternative on how the car can be controlled is briefly elucidated. In the end some experiments to show how the vehicle fulfills the requirements are presented.

Chapter 2

Simulation

To be able to test the developed program, a test environment is needed. A simulation is a fitting environment to conduct experiments with autonomous driving for the purpose of finding a strategy for the operation of the system.

2.1 TORCS simulator

In the thesis *TORCS*, the open source 3D car racing simulator, is used. It was created by Eric Esipé and Christophe Guionneau and has its initial release in 1997. It is a cross-platform game which is written in C++. Since *TORCS* is an open source, modular and easily expandable simulator, it has been adopted as a base for many research projects. Since 2008, *TORCS* has also played an important role in various research fields within the IEEE Conference on Computational Intelligence and Games [BW14]. Additionally the *TORCS* simulator, which means the physics and the car engine, runs at a frequency of 500Hz whereas the car updates at a frequency of 50Hz [BWS13].

2.2 Sensor simulation

In *TORCS* the data obtained from the simulation perfectly represents the current physical state of the car and the track. To have a more realistic reproduction of the reality, additive white Gaussian noise (AWGN) is added to certain values obtained from the simulator.

2.2.1 Additive White Gaussian Noise

The following section is inspired by [CT12]. Additive white Gaussian noise is a basic noise model used in information theory to mimic the effect of many random processes. The artificial noise has the following characteristics:

- Additive means that it is added to the perfect data provided by the simulator.
- White refers to having the same intensity at different frequencies giving it a constant power spectral density. The analogy to the color white refers to the uniform emissions at all frequencies in the visible spectrum. In other words, the noise is uncorrelated with itself.

- Gaussian refers to the Gaussian or Normal distribution used to generate the noise.

The additive white Gaussian noise is mostly determined by front-end analog receiver thermal noise. The Gaussian assumption is a consequence of the fact that many small noise sources contribute to the noise, thus invoking the Central Limit Theorem which states that when independent random variables are added, their probability normalized sum tends toward a normal distribution, even if the original variables themselves are not normally distributed.

Other than in the simulator, it is not possible for a sensor to measure certain physical values, e.g. the current position, with infinite accuracy. Thus to make the application more realistic, AWGN is added to the data retrieved from the simulator.
The sensors are simulated by adding the random distributed variable

$$Z_i \sim \mathcal{N}(0, \sigma) \tag{2.1}$$

to the simulator value.

In Equation 2.1 the noise in the i^{th} time-step Z_i is described. In Equation 2.2 it is described how the noise Z_i is added to the value retrieved from the simulator X_i to shape the sensor-value Y_i.

The distributed sensor value Y_i is computed like

$$Y_i = X_i + Z_i \tag{2.2}$$

where X_i is the value retrieved from the simulator. X is a placeholder for any measured quantities.

Chapter 3

Preliminaries

3.1 Connector and Component models

This section is inspired by [Wor16], [SM13] and [EK16]. Component and connector architecture description languages enable developers to compose complex systems from component models. The models abstract from implementation details and provide a well-defined interface to hide the complexity of one single component. Thus the developer can focus on the functionality of the system instead of the low-level implementation. Connector and Component models (C&C models) consist of components, directed, typed, and named ports together with connectors that connect the different ports. Imagine two components, *componentA* and *componentB*, having one input and one output port each. If the output port of *componentA* is connected to the input port of *componentB* the input port of *componentB* takes the value of the output port of *componentA*. A component can contain other components which means that one or more components can be aggregated to create another component. One component can have an arbitrary amount of input and output ports.
C&C models are widely used for the design and development of Cyber-physical systems to represent features and their logical interaction. C&C models have the advantage of being modular and extendable. One component in the model can easily be replaced by another. Additional components can be simply added to the model. Another advantage is that complex features can be hierarchically decomposed into subfeatures, developed and managed by different domain experts. The paradigm focuses on software features and their communication. Prominent examples for C&C languages, used in academia as well as in the industry, are Simulink and LabView [EK16].

3.1.1 MontiCAR

In this thesis the textual modeling family MontiCAR (**Mo**deli**n**g and **T**est**i**ng of **C**yber-Physical **Ar**chitectures) is used to model the controller for the autonomous vehicle. Unlike many other C&C languages, MontiCAR has full SI unit and unit conversion support. In Simulink for example units are used for documentation purposes only.

3.2 PID Controller

This section is inspired by [ÅH95]. PID (propotional integral derivative) control is one of the early control strategies. The PID controller is a common control algorithm which is mostly used in feedback loops. A feedback loop is a system structure that causes the output from one component to eventually influence the input of the same component. The idea of feedback is simple yet, extremely powerful. Application of the feedback principle has resulted in major breakthroughs in control, communication and instrumentation. The feedback is responsible to make the process variables close to their respective set-point despite of disturbances and variations of the process characteristics. Since many control system used PID controllers, they have proved to be satisfactory and still have a wide range of applications in industrial control [ÅH95].

3.2.1 The PID actions

The behavior of a PID controller in a time continuous system can be described as:

$$u(t) = K_p \left(e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt} \right) \tag{3.1}$$

where u is the control variable and e is the control error defined as

$$e = y_{sp} - y \tag{3.2}$$

where y is defined as the system output and y_{sp} is the set-point value. Thus the control variable is a sum of three terms: the P-term, which is proportional to the error, the I-term, which proportional to the integral of the error, and the D-term, which is proportional to the derivative of the error. The three parameters to choose are K_p, the proportional gain, K_i the integral gain, and K_d the derivative gain.

3.2.2 Proportional Action

In case of pure proportional control, the equation is reduced to

$$u(t) = K_p e(t) + u_b \tag{3.3}$$

The control variable equals the error e at time t multiplied by the P-gain K_p. Thus the control action is proportional to the control error. The variable u_b is a bias or reset. If the error is zero, the control variable is equal to the bias u_b.
The steady state of equilibrium point is when the system variables which define the behavior of the system are unchanging in time. When using proportional control only, the equilibrium point does not agree with the set-point except when it equals the initial system output y_0.

3.2.3 Integral Action

The main function of the integral action is to make sure that the system output agrees with the set-point in the steady state. A small positive error will always lead to an increasing control signal whereas a small negative error will lead to a small decreasing control signal.

In case of pure integral control, the equation is reduced to

$$u(t) = K_i \int_0^t e(\tau) d\tau. \tag{3.4}$$

The I-term not only increases the action with respect to the error but also to the time time it has persisted. The control action of the I-term will be increased as times passes. A pure integral controller is able to bring the error to zero, however it would be slow and heavily oscillating.

3.2.4 Derivative Action

The purpose of derivative action is to improve the closed-loop stability. The intuitive explanation for the instability without the derivative term is, because of the process dynamics, it will take some time before a change in the control variable is noticeable in the system output. That is why the the control system will be late correcting the error. The derivative term takes action on small errors and thus predicts the error before it is noticeable for the P- and I-term.

In case of pure derivative control, the equation is reduced to

$$u(t) = K_d \frac{de(t)}{dt} \tag{3.5}$$

A purely derivative controller can not bring the error to zero, or the system to its set-point. The D-term aims at bringing the rate of change of the error to zero.

3.2.5 Windup Guard

All actuators have limitations, e.g. a motor has a maximum revolutions per minute or the steering angle of a car is bounded by -45 degrees to 45 degrees. It may happen that the control variable reaches the actuator limits. If a controller with integrating action is used, the error will continue to be integrated. Thus the integral term will wind up and become very large. The windup may occur in large set-point changes or large disturbances or malfunctions in the system.

To prevent that from happening an windup guard is introduced. The windup grad checks if the integral error is beyond a certain range. If this is the case, the absolute value of integral error will be reduced accordingly.

3.2.6 Digital implementation

For the digital implementation of the PID controller, the parallel form of the PID controller is used. This means that the PID output value is computed according to

$$u(t) = P_{term} + I_{term} + D_{term} \tag{3.6}$$

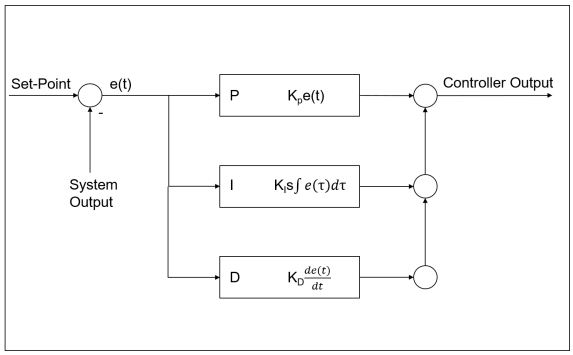


Figure 3.1: The P-, I-, and D-term are being calculated before being summed up and outputted.

where

$$P_{term} = K_p e(t) \tag{3.7}$$

$$I_{term} = K_i \int_0^t e(\tau) d\tau \tag{3.8}$$

$$D_{term} = K_d \frac{de(t)}{dt}. \tag{3.9}$$

In Figure 3.1 the computation of the parallel form of the PID controller is illustrated.

To implement the PID controller digitally, the proportional, integral and derivative term need to be discretized. The P-term is discretized to

$$P_{term}(t_k) = K_p e(t_k). \tag{3.10}$$

The I-term is discretized to

$$I_{term}(t_k) = K_i \frac{1}{2} \Delta T (e(t_k) - e(t_{k-1})) \tag{3.11}$$

The D-term is discretized to

$$D_{term} = K_d \frac{e(t_k) - e(t_{k-1)}}{\Delta T} \tag{3.12}$$

where t_k denotes the sampling instants, $e(t_k)$ is the error at the sampling instant k and ΔT is the time interval between t_k and t_{k-1}.

Chapter 4

Architecture and Models

4.1 Overall Architecture

In a simulation environment there is a lot of room for experiments and trial and error. To efficiently test a lot of different approaches, the single modules of the application need to be highly portable and exchangeable. Furthermore the parameters of the PID controller can not be set haphazardly because the optimal parameters might be different depending on the noise level, the car properties and where a certain PID controller is deployed in the architecture. Thus an approach to automate the tuning of the PID parameters is needed. The application is a closed loop architecture containing the simulation, the controller and a parameter tuning component using a genetic algorithm. As shown in Figure 4.1 the simulation contains an autonomous vehicle which communicates with the controller via the data adapter. The autonomous car itself has a trajectory planing component which computes a target point in every time step. The simulator sends the sensor data through the data adapter to the controller where it is filtered. By means of the filtered sensor data, the controller computes the actuating values which are then send through the data adapter back to the autonomous vehicle in the simulation. Furthermore, the controller communicates with the tuning component. The tuning component consists of a genetic algorithm whose goal is to optimize the tuning parameters of the controller.

4.2 Sensor simulation

The perfectly accurate sensor data, retrieved from the simulator, is distorted according to section 2.2. The following sensor values are distorted with AWGN using an independent variance σ: the steering angle, the target position, the current position, the car yaw-angle, the current velocity and the gas-pedal value. The distortion component is depicted in Figure 4.2.

4.3 Communication between the simulator and the controller

After the models are designed in MontiCAR, there comes the need to generate and integrate them into an efficient modular and expandable architecture. Each module computes

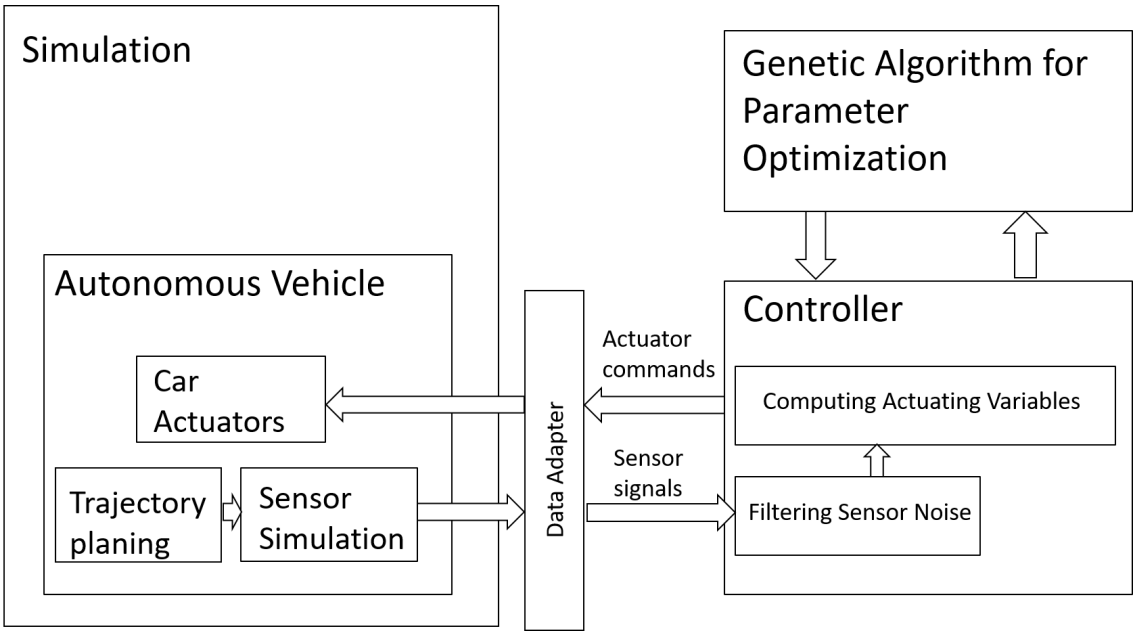


Figure 4.1: Controller Architecture.

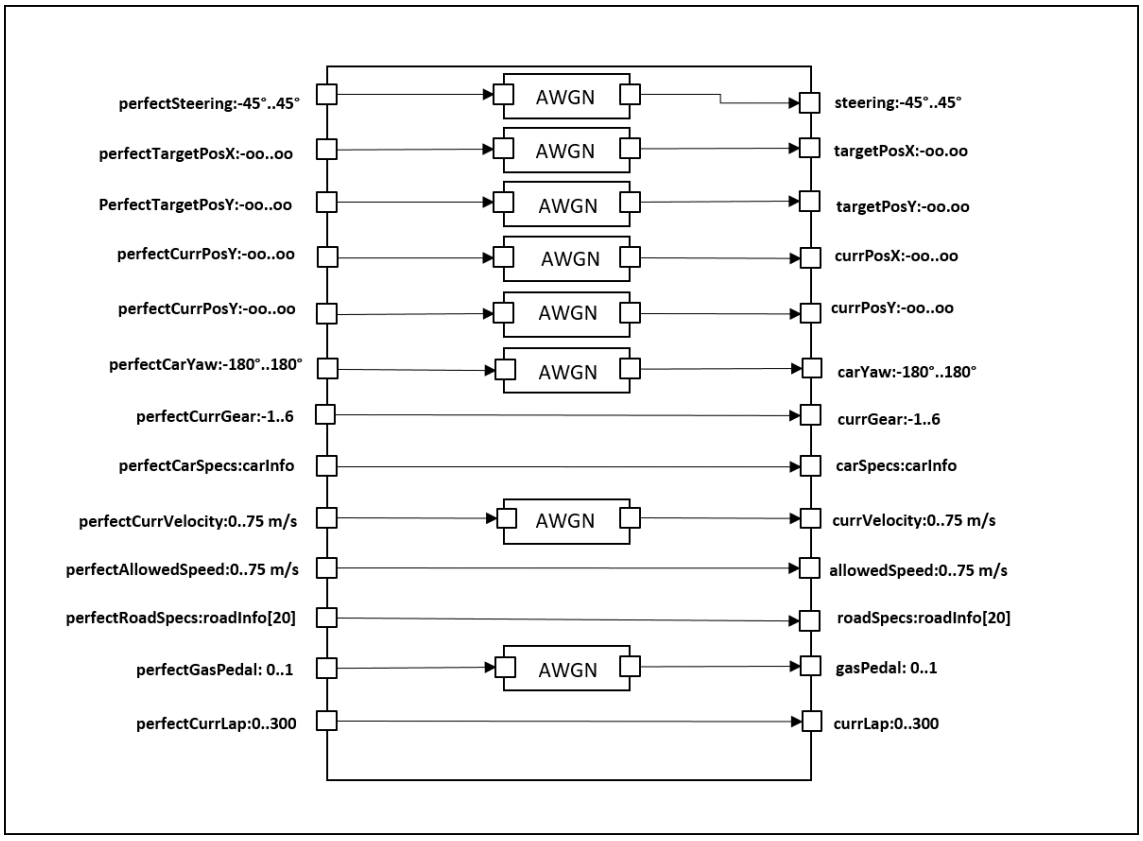


Figure 4.2: Distortion component, adding noise to the values retrieved from the simulator.

its tasks in complete isolation before sharing its results with the other components by means of calls for the distributed system. For the convenient implementation of the communication between the components of the distributed system a compact middleware

	Producer	Consumer
sensor interface	Simulator	Controller
actuator interface	Controller	Simulator

Table 4.1: Overview of the producer and consumer roles.

called OpenDaVinci [Ber16] is used.
OpenDaVinci is a useful tool to develop distributed applications on the same or on different machines. When running on the same machine the interprocess communication engine is realized using *shared memory* and semaphores. The OpenDaVinci middleware offers real-time capability through its concurrency engine. It restricts the module to finish its computation up to a specified time point.
In the thesis there are two applications running simultaneously which need to exchange data in real time, the controller and the simulation. The real-time data exchange is needed in order to have a fast reaction time which is vital in the time critical applications like autonomous driving.
In order to exchange data in the context of the sensor and actuator interface, the components are split into producer and consumer. Hence the simulator acts at first as a producer and creates the *shared memory* which acts as the sensor interface. The controller which only reads from the sensor interface acts as the consumer.
To exchange data between the controller and the simulation, the roles of the producer and consumer are the other way around. The controller creates the shared memory which acts as the actuator interface and writes the computed actuating values into it. The simulation reads from the actuator interface and writes the values to the respective actuator of the car. In Tabular 4.1 there is an overview of which component takes care of which role.

The data transferred over the shared memory is encapsulated into objects. There is an *Input object* which contains the sensor data with the respective getters and setters and an *Output object* containing the actuating values including getters and setters.
The controller reads an *Input object*, computes the actuating values and writes them into the *Output object* in the shared memory. The simulator then again reads from the *Output object*. On one hand, if the shared memory of the sensor interface is not valid, that means if it has not been created yet or it is empty, the controller waits until the shared memory gets valid. On the other hand if the actuating shared memory is not valid, nothing is written to the actuators, which means their current value stays the same and an error message is displayed on the standard output. Semaphores which are built in the OpenDaVinci middleware make sure that both applications do not access the shared memory simultaneously. [Ber15]

4.4 Preprocessing

To avoid processing the raw sensor data, it needs to be filtered before being used to compute the actuating variables. The filtering process enables the experimentalist to eliminate or greatly reduce the amount of high frequency noise and get a clean, undistorted representation of the true phenomenon. The filter process should be simple yet efficient. For analog data the filtering of the sensor data is commonly done using low-pass RC filters. Since the sensors are simulated and in digital form, there is a need to perform the same filtering using digitized data [KR77].

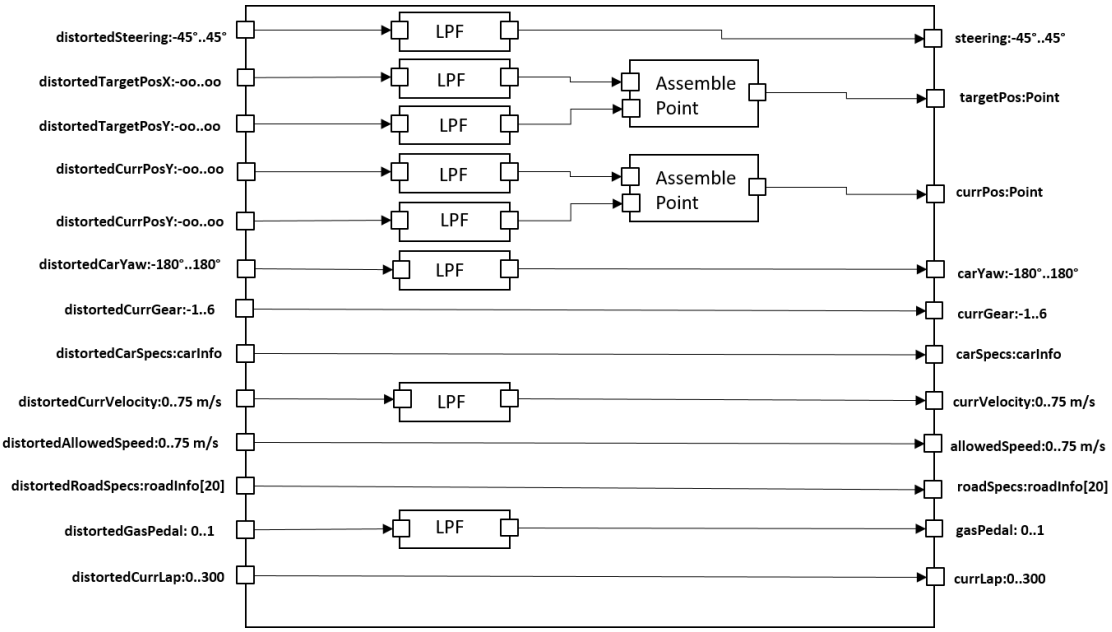


Figure 4.3: The distorted sensor values get filtered before being forwarded to *Main Controller* component.

Low-pass Filter

For the digital implementation of a low-pass filter the output corresponds to the Exponentially Weighted Moving Average (EWMA). The past observations are assigned exponentially decreasing weights over time. It is called moving average, because the output averages the current observation with the previous ones. Old components get insignificant small over time due to the exponential weighting. The EWMA is calculated the following way

$$x_{t+1} = x_t + \alpha(x_t - y_t) \tag{4.1}$$

whereas x_{t+1} is the predicted value at time $t+1$ (new EWMA), y_t is the observed value at time t, x_t is the predicted value at time t (old EWMA) and α is the smoothing factor with $0 < \alpha < 1$.
In context of digital low-pass filter, the smoothing factor α is defined as

$$\alpha = \frac{2\pi\Delta T f_c}{2\pi\Delta T f_c + 1} \tag{4.2}$$

where ΔT is the time step, and f_c is the cutoff frequency [H+86].

The filtering component is depicted in Figure 4.3. It uses low-pass filters to smooth the signal of the distorted sensor signals.

4.5 Main Controller

The goal of the controller is to regulate the actuators of the car with respect to the control signals emitted by multiple sensors. An efficient controller architecture is needed to make sure the influence of disturbances to the car's behavior and the reaction time regarding

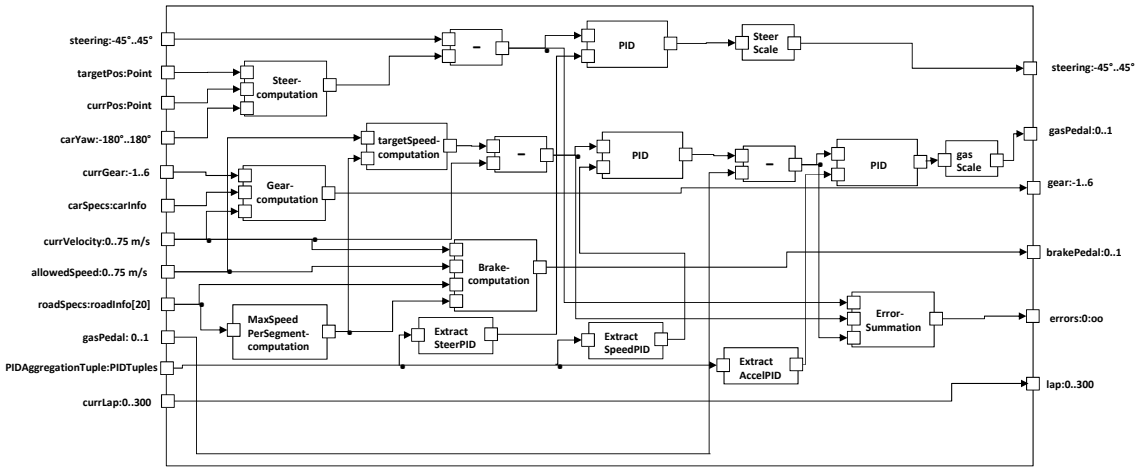


Figure 4.4: Controller Architecture with the sensor interface and the interface to the genetic algorithm on the left side. On the right side there is the interface to the actuators and the genetic algorithm.

changes in the target values is minimal. In Figure 4.4 the architecture of the component responsible for the computation of the actuating variables is illustrated. On the very left the ports for the filtered sensor data and a port for the genetic algorithm are represented. On the very right ports to output the actuating variables plus an interface to the genetic algorithm are given. The sensor data from the simulator is being filtered before it is used to calculate the control values outputted to the actuators. Additionally the PID parameters for the given PID controllers are inputted via the interface port for the genetic algorithm. The boxes inside the controller box are the controller's components. In every time step every component computes one or more output values by means of their input values. The different components' input and output ports are connected in a way to compute the desired controller outputs. The `brake`, `steering`, `gear` and `gasPedal` value are forwarded to the actuator interface whereas the `errors` value and the lap if forwarded to the genetic algorithm.

The MontiCar implementation for the controller architecture is illustrated in Figure 4.5. One component can be declared with the keyword *instance*. After declaring a component it needs to be connected to other components in order to do its computation. One port of a component can be connected to another port of another component. With the keyword `connect` the connection is declared and with the symbol $->$ the two ports to connect are defined. This will make sure that the output of one component gets passed to the input of the next component.

4.5.1 PID Controller

After the fundamentals of PID controllers were elucidated in Section 3.2, a genetic PID controller component needs to be implemented and deployed. There are three PID controllers deployed in the main controller with the objective of regulating the steering angle, the velocity, and the gas pedal value of the car. If the output of the PID controller should be bounded, additional parameters can be passed to the respective instance of the PID controller like depicted in line 9 in Figure 4.5. This will have the effect that all the values higher than the upper bound will be mapped to the upper bound and all the values lower

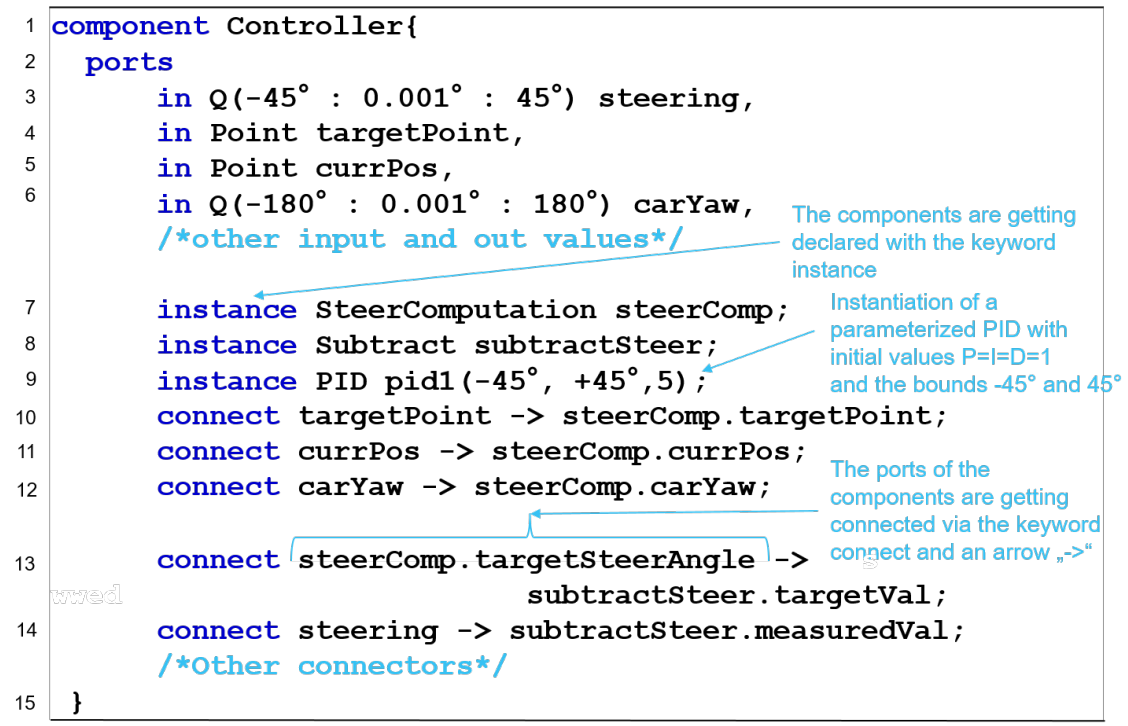


Figure 4.5: The Code that belongs to the Figure 4.4. The inputs and the components are getting declared and connected.

than the bottom limit will be mapped to the bottom limit. In Figure 4.6 there is the implementation of the PID controller in MontiCAR. The PID controller component gets three arguments: the upper and lower bound of the output, as well as the windup guard. The keyword `static` defines that the variable is not reset by the next call of the component but keeps the current value. Thus the variables which save the previous time, the integral error and the previous error are declared static in order to access them in future time steps. In Figure 4.7, there is the code defining the behavior of a PID controller, implemented according to section 3.2.

4.5.2 Extracting the PID Tuples

Before stepping into the details of how the actuating commands are computed, the extract components are explained. These are needed to understand how the genetic algorithm communicates with the controller. To optimize the parameter values for the three PID controllers, their parameters need to be passed to the controller as an input. This way, the quality of the parameter sets can be evaluated during the driving process. Each PID controller takes three parameters, the P-term, the I-term and the D-term, forming a parameter tuple. To bundle these three parameter tuples for all three PID controllers, a struct called *PIDTupleAggregation* is used. Additionally to the parameters, the struct contains a fitness value which is used later on for the evolutionary algorithm. Both structs are depicted in 4.11.
The components *Extract SteerPID*, *Extract SpeedPID* and *Extract AccelPID* are used to extract the right PID parameter tuple from the *PIDAggregation* struct before being inputted to the respective PID controller.

```
1  component PID(Q( -oo : 0.0001 : oo) upperBound = oo,
                 Q( -oo : 0.0001 : oo) lowerBound = -oo
                    Q( -oo : 0.0001 : oo) windup_guard)
2     ports
3        in in Q( -oo : 0.001 : oo) error,
4        in PIDTuple tuple,

5        out Q( -oo : 0.0001 : oo) output;

6        static Z(0:1:oo) time_prev = now()-1,
7        static Q(-oo :0.001 : oo) int_error = 1,
8        static prev_error = 0;
```

Arguments of a PID component

Definition of static variables

Figure 4.6: The definition of the input, output, and static variables of the PID component.

```
9  implementation Math{
10   Z(0:1:oo) time_now = now();
11   Z(1: 1 : oo) time_diff = time_now - time_prev;

12   int_error = 0.5*time_diff*(error-prev_error);

13   int_error = range(-windup_guard , windup_guard,
                       int_error);

14   Q(-oo: 0.001 :oo) P_term = tuple.P * error;
15   Q(-oo: 0.001 :oo) I_term = tuple.I * pid.int_error;
16   Q(-oo: 0.001 :oo) D_term = tuple.D * ((error -
                           prev_error) / time_diff);

17   time_prev = time_now;
18   prev_error = error;

19   output = range(lowerBound, upperBound, P_term + I_term +
                                                D_term);

20 }
21 }
```

Figure 4.7: The math implementation of the PID component.

4.5.3 Calculating the fitting gear

To compute the gear some information about the car is needed like the gear-ratios, the wheel radius, and the red-line-rpm of the engine. The gear-ratios are stored in a array. Since the reverse shift has the value -1 and the gear-ratios are stored in an array, there is a need for a gear-offset attribute to be able to map the gear-ratios in the array. These attributes are stored in the struct called `carInfo`. Additionally to the `carInfo` struct, the *GearComputation* component takes the current gear and the current speed as input. By means of these values, the fitting gear is calculated according to [BWS13]. In Figure 4.8 the implementation of the fitting gear is elucidated.

```
component GearComputation{
      ports
            in (0 m/s : 75 m/s) currSpeed,
            in carInfo carSpecs,
            in (-1 : 1 : 6 ) currGear,

            out (-1 : 1 : 6 ) gear;

implementation Math{
  Q(-oo : 0.01 : oo) gr_up = carSpecs.gearRatio[currGear +
                             carSpecs.gearOffset];
  Q(-oo : 0.01 :oo) omega = carSpecs.enginerpmRedLine/gr_up;
  if(omega*carSpecs.wheelRadius < currSpeed){
    gear = currGear+1;
  }else{
     Q(-oo:oo) gear_down = carSpecs.gearRatio[currGear +
                           carSpecs.gearOffset-1];
     Q(-oo:oo) omega = carSpecs.engineRpmRedLine/gear_down
     if (currGear > 1 && omega*carSpecs.wheelRadius >
                                                 currSpeed){
       gear = currGear-1;
     }else{
        gear = currGear;
     }}}}
```

Figure 4.8: Computation of the fitting gear.

4.5.4 Calculating the acceleration and braking

In order to compute the fitting acceleration and braking, information about the road is required. In the torcs simulator the road is divided into segments. The sensors input the following segment properties into the controller: length, radius, friction coefficient, an *enum* value if it is straight or curved and the distance to the segment end, to determine where the car is situated in the current segment. The distance to the segment end attribute equals the length of the segment for every segment, except the one the car is currently on. These properties are inputted to the controller as form of an array of structs called *roadInfo*. That struct is outlined in Figure 4.9.

The acceleration in driving direction is regulated by the gas pedal. The gas pedal takes an input value in the range from zero to one, where one means full acceleration and zero means no acceleration in driving direction.
First of all the maximal speed, the car is able to drive in the foreseeable road segments, is calculated. Thus a maximal speed value for each of these road segments is calculated. These values are needed for the car to stay on track and not lose control in the curves due to a too high velocity. The maximal speed values per segment are computed in the *MaxSpeedPerSegment-computation* component, by means of the roadInfo array. The output will be another array containing the maximal speed value for each segment.

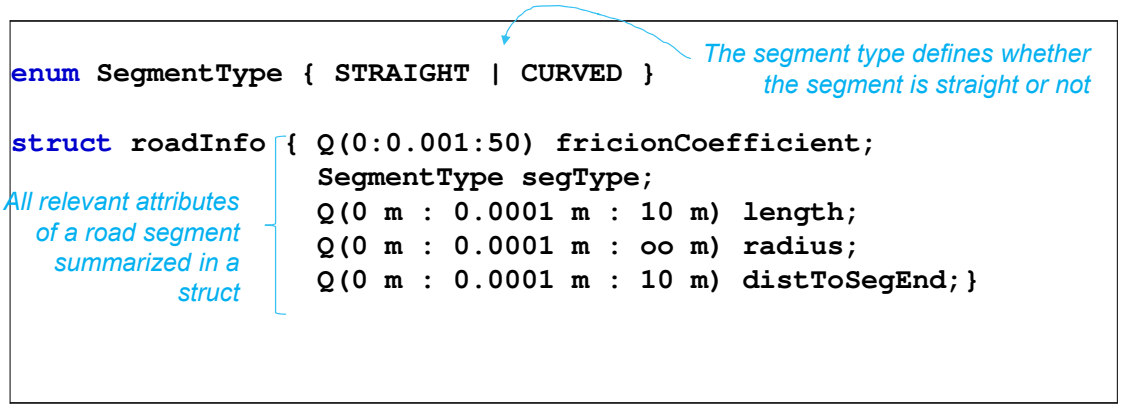


Figure 4.9: The struct containing road information about one segment. An array of twenty of these structs are inputted to the controller.

The following equation

$$\frac{m \cdot v^2}{r} \tag{4.3}$$

defines the force that pushes the car outside the turn which is called centrifugal force. The friction force is defined as

$$m \cdot g \cdot \mu \tag{4.4}$$

where μ is the friction coefficient, g is the gravitational acceleration, m is the mass of the car, r is the radius of the turn and v is the maximal speed allowed in the specific turn. [BW14] In order to pass a turn, the centrifugal force needs to be smaller or equal to the friction force which leads to the following condition

$$\frac{m \cdot v^2}{r} \leq m \cdot g \cdot \mu. \tag{4.5}$$

Solving the equation 4.5 for v results in equation 4.6 which describes the maximal speed in a turn.

$$v \leq sqrt(g \cdot \mu \cdot r) \tag{4.6}$$

Furthermore the target speed needs to be computed. The *TargetSpeedComputation* module takes as input the allowed speed, which can be for example the speed limitation for that road, and the array of the maximal speed values per segment. Additionally it has a parameter defining how many track segments to consider. By means of these values the target speed is computed.
The target speed value is subtracted from the current speed value to get the speed error. This happens in a subtraction component. The speed error is then inputted in a PID component together with the designated PID parameter tuple. By means of these two inputs the PID component calculates the desired acceleration.
The desired acceleration is then subtracted from the current acceleration to compute the acceleration error. It is then inputted along with the designated acceleration parameter tuple into the PID controller. The PID controller, responsible for the regulation of the gas pedal, outputs, by the means of these two inputs, a fitting value for the gas pedal. Since the PID controller does not output bounded gas pedal values, it has to be bounded.

Therefore the gas pedal value is inputted in the *GasScale* component which restricts the gas value to the interval zero to one. The *GasScale* component does that by saving the maximum output value of the PID controller's output and dividing all the incoming values by the maximum output value. The bounded value is then forwarded to the gas pedal actuator. Thus the vehicle is able to adjust its velocity according to a reference value with respect to soft acceleration constraints and requirement (R2) is fulfilled.

The error calculated by each of the subtraction components gets inputted to the *Error-Summation* component. There the absolute value of each error is calculated before being summed up and outputted to the evolutionary component. The evolutionary component needs the summation of the error values to measure the performance of the currently used parameter tuples.
Since the evolutionary component also needs access to the current lap but does not have direct access to the sensor interface, the current lap passes unprocessed through the controller to the evolutionary component.
Lastly the components used to calculate the appropriate brake pedal value are elucidated. The *Brake-computation* component has four input values: the current velocity, the allowed speed, the road information array, and the maximum speed per segment array. There are two different cases on how the brake value is computed. The first one is, when the allowed speed, e.g. the speed limit of the road, is lower than the current speed. Then the brake value is computed the following way:

$$brake = \frac{allowedSpeed - currSpeed}{allowedSpeed}. \tag{4.7}$$

The second case where braking is needed is when it is assumed the car drives on a straight with the speed v_1. In distance d there is a turn where the allowed speed is v_2 with $v_2 < v_1$. In order to know when to start braking, the minimal braking distance s needs to be computed. The car has a certain amount of kinetic energy and the car needs to have a lower amount of kinetic energy in order to ride safely through the turn. When braking the car loses kinetic energy. According to the principle of conservation of energy,

$$\frac{m \cdot v_1^2}{2} - \frac{m \cdot v_2^2}{2} = m \cdot g \cdot \mu \cdot s[J] \tag{4.8}$$

can be formed. Here μ is the friction coefficient, m is the mass of the car and g describes the gravitational acceleration($9.81\ m/s^2$).

The equation

$$s = \frac{v_1^2 - v_2^2}{2 \cdot g \cdot m}[m] \tag{4.9}$$

is obtained when solving equation (4.8) for s.

When the braking distance s is equal or less than the distance d to the curve, the car needs to brake. The braking values are computed according to [BWS13].

Equation 4.10 is obtained when solving equation 4.8 for s.

$$s = \frac{v_1^2 - v_2^2}{2 \cdot g \cdot m}[m] \tag{4.10}$$

4.5.5 Trajectory Planing

Before elucidating how the computation of the steering angle works, the *Trajectory Planing* component is explained, which is responsible for the generation of the target point. The *Trajectory Planing* component is situated in the simulator and it gets passed to the *Sensor Simulation* component before being forwarded through the data adapter to the controller. The target point is calculated by means of the current position, the current speed, information about the road. Firstly the look-ahead distance d is calculated based on the current speed. Afterwards the segment which is at distance d in front of the car is identified. In that segment, the global coordinates of the middle of the track are returned.

4.5.6 Calculating the Steering angle

The *Steer-computation* component takes the target position, the current position and the current car yaw angle as inputs. By means of these three values the output steering angle will be computed according to [BWS13]. MontiCAR allows the programmer to specify the bounds of the input and output variables. This feature comes in handy when the actuator and sensors have a certain range of values where they operate in. In case the input or output values do not stay within that range, the application will throw an exception. Furthermore there is a unusual type-system in MontiCAR. The numeric variables can either be whole numbers written `Z` or rational numbers written `Q`. The step size defines the precision of the ports and variables. Additionally MontiCAR supports typical mathematical operations like *atan*(). In Listing 4.10 the implementation of the steering angle computation is depicted. The current steering angle is then subtracted from the target steering angle to get the steering error. The steering error then acts as one input value of a PID controller. The other input for the PID controller is the PID parameter tuple. The PID component is implemented as a parallel structure accordingly to [ÅH95]. Since the output of a PID component is theoretically not bounded, it needs to be constrained to the maximal output bounds which are in the case -45 to 45 degrees. This is done similar to Simulink by providing additional parameters to the PID in line 9 of 4.5. Regulating the steering with a PID controller assures that the car follows a given trajectory, thus requirement (R1) is fulfilled.

After the development of all the components named in this section, the complexity of the underlying heterogeneous sensor and actuator technology is abstracted away, i.e., the software developer should only have to deal with a homogeneous interface providing access to all sensor signals as well as all possible actuator inputs and requirement (R6) is fulfilled.

4.6 Controller Tuning

4.6.1 Evolutionary Optimization

An evolutionary algorithm is a meta-heuristic optimization algorithm which is inspired by the biological evolution such as reproduction, selection, recombination and mutation. Since the 1950s the natural evolution serves as a model for solving optimization problems. Biologists study the evolution as a mechanism which solves certain problems in nature. Evolutionary algorithms combine the computer as universal calculator with the problem solving ability of the natural evolution process. Simulating the natural evolution on a

```
component SteerComputation{
  ports
      in Point targetPoint,
      in Point currPos,
      in Q(-180° : 0.001° : 180°) carYaw,

      out Q(-45° : 0.001° : 45°) targetSteerAngle;
  implementation Math{
    Q(-oo : 0.0001 : oo) targetAngle = atan(targetPoint.y -
          currPos.y, targetpoint.x - currPos.x) - carYaw;}
    targetSteerangle = targetAngle;
  }
}
```


Figure 4.10: The target steering angle is computed by means of the three input values: `targetPoint`, `currPos` and `carYaw`. In the end the target steering angle is written to the output.

```
struct PIDTuple{Q(0 : 0.001 : 20) P;
                Q(0 : 0.001 : 20) I;
                Q(0 : 0.001 : 20) D;}

struct PIDTupleAggregation{ PIDTuple pidTuples[3];
                            Q(-oo : 0.001 : oo) fitness;}
```

Figure 4.11: Structs which are needed to define an individual.

computer will lead to an approximately exact solution for almost any problem. Potential solutions for a problem get treated as an organism, which will get modified and reproduced according to the evolutionary operations. These modifications are defined by random numbers. Thus evolutionary algorithms belong the the stochastic optimization algorithms which do not have a guarantee of finding an exact solution in a given time.

A potential solution is called an individual. An individual consists of the actual solution-parameters and the fitness function. In this thesis an individual consists of three PID parameter tuples and the fitness value. A PID parameter tuple is again a struct consisting of three parameters, the P-, I-, and D-gain. An individual is modeled as struct called `PIDTupleAggregation`. Both structs are depicted in Figure 4.11. The fitness value defines how good an individual can perform a certain task.

The set of potential solutions of the problem is called population. The population can be compared again to a population of animals in nature. Based on the fitness values of the individuals of the population, certain individuals will be selected to recombine themselves to create a new individual. Once the recombination step is executed often enough, the next generation of the population is created.

4.6.2 Genetic algorithm (GA)

Genetic algorithms belong to the family of evolutionary algorithms. Genetic algorithms are defined by the probabilistic selection of the parents and the recombination. The mutation operates more in the background as it only gets executed with a relative low probability [Wei07]. The GA guarantees the reachability of every point in the search space and keeps the genetic diversity of the population. In this paper an individual is a set of three PID parameter tuples containing the speed-, steer- and acceleration-PID tuple. A set of individuals form a population. The current population produces new individuals that form the new generation. The individuals of the new generation are supposed to have a better average performance than the individuals from the previous generations. [KKP+08] There are two type of GAs, the Standard-GA or Generational-GA and the Steady-state-GA. The Standard-GA is defined by replacing the whole current generation by the next generation whereas the steady-state GA replaces one individual at a time. This means that as soon as a new individual is created, the individual with the lowest fitness value in the population gets replaced by the new individual. The Standard-GA performs better in optimization tasks [NB92], thus a Standard-GA is used to optimize the PID parameters.

Fitness function

During one lap the errors of steering, speed and acceleration are measured in each time step. The errors get squared before being multiplied by the time step ΔT. Summing up that expression for every time step, dividing it by the total time T_{total} results in the mean squared error

$$MSE = \frac{1}{T_{total}} \cdot \sum_{n} \|e(n)\|_2^2 \cdot \Delta T. \tag{4.11}$$

The fitness function is used to measure the performance of individuals during the process. The higher the fitness value the better the performance of one individual. The fitness function is the foundation of the optimization problem, the optimization direction is given by means of that function.

The fitness function used in this thesis is defined as the negative value of the mean squared error:

$$f = -MSE. \tag{4.12}$$

The time step ΔT in the nth step is defined as the difference between t_{n-1} and t_n. Summing up all the time steps results in the total time T_{total}. Thus the maximal fitness value is zero which means that the the summed up error values are equal to zero. When the fitness value is further away from zero means that it performs worse. The fitness value is then saved as an attribute in an individual, where is gets evaluated during the evolution process.

Evolution process

The *Genetic Algorithm* component gets as input the current lap and the error value on the individual which is currently evaluated. If the current lap input changes between two time

steps, the genetic algorithm knows that the evaluation of the current PID parameters is over and returns the next individual to the controller. After the fitness of every individual is evaluated, a new generation is created. The following steps are performed in order to generate a new population:

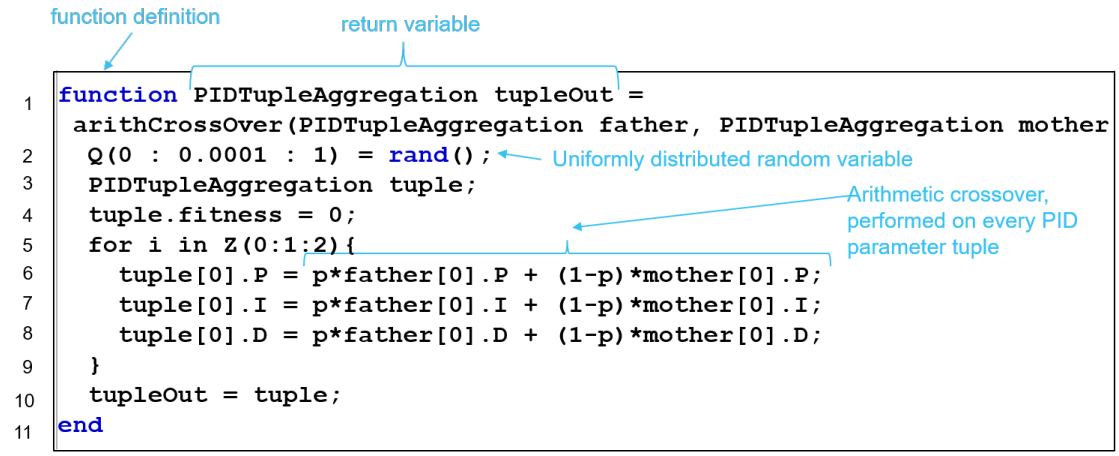
- Probabilistic parent selection
- Recombination
- Mutation

The first step consists of the probabilistic selection of an appropriate individual based on the fitness value. In this thesis the selection is done with tournament selection according to [Wei07]. The tournament selection function consists of one argument k, the population as array of input parameters and the winning individual as output parameter. The tournament selection function picks k individuals of the population at random and returns the individual with the best fitness. The selection process can be compared to the natural selection is nature. Only the fittest individuals in a population survive. When the African lion chases a horde of zebras, the fast zebras survive and the slow zebras become an easy prey. In context of the genetic algorithms we can say that faster zebras have a better fitness value than the slow ones, since they are more likely to survive and reproduce. An genetic algorithm uses that same principle by assigning a fitness value to each individual based on their performance and chooses artificially which individuals fit best to reproduce.

After selecting the best individuals from the population, the recombination step is executed with a certain probability p. The recombination step recombines two of the selected individuals using arithmetic crossover as prescribed by [Wei07] and depicted in Figure 4.12. With a probability of *(1-p)* no crossover is performed and one selected individual immediately arrives in the next step. The arithmetic crossover function gets two individuals, father and mother, as input parameters and returns one individual, a child. By means of a random uniformly distributed variable $\alpha \sim unif(0, 1)$, an arithmetic crossover between the two parent is performed. As depicted in equation 4.13 the child parameters are obtained by multiplication of the father's parameter with α and adding to the result the product of the mother's parameter with one minus α.

$$child = \alpha \cdot father + (1 - \alpha) \cdot mother \tag{4.13}$$

In the mutation step the individual mutates with a certain probability q. The mutation is performed according to the gaussian mutation in [Wei07]. The mutation step for one parameter is performed by means of a normal distributed number $\beta = \mathcal{N}(\mu, \sigma^2)$. Here, μ is defined as the mean and σ as the variance [CKS08]. For each parameter of one individual a normal distributed number is added to form the new parameter. The mean μ of the normal distribution is the original parameter value and the variance σ is equal to ten percent of the parameters upper bound.
The mutation step has two functionalities: firstly the exploitation, which finds the local optimum of a good solution candidate. Secondly it explores solutions in a more remote area in the search space to find a potentially better local optimum. The selection, recombination and mutation steps are repeated until a new population, with the same size as the old population, is reached. An overview of the genetic algorithm is illustrated in Figure 4.13.




```
1  function PIDTupleAggregation tupleOut =
    arithCrossOver(PIDTupleAggregation father, PIDTupleAggregation mother
2    Q(0 : 0.0001 : 1) = rand();
3    PIDTupleAggregation tuple;
4    tuple.fitness = 0;
5    for i in Z(0:1:2){
6      tuple[0].P = p*father[0].P + (1-p)*mother[0].P;
7      tuple[0].I = p*father[0].I + (1-p)*mother[0].I;
8      tuple[0].D = p*father[0].D + (1-p)*mother[0].D;
9    }
10   tupleOut = tuple;
11 end
```

Figure 4.12: Function performing arithmetic crossover for two PIDTupleAggregation structs.

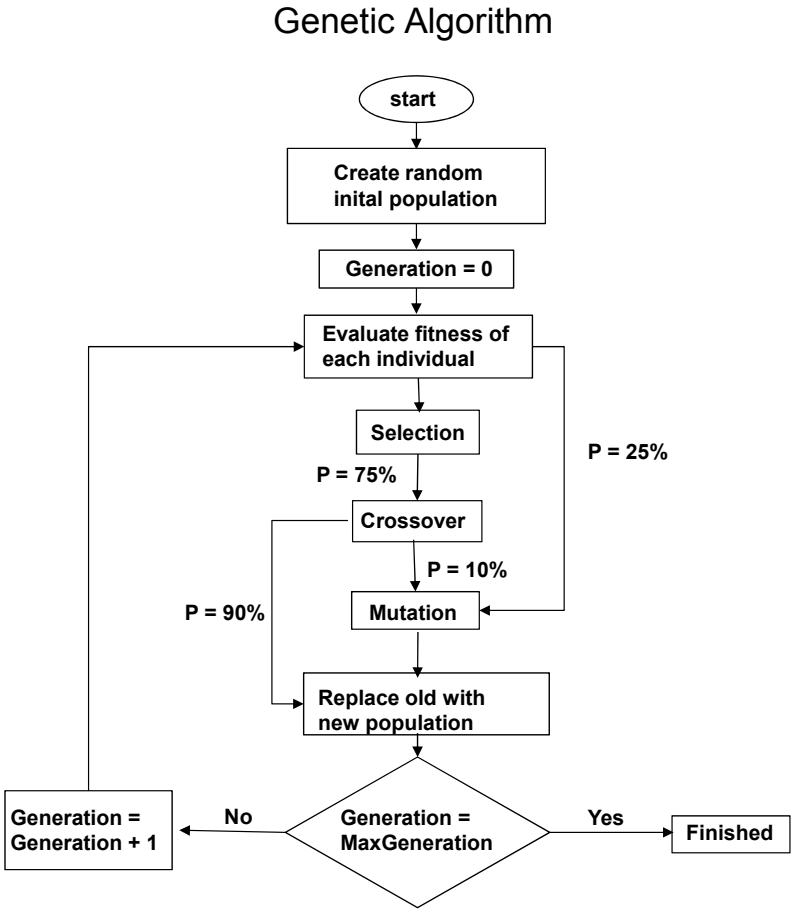


Figure 4.13: Main process of the deployed genetic algorithm inspired by [RK15].

Chapter 5

Alternative Control Strategy

5.1 Model Predictive Control

5.1.1 General functionality

Model Predictive Control (MPC) originated in the late seventies and made considerable progress since then. In the recent years MPC schemes have established themselves as a popular control strategy. The term Model Predictive Control does not refer to a specific control strategy but a very ample range of control methods. MPC makes explicit use of a model of the process to make predictions about the real systems behavior. The ideas appearing in greater or lesser degree in all MPC controllers are:

- Start at time t and predict a number of future output signals by means of the system model.
- Build a cost function based on the future output and control signals and optimize with respect to the control signal.
- Input the control signal to the system [Pal16].

In Figure 5.1 the basic functionality is depicted. The output is predicted by means of the process model. The reference trajectory represents the target values for that moment in time. In every time step k, an optimization problem is solved. The point of optimization is to find a control input such that the cost function is minimal. That function describes how good the predicted output fits the target output. To predict the output the state space model can be used.

The general optimization function for the state space model is defined as

$$\begin{aligned} \underset{u}{\text{minimize}} \quad & \sum_{i=1}^{H} J(u_i, z_i) \\ \text{subject to} \quad & z_{i+1} = f(z_i, u_i, p) \end{aligned} \tag{5.1}$$

Whereas u_i is a vector of control input signals at time i, H is the horizon, z_i is the state of the system at time i. J is the cost function which needs to be minimized with respect

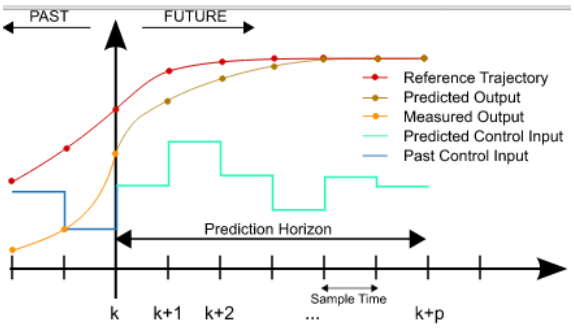


Figure 5.1: Working principle of a MPC Controller.

to u. The horizon describes how many steps to consider in the future. When the horizon is chosen very big, the controller is looking further in the future, which improves the control signals. However the bigger the horizon, the more computational intense will the optimization problem be.

5.1.2 State Space model in discrete time

The state space model describes a system of first order differential equations. These equations describe the relation between the state z, the input values u, the time step i and the parameters p. The output values y are determined by the algebraic equations containing z, u, i and p. The general time-discrete state space model looks like the following:

$$z_{i+1} = f(z_i, u_i, p) \tag{5.2}$$

$$y_i = h(z_i, u_i, p) \tag{5.3}$$

[Mar15]

5.1.3 Bicycle model

The following section is inspired by [BFK+05], [Pal16] and [Raj11]. In order to predict the future outputs a system model is needed. To describe the dynamics of the car the bicycle model is used. The bicycle model is a good trade-off between accuracy and computational intensity. The simplification in the Bicycle model is to model the two front wheels and the two rear wheels as one front and one rear wheel. In the Figure 5.2 the simplified vehicle model is depicted. The following nomenclature is used:

- F_l , F_c are the longitudinal and lateral tire forces
- δ is the wheel steering angle
- l_r, l_f are the distances of front and rear wheels from the center of gravity (CoG)
- X, Y is the position of the center of gravity in the global coordinate system
- ψ is the yaw angle of the car

- α is the slip angle
- v_{lf}, v_{cf} are the longitudinal and lateral wheel velocities of the front wheel
- F_a is the air drag
- F_r is the rolling resistance
- m is the mass of the vehicle
- I is the vehicle inertia around the z-axis
- F_x, F_y are the longitudinal and lateral forces acting on the Center of Gravity of the vehicle.
- F_z is the vertical load acting on the wheels of the vehicle
- C is the tire stiffness parameter
- v_f is the resultant velocity vector
- θ is the angle of the resultant velocity vector
- D_r is the rolling resistance coefficient
- C_D is the air drag coefficient
- A is the maximum cross sectional area of the vehicle
- ϕ is the density of the air
- $(.)_f$, $(.)_r$ means the front respectively the rear wheel

The states of the vehicle can be expressed as:

$$\dot{X} = v_x \cos(\psi) - v_y \sin(\psi) \tag{5.4}$$

$$\dot{Y} = v_x \sin(\psi) - v_y \cos(\psi) \tag{5.5}$$

$$\dot{\psi} = \frac{v_x}{l_f + l_r} \tan(\delta) \tag{5.6}$$

$$\dot{v_x} = \frac{1}{m}(F_x + m v_y \dot{\psi} - 2F_{cf} \sin(\delta) - F_a - F_r \tag{5.7}$$

$$\dot{v_y} = -v_x \dot{\psi} + \frac{2}{m}(F_{cf} \cos(\delta) + F_{cr}) \tag{5.8}$$

$$\ddot{\psi} = \frac{2}{I}(l_f F_{cf} \cos(\delta) - l_r F_{cr}). \tag{5.9}$$

According to Newtons second law, $\sum F = ma$ the equations (5.7) (5.8) and (5.9) can be derived.
By means of equations (5.4) and (5.5), the change of position in the global coordinate system can be expressed. Equation (5.6) denotes how the yaw rate is calculated. The forces F_x and F_y acting on the CoG, can be expressed in terms of the longitudinal and lateral tire forces F_l and F_c:

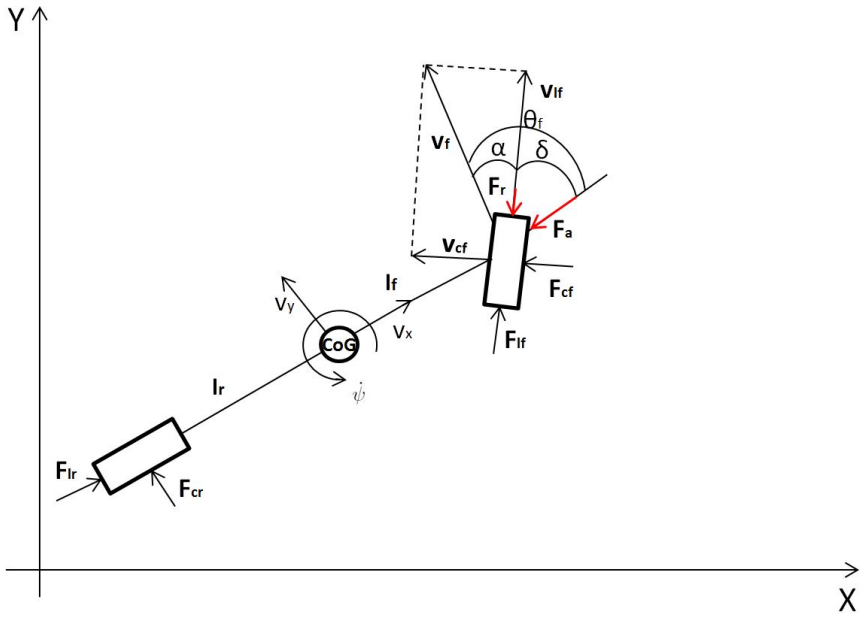


Figure 5.2: The bicycle model.

$$F_x = F_l \cos(\delta) - F_c \sin(\delta) \tag{5.10}$$

$$F_y = F_l \sin(\delta) + F_c \cos(\delta) \tag{5.11}$$

F_l and F_c can be furthermore expressed as a function of the slip angle α, the friction coefficient μ, and the force F_z. Assuming a small slip angle, the tire forces can be approximated with linear equations:

$$F_{lf} = C_f \alpha_f \tag{5.12}$$

$$F_{lr} = C_r \alpha_r. \tag{5.13}$$

The slip angles can be expressed by the following equations:

$$\alpha_f = \delta - \theta_f \tag{5.14}$$

$$\alpha_r = -\theta_r. \tag{5.15}$$

The angle of the resulting velocity can be calculated according to:

$$\theta_f = arctan(\frac{v_y + l_f \dot{\psi}}{v_x}) \tag{5.16}$$

$$\theta_r = arctan(\frac{v_y + l_r \dot{\psi}}{v_x}). \tag{5.17}$$

Assuming the road is flat, the rolling resistance F_r can be expressed as:

$$F_r = D_r mg. \tag{5.18}$$

The air drag force can be calculated according to [Raj11]:

$$F_a = \frac{1}{2} C_D A \phi (v + v_{wind})^2. \tag{5.19}$$

Since the bicycle model is non-linear the MPC controller need to be either linearized which could result an inaccurate model or the non-linear model could be used which makes the application computationally more intense and more complicated. Since both methods require much effort it was decided not to implement that approach because it would go beyond the scope.

5.1.4 Summary

On one hand, the concepts of the MPC approach are intuitive and at the same time tuning is relatively easy, thus no complex tuning algorithm is needed. MPC also has the ability to handle constraints and multi-variable processes.
On the other hand there are also some drawbacks. The MPC needs to solve an optimization problem in every time step, which makes the approach computationally expensive. Another drawback is the need for an appropriate model of the process. The control algorithm is based on a prior knowledge of the model. The benefits obtained from the MPC method will be affected by the discrepancies between the real process and the model used. [CA13].

Chapter 6

Experiments and Results

6.1 Parameter Tuning: Experiment setup

The experiments were done using the *TORCS* simulator. Since *TORCS* is a racing car simulator the tracks are circuits. The track which was chosen to optimize the parameters on has a width of 15 meters and a length of 2057.56 meters. The physics and the engine simulation is called with a frequency of 500 Hertz whereas the autonomous driving components are called with a frequency of 50 Hertz. The car that was used was a Chevrolet Corvette T-Top [BWS13].

Four experiments were executed. The first experiment consisted of measuring the average error of each generation using the genetic algorithm on all three PID tuples.

In the second experiment the genetic algorithm was applied in a first phase to train the speed and acceleration PID parameters and in a second phase applied to train the steering PID parameters. Every parameter which had not been optimized in one of the two phases was set to a constant value. This means that during phase one, the steering PID parameters were set to constant values and in the second phase, the acceleration and speed PID parameters were set to constant values. Furthermore the acceleration error and the speed error connections to the *Error-Summation* component were dropped in the first phase, to only evaluate the steering error in the fitness function. In the second phase the steering error connection to the *Error-Summation* component was dropped in order to only evaluate the speed and acceleration error in the fitness function.

The error

$$e = x_{target} - x_{measured} \tag{6.1}$$

is defined as the target value subtracted by the measured value. Thereby x may serve as a placeholder for each controlled variable such as the velocity, acceleration, and the steering angle.

In the third experiment it is tested how good the noise filtering using a low-pass filter works. Therefore the current position is distorted and filtered.

In the fourth experiment different noise levels are applied to the current position, the target position, the car yaw angle, the gas pedal values, the steering angle, and the current speed. The genetic algorithm trained the PID parameters for 5 generations for each noise level before recording the MSE for the best PID parameters.

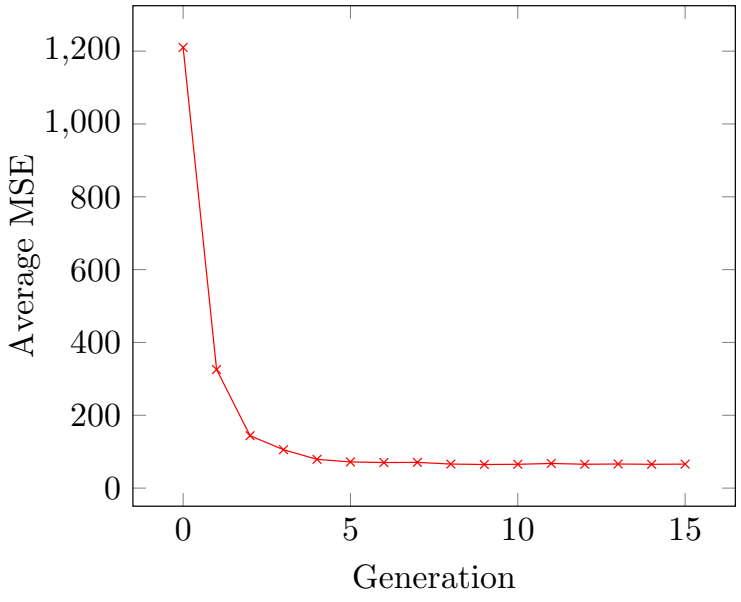
6.1.1 Training all parameters together

To efficiently measure the acceleration error and speed error, the target speed is changed every thirty seconds alternating between ten and seventeen meters per second. This means that every thirty seconds the target speed changes by seven meters per second alternating up and down. Furthermore, AWGN was applied to the current speed, the gas-pedal value, the current steering angle and the car yaw-angle. To have the same conditions for every individual in the population, the errors of one individual is measured during one whole lap (see section 4.6.2). This is extremely important regarding the steering error. If it is not measured under the same circumstances, it may happen that an individual which has a good performance, has a worse fitness value than another parameter set which has a worse performance, but had an "easier" path. Since the steering error is about two orders of magnitude smaller than the acceleration and the speed error, it gets multiplied by hundred before being inputted to the fitness function.

In the Figure 6.1 the average MSE over the generations is illustrated.

The average MSE is calculated as

$$AverageMSE = \frac{1}{N_{individuals}} \sum_{i=1}^{N_{individuals}} MSE(i). \tag{6.2}$$



(a) Average MSE

Figure 6.1: The MSE over generation 0 to 15. On the left side there is the average MSE of the steering, acceleration an speed. On the right side there is the MSE of the acceleration and speed only.

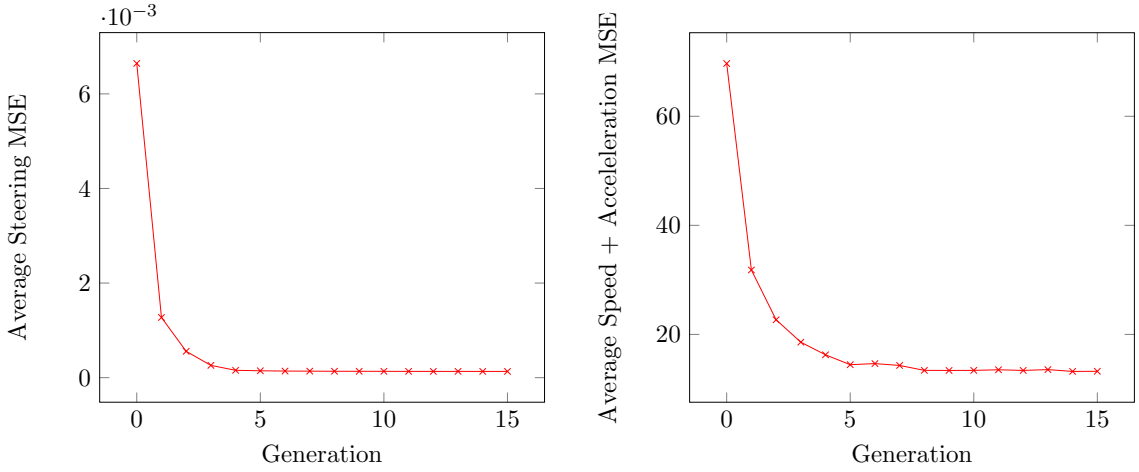
6.1.2 Training in two sets of parameters

This experiment is separated in two phases. In the first phase the acceleration and speed PID parameter tuples are trained. In the second phase the steering PID parameter tuple

is trained. Furthermore, AWGN was applied to the current speed, the gas-pedal value, the current steering angle and the car yaw-angle. In phase one the steering PID parameter tuple needs to be fixed. In order to fix that tuple, the best PID steering tuple of the first experiment is taken. This tuple is then inputted to the PID component which is responsible for the steering. Furthermore the connection between the *Subtraction* component responsible for the steering error and the *Error-Summation* component is dropped. Thus the steering error will not influence the fitness value. The architecture can easily be modified to train the parameters separately in both phases, therefore requirement (R5) is fulfilled.
It is also assured that the steering PID tuple does not bias the speed and acceleration due to a poor choice of parameters. To efficiently measure the acceleration error and speed error, the target speed is changed every thirty seconds alternating between twenty and thirty meters per second.
On the left hand side, the average MSE of the steering error is depicted and on the right hand side of Figure 6.2 the average mean squared error of the speed and acceleration over the generations is illustrated. The MSE of the acceleration and the speed is calculated by simply adding both MSE values.

$$MSE_{Accel\&Speed} = MSE_{Accel} + MSE_{Speed} \tag{6.3}$$

Note that the absolute values of both experiments can not be compared since the meta data of both experiments is very different.



(a) Steering MSE

(b) Average Speed and Accel MSE

Figure 6.2: On the left side there is the average MSE of the steering angle. On the right side there is the average MSE of the speed and the acceleration.

In the second phase of the experiment only the steering PID parameters are trained. Thus we input constant parameter tuples to the PID controllers responsible for the speed and the acceleration. Furthermore the connections between the two *Subtraction* components and the *Error-Summation* component is dropped. Thus only the steering error is evaluated by the fitness function. On the left hand of Figure 6.2, the mean squared error of the steering PID parameter tuple is depicted.

In both experiments, the performance in terms of the MSE is improved over the generations, thus requirement (R4) is fulfilled.

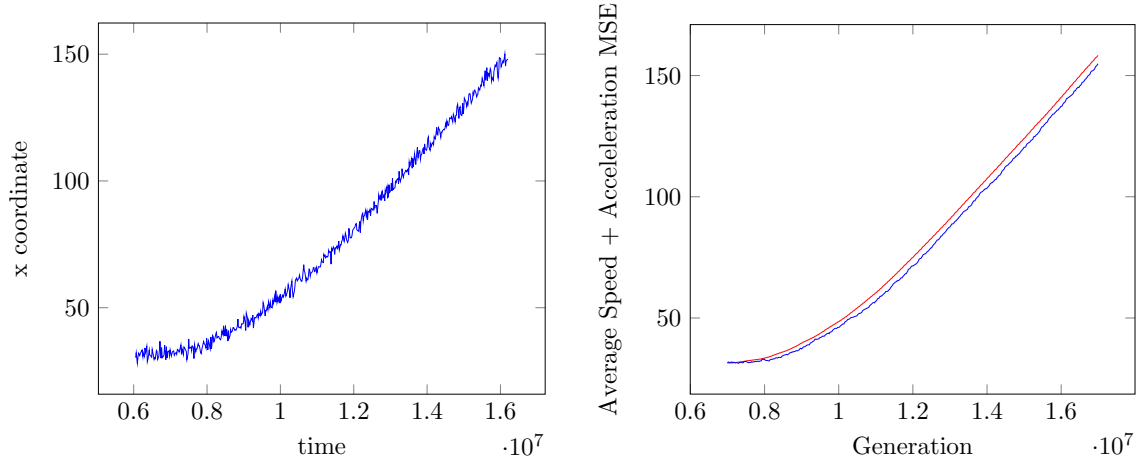
6.2 Noise filtering

This experiment consists of testing the filtering function of the low-pass filter applied the noisy sensor data. Three instances of the current position were recorded: the value retrieved from the simulation, the distorted value, and the filtered value. These values were recorded for the first ten seconds of the first lap. The PID parameters were fixed to the optimal parameters found in 6.1.1. As smoothing factor $\alpha = 0.08$ is used. Thus the cutoff frequency f_c is

$$f_c = \frac{\alpha}{(1-\alpha)2\pi\Delta T} \approx 0.6919Hz \tag{6.4}$$

with $\Delta T = \frac{1}{50}[s]$ as the sampling frequency of the TORCS is $50Hz$.

AWGN with a variance of $\sigma = 1$ was applied to the current speed, the gas-pedal value, the current steering angle and the car yaw-angle. The x-coordinate of the current position was recorded. On the left side of Figure 6.3 the distorted value is depicted. On the right side of the same Figure, there is the plot of the values retrieved from the simulator in red, and the filtered values in blue.



(a)Disturbed data (b) Filtered and simulator data

Figure 6.3: On the left there is the disturbed x-position of the car, and on the right there is the filtered data in blue and the simulator data in red.

It can be seen that the low-pass filter is able to smooth out most of the noise in the sensors. Thus the controller is able to compensate sensor imperfections, thus requirement (R3) is fulfilled.

6.3 Noise level analysis

In the following experiment the following values retrieved from the simulator were disturbed using AWGN: speed, gas pedal value, the steering angle, the yaw-angle of the car, the current position and the target position. The working principle of the AWGN is explained in 2.2.1.

The experiment was executed for five different noise levels. For each of these noise levels, the PID parameters were trained for five generations. After the training the MSE of

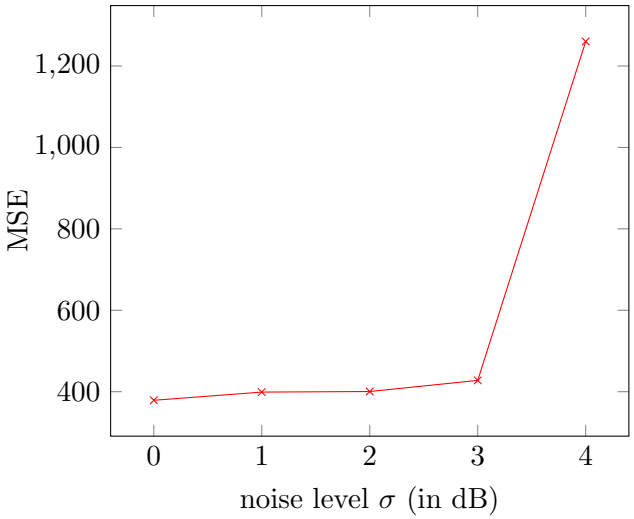


Figure 6.4: The MSE of the best individual of the fifth Generation for different noise levels.

the best PID parameter sets for each noise level was measured during one lap with their respective noise level.

In the Figure 6.4 the MSE with respect to the noise level is depicted. Thus the application is able to handle sensor imperfections and the limits of the maximal SNR can be accessed. Training the three PID parameter sets was only feasible until the forth noise level, 4 dB. For the following noise level, 5 dB is was not possible to train the PID parameters because the car was so unstable that it kept crashing.

Chapter 7

Conclusion

Due to the highly modular connector and component language MontiCAR it is very time efficient to take a shot at several different approaches. The components which are abstracted from the implementation itself can be exchanged in an uncomplicated way. Thanks to these properties a feasible control strategy can be easily developed. In the first experiment the average MSE has converged and optimal PID values were found. In the second experiment the architecture could be modified easily to optimize the steering error and the acceleration and speed error separately or training the parameters with several different noise levels. Thanks to the data adapter realized with the middle-ware OpenDaVinci the Controller is not restricted to TORCS but can be deployed to arbitrary simulators or even a real car.

Chapter 8

Future Work

An extension of the thesis would be trying to find the optimal smoothing factor for each of the low-pass filters, to have an optimal cutoff frequency for each of the sensors signals. This could be realized with the already implemented genetic algorithm. The fitness function would be the same as the one defined in section 4.6.
Furthermore the alternative control strategy MPC presented in 5.1 could be implemented and tested. MPC controllers have the advantage of being easy to tune compared to a PID controller. However, there is a need for an accurate model of the system and an efficient optimization algorithm since the approach needs to execute an optimization in every time step. Nevertheless MPC can be a feasible control strategy for vehicles [BFK+05].

Since the current trajectory planing component always chooses the target point in the middle of the road, the autonomous car does not support choosing or changing a specific lane. A better heuristic to plan the trajectory could be worked out in the future as a nice-to-have feature. Staying on a specific lane or lane changing could be implemented additionally. The controller is not able to detect other vehicles on the road. Collision avoidance and overtaking other vehicles could be added to the application in the future as well. Another possible extension would be testing the controller in different simulators, or even in a real vehicle. However testing on real vehicles is expensive and comes with additional difficulties.

Bibliography

[ÅH95] Karl Johan Åström and Tore Hägglund. *PID controllers: theory, design, and tuning*, volume 2. Isa Research Triangle Park, NC, 1995.

[Ber15] Christian Berger. Opendavinci middleware. `https://github.com/se-research/OpenDaVINCI`, 2015.

[Ber16] Christian Berger. An open continuous deployment infrastructure for a self-driving vehicle ecosystem. In *IFIP International Conference on Open Source Systems*, pages 177–183. Springer, 2016.

[BFK+05] Francesco Borrelli, Paolo Falcone, Tamas Keviczky, Jahan Asgari, and Davor Hrovat. Mpc-based approach to active steering for autonomous vehicle systems. *International Journal of Vehicle Autonomous Systems*, 3(2-4):265–291, 2005.

[BW14] Christophe Guionneau Christos Dimitrakakis Rémi Coulom Andrew Sumner Bernhard Wymann, Eric Espié. TORCS, The Open Racing Car Simulator. http://www.torcs.org, 2014.

[BWS13] Christophe Guionneau Christos Dimitrakakis Remi Coulom Bernhard Wymann, Eric Espi´e and Andrew Sumner. Torcs, the open racing car simulator. `https://www.torcs.org`, 2013.

[CA13] Eduardo F Camacho and Carlos Bordons Alba. *Model predictive control*. Springer Science & Business Media, 2013.

[CKS08] Erhard Cramer, Udo Kamps, and Ansgar Steland. *Grundlagen der Wahrscheinlichkeitsrechnung und Statistik*. Springer, 2008.

[CT12] Thomas M Cover and Joy A Thomas. *Elements of information theory*. John Wiley & Sons, 2012.

[dar07] Darpa grand challenge. http://archive.darpa.mil/grandchallenge/, 2007.

[EK16] Bernhard Rumpe Michael von Wenckstern Evgeny Kusmenko, Alexander Roth. Modeling architectures of cyber-physical systems. 2016.

[H+86] J Stuart Hunter et al. The exponentially weighted moving average. *J. Quality Technol.*, 18(4):203–210, 1986.

[KKP+08] Jin-Sung Kim, Jin-Hwan Kim, Ji-Mo Park, Sung-Man Park, Won-Yong Choe, and Hoon Heo. Auto tuning pid controller based on improved genetic algorithm for reverse osmosis plant. *World Academy of Science, Engineering and Technology*, 47(2):384–389, 2008.

[KR77] JF Kaiser and WA Reed. Data smoothing using low-pass digital filters. *Review of Scientific Instruments*, 48(11):1447–1457, 1977.

[Mar15] Wolfgang Marquardt. *Notizen zu der Lehrveranstaltung Simulationstechnik*. 2015.

[NB92] David Noever and Subbiah Baskaran. Steady state vs. generational genetic algorithms: A comparison of time complexity and convergence properties. *Preprint series*, pages 92–07, 1992.

[New17] Peter Newman. Fully autonomous car update: How self-driving cars care poised to move into the mainstream and upend the automotive industry. `http://www.businessinsider.de/the-fully-autonomous-car-update-2017-2?r=US&IR=T`, 2017.

[Pal16] Mia Isaksson Palmqvist. Model Predictive Control for Autonomous Driving of a Truck. Master's thesis, RKTH Royal Institute of Technology, Stockhold, Sweden, 2016.

[Raj11] Rajesh Rajamani. *Vehicle dynamics and control*. Springer Science & Business Media, 2011.

[RK15] Ankush Rathore and Mahendra Kumar. Robust steering control of autonomous underwater vehicle: based on pid tuning evolutionary optimization technique. *International Journal of Computer Applications*, 2015.

[Sel] Driverless car market watch. `http://www.driverless-future.com/`.

[SG15] Peter Stanchev and John Geske. Autonomous cars. history. state of art. research problems. In *International Conference on Distributed Computer and Communication Networks*, pages 1–10. Springer, 2015.

[SM13] Berhard Rumpe Shahar Maoz, Jan Oliver Ringert. An extensible component and connector architecture description infrastructure for multi-platform modeling. 2013.

[Wei07] K. Weicker. *Evolutionäre Algorithmen*. XLeitfäden der Informatik. Vieweg+Teubner Verlag, 2007.

[Wor16] Andreas Wortmann. An extensible component and connector architecture description infrastructure for multi-platform modeling. 2016.