Industrial Internship

Robin Verschueren

Authors	Robin Verschueren
Date	November 15, 2013
Version	Final
Confidentiality	Confidential

Contents

Preliminaries

Student: Robin Verschueren (2nd Master Wiskundige Ingenieurstechnieken)

Company: LMS,
Interleuvenlaan 68, Researchpark Haasrode Z1, 3001 Leuven

Company contact: Stijn De Bruyne

Supervisor: Karl Meerbergen

Period: July 22nd until August 30th

1 Introduction: LMS, a Siemens Business

1.1 The birth and growth of LMS

LMS was founded in 1980 as a spin-off of the KU Leuven. Initially LMS only offered testing as a service, but later also the development of testing software started. More recently, in the mid-1990s, simulation software was added to the product portfolio of LMS. Through time LMS acquired several companies to make these developments possible. For example, recently (25 Aug 2011), LMS acquired a 60% controlling majority position of SAMTECH, the Liège-based European provider of Computer Aided Engineering (CAE) and structure analysis software. More recently still, at the end of 2012, the German holding Siemens acquired LMS as the 'Test and Simulation' component of Siemens PLM (Product Lifecycle Management)[3].
LMS currently offers the following unique combination of products and services:

- testing systems,
- mechatronic simulation software,
- engineering services.

LMS has become a profitable company, operating in over 30 key locations with about 1200 employees worldwide. In 2011 the company achieved a turnover of about €160 million [2]. LMS has grown to become a worldwide leader in engineering innovation. With multi-domain and mechatronic simulation solutions, LMS addresses the complex engineering challenges associated with intelligent system design and model-based systems engineering. They have become the partner of choice of more than 5000 manufacturing companies worldwide, spread over the automotive, aerospace and mechanical industry segments.

1.2 Activities

The leading partner in testing and mechatronic simulation in the automotive, aerospace and other advanced manufacturing industries, helps customers get better products to market faster.

1.2.1 Testing

LMS has pioneered many innovative techniques in high-end structural and NVH (noise, vibration & harshness) testing over the years. The full portfolio of LMS testing solutions includes transfer path analysis, rotating machinery, structural and acoustics testing, environmental testing, vibration control, report and data management. On the hardware side, the LMS SCADAS family ranges from compact mobile units, autonomous smart recorders, dedicated durability solutions up to high-channel count laboratory systems.

1.2.2 Mechatronic simulation

Over the last decade, LMS has developed an integrated hybrid process solution simulation, enhanced by test. With LMS mechatronic simulation software and solutions, critical functional performance attributes are simulated and designed upfront in the development process. This process has enabled LMS customers to slash development times by 30-50%. This is not only a tremendous advantage in terms of faster time to market, it also significantly reduced risks and costs. Today, LMS tries to take simulation a step further, and focuses on energy management and emission reduction, as well as the

almost exponential explosion of electronics and controls in a variety of products, such as intelligent vehicles.

1.2.3 Engineering services

The engineering service team knows how to support all aspects of the development process, from conceptual design over detailed component development to test-based product refinement and validation. LMS Engineers master a variety of complex domains: noise and vibration, strength and durability, safety and crash, kinematics and dynamics, vehicle handling, eco-engineering and mechatronic simulation. The LMS Engineering Services centers in Europe, the USA and Asia combine state-of-the-art testing facilities with an extensive simulation infrastructure.

2 The internship

2.1 Task Description

The goal of the industrial internship is to create a setup with automated model race cars (scale 1:43) driving on a race track for demonstrating purposes towards potential customers in the automotive sector. Ultimately, a model based controller will be conceived, performing both the planning and tracking of the race car trajectory.
The very first task was to get familiar with the hard/software in place, as well as debugging the Vision Processing Unit (VPU). Next, a simple proportional controller was developed and tested on the track to validate the successful integration of the different elements in the closed loop control system. An essential step on the way to a model based controller is the identification of the vehicle model. In order to do this, existing vehicle models were examined, and experiments were carried out in order to identify the parameters of the model. Finally, a Model Predictive Controller (MPC) for tracking a reference trajectory was implemented and experimentally validated on the race track.

2.2 Role in company activities

The last few years, Advanced Driver Assistance Systems (ADAS) have been introduced in passenger cars, eg. semi-automated parking and pre-crash systems. The next step would be autonomously driving cars, which will be achieved not so far in the future. The internship is a part of the engineering services activities of the company that focus on the usage of model-based control techniques for control applications in the next-generation vehicles.

2.3 The setup

Small scale radio controlled cars (RC-cars) really came of age with the introduction of the Kyosho dNaNo model race cars in 2008. These cars will be driven automatically around a track, controlled by a computer, via a data acquisition card (DAQ) which drives the Kyosho RC-controller. The feedback in terms of the position of the cars will be provided by an infrared camera, which senses the reflection of some small markers placed on the race cars and sends these images to the computer. The entire architecture is shown in Fig. 1. More technical details can be found in appendix A.
In the internship, two different tracks were used. The first is a simple oval track (Fig. 2a), the other is a more sophisticated and bigger track, with a chicane, a U-turn and a longer straight section to gain speed (Fig. 2b). We will call this the final track.

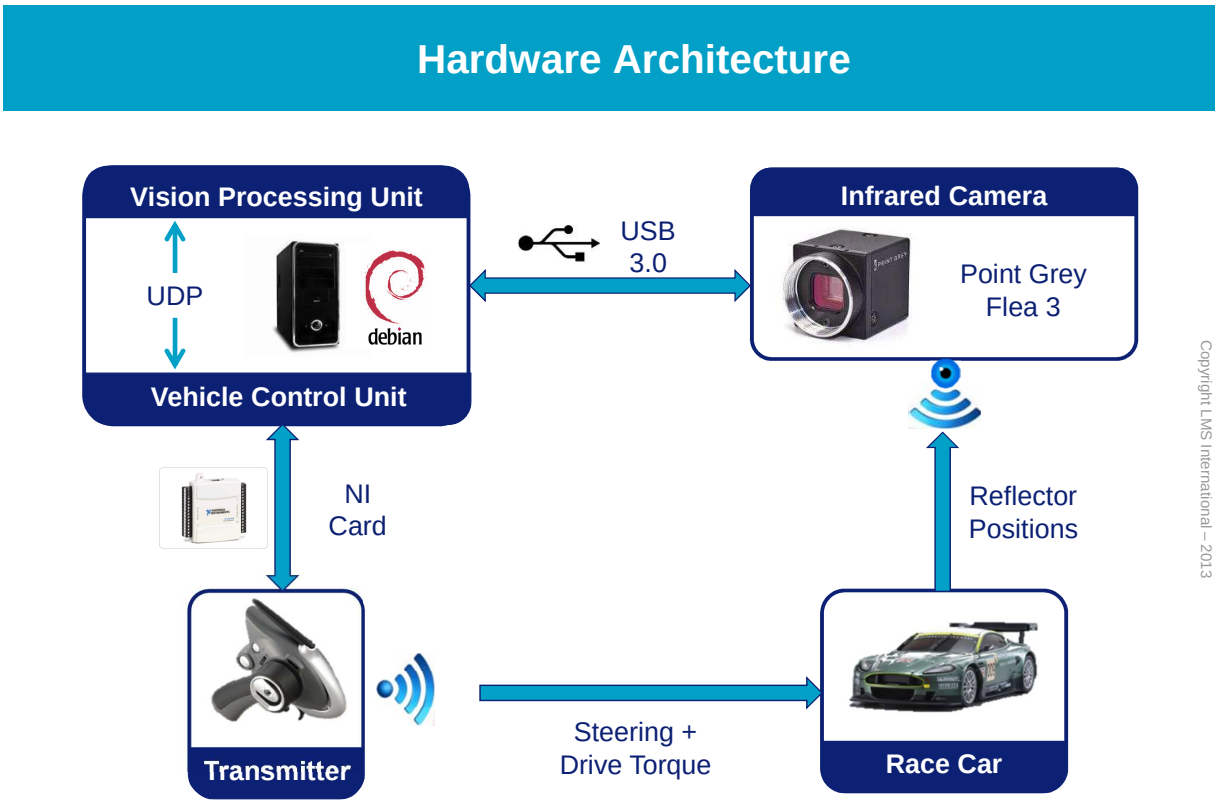
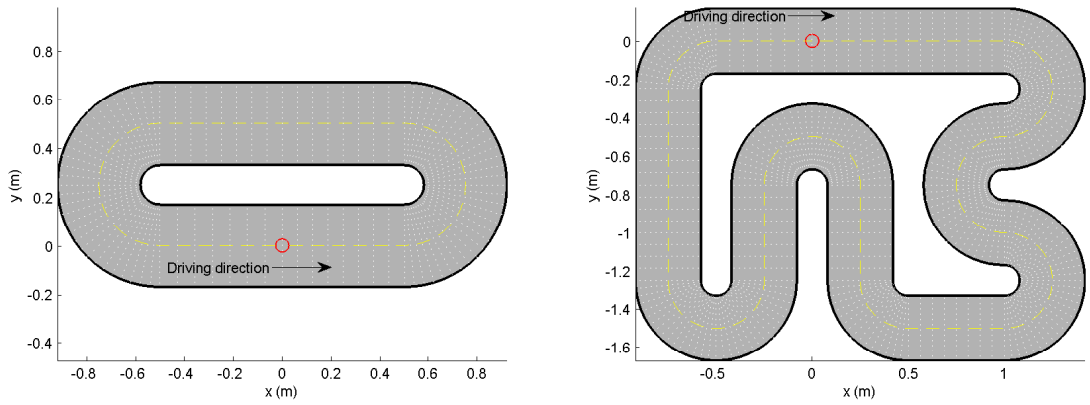


Figure 1: *The automated race car driving setup.*



(a) *The oval track* **(b)** *The final track*

Figure 2: *The race tracks used in the experiments.*

2.4 The project: planning

What follows is a description of the activities by so called ”milestones”. Each milestone is planned to take up two weeks, yielding a six week planning. The milestones are:

- Milestone 1: The car drives around a simple race track using proportional feedback control. The intern is familiar with the setup hardware and software.

- Milestone 2: Integration with Simulink. The car drives around a more difficult race track, still using proportional control. An appropriate vehicle model is identified.
- Milestone 3: Design of an MPC controller for the tracking of a predefined trajectory.

2.5 The project: towards Milestone 1

At the start of the internship, the following components of the setup were not fully functional:

- Camera calibration
- Kalman filter
- The communication between the Vision Processing Unit (VPU) and the Vehicle Control Unit (VCU)
- The communication between the VCU and the Kyosho RC-controller

These components were fixed during the first two weeks of the internship. While proceeding towards milestones 2 and 3, existing components were improved and new components were added.

2.5.1 Camera calibration

The calibration of the camera is crucial for an accurate vision system. It aligns the 3D real world with the 2D internal images. As a model for our camera, we use a *pinhole* model. The calibration of the camera consists of two steps: an intrinsic calibration and an extrinsic calibration. The intrinsic calibration deals with the camera distortions, the extrinsic calibration makes sure that the coordinate system of the camera is aligned with the world coordinate system. A detailed description on pinhole models and camera calibration can be found in [4].

Intrinsic calibration Intrinsic calibration of a camera deals with distortions. The distortion of the camera can be split in two parts: linear distortion and nonlinear distortion. For the linear distortion, the parameters that need to be identified are in the so called camera matrix:

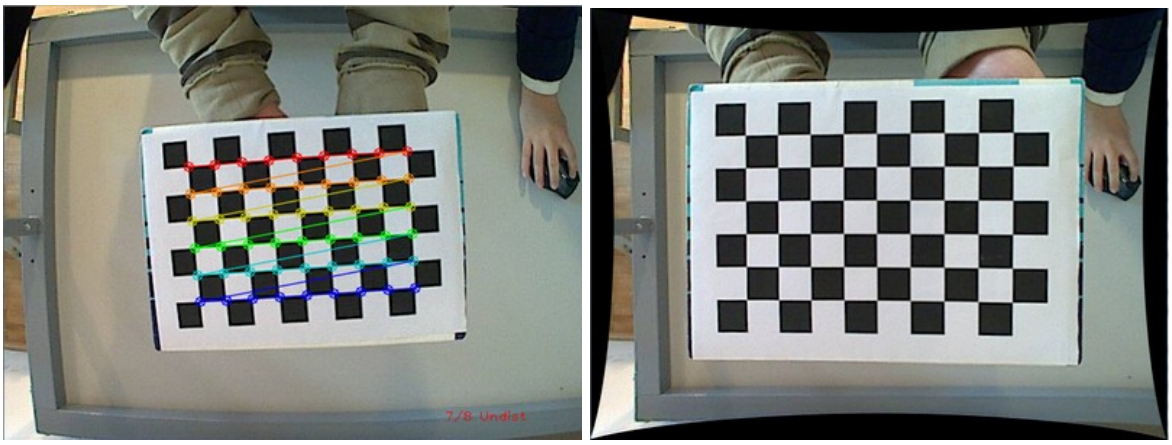
$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}, \tag{1}$$

where X, Y and Z are the world coordinates, x and y are the camera coordinates. The presence of w is because of the use of a homography coordinate system (thoroughly explained in [5]). The unknown parameters to identify are f_x, f_y (camera focal lengths for each direction) and c_x, c_y (the optical center expressed in pixel coordinates).
The nonlinear distortion by the camera comes mainly from the ”barrel effect”, or ”fish-eye effect”, as shown in Fig. 3. These distortions can be corrected as follows:

$$x_{corrected} = x(1 + k_1 r^2 + k_2 r^4) \tag{2}$$

$$y_{corrected} = y(1 + k_1 r^2 + k_2 r^4) \tag{3}$$

For the computation of both the linear and nonlinear distortion coefficients, we use C++ code provided by ETH in Zürich. It is based on the OpenCV code that can be found at [5], and uses the checkerboard



(a) *Distorted. Note the nonlinear barrel effect.* **(b)** *Undistorted.*

Figure 3: *Nonlinear distortion*

method (as shown in Fig. 3). It consists of moving an image of a checkerboard around and letting the camera take several pictures. The coefficients are then calculated automatically afterwards.
This is the point where the camera calibration failed: the computation of the nonlinear distortion coefficients. The undistorted images gave worse results than the original distorted images. The solution to this problem was to keep the coefficients of the linear distortion, and setting all the nonlinear distortion coefficients to zero. This was tested by taking a strip of reflecting material of length $50.5\,cm$, and measuring the number of pixels the camera sees, for different regions of the camera scope. The resolution of the camera is then computed by dividing the length of the strip by the number of pixels. The results are shown in Table 1. Clearly, the results are satisfactory, because the resolution needed is approximately $5\,mm/pixel$, so the nonlinear distortion can safely be ignored.

region	resolution $(mm/pixel)$
middle	1.94
top	1.95
right	1.97
bottom	1.96
left	1.98
minimum	1.94
maximum	1.98
mean	1.96

Table 1: *Resolution of the camera for different regions in the camera scope.*

Extrinsic calibration The extrinsic calibration was already working from the beginning, so it was not a part of this internship.

2.5.2 Kalman filter

A Kalman filter is an algorithm to provide estimates of unknown variables, distorted by noise. In our setup, the camera images are processed by the Vision Processing Unit (VPU) on the real-time Debian computer. However, these measurements are not exact; they contain noise and thus need to

be filtered. The algorithm works in two steps: first a prediction of the next state is made, based on a model. As a model, we used a point mass moving at a constant speed. The possible acceleration or deceleration is considered as noise. This is a very basic model, but good enough for our purposes. The model in the state space form is:

$$\begin{bmatrix} \dot{x} \\ \dot{v}_x \\ \dot{y} \\ \dot{v}_y \\ \dot{\psi} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ v_x \\ y \\ v_y \\ \psi \\ \omega \end{bmatrix} + \begin{bmatrix} 0 \\ a_x \\ 0 \\ a_y \\ 0 \\ \dot{\omega} \end{bmatrix}, \tag{4}$$

$$y = \begin{bmatrix} x \\ y \\ \psi \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ v_x \\ y \\ v_y \\ \psi \\ \omega \end{bmatrix}. \tag{5}$$

In the next step, this prediction is corrected by insertion of the measurement. A trade-off is made between prediction and measurement, based on the covariance matrices of the process noise and measurement noise (the Q and R matrices). In short:

Prediction:

$$\hat{x}(k|k-1) = A\hat{x}(k-1|k-1) + Bu(k) \tag{6}$$

Update:

$$\hat{x}(k|k) = \hat{x}(k|k-1) + W(k)(z(k) - C\hat{x}(k|k-1)), \tag{7}$$

where $\hat{x}(k|k-1)$ means "the estimate of x at time k with measurements up to time $k-1$", and where $W(k)$ is the Kalman gain, calculated from the Q and R matrices at each time step. A detailed description of the Kalman filter is given in [6]. The implementation of the Kalman filter was done using the OpenCV libraries, which provide some routines for the prediction and update steps.
The problem with the Kalman filter was that it acted very slowly, and ever more slowly when the filter ran for a longer time. This seemed a very strange bug, but a solution was found quickly: it was a type error. The framerate of the camera ($100Hz$) was denoted in the C++ code as

```
#define FRAMERATE 100.
```

In the Kalman filter code, the camera frequency was calculated as `float ts = 1/FRAMERATE;`. However, with `FRAMERATE` being an integer, `ts` was always zero. Altering the macro definition to

```
#define FRAMERATE 100.0.
```

solved the problem immediately.

2.5.3 The communication between VPU and VCU

The camera images processed by the Vision Processing Unit (VPU) need to be sent to the Vehicle Control Unit (VCU), so it can take the appropriate actions for steering the race car. For this, the UDP protocol is used. Whenever a camera image is processed by the VPU, the useful information for the VCU (car position: x and y, orientation: ψ, speed: v_x and v_y and rotational speed: ω) is wrapped in a UDP packet and sent through a local loopback to the VCU (using the IP address 127.0.0.0), which is just another application running on the same computer as the VPU.
The `<arpa/inet.h>` and the `<netinet/in.h>` libraries from the Free Software Foundation provide an easy and well-documented interface for the sending and receiving of UDP and TCP packets. With these libraries, the communication was quickly up and running. The low-level code which sends and receives the actual packets was neatly wrapped in two self written classes `UDPclient` and `UDPserver`. The timing of the UDP packets is important: if a packet has a large delay, the information enclosed in it is not up to date, which is detrimental for real-time applications. The timing was tested at the receiving end (the VCU), the results are shown in Fig. 4. Notice that except for the first packet, all packets arrive approximately after $0.01\ s$, which is the sampling speed of the VPU.

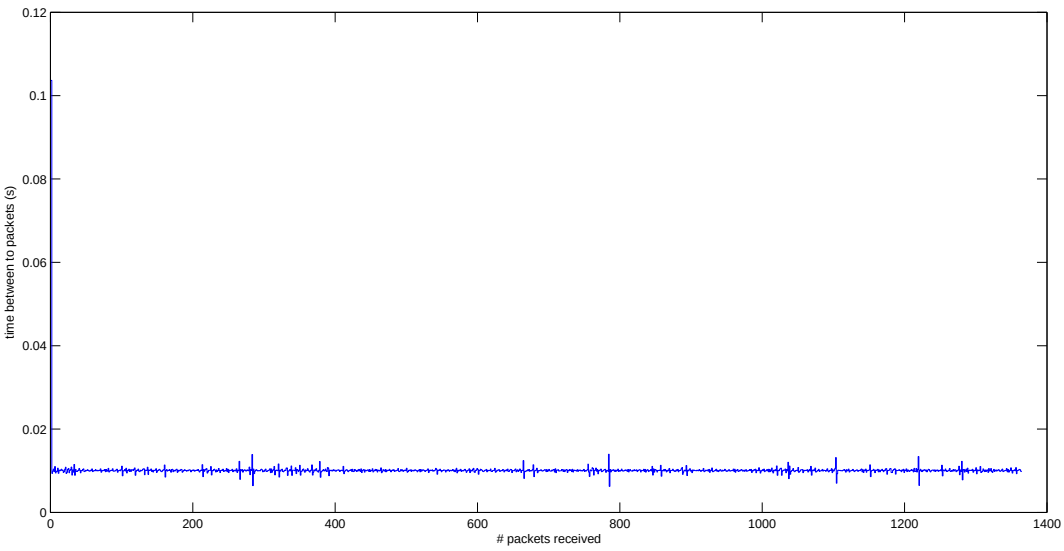


Figure 4: *Time between two UDP packets received by the Vehicle Control Unit.*

2.5.4 The communication between VCU and RC controller

Lastly, the (modified) RC controller had to be addressed from within the VCU. The setup is as follows: the VCU drives the National Instruments Data Acquisition Card (DAQ), and the DAQ overrides the manual controls on the Kyosho RC controller. For this, the free and open source library for device drivers `comedi` was used. At first, the wiring plan of the DAQ had to be examined, and the output voltages had to be measured. The `comedi` library provides routines to switch between voltages and raw data processed by the computer (which takes all integers between 0 and 65535). With a multimeter, the output voltages were assessed to be correct.
The RC controller was modified (this work was finished before the start of the internship) to make it possible to override the signals from the manual controls (the throttle and the steering wheel). Two mini-jack connectors are connected to the central integrated circuit board of the controller: one for

throttle, one for steering. The steering angle could be set directly with the appropriate voltage on the controller. The torque on the rear axle of the car however, was not directly addressable. Instead, the input to the DC motor was the duty cycle of the PWM-modulated motor voltage signal. The DC motor in the race car operates with Pulse Width Modulation (PWM). The duty cycle is the proportion of the pulse width to the sample time. The duty cycle lies thus between 0 (no torque) and 1 (maximum torque) in forward operation.
The mini jack connectors are in turn driven by the DAQ, which gets the signals from the VCU running on the computer. All this made it possible to drive the race car from within a computer program.

2.5.5 The Vehicle Control Unit

The Vision Processing Unit was based on external code, but the Vehicle Control Unit itself had to be written from scratch. The main routine uses the following classes:

- trackParser: for reading the trackfile which contains the track geometry and a reference trajectory on this track. In the following, the used reference trajectory will be driving the centerline at a constant velocity.
- UDPserver: for receiving the UDP packets sent by the VPU.
- RCdriver: for steering the Data Acquisition Card and thus the Kyosho RC controller.
- Pcontroller: which computes the appropriate control actions (steering angle δ and duty cycle D) for a proportional controller. The proportional control laws are as follows:

$$\begin{bmatrix} e_{longitudinal} \\ e_{lateral} \\ e_{orientation} \end{bmatrix} = \begin{bmatrix} \cos(\psi) & \sin(\psi) & 0 \\ -sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{ref} - x \\ y_{ref} - y \\ \psi_{ref} - \psi \end{bmatrix}, \tag{8}$$

$$\delta = K_{lateral} e_{lateral} + K_{orientation} e_{orientation} \tag{9}$$

$$D = K_{longitudinal} e_{longitduinal}. \tag{10}$$

  The obtained steering angle δ and duty cycle D are then translated into an appropriate voltage input via the DAQ.

2.5.6 The integration and Milestone 1

With all the seperate components working and tested, the different parts were integrated and evaluated on a very simple race track, shown in Fig. 2a. In the first milestone, the reference trajectory that the car tries to follow had a constant speed of $0.5\mathrm{m/s}$. By simulation, the gains for the proportional controller were tuned. The results can be seen in Figure 5. The performance of the car on the oval track can be ameliorated still, but this is the proof-of-concept demonstration of the project.

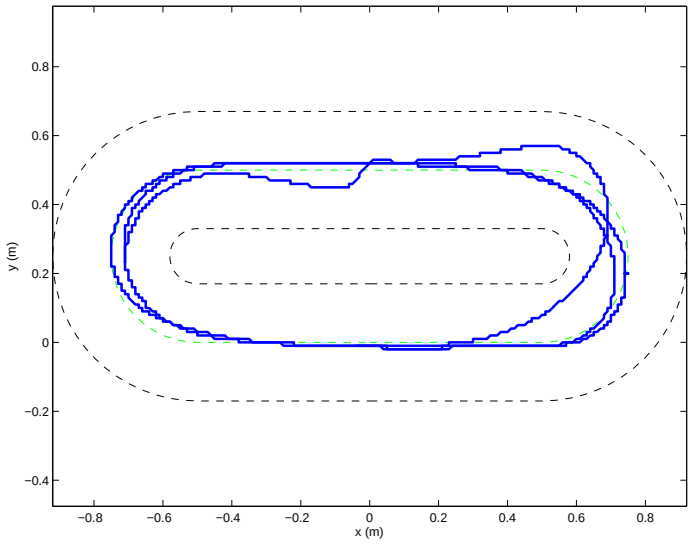


Figure 5: *Performance of the proportional feedback controller on the oval track at a reference speed of* $0.5\mathrm{m/s}$. *The performance is reasonable for such a simple controller, but note the large deviation in the upper right corner of the track.*

2.6 The project: towards Milestone 2

With the setup working at the end of week 2, in the next two weeks a more advanced controller is prepared by creating a user-friendly workflow for designing the controller and by identifying a prediction model for the vehicle.

2.6.1 Integration with Simulink

In order to be able to test performance of a certain controller in advance, simulations in the MATLAB/Simulink framework were developed. However, we need to make sure that the simulated controllers can be used directly in our car control application, for maximum utility of the simulation. To this end, we use Simulink coder. This tool makes it possible to export Simulink blocks to C++ code. The existing simulations had to be rewritten so that they use the subset of Simulink blocks supported for code generation. Also, major alterations had to be made to the existing Pcontroller class inside the VCU. The new code is based on a MATLAB example code for embedded real time applications. A tutorial on the Simulink Coder tool can be found in [7].
The integrated workflow consists of the following steps: the simulation is run and next the performance of the controller is assessed. Then, the entire controller block in Simulink can be exported to C++ code and lastly integrated in the car control application running on the Debian computer.
In the end, the code generated with Simulink was tested. The result was that the car behaved exactly the same as with the old, non-generated code, which was expected.

2.6.2 Measuring the control actions

Until now, the two control actions (steering angle δ and duty cycle D) were sent to the National Instruments Data Acquisition Card by a raw data format: an integer number between 0 and 65535.

However, we would like to know the exact relation between voltages and actual control actions. Therefore, two experiments have been carried out.

1. The steering angle: there is a very simple relation between steering angle and radius of curvature of the curve driven by a car. It is given by

$$\text{radius} = \frac{\text{track}}{2} + \frac{\text{wheelbase}}{\sin(\delta)}, \tag{11}$$

where track and wheelbase of a car are defined as in Fig. 6. By By sending a fixed voltage to the steering system, the car will drive circles. By measuring the radius of these circles, a map from steering angle to voltage can be made using a linear least squares fit. The results are shown in Fig. 7.

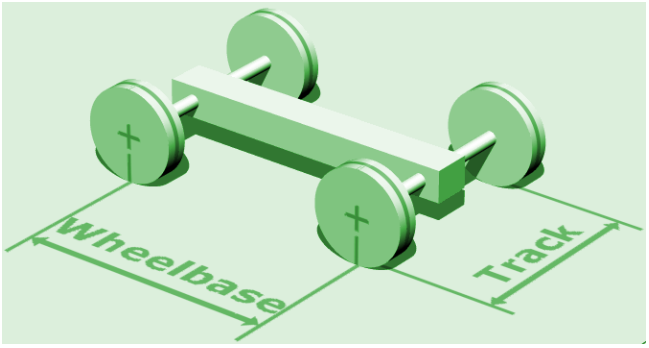


Figure 6: *Definition of track and wheelbase of a car.*

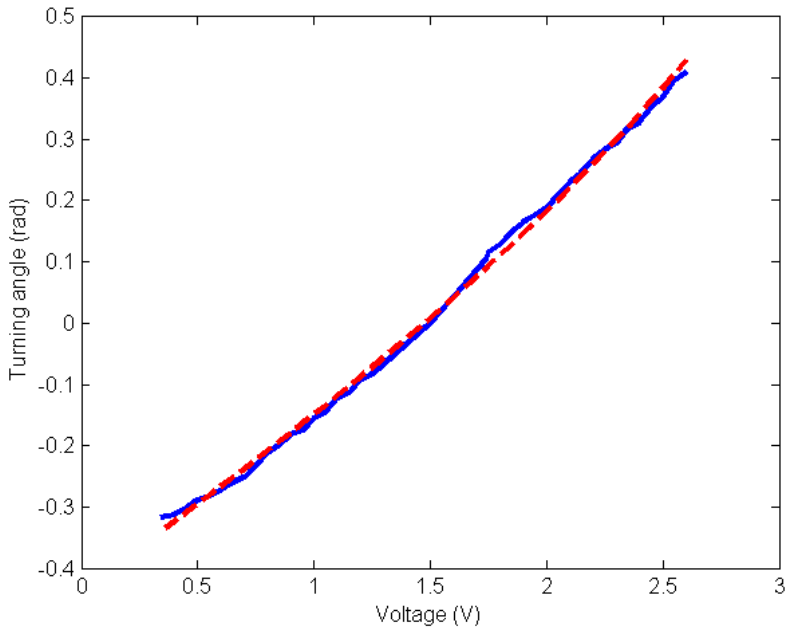


Figure 7: *Relation between the voltage applied to the RC controller and the steering angle of the car. The measurements are shown in blue, the least-squares curve is shown in red.*

2. The duty cycle. To measure the duty cycle-to-voltage mapping, we used a digital oscilloscope. By turning up the voltage applied to the handheld controller, the duty cycle becomes larger and

larger. The duty cycle can be read off the oscilloscope by fixing the screen so that it contains one period of the motor sample time. The proportion of the time that the voltage is high (for the used DC motor, a high voltage corresponds to $2\,V$) is the duty cycle The measured values can be found in Table 2. Note that a low voltage corresponds to a high duty cycle. We also found that the duty cycle changes in a discrete way, the changing points are shown in the table. In Fig. 8, the measurements are plotted along with a linear fit. The duty cycle-to-voltage mapping is obtained by inverting the red curve in the figure.

Voltage (V)	pulse width (μs)	dutycyle (-)
1.39	14	0.07
1.35	31	0.14
1.31	47	0.23
1.28	63	0.30
1.24	79	0.38
1.20	95	0.46
1.17	111	0.53
1.12	127	0.61
1.09	143	0.69
1.05	163	0.78
1.01	176	0.85
0.98	193	0.93
0.93	208	1.00

Table 2: *Measurements of the voltage applied to the RC controller and the corresponding duty cycle applied to the DC motor.*

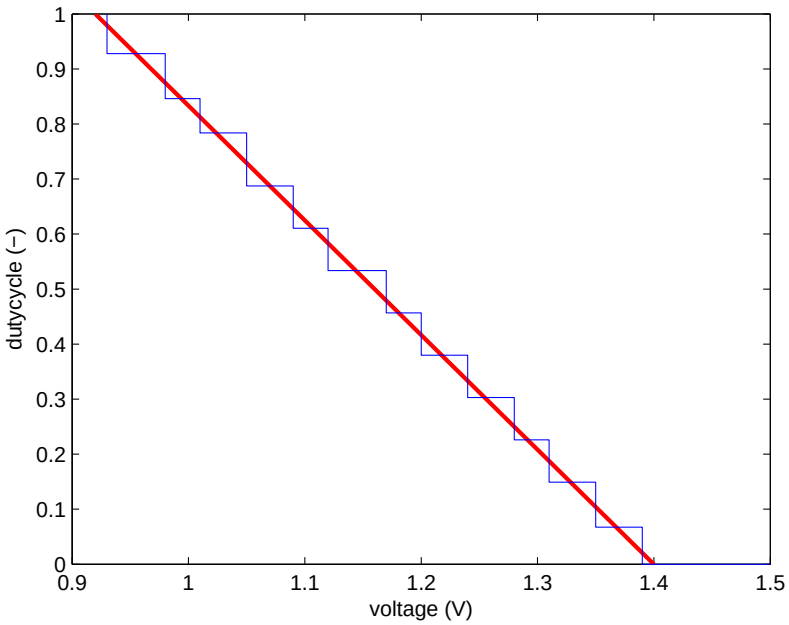


Figure 8: *Mapping of the voltage to the duty cycle. In red, a linear fit is shown.*

2.6.3 Identifying a car model

An important aspect of model based control is the identification of the model. With an appropriate and well-identified model, automatic control can be carried out much more effectively. The car model used in our setup is a non-linear bicycle model (cf. Fig. 9) as described in [8]:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\psi} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} v\cos(\psi + C_1\delta) \\ v\sin(\psi + C_1\delta) \\ v\delta C_2 \\ C_{m1}D - C_{m2}Dv - C_{r2}v^2 - C_{r0} - (v\delta)^2 C_2 C_1 \end{bmatrix},$$

with x, y, ψ, v, D, δ as before and with the following constants to be identified:

parameter	pyhsical meaning	parameter value
$C_1(-)$	*geometrical* ($\frac{l_r}{l}$)	0.5
$C_2(m^{-1})$	*geometrical* ($\frac{1}{l}$)	17.06
$C_{m1}(m/s^2)$	*motor parameter*	12.0
$C_{m2}(1/s)$	*motor parameter*	2.17
$C_{r2}(1/m)$	*second order friction parameter*	0.1
$C_{r0}(m/s^2)$	*zero order friction parameter*	0.6

Table 3: *Parameters used in the non-linear bicycle model, along with their meaning and their identified values.*

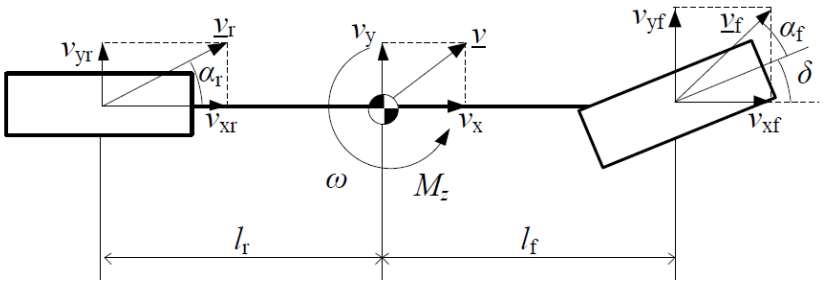
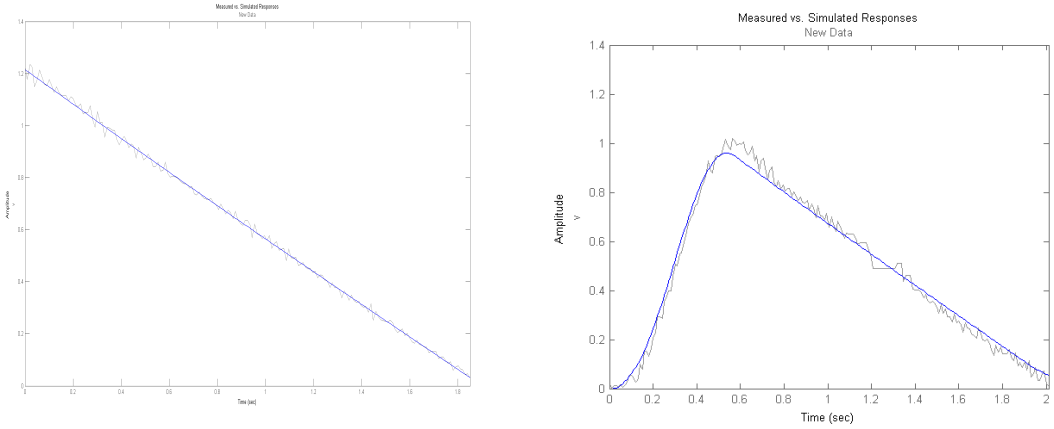


Figure 9: *The bicycle model used in the setup. Note that we simplify to a slip-free model, so $\alpha = 0$ rad.*

The model identification was done as follows. The geometrical parameters did not need to be identified, they were measured. The other parameters were identified using the longitudinal model (this is the model with a fixed steering angle δ of zero). The friction parameters C_{r2} and C_{r0} were estimated via a deceleration experiment. This amounts to pushing the car and let it decelerate until it stops. The motor parameters C_{m1} and C_{m2} by carrying out an acceleration experiment. For this, we applied an initial pulse in the duty cycle. The velocity of the car was then recorded and the identification was done with the Simulink Parameter Estimation Toolbox. The results of these experiments can be seen in Fig. 10.
When tested on a track instead of a straight line, we found that the model with these identified parameters performed badly. Especially the velocity was underestimated in the model. The identification was completed as follows: we took the parameters found from the deceleration and acceleration experiments as starting values for some experiments with more challenging tracks. We applied some inputs D and δ to our radio-controlled car and recorded the trajectory it followed. Next, the same



(a) *Deceleration experiment* **(b)** *Acceleration experiment*

Figure 10: *Parameter identification experiments*

inputs were applied to our model, that was run in MATLAB/Simulink in open loop. This means our model was driving as if it were blind. Lastly, the model parameters were tuned and the model was simulated iteratively until the model trajectory fitted the real trajectory well enough.
The first combination of inputs were obtained by following a curved trajectory, as can be seen in the leftmost half of Fig. 11. Also a step response was tested, as can be seen in the right half of the picture. As can be seen, this model is adequate enough to be used in an MPC algorithm: the speed is followed closely, as well as the yaw rate ω. The position in the 'curves' experiment is not that well followed by the model, but for rather short prediction horizons (cf. MPC), this will do. Following the speed and rotational speed is more important, because the reference trajectory is at a constant speed. Otherwise, the car will lag more and more behind over time. The final parameter set that was chosen can be found in Table 3.

2.6.4 PID control on the final track

A last accomplishment that was made for Milestone 2 was the testing of the PID controller on the final track. This provoked some minor difficulties, a re-tuning of the proportional gains had to be made for example. Also a re-calibration was needed. The extrinsic calibration is unavoidable on a new race track.
The tuning of the three gains ($K_{lateral}, K_{longitudinal}$ and $K_{orientation}$) of the proportional controller is done first in simulation, and then tested on the real setup. A good set of gains seems to be $K_{lateral} = 2, K_{longitudinal} = 4, K_{orientation} = 0.5$. This is what could be expected: diminishing the lateral error is the most important, whereas if the car has the right orientation is not crucial for tracking the centerline; it will correct its orientation by trying to track the centerline automatically.
At a speed of $1.1\mathrm{m/s}$, this gives a quite satisfactory performance for the real car (Fig. 12a). However, if we turn up this reference velocity, the performance deteriorates. For $v_{\mathrm{ref}} = 1.35\mathrm{m/s}$, this is shown in Fig. 12a. Note the bottom left corner, where the car almost hits the border of the track. A possible solution to this deviation from the centerline would be to increase the gain on the lateral error. However, doing this, creates oscillations in the car behavior (Fig. 12b). A last illustration of the behavior of the proportional controller is where the gain on the lateral error is too low: this is also shown in Fig. 12b. Here, the car drives into the wall after the first corner; the proportional controller has no clue that the wall is there.

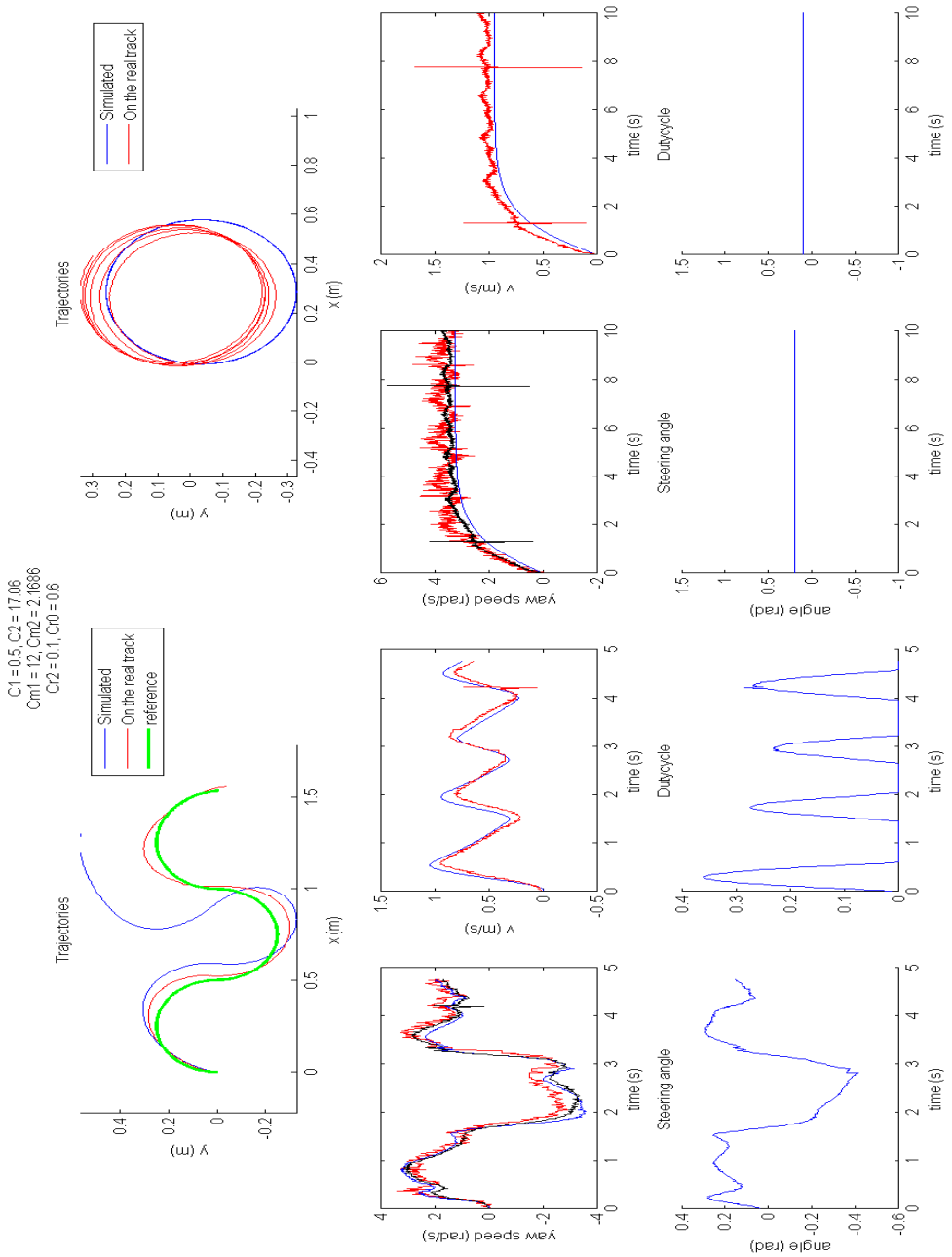
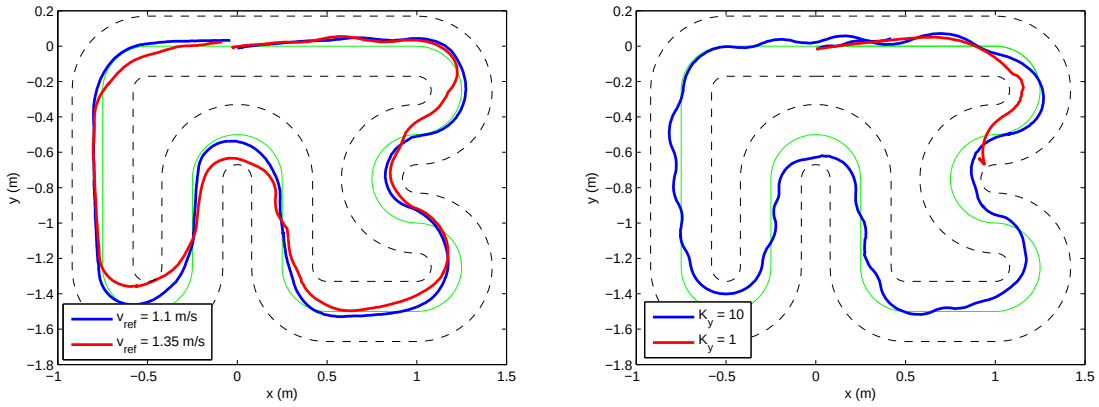


Figure 11: *Simulations of the bicycle model in comparison with the real car. On the bottom half the 'curves' experiment is plotted, the top half of the figure the step response can be seen.*



(a) *The performance of the car for tracking at two different reference velocities for the proportional controller.*

(b) *Car trajectory for different gains for the lateral error ($K_{lateral} = 10, K_{lateral} = 1$). Note the oscillations in the path for the higher gain, which is unnatural behavior. Setting the gain lower drives the car to the wall.*

Figure 12: *The performance of the proportional feedback controller on the final track.*

2.7 The project: towards Milestone 3

The last two weeks of the internship were used for the design of an MPC controller for trajectory tracking.

2.7.1 Problem formulation

The goal of this part of the project is the tracking of a predefined trajectory. In a first stage, this will be the tracking of the centerline. Afterwards, tracking of an online computed trajectory will be attempted. The strategy for the design of the MPC tracking is to reformulate the problem as a regulation problem. This means that we will try to drive the error state to zero instead of tracking a time-varying trajectory. A detailed explanation of the reformulation to error states can be found in appendix B. The used symbols and their meanings can be found in Table 4.

n_x	normalized longitudinal error
n_y	normalized lateral error
n_ψ	normalized error on the orientation
n_v	normalized error on the speed
u_1	simplified steering input
u_2	simplified duty cycle input
T	sampling time

Table 4: *Symbols used in the MPC regulation formulation.*

Note that we simplify our inputs to the system:

$$u_1 = v\delta C_2 - v_n\kappa \tag{12}$$

$$u_2 = \frac{a - a_n}{v_n} \tag{13}$$

The normalized error dynamics are then

$$\begin{bmatrix} n_x(k+1) \\ n_y(k+1) \\ n_\psi(k+1) \\ n_v(k+1) \end{bmatrix} = \begin{bmatrix} 1-T\alpha_n & T\omega_n & 0 & T \\ -T\omega_n & 1-T\alpha_n & T & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1-T\alpha_n \end{bmatrix} \begin{bmatrix} n_x(k) \\ n_y(k) \\ n_\psi(k) \\ n_v(k) \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ T & 0 \\ 0 & T \end{bmatrix} \begin{bmatrix} v\delta C_2 - v_n\kappa \\ \frac{a-a_n}{v_n} \end{bmatrix}, \tag{14}$$

with

$$a = C_{m1}D - C_{m2}Dv - C_{r2}v^2 - C_{r0} - (v\delta)^2 C_2 C_1 \tag{15}$$
$$v = v_n(n_v + 1) \tag{16}$$
$$\alpha_n = \frac{a_n}{v_n} \tag{17}$$
$$\omega_n = v_n\kappa \tag{18}$$

Lastly, we can formulate the MPC regulation problem. In words, this amounts to minimizing the state error and the input error, weighted by matrices Q and R, respectively, subject to the error dynamics (14) and state and input constraints. These constraints will be described below.

MPC formulation

$$\min_{x\in\mathbb{R}^{n_x\times N}, u\in\mathbb{R}^{n_u\times N}} \quad \sum_{k=1}^{N} x_k^T Q_k x_k + \sum_{k=0}^{N-1} u_k^T R_k u_k \tag{19a}$$
$$\text{subject to} \quad x_{k+1} = A_k x_k + B_k u_k, \tag{19b}$$
$$E_{x,k}x_k \leq F_{x,k} \tag{19c}$$
$$E_{u,k}u_k \leq F_{u,k} \tag{19d}$$

with

$$x = \begin{pmatrix} n_x \\ n_y \\ n_\psi \\ n_v \end{pmatrix}, u = \begin{pmatrix} v\delta C_2 - v_n\kappa \\ \frac{a-a_n}{v_n} \end{pmatrix} \tag{20}$$

Input constraints On the actual car, the inputs are constrained to certain values because the car can reach its physical limits. We chose the following constraints, based on measurements:

$$-0.4\text{rad} \leq \delta \leq 0.4\text{rad} \tag{21}$$
$$-10m/s^2 \leq a \leq 10m/s^2 \tag{22}$$

State constraints The most obvious state constraint is the fact that the car cannot drive into the borders surrounding the track. This amounts to (see Fig. 13, left)

$$n_{y,min} = \frac{\Delta w - w/2}{v_n} \tag{23}$$
$$n_{y,max} = \frac{\Delta w + w/2}{v_n}. \tag{24}$$

Another state constraint is that the car, when close to a wall, must still be able to turn away from the wall (Fig. 13, right), or

$$\frac{d}{R} \geq 1 - \cos(\gamma) \tag{25}$$

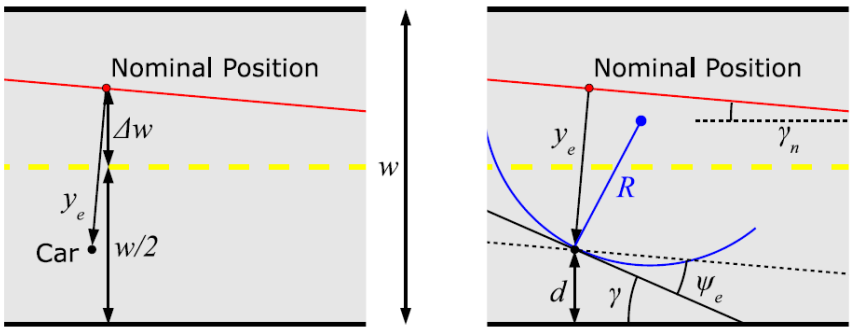


Figure 13: *State constraints on the car needed for the MPC formulation.*

2.7.2 Implementation

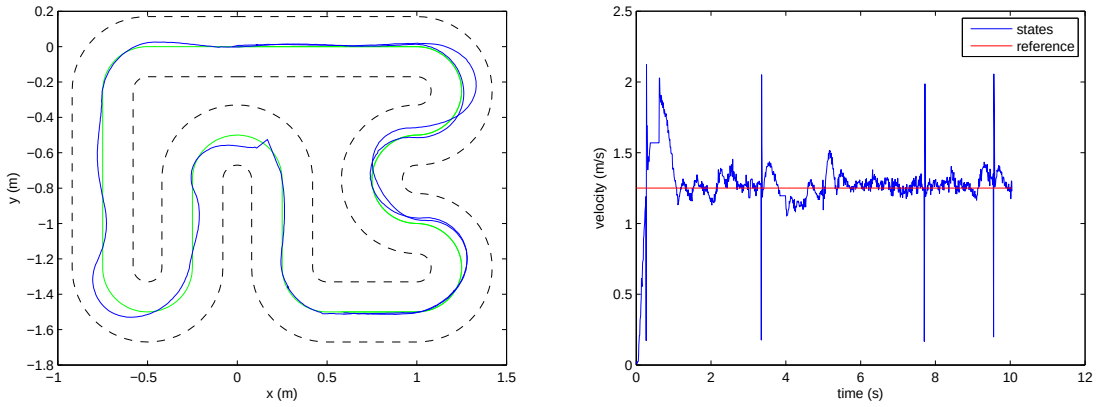
The presented problem formulation matches that of a linear MPC problem: it contains a quadratic cost function with linear state and input constraints. This problem can be formulated as a generic QP (Quadratic Program). This was done as described in [9]. To solve the resulting QP, we used the qpOASES software written by Joachim Ferrea et al. ([10]). This software provides a Simulink interface, as well as tailored C++ code, so it fits nicely into our project. The workflow remains the same: simulate a controller, then export the code and build it on the Debian computer.

2.7.3 Trajectory Tracking: results

The only thing that needs to be done before trying the car on the setup is determining the cost matrices Q and R: with badly chosen weighting matrices, the car will crash into the wall immediately or not drive at all. An initial good guess matrices is

$$Q = \text{diag}([1e3, 1e3, 1e-7, 5]), R = \text{diag}([1e-2, 1e-2]).$$

The results of a run with these values can be seen in Fig. 14. In the following experiments, we chose a reference velocity of $v_{ref} = 1.25m/s$.



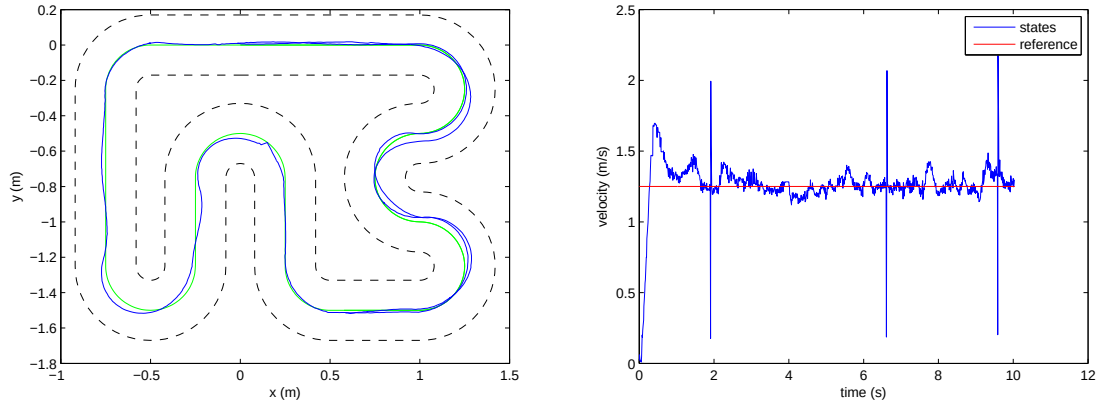
(a) *Trajectory* **(b)** *Velocity profile*

Figure 14: *Experiment with initial values for Q and R.*

After some finetuning, we came to the following, better weighting matrices:

$$Q = \text{diag}([1e3, 1.2e3, 1e-7, 5]), R = \text{diag}([2e-2, 1e-2]),$$

with results depicted in Fig. 15. When we compare, we can say that the centerline is followed more closely, and also in a smoother way. This follows from the higher weighting on the lateral error ($Q_{2,2}$) and the higher weighting on the first input. Look for example at the bottom left corner of the trajectory, where the second experiment has a smaller deviation from the centerline. In this way, the car does not have to correct afterwards for this deviation. Also the first corner (top right) is taken in a much better way.



(a) *Trajectory* **(b)** *Velocity profile*

Figure 15: *Experiment with better values for Q and R.*

2.7.4 Trajectory planning

At the end of the internship, a more involved 2-step algorithm was tried out: At the higher level, an online trajectory optimization is done. At a lower level, this calculated trajectory is tracked using the MPC approach described above. However, this approach caused some problems. The calculations spawned a lot of NaNs, possibly because of a bug somewhere in the code.

3 Conclusion

As a general conclusion, we might say that the implementation of a practical setup always takes more time than the understanding and the simulation of theoretical results behind it. However, at the end, seeing a practical result gives a deeper satisfaction than just assessing some simulation results. Also, the goal of the internship is reached: a demonstrator setup was created which was on display at the European Vehicle Conference of the company. It proved to be a succesful eye-catcher with a lot of interested visitors.
The more difficult part of the internship, on my behalf, was the integration with Simulink. I was surprised by how badly the Simulink coder is documented by the vendor. This caused some difficulties, and quite some time spent in debugging.
One of the nicest parts of the project was, that in the end, with the trajectory tracking using MPC, you can really see an impact of certain parameters in the problem. For example, putting a higher penalty

on the y-error (lateral error), you could really see that the car sticked closer to the center. In a PID context, this is generally more difficult.
Lastly, the supervision of the internship was very satisfactory. Whenever problems were encountered, or questions came up, help was at hand. Moreover, the working atmosphere turned out to be pleasant.
However, the setup can be ameliorated in different ways. First of all, the trajectory planning can be made to work. Another possibility for a further project is switching from one camera to different cameras, making it possible to have an arbitrarily large track. Of course, more difficult vehicle models and/or switching to non-linear MPC can be investigated also.

4 Link with university programme

For completing the internship, knowledge from following courses (and others) from the mathematical engineering programme (Master in de Ingenieurswetenschappen: Wiskundige Ingenieurstechnieken) were used:

- Computergestuurde Regeltechniek (H03E8A): The last part of this course deals with linear MPC and the tuning process. Also, experience with Simulink, acquired in this course, came in handy during the project.
- System Identification and Modeling (H03E1B): Although only linear systems were covered by this course, it proved to be of great importance in terms of ideas about how to tackle the identification problem.
- Optimization (H03E3A): Especially the switch from MPC problem formulation to QP problem, and the corresponding terminology, were drawn from this course.
- Technisch-Wetenschappelijke Software (H03F0A): Learning how to compile and link a C++ application was crucial for this project, as well as miscellaneous C++ skills and tricks. Also, experience with makefiles proved to be of great convenience.

5 Logbook

Date	Activities
22/07	introduction with Stijn. Installation of software. Reading ETH thesis (Rutschmann) and document on Kalman filter
23/07	continue reading ETH theses (Keiser). Reading of existing code. Test intrinsic calibration
24/07	measuring intrinsic calibration, problem fixed
25/07	looking for blind spots in the vision processing, looking at Kalman filter problem
26/07	Kalman filter problem solved, testing of UDP connection
29/07	testing of National Instruments Data Acquisition Card
30/07	writing of input trackfile parsing. Testing in a straight line
31/07	tuning of PID controller. **Milestone 1 reached**.
01/08	Refactoring of the code. Export code to SVN
02/08	Looking at delay problem at beginning of car run
05/08	Solved delay problem, measuring steering angle
06/08	Fitting of steering angle to voltage. Reading about Simulink code generation
07/08	Simulink model creation. Simple example of C++ code generation
08/08	tuning of Kalman filter. Generation of curves track. SVN troubles
09/08	open loop testing of car model on the curves track
12/08	oscilloscope measurements for duty cycle
13/08	finishing of duty cycle measurements. Start of model identification
14/08	model identification: step response. **Milestone 2 reached**. Building of final track
16/08	tested vision processing on final track
19/08	fixing of parameters. Reading of ETH thesis (Wunderli)
20/08	formulation of MPC problem
21/08	tested qpOASES on real time computer
22/08	MPC problem implementation
23/08	MPC tracking in simulation
24/08	MPC tracking succesful, problem was camera connection. **Milestone 3 reached**.
27/08	started MPC planning algorithm
28/08	MPC planning works in simulation
29/08	MPC planning on track: not satisfactory
30/08	there remain problems in the MPC planning. Wrap up

6 Appendix A: Technical details of the setup

Infrared camera: PointGrey Flea 3 Infrared Camera
Imaging: 1280x1024 at 100 FPS
Connection with computer: USB 3.0

Computer: Real-time Debian system
Intel Core i3-3220 @ 3.3 GHz
OS: Debian 7.0 'Wheezy'
Kernel: Linux 3.8.13 with Preempt RT Patch

DAQ card: National Instruments PCI-6229
4 analog outputs, 32 analog inputs, 48 digital I/O
Driver software: ni_pcimio from Comedi library

Race Car: Kyosho dNaNo FX-101 ASF2.4GHz System

RC-controller: PERFEX KT-18 Transmitter 2.4GHz

7 Appendix B: from states to error states

The non-linear bicycle model is as follows:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\psi} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} v\cos(\psi + C_1\delta) \\ v\sin(\psi + C_1\delta) \\ v\delta C_2 \\ C_{m1}D - C_{m2}Dv - C_{r2}v^2 - C_{r0} - (v\delta)^2 C_2 C_1 \end{bmatrix}. \tag{26}$$

We can now switch to error states like this:

$$\begin{bmatrix} x_e \\ y_e \\ \psi_e \\ v_e \end{bmatrix} = \begin{bmatrix} \cos(\psi_n) & \sin(\psi_n) & 0 & 0 \\ -\sin(\psi_n) & \cos(\psi_n) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x - x_n \\ y - y_n \\ \psi - \psi_n \\ v - v_n \end{bmatrix} \tag{27}$$

where $[x_n, y_n, \psi_n, v_n]$ denotes the nominal trajectory, which is the trajectory that is being tracked, for now the centerline at a constant speed v_n. Next, we would like to determine the error dynamics, which is the result of the combination of the derivative of the error state equation (27) with the vehicle model (26):

$$\begin{bmatrix} \dot{x_e} \\ \dot{y_e} \\ \dot{\psi_e} \\ \dot{v_e} \end{bmatrix} = \begin{bmatrix} v\cos(\psi_e + C_1\delta) - v_n(1 - \kappa y_e) \\ v\sin(\psi_e + C_1\delta) - v_n(\kappa x_e) \\ v\delta C_2 - v_n\kappa \\ a - a_n \end{bmatrix} \tag{28}$$

with

$$\kappa = \frac{\omega_n}{v_n} \tag{29}$$

References

[1] Wang and Boyd, *Fast Model Predictive Control Using Online Optimization,* IEEE Transactions on Control Systems Technology, vol. 18, no. 2, pp. 267–278, Mar. 2010.

[2] LMS Facts and Figures, `http://www.lmsintl.com/fact-figures`, last accessed August 2nd, 2013

[3] Siemens neemt LMS International over, *De Tijd, November 8th, 2012*, `http://www.tijd.be/r/t/1/id/9265205`, last accessed August 2nd, 2013

[4] Marc Pollefeys, *Visual 3D modeling from Images,* University of North Carolina, pp. 21–27, 2001

[5] *Camera calibration With OpenCV*, `http://docs.opencv.org/doc/tutorials/calib3d/camera_calibratio`

[6] Hugh Durrant-Whyte, *Introduction to Estimation and the Kalman filter*, University of Sydney, 2001

[7] José Carlos Molina Fraticelli, *Simulink Code Generation*, april 2012, accessed from `http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20120014980_2012014792.pdf`

[8] P. Spengler, C. Gammeter, *Modeling of 1:43 scale race cars*, ETH Zürich, june 2010

[9] L. Wunderli, *MPC Trajectory Tracking*, ETH Zürich, april 2011

[10] `qpOASES` Homepage, `http://www.qpOASES.org`, 2007-2013