Technical University of Denmark

Thomas Passer Jensen

Pre-competition preparations for an autonomous Shell Eco-marathon car

Simulations, steering and braking

Synthesis project, August 2018

DTU Electrical Engineering
Department of Electrical Engineering

Pre-competition preparations for an autonomous Shell Eco-marathon car, Simulations, steering and braking

Author:
Thomas Passer Jensen (s134234@student.dtu.dk)

Advisors:
Jens Christian Andersen (jca@elektro.dtu.dk)
Henning Si Høj (hsih@elektro.dtu.dk)

DTU Electrical Engineering
Automation and Control
Elektrovej
Building 326
2800 Kgs. Lyngby
Denmark
Tel: 45 25 35 76

studieadministrationen@elektro.dtu.dk

Project period:	December 1st, 2017 – August 19th 2018
ECTS:	10
Education:	MSc
Field:	Electrical Engineering
Remarks:	This report is submitted as partial fulfilment of the requirements for graduation in the above education at the Technical University of Denmark.
Copyrights:	

Abstract

Before entering in the Autonomous UrbanConcept category of Shell Eco-marathon, the Ecocar Dynamo 14.0 from Technical University of Denmark still needed much work. In this report some of the work is outlined, including setup of a simulation environment, further development of the autonomous system integration, work on ensuring precise and safe autonomous steering and braking, and improvements of the algorithm for estimation of vehicle velocity. The whole Ecocar with all the autonomous sensors were setup in a simulation, to allow for more rapid development of the navigation software. For the starting and stopping of the autonomous Ecocar, software was developed to automatically launch the autonomous software at the push of a button on the steering wheel. The steering system was improved to be able to accurately turn the front wheel angles to the desired angle at a fast wheel turning speed, enabling the Ecocar to drive the track autonomously at a higher speed. In the end, the improvements helped in the way of winning the Autonomous UrbanConcept category in Queen Elizabeth Olympic Park, London 2018.

Preface

This report is the result of a 10 ECTS point synthesis course conducted at the Technical University of Denmark. The project is carried out as a member of DTU Roadrunners.
The report is divided into chapters, corresponding to work on different parts of the car, as described in the introduction. Each chapter begins with a brief overview and usage instructions. It is recommended to begin by reading these overviews, if reading the report as a new member on the team. If further work is to be done on a certain part described in this report, that whole chapter should be read for the full documentation.

A special thanks to the whole autonomous team for a joyful time together. Also thanks to the rest of DTU Roadrunners for a great team experience throughout the semester.

Thomas Passer Jensen

Contents

Chapter 1

Introduction

The field of autonomous vehicles is in rapid growth with many large companies such as Waymo (Google), Uber, General Motors, Daimler and Tesla competing to reach full autonomy first. With a decreasing prices of technology, the barrier of entry to get started with self-driving vehicles is continuously lowering, enabling more people to obtain insight in the field.

1.1 Shell Eco-marathon

The Shell Eco-marathon is a competition for fuel efficiency held annually in Asia, America and Europe [1]. There are two types of vehicles in the competition; Prototype and UrbanConcept, and three types of energy categories for each: battery-electric, hydrogen fuel cell and internal combustion engine.
A new category was added in the 2018 competition in Europe; Autonomous UrbanConcept. In this competition the UrbanConcept vehicle must autonomously navigate through certain challenges.

Figure 1.1: Part of the track at Shell Eco-marathon 2018 at the Queen Elizabeth Olympic Park in London. [Ed Robinson/Shell]

1.2 DTU Roadrunners

DTU Roadrunners is a student driven project about building a fuel efficient car in the UrbanConcept type to compete in Shell Eco-marathon. The car has an internal combustion engine which runs on ethanol [2]. The goal is to run as far as possible on a small amount of fuel. The team consists of engineering students from the mechanical and electrical engineering departments. The team has won the UrbanConcept category 11 times and the Autonomous UrbanConcept category once.

Figure 1.2: The DTU Roadrunners team at Shell Eco-marathon in London 2018 when receiving the first prize in the category UrbanConcept Internal combustion engine.

1.3 Purpose

Talk of an autonomous competition began in 2016 which set off the work on the augmentation of the Ecocar Dynamo with sensors and actuators for autonomous driving. The integration began in 2017 [3], and software for processing of the raw LiDAR data for ground- and obstacle detection was done in early 2018 [4]. The autonomous Ecocar had to be ready for competition in July 2018. This synthesis project describes a collection of work done in preparation for the competition.

1.4 Report overview

The work described in this report began with simulations. The original problem formulation for this synthesis project was as follows:

> *The DTU Ecocar is to be augmented with autonomous capabilities by equipping sensors and actuators. The aim of this project is to construct a simulation environment where every part of the autonomous Ecocar is described, which will aid in the development of navigation algorithms for the car, allowing more rapid development before the upcoming Shell Eco-marathon 2018 competition for self-driving vehicles.*

As the competition neared more time was spent on other parts of the system, where improvements were needed. In the end, the work done was within some quite different aspects of the autonomous Ecocar, and as such the report is divided into five different chapters:

- **Chapter 2 - Simulation**: Setup of a simulation environment for the autonomous car.
- **Chapter 3 - Launching of the autonomous system**: Design of software for starting and stopping the autonomous mode.
- **Chapter 4 - Steering**: Improvements to the steering node in ROS which controls the steering actuator.
- **Chapter 5 - Braking**: Improvements to the braking node in ROS which control the brake actuator.
- **Chapter 6 - Velocity estimation**: Improvement of the velocity estimation for the Ecocar.

Chapter 2

Simulation

Robot simulation is an important tool for development in many robotics applications. Because use of simulations can reduce testing time tremendously, it was decided to set up a full simulation environment of the autonomous Ecocar with all the sensor data available. For this task, the open-source robotics simulation environment Gazebo (gazebosim.org) was chosen, as it has the most community support and is relatively easy to get started with. It is also integrated with ROS (ros.org), the robotics framework used for the Ecocar.

Figure 2.1: Gazebo logo.

The ROS packages described in this chapter of the report are the `ecocar_gazebo` and `ecocar_description` packages. The packages are developed for ROS Kinetic Kame and Gazebo 7. The simulation environment is developed in C++, except for the track generation which is written in Python.

2.1 Getting started

The simulation environment packages is residing in its own catkin workspace in the `catkin_ws_sim` directory residing in the autonomous-ecocar Bitbucket repository. Installation instructions can be found in the README file.
A list of available launch files with descriptions is shown in Table 2.1. These are started with the following command:

```
roslaunch ecocar_gazebo <launch file>
```

Launch file	Description
`model.launch`	Empty world with only the Ecocar.
`autonomous.launch`	Starts the autonomous navigation stack.
`challenge1.launch`	Randomly generated tracks. See command line options below.
`challenge1_london.launch`	Main track from SEM London 2018 (without elevation data)
`challenge2.launch`	Straight track with three box type obstacles.

Table 2.1: Available launch files.

Command line options exist for the challenge 1 launch file. These are passed in the following way, with a space between the parameter and the value:

```
roslaunch ecocar_gazebo challenge1.launch track:="-parameter value"
```

Parameter	Description	Default value
-size	Dimension of box to generate track in	100
-diff	Difficulty of track, larger is harder	1
-points	Number of points to use for initial track generation	10
-o	Filename for the generated world file	track_[datetime].world
-maxdisp	Max displacement of points on track added to generate more turns, larger number increases difficulty	size/20
-mindist	Minimum distance between points used for track generation. Smaller values increase difficulty	size/10
-roadwidth	Width of the road	6.5
-filename	World file to load instead of generating a new track	N/A

Table 2.2: Command line parameters for the challenge 1 launch file.

The available parameters are shown in Table 2.2. Simulation parameters can be changed without needing to recompile the simulation packages. The parameters for the autonomous Ecocar can be set in the `params.xacro` and `ecocar.gazebo` files in the `urdf` directory of the `ecocar_description` package. A description of the parameters in these files can be found in Table 2.3 and 2.4.

Parameter	Description	Unit
width	Total width of vehicle body	m
length	Total length of vehicle body	m
height	Total height of vehicle body	m
rear_track_width	Distance between rear wheels	m
front_track_width	Distance between front wheels	m
wheelbase	Distance between front and rear axle	m
wheel_mass	Mass of each wheel	kg
wheel_radius	Radius of wheels	m
wheel_width	Effective wheel width	m
body_mass	Total body mass	kg
body_above_ground	Distance from ground to undercarriage when resting on wheels	m
wheel_mid_offset	Offset between middle of the four wheels and middle of vehicle body	m
wheel_joint_damping	Joint velocity dependent resistance to motion	N m / s
wheel_joint_friction	Joint constant resistance to motion	N m
max_turn	Limit of steering joints, ±	degrees
velodyne_lidar_frequency	VLP-16 LiDAR rotation frequency	Hz
velodyne_lidar_samples	Number of horizontal rotating samples for VLP-16 LiDAR	

Table 2.3: Simulation parameters in the `params.xacro` file.

Parameter	Description	Unit
steering_p_gain	Proportional gain for internal steering PID	N m
steering_i_gain	Integral gain for internal steering PID	N m/s
steering_d_gain	Derivative gain for internal steering PID	N m s
steering_max_force	Max torque to exert on each steering joint	N m
max_brake_pressure	Maximum brake system pressure	bar

Table 2.4: Simulation parameters in the `ecocar.gazebo` file.

2.2 Implementation

The simulation environment is built up around ROS and Gazebo 7. An overview of the structure is shown in Figure 2.2. The Ecocar model is loaded into Gazebo together with its own plugin. The model consists of joints and sensors as specified in the URDF file. The sensors have their own plugins which generate the sensor data. Most sensors one can think of can already be found in Gazebo, and they can be used directly with minor setup. The plugin for the Ecocar is a ROS node, which makes it able to connect to other ROS nodes.
When launching Gazebo, a world file is loaded which specifies the objects and parameters of the world, and this file is used to define the track in Gazebo. The world can also have its own plugin to enable dynamic worlds. The world plugin is used in this project to spawn the barriers in the world. This could also be done by specifying the position of each barrier directly in the world file, but then the barrier positions would have to be calculated beforehand, which this plugin does on launch.
The source for the Gazebo plugins reside in the `plugins` folder in the `ecocar_gazebo` package. The plugins are C++ classes, so each have a header file (.h) and a code file (.cpp).

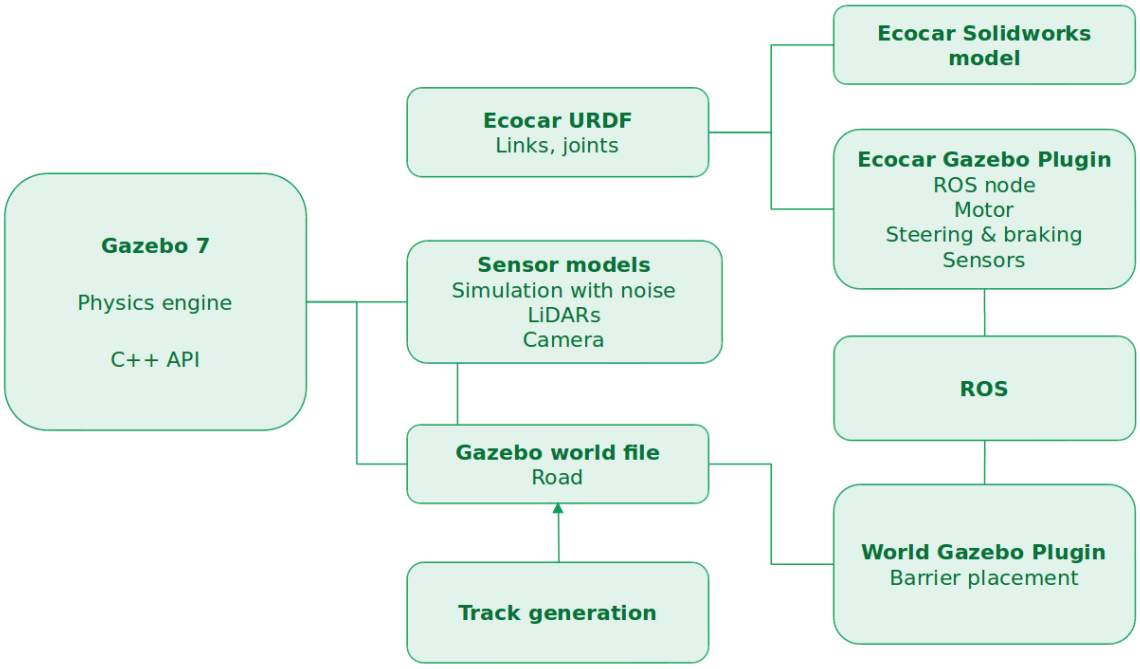


Figure 2.2: Illustration showing how the different parts of the simulation environment are intertwined.

2.2.1 Unified Robot Description Format

In ROS there exists an XML specification for describing robots, which is used by many commonly used ROS nodes. The specification is known as the Unified Robot Description Format (URDF). This file can be parsed by Gazebo to generate a robot. The file defines how links and joints of the robot are connected.

The links are entities with mass and inertia, which can be given visual descriptions either by combining simple geometries or by importing meshes. Links are connected by joints which can be either fixed or able to move relative to each other. In-depth documentation of URDF files can be found on the ROS wiki [5].
In the `ecocar_description` package, the URDF file is generated from a `.xacro` file. This is done at launch time, using the parameters in the `params.xacro` file and specifications in the `ecocar.gazebo` file within the `urdf` directory.

2.2.2 EcocarPlugin

The interfaces and dynamics of the simulated autonomous Ecocar is handled in a Gazebo plugin called `EcocarPlugin`. This plugin is also a ROS node which subscribes to the same command topics as the real car. Each part of this plugin is now explained in detail.

Motor

The motor is implemented by a torque applied on the rear right wheel. The plugin has access to each joint of the Ecocar, and is able to apply any torque to any joint.
This torque can be constant or dependent on wheel RPM, and both options are implemented.
To validate the integrity of the simulation environment, the vehicle acceleration is compared with expectations from a simple one-dimensional dynamics model.
Assuming initial velocity of zero, the work done by the motor is equal to the kinetic energy in the system:

$$\int \tau d\theta = \frac{1}{2} I_{wheels}\omega^2 + \frac{1}{2} M v^2 = \frac{1}{2}(I_{wheels} + Mr^2)\omega^2 \tag{2.1}$$

Taking the derivate on each side of the equal sign, assuming constant torque the following is obtained for the left hand side

$$\frac{d}{dt}\int \tau\omega dt = \int (\frac{d\tau}{dt}\omega + \tau\frac{d\omega}{dt})dt = \tau\int \alpha dt = \tau\omega \tag{2.2}$$

and for the right side:

$$\frac{d}{dt}\left(\frac{1}{2} I_{wheels}\omega^2 + \frac{1}{2} M v^2 = \frac{1}{2}(I_{wheels} + Mr^2)\omega^2\right) = (I_{wheels} + Mr^2)\omega\dot{\omega} \tag{2.3}$$

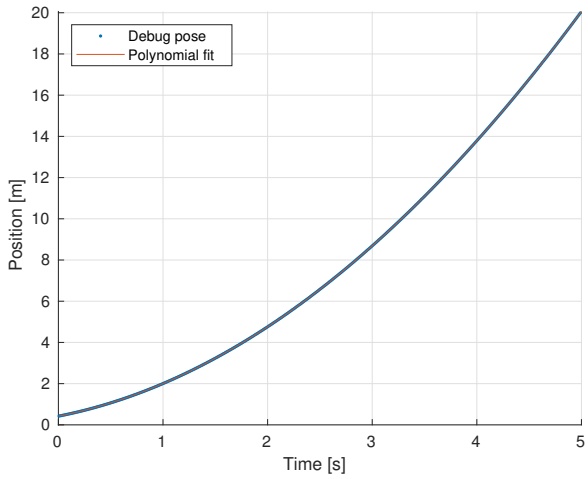
The equation is solved for the acceleration:

$$\tau\omega = (I_{wheels} + Mr^2)\omega\dot{\omega} \Rightarrow a = r\dot{\omega} = r\frac{\tau}{I_{wheels} + Mr^2} \tag{2.4}$$

This is compared to simulation data in Figure 2.3. The simulation gives debug pose on the `/sim/debug/pose` ROS topic, which is fit to a polynomial to find the acceleration. The expected acceleration matches very closely to the observed.
Instead of using a constant torque model it is possible to implement a look up table of torque versus speed as the one shown in Figure 2.4. This has been implemented together with gear change logic, but the feature is still experimental. It can be enabled or disabled in the simulation by setting the constant `ENABLE_MOTOR_SIMULATION` in `EcocarPlugin.h`.

Steering

On the Ecocar, the steering is implemented by a rack-and-pinion mechanism. This mechanism is not modeled in the simulation, instead each front wheel is attached to its own continuous joint which rotates about the z-axis. The two wheels can rotate independently of each other, but each are controlled by a PID controller in the plugin. Gazebo has an interface to apply torque to a joint, which is used to control the wheel angles.
The reference angle to the simulation is given as the angle of a virtual front wheel angle positioned in the middle. When a reference angle is given on the steering topic, the target angle of each front wheel is calculated from ideal Ackermann equations, to put each wheel perpendicular to the instantaneous



Torque	50 N m
Acceleratio, fit	1.17307 m/s
Acceleration, theory	1.17305 m/s

Figure 2.3: Acceleration curve for a constant torque of 50 Nm.

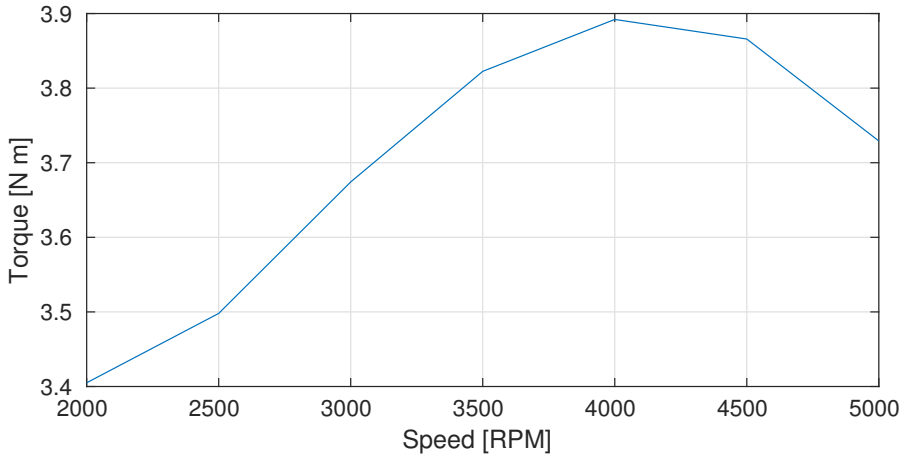


Figure 2.4: Engine torque vs. speed, data from [6].

center of rotation. An illustration of this geometry and a derivation of these equations can be found in Appendix A.1.
A step response of the steering system in simulation is shown in Figure 2.5.
The reference is on the virtual angle β which is plotted together with the angles θ_l and θ_r. This controller in the simulation is able to move the front wheels to a given position, but it is not moving in a way that is close to how the wheel turn on the real Ecocar. Further work could be conducted to mimic the speed and characteristics of the stepper motor.

Braking

In the simulation the brakes are implemented by a frictional torque on all four wheels. In Gazebo this kind of torque can be applied easily, and the difference from the normal function to apply torque is that this function will not make the car move when it is standing still.
From experiments on the car it was determined that the relationship between brake pressure and deceleration is approximately

$$a = -0.2778 \frac{\mathrm{m}}{\mathrm{s}^2 \cdot \mathrm{bar}} \cdot p \tag{2.5}$$

where p is the pressure in bar. The deceleration torque on each wheel is adjusted to match this, using the relation in Eq. (2.4):

$$\tau_{brake} = -\frac{I_{wheels} + Mr^2}{4r} \cdot 0.2778 \frac{\mathrm{m}}{\mathrm{s}^2 \cdot \mathrm{bar}} \cdot p \tag{2.6}$$

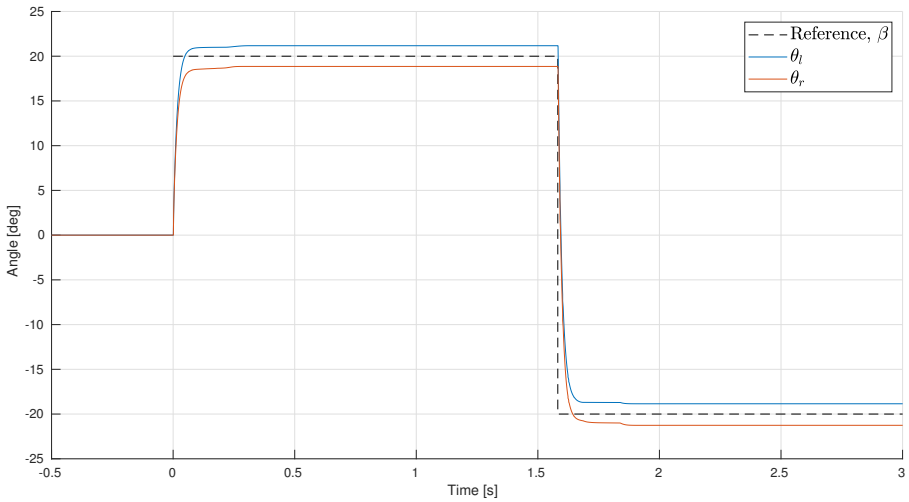


Figure 2.5: Step response of steering system in simulation.

where the factor of 4 is to divide the torque out on each wheel. The brakes are tested in simulation, which is shown in Figure 2.6. The deceleration is found from a polynomial fit of the position, which is found to agree with the theory.

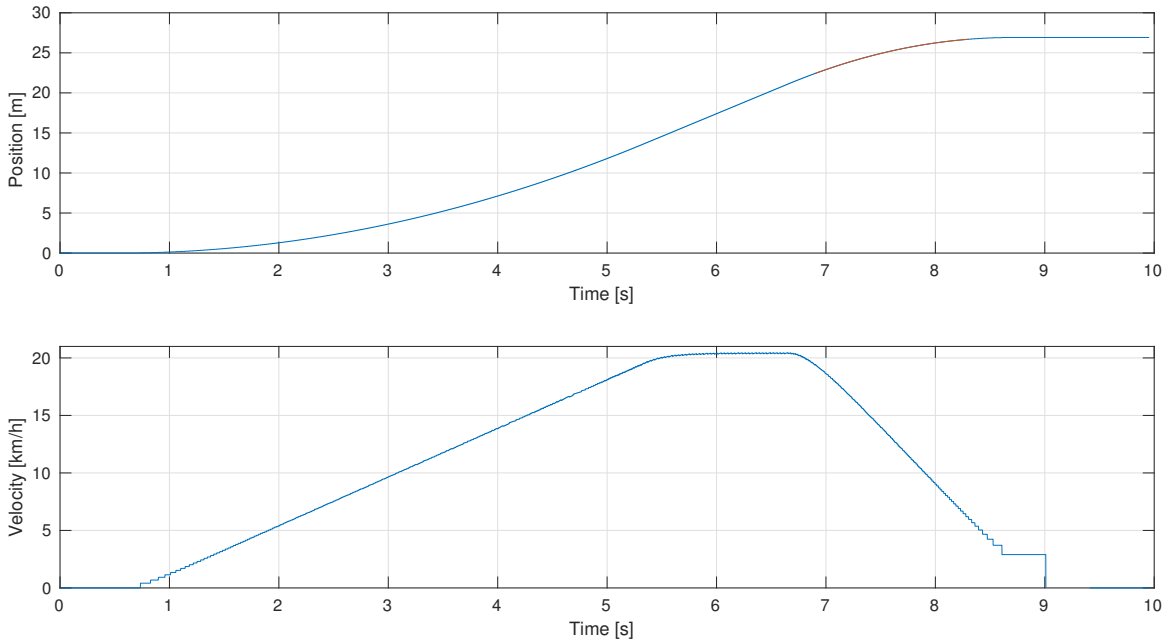


Figure 2.6: Testing the brake in simulation by accelerating to approximately 20 km/h and braking with 10 bar. From a polynomial fit of the position graph, a deceleration of $-2.778\,\mathrm{m/s^2}$ is found, agreeing very well with expectations. The part of the graph used for the fit is colored in red on the first plot.

Drag force

The drag force is modeled by the following equation:

$$\vec{F}_{drag} = -\frac{1}{2}\rho A_{front} C_d |v|^2 \hat{v} \tag{2.7}$$

where ρ is the density of air, A_{front} is the cross-sectional area, C_d is the drag coefficient and v is the relative velocity between the air and vehicle. The vector $\hat{v}$ is the normalized velocity vector, giving the direction of the relative velocity. The drag is always working against the direction of the vehicle.

The drag coefficient C_d and cross-sectional area A_{front} depends on the angle between the velocity vector and front of the car. Given the velocity vector in the frame of the car, this angle is found by

$$\phi = \text{atan2}(v_y, v_x) \tag{2.8}$$

where v_x and v_y are the x and y-components of the relative velocity vector, respectively. The drag coefficient and cross-sectional area has been found for different angles ϕ by measuring forces in a wind tunnel, and this data has been used for simulating driving patterns for the fuel efficiency challenge. A look up table for this data was originally written in MATLAB [7], but has been ported to a C++ class for use in the `EcocarPlugin`.
The drag force has been tested and a plot is shown in Figure 2.7 where the square root of the drag force magnitude is plotted against speed to get a straight line. It is compared to expectations from Eq. 2.7 for different angles of attack ϕ, and is found to be working as intended.

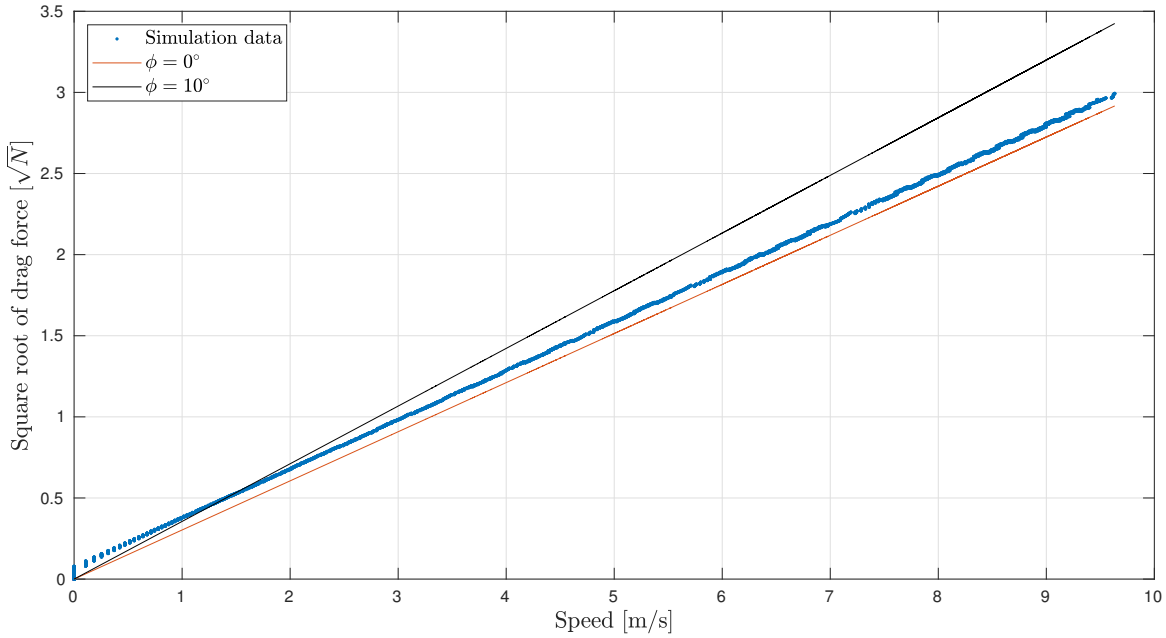


Figure 2.7: Comparison of drag force from the `/sim/debug/drag` ROS topic with analytical expression for different attack angles. The data is from a simulation of driving straight and plotted as square root of drag force to get straight lines. Numerical errors result in higher than expected drag force at low velocity, as angle of attack is never exactly zero.

The drag force can be enabled or disabled in the simulation by setting the constant `ENABLE_DRAG_FORCE` in `EcocarPlugin.h`. Debug info for the drag force calculation can be found on the `/sim/debug/drag` topic.

Rolling resistance

Rolling resistance is modeled by the following approximation

$$F_{roll} = -C_{rr} mg \tag{2.9}$$

for movement only in the x-y plane. The direction is always against the direction of motion. The coefficient depends on the radial acceleration a_r and has been modeled in the following way [7]:

$$C_{rr}(\beta) = 0.001\beta^2 + 0.002\beta + 0.0015 \ , \ \beta = 0.0425 a_r \tag{2.10}$$

In the simulation, the radial acceleration is calculated in the following way. The total acceleration is given by Gazebo. The tangential acceleration is found by taking the projection of the acceleration vector on the velocity vector:

$$\vec{a}_{||} = \left(\vec{a} \cdot \frac{\vec{v}}{|\vec{v}|}\right) \hat{v}. \tag{2.11}$$

Then, the radial acceleration is found by subtracting the tangential acceleration from the total acceleration:

$$\vec{a}_r = \vec{a} - \vec{a}_{||}. \tag{2.12}$$

The length $|\vec{a}_r|$ is then used in the calculation of rolling resistance.
The implementation of this in simulation is still experimental, and the agreement with reality is yet to be assessed. It can be enabled or disabled in the simulation by setting the constant `ENABLE_ROLLING_RESISTANCE` in `EcocarPlugin.h` and recompiling the catkin directory.
Debug info for the rolling resistance calculation can be found on the `/sim/debug/rolling_resistance` topic.

2.2.3 TrackPlugin

The fundamental assumption for the autonomous navigation is the presence of barriers on each side of the track. In the simulation, the placement of those barriers is carried out by another Gazebo plugin named `TrackPlugin`. This plugin looks for a road in the world file and places barriers on each side. Parameters are passed to the plugin from the world file. A list of the possible parameters are shown in Table 2.5.

Parameter	Type	Description
loop	boolean	Whether the track is a loop
sharpTurns	boolean	Whether the track has sharp turns. Enables more computationally heavy barrier placement algorithm.
numGaps	unsigned integer	Number of gaps to put in between barriers.
widthGaps	double	Average width of the gaps between barriers.
stdGaps	double	Standard deviation of the normally distributed gap length between barriers.

Table 2.5: List of available parameters for the `TrackPlugin` Gazebo plugin.

The texture of the road specified in the world file is drawn on the ground by Gazebo itself. The barrier positions are calculated from the same points. Firstly, more points are added with a Catmull-Rom spline interpolation. Then the edges of the road are found by offsetting the path by the vector perpendicular to the path at each point.
The angles of the individual barriers are found as the tangent to the path. When the track contains sharp turns, the offset path used for barrier placement might contain loops, which places barriers in strange positions. If the `sharpTurns` option is specified, an algorithm tries to find these loops and eliminate them, to make the barrier placement better. This algorithm is computationally heavy and increases the launch time of the simulation by a few seconds.
When the path for the track edge is found, the following algorithm is used to select the coordinates for each barrier (illustrated in Figure 2.8):

1. Place first barrier at first point, calculate end of barrier coordinates. The barrier is put tangent to the line.
2. Find point in path where the placement of the barrier would place the start of the barrier closest to the end of the last barrier by going over the points, but look no further ahead than the length of two barriers.
3. Place barrier at this position and calculate end of barrier coordinates.
4. Go back to 2. if there are more than 10 points left in path.

The placement of gaps between barriers are done at the same time, if `numGaps` is larger than zero.
An image from the simulation environment showing the barriers is shown in Figure 2.9 beside an image of the barriers on the test track in Figure 2.10.

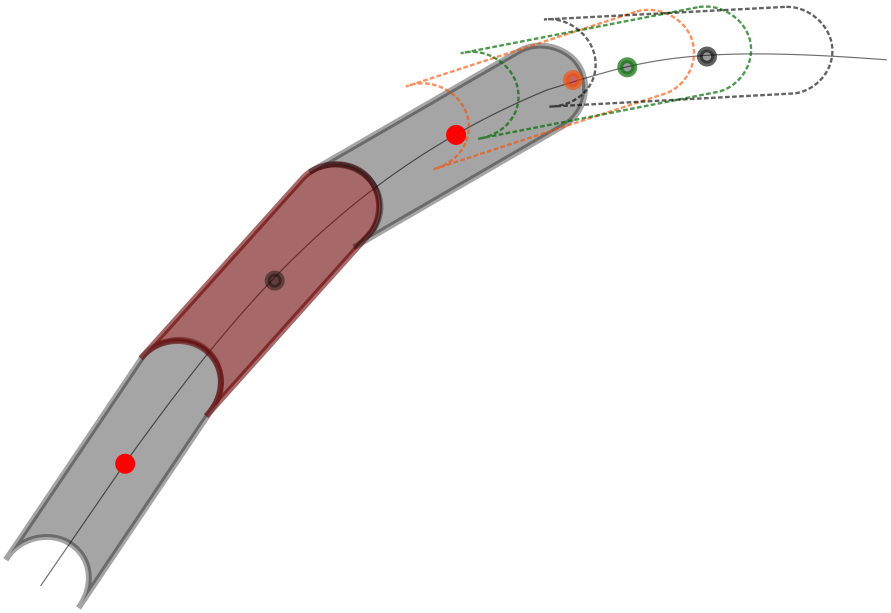


Figure 2.8: Illustration of barrier placement algorithm. The dashed lined orange, green and black barriers are candidates in the algorithm. The middle of each barrier (colored dot) is placed on the line, and the barrier with the start of the barrier closest to the previous barrier is found and placed.

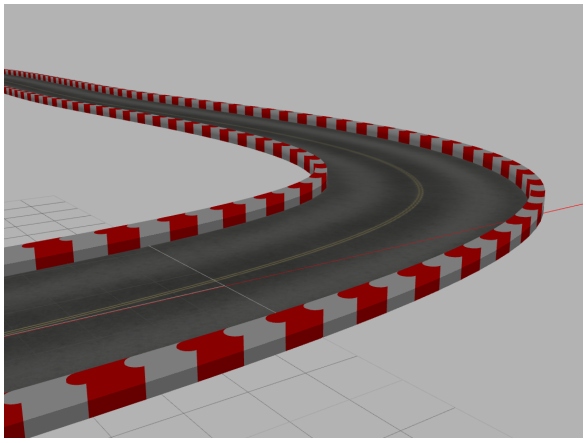


Figure 2.9: Screendump from simulation.

Figure 2.10: Image of test track.

2.2.4 Sensors

One of the most important parts of the simulation environment is the simulation of realistic sensor data, as the nature of this data will determine the usability of testing navigation algorithms in simulation. The implementation of each sensor will now be described.

LiDAR

For simulation of the VLP-16 LiDAR, a Gazebo plugin exists already, which can be found here. A copy of it has been put in the `catkin_ws_sim` workspace. To obtain the latest versions of this plugin, make sure to pull it from the source.
When the LiDAR plugin is used, the simulation run time increases significantly because it is computationally expensive to calculate all the points in the point cloud. Support for GPU acceleration of the calculation has recently been added to the plugin, but is yet to be tested in this project.
The parameters for the LiDAR simulation are set in the `params.xacro` file, see Table 2.3. The simulation runs faster if the samples are decreased from the default of 1875 horizontal rotating samples. A visualization of the LiDAR data from simulation is shown together with a model of the vehicle in

Figure 2.11.

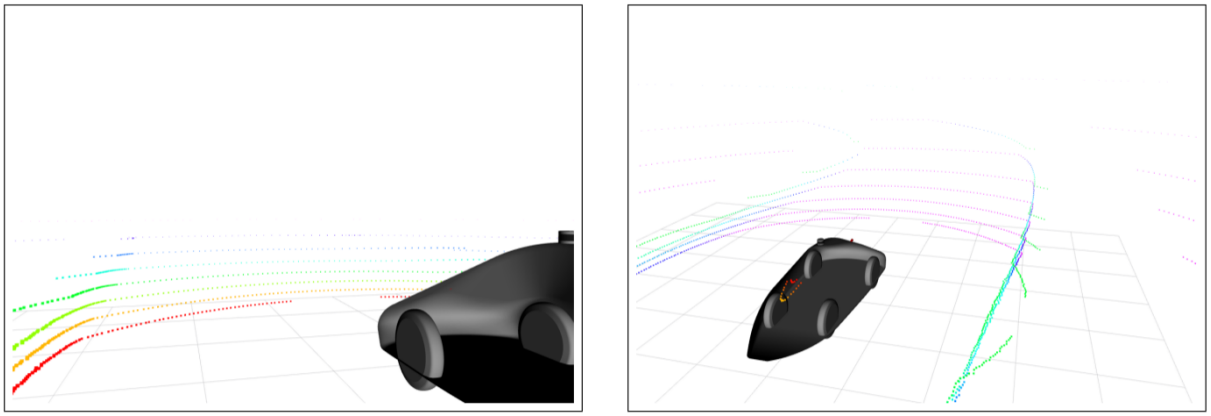


Figure 2.11: ROS visualization of simulated LiDAR data with vehicle seen from different perspectives.

Camera

Camera sensors are easily added with built in Gazebo functionality and a camera has been added on top of the vehicle. A picture from this camera simulation is shown in Figure 2.12. As can be seen, the tip of the car can be seen on the picture, which is not present on the pictures from the real camera on the car, as seen in Figure 2.13.

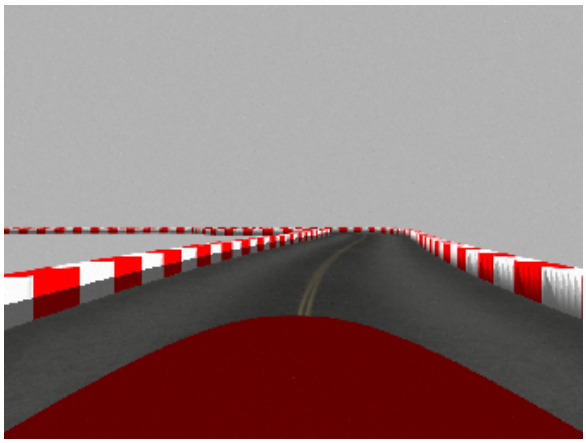


Figure 2.12: Picture from camera in simulation.

Figure 2.13: Picture from real camera.

The camera has not been used for any testing in the simulation yet, and care should be taken to ensure correct positioning and parameter values in the configuration files. The position is set in `ecocar.xacro` and parameters in `ecocar.gazebo`.

Wheel encoder

The wheel encoder code resides in the EcocarPlugin class. The wheel rotation is obtained by integrating the velocity given by Gazebo and converting this to discrete encoder ticks. The value is then converted to a driven wheel distance and published on the `/sim/raw/distance_wheel` topic. A comparison of this sensor data with a debug pose from Gazebo is shown in Figure 2.14, which shows the discretization of the odometry distance.

Gyroscope

Gazebo has a built in IMU sensor, which is used and publishing on `/sim/raw/imu`. The settings are in `ecocar.gazebo` in the `ecocar_description` package. The topic organizer is in charge of converting

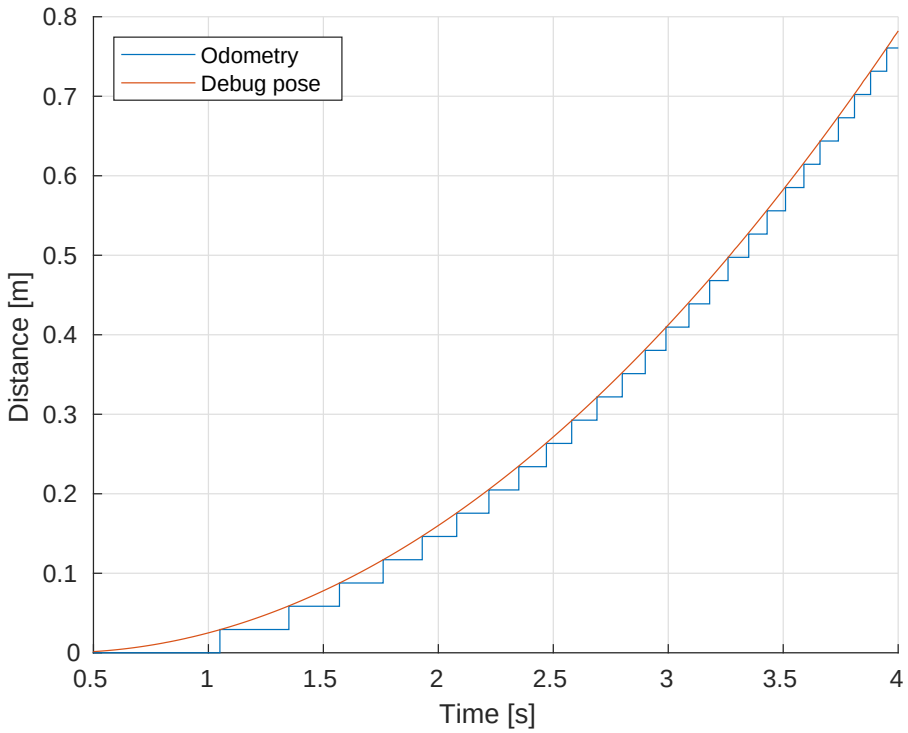


Figure 2.14: Simulation of encoder data with discrete steps in distance compared to a debug position from Gazebo.

this data to the same type of angle and unit as the real gyroscope.
The angles from the IMU are so called quaternions, while the gyroscope node only reports the yaw Euler angle. This can be calculated by [8]

$$\sin\theta = 2(q_w q_z + q_x q_y)\ ,\ \cos\theta = 1 - 2(q_y^2 + q_z^2) \tag{2.13}$$

$$\theta = \text{atan2}(\sin\theta, \cos\theta) \tag{2.14}$$

where q is the quaternion.

2.2.5 Topic organizer

The data from simulation is scattered around on different ROS topics, and to organize them into the same units and topics as on the real car an extra node was made to take care of this. The flow of data in the simulation is shown in Figure 2.15.
The IMU data is calculated into one angle (as explained previously) and published on the `/gyro_angle` topic. The rest of the sensors are combined and put into their respective fields in the `/teensy_read` topic.

Topic	**Direction**	**Data**	**Handler**
/teensy_read	From simulation	Encoder distance, speed, gyroscope, single point lidars	topic_organizer
/teensy_write	Input to simulation	Burn	EcocarPlugin
/cmd_steering_angle	Input to simulation	Steering	EcocarPlugin
/cmd_brake_power	Input to simulation	Braking	EcocarPlugin
/sim/velodyne_points	From simulation	Velodyne LiDAR	Velodyne Simulator

Table 2.6: Overview of simulation topics.

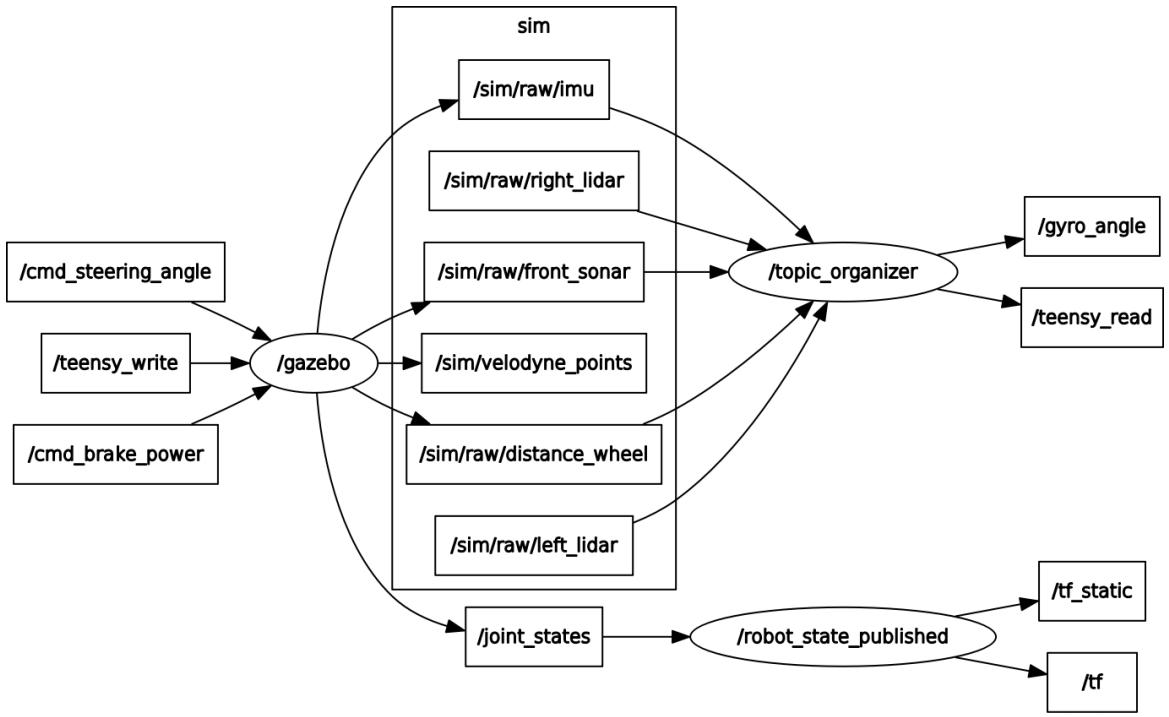


Figure 2.15: Graph of how the data flows through the simulation. Made with rqt_graph.

2.2.6 Challenge 1 track generation

For challenge 1, a method was implemented to generate random tracks. The algorithms used are heavily inspired from some found online [9]. The tracks can be used to test the autonomous navigation system on an unlimited number of test scenarios. The tracks generated are all in the $z = 0$ plane and has no splits.
The procedure to generate a track is as follows

1. Generate N random points in the 2D space.
2. Take the convex hull of these points.
3. Sort the points counter-clockwise with respect to (0,0).
4. Push the points apart if they are closer than `mindist` (function is called 3 times).
5. Add points in between each point, displaced a distance from the middle between the two points, to allow the track to go inwards.
6. Again, push the points apart if they are closer than `mindist` (function is called 3 times).
7. Increase the number of points with a Catmull-rom spline interpolation.
8. Displace the whole track such that some point of the track is at (0,0).

The track points are then exported to a `.world` file which can be loaded in Gazebo. The `TrackPlugin` takes care of placing the barriers. All the parameters for road generation are listed in Table 2.2. The implementation is done in Python and can be found in the `scripts/generateTrackPoints.py` file in the `ecocar_gazebo` package.

2.3 Future work

The simulation environment is still in active development, as it is a huge endeavor to be able to simulate a vehicle in an accurate way. Below are listed several aspects of the simulation which would be essential to work on.

- Work on the steering algorithm to mimic the stepper motor on the real Ecocar.
- Work on the camera setup to ensure the position matches that on the real Ecocar, and create challenges with textures to detect on the ground or on signs.
- Validate the authenticity of the rolling resistance implementation. The interaction between the tires and the ground is already modeled in Gazebo, which means that the implementation mentioned here might not be needed, and the parameters in Gazebo just have to be tweaked.
- Implement suspension on some or all of the wheels. It was observed that tracks with elevation can be problematic in simulations without suspension.
- Try out GPU acceleration of the LiDAR plugin as the computation speed is quite slow without it.
- Setup a PC to automatically run simulations on a bunch of randomly generated tracks each time the navigation code is changed on Bitbucket.

Chapter 3

Launching of the autonomous system

Software has been developed for easy launching and stopping of the autonomous system. A ”dynamo helper” ROS node is responsible for this process. The code resides in the `dynamo_helper` package, which is written in C++.
The autopilot button on the steering wheel transmits a message on the CAN bus, which is received on the PC from the Teensy node. When the PC is booted, the Teensy node and the dynamo helper is automatically started. The dynamo helper then listens for messages from the Teensy node, and acts accordingly.

3.1 Autopilot configuration file

For the different challenges at Shell Eco-marathon, it is convenient to do a little pre-planning. This amounts to not looking for parking spots or gates if there is none in the challenge, or selecting desired vehicle velocities based on the overall track difficulty. When the autopilot activation command is received from the Teensy node, the dynamo helper will find the USB with the configuration file and launch the executables listed in the configuration file.
The configuration file is a comma-separated file of strings enclosed in quotation marks. Each string is a command that will be executed with ”rosrun” when the autopilot button is pressed.
To add an executable to the launch process, it simply needs to be added in this file, by writing in quotation marks what one would write in the terminal after ”rosrun” to launch it.
Below is shown an example of an `autopilot.conf` file:

```
"gyro gyro_node",
"navigation estimate_pose",
"velodyne_driver velodyne_node _model:=VLP16",
"usb_cam usb_cam_node _image_width:=1280 _image_height:=960 _framerate:=15"
```

When modifying configuration files, one should check the syntax of the file with the command

```
rosrun dynamo_helper check_usb
```

3.2 Node parameters in the configuration file

To be able to pass parameters through the autopilot configuration file to each node, the following procedure can be used when developing ROS nodes in C++:

1. Initialize the node handle with a tilde, as

   ```
   ros::NodeHandle nh("~");
   ```

2. Add a slash in front of each topic you subscribe or publish to in the code, like `/teensy_read`, NOT just `teensy_read`, otherwise it will subscribe or publish to a local scope.

3. When starting the node, pass the argument as

```
rosrun <package> <executable> _[variable name]:=[variable value]
```

4. Parse each argument in the node in the following way:

```
double my_param;
nh.param<double>("[variable name]", my_param, [default value]);
```

3.3 Implementation

Functions for locating and loading configuration files are implemented in its own code file included as `usb_utilities.cpp`. These functions can also be used in other nodes if needed. The implementation will now be explained in detail.

3.3.1 Detection of the removable USB storage

The removable USB storage is found by looking in the `/dev/disk/by-id/` directory for files beginning with `usb`. These files are symbolic links to the device partition files, for example `/dev/sdc1`. The partitions are automatically mounted in the linux file systems, but the path where they are mounted has to be found. This information can be found in the `/etc/mtab` file, which is parsed to find the path from the given device partition file. Each attached USB device is then searched for a configuration file with the correct filename in the root folder, named `autopilot.conf`. The function will return the first such file that it finds, ignoring all other.

3.3.2 Configuration file validation

A software tool has been developed for validating the autopilot configuration file before trying to launch the car with it. The tool can be run on another computer where the development environment is downloaded and compiled. The syntax of the file is checked, and each string is checked to see of the executable it is trying to launch exists on the development system. In this way typos can be easily found and eliminated.

3.3.3 Starting the autopilot

When the autopilot button is pressed, the USB is detected and the configuration file found. Each ROS node is started one by one using `rosrun`. Two additional nodes are always started, a camera image processing node (because it proved to be problematic to pass an extra environment variable to it with the configuration file) and a `rosbag` node for logging.
All the programs are launched in their own terminal window, using the `gnome-terminal` command.
The `rosbag` node is configured to log everything, but excluding raw camera data using a blacklist because logging that data would result in very large log files. Compressed image files are saved, and the saved rosbags take up approximately 5 MB of hard disk space per second.

3.3.4 Stopping the autopilot

When the autopilot button is pressed again, to deactivate it, the dynamo helper will shut down all nodes not in a whitelist. The whitelist is defined as a vector of strings in the code. Each running node is compared against the list, and shut down if it is not in the list. Only the Teensy node, the dynamo helper itself and some testing tools are included in the whitelist. If an entry in the whitelist ends with an asterisk (*), it will act as a wildcard, which can be useful for some nodes which add seemingly random numbers after the main node name.

3.3.5 Additional functionality

The `dynamo_helper` also takes care of restarting the Teensy node if it is stopped for any reason. The Intel NUC has a speaker connected through an USB sound card which is useful in many situations. It uses a text-to-speech engine; below is a list of things it will say:

- If the Teensy node stops publishing messages, it will say ”Problem with Teensy”.
- In case of low voltage on the battery for the autonomous system, it will say ”Battery voltage low”.
- It notifies the passenger if the autonomous system is about to engage the brake.
- It will say anything published to the `/speak` topic.

Chapter 4

Steering

The autonomous steering mechanism is an essential part of an autonomous car. It has to be able to react promptly and in a safe manner to ensure that the vehicle is able to avoid obstacles and follow a given path in a precise and safe way. The autonomous steering system is described in this chapter.

4.1 Overview

The Dynamo 14.0 utilizes a rack-and-pinion steering mechanism to turn the front wheels. For autonomous steering, a stepper motor is attached to the pinion with a slip gear and the motor has an encoder for feedback. A potentiometer is attached to the rack to measure the absolute position of it. Both the encoder and potentiometer is connected to the Auto board. The values are received by the Teensy node on the Intel NUC and available on the `/teensy_read` ROS topic [3].
A steering node has been developed in ROS to take care of the autonomous steering operation. The steering node accepts a reference angle in degrees on the `/cmd_steering_angle` topic, using the `dynamo_msgs/SteeringStepper` message type. The steering node will move the wheels to the reference if the `steering_stepper_engaged` bool in the message is `true`.
For calibration purposes, two ROS services exist. If the potentiometer placement has been altered, run the `/calibrate_steering service`. If a heavier than usual person or if substantial changes has been made to the car which would result in increased wheel friction, run the `/calibrate_steering_velocity` service. If changes has been made to the steering geometry, the parameters in `SteeringSystem.h` has to be updated.
The steering ROS node is written in C++. The main node is in `steering_node.cpp`, which takes care of the communication with ROS, but the controller is implemented in its own class named `SteeringSystem` which is included in the node.

4.2 Steering system design

The steering node operates around states as shown in Figure 4.1. When the node starts it initializes the connection to the stepper motor and waits for the first measurements from the Teensy node. If a calibration file exists, it will go into working mode (OK) where it is able to send commands to the stepper motor. Each time sensor data is received, the data is checked against hard coded limits. If the potentiometer is not connected, this check will ensure that no commands are sent to the stepper motor when there is no feedback. This is also the case if messages stop arriving from the Teensy node for any reason.

4.2.1 Sensor feedback

The steering system has three feedback values:

- The current stepper motor step position, provided by the stepper motor library. This value is where the library thinks the motor is, but it is incorrect if steps are skipped.
- The number of ticks from the optical rotary encoder attached to the stepper motor.

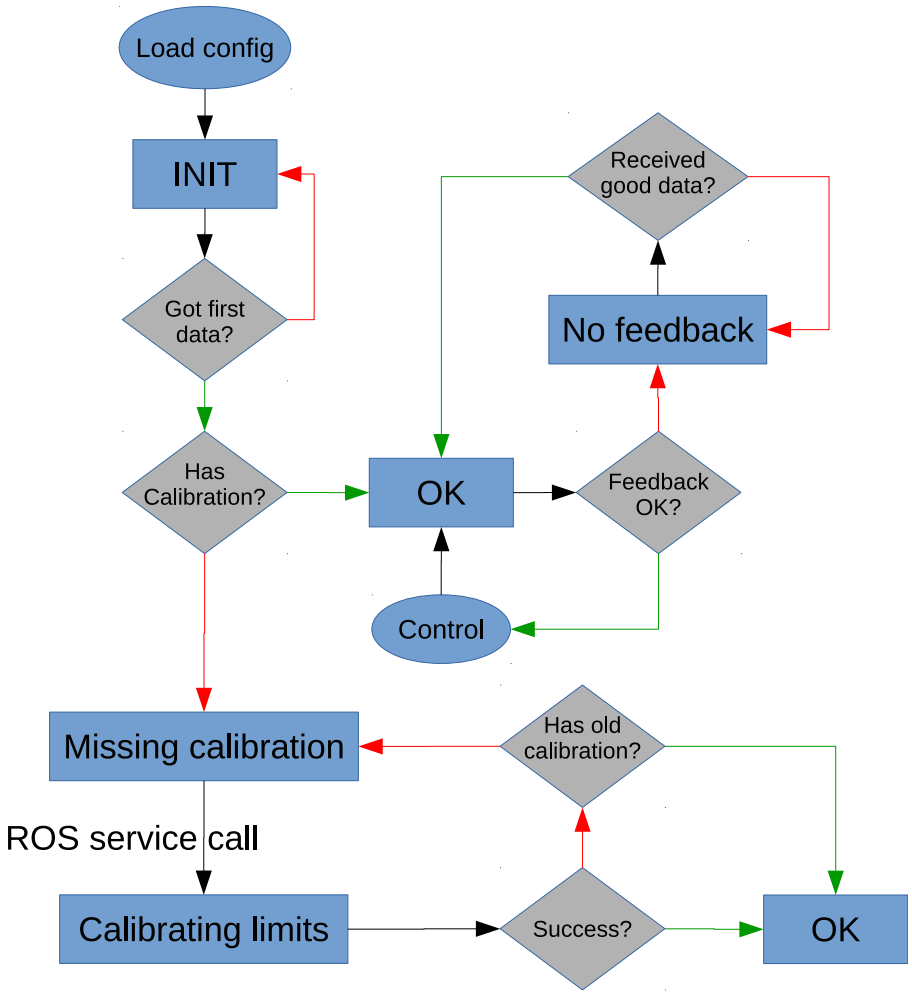


Figure 4.1: Steering node state diagram. Boxes denote states, diamonds are conditions and ellipses are actions. Only the most central actions and states are shown in the figure.

- The voltage from the potentiometer on the rack. This gives the absolute rack position, from which wheel angles can be calculated.

The step position is given in units of microsteps, and the stepper does 16 microsteps per step and has 200 full steps per revolution. The encoder gives 1200 ticks per revolution. This gives a useful ratio in calculation between stepper steps and encoder ticks as:

$$\frac{1200}{200 \cdot 16} = 0.375. \tag{4.1}$$

The potentiometer value is a 10 bit ADC value. The conversion factor from ADC value to position is found from the measurements shown in Figure 4.2.

4.2.2 Steering angle reference

The steering reference angle is given as the angle of a virtual front wheel in the middle of the car. For the control of the wheel angles, we have to calculate the potentiometer position that will give the reference angle.
The formula to calculate the angle of the front wheels from the rack displacement is derived in Appendix A. This result is used to calculate the angles from the potentiometer measurement. The inverse formula is also derived, to calculate the required rack position from a given wheel angle. This however requires the angle of one of the front wheels, not the angle of the virtual front wheel, β. To get around this, the angle β is used as front left wheel angle to get a rack displacement s_l, and used as front right wheel angle to get s_r, and the average is used as reference rack displacement.

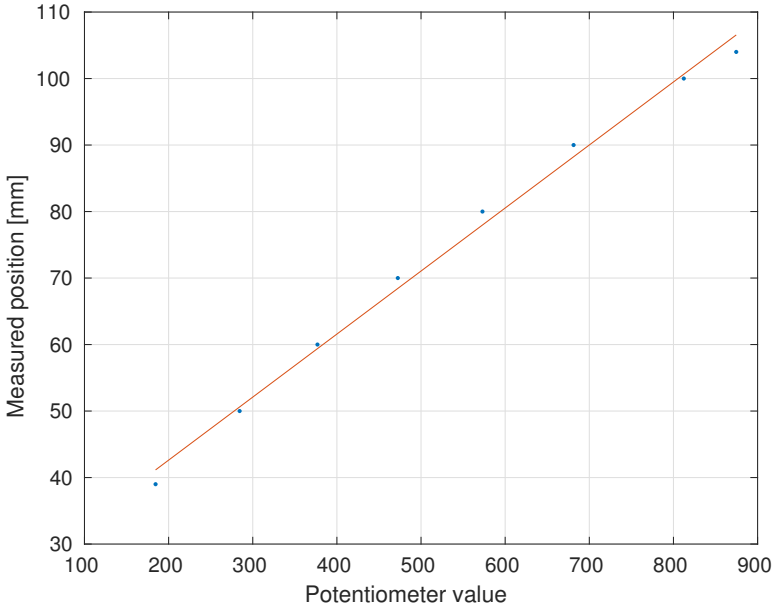


Figure 4.2: Linear fit of measured positions and potentiometer values for 8 different rack displacements. The positions were measured with a folding ruler. The slope from this plot is used to convert between potentiometer position and ADC value.

The displacement value is converted to a value on the potentiometer by using the zero value from the limit calibration and the conversion factor found by measurements. This potentiometer reference is used in the control algorithm.

4.2.3 Control algorithm

Stepper motors do not work well with classical control methods such as PID control. They are typically used in open-loop applications for position control. For the steering, the stepper is used in a closed-loop setup with the potentiometer, providing robustness against skipped steps. The stepper motor is commanded with the phidget21 library which has easy to use functions to set the target position, acceleration and maximum velocity. The stepper motor control board is connected to the Intel NUC by USB.
At each time step, the required steps are calculated from the potentiometer reference and current potentiometer value, using the effective radius of the pinion. The new target position is then set if the number of steps is more than 400, or if the stepper is not moving if it is more than 100 steps. To calculate the required number of steps, the following equation is used:

$$n_{steps} = \frac{x - x_{ref}}{2\pi r_{\text{pinion}}} k_{steps} \tag{4.2}$$

where the constant can be found by multiplying the number of full steps with the number of micro steps per step and the gear ratio:

$$k_{steps} = 200\,\text{steps/rev} \cdot 16\,\text{microstep/step} \cdot \left(15 + \frac{3}{10}\right) = 48\,960\,\text{microstep/rev}. \tag{4.3}$$

It was observed that the potentiometer value has a slight time delay when comparing with the step position given by the stepper motor control board. The limit on the smallest number of steps to move was implemented because of this, because otherwise the system would oscillate around the reference. The 100 steps correspond to $\approx 0.3\,\text{mm}$ on the rack.

4.2.4 Adaptive stepper velocity

Wheel friction depends highly on vehicle velocity. The wheel friction is largest when the car is stationary and decreases when the car is moving. When the friction is large, the stepper needs

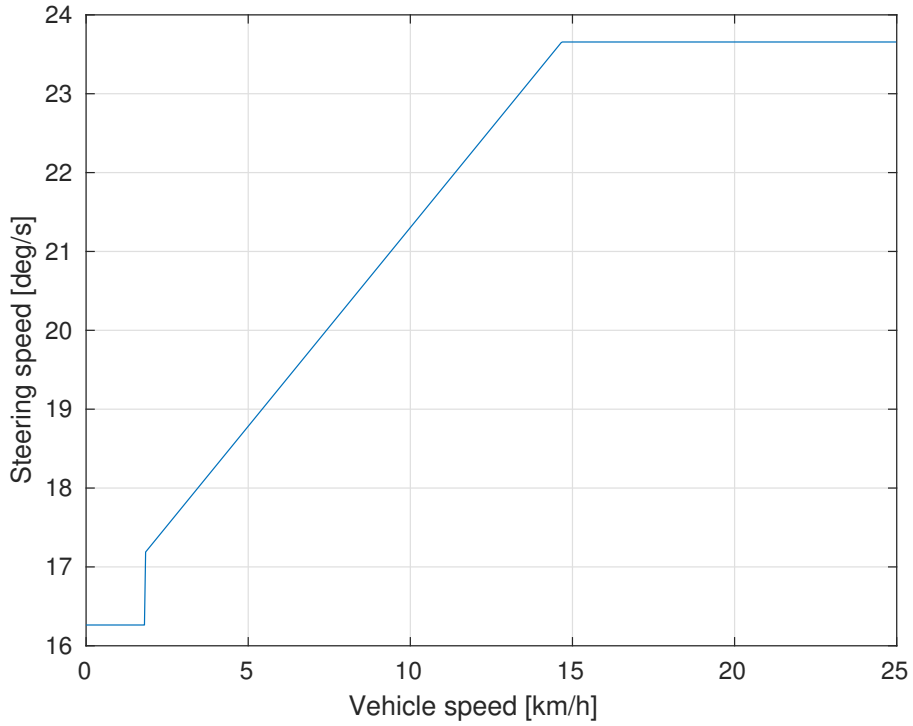


Figure 4.3: Graph of function used for adaptive front wheel steering speed.

to be run slower to avoid skipping steps. As the vehicle speed increases, the stepper motor is able to run faster, as less torque is needed. The stepper is able to turn at maximum 116 RPM at no load, and the max holding torque is 192 kg-cm.
The stepper velocity is calculated in the following way

$$\omega_{stepper} = \begin{cases} \omega_0 & v < v_{min} \\ \omega_0 + k_\omega v & \frac{\omega_{max}-\omega_0}{k_\omega} > v > v_{min} \\ \omega_{max} & v > \frac{\omega_{max}-\omega_0}{k_\omega} \end{cases} \tag{4.4}$$

where v is the vehicle speed in (m/s), ω_0 is the base turning speed found from the velocity calibration (steps/s), ω_{max} is a hard limit on the stepper velocity, v_{min} is the lower threshold at which to begin to increase the steering velocity, and k_ω is a constant.
This function is plotted in Figure 4.3 where the speed has been converted from microsteps per second to approximate degrees on the wheels per second.

4.2.5 Steering speed estimation

The stepper motor velocity is specified in units of microsteps per second when using the stepper motor library (phidget21). It is convenient to convert this velocity to units of degrees per second. The relationship between rack position and wheel angle is nonlinear, but can be linearized to obtain a simpler expression for steering speed for a given stepper velocity:

$$\frac{d\theta}{dt} = \frac{d\theta}{dx}\frac{dx}{dt} = \frac{d\theta}{dx} r_{\text{pinion}} \frac{d\varphi}{dt} \tag{4.5}$$

where s is the rack displacement, r is the pinion radius and φ is the stepper angle.
Using a Taylor approximation to first order, the derivative $\frac{d\theta}{dx}$ is

$$\frac{d\theta}{dx} \simeq \frac{-\nu}{r_0^2} - d_0 \frac{(a^2-b^2)-r_0^2}{r_0^2\sqrt{4b^2r_0^2-(b^2+r_0^2-a^2)^2}} \tag{4.6}$$

$$d_0 = \frac{W-L}{2}\ ,\ r_0 = \sqrt{d_0^2+\nu^2} \tag{4.7}$$

The derivative $\frac{d\varphi}{dt}$ is the selected stepper motor velocity in the steering node. When the steering node starts, the constant $\frac{d\theta}{dx} r_{\text{pinion}}$ is calculated after which it is used to publish the current steering velocity.

4.2.6 Fault detection

Because of the sensor redundancy we are able to detect certain faults in the system. These faults can be normal or critical depending on the situation. The ideal steering actuation system can be written in two equations:

$$\omega_{\text{enc}} = \omega_{\text{stp}} \tag{4.8}$$

$$v_{\text{pot}} = r_{\text{pinion}} \omega_{\text{stp}} \tag{4.9}$$

where ω_{enc} is the encoder measured angular velocity, ω_{stp} is stepper angular velocity, v_{pot} is the potentiometer linear velocity and r is the pinion radius.
These relations are only true if the slip gear is not slipping, and the stepper motor is not skipping steps. The equations are rearranged and filtered, to give a quantity which is zero in the nominal case:

$$r_1(s) = f(s)\left(\omega_{\text{stp}} - \omega_{\text{enc}}\right) \tag{4.10}$$

$$r_2(s) = f(s)\left(r_{\text{pinion}}\omega_{\text{stp}} - v_{\text{pot}}\right) \tag{4.11}$$

where $f(s)$ is a filter transfer function. These functions are called residuals [10]. The velocities are given in Laplace domain by multiplying the position with s:

$$r_1(s) = f(s)\left(s\theta_{\text{stp}} - s\theta_{\text{enc}}\right) \tag{4.12}$$

$$r_2(s) = f(s)\left(r_{\text{pinion}} s\theta_{\text{stp}} - s x_{\text{pot}}\right) \tag{4.13}$$

If the filter is a simple low pass filter, the residuals can be written as

$$r_1(s) = \frac{s\omega_c}{s + \omega_c}\left(\theta_{\text{stp}} - \theta_{\text{enc}}\right) \tag{4.14}$$

$$r_2(s) = \frac{s\omega_c}{s + \omega_c}\left(r_{\text{pinion}}\theta_{\text{stp}} - x_{\text{pot}}\right) \tag{4.15}$$

which is what is implemented in the steering node. The sensor values are converted to radians and millimeters for this purpose.
A change detection algorithm known as CUSUM is applied to detect when the residual shows significant deviation from zero. The recursive implementation used is given by [10]

$$g(k) = \min\left(0, g(k-1) + \frac{\mu_1 - \mu_0}{\sigma^2}\left[z - \frac{\mu_1 + \mu_0}{2}\right]\right). \tag{4.16}$$

The parameters μ_0 and μ_1 are the mean of the signal before and after the change. For this use case μ_0 is zero and μ_1 is chosen as the steering velocity. If the stepper motor is spinning, but this is not reflected on the encoder or potentiometer, the residual will show the difference in velocity between the two entities being compared in the residual.
The parameter σ^2 is the variance of the residual signal, which can be estimated from Figure 4.6 by looking at the residual signal at the time before the fault occurs.
A fault is detected when this sum is larger than a threshold h. The threshold is selected to avoid false alarms and ensure short detection time. An example of this is plotted in Figure 4.7 where $g(k)$ is plotted together with the decision function that is 1 when a fault is detected and zero when no fault is detected. It is seen the the fault is detected much faster on r_2 in this case, but that will depend on the nature of the fault.
A couple of possible faults can be detected with this method, here listing a few

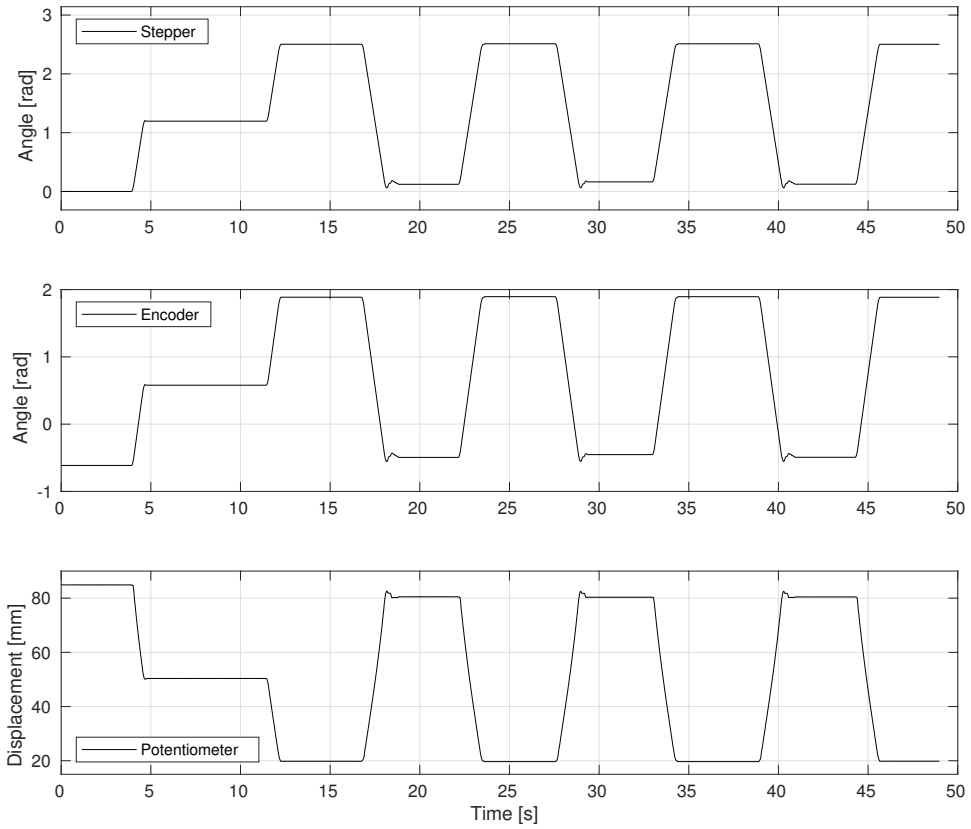


Figure 4.4: Data from sensors when turning from side to side. The raw data is converted to meaningful units. It is seen how the sensor values are correlated and have different offsets.

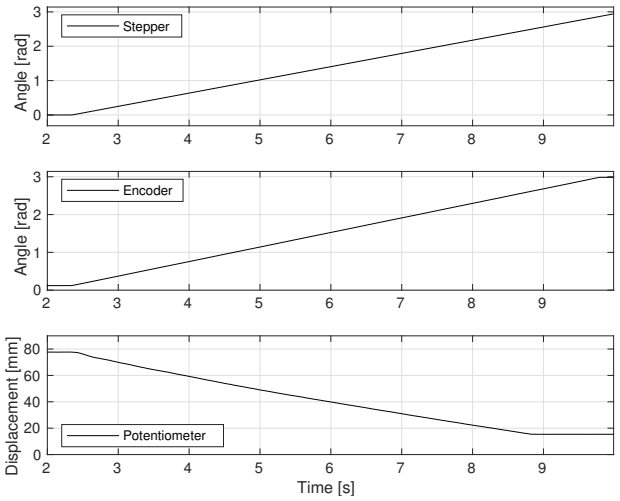


Figure 4.5: Sensor signals when hitting the mechanical limit at $t \approx 8.85\,\mathrm{s}$.

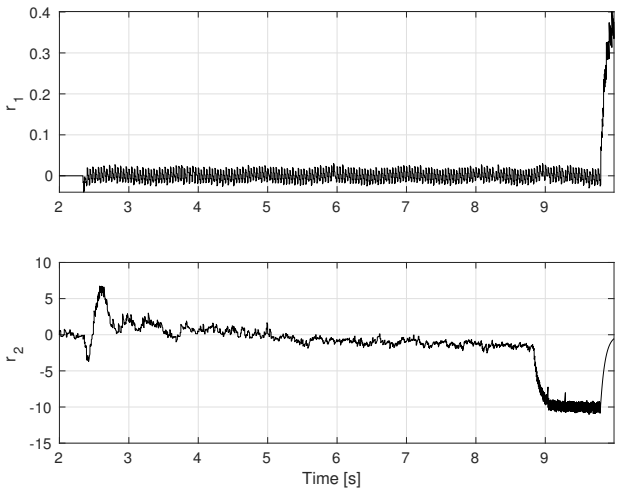


Figure 4.6: Residuals when hitting mechanical limit.

- Rack cannot move further because of mechanical limits.
- Steering wheel is moved manually while stepper motor is idle.
- Steering wheel is held in place while stepper motor is turning.

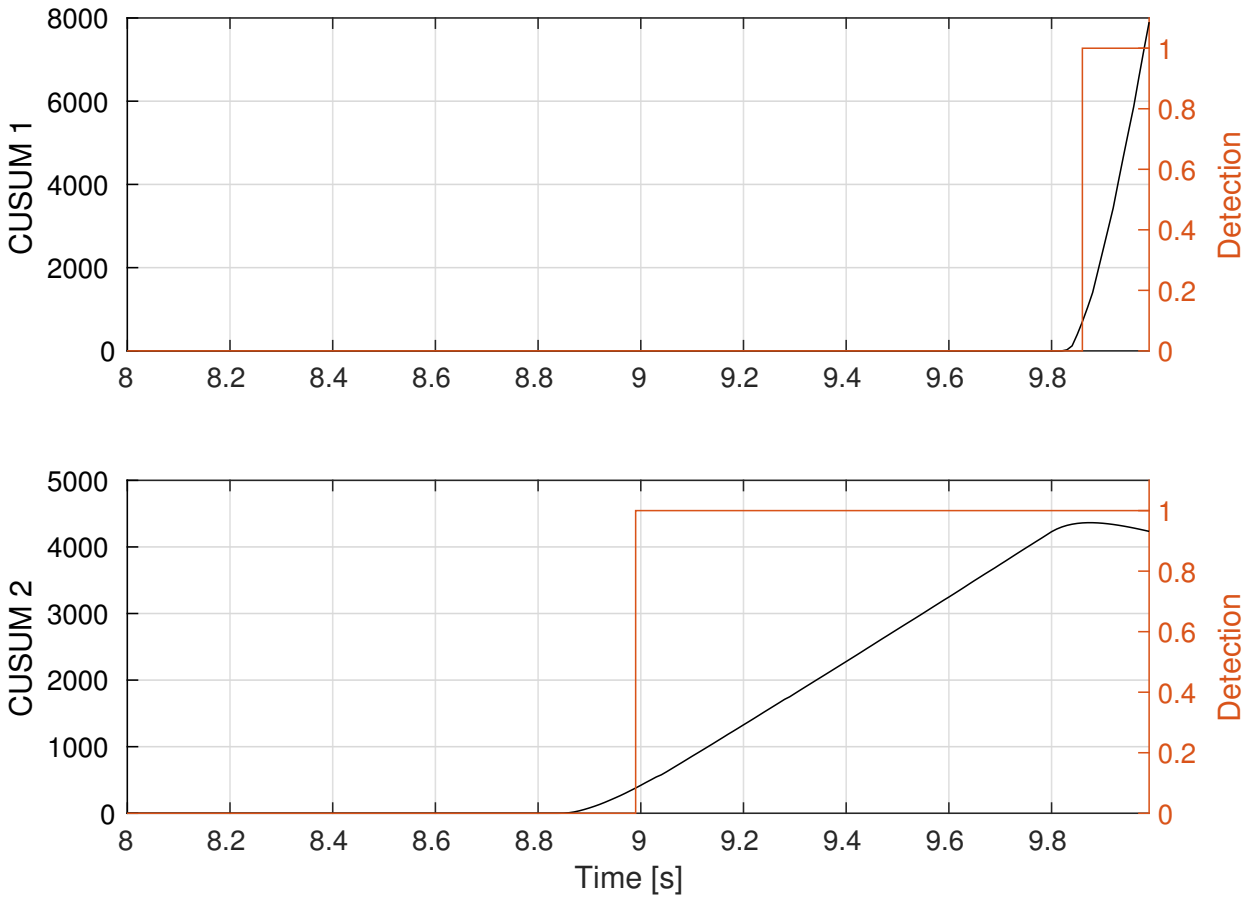


Figure 4.7: Plot of the value of $g(k)$ for each change detector when hitting the mechanical limit. The fault happens at $t \approx 8.85\,\mathrm{s}$ and is detected on r_2 after approximately 140 ms.

4.3 Calibration

Calibration of the steering system is needed in different cases. Two different calibration sequences are implemented.

4.3.1 Limit calibration

This calibration sequence finds the mechanical limits in the steering system. The limits are where the end of the mechanical coupling between the rack and the tie rod collides with the rack mount, see Figure 4.8. When turning too far to one side, this collision happens in the opposite side.
The calibration sequence uses the implemented fault detector. The wheels are continuously turned to one side, until a fault is detected because the potentiometer value stops changing while the stepper motor is turning, which triggers the detector. The wheels are then turned to the other side to find the other limit. The potentiometer values are saved at each limit, and the zero position of the wheels is assumed to be in the middle of these limits. In the steering node, the reference angles given are truncated if they correspond to a rack position out of the range of the calibration limit.

4.3.2 Turning speed calibration

For the adaptive stepper velocity, a base speed can be found which is used when the vehicle is at standstill. The limit calibration has to be done first. This sequence turns the wheels from side to side at a fast speed. If a fault is detected, which happens if the speed is too high, the speed is decreased. The sequence stops when it has turned 2 full times from right to left and back without any faults occurring. The final speed is then saved and used as base speed. This speed depends on the weight of the driver.

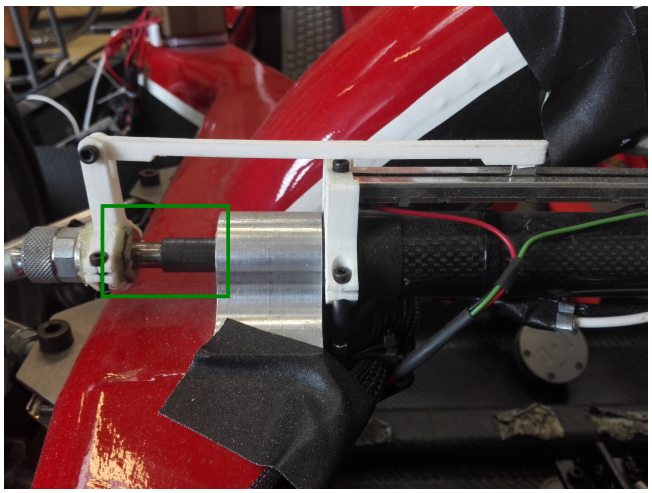


Figure 4.8: Picture showing the end of the rack where it connects to the tie rod where it is limited by the rack mounting.

4.4 Evaluation

The steering system described was used when driving autonomously at Shell Eco-marathon in July 2018. The system performed well, and the car was able to turn promptly even when going through a corner at 18 km/h. A plot of the steering angle and reference from the competition is shown in Figure 4.9, which shows that the controller was able to follow the reference throughout the whole lap.

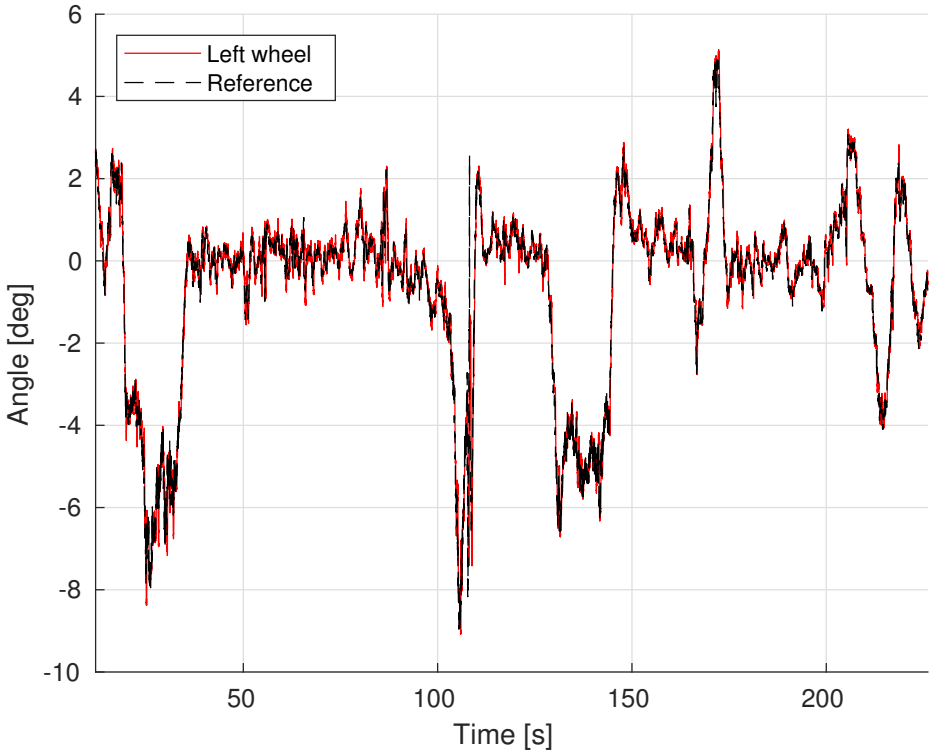


Figure 4.9: Plot of logged steering angle and reference steering angle on London track when driving autonomous. The left and right wheel angles differ by only a very small value. It is seen that the steering controller is able to follow the reference closely through the whole lap.

Chapter 5

Braking

Brief work was done on the brake system in final preparation for the Shell Eco-marathon 2018. The brake system was not performing satisfactory because of unfortunate control design decisions and thus a temporary work-around was made.

5.1 Overview

The brake system consists of two pistons and disc brakes on each wheel. The brake pedal pushes the piston in, which increases the pressure in the system. The pressure system is separated in two (for the front and back wheels), and the pistons are each connected to one part of the system. Pressure sensors on each system provide feedback in the range of -1 to 34 bar, which can be found on the `/teensy_read` ROS topic.
The autonomous brake system consists of a linear actuator from Linak that engages the same brake piston as the brake pedal. The mechanical system is made such that the driver can brake even if the autonomous system is not braking.

5.2 Controller

A control box was received together with the actuator which is able to regulate its position, but using this box proved to be a less than ideal choice. In cases where it is desired to increase the pressure slightly, the actuator only has to move slightly, but the controller does not follow small changes in the reference. A PID control loop was tested, controlling the position reference to the box, using the pressure error, but it did not work well with the position controller. A much better solution would be to control the actuator voltage directly.
Because of these problems and time constraints, an open loop control scheme was deployed. In this scheme, the brake has to be calibrated at least once each day, preferably more often. The calibration consists of moving the actuator out in small steps and noting the pressure in the system at each step. The process is done automatically with a ROS service call, and saves the calibration data to a text file. The calibration data is then used as a look up table for the open loop control. Linear interpolation is used between calibration points. The calibration increases the position of the actuator until it reaches a preset limit. For every position, the actuator moves back to zero before moving to the next step, because the steps are too small for the position controller to move directly from one to the next.

5.3 Evaluation

The brake controller described in this report was used at the Shell Eco-marathon 2018 when driving autonomously. Plots showing the performance is seen in Figure 5.1 and 5.2.
As can be seen, the reference pressure is changing quite often because of decision making problems in the navigation system. However, the plots show that the actuator starts moving a very short time after the command is received, and the pressure in the system increases accordingly. As the system is open-loop, it is not surprising that it is not able to track the reference very well, however the system

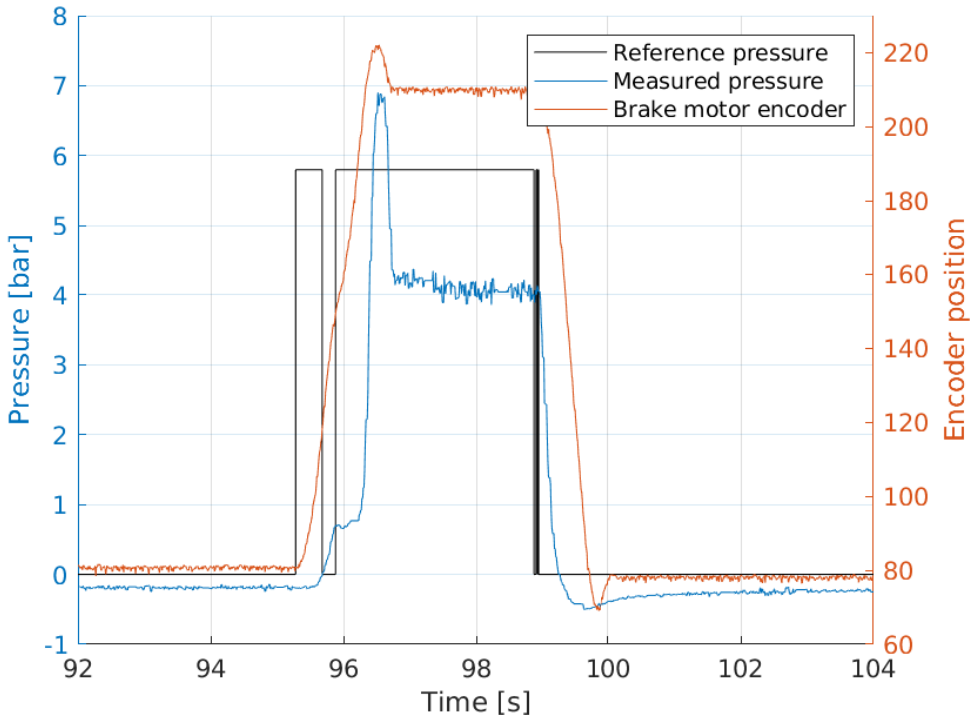


Figure 5.1: Plot of braking log data from Shell Eco-marathon 2018 in London. At this point on the track the car detects a bend coming up on the track and aims to decrease the speed. The command to brake is initially sent for less than half a second, after which the command to brake is dismissed for a short time, and then the command to brake is sent again. The brake node reacts on the first command and starts to brake, and the short duration where the command is dismissed is not long enough to affect the braking system.

is able to brake when asked to, although it will not brake with the exact pressure asked for. When compared to the previous controller [11], improvements are seen, as the actuator does not show the same behavior where it moves in steps, stopping at every step for quite some time before reaching an acceptable pressure. This means that the braking controller was able to react faster, which is quite important if the Ecocar has to be able to succeed in challenges in the future.

5.4 Future work

As mentioned, the controller described here was a temporary fix before the competition. In the future, the braking action should be controlled by a closed-loop controller which regulates the speed of the actuator from the pressure error. A more robust braking solution is necessary for the Ecocar to succeed in future competitions.

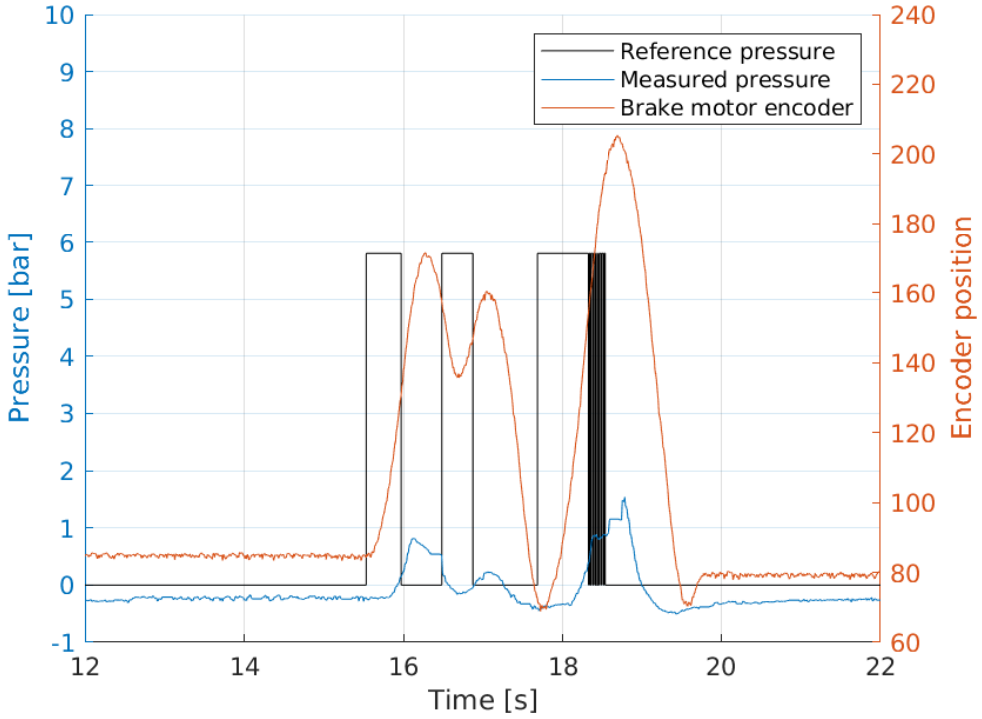


Figure 5.2: Plot of braking log data from Shell Eco-marathon 2018 in London. The plot shows that the braking node is quick to react even in cases where the reference pressure changes quickly. However, the braking action is not very effective in this case, as it takes a considerable amount of time to engage the brake compared to the fast changes in reference pressure.

Chapter 6

Velocity estimation

When driving the Ecocar, in autonomous mode or not, it is essential to have a measurement of the vehicle velocity. This chapter is about improving the velocity estimation algorithm for more robust performance under any conditions.

6.1 Motivation

The rotation of the rear right wheel on the Ecocar is measured with an inductive encoder on a disk on the wheel. It has been observed that the sensor will sometimes give erroneous readings, and the distance measurement will display large jumps. It was decided to design a filter to counteract this to improve the robustness of the systems relying on the velocity measurements.

6.2 Derivation and implementation

To counteract the problem with false readings, the encoder ticks are filtered based on the time since the previous step. If the ticks are spaced so closely in time, that they correspond to a vehicle velocity higher than 54 km/h, they are discarded.
The encoder ticks are counted on a hardware interrupt, and the difference in this value is checked every 1 ms. If the sensor problem occurs, the ticks are discarded. Otherwise they are used to calculate the speed. Just applying this filtering method, the distances look much better as can be seen in Figure 6.1.

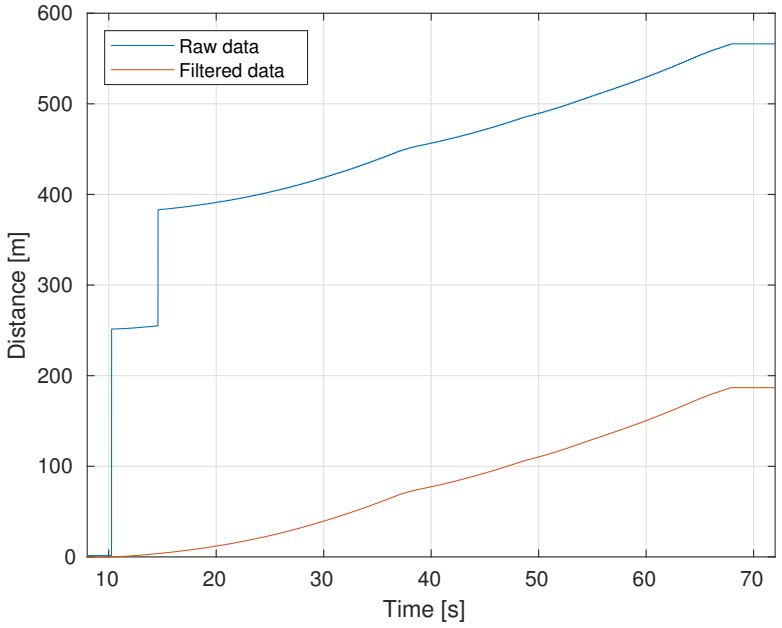


Figure 6.1: Raw and filtered distances from wheel encoder ticks.

To obtain more stable velocity measurements, a low pass filter is applied. The differentiation and low pass filtering is combined in the following transfer function:

$$G(s) = \frac{s}{\tau s + 1}, \tag{6.1}$$

where τ is a chosen time constant. This transfer function is discretized using the bilinear transform (Tustin approximation), by using the substitution [12]

$$s = \frac{2}{T}\frac{z-1}{z+1}, \tag{6.2}$$

which gives the result

$$H(z) = \frac{\frac{2}{T}\frac{z-1}{z+1}}{\tau\frac{2}{T}\frac{z-1}{z+1} + 1} = \frac{2(z-1)}{2\tau(z-1) + T(z+1)} = \frac{z-1}{\tau(z-1) + \frac{T}{2}(z+1)} = \frac{z-1}{(\tau + \frac{T}{2})z + \frac{T}{2} - \tau} \tag{6.3}$$

$$\Rightarrow H(z) = \frac{1 - z^{-1}}{(\frac{T}{2} - \tau)z^{-1} + \frac{T}{2} + \tau} = \frac{v(k)}{s(k)}. \tag{6.4}$$

The velocity at a given time step can then be calculated by

$$v(k)((\frac{T}{2} - \tau)z^{-1} + \frac{T}{2} + \tau) = s(k)(1 - z^{-1}). \tag{6.5}$$

Applying that $f(k)z^n = f(k+n)$ results in

$$v(k-1)(\frac{T}{2} - \tau) + v(k)(\frac{T}{2} + \tau) = s(k) - s(k-1), \tag{6.6}$$

which is solved for $v(k)$ to give the following filter update equation:

$$v(k) = \frac{s(k) - s(k-1)}{\tau + \frac{T}{2}} + \frac{\tau - \frac{T}{2}}{\tau + \frac{T}{2}}v(k-1). \tag{6.7}$$

The sampling time T in this formula need not be constant, which is very convenient as the aforementioned sensor problem could occur. The time T in the equation is then set to the time since the last good measurement. In this way the low pass filter takes the increased time between measurements into account. The time constant τ is selected to 200 ms. The result of using this filter is shown in Figure 6.2. Note that the plotted velocities without filter has the erroneous happening ticks filtered away, as this method was already used, but it does not use a low pass filter with varying time step.

6.3 Evaluation

This velocity estimator has been implemented on the Teensy 3.6 microcontroller on the ECU (engine control unit) of the Ecocar Dynamo 14.0. The value is transmitted on the CAN bus and used to display velocity on the steering wheel, and for speed control in the autonomous navigation. In ROS it can be found in the `speed_wheel` field on the `/teensy_read` topic in units of km/h. A plot showing the performance of the estimator at Shell Eco-marathon 2018 is shown in Figure 6.3.

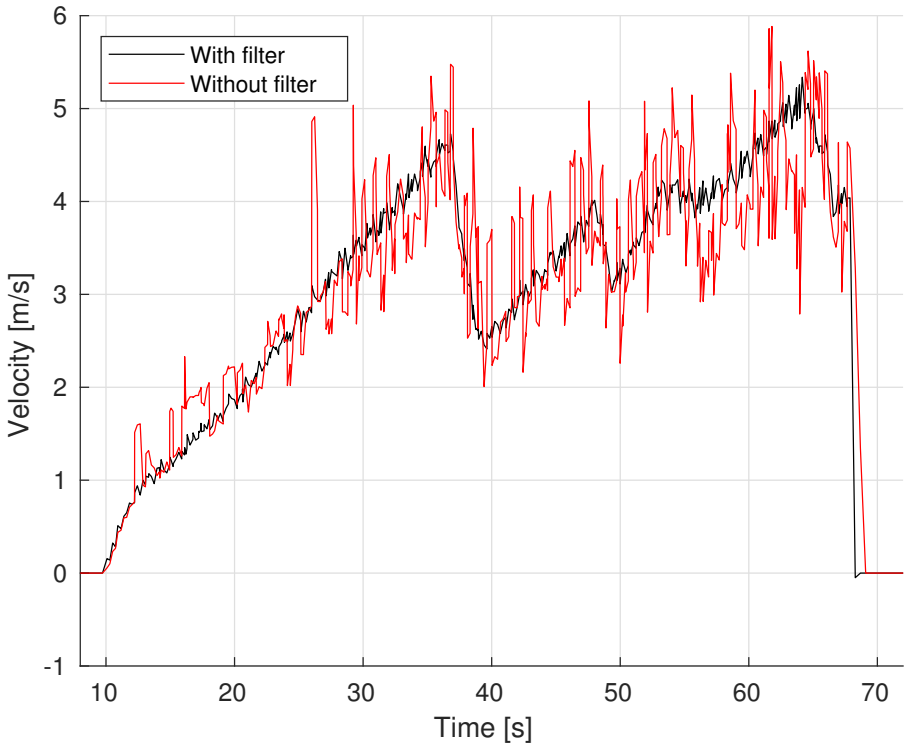


Figure 6.2: Comparison of speed estimation before and after implementation of the filter.

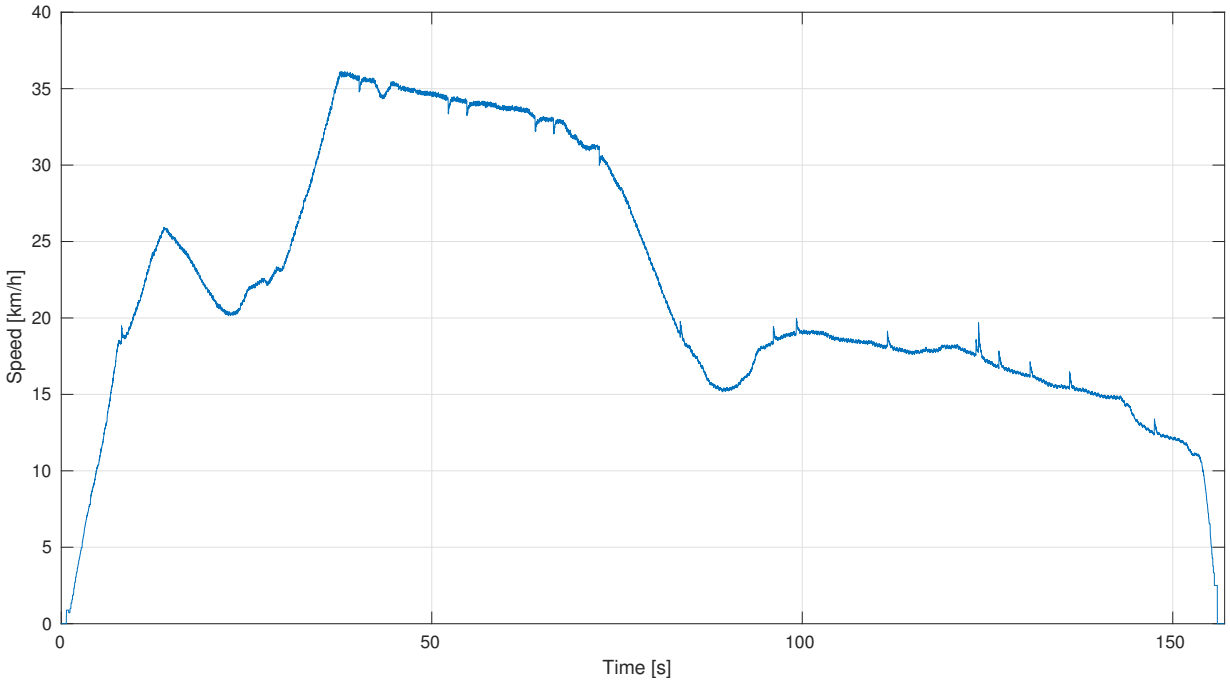


Figure 6.3: Plot of speed during one lap on the London track at Shell Eco-marathon 2018. Generally the value is smooth although small spikes happen once in a while.

Chapter 7

Conclusion

In this report, some of the work on the autonomous Ecocar Dynamo 14.0 before the competition in July 2018 has been described. The entire Ecocar was setup in simulation to enable other team members to more effectively test the autonomous navigation software without damaging people or the car itself in the initial testing phase.
The ecosystem to effectively launch the autonomous system was laid down, featuring configuration files which enables the team to quickly switch between challenges and tweak parameters when time is limited.
The software for the autonomous steering system was completely reworked to include calculation of wheel angles from the rack-and-pinion steering geometry. This implementation allows for precise autonomous navigation where the wheel angles are accurately known. Control of the steering speed was implemented as a function of vehicle velocity, to allow fast turning when driving through a curve at high speed.
The autonomous braking system was adjusted to ensure fast reaction, although it is still not able to regulate the pressure in the braking system exactly to the desired pressure, and further work is needed on this system to ensure more robust performance in future competitions.
Lastly, the algorithm for estimation of the vehicle velocity was improved to give less noisy values both when driving autonomously and with a human driver.
These improvements played a key role when DTU Roadrunners won the category of Autonomous UrbanConcept at Shell Eco-marathon 2018 as the first team to complete a lap on the track fully autonomously.

Bibliography

[1] Shell. (2018). About shell eco-marathon, [Online]. Available: https://www.shell.com/energy-and-innovation/shell-ecomarathon/about.html (visited on 08/14/2018).

[2] L. Lassen. (2017). About dtu roadrunners, [Online]. Available: http://www.ecocar.mek.dtu.dk/english/about-dtu-roadrunners (visited on 08/14/2018).

[3] H. Si Høj, "Platform integration for autonomous self-driving vehicles", Master thesis, Technical University of Denmark, 2017.

[4] O. Lynggaard Topp, "Situation awareness for an autonomous vehicle", Master thesis, Technical University of Denmark, 2018.

[5] Wiki. (2014). Urdf, [Online]. Available: https://wiki.ros.org/urdf (visited on 08/14/2018).

[6] R. Hermansen, "Simulation of the dtu ecocar", Bachelor thesis, Technical University of Denmark, 2017.

[7] A. Ravn Jørgensen and B. Fuglsang Nielsen, "Dtudrive - a simulation tool for vehicle performance optimization and on-track strategic planning in fuel efficiency competitions", Bachelor thesis, Technical University of Denmark, 2012.

[8] J.-L. Blanco, "A tutorial on se (3) transformation parameterizations and on-manifold optimization", *University of Malaga, Tech. Rep*, vol. 3, 2010.

[9] G. Maciel. (2013). How to generate procedural racetracks, [Online]. Available: http://blog.meltinglogic.com/2013/12/how-to-generate-procedural-racetracks/ (visited on 08/15/2018).

[10] M. Blanke, M. Kinnaert, J. Lunze, M. Staroswiecki, and J Schröder, *Diagnosis and fault-tolerant control*. Springer, 2006, vol. 2.

[11] C. H. M. Johansen, "Development and implementation of an automatic brake system for the dtu road-runner", Synthesis project, Technical University of Denmark, 2018.

[12] G. F. Franklin, J. D. Powell, and M. L. Workman, *Digital control of dynamic systems*. Addison-wesley Menlo Park, CA, 1998, vol. 3.

Appendix A

Steering geometry derivations

The steering system is illustrated in Figure A.1 and the parameters are shown in Table A.1. Formulas for the steering system will now be derived.

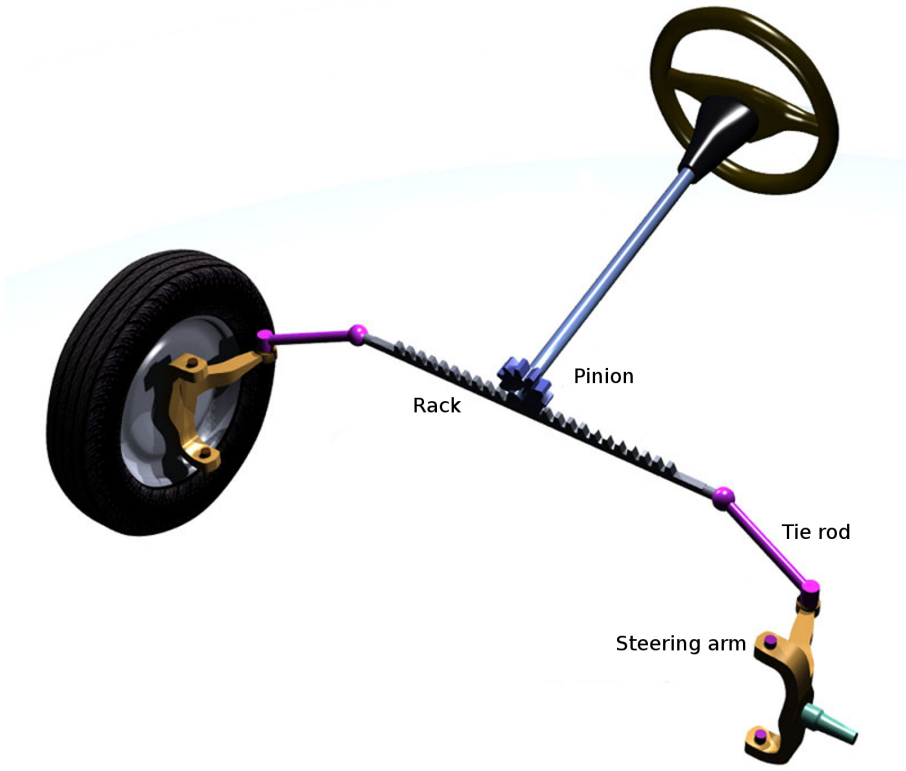


Figure A.1: Illustration of a rack-and-pinions steering system. Adapted from Wikimedia Commons (CC BY-SA).

Symbol	Description	Value [mm]
a	Tie rod length	150
b	Steering arm length	125
ν	Offset	92.65
W	Distance between wheel axles	921.78
L	Rack length	615

Table A.1: Steering system parameters.

For the derivation, Figure A.2 and A.3 are used.
When the rack is moved by turning the pinion, the distance d for each side is given by

$$d_l(x) = \frac{W-L}{2} + x\,,\; d_r(x) = \frac{W-L}{2} - x \tag{A.1}$$

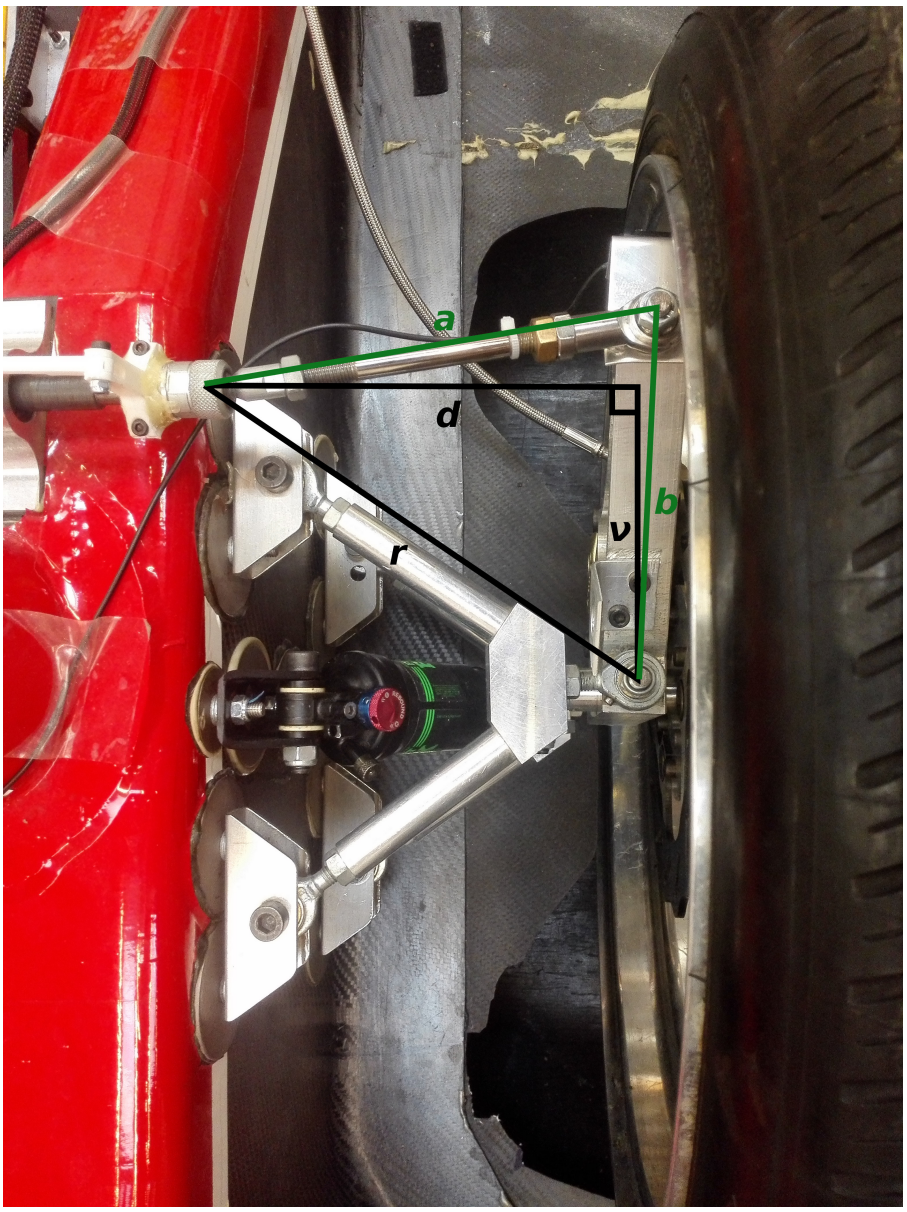


Figure A.2: Picture of the wheel mounting on the car with variables overlaid.

where x is the displacement of the rack, taken positive when it is moving towards the front right wheel. The distance r for each side is then given by

$$r_l(x) = \sqrt{d_l(x)^2 + \nu^2}\,,\; r_r(x) = \sqrt{d_r(x)^2 + \nu^2} \tag{A.2}$$

where ν is the offset between the rack and the wheel turning axle.
Looking at the right wheel in Figure A.3, the angle θ can be found by subtracting one angle from another:

$$\theta = \sin^{-1}\frac{d}{r} - \cos^{-1}\frac{b^2 + r^2 - a^2}{2br} \tag{A.3}$$

This is not the wheel angle, as the wheel might be at a constant angle to the steering arm. If we assume that the wheel has angle $\theta_r = 0$ when $x = 0$, we can find the right wheel angle for some x by

$$\theta_r = \sin^{-1}\frac{d_r}{r_r} - \cos^{-1}\frac{b^2 + r_r^2 - a^2}{2br_r} - \theta_0 \tag{A.4}$$

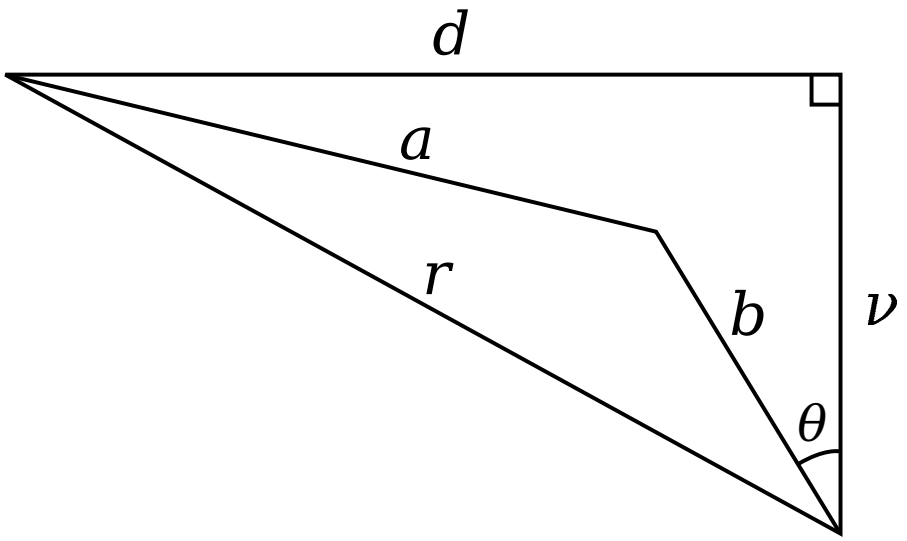


Figure A.3: Illustration from which the steering angles are derived from d, which is a function of the rack displacement x.

and similarly for the left wheel

$$\theta_l = -\sin^{-1}\frac{d_r}{r_r} + \cos^{-1}\frac{b^2 + r_r^2 - a^2}{2br_r} + \theta_0. \tag{A.5}$$

The constant θ_0 can be found by setting x equal to zero (assuming that the wheel angles are zero when $x = 0$):

$$\theta_0 = \sin^{-1}\left(\frac{W-L}{2\sqrt{(\frac{W-L}{2})^2 + \nu^2}}\right) - \cos^{-1}\left(\frac{b^2 + (\frac{W-L}{2})^2 + \nu^2 - a^2}{2b\sqrt{(\frac{W-L}{2})^2 + \nu^2}}\right). \tag{A.6}$$

It is also convenient to be able to calculate the required rack position for a given wheel angle. This can be calculated by inverting either of the two equations:

$$x = \frac{W-L}{2} - b\sin(\theta_r + \theta_0) - \sqrt{a^2 - b^2\cos^2(\theta_r + \theta_0) - \nu^2 + 2b\nu\cos(\theta_r + \theta_0)} \tag{A.7}$$

$$x = -\frac{W-L}{2} + b\sin(\theta_0 - \theta_l) + \sqrt{a^2 - b^2\cos^2(\theta_0 - \theta_l) - \nu^2 + 2b\nu\cos(\theta_0 - \theta_l)}. \tag{A.8}$$

A.1 Ideal Ackermann angles

For the vehicle to turn without the wheels slipping, all the wheels need to be perpendicular to the instantaneous centre of rotation. This is illustrated in Figure A.4, from which a condition can be derived.
From geometry, the angles of the left and right wheel can be found from the angle of the virtual front wheel β. The turning radius r can be found:

$$\tan\beta = \frac{L}{r} \Rightarrow r = \frac{L}{\tan\beta}. \tag{A.9}$$

And substituting that expression gives the wheel angles directly:

$$\tan\theta_l = \frac{L}{r - \frac{T}{2}}\ ,\ \tan\theta_r = \frac{L}{r + \frac{T}{2}} \tag{A.10}$$

$$\tan\theta_l = \frac{L}{\frac{L}{\tan\beta} - \frac{T}{2}} = \frac{L\tan\beta}{L - \frac{T}{2}\tan\beta} \tag{A.11}$$

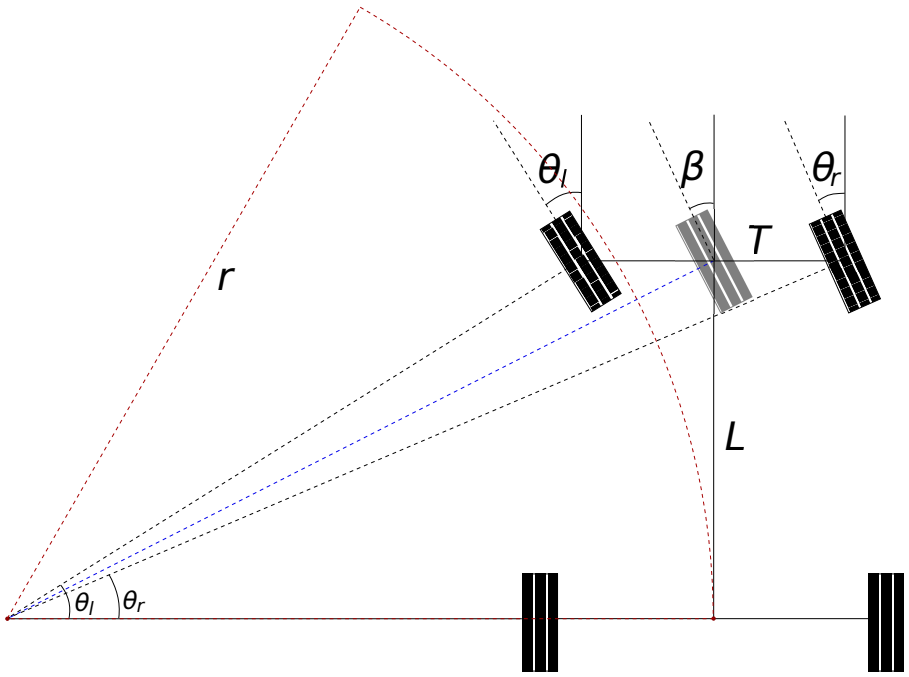


Figure A.4: Illustration of the wheel angle condition for no slip.

$$\tan\theta_r = \frac{L\tan\beta}{L + \frac{T}{2}\tan\beta} \tag{A.12}$$

where T is the track (distance between front wheels), L is the wheelbase (distance between front and rear axles), r is the radius of the circular arc and β is the angle of the virtual front wheel. To get the angles, `atan2` is used.