Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Engineering Cybernetics

Jon Eivind Stranden

Autonomous driving of a small-scale electric truck model with dynamic wireless charging

Master's thesis in Cybernetics and Robotics
Supervisor: Jon Are Suul
June 2019

NTNU
Norwegian University of Science and Technology

SINTEF

Jon Eivind Stranden

Autonomous driving of a small-scale electric truck model with dynamic wireless charging

Master's thesis in Cybernetics and Robotics
Supervisor: Jon Are Suul
June 2019

Norwegian University of Science and Technology
Faculty of Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Abstract

Automated wireless power transfer can be seen as an enabling technology for autonomous vehicles. At SINTEF Energy Research a 1/14 scale electric truck model [13] had been used for demonstrating such technology. The truck was fitted with an induction system, but was otherwise stock. In this master thesis a system for autonomous path following has been developed and successfully implemented to demonstrate self-driving capabilities on a path that includes an inductive charger. The purpose of the vehicle is to visualize the concept of wireless charging for a self-driving truck, but on a smaller scale.

Three different methods has been tested and implemented.

1. A LiDAR-based approach that utilizes Hector SLAM for mapping and localization together with a Pure Pursuit and Stanley Steering path tracking steering controller.
2. A machine-learning and camera-based approach that can navigate a track using deep learning and behavior cloning/supervised learning. This was inspired by the convolutional neural network used in Nvidia's DAVE-2 self-driving car.
3. A classic computer vision approach for detecting lane curvature and using PID for control.

ROS nodes that integrates the output from the SLAM- and deep learning-methods and combines them with a path recorder and a path tracking algorithm has been made, together with a graphical interface for monitoring vehicle states. Methods for estimating odometry without wheel encoders, dynamic speed control and obstacle detection with start/stop functionality has been implemented, along with a computer vision method for detecting ArUco markers. A complete, self-driving ROS-based system has been created. An Nvidia Jetson TX2 has been used as the main embedded computing unit. The system implements Hector SLAM as the primary SLAM method, since it does not require odometry. The Pure Pursuit path tracker was chosen as the preferred steering controller since it resulted in smooth and stable tracking of the path. The neural network of the machine-learning approach was able to steer the truck reliably and is implemented in the final version as a separate mode. This works independently of the LiDAR for when the LiDAR is out of range. The neural network has also been tested on the Udacity self-driving simulator [4]. The computer vision approach was found to be too demanding for the embedded hardware, and ended up not being used. The inductive charger system has successfully been connected and integrated to the Jetson TX2 through a CAN-bus interface in order to receive the battery state. The amount of charge received is dependent on the positioning of the truck, and is not optimized for a system with manual path creation. A better autonomous positioning system when driving across the charger is recommended, along with a better and more precise LiDAR and vehicle platform.

Preface

In this master thesis a self-driving system for a small-scale R/C semi-trailer with inductive charging has been created. This contributes to the development of a demonstration platform for an autonomous vehicle with inductive charging at SINTEF Energy Research. The system is based on ROS (Robot Operating System), OpenCV and Tensorflow/Keras running on an embedded Linux platform. The majority of the development is done in Python, with some minor use of C and C++. All hardware was provided by SINTEF by request from the student before the project was started. This included the 1/14 scale semi-trailer with an existing inductive charger system [13], an Nvidia Jetson TX2 embedded computer, a LiDAR, camera, IMU, microcontroller and a joystick controller. All hardware has been mounted and integrated into the truck's electrical system. Custom laser-cut mounting hardware has been made. ROS packages such as `hector_slam`, `rplidar`, `joy` and `rosserial` were used to achieve some of the hardware integration and localization. ROS nodes for path recording, path tracking, CNN steering, lane assist (computer vision) steering, encoder-free Ackermann odometry w/IMU, velocity estimation and car control has been made. The truck has been operated with PWM signals though a serial interface between the microcontroller and the Jetson TX2. The microcontroller is also responsible for publishing IMU data to ROS via I2C, and to control a WS2812B LED strip indicator. Integration of the existing onboard induction system was done using CAN-bus, with the help of Giuseppe Guidi at SINTEF Energy Research to configure the CAN-messages and the charging system. A circuit board was made in order to connect a CAN transceiver to the TX2, with an associated ROS node and GUI to publish and display the battery state respectively. The project takes inspiration from [35], [36] and [21].

I would like to thank my supervisor Associate Professor Jon Are Suul at the Department of Engineering Cybernetics, Norwegian University of Science and Technology (NTNU), and my co-supervisor Research Scientist Dr. Giuseppe Guidi at SINTEF Energy Research for supporting me in this project.

Table of Contents

List of Figures

List of Tables

Nomenclature

Symbols

α	Angle	[rad]
δ	Steering angle	[rad]
ψ	Yaw	[rad]
θ	Heading	[rad]
ω	Angular velocity	[rad/s]
κ	Curvature	[m^{-1}]

Indicies

x	Longitudinal direction
y	Lateral direction
v_x	Longitudinal velocity
v_y	Lateral velocity
t	Discrete time step
u	Controller input
L	Wheelbase length
R	Turning radius
g_x, g_y	Goal point
l_d	Look-ahead distance

Acronyms and Abbreviations

ROS	Robot Operating System
SLAM	Simultaneous Localization and Mapping
VSLAM	Visual Simultaneous Localization and Mapping
Pose	Position and Orientation
LiDAR	Light Detection And Ranging Sensor
IMU	Inertial Measurement Unit
PID	Proportional Integral Derivative Controller
R/C	Radio Control
ESC	Electronic Speed Controller
VCU	Vehicle Control Unit
SSH	Secure Shell
PWM	Pulse-width Modulation
CAN	Controller Area Network
DNN	Deep Neural Network
CNN	Convolutional Neural Network
GUI	Graphical User Interface

Chapter 1

Introduction

This thesis aims to demonstrate wireless inductive charging on a moving electric vehicle, though the development of a self-driving system on a small-scale vehicle. The project is in collaboration with SINTEF Energy Research. SINTEF is a norwegian, interdisciplinary research institute with leading international competence within the fields of technology and science. They are among the largest European research institutes and do research and development projects for businesses and institutions. This chapter gives an introduction to the project, the motivation behind and the problems this project will aim to solve.

1.1 Background & Motivation

Wireless charging has in the recent years become a common standard in consumer-oriented applications such as e.g. wireless phone chargers, but the technology has also been tested and implemented in more industrial applications such as cars, busses and electrical ferries [14]. Other uses include fixed charging spots for taxies [8] and wireless home charging pads for electrical cars. Wireless power transferring can also be seen as an enabling technology for autonomous vehicles [2] that can operate without the need for a physical charger connection. This can make for the creation of a self-supplied, fully autonomous and independent system. Imagine e.g. drones having an inductive platform for landing and charging. Or autonomous ferries with auto-docking and connection-less charging. The technology can have a huge impact on the future and can also change the transportation industry in general. Norway has set the ambition to become climate neutral by 2050 [7]. In order to achieve this goal, all modes of transport need to become as climate efficient as possible. Freight transport is decisive for business to thrive, but generates large emissions of greenhouse gases. Norwegian freight transport is expected to increase by 65% by 2050, in parallel with the target of climate neutrality. This is why it is important to work for the electrification of freight transport. Technology for dynamic on-road wireless power transfer to electric vehicles is envisioned as a suitable solution for power supply for autonomous vehicles. When it comes to vehicles, the autonomous capabilities are usually divided into five levels [17], where level 0 is a vehicle that is operated manually by a

human driver and level 5 is full self-driving without any human input. As the car manufacturers are approaching level 5 autonomy in self-driving cars, the benefits of having a fully self-sustained charging system becomes apparent. Not only the possibility for the car to drive itself to the nearest charger when needed, but also be able to charge autonomously on the move. Researchers at SINTEF has proposed an electric road infrastructure with built-in coils for wireless charging. By enabling fully automated power transfer to moving vehicles, such technology can help overcome the limitations of driving range for electric vehicles and allow for fully autonomous long-term operation.

1.2 Objectives

The objective of this project has been to propose and develop a strategy for self-driving technology that could be tested on a small-scale model of an electric truck with dynamic wireless charging. An electric truck model in scale 1/14 was already available at SINTEF Energy Research as a small-scale demonstration platform for inductive charging technology. The main task was to develop a self-driving strategy that could be implemented on this truck model to achieve a demonstration of a fully autonomous electric truck running on a path that included dynamic inductive charging.

1.3 Contribution

Three different methods for achieving a self-driving system on a small-scale truck has been implemented and tested.

- A SLAM-based method for localization and navigation.
- Using machine-learning with a deep convolutional neural network.
- Using classic computer vision methods.

Also a method for a detection system for the inductive charging area has been proposed

- Using computer vision to recognize the charging zone.

A working and easy-to-use system for following a path has been made. The project shows that the proposed strategies will work as methods for enabling autonomous driving along a path or track on a small-scale truck model. Each method comes with its own strengths and weaknesses depending on the area, but also due to the chosen hardware.

1.4 Thesis Overview

This section contains a short description of the chapters in this report:

1. **Introduction**
   An introduction to the project and the motivation and objectives behind it.

2. **Hardware and Software**
   Describes the hardware components and the software stack. Overview of the system hardware architecture.

3. **SLAM-based Path Planning**
   Contains the specific methods that were used to develop and implement a SLAM-based path tracking system for autonomous driving using LiDAR. Results of different tuning parameters are presented.

4. **Deep Learning Steering Controller**
   A steering controller based on deep learning with a convolutional neural network is presented, together with training procedures and dataset creation. The controller uses a camera to follow a path.

5. **Computer Vision Steering Controller**
   Describes a steering controller based on classic computer vision methods, with lane detection and PID-control.

6. **Integration with Wireless Inductive Charging**
   Describes how the wireless charging system is connected to the truck, and how information from the system is presented. A detection system for the charger area based on computer vision/ArUco is tested.

7. **Discussion**
   Discussions of the results from the previous chapters, along with suggested improvements.

8. **Conclusion and Outlook**
   The conclusion of the project and the way forward.

9. **Appendix**
   Information about how to operate the truck, with procedures and installations, and an overview of the ROS system structure and code.

Chapter 2

Hardware and Software

The basis for this project was an 1/14 scale semi-trailer provided by SINTEF. The truck had earlier been used for technology demonstrations of inductive charging [13], but with manual control from an R/C radio-transmitter only. The task at hand was to develop a self-driving system to control the truck autonomously while driving on a path that included the inductive charger. The truck had already been fitted with the power electronics for the earlier demonstrations to provide wireless power to the battery, but was otherwise stock. This chapter describes the hardware components and software stack used for building and developing the self-driving truck, shown in figure 2.1. The hardware and software components need to be able to provide environmental feedback through perception, process the information and finally be able to operate the truck through control of the hardware. At the end of this chapter an overview of the system structure based on the methods from chapter 1 is presented.

Figure 2.1: Tamiya R/C Scania self-driving semi-trailer.

2.1 Hardware

2.1.1 Component Overview

Here, a list of the parts and components used for this project is presented. All parts were ordered before the project was started, but also some custom parts were made such as laser-cut mounting hardware for the Nvidia Jetson TX2 and the RPLIDAR, power connector adapters and a CAN-bus interface. See the following subsections for descriptions of the main components.

Electronics	**Amount**	**Type**
Nvidia Jetson TX2 Developer kit with Jetpack 3.3	1	Embedded SoM
Slamtec RPLIDAR A2M8	1	LiDAR
PJRC Teensy 3.2	1	Microcontroller
Logitech C922 Pro	1	Webcam
Logitech F710	1	Joystick
Sparkfun BNO080	1	IMU
Texas Instruments SN65HVD230D	1	CAN transceiver
WS2812B	4	Neopixel RGB LED
120 ohm resistor	2	For CAN-bus
SINTEF Induction system w/CAN-bus	1	Wireless charger
Mechanical		
Tamiya R/C Scania Semi-Trailer 1/14 scale	1	R/C car
Laser-cut mounting	1	For Nvidia Jetson TX2
Laser-cut mounting	1	For RPLidar A2M8
XT60 to barrel-plug adapter	1	Custom connector
Miscellaneous		
A laptop with Ubuntu 16.04 LTS and ROS Kinetic	1	Computer
Turnigy 5200mAh 3S Li-Po	2	Battery
Turnigy 6500mAh 2S Li-Po	1	Battery
Biltema 12V LiFePO4	1	Battery
Belkin 4-port USB 3.0 hub	1	USB hub
SanDisk 128 GB SDcard	1	Memory card

Table 2.1: List of components.

2.1.2 Nvidia Jetson TX2

The Nvidia Jetson TX2 devkit (figure 2.2) was selected as the main computing unit for this project, due to its relative high performance pr dollar. The Jetson TX2 is a high performance embedded system-on-module (SoM) with six CPUs cores (Dual-core Denver 64-bit + quad-core ARM Cortex A57). It has a GPU based on the Nvidia Pascal architecture with

256 CUDA cores that is useful for deploying computer vision and machine learning. Jetson TX2 runs an ARM-version of Linux/Ubuntu and the typical power consumption is about 7.5W (15W max). The low power consumption is useful in battery powered applications such as this, as opposed to using a small desktop computer or a laptop.

Nvidia Jetson TX2	I/O	Power
256-core Nvidia Pascal Architecture GPU	HDMI 2.0	5.5-19.6V DC
2 Denver 64-bit CPUs + Quad-Core A57 Complex	USB 3.0 Type A	
8 GB L128 bit DDR4 Memory	USB 2.0 Micro	
32 GB eMMC 5.1 Flash Storage	SATA, SDcard	
802.11a/b/g/n/ac 22 867Mbps WiFi	Dual CAN bus	
10/100/1000 BASE-T Ethernet	UART, SPI, I2C, GPIO	

Table 2.2: Technical specifications Nvidia Jetson TX2.

Figure 2.2: Nvidia Jetson TX2 devkit. Ill.: Nvidia

2.1.3 Slamtec RPLIDAR A2M8

A LiDAR is a laser-based range measurement sensor that measure distances by utilizing time-of-flight of an emitted light-pulse and the reflected light. It has become the standard for self-driving vehicles where mapping of the environment is important. Obstacle avoidance is another use-case for this kind of sensor. For this project a low-cost 2D LiDAR was selected, as the vehicle will only move in the plane and because good 2D and 3D LiDARs can be quite expensive. A 2D LiDAR only measure distances in horizontal direction. The chosen LiDAR was the Slamtec RPLIDAR A2M8 360° with a distance range of 0.15 - 12m, a scan rate of 5-15 Hz and an angular resolution of 0.45° - 1.35° depending on the rotation speed of the unit. It measures distances in a 360° field of view. The RPLIDAR

A8M8 also has the ability to measure reflection intensity for distinguishing between different materials. A negative aspect of choosing a low-cost LiDAR such as the A8M8 is the slow scan rate that can affect the maximum turning rate of the vehicle it is mounted on. If the vehicle turns too quickly the LiDAR will not be able to keep up. A scan rate of 30-40 Hz would be ideal, such as on the Hokuyo UST-10LX which is another 'budget' laserscanner. The maximum measured distance may also cause some restrictions in larger areas, but this depends on the application and the A8M8 should work well in smaller areas. The LiDAR was mounted on top of the roof of the R/C truck with custom hardware (figure 2.3), in order to clear the trailer and get a 360° view of its surroundings while driving.

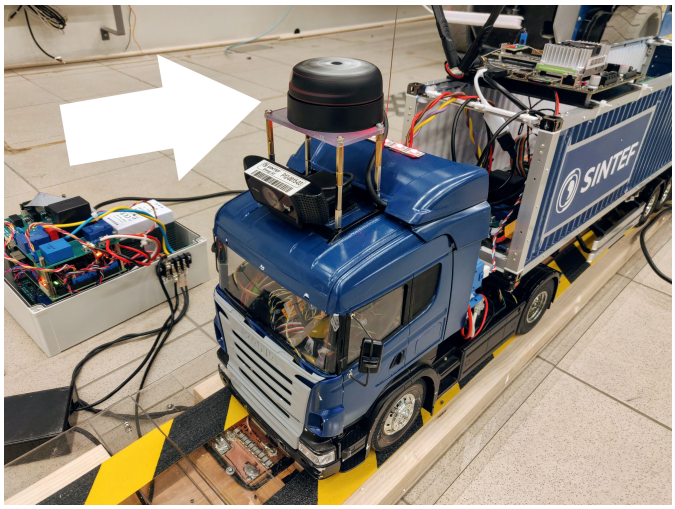

Figure 2.3: The Slamtec RPLIDAR A2M8 mounted on top of the 1/14 truck.

2.1.4 Logitech C922 Pro

The camera is another commonly used sensor in self-driving vehicles. It can be used for object recognition and distance measurements, or as input to a deep neural network. In contrast to the LiDAR, cameras are good at capturing color data and textures, and can more easily distinguish objects. Cameras are more sensitive to environmental changes such as varying light conditions than the LiDAR. For fast moving applications a camera with a global shutter is preferred. This implies that the image is updated as a whole, in contrast to the more regular rolling-shutter cameras that use scan-lines. A rolling-shutter camera will update the image line-by-line which may cause 'tearing' of the image if the camera is moved too quickly. A camera with a global shutter will usually cost more than a camera with a rolling shutter. For this project a rolling-shutter camera was chosen due to the budget and the slow-moving application. The chosen camera was the Logitech C922 Pro webcam. It has a maximum resolution of 1920 x 1080 pixels, a refresh rate of up to 60Hz and a field of view of 78°. It is powered by a regular USB-cable. A stereo-camera could also be considered in the future as it would make visual odometry easy to implement. Visual odometry is possible with monocular cameras, but they need a reference or scale-

factor to be able to estimate the correct distance. A stereo-camera could also be used as a redundant distance measurement sensor to complement the LiDAR scanner. The C922 camera was mounted on the roof of the truck beneath the LiDAR such that it had a clear view of the road in front of the vehicle. This can also be seen in figure 2.3.

2.1.5 Bosch BNO080 IMU

An inertial measurement unit (IMU) is used to measure orientation, linear velocity and angular rate in 3D-space. It uses a combination of accelerometers, gyroscopes and magnetometers. For this project the Bosch BNO080 IMU was chosen. It has 9 degrees of freedom (Acc, Gyro, Mag) and a dedicated ARM Cortex M0+ processing unit for drift correction. This leads to accurate rotation vector headings, with a static rotation error of 2° or less, and a linear acceleration accuracy of 0.35 m/s^2. The BNO080 can be interfaced with I2C (400kHz), SPI (3MHz) or UART (3Mbps) and has an operating voltage of 1.65V - 3.6V DC.

2.1.6 PJRC Teensy 3.2

In order to connect and interface the electronic components such as the servos, ESC, IMU and lights, a microcontroller was needed. The PJRC Teensy 3.2 is a small, powerful microcontroller with support for the well-know Arduino platform. It has a 32-bit ARM Cortex-M4 running at 72 MHz, and a logic level of 3.3V. The Arduino support means it is ROS-compatible and can be used to connect to a ROS network for direct control and interaction via a serial interface, by using the `ros_serial` package. There is also support for I2C, SPI, UART and CAN.

2.1.7 SN65HVD230D/VP230 CAN transceiver

The Nvidia Jetson TX2 features two build-in CAN controllers, *can0* and *can1*. To make use of CAN communication, a CAN transceiver has to be added to the controller. The CAN transceiver is the interface between the CAN protocol controller and the physical bus. The CAN transceiver used for this project was the Texas Instruments SN65HVD230D (marked VP230).

2.2 Hardware System overview

This section presents an overview of the hardware system architecture, see figure 2.4. The LiDAR, camera and microcontroller are connected to the Nvidia Jetson TX2 via USB for both communication and power. The joystick has a small receiver that is also plugged in via USB, but uses wireless 2.4 GHz RF communication. Most USB peripherals except the power connector for the LiDAR, are connected via the USB hub. The Jetson is powered by a separate battery and can accept voltages from 5.5-19.6V DC. The battery used to power the Jetson is a 3-cell 5200mAh Li-Po (11.1V). The Teensy microcontroller is used to control and interface with the car controls (servos and ESC), the IMU and the LED indicator strip. A laptop or computer can be connected to the Jetson via WiFi/SSH for startup and monitoring. See the appendix on how to connect.

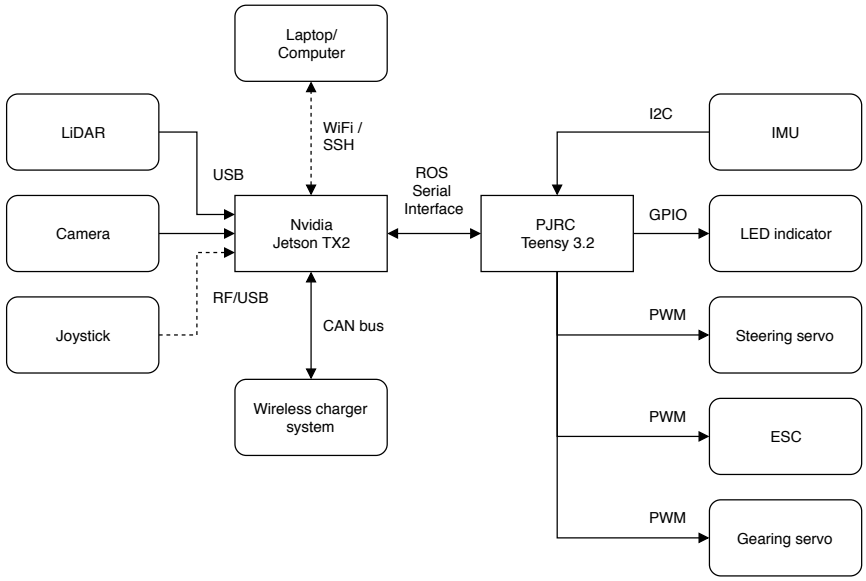

Figure 2.4: Block diagram of the hardware system architecture.

2.2.1 Vehicle Control

The truck is controlled by pulse-width modulated signals (PWM) going from a radio receiver to a vehicle control unit (VCU) that controls the steering servo, gearing servo and the electronic speed controller (ESC). The Teensy 3.2 microcontroller replaces the stock radio receiver in order to control the servos and ESC. Since the microcontroller only has 3.3V output, 5V is supplied from the VCU with a common ground for both the microcontroller and the servos/ESC, as shown in figure 2.5. The microcontroller is connected to the Jetson with a USB interface for serial communication. It has been programmed to receive input data and translate them into PWM control signals.

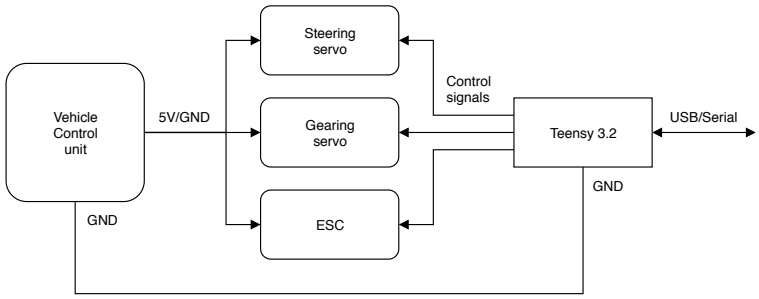

Figure 2.5: Block diagram of the vehicle control configuration.

2.2.2 Wireless Charging System

A system for wireless inductive charging has been developed by SINTEF Energy Research as a demonstration unit. This can be seen in figure 2.6. The system consists of power electronics that controls the current flow, and two induction coils mounted in 3D-printed housings (figure 2.7). Plexiglas and grip-tape has been added to the top to make the surface more drivable. The truck has had an induction coil mounted beneath the drive train in order to receive charge, seen in figure 2.8, with onboard control and power electronics mounted in the trailer and instruments for measuring real-time current and voltage. The control electronics on the truck are powered by a separate 12V LiFePO4 battery mounted in the trailer. The truck itself is powered by a 2-cell 6500mAh Li-Po battery that receives charge when driving over the coils. The power electronics can be interfaced via a CAN-bus interface.

Figure 2.6: The inductive charger setup.

Figure 2.7: A closer look at one of the induction coils.

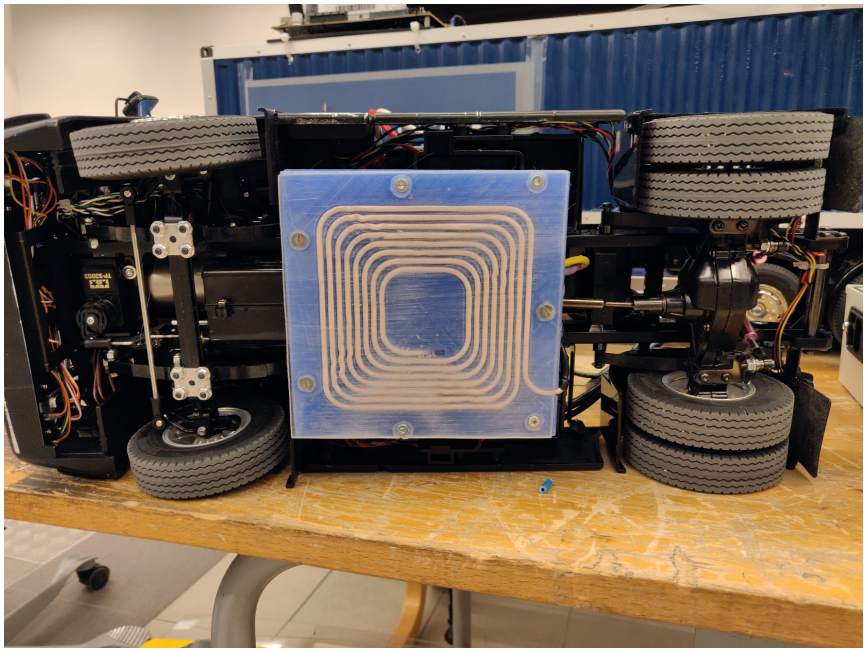

Figure 2.8: Induction coil mounted beneath the truck.

2.3 Software

To develop a self-driving vehicle, different software frameworks are required. These frameworks should be able to run on the ARM-based embedded hardware of the car, as well as on the host computer, and make development and interfacing of different sensors easier. It should also help with testing and visualizing during the development process. This section presents the frameworks and software used for the self-driving system.

2.3.1 ROS (Robot Operating System)

The Robot Operating System, or ROS [24], is an open-source meta-operating system that is developed specifically for robotics. It runs on top of any existing Linux-distribution, such as Ubuntu. ROS implements hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management. It functions as a peer-to-peer network of processes that can communicate via message passing. ROS comes with tools and libraries to visualize data and to run code on multiple platforms, including embedded computers and microcontrollers. For this project, ROS Kinetic was used. Core elements of the ROS system are listed below:

- `Packages` contains the ROS run-time processes (nodes), libraries and configuration files that is usefully organized together.
- `Nodes` are ROS run-time processes that perform computation. They can publish and/or subscribe to topics for inter-node communication. A full ROS system is usually build up by multiple nodes doing specific tasks.
- `Master` is responsible for name registration and lookup in the ROS network. It can run across multiple machines for communication via Ethernet or WiFi, and does so that the nodes can find each other and send messages.
- `Messages` are used for standardized communication between nodes. There are several standard message formats, such as int, float, sensors, navigation etc. It is also possible to make custom message formats.
- `Topics` are the channels that are used for message passing between the nodes. The message passing is based on a publish/subscriber scheme. A node can *publish* a message on a given topic. The topic has a name identifier that makes it possible for other nodes to identify and *subscribe* to it. The publisher is not aware of who is subscribing to the topic. Multiple nodes can publish or subscribe to the same topic.
- `Rosbags` are used for saving and playing back ROS message data from topics.
- `Rviz` is a tool for visualizing ROS messages and transformations, and can be used to visualize laser scans, maps, paths and current positions and orientations.
- `rqt` is a tool for plotting data and display connections.
- `map_server` is a tool in the ROS navigation package for hosting and serving map data to nodes.

2.3.2 OpenCV

OpenCV is an open-source computer vision and machine learning software library. It has C++, Python, Java and MATLAB interfaces and supports Windows, Linux, Android and Mac OS. It can also be compiled with support for CUDA, that can make use of the CUDA-cores on the Nvidia Jetson TX2 to achieve GPU-accelerated functionality. CUDA is currently only supported in C++. For this project, OpenCV 3.4.0 was used on both the host computer and the Nvidia Jetson TX2. The Jetson had an ARM-version of OpenCV 3.4.0 with CUDA enabled.

2.3.3 Camera Calibration Software

To compensate for lens distortion a camera calibration is needed. This will also obtain the intrinsic parameters [10] of the camera. The intrinsic parameters are specific to the camera, such as the focal length (fx, fy) and optical centers (cx, cy). They can be represented in a 3×3 matrix known as the camera matrix, see equation 2.1. The distortion matrix seen in equation 2.2, is a 5×1 matrix of the distortion coefficients. The camera model used is the pinhole model [10].

$$Camera\ matrix = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

$$Distortion\ coefficients = \begin{bmatrix} k_1 & k_2 & p_1 & p_2 & p_3 \end{bmatrix} \tag{2.2}$$

There are several tools available for doing camera calibration, e.g. the `camera_calibration` tool in ROS shown in figure 2.9, or the built-in functions in OpenCV. Common for these tools is the use of a checkerboard pattern with squares of known size. The calibrated parameters for the Logitech C922 Pro is shown in equations 2.3 and 2.4. The numbers are rounded to the third decimal. The camera was calibrated with a resolution of 640x480 and the calibration will not be valid if the resolution is changed.

$$Camera\ matrix_{\ C922} = \begin{bmatrix} 708.434 & 0 & 317.966 \\ 0 & 724.304 & 274.086 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

$$Distortion\ coefficients_{\ C922} = \begin{bmatrix} 0.097 & -0.184 & 0.014 & 0.008 & 0 \end{bmatrix} \tag{2.4}$$

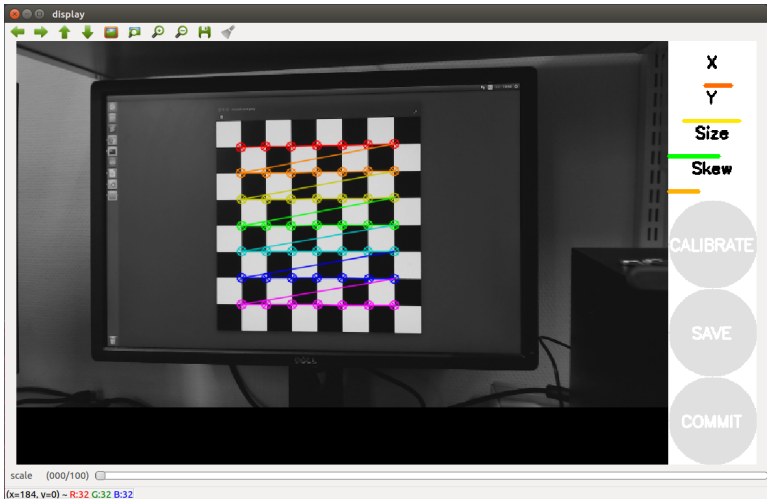

Figure 2.9: Camera calibration process with camera_calibration in ROS.

2.3.4 Deep Learning Frameworks

TensorFlow

TensorFlow is an open-source machine-learning framework from Google that can be used to build and design neural networks. The GPU-version of TensorFlow can also take advantage of the Pascal GPU on the Nvidia Jetson TX2, utilizing its CUDA-cores.

Keras

Keras is another open-source machine-learning framework that runs on top of the TensorFlow framework, as a more user-friendly interface. Keras makes it easier to interact with the TensorFlow framework through high-level abstractions for creating deep neural networks.

Chapter 3

SLAM-based Path Tracking

This chapter describes the methods of building a path-following autonomous car based on simultaneous localization and mapping (SLAM) using a LiDAR sensor. It will go trough some different methods for SLAM, odometry, path tracking and dynamic speed control. In the end of this chapter is a description of the results when testing with different methods and tuning parameters.

3.1 Related Work

In this section some of the work that inspired this project's approach to achieving self-driving using SLAM and path tracking is presented.

3.1.1 Stanley

Stanley [35], created by Stanford University's Stanford Racing Team, was the car that won the 2005 DARPA Grand Challenge. The DARPA Grand Challenge was an autonomous car racing competition across the Mojave desert that lasted 240 km. Stanley (figure 3.1) was based on a VW Touareg. The onboard autonomous system used five Sick AG LiDAR units for 3D-mapping, a GPS system for global map localization together with IMUs for pose estimation, wheel sensors for odometry and a video camera for detecting road conditions. It had six 1.6 GHz Intel Pentium M based computers running different versions of Linux. It was operated by a combination of electric motors, hydraulics and the built-in drive-by-wire system from VW. Stanley used supervised machine-learning based on human driver input to be able to drive. This method is explored further in chapter 4. The Stanford Racing Team implemented a steering controller called the *Stanley Steering Controller*. The theory behind this controller is explained in section 3.6.3. The Stanford Racing Team was led by Associate Professor Sebastian Thrun, who later on founded the Google self-driving car company Waymo and the online learning site Udacity.

Figure 3.1: Stanley - 2005 DARPA Grand Challenge. Ill.: Stanford University.

3.1.2 MIT RACECAR

The MIT RACECAR [29] (Rapid Autonomous Complex-Environment Competing Ackermann-steering Robot) is an open-source hardware platform for robotics research and education from the Massachusetts Institute of Technology. An example of a car built on the platform can be seen in figure 3.2. The platform is part of a course to teach students about perception and planning algorithms for cars that can quickly navigate through complex environments. But there is also a competition, where student teams will race through the corridors and tunnels of MIT. The platform is based on the Nvidia Jetson TX1, the predecessor to the Jetson TX2. It is equipped with an IMU, a visual odometer, a laser scanner (Hokuyo UST-10LX), and a stereo-camera (Stereolabs ZED). As for software, it is based on the ROS software stack. Software resources for the platform can be found on the projects github-page [1]. This includes a fast particle filter localization method [5], see section 3.3, and a method for estimating odometry without wheel encoders, see section 3.4.2.

Figure 3.2: MIT RACECAR / RACECAR/J. Ill.: JetsonHacks

3.1.3 F1/10

F1/10 [22] is an international autonomous racing competition for students. The competition involves designing, building, and testing an autonomous 1/10th scale F1 race car capable of speeds in excess of 60 km/h. All while learning about perception, planning, and control for autonomous navigation. The race itself is a corridor race similar to the MIT RACECAR, and F1/10 takes a lot of inspiration from this, e.g. the sensor package. The homepage contains resources about hardware and algorithms for localization and navigation.

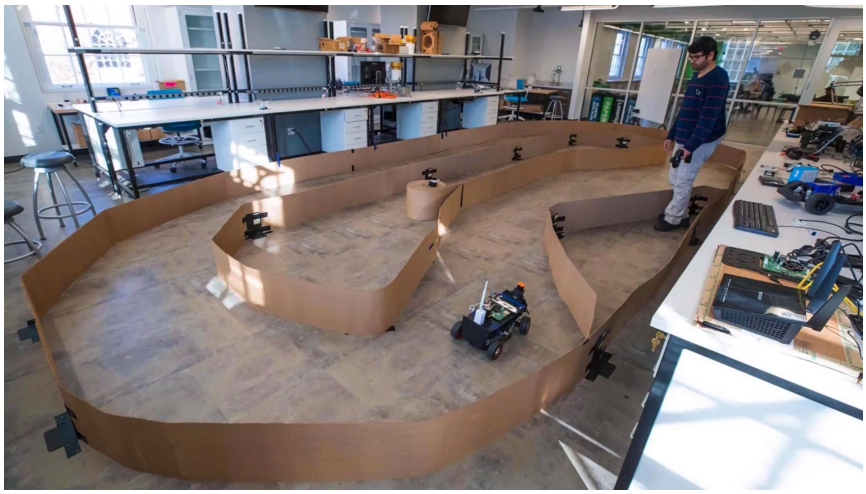

Figure 3.3: F1/10 experimental track setup. Ill.: F1/10

3.2 Simultaneous Localization and Mapping (SLAM)

Simultaneous Localization and Mapping (SLAM) is a technique that provides both localization and mapping so that an agent/robot is able to navigate in an unknown environment. The map is built and updated while simultaneously keeping track of the agent's position within it. SLAM is known as a chicken-or-egg problem because in order to localize, a map is needed. Vice versa a pose estimate (localization) is needed for mapping. Both of these factors are unknown. The SLAM problem [15] is defined as a set of given and wanted values over discrete time steps t.

Given values:
Robot controls:

$$u_{1:t} = \{u_1, u_2, u_3, ..., u_t\} \tag{3.1}$$

Relative observations:

$$z_{1:t} = \{z_1, z_2, z_3, ..., z_t\} \tag{3.2}$$

Wanted values:
Environment map:

$$m = \{m_1, m_2, m_3, ..., m_n\} \tag{3.3}$$

Robot's path:

$$x_{1:t} = x_1, x_2, x_3, ..., x_t \tag{3.4}$$

An estimate of the robot's path and the map is given by:

$$p(x_{0:t}, m|z_{1:t}, u_{1:t}) \tag{3.5}$$

Common methods for solving the SLAM problem are particle filters, extended Kalman filters, and GraphSLAM. Most methods rely on a combination of LiDAR distance readings, odometry, and/or camera images (VSLAM). When using a planar LiDAR such as the RPLIDAR A8M8, maps are represented in ROS as a 2D occupancy grid map with x,y-coordinates, as seen in figure 3.4. Maps can also be represented as a 3D point cloud when using a 3D LiDAR or VSLAM. In the following subsections three different methods for SLAM with ROS support are presented; Hector SLAM and Gmapping which are LiDAR-based options, and ORB-SLAM2 which is a camera-based approach.

3.2.1 Hector SLAM

Hector SLAM [20] developed by Team Hector from the Technische Universitt Darmstadt in Germany, is an odometry-free SLAM approach that can generate maps and estimate 2D pose based off LiDAR distance readings. Many LiDAR-based SLAM methods require additional odometry in some form to estimate an accurate pose of a vehicle. This could be a problem in applications where there is no room for odometry sensors such as wheel encoders, which is the case for the truck model used in this project. Visual odometry is a possible option but this has a high load on the processor and would ideally require a stereo camera setup. Hector SLAM is a robust SLAM that is built with high scan rate

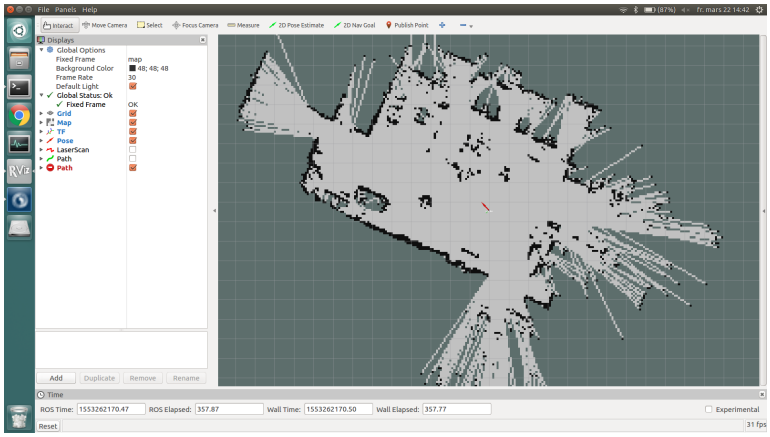

Figure 3.4: The NTNU/SINTEF National Smart Grid laboratory mapped with Hector SLAM.

LiDARs like the Hokuyo UTM-30LX (40Hz) in mind. It will also work with lower scan rate LiDARs like the RPLIDAR A8M8 (5-15Hz), but this is less robust to rapid angular motion. Hector SLAM does not provide loop closure, but is accurate enough to not need explicit loop closure in many real world environments. The system has been successfully implemented on Unmanned Ground Vehicles (UGV) where as in harsh terrain the vehicle cannot rely on wheel odometry. It has also been used in unmanned surface vehicles (USV), unmanned aerial vehicles (UAV), and handheld mapping devices. Figure 3.5 shows a Hokuyo UTM-30LX LiDAR mounted on top of a small embedded computer running Linux. Hector SLAM comes with a ROS API for publishing maps and odometry/pose when given a LiDAR scan input. The published pose representes the current transformation between the map and the vehicle.

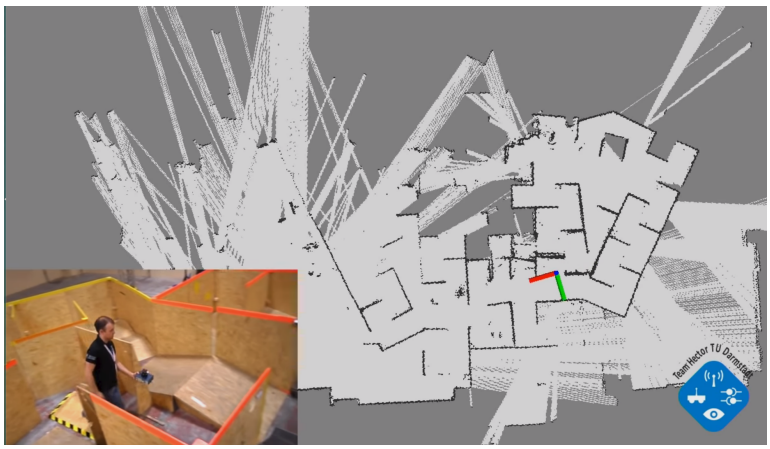

Figure 3.5: Hector SLAM on a handheld mapping device. Ill.: Team Hector Darmstadt/Youtube.

Hector Mapping

The mapping and localization in the Hector SLAM package is performed by the Hector Mapping ROS node. Below is a basic overview of how the mapping and localization process in Hector Mapping works (one iteration).

1. Receive a scan from the LiDAR
2. Transform the scan endpoints into the `base_link` transformation frame. This is the frame used for localization and for transformation of laser scan data.
3. Throw out the endpoints outside of the cut-off parameters `laser_min_dist` and `laser_max_dist`, to limit the range.
4. Perform a 2D pose estimation.
5. Update the map if the robot is estimated to have travelled more than the thresholds indicated by the `map_update_distance_thresh` and `map_update_angle_thresh` parameters.

Hector Mapping Tuning Parameters

The parameters used for the map update settings in Hector Mapping can be seen in table 3.1 and has been found to be a good balance between update frequency and stability for the Slamtec RPLIDAR A8M8 LiDAR. The scan rate was set to 15Hz by increasing the motor speed on the A8M8 from 600 rpm (10Hz) to 900 rpm with the help of a customized ROS package of the original driver software. This lowers the angular resolution but also makes it more robust to higher angular rates. The laser range was reduced to 12 m, which is the maximum usable range of the A8M8.

Parameter	Description	Used value	Default
map_update_distance_thresh	map update distance threshold [m]	1.5	0.4
map_update_angle_thresh	map update angle threshold [rad]	1.6	0.9
laser_min_dist	laser minimum distance [m]	0.3	0.4
laser_max_dist	laser maximum distance [m]	12.0	30.0

Table 3.1: Tuning parameters Hector SLAM.

Hector Trajectory Server

Hector SLAM includes a tool for logging trajectories. This is the Hector Trajectory Server. The Hector Trajectory Server can be used to log trajectory data based on the transform/pose estimation on the vehicle. The trajectory is published as a ROS Path message and can be visualized in Rviz for tracking performance and compare results.

3.2.2 Gmapping

Gmapping [11] is one of the most common SLAM methods in ROS and is part of the ROS navigation stack. It is based on a highly efficient Rao-Blackwellized particle filter to generate grid maps from laser range data. Rao-Blackwellized particle filters have been introduced as effective means to solve the simultaneous localization and mapping (SLAM) problem. Gmapping differs from Hector SLAM in that it requires odometry data in addition to the laser scan data. Odometry can be emulated and estimated using laser scan data, cameras, IMUs or motor output but this can lead to less accurate performance in comparison to using wheel encoders, and may also require extra hardware resources. Some methods for odometry is further investigated in section 3.4.

3.2.3 ORB-SLAM2

As an alternative to laser-based SLAM, it is also possible to use a monocular camera like the Logitech C922 Pro, for mapping and localizing. A library called ORB-SLAM2 [25] has been tested to achieve visual SLAM. ORB-SLAM2 is a real-time SLAM library for Monocular, Stereo and RGB-D cameras that computes the camera trajectory and a sparse 3D reconstruction, as seen in figure 3.6. It also features loop-detection and support for ROS. ORB-SLAM2 was tested as a redundant system or replacement to the LiDAR setup for this project and seemed to be a robust solution but also computationally intensive on the Jetson TX2. A scaling factor is required with a monocular setup, whereas stereo and depth cameras would have true scale. A stereo-camera setup, like for example Stereolab's ZED camera or the Intel Realsense would have been preferred in order to estimate the correct scaling. Still, ORB-SLAM2 showed impressive results even with only a monocular setup, and could be a topic for future experimentation and development.

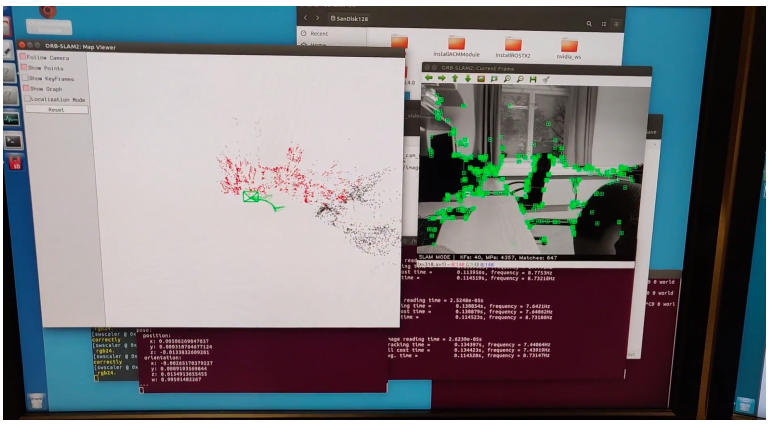

Figure 3.6: ORB-SLAM2 VSLAM test in the authors office.

3.3 Particle Filter Localization

SLAM is a computationally intensive process, and if the vehicle is operating in the same room or environment all the time, then it shouldn't always be necessary to generate a new map each time. If it is assumed that the world will not change after the mapping process is complete, then a static map can be used to determine the location and orientation. A static map can be obtained by mapping the environment with SLAM, then save the generated map to file with the `map_server` package from the ROS navigation stack. The ROS navigation stack is a standard lib for navigation with ROS. The map server can load the saved map and distribute it to the ROS network. In order to localize in the static map, a particle filter like the Adaptive Monte Carlo Localization (AMCL) [9] or the MIT RACECAR Particle Filter [5] can be used. AMCL is the standard particle filter implemented in the ROS navigation stack, but the MIT RACECAR particle filter has the advantage that it is much faster than AMCL (30Hz vs. 4Hz respectively). It also has support for GPU-acceleration (RMGPU), meaning that it can utilize several times more particles hence increasing tracking accuracy. The particle filters can be used to obtain both positioning and orientation. The MIT RACECAR particle filter was tested, but was not seen as accurate enough compared to using SLAM directly, often with noisy outputs due to the limitations of the LiDAR. The car would also be operated in different unknown areas every time, so the need for static maps would be less although the particle filters are using less resources.

3.4 Odometry

Odometry is the use of data from sensors to estimate change in position over time. In ROS, odometry is represented as a series of time-stamped poses and twists. A pose represent the position and orientation of the agent relative to the map, while a twist represent the linear and angular velocity vectors in space. To improve localization it is common to use wheel rotation encoders to measure distance and difference of wheel travel. Since the truck did not have room for encoders that could monitor wheel speed, alternative methods for estimating odometry was investigated. The vehicle is equipped with an accelerometer as part of the IMU, but pure accelerometer-based odometry was seen as too inaccurate since it would have implied double integration of the acceleration to get the distance, and estimation errors would quickly accumulate. The methods investigated for encoder-free odometry were based on LiDAR and ESC/IMU. Visual odometry was briefly tested with the `rovio` [3] ROS package, but not further used.

3.4.1 Laser-based Odometry

The Hector SLAM mapping package described section 3.2.1, can emulate odometry by using its pose estimator and the `laser_scan_matcher` package. This can be very accurate when there are enough features for the laser scanner to detect, but in areas where there is a lack of features, such as long straight hallways or large open areas it can struggle a bit, especially when using a low-cost LiDAR with limited range, update rate and resolution. The Hector odometry function can be used for emulating odometry where this is required. It can be a bit more noisy and less accurate than real wheel encoders in certain

areas as described earlier, but avoids problems like wheel spin and has proven to work well in smaller environments where the tracking has shown to be very good.

3.4.2 ESC/IMU-based Odometry

A different approach to achieve odometry without wheel encoders is to use the controller input to the motor/ESC on the truck model directly. In difference to using laser-based odometry it does not rely on surrounding feature detection, and would ideally be a more robust solution. As mentioned in the beginning of this section, a ROS odometry message consists of a series of poses and twists. The pose of the vehicle can be found by combining the current velocity and the yaw angle. The current velocity can be estimated from the relationship between the controller input u and the longitudinal velocity output v_x. To find this relationship, the pose output from the Hector SLAM package was used to get the current position in x- and y-direction in order to build a model. A longitudinal velocity reference based on the laser measurements v_{lidar} was then found by taking the difference in position over time.

$$v_{lidar} = \frac{\sqrt{(x_n - x_{n-1})^2 + (y_n - y_{n-1})^2}}{dt} \tag{3.6}$$

where x_n and y_n represent the current position of the vehicle in x- and y-direction relative to the map.

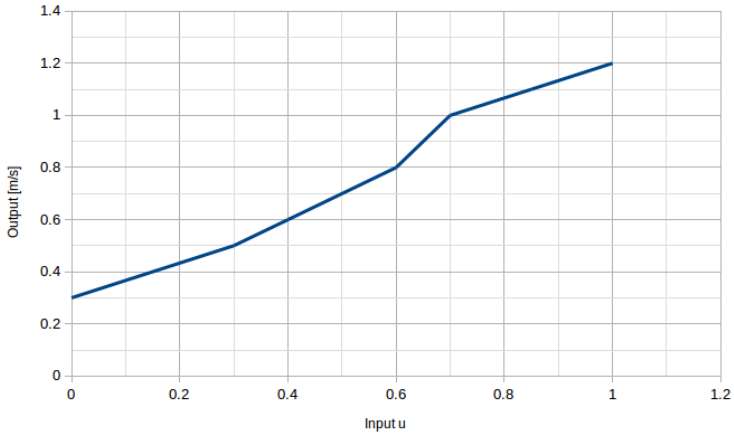

Figure 3.7: Controller input vs. longitudinal velocity output [m/s].

The velocity output of the truck model was then measured with different constant controller inputs on a flat surface. As shown in figure 3.7 the longitudinal velocity of the vehicle is almost linear to the controller input if one take into account the measurement noise from the laser distance readings. Here, input ranged from 0-1 where 0 was no input

and 1 was max input. By assuming a linear relationship, the longitudinal velocity can now be expressed as a linear velocity v_l with input u [0,1]

$$v_l = 0.9u + 0.3 \tag{3.7}$$

The truck model has an (anti-) Ackermann-based geometry and a single motor output to the wheels, therefor it is not possible to estimate yaw directly without the use of wheel encoders to measure rotation difference, as would be the case for a differential drive vehicle. But it is possible to use the angular velocity output ω of the IMU. Now, yaw ψ can be expressed as the integral of the angular velocity in z-direction ω_z

$$\psi = \int_0^t \omega_z dt \tag{3.8}$$

Then, vehicle velocities in x- and y-direction, $\dot{x}$ and $\dot{y}$ respectively, can be calculated

$$\begin{aligned} \dot{x} &= v_l \cdot cos(\psi) \\ \dot{y} &= v_l \cdot sin(\psi) \end{aligned} \tag{3.9}$$

Finally, the vehicle position in x and y can be expressed as

$$\begin{aligned} x &= \int_0^t \dot{x} dt \\ y &= \int_0^t \dot{y} dt \end{aligned} \tag{3.10}$$

A complete ROS odometry message with pose and twist can now be created, and encoder-free odometry is achieved. It should be noted that this approach is only as accurate as the model for longitudinal velocity, and there is also a lack of feedback since the velocity is calculated from the controller input u. v_{lidar} can be used as a velocity reference, but unfiltered it is too noisy and inaccurate to function as feedback for f. ex. a PID controller. The linear velocity output v_l has been found to be relatively accurate when doing constant velocity, but is inaccurate during acceleration, which is probably caused by an inaccurate model. The IMU does a good job at finding the direction, but drift can be a problem over time. Another problem with an ESC-based approach is wheel spin, as it will assume movement when controller input is given regardless of any real movement. The ESC-based odometry module and the VESC-odometry module from MIT RACECAR (see section 3.1.2) shares some of the above functionality and design and was the inspiration behind the development. What differs the two is mostly the use of an IMU to retrieve the angular velocity instead of using the steering angle input. The solution was chosen because the steering mechanics on the truck model was not very accurate, with a lot of play.

3.5 Path Generator

In order to get the R/C truck to drive autonomously along a predefined path, a path that the vehicle can follow has to be created. The current pose of the vehicle can be retrieved from the `/scanmatch_odom` topic that is generated with Hector SLAM, where the x, y-coordinates and orientation are given. A ROS waypoint logger node has been created, that can record data for the driven trajectory and then publish the path on the ROS network. An example of a recorded path can be seen in figure 3.8, where the path is visualized in Rviz. The path is made up as a sequence of waypoints that consist of the recorded poses. When the waypoint logger is enabled, a new waypoint is added to the path every time a new pose is published from the `/scanmatch_odom` topic. The waypoint logger is activated from the joystick controller during manual operation of the truck, and an operator can create new tracks in real-time by driving the vehicle along the desired path.

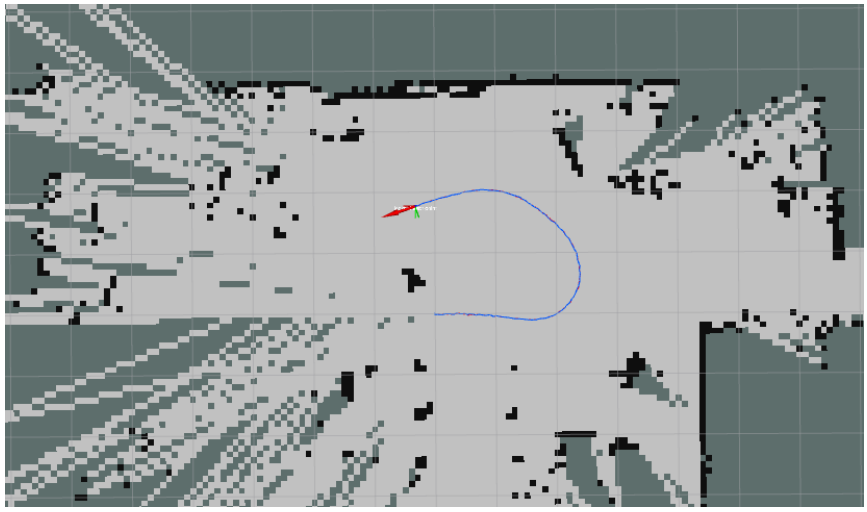

Figure 3.8: Recording a new path.

3.6 Path Tracking

A control law or tracking method is needed to translate a path into steering and throttle/brake actions, so that the vehicle can navigate the path correctly. Such controller should also consider the limitations of the vehicle such as axle length and max steering angle. This section aims to explain two of the most popular path trackers in robotics, namely Pure Pursuit and Stanley Steering Control, as described in [36].

3.6.1 Simplified Bicycle Model

Most car-like vehicles are using the *Ackermann* steering geometry model [19]. The Ackermann model works so that the wheel on the side the vehicle is turning to has a smaller turning radius than the other outer wheel for more traction and less tire slippage while cornering. The priciple is illustrated in figure 3.9. To do path tracking, a model of the car is needed. Instead of modeling the vehicle with all four wheels, it is possible to model it using the *Simplified Bicycle Model* [27]. Here, the two wheels on each axle are combined to form a two-wheeled model, much like a bicycle. It is assumed that the car will only move in a 2D-plane with no tire slippage. This simplification of the kinematics means that instead of having to control each front wheel individually it is now possible to control with only one steering output. The Simplified Bicycle Model is supposed to work well in low speeds with limited lateral acceleration and low steering angle rate.

The geometric relationship of the Simplified Bicycle Model can be written as

$$\delta = arctan\left(\frac{L}{R}\right) \tag{3.11}$$

where δ is the steering angle of the virtual front wheel, L is the distance between the front and rear axle, and R is the radius of circle the rear axle will travel with the steering angle δ, as shown in figure 3.10.

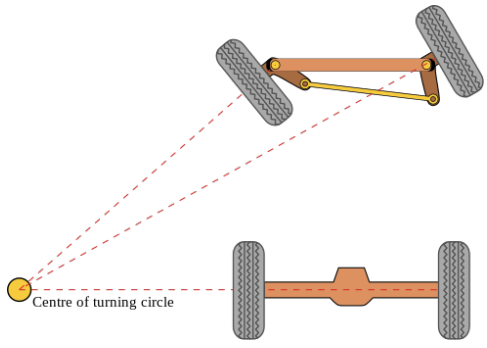

Figure 3.9: Ackermann steering geometry. Ill.: Bromskloss/Wikipedia

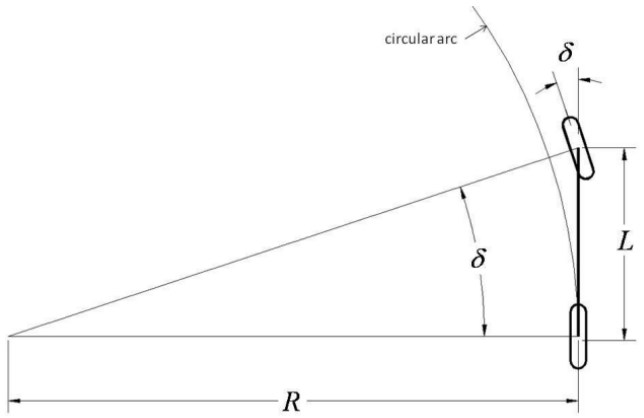

Figure 3.10: Simplified Bicycle Model. Ill.: [36]

Anti-Ackermann Model

The truck model used in this project has an *anti*-Ackermann steering layout [23], meaning that the steering rods sits in front of the wheels, opposite of the regular Ackermann model. This is a common layout in race-cars, but creates a different steering geometry that is opposite of the regular Ackermann model. This was ignored since the Ackermann model was converted to the Simplified Bicycle Model, and the car was operated only at lower speeds. Hence, the inversion of the steering layout was assumed to not have any significant effect on the driving performance.

3.6.2 Pure Pursuit

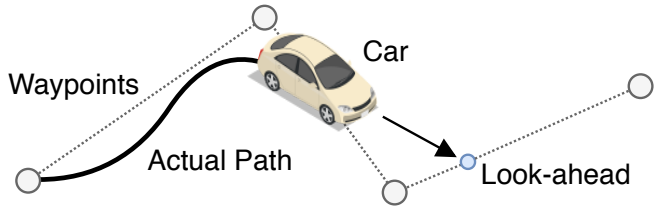

Figure 3.11: Principle of Pure Pursuit.

Pure Pursuit [6] is a popular path tracking algorithm within robotics. It calculates the appropriate curvature and steering angle so that the vehicle is able to move from its current position to a goal point (look-ahead point) on a reference path, see figure 3.11. The longitudinal velocity is assumed constant, which mean it can be changed at any point. As the vehicle moves forward, it will also push the goal point forward on the path with a predefined look-ahead distance.

How the Pure Pursuit algorithm works:

1. Determine the current location of the vehicle.
2. Find the waypoint closest to the vehicle.
3. Find the goal point
4. Transform the goal point to vehicle coordinates.
5. Calculate the curvature and request the vehicle to set the steering to that curvature.
6. Update the vehicles position.

The look-ahead distance can also be set to scale with the velocity of the vehicle. This is a commonly used method and will ensure less steering noise and more stability in higher speeds. In figure 3.12, the goal point (g_x, g_y) is set by the look-ahead distance l_d from the rear axle of the Simplified Bicycle Model to the reference path. The steering angle δ can be determined by using the angle α between the vehicle heading direction and the direction to the goal point from the rear axle. To calculate the steering angle, first an expression of the curvature κ of the circular arc between the rear axle and goal point is found using the Law of Sines [33].

$$\begin{aligned} \frac{l_d}{sin(2\alpha)} &= \frac{R}{sin(\frac{\pi}{2} - \alpha)} \\ \frac{l_d}{2sin(\alpha)cos(\alpha)} &= \frac{R}{cos(\alpha)} \\ \frac{l_d}{sin(\alpha)} &= 2R \\ \kappa &= \frac{2sin(\alpha)}{l_d} \end{aligned} \tag{3.12}$$

Then, using the Simplified Bicycle Model, the control law for the steering angle δ can be expressed as

$$\delta = arctan(\kappa L) \tag{3.13}$$

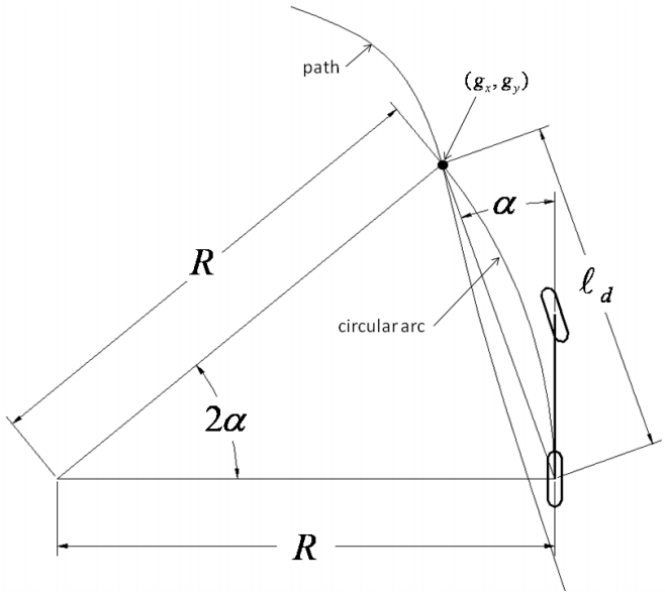

Figure 3.12: Pure Pursuit geometry. Ill.: [36]

3.6.3 Stanley Steering Controller

The Stanley Steering controller [35] was used by Stanley (section 3.1.1) in the 2005 DARPA Grand Challenge. The method is supposed to be well suited for higher driving speeds than Pure Pursuit. The Stanley controller is a non-linear feedback function of the cross-track error e_{fa}, where e_{fa} is the distance from the center of the front axle to the nearest point on the path (C_x, C_y), as shown in figure 3.13. The steering angle δ is equal to the heading error θ_e plus the cross-track error angle θ_d. It differs from Pure Pursuit in that it also needs the yaw angle of the path point.

First, the heading error θ_e is expressed as

$$\theta_e = \theta_p - \theta_v \tag{3.14}$$

where θ_p is the yaw angle of the tangent in the nearest path point (C_x, C_y), and θ_v is the yaw angle of the vehicle.

Then, the cross-track error angle θ_d is calculated

$$\theta_d = arctan\left(\frac{k \cdot e_{fa}}{v_x}\right) \tag{3.15}$$

where k is the gain tuning parameter and v_x is the longitudinal velocity of the vehicle.

The control law for the steering angle can now be written as

$$\delta = \theta_e + \theta_d \tag{3.16}$$

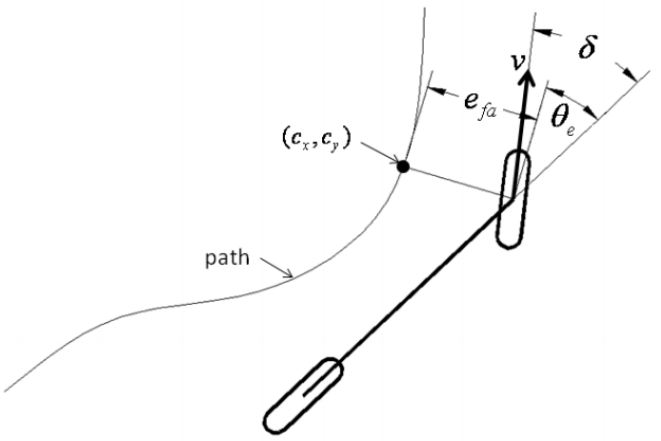

Figure 3.13: Stanley steering geometry. Ill.: Jarrod M. Snider

3.6.4 Finding the Shortest Distance to the Path

In order to navigate correctly, both the Pure Pursuit and the Stanley Steering controller relies on the shortest distance from the vehicle to the path. The path is made up of an array of indexed waypoints that are represented as x- and y-positions in a 2D-plane. The nearest point on the path relative to the vehicle can be found by simple geometry. Calculate the distance to each point on the path, as seen in equation 3.17, and select the waypoint with the shortest distance. The formula for calculating the distance is

$$\begin{aligned} dx &= x_{vehicle} - x_{waypoint} \\ dy &= y_{vehicle} - y_{waypoint} \\ d &= |\sqrt{(dx^2 + dy^2)}| \end{aligned} \tag{3.17}$$

where d is the distance from the vehicle relative to the waypoint.

The method has been extended in the code to include a dynamic look-ahead distance $L_{dynamic}$ that can vary with the longitudinal velocity v_x of the vehicle, see equation 3.18. This is used with the Pure Pursuit controller, and is a common method for gaining stability with higher speeds. The formula for calculating the dynamic look-ahead distance is

$$L_{dynamic} = k_{pp} \cdot v_x + L_{lookahead} \tag{3.18}$$

where k_{pp} is the look forward gain, v_x is the longitudinal velocity of the vehicle and $L_{lookahead}$ is the base look-ahead distance.

3.7 Dynamic Speed Control

When the autonomous driving mode is engaged the longitudinal velocity of the vehicle v_x is expressed as a function of the steering angle δ. The vehicle can increase its speed when going in a strait line, and slow down if the steering angle exceeds 0.1 radians, or 5.73 degrees, as shown in equation 3.19. This allows for dynamic speed control and can increase accuracy when following the reference path at higher speeds. There is also a velocity setting for when the charger is detected, to reduce the speed. Note that this method assumes that the speed is constant and independent of any external forces. In reality the speed will vary, pending on driving style, surface or surface materials. If an accurate odometry sensor with high refresh rate was added, this could be used in e.g. a PID controller and the velocity would be kept constant. V_{lidar} from section 3.4.2 was tried as a sensor input to a PID controller in order to control the speed, but was found to be too inaccurate as the output had to much noise on the readout, and filtering it caused some time delays making the car 'jumpy'. In manual mode the longitudinal velocity is simply controlled by the input from the left thumbstick of the joystick controller. Another method for velocity control could be to implement the look-ahead-distance from the Pure Pursuit path tracker controller to allow reduction of speed before the turn occurs by looking at the future path in front, instead of waiting until the vehicle starts to turn. This would allow for higher speeds with even sharper cornering. Since this is not a high-speed application, the implemented method has proven to be sufficient.

$$v_x(\delta) = \begin{cases} Speed_{straight} \text{ for } -0.1 < \delta < 0.1 \\ Speed_{corner} \text{ for } |\pm 0.1| < \delta < \delta_{max} \end{cases} \tag{3.19}$$

3.8 Obstacle Detector

A simple obstacle detector has been implemented in the system, where any object within a 40° field of view in front of the vehicle is detected by the LiDAR. The steering angle of the vehicle is used to adjust the angle of the field of view. If the LiDAR can detect an object in the field of view that is closer than 1m, the vehicle will come to a halt. It will resume when the object is removed. The object has to block a least 3° to be concidered an obstacle. This is to reduce the sensitivity when driving close to tables and chairs, but can easily be adjusted or turned off, along with the other parameters.

3.9 Implementation

Figure 3.14 shows how the SLAM path tracker system is connected, from the laser input to the control of the vehicle. Three different methods for SLAM were tested in this project, namely Hector SLAM, Gmapping and ORB-SLAM2. The MIT RACECAR particle filter was also tested as an alternative method for localization. In the end, Hector SLAM proved to be the best candidate for the system. It does not require odometry to work in contrast to Gmapping and compared to using the camera-based ORB-SLAM2, Hector SLAM is easier to use and delivers correct distance measurements. It also has built-in tools for trajectory tracking and pose estimation, and can easily be tuned. Some problems occurred when mapping large areas, whereas the SLAM algorithm suddenly would loose track of where is was relative to the mapped walls and objects. This seemed to be because of the limited range of the LiDAR, and not a problem with the software itself. When testing in the NTNU/SINTEF National Smart Grid laboratory the tracking worked very well and the truck could precisely map the area and track its own location in 2D. By tuning down the update thresholds for the mapping algorithm in Hector SLAM it also became more robust when doing quick turns. The implementation of Pure Pursuit and Stanley steering is based on [34] and [6]. The LiDAR provides distance readings to Hector SLAM and the obstacle detector. Hector SLAM will then create a map and estimate a position and orientation in the map. This is then used by the path generator to record the poses of the truck in order to create a path. The path tracker receives the recorded path and outputs a steering angle based on the current pose to the steering control on the vehicle. Throttle and brake is controlled either with the joystick, or the dynamic speed controller. The dynamic speed controller outputs a velocity command based on the steering angle. The velocity of the truck is calculated using laser-based odometry from section 3.4.1, but is only used for display purposes in the GUI (section 6.2), and has not been included here. All the different odometry methods has been implemented for any later use.

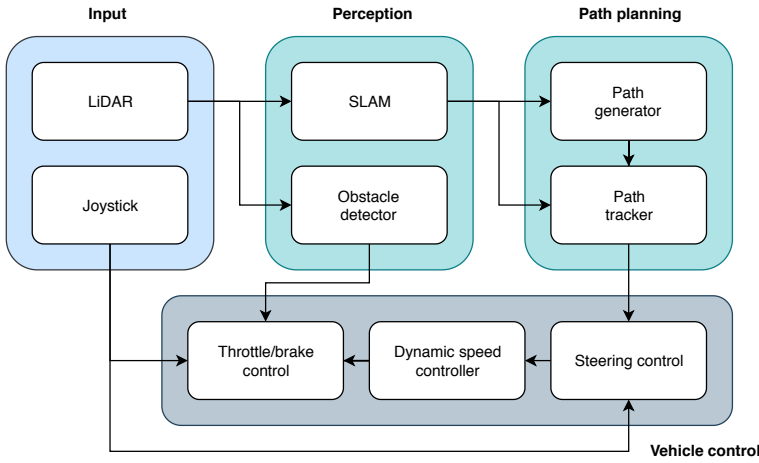

Figure 3.14: SLAM path tracker system architecture.

3.10 Results

3.10.1 Path Tracker Tuning

The path tracker steering controllers can be tuned by adjusting parameters for wheelbase length, gain and look-ahead distance. Here, the results of testing different tuning parameters are presented. Hector SLAM with the Hector trajectory server and Rviz was used to log the results. The wheelbase was measured to 0.28m between the axles.

Pure Pursuit

Pure Pursuit has two tuning parameters except the wheelbase length, namely the look-ahead distance and the look-forward gain. Increasing the look-ahead distance will gain stability but it will also tend to smooth out the corners. This can be counteracted by tuning the look-forward gain or reducing the look-ahead distance. The Pure Pursuit implemented here uses a dynamic look-ahead distance with gain, that will increase the distance as a function of the forward velocity. The result of increasing the look-ahead distance can be seen by comparing figure 3.15 and figure 3.16. The truck is presented as a red arrow, while the look-ahead point can be seen as the coordinate transform in front. The red path is the reference, while the truck's driven trajectory is the blue path. When using a look-ahead distance of 0.6 m the corners are more precise. Note that there is a small deviation of the trajectory in figure 3.15 due to the overlapping of the endings of the reference path. Table 3.2 shows an overview of some parameters that proved to be a good compromise between precision and stability.

Parameter	Description	Tuned value
k_{pp}	look-forward gain	0.8
$L_{lookahead}$	look-ahead distance [m]	0.6
L	wheelbase [m]	0.28

Table 3.2: Tuning parameters Pure Pursuit.

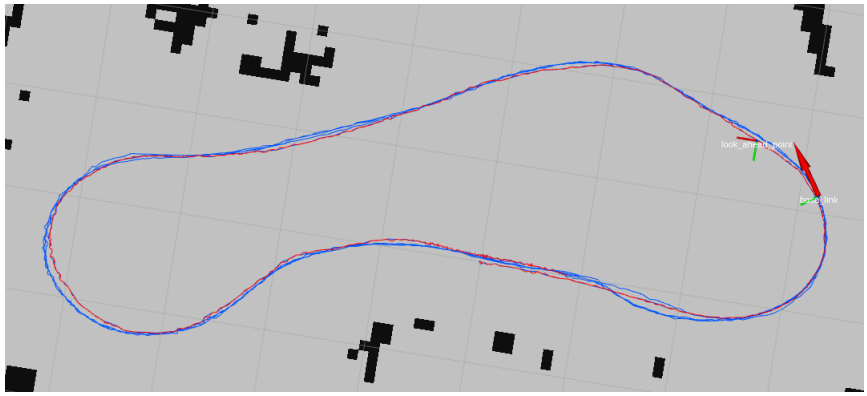

Figure 3.15: Pure Pursuit with 0.6m look-ahead, 0.8 look-forward gain.

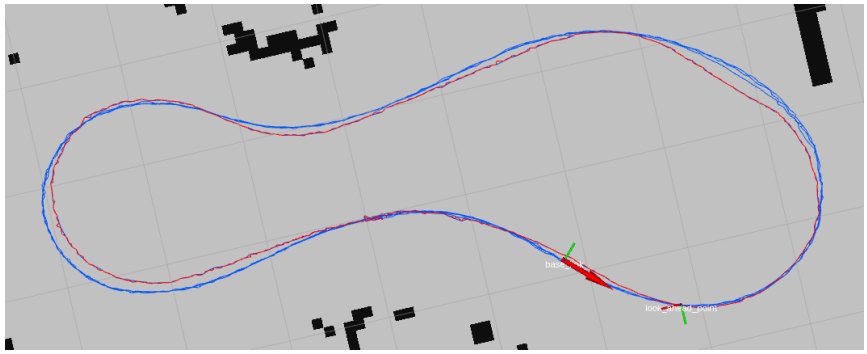

Figure 3.16: Pure Pursuit with 1.0m look-ahead, 0.8 look-forward gain.

Stanley Steering

The Stanley Steering controller has only one tuning parameter except the wheelbase length, namely the cross-track error gain. A high cross-track error gain will lead to overshoot steering, while it may struggle to follow the path if the cross-track error gain is too low. Different values for the cross-track error gain were tested, but increasing the gain quickly led to an unstable behavior of the vehicle. One reason for this could be an inaccurate steering mechanism on the truck, or a time delay between the control signal and the steering servo. Table 3.3 shows the parameters that proved to be a good compromise between precision and stability. The driving behavior for these settings can be seen in figure 3.17 where the speed is set to a constant. The truck is presented as a red arrow. The red path is the reference, while the truck's driven trajectory is the blue path. The controller was precise at low speeds but became unstable and would overshoot when going into the corners at high speeds, even when adding the dynamic speed controller to lower the cornering speed. Stanley Steering showed to be very precise, but was not as smooth compared to Pure Pursuit. This can also be because the path itself has some noise to it, and the Pure Pursuit controller will smooth it out due to it's look-ahead point.

Parameter	Description	Tuned value
k_{ss}	cross-track error gain	0.1
L	wheelbase [m]	0.28

Table 3.3: Tuning parameters Stanley steering controller.

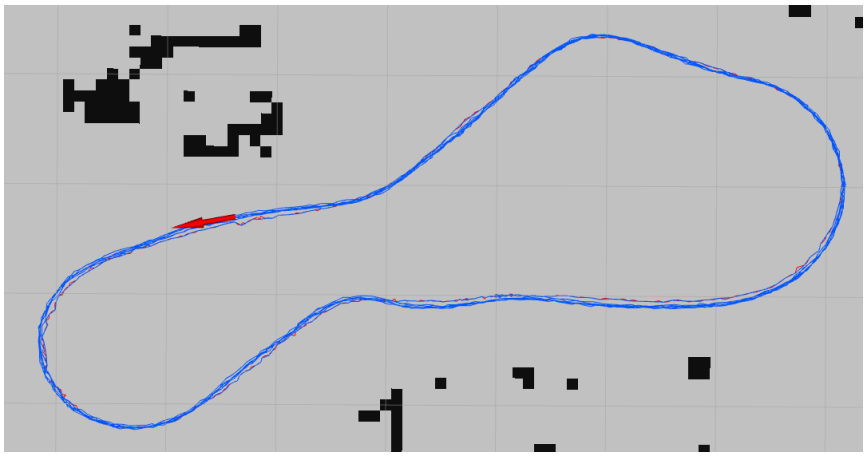

Figure 3.17: Stanley steering with 0.1 cross-track error gain.

3.10.2 Dynamic Speed Control

As mentioned in section 3.7, dynamic speed control has been implemented to increase the accuracy of the truck when driving through tight corners. The driving accuracy for both the Pure Pursuit and the Stanley steering controller were compared when dynamic speed control was enabled. The algorithm reduces the speed from $speed_{straight}$ to $speed_{corner}$ when the steering angle exceeds 0.1 radians. The path trackers used the same settings as in table 3.2 and 3.3. Three different speed settings were tested as listed in table 3.4. In the first test, both $speed_{straight}$ and $speed_{corner}$ were the same; 18% of max speed. For the second test the speed on the straight parts of the track was increased to 21%. For the third test, the straight speed was set to 27%. For each test the truck (red arrow) would have to create a new reference path (red line) since changing steering controller or parameters required the system to be restarted, hence the tracks were not exactly the same but similar. The blue lines are the driven trajectories of the truck.

Test #	$Speed_{straight}$ **[%]**	$Speed_{corner}$ **[%]**
1	18	18
2	21	18
3	27	18

Table 3.4: Dynamic speed control testing parameters.

Test #1

The results of the first test where both parameters were the same can be seen in figure 3.15 and 3.17. The truck is tracking the reference path well, with not much deviation in both cases, but it has an overall slow pace when going around the track.

Test #2

The results from the second test is shown in figure 3.18 and 3.19. Here, the truck is speeding up a little bit on the straight parts of the track. This is visible in the plots when the truck is driving into the corners. When driving out of the corner it will increase its speed. Pure Pursuit seems to track the reference path much smoother than the Stanley controller, but the Stanley controller is overall more precise. In the plot of the Stanley controller one can see that the car overshoots when leaving a corner. This can possibly be improved by further tuning of the controller.

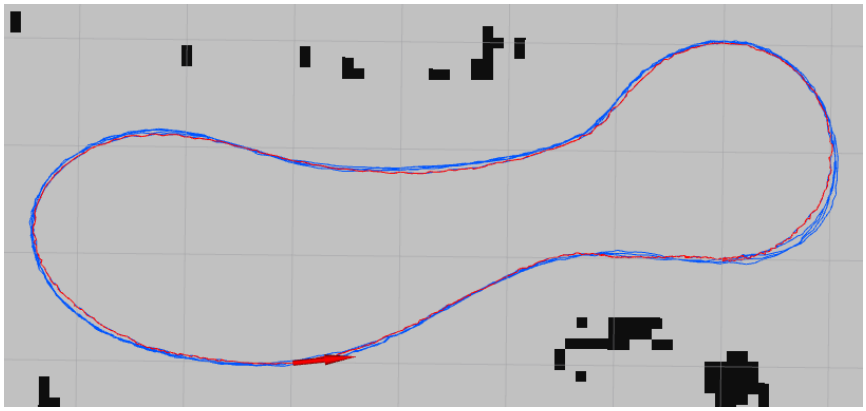

Figure 3.18: Pure Pursuit with 21% straight speed, 18% cornering speed.

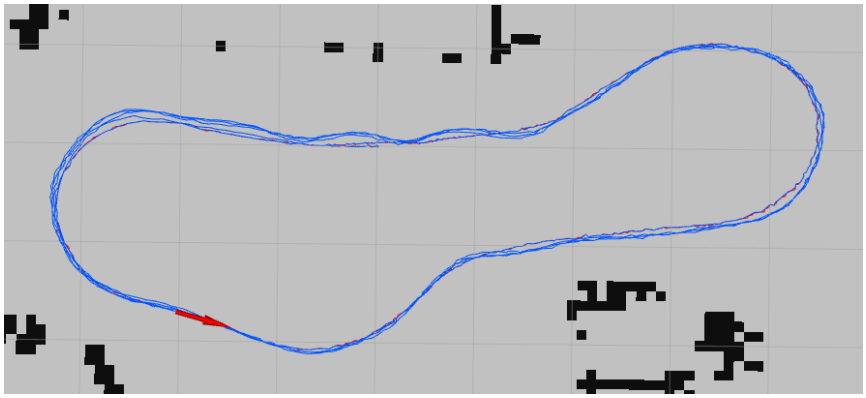

Figure 3.19: Stanley steering with 21% straight speed, 18% cornering speed.

Test #3

For the third test, the straight speed was increased by 9%. From figure 3.20 and 3.21 one can see the effect of the increased speed, with much less accurate tracking of the reference path. The overshoot on the Stanley controller is more visible than in the previous tests, when coming from a straight section. Also the Pure Pursuit controller struggles a bit to slow down fast enough.

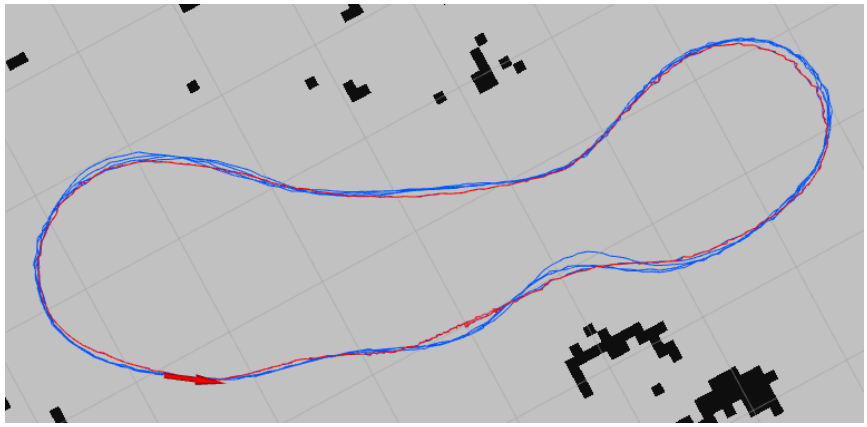

Figure 3.20: Pure Pursuit with 27% straight speed, 18% cornering speed.

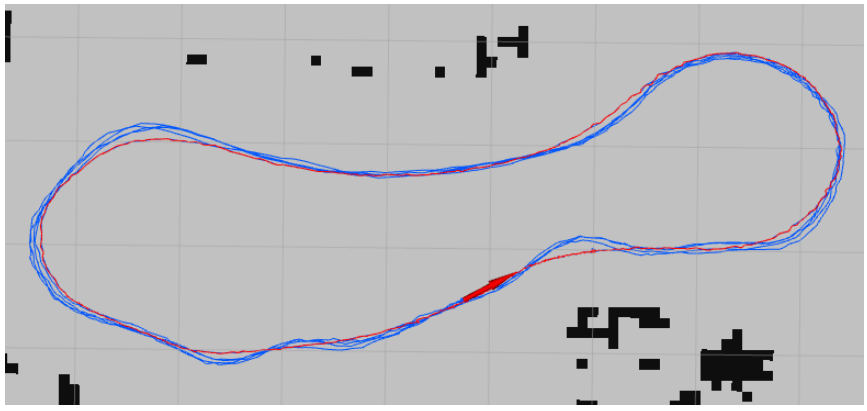

Figure 3.21: Stanley steering with 27% straight speed, 18% cornering speed.

Chapter 4

Deep Learning Steering Controller

4.1 Introduction

In recent years artificial intelligence and machine-learning has been one of the key factors to realizing self-driving cars, with companies like Tesla and Google/Waymo leading the way. Deep learning – a method that uses multi-layered neural networks – has been widely used as a method for extracting and interpret information from huge datasets. There are different methods of machine-learning that can be used to control a self-driving car, e.g. supervised or unsupervised learning. In this chapter, a form of supervised learning called behavioral cloning is tested as a way to control the steering angle of the R/C truck model based on a single camera input. The goal of behavioral cloning is to mimic the behavior of the data the model has been trained on, for example by logging steering angles from a human driver while recording camera images to create a dataset, then use the dataset to train the car controller. Camera images are fed though a convolutional neural network (CNN) – a deep network that is specifically built for processing image data. At the end the car should ideally copy the behavior of the human driver by generating a policy with actions based on the current state and observation. The work presented in this chapter is based on the DAVE-2 CNN from Nvidia described in [21], and also an article [16] where an adaptation of this CNN is used to drive a car autonomously on the Udacity self-driving car simulator [4]. Udacity is an online learning-site that was founded by former Google and Stanford University self-driving car project lead Sebastian Thrun. In this project the Udacity simulator was used to test the neural network model under development, before implementing a version on the real truck. Many people has implemented some version of this CNN on a simulator before, but the author has not observed it implemented on a small-scale electric car in real life, although Nvidia had successfully tested its model on a full-scale gas-powered car.

4.2 Related Work

4.2.1 Nvidia DAVE-2

DAVE-2 [21] is a well-known deep neural network model for self-driving cars created by Nvidia back in 2016. It is designed to be an end-to-end learning system, with only the human steering angle as the training signal and three front-facing cameras as sensor input. Performance-wise it should lead to better performance than using human-optimized criteria such as with e.g. lane detection, because it will self-optimize with a minimum number of steps. DAVE-2 was inspired by the DARPA Autonomous Vehicle (DAVE) [26] in which an R/C car was trained on hours of human driving. The training data included video from two cameras coupled with the steering input. Another inspiration for DAVE-2 was the Autonomous Land Vehicle in a Neural Network (ALVINN) system [28], where an end-to-end neural network was demonstrated on a road-going car. A block diagram of the training method for DAVE-2 is shown in figure 4.1.

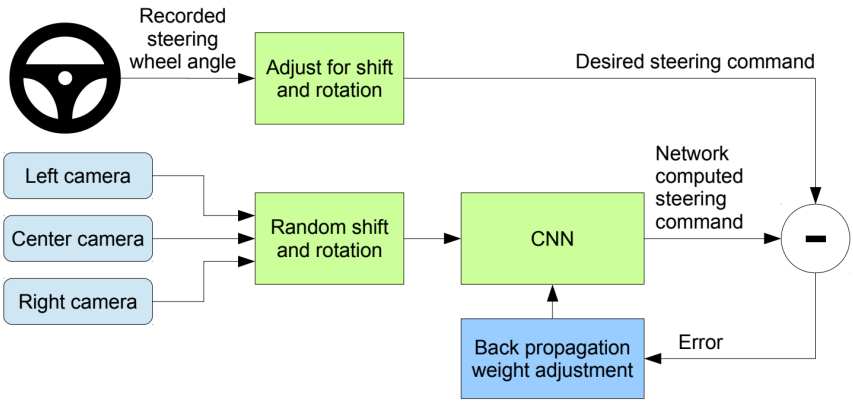

Figure 4.1: Block diagram of the Nvidia CNN training setup. Ill: Nvidia .

Images are fed into a CNN which computes a proposed steering angle. The proposed steering angle is then compared to the desired steering angle from the human driver, and the weights of the CNN are adjusted through back propagation. The model was trained with data where the car was staying within a lane on a road, and the training data was sampled at 10 frames per second to remove highly similar data input. After training, DAVE-2 could generate steering angles from a single center camera input image shown in figure 4.2. The network architecture for DAVE-2 is shown in figure 4.3.

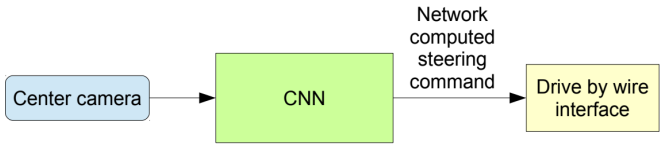

Figure 4.2: Block diagram of the Nvidia CNN steering control from a single camera. Ill: Nvidia

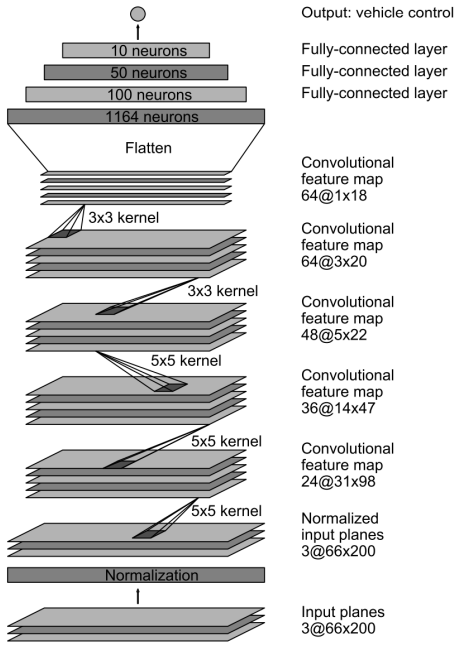

Figure 4.3: Nvidia DAVE-2 network architecture. Ill: Nvidia .

4.3 Creating Datasets

In this project, the datasets required for training the model were created in two ways, one for the simulator and one for the actual truck. Image data and steering angles were logged to file/disk for later processing. The two setups differed mostly by the number of cameras used, but the general principles were the same on both platforms, hence the same neural network was assumed to work for both cases.

4.3.1 Simulator

To test and evaluate the CNN, an open-source Unity-based simulator from Udacity was used. The Udacity self-driving car simulator [4] (Term 1, Version 2, 2/07/17), shown in figure 4.4, was built for the Udacity Self-Driving Car Nanodegree [44] to teach students how to train cars to navigate roads using deep learning. The simulator can output images from three virtual cameras in front of the car, as well as the steering angle and the speed of the vehicle. This is similar to the Nvidia DAVE-2 setup. The car can be controlled manually with a keyboard or mouse in training mode. There is also an autonomous mode where the car can be controlled by external throttle, brake and steering using an API. This allows for control via for example a python script running *SocketIO* and *Flask* which is what was used for this project. The term 1, version 2 edition of the simulator has two tracks, a race circuit and a road circuit.

Figure 4.4: Screenshot of the road circuit on the Udacity Self-Driving Car simulator.

4.3.2 Simulator Dataset

For logging training data, the simulator was started in training mode and data was recorded while driving the car manually for about four laps (8 min) on the race circuit using the built-in data-logger. Images from all three cameras along with the steering angle and speed of the car were logged. Figure 4.5 shows an example output from each of the three camera angles. For smooth steering a mouse was used as the steering controller, while the throttle and brake was controlled from the keyboard. The output was the image files and a .csv-file containing the steering angle, speed and file-paths, with one line per sample. The format of this file was: *filename-center-img*, *filename-left-img*, *filename-right-img*, *steering angle*, *not-used*, *not-used*, *speed*.

Figure 4.5: Example of image output from the Udacity Self-Driving Car simulator with left, center and right camera respectively. Images are recorded on the road circuit.

4.3.3 Truck Datasets

A test track was made in the NTNU/SINTEF National Smart Grid laboratory (figure 4.6) using black electrical tape so that the truck could be driven manually with the joystick controller, while recording data from the controller and the camera. To evaluate the amount of training data needed, two datasets were recorded from the test track.

- For the first set, the truck was driven for about 3 minutes in each direction for a total of 6 minutes. The truck was driven carefully so that it was inside the lanes at all time.
- For the second set, the recorded data was added to the data from the first set. The truck was driven for 5 minutes in each direction for a total of 10 + 6 = 16 minutes.

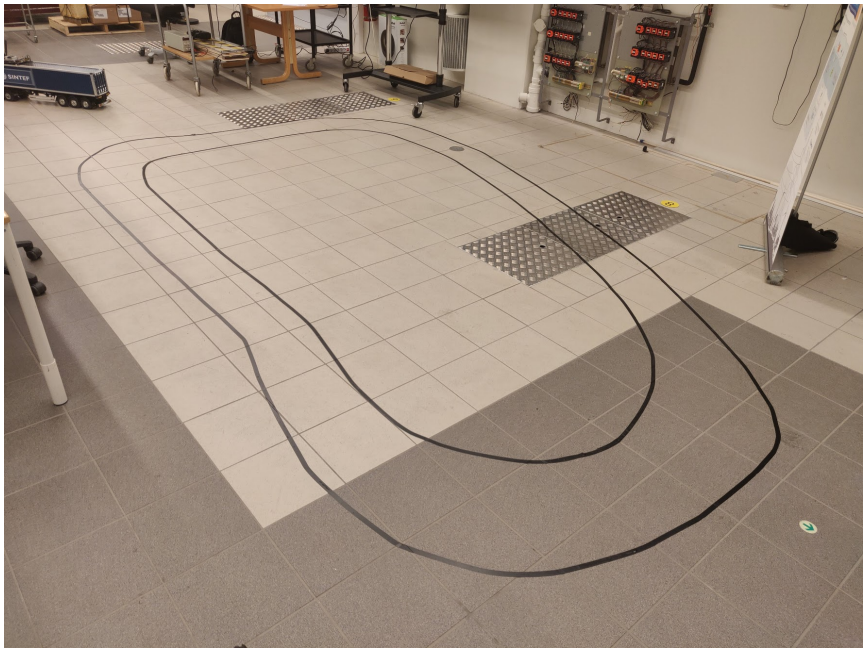

Figure 4.6: The test track used for gathering training data.

4.3.4 Getting Training Data from the Truck

A datalogger was written in Python with OpenCV and ROS to capture images from the truck's onboard camera (figure 4.7) and the steering commands from the joystick controller, with a recording rate of 20Hz. The camera images (figure 4.8) were recorded at 340x180 pixels and saved to disk. Logging was only done when the truck was moving forward to avoid saving empty data. The captured steering data and the imagefile-paths were written to a .csv-file, so that it could easily be imported into the CNN training script. The format of the .csv-file was: *filename-img*, *steering angle*, with one line per sample.

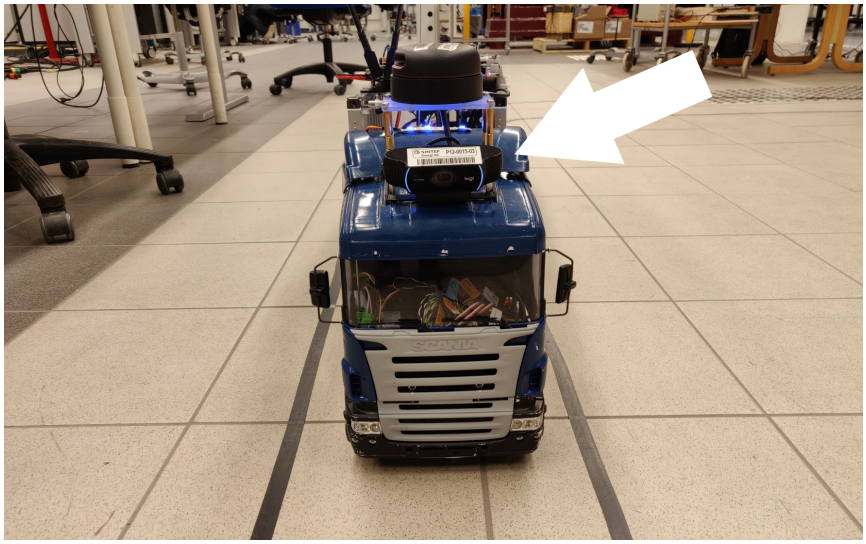

Figure 4.7: Placement of the Logitech C922 camera.

Figure 4.8: Samples of training data from the camera.

4.4 Network Model and Training

4.4.1 Neural Network Model

The convolutional neural network in this project is based on the DAVE-2 model for self-driving cars, but with some modifications suggested in an article [16] by Matt Harvey, founder of Coastline Automation, regarding implementation of the model for the Udacity simulator. The input to the network is an RGB image consisting of a separate layer for red, green and blue, and the associated steering angle. The input images are normalized to better fit the steering angle value and is resized to 100x100 pixels. A series of convolutional layers are added before the layers are flattened. Then follows two fully connected layers with dropout layers between to prevent overfitting. The final layer is the output layer with a single neuron for the steering command. The network model can be seen in figure 4.9. It was programmed in Python using Keras on top of the Tensorflow framework for easy implementation. When training is done, an .h5-file containing the finished model is generated.

4.4.2 Dataset Augmentation

To maximize the effect of the training data, several methods of dataset augmentations were used. Dataset augmentation can be seen as a way to get more data out of the same amount of input by processing the images while training. This can also make the model more robust to unknown states.

Augmentation methods used on the dataset in this project were

- Randomized image and steering-angle horizontal flipping to vary the amount of left and right turns in order to avoid any biases.
- Randomized darkening of parts of the images to simulate shadows and dark surfaces.
- Normalization of image pixel values from the default range [0-255] to [$\pm$1.0] for the real truck and [$\pm$0.5] for the simulator, to better fit the actual steering angle inputs.

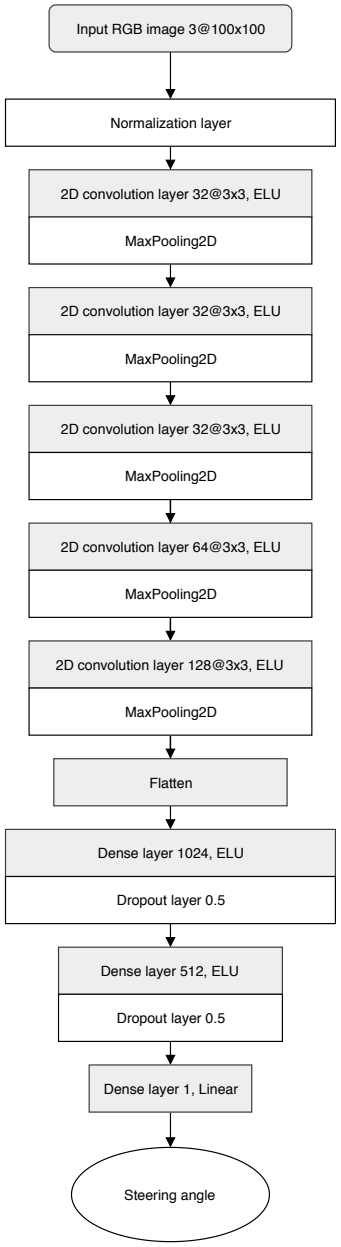

Figure 4.9: The CNN model.

4.4.3 Training

Training the Simulator

The simulator model was trained on the dataset created from driving around the race circuit in the simulator as explained in section 4.3.2. For the simulator, all three camera outputs were first used, but tests were also done using only the center camera, since this was relevant to the truck. With only one camera output, the amount of required logging time increased significantly, as expected. Training was done by trial-and-error with different variations of parameters for batch size, epochs and steps per epoch. A batch generator that splits the dataset into batches was used in order to preserve memory on the system while training. For the simulator stable results was found using the parameters in table 4.1. The simulator model was trained on a desktop computer, a Dell Optiplex 9020 with an Intel Core i7 @ 3.6 GHz x 8, 16GB RAM and without a discrete GPU.

Parameter	**Value**
Batch size	64
Number of epochs	5
Steps per epoch	Number of images / Batch size
Input shape	100x100
Optimizer	adam
Loss function	Mean Squared Error (MSE)
Activation Functions	ELU & Linear
Cameras used	Left, Right, Center

Table 4.1: CNN training parameters for simulator model.

Training the Truck

The truck model was trained on the Jetson TX2 using the GPU-version of Tensorflow and similar parameters as the simulator. Two models, one for each dataset (6 min and 16 min of driving) were created with the parameters found in table 4.2. Figure 4.10 shows an overview of how the training process works on the truck model. This is similar to the one used by Nvidia, but with only one camera output used for collecting training data. A plot of the model loss function with respect to the number of epochs can be seen in figure 4.11, where the first and second dataset are presented respectively. The best result was obtained on the second dataset with a loss of 0.0545 after completing 5 epochs.

Parameter	Value
Batch size	32
Number of epochs	5
Steps per epoch	Number of images / Batch size
Input shape	100x100
Optimizer	adam
Loss function	Mean Squared Error (MSE)
Activation Functions	ELU & Linear
Cameras used	Center

Table 4.2: CNN training parameters for 1/14 truck model.

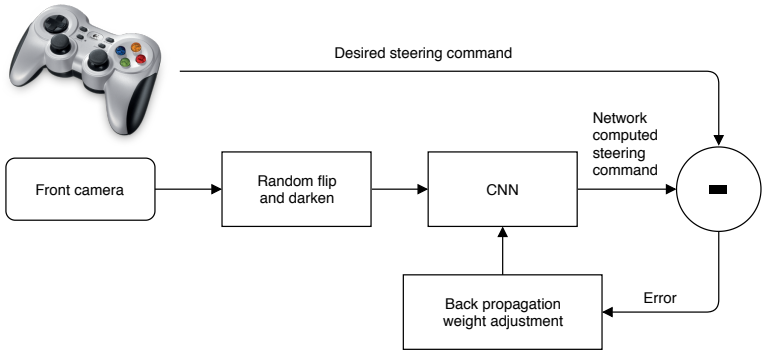

Figure 4.10: Training setup for the truck model.

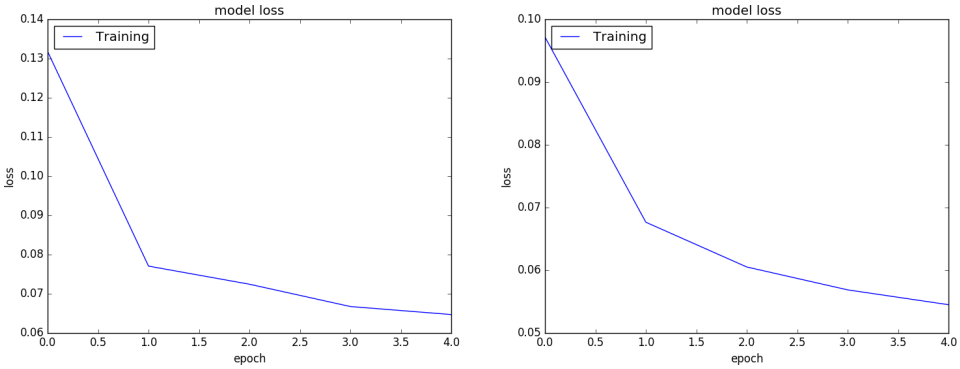

Figure 4.11: Model loss vs. number of epochs for the first (left) and second (right) dataset.

4.5 Results

4.5.1 Simulator

The virtual car was trained on the race circuit map in the Udacity self-driving simulator with a 3-camera setup similar to Nvidia DAVE-2. After completing the training process the car was set in autonomous mode and a driving script was executed. The script loads the CNN model from the generated .h5 model-file and receives the current speed and front camera image from the simulator, as seen in figure 4.12.

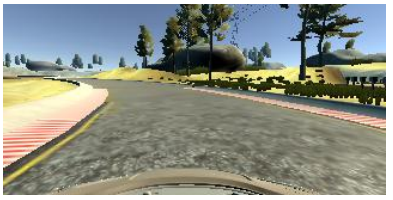

Figure 4.12: Output from the center camera image during autonomous driving.

A proportional-controlled throttle command and CNN steering command is returned to the simulator. With the settings used in table 4.1 the car was able to complete a full lap autonomously, even handling corners as seen in figure 4.13. A video demonstrating the performance of the network model can be seen here [42].

Figure 4.13: CNN controlling steering commands on the Udacity simulator.

4.5.2 Truck

On the truck, the model from each of the two datasets were tested by driving the vehicle autonomously on the track. The steering control was handled by a custom ROS steering node that captures images from the front camera with OpenCV and loads them into the CNN containing the generated .h5 model-file. The CNN then returns a corresponding steering angle and the steering node publishes it on a ROS topic to the car controller node, as shown in figure 4.14. This is similar to how DAVE-2 (section 4.2.1) is controlled. The speed was dynamically controlled as a function of the steering angle, such that it would slow down a bit in corners and accelerate on straight parts.

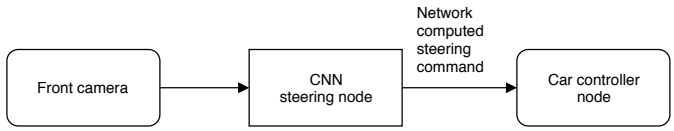

Figure 4.14: Using the CNN to output steering commands.

In order to compare the results of the two dataset models, Hector SLAM was used to track the position of the vehicle while moving (figure 4.15). As can be seen from figure 4.16 and 4.17, the difference between 6 min. and 16 min. of training data does not seem significant, and it is impressive to see how little data is required for the truck to drive itself. In real life it was easier to see the difference. The map plots shows the vehicle (red arrow) driving around the test track multiple rounds (blue lines). On the second dataset the truck seems somewhat more consistent, and driving in the opposite direction showed that the model was indeed more robust due to the increased amount of training data, naturally. Testing was also done on a reworked test track to confirm that the model would generalize, at least in the same environment as the training data was gathered. This also worked very well. On narrow corners the truck sometimes lost track of the road since the camera no longer was able to see the lanes, but this was solved by making the track a bit wider. Overall the results of the testing was showing that the system was working properly.

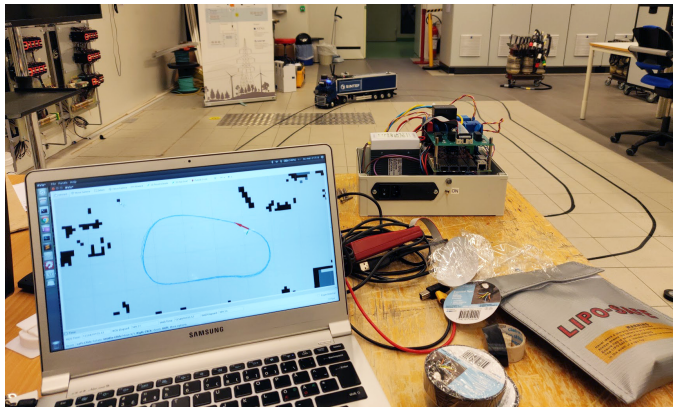

Figure 4.15: Comparing CNN models with Hector SLAM.

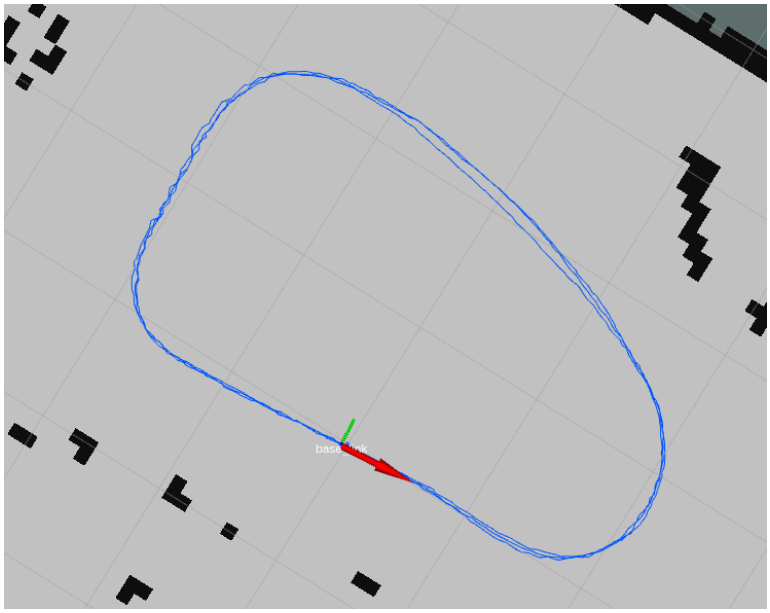

Figure 4.16: Driving with 6 minutes of training data.

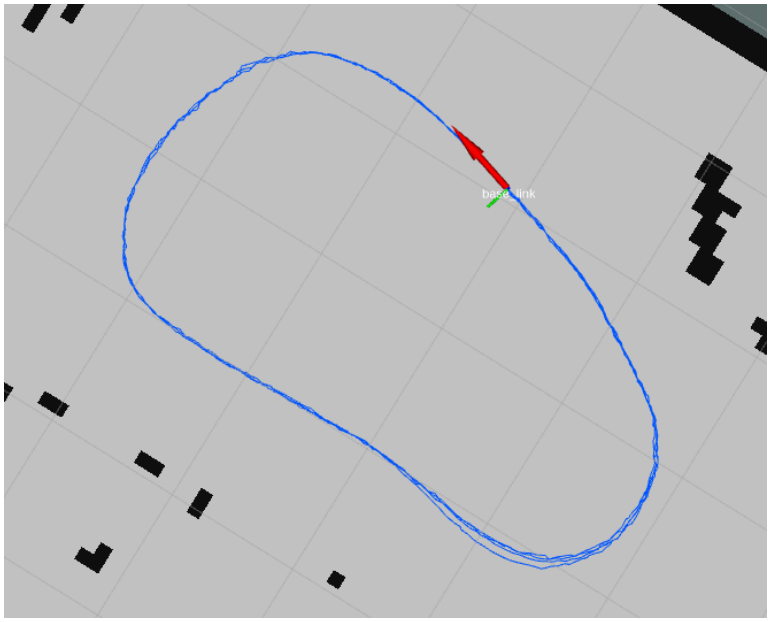

Figure 4.17: Driving with 16 minutes of training data.

Chapter 5

Computer Vision Steering Controller

5.1 Introduction

Today, many self-driving cars use one or several cameras as one of their main sensory inputs to build situation awareness. The advantage of using cameras over LiDAR is that they are better at distinguishing objects and pattern. They are also usually less expensive and has no moving parts. One use of a camera system is to detect lanes on the road while driving. Once a lane is detected, a simple controller such as PID can be used to keep the car in the middle of the lane. Many major automakers such as Volvo and Tesla has implemented some version of such system with less or more self-driving capabilities. The Tesla autopilot software [43] is an example of a current (2019) state-of-the-art system where a range of cameras and ultrasonic sensors work together to form a complete understanding of the environment surrounding the car, where the ultimate goal is to achieve full level-5 autonomy [17]. An illustration of how the car perceives the world though the camera can be seen in figure 5.1, where the image is split into segments to detect cars, road surfaces and lanes. This is also known as semantic segmentation [45]. Although the system is heavily based on supervised deep learning (explored in chapter 4) from millions of driven miles, the principle of lane detection for control using cameras and computer vision can be scaled down to achieve a simpler but still functional result. This chapter will explore the possibility to control the R/C truck using classic computer vision methods with a single camera in order to achieve path following on a road with visible lanes.

Figure 5.1: Tesla autopilot lane detection (04/2019). Ill.: [12]

5.2 Lane Keep Assist

The goal of this section is to implement a lane keep assist system by using a steering controller with lane detection. The system should be able to detect curved lanes and calculate the center position between the lanes in order to create an offset measurement for the controller.

5.2.1 Lane Detection

In order to do lane detection on curved lanes, an open-source Python script [30] by Vamsi Ramakrishnan was used as a basis for the implementation. The script calculates the position and curvature of the lanes, but there were still some adjustments required such as filtering and platform conversion. The code was modified to support live video from the camera via OpenCV instead of a video file, and ROS support was added. It also had to be converted to run on an ARM-based platform such as the Jetson TX2, although the main development and testing was done on a regular x86-computer with Linux/Ubuntu. The truck features a single camera with a resolution of up to 1080p, but for efficiency this was reduced to 640x480. The camera matrix and distortion coefficients from section 2.1.4 were used as camera parameters. As part of the image processing, the image was warped to get a bird-eye view of the road using the perspective transform [32] function in OpenCV. The image also had to be filtered in order to separate the lanes from the background. This can be challenging due to differences in illuminance and ground surface patterns. For this project black electrical tape was used as lane markers.

To filter out the background in OpenCV, the following steps were taken:

- Converting the image from RGB color to greyscale.
- Blurring the image with a 5x5 kernel.
- Creating a mask of the pixels with values between 0-110 (Pending on light conditions).
- Perform a bitwise-OR operation on the background/mask to get a binary representation of the lanes.

In order to detect lanes, the script from Ramakrishnan utilizes *blind search*, also known as uninformed search, meaning that it has no additional information about the states beyond what is provided in the problem definition. Here, the blind search algorithm will search for white pixels within the binary, filtered image (figure 5.2). The blind search algorithm can be divided into the following steps:

- Divide the image into n segments along the y-axis.
- Find the peak in each segment by using histograms.
- Validate the peak by comparing it with other/previous segments.

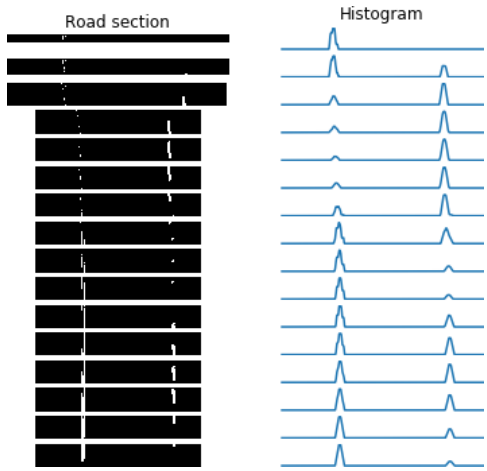

Figure 5.2: Lane detection using histogram.

Once the lanes has been detected, a Polyfit-function creates a polynomial across the detected lane points on each lane. The author has used the coordinates of these points to find the center point of the lanes, by dividing the distance between them. An interface was then made to visualize the data. The polynomials are placed on the previously transformed image and then transformed back to it's original shape using inverse perspective transform. This is placed as an overlay on top of the original camera image with some added graphics.

The measured offset between the lane center and the camera center seen in figure 5.3 and 5.4, is published to the ROS network.

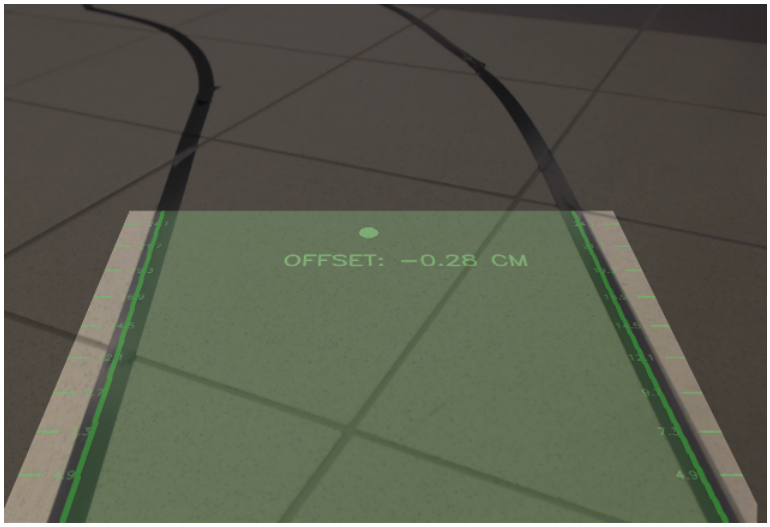

Figure 5.3: Using OpenCV for lane detection.

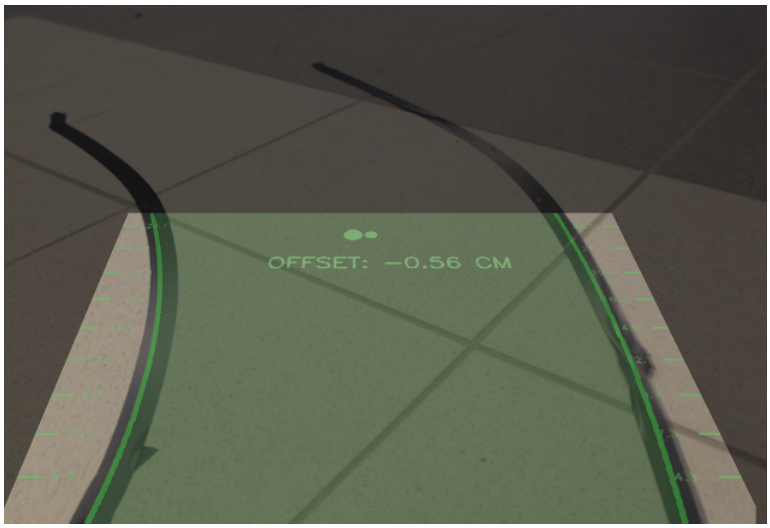

Figure 5.4: Measuring center offset during cornering.

5.2.2 Steering Controller

Once a measurement for the lane center offset is obtained, this value can then be used to calculate an error sum in a closed-loop feedback-controller using PID (figure 5.5).

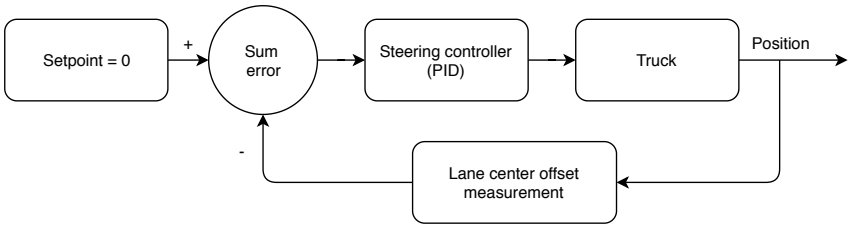

Figure 5.5: Negative feedback-loop for steering control.

PID is a common method of controlling a process by using measurements from a sensor (camera) and is used in varies applications. The goal of the controller is to reach a defined setpoint value by correcting for offset error. A PID-controller consist of three parts, the proportional (P), the integral (I) and the derivative (D). It uses a measurement error $e(t)$ to control a process, in this case the R/C truck. $e(t)$ is calculated between the camera center point and the lane center point.

Equations for the PID-controller are the following:

$$P = K_p e(t) \tag{5.1}$$

$$I = K_i \int_0^t e(t)dt \tag{5.2}$$

$$D = K_d \frac{de(t)}{dt} \tag{5.3}$$

where K_p, K_i and K_d are the tuning parameters for the controller. For tuning of the controller, a method like Ziegler-Nichols [18] can be used.

In this case the desired setpoint is set to 0 as we want the car to turn toward the center of the lane. If the center offset error increases, so will the amount of steering angle and the vehicle will self-correct.

5.3 Results

5.3.1 Performance

The computer vision method for detecting lanes was developed using Python on a x86-desktop computer running Ubuntu 16.04 with specs; IntelCore i7 @ 3.6 GHz x 8, 16GB RAM, AMD OLAND GPU. The Logitech C922 camera was connected to the computer via USB, but still mounted on the truck. During testing the system ran fine, with no major time delays. A demonstration of the lane detector running on the host computer can be seen here [41]. A P-controller was implemented on the truck, using the center offset measurement from the lane detector as error. The output from the lane detector was sent to the truck via WiFi/ROS from the desktop computer. Unfortunately, when trying to run the lane detector on the Jetson TX2, it was clear that the hardware was not up to the task with this kind of heavy image processing in Python. The system would work, but with delays of up to 1 second it made the truck not able to react in time to steer through the path. The OpenCV library installed on the Jetson has support for CUDA to enable hardware acceleration, but for the moment when this report was written, this was currently only supported in C++. A solution would be to rewrite the Python code into C++, to make it run faster or replace the Jetson with a faster computer.

Chapter 6

Integration with Wireless Inductive Charging

6.1 Onboard Charger Communication

The charger system on the truck is connected to the Jetson TX2 via a CAN-bus connector, utilizing the *can0* interface, see figure 6.1. The main purpose for this communication is to monitor energy usage and charging current in real-time, and detect whether the charger system is active. An adapter for the CAN-bus transceiver was made in order to connect the Jetson and the charger, shown in figure 6.2. The custom adapter is mounted in the J26 socket on the Jetson. The twisted wire is connected to the onboard charging system. A ROS node `can_charger_node` has been written to be able to receive CAN messages from the charger, and distribute them on the ROS network via the `/CAN_bus` subtopics. The node is using the *python-can* library and the *SocketCAN* interface module to enable support for CAN in Python. The hexadecimal values of the CAN messages are converted to a ROS message format (Int64MultiArray) with message id and data as first and second field respectively. The port was configured to communicate at 1 Mbps, see appendix. A graphical interface (section 6.2) created for this project, has been used for easy monitoring and display of the data during demonstrations of the system.

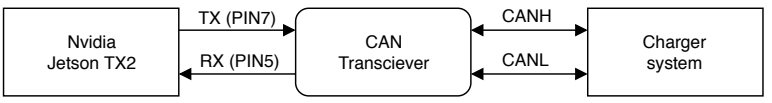

Figure 6.1: Block diagram of the CAN communication.

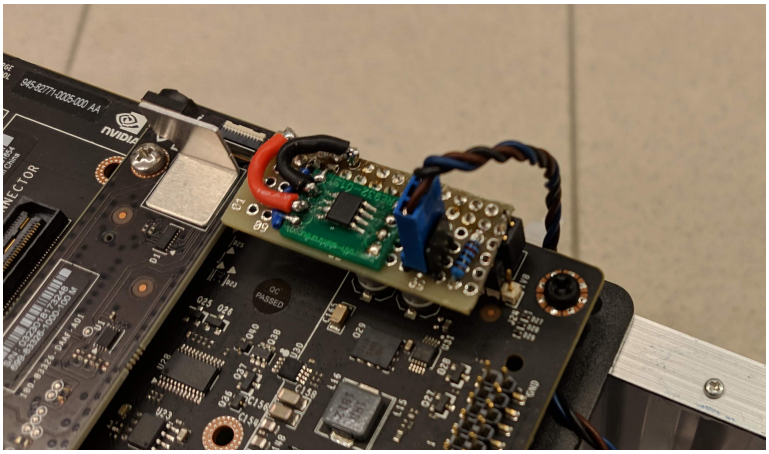

Figure 6.2: CAN-bus transceiver adapter.

6.2 Graphical User-Interface

A stand-alone GUI, shown in figure 6.3, was made with Python, Tkinter, Pillow and ROS to show current states of the vehicle. Tkinter is the standard graphical user-interface package in Python. The GUI is used to present data in a more user-friendly way. It pulls data from the relevant ROS topics, such as driving mode, speed and button states, gear and dead-switch status. But most importantly it shows the current amount of charge when driving across the induction coils, indicated by a moving needle. A video demonstration of the GUI is posted here [40]. All graphics were made from scratch in Adobe Photoshop.

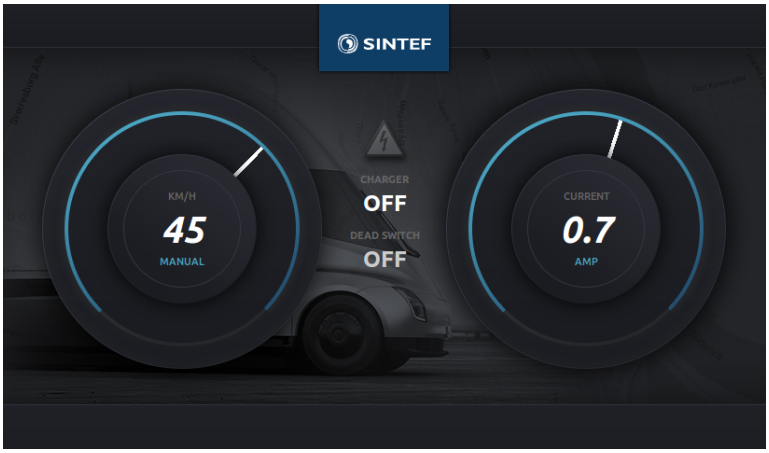

Figure 6.3: GUI made with Tkinter.

6.3 Charging Area Detection using ArUco

The onboard charging system requires extra energy to operate and is powered by a separate battery in the trailer. To be able to save energy when the charger is not active, a system for detecting the charging area using computer vision has been created. This makes automatically turning on and off the onboard charging system possible, by for example using an electronic power switch. Other uses for this system would be to switch the steering controller on the truck model when passing the charging area, from passive path following to optimal-charge path following described in section 7.7. The created system is based on detecting ArUco markers using computer vision. An ArUco marker is a binary 2D barcode often used in augmented reality and robotics. The marker corresponds to an identifier in form of a single number. A detected marker can be seen in figure 6.4. This project utilizes the ArUco library in OpenCV to generate and detect markers. The detector is implemented with ROS so that it will publish a binary number on the ROS network when a marker is detected, based on the identifier. The markers with id 1 and 2 are corresponding to charger area detection true or false respectively. It will also detect the position of the corners of the marker, so that the detector can trigger on a specific location inside the camera window frame or on a set size of the marker. This can be useful if the camera is tilted upward in such way that there is a large forward line-of-sight, to filter out distant markers. A video demonstration of the system is posted on [37] and [38].

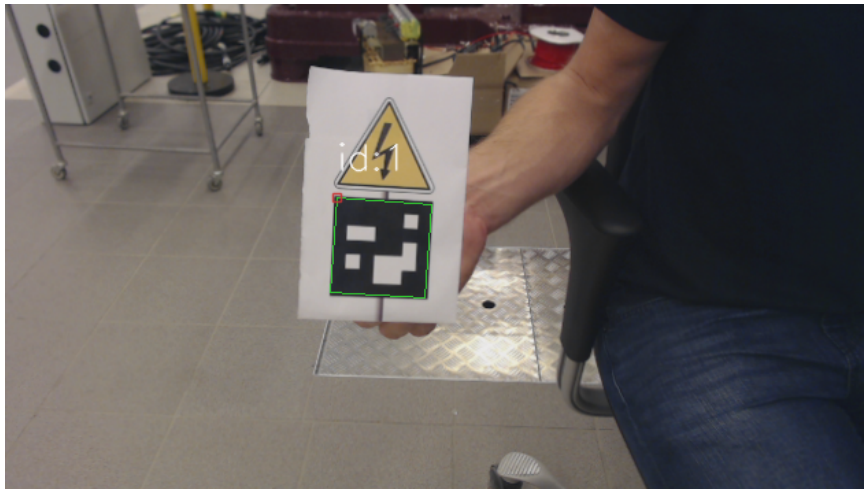

Figure 6.4: Using OpenCV to detect an ArUco marker in real-time.

6.4 Results

Here, the results of driving on a path containing the wireless charger is presented. A path was created with the SLAM- and path tracking approach from chapter 3. All data/topics were saved to a *rosbag* and resampled to 10Hz with a custom Python resampling script. In figure 6.5 the receiving current from the charger is plotted as a function of the position of the R/C truck during three laps. The size of the circles correspond to the amount of charge the truck receives. Two distinct areas shows where the charging coils are placed along the path. There is some noise from the onboard charger system, so small current values are visible even when the truck is not driving across the charger.

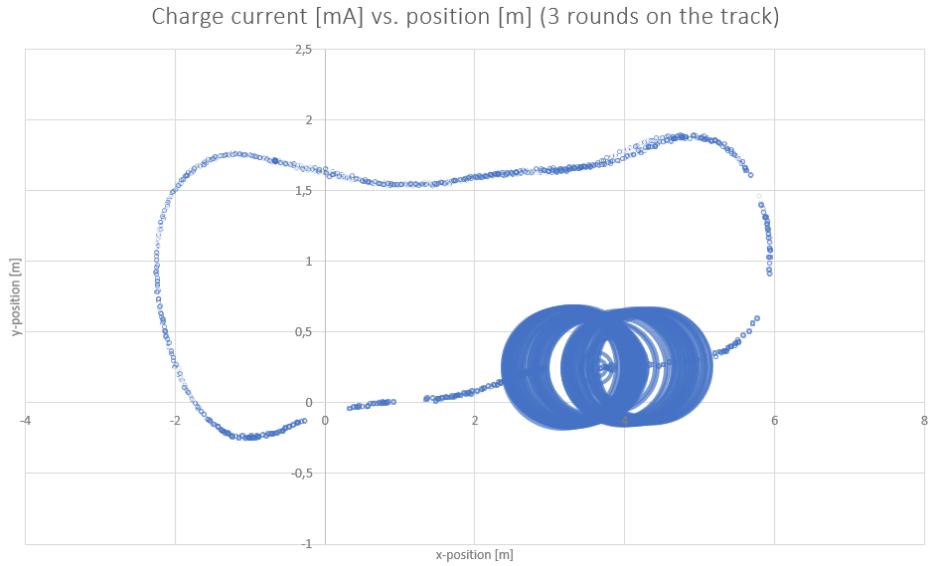

Figure 6.5: Charge current vs. position.

In figure 6.6 three states of the R/C truck's battery is plotted, namely the current consumption, the battery voltage and the charge current. The truck was driven for three rounds, hence there are three spiking areas in the plot. One for each round when the charger was passed. One can observe that charge is received from each of the two coils, with the inversion of the current consumption and a temporary increase in battery voltage. The amplitude of the current consumption and charge current varies when the truck is driving over the charging area, since the transmitted power is depending on the position. The LiDAR-based positioning and tracking system is not perfect and the vehicle is driven manually when a path is made, making it hard to optimize the power transfer. A method for optimal placement of the truck as a function of the charge value should be developed in the future in order to obtain optimal charging conditions. The average time taken to run across one induction coil was measured to about 1.3 second (An average of 13 samples at 10Hz). It is noted that the overall effect of driving across the charger at this speed is not enough to increase the amount of energy stored in the battery by any significance.

This will make the vehicle eventually run out of energy if kept driving for too long. A solution would of course be to extend the charging area with more coils or to stop or slow down the vehicle when an induction coil is passed.

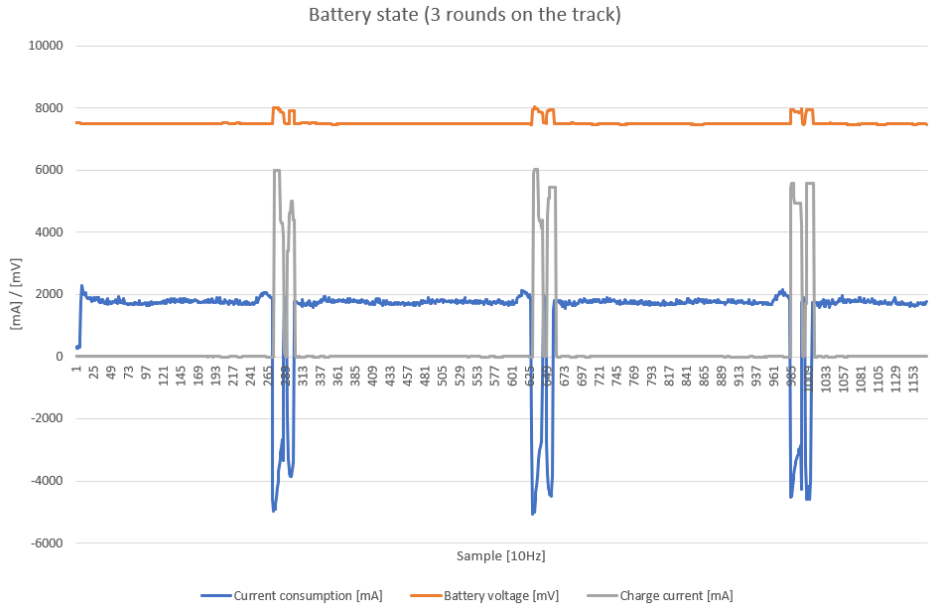

Figure 6.6: Battery states plotted with a 10Hz sampling-rate.

Chapter 7

Discussion

This chapter contains discussions regarding the results found in the previous chapters. Also general improvements of the truck's autonomous system are discussed.

7.1 Hardware

All hardware was provided by SINTEF Energy, and was ordered before the project was started. The author chose the specs of the hardware, except the truck itself which was already in use at SINTEF for manual demonstrations of the dynamic inductive charging system. This meant that the hardware specs were set from the start, so the project had to be developed under these restrictions. Some limitations were observed during the development regarding the hardware specs. The truck had low abilities for precision steering because of severe front axle play, but it would still work with the created steering controllers. In the future a more precise driving platform would probably be to prefer. Regarding the sensor systems and computing units, some bottlenecks were also discovered. The RPLIDAR is a low-cost LiDAR with limited range. During testing in long corridors and large open areas without distinct features the SLAM system would sometimes loose track of the truck's position and orientation, hence not being able to follow the path. This was somewhat compensated for by tuning down the update threshold of the mapping in Hector SLAM, such that it was more robust to rapid changes in orientation. But it still struggles in large areas where walls and objects become out of range. Fortunately, most of the testing has been in the National Smart Grid laboratory where the wall- and object distances worked very well with the RPLIDAR, and this did not turn out to be a big problem. It is still important to be aware of these limitations, and a better LiDAR could be considered. The computer chosen for the project was the Nvidia Jetson TX2 due to its high performance pr dollar. It is also a preferred hardware platform for lower-end robotics and machine-learning, and has strong community support. Also, running an ARM version of Ubuntu makes the integration easy to work with. It is not the most powerful computer though, which was noticed when trying to run un-accelerated computer vision with Python.

But this can be improved greatly by utilizing CUDA-optimized compiled code like C++. For both SLAM and machine-learning the Jetson TX2 performed well.

7.2 Localization Methods

Different methods for localization were tested, namely Hector SLAM, Gmapping, the MIT RACECAR particle filter, and ORB-SLAM2. Particle filters require a pre-mapped area to work, so one would still need to map the area with SLAM first. The benefit with particle filters is less computational cost when driving, since the mapping is already done. It only tries to relocate itself on the map. But this leads to an extra step in order to drive the truck, and a new map has to be created and saved for every new environment the truck encounters. MIT uses this filter on their RACECAR platform, but this is set in the same, constant environment. ORB-SLAM2 seemed to work fine, but was running slow on the Jetson. It had good tracking and pose estimation, but was ditched in favor of LiDAR-based SLAM. It would also have been difficult to estimate true scale from the monocular camera. Both Gmapping and Hector SLAM were considered, mostly because Gmapping is the standard SLAM mapping method in the ROS navigation package. But in contrast to Gmapping, Hector SLAM does not require odometry, which the truck did not have. Some attempts were made to make an odometry module, and these would work to some degree. But they also weren't as precise as needed, hence Hector SLAM proved to be the preferred method.

7.3 Path Tracking Methods

Two path tracking methods were tested and compared, Pure Pursuit and Stanley Steering. It was concluded that the final path tracking method for the truck, when in SLAM mode, should be to use the Pure Pursuit path tracker with the parameters and dynamic speed settings from test 1 (figure 3.18). This seemed to be the best compromise between speed and accuracy, especially when the charger platform was included along the path. Pure Pursuit also proved to be more stable and smoother than the Stanley Steering controller, with simpler tuning to get good results.

7.4 Comparison of Methods for Path Following

Three different methods for autonomous path following were proposed throughout this report. Each method has it's own strengths and weaknesses. Some of these are listed below:

SLAM-based Path Tracking

+ Does not require a track
+ Precise
+ Easy to make new paths
- Can loose track of pose in feature-less hallways or large areas

Deep Learning Steering Controller

+ Works well in large, open areas
+ Can be adapted to different areas through training
- Requires a track
- Requires training
- Only as precise as the human driver

Computer Vision Steering Controller

+ Works well in large, open areas
+ Does not require training
- Requires a track
- Not as robust as AI
- Can be hard to adapted to different areas (light conditions etc.)
- Computationally expensive

7.5 Improved Obstacle Detection

To improve the obstacle detection, a convolutional neural network like YOLOv3 [31] (You Only Look Once) could be implemented. This would add the ability to recognize people and a range of different objects instead of just measuring the distance to the objects in front. Implementing YOLO would probably require a stronger computer than the TX2, since it will currently only run at about 3 fps. An example of YOLOv3 running on the Jetson TX2 can be seen in figure 7.1. Note the high GPU usage. The network can be

trained on custom datasets, and therefore also be used to detect special objects, much like the ArUco detector tested in this project. It could also be possible to detect objects for relative positioning of the truck in respect to for example the wireless charger platform.

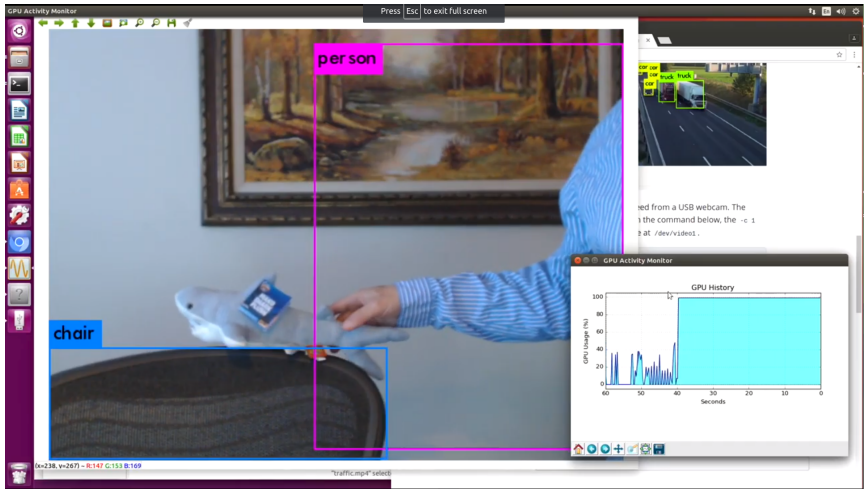

Figure 7.1: Object detection with YOLOv3 on TX2. Ill.: JetsonHacks/YouTube.

7.6 ArUco Detection

For this project, an ArUco detector was implement as described in section 6.3, to detect the charging area. When trying to do marker detection during fast driving and cornering especially, one quickly recognizes the need for a global shutter camera or faster processing speed than what is available on the Jetson TX2. As mentioned earlier in the report the Logitech C922 is a rolling shutter camera meaning that the image can refresh before it is complete. This can lead to tearing in the image, hence making detection difficult. Fast movement can also cause motion blur. This has been tried compensated by reducing the frame rate of the camera input, to lessen the load on the processor. Auto focus and auto exposure is another problem that has been looked into, but no solution for disabling these features has been found when running on an ARM-based solution such as the Jetson TX2. Note that it is possible to disable both features when running on a regular x86-based platform.

7.7 Optimal Positioning during charging

In this project the vehicle is not actively detecting the wireless charger, and it will only follow the current path independently of the power delivery from the charger coils. In order to maximize efficiency, the car should position itself in the most optimal spot available while driving across the coils. A closed-loop feedback controller that could ensure optimal charge is proposed in figure 7.2.

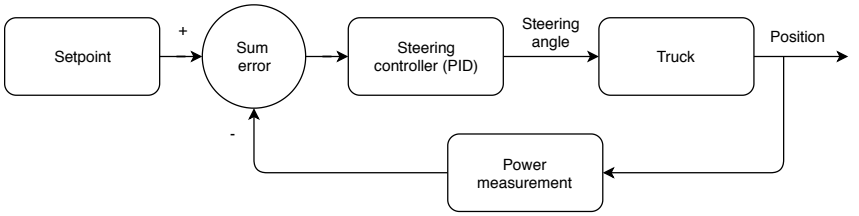

Figure 7.2: Feedback-loop based on power measurement.

This is based on a PID-controller that will use power measurements as the sensor input to calculate the positioning error offset of the truck. The controller will output a steering angle to correct the position of the vehicle for maximum charge. Currently, the truck has only one charging coil with an attached amp/voltmeter that can be used for measuring. This can make an implementation difficult, since there is no way to tell what direction the vehicle is heading when the error increases/decreases, and therefore not knowing which way to turn the steering wheel to correct the error. If possible, an array of smaller coils with individual input power measurement would be able to detect the direction by combining the measurement values. Using an example of four individual coils mounted from left to right in a straight line beneath the vehicle (figure 7.3), the two coils on the left would give a positive value while the two on the right would give a negative value. Then, the optimal path would be where the combined measured value is zero. Other, similar approaches can include computer vision methods as described in chapter 5 to detect lanes across the charging area, or hall effect sensors to detect a magnetic field across multiple sensors. These are both methods that would scale well if implemented on larger moving vehicles, but would of course require extra features such as lane markers or magnets to function. NFC or RFID would also be suitable sensors, at least on a small-scale model such as in this project, since they also can be used as distance monitors.

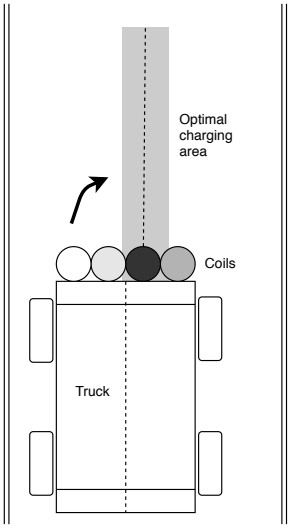

Figure 7.3: Optimal line-following based on power measurements.

Chapter 8

Conclusion and Outlook

This chapter contains the conclusion of the work done for this thesis, and also some advice on future work that could further improve the system.

8.1 Conclusions

The goal of this thesis was to propose methods for creating and implementing a self-driving system for a small-scale truck model. The system should be able to follow a path that included a wireless dynamic charger. Three different methods were implemented and tested. Both the SLAM- and AI-based methods worked very well, with different strengths and weaknesses. The pure computer vision-based method showed promises during testing. The OpenCV lane tracking worked well in optimal conditions. But especially when implemented in Python on the Jetson TX2 this method was very hardware demanding, and also dynamic adaption to different environments and light conditions proved to be difficult. The method would benefit from a compiled C++ implementation to speed up the algorithms and make use of the hardware acceleration that comes with CUDA. For now, this was not included as a usable method in the final system, and better results were obtained with the deep learning method. Regarding the SLAM-method this does not require a track in contrast to the camera-based methods, but it needs to have a pre-recorded path to be able to drive, hence a path generator/recorder had to be made. Some different SLAM-methods were reviewed, and Hector SLAM was chosen as the preferred method to gain vehicle pose and mapping due to not needing odometry to function. The combination of SLAM and a path tracker like Pure Pursuit is a very robust and precise solution as long as the LiDAR is able to detect static references like walls or objects in the surrounding environment. Pure Pursuit was also the algorithm that was chosen as the preferred path tracking method in the final version of the software. It provided smooth steering control and easy tuning. There were some limitations discovered regarding the range, rate and angular resolution of the RPlidar A8M8. But the National Smart Grid lab environment, where most of the testing was done, was small enough that this did not cause any concern. Still, the author would recommend an upgrade to the LiDAR if an even more robust sys-

tem is required. The final method, based on deep learning with a CNN does not need a pre-recorded path like the SLAM-method, but it needs a visible track. This makes it a bit less flexible than the SLAM-method since a new track has to be made if one is to move the system to a different location. The method can easily be adapted to different conditions and tracks though training, but since the method is based on supervised machine-learning the precision depends on the quality of the training data. Still, with minimal amounts of training data the truck was able to produce some impressive results of relating the camera input to a steering output. The neural network was based on the one used in Nvidia's DAVE-2 self-driving car, which had been modified to work on a car-simulator/video game. This project shows that the network could be transferred to a real-life vehicle by providing new training data and a different set of inputs and outputs. The main drawback of this method is the dependency of the camera view, but this can be solved with better camera placement, a wider lens or by using multiple cameras. The second part of the project was to integrate the wireless charging system that already had been mounted on the car. This was done in order to monitor the different states of the battery. CAN-bus and ROS worked as a bridge between the two systems after adding a transceiver on a custom circuit board to the TX2. By integrating the charger system with ROS it was also easy to record and log data for plotting and analysis. A GUI was made to present the data on a TV-screen during demonstrations. The software stack was mainly based on ROS, OpenCV and Python in an embedded Linux environment. Although learning ROS can be a steep learning-curve for newcomers, it gives the user a wide variety of tools that makes module-based development, sensor integration and communication easier. There is also a large community for support. Overall, the project has demonstrated a working small-scale self-driving truck with integrated wireless dynamic charging. A video demonstrating the final system can be viewed here [39].

8.2 Outlook

Although the system is in a fully functional and working condition, some improvements can still be made. This is mostly regarding the positioning of the truck while driving across the wireless charging setup. This currently depends on the positioning from the SLAM system, and not the placement of the charger itself. There are many approaches that can be investigated to optimize the placement during charging, with some methods mentioned in section 7.7. Since the truck is built for live demonstration purposes it would also be nice to have a more intelligent obstacle avoidance system than the simple start-stop functionality currently implemented. The sensor package can also be upgraded with some kind of odometry based on wheel-encoders or a stereo-camera to test more methods for estimating the pose in addition to only using a LiDAR. If further investigation is wanted regarding the computer vision steering controller, this should be converted to C++ to utilize the OpenCV CUDA-functionality that does not yet exist on Python in the moment of writing. Alternatively, a more powerful computing unit can be recommended.

Bibliography

[1] Abhishek Agarwal, Ariel Anders, A. F.-k. S. K. T. H., 2019. mit-racecar.
URL https://github.com/mit-racecar

[2] Alistair Charlton, T., 04 2018. Wireless electric vehicle charging explained.
URL https://www.techradar.com/news/wireless-electric-vehicle-charging

[3] Bloesch, M., Burri, M., Omari, S., Hutter, M., Siegwart, R., 2017. Iterated extended kalman filter based visual-inertial odometry using direct photometric feedback. The International Journal of Robotics Research 36 (10), 1053–1072.
URL https://doi.org/10.1177/0278364917728574

[4] Brown, A., 2017. Udacity self-driving car simulator.
URL https://github.com/udacity/self-driving-car-sim

[5] Corey Walsh, S. K., 2017. Cddt: Fast approximate 2d ray casting for accelerated localization abs/1705.01167.
URL http://arxiv.org/abs/1705.01167

[6] Coulter, R. C., Januar 1992. Implementation of the pure pursuit path tracking algorithm. Technical report, Carnegie Mellon University.
URL https://tinyurl.com/y3797vsk

[7] Elvestuen, O., 07 2018. Norway's low emissions policy.
URL https://www.regjeringen.no/en/aktuelt/norways-low-emissions-strategy/id2607245/

[8] Fortum, 2019. Fortum bygger verdens første traadløse hurtigladestasjoner for el-taxier.
URL https://tinyurl.com/y3hjjyug

[9] Fox, D., 1998. Kld-sampling: Adaptive particle filters. Technical report, University of Washington.

[10] Fusiello, A., 2019. Elements of geometric computer vision.
URL http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/FUSIELLO4/tutorial.html

[11] Giorgio Grisetti, C. S., Burgard, W., 2007. Improved techniques for grid mapping with rao-blackwellized particle filters. IEEE Transactions on Robotics 23, 34–46.

[12] Greentheonly, 09 2018. Paris streets in the eyes of tesla autopilot.
URL https://www.youtube.com/watch?v=_1MHGUC_BzQ

[13] Guidi, G., 2018. Small-scale model of inductive charging system for long-haul trucks. Technical report, SINTEF Energy Research.
URL https://tinyurl.com/y4sbty3t

[14] Guidi, G., Suul, J. A., Jenset, F., Sorfonn, I., Sep. 2017. Wireless charging for ships: High-power inductive charging for battery electric and plug-in hybrid vessels. IEEE Electrification Magazine 5 (3), 22–32.

[15] H. Durrant-Whyte, T. B., 2006. Simultaneous localization and mapping: part i. IEEE Robotics & Automation Magazine 13 (2), 99–110.

[16] Harvey, M., 2017. Training a deep learning model to steer a car in 99 lines of code.
URL https://tinyurl.com/ycmcjg28

[17] Isabel Harner, i., 10 2017. The 5 autonomous driving levels explained.
URL https://www.iotforall.com/5-autonomous-driving-levels-explained/

[18] J.G. Ziegler, N. B. N., 1942. Optimum settings for automatic controllers. Transactions of the ASME, 759768.

[19] King-Hele, D., 2002. Erasmus darwin's improved design for steering carriages–and cars. Notes and Records of the Royal Society of London 56 (1), 41–62.
URL http://www.jstor.org/stable/532121

[20] Kohlbrecher, S., Meyer, J., von Stryk, O., Klingauf, U., November 2011. A flexible and scalable slam system with full 3d motion estimation. In: Proc. IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR). IEEE.

[21] Mariusz Bojarski, Beat Flepp, U. M.-D. D. T. P. G. J. Z. D. D. L. D. J. X. Z. B. F. M. M. J. Z. K. Z., 2016. End to end learning for self-driving cars.
URL https://arxiv.org/pdf/1604.07316v1.pdf

[22] Matthew O'Kelly, Dr. Madhur Behl, V. S. D. H. A. L. W. P. N. D. R. M., 2019. F1tenth.
URL http://f1tenth.org

[23] Meywerk, M., 2015. Vehicle Dynamics. Automotive Series - Wiley. Wiley.
URL https://books.google.no/books?id=VTzODQAAQBAJ

[24] Morgan Quigley, Brian Gerkey, K. C. J. F. T. F. J. L. E. B. R. W. A. N., 2009. Ros: an open-source robot operating system. ICRA Workshop on Open Source Software.

[25] Mur-Artal, Raúl, M. J. M. M., Tardós, J. D., 2015. ORB-SLAM: a versatile and accurate monocular SLAM system. IEEE Transactions on Robotics 31 (5), 1147–1163.

[26] Net-Scale Technologies, I., 7 2004. Autonomous off-road vehicle control using end-to-end learning. Final technical report.
URL http://net-scale.com/doc/net-scale-dave-report.pdf

[27] Philip Polack, Florent Altch, B. D.-N. A. D. L. F., 06 2017. The kinematic bicycle model: a consistent model for planning feasible trajectories for autonomous vehicles?

[28] Pomerleau, D. A., 1995. Alvinn, an autonomous land vehicle in a neural network. Technical report, Carnegie Mellon University.
URL https://tinyurl.com/yyw6qkaz

[29] RACECAR, M., 2019. Racecar.
URL https://mit-racecar.github.io

[30] Ramakrishnan, V., 09 2017. Lane identification system for camera based systems.
URL https://github.com/vamsiramakrishnan/AdvancedLaneLines

[31] Redmon, J., Farhadi, A., 2018. Yolov3: An incremental improvement. CoRR abs/1804.02767.
URL http://arxiv.org/abs/1804.02767

[32] Rosebrock, A., 08 2014. 4 point opencv getperspective transform example.
URL https://tinyurl.com/ydbnn8tx

[33] Russell, R. A., 2019. Generalized Law of Sines from mathworld–a wolfram web resource, created by eric w. weisstein.
URL http://mathworld.wolfram.com/GeneralizedLawofSines.html

[34] Sakai, A., Ingram, D., Dinius, J., Chawla, K., Raffin, A., Paques, A., 2018. Python-robotics: a python code collection of robotics algorithms.

[35] Sebastian Thrun, Mike Montemerlo, H. D.-D. S. A. A. J. D. P. F. J. G. M. H. G. H. K. L. C. O. M. P. V. P. P. S., 2006. Stanley: The robot that won the darpa grand challenge. Journal of Field Robotics 23 (9), 661–692.

[36] Snider, J. M., 02 2009. Automatic steering methods for autonomous automobile path tracking. Technical report, Carnegie Mellon University.
URL https://www.ri.cmu.edu/pub_files/2009/2/Automatic_Steering_Methods_for_Autonomous_Automobile_Path_Tracking.pdf

[37] Stranden, J. E., 2019. Aruco marker detector - sintef r/c truck.
URL https://www.youtube.com/watch?v=q0i1uxSa3Lc

[38] Stranden, J. E., 2019. Aruco marker detector, camera view - sintef r/c truck.
URL https://www.youtube.com/watch?v=knDvOXJkvMM

[39] Stranden, J. E., 2019. Demonstration video, full system - sintef r/c truck.
URL https://www.youtube.com/watch?v=N_L3MuPEHa8

[40] Stranden, J. E., 2019. Graphical user-interface - sintef r/c truck.
URL https://www.youtube.com/watch?v=Jf4LbMbmYdw

[41] Stranden, J. E., 2019. Lane offset detector - sintef r/c truck.
URL https://www.youtube.com/watch?v=ZcOG-n89rWU

[42] Stranden, J. E., 2019. Udacity self-driving car simulator, neural network performance.
URL https://www.youtube.com/watch?v=fdha7VQqisE

[43] Tesla, I., 2019. Model 3.
URL https://www.tesla.com/model3

[44] Thrun, S., 2017. Self driving car engineer nanodegree.
URL https://www.udacity.com/course/self-driving-car-engineer-nanodegree--nd013

[45] Wang, T., 2019. Semantic segmentation. University Lecture.
URL http://www.cs.toronto.edu/~tingwuwang/semantic_segmentation.pdf

Appendix

8.3 Getting Started

This section describes how to get started with using the truck. There are no special requirements regarding the host computer to start the truck except an SSH interface, but for full visualization Ubuntu, ROS and the desktop GUI node are required. **The system has only been tested with Ubuntu 16.04 LTS and ROS Kinetic.**

Figure 8.1: Autonomous driving at the SINTEF Smartgrid Lab.

8.3.1 ROS Remote Network Setup

Optional: These steps are not required if visualization is not needed. In order to subscribe to the relevant ROS topics to run visualizations with the custom desktop GUI or Rviz, the host computer/laptop needs to connect to the ROS network remotely. To be able to connect to the ROS master on the truck via WiFi the computer needs to be configured.

1. Install Ubuntu with ROS on the desktop or laptop.
2. Open the hidden file /.bashrc in the Home-folder (ctrl + h to show hidden files).
3. Add the following to the bottom of the /.bashrc-file, remember to change the IP adress (f.ex. 10.42.0.170).
   `export ROS_MASTER_URI=http://10.42.0.1:11311`
   `export ROS_IP=<YOUR_IP_HERE>`
4. Save the file and close any open terminal windows.

8.3.2 Connecting to the Truck

1. Plug in the power sources and connect all wires, then power up the truck (on/off-switch) and the Jetson TX2 by pressing the power button (see figure 8.2).
2. Wait a bit until the Jetson has booted, then connect to the WiFi hotspot set up by the Jetson named `Sintef_truck`
3. The WiFi password is `JetsonTX2`
4. Log in by typing `ssh nvidia@10.42.0.1` in the terminal.
5. The password is `nvidia`

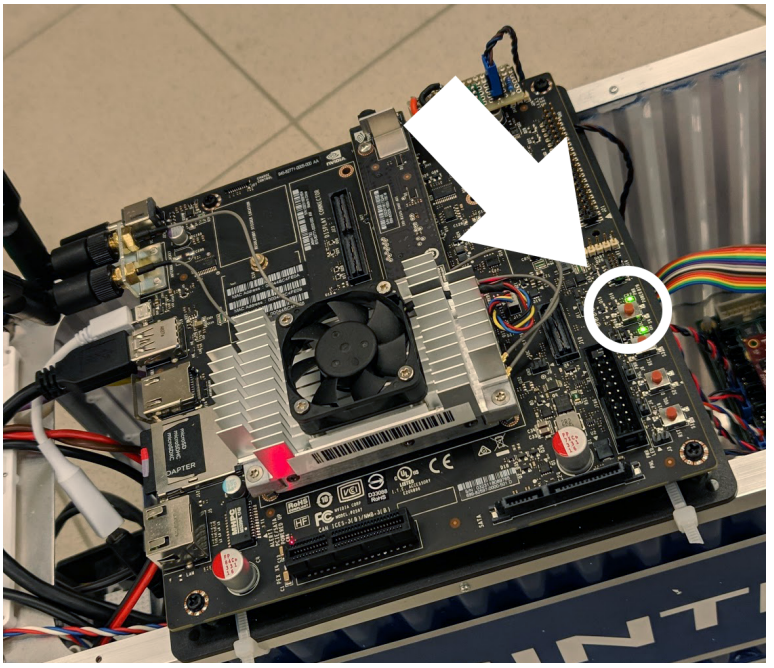

Figure 8.2: Placement of the power button on the Jetson TX2.

8.3.3 Launching the Truck

1. The truck can be launched from a single file in the Home-folder of the Jetson. Start the truck by typing `./ros_startup.sh` in the terminal window after you have logged in (Alternatively, type `roslaunch car_cmd run.launch`).

2. Push the center button on the joystick controller with the Logitech logo to wake up and connect the joystick.

3. For visualization, start Rviz (figure 8.3) by typing `rviz` in a new local terminal window. This requires ROS on the host computer.

4. After installing the `car_gui` node, the graphical interface can be accessed by the following command on the host computer: `rosrun car_gui car_gui.py`

5. Customization and overview of the ROS startup nodes can be done in the launch file called `run.launch` in the `nvidia_ws/src/car_cmd/launch` folder.

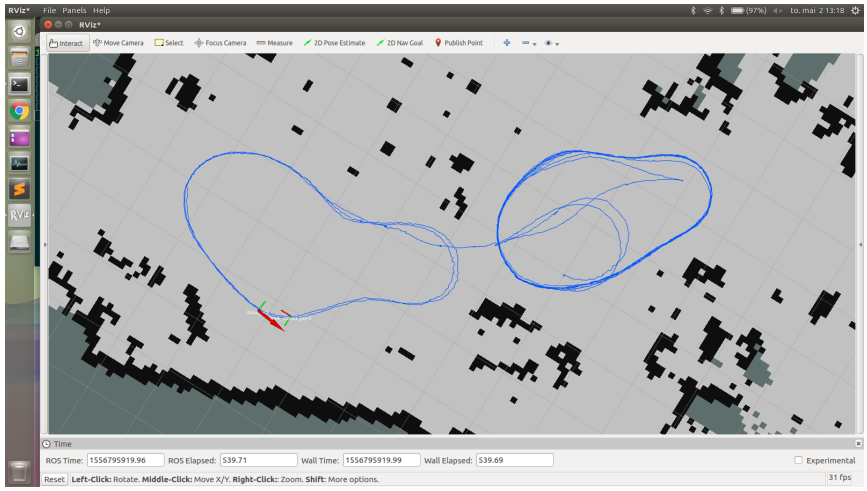

Figure 8.3: Trajectory tracking in Rviz.

8.3.4 Shutting down

To shut down the car, press ctrl + c to abort the script, and then type `sudo shutdown now` in the terminal. **Do not plug out the power directly without shutting down properly, as this may cause harm to the system.**

8.3.5 Joystick Controls

This is an overview of the joystick control functions and driving modes, seen in figure 8.4 for reference.

Figure 8.4: Overview of the joystick controls.

Dead Switch

In order to drive the truck, the dead switch button (RB) has to be kept pushed down. This is a safety measurement. Dead switch requirement can be disabled by pressing and releasing the RT-trigger on the joystick controller. This will lock the dead switch in the 'On' position so there is no need to hold down the dead switch button (RB). Push the RT-trigger again to go back to normal dead switch mode.

Manual mode

This mode is used to drive the truck manually with the joystick controller. The throttle is controlled with the left joystick, while steering is handled by the right joystick. Gears (1-2-3) can be shifted by pressing up and down on the D-pad. The mode is indicated by green solid lights.

Record a new path

Manual mode is also used to record a new path for the path tracker mode to follow. When in manual mode, push the blue X-button to start recording a new path, then drive the desired trajectory. Push the X-button again to save the path. The path can be displayed in Rviz by subscribing to the topic `/wp_path`. The mode is indicated by blue solid lights.

Path tracker mode

Path tracker mode is using the Pure Pursuit path tracker as a steering controller to follow the recorded path. The LiDAR is used for positioning, and the truck will drive to the nearest point on the path. It has dynamic speed control based on the steering angle so it will slow down in corners. This mode require a pre-recorded path. The mode is indicated by blue blinking lights.

AI/Lane follow mode

This mode uses an AI as a steering controller to follow marked lanes. It also has dynamic speed control to reduce speed while cornering. It does not require a pre-recorded path, but the camera needs to see the lanes. The AI has been trained to follow lanes consisting of black electrical tape. The mode is indicated by green blinking lights.

8.4 AI Training Procedure

This section will go through how to gather training data for the deep neural network steering controller, train a new model for path following, and run the model on the truck for autonomous driving. All files are located in the `nvidia_ws/src/dnn_steering_node/src/`-folder

8.4.1 Create a New Dataset

Datasets are created with `data_logger.py`. It uses OpenCV and ROS to capture image and control data. Follow the steps below to create a new dataset:

1. First, set up a new track that the vehicle can be driven on. It is important that the camera can see the lanes of the track while driving.
2. Connect and log in to the car via SSH, see section 8.3.2 in appendix.
3. The datalogger script subscribes to the /joy-topic. Therefore the truck's driving system has to be started by typing `./rosstartup.sh` (home-folder) or `roslaunch car_cmd run.launch` in a new terminal window.
4. Navigate to the `nvidia_ws/src/dnn_steering_node/src/`-folder and start the datalogger by typing `python data_logger.py`.
5. Now, use manual mode to drive the truck around the track with the joystick controller. Data is only saved when driving forward.
6. After logging is done, close the scripts by pressing ctrl + c in the active terminals. The driving script should also be aborted.
7. Images and steering angles are saved in the `/img`-folder and `training_data.csv`. Since the data is timestamped, you can log multiple times to the same dataset. Delete these files if a new dataset is created, or rename the output from the datalogger-script.

Troubleshooting: If the camera cannot start, make sure no other ROS node such as the `dnn_steering_node` is currently using it. Try to comment out this node from the launch-file (or create a new launch-file without DNN steering enabled).

8.4.2 Train a New Model

After the dataset has been created, the model has to be trained. This can be done by using the script called `train.py` in the node-folder. Training parameters can be adjusted in the train() module at the bottom of the script. The script will pull data from the `/img`-folder and `training_data.csv`.

1. **Optional:** Adjust the training parameters in the script for batch size and number of epochs. Batch size=32 and number of epochs=5 works fine.
2. Run the training-script by typing `python train.py` in the node-folder.
3. Wait for the training procedure to complete. A graph will pop up and show the results.
4. A new model-file called `model.h5` has now been created.

Note: The truck uses the default file called model.h5 directly from the node-folder. Be sure to backup any previous models before training.

8.4.3 Drive

The truck will automatically load the model.h5-file when the default launch-file (`run.launch`) is started, and the model is updated. The driving script is called `drive.py` and is also placed in the node-folder (`nvidia_ws/src/dnn_steering_node/src/`).

1. Place the truck inside the track.
2. Run the `run.launch` again to start the truck.
3. Switch to AI-mode on the joystick controller.
4. Hold down the deadswitch-button (RB) or press the deadswitch-lock-button (RT) to make the truck drive autonomously.
5. Release deadswitch-button (RB), press the deadswitch-lock-button (RT) or switch to manual mode to stop the truck.

8.5 ROS System Structure

This is an overview of the ROS nodes used for the final version of the truck's autonomous system. A graphic representation can be seen in figure 8.5, where nodes and topics are represented as ovals and rectangles respectively. **Note:** the GUI is not represented here.

- `/car_cmd` or car commander is the main system controller for distributing signals to the Teensy microcontroller in order to control and move the vehicle. It will take steering angle input data from the steering controller and steering angle plus speed from the joystick and convert it to PWM values for the microcontroller to read. It also controls the vehicle speed and the different driving modes. The driving modes can be switch by using the buttons on the joystick controller. The different LED indicator modes are controlled from this node and it has a terminal GUI for system status.
- `/hector_mapping` is the Hector SLAM node used for localization and mapping. It will estimate position and orientation (pose) using the laser scan matcher package, and return a map of the surrounding environment.
- `/hector_trajectory_server` is used to record the trajectory the vehicle has driven. This is useful for measuring path tracking performance.
- `/waypoint_logger` is used to record a path during driving. The node is activated by a buttonpress on the joystick controller. When finished recording the node will publish the path to the path tracker.
- `/pure_pursuit` is the path tracker steering controller node for Pure Pursuit. It will subscribe to the current state of the vehicle (pose and velocity) and return a steering angle in radians based on the state.
- `/dnn_steering` is the deep neural network steering controller. It uses the front camera to track black lanes and outputs a steering angle to the car controller.
- `/joy` is the joystick node that will publish joystick input commands.
- `/can_charger_node` talks with the onboard wireless charging system to retrieve data for the GUI mainly.
- `/heartbeat_broadcaster` is used to send an alive-message from the Jetson TX2 to the Teensy micro-controller so it will stop the vehicle if the Jetson fails during operation.
- `/esc_vel_pub` is used to publish the current velocity of the vehicle in [m/s] by using motor-based odometry based on the output from the car command node to the ESC.
- `/rplidar_ros` is the RPLIDAR node for publishing laser distance measurements from the LiDAR.
- `/serial_node` is used for serial communication between the Jetson TX2 and the Teensy micro-controller in ROS.

- `/base_to_laser_broadcaster` publishes the transform between the base_link-frame and the laser-frame
- `/scan_to_base_broadcaster` publishes the transform between the scan-frame and the base_link-frame

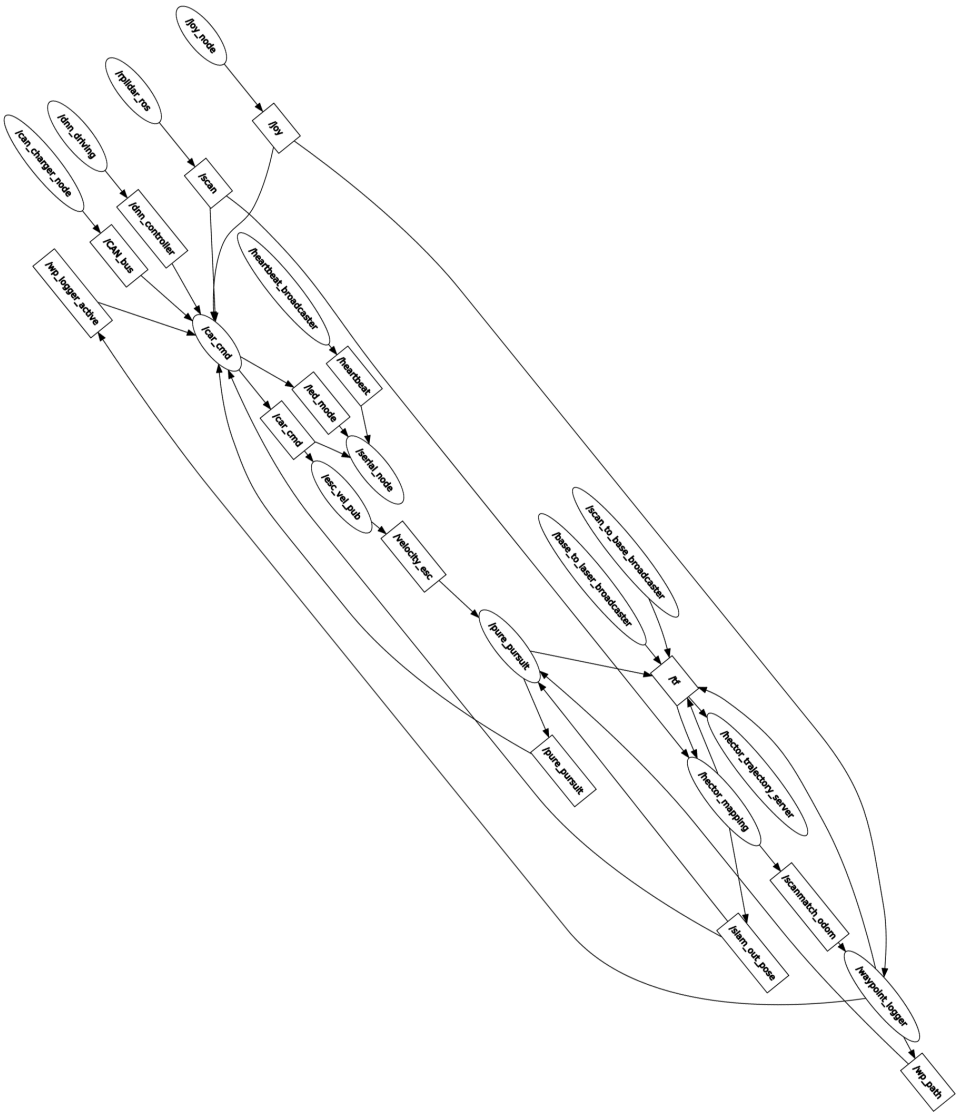

Figure 8.5: Overview of the ROS system structure.

8.6 Installation

8.6.1 ROS Kinetic

To install ROS Kinetic on a computer, follow the instructions on:
`http://wiki.ros.org/kinetic/Installation/Ubuntu`.

To install ROS Kinetic on the Jetson TX2, follow instructions on:
`https://github.com/jetsonhacks/installROSTX2`.

ROS node installation

ROS nodes can be installed by placing them in the workspace folder, e.g. *nvidia_ws/src* for this project.

8.6.2 Desktop GUI

The desktop GUI can be downloaded from the project folder on GitHub:
`https://github.com/joneivind/Self-Driving-Truck`.

8.6.3 ROS Nodes

These are the ROS nodes used for developing and testing the system that were not made by the author. The installed nodes are placed in the `nvidia_ws/src` workspace folder:

Nodes used in the Final Version

- **Rplidar** - adds support for the RPLIDAR A8M8
  `http://wiki.ros.org/rplidar`
- **Joy** - adds joystick support
  `http://wiki.ros.org/joy`
- **rosserial** - adds serial communication
  `http://wiki.ros.org/rosserial`
- **Hector-SLAM** - adds the Hector SLAM ROS package
  `http://wiki.ros.org/hector_slam`

Nodes used during Research & Development

- **usb_cam** - adds USB camera support
  `http://wiki.ros.org/usb_cam`
- **MIT RACECAR Particle Filter**
  `https://github.com/mit-racecar/particle_filter`
- **ORB-SLAM2** - ORB-SLAM2 package with ROS and CUDA support for Jetson
  `https://github.com/hoangthien94/ORB_SLAM2_CUDA`

- **ROVIO** - (Robust Visual Inertial Odometry) framework for ROS
  `https://github.com/ethz-asl/rovio`

8.6.4 Udacity Self-Driving Car Simulator

The Udacity Self-Driving Car Simulator (Term 1, version 2) was used to test and train the AI model, and is available for Windows, Mac and Linux as an open-source project on GitHub:
`https://github.com/udacity/self-driving-car-sim`

8.6.5 OpenCV 3.4.0 on Jetson TX2

Nvidia Jetson TX2 requires a special version of OpenCV to enable CUDA support. A description on how to compile OpenCV 3.4.0 with CUDA can be found in this blogpost from JK Jung: `https://jkjung-avt.github.io/opencv3-on-tx2/`

8.6.6 Controller Area Network (CAN)

The CAN controllers on the Jetson TX2 are not enabled by default from Nvidia. To enable CAN bus, the CAN controller modules must be added to the kernel. A tutorial for building custom kernels on the TX2 can be found on the JetsonHacks website:

`https://www.jetsonhacks.com/2017/03/25/build-kernel-and-modules-nvidia-jetson-tx2/`

1. Download the *buildJetsonTX2Kernel* repository from JetsonHacks (GitHub), then run the `./getKernelSources.sh` script. This will download the sources from Nvidia and open an editor where the kernel modules can be selected.

2. Under the tab called `CAN Device Drivers`, enable `Bosch M_TTCAN`. Give the kernel a unique name, press save, and exit the editor.

3. Follow the rest of the tutorial from JetsonHacks. Run `./makeKernel.sh` to build the kernel, then `./copyImage.sh` to copy the kernel to the boot directory. Restart the Jetson to activate the new kernel.

4. Open a new terminal window and type the following to enable the *can0* controller:
   ```
   $ modprobe can
   $ modprobe can_raw
   $ modprobe mttcan
   $ ip link set can0 type can bitrate 1000000 dbitrate 2000000
   berr-reporting on fd on
   $ ip link set up can0
   ```
   then type `ifconfig -a` to check if the new controller is visible.

5. Open `/etc/modprobe.d/blacklist-mttcan.conf` and comment out the first line to ensure the mttcan module is loaded.

6. Add the following to `/etc/network/interfaces` to load the module on boot

```
source-directory /etc/network/interfaces.d

auto can0
iface can0 inet manual
pre-up /sbin/ip link set $IFACE type can bitrate 1000000 dbitrate
2000000 berr-reporting on fd on
pre-up /sbin/ip link set up $IFACE
up /sbin/ifconfig $IFACE up
down /sbin/ifconfig $IFACE down
```

7. The connections between the CAN controller (*can0*) and the TI SN65HVD230D CAN transceiver is shown in table 8.1. The R_s pin on the transceiver is grounded (logic low) to enable high speed mode (1Mbps). **Note that the CAN controller interface is located at the J26 header pins on the Jetson.**

Nvidia Jetson TX2		**SN65HVD230D (VP230)**	
Connector Label	**Pin (J26)**	**Connector Label**	**Pin**
3.3V DC	2	VCC	3
GND	10	GND	2
CAN0 TX	7	D	1
CAN0 RX	5	R	4
-	-	CANH	7
-	-	CANL	6
GND	10	Rs	8
-	-	Vref	5

Table 8.1: Nvidia Jetson TX2 to CAN transceiver.

8.6.7 CP210x Support for Jetson TX2 with RPLIDAR

The RPLIDAR A8M8 is using a CP210x USB to serial converter (FTDI) to communicate over USB. The stock Jetson TX2 kernel does not have built-in support for CP210x in the kernel, so a CP210x USB to serial converter module has to be added. To enable CP210x support, download the *installACMModule* repository from JetsonHacks (GitHub), and run the `./installCP210x.sh` script.

8.7 Teensy Connection Chart

5V and GND is connected from the VCU to the servos, ESC and LED strip.

PJRC Teensy 3.2	Device	Connector Label
3.3V	BNO080 IMU	3V3
19 (SCL0)	BNO080 IMU	SCL
18 (SDA0)	BNO080 IMU	SDA
GND	BNO080 IMU	GND
2	Ws2812B LED strip	Data
20	Steering servo	Signal
21	ESC	Signal
22	Gearing servo	Signal
GND	Vehicle Control Unit	GND

Table 8.2: Teensy 3.2 connection chart.

8.8 Code Overview

This is an overview of the final code parts of the project with a short description for each part. The descriptions are separated by folders. For the latest update, visit the project repository on GitHub: `https://github.com/joneivind/Self-Driving-Truck`

Teensy

Contains the code for the Teensy microcontroller that is used to control the servos, IMU and LEDs with rossserial.

car_cmd

This contains the car controller node (`car_cmd.py`) and is the central hub for controlling the truck. This is the node that will communicate with the microcontroller in order to control the vehicle, or to change the color of the led strip. The folder also contains the main launch-files for ROS such as `run.launch`.

dnn_steering_node

This folder contains the Deep Neural Network steering controller software (`drive.py`). It also contains the software to record training data (`data_logger.py`) and the AI trainer (`train.py`) that will generate a model-file in .h5-format.

ackermann_odom

ROS node for estimating odometry using velocity commands from the car controller and angular velocity from IMU, without the use of wheel sensors. See description in section 3.4.2.

aruco_detector

Contains the ArUco marker detector described in section 6.3. Uses OpenCV to detect markers and publishes the id's on a ROS topic.

can_charger_node

Contains the CAN bus to ROS interface node for publishing charger system CAN messages such as battery current and voltage. See description in section 6.1.

car_gui

Contains the GUI (`gui.py`) described in section 6.2.

car_setup_tf

Tf broadcaster ROS node for publishing the transform between the truck and the LiDAR.

cv_lanetracker

Contains the OpenCV-based lane detector ROS node described in chapter 5. The node will output a lane center offset [cm] if two lanes are detected by the camera.

esc_vel_pub

ROS node for publishing an estimate of the velocity based on the control input from the car controller to the ESC. See description in section 3.4.1.

heartbeat_broadcaster

ROS node for publishing alive-messages to the microcontroller. The truck will stop if this is not received.

path_trackers

Contains the two path trackers used in this project, namely Pure Pursuit (`pure_pursuit.py`) and Stanley Steering (`stanley_steering.py`). Both nodes will publish a steering angle based on the current location relative to a reference path.

rosbag_resampler

Script for resampling logged rostopics from a rosbag to a given sampling-rate and output the data to a .csv-file.

rplidar_pwm_ros

This is a modified version of the official RPlidar ROS node, with support for changing the PWM value of the motor in order to control the rotational speed of the LiDAR. Code by David Portugal: `https://github.com/davidbsp/rplidar_ros`

waypoint_logger

Contains a ROS node for creating path messages as a series of waypoints, in order to generate paths for the path trackers to follow.

Jon Eivind Stranden

Autonomous driving of a small-scale electric truck model with dynamic wireless charging

NTNU
Norwegian University of
Science and Technology

SINTEF