CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Predictive Control for Autonomous Articulated Vehicles

Bachelor Thesis

Nils Andrén, Alicia Gil Martín, Kevin Hoogendijk,
Lars Niklasson, Fanny Sandblom, Filip Slottner Seholm

Department of Computer Science
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden 2017

Bachelor Thesis: DATX02-17-83

Predictive Control for Autonomous Articulated Vehicles

Nils Andrén, Alicia Gil Martín, Kevin Hoogendijk,
Lars Niklasson, Fanny Sandblom, Filip Slottner Seholm

Department of Computer Science
Chalmers University of Technology
University of Gothenburg
Göteborg, Sweden 2017

Predictive Control for Autonomous Articulated Vehicles

Nils Andrén, Alicia Gil Martín, Kevin Hoogendijk,
Lars Niklasson, Fanny Sandblom, Filip Slottner Seholm

Bachelor Thesis: DATX02-17-83
Department of Computer Science
Chalmers University of Technology
University of Gothenburg
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Cover: The model semi-trailer truck used in this project.

Göteborg, Sweden 2017

Abstract

Autonomous driving is a highly topical research area, where significant positive impacts on safety and environment can be made, especially in the trucking industry. The vehicles in this industry often consist of a tractor unit combined with a trailer. This project focuses on navigating a model semi-trailer truck through an urban-like environment. A number of challenges arise from these settings, such as path planning and control through sharp turns and crossings, combined with obstacle avoidance. This needs to be done with high precision, considering that the whole articulated vehicle needs to stay within the bounds of the road. Since the vehicle will need to take critical decisions quickly, the performance and reliability of the control system is also important.

Working towards a real world solution, this project offers a complete prototype implementation in a scaled testbed environment for articulated vehicles. To achieve this, we have mathematically modeled the vehicle, created a path planning algorithm that takes the trailer into account when calculating a suitable path, and developed a controller that makes the vehicle follow this path. These components have been integrated on a single-board computer (Raspberry Pi 3) embedded on the vehicle. The evaluation of the system shows satisfying results, where the prototype is able to do on-the-fly path planning while staying within the allowed areas of the test track. The system is also extensible and modifiable, and can be extended in future student projects.

Keywords: Automation, Automated Control, Path planning, Articulated vehicles, Autonomous vehicles, PID Controller

Sammanfattning

Autonom fordonskörning är ett högaktuellt forsknings- och utvecklingsområde som kan bidra med stora positiva effekter för miljö och säkerhet, framför allt inom lastbilsindustrin. Ett fordon inom den industrin består oftast av en dragvagn kombinerat med en släpvagn. Det här projektet fokuserar på att navigera en modell-lastbil genom en stadsliknande miljö. Ett antal utmaningar uppstår i en sådan miljö, såsom vägplanering och styrning genom skarpa svängar och korsningar, kombinerat med undvikande av trafikhinder. Detta måste göras med hög precision då hela det ledande fodonet måste hålla sig i körbanan. Eftersom fordonet snabbt behöver ta viktiga beslut, är prestandan och pålitligheten på reglersystemet viktig.

Som resultat har en fullständig implementering av en nedskalad, självkörande lastbilsprototyp framställts. Detta har krävt matematisk modellering av det ledade fordonet, skapande av en algoritm för vägplanering som tar hänsyn till släpet vid uträknandet av en passande referensväg, och utveckling av en regulator som får fordonet att följa denna referensväg. Dessa komponenter är integrerade på en enkortsdator (Raspberry Pi 3) på fordonet. Evalueringen av systemet visar tillfredställande resultat då den färdiga prototypen klarar av vägplanering i farten och samtidigt hålla sig inom tillåtna områden på testbanan. Systemet är även gjort för att enkelt kunna modifieras och vidareutvecklas för att underlätta för framtida studentarbeten.

Den här rapporten är skriven på engelska.

Nyckelord: Automation, Reglering, Vägplanering, Ledade fordon, Autonoma fordon, PID-regulator

Acknowledgements

We would like to express our gratitude to Fredrik Svensson from Volvo Group Trucks Technology for the proposal of this project, as well as the engagement and interest in our work.

We also thank Chalmers for giving us the opportunity to work in a team and take on a real world problem. We especially thank our supervisors Thomas Petig and Elad Schiller for their expert advice, encouragement, and enthusiasm during the project.

Thanks also to the kind and helpful people in the Bachelor's group *Evaluation and Development of Imagebased Positioning Systems for Self Driving Scaled Vehicles*, with whom we have shared the access to our lab and the model truck.

Finally, we thank Andrew Söderberg-Rivkin and Sanjana Hangal, who developed the position estimation system used in our project, and last years Bachelor's group *An affordable vision-based alternative for global localization and steering of autonomous vehicles*, who further contributed to the system.

Contents

1. Introduction

According to the WHO [1], road traffic accidents cause over 1 million deaths and up to 50 million non-fatal injuries every year. Almost all accidents are caused by human errors such as having a slow reaction time, driving aggressively or being distracted [2]. With the emergence of autonomous vehicles, these factors can largely be eliminated, leading to a dramatic decrease in the number of traffic accidents [3]. Without the human factor, the driving can also be optimized, with increased fuel efficiency, shorter travelling times and reduced carbon emissions as a result.

As with all new technology, the mechanics need to be thoroughly tested in secure environments before being employed in the real world. Autonomous driving is a hot research topic, with numerous current projects at both Chalmers and KTH [4],[5],[6],[7]. In this project, we are working with a physical vehicle, maneuvered on a realistic track with roadways and restricted areas, whereas many other projects are limited to simulations [4] or less challenging environments [6].

With a more well-defined environment, and the development of an effective path planning algorithm, we are able to run trajectory planning in real time. We do this on a single-board computer, while others have had performance issues even with the use of desktop computers [6].

Much of the research in this field is focused on smaller, car-like vehicles [4],[5],[6]. When it comes to trucks and other articulated vehicles, the automation process involves additional challenges due to the motion patterns of the trailer and the situational need for extra road space. The purpose of this project is to explore these challenges, with the use of a model semi-trailer truck in a scaled testing environment. The resulting system will be made publicly available, and is intended to work as a testbed for future student projects.

1.1 Problem Description

The main objective of this project is to create a complete system, designed to make an articulated vehicle drive through various traffic scenarios on a test track. The system should be installed on a model semi-trailer truck and run in a controlled test-environment. With the position and direction of the truck given, the system should be able to navigate from point A to point B in a safe manner.

To achieve this, various subproblems needs to be solved. In order to get from one location to another, the vehicle will have to follow a path. This path needs to be carefully planned, to ensure that the vehicle stays on the road and avoids obstacles.

For articulated vehicles, this problem becomes more difficult, as the trailer needs to be taken into consideration. These vehicles usually need to deviate from the current lane in order to manage for example a T-crossing. To be able to predict the behaviour of the trailer and avoid collisions, mathematical models of the articulated vehicle are needed for the path planning.

Given continuous updates on vehicle state, this path will then have to be followed. For this, a control mechanism needs to be implemented to minimize the deviation from the projected path. The performance of the system, and in particular the path-planning, is important, as the time the vehicle is idle and waits for instructions should be as short as possible.

Submodules for different tasks will have to be built, and a big challenge will be to integrate these modules into a complete working system. The system architecture needs to be carefully designed, to allow for effective communication between the different components.

Another objective of this project is to create a testbed for further development. Therefore, the components should be loosely coupled, to increase reusability and maintainability. It should be possible, with minimal effort, to replace a component with an alternative implementation that provides the same service.

1.2 Evaluation Criteria

To achieve acceptable levels of safety, portability and performance, the finished system should satisfy the following criteria:

1. When driving, the entire vehicle should stay within the bounds of the road.

2. The vehicle should make minimum use of the opposite lane.

3. The path planning should be fast and flexible enough, so that the vehicle does not have to stand still and wait for instructions for more than a few seconds.

4. All parts of the system, except for the user interface, should be able to run on a single-board computer embedded on the vehicle.

1.3 Scope

The main focus of this project is to create a functioning control system, for automating an articulated vehicle. This includes the development of a hardware API, a manual control system, an automatic control system, a path planning algorithm, and a user interface. The development of a positioning system is not included in the scope of the project, and we assume the availability of such a system.

The practical elements of the project are carried out in a scaled environment, limited by the boundaries of a medium sized room. The vehicle used is a modified model semi-trailer truck, which is run on a test track designed to cover a set of challenging traffic scenarios, including T-crossings and a roundabout. The track is far from easy to navigate, so precise control and efficient and accurate motion planning is required.

The scope covers a single vehicle. Obstacle avoidance is implemented, but is limited to static objects. Neither reversing nor parking is considered.

1.4 Our Contribution

The finished project features a complete autonomous control system, that allows a scaled semi-trailer truck to safely navigate through a variety of complex traffic scenarios, including T-crossings and roundabouts, with excellent reliability. The automation strategy consists of two parts: path following using a feedback loop control, and dynamic motion planning using our self-designed path planning algorithm. The relative performance of the system is satisfactory. We are able to perform on-board path planning with real-time obstacle avoidance, all while the truck is in motion.

As we have developed the system from scratch, we chose to design it with safety concerns, portability, and modularity in mind. A dedicated simulation environment was implemented, which has allowed us to develop, test and analyse parts of the system, without being dependant on any hardware. The final system has been installed and tested on a single-board computer (Raspberry Pi 3) embedded on the truck, to ensure the portability of the system.

We intend to make our testbed open-source, as it is extensible and modifiable, and could be of use to similar projects in the future.

2. Background Knowledge

This chapter goes through the theory behind two main parts of the system: the controller and the path planning. It also gives a general overview of the Robot Operating System (ROS) framework and the GulliView localization system. Each section provides the basic knowledge required to understand the implemented system, as well as the decisions and results presented later on.

2.1 PID Feedback Loop

Control theory has allowed automation to become a reality by managing the behaviour of devices and making them perform in a desired way. In a controlled system, the control actuators, the reference and the output need to be identified in order to understand the system. The mathematical description of the system behaviour (the kinematic model for the tractor unit) is called the plant of the system, and will be used to find the expected behaviour of the system.

Once the plant of the system has been studied, its stability is checked [8],[9]. This is done in order to see if the output of the system remains naturally bounded for any initial state, and then select the suitable control strategy. For unstable systems, closed-loop strategies are selected, in order to obtain a final stable system and make it robust. The PID (proportional, integral and derivative) feedback loop [10] is one of many closed-loop strategies. It balances the unmeasurable disturbance that the vehicle can experience, and eliminates any steady errors on the performance.

In a feedback control loop, the actuation signal sent to the plant is changed in real time, so that its output follows the reference signal. Then, the output of the plant is gathered by use of sensors and compared to the reference signal. The difference is applied as an error signal, to bring the output of the plant closer to the reference, as shown in Figure 2.1.

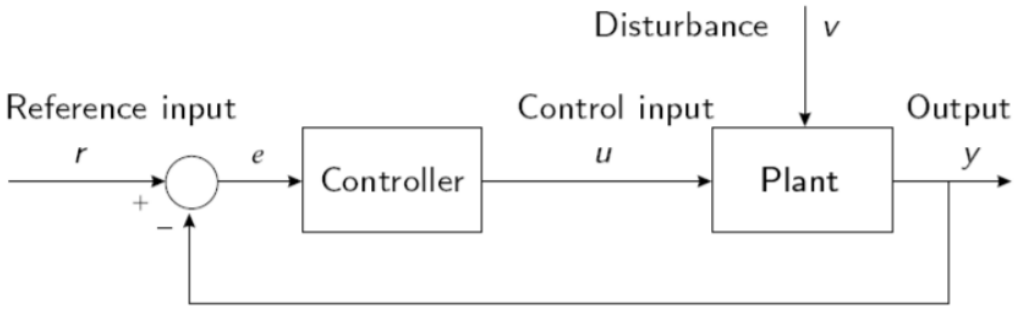


Figure 2.1: Feedback loop strategy

The control applies a proportional, integral and derivative response to the error. The proportional part introduces a gain to the error. The integral accumulates the error, to compensate it even if constant disturbances affect the modelled system. For contrasting the slow response of the integral part, the derivative is lastly introduced, and generates an actuation proportional to the derivative of the error. Once a suitable control strategy is selected according to the system and desired performance, the control parameters Kp, Ki and Kd can be selected. These values will be used in the control, to output the actuator signal in continuous time with the following equation:

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

2.2 Path Planning

In order for an automated vehicle to know the path to its final destination, a motion planning algorithm is needed. Going from point A to point B in a quick and efficient manner is one of the difficulties the planning needs to solve. In an environment with realistic traffic scenarios, the planning also needs to consider that the vehicle should stay in its own lane rather than the opposite lane, to not collide with obstacles, and to always remain on the road. Any good path planning algorithm should take these constraints and priorities into account and produce an efficient path.

In order for the planner to produce a path, a predictive behaviour is needed, and the planner needs to some time in advance find a path that works. Otherwise, the vehicle would end up in a situation where finding a valid path requires stopping, reversing, and redoing the turn as shown in Figure 2.2. This is not a viable behaviour.

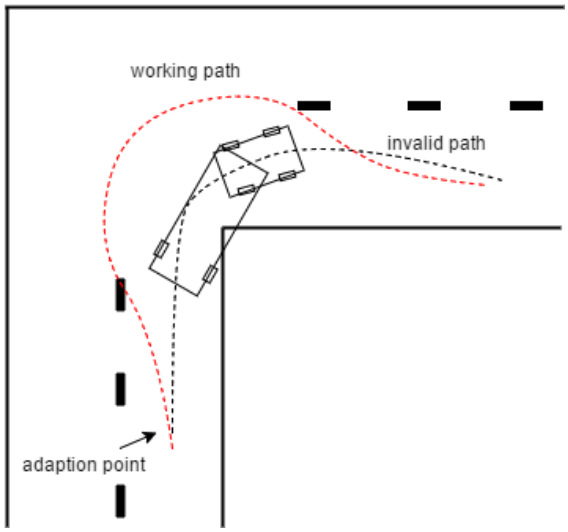


Figure 2.2: Showcase of the need for a predictive behaviour. The adaption point is where the vehicle needs to start adapting for the turn.

To be able to find an efficient path, one approach is to sample different paths and compare them to find the best one. To make a path measurable, a cost function needs to be defined, where the path with the lowest cost is considered the most efficient solution. The cost function for each sampled path is then compared and the path with the lowest cost is considered the most efficient one. For a comprehensive explanation of the path planning problem and different ways to solve it, we refer the reader to S.M. LaValle's book *Planning Algorithms* [11].

2.3 ROS: Robot Operating System

Robot Operating System [12] is a set of frameworks that assist in the development of software for robots. The distributed node network spine and built-in publisher subscriber system makes the developed systems extremely flexible. The whole system can run on one computer, or easily be divided amongst several computers connected with network cables or Wi-Fi.

The core of ROS is the publisher subscriber system where messages can be published on topics. A topic is like a billboard, on which publishers can publish messages (put a poster on the board). Each time this happens, all of the subscribers get notified (sees the poster) and gets the message. There can be several publishers and several subscribers on the same topic. Messages are automatically sent over the network if your setup allows that. Each topic has a specific type of message and there is a set of predefined message types. However, custom messages can be created that consist of several other message types. For an example on how to use publishers and subscribers in ROS, see Appendix A.

The distribution points that publish and subscribe to topics are called nodes. A node is a process that uses ROS to communicate with other nodes. A system can be retained on one computer or distributed over several different computers over several different geographical locations.

For visualization of the system there is a package called RViz, that visualizes a selection of standard message types effortlessly. RViz shows a 3D-world in which objects can be placed, maps can be shown and input can be taken (see Figure 4.3 on page 18).

2.4 GulliView: Vision Based Localization System

GulliView [13] was created by students at Chalmers University of Technology as a part of the Gulliver project, a testbed for development of vehicular systems. It is a low cost solution for localization of moving vehicles, using normal USB cameras, and open source libraries such as OpenCV [14]. The positioning is built on a set of easily detected QR code style tags called AprilTags [15], which are developed specifically for vision localization. For a vehicle to be detectable, it needs to be fitted with a tag, as can be seen in Figure 2.3.

The localization system used in this project consists of 4 cameras, mounted in the ceiling above the test-track. This setup is the result of a previous Bachelor's project [5].

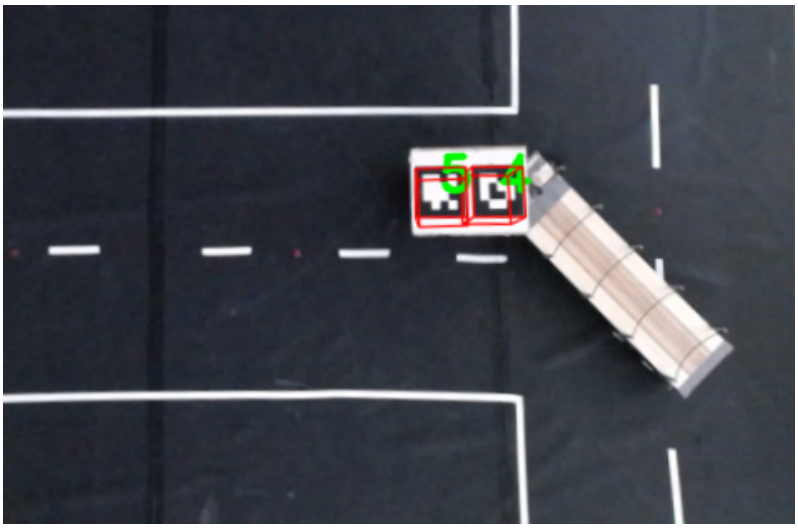


Figure 2.3: An example of GulliView locating a moving vehicle by detecting the AprilTags mounted on top. The position of the vehicle is published on a ROS topic.

3. Vehicle and Testbed Environment

In this chapter, we describe the environment in which the experiments are conducted. First, the given materials including the sensors and data gathering are detailed. Then, the motion of the scaled truck is presented as a mathematical model to use further on during the implementation steps. Finally, the chapter is closed with information about the lab, and the virtual visualization environment.

3.1 The Vehicle

The vehicle is a modified radio controlled model truck, in scale 1:14 (see Figure 3.1). For measurements of the truck, see Appendix B. It is equipped with a single-board computer (Raspberry Pi 3), replacing the radio receiver. This lets us connect to it via Wi-Fi to gather and monitor sensor data as well as control the servos.

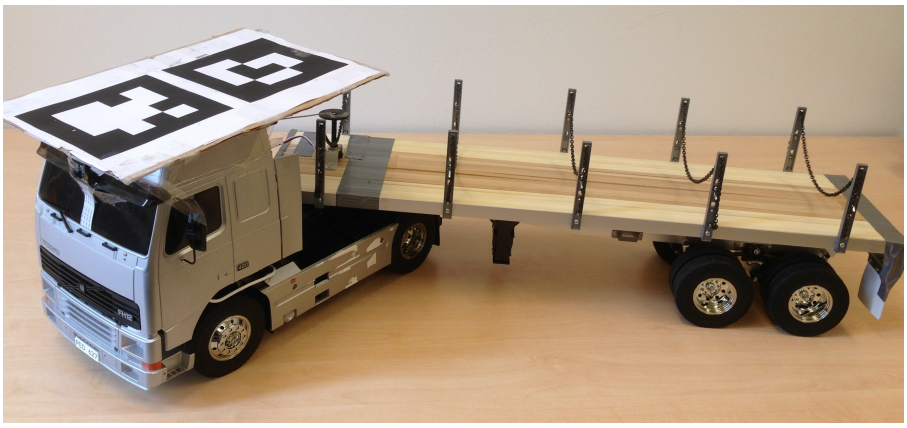


Figure 3.1: The model truck used in this project.

3.1.1 Materials used

In addition to the computer it is also equipped with an analog to digital converter, a potentiometer and a powerbank. To keep this mounted on the truck, we created a polymer plate using a 3D-printer. It was designed to be mounted using existing screwing holes. On this plate we placed the Raspberry Pi, the analog converter, as

well as servo cables (see Figure 3.2). The truck has two servos, used for steering and shifting, and an Electronic Speed Control (ESC) for the motor. The Raspberry Pi is directly wired to the servos and ESC, and controls them using Pulse Width Modulation (PWM).

3.1.2 Sensors and data gathering

The potentiometer is serving as an angle sensor, used to keep track of where the trailer is, relative to the tractor. In order for the Raspberry Pi to be able to read the value of the sensor, the analog signal needs to be converted to a digital signal. On top of the tractor unit, we have placed two AprilTags to be able to locate it using the GulliView localization system. The use of two tags makes it possible to track the orientation as well as the position.

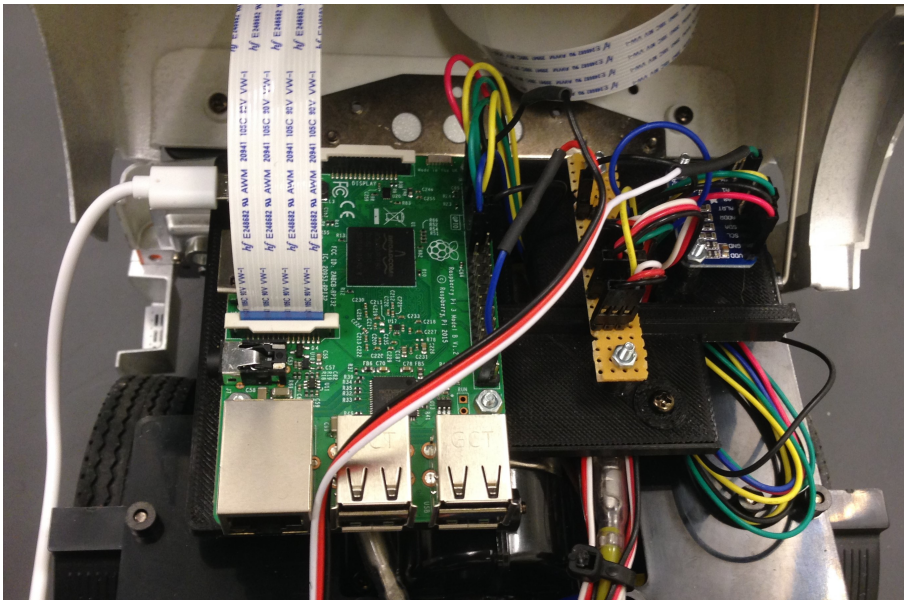


Figure 3.2: Setup of onboard computer, servos, and sensor.

3.1.3 Kinematic model

A mathematical description of the system is needed for the path-planning algorithm to be able to predict the future behaviour of the truck [16]. As the truck will be driving under relatively low speed, a simplified 2D kinematic model is enough to describe the behaviour. Our vehicular system consists of two main sub-components. The tractor unit, which has two front steering wheels and two back rear wheels, and a trailer that has four more back rear wheels and is attached to the tractor at a connection point (x_c, y_c) as presented in Figure 3.3.

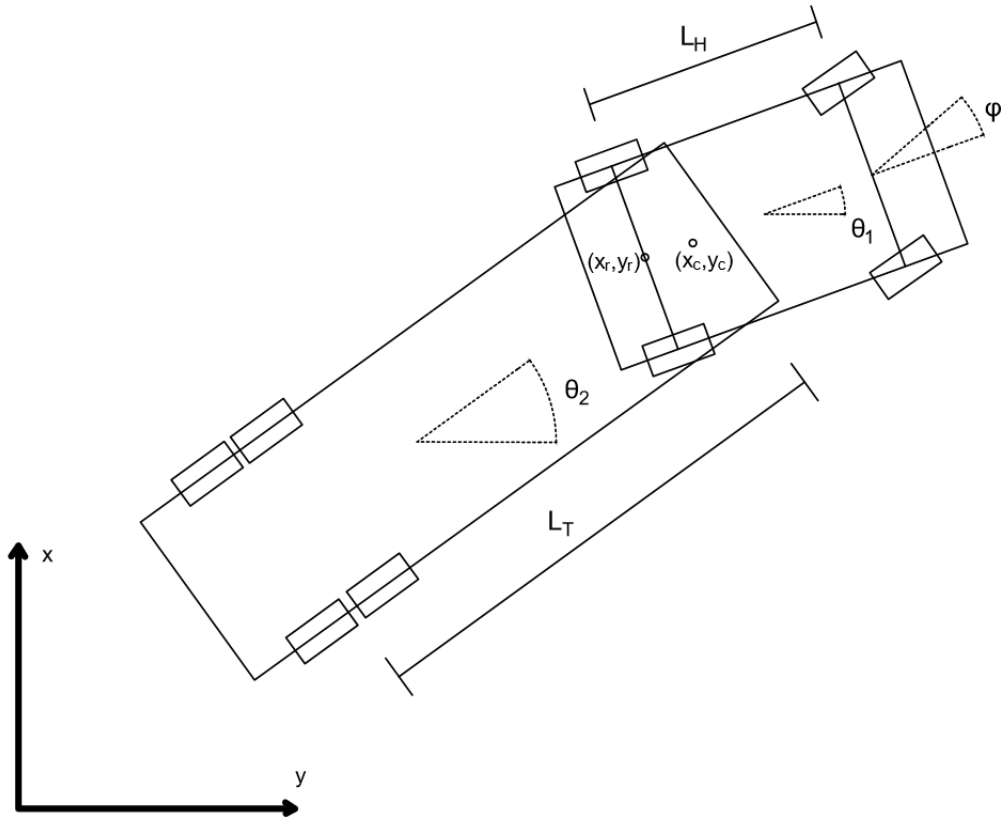


Figure 3.3: Kinematic model of the truck

The point (x_r, y_r) represents the middle of the tractor rear wheels and will be used later on to calculate the centre of rotation. Symbols θ_1 and θ_2 will represent the rotation of the tractor and the trailer respectively and φ will be used to represent the steering angle. The steering angle will be approximated as in the bicycle model [7],[17],[18] by assuming the same steering angle for both tractor front wheels, and translating it to the middle of the line that connects the wheels. L_H and L_T define the distance between front and back wheels of the tractor and the distance between middle of back wheels and connection point (x_c,y_c) on the trailer.

The nonlinear equations that describe the kinematics of the truck are derived as follows. For the tractor, the centre of instant rotation can be calculated by delineating the perpendicular lines to the rear wheels and the steering angle direction, as presented in Figure 3.4. The velocity of the rear wheels of the tractor is denoted by v. Equations for $\dot{x}$ and $\dot{y}$, which represent the first order derivative for x and y, can be easily extracted by projecting the velocity onto the selected axis ($\dot{y} = v \cdot cos(\theta_1)$ and $\dot{x} = v \cdot sin(\theta_1)$). The centre of rotation of the tractor is used to define the angular velocity as follows: $tan(\varphi) = \frac{L_H}{R} \rightarrow R = \frac{L_H}{tan(\varphi)}$ then $\dot{\theta_1} = \frac{v}{R} = v \cdot \frac{tan(\varphi)}{L_H}$.

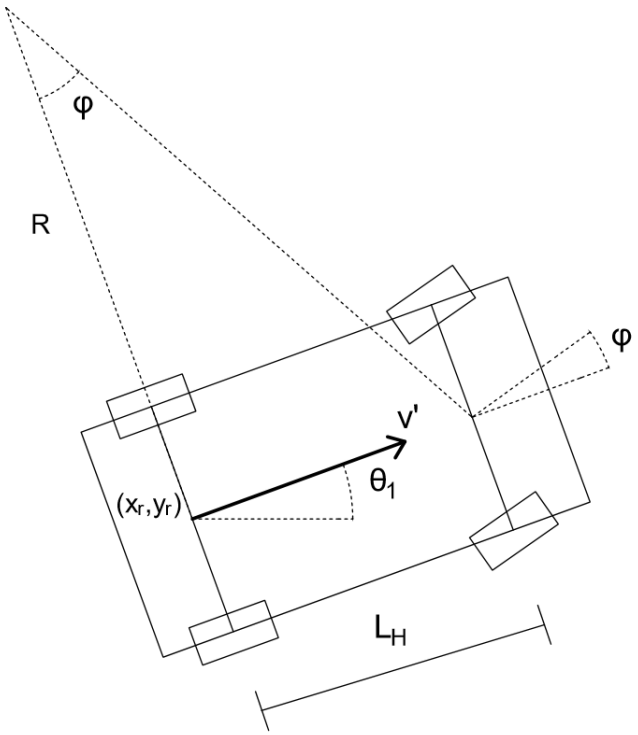


Figure 3.4: Kinematic model of the tractor unit

The equation for $\dot{\theta_2}$ is similarly extracted, after deriving the velocity of the connection point of the trailer, as shown in Figure 3.5. With r, the distance between point (x_r, y_r) and the connection point (x_c, y_c) with the trailer, we can derive the rotating velocity to determine v', which is the total velocity of the connection point. Then the centre of rotation is derived perpendicular to the velocity v' and the back wheels of the trailer.

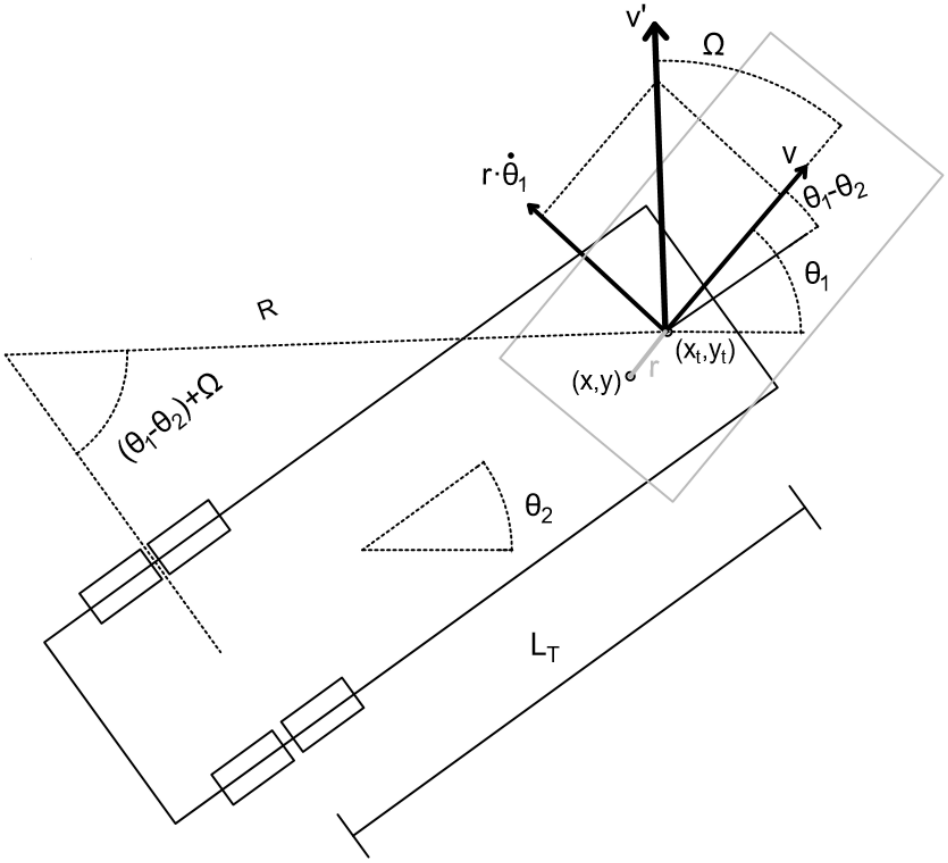


Figure 3.5: Kinematic model of the trailer

The derived centre of rotation is used to define the angular velocity as it follows: first we calculate the radius of rotation, $sin(\theta_1 - \theta_2 + \Omega) = \frac{L_T}{R} \rightarrow R = \frac{L_T}{sin(\theta_1 - \theta_2 + \Omega)}$ to obtain the trailer's connection point velocity $v' = \sqrt{(r\dot{\theta}_1)^2 + v^2}$ to finally obtain $\dot{\theta}_2 = \frac{v'}{R} = \sqrt{(r\dot{\theta}_1)^2 + v^2} \cdot \frac{sin(\theta_1 - \theta_2 + \Omega)}{L_T}$.

Finally, the state vector description for the kinematic model of the truck can be defined as:

$$\begin{cases} \dot{y} = v \cdot cos(\theta_1), \\ \dot{x} = v \cdot sin(\theta_1), \\ \dot{\theta}_1 = v \cdot \frac{tan(\varphi)}{L_H}, \\ \dot{\theta}_2 = \sqrt{(r\dot{\theta}_1)^2 + v^2} \cdot \frac{sin(\theta_1 - \theta_2 + \Omega)}{L_T}. \end{cases} \tag{3.1}$$

3.1.4 Collision model

In order for the truck to recognize when it hits an obstacle, a model for collision detection is needed. In this project, that is done by defining a set of key points on the truck, which are continuously checked for collision to detect when an obstacle is reached. Obstacles that are checked for include both actual obstacles, as well as being outside the bounds of the road.

To check for collisions, the collision points are checked with a map database, where the obstacles are stored. If any of the points are within an obstacle, the truck has collided with that obstacle. The collision model takes a state vector, where the x and y positions are on the back of the trailer. By using the x and y position, tractor angle, and trailer angle, the key points as seen in Figure 3.6 are calculated.

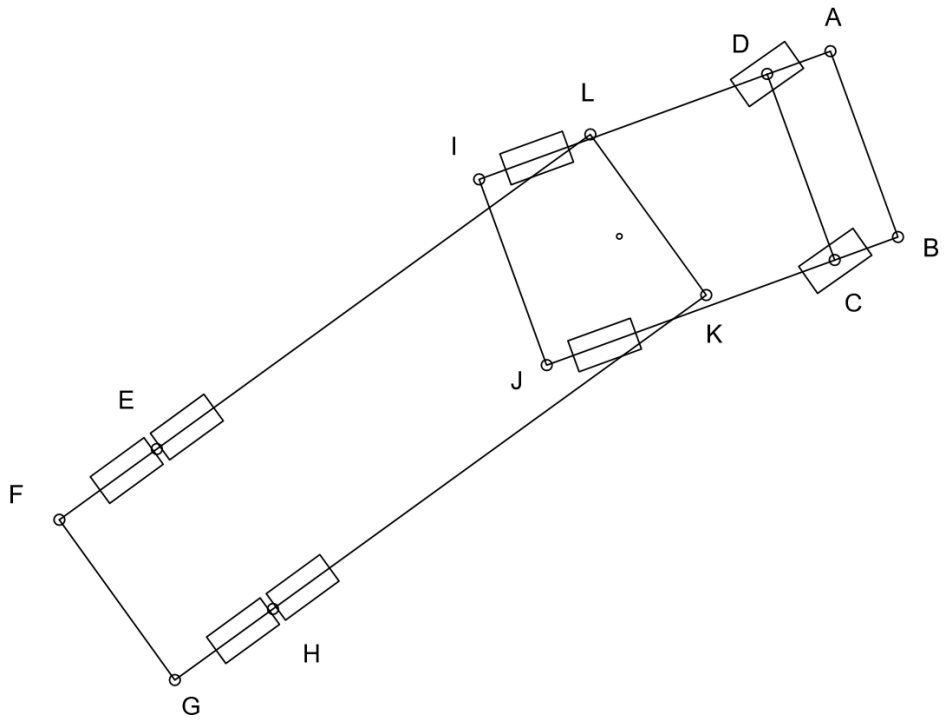


Figure 3.6: Collision model of the truck

3.2 Testbed Environment

The evaluation environment consist of two main parts. The lab which is a physical, real world environment and the simulator which is a synthetic environment.

3.2.1 Physical evaluation environment

The lab that was provided for this project is equipeed with a black mat covering the floor with white tape as road markings, as you can see in Figure 3.7. The track has been carefully created to match our goals with the project.

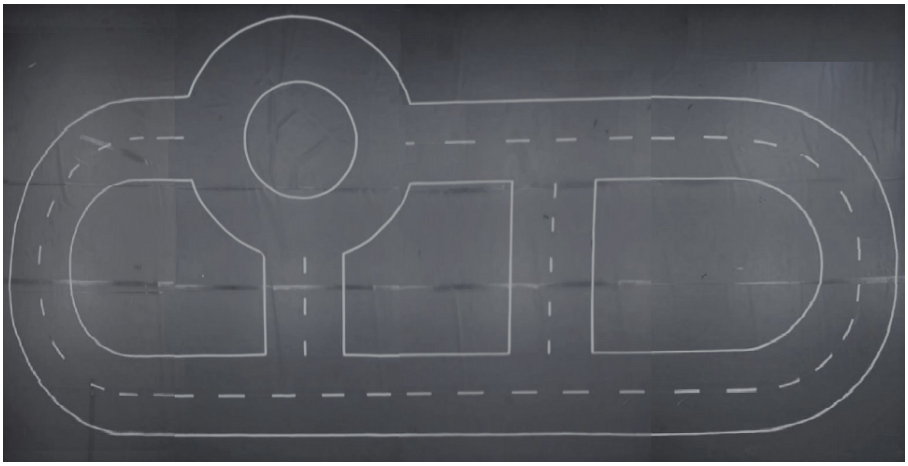


Figure 3.7: Testing track

In the ceiling there are four cameras for the GulliView localization system, as described in Section 2.4. GulliView allows us to get the position of anything eqiupped with an AprilTag. This system was provided to us, but the software as well as the camera settings have been tweaked to fit the project.

3.2.2 Visualization of the system

We are using a third party package called RViz to visualize both the simulations and the physical tests in real time. RViz connects to the ROS network as any other node and enables visualization of robot data in both 2D and 3D.

RViz can be used to visualize the physical vehicle, showing a virtual representation of the whole testbed. The visualizer displays the map, the truck with trailer, and the different paths. Everything is updated in real time so that one can follow exactly what happens in the system.

4. System Architecture

In this chapter, a high-level description of the system architecture, including a brief description of each component, is presented. For further details of the components *Automatic Control*, *Path Planning* and *Map Service*, including ROS communication, refer to Chapter 5.

4.1 Overview

At a high level, the software system consists of a few loosely coupled components, each serving a specific purpose. Some components handle inputs and outputs to the system, some implement algorithms, and some regulate information flow. The components communicate mainly through the ROS framework, and are divided across three different hosts in a shared Wi-Fi network. An overview of the complete system, including external components, can be seen in Figure 4.1.

Hosts:

1. The GulliView Host: A computer placed in the ceiling of the lab room, running the GulliView system.
2. The User Host: Typically a laptop, used to display information and capture user input.
3. The Raspberry Pi embedded on the truck.

System inputs:

1. Image stream from cameras mounted in the ceiling of the lab room.
2. User input from the graphical environment RViz, providing goal and subgoal destinations.
3. User input from a second GUI, used to handle virtual obstacles.

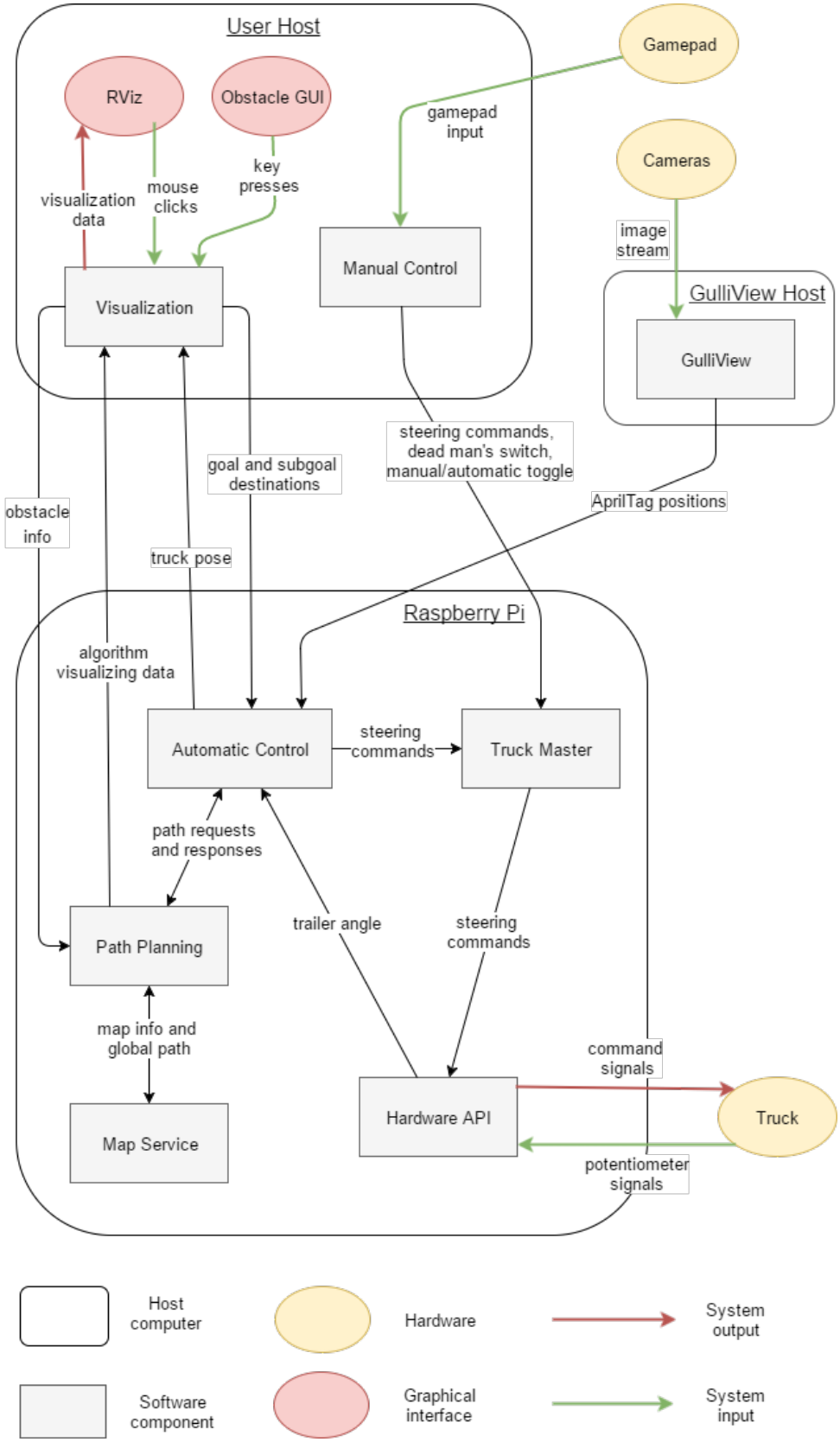


Figure 4.1: A high-level overview of the system architecture. The diagram shows how components are distributed on different hosts and how they communicate with each other and with the hardware, highlighting the inputs and outputs of the system.

4. User input from a gamepad. Used for manual driving, toggling between manual and automatic control, and the fail-safe mechanism for the automatic control.

5. Signals from the potentiometer (trailer angle sensor) mounted on the truck.

System outputs:

1. Command signals for steering angle and speed, sent to the truck hardware.

2. Visualization in RViz, displaying the track, the obstacles, the truck, and the progress of the path planning algorithm.

4.2 Components

This section contains a short description of the purpose and tasks of each component.

GulliView

Continuously reads the image stream from the ceiling-mounted cameras and recognizes the AprilTags on top of the truck. The tag positions are translated to global coordinates and sent to *Automatic Control*.

Automatic Control

Central component that handles the automatic control of the vehicle. With the current truck position as a starting point s, and given goal and subgoal destinations g and $\vec{sg}$, a valid path from s to g, sequentially visiting each position in $\vec{sg}$, is requested from the *Path Planning* component.

When a path has been provided, the distance from the truck to this path is continuously calculated. The distance, or error, is fed into a PID controller which outputs a steering angle to compensate for the deviation from the path. This steering angle is, along with a constant speed value, passed down to the *Truck Master* component.

Visualization

Redirects ROS messages to and from RViz. This includes:

1. Capturing user clicks for goal and subgoal destinations, and sending these to *Automatic Control* (see Figure 4.2)

2. Gathering data from *Automatic Control* and *Path Planning*, and transforming it into the specific message types required for visualization in RViz (see Figure 4.3).

Also provides a GUI used to add virtual obstacles to the track (see Appendix D). When an obstacle is activated, information is sent to *Path Planning* so that the algorithm can adapt and avoid the obstacle.

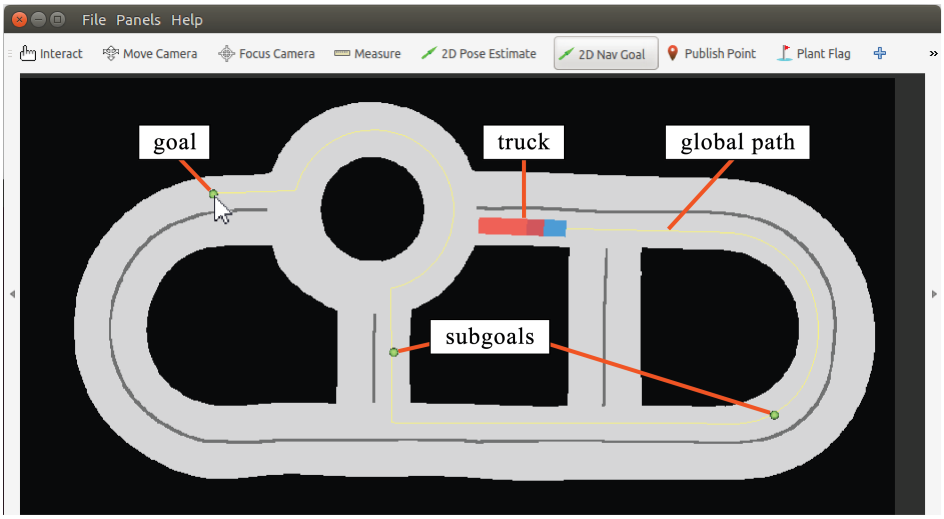


Figure 4.2: Gathering goal and subgoal destinations in RViz by mouse click. The resulting global path shows how each subgoal is sequentially visited.

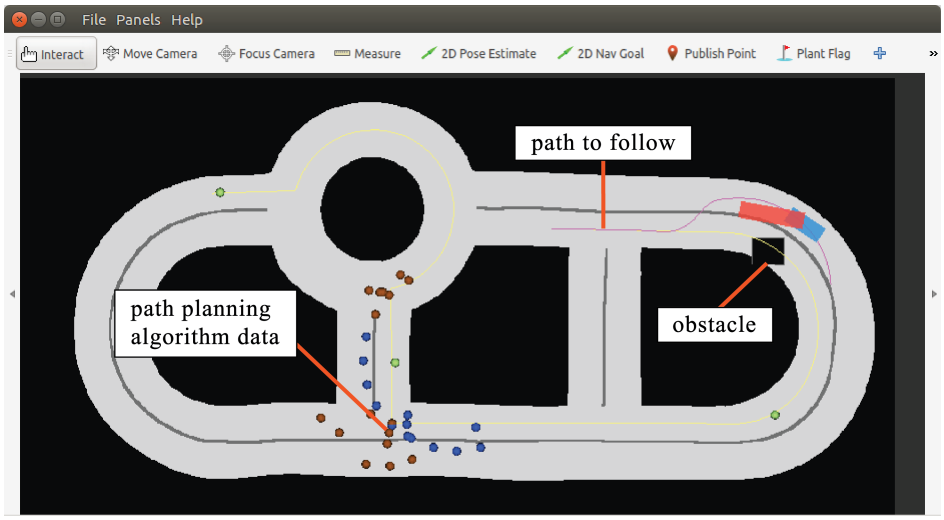


Figure 4.3: Displaying the output from the path planning in RViz. Notice how the added obstacle is avoided, and how a small finished section of the path is already in use, even though the path planning is still calculating.

Path Planning

Contains the implementation of our own dynamic path planning algorithm. After receiving a path request from *Automatic Control*, a map and a global shortest path is fetched from *Map Service*. Next, the algorithm is executed and the resulting path is returned to *Automatic Control* in small sections (see Figure 4.3).

Also handles the dynamic obstacle avoidance. When an obstacle appears, the algorithm adapts on the fly and recalculates the path.

Map Service

Stores the map as a binary image, where white and black pixels represent allowed and forbidden areas of the track. Also provides a representation of the track as a directed graph, with the edges marking the middle of the lane (See Figure 5.5 on page 29). The graph is used in the implementation of a *k-shortest-paths* algorithm, to compute a global path from given start, goal and subgoal locations.

Manual Control

Captures user input from a gamepad, typically a wired Xbox 360 controller. Handles manual driving, toggling between automatic and manual control, and a fail-safe mechanism for the automatic control. Gamepad input is mapped to speed and steering angle commands, and sent to *Truck Master*. One button acts as a dead man's switch, which always needs to be pressed in order for the truck to move, stopping it immediately upon release. A control scheme for the gamepad can be seen in Appendix E.

Truck Master

Regulates steering commands to the *Hardware API*, choosing between automatic and manual control.

Hardware API

Receives speed and steering angle commands from *Truck Master* and translates these into command signals for the steering servo and the electrical speed controller. It also continuously reads the potentiometer signals and translates the output voltage to an angle, representing the direction of the trailer relative to the tractor unit.

Truck Simulator

The loose coupling between components allows us to replace *GulliView* and the *Hardware API* with a simulator component, serving the same purpose (see Figure 4.4).

The simulator stores the current state of the truck, and uses the kinematic model (see Section 3.1.3) to discretely calculate vehicle movements. Given speed and steering angle commands, the next vehicle state is calculated and sent back to *Automatic Control*.

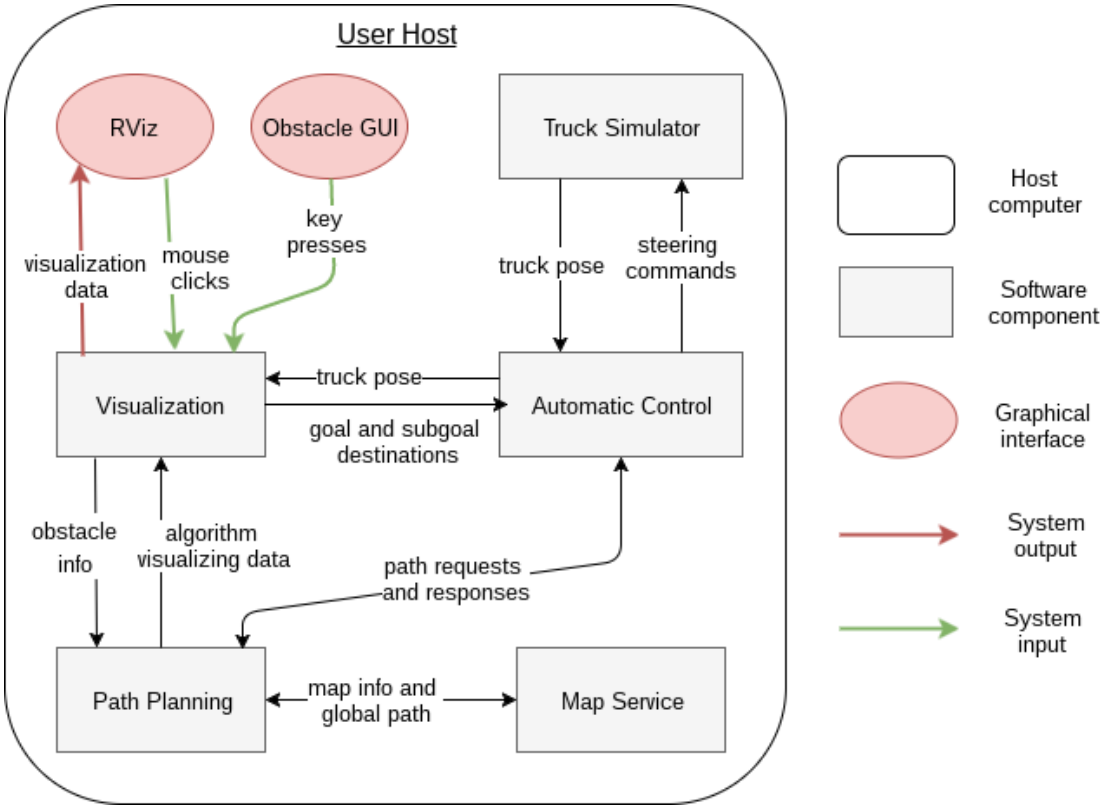


Figure 4.4: An overview of the system architecture when using the truck simulator.

5. Algorithms and Implementation

This chapter contains the selected strategy to reach the automation goal. First, a path-planning strategy was developed, to search in advance for a route that ensures that both the tractor and the trailer stays within the bounds of the road. Then, a PID controller for the tractor was implemented, to achieve trajectory following by estimating the deviation from the desired path and translating it into a steering command. Finally, the two ROS nodes containing the mentioned algorithms and their communication are explained, as well as the map module that provides the global reference path.

5.1 Path Planning

Path planning is a well known and general problem, which has generated many different algorithms, that are good in different scenarios. Most algorithms have in common that they use a graph of possible discrete states of the vehicle and do a search of the graph to either find a path or a shortest path from $\vec{s}_{start}$, which is the start discrete state, to P_{goal}, which is the navigation goal. A mathematical model of the vehicle is then used to find new discrete states to use in the graph.

In this section the following definitions will be used:

- $\vec{s} = (x, y, \theta_1, \theta_2)$: A discrete state of the truck which is defined by its position $(x, y) \in \mathbb{R}^2$, its global tractor angle $\theta_1 \in [0, 2\pi)$, and its global trailer angle $\theta_2 \in [0, 2\pi)$.

- C_{free}: The free space, a set of all discrete states that avoids collision with forbidden areas.

- $\vec{s}_{start}$: The start discrete state, $\vec{s}_{start} \in C_{free}$.

- P_{goal}: The position that is chosen as the navigation goal, consists of a position $(x, y) \in \mathbb{R}^2$.

The model used in our solution is the kinematic model which takes a state $\vec{s}_1$ and some driving command as input, and together with a step size, gives a new state $\vec{s}_2$ as output. If we want all possible states $C_{\vec{s}_1} \subset C_{free}$ reachable from state $\vec{s}_1$, we need to give all possible steering commands where the output state of the model belongs to the set C_{free}.

This however leads to a very big graph if we want all possible states for the truck. Consider that we would have to visit all discrete states in $C_{\vec{s}_1}$ and give all possible steering commands to each of those states. Then do the same for all new states we find and so on until we find no new states. To avoid long execution times caused by large graphs, a heuristic algorithm is used. The heuristic algorithm trades solution accuracy for performance.

5.1.1 Heuristics

The complexity of the path planning problem and the fact that the software will run on a Raspberry Pi gives a constraint on how fast the algorithm needs to run. Trying all possible solutions is not fast enough. Therefore, a heuristic algorithm is needed and the best possible path can not be guaranteed. A grid based search and limited steering commands are the heuristics used in our algorithm. By adding these heuristics the amount of nodes the algorithm needs to visit is limited. These heuristics can also be tuned using different parameters.

Grid search

The amount of nodes in the graph is limited by a grid. Each grid consists of a state $\vec{s} = (x, y, \theta_1, \theta_2)$ where the grid of a state can be computed by calling a rounding function, which can be found in Appendix F. If the grid of a state is already visited we do not visit that state. This is done to not visit similar nodes multiple times. All parameters in $\vec{s}$ are used because even if just one parameter, say the trailer angle θ_2, is different while the other parameters are similar, it will lead to completely different new states from $\vec{s}$. However, if all parameters are similar, the new states from $\vec{s}$ would be similar.

Limited steering commands

The algorithm always uses the same step length and five different steering angles to find the new states from $\vec{s}_{current}$ when using the kinematic model. The algorithm only uses five angles to limit the amount of nodes in the graph.

Since we use few steering commands we have to choose those in a clever way that can lead to a good trajectory for the truck. The steering angles used are: $\varphi_{straight}$, $\varphi_{max-left}$, $\varphi_{max-right}$, $\varphi_{middle-lane}$, and $\varphi_{outer-lane}$. Where $\varphi_{middle-lane}$ is the steering angle that takes $\vec{s}_{new}$'s x and y position to be in the middle of the road and $\varphi_{outer-lane}$ is the steering angle that takes $\vec{s}_{new}$'s x and y position to be in the outside of a turn.

To find $\varphi_{middle-lane}$ a binary search is used. Since the middle of the lane is stored in a map we reapply the kinematic model with different steering commands until the distance to it is less than some ϵ, which is set to 0.1 cm. Same strategy is used to

find $\varphi_{outer-lane}$. By using the map we know if there is a left or right turn. The width of the lane and the truck width is also known. The distance d from the middle of the road is: $d = lanewidth - \frac{truckwidth}{2}$. The binary search finds a steering angle towards the outside of the turn, in the trucks lane, d distance from the middle of the lane. A visualization of the angles and position of the middle lane and outer lane can be found in Figure 5.2

5.1.2 The algorithm

The algorithm designed by the group consists of two layers which both use the heuristics explained above. The first layer computes a path from $\vec{s}_{start}$ to P_{goal} as fast as possible and uses the cost of that path, calculated with the cost function, as an upper bound for the second layer.

The second layer of the algorithm searches for all possible solutions. It uses the upper bound to limit the search and avoid unnecessary solutions that are worse than the best path found this far.

A cost function is used to make a path measurable and is implemented by measuring every discrete state that makes the path. By applying a cost to each state and calculating the sum of those costs we get the total cost of the path. The cost for each state is calculated using the two front wheels of the tractor unit and the two back wheels of the trailer. The distance from the optimal position of the trucks wheels in reference to the middle of the road will then be calculated and used as error which is shown in Figure 5.1. If any wheel is in the opposite lane we increase the error of that wheel by multiplying it with some weight, which in our case is set to 10. The total cost for the state is then calculated by taking the sum of the errors for all four wheels.

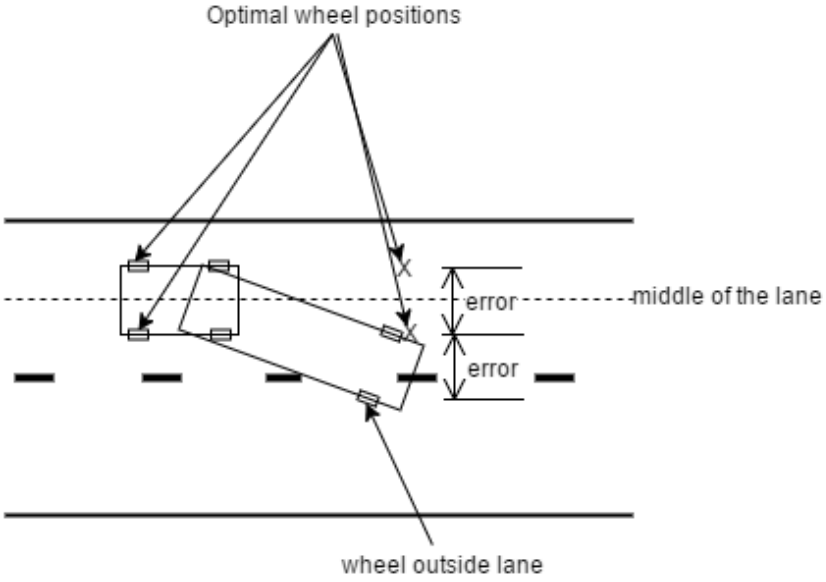


Figure 5.1: Cost function for a discrete state

Three main data structures are used in the first layer of the algorithm. A stack of discrete states where every element is in C_{free} that the algorithm is going to visit, A set *visited* of grids of states that has been visited, and a key value pair. Given a state as key, the state that lead to that state is given as value. Below comes the pseudo-code for the first layer of the algorithm:

```
1  addPossiblePaths(s⃗_start);
2  while stack not empty do
3  |  while True do
4  |  |  node ← stack.pop();
5  |  |  grid ← round(node);
6  |  |  if grid ∉ visited then
7  |  |  |  break;
   |  |  end
   |  end
8  |  if dist(node, P_goal) < ϵ then
9  |  |  return gatherPath(node);
   |  else
10 |  |  if left of middle-lane then
11 |  |  |  addPossiblePathsLeft(node);
   |  |  else
12 |  |  |  addPossiblePathsRight(node);
   |  |  end
13 |  |  grid ← round(node);
14 |  |  visited.insert(grid);
   |  end
   end
```

Algorithm 1: Layer one of the algorithm

`'addPossiblePaths'` is a method that takes a state $\vec{s}_{in}$ as input and appends new states to the stack. The new states from $\vec{s}_{in}$ are calculated using the kinematic model and the steering angles defined above. A state $\vec{s}_{new}$ calculated from $\vec{s}_{in}$ will only be added to the stack if travelling from $\vec{s}_{in}$ to $\vec{s}_{new}$ stays in the free space the whole way. This is checked using the collision model. The different new states are added in a specific order to visit the nodes that are most likely to lead to the goal first. The order of the stack from top to bottom after the method call:

`'addPossiblePathsLeft'`	`'addPossiblePathsRight'`
$\vec{s}_{middle-lane}$	$\vec{s}_{middle-lane}$
$\vec{s}_{outer-lane}$	$\vec{s}_{outer-lane}$
$\vec{s}_{max-right}$	$\vec{s}_{max-left}$
$\vec{s}_{max-left}$	$\vec{s}_{max-right}$
$\vec{s}_{straight}$	$\vec{s}_{straight}$
previous nodes	previous nodes

Order of the stack after call to `'addPossiblePaths'`

`'addPossiblePaths'` also adds $\vec{s}_{in}$ as a parent node to the new states that are added to the stack. The new states are added as keys with the parent node $\vec{s}_{in}$ as value in the key-value data structure.

The key-value data structure is then used in the gatherPath method. In this method the parent nodes are used to backtrack to $\vec{s}_{start}$. By backtracking and sampling every state, a path from the start position to the given node is achieved. A visualization of the algorithm can be seen in Figure 5.2.

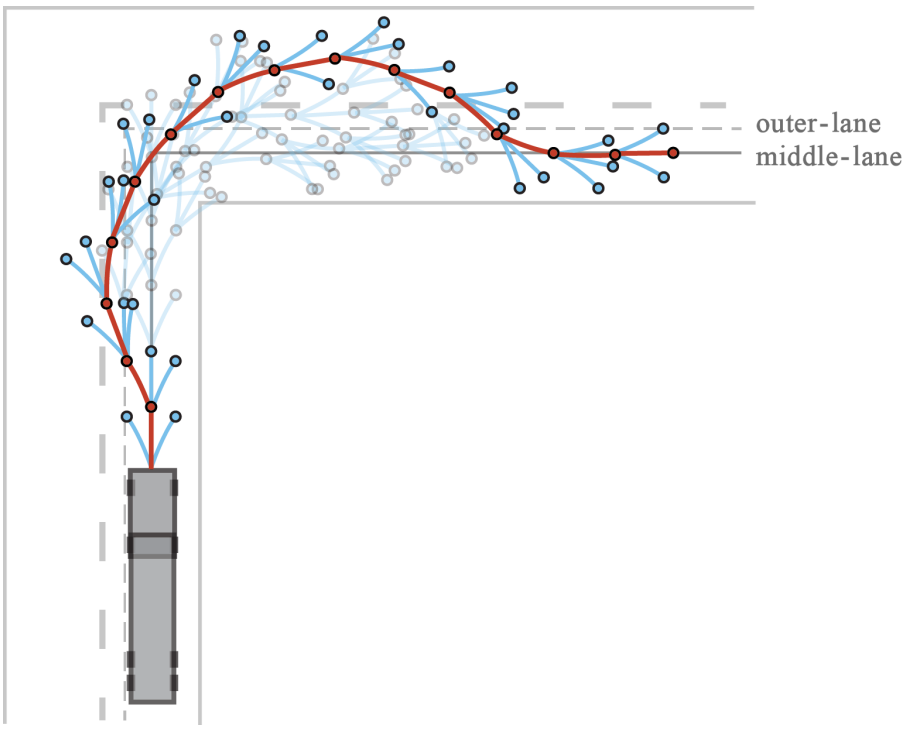


Figure 5.2: Visualization of the algorithm finding a path. The red path is the path from $\vec{s}_{start}$ to P_{goal}. The transparent nodes and edges are nodes that did not contribute to the solution, and the solid ones contributed to the solution. A lot of none-contributing nodes have been skipped in the picture for clearer visualization.

After the first layer has found a path the cost function is used on that path. The cost is used as a start upper bound for the second layer and the path is saved as the best path found this far. The second layer of the algorithm is similar to the first layer. It uses the same three data structures as well as another key-value data structure, that stores a grid as key and the lowest error for the grid as value.

This layer doesn't stop the main loop when a solution is found, instead it calculates the cost of the solution. If the cost is lower than the currently lowest cost it gets updated and the path is saved. The next node on the stack is then visited, and the algorithm only terminates when there are no more nodes to visit. Hence line 8 and 9 are replaced with:

if *dist(node, P_{goal}) < ϵ* ***and*** *gatherCost(node) < lowestCost* **then**
    lowestCost ← gatherCost(node);
    cheapestPath ← gatherPath(node);
end

Another difference is that when a node is being visited the total error it took to reach that node is added to the new error data structure.

When to visit nodes is the last difference between the layers. In layer one a node $\vec{s}$ in the stack will be visited if $\vec{s} \in C_{free}$ and the grid of $\vec{s} \notin visited$. However in layer two $\vec{s}$ will be revisited if the current path to the grid of $\vec{s}$ is cheaper than the previous visit to that grid. The second layer also doesn't visit nodes which total cost is higher than the currently cheapest solution (the upper bound). Line 6 and 7 is changed to:

if *gatherCost(node) < lowestCost* **then**
    if *grid ∉ visited* ***or*** *getCost(grid) < getLowestError(grid)* **then**
        break;
    end
end

The final result of the algorithm and the path that is returned is the cheapest solution found this far, as a list of discrete states $\vec{s}_{path} \subseteq C_{free}$ that does not collide into any obstacles while travelling between the states.

5.2 Feedback Control Loop

The control system can be represented as the feedback control loop shown in Figure 5.3, where the input to the control loop is a reference path. This section covers how this control loop works.

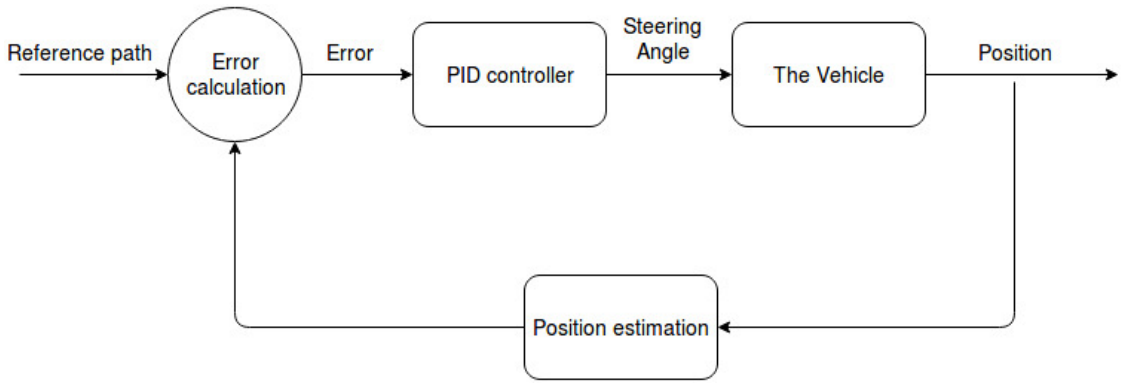


Figure 5.3: The control loop

The reference path is a set of positions represented as dots connected with line segments, as shown in the right-hand side of Figure 5.4. The error from the path is calculated as the perpendicular distance from a point p, to some line segment l. As the path consists of many line segments, sometimes overlapping, determining which of these to use becomes a problem. The solution is to, with each position update, traverse the path and remove line segments already passed. The first line segment of the remaining path can then be used for the error calculation.

The left part of Figure 5.4 displays how a look-ahead point, $P_{lookahead}$, is used. This is the position that the error calculation compares with the reference path. The reason for this is to be able to respond to changes in the trajectory a bit earlier to prevent a large error when the path turns.

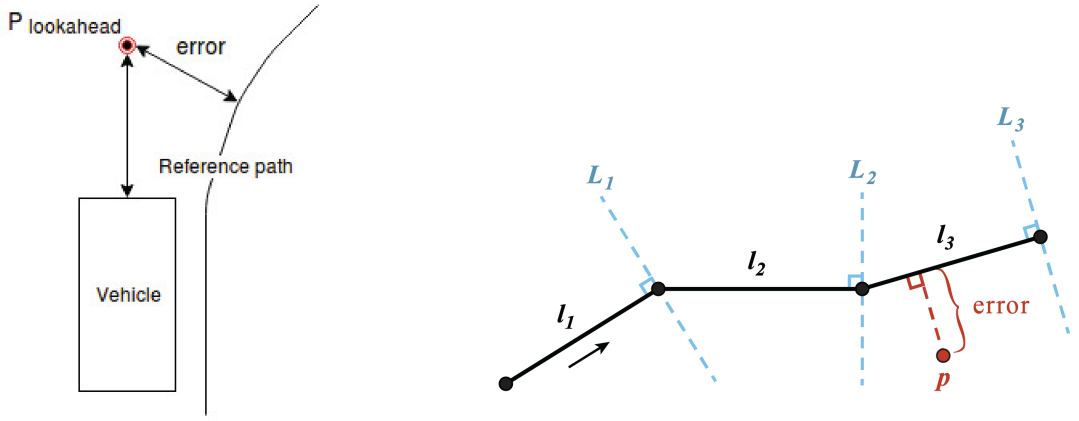


Figure 5.4: To the left the error from the look-ahead point is displayed. To the right the path traversal is shown.

The controller works in discrete time and receives the current calculated error as input. The output is a steering command to the hardware API with the purpose to correct the error from the reference path. The steering command consists of a P, I and D term. The control signal (steering angle) that the hardware API receives is calculated by

$$U = Kp \cdot e_{current} + Ki \cdot e_{tot} + Kd \cdot \Delta e / \Delta t$$

where Kp, Ki and Kd are adjustable parameters, $e_{current}$ is the input and the rest is calculated. The time difference since the last iteration is determined by

$$\Delta t = T - T_{last}$$

where T is the current timestamp and T_{last} is the timestamp of the last iteration.

The total error since the start

$$e_{tot} = e_{totprev} + e_{current}$$

where $e_{totprev}$ is the previous total error. The error difference since last iteration

$$\Delta e = e_{current} - e_{last}$$

where $e_{current}$ is the current error and e_{last} is the error of the last iteration. The speed input control signal is constant and determines the new position of the vehicle along with the steering angle control signal. To prevent e_{tot} to run away to extreme numbers, for example if the vehicle stands still with an error, an integral anti windup guard I_{aw} is used. If e_{tot} sums up to larger than $\pm I_{aw}$, it is set to $\pm I_{aw}$.

5.3 Map and Global Path

The map module provides the path planner with an up-to-date map of the track and a global path which functions as a starting point for the path-search. The map is stored as a binary image, where white and black pixels correspond to allowed and forbidden areas of the track. During run time, the map is represented by a two dimensional array of 1:s and 0:s. Internally, the module contains a representation of the track as a directed graph, with the edges marking the middle of the lane.

5.3.1 The graph

The graph is a cornerstone in the implementation of a *k-shortest-paths* algorithm, used to compute a global path between two points on the track. It is designed so that it always gives a valid path if one exists, with special care taken in the T-crossings. Since the turning radius of the truck is too large for it to manage U-turns, the nodes are connected in a way that does not allow them. For a visualization of the graph, see Figure 5.5.

5.3.2 Computing a global path

When requesting a global path, the current state of the vehicle and a list of desired goal and sub-goal positions are given as parameters. The first step in constructing the path is to find a start node. The start node should be as close as possible to the current position of the vehicle, and have an out-edge in the same direction. After that, each of the given goal and sub-goal positions are paired with the closest node. The final step is to compute the shortest path between each pair of nodes, and then concatenate all the subpaths.

All resulting paths are feasible, but can still prove too difficult to manage if the truck is positioned in an unfavorable way. If any of the subpaths are impossible to follow, the path planner can request an alternative path for those sections. We then compute all possible loop-less paths between the affected nodes, and try each of them until a working path is found, or there are no more alternatives.

To find these alternative paths, we use a generalised version of Dijkstra's algorithm [19], as shown in Appendix G. Instead of just returning the shortest path between two nodes, the algorithm is extended to find the k shortest loop-less paths. By choosing a high enough value for k, we can assure that we have exhausted all possibilities.

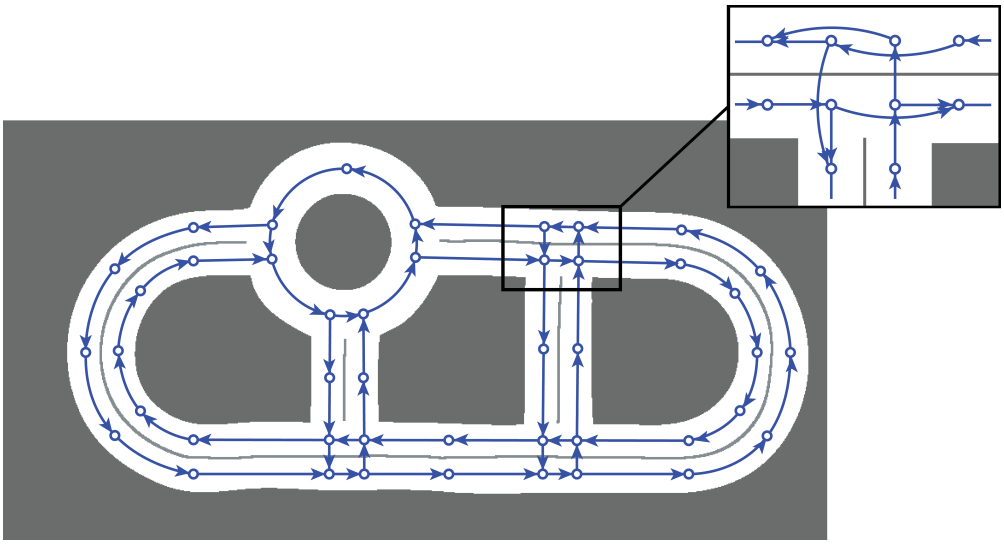


Figure 5.5: A simplified version of the directed graph used to compute a global path. The zoomed in section shows how the nodes are connected in the T-crossings, to achieve desired behaviour.

5.4 Path Following, Dynamic Path Planning and ROS Integration

This section describes two central parts of the software system, the ROS-nodes *AutoMaster* and *PathPlanningNode*. It is explained how they communicate with each other and how they integrate and use the algorithms described in this chapter. Furthermore, the dynamic path planning with the real-time obstacle avoidance is explained.

An overview of this part of the system can be seen in Figure 5.6, showing the communication interface between *AutoMaster* and *PathPlanningNode*, and how the non-ROS classes are integrated into the system.

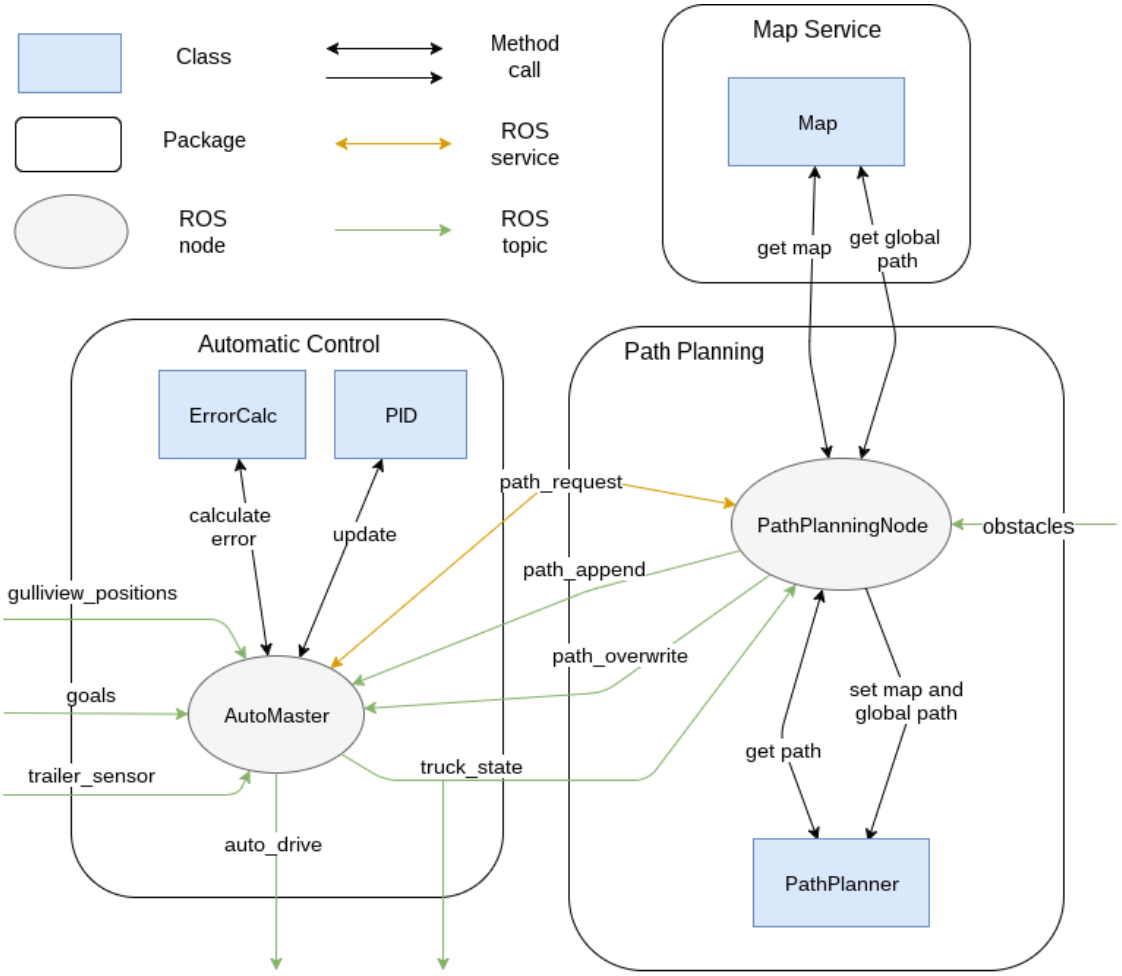


Figure 5.6: An overview of the internal and external communications of the components Automatic Control, Path Planning and Map.

5.4.1 The automatic control node

This node, named *AutoMaster*, requests and receives paths from *PathPlanningNode* and, based on continuous position updates from GulliView, makes sure that the truck follows these paths as closely as possible.

Requesting and receiving paths

The list representing the path in AutoMaster can be both appended and overwritten, and is done so by publishing data on the topics `'path_append'` and `'path_overwrite'`. Both of these topics use the `Path` message type, which is simply a list of (x,y)-coordinates.

When a list of goal destinations is received from Visualization, the `path_request` service, implemented in PathPlanningNode, is called. The service takes a starting position and a list of goals as input, and tries to find a global path that visits each goal sequentially. If no such path is found, the request fails. Otherwise, the request succeeds, indicating that PathPlanningNode has initialized the path planning, and that it will soon start publishing paths on `'path_append'`.

Translating tag positions to driving commands

With each tag position update, a look-ahead point is calculated and the path stored in AutoMaster is traversed. A drive command is then prepared, consisting of a speed value s and a steering angle φ.

If the path is empty, the goal has been reached and s and φ are both set to 0. Otherwise, the error from the path is calculated, and fed into the PID-controller's update function. The returned value is assigned to φ, and some constant value is assigned to s. The drive command is then published on the topic `'auto_drive'`.

AutoMaster also receives trailer angle updates, and together with the tag positions, a complete state vector $\vec{s} = (x, y, \theta_1, \theta_2)$ can be put together. The same state vector is used in the kinematic model, and is described in section 3.1.3. With each position update, a new state is compiled, and published on the topic `'truck_state'`.

5.4.2 The path planning node

This node coordinates how the path planning algorithm is utilized. It handles the on-the-fly path planning, the recalculation of global paths, as well as the dynamic avoidance of virtual obstacles.

On the fly path planning

The path planning is, by far, the most computationally expensive part of the system. As we want to minimize the time the truck stands still and waits for directions, the path is computed and sent to AutoMaster in smaller sections. This allows the truck to start following the early parts of the path, while the latter ones are still being calculated.

A problem becomes selecting start and end points for different subsections. Simply dividing the path into sections of equal length, and computing each section separately will not work, as some start and endpoints might end up just before a sharp turn or an obstacle. At such a point, it might be too late to adapt to the difficult circumstances ahead, since the truck would've needed to start turning during the previous section. (See Figure 5.7)

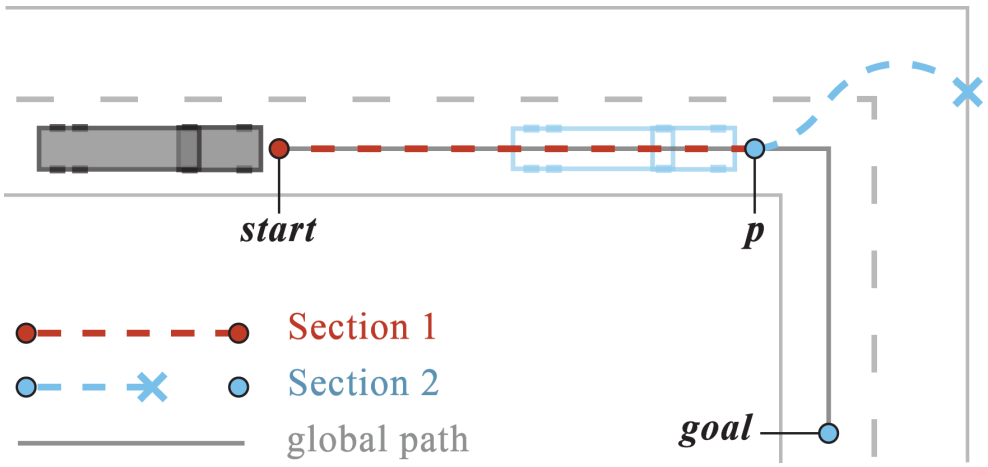


Figure 5.7: Poor choice of start/end point. If the point p is selected as the endpoint of Section 1 and the start point of Section 2, the truck won't have time to adapt to the sharp turn.

The following solution is used to solve this problem: For the first section, a path P_1 with some length l is calculated along the global path. Then, some point p along P_1 is chosen as the starting point for the next section. In doing so, the end part of P_1 is cut off, and discarded. The cut-off point is chosen a fixed distance d from the end of P_1. This procedure is illustrated in Figure 5.8.

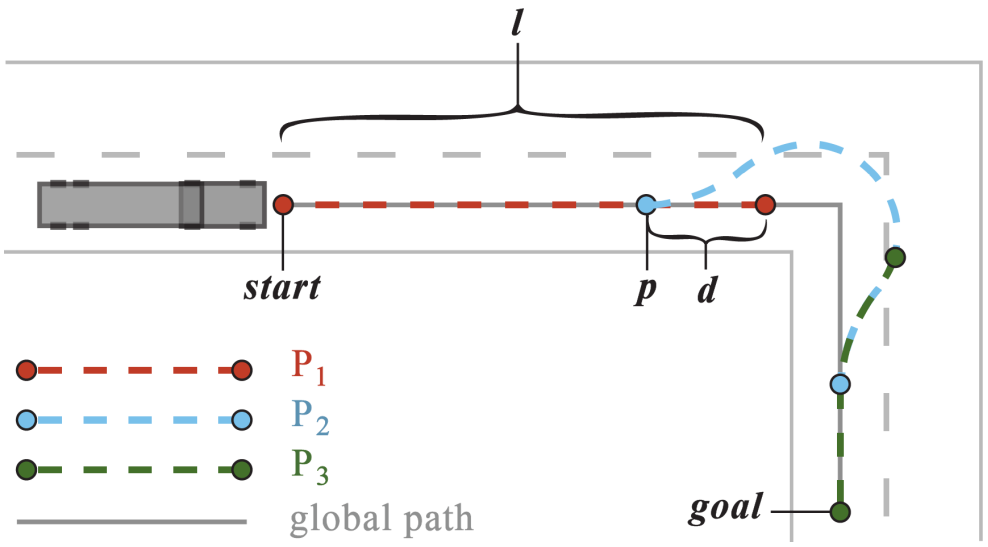


Figure 5.8: A solution to the problem demonstrated in Figure 5.7. Overlapping the different sections allows the truck to adapt to the turn in time.

Recalculating the global path

Sometimes, the global path turns out to be impossible to follow. Maybe there is a turn that's too difficult to take, or there is an obstacle blocking the way. In either case, another global path needs to be found. This is done by choosing the second shortest path instead of the shortest one. (See Fig 5.9) If this path doesn't work either, the third shortest path is used, and so on. In the end, we either find a feasible path or exhaust all possibilities.

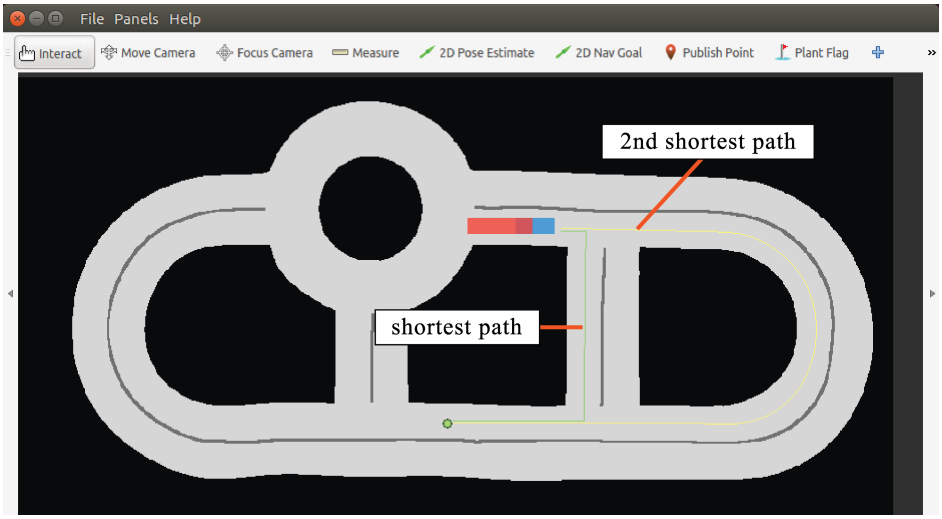


Figure 5.9: The shortest path is impossible to follow so the second shortest path is used instead.

Dynamic obstacle avoidance

When a virtual obstacle is added to the track, the map is updated and the path P that the truck is currently following, is examined. Continuing on P might lead to a collision with the newly added obstacle. If a collision point q is found, P is cut off some distance d before q, and the remaining path P' is sent to AutoMaster on the `'over_write'` topic, overwriting the old path. The cut-off point c is chosen as the new starting point, and the path can be recalculated. (See Figure 5.10)

If the length of P is less than d, the current truck position, received on topic `'truck_state'`, is chosen as the new starting point, and an empty path is published on `'over_write'`.

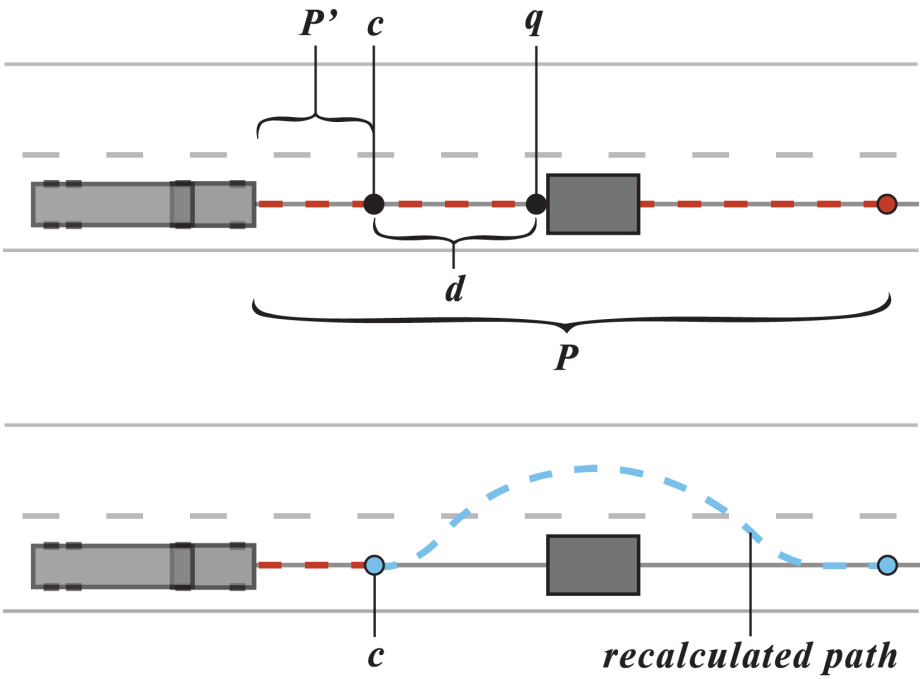


Figure 5.10: Cutting off and recalculating a part of the path, to avoid an obstacle. Notice how the truck doesn't have to stop, as it can still follow P' while the new path is being calculated.

6. Evaluation and Results

In this section the results from the different tests are presented and the error sources are briefly discussed. The performed tests include: the PID controller performance on path following, how accurately the derived kinematic model represents the model truck, and the performance and correctness of the path planning algorithm. The tests have been carefully designed to showcase that the system can handle difficult scenarios.

6.1 PID Controller

For the vehicle to stay on the road, it is crucial that it is able to follow the reference path without a large error. The parameters of the PID controller has been tuned strategically to satisfy this need. In order to validate the performance of the PID controller, a few steps of evaluation has to be made on the prototype. With a given path through the track, we study how well the PID controller is able keep the error to a minimum.

In Figure 6.1 three test runs are plotted which shows how it follows the reference path with the speed input set to 0.35 m/s. The reference path runs through the hardest parts of the track including for example a sharp right turn in the T-crossing followed by a left turn in the roundabout. There is a very small difference between the laps and the vehicle follows the reference path well.

A more precise view of the tests are shown in Table 6.1. The different results we have decided to study are the maximum absolute value of the error, $|e_{max}|$, the average absolute value of the error, $|e_{avg}|$, the mean value of the error, $\overline{e}$, and the percentage of the time the error is below 1 (t_{1cm}), 3 (t_{3cm}) and 5 (t_{5cm}) centimeters respectively. The results show a consistency in the controller even if there is a small difference between the test runs. This is the result of using a real model vehicle where physical disturbances always are present.

In this case, there is also an issue with the positioning system when the vehicle is between two cameras. This leads to a large maximum error shown in Figure 6.2, a small jump in position (like a step response), where the PID controller quickly compensates to a negative error. This results in an unwanted behaviour during a few seconds.

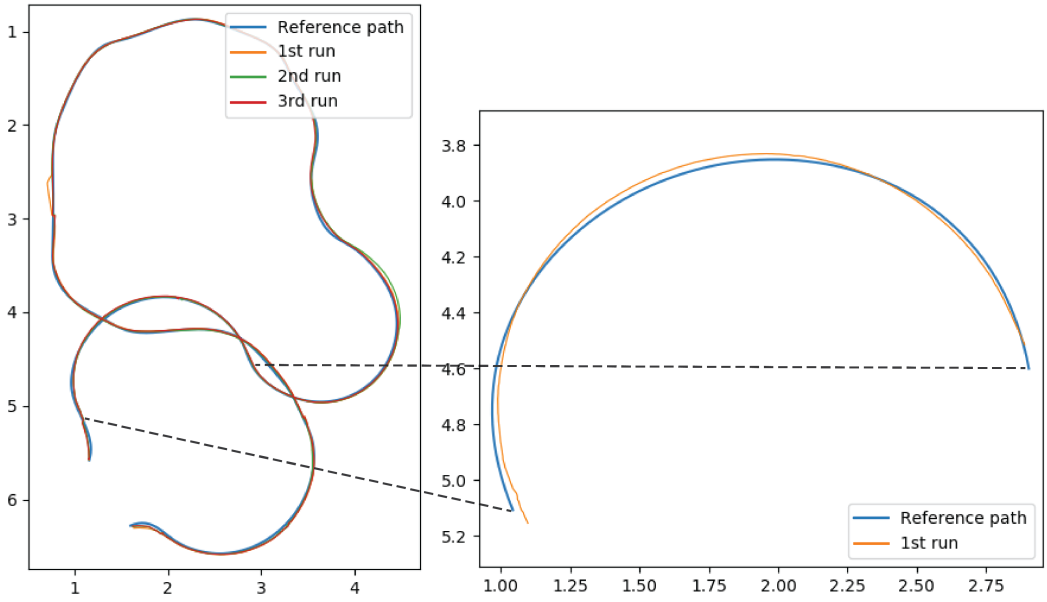


Figure 6.1: The left plot contains the three test run and the reference path. The right plot shows a zoomed in view on a part of the trajectory for the first test run. Both plots have coordinates in meters on the axes.

Test run	$\lvert e_{max} \rvert [mm]$	$\lvert e_{avg} \rvert []$	$\overline{e} [mm]$	$t_{1cm} [\%]$	$t_{3cm} [\%]$	$t_{5cm} [\%]$
1	62.3	13.1	-0.4	94	97	99
2	53.3	12.9	2.7	88	93	98
3	40.1	11.7	-3.2	96	99	100

Table 6.1: Test results for the PID controller

The percentage of the time the error is below 5 centimeters, t_{5cm}, is above 98 % for all the test runs which is a good result in order to meet evaluation criteria. Furthermore, the results in Table 6.1 show that there is a small variation between the test runs which is desired. This shows that the controller is robust against uncertainties and disturbances in the model. The average absolute value of the error for the test runs are all less than 15 mm. The mean values of the error, $\overline{e}$, are all close to 0 . This means that there is almost no offset error after a long run. Worth to mention here is that there are additional uncertainties in the positioning system which could have an effect on the results.

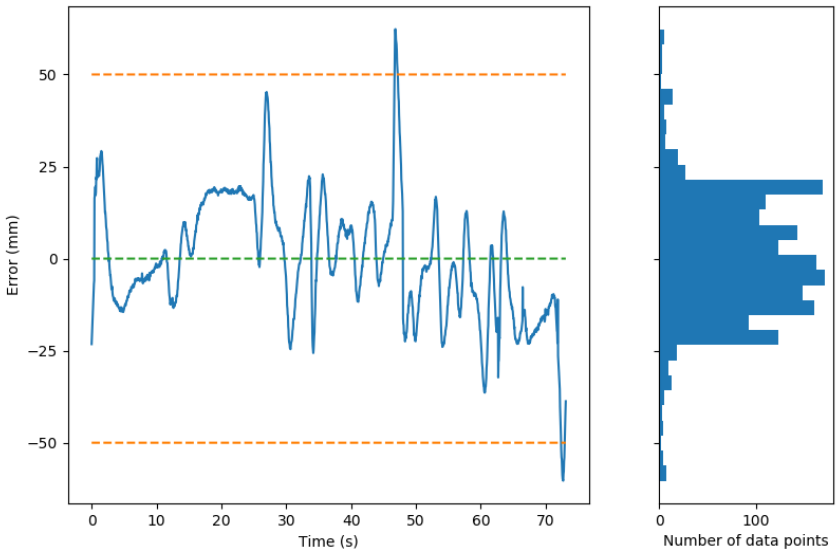


Figure 6.2: The error over time for the first test run. The green dotted line is where the error is 0 and the two orange dotted lines shows where the absolute value of the error is above 5 centimeters. The histogram to the right shows the distribution of data points with the same Y-axis values as the left graph.

6.2 Kinematic Model

To be able to accurately plan a path for the truck and trailer to follow, the internal mathematical representation of the vehicle has to be accurate. To verify the kinematic model we have constructed a test that makes the truck drive through some difficult parts of the testing track. The path used includes both the roundabout and a T-crossing. Making the truck follow this given path we can study how well the trailer's position is represented by the model throughout the run. Trailer angle relative to the tractor unit is given by an angle sensor and the actual position is then calculated using simple trigonometry. The position calculated is the middle of the back of the trailer. This test has been run twice.

Figure 6.3 shows that both the tractor unit, as previously evaluated, and the trailer follows the path calculated very well. There are several disturbances in the system. The tractor is not following the reference path exactly which means that there is almost always an initial error from which the trailers position will be measured. Another source of error is the sensor itself, it introduces a lot of noice. Also, the tractors position is measured using GulliView and by that the position gathered is not a precise reproduction of reality, even though it is very close.

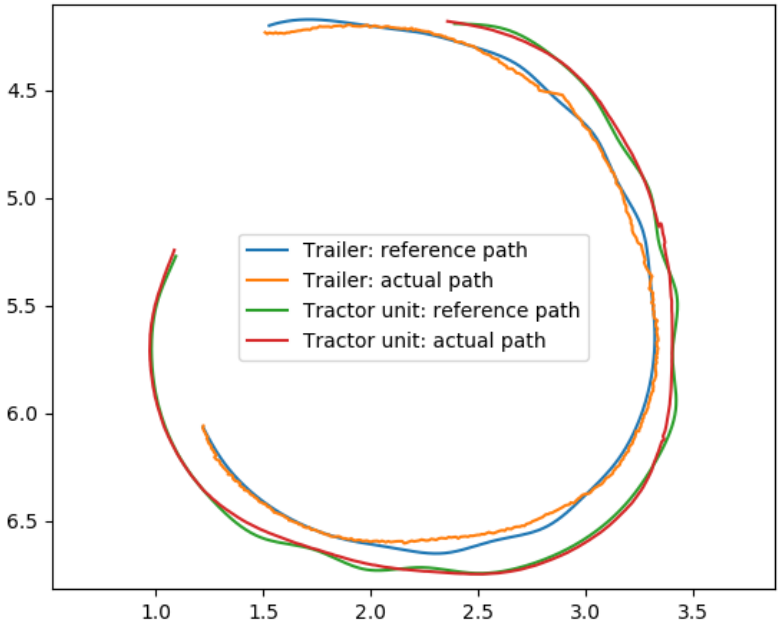


Figure 6.3: A plot of how well the trailer and tractor unit is following their given paths. The data points are from the first run, values can be found in Table 6.2.

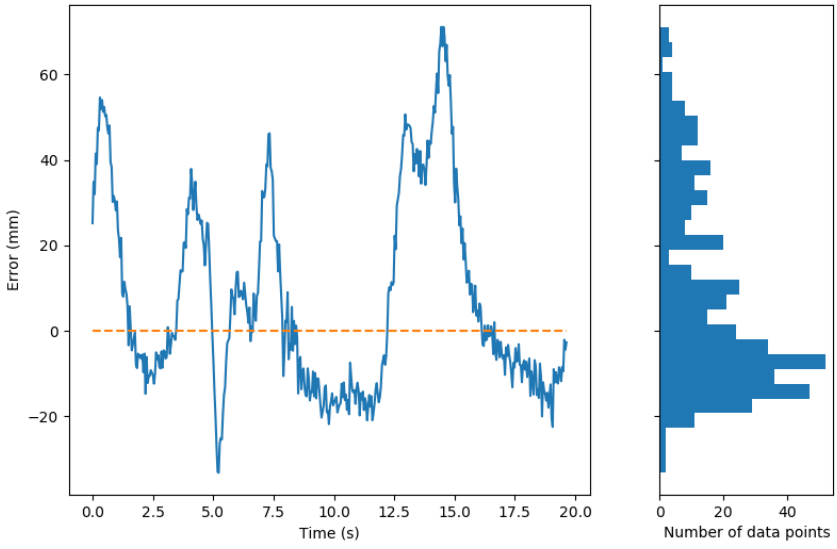


Figure 6.4: The error over time comparing the position of the trailer with the calculated estimated position

Given all these error sources the model is rather accurate, the absolute error is always rather low. Figure 6.4 shows how the error changes over time and also the distribution of data points. We can observe some instant value changes at, for example, $t = 5s$. For one or two data points the error goes from around 20 mm to -30 mm and then back again to around 10 mm. This is most certainly because of the error sources in the system. It is impossible for the truck to move like that.

Table 6.2 uses the same parameters as used for the PID evaluation in section 6.1. As shown in the table, 95% of the time we have an error of less than 5 cm. The absolute max error is 111 mm which is rather high, but again, this is probably because of some error in the localization system.

Test run	$\|e_{max}\|[mm]$	$\|e_{avg}\|[mm]$	$\overline{e}[mm]$	$t_{1cm}[\%]$	$t_{3cm}[\%]$	$t_{5cm}[\%]$
1	71.1	18.5	7.49	64	78	94
2	110.9	19.7	0.04	74	88	95

Table 6.2: Test results for the kinematic model

6.3 Path Planning

For the system to function well, the path planner has to be fast. When computing the path for longer sections of the road, the route is split into several smaller sections, each handled sequentially by the path planner. This way the truck can start driving along the first part of the route, meanwhile a path is being calculated for the next section. Ideally, the path planner should return a path for the subsequent section before the vehicle gets there, so that it does not have to stop and wait.

While speed is important, we also want the calculated paths to be as good as possible. To be able to measure and compare the quality of different paths, we use a cost function. This function takes into account both the deviation from the global path and the amount of time spent in the opposite lane.

Both the runtime and the quality of the path depends largely on the values of the heuristics. Since their interests are in conflict, we need to find a middle way when deciding on appropriate parameter values. The test cases presented here are focused on evaluating the effect on runtime and path quality with different grid size parameters. We have tried a few combinations, on two different traffic scenarios. The scenarios are referred to as S_1 and S_2, and can be seen in Figures 6.5 and 6.6. All tests are performed on the Raspberry Pi.

Most of our system is written in Python, including the original version of the path planner. However, since this is the most computation heavy part of the system, we decided to translate the path planner into C++, with the hope of improving the performance. In Table 6.3, we show an example, highlighting the difference in runtime between the two versions.

Version	Traffic scenario	Grid size parameters	Step size	Runtime [s] (first layer)	Runtime [s]
Python	S_1	(6, 0.4)	25	1	**36**
C++				0.2	**6**

Table 6.3: Difference in runtime between Python and C++ versions, with the exact same parameters.

Without any further optimization, this made the algorithm run about 6 times faster. A shorter execution time means that we can get better paths within an acceptable time frame, and is especially beneficial when running the system on the Raspberry Pi. All other tests presented in this section are executed with the C++ version of the path planner.

In Table 6.4 and Figure 6.5, we show the effects on cost function output and overall runtime when changing the values of the heuristics. The grid size parameters decide the rounding values for the (x,y)-coordinates and vehicle angles, and are displayed as **(coordinate_value, angle_value)**.

Traffic scenario	Grid size parameters	Visited nodes	Cost (first layer)	Cost	Runtime [s] (first layer)	Runtime [s]
S_1	**(10, 0.5)**	956	783	**390**	0.12	**2.2**
	(6, 0.4)	3518	710	**329**	0.2	**6**
	(3, 0.3)	16054	710	**310**	0.24	**26**
S_2	**(6, 0.4)**	446	640	**499**	0.16	**0.92**
	(3, 0.3)	895	437	**423**	0.29	**1.6**
	(1, 0.1)	4755	437	**423**	0.5	**5.6**

Table 6.4: Difference in runtime and cost with different grid size parameters. The displayed cost is an average value of the cost function, for each discrete step of the path. The step size is set to 25 for all test cases.

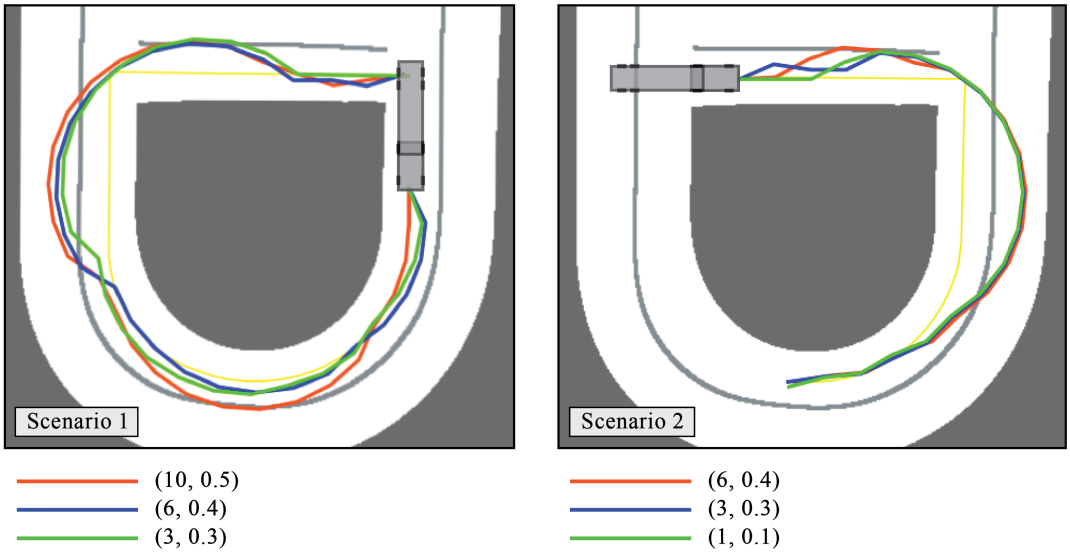


Figure 6.5: Difference in path generated with different grid size parameters. The yellow line represents the global reference path.

As can be seen in Table 6.4, the lower parameter values gives a lower average cost, and thus provides better solutions. At the same time, the runtime increases as a result of visiting more nodes. As mentioned before, the computations are time sensitive, and we have to weigh the gain in path quality against the impacts on overall performance. Given the speed of the truck (0.5 m/s), a runtime of more than 20 seconds for a path of relatively short length, as we see in the last entry for S_1, is not acceptable.

One can also see that the difference in cost, when comparing the lowest parameter values with the middle range values, is not proportional to the difference in runtime.

Figure 6.6 shows the difference in quality between the paths generated by the first and second layer of the path planning algorithm. While the first layer paths are feasible, in the sense that the vehicle does not drive outside of the track, they are far from optimal. Large sections of the paths are drawn in the opposing lane, especially for S_1, which makes the cost function value go up to more than the double for the worst cases. This motivates the need for a second layer, and shows why we can not simply take the path generated by the first layer and be satisfied with that.

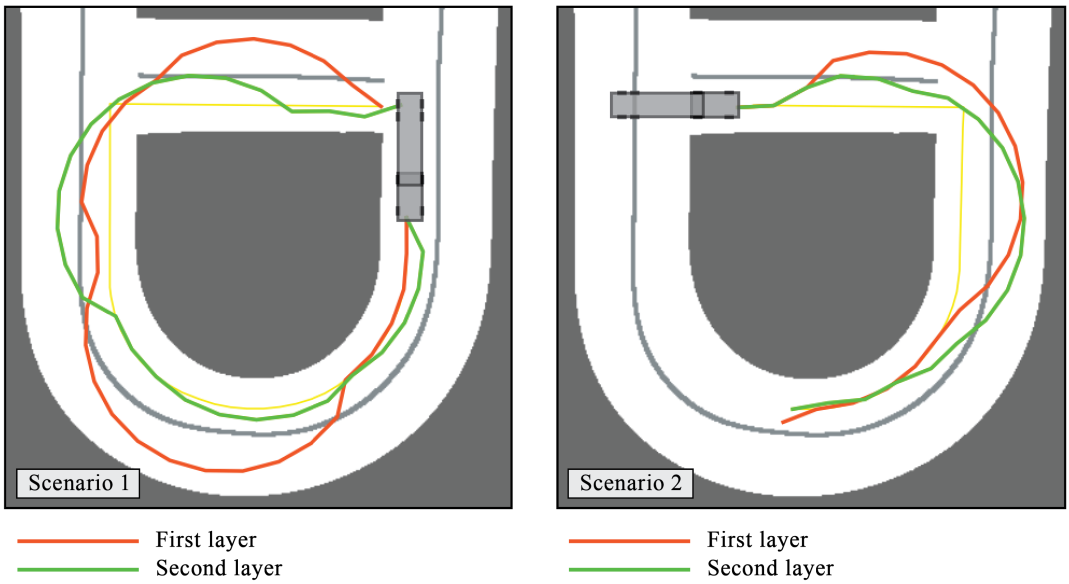


Figure 6.6: Difference in path generated by the first and second layer of the algorithm. Grid size parameters are (6, 0.4) for both cases. The yellow line represents the global reference path.

7. Discussion and Conclusion

This Bachelor's thesis proposed a predictive control solution to automated heavy articulated vehicles. The implemented solution integrated the discussed PID control and a path planning strategy that turn out in a complete and efficient testbed that responds under difficult traffic scenarios such as roundabouts and T-crossings. The system also calculates the trajectory path on-the-fly and readjusts if obstacles are detected on the road. Along this chapter the obtained results are discussed to demonstrate how the initial requirements have been achieved. Furthermore, the performance of the system is compared to related work to highlight the advantages of the presented work.

7.1 Meeting the Evaluation Criteria

One of the main focuses of this project was to run all parts of the system, excluding the user interface. This makes the testbed portable and adaptable to other systems or environments. Being able to run on-the-fly the testbed on the Raspberry Pi proves that the designed automated system is fast enough to satisfy the evaluation criteria as presented in Table 6.3.

The ultimate aim of this project is to drive the articulated model truck autonomously to a goal destination. For this, the designed testbed calculates a path that stays in its own lane when it is possible by minimizing the path planning cost function, and to prevent getting out of the road. For this purpose it has been shown in the previous chapter that the PID control follows the given paths within a 5 cm error, Table 6.1, and that the path planning provides a collision model accounting for this margin. Then, the initial statement that the truck always drives inside the given track is, with high certainty, fulfilled.

7.2 Related Work

When looking into previous work carried out in this field, we could not find a single project that attempts to solve the very same problem as we do. It is therefore hard to compare our results to the work of others in a formal way. Instead, we will do a general comparison to a somewhat similar project [6], and then discuss some of the components of our system.

A Related Project

The project in question is a KTH Masters Thesis [6], where they implement an autonomous driving system for heavy-duty vehicles, moving in unstructured environments. The vehicle is very similar to ours, using almost the exact same kinematic model, but the environment is different. While we have a set environment, with roadways and opposing lanes, they have a large open space and use movable boxes to create dynamic structures.

They use the Rapidly exploring Random Trees (RRT) algorithm [20] to plan their trajectory, and a Model Predictive Controller (MPC) [21] to predict and regulate the movement of the vehicle. We have chosen a different approach, with our own, deterministic path planning algorithm, and a PID controller combined with simulations based on the kinematic model. Both approaches seems to be working well for their respective applications, and it is hard to say which one is better.

When comparing the performance of the two systems, our system seems to be working faster. Although, since they do not provide any actual runtime data, we can only judge from visual inspection. Some of the difference in runtime is likely due to the fact that we split longer paths into several smaller sections. This way the truck can start driving within a few seconds, while simultaneously computing a path for the subsequent section, and hardly ever has to stop and wait. They do not start driving until they have calculated the entire path, and thus have to stand still for the full computation time.

Predictive Control

The standard method when implementing autonomous vehicular systems involves using some kind of predictive control. Predictive control can be accomplished in many different ways, for example using MPC [21]. Even though MPC seems to be widely used, we could not find any actual examples on how to implement one. Since we found that a PID controller, combined with predictions based on the kinematic model, was fully sufficient for this project, we decided to go with this more straightforward approach.

Path Planning

When creating time efficient algorithms, a common approach is to introduce an element of randomness. An example of this is RRT [20] which has been used in various projects [22],[23]. We instead decided on a deterministic heuristic approach. We use a search grid to limit the number of nodes, and a cost function that ensures favoring of the own lane before the opposing. Having a deterministic algorithm means that there is no risk for unfavorable worst-case behaviour, and that we know how the soft parts of the system will behave, given the same set of heuristic parameters. Even

though we can not guarantee that a computed path is the global optimum, we can say with certainty that the path will always be equally good or better if we lower the grid size.

7.3 Future Work and Extendability

The flexibility of a system like this is of great importance, and that is something we have achieved. Great flexibility means that the platform can be extended to conduct other studies and experiments. Our platform can easily be extended with other positioning systems, better sensors and other path-planning algorithms. The model can be exchanged or extended to work for backwards driving vehicles[24].

Currently the truck is always supplied with the same signal for speed, meaning both that the speed is dependent of the resistance of the wheels and the battery level, also that the speed is not adapted to the road conditions. For future work a speed controller can be implemented to keep the desired speed at all times. Some system to get the current speed limit to allow for acceleration and deceleration could also be implemented.

For future projects, the camera system could be improved, replaced or combined with something else. Preferably, the localization system should be integrated on the truck, to not be dependant on external systems. Positioning is important for being able to accurately follow a path and the position received from the camera is not stable enough. It does give more or less the correct value, but it needs some filtering. For this purpose a Kalman filter [25] would fit well. Introducing a filter to both the angle sensor and the positioning system would significantly improve the robustness of the system.

Bibliography

[1] World Health Organization (WHO), "World health statistics 2016 report," http://www.who.int/gho/publications/world_health_statistics/2016/whs2016_AnnexA_RoadTraffic.pdf?ua=1&ua=1.

[2] O. Olarte, "Human error accounts for 90% of road accidents," http://channel.staging.alertdriving.com/home/fleet-alert-magazine/international/human-error-accounts-90-road-accidents, April 2011.

[3] Business Insider, "8 ways driverless cars will drastically improve our lives," http://www.businessinsider.com/8-ways-driverless-cars-will-drastically-improve-our-lives-2015-12?r=US&IR=T&IR=T/#thousands-of-lives-will-be-saved-each-year-1.

[4] S. Dädeby, A. Eriksson, P. Khosravi, K. Onsjö, and K. Sandell, "Simulation av gulliver - en virtuell robotmiljö för skalade autonoma fordon," 2015, Bachelor Thesis, Department of Computer Science and Engineering, Chalmers University of Technology.

[5] A. Arkheden, A. Lindhé, R. Gustafsson, and R. Zaragatzky, "Ett prisvärt alternativ för global visuell lokalisering och styrning av autonoma fordon," 2016, Bachelor Thesis, Chalmers University of Technology.

[6] R. Oliveira, "Planning and motion control in autonomous heavy-duty vehicles," 2014, Masters Thesis, KTH.

[7] A. Elhassan, "Autonomous driving system for reversing an articulated vehicle," 2015, Masters Thesis, KTH.

[8] I. Nagrath and M. Gopal, *Control Systems Engineering*. John Wiley & Sons Canada, Limited, 1982. [Online]. Available: https://books.google.se/books?id=XZpyPwAACAAJ

[9] J.-J. E. Slotine and W. Li, *Applied nonlinear control*. Upper Saddle River, NJ: Pearson, 1991, the book can be consulted by contacting: BE-ABP-CC3: Pfingstner, Juergen. [Online]. Available: https://cds.cern.ch/record/1228283

[10] K. J. ström and T. Hägglund, *PID Controllers: theory design and tuning*. International Society of Automation, 1934.

[11] S. M. LaValle, *Planning Algorithms*. New York, NY, USA: Cambridge University Press, 2006.

[12] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, and R. Wheeler, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[13] S. Hangal and A. Söderberg-Rivkin, "GulliView: A Vision Based Localization System for Autonomous Vehicles," March 2014, Final project report in Autonomous and Cooperative Vehicular Systems (DAT295), Department of Computer Science and Engineering, Chalmers University of Technology.

[14] Itseez, "Open source computer vision library," https://github.com/itseez/opencv, 2015.

[15] E. Olson, "AprilTag: A robust and flexible visual fiducial system," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, May 2011.

[16] A. L. Romain Pepy and H. Mounier, "Path planning using a dynamic vehicle model," Université Paris-Sud XI. Orsay, France., Tech. Rep.

[17] J. M.Snider, "Automatic steering methods for autonomous automobile path tracking," Carnegie Mellon University, Pittsburg, Pennsylvania, Tech. Rep., February 2009.

[18] R. E. Colyer and J. T. Economou, "Comparison of steering geometries for multi-wheeled vehicles by modelling and simulation," in *IEEE Conference on Decision and Control*, 1998.

[19] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, no. 1, pp. 269–271, Dec. 1959. [Online]. Available: http://dx.doi.org/10.1007/BF01386390

[20] S. M. Lavalle, "Rapidly-exploring random trees: A new tool for path planning," Tech. Rep., 1998.

[21] J. Rawlings and D. Mayne, *Model Predictive Control: Theory and Design*. Nob Hill Pub., 2009. [Online]. Available: https://books.google.se/books?id=3_rfQQAACAAJ

[22] Y. Kuwata, J. Teo, G. Fiore, S. Karaman, E. Frazzoli, and J. P. How, "Real-time motion planning with applications to autonomous urban driving," *IEEE Transactions on Control Systems Technology*, vol. 17, no. 5, pp. 1105–1118, 2009.

[23] R. Pepy, A. Lambert, and H. Mounier, "Path planning using a dynamic vehicle model," in *Information and Communication Technologies, 2006. ICTTA'06. 2nd*, vol. 1. IEEE, 2006, pp. 781–786.

[24] C. Altafini, A. Speranzon, and B. Wahlberg, "A feedback control scheme for

reversing a truck and trailer vehicle," *IEEE Transactions on Robotics and Automation*, vol. 17, no. 6, pp. 915–922, Dec 2001.

[25] R. E. Kalman and Others, "A new approach to linear filtering and prediction problems," *Journal of basic Engineering*, vol. 82, no. 1, pp. 35–45, 1960.

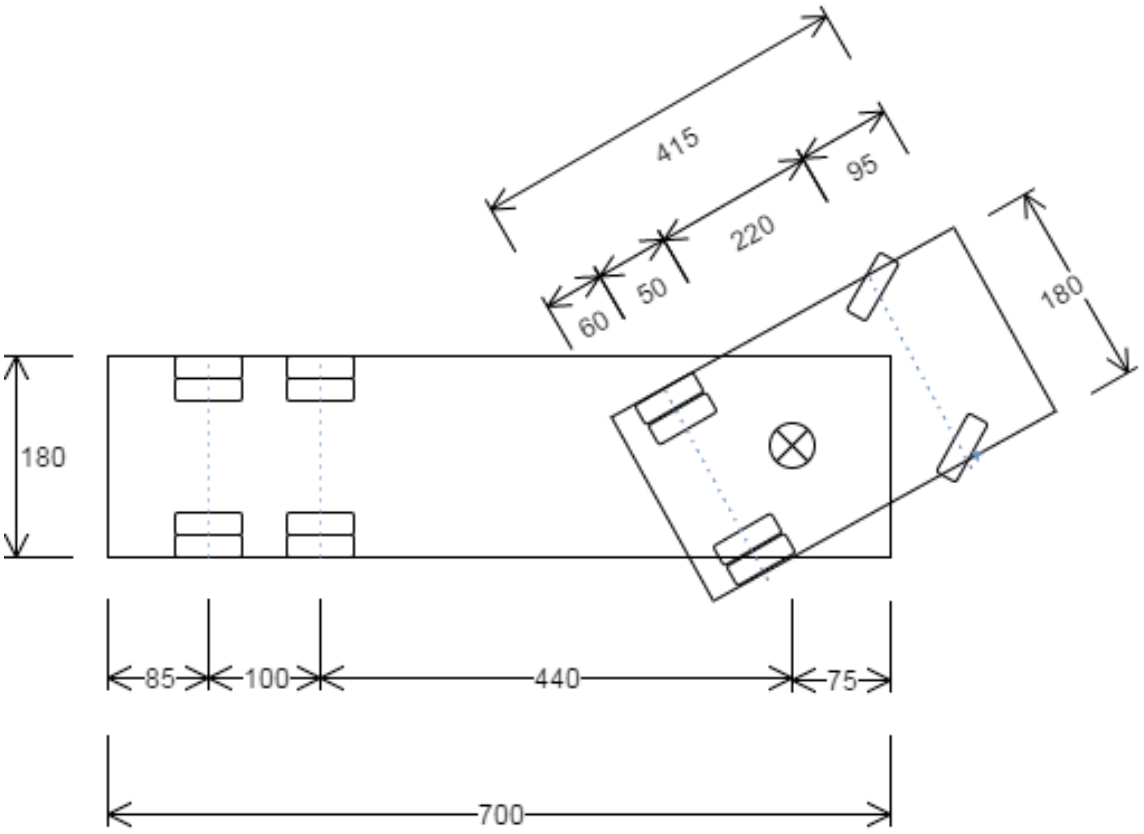
Appendix

A Publisher subscriber system example

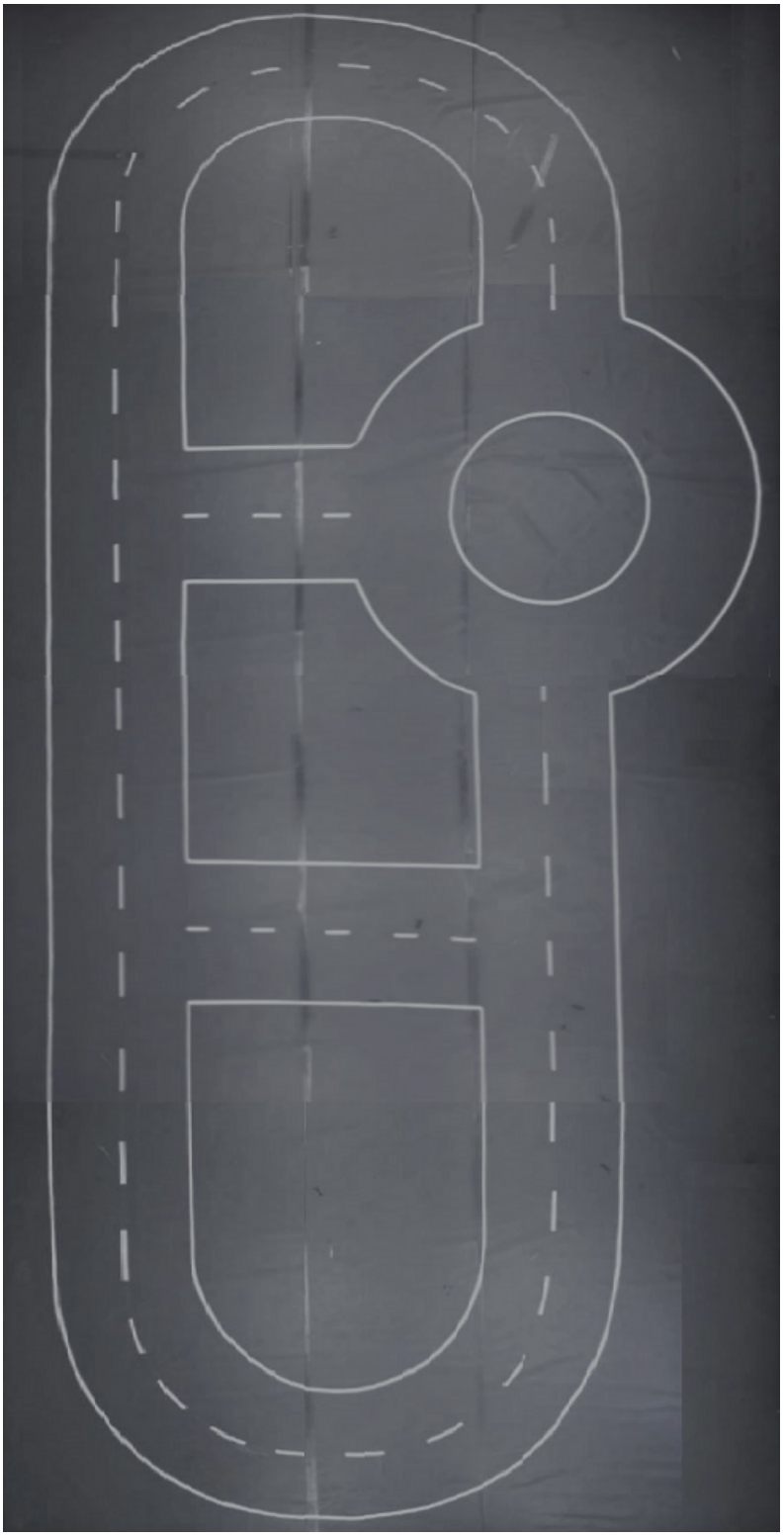
To make a publisher, you use the built in class that rospy provides to create a publisher object. Use that object to call the publish function with your message as argument. To make a subscriber you need a callback function that is called each time a message is sent on the topic. Then just create the subscriber with the built in constructor that the rospy package provides. Following is some sample code that shows how simple it is to use topics with primitive message types in ROS.

```
# publisher.py
import rospy
from std_msgs.msg import Float32

class Publisher:
    def __init__(self):
        rospy.init_node('pub_node')
        self.pub = rospy.Publisher('test', Float32, queue_size=10)
        i = 0
        while(1):
            self.pub.publish(i)
            i = i + 1

if __name__ == "__main__":
    Publisher()

# subscriber.py
import rospy
from std_msgs.msg import Float32

class Subscriber:
    def __init__(self):
        rospy.init_node('sub_node')
        rospy.Subscriber('test', Float32, self.callback)
    def callback(self, data):
        print "received: ", data.data

if __name__ == "__main__":
    Subscriber()
    rospy.spin()
```

B Truck measurements (mm)



C The test track



D Obstacle GUI

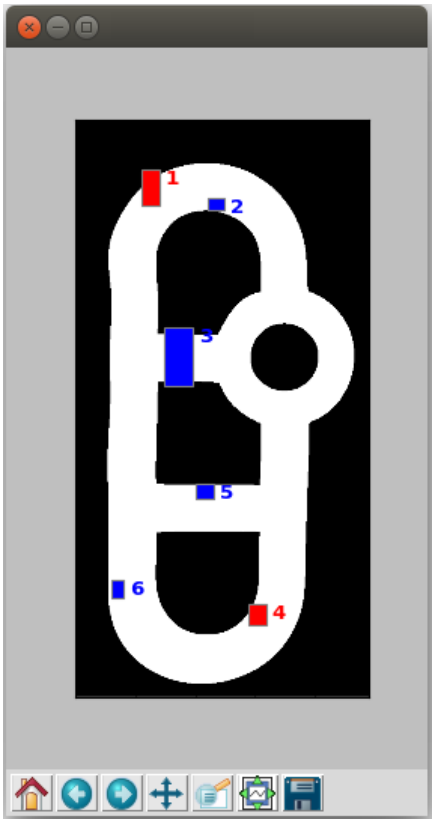


Figure 7.1: Obstacle GUI, used to add virtual obstacles to the track. In this example, obstacle one and four are active.

E Manual Control

Figure 7.2: Control scheme for an Xbox 360 controller, mapping buttons to control actions.

F Rounding function

The rounding function is used to find the grid of a state.

```
def rounding(x, y, theta1, theta2):
    modPoint = 3                        #tuning parameter
    modTheta = 0.3                      #tuning parameter

    m_x = x % modPoint
    if m_x >= (modPoint/float(2)):   #round up
        x = x-m_x + modPoint
    else:                               #round down
        x = x - m_x

    m_y = y % modPoint
    if m_y >= (modPoint/float(2)):   #round up
        y = y-m_y + modPoint
    else:                               #round down
        y = y - m_y

    theta1 = round(theta1, 1)
    m_t1 = round(theta1 % modTheta, 1)
    if m_t1 >= (modTheta/float(2)): #round up
        theta1 = theta1-m_t1 + modTheta
    else:                               #round down
        theta1 = theta1 - m_t1

    theta2 = round(theta2, 1)
    m_t2 = round(theta2 % modTheta, 1)
    if m_t2 >= (modTheta/float(2)): #round up
        theta2 = theta2-m_t2 + modTheta
    else:                               #round down
        theta2 = theta2 - m_t2

    return ((x,y),theta1,theta2)
```

G k-shortest-paths function

The k-shortest-paths function is used to find the k shortest loop-less paths between two nodes in a directed graph.

```
def kShortestPaths(graph, start_node, end_node, k):
    if start_node == end_node:
        return []

    path_heap = []        # Heap data structure, holding partial paths
    paths = []            # Array, holding the complete paths

    graph.resetGraph()    # Resetting node.count for all nodes
    heappush(path_heap, (0, [(start_node.x, start_node.y)]))

    # Repeating until the end node has been visited k times,
    # or there are no more paths from the start node to the end node
    while end_node.count < k and path_heap:
        cost, path = heappop(path_heap)
        current_node = graph.getNode(path[-1][0], path[-1][1])
        current_node.count += 1

        # If the end node has been reached,
        # this path is added to the array of shortest paths
        if current_node == end_node:
            paths.append(path)

        # Otherwise if the current node is not already in all k paths,
        # all possible paths forward from this node are added to the heap
        elif current_node.count <= k:
            for out_edge in current_node.out_edges:
                if (out_edge.x, out_edge.y) not in path:
                    new_path = []
                    for coord in path:
                        new_path.append(coord)
                    new_path.append((out_edge.x, out_edge.y))
                    new_cost = cost + current_node.getEdgeLength(out_edge)
                    heappush(path_heap, (new_cost, new_path))

    return paths
```
