CODEN:LUTEDX/(TEIE-3070)/1-115/(2017)

Industrial Electrical Engineering and Automation

Software development of a speed dependent control system for the electric steering of a vehicle operated by joysticks

Tom Andersson
Jacob Ingvarsson

Division of Industrial Electrical Engineering and Automation
Faculty of Engineering, Lund University

Software development of a speed dependent control system for the electric steering of a vehicle operated by joysticks

LUNDS UNIVERSITET
Lunds Tekniska Högskola

LTH School of Engineering at Campus Helsingborg
Division of Industrial Electrical Engineering and Automation

Bachelor thesis:
Tom Andersson
Jacob Ingvarsson

LTH School of Engineering
Lund University
Box 882
SE-251 08 Helsingborg
Sweden

LTH Ingenjörshögskolan vid Campus Helsingborg
Lunds universitet
Box 882
251 08 Helsingborg

Printed in Sweden
Industrial Electrical Engineering and Automation
Lunds universitet
Lund 2017

Abstract

The technology has advanced but the vehicle industry has not evolved at the same pace. Mechanical constructions with a steering wheel are still being used to control the steering of cars. The mechanical construction makes it harder to implement autonomous features, it adds weight and takes up space. It is commonly known that the steering wheel has been along since the early days of the car but is it still, with all this technological development, the best way of steering a vehicle? In this thesis, a control system for an electrical steering of a vehicle was developed. The vehicle was using two joysticks to steer and since these two have a lower range of motion than a normal steering wheel a system that makes it possible to steer the vehicle in a safe way at various speeds was developed.

The control system was created by analysing the physical system and approximating mathematical models. Since the dynamics changes when the friction between the wheels and the ground changes several models that operates in different speed intervals were made. Thereafter, a PID parameters were adapted to get optimal step responses. The implementation succeeded but it was proven that the servo that created the turning of the wheels was too weak to achieve a natural steering.

By analysing which angles of the wheels that are needed at different speeds and how cars today are constructed a mathematical function where the maximum angle of the wheels decreases with increased speed was developed. In addition, the closer the steering controllers are to the middle point of their range of motion the more is the sensitivity decreased when the speed is increased. This makes it possible to drive straight ahead without small unintended movements in the hands affecting the movement of the vehicle while it is also possible to make sudden evasive manoeuvres. A VR-environment made to scale was created to test this system. This test was successful but the system will however need to be tested in a real situation to be able to assure its suitability.

To create a natural steering for the vehicle force feedback from the wheels to the steering controllers is necessary. This limits the positions of the steering controllers if the desired wheel angle cannot be achieved as well as making it possible for the driver to feel the road surface. A theoretical design was made but due to delays in the mechanical construction implemented by a mechanical engineer this could not be implemented within the frame of the thesis.

This thesis has mainly been focusing on functionality but some safety functions have been implemented and tested with successful results. In addition, a theoretical study regarding redundancy was performed to facilitate future implementation.

Keywords:

- Steer by wire
- Handles
- Control system
- Speed dependence
- Parameter control

Abstract in Swedish: Sammanfattning

Tekniken har avancerat men fordonsindustrin har inte utvecklats i samma takt. Fortfarande används mekaniska konstruktioner med ratt för att styra bilar. Den mekaniska konstruktionen gör det svårare att implementera autonoma funktioner, den tillför vikt och tar upp plats. Det är allmänt känt att ratten har funnits med sedan bilens tidiga år men med all teknisk utveckling som skett, är ratten fortfarande det bästa sättet att manövrera ett fordon på? I detta examensarbete utvecklades ett reglersystem för en elektrisk styrning till ett fordon. Fordonet styrdes med två joysticks och då dessa har ett mindre rörelseomfång än en normal ratt utvecklades ett system som gör det möjligt att på ett säkert sätt styra fordonet vid olika hastigheter.

Reglersystemet skapades genom att analysera det fysiska styrsystemet och approximera matematiska modeller. Då dynamiken för systemet förändras då friktionen mellan hjulen och marken ändras skapades flera modeller som verkar inom olika hastighetsintervall. Därefter anpassades PID-parametrar för att få optimerade stegsvar. Implementeringen lyckas men det visade sig att servot som skapar hjulens vridning var för svagt för att ge en naturlig styrning.

Genom att analysera vilka hjulvinklar som krävs vid olika hastigheter och hur bilar idag är konstruerade skapades en matematisk funktion där den maximala vinkeln på hjulen minskas med ökad hastighet. Dessutom minskar känsligheten ju närmre man kommer mitten av de två styrkontrollernas rörelseomfång vid ökad hastighet. Detta för att göra det möjligt att både köra rakt fram utan att små oavsiktliga rörelser påverkar fordonets körriktning samtidigt som man kan utföra plötsliga undanmanövrer. En skalenlig VR-miljö skapades för att testa detta system. Resultatet var lyckat men systemet kommer ändå behövas testas i en verklig situation för att man ska kunna försäkra sig om dess lämplighet.

För att skapa en naturlig styrning hos fordonet behövs kraftåterkoppling från hjulen till styrkontrollerna. Detta för att bland annat begränsa styrkontrollernas position om önskad hjulvinkel inte kan uppnås och för att göra det möjligt för föraren att känna vägens underlag. En teoretisk design skapades men på grund av förseningar i den mekaniska konstruktionen som utfördes av en maskiningenjör kunde detta inte implementeras inom ramen för examensarbetet.

Detta examensarbete fokuserade till huvudsak på funktionalitet men en del säkerhetsfunktioner implementerades och testades med lyckade resultat. Dessutom gjordes en teoretisk studie angående redundans för att underlätta när företaget ska implementera det i framtiden.

Keywords in Swedish: Nyckelord:

- Elektronisk fordonsstryning
- Styrspakar
- Reglersystem
- Hastighetsberoende
- Parameterstyrning

Table of contents

Preface

We would like to thank everyone who was involved in this thesis. Special thanks to our supervisor at LTH, Mats Lilja, our examiner at LTH, Johan Björnstedt, our supervisors at Uniti, Michael Bano and Kristofer Jansson, as well as our project manager at Uniti, Anton Franzén.

Also, big thanks to the whole staff at Uniti for letting us be a part of your great vision. The people at Uniti have not only become our co-workers but also our friends.

Finally, special thanks to all our friends and family who have been supporting us throughout the thesis.

1 Introduction

This section gives a background and a short presentation of the company, at which the thesis work was done, followed by an overview of the goals and the problem formulation of the work. It also outlines the motivation for the thesis work and describes some limitations.

1.1 Background

The thesis was executed at Uniti Sweden AB in Lund. Uniti is a company that is developing an electric vehicle. To do that, Uniti bought a commercial electric car as a first test vehicle on which the systems for Uniti's future vehicle will be implemented to make sure they work properly together.

The thesis consisted of developing and implementing a steer by wire system on the early test vehicle. Steer by wire is an electrical system, instead of a traditional mechanical system, to manage the steering of a vehicle. Steer by wire technology is seldom seen in automobiles. While it has existed in a few concept cars in the past, Uniti intends to release a commercial vehicle that utilizes this technology. In addition, the vehicle will have two joysticks instead of a steering wheel. The same applied for the test vehicle. The test vehicle is shown in figure 1 and 2.

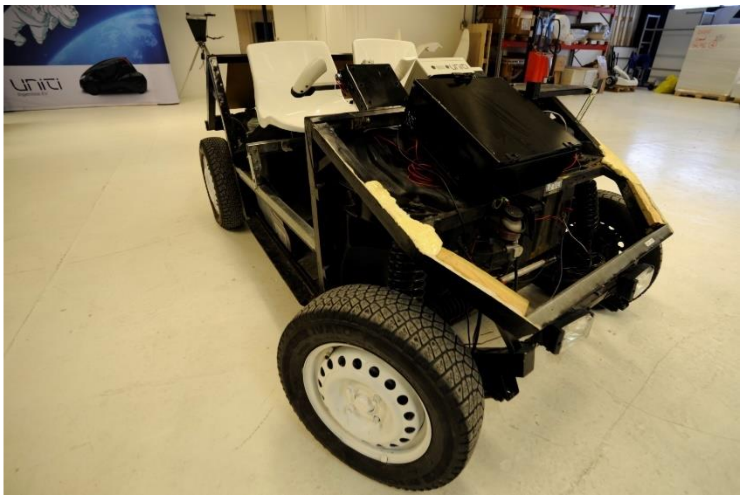

Figure 1: The test vehicle.

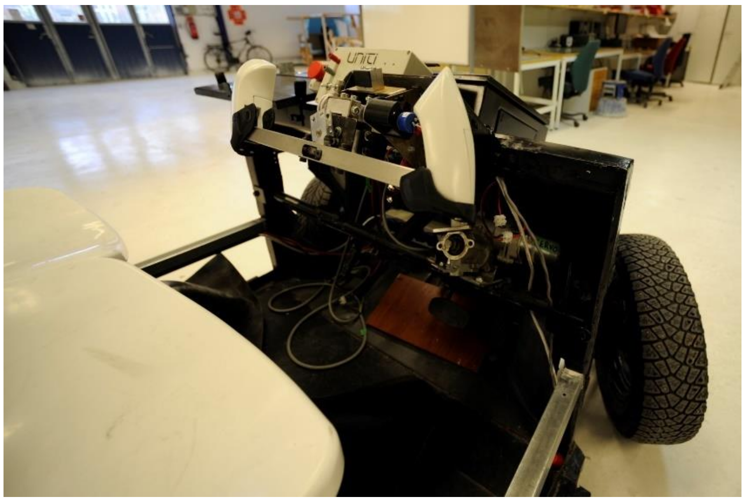

Figure 2: The joystick setup of the test vehicle.

1.2 Purpose

A steer by wire system is more adjustable and dynamic compared to a mechanical system which prepares the vehicle for a future system for autonomous driving as well as making it easier to integrate the steering system with other systems of the vehicle, for example the heads-up display. It will also take up less space and the weight will be lower.

1.3 Goal setting

The goals were to develop a control system for the steering of the test vehicle where the mismatch between driver input and the real steering angle is eliminated in a way that gives a smooth and natural steering experience at various speeds. Also, force feedback in the joysticks were going to be implemented.

1.4 Problems

This section lists the problems that had to be solved throughout the thesis.

1. How should the steering control depend on the current speed?
2. What response time, overshoot, rise time and settling time yields a smooth and natural steering experience?
3. How should response time, overshoot, rise time and settling time be prioritized to achieve a smooth and natural steering experience?
4. What control technique is most suitable to achieve the specified response time, overshoot, rise time and settling time?
5. Which safety requirements are relevant?
6. How should the safety requirements be achieved?

1.5 Motivation

We found the vision and passion at Uniti to be very inspiring and something that we wanted to be a part of. The steer by wire project combined the knowledge from several of our favourite courses which in conjunction with the vision of developing an electric vehicle made the project exciting and challenging.

To ensure our planet’s future we need to reduce the carbon dioxide emissions. The traditional cars with combustion engines stands for a large part of those and therefore it is a necessity to get electrical vehicles to the market.

1.6 Limitations

The system was only implemented on the test vehicle. The focus was on software while two other teams focused on hardware and mechanical implementations. The safety features that were implement did not include redundancy. However, a theoretical analysis regarding redundancy was made.

2 Steering systems

This section consists of studies that were done at the beginning of the thesis with the purpose of gaining knowledge regarding existing applications that tackles tasks similar to the ones in this thesis. It is also explained how the system was structured at the beginning of the thesis with explanations of its shortcomings. Finally, different ways of implementing redundancy is studied and a suitable redundancy method for a steer by wire system is recommended.

2.1 The structure of mechanical steering constructions

Rack and pinion steering is the current design that is prominent in the automotive industry. This system consists of a rack that moves to the right or left when pushed by a fixed cog directly connected to the steering wheel by an axle. When the rack moves to the right or left the wheels turn correspondingly since they are connected to it with a fixed armature [12]. This is illustrated in figure 3 [22].

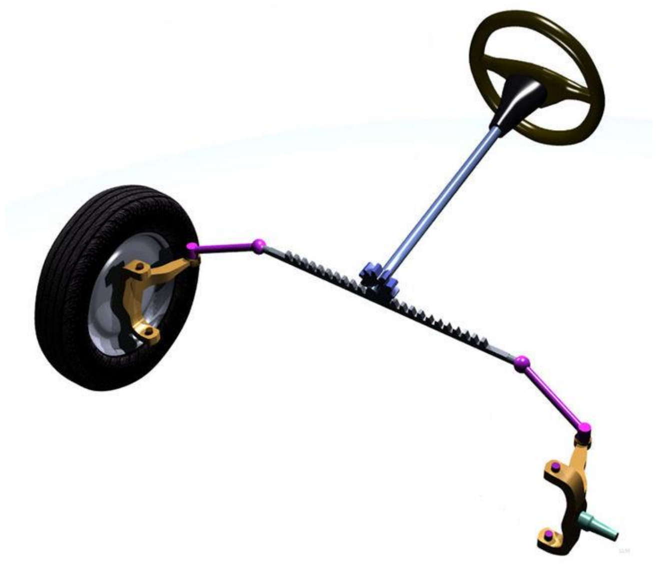

Figure 3: Illustration of a rack and pinion setup [22].

When talking about the current design of the steering system in vehicles the term steering ratio comes up. The values of this is stated as how many degrees the steering wheel turns to turn the wheels one degree. The relation between them may vary depending on the degrees the wheels are at which in turn means which position the pinion is at. This is a mechanical solution aimed to aid the driver when at a sharp steering angle. To manage this, the cogs of the pinion and the gear that rotates differ from each other in both size and distance depending on the degrees of rotation [18].

Steering assist is another term also having a hand in replacing the current design since it has to do with what steering ratio one should choose. A higher steering ratio and a larger diameter on the steering wheel allows a lower amount of input force to be applied by the driver on the steering wheel to move the wheel direction. The assistance needed may vary since the wheel turning friction between the road and the wheels contact surface is lower at high speeds. At low speeds the driver needs more assistance to turn the wheels. This is done by adding either a hydraulic or an electrical system to the pinion [12]. The system applies a force proportional to the force applied by the driver resulting in the turning being easier to perform. Looking at the design of modern steering wheels compared to older ones the size of the steering wheel has been reduced. This is the result of implementing the steering assist [18].

The current rack and pinion design of the steering setup has today experienced some design changes and improvements. These improvements come in varying form. They tend to solve the same issue but in different ways. When it comes to alter the steering ratio the more common one is the variable pinion cogs mentioned before. This method relies on changing the cog gearing at the extremes. Other examples of implementing a variation is adding a planetary gearbox on the steering shaft. This is a newer method, called active steering, which aims to aid the driver at both lower and higher speeds. At lower speeds the gearing is set up in a way so that the driver does not have to turn the steering wheel as far as he or she used to for the same turning angle. When driving at higher speeds, it is set up to increase the directional detail which gives a variable steering ratio [13].

The common way to implement a steering assist is to aid the driver input at lower speeds with an electrical motor or a hydraulic system that applies a force which is proportional to the driver’s input force. The torque is measured at the lower end of the steering column by a so-called torsion bar [18].

Steering systems have gone from being a fixed rack and pinion system without any sort of assistance to be further developed into a system with a hydraulic or electrical assistance to aid the driver. Next step of development can be to replace the mechanical linkage with a steer by wire system. A steer by wire system can replace the need for having a planetary gearing ratio on the steering column for aiding the driver as well as the need for having a variable gearing on the pinion for varying the ratio at the extremes. This would allow further customization and more advanced systems to improve the steering sensation and responsiveness.

2.2 Existing steer by wire systems

So far, Nissan is the only company that manufactures a commercial car with a steer by wire system but they still use a mechanical system as a backup. They announced it in the 2014 Infiniti Q50 which was released in 2013 [1]. A steer by wire system is still used in the model of 2017 [2]. Nissan sold a record of 215,250 Infiniti vehicles in 2015 which was an increase of 16 percent since the year before. The Q50 alone sold 78,800 units, an increase of 44 percent since the year before, which made it the bestselling vehicle of the Infiniti series in 2015 [3]. That steering system uses not only a mechanical backup but the electrical system is also redundant with three separate control units [11].

In the 2014 model the steer by wire system lacked a natural feedback from the road surface and the steering was almost too quick [25]. However, in the 2016 model feedback was implemented which made the steering experience better [26].

In 2013, 23 Q50 vehicles experienced problems with the steer by wire system. The steering actuator motors receives a current that is inversely proportional to the ambient temperature. These 23 vehicles were given a software update regarding a calibration issue with the power steering assistance which resulted in a possibility of sudden interrupts of the current to the actuator motors when the ambient temperature reached below zero. This did not only affect the steer by wire system but also the mechanical backup system which experienced delays. However, all the vehicles received a new update before any accidents occurred [4].

The 23 vehicles experiencing problems, out of all sold vehicles, got a lot of attention even though no accident occurred. This shows how doubtful the market is when it comes to safety in steer by wire systems. Accidents in a steer by wire system will happen sooner or later and when it does it will be important that there are track records enough to make a solid analysis of why the accident occurred, according to Andrew Del-Colle at Popular Mechanics [5].

This concludes that when designing a steer by wire system it is important to test the system in temperatures below zero and to ensure the market that the system is as safe as a mechanical system. But since this project only will be implemented on the test vehicle for now, cheaper and simpler components that does not work well in cold temperatures can be used as long as they are replaced in the production vehicle. Furthermore, this concludes that today there is no commercial vehicle that uses a steer by wire system without a mechanical backup system, which shows the complexity of developing a safe and reliable electronic steering system.

2.3 Fly by wire systems

Fly by wire systems used in aircrafts are related to the steer by wire systems used in cars. The system used in airplanes uses the same fundamental principle of transmitting the control signal by an electrical system rather than in a mechanical way. By using this method, the aircraft industry has been able to advance towards more advanced planes with the help of research from for example NASA, who has been using fly by wire systems on their spacecrafts. Airplanes designed to work in an intentional unstable condition, also known as relaxed stability, is one of the results of this research. The fly by wire system allows the plane to still be controllable in these conditions [6].

The aircraft industry uses safety specifications set by the US-military for stability- and control characteristics, per standard MIL-HDBK-1797 which is often referred to in modern applications [7]. The variables mentioned in the standard are based on the dynamics of an aircraft and depends on the size of control surfaces, wing surfaces etc. The specifications are set during the design of the airplane [7].

The fly by wire system also integrates with stabilization when control surfaces on the wings of the airplane malfunction. Furthermore, aspects like dangerous control inputs that otherwise would have resulted in a fatal crash can be damped and counteracted when switching to a fly by wire system. Benefits such as weight savings and efficiency savings, since mechanical and hydraulic systems are heavier than their electrical counterpart, is also experienced when switching to a fly by wire system [8].

Airplanes using fly by wire to increase the complexity of the airplanes is more common and have been used longer than steer by wire systems in cars. The most important challenge has been to make the system safe, making redundancy a main goal when designing these control systems [8].

2.4 Overview and review of the currently implemented system

The existing systems of the vehicle were studied to learn how the test vehicle was structured and to get an idea about which knowledge gaps that needed to be filled before starting the thesis work. After these knowledge gaps had been filled, the test vehicle was tested to see what it was that made the steering system that had been implemented by the staff before the thesis insufficient.

The existing steering system of the vehicle, se figure 4, consisted of a proportional controller programmed in a Controllino mega PLC unit located in the back of the vehicle. Controllino is an open source PLC developed for industrial use. It is programmed using the Arduino IDE. The current angle of the wheels and the angle of the steering controllers were measured by dividing a voltage using two potentiometers, one for the angle of the wheel and one for the angle of the steering controllers. The inputs worked as a number between 0 and 1024. The steering controllers had a mechanical linkage which means that they always were in the same position as each other. A servo motor was used to change the angles of the wheels. The servo motor had, besides the power supply, three inputs. One was a PWM input where higher duty cycle equalized higher speed of rotation. The two other pins decided if the servo should run forwards or backwards. How this was coded is shown and explained in appendix 1. The thesis workers have not coded this but have added some more comments to make it easier to understand.

The control of the throttle was not a part of this thesis but the ratio between the steering controller angle and wheel angle should depend on the current speed, to make it possible to drive straight at high speeds. That combined with the logic that different speeds might give different friction between the tires and the road the current speed also mattered. The measuring of the speed was handled by an Arduino Uno unit in the front of the vehicle. It measured the speed using two hall sensors with trigger wheels, one sensor for each wheel. This input was also a number between 0 and 1024. There was also a lidar in front of the vehicle to measure distance to objects in front of it. This was handled by the same Arduino Uno. How this was coded is shown and explained in appendix 2. The thesis workers have not coded this either but have added some more comments to make it easier to understand.

The structure of both the main program, appendix 1, and the lidar control and speed measurement, appendix 2, made them seem like they had been made under stress. Therefore, the code was analysed and fixed as flaws appeared throughout the thesis. The Controllino in the back of the vehicle and the Arduino in the front of the vehicle used ROS to communicate with each other. ROS stands for robot operating system and is a command based open source collection of software tools and libraries that are used to simplify the implementation of robotic systems. In this application, ROS Indigo was used to send the values of variables between programming units and between programming units and a computer. ROS was for the thesis workers unknown and therefore something that had to be learned. This was learned by going through the beginner tutorials offered by ROS.org [9]. In this implementation, ROS is used to send variables between the Controllino in the back and the Arduino in the front. Variables can also be displayed or saved to files.

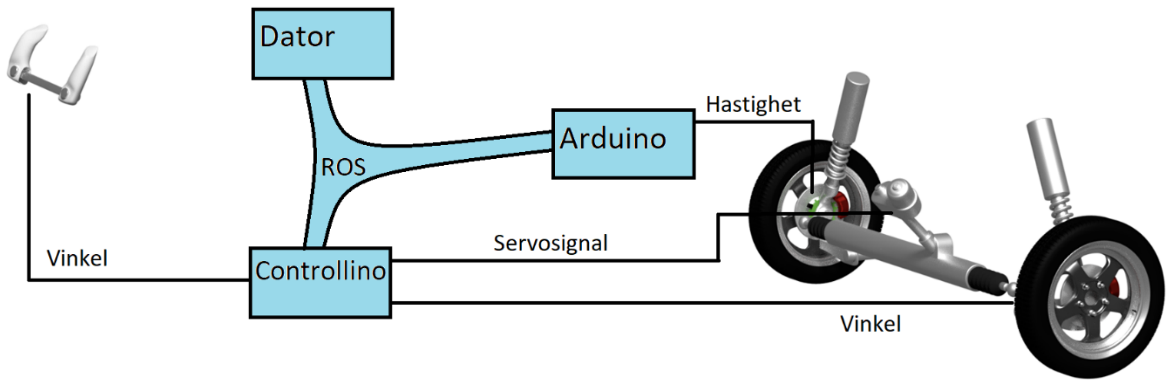

Figure 4: The electrical structure of the vehicle at the beginning of the thesis.

When test driving the test vehicle the analysis of the steering gave the following results.

- The steering was to slow.
- At higher speeds, it became hard to do soft turns.
- When the steering controllers were aimed straight forward the wheels were not.

Together this made it hard and unnatural to steer the vehicle. Since this test relied on user experience rather than measurements 5 persons from the staff were asked about how they experienced the steering. All of them agreed with the bulleted list in this section.

2.5 Theoretical overview regarding redundancy

A redundant system is a system where one or several units can break down without harming the performance of the system. National instruments explains that there are three different approaches to achieve redundancy, Standby redundancy, N modular redundancy and 1:N redundancy [10].

Standby redundancy is, according to National instruments, a system where you have an identical unit that can back up the system in case the primary unit breaks [10]. This will either need a third unit that is monitoring the system or the secondary system will have to monitor the system. However, the system prepared to back up the system is not fully synchronized with the primary unit which can create a rough transition when changing unit [10]. Therefore, standby redundancy is not suitable when implementing redundancy for the steering of a car.

Furthermore, National instruments describes the theory behind N modular redundancy [10]. This kind of redundancy consists of several units, typically between two and four, that all receive the same synchronized inputs. Thereafter, another unit at every sample analyses the outputs and decides which unit to use. The more units that are used the higher reliability and if the units are more than two the decision can be based on the most common output from the units. However, more units lead to a more expensive system. If all outputs are different it can be decided to either have one unit that is considered more trustworthy than the other and use its output or to turn the system off. To lower the risk of common mode errors these units, with the same task to solve, can be built on different hardware and implemented with different software. Triple modular redundancy is, according to National instruments, common in aerospace applications [10].

1:N redundancy, the last redundancy technique explained by National instruments, consists of one backup system that can replace one of several systems in case one of them breaks [10]. For this to be possible, the systems that can be replaced need to be similar enough so that one single unit can perform the tasks of any of these units. This lowers the cost of redundancy but adds complexity to the implementation as well as this solution only allows one unit to be broken at the same time.

From this information, N modular redundancy with three or more units implemented on different hardware and software was chosen as the best option for the steering system of a vehicle. To be able to safely decide which unit to use at a specific sample at least 2 units need to work and give the same output. If the probability of failure during an arbitrary hour for each unit in the redundant system while operating as a nonredundant system is found, by doing long time tests and using equation (1), the average numbers of hours before failure can be calculated as shown in equation (2). The numbers of units should be chosen so that the average number of hours before failure at least is higher than the expected life time of the vehicle.

Probability of failure at an arbitrary hour for a unit:

$$P_{(failure)} = \frac{\text{Numbers of hours where a failure occured}}{\text{Total number of hours for the test}} \tag{1}$$

Average numbers of hours before failure for the redundant system:

$$Average\ numbers\ of\ \text{h}ours\ to\ failure = \frac{1}{P_{\text{tot}(The\ whole\ system\ fails)}} = \tag{2}$$

$$= \frac{1}{P_{\text{tot}(All\ units\ fail)} + P_{\text{tot}(All\ but\ one\ unit\ fail)}} =$$

$$= \frac{1}{\prod_{k=1}^{n}\left(P_{k(failure)}\right) + \sum_{b=1}^{n}\left(\frac{\prod_{i=1}^{n}\left(P_{i(failure)}\right)\cdot\left(1-P_{b(failure)}\right)}{P_{b(failure)}}\right)}$$

$P_{\text{tot}(x)}$ = The probability that x occurs in the redundant system.

$P_{a(x)}$ = The probability that x occurs in unit a.

n = The amount of units.

As an example, if three units are used and they all have the same probability of failure, let us say 0.01. The average numbers of hours before failure would be 3355.7, which approximately equals four and a half months. If instead four units with the same probability of failure are used the result would be approximately 28 years and 9 months.

Another way of doing this would be to calculate the probability of failure at an arbitrary sample. This would give a more precise solution but it would also entail handling of larger numbers since the tests should run for at least a few weeks depending on how many failures that occurs during the test.

The safety features implemented in this thesis will then be adapted to operate on the output the redundant system has chosen to use. To improve the security more all connections shall be done with double wires and all components should be placed away from areas that are either hot or are vibrating. The components should also be spread out to lower the risk of having several components breaking from the same physical cause. If one unit breaks a warning should appear to the driver, if less than 2 units are working the engine should stop.

3 Designing the controller

This section explains how the controller was designed and why it was designed in this certain way.

3.1 Theoretical design regarding the controller

This section explains the theoretical design of the controller that was made based on the review of the currently implemented system. It was decided which inputs and output to use and how the controller should be structured.

It was decided that the current speed and target angle will be used as inputs and, of course, the angle of the wheels will be the output. The current speed will be used as an input because it will probably be different friction between the tires and the surface at different speeds. This was assumed by the thesis workers with the background that it takes a larger torque to turn a steering wheel when standing still compared to when driving if there is no power assisted steering implemented.

It was decided to do a controller with three different parameter settings, one for optimised for driving from 0 to 1.1 km/h, one for 1.1 to 11 km/h and one for above 11 km/h, see figure 5. The speed is measured and then the controller parameters is set depending on which interval the speed was in at that sample. This is done for each sample.

The small number of intervals was chosen due to the test vehicle’s lack of speed. Also, it would not be safe to drive faster on it due to the easily detached chairs and the lack of seatbelts. If, however, the steering felt unnatural when switching between parameters, more intervals would have been implemented. If this is transferred to the final vehicle more intervals should be used to get a more exact control system since that car will have a higher maximum speed.

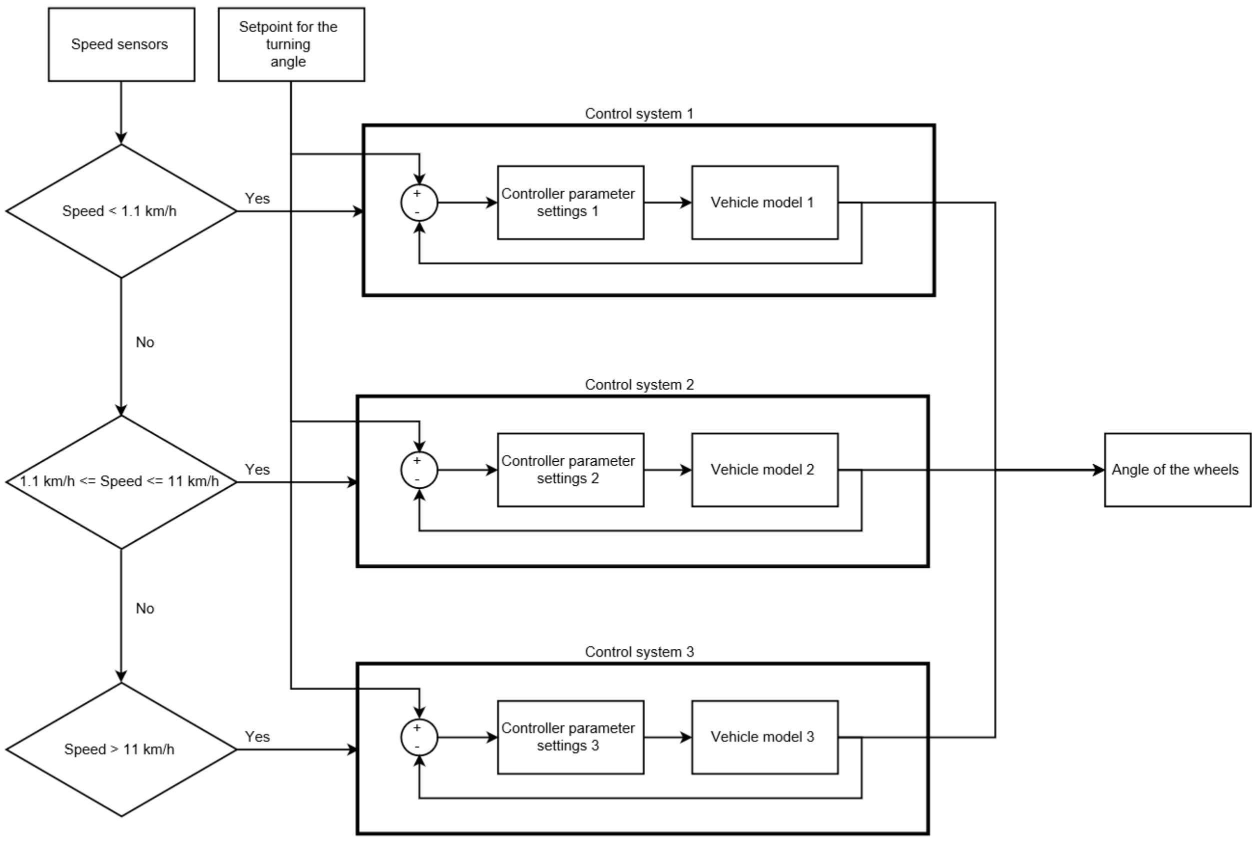

Figure 5: Flowchart for the controller for the steering. Three different parameter settings optimised for operating in one of the three speed intervals.

An alternative of having a controller with different parameters working at different speeds could be to implement an adaptive controller that continuously makes a system identification and adapts the controller parameters to the result. But that would have taken more computing power and might have slowed down the Controllino. This solution can however adapt to changes occurred due to different parameters than the speed, for example road conditions. These changes were assumed to be negligible compared to the changes depending on the speed, especially the changes between standing still and moving. If these assumptions had turned out to be wrong this solution would have been tested.

3.2 Code preparations

At the beginning of the thesis the ROS communication between the Arduino and Controllino at first did not work. This was due to lack of memory in the Arduino. The code was studied and the total dynamic memory usage was 85% of max divided like this.

- 3% units for the including of the lidar lite library.
- 10% units for the usage of the lidar lite library.
- 2% units for global variables.
- 67% units for the ROS communication.
- 3% for others.

The constants were declared as global variables and were because of that taking up unnecessary memory. This was fixed by declaring them using the define command that makes the compiler replace the variable with its value at every place where it is used instead of saving the variable in the dynamic memory. However, since the global variables only used 2% units this was not enough to fix the problem and therefore the Arduino Uno was replaced by an Arduino Mega that has more dynamic memory. The percentage of total dynamic memory usage was 27% in the Arduino Mega and the communication started working.

The existing main program did not take in consideration that the angles of the wheels and steering controllers had different ranges of motion. This needed to be implemented in the code so that the target angle of the wheels, based on the steering controller angle, is rescaled to match the range of motion for the wheels instead of the steering controllers. This was done by getting the range of motions from the mechanical team. When turning, the inner and outer wheel had different angles. An average value of the inner and outer wheel was used. Thereafter, a variable for the steering ratio was added to rescale the input value, see appendix 3. The steering ratio is calculated as shown in equation (3).

Steering ratio:

$$Steering\ ratio = \frac{Average\ maximum\ angle\ for\ the\ wheels}{Maximum\ angle\ for\ the\ steering\ controllers} \tag{3}$$

The ranges of motion for the steering controllers and wheels, used for the steering ratio, were the following:

- Maximum angle for the steering controllers: 45°
- Maximum angle for the inner wheel: 30.5°
- Maximum angle for the outer wheel: 22.8°
- Average maximum angle for the wheels: 26.5°

When the steering ratio was implemented, a problem occurred. The Controllino board was unknown to the Arduino IDE and therefore the code could not be uploaded. This was due to a faulty update by Controllino. When updating, the outdated version of the board was not deleted automatically. It had to be done manually before it was possible to use the updated board.

As explained in section 2.4, the steering was not calibrated correctly. To fix this the steering controllers and wheels were positioned straight forward. At this point the steering controller angle variable and the current angle variable should be zero. If not, an offset variable with a suitable value would be added. With the sensors in the right positions the offset values were calculated by taking an average value from at least a thousand measurements.

When the first calibration was done, the calibration did not last for long due to voltage drops over the potentiometers. The hardware team fixed this problem and a new calibration was done. The second calibration resulted in the following offset variables.

- Offset for the steering controllers: -9.45°
- Offset for the wheels: 13.30°

However, when this was implemented the steering interval itself was moved, not the middle point. Instead the variables representing the digital input values from the potentiometers at maximum and minimum angle were updated by manually moving the wheels and steering controllers fully to the sides and checking the input values. After this was done the calibration worked.

- Input at maximum angle, steering controllers = 246
- Input at minimum angle, steering controllers = 384
- Input at maximum angle, wheels = 241
- Input at minimum angle, wheels = 747

The implementation of the ratio between the ranges of motion for the wheels and the steering controllers as well as the calibration gave the expected result. When the steering controllers were straight forward so were the wheels, except for the standing deviation caused by the proportional controller. When the steering controllers, for example, were halfway to one side the setpoint for the angle of the wheels became halfway to the limit for the angle of the wheels. Note that the proportional controller had not been replaced yet.

The staff at Uniti suspected that the speed measurement was faulty and to make a control model for the steering that works at different speeds the speed measurement needs to be correct.

When analysing the code, it was found that the speed measurement didn’t use the time difference between the metal pieces passing by the hall sensors to calculate the speed. Instead the numbers of micro seconds that had passed since the system started was used which gave a faulty result, see appendix 2. This was changed and later in the thesis the speed measurement was moved to the main Controllino, see appendix 3.

The code measuring the speed was analysed and thereafter the travelled distance when driving 11 meters straight towards a wall was verified by measuring both with the hall sensors and the lidar attached to the front of the vehicle. With the assumptions that the internal clock of the Controllino as well as the distance measurement of the lidar were correct, the speed measurement would work if these two measurements got the same travelled distance, since the same mathematical constants are used when calculating the travelled distance and the speed, see appendix 2 and 3. The lidar and hall sensors got the same distance with an accuracy of ± 0.5 cm. With the mentioned assumptions, this concluded that the speed measurement was correct.

3.3 Developing mathematical models

This section explains how the mathematical models for the steering system was approximated.

3.3.1 Preparations and explanation of the approach

How to develop a model that is precise enough was not known by the thesis workers. When looking in a textbook targeting system identification, it was found that the system identification toolbox in MATLAB could be used for this [24]. The system identification toolbox in MATLAB is used to create mathematical models from collected input and output data with a specified sample period. By entering two sets of this data a model can be approximated. One dataset is used for making the model and the other dataset is used for verifying its accuracy. Various kinds of mathematical models can be approximated, for example a discrete transfer function that can be used when choosing controller parameters. When approximating a discrete transfer function, the toolbox lets you test various amounts of poles and zeroes manually. When a transfer function, or any other mathematical model, has been approximated the toolbox can display its accuracy measured in percentage. This is the accuracy of the mathematical model compared to the physical system when they both react to the validation input. The toolbox can also show how the mathematical model would have reacted compared to the physical system if the inputs from the verification dataset were applied to it. The poles and zeroes for the mathematical function can be displayed in a graph by the toolbox.

When developing a model, the signals sent to the servo, the current angle of the wheels and the current speed were sampled by saving the ROS messages to a file. However, there was no ROS message for the signal sent to the servo, the variable called SteeringAdjustment, so that was implemented by copying the structure of the previously implemented ROS messages, see appendix 3.

The following commands was used in the terminal on the operating system Ubuntu 14.04 to save ROS messages to files.

roscore: Starts ROS.
rosrun rosserial_phyton serial_node.py /dev/ttyACM0: Starts a ros node on USB port ttyACM0.
rostopic echo -p /MessageName > fileName.txt: Choses a message to save to a file.

To make a model the sampling needed to be done with a specific sample period. The shorter sample period the more precise result. However, the sample period needed to be high enough so that the Controllino could handle it and the shorter it was the higher would the percentage deviation caused by using an if-statement be. 10ms was chosen as a compromise between these factors.

Since this program ran the most important systems of the car an interrupt might have caused problems to the other parts of the code. Therefore, it was decided to use an if-statement with the sample time as its condition instead of an interrupt. This will not cause problems for the other parts of the code but the sample period will wary depending on the time it takes for the Controllino to execute the rest of the code in the loop.

Inside this if-statement all the functions that update outputs to the motors and publishes the ROS messages for the signal to the servo and the current angle of the wheels were put. The previously declared interval for publishing ROS messages, see appendix 1, were no longer used. The current speed was however, as mentioned in section 2.4, measured by the Arduino. Therefore, that ROS message could not easily be synced. It was decided to only lower the publish rate from 20 to 8ms. The Controllino then read the latest ROS message for the current speed and published it synchronized with the other messages. Since the mechanical acceleration is assumed to be slow compared to the sample period it is assumed that the lack of synchronization between the speed measurement and the other messages will not harm the modelling. Furthermore, the exact speed at a certain sample is not needed as long as it can be seen that the speed is in the correct interval when collecting data for the models.

To minimize the deviation, the standard time it took for the Controllino to execute everything in the loop of the Controllino except the updating of outputs and publishing of ROS messages, that is everything outside of the if-statement with the sample period as its condition, was measured and the condition in the if-statement was adapted according to that. The time it took was 0.64ms. Therefore, the condition of the if-statement was set to 9.68ms, see appendix 3. This equalises in a sample period of 10 ± 0.32ms with the currently implemented code. The deviation of the sample period will however change when other code is added outside of the if-statement but this change made the real sample interval closer to the desired one.

First, a model was done by collecting the data while driving at different speeds between 0 and 14.5 km/h. When this data was used in the system identification toolbox a precise model could not be found which shows that the premade assumptions regarding the impact of speed differences were correct.

Six collections of data were then generated, two for each interval where one was used to estimate a model and the other one to verify it. Different routes were driven when collecting the estimation data and validation data to make sure the model did not only work for one type of route. The data were put into MATLAB’s system identification toolbox and thereafter the transfer functions with the highest accuracy that was controllable by a PID-controller were chosen by testing different numbers of poles and zeroes. To test if the found models were controllable by a PID-controller the static gain for the transfer functions were calculated. If the static gain for a transfer function equals zero that model cannot be controlled by a PID-controller. Thereafter the PID tuner in Simulink was used to see if any PID-parameters that would give a stable system could be found.

When approximating the mathematical models, it was noticed that the accuracy increased with increased order of the transfer function until a certain point. Beyond that point, the accuracy started to decrease. For each speed interval, the model with the highest possible accuracy was chosen.

After two out of the three models had been made the test vehicle was taken to a fair by the staff at Uniti. An accident occurred there and several components in the test vehicle were burned. The staff decided to rewire the test vehicle to make it safer and more reliable. This delayed the modelling and after the rewiring was done the code had to be adapted to the changes in the wiring.

After the rewiring, the speed measurement was taken care of by the main Controllino. The code for the speed measuring was transferred and adapted from the Arduino code, appendix 2, to the main code, see appendix 3. This implemented interrupts in the main code but when test driving it had no impact on the steering.

3.3.2 The model operating from 0 to 1.1 km/h

The collected data is shown in figure 6. As shown in figure 7, this transfer function, shown in equation (4), has an accuracy of 76% compared to the physical system when reacting to the same inputs. The poles and zeroes for the transfer function is shown in figure 8.

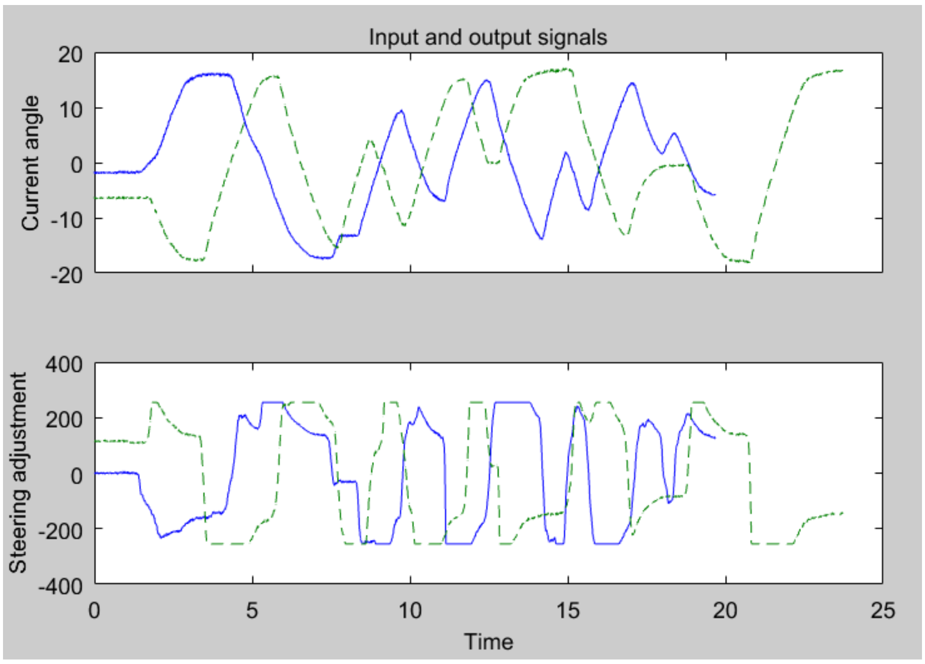

Figure 6: Collected data from the 0 to 1.1 km/h measurements. Blue is estimation data and green, dashed, is validation data. The lower graph is the signals sent to the servo and the higher graph is the angle of the wheels at the same time. The diagram was made using MATLAB.

Transfer function from 0 to 1.1 km/h:

$$H_{1\mathrm{Servo}(z)} = \frac{0.01936 \cdot z^{-1} - 0.04035 \cdot z^{-2} + 0.02078 \cdot z^{-3}}{1 - 1.608 \cdot z^{-1} + 0.7398 \cdot z^{-2} - 0.2518 \cdot z^{-3} + 0.0325 \cdot z^{-4} + 0.08851 \cdot z^{-5}} \tag{4}$$

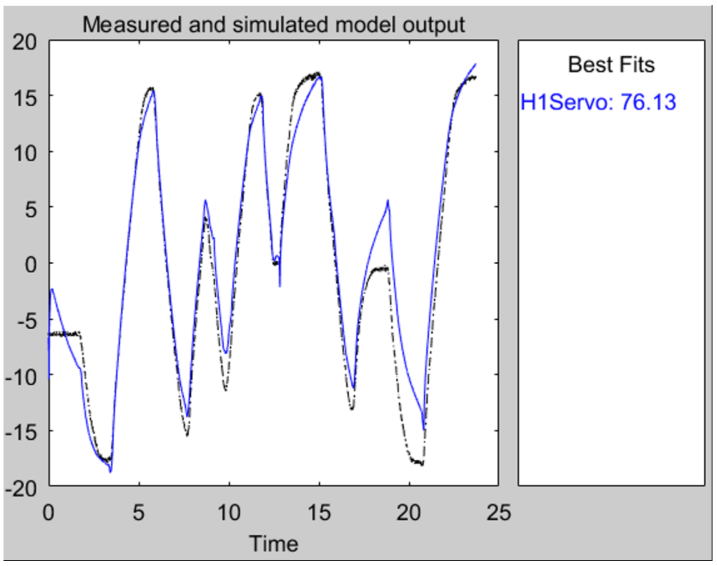

Figure 7: The physical system, black and dashed, and the transfer function operating from 0 to 1.1 km/h, blue, reacting to the same inputs. The diagram was made using MATLAB.

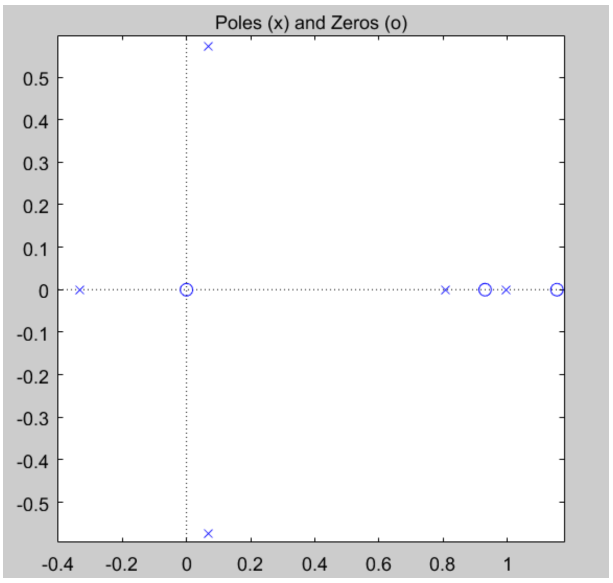

Figure 8: Poles and zeroes for the transfer function operating from 0 to 1.1 km/h. The diagram was made using MATLAB.

3.3.3 The model operating from 1.1 to 11 km/h

The collected data is shown in figure 9. As shown in figure 10, this transfer function, shown in equation (5), has an accuracy of 79% compared to the physical system when reacting to the same inputs. The poles and zeroes for the transfer function is shown in figure 11.

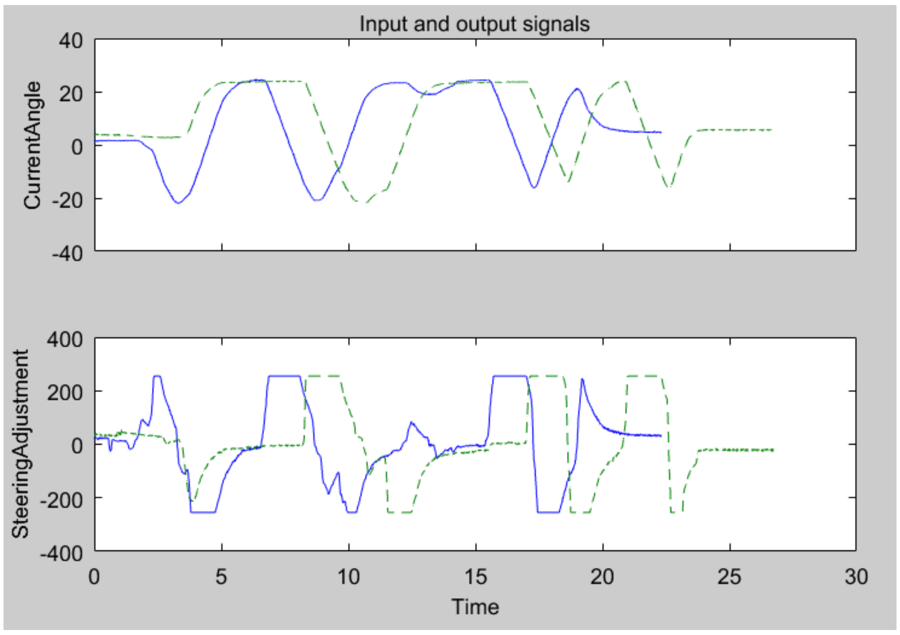

Figure 9: Collected data from the 1.1 to 11 km/h measurements. Blue is estimation data and green, dashed, is validation data. The lower graph is the signals sent to the servo and the higher graph is the angle of the wheels at the same time. The diagram was made using MATLAB.

Transfer function from 1.1 to 11 km/h:

$$H_{2Servo(z)} = \frac{0.05172 \cdot z^{-1} - 0.1062 \cdot z^{-2} + 0.05442 \cdot z^{-3}}{1 - 1.117 \cdot z^{-1} + 0.02384 \cdot z^{-2} - 0.1682 \cdot z^{-3} - 0.2219 \cdot z^{-4} + 0.4837 \cdot z^{-5}} \qquad (5)$$

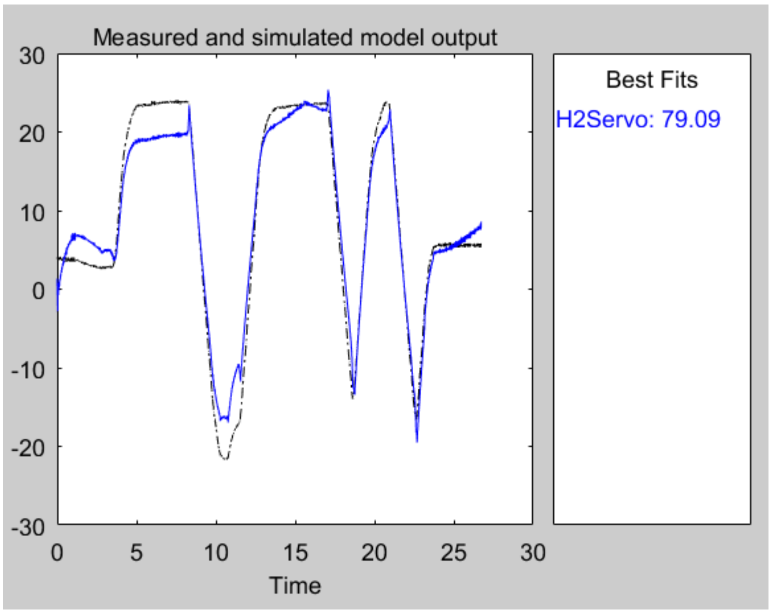

Figure 10: The physical system, black and dashed, and the transfer function operating from 1.1 to 11 km/h, blue, reacting to the same inputs. The diagram was made using MATLAB.

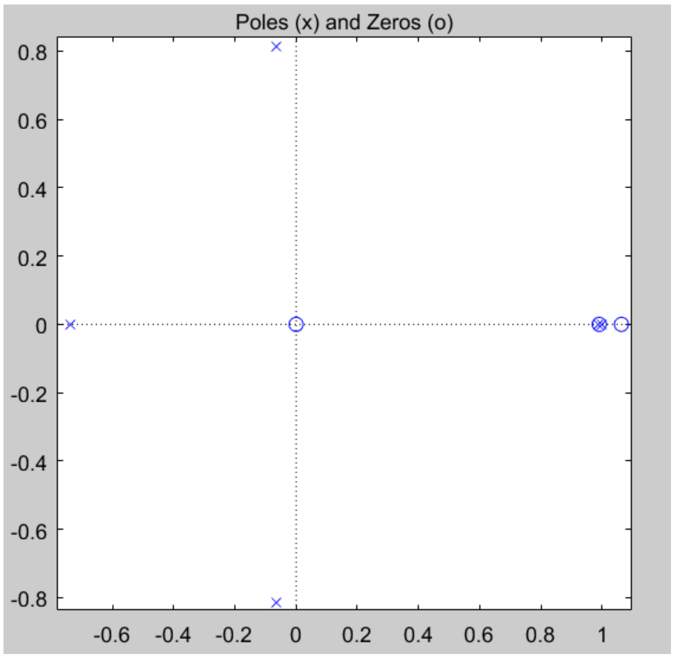

Figure 11: Poles and zeroes for the transfer function operating from 1.1 to 11 km/h. The diagram was made using MATLAB.

Figure 10 shows that the model gives undershoots even though the physical system does not. This is an undesired behaviour but when testing models with other orders no model without undershoots but with equally high accuracy was found.

3.3.4 The model operating above 11 km/h

The collected data is shown in figure 12. As shown in figure 13, this transfer function, from equation (6), has an accuracy of 77% compared to the physical system when reacting to the same inputs. The poles and zeroes for the transfer function is shown in figure 14.

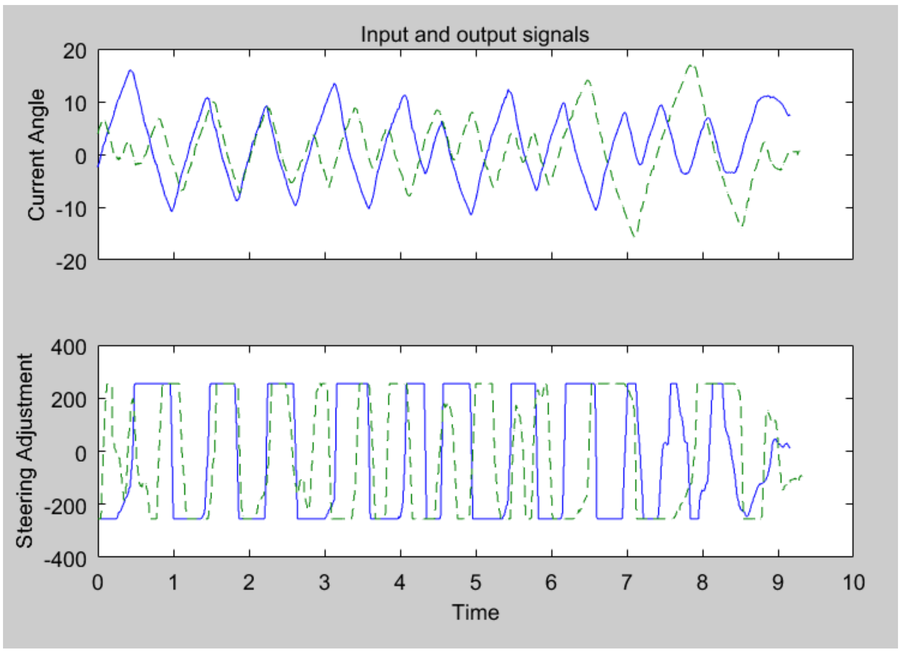

Figure 12: Collected data from measurements above 11 km/h. Blue is estimation data and green, dashed, is validation data. The lower graph is the signals sent to the servo and the higher graph is the angle of the wheels at the same time. The diagram was made using MATLAB.

Transfer function above 11 km/h:

$$H_{3\mathrm{Servo}(z)} = \tag{6}$$

$$= \frac{-0.003271z^{-1} + 0.003145z^{-2}}{1 - 0.359z^{-1} - 1.665z^{-2} - 0,4268z^{-3} + 1.185z^{-4} + 1.123z^{-5} + 0.03524z^{-6} - 0.9554z^{-7} - 0.4331z^{-8} + 0.4969z^{-9}}$$

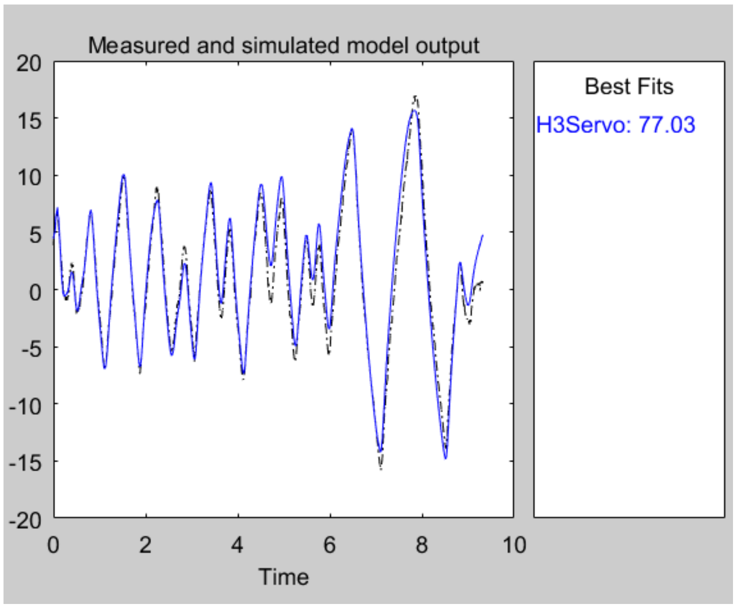

Figure 13: The physical system, black and dashed, and the transfer function operating above 11 km/h, blue, reacting to the same inputs. The diagram was made using MATLAB.

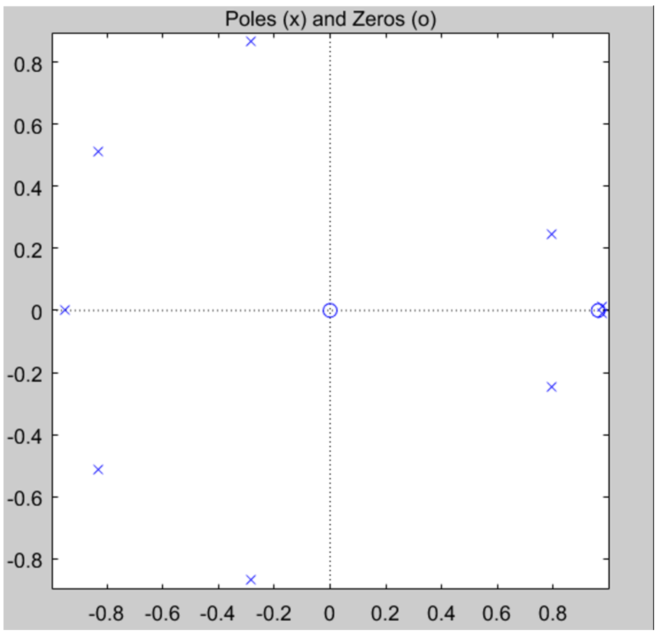

Figure 14: Poles and zeroes for the transfer function operating above 11 km/h. The diagram was made using MATLAB.

All of the models have an accuracy of below 80% therefore the controller parameters for these models, defined in section 3.4, might have to be adjusted slightly to achieve the desired behaviour, this behaviour is specified in section 3.4.

3.4 Choosing the controller parameters

This section explains how the controller parameters were chosen and how the theoretical steering responses turned out.

3.4.1 Explanation of the approach

The first approach was to make the controller using the pole placement method. It was however hard to choose optimal pole placements because of the relatively high number of poles and zeroes on the models. Therefore, it was chosen to use a PID-controller instead [23]. MATLAB's PID tuner in Simulink can be used to optimise PID-parameters. This was in the PID-tuner done by dragging two parameters, response time and transient behaviour, and watch how the step response changed. A general block diagram is shown in figure 15.

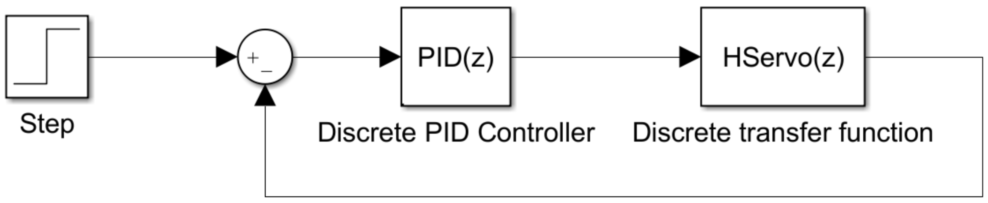

Figure 15: A general block diagram for the controller. This was done using MATLAB.

When using the PID tuner a step response that minimized the rise time, overshoot and undershoot was searched for since those factors could have an unnatural impact on the steering. The settling time however, was seen as less important as long as the deviation was relatively small so that the driver would not notice it. The used transfer function that explains the control parameters in section 3.4.2 to 3.4.4 is shown in equation (7)**.** All the variables and parameters in section 3.4.2 to 3.4.4 except the undershoot were displayed in the PID tuner. The undershoot was calculated by analysing the step response, 0-1°.

The transfer function for the PID-controller:

$$H_{(z)} = k_P + k_I \cdot \mathrm{h}\frac{z}{z-1} + k_D \frac{N}{1 + N \cdot \mathrm{h}\frac{z}{z-I}} \tag{7}$$

h = sample period [s].

k_P = proportional gain.

k_I = integral gain.

k_D = derivate gain.

N = filter factor.

3.4.2 The controller with parameters operating from 0 to 1.1 km/h

The control parameters are displayed in table 1 and the performance and robustness variables in table 2. The step response, 0-1°, can be seen in figure 16. A response from a 0 to 15° step can be seen in figure 17.

Control parameters	Values
k_P	-29.4106
k_I	-12.1274
k_D	3.7603
N	7.512

Table 1: Control parameters for the controller with parameters operating from 0 to 1.1 km/h.

Performance and robustness variables	Values
Rise time	0.72s
Settling time	1.24s
Overshoot	1.53%
Undershoot	7.59%
Gain margin	15.8dB at 17rad/s
Phase margin	74.8° at 2.14rad/s
Closed-loop stability	Stable

Table 2: Performance and robustness variables for the controller with parameters operating from 0 to 1.1 km/h.

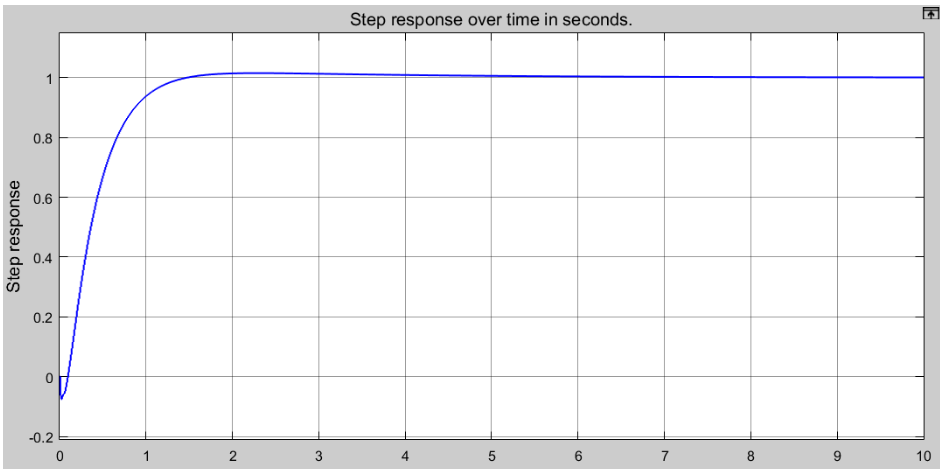

Figure 16: Step response, 0-1°, for the controller with parameters operating from 0 to 1.1 km/h. This was done using MATLAB.

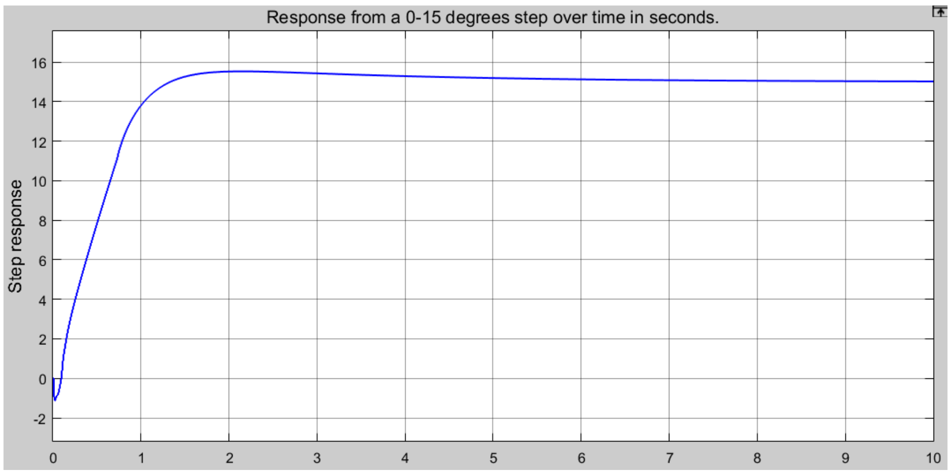

Figure 17: Step response, 0-15°, for the controller with parameters operating from 0 to 1.1 km/h. This was done using MATLAB.

3.4.3 The controller with parameters operating from 1.1 to 11 km/h

The control parameters are displayed in table 3 and the performance and robustness variables in table 4. The step response, 0-1°, can be seen in figure 18. A response a from 0 to 15° step can be seen in figure 19.

Control parameters	Values
k_P	-7.5771
k_I	-7.3581
k_D	0.42957
N	15.3967

Table 3: Control parameters for the controller with parameters operating from 1.1 to 11 km/h.

Performance and robustness variables	Values
Rise time	0.92s
Settling time	5.31s
Overshoot	0.189%
Undershoot	9.77%
Gain margin	16.7dB at 9.88rad/s
Phase margin	59° at 1.59rad/s
Closed-loop stability	Stable

Table 4: Performance and robustness variables for the controller with parameters operating from 1.1 to 11 km/h.

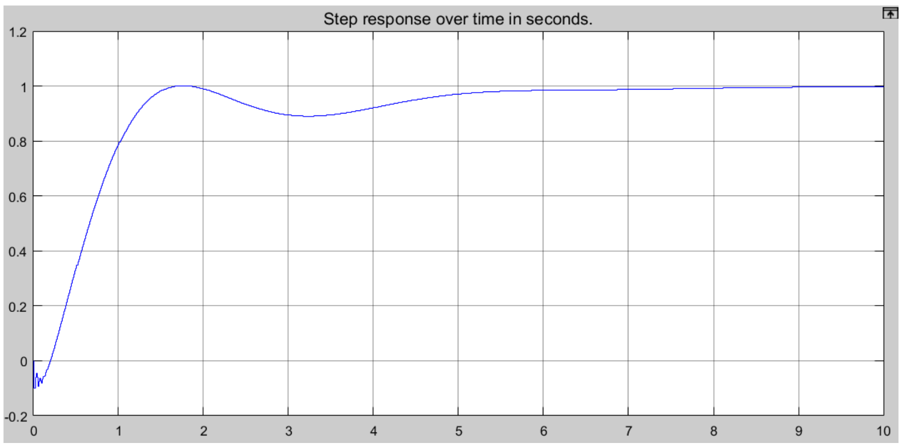

Figure 18: Step response, 0-1°, for the controller with parameters operating from 1.1 to 11 km/h. This was done using MATLAB.

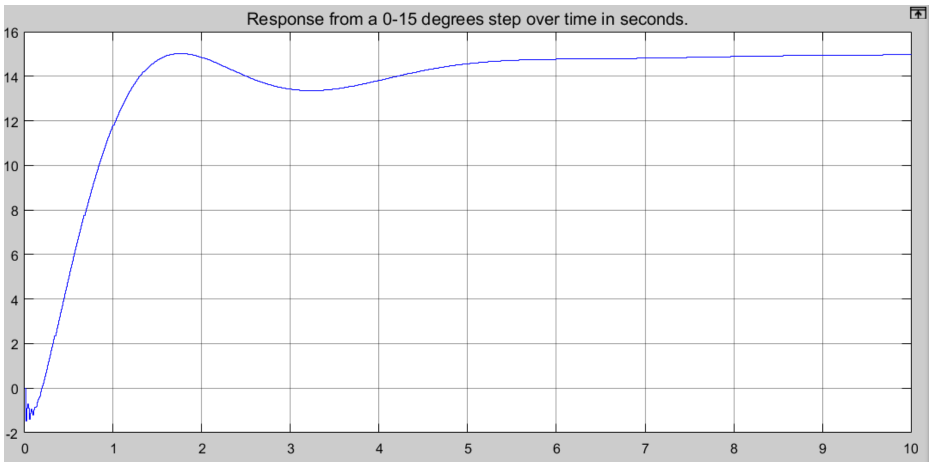

Figure 19: Step response, 0-15°, for the controller with parameters operating from 1.1 to 11 km/h. This was done using MATLAB.

3.4.4 The controller with parameters operating above 11 km/h

The control parameters are displayed in table 5 and the performance and robustness variables in table 6. The step response, 0-1°, can be seen in figure 20. A response a from 0 to 15° step can be seen in figure 21.

Control parameters	Values
k_P	-39.9944
k_I	-60.3186
k_D	0.90866
N	35.8523

Table 5: Control parameters for the controller with parameters operating above 11 km/h.

Performance and robustness variables	Values
Rise time	0.2s
Settling time	0.35s
Overshoot	1.34%
Undershoot	0%
Gain margin	18.7dB at 259rad/s
Phase margin	76.2° at 8.01rad/s
Closed-loop stability	Stable

Table 6: Performance and robustness variables for the controller with parameters operating above 11 km/h.

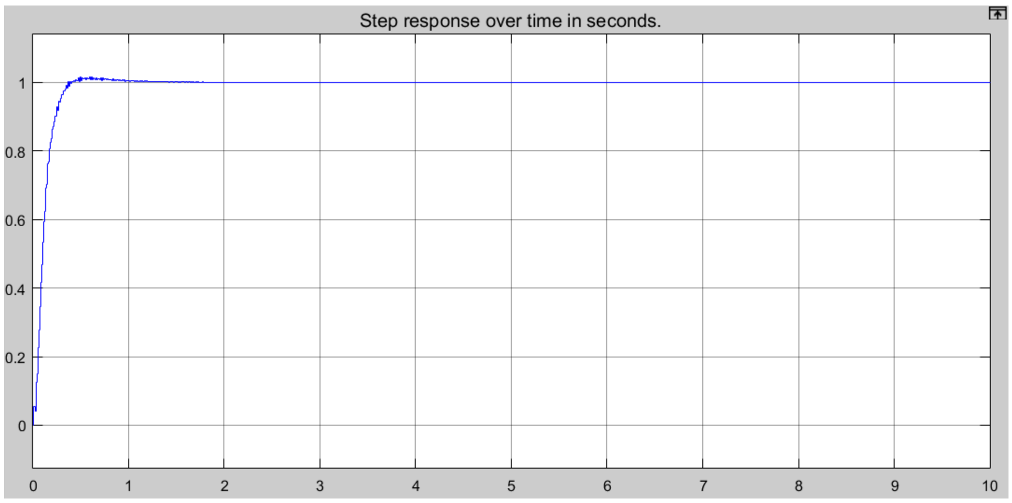

Figure 20: Step response, 0-1°, for the controller with parameters operating above 11 km/h. This was done using MATLAB.

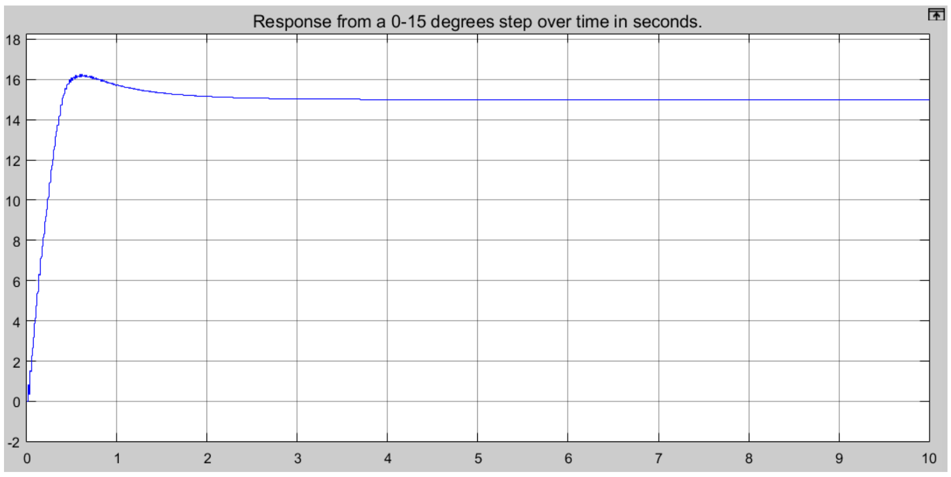

Figure 21: Step response, 0-15°, for the controller with parameters operating above 11 km/h. This was done using MATLAB.

When analysing the controller, it can be seen that the rise time for the system operating above 11 km/h is significantly lower than the rise time for the other two. This shows that the difference in dynamic when driving at various speed should not be ignored. The size of the rise time shows that the steering might still feel to slow, this was tested later in the thesis, section 8.1. Also, the overshoot, and in the 0 to 1.1 km/h and 1.1 to 11 km/h case also the undershoot, could not be eliminated completely and this might have a negative effect on the steering experience. If this turns out to be a problem when testing the system in section 8.1 the parameters can be adjusted slightly since the mathematical models defined in section 3.3 have an accuracy of below 80%. If that does not help, a stronger and faster servo would be needed. The step response in the 1.1 to 11 km/h case, figure 18, shows that the corresponding model was harder to control compared to the other two. This can be due to the models zero-position that is located close above one.

The step responses show a rather low amount of time constants which tells that the mathematical models might have been overparameterized. However, when testing all combinations of poles and zeroes with a lower order than the chosen models no estimation with a higher accuracy could be found.

4 Designing the speed dependency

The small range of motion for the steering controllers made it hard to do small angle adjustments often needed at high speeds. Therefore, the steering behaviour need to depend on the speed. This section explains how the relation between the angle of the steering controllers, the speed and the setpoint for the angle of the wheels was designed and why it was designed in this certain way.

4.1 Obtaining baselines for the speed dependency

To be able to develop a mathematical function for the speed dependency studies were done to find out how sharp the maximum wheel angles needed to be at different speeds.

4.1.1 Researching road design

To get a minimum baseline for the turning angles needed studies were made on how roads are designed today. In Sweden, Trafikverket is the department supervising road construction and design. Trafikverket have guidelines targeting road design on their webpage describing the correlation between road speed and minimum curve radius, see equation (8) [15]. Note that the equation mentioned in the paper is faulty written and differ on the placement of the gravity, g. This error was found and fixed by testing out the equation until realistic values were found.

Minimum curve radius depending on the speed:

$$r > \frac{(\frac{v}{3.6})^2}{(0.28e^{-0,0096v} + E) \cdot g} \tag{8}$$

$r =$ Curve radius. [m]

$v =$ Speed. [km/h]

$E =$ Road banking [%]

$g =$ Gravity [m/s^2]

Thereafter, when knowing the curve radius, the wheel angle was calculated and a plot describing the relation between speed limits and maximum curve radius was made. The road banking was set to 0% since equation (8) gives the radius in the mathematical plane of the ground surface, not in the same plane as the road itself. If the road banking was set to anything else than 0% the radius would be smaller but only in the plane of the mathematical ground surface. The radius in the plane of the road would be the same and it is this radius that the steering angle depend on.

In figure 22 it is illustrated how the angle of the wheel when knowing the curve radius, from equation (8)**,** was calculated. The calculations are then shown in equation (9)**.** Figure 23 describes this relation between speed limitations and maximum steering angle for vehicles following the curves of the roads.

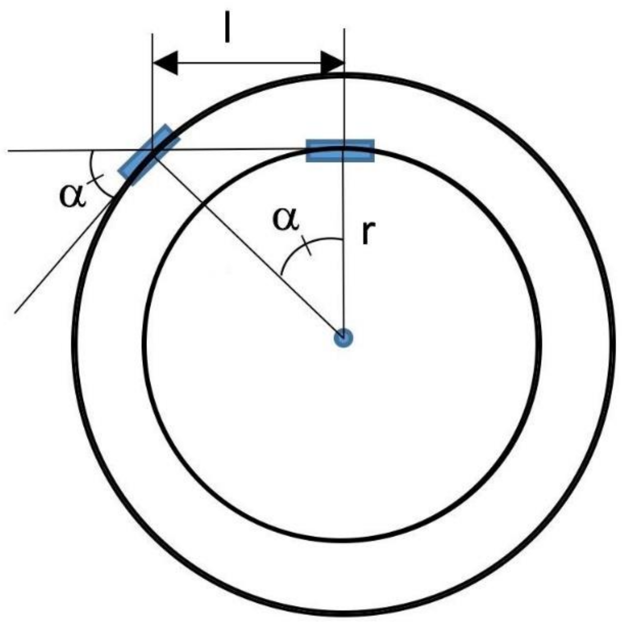

Figure 22: How to calculate the steering angle when the curve radius is known. The figure shows the circular curve and the inner wheels of the vehicle. r is the curve radius, l is the length of the car and α is the steering angle.

The steering angle as a function of the speed limit made by using equation (8)**:**

$$\alpha = \arctan\left(\frac{l}{r}\right) = \arctan\left(\frac{l}{\frac{\left(\frac{v}{3.6}\right)^2}{(0.28 \cdot \mathrm{e}^{-0.0096v} + E) \cdot g}}\right) = \tag{9}$$

$$= \arctan\left(\frac{l \cdot (0.28 \cdot \mathrm{e}^{-0.0096v} + E) \cdot g}{\left(\frac{v}{3.6}\right)^2}\right)$$

The length of the vehicle, l, is 1.838m. The road banking, E, is set to 0%.
The gravity, g, is 9.82 m/s^2.

$$\alpha = \arctan\left(\frac{65.4968 \cdot \mathrm{e}^{-0.0096v}}{v^2}\right)$$

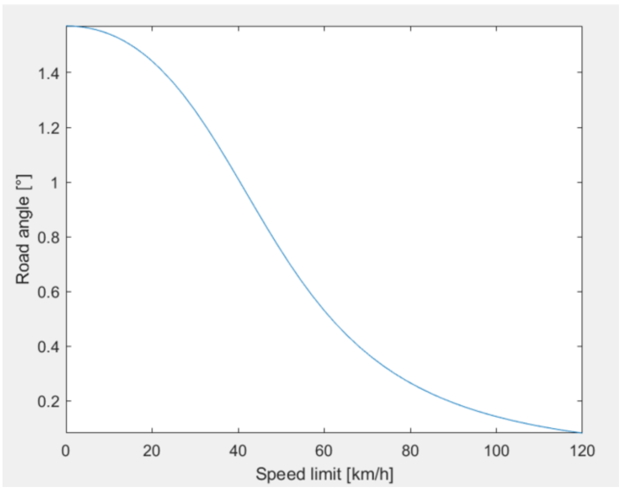

Figure 23: The steering angle needed to handle the maximum allowed curve radius for roads with a certain speed limit. The graph was made using MATLAB.

Figure 23 shows that the maximum steering angle depending on the speed limitation is small and should not be a problem for the maximum angle of the vehicle.

4.1.2 Physical tests while driving

While driving a rather small car, a Subaru Justy, two tests were done.

The first test was made to find out how fast it is comfortable to drive when steering fully to one side. This was done by turning fully to the side and accelerating until the centripetal force got uncomfortable high. At 10 km/h the centripetal force got uncomfortable high and the wheels started to slide outwards from the circle.

The second test was made to find out which steering angles were used while doing sharp turns while driving at a highway in 110 km/h. The turn should be as sharp as possible while still being safe. By measuring the time to do a lane change, from driving fully to the left in the right lane to driving fully to the left in the left lane, at a fixed speed knowing the width of the lane the angle can be calculated according to equation (10)**,** as illustrated in figure 24**.** The lane width of a highway is 3.5m [20].

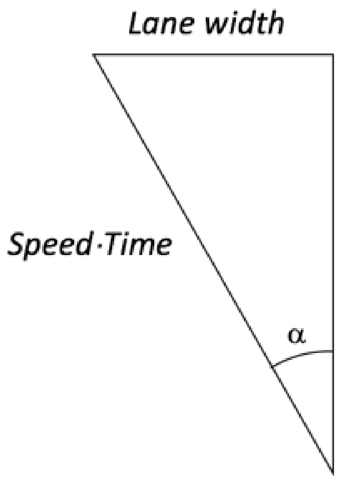

Figure 24: Illustration of how to calculate the angle when doing a lane change.

Calculating the achieved steering angle when doing a lane change.

$$\alpha = \arcsin\left(\frac{l}{\frac{s}{3.6} \cdot t}\right) \tag{10}$$

$\alpha =$ Steering angle. [°]

$l =$ Lane width. [m]

$s =$ Speed. [km/h]

$t =$ Time. [s]

The lane width, l, is 3.5m and the speed, s, is 110km/h. The fastest achieved time for doing a lane change while still driving in a safe way, t, was 1.41s. The corresponding angle was rounded up since other drivers might be able to perform higher angles.

$$\alpha = \arcsin\left(\frac{3.5}{\frac{110}{3.6} \cdot t}\right) = [t = 1.41s] = 4.65672° \approx 5°$$

4.1.3 Research regarding evasive manoeuvres

The moose test included in the standard ISO3888-2 is a test conducted on vehicles [21]. In Sweden, it has been conducted by a journal named Teknikens värld since the 1970s [14]. This journal does tests on a variety of vehicles of all sizes. The common factor is that they setup a standardized course and run at a high speed, compared to the course setup, through the course. The speed is increased until the car fails the test by either running over cones or behaves unpredictable. This course responds to the track the car should follow when doing an evasive maneuver. When a car is failing the test, the common practice is that the car manufacturer takes the data and try to refine the car and correct the issue.

By analysis a video on YouTube of Teknikens värld performing a moose test with a Jeep Grand Cherokee, [19], the angle needed to do an evasive manoeuvre at a certain speed was calculated by noticing how many degrees on the steering wheel the driver achieved during the test. The course setup is the same for all cars independent of the car’s size. The used car has a steering ratio of 17.5:1 [16]. It was not taken into consideration that the performed steering angle might cause a rollover since the dynamics of the vehicle were not finalised as well as an electronic stability control, ESC, should handle this kind of problems.

In the video, the steering wheel was turned 270° when driving at 60km/h [19]. With a steering ratio of 17.5:1 this equals a steering angle of 15.4°.

4.2 Developing a mathematical function for the speed dependency

With the obtained baselines for the speed dependency in regard a mathematical function was designed. After positioning points in the coordinate system, the curve fitting tool in MATLAB was used to estimate a mathematical function.

The curve fitting tool in MATLAB is used to fit mathematical curves to points in a two or three-dimensional coordinate system. The points are added as separate vectors for each coordinate axis. Thereafter, the tool lets you choose between various kinds of mathematical functions, for example a polynomial function. When using this type of function the tool lets you test various degrees for the approximated function. When an approximation has been made, the function is displayed both visually and mathematically. The accuracy of the function is also displayed.

After the estimation was done it was checked so that the requirements from the road regulations were met and that the wheel angle used in the moose test for an evasive manoeuvre at a certain speed could be achieved at that speed.

Since it was comfortable to drive with the wheels fully turned at 10 km/h and below the setpoint for the angle of the wheels were chosen to be equal to the angle of the steering controllers in that interval.

When driving faster the range of motion for the wheels need to decrease to make it possible to achieve the small angles needed to manoeuvre the vehicle but to make it possible to do sharper evasive manoeuvres and also preventing unintended movements in the hands from interfering with the direction of the car, it was chosen to decrease the sensitivity more when being close to the middle point of the steering controllers range of motion while the speed is increased.

Looking at the maximum speed of 110km/h the results from the highway tests were used. The highway test resulted in a maximum angle of around 5 degrees while switching lanes. This is without a margin for evasive manoeuvres. This value was increased by 75% to include the ability to do evasive manoeuvrings at this speed. With a 75% higher maximum, the setpoint was set to 8,75°. This is the maximum speed for the function. It was chosen that if the speed is increased beyond 110 km/h the same function as when driving at 110 km/h would be used.

An aspect that was taken into consideration were that the range of motion for the wheels should not change too quickly since that could make it more complex for the driver. Also, the range of motions should not drop too much at lower speeds since that could limit the driver's possibility to navigate in, for example, cramped cities. With these characteristics in regard it was chosen to have a function similar to an arctangent where 85% of the range of motion for the wheel were kept at 35km/h. This results in a maximum wheel angle of 23,965° at that speed. To retain the symmetry in the function, the range of motion while driving at 85 km/h was chosen to be 15% above the minimum range of motion. That equals 11.435°.

To decrease the sensitivity more when being close to the middle point of the steering controllers range of motion when the speed is increased it was chosen to transit from the linear relation at 10 km/h and lower to having a function similar to a tangent where one third of the range of motion for the wheels operates within the first half of the range of motion for the steering controllers at 110 km/h. Leaving two thirds of the range of motion for the wheels to be operated by the second half of the range of motion for the steering controllers. This was chosen as a compromise between having the highest possible accuracy when driving straight forward and not having impractically low accuracy at the end of the range of motion for the steering controllers.

More points that were calculable from the existing points where added in a way that kept the symmetry of the function.

At 10 km/h and lower the setpoint for the angle of the wheels is equal to the angle of the steering controllers. The function describing the relation between setpoint, steering controller angle and speed when operating from 10 km/h to 110 km/h is shown in equation (11) and in figure 25. The function describing the relation between the setpoint and the steering controller angle when operating above 110 km/h is shown in equation (12) and figure 26. Both these functions have an accuracy of 100% which means that the calculated points can be found in the functions without any rounding.

The setpoint for the angle of the wheels as a function of the speed and the steering controller angle when operating between 10 and 110 km/h:

$$f_{(x,y)} = p_{00} + p_{10}x + p_{01}y + p_{20}x^2 + p_{11}xy + p_{02}y^2 + p_{30}\mathrm{x}^3 \\ +p_{21}x^2y + p_{12}xy^2 + +p_{03}y^3 + p_{40}x^4 + p_{31}x^3y \\ +p_{22}x^2y^2 + p_{13}xy^3 + p_{04}y^4 \tag{11}$$

f = Setpoint for the angle of the wheels. [°]

x = Steering controller angle. [°]

y = Speed. [km/h]

$p_{00} = 1.22 \cdot 10^{-15}$
$p_{10} = 0.9872$
$p_{01} = -2.126 \cdot 10^{-16}$
$p_{20} = -1.826 \cdot 10^{-17}$
$p_{11} = 0.003717$
$p_{02} = 1.936 \cdot 10^{-17}$
$p_{30} = -2.055 \cdot 10^{-5}$
$p_{21} = -4.383 \cdot 10^{-19}$
$p_{12} = -0.0002579$
$p_{03} = -3.516 \cdot 10^{-19}$
$p_{40} = 3.223 \cdot 10^{-20}$
$p_{31} = 2.055 \cdot 10^{-6}$
$p_{22} = 4.636 \cdot 10^{-21}$
$p_{13} = 1.433 \cdot 10^{-6}$
$p_{04} = 1.739 \cdot 10^{-21}$

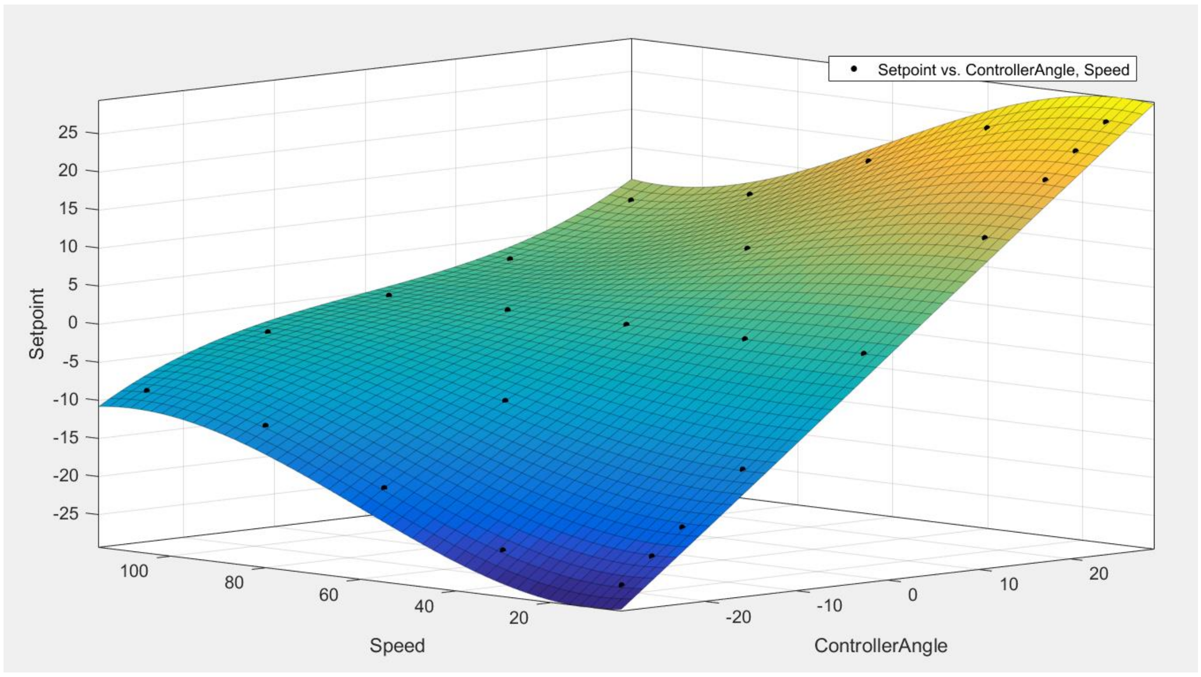

Figure 25: Main graph for how the setpoint depend on the steering controller angle and the speed when operating between 10 and 110 km/h. The graph was made using MATLAB.

The setpoint for the angle of the wheels as a function of the steering controller angle when driving over 110 km/h:

$$f_{(x)} = p_0 + p_1 x + p_2 x^2 + p_3 x^3 \tag{12}$$

f = Setpoint for the angle of the wheels. [°]

x = Steering controller angle. [°]

$p_0 = 4.643 \cdot 10^{-16}$
$p_1 = 0.0002055$
$p_2 = -1.949 \cdot 10^{-18}$
$p_3 = 0.1824$

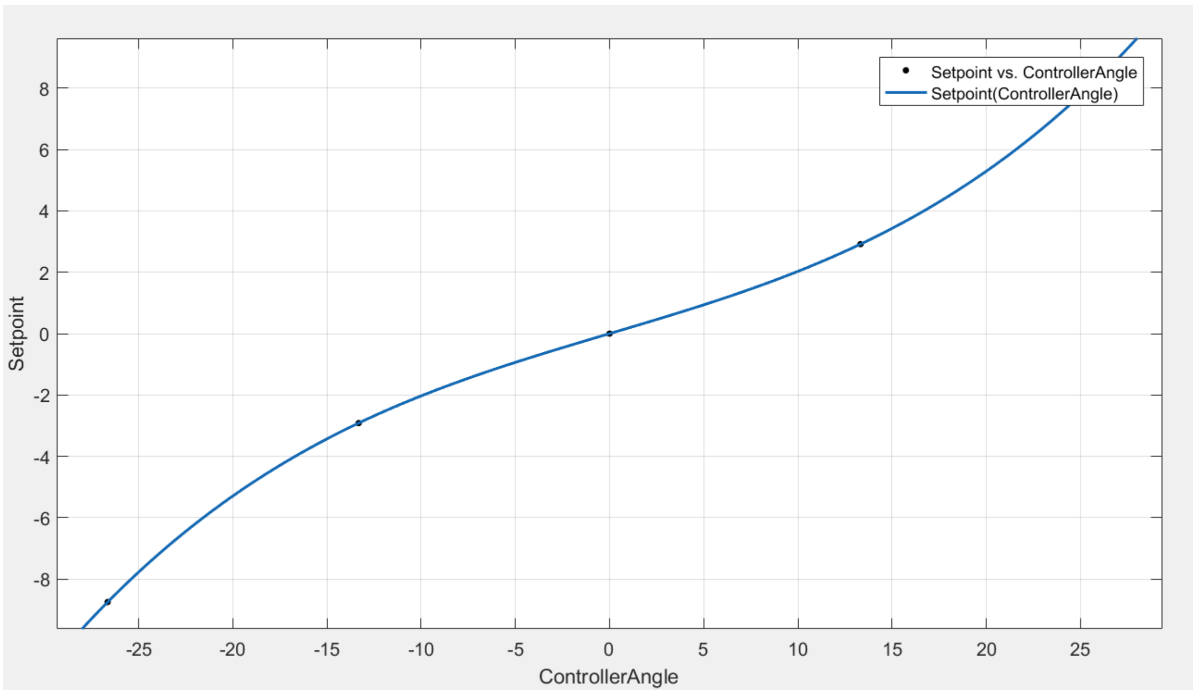

Figure 26: How the setpoint depend on the steering controller angle when driving above 110 km/h. This graph was made using MATLAB.

With these functions, the maximum steering angle is higher than what is needed because of the road regulations, compare to figure 23**.** At 60 km/h the maximum angle is 17.7° which is higher than the baseline for evasive manoeuvres which is 15.4°.

5 Designing safety features

This section explains how the safety features were designed and why they were designed in this certain way.

The company wanted this project to focus mainly on functionality rather than safety. However, with safety is regarded while designing the system it will be easier to achieve the safety goals when all the safety features are going to be implemented. Therefore, a few safety features were implemented.

It was decided that the system always should check that the input values from the sensor measuring the steering controller angle were realistic. As long as realistic values are given from the steering potentiometers the vehicle can drive but if a value that should not be possible to achieve in a normal mode of operation is received the motor turns off and the values has to go back to normal as well as the speed need to reach 0.4 km/h or below before the motor can start again. The first test for realistic values consists of making sure the steering controller input is inside the steering controllers range of motion, with a margin to prevent from unnecessary shut downs of the motor. The margin was chosen by taking a value close to the smallest value that cannot be achieved by pressing the steering controllers against the end of their range of motion. The second test for realistic values handles changes of the steering controller input between two measurements that are larger than what can be achieved during a normal mode of operation. It was measured how much the value could possibly change when steering. This maximum value, with an added margin, was used as the highest allowed change between two measurements.

It was chosen to set the speed that resets the safety error to 0.4 km/h to prevent from unnecessary delays because of noise in the speed measurements that could keep the speed measurement from reaching zero. It was chosen to let the input from the angle of the steering controllers continue to generate the angle of the wheels even when faulty inputs had been given. This because the risk of an accident was seen as higher if, for example, the angle of the wheels was set to zero after a faulty input had been received compared to letting the steering controllers continues to generate the steering angle since errors could be transient.

The highest achievable input change from the steering controllers between two measurements when handheld was measured to 3.4. 5 was chosen as the highest allowed value. When pressing the steering controllers to the end of their range of motion it was possible to get an input value of around 5 outside the range. 10 was chosen as a maximum.

The system would be safer if redundancy was implemented but that has already been done on a test rig and therefore it was not implemented on this test vehicle. The system might turn off the engine at a point where it is not needed but to avoid accidents it is better to turn it off to often than to seldom. With redundancy later implemented in the production vehicle these safety features can operate above the redundancy and work as an emergency stop if the redundant system fails.

6 Synchronization of the steering controllers and force feedback from the wheels

This section explains a theoretical design regarding synchronization of the steering controllers and force feedback from the wheels. It is also explained why this was not implemented in the thesis.

At first, the plan was to have the two steering controllers working without a mechanical linkage. This could be solved by having a DC motor connected to each steering controller and synchronize their position by using the already installed position sensors. With force sensors on both steering controllers the motors could start rotating and perform a steering movement when a force is applied on a steering controller. Higher force would equal a faster movement. To synchronize the steering controllers the sum of the forces applied on both steering controllers could be used to determine the rotation and position of both motors. The mechanical team and the hardware team did however decide to keep the mechanical linkage.

To achieve a natural steering experience the driver should not be able to turn the steering controllers in case the wheels cannot perform that turn. This could occur when, for example, a physical object is blocking the possibility for the wheels to perform a certain turn. Besides that, depending on personal preferences, some drivers might like to feel the vibrations from the road acting on the wheels to feel the current road conditions and adapt their driving to it.

This can be implemented by connecting a DC motor to the steering controllers and implement three force sensors, one for the steering controllers and one for each wheel and use them together with the already existing position sensors on the wheels and the steering controllers. The force on the steering controllers should determine the rotation of the motor, as explained above in this section when talking about synchronization, but now only one motor and one force sensor is needed since the steering controllers have a mechanical linkage.

When a force is applied on the steering controllers the wheels shall follow the steering controllers. When no force is applied to the steering controllers they shall follow the wheels. If the mismatch between setpoint and the angle of the wheels cannot be eliminated the steering controllers shall stop close to the position of the wheels so that the servo keeps trying to eliminate the deviation but it will, from a human's perspective, seem as the steering controllers are in the same position as the wheels. Also, it shall not be possible to move the steering controllers faster than the wheels can turn, from a human's perspective.

This can be implemented using the following pseudocode. The mentioned force on the steering controllers refers to a lateral force applied by the driver in the steering direction, left or right. The mentioned force on the wheels refers to a lateral force applied by, for example, objects on the road surface in the steering direction.

```
if(The force applied on the steering controllers is
close to zero.){
   Control the position of the steering controllers
   by using a PID-controller with the position of
   the wheels as the setpoint.

} else if(The force applied on the steering
controllers is below a certain size.){ //As if the
driver holds the steering controllers but do not try
to move them.

   if(The steering controllers are inside the
   proximity interval of the wheels.){
      Control the angular velocity of the steering
      controllers with zero as the setpoint.
      Control the position of the wheels with the
      control system implemented in this thesis.

   } else {
      Control the position of the steering
      controllers by using a PID-controller with the
      position of the wheels as the setpoint.
   }
```

```
} else { //The driver tries to move the steering
controllers.

    if(The steering controllers are outside of the
    proximity interval to the right){
        Control the position of the steering
        controllers by using a PID-controller with a
        position right inside of the right end of the
        proximity interval as the setpoint.

    } else if(The steering controllers are outside of
    the proximity interval to the left){
        Control the position of the steering
        controllers by using a PID-controller with a
        position right inside of the left end of the
        proximity interval as the setpoint.

    } else if(The force is applied to the right and
    the steering controllers are close to the right
    end of the proximity interval.
    || The force is applied to the left and the
    steering controllers are close to the left end of
    the proximity interval.){
        The maximum absolute value of the angular
        velocity of the steering controllers is set to
        the current angular velocity of the wheels.
        Control the angular velocity of the steering
        controllers by using a PID-controller.

        if((The force applied to the steering
        controllers –A·(The force applied on the
        wheels))· (The force applied to the steering
        controller) < 0 ){
            //A is a scaling constant that sets the
            amount of force needed to turn the
            steering controllers.
            The angular velocity setpoint is set to
            zero.
        else {
            The angular velocity setpoint is
            proportional to (The force applied to the
            steering controllers –A·(The force applied
            on the wheels)).
        }
        Control the position of the wheels with the
        control system implemented in this thesis.

    } else {
        Control the angular velocity of the steering
        controllers by using a PID-controller.

        if((The force applied to the steering
        controllers –A·(The force applied on the
        wheels))· (The force applied to the steering
        controller) < 0 ){
            //A is a scaling constant that sets the
            amount of force needed to turn the
            steering controllers.
            The angular velocity setpoint is set to
            zero.

        else {
            The angular velocity setpoint is
            proportional to (The force applied to the
            steering controllers –A·(The force applied
            on the wheels)).
        }
        Control the position of the wheels with the
        control system implemented in this thesis.
    }
}
```

If, for example, the position of the wheels would change quickly and create a mismatch between setpoint and angle of the wheels when no force is applied to the steering controllers the steering controllers would follow the wheels. If the user tries to hold the steering controllers still when they try to follow the wheels a force would be applied on the steering controllers and the wheels would follow the steering controllers again. In this way, the user would feel the force feedback. The size of the force the user would feel can be adapted thorough a variable to fit the users taste.

However, the mechanical construction for this setup was delayed 4 weeks due to a delayed delivery of the machines needed by the mechanical team to build the setup. Therefore, this could not be implemented in the frame of the thesis. The company will however implement it later and can then use this theoretical design.

7 Implementation

This section explains how the system was implemented. In figure 27 it is shown how the controller and speed dependency together build up the steering system.

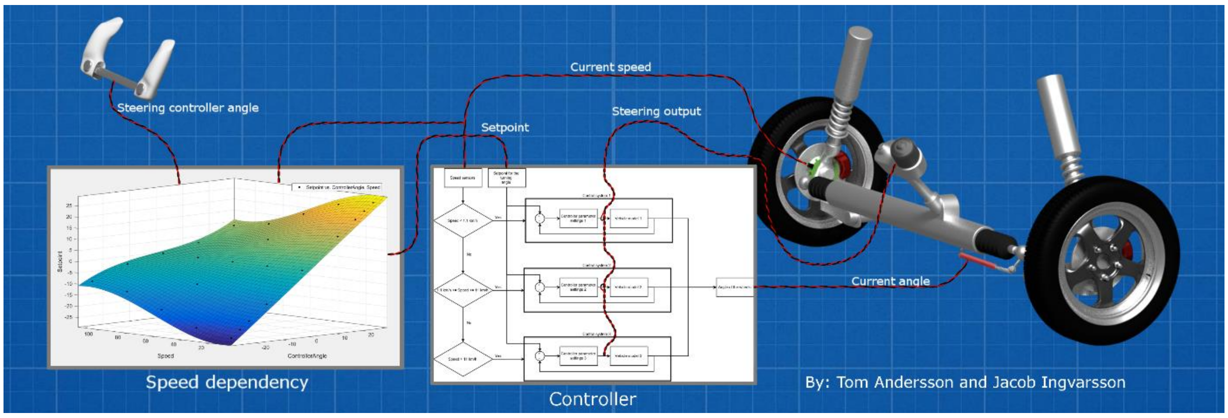

Figure 27: How the controller and speed dependency together build up the steering system.

7.1 Implementation of the controller

The controller was implemented in the Controllino program by assigning different values to the PID-parameters depending on the speed. Thereafter, the transfer function that was used by MATLAB when optimising the parameters, shown in equation (7), was z-inverse transformed as three separate transfer functions, one for each PID-component, using the backward Euler method, see equation (13). The result was implemented in the Controllino program.

Splitting up the transfer function from equation (7) into three transfer functions, one for each PID-component:

$$H_{(z)} = k_P + k_I \cdot \mathrm{h}\frac{z}{z-1} + k_D \frac{N}{1 + N \cdot \mathrm{h}\frac{z}{z-I}} \tag{13}$$

$$H_{P(z)} = k_P$$

$$H_{I(z)} = k_I \cdot \mathrm{h}\frac{z}{z-1}$$

$$H_{D(z)} = k_D \frac{N}{1 + N \cdot \mathrm{h}\frac{z}{z-I}}$$

h = sample period. [s]

In equation (14) to (16) it can be seen how the transfer functions from equation (13) were rearranged before the z-inverse transformation was done.

Rearranging of $H_{P(z)}$ from equation (13):

$$H_{P(z)} = \frac{U_{P(z)}}{E_{(z)}} = k_P \Leftrightarrow U_{P(z)} = k_P \cdot E_{(z)} \tag{14}$$

$U_{P(z)}$ = output from the P-component of the controller.

$E_{(z)}$ = deviation between setpoint and the angle on the wheels.

Rearranging of $H_{I(z)}$ from equation (13):

$$H_{I(z)} = \frac{U_{I(z)}}{E_{(z)}} = k_I \cdot \mathrm{h} \frac{z}{z-1} \Leftrightarrow U_{I(z)} = z^{-1} U_{I(z)} + k_I \cdot \mathrm{h} \cdot E_{(z)} \tag{15}$$

$U_{I(z)}$ = output from the I-component of the controller.

$E_{(z)}$ = deviation between setpoint and the angle on the wheels.

h = sample period. [s]

Rearranging of $H_{D(z)}$ from equation (13):

$$H_{D(z)} = \frac{U_{D(z)}}{E_{(z)}} = k_D \frac{N}{1 + N\mathrm{h}\frac{z}{z-I}} \Leftrightarrow \tag{16}$$

$$\Leftrightarrow U_{D(z)} = \frac{1}{N\mathrm{h}+1} z^{-1} U_{D(z)} + k_D \frac{N}{N\mathrm{h}+1}\left(E_{(z)} - z^{-1}E_{(z)}\right)$$

$U_{D(z)}$ = output from the D-component of the controller.

$E_{(z)}$ = deviation between setpoint and the angle on the wheels.

h = sample period. [s]

The z-inverse transformations of equation (14) to (16) is shown in equation (17) to (19).

Z-inverse transformation of equation (14):

$$U_{P(z)} = k_P \cdot E_{(z)} \Leftrightarrow u_{P(k)} = k_P \cdot e_{(k)} \tag{17}$$

$u_{P(k)}$ = output from P-component of the controller at this sample.

$e_{(k)}$ = deviation between setpoint and the angle on the wheels at this sample.

Z-inverse transformation of equation (15):

$$U_{I(z)} = z^{-1}U_{I(z)} + k_I \cdot \mathrm{h} \cdot E_{(z)} \Leftrightarrow u_{I(k)} = u_{I(k-1)} + k_I \cdot \mathrm{h} \cdot e_{(k)} \tag{18}$$

$u_{I(k)}$ = output from I-component of the controller at this sample.

$u_{I(k-1)}$ = output from I-component of the controller at the last sample.

$e_{(k)}$ = deviation between setpoint and the angle on the wheels at this sample.

h = sample period. [s]

Z-inverse transformation of equation (16):

$$U_{D(z)} = \frac{1}{N\mathrm{h}+1} z^{-1}U_{D(z)} + k_D \frac{N}{N\mathrm{h}+1}\left(E_{(z)} - z^{-1}E_{(z)}\right) \Leftrightarrow \tag{19}$$

$$\Leftrightarrow u_{D(k)} = \frac{1}{N\mathrm{h}+1} u_{D(k-1)} + k_D \frac{N}{N\mathrm{h}+1}\left(e_{(k)} - e_{(k-1)}\right)$$

$u_{D(k)}$ = output from D-component of the controller at this sample.

$u_{D(k-1)}$ = output from D-component of the controller at the last sample.

$e_{(k)}$ = deviation between setpoint and the angle on the wheels at this sample.

$e_{(k-1)}$ = deviation between setpoint and the angle on the wheels one sample ago.

h = sample period. [s]

To get the total output from the controller the individual PID-components outputs are added. How the controller then was implemented in the Controllino program is shown in appendix 3.

An anti-windup was implemented by setting a maximum value for the absolute value of the integral part of the PID-controller. If this value was to be exceeded the integral part would not be updated. This maximum value was chosen by monitoring the values during normal operating conditions, trying to achieve the highest possible absolute value of the integral part of the PID-controller, where windups of the integral part did not occur. The reached value was 650. 700 was set as a maximum value, see appendix 3 for the complete code.

The following code shows how the PID-controller and the anti-windup were implemented.

```
 double Kp;
  double Ki;
  double Kd;
  double N;
  double h = 0.01;  //Sample period
  if(CurrentSpeed<0.3){   //Parameter control on the
current speed [m/s]
    Kp = -29.4106;
    Ki = -12.1274;
    Kd = 3.7603;
    N = 7.512;
  }else if(CurrentSpeed>3){
    Kp = -39.9944;
    Ki = -60.3186;
    Kd = 0.90866;
    N = 35.8523;
  }else{
    Kp = -7.5771;
    Ki = -7.3581;
    Kd = 0.42957;
    N =  15.3967;
  }

  double Deviation = TargetSteeringAngle -
CurrentSteeringAngle;    //PID controller starts
here

  SteeringAdjustmentP = Kp*Deviation;

  if( abs(Ki*Deviation) < 700 ){
//Anti-windup
```

```
    SteeringAdjustmentI = SteeringAdjustmentI
+Ki*h*Deviation;
  }
 if( abs(SteeringAdjustmentD -
1/(N*h+1)*SteeringAdjustmentD+Kd*N/(N*h+1)*(Deviatio
n - LastDeviation)) <400 ){ //To prevent from spike
in the D-part when the sensors are powered on
    SteeringAdjustmentD =
1/(N*h+1)*SteeringAdjustmentD+Kd*N/(N*h+1)*(Deviatio
n - LastDeviation);
  }

  SteeringAdjustment = SteeringAdjustmentP +
SteeringAdjustmentI + SteeringAdjustmentD;
  LastDeviation = Deviation;
```

7.2 Implementation of the speed dependency

The developed mathematical functions were implemented in the code fully shown in appendix 3. The angle of the steering controllers and the speed are used in one of the functions, depending on which interval the speed is at that moment, to get the setpoint for the angle of the wheels.

The following code shows how the speed dependency was implemented.

```
if(CurrentSpeed*3.6 < 10) //[CurrentSpeed]= m/s, the
formla uses km/h.
  {
    TargetSteeringAngle =
ControllerSteeringAngle*STEERING_RATIO;
  }
  else if(CurrentSpeed*3.6 > 110)
  {
    double p1 = 0.0002055;
   double p2 = -1.949*pow(10,-18);
   double p3 = 0.1824;
   double p4 = 4.643*pow(10,-16);

   TargetSteeringAngle =
p1*pow(ControllerSteeringAngle*STEERING_RATIO,3) +
p2*pow(ControllerSteeringAngle*STEERING_RATIO,2) +
p3*ControllerSteeringAngle*STEERING_RATIO + p4;
  }
  else
  {
  double p00 = 1.22*pow(10,-15);
  double p10 = 0.9872;
  double p01 = -2.126*pow(10,-16);
  double p20 = -1.826*pow(10,-17);
  double p11 = 0.003717;
  double p02 = 1.936*pow(10,-17);
  double p30 = -2.055*pow(10,-5);
  double p21 = -4.383*pow(10,-19);
  double p12 = -0.0002579;
  double p03 = -3.516*pow(10,-19);
  double p40 = 3.223*pow(10,-20);
  double p31 = 2.055*pow(10,-6);
  double p22 = 4.636*pow(10,-21);
  double p13 = 1.433*pow(10,-6);
  double p04 = 1.739*pow(10,-21);

  TargetSteeringAngle = p00 +
p10*ControllerSteeringAngle*STEERING_RATIO +
p01*CurrentSpeed*3.6 +
p20*pow(ControllerSteeringAngle*STEERING_RATIO,2)
 +

p11*ControllerSteeringAngle*STEERING_RATIO*CurrentSp
eed*3.6 + p02*pow(CurrentSpeed*3.6,2) +
p30*pow(ControllerSteeringAngle*STEERING_RATIO,3)
+
p21*pow(ControllerSteeringAngle*STEERING_RATIO,2)*Cu
rrentSpeed*3.6  +
p12*ControllerSteeringAngle*STEERING_RATIO*pow(Curre
ntSpeed*3.6,2) + p03*pow(CurrentSpeed*3.6,3) +
p40*pow(ControllerSteeringAngle*STEERING_RATIO,4) +
p31*pow(ControllerSteeringAngle*STEERING_RATIO,3)*Cu
rrentSpeed*3.6 +
p22*pow(ControllerSteeringAngle*STEERING_RATIO,2)*po
w(CurrentSpeed*3.6,2) +
p13*ControllerSteeringAngle*STEERING_RATIO*pow(Curre
ntSpeed*3.6,3) + p04*pow(CurrentSpeed*3.6,4);
  }
}
```

7.3 Implementation of the safety features

The safety features were implemented as fully shown in appendix 3. When a value that should not be possible to achieve during a normal mode of operation is received from the steering controllers a variable is set to false. This variable need to be true for it to be possible to send a throttle signal to the engine. It will become true as soon as the speed reaches 0.4 km/h or below as well as the values from the steering controllers have gone back to be realistic.

The following code shows how the safety features were implemented where the steering input is read.

```
 if( (SmoothSteeringInput < CONTROLLER_STEERING_MAX
- 10) ||
(SmoothSteeringInput > CONTROLLER_STEERING_MIN +
10))
//Safety feature (Min/Max stands for angle not input
value)
  {
    Safe = false;
  }
  if( abs(SmoothSteeringInput -
SmoothSteeringInputOld) > 5) //Safety feature.
  {
      Safe = false;
  }
```

The following code shows how the safety features were implemented where the throttle is set.

```
    if(ControllerThrottle < 0.05f || !Safe)   // The
safe variable is set to false by the safety features
when needed.
  {
    OutputThrottle = 0.0f;
    OutputBrake = 1.0f;
    if(CurrentSpeed < 0.1){
      Safe = true;
    }
  }
```

8 Test and verification

This section explains how the system was tested and what the results were.

8.1 Testing the controller

The controller for the steering was tested by doing turns while plotting the setpoint and the angle of the wheels to see that the basics of the controller worked as planned. Thereafter, a test drive was done. In this test drive the test vehicle was driven at various speeds while doing various kinds of turns. Finally, two graphs were made. One with a steering response from a 0° to around 15° step at zero speed for the old proportional controller and one with a steering response from a similar step for the new PID-controller. These were then analysed. It was chosen to only make graphs for the steering response at zero speed due to the complexity of performing the exact same turn at the same road surface twice while driving.

It was found that when switching of the main switch while having the Controllino plugged in to a computer through USB the wheels turned fully to one side as soon as the main switch was switched on again. This occurred because all the sensors lost their power while the Controllino continued its calculations. This resulted in a quick change to a high value in the derivative part of the PID-controller. The first approach to solve this was to set a maximum value for the absolute value of the derivative part but it was realised that it could occur higher values in a normal mode of operation compared to when turning the power supply to the sensors on and off. The next approach was to set a maximum value for how much the derivative part could change between two measurements. In normal mode of operation, the change could reach 380. 400 was set as a maximum value. See appendix 3.

The general test showed that the controller worked as a PID-controller is supposed to, by eliminating the standing deviation, and had therefore been implemented correctly in the code.

When test driving the steering still felt to slow. To fix this a new stronger and faster servo would be needed. Also, when trying to drive straight forward small unwanted movements in the hands that were too small for the proportional controller to handle now got transferred to the wheels and made the steering wobbly. This will be less of a problem when force feedback is implemented since that will make a higher force needed to move the steering controllers. The prototype will have a more stable mechanical construction with less vibrations which also would decrease these unwanted movements. However, the transitions between the three different parameter settings was not noticeable neither when driving straight ahead or when changing speed while turning. Also, it was possible to achieve a sharper angle than before. Also, the overshoot was not noticeable from the driver’s perspective.

In figure 28 and 29 a steering response from an angle of around 0° to an angle of around 15° at zero speed is shown for the proportional controller, figure 28, and the PID-controller, figure 29. These shows how the PID-controller eliminated the deviation between setpoint and angle of the wheels and that the proportional controller could not do that.

It can be seen that the PID-controller has a lower rise derivate compared to the proportional controller. The response time can be measured to 0.25 seconds which is higher compared to the proportional controller that has 0.18 seconds. Furthermore, an undershoot was expected, see figure 17, but no undershoot occurred in this test. It is also shown that the overshoot is larger than expected, 14% compared to 1.53% in figure 17. This could be fixed by, for example, decreasing the absolute value of the proportional gain or increasing the absolute value of the integral gain but that would have made the steering slower and the steering speed was already the biggest problem with the steering. It could also have been solved by increasing the absolute value of the derivative gain. However, when this was attempted no values that resulted in a smaller overshoot were found. Therefore, it was chosen to let the overshoot be and the PID-parameters were deemed the most optimal for this application. The complexity of analysing steering responses increased because the steering response was not identical every time the same steering input was given.

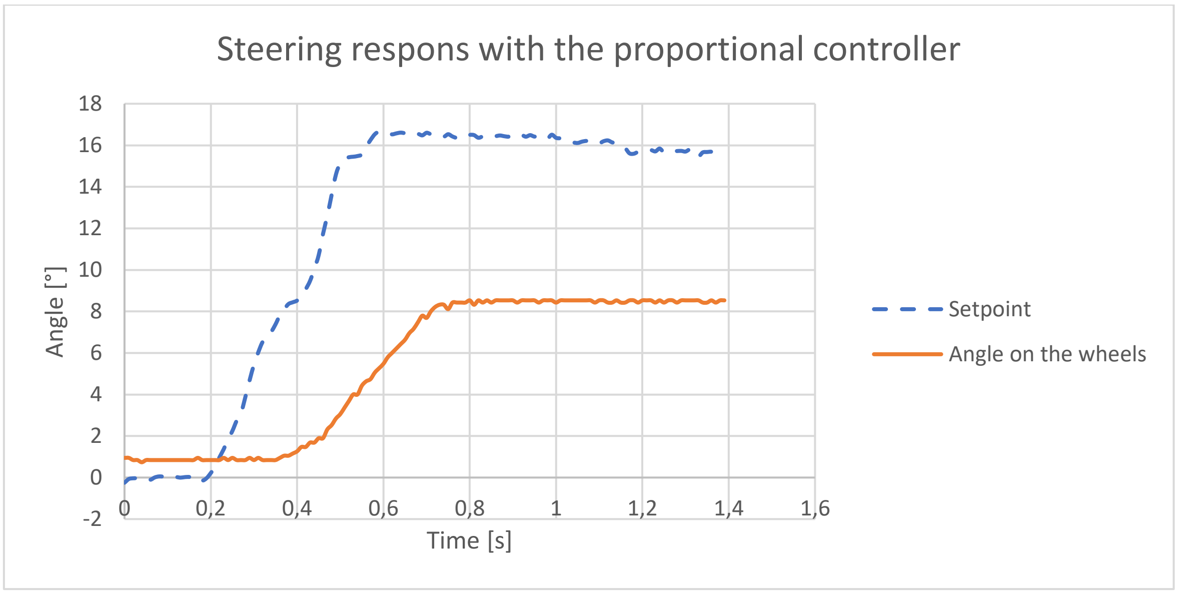

Figure 28: Steering response with the proportional controller.

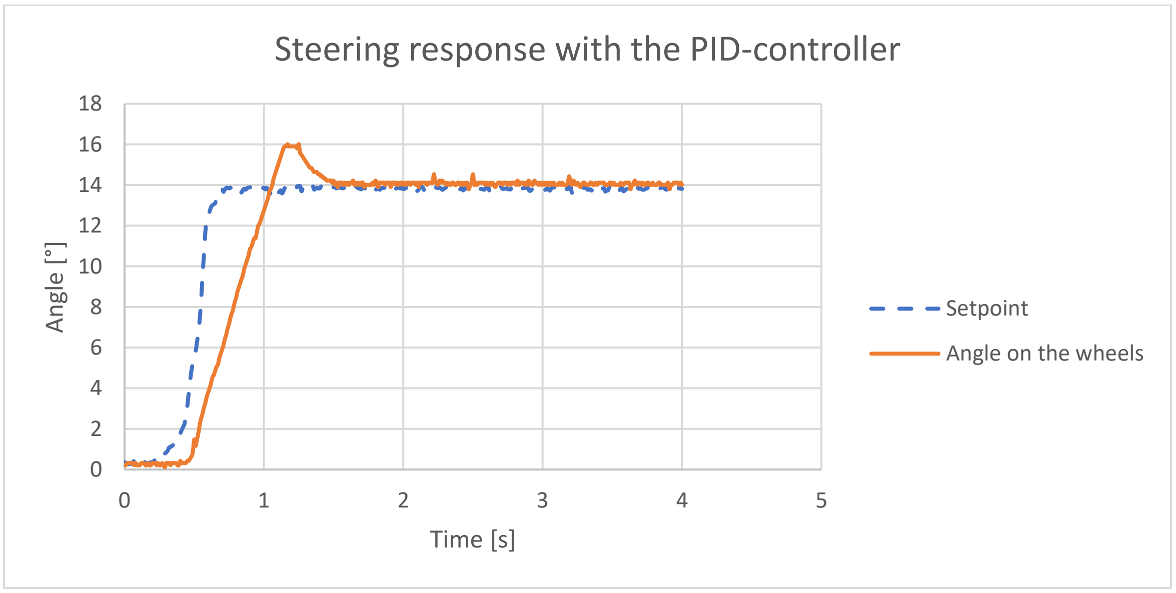

Figure 29: Steering response with the PID-controller.

8.2 Physical test of the speed dependency

The tests consisted of driving at various speeds within the limited speed range of the test vehicle to see that it worked when operating at low speeds. Thereafter, the variable representing the speed was set to higher values, for example 60, 90 and 120 km/h, to test so that the mathematical functions worked as expected at those speeds as well. Note that the speed of the test vehicle still was low, only the variable in the code was increased.

When first testing the speed dependency the position of the wheels experienced jitter. As mentioned in section 3.2, the speed measurement had already been corrected and therefore the speed measurement was considered to be correct. Though, when again studying the speed data and simultaneously driving interference was noticed. The speed data spiked and then slowly, thanks to the filtering already applied in the code, dipped, see appendix 3. Since our speed dependency rely on a smooth and correct speed input the spikes in the speed measurement resulted in sudden changes regarding the angle of the wheels. This was fixed by capping the maximum difference between two sample periods so that the spikes are ignored. The maximum speed change between two samples was set to 0.7 km/h. This value eliminates the spikes but is big enough so that quick accelerations and decelerations still are read as supposed.

Both when driving within the speed range of the test vehicle and when changing the speed variable to a higher value the speed dependency worked as expected. The transition between the first and second speed interval was not noticeable and the range of motion and sensitivity did not change in an uncomfortable way when steering while accelerating or decelerating. When having the variable put to higher speeds it could be seen how the range of motion for the wheels and the sensitivity close to the middle point of the steering controllers range of motion decreased.

8.3 Virtual reality test of the speed dependency

A VR-simulator was used to test the steering experience at various speeds. Uniti had a driving simulation set up but it had to be altered to incorporate the speed depending equations for the angle of the wheels, (11) and (12). The speed depending part was added between the steering controller input and steering output in the VR, as seen in appendix 4. The used game engine was Unity. Unity is a game engine that combines scripting, animation, 3D assets, physics simulations and sound files into one executable file. Unity uses C# as programming language. Uniti had already implemented a physics model of the vehicle made to scale with a simulation level car script applied to simulate traction and the physical movement.

To get a steering input to the VR-simulator the output signal from the currently installed potentiometers on the joystick setup on the test vehicle was sent into the Unity game engine. This was done by breaking apart an existing gaming controller and replacing the internal potentiometer signal from the thumb stick with the signal from the joystick setup on the test vehicle. This made the connection to the Unity game engine simpler since it could be connected as if it was a normal gaming controller as well as specially designed electronics were not needed.

To test the speed dependency, an area for a moose test and a road were created as models in Blender 3D, both were made to scale. Blender 3D is a 3D modelling software under the GNU license used to create 3D models. The created models consist of vertices, dots, in a three-dimensional space with coordinates referenced to a vertex. Three vertices are then assigned to span a triangle that has a normal that faces a certain direction. A complete 3D model comprises of one or more vertices and multiple surfaces connecting the vertices. These surfaces are called faces. The models were imported to the Unity game engine. The moose test was set up as a track with 3D models of cones, see figure 30, according to the ISO-standard ISO3888-2 [14]. The road was set up as a 4-lane road with a lane width of 3,5 meters. The curve radiuses were set according to Trafikverkets road regulations [15].

The modelled curves were the following, all at a banking of 2,5%.

- A radius of 20 meters for 30 km/h.
- A radius of 100 meters for 50 km/h.
- A radius of 220 meters for 70 km/h.
- A radius of 450 meters for 90 km/h.
- A radius of 800 meters for 110km/h.

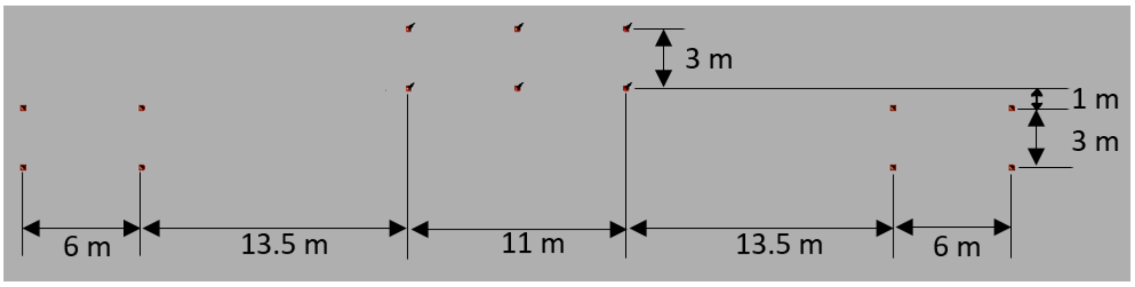

Figure 30: The setup for the moose test.

The created VR-environment is shown in figure 31 and 32**.**

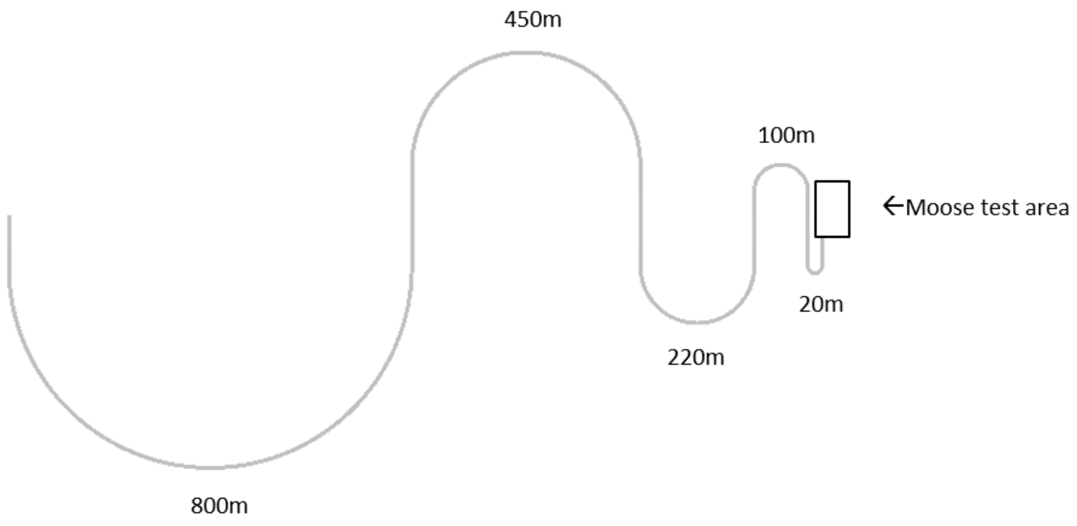

Figure 31: A drawing of the VR-environment with the radius for each curve noted.

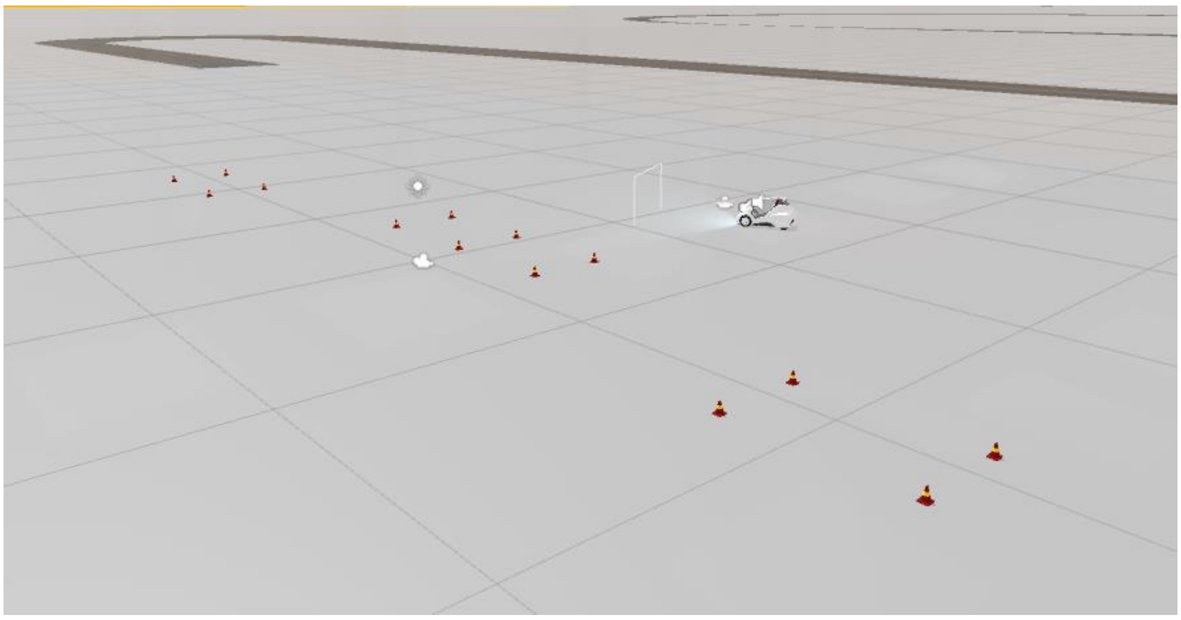

Figure 32: A screenshot of the VR-environment in Unity.

With the VR-environment set up tests were performed. The first test consisted of performing the moose test at 60 km/h, the same speed that Teknikens värld used, to see if it was possible to achieve the needed steering angle and make it through the track [19]. The second test consisted of driving along the road following the curves at the speed the curves were dimensioned for and also while accelerating and decelerating to see if the road with its curves were comfortable to follow. The third test consisted of doing lane changes and evasive maneuvers along the road at various speeds to make sure that these kinds of maneuvers could be performed in a comfortable way. In these three tests the steering should feel natural for the driver.

To get stable input values they needed to be filtered. The staff at Uniti had fixed this by always keeping a percentage of the old value when reading a new input value. With this filtration, the steering experienced a delay. Furthermore, the acceleration and deceleration in the virtual environment was greater compared to a real vehicle. This made the testing more difficult to complete. After some driving the driver got used to the driving simulator and the speed dependency could be evaluated fairly.

The moose test succeeded. The driver was able to drive through the track at the correct speed, 60 km/h, and the steering was experienced as natural and comfortable. Also, when driving along the road the driver experienced the steering as natural and comfortable, both regarding following the curves, changing lanes and doing evasive maneuvers. The staff at Uniti agreed that the implemented speed dependency had improved the steering at higher speeds.

8.4 Testing the safety features

Theses safety features were tested by changing the maximum values, described in section 5, to values that could be reached while driving. They worked as planned from the first attempt. With their right values, they never triggered in a normal mode of operation.

9 Conclusion

This section summarizes the essentials of the results and ties it together with the problems and goals of the thesis defined in the introduction. Also, alternative approaches as well as future development opportunities and ethical aspects are discussed.

9.1 Summarization of the essentials of the results and how it will be used

The test and verification section, section 8, showed that the method of implementing a control system for the steering of a vehicle by using a PID-controller with a parameter control on the current speed worked. It also showed that it is possible to safely steer a vehicle at different speeds using joysticks by letting the maximum angle of the wheels decrease with increased speed as well as decreasing the sensitivity more when being close to the middle point of the steering controllers range of motion when the speed is increased. It was proven that the installed servo was to slow to achieve a natural steering but with a stronger and faster servo that problem would be eliminated.

The approaches of making system identifications and then choosing PID-parameters can be used as a recipe for implementing this functionality in any vehicle using a steer by wire system, for example the upcoming prototype that the company is developing.

The dependency between the angle of the steering controllers and the speed of the vehicle can be directly copied to any road driven vehicle operated by joysticks, or something similar to joysticks, that have a similar range of motion compared to this case, for example the upcoming prototype that the company is developing. The system will however need to be tested in a real situation while steering and accelerating until the maximum speed has been reached and the steering still feels natural or until the steering becomes unnatural. Having a stable speed measurement input to the control system is necessary for this system to behave as expected. An unstable speed measurement could result in jitter at the steering wheels when the handles is at a fixed position. This could result in unexpected turning input that could be dangerous in situations where precise and consistent steering is needed, for example at high speeds.

The theoretical design regarding redundancy describes that N modular redundancy can be used in a steering system for a vehicle. The mean time until failure should be higher than the estimated lifetime of the vehicle and the redundant system should consist of various kinds of hardware implemented with different software. All physical components should be placed away from each other and away from hot or vibrating parts of the vehicle. All connections should consist of double wires.

The theory regarding synchronization of joysticks and force feedback from the wheels can be used as a base when the company in the future will implement these features.

However, with most drivers being used to the traditional mechanical system with a steering wheel it could be hard to get used to an electrical system with steering controllers. Also, it is hard to make an electrical system as reliable as a mechanical one and to assure potential buyers that the system is safe.

9.2 Answers to the defined problems

1. Our studies show that one solution is to let the maximum angle of the wheels decrease with increased speed as well as decreasing the sensitivity more when being close to the middle point of the steering controllers range of motion when the speed is increased. The mathematical functions are shown in equation (11) and (12) and figure 25 and 26**.**
2. The servo was to slow to achieve a natural steering experience therefore the needed rise time could not be defined. The overshoot became 14% and that was not noticeable from the perspective of a driver. The response time became 0.25 seconds and that was not noticeable from the driver's perspective either. As long as the output does not oscillate around the setpoint but rather moves calmly towards it the size of the settling time is not critical.
3. The rise time is of the highest priority, it always needs to be minimised. Then comes the response time and overshoot, these should be minimised but according to our result they may exist to a certain degree, see the answer to problem 1. Last comes the settling time which is less critical.
4. The relatively high numbers of poles and zeroes for the most precise mathematical models for the system made the pole placement method impracticable. Therefore, a PID-controller was the solution. The dynamics changes with the speed and therefore several PID parameter settings operating in different speed intervals improved the result.
5. This thesis was mainly focused on functionality. But it was desired to be able to detect faulty inputs from the position sensors on the steering controllers.
6. The safety requirements, mentioned in point 5 in this section, were partly met by detecting if the input value from the position sensor for the steering controllers was outside of its range or had changed to much between two samples. If this occurred the engine turns of and cannot start until the error has been fixed and the speed has gone down to zero. However, faulty errors can occur in other ways. The input signal can, for example, freeze while the angle of the steering controllers change. This cannot be noticed by the implemented system.

9.3 Evaluation according to the goal setting

The control system was developed and implemented but the steering did not feel natural due to the servos lack of speed. However, besides the rise time the implemented steering felt natural at different speeds. Force feedback was never implemented but a theoretical preparation for the implementation was made.

This concludes that not all the goals were reached but the gained knowledge is enough for the company be able to achieve a natural steering in the company's prototype vehicle.

9.4 Alternative approaches and future development opportunities

The implemented system can handle variations in the dynamics that occur due to speed variations. However, if any other factors affect the system in a way that gives a negative impact on the steering experience the implemented system cannot adapt according to those changes. As mentioned when designing the system theoretically, section 3.1, an adaptive system could have fixed that issue but that could result in a slower system that could harm the steering experience. However, with more computing power this problem could be eliminated. Still, the adaptive controller would need to adapt its parameters in real time without affecting the steering output since, for example, the adaption would sometimes have to happen at high speeds.

Another way of handling dynamic variations that depend on other factors than the speed could be to use the same approach as in this thesis but not only having different controller parameters depending on the speed but to define intervals for several different variables that could have an impact on the steering and have one controller for every combination. These variables need to be measurable or work as different modes to be chosen by the driver.

Here are some examples of measurable variables that could have an impact on the steering.

- Humidity.
- Temperature.
- Road incline.
- Road superelevation.
- Weight and weight distribution.
- Tire pressure.

Here are some examples of potential modes a user could choose between.

- Dry asphalt.
- Wet asphalt.
- Icy road.
- Gravelled road.

The speed dependency implemented in this thesis makes it possible to drive a car with joysticks at various speeds. This could instead have been solved by having cameras around the car and not only use the current speed but also the curvature of the road ahead. However, that could entail faster variations in, for example, the maximum angle of the wheels which could make the steering more complex for the driver. Yet, for the future when the vehicle has autonomous features implemented this could be used alongside with other information around the car, for example the location of other vehicles on the road and different obstacles, to limit the driver from making dangerous turns.

Regarding estimating the wheels steering angle and corresponding curve radius a more precise physical model can be estimated when the tire stiffness has been defined. This would include steering slip and loss in steering response when calculating the curve radius from the angle of the wheels. How the radius can be calculated is shown in equation (20) [17].

The curve radius as a function of the angle of the wheels:

$$\frac{1}{R} = \frac{\delta_S}{l_f + l_r + \frac{m\left(\frac{v}{3.6}\right)^2 \left(l_r C_{\alpha r} - l_f C_{\alpha f}\right)}{2 C_{\alpha f} C_{\alpha r} (l_r + l_f)}} \tag{20}$$

R = Curve radius. [m]

m = Mass of the vehicle. [kg]

v = Speed. [km/h]

l_f = Length between the front axis and center of mass. [m]

l_r = Length between the rear axis and center of mass. [m]

$C_{\alpha r}$ = Tire stifness for the front wheels. [N/rad]

$C_{\alpha f}$ = Tire stiffness for the rear wheels. [N/rad]

δ_S = Angle of the wheels. [°]

The preciseness of the control system can be increased if the sample period is generated with an interrupt that is executed at an exact rate. However, to do this it is necessary to make sure that the rest of the systems running on the same control unit is not affected negatively by this interrupt.

9.5 Reflections regarding ethical aspects

Since this system is a part of the development of an electric vehicle that aims to replace big cars with combustion engines in big cities around the world this system has a direct connection to the desire of reducing the carbon emissions to counteract the global warming.

From another perspective, steering a vehicle with joysticks instead of a steering wheel can potentially be easier for people with various disabilities. When steering with a conventional steering wheel the driver needs to perform big motions that involve the whole arms. With a joystick setup, the wrists would perform most of the movements. Accelerating and breaking is not a part of this thesis but the joystick setup will handle that as well which means that, technically, people who have no legs still could drive the vehicle.

Furthermore, a steer by wire system makes it easier to implement autonomous features since the movements of the car can be controlled with electric signals. This makes it easier to implement various safety features such as making the vehicle avoid pedestrians, animals and such as well as stopping the vehicle at the side of the road if the driver has fallen asleep.

10 Vocabulary

Autonomous: Self driving.

Electronic stability control: A control system implemented in a vehicle to prevent from under steering, over steering and rollovers.

Heads-up display: Images are projected on a display or windshield. These images should appear to be out in the world behind the screen.

Redundancy: Having one or several systems that can take control of a task in case one unit breaks.

Setpoint: The desired output value from a controller.

Speed dependency: In this thesis speed dependency means the relation between the angle of the steering controllers, the speed and the setpoint for the angle of the wheels.

Vertex: The point in a coordinate system where all axes equals zero.

VR: Virtual reality.

11 Source criticism

[1], [2] and [3] gives information about a vehicle made by Nissan. These sources are owned by Nissan themselves and are therefore trustworthy since the used information is technical facts and not opinions.

[4] is an e-mail from Nissan themselves. Since the information there is regarding one of Nissans vehicles this source is trustworthy since also this information is technical facts and not opinions.

[5] and [11] are established technical journals.

[6] is an independent agency of the executive branch of the federal government of USA. The used information is technical information and not opinions, therefore this source is trustworthy.

[7] is an executive branch department of the United States federal government. The used information is technical information and not opinions, therefore this source is trustworthy.

[8] is an established aerospace magazine. The used information is technical information and not opinions, therefore this source is trustworthy.

[9] gives information regarding ROS. This source is owned by ROS themselves and therefore this source is trustworthy.

[10] is an established multinational technology company this makes their information trustworthy since the used information is not related to their products.

[12] is technical information found in a published encyclopedia on electronics and is therefore a trustworthy source.

[13] is technical and published information about BMWs own technology taken from their own publication and thus a trustworthy source.

[14] is also a source from an established technical journal which makes it trustworthy.

[15] and [20] are information about road design taken directly from the Swedish department of traffic.

[16] is facts taken from an established source collecting information on vehicles.

[17] is a published textbook targeting mechanical engineering.

[18] is an establish webpage explaining how various techniques works.

[19] is an established technical journal posting a video on an established platform for sharing videos.

[21] is information regarding ISO-standards at their own webpage.

[22] is a picture of a mechanical construction that matches the information given by [12].

[23] is a published textbook targeting control systems.

[24] is a published textbook targeting system identification.

[25] and [26] are from an established technical journal focused on car reviews.

12 References

[1] (InfinitiNews, *2014 Infiniti Q50 Specifications,* Available: http://infinitinews.com/en-CA/infiniti/canada/presskits/ca-2014-infiniti-q50-press-kit/releases/2014-infiniti-q50-key-dimensions-and-specifications, (2017-01-30))

[2] (Infiniti, *It Puts the Future of Driving in Your Hands,* Available: http://www.infinitiusa.com/sedan/q50/highlights/technology.html, (2017-01-30))

[3] (NissanNews, 2016-01-11, *Infiniti breaks new global sales record in 2015,* Available: http://nissannews.com/en-US/nissan/usa/releases/infiniti-breaks-new-global-sales-record-in-2015, (2017-01-30))

[4] (Nissan North America INC, 2013-11-19, *Defect information report,* Public e-mail from Nissan North America INC to Nancy Lewis at the National Highway Traffic Safety Administration (PDF), Available: https://static.nhtsa.gov/odi/rcl/2013/RCDNN-13V588-7078.pdf, (2017-01-30))

[5] (Andrew Del-Colle for Popular Mechanics, 2013-12-17, *Infiniti's Steer by wire Recall is a Cautionary Reminder on the Road to Autonomy,* Available: http://www.popularmechanics.com/cars/a9940/infiniti-recalls-steer-by-wire-system-16276674/, (2017-01-30))

[6] (APPEL News Staff, *This Month in NASA History: The Digital Fly-By-Wire Program Revolutionized Flight*, 2016-03-16 Available: https://appel.nasa.gov/2016/05/16/this-month-in-nasa-history-the-digital-fly-by-wire-program-revolutionized-flight/, (2017-02-03))

[7] (Department of Defence USA, *Flying Qualities of Piloted Aircraft*, 1997-12-19, Available: http://www.mechanics.iei.liu.se/edu_ug/tmme50/MIL-HDBK-1797.PDF, (2017-01-30))

[8] (H. Hill at Flight International, 1972-01-20, Available: https://www.flightglobal.com/pdfarchive/view/1972/1972%20-%200147.html, (2017-01-30))

[9] (ROS.org, 2016-12-03, Available: http://wiki.ros.org/ROS/Tutorials, (2017-01-23))

[10] (National instruments, 2008-01-11, *Redundant system Basic Concepts*, Available: http://www.ni.com/white-paper/6874/en/, (2017-04-13))

[11] (Anthony Doman for Popular Mechanics, *Clever cars: Infiniti Q50 can steer by wire*, 2014-01-14, Available:

http://www.popularmechanics.co.za/wheels/infiniti-q50-can-steer-by-wire/, (2017-04-14))

[12] (*2016 Columbia Electronic Encyclopedia*, 6th edition, Steering system p. 1, New York: Columbia University Press)

[13] (BMW Technology Guide, *Active Steering*, Available: http://www.bmw.com/com/en/insights/technology/technology_guide/articles/mm_active_steering.html, (2017-03-27))

[14] (Teknikens värld, *Resultat i Teknikens Världs älgtest*, 2014-08-25, Available: http://teknikensvarld.se/algtest/, (2017-03-28))

[15] (Trafikverket, *Linjeföring, 6 Horisontalkurvor*, 2004-05, Available: http://www.trafikverket.se/TrvSeFiler/Foretag/Bygga_och_underhalla/Vag/Vagutformning/Dokument_vag_och_gatuutformning/Vagar_och_gators_utformning/Linjeforing/06_horisontalkurvor.pdf, (2017-03-28))

[16] (Edmunds, *2012 Jeep Grand Cherokee - Road Test Specs*, Available: https://www.edmunds.com/jeep/grand-cherokee/2012/road-test-specs/, (2017-04-11))

[17] (Rajamani, Rajesh. 2012, *Vehicle dynamics and control*, Second edition, page 209, Boston: Springer US)

[18] (Karim Nice for how stuff works, *How car steering works*, page 2 and 4, Available: http://auto.howstuffworks.com/steering.htm, (2017-03-29))

[19] (Teknikens värld on Youtube, *Jeep Grand Cherokee moose test – the full story*, 2012-07-16, Available: https://www.youtube.com/watch?v=zaYFLb8WMGM&feature=youtu.be&t=37, (2017-03-20))

[20] (Trafikverket, *Vägtyper*, 2004-05, Available: http://www.trafikverket.se/TrvSeFiler/Foretag/Bygga_och_underhalla/Vag/Vagutformning/Dokument_vag_och_gatuutformning/Vagar_och_gators_utformning/Sektion_landsbygd-vagrum/05_vagtyper.pdf, (2017-03-23))

[21] (International organization for standardization, *ISO 3888-21:2011*, 2011-03, Available: https://www.iso.org/standard/57253.html, (2017-03-23))

[22] (LaurensvanLieshout, *Steer system*, 2010-08-08, Available: https://commons.m.wikimedia.org/wiki/File:Steer_system.jpg, (2017-05-06))

[23] (Bertil Thomas, *Modern reglerteknik*, 4th edition**,** Stockholm Liber)

[24] (Lennart Ljung, Torkel Glad, *Modeling and identification of dynamic systems*, Lund: Studentiltteratur AB)

[25] (Mike Sutton for Car and driver, *Infiniti Q50S*, 2013-10, Available: http://www.caranddriver.com/reviews/2014-infiniti-q50s-test-review, (2017-06-02))

[26] (Steve Siler for Car and driver, *2016 Infiniti Q50 Red Sport 400*, 2016-02, Available: http://www.caranddriver.com/reviews/2016-infiniti-q50-red-sport-400-first-drive-review, (2017-06-02)

Appendix 1: Main program at the beginning of the thesis.

```
#include <Controllino.h>

//ROS Includes
#include <ros.h>
#include <std_msgs/Int16.h>
#include <std_msgs/Float32.h>
#include <std_msgs/Bool.h>

#include <PID_v1.h>

#define ADJUST_GAINS


//////////////////////////////////////////
//            CONSTANTS               //
//////////////////////////////////////////

int CONTROLLER_STEERING_LEFT_MAX = 276;         //Digital input value from controller when turning fully to the left. (not used)
int CONTROLLER_STEERING_RIGHT_MAX = 420;        //Digital input value from controller when turning fully to the right. (not
used)

int                    CONTROLLER_STEERING_MIN = 276;                    //Digital input value from controller when turning fully
to the left.
int CONTROLLER_STEERING_MAX = 420;              //Digital input value from controller when turning fully to the right.

int CONTROLLER_THROTTLE_MIN = 618;
int CONTROLLER_THROTTLE_MAX = 516;

int ANALOG_IN_RESOLUTION_12V = 1023;            //Analog 12V input is converted into digital input 0-1023.
int ANALOG_IN_RESOLUTION_5V = 426;

int                    PWM_RESOLUTION_12V = 255;   //The duty-cycle is between 0 and 255 at 12V.
int PWM_RESOLUTION_5V = 106;

float MAXIMUM_STEERING_ANGLE = 25.0f;
float STEERING_ANGLE_OFFSET = 0.0f;                              // Used to calibrate the center position of the joystick.

float ROS_PUBLISH_RATE = 50.0f;                 // Delay between each ROS publish.


//////////////////////////////////////////
//            PINS                    //
//////////////////////////////////////////

// Motor controller pins
  int Pin_ThrottleOut = CONTROLLINO_D1;                  // Sends PWM signal to main motor
  int Pin_BrakeOut = CONTROLLINO_D3;                     // Actuates regenerative brake NOTE: Very weak

// Controller pins
  int Pin_ThrottleIn = CONTROLLINO_A8;                   // Throttle potentiometer on steering controller, used for manual driving
  int Pin_SteeringIn = CONTROLLINO_A9;                   // Steering potentiometer on steering controller, used for manual driving

// Steering servo controller pins
  int Pin_SteeringOut = CONTROLLINO_D2;                  // Sends PWM signal to steering motor, Higher duty-cycle => faster rotation
  int Pin_SteeringLeftOut = CONTROLLINO_D4;              // Sets turning direction for steering motor
  int Pin_SteeringRightOut = CONTROLLINO_D3;              // Sets turning direction for steering motor

// Drive mode pins
  int Pin_ForwardModeIn = CONTROLLINO_A5;
  int Pin_NeutralModeIn = CONTROLLINO_A6;
  int Pin_ReverseModeIn = CONTROLLINO_A7;
  int Pin_AutonomousModeIn = CONTROLLINO_A4;
  int Pin_ForwardModeOut = CONTROLLINO_R9;
  int Pin_ReverseModeOut = CONTROLLINO_R8;

// Feedback pins
  int Pin_SteeringEncoder = CONTROLLINO_A1;              // Linear potentiometer on steering assembly

// Light pins
  int Pin_LightSwitchIn = CONTROLLINO_A0;
  int Pin_LightLeftOut = CONTROLLINO_R13;
  int Pin_LightRightOut = CONTROLLINO_R14;
  int Pin_LightBackOut = CONTROLLINO_R15;
```

```
// Misc pins
  //int Pin_ArduinoLEDOut = CONTROLLINO_D14;
  //int Pin_AutonomousModeSwitchIn = CONTROLLINO_D11;                 // Extra control box (not used)
  //int Pin_SpeedIndicatorOut = CONTROLLINO_D2;        // Speed indicator (burned...)


/////////////////////////////////////////////
//          GLOBAL VARIABLES            //
/////////////////////////////////////////////

float OutputThrottle = 0.55f;                 // Throttle signal sent to the motor controller scaled 0-1
float OutputBrake = 0.9f;                      // Brake signal sent to the motor controller scaled 0-1

float ControllerThrottle = 0.0f;              // Throttle set by controller input
float ControllerSteeringAngle = 0.0f;     // Signal from steering potentiometer (from controller), converted to angle

float CurrentSpeed = 0.0f;                    // Current speed as reported by wheel encoders
float TargetSpeed = 0.7f;                      // Target speed set by autonomy software
float TargetBrake = 0.0f;                      // Target brake set by autonomy software
float AutonomousThrottle = 0.61f;          // Throttle set by autonomy software

float CurrentSteeringAngle = 0.0f;
float TargetSteeringAngle = 0.0f;           // TODO: Check control loop for steering; use PID?

float Autonomy_SpeedController = false;
float Autonomy_ManualSteering = false;
float Autonomy_ManualThrust = false;

bool ForwardMode = false;
bool ReverseMode = false;

// Data reported by the LIDAR
  float LidarDistance = 0.0f;
  float LidarSpeed = 0.0f;

// Variables for PID controller
  double PID_Setpoint, PID_Input, PID_Output;
  double MIN = 0.55f;
  double MAX = 0.9f;

#ifdef ADJUST_GAINS
  double PID_kp = 0.9f;
  double PID_ki = 0.01f;
  double PID_kd = 1.0f;
#endif

PID SpeedPID(&PID_Input, &PID_Output, &PID_Setpoint, PID_kp, PID_ki, PID_kd, DIRECT);


/////////////////////////////////////////////
//           ROS Variables             //
/////////////////////////////////////////////

// Node Handle
  ros::NodeHandle  ROS_NodeHandle;

// Callbacks
  void ROS_Steering_CB(const std_msgs::Float32& steering_msg) { TargetSteeringAngle = steering_msg.data; }
  ros::Subscriber<std_msgs::Float32> ROS_Steering_Sub("/angle", ROS_Steering_CB);

  void ROS_CurrentSpeed_CB(const std_msgs::Float32& currentspeed_msg) { CurrentSpeed = currentspeed_msg.data; }
  ros::Subscriber<std_msgs::Float32> ROS_CurrentSpeed_Sub("/mule_speed", ROS_CurrentSpeed_CB);

  void ROS_TargetSpeed_CB(const std_msgs::Float32& targetspeed_msg) { TargetSpeed = targetspeed_msg.data; }
  ros::Subscriber<std_msgs::Float32> ROS_TargetSpeed_Sub("/input_mule_target_speed", ROS_TargetSpeed_CB);

  void ROS_ForwardMode_CB(const std_msgs::Bool& forwardmode_msg) { ForwardMode = forwardmode_msg.data; }
  ros::Subscriber<std_msgs::Bool> ROS_ForwardMode_Sub("/input_mule_forward_mode", ROS_ForwardMode_CB);

  void ROS_ReverseMode_CB(const std_msgs::Bool& reversemode_msg) { ReverseMode = reversemode_msg.data; }
  ros::Subscriber<std_msgs::Bool> ROS_ReverseMode_Sub("/input_mule_reverse_mode", ROS_ReverseMode_CB);

  void ROS_AutonomySpeedControl_CB(const std_msgs::Bool& autonomyspeedcontrol_msg) { Autonomy_SpeedController =
autonomyspeedcontrol_msg.data; }
  ros::Subscriber<std_msgs::Bool> ROS_AutonomySpeedControl_Sub("/input_mule_autonomy_speed_control",
ROS_AutonomySpeedControl_CB);
```

```
  void ROS_AutonomyManualSteering_CB(const std_msgs::Bool& autonomymanualsteering_msg) { Autonomy_ManualSteering =
autonomymanualsteering_msg.data; }
  ros::Subscriber<std_msgs::Bool> ROS_AutonomyManualSteering_Sub("/input_mule_autonomy_manual_steering",
ROS_AutonomyManualSteering_CB);

  void ROS_AutonomyManualThrust_CB(const std_msgs::Bool& autonomymanualthrust_msg) { Autonomy_ManualThrust =
autonomymanualthrust_msg.data; }
  ros::Subscriber<std_msgs::Bool> ROS_AutonomyManualThrust_Sub("/input_mule_autonomy_manual_steering",
ROS_AutonomyManualThrust_CB);

  void ROS_Throttle_CB(const std_msgs::Float32& throttle_msg)
  {
    OutputThrottle = throttle_msg.data;
    if(OutputThrottle < 0.55f)
      TargetBrake = 0.9f;
    else
      TargetBrake = 0.0f;
  }
  ros::Subscriber<std_msgs::Float32> ROS_Throttle_Sub("/input_mule_throttle", ROS_Throttle_CB);

  void ROS_TargetBrake_CB(const std_msgs::Bool& targetbrake_msg)
  {
    TargetBrake = ((targetbrake_msg.data == true) * 0.9f);
  }
  ros::Subscriber<std_msgs::Bool> ROS_TargetBrake_Sub("/brake", ROS_TargetBrake_CB);

  void ROS_LidarDistance_CB(const std_msgs::Float32& lidardistance_msg) { LidarDistance = lidardistance_msg.data; }
  ros::Subscriber<std_msgs::Float32> ROS_LidarDistance_Sub("/lidar_distance", ROS_LidarDistance_CB);

  void ROS_LidarSpeed_CB(const std_msgs::Float32& lidarspeed_msg) { LidarSpeed = lidarspeed_msg.data; }
  ros::Subscriber<std_msgs::Float32> ROS_LidarSpeed_Sub("/lidar_speed", ROS_LidarSpeed_CB);


// Publishers
  std_msgs::Float32 targetspeed_msg;
  ros::Publisher ROS_TargetSpeed_Pub("/mule_target_speed", &targetspeed_msg);

  std_msgs::Float32 throttle_msg;
  ros::Publisher ROS_Throttle_Pub("/mule_throttle", &throttle_msg);

  std_msgs::Float32 current_steering_msg;
  ros::Publisher ROS_Current_Steering_Pub("/output_mule_current_steering", &current_steering_msg);

  std_msgs::Float32 target_steering_msg;
  ros::Publisher ROS_Target_Steering_Pub("/output_mule_target_steering", &target_steering_msg);


// PID control
  #ifdef ADJUST_GAINS
    void UpdatePIDGains() { SpeedPID.SetTunings(PID_kp, PID_ki, PID_kd); }

    void ROS_PID_kp_CB(const std_msgs::Float32& kp_msg) { PID_kp = kp_msg.data; UpdatePIDGains(); }
    ros::Subscriber<std_msgs::Float32> ROS_PID_kp_Sub("/kp", ROS_PID_kp_CB);

    void ROS_PID_ki_CB(const std_msgs::Float32& ki_msg) { PID_ki = ki_msg.data; UpdatePIDGains(); }
    ros::Subscriber<std_msgs::Float32> ROS_PID_ki_Sub("/ki", ROS_PID_ki_CB);

    void ROS_PID_kd_CB(const std_msgs::Float32& kd_msg) { PID_kd = kd_msg.data; UpdatePIDGains(); }
    ros::Subscriber<std_msgs::Float32> ROS_PID_kd_Sub("/kd", ROS_PID_kd_CB);
  #endif


//////////////////////////////////////////
//              Setup                   //
//////////////////////////////////////////

void setup()
{
  ROS_NodeHandle.initNode();

  ROS_NodeHandle.advertise(ROS_TargetSpeed_Pub);
  ROS_NodeHandle.advertise(ROS_Throttle_Pub);
  ROS_NodeHandle.advertise(ROS_Current_Steering_Pub);
  ROS_NodeHandle.advertise(ROS_Target_Steering_Pub);
  //ROS_NodeHandle.advertise(ROS_Distance_Pub);
  ROS_NodeHandle.subscribe(ROS_Steering_Sub);
  ROS_NodeHandle.subscribe(ROS_CurrentSpeed_Sub);
  ROS_NodeHandle.subscribe(ROS_TargetSpeed_Sub);
  ROS_NodeHandle.subscribe(ROS_TargetBrake_Sub);
  ROS_NodeHandle.subscribe(ROS_LidarDistance_Sub);
  ROS_NodeHandle.subscribe(ROS_LidarSpeed_Sub);
  ROS_NodeHandle.subscribe(ROS_Throttle_Sub);
  ROS_NodeHandle.subscribe(ROS_ForwardMode_Sub);
  ROS_NodeHandle.subscribe(ROS_ReverseMode_Sub);
  ROS_NodeHandle.subscribe(ROS_AutonomySpeedControl_Sub);
  ROS_NodeHandle.subscribe(ROS_AutonomyManualSteering_Sub);
  ROS_NodeHandle.subscribe(ROS_AutonomyManualThrust_Sub);

  #ifdef ADJUST_GAINS
    ROS_NodeHandle.subscribe(ROS_PID_kp_Sub);
    ROS_NodeHandle.subscribe(ROS_PID_ki_Sub);
    ROS_NodeHandle.subscribe(ROS_PID_kd_Sub);
  #endif

//  //Analog Outputs
//  //pinMode(Pin_SpeedIndicatorOut, OUTPUT);
//  pinMode(Pin_SteeringOut, OUTPUT);
//  pinMode(Pin_ThrottleOut, OUTPUT);
//  pinMode(Pin_BrakeOut, OUTPUT);
//
//  //Digital Inputs
//  pinMode(Pin_ForwardModeIn, INPUT);
//  pinMode(Pin_NeutralModeIn, INPUT);
//  pinMode(Pin_ReverseModeIn, INPUT);
//  pinMode(Pin_AutonomousModeIn, INPUT);
//
//  //Digital Outputs
//  pinMode(Pin_ForwardModeOut, OUTPUT);
//  pinMode(Pin_ReverseModeOut, OUTPUT);
//  pinMode(Pin_SteeringLeftOut, OUTPUT);
//  pinMode(Pin_SteeringRightOut, OUTPUT);
//  //pinMode(Pin_ArduinoLEDOut, OUTPUT);
//
//  digitalWrite(Pin_SteeringLeftOut, LOW);
//  digitalWrite(Pin_SteeringRightOut, LOW);
//
//  //analogWrite(Pin_SpeedIndicatorOut, 4);

  // Set PWM frequencies (required for low pass filter PWM to DC)
    TCCR1B = (TCCR1B & 0xF8) | 0x01;
    TCCR2B = (TCCR2B & 0xF8) | 0x01;
    TCCR3B = (TCCR3B & 0xF8) | 0x01;
    TCCR4B = (TCCR4B & 0xF8) | 0x02;

  //make sure controller output is within PWM range
  SpeedPID.SetOutputLimits(MIN, MAX);

  //turn the PID on
  SpeedPID.SetMode(AUTOMATIC);
}


/////////////////////////////////////////
//              Loop                  //
/////////////////////////////////////////

void loop()
{
  ReadSteeringEncoder();
  GetControllerInput();
  LightControl();

  if(digitalRead(Pin_AutonomousModeIn))
    RunAutonomousMode();
  else
  {
    GetConsoleInput();
    ManualSteering();
    ManualThrust();
  }
  RunCar();

  PublishROSData();

  ROS_NodeHandle.spinOnce();
}

void RunCar()
{
  if(ForwardMode)
    RunForwardMode();
  else if(ReverseMode)
    RunReverseMode();
  else
    RunIdle();

  analogWrite(Pin_ThrottleOut, OutputThrottle*PWM_RESOLUTION_5V);
  analogWrite(Pin_BrakeOut, OutputBrake*PWM_RESOLUTION_5V);
  float SteeringAdjustment = (CurrentSteeringAngle-TargetSteeringAngle)/10.0f; //Proportional controller. The higher value the faster
//rotation on the servo motor.
  SteeringAdjustment                    *= PWM_RESOLUTION_5V;           //Rescale to fit the wanted PWM signal.

if(SteeringAdjustment                   > PWM_RESOLUTION_5V)            //Makes sure that the max value isn't exceeded.
                                                                        SteeringAdjustment = PWM_RESOLUTION_5V;

  if(SteeringAdjustment < -PWM_RESOLUTION_5V)
    SteeringAdjustment = -PWM_RESOLUTION_5V;

  if(abs(SteeringAdjustment) < 100)
    SteeringAdjustment /= 10;

if(SteeringAdjustment > 0) //The PWM signal shall look the same regardless the turning direction, the sign of SteeringAdjustment
decides                                                     //the direction.
  {
    digitalWrite(Pin_SteeringRightOut, HIGH);
    digitalWrite(Pin_SteeringLeftOut, LOW);
    analogWrite(Pin_SteeringOut, SteeringAdjustment);
  }
  else if(SteeringAdjustment < 0)
  {
    digitalWrite(Pin_SteeringLeftOut, HIGH);
    digitalWrite(Pin_SteeringRightOut, LOW);
    analogWrite(Pin_SteeringOut, -SteeringAdjustment);
  }
  else
  {
    analogWrite(Pin_SteeringOut, 0);
  }
}


//////////////////////////////////////////
//          Manual Drive            //
//////////////////////////////////////////

void GetConsoleInput()
{
  if(!digitalRead(Pin_NeutralModeIn))
  {
    ForwardMode = digitalRead(Pin_ForwardModeIn);
    ReverseMode = digitalRead(Pin_ReverseModeIn);
  }
}

void ManualSteering()
{
  TargetSteeringAngle = ControllerSteeringAngle;
}

void ManualThrust()
{
  if(ControllerThrottle < 0.1f)
  {
                                                   OutputBrake = 1.0f;
                                 OutputThrottle = 0.55f;                //The motor starts rotating at 0.55 duty-cycle.
  }
  else if(ControllerThrottle > 0.2f)
  {
    OutputBrake = 0.0f;
    OutputThrottle = 0.55f + ControllerThrottle*ControllerThrottle*0.35f;
  }
  else
  {
    OutputBrake = 0.0f;
    OutputThrottle = 0.55f;
  }
}


//////////////////////////////////////////
//              Autonomy                //
//////////////////////////////////////////

void RunAutonomousMode()
{
  if(Autonomy_SpeedController)
  {
    PID_Setpoint = TargetSpeed;
    PID_Input = CurrentSpeed;
    SpeedPID.Compute();
  }
  if(Autonomy_ManualSteering)
    ManualSteering();
  if(Autonomy_ManualThrust)
    ManualThrust();

//  if(ControllerThrottle > 0.5f)
//  {
//    if(TargetBrake == 0.0f)
//    {
//      KeepDistance(3.0f, 1.0f, 1.0f);
//
//      PID_Setpoint = TargetSpeed;
//      PID_Input = CurrentSpeed;
//      SpeedPID.Compute();
//
//      OutputThrottle = PID_Output;
//      OutputBrake = 0.0f;
//    }
//    else
//    {
//      OutputThrottle = 0.0f;
//      OutputBrake = 0.9f;
//    }
//  }
//  else
//  {
//    OutputThrottle = 0.0f;
//    OutputBrake = 0.9f;
//  }
}

void KeepDistance(float DistanceToKeep, float ApproachSpeed, float DistanceTolerance)
{
  if(LidarDistance < 0.5f)
    LidarDistance = 100.0f;

  if(LidarDistance < 30.0f)
  {
    if(LidarDistance < DistanceToKeep-DistanceTolerance*2)
      TargetSpeed = LidarSpeed + CurrentSpeed - ApproachSpeed;
    else if(LidarDistance < DistanceToKeep-DistanceTolerance)
      TargetSpeed = LidarSpeed + CurrentSpeed - ApproachSpeed*(DistanceToKeep-LidarDistance);
    else if(LidarDistance > DistanceToKeep+DistanceTolerance*2)
      TargetSpeed = LidarSpeed + CurrentSpeed + ApproachSpeed;
    else if(LidarDistance > DistanceToKeep+DistanceTolerance)
      TargetSpeed = LidarSpeed + CurrentSpeed + ApproachSpeed*(LidarDistance-DistanceToKeep);
    else
      TargetSpeed = LidarSpeed + CurrentSpeed;
  }
  else if(TargetSpeed < 2.0f)
```

```
    TargetSpeed = 2.0f;
}


//////////////////////////////////////////
//              Steering                //
//////////////////////////////////////////

float SmoothThrottleInput = 0; // Smoothed signal from throttle potentiometer, range: 0 - 1023
float SmoothSteeringInput = 0; // Smoothed signal from steering potentiometer (from controller), range: 0 - 1023
void GetControllerInput()
{
  SmoothSteeringInput = SmoothSteeringInput*0.9f + 0.1f*analogRead(Pin_SteeringIn);     //Last value*0.9 + 0.1* current value =>
filtration
  SmoothThrottleInput = SmoothThrottleInput*0.9f + 0.1f*analogRead(Pin_ThrottleIn);

ControllerSteeringAngle = -(((SmoothSteeringInput-CONTROLLER_STEERING_MIN)/(CONTROLLER_STEERING_MAX-
CONTROLLER_STEERING_MIN))*2.0f-1.0f)*MAXIMUM_STEERING_ANGLE + STEERING_ANGLE_OFFSET; //rescale from
276-420(0-1023) to
    //-25 - 25
  ControllerThrottle = 1.0f-((SmoothThrottleInput-CONTROLLER_THROTTLE_MIN)/(CONTROLLER_THROTTLE_MAX-
CONTROLLER_THROTTLE_MIN));
}

void ReadSteeringEncoder()
{
  int SteeringEncoderVal = analogRead(Pin_SteeringEncoder);

                                                CurrentSteeringAngle = CurrentSteeringAngle*0.0f +
1.0f*((SteeringEncoderVal-313.0f)/(926.0f-313.0f)*2.0f - 1.0f)*30.84f;          //rescales from 313 - 926(0-1023) to -30.84 - 30.84
}


//////////////////////////////////////////
//            Control Panel             //
//////////////////////////////////////////

void LightControl()
{
  if (digitalRead(Pin_LightSwitchIn) == HIGH)
  {
    digitalWrite(Pin_LightLeftOut, HIGH);
    digitalWrite(Pin_LightRightOut, HIGH);
    digitalWrite(Pin_LightBackOut, HIGH);
  }
  else
  {
    digitalWrite(Pin_LightLeftOut, LOW);
    digitalWrite(Pin_LightRightOut, LOW);
    digitalWrite(Pin_LightBackOut, LOW);
  }
}

void RunForwardMode()
{
  digitalWrite(Pin_ForwardModeOut, HIGH);
  digitalWrite(Pin_ReverseModeOut, LOW);
}

void RunReverseMode()
{
  digitalWrite(Pin_ForwardModeOut, LOW);
  digitalWrite(Pin_ReverseModeOut, HIGH);
}

void RunIdle()
{
  digitalWrite(Pin_ForwardModeOut, LOW);
  digitalWrite(Pin_ReverseModeOut, LOW);
}


//////////////////////////////////////////
//            ROS Publisher             //
//////////////////////////////////////////
```

```
unsigned long LastTimeROSPublish = 0;
void PublishROSData()
{
  unsigned long CurrentTime = millis();

  if (CurrentTime - LastTimeROSPublish >= ROS_PUBLISH_RATE)
  {
    unsigned long ElapsedTime = CurrentTime - LastTimeROSPublish;

    current_steering_msg.data = CurrentSteeringAngle;
    ROS_Current_Steering_Pub.publish(&current_steering_msg);

    target_steering_msg.data = TargetSteeringAngle;
    ROS_Target_Steering_Pub.publish(&target_steering_msg);

    targetspeed_msg.data = TargetSpeed;
    ROS_TargetSpeed_Pub.publish(&targetspeed_msg);

    throttle_msg.data = OutputThrottle;
    ROS_Throttle_Pub.publish(&throttle_msg);

    LastTimeROSPublish = millis();
  }
}
```

Appendix 2: Lidar control and speed measurement at the beginning of the thesis.

```
#include <LIDARLite.h>

#include <ros.h>
#include <std_msgs/Float32.h>


////////////////////////////////////////////
//              CONSTANTS                 //
////////////////////////////////////////////

#define WHEEL_DIAMETER 0.535
#define WHEEL_ENCODER_TICKS 40

#define LIDAR_SPEED_CALCULATION_RATE 20.0
#define LIDAR_PUBLISH_RATE 50.0
#define WHEEL_ENCODER_RATE 50.0

#define PWM_RESOLUTION_12V 255
#define PWM_RESOLUTION_5V 106


////////////////////////////////////////////
//              PINS                      //
////////////////////////////////////////////

#define Pin_LeftWheelEncoder 2 // Rotary encoder on wheel
#define Pin_RightWheelEncoder 3 // Rotary encoder on wheel


////////////////////////////////////////////
//           GLOBAL VARIABLES             //
////////////////////////////////////////////

// Variables for LIDAR
LIDARLite Lidar;
float RawLidarDistance = 0.0f;
float LidarDistance = 0.0f;
float LastLidarDistance = 0.0f;
float LidarSpeed = 0.0f;

volatile int LeftWheelEncoderCounter = 0;
volatile int RightWheelEncoderCounter = 0;
volatile unsigned long LastTimeLeftWheelEncoderTick = 0;
volatile unsigned long LastTimeRightWheelEncoderTick = 0;
volatile unsigned long LastTimeLeftWheelEncoderTickTime = 0;
volatile unsigned long LastTimeRightWheelEncoderTickTime = 0;

float LeftWheelSpeed = 0.0f;
float RightWheelSpeed = 0.0f;
float CurrentSpeed = 0.0f;

double DistanceTravelled = 0.0f;


////////////////////////////////////////////
//           ROS Variables                //
////////////////////////////////////////////

// Node Handle
  ros::NodeHandle  ROS_NodeHandle;

// Publishers
  std_msgs::Float32 lidar_dist_msg;
  ros::Publisher ROS_Lidar_Distance_Pub("/lidar_distance", &lidar_dist_msg);

  std_msgs::Float32 lidar_speed_msg;
  ros::Publisher ROS_Lidar_Speed_Pub("/lidar_speed", &lidar_speed_msg);

  std_msgs::Float32 distance_travelled_msg;
  ros::Publisher ROS_DistanceTravelled_Pub("/distance_travelled", &distance_travelled_msg);

  std_msgs::Float32 speed_msg;
  ros::Publisher ROS_Speed_Pub("/mule_speed", &speed_msg);


void setup()
{
  ROS_NodeHandle.initNode();

  ROS_NodeHandle.advertise(ROS_Lidar_Distance_Pub);
  ROS_NodeHandle.advertise(ROS_Lidar_Speed_Pub);

  ROS_NodeHandle.advertise(ROS_DistanceTravelled_Pub);
  ROS_NodeHandle.advertise(ROS_Speed_Pub);

 Lidar.begin(0, true); // Set configuration to default and I2C to 400 kHz
 Lidar.configure(0); // Change this number to try out alternate

  //Interrupts
  attachInterrupt(digitalPinToInterrupt(Pin_LeftWheelEncoder), LeftWheelEncoderTransition, CHANGE);
  attachInterrupt(digitalPinToInterrupt(Pin_RightWheelEncoder), RightWheelEncoderTransition, CHANGE);

  LastTimeLeftWheelEncoderTickTime = micros();
  LastTimeRightWheelEncoderTickTime = micros();

}

void loop()
{
  GetLidarDistance();
  CalculateLidarSpeed();
  PublishLidarData();

  ReadWheelEncoders();

  ROS_NodeHandle.spinOnce();
}

void GetLidarDistance()
{
  RawLidarDistance = Lidar.distance(true)*0.01f;
  LidarDistance = 0.9f*LidarDistance + 0.1f*RawLidarDistance;
}


unsigned long LastTimeLidarSpeed = 0;
void CalculateLidarSpeed()
{
  float CurrentTime = millis();
  if(CurrentTime - LastTimeLidarSpeed > LIDAR_SPEED_CALCULATION_RATE)
  {
    float dt = LastTimeLidarSpeed-CurrentTime;
    LastTimeLidarSpeed = CurrentTime;

    LidarSpeed = LidarSpeed*0.9f + 0.1f*(LastLidarDistance-LidarDistance)/(dt/1000.0f);

    LastLidarDistance = LidarDistance;
  }
}


unsigned long LastTimeLidarPublish = 0;
void PublishLidarData()
{
  float CurrentTime = millis();
  if(CurrentTime - LastTimeLidarPublish > LIDAR_PUBLISH_RATE)
  {
    LastTimeLidarPublish = CurrentTime;

    lidar_dist_msg.data = LidarDistance;
    ROS_Lidar_Distance_Pub.publish(&lidar_dist_msg);

    lidar_speed_msg.data = LidarSpeed;
    ROS_Lidar_Speed_Pub.publish(&lidar_speed_msg);
  }
}

unsigned long LastTimeWheelEncoder = 0;
```

```
void ReadWheelEncoders()
{
  unsigned long CurrentTime = millis();

  if (CurrentTime - LastTimeWheelEncoder >= WHEEL_ENCODER_RATE)
  {
    unsigned long ElapsedTime = CurrentTime - LastTimeWheelEncoder;

    detachInterrupt(Pin_LeftWheelEncoder);                   //Disable interrupt when calculating
    detachInterrupt(Pin_RightWheelEncoder);                  //Disable interrupt when calculating

      LeftWheelSpeed = 0.9f*LeftWheelSpeed +
0.1f*(WHEEL_DIAMETER*PI/WHEEL_ENCODER_TICKS)/LastTimeLeftWheelEncoderTick*1000000; //0.9*previous +0.1*current
=> filtration
      RightWheelSpeed = 0.9f*RightWheelSpeed +
0.1f*(WHEEL_DIAMETER*PI/WHEEL_ENCODER_TICKS)/LastTimeRightWheelEncoderTick*1000000;
      CurrentSpeed = (LeftWheelSpeed + RightWheelSpeed)/2/1.25;

      //analogWrite(Pin_SpeedIndicatorOut, 4+(AnalogRes-4)*((CurrentSpeed*3.6f)/131.5f/4));

      DistanceTravelled = (LeftWheelEncoderCounter+RightWheelEncoderCounter)/2 *
(WHEEL_DIAMETER*PI/WHEEL_ENCODER_TICKS);

      distance_travelled_msg.data = DistanceTravelled;
      ROS_DistanceTravelled_Pub.publish(&distance_travelled_msg);

      speed_msg.data = CurrentSpeed;
      ROS_Speed_Pub.publish(&speed_msg);

      LastTimeWheelEncoder = millis(); // Update lastmillis

    attachInterrupt(digitalPinToInterrupt(Pin_LeftWheelEncoder), LeftWheelEncoderTransition, CHANGE);          //enable
interrupt
    attachInterrupt(digitalPinToInterrupt(Pin_RightWheelEncoder), RightWheelEncoderTransition, CHANGE);        //enable
interrupt
  }
}


void LeftWheelEncoderTransition()
{
  LeftWheelEncoderCounter++;
  float CurrentTime = micros();
  LastTimeLeftWheelEncoderTickTime = LastTimeLeftWheelEncoderTick-CurrentTime;
  LastTimeLeftWheelEncoderTick = CurrentTime;
}

void RightWheelEncoderTransition()
{
  RightWheelEncoderCounter++;
  float CurrentTime = micros();
  LastTimeRightWheelEncoderTickTime = LastTimeRightWheelEncoderTick-CurrentTime;
  LastTimeRightWheelEncoderTick = CurrentTime;
}
```

Appendix 3: The program at the end of the thesis

```
#include <Controllino.h>

//ROS Includes
#include <ros.h>
#include <std_msgs/Int16.h>
#include <std_msgs/Float32.h>
#include <std_msgs/Bool.h>

#include <PID_v1.h>

#include <math.h> //Used for the speed dependency for TargetSteeringAngle

#define ADJUST_GAINS


////////////////////////////////////////////
//            CONSTANTS                 //
////////////////////////////////////////////

#define CONTROLLER_STEERING_MIN 384       //Value of the input at minimum angle
#define CONTROLLER_STEERING_MAX 246

#define CONTROLLER_THROTTLE_MIN 547.0  //Value of the input at 0 speed
#define CONTROLLER_THROTTLE_MAX 442.0

#define WHEEL_STEERING_MIN 747.0  //Value of the input at minimum angle
#define WHEEL_STEERING_MAX 241.0

#define ANALOG_IN_RESOLUTION_12V 1023
#define ANALOG_IN_RESOLUTION_5V 426

#define PWM_RESOLUTION_12V 255
#define PWM_RESOLUTION_4_5V 88       // Voltage in max 13V. Voltage out max 4.5V. Duty_Max*Vout/Vin = 255*4.5/13=88

#define CONTROLLER_MAX_ANGLE 45.0
#define WHEEL_MAX_ANGLE 26.65

#define STEERING_RATIO (WHEEL_MAX_ANGLE / CONTROLLER_MAX_ANGLE)      //Ratio between joystick- and wheel range
(Joystick range)*STEERING_RATIO = (Wheel range)

#define WHEEL_DIAMETER 0.535
#define WHEEL_ENCODER_TICKS 40
#define WHEEL_ENCODER_RATE 10000.0

////////////////////////////////////////////
//            PINS                  //
////////////////////////////////////////////

// Motor controller pins
  int Pin_ThrottleOut = CONTROLLINO_D10; // Sends PWM signal to main motor
  int Pin_BrakeOut = CONTROLLINO_D6; // Actuates regenerative brake NOTE: Very weak

// Controller pins
  int Pin_ThrottleIn = CONTROLLINO_A7; // Throttle potentiometer on steering controller, used for manual driving
  int Pin_SteeringIn = CONTROLLINO_A6; // Steering potentiometer on steering controller, used for manual driving

// Steering servo controller pins
  int Pin_SteeringOut = CONTROLLINO_D0; // Sends PWM signal to steering motor, H�gre v�rde => Snabbare rotation
  int Pin_SteeringLeftOut = CONTROLLINO_D2; // Sets turning direction for steering motor
  int Pin_SteeringRightOut = CONTROLLINO_D1; // Sets turning direction for steering motor

// Drive mode pins
  int Pin_ForwardModeIn = CONTROLLINO_A0;
  int Pin_NeutralModeIn = CONTROLLINO_A3;
  int Pin_ReverseModeIn = CONTROLLINO_A1;
  int Pin_AutonomousModeIn = CONTROLLINO_A2;
  int Pin_ForwardModeOut = CONTROLLINO_D4;
  int Pin_ReverseModeOut = CONTROLLINO_D3;

// Feedback pins
  int Pin_SteeringEncoder = CONTROLLINO_A8; // Linear potentiometer on steering assembly
  int Pin_LeftWheelEncoder = digitalPinToInterrupt(CONTROLLINO_IN0); // Rotary encoder on wheel
  int Pin_RightWheelEncoder = digitalPinToInterrupt(CONTROLLINO_IN1); // Rotary encoder on wheel
```

```
// Light pins
  int Pin_LightSwitchIn = CONTROLLINO_A0;
  int Pin_LightLeftOut = CONTROLLINO_R9;
  int Pin_LightRightOut = CONTROLLINO_R9;
  int Pin_LightBackOut = CONTROLLINO_R9;

// Misc pins
    int Pin_Fans = CONTROLLINO_D7;
  //int Pin_Stable12V = CONTROLLINO_D15;
    int Pin_EmergencyStop = CONTROLLINO_A9;
    int Pin_MotorController = CONTROLLINO_R6;
  //int Pin_ArduinoLEDOut = CONTROLLINO_D14;
  //int Pin_AutonomousModeSwitchIn = CONTROLLINO_D11; // Extra control box (not used)
  //int Pin_SpeedIndicatorOut = CONTROLLINO_D2; // Speed indicator (burned...)


////////////////////////////////////////////
//          GLOBAL VARIABLES            //
////////////////////////////////////////////

float OutputThrottle = 0.55f; // Throttle signal sent to the motor controller scaled 0-1
float OutputBrake = 0.9f; // Brake signal sent to the motor controller scaled 0-1

float ControllerThrottle = 0.0f; // Throttle set by controller input
float ControllerSteeringAngle = 0.0f; // Signal from steering potentiometer (from controller), converted to angle
bool Safe = true;

float CurrentSpeed = 0.0f; // Current speed as reported by wheel encoders, m/s
float TargetSpeed = 0.7f; // Target speed set by autonomy software
float TargetBrake = 0.0f; // Target brake set by autonomy software
float AutonomousThrottle = 0.61f; // Throttle set by autonomy software
float SmoothThrottleInput = 0; // Smoothed signal from throttle potentiometer, range: 0 - 1023

float CurrentSteeringAngle = 0.0f;
float TargetSteeringAngle = 0.0f;
float SteeringAdjustmentP = 0.0f;
float SteeringAdjustmentI = 0.0f;
float SteeringAdjustmentD = 0.0f;
float SteeringAdjustment = 0.0f;
float LastDeviation = 0.0f; //Deviation between CurrentSteeringAngle and TargetSteeringAngle one sample period ago.
float SmoothSteeringInput = 0; // Smoothed signal from steering potentiometer (from controller), range: 0 - 1023
float SmoothSteeringInputOld = 0;
float OldSteeringInput = 0;
float Autonomy_SpeedController = false;
float Autonomy_ManualSteering = false;
float Autonomy_ManualThrust = false;

unsigned long LastTime = 0;

bool ForwardMode = false;
bool ReverseMode = false;

//Speed measurement
volatile int LeftWheelEncoderCounter = 0;
volatile int RightWheelEncoderCounter = 0;
volatile unsigned long LastTimeLeftWheelEncoderTick = micros();
volatile unsigned long LastTimeRightWheelEncoderTick = micros();
volatile unsigned long LastTimeLeftWheelEncoderTickTime = 10000000;
volatile unsigned long LastTimeRightWheelEncoderTickTime = 10000000;

float LeftWheelSpeed = 0.0f;
float RightWheelSpeed = 0.0f;

double DistanceTravelled = 0.0f;

unsigned long LastTimeWheelEncoder = 0;

volatile int wwert = 0;
// Data reported by the LIDAR
  float LidarDistance = 0.0f;
  float LidarSpeed = 0.0f;

// Variables for PID controller for speed
  double PID_Setpoint, PID_Input, PID_Output;
  double MIN = 0.55f;
  double MAX = 0.9f;
```

```
#ifdef ADJUST_GAINS
  double PIDspeed_kp = 0.9f;
  double PID_ki = 0.01f;
  double PID_kd = 1.0f;
#endif

PID SpeedPID(&PID_Input, &PID_Output, &PID_Setpoint, PIDspeed_kp, PID_ki, PID_kd, DIRECT);

////////////////////////////////////////////
//            ROS Variables              //
////////////////////////////////////////////

// Node Handle
  ros::NodeHandle  ROS_NodeHandle;

// Callbacks
  void ROS_Steering_CB(const std_msgs::Float32& steering_msg) { TargetSteeringAngle = steering_msg.data; }
  ros::Subscriber<std_msgs::Float32> ROS_Steering_Sub("/remote_mule_steering_angle", ROS_Steering_CB);

  void ROS_TargetSpeed_CB(const std_msgs::Float32& targetspeed_msg) { TargetSpeed = targetspeed_msg.data; }
  ros::Subscriber<std_msgs::Float32> ROS_TargetSpeed_Sub("/input_mule_target_speed", ROS_TargetSpeed_CB);

  void ROS_ForwardMode_CB(const std_msgs::Bool& forwardmode_msg) { ForwardMode = forwardmode_msg.data; }
  ros::Subscriber<std_msgs::Bool> ROS_ForwardMode_Sub("/input_mule_forward_mode", ROS_ForwardMode_CB);

  void ROS_ReverseMode_CB(const std_msgs::Bool& reversemode_msg) { ReverseMode = reversemode_msg.data; }
  ros::Subscriber<std_msgs::Bool> ROS_ReverseMode_Sub("/input_mule_reverse_mode", ROS_ReverseMode_CB);

  void ROS_AutonomySpeedControl_CB(const std_msgs::Bool& autonomyspeedcontrol_msg) { Autonomy_SpeedController = 
autonomyspeedcontrol_msg.data; }
  ros::Subscriber<std_msgs::Bool> ROS_AutonomySpeedControl_Sub("/input_mule_autonomy_speed_control", 
ROS_AutonomySpeedControl_CB);

  void ROS_AutonomyManualSteering_CB(const std_msgs::Bool& autonomymanualsteering_msg) { Autonomy_ManualSteering = 
autonomymanualsteering_msg.data; }
  ros::Subscriber<std_msgs::Bool> ROS_AutonomyManualSteering_Sub("/input_mule_autonomy_manual_steering", 
ROS_AutonomyManualSteering_CB);

  void ROS_AutonomyManualThrust_CB(const std_msgs::Bool& autonomymanualthrust_msg) { Autonomy_ManualThrust = 
autonomymanualthrust_msg.data; }
  ros::Subscriber<std_msgs::Bool> ROS_AutonomyManualThrust_Sub("/input_mule_autonomy_manual_steering", 
ROS_AutonomyManualThrust_CB);

  void ROS_Throttle_CB(const std_msgs::Float32& throttle_msg)
  {
    float RawThrottle = abs(throttle_msg.data);

    if(throttle_msg.data < -0.1f)
    {
      ForwardMode = false;
      ReverseMode = true;
    }
    else if(throttle_msg.data > 0.1f){
      ForwardMode = true;
      ReverseMode = false;
    }
    else
    {
      ForwardMode = false;
      ReverseMode = false;
    }

    if(RawThrottle < 0.1f)
    {
      OutputBrake = 1.0f;
      OutputThrottle = 0.0f;
    }
    else if(RawThrottle > 0.2f)
    {
      OutputBrake = 0.0f;
      OutputThrottle = 0.15f + RawThrottle*0.6;
    }
    else
    {
      OutputBrake = 0.0f;
      OutputThrottle = 0.0f;
    }
  }
  ros::Subscriber<std_msgs::Float32> ROS_Throttle_Sub("/remote_mule_throttle", ROS_Throttle_CB);

  void ROS_TargetBrake_CB(const std_msgs::Bool& targetbrake_msg)
  {
    TargetBrake = ((targetbrake_msg.data == true) * 0.9f);
  }
  ros::Subscriber<std_msgs::Bool> ROS_TargetBrake_Sub("/brake", ROS_TargetBrake_CB);

  void ROS_LidarDistance_CB(const std_msgs::Float32& lidardistance_msg) { LidarDistance = lidardistance_msg.data; }
  ros::Subscriber<std_msgs::Float32> ROS_LidarDistance_Sub("/lidar_distance", ROS_LidarDistance_CB);

  void ROS_LidarSpeed_CB(const std_msgs::Float32& lidarspeed_msg) { LidarSpeed = lidarspeed_msg.data; }
  ros::Subscriber<std_msgs::Float32> ROS_LidarSpeed_Sub("/lidar_speed", ROS_LidarSpeed_CB);


// Publishers
  std_msgs::Float32 targetspeed_msg;
  ros::Publisher ROS_TargetSpeed_Pub("/mule_target_speed", &targetspeed_msg);

  std_msgs::Float32 throttle_msg;
  ros::Publisher ROS_Throttle_Pub("/mule_throttle", &throttle_msg);

  std_msgs::Float32 current_steering_msg;
  ros::Publisher ROS_Current_Steering_Pub("/output_mule_current_steering", &current_steering_msg);

  std_msgs::Float32 target_steering_msg;
  ros::Publisher ROS_Target_Steering_Pub("/output_mule_target_steering", &target_steering_msg);

  std_msgs::Float32 steering_adjustment_msg;
  ros::Publisher ROS_Steering_Adjustment_Pub("/output_mule_steering_adjustment", &steering_adjustment_msg);

  std_msgs::Float32 distance_travelled_msg;
  ros::Publisher ROS_DistanceTravelled_Pub("/distance_travelled", &distance_travelled_msg);

  std_msgs::Float32 speed_msg;
  ros::Publisher ROS_Speed_Pub("/mule_speed", &speed_msg);

// PID control
  #ifdef ADJUST_GAINS
    void UpdatePIDGains() { SpeedPID.SetTunings(PIDspeed_kp, PID_ki, PID_kd); }

    void ROS_PIDspeed_kp_CB(const std_msgs::Float32& kp_msg) { PIDspeed_kp = kp_msg.data; UpdatePIDGains(); }
    ros::Subscriber<std_msgs::Float32> ROS_PIDspeed_kp_Sub("/kp", ROS_PIDspeed_kp_CB);

    void ROS_PID_ki_CB(const std_msgs::Float32& ki_msg) { PID_ki = ki_msg.data; UpdatePIDGains(); }
    ros::Subscriber<std_msgs::Float32> ROS_PID_ki_Sub("/ki", ROS_PID_ki_CB);

    void ROS_PID_kd_CB(const std_msgs::Float32& kd_msg) { PID_kd = kd_msg.data; UpdatePIDGains(); }
    ros::Subscriber<std_msgs::Float32> ROS_PID_kd_Sub("/kd", ROS_PID_kd_CB);
  #endif


//////////////////////////////////////////////
//                Setup                   //
//////////////////////////////////////////////

void setup()
{
  ROS_NodeHandle.initNode();

  ROS_NodeHandle.advertise(ROS_TargetSpeed_Pub);
  ROS_NodeHandle.advertise(ROS_Throttle_Pub);
  ROS_NodeHandle.advertise(ROS_Current_Steering_Pub);
  ROS_NodeHandle.advertise(ROS_Target_Steering_Pub);
  ROS_NodeHandle.advertise(ROS_Steering_Adjustment_Pub);
  //ROS_NodeHandle.advertise(ROS_Distance_Pub);
  ROS_NodeHandle.advertise(ROS_DistanceTravelled_Pub);
  ROS_NodeHandle.advertise(ROS_Speed_Pub);

  ROS_NodeHandle.subscribe(ROS_Steering_Sub);
  ROS_NodeHandle.subscribe(ROS_TargetSpeed_Sub);
  ROS_NodeHandle.subscribe(ROS_TargetBrake_Sub);
  ROS_NodeHandle.subscribe(ROS_LidarDistance_Sub);
  ROS_NodeHandle.subscribe(ROS_LidarSpeed_Sub);
  ROS_NodeHandle.subscribe(ROS_Throttle_Sub);
  ROS_NodeHandle.subscribe(ROS_ForwardMode_Sub);
  ROS_NodeHandle.subscribe(ROS_ReverseMode_Sub);
  ROS_NodeHandle.subscribe(ROS_AutonomySpeedControl_Sub);
  ROS_NodeHandle.subscribe(ROS_AutonomyManualSteering_Sub);
  ROS_NodeHandle.subscribe(ROS_AutonomyManualThrust_Sub);

  #ifdef ADJUST_GAINS
    ROS_NodeHandle.subscribe(ROS_PIDspeed_kp_Sub);
    ROS_NodeHandle.subscribe(ROS_PID_ki_Sub);
    ROS_NodeHandle.subscribe(ROS_PID_kd_Sub);
  #endif

  // Set PWM frequencies (required for low pass filter PWM to DC)
    TCCR0B = (TCCR0B & 0xF8) | 0x03; //Need to be 3 to make micros(), millis() work.
    TCCR1B = (TCCR1B & 0xF8) | 0x01;
    TCCR2B = (TCCR2B & 0xF8) | 0x05;
    TCCR3B = (TCCR3B & 0xF8) | 0x01;
    TCCR4B = (TCCR4B & 0xF8) | 0x02;

  //make sure controller output is within PWM range
  SpeedPID.SetOutputLimits(MIN, MAX);

  //turn the PID on
  SpeedPID.SetMode(AUTOMATIC);

  analogWrite(Pin_Fans, 40);
  //digitalWrite(Pin_Stable12V, HIGH);

  //Serial.begin(9600);

  attachInterrupt(Pin_LeftWheelEncoder, LeftWheelEncoderTransition, RISING);
  attachInterrupt(Pin_RightWheelEncoder, RightWheelEncoderTransition, RISING);
}


//////////////////////////////////////////////
//                Loop                      //
//////////////////////////////////////////////

void loop()
{
  ReadSteeringEncoder();
  GetControllerInput();
  ReadWheelEncoders();
  LightControl();

  if(digitalRead(Pin_EmergencyStop) == HIGH && digitalRead(Pin_AutonomousModeIn))
  {
    RunAutonomousMode();
  }
  else
  {
    GetConsoleInput();
    ManualSteering();
    ManualThrust();
  }

  if(micros() - LastTime > 9680) //Önskat samplingsintervall 10ms. Det utanför if-satsen tar ca 640 microsekunder. Samplingsintervallet
blir nu ca 10ms +/- 0.32ms
  {
  RunCar();

  PublishROSData();

  ROS_NodeHandle.spinOnce();
  LastTime = micros();
  }
  digitalWrite(Pin_MotorController,HIGH);
}

void RunCar()
{
  if(ForwardMode)
    RunForwardMode();
  else if(ReverseMode)
    RunReverseMode();
  else
    RunIdle();

  analogWrite(Pin_ThrottleOut, OutputThrottle*PWM_RESOLUTION_12V);

  double Kp;
  double Ki;
  double Kd;
  double N;
  double h = 0.01;  //Sample period
  if(CurrentSpeed<0.3){   //Parameter control on the current speed [m/s]
    Kp = -29.4106;
    Ki = -12.1274;
    Kd = 3.7603;
    N = 7.512;
  }else if(CurrentSpeed>3){
    Kp = -39.9944;
    Ki = -60.3186;
    Kd = 0.90866;
    N = 35.8523;
  }else{
    Kp = -7.5771;
    Ki = -7.3581;
    Kd = 0.42957;
    N =  15.3967;
  }

  double Deviation = TargetSteeringAngle - CurrentSteeringAngle;    //PID controller starts here

  SteeringAdjustmentP = Kp*Deviation;

  if( abs(Ki*Deviation) < 700 ){                          //Anti-windup
    SteeringAdjustmentI = SteeringAdjustmentI +Ki*h*Deviation;
  }
 if( abs(SteeringAdjustmentD - 1/(N*h+1)*SteeringAdjustmentD+Kd*N/(N*h+1)*(Deviation - LastDeviation)) <400 ){ //To prevent
from spike in the D-part when the sensors are powered on
    SteeringAdjustmentD = 1/(N*h+1)*SteeringAdjustmentD+Kd*N/(N*h+1)*(Deviation - LastDeviation);
  }

  SteeringAdjustment = SteeringAdjustmentP + SteeringAdjustmentI + SteeringAdjustmentD;
  LastDeviation = Deviation;

  //SteeringAdjustment = (CurrentSteeringAngle -TargetSteeringAngle)*15;
  if(SteeringAdjustment > 0)
  {
    if(SteeringAdjustment > 255){
      SteeringAdjustment = 255;
    }
    digitalWrite(Pin_SteeringRightOut, HIGH);
    digitalWrite(Pin_SteeringLeftOut, LOW);
    analogWrite(Pin_SteeringOut, SteeringAdjustment);
  }
  else if(SteeringAdjustment < 0)
  {
    if(SteeringAdjustment < -255){
      SteeringAdjustment = -255;
    }
    digitalWrite(Pin_SteeringLeftOut, HIGH);
    digitalWrite(Pin_SteeringRightOut, LOW);
    analogWrite(Pin_SteeringOut, -SteeringAdjustment);
  }
  else
  {
    analogWrite(Pin_SteeringOut, 0);
  }
}


//////////////////////////////////////////////
//            Manual Drive              //
//////////////////////////////////////////////

void GetConsoleInput()
{
  if(digitalRead(Pin_NeutralModeIn) == LOW)
  {
    ForwardMode = digitalRead(Pin_ForwardModeIn);
    ReverseMode = digitalRead(Pin_ReverseModeIn);
  }
}

void ManualSteering() //TargetSteeringAngle gets its value depending on CurrentSpeed, ControllerSteeringAngle and and
SEERING_RATIO
{
 if(CurrentSpeed*3.6 < 10) //[CurrentSpeed]= m/s, the formla uses km/h.
  {
    TargetSteeringAngle = ControllerSteeringAngle*STEERING_RATIO;
  }
  else if(CurrentSpeed*3.6 > 110)
  {
    double p1 = 0.0002055;
   double p2 = -1.949*pow(10,-18);
   double p3 = 0.1824;
   double p4 = 4.643*pow(10,-16);

   TargetSteeringAngle = p1*pow(ControllerSteeringAngle*STEERING_RATIO,3) +
p2*pow(ControllerSteeringAngle*STEERING_RATIO,2) + p3*ControllerSteeringAngle*STEERING_RATIO + p4;
  }
  else
  {
  double p00 = 1.22*pow(10,-15);
  double p10 = 0.9872;
  double p01 = -2.126*pow(10,-16);
  double p20 = -1.826*pow(10,-17);
  double p11 = 0.003717;
  double p02 = 1.936*pow(10,-17);
  double p30 = -2.055*pow(10,-5);
  double p21 = -4.383*pow(10,-19);
  double p12 = -0.0002579;
  double p03 = -3.516*pow(10,-19);
  double p40 = 3.223*pow(10,-20);
  double p31 = 2.055*pow(10,-6);
  double p22 = 4.636*pow(10,-21);
  double p13 = 1.433*pow(10,-6);
  double p04 = 1.739*pow(10,-21);

  TargetSteeringAngle = p00 + p10*ControllerSteeringAngle*STEERING_RATIO + p01*CurrentSpeed*3.6 +
p20*pow(ControllerSteeringAngle*STEERING_RATIO,2) +
                        p11*ControllerSteeringAngle*STEERING_RATIO*CurrentSpeed*3.6 + p02*pow(CurrentSpeed*3.6,2) +
p30*pow(ControllerSteeringAngle*STEERING_RATIO,3) +
                        p21*pow(ControllerSteeringAngle*STEERING_RATIO,2)*CurrentSpeed*3.6  +
p12*ControllerSteeringAngle*STEERING_RATIO*pow(CurrentSpeed*3.6,2) +
                        p03*pow(CurrentSpeed*3.6,3) + p40*pow(ControllerSteeringAngle*STEERING_RATIO,4) +
p31*pow(ControllerSteeringAngle*STEERING_RATIO,3)*CurrentSpeed*3.6 +
                        p22*pow(ControllerSteeringAngle*STEERING_RATIO,2)*pow(CurrentSpeed*3.6,2) +
p13*ControllerSteeringAngle*STEERING_RATIO*pow(CurrentSpeed*3.6,3) +
                        p04*pow(CurrentSpeed*3.6,4);
  }
}

void ManualThrust()
{
  if(ControllerThrottle < 0.05f || !Safe)   // The safe variable is set to false by the safety features when needed.
  {
    OutputThrottle = 0.0f;
    OutputBrake = 1.0f;
    if(CurrentSpeed < 0.1){
      Safe = true;
    }
  }
  else if(ControllerThrottle > 0.1f)
  {
    OutputBrake = 0.0f;
    OutputThrottle = 0.30f + ControllerThrottle*0.5;
  }
  else
  {
    OutputBrake = 0.0f;
    OutputThrottle = 0.0f;
  }
```

```
}


//////////////////////////////////////////
//            Autonomy              //
//////////////////////////////////////////

void RunAutonomousMode()
{
  if(Autonomy_SpeedController)
  {
    PID_Setpoint = TargetSpeed;
    PID_Input = CurrentSpeed;
    SpeedPID.Compute();
  }
  if(Autonomy_ManualSteering)
    ManualSteering();
  if(Autonomy_ManualThrust)
    ManualThrust();

//  if(ControllerThrottle > 0.5f)
//  {
//    if(TargetBrake == 0.0f)
//    {
//      KeepDistance(3.0f, 1.0f, 1.0f);
//
//      PID_Setpoint = TargetSpeed;
//      PID_Input = CurrentSpeed;
//      SpeedPID.Compute();
//
//      OutputThrottle = PID_Output;
//      OutputBrake = 0.0f;
//    }
//    else
//    {
//      OutputThrottle = 0.0f;
//      OutputBrake = 0.9f;
//    }
//  }
//  else
//  {
//    OutputThrottle = 0.0f;
//    OutputBrake = 0.9f;
//  }
}

void KeepDistance(float DistanceToKeep, float ApproachSpeed, float DistanceTolerance)
{
  if(LidarDistance < 0.5f)
    LidarDistance = 100.0f;

  if(LidarDistance < 30.0f)
  {
    if(LidarDistance < DistanceToKeep-DistanceTolerance*2)
      TargetSpeed = LidarSpeed + CurrentSpeed - ApproachSpeed;
    else if(LidarDistance < DistanceToKeep-DistanceTolerance)
      TargetSpeed = LidarSpeed + CurrentSpeed - ApproachSpeed*(DistanceToKeep-LidarDistance);
    else if(LidarDistance > DistanceToKeep+DistanceTolerance*2)
      TargetSpeed = LidarSpeed + CurrentSpeed + ApproachSpeed;
    else if(LidarDistance > DistanceToKeep+DistanceTolerance)
      TargetSpeed = LidarSpeed + CurrentSpeed + ApproachSpeed*(LidarDistance-DistanceToKeep);
    else
      TargetSpeed = LidarSpeed + CurrentSpeed;
  }
  else if(TargetSpeed < 2.0f)
    TargetSpeed = 2.0f;
}


//////////////////////////////////////////
//            Steering              //
//////////////////////////////////////////

void GetControllerInput()
{
  SmoothSteeringInputOld = SmoothSteeringInput;
  SmoothSteeringInput = SmoothSteeringInput*0.9f + 0.1f*analogRead(Pin_SteeringIn);
  SmoothThrottleInput = SmoothThrottleInput*0.9f + 0.1f*analogRead(Pin_ThrottleIn);

  if( (SmoothSteeringInput < CONTROLLER_STEERING_MAX - 10) || (SmoothSteeringInput > CONTROLLER_STEERING_MIN +
10)) //Safety feature (Min/Max stands for angle not input value)
  {
    Safe = false;
  }
  if( abs(SmoothSteeringInput - SmoothSteeringInputOld) > 5) //Safety feature.
  {
      Safe = false;
  }
  ControllerSteeringAngle = -(((SmoothSteeringInput-CONTROLLER_STEERING_MIN)/(CONTROLLER_STEERING_MAX-
CONTROLLER_STEERING_MIN))*2.0f-1.0f)*CONTROLLER_MAX_ANGLE;  //Rescaling from digital input to degrees.
  ControllerThrottle = ((SmoothThrottleInput-CONTROLLER_THROTTLE_MIN)/(CONTROLLER_THROTTLE_MAX-
CONTROLLER_THROTTLE_MIN)); //Rescaling from digital input to 0-1.
}

void ReadSteeringEncoder()
{
  int SteeringEncoderVal = analogRead(Pin_SteeringEncoder);

  CurrentSteeringAngle = ((SteeringEncoderVal-WHEEL_STEERING_MIN)/(WHEEL_STEERING_MAX-
WHEEL_STEERING_MIN)*2.0f - 1.0f)*WHEEL_MAX_ANGLE; //Rescaling from digital input to degrees.
}


//////////////////////////////////////////////
//            Control Panel             //
//////////////////////////////////////////////

void LightControl()
{
  if (digitalRead(Pin_LightSwitchIn) == HIGH)
  {
    digitalWrite(Pin_LightLeftOut, HIGH);
    digitalWrite(Pin_LightRightOut, HIGH);
    digitalWrite(Pin_LightBackOut, HIGH);
  }
  else
  {
    digitalWrite(Pin_LightLeftOut, LOW);
    digitalWrite(Pin_LightRightOut, LOW);
    digitalWrite(Pin_LightBackOut, LOW);
  }
}

void RunForwardMode()
{
  digitalWrite(Pin_ForwardModeOut, HIGH);
  digitalWrite(Pin_ReverseModeOut, LOW);
}

void RunReverseMode()
{
  digitalWrite(Pin_ForwardModeOut, LOW);
  digitalWrite(Pin_ReverseModeOut, HIGH);
}

void RunIdle()
{
  digitalWrite(Pin_ForwardModeOut, LOW);
  digitalWrite(Pin_ReverseModeOut, LOW);
}

//////////////////////////////////////////////
//          Speed measurement           //
//////////////////////////////////////////////

void ReadWheelEncoders()
{
  unsigned long CurrentTime = micros();

  if (CurrentTime - LastTimeWheelEncoder >= WHEEL_ENCODER_RATE)
  {
    unsigned long ElapsedTime = CurrentTime - LastTimeWheelEncoder;
    detachInterrupt(Pin_LeftWheelEncoder);//Disable interrupt when calculating
    detachInterrupt(Pin_RightWheelEncoder);//Disable interrupt when calculating

      LastTimeRightWheelEncoderTickTime = LastTimeRightWheelEncoderTickTime + ElapsedTime;

      LeftWheelSpeed = 0.9f*LeftWheelSpeed + 
0.1f*(WHEEL_DIAMETER*PI/WHEEL_ENCODER_TICKS)/LastTimeLeftWheelEncoderTickTime*1000000;
      RightWheelSpeed = 0.9f*RightWheelSpeed + 
0.1f*(WHEEL_DIAMETER*PI/WHEEL_ENCODER_TICKS)/LastTimeRightWheelEncoderTickTime*1000000;
      if( abs((LeftWheelSpeed + RightWheelSpeed)/2 - CurrentSpeed) < 0.2)
      {
        CurrentSpeed = (LeftWheelSpeed + RightWheelSpeed)/2;
      }

      DistanceTravelled = DistanceTravelled + (LeftWheelEncoderCounter+RightWheelEncoderCounter)/2 * 
(WHEEL_DIAMETER*PI/WHEEL_ENCODER_TICKS);

      LeftWheelEncoderCounter = 0;
      RightWheelEncoderCounter = 0;
      LastTimeWheelEncoder = micros();

      LastTimeLeftWheelEncoderTickTime = LastTimeLeftWheelEncoderTickTime + ElapsedTime;
      LastTimeRightWheelEncoderTickTime = LastTimeRightWheelEncoderTickTime + ElapsedTime;

    attachInterrupt(Pin_LeftWheelEncoder, LeftWheelEncoderTransition, RISING); //enable interrupt
    attachInterrupt(Pin_RightWheelEncoder, RightWheelEncoderTransition, RISING); //enable interrupt
  }
}

void LeftWheelEncoderTransition()
{
  LeftWheelEncoderCounter++;
  unsigned long CurrentTime = micros();
  LastTimeLeftWheelEncoderTickTime = CurrentTime - LastTimeLeftWheelEncoderTick;
  LastTimeLeftWheelEncoderTick = CurrentTime;
}

void RightWheelEncoderTransition()
{
  RightWheelEncoderCounter++;
  unsigned long CurrentTime = micros();
  LastTimeRightWheelEncoderTickTime = CurrentTime - LastTimeRightWheelEncoderTick;
  LastTimeRightWheelEncoderTick = CurrentTime;
}
////////////////////////////////////////////
//            ROS Publisher             //
////////////////////////////////////////////

unsigned long LastTimeROSPublish = 0;
void PublishROSData()
{
    current_steering_msg.data = CurrentSteeringAngle;
    ROS_Current_Steering_Pub.publish(&current_steering_msg);

    target_steering_msg.data = TargetSteeringAngle;
    ROS_Target_Steering_Pub.publish(&target_steering_msg);

    targetspeed_msg.data = TargetSpeed;
    ROS_TargetSpeed_Pub.publish(&targetspeed_msg);

    throttle_msg.data = OutputThrottle;
    ROS_Throttle_Pub.publish(&throttle_msg);

    steering_adjustment_msg.data = SteeringAdjustment;
    ROS_Steering_Adjustment_Pub.publish(&steering_adjustment_msg);

    distance_travelled_msg.data = DistanceTravelled;
    ROS_DistanceTravelled_Pub.publish(&distance_travelled_msg);

    speed_msg.data = CurrentSpeed;
    ROS_Speed_Pub.publish(&speed_msg);
}
```

Appendix 4: The speed dependency implemented to the VR-environment

```
//------------------------------------------------------------------------------------------------
// Edy's Vehicle Physics
// (c) Angel Garcia "Edy" - Oviedo, Spain
// http://www.edy.es
//------------------------------------------------------------------------------------------------

using UnityEngine;
using UnityEngine.UI;

namespace EVP
{

    public class VehicleStandardInput : MonoBehaviour
    {
        //Define vehicle parameters

        public float cutoff_speed = 40.0f; //Speed in kph where steering output is limited according to speed
        public float min_control_speed = 0.7f; //Steering output at max speed

        //Steering curve parameters
        public float deadzone = 0.1f;
        public float param_lin = 30.0f;
        public float param_square = 10.0f;

        //Acceleration&deceleration parameters
        public float acc_deadzone = 0.04f;
        public float acc_param_pos_lin = 10.0f;
        public float acc_param_pos_const = 0.1f;
        public float acc_param_neg_lin = 10.0f;
        public float acc_param_neg_const = 0.1f;
        public float max_reverse = 0.2f;
        public float max_brake = -0.5f;
        public float brake_exp = 2.0f;

        //Axle offsets
        public float SteeringOffset = -0.615f;
        public float SteeringFactor = 3.0f;
        public float AccelerationOffset = 0.0f;
        public float AccelerationFactor = 3.0f;

        float RawSteeringInput = 0.0f;
        float RawAccInput = 0.0f;

        public VehicleController target;


        bool continuousForwardAndReverse = true; //Drivetrain active
        string steerAxis = "Vertical"; //Define steerAxis as Horizontal
        string throttleAndBrakeAxis = "Horizontal"; //Define acceleration axis as Vertical
        string handbrakeAxis = "Handbrake"; //Define handbrake axis as handbrake
        string resetVehicleKey = "Respawn"; //Define resetkey as Respawn

        bool m_doReset = false; //Define Respawn as inactive until input is given

        //Transform StartTransform;


        void Start()
        {
            //StartTransform = transform;
        }

        void OnEnable()
        {
            // Cache vehicle

            if (target == null)
                target = GetComponent<VehicleController>();
        }


        void Update()
    //Update vehicle state
    {
       if (target == null) return;

       if (Input.GetButtonDown(resetVehicleKey)) m_doReset = true; //If respawn input is given, reset vehicle
    }


    void FixedUpdate()
    {
       if (target == null) return;

       float kph_speed = Mathf.Abs(target.speed) * 3.6f; // Convert to kph

       RawSteeringInput = RawSteeringInput * 0.95f + ((Mathf.Clamp(Input.GetAxis(steerAxis) * SteeringFactor + SteeringOffset, -
1.0f, 1.0f))) * 0.05f;//limit steering input from controller, offset and scale
       float steerInput = -RawSteeringInput;
       steerInput = Mathf.Clamp(steerInput, -1.0f, 1.0f);      //Limit steering input after offsetting and scaling between -1 and +1

       RawAccInput = RawAccInput * 0.95f + Mathf.Clamp(Input.GetAxis(throttleAndBrakeAxis) * AccelerationFactor +
AccelerationOffset, -1.0f, 1.0f) * 0.05f; //limit acceleration between -1 and +1
       float accInput = RawAccInput;

       target.steerInput = ManualSteering(Mathf.Abs(target.speed), steerInput*45.0f)/45.0f;
       target.throttleInput = ManualThrust(Mathf.Abs(target.speed), accInput);
       target.brakeInput = ManualBrake(Mathf.Abs(target.speed), accInput);
       target.handbrakeInput = 0.0f;



       //    float smooth = 0.1f; //defines the smoothing of the curve

       //    //Steering coefficient follows arctan curve. Good or not?
       //    float steering_speed_scale = -1 * Mathf.Atan(smooth * (kph_speed - cutoff_speed))
       //                                 / Mathf.PI * (1 - min_control_speed)
       //                                 + 1 - (1 - min_control_speed) / 2; //Adjust steering output according to speed.


       //    float sign = Mathf.Sign(steerInput);                    //Define if steering is left or right
       //    steerInput = Mathf.Abs(steerInput);                         //Absolute value of steering
       //    float steerInputComp = 1 / (1 / (param_square * (steerInput - deadzone) * (steerInput - deadzone)
       //              + 1 / (param_lin * (steerInput - deadzone)))
       //              * (1 / (param_square * (1 - deadzone) * (1 - deadzone)) + 1 / (param_lin * (1 - deadzone))); //Adjust steering for
progressive steering output

       //    //The resulting curves are not really adjustable. Parameters barely matter
       //    if (steerInput < deadzone)
       //        steerInputComp = 0.0f; //If within deadzone, set to zero

       //    steerInput = steerInputComp * sign * steering_speed_scale; //Calculate steering from adjusted steering, direction and speed
compensation


       //    //float handbrakeInput = Mathf.Clamp01(Input.GetAxis(handbrakeAxis));

       //    float handbrakeInput = 0.0f; //No handbrake

       //    float acc_sign = Mathf.Sign(accInput); //Sign of throttle axis
       //    accInput = Mathf.Abs(accInput);

       //    float forwardInput = 0.0f;
       //    if (acc_sign == 1.0f && accInput > acc_deadzone)
       //        forwardInput = 1 / (1 / ((accInput - acc_deadzone) * (accInput - acc_deadzone))
       //                    + 1 / (acc_param_pos_lin * (accInput - acc_deadzone))
       //                    + 1 / acc_param_pos_const)
       //                    * (1 / ((1.0f - acc_deadzone) * (1.0f - acc_deadzone))
       //                    + 1 / (acc_param_pos_lin * (1.0f - acc_deadzone))
       //                    + 1 / acc_param_pos_const);

       //    float reverseInput = 0.0f;
       //    if (acc_sign == -1.0f && accInput > acc_deadzone)
       //        reverseInput = 1 / (1 / ((accInput - acc_deadzone) * (accInput - acc_deadzone))
       //                    + 1 / (acc_param_neg_lin * (accInput - acc_deadzone))
       //                    + 1 / acc_param_neg_const)
       //                    * (1 / ((1.0f - acc_deadzone) * (1.0f - acc_deadzone))
       //                    + 1 / (acc_param_neg_lin * (1.0f - acc_deadzone))
//                        + 1 / acc_param_neg_const);


////Debug.Log(RawSteeringInput);
////Debug.Log(RawAccInput);
////Debug.Log(accInput);
////Debug.Log(forwardInput);
////Debug.Log(reverseInput);

//float throttleInput = 0.0f;
//float brakeInput = 0.0f;

//if (continuousForwardAndReverse)
//{
//    float minSpeed = 0.1f;
//    float minInput = 0.1f;

//    throttleInput = forwardInput;

//    if (reverseInput > 0.8f && target.speed < 0.1f)
//        throttleInput = -reverseInput * max_reverse;
//    else
//        brakeInput = reverseInput;
//}
//else
//{
//    bool reverse = Input.GetKey(KeyCode.LeftControl) || Input.GetKey(KeyCode.RightControl);

//    if (!reverse)
//    {
//        throttleInput = forwardInput;
//        brakeInput = reverseInput;
//    }
//    else
//    {
//        throttleInput = -reverseInput;
//        brakeInput = 0;
//    }
//}

//// Apply input to
//target.steerInput = steerInput;
//target.throttleInput = throttleInput;
//target.brakeInput = brakeInput;
//target.handbrakeInput = handbrakeInput;

//if (handbrakeInput > 0.0f)
//    target.throttleInput = 0;

// Do a vehicle reset
if (m_doReset)
{
    if (FindObjectOfType<RaceTimer>().GetComponent<Text>().enabled)
    {
        FindObjectOfType<RaceTimer>().GetComponent<Text>().enabled = false;
        FindObjectOfType<CheckpointIndicator>().GetComponent<Text>().enabled = false;

        foreach (Checkpoint cp in FindObjectsOfType<Checkpoint>())
        {
            cp.GetComponentInParent<MeshRenderer>().enabled = false;
            cp.Taken = false;
        }

        foreach (Goal cp in FindObjectsOfType<Goal>())
        {
            cp.GetComponentInParent<MeshRenderer>().enabled = false;
            cp.Taken = false;
        }

        //target.ResetVehicle();

        FindObjectOfType<RaceTimer>().ResetTimer();
        //FindObjectOfType<RaceTimer>().StartTimer();

        FindObjectOfType<CheckpointCounter>().NumberOfCPsTaken = 0;
    }
```

```
            else
            {
               FindObjectOfType<RaceTimer>().GetComponent<Text>().enabled = true;
               FindObjectOfType<CheckpointIndicator>().GetComponent<Text>().enabled = true;

               foreach (Checkpoint cp in FindObjectsOfType<Checkpoint>())
               {
                  cp.GetComponentInParent<MeshRenderer>().enabled = true;
                  cp.Taken = false;
               }

               foreach (Goal cp in FindObjectsOfType<Goal>())
               {
                  cp.GetComponentInParent<MeshRenderer>().enabled = true;
                  cp.Taken = false;
               }

               target.ResetVehicle();

               FindObjectOfType<RaceTimer>().ResetTimer();
               FindObjectOfType<RaceTimer>().StartTimer();

               FindObjectOfType<CheckpointCounter>().NumberOfCPsTaken = 0;
            }

            m_doReset = false;
         }
      }

      float ManualSteering(float CurrentSpeed, float ControllerSteeringAngle) //TargetSteeringAngle gets its value depending on
CurrentSpeed and SEERING_RATIO
      {
         float STEERING_RATIO = 26.65f / 45.0f;

         float TargetSteeringAngle = ControllerSteeringAngle;

         if (CurrentSpeed * 3.6 < 10) //[CurrentSpeed]= m/s, the formla uses km/h.
         {
            TargetSteeringAngle = ControllerSteeringAngle * STEERING_RATIO;
         }
         else if (CurrentSpeed * 3.6 > 110)
         {
            double p1 = 0.0002055;
            double p2 = -1.949 * Mathf.Pow(10, -18);
            double p3 = 0.1824;
            double p4 = 4.643 * Mathf.Pow(10, -16);

            TargetSteeringAngle = (float)(p1 * Mathf.Pow(ControllerSteeringAngle * STEERING_RATIO, 3) + p2 *
Mathf.Pow(ControllerSteeringAngle * STEERING_RATIO, 2) + p3 * ControllerSteeringAngle * STEERING_RATIO + p4);
         }
         else
         {
            double p00 = 1.22 * Mathf.Pow(10, -15);
            double p10 = 0.9872;
            double p01 = -2.126 * Mathf.Pow(10, -16);
            double p20 = -1.826 * Mathf.Pow(10, -17);
            double p11 = 0.003717;
            double p02 = 1.936 * Mathf.Pow(10, -17);
            double p30 = -2.055 * Mathf.Pow(10, -5);
            double p21 = -4.383 * Mathf.Pow(10, -19);
            double p12 = -0.0002579;
            double p03 = -3.516 * Mathf.Pow(10, -19);
            double p40 = 3.223 * Mathf.Pow(10, -20);
            double p31 = 2.055 * Mathf.Pow(10, -6);
            double p22 = 4.636 * Mathf.Pow(10, -21);
            double p13 = 1.433 * Mathf.Pow(10, -6);
            double p04 = 1.739 * Mathf.Pow(10, -21);

            TargetSteeringAngle = (float)(p00 + p10 * ControllerSteeringAngle * STEERING_RATIO + p01 * CurrentSpeed * 3.6 + p20 *
Mathf.Pow(ControllerSteeringAngle * STEERING_RATIO, 2) +
                           p11 * ControllerSteeringAngle * STEERING_RATIO * CurrentSpeed * 3.6 + p02 * Mathf.Pow(CurrentSpeed *
3.6f, 2) + p30 * Mathf.Pow(ControllerSteeringAngle * STEERING_RATIO, 3) +
                           p21 * Mathf.Pow(ControllerSteeringAngle * STEERING_RATIO, 2) * CurrentSpeed * 3.6 + p12 *
ControllerSteeringAngle * STEERING_RATIO * Mathf.Pow(CurrentSpeed * 3.6f, 2) +
                           p03 * Mathf.Pow(CurrentSpeed * 3.6f, 3) + p40 * Mathf.Pow(ControllerSteeringAngle * STEERING_RATIO,
4) + p31 * Mathf.Pow(ControllerSteeringAngle * STEERING_RATIO, 3) * CurrentSpeed * 3.6 +
                              p22 * Mathf.Pow(ControllerSteeringAngle * STEERING_RATIO, 2) * Mathf.Pow(CurrentSpeed * 3.6f, 2) +
p13 * ControllerSteeringAngle * STEERING_RATIO * Mathf.Pow(CurrentSpeed * 3.6f, 3) +
                              p04 * Mathf.Pow(CurrentSpeed * 3.6f, 4));



            }

            return TargetSteeringAngle;
        }

        float ManualThrust(float CurrentSpeed, float ControllerThrottle, bool Safe = true)
        {
            float OutputThrottle = 0.0f;

            if (ControllerThrottle < 0.05f || !Safe)
            {
                OutputThrottle = 0.0f;
                if (CurrentSpeed < 0.1)
                    Safe = true;
            }
            else if (ControllerThrottle > 0.1f)
                OutputThrottle = 0.30f + ControllerThrottle * 0.5f;
            else
                OutputThrottle = 0.0f;

            return OutputThrottle;
        }

        float ManualBrake(float CurrentSpeed, float ControllerThrottle, bool Safe = true)
        {
            float OutputBrake = 0.0f;

            if (ControllerThrottle < 0.05f || !Safe)
            {
                OutputBrake = 1.0f;
                if (CurrentSpeed < 0.1)
                    Safe = true;
            }
            else if (ControllerThrottle > 0.1f)
                OutputBrake = 0.0f;
            else
                OutputBrake = 0.0f;

            return OutputBrake;
        }
    }
}
```