Norwegian University
of Life Sciences

Master's Thesis 2020 30 ECTS
Faculty of Science and Technology

The Thorvald Agricultural Robot: Using Sensor Data to Navigate in Tall Grass

Simon Forsetlund
Mechanical Engineering

Preface

The submission of this thesis concludes my years of studies at The Norwegian University of Life Sciences (NMBU), in fulfilment of the requirements of the master's program machine, process- and product development. Although challenging, the last five years have been very rewarding, and I will take the experiences and knowledge with me into the next chapter of my professional life. The subject of this master's thesis reflects the years spent at the university, where I had the chance to dive into the world of autonomous robots and general automation. My interest and curiosity in this field was further sparked by the presence of the Robotics and Control Group at NMBU, where master's- and PhD-students have been developing agricultural robots for several years. I have also previously studied agronomy, and I am passionate about contributing to finding sustainable farming techniques.

This thesis has been quite challenging as I had never before worked with the Robot Operating System (ROS), nor coded in C++, and a lot of time was therefore spent becoming proficient in the robotics software framework and the programming language. However, for the very same reason, this thesis has been extremely rewarding and has allowed me to learn more about the exciting subject of robotics programming.

Ås, 01.06.2020

_____

Simon Forsetlund

Acknowledgments

I would be remiss if I did not extend my gratitude to the people around me, taking time out of their busy schedules to make this master's thesis possible. I would like to thank my main thesis advisor Lars Grimstad for his time and his many valuable inputs. His experience and help have guided me through this process. I would also like to thank my thesis co-advisor Jose Carlos Mayoral Banos for graciously allocating time from his own PhD thesis work. Interacting with fellow NMBU master-students writing theses in related subjects has also been very helpful, and I am thankful that I had someone to bounce ideas off.

While working on the thesis I also had the opportunity to visit the Pontifical Catholic University of Rio de Janeiro, for a workshop in vision-based control with applications to robotic systems. I am grateful for the warm welcome I received from the Brazilian hosts, and I hope this collaborating project between our educational institutions will continue in the future.

Lastly, I would like to thank my father, mother and sister for their continuous support, and my girlfriend for her patience with me as I spent many late nights working.

Abstract

This master's thesis presents algorithms and new methods for autonomous navigation of a grass cutting robot. For the purpose of the current research, the selected platform was the Thorvald II mobile grass cutting robot. Thorvald II is an agricultural robot capable of working in many different environments such as greenhouses and open fields. As the modular robotic system is developed to be flexible, the robot can perform many different operations and may be configured in different shapes and forms. However, this thesis studies the specific grass cutting robot configuration.

For outdoor applications, the use of Global Navigation Satellite System (GNSS) based navigation has become a trend due to its ability to deliver high precision positioning. However, many fields lie in undulating terrain. This makes GNSS based navigation unreliable and error prone, resulting in ineffective and wasteful cutting of grass. Optimizing the robot's ability to efficiently navigate and correct its course in real time, saves the farmer money and time which ultimately promotes the use of the Thorvald robotic system.

In this thesis, a method is developed to allow the robot to identify the edge separating the cut and uncut grass. By detecting this line, the robot and its cutting tool can autonomously manoeuvre to precisely follow the line as the robot makes its way through the field. By applying a Multiple-criteria decision-making (MCDM) technique, the use of a Light Identification Detection and Ranging (LIDAR) sensor is suggested to generate the necessary point cloud data. Algorithms are developed to manipulate the point cloud and estimate the line, using the Random Sample Consensus (RANSAC) robust estimation model. The Robotic Operating System (ROS) framework and its key features are discussed, and finally used to implement the developed algorithms. A motion model is defined, and velocity control is implemented using a proportionate (P) controller, and a proportionate-derivate (PD) controller.

Testing on captured field data and simulations show that the robot is successfully able to detect the line in the grass. The algorithm's robustness is tested and performs well, even in environments with significant noise and obstacles present. Testing of the implemented velocity control have produced promising results, as the robot is able to navigate along the line autonomously.

Sammendrag

I denne masteroppgaven presenteres algoritmer og nye metoder for autonom navigasjon av en grasklippende robot. Den mobile grasklipper-roboten Thorvald II ble valgt som plattform for denne oppgavens formål. Thorvald II er en jordbruksrobot som kan arbeide i mange forskjellige miljøer, som drivhus og åpne jorder. Ettersom det modulære robot-systemet er utviklet for å være fleksibelt, kan roboten utføre mange forskjellige operasjoner og kan konfigureres i forskjellige former og modeller. I denne oppgaven blir imidlertid den grasklippende robotkonfigurasjonen studert.

Ofte blir Global Navigation Satellite System (GNSS) basert navigasjon brukt for applikasjoner som navigerer utendørs, på grunn av den høye presisjonen systemet leverer. Mange jorder ligger imidlertid i bølget terreng, noe som gjør GNSS basert navigering upålitelig og utsatt for feil, og som resulterer i ineffektiv og ikke-produktiv kutting av gress. Optimalisering av robotens evne til effektivt å navigere og korrigere kursen i sanntid, sparer bonden for penger og tid, og vil fremme bruken av robotsystemet Thorvald.

I denne masteroppgaven utvikles en metode som lar roboten identifisere graskanten som skiller det klippede og uklippede graset. Ved å oppdage denne linjen kan roboten og dens klippeverktøy autonomt manøvrere for å følge denne linjen nøyaktig, mens roboten beveger seg over jordet. Ved bruk av multikriteria-analyse og seleksjonsteknikk (MCDM), foreslås bruk av en Light Identification Detection and Ranging (LIDAR) sensor for å danne den nødvendige punktskydataen. Algoritmer utvikles for å manipulere punktskyen og estimere linjen ved bruk av den robuste estimeringsmodellen Random Sample Consensus (RANSAC). Rammeverket Robotic Operating System (ROS) og dens nøkkelfunksjoner blir diskutert, og til slutt brukt i implementeringen av de utviklede algoritmene. Et bevegelsessystem blir definert, og hastighetskontroll implementeres ved hjelp av en proporsjonal (P) kontroller og en proporsjonal-derivat (PD) kontroller.

Testing av feltdata og simuleringer viser at roboten er i stand til å oppdage linjen i graset. Algoritmens robusthet er testet og klarer seg bra selv i miljøer med betydelig støy og hindringer til stede. Testing av hastighetskontrollen har vist tilfredsstillende resultater, da roboten er i stand til å autonomt navigere langs linjen.

Table of Contents

Preface i

Acknowledgments iii

Abstract v

Sammendrag vii

List of Figures xv

List of Tables xxi

List of Abbreviations xxiii

1. Introduction 1

1.1 Background 1

1.2 Motivation 2

1.3 The Agricultural Robot Thorvald 3

1.3.1 Thorvald I 3

1.3.2 Thorvald II 4

1.3.3 GrassRobotics: The Cutting-Edge Grass Cutter 5

1.4 Problem Statement 6

1.4.1 Thesis Main Objective 7

1.4.2 Thesis Sub Objectives 7

2. Theory and Technology 9

2.1 GNSS RTK 9

2.2 Topological Maps 9

2.3 Navigation and Positioning 11

2.4 Thermographic Camera 13

2.5 RGB-D Camera 14

2.6 LIDAR 15
2.7 Point Clouds 17
2.8 Estimation Models 18
2.8.1 Least Squares Regression 18
2.8.2 RANSAC 20
2.9 Multi Criteria Selection 22
3. Control Systems 25
3.1 Laplace Transform 25
3.2 Transfer Functions 26
3.3 Proportional-Integral-Derivative Control 28
4. Robot Programming 31
4.1 Introduction to Robot Programming 31
4.2 Robot Operating System (ROS) 31
4.3 Brief History of ROS 32
4.4 The Benefits of Using ROS 32
4.5 ROS Architecture and Key Features 33
4.5.1 Reference Frames and the ROS Transform Library 35
4.5.2 RViz: Data Visualizing Software 36
4.5.3 Gazebo: Robot Simulation Software 37
4.6 Point Cloud Library 37
5. Sensor and Software Selection 39
5.1 Application of Sensors for Grass Line Detection 39
5.2 Multicriteria Selection of Sensor Technology 41
5.3 Software Selection 44

5.4 Programming Language Selection 44
6. Control System Design 47
6.1 Reference Systems of the Thorvald Robot 47
6.2 Error Estimation 48
6.3 Motion Modelling and Velocity Control 50
6.3.1 Angular Velocity Control 52
6.3.2 Linear Velocity Control 54
7. Perception and Navigation Strategy 57
7.1 Package Architecture 57
7.2 Perception and Navigation Algorithm 58
7.2.1 Perception Algorithm in Node 1 59
7.2.2 Navigation Algorithm in Node 2 65
8. Testing and Verification 71
8.1 Simulation Testing 71
8.1.1 Line Detection with Increasing Lateral Displacement 72
8.1.2 Autonomous Navigation Test 73
8.2 Testing on Captured Field Data 74
8.2.1 Non-stationary Left-Sided Line Detection Uphill 75
8.2.2 Non-stationary Right-Sided Line Detection Downhill 77
8.2.3 Stationary Line Detection Test with Moving Objects 78
9. Results 81
9.1 Simulation Testing 81
9.1.1 Line Detection with Increasing Lateral Distance 81
9.1.2 Autonomous Navigation Test 83

9.2 Testing on Captured Field Data 85
9.2.1 Non-Stationary Left-Sided Line Detection Uphill 85
9.2.2 Non-Stationary Right-Sided Line Detection Downhill 87
9.2.3 Stationary Line Detection Test with Moving Objects 89
10. Discussion 91
10.1 The ROS framework 91
10.2 Simulation Testing 92
10.2.1 Line Detection with Increasing Lateral Distance 92
10.2.2 Autonomous Navigation Test 93
10.3 Testing on Captured Field Data 94
10.3.1 Non-Stationary Line Detection 94
10.3.2 Stationary Line Detection Test with Moving Objects 94
10.4 Assumptions and Simplifications 94
10.5 Achievement of Objectives 95
11. Conclusion and Future Work 97
11.1 Conclusion 97
11.2 Future Work 98
11.2.1 Sensor Placement 98
11.2.2 Further Testing 98
11.2.3 Tuning of Controller 98
11.2.4 Algorithm Development 99
References 101
Appendices 105
C++ Code 106

A.1 ROS Nodes 106
A.1.1 Perception Node 106
A.1.2 Navigation Node 106
A.2 PointCloudSensor Class 113
A.3 Parameter Server 120
A.4 Launch File 121

List of Figures

Figure 1.1: Thorvald as presented in 2016 (9). 4

Figure 1.2: Some of the different configurations of Thorvald II modules (11). 4

Figure 1.3: The grass cutting Thorvald configuration (12). 5

Figure 1.4: The robot as it cuts grass in the field, optimally aligned and following the separation line represented by the red arrow. 7

Figure 2.1: A directed graph with four vertices connected by three edges (specified direction). 10

Figure 2.2: Illustration of the topological map, demonstrating the placement of nodes around a field, and the edges connecting them. 11

Figure 2.3: Illustration of a field with a 45-degree slope, visualised in 2D. 12

Figure 2.4: a) The TIBA Robot in the field; b) the RGB pictures it took in the sugarcane tunnels, compared to the correlating IR-images (17). 13

Figure 2.5: a) A coloured image showing the RGB component of the image; b) The depth component of the image produced by the RGB-D camera. 15

Figure 2.6: The RGB-D camera RealSense D415 from Intel (20). 15

Figure 2.7: The VLP-16 has a 30 degree field of view, it is mounted on Thorvald and angled according to the range needed (the illustration is not to scale). 16

Figure 2.8: An example of a point cloud of a house (24). 17

Figure 2.9: The coordinates of points in a point cloud, arranged in a matrix form. 18

Figure 3.1: A block diagram of an LTI system, the input is a Dirac delta function which operates through the system's impulse response and produces the output function. 26

Figure 3.2: A block diagram of an input signal operating on a transfer function to produce an output signal. 27

Figure 3.3: A block diagram of a closed-loop system. 28

Figure 3.4: A block diagram of a closed-loop system including a feedback element. 28

Figure 4.1: Block diagram of the ROS environment's inter process communication (34). 34

Figure 4.2: The Thorvald grass cutter (not equipped with its cutting tool), shown with all its reference frames. 36

Figure 5.1: a) The RGB coloured image depicting a field of grass, with a section of cut grass in the top-right corner; b) The depth component of the picture. 40

Figure 6.1: Illustration of Thorvald with the global (Ef) and local (Et) reference frames defined. 48

Figure 6.2: Illustration of the error estimation from the line. 50

Figure 6.3: A block diagram of the robot's angular velocity controlled by a PD-controller. 53

Figure 6.4: An illustration of the intended path generation, where a setpoint and line is generated to which the robot navigates until it reaches a set distance to the setpoint, at which time another setpoint is generated, and so on. 54

Figure 6.5: A block diagram of Thorvald's linear velocity controlled by a P-controller. 55

Figure 7.1: The package architecture. 58

Figure 7.2: Flowchart of the suggested algorithm, and how it is divided in two nodes. 59

Figure 7.3: The critical zone where Thorvald will detect the grass line (the x_t, y_t, z_t axis are shown as red, green, and blue respectively). 60

Figure 7.4: a) A possible noisy environment b) The same environment represented as a point cloud, the extracted plane (+/- 15 cm) that is perpendicular to the z-axis has been coloured red, points that do not lie on the plane are coloured black, points that do lie on the plane are enlarged and coloured dark red. 61

Figure 7.5: a) A point cloud representing the plane of the uncut grass in the critical zone; b) The same point cloud divided into 25 segments with a width of 10 cm. 62

Figure 7.6: The green points represent the points having the highest value on the local yt-axis in their respective segment. 63

Figure 7.7: An example of a line estimated by a RANSAC model (threshold value in dashed lines), using a dataset with 22 inliers (coloured green) and three outliers (coloured yellow). 64

Figure 7.8: The points P1 and P2 are defined a set distance from P0. 65

Figure 7.9:As the orientation of Thorvald changes in the global reference system, the position of the tip of the cutter is rotated about the base_link frame. 67

Figure 8.1: Thorvald as visualised in RViz. 72

Figure 8.2: The simulated environment, showing how the robot is placed in relation to the "grass", and moved laterally in relation to the line. 73

Figure 8.3: The simulated field in the autonomous navigation test, seen from a side-view on the left, and top-view on the right. 74

Figure 8.4: Thorvald during a field trial on May 21st 2019. 75

Figure 8.5: The RGB-D image at an instance in bagfile 1, compared to the equivalent point cloud. The edge where the uncut and cut grass meets is clearly visible in both pictures, as is the person standing in the field. 76

Figure 8.6: a) A top-view of the point cloud environment, the dense point cloud representing the uncut grass visible to the robot's right. The dashed line represents the desired line to be produced by the algorithm; b) A close up of the environment, in front of Thorvald the uncut grass can clearly be distinguished by its elevation over the ground. 76

Figure 8.7:The RGB-D image at an instance in bagfile 2 (on the left), compared to the equivalent point cloud (on the right). 77

Figure 8.8: a) A birds-eye view of the point cloud environment, with the dense point cloud representing the uncut grass visible to the robot's left. The dashed line represents the desired line produced by the algorithm; b) A close up of the environment, in front of Thorvald the uncut grass can clearly be distinguished by its elevation over the ground. 78

Figure 8.9: An image taken by the RGB-D camera, where two people move around in the critical zone. 79

Figure 9.1: True lateral and angular displacement values compared to estimated values, when the robot was placed the left of the line. 82

Figure 9.2: True lateral and angular displacement values compared to estimated values, when the robot was placed to the right of the line. 82

Figure 9.3: The sequence of positions and orientations of the robot as it navigates autonomously in the simulated environment. 83

Figure 9.4: The robot's tracked path exhibiting slight overshoot but little oscillation. 83

Figure 9.5: Linear forward-velocity of the robot recorded over 30 seconds. 84

Figure 9.6: Angular velocity of the robot recorded over 30 seconds. 84

Figure 9.7: a) The robot seen from its left side, before a plane is estimated; b) The plane of the uncut grass is coloured white, estimated by the RANSAC method applied in the node. 85

Figure 9.8: a) The points in the estimated plane are coloured green. The white points along the left side of the plane will be fitted to a line; b) The estimated plane with the white points on the left edge of the plane at another instance. 85

Figure 9.9: a) The RANSAC method demonstrated as it estimates a line along the edge of the uncut grass, with five outliers present in the last 50 cm of the critical zone; b) Another instance of a successful line estimation under the presence of six outliers. 86

Figure 9.10: a) The estimated line produced by the algorithm, shown as the blue marker is visualized in RViz; b) Another example of the detected line, which Thorvald will navigate along. 86

Figure 9.11: The robot is turned up to ca. 35-degrees, showing no effect on the generation of the line. 87

Figure 9.12: a) The robot seen from its right side, the plane of the uncut grass clearly distinguishable; b) The plane of the uncut grass coloured white, estimated by the RANSAC method applied in the node. 87

Figure 9.13: a) An estimated plane with selected points in white; b) Another estimated plane with selected points. 88

Figure 9.14: a) A line successfully estimated by the RANSAC method under the presence of three outliers in the field of cut grass; b) The RANSAC method successfully estimating a line in the presence of four outliers 88

Figure 9.15: a) A successful estimation of the line at an instance of time; b) The robot and the detected line as seen from the right, as it makes its way along across the field. 89

Figure 9.16: a) The line is successfully generated (coloured blue) as three people are situated in the critical zone; b) The presence and motion of the people have no effect on the line generation algorithm, even when the person is crossing the line. 89

List of Tables

Table 5.1: Defining the sensor alternatives. 41

Table 5.2: Description of the criteria used to select the sensor. 41

Table 5.3: Ranking of criteria from 1-5. 42

Table 5.4: The final score of the three sensors. 44

Table 7.1: Action server algorithm. 69

Table 8.1: Specifications of the captured data. 75

List of Abbreviations

Abbreviation	Meaning
API	**A**pplication **P**rogramming **I**nterface
FLIR	**F**orward-**L**ooking **I**nfra**r**ed Camera
GNSS	**G**lobal **N**avigation **S**atellite **S**ystem
GUI	**G**eneral **U**ser **I**nterface
IR	**I**nfra**r**ed
LIDAR	**L**ight **I**dentification **D**etection **a**nd **R**anging
LTI system	**L**inear **T**ime **I**ndependent system
MCDM	**M**ulti-**C**riteria **D**ecision **M**aking
PCL	**P**oint **C**loud **L**ibrary
PID controller	**P**roportional **I**ntegral **D**erivative controller
RANSAC	**RAN**dom **S**ample **C**onsensus
RGB-D	**R**ed **G**reen **B**lue-**D**epth (camera)
ROC	**R**ank **O**rder **C**entroid
ROS	**R**obot **O**perating **S**ystem
RTK	**R**eal **T**ime **K**inematics
SAW	**S**imple **A**dditive **W**eighting
SL	**S**tructured **L**ight
ToF	**T**ime **o**f **F**light

Chapter 1

Introduction

1.1 Background

In the latter parts of the 18th century, the Industrial Revolution took place in Great Britain, harvesting the power of running water and steam. In the early stages of the 20th century, the first conveyor belts were introduced, allowing for a new age of mass production. A continuation of these advancements was the introduction of digital automatic control systems, now an integral part of today's society. Since the introduction of these systems we rely on automated control to perform tasks and produce a vast selection of products for us, like the repetitive pre-programmed tasks performed by a packaging-robot or a machine in an automized bottle filling factory.

The advancements in technology and wireless communication have recently led to a new phase in automation technology: The introduction of autonomous robots. Whether the robot is a rover deployed on another planet, a reconnaissance drone or a driving-assisted car, there are many sectors that already benefit from autonomous robotics (or soon will). The use of autonomous robotics will only increase with the widespread implementation of internet of things and Industry 4.0, the latest advancement in the list of industry revolutions (1).

One of the sectors that could greatly benefit from autonomous robotics is the agricultural sector, a sector that today relies on extensive manual labour. The agricultural domain has a relatively small degree of implemented automation and agricultural robotics. By developing robots to suit the needs of the agricultural sector, one can lessen the burden on farmers, reduce the packing of soil, and the use of pesticides, all while performing tasks more cost-efficient than in conventional farming.

One of these tasks is the cutting of grass fields, a process which can be greatly improved and made autonomous by implementing existing technology. In cooperation with the Norwegian University of Life Sciences (NMBU), the GrassRobotics-project was initiated in 2018. The goal was, and still is, to automate the task of forage production, envisioning a lightweight robot with the ability to cut and collect grass (2). If this goal can be achieved, it will be a significant step towards sustainable farming. A crucial part of achieving this goal, however, a milestone that has not yet been reached, is the development of an algorithm which gives the robot the ability to detect the ideal path in the grass field, and autonomously navigate along it.

1.2 Motivation

Since we first learned how to farm soil around ten thousand years ago, a lot has changed. A farmer would farm for his family, and perhaps a small community, only using a small patch of soil to do so. Farming is by its very nature wearing on the soil, and it was of course important, as is it now, to keep the soil healthy for future generations. By farming in such small scale, however, it was possible to simply move on to new land when the field was overused. This allowed for the overused fields to naturally recover.

In today's modern farming that is not the case, as it is simply not an option. Food production is a billion-dollar industry, and while the human population keeps increasing, that will most likely not change. The soil rarely has time to naturally recover, as the soil is constantly turned, ploughed, ripped, and sprayed with pesticides. Every acre is utilised for short term gain, as heavy tractors and equipment are responsible for compacting the soil.

Soil is often wrongly regarded as a homogeneous dead substance, but that is certainly not the case. Fungi, bacteria, and microbes live in a healthy soil, and worms and roots are important for the symbiosis of the soil's advanced ecosystem (3). Channels and pores in the soil are important to make sure the roots can reach the nutrients that are stored in the soil, and water can flow freely and unobstructed. When heavy machinery drives on the fields, these pores and channels are closed. This leads to a compacted structure and the complicated system of decomposers that thrive on dead plant material is disturbed.

The strategy of making tractors and equipment larger and heavier to cope with ever increasing demands has reached its capacity. We must now think differently, and the Thorvald project can be part of a potential solution. The project has created an innovative robot that will automate

and simplify the farmer's workday, while at the same time lessen the negative effects of tractors and the soil compaction their use entails.

1.3 The Agricultural Robot Thorvald

This thesis is written for the Robotics and Control Group at NMBU, contributing to the already developed robot named Thorvald. This robot is an agricultural robot, made modular to be flexible and applicable to several different environments, like greenhouses and open fields. It may be used in several different ways, for example as a harvester, a data collector and/or for disease management.

The Thorvald robot is linked with, and produced by, the company Saga Robotics. This is a company that is working to mass produce and commercialize the robot which is developed at NMBU. Saga Robotics seeks to make the agricultural sector safer, sustainable and more productive, whilst producing nutritious food that cost less for the consumer (4).

1.3.1 Thorvald I

In 2014 a group of students developed the first version of the agricultural robot Thorvald, now known as Thorvald I, together with professor and leader of the robotic research team at NMBU Pål Johan From (5),(6),(7),(8). The idea was to design and build a lightweight autonomous agricultural robot, reducing soil compaction and providing an electrical alternative to the traditional heavy tractors running on fossil fuels.

The result was a four-wheel drive robot, equipped with an electrical motor (600 W) on each wheel for four-wheel drive. The robot itself weighed no more than 150 kg and could be used for tasks like carrying seeds and equipment. Additionally, it was regarded as particularly practical to use the robot to plant seeds and implement disease-control. The robot can be seen in Figure 1.1.

Figure 1.1: Thorvald as presented in 2016 (9).

1.3.2 Thorvald II

In 2016 the next chapter of the Thorvald story commenced, as the approach diversified in a more modular direction. It was realised that for an agricultural robot to be truly innovative and effective from a farmer's perspective, it had to be able to adjust to the plethora of challenges it may face in different farming environments. To be as flexible as possible in different topology and production systems, the hardware was made modular and the software generalised to work in the different configurations (10). This project was formed over a period of time by students working on their master's theses and PhD dissertations. Configuring the robot in different modules with different rack width, power consumption, power distribution, sensor equipment, load capacity, etc., allowed the Thorvald project to easily and quickly transform to be used in completely different environments like greenhouses, tunnels and open fields. Some of the current fleet of robots assembled from Thorvald II modules are shown in Figure 1.2 below.

Figure 1.2: Some of the different configurations of Thorvald II modules (11).

1.3.3 GrassRobotics: The Cutting-Edge Grass Cutter

Several previous master's theses and PhD dissertations have been written on the topic of Thorvald, and students at NMBU have been continuously developing the associated technology and hardware used by the robot. Particularly relevant to this thesis, is a thesis that was submitted in 2018 on the subject of researching energy efficient methods to configure Thorvald to be used to cut grass. The thesis was written by the students Nickolas Grelland and Andreas Xepapadakis Isaksen (12).

The thesis that Grelland and Isaksen submitted, developing and equipping Thorvald with a tool for cutting grass, was a part of a continuing project named GrassRobotics. The project has several partners, including both the academic and private sector. They are (listed in no particular order): NIBIO, Norsk landbruksrådgiving Agder, Saga Robotics, Fylkesmannen i Vestland, HMR Voss, Orkel, Felleskjøpet Agri, TINE, and The University of Lincoln (2). The project span is from the 1st of April 2018 to the 31st of December 2021, and funded by the Foundation for Research Levy on Agricultural Products (FFL), as well as the Agricultural Agreement Research Fund (JA) (2). The objective of this project is to produce an agricultural robot that can be applied to the process of sustainable forage production. The lightweight robot's main tasks will be to cut grass, gather and transport equipment (2).

The aforementioned thesis by Grelland and Isaksen is relevant to the work presented in this thesis, as the aim is to further build on this configuration of Thorvald and contribute to the fulfilment of the GrassRobotics-project's goals. Figure 1.3 presents the current prototype of the grass cutting Thorvald configuration.

Figure 1.3: The grass cutting Thorvald configuration (12).

1.4 Problem Statement

In its current state, Thorvald uses the GNSS system to navigate both on and off the field. Although the use of a Real Time Kinematics (RTK) capable GNSS receiver allows for a very high positioning accuracy, it works rather inefficiently for example when used to navigate Thorvald in a sloped field. Time has shown that the current configuration is unable to efficiently cut the grass in such a field, leaving pieces of uncut grass behind.

It can also be a potentially fragile system susceptible to external influences like jamming, as well as being affected by potentially poor coverage in rural areas. Many of Norway's fields are located in valleys or mountainous areas, particularly on Norway's west coast. As these areas lie in regions of high northern latitude, the GNSS signals often have a lower elevation. This can lead to the signal being blocked, and never received (13).

Most of the fields are found outside of cities, and so the mobile data coverage can also vary greatly. Access to mobile data is essential to maintaining the high RTK position accuracy, as the correctional data is transmitted over the internet. Alternatively, a local base station must be set up to broadcast the correction data over radio transmissions (usually in the ultra-high frequency range 300 MHz – 3 GHz). This can however be a costly and comprehensive solution.

To increase Thorvald's reliability and effectiveness, it is preferable to equip the robot with another sensor technology, supplementing the GNSS RTK receiver. An optimal solution would see these sensors work in unison with the GNSS RTK, weighting which sensor-input to listen to in real time as the robot navigates. Identifying and autonomously positioning the cutting tool along the edge of the uncut grass as the robot traverses the field, means that the robot will cut the grass more efficiently.

Firstly, the robot will have to localize the line/edge of the grass that has not yet been cut, by differentiating between the cut and uncut grass (hereby referred to as the *grass line* or *separation line*). In Figure 1.4, this line is represented as a red arrow. The robot then has to realign and navigate itself to the separation line. Lastly, it must follow the line precisely enough, so that the grass is cut without the cutting tool overlapping the line or leaving uncut grass behind.

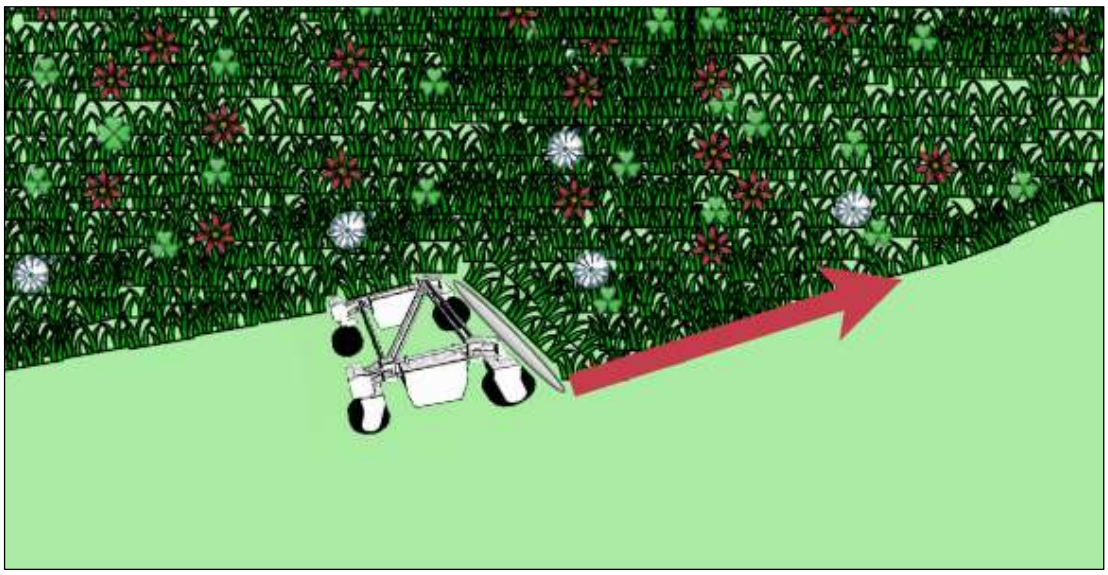


Figure 1.4: The robot as it cuts grass in the field, optimally aligned and following the separation line represented by the red arrow.

1.4.1 Thesis Main Objective

The following main objective have been defined for this thesis:

This thesis' main objective is to establish methods allowing a robot to identify, and autonomously follow, the line separating cut and uncut grass. The best suited sensor technologies will be selected and implemented in the real-time autonomous system.

The main objective can be broken into four crucial steps:

1. Observation: The robot must extract relevant three-dimensional information from its sensorRobot localization: To make decisions and navigate autonomously the robot must know its orientation and location relative to the grass line.
2. Perception: The robot must be provided with the algorithms to determine what course of action will be taken to achieve its objectives, by interpreting the sensor data.
3. Navigation Through Velocity Control: The robot must regulate the actuators to successfully perform its tasks and follow the grass line.

1.4.2 Thesis Sub Objectives

The following sub objectives have been defined for this thesis:

- Research and select the best suited sensor technology for the detection of the separation line.
- Develop algorithms for the real-time estimation of the separation line.
- Develop feedback-control algorithms for the robot, to calculate its error offset and adjust its trajectory accordingly.
- Test the algorithm with simulation software to evaluate the suggested detection and control algorithms.
- Evaluate the suggested detection and control algorithms in field trials.

Chapter 2

Theory and Technology

2.1 GNSS RTK

Modern society's main tool for outdoor navigation is the Global Navigation Satellite System (GNSS). Satellites orbiting the earth transmit data so that the location of the receivers can be calculated. GNSS based navigation technology, however, has its limitation when it comes to position data precision.

Errors like signal propagation due to the signal passing through the layers of the atmosphere, and orbital errors in the estimated orbits of the satellite, cause inaccuracies in the location calculations (14). These errors are approximately identical for a given area at a given time and so they can be removed by applying a differential processing technique, known as real-time kinematic GNSS (GNSS RTK). A base station compares the signal from a satellite with its own known, stationary position, thus calculating errors that exists in the GNSS signal.

The errors are then passed on to non-stationary GNSS receivers via radio waves, allowing them to correct the signal they receive. Signal corrections can also be transmitted over the internet, via a cellular modem. The GNSS RTK technique allows for high accuracy localisation up to 1-2 centimetres in the horizontal plane (15).

2.2 Topological Maps

Before assessing the various ways one can improve the grass cutting robot's efficiency, one must understand how the robot configuration's use of navigation-technology and topological mapping works in its current state. To navigate in and around the fields, a topological map is defined as a graph. In graph theory (discrete mathematics), a graph (G) is a data structure used

to describe ordered pairs of *vertices (V)* and *edges (E).* Any graph can be described with the following graph equation:

$$G = (V, E) \tag{1}$$

The vertices are connected by edges, as illustrated in Figure 2.1. These edges are either defined with an unspecified direction (*undirected graphs*), or with a specified direction (*directed graphs*). The two sets of vertices and edges in the example graph presented in Figure 2.1, are defined as follows:

$$V = \{v_1, v_2, v_3, v_4\} \tag{2}$$

$$E = \{e_1, e_2, e_3\} \tag{3}$$

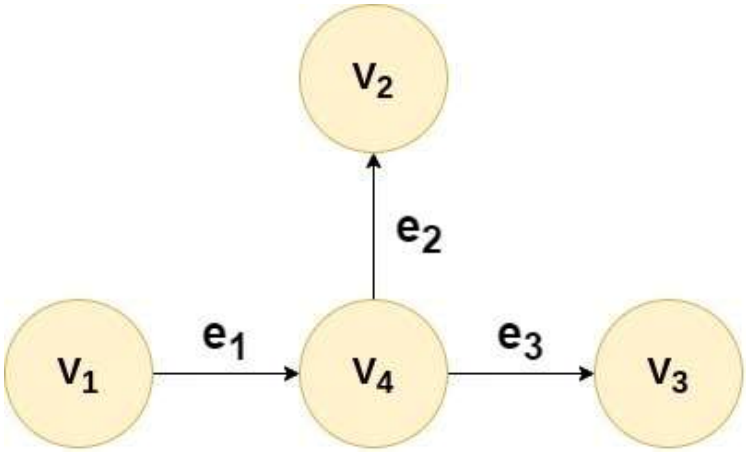


Figure 2.1: A directed graph with four vertices connected by three edges (specified direction).

This data structure is used to represent connected data in a vast range of applications, one of which is the topological map used by Thorvald for navigation purposes. The environment is described in the map by applying graph theory, i.e. by using vertices and edges. The vertices, or *nodes,* represent locations. The directed and undirected edges connect the nodes and represent *actions* which implement controllers, which then move the robot from one node location to the next. Such a network is illustrated in Figure 2.2, illustrating a field and how the topological map may be set up. A set of nodes and edges are defined manually to contain certain positions, for example starting from a docking station to the field itself, as well as the edges specifying how the robot can move to and from these points. Different navigation behaviours may apply between nodes, for instance the robot may have to navigate between the nodes in a straight line. When asking the robot to move to a certain node's position, for example from the

charging station to the edge of the field, it will calculate its route between the starting node and the final node. By using the edges that connect all the nodes, it will pass through other nodes on its path to reach its destination.

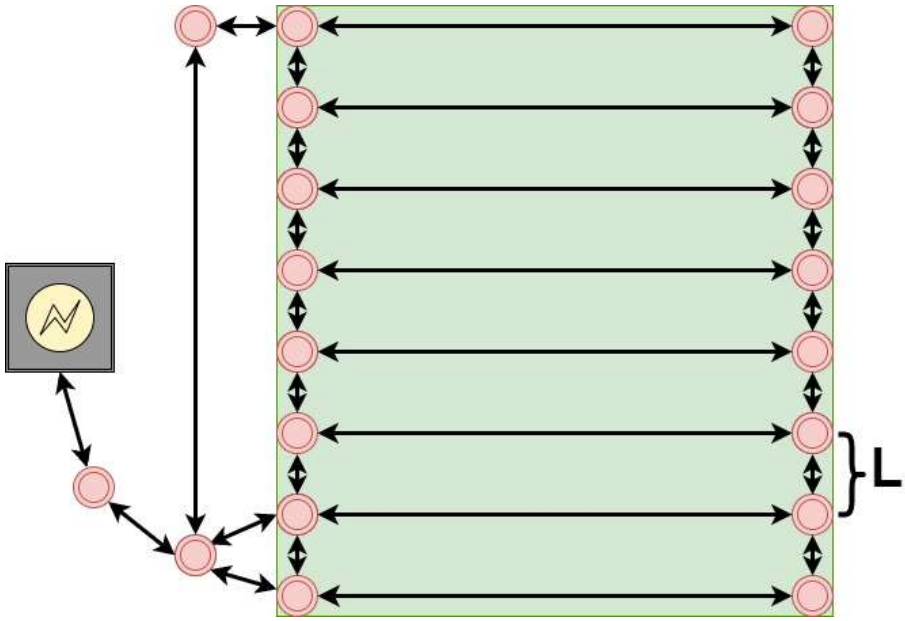


Figure 2.2: Illustration of the topological map, demonstrating the placement of nodes around a field, and the edges connecting them.

The nodes along the edges of the field are automatically generated from the dimensions of the field. A node is generated on each side of every row, as shown in Figure 2.2. The number of nodes generated depends on a user set distance between each row (L). This distance could for instance be the width of the cutting tool on Thorvald. The robot will be instructed to move to the field's starting node (illustrated by the lower left node in the field, Figure 2.2), it knows where its next node is located (lower right node in Figure 2.2), and from there it works its way across the field controlled by the action defined by the edge. When it reaches the node at the end of the row, it navigates to the node located at the starting position of the next row, and so on.

2.3 Navigation and Positioning

The Thorvald robot is equipped with a GNSS RTK receiver. However, an obvious challenge with deriving topological maps from the GNSS RTK receiver, is that it is a 2D visualization of the 3D topography of the field. This might be a very effective approach in some places, for instance there is a Thorvald module running at the University of Florida, picking strawberries,

and treating them for mildew with UV-light. Equipped with just the GNSS RTK receiver and by following these nodes, this simple technique works well. Crucially, however, the fields in Florida are rarely sloped like the fields you would find in Norway, particularly along the west coast.

With sloped fields, the two-dimensional mapping will lead to a topological map with an erroneous, smaller dimension of the field. This will lead to lines of uncut grass between the rows, with a width depending on the slope of the field.

Imagining that the field illustrated in Figure 2.3 has a 45-degree rising slope, the Pythagoras theorem dictates that the two-dimensional projecting of the three-dimensional rows traversing the field, would result in the robot leaving behind a line of uncut grass with the width e:

$$e = \left(\sqrt{2} - 1\right)L \tag{4}$$

For a truly autonomous approach, instead of having pre-decided paths, the robot should "see" the environment, find the ideal path for the cutting tool, and adjust to it accordingly.

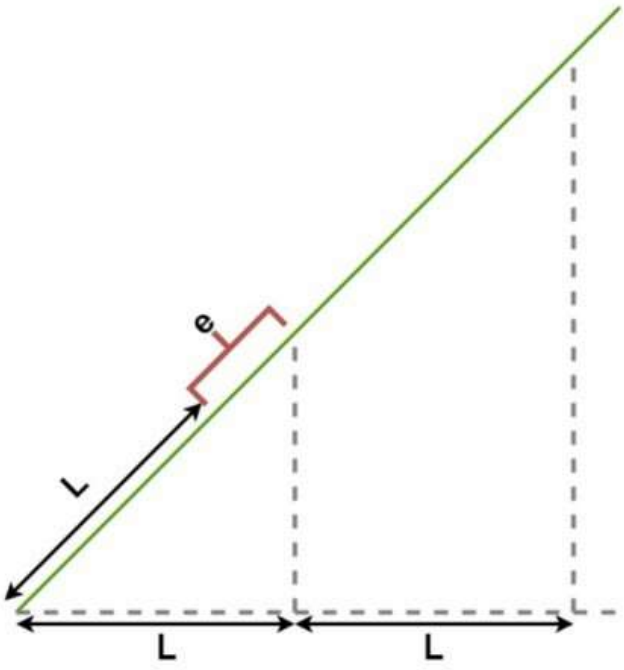


Figure 2.3: Illustration of a field with a 45-degree slope, visualised in 2D.

2.4 Thermographic Camera

A Forward-looking infrared camera (FLIR) is a thermographic camera that registers radiation in the infrared spectrum, i.e. thermal radiation. By registering the different emission levels of natural infrared radiation, it creates and divides an image in regions of different temperatures. Furthermore, these regions can then be represented in different shades of colour. The convention is to assign warm objects a colour in a shade of red and yellow, and cold objects in a shade of blue and green. There are many different thermographic cameras on the market, designed to fit specific purposes. Difference in wave range detection, software compatibilities like digital image processing and design in terms of weight and/or how compact the camera is, varies. Naturally, so does the price.

A collaborating project between the educational institutions NMBU and the Pontifical Catholic University of Rio de Janeiro, partially supported by the UTFORSK Partnership Program from The Norwegian Centre for International Cooperation, has explored thermal based navigation. By utilizing a FLIR® One V2 infrared camera in a smartphone, mounted to a robot moving though corridors of vegetation in a sugarcane crop, thermal images and videos of the surroundings were captured (16). As presented at the International Conference on Robotics and Automation (ICRA) in 2019, the different heat signatures of the environment produced infrared images that allowed for effective navigation through the narrow crop corridors (17). The method is based on the simple fact that the ground emits more heat, i.e. infrared radiation, than the surrounding vegetation.

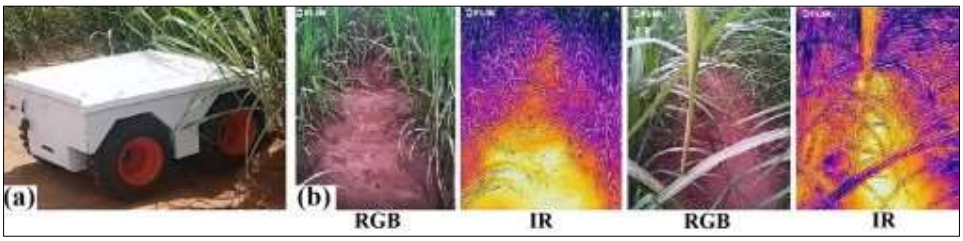


Figure 2.4: a) The TIBA Robot in the field; b) the RGB pictures it took in the sugarcane tunnels, compared to the correlating IR-images (17).

As seen in Figure 2.4, compared to an RGB camera, the poor visibility conditions of the environment are made better with the thermal camera.

2.5 RGB-D Camera

Another possible sensor to be used for position estimation and autonomous navigation is one that produces camera image streams. The RGB-D camera produce digital images in colour (RGB refers to the colour space with red, green and blue primaries) and with depth data on each pixel (D). This means that the colour images are accompanied by an interrelated point cloud, allowing for high level feature manipulation like plane segmentation. The depth information can be acquired either by active or passive methods, depending on camera model and manufacturer.

A passive approach to depth calculation uses stereo vision and is simply two cameras that take advantage of the known relationship between the two produced images. By triangulating and comparing the points in the two images, considering the known transformation from one frame to the other, the depth can be accurately calculated (18).

There are also several active approaches to acquire depth information of the environment, where the word active refers to the actively altering of the scene in the process. The two most common active approaches are known as *Time-of-Flight* (ToF) and *Structured Light* (SL). The active ToF-method works in many ways like a LIDAR (see Chapter 2.5), emitting infrared pulses and detecting the reflected rays. The distance to the objects in the scene can continuously be calculated by knowing the respective flight time of the registered ray. The SL-method, however, replaces one of the cameras in the stereo system with a projector or other light source. It projects a distinguishable structured light pattern into the scene, effectively simplifying the triangulation operation (19). The patterns can be points, lines or light sheets, and by projecting them in known arrangements, they can be transformed to the camera's frame and used as reference to calculate depths.

Combinations of the technologies also exists, where a projector or other light source is present in addition to the two cameras in the stereo system. This way the best suited approach can be chosen, either active or passive. For example the Intel RealSense line of RGB-D cameras offer such a combination, being equipped with two IR-cameras as well as an IR-projector (19). The Intel RealSense Depth Camera D415, one of the sensors previously used by the Thorvald-project, is such an RGB-D camera. The images produced by this camera can be seen in Figure 2.5, and the camera itself is pictured in Figure 2.6.

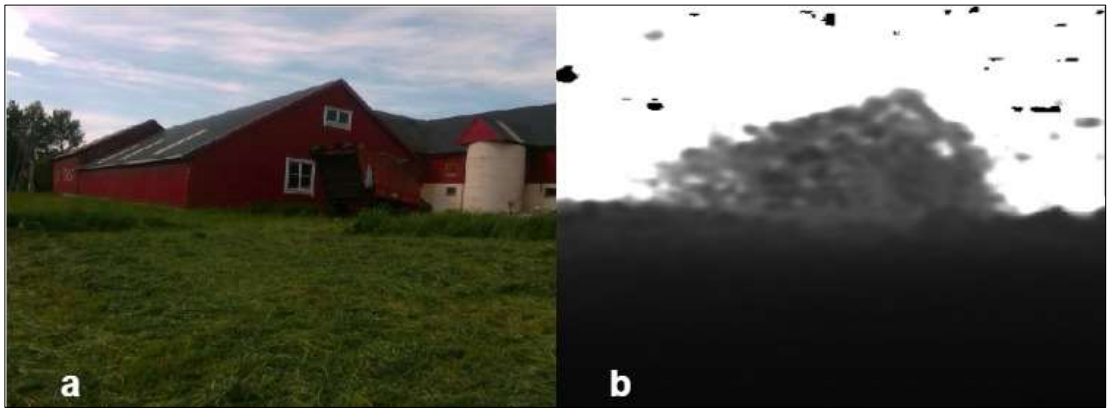


Figure 2.5: a) A coloured image showing the RGB component of the image; b) The depth component of the image produced by the RGB-D camera.

The Intel RealSense Depth Camera D415 has a frame rate of 30 frames per second (fps) and an output resolution of 1920x1080p. The minimum distance to target for depth measurements is 0.3m. The depth sensor has a vertical view of 40 ± 1 degrees and a horizontal view of 65 ± 2 degrees (20). The maximum range for depth calculations is 10 metres, but it is specified by the producer that it varies with lighting conditions and calibration of the camera.

Figure 2.6: The RGB-D camera RealSense D415 from Intel (20).

2.6 LIDAR

LIDAR is an acronym for Light Identification Detection and Ranging, and is a technology widely used in robotics and autonomous vehicles for navigational purposes. The LIDAR provides three-dimensional spatial information and allows for accurate position measurement and motion calculation. The sensor works by pulsating emissions of a laser beam, and then registering the returning beams and their intensities as they hit objects and reflect back to the sensor's light receiver. The distance to the object (d) is then calculated as a function of flight

time ($t = t_{sent} - t_{received}$) and the speed of the laser beam propagation through space (c) (21). As the registered flight time covers both the time to and back from the object reflecting the laser beam, it must be divided by 2 (as presented in equation 5).

$$d = \frac{t}{2} \cdot c \tag{5}$$

LIDAR sensors come in several varieties, however, the grass cutting robot is already equipped with a VLP-16 produced by Velodyne. The VLP-16 is a widely used LIDAR sensor and has been equipped by the Thorvald-project at NMBU to detect and avoid humans as a safety implementation on the Thorvald grass cutter.

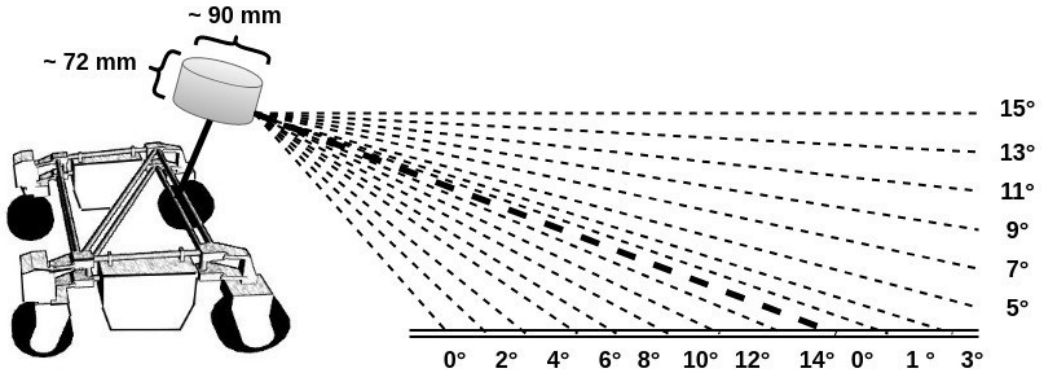


Figure 2.7: The VLP-16 has a 30 degree field of view, it is mounted on Thorvald and angled according to the range needed (the illustration is not to scale).

The VLP-16 has, as its name implies, 16 channels where each laser beam has a wavelength of 905 nm (22). The laser's (and receiver's) vertical alignment is depicted in Figure 2.7, illustrating the 30-degree vertical field of view. The lasers spin around the vertical axis at a rate of 5 – 20 Hz, allowing for a 360-degree horizontal field of view. The sensor has a range of around 100 metres, and with its low weight (830 grams), small size, and low power consumption (around 8 W), it can easily be fitted on Thorvald and powered by the robot's batteries (22).

The generated sensor data is structured in header files containing time stamps, sensor model, etc., as well as the generated point cloud data like distance and angle to each point. The data also contains a scaled intensity factor based on the reflectivity of the reflecting surface. By decoding this data, a point cloud can be visualised in a Cartesian coordinate system. The VLP-16 has the ability to produce around 300 000 such points per second, which allows for high quality and accurate information of the environment surrounding Thorvald.

2.7 Point Clouds

Point clouds are an effective format for manipulating, visualizing and mapping 3D data, and as such the format is the data output of many 3D data capturing systems and depth sensors like the discussed LIDAR and RGB-D camera. It is also a flexible format which allows for easy conversions to and from other types of surface representation, like a surface mesh. This conversion can be done by sampling, simply performed by generating points within a random sample of a polygonal mesh (23). Figure 2.8 shows an example of a point cloud in the shape of a house, superposed over an image of the outer wall.

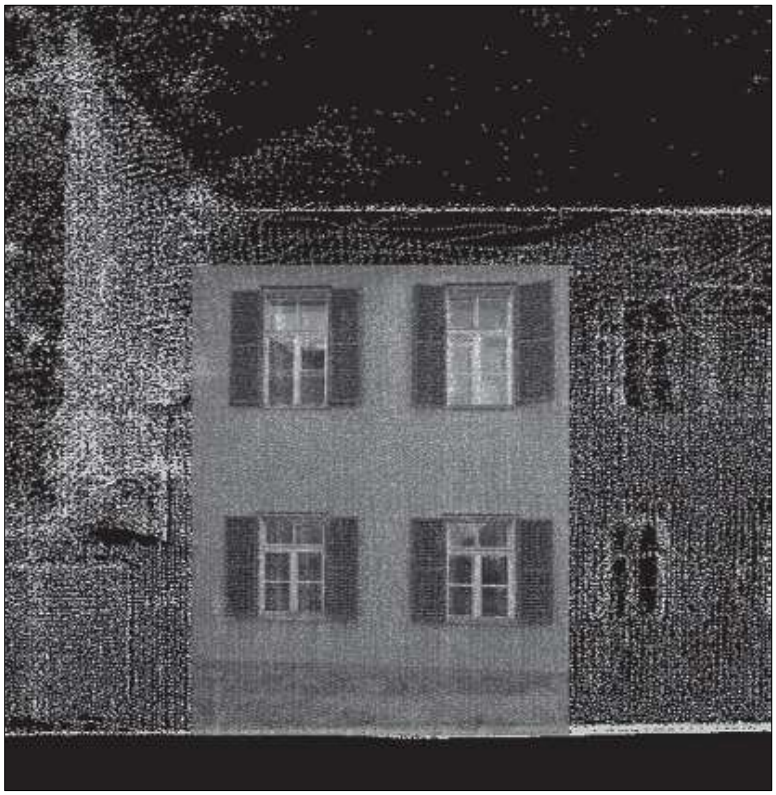


Figure 2.8: An example of a point cloud of a house (24).

The point cloud data output consists of a number of data points which are all defined in a 3D Euclidian coordination system. This means that every single point in the point cloud can be located by their x, y and z coordinates. Figure 2.9 illustrates how the point cloud's representation of a geometric shape in space, like a meadow of grass, can be represented as an m x 3 matrix where every row represents a single point in a cloud with a total of m points.

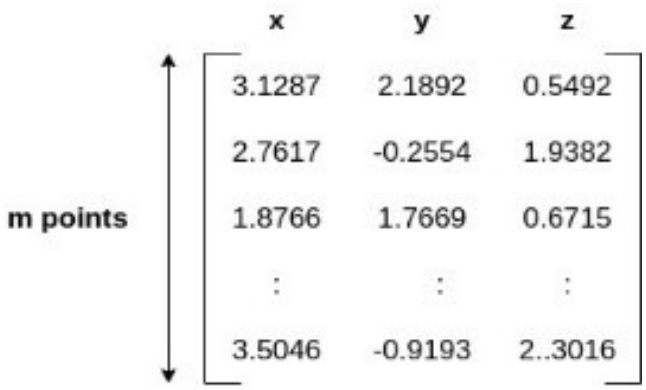


Figure 2.9: The coordinates of points in a point cloud, arranged in a matrix form.

2.8 Estimation Models

2.8.1 Least Squares Regression

The Least Squares Regression method is a common statistical analysis technique used to fit a line, a plane, or other geometric shapes to a set of points in a dataset. In the case of fitting a regression line to the linear relationship of paired data (x,y), it is done by attempting to minimize the squares of the residuals, i.e. the variance. The fitted line is defined by the following linear equation:

$$y = ax + b \tag{6}$$

Here *a* represents the slope of the line and *b* is the intercept of the y-axis. To minimize the squares of the residuals, the unique values of a and b are found to minimize the following equation:

$$F(a,b) = \sum_{i}^{n} (y_i - ax_i - b)^2 \tag{7}$$

Points in the dataset are represented by (x_i,y_i), and *n* represents the number of points in the set. To minimize the function *F(a,b)* the derivative with respect to *a* and *b* are taken and the equations are solved for equal to zero :

$$\frac{\partial F}{\partial a} = 0 \tag{8}$$

$$\frac{\partial F}{\partial b} = 0 \tag{9}$$

These equations are linear in *a* and *b* and will therefore produce unique solutions for the two variables. Alternatively, the following equation (equation 10) is also true for variable *a* and the system can then be solved for variable *b*.

$$a = \frac{\sum_1^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_i^n (x_i - \bar{x})^2} \tag{10}$$

Here $\bar{x}$ and $\bar{y}$ are the means of the *x* and *y* values respectively. The least squares regression model can also be extended to the process of fitting points to a plane *P*, as the plane is nothing more than a linear object where the number of dimensions is higher than two. The equation of a plane *P* can be expressed by a point p_0 on the plane (x_0,y_0,z_0) and a non-zero normal vector $v = [a, b, c] \in \mathbb{R}_3$. Any point *p* on the plane can then be found by a vector orthogonal to the normal vector (x,y,z). An expression of the plane can then be written as:

$$P(a, b, c) = v \cdot (p - p_0) = ax + by + cz + d \tag{11}$$

Here *d* is dependent on the point p_0, as *d* can be expressed by the following relation:

$$d = -ax_0 - by_0 - cz_0 \tag{12}$$

Another way of expressing the same plane is through a minimum of three points on the plane, $\{p_1, p_2, p_3\} \in P$. The alternative way of writing the plane equation is then:

$$z = d + mx + ry \tag{13}$$

The *d* variable represents the interception on the z-axis, and the *m* and *r* variables represent the slopes of the x- and y-axis respectively. To fit a plane to a set of points (x_i,y_i,z_i) by using the least squares (plane) method the residuals are minimised, i.e. the values of *d*, *m* and *r* are found so that the following function is minimised:

$$H(d, m, r) = \sum_1^n (z_i - d - mx_i - ry_i)^2 \tag{14}$$

To minimize the function *H(d ,m, r)* the derivative with respect to *d*, *m* and *r* are taken and the equations are solved for equal to zero :

$$\frac{\partial H}{\partial d} = 0 \tag{15}$$

$$\frac{\partial H}{\partial m} = 0 \tag{16}$$

$$\frac{\partial H}{\partial r} = 0 \tag{17}$$

These equations are linear in *d*, *m* and *r,* and so the system can be represented and solved as a system of linear equations, using the following matrix equation (AX = B):

$$\begin{bmatrix} \sum_i^n x_i^2 & \sum_i^n x_i y_i & \sum_i^n x_i \\ \sum_i^n x_i y_i & \sum_i^n y_i^2 & \sum_i^n y_i \\ \sum_i^n x_i & \sum_i^n y_i & n \end{bmatrix} \begin{bmatrix} d \\ m \\ r \end{bmatrix} = \begin{bmatrix} \sum_i^n x_i z_i \\ \sum_i^n y_i z_i \\ \sum_i^n z_i \end{bmatrix} \tag{18}$$

Solving this matrix equation will give a single solution for the variables *d*, *m*, and *r,* as long as matrix A is linearly independent. If matrix A is linearly independent it has a non-zero determinant (the matrix is invertible), and a single plane is fitted to the inliers.

2.8.2 RANSAC

RANSAC is an acronym for RANdom Sample Consensus and is an algorithm that is often used in manipulation and segmentation of point clouds. Processing the 3D data obtained from sensors is an integral part of autonomous navigation, and the RANSAC algorithm is perceived as a particularly robust method. The reason why is that unlike most other estimators, the RANSAC algorithm can be effective even in cases where there are as many as 50 % outliers present in the dataset (25). RANSAC can also be used to estimate many different shapes, like cylinders, lines, cones and spheres (26). Because the method is so general and applicable to a vast set of problems, there are many examples of practical applications of the technique. As a recent example, the RANSAC algorithm was used to identify roof slopes in a 3D city model generation project (27). The algorithm is an iterative process that repeatedly takes a small random sample of the dataset to form a hypothesis of the shape, then chooses the solution that maximizes the amounts of inliers based on a threshold parameter. The threshold parameter is

defined as the maximum distance a point can be from the hypothesised shape and be considered an inlier. Sometimes the model coefficients are further improved by applying an estimation model that is more sensitive to outliers, like Least Squares Regression, on the solution containing the most inliers found by the RANSAC algorithm.

To fit a line or a plane to the data, the RANSAC algorithm works by iterating the following process:

1. The dataset is represented in a point cloud $\beta = \{p_1 p_2 \dots p_n\}$. The smallest possible initial data sample-set is randomly selected. If the model is a line, then a sample-set of two points ($p_i and\ p_j$) is selected.

$$\{p_i p_j\} \in \beta \tag{19}$$

   If the model is a plane, then three non-collinear (the points cannot all lie on a straight line) points ($p_i, p_j\ and\ p_k$) in the data set are sampled, as this is the minimum points needed to create a plane.

$$\{p_i p_j p_k\} \in \beta \tag{20}$$

2. The coefficients of the respective models are calculated by applying the least squared method. For a line model the six coefficients produced by the algorithm are a point (x,y,z) on the line, and a non-zero vector parallel to the line, i.e. the direction (x,y,z). These coefficients, and the scalar λ. give the following vector equation:

$$f(p_i p_j) \rightarrow \vec{\ell}(a,b,c,d,e,f) = \begin{bmatrix} a \\ b \\ c \end{bmatrix} + \lambda \begin{bmatrix} d \\ e \\ f \end{bmatrix} \tag{21}$$

   In the case of a plane model however, there are four coefficients calculated; The plane's normal (x,y,z) and the distance from the origin to the plane.

$$f(p_i p_j p_k) \rightarrow\ P(a,b,c,d) = ax + by + cz + d \tag{22}$$

3. Define a set $S \in \beta$ of inlier points. These are points in the point cloud that are within the specified threshold distance T from the model:

$$S = \{p \in \beta \mid 0 \leq |d(p, P(a,b,c,d)| \leq |T|\ \} \tag{23}$$

After a set number of iterations, the hypothesised line or plane that contains the highest number of inliers, is chosen as the most likely solution. However, post processing coefficient optimization can also be performed. The initial solution is then optimised with the Least Squares method applied on all the inliers in the solution set *S*.

2.9 Multi Criteria Selection

A Multi-Criteria Decision Making (MCDM) technique can be described as a way to identify, rank and select an alternative from a defined set of alternatives. This is done by defining criteria and applying personal judgement (28). The technique is applied over four steps:

1. *Defining criteria and alternatives*

Firstly, the alternatives must be known, and the criteria of which the alternatives are evaluated and is chosen by, must be well defined. The criteria are divided in the following two categories:

- *Benefit:* These criteria have positive characteristics, meaning that a higher value is always preferred.
- C*ost:* These criteria have negative characteristics, meaning that a smaller value is always preferred.

2. *Weighting of criteria*

The different criteria are ranked, then the rank is converted to a more precise weight. Assigning weights straight away can be tedious and often imprecise. Ranking the criteria by priority, however, has been shown to be more effective (29). By using a defined formula to determine the weights of each criteria, i.e. its importance, makes the process less error prone and more reliable. Each of the criteria weights sum up to a value of 1 ($w_1 + ... + w_5 = 1$) and represent the impact each criterion has on the final choice.

The Rank Order Centroid (ROC) method is applied to generate the weight vector $W = [w_1, w_2, ..., w_i]$, in which $w_1 > \ldots > w_i$. All potential weights can be imagined as a simplex, a triangle shape where datapoints are arranged in a specific pattern with a uniform weight-vector distribution. As the name implies, the ROC method then calculates the centroid of this simplex, thus minimizing the potential error of the weights. The (ordinal) weights are calculated by the following ROC formula:

$$W_i = \frac{1}{m}\sum_{k=i}^{m}\frac{1}{k} \tag{24}$$

Here *m* is the total number of criteria.

3. *Determining the decision matrix*

The relationships between the alternatives (S_i) and the criteria (C_j) are established in the decision matrix $D = [d_{ij}]_{nxm}$, where every alternative is evaluated numerically with a performance score against the criteria. The performance score is dependent on the criteria's category, which means that for a benefit criterion, a sensor that perfectly fills the criteria is valued as a 5, whilst for a cost criterion it would be valued at 1.

To be measured on the same scale so that all the criteria can be considered benefit criteria, the criteria in matrix *D* must be normalised to create matrix $N = D[n_{ij}]_{nxm}$. Several normalization techniques could be applied, however vector normalization is the chosen method as studies have found it to be the optimal technique when dealing with similar MCDM problems (30). Benefit criteria is normalised by equation 25 and cost criteria by equation 26:

$$n_{ij} = \frac{d_{ij}}{\sqrt{\sum_{i=1}^{n} d_{ij}^2}} \tag{25}$$

$$n_{ij} = 1 - \frac{d_{ij}}{\sqrt{\sum_{i=1}^{n} d_{ij}^2}} \tag{26}$$

4. *Scoring the alternatives*

The last step is to apply the *Simple Additive Weighting* (SAW) technique to rank the alternatives and ultimately choose the one most applicable to the project. The technique is implemented to score the alternatives by establishing a weighted average based on the all the different criteria and their established importance. The score *R* is the sum of the normalised values of the decision matrix multiplied with the relevant criteria weight, specifying each criterion's importance. The method is sometimes referred to as a weighted linear combination, as it is simply a proportional linear transformation of the normalised matrix data.

Thus, the relative order of magnitude of the data stays the same. The scoring equation is defined as following:

$$R_i = \sum_{j=1}^{n} d_{ij} w_j \tag{27}$$

Chapter 3

Control Systems

3.1 Laplace Transform

The Laplace integral transform maps a function in the time domain to a frequency domain, known as the s domain. The Laplace transform is defined below, where *f(t)* represents the function in the time domain, and $\mathcal{L}\{f(t)\}$ or *F(s)* represents the Laplace transform of this function.

$$\mathcal{L}\{f(t)\} = F(s) = \int_0^\infty f(t)\, e^{-st} dt \tag{28}$$

The s represents a *complex frequency*, i.e. a frequency which is expressed as a complex number. The frequency of the oscillation is expressed in a real number, and how the oscillation amplitude is changing over time (amplifying or shrinking) is expressed by a complex number. The s plane is simply a complex frequency plane allowing for the plotting of the Laplace transform and its behaviour in respect to these complex frequencies.

A linear time invariant system (LTI) is a system, like the name implies, where the input has a linear relation to the output, and the output does not change if the input is applied at different times. Calculating the output of an LTI system given any input signal, can be done by determining the system's *impulse response*. This response is simply the output of the system when the input is an impulse function, a function that can be described as an infinitively short pulse containing all frequencies in equivalent quantities. This means the Laplace transform of the impulse function is equal to 1, and the impulse response embodies the response of the system to any frequency. An RLC circuit is an example of such a system, where the output

signal can be predicted if the impulse response is known. The same is true for many other systems, and even systems which are not linear and time independent are often modelled as if they were, as it allows us to predict an estimated output of the system.

The Laplace transform simplifies several mathematical operations when dealing with LTI systems. However, the main advantage of using Laplace transformation in control theory, is that one avoids having to deal with convolution operations. Solving convolution equations is often required to solve problems in the time domain. In the s domain, however, convolution is performed by a straightforward multiplication, thus allowing the output functions of LTI systems to be calculated with a smaller degree of complexity.

3.2 Transfer Functions

The block diagram shown in Figure 3.1 shows an LTI system where the Dirac delta δ(t) function is the input. The Dirac delta function is a mathematical model, a generalised function, that is used to represent a unity impulse. Its amplitude is equal to zero for all values of *t*, except for when *t* is equal to zero, where its amplitude is infinite. Therefore, the integral of the Dirac delta function is equal to 1, and thus such an input is called a unit impulse input.

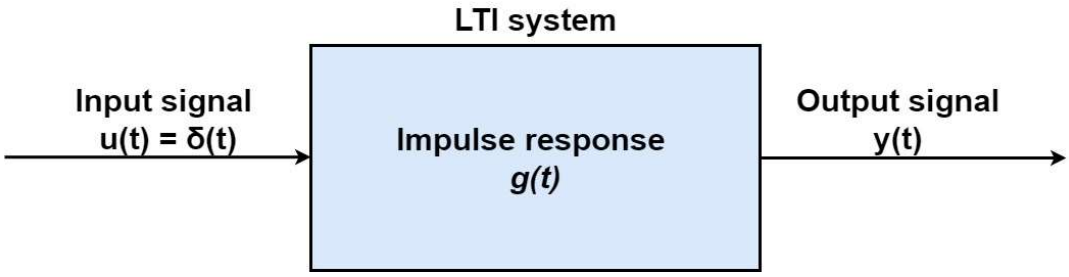


Figure 3.1: A block diagram of an LTI system, the input is a Dirac delta function which operates through the system's impulse response and produces the output function.

By determining the system's impulse response function *g(t),* operating in the time domain, the output of the system *y(t)* can now be calculated by convolving the impulse response function with any input signal *u(t)*.

$$y(t) = u(t) * g(t) \tag{29}$$

Operating in the s-domain, this relationship can also be expressed as follows:

$$y(s) = u(s) \cdot g(s) \tag{30}$$

The Laplace transform of the unit impulse function $\delta(t)$ is unity, thus the Laplace transform of the output response of an LTI system, *y(s),* to a unit impulse input is equal to the impulse response *g(s)* (when the initial conditions are zero).

This impulse response of an LTI system, i.e. the ratio of *y(s)* to *u(s)*, is called the *transfer function* (again, given that every initial condition is equal to zero). The transfer function can therefore be described as the relationship between any input signal and the output signal, as shown in Figure 3.2. The transfer function can be expressed mathematically as follows:

$$G(s) = \frac{Y(s)}{U(s)} \tag{31}$$

The equation shows how any input function *U(s)* multiplied with the transfer function *G(s)* produces the output function *Y(s).* In other words, the control system receives an input which is processed through the transfer function, which then leads to the generation of the output. This means that a transfer function is a function that can be used to model and analyse different types of components, like an actuator, a controller, or a filter, simplifying the process of control system design. The transfer function also simplifies the process of analysing control systems, as it can be used to describe its properties, like its stability.

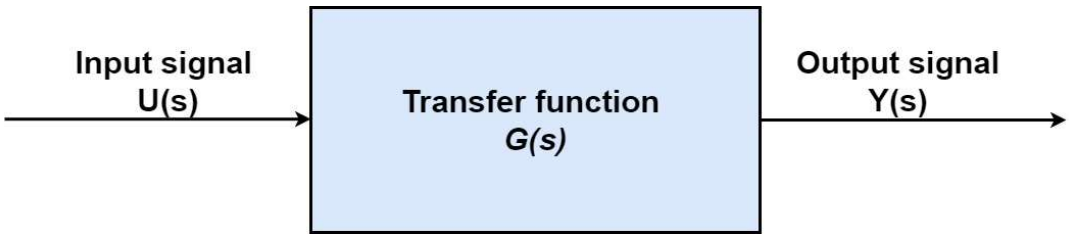


Figure 3.2: A block diagram of an input signal operating on a transfer function to produce an output signal.

The block diagram presented in Figure 3.2 can now be extended to control the output signal in a closed-loop system. Figure 3.3 illustrates the block diagram of such a system, where the output signal Y(s) is compared to the reference input *R(s)* (the desired value) at the summing point. Thus, the input signal U(s) is the reference input minus the output signal, where the output signal is obtained by multiplying the transfer function G(s) with the input signal U(s).

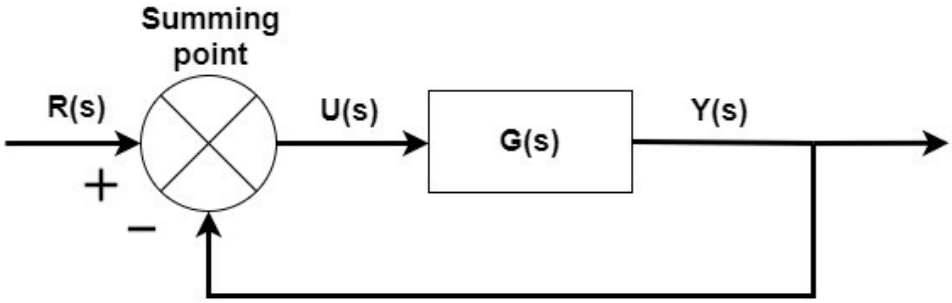


Figure 3.3: A block diagram of a closed-loop system.

The block diagram presented in Figure 3.3 assumes that the form of the output signal is the same as the reference input signal. If the input and output dimensions are not the same, however, the output signal has to be changed before it is compared to the reference signal at the summing point. Figure 3.4 shows the closed-loop system that includes a feedback element, for example some sensor that reads the signal output, to handle this manipulation of the output signal. The manipulated output signal *B(s)* is produced by multiplying the feedback element's transfer function *H(s)* with the output signal.

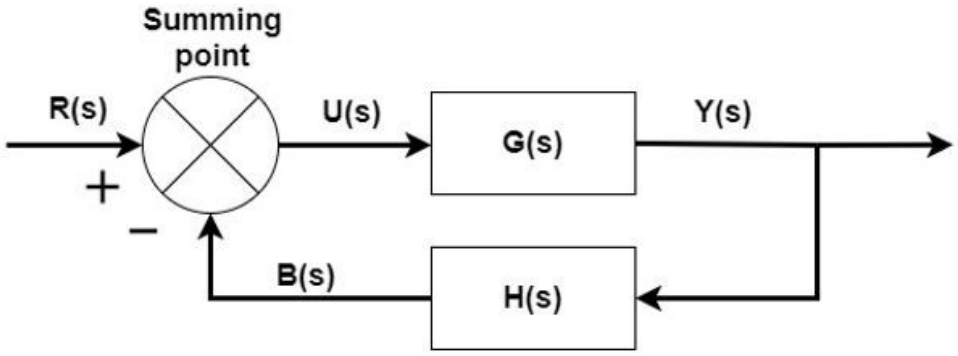


Figure 3.4: A block diagram of a closed-loop system including a feedback element.

3.3 Proportional-Integral-Derivative Control

There are several types of feedback control systems, but perhaps the most popular one is the *proportional-integral-derivative* (PID) controller (and its different variations). The controller is widely used for its simple and intuitive form, as well as the effective control loop feedback algorithm that can be useful for many different applications. The PID controller *C(s)* in its ideal form is expressed by the following equation:

$$C(s) = \frac{U(s)}{E(s)} = K_p(1 + \frac{1}{T_I s} + T_D s) \quad (32)$$

With the relationship defined in equation (33), the PID controller can also be expressed in the parallel form shown in equation (34):

$$K_p = K_I T_I = K_D T_D \quad (33)$$

$$C(s) = \frac{U(s)}{E(s)} = K_p + K_I \frac{1}{s} + K_D s \quad (34)$$

In these equations *E(s)* is the input and *U(s)* is the output of the controller in the loop, called error and control signal, respectively. *S* is the complex argument in the Laplace transform, K_p represents a proportional gain, K_I represents an integral gain and K_D represents a derivative gain. Finally, the two constants T_I and T_D are the integral time constant and the derivative time constant, respectively. When written in its parallel form it is obvious that the PID control consists of three terms:

- The *proportional (P)* term is the overall control response, containing a value that is proportional to the error signal, which means that if Thorvald is following the grass line perfectly and the error offset is zero, the expected proportionate value is also zero.
- The *integral (I)* term minimizes the steady state error by introducing an integrator, meaning that it will represent the history of Thorvald's overall combined movements by summing up the P-values.
- The *derivate (D)* term minimizes the transient response of the system by introducing a differentiator, handling the rate at which the proportional term changes values.

The terms all depend on the control error. This error can be expressed as a function of time where the proportional term depends on the present error, the integral term depends on the past error, and finally the derivative depends on the estimated future error. Independently, the three terms have different effects on the performance of the system. By increasing the K_p gain there will be a decrease in the time to target-value (or rise time), as well as a decrease in the steady state error. There will, however, be downsides as well, as the overshoot and settling time will increase as the overall stability of the system is reduced. By increasing the K_I gain, the time to target-value will decrease slightly, although the biggest effect will be a reduction in the steady state error. The gain increase will, however, lead to a bigger overshoot and a longer settling

time, and again the overall stability of the system is reduced. By increasing the K_D gain, the time to reach target-value will increase somewhat along with the steady state error, but the total period of the loop may in fact decrease as the system recovers faster from any oscillation or other disturbance. The overshoot and settling time will decrease as the overall stability of the system generally increases.

A combination of the P, I and D terms may be used as a control loop mechanism, as it is often not necessary to apply all the terms. Depending on the intended application and factors like the dynamics of the system, a P-controller, PI-controller, or a PD-controller may be the best suited controller type.

Chapter 4

Robot Programming

4.1 Introduction to Robot Programming

Robot programming can be described as a subdivision of conventional computer programming, where input and output devices are extensively used. Inputs can be received from the data generated by equipment like sensors or control boxes. The outputs can be sounds, lights, or messages appearing on displays. The main purpose of robot programming, whether its Thorvald or any other robot, is the application of the robot's actuators to perform a specific task. For this reason, many would not define a machine as a robot unless the system's outputs include actuator devices, which can mechanically move the joints and parts of the system. Potential sensor feedback is often required to successfully execute said task. To evaluate data and ultimately decide to perform an action based on its inputs, a robot will use its "brain". This computing unit can be a microcontroller or a more advanced computer.

All programming languages can be applied to the programming of robots, but there are some that are better suited than others. Although languages like C# and Java are used, Python and C++ are by many still considered the most popular languages (31).

4.2 Robot Operating System (ROS)

ROS is an open source robotic software framework, which means that is it free and openly distributed. The name can be slightly misleading, as it is not a full-scale operating system, but a *meta*-operating system (32). This means that it is more advanced than middleware, but not a complete operating system either. It provides many extremely useful services, like facilitatin interprocess communication (message passing interface) allowing programs and/or processes

to work together. It also provides package management for effectively managing and distributing software, low level device control, and making sure the distribution of hardware resources is well organised (31), (33). Most importantly, ROS also has a vast community of dedicated users and third-party libraries, open source robotic algorithms, as well as sensor visualization tools like RViz and simulation tools like Gazebo (both discussed in more detail later in this chapter). The software framework supports client libraries and several programming languages like C++ and Python.

4.3 Brief History of ROS

In 2007 a collection of robot orientated software was developed at Stanford University, which later that same year was continued by a company named Willow Garage (31). At the time of writing this company still develops hardware and open source software, and it was with Willow Garage that the name ROS was first used. The first version of ROS was released in 2009 (ROS 0.4), and an early version of a ROS developed research-robot named R2 was released. One year later the first full-scale ROS software framework was released, with libraries and tutorials for open source use. Since then the ROS project has been managed by the Open Source Robotics Foundation (OSRF), and at the time of writing the twelfth version of ROS is currently the latest, named Melodic Morenia.

4.4 The Benefits of Using ROS

Modern robots can serve many different purposes, with a wide variety of options when it comes to sensor inputs. Hardware can also be very different from one robot to another, making the software and code required quite extensive if one were to build it all from scratch for every new robot. Before the release of ROS, this was exactly what developers were doing. With every new robot one would be required to rewrite code and algorithms specifically for that configuration. The objectives of the makers of ROS was from the very beginning to make a tool-based and multi-lingual framework, making ROS applicable to very different hardware, sensor usage and other project requirements (33).

ROS software is maintained and supported by a vast user community, and one of the main advantages with ROS is its use of highly independent libraries. This allows the different processes that perform computations to be written in different languages. There are reusable libraries available, a common platform of small parts of code that can be used together and

modified to suit a specific purpose or work environment. In addition, many different sensor packages are supported by ROS. That makes the packages stable, well documented, and safe to use. If necessary, code written in Python can also easily be implemented in an application running mostly on C++ code, and vice versa. If a sensor is changed, there is no need to change the whole code, simply the parts that subscribe to that specific sensor output. For these reasons, ROS can be used to create highly capable robot-applications while minimizing development time.

4.5 ROS Architecture and Key Features

A robot equipped with several sensors and actuators will have many processes and programs running, needing to communicate with one another. Thorvald is no exception. Allowing one program to handle all these processes is possible but may result in an ineffective use of power and a high degree of complexity (which increases with every added sensor and actuator). A modular approach where there are several smaller programs in charge of sensor data and other programs in charge of actuators, working independently, is for many the preferred way of robot programming. It does, however, require the programs and processes to communicate, and that is exactly the service the software framework ROS provides.

In ROS terminology a *node* is a program (or a process) that computes using ROS API. Nodes that send data are known as *publisher nodes*, and nodes that receive data are known as *subscriber nodes*. However, a node is not restricted to strictly either subscribe or publish. The node can do both if required. Publisher-subscriber communication happens through *messages,* and the data itself can consist of several types of data values. The values in these messages are described by a message description language and the message path where the messages are sent are known as *topics*, to which individual names are given.

A starting program will automatically communicate with the *ROS master*, a ROS program that receives the complete information of the node, facilitating its ability to communicate with other relevant nodes. Figure 4.1 shows a simple block diagram of how the communication between nodes and the ROS master is set up in the ROS environment.

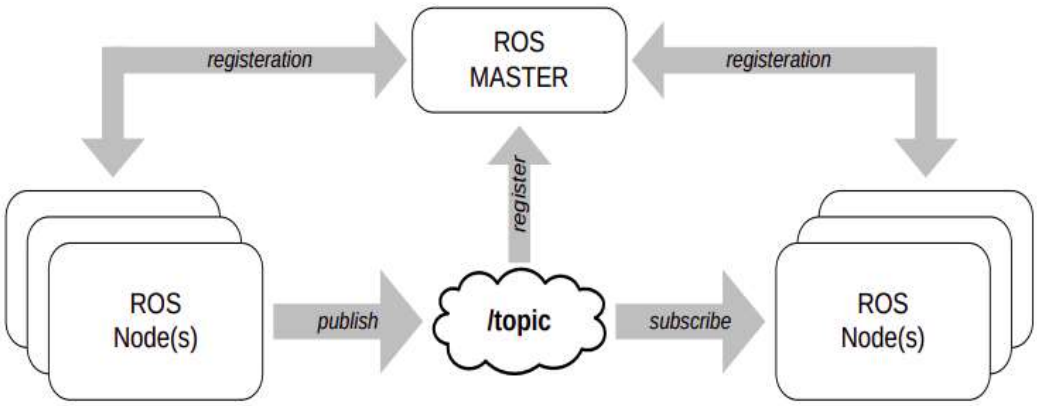


Figure 4.1: Block diagram of the ROS environment's inter process communication (34).

Launching the ROS master will also generate the ROS *parameter server*. In the parameter server global variables (ROS parameters) of many different data types can be stored, allowing any node to access the values of these parameters, change them or create new ones. An obvious advantage of storing ROS parameters in a parameter server, is that the parameters do not have to be hard coded in the node itself. Thus, one can easily change important parameters in the code, without having to recompile everything. In addition, the ROS parameters can be stored in a *YAML* format, an easily read configuration file. This YAML file can then be loaded directly into an XML configuration file (a ROS *launch* file). Such a launch file can (among other things) start all the nodes, as well as load the ROS parameters defined in one or more YAML files.

Another fundamental aspect of the ROS framework is the concept of *services*. A service is mainly another way nodes can exchange information, used when standard messages are not sufficient. The service differentiates itself from the message by being a two-way communication channel (bi-directional). A node does not simply publish a message, it waits for a response to the sent data. The other differentiating factor is that this two-way communication channel is strictly between the data-sending node and the receiving node. The node sending the data, referred to as a *request* (as there is a request for a response), is called a *client*. The node receiving the information is called a *server*. The client then performs some action, and then sends back the reply that the server is waiting for.

The data exchanged between the two nodes are referred to as the type *service data type,* and subsequently divided into request and reply. Like in the case of subscribers and publishers, however, a node can be both a client and a server at the same time. In fact, a single node can be both a subscriber and a publisher of different topics, as well as a client and a server of different services. As there is no feedback from the client as it performs its action, a service is

simply meant for procedures that do not run for too long. To be able to cancel and/or monitor the action being performed, to continuously evaluate the feedback in a closed loop, an *action server* must be defined.

The action server and the action client communicate with five defined topics. Two topics can be published by the client, containing a *goal* and a *cancel* message. The goal message is simply the intended goal of the action, and the cancel message can be sent to request a cancelation of the action at any time during its execution. In turn, three topics can be published by the action server, containing the *status, feedback* and *result* messages. The status message informs the client of the current state of all goals, and the feedback message is periodically sent to give additional information on the execution of the action. The result message can be sent once, to inform the client of the final state of the goal upon its completion.

The *actionlib* package is a useful ROS package for the implementation of an action server in the server node, and once it is set up, any other node that has an action client can call the task or action to be performed.

4.5.1 Reference Frames and the ROS Transform Library

There are many ways and different perspectives from which to describe the motion of the robot, through fixed global reference frames or moving reference frames that can be transformed and rotated. As Thorvald moves around in a three-dimensional environment while taking in sensor data, there must exist a relationship between multiple reference frames over time. Only this way can Thorvald efficiently manoeuvre in the field, as the localization system continually updates the robot's position in the map. The ROS *transform* (tf) library is one of the ROS core components and is a useful tool, as the relationship of the different coordination systems of the nodes are continuously broadcasted to tf.

As shown in Figure 4.2, all the components of the Thorvald configuration considered in this thesis, have their own reference frame. In addition, other reference frames are defined which are not related to a physical part, like the *base_link* frame located between the two front wheels.

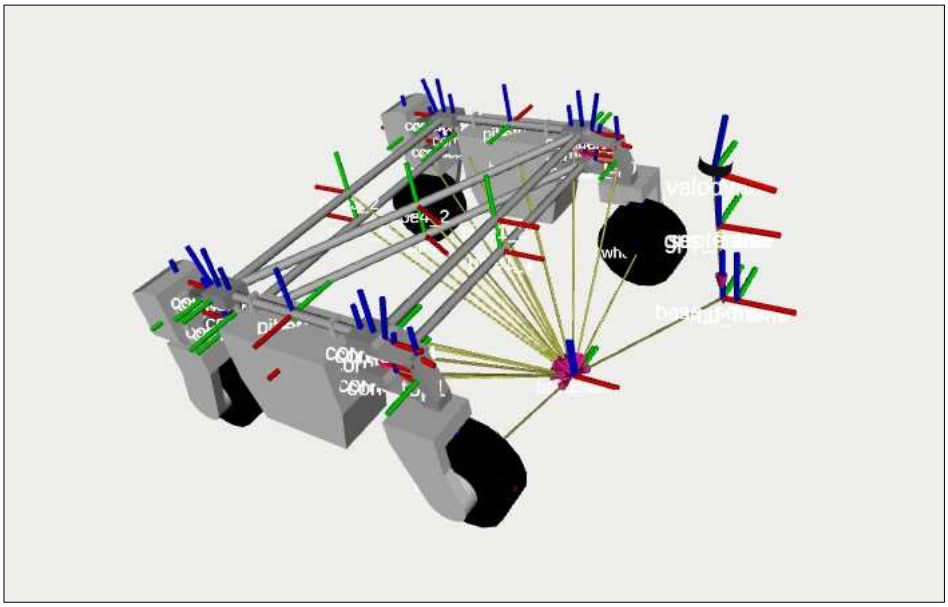


Figure 4.2: The Thorvald grass cutter (not equipped with its cutting tool), shown with all its reference frames.

4.5.2 RViz: Data Visualizing Software

RViz is a powerful open source tool for three-dimensional visualization of many different datatypes and information, published on different topics. RViz stands for ROS Visualization, a ROS package that provides a simple yet effective interface to present the data provided by the robot's sensors. Three-dimensional data, including, but not limited to, point clouds and camera images, can be visualised in real time or from a recorded file. The data can be inspected and interacted with efficiently by using interactive markers and selection methods, as the user is able to move around in the three-dimensional visualised environment. Published markers, depicting primitive shapes like lines and arrows, can be read by RViz and utilised to further visualize information like topological maps, current paths, and goals of the robot.

The software can be very useful in robot programming, by visualizing the environment that the robot is registering from its sensors, which is essentially what the robot "sees" and bases its decisions on. Complex and unintuitive units and systems like quaternions can be difficult to understand without the aid of proper visualization software. The software can be applied in development, post-processing and when debugging, as well as for the visualization of the final results of any applied algorithm.

4.5.3 Gazebo: Robot Simulation Software

Gazebo is a powerful open source simulation tool. The software simulates the physics of an open and dynamic three-dimensional world, as well as providing simulated sensor feedback. The virtual worlds in Gazebo can be custom built to simulate specific conditions and environments, and there exists many prebuilt objects and shapes one can utilize in the simulation. Nearly all conditions of the simulation can be completely controlled by the user, as aspects like mass, friction- and reflectivity-coefficients are all parameters that can be tuned and changed. The powerful physics engine use these, and several other parameters, to efficiently simulate dynamic velocities, accelerations, and forces (angular and linear), allowing for realistic testing of the system. Specific robot configurations and their (joint) mobility in different environments can be dynamically simulated in these virtual worlds, by launching configuration files and kinematic models of the robot alongside the simulation software.

The simulation software can be efficiently integrated with ROS by implementing a few ROS packages, allowing ROS wrappers to establish an interface to communicate with the software using messages.

4.6 Point Cloud Library

When manipulating and working with point cloud data, one can greatly simplify the process by taking advantage of the Point Cloud Library (PCL). By using PCL there is no need to write a parser to convert the data published by the sensor into workable formats. The library is also a framework that provides powerful algorithms for efficient processing of point cloud data. The software can be applied to and included in Thorvald's algorithms, as it is open source licensed under Berkeley Software Distribution (BSD). This means it can be used for commercial and research purposes (35). PCL boasts a big user community and has become somewhat of a reference for the handling of 3D-data.

There are a plethora of tools and algorithms available from PCL, organised in different sectioned libraries. These libraries can be used for standalone applications or combined to perform tasks like cloud segmentation, object recognition, and surface construction. There are numerous examples of different possible applications of the PCL library and the algorithms it contains. For example, it has been used in research projects to build Digital Face-Inspection

systems, and to detect buildings and separate clusters in urban areas (36), (37). PCL also includes several types of robust estimation models in its library, RANSAC being one of them.

Chapter 5

Sensor and Software Selection

5.1 Application of Sensors for Grass Line Detection

The theoretical background of three potential sensor technologies were presented in Chapter 2: Thermographic cameras, RGB-D cameras, and the LIDAR sensor. These sensors all have applicational upsides and downsides, as the selection and use of a sensor will always depend on the specific task at hand, in this case finding the separation line in a grass field.

Thermographic Camera

Thermographic camera technology could potentially be used for autonomous navigation in a grass field if the temperatures of the cut grass and the uncut grass vary sufficiently. Unlike laser-based systems, the upsides of using a thermographic camera are that it is not as affected by fog, smoke or other atmospheric conditions as visible light-based systems (38). For Thorvald there exists thermographic cameras that do not weigh too much and are compact enough to be securely fixed on the robot and powered by its batteries.

In theory, the surface of the uncut grass will be cooler than that of the cut grass. The tall grass will be further away from the ground and have a smaller area of the blade of grass in the sun, and a larger area in the shade. In addition, the uncut grass will also have a less dense structure compared to the cut grass, allowing for a greater cooling effect by the wind. The cut grass will be laid out, allowing for a greater area to receive the sun's rays, as well as being compact and close to the ground which will shield the grass from cooling wind gusts. A colour thresholding (CT) technique based on a Hue Saturation Value (HSV) scale to quantifying colours, may be enough to find the line between the cut and uncut grass. However, the Norwegian climate

differs vastly from the South American climate, and testing would be required to see how effective the sensor would perform on a cloudy, sunny, or rainy day in Norway.

RGB-D Camera

An active sensor could be unadvisable for the project discussed in this thesis, as the robot will work in an open outdoor environment. The fields most likely offer little or no shielding from the sun, and as the sun emits infrared light it will often interfere with an IR emitting pulse from the sensor. This can saturate the sensor, potentially causing missing depth data (19).

A passive sensor, however, or a hybrid sensor offering the possibility to change between active and passive methods depending on the amount of sunlight, could produce a satisfying result. This sensor's main advantage is that it is low-cost, lightweight, and mobile, as well as having a low power consumption.

Figure 5.1 shows an RGB-D image taken in a field of grass with an Intel RealSense Depth Camera D415 on a cloudy day. The image shows that the depth difference of the cut and uncut grass is registered, thus the separation line on the right of the field is visible. The image could also be represented as a point cloud. However, the quality of the depth data diminishes with distance from the sensor, and it can still be sensitive to outside environments with heavy and direct sunlight.

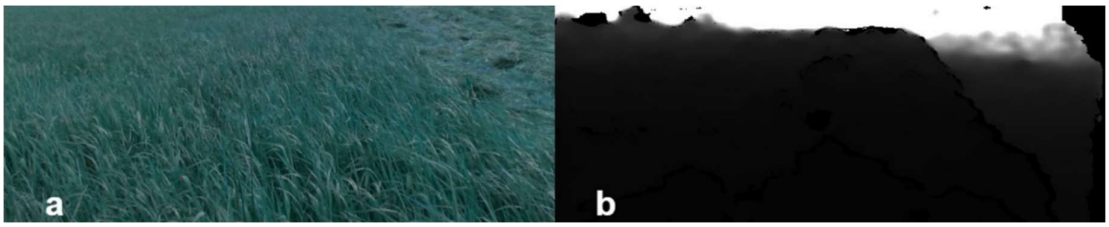


Figure 5.1: a) The RGB coloured image depicting a field of grass, with a section of cut grass in the top-right corner; b) The depth component of the picture.

LIDAR

The LIDAR sensor will produce the most detailed and dense point cloud, compared to the RGBD camera. The perception range is the highest of the three sensors, and the sensor offers high accuracy. Sunlight does not affect the sensor in the same way as the RGB-D camera, and so the application of this sensor would provide a more robust system. The cost of the system is, however, quite high. It is the most expensive sensor of the three considered.

5.2 Multicriteria Selection of Sensor Technology

To best select the appropriate sensor, an MCDM technique (as presented in Chapter 4) has been applied to determine ranked weights for all the criteria, and consequently find the sensor that best suits the needs of this project.

1. *Defining criteria and alternatives:*

Three different sensor-alternatives have been considered in this chapter and are listed in Table 5.1:

Table 5.1: Defining the sensor alternatives.

Sensor no.	Sensor
S_1	*RBG-D camera*
S_2	*Thermographic camera*
S_3	*LIDAR*

A selection will be made based on the following criteria, listed in table 5.2:

Table 5.2: Description of the criteria used to select the sensor.

Criteria no.	Attribute	Description
C_1	*Effectiveness*	How precise is the sensor expected to be able to provide the information required, to produce a good enough estimate of a line in the grass that Thorvald can follow?
C_2	*Versatility*	How well will the sensor work in different environments, like night-time versus daytime, clear weather versus fog, etc.?

Table 5.2: Description of the criteria used to select the sensor. (continued).

Criteria no.	Attribute	Description
C_3	*Generality*	Can the sensor and the data it outputs be used for any other relevant tasks across the Thorvald platform?
C_4	*Complexity of data manipulation*	How easy is it to work with the data format produced by the sensor?
C_5	*Price*	Although the university is in possession of all the considered sensors at the time of writing, the price is an important factor to consider for potential mass production in the future. It might also be necessary to repair and/or replace damaged sensors.

The criteria are divided in the following two categories:

- C_1-C_3 are benefit criteria.
- C_4-C_5 are cost criteria.

2. *Weighting of criteria:*

The criteria are ranked as presented in Table 5.3, where criteria 1, Effectiveness, is deemed as the most important criterion, and Price is considered the least important:

Table 5.3: Ranking of criteria from 1-5.

Criteria	Rank
C_1	1
C_2	2
C_4	3

Table 5.3: Ranking of criteria from 1-5 (continued).

Criteria	Rank
C_3	4
C_5	5

The ROC method is applied to generate the following weight vector *W:*

$$W = [\,0.45, 0.26, 0.16, 0.09, 0.04]. \tag{35}$$

3. *Determining the decision matrix:*

The relationships between the sensors (S_i) and the criteria (C_j) are presented in the decision matrix $D = [d_{ij}]_{3x5}$ below, where the scale has values from 1 - 5. Benefit and cost criteria are evaluated as previously defined, for example, the sensor deemed the most versatile (benefit criteria) is valued at 3, and the most expensive sensor (cost criteria) is valued at 4. The author's evaluation of the sensor alternatives in relation to the criteria is as presented in matrix *D*:

$$D = \begin{array}{c} \\ \boldsymbol{S_1} \\ \boldsymbol{S_2} \\ \boldsymbol{S_3} \end{array} \begin{array}{ccccc} \boldsymbol{C_1} & \boldsymbol{C_2} & \boldsymbol{C_3} & \boldsymbol{C_4} & \boldsymbol{C_5} \\ 3 & 2 & 5 & 1 & 2 \\ 2 & 1 & 2 & 2 & 2 \\ 5 & 3 & 4 & 1 & 4 \end{array} \tag{36}$$

The criteria in matrix *D* are then normalised, and the following normalised matrix *N* is obtained:

$$N = \begin{array}{c} \\ \boldsymbol{S_1} \\ \boldsymbol{S_2} \\ \boldsymbol{S_3} \end{array} \begin{array}{ccccc} \boldsymbol{C_1} & \boldsymbol{C_2} & \boldsymbol{C_3} & \boldsymbol{C_4} & \boldsymbol{C_5} \\ 0.49 & 0.53 & 0.74 & 0.59 & 0.59 \\ 0.32 & 0.27 & 0.30 & 0.18 & 0.59 \\ 0.81 & 0.80 & 0.60 & 0.59 & 0.18 \end{array} \tag{37}$$

4. Scoring sensor alternatives:

The SAW technique explained in Chapter 2.9 is applied to rank the three sensors, and ultimately choose the best suited sensor for the task of identifying the separation line in the grass field. The result is presented in Table 5.4. The three sensors are scored by the scoring equation, also defined in Chapter 2.9, producing three scores: R_1, R_2, R_3, belonging to alternatives S_1, S_2, S_3 respectively:

Table 5.4: The final score of the three sensors.

Sensor	Final score	Position
RBG-D camera (S_1)	0.553 (R_1)	2
Thermographic camera (S_2)	0.302 (R_2)	3
LIDAR (S_3)	0.729 (R_3)	1

As the table above shows, by applying a multicriteria decision making technique to the specified criteria, defining the weights with the ROC method, and finally scoring the sensors with the SAW equation, the LIDAR is chosen as the best suited sensor.

5.3 Software Selection

To further build on the Thorvald project's established software platform and modular code, ROS has been selected as the software framework. The latest ROS distribution Melodic Morenia is utilised in this master's thesis, as the core packages are maintained until the distribution's end of life, May 2023. Furthermore, Gazebo is chosen as the main simulation platform to simulate real world environments and situations that the robot might encounter, and RViz is used as the main data visualization software.

While manipulating and working with the data produced by the LIDAR sensor, the approach presented takes advantage of the open sourced Point Cloud Library. The PCL algorithms have been used in the process of filtering, segmenting, and concatenating point cloud data, as well as to implement robust estimators to estimate model coefficients and to transform reference systems. Linear Squares Regression and RANSAC is used as explained in the model algorithms presented in Chapter 2.8, to fit lines and planes to a given set of inliers. RANSAC is chosen for its conceptual and applicational simplicity by using the PCL library, as well as its robust nature.

5.4 Programming Language Selection

Though one can use several different programming languages with ROS, C++ was chosen as the programming language used during this project. The use of C++ does not mean that it cannot be supplemented with code written in other languages, that is after all one of the benefits

when working with ROS, as covered in Chapter 4. However, C++ has been chosen for the development of this thesis's algorithm for several reasons:

- C++ is a *typed language*. All variable types are defined, and any errors or unwanted behaviour are more likely to be found at an early stage in the process, during compilation. This means that the code is less likely to contain serious bugs, crash or perform differently in the field during runtime. The algorithms are well defined and rigid which allows for scalability and further development. This was deemed important for the continuation of the Thorvald project in the years to come.
- ROS is fully supported and developed in C++, and there will rarely be libraries not available in C++.

Chapter 6

Control System Design

6.1 Reference Systems of the Thorvald Robot

The *tf* library is used in this thesis to transform sensor-centric data from the LIDAR to a body-fixed reference frame called *base_link*. The origin of base_link is located in the front of Thorvald, between the two front wheels as shown in Figure 6.1. This local frame is defined as the *thorvald frame* $\vec{E}_t = \{\vec{x}_t, \vec{y}_t, \vec{z}_t\}$. The orientations of $\vec{E}_t$ are as illustrated in Figure 6.1, $\overrightarrow{x_t}$ is forward orientated and $\overrightarrow{y_t}$ is pointing to the robots left as the robot is moving forward. The unit vector $\overrightarrow{z_t}$ is normal to the skeleton of the robot. As a first step on the way to defining a closed-loop control system, a second *world-fixed frame* of global reference is defined: $\vec{E}_f = \{\vec{x}_f, \vec{y}_f, \vec{z}_f\}$. The global reference frame is used to implement the control system, where the position and orientation of the robot is described to assess Thorvald's movement in the environment and produce feedback.

By simplifying the three-dimensional world to a flat surface (z = 0), the position and velocity of the robot at any point in time as it is moving through space can now be expressed by the state vector:

$$\psi(t) = \begin{bmatrix} q(t) \\ v(t) \end{bmatrix} \tag{38}$$

Here the position-vector q and the velocity-vector v are defined by the Cartesian coordinates (X,Y) and the angle α (see Figure 6.1). The coordinates (X,Y) are those of the origin of the

thorvald frame $\vec{E}_t$, expressed in the fixed frame $\vec{E}_f$. The angle α represents the angular offset of the thorvald frame in reference to the fixed frame.

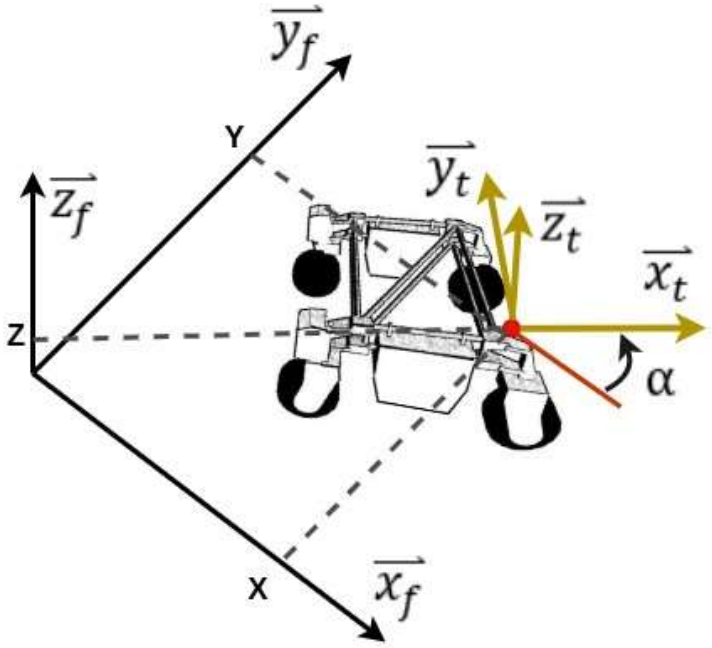


Figure 6.1: Illustration of Thorvald with the global ($\vec{E}_f$) and local ($\vec{E}_t$) reference frames defined.

The position- and velocity-vector are mathematically described in the following way:

$$q = [X, Y, \alpha]^T \tag{39}$$

$$v = \dot{q} = [\dot{X}, \dot{Y}, \dot{\alpha}]^T \tag{40}$$

Having defined the state vector wherein each variable is obtainable, the control system can be designed to use the variables as the origin of feedback signals. Relative displacement to the line can be calculated, with the referent point being a position in the world-fixed reference frame.

6.2 Error Estimation

For the robot to be able to navigate autonomously, the robot must detect the separation line and then estimate its position in relation to said line. The objective is therefore twofold:

1. *Grass line estimation*: The complete process of filtering and segmenting the raw sensor data to extract meaningful information, so that robust estimators can produce the desired model coefficients and fit the most probable grass line.
2. *Error offset calculation*: To compute and quantify how far the robot is from following its desired path, which in this case is a straight line. This is quantified by the lateral displacement e_y and the angular displacement e_φ of Thorvald, in reference to the line. These two displacements make up the total error offset, which can therefore be expressed as $e = [e_y, e_\varphi]^T$.

The suggested perception strategy for the grass line estimation approach will be discussed in detail in Chapter 7, but first the error offset calculation must be defined. Once the line is detected, this error offset can be calculated. The separation line will be defined by two points, p_1 and p_2. The lateral displacement e_y is then defined as the minimum distance between the line and the part of Thorvald that should touch said line. As the robot follows its determined path, the tip of the cutting tool should be exactly on the line. This way the grass is efficiently cut without overlapping or leaving rows of uncut grass behind. The cutting tool is 1800 mm wide and is placed as illustrated in Figure 6.2 with its centre of mass in origo (O_t). The point of contact, i.e. the tip of the cutting tool, is referred to as p_k and is located 900 mm along the y-axis in the local *thorvald frame*.

As illustrated in Figure 6.2, the minimal distance between the tip of the cutter and the line will always be the norm of a vector, $u_\perp$, between an intersection point on the line, p_i, and the point p_k, orthogonal to the line:

$$e_y = \|u_\perp\| = \sqrt{(p_{i,x} - p_{k,x})^2 + (p_{i,y} - p_{k,y})^2} \tag{41}$$

Also illustrated in Figure 6.2, the angular displacement is defined as the misalignment of the orientation of Thorvald in relation to p_1 on the line:

$$e_\varphi = tan^{-1}\left(\frac{p_{1,y} - p_{k,y}}{p_{1,x} - p_{k,x}}\right) - \alpha \tag{42}$$

As in Figure 6.1, the angle α represents the angular offset of the thorvald frame in reference to the world-fixed frame. The sign convention is such that a counterclockwise angular offset (like in Figure 6.2) translates to a positive α value.

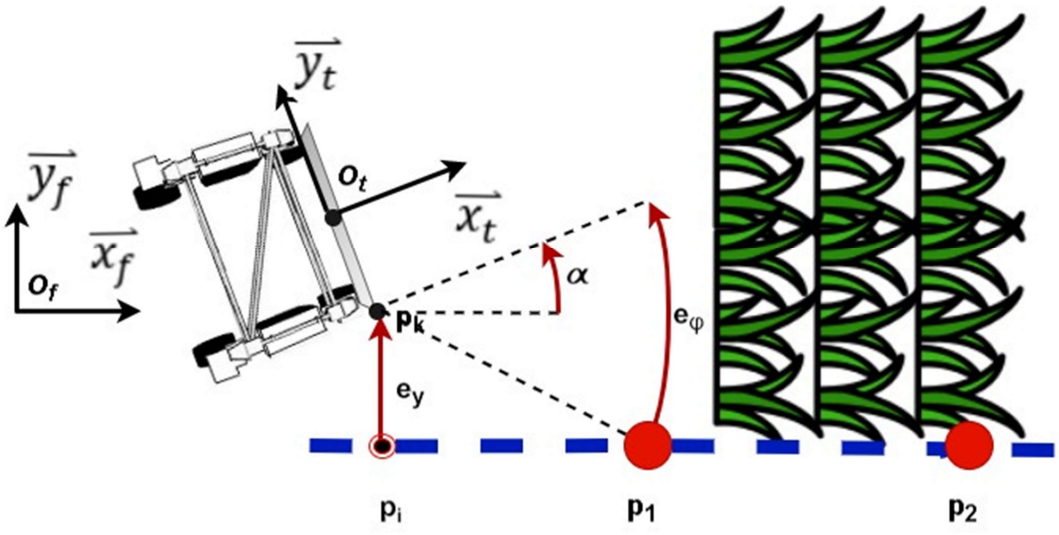


Figure 6.2: Illustration of the error estimation from the line.

6.3 Motion Modelling and Velocity Control

To control the robot's movements, a motion model must be defined. The robot can be actuated by two velocity controllers working independently of each other to control the robot's movement. One controller is implemented to control the linear velocity v in the $\vec{x}_t$ direction, and the other controller controls the angular velocity ω.

The grass cutting Thorvald configuration is configured with a kind of Ackerman-steering which will behave very similarly to a differential drive robot, meaning that the linear velocity in the $\vec{y}_t$ direction is always equal to zero, in the robot's body-fixed frame base_link. The velocity of the robot in the world-fixed reference frame can be expressed as follows:

$$\dot{X} = vcos(\alpha) \tag{43}$$

$$\dot{Y} = vsin(\alpha) \tag{44}$$

However, the kinematic model that governs the movement and engagement of individual actuators is pre-existing and outside of the scope of this thesis (39). By taking advantage of this pre-existing internal kinematic model though, velocity commands can be published to activate the actuators and move the robot with the desired orientation and forward-moving motion. The velocity commands u_t, that are passed to the robot's motors at time t is dependent on the error offsets, and can be expressed in the following way:

$$u_t(e_y, e_\varphi) = [v(e_y),\ \omega(e_\varphi)]^T \tag{45}$$

The sign convention is chosen arbitrarily: Positive linear velocities *(v)* translate to a forward moving motion, and positive angular velocities (ω) represent the robot turning left (counterclockwise).

The development of a control algorithm is done under the assumption that the mobile robot system can be approximated as an LTI system. A feedback system will control the robots "free-spaced" velocity by the two parameters, i.e. the linear and angular velocity vectors. There are several possibilities for such a system, however, this thesis considers the most popular and simplest form of control loop feedback: The PID controller introduced in Chapter 3.3. It is chosen for its simple directness and transparent functionality. The following are important vocabulary used to describe the variables of the controller:

- **Setpoint Values:** These are the values that define the goal of the robot, the reference values. In our case we want Thorvald to move towards a point on a line. From the setpoints the angular displacement is calculated, i.e. the angle that Thorvald must turn to face this point. The setpoint also defines the line, together with a second point. There will be a constant renewal of setpoints and lines to follow as Thorvald moves forward and the line detection algorithms are run repeatedly at set intervals.
- **Current Position Values**: These are the repeatedly updated position and orientation values.
- **Error**: The error offset has been described as in Chapter 6.2. This error will be the input of the velocity controllers, as the objectives in this thesis requires the design of controllers that will force the angular displacement and linear displacement towards zero.

The dynamics of the systems can be explained by simplifying the process, without loss of generality, as using a force to move the mass of the robot. As a simplification, the models for driving and steering are assumed identical, and independent. The two actuated velocity-vectors are both represented as second-order systems, i.e. the output (being the *acceleration* of the robot) are two integrations away from the input (the displacement *distances* from the error estimation).

The robot is equipped with brushless DC motors to regulate velocity, which is a motor that exhibits a linear speed to torque relationship. In addition, the motor has a high dynamic

response due to small amounts of rotor torque. For these reasons, the motor is a good fit for servo-applications, meaning that it is applied in a closed-loop system with a feedback element as shown in Figure 3.4. If the robot is represented as such a servo system, composed of a proportional controller, an inertia element and a viscous-friction element, the transfer function of the plant $G_{plant}(s)$ (the pre-existing kinematic models of Thorvald) can be expressed in a second-order system's standard form:

$$G_{plant}(s) = \frac{\omega_n^2}{s^2 + 2\xi\omega_n s + \omega_n^2} \tag{46}$$

Here ω_n is the natural frequency and ξ is the damping ratio of this system.

6.3.1 Angular Velocity Control

The purpose of the angular velocity control is to turn the robot to face the desired direction, i.e. the generated setpoint. The speed at which Thorvald turns is dependent on the magnitude of the angular displacement error. The robot must be able to adjust its course quickly, and by introducing a proportional term, the velocity commands will be proportionate to the angular displacement error of the robot. However, as the angular velocity control is modelled a second order system, it could be prone to overshoot and oscillation.

As the robot turns, the acquired angular speed and the fact that the proportional term is still producing acceleration until the angular displacement is equal to zero, might lead to the robot overshooting its desired orientation. If that happens, the sign of the angular displacement error will change, and the robot will turn back the other way, and so on. The higher the proportionate gain, the higher the risk of overshoot and oscillation in the system.

To avoid this overshoot and oscillation in the case of the angular velocity control, the second-order degree system justifies the use of a derivative term to dampen the system. The derivative term is introduced to account for the rate of change of the angular displacement error. This term will restrain the robot's angular velocity, to make sure it does not turn too quickly.

Every time a new setpoint is introduced, it can be described as a new step input. However, it is important to note that the steady state error e_{ss} of the system will never be zero without an integral term. This is evident when e_{ss} is expressed in the following way, as there will always be an error present:

$$e_{ss} = \Phi \cdot \frac{1}{1 + K_p} \tag{47}$$

In equation 47, Φ represents the new step value which is introduced. The integral term is, however, deemed unnecessary in this system, as the setpoints are updated rapidly and the long-term steady state error is not assumed to be important enough to warrant the added complexity. Testing will have to decide if the steady state error is deemed acceptable, however, it can be reduced by increasing the proportional gain. For these reasons, a PD-controller is chosen, and the output *u(t)* of the controller can be mathematically expressed both in the time domain (equation 48) and the frequency domain (equation 49) in the following way:

$$u(t) = K_p \cdot e(t) + K_D \cdot \frac{de(t)}{dt} \tag{48}$$

$$U(s) = \big(K_p + s \cdot K_D\big)E(s) \tag{49}$$

As setpoints will be updated and changed continuously, the derivative gain must, however, be small enough so the system does not experience a *derivative kick*. This term is used to describe what happens when a sudden change in angular displacement error leads to the derivative of the error suddenly increasing. This can lead to erroneous control outputs. Figure 6.3 illustrates the block diagram for angular velocity PD-controller. The system's transfer function (closed-loop) with the angular velocity controller is expressed as follows:

$$G(s) = \frac{(K_p + K_D s)\omega_n^2}{s^2 + (K_D\omega_n^2 + 2\xi\omega_n)s + \omega_n^2(1 + K_p)} \tag{50}$$

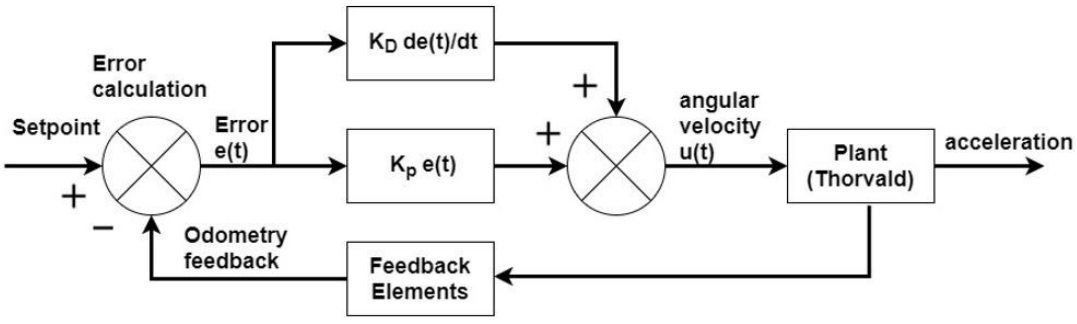


Figure 6.3: A block diagram of the robot's angular velocity controlled by a PD-controller.

6.3.2 Linear Velocity Control

The purpose of the linear velocity control is to move Thorvald forward. The speed at which Thorvald should move forward is decided by the Robotics group at NMBU, set to between 0.6 and 1.5 m/s. However, the lateral displacement error from the line will be the determining factor for the exact speed, within this range. A constant speed of 0.6 m/s will be applied, and then the further away the robot finds itself, the faster it should move to get back to its intended path. Again, the linear velocity control is modelled as a second order system which could be prone to overshoot and oscillation. Even so, a simple P-controller has been chosen as the preferred controller of this velocity. The reasons are two-fold, a low complexity necessity being the first. Figure 6.4 shows an example of the path Thorvald follows. The first line l_1 and the setpoint p_1 are generated by the algorithm, and the robot moves towards it. As the robot gets within a set distance of the point, symbolised by the circles encapsulating the points, another setpoint and line is generated by the algorithm (p_2 and so on). Consequently, the risk of overshooting the line is virtually non-existent, as every setpoint is renewed before the previous setpoint is reached.

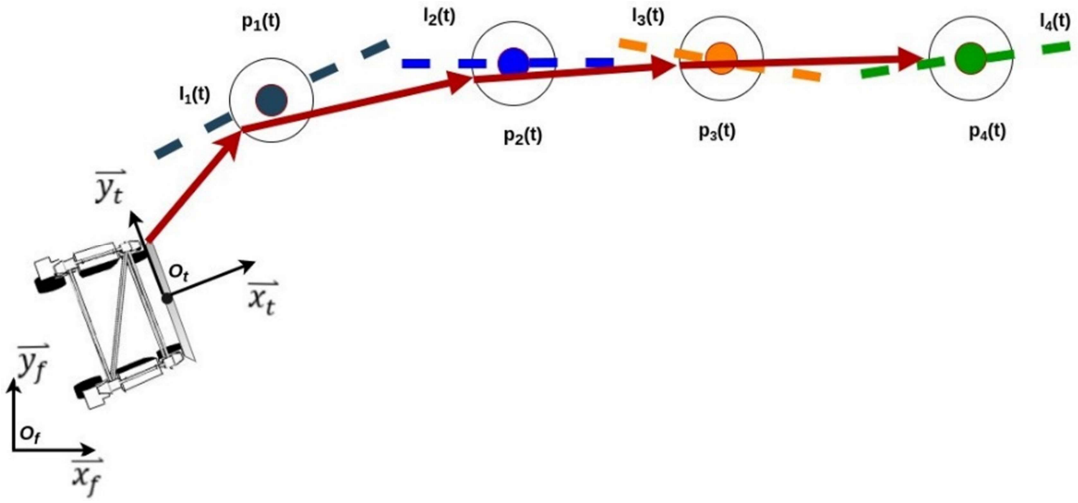


Figure 6.4: An illustration of the intended path generation, where a setpoint and line is generated to which the robot navigates until it reaches a set distance to the setpoint, at which time another setpoint is generated, and so on.

The reality of a constant steady state error is also uninteresting, as the lateral displacement error e_y only impacts the speed at which the robot moves toward the defined setpoint. As a constant speed of 0.6 m/s is applied, the robot can in fact find itself exactly on the line at which time the

lateral displacement error e_y will be zero. In addition, the speeds at which the robot moves are quite low, never more than 1.5 m/s, which also brings stability to the system.

The second reason for choosing a P-controller is its simplicity. A P-controller is easy to test and tune, simple to create and is easily understood for different users of the algorithm developed in this thesis. Although the least complex of the PID controllers, one must not mistake its lack of complexity for ineffectiveness. For these reasons a P-controller is chosen, and the output *u(t)* of the controller can be mathematically expressed in the time domain (equation 51) and the frequency domain (equation 52) in the following way:

$$u(t) = K_p \cdot e(t) \tag{51}$$

$$U(s) = K_p E(s) \tag{52}$$

The system's transfer function (closed-loop) with the linear velocity controller is then:

$$G(s) = \frac{K_p \omega_n^2}{s^2 + 2\xi\omega_n s + (1 + K_p)\omega_n^2} \tag{53}$$

Figure 6.5 illustrates the closed-loop control system block diagram of the linear velocity P-controller.

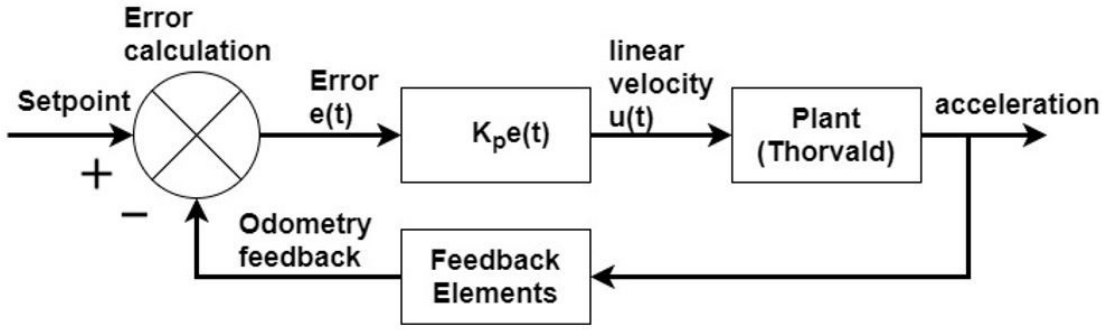


Figure 6.5: A block diagram of Thorvald's linear velocity controlled by a P-controller.

Chapter 7

Perception and Navigation Strategy

7.1 Package Architecture

This thesis presents an efficient, yet simple, algorithm for line detection in an unorganised point cloud, and the consequent velocity control along the separation line. A package called *package_find_line* was created, the nodes and files were organised to obtain a high degree of maintainability and readability.

The package contains two nodes, a header file, a launch file, a parameter server, the package manifest, and the CMakeLists file used to create the necessary build files. The first node is named *line_node*, henceforth referred to as *the perception node,* and the second node is named *move_thorvald*, henceforth referred to as *the navigation node.*

The navigation node contains a class named *RobotCommandAction* and the header file *PointCloudSensor.h* contains the class *PointCloudSensor*. The latter was created to be used in the algorithms defined in the perception node. Unlike the RobotCommandAction class, the PointCloudSensor class definition is placed in a header file. The reason for this is that the class in the perception node is considered more general than the class in the navigation node. Thus, to make it easier to access and potentially use the class in multiple projects, the PointCloudSensor class definition is placed in the header file.

The parameter server *grass_cutter_params.yaml* contains ROS parameters stored in a YAML file, which are loaded in the launch file named *autonomous_cutter.launch*. When launched, the launch file will also run the two nodes. An overview of the developed package architecture is presented in Figure 7.1.

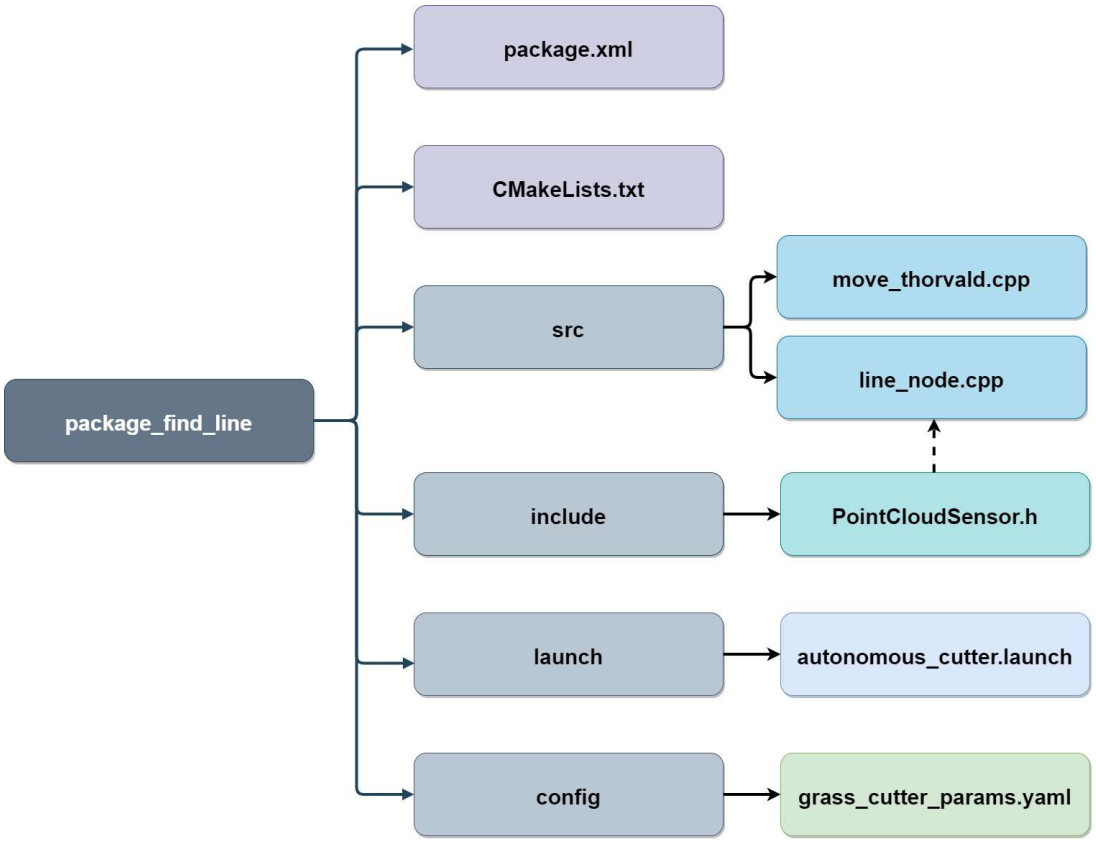


Figure 7.1: The package architecture.

7.2 Perception and Navigation Algorithm

Although the algorithm is developed for, and will be tested on, a Velodyne 16 LIDAR sensor, the code is general and may be applied to any sensor that produces point cloud data. The code is written in C++ and attached in its entirety in *Appendices*.

This subchapter will explain the developed algorithm and present selected pseudocode for a better understanding of the process. To achieve the objective of the robot identifying and autonomously following the separation line, the suggested algorithm is divided into two sections. These sections are separated by nodes that perform the task of perception and navigation:

A) The *perception* part of the algorithm is contained in the perception node. This is the process of detecting the line that we want Thorvald to follow. This includes all the processing, filtering, manipulation, and segmentation of the original point cloud produced by the sensor, before finally publishing the line.

B) The *navigation* part of the algorithm is contained in the navigation node, subscribing to the published line from the perception node. This includes the error offset calculation and the action server which implements the velocity control, moving Thorvald along the separation line so the robot can efficiently cut the grass. Here the robot's offset from the desired path is calculated and PID velocity control is applied to minimize said error. This error offset is, as explained in Chapter 6.2, a function of two variables: The lateral displacement from the detected line and the angular displacement from a setpoint on this line. Figure 7.2 shows the flowchart of the algorithm suggested in this thesis.

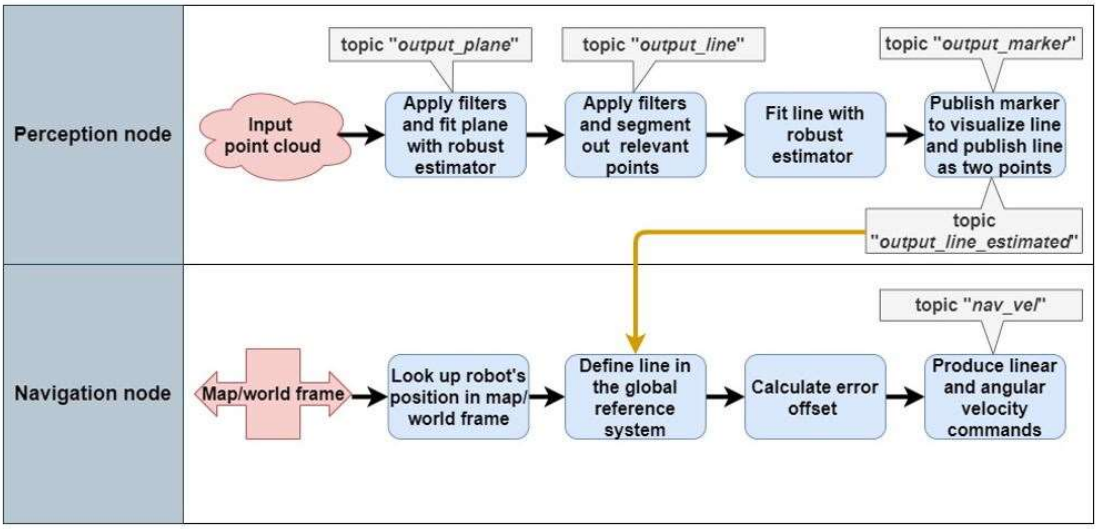


Figure 7.2: Flowchart of the suggested algorithm, and how it is divided in two nodes.

7.2.1 Perception Algorithm in Node 1

This section will give a thorough explanation of the algorithm contained in the perception node.

1. Transforming sensor-centric data

Firstly, the node subscribes to the point cloud produced by the sensor. However, the point cloud is received in the reference system of the sensor. By transforming the point cloud to a reference system defined by a ROS transformation message (the *base_link* frame described in Chapter 6.1), the LIDAR can be placed in any direction and position, and then be transformed to the coordinate system frame assumed in the method.

2. Defining the area of interest by applying pass-through filters

The second step in the point cloud manipulation is to identify the plane of the grass we would like to cut. Making use of more open source PCL algorithms, the point cloud is filtered to remove point cloud data that is deemed unnecessary. This is done by utilizing a *pass-through* filter. The points that are associated with the ground are removed first, by filtering the point cloud based on the height of the grass measured in metres. This value should be a little lower than the actual height of the grass, so that the points relating to the plane of the uncut grass are not removed. Only the points which fall within the segment starting from this height to an arbitrary set maximum limit are kept in the point cloud. Then a second and third passthrough filter is applied, to filter out any point that is not within three metres on either side of Thorvald, and not within 2 – 4,5 metres ahead of the robot. This zone, depicted in Figure 7.3, will henceforth be referred to as the *critical zone*, as only the information that is contained in this zone will be used to identify the separation line.

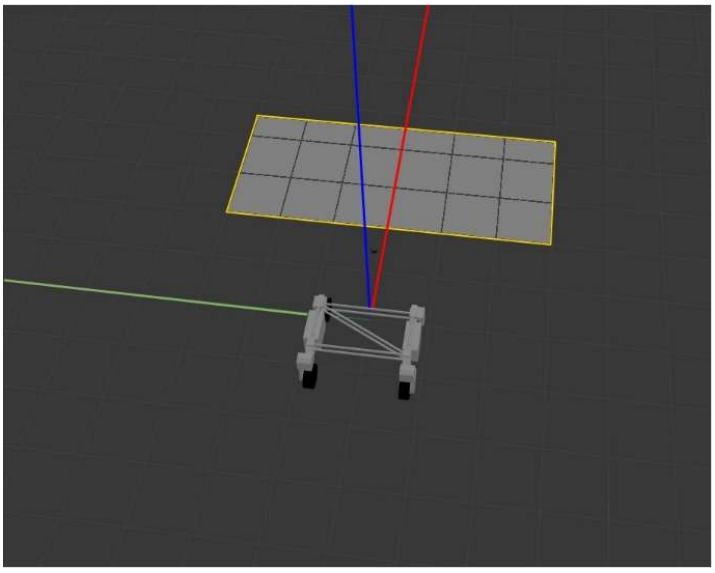


Figure 7.3: The critical zone where Thorvald will detect the grass line (the x_t, y_t, z_t axis are shown as red, green, and blue respectively).

3. Plane segmentation

To make the code more robust, a plane segmentation is then performed. At this point, the point cloud data that is left in the critical zone can still contain any object that finds itself within the critical zone (meaning it is also high enough to reach the specified height segment defined in the passthrough filter). Detecting objects and people on the field is of course important,

however, object detection and avoidance is outside of the scope of this thesis, and only the detection of the separation line is considered. Parts of trees, nearby people, fences, equipment, or big rocks are examples of non-relevant points that, potentially, would not be filtered out in the previous pass-through filters. This can negatively affect the algorithm's ability to fit a line at a later step in the process. However, by locating the most dominant plane in the critical zone, that which does not belong to this plane will be filtered out.

In addition, the raw data from any 3D sensor working in a field of grass is expected to produce a significant output of outliers. It will most certainly be unorganised data with a significant degree of unpredicted data points in the dataset, as the environment consists of moving blades of grass that are of different individual heights, not a solid predictable surface. The law of large numbers, however, assumes overall order and predictability in the field, as most of the blades of grass will grow in the same direction, be in the same plane and have roughly the same height. Plane segmentation takes advantage of this and will leave a point cloud that is more uniform and with less noise.

A RANSAC robust estimator is applied, and the most dominant plane perpendicular to the z-axis is found (with a specified tolerance of 20 degrees, arbitrarily chosen and user specified, to account for sloped fields). A threshold value was chosen empirically to be 15cm, to define how close a point would have to be to the plane to be considered an inlier in the RANSAC model. This ensures a dense point cloud of only the uncut grass. This plane, depicted in Figure 7.4, will be that of the uncut grass under normal operating conditions.

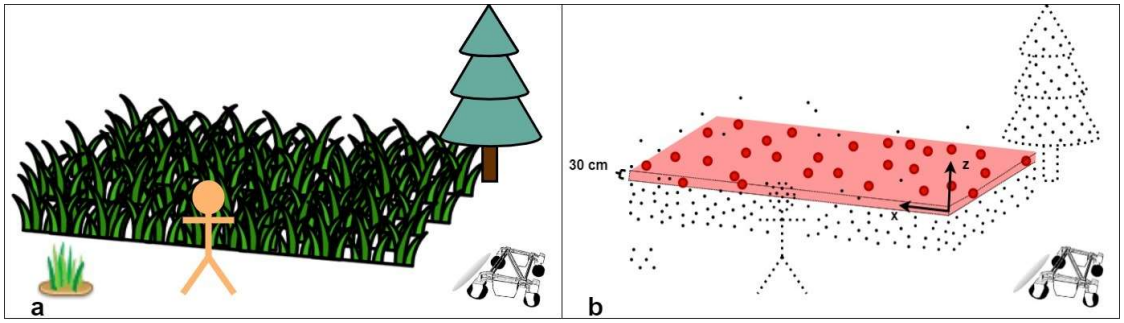


Figure 7.4: a) A possible noisy environment b) The same environment represented as a point cloud, the extracted plane (+/- 15 cm) that is perpendicular to the z-axis has been coloured red, points that do not lie on the plane are coloured black, points that do lie on the plane are enlarged and coloured dark red.

The RANSAC model returns four model coefficients specifying the plane: The x, y and z coordinate of the plane's normal (i.e. the normal vector), as well as the fourth Hessian member

of the equation representing the plane (the offset from the origin). The RANSAC algorithm also returns the indices of the points in the plane. The indices themselves hold no point cloud data but are the index of the points representing the plane that we want to extract to a new point cloud. From these index values the plane is extracted from the point cloud.

4. Isolating points along the separation line

At this point in the process, the point cloud is now representing only the plane of uncut grass. Before a line can be fitted, the points that are likely to lie on the separation line must be isolated and extracted. Firstly, the plane is divided into 25 segments as shown in Figure 7.5, all with the width of 10 cm.

Then, another pass-through filter is applied to the data. Each segment is incrementally checked to find if it contains points. If the segment does contain one or more points, the points that hold the maximum and minimum values on the local y_t-axis are located. If Thorvald is driving with the cut grass to its left (like in Figure 7.5), the point with the highest value on the y_t-axis passes through the filter. If the robot is driving with the cut grass to its right, however, the point with the lowest value in the y_t-axis passes through the filter. Either way, only one point can pass through the pass-through filter in each 10 cm segment.

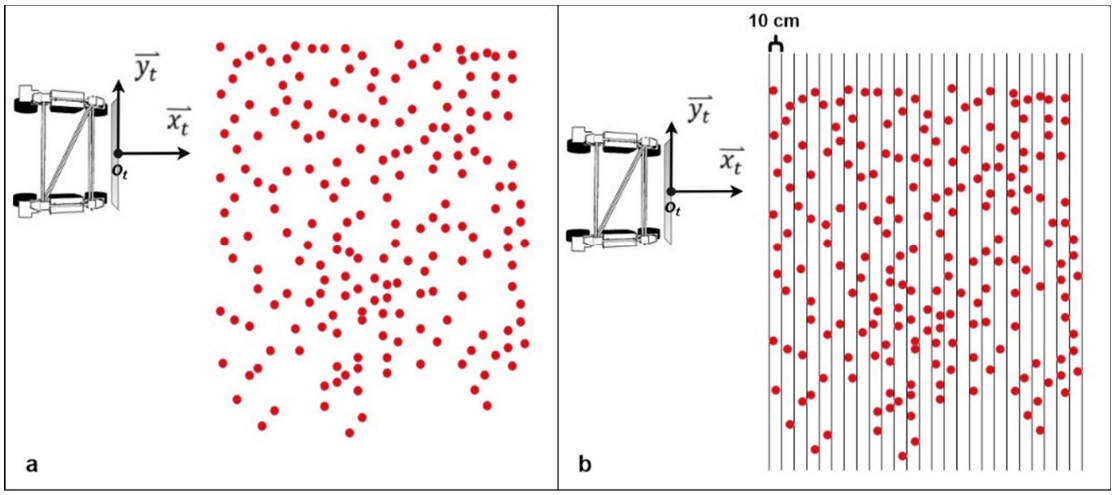


Figure 7.5: a) A point cloud representing the plane of the uncut grass in the critical zone; b) The same point cloud divided into 25 segments with a width of 10 cm.

Assuming all segments contains a point, there will be 25 unique points along the edge of the uncut grass as shown in Figure 7.6, to which a line can be fitted.

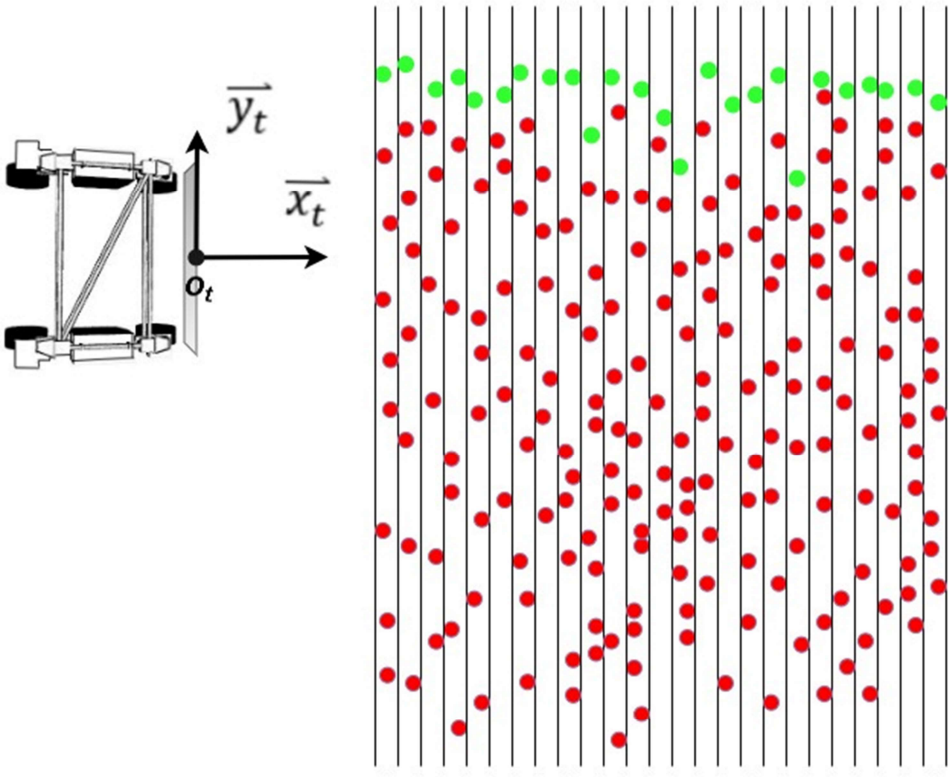


Figure 7.6: The green points represent the points having the highest value on the local y_t-axis in their respective segment.

5. Estimating line with RANSAC models

At this point the point cloud consists of the (up to) 25 remaining points. A line is now fitted to these points with a PLC RANSAC algorithm. The threshold was chosen empirically to be 8 cm, meaning a point would have to be this close to the line to be considered an inlier in the RANSAC model.

Individually, there is expected to be several blades of grass that for any number of reasons will be placed as outliers. Any robust estimator where outliers produce a disproportionate effect on the fitting of the model, would thus be expected to fail. RANSAC, however, being not that sensitive to outliers, is therefore applicable to the work environment in a field of grass.

The RANSAC model produces six model coefficients that are calculated to describe the line. The first three describe the position of a point in the Cartesian coordinate system (P_0) as shown in Figure 7.7. The next three coefficients describe the orientation of the line from the point in Euler angles: Roll about the x-axis, pitch about the y-axis and yaw about the z-axis.

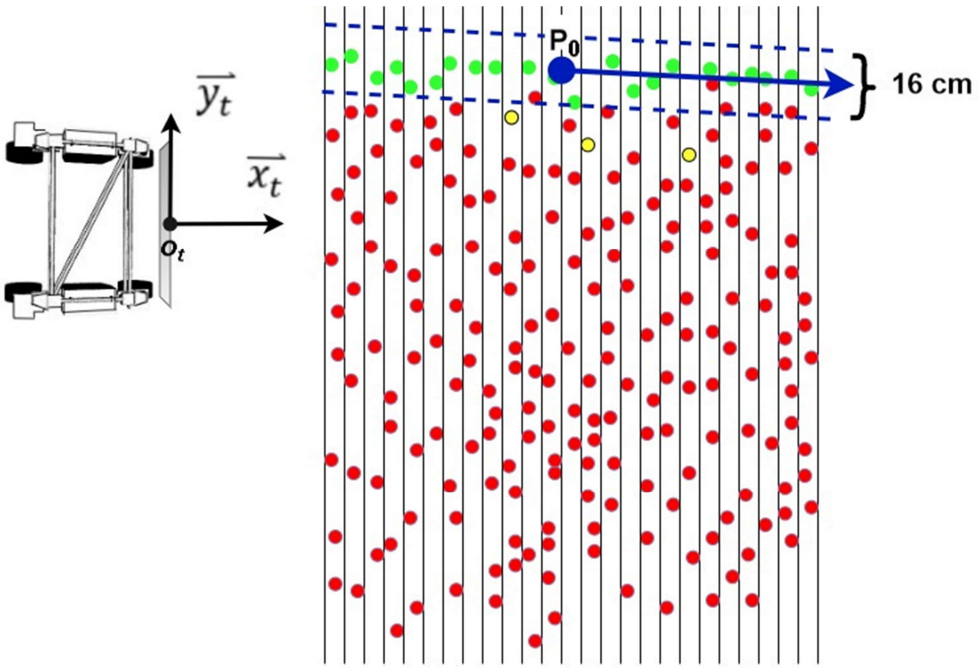


Figure 7.7: An example of a line estimated by a RANSAC model (threshold value in dashed lines), using a dataset with 22 inliers (coloured green) and three outliers (coloured yellow).

7. Averaging the line

As the data is received at a high frequency, the lines are also fitted at a high frequency. To make the fit more robust and stable, the difference between one line to the next is minimised by averaging a set numbered of estimated lines.

8. Generate and publish two points used for navigation

Another way of defining the line, compared to one point and a direction vector, is by using two points on the line. The point P_0 that was defined by the first three coefficients from the RANSAC model is placed halfway between the first and the last inlier.

Assuming the critical zone contains points from 2 – 4,5 metres in front of the robot, and assuming the first and last segment contains inliers, then P_0 would be located roughly 3,25 metres ahead of the robot. As explained in Chapter 6.2, the angular displacement error, and thus the angular velocity, is defined in reference to a setpoint on the line. This point can be any point on the defined line. The closer this setpoint is to the robot, the faster it will reach the set distance to the point where a new line (and setpoint) is generated for velocity control. If the point is too close to the robot, the updates happen too rapidly, and the system loses stability. On the other hand, if the setpoint is too far away from the robot, the system updates too slowly,

and loses flexibility and adaptability to rapid path adjustments. In other words, the setpoint should be a value that can be adjusted, depending on the environment.

The suggested solution is to define the line by two points, *P1* and *P2*, with an equal (user set) distance *L* from P_0. This distance is set as a multiplication of the directional vector (in quaternions). Figure 7.8 depicts how the two points now define the line. Finally, the two points P_1 and P_2 are published, along with a marker that visualizes the line in RViz.

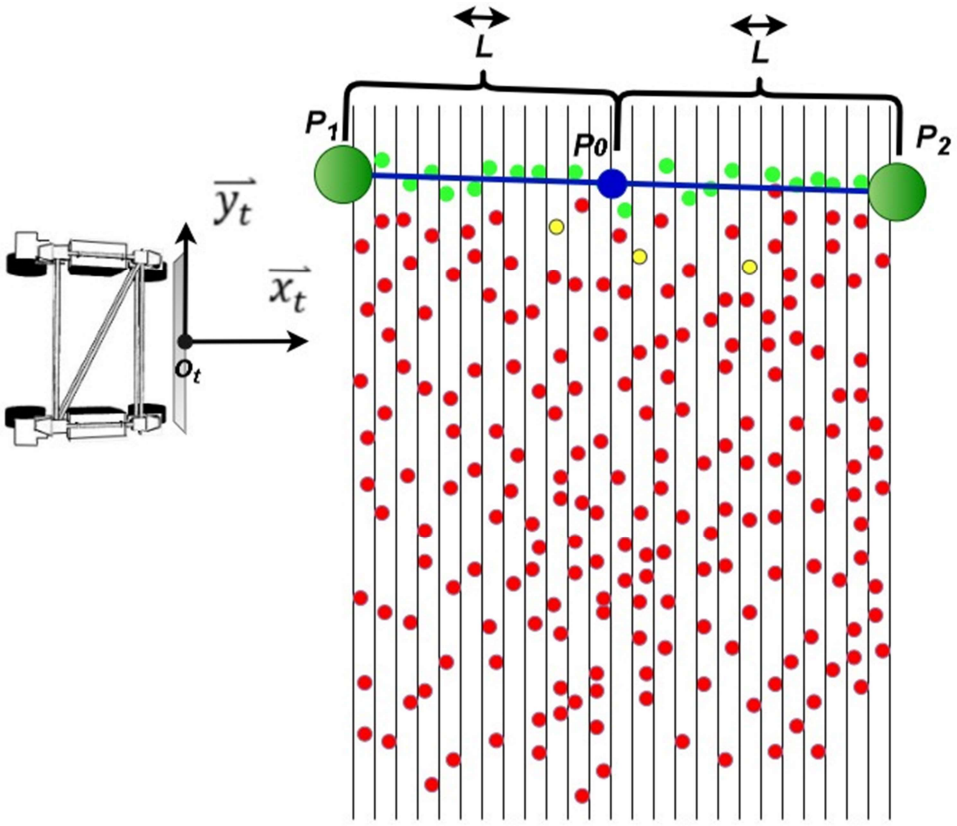


Figure 7.8: The points P1 and P2 are defined a set distance from P0.

7.2.2 Navigation Algorithm in Node 2

A topological navigation system has previously been adapted for Thorvald, allowing the robot to navigate to and position itself at a starting node in front of the field. The topological system will function as the action client, communicating with the node *move_thorvald* which contains the action server. The topological navigation system will call the action server by sending a message containing a goal, i.e. a *move_base_msgs/MoveBaseGoal* message. This goal message

is defined as a *target_pose* consisting of a *Pose message* (a position and orientation expressed in quaternions), as well as *Header message* (a sequence ID, time stamp and frame ID). In other words, the goal will contain the information on where Thorvald should move, i.e. the position of the node on the other side of the field. The action server in the node *move_thorvald* will be executed until Thorvald reaches this position.

The topological navigation system is outside of the scope of this master's thesis, however, the goal message generated by the client can be "faked" by implementing the action client in the terminal and sending the *move_base_msgs/MoveBaseGoal* message directly from there. Using the *axclient,* a tool included in the *actionlib* package, facilitates interaction directly with the action server. This way goal messages can easily be sent in the general user interface (GUI). The algorithm of the node is as follows:

1. Looking up the robot's pose in the map/world frame

Firstly, the node must look up the robot's pose in a global reference system. When working in Gazebo, the navigation node assumes that the robot's estimated pose in a world-fixed frame is available as nav_msgs/Odometry type messages, published by a separate (pre-existing) node. In the real world, however, the robot's pose would be looked up in a map frame, possibly estimated by the GNSS RTK system.

As the robot will be moving towards a position defined in a goal message sent from the Action Client, as well as using an error input to control velocity with closed-loop feedback, it is necessary to update the position and orientation of the robot in the global reference frame. When working in Gazebo, the orientation of Thorvald in the world frame is expressed as a quaternion, and so it is transformed to Euler angles.

2. Rotation to determine cutting tool's position

The two-dimensional position and orientation that we want, is that of the tip of the cutting tool (which side depending on which way Thorvald is moving). The position in the Gazebo world frame provided by the nav_msgs/Odometry message, however, is that of the origin of the base_link reference frame. The cutting tool is 1,8 metres long and (for the purpose of this thesis) placed 30 cm in front of base_link. To determine the position of the cutting tool's tip in the global reference frame, the position must be rotated on the two-dimensional global XY_f-plane depending on the orientation of the robot. The rotation expressed in matrix form is as follows:

$$\begin{bmatrix} X_{Rot} \\ Y_{Rot} \end{bmatrix} = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix} \begin{bmatrix} X_{\alpha=0} \\ Y_{\alpha=0} \end{bmatrix} \quad (54)$$

Figure 7.9 illustrates the rotation of the point P_1, representing the tip of the cutting tool, as the robot rotates in the global reference frame with an angle α.

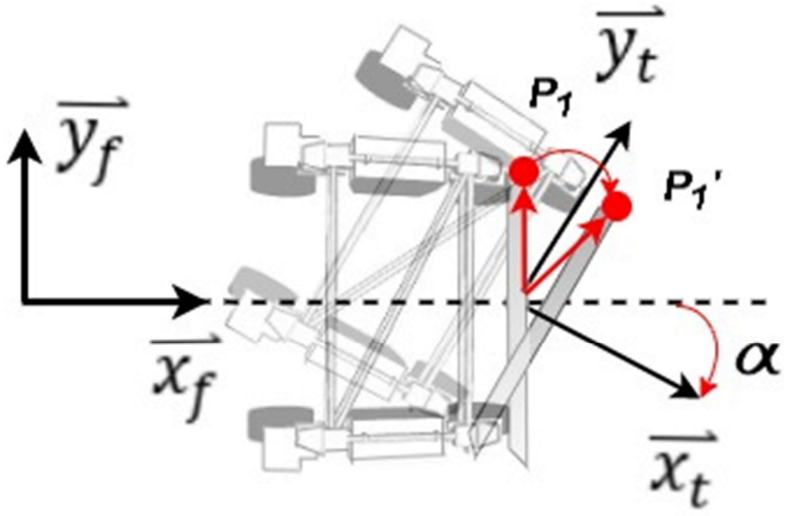


Figure 7.9:As the orientation of Thorvald changes in the global reference system, the position of the tip of the cutter is rotated about the base_link frame.

.

3. Subscribe to and rotate points published in the perception node

The two points that define the calculated line are now updated, and their positions in the global reference system are registered. As the points are defined locally in relation to base_link, they must also be rotated in the two-dimensional plane in the same way the position of the tip of the cutting tool was rotated. The points are then stored in containers, hereby referred to as point-holders, which are constantly updated.

4. Execute Action

At this point, the position of the line and the robot are fully defined in the global frame. The points defining a line are continuously updated and placed in a point-holder container. When the goal message is received from the client as a *move_base_msgs::MoveBaseGoal* message, the first thing that is done is to calculate the distance between the current position of Thorvald and the client goal position (the node on the other side of the field). This way the robot can stop

when it reaches its destination. Secondly, the two points that currently define the line and are in the point-holder at that time, are registered and the distance to the setpoint is calculated. If the distance is less than one metre, the robot is asked to keep updating the setpoint without moving. The robot can stay in this loop for 10 seconds, however, if it remains idle for more than that the action is cancelled.

When the setpoint is further than one metre away, two line-equations are calculated. One equation is that of the line separating the uncut and cut grass that we want the robot to follow. The second line equation defines a line that is orthogonal to the first line, spanning between it and the robot's position. Solving for where these two line equations are equal, the intersection point is found. The magnitude of this second line is then the smallest distance from the separation line and the robot, i.e. Thorvald's lateral displacement.

The objective is for Thorvald to move towards the setpoint and converge on the line until it gets within one metre of the setpoint, then take in another setpoint and so on. Throughout the process it will also always be possible to end the action if it is pre-empted. If Thorvald does not reach the setpoint within 10 seconds, it is considered likely to be the result of an error or an obstacle, and the robot will look for another setpoint.

The error offset is calculated, and the velocity is controlled, as explained in Chapter 6.2 and Chapter 6.3, respectively. A proportional controller is implemented to apply the linear velocity by multiplying the lateral displacement error e_y with the proportional gain K_v. The angular velocity is regulated with a proportional-derivative controller. The angular displacement error e_φ is multiplied with the proportional gain K_h, and there is also a differentiator term where the derivative gain K_d is multiplied with the rate of change of the angular displacement error.

Thorvald's maximum forward speed is set to 1.5 m/s until the lateral error is sufficiently small (less than 1 metre), and the robot's speed converges on 0.6 m/s plus the steady state error. Once Thorvald has reached the client goal position, or the action has been pre-empted at any point, a command will be sent to stop the robot until a new request is received from the client.

To summarise the series of actions that commences when the robot receives the goal message from the action client, pseudocode is presented in Figure 7.10 showing the flow of this part of the algorithm. Cancelation and shutdown of ROS is accounted for in all steps of the algorithm, but for simplicity, these loops have been left out of the pseudocode.

Table 7.1: Action server algorithm.

Input: Goal message. Continuously updated point holder. Odometry information
Output: Thorvald autonomously moving along grass line

```
BEGIN
Calculate distance to goal;
while not at goal and action is active do
|       Calculate distance to setpoint;
|       while distance to setpoint < 1m do
|       |       take in new points from point holder;
|       |       calculate distance to setpoint;
|       |       if in this idle state for 10s then
|       |       |       break loop and cancel action;
|       |       end if
|       end while
|       while distance to setpoint >= 1m do
|       |       calculate the line equation based on the two points fetched;
|       |       calculate the line equation of orthogonal line to the first line;
|       |       find the intersection point;
|       |       for 10s and distance to setpoint > 1m and not at goal do
|       |       |       calculate lateral displacement;
|       |       |       calculate angular displacement;
|       |       |       if lateral displacement < 1m then
|       |       |       |       apply P controller so linear velocity converges on 0.6 m/s;
|       |       |       else
|       |       |       |       apply a constant linear velocity of 1.5 m/s;
|       |       |       end if else
|       |       |       apply PD controller to regulate angular velocity;
|       |       |       if angular velocity control signal > 0.3 rad/s then
|       |       |       |       cap angular velocity at a constant 0.3 rad/s;
|       |       |       end if
|       |       |       update distance to setpoint;
|       |       |       update distance to goal;
|       |       end for
|       |       if action has been cancelled or client goal is reached then
|       |       |       break loop;
|       |       else
|       |       |       take in new points from point holder;
|       |       |       calculate distance to setpoint;
|       |       |       while distance to setpoint < 1m do
|       |       |       |       take in new points from point holder;
|       |       |       |       calculate new distance to setpoint;
|       |       |       |       if in this idle state for 10s then
|       |       |       |       |       break loop and cancel action;
|       |       |       |       end if
|       |       |       end while
|       |       end if else
|       end while
end while
stop robot;
END
```

Chapter 8

Testing and Verification

8.1 Simulation Testing

This chapter will test the developed method to assess the robot's ability to perform according to the main and sub objectives defined in Chapter 1 of this thesis. The tests are divided into simulation testing and tests on captured field data.

Two test-objectives are defined: To further test the line detection algorithm in both simulated environments and on field data, and to evaluate the velocity control and autonomous navigation of the robot. The grass cutting configuration and its configuration files are used in the simulations, as well as the Velodyne simulation package to simulate the use of a VLP-16 LIDAR.

The sensor itself is positioned 90 centimetres above ground level, in the front section of the robot (with equal distance to each front wheel). The sensor is angled downwards with a pitch of 20-degrees. Figure 8.1 shows Thorvald visualised in RViz with the body-fixed *base_link* frame, the sensor frame *lidar_frame*, and the Gazebo world-fixed *odom* frame.

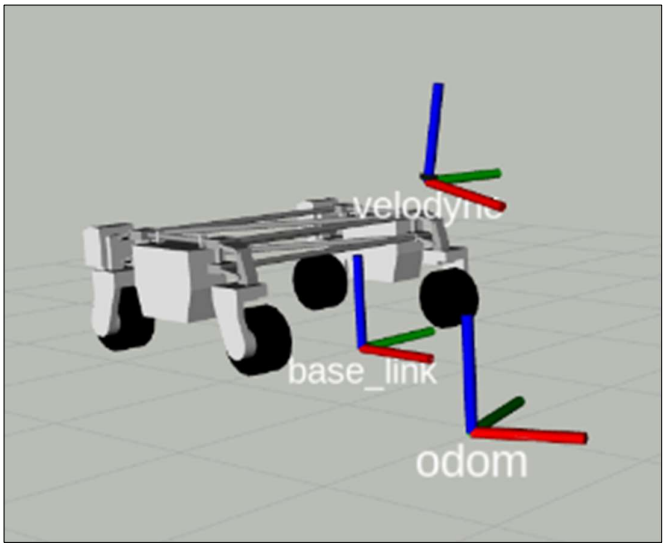


Figure 8.1: Thorvald as visualised in RViz.

8.1.1 Line Detection with Increasing Lateral Displacement

One cannot always guarantee that the robot will start off in the ideal position, or that the grass line will be continuously produced in the same lateral distance to Thorvald. As the robot's lateral distance to the line increases, naturally the angular displacement will increase as well. This test is conducted in Gazebo, to test the algorithms ability to identify the line as the robot is positioned with increasing lateral displacements from the line. The true line position will be compared to the estimated line by comparing the true lateral and angular error offsets with the corresponding estimated error offsets.

The stationary robot is placed in the Gazebo simulated environment, with a rectangular object meant to represent a field of grass. The rectangle's dimensions are 800 cm x 800 cm x 50 cm. The exact position of this simulated field is defined in the simulation software's world-fixed frame, as is the robot's position and orientation. This allows for the true angular displacement error to be calculated manually with equation 42, as the robot's true lateral displacement values are incrementally set. The robot's reference frame is aligned with the fixed reference frame, so that $\vec{x}_t \parallel \vec{x}_f$. The robot is then laterally displaced in increments of 10 centimetres, up to 1 metre on each side of the line. Figure 8.2 shows the design of the simulated environment.

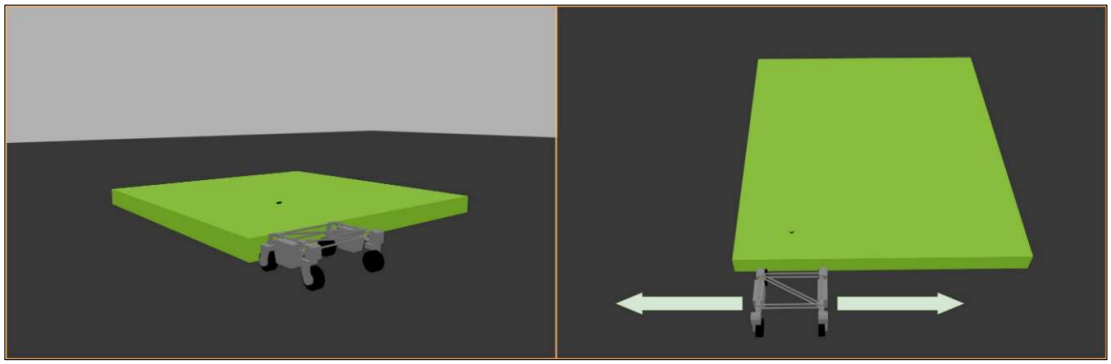


Figure 8.2: The simulated environment, showing how the robot is placed in relation to the "grass", and moved laterally in relation to the line.

8.1.2 Autonomous Navigation Test

Again, Gazebo is used to simulate a real environment. The purpose of this simulation test is to assess the proposed solution as a whole, including the controller in the action server. A goal message will be sent through the GUI of the previously described axclient. The robot will detect the line, assess its error offset, and autonomously navigate in the simulated grass field.

The simulated field is 24 metres long, with changing width to assess the line detection algorithm's efficiency when the line is not straight. This also tests the controller' ability to successfully change trajectories and realign the robot. The field consists of three sections of 8 metres length each, all with a height of 50 centimetres.

The robot's starting position (its position is measured from the left tip of the cutting tool) is 1.55 metres from the field, with an error offset of 22 degrees. The robot will navigate along the first section for 8 metres, then the width of the field will increase with one meter to the robot's left side. After executing a left turn, the robot should then be able to realign its cutting tool to the new grass line. This line shall again be followed for 8 metres, until the robot reaches section three. This section of "grass" has a decreasing width of 0.5 metres, so the robot will have to turn right and realign itself again. Figure 8.3 shows the setup of this test.

The controller's parameters used in the test are:

- Proportional gain for linear velocity: $K_v = 0.5$.
- Proportional gain for angular velocity: $K_h = 0.032$.
- Differential gain for angular velocity: $K_d = 0.01$.

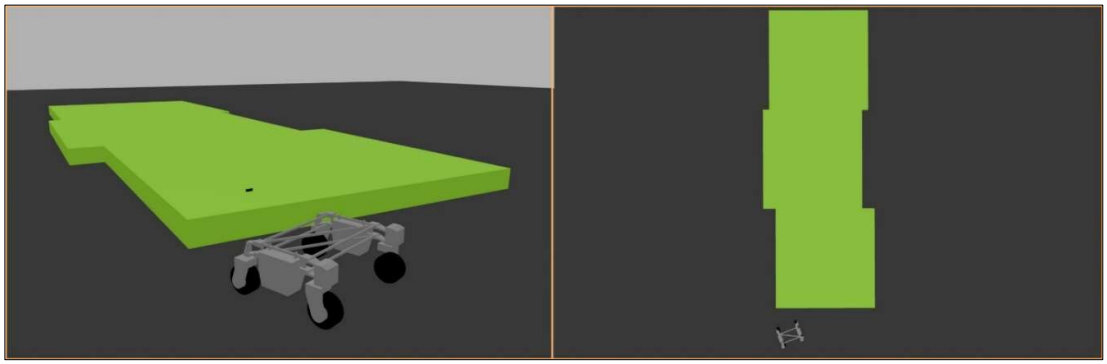


Figure 8.3: The simulated field in the autonomous navigation test, seen from a side-view on the left, and top-view on the right.

8.2 Testing on Captured Field Data

The objective of these tests is to evaluate the line detection algorithm in a real field, under different conditions that the robot might encounter. To accomplish this, applicable recorded data-streams were selected from a series of field trials already conducted by the GrassRobotics project.

The grass cutter configuration was tested on May 21st 2019, in a field on the west coast of Norway. The original purpose of this field test, in 2019, was to gather data and build a dataset that would be used to train a neural network to differentiate between humans and grass in the field.

By subscribing to the topics containing the message data produced by the sensor in real time, the point cloud data (as well as a raw video file recorded by a mounted RGB-camera) was saved in several *rosbag* files. This useful ROS format allows the LIDAR-data produced that day to be applied to the methods developed in this thesis as well. All the raw sensor data, as well as the messages produced by the algorithms (like the markers), are visualised in RViz by subscribing to the published topics (see Figure 8.2).

The existing prototype of the grasscutter configuration was used when testing the robot in the field, with Ackerman steering (angular and forward moving linear velocity control) and a Velodyne VLP-16 LIDAR sensor mounted 90 cm above ground level. Like in the simulation tests, the LIDAR is placed in the front section of the robot (with equal distance to each front wheel). The sensor is also tilted downwards with a pitch of around 20-degrees. Figure 8.4 shows the setup of the robot as it is cutting grass in the field during the field trial in 2019.

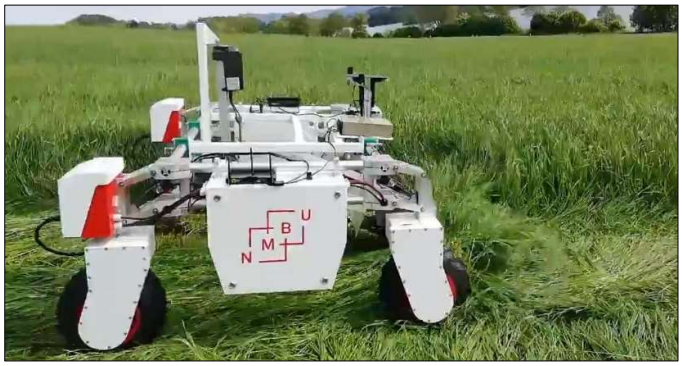


Figure 8.4: Thorvald during a field trial on May 21st 2019.

Table 8.1 lists the series of rosbags that were selected for testing.

Table 8.1: Specifications of the captured data.

Bagfile no.	Runtime [s]	Height of Grass [cm]	Position of line	Slope of field
1	33.9	~ 70	On Thorvald's left side	Uphill
2	86.6	~ 70	On Thorvald's right side	Downhill
3	89.1	~ 70	On Thorvald's right side	Downhill

8.2.1 Non-stationary Left-Sided Line Detection Uphill

In this test, based on data contained in bagfile no. 1, the robot was placed in the field and its velocity and steering was manually controlled. The robot moves uphill at a speed between 0.6 and 1.5 m/s, cutting the grass as it moves. An ideal trajectory was attempted to be maintained by the person controlling the robot, and the field test was recorded on a partly clouded day. Figure 8.5 shows an image taken by the depth camera, and the correlating point cloud produced by the LIDAR. With the naked eye it is quite easy to distinguish the line that separates the cut and uncut grass, and subsequently draw the line, as shown in Figure 8.6.a. If successful, the line should be produced by the algorithm with no significant diversions.

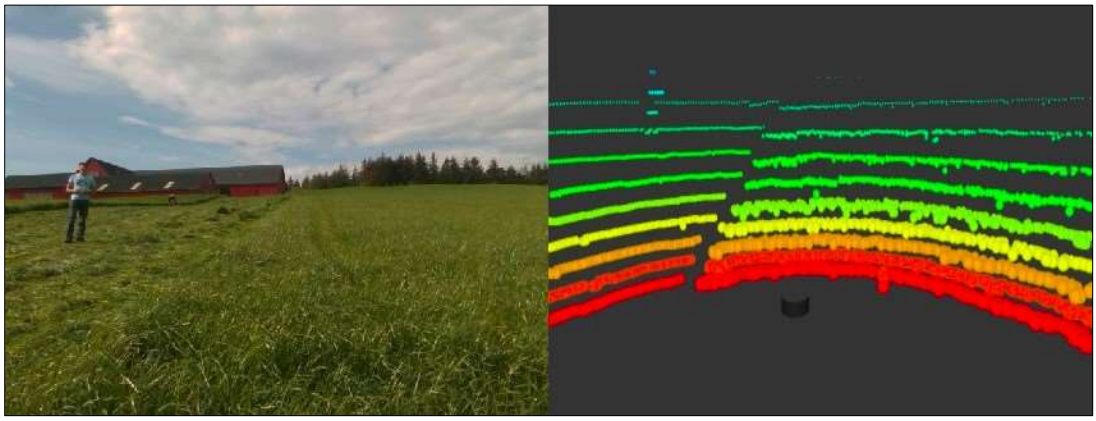


Figure 8.5: The RGB-D image at an instance in bagfile 1, compared to the equivalent point cloud. The edge where the uncut and cut grass meets is clearly visible in both pictures, as is the person standing in the field.

The test will show the algorithm's ability to detect a line in the grass while moving in a field with an uphill slope, with the cut grass to the robot's left side. This means that the line is also generated to the left of the robot, and the left tip of the cutting tool will be following the line (in the code, this translates to the variable *right_side* set to true). The grass is cut low, thus presenting low amounts of noise in the data.

At times, the robot is turned manually to adjust its course, and so the algorithm's ability to detect the line whilst in different orientations is tested. People are also present in the field (although not in the critical zone), which allows for testing of the algorithm's robustness and its ability to filter out objects that are undesirable. The setup of the sensor produces point cloud data as shown in Figure 8.6, visualised in RViz.

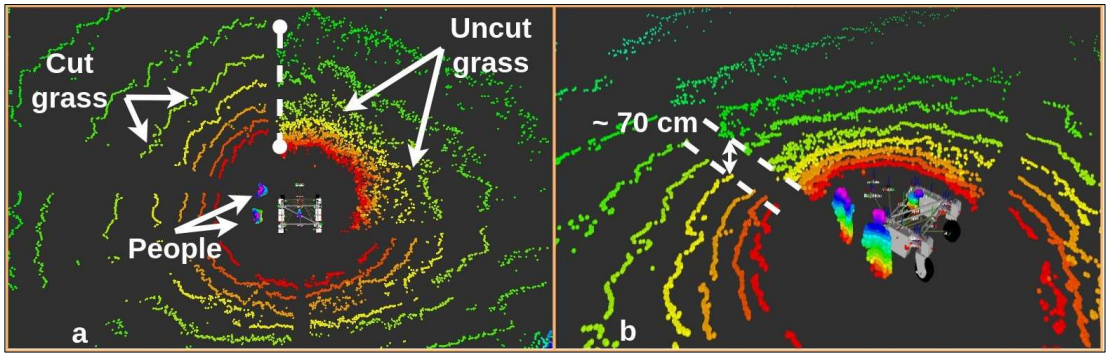


Figure 8.6: a) A top-view of the point cloud environment, the dense point cloud representing the uncut grass visible to the robot's right. The dashed line represents the desired line to be produced by the algorithm; b) A close up of the environment, in front of Thorvald the uncut grass can clearly be distinguished by its elevation over the ground.

The bagfile is played and the perception node is run as well, applying the developed algorithms on the data.

8.2.2 Non-stationary Right-Sided Line Detection Downhill

The conditions in this field test, contained in bagfile no. 2, differ from those in bagfile no. 1. The robot is placed in another field, with the cut grass on the robot's right side. This means that the error offset is calculated from the right tip of the cutting tool this time, and the line is also detected on the right side of the robot (in the code, this translates to the variable *right_side* set to false). In contrast to the environment in bagfile no. 1, it is now a downhill slope, which allows for testing of the algorithm's flexibility.

The present cut grass is cut quite roughly and collected in piles, so there is a substantial element of noise in the data. This noise is accentuated because the height of the uncut grass is quite low in May, compared to what it would be at the time of harvest. Parts of the cut grass reaches heights of around 70 cm, and thus could be registered as part of the plane of the uncut grass. However, this provides excellent conditions to test the applied RANSAC robust estimation model, to fit the correct plane and line. In addition, one of the goals of the GrassRobotics project is to be able to increase the amount of harvests each season. Consequently, the grass is expected to be shorter at the time of cutting, and so it is beneficial to test the algorithm on grass that is not that tall. In this field test, during which data was captured, the robot was placed in the field and its velocity and steering was again manually controlled. As shown in Figure 8.7, comparing the image produced with the RGB-D camera and the point cloud produced by the LIDAR, the separation line is clearly visible.

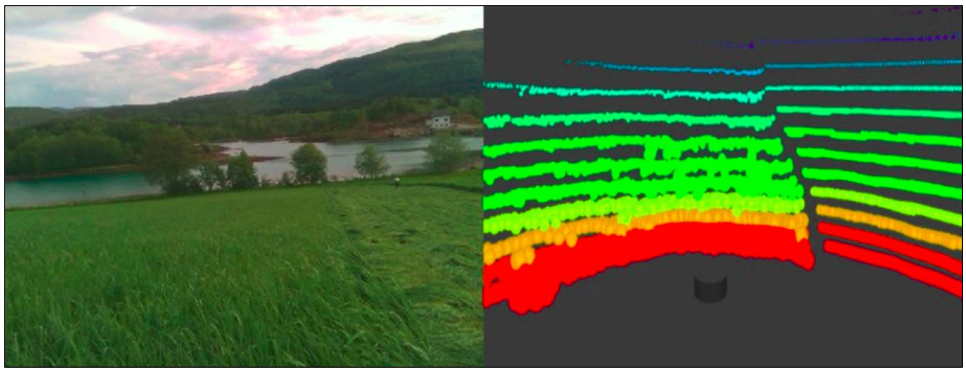


Figure 8.7:The RGB-D image at an instance in bagfile 2 (on the left), compared to the equivalent point cloud (on the right).

In the recorded field trial, the robot moves downhill at a speed between 0.6 and 1.5 m/s, again cutting the grass as it moves. An ideal trajectory is once more attempted to be maintained by the person controlling the robot, and the field trial data was recorded on the same partly clouded day.

To test the algorithm, the bagfile is played so the sensor data is published, whilst running the perception node. The setup of the sensor produces point cloud data that is visualised in RViz, as shown in Figure 8.8.

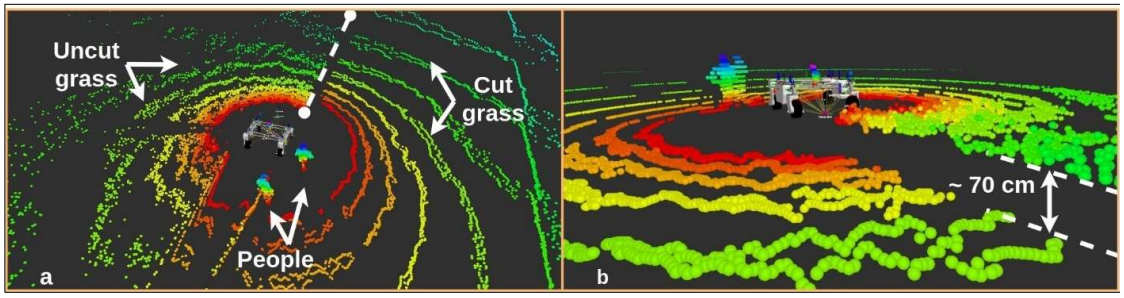


Figure 8.8: a) A birds-eye view of the point cloud environment, with the dense point cloud representing the uncut grass visible to the robot's left. The dashed line represents the desired line produced by the algorithm; b) A close up of the environment, in front of Thorvald the uncut grass can clearly be distinguished by its elevation over the ground.

8.2.3 Stationary Line Detection Test with Moving Objects

In this test, using captured field data contained in bagfile no. 3, the robot is stationary and placed in the same field as in bagfile no. 2. There is a slight downhill slope and the weather conditions are partly cloudy. The algorithm will have to produce the line even as up to three people are present in front of the robot, moving in and out of the critical zone. As seen in Figure 8.9 (only two of the three people are visible), the motion of the people in the field is intentionally diversified. The people in the critical zone are running, walking, and crouching, to be as unpredictable and disturbing to the sensor as possible.

This will test the robot's ability to find the line even under these extreme conditions, emulating conditions the robot might face for example if it comes across a group of deer in the field, or a flock of birds. In which case, obvious steps should be taken as safety procedures, like immediately stopping the cutting tool. However, safety procedures are not the topic of this thesis, and the ability to maintain the line detection algorithm is still important in these circumstances, so as to not lose control of the navigation.

Figure 8.9: An image taken by the RGB-D camera, where two people move around in the critical zone.

Chapter 9

Results

The following results are obtained from the tests conducted in Chapter 8. The results have been divided into those found in simulation testing, and those found using captured field trial data.

9.1 Simulation Testing

9.1.1 Line Detection with Increasing Lateral Distance

The average estimated lateral displacement was 1.90 centimetres from the true lateral displacement. The average angular displacement estimation was 0.38-degrees from the true angular displacement. The trend showed an estimated angular displacement that was slightly higher than the true value when the robot was placed to the left of the line (represented as negative displacement values in the graph of Figure 9.1), and slightly lower compared to the true angular displacement as the robot was moved to the right of the line (as seen in Figure 9.2).

The results showed no apparent pattern of estimated lateral displacements being either greater or lower than the true values. Only one estimation to the left of the line deviated more than 2 cm from the true value (at 10 cm true lateral displacement), and the average deviation was 1.19 cm, whereas to the right of the line the estimations had larger errors. When the robot was placed to the right of the line, the average estimated lateral displacement was 2.78 cm from the true lateral displacement. The highest deviation in lateral displacement values was 8.28 cm, and the highest deviation in angular displacement values was 2.44-degrees, both found at 80 cm to the right of the true line.

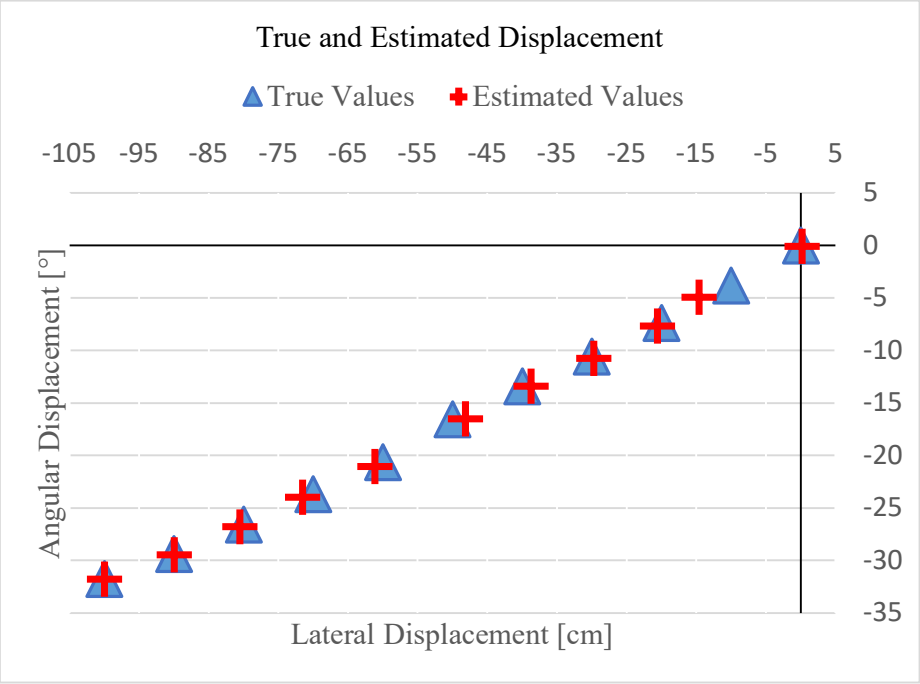


Figure 9.1: True lateral and angular displacement values compared to estimated values, when the robot was placed the left of the line.

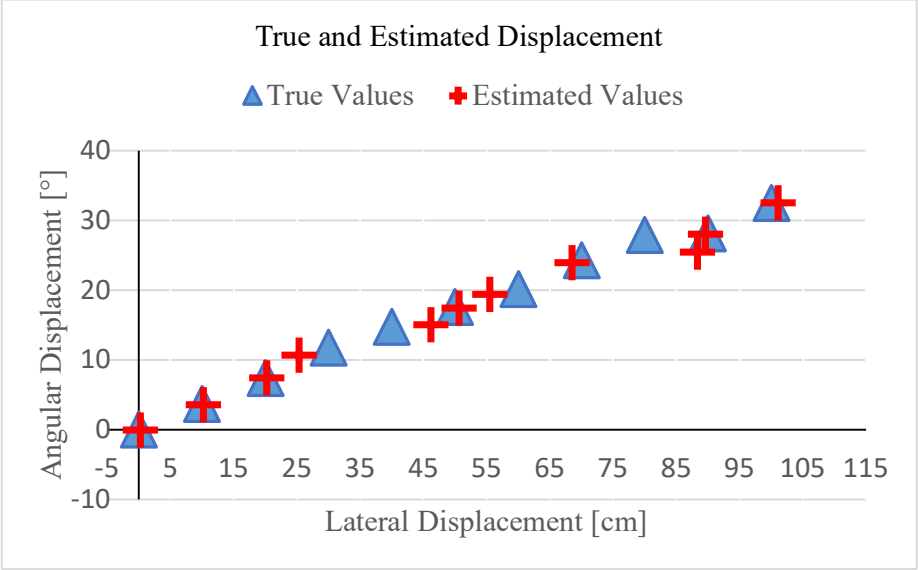


Figure 9.2: True lateral and angular displacement values compared to estimated values, when the robot was placed to the right of the line.

9.1.2 Autonomous Navigation Test

For the duration of the test, the robot detected the separation line and autonomously navigated along this line. The setpoints on the line were generated and applied in the controller, and the controller successfully updated the setpoint when the robot was within 1 metre of the used setpoint. Figure 9.3 illustrates the trajectory of the robot during the test (the cutting tool is not included in the illustration of Thorvald). The tip of the cutting tool, which is the position the algorithm aligns to the line, would be situated just in front of the front left wheel.

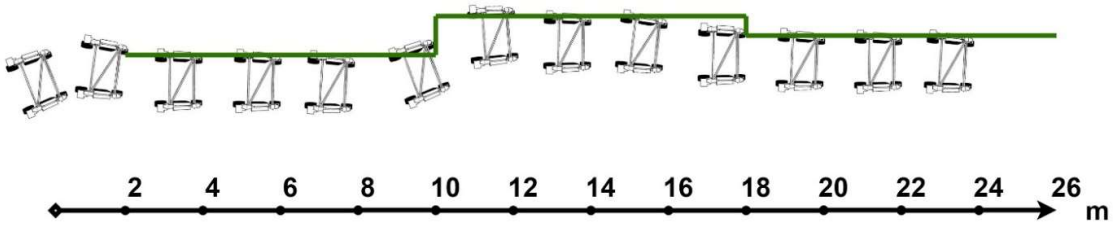


Figure 9.3: The sequence of positions and orientations of the robot as it navigates autonomously in the simulated environment.

The sequence of motion can also be presented graphically as in Figure 9.4, by plotting the robot's position relative to the true line over time.

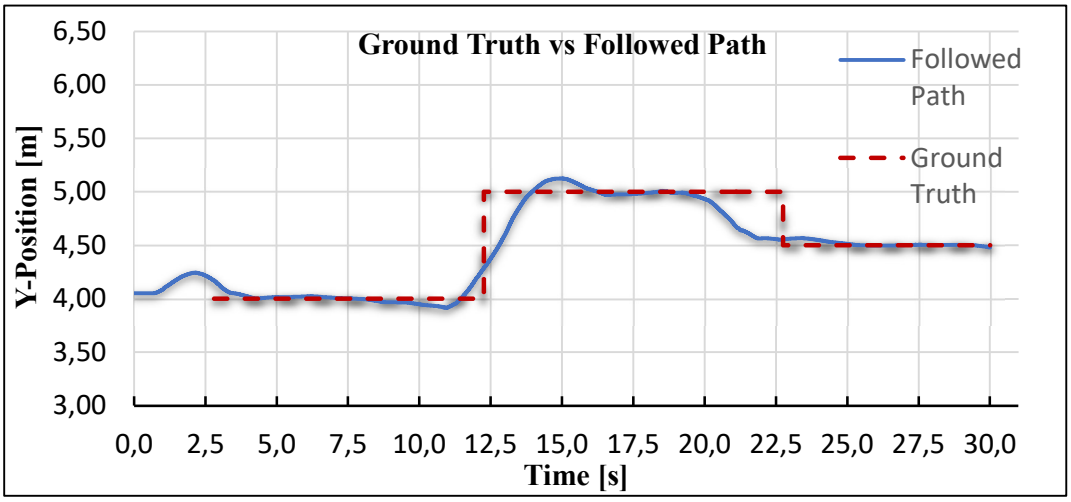


Figure 9.4: The robot's tracked path exhibiting slight overshoot but little oscillation.

Figure 9.5 presents the recorded linear velocity during the test. The linear velocity of the robot remained quite stable, between 0.6 m/s and 1.0 m/s. The linear velocity reaches maximum values at around 12 seconds, as the robot reaches section two of the field and the estimated separation line is at the highest lateral distance from the robot.

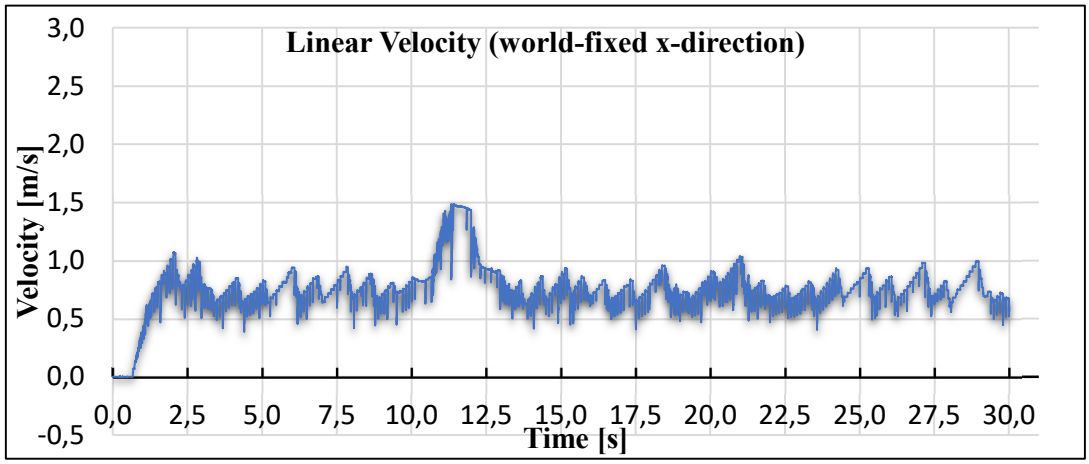


Figure 9.5: Linear forward-velocity of the robot recorded over 30 seconds.

Figure 9.6 presents the recorded angular velocities during the test. Similar to the linear velocity, the angular velocity exhibited quite stable tendencies. The maximum angular velocity (a left turn) occurs in the turn to transition to section 2 of the field at around 12 seconds.

The minimum angular velocities (right turn) occurs in the following seconds as the robot reorients itself after a slight overshoot, and in the first few seconds as the robot adjusts its starting angle of 22 degrees.

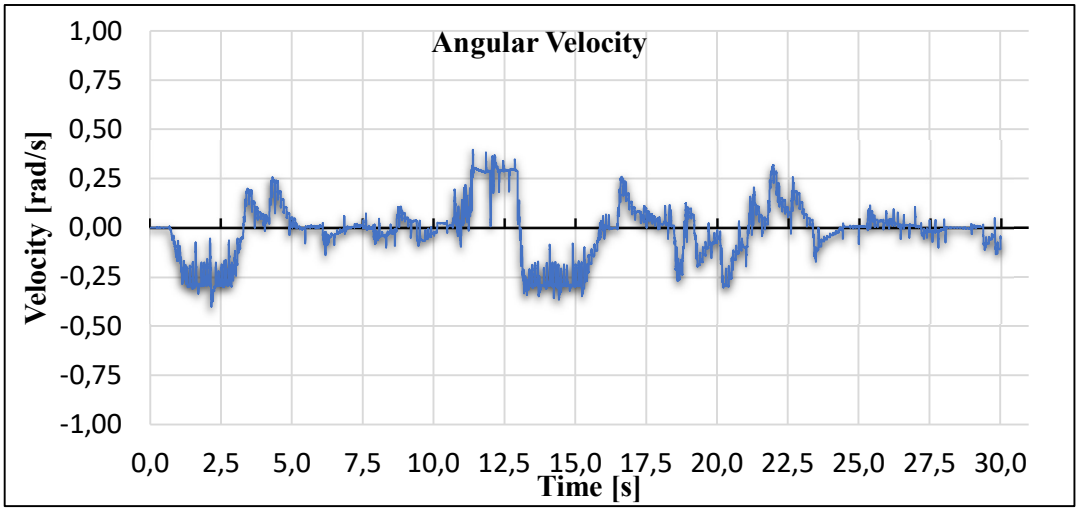


Figure 9.6: Angular velocity of the robot recorded over 30 seconds.

9.2 Testing on Captured Field Data

9.2.1 Non-Stationary Left-Sided Line Detection Uphill

The positively sloped plane of the uncut grass was extracted continuously without noticeable errors, filtering out the ground and the surrounding environment with the applied passthrough filter. As seen in Figure 9.7, the implemented RANSAC algorithm is successfully able to produce an estimate of the plane, based on maximizing the amount of inliers.

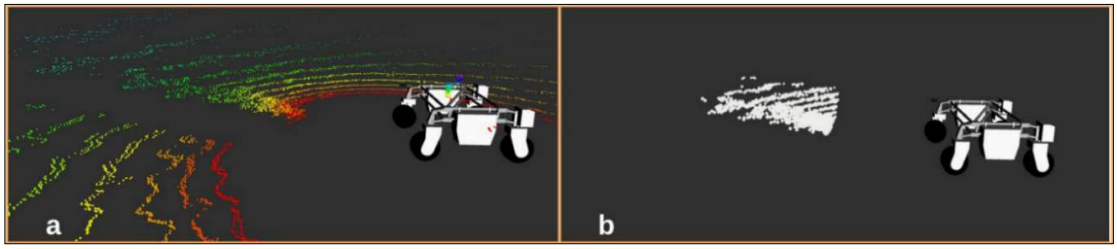


Figure 9.7: a) The robot seen from its left side, before a plane is estimated; b) The plane of the uncut grass is coloured white, estimated by the RANSAC method applied in the node.

The points that lie along the edge of the uncut grass were successfully segmented from the point cloud, as can be seen in Figure 9.8.

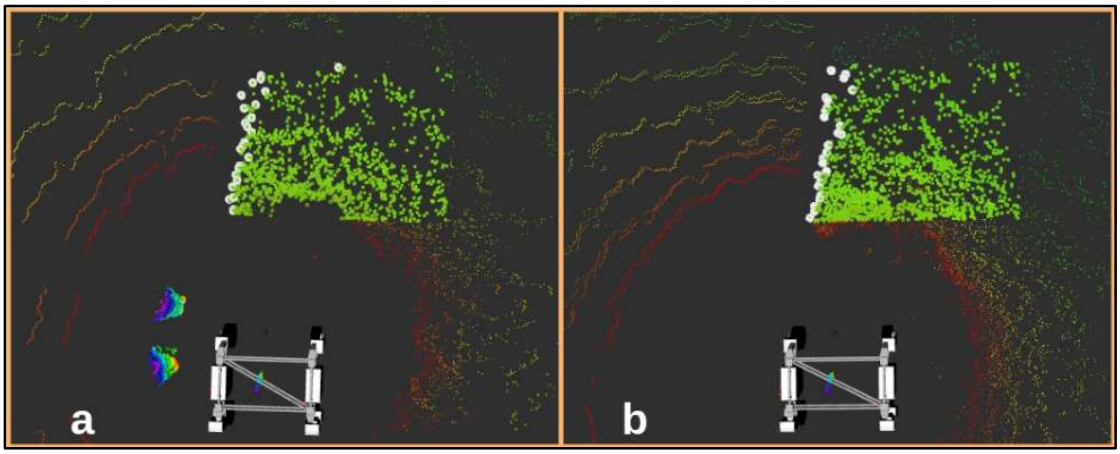


Figure 9.8: a) The points in the estimated plane are coloured green. The white points along the left side of the plane will be fitted to a line; b) The estimated plane with the white points on the left edge of the plane at another instance.

The line was continuously produced and showed no signs of significant errors throughout the runtime of the test. The RANSAC model was able to find the line even when the point distribution in the field was not ideal. The published blue marker is visualized in RViz and shown in Figure 9.9.

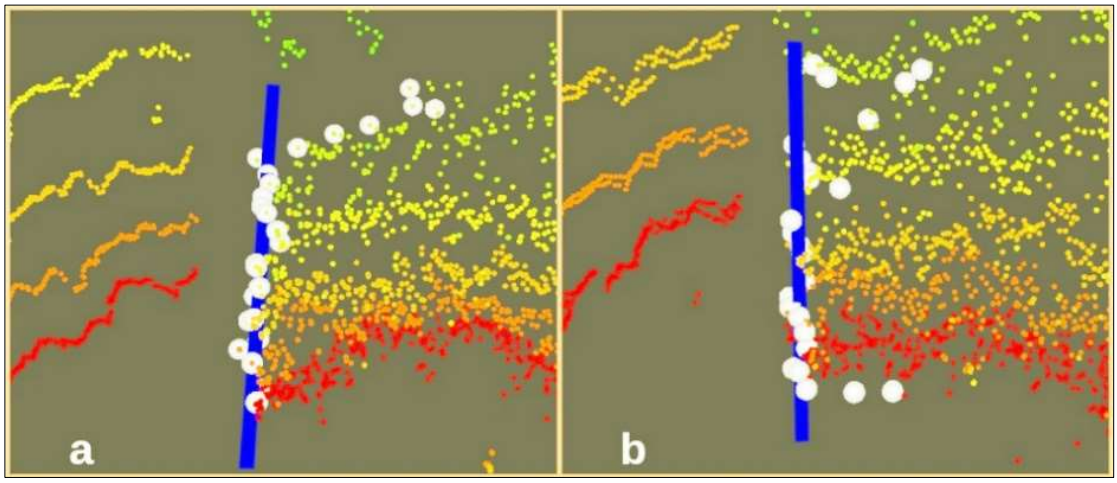


Figure 9.9: a) The RANSAC method demonstrated as it estimates a line along the edge of the uncut grass, with five outliers present in the last 50 cm of the critical zone; b) Another instance of a successful line estimation under the presence of six outliers.

Figure 9.10 illustrates the effectiveness of the algorithm during the test, depicting the successful generation of a line on which the robot could align its cutting tool with and follow.

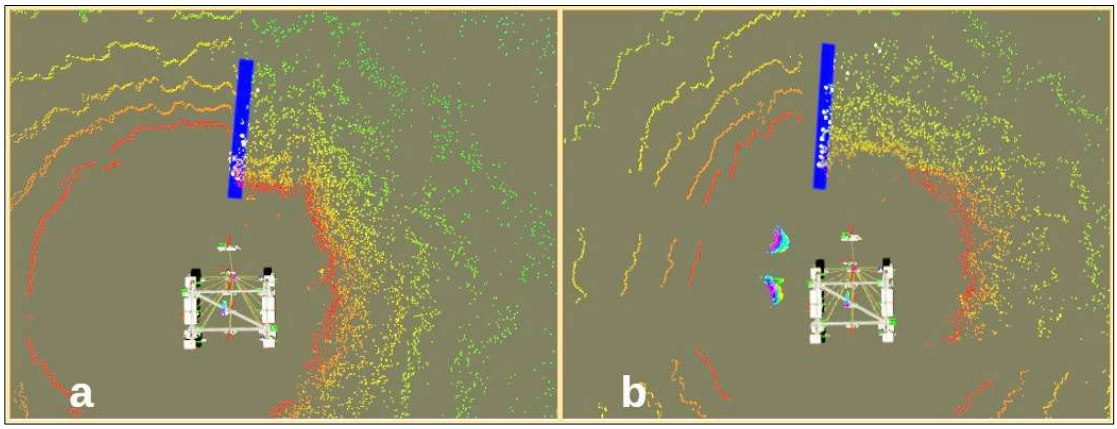


Figure 9.10: a) The estimated line produced by the algorithm, shown as the blue marker is visualized in RViz; b) Another example of the detected line, which Thorvald will navigate along.

As the robot turned, the line was still generated with a satisfactory result. Figure 9.11 illustrates an example of such conditions, as the robot is manually turned about 35-degrees to the left.

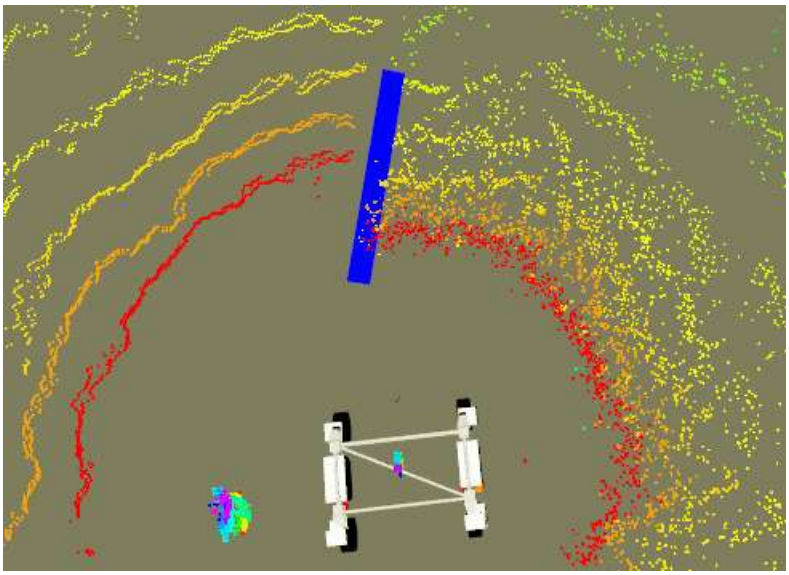


Figure 9.11: The robot is turned up to ca. 35-degrees, showing no effect on the generation of the line.

9.2.2 Non-Stationary Right-Sided Line Detection Downhill

As can be seen in Figure 9.12, the RANSAC algorithm was successful in filtering out the dominant, negatively sloped plane as well, which represents the uncut grass. The plane was again estimated continuously without noticeable errors after filtering out ground and the surrounding environment efficiently. However, the plane was slightly more unevenly spread out, compared to the plane in the previous non-stationary test.

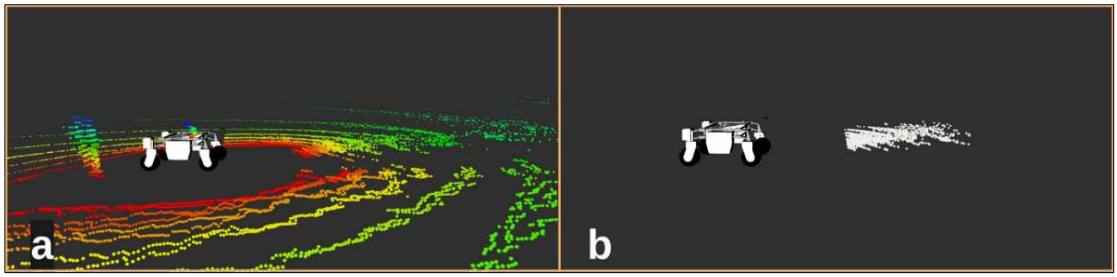


Figure 9.12: a) The robot seen from its right side, the plane of the uncut grass clearly distinguishable; b) The plane of the uncut grass coloured white, estimated by the RANSAC method applied in the node.

As Figure 9.13 shows, the algorithm was also successful in identifying the points that form the edge of the uncut grass, which were used to fit the line. In the figure, the estimated plane is coloured green, and the enlarged, white points represent the filtered edge of the uncut grass.

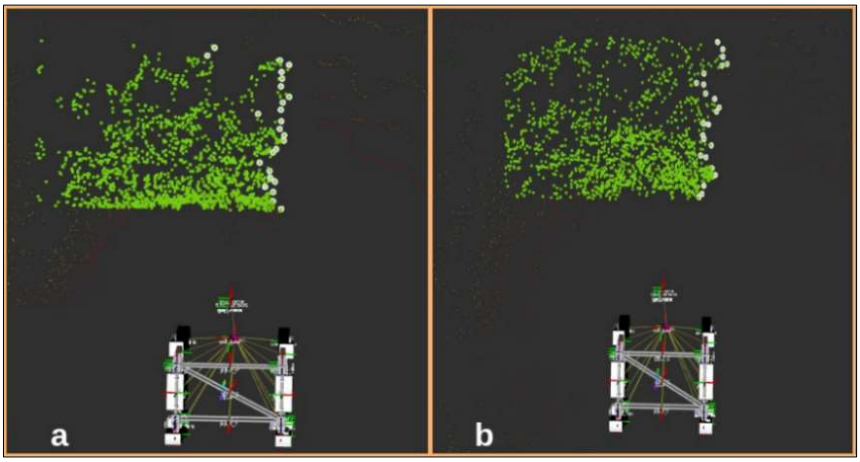


Figure 9.13: a) An estimated plane with selected points in white; b) Another estimated plane with selected points.

The RANSAC algorithm estimated the lines continuously and was able to fit a line successfully to the inliers, showing no significant and important signs of errors throughout the runtime of the test.

At a few instances, piles of the cut grass reached high enough to be falsely considered as inliers in the RANSAC algorithm that estimated the plane. This led to segmented points which were supposed to be representing the line of the uncut grass, being located to the right of this line (see Figure 9.14.a). Despite these disturbances, the robot was successful in estimating the grass line, and did so consistently throughout the test.

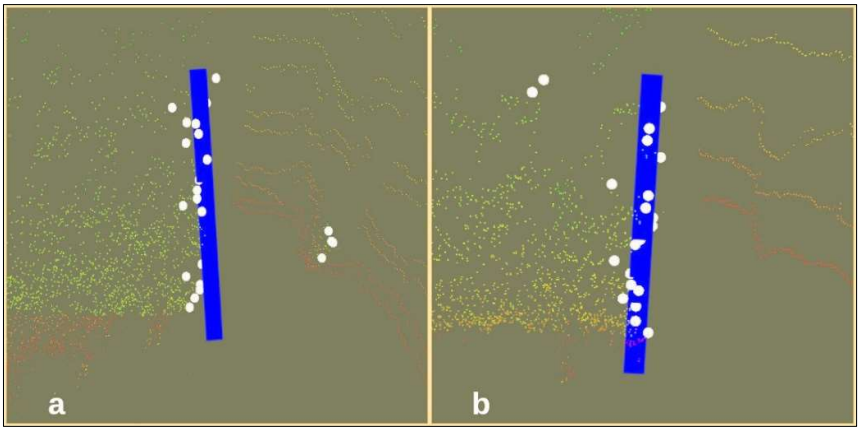


Figure 9.14: a) A line successfully estimated by the RANSAC method under the presence of three outliers in the field of cut grass; b) The RANSAC method successfully estimating a line in the presence of four outliers.

Figure 9.15 illustrates how effective the algorithm performed during the test. The figure depicts the successful generation of a line, represented by the blue marker published in the perception node.

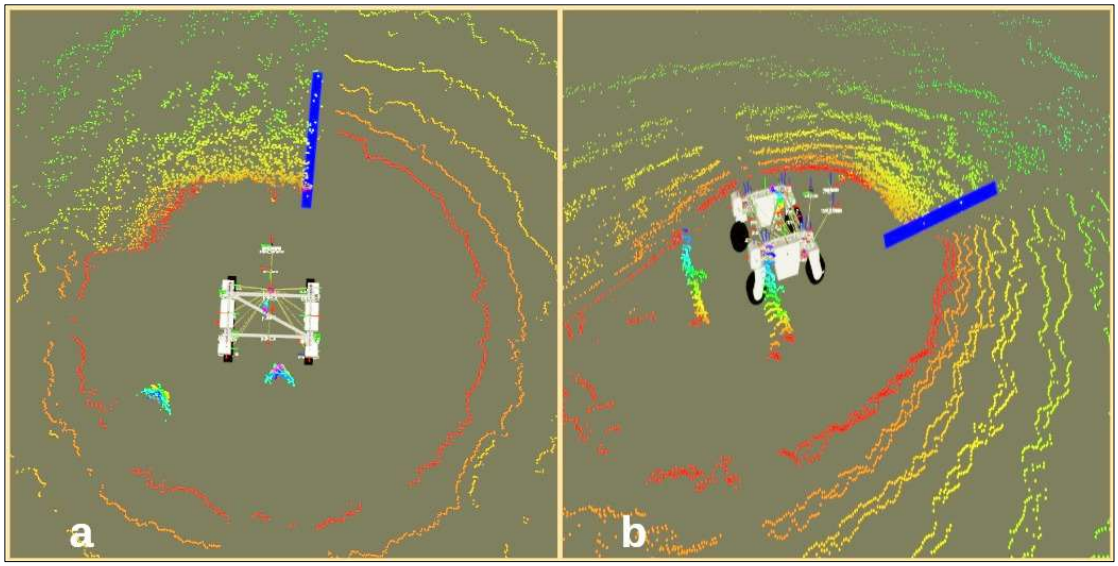


Figure 9.15: a) A successful estimation of the line at an instance of time; b) The robot and the detected line as seen from the right, as it makes its way along across the field.

9.2.3 Stationary Line Detection Test with Moving Objects

The line was continuously and successfully generated in this test as well, showing no signs of being affected by the three people in the critical zone. The line is detected and generated even as the subjects in the test walk in and out of the critical zone.

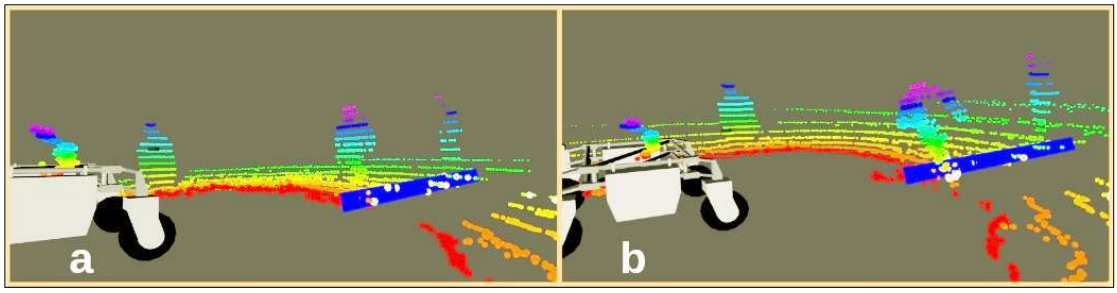


Figure 9.16: a) The line is successfully generated (coloured blue) as three people are situated in the critical zone; b) The presence and motion of the people have no effect on the line generation algorithm, even when the person is crossing the line.

Chapter 10

Discussion

10.1 The ROS framework

ROS has proven a good option for prototyping throughout this thesis, as it is timesaving and lightweight. It is, however, important to realise it might not be the software framework used in a final commercialised grass cutting robot. In some cases, efficiency and real-time processing is sacrificed for higher stability and management of complexity, by placing core services in a modular design (31), (33). The lack of safety protection can also be an issue, and on a robot equipped with knives, safety is obviously a crucial aspect that must be addressed before mass production. The communication between nodes in ROS is not encrypted. This makes it fast, easy to use, and results in general high performance, however, it also means that it might be exposed to external interference that can lead to dangerous situations. It is quite simple to pick up a message externally and also deceive the node to generate false messages (40). The modular approach of ROS can also be a security risk, as the way publishers and subscribers are used to pass messages between the nodes, also ensure that the nodes are completely independent and anonymous. This means that in a topic where there are several subscribers, each subscriber is unaware of the other subscribers. Perhaps more worrying, the publisher is also unaware of the subscribers, or even if there are other publishers publishing data to the same topic. This means that anyone, that for whatever reason is able to access the same network that the grass cutting robot is running on, will have access to all of its data. With this access one is able to send commands, overload the network, and tap into published sensor data to watch point clouds or camera-streaming. It is also worth noting that the transmission control protocol (TCP), which handles the communication between the internet protocol (IP) and the running application, is also unprotected. This means that unauthorised third parties can quite easily send external

packages to the ROS ports, put in messages, and even completely replace publishers or subscribers. At the time of writing, however, a ROS 2.0 project is currently in development, addressing these issues. ROS 2.0 could perhaps be an option for future Thorvald production.

10.2 Simulation Testing

10.2.1 Line Detection with Increasing Lateral Distance

The results presented in Chapter 9.1.1 illustrated the robot's ability to generate the line for the duration of the test. The algorithm was proficient at estimating both the lateral displacement offset, and the angular offset. The estimated displacement values generally coincide well with the true values, though the estimated angular and lateral displacement does appear to be more accurate to the left of the line.

A possible explanation for this finding could be that when the sensor is placed to the left of the line, it is positioned over the "cut grass". This means that the edge of the "field" will be more distinct, as the horizontal side of the "uncut grass" is visible from the sensor's perspective. When the robot is moved to the right of the line, it is positioned over the "uncut grass", and so the edge is not as distinct. An option that can be explored to take advantage of these findings, is equipping the current rigid sensor mount with an automated motorised slider, where the sensor is slid from side to side depending on which side the uncut grass is on.

The difference between the true and estimated values shows no apparent linear relation with the lateral distance to the line on either side. The differences did not increase constantly as the robot was placed further away from the line during the simulated test. However, the overall accuracy of the estimations appears to be the most stable within 20 cm true lateral displacement on either side of the line. The average difference that was found when comparing lateral displacement estimation and true values (1.19 cm and 2.78 cm depending on which side of the separation line the robot was placed), suggest that requiring the tip of the cutting tool to position itself 2-3 cm over the estimated line could improve results when cutting. Overall, the results are promising, and show that the algorithm can deliver quite stable and consistent results, somewhat independently of lateral displacement up to one metre from the line.

10.2.2 Autonomous Navigation Test

The autonomous navigation test showed that the robot was able to detect the edge of the “uncut grass” and generate the line effectively. The idea behind the algorithm seemed to work well, as the setpoints were consistently updated and used by the controller.

During the design of the velocity control, the gain was tuned by simple trial and error methods. To reach a satisfactory transient response, the parameters were tuned to satisfy certain conditions: The goal of the tuning was to have minimal overshoot as the robot reached the line, while balancing the response time of the system. The rise time (the time the robot used to reach its setpoints) and the settling time (the time before the robot stabilised its trajectory) also had to be balanced.

The rise time seemed quite good, as even the unnaturally sharp turns of the field are quickly addressed, and the new lines are reached quickly. The trajectory of the robot (see Figure 9.3 and 9.4) showed that some overshoot was present, but with very little oscillation as the robot seemed to be able to stabilize its course within few seconds. The steady state error did not affect the overall result noticeably, as expected. Overall, the P- and PD-controller implemented in the action server seemed to work well, but to address the slight overshoot the parameters of the PD-controller could be tuned more extensively.

In the straight sections, the robot seemed to keep a very consistent trajectory, showing that every new line estimation did not vary much from the previous line. During the test, every two lines which were generated were averaged to produce one line. This averaging frequency seemed to work well, but it would be interesting to see how much more stability could be produced by further increasing the amount of individual line estimations that are averaged. By increasing the amount of lines that are averaged, however, the lines would not be generated as often, and the system would be slower.

The linear velocities presented in Figure 9.5 shows that the velocities stay within the desired speed range of 0.6 m/s to 1.5 m/s, disregarding the horizontal lines that are likely insignificant errors in the simulated values. These horizontal “jumps” likely stem from the fact that the simulation was run with a low real time factor (ca. 0.1) due to a high complexity of the models, meaning that the simulation was lagging a little bit. The angular velocities graphically presented in Figure 9.6 also stay within the desired range of -0.3 rad/s to 0.3 rad/s (again disregarding horizontal lines).

10.3 Testing on Captured Field Data

10.3.1 Non-Stationary Line Detection

The algorithm performed well throughout the recorded tests. The grass field environment proved to yield quite a few outliers, but the robustness of the estimation model seemed to be sufficient. The line was successfully and consistently estimated correctly, even under presence of a high number of outliers. For example, the RANSAC-produced line in Figure 9.8.b, has a total of six outliers and 19 inliers. This means that the line seemed correctly estimated, even though 24 % of the points in the data set are correctly classified as outliers.

The results of the test suggest that the applied threshold distance of 15 cm in the RANSAC plane estimation model, as suggested in Chapter 7, produced an estimated plane that remained sufficiently dense. The threshold distance of the RANSAC line estimation model was set to 8 cm, a value that seemed to perform well during the tests.

The critical zone was defined in Chapter 7.2 as 2.5 metres long, and 6 metres wide, an area that proved big enough to gather all the information required to produce the line. The slope of the field had no apparent effect on the algorithm, as it was able to deal with the unstructured environment in both a positively and negatively sloped field. The orientation of the robot did not affect the successful generation of the line either.

10.3.2 Stationary Line Detection Test with Moving Objects

It was shown that placing people (or objects) in the critical zone did not disrupt the algorithm, even when people were moving on the line itself. The RANSAC model successfully fit the planes to the points that were classified as inliers, independently of the people present in the critical zone. The planes were dense with points, making further manipulation of the point cloud possible. This attests to the efficiency of the plane segmentation algorithm, and to the line fitting model, even under similar conditions with high level of noise.

10.4 Assumptions and Simplifications

During the control system design in Chapter 6, certain potentially unrealistic assumptions were made to simplify the presentation of the problem. Specifically, the assumption that the robotic system can be approximated as a linear and time invariant system, that the models for driving

and steering are identical and independent and can be represented by a second order transfer function.

This approximation of an LTI system might not be realistic, as there most certainly will exist nonlinearities in the real world like static friction and actuator limitations. By linearizing the system, however, allowed for the mathematical expression of the system's closed-loop transfer function, and implementation of linear PID control that will hopefully prove effective at keeping the system in a linear operational region. All the stated assumptions and simplifications most be further tested to be proven acceptable (see Chapter 11.2 *Further Work)*.

10.5 Achievement of Objectives

The main objective of this thesis was to implement sensor technology and develop a method for a robot to autonomously identify the separation line and be able to follow it whilst cutting grass in the fields. Sub objectives were defined to find the best suited sensor technology for the task, to develop an algorithm to find the separation line, and to control the robot in a feedback system. The sub objectives also included testing the suggested solution with simulation software and in the field.

The objectives have been reached satisfactorily, as this thesis has presented an approach based on the best suited sensor and the ROS framework. The solutions have been thoroughly tested in simulation software, as well as on captured field trial data. Due to efforts to contain the spread of the COVID-19 virus in the Spring of 2020, however, new field trials were difficult to organize and conduct. Fortunately, by having access to applicable field trial data from May 2019, the perception and line detection methods presented in this thesis could be tested on this captured data.

The tests, both simulated in Gazebo and conducted on captured field trial data, were all successfully conducted and achieved their respective objectives, defined in Chapter 8.

Chapter 11

Conclusion and Future Work

11.1 Conclusion

The work presented in this thesis shows that a grass cutting robot can detect the line separating uncut and cut grass, calculate the ideal path for the cutting tool, and ultimately adjust to and follow this line. By taking advantage of the data produced by a single LIDAR-sensor, Thorvald can cut grass without leaving behind unnecessarily large rows of uncut grass in the field.

Preliminary testing showed promising results, although further field trial testing is required. The testing on captured field data showed that the developed algorithms and methods were effective at detecting the ideal path through the grass field. The detection methods proved successful in a series of different environments. Negatively and positively sloped fields, orientation of the robot, and obstacles in the immediate surroundings of the robot showed no negative effects on the estimation of the line. Furthermore, the robustness of the RANSAC estimation model produced promising results and showed that a grass-field is an environment that the Thorvald system can operate in successfully. The simulations illustrated the methods' ability to estimate the line with different lateral distances to the robot, efficiently calculating error offsets with increasing angles. The simulations also proved that a proportional controller is sufficient for linear velocity control, and a PD-controller can proficiently control the angular velocity, although the current implementation of closed-loop angular velocity control leads to small degrees of overshoot and oscillation.

Furthermore, the general process described in this thesis is one that could be applied to several other types of crops, and the algorithms developed are general enough to be applied to any sensor producing point cloud data.

11.2 Future Work

11.2.1 Sensor Placement

During the field trials and simulation testing presented in this thesis, the LIDAR sensor was placed at around 90 centimetres above ground level, centrally placed in the front section of the robot and with equal distance to each front wheel. The simulated tests of line detection with increasing lateral displacement from the robot, however, showed that the estimated error offsets were more precise when the sensor was located to the left of the line (over the uncut grass).

Future work can consider taking advantage of these findings, by testing different placements and mounts of the sensor. Placing the sensor in different distances to ground level and recording its effect on data density and overall efficiency has not been explored in this thesis but should be explored in future work.

11.2.2 Further Testing

In the future, more rigorous field trials should be performed to test the complete autonomous perception *and* navigation system in a grass field. The future field trials should include different weather conditions that may impact the system. For instance, the reflectivity of the grass blades, and thus the data produced by the sensor, might differ in rain. Additionally, it would be interesting to see how the data is affected by fog, mist, and other atmospheric conditions.

Further field testing is also required to test the suggested control system design, to see if the nonlinear dynamics are in fact negligible. If the tests prove unsuccessful, one might have to implement a nonlinear controller.

11.2.3 Tuning of Controller

A more extensive tuning can be performed in the future, to optimize the selection of controller parameters. Although the control design performs satisfactorily in the test performed in this thesis, it could be even better adjusted by applying popular tuning techniques like the Ziegler Nichols Method.

11.2.4 Algorithm Development

The algorithm should be continued to define what happens when the client goal is reached, and how it is reached. Currently, the robot stops, and the action is completed when the distance to the client goal is 0.0 m. This is an unlikely precision that can only be obtained during simulations, and so a tolerance should be applied. Future algorithm development can explore the idea of adding a line based on the client goal, that runs along the edge of the field, so that the robot never crosses this line.

In addition, for a higher degree of control of the trajectory of the robot, the algorithm can be developed to define a line between the starting node and the end node defined in the topological map (on opposite sides of the field). This way, if the robot moves too far from the projected row, some action can be taken.

References

1. Bahrin MAK, Othman MF, Azli NHN, Talib MF. Industry 4.0: A review on industrial automation and robotic. Jurnal Teknologi. 2016;78(6-13):137-43.
2. From PJ. GrassRobotics - A novel adaptation strategy for forage production under wet growing conditions - robotization and high quality forages [Internet]. Ås: Norges miljø- og biovitenskapelige universitet; 2019 [cited 2020 10. February]. Available from: https://www.nmbu.no/en/projects/node/34459.
3. Dyer A. Chasing the Red Queen: the evolutionary race between agricultural pests and poisons. Washington: Island Press; 2014.
4. Saga Robotics. Get to know us [Internet]. Ås: Saga Robotics; 2020 [updated 2020; cited 2020 24. January]. Available from: https://sagarobotics.com/pages/about-us.
5. Torgersen J. Mobile agricultural robot [master's thesis]. Ås: Norges miljø- og biovitenskapelige universitet; 2014.
6. Meltzer F. Fremdrift og energiforbruk for autonom landbruksrobot [master's thesis]. Ås: Norges miljø- og biovitenskapelige universitet; 2014.
7. Blomberg F. Rammekonstruksjon til autonom landbruksmaskin [master's thesis]. Ås: Norges miljø- og biovitenskapelige universitet; 2014.
8. Grimstad L. Powertrain, steering and control components for the NMBU agricultural mobile robotic platform [master's thesis]. Ås: Norges miljø- og biovitenskapelige universitet; 2014.
9. From PJ. Roboter kan løse matvarekrisen [Internet]. Oslo: Aftenposten; 2016 [updated 15.02.2016; cited 2020 02. February]. Available from: https://www.aftenposten.no/meninger/debatt/i/a7OO/roboter-kan-loese-matvarekrisen-paal-johan-from.
10. Grimstad L, From PJ. The Thorvald II agricultural robotic system. Robotics. 2017;6(4):24.
11. Saga Robotics. Meet Thorvald - the modular agricultural multipurpose robot [Internet]. Ås: Saga Robotics; 2020 [updated 2020; cited 2020 02. February]. Available from: https://sagarobotics.com/pages/thorvald-platform.

12. Isaksen AX, Grelland N. Utredning av energieffektive metoder for å kutte gras med den autonome landbruksroboten Thorvald [master's thesis]. Ås: Norges miljø- og biovitenskapelige universitet; 2018.

13. Groves PD. Principles of GNSS, inertial, and multisensor integrated navigation systems: Artech House; 2008.

14. Prochniewicz D, Szpunar R, Walo J. A new study of describing the reliability of GNSS Network RTK positioning with the use of quality indicators. Measurement Science and Technology. 2016;28(1):015012.

15. El-Mowafy A. Precise real-time positioning using network RTK. In: Jin S, editor. Global navigation satellite systems: signal, theory and applications. Rijeka, Croatia: InTech; 2012. p. 161 - 88.

16. Xaud MFS, Leite AC, Barbosa ES, Faria HD, Loureiro GSM, From PJ. Robotic tankette for intelligent bioenergy agriculture: Design, development and field tests. The XXII Brazilian Conference on Automation. João Pessoa: The Brazilian Society of Automation; 2018.

17. Xaud MFS, Leite AC, From PJ. Thermal image based navigation system for skid-steering mobile robots in sugarcane crops. 2019 International Conference on Robotics and Automation (ICRA): IEEE; 2019. p. 1808-14.

18. Siciliano B, Khatib O. Springer handbook of robotics. Berlin: Springer; 2016.

19. Rosin PL, Lai YK, Shao L, Liu Y. RGB-D image analysis and processing. Berlin: Springer international publishing; 2019.

20. Intel Corp. Depth camera D415 [Internet]. 2020 [cited 2020 15. April]. Available from: https://www.intelrealsense.com/depth-camera-d415/.

21. Okunsky MV, Nesterova NV. Velodyne LIDAR method for sensor data decoding. IOP conference series: materials science and engineering. 516. Tomsk: IOP Publishing; 2019. p. 012018.

22. Velodyne LIDAR puck [Internet]. Velodyne Acoustics, Inc.; 2020 [cited 2020 23. April]. Available from: https://www.amtechs.co.jp/product/VLP-16-Puck.pdf.

23. Schnabel R, Wessel R, Wahl R, Klein R. Shape recognition in 3d point-clouds. The 16-th International Conference in Central Europe on Computer Graphics. Visualization and Computer Vision: Václav Skala - UNION Agency; 2008. p. 65-72.

24. Leberl F, Irschara A, Pock T, Meixner P, Gruber M, Scholz S, et al. Point clouds. Photogrammetric engineering & remote sensing. 2010;76(10):1123-34.

25. Fischler MA, Bolles RC. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. Communications of the ACM. 1981;24(6):381-95.

26. Bolles RC, Fischler MA. A RANSAC-based approach to model fitting and its application to finding cylinders in range data. the 7th International Joint Conference on Artificial Intelligence 1981. Vancouver: IJCAI; 1981. p. 637-43.

27. Baltsavias EP, Gruen A, VanGool L. Automatic extraction of man-made objects from aerial and satellite images III. Lisse: CRC Press; 2001.

28. Hwang CL, Yoon K. Multiple attribute decision making. Basic concepts and foundations. Lecture notes in economics and mathematical systems. 186. Berlin: Springer; 1981.

29. Roszkowska E. Rank ordering criteria weighting methods–a comparative overview. Studia Ekonomiczne. 2013;5(65):14-31.

30. Vafaei N, Ribeiro RA, Camarinha-Matos LM. Normalization techniques for multi-criteria decision making: Analytical hierarchy process case study. In: S N, editor. Technological innovation for cyber-physical systems. 470. Costa de Caparica, Portugal: Springer, Cham; 2016.

31. Joseph L. Robot operating system (ROS) for absolute beginners. New York: Springer; 2018.

32. Pajaziti A, Avdullahu P. SLAM–map building and navigation via ROS. International Journal of Intelligent Systems and Applications in Engineering. 2014;2(4):71-5.

33. Quigley M, Conley K, Gerkey B, Faust J, Foote T, Leibs J, et al. ROS: an open-source Robot Operating System. International Conference on Robotics and Automation. Kobe Open-Source Software workshop; 2009.

34. Achmad MS, Priyandoko G, Roali R, Daud M. Tele-operated mobile robot for 3D visual inspection utilizing distributed operating system platform. International Journal of Vehicle Structures and Systems. 2017;9(3):190-4.

35. Point Cloud Library. About: What is PCL? [Internet]. PCL; 2020 [cited 2020 26. February]. Available from: http://www.pointclouds.org/about/.

36. Hsieh CT. An efficient development of 3D surface registration by Point Cloud Library (PCL). 2012 International Symposium on Intelligent Signal Processing and Communications Systems (ISPACS). Taipei: Ming Chi University of Technology; 2012. p. 729-34.

37. Ramiya AM, Nidamanuri RR, Krishnan R. Segmentation based building detection approach from LiDAR point cloud. The Egyptian Journal of Remote Sensing and Space Science. 2017;20(1):71-7.

38. Iwasaki Y, Misumi M, Nakamiya T. Robust vehicle detection under various environmental conditions using an infrared thermal camera and its application to road traffic flow monitoring. Sensors. 2013;13(6):7756-73.

39. Grimstad L, From PJ. Software components of the Thorvald II modular robot. 2018;39:157-65.

40. Yampolskiy RV. Artificial intelligence safety and security. New York: Chapman and Hall/CRC; 2018.

Appendices

Appendix A

C++ Code

A.1 ROS Nodes

A.1.1 Perception Node

This source code is located in the file *line_node.cpp:*

```
1  #include <package_find_line/PointCloudSensor.h>
2
3  int main(int argc, char **argv)
4  {
5     ros::init(argc, argv, "line_node");
6     ros::NodeHandle nh;
7
8     // The class is instantiated
9     PointCloudSensor pointcloudSensor(&nh);
10    ros::spin();
11 };
```

A.1.2 Navigation Node

This source code is located in the file *move_thorvald.cpp:*

```
1  #include <ros/ros.h>
2  #include <actionlib/server/simple_action_server.h>
3  #include <move_base_msgs/MoveBaseAction.h>
4  #include <geometry_msgs/Twist.h>
5  #include <iostream>
6  #include <cmath>
7  #include <sstream>
8  #include <nav_msgs/Odometry.h>
9  #include <tf/tf.h>
```

```
10 #include <tf/transform_datatypes.h>
11 #include <geometry_msgs/PoseArray.h>
12 #define PI 3.14159265
13
14 class RobotCommandAction
15 {
16 protected:
17    ros::NodeHandle nh_;
18    ros::Publisher cmd_pub_;
19    ros::Subscriber line_sub_;
20    ros::Subscriber odom_sub_;
21    // action server object
22    actionlib::SimpleActionServer<move_base_msgs::MoveBaseAction> as_;
23    std::string action_name_;
24    bool right_side_ = true;
25    float pointDistance_;
26    geometry_msgs::Point pointHolder1_;
27    geometry_msgs::Point pointHolder2_;
28    double grassCutterPos_x_ = 0.0;
29    double grassCutterPos_y_ = 0.0;
30    double thorOrientation_ = 0.0;
31    double thorOrientationRad_;
32    float Kv_ = 0.5;
33    float Kh_ = 0.032;
34    float Kd_ = 0.001;
35    float knife_x_ = 0.3;
36    float knife_y_ = 0.9;
37    double prev_error_o_ = 0.0;
38    double freq_ = 10.0;
39    double baselinkPos_x_;
40    double baselinkPos_y_;
41
42 public:
43
44    RobotCommandAction(std::string name) :
45    as_(nh_, name, boost::bind(&RobotCommandAction::executeCB, this, _1),
      false),
46    action_name_(name)
47    {
48       // Subscribes to estimated line, pose data and initializes the
         ROS publishers
49       cmd_pub_ = nh_.advertise<geometry_msgs::Twist>("nav_vel", 1);
50       line_sub_ = nh_.subscribe("output_line_estimated", 10,
         &RobotCommandAction::makeLine, this);
51       odom_sub_ = nh_.subscribe("odometry/gazebo",10,
         &RobotCommandAction::thorvaldOdomCB, this);
52       // Parameter values are loaded from the parameter server
53       nh_.getParam("/lin_vel_P_gain_float", Kv_);
54       nh_.getParam("/ang_vel_P_gain_float", Kh_);
55       nh_.getParam("/ang_vel_D_gain_float", Kd_);
```

```
56      nh_.getParam("/tip_of_cutter_length", knife_x_);
57      nh_.getParam("/tip_of_cutter_width", knife_y_);
58      nh_.getParam("/grass_right_side_bool", right_side_);
59      nh_.getParam("/loop_freq_double", freq_);
60
61      ROS_INFO("Starting action server..");
62      as_.start();
63      ROS_INFO("Waiting for requests..");
64  }
65
66  void thorvaldOdomCB(const nav_msgs::OdometryConstPtr& input_odo)
67  {
68      // This code updates the position of the
        robot's cutting tool tip in the Gazebo world frame
69
70      // Orientation is transformed to euler angles (roll,pitch,yaw)
71      tf::Quaternion quatOdom(
72          input_odo->pose.pose.orientation.x,
73          input_odo->pose.pose.orientation.y,
74          input_odo->pose.pose.orientation.z,
75          input_odo->pose.pose.orientation.w );
76      tf ::Matrix3x3 matrixOdom(quatOdom);
77      double roll, pitch, theta;
78      matrixOdom.getRPY(roll,pitch,theta);
79      thorOrientationRad_ = theta; // radians
80      thorOrientation_ = theta *180 / PI; // degrees
81
82      // The position of the cutting tool is rotated with reference to
        the orientation of the robot
83      double knife_x_rot = (cos(thorOrientationRad_)*knife_x_) -
        (sin(thorOrientationRad_)*knife_y_);
84      double knife_y_rot = (sin(thorOrientationRad_)*knife_x_) +
        (cos(thorOrientationRad_)*knife_y_);
85      baselinkPos_x_ = (input_odo->pose.pose.position.x);
86      grassCutterPos_x_ = (input_odo->pose.pose.position.x) +
        knife_x_rot;
87      baselinkPos_y_ = (input_odo->pose.pose.position.y);
88      if (right_side_ == true)
89      {
90          grassCutterPos_y_ = (input_odo->pose.pose.position.y) +
            knife_y_rot;
91      }
92      else
93      {
94          grassCutterPos_y_ = (input_odo->pose.pose.position.y) -
            knife_y_rot;
95      }
96  }
97
98  void makeLine(const geometry_msgs::PoseArray input)
```

```
99  {
100     // This code receives the two points that define the estimated
        separation line
101     // The points are rotated and pointHolder variables are updated
        for further error offset calculation
102
103     geometry_msgs::Point point1 = input.poses[0].position;
104     geometry_msgs::Point point1_rot;
105     geometry_msgs::Point point2 = input.poses[1].position;
106     geometry_msgs::Point point2_rot;
107
108     // Rotating point1_ and point2_ according to the robot's orientation in the global
        coordinate system
109     point1_rot.x = (cos(thorOrientationRad_)*point1.x) -
        (sin(thorOrientationRad_)*point1.y);
110     point1_rot.y = (sin(thorOrientationRad_)*point1.x) +
        (cos(thorOrientationRad_)*point1.y);
111
112     point2_rot.x = (cos(thorOrientationRad_)*point2.x) -
        (sin(thorOrientationRad_)*point2.y);
113     point2_rot.y = (sin(thorOrientationRad_)*point2.x) +
        (cos(thorOrientationRad_)*point2.y);
114
115     // Storing points for later use
116     pointHolder1_.x = point1_rot.x + baselinkPos_x_;
117     pointHolder1_.y = point1_rot.y + baselinkPos_y_;
118
119     pointHolder2_.x = point2_rot.x + baselinkPos_x_;
120     pointHolder2_.y = point2_rot.y + baselinkPos_y_;
121
122     point1.x = 0;
123     point1.y = 0;
124     point1.z = 0;
125     point2.x = 0;
126     point2.y = 0;
127     point2.z = 0;
128 }
129
130 // When a client sends an action request, this code is executed
131 void executeCB(const move_base_msgs::MoveBaseGoalConstPtr &goal)
132 {
133     ROS_INFO("Request received!");
134     // Calculating distance to client goal
135     double clientGoal = (sqrt(pow((goal->target_pose.pose.position.x-
        grassCutterPos_x_),2)));
136     // Defining containers
137     double error_y, error_o;
138     bool success = true;
139     float a1,a2,b1,b2,i_x,i_y;
140     int counter = 0;
```

```
141     // The current setpoint is defined
142     float goalPos1_x = pointHolder1_.x;
143     float goalPos1_y = pointHolder1_.y;
144     float goalPos2_x = pointHolder2_.x;
145     float goalPos2_y = pointHolder2_.y;
146
147     while (clientGoal > 0.0 && success == true)
148     {
149       //The distance to the setpoint is calculated
150       pointDistance_ = sqrt(pow((goalPos1_x-grassCutterPos_x_),2) +
          pow((goalPos1_y-grassCutterPos_y_),2));
151       // If the setpoint is too close (1m), wait for the next point
152       ros::Rate rate1(freq_);
153       while (pointDistance_ < 1.0)
154       {
155         // If the action is pre-empted, the loop is broken and the
          action is stopped
156         if (as_.isPreemptRequested() || !ros::ok())
157         {
158           ROS_INFO("%s: Preempted", action_name_.c_str());
159           as_.setPreempted();
160           success = false;
161           break;
162         }
163         // Update setpoint and calculate distance to it
164         goalPos1_x = pointHolder1_.x;
165         goalPos1_y = pointHolder1_.y;
166         goalPos2_x = pointHolder2_.x;
167         goalPos2_y = pointHolder2_.y;
168         pointDistance_ = sqrt(pow((goalPos1_x-
          grassCutterPos_x_),2) + pow((goalPos1_y-
          grassCutterPos_y_),2));
169         counter ++;
170         //If the robot remains idle for more than a user set time,
          it cancels the action
171         if (counter > 100)
172         {
173           ROS_INFO("%s: Preempted", action_name_.c_str());
174           as_.setPreempted();
175           success = false;
176           break;
177         }
178         rate1.sleep();
179       }
180       // When a setpoint that is further than 1m away is registered:
181       while (pointDistance_ >= 1.0)
182       {
183         // if the action is preempted, the loop is broken and the
          action is stopped
184         if (as_.isPreemptRequested() || !ros::ok())
```

```
185         {
186             ROS_INFO("%s: Preempted", action_name_.c_str());
187             as_.setPreempted();
188             success = false;
189             break;
190         }
191         // Calculate separation line equation: y = ax + b
192         a1 = ((goalPos2_y - goalPos1_y) / (goalPos2_x-
            goalPos1_x));
193         b1 = goalPos1_y - (a1 * goalPos1_x);
194         // Calculating line equation of orthogonal line
195         a2 = -(1/a1);
196         b2 = grassCutterPos_y_ - (a2*grassCutterPos_x_);
197         // Solving for y1 = y2 will give the point of intersection
            of the lines (i_x,i_y)
198         i_x = (b2 - b1)/(a1-a2);
199         i_y = a1*i_x + b1;
200
201         ros::Rate rate2(freq_);
202         // Update line/setpoint at a set interval, or when it gets
            within 1m of the goal point
203         // If at any point the action is preempted, the loop is
            broken and the action is stopped
204         for (int i=0; i<100 && pointDistance_ >= 1.0 &&
            clientGoal > 0.0; i++)
205         {
206             if (as_.isPreemptRequested() || !ros::ok())
207             {
208                 ROS_INFO("%s: Preempted",
                    action_name_.c_str());
209                 as_.setPreempted();
210                 success = false;
211                 break;
212             }
213
214             geometry_msgs::Twist cmd_msg;
215             // Calculating lateral displacement to separation line
216             error_y = (sqrt(pow((i_x-grassCutterPos_x_),2) +
                pow((i_y-grassCutterPos_y_),2)));
217             // Calculating angular displacement to setpoint
218             error_o = (atan((goalPos1_y-grassCutterPos_y_) /
                (goalPos1_x-grassCutterPos_x_))*180 / PI) -
                thorOrientation_;
219             // Proportional controller with a predefined gain
220             // The operating speed is set in the range of [0.6:1.5] m/s:
221             if (abs(error_y) < 1.0)
222                 cmd_msg.linear.x = 0.6 + Kv_ * error_y; // m*s^-1
223             // Lateral displacement greater than 1m means maximum
                operating speed:
224             else
```

```
225             cmd_msg.linear.x = 1.5;
226           cmd_msg.linear.y = 0.0;
227           cmd_msg.linear.z = 0.0;
228           // Angular velocity is controlled by a PD-controller
229           // The maximum angular speed is set to 0.3 rad/s
230           cmd_msg.angular.z = Kh_ * error_o + (Kd_ * ((error_o-
              prev_error_o_)/(1.0/freq_))); // rad/s
231           if (cmd_msg.angular.z > 0.3)
232             cmd_msg.angular.z = 0.3;
233           if (cmd_msg.angular.z < -0.3)
234             cmd_msg.angular.z = -0.3;
235           cmd_pub_.publish(cmd_msg);
236           pointDistance_ = sqrt(pow((goalPos1_x-grassCutterPos_x_),2) +
              pow((goalPos1_y-grassCutterPos_y_),2));
237           clientGoal = (sqrt(pow((goal-
              >target_pose.pose.position.x-grassCutterPos_x_),2)));
238           prev_error_o_ = error_o;
239           rate2.sleep();
240         }
241         if (success == false || clientGoal <= 0.0)
242           break;
243         else
244         {
245           int counter2 = 0;
246           goalPos1_x = pointHolder1_.x;
247           goalPos1_y = pointHolder1_.y;
248           goalPos2_x = pointHolder2_.x;
249           goalPos2_y = pointHolder2_.y;
250           pointDistance_ = sqrt(pow((goalPos1_x-
              grassCutterPos_x_),2) + pow((goalPos1_y-
              grassCutterPos_y_),2));
251
252           ros::Rate rate3(freq_);
253           while (pointDistance_ < 1.0)
254           {
255             goalPos1_x = pointHolder1_.x;
256             goalPos1_y = pointHolder1_.y;
257             goalPos2_x = pointHolder2_.x;
258             goalPos2_y = pointHolder2_.y;
259             pointDistance_ = sqrt(pow((goalPos1_x-
                grassCutterPos_x_),2) + pow((goalPos1_y-
                grassCutterPos_y_),2));
260             counter2 ++;
261             // If the robot remains idle for more than a user
                set time, it cancels the action
262             if (counter2 > 100)
263             {
264               ROS_INFO("%s: Preempted",
                  action_name_.c_str());
265               as_.setPreempted();
```

```
266                     success = false;
267                     break;
268                 }
269                 rate3.sleep();
270             }
271         }
272     }
273 }
274 // Sending a zero command to stop the robot
275 geometry_msgs::Twist zero_cmd_msg;
276 cmd_pub_.publish(zero_cmd_msg);
277
278 move_base_msgs::MoveBaseResult result;
279 as_.setSucceeded(result);
280 ROS_INFO("Action completed!");
281 }
282 };
283
284 int main(int argc, char** argv)
285 {
286   ros::init(argc, argv, "move_thorvald");
287   // The class is instantiated
288   RobotCommandAction move_thorvald("move_thorvald");
289   ros::spin();
290
291   return 0;
292 }
```

A.2 PointCloudSensor Class

This source code is located in the file *PointCloudSensor.h:*

```
1  #include <ros/ros.h>
2  #include <pcl_conversions/pcl_conversions.h>
3  #include <visualization_msgs/Marker.h>
4  #include <pcl/filters/passthrough.h>
5  #include <pcl/common/common.h>
6  #include <pcl/segmentation/sac_segmentation.h>
7  #include <tf/transform_listener.h>
8  #include "pcl_ros/transforms.h"
9  #include "pcl/common/angles.h"
10 #include <pcl/filters/extract_indices.h>
11 #include <geometry_msgs/PoseArray.h>
12 #define PI 3.14159265
13
14 #ifndef
PACKAGE_FIND_LINE_INCLUDE_PACKAGE_FIND_LINE_POINTCLOUDSENSOR_
H_
15 #define
```

```
PACKAGE_FIND_LINE_INCLUDE_PACKAGE_FIND_LINE_POINTCLOUDSENSOR_
H_
16
17 class PointCloudSensor
18 {
19 private:
20    ros::Publisher pcpub_line_;
21    ros::Publisher pcpub_plane_;
22    ros::Publisher marker_pub_;
23    ros::Publisher pose_pub_;
24    ros::Subscriber pcsub_;
25
26    // True if the robot has the uncut grass on its right side.
27    bool right_side_ = true;
28    float height_grass_ = 0.25;
29    float slope_ = 0.02;
30    tf::TransformListener tf_listener_;
31    int setOfLines_;
32    int AveragingSet_;
33    std::vector<geometry_msgs::Point> pointsToAverage1_;
34    std::vector<geometry_msgs::Point> pointsToAverage2_;
35    visualization_msgs::Marker sepLineMarker_;
36    // Defines the space between the points that form the line
37    int length_factor = 2;
38
39 public:
40    PointCloudSensor(ros::NodeHandle *nh)
41    {
42       // Subscribes to the point cloud topic and initializes the ROS publishers
43       pcsub_ = nh->subscribe("velodyne_points", 10,
         &PointCloudSensor::callbackPointCloudSensor, this);
44       pcpub_line_ = nh->advertise<sensor_msgs::PointCloud2> ("output_line", 1);
45       pcpub_plane_ = nh->advertise<sensor_msgs::PointCloud2>
         ("output_plane", 1);
46       marker_pub_ = nh->advertise<visualization_msgs::Marker>
         ("output_marker", 1);
47       pose_pub_ = nh-
         >advertise<geometry_msgs::PoseArray>("output_line_estimated", 1);
48
49       setOfLines_ = 0;
50       AveragingSet_ = 2;
51       pointsToAverage1_.clear();
52       pointsToAverage2_.clear();
53       //Variables get their values from parameters in a parameter server
54       nh->getParam("/height_grass_float", height_grass_);
55       nh->getParam("/field_slope_float", slope_);
56       nh->getParam("/length_of_line", length_factor);
57       nh->getParam("/grass_right_side_bool", right_side_);
58       nh->getParam("/avaraging_no_int", AveragingSet_);
```

```
59  };
60
61  sensor_msgs::PointCloud2 findPointCloudSensorPlane(const
    sensor_msgs::PointCloud2ConstPtr& sensorPCloud)
62  {
63    // This code reads the point cloud data and applies a RANSAC model to extract
      relevant indices
64    // Returns and publishes a point cloud with the segmented dominant plane of the
      uncut grass
65
66    // Defining containers
67    sensor_msgs::PointCloud2 cloud_tf;
68    pcl::PCLPointCloud2* cloud = new pcl::PCLPointCloud2;
69    pcl::PCLPointCloud2ConstPtr cloudPtr(cloud);
70    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud2;
71    cloud2 = pcl::PointCloud<pcl::PointXYZ>::Ptr (new
      pcl::PointCloud<pcl::PointXYZ>);
72
73   // Transforming sensor coordination-system to base_link
74   pcl_ros::transformPointCloud ("base_link", *sensorPCloud, cloud_tf,
     tf_listener_);
75   // Convert to PCL data type to be able to use filtering algorithms
76   pcl_conversions::toPCL(cloud_tf, *cloud);
77
78   // Applying initial pass-through filtering to remove ground and
     unwanted data
79   pcl::PassThrough<pcl::PCLPointCloud2> criticalZone;
80   criticalZone.setInputCloud (cloudPtr);
81
82   criticalZone.setFilterFieldName("z");
83   criticalZone.setFilterLimits(height_grass_,2.5);
84   criticalZone.filter(*cloud);
85
86   criticalZone.setFilterFieldName("y");
87   criticalZone.setFilterLimits(-3,3);
88   criticalZone.filter(*cloud);
89
90   criticalZone.setFilterFieldName("x");
91   criticalZone.setFilterLimits(2,4.5);
92   criticalZone.filter(*cloud);
93   pcl::fromPCLPointCloud2(*cloud, *cloud2);
94
95   // Segment out dominant grass-plane
96   pcl::PointCloud<pcl::PointXYZ>::Ptr plane_cloud (new
     pcl::PointCloud<pcl::PointXYZ>);
97   pcl::ModelCoefficients::Ptr grassPlaneCoefs (new
     pcl::ModelCoefficients);
98   pcl::PointIndices::Ptr indPlane (new pcl::PointIndices);
99   pcl::SACSegmentation<pcl::PointXYZ> seg;
100  seg.setOptimizeCoefficients(true);
```

```
101     // Estimate plane using a PLC RANSAC algorithm.
102     seg.setModelType(pcl::SACMODEL_PERPENDICULAR_PLANE);
103     seg.setMethodType(pcl::SAC_RANSAC);
104     seg.setMaxIterations (1000);
105     // Setting threshold to 15 cm
106     seg.setDistanceThreshold(0.15);
107     seg.setInputCloud(cloud2);
108     // Defining the plane to be perpendicular to the z-axis, 20-degree slope tolerance
109     Eigen::Vector3f zAxisPerp;
110     zAxisPerp << 0, 0, 1;
111     seg.setAxis(zAxisPerp);
112     seg.setEpsAngle(pcl::deg2rad(20.0));
113     // grassPlaneCoefs will contain the coefficients of the plane: ax + by + cz + d = 0
114     seg.segment(*indPlane, *grassPlaneCoefs);
115     if (indPlane->indices.size() == 0)
116     {
117        ROS_ERROR("Unable to find surface.");
118     }
119
120     // Extracting the plane into plane_cloud and plane_cloudPCL
121     pcl::ExtractIndices<pcl::PointXYZ> grassPlaneExtraction;
122     grassPlaneExtraction.setInputCloud(cloud2);
123     grassPlaneExtraction.setIndices(indPlane);
124
125     // All indices are returned, including those contained in "indices"
126     grassPlaneExtraction.setNegative (false);
127     grassPlaneExtraction.filter (*plane_cloud);
128     pcl::PCLPointCloud2* plane_cloudPCL = new pcl::PCLPointCloud2;
129     pcl::PCLPointCloud2ConstPtr cloudPtr2(plane_cloudPCL);
130     pcl::toPCLPointCloud2(*plane_cloud, *plane_cloudPCL);
131
132     // Finally, the plane is returned and published
133     sensor_msgs::PointCloud2 output_plane;
134     pcl_conversions::fromPCL(*plane_cloudPCL, output_plane);
135     output_plane.header.frame_id="base_link";
136     output_plane.header.stamp=sensorPCloud->header.stamp;
137     pcpub_plane_.publish(output_plane);
138     return output_plane;
139 }
140
141 sensor_msgs::PointCloud2 readPointCloudSensorLine(const
    sensor_msgs::PointCloud2ConstPtr& sensorPCloud)
142 {
143    // This code will take in the segmented plane of the uncut grass in front of Thorvald
144    // Returns and publishes a point cloud of points outlining the separation line
145
146    // Defining containers
147    pcl::PCLPointCloud2* cloud = new pcl::PCLPointCloud2;
148    pcl::PCLPointCloud2ConstPtr cloudPtr(cloud);
149    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud2;
```

```
150  cloud2 = pcl::PointCloud<pcl::PointXYZ>::Ptr (new
     pcl::PointCloud<pcl::PointXYZ>);
151  pcl::PointXYZ minPt, maxPt;
152  pcl::PointCloud<pcl::PointXYZ> cloud_seg;
153  //Fetching the point cloud-plane
154  sensor_msgs::PointCloud2 output_plane =
     findPointCloudSensorPlane(sensorPCloud);
155
156  // Filtering the point cloud in segments, each segment 10 cm wide
157  for (float i = 0.0; i < 2.5; i+= 0.1 )
158  {
159     // Convert to PCL data type
160     pcl_conversions::toPCL(output_plane, *cloud);
161     pcl::PassThrough<pcl::PCLPointCloud2> criticalZoneSeg;
162     criticalZoneSeg.setInputCloud (cloudPtr);
163     criticalZoneSeg.setFilterFieldName("x");
164     criticalZoneSeg.setFilterLimits(2+i,2.1+i);
165     criticalZoneSeg.filter(*cloud);
166
167     // Storing the min & max valued points (if there are points)
168     pcl::fromPCLPointCloud2(*cloud, *cloud2);
169     size_t num_points = cloud2->size();
170     if (num_points > 0)
171     {
172        pcl::getMinMax3D(*cloud2, minPt, maxPt);
173        pcl::toPCLPointCloud2(*cloud2, *cloud);
174        // Selecting points depending on where the separation line is
175        if (right_side_ == true)
176        {
177           criticalZoneSeg.setFilterFieldName("y");
178           criticalZoneSeg.setFilterLimits(maxPt.y,maxPt.y);
179           criticalZoneSeg.filter(*cloud);
180           pcl::fromPCLPointCloud2(*cloud, *cloud2);
181           cloud_seg += *cloud2;
182        }
183        else
184        {
185           criticalZoneSeg.setFilterFieldName("y");
186           criticalZoneSeg.setFilterLimits(minPt.y,minPt.y);
187           criticalZoneSeg.filter(*cloud);
188           pcl::fromPCLPointCloud2(*cloud, *cloud2);
189           cloud_seg += *cloud2;
190        }
191        // Cloud_seg now holds the desired line of points
192     }
193  }
194  // Converting final point cloud back to PCLPointCloud2 format
195  pcl::toPCLPointCloud2(cloud_seg, *cloud);
196  // Converting to ROS data type, returns and publishes the point cloud
197  sensor_msgs::PointCloud2 output_line;
```

```
198     pcl_conversions::fromPCL(*cloud, output_line);
199     output_line.header.frame_id="base_link";
200     output_line.header.stamp=sensorPCloud->header.stamp;
201     pcpub_line_.publish(output_line);
202     return output_line;
203 }
204
205 void callbackPointCloudSensor(const sensor_msgs::PointCloud2ConstPtr&
    sensorPCloud)
206 {
207     // This function estimates the line with a RANSAC model
208     // Publishes two points that define the line, as well as a marker
209
210     // Defining containers
211     pcl::PCLPointCloud2* cloud = new pcl::PCLPointCloud2;
212     pcl::PCLPointCloud2ConstPtr cloudPtr(cloud);
213     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud2;
214     cloud2 = pcl::PointCloud<pcl::PointXYZ>::Ptr (new
        pcl::PointCloud<pcl::PointXYZ>);
215     geometry_msgs::Point point1;
216     geometry_msgs::Point point2;
217
218     // Generating a set number of lines for averaging
219     if(setOfLines_ < AveragingSet_)
220     {
221         // Fetching the pointcloud holding the line of points
222         sensor_msgs::PointCloud2 cloud_line =
            readPointCloudSensorLine(sensorPCloud);
223         pcl_conversions::toPCL(cloud_line, *cloud);
224         pcl::fromPCLPointCloud2(*cloud, *cloud2);
225
226         //Apply RANSAC algorithm to fit line, threshold set to 8 cm.
227         size_t num_points = cloud2->size();
228         if (num_points > 0)
229         {
230             pcl::ModelCoefficients::Ptr sepLine (new
                pcl::ModelCoefficients);
231             pcl::PointIndices::Ptr indSepLine(new pcl::PointIndices);
232             pcl::SACSegmentation<pcl::PointXYZ> seg2;
233             seg2.setOptimizeCoefficients(true);
234             seg2.setModelType(pcl::SACMODEL_LINE);
235             seg2.setMethodType(pcl::SAC_RANSAC);
236             seg2.setDistanceThreshold(0.08);
237             seg2.setInputCloud(cloud2);
238             seg2.segment(*indSepLine, *sepLine);
239
240             if (indSepLine->indices.size () == 0)
241             {
242                 ROS_ERROR ("Could not estimate a LINE model.");
243             }
```

```
244
245         else
246         {
247             // Defining quaternions from the direction coefficients
248             tf::Quaternion quat =
                tf::createQuaternionFromRPY(sepLine->values[3],
                sepLine->values[4], sepLine->values[5]);
249             // Creating a LINE_STRIP marker_ to visualize the line
250             sepLineMarker_.ns = "vehicle_orientation";
251             sepLineMarker_.header.frame_id = "base_link";
252             sepLineMarker_.header.stamp = ros::Time::now();
253             sepLineMarker_.type =
                visualization_msgs::Marker::LINE_STRIP;
254             sepLineMarker_.action =
                visualization_msgs::Marker::ADD;
255             sepLineMarker_.pose.orientation.w = quat.w();
256             sepLineMarker_.id = 0;
257             sepLineMarker_.scale.x = 0.05;
258             sepLineMarker_.color.b = 1.0;
259             sepLineMarker_.color.a = 1.0;
260
261             // Defining two points to draw a line between
262             geometry_msgs::Point p1;
263             p1.x = sepLine->values[0] - (length_factor*quat.x());
264             p1.y = sepLine->values[1] - (length_factor*quat.y());
265             p1.z = sepLine->values[2] - (length_factor*quat.z());
266             geometry_msgs::Point p2;
267             p2.x = sepLine->values[0] + (length_factor*quat.x());
268             p2.y = sepLine->values[1] + (length_factor*quat.y());
269             p2.z = sepLine->values[2] + (length_factor*quat.z());
270
271             // Placing the points in a vector for averaging
272             pointsToAverage1_.push_back(p1);
273             pointsToAverage2_.push_back(p2);
274             setOfLines_++;
275         }
276     }
277 }
278 else
279 {
280     // Averaging the lines to make it less volatile and more robust
281     for (int i=0; i < AveragingSet_; i ++)
282     {
283         point1.x += pointsToAverage1_[i].x;
284         point1.y += pointsToAverage1_[i].y;
285         point1.z += pointsToAverage1_[i].z;
286         point2.x += pointsToAverage2_[i].x;
287         point2.y += pointsToAverage2_[i].y;
288         point2.z += pointsToAverage2_[i].z;
289     }
```

```
290         sepLineMarker_.points.clear();
291         point1.x = point1.x / (float)pointsToAverage1_.size();
292         point1.y = point1.y / (float)pointsToAverage1_.size();
293         point1.z = point1.z / (float)pointsToAverage1_.size();
294
295         point2.x = point2.x / (float)pointsToAverage2_.size();
296         point2.y = point2.y / (float)pointsToAverage2_.size();
297         point2.z = point2.z / (float)pointsToAverage2_.size();
298
299         pointsToAverage1_.clear();
300         pointsToAverage2_.clear();
301         setOfLines_ = 0;
302         sepLineMarker_.lifetime = ros::Duration();
303         // Pushing back the two points and publishing marker of line
304         sepLineMarker_.points.push_back(point1);
305         sepLineMarker_.points.push_back(point2);
306         marker_pub_.publish(sepLineMarker_);
307
308         geometry_msgs::PoseArray p_array;
309         // Timestamp: when the message was created
310         p_array.header.stamp=sensorPCloud->header.stamp;
311         // Frame ID related to the points
312         p_array.header.frame_id = "base_link";
313         geometry_msgs::Pose firstPointLine;
314         geometry_msgs::Pose secondPointLine;
315         firstPointLine.position = point1;
316         secondPointLine.position = point2;
317         // The two points are pushed in array and published
318         p_array.poses.push_back(firstPointLine);
319         p_array.poses.push_back(secondPointLine);
320         pose_pub_.publish(p_array);
321       }
322     }
323 };
324#endif
```

A.3 Parameter Server

This source code is located in the file *grass_cutter_params.yaml:*

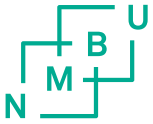
```
1  lin_vel_P_gain_float: 0.5
2  ang_vel_P_gain_float: 0.032
3  ang_vel_D_gain_float: 0.001
4  tip_of_cutter_width: 0.9
5  tip_of_cutter_length: 0.3
6  grass_right_side_bool: true
7  height_grass_float: 0.25
8  field_slope_float: 0.02
9  length_of_line: 2
```

```
10 use_sim_time: true
11 loop_freq_double: 10.0
12 avaraging_no_int: 2
```

A.4 Launch File

This source code is located in the file *autonomous_cutter.launch:*

```
1  <?xml version="1.0" ?>
2  <!-- This file launches the two nodes find_line and move_thorvald,
   as well as load parameter server -->
3
4  <launch>
5  <!-- Loading parameters from the YAML-file -->
6  <rosparam file="$(find package_find_line)/config/grass_cutter_params.yaml"
   />
7
8  <!-- Starting the two nodes -->
9  <node name="line_node" pkg="package_find_line"
   type="package_find_line_node"/>
10 <node name="move_thorvald" pkg="package_find_line" type="move_thorvald"/>
11 </launch>
```



Norges miljø- og biovitenskapelige universitet
Noregs miljø- og biovitskapelege universitet
Norwegian University of Life Sciences

Postboks 5003
NO-1432 Ås
Norway