Master's thesis

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

Marte Nordbotten Ruud Olsen

# Path planning of a nRF52850 robot: return home without server input

Master's thesis in Cybernetics and Robotics
Supervisor: Tor Onshus
May 2023

NTNU
Norwegian University of
Science and Technology

Marte Nordbotten Ruud Olsen

# Path planning of a nRF52850 robot: return home without server input

Master's thesis in Cybernetics and Robotics
Supervisor: Tor Onshus
May 2023

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

# Summary and conclusion

This thesis was written as a part of a larger project, the *SLAM Robot Project*, which aims to use a team of collaborative mobile robots to explore and map an unknown environment.

The main purpose of this thesis was to implement a way for the robot to get back to its initial position without using inputs from the server, a process referred to as the *homing procedure*. The first approach was to have the robot backtrack its previous targets and thus return to the start position. This functionality worked as intended, and the robot will trace its previous movements and stop when the start position is reached. If the same area has been visited more than once, the robot will find the shortest way home, and not go by unnecessary destinations.

While working with the homing procedure, it became clear that an improvement in the robot's ability to drive straight ahead was necessary. Two PID controllers were tuned, and the resulting behaviour was studied. Additionally, the effect of the maximum heading correction value was examined. These efforts resulted in an improved movement and the paths created by the robot became a lot cleaner. However, the robot did still drift to the right. To combat this, a more frequent readjustment of the path was implemented. This caused the robot to reach its destinations faster as it went less off course.

The next step was to have the initialization of the homing procedure cause the robot to drive straight back to its initial position. During this drive the robot needed to be able to detect obstacles and avoid them. There exist a lot of possible ways to implement obstacle avoidance, however, the state of the robot and its limited memory makes it unrealistic to implement a complex variant at this point. A simpler version was chosen, where the robot will start backtracking if an obstacle is detected. This procedure was tested and worked as intended, and the robot will return home the way it came. However, the robot is not capable of stopping in time to avoid crashing into the obstacle. Different detection distances were attempted, but while they affected the behaviour, neither was able to prevent the robot from hitting the obstacle.

The robot displayed some unexpected behaviour during driving. It became apparent that an improvement of the robot's position estimate is necessary, as it is prone to drift and occasionally fails, causing the robot to drive in the wrong direction. A combination of these elements, and the fact that the code base is not well structured, caused everything to take longer than expected.

In addition to implementing these features, a collaborative effort was made with the students Tran, Berg, and Kolbeinsen to improve the workflow and code quality of the *SLAM Robot Project*. A GitHub organization was created, and the project code was separated and put into seven repositories. The repositories were cleaned up, given functional *gitignore* files, and all unnecessary files were deleted. Documentation was created to gather all the relevant information in one place. A naming convention was created and implemented for the robot code. While the system was greatly improved through this work, there remains some major refactoring to get the project where it needs to be.

# Oppsummering og konklusjon

Denne oppgaven er skrevet som en del av et større prosjekt, kalt *SLAM Robot Project*, som har et overordnet mål om å bruke flere samarbeidene roboter til å utforske og kartlegge et ukjent område.

Hovedformålet med denne avhandlingen har vært å implementere en måte for roboten til å komme seg tilbake til startpunktet sitt uten hjelp av serveren, en prosess som er referert til som *homing procedure*. Den første løsningen var å få roboten å returnere til sine tidligere destinasjoner etter tur, og på den måten komme seg tilbake til start. Denne metoden fungerer som den skal, og roboten er i stand til å kjøre innom alle posisjonene den skal til og stoppe når den har nådd startposisjonen. Hvis det samme området har blitt besøkt flere ganger, er vil roboten finne korteste vei hjem og unngå unødvendige stopp.

Gjennom denne jobbing ble det klart at det var nødvendig å forbedre robotens evne til å kjøre rett fram. Parameterne til to PID-regulatorer ble tilpasset og den resulterende oppførselen analysert. I tillegg ble effekten av maksimalverdien til retningskorrigeringen undersøkt. Disse tiltakene førte til en forbedret oppførsel og robotens bevegelse ble enklere å følge, men roboten drev fortsatt mot høyre. For å redusere effekten av dette ble koden tilpasset slik at roboten justerer retningen sin oftere. Dette førte til at roboten nådde målene sine fortere og avvek mindre fra kursen.

Det neste steget var å få roboten til å gå direkte tilbake til start så fort returneringsprosessen ble iverksatt. På vei hjem måtte roboten være i stand til å oppdage og unngå hindringer. Det finnes mange metoder for å unngå hindre, men robotens nåværende tilstand og dens begrensede minnekapasitet gjør at veldig komplekse metoder er utelukket på dette tidspunktet. I stedet ble en enklere variant valgt, hvor roboten begynner å returnere til tidligere destinasjoner dersom et hinder blir oppdaget. Denne funksjonaliteten ble testet og fungerer som den skal, og roboten er dermed i stand til å detektere et hinder, for så å gå tilbake til start samme vei den kom. Den er derimot ikke i stand til å stoppe tidsnok til å unngå å kollidere med hinderet. Ulike deteksjonsavstander ble testet, men selv om de påvirket robotens bevegelse var det ikke nok til å unngå at roboten kolliderte.

Roboten viste enkelte ganger en uventet oppførsel mens den kjørte. Det ble klart at det vil være nødvendig å forbedre robotens posisjonsestimat, ettersom denne drifter og enkelte ganger slutter å funke, noe som får roboten til å kjøre i feil retning. I kombinasjon med en rotete kode gjør dette at alt tok lengre tid enn forventet.

I tillegg til å implementere denne funksjonaliteten ble kodekvaliteten til prosjektet forsøkt forbedret i samarbeid med studentene Tran, Berg, og Kolbeinsen. En GitHub-organisasjon ble opprettet, og prosjektkoden ble splittet og plassert i sju separate repositorier. Disse ble ryddet i, gitt funksjonelle *gitignore*-filer, og alle unødvendige filer ble slettet. Det ble laget en dokumentasjon for å samle all informasjon på ett sted. Retningslinjer for navngivning ble lagd, og koden tilpasset til å samsvare med disse. Systemet ble betraktelig mer oversiktlig etter disse forbedringene, men det er fortsatt mye som gjenstår før kvaliteten er så god som er ønsket.

# Problem statement

The robots used in this project are usually controlled by a C++ server which tells the robot where to go. The overall task of this thesis was to make the robot capable of returning to its initial position on its own, in case something was to happen. In order for this goal to be reached, several smaller tasks, which are listed below, must work. All implemented functionality will need to be tested thoroughly to ensure it functions as expected.

- The robot should be able to give itself a new target position instead of relying on the server.
- The robot should be able to return to its initial position by itself.
- The robot will need to be able to reach the targets it sets.
- When going home, it should detect and avoid potential obstacles and still reach its destination.

In addition to solving this individual task, a collaborative task was assigned to a group of four master students. The students will all use the same hardware and access the same code. The group's purpose is to improve the code by increasing its readability and making the project easier to maintain, understand, and develop further. The steps that are achievable within the given time frame will need to be implemented, while the more time-consuming tasks ought to be suggested for future students. The improvements done will need to be well documented and justified.

# Preface

This thesis has been written as a master's thesis at the Norwegian University of Science and Technology in the spring of 2023. The thesis terminates five years of studying Cybernetics and Robotics.

I would like to thank my supervisor Tor Onshus for guidance throughout the project. A special thanks to my fellow students on the SLAM Robot Project for their help, useful discussions, and general moral support. I would also like to thank my family and friends for their support throughout these five years, and for making my time in Trondheim so great.

Marte Nordbotten Ruud-Olsen

Trondheim, May 31, 2023

# Contents

# List of Tables

# List of Figures

# Acronyms

|  |  |
|---|---|
| **EKF** | Extended Kalman Filter. 6, 14, 59 |
| **GUI** | Graphical User Interface. 7 |
| **IMU** | Inertial Measurement Unit. 4, 6, 14, 26 |
| **IR** | Infra-Red. 3, 4, 6, 7, 20, 21, 29, 44, 50 |
| **MQTT** | Message Queuing Telemetry Transport. 7, 8 |
| **MQTT-SN** | MQTT for Sensor Network. 3, 4, 6–8, 59 |
| **PID** | Proportional-Integral-Derivative. i, ii, viii, 6, 19, 30, 32, 33, 35, 38, 39 |
| **RRT** | Rapidly-exploring Random Tree. 10 |
| **RTOS** | Real-Time Operating System. 6 |
| **SDK** | Software Development Kit. 59 |
| **SES** | Segger Embedded Studio. 4, 59 |
| **SLAM** | Simultaneous Localisation And Mapping. ii, 1, 3, 9, 10, 20 |
| **TCP/IP** | Transmission Control Protocol/Internet Protocol. 7, 8 |
| **UDP** | User Datagram Protocol. 8 |
| **VFF** | Virtual Force Field. 11 |

**VFH** Vector Field Histogram. 11, 12

**vSLAM** Visual SLAM. 9

**WSN** Wireless Sensor Networks. 8

Chapter 1

# Introduction

## 1.1 Motivation

Autonomous mobile robots present a huge area of development and possible applications. They can be an important resource for the industry, for example used in the inspection of complex structures both in the air [21] and underwater [17], where human action would be either dangerous or difficult. Mobile robots are also becoming a large part of everyday life, as seen in the rise of autonomous lawnmowers and vacuums in recent years. Transportation is another area of interest, as cars are equipped with more self-driving features [23] and research has started to get fully autonomous busses [24].

Indoor navigation poses an extra challenge when it comes to the localization of the robot, as there is typically no GPS available. In order to be able to map its surroundings as it explores, the robot will need to solve the Simultaneous Localisation And Mapping (SLAM) problem. This is a well-studied problem and several different solutions have been proposed [14].

Using several robots concurrently able to communicate can improve the exploration procedure. When done correctly, using several robots in collaboration can greatly reduce the exploration time, as the robots can explore different areas [12]. Using several robots at once can also improve the position estimate of each robot, as they can rely on the positions of the other robots when calculating their own [15].

This thesis is part of a bigger project which aims to use several collaborative wheeled robots using SLAM to explore and map an unknown indoor area. This is part of an exciting future, combining several interesting areas to solve an essential task. Creating a homing procedure will help to drive the overall project in the direction of a functional setup by making it more autonomous.

## 1.2 Report outline

In Chapter 2 the project is described as it was at the start of this thesis and the system setup is explained. Chapter 3 follows with theory for some relevant topics. The implementation is described in Chapter 4, while the results are presented and discussed in Chapter 5. Discussion surrounding the quality of the code base and its improvements are found in Chapter 6. Chapter 7 presents possible further work. The documentation created during this work can be found as an appendix.

# Chapter 2

# Background

## 2.1 Previous work

The system setup and the code base used in this thesis are the results of several previous projects and master theses. Since the project's beginning in 2004 with a simple LEGO robot and some distance sensors [11], more than 40 students have participated in the project. It has evolved from a LEGO robot with basic functionality, to a unique setup with more sensors, circuit boards, and functionality.

The results of these efforts are 12000 lines of poorly documented code [32]. As each student adds their code on top of the already existing system, the project code has a lot of dependencies and non-intuitive locations. Version control was not added until autumn 2022, and as students have worked on different features in parallel, they are not all merged together and available in the same place.

The long-term goal of these projects is to create a system where several small robots may work together with a server to map out and explore an unknown environment with obstacles. It should include SLAM and obstacle avoidance. There are two servers available, one written in Java and one in C++. This thesis only concerns itself with the latter.

Some noticeable work includes the work done in Stenset [29]. The thesis focused on analyzing and improving the behaviour of the robot code, based on feedback from previous students and tests done as a part of the thesis. In addition, a basic collision avoidance procedure was added for when the robot is moving forwards.

Mullins [28] implemented an online SLAM algorithm on the server, allowing the robots to explore and map an area, both with a real robot and in simulation. The algorithm uses the input from the IR sensors, and was created with a focus on scalability and computational efficiency.

Andersen [31] created a multi-threaded MQTT-SN client for the robot, using a Raspberry

Pi as a broker. This allowed the C++ server to communicate with the robots by MQTT-SN instead of Bluetooth as previously done. A detailed description of the hardware can also be found in Andersen's work, some of which are described in Section 2.2.1.

In his project, Frestad [32] presents a detailed guide for the setup of the C++ server and the network that was used in the initial phase of this project. In addition, there are two sections, one each for the robot and the C++ server, explaining the main functionality of their respective software.

## 2.2 Setup

The setup described in this section shows the status of the robot at the beginning of this thesis. The hardware was not adapted in any way throughout the project. The changes made to the software functionality as a part of this thesis are described in Section 4.

The basis of this thesis is the result of Frestad's work in [32]. The system consists of a server, a robot and a Raspberry Pi working as a gateway and broker. The broker makes sure that the necessary data get from the publishers to the subscribers, while the gateway translates between the two communication protocols used, as described further in Section 2.2.4.

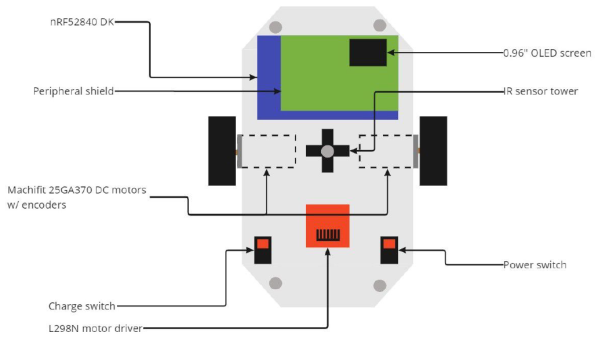
The setup is created to handle several robots, as is the goal of the overall project. There are six robots currently fit to be used on the project, named NRF1 through NRF6. The work in this thesis is mainly done on the robot NRF2.

The robot program is written in C and is designed on a nRF52840-DK card made by Nordic Semiconductor [43]. The server is written in C++ and can be run through Visual Studio 2019, which can be downloaded from [39]. The simulator is also available through the C++ server code base. Segger Embedded Studio (SES) v5.68 for Windows [42] is used for programming and debugging the nRF52840-DK. Motion capture of the robot is done using the software and setup provided by OptiTrack Motive [40].

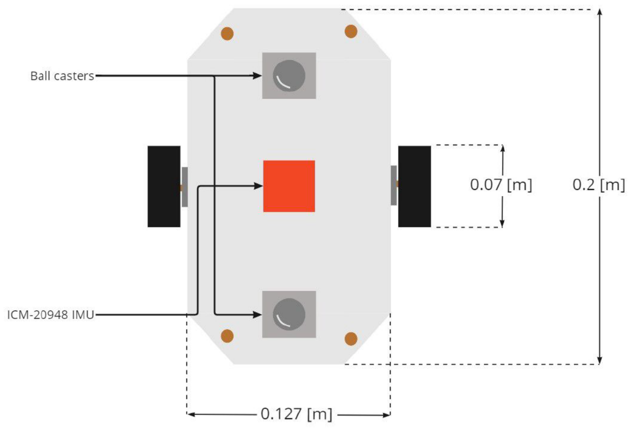
### 2.2.1 Hardware

The robot consists of a body frame with two motors, each controlling one wheel, and two ball casters for stabilizing the robot. Three circuit boards are present to control the robot. In addition, there is a battery, a rotating tower for IR readings, and an IMU. Five small, silver orbs are there to enable tracking by OptiTrack Motive. An overview of the robot can be seen in Figure 2.1.

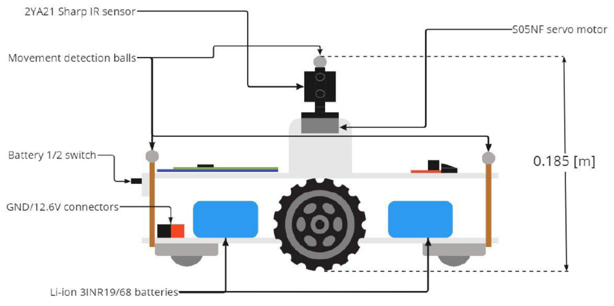
The IMU used, located below the robot close to the ground, is the ICM-20948 [34], consisting of a gyroscope, accelerometer and compass, all in 3-axis. The rotating tower on top of the robot has four IR sensors attached to it, namely the 2Y0A21 sensor from Sharp [44]. As is it necessary for the IR readings to be taken in all directions, this tower is connected to a motor allowing it to rotate the necessary 90 degrees. The servo motor used is the S05NF STD.



**(a)** Top view of the robot. The front is facing down.



**(b)** Bottom view of the robot



**(c)** Side view of the robot

**Figure 2.1:** Hardware overview of the robot. The figures are taken from [31] page 10.

There are three boards present on the robot. The one in the front is the motor driver, namely the UK1122 motor driver from CanaKit [33]. This driver is inspired by the L298 H-Bridge Motor Driver, and is well-suited for robotic applications. The motors are 12V DC equipped with encoders to allow for angle readings [31].

The two boards at the back of the robot are connected and work together to control the robot. The green one on the top functions simply as a connection with the board below and the rest of the robot. The board used for the main functionality of the robot is the PCA10065 board for the nRF52840 Development Kit by Nordic Semiconductors [43].

### 2.2.2 Software base: Robot

The code for controlling the robots is written in $C$ and is implemented using the Real-Time Operating System *FreeRTOS*. Created for use in real-time applications, it is ideal for a system like this. Code created in this manner is structured around several tasks, where each task has different responsibilities. The different tasks can be run at practically the same time [37]. This sort of setup is appropriate for situations where the system has to react to outside stimuli, such as buttons or sensor input. A fixed priority ensures consistent reactions in the system.

There are five main tasks present in the code for the robot [31], responsible for respectively controlling the pose, controlling the motor speed, pose estimation, IR sensor readings, and mapping [32]. In addition, there are two smaller tasks: $thread\_stack\_task$ and $mqttsn\_task$. The former handles the different threads. The latter is responsible for the MQTT-SN communication with the gateway, as described further in Section 2.2.4.

The pose controller task (*vPoseControllerTask*), receives a target position given by the server through the MQTT-SN task, and calculates the difference between the desired position and current position based on a pose estimate from the pose estimator task. This results in a straight line between the robot and its target. The robot will then rotate until it aligns itself with the error line, and then drives the length of the line. As it is driving it will also check for collision. If a possible collision is detected, or the robot has driven further than the error distance, it will stop. Both the process of rotating and of driving forwards uses a PID regulator to calculate the power of the left and right motor.

The task responsible for the speed of the motors, *vMotorSpeedControllerTask*, implements a PID controller. Readings from the encoders on the wheels are used to calculate the necessary speed of the robot. The power calculated by the pose controller is used as a reference in the regulator.

The pose is estimated in *vNewMainPoseEstimatorTask*. Here, data from the IMU and the encoders are collected, and an Extended Kalman Filter (EKF) is used to create a good estimate of the pose.

The sensor tower task (*VMainSensorTowerTask*) is responsible for scanning the surroundings with the IR sensors attached to the tower. The tower is capable of rotating 90 degrees, and a measurement is taken each degree. The tower is only rotating when the robot is stationary. The measurements are then sent to the server through the MQTT-SN task.

The mapping task (*mapping_task*) uses the measurements from the IR sensors and the pose of the robot to create a map of its surroundings. In order to achieve this, a line extraction is performed and the resulting line segment is added to the map. Greater details can be found in [31], as the task was created as a part of that thesis.

### 2.2.3 Software base: Server

The server is written in C++ and is a multi-threaded program dealing with keeping track of the robots and their sensor inputs, generating new positions for the robots, and creating a map of the surroundings based on the robot inputs.

The server generates a Graphical User Interface (GUI) consisting of two parts: a control panel and a blank map. The control panel has quite a lot of functionality, such as allowing the users to control the robot remotely by setting manual targets, seeing which topics the server subscribes to, and adding new robot connections. The blank map becomes increasingly detailed as the robot explores an area.

The server code also includes a simulator, which can be initialized through the control panel. By setting a target on the empty map, the simulated robot will move in that direction and its sensor input will start to create a map of the surroundings. As an alternative to setting manual targets, the box "Set auto simulation path" can be checked and the simulated robot will explore the entire area it is given. The resulting measurements will create an accurate map.

In addition to the GUI and running the simulation, the server is responsible for creating new targets that the robot should reach. If manual input is turned on in the GUI, the server will simply pass this target on to the robot. Otherwise, the server sends random targets in the unexplored area.

[32] describes the concurrent threads in the server as the *gui_thread* dealing with the GUI, the *robot_simulation_thread* responsible for the simulation, and the *mqtt_thread* used for managing the communication with the MQTT-broker. The two last threads, *robot_inbox_thread* and *robot_outbox_thread*, respectively update the robot state based on incoming messages and creates new commands for the robot to execute. In addition to this multi-threaded process, the server has to keep track of the robots and their states.

### 2.2.4 Robot-server communication

The server and robots communicate through some protocols called MQTT and MQTT-SN. Message Queuing Telemetry Transport (MQTT) is a publish/subscribe messaging protocol designed to work with networks using TCP/IP. The protocol has many advantages and use cases, as the clients are compact and can be used even on small microcontrollers [41]. The clients can subscribe and publish to topics, where each topic deals with a certain kind of data. The clients that are interested in this data will subscribe to the topic, and will then receive everything that is published.

Between the clients is the MQTT broker, which is responsible for making sure that all messages get where they are supposed to go. The broker, sometimes known as a server, deals with the subscribe and unsubscribe requests from clients, opens and closes the network connections, and matches the incoming published messages to the right clients [26].

A version of the MQTT is the MQTT for Sensor Network (MQTT-SN). It uses UDP instead of TCP/IP, and is specifically designed to work well with Wireless Sensor Networks (WSN)s [13]. Additionally, it is well suited for appliances with limited storage and processing power [20]. The MQTT-SN clients will connect themselves to MQTT-SN Gateway, which is then connected to a MQTT broker. The gateway is responsible for translating between the two protocols.

The setup in this project can be seen in Figure 2.2. The robots function as the MQTT-SN clients, while the server is the MQTT client. Between them there is a broker and a gateway. These are installed on a Raspberry Pi 3b+, with a nRF52840 dongle connected via USB. The broker used is the Mosquitto MQTT broker, version 2.0.14, which is open source and available at [35]. The MQTT-SN gateway is from Eclipse Paho [36], and was installed as a part of the Thread Border Router image from Nordic Semiconductor.

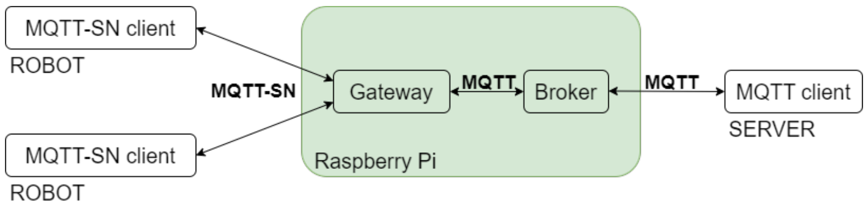


**Figure 2.2:** The MQTT and MQTT-SN setup. This figure describes the connection between the different clients, gateway and broker.

In the robot, it is the *mqttsn_task* that handles the messages. It receives message queues from the other tasks and deals with passing these on to the server. Pose estimate and data readings from the sensor tower as amongst the things sent to the server. Incoming messages are handled by the same task. These messages are being sent to the *vMainPoseControllerTask*, which deals with the information.

On the server side, it is the *mqtt_thread* that deals with the communication with the broker. If there is a simulation running, the *robot_simulation_thread* is also involved.

Chapter 3

# Theory

## 3.1 Path planning

The ability to safely navigate through its workspace is an essential task for any autonomous mobile robot. The path planning algorithm used has to ensure safety and feasibility in addition to being adapted to the purpose of the robot. Robots such as lawnmowers or drones designed for inspecting structures need to guarantee full coverage of their designated areas. A survey of some coverage path planning methods is presented in [18]. Robots with other tasks, such as exploration or simply going from one place to another, will need to focus on other aspects.

Path planning algorithms can be split into two categories: on-line and off-line. The former requires no knowledge about the environment and gets all necessary information from its sensors, while the latter needs to be aware of its surroundings before creating a path [7]. On-line methods are suitable when the robot's task involves exploration, as no knowledge of the environment is available prior to execution. When the robot is attempting to map its surroundings while also figuring out its position on said map, it needs to use a version of Simultaneous Localisation And Mapping (SLAM). Many solutions are available and the problem is well tested [14]. The methods differ based on the sensors available. As the field of computer vision evolves, Visual SLAM (vSLAM) methods become more relevant, as these algorithms rely solely on visual information. [22] presents a survey of some vSLAM methods and categorizes them based on the approached used.

It is often beneficial to go a step further and use a version of active SLAM. Active SLAM differs from ordinary SLAM in that the entire process is autonomous, and the robot is able to find its exploration path on its own [30]. A survey of some active SLAM methods is found in [30]. The methods presented uses a variety of sensors (sonar, LIDAR, RGBD Cameras etc.), and have different world representations and exploration techniques.

A popular world representation is the grid-based world representation, where the environment is separated into smaller sections. Each grid is either an obstacle or free space [30]. One can then navigate the free space by a depth-first search on the grid map. [25] uses one stereo visual camera to create a grid map of its surrounding as a part of the active SLAM process. After mapping the immediate area, they use a cognitive-based adaptive optimization algorithm to find the next motion of the robot.

Another popular method is to use a sampling-based algorithm, for example the Rapidly-exploring Random Tree (RRT). The RRT is an iterative algorithm which samples randomly in the free space. The new sample is then connected to the closest node already in the tree [8]. Once a sample is done within the goal area, the tree will provide a feasible path from start to goal. The RRT is particularly good in systems where differential constraints are present and is guaranteed to find a feasible solution [16]. The paths created avoid all obstacles, as the samples are only done in the free space. [19] uses an algorithm similar to the RRT to guarantee optimal coverage of a 3D space. [27] uses a graph-based method to create an exploration path in an unknown subterranean environment.

## 3.2 Obstacle avoidance

Obstacle avoidance is a necessary part of any realistic path-planning algorithm. The methods presented in Section 3.1 all have integrated their obstacle avoidance into the path planning, as occupied spaces are not considered a viable path. One can also look at obstacle avoidance as a separate task. In these scenarios, the obstacle avoidance algorithm will adapt the path of the robot in real-time and thus avoid the obstacles.

The complexity involved in obstacle avoidance varies greatly. The simplest version consists of simply detecting an obstacle and promptly stopping, while more advanced versions involve determining the size of the obstacle and navigating around it [4]. The amount of processing power available and which sensors are used limit the amount of suitable algorithms.

One way to incorporate obstacle avoidance is to use the wall-following approach. A robot equipped with a distance sensor can be told to follow a wall in order to navigate a room [9]. The distance sensors can ensure that the robot remains close to the wall. If any obstacle is encountered along the path, the robot will simply treat this as another wall, and consequently move around it.

Another popular method is to use the available sensors to find the vertical edges of the obstacle, and then have the robot navigate around them [5]. Visual sensors are well equipped for edge detection, and a large variety of techniques exist [1]. [3] presents a method using ultrasonic range sensors. As the robot moves, an obstacle is detected if the distance measured is less than a predetermined threshold, and the robot is moving closer to the obstacle. When an obstacle is detected, the robot will attempt to determine the edges of the obstacle by rotating the sensor to get a wider view. If there is a large difference in two subsequent measurements as the sensor rotates, an edge is detected. When both the left and right edge is found, the obstacle is added to the map and a new path can be created. The drawbacks of this method are that the ultrasonic sensors are prone to inaccuracies and that the robot will need to stop while the edge detection measurements take place.

Khatib [2] introduced obstacle avoidance using an artificial potential field. The robot's target is assigned an attractive force, while the obstacles are given repulsive forces. For each robot position, the sum of these forces is calculated and used to determine the next motion [2]. The planner will favour paths that are far from the obstacles, and they are thus avoided.

Virtual Force Field (VFF) algorithms combine obstacle representation by certainty grids and navigation by potential fields [4]. The certainty grid reduces the effects of inaccurate sensor measurements, as the grid's corresponding value will increase each time a measurement detects an obstacle. This way rouge detections have a smaller impact. The cells close to the robot will be assigned a repulsive force. The force is bigger the higher the certainty value is, and also higher the closer the robot is to the cell. The robot's target has a fixed attractive force. Combing this attractive force with the sum of all the repulsive forces from each grid, a new vector is created, as illustrated in Figure 3.1. This new vector determines the direction the robot is heading and thus dissuades the robot from getting too close to obstacles. The method is fast but struggles when the obstacles are close to each other. As all repulsive forces are summed into one repulsive force, information regarding the obstacle distribution is no longer available [5].

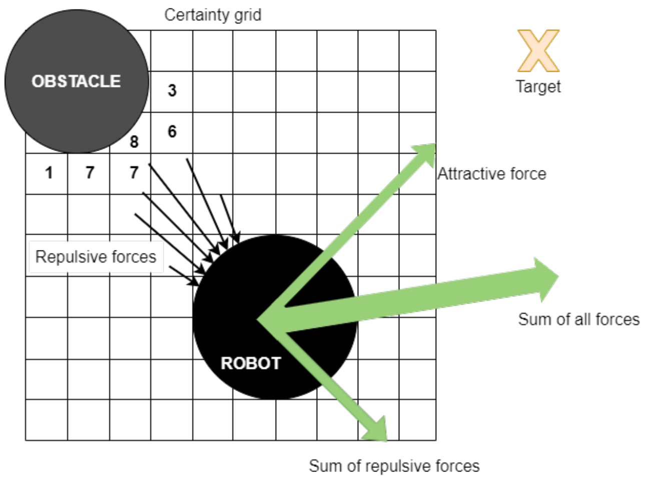


**Figure 3.1:** Illustration of the Virtual Force Field. Combining the attractive and repulsive forces results in the movement direction.

To get past this limitation in the VFF, the Vector Field Histogram (VFH) method was developed [5]. In addition to the two data levels found in the VFF, each grid's value and the combined repulsive force, the VFH also includes a third level. This level consists of a polar histogram created around the robot's position, split into sections of equal width. The grid values are then mapped into their corresponding section, creating a new obstacle density value for each section. This process is visualized in Figure 3.2. This additional layer makes the VFH especially suited for sensor fusion and inaccurate sensor data.

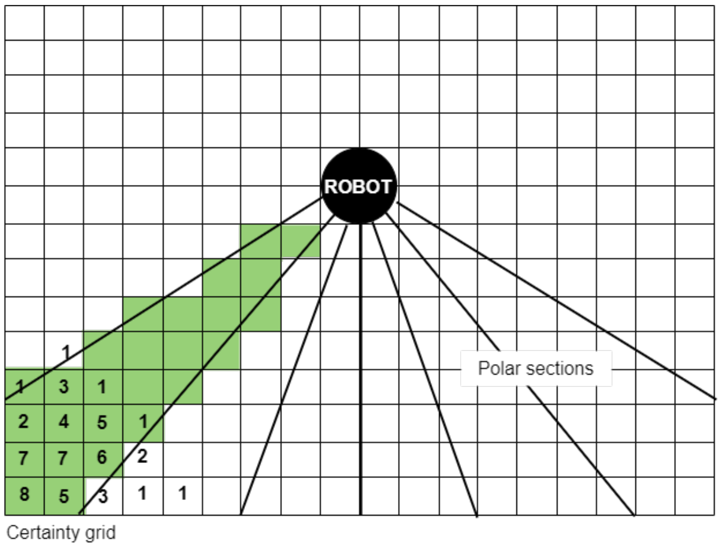
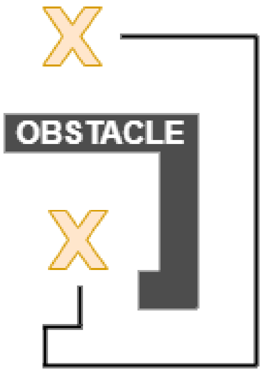
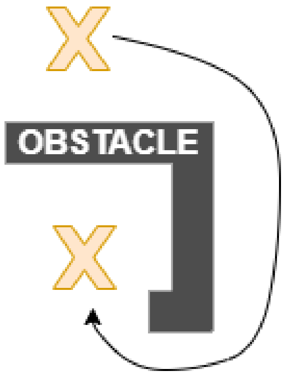


**Figure 3.2:** Vector Field Histogram: Data reduction into a polar histogram.

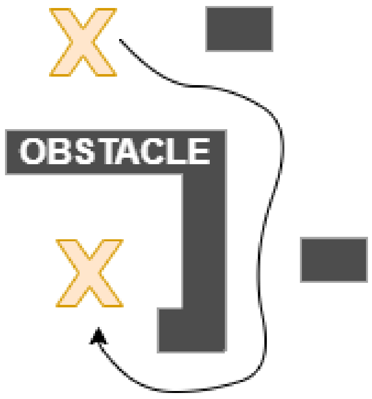
Another method for obstacle avoidance is presented in [6]. While using a global path planner to get the robot to its destination, a control system is designed to deform the path from the planner to avoid any obstacles that may appear. The concept of real-time adaptions to account for unexpected changes in the environment is referred to as *elastic bands*. Additionally, as many path planners only care about feasibility, the elastic band procedure also assists in creating a smooth path that is easy for the robot to navigate, as seen in Figure 3.3. When a path is received from the planner, a contraction force will simulate that of an elastic band and smooth the path. In order to avoid the obstacles, the path is also susceptible to the repulsive forces of the obstacles.



**(a)** A path created by a path planner



**(b)** Elastic band: contraction and repulsion forces added



**(c)** New obstacles deforms the path

**Figure 3.3:** The elastic band method avoids obstacles and smooths the path by adding contraction and repulsive forces.

Chapter 4

# Implementation

The main task of this thesis was to look into how a robot can return home without any input from the server. This behaviour, referred to as the *homing procedure*, is intended to be used in special cases where the server cannot be relied on. Examples could be a critically low battery where the robot will need to override the instructions from the server, or the server disconnecting. The overall goal was split into smaller tasks, such as getting the robot to give itself a position, being able to return home, and avoiding obstacles. As the work evolved, some of these tasks were altered, discarded or created to accommodate the experiences gained while working. Improving the robot's position estimate was not a part of the scope.

The project started by getting the setup to work, and the robot to run with the code that has handed out a the beginning of the semester. A lot of time was spent trying to get a good understanding of how the code worked and how the different tasks communicated. Most effort was put into understanding how the robot moved, the motor speed controller, the communication with the server, and the existing obstacle avoidance features, as these parts seemed most relevant.

The robot moves by calculating the difference between its current estimated position and the target position provided by the server. The resulting error is a straight line, and the robot will first rotate to align itself with said line, and then move straight ahead as long as needed. As it is moving, it is able to correct the heading to a certain degree. This is only for small adjustments and will not enable full turning. The estimation of the pose is done in the pose estimate task, where data is collected from the encoders and the IMU. Next, an Extended Kalman Filter (EKF) is applied, and the resulting estimate for the x position, y position, and heading is saved. The robot will stop if it encounters an obstacle, is within a certain distance of the target, or if it has driven further than it was supposed to.

When using manual inputs, the server publishes the target position several times. This causes the robot to recalculate its path every time a new target position arrives from the server, even though it is the same as the previous target. Consequently, it also enables the turning, giving the robot a chance to face the target again if the robot has drifted off course, before resuming to drive.

## 4.1 Return to previous targets

The first issue to be solved is how the robot can give a target to itself, instead of getting it from the server. As there are several possible use cases for a scenario where a homing procedure is useful, which can include the server being both on and off, the robot is simply told to ignore any potential inputs from the server whilst returning.

One solution to getting back home involves simply backtracking its path. This is achieved by the robot locally storing the new targets it receives from the server. When the homing procedure is initialized, the robot will give itself the new targets, where the first one is the last target received from the server. Once the robot reached that point, it will move on to the next, and in that manner iterate backwards through the targets. The initial point of the robot is always (0,0), and this is where the robot will end.

In order to achieve this behaviour a list was created using an array of a self-defined structure called *position*, intended to store the targets from the server as $x$ and $y$ coordinates. This array is initialized with a fixed number of elements, namely *TARGET_DIM*. The alternative, some form of dynamic memory allocation, was deemed less appropriate for this real-time, embedded system. Embedded systems often have limited memory and processing power, and real-time environments require a higher degree of determinism and predictability. Dynamic memory allocation may cause non-deterministic behaviour, memory leaks, and sub-optimal memory usage and was thus discarded. The dimension of the target array was set to 10, which is sufficient for the amount of testing necessary in this thesis. If the SLAM Robot Project should evolve to allow for more, this can easily be increased.

If the robot has been receiving a lot of targets before the homing procedure starts, it is possible that the server will lead it to a point it has been at before. In this scenario, there exists a shorter route home than backtracking to all previously visited targets. Instead, the robot should find the previous time it was in this spot, and use the path to that point. Several attempts at adding the same point to the list should also be ignored. The resulting algorithm for adding targets to the list is shown in Algorithm 1. If the new point from the server is (0,0) the entire array can be set to a default value of 0, as we approach the origin. Otherwise, the function deals with the logic of manipulating the array by either adding to the end (new separate target), leaving the list as it is (new target is close to the target already at the back of the list), and resetting the path by deleting the entries behind the target being re-visited.

As the array keeping track of the target position is of fixed size, a way of counting the number of targets visited was needed. The default value of the array is to have [x=0, y=0] on each spot. Thus one can simply count how many entries differ from that to figure out how many targets are in the array. The implementation assumes that the robot's starting position is (0,0), as this is the value that the robot will eventually return to. In addition

**Algorithm 1** add_to_target_array(newPoint, targetArray)

```
numTargets ← get_number_of_targets(targetArray)
matchIndex ← -1
if newPoint == (0,0) then
    Set all values to 0
    return
end if
for i=0; i < TARGET_DIM; i++ do
    if newPoint is close enough to targetArray[i] then
        matchIndex ← i
        break
    end if
end for
if matchIndex == -1 then                          ▷ newPoint does not exist in targetArray
    push_to_list(newPoint, targetArray)
else if  matchIndex == numTargets -1 then         ▷ newPoint is already last element
    Do nothing
else                                              ▷ Robot has been at this position before
    for  int i=matchIndex+1 ; i < TARGET_DIM , i++ do
        targetArray[i] = 0           ▷ Everything behind previous match is reset to zero
    end for
end if
```

to counting entries and adding to the list as described above, functionalities for reading from the list and removing elements were created.

A robot may revisit a previous target without receiving the exact same coordinates. Thus it is necessary to add some logic defining an area that counts as the same point. This is defined as a circle around a point, where the radius is set to the same as the target radius. Everything within this radius of a previously registered target will count as that target and treated as such.

The homing procedure was written as a part of the pose controller. The overall logic of the implemented homing procedure is shown in Algorithm 2. See Figure 4.1 for an explanation of some of the variables used. Logic and code regarding other aspects of the pose controller have been simplified or removed for clarity. In its default state, the robot is expecting server input and will move to wherever the server tells it to go. When a new target is received it will call *add_to_target_array* (Algorithm 1).

At some predefined condition, the boolean *goHome* will be set to true, and the robot will start its homing procedure. This condition can be a server disconnect, detection of battery or something else entirely. The testing of this thesis was done using a fixed number of targets in the array as a condition for returning. From this point on, the robot will only accept targets from the targetArray and will ignore any possible input from the server.

The pose controller task includes a while(1)-loop. User input or the server's exploration

**Algorithm 2** Logic regarding homing in the controller task

```
1:  goHome ← FALSE                    ▷ Boolean describing status of the homing procedure
2:  readyForNewReturnTarget ← FALSE
3:  drivenTooFar ← FALSE
4:  doneTurning ← FALSE
5:  targetRadius ← 150mm                                          ▷ Accepted target area
6:  initialize targetArray [TARGET_DIM]
7:  while TRUE do
8:      if Incoming server message AND !goHome then                ▷ New target from server
9:          xTarget, yTarget ← target from server
10:         add_to_target_array(targetArray, xTarget, yTarget)
11:         Calculate distanceToTarget
12:         doneTurning ← FALSE
13:     else if readyForNewReturnTarget AND goHome then ▷ New target from array
14:         xTarget, yTarget ← get_last_value(targetArray)
15:         readyForNewReturnTarget ← FALSE
16:         Calculate distanceToTarget
17:         doneTurning ← FALSE
18:     else if drivenTooFar then                               ▷ Want to recalculate path
19:         drivenTooFar ← FALSE
20:         Calculate distanceToTarget
21:         doneTurning ← FALSE
22:     end if
23:     Calculate distanceDriven and distanceError
24:     if distanceDriven > distanceToTarget then               ▷ Robot has driven too far
25:         Stop driving
26:         drivenTooFar ← TRUE
27:     else if distanceError <= targetRadius then                 ▷ The target is reached
28:         Stop driving
29:         if some condition for init goHome AND !goHome then
30:             goHome ← TRUE                             ▷ Homing procedure is initialized
31:             readyForNewReturnTarget ← TRUE
32:         else if goHome AND target reched then
33:             remove_last_from_target_arr(targetArray)
34:             readyForNewReturnTarget ← TRUE
35:         end if
36:     end if
37:     if distanceError > targetRadius AND not stopped then
38:         if doneTurning then
39:             Drive straight ahead
40:         else
41:             Start turning towards target
42:             doneTurning ← TRUE when robot faces target
43:         end if
44:     end if
45: end while
```

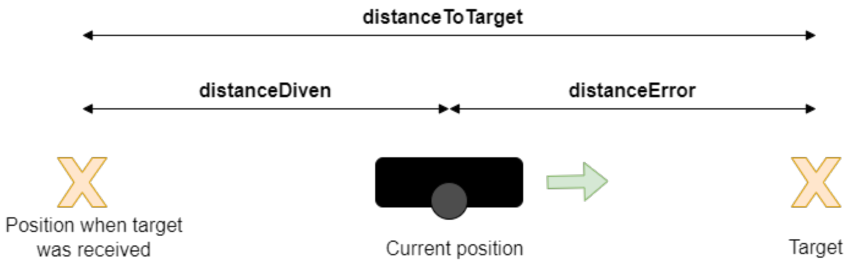


**Figure 4.1:** The different distance errors calculated in the controller task.

mode will both ensure that a new target is not given before the current target is reached. This delay is not present from the robot's side. A conditional boolean is therefore necessary to determine when the robot is ready for new inputs. This value, *readyForNewTarget*, is set to false as soon as the first target is given from the list. It is not set to *true* before the robot has reached this position. A target is said to be reached once the robot is within a circle with the target coordinates as the centre, and *targetRadius* as the radius. Once there, the target will be deleted from the array and *readyForNewTarget* becomes *true*. After a new target is given, *readyForNewTarget* is set to false again as it starts to move towards the new target. This process repeats itself until the robot reaches its starting position.

As a part of the pose controller loop, the robot performs a check to see if it has driven too far, that is $distanceDriven \geq distanceToTarget$. The original behaviour was for the robot to stop, and then await a new target from the server. During the homing procedure, however, this logic will not suffice as a new target will not be given until the current one is reached. Instead, the robot will need to restart its driving process by turning to face the target, recalculating *distanceToTarget*, and then start driving again.

## 4.2 Moving straighter

When testing the homing procedure it became apparent that the robot was not able to drive straight ahead. The drift was so dominant that it became difficult to understand if the logic regarding the homing procedure actually worked. To determine how the robot moved, some driving tests were performed. The first was a square test, where the robot was told to move to targets corresponding to the corners of a square. This was performed without a server connection and is thus just a test of the robot's ability to drive straight and determine when it has reached its target. The new logic regarding whether the robot has driven too far was also present. The square used was intended to have sides of 1 meter, but the robot's ability to drive straight was so poor that the sides had to be reduced to 40cm in order to get the robot to complete the route.

The logic implemented in Section 4.1 appeared to work as intended on small distances when the robot would hit within the target circle even if it drifted. On longer distances, however, without the frequent updates from the server causing a recalculation of the path, the drift caused severe problems. When a robot is given a target, it will first turn towards the target, and then go straight ahead for the calculated distance between the start position and target, *distanceToTarget*. Only when a target is reached, or the driven distance is longer than the error, will it stop to recalculate its path. The variable *distanceDriven* is calculated simply based on the difference between the current position estimate and the position of the robot when the target was received, and not on the distance driven by the wheels. If the robot drifts while attempting to drive straight ahead, it will eventually start to move in circles. If the diameter of this circle is smaller than *distanceToTarget*, the test for checking whether the robot has driven too far will never be true, as *distanceDrive* never will be larger than *distanceToTarget*. This causes the path never to be recalculated, and the robot is stuck in this circular motion, never reaching its goal. Examples of this are shown in Section 5.2. It became clear that getting the robot to move better was necessary.

### 4.2.1 Removing drift

The system is written such that the reference input to the motors is calculated with a PID regulator, using feedback and the distance to the target to determine the inputs necessary to get the robot to move forwards as desired. This regulator returns the same input to both motors, which is then adjusted with a value created to correct the drift, *thetaCorrection*. This correction value is calculated by a separate PID regulator, this time with the heading error as input. In addition to these two regulators used for correcting the position of the robot, the calculated motor input is used as a reference in a new PID regulator controlling the speed of the motor on each wheel. All three PID regulators have different parameters. The latter also has a separate set of values for each wheel, and these are robot specific. All the PID parameters are found in the configuration file.

The first test was to check how the robot behaved when only the motor speed controller was active. The robot was given the same reference input on both wheels. The result was a heavy drift to the right, where the robot was practically spinning clockwise on the spot. When including the two pose controller regulators, the robot still moved clockwise in circles, albeit with a larger radius than before.

Suspecting hardware error, a second robot, the NRF6, was acquired to compare its behaviour to the one of the NRF2. The two robots have very different PID parameters for the motor speed regulator, and some tests on the NRF6 showed that it behaved better. When starting to tune the motor speed regulator for the NRF2, the parameters used for the NRF6 was a starting point. Results are shown in Section 5.2.1. As it is the drift that is the main issue, the integral gain was prioritized. As there is such a tendency to drive to the right, tests were also done using different values on the two wheels. The right wheel, originally underperforming, got higher gains than the left.

The controller used to calculate *thetaCorrection* was also tuned. When tuning this regulator, the original values were used for the other controllers and vice versa. *thetaCorrection* is used to offset the value returned from the distance controller, *u_distance*. The left wheel is then given the value *u_distance -thetaCorrection*, and the right wheel *u_distance + thetaCorrection*. The resulting value is then put within [-100,100] to ensure it is physically possible.

*thetaCorrection* was originally set to have an upper limit of 15, giving the wheels a maximum offset of 30. Tests were conducted to see how increasing this limit affected the robot's behaviour. As there seemed to be a consistent drift to the right in all scenarios, a minimum value for *thetaCorrection* was also added and tested.

### 4.2.2 Increasing the frequency of path calculation

As the code is intended to work on several robots, it is important that the solution will work on all of them, not just the NRF2. While reducing the drift is important, it is also essential to create a solution that does not rely on a well-tuned regulator to function.

The robots are able to find their targets when they receive regular updates from the server, as the heading is recalculated and adjusted every time a new update arrives. When returning, however, it will only receive the target, and thus calculate the heading, once. This latter approach is very prone to drift. In order to fix this, a timed recalculation was implemented, mimicking the functionality found when the inputs are received from the server. This way the path will be calculated more often, and the drift will be of less importance.

## 4.3 Obstacle avoidance

While backtracking to previous targets in order to get home always will work given no dynamic obstacles, it is often not the most efficient method. It might be necessary for a labyrinthine environment, but often a much simpler approach is sufficient. Without any obstacles present, the easiest way to get home is to simply give the robot its starting point as a target and have it go straight there. If obstacles are present, however, the robot will need to have a way of dealing with them.

The robot was equipped with a way of detecting collision at the start of the project. When to robot is moving forwards, the IR sensor facing ahead will take measurements. If the sensor detects anything within a certain threshold, an obstacle has been detected. As the turning happens more or less on the spot, the collision check is only performed when the robot is moving forward. This check is performed by the robot itself and does not require input from the server.

Once a collision is detected, the robot will stop. The existing logic regarding this at the beginning of this thesis was to wait for a new target from the server, and then start heading for that new target. Any form of obstacle avoidance implemented in the homing procedure would have to be simple enough to run solely on the robot, as the server is not necessarily available. If the server has implemented some version of SLAM, the robot will not be able to use this information.

The obstacle avoidance implemented as a part of this thesis is only intended to work for the homing procedure. When the robot is exploring it can use the already existing method for avoiding obstacles, and when given manual targets it is assumed that the input given will lead to a collision-free route.

One solution is for the robot to attempt to go straight back home once the homing procedure is initialized, also when there are obstacles present in the environment. If an obstacle is detected, the robot could simply start backtracking and thus avoid the obstacle in the same manner it did on the way out. This approach was implemented and tested.

### 4.3.1 Obstacle detection

To get a better understanding of how the IR tower worked and what size of the obstacle is needed to get an accurate detection, tests were done utilizing the mapping functionality of the server. The robot sends its IR readings to the server, which is able to display them. This testing was done in a designated area, a circular platform of a 1m radius with surrounding walls. This area makes it easy to control what obstacles are present. With the robot in this environment, different sized obstacles were introduced and the effects it had on the IR readings were studied.

The configuration file offers a possibility to determine how far away the sensors should detect an obstacle, namely a variable called *COLLISION_THRESHOLD_MM*. Different values were tested to see how the variable affected the behaviour of the robot.

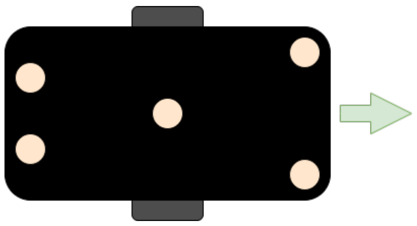
The testing also showed that occasionally an obstacle is detected despite there being no obstacle present. To avoid these false readings initializing the backtracking, a threshold was added. When the number of obstacle detection surpasses the threshold, the backtracking procedure is initialized. The counter is reset every time a new target is reached to avoid a build-up of false detections.

Chapter 5

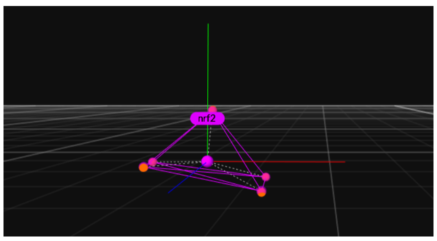
# Results and discussion

The results shown in this chapter are obtained from using the software Motive OptiTrack [40], and its corresponding fixed camera setup. The robot is equipped with five small, silver orbs as described in Figure 5.1a. When these are introduced to the area covered by the cameras, Motive is able to track their motion. In Motive, the five balls are seen together as a rigid body as displayed in Figure 5.1b, and it is the motion of this body that is recorded.

The Motive software is set up with a fixed coordinate frame, and it is this coordinate frame that is used when the motion is tracked. In order to get consistent results, the robot will always start facing the same direction, as seen in Figure 5.2a. Motive is also able to capture height movement, but as the robot will always move along the floor this data is safely ignored. It is important that the robot stays within the area covered by the cameras throughout the entire movement, otherwise the movement will be lost. The initial position of the robot is adapted accordingly.

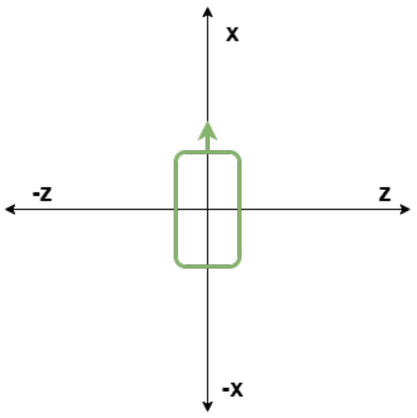


**(a)** Tracker placement on the robot as seen from above.

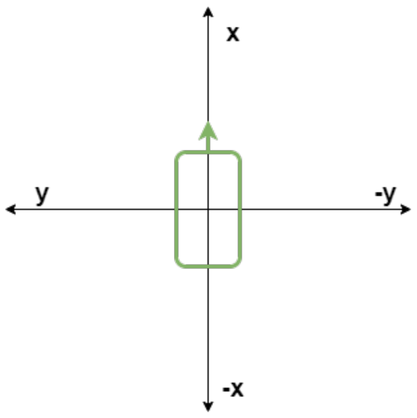


**(b)** How the trackers appear in Motive.

**Figure 5.1:** Motive setup for recording movement data. The orbs on the robot are picked up by the cameras and then seen together as a rigid body.



**(a)** The horizontal coordinate frame used in Motive, seen from above.



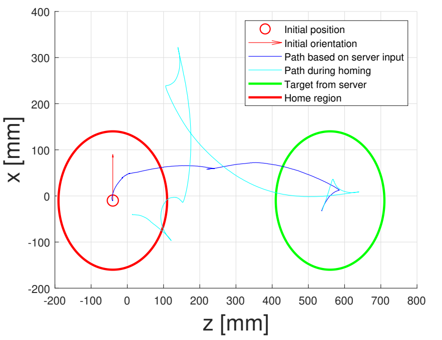
**(b)** The robot's internal coordinate frame, seen from above.

**Figure 5.2:** The different coordinate systems used for plotting and controlling the robot.

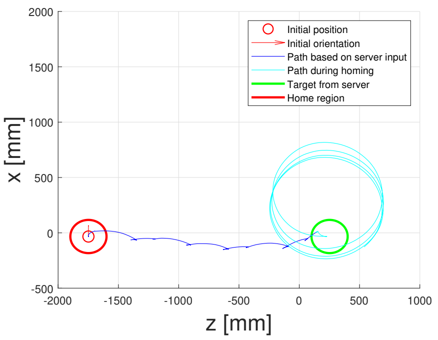
The robot's internal coordinate frame is used for position estimate and receiving targets and is shown in Figure 5.2b. This does not match the coordinate frame used in Motive. As it is Motive that generates the plots, that coordinate frame is used in all the plots in this thesis. The target positions given to the robot are transformed into the Motive coordinate frame before being plotted. In addition, it was discovered early in the testing that the robot's position estimate show half the actual distance. Giving a robot the target *x=300* will thus cause it to move 600mm straight ahead. The robot accepts anything within a radius of 150mm as the target, and the plot is reflected to show this region.

## 5.1 Initial backtracking

The first step attempted as a part of the backtracking was to get the robot to drive home without server input, after reaching its first target. It soon became clear that the robot had a problem with driving straight ahead, a problem that became more apparent at longer distances and made it difficult to test the backtracking logic. Figure 5.3 shows to run where the robot was told to return home after reaching the first target. The only difference between the two procedures is how far away the target position is. In both scenarios, the robot is able to find its target. Once there, it turns and faces home. As it starts to drive home, it is apparent that it drifts considerably to the right. When the distance driven exceeds the distance to the target, the robot realizes it has driven too far and corrects its heading to face home again. On shorter distances, such as the 600mm seen in Figure 5.3a, this logic works. The robot eventually ends up within 150mm of home, where it is satisfied and stops. On larger distances, however, the response to having driven too far will never be triggered, as the distance to the target is larger than the diameter of the circle caused by the drift. As a result, the robot will be stuck in a circular motion. This is clearly seen in Figure 5.3b.



**(a)** The robot is able to return to home with a 600mm distance.



**(b)** The robot is not able to return home with a 2000mm distance.

**Figure 5.3:** Return from one target. The success of the robot is determined by the distance between the initial position and target.

It is clear that while the robot's ability to return home is dependent on the distance, the distance does not affect the success of reaching a target received from the server. The difference between the two is that the path is recalculated regularly when the server is active. This is caused by the server publishing the current input position with even intervals when set to using manual input. A new input causes the robot to allow for the heading to be corrected properly before continuing to drive. This behaviour is clearly seen in Figure 5.3b, where an abrupt, yet small, correction of path happens often. When returning, the robot should not receive targets from the server and this automatic updating will not happen.

Using very small distances to avoid the circular motion seen in Figure 5.3b, an attempt was made to see how the implemented backtracking worked. The result in Figure 5.4 is of little use, as the path is chaotic and the targets are so close together their accepted regions overlap. The robot is able to reach its targets on the way out. As the homing procedure is initialized, the robot starts to make its way back to the first target, but the path quickly gets so intertwined it is difficult to keep track of. However, the fact that the robot does start out in the right direction when the homing starts hints that at least some of the backtracking functionality works as it should. The robot was stopped manually after a while. It is very clear that improving the robot's ability to drive straight is essential, and that it needs to improve before the backtracking functionality can be properly tested.

The robot's response when it realizes it has driven too far was altered to make it recalculate the path instead of simply stopping. The results from these initial tests clearly show that this works as intended. It is able to recalculate the path, rotate to face the target and start to drive in that direction without server input. This functionality is an essential part of getting the robot to its destination. An example of the robot's behaviour, as this path adjustment happens, is shown in Figure 5.5.

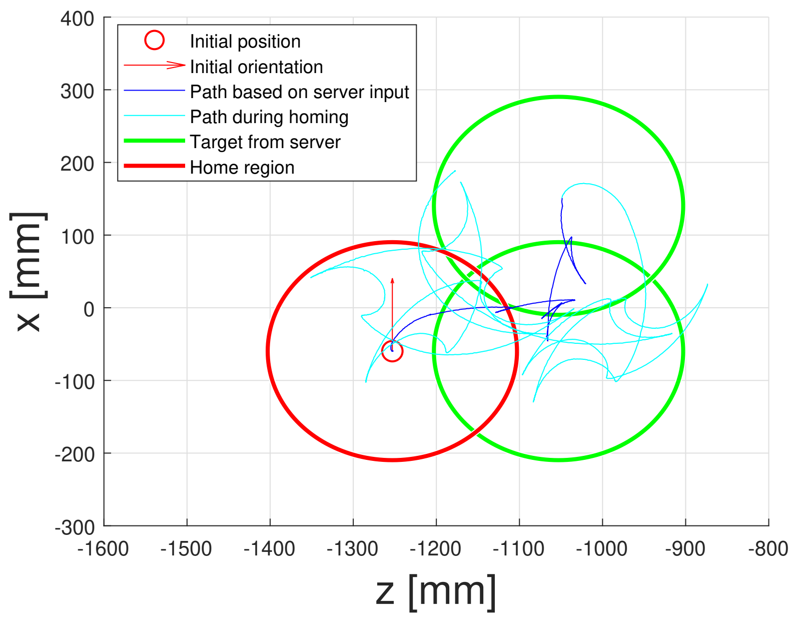


**Figure 5.4:** Backtracking when reaching the second target with the original configuration.

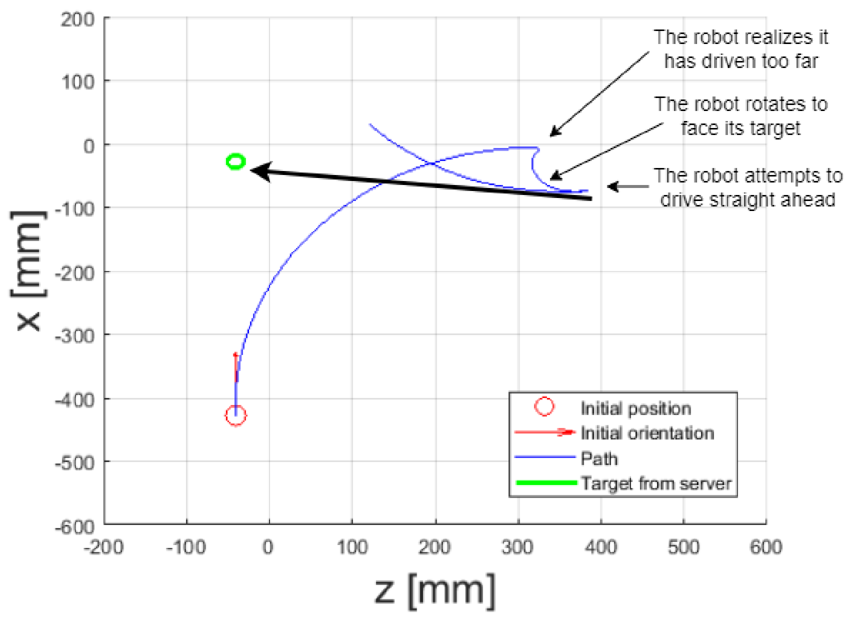


**Figure 5.5:** The different steps involved when the path is adjusted after the robot realizes it has driven too far.

## 5.2 Moving straighter

First, a test to determine how the robot moved was attempted. The robot was given targets that correspond to the corners of a square without any help from the server. The results are seen in Figure 5.6. The only functionality present here is that the robot should turn towards its target and then drive straight to it. If it notices that it has moved too far, it will turn towards the target again and repeat the process. To avoid the phenomena seen in Figure 5.3b, where the robot never realizes that it has driven too far, the size of the square is only 400mm along each side.

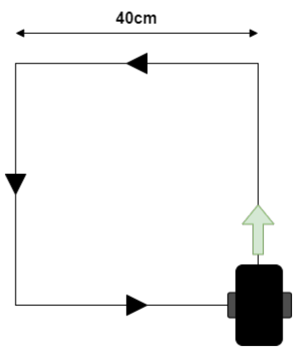
Both for the left and right square, the robot did reach all targets according to its own position estimate, eventually returning to its start position and consequently stopping. The path is however very chaotic and difficult to trace. Additionally, there seems to be a slight error in the position estimate. In Figure 5.6b one can clearly see that the robot stops, and thus thinks that it is home again, outside of the accepted home region. The right square test, Figure 5.6d, completely misses its third target in z,x=(-200,-500). In both scenarios, the robot is able to hit its first target well within the accepted area, while it struggles more with the later targets. The position estimate is based on input from the IMU and the encoders in the wheels, both of which are prone to accumulative error. It is therefore to be expected that the position estimate will worsen slightly as the robot moves.

To get an actual understanding of how the robot moves and turns with as little as possible clutter and overlapping paths, it was told to go straight ahead for 400mm, and then straight to the left and straight to the right for the same distance. These results are in Figure 5.7.

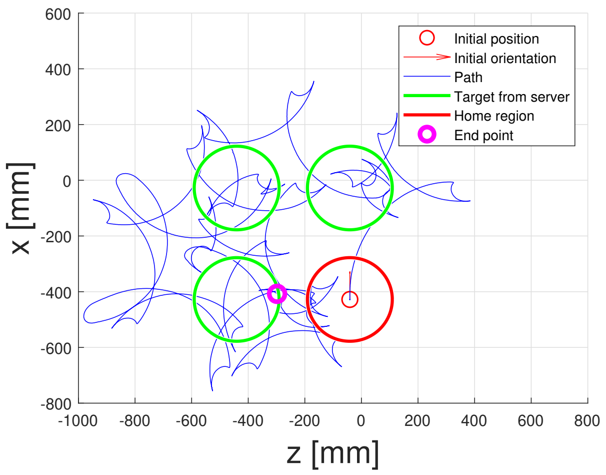
These figures make it easier to understand the movement of the robots at the start of this project. It has a heavy tendency to drift to the right. One can clearly see the points where the robot realizes it has driven too far and adjusts its heading to face the target before starting to drive again. While the robot's attempts at driving straight are not very successful, it seems that the robot's heading adjustment works as it should. One can see that the robot is facing the target after the heading correction before it starts to drift.

Further testing was done to determine where the drifting issue was located. Figure 5.8a shows the result of putting 40% power on both wheels. The pose controller was not involved, and this movement is thus only based on the input reference of 40 and the motor speed controller. The result shows the robot driving heavily to the right, resulting in it circling on the spot. The figure only shows the first couple of turns before the robot was manually stopped.

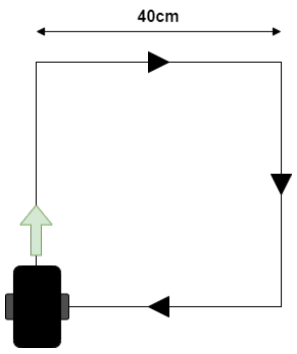
Figure 5.8b shows the behaviour of the robot when given a fixed target straight ahead of its starting position. Here, the pose controllers are also utilized, and the robot attempts to drive straight. The result is considerably better than without the pose regulators, as the radius of the circle has increased significantly. The result is however far from straight, and it is apparent that the pose regulator is not able to cancel out the drift.



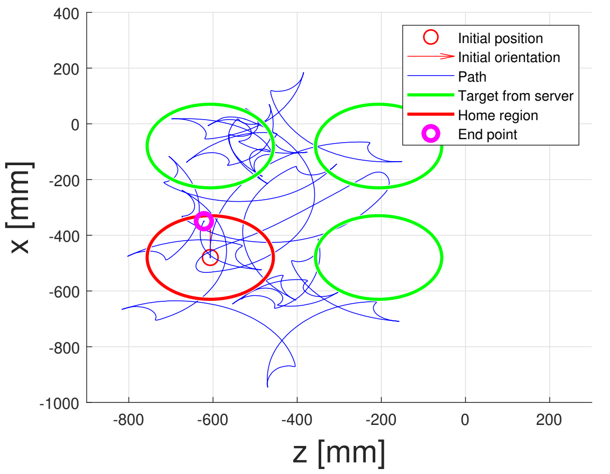
**(a)** Intended behaviour of the left square test.



**(b)** Actual behaviour of the left square test.

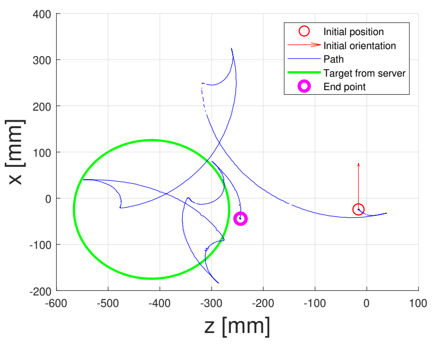


**(c)** Intended behaviour of the right square test

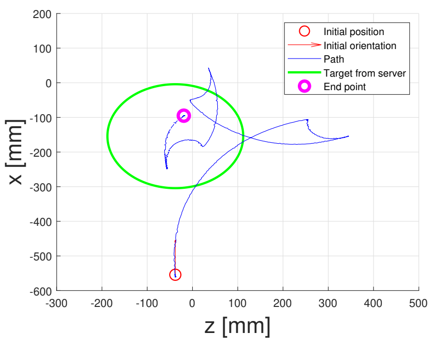


**(d)** Actual behaviour of the right square test

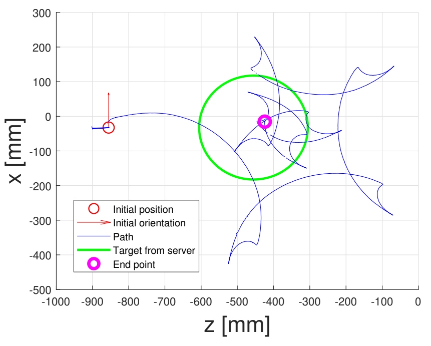
**Figure 5.6:** Square tests to determine robot movement before attempting to improve it. The path is chaotic and difficult to follow.



**(a)** Driving 400mm to the left

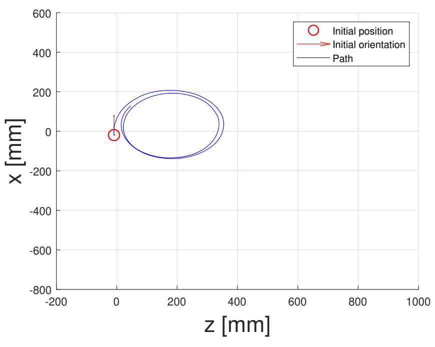


**(b)** Driving 400mm straight ahead

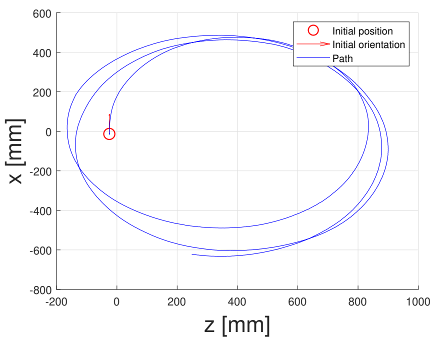


**(c)** Driving 400mm to the right

**Figure 5.7:** Robot behaviour when driving 400 mm in different directions.



**(a)** Both wheels on 40% power. Only motor speed controller active.



**(b)** Attempt at going straight ahead with original where all controllers are active

**Figure 5.8:** The same power input on both motors caused a hard right turn when only the motor speed controller was active. Adding the other controllers improves the path, but the robot is still not able to drive straight.

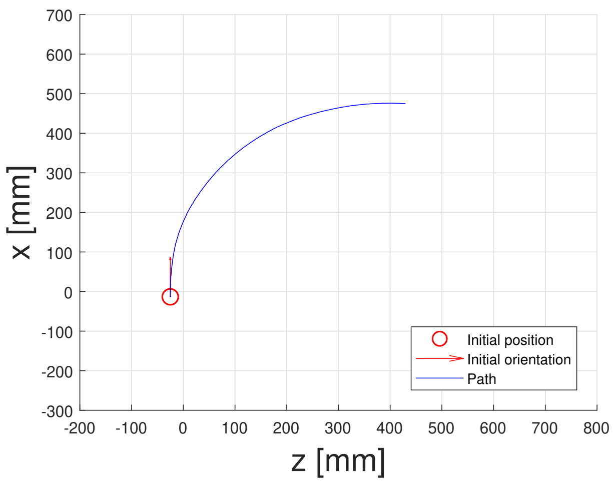


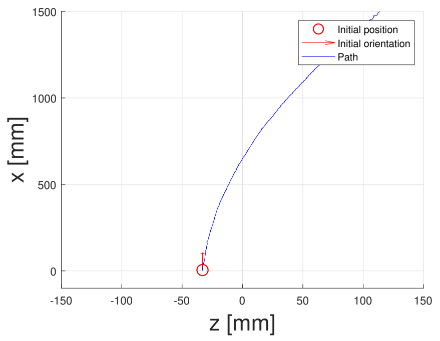
**Figure 5.9:** The original robot behaviour when driving straight. Same data as Figure 5.8b, but adjusted to better visualize the x-difference.

### 5.2.1 Fixing drift

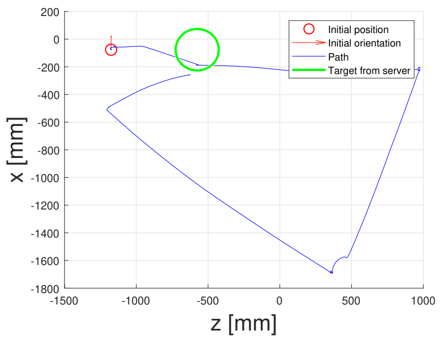
The first attempt at reducing the drift was to tune the motor speed controller. Different configurations were attempted, as shown in Table 5.1. To more easily compare the different configurations, only the path until the robot has reached the maximum x value is shown. The quality of the behaviour is quantified by looking at the interval between the initial x value and the maximum x value, referred to as the *x-difference*. The behaviour of the original values is shown in Figure 5.9 for comparison. Here there is a maximum x-difference of just under 500mm.

The first attempt at removing the drift, using the configuration designed for the NRF6 robot, significantly improved the robot's ability to drive straight, drifting less than 100mm for each 1000mm ahead as seen Figure 5.10a. The NRF6 had gain values that match the ones of configuration *B* in Table 5.1. which is considerably higher than the NRF2's original values and the robot moved a lot faster.

The motor speed controller's parameters are not the only thing that is adapted to a specific robot in the configuration file. Physical constants such as total width and length, and calibration for the IR sensors differ as well. Using the NRF6 configuration on the NRF2 robot caused undesirable behaviour when the robot was given a target, and told to return to home once it was reached as is seen in Figure 5.10b. The robot does not stop at the target, and instead of returning home, it continues to drive for a long time in non-logical directions.



**(a)** Attempting to drive straight ahead.



**(b)** Attempting to drive to one target and back again.

**Figure 5.10:** Robot behaviour using the configurations designed for NRF6.

**Table 5.1:** Different PID configurations used for the motor speed controller.

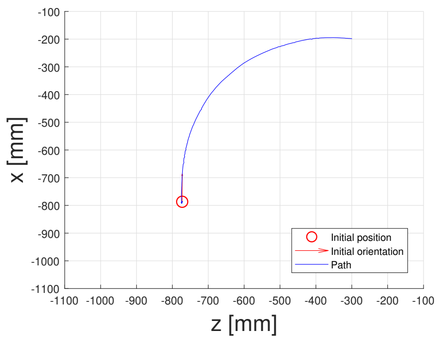
| Name | Left wheel: P, I, D | Right wheel: P, I, D |
|---|---|---|
| Original | 100, 150, 0.0005 | 100, 150, 0.0005 |
| A | 800, 1000, 0.0005 | 800, 1000, 0.0005 |
| B | 400, 1000, 0.0005 | 400, 1000, 0.0005 |
| C | 100, 150, 0.0005 | 400, 1000, 0.0005 |
| D | 100, 150, 0.0005 | 300, 500, 0.0005 |

The rest of the tests were done using the configuration designed for NRF2, but simply changing the PID values. The values are shown in Table 5.1. Both driving straight ahead and driving to a target and back were attempted, and the results are shown in Figure 5.11 and Figure 5.12 respectively. The first configuration tried was to match the values from the NRF6, as it moved relatively straight with these parameters. Using these values, without the rest of the NRF6 configuration, resulted in the robot not being able to drive as straight, as seen in Figure 5.11a. The drift is however less than the original tuning, with a maximum x-difference of 600mm compared to 480mm.

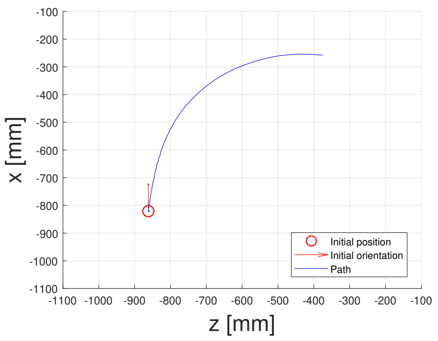
The next values, *B*, have halved the proportional gain while keeping the integral gain from *A*, and the results are found in figures 5.11b. The drift, at a x-difference of 550mm is slightly worse than configuration *A*, but still better than the original.

Figure 5.11c shows the result from driving straight using configuration *C*, which have different values for the regulators on the two wheels. This caused a new effect when driving straight: it started out to the left, but then over-corrected and ended up drifting to the right. It resulted in the biggest x-difference, namely 700mm.

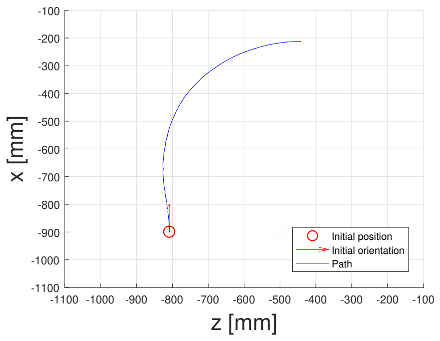
When the PID values were tested on the robot going to a target and back, Figure 5.12a shows the same tendency to drive past its target as was seen when using the configuration for NRF6. The robot does not stop at the target, drove in unpredictable directions, and eventually it had to be manually turned off.



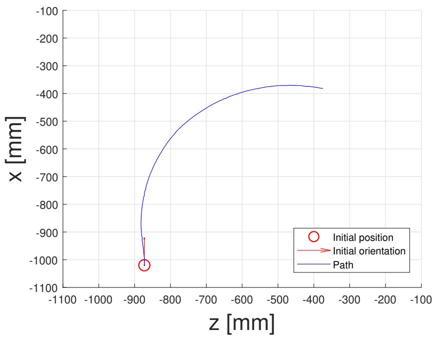
**(a)** PID values $A$: X-difference $\approx 600mm$.



**(b)** PID values $B$: X-difference $\approx 550mm$.

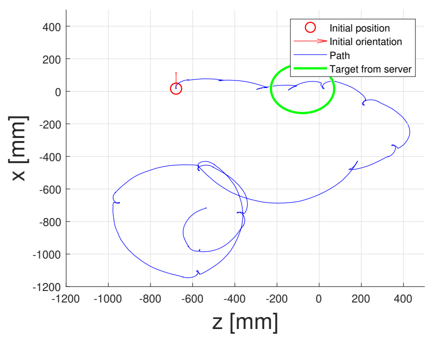


**(c)** PID values $C$: X-difference $\approx 700mm$.

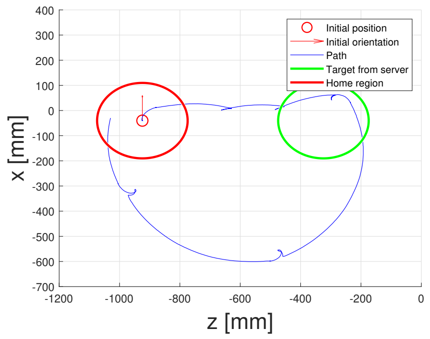


**(d)** PID values $D$: X-difference $\approx 650mm$.

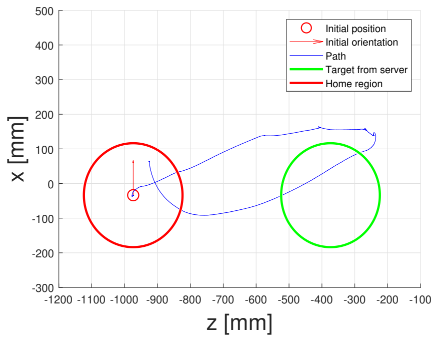
**Figure 5.11:** Driving straight ahead with different different PID values on the motor speed controller.



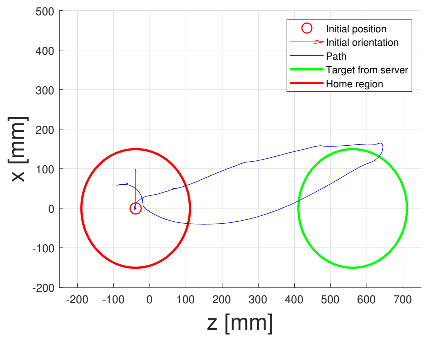
**(a)** To target and back with configuration *A*.



**(b)** To target and back with configuration *B*.



**(c)** To target and back with configuration *C*.



**(d)** To target and back with configuration *D*.

**Figure 5.12:** Driving to a target and back with different different PID values on the motor speed controller.

The *B* values resulted in an important improvement in the robot's behaviour. The robot is now able to stop at its target and turn to drive home. While the path from there is not ideal, the robot is able to get home and does stop within the accepted region.

Using different PID values on the two wheels, as is done with values *C*, caused the relatively smooth path seen in Figure 5.12c. However, the robot's tendency to start out to the left seems to affect the path negatively, and the robot first passes outside of the accepted target region.

In a run like this where the path is relatively smooth, it is difficult to determine where the robot switches from being controlled by the server to giving itself the targets. One possibility is that the robot first passed outside the target, before realizing it had driven too far, turned and drove towards the target and ended up inside the accepted region. Here the homing procedure was initialized, and as the robot would already be facing home, it would require little further heading adjustments to continue. Another option is that the position estimate is not accurate and that the switch happens at the rightmost point of the path. In this case, the robot would miss the target, and it is just a coincidence that the return path takes it through the target. Either way, the robot ends up inside the accepted home region.

To reduce the left turn before drifting right, and hopefully decrease the off-course path when reaching targets, the *D* values were attempted. The two wheels are still given different gains, but the right wheel values were reduced compared to *C*. From Figure 5.11d we see that the x-difference is slightly reduced from *C*, but it is still better than the other ones. Figure 5.12d show how the robot does move by the target on the way out, and does not go as far over it as configuration *C* does. It stops nicely within the homing area. Combining the ability to drive straight and the ability to reach and stop at a target, this configuration is considered the best.

The PID values that affect the regulator for the heading while driving was also experimented with to see how they affect the behaviour. This regulator is one of two present in the pose controller, where the other is used for the distance. As the distance controller returns the same value for both wheels, it does not affect the drift and is thus not tuned. The results of the tuning are shown in Figure 5.13, while different configurations used are shown in Table 5.2. Only different integral gains were tested, as that is the component that ought to remove drift.

**Table 5.2:** Different PID configurations used for controlling the heading while driving.

| Name | Proportional gain | Integral gain | Derivative gain |
|---|---|---|---|
| Original | 200 | 0 | 0 |
| I | 200 | 200 | 0 |
| II | 200 | 500 | 0 |
| III | 200 | 1200 | 0 |

Changing the integral gain had less impact than expected, where configuration *I* and *II* both resulted in only a slightly better x-difference than the original one. Configuration

*IV* drastically increased the integral gain, and it appears to be too high as the system response got worse. The best results are the ones obtained from either configuration *I* or *II*.

This regulator is used to create the *theta_correction* variable, which is then used to differentiate the input for the left and right wheels. In the configuration file, there is a constant limiting the maximum value of this variable. Tuning the regulator is of little help if the output is reduced to stay within a max threshold on the next step. Consequently, tests were done to determine the importance of the threshold value. Both the upper and lower limit was changed, and the results are in Figure 5.14.

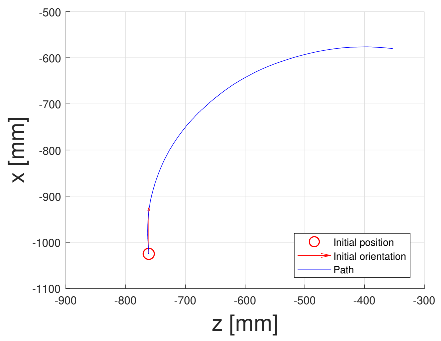
The figures 5.14b and 5.14e show how the behaviour is affected by increasing the maximum value to 30 and 50 respectively. Changing the max value from the original 15 to 30 does increase the x-difference by roughly 150mm to 500mm. Further increasing the max value to 50 has less of an impact on the x-difference, only making the robot go 50mm further. The fact that increasing the maximum *thetaCorrection* does make the robot move straighter aligns with the expected result. It increases the importance of the controller, as its results are not limited as much.

As the drift always seems to be present when driving the robot, a minimum value for the *thetaCorrection* was added and tested. This will create an offset between the wheels that are always present. Keeping the upper limit to 30, figures 5.14c and 5.14d show the result of adding a lower value of 5 and 10. This will set a minimum offset between the wheels to 10 and 20 respectively. While the former increases the x-difference by 50mm compared to having no lower limit, the latter has no apparent effect.

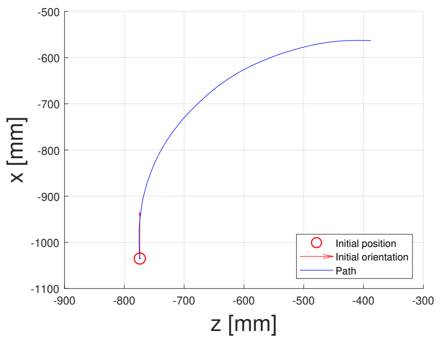
By allowing corrections up to 50 and forcing them to be more than 10, as seen in Figure 5.14f, an x-difference of 600mm was achieved, which was the best result of this tuning. Using no lower limit and max 50, and using lower limit 5 and max 30 follow close behind with 550mm. Having a minimum correction might not be the ideal solution, however, as it creates an offset. If such a thing should be implemented, it would need to be robot specific to avoid affecting the other robots' driving if their wheel alignment is different. As it is possible to achieve a relatively good result without using a lower limit, it could be argued that using no lower limit and max 50 upper limit is the best approach.

It must also be noted that after the distance controller has returned its value, and it has been adjusted by *theatCorrection*, the value still needs to abide by the physical constraints of the robot. It is not able to take any higher input than 100. If the distance controller returns values close to 100, the left wheel will decrease in value as directed by *thetaCorrection*, but the right wheel input would remain almost the same as it is maxed out at 100.

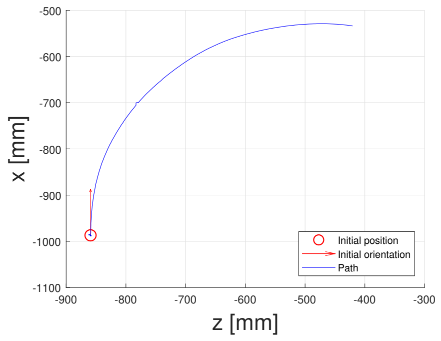
All the testing so far has been on one feature at a time. However, using different combinations could lead to even better behaviour. Combining the best results from the three different variables (configuration *D* in the motor speed controller, *III* in the heading controller, and *thetaCorrection* having upper limit set to 50 with no lower limit) created the result found in Figure 5.15a.



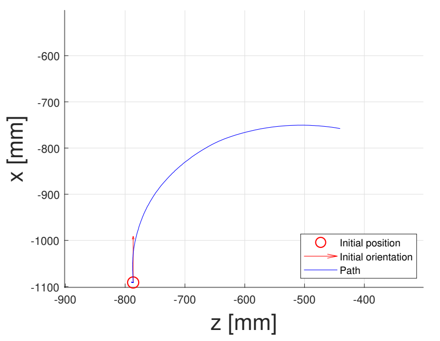
**(a)** Original behaviour. X-difference $\approx 450mm$



**(b)** Behaviour with PID I. X-difference $\approx 480mm$

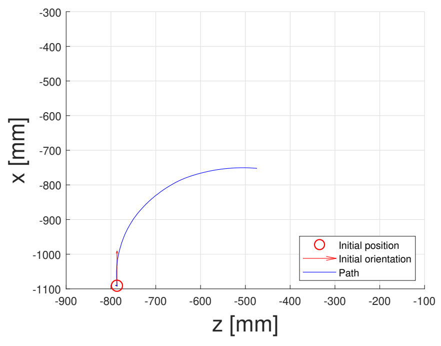


**(c)** Behaviour with PID II. X-difference $\approx 480mm$

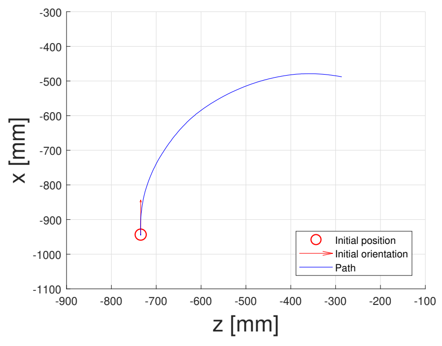


**(d)** Behaviour with PID III. X-difference $\approx 350mm$

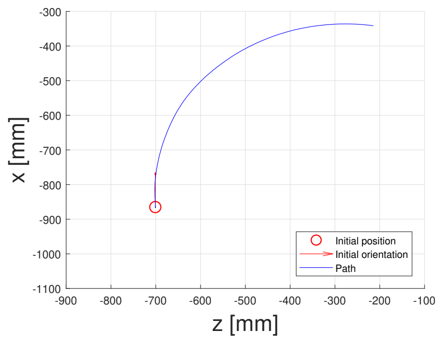
**Figure 5.13:** Robot behaviour with different PID values for controlling the heading while driving. All are with the original PID values D for the motor speed controller.



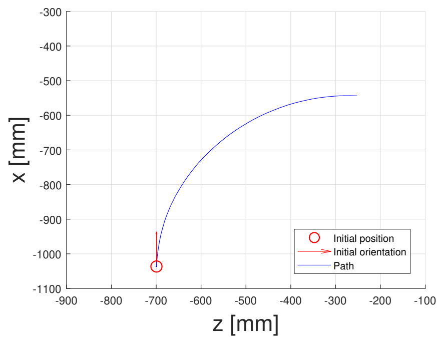
**(a)** Original behaviour: No lower limit, upper limit is 15. Same data as in 5.13a. X-difference ≈ $350mm$



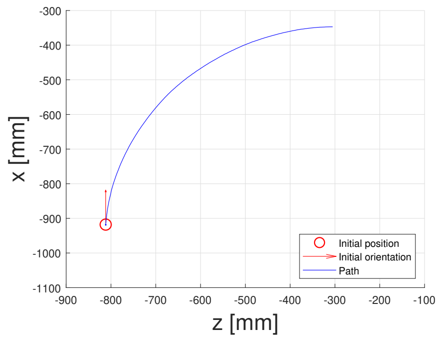
**(b)** No lower limit, upper limit is 30. X-difference ≈ $500mm$



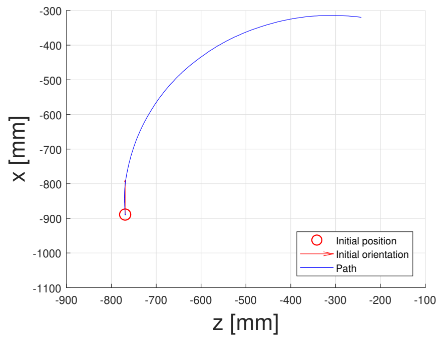
**(c)** Lower limit 5, upper limit 30. X-difference ≈ $550mm$



**(d)** Lower limit 10, upper limit 30. X-difference ≈ $500mm$

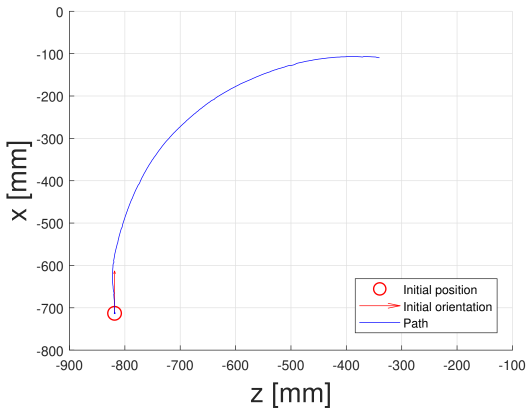


**(e)** No lower limit, upper limit 50. X-difference ≈ $550mm$

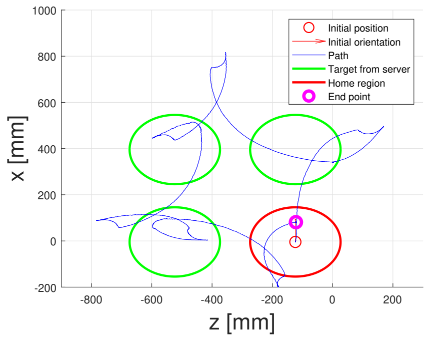


**(f)** Lower limit 10, upper limit 50. X-difference ≈ $600mm$

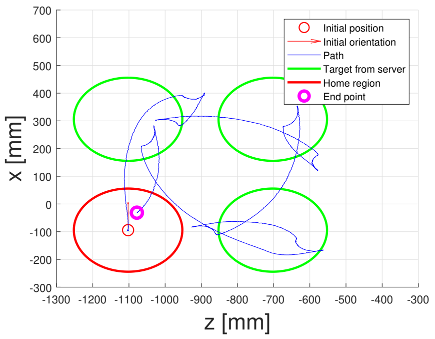
**Figure 5.14:** The effects of altering *thetaCorrection*.



**(a)** Straight ahead. X-difference $\approx 600mm$.



**(b)** Left square test.



**(c)** Right square test.

**Figure 5.15:** Robot behaviour achieved when combining the best behaviours: D configuration on the motor speed controller, III for controlling the heading while driving, and max *thetaCorrection* set to 50.

Combining the different configurations resulted in an x-difference of 600mm. That is better than the result of the configuration chosen by altering only *thetaCorrection* or the heading controller. It is however performing slightly worse in this aspect than using configuration $D$ in the motor speed controller on its own, as that resulted in 650mm (Figure 5.11d). Simply using the $D$ configuration while leaving everything else as it is the setup chosen for the remainder of the thesis unless otherwise specified.

While the robot still drifts noticeably to the right, one can see from the square tests in figures 5.15b and 5.15c that the behaviour is considerably better than the original tuning in Figure 5.6. The robot is now able to reach the targets after only adjusting the once between the targets, and the path is a lot more manageable.

### 5.2.2 Increasing frequency of path calculation

Allowing for adjustments of heading more often results in the robot behaving as seen in Figure 5.16, performed with PID values $C$ from Table 5.1. It leaves home, moves relatively straight to the target, turns around and goes straight back home again. During this entire trip where it has to move 1000mm both ways, it stayed within a band of 200mm in the x direction. It is clear that the updates caused the desired effect, namely that the robot corrects the path by stopping and turning towards the target before continuing to drive. This significantly reduces the effect of the drift, and the robot wanders less of course.

Recalculating the path on the way home has an even bigger impact on bad tunings, as seen in Figure 5.17 which uses the original PID values. The robot drifts heavily to the right, but due to the recalculations, it is able to get home. This behaviour can easily be compared to the one without the updates during the homing procedure. Here, the drift caused a circular motion which lead to the robot never reaching its initial point. Note the change in direction at roughly (-200,0), where the robot seems to realize it has driven too far, turns to face the initial point but then starts turning again.

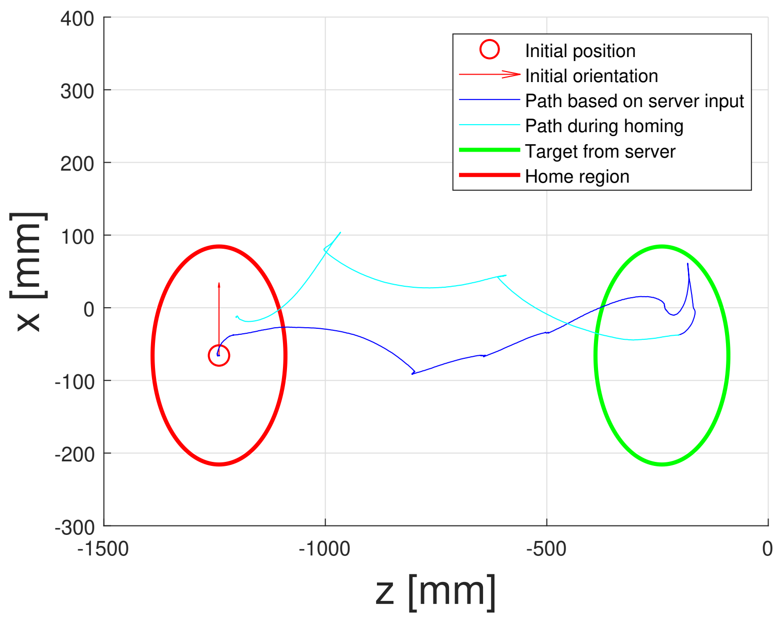
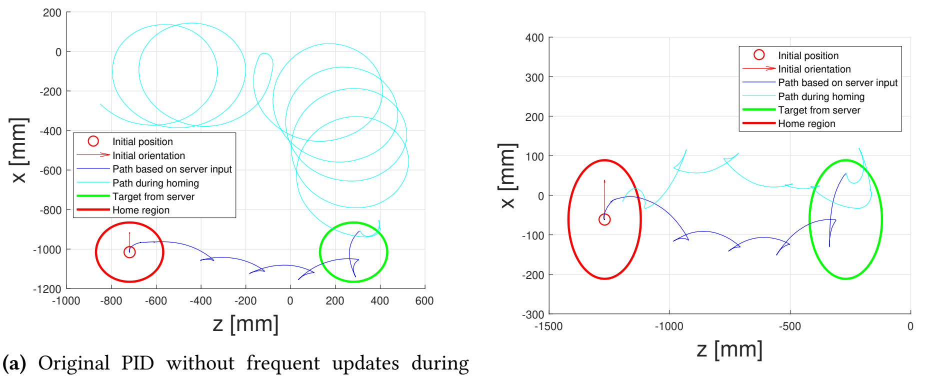


**Figure 5.16:** Recalculating path more often during the homing procedure. This improves the path and the robot is able to return home easily.



**(a)** Original PID without frequent updates during homing with original PID

**(b)** Original PID values with updates

**Figure 5.17:** Recalculating path improves the behaviour greatly when the robot is badly tuned. The frequent recalculation during the homing procedure efficiently reduces the effect of drift.

## 5.3 Improved backtracking

After reducing the drift and updating the path more often, as described in the two previous sections, the robot is now able to reach its targets every time. This makes it possible to test the backtracking algorithm properly.

The first test was to drive to two targets before returning home, as was also attempted in Figure 5.4. The improved result is shown in Figure 5.18. It clearly moves a lot straighter than the previous version, and it is now possible to see that the robot does indeed go where it is supposed to: First going to the two targets as instructed by the server, and then returning home in the same way.

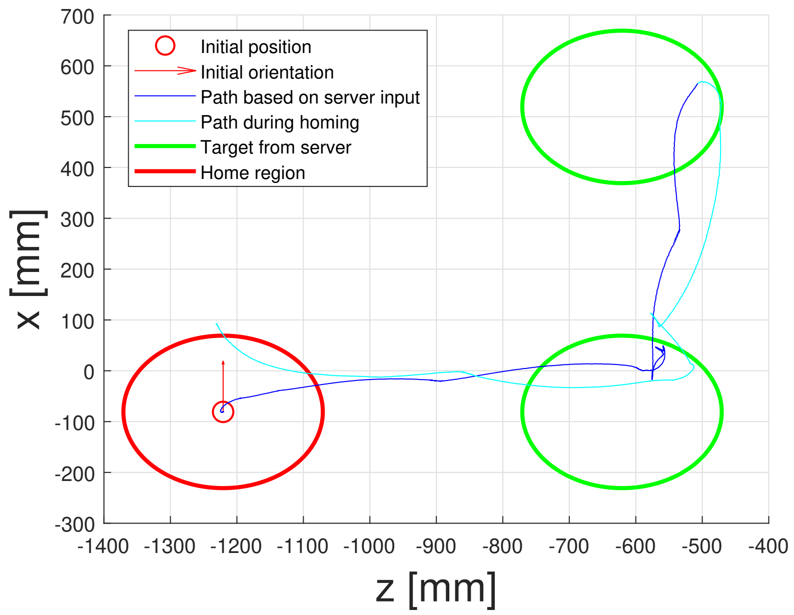


**Figure 5.18:** Start backtracking when reaching the second target. While the position estimate is a bit off, the backtracking logic works as intended.

The backtracking was also tested on three targets. Several tests were done to confirm that it indeed works as intended. The behaviour of the robot is shown in Figure 5.19. The figure displays four runs of the same code. In every run, the robot retraces its previous path by reaching the same targets in the opposite order.

While the backtracking appears to work as it should, it is clear that the position estimate in the robot is not ideal, as has been discussed before. From the paths in Figure 5.19, one can see that the robot turns at different spots every time. While it will usually hit within the accepted region, that is not always the case. In the run display in the upper right corner of Figure 5.19, the robot stopped before reaching the home region.

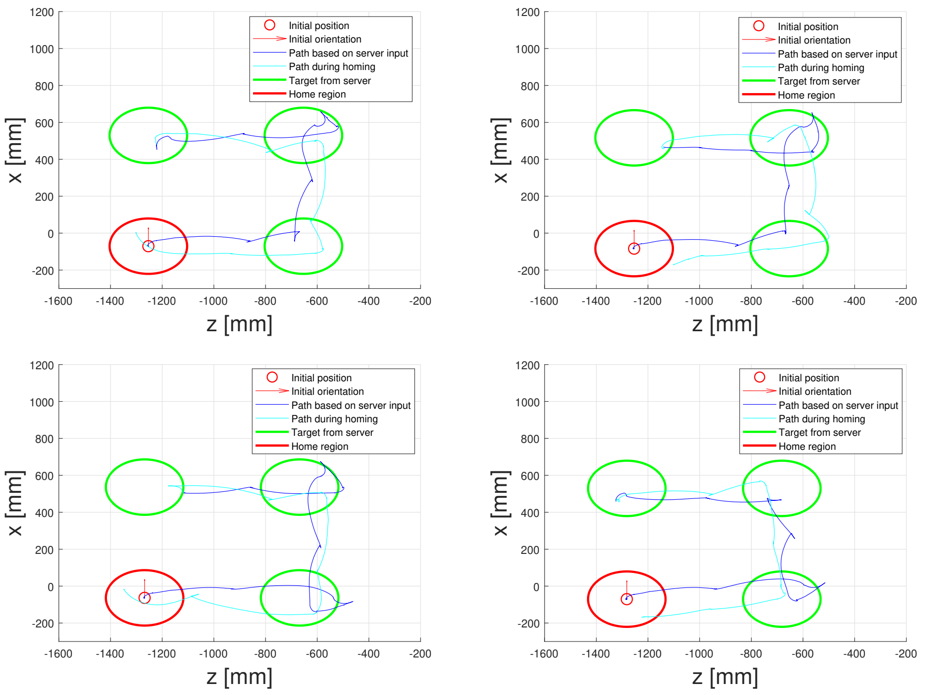
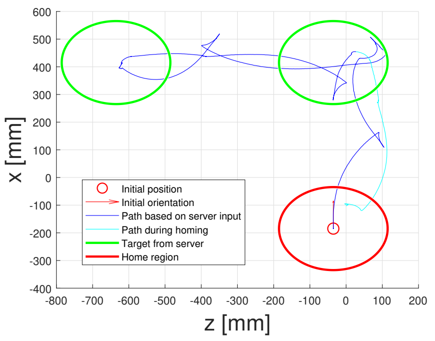
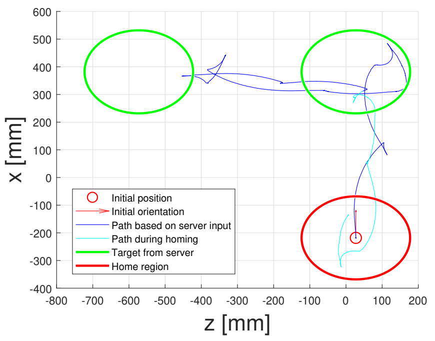


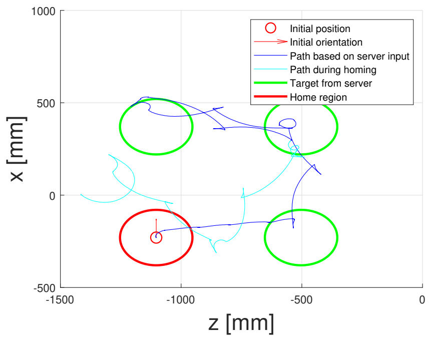
**Figure 5.19:** Start backtracking when reaching the third target. While the position estimate is somewhat inaccurate at times, the backtracking functionality works as intended.



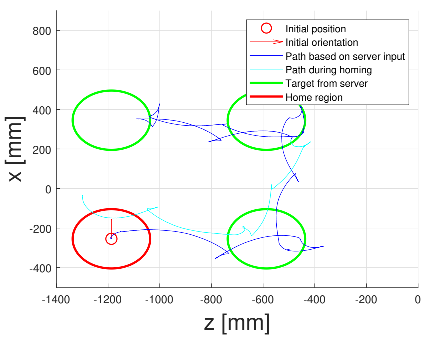
**(a)** Backtrack after 3 targets are visited



**(b)** Backtrack after 3 targets are visited



**(c)** Backtrack after 4 targets are visited



**(d)** Backtrack after 4 targets are visited

**Figure 5.20:** Backtracking behaviour when a previous target is revisited. In stead of revisiting all previous targets, the robot finds the shortest way home.

Tests were also done to check that the logic worked as intended when a target is given that the robot has previously visited. This behaviour is shown in Figure 5.20. Two different scenarios are shown, with two runs of each.

As we can see from the figures, the logic presented in Algorithm 1 is able to reset the path as intended. Instead of the robot backtracking first to the upper left target, it realizes that it has been at this position before, and thus that the shortest way home is the path it took to get there the first time.

Again we see that the position estimate is not accurate. This is particularly visible in Figure 5.20c, where the homing procedure misses both the target at (-500,-200) and home. Figure 5.20d is able to hit the lower right target as it is returning, but it stops outside the home region. The same behaviour is not as present in the runs with only two targets. Figures 5.20a and 5.20b both show a homing procedure that ends well within the accepted home region. This corresponds well with the idea of the position estimate being prone to accumulative drift.

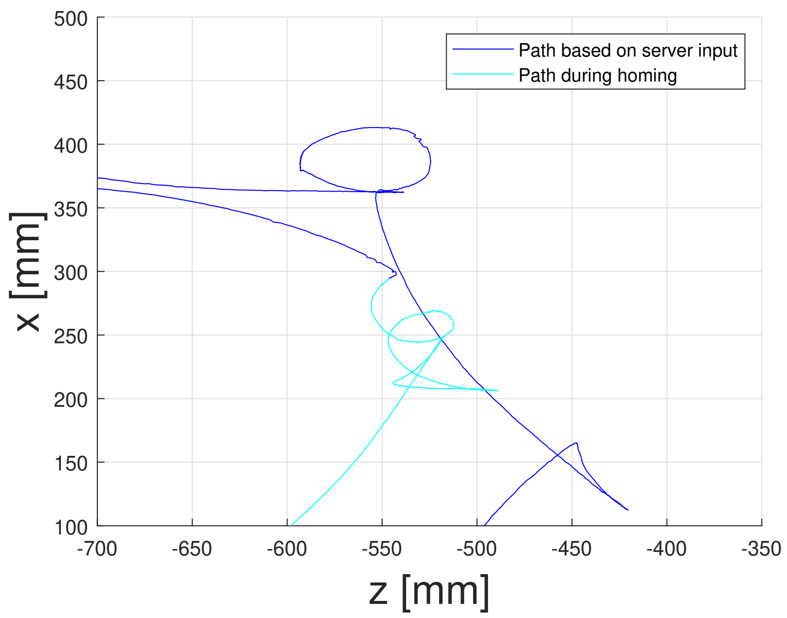


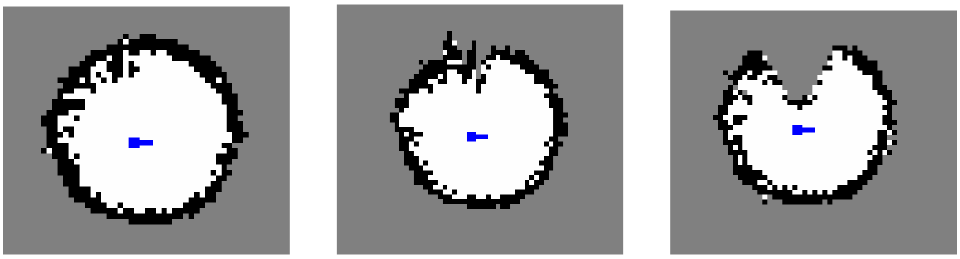
**Figure 5.21:** Unexpected behaviour when switching to the homing procedure. The robot moves a lot on the spot and it affects the position estimate.

In addition to the position estimate being off, Figure 5.20c also displays an unexpected robot movement just as the homing procedure is initialized. Figure 5.21 shows a zoomed-in version of Figure 5.20c, focusing on the unexpected movement. It is possible that this behaviour is too fast for the position estimate to keep up with, as the position estimate appears to be slightly off for the remainder of the run. When the robot was done turning, and was supposed to drive straight to the target, the heading is not correct. The robot is not aimed at the target, but rather facing 45 degrees clockwise from where it should be. This sudden worsening of the position estimate implies that it is caused by something other than accumulative drift. Additionally, if one were to rotate this part of the graph back to where it was supposed to be headed shortly after starting homing procedure, the robot seems to move more reasonably.

## 5.4 Obstacle avoidance

### 5.4.1 Obstacle detection

To get the best behaviour from the obstacle avoidance procedure, it was necessary to investigate further how the currently implemented method for obstacle detection worked. The server is able to display the IR readings on a screen, and this feature was used to determine the necessary size of an obstacle for it to be registered. The robot was left for a while to allow the sensors to gather information from all angles as the tower rotates.



**(a)** IR readings with no obstacles present

**(b)** IR readings with one obstacle of height 15cm, width 10cm.

**(c)** IR readings with one obstacle of height 20cm, width 15cm. .

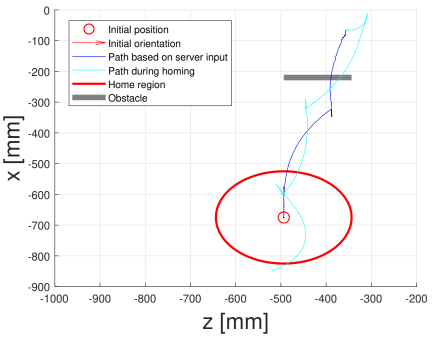
**Figure 5.22:** IR readings with different sized obstacles.

The results are presented in Figure 5.22, where the white area is free space, the black is a detected obstacle, and the grey is an unexplored area. The circular obstacle detection present on all three figures corresponds well to the area the robot was placed in during these measurements. The first reading was done without any obstacles present to get a clear reference. On the second, with an obstacle of height 15 cm, there seems to be some disruption on top of the figure, but not clear enough to be certain. When the height of the obstacle was increased to 20 cm, it became clear that the sensors' view was obstructed, as it left the background unexplored. This is the size of the obstacle that is used in the rest of the testing. It is reasonable that the obstacle has to be quite tall for it to be noticed, as the rotating IR sensors are placed on top of the robot.

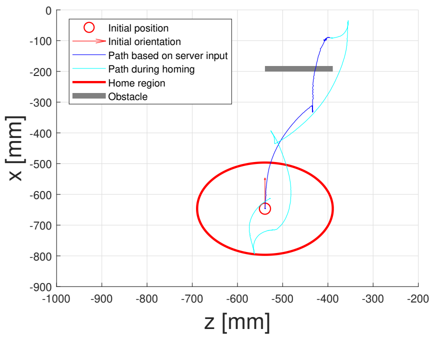
Tests were done to determine the importance of the variable *COLLISION_THRESHOLD_MM*. With only this parameter changing, the robot was told to drive straight ahead to a target far off, and then turn and return to home once an obstacle is detected. The obstacle was light and not attached to the ground, so the robot was able to move the obstacle and drive through it. That also meant that as the robot is returning, the obstacle is no longer in the way. The three variants tested were 100mm, 200mm and 300mm. Any longer values were not tested, as obstacles detected more than 300mm in front of the robot might not be a part of the path.

The first tests were done with the obstacle being placed 450mm ahead of the robot's initial position. The results are shown in Figure 5.23. The three different configurations seem to cause the same behaviour, where the robot drives straight through the obstacle before turning roughly 100mm behind it. It should be noted that the off-course heading causes some issues here. The IR sensor detecting any obstacles is facing straight ahead as the robot drives. When the robot drifts as it does, the sensor might miss the obstacle altogether. Then after a little while, the path is updated and the robot turns to adjust to its target and is suddenly facing the obstacle. Being only 100mm away from the obstacle at this point, an obstacle detection will be triggered in all three scenarios. This could explain why the three runs behave so similar.

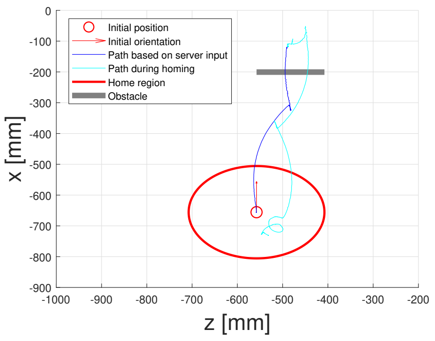
In Figure 5.24, the target was placed further away to avoid this issue. Two runs of each threshold value have been plotted. These results vary more than the ones with a shorter distance, however, they still show roughly the same behaviour. All runs show the robot



**(a)** 100mm threshold

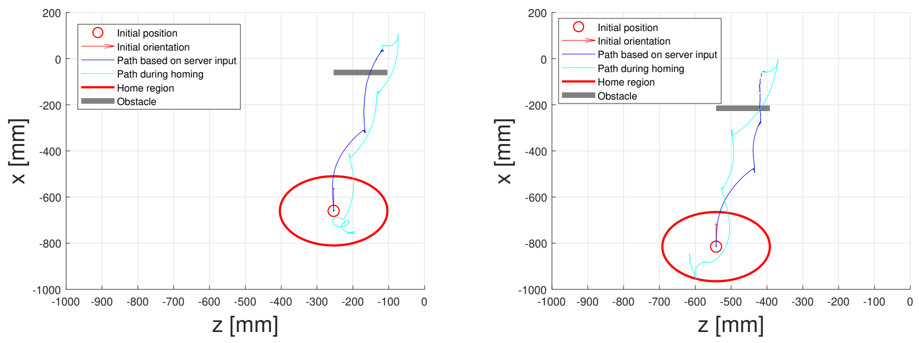


**(b)** 200 mm

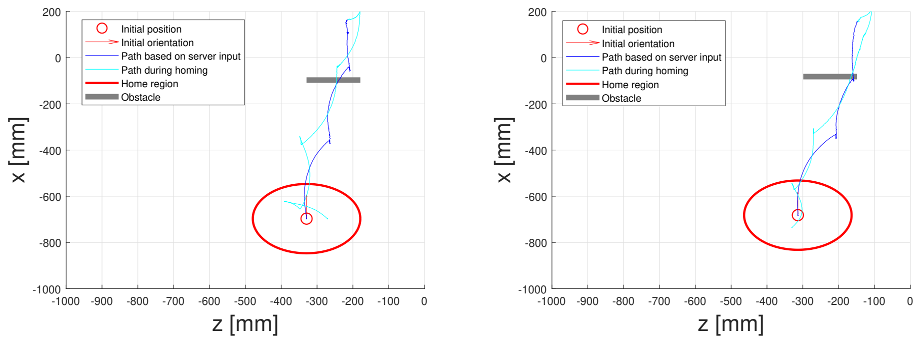


**(c)** 300mm

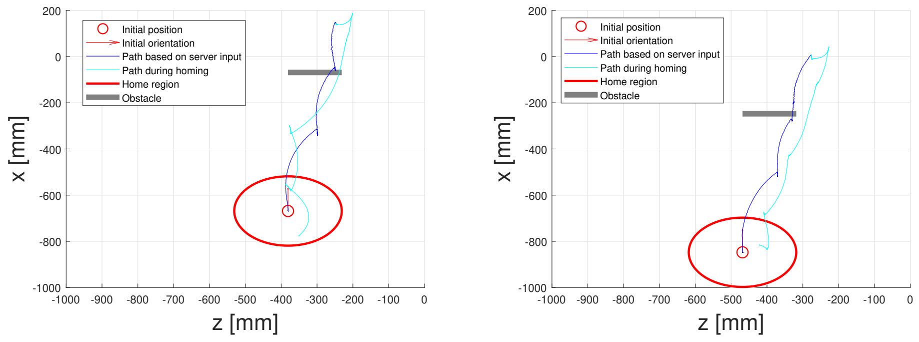
**Figure 5.23:** Behaviour with different detection distances when the obstacle is 450mm away. The different thresholds result in similar behaviour, and all result in the robot colliding with the obstacle.



**(a)** Threshold 100mm.



**(b)** Threshold 200mm.



**(c)** Threshold 300mm.

**Figure 5.24:** Behaviour with different detection distances when the obstacle is 600mm away.

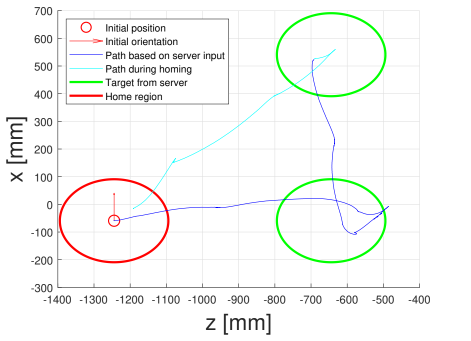
going straight through the obstacle before turning and heading home. In every case, the robot is able to successfully find the way back home and stop within the accepted home region.

These results imply that the value of the variable *COLLISION_THRESHOLD_MM* is of less importance than the robot's ability to react to a detection fast enough. The robot's code consists of several tasks run in parallel. In order for the obstacle avoidance to work properly, the robot needs to be able to prioritize obstacle detections and immediately stop. As it is now, the robot is able to detect an obstacle, but the delayed reaction time makes it collide and drive through the obstacle before turning and going back home.

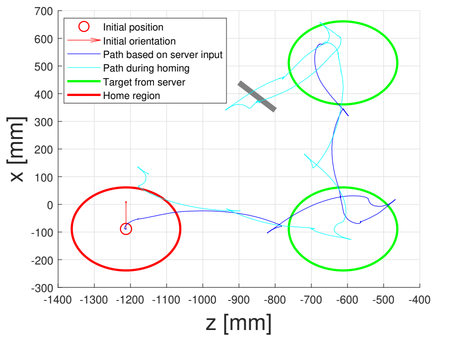
In neither scenario was the robot able to stop in time to avoid the obstacle altogether. Assuming that the obstacle is not a fragile thing, this is not practically speaking very important. If the robot hits a wall, it will simply be stuck there for a little while before realizing it is an obstacle there. The robot is robust enough to handle the impact. A bigger issue is the process of turning to face home, as one can see that this process does in fact lead the robot even further away. If the obstacle is immovable, the robot might not be able to turn around, as it will not have the room to do so. A possible solution could be to have the robot back up a bit when an obstacle is encountered, before turning to drive back home.

## 5.4.2 Backtrack if an obstacle is encountered

The simplest form of obstacle avoidance simply involves attempting to go straight home to (0,0), but reverting to backtracking if an obstacle is encountered. Figure 5.25 shows the robot going to two targets, before attempting to head straight home.



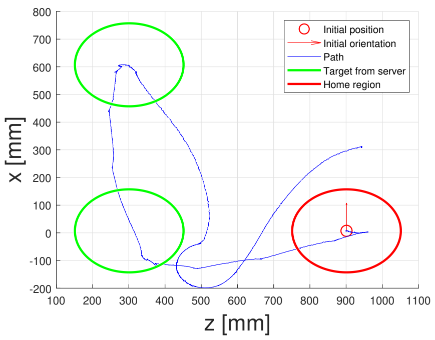
**(a)** No obstacles present, and the robot can return straight to home.



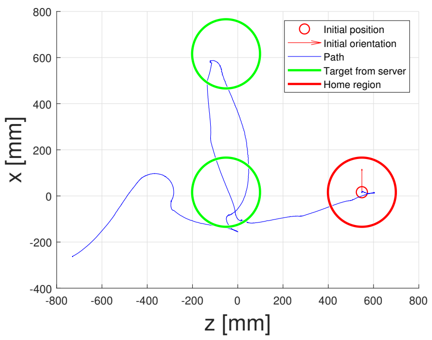
**(b)** Obstacle prevents return to home, and causes backtracking.

**Figure 5.25:** Behaviour when set to go straight home when reaching the second target, and backtrack only if an obstacle is detected.

In the first run, there were no obstacles present in the area, and the robot was able to return straight home. Figure 5.25a confirms this behaviour. Figure 5.25b shows the behaviour of the robot when an obstacle blocks its return path. After the obstacle is de-



**(a)** Expected behaviour during backtracking.



**(b)** Unexpected behaviour.

**Figure 5.26:** Unexpected behaviour during backtracking with configuration *C*. These results were created using the exact same code and targets given from the server.

tected, the robot turns and starts to backtrack its position in the same manner as in previous sections. One can see that the position estimate is not great here either, and the robot ends up missing home. As is the case in Section 5.4.1, the robot is not able to stop in time to avoid colliding with the obstacle.

## 5.5 Unexpected behaviour

In addition to the unexpected behaviour caused by bugs or the code crashing during implementation, the robot occasionally showed irregular behaviour seemingly without any reason. This could happen at any time, both during the homing procedure and when controlled by the server. The robot would in those cases just start driving in the wrong direction, and it is difficult to understand why. Re-running the exact same code usually fixed this issue, and the robot behaved as expected again. This unpredictable behaviour made the system more difficult to debug, as it often was unclear whether the behaviour was caused by an error in the programming or simply the robot behaving unpredictably.

In some cases, the errors appear to be caused by a miscalculation of the position estimate, as seen in Figure 5.26b. Here, the robot reaches both targets when going out, and it is able to return to the first target when the homing procedure is initialized. At this point, however, something weird happens, and the robot seems to lose track of where it is. As it attempts to reach the initial position, it starts to drive in the wrong direction. After a while it stops and appears to be satisfied with its movements. The fact that the robot actually stops, as it only should do after reaching home, is an indicator that the position estimate is wrong and the robot really does believe it has reached its destination. If one focuses on only the of course part of the path in Figure 5.26b, and rotates it 180 degrees, the result is not too far off the expected behaviour in Figure 5.26a. It is thus possible that the error is caused at one specific point, and that the remaining path works as it should.

The unexpected behaviour can happen at any point during execution. Figure 5.27a shows the expected path when the robot was supposed to go the two points received from the server, and then return straight to home. When the same code, with the same server inputs, is run again in Figure 5.27b, it behaves very differently. The robot had successfully reached its first target. When given the next, it starts off in the opposite direction of the target. The robot was supposed to return straight to home after reaching it second target. The abrupt change of direction around (-500, -800) might suggest that the robot thought it had reached the second target and started to drive towards home. Its attempt at reaching home leads it even further away, where it eventually stops and is satisfied. This behaviour might also be caused by an error in position estimate at the first target, though it is unclear what it is that caused this miscalculation.

Other times, there seems to be no logic behind its failures, and the robot simply drives off in a seemingly random direction. Figures 5.27c and 5.27d show two successive runs of the same code. The run is with the original configuration, intended to test the backtracking functionality on short distances (400mm). The graph shown is before the backtracking is initialized, and is just when the robot is intended to go to the targets sent from the server. The second plot shows the unexpected response of the robot after reaching the first target. The robot then seemingly loses track of its whereabouts and the pattern does not have any clear explanation. In this scenario, the robot never stopped and had to be turned off manually.

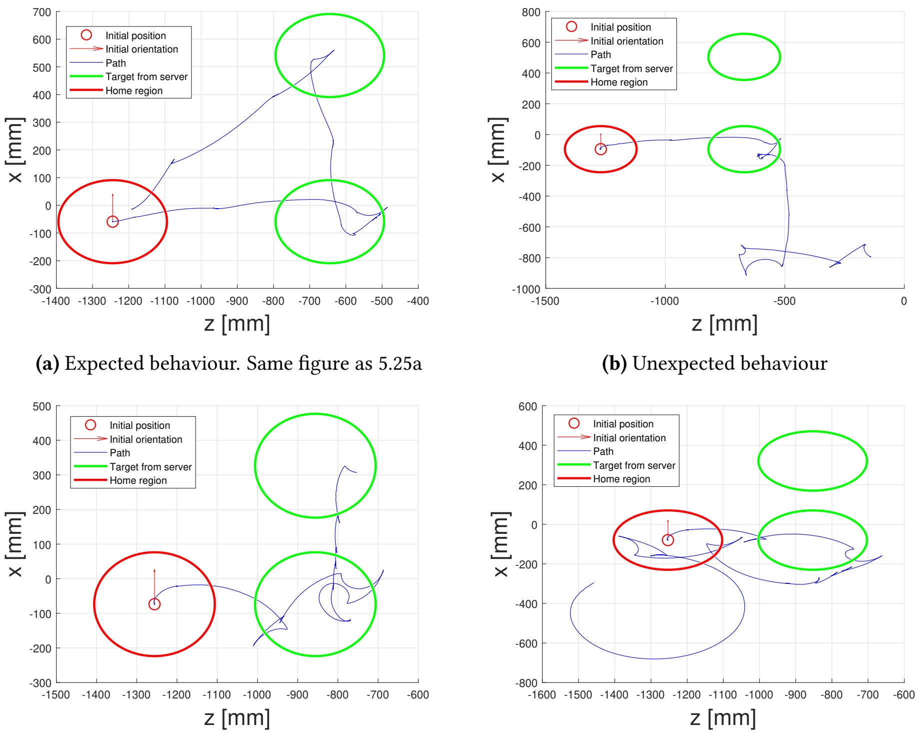
The kind of unpredictable behaviour discussed in this section was a frequent occurrence. Sometimes it was seemingly caused by some miscalculation of the position estimate, other times with no clear explanation. The behaviour made testing the code a lot more time consuming and tiresome.

## 5.6 Overall discussion

The functionality implemented as a part of this thesis is intended to be functional in the special case where the server is not available, or the robot needs to override the instructions from the server to get back home. Any discussion on how the implementations work will need to have this in mind. It is assumed that the server is able to give targets that make the robot avoid any potential obstacle, and the implemented obstacle avoidance is thus only intended to use during the homing procedure.

The implemented functionality regarding making the robot backtrack its previously visited targets in order to get home works as intended. It could either retrace every target to get home or find the shortest way if the same area is visited several times. This method is well suited for environments with many obstacles and walls, such as a labyrinth. In this scenario, the robot will be able to get out the same way it came in and avoid getting lost elsewhere.

The biggest advantage of using backtracking as a way of getting back home is that it will always work, given that there are no dynamic obstacles present. A downside is that its ability to find the shortest way home is limited to the targets given, which are often not ideal. As low battery on the robot could be a potential scenario where the homing procedure could be initialized, finding a short path home ought to be prioritized.



**(a)** Expected behaviour. Same figure as 5.25a

**(b)** Unexpected behaviour

**(c)** Expected behaviour.

**(d)** Unexpected behaviour

**Figure 5.27:** Unexpected behaviour during manual input from server.

Going straight home to the robot's starting position could be the shortest way home, especially if there are few obstacles in the environment. This method does however not guarantee that there are no obstacles in the way, and the robot will need to be able to detect obstacles and respond accordingly.

Obstacle detection is done using the IR sensors. The testing showed that while the robot indeed is able to detect obstacles given they are high enough, the robot is not able to stop in time to avoid collision. It is important that obstacles are detected far enough ahead to enable the robot to stop in time, but not so far that it interferes with ordinary driving through cluttered environments.

The simplest way of responding to an obstacle detection is to use the already implemented backtracking, which guarantees an obstacle-free path. This method might lead the robot out on a long path, much longer than just manoeuvring around the obstacle. This approach is tested on the robot and works well.

If a more advanced obstacle avoidance approach is to be implemented, one would have to consider several parameters to choose the best option. One important thing is the memory required. The robot's memory is limited to that of the nRF52840 card used. As the homing procedure is only intended to use in emergencies, it is not wise for it to take up too much space and potentially limit further functionality that is used more frequently. It becomes a trade-off between the complexity of the method used and the efficiency of the path created.

Of the obstacle avoidance technique presented in Chapter 3, the edge detection method is deemed most appropriate. It is based on rotating rage sensors, which are already present in the robot. The sensors can already be used for detecting how far away an obstacle is. What remains is to use the data from the sensors to determine an edge, use the edges to create the obstacle boundaries, and then create a path that will lead it around the obstacle. A disadvantage of this method is that the robot will have to stand still while taking its edge-detecting measurements. However, spending some time in front of the obstacle to navigate around it could still be more time efficient than having the robot backtrack to its previous position on a potentially quite long path.

Before implementing a more advanced version of obstacle avoidance, certain things need to be fixed. The robot has to be able to stop in time to avoid colliding with the obstacle, as discussed above. The robot's position estimate will need to improve to guarantee that the robot will actually move to the target that will lead it outside the obstacle boundaries. Another issue is the robot's ability to drive straight, as one has to avoid a situation where the robot is attempting to navigate around an obstacle but drift in the heading causes the robot to hit the obstacle anyways.

Chapter 6

# Improving the code base

Alongside the implementation of the functionality described in the previous chapters, attempts were made to document the code and improve the code base. This was done in collaboration with the students Emanuela Tuong Vi Thi Tran, Kristian Forsdahl Berg, and Magnus Isdal Kolbeinsen, as the four students have the same hardware and started out with the same code. It was important that the proposed method for code quality improvement still was within a reasonable scope.

An overview of the work done is found below. This text was written in collaboration with the other students and can be found in each student's individual thesis.

> “ The four students working on the SLAM Robot project this semester have made an attempt to improve the overall quality of the project, alongside our individual projects. The goal of this collaborative effort has been to ease the process of any future students, and to make the code easier to develop and maintain.
>
> The project contains large amounts of code with little to no regulation of code quality. Consequently, it has been regarded as hard to comprehend for newer students. This has in turn affected the progress of the SLAM Robot Project over time. Ideally, new students should be starting on the project where the previous students left off, but this has not been the case. Students have reported that a lot of time has been spent in the initial stage of learning how the robots work, which could be prevented if necessary information was easy to find.
>
> There is a large amount of information available regarding how the code works, and the reasoning behind certain features. However, there was no overview of which theses contained what information, and to get a full understanding of the current code you would have to read through a lot of the-

ses in order to find what is relevant. Additionally, it was up to each person to decide how to document their code and to write the code in any way they liked. This has caused a complete lack of structure, making it unnecessarily difficult to understand the code.

Upon completion of the project, a student would submit their work as a zip file containing their entire project. As multiple students were working on different parts of the robot, this would result in multiple zip files of code being delivered separately, with no designated main project. The following semester, the new students would then be presented with the question as to which project to base their work on, leaving documentation and work from the other prior projects behind. As follows, the general project would miss out on useful features and bug fixes throughout the revisions. Making use of the GitHub organization, one would open for version control and collaborative work on the projects, with the possibility of creating new repositories for experimental work, and merging these into the main project if they are decided to become a main feature. This way, new students will always have one place to look for their project material, and all documentation and code features will be preserved and accessible down the line.

Listed below is the work that has been done to improve the project and its workflow and code base :

- Weekly meetings to update on workflow and the state of the project
- Set name conventions for the software running on the robots nRF52840DK.
    - Change the code to comply with the name conventions
    - Deleted unnecessary comments
- Changed the file structure of the project
    - Changed file names
    - Divided the project into more appropriate sections
    - Changed the path of software running on the robots nRF52840DK.
- Created a GitHub organization
    - Added the projects' code to GitHub
    - Created git ignores
    - Deleted unnecessary and unused files in the project
    - Created appropriate repository names
    - Added informal readMe files to each repository.
- Created a Wiki for documentation of the project

- Added relevant project and master theses that have contributed to the project.
- Added information about the project in parts such as software, hardware, known bugs etc.
- Added How-To-Guides:
  - How to start and end the project after a work period or semester
  - How to create a new software project with FreeRTOS
  - How to edit the Wiki

”

*Berg, Kolbeinsen, Ruud-Olsen & Tran*

A list of the work that was specifically done by this author is found below. A more detailed description and explanation is found in the subsequent sections.

- Github organization and structure.
  - Split the original repository into several more fitting repositories.
  - Created gitignore files for all new repositories.
  - Deleted unnecessary and unused files before pushing the code to the new repository.
  - Created the organization-wide README and welcome message.
- Creation of a wiki as shown in Appendix A.
  - Created and wrote the page *Naming conventions.*
  - Created and wrote the page *C++ Server.*
  - Created and wrote the page *Get started.*
  - Created and wrote the page *Ending the project.*
  - Created and participated on the page *Previous work.*
  - Created and participated on the page *Known bugs.*
  - Created and participated on the page *Useful tips.*
  - Contributed to the front page *Home.*
- Improvement of the *robot-code.*
  - Suggested what aspects would need to be covered by the naming convention.
  - Participated in determining the naming conventions for the *robot-code.*
  - Created an extensive list of everything needing to change in order for the code to comply with the new naming convention.

- (In collaboration with Tran) Adapted the code to correspond with the new naming convention.
- (In collaboration with Tran) Changed the file names to be more informative and on the same format.
- Ensure that the implemented functionality remains functional after the renaming.

## 6.1 Naming conventions

The SLAM Robot Project involves a large number of students, and each student is only working on the project for a limited amount of time. When approaching this project, a naming convention will make it easier to learn how the code works and to start coding [10].

The naming conventions were mainly suggested by this author, and then it was either approved or changed after a discussion with the other students. Related responsibilities for this author included writing the naming conventions in a manner which is easily understood by future students, comparing the current robot code to the new convention, and creating an extensive list of changes that needed to be done. The convention can be found as a part of the wiki in Appendix A under the heading *Naming convention*.

The amount of different variable naming conventions was most noticeable in the robot code. At the start of this thesis, there were a lot of inconsistent names used. Examples include the tasks in the robot, called things such as *vMainPoseControllerTask* and *mapping_task*. The queues present in the robot also have different styles, for example *EncoderTicksToEstimatorTaskQ* and *ir_measurement_queue*. While each variable name is descriptive and works as intended on its own, the different styles make the overall readability of the project worse.

The students working on the SLAM Robot Project are fifth-year students, mostly from the Department of Engineering Cybernetics, who have been through several programming courses. The naming convention is not very formal or detailed, as it is assumed that the students are capable of creating variable names that are descriptive and follow basic concepts of good code practice. The focus has been to create a common convention for a more readable code.

While it is often true that any convention is better than none [10], there are still some guidelines for good naming conventions. It is deemed a good idea to differentiate between names for routines and variables and for this project *snake_case* was chosen for the former, and *camelCase* for the latter. This corresponds well to the language-specific tips found in [10] for *C* on Windows. Other tips include identifying global variables and constants. The former is done here by adding the prefix g (*gVariableName*), and the latter by using all upper case with underscores as a delimiter (*CONSTANT_NAME*).

The naming convention created for this project follows these guidelines, in addition to some specific adaptations. The robot code is written using a number of tasks that run in parallel. These tasks are of great importance, and thus it was decided that all tasks should include the *task_* prefix to make them easier to identify. In addition, some rules for best practice were included in the naming convention list, for instance using a *ifndef*, *define*, *endif* in the header files to avoid multiple definitions.

## 6.2 Documentation

A big issue with this project is that there have been many people involved and it is easy to lose track of where information is located. Students have approached the project by building their implementations on top of already existing code. The changes done are well documented and described in their respective reports, but there is no common place to find all relevant information about the code. If one desires an explanation of the code, it is necessary to read through several master theses and project reports made by a lot of different people.

As an alternative to this, documentation was created in the form of a wiki. The wiki is intended to provide useful information for a student to read as they start their work on the SLAM Robot Project. It consists of a brief explanation of the different parts of the available code (C++ server, robot code, and Java server) and the robot hardware. It also contains some useful experiences gain during this work, such as known bugs, weird behaviour and suggested debug methods if things don't work as expected. A list of some relevant previous master theses was included with some keywords describing the main topics. This is intended to make it easier to know where to start searching for more information. A page for naming conventions was added as described in Section 6.1. The resulting wiki of all this is attached in Appendix A.

In addition to the page about naming conventions, which is described above in section 6.1, this author was also solely responsible for the page *C++ Server*, *Get started* and *Finishing the project*, in addition to participating on *Known bugs*, *Useful tips*, and *Previous work*. The page about the C++ server was written with the intention to provide some useful tips for working efficiently with the server, in addition to a brief introduction to how it is constructed. Emphasis was put on not repeating too much of already documented information, but rather directing the readers to more detailed descriptions found in previous theses. Additionally, valuable experiences gained throughout this semester were added.

The pages on how to start and end working with the project were created to improve the workflow of the SLAM Robot Project and all its students. The aim of the initial instructions was to make it easy for a new student to find all relevant information for starting their project. Additionally, some tips are added on how to utilize GitHub the best way. The purpose of the instructions on how to leave the project was to ensure that as much of the completed work as possible becomes integrated into the code base. Furthermore, it instructed the user to document their work correctly to make it easy to find for future students.

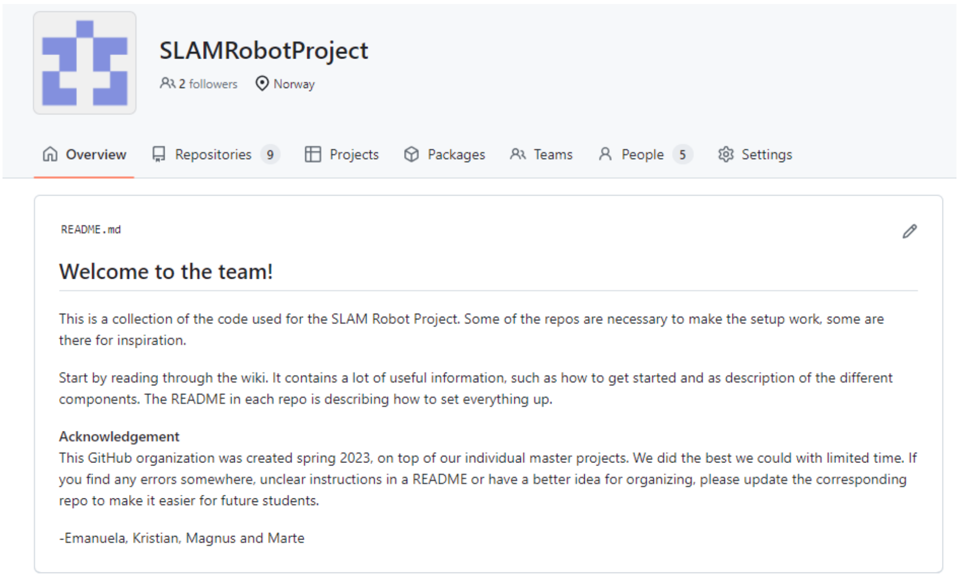
The pages *Known bugs* and *Useful tips* contain a lot of the experiences gained by the team this semester, and each student added any known bugs as they encountered them.

Additional useful information not being covered elsewhere was added in *Useful tips*. This includes some debugging tips and some information on some of the software utilized.

The page *Previous work* is intended to be a starting point for exploring the previous work done by other students. It contains a list of relevant theses, along with some keywords describing the main features of the thesis. Each member added the theses they found useful during their work, as well as briefly describing their own thesis.

## 6.3 GitHub organisation and structure

A GitHub organization called *SLAMRobotProject* was created. An organization makes it easier to keep all relevant repositories in the same place. New students will then only need to be added to the organization, instead of several separate repositories. The front page of the organization is shown in Figure 6.1. It contains a small welcome message in addition to providing access to the nine repositories.



Welcome to the team!

This is a collection of the code used for the SLAM Robot Project. Some of the repos are necessary to make the setup work, some are there for inspiration.

Start by reading through the wiki. It contains a lot of useful information, such as how to get started and as description of the different components. The README in each repo is describing how to set everything up.

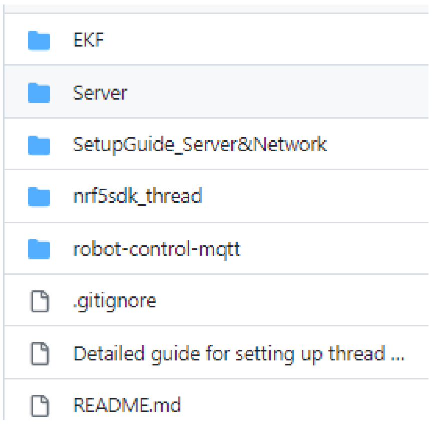
**Acknowledgement**
This GitHub organization was created spring 2023, on top of our individual master projects. We did the best we could with limited time. If you find any errors somewhere, unclear instructions in a README or have a better idea for organizing, please update the corresponding repo to make it easier for future students.

-Emanuela, Kristian, Magnus and Marte

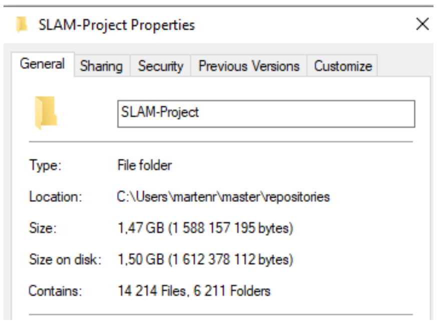
**Figure 6.1:** Front page of the GitHub organization.

The code received at the beginning of this thesis was the result of work done by Frestad in [32], where the code used by Andersen [31] was adapted and then moved to GitHub. The result is as displayed in Figure 6.2a: a single repository containing five folders (EKF, Server, SetupGuide_Server&Network, nrf5sdk_thread, and robot-control-mqtt), in addition to a basic gitignore, a PDF version of the setup guide, and a short README. The total size of the repository once cloned was 1.5 GB spread over more than 14 thousand files, as seen in Figure 6.2b.

It was the job of this author to split this repository into more appropriate sections, find


EKF
Server
SetupGuide_Server&Network
nrf5sdk_thread
robot-control-mqtt
.gitignore
Detailed guide for setting up thread ...
README.md

**(a)** GitHub structure of the original repository



SLAM-Project

Type: File folder
Location: C:\Users\martenr\master\repositories
Size: 1,47 GB (1 588 157 195 bytes)
Size on disk: 1,50 GB (1 612 378 112 bytes)
Contains: 14 214 Files, 6 211 Folders

**(b)** Size of the original GitHub repository

**Figure 6.2:** Code structure and size at the start of this project in the *SLAM-Project* repository.

appropriate names and then create the repositories on the new GitHub organization. Additionally, it was the responsibility of the author to create functioning *gitignore* files, and to make sure that the untracked files did not end up being in the new repository. The process of doing so and the reasoning behind it is described further below.

Neither of the folders in Figure 6.2a was actually dependent on each other and did very different things. Apart from *SetupGuide_Server&Network*, separate repositories were created for each of the folders within the new GitHub organization. The setup guide contained information relevant to both the C++ server and the robots, and consequently the guide was split and the relevant parts moved to their respective new repositories *cpp-server* and *robot-code*.

The original GitHub repository contained a large number of files, some of which were not necessary to make the project build. Examples of such files are object files, libraries, and build results. These files are of no interest to other users and can be safely ignored by Git. This is achieved by adding a *gitignore* file, which is responsible for telling Git which files not to keep track of [38]. The original repository did contain a very basic *gitignore*, as seen in Figure 6.3, however, it was not sufficient and a lot of unwanted files remained.

```
Server/SLAM-application/logs/
nrf5sdk_thread/examples/thread/freertos_mqttsn/pca10056/blank/ses/Output/
**/Debug/*
*.lnk
```

**Figure 6.3:** Original *gitignore* file.

The newly created repositories in the new GitHub organization were all equipped with more extensive *gitignore* files. As adding a *gitignore* does not affect files that are already tracked by Git [38], it is essential to delete all the unnecessary files prior to pushing the code to the new repository. This ensures that the untracked files do not end up in the

GitHub history, and the repository ends up looking a lot cleaner. This process reduced the size of the repositories greatly. The EKF folder was left as it is, merely an inspiration for future students, as the work done there has not been finished and merged into the robot code [31]. The only change was the name, which was adapted to better reflect its content. A summary of the changes done, with the name change and the reduction in size, is shown in Table 6.1.

**Table 6.1:** Reduction of folder size after deleting unnecessary files.

| Previous folder name | Current repository name | Previous size | Current size |
|---|---|---|---|
| nrf5sdk_thread | robot-code | 1,22 GB | 552 MB |
| robot-control-mqtt | python-robot-control | 332 KB | 18 KB |
| Server | cpp-server | 82,7 MB | 2,76 MB |
| EKF | EKF-SLAM | 5,5 MB | 5,5 MB |

The code for controlling the robots was originally found in the folder *nrf5sdk_thread*, and needed additional work before being placed in its new separate repository. This folder was originally acquired from Nordic [31], and the project was created within an example project. As a result, the path for navigating to the robot SES project file was *\nrf5sdk_thread \examples \thread \frertos_mqttsn \pca10056 \blank \ses \thread_freertos_coap_server_pca10056.emProject*. The folder also included all the other examples provided by Nordic and a lot of other files that were not actually utilized. Work was done by Tran to improve the file structure, while this author was responsible for creating a proper *gitingore* and removing the files related to this. In the end, the folder was reduced by from 1.22GB to 466MB and the path to the project file is simply *\robot_code \robot_code.emProject*.

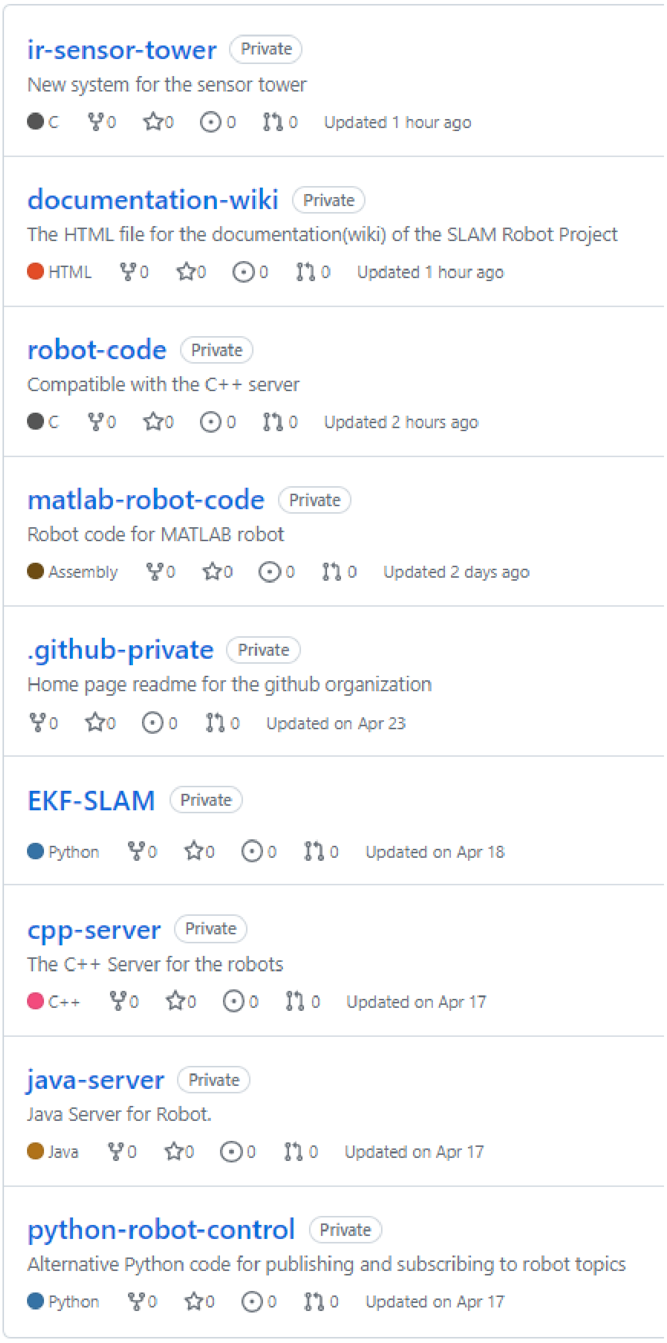
Ideally, it should be sufficient to download the correct version of the SDK from the Nordic website in order to make the system run. However, Andersen adapted the SDK as a part of implementing MQTT-SN during his work in [31]. While this is considered bad practice, changing this would require a major rewriting of Andersen's work and was not a part of the improvements done. Instead the SDK was left as a part of the *robot-code* repository.

After the refactored robot code was functional and in its new repository, the work began to update its code to match the naming conventions set in the wiki. This included changing all function names to snake case, all variables to camel case, and changing the names of queues and tasks, in addition to the files themselves. This was cumbersome work done mainly in collaboration with Tran. This author was responsible for making sure that the project still worked as intended, which was achieved by testing the robot's functionality, and fixing any error that was caused by the name changes.

The resulting structure of the new GitHub organization, as it is at the end of this project, is shown in Figure 6.4, with a short description added in Table 6.2. In addition to the repositories mentioned in Table 6.1, five other repositories were created as a part of the mission to have all relevant code in one place. The Java alternative to using the C++ server can be found in *java-server*. *wiki* and *.github-private* were created by this author for storing respectively the wiki and a welcome message and acknowledgements to new members of the GitHub organization. The remaining two repositories contain the work done by other students concurrently with this thesis.

**Table 6.2:** Repositories in the *SLAMRobotProject* GitHub organization.

| Name | Language | Brief description |
|---|---|---|
| ir-sensor-tower | C | Work by Tran |
| documentation-wiki | HTML | The HTML file for the wiki |
| robot-code | C | Robot code |
| matlab-robot-code | Assembly | Work by Kolbeinsen |
| .github-private | - | Stores the organization-wide README |
| EKF-SLAM | Python | Work from [31] regarding EKF-SLAM |
| cpp-server | C++ | C++ server |
| java-server | Java | Java server |
| python-robot-control | Python | Basic alternative for communicating with the robot |


ir-sensor-tower Private
New system for the sensor tower
● C 0 0 0 0 Updated 1 hour ago

documentation-wiki Private
The HTML file for the documentation(wiki) of the SLAM Robot Project
● HTML 0 0 0 0 Updated 1 hour ago

robot-code Private
Compatible with the C++ server
● C 0 0 0 0 Updated 2 hours ago

matlab-robot-code Private
Robot code for MATLAB robot
● Assembly 0 0 0 0 Updated 2 days ago

.github-private Private
Home page readme for the github organization
0 0 0 0 Updated on Apr 23

EKF-SLAM Private
● Python 0 0 0 0 Updated on Apr 18

cpp-server Private
The C++ Server for the robots
● C++ 0 0 0 0 Updated on Apr 17

java-server Private
Java Server for Robot.
● Java 0 0 0 0 Updated on Apr 17

python-robot-control Private
Alternative Python code for publishing and subscribing to robot topics
● Python 0 0 0 0 Updated on Apr 17

**Figure 6.4:** Current repositories in the *SLAMRobotProject* GitHub page.

Chapter 7

# Further work

While the project has been further developed and the code base has been improved as a part of this thesis, more work is needed to get the project to the desired goal.

The homing procedure implemented is intended to be only for emergencies. It is not very advanced, but it does not need to be either. However, it will be of no use if it is not initialized when needed. The testing done during this thesis used a fixed amount of targets reached to initialize the homing. This is not realistic in many scenarios, as it is simply not necessary to return home at that point. A way of determining when an event happens that is important enough to trigger the homing procedure will need to be implemented. One scenario could be if the server was to disconnect or get turned off. The robot will then have to notice that the server is not working and then initialize the homing procedure.

The implemented obstacle avoidance during the homing procedure has room for improvement as well, as it is very basic. While the current version will work in any scenario where the obstacles are not dynamic, it is not necessarily the most efficient way. One could use the sensor inputs to gain an understanding of the shape of the obstacle, and then create a path around it. That way the obstacle avoidance will not be reliant on the previous motion. Using an edge detection technique is considered a good option. Work is also needed to make the robot able to react fast enough to a detection, in order for it to not collide with the obstacle.

During the work on this project, it became apparent that the robot's pose estimate is not always accurate. Usually, this presents itself as accumulative errors in the sensors and causes the estimate to worsen as the robot drives. Sometimes, however, the robot makes some sudden movements which seem to interfere with the estimator. The robot will then move in a different direction than expected and stop at a target that is nowhere close to the position it is supposed to be in. This will need to be looked into to make the response of the robot more consistent and reliable.

The code basis and organizational structure of the project were greatly improved by the collaborative effort done throughout the semester, however many areas of future improvement still exist. While the variable and function names were changed to match the new naming conventions, no checks were performed to make sure that all of them are actually utilized. As a result, there may exist code that is unused and can be safely removed.

In order for the SLAM Robot Project to remain at the level of organization it currently is, it is essential that future students maintain the wiki and use GitHub to its full potential. The wiki is now a good resource for information on the system, but it will soon become outdated if new implementations, bugs and changes are not documented properly. The *SLAMRobotProject* GitHub page is also important to maintain in order for its potential to be utilized. That includes updating the *README*s and working in separate branches to avoid interfering with the original code.

As the scope of this work was limited, there remains quite a lot of work before the SLAM Robot Project is at the desired stage. No attempts have been made to refactor the robot code, which is sorely needed. The code is still not easy to maintain and understand, and would greatly benefit from becoming more modular. Implementing new features on top will only make matters worse, so proper refactoring ought to be prioritized.

# Bibliography

[1] L. S. Davis, “A survey of edge detection techniques,” *Computer graphics and image processing*, vol. 4, no. 3, pp. 248–270, 1975.

[2] O. Khatib, “Real-time obstacle avoidance for manipulators and mobile robots,” in *Proceedings. 1985 IEEE international conference on robotics and automation*, IEEE, vol. 2, 1985, pp. 500–505.

[3] J. Borenstein and Y. Koren, “Obstacle avoidance with ultrasonic sensors,” *IEEE Journal on Robotics and Automation*, vol. 4, no. 2, pp. 213–218, 1988.

[4] J. Borenstein and Y. Koren, “Real-time obstacle avoidance for fast mobile robots,” *IEEE Transactions on systems, Man, and Cybernetics*, vol. 19, no. 5, pp. 1179–1187, 1989.

[5] J. Borenstein, Y. Koren, *et al.*, “The vector field histogram-fast obstacle avoidance for mobile robots,” *IEEE transactions on robotics and automation*, vol. 7, no. 3, pp. 278–288, 1991.

[6] S. Quinlan and O. Khatib, “Elastic bands: Connecting path planning and control,” in *[1993] Proceedings IEEE International Conference on Robotics and Automation*, IEEE, 1993, pp. 802–807.

[7] H. Choset, “Coverage for robotics–a survey of recent results,” *Annals of mathematics and artificial intelligence*, vol. 31, pp. 113–126, 2001.

[8] S. M. LaValle and J. J. Kuffner Jr, “Randomized kinodynamic planning,” *The international journal of robotics research*, vol. 20, no. 5, pp. 378–400, 2001.

[9] R. Carelli and E. O. Freire, “Corridor navigation and wall-following stable control for sonar-based mobile robots,” *Robotics and Autonomous Systems*, vol. 45, no. 3-4, pp. 235–247, 2003.

[10] S. McConnell, *Code Complete: A practical handbook of software construction*, 2nd ed. Microsoft, 2004.

[11] H. Skjelten, *Fjernnavigasjon av LEGO-robot: Project report for TTK4551*, Norwegian University of Science and Technology, 2004.

[12] M. N. Rooker and A. Birk, “Multi-robot exploration under the constraints of wireless networking,” *Control Engineering Practice*, vol. 15, no. 4, pp. 435–445, 2007.

[13] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, “Mqtt-s—a publish/subscribe protocol for wireless sensor networks,” in *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE’08)*, IEEE, 2008, pp. 791–798.

[14] G. Grisetti, R. Kümmerle, C. Stachniss, and W. Burgard, “A tutorial on graph-based slam,” *IEEE Intelligent Transportation Systems Magazine*, vol. 2, no. 4, pp. 31–43, 2010.

[15] S. Tully, G. Kantor, and H. Choset, “Leap-frog path design for multi-robot cooperative localization,” in *Field and Service Robotics: Results of the 7th International Conference*, Springer, 2010, pp. 307–317.

[16] S. Karaman, M. R. Walter, A. Perez, E. Frazzoli, and S. Teller, “Anytime motion planning using the rrt,” in *2011 IEEE international conference on robotics and automation*, IEEE, 2011, pp. 1478–1483.

[17] B. Englot and F. S. Hover, “Three-dimensional coverage planning for an underwater inspection robot,” *The International Journal of Robotics Research*, vol. 32, no. 9-10, pp. 1048–1073, 2013.

[18] E. Galceran and M. Carreras, “A survey on coverage path planning for robotics,” *Robotics and Autonomous systems*, vol. 61, no. 12, pp. 1258–1276, 2013.

[19] G. Papadopoulos, H. Kurniawati, and N. M. Patrikalakis, “Asymptotically optimal inspection planning using systems with differential constraints,” in *2013 IEEE International Conference on Robotics and Automation*, IEEE, 2013, pp. 4126–4133.

[20] A. Stanford-Clark and H. L. Truong, “MQTT for sensor networks (MQTT-SN) protocol specification,” *International business machines (IBM) Corporation version*, vol. 1, no. 2, pp. 1–28, 2013.

[21] A. Bircher, M. Kamel, K. Alexis, *et al.*, “Three-dimensional coverage path planning via viewpoint resampling and tour optimization for aerial robots,” *Autonomous Robots*, vol. 40, pp. 1059–1078, 2016.

[22] T. Taketomi, H. Uchiyama, and S. Ikeda, “Visual slam algorithms: A survey from 2010 to 2016,” *IPSJ Transactions on Computer Vision and Applications*, vol. 9, no. 1, pp. 1–11, 2017.

[23] R. Hussain and S. Zeadally, “Autonomous cars: Research results, issues, and future challenges,” *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1275–1313, 2018.

[24] M. Azad, N. Hoseinzadeh, C. Brakewood, C. R. Cherry, and L. D. Han, “Fully autonomous buses: A literature review and future research directions,” *Journal of Advanced Transportation*, vol. 2019, pp. 1–16, 2019.

[25] V. S. Kalogeiton, K. Ioannidis, G. C. Sirakoulis, and E. B. Kosmatopoulos, “Real-time active slam and obstacle avoidance for an autonomous robot based on stereo vision,” *Cybernetics and Systems*, vol. 50, no. 3, pp. 239–260, 2019.

[26] Oasis, *MQTT Version 5.0 - Oasis standard*, 2019. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf.

[27] T. Dang, M. Tranzatto, S. Khattak, F. Mascarich, K. Alexis, and M. Hutter, "Graph-based subterranean exploration path planning using aerial and legged robots," *Journal of Field Robotics*, vol. 37, no. 8, pp. 1363–1388, 2020.

[28] M. S. Mullins, "Implementation of Simultaneous Localisation and Mapping in Robotic System using the improved Rao-Blackwellized Particle Filter," M.S. thesis, Norwegian University of Science and Technology, 2020.

[29] A. Stenset, "nRF52 robot with OpenThread," M.S. thesis, Norwegian University of Science and Technology, 2020.

[30] I. Lluvia, E. Lazkano, and A. Ansuategi, "Active mapping and robot exploration: A survey," *Sensors*, vol. 21, no. 7, p. 2445, 2021.

[31] T. Andersen, "Sparse IR sensor EKF-SLAM for MQTTSN/Thread connected robot," M.S. thesis, Norwegian University of Science and Technology, 2022.

[32] H. Frestad, *The SLAM project: Project report for TTK4551*, Norwegian University of Science and Technology, 2022.

[33] CanaKit, *UK1122: L298 H-Bridge Dual Bidirectional Motor Driver*, 2023. [Online]. Available: https://www.canakit.com/Media/Manuals/UK1122.pdf.

[34] T. Corporation, *ICM-20948*, 2023. [Online]. Available: https://invensense.tdk.com/products/motion-tracking/9-axis/icm-20948/.

[35] Eclipse, *Eclipse Mosquitto: An open source MQTT broker*, 2023. [Online]. Available: https://mosquitto.org/.

[36] Eclipse, *paho: MQTT-SN Transparent Gateway*, 2023. [Online]. Available: https://www.eclipse.org/paho/index.php?page=components/mqtt-sn-transparent-gateway/index.php.

[37] FreeRTOS, *FreeRTOS: Real-time operating system for microcontrollers*, 2023. [Online]. Available: https://www.freertos.org/index.html.

[38] Git, *gitignore - Specifies intentionally untracked files to ignore*, 2023. [Online]. Available: https://git-scm.com/docs/gitignore.

[39] Microsoft, *Download Visual Studio*, 2023. [Online]. Available: https://visualstudio.microsoft.com/vs/older-downloads/.

[40] OptiTrack, *Motive: Optical motion capture software*, 2023. [Online]. Available: https://optitrack.com/software/motive/.

[41] M. org, *MQTT: The Standard for IoT Messaging*, 2023. [Online]. Available: mqtt.org.

[42] SEGGER, *Embedded Studio for ARM*, 2023. [Online]. Available: https://www.segger.com/downloads/embedded-studio/.

[43] N. Semiconductor, *nRF52840-DK*, 2023. [Online]. Available: https://www.nordicsemi.com/Products/Development-hardware/nrf52840-dk.

[44] Sharp, *GP2Y0A21YK0F*, 2023. [Online]. Available: `https://global.sharp/products/device/lineup/data/pdf/datasheet/gp2y0a21yk_e.pdf`.

# Appendices

Appendix A

# SLAMProject documentation

This is a PDF version of the documentation. Full version of the wiki, with hyperlinks and proper layout, can be found in the *documentation-wiki* repository on the SLAMRobot-Project GitHub page.

# Home

20th April 2023 at 3:39pm

home

Welcome to the hub for information management regarding the SLAM projects. Below you will find some useful links to get started. If you wish to make changes to the wiki, visit the **Edit Wiki** page.

## Setting up

In this section you will find all information needed to **Get Started** with the SLAM project. In order to make the project as clean as possible, make sure you know how to use ***GitHub*** and the **Naming conventions**.

## System architecture

The following pages contain a generalized overview of the robot control system, from server to robot, and from hardware to software.

- **C++ Server**
- **Java server**
- **Robot HW**
- **Robot SW**

## Useful information

Familiarize yourself with the good-to-know aspects of the project, such as **Known bugs**, **Useful tips** etc. The theses of previous students are a good way to learn more about the system, and an overview of some of them are found in **Previous work**.

Please also know what to do when **Finishing the project** to make it easier for future students.

# Edit Wiki

13th April 2023 at 1:27pm

how-to

For making changes to the wiki, there are quite a few things to keep in mind. This page is intended to help you out by providing some information on how this wiki is structured, and some tips and tricks on how to navigate.

## Making a new page

In order to make a new page, navigate to the «+» sign located beneath the title on the top right section of the screen. A blank page will appear on the left hand side. These pages are natively called «tiddlers», and will be referred to as such by the system. The new pages will be tagged as «untagged» by default, remember to remove this tag and provide a suitable tag, preferably from the existing selection to maintain a consistent overview, but in case noone suits the use properly, you can make your own. The idea is to keep it simple.

## Formatting

When writing your tiddler, the toolbar on top of the text editor contain some helpful tools, but learning how to perform these things manually can also be time saving so here are some helpful tips:

- **Headers** In order to make headers, simply write an exclamation mark "!" at the start of the respective line, followed by a space and your header text. The amount of exclamation marks correspond to the header types, i.e. "!" corresponds to Header 1, "!!" to Header 2 etc.
- **Linebreak** To make a linebreak, simply hit enter two times. two linebreaks in the text editor results in a single linebreak in the output.
- **Fontstyles** To make text **bold**, enclose it in double apostrophes ( ' ). For *italic* style, enclose it in double slashes ( / ). For underlined text, enclose it in double underscores ( _ ). For overlined text, enclose it in double tildes ( ~ ).
- **Images** To include images, firstly make sure the image is imported using the binder icon beneath the wiki title on the top right of the screen. If you can't find it there, you can find it in the Tools tab located in the same area. Make sure the image file has a simple name. Then, include it in the text by making double brackets, with "img" between the first and second bracket, followed by the filename inside the second bracket. Example [ img [ filename.png ] ] without the spacing.
- **Bullet points** are made using the star symbol "*" followed by space and your text. Lists are not made automatically, so you will need to make a linebreak and follow up with a new bullet point manually.
- **Links to other tiddlers** If you want to reference another tiddler, you can link it by simply encasing the title of the page in double brackets. For example, [ [ Home ] ] will create a link to **Home**

## Saving the wiki

When you want to save your changes, make sure all the tiddlers, i.e. the "pages" you've made changes to, are saved by clicking the checkmark in the top right corner of each tiddler. To make the wiki a little less messy for the next person opening it, make sure to hit the Home button before the final save. This closes all open tiddlers except from the start page. Then, you can save the entire wiki by hitting the red circle icon on the top right of the screen, under the project title. Any unsaved tiddlers will lose their changes when you press this button. If you are working directly in the html-file, you are working offline and saving will produce a new html-file in your download folder for you to share with the project members. If you have uploaded the file to TiddlyHost, which is the online host service, the changes will be saved to your online project and you will need to download the updated project to an html file in order to share it.

**Note:** The system has no guarantees for your unsaved changes to be stored, and so if the browser you're working in goes idle or anything like that, the page may be refreshed and you might lose all your progress so **make sure you save often.**

# Get Started

15th May 2023 at 2:00pm

how-to

## Before you start

Make sure you read through this wiki. It contains a lot of useful information and details on where to find more. There are also helpful insights from previous students that will hopefully decrease the time needed to understand the setup and code.

Familiarize yourself with the ***GitHub*** organization and its repositories. What repos are necessary for you depends on what version you are going to work on.

- C++ Server and robots
  - You will need to clone the repositories **cpp-server** and **robot-code**. A detailed guide on how to set up the server and how to flash code to the robots are found in their respective repos.
- Java server and robots
  - The server is available in **Java-Server**
  - What robot-code you want to base your project on depends on your task.
    - The code in **robot-code** is based on the work from Frestad(2022), and a naming convention was added and the file structure greatly improved spring 2023. The functionality for using the Java server was not removed, but the updated version was only tested with the C++ server. It is quite possible that this robot code will work with the Java server, but no guarantees can be made.
    - The code in **robot-code-java-server** is based on the work done by Andersen (2022), but it is without the work done by Frestad(2022) and the improvements done to **robot-code** during spring 2023. It is however guaranteed to be compatible with the Java server.

When you start your work, make sure that you work on a separate branch, and not directly into the main branch.

There are six robots available for this project, named NRF1 through NRF6. The robots have some differences in hardware. Make sure you choose the correct robot in *robot_config.h* in the robot code.

## Charging the robots

- The orange covers on the wire are there to protect them. Remove these on the two free wires at the front of the robot.
- Connect the black charger plug to the black wire, and the red charger plug to the red wire.
- Flip the charger switch on the robot.
- Connect to power.

# Naming conventions

14th April 2023 at 1:46pm

how-to

When beginning to work on this project, there is a lot of code to understand and read through. In addition, a lot of people have contributed to the current code with their own manner of programming. Consistent use of the naming conventions will improve the readability greatly. Note that these naming conventions, especially the delimitation, applies mainly to the robot code. The C++ server is written with snake case for everything. Whatever repo needs work done, please think through how to name things to make it easier for everyone.

Delimitation of words:

- Snake case (snake_case) is to be used for functions and structs.
  - The tasks should be prefixed with "task_". Eg. task_sensor_tower
- Camel Case (camelCase) is the default way to name variables.

To further differentiate between some key concepts an initial letter can be used on the variables:

- xName for semaphores and mutexes.
- gName is used for global variables.
- pName is used for pointers.
- qName is for queues (eg. qEncoderTicks)

Some general advice for best practice:

- Boolean variables should be named so it is easy to understand what its value means. Usually, this means including words as is, was, has and should, for example should_update and is_available. Avoid using not as that causes non-intuitive results of true and false.
- Header files ought to include
  - *#ifndef EXAMPLE_H* and *#define EXAMPLE_H* at the top of the file
  - *#endif /*EXAMPLE_H*/* at the end of the file
- #define constants should be capitalized
  - #define BUFFER_LENGTH 6

# C++ Server

15th May 2023 at 2:03pm

architecture

The C++ server is a multithreaded process that can be run in Visual Studio 2019. Frestad (2022) contains a more detailed description of details of the different threads. Information about MQTT, MQTT-SN and the communication between the robots and the server can be found in Andersen (2020). An updated guide for how to set up the server correctly can be found in the repo **cpp-server**.

*main.cpp* sets the MQTT address to the Raspberry Pi. It also creates a couple of call-back functions, and is responsible for starting the five threads responsible for respectively the GUI, MQTT, robot inbox, robot outbox and simulation. The necessary data for each robot is found in the robot class in robot.cpp. It is *robots.cpp* that handles the updating of the different robots as new updates arrive from the inbox. *robots.cpp* also calculates the new target destination.

When running the program, a GUI window appears. Some of the important parts of this window are the possibility to choose between simulation and physical robots, register robots, start exploration of the area, and choose manual input. Note that switching between simulation and real driving also requires one line of code to be changed. As an alternative to manual input, one can also set target position by right clicking on the screen. The IR readings from the robots are displayed in the GUI as well, with possible obstacles displayed as black areas.

The simulation is handled by a separate thread. When the simulation is running, it reacts to inputs from the GUI window in the same manner a robot would. By letting the simulated robot explore, it will eventually reconstruct the predefined map it is in.

Incoming messages from the robots are registered by call-backs in the MQTT thread in *matt_handler.cpp*. If the message is from a simulation, the message is passed on to the simulation thread. Otherwise, it is passed on to the robot inbox thread, via a channel called slam_ch, which again forwards it to robots.cpp for further updating. The robot outbox thread is responsible for sending the correct new position to the correct robot and passing it on to the MQTT thread.

Choosing “Search Grid” after creating a connection to a robot will lead to the server choosing a target point and sending it to the robot. When the robot has moved, the server will add a new random path to its path. When “Enable Manual Drive” is checked, the server will send whatever value is in the Manual Input GUI slots with even intervals, roughly every second. Note that it does not have any concept of whether you finished writing your target points, and will potentially send an unfinished position, before sending the correct one.

When using the manual inputs, be aware of the coordinate frames used. The targets are written on the form (x,y) with millimetres as the unit. The robot’s internal coordinate frame has the positive x-axis going straight ahead, and the positive y-axis to the left. In addition, somewhere along the line this input is multiplied by two. As a result, setting an input (0,-300) will take the robot 600mm directly to the right of its starting point.

# Java server

24th April 2023 at 4:35am

architecture

The Java server is a server application that provides real time features to control the robots. By using BLE communication through nRF51 Dongle it is able to communicate with multiple robots and extract data from them, and ultimately send position commands to them. The server also contains a GUI with a map drawn from sensor data, and a simulator that can be used for testing.

It is based on a older implementation in MATLAB which was first introduced in 2005 by Syvertsen. The Java server was built in 2016 by Andersen and Rødseth. More details can be found in their report.

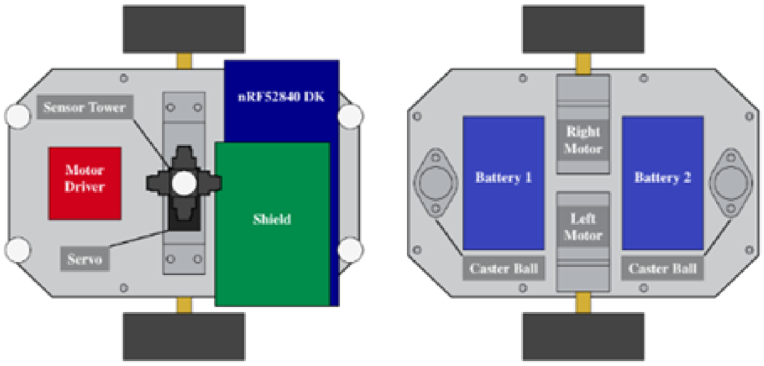
In order to use the Java server to control the robots, it needs to be flashed with the Java server robot code. A nRF 51 dongle with the correct hex file is also needed.

A user manual to the Java server can be found in the **SSNAR - cart/manuls** folder in the java-server repository. Master thesis' reports working with and alongside the java server can be found in the **reports/** folder in the repository.

# Robot HW

29th May 2023 at 4:24pm

architecture



The figure above is an illustration of the robots hardware. The left side of the figure shows the view above the robot and the right side shows the bottom side of the robot.

## nRF52840DK Shield

The nRF52840 shield was custom made by Jølsgard and is attached on the top of the nRF52840 DK. The purpose of the shield is to connect all peripherals to the GPIO headers of the development kit.

## nRF52840DK

The nRF52840-DK development kit from Nordic Semiconductor serves as the primary computing module for the robot. The nRF52840 System on a Chip on the development kit supports Bluetooth low energy, Bluetooth mesh, NFC, Matter, Thread, and Zigbee. It is compatible with Arduino UNO Revision 3, making it easy to use third-party shields. The development kit comes equipped with an on-board SEGGER J-Link debugger that allows for programming and debugging of both the on-board SoC and external targets through the debug header. The nRF52840 DK can be powered through a USB connection, but it also has the capability to be powered by a wide range of external sources within the 1.7 to 5 volt supply range. In addition to USB, it has a CR2032 battery holder and a Li-Po battery connector, giving it the flexibility to be powered by various sources depending on the needs of the project.

## Sensor tower

The sensor tower is made up of four 2YA21 Sharp infrared sensors. These sensors are mounted on a S05NF servo motor and arranged radially with respect to each other, allowing the set of IR sensors to rotate. The sensor tower has a maximum rotation angle of 90 degree and each of the IR sensors have a valid measurement range of 0.1 to 0.8 m.

## Motor driver

The L298 motor driver is a bi-directional motor driver based on the L298 Dual H-Bridge Motor Driver Integrated Circuit. According to its data sheet, it is ideal for robotic applications and can be easily connected to a microcontroller, requiring only a few control lines per motor. It is capable of controlling two motors at up to 2 A each in both directions.

## DC motors + encoders

The robot uses two 12V DC-motors from Machifit. Which are rated up to 100 rpm. Each motor has built-in quadrature encoders for measuring wheel angle and speed.

## IMU

The ICM-20948 Measurement Unit is made up of a 3-axis MEMS-based gyroscope, accelerometer, and compass. It is located underneath the robot's chassis and is equipped with a digital motion processor that is used for simple filtering of sensor measurements and power management. The IMU is not illustrated on the figure above.

## Sensor data acquisition board

In addition to the existing hardware on the robot, there is an ongoing project that works with splitting up the hardware on the robot.

This project is under development and a custom printed circuit board has been created. Read more about the Sensor data Acquisition board here: **Sensor Data Acquisition Board**

# Robot SW

29th May 2023 at 4:40pm

architecture

## Tutorials on how to create new software project for a new system

The following are tutorials on how to create a new software project with ***FreeRTOS***. Method 1 creates a project from scratch by including libraries. The 2nd method uses a nRF SDK example project as its base. The 2nd method is preferred as it is easier when troubleshooting.

**How to create new nrf software project method 1**

**How to create new nrf software project method 2**

# How to create new nrf software project method 1

29th May 2023 at 3:27pm

how-to nrf52840 software

1. Start Embedded Studio.
2. (optional) install CPU support Package for your device family via Tools → Package Manager. Nordic Semiconductor nRF CPU Support Package were used here.
3. Create new project via File → New Project → Create the project in a new solution.
    1. Set a project name. If no CPU package is used then select "A C/C++ executable for a Cortex-M processor". Press Next, select your target device and keep pressing Next until your project is finished.
    2. If you are using a CPU support package select the "A C/C++ executable for..." from the corresponding package and finish your project as described in the point above.
4. Connect the nRF52840DK to a PC. Ensure that both the Jlink on the development board and the nRF52840 are powered, not just the nRF52840.
5. Give your newly created project a try by building it with F7 and executing it with F5.
6. Create a new folder /lib/***FreeRTOS***/***FreeRTOS***-Kernel in both the project explorer in Embedded Studio and on your hard drive in the project folder.
7. Download and unpack the ***FreeRTOS*** software to any location.
8. In the unpacked folder open folder /***FreeRTOS*** and copy the folder /***FreeRTOS***/Source to the /lib/***FreeRTOS***/***FreeRTOS***-Kernel in your ES project folder.
9. Add the same files to the Embedded Studio project explorer. The easiest way is to drag and drop the folder onto the /lib/***FreeRTOS***/***FreeRTOS***-Kernel folder.
10. Right click the new folder and select Setup. Check the box "Recurse into Subdirectories" and press OK.
11. This should add all ***FreeRTOS*** sources to your setup. However not all files are needed so the wrong files have to be removed again. To do this first convert the folder to a regular folder by right clicking it and select "Convert to regular folder".
12. You will need all .c source files from the /Source folder. All include files from /Source/include and the folders /Source/portable/GCC and /Source/portable/***MemManage***.
13. All other folders and files can be safely removed by simply selecting them and pressing the DEL key our right click and delete.
14. Next make sure that in /Source/portable/***MemManage*** you only have one .c file selected e.g. heap 1.c. Remove all other .c files, otherwise the project will later not build.
15. In /Source/portable/GCC make sure that only the folder is included that is the architecture of your target platform. In our example it is a Cortex-M4 target device so only folder /Source/portable/GCC/ARM CM4F stays. All other folders can be safely removed as before.
16. Next you will need to create a ***FreeRTOSConfig***.h file which will configure your ***FreeRTOS*** setup. For references see the ***FreeRTOS*** documentation or use one of the config headers from the many samples out there as reference. For example the one from the example project above. We recommend to place this file into your source folder where your main.c file is. In this case it is folder /source in the project folder.
17. Next all include paths need to be set. You can add this in project options under Project → Options → Preprocessor → User Include Directories. If you are using the same folder structure as recommended the following three include paths must be set. If you are using another project structure adjust the paths accordingly.
    1. $(***ProjectDir***)/source

   2. $(***ProjectDir***)/lib/***FreeRTOS***/***FreeRTOS***-Kernel/Source/include
   3. $(***ProjectDir***)/lib/***FreeRTOS***/***FreeRTOS***-Kernel/Source/portable/GCC/ARM CM4F
18. Now edit your main.c to include ***FreeRTOS***.h and task.h and add your ***FreeRTOS*** application code to the main.c.
19. (optional) add your third party libraries, HALs, drivers etc. to your project by including their paths. In this project the nRF SDK folder is outside of this project folder. The paths for the nRf SDK therefore start with "../". Your paths should look something like this:
   1. $(***ProjectDir***)/source
   2. $(***ProjectDir***)/lib/***FreeRTOS***/***FreeRTOS***-Kernel/Source/include
   3. $(***ProjectDir***)/lib/***FreeRTOS***/***FreeRTOS***-Kernel/Source/portable/GCC/ARM CM4F
   4. ../nRF5 SDK 17.1.0 ddde560/modules/nrfx
   5. ../nRF5 SDK 17.1.0 ddde560/components/boards
   6. ../nRF5 SDK 17.1.0 ddde560/components/libraries/util
   7. ../nRF5 SDK 17.1.0 ddde560/modules/nrfx/hal
   8. ../nRF5 SDK 17.1.0 ddde560/modules/nrfx/drivers
   9. ../nRF5 SDK 17.1.0 ddde560/integration/nrfx/legacy
   10. ../nRF5 SDK 17.1.0 ddde560/modules/nrfx/templates
   11. ../nRF5 SDK 17.1.0 ddde560/modules/nrfx/templates/nRF52840

Once all this is done your application should build now and you should be able to debug a ***FreeRTOS*** application in Embedded Studio.

# How to create new nrf software project method 2

29th May 2023 at 3:29pm

how-to nrf52840 software

**Step 1: Download nRF5 SDK from Nordic Semiconductor**

- https://www.nordicsemi.com/Products/Development-software/nrf5sdk/download
- The version used for this project is 17.1.0

**Step 2: Make a copy of the downloaded folder and rename to desired project name**

1. In the copied folder go click into:
    1. examples → peripheral → blinky freeRTOS → pca10056 → blank → ses
2. The new project will be the blinky ***FreeRTOS*** pca10056 (type: SEGGER Embedded Studio ARM Procjet file).
3. Build and run this project to check that you have a working ***FreeRTOS*** project.
4. Have the new project folder close to the original downloaded folder, preferably in the same folder. This is due to paths and project setup later on.

**Step 3: Reorganize the structure of your project in the file explorer**

Modifications will be made in the copied folder (new project folder) and not in the ”nRF5 SDK 17.1.0 ddde560” folder. Unused files/folder will be deleted.

1. In the ../examples/peripheral folder, move the ”blinky_freertos” folder to the root folder
2. In the root folder, delete all folders except for the folder names ”blinky_freertos”
3. Inside the ”blinky freertos” folder delete the following folders:
    1. pca10040 and pca10100
4. In the ../blinky freertos/pca10056/blank/ses folder, move all files/folders to the root folder.
5. In the ../blinky freertos/pca10056/blank folder, delete the ses folder and move all remaining folders/folders to the root folder.
6. In the ../blinky freertos folder, delete pca10056 folder, hex folder and move the remaining filder/folders to the root folder.
7. In the root folder, delete ”blinky freertos”

**Step 4 (Optional): Reorganize the structure of you project in SEGGER** You are free to choose wether you want to restructure the folder structure inside your project (SEGGER)

Right click on your project and add folders. # For this project two folders were created: src and config.

1. The application folder was deleted.
2. Add existing files into the project by right clicking on your folder.
3. The src folder contains main.c.
4. The config folder contains sdk config.h and ***FreeRTOSConfig***.h.

**Step 5: Include paths and files** Open the new project in Segger. It is likely that the build process will encounter errors since all the original files have been removed. Therefore, it is essential to configure the correct paths for the new project so that it can locate the necessary files.

1. Right click on your project and select options.

2. On the top to the left select ”Common” for private configurations.
3. Include and the right paths in Preprocessor → User Included Directories.

The existing paths are referencing the SDK files that have been deleted. The new project should utilize the SDK files from the original folder that was initially downloaded.

Here is an example of how the paths should appear if the root folder and the original SDK folder are located in the same directory:

*../nRF5_SDK _17.1.0_ddde560/components*

After setting the correct paths, the necessary files must be included from the original nRF5 SDK 17.1.0 ddde560 folder. The specific files that are required depends on the driver you wish to use in the project. Navigate to the original nRF5 SDK 17.1.0 ddde560 folder and simply drag and drop the necessary files into the SEGGER project. This guide is specifically based on the blinky_freertos example and includes the same files that are used in that project.

Compile and execute the code to verify if it functions similarly to the original blinky_freertos example project. If any issues arise, double-check the paths and ensure that the correct files have been included. The sdk_config used in this project are similar to the one used ins the blinky_freertos example.

# Known bugs

14th April 2023 at 2:08pm

bugs faq good-to-know

These are some known bugs. The robots might suddenly decide to behave unexpectedly, in this case flash the same code again and try again.

- C++ server
  - The server only sends a new target if the robot has moved. “If the robot is stuck against a wall and consequently does not move, this bug causes server to never send new targets.” ( *robot_outbox_thread()* in *robot_outbox_handler.cpp*)
  - When using manual input, the server will send the input to the robot roughly once every second. It is not aware of whether you are currently writing, and as a result it might send a target (0, -1) as you are trying to write (0,-100). The next time the correct target will be sent.
- Java server
  - Robots connecting to the server may occasionally disconnect. The reason for this is unknown.
- Hardware
  - NRF3: Only one of the battery packs works.
  - NRF3: The robot cannot start its program by pushing the power button. It needs a kickstart with USB on the short side of the NRF52840 and can maintain power by pushing the power button.
  - The robot will reset its program if a problem occurs during runtime. An example would be a SEGFAULT (running out of memory, out of bounds in an arrary or any similar errors).

# Useful tips

15th May 2023 at 2:06pm

faq good-to-know

## Debugging

Segger has a very useful debugging tool. Nordic extension on VS code is another way to debug the robot through USB.

If an error happens during runtime that causes the robot code to crash, it will not notify you in any way. If a motor input is set before it crashed, this input will continue, and the robot will keep moving as it did before it crashed.

The IR tower is supposed to start rotating shortly after turning the robot on. If this does not happen, it is likely due to an issue with the gateway (Raspberry Pi). Make sure that the dongle is properly attached and try to reset it by removing the power source.

Other general tips include:

- Test the different hardware components if you get unexpected behavior repeatedly. They have a tendency to not work as expected.
- Turn everything off and on again, including the Raspberry Pi used.
- Try the code on a different robot to determine if it is a hardware or software error.

## Software used

There are a lot of useful software used. Some have their own quirks that is useful to be aware of.

- Segger
  - The file system is weird. How the files are organized in the file explorer does not necessarily match the one inside the emProject file. If you add a file to the correct directory, it will not automatically appear in SEGGER. You will have to manually add them there as well.
- ***OptiTrack*** Motive
  - The axes used by Motive is non intuitive, and might lead to confusion when exporting the data. The floor is the x-z-plane, while the y-axis describes the height.

## Corrupted NRF52840

If the NRF52840 becomes corrupted, you would need to bootload the device and pass it a new firmware file. If the NRF52840 does not show up in device manager as JLINK, this may very well be the case.

Useful posts about this issue:

https://devzone.nordicsemi.com/f/nordic-q-a/73999/nrf52840-unseen-under-device-manager-how-to-restore https://devzone.nordicsemi.com/f/nordic-q-a/41192/where-to-download-j-link-ob-firmware-j-link-ob-sam3u128-v2-nordicsemi-170213-bin

# Previous work

29th May 2023 at 3:37pm

good-to-know

In this section you can find a summary of some of the previous thesis on the project. Note that this list is not complete.

## Master theses

- Berg 2023
- Kolbeinsen 2023
- Ruud-Olsen 2023
  - Getting the robot home without server input
  - Some basic obstacle avoidance
- Tran 2023
  - Hardware development
  - Hardware testing
  - Software development
  - nRF52840
- Andersen 2022
  - MQTT and MQTTSN
  - EKF SLAM (not implemented on the robots)
- Mullins 2020
  - Online SLAM

## Project theses

- Tran 2022
  - Printed circuit board design
  - Sensor Data Acquisition Board
  - nRF52840
- Frestad 2022
  - Software structure
- Jølsgård 2020
  - nRF custom peripheral shield
- Korsnes 2018
  - Hardware and software development
  - Low level embedded

# Finishing the project

14th April 2023 at 1:55pm

untagged

If your work is finished and works as intended, merge it into the main branch of the repo so it becomes available to future students. If there are several students working the same repo, this might require some work.

If your work did not result in a finished feature, or the merging was too much work, leave your project as a branch on the repo. Name the branch something that clearly describes the feature it includes. Add a readme in the branch that describes it further, and add your name so future students can refer to your report for further information. Please remove any unnecessary comments and add explanations where needed. Delete all the other branches you have worked on.

Please add a brief summary of the work done to **Previous work**.

# How to create new nrf software project method 1

29th May 2023 at 3:27pm

how-to nrf52840 software

1. Start Embedded Studio.
2. (optional) install CPU support Package for your device family via Tools → Package Manager. Nordic Semiconductor nRF CPU Support Package were used here.
3. Create new project via File → New Project → Create the project in a new solution.
    1. Set a project name. If no CPU package is used then select "A C/C++ executable for a Cortex-M processor". Press Next, select your target device and keep pressing Next until your project is finished.
    2. If you are using a CPU support package select the "A C/C++ executable for..." from the corresponding package and finish your project as described in the point above.
4. Connect the nRF52840DK to a PC. Ensure that both the Jlink on the development board and the nRF52840 are powered, not just the nRF52840.
5. Give your newly created project a try by building it with F7 and executing it with F5.
6. Create a new folder /lib/***FreeRTOS***/***FreeRTOS***-Kernel in both the project explorer in Embedded Studio and on your hard drive in the project folder.
7. Download and unpack the ***FreeRTOS*** software to any location.
8. In the unpacked folder open folder /***FreeRTOS*** and copy the folder /***FreeRTOS***/Source to the /lib/***FreeRTOS***/***FreeRTOS***-Kernel in your ES project folder.
9. Add the same files to the Embedded Studio project explorer. The easiest way is to drag and drop the folder onto the /lib/***FreeRTOS***/***FreeRTOS***-Kernel folder.
10. Right click the new folder and select Setup. Check the box "Recurse into Subdirectories" and press OK.
11. This should add all ***FreeRTOS*** sources to your setup. However not all files are needed so the wrong files have to be removed again. To do this first convert the folder to a regular folder by right clicking it and select "Convert to regular folder".
12. You will need all .c source files from the /Source folder. All include files from /Source/include and the folders /Source/portable/GCC and /Source/portable/***MemManage***.
13. All other folders and files can be safely removed by simply selecting them and pressing the DEL key our right click and delete.
14. Next make sure that in /Source/portable/***MemManage*** you only have one .c file selected e.g. heap 1.c. Remove all other .c files, otherwise the project will later not build.
15. In /Source/portable/GCC make sure that only the folder is included that is the architecture of your target platform. In our example it is a Cortex-M4 target device so only folder /Source/portable/GCC/ARM CM4F stays. All other folders can be safely removed as before.
16. Next you will need to create a ***FreeRTOSConfig***.h file which will configure your ***FreeRTOS*** setup. For references see the ***FreeRTOS*** documentation or use one of the config headers from the many samples out there as reference. For example the one from the example project above. We recommend to place this file into your source folder where your main.c file is. In this case it is folder /source in the project folder.
17. Next all include paths need to be set. You can add this in project options under Project → Options → Preprocessor → User Include Directories. If you are using the same folder structure as recommended the following three include paths must be set. If you are using another project structure adjust the paths accordingly.
    1. $(***ProjectDir***)/source

   2. $(***ProjectDir***)/lib/***FreeRTOS***/***FreeRTOS***-Kernel/Source/include
   3. $(***ProjectDir***)/lib/***FreeRTOS***/***FreeRTOS***-Kernel/Source/portable/GCC/ARM CM4F

18. Now edit your main.c to include ***FreeRTOS***.h and task.h and add your ***FreeRTOS*** application code to the main.c.
19. (optional) add your third party libraries, HALs, drivers etc. to your project by including their paths. In this project the nRF SDK folder is outside of this project folder. The paths for the nRf SDK therefore start with "../". Your paths should look something like this:
    1. $(***ProjectDir***)/source
    2. $(***ProjectDir***)/lib/***FreeRTOS***/***FreeRTOS***-Kernel/Source/include
    3. $(***ProjectDir***)/lib/***FreeRTOS***/***FreeRTOS***-Kernel/Source/portable/GCC/ARM CM4F
    4. ../nRF5 SDK 17.1.0 ddde560/modules/nrfx
    5. ../nRF5 SDK 17.1.0 ddde560/components/boards
    6. ../nRF5 SDK 17.1.0 ddde560/components/libraries/util
    7. ../nRF5 SDK 17.1.0 ddde560/modules/nrfx/hal
    8. ../nRF5 SDK 17.1.0 ddde560/modules/nrfx/drivers
    9. ../nRF5 SDK 17.1.0 ddde560/integration/nrfx/legacy
    10. ../nRF5 SDK 17.1.0 ddde560/modules/nrfx/templates
    11. ../nRF5 SDK 17.1.0 ddde560/modules/nrfx/templates/nRF52840

Once all this is done your application should build now and you should be able to debug a ***FreeRTOS*** application in Embedded Studio.

# How to create new nrf software project method 2

29th May 2023 at 3:29pm

how-to nrf52840 software

**Step 1: Download nRF5 SDK from Nordic Semiconductor**

- https://www.nordicsemi.com/Products/Development-software/nrf5sdk/download
- The version used for this project is 17.1.0

**Step 2: Make a copy of the downloaded folder and rename to desired project name**

1. In the copied folder go click into:
    1. examples → peripheral → blinky freeRTOS → pca10056 → blank → ses
2. The new project will be the blinky ***FreeRTOS*** pca10056 (type: SEGGER Embedded Studio ARM Procjet file).
3. Build and run this project to check that you have a working ***FreeRTOS*** project.
4. Have the new project folder close to the original downloaded folder, preferably in the same folder. This is due to paths and project setup later on.

**Step 3: Reorganize the structure of your project in the file explorer**

Modifications will be made in the copied folder (new project folder) and not in the "nRF5 SDK 17.1.0 ddde560" folder. Unused files/folder will be deleted.

1. In the ../examples/peripheral folder, move the "blinky_freertos" folder to the root folder
2. In the root folder, delete all folders except for the folder names "blinky_freertos"
3. Inside the "blinky freertos" folder delete the following folders:
    1. pca10040 and pca10100
4. In the ../blinky freertos/pca10056/blank/ses folder, move all files/folders to the root folder.
5. In the ../blinky freertos/pca10056/blank folder, delete the ses folder and move all remaining folders/folders to the root folder.
6. In the ../blinky freertos folder, delete pca10056 folder, hex folder and move the remaining filder/folders to the root folder.
7. In the root folder, delete "blinky freertos"

**Step 4 (Optional): Reorganize the structure of you project in SEGGER** You are free to choose wether you want to restructure the folder structure inside your project (SEGGER)

Right click on your project and add folders. # For this project two folders were created: src and config.

1. The application folder was deleted.
2. Add existing files into the project by right clicking on your folder.
3. The src folder contains main.c.
4. The config folder contains sdk config.h and ***FreeRTOSConfig***.h.

**Step 5: Include paths and files** Open the new project in Segger. It is likely that the build process will encounter errors since all the original files have been removed. Therefore, it is essential to configure the correct paths for the new project so that it can locate the necessary files.

1. Right click on your project and select options.

2. On the top to the left select ”Common” for private configurations.
3. Include and the right paths in Preprocessor → User Included Directories.

The existing paths are referencing the SDK files that have been deleted. The new project should utilize the SDK files from the original folder that was initially downloaded.

Here is an example of how the paths should appear if the root folder and the original SDK folder are located in the same directory:

*../nRF5_SDK _17.1.0_ddde560/components*

After setting the correct paths, the necessary files must be included from the original nRF5 SDK 17.1.0 ddde560 folder. The specific files that are required depends on the driver you wish to use in the project. Navigate to the original nRF5 SDK 17.1.0 ddde560 folder and simply drag and drop the necessary files into the SEGGER project. This guide is specifically based on the blinky_freertos example and includes the same files that are used in that project.

Compile and execute the code to verify if it functions similarly to the original blinky_freertos example project. If any issues arise, double-check the paths and ensure that the correct files have been included. The sdk_config used in this project are similar to the one used ins the blinky_freertos example.