NTNU
Norwegian University of
Science and Technology

Implementation of Mapping and Navigation on an Autonomous Robot

Geir Henning Eikeland

Master of Science in Cybernetics and Robotics
Submission date: March 2018
Supervisor: Tor Engebret Onshus, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem description

We have a robot based on the Lego Mindstorms NXT development kit which is running an implementation of the real-time operating system FreeRTOS. It is part of a system used for exploring and generating a map of an unknown environment, together with two other robots. Sensor and position data is sent to a server, which then returns new navigation commands. This thesis will attempt to make the robot operate more autonomously, and handle more of the mapping and navigation tasks on its own. This will be achieved through the following steps.

- Resolve any hardware or software issues present at the start of the project and implement the improved positioning and motor control algorithms from the application in the Arduino robot.

- Implement a universal application interface to make it easier to transfer new functionality to other robots in the system.

- Implement mapping functionality locally in the NXT robot so that it can generate a feature-based representation of the environment and send it to the server.

- Implement navigation functionality locally in the NXT robot so that it can perform some simple navigation autonomously.

- Make the necessary changes to the server application to support the new functionality in the robot.

New functionality will be tested using the simulator software integrated into the server application before being implemented on the physical robot. Any other changes made to the robot application will also be documented.

Preface

I would like to thank my supervisor Professor Tor Onshus for his guidance and support during the last year. Håvard Skåra Mellbye of the Ascend NTNU team 2018 deserves to be mentioned, for taking the time to demonstrate the setup procedure and usage of the OptiTrack system.

I am very grateful for all the assistance I received from the people at the Technical Workshop at ITK, with hardware modifications and helpful advice along the way. Thanks also to the guys at the office for making this a great final semester.

Finally, I would like to thank all the wonderful people I have met during my time in the Student Society in Trondheim for some fantastic years, and for making my life as a student better than I could have possibly hoped for.

[illegible]

Geir Henning Eikeland
Trondheim, 22.03.2018

Summary and conclusion

In this thesis functionality for mapping has been implemented in the NXT robot, so that it can perform this task independently of the server application. The structure of the robot application has been modified to accommodate the new features, and some improvements have been made to parts of the code that were not performing optimally in the previous version of the application. A proposed implementation of a module for navigation functionality in the robot was described.

NXT robot Initial testing of the robot showed systematic errors in the ability to drive in a straight line towards a target. Only the data from the wheel encoders and the gyro was used for pose estimation. The sensors were also poorly calibrated. These issues were kept in mind during development but have not affected the work on the problems that have been solved in this thesis. It should be a priority for coming projects to improve this functionality.

A new solution for power supply to the IO circuit board has been developed. It now draws current directly from the NXT battery pack, instead of being supplied via the connection from the NXT brick to the IO circuit board. This solution has solved the issue of unstable power supply to the IR sensors.

The top of the sensor tower was mounted securely, and there is no longer any risk of it falling off during operation. The foundation of the tower construction, however, is quite unstable. This is a drawback of using Lego for the physical construction and will be difficult to improve without adding more solid material to the construction. It is worth considering if the robot should be developed any further in its current physical state.

Robot application The amount of global data has been reduced and replaced with queues. A separate task for handling the line extraction procedure was added. Parameters for the pose controller have been optimized some, but more precise calculations need to be made. However, a noticeable improvement from the original state can be observed. The performance of the robot is seen to be just as good as before the changes made

in this thesis. Macros have been added for switching between the new and original functionality, which ensures backward compatibility with the mapping functionality in the server application.

Server application The format for sending orders to the robot has been changed to Cartesian coordinates. Changes have been made in both the robot and the server to accommodate this. Polymorphism was introduced in the simulator for it to support robots with different configurations. The new mapping functionality was tested in the simulator before being implemented in the robot, and it was shown to function as intended. Functionality for handling the new message type containing line updates was implemented. The modules that were interacting with the robot, MappingController, and NavigationController, were modified not to affect the corresponding functionality that was added locally in the robot. This was done by just excluding the robot from the process, based on its name.

Mapping An algorithm for extraction of line segments was adapted for use in a situation with short-range IR sensors, as in the NXT robot. The mapping task that was implemented required several buffers for its procedure, and could potentially require much computational power for the merging of line segments. However, testing showed that the mapping functionality does not require as many resources as anticipated. It turned out to have no observable impact on the system performance, meaning that the NXT is capable of supporting an application containing significantly more of the functionality that is currently handled by the server application. The performance of the line segment generation has been tested without navigation functionality enabled, but this was not required for finding the optimal parameters for the procedure. After the right values for the parameters were found, the algorithm proved to work as intended.

Navigation A navigation algorithm based on the principle of wall-following was described, and a possible implementation for a robot with the sensor configuration of the NXT was shown. This thesis did not make any progress on the implementation of a module in the robot application, but a few basic simulations have been run to see if the principle is viable for this system. The prerequisites that are necessary for realizing the solution have been established, and also the resource usage required by this task in the application. Based on the experience with the mapping task, the hardware should be more than capable of running an extra task. Finally, some the parameters of the algorithm have been described, and values for them have been suggested.

Oppsummering og konklusjon

I denne oppgaven har det blitt implementert funksjonalitet for kartlegging i NXT-roboten, slik at den kan utføre denne oppgaven uavhengig av serverapplikasjonen. Robotapplikasjonens struktur har blitt tilpasset den nye funksjonaliteten, og enkelte forbedringer har blitt gjort i deler av koden som ikke fungerte optimalt i tidligere versjoner av applikasjonen. En foreslått implementasjon av en modul for navigasjonsfunksjonalitet ble beskrevet.

NXT-robot Innledende testing av roboten viste systematiske feil i evnen til å kjøre i en rett linje mot et mål. Kun encoder-data fra hjulene og darta fra gyroen ble benyttet i positur-estimeringen. Sensorene var i tillegg dårlig kalibrert. Disse problemene ble holdt i bakhodet under utviklingsprosessen, men de har ikke påvirket arbeidet med hovedproblemstillingene i denne oppgaven. Det burde prioriteres å forbedre denne funksjonaliteten i kommende prosjekter.

En ny løsning for strømforsyning til IO-kretskortet har blitt utviklet. Det trekker nå strøm direkte fra NXT-batteripakken, i stedet for å bli forsynt gjennom tilkoblingen til NXT-enheten. Denne løsningen har løst problemet med ustabil forsyning av strøm til IR-sensorene.

Toppen av sensortårnet ble festet ordentlig, og det er ikke lenger noen fare for at det kan falle av under kjøring. Fundamentet til tårnkonstruksjonen er derimot nokså ustabilt. Dette er en ulempe ved å bruke Lego i den fysiske konstruksjonen, og det vil være vanskelig å forbedre uten å legge til et mer solid materiale. Det er verdt å vurdere om roboten skal utvikles videre med dens nåværende oppbygning.

Robotapplikasjon Mengden globale data har blitt redusert og erstattet med køer. En separat task for å håndtere ekstrahering av linjer ble lagt til. Parametere for positur-regulatoren har blitt noe optimalisert, men mer presise beregninger trengs her. Det er allikevel mulig å se en forbedring fra den opprinnelige tilstanden. Robotens ytelse er sett å være minst like god som før endringene som ble gjort i denne oppgaven. Makroer

for å bytte mellom ny og opprinnelig funksjonalitet har blitt lagt til, noe som sikrer bakoverkompatibilitet med kartleggingsfunksjonaliteten i serverapplikasjonen.

Serverapplikasjon Format for sending av ordre til roboten har blitt endret til kartesiske koordinater. Det har blitt gjort endringer i både roboten og serveren for å støtte dette. Polymorfisme ble introdusert i simulatoren slik at den støtter roboter med ulik konfigurasjon. Den nye kartleggingsfunksjonaliteten ble testet i simulatoren før den ble implementert i roboten, og den viste seg å fungere etter hensikten. Modulene som kommuniserte med roboten, MappingController og NavigationController, ble endret til å ikke påvirke den tilsvarende funksjonaliteten som ble lagt til lokalt i roboten. Dette ble gjort ved å ekskludere roboten fra prosessen, basert på navn.

Kartlegging En algoritme for ekstrahering av linjestykker ble tilpasset for bruk i et tilfelle med IR-sensorer med kort rekkevidde, slik som er tilfelle i NXT-roboten. Kartleggingstasken som ble implementert krever flere buffere for å gjennomføre prosedyren sin, og kunne potensielt kreve mye regnekraft for å slå sammen linjestykker. Testing viste derimot at kartleggingsfunksjonaliteten ikke ikke krever like mye ressurser som forventet. Tasken viste seg å ikke ha noen observerbar innvirkning på systemets ytelse, som betyr at NXT-roboten er i stand til å støtte en applikasjon som inneholder betydelig mer av funksjonaliteten som for øyeblikket er håndtert av serverapplikasjonen. Ytelsen til linjestykke-genereringen har blitt testet uten navigasjonsfunksjonalitet aktivert, men dette var ikke nødvendig for å finne de optimale parameterne for prosedyren. Etter at de rette verdiene for parameterene var funnet, viste algoritmen seg å fungere som tiltenkt.

Navigasjon En navigasjonsalgoritme basert på prinsippet om vegg-følging ble beskrevet, og en mulig implementasjon for en robot med en sensorkonfigurasjon lik den for NXTen ble vist. Denne oppgaven gjorde ingen fremgang med å implementere en modul for dette i robotapplikasjonen, men noen få grunnleggende simuleringer har blitt kjørt for å se om prinsippet kan anvendes i dette systemet. Forutsetningene som er nødvendige for å realisere løsningen har blitt etablert, sammen med ressursene som trengs i applikasjonen. Basert på tidligere erfaring med kartleggings-tasken, skulle det være mulig å kjøre en ekstra task på den hardwaren som finnes. Til slutt ble noen av parameterene for algoritmen beskrevet, og verdier for dem har blitt foreslått.

Contents

List of Figures

List of Tables

List of Listings

Abbreviations

IDE	Integrated Development Environment
SLAM	Simultaneous Localisation and Mapping
RTOS	Real-Time Operating System
API	Application Programming Interface
SSNAR	System of Self-Navigating Robots
GUI	Graphical User Interface
IMU	Inertial Measurement Unit
LIDAR	Light Detection and Ranging
PI	Proportional-Integral
ARQ	Automatic Repeat Request
MCU	Micro-Controller Unit
IR	Infrared
PB	Point Buffer
LB	Line Buffer
LR	Line Repository
ITK	The Department of Engineering Cybernetics
NTNU	Norwegian University of Science and Technology

Introduction 1

1.1 Motivation and background

The overall aim of this project is to further improve the system of self-navigating robots, referred to as SSNAR, that has been developed in several master's and project theses, starting in 2004 at ITK. It has been improved significantly over the years, originally running a just a single robot using a MATLAB toolbox. Today the total system consists of a server application written in Java which controls four different types of robots that operate out in the field. The server receives measurement and position data from the robots, which it uses to generate a map of the environment and navigate the robots to unexplored areas.

Since the server application currently handles all navigation and mapping, the robots all depend on communication with the central control program. To make the robots operate more autonomously, they would need to handle more of these tasks themselves. It could be interesting to investigate if the robots have the processing power required for this. Eikeland (2016) suggests that this solution could be viable, and this serves as the starting point for this thesis. The ultimate goal is to make the robots navigate independently of the server, and collaborate on generating a map directly from the information they exchange with each other. The server would then be used only as a GUI for displaying the map graphic.

1.2 Starting point

The work in this thesis starts where the master's students in the spring of 2017 left off. The development was intended to be done using the AVR robot as a base, but for reasons explained in Subsection 1.4.1 the NXT robot has been used instead. This has made little difference since the application is pretty much the same for all robots.

The application running on the NXT robot is the same one as for the other robots and was initially developed for the AVR by Ese (2016). Since then it has received new functionality from the work of two master's theses during the spring of 2017. Lien (2017) implemented a new communications protocol, and the improvements made to the pose estimation functionality by Amsen (2017) have been imported from the original Arduino robot. In addition to this, the principles from Eikeland (2016) have been continued, but no implementation of this had been made from before.

The system was functioning as described in those reports when the development of the thesis began. The robot connected successfully to the server application and performed a short mapping session on the office floor. The map was generated as expected, and the robot followed way-points correctly and stopped when the environment was fully explored. Still, there were several issues present that did not impact the overall performance of the mapping procedure but would still need to be improved. More on this can be found in Chapter 3.

1.3 Development

All work on this thesis has been done at the offices of ITK at NTNU. The thesis is written as part of a larger project, with several other students writing their thesis on similar problems. However, each student has had their own problem description to solve, and have written their theses independently of each other.

Regular meetings with supervisor Professor Tor Onshus were held, who has provided help and guidance along the way. I have often cooperated with the other students working on the project, whom I shared an office with. Students who graduated last year, Kristian Lien and Jørund Amsen, have been contacted by email when I had questions that were related to their work. The employees at the Technical Workshop at ITK, have assisted with modification of physical construction and power supply of the NXT robot.

1.4 Current condition

1.4.1 Robots

There are currently three robots in the system that are more or less operational: The AVR, NXT and the original Arduino robot. In addition to these, three new units are under development: The EV3 and two new robots based on the Arduino platform.

AVR The AVR robot is able to connect to the server application and will seem to work as expected for a while. However, as described by Strande (2017), its program will eventually crash. The previous developer did not find the cause of this, and it is hard to see any pattern in how long it can operate before it malfunctions. Since both its hardware and software had been meticulously debugged already, it is now scrapped and used for parts.

Original Arduino The original Arduino robot is in good condition and works as intended. It contains its original application, with the improvements in position estimation from Amsen (2017) and the communication protocol developed by Lien (2017). There is some trouble with the physical wiring that connects the nRF51 dongle to the Arduino circuit board. This seems to cause the connection to the server to be unstable and less reliable.

NXT The NXT robot is last worked on by Lien (2017) and contains the new communication protocol, but not all the improvements made to the position estimation developed by Amsen (2017). There are some issues with the readings from the IR sensors, seemingly due to insufficient supply of current to the IO circuit board. The sensor tower is not adequately attached to its foundation, and it will sometimes fall off after a few rotations back and forth.

EV3 Helders (2017) started work on an EV3 brick to implement FreeRTOS on it. The port of the operating system is not finished, and more work remains before it is possible to implement the application from the other robots. It currently only consists of the EV3 brick alone with a partial implementation of FreeRTOS, and no physical robot construction was completed. Like the NXT, it needs an external circuit board to connect all the sensors.

New Arduinos Two of the project theses related to the system during the fall of 2017 are focusing on constructing new robots based on the platform of the old Arduino robot, but with some improvements. The physical construction of these is mostly completed, and they will be used in the master's thesis work during the spring of 2018.

1.4.2 Server application

The server application was last worked on by Melbø (2017), who added an algorithm for SLAM using a particle filter. Changes have also been made by Lien (2017) who added functionality for handling a Drone unit. Some modifications to the mapping module were made for this to work. Except for these changes, the application is in the same condition as when it was first developed by Thon (2016) and Andersen and Rødseth (2016).

1.5 Report structure

This report consists of nine chapters and two appendices, as well as the final summary and conclusions which are presented at the beginning.

- Chapter 1 provides an overview that is useful for a reader that is not familiar with the Lego project, and a summary of the motivation behind this work.
- Chapter 2 contains theory and information that the reader should be aware of when reading later chapters.
- Chapter 3 is about the work that concerns the NXT robot type specifically
- Chapter 4 and 5 describe work on the software that is relevant for all the robots. Documentation of the application structure is found here.
- Chapter 6 and 7 describe the process of developing the two new features that should make the robot operate more autonomously
- Chapter 8 contains the result of tests of the work done in Chapter 6 and discussion of this. It also discusses the possible implementation of the task described in Chapter 7, which was not finished.
- Chapter 9 lists some tasks that were not completed or which were discovered during the work on this project, and some suggestions for problems that can be taken on in future projects.

Appendix A lists material that is relevant to this project, which was submitted along with this thesis.

1.6 Equipment

The following equipment has been used during development. It was either supplied by ITK or downloaded from the internet if it was available free of charge.

1.6.1 Software

Atmel Studio 7 Free IDE from Atmel used for programming and debugging the AVR micro-controller mounted on the IO circuit board.

IAR Embedded Workbench IDE 8 IDE used for programming and debugging the ARM7 micro-controller in the NXT. IAR offers a free trial of the full version of the program for 30 days, and this was used in the startup phase of the project. After that time the software is downgraded to a kick-starter version, which provides all the features contained in the full version that are relevant to the development in this project. It does, however, have one major drawback; the compiler enforces a 32 KB size limit for the program code.

MATLAB R2017a Used for plotting the position data from the OptiTrack system. See Figure 3.2 for an example.

Sourcetree A desktop client which provides a convenient GUI for managing version control with git.

NetBeans IDE 8.2 An IDE for Java. Its integrated GUI design tool was used for developing the interface of the server application. The version for Windows 10 was used.

OptiTrack Motive:Tracker Motion capture system which is mainly intended for use in video capture, but it can also be used for showing real-time positioning data of an object in up to six degrees of freedom. ITK has a system with 16 cameras aligned against the center of a room. Reflective bullets are attached to the object that is to be tracked, and the position can be determined if approximately three or more of these are visible to a given number of cameras at any time. Version 1.5.0 (64-bit) was used in this project.

1.6.2 Hardware

Dell OptiPlex 9020 Desktop Computer Personal computer with a fourth-generation Intel Core i7-4790 running at 3.6 GHz, with 16.0 GB of DDR3 RAM. Contains an SSD-disk with 256 GB of local storage.

Atmel-ICE Development tool for debugging and programming of Atmel-based microcontrollers. Used with the AVR-microcontroller on the external IO board of the NXT. Connected with the JTAG interface.

SEGGER J-LINK EDU Debug probe with USB-interface to PC and a 20-pin JTAG interface to the target. Since this is only an educational version, it does not support more advanced debugging features, such as setting breakpoints that watch for a change in data. It also has a pop-up dialog which reappears each day, that requires the user to accept some terms and conditions for educational use. Used with the NXT brick.

nRF51 dongle A USB development dongle for Bluetooth Low Energy from Nordic Semiconductor. The robots and the server application each have one of these. The one connected to the USB port of the PC contains a server version of the dongle application, and the robot ones contain a client-version. Server and client dongles are set up to automatically establish a connection between each other, which makes the server application the host of the communication, and the robots are the peripherals.

1.7 Work by others

Walid Faryabi has continued the work on the EV3-platform. He has worked on adapting a FreeRTOS port for the microcontroller as an official version for this platform does not exist.

Simen Robstad Nilssen continued the work on one of two new Arduino-based robots. He finished constructing the hardware and implemented the FreeRTOS application. This robot is set up with a sensor tower with IR sensors of the same type as the older robots.

Sondre Martin Kvellestad Jensen continued the work on one of two new Arduino-based robots. He finished constructing the hardware and implemented the FreeRTOS application. Unlike Jensen's robot, this one is fitted with a LIDAR-sensor instead of IR.

Bendik Bjørndal Iversen worked on the software that will be used in a device equipped with a stereo camera to be used for feature extraction from the environment. The goal is to use this camera in a future drone that will collaborate with the other robots in the mapping process.

Marius Nordness Blom resolved minor hardware issues present in the oldest Arduino-robot. He also worked on improving the local anti-collision feature.

Johan Korsnes developed a new robot on a different platform than what has previously been used in the Lego project. It is based on the nRF52 development kit from Nordic Semiconductors, which contains a more powerful CPU with floating point calculation capacities that have not been available in previous robots.

Theory 2

2.1 Robotic mapping

This section explains some important keywords which are useful for understanding how a robot used for mapping an environment can be implemented. The amount of theory about this field of study gets advanced very quickly, so this provides only a brief overview. For more details and more thorough explanations, see Wallgrün (2010).

The concept of generating and storing a representation of the environment which the robot is located inside, and how it can obtain and process the information needed for this, is a large scientific field, and there is a vast amount of possible approaches to this. A good solution for gathering and processing measurement data is essential for achieving a good result. In addition to the odometry data from the wheel encoder, the NXT robot has an IMU containing a gyroscope, an accelerometer, and a compass. These all provide their unique form of spatial information, which needs to be processed and used to update the current spatial representation. Any uncertainties and other considerations are also included here, but in the NXT robot, there is no implementation for indicating the probability of correctness for the measurements.

2.1.1 Simultaneous localization and mapping (SLAM)

The primary challenge when attempting to create an accurate map of the environment is the fact that errors in the robot's model of the environment and its estimated position will differ and affect each other. This becomes a chicken-and-egg problem, and finding accurate solutions to it is both sophisticated and very computationally intensive. It is a field of study that goes far beyond the scope of this thesis, but it is an important concept to be aware of. A robot containing as basic a micro-controller as the NXT must use a simplified approach. A possible choice in this situation would be a simple Kalman filter, something which is already used in the pose estimation task, which currently does not include correction of the map. Methods for both topological and grid-based maps exist, and some alternatives are the particle filter, GraphSLAM, FastSLAM and EKF SLAM.

2.1.2 Map representation

The data structure used for storing the spatial data will determine which navigation approach it is possible to use. The need for data storage and how it will scale as the environment is explored is an important factor to consider when choosing the solution. It is also important to consider which external sensors that are available in the system. A system with a robot equipped with only short-range IR sensors like in the NXT robot might need a different solution than for instance one that has a LIDAR available. Human readability may also be a factor. If the generated map is intended to be used directly for navigation by humans, it will have to be easier to represent visually than in a case where the map can be more abstract and used only internally in the robot for navigation. The following is a description of the two main paradigms of map representation.

Metric representation

Also referred to as **grid maps**. 2-dimensional arrays are used to represent the world with discrete, square locations, or cells, at a given resolution. Each cell is represented by some form of coordinates. The simplest form may use only a bitmap where each cell is either obstructed or free, which is suitable for systems with limited memory available. A more advanced approached called occupancy grid assigns a probability value, from guaranteed empty to guaranteed occupied, to each cell to indicate how strong or certain the observed value of that cell is. The original mapping algorithm in this system uses the concept of *frontiers*, which is a term for the cells that lie on the border between the observed cells and the unobserved area.

An alternative to using grid cells is to construct a map from **geometric** objects, for instance, points and lines. This is the approach used in the new mapping algorithm described in Chapter 6. It is most suitable for use with range scans from LIDARs or sonars, but it can also work with one-dimensional IR sensors. It needs less space for storing data since it only needs to store the coordinates of the endpoints of the corners or endpoints of the geometric objects instead of every location that it affects. However, path planning may require more computational power with this approach, since a discrete search space needs to be extracted for most algorithms to work.

Landmark-based, or stochastic maps, also makes use of a coordinate system and focuses specifically on the most significant features of the environments, which can be extracted from sensor data with the highest certainty. However, this also means that they contain a very sparse model of the environment, and are therefore not very suitable for the robot to use exclusively. One benefit is that humans can easily read them.

Relational representation

This approach generates a model of how features in the environment are related to each other, instead of mapping the actual locations in the real world. It is often called a **topological** map, and is usually represented using *graph structures*.

View graphs contain vertices distributed through the environment to cover all the area that is free space. The spacing and placement of the vertices can indicate height, or provide other information used for navigation. The graph in this representation will contain more vertices and edges than what is necessary for determining all the features of the real world.

Route graphs, however, contain only the minimal set of nodes and connecting vertices for representing the possible combinations one can move around the map. Each node represents a distinctive place, and each edge is a distinctive path between two places. For this to work, the environment needs to have a rectilinear structure, with clear distinctions between what can be considered places.

2.2 Navigation

Robotic navigation is a field that contains several challenges, including the concepts described in Section 2.1. There are both simple and complex approaches to choose from. This section describes both the mechanism that is used in the original robot and server application, as well as few that are less reliant on a map representation.

2.2.1 Original approach

The process that is currently used in the system is based on the following four steps.

Localisation

For the robot to know what it should do to reach a specified location, it first needs to determine its position and heading in the environment, commonly referred to as its *pose*. For this, we require a spatial model of the environment, usually a map. In a grid map, this would be the coordinates of the cell it currently occupies, while in a route graph representation it could be as little specific as a *distinct place*. In the system in this thesis, the initial pose of each robot needs to be specified at initialization time. After this, the pose will be estimated and updated continuously based on the fusion of odometry and external sensor data, and sent to the server application which is responsible for the path planning.

Path planning

Path planning refers to the approach and algorithms used to reach a target location in the map. In this case, we can assume that we have a grid map. The result of the calculations should be a set of way-points or movements which provide the optimal way of reaching the target, given some environmental constraints. There are a plethora of algorithms one can choose from, but for the situation in our system, the **A*** search algorithm is a well-tested and solid choice. It is a popular algorithm for use in path-finding and is especially useful for use with a grid. Section 2.3 in Thon (2016), which the original server application is based on, describes its working principle, so it will not be elaborated any further here.

Collision handling

It is vital that the robot be able to stop and take action in case an object appears in its way. This is collision *detection*. To prevent impact or maneuver around it, we need collision *avoidance*. In the most simple case, the object that needs to be circumvented is static, meaning that no coordination with other entities is needed. The robot can then take the actions that are most efficient for itself. A more challenging problem is handling a conflict between two or more moving objects, which may all run their separate collision avoidance procedures. To solve this problem, all robots need to communicate and agree on which action should be taken by who. In this system, the server handles everything related to collision detection and management, by sending priority commands to the robots in the event of a conflict situation. This means that the robots do not need to worry about coordinating their movements with other robots in the same area.

Exploration and map building

This step is based on the resulting way-points and directions from the path planning step. The task is to execute the commands it has been given efficiently and keep track of which actions have been completed and which that are remaining, while at the same time acquire information about the environment in the form of sensor data or any discovered irregularities that are not detected by sensors or marked in the map. This can happen in this system if, for instance, an object moves into the robot's path while in the blind zone of the IR sensors. The robot will then discard its currently assigned list of way-points, and inform the path planning task that it must run a new calculation.

2.2.2 Other approaches

Wall-following (without mapping) The algorithm that was proposed for implementation in the NXT robot contains a quite primitive wall-following behavior. It is however sufficient for demonstrating how a representation of the environment can be built without using a grid map. The representation will have several flaws, including that it will not be able to detect any island or obstacles that are outside of sensing range when traveling along the wall. It will also not be able to detect when it has reached its starting point; a problem often referred to as *closing the loop*. Since it does not perform any localization calculations or path planning, it can not be said to have contributed to any progress on the issue of making the robot more autonomous with regards to navigation functionality.

Wall-following (with topological mapping) The strategy is based on the paper „Topological Mapping with Sensing-Limited Robots", Huang and Beevers (2005). It uses a robot with similar sensing capabilities as our Lego robot. The fundamental principle is to follow walls until the robot has reached its starting point while at the same time generating a graph from corners and features that are detected along the way.

Roaming Probably the least sophisticated algorithm for exploration is a simple random walk through the environment. The robot can be programmed to execute a certain movement each time it encounters an obstacle. A simple behavior could be to rotate

90° in the clockwise direction and resume translation. If still obstructed, repeat. A more advanced form of this is to choose to rotate or reverse, depending on the angle and distance to other obstacles and to chose the most efficient direction of rotation. It is easy to assume that this approach would be very ineffective and that exploration would take a long time to finish. However, simple tests have shown that it performs rather well in test environments of the size used in this project.

2.3 FreeRTOS

FreeRTOS is a real-time operating system which allows functions to be run concurrently as tasks controlled by a scheduler. It is a lightweight system which has ports available for a wide variety of platforms. Many applications require only the minimal amount of code, which consists of only three files. It has been chosen for many of the previous robot projects due to its low memory footprint and good performance on the system with little processing power. Good portability is also beneficial in our work since our robots are based on different hardware platforms. It provides methods for multi-threading and synchronization mechanisms such as mutexes, semaphores and software timers, as well as several heap implementations for various application designs.

Queues are used extensively in this application for synchronization, message passing or both of them combined. They have replaced some of the existing global variables and data structures since they provide thread-safe concurrent access to data. They are also useful for visualizing the data flow in the system, as it is easy to see where the receive and sends occur. One may also notice that the use of traditional semaphores has been kept to a minimum. Instead, a feature named RTOS Task Notifications has been used. This provides a lightweight alternative to a traditional semaphore, especially in the case of a binary semaphore, which is what is required in most cases. It has the benefit of being up to 45% faster, and it requires less RAM than an ordinary semaphore. However, it can only be used for cases where there is only a single task that is blocking on the notification.

The tasks that are run concurrently should be implemented with functions containing endless loops, and should never return. This means that as a rule all tasks are created before the system scheduler is started, and will generally exist until the program exits.

2.4 Collinearity

Some of the following sections refer to the term collinearity when describing either a set of points or line segments. This is an essential property to be able to understand and calculate when dealing with line detection. A set of three or more points are said to be collinear if they lie on a single straight line. However, when dealing with points represented by Cartesian coordinates in floating point numbers, this will never be *exactly* true.

Equation (2.1) shows how we determine if three points (x_1, y_1), (x_2, y_2), (x_3, y_3) are collinear in this application. The parameter ϵ determines how much much deviance from the line and the rounding error we want to tolerate.

$$|(y_1 - y_2)(x_1 - x_3) - (y_1 - y_3)(x_1 - x_2)| \leq \epsilon \tag{2.1}$$

2.5 Line merging

When trying to build a map based on line segments, it is beneficial that we can obtain a representation using as few line segments as possible, to reduce the memory usage. The following is a description of the algorithm used for combining two line segments. It follows the approach used in Hussien and Sridhar (1993).

The two line segments L_1 and L_2 are determined by the start and end points $(\mathbf{p_1}, \mathbf{q_1})$ and $(\mathbf{p_2}, \mathbf{q_2})$ respectively, where $\mathbf{p} = (x, y)$. First we find the parameters for the line equations corresponding to each segment given in the slope-intercept form $y = ax + b$.

$$a_1 = \frac{y_{q_1} - y_{p_1}}{x_{q_1} - x_{p_1}} \tag{2.2}$$

$$a_2 = \frac{y_{q_2} - y_{p_2}}{x_{q_2} - x_{p_2}} \tag{2.3}$$

$$b_1 = y_{p_1} - a_1 x_{p_1} \tag{2.4}$$

$$b_2 = y_{p_2} - a_2 x_{p_2} \tag{2.5}$$

The length of the lines are found by simple trigonometry:

$$l_1 = \sqrt{(x_{q_1} - x_{p_1})^2 + (y_{q_1} - y_{p_1})^2} \tag{2.6}$$

$$l_2 = \sqrt{(x_{q_2} - x_{p_2})^2 + (y_{q_2} - y_{p_2})^2} \tag{2.7}$$

The parameters for the resulting merged line are then calculated.

$$a = \frac{l_1 a_1 + l_2 a_2}{l_1 + l_2} \tag{2.8}$$

$$b = \frac{l_1 b_1 + l_2 b_2}{l_1 + l_2} \tag{2.9}$$

To find the endpoints for the line segment resulting from the merge, the endpoints of the initial line segments L_1 and L_2 are projected onto the new line. The mapping of an arbitrary point (x, y) onto a point (x', y') on the line is given as

$$x' = \frac{x + ay - ab}{1 + a^2} \tag{2.10}$$ $$y' = \frac{ax + a^2 y + b}{1 + a^2} \tag{2.11}$$

The points that lie farthest away from each other on the line are then selected as the new endpoints, and we have successfully obtained the merged line segment. Figure 2.1 shows how existing endpoints are mapped onto the new line with parameters calculated in Equations (2.8) and (2.9). In this example, it can be seen that the projected points $\mathbf{p}_1'$ and $\mathbf{p}_2'$ are the furthest away from each other and constitute the longest segment.

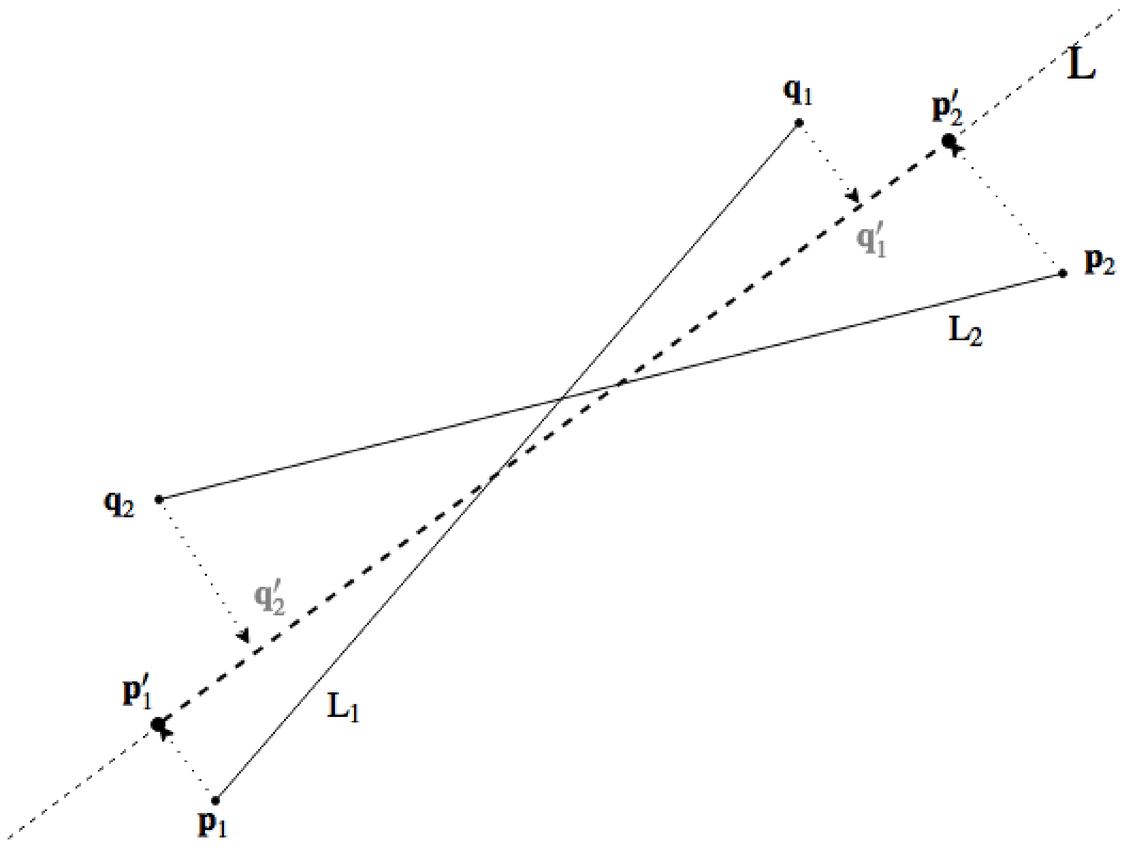


Fig. 2.1.: Description of how new endpoints are selected by projecting existing ones onto the new line.

NXT Robot 3

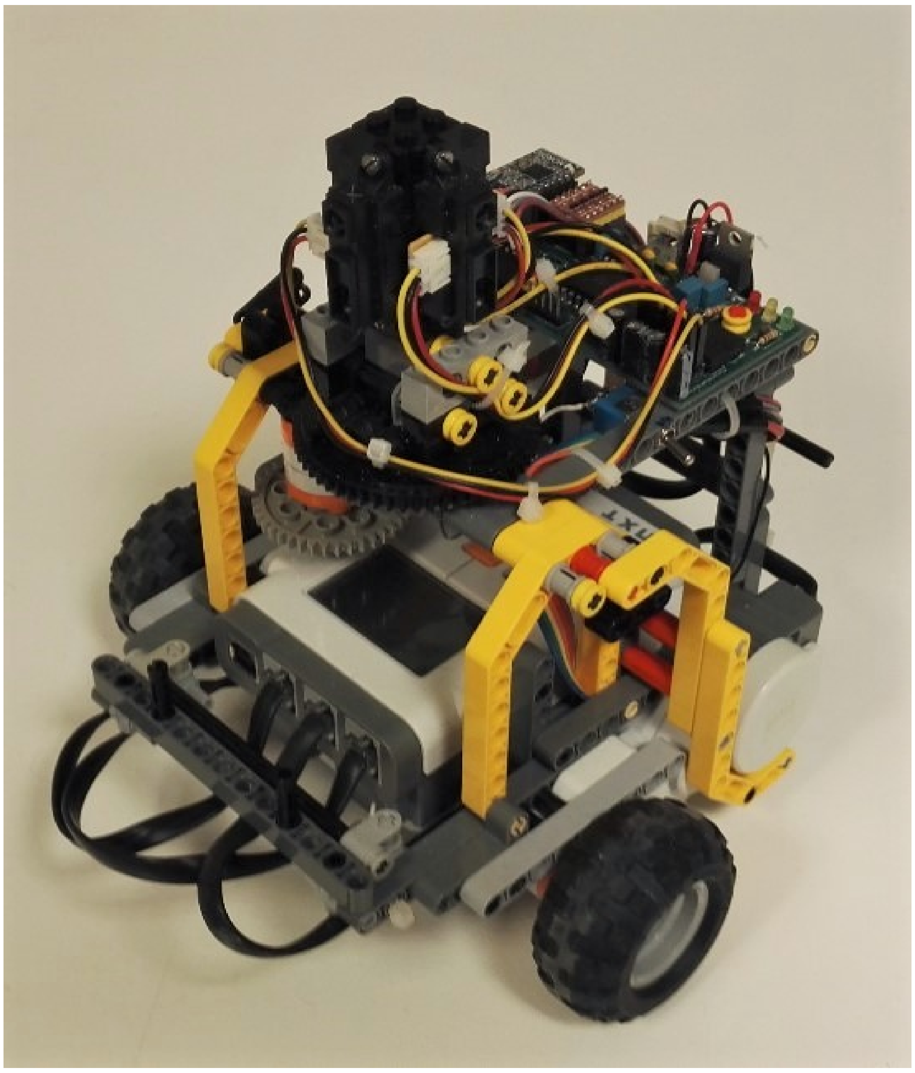


Fig. 3.1.: The NXT robot

Figure 3.1 shows the structure of the NXT robot. This chapter describes and comments on the issues that were present at the start of this project, and how they were resolved.

3.1 Initial testing

The current condition of the pose estimation and control system of the robot was tested before making any changes to the code. The aim of this was to get a rough impression of its performance and to help determine which, if any, irregularities needed to be

Tab. 3.1.: Data from the counter-clockwise square test with the OptiTrack system

	Target		Measurement		Error		Error distance
n	x	y	x	y	x	y	d
1	1000	0	998	-15	-2	-15	15.1
2	1000	1000	1043	973	43	-27	50.8
3	0	1000	56	1039	56	39	68.2
4	0	0	-45	54	-45	54	70.3

Tab. 3.2.: Data from the clockwise square test with the OptiTrack system

	Target		Measurement		Error		Error distance
n	x	y	x	y	x	y	d
1	0	1000	15	998	15	-12	19.2
2	1000	1000	986	879	-14	-121	121.8
3	1000	0	885	-92	-115	-92	147.3
4	0	0	-90	4	-90	4	90.1

accounted for in the development of a new navigation functionality. The procedures described in Chapter 4 of Amsen (2017) were used. Testing was performed with manual mode enabled for the robot. When enabling this option in the robot info GUI of the server, no orders are received from the server and collision management is disabled for both encountered objects and other robots in the environment.

The code for the estimator task contains several constants related to variance and calibration of the IMU and the compass. These values were likely to be the cause of any faults in pose estimation, primarily related to the estimation and control of the rotation and heading. It was unclear how previous students ended up with these values. Inline comments refer to offline calculations that should be available in previous reports, but no sufficiently detailed description of how this was done could be found. Observations of a few simple test runs on the office floor showed deviations in the heading control and estimation that were significant enough to have an impact on a test procedure. This impression is consistent with the conclusion in Amsen (2017) which comments on the noisy sensor readings.

Two tests were run using the OptiTrack system. The robot was given four target commands in sequence so that it moved in a full square. This was done for counter-clockwise (Table 3.1) and clockwise rotation (Table 3.2) - in that order. n describes which step of the command sequence the values belong to. x and y are the position values in Cartesian

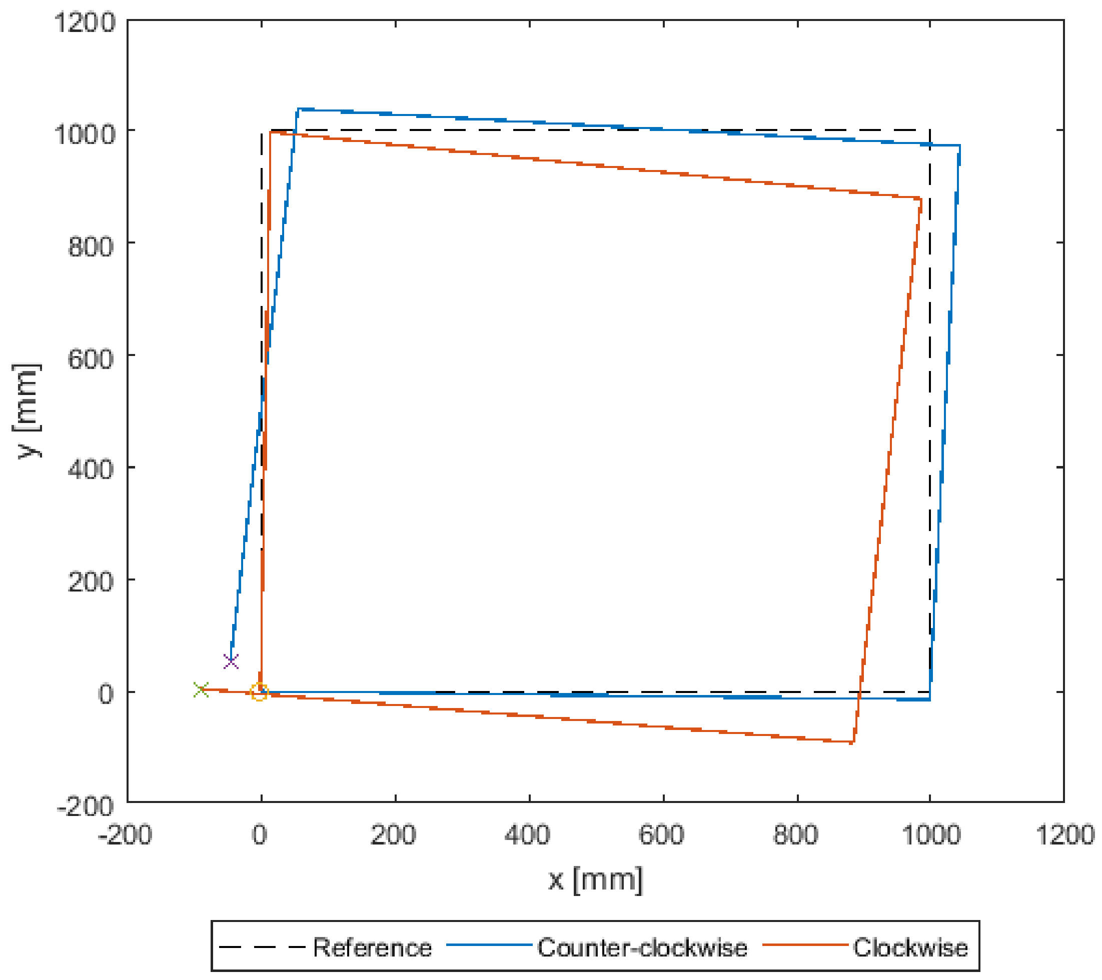


Fig. 3.2.: Trajectory of the robot from the circle-tests with the OptiTrack system

coordinates. The error distance $d = \sqrt{x^2 + y^2}$ is the distance from the position of the target command to where the robot ended up. The system was calibrated to give an accuracy of 0.2 mm for the position measurements. Therefore, the values in the tables are given in whole millimeters. Precision higher than that would be more than we require for our purpose, as the position values used by the server application for generating the map are given in centimeters. Note that this test only describes the state that the robot has ended up in when it has completed its current task, which is its position at each "corner" in Figure 3.2. It does not tell us anything about the robot's behavior on its journey between these.

Figure 3.2 shows a graphical representation of the measurement data from Tables 3.1 and 3.2. When observing the robot while it moved along the square, it seemed to maintain its direction quite precisely while translating forward. However, the rotation movement before heading towards the given target did not seem optimal. It was clear that there was a systematic error in the rotation which caused the robot to stop its rotation too early. This can be seen in every rotation movement for both rotation directions. This points in the direction of poorly calibrated values for the gyro and compass parameters. Also, the second test showed higher deviations from the optimal trajectory. No clear cause of this could be found at first glance, but it is possible that the gyro and compass behave differently depending on the direction of rotation. Also, the tests were not performed right after each other, since the robot was used some before starting the second test. The battery capacity would then be somewhat lower at the start of the second run, something that could have affected the torque available to the wheels when rotating.

3.2 Power supply issues

3.2.1 Initial condition

When testing the NXT for the first time in an open environment without obstacles, the readings from the distance sensors were indicating that obstacles surrounded the robot. The sensors consistently returned too low distance values. One theory was that the cause of this could be that the voltage supplied could to the IO circuit board, which would include the IR sensor operating voltage was too low, or unstable. In Section 6.6, Lien (2017) describes how the distance sensors draw a large amount of current when they are sampling. The irregularities that were experienced was considered to be related to this. Voltage and current values were measured, and the bus voltage did show a tendency to sink slightly from time to time. No clear correlation with the timing of the sensor samplings could be seen, but this was an indication that the issue was related to the stability of the bus voltage.

The voltage supply to the NXT initially consisted of 6 x 1,5 V AA batteries, connected in series. These were replaced with the most recent version of the rechargeable batteries for NXT, specified to deliver 7,4 V and with a capacity of 2200 mAh. This made no impact on the IO circuit board's bus voltage but provided the benefit of not needing to replace several batteries with regular intervals. The NXT regulates its input voltage down to 5 V, and as the IO circuit board operates on a bus voltage of 4,3 V, this should not be problematic. This was also verified with measurements of the voltage received from the input port on the NXT, but the trouble with the sensor readings persisted.

Tab. 3.3.: Characteristics of the NXT battery

Average consumption [mA]	Energy delivered [Wh]	Discharge time [h]	Capacity [mAh]	Mid-discharge voltage [V]	Voltage variation [%]
250	16.22	8.98	2260	7.12	9.2
750	14.91	2.94	2200	6.85	6.9
1500	14.2	1.47	2172	6.47	9.0

3.2.2 New solution

Lien (2017) made an alternative circuit board with a voltage regulator and a connector for a 9 V battery. The board that was mounted of the start of the project was replaced with this one, and a jumper was set so that the IO circuit board drew all its power from the attached battery instead of the NXT. This stabilized the bus voltage on the IO circuit board, and the voltage from the sensor measurement was now stable and working as intended. However, this solution was not very robust and required that the 9 V battery had to be manually connected and disconnected for every restart of the system. The connectors and wires were not safely attached, and would probably not withstand much wear and tear.

A more durable solution was required. Table 3.3 shows the characteristics of the battery obtained by testing done by Hurbain (2017). It shows that the battery is capable of delivering far more current than what the NXT brick limits itself to; hence it was decided to take advantage of this. The voltage from the battery is well within the limits of what the voltage regulator on the IO circuit board can handle, so a connection directly between the battery and the IO circuit board was possible. Small holes have been drilled in the battery housing, and wires have been put through and soldered onto the positive and negative terminals corresponding to a parallel coupling with the battery and the NXT brick. A simple flip-switch was fitted into a Lego piece for easy control of power to the card. This is shown in Figure 3.3.

3.3 Motor control

There is a physical difference from the other robots which has an impact on the precision of the tower measurements. The sensor tower on the NXT is controlled by the same type of servo as its wheels, which is the motor provided by Lego. This part has some

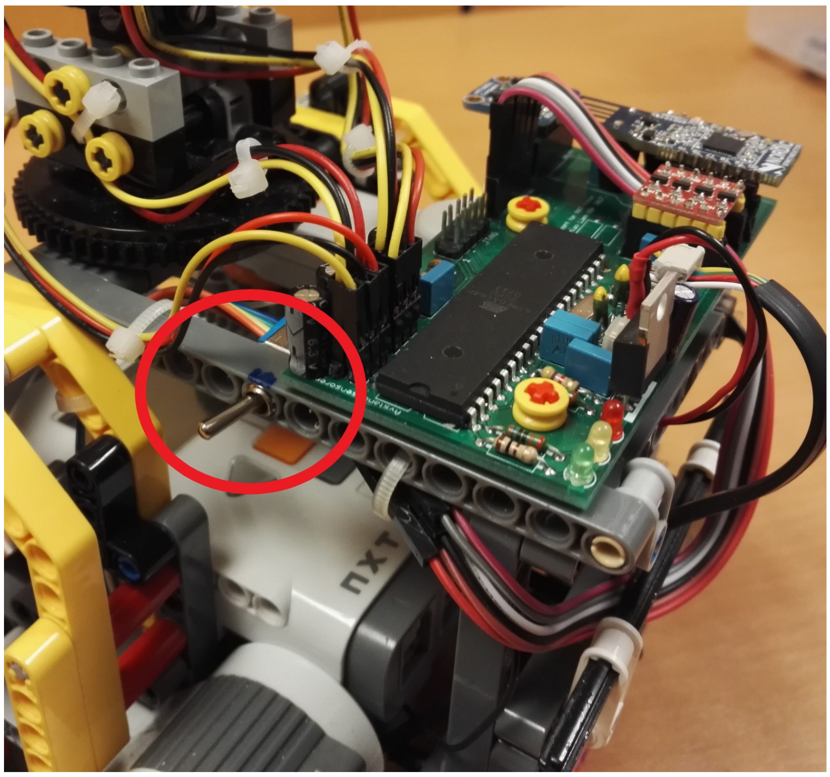


Fig. 3.3.: Placement of the switch, marked by the red circle. The new power and ground wires can be seen next to the voltage regulator, coming up from underneath the circuit board.

internal backlash, which together with the gears used to connect it, gives a possible error between the angle value used in the application and the actual one. The physical structure of the robot is also not very rugged, so it is also possible for the tower to tilt back and forth, causing an error in the vertical angle of the sensors as well. These errors have been estimated to be in the worst case roughly 10° for pitch and 1° for rotation, as illustrated in Figure 3.4.

Another weakness with these motors is that it is not possible to detect which way the tower is facing when the robot is powered on. This means that the sensor tower does not revert itself to the forward-facing position when powering on the device like the servos on the other robots do. This contributes to reducing the precision of the tower angle since it is difficult always to reset the sensor tower to a perfect straight-forward heading, although that is what is reported to the server. It should also be noted that the rotation of the tower will tend to drift a little for each sweep of its rotation interval. Each time it rotates right towards 0°, it will not fully reach the initial position, but instead stop approximately 1° short. However, when rotating towards the left, it will move through its entire interval. As a consequence, the whole interval of rotation will over time shift more and more to the left.

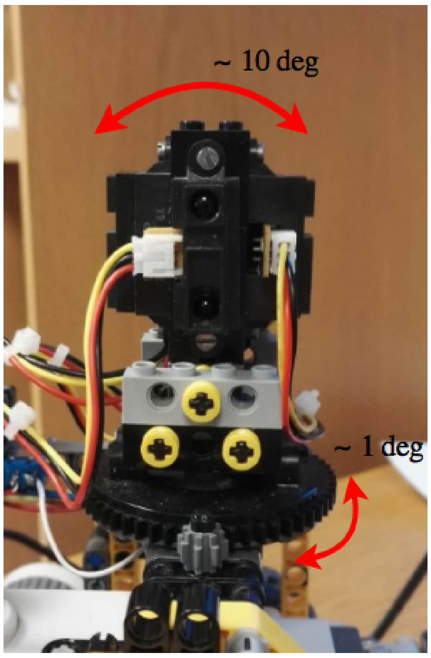


Fig. 3.4.: Possible inaccuracies in pitch and rotation of the NXT sensor tower

3.4 Other

- The variable `SERVO_RESOLUTION` in the sensor tower task has now been changed to a constant. Its value was previously changed by the application depending on the movement status of the robot, but this has not been found to be a good solution for the NXT. The speed of the tower rotation is now constant while the robot is translating, and is frozen while the robot rotates, as before. The speed has been increased some and is now set to 5 degrees per 200 ms. This has proven to be a suitable value and increases the robot's efficiency somewhat since this makes it more likely to catch any obstacles appearing in front of it.

- The upper part of the sensor tower was not fastened well enough, which led to it falling off after a few rotations back and forth. It has now been fastened properly using glue, with the assistance of the people at the Technology Workshop at ITK.

4 Robot Application

This chapter provides an overview of the structure of the application that has been implemented in the NXT robot. While the development has only been done on this robot, the code is written so that it should be possible to transfer it to the other robots in the system quite quickly. It focuses on the parts of the code that has undergone significant changes, and the new features added. It does not go into detail about the inner workings of each task and the philosophy behind their initial designs. This was considered to be outside of the scope of this thesis, and for more background on this, the reader is referred to many of the past theses related to the Lego project, including Ese (2016), Thon (2016) and Amsen (2017).

4.1 System overview

Figure 4.1 describes how the application as a whole is organized with tasks, queues, direct-to-task notifications and input/output from hardware. The graphics are inspired by Figure 6.1 in Ese (2016), which makes it possible to see how the development of the robot has progressed since then.

Tasks

The application consists of the following five main tasks.

Communication Responsible for establishing and maintaining the connection to the server and parsing messages that are received. It updates the global status variables and sends target updates to the pose controller queue.

Sensor tower The task is responsible for controlling the rotation of the sensor tower. Data from the IR sensors is read here, before being sent to the mapping task together with the current rotation of the sensor tower. It also includes a primitive collision handling feature for situations where the server application fails to react to an obstacle in time.

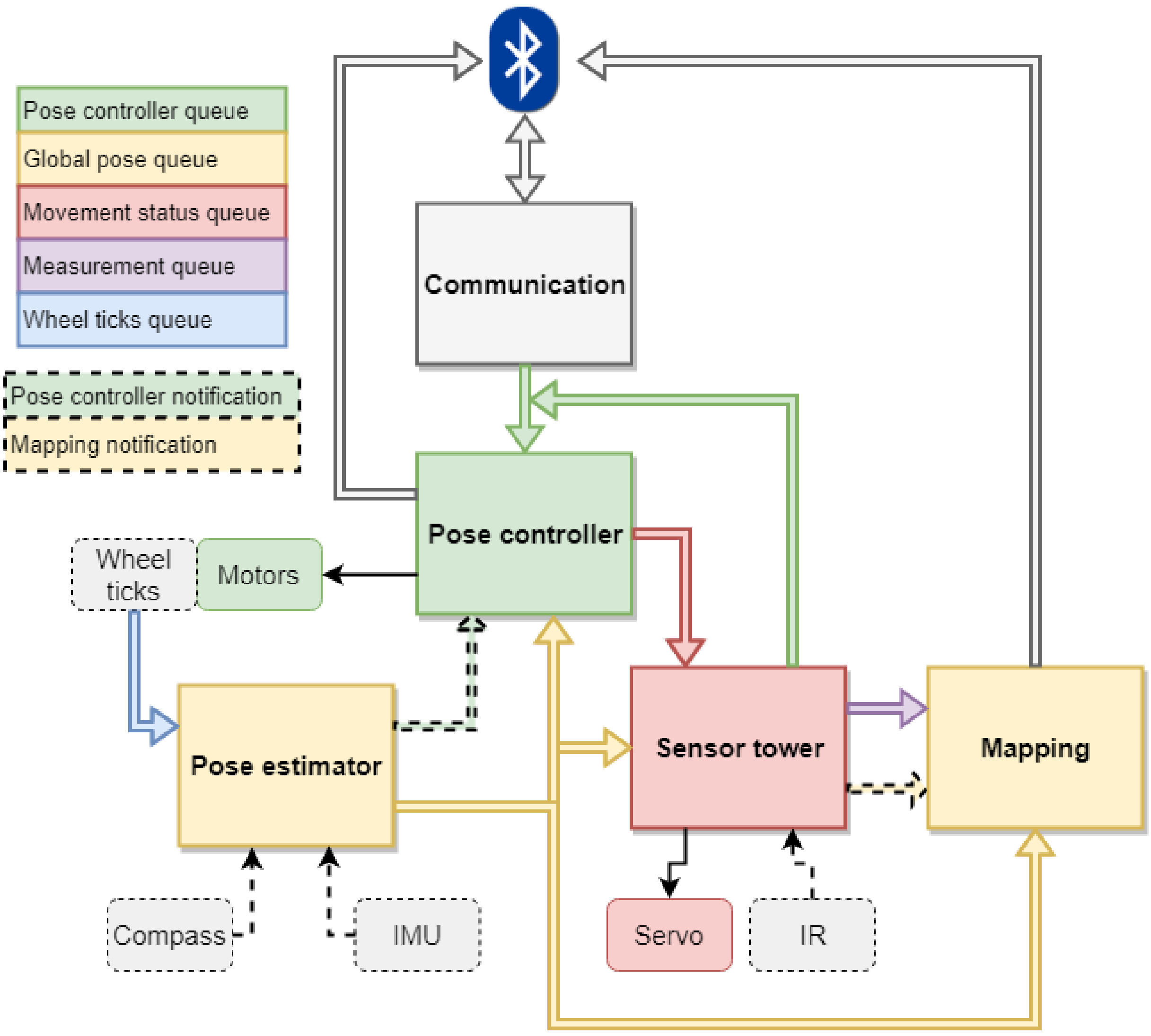


Fig. 4.1.: Structure of the robot application

Pose controller Implements a simple PI-controller for setting the speed of the servos that control the wheels. It receives target setpoints from the pose controller queue. It is synchronized with the pose estimator task since it needs to receive updated pose values to operate. The synchronization is achieved through direct-to-task notification, as indicated by a dashed line in the figure.

Pose estimator Estimates the position and heading of the robot using a Kalman filter-algorithm, which fuses input from the encoder, IMU, and compass. For the work in this project, primarily the encoder tick values have been used.

Mapping This is the new feature implemented in this project. It processes the measurement data received from the sensor tower task and generates line segments that are sent to the server, along with the current data from the pose estimator. This replaces the previous method of sending distance measurements directly and letting the server take care of the processing. The task is described more thoroughly in Chapter 6.

In addition to the main tasks, there exists functionality for calibration of the compass and the IR sensors. Due to the changes made in this project the tasks that are used for this are no longer working. There needs to be made some modifications to adapt this code to the new structure of the application, and to re-calibrate these components.

Queues and synchronization mechanisms

Queues are used for passing data and allow for synchronous and asynchronous communication between tasks. Table 4.1 shows the characteristics of each one that is used in this application.

poseControllerQ The value in this queue is always overwritten when a new target is received in the communication task, and the pose controller will use the value that is currently in the queue as its target. If the queue is emptied by the collision handler in the sensor tower, or if the robot is disconnected or paused, the robot's target will be set to its current position.

movementStatusQ Passes a value to the sensor tower indicating if the robot is translating, rotating or standing still.

Tab. 4.1.: Description of the queues used in the system

Handle	Data type	Content		Length
poseControllerQ	*point_t*	*float* *float*	x y	1
movementQ	-	*uint8_t*	status	1
wheelTicksQ	*wheel_ticks_t*	*int16_t* *int16_t*	left right	1
globalPoseQ	*pose_t*	*float* *float* *float*	theta x y	1
measurementQ	*measurement_t*	*uint8_t* *uint8_t*	data[4] servoStep	3

wheelTicksQ Contains the current encoder tick values of each motor. The queue only has room for one element, which is a struct of type *wheel_ticks_t*. The value is overwritten every 1 ms by sampling the internal AVR helper micro-controller. At the receiving end is the pose estimator task, which calls `xQueuePeek()` to read the value without removing it from the queue. This ensures that the data is only interacted with by the tasks that need it.

globalPoseQ Always contains the currently estimated pose of the robot in a struct of type *pose_t*. It is used in the same way as the poseControllerQ. It is updated by the pose estimator and read from by all other tasks in the system, except for the mapping task.

measurementQ Relays IR sensor data along with the corresponding servoStep value from the sensor tower. It is used by the mapping task in the line extraction procedure, which working principle is described in Section 6.3. It has room for three elements, which lets the mapping task tolerate any possible delays in the sensor tower.

Pose controller notification The pose controller will block while waiting for a notification signal from the pose estimator. The pose estimator sends this signal at the end of each execution period so that the pose controller will not run before an updated pose estimate has been calculated.

Mapping notification The mapping task relies on the sensor tower to send a notification that tells it to start the line extraction algorithm.

Input and output

The grey, rounded squares with dashed outlines in Figure 4.1 indicate the sensor data used by the application and shows which task that receives it. The distance values from the IR sensors are read by the sensor tower task. The data is then passed to the mapping task, as indicated by the measurement queue. The sensor tower task also handles the servo controlling the sensor tower.

The pose estimator task reads data directly from the compass and the IMU and uses `xQueuePeek()` to read the most recent encoder values from the wheel ticks queue. This data is only used internally in the data fusion algorithm. The pose controller sets actuation values for the servos that control the wheels, indicated as "Motors" in the figure.

4.2 Changes made

The most significant change that has been made to the application, except for the introduction of the mapping task, is to separate the source code and related functionality for each of the tasks into the system into multiple files. Before the start of this project, all the code that implemented the tasks in the system was written in the `main.c` file, which contained well over 1000 lines of code. There could be good reasons for structuring the code this way since so many of the tasks were sharing the same global variables and queues. It was however decided that some of that simplicity should be sacrificed to make the code more modular and to make it easier to find the part of the code that performs the tasks that one is looking for while developing. New .c files and corresponding header files have been added for each of the tasks listed in Section 4.1, which are only imported in other files where they are actually needed.

New format for set-points from server

In Section 5.1.5 it is explained how the format for sending way-points from the server to the robot was changed. The changes to the server code led to some changes on the robot side as well. Listing 4.1 shows the part of the communication task that handles incoming orders. The data received is in integer values in centimeters. The robot application operates in floating point values in millimeters, so the target variables are type-casted

Listing 4.1: Part of communication task that puts received orders on the poseControllerQ

```
...
// Coordinates received in cm, convert to mm for internal use in the
    robot.
point_t Target = {
     (float) command_in.message.order.x * 10,
     (float) command_in.message.order.y * 10
};
// Relay new coordinates to position controller
xQueueOverwrite(poseControllerQ, &Target);
...
```

and converted to millimeters, before being written to the poseControllerQ. Details on this can be found in Section 4.1.

Finally, the incoming target is read from the poseControllerQ in the pose controller task, as shown in Listing 4.2. By using `xQueuePeek()` the value in the queue is only read and not removed so that the queue always contains the current target of the robot. If there is no specified target, the controller's set-point will be the current estimated position. The distance calculation remains the same as for the original solution.

Global data

In the original application where most of the program code was located in `main.c`, the data that was shared between the tasks was located there as well. The application has been made more modular by removing several variables from the global scope. The three variables for storing the current pose were protected by a mutex. As explained in the description of the queues in Section 4.1, these variables are now stored in a struct, and globalPoseQ always contains the most recent values. Since queues in FreeRTOS are thread-safe, it can safely be accessed and written to by the tasks that require it, and a mutex is no longer needed. The same is done for the previously global variables which stored the wheel tick-values, which are now replaced by the wheelTicksQ.

The type definitions for the structs containing pose and set-point data have been moved into the new header file `types.h`, along with the definitions for the new data structures developed in this project. This makes it easier to use the same types at several places in the application, by just including the header file where necessary.

Listing 4.2: New sequence for receiving targets in pose controller

```
...
// Read the current target from the queue
if (xQueuePeek(poseControllerQ, &Target, 0) == pdTRUE) {
    xTargt = Target.x;
    yTargt = Target.y;
} else {
    // Queue empty, target is set to current position
    xTargt = xhat;
    yTargt = yhat;
}

// Distance between current position and and target
distance = sqrt((xTargt-xhat)*(xTargt-xhat) + (yTargt-yhat)*(yTargt-yhat
   ));
...
```

Some variables that remain global are the flags indicating whether or not the robot is connected to the server. These are the variables `gHandshook` and `gPaused`, which are used by all tasks for determining. It is not necessary to protect these with mutexes, since they have a size of just 1 byte, and read/write-operations will, therefore, be atomic.

Synchronisation

As shown in Figure 4.1, direct-to-task notification has been introduced at introduced for two pairs of tasks. The notification mechanism from the pose estimator to the pose controller replaces the previous synchronization using a binary semaphore. The reason for doing this is simply to remove an unnecessary variable, and the task notification feature of the FreeRTOS API requires minimal resources when used as a binary semaphore (see Section 2.3). The implementation is simple: The pose estimator increments the notification value of the pose controller task, which unblocks when this happens. It then clears the notification value, and continue execution. It will also continue if no notification is received after 1000 ms, in case there should be an error in the pose estimator task.

The mapping notification works in the same way. At every iteration, the mapping task checks if it has received a notification. If it has, it will run the line extraction algorithm and clear the notification value. The notification value can only be set by the sensor

Tab. 4.2.: Values for the parameters in the pose controller for the NXT robot

Variable	Value	Unit
`radiusEpsilon`	`5`	[mm]
`maxRotateActuation`	`40`	
`maxDriveActuation`	`45`	
`rotateThreshold`	`0.5235`	[rad]
`driveTreshold`	`0.0174`	[rad]
`driveKp`	`600`	
`driveKi`	`10`	
`speedDecreaseThreshold`	`300`	[mm]

tower task, and it does so when the sensor tower has reached one of the endpoints of its allowed rotation range, or when the translation or rotation status of the robot changes.

Pose controller and estimator

The pose controller in its current state is not performing as well as it should. No detailed test have been made for its performance, but it can be easily seen that the robot deviates significantly from its path when translating, and it does not rotate as accurately as we require. Figure 3.2 illustrates this.

The parameters for the PI-controller algorithm and the calibration values in the pose estimator determine this performance. At the start of this project, these were mainly adjusted to perform well with the Arduino robot. Some changes had to be made when implementing the improved functionality from that robot into the NXT. These were mostly estimated from observations, and no new calibration of the sensors was made. Poor calibration is likely to be the most significant source of error for the pose estimator task. Table 4.2 lists the variables in the pose controller that need to be tuned and their current value.

- `speedDecreaseThreshold` specifies the distance from the target at which the robot should begin to decrease its speed so that it gently reduces motor actuation and coasts for a short while before it reaches the target. This value was initially set too high, which caused the robot to stop inside the tolerate radius of error around

Tab. 4.3.: Macros for enabling robot functionality

`#define ...`	Description
`COMPASS_CALIBRATE`	Affects the task creation in `main.c`, so that only the communication and compass calibration tasks are created. Will not function correctly until the calibration task is properly implemented.
`SENSOR_CALIBRATE`	Not in use
`MAPPING`	Enables the mapping task
`SEND_LINE`	Enables the sending of line updates in the mapping task. Has no effect if `MAPPING` is disabled. Should be disabled when `SEND_UPDATE` is enabled.
`SEND_UPDATE`	Enables the original sending of measurement updates to the server from the sensor tower task. Should be disabled when `SEND_LINE` is enabled.
`MANUAL`	Disables low level anti-collision in the sensor tower task. Must be enabled when the robot is controlled in manual drive mode.

the target (`radiusEpsilon`), but without actually reaching the goal. The value has been increased to 300 millimetres, and the target is now reached successfully.

- The value of the variables that specify the maximum allowed actuation for the motors during translation and rotation, `maxRotateActuation` and `maxDriveActuation`, has been reduced some. The values used for the Arduino motors were too high for the NXT, and they have been reduced by approximately 50%.

Defines

For the robot to be compatible with the original system, some preprocessor directives have been added in parts of the code that enable certain features, so that some functionality can be enabled/disabled by defining certain macros. The macros in Table 4.3 are all located in `defines.h`.

Other changes

- The movement task which remained after the work done by Amsen (2017) has been entirely removed. This has not made any impact on the system, as it was no longer interacting with any other task.

- Some of the features that were implemented by Lien (2017) were located in `server_communication.h`. It contained the implementations of the functions that are called by other tasks in the system for sending data to the server, such as `send_handshake()` and `send_update()`, in addition to the handling of incoming messages and part of the ARQ protocol functionality. When creating the new module for the communication task, which was extracted from `main.c`, it was decided to merge `server_communication.h` with the new `communication.h`. This has made the module naming more intuitive since it avoids confusion about the difference between these two.

4.3 Creating a universal application

One of the goals of the robot project is to make the software as generic as possible so that it is easy to transfer it to a new system. This makes it easier to add new robots and to distribute new features as they are developed. Helders (2017) started working on this but did not make much progress. Several things had to be considered for the design:

- The robots are based on different platforms. The Arduino contains an 8-bit AVR based MCU, while the NXT and EV3 are both based on ARM 32-bit MCUs.

- Platform specific functions from native libraries have used some places in the code, such as the AVR's atomic blocks. These should be replaced with equivalents from the FreeRTOS API.

- The application must meet the memory constraints of all robot types. It can not, for instance, require more than the 16 KB of SRAM available in the Arduino's MCU.

- Some variation in hardware versions should be tolerated. For instance, not all data sources used for position estimation may be available in all robots.

Variations in word length

The differences in worth length can have consequences for data integrity if one is not careful when using variables that are read and written to by more than one task. For instance, an integer variable of 16-bit length may require several load and store operations in the MCU to alter the data, while the operation may be done atomically in 32-bit architecture. This means that it is important to protect critical sections with mutexes or semaphores when designing an application where we should not care about which platform it ends up being used on. Luckily, FreeRTOS provides plenty of mechanisms to ensure thread safety and data integrity.

Stack size for tasks

Listing 4.3 describes the call to the FreeRTOS function that creates the communications task with a priority of 3. The third parameter determines the stack size in the number of words. This means that the amount of memory allocated for a given argument value will vary across platforms. If this call is made by the NXT application, which has a word length of 32 bits, the allocated stack size would be $250 * 32$ b $= 8$ KB. For the Arduino, which has a word length of 8 bits, the result would be $250 * 8$ b $= 2$ KB.

Listing 4.3: Function from the FreeRTOS API for creating a task

```
xTaskCreate(vMainCommunicationTask, "Comm", 250, NULL, 3, NULL);
```

Physical dimensions and other constants

For the pose controller task to function correctly, it needs information about the specific robot's wheel dimensions, width, length, sensor tower placement among other things. These are currently specified with preprocessor macros in an include-file in each robot. A universal application will have to take these differences into account.

Server Application 5

5.1 Changes made

For the server to interact properly with the new version of the NXT robot, some changes had to be made to the communication, mapping and navigation modules. Compared to the work done on the robot application, the time spent on this was relatively short. All the modifications to the existing code were attempted to be as compatible with the original system as possible so that the server application can be used directly by other people using it in their project work. This has mostly been achieved, but in some cases, larger modifications have been made which broke some of the compatibility. This report should comment on this where this issue is present.

5.1.1 The simulator

Simulated robot class

An instance of the SimRobot class is created to simulate the behavior of a robot. The simulator was originally built around a single type of robot that contains all the parameters that are needed for emulating the properties of a real-life robot. This includes variables such as speed, movement direction and status, physical constants and the current measurement data for the distance sensors. It also stores the buffers used by the new mapping process and contains the necessary methods for manipulating these parameters. These are implemented so that they simulate the way a real-life robot receives and executes commands.

Some of the new functionality added in this project required changes to this class that would break its compatibility with the existing simulator process. To maintain compatibility with the original robots some polymorphism was introduced to make the system treat the new type of robot functionality as a special case. Figure 5.1 shows how the subclasses SlamRobot and Drone inherits the SimRobot class. This means that a standard robot in the system is represented by an instance of SimRobot, while the SlamRobot and Drone types are extended versions of this.

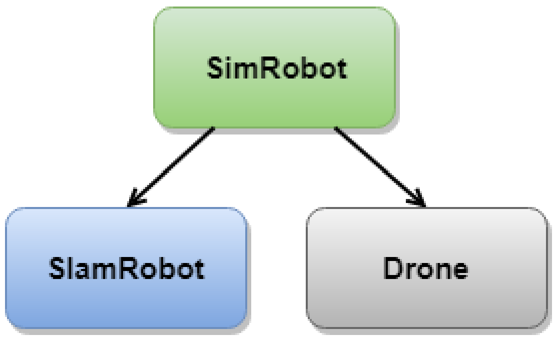


Fig. 5.1.: Hierarchy of the simulated robot objects

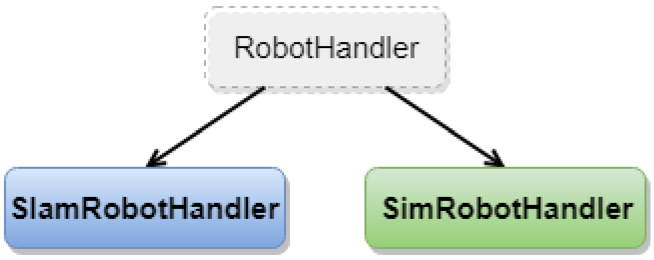


Fig. 5.2.: Hierarchy of the robot handler classes

There is only one significant modification that has made this inheritance necessary. The method `moveRobot()` contained in SimRobot is called by the simulator when it wants to simulate a change in the robot's pose. This movement depends on the current target that has been set by the simulator. This will not work correctly for the SlamRobot since it handles the movement and navigation by itself, and does not receive target commands from the navigation module in the server. This method is therefore overridden in this subclass to comply with the navigation procedure further explained in Chapter 7. The Drone subclass overrides some methods in a similar way, but this is not explained any further here since that feature is not included in this project's problem description. The reader is referred to Bjørnsen (2017) for more on this.

Robot handler

Each instance of the aforementioned simulated robot-classes is handled by a designated thread within the simulator module called a robot handler. Instances of SlamRobot must be treated differently from the original robots. The controlling thread must treat them as a special case to ensure this. The solution was to introduce inheritance to separate the two types of behavior, similarly to the way it is done for the simulated robot class. The existing class RobotHandler was converted to an abstract class which provides the most basic methods, such as pausing and unpausing the robot. Figure 5.2 shows the hierarchy of inheritance.

The SimRobotHandler is assigned to a robot of the generic type SimRobot or a Drone. This thread provides the behavior of a default simulated robot in the system, which ensures that the basic type of robot behavior remains unchanged.

Instances of the SlamRobotHandler class are assigned to robots of type SlamRobot. The difference from the SimRobotHandler is that it contains the methods for updating the point buffers and creating and merging line segments (see Section 6.2) in addition to the navigation functionality. The line extraction process is simplified quite a lot here since it was only intended to be used for testing the concept before implementing it on the physical robot, and not to get any accurate results.

5.1.2 Communication

To transfer line segment data from the robot to the server some additions to the receiving end was made. The way that incoming messages are parsed needed to be adapted to accommodate the new message structure for data related to the generation of the map. This functionality is located in the package named Communication.

The first step of parsing received messages is executed in the InboxReader class, where data is treated according to the message type. In the original system of message types, the data used for mapping was transmitted using the type called UPDATE. For the transmission of line segments, a new type called LINE has been added. The data fields contained in each of the messages are illustrated in Table 5.1. All variables contain integers. Both types contain the robot's pose, which always needs to be kept up to date regardless of any other data that is exchanged. Since the LINE message contains all the coordinates that are needed for the server to add the new data to the map, it does not need to transmit the angle of the sensor tower. This value is only needed in the process of placing a point measurement in the grid map when only the distance values are known.

Although they contain different data, the same methods are called from the InboxReader and further into the RobotController. As shown in Listing 5.1 and Listing 5.2 they both process to call the robot objects method `addMeasurement()`, but in the case of line data the heading is set to 0, and the irData array is replaced with an array containing the coordinates of the line segment's endpoints.

The `addMeasurement()` method in the Robot class generates an object of type Message which it puts at the end of the queue that leads to the mapping controller. At this stage,

Tab. 5.1.: Data contained in message types UPDATE and LINE

UPDATE	
Field	Bytes
x	2
y	2
heading	2
towerAngle	1
[]sensorValues	4

LINE	
Field	Bytes
x	2
y	2
heading	2
startX	2
startY	2
stopX	2
stopY	2

Listing 5.1: Method called from the robot controller for adding new distance measurement data to the robot object

```java
public boolean addMeasurment(int address, int measuredOrientation, int[]
    measuredPosition, int irHeading, int[] irData) {

    Robot robot = getRobotFromAddress(address);
    if (robot == null) {
        return false;
    }

    return robot.addMeasurement(measuredOrientation, measuredPosition,
        irHeading, irData);
}
```

Listing 5.2: Method called from the robot controller for adding new line segment data to the robot object

```java
public boolean addLineMeasurement(int address, int measuredOrientation,
    int[] measuredPosition, int[] line) {

    Robot robot = getRobotFromAddress(address);
    if (robot == null) {
        return false;
    }

    return robot.addMeasurement(measuredOrientation, measuredPosition,
        0, line);
}
```

Listing 5.3: Method in the Robot class that puts new measurement data into the queue for processing by the MappingController

```
public boolean addMeasurement(int measuredOrientation, int[]
    measuredPosition, int towerHeading, int[] irData) {

     int[] irHeading = new int[irData.length];
     for (int i = 0; i < irData.length; i++) {
          irHeading[i] = (towerHeading + irSensors.getSpreading()[i]);
     }
     Measurement measurement = new Measurement(measuredOrientation,
         measuredPosition, irHeading, irData);

     return measurements.offer(measurement);
}
```

it is not possible for the mapping controller to tell if that Message contains distance measurement data or the coordinates of a line segment.

5.1.3 Mapping

The module that handles the actual generation of the map has undergone significant changes. Here too it was kept in mind that the system should still function with other types of robots independent of the work done in this project, and this has for the most part been handled successfully.

The process begins with the MeasurementHandler method being called. Its job is to process the distance and angle values and calculate the positions in the real world that correspond to these. This is achieved through some trigonometric operations that are not very advanced. This process cannot be applied to line segment data.

As mention in Sub-section 5.1.1 there exists some functionality for handling measurements of the type originating from a drone, but this was not entirely functional when work on this project started. Since the new version of the NXT robot and the Drone both transmit their measurements as line segments, the implementation handles data from those types equally. A switch-case structure was added, treating these as special cases and executing default behavior for the existing type of robot. The part of the code where this is handled is shown in Listing 5.4. The default behavior is to update each of the four position elements, one for each distance sensor, in an array of Sensor objects. These are then read by the MappingController which finds the location in the map corresponding to each of these positions, and sets the status of the cell in the grid map at that location to "occupied." The measurement from the NXT only contains two sets of coordinates, but it is still desirable to use the existing procedure in the mapping controller as far as possible. This was solved by adding each of the endpoints of the line segment to a Sensor-object so that only two of them are in use.

When the MeasurementHandler has finished, the MappingController retrieves the sensor data in the same way as it would for the default robot type. However, if the name of the robot is "NXT" or "Drone," the data is treated differently, as shown in Listing 5.5. The two positions in the Sensor array mark the start and end point of the line, instead of two measurement positions as they otherwise would. The map is re-sized to fit the new addition, and the position values are converted to locations in the map. Finally, each cell between and including start and end are set to "occupied." This also means that no cells will be marked as unoccupied, so unless some other robot with distance sensors have explored a cell, every cell in the map except the ones that are part of a received line segment, will be unexplored.

5.1.4 Navigation

If the robot should be able to determine its directions and way-points all by itself, the NavigationController module of the server must be prevented from interfering. The working principle behind the NavigationController and path planning algorithms are rather complex, and will not be discussed any further here. However, for this process to have any effect at all, the robot in question needs to be added to an array of names that should receive orders from the server. The solution to this was similar to the one used in the MappingController; the NXT robot was treated as a special case judged by its name. If it is found in the list of robot names that are to be controlled, the algorithm skips it and proceeds, as shown by the simple code snippet in Listing 5.6.

The other module which would interfere with the navigation is the CollisionManager, which can send orders to the robot to prevent collision with either walls, other objects or robots. The same solution is used here: The algorithm ignores any robots with the name NXT.

Listing 5.4: Switch-case structure in MeasurementHandler for treating NXT and Drone as special cases

```
String unitName = robot.getName();
switch (unitName)
{
case "Drone":
case "NXT":
sensors[0].setPosition(new Position(irData[0], irData[1]));
sensors[1].setPosition(new Position(irData[2], irData[3]));
break;

default:
for (int i = 0; i < 4; i++) {
    int measurementDistance = irData[i];
    if (measurementDistance == 0 || measurementDistance > sensorRange) {
        sensors[i].setMeasurement(false);
        measurementDistance = sensorRange;
    } else {
        sensors[i].setMeasurement(true);
    }

    Angle towerAngle = new Angle((double) irheading[i]);
    Angle sensorAngle = Angle.sum(towerAngle, robotPose.getHeading());

    double xOffset = measurementDistance * Math.cos(Math.toRadians(
       sensorAngle.getValue()));
    double yOffset = measurementDistance * Math.sin(Math.toRadians(
       sensorAngle.getValue()));

    Position measurementPosition = Position.sum(robotPose.getPosition(),
        new Position(xOffset, yOffset));

    sensors[i].setPosition(measurementPosition);
}
}
```

Listing 5.5: Handling of sensor data for the NXT robot and Drone

```java
Sensor[] sensors = measurementHandlers.get(name).getIRSensorData();
if (robot.getName().equals("Drone") || robot.getName().equals("NXT")) {
    Position start = sensors[0].getPosition();
    Position end = sensors[1].getPosition();
    map.resize(start);
    map.resize(end);
    ArrayList<MapLocation> line = getLineBetweenPoints(map.
       findLocationInMap(start), map.findLocationInMap(end));
    line.forEach((location) -> {
        map.addMeasurement(location, true);
    });
    continue;
}
```

Listing 5.6: Handling of the NXT robot in the NavigationController

```java
for (int i = 0; i < robotNames.size(); i++) {
    String name = robotNames.get(i);
    Robot applicationRobot = robotController.getRobot(name);

    if (name.equals("NXT")){continue;}
    ...
}
```

Listing 5.7: Method in the server that returns a new command in polar form

```
findCommandToTargetPoint(target, currentPosition, robotHeading)
    distance = distanceBetween(currentPosition, target);
    rotation = angleBetween(currentPosition, target) - robotHeading +
       360) % 360;

    if (rotation > 180) {
        rotation -= 360;
    }
    int[] command = {rotation, distance};
```

5.1.5 Data format for set-points from server

Original solution

The original way for the server to tell the robot where to go next was to send an order message containing a heading and a distance value in a polar coordinate form, (θ, d). The final step of this process on the server side is shown in Listing 5.7. θ is the desired rotation, given in degrees. It can contain a positive or negative value, where $\theta < 0$ represents counter-clockwise rotation. d is the distance to translate, given in centimeters. This solution was seen to have an impact on the accuracy of the navigation performance and was part of the problems attempted solved by Amsen (2017). Section 2.5 of his thesis provides a summary of this.

New solution

To allow for a more precise communication of movement commands from the server to the robot, the format of the order message was changed. Instead of using relative values (θ, d) we now operate in Cartesian coordinates (x, y), with a rewritten position control algorithm. Listing 5.8 shows the code line in the NavigationController in the server that has been altered. Instead of calculating way-points in the old format before sending them to the robot, the target is now sent directly.

Listing 5.8: Creation of new way-point in NavigationController

```
// int[] newCommand = findCommandToTargetPoint(nextWaypoint,
    currentPosition, currentOrientation);
int[] newCommand = {(int) nextWaypoint.getXValue(), (int) nextWaypoint.
    getYValue()};
```

5.1.6 Other

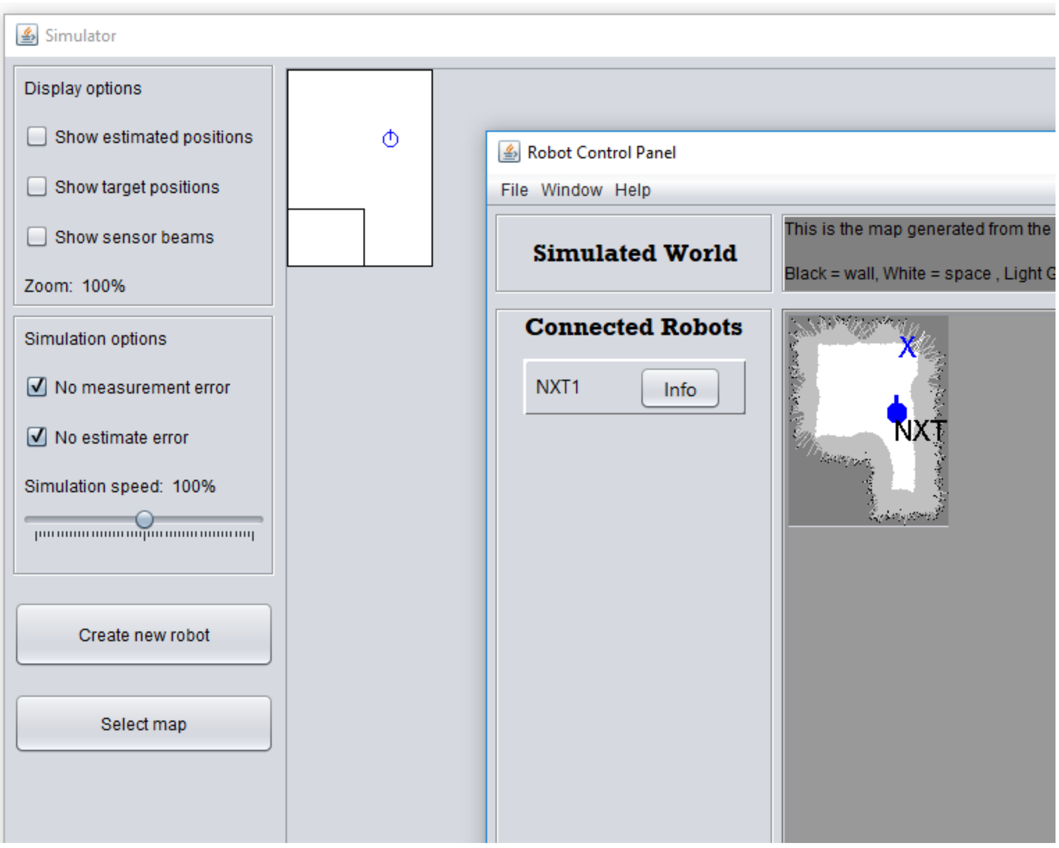
Manual drive

The server provides an option when connecting to a new robot that enables manual drive. This means that the robot will not receive any way-points from the server, and the user can specify the exact movements it should make. In the original application, the movement commands needed to specify an angle to rotate, and a distance to travel. Due to the changes made to the data format for communication as described in Section 5.1.5, this would no longer work. The manual drive GUI has therefore been modified to support the new command format with Cartesian coordinates.

5.2 Simulation

The server contains a simulator module which is well suited for testing out functionality before implementing in the real robot. It was a necessary tool when developing the application since access to a physical area for testing was often limited. Compiling the program and downloading it to the robot, starting the simulator application and initiating the connection is quite time-consuming. All the functions that are part of the new mapping task in the robot were developed and tested by simulation before any development began on the physical robot. Figure 5.3 shows the GUI of the server application used together with the simulator. The map used here has the same dimensions as the test labyrinth set up on the office floor, as shown in Figure 6.4.

Since the server application is written in Java, it was tempting to utilize the full potential of object-oriented programming when developing the new methods. This does of course not transfer very well into an embedded system with a microcontroller running an application written in C. This was kept in mind while developing the methods for the


Simulator

Display options

Show estimated positions [ ]

Show target positions [ ]

Show sensor beams [ ]

Zoom: 100%

Simulation options

No measurement error [✓]

No estimate error [✓]

Simulation speed: 100%

Create new robot

Select map

Robot Control Panel

File Window Help

Simulated World

This is the map generated from the

Black = wall, White = space , Light G

Connected Robots

NXT1 Info

Fig. 5.3.: Map generated from simulation of the office test track

line creation procedure in the simulator. As a result, much of the new code related to the Line-class is not very Java-idiomatic.

When developing an application that is to be run on a PC, we can consider memory to be an almost unlimited resource. It is not a limiting factor when choosing which algorithms we want to use, for instance for path planning on a large grid map. This has also been kept in mind since part of the challenge in this project lies in the efficient utilization of limited memory resources.

Mapping 6

From looking at previous theses written that include the topic of mapping, we can see that the majority of the results do not satisfy our needs with regards to map quality. The map representations generated are inaccurate, skewed, or not compensated for observational and positional errors. This chapter presents an alternative way to handle the mapping part of the system, and how it can be implemented as a task in the robot application.

6.1 Map representation

The server currently generates a graphical representation of the environment from a grid of cells that are shown as occupied, unoccupied or unexplored. It also shows the cells marked as weakly restricted in grey. This is a feature used for navigational purposes and is explained further in Chapter 7. The grid map does not show a 1-to-1 representation of the environment. To reduce the memory consumption and CPU usage of the server application host, each location in the map covers a 2x2 area of the real world. The significant difference in memory requirement can be shown by using the measurements of the test track in Section 6.4 as an example. The application requires at least 1 byte per cell to store all the parameters it needs for the navigation algorithm.

Number of cells needed for a 1x1 representation: $104 * 141 - 55 * 41 = 12409$ cells.

If we use a 2x2 representation, this can be reduced to the square root $\sqrt{12409}$ KB ≈ 111 KB. This means that even for a test track as small as this we would need for than 12.4 KB to store the grid map if we did not decrease its resolution. This is a challenge when trying to implement mapping in a system with low memory such as this. Transferring this way of representing the environment directly into the robot is likely to function well in small, closed of environments. However, it is desirable to make the new functionality more general so that it can operate on more than a few square meters, and also be transferred onto other robots in the system.

Figure 6.1 shows the graphical difference between the two resolutions. The quality improvement that comes with a higher resolution is noticeable. However, depending on how accurate a representation we need in a given situation, a lower quality image may

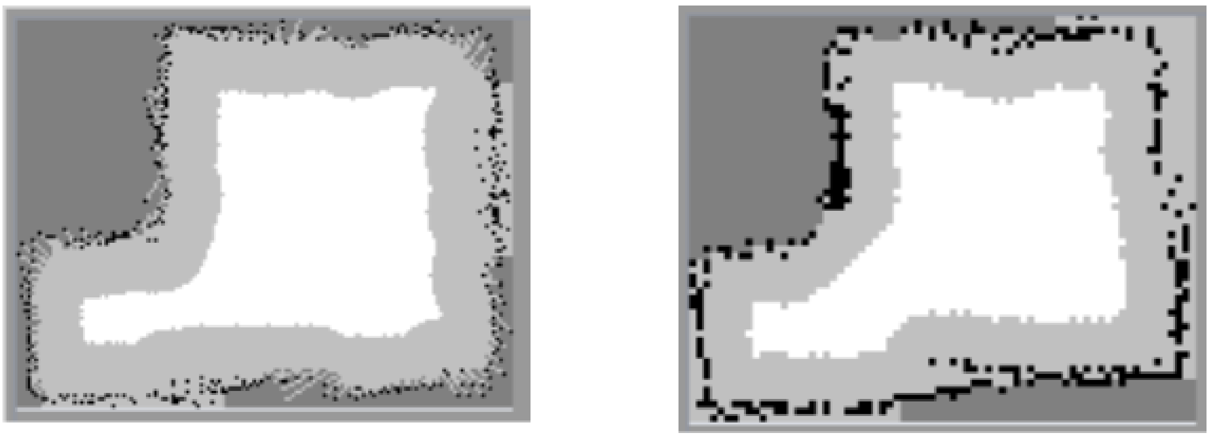


Fig. 6.1.: Generated map of the test labyrinth for different map resolutions. Left: 1x1 cm, right: 2x2 cm.

be sufficient. For the testing of new functionality in this project, a 1x1 grid has been used primarily, since a higher resolution more clearly shows the result of the new solution.

6.2 Line extraction procedure

The algorithm presented in this chapter is based on the procedure from Section 4.4 in Masehian et al. (2017). The robots in our system are quite similar to the ones used there, so it could be interesting to investigate if it was possible to do an implementation that would work with the robot and server configuration in this project.

The new approach is based on representing detected obstacles in the environment as line segments. A line segment l is represented by two endpoints p and q. This has the potential for reducing the storage space needed for sending measurement data, and the server application has been adapted to accept line data and add it to the grid map. The line segment data is also what is used in feature-based mapping algorithms, which provides an alternative to the current grid-based approach. Section 2.1 has more on this.

Sensing

Whenever an intersection point with an obstacle is calculated by an IR measurement, its coordinates (x, y) are added to the sensor's corresponding point buffer (PB). Each sensor has a unique PB so that measurements made be different sensors are not mixed

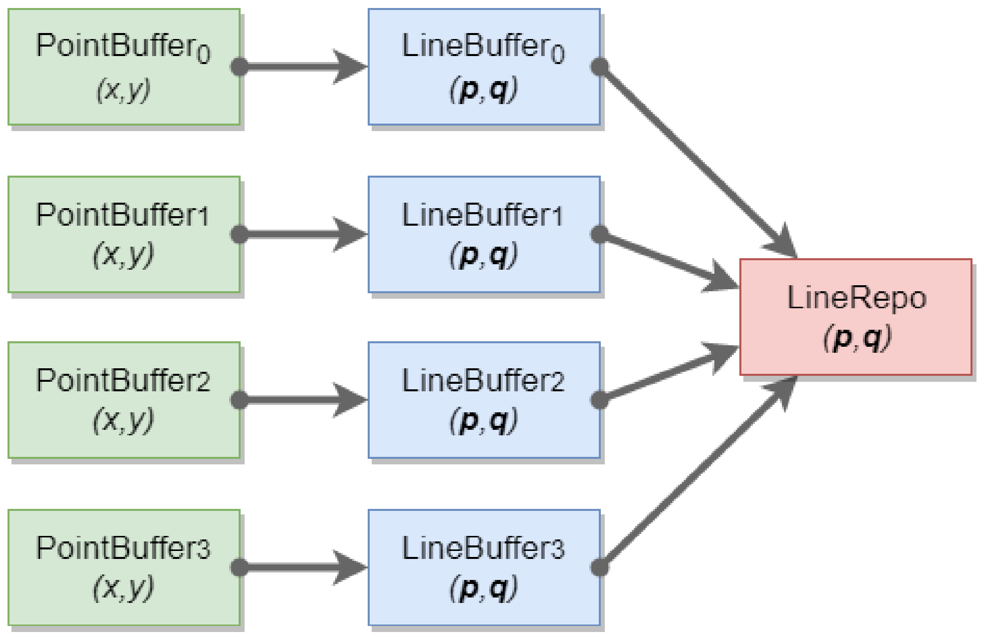


Fig. 6.2.: Buffers used in the steps of the line extraction procedure

with each other in the next step of the procedure. This goes on until the sensor tower either stops rotating or changes direction, which ensures that the content of the PBs are in a sequence corresponding to the movement in the environment before the points are evaluated. The detection process is then started. It consists of three main steps as depicted in Figure 6.2.

Line creation

The `line_create()` method starts by checking if the first three points in PB are collinear. If this is true, it moves on to check if the fourth one is collinear with the first two. This is repeated until a point fails the test. A line segment is then generated with the last collinear point as the endpoint. The process then continues with the non-collinear point as starting point until the end of the buffer. If a new line segment is attempted to be started from the last or second last point in the buffer, they will be discarded, and the procedure will end. This is because at least three points are required for a line segment to be attempted created. The created line segments are added to a line buffer (LB) corresponding to the given sensor. Finally, the PB is emptied. A parameter for determining the threshold for whether or not three points are collinear needs to be set.

Line merging

Several parameters need to be set for this process. When all of the line buffers have been filled with short line segments, these need to be combined into a common repository which serves as the basis for the map representation. This is done through the `line_merge()` function which is applied to each LB in turn. The algorithm compares every line segment in the LB with the ones in the repository to determine if it is feasible to combine the two candidates into one. In the first iteration there will be no lines in the repository to compare with, and so the lines in the LB will be added directly to the repository. This will usually only happen for the first LB that is attempted to be merged.

The candidates for merging must pass two tests before the process is started.

1. The slopes m_1 and m_2 are compared. If $|m_1 - m_2| \leq \mu$, where μ is a threshold parameter, the lines move on to the next test.
2. The distances between all the distances from endpoint to endpoint of the segments are compared. These are the four values $dist(\mathbf{p}_1, \mathbf{p}_2)$, $dist(\mathbf{p}_1, \mathbf{q}_2)$, $dist(\mathbf{q}_1, \mathbf{p}_2)$ and $dist(\mathbf{q}_1, \mathbf{q}_2)$. If either of these are less than or equal to a distance threshold δ, the lines may be merged.

The mathematical expression for both tests is shown in equation (6.1).

$$\begin{aligned}&(|m_1 - m_2| \leq \mu)\wedge\\&((dist(\mathbf{p}_1,\mathbf{p}_2) \leq \delta) \vee (dist(\mathbf{p}_1,\mathbf{q}_2) \leq \delta) \vee (dist(\mathbf{q}_1,\mathbf{p}_2) \leq \delta) \vee (dist(\mathbf{q}_1,\mathbf{q}_2) \leq \delta))\end{aligned} \tag{6.1}$$

Under ideal conditions with little variance in the measurements, the parameters μ and δ can be set to minimal values just to compensate for floating point rounding errors. In reality, however, these will need to have higher values and be appropriately tuned. This process is discussed in Section 8.1.

Repository self-merge

After combining the line buffers into the common line repository, it is possible that some lines in the repository could be merged with each other. This is handled in the third step of the line extraction process in the function `repo_merge()`. The function works by comparing every line in the repository with every other element and merging those that are eligible. The resulting merged lines are added to a newly allocated array, along with the lines that could not be merged with any others. The function is called in a loop which terminates when the length of the resulting buffer is equal to the length of the buffer generated in the previous iteration. This buffer now contains the lines that are ready for sending to the server.

6.3 Task outline

The working principle of the mapping task is illustrated by the flowchart in Figure 6.3. The functions from the FreeRTOS API that are used are shown in green color, and the functions written in this project are in blue.

1. When the task is created, and the system scheduler is started, it will first allocate memory for and initialize the buffers used by the mapping functions. It then enters a loop that it never returns from, and reads the global status variables. If the hand-shook variable is set to FALSE, which means that the robot is not connected to the server, or if the system is paused, the task will not run. It will instead yield and delay for 200 ms before checking again.

2. If it is connected and not paused, the next step is to read the current pose of the robot from the globalPoseQ. This queue always contains just one element, which is the most recently estimated position and heading of the robot. Here the function `xQueuePeek()` is used. This call will simply read the data of the content of the queue, and not remove what it reads. A slight wait time of 10 ms is added here in case for some reason the queue cannot be read from, or that it is empty. That should however not normally happen, and the execution should continue immediately.

3. Next it checks if there are any new measurement data has arrived from the sensor tower task. Unlike the previous step the function `xQueueReceive()` is used. The difference between this and `xQueuePeek()` is that it will remove the item that it reads from the queue. For it to be possible to add new data to the point buffers, the

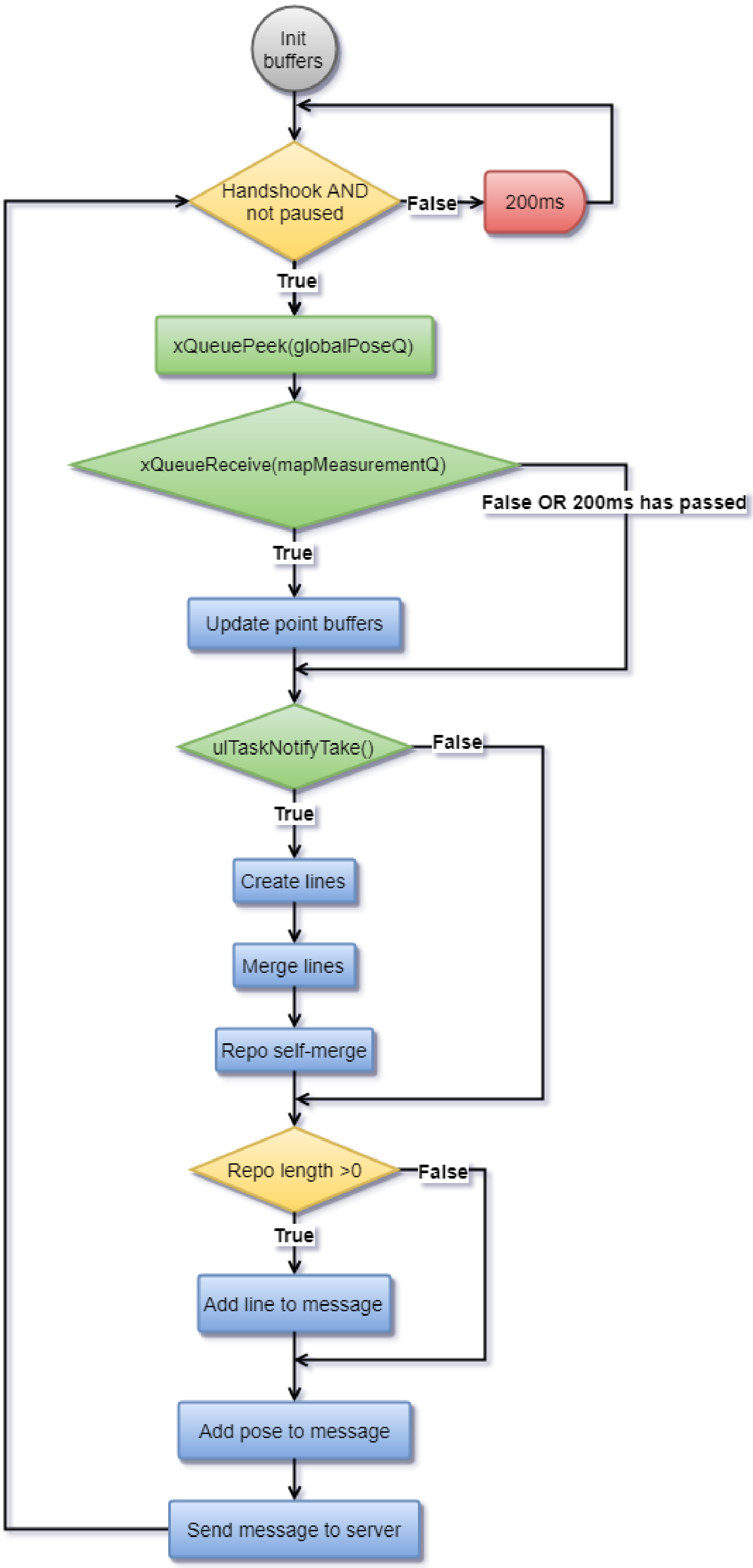


Fig. 6.3.: Flow chart for the mapping task

sensor tower task needs to put a new measurement into the queue first. If it has, the point buffers are updated. If not, there is no data to add and execution proceeds. The IR sensors are sampled every 200 ms, which means that the measurementQ will never be updated more frequently than that. Because of this, a block time of 200 ms is specified `xQueueReceive()`, so that it will wait up to that amount of time for data to appear on the queue. This ensures that the task is not run more often than necessary since it has little purpose without any data to process.

4. The task makes use of the direct to task notification mechanism of FreeRTOS (explained in Section 2.3) to determine when to run the line merging procedure. The sensor tower task signals the mapping task by calling `xTaskNotifyGive()`. The point buffers will continue to be updated until the task is told to run the line merging procedure. When this should happen, is decided by the sensor tower task. For the point measurements to appear in a line so that the line merge functions can run correctly, the merge process needs to be initiated at the correct time. This is set to happen when one of the following events occur.

   a) The movement status of the robot changes, meaning that it changes rotation direction or starts or stops translating.

   b) The sensor tower servo has reached one of the endpoints of its rotational interval $[0°, 90°]$, and the rotation direction changes.

   There is also a simple check for buffer overflow in the function for updating point buffers, which initiates the line merging procedure when one or more of the point buffers are full. However, this should be considered redundant since that will only occur if the sensor tower task fails to meet its deadline. That would most likely mean that the system has crashed. `ulTaskNotifyTake()` checks if a notification has been received. If it returns TRUE, the procedure in Section 6.2 is initiated. If not, this step is skipped.

5. The last step of the task is to send data to the server. The line repository is checked to see if it contains any new lines. If it does, that line is added to the outgoing message and removed from the line repository by decrementing its length variable. Since the previous steps of the task will not always result in new lines being detected, the line repository will often be empty. The coordinates of the endpoints of the line added to the outgoing message are then all set to 0. The server then treats this as "no new line received." Finally, the pose read in the second step is

added to the message before it is sent. The original update frequency of 200 ms is maintained.

6. It is important to be aware of which measurements units are used. The mapping task reads values in radians in the interval $[-\pi, \pi)$ and millimeters from the poseQ, which it then converts to radians in the interval $[0, 2\pi)$ and centimeters for use inside the task. The IR data it receives from the measurementQ is in centimeters, and the servoStep-value is in degrees in the interval $[0, 90]$. Finally, it rounds the floating point position values and casts them to *int16_t*, and the heading value is converted to degrees in the interval $[0, 360)$ before being sent to the server.

6.4 Testing

For preliminary testing during development of the steps of the procedure described in Section 6.2, a test track was set up on the floor of an office landscape. Its dimensions are shown in Figure 6.4. It was constructed from boards made of wood with a smooth surface. The surface of the floor is made of linoleum. The server application was run on a stationary computer that was located approximately 3 meters away from the center of the labyrinth on a table approximately 1,5 meters above the floor. This distance is short enough to guarantee minimal impact from Bluetooth connection disturbances, which is an unwanted factor in this specific test case.

To get an impression of how the algorithms were performing and what line segments it generated it was necessary to obtain a visual representation of the result of the line creation. A temporary solution for drawing all generated segments on top of the GUI of the server application was therefore implemented. This solution circumvented the ordinary procedure of passing measurements through the communications module before they are added to the grid map.

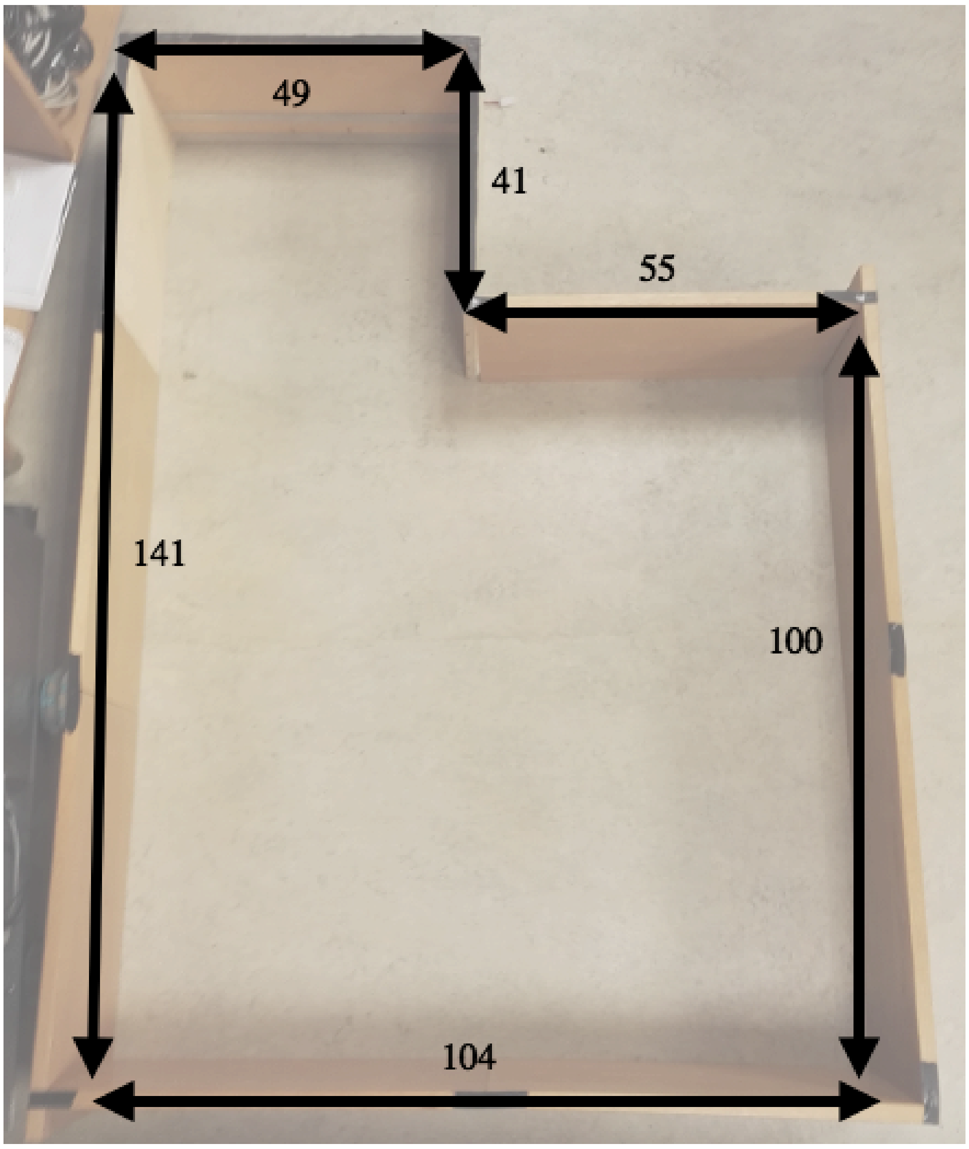


Fig. 6.4.: Dimensions of the test labyrinth. Lengths are in centimetres.

Navigation 7

For the robot to operate more autonomously, it needs to have a strategy for navigation. The server has been handling everything related to map generation, anti-collision handling and calculation of new way-points for the robots. This chapter describes the process of implementing more of the navigation functionality into the NXT robot so that it can efficiently explore the environment without receiving orders from the server.

Because of the new mapping functionality, navigation using a grid map will no longer be possible. When sending the endpoints of lines only to the server, no cells will be marked as explored, since there is no ray of free cells from the robot until the measurement point, as was the case with the old solution. This means that the algorithm used in the navigation controller module of the server application can no longer be used. In theory, it could be possible to implement the grid map solution into the robot, so that it kept an internal representation of the environment in that format, and ran the frontier-based exploration algorithm itself. However, as explained in Section 6.1 the constraints on processing power and RAM prevent this solution.

Unfortunately, this project did not make as much progress as was planned for developing a new navigation algorithm, so this chapter will only describe the theory behind the suggested new implementation. Still, the principle behind it has been well thought through, and the following documentation should serve as a good foundation for future development and realization of the new solution.

7.1 Wall-following

As mentioned in Subsection 2.2.2 it is possible to implement a navigation algorithm which relies on readings from short-range distance sensors for following a wall. This is a fundamental subject in robotic mapping which has been discussed a lot in both earlier works at ITK, and in other academic reports, so it was very much possible to end up "reinventing the wheel" here. The result in this thesis is based on *Wall Follower* (2006), but with a significantly modified implementation.

Figure 7.1 shows the program flow of the suggested algorithm. It could be thought of as a separate task in the robot application if it was added to the existing implementation.

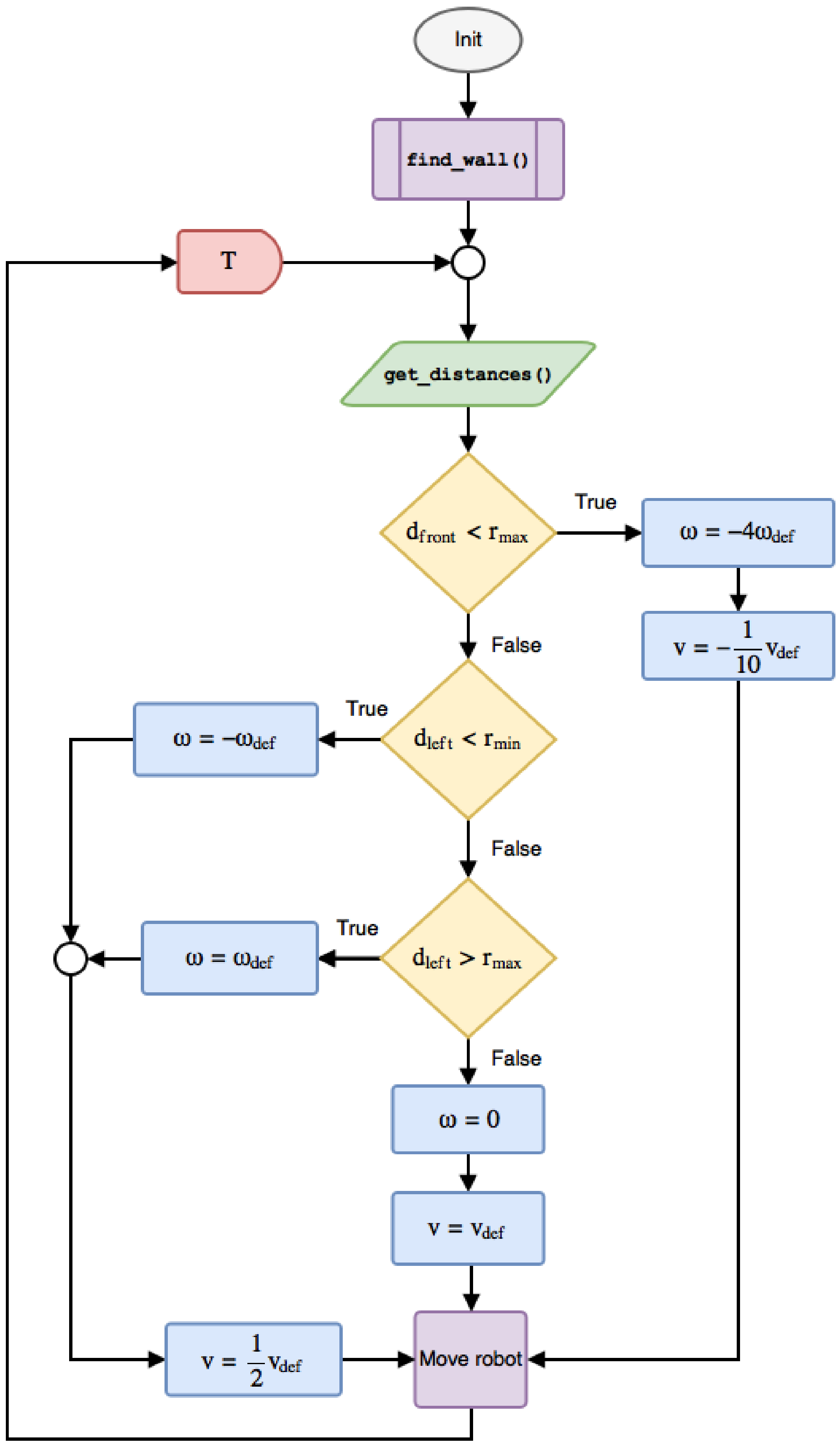


Fig. 7.1.: Flow chart for the proposed wall-following algorithm

In a functioning implementation, it would, of course, need some form of synchronization and data exchange with the other tasks for sensor data and motor control. As can be seen from Figure 7.1 the loop consist of the following steps.

1. Perform initial setup and wait for the system to connect
2. Execute `find_wall()` (see Subsection 7.1.1)
3. Get the distance values on the front and left side. `get_distances()` will include the calculations from `get_left()` and `get_front()` in Listings 7.1 and 7.2 respectively.
4. Check if the distance values are within the allowed range $[r_{min}, r_{max}]$, and determine if it is necessary to rotate. Determine and set suitable values for ω and v.
5. Tell the robot to start moving at the specified speeds
6. Delay for a specified period T, allowing the robot to move a distance according to the set speeds.
7. Repeat

Variables

Table 7.1 describes the variables that are used in the navigation process. The period T is the one used in the delay blocks in Figure 7.1 and 7.2. The unit for distance has been set to millimeters. This is an intuitive choice for this process since the existing tasks in the system operate in this unit, except for the readings from the IR sensors which are in centimeters. The existing pose estimator and pose controller tasks operate in millimeters and radians, so it is natural to use the same precision level for this task.

Suitable values for r_{min} and r_{max} need to be selected. This can be based on the existing concepts of *restricted* and *weakly restricted* cells in the grid map navigation algorithm (Section 5.2 in Thon (2016)). As he describes, the reason behind implementing this feature was that the robot would not always move and respond with complete accuracy to the commands from the server, and a form of a safety net was needed in case of too long reaction times. This implementation does not consider this problem since all decision making, and execution of movement commands will happen internally with no possibility of errors in communication with an external unit.

Tab. 7.1.: Variables related to the navigation functions

Variable	Unit	Description
v	[mm/T]	Speed in the forward direction
v_{def}	[mm/T]	Default speed in the forward direction
ω	[rad/T]	Rotational speed
ω_{def}	[rad/T]	Default rotational speed
θ	[rad]	Angle of the sensor tower relative to the robot heading
d_{front}	[mm]	Distance in front
d_{left}	[mm]	Distance to the left
$d_{left\_previous}$	[mm]	Value of d_{left} from the previous iteration
d_{max}	[mm]	Maximum value of a sensor reading
r	[mm]	Distance between robot and target wall
r_{min}	[mm]	Minimum allowed distance between robot and wall
r_{max}	[mm]	Maximum allowed distance between robot and wall

r_{min} corresponds to the width of what would be the restricted area in the vicinity of an obstacle. Thon (2016) set this value to 15 cm, but this is explicitly calculated for the old AVR robot, and may not work for the NXT. The property of weak restriction adds merely another area where movement is restricted outside the restricted area. This is not necessary to consider in this implementation.

The default speeds v_{def} and ω_{def} must not be set so high that the robot can end up very close to the wall or far outside r_{max} in only one iteration of the algorithm.

Closing the loop

According to Figure 7.1, the process will run in an eternal loop as long as there exists some wall to follow. The environments that the robot operates in are usually enclosed in some way so that it will eventually end up at the same place at some point. This means that there should be some condition for when a complete lap along the enclosing walls of the environment has been completed, and the program should terminate.

The concept of loop closing is a very advanced issue, and the difficulty level of implementing this makes it unrealistic to add this feature to this project. Solutions to the problem involve advanced mathematical theory, and some examples which show the complexity of the issue can be found in Section 3.3 in Huang and Beevers (2005).

7.1.1 Find wall

The second block after initialization indicates that a subsystem will execute before the first iteration of the loop begins. This block represents a separate function which makes the robot start translating until it encounters a wall, which it then aligns with on its left side. This functionality makes it possible to start the system regardless of where in the environment the robot is located, or even if its actual location is unknown.

Figure 7.2 shows the program flow of how the robot behaves from it is first placed at an arbitrary location in the environment and until it is aligned with a wall and ready to start following it. After acquiring the initial distances d_{front} and d_{left}, it checks if both of them are longer than the maximum wall distance threshold r_{max}. If that is true, the robot moves a short distance forward, and the execution delays for a short period before rechecking the distances.

If the value of d_{left} or d_{front} is found lower than r_{max}, an obstacle has been encountered and the alignment procedure begins. This is done by monitoring the change in d_{left} as the robot rotates to the right. The value is sampled at a given interval T, and stored in the $d_{left\_previous}$ for comparison in the next iteration.

If the value of d_{left} remains the same as $d_{left\_previous}$ after an iteration, the robot is considered aligned with the wall, and the function will exit.

7.1.2 Distance values

It is common for robots that are intended for wall-following and equipped with short-range IR sensors to have them placed with a fixed orientation in the directions that the algorithm needs measurements from. In this case, it would be natural to have 1-3 sensors pointing left in a slightly spread out pattern, and 1-2 facing forwards with a similar setup. The sensors on the NXT are however mounted on a rotating tower at the top of the robot. This means that to get the measurements that we need in the left and forward directions some calculations need to be made.

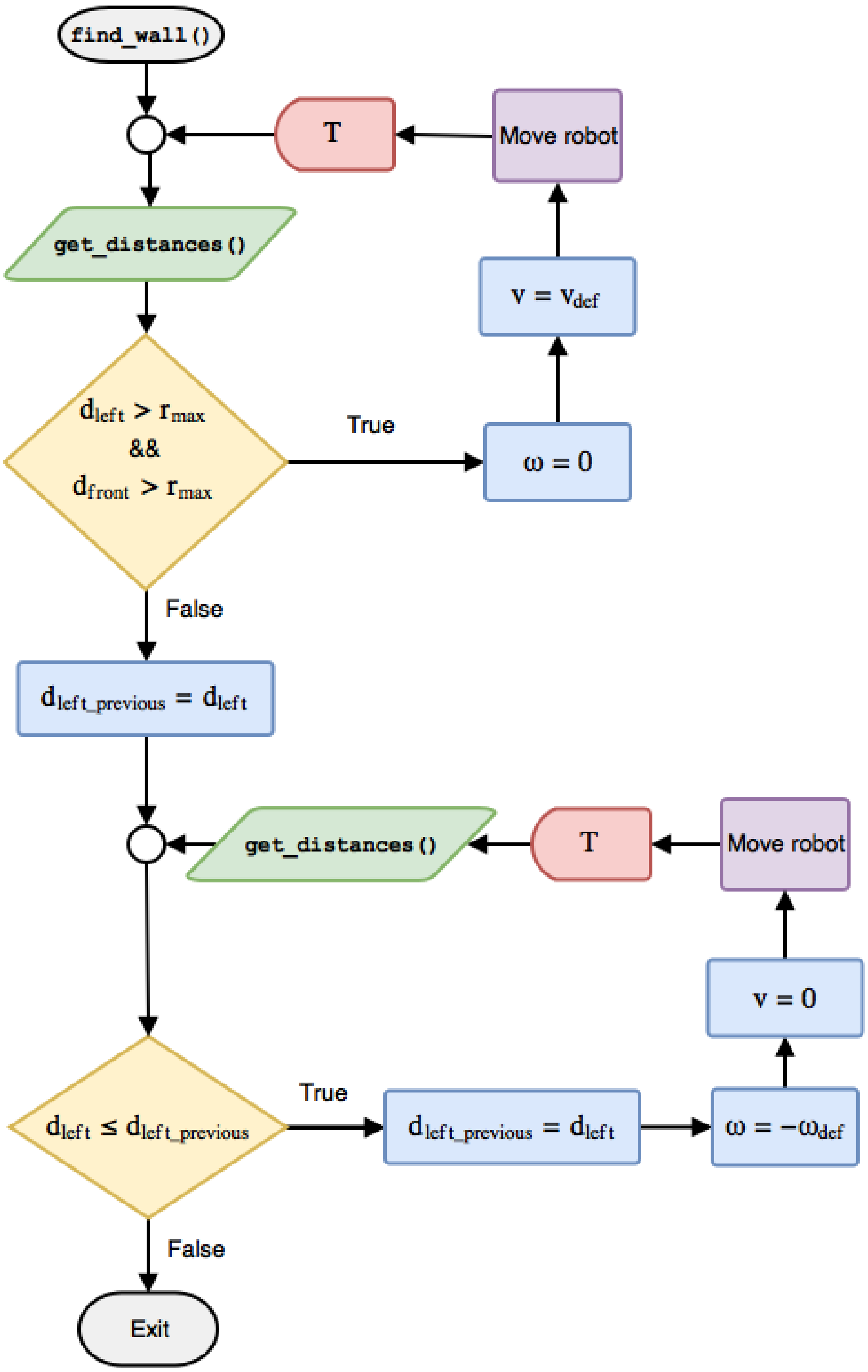


Fig. 7.2.: Flow chart for the find wall feature

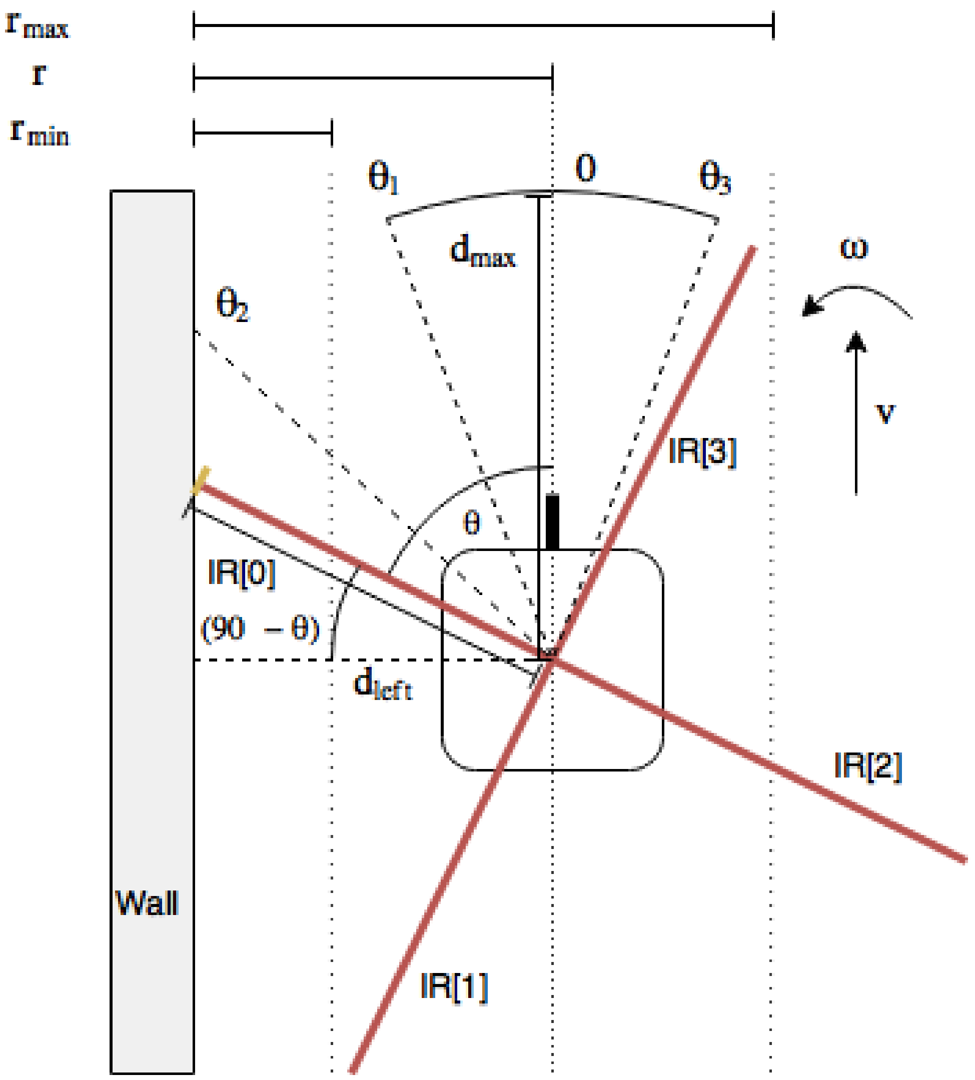


Fig. 7.3.: Illustration of the values in Table 7.1 while following a wall

Listing 7.1: get_left()

```
if (theta == 0) {
    dLeft = IR[1];
} else if (theta > 0 && theta < theta_2) {
    dLeft = // previous value
} else if (theta >= theta_2 && theta <= 90) {
    dLeft = cos(90 - theta) * IR[0];
} else {
    // Invalid tower angle
    dLeft = // previous value
}
```

Figure 7.3 shows a simplified outline of the NXT robot (not to scale) with four sweeping IR sensors mounted to a rotation tower on top of the robot. It is assumed here that the tower is located in the center of the robot, while in reality it is located about 2 cm behind the center point between the wheels. The tower is rotating continuously back and forth in the interval $[0°, 90°]$ at a speed of approximately 5 $\mathrm{deg/s}$.

For the calculations of d_{front} and d_{left}, only measurements for some specific intervals of θ are used. As can be seen from the figure these are in the ranges $0°$ and $[\theta_2, 90°]$ for d_{left}, and $[0°, \theta_1]$ and $[\theta_3, 90°]$ for d_{front}. To avoid excessive text on the left side of the figure, θ_3 has been marked on the right side of the forward facing cone. This makes more sense if the angle is thought of as relative to the starting point of the ray IR[3]. When the ray of an IR sensor is within one of these sectors, we can use its measured value to calculate the distance in front and to the left by performing some simple trigonometry. Knowing the angle θ and letting the IR value be the hypotenuse of a right triangle, the side representing d_{left} or d_{front} can be found. Listings 7.1 and 7.1 show the working principle of the functions called for reading distance values.

d_{max} is the defined maximum value for distance readings that are used in the application. The sensor is physically capable of delivering readings for higher distances, but the accuracy decreases rapidly from that threshold and out. This is, therefore, the maximum distance that the robot can observe around itself. Note that the IR data is given in centimeters, and needs to be converted to millimeters before any further calculations are made.

Listing 7.2: get_front()

```
if (theta >= 0 && theta <= theta_1) {
    dFront = cos(theta) * IR[0];
} else if (theta > theta_1 && theta < theta_3) {
    dFront = // previous value
} else if (theta >= theta_3 && theta <= 90) {
    dFront = cos(90 - theta) * IR[3];
} else {
    // Invalid tower angle
    dFront = // previous value
}
```

Results and Discussion 8

8.1 Mapping

In order to get the most optimal result from the line extraction procedure there are several parameters that need to be tuned.

8.1.1 Buffer lengths

As described in Section 6.2 and shown in Figure 6.2 three types of buffers are used in the process, each with a size that needs to be specified. These are defined as `PB_SIZE`, `LB_SIZE` and `L_SIZE` in the application. They were all set to 50 during development. No overflow of the point buffers or line buffers has been seen, but the line repository will fill up if no lines are sent to the server or if it is not otherwise emptied. This means that these values could probably be reduced, but since there is a good amount of free RAM in the system they have been left as they are.

8.1.2 Test setup

The line segment extraction procedure has been tested by placing the robot at the center of the narrow corner part of the labyrinth from Figure 6.4. Exact placement is shown in Figure 8.1. This is an area with walls that are perpendicular to each other and the floor. This was beneficial since we wanted as ideal an environment as possible for this basic test procedure. All navigational functionality was disabled so that the robot stood still in place while the mapping process was active. The parameters mentioned above were then changed in turn to see what effect they had on the result, and to try to determine the optimal values for achieving the most accurate representation of the environment.

The dimensions of the images generated by the server vary somewhat for each mapping session. This is unavoidable, since the map is expanded only if a new line continues outside the current image, and this will differ between each choice of parameter values. The overlay showing the actual layout of the walls should make it easier to compare the results visually.

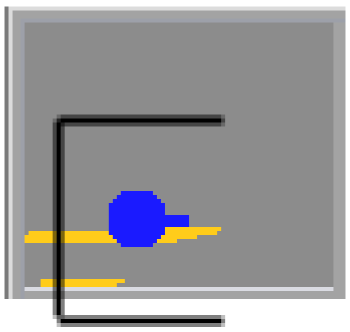


Fig. 8.1.: Map result using very large parameter values: $\epsilon = 100, \mu = 100, \delta = 100$.

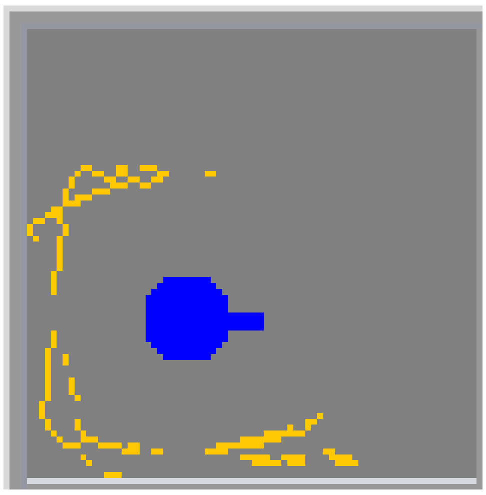
The representations shown in the figures are generated after 20 line segments have been sent from the robot to the server. Figure 8.1 shows the result after setting all the parameters to a very large value. This means that a line segment is always generated between the first and the last points in the point buffer supplied to the `line_create()` function since every point will always be within the tolerance which decides if they are candidates for extending the line. Two line segments may lie on different sides of a corner and have completely different directions, but they will still be merged. This leads to the unusable representation in Figure 8.1.

8.1.3 Collinearity tolerance ϵ

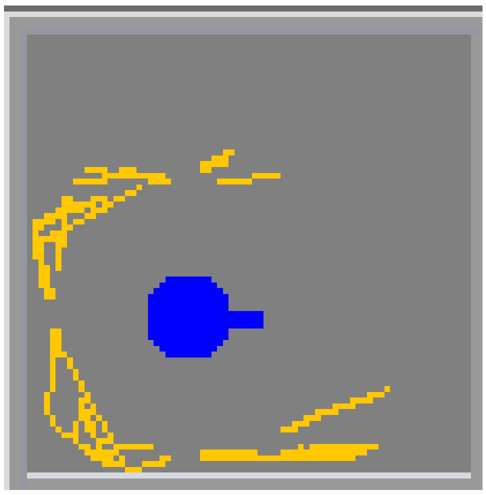
Equation (2.1) shows that a parameter value must be specified to determine how strict we want to be when testing if three points should be allowed to form a line segment. In a situation with perfectly accurate measurement data, and no noise or other disturbances, this parameter would only be there to account for rounding errors in floating point calculation, and should be set to a minimal value. In our system, however, it is set high enough to account for those factors as well.

To get an impression of the impact that the collinear tolerance value ϵ has on the result, μ and δ were kept fixed while experimenting. The values of μ and δ were set based on previous experience with the system and should make it easy to see what the difference in ϵ alone could make. Figure 8.2 shows the results that were obtained for the different values. The overlay that shows the actual walls of the corner has been left out in these graphics, to make it easier to see the difference in this specific test.

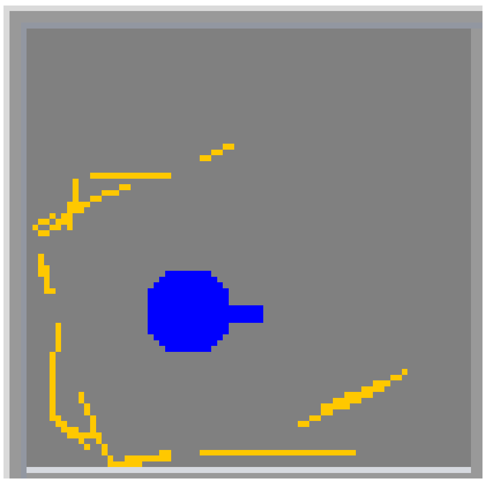
Subfigure 8.2a shows how a minimal value of collinear tolerance causes the detection process to generate multiple short line segments, as opposed to a larger value in Subigure 8.2c resulting in longer, straighter ones. When using a very large value of ϵ, as seen in Subfigure 8.2d, the threshold for including new points in the line creation process is



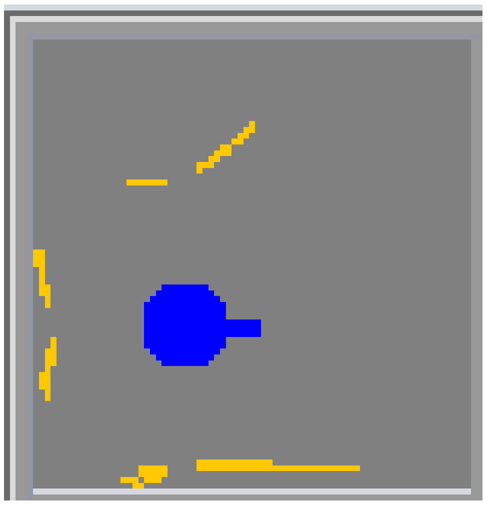
(a) $\epsilon = 1$



(b) $\epsilon = 5$



(c) $\epsilon = 10$



(d) $\epsilon = 20$

Fig. 8.2.: Results for different values of ϵ, with $\mu = 0.6$ and $\delta = 1$.

Tab. 8.1.: Angle values corresponding to different values of μ

μ	θ [deg]
0.1	5.7
0.2	11.3
0.3	16.7
0.4	21.8
0.5	26.6
0.6	31.0
0.7	35.0
0.8	38.7
0.9	42.0
1.0	45.0

so low that the resulting line will be continuous from the first to the last of the included points. The difference in angle between these long lines and the distance between their endpoints will then almost always be small enough for the segments to merge. The result of these merges will be short segments that follow the directions of the walls.

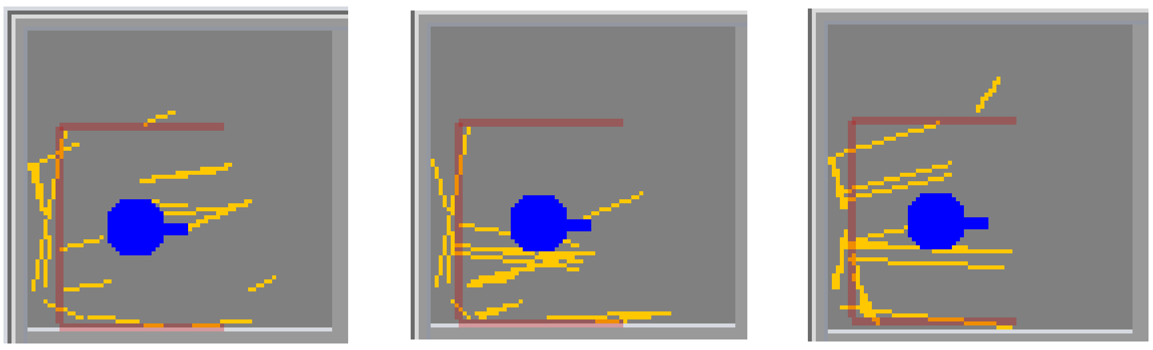
8.1.4 Slope threshold μ

The first criterion for two line segments to be mergeable is that the difference between their slopes is with the tolerance specified by the parameter μ (see Equation (6.1)). The values in degrees corresponding to selected values of difference in slope are shown in Table 8.1. The angle can be found as

$$\theta = \arctan(\mu)$$

Testing of values for μ was done with ϵ set to 10, which resulted in several quite short line segments that are candidates for a merge test. δ was set to 100, which is high enough for it to not prevent any lines from being merged.

Figure 8.3 shows how the location of the merged lines changed for different slope threshold values. The overlay shown in red gives an outline of the real-life environment. The results did not differ significantly, except for a tendency of the line segments to be longer with increasing values. It is important to notice how the lines that are oriented in approximately the same direction as the robot is heading are located at the center



(a) $\mu = 0.1$ (b) $\mu = 0.2$ (c) $\mu = 0.5$

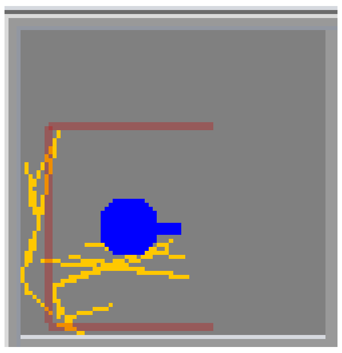
Fig. 8.3.: Results for different values of μ, with $\epsilon = 10$ and $\delta = 100$.

between the walls, almost right on top of the forward facing axis of the robot. This can be explained by the high value of δ, which removes all restrictions on the maximum distance between two line segments before they can be merged. Line segments that are generated along each of the parallel walls will only have to pass the slope test, and the merge procedure will give a resulting line segment that is located approximately in the middle of these lines. Note that this only happened for the segments along the parallel wall. The wall to the left will not have any parallel lines within the part of the environment shown in the figure. This could be seen quite consistently in all three subfigures in Figure 8.3 since the parameter δ did not come into play in this case.

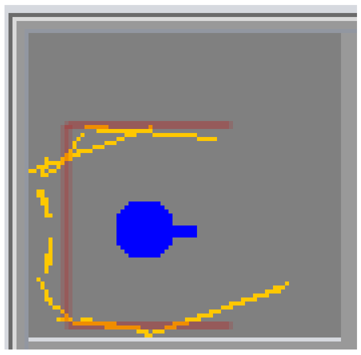
8.1.5 Distance threshold δ

As for the two other parameters, the same method for comparing values was used for δ. The physical interpretation of the parameter is related to the distance between features in the environment that should be represented by unique lines. To determine a range of reasonable values to test the measurements of the corner section in Figure 6.4 should be considered. The benefit of correctly tuning this parameter is to avoid the issue presented in Figure 8.3, where lines from opposite walls are merged. The test was set up to find the threshold value which prevents that situation.

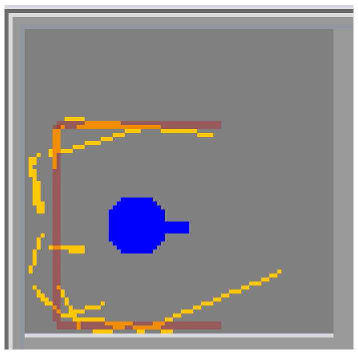
It would be reasonable to assume that this value should be close to the distance between the two parallel walls in the corner, which is shown in Figure 8.5. The result of the test with this parameter (Subfigure 8.4a) shows that this was not the case. The result indicated that too many lines were merged, so the threshold had to be less than this. The results were improving for decreasing values down to $\delta = 45$, as shown in the rest of the subfigures in Figure 8.4. This value prevented any undesirable merges. Subfigure 8.4c also shows a good result, so an optimal value is likely to be between 45 and 46.



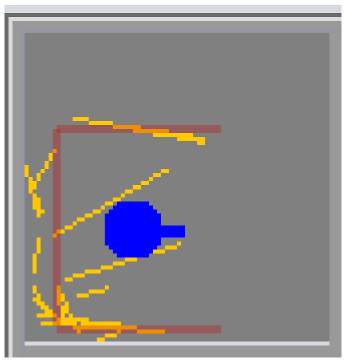
(a) $\delta = 49$



(b) $\delta = 45$



(c) $\delta = 46$



(d) $\delta = 47$

Fig. 8.4.: Results for different values of δ, with $\epsilon = 10$ and $\mu = 0.5$.

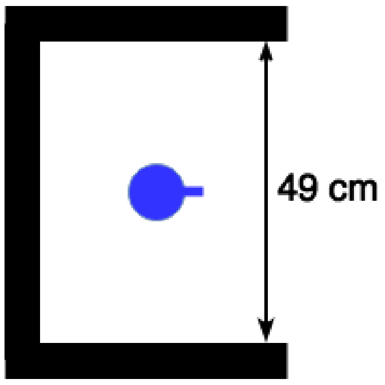


Fig. 8.5.: Distance between the opposing walls of the corner

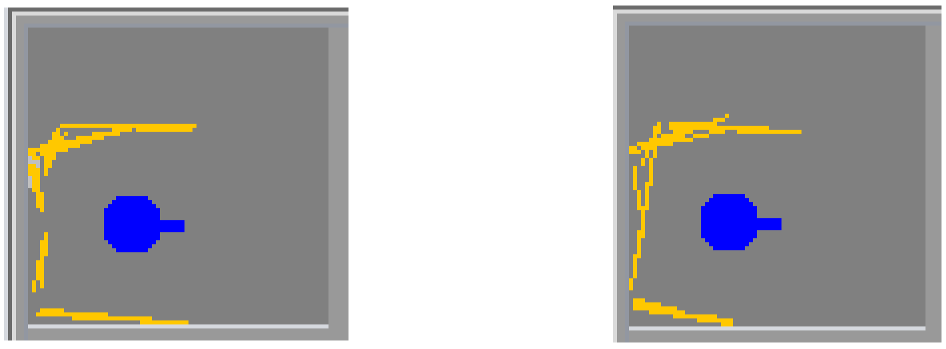

(a) $\mu = 0.3$ (b) $\mu = 0.4$

Fig. 8.6.: Results with optimal values $\epsilon = 10$ and $\delta = 20$ for two different values of μ.

The result of this confirmed that the physical interpretation of the value δ corresponds to the way it is used in the robot application. However, the error between the real-life distance between the parallel walls and the test result is significant. One possible explanation for this is that the distance sensors could be poorly calibrated. If they are not accurate, the IR data readings may indicate that objects are closer to the robot than they are in reality. This could explain how the threshold value is lower than expected. The type of surface on the walls can also have a significant impact on the measurements. In this case, they are made of wooden boards. A wholly dark or a more reflective surface may give different results.

8.1.6 Optimal combination

From the experiments for each of the parameters, it was possible to find the combination of values that gave the most accurate representation of the corner environment. Figure 8.6 shows the best result that was obtained from the knowledge of the previous tests, and some trial and error. The value of μ turned out to be the deciding factor, and a good value for this turned out to be within the range of 0.3 to 0.4. How this affects the line generation can be seen from how the lines seem to be rotated more in the clockwise direction in Subfigure 8.6b. This is because a more relaxed restriction on the maximum angle difference require for merging will cause more of the line segments with orientation along separate sides of a corner to be merged. The resulting segments when these are merged will have an angle that is influenced by both of the segments, as described in Section 2.5.

When the slope threshold is stricter, as shown in Subfigure 8.6a, not enough segments are merged, and gaps appear along the left wall. The segments are slightly rotated

Tab. 8.2.: Optimal values for the parameters in the line extraction process

Parameter	Value
ϵ	10 [cm]
μ	0.3 – 0.4
δ	20 [cm]

inward because the sensors are picking up points on both sides of the corner. Table 8.2 lists the optimal values found for the parameters in the mapping task.

8.2 Navigation

Testing

Because the project did not result in a working implementation of the navigation functionality, it was not possible to perform any tests to determine how well the proposed algorithm actually would perform. There was only made a simple addition to the robot application, where a dummy task which was supposed to simulate the resource usage of the functionality in Chapter 7. It was set up with an execution frequency approximately like the one for the mapping task so that it would consume significant CPU time. No noticeable difference in the behavior of the robot could be seen with this task enabled, so it was concluded that the robot's hardware would most likely be able to support this added functionality.

Choosing the correct variables

The algorithm's performance is mostly dependant on values set for the variables in Table 7.1. The angle values θ_{1-3} must be set so that the leftmost sector will cover an area that will make sure the robot's left side will not encounter any obstacles that may be missed by the cone formed by the interval in front, in case the robot rotates at the same time as it is moving forward.

r_{min} depends on the length from the sensor tower to the farthest point away from it on the robot. The minimum allowed distances to the wall must be at least as high as this so that the robot always has room to turn if it gets within this threshold. r_{max} should

Tab. 8.3.: Values for variables related to navigation

Variable	Value	Unit
v_{def}	200	[mm/s]
ω_{def}	$\frac{\pi}{6}$	[rad/s]
T	100	[ms]
θ_1	$\frac{\pi}{6}$	[rad]
θ_2	$\frac{\pi}{4}$	[rad]
θ_3	$\frac{\pi}{3}$	[rad]
d_{max}	400	[mm]
r_{min}	150	[mm]
r_{max}	200	[mm]

not be so high that the robot loses sight of the wall entirely. If that happens, it would be considered to be lost, and the find wall procedure in Subsection 7.1.1 would have to be rerun. Approximations of these values can be found in the collision management and navigation module of the server application, and from knowing d_{max}.

As for the default speed values, the ones that are set in the current application in the NXT will most likely work well. These are quite a bit lower than what the servo motors can maximally deliver, to save energy and to make the acceleration and deceleration more smooth.

The resulting values from both calculation and estimation are listed in Table 8.3.

9 Further Work

9.1 Communication

Implement mesh network Make the robot even less dependant on a central host by implementing a network where every robot communicates directly and shares data with the other robots. This way the robots can cooperate on solving the mapping-task without receiving any directions from the server. The server would then only serve as a GUI for viewing the map graphic.

Move communication protocol from application to dongle The communication protocol developed by Lien (2017) is currently implemented as a part of the robot application. It would be more convenient if this functionality was moved into the dongle so that the developer does not have to concern herself with this when working on a separate part of the application.

Exceptions at connection time At the end of this project the server application began to throw exceptions related to the communication protocol, more specifically BufferUnderflowException. It originates from the SerialCommunication class which receives and decodes new data frames from the incoming data stream, which means that it could be caused by wrongly formatted data messages sent by the robot. The issue is not critical, but it should be looked at to increase system stability.

Debugging Some form of debugging functionality should be re-introduced. The existing function for this was disabled when working on this project since the program size was close to the allowed limit by the compiler. The function used much memory since it included the c-library for string formatting.

9.2 Robot application

Improve pose estimation and control There is room for improvement of the Kalman filter process in the estimator task, and the whole pose estimation algorithm should receive an overhaul at some point. The parameters of the pose controller should be tuned correctly, and the performance of the PI algorithm should be evaluated. It should be made more modular, so that it is possible to disable it and use another controller, for instance for wall following.

Utilise all sensor data All the sensors that are mounted on the robot should be used in the estimation process. They need to be correctly calibrated, and it should be considered to add some form of automatic calibration at system start-up. If some of the sensors are too sensitive for use in these robots, e.g., the accelerometer, it should be considered removed and not added in future robots.

Further improve line generation With the new solution, the line segment data that is received by the server is still added to the same grid map structure as it was with the previous solution. As long as this way of representing the environment is used, there is no significant advantage to transmitting line data instead of point measurements from the robot. To take full advantage of a line segment representation, changing the way the server stores and displays its map data should be considered.

Add reversing-functionality After converting the format of the order messages from polar to Cartesian coordinates, the robot will not behave as before when it receives a target that is directly behind it. The ability to reverse is a feature of the collision manager and is often the most suitable way to get out of a conflict situation. It should be possible to implement some form of functionality that will enable the robot to move backwards if it receives a target within a specified area behind it.

Navigation Continue the work on a navigation task that was started in this thesis. Some solution should be found for the problem of closing the loop.

9.3 Server application

Adapting line data to grid map When the server receives only lines, it is currently not able to determine which cells should be set as explored. A solution for this must be found for the current navigation algorithm to work.

Creating a new application To take full advantage of a line-based map algorithm, it should be considered to write a new server application which is based on a feature based map representation. The application does not need to include navigation functionality if this is successfully implemented in the robots. Alternatives to using

Java for implementation should be considered. The possibility of writing it in C++ using the Qt framework could be interesting to look into.

9.4 Conversion to new IDE and compiler

The application for the NXT robot is based on a FreeRTOS port explicitly designed for use with IAR Embedded Workbench and its proprietary tool-chain. This means that there is no simple way to convert the project into using another compiler. After the work in this thesis the size limit is very close to being reached, and adding any more functionality the application becomes challenging when one continually needs to keep this issue in mind. Other options for compilers and IDEs should be investigated so that further functionality may be added more easily.

A solution that has been looked at by ITK is to use the GCC compiler, with is open source and may be used with any IDE, and together with the GDB debugger. SEGGER Embedded Studio or Keil μvision from Nordic Semiconductor are possible choices for a developing environment. However, this requires a port of FreeRTOS for the specific micro-controller that also has a port available for GCC. This may not always be available, but it is usually possible to adapt ports onto similar platforms without too much work.

References

Amsen, Jørund Øvsttun (2017). „Improving Navigation and Mapping with Arduino robot“. MA thesis. Norwegian University of Science and Technology.

Andersen, Thor Eivind Svergja and Mats Gjerset Rødseth (2016). „System for Self-Navigating Autonomous Robots“. MA thesis. Norwegian University of Science and Technology.

Bjørnsen, Kristian (2017). „Mapping a maze with a camera using a Raspberry Pi“. MA thesis. Norwegian University of Science and Technology.

Eikeland, Geir Henning (2016). „SLAM Implemented in Robot“. Project thesis. Norwegian University of Science and Technology.

Ese, Erlend (2016). „Real-Time Programming on Collaborating Mobile Robots“. MA thesis. Norwegian University of Science and Technology.

Helders, Konstantino Zuraris (2017). „Real-Time System Implementation on Autonomous Lego-Robot“. MA thesis. Norwegian University of Science and Technology.

Huang, Wesley H. and Kristopher R. Beevers (2005). „Topological Mapping with Sensing-Limited Robots“. In: *Algorithmic Foundations of Robotics VI*. Ed. by Michael Erdmann, Mark Overmars, David Hsu, and Frank van der Stappen. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 235–250.

Hurbain, Philippe (2017). *Batteries and Battery packs. NXT rechargeable batteries*. URL: `http://www.philohome.com/batteries/bat.htm` (visited on Nov. 22, 2017).

Hussien, Bassam and Banavar Sridhar (1993). „Robust line extraction and matching algorithm“. In: *Intelligent Robots and Computer Vision XII: Algorithms and Techniques*. Ed. by David P. Casasent. SPIE.

Lien, Kristian (2017). „Embedded utvikling på en fjernstyrt kartleggingsrobot“. MA thesis. Norwegian University of Science and Technology.

Masehian, Ellips, Marjan Jannati, and Taher Hekmatfar (2017). „Cooperative mapping of unknown environments by multiple heterogeneous mobile robots with limited sensing“. In: *Robotics and Autonomous Systems* 87, pp. 188–218.

Melbø, Henrik Kaald (2017). „Autonomous Multi-Robot Mapping“. MA thesis. Norwegian University of Science and Technology.

Strande, Lars Marius (2017). „Autonom retur og dokking av AVR robot“. MA thesis. Norwegian University of Science and Technology.

team, Javaclient (2006). *Wall Follower*. URL: `http://java-player.sourceforge.net/examples/2.x/WallFollowerExample.tgz` (visited on Mar. 12, 2018).

Thon, Eirik (2016). „Collaborating Robots for Simultaneous Localization and Mapping“. MA thesis. Norwegian University of Science and Technology.

Wallgrün, Jan Oliver (2010). „Hierarchical Voronoi Graphs. Spatial Representation and Reasoning for Mobile Robots“. In: Springer Berlin Heidelberg. Chap. 2 - Robot Mapping.

Relevant material

A

A.1 Files

1. The following source code is appended

   - The μVision project files and code for the nRF51 dongles
   - Atmel studio project with source code for the AVR microcontroller on the IO circuit board
   - IAR project for the main NXT unit
   - Netbeans project and libraries for the server application

2. Data sheets for the hardware components, and a developer's manual for the NXT

3. Design-files for the IO circuit board layout

4. Reports from previous projects and DVDs with older versions of the system

5. A copy of this report

A.2 Equipment

– NXT robot
– Charger for NXT robot
– SEGGER J-LINK debug probe
– USB cable for probe
– 20-pin flat cable
– nRF51 Bluetooth dongle (server software)

The equipment was submitted to the institute reception when to project was completed. Various older equipment and spare parts are stored in the offices at room G242.