NTNU – Trondheim
Norwegian University of
Science and Technology

Collaborating Robots

Multi-robot Exploration of an Unknown Environment

Øyvind Ulvin Halvorsen

Master of Science in Cybernetics and Robotics
Submission date: June 2014
Supervisor: Tor Engebret Onshus, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem description

Two robots with appurtenant code is given, which are able to map an unknown area using a Simultaneous Localization and Mapping (SLAM) algorithm and Bluetooth communication. Further, they are able to run the same code in parallel, and communicate over an internet connection. This thesis aims to develop an algorithm for collaboration by solving the tasks listed below.

- Develop the simulator in order to handle parallel operations in order to simplify the work with developing the algorithms.
- Improve the communication protocol (over TCP/IP) between the robots in order to achieve reliable data transfer.
- Create one common map in which both robots operate.
- Improve performance of the NXT-robot by implementing positioning and feedback on the sensor tower angle.
- Design an algorithm for collaboration, which explores the area in an efficient manner. This task can further be broken into the

objectives listed below.

- Design an algorithm for navigation, which makes optimal decisions based on the common map and the positions of the robots.
- Design a path planner, which ensures safe driving to the new position calculated by the exploration algorithm.
- Coordinate the robots such that they avoid collisions and chooses different targets across the workspace.

Preface

This thesis has been conducted as the final part of my MSc degree at the Department of Engineering Cybernetics at The Norwegian University of Science and Technology.

This thesis has been based on my project work done fall of 2013, which has given a good foundation and understanding of the system, which is used in this thesis. This has given insight into the possibilities of the system, which have enabled me to have high ambitions for the system.

This project is the latest contribution to a long series of master theses and project works by students at NTNU, which has continuously improved the capabilities and performance of the system. It has been most motivational to be a part of this bigger project, and to further develop and expand this system to include collaborating robots.

This thesis has been most educational, and I have gained insight into a new field of studying coordinated multi-robot exploration, and to implement several modules, which has to function together. I have also gained insight about realizing a physical system and all the chal-

lenges the real world poses on a system.

This thesis is meant to be used for future work, and have also aimed to document the latest update of the system. Additional documentation can be found in previous theses and project works.

Acknowledgements

I would like to thank my supervisor Tor Onshus for good advice throughout the semester.

I would also like to thank my fellow student at the office Kristian Stormo, Simen Fuglaas, Erlend Kvinge Jørgensen, Håkon Bøe, Sverre Kvamme, Håkon Søhoel and Jørgen Herje Nilsen for taking your time to have technical discussions, and for a great final year at NTNU.

Abstract

This thesis aims to develop and implement an algorithm for collaboration, such that two LEGO-robots can collaborate on mapping an unknown area using infrared sensors, a simultaneous localization and mapping (SLAM) algorithm and controlled from MATLAB.

In order to have a testing platform during development of the algorithms, the simulator was updated to allow for parallel operations and communication.

Further, the communication protocol between the robots were implemented such that data could be exchanged in a reliable fashion, which was the key for allowing collaboration.

The robots exchanged their local SLAM-generated maps such that they could operate in one common map. The map was merged by removing the initial distance between the robots.

For the NXT-robot, positioning and feedback on the sensor tower was implemented, which improved the sensor data significantly.

The main objective was to develop and implement a navigation algorithm for choosing target points, a path planner for ensuring col-

lision free driving and coordination of the robots in order to exploit the use of multiple robots.

The navigation algorithm was implemented with a Frontier-based approach and a cost function, which was minimized in order to find the next target point.

The path planner was implemented by a Breadth-first search (BFS) algorithm and the coordination was achieved by a CLIENT/SERVER-structure.

Finally, the algorithm was tested in the simulator and on the physical system. It was shown that the robots were able to collaborate on the mapping task, as was intended in the problem description.

Sammendrag

I denne oppgaven skal det utvikles og implementeres en algoritme for samarbeid, slik at to LEGO-roboter kan samarbeide om å kartlegge et ukjent område ved hjelp av infrarøde sensorer, en samtidig posisjonering og kartleggings algoritme, og som blir styrt fra MATLAB.

For å ha en testplattform under utviklingen av algoritmene ble simulatoren oppdatert slik at to simulatorer kunne kjøres i parallell, samt kommunisere.

Videre ble kommunikasjons-protokollen mellom robotene utbedret slik at data kunne utveksles på en pålitelig måte, noe som var nøkkelen til å realisere samarbeidet.

Robotene utvekslet SLAM-generert kart-data slik at de kunne jobbe sammen i ett kart. Kartene ble flettet sammen ved å trekke fra den initielle avstanden mellom robotene.

På NXT-roboten ble posisjonering og tilbakekobling på sensortårnet implementert, noe som forbedret måledataene signifikant.

Hovedmålet for oppgaven var å utvikle og implementere en navigasjonsalgoritme som kunne velge nye posisjoner å kjøre til, en vei-

planlegger som sikrer at robotene ikke kolliderer med objekter samt koordinering av robotene for å kunne utnytte bruken av flere roboter.

Navigasjonsalgoritmen ble implementert med en Frontier-basert framgangsmåte, og en kost-funksjon som ble minimert for å finne neste posisjon å kjøre til.

Vei-planleggeren ble implementert ved hjelp av et Bredde-først søk (BFS) og koordineringen ble oppnådd ved hjelp av en KLIENT/SERVER-struktur.

Til slutt ble samarbeids-algoritmen testet på simulatoren og robotene. Det ble vist at robotene var i stand til å samarbeide om kartleggingen, som var målet for oppgaven.

Summary and conclusion

This chapter will conclude all the tasks in the problem description based on the testing in chapter 9 and the discussion in chapter 10, and determine if the goals of this thesis were reached.

The first task was to update the simulator such that it could handle parallel operations, which was achieved by letting the simulators communicate with each other as if they were the physical robots.

In order to improve the foundation for debugging, the GUI of the system was extended such that more plots could be analyzed, and the GUI was made more flexible such that the plots from different iterations could be viewed.

A fundamental condition for the collaboration was to ensure reliable communication over the TCP/IP connection between the robots. This was accomplished by writing a new data transfer protocol on top of the TCP/IP protocol such that messages was taken care of in MATLAB. This protocol allowed for a flexible and reliable communication between the robots, such that important data could be guaranteed delivery.

In order to have the robots collaborate in the same map, their local SLAM-generated map was exchanged such that both robots knew what the other had explored. As a result, the robots could exploit each others information. The map merging was implemented by removing the initial offset between the robots, which were sufficient when the system performed a small number of scans, since the position and measurement errors grew over time.

For the NXT-robot to be as functional as the IR-robot, positioning was implemented based on odometry on the NXT-brick. In addition, feedback was implemented on the sensor tower heading such that the sensor data from the NXT-robot has been improved significantly.

The main objective of this thesis was to design an algorithm for collaboration. This was realized by the Exploration algorithm, which consisted of a navigation part and a path planning part.

Before the work with developing the exploration algorithm started, a literature study was conducted in the field of coordinated multi-robot exploration in order to get insight in this field, and ideas for this thesis.

The navigation algorithm was developed with a Frontier-based approach, which aims to maximize the discovered area at each step. Further, each frontier was assigned a cost in the cost function, and the frontier that minimized the cost function was the new target destination for the robot.

Next, the path planner calculated a path to the target point by a Breadth-first search (BFS), and the path is smoothed by a line-fitting approach.

In order to coordinate the robots, a CLIENT/SERVER structure was used. The SERVER-robot finds target points for both robots and is responsible for the coordination.

Extensive testing in the simulator has shown that the Exploration algorithm has completed the mapping task both for a single and two-robot system. The algorithm was finally tested on the physical system, where both robots contributed to the mapping. Hence, the robots collaborated in the mapping task as intended. Therefore, it is concluded that the tasks described in the problem description has been fulfilled.

Contents

Chapter 1

Introduction

1.1 Motivation

The multi-robot exploration problem is a fundamental problem in robotics with many applications such as search and rescue, military use, minesweeping or generally searching hostile environments. For humans, the ability to navigate in an arbitrary environment is taken for granted, but is a challenge for a robot team, and a fundamental ability if considering for instance humanoid robots.

Multi-robot exploration has several advantages over a single robot system. First, a robot team is potentially more effective than a single robot, given that the robot team is coordinated. Second, the use of several cheap robots instead of one expensive robot is probably cheaper, and introduces redundancy since the system will tolerate loosing robots. Moreover, several robots is advantageous for the si-

multaneous localization and mapping (SLAM) algorithm, where the robots can use each other's position and observations for correcting their own positioning errors and map displacements.

1.2 Outline of the thesis

This thesis uses two different LEGO-robots, controlled from MATLAB over a Bluetooth connection, and an internet connection for communication between the robots.

The aim of this thesis is to design an algorithm for collaboration, such that the use of two robots is exploited. Most of the development will be done in MATLAB with the simulator, but the goal for the thesis is to run the two LEGO-robots with the new algorithm. In order to run the physical robots, a few improvements has to be done in order to make the robots fully functional and ready for running.

The algorithm for collaboration is called 'The Exploration algorithm' and will consist of two main parts. First, a navigation algorithm will find a target position based on a Frontier-based approach. Next, a path planner will calculate a path to this target point.

In addition to this, the robots are coordinated by a SERVER/CLIENT-structure where the SERVER ensures the coordination for the robot team.

1.3 History of the LEGO-robot project

This thesis is based on the work of several master theses and project works. Throughout the years, this system has been developed to in a system for mapping an unknown area with infrared sensors, a Bluetooth connection and a SLAM-algorithm. The first master thesis was conducted in 2004 by Skjelten [15], which built the IR-robot. Since then this robot has been further improved, and its functionality developed. The version of the robot which is used in this thesis was last improved by Tusvik [16] in 2009. The NXT-robot was included later, and rebuilt in 2013 by Homestad [7] to the robot which is used in this thesis. More of the history of the project can be found in [7], [16], [19] and all the previous theses can be found on the CD.

Structure of the thesis

Chapter 2 gives a general overview of the entire system, the different modules and how they are connected. Chapter 3 describes the changes and improvements done to the system. In chapter 5 the navigation algorithm is explained in detail, and the path planner is elaborated in chapter 6. The coordination of the robots follows in chapter 7. A few more considerations are made in chapter 8 in order to make the physical system function properly. The entire system is tested in chapter 9, and the results are discussed in chapter 10.

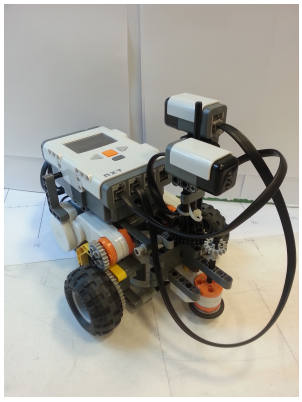
Chapter 2

Description of the robot system

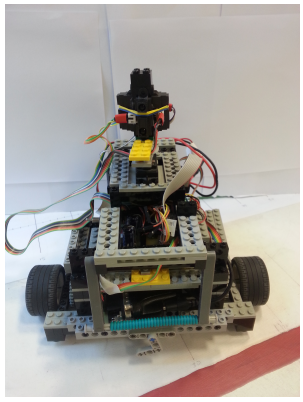
This chapter will give a general overview of the system used in this thesis as it was at the end of the spring semester 2014.

2.1 The robots

For this thesis two robots has been used called the NXT-robot and the IR-robot, which is shown in figure 2.1. For a more thorough explanation of the NXT-robot, see the master thesis of Homestad[7]. For a more thorough explanation of the IR-robot, see the project work of Tusvik[16]. A short description of the robots is given below.



(a) The NXT-robot.



(b) The IR-robot.

Figure 2.1

The NXT-robot

The NXT-robot is built from a LEGO Mindstorms 2.0 construction kit, and is a commercial product. The NXT-robot as it is now, was built and implemented by Homestad [7] in 2013. The NXT 2.0 brick is programmed in the programming language called 'Not Exactly C' (NXC). As the name suggests, it is similar to C, but is a primitive language not designed for as a general-purpose use. It supports the use of threads called **tasks**.

Since the NXT-robot is a commercial product, it has been developed dedicated toolboxes for this type of bricks, both for MATLAB use and for programming the NXT brick. For this robot the RWTH-Mindstorms NXT Toolbox [17] is used with MATLAB. In addition to the MATLAB toolbox, [17] has provided code in NXC for controlling

the LEGO motors, which has made the foundation for the current code on the NXT 2.0 brick. There are also built-in NXC-functions that can be used to control the motors directly. These functions are documented by the IDE[1] (BricxCC) for NXC.

The functionality on the NXT-brick is only to control the robot. For example controlling the motors, sending measurements from the sensors to MATLAB and estimating the position.

The NXT-robot is driven by two electrical motors, and has a third contact point at the back, which is a drag point. The third electrical motor rotates the tower, which is independent of the wheel motors.

The IR-robot

The IR robot was originally built in 2004 by Skjelten [15]. Except from the infrared sensors and the battery (and the LEGO parts obviously), all parts are built by students through master theses and project work, and has been continuously developed since 2004.

The processor is a ATMEGA32 which is programmed in C, and downloaded using a JTAG (see Tusvik [16]).

The IR-robot has also two wheels and a drag point at the back, while the sensors is mounted directly on top of a servo motor.

No work has been done on the IR-robot for this thesis since it is considered to be in a good condition. Therefore, it will not be explained in more detail in this thesis.

[1] Integrated development environment

2.2 The software

The system consists of several software modules, which interacts with each other. The system is designed such that the SLAM- and navigation algorithm operates independent of which robot that is connected, which makes the system flexible and fault tolerant. The division of the software into modules makes the system understandable and easier to debug, since the origin of the errors are easier recognized. The software modules are explained below.

The Graphical User Interface (GUI) is the interface where the robots is controlled from. The most important button is the 'Start'-button, but several other buttons are implemented in order to allow to test basic functionalities for the robots, for example the 'FullScan'-button which performs a full scan e.g. for sensor testing. The GUI also displays the main map, and several other plots which are helpful for debugging.

Each robot has its own GUI to allow for different functionalities and capabilities of different robots.

Not all buttons are supported, which is result of several projects working with different topics.

The Simulator provides six maps, which can be used for testing the algorithms of the system. The simulator is found in the GUI by choosing 'Simulator' when connecting.

SLAM and Navigation module builds the map based on the sensor data from the robots, and estimates the robot position in this map. The SLAM-algorithm is landmark based [2], which uses 'lines' and 'beacons' as landmarks. The SLAM-algorithm extracts lines from the sensor data, and beacons are extracted from the remaining measurement points. These landmarks forms the basis for the wall segments in the map. The landmarks are used for estimation of the robot position and map updating. If a landmark is observed from several different viewpoints, this information can be used to estimate the positioning error of the robots. Usually, the sensor data is more accurate than the positioning, which is exploited by the SLAM-algorithm by trusting the sensor data more than the positioning. SLAM is a major field of study, and there can be found a lot of literature on the subject.

The SLAM-algorithm used in this thesis is provided by The Centre of Autonomous Systems - Robot Navigation Toolbox, created by Arras [1].

After the map has been updated the new target destination is calculated by the navigation algorithm. This part is the main objective of this thesis, and will be treated further.

This module interacts with the robot only by receiving sensor data and the position, and sending the new position back. This module does not need information about which robot it communicates with.

This module is the 'main loop' of the system and implemented in `lineBeaconSLAM.m`, which is called from the callback function of the 'Start'-button in the GUI.

The Communication module is responsible for the communication between the two robots, over a TCP/IP connection. This module holds the TCP/IP connection handle, and is responsible for creating-, sending-, reading-, and interpretation of packages.

RobotHandler is the module, which is responsible for the communication between the SLAM and Navigation module and the robots. This module separates the SLAM and Navigation module from the robots, such that multiple robots can be added dynamically.

This module needs to know which robot is communicates with, such that the data flow between the current robot and the SLAM and Navigation module is correct.

IRInterface and NXTInterface are the modules for direct control and communication with the robots. These modules are implemented specially for controlling each robot, with functionality tailored for the robot of interest. The design with the robot interfaces allows the system to connect different robots to the system.

The Hardware module is the embedded system on the physical robots that executes the commands from the interface of the

specific robot. The functionality implemented on the physical robots are only for controlling the motors and sensors.

A graphical description of the complete system is shown in figure 2.2. The figure shows how the modules are connected, and the data flow between them. The SLAM and Navigation module is drawn to the left, with the main loop in it. The blue communication arrows illustrates the data flow between the SLAM and Navigation module and the current robot. This data flow is considered as local, since this is of no concern for the other robots. The green communication arrows illustrates the data flow between the robots, which is the communication over the internet.

Each robot will have this structure and communicate with the other robots through the communication module. As can be seen, this system design makes it simple to add or loose robots. Hence, the system can be considered as fault tolerant since it is able to loose robots, and it is flexible since it can use different types of robots, as this thesis does.

2.2.1 Program flow

This subsection will describe the program flow of the main loop and the program flow of the NXT-robot. The main loop of the entire system is summarized in figure 2.2, and will be explained more into detail below. The main loop is also well explained in [16], [7], [19].

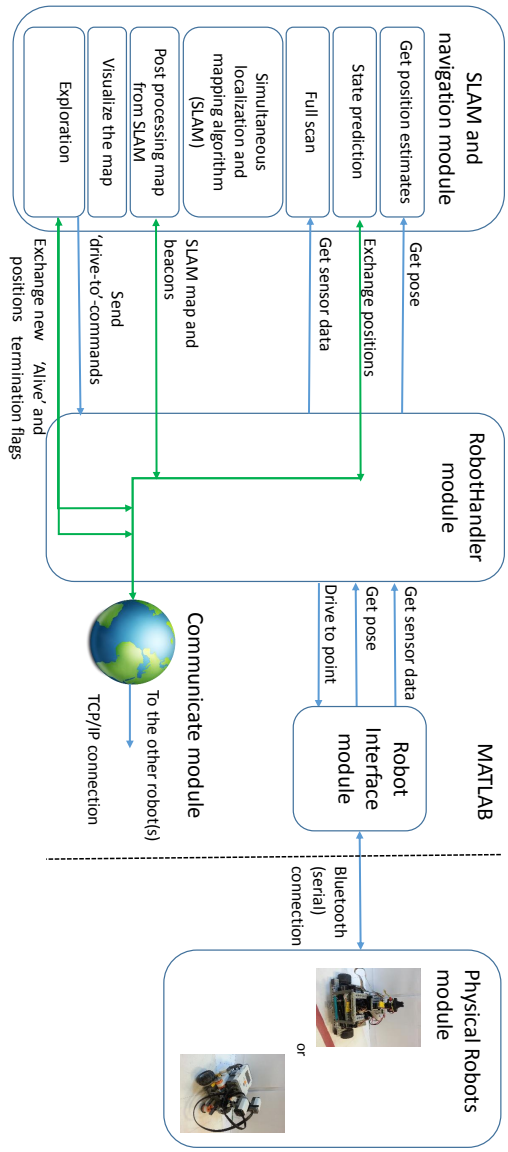


Figure 2.2: The structure of the robot system, with all modules, the main loop, communication lines (green lines are communication over TCP/IP, while the blue lines are internal communication within MATLAB and over Bluetooth with the robots.), data flow and the physical robots.

Main-loop

Get position estimate from the robot. The position is stored in the *navData*-structure and displayed in MATLAB. The position is also exchanged with the other robot, and the second robot position is also stored in *navData*.

State prediction predicts the position of the robot based on estimating the angular displacement of the wheels.

Full scan commands the robot to perform a full scan, and gets the sensor data from the robot.

SLAM uses the sensor data for extracting lines and beacons, and updates the map. The robot position is estimated by an extended Kalman filter (EKF).

Post processing the SLAM-map removes beacons close to wall segments.

Visualizing the map plots the SLAM-map in the main plotting window of the GUI.

Navigation This module calculates the new position of the robot based on the SLAM-map. There are two approaches which is the old algorithm called 'Left-Wall-Follower to Backtracker' which is implemented for only one robot. The second approach is the 'Frontier based exploration'-algorithm which is designed and implemented in this thesis. The navigation algorithm calculates the

new target destination for the robots by solving an optimization problem, and then plans a path to this position if needed.

The NXT-robot main-loop is a state machine, which chooses action based on the received command. The while-loop will check the INBOX each iteration to see if a new message had arrived from MATLAB. The state machine, together with the NXT-Matlab- protocol is summarized in table A.3.

2.2.2 Communication

The system uses two types of communication. The first is the Bluetooth connection, which connects the robots to the computer. The Bluetooth connection is set up with the use of a Bluetooth dongle, which both robots can communicate with simultaneously. This connection type is limited by the range, but for the test set up the distance from the computer to the robot is small.

The second communication type is the internet over a TCP/IP connection. This connection is used for communication between the robots, in order to allow for collaboration. The connection is between two instances of MATLAB, between the `RobotHandler/communicate`-modules of the robots.

The complete structure of the communication modules is shown in figure 2.3.

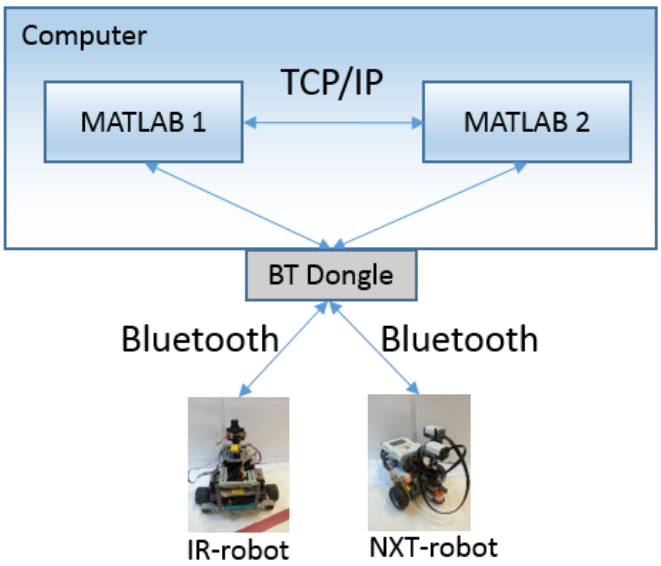


Figure 2.3: The communication structure of the system. The robots are controlled from separate MATLAB instances over a Bluetooth connection, while the communication required for collaboration between the two MATLAB instances is over TCP/IP. This design also allows the use of two separate computers.

Chapter 3

Changes made to the previous robot system

This chapter will discuss the changes made to the original system. In order to properly debug the system, the GUI was extended to show more plots. The NXT-robot needed some improvements in order to increase the precision and accuracy of the sensor data. Further, the simulator had to be updated in order to allow for parallel operations.

3.1 GUI

The graphical user interface was extended to allow for more plotting, such that the system could be visualized in more detail. This was helpful in the debugging process, and simplified the work with producing plots for this thesis. The main GUI window is shown in figure

3.1(a).

The main change on the main GUI window was that each iteration could be viewed, by clicking on the arrows. This way, each iteration could be studied in detail, instead of just having the final image when the scan was done.

The button 'Plot Iterations' were added to the main GUI, which extended the plotting GUI of Homestad [7] with two plotting windows below the three existing plotting windows, which is shown in figure 3.1(b). The left plotting window shows the tactical map as a contour plot, the border of the discovered area and the frontier points, together with the wall segments. The right plotting window shows both robot paths with the wall segments, in order to see if they are properly coordinated. These plotting windows also allowed the user to browse through the iterations.

For all new plot windows, it was made a button for plotting in a separate figure, such that results could easily be retrieved from the GUI.

Because of the network roles, the plots shown for each robot is different. Only the SERVER-robot will have access to the tactical map, since the CLIENT does not calculate the frontier points.

Under 'Navigation Strategy', the 'Frontier-based optimization' button was added, such that it is possible to choose between the two navigation strategies. Further, it can be specified if collaboration is used. If this check box is marked, the robot will initiate a TCP/IP connection, which will block until a CLIENT connects. Choosing the

'Left-wall follower to backtracker' navigation strategy will disable the 'Collaboration'-check box.

Both robots are calibrated from the GUI, which is done by clicking on the 'Calibrate NXT' for the NXT GUI. For the IR GUI, the calibration is performed from the main GUI window.

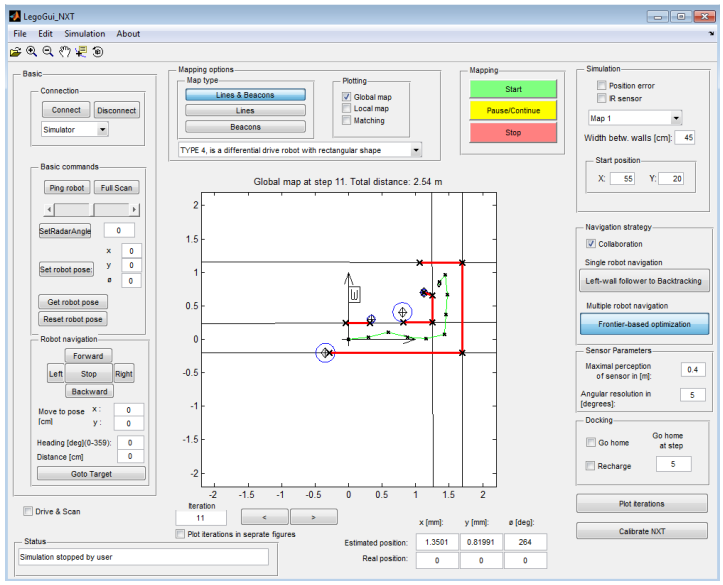
The GUI windows can be edited by 'GUIDE'. To access the GUI editor, type `guide` in the command prompt, and choose the .fig-file to edit.

3.2 NXT-robot

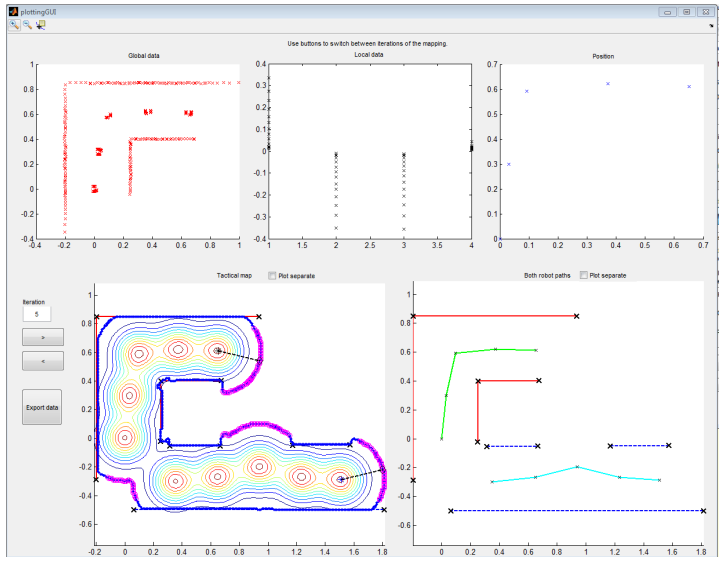
It was called to attention in [19] that the performance of the NXT-robot was not satisfactory. The sensor data had low precision and the positioning was inaccurate.

The sensor tower did not have feedback implemented. It was supposed to rotate from 0°to 180°and take one measurement every 5°. It was assumed that this happened precisely, but this was not the case. The sensor tower accumulated a growing offset such that the 0° position approached a 45°-90°offset, after a few scans. This had a great effect on the sensor data, which quickly became erroneous.

The positioning had the same issue as the sensor tower. The motor position was not measured, such that there were no knowledge of the real position. The position of the robot was calculated in MATLAB based on the reference. This approach assumed no errors in the motors, which was not the case, and lead to that the position estimate



(a) Main GUI window



(b) The plotting GUI

Figure 3.1

error grew quickly.

These two issues were solved as described in the next sections.

3.2.1 Positioning

The positioning is implemented the same way as for the IR-robot and given by the equations in 3.1(a-c) [16]. This equation needs the assumption that the change in angle $\Delta\theta_t$ is small. This assumption holds since the robot drives either forward, or turns by turning the wheels in opposite directions. This driving pattern keeps the $\Delta\theta_t$ small.

$$D_t = \frac{D_t^r + D_t^l}{2} \; [m] \tag{3.1a}$$

$$\Delta\theta_t = \frac{D_t^r - D_t^l}{B} \; [rad] \tag{3.1b}$$

$$\begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} = \begin{bmatrix} x_{t-1} + D_t \cos(\theta_{t-1} + \frac{\Delta\theta_t}{2}) \\ y_{t-1} + D_t \sin(\theta_{t-1} + \frac{\Delta\theta_t}{2}) \\ \theta_{t-1} + \Delta\theta_t \end{bmatrix} \tag{3.1c}$$

B is the length between the midpoint of the wheels, and D_t is the distance the midpoint of the robot has travelled between time $t-1$ and t. D_t^r and D_t^l is the distance travelled by the right and left wheel and calculated based on the *tacho* readings from the motor, between time $t-1$ and t. A *tacho* is one length unit in the motor encoders, and one motor revolution is 360 tachos.

Note that the $sin()/cos()$-functions used, provided by NXC, uses degrees instead of radians. The angles are transformed to radians before they are used in the main loop `lineBeaconSLAM` in MATLAB.

Equation 3.2 [7] shows the relation between *tachos* and travelled distance.

$$D_t^{r,l} = \frac{N2}{N1}\frac{2\pi r}{360}T_t \ [m] \tag{3.2}$$

$\frac{N2}{N1}$ is the gear ratio from input cogwheel (attached to the motor) to the output cogwheel (attached to the wheel). r is wheel radius and T_t is the tacho count. Each time the motor is run by the motor control functions provided by RWTH Aachen-team [17], the tacho counter is reset such that T_t is the change in tachos between time $t-1$ and t.

This approach is known as *dead reckoning*, which is the process of calculating the position based on the change since last estimation. This approach accumulates errors fast, and will therefore need correction. The error is often caused by wheel spin, since the surface might have high friction or be uneven. This leads to that the robot wheels turn, but the robot might not move, or at least not as intended.

Another source for error is the measurement of the length B, which varies with the contact point of the wheels. The reason for this is caused by e.g. uneven surface.

The sending protocol for the position estimate was almost completely implemented, both on the NXT brick and in MATLAB. The `getPos` function in `NXTInterface` had to be extended in order to interpret the package received from the NXT-robot. The format was

```
xnnnnynnnntnnnn
```

where $[x, y, t]$ marks if it is the (x, y) position or the orientation t, while n is the raw data.

3.2.2 Feedback on sensor tower angle

The first issue to solve, was to track the tower angle. This was done by reading the tachos before and after the motor was run, and calculating the change in angle. The angle was calculated by equation 3.3.

$$\Delta\theta_t^s = \frac{N2}{N1}(T_t - T_{t-1}) \ [deg] \tag{3.3}$$

The tacho count is calculated differently here than in equation 3.2 since this angle is calculated continuously as the motor runs, unlike equation 3.2 where the position is calculated when the motor has stopped.

During the full scan, when the tower angle has reach approximately 180°, the motor is run continuously in the opposite direction until approximately 0°. This approach removed the problem with the growing offset, by keeping the tower approximately in the range (0°,180°). Further, the angle were sent back to MATLAB with the corresponding measurement, such that the correct angle was used when transforming from polar- to cartesian coordinates of the measurement. This improved the quality of the NXT-measurements significantly.

In order to send the angle back to MATLAB, the communication protocol for the `fullScan` procedure had to be extended. The new protocol is presented below.

```
s1xxxxms2xxxxmayyyyyyyyfi
```

S1/S2 - marks that the next data is measurement from either sensor 1 or sensor 2.

x - raw measurement data.

m - marks the end of a measurement.

a - marks that the next data is the angle.

y - angle data.

f - marks that the next data is the measurement number.

i - the measurement number.

3.3 Simulator

This section will discuss how the simulator of the system was made able to run with two robots.

At the starting point of this project, the simulator was able to run one robot through a few different mazes. The simulator was merged together with the main loop, separated from the robot interfaces. This presented the challenge of how the simulator should be connected to the interfaces without blending the modules. With this in mind, the simulator was updated as explained below.

3.3.1 Updating the simulator to allow collaboration.

The main issue to consider was that the simulator was developed before the software was restructured, when the robot interfaces were implemented. This meant that the simulator was completely separated from the robot interfaces. The objective of the simulator was to simulate the SLAM-algorithm, and therefore the robot interfaces was previously of no concern. However, due to the required communication with the collaborating robots, the role of the interfaces has become more significant for the main loop. This meant that the simulator needed to be connected to the interfaces in order to allow communication between two simulators.

The functions the simulator needed access to resided in `RobotHandler` and were independent of which robot that were connected. Because of this design, these functions could be called by the simulator by simply extending `RobotHandler` with few new functions. This allowed the simulator to use the interface as if it was a robot. An overview of the new functions are listed in table A.1

This allowed two simulators run the 'same' map, and collaborate on the mapping task. However, the robots were not simulated to actually be in the same map, more as if they were running in two copies of the same map. This meant that the robots did not crash and were unable to 'see' each other with the sensors. Still, the simulator were fully capable of testing the algorithms of the system.

Chapter 4

The Communication protocol

This chapter will discuss how the internet connection between the robots was resolved. The purpose of this communication was to allow the robots to communicate with each other for collaboration purposes. The internet communication would also allow the robots to be controlled from separate computers, and is therefore flexible.

The solution presented for this thesis is based on the project work by Halvorsen [19]. The solution for this project will extend the previous solution into a reliable data transfer protocol. The aim is to guarantee delivery of the data packages regardless of which data or which time it is sent.

4.1 Reliable data transfer

In order to be able to guarantee a reliable data transfer protocol, a new protocol had to be implemented on top of the TCP/IP protocol. Although the TCP/IP protocol is a reliable protocol, messages could be lost when arriving in MATLAB if they were not interpreted correctly.

When the reliable data transfer protocol was implemented, a more flexible communication could be achieved.

4.1.1 New data transfer protocol

The creation of a new data transfer protocol involved a modification of the `communicate`-protocol, which was implemented for the project work of Halvorsen [19]. This protocol sent data packages formatted as strings, which was required by the read/write functions provided by the MATLAB-toolbox Instrument Control Toolbox. The new protocol extended the data string to include a header. This header held information regarding which type of data was sent, whether it was a vector or a matrix, together with symbols marking the beginning and the end of the packages. The format is shown below.

```
:dataTypeHeader!datastring,
```

The sign ":" marks the start of the header and the package, the sign "!" marks the end of the header, and the "," marks the end of the package.

The main reason for package loss previously was caused by receiving packages on the wrong format, e.g. receiving incomplete packages, which generated various errors in MATLAB. With the new protocol the packages was temporarily stored in a `persistent`-variable until the termination sign was read. The package was then returned by the `read`-function of the `commuication`-module. With this protocol all packages was received regardless if the data in the input buffer contained partial or several packages. Hence, the data transfer could now be regarded as reliable.

Further, with the new header containing the data type, the received packages could easily be interpreted, and stored correctly in the *navData*-data structure for further use. This made it easy to send various data at the same time, which made the communication more flexible.

With this new communication protocol, the TCP/IP connection was set up before the main loop. The SERVER/CLIENT-structure still applied, but once the TCP/IP connection was set up, the network role had no impact on the direction the data was sent.

4.2 Integrating the communication module

In order to use the communication protocol with the main loop, a new function was implemented in `RobotHandler` called `exchangeData`. This function was responsible for sending and reading data, and storing the received data in a data structure such that it could be used

throughout the main loop. This method required that all data that was sent had defined data-types, regardless if it was a scalar, vector or matrix. The different data types are listed below.

- `'connected'`
- `'pose'`
- `'nextpos'`
- `'map'`
- `'beacons'`
- `'finished'`
- `'initoffset'`

With this protocol, messages could be read when they were available and stored safely. To assure the newest update was received, the input buffer was checked regularly throughout the main loop, often right before data from the second robot was used. This way the robots could use the latest updates when navigating and travelling through the workspace.

The structure of the robot-to-robot data flow is shown in figure 4.1. The main loop `LineBeaconSLAM` requests various data, while `RobotHandler` handles the request for the current robot. The actual sending and receiving is performed in `communicate`. With the design of the communication module, packages could be sent from anywhere

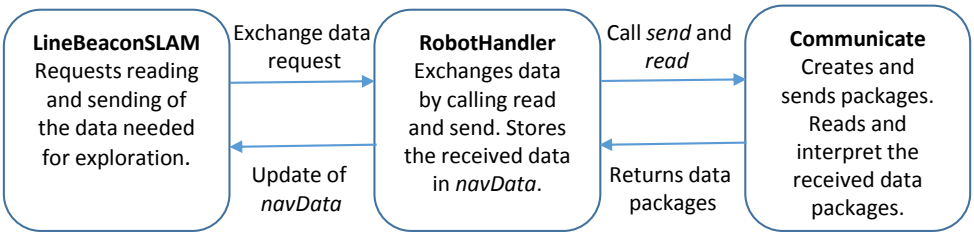


Figure 4.1: Data flow for data sent and received by the internet communication. All data exchanged is stored in *navData*, which is used in the main loop.

in the software since the `communicate`-function holds the connection handle, and handles all requests regarding the communication-protocol.

Chapter 5

The Navigation algorithm

In this chapter, a new navigation algorithm will be designed and implemented. First, a literature study will be conducted and review several approaches to the navigation problem with multiple robots. Then a new algorithm will be developed based on this review. This algorithm will replace the 'Left-wall-follower to backtracker'- algorithm, which has been used previously. The main reason for replacing this algorithm is that both the navigation- and coordination problem will be solved by one exploration-algorithm. The approach of this algorithm is more flexible, and can be easily extended to include more robots. The second reason is that the new exploration-algorithm is more suited for multiple robots.

Before a new navigation-algorithm can be designed, the multi-robot navigation problem has to be formulated.

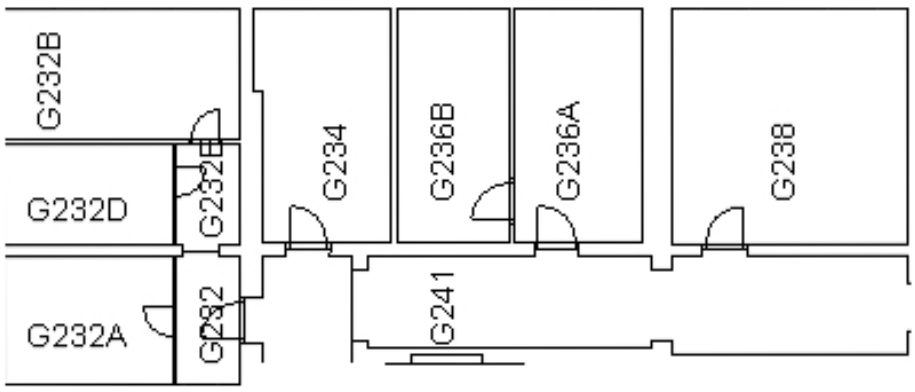


Figure 5.1: This floor plan is a typical example of an indoor environment suited for this algorithm.

5.1 The multi-robot navigation problem

Multiple robots are potentially able to explore an unknown area faster than a single robot [18]. By using optimization, several good approaches has been developed. However, it is not possible to find a globally optimal search path before the entire map is explored, hence only locally optimal decisions can be made at each step [11]. The challenge lies in designing an algorithm, which can as efficiently as possible explore an area, and minimize the costs defined for the cost function. The costs can be, travelling distance, time, information gain and so forth.

This algorithm will be designed for 2D indoor environments, where the landscape is either open, or consists of corridors and rooms. Figure 5.1 shows a typical workspace where this algorithm would be suited.

5.2 Research on multi-robot exploration

There are many articles written on this subject, and this section will review different ideas, which will form the foundation for the algorithm used in this thesis.

Yamauchi 1998 - Frontier-based Exploration Using Multiple Robots

This article [18] is referenced in almost all of the articles below, and presents a fundamental base for a solution of the navigation problem.

The central idea behind Frontier-based exploration is to gain as much new information as possible at each step. An occupancy grid is used to keep track of the discovered area. The method classifies discovered cells adjacent to undiscovered cells in the occupancy grid as frontier-points. Adjacent frontier-points are then grouped into frontier-regions, which is assigned to- and visited by the robots.

The system presented in this article is decentralized, with asynchronous communication, where each robot has a global map, which is broadcast to the other robots. The maps are then fused together by adding two probabilistic maps together, which shows the probability for a cell in the grid to be occupied. In addition, this system is fault tolerant since if a robot stops working, the remaining robots can continue the search without problems.

Burgard et al.

The approach described in [5],[6] is based on the approach of Yamauchi [18], and extends this method in order to coordinate the robots. The coordination is solved by calculating the utility of a target point, in addition to the travel costs. The utility of a target point is given by the information gain at this point, and reduced for the other robots when it is assigned. The target point is chosen based on a trade-off between the travel cost and utility. This way, the robots will choose different target points, and disperse better throughout the workspace.

Quottrup et al.

The method used in this article [12] is to build a planar grid, which defines the feasible paths for the robots. They can move forward, backward, or sideways. This approach has well defined possibilities for the direction, but is primitive, and for a smaller workspaces, the difficulty of coordination increases.

Méndez-Polanco et al.

The exploration in this article [9] is based on that a server assigns regions to the autonomous robot team. This is a centralized system, which is not desirable when designing a completely autonomous system.

MAGIC 2010 Robot Challenge

Three articles [8], [3], [4] have been reviewed, which was used by team MAGICian for the MAGIC 2010 robot challenge. The exploration is based on a flood-fill with ray casting approach. This approach finds frontier points, and defines frontier regions, as described in Yamauchi [18]. Then an information gain is calculated for each region by using ray casting. This method casts rays in a radius from the center of each frontier region, and counts number of unexplored pixels. Then an estimate of the information gain is calculated. Further, a new ray casting is performed in order to check the position of the other robots, before deciding on a new position.

A few more improvements were applied to the algorithm, such as favoring the forward direction and assigning bounding regions, which defined areas the robots could explore.

A tactical planner, with help from an influence map, is used to assist the navigation. Each robot then calculates a global path using A*/D* Lite algorithm, while a mid-level planner modifies the path in real-time using elastic bands.

Finally, a trajectory planner, which modifies the immediate waypoints and trajectory commands, based on Dynamic Window approach, avoids collisions.

However, team MAGICian placed 4th, but their algorithms were similar to the winners, team Michigan, which is documented in [13], [10]. These papers were more directed towards the competition, which allowed human interaction. They realized that making an autonomous

system with human interaction was rather difficult, and focuses somewhat on that aspect.

D. Puig et al.

The approach described in this article [11] divides the workspace into regions, assigns the regions to the robots, and coordinates the robots in order to reduce completion time. The cost function minimizes the variance of the average waiting time for each region, and the variance of the travel cost. In other words, this system focuses on distributing the workload as even as possible in order to reduce the overall costs, as e.g. time.

Summary

The most important issues mentioned by these articles are listed below. These terms will become in focus for the algorithm in this thesis.

- Frontier regions.
- Track the movement of the robots with an occupancy grid.
- Information gain.
- Dispersing of the robots across the workspace.
- Assigning regions to the robots.

5.3 Outline of the algorithm

The design of this algorithm aims to solve both the navigation- and coordination problem in one algorithm. This will be done by formulating the exploration as an optimization problem.

First, the algorithm will build a tactical map based on the common map and the robot positions. The tactical map will track how well the robots has covered an area, and accounts for overlapping of the robots.

Second, the discovered area is computed based on the robot paths and the reliable range of the sensors, constrained by the wall segments. Further, the border of the discovered area is found. The feasible border points are the frontiers, and therefore candidates for the next destination.

If all border points of the discovered area are adjacent to wall segments, the search is done, since then no possible next destinations exists.

If the search is not terminated, the optimizing part will find the frontier, which minimizes the cost function. This frontier will then become the new target destination. The pseudo code for the algorithm is presented in 5.1.

The exploration algorithm is based on the Frontier-based approach of Yamauchi [18], and the tactical map is inspired from the work of team Magician [8], [3], [4] of the Magic 2010 Robot challenge.

This exploration algorithm calculates new positions without considering if the path along the straight line from the current to the

next position is collision free. For this solution to be complete, a path planner is needed. This is discussed in chapter 6.

Algorithm 5.1 Outline of the exploration algorithm

Input: Merged map, robot paths.
1. Build tactical map based on SLAM map and the robot paths.
2. Find the discovered area and its border.
3. Determine the frontier points.
4. Optimize: Assign costs to all frontiers and choose the frontier point which minimizes the cost function.

Output: New position.

5.4 Developing the Navigation algorithm

In this section, each step of algorithm 5.1 will be explained thoroughly. First, the algorithm will be developed for one robot, and afterwards extended to include the second robot.

5.4.1 The Tactical Map

The purpose of the tactical map is to track how well an area is covered. This information is used in the cost function when calculating new destination points for the robots. It is desirable to find new destination points which can give as much new information as possible. The tactical map will give a measure of the *information gain* at each point. By summing the function values in some neighbourhood of the frontier point, an estimate of how well this point is covered can be

calculated. It is therefore desirable to find the frontier point with the lowest sum, which will give the highest information gain if visited.

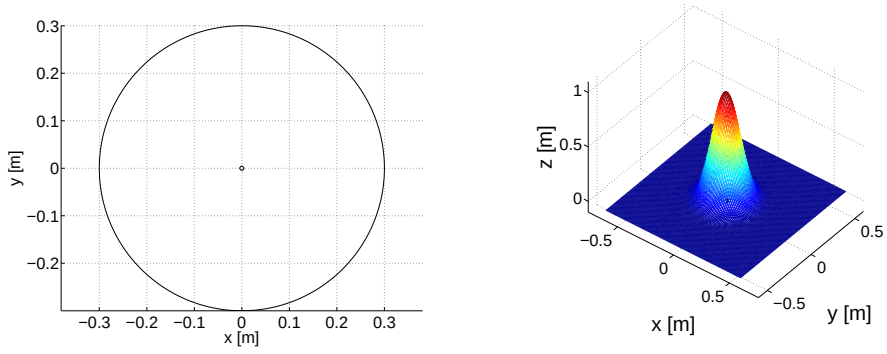
The tactical map is represented by a multivariate normal distribution. This representation is considered to be fair, since it is assumed that the robot position is very likely to have been discovered, but less likely further away from this point. The range of the robot is decided by the reliable range r of the sensors, hence the line of sight of the robots.

The tactical map is based on the robot paths, and is represented by a grid with resolution in cm. Each robot position is considered as an expectation value for the normal distribution, with a standard deviation proportional to the line of sight parameter r. The mathematical representation of the tactical map is presented in equation 5.1.

$$T = \sum_{j=1}^{N} \alpha e^{-(\frac{(x-x_{c_j})^2}{2\sigma^2}+\frac{(y-y_{c_j})^2}{2\sigma^2})} \tag{5.1}$$

where α is a scaling factor, N is the number of robot positions in the path, (x_{c_j}, y_{c_j}) denotes robot position j and the expectation values, and σ is the standard deviation. σ is the same for both x and y since the distribution is wanted to be circular, and not elliptical. r is defined to be 3σ and marks the edge of the distribution at that point, since about 99,7 % of the volume lies inside three standard deviations of a normal probability distribution.

The size of the grid is decided by the greatest and smallest (x, y) values of the positions, and extended by the line of sight parameter r,



(a) (b)

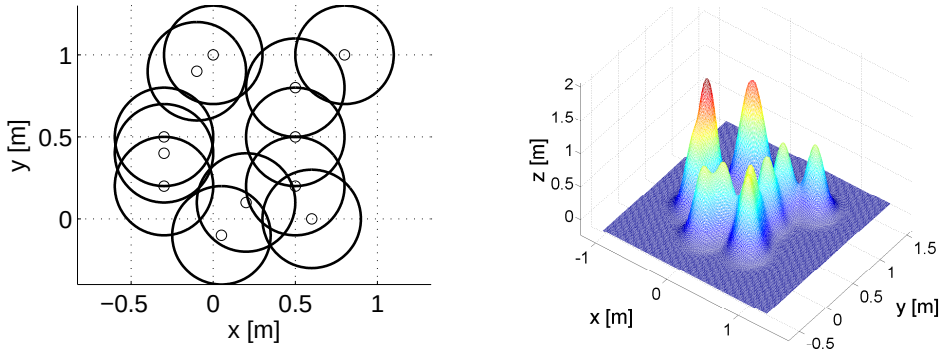
Figure 5.2: 5.2(a) shows the circle with centre in robot position (x_j, y_j), radius r. Figure 5.2(b) shows when a normal distribution is added to the robot position with $\sigma = \frac{r}{3}$ and $\mu = (x_j, y_j)$.

as shown in 5.2.

$$Grid\ corner\ points = \begin{bmatrix} x_{min} - 2r, & y_{min} - 2r \\ x_{min} - 2r, & y_{max} + 2r \\ x_{max} + 2r, & y_{max} + 2r \\ x_{max} + 2r, & y_{min} - 2r \end{bmatrix} \tag{5.2}$$

This makes the grid dynamical, which is important since the area is unknown. It is possible to assume a fixed size for the area, but for this project, it is considered to be undesirable. An example of the procedure of how the map is built is shown in figure 5.2.

This kind of representation will resemble a mountain scenery, which is shown in figure 5.3. From this map it can easily be found points which are local minimas, and therefore points of interest when choosing



(a) (b)

Figure 5.3: The robot paths in 5.3(a) will generate the tactical map displayed in 5.3(b)

new destination points during navigation.

5.4.2 The border of the discovered area

After defining the tactical map, the next step is to find the discovered area and its border. The border points will be candidates for frontiers, and therefore the new positions of the robot. By going to points on the border to an unknown area, the extension of the discovered area can be maximized, which will lead to a more efficient exploration. The role of the discovered area at this step is to find the border of it, but later this bitmap is used for the path planning algorithm.

When finding the discovered area the wall segments had to be taken into account, such that the workspace was found as the set of points,

which were visible from the robot positions.

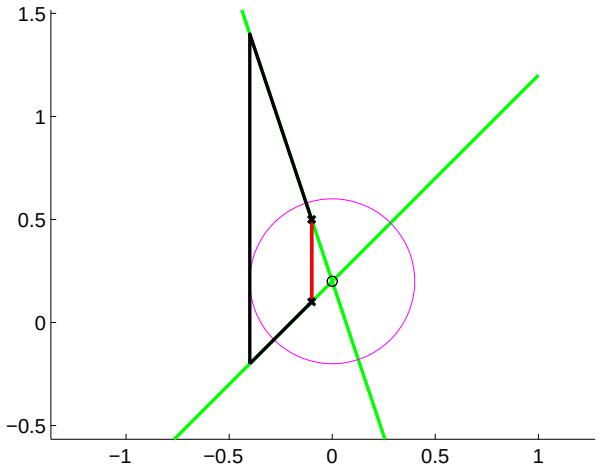
Calculating the discovered area was a challenging task, since it could be difficult to define which areas was reachable or not. The same area could be considered as both reachable and unreachable, depending on the viewpoint.

To derive a method to find the discovered area, one point and one wall segment was considered first. The fundamental idea was that points behind the wall segments was unreachable. The unreachable area behind the wall segment was represented as a polygon with sides along the lines from the robot through the end points of the wall segment, and the range of the robot sensors. If a point was found to be inside or on the polygon, it was considered as unreachable and given a negative weight.

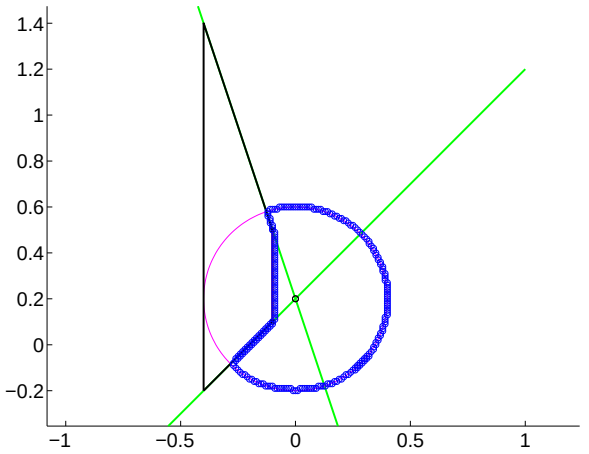
One polygon was created for each wall segment in range of the current robot position, which defined the unreachable area for this point.

When all the robot positions had been considered, the negative weighted points were defined as unreachable, and positive weighted points were defined as reachable. The process of finding the unreachable area is illustrated in figure 5.4, and the procedure is shown in algorithm 5.2.

After the procedure in 5.2, the map was transformed to an occupancy grid where 0 marked an unknown cell, and 1 marked a known cell. Then, the MATLAB function `edge` was used to find the border of the discovered area. The border was on the format $border = [x, y]$,



(a) Based on the lines (green), the wall segment (red) and the line of sight circle (magenta), the polygon (black + wall segment) is created.



(b) All points inside the overlapping area between the circle and polygon (including the polygon edges) are marked unreachable, the remaining points within the circle are marked reachable. The resulting discovered area has the border points marked with blue circles.

Figure 5.4: A step by step generation of the border for one point and one segment.

Algorithm 5.2 Finding the discovered area

```
for each robot position do
    Find possibly conflicting wall segment within range r
    if No conflicting wall segments then
        Add a positive weight to the entire range
    else
        for each possibly conflicting segment do
            Create all polygons which defines the unreachable area for
            the current robot position
        end for
        for each point in range r of the current robot position do
            Check if the current point lies in the unreachable area
            if point is in or on the unreachable area then
                Add a negative weight to this point
                break
            end if
            if not then
                Add a positive weight to this point
            end if
        end for
    end if
end for
```

with values sorted on x- and y-values.

The importance of the correctness of this map was crucial. This border was the set of possible next destinations, and if an infeasible border point was created, the robot could collide. Moreover, in order for the end criterion to work properly, the border had to be correctly fitted to wall segments where this was the case.

The most important problem was that different viewpoints gave different answers to if a point was unreachable or not. Two robot positions on each side of a corner illustrates this issue, and is shown in figure 5.5. What is unreachable for the first point is reachable for the second point, and opposite. This unfortunate case would have to be treated in the post-processing step of the border, when the frontier points is found.

5.4.3 Determining the Frontier points

After calculating the border of the discovered area, these points were considered for frontier points. First, all points close to wall segments were removed to avoid collisions with the wall segments. Moreover, the end criterion depended on that points fitted to a wall segment were removed in order to know when the exploration was done. This procedure would then divide the original border into border segments, which had to be identified.

The identification process of the border segments was done with a window scan. When a point from the border was found, the border was tracked both ways (if possible). All border segments over a certain

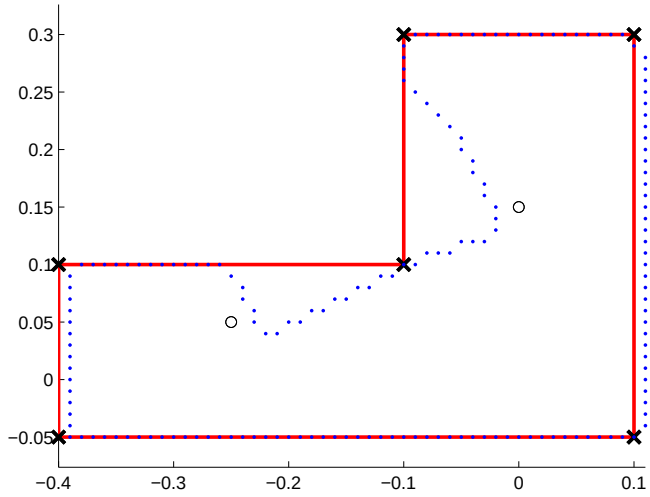


Figure 5.5: When finding the discovered area, this ambiguous case often occur. The border bulges around the corner, since what is in the blind spot for one point is not for the other, and opposite.

size was stored in a cell structure. This method also sorted the border points in a chronological order, which was needed for the next step, when the border segments was merged. The algorithm is shown in 5.3.

This procedure also solved the problem illustrated in figure 5.5. Since the bulges around the corners are close to the walls, they will be removed if they are within some range of the wall. Further, if parts of the bulges remains after the first step, they are filtered out at the next step since these border segments would be too small.

Algorithm 5.3 Identifying and sorting the border segments

```
Transform border from [x, y] vector, to a (i, j) bitmap.
for all i do
  for all j do
    if border point found then
      Mark start point
      With a window scan, trace the border
      if end of border found then
        Go back to start, or store border if starting point has al-
        ready been visited.
      end if
    end if
  end for
end for
```

After algorithm 5.3, the border consists of several sorted border segments. Some of these segments may belong to each other. Therefore, a merging procedure follows algorithm 5.3. This algorithm checks if start or end points of the border segments are sufficiently close to each other. If they are, the segments are merged together.

The output of this border post processing is a set of frontier points. In the next phase, a cost function will be defined, and all the frontier points will be assigned a cost, which will in turn determine the next point to visit.

5.4.4 Optimization and the Cost function

This section will define the cost function for the optimization problem, and compute the next position for the robot. First, the costs have to be defined.

Travelling cost It was desirable to minimize the overall travelling distance of the robots, since the system accumulates positioning errors over time, and to exploit most of the batteries. By adding travel distance as a cost, the unexplored areas close by would be favored over areas farther away. The cost is defined as the distance travelled, scaled in order to match the information gain.

$$d(\bar{d}_d) = \kappa \cdot \bar{d}_d \tag{5.3}$$

κ is the scaling factor and $\bar{d}_d$ is the distance in $[m]$ from the current robot position to the current border point.

Information gain It is desired to gain as much new information as possible at each step. The goal is to maximize the expansion of the discovered area at each step in order to have an efficient navigation. By summing all function values of the tactical map T in a neighbourhood

of each frontier, a measure of how much new information it is possible to obtain from a frontier is estimated. The cost is shown below.

$$g(\bar{x}_i, \bar{y}_i) = \sum_{k,l} T(x_k, y_l), \quad s.t\ (\bar{x}_i - x_k)^2 + (\bar{y}_i - y_l)^2 \leq R^2 \tag{5.4}$$

(x_k, y_l) is the index representation of a metric (x, y) point in the tactical map T. $(\bar{x}_i, \bar{y}_i)$ is the current frontier (index) point in T. R marks the radius of the neighbourhood from the current frontier.

Close to wall segment Points close to a wall segment will get its view obscured by the wall, and it is desirable to lead the robot into open spaces. The cost is defined as

$$w(d_w) = \frac{1}{d_w^2} \tag{5.5}$$

where d_w is the distance from a point to the wall segment.

The cost function will then be the sum of the costs listed above, for each frontier point.

$$f = \sum_{i=1}^{m} d_i + g_i + w_i \tag{5.6}$$

where d is the travelling distance, g is the information gain, w is the distance from a point to a wall segment, i is the number of current frontier and represents a (x, y) point, and m is the number of frontiers

to be considered.

The cost function is built and minimized simultaneously, by storing the lowest sum which is calculated, and the corresponding frontier point. This way, when all frontier points has been considered, the global minimum is found. If there were multiple minima with similar cost function values, the algorithm would choose the first minimum point it considered.

An example is shown in figure 5.7. For this example, figure 5.6 shows the current map of the exploration. For each frontier point, the cost function has calculated the individual costs, as shown in figure 5.7(a), which when summed becomes the cost function shown in figure 5.7(b). Finally, the minimum of this function is found, and the position is returned by the navigation algorithm.

The placement of the new target position shows the effect of each cost. First, the travel cost prefers the frontiers at the shortest range of the robot. Second, the wall cost forces the point to the right, away from the wall. Finally, the information gain makes sure that the target point will not overlap too much with an already visited area, and limits how far to the right the target point can be.

How much the point is shifted away from the wall and the already discovered area depends on the relative weighting of the costs, which have to be considered carefully. It is desirable to lead the robot into open spaces, but also to maximize the expansion of the discovered area. Although the information cost increases the efficiency of the algorithm, the wall cost is a preventer of collisions. Therefore, these

two costs must be balanced properly with the travel distance, which is the most important cost.

Finally, the navigation algorithm was tested in the simulator, as explained in chapter 9.1.

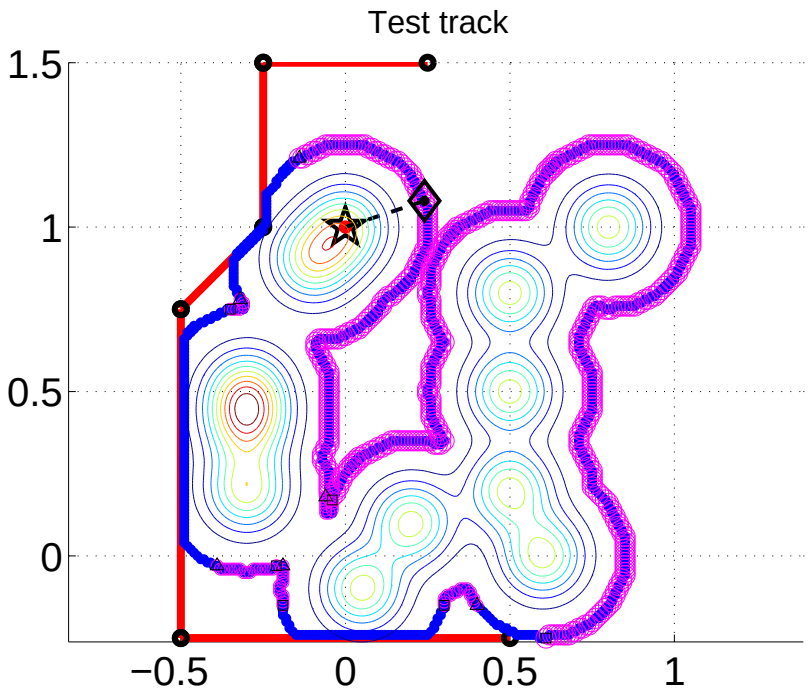
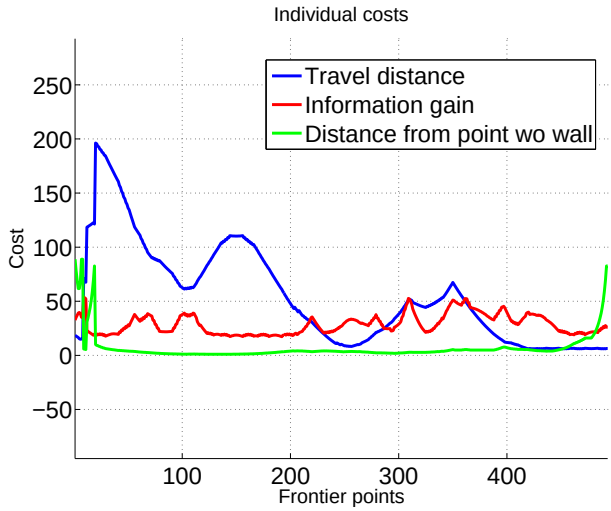
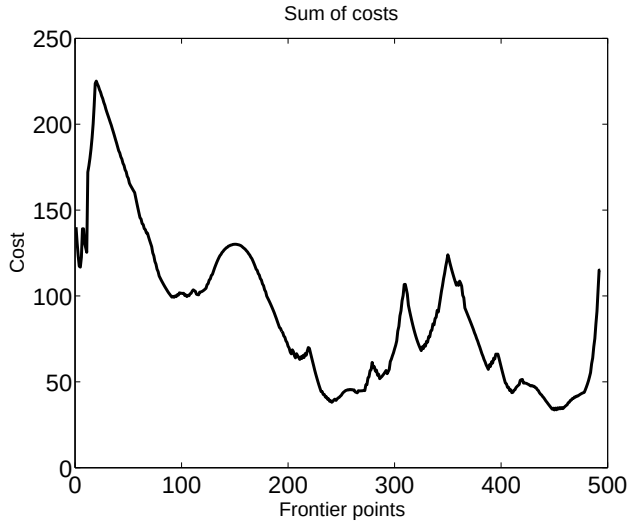


Figure 5.6: The red segments are the wall segments, the border is marked in blue, and the frontier points are marked with pink circles. The tactical map is represented by a `contour`-plot, and also shows approximately the robot path. The star marks the current position while the square marks the new position the algorithm has calculated.



(a)



(b)

Figure 5.7: An example of the exploration algorithm. The track is shown in figure 5.6, while the individual costs are shown in figure 5.7(a), and the final cost function is shown in figure 5.7(b).

Chapter 6

Path planning

This chapter will discuss how the path planning problem was solved for this thesis. From the navigation algorithm the robot will receive a destination point, but there is no guarantee that the path along the line between the current position and the next position would be collision free. This chapter aims to solve this problem.

6.1 The path planning problem

The general path planning problem is to travel from A to B without any collisions. For this project, the problem is to find a path in 2D which avoids collisions with obstacles in the work space. The workspace for the path planning algorithm is defined to be a subspace of the discovered area called the *free configuration space*.

There is one basic requirement for the path planning algorithm,

which is to avoid collision with wall segments. Avoiding collision with wall segments will imply finding a path in a static environment. A second basic requirement could also be considered, which is to plan a path which avoids the other robot. This would be a more difficult part, since this robot will be acting as a dynamical obstacle.

Furthermore, it is required that the number of intermediate waypoints is low in order to avoid a jagged path, allowing smooth driving. This is due to the driving method of the robots, which is composed by sequential turning then driving. By making the robot follow a path consisting of a significant amount of waypoints, the robot will make many turns and drive short distances, leading to inefficient driving and fast growth of the positioning error.

In order to make the path planning dynamical during runtime, the robots will recalculate their path at each waypoint and check if the path is still collision free.

6.2 Outline of the algorithm

The exploration-algorithm will in most cases return a point at a distance close by, with no obstacles interfering with the path. It is when the robot comes to the end of a search path, and wants to explore a region across the workspace, that it needs path planning.

An example is when the robots is done exploring a room, and wants to go out into the corridor and explore the next room. For this reason, the first part of the path planning algorithm will check if there is need

to plan a path, by checking for collisions with obstacles. If the path is free, the robot can proceed with no further planning.

The first step of the path planning algorithm is to create a graph, which represents the free configuration space. Based on this bitmap, a breadth-first search (BFS) is conducted. This algorithm finds the shortest path in the free configuration space. Then the path is post-processes such that only the most necessary waypoints from the BFS-algorithm is included.

The outline of the algorithm is shown in algorithm 6.1. The next section will explain each line of this algorithm into more detail.

Algorithm 6.1 Outline of the Path Planning algorithm

```
1: if not collision then
2:     return
3: end if
4: G ← the free configuration space
5: BFS_Path = BFS(G,v,w)
6: Calculate the relative Angles between each edge in BFS_Path
7: if relative Angles > threshold then
8:     add way-point to path
9: end if
```

6.3 Developing the path planning algorithm

6.3.1 When to use path planning

The robot will for most cases travel relative short distances, and therefore not need path planning. In order to reduce run time and the number of scans, it is desirable to travel along the line from the current pose to the next pose, when it is possible.

During 'normal' navigation, the robot will usually drive a distance equal to the range of the sensors, such that the next position will be in sight.

In order to check if path planning is needed, the line between the current position and the next position is checked for intersection with the wall segments. It must also check if the path is too close to a wall segment, for example to avoid cutting corners.

When the physical system is run, the SLAM-algorithm generates significantly more beacons than the simulator. Therefore, it is important to check if the path is close to a beacon. Moving close to a beacon will often result in a collision, and must therefore be avoided.

If one of these tests show that a collision will or may occur, then path planning is needed.

6.3.2 Configuration space

The first step of the path planning algorithm is to find the feasible area the robot can travel in. This is the free configuration space. The free configuration space is the set of all configurations the robot can be

in, without being in conflict with any obstacles. The possible configurations are $[x, y, \theta]$. For this system the free configuration space would be a 2D map with a (x, y) metric representation. At each position the robot has the orientation θ, but for this system $\theta \in [0, 360] \; \forall \; (x, y)$. The system is *holonomic*, and the orientation can therefore be disregarded. Hence, the free configuration space is a subset of the discovered area.

By restricting the movements of the robot to the free configuration space, it can be guaranteed that the robot path would be feasible. When travelling in the free configuration space, all possible hazards are removed prior to the path planning, ensuring that all paths will avoid collisions. From the free configuration space it can also be determined if a feasible path exists.

The free configuration space is found by removing all points from the map showing the discovered area, which is at some range from the wall segments. In addition, all points around the other robot is removed in order to avoid that the robots cross paths. This process would remove narrow openings, which could block the robot. An example is shown in figure 6.1.

When the free configuration space is found, the robot can be treated as a point.

6.3.3 Finding a path

The procedure to find the path will consist of two stages. First, the shortest path will be found by a shortest path algorithm, then this path

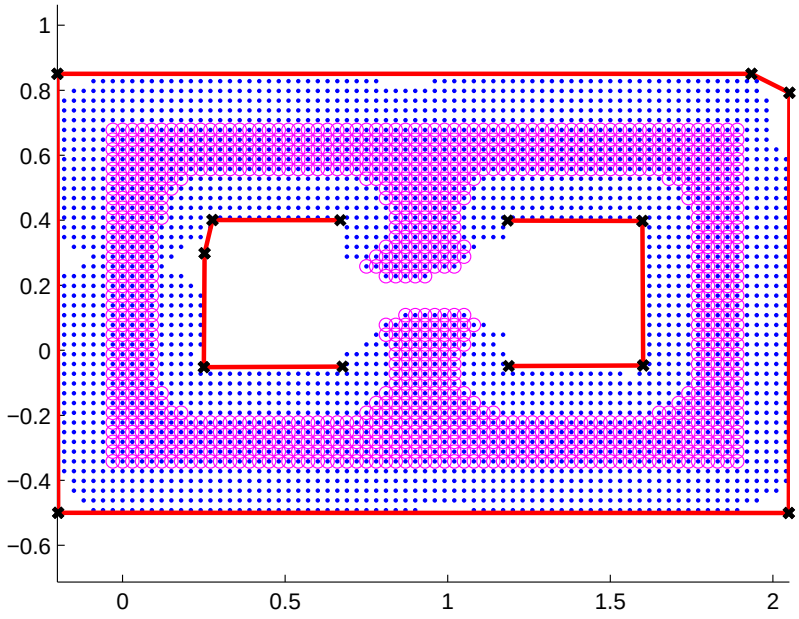


Figure 6.1: Example of a configuration space. The blue dots marks the discovered area, while the configuration space is marked with pink circles.

will be smoothed such that only a reduced amount of intermediate way-points will be used. The smoothing algorithm will try to fit as many points as possible to a line, in order to avoid zigzag driving for the robots.

In order to choose the shortest path algorithm, the shortest path problem was defined first.

Given an undirected and unweighed graph G, start node v and target node w, find the shortest path from v to w in G.

For this system, the graph G is the free configuration space, the start node v is the current position, and the target node w is the next position for the robot.

Further, it is desirable to find an optimal path which produces the best foundation for the second stage. For example, if the algorithm tries to follow the straight line towards the goal when possible, the path would not be optimal. The robot would approach the obstacle head on, circumnavigate it, and continue towards the target when it finds the straight line again. Such algorithms would try to minimize the distance to the goal at each step.

A number of shortest path algorithms were considered such as the A*-, Bellman-Ford- and Dijkstra-algorithm, but for the shortest path problem as described above, the breadth-first search (BFS) algorithm was found to be most suitable. The BFS is actually a special case of the Dijkstra algorithm, but without costs/weights on the edges. The

BFS-algorithm would find a path based on the distance in the terms of number of nodes between the start and goal node. An example is shown in figure 6.2.

As can be seen from figure 6.2, the path is not optimal. When the BFS-algorithm searches in the tree, it looks at the neighbours to the current node, which has not yet been marked, and adds itself as the parent to the neighbours. When the goal node is found, the path is calculated backwards by adding the parent of each node to the path. This way, the number of nodes is minimized, but the metric distance of the path might not be. This is due to the order the nodes are visited in. In the direction of the parent, there are usually three possible parent nodes (due to the grid structure), which are at the same distance from the start node, and therefore they will seem equal. This problem can be solved by adding a cost to diagonal movement, such that a more optimal path will be found.

When the shortest path is found, the second stage will try to pick out only the necessary waypoints from the BFS-algorithm. This is done by looking at the relative angles between the edges of the BFS-path.

If the orientation θ is the same when travelling between several waypoints, all of them would not be needed. The necessary way-points would be points where the robot has to turn. Therefore, by setting a tolerance for deviation from the previous orientation θ_{t-1}, the points which could be fitted to the same line with a small quadratic error, can be grouped together. Due to the 8-way direction behaviour of

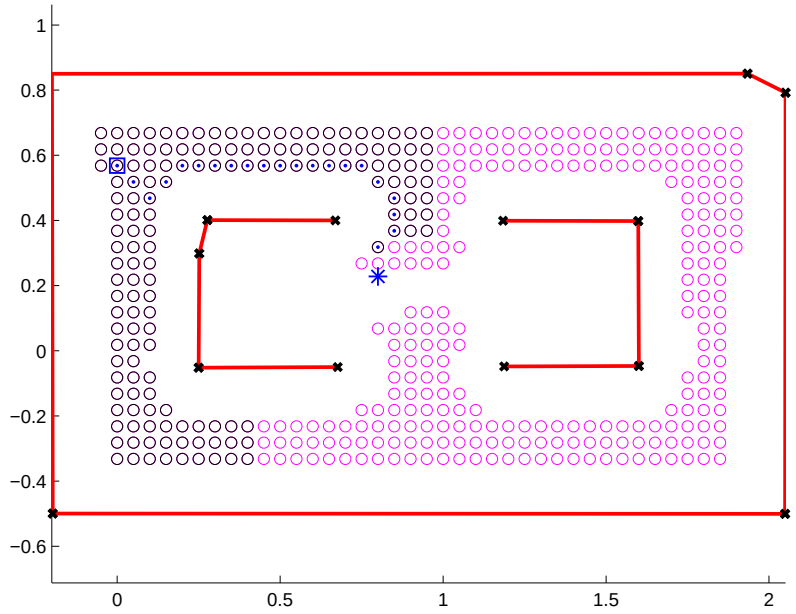


Figure 6.2: Example of a path found by the BFS algorithm. The robot starts at the point marked by the blue square, and wants to go to the position marked by the blue star. The circles marks the graph G, and the black circles marks the investigated nodes of the BFS-algorithm. The path found are marked with blue dots.

the grid, turning points are easily discovered. The smallest turn in the grid would be 45 °, which would be a reasonable threshold for the turning points.

An example of these relative angle differences is shown in figure 6.3. The points picked out for the path is marked with a cross. From this analysis, the waypoints for the path is picked out as shown in figure 6.4. As can be seen from the figure, only the turning points are included, which means that the number of way-points is reduced significantly.

As can be seen from figure 6.4, some of the waypoints could be merged together by averaging, but this will raise the question if that new path is still inside the configuration space.

With waypoints as in figure 6.4, the robot will follow the path found from the BFS-algorithm, which guaranteed feasibility. However, since the path planner recalculates the path at each waypoint, and if the robot can reach the destination without going via the last waypoints, it does. This means that the number of waypoints in the planned path is 'worst case', and that the robot might arrive the new destination by a shorter path.

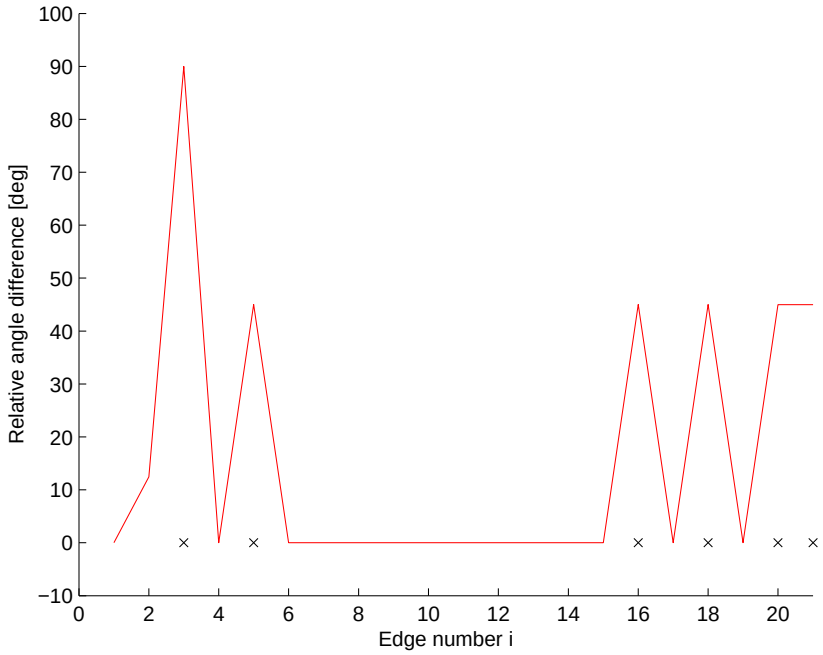


Figure 6.3: The relative angle differences between edge i and $i+1$. The waypoints picked out for the path is marked with a cross.

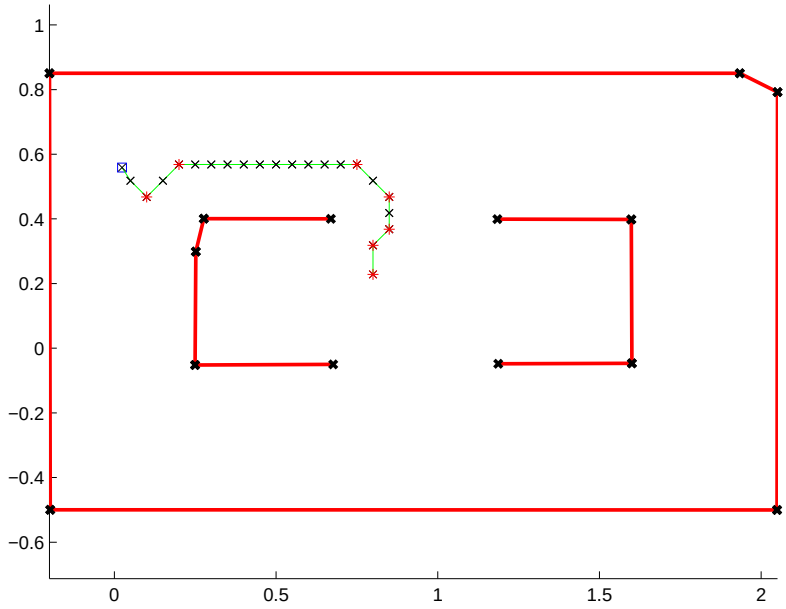


Figure 6.4: The path produced by the path planner. The BFS-path is marked with crosses, while the waypoints picked out are marked with red. The robot starts at the point marked by the blue square.

Chapter 7

Coordination of the robots

This chapter will discuss how the two robots were coordinated in order to fully exploit the use of two robots. While the vision of the navigation- and the path planning algorithm have aimed beyond the capabilities of the physical system, the work performed in this chapter focuses on this system in order to make it work.

7.1 From one to two robots

To use the navigation- and path planner algorithm alone would not be sufficient in order to avoid collisions. Moreover, it is difficult to achieve an efficient system without coordination since the robots tends to converge towards the same path after a while. This effect is caused by that the robots are forced to choose target points close to each other towards the end of the exploration, due to lack of options. Moreover,

when the robots choose points close to each other, they will find new frontiers close to each other and continue along the same path. This must be avoided by ensuring that the robots choose different target points.

In summary, the main issues with the coordination of the robots is to avoid collisions with each other, and to make them disperse across the workspace by choosing different target points.

In order to accomplish this, the robots would need knowledge about where the other robot is planning to go next. Due to this requirement, it was decided that one of the robots would act as the leader of the group, and calculate new positions for both. Since the SERVER/-CLIENT structure already was implemented, this was exploited in order to determine the role of the robots. This structure would make the system centralized, which would compromise the desired autonomous nature of the system.

However, this would only be a problem if the robots got out of range with each other. Nevertheless, the problem of network range did not apply to this system, since the workspace was quite small and the communication was performed internally on one computer.

Anyway, the system would be implemented as autonomous as possible, and with the network range problem in mind, the SERVER/-CLIENT structure was implemented such that the system could tolerate that one was disconnected. If this occurred, the remaining robot would explore as if it was alone, only considering the last known position of the other robot as an obstacle. If the connection loss were

due to the network range, it would still be safe for both robots to act as if they were alone, since they would be far from each other. If the robots eventually appears in range again, they would exchange the data, choose a leader and continue the search together.

7.2 Implementation of Coordination mechanisms

When including the second robot into the exploration algorithm a few extensions had to be done. First, a SERVER/CLIENT structure had to be implemented in the main loop. Before the main loop started the SERVER-robot was picked, which calculated the new positions. The SERVER calculated its own target destination first, and used this when calculating for the CLIENT. Then this information was sent to the CLIENT.

Further, some flags which marked if the robots were done had to be sent. This way could the SERVER notify the CLIENT if the SERVER was done, or if the CLIENT was done. If one of the robots were done before the other, the remaining robot would act as it was alone, and continue the search.

In order for the SERVER-robot to coordinate the robots, a few extension to the navigation- and path planning algorithm had to be made, which are explained below.

Defining frontier regions

An important factor for efficiency and a precaution for collision avoidance was to ensure that the robots did not choose the same target points. In order to accomplish this, a frontier region was defined around the target point, which was calculated first. This way it was avoided that the target position of the CLIENT was calculated to be in the immediate neighbourhood of the SERVER's target destination. Hence, the robots would choose target points at different places in the workspace.

Dispersion

In addition to choosing different target points, it was important to make the robots disperse across the workspace. This was done by adding a new cost to the cost function, which punished frontier points close to the target point of the other robot. The cost was defined as

$$c(\bar{d}_c) = \kappa_c \frac{1}{\bar{d}_c^2} \tag{7.1}$$

where $\bar{d}_c$ is the distance from a frontier point to the target destination of the other robot, and κ_c is a scaling factor.

This lead to that the robot always wanted to increase the distance between them, and therefore dispersed across the workspace.

Accounting for the second robot in the path planner

Although the robots choose different target points and disperse across the workspace, the robots may want to cross paths. A typical case is if the robots approach the same corner from different directions. When planning a path, a neighbourhood of the current position of the other robot is removed from the configuration space, such that the other robot would block the hallway. This will, e.g. for the corner case (where the robots approach a corner from different directions), force the robot who wants to go past the other robot, to go the other way.

These coordination mechanisms have proved to be sufficient in order for the robots to collaborate in a good manner. The complete exploration algorithm is tested in chapter 9.3.

Chapter 8

Implementing the physical robot system

This chapter will discuss how the physical system was implemented with the algorithm for collaboration developed through chapters 5-7. The physical robot system introduces significant uncertainties such as positioning- and measurement errors, as well as the distance between the robots will change with the positioning error, and produce uncertainties for the resulting map.

With these uncertainties, the map produced by the SLAM-algorithm will not be an exact mapping of the true environment, and this must be accounted for throughout the exploration algorithm. The first phenomenon to consider is that a scan with much more uncertainties produces significantly more beacons, and less wall segments. This is a challenge and have to be considered in order to be able to map an

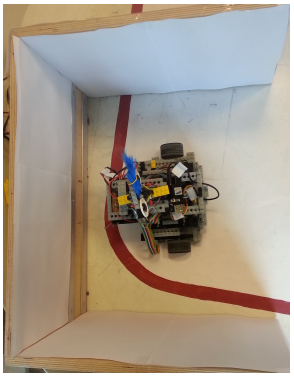
area without colliding with beacons.

8.1 Exploring with beacons

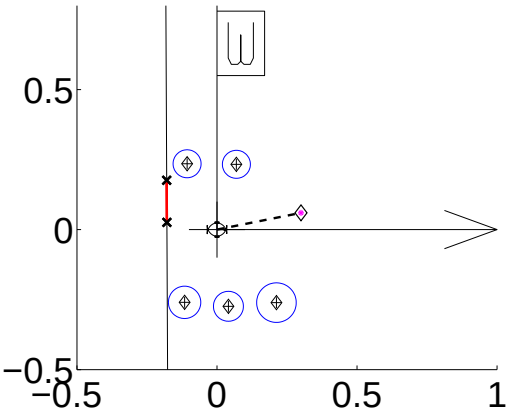
Beacons (also called point feature) is the second landmark the SLAM-algorithm uses, after the lines. Beacons are generated by small clusters of measurement data, which are not part of a line. It can seem to be random when beacons are generated by the robots, but it happens more often when the sensors produce measurement data with some dispersion. This is common if the wall is at long range, or the viewpoint is skew relative to the wall.

A good example of the randomness of the beacon generating is shown in figure 8.1. This is actually two consecutive scans of the very same environment, shown in figure 8.1(a). The first scan shown in figure 8.1(b) produces many beacons and one wall segment, while the second scan shown in figure 8.1(c) renders the real environment quite good.

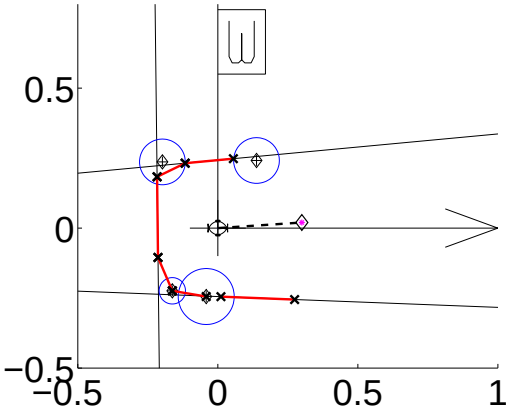
As can be seen from this example, when a beacon is generated it means that the robot has discovered something. This means that the robot must avoid these points when exploring the environment. This can be accounted for in the navigation algorithm and in the path planner algorithm.



(a) The actual environment



(b) First scan produced many beacons



(c) Second scan reproduces the real environment in a good fashion.

Figure 8.1: Illustrating issue with beacons

8.1.1 Beacons and the navigation algorithm

The ideal solution to account for beacons would be to use the same approach as for the wall segments, but this approach is rather difficult when considering points. This is simply because is hard to define what is behind a point. It could be considered that beacons could be grouped together and be considered as wall segments, but this approach can close valid gaps between walls and prevent the robot from completing the mapping task.

The solution at this point was to lead the robot away from choosing target points near beacons. This was simply done by penalizing the distance from a frontier to a beacon in the cost function. The penalty was defined similar to the wall cost, as shown in equation 8.1

$$b(\bar{d}_b) = \frac{1}{\bar{d}_b^2} \tag{8.1}$$

where $\bar{d}_b$ is the distance to beacon. This cost will be very high close to beacon, and quickly decline when the distance increases from the point feature. Hence, the target points would not be chosen to be close to beacons.

8.1.2 Beacons and the path planning algorithm

The next step in order to account for beacons is to avoid driving close to them. A beacon would not stop the navigation algorithm from choosing a target destination on the other side of a beacon. This would most likely lead the robot to drive into a wall. It was therefore

checked if the straight line from the robot to the target destination was sufficiently close to a beacon, and if a path had to be planned to avoid it.

The next step was to remove all configuration points around the beacon such that a path cannot be planned to travel near it. The target destination would then be moved from outside the free configuration space to the closest point in the configuration space, such that a valid target point was created near the actual target point.

When the robot was able to move, it could get a different viewpoint, and then identify a wall segment in the area the beacon was.

If no path is found

In theory, it would always exist a path since the discovered area should always be connected. This is because the robot would never drive into completely unexplored area, it would always drive to a frontier point, which is connected to the discovered area. The robot would therefore, in theory, only make a new bulge in the discovered area for each position it visited. This was also the case when testing the path planner in the simulator.

However, this was not the case for the physical system. The reason originated from the uncertainties of the physical system, such as producing incomplete walls even though the robot was completely surrounded by walls. This could produce infeasible target destinations in a part of the configuration space, which were not connected to the part of the configuration space the robot resided in.

The beacons could also separate the configuration space into different parts since all configuration points around the beacons were removed to avoid planning a path near them.

A consequence of a disconnected configuration space was that the robot might be 'stuck' at one position. This stops the progression of the exploration, since several scans at the same position may not give new information. This will make the navigation algorithm choose the same target point each time. The robot would then remain at the same position, trying to find a path to an unreachable destination. It is therefore very important that the robot get to move.

The solution to this problem was to avoid unreachable target points, and an area around them. This had to be performed by the SERVER-robot of the system, which calculated the target points for both robots. Therefore, if the robot remained at the same position too many iterations, the current target point was classified as unreachable, and added to the list of unreachable areas.

This forced the robot to choose a different target point, which then made it move and continue the exploration.

8.2 More improvements

In order to further improve the performance of the physical robot system, a few more improvements were made.

Connecting wall segments was done since the SLAM-algorithm often produced glitches between wall segments, which would lead to the creation of frontier points inside walls. This prevented many situations, which would lead the robot to infeasible target points.

Filtering sensor data in order to remove obviously erroneous measurements. This has been especially relevant for the IR-robot, but after a closer look at the wires connected to sensor, the sensor data has been improved significantly. However, both robots might still produce a few erroneous measurement points, which is filtered out. The main case is if the points are too far away, or if the dispersion (distance between the measurements) is too great. Single measurement points are usually originated from erroneous sensor data.

Chapter 9

Testing

9.1 Testing the navigation with one robot in simulator

This section presents the very first testing of the the new navigation algorithm for one robot, and compares its performance to the previous 'Left-wall-follower to backtracker'-algorithm.

For this test, the simulator with Map 1 was used, and the results is shown in figure 9.1. The new navigation algorithm used 18 steps and travelled 5.6 meters, while the 'Left-wall-follower to backtracker'-algorithm used 51 steps and travelled 5.48 meters.

However, the 'Left-wall-follower to backtracker'-algorithm was unable to finish the exploration since it crashed into the wall during the backtracking phase, which shows that the old algorithm had some errors. Hence, the actual number of steps and travel distance would be

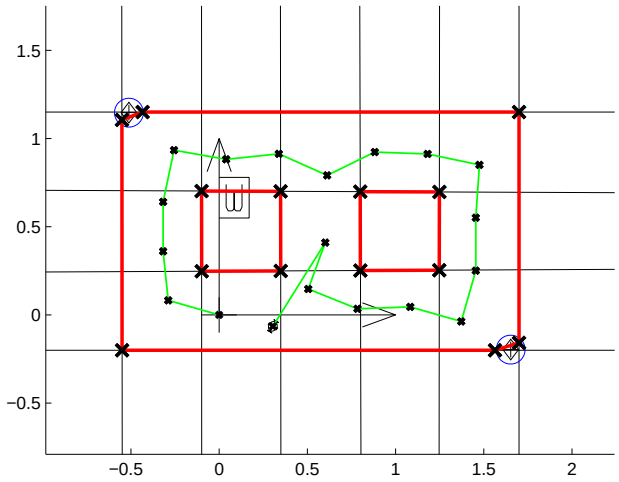
higher.

This test shows that the new navigation algorithm is more efficient, since it reduces the number of steps quite drastically. For the real robot system, each step will take some time due to the full scan procedure, and therefore will the new exploration algorithm reduce runtime significantly for this case.

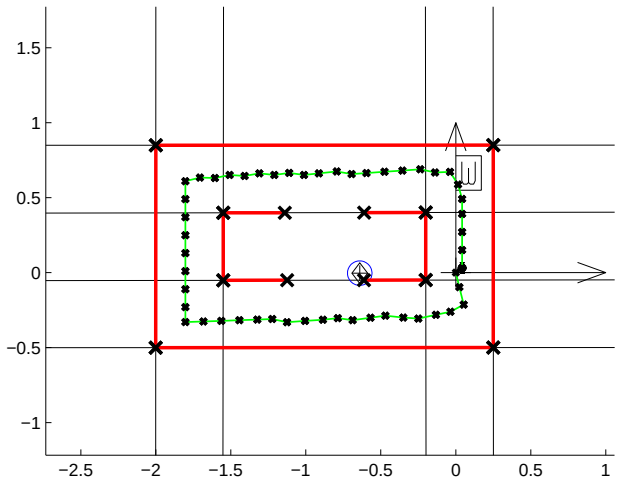
Another interesting test was with a convex and open test course, as Map 6 of the simulator provides. The result is shown in figure 9.2. The exploration algorithm completed this track with 29 steps and travelled 12 meters. The 'Left-wall-follower to backtracker'-algorithm used 62 steps and travelled 6.2 meters. Again the 'Left-wall-follower to backtracker'-algorithm failed when it initiated the backtracking state, which means that the actual numbers are higher. Hence, the new navigation algorithm would perform better in the case of open environments.

This test illustrates a distinction between the two algorithms. The 'Left-wall-follower to backtracker'-algorithm, has a searching strategy which follows the left wall, while the new navigation algorithm picks new position more at random.

An interesting phenomenon is the remarkably even distribution of positions produced by the new navigation algorithm, which tells that the search environment is covered in a good fashion. Figure 9.3 shows the robot positions without the path drawn between the points.

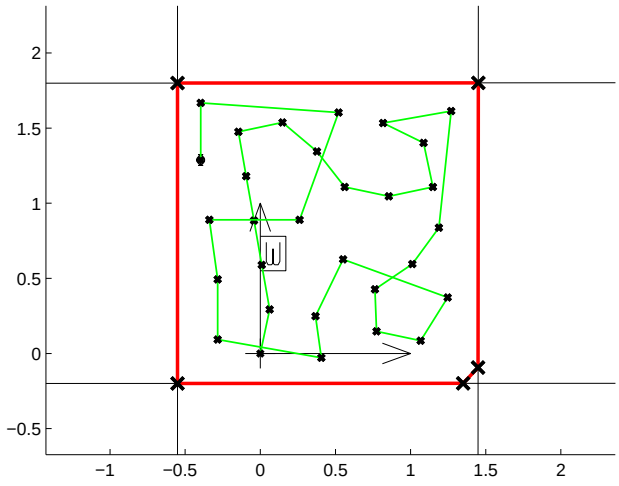


(a)

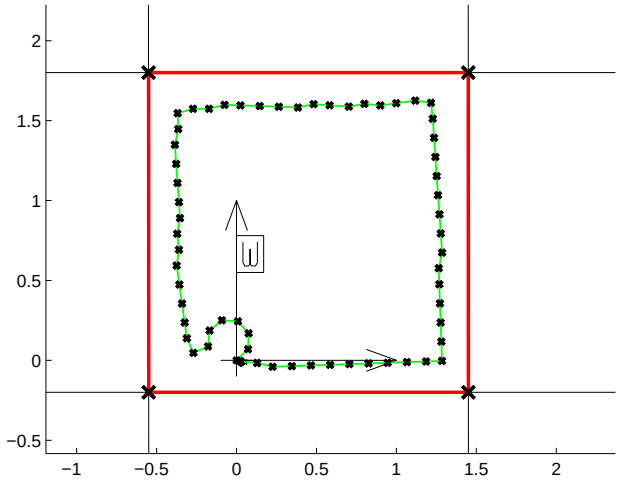


(b)

Figure 9.1: The first run of the new algorithm in Map 1. The result of the new algorithm is shown in 9.1(a) and compared to the previous algorithm in 9.1(b).



(a)



(b)

Figure 9.2: Testing in Map 6, a convex environment with no hulls. The result of the new algorithm is shown in 9.2(a) and compared to the previous algorithm in 9.2(b).

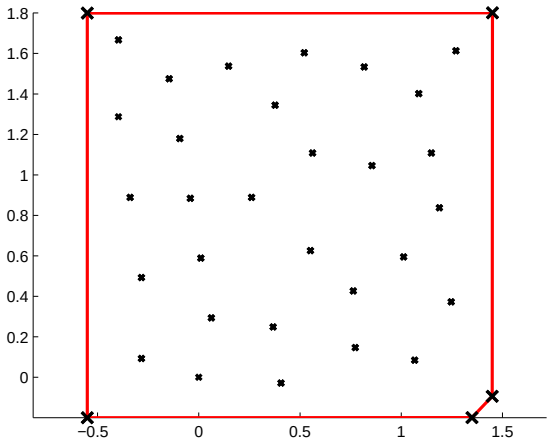


Figure 9.3: The even distribution of robot positions shows good coverage of the area by the new navigation algorithm.

9.2 Testing the path planner in simulator

This section presents the testing of the exploration of an area with focus on the path planner algorithm. For this test Map 3 was used, which is a more complex map than in section 9.1. This course were used since a path planning case had to occur.

The result from the scan is presented in figure 9.4. For this course, the path planner was needed twice. Figure 9.5 shows the last case where the path planning was needed. This is a typical case, since the robot had explored everything within the minimum range. This meant that it had to travel across the workspace to find an area which was not explored.

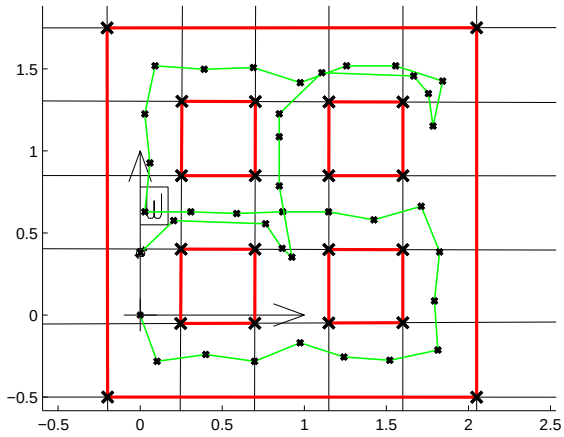
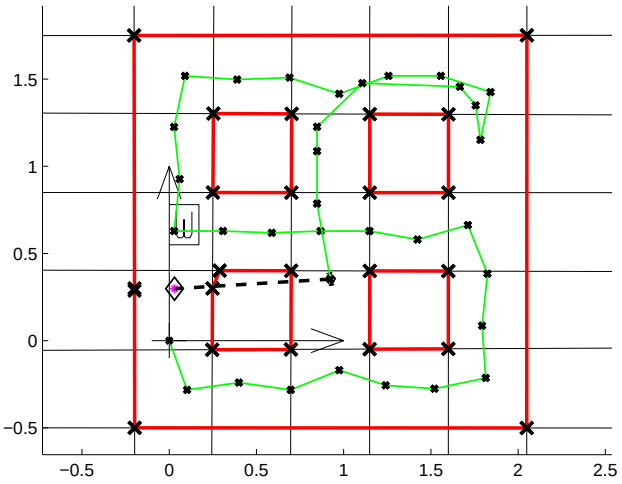


Figure 9.4: The result from Map 3 with the new navigation algorithm and path planning.

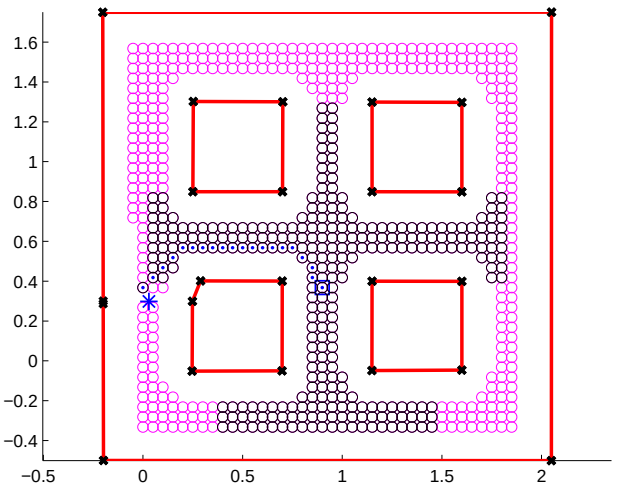
Figure 9.5(b) shows the path found by the BFS-algorithm, while figure 9.6 shows the final path.

This test shows that the new navigation algorithm were able to explore an arbitrary indoor environment in an efficient manner. It could now pick target points anywhere in the work space, and get there safely.

The 'Left-wall-follower to backtracker'-algorithm were not able to explore this course due to the error during the backtracking phase, and could therefore not compare its performance to the new algorithm.



(a) The next destination received from the navigation is such that the straight line collides with the wall segments.



(b) The circles marks the free confuration space, the black circles are the nodes investigated by the BFS-algorithm, while path is marked with blue dots.

Figure 9.5: Figure 9.5(a) shows a typical case where path planning is needed. The shortest path is calculated in figure 9.5(b)

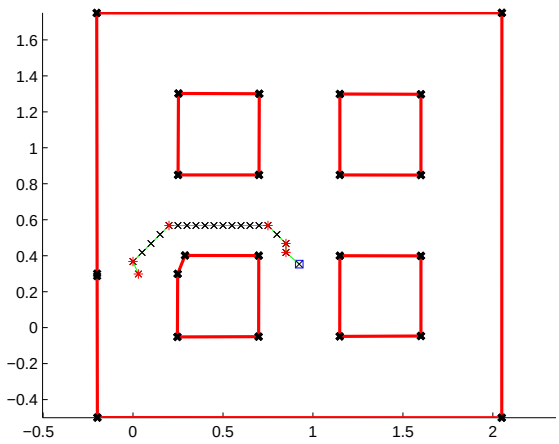


Figure 9.6: The path planned for the case in figure 9.5(a)

9.3 Testing with two robots in simulator

The test performed in this section tests the coordination capabilities of the system, which were described in chapter 7.

For this test, all three maps (Map 1,3,6) were used such that the results could be compared with the performance of the single robot system. The performance of the multi-robot system would equal the number of steps and travel distance of the slowest robot, in terms of the robot which performs most scans.

Further, in all three tests the robots starts near each other, since this would be the most realistic for a real search in a collapsed building, mine sweep and so forth. The test results is displayed in table 9.1 together with the results from exploration with one robot for compar-

ison.

Test 1, Map 1

For the first test simulator map 1 was used, and the result is shown in figure 9.7.

The robots started down in the left corner, and chose to move in different direction at step 1. After the first scan, the discovered area of the two robots overlapped such that there were three frontier regions to choose target points from, as shown in figure 9.8(a). One of the frontier regions is in the middle of the robots, but because of the cost function, neither of the robots choose this target.

The next steps did not require any coordination, until they approached the upper right corner from each direction. In figure 9.7 the two paths of the robots is displayed together, which shows that this 'corner'-case was solved. As the robot travelling along the blue path approach the corner, it initially chooses a target point in the middle of the map, while the second robot finishes searching the corner. Since the path planner for the blue robot cannot move past the green robot, it starts to move in the direction it came from. When the green robot is finished in the corner, there are two frontier regions left, one in the middle and one near the starting position, as shown in figure 9.8(b). Now, the algorithm assigns the middle target to the green robot, and the target near the start position to the blue robot. Now, the optimal solution for the overall system is made, and the number of steps is minimized.

For this test, the two separate SLAM-maps generated from both robots is shown in figure 9.9. They make the foundation for the result shown in figure 9.7. Since this simulation is conducted without measurement- and positioning errors, removing the initial offset between the robots is sufficient for the map merging.

Table 9.1 shows the results from this test together with the corresponding test with one robot. An interesting fact is that using two robots does not improve the run time significantly for this map. The reason is that both robots needs to use the path planner in order to cover the frontiers, which were left unexplored at 'crossroads', where the robot had to choose between two possibilities. Therefore, the use of multiple robots for smaller maps would not give a great advantage over one robot, as could be expected.

Test 2, Map 3

For the second test, simulator Map 3 was used in order to check how the system handled a bigger environment. The result is shown in figure 9.10. The tests shows that the robots managed to collaborate in a good fashion, and dividing the workspace fairly.

The robots had to be coordinated, similar to the previous test, at the beginning and the end of the exploration. The first case was identical to the case in test 1.

The critical coordination case occurred towards the end of the exploration, as shown in figure 9.11(a). Since the green robot chooses target point first, the blue robot is assigned the target point on the

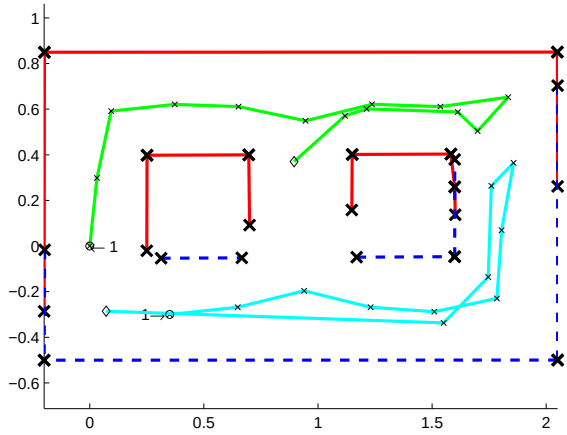
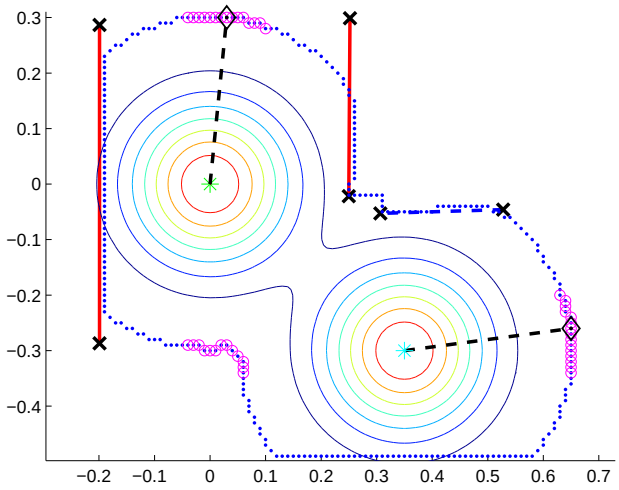
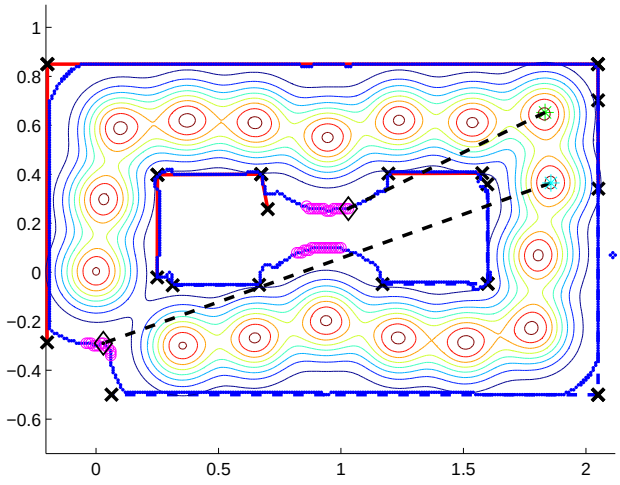


Figure 9.7: The resulting map from exploration in Map 1. The two SLAM-maps from figure 9.9 has been merged together. The 'server'-robot is marked with green and contributed with the red wall segments.

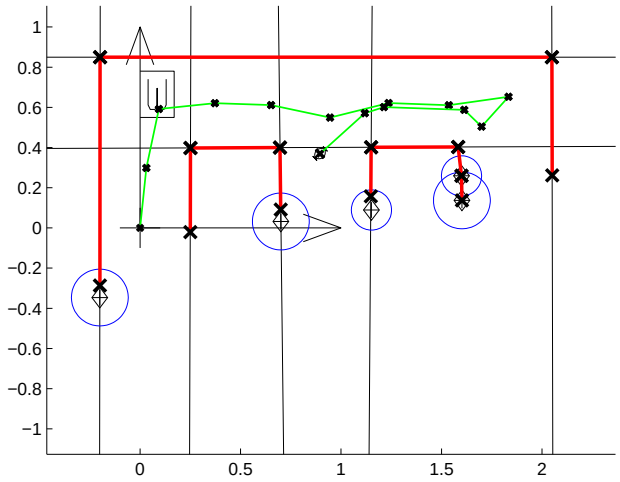


(a) Step 1

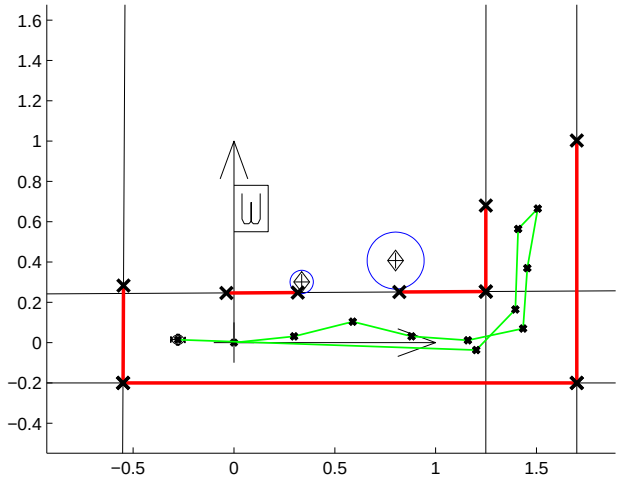


(b) Last coordination decision.

Figure 9.8: Coordination cases from Test 1. The Tactical map is marked by a contour-plot and the frontier points are marked with pink on the blue border.



(a)



(b)

Figure 9.9: The two SLAM-maps generated by the robots. These maps shows what the robots has discovered separately. These figures also shows the line- and beacon features, which is used by the SLAM-algorithm.

other side of the green robot, which has the 'server'-role. At first, the blue robot must plan to go around, since the area in front of is not connected. As the green robot moves upwards, the blue robot is allowed to take a shorter route and begins to move the other way again. This makes the blue robot go back and forth, as can be seen by the dense gathering of position points in figure 9.10. As the green robot moves further, as shown in figure 9.11(b), it assigns the last frontier to itself, and tells the other robot that it is done. Hence, the coordination was solved.

As can be seen from table 9.1, the test results shows that the use of two robots improved the run time of the exploration significantly. The number of scans and the distance the robots had to travel was reduced by more than half. Hence, the test shows that the system benefits of using more robots for larger workspaces is large, which shows that the algorithm for collaboration is working.

Test 3, Map 6

The final test was in the open environment, and this test would really test the coordination of the system.

The path planner could not help the coordination for this case, so it was crucial that the target points were chosen such that the robots did not cross paths.

The results is shown in figure 9.12. As can be seen from the figure, the two robots divided the area between each other, and finished the mapping without any collisions. This test shows the effect of the

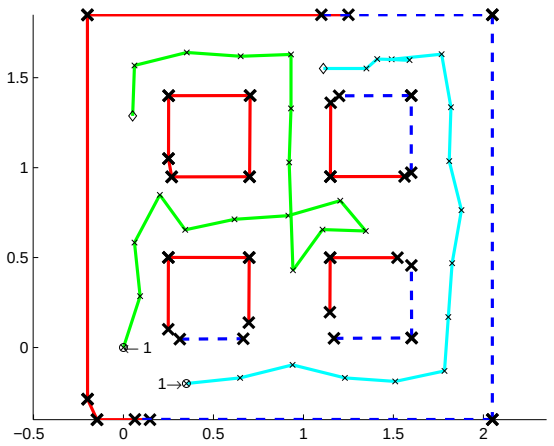
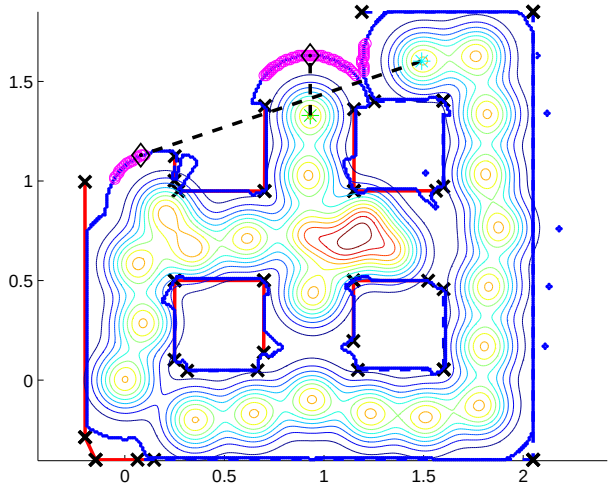
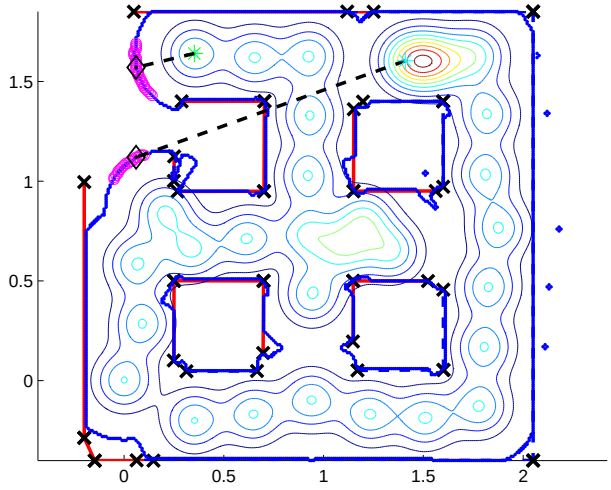


Figure 9.10: Test results from Test 2 with two robots. The robot travelling along the green path produced the red wall segments, while the robot travelling along the blue path produced the blue wall segments.



(a)



(b)

Figure 9.11: Coordination case for Test 2. These maps shows the Tactical map marked by a contour-plot and the frontier points are marked with pink on the blue border.

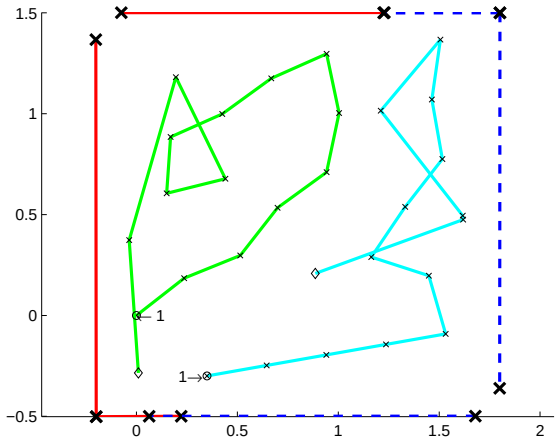


Figure 9.12

'distance to the other robot'-cost, which was sufficient for coordinating the robots on this case.

The results for this test shows that the multi-robot system performed about twice as good as the single robot system. This test also shows that the robots are coordinated such that the workspace is evenly divided between the robots. The robots are dispersing at each step, trying to move in different directions at each step, which was one of the goals for the exploration algorithm.

	Two robots		One robot
Map 1	Robot 1, Server	Robot 2, Client	
Steps	14	12	18
Distance [m]	3.48	4.36	5.6
Map 3			
Steps	18	17	38
Distance [m]	5.12	4.12	11.1
Map 6			
Steps	15	16	29
Distance [m]	5.32	4.92	12

Table 9.1: Comparing simulation results with one and two robots.

9.4 Testing the improvements of the NXT-robot

This section performs the test of the positioning and sensor tower angle feedback.

For the positioning test, the NXT-robot had to follow a square path by 10x10 cm. The NXT-robot got the commands to turn -90°and drive 10 cm four times. Figure 9.13(a) shows the result of this test track together with the ideal path. As can be seen, the NXT-robot performed well with a small error. The final destination should be in the origin, but is displaced by a few millimeters.

For the tower angle feedback test, the NXT-robot performed five scans of the same area from the same positions. This test would show the tower angle offset. The result is shown in figure 9.13(b). The test course is similar to the test course depicted in figure 8.1(a).

As can be seen from the results, the measurements twists some after a few iterations, but the results also show that the robot is still able to recognize the area in a good fashion.

9.5 Testing the exploration on the physical robot system

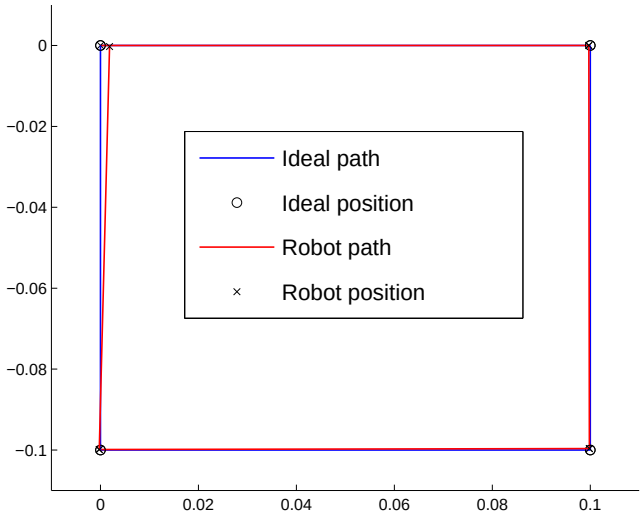
This section will describe the testing of the physical robot system, with both the NXT- and IR-robot. This test will show that the algorithm was working properly with the physical system.

The test course is presented in figure 9.14, which is an arbitrary non-convex course which needs good coordination in order to be completed without collision due to the small size.

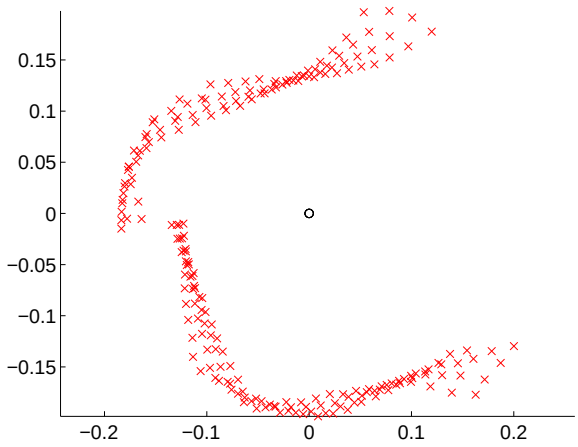
The resulting map is presented in figure 9.15, together with the paths of the robots.

As can be seen from the result, the map has reproduced the real environment in a fairly good fashion. It is possible to recognize the area as the two rectangles is composed of, which means that the mapping task has been completed.

The sensor data from the IR-robot is shown in figure 9.16, and shows what the IR-robot has seen. As can be seen from the figure, the sensor data of the IR-robot has mapped the area in a good fashion. For this reason, one could expect better results from the SLAM generated map. However, since the SLAM-algorithm tries to fit lines to sensor data, it may have problems if the data fits badly to a line, even though



(a) Testing of the positioning. Robot starts in the origin, and drives 10 cm forwards. Then turns 90°and drives 10 cm until it get back to the origin.



(b) Testing the sensor tower feedback.

Figure 9.13

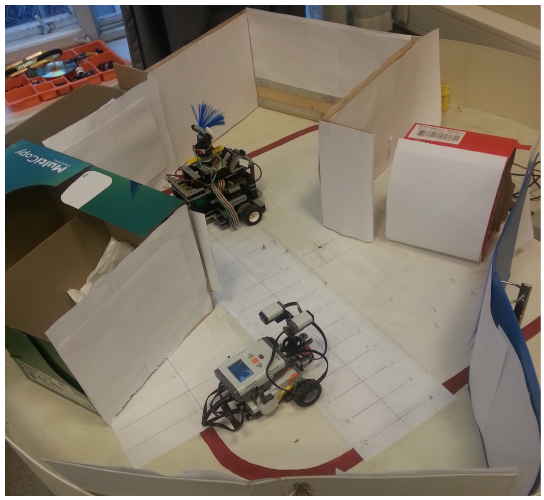


Figure 9.14: Test course

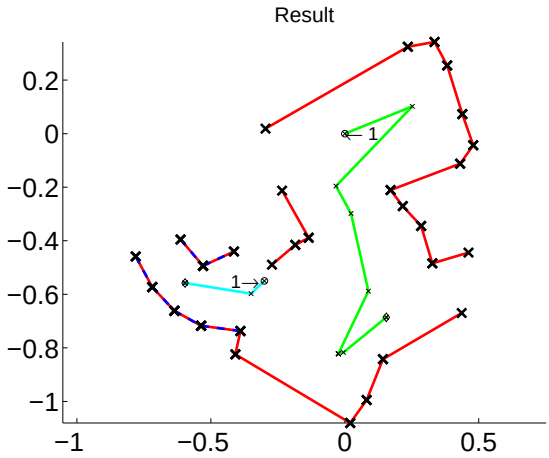


Figure 9.15: The resulting map and the robot paths in it. The IR-robot followed the green path, and contributed with the red wall segments. The number '1' marks the starting position of the robots.

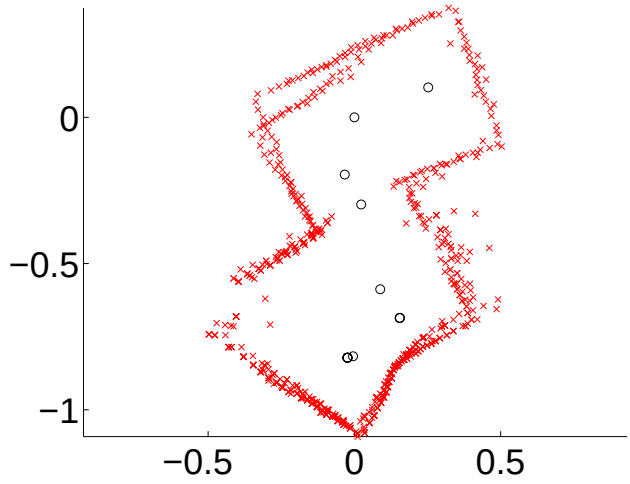


Figure 9.16: Sensor data from the IR-robot and the positions it visited.

the scanning is accurate. This is a problem since each iteration the positioning error will grow, which results in that the sensor data slowly rotates, and is therefore slightly displaced compared to the previous scan, even of the same area.

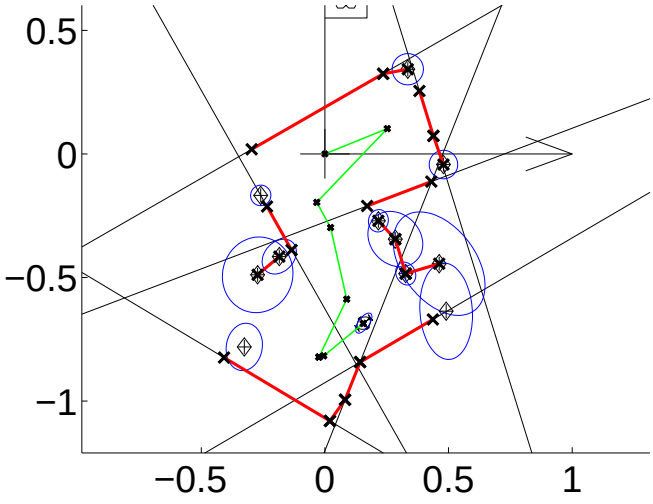
Further, the map data from the robots has not been matched, which can be seen from the result in figure 9.15 where it seems like the NXT-robot starts within a wall the IR-robot has mapped. The two separate SLAM-maps is shown in figure 9.17. From these figures, it is possible to see that the corners have glitches which the `linkWallSegments`-procedure has fixed.

Maybe the most visible issue is that the IR-robot has mapped

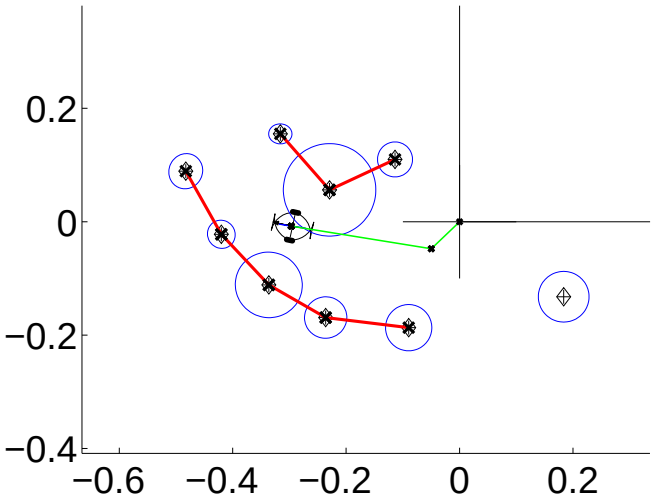
significantly more than the NXT-robot. This is because the IR-robot performs the `fullscan`-procedure much faster than the NXT-robot. For this reason, the IR-robot has been chosen as the leader of the two, the SERVER, and calculates the target positions for both robots. Hence, the IR-robot does not have to wait for the slower NXT-robot to complete the `fullscan` before it calculates the new position.

Another reason for the uneven division of the workspace is that the NXT-robot is assigned more infeasible target points. This is caused by creating incomplete wall segments, and producing many beacons. For this reason, the NXT-robot would remain at the same position until it finds a feasible target point. It will identify the infeasible target points, but by the time the NXT-robot has done that, the IR-robot has almost completed mapping the entire area.

This test has showed that the system is able to complete a mapping task by collaboration, and hence the system is more effective than the single robot system.



(a) IR-robot SLAM map



(b) NXT-robot SLAM map

Figure 9.17: The two separate SLAM-maps of the robots.

Chapter 10

Discussion

This chapter will discuss the results of this thesis, and consider how each topic presented in the problem description was solved. This chapter will then form the basis for the conclusion of the thesis.

The tests performed in chapter 9 were discussed as the tests were presented, but in this chapter the discussion will have a more general view point and look at the entire system.

10.1 The Simulator

The simulator was the most important tool for developing the algorithms, which were designed in this thesis. Therefore, it was important that the simulator worked properly, and that it could be used for simulating multiple robots as if they were the physical robots.

What the simulator needed was to communicate with the other

simulator, and to set some variables in `RobotHandler` which was achieved by letting the simulator access `RobotHandler`. Hence, the integration of the simulator was achieved, and the simulator performed well throughout the entire thesis.

It was discovered that some of the maps had some flaws, which allowed the robot to drive through the wall without colliding.

Further, there is no knowledge of how the maps looks like prior to running the simulator. However, they are displayed in Tusviks [16] report, but there would still be a problem to set the initial position of the robot. The problem is easily solved by trial and error, but it would be convenient to know where the robots were placed, and how the map looked like.

10.2 The Communication protocol

The communication between the robots were the most important module, and the key to allowing the robots to collaborate. The previous version from the project of [19] were incomplete, and could therefore not guarantee reliable communication.

This problem was solved for this thesis, which were able to send and receive packages by the new protocol of the system, and in addition without being concerned about when a package was sent.

However, the `write` functions of MATLAB had their limitation when it came to size of the packages. If the for instance the SLAM-map was too big, an error occurred which complained that the package

size was too large for sending.

The same problem applied to the `read` function too, but this was the main problem for this thesis and solved by temporarily storing incomplete packages. This assured that packages was not lost due to protocol errors, i.e. because a package could not be recognized on the correct format, which would happen if the package were incomplete.

Still, this protocol has its limitations since the `read`-function has to be called in order to check if a package has arrived in the input buffer. A better solution would be if the robot was notified when a package arrived such that the communication was event based. This way, data packages could be read and stored right away, such that there would be no delay in the communication. This will in turn allow a more autonomous and dynamical system, since if the robots continuously broadcasted their position and other information, it could be considered that the robots travelled continuously and scanned while driving.

10.3 The map merging

The two SLAM-generated maps were merged by removing the initial distance between the robots. This is a simple approach, which will only give reasonable results for small environments and few scans. These conditions apply to this robot system, since the test bed is smaller than $2x2m$, and the robots uses less than 20 steps in order to map an area in this environment.

The map merging from the simulator is not show displacement after the map merging. But, the test result from the physical robots shows that there is a displacement between the merged maps, as can be seen from figure 9.15. This is due to the positioning errors, and measurement errors.

A map matching approach should be implemented in order to match the maps properly.

10.4 The Navigation algorithm

The approach of the navigation algorithm was to find the frontier point, which minimized the cost function defined for the algorithm. This approach has been working fine, but there are of course improvements which can be made. Anyway, the navigation algorithm has produced reasonable target destinations, and is easily controlled by adding or removing costs.

The tactical map

The purpose of the tactical map was to give an estimate of the utility to a point, or the information gain. The estimate was calculated based on how well a frontier was covered by the robots. This solution is the opposite of the ray-casting approach, which many of the articles mentioned in the literature study used. The ray-casting approach investigated how much unexplored area surrounds a frontier, which might give a better estimate, but depends on the size of the

occupancy grid which is used. For this thesis, the occupancy grid has been dynamical, such that there was no knowledge of the size of the environment which were explored. Therefore, the ray-casting approach would maybe not give a better information gain estimate, since it did not have information about how far it could send rays, except from that the range will have to be minimized since the map would be scaled to only include the discovered area.

Finding frontiers

This part was the most crucial part of the navigation algorithm, and had to function properly. The process of finding the frontiers began with finding the border of the discovered area. This was solved by looking at what was visible from each robot position, and considering points behind wall segments to be unreachable.

This method seemed to work fine for the simulator, since all wall segments which divided the environment were sufficiently far away from each other. As mentioned, it was a problem of deciding what was reachable or not, illustrated by the considering a corner as in figure 5.5. A more difficult consideration would be if one wall segment acted as a partition wall. If a single wall segment is considered from each side, this method will fail to provide a reasonable result. Therefore, this method of finding frontiers would only work if the wall were sufficiently thick.

Still, the method has proved to function properly, together with the identification- and merging process of border segments, which follows

the procedure of finding the border of the discovered area.

The Cost function

The costs defined for the cost function decides where the next target destination of the robot is. Five costs has been used, which are travel distance, information gain, nearness to walls, beacons and the other robots target position. In spite of the number of costs, the algorithm has calculated reasonable target positions for the robots. However, too many costs may cause confusion, and it is not advised to add more costs to this cost function.

The costs has been carefully weighted relative to each other such that all costs can be 'heard', and it seems like this has been done reasonably. It can be seen from the tests that the effect of each costs has had an effect on the decision made. The robots prefer targets close by, in open areas as far away from the other robot as possible.

The cost estimates could be improved, as for instance the travel cost. For this thesis, the path along the straight line has been used, but the real travel cost would be the path the robot has to follow in order to reach the target. But, planning a path for each frontier has been considered to be too time consuming and therefore not used.

10.5 The Path planner

The path planner for this thesis has been found to work properly, but this is a part which can be improved. The greatest limitation

is caused by the driving nature of the robot, which is composed by sequential turning and driving. A more smooth driving style would benefit the efficiency of the driving, such as being able to turn while driving forward.

The goal was to plan a path such that the number of turns could be minimized. This has not been accomplished in an optimal manner. As can be seen from the tests of the path planner, the path will make a 'v'-shape if it travels in open areas of the graph. This phenomenon can be seen from the figure 6.4. It approaches the goal head on when possible, but if the robot has to go back around an object it turns back up and the 'v'-shape were made.

This is caused by the way the path is generated when the goal node is found. Then the BFS-algorithm adds the parent-nodes to the path, but for each node there are more than one potential parent. By adding weights to the nodes in certain directions, this problem can be solved.

In summary, the algorithm will minimize the distance in terms of number of nodes, but not in terms of meters.

10.6 Comparing the Exploration algorithm with the Left-wall-follower to backtracking algorithm.

The Left-wall-follower to backtracking (LWFB) algorithm and the new exploration is quite different, and will be compared to see if it benefited the system to write a new navigation algorithm.

The most significant difference is the searching strategy of LWFB, which is to follow the left wall until a familiar area is found, then backtrack until a new unexplored area is found near the backtracking path. The optimization approach will choose points at random, and always try to maximize the expansion of the discovered area.

Another significant difference between the two algorithms is that the step length of the new navigation algorithm can be adjusted, while the LWFB has a fixed step length. By increasing the step length, the robot can explore more at each scan with the exploration algorithm, and would therefore be faster. The step length would often be limited by the range of sensors, but would allow the robot to travel across the workspace, through known areas, in order to find a frontier.

The step length of LWFB is very limited in order to avoid path planning, which means that the robot spends more time in the discovered area. Since the robot performs significantly more scans with this algorithm, the time efficiency would be poorer due to the time it takes to complete a full scan.

The dynamic step length is one of the reasons why the new explo-

ration algorithm is significantly more efficient in the tests performed in chapter 9.1.

Further, the approach of LWFB is much more suited for a single robot maze solving system, since the algorithm is based on a maze-solving algorithm.

The new exploration algorithm is also more scalable, and can more easily include more robots, since the optimization approach would only find new target points as long as there were frontier points to visit.

In conclusion, the new exploration algorithm performs better than the LWFB-algorithm, and was necessary in order to make the robots able to collaborate.

10.7 Overall performance of the multi-robot system

As can be seen from the testing in chapter 9, the algorithm is able to complete the mapping task for both robots, which means that the algorithm is working properly. There are however a few issues to look closer at.

During the early stage of the exploration, the robots can navigate without being concerned with the other robot. For this phase, the decisions made are usually good. The robots travels in different directions as can be seen from test results in figure 9.7 and figure 9.10.

The real issue appears towards the end of the navigation when there are few and small frontier regions left. This is when the coor-

dination is crucial, and target positions have to be chosen carefully. The outcome of the algorithm is that these coordination problems are solved, but maybe not in an optimal manner. This caused by the SERVER/CLIENT-structure, where the SERVER-robot is too 'bossy'. As the main rule, the SERVER-robot will choose target destination first, and then choose the target which is optimal for itself and not for both. This leads to that the SERVER might take over a target destination from the CLIENT, which the CLIENT may be on its way to. Then the CLIENT is assigned another target position, which can have the effect that the CLIENT drives back and forth when its target position is changed before it reaches a target.

This problem has been addressed in the literature [14],[20], where the solution has been that the robots bids on target positions such that the optimal solution for both robots can be made. The implementation can lead to a more autonomous system, since both robots can pick target positions and offer its bids to the other robot. Then after a bidding round it is decided which target the robots go to.

In order to even out the power difference between the robots some, for this thesis, it is considered who gets the last frontier. If the SERVER-robot sees that the CLIENT-robot is closer, the CLIENT-robot get it. This last coordination case has often been the one which is least optimal, since the SERVER-robot has taken it regardless of where the CLIENT-robot has been. However, since this case involves only one frontier, it has been easy to correct it by checking which one who is closest.

10.7.1 The performance of the physical robot system

The physical robot system were able to complete the mapping task with the use of two robots. However, the distribution of the workload was quite uneven. As mentioned in the test chapter, the IR-robot is faster than the NXT-robot and will therefore be able to perform several scan for each NXT-scan.

In addition to this, the NXT-sensors does not produce as good results as the IR-sensors, which leads to the creation of more beacons and more incomplete walls. Therefore, the NXT-robot will be assigned more infeasible target destinations which the robot is unable to travel to. Thus, the NXT-robot remains at the same position for longer before it finds a feasible target point.

10.8 Improvements of the NXT-robot

The improvements done on the NXT-robot was to implement positioning based on odometry, and feedback on the sensor tower heading. These improvements were crucial in order for the NXT-robot to deliver reasonable sensor data, and for the robot to be of use in the mapping task.

The positioning were implemented, and the NXT-robot were able to send the position estimate back to MATLAB with the protocol described in chapter 3. This position error were now significantly improved compared to the previous approach. However, the NXT-

robot still has a positioning error due to the dead reckoning approach, which accumulates an error over time. Nevertheless, it was considered to be as good as for the IR-robot, which was the goal.

In order to correct this position error the robot should have a reference point where it knows the environment, such that a scan at this point can show the positioning error and then correct it. It has been tried to implement a 'Go home'-function in an earlier project, but this functionality has not been tested or investigated more closely in this project.

The sensor tower also performed significantly better than the previous approach, but it still gets an offset, which grows for each scan. However, this offset is much smaller than for the previous approach. This can be seen from the global sensor data, since the data is drastically more coincident. For the previous sensor tower approach the sensor data was almost unusable after a few scans, due to the great offset which was accumulated.

The surface may influence the positioning if it is uneven or has high friction, such that the robot wheels spins. This may prevent the robot from turning and driving the desired distances, and lead to positioning errors.

10.9 Conclusion is stated at the beginning of the thesis

The summary and conclusion is stated at the beginning of the thesis at page ix.

Chapter 11

Further work

11.1 SLAM

A new SLAM-algorithm could be implemented in order to improve the system and exploit the use of multiple robots.

SLAM for multiple robots The SLAM algorithm uses landmarks for estimating the position, and averaging out the accumulated error from the positioning from the robot. It is therefore desirable to observe the same landmark from several positions in order to improve the estimate. Since the SLAM-framework provided by Kai Arras [1] is designed for one robot, the robots cannot use each other's observations for localization and correction of the map. This is a weakness of the system since valuable information is not exploited.

However, due to the primitive infrared-sensors, the robots are un-

able to recognize each other. Nevertheless, their positions could be used for estimation of walls and their own position.

SLAM for arbitrary environments The current SLAM-algorithm uses lines and beacons for map building, which means that it is only capable to recognize straight walls correctly. If the robots see a curved wall, it will try to fit lines to the sensor data. In order to render curved objects, another SLAM approach has to be implemented.

11.2 Map merging

The two maps produces by the robots should be matched properly such that overlapping segments are recognized and merged together. Further, with positioning and measurement errors, map matching is needed in order to avoid the two maps to be crooked and displaced relative to each other.

11.3 Communication

The communication module can be made event-based, such that the robots are notified when a message arrives. This method would be more reliable and more accurate, since the new information can be known immediately. Further, if the robots could broadcast their position continuously, then a more flexible and autonomous navigation algorithm could be implemented.

11.4 Simulator

Currently, when two robots are simulated they explore in two copies of the same map. This means that they cannot see each other or collide. The second robot can be included in the simulator-code such that it is detected during the full scan procedure of the simulator.

11.5 The Robots

- Extra functionality can be implemented, such as manual driving for the NXT-robot.

- The dead reckoning approach of the position estimation accumulates error over time. It has been attempted to have a known reference position the robot can go back to in order to check its position, and correct it.

11.5.1 NXT-robot

Collision detection

The NXT-robot does not have collision detection implemented. This is a basic functionality that should be implemented on the NXT 2.0 brick. During the driving stage, the sensors should take measurements, and stop the robot if a wall is detected within some range.

Set sensor tower to 0°

The robot is now able to scan approximately from 0°to 180°, but the tower has to be set to approximately 0° by hand before the program starts. Knowing where absolute 0° on the motor would not be enough since there are two cogwheels between the motor and the sensors.

A possible way might be to find 0° on the motor, then try to adjust the sensors to point in 0° direction. However, it may be difficult to do this precisely.

Remove sensor-tower angle offset

The sensor-tower angle offset is still a problem, which has to be removed in order to have the robot produce good sensor data.

11.5.2 IR-robot

Noisy measurements

The sensor data from the IR-robot often contains data which is clearly wrong. It has been attempted to filter this data out, but some erroneous points remain. Other approaches can also be tried, since it is not know exactly what is the origin of this error.

11.6 Path planner

A new path planning algorithm can be implemented instead of the BFS-algorithm. For instance the A* algorithm, or a modify version could be more effective.

11.7 Navigation algorithm

The robots can be more autonomous by decentralizing the system, such that each robot makes its own decision for the target point. The robots should agree on positions, such that the optimal target points for both robots can be found. The exploration algorithm should make out a good foundation for this work, with the frontier-based approach.

Bibliography

[1] Kai O. Arras. *CAS Robot Navigation Toolbox Version 1.0.* 2004. http://www.cas.kth.se/toolbox/.

[2] Mikael Berg. *Master thesis: Navigation with Simultaneous Localization and Mapping, For Indoor Mobile Robot.* NTNU, 2013.

[3] Adrian Boeing, Thomas Braunl, Robert Reid, Aidan Morgan, and Kevin Vinsen. Cooperative multi-robot navigation and mapping of unknown terrain. In *Robotics, Automation and Mechatronics (RAM), 2011 IEEE Conference on*, pages 234–238. IEEE, 2011.

[4] Adrian Boeing, Sushil Pangeni, Thomas Braunl, and Chang Su Lee. Real-time tactical motion planning and obstacle avoidance for multi-robot cooperative reconnaissance. In *Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on*, pages 3117–3122. IEEE, 2012.

[5] Wolfram Burgard, Mark Moors, Dieter Fox, Reid Simmons, and Sebastian Thrun. Collaborative multi-robot exploration. In

Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on, volume 1, pages 476–481. IEEE, 2000.

[6] Wolfram Burgard, Mark Moors, Cyrill Stachniss, and Frank E Schneider. Coordinated multi-robot exploration. *Robotics, IEEE Transactions on*, 21(3):376–386, 2005.

[7] Trond Kåre Homestad. *Master thesis: Fjernstyring av legoroboter*. NTNU, 2013.

[8] Samuel Lopes, Brian Frisch, Adrian Boeing, Kevin Vinsen, and Thomas Braunl. Autonomous exploration of unknown terrain for groups of mobile robots. In *Intelligent Vehicles Symposium (IV), 2011 IEEE*, pages 157–162. IEEE, 2011.

[9] José Alberto Méndez-Polanco and Angélica Munoz-Meléndez. Collaborative robots for indoor environment exploration. In *Control, Automation, Robotics and Vision, 2008. ICARCV 2008. 10th International Conference on*, pages 359–364. IEEE, 2008.

[10] Edwin Olson, Johannes Strom, Ryan Morton, Andrew Richardson, Pradeep Ranganathan, Robert Goeddel, Mihai Bulic, Jacob Crossman, and Bob Marinier. Progress toward multi-robot reconnaissance and the magic 2010 competition. *Journal of Field Robotics*, 29(5):762–792, 2012.

[11] Domènec Puig, Miguel Angel García, and L Wu. A new global optimization strategy for coordinated multi-robot exploration: De-

velopment and comparative evaluation. *Robotics and Autonomous Systems*, 59(9):635–653, 2011.

[12] Michael Melholt Quottrup, Thomas Bak, and RI Zamanabadi. Multi-robot planning: A timed automata approach. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 5, pages 4417–4422. IEEE, 2004.

[13] Pradeep Ranganathan, Ryan Morton, Andrew Richardson, Johannes Strom, Robert Goeddel, Mihai Bulic, and Edwin Olson. Coordinating a team of robots for urban reconnaisance. In *Proceedings of the Land Warfare Conference (LWC)*, 2010.

[14] Reid Simmons, David Apfelbaum, Wolfram Burgard, Dieter Fox, Mark Moors, Sebastian Thrun, and Håkan Younes. Coordination for multi-robot exploration and mapping. In *AAAI/IAAI*, pages 852–858, 2000.

[15] Håkon Skjelten. *Project work: Fjernnavigasjon av LEGO-robot.* NTNU, 2004.

[16] Jannice Selnes Tusvik. *Project work: Fjernstyring av legorobot.* NTNU, 2009.

[17] RWTH Aachen University. *RWTH-Mindstorms NXT Toolbox for MATLAB.* http://www.mindstorms.rwth-aachen.de/.

[18] Brian Yamauchi. Frontier-based exploration using multiple robots. In *Proceedings of the Second International Conference*

on Autonomous Agents, AGENTS '98, pages 47–53, New York, NY, USA, 1998. ACM.

[19] Øyvind Ulvin Halvorsen. *Project work: Collaborating robots.* NTNU, 2013.

[20] Robert Zlot, Anthony Stentz, M Bernardine Dias, and Scott Thayer. Multi-robot exploration controlled by a market economy. 2002.

Appendix A

Tables

Functionality	**Description**
`case {'simulatorset'}`	Allows the simulator to set the position variable `currentPos`.
`case {'removepointsnearcoopbot'}`	Remove measurement points near the other robot. For future use.
`case {'simulatorinit'}`	Sets which robot is simulated.
`case {'setinitoffset'}`	Sets the initial distance between the robots. This is used when the two maps are merged.
`case {'exchangedata'}`	Handles the communication between the robots. Sends and receives data, and stores the received data in the data structure *navData*.

Table A.1: Functions added to `RobotHandler`.

Function	Description
`mergeMaps`	Removes the initial offset between the individual robot maps.
`buildTacticalMap`	Builds a 3D probabilistic map based on the robot paths
`computeBoundary`	Calculates the discovered area, and its border
`optimizeExploration`	Removes infeasible border points, and post processes the border. Then calculates a position which minimizes the cost function.
`findBorderSegments`	Identifies border segment after border points near walls are removed. Sorts the border points in chronological order.
`mergeBorderSegments`	Merges border segments found in `findBorderSegments`, by checking if border segments are sufficiently close to each other.
`planPath`	Calculates the free configuration space, finds the shortest path from start to goal using a BFS-algorithm, and finally reduces number of way-points by only including the points where the robot has to turn.
`getDistBetween2segments`	Calculates the shortest distance between two line segments. Used in `planPath` when checking for collisions.

Table A.2: Functions implemented for the Collaboration module.

Action	Command
MOVE	0
Drive robot forward	00
Turn robot	01
MEASURE	1
Full scan	10
Single measurement sensor 1	11
Single measurement sensor 2	12
COM	2
DEBUG	3
PING	4
ACCESS	5
Reset: pose, tower angle, all	500,501,502
Get: pose, tower angle	510,511
Set: pose, tower angle	520,521

Table A.3: NXT state machine and sending protocol. Only the most common sub-actions are mentioned here. See NXTSLAM.nxc for the complete list. For curtain commands, additional data is sent together with the command. For instance the **MOVE** command also needs distance and power commands.

Appendix B

Equipment and set up

Computer: Dell Optiplex 9010

OS: Windows 7 Enterprise 64-bit

MATLAB version: 8.1.0.604 (R2013a) 64-bit

RWTH-Mindstorms NXT Toolbox version: 4.07–8. Februar, 2012

Bricx Command Center: Version 3.3

Firmware version: 1.31

Mindstorms drivers: Fantom version 1.1.3

USB-driver(NXT): libusb-win32

Bluetooth dongle: Targus micro USB BLUETOOTH 4.0 adapter

Bluetooth device for IR: Free2move and configuration software

USB cable: From LEGO Mindstorms

RS-232: serial connection for the IR-robot

NXT

The NXT-robot consisted of the following components.

- NXT 2.0 chip from LEGO Mindstorms
- Two EOPD[1]-sensors, NEO1048, from HiTechnic, range 20 cm.
- Three electrical motors from LEGO Mindstorms
- Construction kit from LEGO Mindstorms

The NXT-brick is programmed in NXC, and the IDE used for this brick is BricxCC.

In order to program the NXT-brick, open one of the files in *NXT SLAM*-directory, or a new file. Then the program can be compiled and downloaded to the brick. Only the main loop *NXTSLAM* needs to be compiled, the other files are included here.

The NXT-brick should have no problem connecting to BricxCC by USB, but if it has try updating the drivers (libusb-win32).

[1]Electro-Optical Proximity Detector

IR

The IR-robot consisted of the following components.

- Four GP2D12-sensors from Sharp
- One servomotor, HS-5925MG, from Hitec
- Two electrical motors, 71427, from LEGO
- Two OPIC photointerrupter, GP1A53HR, from Sharp
- Main board with ATMEGA 32 microprocessor from Atmel
- Motor control board
- Battery
- Free2move Bluetooth device
- Common LEGO parts

Bluetooth

The Bluetooth should be easily set up if the Bluetooth dongle is compatible with the system. Let windows install the drives automatically, and then the NXT-brick and the 'Free2move'-device (for the IR-robot) can be added to Hardware and Devices in Windows. The 'Free2move'-device will require configuration software in order to run, which can be found on `http://web.free2move.se/`.

If the IR-robot performs several full scans in a row, this is caused by a weak Bluetooth connection, and not a software error. This is because the IR-robot has to be told to stop the procedure.

Running the program

In order to run the system, start `LegoGUI_IR` or`LegoGUI_NXT`. If both GUIs are started, it must be done from separate MATLAB instances.

In the GUI, choose 'Simulator' or 'RobotHandler' under the connection options and press 'Connect', or 'Start' if the the main loop shall be started right away.

The map will then appear in the main plotting window, and more plots can be found under 'Plotting Iterations'.

The simulator should run without no further set up. The only issue is to find a valid starting point for the robot.

Appendix C

CD

Attached to this thesis is a CD with the following contents.

- MATLAB code
- NXC code for the NXT-brick
- This report on pdf-format
- A demonstration video of running the robots
- Previous works