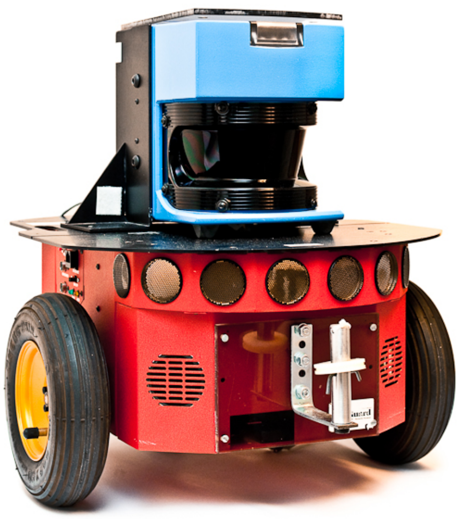
Technical Documentation
AB Mail Robot

Version 1.0

Author: Martin Melin

Date: December 16, 2010



Course name:	Control Project	E-mail:	danba185@student.liu.se
Project group:	AB Mail Men	Document responsible:	Martin Melin
Course code:	TSRT10	Author's E-mail:	marme287@student.liu.se
Project:	AB Mail Robot	Document name:	TechnicalDocumentation_v10.pdf

Status

Reviewed	Karl Granström André Carvalho Bittencourt	2010-12-15
Approved	Karl Granström	2010-12-15

Project Identity

Group E-mail:	danba185@student.liu.se
Homepage:	http://www.isy.liu.se/edu/projekt/tsrt10/2010/postrobot-2010/
Orderer:	Karl Granström, Linköping University
	Phone: +46 13 281333, **E-mail:** karl@isy.liu.se
Customer:	Marcus Pettersson, ABB Västerås
	Phone: +46 21 345194, **E-mail:** marcus.pettersson@se.abb.com
Course Responsible:	David Törnqvist, Linköping University
	Phone: +46 13 281882, **E-mail:** tornqvist@isy.liu.se
Project Manager:	Daniel Barac
Advisors:	André Carvalho Bittencourt, Linköping University
	Phone: +46 13 282622 , **E-mail:** andrecb@isy.liu.se

Group Members

Name	Responsibility	Phone	E-mail (@student.liu.se)
Daniel Barac (DB)	Project Leader	+46 702641857	danba185@student.liu.se
Martin Melin (MM)	Document responsible	+46 702887793	marme287@student.liu.se
Erik Erkstam (EE)	Software responsible	+46 737266812	erier982@student.liu.se
Hugo Kinner (HK)	Test responsible	+46 733688378	hugki634@student.liu.se
Manfred Hallström (MH)		+46 703679569	manha932@student.liu.se
Simon Hugosson (SH)	Control responsible	+46 708778471	simhu610@student.liu.se
Niklas Carlsson (NC)	Information responsible	+46 709601672	nikca291@student.liu.se
Nicklas Forslöw (NF)	SLAM responsible	+46 739516430	nicfo307@student.liu.se

Document History

Version	Date	Changes made	Sign	Reviewer
0.1	2010-12-01	First draft	SH MM EE MH NC NF DB HK	SH MM EE MH NC NF DB HK
0.2	2010-12-03	Second draft	SH MM EE MH NC NF DB HK	EE MM MH NF
0.3	2010-12-07	Third draft	SH MM EE MH NC NF DB HK	DB
0.4	2010-12-12	Fourth draft	SH MM EE	DB
0.5	2010-12-15	Fifth draft	MM SH EE	DB
1.0	2010-12-15	First version	DB	DB

Contents

1 Introduction

Fido-Dido (Figure 1) is a wheeled mobile robot manufactured by MobileRobots Inc. Its most important features are a SICK laser range finder, 8 forward-facing ultrasonic array sonars, odometers and a Micro-Controller. Fido is able to autonomously navigate in an office-landscape using a pre-defined map which it updates when changes are detected. Fido is also able to find its way from point A to point B.

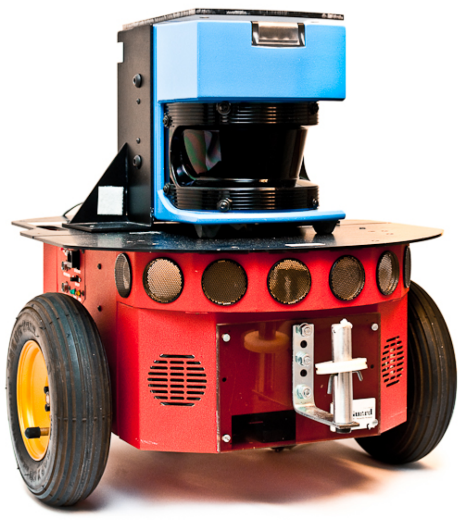


Figure 1: Fido-Dido

This document provides a detailed description of how the software and hardware is used to achieve the goals of the project, specified in Section 1.2.

1.1 Parties

The client of the project is Karl Granström, PhD-student in Automatic Control, Department of Electrical Engineering at Linköping University. Supervisor of the project is André Carvalho Bittencourt, PhD-student in Automatic Control, Department of Electrical Engineering at Linköping University. Customer of the project is Marcus Pettersson at ABB Cooperate Research Center (ABB CRC). The development is performed by the project group ABB MailMen.

1.2 Purpose and goal

The purpose of the project was to develop software and then implement it on a robot so that the robot can navigate in an office environment. The project group has evaluated future possibilities to make this robot able to deliver objects, e.g. mail. The goal of the project was to develop a system for an autonomous robot which shall be able to navigate from point A to point B in an office environment, specifically at ABB CRC. The long term goal of the product is to extend the robot with a robotic arm to enable the delivery of objects, e.g. mail.

1.3 Usage

The product can be used as a platform for further research at ABB CRC or at Linköping University.

1.4 Background information

This project is a part of the CDIO-course TSRT10 at Linköping University and is performed in collaboration with ABB CRC. The group consists of eight students with expertise within automatic control and sensor fusion.

1.5 Scope

The robot is limited to operate in an office environment. The robot cannot be expected to be able to open doors or move objects. The floor on which the robot will operate must be free from obstacles which the robot cannot detect.

2 Fulfilled requirements

This section lists fulfilled requirements relevant to the technical documentation, as described in the requirements specification, and gives references to where in the technical documentation the functionality of the requirement in question is described. The requirements will be listed as follows:

Requirement no.	Requirement	Chapter/-s

The column "Requirement no" states the requirement number, the column "Requirement" states the requirement and the column "Chapter/-s" states the chapter or chapters in which the implementation of the requirement is described.

2.1 Software requirements

This section lists the software requirements as described in the requirements specification.

2.1.1 Mapping requirements

This section lists fulfilled requirements regarding the mapping.

Requirement 1	The software shall be able to build a map by using sensor data from an unexplored environment.	5.2
Requirement 2	The software shall be able to update a map when new obstacles are detected and when old obstacles are moved or removed.	5.4
Requirement 4	The software shall be able to build a map from environments that are flat.	5.2
Requirement 5	There shall be a possibility to load a previously constructed map.	7

2.1.2 Trajectory planning requirements

This section lists fulfilled requirements regarding the trajectory planning.

Requirement 46	The software shall be able to plan a route from one given position in the map to another position in the map. The robot shall be able to follow the route without hitting objects close to its path.	6
Requirement 11	Whenever a new fixed obstacle is detected, the software shall be able replan the route if necessary.	7 6.4

2.1.3 Navigation and localization requirements

This section lists fulfilled requirements regarding navigation and localization.

Requirement 15	Given an approximate map and an initial pose in the map, the software shall be able to estimate the robot pose in the map when the robot is moving around.	5.3.1

Requirement 16	The software shall be able to give some indication of the uncertainty in the robots pose.	4 5.3.1

Requirement 17	The software shall be able to estimate the robot's pose in environments that are flat.	5.3.1

2.1.4 Other requirements in the software

This section lists other fulfilled requirements in the software.

Requirement 20	There shall be a possibility to maneuver the robot using the onboard laptop.	3.1.2

Requirement 21	The software shall be written in C++ or MATLAB.	3.1.2

2.2 Robot requirements

This section lists fulfilled requirements regarding the robots physical movement.

Requirement 28	Given the initial position on the route, the robot shall be able to follow the route and avoid objects by slowing down when closer than 60cm from the center of the robot, and stopping when closer than 30cm.	5 6 7

Requirement 30	If the end position is arbitrary, the robot shall be able to stop within 4 decimeters from it. The error of the robot heading is insignificant.	5 6 7

2.3 Requirements regarding further development

This section lists fulfilled requirements regarding further development.

Requirement 41	Suggestions regarding the implementation of IRB 120 robot shall be discussed in the technical documentation	9

3 Overview of the System

The system consists of a MobileRobots P2-DX robot connected to a HP ProBook 6450b laptop. The laptop runs the application that controls the robot and can also be used as an interface if a user wants to manually control the robot or view the robot's map. The robot and the laptop communicate via a RS232 serial port. The different parts of the system and how they communicate can be viewed in Figure 2. The laser and sonars send range measurements and the odometers send wheel position and speed estimates. The micro controller (MC) decodes the data from the sonars and odometers and passes it on to the program. The MC also translates the motion commands from the program to control signals for the motors. When collecting data for Offline-SLAM, see Section 5, the user drives the robot with the laptop's arrow keys. When mapping is done, the user can tell the program where the robot should go. The output from the program to the user is a map and the robot's current position in the map.

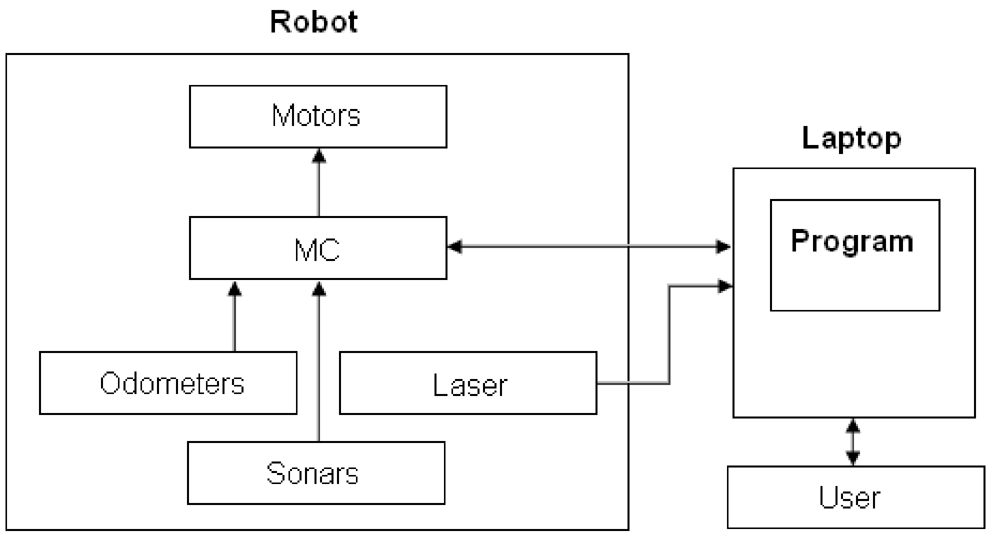


Figure 2: Overview of the system.

3.1 Subsytems

This section includes a short description of the subsystems.

3.1.1 Robot

The robot is a wheeled MobileRobots P2-DX model equipped with odometers, a laser, sonars and a micro controller (MC). The micro controller receives data from the sonars and odometers. It also sends control signals, calculated by the software on the laptop, to the servos.

3.1.2 Laptop

The laptop receives sensor data from the laser and the robot's micro controller. It is also responsible for the software which handles SLAM, trajectory planning and calculation of the control signals.

The application is written in C++ and uses the ActivMedia Robotics Interface for Application (ARIA) library for all communication with the robot. ARIA has utilities for both receiving data from the sensors and controlling the motors. The program can be divided into three subsystems with specific tasks according to Figure 3. These subsystems are thoroughly explained in Sections 5 to 7. The SLAM system receives range measurements from the laser. The control system receives a pose estimate from the odometers and passes it on to the SLAM system together with a calculated covariance matrix. The SLAM system sends a grid map and an improved pose estimate to the trajectory planner which plans a trajectory to the goal and sends the next desired position to go to, to the control system. It, in turn, sends the desired wheel velocities to the motors.

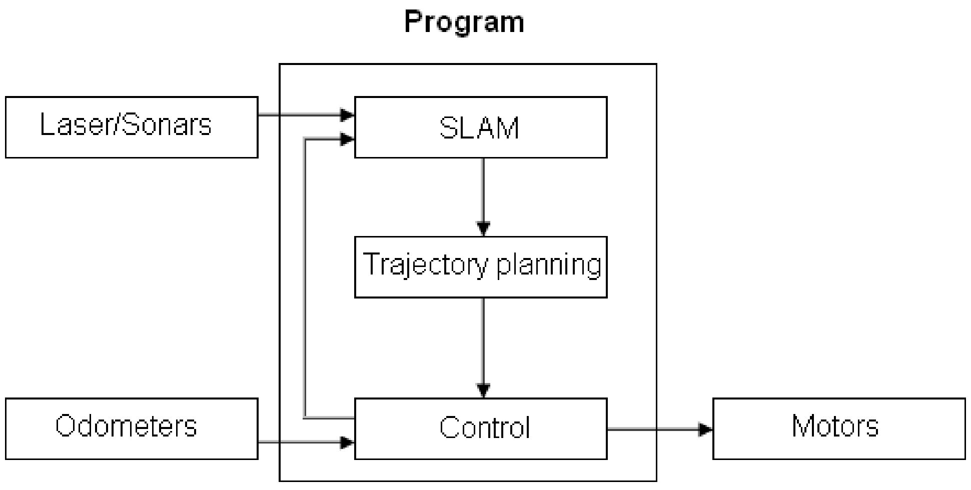


Figure 3: Program structure.

4 Pose Uncertainty

The robot pose is given by the following vector, where x and y are the position coordinates and θ is the heading direction, $X_i = (x_i, y_i, \theta_i)^T$. To estimate the uncertainty of each pose, the uncertainties of the odometers are taken into consideration. The uncertainty of a pose X_i, given the previous pose X_{i-1}, is described as $P(X_i|X_{i-1}) = u_i^T \cdot \Sigma$, where u_i is the motion command defined as $u_i = X_i - X_{i-1} = (\Delta_x^{odo}, \Delta_y^{odo}, \Delta_\theta^{odo})^T$, view Figure 4, and Σ is a diagonal matrix with the variances of $\Delta_x^{odo}, \Delta_y^{odo}$ andΔ_θ^{odo} in the diagonal, defined in equation 1.

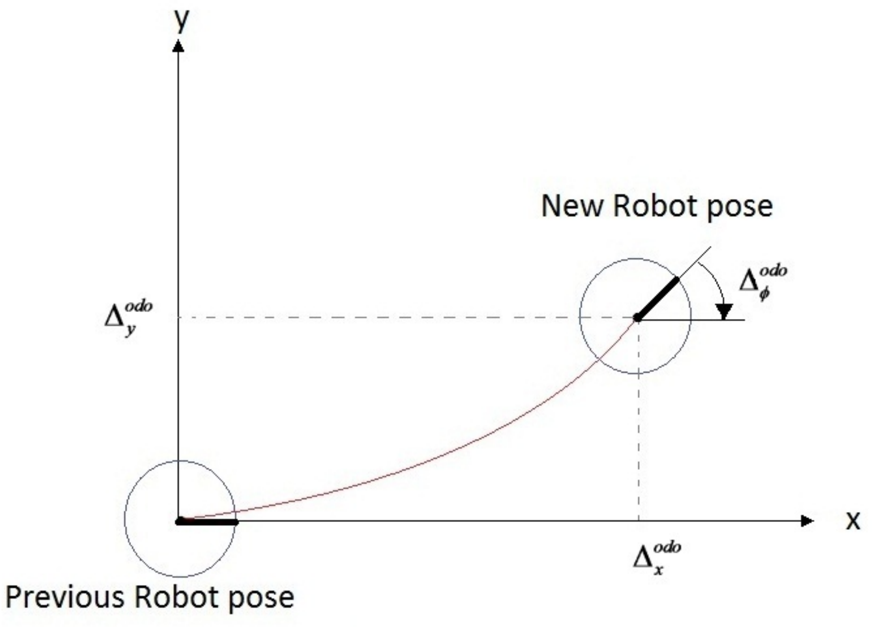


Figure 4: Model of motion command u_i

The distribution of the new pose is considered to have a mean equal to the odometer movement added to the previous pose. To obtain the variance of each odometer, the initial approximation $\sigma_{\Delta_x^{odo}} = 0.1$, $\sigma_{\Delta_y^{odo}} = 0.1$ and $\sigma_{\Delta_\theta^{odo}} = 0.15$ was used, e.g. after driving 1 meter in the x direction, the standard deviation would be 10cm in x. This approximation was used while implementing the SLAM and Pathfinding algorithms and proved to give good results in pose accuracy. Different approaches were used to improve this accuracy, for example was the motion model described in [3] used, but without any improvements. Therefore the final variance that was used was also the initial guess.

$$\Sigma = \begin{pmatrix} (\sigma_{\Delta_x^{odo}})^2 & 0 & 0 \\ 0 & (\sigma\Delta_y^{odo})^2 & 0 \\ 0 & 0 & (\sigma\Delta_\theta^{odo})^2 \end{pmatrix} . = \begin{pmatrix} 0.1^2 & 0 & 0 \\ 0 & 0.1^2 & 0 \\ 0 & 0 & 0.15^2 \end{pmatrix} . \qquad (1)$$

4.1 Future development regarding pose uncertainty

Further development and tests regarding the pose uncertainty is advised. The use of a motion model is strongly recommended to increase pose accuracy. To set the covariances to zero as done above is a major simplification since a standard deviation in θ effects the standard deviations of x and y.

5 SLAM

This subsystem is responsible for the simultaneous localization and mapping (SLAM). Laser scan measurements are used to estimate the robot position and heading in a map and to update the map as changes are observed. Here, sensor models, odometry and laser scan measurements are used.

The system contains two different SLAM-algorithms which are called Offline-SLAM and Online-SLAM. The purpose of Offline-SLAM is to create an accurate map of the environment in which the robot will operate. This algorithm is too computationally demanding to be used in real time and therefore Online-SLAM is needed for real time applications. Online-SLAM runs in real time and it uses a map built with the help of Offline-SLAM in which it localizes the robot.

Offline-SLAM only uses data from the odometers and the laser. Therefore, the resulting map only contains "what the laser sees". When the map is built it needs to be converted for computational reasons. The converted map is a line based map which Online-SLAM uses for localization. The idea is that the line based map is a filtered version of the map arising from Offline-SLAM in the way that it manly contains larger objects such as walls. Because these kind of objects are not likely to be moved, removed or added, Online-SLAM can use the line based map to achieve good localization without updating it.

However, the line based map cannot be used for navigation. The reason for this is that the robot has to know about any obstacles that it could collide with and it is not likely that the line based map contains this information. Therefore an occupancy grid map is built and used for navigation. Initially the grid map is constructed from the data generated by the Offline-SLAM algorithm.This map is then updated in real time in the Online-SLAM algorithm using measurements from the laser.

How these tasks are solved and what sensor models are used is described below.

5.1 Sensor Models

This section describes what data is delivered by each type of sensor and how this data can be related to a global coordinate system. To derive the sensors locations in the global coordinate system there is a need to define the robot's local coordinate system. If a position is given in the robot's local coordinate system as $(p_x^r, p_y^r)^T$ the corresponding global position $(p_x, p_y)^T$ is given by

$$\begin{pmatrix} p_x \\ p_y \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} p_x^r \\ p_y^r \end{pmatrix} \tag{2}$$

where $X = (x, y, \theta)^T$ is the pose of the robot. This operation is denoted by

$$(p_x, p_y)^T = X \oplus (p_x^r, p_y^r)^T. \tag{3}$$

5.1.1 Laser Scans

The laser sensor delivers scans of the enviroment at a rate of about 4 Hz. One laser scan is a sequence $E = (e_0, e_1, ..., e_{360})$, where e_i is a distance measurement taken from the robot's position with an angle ψ_k relative to the robot's heading. The angle ψ_k is given by $\psi_k = -90 + 0.5k$. If the robot's pose is $X = (x, y, \theta)^T$ and the position of the laser sensor is $(0, 0)$ in the robot's local coordinate system, then each measurement e_i

corresponds to a physical position $(u_x, u_y)_i$ by

$$(u_x, u_y)_i = (x, y) + e_i(\cos(\psi_i + \theta), \sin(\psi_i + \theta)) \tag{4}$$

and a scan E corresponds to a set of positions $U = \bigcup_{k=0}^{360} (u_x, u_y)_i^k$.

5.2 Building the Map

This section describes how the map of the environment, in which the robot will operate, is built. It is divided into three subsections: Data Collection, Offline-SLAM and Map Conversion.

5.2.1 Data Collection

The robot is manually maneuvered in some area and at certain moments data are saved. The saved data contain an estimate of the relative motion since the previous saved point as well as laser measurements. Instead of saving data at certain points in time, data is only saved when the robot's pose has changed significantly. New data is saved if new laser measurements are collected and if the odometers estimate that the robot has traveled a distance greater than some distance d, or that the robot has changed its heading with an angle greater than some angle β. The values of d and β are set experimentally. The reason why data is saved in this manner is because there is no use of saving a lot of data if the robot does not change its pose.

5.2.2 Offline-SLAM

Since the laser scans are very accurate, the problem of building an accurate map from laser scans is the problem of estimating the poses from which the scans were taken as accurately as possible. Therefore, one can consider the following estimation problem.

Assume that the robot has taken $n + 1$ laser scans. Let $X_i = (x_i, y_i, \theta_i)^T$ describe the robot position and heading when scan i was aquired and simply let $X_0 = (0, 0, 0)^T$. Let the relative pose change of X_i seen from X_j

$$D_{ij} = X_i \ominus X_j := \begin{pmatrix} \cos\theta_j & \sin\theta_j & 0 \\ -\sin\theta_j & \cos\theta_j & 0 \\ 0 & 0 & 1 \end{pmatrix} (X_i - X_j) \tag{5}$$

be the measurement equation. An observation $\overline{D}_{ij}$ of D_{ij} is defined as

$$\overline{D}_{ij} = D_{ij} + \Delta D_{ij} \tag{6}$$

where ΔD_{ij} is a random Gaussian noise with zero mean and known covariance matrix C_{ij}. Assume that there is a set S consisting of pairs (i, j), where $0 \leq i < j \leq n$ such that there is an observation of $\overline{D}_{ij}$ with a corresponding covarince matrix C_{ij} for each pair $(i, j) \in S$.

Then a criterion of optimal estimation can be formed based on the *maximum likelihood* concept. By assuming that the observation errors are mutually independent, the criterion is to minimize the following Mahalanobis distance

$$W = \sum_{(i,j)\in S} (D_{ij} - \overline{D}_{ij})^T C_{ij}^{-1} (D_{ij} - \overline{D}_{ij}) \tag{7}$$

If one has an initial estimate

$$\hat{\mathbf{X}} = (\hat{X}_0, \hat{X}_1, .., \hat{X}_n)^T \tag{8}$$

of

$$\mathbf{X} = (X_0, X_1, .., X_n)^T \tag{9}$$

an approximate solution of (7) is given by linearization. The solution, which can be found in [5], yields a new approximation $\hat{\mathbf{X}}$ of $\mathbf{X}$.

The algorithm, which is developed with inspiration from [5], contains two different kinds of observations, called weak and strong links. A weak link is an estimate, taken from the odometry, of the relative pose difference between two adjacent poses. While weak links are taken from the motion model, strong links are more sophisticated. The idea is to find pairs of poses with scans that are strongly correlated, i.e pairs of poses from which the laser detects the same physical objects has to be found. By matching the scans with a scan matching algorithm, an estimate of a relative pose difference can be obtained.

The algorithm makes use of the above mentioned ideas in two steps. From the motion model we have a set of weak links as

$$\overline{\mathbf{D}}_{\mathbf{W}} = (\overline{D}_{W,1}, \overline{D}_{W,2}, .., \overline{D}_{W,n}) \tag{10}$$

where

$$\overline{D}_{W,i} \approx X_i \ominus X_{i-1}. \tag{11}$$

In the same manner, we define the set of $m(n)$ strong links as

$$\overline{\mathbf{D}}_{\mathbf{S}} = (\overline{D}_{S,(i,j)_1}, \overline{D}_{S,(i,j)_2}, .., \overline{D}_{S,(i,j)_{m(n)}}) \tag{12}$$

where

$$\overline{D}_{S,(i,j)_l} \approx X_{j_l} \ominus X_{i_l} \tag{13}$$

with $j_l \neq i_l$ and $j_l, i_l \in (1, 2, ..., n)$.

For $k \leq n$, let

$$\hat{\mathbf{X}}_{\mathbf{k}} = (\hat{X}_0, \hat{X}_1, .., \hat{X}_k)^T \tag{14}$$

be an approximation of

$$\mathbf{X} = (X_0, X_1, .., X_k)^T \tag{15}$$

and let

$$\overline{\mathbf{D}}_{\mathbf{W_k}} = (\overline{D}_{W,1}, \overline{D}_{W,2}, .., \overline{D}_{W,k}) \tag{16}$$

and

$$\overline{\mathbf{D}}_{\mathbf{S_k}} = (\overline{D}_{S,(i,j)_1}, \overline{D}_{S,(i,j)_2}, .., \overline{D}_{S,(i,j)_{m(k)}}). \tag{17}$$

The main idea of both of the algorithms is as follows

I Initialize $\hat{\mathbf{X}}_{\mathbf{k}}$, $\overline{\mathbf{D}}_{\mathbf{W_k}}$ and $\overline{\mathbf{D}}_{\mathbf{S_k}}$ as $\hat{\mathbf{X}}_{\mathbf{k}} = (0,0,0)^T$ and $\overline{\mathbf{D}}_{\mathbf{W_k}} = \overline{\mathbf{D}}_{\mathbf{S_k}} = \emptyset$.

II Update $\hat{\mathbf{X}}_{\mathbf{k}}$ as $\hat{\mathbf{X}}_{\mathbf{k}} = (\hat{X}_0, \hat{X}_1, ..., \hat{X}_{k-1} \oplus \overline{D}_{W,k})$ and $\overline{\mathbf{D}}_{\mathbf{W_k}}$ as $\overline{\mathbf{D}}_{\mathbf{W_k}} = (\overline{\mathbf{D}}_{\mathbf{W_{(k-1)}}}, \overline{D}_{W,k})$.

III For $i = 0, 1..., k-1$ check if there is a possible strong link between $\hat{X}_i$ and $\hat{X}_k$. For every such pair, do scan matching to achieve a new strong link and add this link to $\overline{\mathbf{D}}_{\mathbf{S_k}}$. How the detection of the strong links is done and how the scan matching works are explained in the next two sections.

IV At this stage (7) is formed for the first $k+1$ poses with the two sets of observations $\overline{\mathbf{D}}_{\mathbf{W_k}}$ and $\overline{\mathbf{D}}_{\mathbf{S_k}}$. It is then solved by linearization around $\hat{\mathbf{X}}_{\mathbf{k}}$. The solution gives a new estimate of $\hat{\mathbf{X}}_{\mathbf{k}}$.

V If there are any weak links left to add, go back to step II. Otherwise, the algorithm is finished.

However, considering that the order of the detection of strong links is approximately n, one might realize that iterating through this algorithm will take too long time if n is large. Consider again the detection of strong links, the order of iterating through the algorithm is then approximately $n!$. Therefore the algorithm is separated into two steps.

In the first step the data given by the robot is separated into equally big pieces. The first piece consists of the initial pose $(0,0,0)^T$ together with the first 5 weak links and corresponding measurements. The above described algorithm is run on this data and results in an improved estimate of the first 6 poses. The second piece of data then consists of weak links number 6 to 10, but at this stage the initial pose is set to the last pose generated by running the algorithm on the first piece. Together with the corresponding measurements the algorithm is run again. In this way a set of pieces is generated where the last pose in a piece is the first pose in the next piece. This algorithm is, of course, only of order n. A piece P_i containing 6 poses and corresponding measurements can mathematically be described as

$$P_i = \{(\hat{X}_{5i}, M_{5i}), (\hat{X}_{5i+1}, M_{5i+1}), ..., (\hat{X}_{5i+5}, M_{5i+5})\} \tag{18}$$

where $M_i = \bigcup_{k=0}^{360} (m_x, m_y)_i^k$ denote the set of points in the scan taken from X_i relative to X_i.

This algorithm generates pieces that are of high quality on their own, but it might still be the case that the approximation of the relative pose between the first poses in two pieces generated far after each other are of low quality. For every piece one can get good estimates of the relative positions between the measurements in the piece and the first pose in the piece. By considering the first pose in a piece as the representive pose X_i^p of the piece

$$X_i^p = X_{5i} \tag{19}$$

and the set of measurements M_i^p in a piece relative to X_i^p

$$M_i^p = \bigcup_{k=0}^{4} (X_{5i+k} \oplus M_{5i+k}) \ominus X_{5i} \tag{20}$$

of the piece as the representive measurements of the piece, one can again run the algorithm on the set of pieces in order to increase the quality of the relative poses between the pieces. An illustration of two adjacent pieces are shown in Figure 5. Since the last pose in a piece is the first pose in the next, the set of weak links $\overline{\mathbf{D}}_{\mathbf{W}}^{\mathbf{p}}$ is now naturally given as these relative poses

$$\overline{\mathbf{D}}_{\mathbf{W}}^{\mathbf{p}} = (\overline{D}_{W,1}^p, \overline{D}_{W,2}^p, ..) \tag{21}$$

where

$$\overline{D}_{W,i}^p = \hat{X}_i^p \ominus \hat{X}_{i-1}^p = \hat{X}_{5i} \ominus \hat{X}_{5(i-1)} \tag{22}$$

.

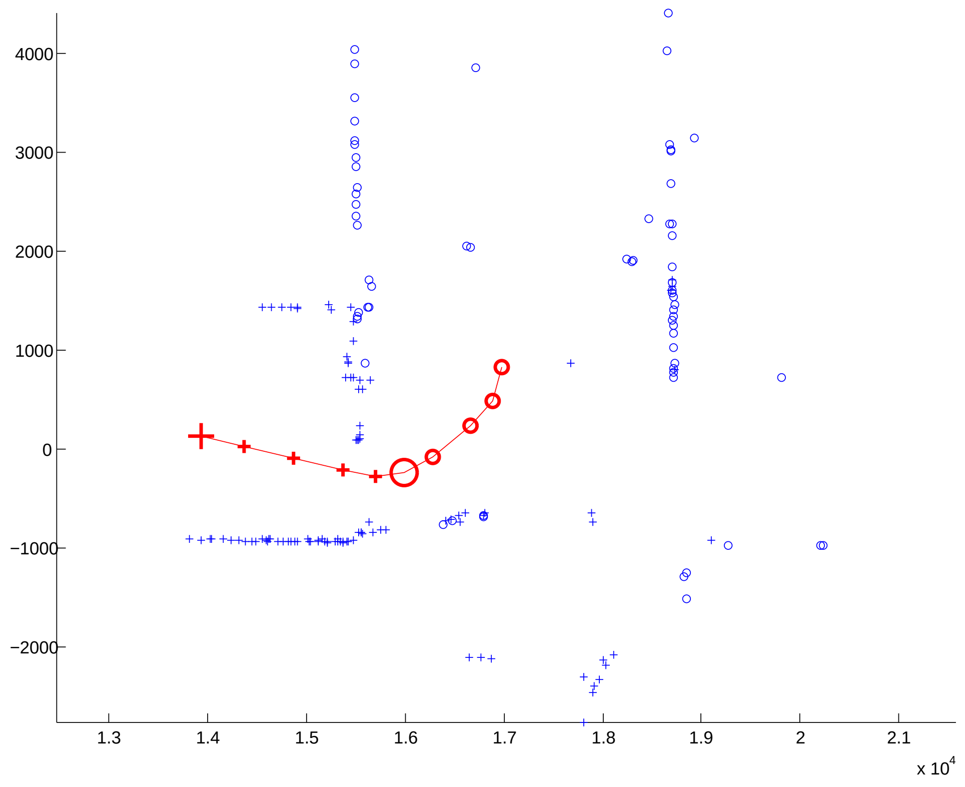


Figure 5: Illustration of two adjacent pieces. The big "+" and "∘" represents two poses X_i^p and X_{i+1}^p. Also shown in the figure are the other poses and the measurements that generates the whole pieces. Notice that a lot of measurements are thrown before generating this figure. This was made to make it possible to distinguish the two pieces. The axes are given in millimeters.

The covariance matrices for the weak links in the first step of the algorithm are given by the motion model. In the second step, the covariance matrices for the weak links are set experimentally as a diagonal-matrix C_W^p

$$C_W^p = \begin{pmatrix} 0.01^2 & 0 & 0 \\ 0 & 0.01^2 & 0 \\ 0 & 0 & (0.1\pi/180)^2 \end{pmatrix} \tag{23}$$

and so are the covariance-matrices for the strong links in both steps of the algorithm

$$C_S = C_S^p = \begin{pmatrix} 0.05^2 & 0 & 0 \\ 0 & 0.05^2 & 0 \\ 0 & 0 & (0.5\pi/180)^2 \end{pmatrix}. \tag{24}$$

The reason of this is that it is hard to get a good estimate of the covariance matrices from the scan matching algorithm that is used. However, if one could get a good estimate of the covariance matrices from the scan matching algorithm, using these would most likely improve the result. The variances in the covariance matrices are given in $[m^2]$ and $[\text{rad}^2]$.

The result of the algorithm run at a data set is shown below. In Figure 6 the map generated from odometry is shown.

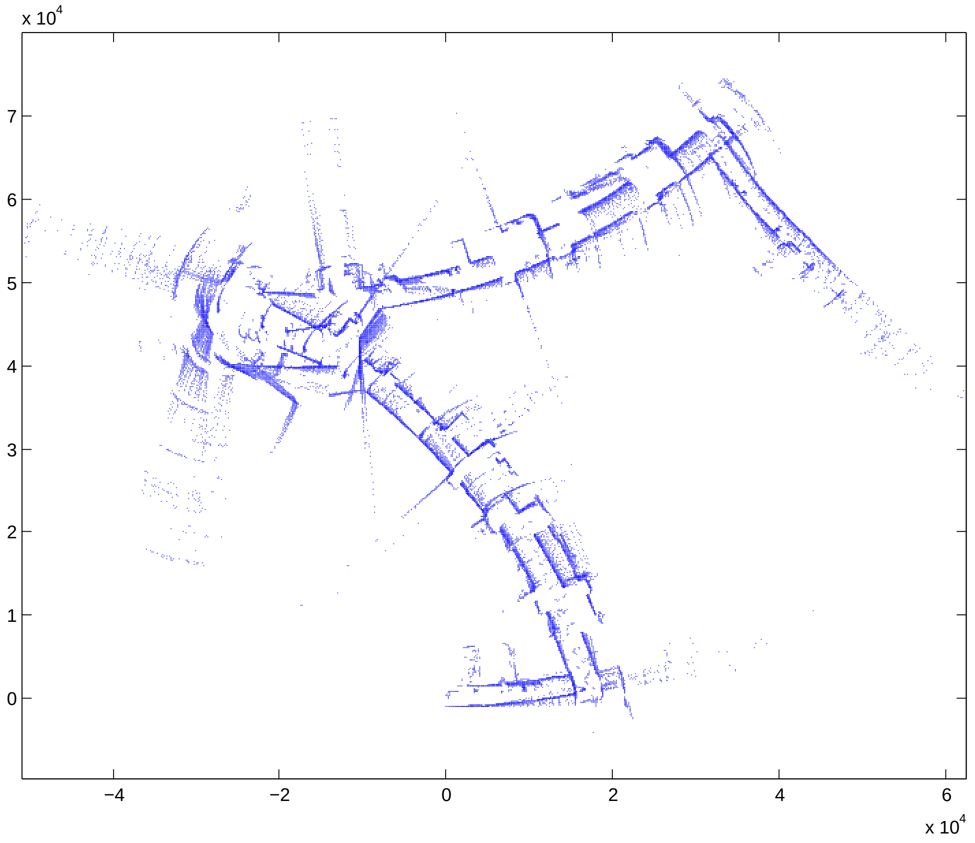


Figure 6: Map generated from the odometry. The axes are given in millimeters.

In Figure 7 a comparsion of the resulting map from the first and second step is shown. The robot has travelled from the bottom left, up and then back again when collecting the data. In Figure 8 one can see the improvement made when closing the loop in the second step of the algorithm. Notice that these maps are generated from the data set which was used to set all design variables. The axes are given in millimeters.

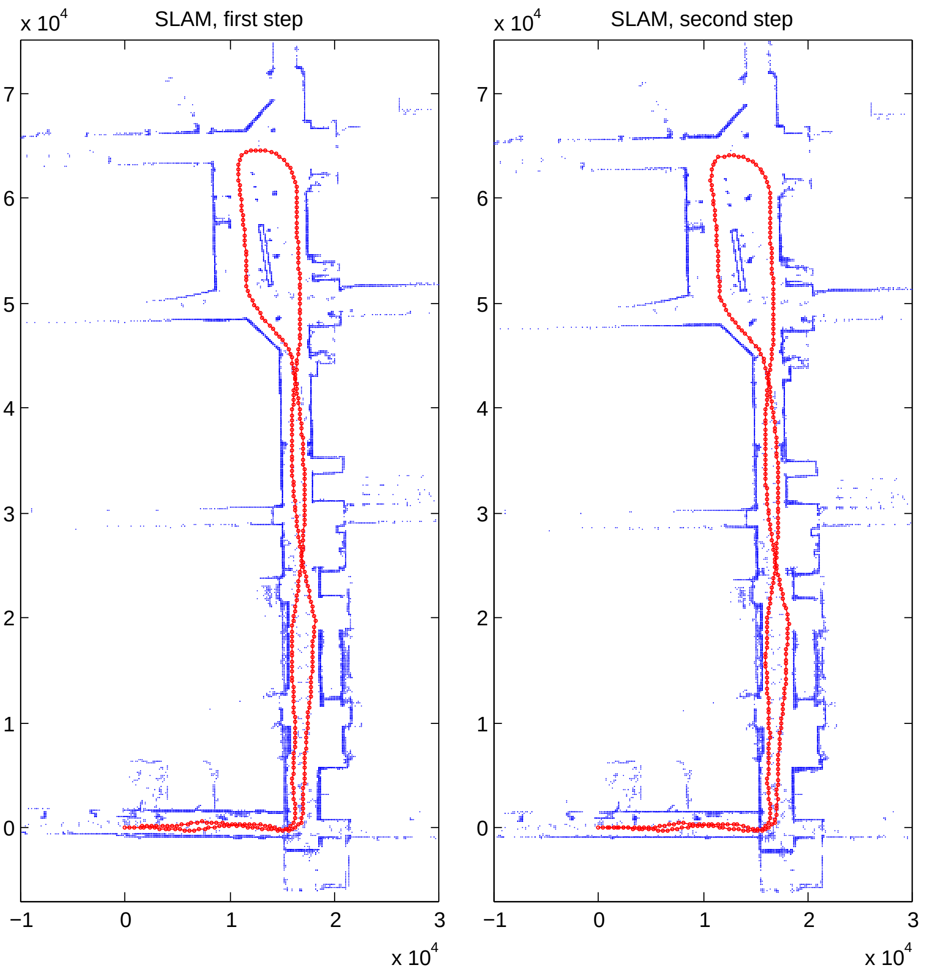
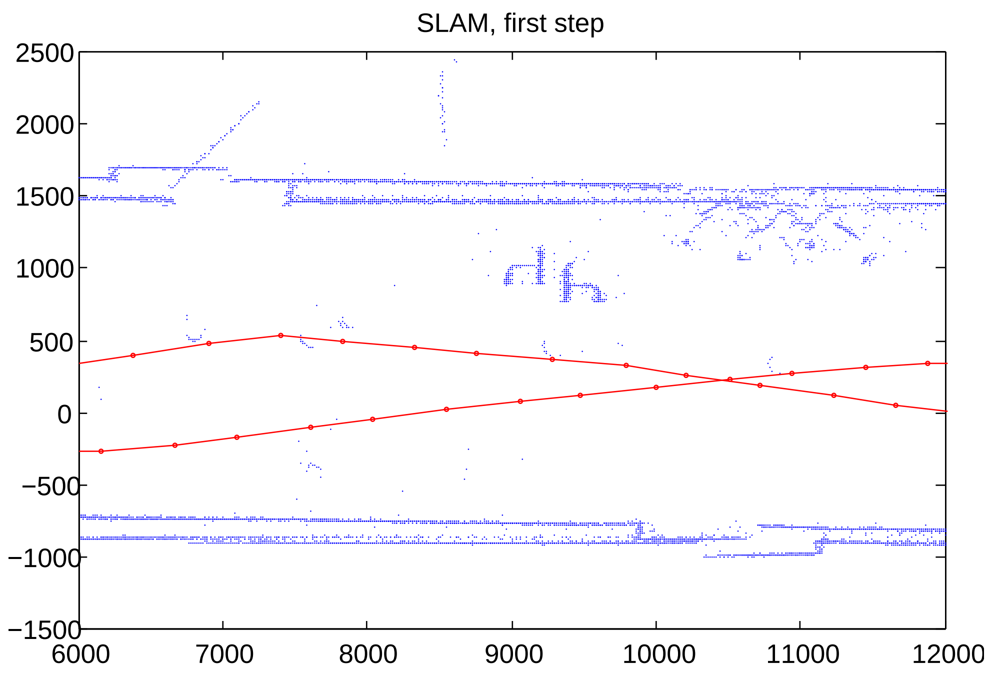


Figure 7: To the left, the resulting map after the first step. To the right, the resulting map generated after the second step. The axes are given in millimeters.



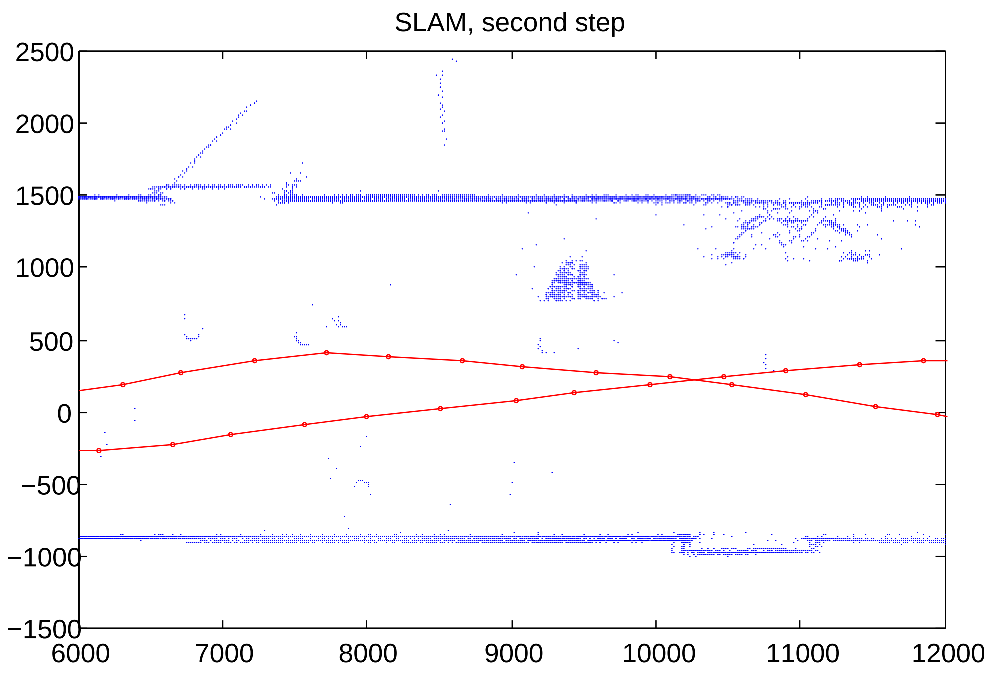


Figure 8: Closeup of the generated maps, first step on top and second step at the bottom. The axes are given in millimeters.

5.2.3 Detection of Strong Links

The detection of the strong links is a key to estimate the poses well. The question is how to detect that there is a a strong link between two poses. Hence, one must find conditions such that if they are fulfilled, it is likely that the laser scans overlap. The fact that the laser scans only cover a field of view of 180 degrees limits the ways of detecting strong links. However, the following idea will handle the problem of the limited field of view.

In this section translations are given in millimeters and angles are given in radians. Let $U_i = \bigcup_{k=0}^{N}(u_x, u_y)_i^k$ denote the set of points in the scan taken from $X_i = (x_i, y_i, \theta_i)^T$. To decide if there is a strong link $i \leftrightarrow j$ the following will be done. Define a set W_i as

$$W_i = \{(x,y) = (x_i, y_i) + r(\cos\alpha, \sin\alpha) : r_{min} \le r \le r_{max}, |\theta_i - \alpha| \le \varphi\} \tag{25}$$

where

$$r_{min} = (1 - r_0) \cdot \min_{0\le k\le N} ||(x_i, y_i) - (u_x, u_y)_i^k||_2 \tag{26a}$$

$$r_{max} = (1 + r_1) \cdot \min\big(\max_{0\le k\le N} ||(x_i, y_i) - (u_x, u_y)_i^k||_2, r_2\big) \tag{26b}$$

$$\varphi = \pi/2 + \varphi_0 \tag{26c}$$

Let $Q_{ij} = \#(u_x, u_y) \in U_j \bigcap W_i$. Define Q_{ji} in the same way and regard the link between i and j as a strong link if and only if $\min(Q_{ij}, Q_{ji}) > Q$. If the link is strong, $U_j \bigcap W_i$ and $U_i \bigcap W_j$ is used for scan matching.

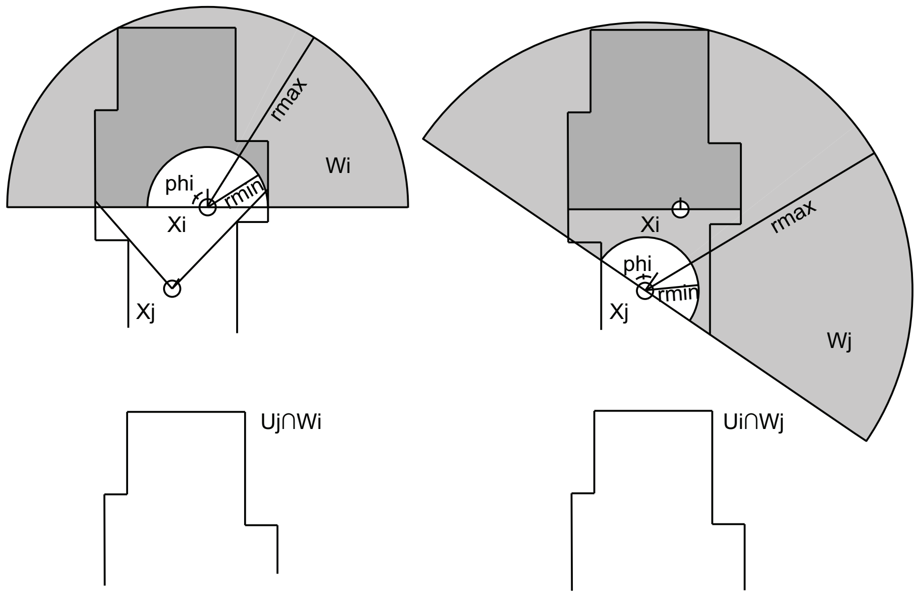


Figure 9: Detection of strong link

By setting $r_2 = \infty$, the value of r_{max} becomes

$$r_{max} = (1 + r_1) \cdot \max_{0\le k\le N} ||(x_i, y_i) - (u_x, u_y)_i^k||_2$$

which also was the initial idea. An illustration of this, with r_0, r_1 and φ_0 all set to zero, is shown in Figures 9 and 10. The purpose of the parameters r_0, r_1 and φ_0 is because the pose X_i can only be estimated. With help of the illustrations one might realize that if the estimations are bad a strong link might be missed if the parameters are set to zero. By increasing the values of these three parameters, the sets W_i and W_j will grow and the risk of missing a strong link decreases. However, increasing them too much could result in a false detection.

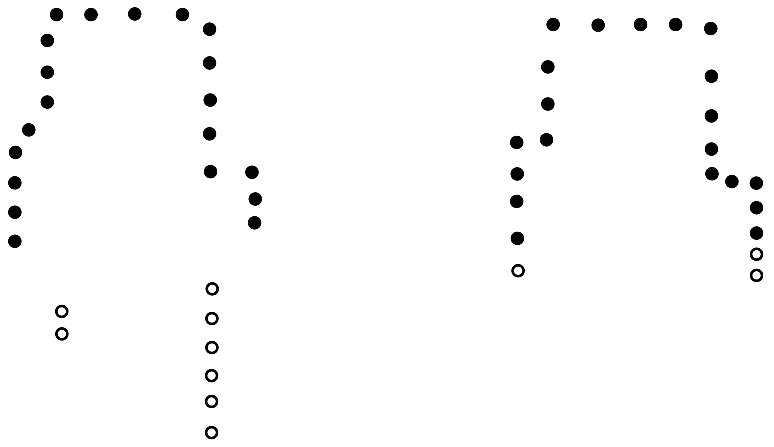


Figure 10: Illustration of U and $U \bigcap W$. The circles represent the measured points U. The filled circles represent the points that are counted and eventually used for scan matching $U \bigcap W$. To the left is U_j and $U_j \bigcap W_i$ and to the right is U_i and $U_i \bigcap W_j$.

The fact that the operation area of the robot is office landscapes, which often have long corridors and sometimes windows, lead to the need defining r_{max} as in (26). In this kind of environment most scans include a smaller set of points that are very far away from the robot's position. This means that the size of W_k for an arbitrary $k \in \{0, 1, ..., n\}$ is most likely very large if $r_2 = \infty$, which in turn leads to a lot of false detections. The values of the parameters are $r_0 = 0.1$, $r_1 = 0.2$, $r_2 = 10000$ and $\varphi_0 = \pi/10$. Results from running the algorithm on real data are shown in Figure 11.

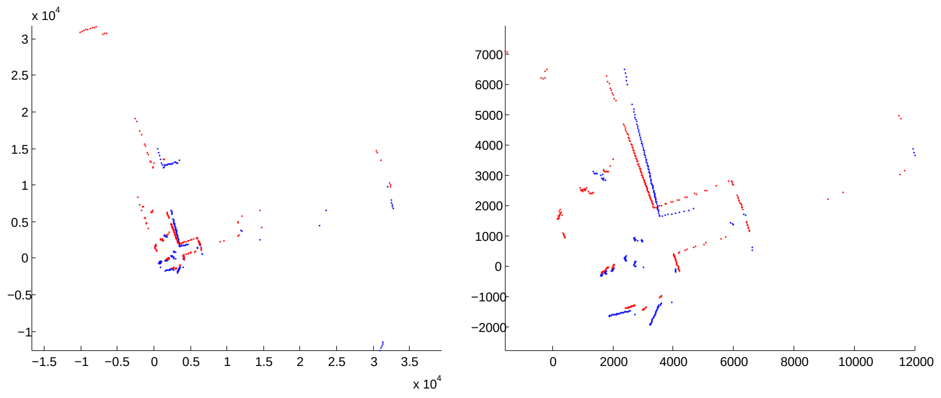


Figure 11: Detection of strong links for the first step. To the left two scans U_i and U_j. To the right $U_i \cap W_j$ and $U_j \cap W_i$. The axes are given in millimeters.

The above described algorithm only works in the first step of the SLAM-algorithm, since in the second step every pose has 5 sets of scans belonging to it and therefore one cannot define a set W_i as above. To detect strong links in this step the following is done. Consider two poses X_i^p and X_j^p with corresponding relative measurements M_i^p and M_j^p. Define W_{ij}^p as

$$W_{ij}^p = \{(x,y) \in X_i^p \oplus M_i^p : \min ||(x,y) - X_j^p \oplus M_j^p||_2 < 250\} \tag{27}$$

Define W_{ji}^p in the same way. A strong link $i \leftrightarrow j$ is detected if and only if

$$\frac{\#W_{ij}^p}{\#M_i^p} > 0.35 \tag{28}$$

and

$$\frac{\#W_{ji}^p}{\#M_j^p} > 0.35 \tag{29}$$

In words, this means that the amount of points in the set M_i^p with a distance to the closest point in M_j^p lower than 250 mm must be more than 35 percent and vice versa. Results from running the algorithm on real data are shown in Figure 12.

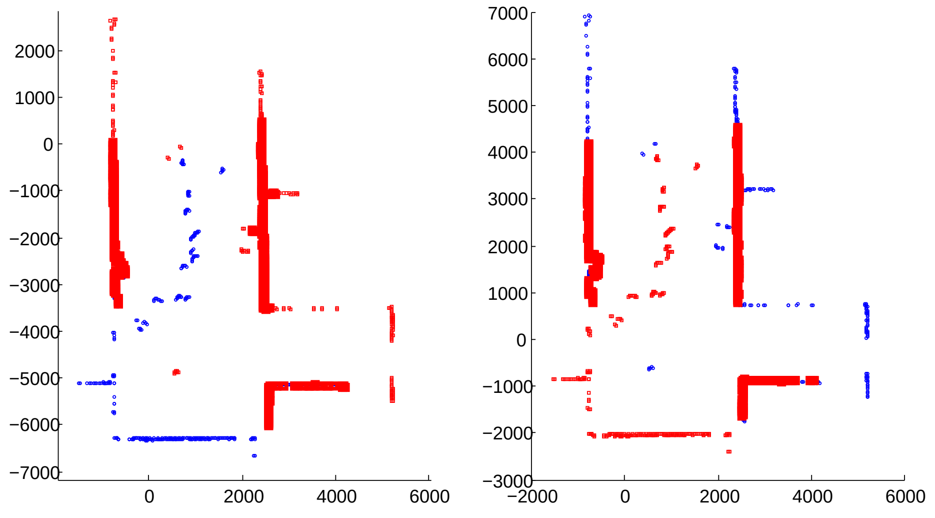


Figure 12: Detection of strong links for the second step.To the left, M_j^p small blue circles, M_i^p small red squares and W_{ij}^p red big squares. To the right, M_i^p small blue circles, M_j^p small red squares and W_{ji}^p red big squares.

5.2.4 Scan Matching

Scan matching is the process of aligning an observed set of measurement positions U_o with a reference set of measurement positions U_r. Here U_o and U_r refers to the sets of measurement positions generated by a laser range scans at two different robot poses X_o and X_r. A set of measurement positions can be described as $U = \bigcup_{k=0}^{360}(u_x, u_y)^k$ in the local cartesian coordinate system of the specific pose according to Eq. (4).

For scan matching an ICP (Iterative Closest Point) algorithm is used. The ICP algorithm is iterative and consists of two steps. First, corresponding measurement positions from the observed and reference measurement positions sets are paired to form (o_l, r_m) using an initial guess of the relative pose and a nearest neighbor approach where $o_l = (u_x, u_y)_o^l \in U_o$

and $r_m = (u_x, u_y)_r^m \in U_r$. Here positions are only paired if a measurement position in the observed set has a squared distance smaller than E to a position in the reference set. Also a position in the reference set can be paired with many positions in the observed set. Second, a new relative pose is estimated by calculating the relative pose that minimizes the least-mean-squared error between the associated measurement positions. Denote the set of all associated measurement position pairs as A and $\Delta = (\Delta x, \Delta y, \Delta\theta)^T$ as the relative pose displacement. By using Δ, o_l can be projected onto the reference frame in which r_m is described by

$$o_l^r = \begin{pmatrix} cos(\Delta\theta) & sin(\Delta\theta) \\ -sin(\Delta\theta) & cos(\Delta\theta) \end{pmatrix} o_l + \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} = Ro_l + T \tag{30}$$

where R is a clockwize rotation by the angle $\Delta\theta$ about the origin and $T = (\Delta x, \Delta y)^T$ a translation. The aim of ICP is to obtain the rotation and translation between two scans that minimizes the MSE (mean square error) which is defined by

$$\epsilon^2 = \min_{\Delta} \sum_{\forall(o_l, r_m)\in A} \|r_m - Ro_l - T\|^2 \tag{31}$$

Each iteration will yield a new estimation of relative pose displacement

$$\Delta^{new} = (\Delta x^{new}, \Delta y^{new}, \Delta\theta^{new})^T$$

and our initial guess is the odometry estimation of $\overline{D}_{or} = \Delta_{ini}$. In every iteration E is decreased until $E \leq E_{min}$. Initially $E = E_{max}$ where E_{max} is an upper bound of the expected squared odometric errors between the two scans, i.e. $E_{max} = \sigma_x^2 + \sigma_y^2$. E_{min} is based on the uncertainty of the laser measurements. If the uncertainty of the laser is modeled as a normal distribution with standard deviation σ_{laser}, the squared match errors $\|r_m - o_l^r\|^2$ can be modeled as a chi-squared distribution of one degree of freedom according to [7]. Then the squared threshold value that concentrates 99% of valid point correspondences is $(\chi^2_{0.99})^2 = (6.63\sigma_{laser})^2$ and thus $E_{min} = (6.63\sigma_{laser})^2$. For a thorough explanation of ICP and the analytical solution of the optimization problem, refer to [7].The ICP algorithm will give a new estimate of D_{ij} i.e. $\overline{D}_{ij}$ for the strong links. Since it is difficult to get a good estimate of the uncertainty of $\overline{D}_{ij}$ the corresponding C_{ij} is set to a constant diagonal matrix. δ_x and δ_y are given in m and δ_θ is given in rad and are set experimentally to from:

$$C_{ij} = \begin{pmatrix} \delta_x^2 & 0 & 0 \\ 0 & \delta_y^2 & 0 \\ 0 & 0 & \delta_\theta^2 \end{pmatrix} = \begin{pmatrix} 0.05^2 & 0 & 0 \\ 0 & 0.05^2 & 0 \\ 0 & 0 & (0.5\pi/180)^2 \end{pmatrix} \tag{32}$$

5.2.5 Mapping and Map Conversion

The Offline-SLAM algorithm generates a set of estimated robot poses that together with their corresponding laser measurements form a data set from which maps can be constructed. From this data a grid map is constructed to be used for trajectory planning. Using the extracted data set one can plot every measurement in a global coordinate system forming a map consisting of points. This map is not particularly useful and needs to be converted. A line based map to be used in Online-SLAM is extracted from these points.

5.2.6 Line Based Map

To minimize the computation in the Online-SLAM algorithm a set of lines are extracted from the point map described in the previous section. These form a line based map. The lines are computed by an iterative procedure which is based on the Hough transform [4]. The algorithm is similar to the one used in [8]. The algorithm consists of the following steps.

First the algorithm is initialized by computing the Accumulator A of the Hough transform for all the points described in a global coordinate system. The Accumulator is a two-dimensional matrix where an element $A(i, j)$ describes how many times a line on the form

$$\rho = xcos(\theta) + ysin(\theta) \tag{33}$$

has been generated. Here i and j corresponds to actual angles and distances θ and ρ. The parameters ρ and θ are defined as illustrated by Figure 13.

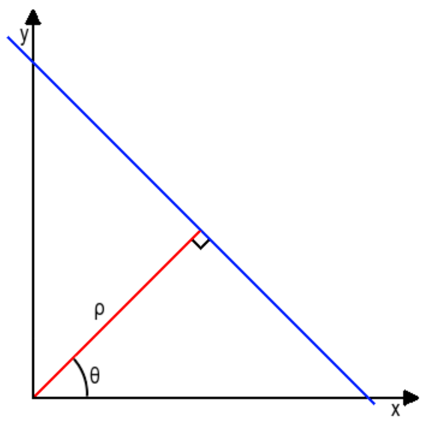


Figure 13: Line (blue) on the form $\rho = xcos(\theta) + ysin(\theta)$

For every point, with coordinates (x, y), ρ is caluculated by inserting x, y and discrete values of θ, where $0 \leq \theta \leq \pi$, into Eq. (33). For every line that is generated in this way the corresponding element in the Accumulator is increased by 1. Extracting the indexes that corresponds to the maximum value of the Accumulator gives the parameters ρ and θ that corresponds to the line that intersects the largest amount of points.

The following steps are then iterated:

I The line corresponding to the largest value of the Accumulator is calculated and points that are within a distance a_1 from this line are extracted.

II The extracted points are split into a number of segments. This is done by iteratively adding points to a segment if the distance between any of the points in the segment and the current point is less than a certain value a_2. If a segment consists of less than a_3 points the segment is disregarded.

III The remaining segments are then evaluated to see if they correspond to lines of sufficient length. This is done by calculating the maximum distance between all points in a segment and extracting the two points corresponding to this value if the maximum distance is larger than a_4. If the distance is less than a_4 the segment is disregarded. By calculating the intersection of the segment line and the lines that are orthogonal to the segment line and goes through the two extracted points, the end points of the segment line are extracted.

IV The points corresponding to the remaining segments will be removed from the total set of points, i.e from the map consisting of points, and the extracted end points are saved. In addition these points are removed from the Accumulator. The process is then iterated for the remaining points until the maximum value of the Accumulator falls below the threshold a_5.

Since the lines that have been extracted has to be of at least length a_4 and be generated by at least a_3 points, small objects will be filtered out.

The map that can be constructed from the extracted end points have undesired line gaps that needs to be connected. This is done in the following way. First the distance between all the extracted end points are calculated. If the distance between two end points is less than a_6 the intersection point between these lines is calculated as well as the squared distances r_1 and r_2 from the end points to the intersection point. If the criteria $r_1 + r_2 \leq 2a_6^2$ is fulfilled the corresponding end points of the lines are set to the intersection point. If the lines can not be extended in this way but the distance between end points is less than a_6 a new line will be added to fill the gap. The value a_6 is low meaning that only very short lines can be generated and that lines cannot be extended much.

The values of the distances a_1, a_2, a_4, a_6 , the number of points a_3 and the value a_5 are set experimentally. The following values are used $a_1 = 10, a_2 = 15, a_4 = 25, a_6 = 20, a_3 = 8$ and $a_5 = 2$. All distances are in cm.

Figure 14 illustrates the extraction of a line based map from real data. To the left is the map consisting of points. To the right the lines have been extracted. The red points represent points that have been discarded or that has not contributed to any line and the green points represent points that have contributed to a line.

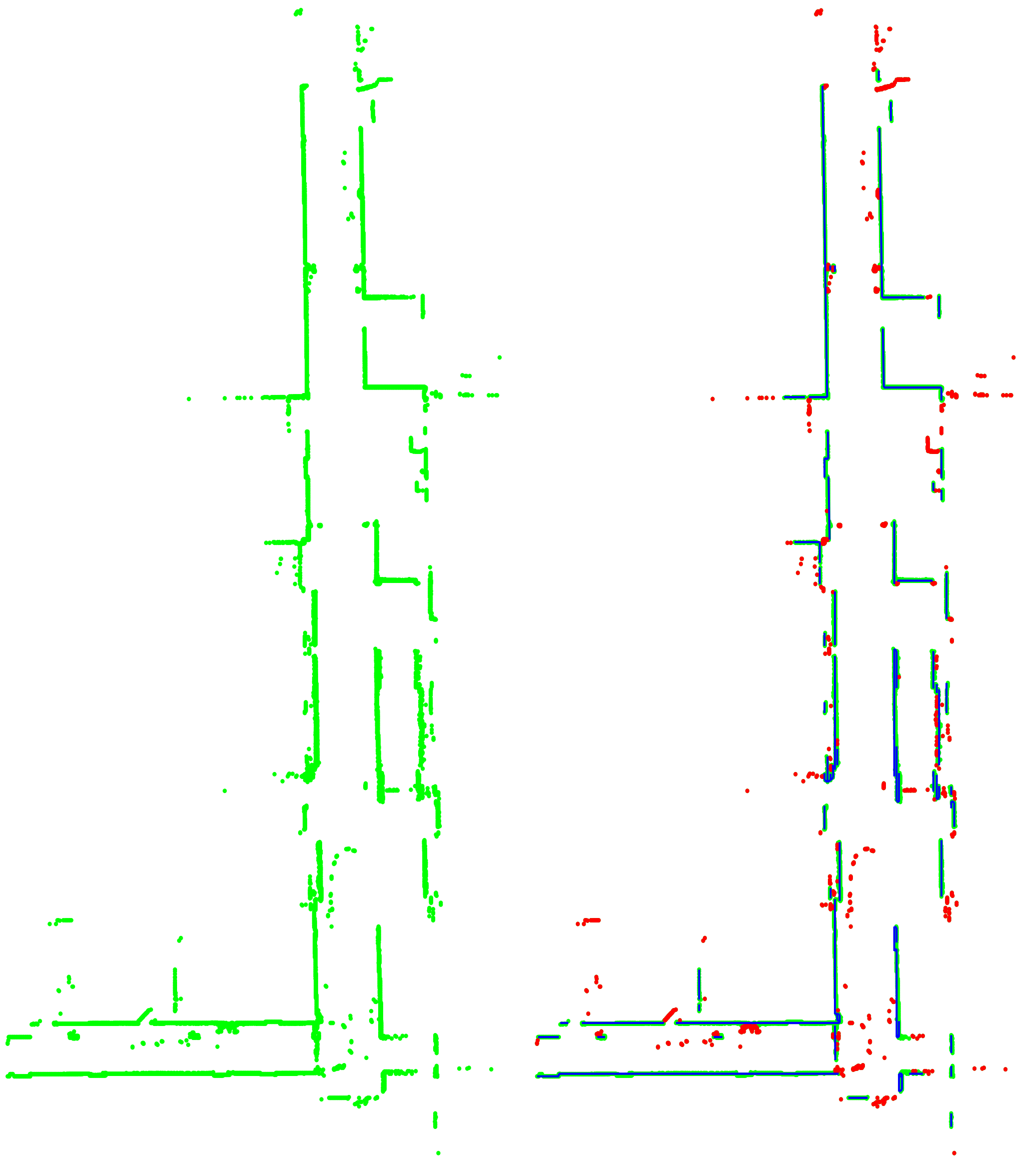


Figure 14: To the left is a point map extracted from real data generated by Offline-SLAM. To the right is a line map that has been extracted from the point map. The red points are discarded or redundant and the green points have contributed to a line

Figure 15 displays the line map extracted from the point map in Figure 14.

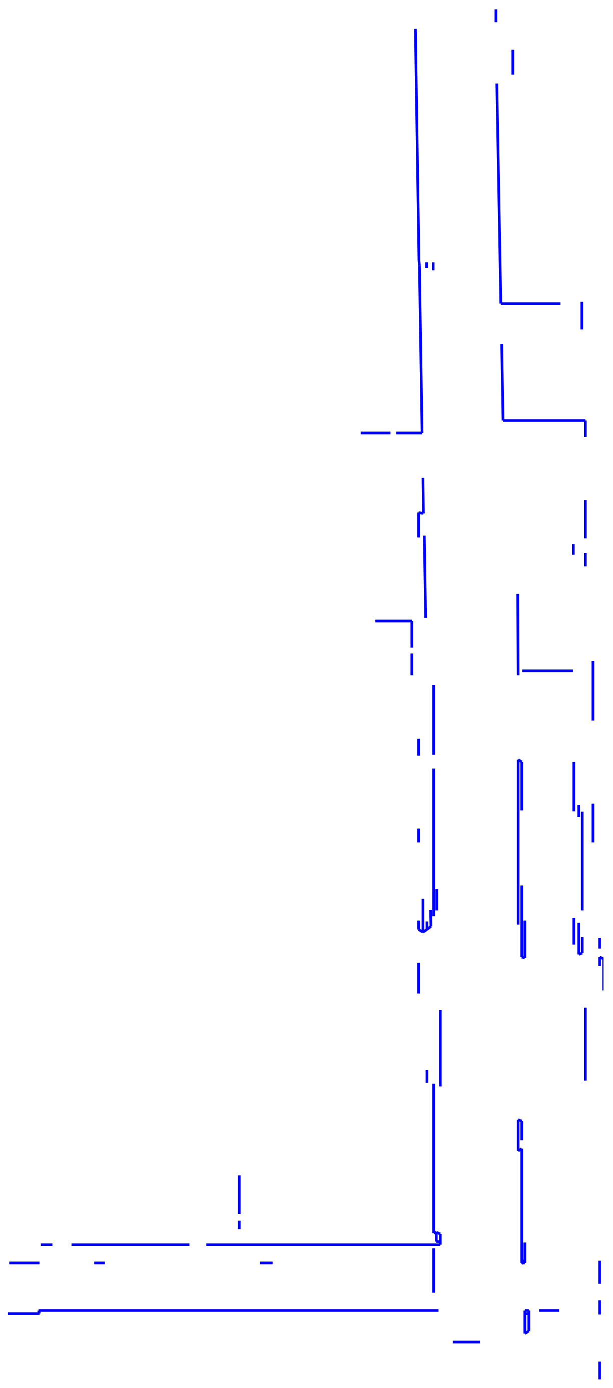


Figure 15: The line map extracted from the Offline-SLAM point map in Figure 14

The algorithm can generate lines that are redundant. One way to improve the algorithm and get rid of these lines could be to simulate laser scans from different robot poses in the generated map and remove lines that are never intersected. Typically one could use the estimated poses from the Offline-SLAM algorithm to simulate measurements.

5.3 Online-SLAM

This section describes how the localization and mapping is performed when the robot is operating.

5.3.1 Monte Carlo Localization

The robot localizes itself using a Monte Carlo Localization (MCL) algorithm. The MCL uses a collection of samples (also known as particles) to estimate the robot's pose. These particles are spread out on the predefined map which is built as described above. All particles describe a possible pose for the robot and the possibility (importance weight) that the robot actually have that pose.

The MCL algorithm, also illustrated by Figure 16, is:

I All particles are uniformly distributed within a radius of 0.8 meter from the robots inital pose on the map, with equal importance weights.

II The robot continuously scans the surroundings using the laser range sensor. When the robot receives new measurement data it checks if the robot has moved a distance greater than 0.5 meter or changed its heading with an angle greater than 15 degrees. When the condition is reached the robot will proceed to the next step or else wait for new data and redo the procedure.

III All particles poses are updated by sampling new positions from the motion model.

IV Each robot laser scan consists of 361 measurements. The 180 degree field of view is divided into nine segments. Each segment is filtered to remove measurements which has detected short distances. This is to prevent that smaller objects close to the robot affect the next step where one measurement is extracted from each segment. To determine which samples to choose the ordinary least squares(OLS) is applied to each segment and the sample that is closest to the line is chosen. If the number of samples remaining in a segment after the filtering step is less than half, a random sample is instead chosen from the segment. The nine samples' measured distances and their corresponding angles are stored. Then each particle simulates nine sensor measurements in the same angles as the previously extracted samples. The simulated laser rays are vectors originating from the particle locations. When they intersect with a line segment in the map the measured distance from the particle to the intersection is stored.

V The N measured distances for each particle are subtracted with the M actual measured distances. The error between the measurements is accumulated and is used to determine the importance weight for the particles. The weight for each particle is the inverse of the error divided by the total error of all particles so that the importance weights sum up to one. If the particles measurement data is similar to the actual data the environment for both the particle and the robot are similar therefore the importance weight increases. In the case when the measurement data are very different the importance weight decreases. A point estimation of the robots pose is done by calculating a weighted mean of all particles. To be able to determine a measure of the uncertainty of the estimated robot pose a new measurement simulation is performed from the estimated pose. The squared error between these measurements and the robots measurements serves as an indication of the robots uncertainty.

VI A new set of particles is created. By sampling(with replacement) from the current set the new particles will be distributed according to the importance weights of the current particles. More particles will be sampled from the particles with high importance weight since they have a higher probability. This method is used to draw 90 percent of the new particles. The remaining 10 percent are randomly distributed around the estimated position of the robot similarly to I. This solution adds a possibility to correct the estimated robot pose if the particles have wrong poses.

VII This new set will replace the old set of particles and all particles will have an equal importance weight. After this the whole procedure is repeated from step II.

An addition to the MCL algorithm presented above is a function that has been named saveSlam. This function is called upon when the robot has been stuck in the same pose more than a few seconds for some reason. The function reinitializes all particles and spreads them uniformly around the last estimated pose. In almost all cases when the robot gets stuck it does that because it has a bad estimate of its pose. The reinitialization solves this problem since the particles get a new chance of making a better estimation of the pose.

As already mentioned, the map used for localization consists of a set of lines. The reason for this choice is to make the MCL as efficient as possible. By using a line based map, the environment can be described by a relative small set of objects without loss of resolution. Also, since the only measurements used for localization arise from the laser sensor and since a laser ray can be described as a line, the simulation of the measurements is simple. However, as mentioned earlier, the map used for localization will almost never contain small objects and therefore it is important that measurements arising from small objects are detected and filtered out as described in IV.

5.3.2 Online Mapping

The online mapping differs from the offline mapping previously described. For the navigation to work the map needs to be updated when new objects are detected and objects that are no longer present should be removed from the map. This is done by using two different maps, one grid map and one map containing line segments.

The grid map based is dynamic and is continuously updated if changes in the map are detected so that the control unit knows about the environment. The line based map is static and is only used by the MCL algorithms particles for localization. The reason for this is that the main objects in the line based map, e.g. walls, will not likely be moved or removed. Therefore, the big difference between the line based map and the reality will be dynamic objects such as people, smaller objects such as tables and chairs and objects that the laser cannot detect, e.g. glass. This is why the algorithm that chooses the angles at which the particles simulate range measurements in MCL, is used to filter out the possible outliers caused by objects not included in the grid map. However the algorithm is not able to detect all dynamic objects which cause the robot to be more uncertain of its pose at those locations. Since the MCL is used, the particles will spread out over a larger area when the robot moves in an environment where the map differs from the reality. But when it exits that area the particles will gather around the robot again making the uncertainty of the pose smaller.

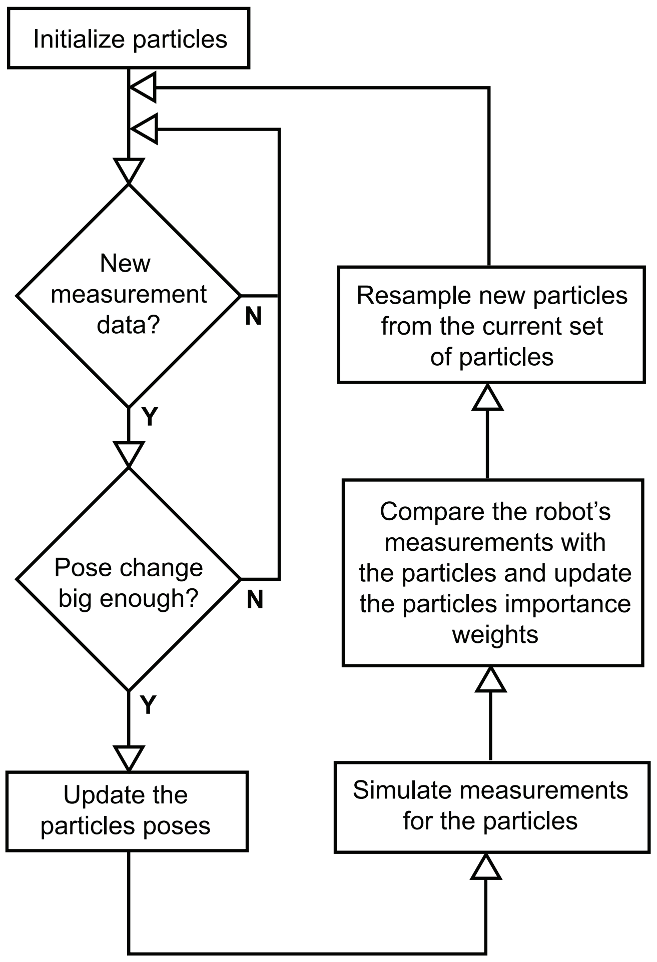

Initialize particles
New measurement data?
N
Y
Pose change big enough?
N
Y
Update the particles poses
Simulate measurements for the particles
Compare the robot's measurements with the particles and update the particles importance weights
Resample new particles from the current set of particles

Figure 16: The MCL algorithm

5.4 Grid mapping

The grid map is constructed from laser measurements. Since this map is used for trajectory planning it will be much more detailed than its line based counterpart. This map will also be updated by the Online-SLAM algorithm. Throughout this section, we will use the terms grid, cell and grid cell interchangeably.

5.4.1 Occupancy grid mapping

Occupancy grid mapping generate maps under the assumption that the robot pose is known, from noisy and uncertain sensor measurement data. The basic idea of the occupancy grid is to represent a map of the environment as binary random variables. Each random variable is binary and represents the presence of an obstacle at that location in the environment. Occupancy grid algorithms compute approximate posterior estimates for these random variables. The occupancy grid is used after solving the SLAM problem and therefore take the path estimate as given. Hence the controls and odometry data have no part in the occupancy grid since the path is assumed to be known.

The occupancy grid is used to estimate the posterior probability over maps given the data [9],

$$p(m|z_{1:t}, x_{1:t}), \tag{34}$$

where m is the map, $z_{1:t}$ is the set of measurements from time 1 to t, and $x_{1:t}$ is the set of robot poses from time 1 to t. An occupancy grid represents the map as a fine-grained grid over the area in the environment that represent the map m. Consequently this posterior is almost impossible to calculate because an occupancy map divides the environment into finitely many grid cells. The number of maps that can be represented by a map with 10000 grid cells is 2^{10000}. Therefore it is intractable to calculate the posterior probability for each single map. The map that have been generated for this case contains 405000 grid cells, see Figure 22.

Denote m_i as the grid cell with index i, then $p(m_i)$ represents the probability that grid cell i is occupied. The idea is then to break down the problem into smaller problems of estimating

$$p(m_i|z_{1:t}, x_{1:t}) \tag{35}$$

for all grid cells. Each of these estimation problems is then a binary problem. This breakdown is appropriate but is not capable to represent dependencies between neighboring cells. Instead, the posterior of a map is approximated as the product of each grid cell

$$p(m|z_{1:t}, x_{1:t}) = \prod p(m_i|z_{1:t}, x_{1:t}). \tag{36}$$

This factorization makes it possible to use a binary Bayes filter to estimate the occupancy probability for each grid cell.

A very useful algorithm in grid mapping is the occupancy grid algorithm which apply the log odds representation of occupancy. The odds of a state is defined as the ratio of the probability for an event divided by the probability of its negate:

$$\frac{p(x)}{p(-x)} = \frac{p(x)}{1 - p(x)} \tag{37}$$

As a consequence, the log odds ratio is the logarithm of the odds:

$$l(x) = log\frac{p(x)}{1 - p(x)} \tag{38}$$

A belief reflects the robot's internal knowledge about the state of the environment and will be denoted $bel(x_t)$, where:

$$bel(x_t) = p(x_t|z_{1:t}, u_{1:t}) = p(x_t|z_{1:t}) \tag{39}$$

The controls and odometry $u_{1:t}$ play no role here either because the belief only depend on the measurements when the state is static. The belief is commonly implemented as a log odds ratio and denoted $l_t(x_t)$, the log odds belief at time t is given by [9]:

$$l_t(x_t, z_t) = l_{t-1}(x_t, z_{t-1}) + log\frac{p(x_t|z_t)}{1 - p(x_t|z_t)} - l_o(x) \tag{40}$$

where

$$l_{t-1}(x_t, z_{t-1}) = log\frac{p(x_t|z_{1:t-1})}{1 - p(x_t|z_{1:t-1})} \tag{41}$$

and the prior

$$l_o = log\frac{p(x_t)}{1 - p(x_t)} \tag{42}$$

The algorithm for occupancy gridmapping can be determined by applying the log odds ratio for every grid which is a combination of equation 35 and 40:

$$l_{(t,i)}(m_i, x_t, z_t) = l_{t-1}(m_i, x_{t-1}, z_{t-1}) + log\frac{p(m_i|x_t, z_t)}{1 - p(m_i|x_t, z_t)} - l_o \tag{43}$$

where

$$l_{t-1}(m_i, x_{t-1}, z_{t-1}) = log\frac{p(m_i|x_{1:t-1}, z_{1:t-1})}{1 - p(m_i|x_{1:t-1}, z_{1:t-1})} \tag{44}$$

and the prior

$$l_o = log\frac{p(m_i)}{1 - p(m_i)} \tag{45}$$

Algorithm 1 Occupancy grid mapping

The general idea of the algorithm is:

```
1: Input: $l_{(t-1,i)}, x_t, z_t$
2: for all grid cells $m_i$ do
3:     if the cell $m_i$ is in the measurement $z_t$, i.e. the grid cell $m_i$ is covered by the sensor then
4:         update the log odds of the grid cell with
               $l_{(t,i)} = l_{(t-1,i)} + InverseSensorModel(m_i, x_t, z_t) - l_0$    (46)
5:     else {$m_i$ is outside $z_t$}
6:         then the log odds of the grid is exactly the same as it was before
               $l_{(t,i)} = l_{(t-1,i)}$    (47)
7:     end if
8: end for when all grid cells have been examined
9: Output: $l_{(t,i)}$
```

The function $InverseSensorModel(m_i, x_t, z_t)$ implements the inverse measurement model $p(m_i|x_{1:t}, z_{1:t})$ in its log odds form:

$$InverseSensorModel(m_i, x_t, z_t) = log\frac{p(m_i|x_t, z_t)}{1 - p(m_i|x_t, z_t)} \tag{48}$$

Algorithm 1 defines the basic principles of the occupancy gridmapping algorithm.

Inverse models are often used in situations where the measurement distribution is more difficult to produce than a binary state. An example of such a situation is the problem of estimating a door being open or closed using camera images. The state, open or closed, is extremely simple while the measurement model $p(z_t|x)$ is much more complex, i.e. it is easier to determine the probability of a door being open from a camera image than calculating the distribution of all camera images showing an open door.

A very neat property of the log odds representation is that the occupancy probabilities are easily recovered:

$$p(m_i|z_t, x_t) = 1 - \frac{1}{1 + \exp(l_{t,i})} \tag{49}$$

Although the sonars are not used for occupancy grid mapping in this project, there will be a thorough explanation of the inverse sensor model for sonars since the concept is useful to understand when the laser is implemented. The laser can actually be seen as a special case of the sonar where the opening angle is equal to zero.

The inverse sensor model can be designed in many different ways, here a very basic version will be used. The inverse measurement model $p(m_i|z_t, x_t)$ describes the probability of a grid being occupied given the difference between the measured distance z_t and the distance of the grid m_i relative the robot pose x_t. The signals from sonars are typically emitted in a cone, the angle between the grid and the horizontal axis will therefore be considered in $p(m_i|z_t, x_t)$, see Figure 17.

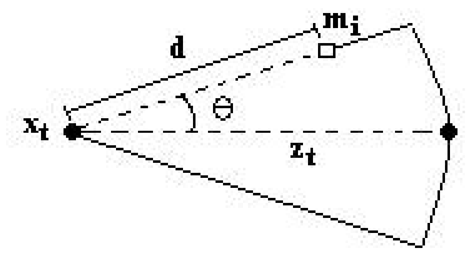


Figure 17: The occupancy probability of a grid m_i depends on the distance d to x_t and the angle θ from the horizontal axis of the sonar.

The inverse measurement model can be determined by using a nonlinear function, [10], s(z_t,θ) which is a combination of a Gaussian $\mathcal{N}(0, \sigma_\theta)$ and a linear function $f(z_t)$, see Figure 18, where the Gaussian part describes the angular range of the sonar and the linear part is determined by the measured distance, see Figure 19.

$$s(z_t, \theta) = f(z_t) \cdot \mathcal{N}(0, \sigma_\theta) \tag{50}$$

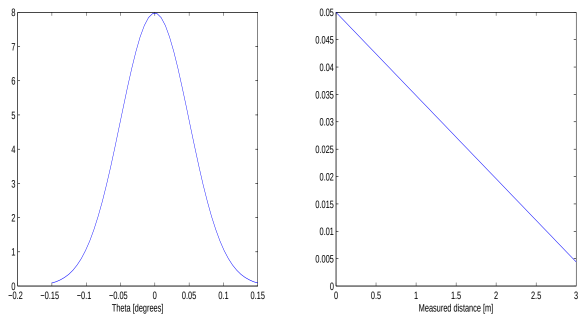


Figure 18: Functions used to compute s(z_t,θ), left - Gaussian function, right - linear function.

The shape of the linear function and the standard deviation of the Gaussian distribution depend on the characteristic of the sonars. To determine these functions one has to calculate a fitness function with sensor measurements where the map and the position are known. A common way to solve this kind of problem is the iterative optimization algorithm such as gradient descent[1]. The sonars are not used for gridmapping in this

[1]Thank you Jörg Müller, Albert-Ludwigs-University of Freiburg, for your support.

project and therefore we have not made any effort to determine these functions. The functions shown in Figure 18 will be used only for illustration, new functions must be determined to obtain suitable results. However, the variance of the Gaussian distribution is 0.05 which corresponds to the opening angle of 15 degrees for the used sonar in the illustrations. Since the output from s(z_t,θ) is probability, there will not be any specific unit on the Gaussian and the linear function. More information about these functions can be found in [10].

The function s(z_t,θ) is used to determine how the probability of a grid will be changed relative to the prior probability of the grid. To clarify the functionality, an example will be shown. Suppose the sonar gives a measured distance of 1.5 m and the grid to be updated is located along the horizontal axis, i.e. θ equals zero. According to Figure 19 this measurement corresponds to a value of approximately 0.15, assuming that the prior probabilities for all grids are 0.5. Figure 20 shows the occupancy probability where d_1, d_2 and d_3 are design parameters and the top and bottom levels respectively, are determined by the prior $\pm$ s(z_t,θ), i.e. $0.5 + 0.15 = 0.64$ and $0.5 - 0.15 = 0.35$. The constant d_1 specifies how fast the probability of grids close to the measured distance will increase or decrease. The average depth of obstacles in the gridmap is specified by the $d_2 - d_1$ while the constant d_3 have the functionality of linearly decreasing the probability behind the obstacle to the prior occupancy probability.

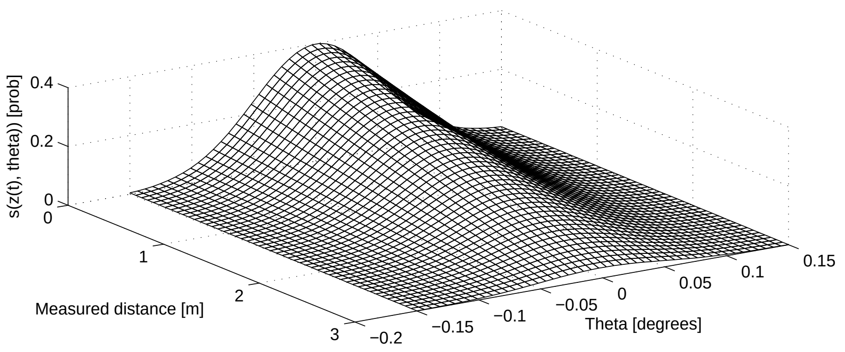


Figure 19: Function s(z_t,θ) is used to determine how the probability of a grid will be changed relative to the prior probability of the grid

When $p(m_i|z_t, x_t)$ is determined, the inverse sensor model can be calculated and the occupancy probability of a grid is updated. If the probability from $p(m_i|z_t, x_t)$ is greater than 0.5 it will generate a positive value of the inverse sensor model, i.e. the occupancy probability for a grid will increase according to the occupancy grid mapping algorithm, which can easily be verified by the summation of the log odds ratio. Analogically, values less than 0.5 will decrease the occupancy probability. A special case occurs when the distance to the grid is equal to the measured distance. The inverse measurement model will then be 0.5 which corresponds to zero for the inverse sensor model, i.e. the probability of a grid will not be changed and the previously value is kept, [10]. This feature is very useful for small grids because the measurement uncertainty will be taken into account. With small grids it is difficult to know in which grid cell the measurement actually ended up in, i.e. "the probability that the measurement was too long and the cell is free is equal to the probability that the measurement was too short and the cell is occupied" [2] .

[2]Once again, thank you Jörg for the discussions and the expertise.

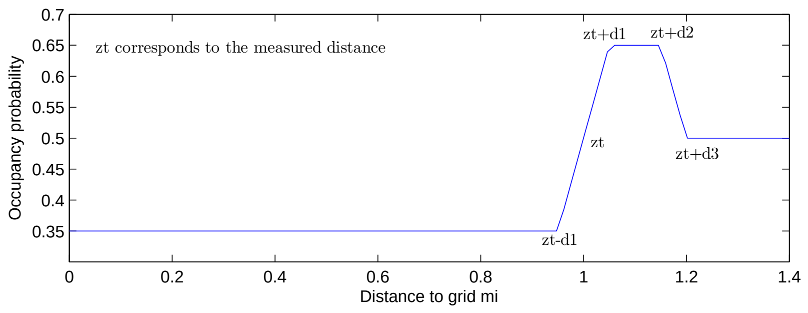


Figure 20: The probability $p(m_i|z_t, x_t)$ that a grid is occupied as a function of the distance to the grid.

Unfortunately there was not any time left to implement the inverse measurement model for the sonar. Since the laser can be seen as a special case of the sonar, the procedure will almost be the same for the laser. The major difference between the laser and the sonar is the lack of angular dependence, which corresponds to a two dimensional version of the function shown in Figure 19.

To simplify the inverse measurement model even more, the linear function s(z_t) was not used at all, instead the inverse measurement model was represented by the function in Figure 20. The simplification refers to a function with fixed values of the top and bottom levels regardless of the measured distance, [10]. These simplifications can be done because the laser does not have the same range and angular uncertainty as the sonar.

Algorithm 2 defines the basic principles of the occupancy gridmapping algorithm for the laser and the result can be seen in Figure 21.

Algorithm 2 Occupancy grid mapping with a laser range finder

The algorithm for the laser range finder is:

```
1:  Input: $l_{(t-1,i)}, x_t, z_t$
2:  for each single laser beam do
3:      Calculate all cells of the grid map covered by the laser beam extended by $d_3$ using
        Bresenham's algorithm (we assume the measurement to be a perfect line)
4:      for each covered cell do
5:          Calculate the distance of the center of the cell to the origin of the laser beam
6:          Calculate the occupancy probability $p(m_i|z_t, x_t)$ given by the inverse sensor model
            specified in Figure 20
7:          Update the log odds, $l_{(t,i)}$, of the cell
8:      end for when all covered cell in the laser beam have been updated
9:  end for when all laser beams have been examined
10: Output: $l_{(t,i)}$
```

The constants d_1, d_2, and d_3 specify the interval in which the different linear approximations are valid, see Figure 20, were d_1 and d_2 correspond too the average depth of obstacles, [10]. Using a standard laser range finder from Sick the parameters are set: d_1 = 0.05m, d_2 = d_1 + 0.1 m and d_3 = d_2 + 0.05m.

Since the pathfinding algorithm only has binary decisions, i.e. the grid is empty and possible to cross or the cell is occupied and not suitable for entering, two different gridmaps will be used. The first gridmap employs the log odds ratio, defined in equation 40, and consists of values from minus infinity to infinity. As earlier mention, zero in the log odds ratio corresponds to an occupancy probability of 0.5 while a large positive value corresponds to high probability and vice-versa. The other gridmap is binary (used by the pathfinding algorithm) and consists of zeros and ones extracted from the logarithmic gridmap. Therefore a threshold level, typically chosen to 0.5, must define a probability where the grid is considered occupied or not, see Figure 22.

Algorithm 3 Basic gridsearching algorithm

The basic gridsearching is defined as:

```
1: Input: distance to target in polar coordinates
2: distance to the grid = distance to target + $d_3$
3: **while** distance to grid $\geq 0$ **do**
4:    Convert the distance to cartesian coordinates and round to nearest grid
5:    Update the probability according to Algorithm 1
6:    distance to grid = distance to grid - grid size
7: **end while**
```

Another important thing to consider in the binary gridmap is the size of the robot. Usually the grid size is much smaller than the robot which creates problems because the robot position in pathfinding algorithm will always be illustrated by one grid (the center of the robot), regardless of the actual robot size. The problem with this algorithm is quite obvious and can be solved by padding the surrounding environment. The pathfinding algorithm will then interpret the map as being smaller and therefore always choose a path far enough for the robot to avoid a collision with the surrounding obstacles.

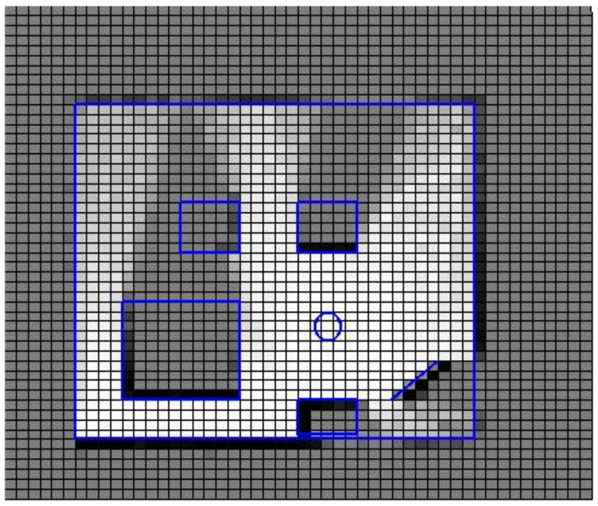


Figure 21: A logarithmic gridmap created in Matlab to simulate the occupancy grid mapping algorithm. The blue lines defines the obstacles in the environment and the color of each grid corresponds to the probability for a grid being occupied (white = low probability, dark = high probability, grey = prior probability, i.e. unexplored areas).

To determine if a laser beam cross a grid cell m_i we have implemented Bresenham's line algorithm [6]. The algorithm is used to determine which grid cells is covered by the laser beam. Since time was running out, we did not have time to test and evaluate the performance of Bresenham's line algorithm. Instead, a very simple algorithm is used to identify which grids are passed by the laser beam, see Algorithm 3. But the Bresenham's line algorithm is much more sophisticated and will therefore be discussed here. It is an algorithm which determines which pixels that should be plotted in order to form a close approximation to a straight line between two given points [1]. The general equation of the line through the endpoints is given by,

$$\frac{y-y_0}{y_1-y_0}=\frac{x-x_0}{x_1-x_0}, \tag{51}$$

were the the endpoints of the line are the pixels at (x_0, y_0) and (x_1, y_1). Hence the algorithm find out if we have to move one step right or one step up and one to the right, for the example were the start pixel (x_0, y_0) coordinates are smaller then the end pixels (x_1, y_1), i.e. $x_0 \leq x_1, y_0 \leq y_1$, we know the cell column,x and the row, y, is given by:

$$y=\frac{y_1-y_0}{x_1-x_0}(x-x_0)+y_0=k(x-x_0)+y_0. \tag{52}$$

So from the start pixel the rows are counted up until the rows are equal to x_1 and the corresponding column, y, are calculated for each x. By analogy to other circuits. Since the algorithm is very simple and fast, uses only integers, it is commonly used to draw lines on a computer screen.

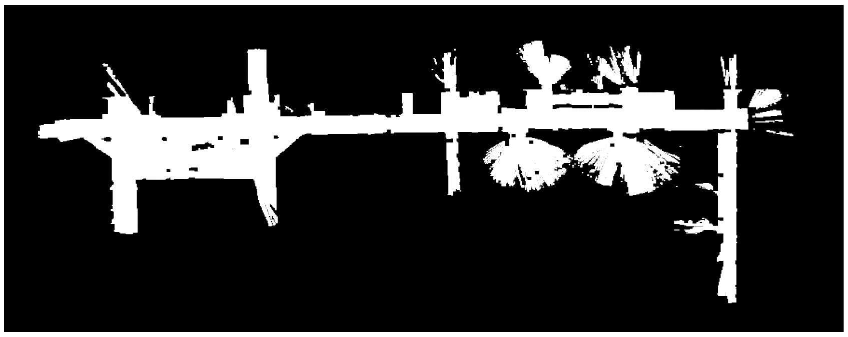


Figure 22: A binary version of the occupancy gridmap is used in the pathfinding algorithm.

5.4.2 Future development

There is a lot of thing that can be improved in the occupancy grid mapping algorithm. For instance, the sonar is highly recommended to evaluate and implement in the grid map. The easiest and most intuitive way is probably the previously mentioned method where the non linear function s(z_t,θ) is used. Since the laser and the sonars can detect different obstacles one has to merge the result from these sensors. One approach is to build separate maps for each sensor and employ the most pessimistic map, i.e. if one of the sensors detects an occupied grid, so will the combined map. More about this can be found in [9].

Another useful improvement is the implementation of an algorithm which automatically calculates the desired gridmap size and the corresponding offset. The robot always defines origo as the position where the robot begins when the data gathering for the offline-slam algorithm starts. Negative robot positions will then occur if the robot returns to its start position and goes in opposite direction relative the heading originating from the start pose. This will cause problems in the occupancy gridmapping algorithm since the map is created by multidimensional vectors where negative elements does not exist, therefore an offset but also a large enough map is needed.

Bresenham's line algorithm is already implemented in c-code but is not used at the moment. The time was running out and there were a lot of other things we had to accomplish before the algorithm could be tested. Bresenham's has been evaluated in c-code with simulated measurements from matlab and is much faster than the basic algorithm which is implemented right now.

6 Pathfinding

The robot employs an implementation of the Dynamic A* Lite (D* Lite) pathfinding algorithm. It is used on a supplied grid map, where each cell corresponds to a node. If a cell is blocked by an obstacle, the cost of traveling to the node it represents will be set to infinity. The same applies for a node to which a path is not yet known. The cost of moving between two adjacent nodes is set to the Euclidean distance, given by the Pythagorean Theorem. Also, the cost of traversing a cell next to a wall has been increased by a given value, in order to make the robot prefer paths away from obstacles.

Both these changes differ from the original algorithm, where the cost of moving between two adjacent nodes is always set to one, and have been done purely from observational results.

The computation results in a cost matrix, where each expanded node is associated with the accumulated cost of moving there from the goal node; the path is subsequently calculated by minimizing the cost from the robots current node to the goal node. In the same way, re-calculation of the shortest path implies updating the cost matrix.

6.1 Definitions

The following notations are used throughout the description.

1. s refers to an arbitrary node.
2. s_{start} refers to the current node for the robot.
3. s_{goal} refers to the goal node for the robot.
4. $c(s', s)$ is the cost of moving from node s' to node s. It is set to 1 if s' and s are next to each other horizontally or vertically and $\sqrt{2}$ if they are next to each other diagonally.
5. *map* refers to a given predefined grid map.
6. $g(s)$ is the recursively calculated accumulated cost of reaching node s from s_{goal} and can be defined as the lowest $g(s) = g(s') + c(s', s)$ where s' is one of the eight nodes next to s. If the node s is adjacent to an obstacle, a predefined cost is added to $g(s)$. Nodes that have not yet been expanded or represent obstacles have a $g(s)$ of infinity.
7. $rhs(s)$ is the one step look ahead accumulated cost of reaching a node s from s_{goal}. It is defined in the same way as $g(s)$ but is computated one step earlier in the algorithm. If $g(s) \neq rhs(s)$ the node is said to be inconsistent; after the first time computation of the path, inconsistency of a node implies that the cost of reaching it has changed (i.e. a new obstacle has been detected or an old one has disappeared) and the path needs to be recalculated.
8. $h(s', s)$ is the heuristic cost of reaching node s from node s'. Here, this corresponds to the shortest path Euclidean wise with no consideration of walls or obstacles.
9. k_m is the key modifier value, used when recalculating the path. It is initially set to 0.
10. k_1 and k_2 are the two key values each node is given. They determine in which order the nodes will be searched and are combined into a two component vector, where k_2 is used as a tie breaker when the k_1 for two compared nodes are equal.

11. U is the priority queue for nodes where each node is stored with an associated key, from lowest (top) to highest key. The nodes will be added or removed during the computation under given conditions.

6.2 Initialization

The initialization is done by the function $newRoute(s_{start}, s_{goal}, map)$ when executed with a given curren node s_{start} of the robot, a goal node s_{goal} and a predefined grid map from the control system. The algorithm performs a backward search; thus $g(s)$ and $rhs(s)$ is set to infinity for each node s representing a cell, except $rhs(s_{goal})$ which is set to zero. The key for a node s is a two component vector

$$\begin{pmatrix} k_1 \\ k_2 \end{pmatrix} = \begin{pmatrix} min(g(s), rhs(s)) + h(s_{start}, s) + k_m \\ min(g(s), rhs(s)) \end{pmatrix} \tag{53}$$

calculated with the function $calculateKey(s)$ on a given node s. The key modifier variable k_m is used later on when determining keys for subsequent nodes after changes in edge costs and is initially 0. As a last step of the initialization process, the goal node is associated with a key and pushed to the priority queue U .

6.3 First time computation of the path

The path is calculated by the function $computeShortestPath()$ which is subsequently called by $newRoute()$ after the initialization process has been carried out. The function will run as long as the lowest priority key in the queue is lower than the key of the start node (i.e. the current node), or as long as the start node is locally inconsistent (i.e. $g(s_{start}) \neq rhs(s_{start})$).

The function picks and removes the node s with the lowest associated key from the priority queue U and investigates it under three given conditions:

1. If the node has a key in the priority queue U which is lower than its current key, the node will be inserted once again in the priority queue U together with its updated key value.

2. If the node is found out to be over consistent ($g(s) > rhs(s)$) then $g(s)$ is set to $rhs(s)$ and the function $updateNode(s)$ is executed on all its neighboring nodes.

3. If none of these scenarios apply then $g(s)$ is set to infinity for the node and the function $updateNode(s)$ is executed on all its neighboring nodes.

The function $updateNode(s)$ updates a node s under three different conditions:

1. If the node is not the goal-node, $rhs(s)$ is calculated as the lowest cost of reaching the node by the function $lowestCost(s)$.

2. If the node s is in the queue U, it is removed.

3. If the node is inconsistent (i.e. $g(s_{start}) \neq rhs(s_{start})$) it is added to the queue U along with a calculated key.

The function $lowestCost(s)$ returns the lowest accumulated cost of reaching a node s. This is done by investigating all eight neighboring nodes s' and returning the lowest $g(s') + c(s', s)$. If the node s represents a cell occupied with an obstacle, a cost of infinity is returned.

6.4 Dynamic re-calculation of the path

The function $nextNode(s_{start})$ is called by the control system and yields the next node to which the robot should travel in order to reach its destination, where the given node s_{start} is the actual confirmed location of the robot.

When called, s_{start} is compared to the expected node from when the function was called last time. If they are equal, and no updates in the map have been detected, simply the next node s that minimizes the cost of reaching s_{goal} is returned.

If they are not equal, the robot is not in the cell where it is expected to be, and s_{start} is added to the list of nodes that needs to be updated. This is also an enhancement of the original algorithm, which does not cope with faulty movement of the robot. The map is also checked for changed cells (new obstacle or old ones that have disappeared). Each node that corresponds to such a cell is added to the same list of nodes.

The function *updateShortestPath*() sets the key modifier value k_m as $k_m = k_m + h(s_{last}, s_{start})$ where $h(s_{last}, s_{start})$ is the heuristic value between either the original s_{start} or the latest node s in which the path was updated and the current node s_{start}. The function then steps through the list of nodes and perform *updateNode*(s) on each node s and its neighbors. Afterwards, the function *computeShortestPath*() runs once again and the next node s that minimizes the cost of reaching s_{goal} is returned.

As a special case, if $rhs(s_{start})$ is still infinity after the execution *updateShortestPath*() then no known path has been found. This should normally not happen and is quite computational heavy for the algorithm since all available nodes are expanded before it can be concluded. Should it occur, a flag is returned, but no other actions are taken.

6.5 Future development

Actuation of the implemented pathfinding algorithm shows an initial computation time of approximately 3 to 4 seconds to find a path through a normal sized mapped area (400 by 500 grids). However, the dynamic re-calculation of the path results in computation times of larger magnitude, even for relatively small changes, up to ten seconds.

This is unacceptable and the pathfinding can obviously be optimized, but the direct cause of the problem is yet to be found. Since it did not show up during simulation it is thought to be related to the chosen programming solutions, not the implementation of the algorithm itself.

As a comparison, the priority queue for nodes and keys was initially implemented as a sorted vector which resulted computation times of well over three minutes. When switched to the current solution of a permanently sorted C++ multimap, computation times dropped to the current time of a couple of seconds. This problem is thought to be related to a similar cause, but no further guidance can be given.

Further on, the algorithm and thus the pathfinder always chooses the shortest path whether or not it is the most feasible route out of practical reasons. For example, one might prefer the robot not to enter the goal diagonally, taking a route with as few bends and corners as possible (i.e. favoring nodes in straight distance from the current one) or perhaps avoiding open spaces. Such solutions can be done with more creative and dynamic costs between nodes used by the *lowestcost*(s) function, or different constraints when choosing the next node with the $nextNode(s_{start})$ function.

7 Control

On program start up the following is done:

- Connections with the robot and the laser are set up
- The graphics objects are initialized
- The maps are read from file and drawn
- An object for path finding is created
- A RobotThread and a SLAMThread are started

After initialization the program waits until the program window is closed before all connections are closed and the program exits.

The RobotThread contains the loop that controls the robot. The SLAMThread does the localization which takes some time and therefore needs to be done in parallel with robot control. Localization is done as often as possible. The main loop runs once every 100 ms, because that is how often new odometry information is available and new control signals can be sent. Every iteration of the loop does the following:

- If there is new localization information, the robot's pose estimate is updated
- If the user has set a new destination a path to it is calculated
- If there is new laser data the grid map is updated
- If new obstacles in the path have been detected the path is updated
- The point where the robot should go to follow its path is calculated
- Control signals are sent to the robot

When the localization task is finished it returns an estimate of the pose that the robot had at the time that localization started. The new pose is then set to the one determined by localization plus the relative movement performed during localization, according to odometry. While localization is running the pose estimate is continuously updated by odometry.

The robot's path is often updated quickly, but since one can never be sure of how long it takes, the robot is stopped before updating the path.

The path finding algorithm always returns the next grid to go to. Since the grids are very small and the robot is unable to drive with high precision, always aiming for the next grid would not make the robot drive very smoothly. A solution to this problem is aiming a few grids ahead as long as there is no obstacle along the straight line between this grid cell and the robot's current position.

If the angle, α_{goal}, between the robot's heading and its goal is larger than 15 degrees the robot rotates on the spot towards the goal. Otherwise it drives forward and the rotational velocity, v_{rot}, is set to

$$v_{rot}(t) = 1.8\alpha_{goal}(t) + 3.6(\alpha_{goal}(t) - \alpha_{goal}(t-1)), \tag{54}$$

where $\alpha_{goal}(t-1)$ is the angle between the robot's heading and its goal from the last loop iteration. The proportional and derivative gains of this PD controller are set using the

Ziegler-Nichols method. If the laser or sonar detects something in front of the robot the forward velocity is set to zero.

The ActivMedia Robotics Interface for Application (ARIA) library [11] is used for all communication with the robot. The graphical user interface is built with the Qt framework [2].

7.1 Robot Interface

The following ARIA classes and methods are used.

- ArRobot is used for communication with odometers, sonars and motors
  - getX(), getY(), getTh() returns an estimation of the robot's pose determined by odometry
  - moveTo() updates the robot's pose estimate with the values from Online-SLAM
  - setVel() sets the translational velocity
  - setRotVel() sets the rotational velocity
- ArSick is used for communication with the laser range finder
  - getRawReadings() returns a list with 361 laser readings from the latest scan
- ArPose contains an x and y position as well as a heading

The central class of the interface is ArRobot. It has methods for receiving odometry and sonar data as well as sending commands to the motors. The method getPose() returns an ArPose object which contains the robot's position and heading determined by odometry. ArPose also has helper functions for calculating distance and angle to another pose. Both ArRobot and ArPose have methods getX(), getY() and getTh() for extracting the individual values. These values will be sent to the Online-SLAM. When localization have been performed in Online-SLAM and a more accurate pose is available, moveTo() is used to update the robot's idea of its pose so that the next pose estimate from the odometers will depend on the pose from localization and the movement since localization.

To get the robot to move, setVel() and setRotVel() are used to set the translational and rotational velocities.

ArSick is the class that handles laser data. It has methods that return the closest current reading in a region, currentReadingBox(). To get a list of all current readings getRawReadings() is used, which returns a list of ArSensorReading's. The range of these readings will be sent to the SLAM algorithms.

The robot's physical dimensions and some other parameters that are used in the program are stored in a file (p2d8.p) which is loaded on initialization.

All numbers are of type double. The units are millimeters for lengths, millimeters per second for velocities and degrees for angles.

7.2 Graphical User Interface

The following Qt classes and methods are used.

- QGraphicsScene contains the lines from the line map together with the icons representing the robot and its goal
  - mousePressEvent() and mouseReleaseEvent() are called when the user clicks the window and sets the position of the robot and the goal
- QGraphicsView is the window that displays the scene and that can be painted on
  - paintEvent() draws the grid map and the localization particles' positions

The lines in the map and the icons representing the robot and its goal are QGraphicsItems and belong to the graphics scene. The line map is read from a text file and each line is added to the graphics scene. The scene is then automatically scaled to fit into the window. The coordinates in the graphics scene are the same as the robot's coordinates except that the Y-axis is flipped. When the robot has moved or the user has clicked, update() is called to make sure the window is repainted. The grid map and the particles are painted directly onto the graphics view. Unlike the line map, the grid map is updated while the program is running. When this happens, the graphics view needs to be repainted.

7.3 Future development

The plotting of the grid map could be done more efficiently. To plot a map of 300 000 grids the way it is done now takes more than 100 ms and affects the performance of the robot. The user now has to click in the line map to set the robot's destination. It would be nice to be able to click in the grid map as well. The size and layout of the window is somewhat adopted to a certain map size and could be more dynamic. Also, the size of the line map and the grid map are not exactly the same, which can be annoying. When a new map is used, the programmer has to set some parameters defining the size of the grid. These could, and should, of course be calculated when the map has been loaded.

Possibly the updating of the robot's path should be done continuously and in its own thread. As of now it is done in the robot thread causing the robot to have to stop. Path updating is therefore kept at a minimum.

8 Position Accuracy

In order to test the position accuracy of the robot's movement, the following test was executed. A distance of approximately 10m was driven from different starting positions to different goals. The result can be viewed in Table 1.

Position Accuracy	
Test number	Distance to goal
1	0.57 m
2	0.80 m
3	0.53 m
4	0.35 m
5	0.32 m
6	0.20 m
7	0.09 m
8	0.19 m
9	0.60 m
10	0.25 m
Mean	0.37 m

Table 1: Position Accuracy

9 Robotic arm

The possibility of using a robot arm requires higher accuracy of the robot pose than have been achieved to deliver objects. A possible way to achieve better accuracy in the pose estimate at certain locations can be to mark known locations with some kind of landmark, e.g. reflective tape. The robot arm can then be hard-coded to perform specific movements and tasks. If you place a camera on the robot arm the demand of high accuracy in robot pose decreases. Hence using landmarks, i.e. reflective tape, the camera can detect and estimate the position of a landmark relative to the robot and use this information to perform different tasks. Another issue is where the robot arm should be placed. It should be mounted in the rear to prevent that the laser and sonar sensors are obscured. This requires that the weight of the arm is sufficiently low if the robot shall be able to move. The robot under consideration (IRB 120) is too heavy. There is also the issue of supplying the robot arm with power. Currently the robot arm needs a power cable which obviously becomes a problem on a mobile autonomous platform. Therefore integrating IRB 120 on the robot is not possible.

10 Future development

There have already been things mentioned regarding the future development for each subsystem. But something that has not been described so far is the problems which occurs when all the system s are merged together.

For instance, the path finding algorithm, particle filter and the gridmap works quite well separately, but when they are implemented together a lot of problems will arise. The quality of the gridmap can sometimes be very poor because of the uncertainty in the pose from the particle filter when new obstacles are detected. New obstacles are defined as objects that are not included in the line map which have been extracted from the offline-slam algorithm. Uncertainty in the pose will ruin the gridmap and the pathfinding algorithm must then find a new path which can take a lot of time. This will go on until the obstacle is gone or the robot has passed it.

At the moment, this problem has been solved by solid walls in the initial grid map, i.e. the obstacles detected from the offline-slam can never be removed in the grid map. Probably this is not the best solution since dynamic objects will always be fixed in the grid map even though they have been moved.

Another way of reducing the "map destruction" when the pose has some uncertainty, is the max range limitation. A small uncertainty in theta will correspond to a quite large misestimation of the detected obstacle when the laser range is increasing. Therefore, it is possible to decrease this misestimation by reducing the field of view to 10 meters.

The big bottleneck of the MCL algorithm is the simulation of laser measurements for each particle. When simulating a single measurement the intersection between the simulated laser ray and every line in the line map is calculated with the current solution. This means that the robot cannot handle maps built from many lines without loss of performance. A solution to this problem could be to divide the line map into cells. Here each cell would have information about which lines that could be intersected if the robot is positioned in that cell. Reducing the number of line intersection calculations per measurement in this way would not only make the robot handle large environments but also allow it to have a more detailed line map yielding better localization.

Further on, the dynamic re-calculation of the path is very time consuming at the moment, yielding response times up to 10 seconds at best even for relatively small changes in the map. The cause of this problem is yet to be found but is thought to be related to chosen programming solutions, as described in the pathfinding chapter.

The current path finding algorithm always finds the shortest path to the goal, if there is one. This may not be the most suitable path for the robot though. Since there is no millimeter precision in driving and localization it is probably a good idea to drive in the middle of corridors and door posts and not go as close to corners and objects as possible when passing them. Localization can be hard when the robot turns a lot, because errors in odometry tend to get larger then. Therefore, something that turns the planned path into a smooth trajectory could be an improvement. If the goal is unreachable the path planner will not detect this before it has tried every possible way, which takes a lot of time. Maybe something can be done about this. Otherwise the user should at least be able to select a new goal and make the path planner start over.

The sonar is only used to make the robot stop when there is something directly in front of it. More uses of it can probably be found.

To prevent the robot from going to undesired places there should be a way to add restricted areas, e.g. by clicking in the grid map.

It would be nice if the robot was able to autonomously explore and map an unseen environment.

Driving the robot around with a laptop connected by a serial cable is all but satisfying. Some form of wireless communication is recommended. This could also allow multiple users to request the robot's services. Then it would be interesting to have the robot make more high level plans, like in which order to deliver a couple of letters to do it in the shortest amount of time.

References

[1] Bresenham's line algorithm. `http://en.wikipedia.org/wiki/Bresenham's_line_algorithm`. [Online; accessed 22-oct-2010].

[2] Nokia Corporation. Qt - a cross-platform application and ui framework. `http://qt.nokia.com/products/`. [Online; accessed 2-dec-2010].

[3] Austin Eliazar Eliazar and Ronald Parr. Learning probabilistic motion models for mobile robots. In *In Proc. of the International Conference on Machine Learning (ICML)*. ACM Press, 2004.

[4] K. Lee. Application of the hough transform. `http://www.cs.uml.edu/~lkyewook/hough/`. [Online; accessed 06-Oct-2010].

[5] F. Lu and E. Milios. Globally consistent range scan alignment for environment mapping. *Autonomous Robots*, 4:333–349, 1997.

[6] M. Mäkelä and J. Kurhila. The bresenham line-drawing algorithm. `http://www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html`. [Online; accessed 1-nov-2010].

[7] Jorge L. Martinez, Javier Gonzalez, Jesus Morales, Anthony M, and Alfonso J. Garcia-cerezo. Genetic and icp laser point matching for 2d mobile robot motion estimation.

[8] Daniel Sack and Wolfram Burgard. A comparison of methods for line extraction from range data. In *In Proc. of the 5th IFAC Symposium on Intelligent Autonomous Vehicles (IAV*, 2004.

[9] W.Burgard D. Fox S.Thrun. *Probabilistic Robotics.* MIT press, 2006. pp. 94-96, 281-305.

[10] H. Choset K. M. Lynch S. Hutchinson G. Kantor W. Burgard L. E. Kvarki S. Thrun. *Principles of Robot Motion.* MIT press, 2005. pp. 328-337.

[11] Dimitri van Heesch. Aria reference manual, 2002.