NTNU
Norwegian University of
Science and Technology

Autonomous Multi-Robot Mapping

Henrik Kaald Melbø

Master of Science in Cybernetics and Robotics
Submission date: June 2017
Supervisor: Tor Engebret Onshus, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem Description

The focus of this thesis will be on the server-side handling of data collected by multiple robots exploring a labyrinth. The aim is to improve the map estimate created from the data. The current system simply plots the data from each robot into a common map, with no regards to its accuracy. The server also need exact input from the user on the starting pose of the robots, a quite limiting restraint with a high possibility of introducing errors even before the robots start moving. In this thesis the following improvement to the system will be implemented:

- An improved SLAM algorithm
- No need for initial pose information
- An improved path planning algorithm will be discussed

The idea is that a proper SLAM algorithm will improve the pose estimate and therefore create an overall more consistent map.

By removing the initial pose constraint, a large source of uncertainty can be removed, as each robot can be initialize in separate reference frames. This however requires some method to build a consistent global map from multiple local, and partially overlapping maps.

Lastly, a new approach for the multi-robot exploration algorithm is needed as the current implemented algorithm delegates resources in an inefficient way. Possible exploration algorithms will therefore be discussed.

Summary

In this thesis a multi robot algorithm for Simultaneous Localization and Mapping (SLAM) using a Rao-Blackwellized particle filter was implemented. The particle filter was paired with a map-matching algorithm for calculating particle weights. The particle filter was then used to improve the map estimate of each robot exploring and mapping a labyrinth. The improved maps from each robot was then subjected to a map merging algorithm, which attempts to merge the partial maps from each robot together. The merged map will now be a better estimate of the real world, and remove the constraint that robots need prior knowledge of each others relative position.

The particle filter uses a motion model based on the robots odometry readings to estimate the current pose of the robot. Each particle will move according to the motion model, but will in addition incorporate some Gaussian noise. This will cause a distribution of belief states to appear, as each particle have a different trajectory over time and therefore produces their own version of the map. When a particle returns to a previously known position, a comparison between the current observations and the previous observation is performed using a map matching algorithm. The match between these map gives a score, also called weight, to the particle. The better match, the higher the score. Based on each particles score, it is chosen as a progeny, or discarded in a sequence called resampling. The particle with the highest score at the end of the exploration has a high likelihood of being a good representation of the robots true state.

Sammendrag

Denne oppgaven omhandler implementasjonen av en algoritme for simultan lokalisering og kartlegging (SLAM) ved bruk av flere roboter. Algoritmen bruker et Rao-Blackwellisert partikkelfilter sammen med en algoritme som sammenligner kartene for så å kalkulere partikelenes vekt. Partikkelfiltret brukes så til å laget et forbedret kart estimat for hver av robotene. De forbedrede kartene blir så sydd sammen til et globalt kart. Dette fjerner restiksjonen som krever at robotenes relative posisjon må være kjent på forhånd.

Partikkelfiltret bruker en bevegelsesmodel basert på målt endring i posisjon over tid. Hver partikkel beveger seg så etter denne modelen, men legger i tillegg til Gaussisk støy. Dette vil føre til at partikklene går forskjellige veier og vi får en distribusjon av mulige tilstander. Hver partikkel har nå sitt eget unike kart. Når en partikkel returner til en tidligere kjent posisjon vil den sammenligne de nye observasjonene med de den gjorde sist gang den var på samme sted. Avhenging av hvor bra kart dataen stemmer overens, får hver partikkel en poengsum kalt vekt, jo bedre overlapp med de gamle observasjonene, jo høyere vekt. Partikkler med høy vekt blir så valgt ut i en prosess kalt resampling, siden de har en god sannsynlighet for å være et godt estimat på robotens faktiske posisjon.

Conclusion

The particle filter is shown to improve the pose estimate of individual robots when tuned correctly. The improved pose estimate again manifest itself in a better map estimate. It is however necessary to employ a decent amount of particles for this attempt to be successfully, which comes at a cost of additional computational power. In addition, the uncertainty model parameters must not be to conservative, as to be unable to predict the robots actual motion. Lastly, the map matching algorithm, responsible for weighing the particles, must be tuned appropriately. A large value for the exponential term used in the scoring algorithm gives fast convergence but might wrongfully eliminate relevant states. A to conservative value of the term will cause the particle filter not to converge, and in turn perform badly.

The map merging algorithm uses an approach similar to simulated annealing to optimize a cost function based on the Manhattan distance between corresponding observations, in addition to a term that values the total overlap of the maps. The total overlap term is added to avoid over fitting when the corresponding maps are only partially overlapping. Each iteration of the algorithm a matrix transformation rotates and translates one of the maps slightly. The cost function then evaluates this neighbouring state, to see if it is a better fit than the last state. This part is similar to hill-climbing algorithms. Since there potentially are many local minima of the function, a random selector may chose a less optimal transformation at the beginning of the run, but this becomes less likely for each iteration, known as cooling. If this random selection is "cooled" gradually, the state estimate will settle down into some local, and hopefully, a global minima.

It is shown that the individual partial maps produced from each particle filter can be merged into a global map using the map merger algorithm. The map merging algorithm performs perfectly on two fully overlapping maps, but it is unfortunately prone to certain issues when applied to partially overlapping maps. Situations such as symmetry, over fitting and large sensor errors makes the map matching algorithm perform suboptimal. The approach also suffers from the same problem as any other stochastic optimization algorithm, in that suboptimal solutions may be accepted.

Acknowledgements

I would like to thank Tor Onshus for his support through the process of writing this thesis. And also give a big thank you to the rest of the team that worked on various parts of this project. Your unique insight and feedback has been greatly appreciated.

Contents

List of Figures

Chapter 1

Introduction

A fundamental problem when working with autonomous robots are navigation in unknown or changing environments. For the robot to manoeuvre in the real world and solve problems, a good estimation of the robots surroundings are needed. This can be acquired by using sensors to measure the environment and then constructing a map. The robot then uses this map along with its pose (position and orientation) to successfully navigate the world. In real life however, there will always be uncertainty in all measurements. It can therefore be assumed that both the robots pose and the measured observations are an inaccurate description of the real world. This inaccuracy will accumulate over time, as the robot makes new assumptions on false premises.

The so called Simultaneous Localization And Mapping (SLAM) problem is therefore to simultaneously build a map of the surrounding, and keeping track of the location of the robot. This is an inherently difficult problem due to the correlation between the uncertainty in the pose estimate and the uncertainty in the map estimate. SLAM defines a family of approaches, whose purpose is to make an accurate prediction of the pose of the robot at all times, as well as building an accurate map of the environment.

A SLAM algorithm needs a motion model, to keep track of how the robot moves, and also an observation model, to correctly apply the sensor data to build a map. Many paradigms in SLAM exist, but most are developed from a probabilistic view of the world based on Bayes filter[1]. Some major paradigms in SLAM includes, Kalman filter based SLAM, information filter based SLAM and particle filters based SLAM.

In this thesis a mulitrobot SLAM algorithm for a setup of 4 LEGO-robots exploring a maze using infra-red sensors to map distance, will be implemented. The goal is to improve the map estimate.

1.1 Previous Work

The collaborating LEGO-robot project has been around at NTNU since 2004. The aim was to create a system that efficiently maps an area using relatively cheap sensors and robots.

There has been much work done on this subject through the years, a short summary follows.

The first robot was constructed, using Lego as the framework around an AVR microcontroller, by Håkon Skjelten in 2004[4], the work was further developed in 2005 when Helgeland made the system autonomous[5]. In 2006 a navigation algorithm was implemented by Bjørn Syvertsen[6]. In 2007 Kallevig expanded the sensor capabilities of the robot[7] and Schrimpf improved the real-time characteristics of the robot[8]. In 2008 Bakken made the second, NXT based Lego robot[9], Næss made improvments on the robot power system[10], Maggnusen created a simulator for easier testing and debugging of the software[11] and Haugedal worked on an implemntation for a docking station for the robot[12]. In 2009 the mapping and navigation-algorithm were modified and improved by Tusvik[13], Kristiansen worked on improving the mapping system[14], and Tøraasen worked on using cameras as sensors[15]. In 2011 Hannaas worked on an interface in Matlab for two robots using an IR-sensor[16]. In 2013 Homestad worked on integrating the the NXT robot into the system[17]. In 2014 Halvorsen imporved the system so it handles two robots[18]. In 2015 Ese implemeted FreeRTOS on a robot[19] and then in 2016 he implemented the BLE module[20]. In 2016 the whole server was rewritten and a new Arduino based robot was introduced by Andersen and Rødseth[21], and a new Simulator and an improved navigation algorithm based on A* was implemented by Thon[22].

For a complete picture, the reader is recommended to consult the refereed articles, which is provided in the appendix.

1.2 Hardware

The system consist of multiple robots used to explore the environment using infra-red (IR) sensors mounted on a rotary tower. The robots collects range data and transmits the data to a server. The servers task is to build a map of the environment given the ranged data, keeping track of each robots pose, and also to plan the best trajectory for all robots.

The system so far consist of three Lego robots, one based on an Atmel AVR microcontroller, another using the Lego proprietary system NXT, and the last one is based on an Arduino. Each robot is equipped with the four IR range sensors placed with 90 degrees spread on the rotating tower. The max range of the sensors is 80 cm. In addition, work is currently under way to include a flying drone with a camera to detect edges and aid mapping. The NXT is shown in figure 1.1, the black box on top is the IR sensors.

On the server side there exist a GUI and a simulator, both are programmed in Java. The server implements an A* algorithm for navigation and uses grid-maps to plot the acquired data. The robots communicate with the server using nRF51 Bluethoot Low Energy (BLE), as can be seen by the blue dongle in on the top left of the robot in figure 1.1.

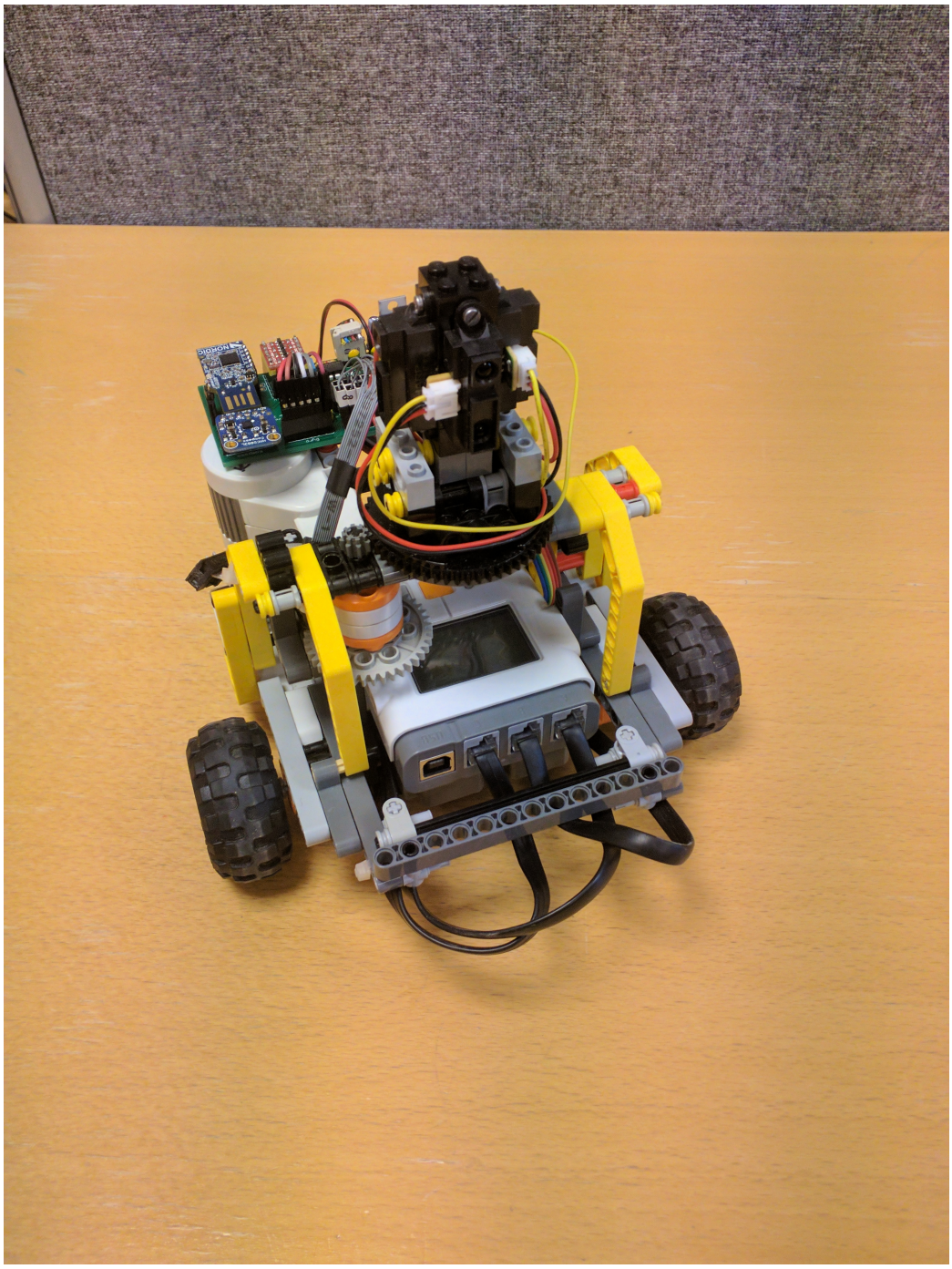

Figure 1.1: The robot running on the NXT system.

Chapter 2

Localization and Mapping

Any autonomous robot, set on exploring the real world, will experience some problems. To be able to successfully interact with the world, a notion of how the world looks like and also where the robot is positioned in relation to the world, is needed. The world may be static, as in a closed space with no moving objects, or it may be dynamic, with people and objects moving as time passes. In any case, the robot needs a way to keep track of itself and any objects in the world. In a ideal world, the robot would be able to perfectly sense its own movement, and at the same time, build a perfect internal map of the world using some form of sensors. Unfortunately, the world is a noisy place, and the robot may end up being mislead by false or noisy information.

2.1 Localization

Localization is the science of estimating the robots pose given a set of actions. If the starting pose of the robot is known and the goal is to keep track of where the robot moves, we deal with what is known as position tracking or local localization.

The localization problem in essence tries to solve the seemingly simple probability density function (PDF) shown in equation 2.1.

$$p(x_t|m, z_{1:t}) \tag{2.1}$$

That is the Bayesian statistic posterior of the current pose x_t, given all previous observations $z_{1:t}$ and the map m.

To track a robots motion, some form of motion model is needed. The term odometry is often used for motion model that uses motion sensor data (distance travelled and direction) to estimate change in the position over time. An other approach would be using GPS to track the absolute position, but GPS are ill suited for indoor use. In any case, all sensors are inherently prone to noise and biases, and some form of processing is needed to efficiently utilize the acquired data. For wheeled robots rotary encoders are often used to produce odometry data, as they are cheap and simple. Other vehicles may use accelerometers, and integrate over time to get change in position, while flighted drones that manoeuvre in 3D space may use a inertial measurement unit (IMU), a combination of accelerometers and gyroscopes, to estimate its movement.

Common for all sensors is that they will always impart noise and errors which will lead to inaccurate odometry measurements. The robot may think it is at one location while the actual location is far of. This problem can be avoided by having a good map of the environment to navigate and correct the pose by. It is worth to mention at this point that if it were given an accurate map of the environment, location would be trivial. You could in fact take a robot, place it at an unknown location and have the robot figure out where it was based on the map it was given, this is known as global localization[1].

2.1.1 Localization Strategies

Some of the most common approaches to localization will now be discussed, it is important to note that localization assumes that a preexisting map is given. For brevity, this section will have few detail and no mathematical derivations. The interested reader is recommended to have a look at "Probabilistic Robotics" by Thrun et al.[1]. The book gives a good overview of localization and other important themes of this thesis such as a refresher in Bayesian statistics, and also gives vigorous mathematical derivations.

One common approach to localization is called Markov localization. This approach is a probabilistic algorithm, based on Bayes theorem, which instead of maintaining a single hypothesis as to where in the world a robot might be, maintains a probability distribution over the space of all such hypotheses. Figure 2.1 shows Markov localization in action. Each frame depicts

the position of the robot in the hallway and where it believe it is located, written $bel(x)$. The observation model $p(z_t|x_t)$ describes the probability of observing a door at the different locations in the hallway. The robot updates its belives after each observation[1]. Markov localization can deal with both the local and global localization problem.

Another approach uses the Extended Kalman Filter (EKF) on a feature based map to estimate the robots position, and update the believes mean μ_t and covariance Σ_t of the pose for every new feature observed. Feature based maps will be discussed further in the section on mapping. There is also a version known as Unscented Kalman Filter (UKF) localization that uses a unscented transform to linearise the the motion and measurement models to improve stability and convergence rate[1].

Monte Carlo Localization (MCL) is another interesting approach, based on what is known as a grid-map. The grid-map will also be discussed in detail in the secton on mapping. The MCL represents the belief state $bel(x_t)$ by a set of random particles and is therefore also known as a particle filter. The particles are random samples from the motion model and they represents all (or most of, depending on the sample size) the different states the robot may be in. The particles are then evaluated (weighted) using the observations the robots make to find the most probable position of the robot. It is worth noting that this approach needs not extract features from the environment, raw sensory data is enough (this is a property of the grid-map). MCL is also non-parametric, and can solve both the local and the global localization problem. Figure 2.2 illustrates the approach in the same way as figure 2.1 illustrates Markov localization. Note that the particles are uniformly distributed at the beginning, and gets more concentrated around the possible solutions as new observations are made[1].

2.1.2 The Motion Model

Any localization algorithm is dependent on predicting the robots movements. In essence the motion model tries to find the PDF as shown in equation 2.2.

$$p(x_t|x_{1:t-1}, u_{1:t}) \tag{2.2}$$

Here x_t is the current pose, $x_{1:t-1}$ is all the previous poses, and $u_{1:t}$ is the input sequence.

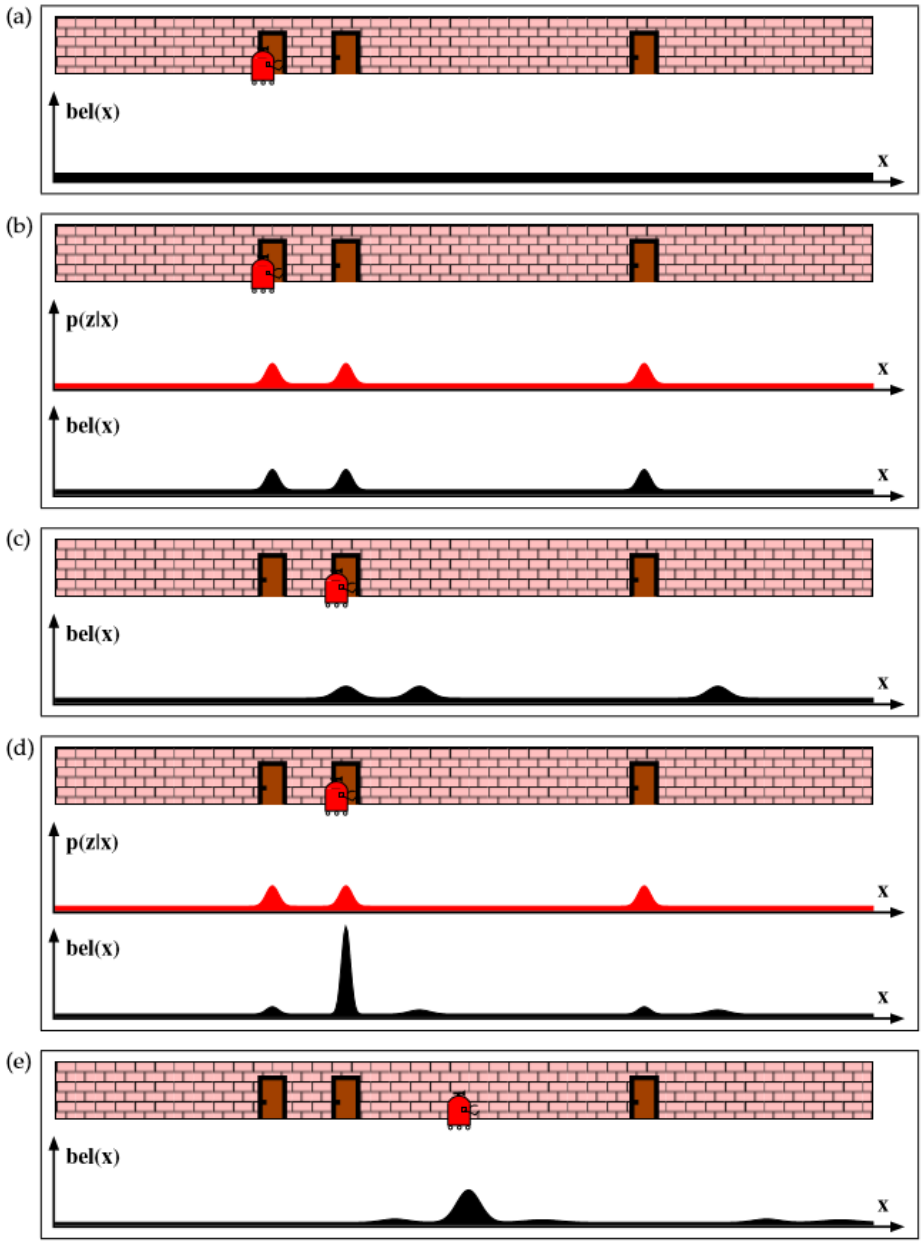

Figure 2.1: The basic idea of Markov localization[1].

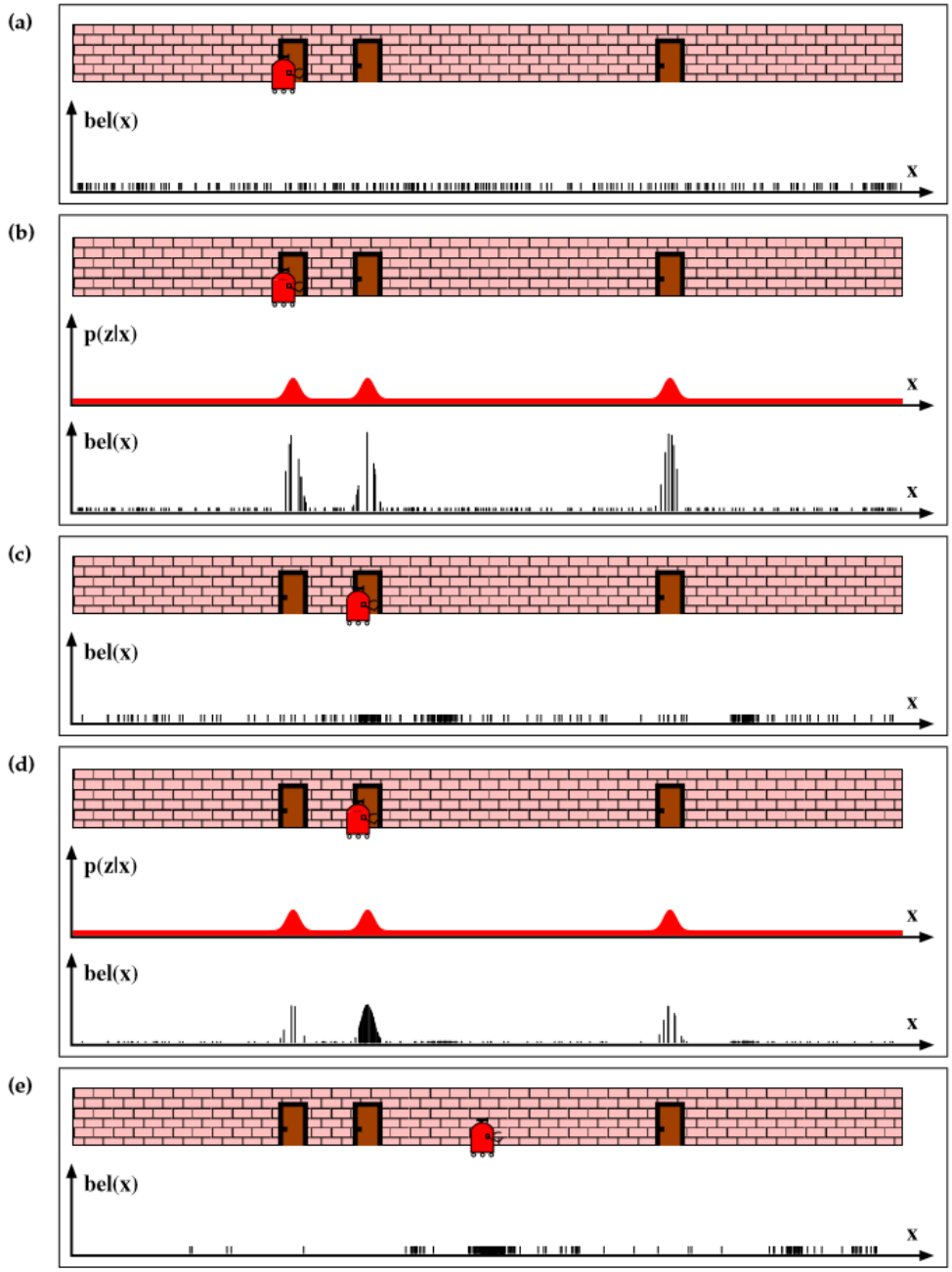

Figure 2.2: Monte Carlo Localization, a particle filter applied to robot localization[1].

There are two dominant approaches in robot kinematics, the velocity motion model and the odometry motion model. The velocity motion model assumes that we use rotational and translational velocity as set point for the robots motor driver. Drivetrains that uses this approach include the differential drives, Ackerman drives, and synchro-drives[1]. The model assumes that the robot is capable of following the given input, and is prone to errors due to drift, slippage and dead zones.

The odomtry motion model uses odomtry data, usually obtained by integrating wheel encoder information, to calculate the robots motion. This model is based on output (sensor data) instead of input commands, but it is common to model the odometry data as if it were control signals. The odomtry motion model is in general a better estimation than the velocity motion model. The model is depicted in figure 2.3, where δ_{rot} is the measured rotation of the robot and δ_{trans} is the measured translation of the robot. The odomtry data and the posees are connected when the robot moves from pose (x, y, θ) to the new pose (x', y', θ') as shown in equation 2.3 to 2.5[1].

$$\delta_{trans} = \sqrt{(x' - x)^2 + (y' - y)^2} \tag{2.3}$$

$$\delta_{rot1} = atan2(y' - y, x' - x) - \theta \tag{2.4}$$

$$\delta_{rot_2} = \theta' - \theta - \delta_{rot1} \tag{2.5}$$

Both of theses motion model can easily be calculated and a probability distribution can be made by introducing Gaussian noise, or other form of noise if the noise attributes are known. The distribution can then be used to make probabilistic estimations on where the robot is currently located.

2.2 Mapping

Mapping is the task of modelling the environment. In all cases this requires data from the surroundings obtained using some form of sensors, such as LIDAR, camera, sonar, tactile sensors or any other device capable of retrieving information from the surroundings.

Maps are needed for any autonomous unit to be able to move around and complete tasks. Often the map may be given in advance from blueprints,

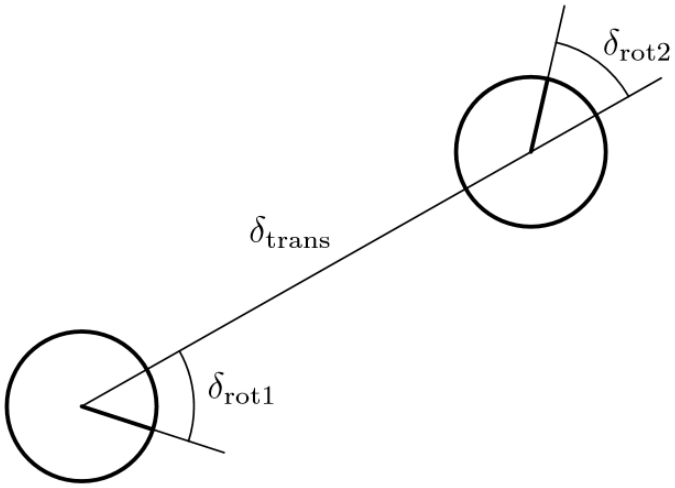

Figure 2.3: The odomotry motion model[1].

satellite data or other sources. This information gives the robot the opportunity to plan its actions and complete complex task with relative ease. In the case of an unknown environment, or in the case of uncertain environments (blueprints may be outdated, and does not include furniture etc.), mapping may be used to gather information about the environment. The problem with mapping is that it requires information on the robots pose, and the pose of the robot is not always exactly known, causing the map to be generated to be incorrect. An example of a occupancy grid-map can be seen in figure 2.4. Here drift in the pose causes several observations of the same walls to be missaligned adn produce a poor map estimate. It is worth noting that given the exact pose of the robot at all times, mapping is trivial[1].

2.2.1 Mapping Strategies

The mapping problem in essence tries to find the PDF shown in equation 2.6.

$$p(m|z_{1:t}, x_{1:t}) \tag{2.6}$$

Where m is the map, $z_{1:t}$ is a sequence of measurements, and $x_{1:t}$ are all the previous poses.

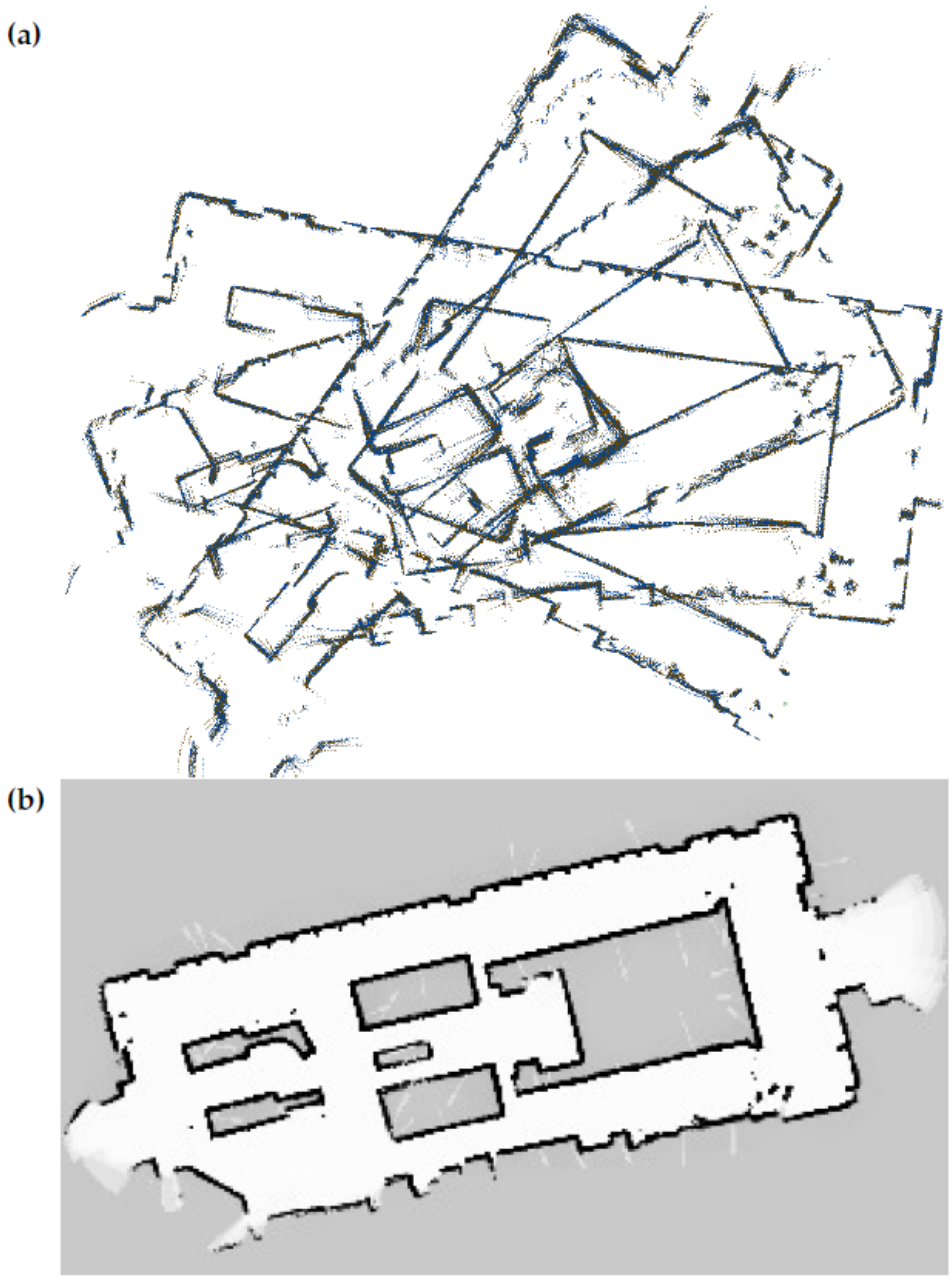

Figure 2.4: Example of a grid-map with uncertain poses (a) and known poses (b)[1].

There are many ways to construct a map, the main paradigms in robotics are landmark based mapping and occupancy grid based mapping. Landmark based mapping or feature based mapping, is based on extracting distinct objects from the real world, such as corners, trees, doors or other easily identified markers in the environment. These can then be used to estimate the position of the moving robot. As opposed to using all the acquired information from the sensors. This might be less computationally taxing, but the features must be extracted from the sensor data, which is not always easy. This approach also suffers from a major problem called data association.

Data association occurs if the robot pose is not exactly known, and may cause the robot to be unable to differentiate between different landmarks. This again may cause the robot to believe it is in a completely different location than it actually is. This is illustrated in figure 2.4, where two different data associations can be made between three star shaped landmarks. This causes major problems when making maps and navigating, and can be fatal for the operation of the robot.

Occupancy grid (grid-map) based mapping is another way to model maps. In 2D space, a floor plan like model of the environment is produced. As can bee seen in figure 2.6, the world is discretized into smaller squares. Each square can take on a value that describes the likelihood of there being an obstacle in that square. Other approaches simply uses a binary value. The advantage of this approach is that it is more intuitive than a landmark based approach, but it requires more data to be stored and may therefore be more computational taxing. Problems may also occur with the resolution of the map, as the real world is not square in nature, and a cell might be partially filled. The larger the size of each cell in the occupancy grid, the lower the accuracy of the map will be, but will requires less data to be stored.

A nice survey of single robot indoor mapping methods is done by Thrun et al.[23].

2.2.2 The Observation Model

When it comes to mapping, there are different types of sensors used to gain knowledge about the environment. Some sensors only gives the the general direction, or bearing, of obstacles, while other gives both the bearing and

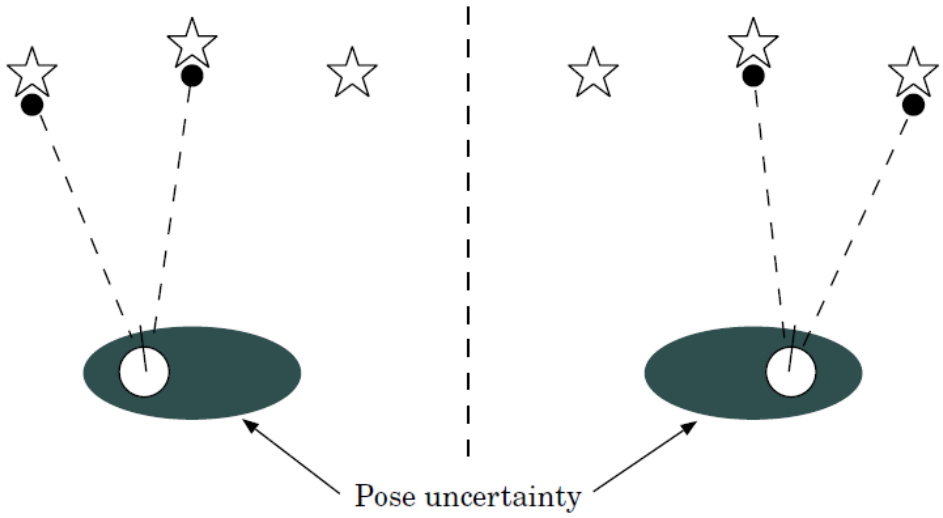

Figure 2.5: The data association problem in SLAM[1].

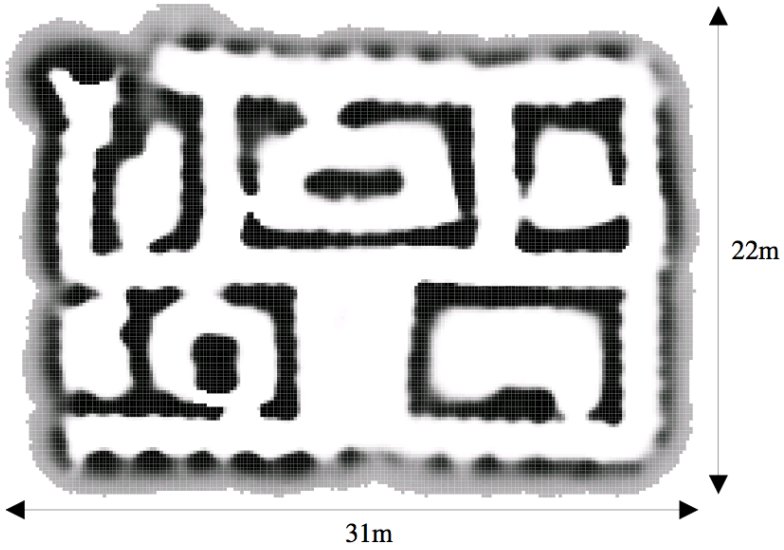

Figure 2.6: Occupancy grid-map of the 1994 AAAI mobile robot competition arena. The grid cells divisions are bearly visable[1].

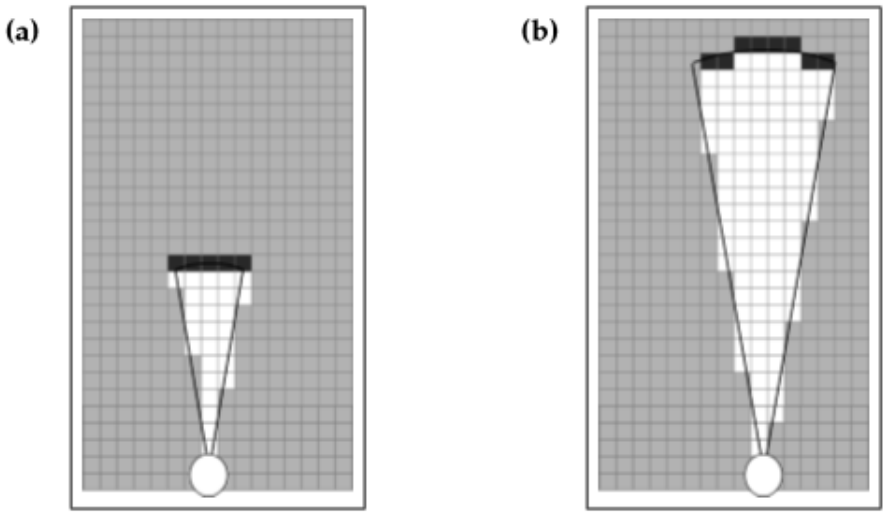

Figure 2.7: The inverse sensor model at two different ranges for a sonar. The darkness of each grid cell corresponds to the likelihood of occupancy[1].

range to obstacles in a wide cone in front of the sensor. The laser-based ranged sensor and sonar is of the latter category.

In essence, any observation model tries to find the PDF in equation 2.7, finding the current observation z_t given the current pose x_t.

$$p(z_t|x_t) \tag{2.7}$$

For range and bearing type sensors, such as a sonar or LIDAR, the inverse range sensor model is often used. The model takes in the angle of the sensor and the distance to an observation and gives out the set of free and occupied cells in a grid-map as shown in figure 2.7.

When using a simple laser range finder, where only one point is being evaluated at a time, and no specific sensor characteristics are known, the simple approach is to add the discovered point straight into the map. This can be done by the adding the robots position to the measured position and taking into account the angle of the sensor relative to the robot. The simple way of calculating this would be a transformation matrix T, which is the combination of a rotation matrix $\boldsymbol{R}$ and a translation matrix $\boldsymbol{t}$ as shown in equations

2.8 to 2.10.

$$\boldsymbol{R} = \begin{bmatrix} cos\ \theta & sin\ \theta \\ -sin\ \theta & cos\ \theta \end{bmatrix} \tag{2.8}$$

$$\boldsymbol{t} = \begin{bmatrix} x_t \\ y_t \end{bmatrix} \tag{2.9}$$

$$\boldsymbol{T} = \begin{bmatrix} cos\ \theta & sin\ \theta & x_t \\ -sin\ \theta & cos\ \theta & y_t \\ 0 & 0 & 1 \end{bmatrix} \tag{2.10}$$

Here θ is the angle between the robot and the sensor, and the variables x_t and y_t is the measurement position, relative to the robot. The measurements position in the map would now simply be $\boldsymbol{Tp}$, where $\boldsymbol{p} = [x, y]^T$ is a vector of the robots position. This transformation however, assumes rotation around the origin. This can simply be mended by first translating the robots pose to the origin, rotate and then translate back to its original position[24].

2.3 SLAM

It has been stated that given a accurate map, location is trivial, and given accurate pose, mapping is trivial. The grater challenge is making a map, while the pose is uncertain, this is referred to as the Simultaneous Location and Mapping (SLAM) problem. The SLAM problem is an inherently difficult problem due to the fact that:

- Robot path and map are both unknown
- Map and pose estimates are correlated
- The mapping between observations and the map is unknown
- Picking wrong data associations can have severe consequences

The SLAM problem in essence tries to find the PDF shown in equation 2.11.

$$p(x_t, m | z_{1:t}, u_{1:t}) \tag{2.11}$$

Finding the current pose x_t and the map m of the robot, given the set of measurements $z_{1:t}$ and the sequence of motion input $u_{1:t}$.

If compared to the location problem in equation 2.1, and the mapping problem in equation 2.6, this is a relatively more complex problem. Especially when you realize that solving the map is dependent on the pose and solving the pose is dependent on the map.

Two main paradigms exists in SLAM: Full-SLAM, where the entire path is calculated and the map is construed after data acquisition is done, and online-SLAM, where the previous poses are a continuous estimate and a map is built on the fly. Both approaches has their own merits. Full-SLAM can be said to be easier, as the data only needs to be processed and often no real-time requirements needs to be fulfilled, while the online-SLAM is more useful if the map is also incorporated in the robot navigation system[1].

Figure 2.8 shows a Bayesian network graph representation of the SLAM problem. Each pose x_t is updated using the motion model via the motion input u_t. The robot than makes measurements z_t from the surroundings and they are then integrate into the map m. The process is then repeated for the next motion input.

Figure 2.9 show how SLAM relates to localization and mapping. In later sections, path planning will be further explored, and the problem then moves into the "integrated approaches" section of the diagram.

2.3.1 SLAM Strategies

There exist numerous methods to solving the SLAM problem, all depending on the specific flavour of problem. What sensors are in use? Are we using grid-map, feature based maps or something completely different? Are we doing online- or full-SLAM? Over the years, many different approaches has cropped up.

A popular approach for feature based maps is the Extended Kalman Filter (EKF), which applies the EKF to online-SLAM using maximum likelihood data association. The EKF assumes the noise is Gaussian for motion and perception. EKF integrates out the robots pose as the robot moves, and therfore only keeps track of the current pose[1].

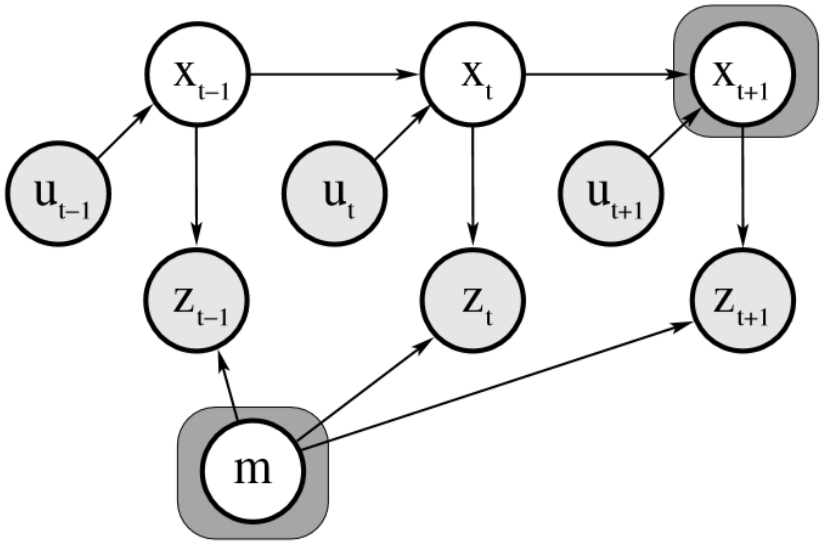

Figure 2.8: Bayes network graph of the SLAM problem[1]

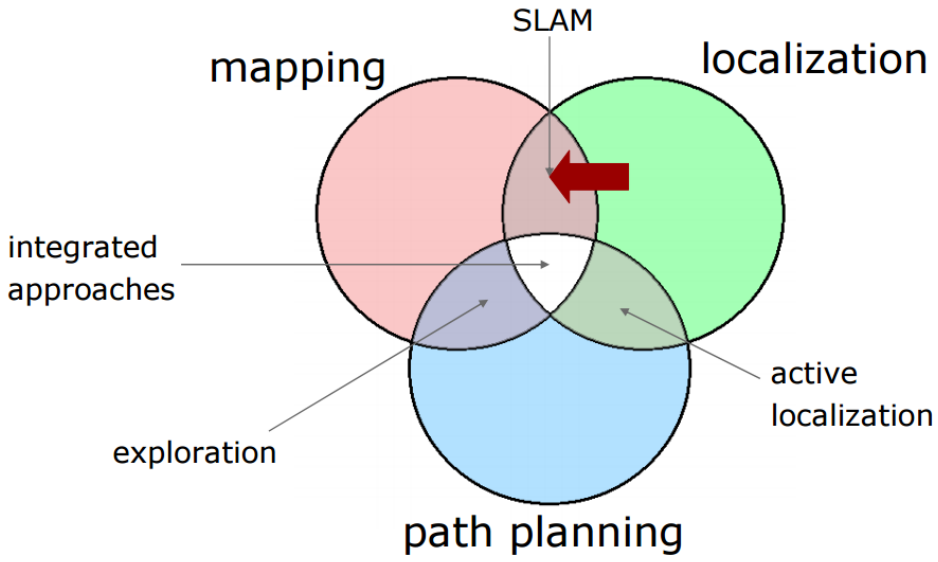

Figure 2.9: A Venn diagram of how SLAM fits inn among localization and mapping[2].

Another popular approach, that solves the full-SLAM problem, is the graph based approach known as Graph-SLAM. In Graph-SLAM the robot poses are represented as a graph where the nodes correspond to the poses of the robot at different points in time, and the edges represent constraints between the poses. The latter are obtained from observations of the environment or from movement actions carried out by the robot. The whole problems boil down to solving a large optimization problem. The graph can be shown to be a sparse graph of nonlinear constraints. Graph-SLAM needs to keep track of all poses and measuremtns at all times, but since it is an ofline approach, does not need to do computations while collecting data[1].

The Sparse Extended Information Filter (SEIF) implements an information solution to the online-SLAM problem and solves feature based maps. As EKF, this algorithm also only needs to keep track of the current pose and the maps, and like Graph-SLAM maintains an information representation of all knowledge[1].

Particle filters or Sequential Monte Carlo (SMC) filters, is a set of genetic-type statistical approaches to solving the filtering problem. Interestingly this approach can be used to solves both the online- and full-SLAM problem simultaneously. This approach will be discussed more in details in the following section.

2.4 Particle Filters

Of particular interest is the particle filter, as it solves the SLAM problem using a grid-map based approach. Particle filters has become one of the most popular approaches to solving the SLAM problem in modern robotics[1]. The particle filter is a tool for tracking the state of a dynamic system, even when the state is not fully observable. Particle filters is in practice a Bayes filter that uses a prediction and update cycle to estimate the state of a dynamical system from sensor measurements, and has similar applications as the Kalman filter, but handles large dimensionality better.

The particle filter estimates the posterior over all state sequences, as shown in equation 2.12.

$$bel(x_{0:t}) = p(x_{0:t}|u_{0:t}, z_{0:t}) \tag{2.12}$$

The state space will become large extremely quickly, and it is therefore not advised to calculate this posterior for all possible states. Instead, random samples are drawn from the motion model. By using Bayes theorem and Markow assumptions, we can show that the posterior can be calculated as shown in equation 2.13.

$$bel(x_{0:t}) = \eta w_t^{[m]} p(x_t|x_{t-1}, u_t) p(x_{0:t-1}|z_{0:t-1}, u_{0:t-1}) \tag{2.13}$$

Where η is a normalizer and w_t is the particle wheight. The whole derivation can be found on page 103 of "Probabilistic Robotics"[1].

Each particle is given an importance weight according to its likelihood of being a correct estimate. This importance weight is used in a process called resampling.

A large source of error in particle filters are related to the fact that we do random sampling. Whenever samples are drawn from a continuous distribution, the statistics of the sample will vary from the statistics of the original distribution. Variability due to random sampling is called variance. A larger set of samples, meaning more particles, will therefore be a more accurate approximation, ie. have a lower variance[1]. The resampling process, as will be discussed in a later section, will further amplify this variance.

2.4.1 Particle Weighting

Every particle filter needs a way to determine if any particular particle is a good estimate of the real world or not. This process is called weighting, as each particles is given an importance weight based on how likely it is to be a good estimate of the true state. There are several ways of weighting the particles.

One approach used to estimate a particle weight is called scan-matching. Scan-matching aims to superimpose two grid-maps (or images for that sake) using a rotation ($\boldsymbol{R}$) and translation matrix ($\boldsymbol{T}$). This is handy, as it gives us a tool to compare two map estimates against each other. Scan-matching can also provide a locally consistent pose correction. Some well know implementations of scan-matching includes the Singular-Value Decomposition (SVD)[25] and the Iterative Closest Points (ICP)[26] algorithms. Scan-matching however requires additional computational resources and might not

be the best real-time approach. The approach is also multi-modal, meaning more than one pose can correctly be identified as the current one. Another shortcoming of this approach is that they often expect two point sets of same size, which any two sensor reading can not guarantee.

Another approach to weighting particles is called map matching. Here two maps will be compared directly and the correlation between the maps gives an indication off how good the particles estimate is, and therefore correlates directly to the weight. A simple way of doing this is scoring $+1$ if two points corresponds and -1 if they don't. It is of interest to not include unexplored grid cells, as they include no information, any attempt to compare with an unexplored cell will therefore return zero value. The final score is then used to compute the weight according to equation 2.14.

$$weight = e^{(c \cdot score)} \tag{2.14}$$

The scoring system gives an exponential weighting distribution which can be adjusted by the paramter c[27].

2.4.2 Resampling

A major optimization for particle filters is resampling, also known as importance sampling. As the uncertainty increases, the particle will spread more and more out in space. This will eventually cause the particles to be spread very thin, and ultimately an infinitely large amount of particle is needed to cover the whole state space. Instead of using infinitely many particles (and infinite RAM), a selection based on the most probable particles are made, and an almost genetic "survival of the fittest" approach is used to propagate these particles onward. This of course introduces a loss of diversity as we remove some particles and may even gain several copies of the same (highly probable) particles. As mentioned earlier, this constitutes an amplification of the sample variance. The set prior to resampling is drawn from the motion model and represents the prior belief, the resampled set represents a more accurate description of the what the posterior belief looks like.

In essence the resampling step modifies the weighted approximate density to an unweighted density by eliminating particles having low importance weights and by reproducing particles having high importance weights.

Some common approaches includes, low variance resampling, multinomial resampling, stratified resampling, systematic resampling, and residual resampling[28].

An interesting approach is the so called low variance resampling. The basic idea is that the selection involves a sequential stochastic process. Low variance resampling computes a random number and selects samples according to this number with a probability proportional to the sample weight. Some of the advantages to low variance resamlping are that it covers the sample space in a systematic manner. Also if we resamople and all particles has the same weight it will return an identical set, meaning we do not lose information. And lastly, it has a time complexity of $O(M\ log\ M)$, where M is the size of the sample set[1].

Even in a large population of particles, it may occur that no particles are in the vicinity of the correct state, as any stochastic algorithm may accedently discard particles near the correct pose. This is known as particle deprivation, and is the result of high variance in random sampling[1]. This mostly tend to happen if the set of particles is small, but the probability of this happening in a population of any given size is non-zero.

2.5 Implementation of the SLAM algorithm

The LEGO project already had an online-SLAM type setup, as it integrates data continuously into a global map while exploring. It was therefore of interest to solve the online-SLAM problem, such that most of the current system could be reused. There was also inbuilt features to handle grid-maps, and therefore the logical choice was to solve the online-SLAM problem for a grid-map.

The approach used in this thesis to solve the SLAM problem is called the Rao-Blackwellized Particle Filter (RBPF) for Grid-based SLAM. This is a way to simplify equation 2.11 by assuming the map estimate and the pose estimate is independent, as shown in equation 2.15, by using the Rao-Blackwell theorem.

$$p(x_{1:t}, m|z_{1:t}, u_{1:t-1}) = p(m|z_{1:t}, x_{1:t})p(x_{1:t}|z_{1:t}, u_{1:t-1}) \tag{2.15}$$

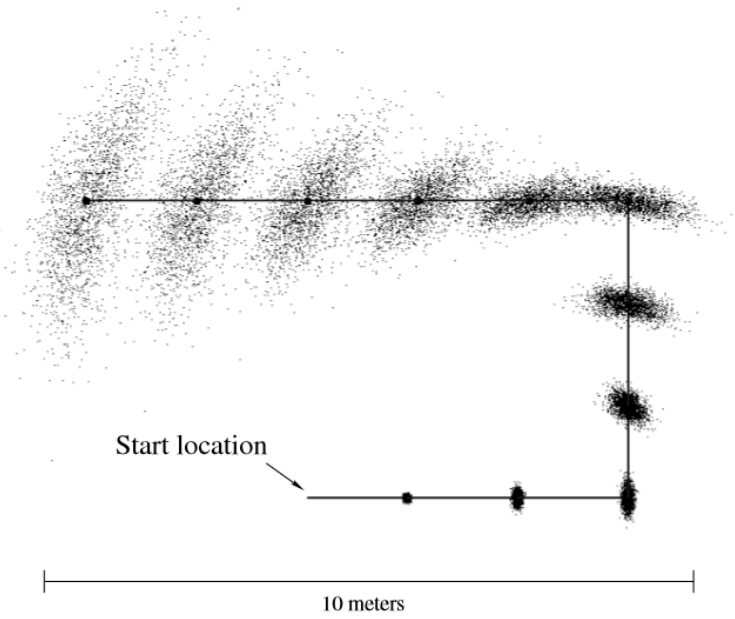

Figure 2.10: Graphical representation of how a particles spread out with time[1].

This factorization allows us to first estimate only the trajectory of the robot and then to compute the map given that trajectory. Since the map strongly depends on the pose estimate of the robot, this approach offers an efficient computation[29].

The base of this SLAM approach is a particle filter, where each particle contains a pose estimate (x,y,θ) as well as a estimate of the map, assembled as the robot acquires measurements. Without loss of generality, we can assume the robot to start mapping at position (0,0,0), however, it is also possible to initialize the particle filter at any other pose. While the robot moves, the particles also moves, according to the probabilistic motion model, which describes the uncertainty in the actual robot motion. Due to this uncertainty, the motion model contains a stochastic component, which results in the particles spreading out and generating slightly different trajectories, as can be seen in figure 2.10. Since the position estimate for each particles are slightly different, the maps will differ as well and each particle will have its unique map of the environment, as shown in figure 2.11.

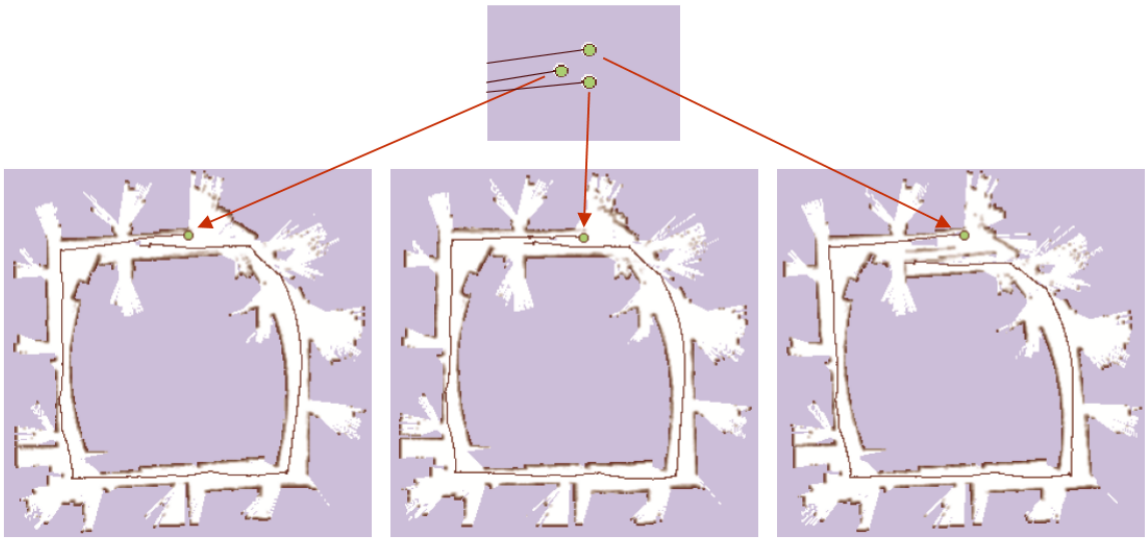

Figure 2.11: Graphical representation of how each particle have a unique representation of the map[1].

In this RBPF SLAM algorithm a map matching algorithm is used to weight the particles. Each particle will in addition to a global map, also have a local map, built from a sequence of recent observations. The global grid-map will not updated immediately, but stored in the local map and updated at discrete times. The trajectories of each particle is stored in a history vector for each particle, so when the particles returns to a known position it will compare (hence map match) the local map to the global map. The correlation between the maps gives a indication off how good the estimate is. An illustration is shown in figure 2.12, here a set of particles with different local and global maps are approaching a known destination. The local map of the particle to the right has a poor match to the global map, while the local map of the particle to the left fits nicely.

The beauty of this approach is that it is a purely algorithmic approach to solving the SLAM problem, we make no assumption on the hardware in question. This approach was proposed by Christof Schroeter and Horst-Michael Gross[3][27], they emphasis that the approach can be used independent of which sensors the robot is equipped with, and also allows for multi-sensor

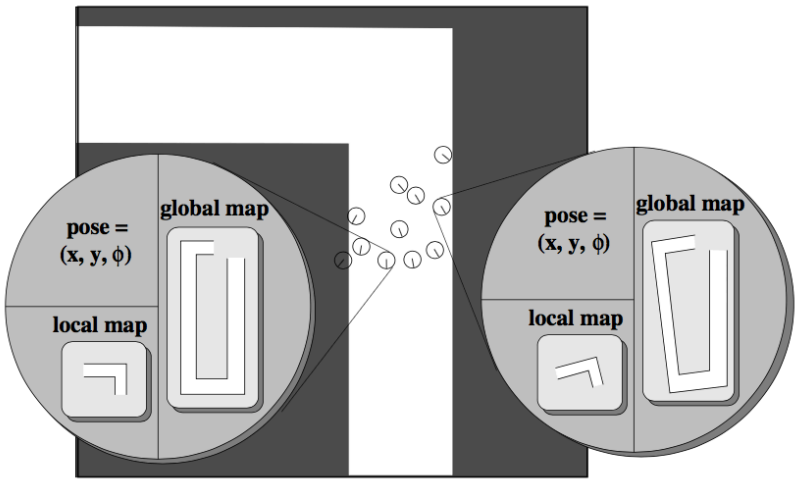

Figure 2.12: Map matching shortly before a loop closure[3].

fusion.

There are some parameters that can be adjusted in the particle filter, that may influence its performance. The GUI gives the opportunity to adjust these before each run, as shown in figure 2.13

The number of particles is an important parameter. When it comes to setting the number of particles, more is always better. This allows us to cover a larger set of pose hypothesis. The drawback is that it consumes more computational resources.

Another thing to tune is the size of local frame. A larger frame gives more data points to compare and therefore a might ensure a more accurate match, but will also come at a higher cost, as all comparison happens at discrete time, simultaneous for all particles. A very large frame will often not have a very large overlap anyway, as the robot mostly traverse unexplored areas and the sensor data is noisy.

The weighting acceptance factor adjusts how strict the resampling will be, a high factor will favour a few good particles, while a low factor will keep as

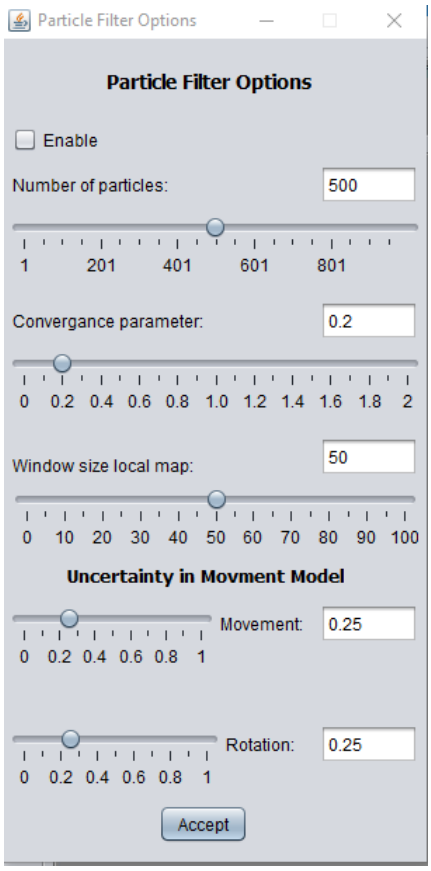

Figure 2.13: The particle filter options in the system GUI.

diverse a collection as possible. A good balance is often a good approach.

Lastly the uncertainty in the motion model can be adjusted. This sets the variance in a Gaussian motion model, and will effect how much the particles spread out. It is important not to make a to conservative assumption.

A user manual for how to apply the particle filter, as well as the map merger (explained later), along with the code documentation can be found in the appendix.

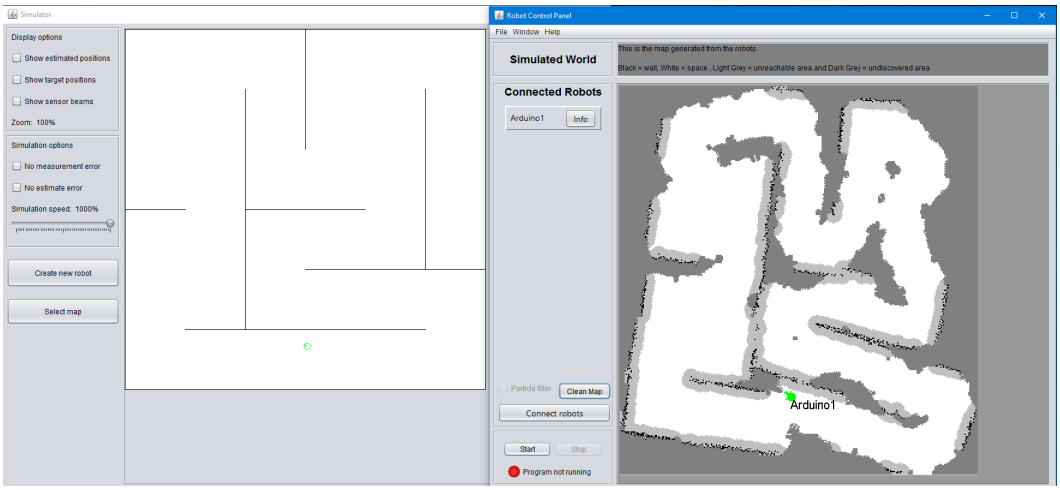

Figure 2.14: One robot simulated using the previously developed exploration algorithm stopped mid run.

2.6 Simulations

After the particle filter was implemented, testing was performed. Firstly a test without the particle filter was done for comparison, and then a the particle filter was applied to see if it had the desired effect.

2.6.1 Robot Without the Particle Filter

Figure 2.14 shows a simulation run using the previously developed exploration algorithm. Here the simulation is stopped relatively early in the run. The map is warped, but still almost usable. Figure 2.15 shows the same simulation some time later. The robot can make no further progress, and the map have actually degenerated to completely useless since the robot has made multiple observation of the same features, only believing that it was at different positions each time.

From these result, we can easily see that if we can remove the position uncertainty, we can reduce the problem to a mapping with known pose problem, a trivial problem.

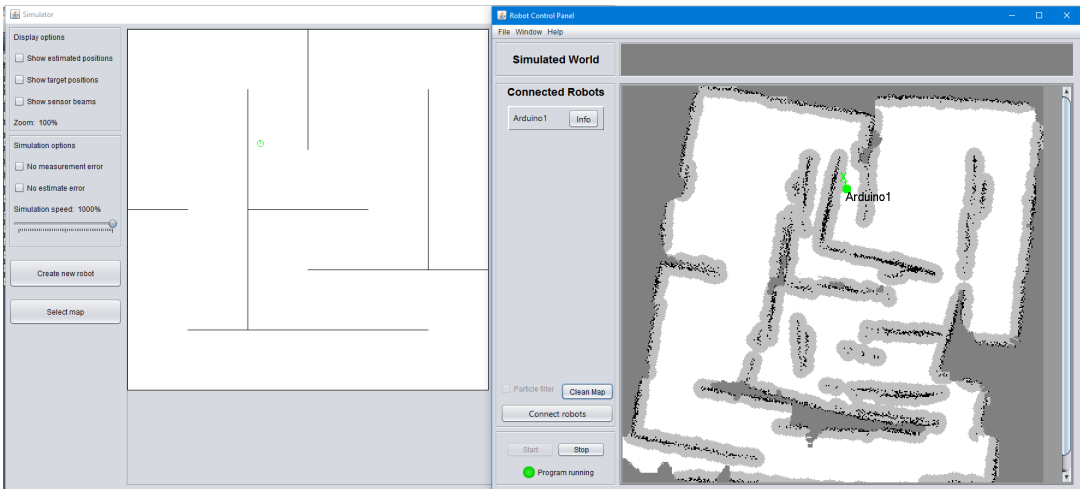

Figure 2.15: One robot simulated using the previously developed exploration algorithm after finished run.

2.6.2 Robot With the Particle Filter

The important part of the particle filter, is to see if it can actually give a god estimate of the robot pose. Since the Simulator gives both the estimated pose, and the actual pose of the robot in the GUI, a visual verification is easy to do.

We now enable the particle filter and test its accuracy. The map was chosen to be a smaller map version, so the particle filter actually re-observed some features and had a chance to close the loop. The larger map even had the possibility to move along without making any observations, as the corridor was wider than the sensor range. As can be seen in figure 2.16, the particle filter performs quite well. The red marker in simulator window shows where the robot believes it is, and the blue where it actual is. The particles (cyan dots) seems to have pinpointed exactly where the robot should be.

2.7 Field Testing

Since this thesis is mostly dedicated to applications on the server side, simulations are often enough to prove the concept of the implementations, but of

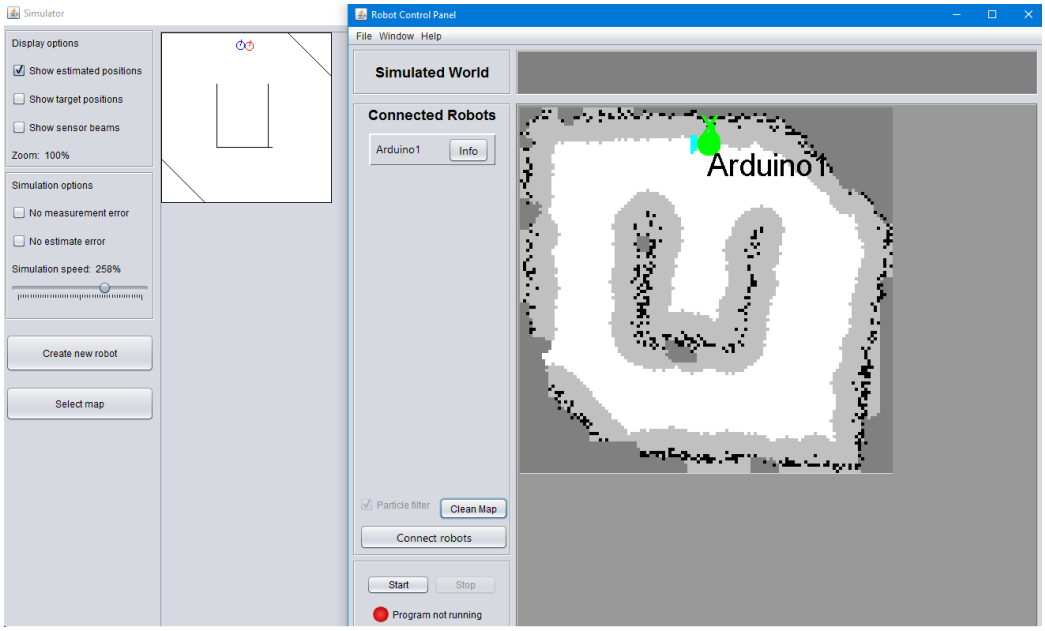

Figure 2.16: Particles (cyan dots) estimating the actual pose of the robot.

course, real life testing is also an important aspect of development. For this purpose a test labyrinth, as shown in figure 2.18, was prepared. The Arduino robot was used during the testing. Unfortunately the Arduino robot proved to have some issues, and would some times disconnect, making longer runs infeasible.

Figure 2.17 shows the results from a test run with the particle filter. The original map is shown to the left, and the cyan dots represents particles. Neither of the maps are perfect, as sensor noise is still present. The map from the best scoring particles is shown to the right. When comparing the two maps, it can easily be verified that the particle filter appears less "noisy". The original map seems to have observed the left and top walls from two different poses, causing the map to have what appears as a an inner and outer wall. There is also indication of warping of the right wall, probably indicating that the robot drifted to the side when it believed it was driving straight. On the other hand, the particle map has none of these issues.

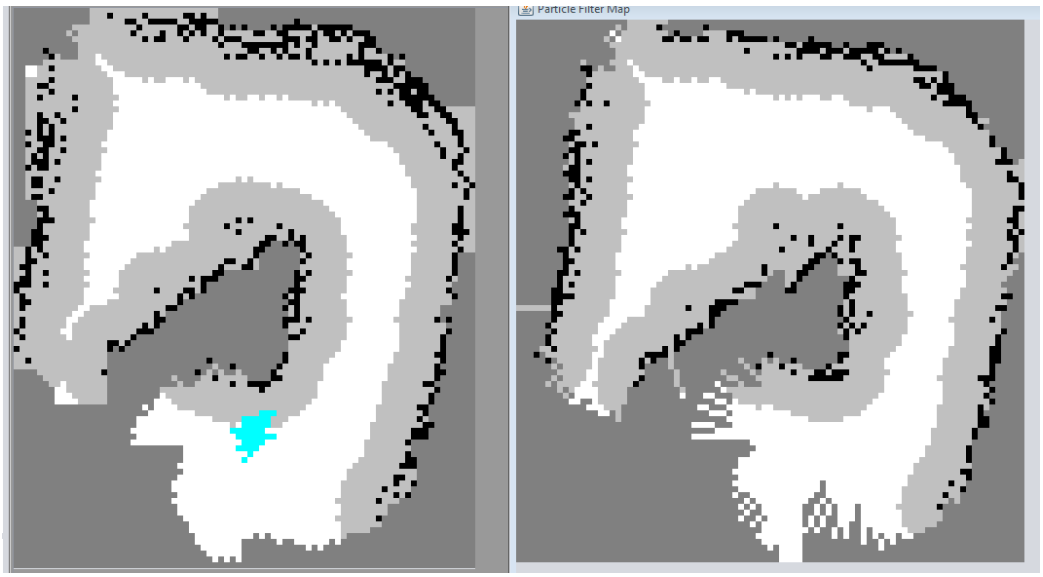

Figure 2.17: The real world test labyrinth. The original map to the left, and the map from the particle filter to the right.

Another thing to draw from this, is that the server and particle filter also functions in the real world, and that the maps produced are not that far off what the simulator would produce.

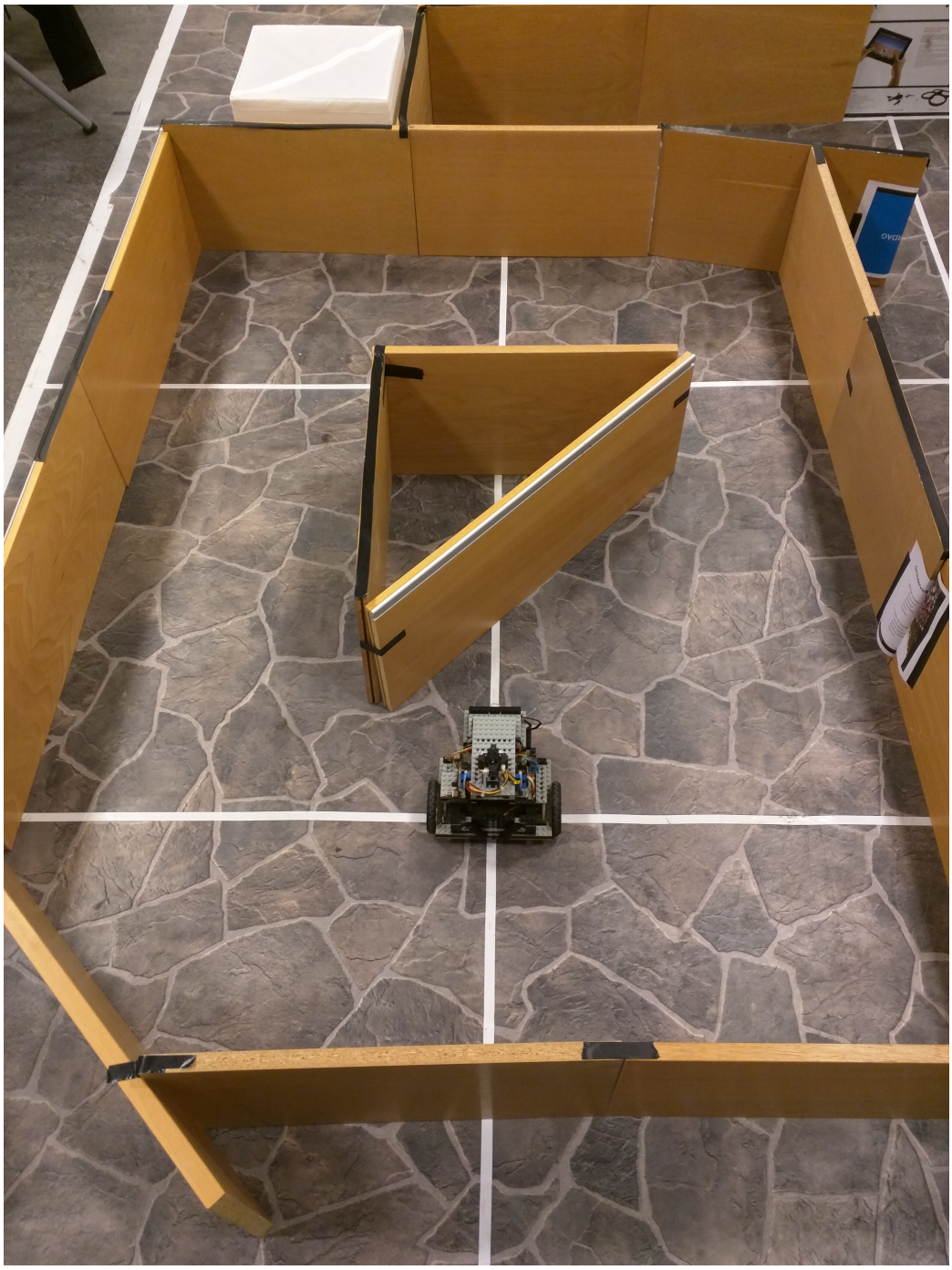

Figure 2.18: The test labyrinth made for field testing the system.

Chapter 3

Map Merging

In the case of multiple collaborating robots, mapping can in theory be done at a much quicker pace. In an ideal case the with N robots, the mapping will be almost N times faster. This is however higly unlikely in the real world, as there is no way of perfectly divideing up the resources, unless you already have a perfect map. The major issue with multi robot exploration is to merge together each robots map into one global map. This can be done in several ways, either in form of start conditions, such as the relative positions of the robots[30], or the identification of the position of a robot within the existing map of another one[31], or by using an explicit rendezvous strategy[32]. All of these approaches requires a way to merge the data from each robot into a coherent map.

The current approach used in the Lego project requires the relative positions of all the robots to be known. This is a major limitation, and also a potential way to introduce errors into the system if one or more of the robots starting pose is slightly different than assumed. The major advantage of this approach is that it is quite simple, all robots can directly add its data to a global map and no advanced merging is needed. But to be able to use other exploration algorithms, or to improve the map estimate when one or more robots are initialize in the wrong position or the robots drift in separate directions while exploring, a map merging algorithm is needed.

3.1 Map Merging Strategies

Merging together map data is closely linked to a problem in computer vision known as image registration. In image registration different sets of data are transformed into one coordinate system, such as matching two or more images, or creating a landscape image from separate but overlapping images, also known as image stitching. Image registration has uses in such fields as computer vision, medical imaging and automatic target recognition[33].

There are several approaches to the image registration problem. A popular approach is to use a transformation model, such as a linear transformation, which include rotation, scaling, translation, and other affine transforms. The goal is then to fit one image on top of the other[33].

Other approaches uses frequency domain methods, such phase correlation which esitmates the relative translative offset between two similar images by applying the Fast Fourier Transform (FFT)[34].

The field of computer vision is large and complex. The interested reader is recommended to consult textbooks such as Szeliskis "Computer Vision: Algorithms and Applications" for an in depth introduction in the field[35].

3.2 Implementation of the Map Merger

Given that the robots does not know the other robots relative position, merging of several local map into a global map becomes necessary. One approach proposed by Birk and Carpin tries to fit two maps, m and m' using a stochastic search algorithm to transform m' by rotations and translations to find the maximum overlap between m and m', using a heuristic similarity metric to guide the search algorithm toward optimal solutions[36]. This non-convex optimization problem can be solved using Carpin's Gaussian Random Walk[37], which is quite similar to simulated annealing. Normal hill-climbing algorithm is not advised in this kind of optimization, as the worst and best solutions may lie in neighbouring states[36]. The problem of merging maps from more than two robots can easily be handled by merging two and two maps at a time.

We first need a cost function to optimize, this function needs to give gradient information to the solver, so that improvements can be made in the estimate. A simple example of a cost function would be: number of agreeing cells - number of disagreeing cells. This gives quantitative data on how well the image fits, unfortunately it provides no gradient information. Another approach is using the image similarity ψ as described in equations 3.1 to 3.2.

$$\psi(m_1, m_2) = \sum_{c \in C} d(m_1, m_2, c) + d(m_2, m_1, c) \tag{3.1}$$

$$d(m_1, m_2, c) = \frac{\sum_{m_1[p_1]=c} min\{md(p_1, p_2)\} | m_2[p_2] = c}{\#_c(m_1)} \tag{3.2}$$

Here C is the set of values, 1 for occupied and -1 for free in this case. Undiscovered cells gives no helpful information, and is therefore not included in the metric. The function $md(p_1, p_2) = |x_1 - x_2| + |y_1 - _2|$ and is also known as the Manhattan distance between two points. $\#_c(m_1)$ stands for the number of cells in m_1 with value c etc.. The strong point of the image similarity ψ is that it can be computed relatively efficiently, and it gives gradient information guiding the search algorithm closer to a solution at each iteration[36].

Image similarity is a good metric for fitting perfectly overlapping images, but when working on partially built maps, where only smaller regions may overlap, some adjustments is needed. The algorithm will potentially over fit as ψ only tries to minimize the Manhattan distance between points. The new cost function $\Delta(m_1, m_2)$ is therefore introduced in equation 3.5, with a term that penalizes over fitting.

$$\Delta(m_1, m_2) = \psi(m_1, m_2) + c_{lock} \cdot (dis(m_1, m_2) - agr(m_1, m_2)) \tag{3.3}$$

$$dis(m_1, m_2) = \#\{p = (x, y) | m_1[p] = m_2[p] \in C\} \tag{3.4}$$

$$agr(m_1, m_2) = \#\{p = (x, y) | m_1[p] \neq m_2[p] \in C\} \tag{3.5}$$

The functions $agr()$ and $dis()$ returns the number of agreeing (both occupied or both free) and disagreeing points in the map. Note that only the information from map parts that are aligned with each other in the current transformation step is used, if one of the points are unexplored, they are not accounted for in the $agr()$ or $dis()$ functions. The parameter c_{lock} is a tuneable parameter that adjusts the amount of necessary overlap between the two maps for a successful merger. This parameter is quite important and

needs to be adjusted for each map. Two perfectly identical maps which only need rotation and translation to perfectly overlap can have the c_{lock} set to zero, while two maps who barely overlaps needs a larger value for c_{lock}.

All stochastic algorithms has the potential of failing, so we need a way of recognizing failure. This can simply be done by introducing the acceptance indicator $ai(m_1, m_2)$ as shown in equation 3.6.

$$ai(m_1, m_2) = 1 - \frac{agr(m_1, m_2)}{agr(m_1, m_2) - dis(m_1, m_2)} \tag{3.6}$$

The acceptance indicator gives a percentage overlap of the images, and only if it is very close to 1.0 is there a good overlap between a region of m_1 and a region of m_2, other solutions should be rejected. Due to noise in the sensor data, a slightly lower value than 1.0 should be acceptable.

3.3 Simulations

To check the capabilities of the map merging algorithm, some simulations were run. The first simulation, shown in in figure 3.1 tested if the maps could be merged given that the wrong assumptions of starting position was given. As can be seen to the left in figure 3.1, the outline of two identical maps are clearly visible, but the actual map makes little sense. Arduino2 also cant explore the interior of the map as it believes a wall (non existent, but discovered by the other robot) is blocking its path. To the right in figure 3.1, the maps are successfully merged. It can be said that this is an extreme case of misplacing the robot, often a few centimetres and a few degrees would be the extent of miscalculation, but the proof of concept for the map merger is clearly shown.

The next test was to show that two barely overlapping maps could be merged together. Figure 3.2 shows the two partial maps from two different robots running particle filters to the right, and the successfully merged map to the right. The positioning of the robots are wrong in the merged map, as the map adds some buffer around the individual maps before merging them. This simulations proves that even partial maps can be merged together using the map merger algorithm.

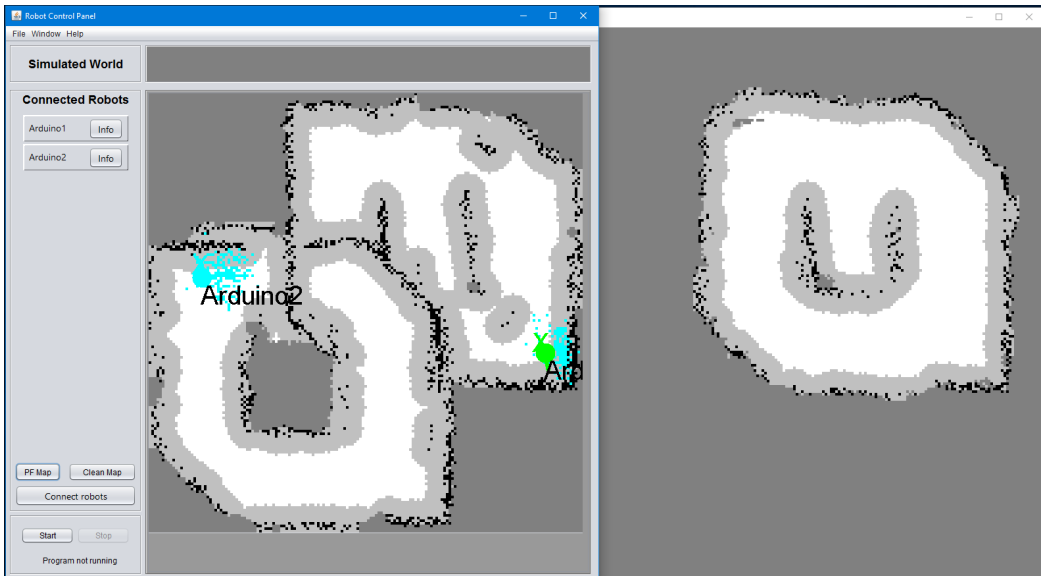

Figure 3.1: Simulations of robots initialized in wrong pose. The original map to the left, and the merged map to the right. The c_{lock} was set to 7.5.

3.4 Field Testing

The simulations shows that the map merging algorithm works, but the true test is see if it can merge maps in a real world setting. Figure 3.3 shows the testing area for the map merger, the area was expanded from the previous test (see figure 2.18) to give more a more realistic and challenging test.

Figure 3.4 show the map merging of two real world maps acquired from the Arduino and NXT robot running particle filters. To the left, the two partial maps from the robots particle filters are shown, and to the right is the merged map. Again, the Arduino had some issues so the length of the test run was limited. Both maps are quite noisy, especially the NXT. In spite of that, the algorithm managed to correctly piece the two maps together even with minimal overlap. The area mapped is the rightmost part in figure 2.18. The triangular wall can almost be identified in the middle of the merged map.

When merging the maps The c_{lock} value was set to 100 in this attempt. The the algorithm did produce some less satisfactory attempts at first, but

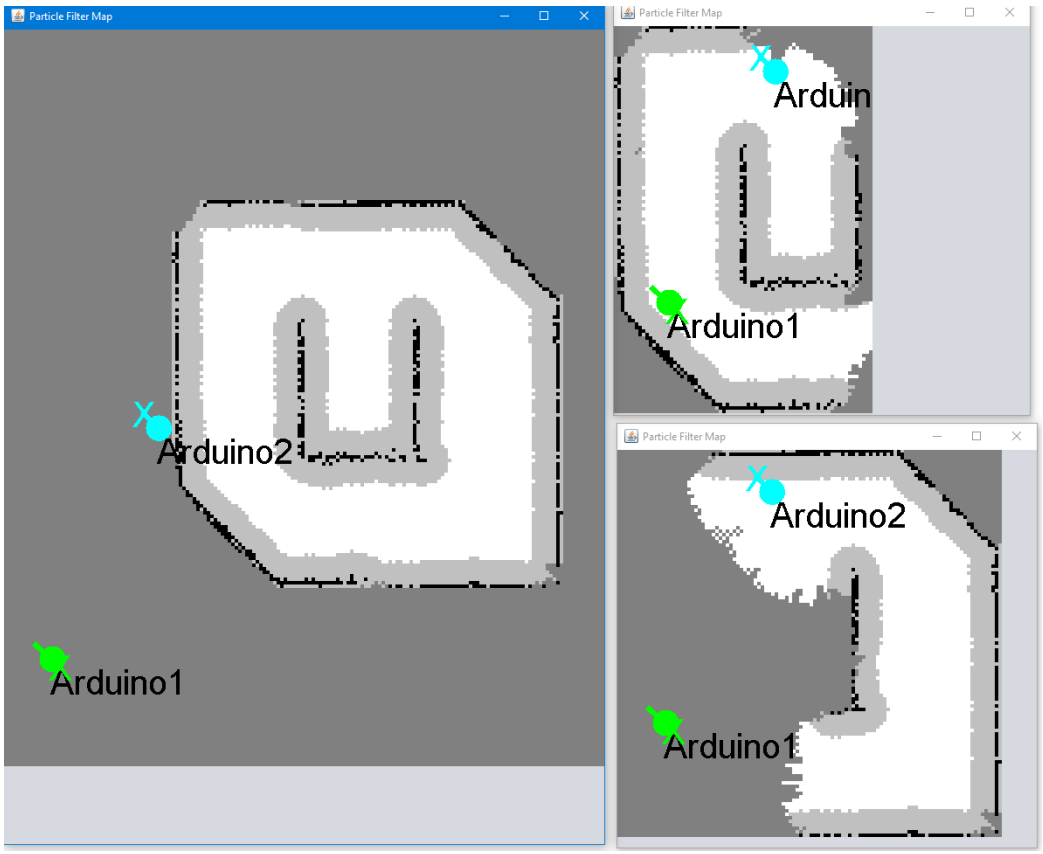

Figure 3.2: This figure shows how the two partially overlapping maps can be combined using the map merging algorithm. The c_{lock} value was set to 100.

by adjusting the values of c_{lock} to account for the small overlap of the maps, a good estimate emerged.

Figure 3.3: The expanded test labyrinth made for field testing the system.

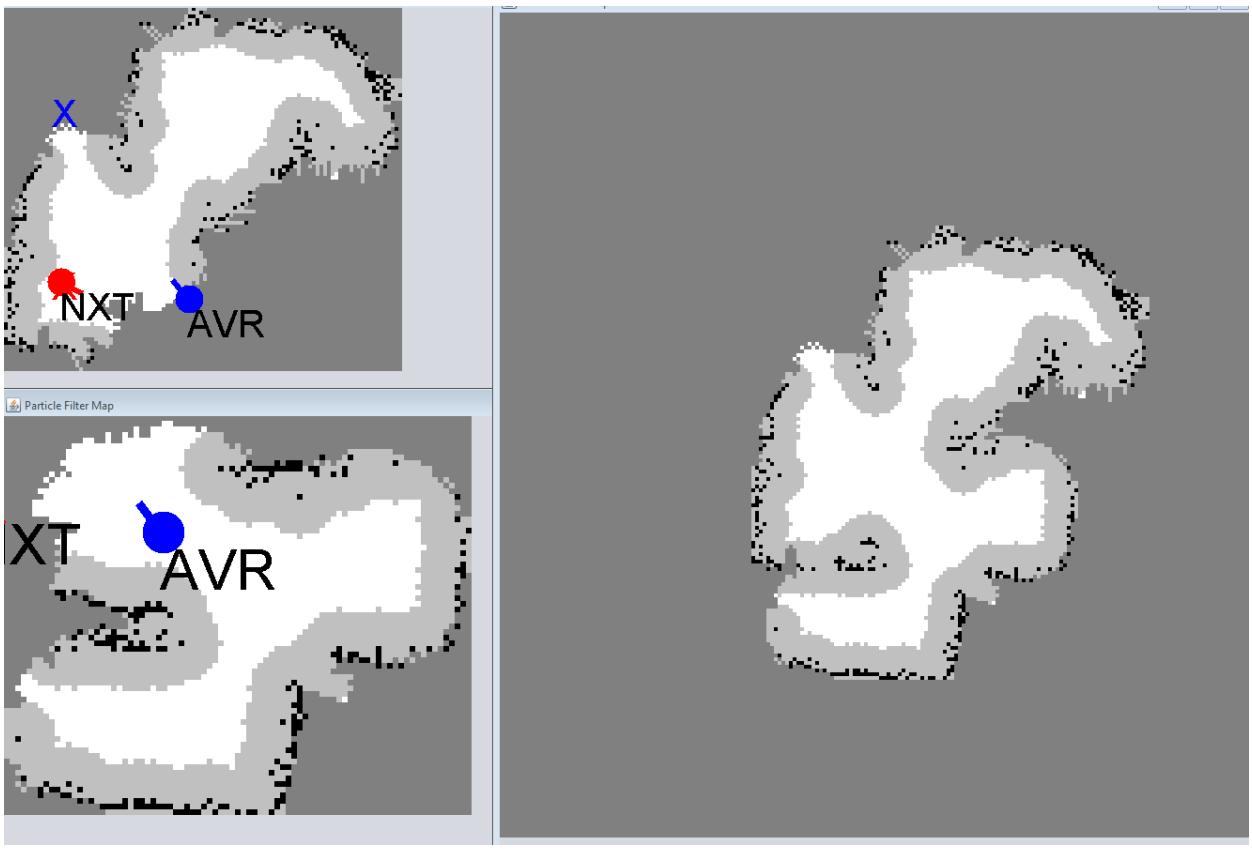

Figure 3.4: Map merging the real world maps.

Chapter 4

Multi-Robot Exploration

When the SLAM problem is solved, we are still left with the task of exploring the environment. We now trespass onto the field of AI, and unveil a completely different, but still extremely important aspect of robotics: exploration. The goal is to fully explore the environment as efficiently and accurate as possible.

Important questions arises when planing with collaborating agents, shall the agents seek out each other to determine the their relative locations, or spread out to cover more ground faster? The robots themselves introduce a dynamic component in an otherwise static environment, how is this handled? Is one unit more accurate than another, in that case, how to allocate the agents to minimize uncertainty? How are collisions handled? These and numerous other questions emerges. Unfortunately all of these questions will not be answered in this thesis. Time ran short, and a proper implementation of the exploration algorithm was never finalized. It is however the last piece of the puzzle, and therefore still has a place here, if only as a theoretical exercise.

4.1 Multi-Robot Exploration Strategies

The most popular of the existing approaches to coordinated multi-robot exploration assume that all robots know their locations in a shared frame of reference. Using the same frame to coordinate exploration frontiers from a common map and assigning robots to the frontier locations[30].

Another approach, makes all robots explore independently of each other. They then depend on coincidentally detecting another robot to determine its position and then combines their maps. This approach scales well, and probably performs better with more robots, but it can result in inefficient exploration. It will take arbitrarily long until a robot coincidentally detects another robot. For instance, if one robot follows the same path as another, both robots might explore the complete map without ever detecting each other[32].

Some other approaches try to connect relative locations between pairs of robots by estimating one robots location in another robots map. This assumes that one robot starts in the map already built by the other robot and can therefore easily fail[31].

4.2 Limitations to the Current System

The current setup uses an A* (frontier) based exploration algorithm. The algorithm assumes that the starting pose of each robot is known, and chooses to ignore all measurements in close proximity of other agents. The A* algorithm and it predecessors back to Dijkstra's algorithm, has been used heavily in the field of AI for finding the shortest path in graphs. A* is an informed search algorithm (best-first search), meaning that it searches among all possible paths to the goal for the one that incurs the smallest cost. In this case, the frontier (closest unexplored region) is the goal, and the agents expands the frontier as they search, and ultimately explores the whole region. The algorithm is therefore complete and will always find a solution if one exists, but does not delegate resources optimally and requires quite a bit of computational power per unit, especially when the number of unexplored nodes gets smaller and the number of explored nodes gets larger. The algorithm also requires all poses to be known at the beginning, or that the agents have global positions such as with GPS.

4.3 Improving the Current System

The plan for this thesis was to implement an algorithm using a multiple objective utility function that take into account distance/effort/time (these are

in essence equal in this case) and the amount of new information available at that goal state[38]. Alas, other aspects of the thesis was prioritized and time ran short. This part was therefore never implemented.

Ideally the map merger would work reliably in real-time and output a global map estimate. In this case, the frontier based approach would actually still work, even without knowing the relative poses of the robots. But the current A* approach still delegates resources inefficiently, something similar to the multiple objective utility function previously mentioned should be implemented. The robots would then use an utility function to optimize the exploration, and optimal use of resources would only depend on how we define the utility function.

The utility function could potentially be constructed with more complex constraints. For example if one robot was low on battery, it would prioritize to explore closer to the charging station. Or if one area was to narrow for one robot to fit, a smaller robot should be prioritized. Or maybe one robot is faster than the other ones, then it could prioritize exploring furthest away, and leave the rest for its slower companions. The flexibility a properly constructed cost function gives, is worth exploring.

Chapter 5

Discussion

The particle filter and map merging algorithm works quite well when tested. Sometimes the particle filter is not a much better estimate than the initial guess of the robot, but it is very seldom worse. The map merger may need a few iterations to get the maps stitched together, especially when there is only a partial overlap, and the c_{lock} value must be experimented with on a case to case basis, but overall it works surprisingly well.

There are of course some issues with the whole approach, some stemming from the fact that some crucial pieces are missing. For example, since the exploration algorithm in the current setup assumes the robots are mapping on a global map and uses a frontier based approach to assign tasks, some issues occur when applying the particle filter and map merging algorithm. If the robots are initialized wrongly or if they drift to much from their estimates while exploring, the current system have a tendency to make a rather bad map. Often robots will bar each other inside, since they believe a wall is in a different places than it actually is. This can cause a robot standing in an open space to believe it cant reach any other place. The exploration algorithm should therefore be re-evaluated.

Another issue is that noise in the IR sensors data will deteriorate the effect of the particle filter, since if the outcome of two scans from the same position is unequal, the overlap from map matching will not be optimal. It is however often assumed that the range and bearing sensors are more accurate than odometer information, and therefore it is valid to correct the pose based on observations.

The noise in the IR sensor also causes trouble for the map merging algorithm. In its basic form ($c_{lock} = 0$), the algorithm tries to get all points as close as possible to any other matching points, ideally having two perfectly overlapping images. The algorithm minimizes the cost function as much as possible and returns the local minima, which hopefully also is the global minima (simulated annealing gives no such guarantee however). Noisy data will create more local minima, as there are now several ways to interpret the data. There is only one way to overlap a perfect line with another line (two, if directions are important), but there are potentially many equally good ways to minimize the distance between two noisy point clouds of a line.

An issue connected to the exploration algorithm and the particle filter, is that there is no guarantee that the robot returns to a known position. The drawback of this is the particle filter has no way of correcting its estimate. Appropriate measures in the exploration algorithm should be taken to ensure optimal behaviour.

The robots themselves proved a challenge when it came to testing. The Arduino had a tendency to disconnect from the server at (seemingly) random intervals. This made the the scale of the real world test we could do limited, since the server is constructed such that it must be relaunched to reconnect to the robots. This gives two important insights: field testing and simulations are vastly different, and no mater how clever your algorithm is, if the hardware does not function properly it is useless.

The server now assumes that the basic properties of all the robots are the same, this is a good assumption when working with simulations, as all the simulated robots are the same. In practice however the different robots have their own behaviour, some move faster, some are more accurate, and some are bigger and need more space when cutting corners etc.. The transition from simulation to reality can sometimes be harsh.

Memory usage quickly becomes a limitation in this system. In an ideal world, where memory is infinite, the particle filter would have infinite particles and all possible states of the system would be compared to the actual state. This would give an ideal pose estimate. In the real world, we must limit the amount of particles, and therefore we run the risk of suboptimal solutions.

Some techniques, such as resampling, can be used to improve the selection of particles, but everything is a trade-off. Resampling runs the risk of particle deprecation and is also in it self a memory and CPU consuming process.

It may be argued that the same algorithm used, only with a scan-matching instead of a map-matching algorithm would be a better option. In fact, the only thing that would need to be changed in the source code would be a call to a scan-matching algorithm, taking the same input, and (depending on how one choose to use the result) might give the same output. This would of course be more memory intensive, as the scan-matcher uses a more exhaustive approach, but can be solved quite efficiently using a Java linear algebra library such as JAMA. Even running these calculations in C/C++ or similar may be an approach worth considering, as they are significantly faster languages than Java using a Just in Time (JIT) compiler and running on a virtual machine (JVM).

In one of the earlier implementation, the GridMap class implemented originally in the server was used for each particle. The performance was horrific. It was clearly not designed to have more than one instance running at a time. Even running 30-40 particles would give most computers trouble after running for a few minute. After identifying the problem, a more efficient, but less intuitive, hashmap was used to hold map locations. This simple change now allowed over 1000 particles to run seamlessly for even larger maps. This shows how smart choices of data types matters, even for modern computers.

As of now, all robot plots on the same grid, the problem is that all data will be overwritten by any robot traversing the same path. If the robot is a tiny bit off, the whole section of map will be shifted, even if it was a better estimate of the real world, since the robots and server don't know what the real world really looks like. This could be avoided by making all data points permanent. Ideally, an estimate of the robots probability of being in the right position should be maintained, and the robot that is most "sure" of being at the correct pose gets precedence. This could be taken as far as denying robots from adding data to the common grid, until it is relatively certain of its position.

Chapter 6

Other Implementations

A series of other functionality has been added to the server side during the work of this project, they do not relate directly with the theme of this thesis, but will be documented here for posterity.

6.1 The CleanMap Function

There were an interest of transforming the data point clouds to nice lines in the final map. A "CleanMap" function was therefore added to make this transformation. No good example algorithms was found in any papers, so a customized variant was developed by trial and error. This function evaluate each 5x5 grid in the map grid and finds the mass center. The points are then connected together if they are within proximity of each other. An example is shown in figure 6.1. As can be seen there are some strange artefacts, especially in corner areas. The algorithm is confused as to which point is the neighbouring point, and connects both. If there is insufficient data points, gaps in the lines will appear, as the algorithm thinks sparse points are outliers. Figure 6.1 was drawn using the simulator and no measurement and position estimate error, in this case the algorithm does little else than remove superfluous data points and connects the dots. In the case of a poor map, the algorithm will naturally preform worse, as illustrated in figure 6.2. As can be seen, the algorithm tries to cope with thick cluster of points and make some fascinating geometrical figures. The figures was an accident, but they look nice, and was therefore kept instead of solid black walls. To be fair, the algorithm is not an oracle, and relies on the data at hand, a perfect

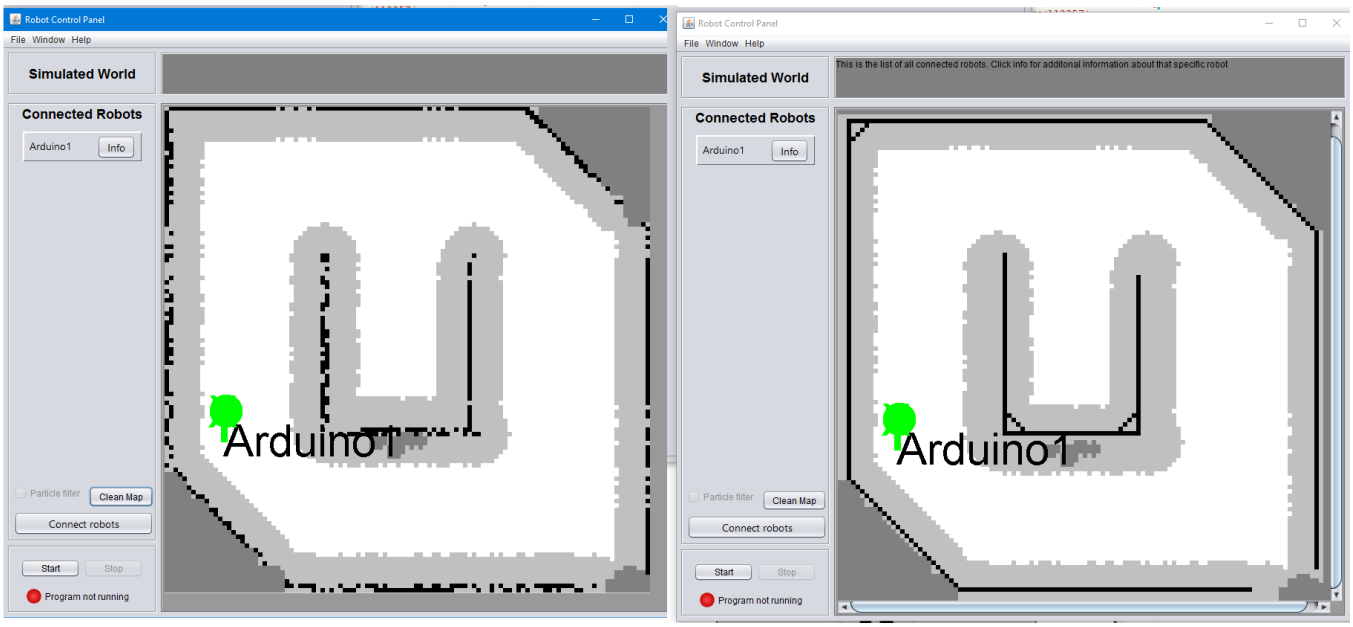

Figure 6.1: The CleanMap algorithm on a good map. Left is before, and right is after the CleanMap function is run.

result from poor data is a bit much to ask for.

6.2 Integration of Drone Data

A drone was added to the project this year, and therefore needed a way to integrate its data into the map. The drone has an birds eye view of the labyrinth and extracts walls in the form of lines. To accommodate this new data transfer, the communication protocol was expanded. The format is the same as all other robots, except the 4 data fields are start of line (x and y position) and end of the line:

$$\{U : X, Y, heading, startX, startY, endX, endY\}$$

The full communication protocol can be found in the thesis by Andersen and Rødseth[21].

At this point the data from the drone is indiscriminatingly added to the

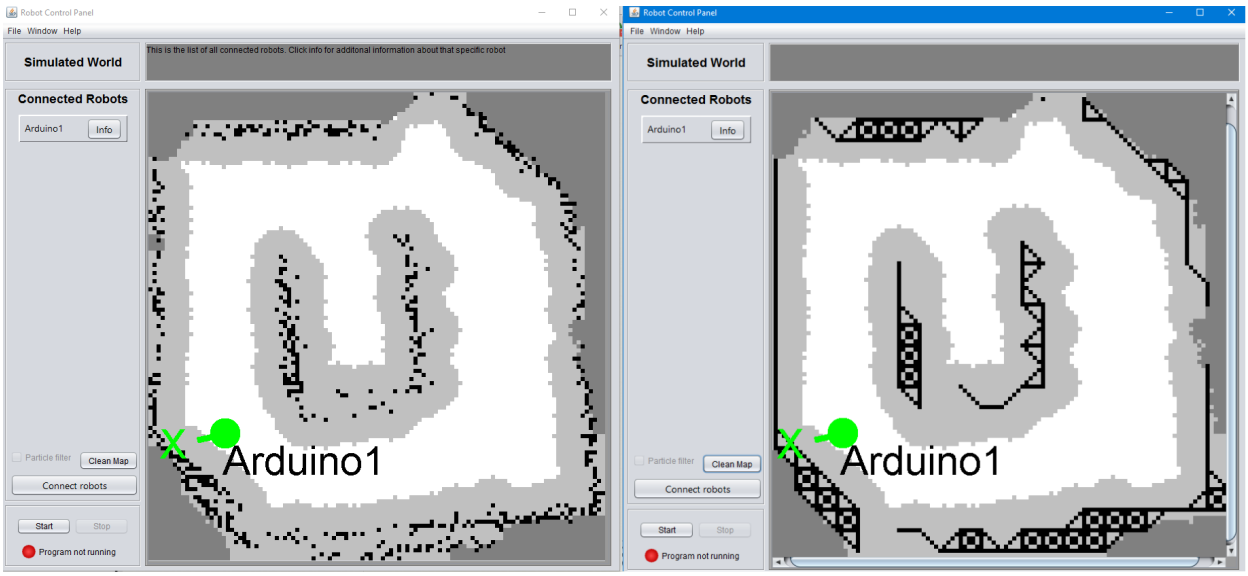

Figure 6.2: The CleanMap algorithm on a poor map. Left is before, and right is after the CleanMap function is run

map as if it were data from any other robots. The only exception is that it is processed to make a line in the map using Bresenham algorithm first[39]. A simple example is shown in figure 6.3.

The grand plan was to integrate the data collected from the drone in some way that improves the map estimate, such as using scan matching to correct the already explored map. The potential for the drone is limitless, better mapping, better cooperation of ground vehicles etc.. Unfortunately the drone project has not yet left the ground (figuratively and literally) and little information is therefore known of the characteristics of the data collected. Is the drone data more or less trustworthy than the ground based vehicles? Can it track moving objects and process information real-time to advise the server? As such, only basic support for a drone is added on the server at the current time.

A drone simulator has been added to the simulator part of the server. This drone can be moved manually (as no motion model is known) and will detect the edges in a given frame. This seems to be working well. Note that no error

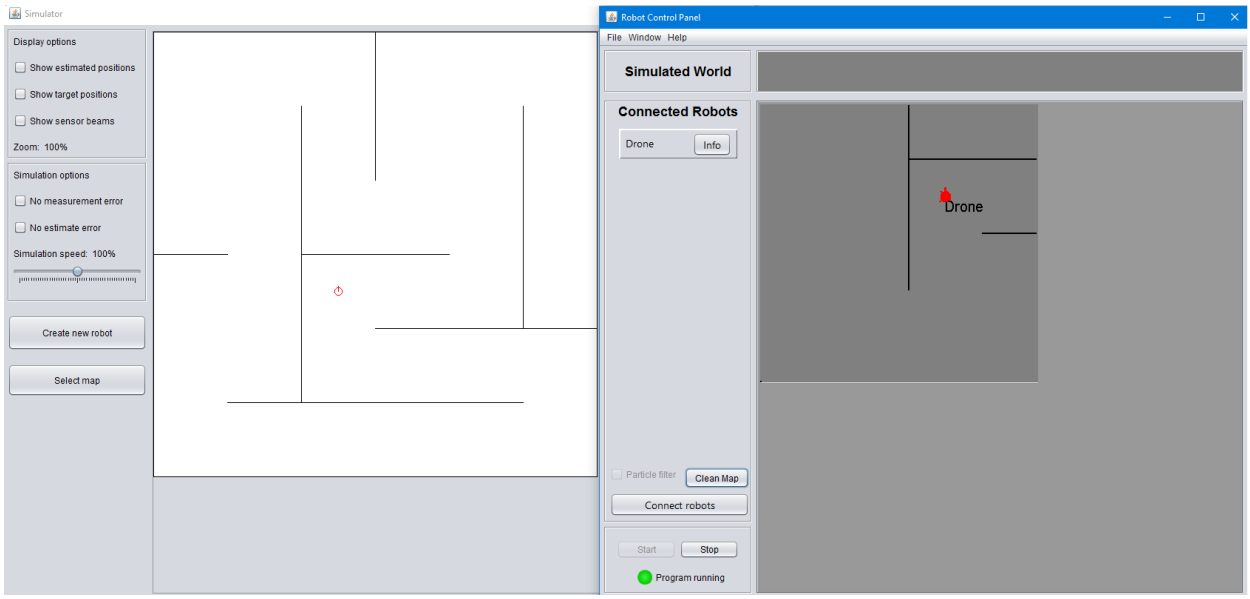

Figure 6.3: The Drone collecting data from a 250 x 250 cm frame of the map.

estimate is included in the simulator, since even the error characteristics of the drone data is unknown.

6.3 Debug Logger Functionality

The robots can now send messages in plain text to the server using the message prefix "D" (for Debug), and the communication protocol: $\{D : Text\}$. The appended text will be added to a "LOG.txt" file in the application folder. This functionality was requested as some of the developers needed a way for the robots to communicate more detailed information to the server while debugging.

6.4 Simulator Changes

The robot selection is made easier, you can now chose what type of robot you want in the simulator from a drop down menu. As functionality for other units such as the drone was added to the simulator, the need for selecting a

specific unit became apparent. The old simulator added robots from a list in a preselected order. This was fine if you wanted to test unit number 1 of the list, but quite tedious if you wanted to test unit number 9. You would then have to add 8 other robots first and then place them outside the borders of the map, and possibly risk a null pointer exception if you initialized them. The addition of a drop down menu saves everyone some time and frustration.

6.5 General Changes to Server

I started out with the mindset of being as little intrusive in the previous implemented code as possible. The ideal was just adding a module and calling it only once or twice in the main loop of the server. This would give a clean and object oriented interface.

In reality i ended up adding a bunch of functions in many different classes. I needed copy constructors, functionality to render new items in the GUI, fixed bugs in the old code, and added template classes for the Drone etc.. The code already bears the mark of being programmed by different people and no clear API. There even exist two different (three if you include the Pose class) ways to express position, one as a class Position(), and one as an int array. Different functions (probably developed by different people) requires one or the other. There are several things that could be improved in the code, and having more people work on the same project will probably bloat the code until it is impossible for anyone to wrap their head around it. The code is already a mouthful at first read, and I have personally made some unintentionally strange side effects and used days to figure out what really happens. I have an eerie feeling that in some years, someone will scrap the whole thing and recreate it in their own image (so to speak), and possibly do the same mistakes again.

Chapter 7

Further Work

7.1 Efficiency

The whole algorithm implemented in this thesis hinges on the number of particles the server is using. Using only one particle is probably useless, but using a million will probably cover most of the possible poses. A million particle will now unfortunately eat all the memory for even a decent computer. Finding clever solutions to save memory can greatly improve the usefulness of the particle filter.

7.2 A Better Exploration Algorithm

The grand plan was to make a better exploration algorithm, but time ran short. Finding an algorithm that can utilize the fact that there are more robots in use, and efficiently distribute task would be a great improvement on the system.

The way the particle filter is now implemented draws great benefit of traversing already discovered places. A* is not the ideal algorithm for this approach, as it will not necessarily retrace its step. A exploration algorithm that "closes the loop" will be more beneficial for this approach.

7.3 Integration of Drone Data

As mentioned earlier, the drone data is only added to the map, as if it were any other robot. But there should be a way of exploiting the characteristic of the drone (when they are known) to improve certain aspects of the map. If we, for instance, know the placement of a line might be off, but the length of the line is quite certain, we can use this to augment the map, or even correct the robots pose based on the knowledge. Integrating this aspect into the server will be an interesting and challenging task.

Bibliography

[1] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[2] Cyrill Stachniss. Lecture notes in robot mapping, 2015.

[3] Horst-Michael Gross Christof Schroeter. A sensor-independent approach to rbpf slam - map match slam applied to visual mapping. *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2008.

[4] Håkon Skjelten. Fjernnavigasjon av lego-robot, 2004. Project report, NTNU, Dept. of Engineering Cybernetics.

[5] Sveinung Helgeland. Autonomous legorobot. Master's thesis, Dept. of Engineering Cybernetics, NTNU, 2005.

[6] Bjørn Syvertsen. Autonom legorobot. Master's thesis, Dept. of Engineering Cybernetics, NTNU, 2006.

[7] Paul Sverre Kallevik. Improvement of hardware on legorobot, 2007. Project report, NTNU, Dept. of Engineering Cybernetics.

[8] Johannes Schrimpf. Improvement of the real-time-characteristics of a legorobot, 2007. Project report, NTNU, Dept. of Engineering Cybernetics.

[9] Jon Martin Bakken. Bygge og programmere ny legorobot, 2008. Project report, NTNU, Dept. of Engineering Cybernetics.

[10] Erik Næss. Legorobot - forbedring av eksisterende system, 2008. Project report, NTNU, Dept. of Engineering Cybernetics.

[11] Trond Magnussen. Fjernstyring av legorobot. Master's thesis, Dept. of Engineering Cybernetics, NTNU, 2008.

[12] Andreas Haugedal. Forbedring av de autonome egenskapene til en legorobot, 2008. Project report, NTNU, Dept. of Engineering Cybernetics.

[13] Jannicke Selnes Tusvik. Fjernstyring av legorobot, 2009. Project report, NTNU, Dept. of Engineering Cybernetics.

[14] Morten Andr Kristiansen. Fjernstyring av lego-robot, 2009. Project report, NTNU, Dept. of Engineering Cybernetics.

[15] Jens Kristian Tøraasen. Kartlegging ved hjelp av robot og kamerasensor, 2009. Project report, NTNU, Dept. of Engineering Cybernetics.

[16] Sigurd Hannaas. Samarbeidende legoroboter. Master's thesis, NTNU, 2011.

[17] Trond Kåre Homestad. Fjernstyring av legorobot. Master's thesis, Dept. of Engineering Cybernetics, NTNU, 2013.

[18] Ø yvind Ulvin Halvorsen. Collaborating robots. Master's thesis, Dept. of Engineering Cybernetics, NTNU, 2014.

[19] Erlend Ese. Fjernstyring av legorobot, 2015. Project report, NTNU, Dept. of Engineering Cybernetics.

[20] Erlend Ese. Sanntidsprogrammering på samarbeidande mobil-robotar. Master's thesis, Dept. of Engineering Cybernetics, NTNU, 2016.

[21] Thor Eivind Svergja Andersen and Mats Gjerset Rødseth. System for self-navigating autonomous robots. Master's thesis, Dept. of Engineering Cybernetics, NTNU, 2016.

[22] Eirik Thon. Mapping and navigation for collaborating mobile robots. Master's thesis, Dept. of Engineering Cybernetics, NTNU, 2016.

[23] Sebastian Thrun. *Exploring Artificial Intelligence in the New Millennium*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[24] Olav Egeland and Tommy Gravdhal. *Modeling and Simulations for Automatic Control*. Tapir Trykkeri, Throndheim, Norway, 2002.

[25] Olga Sorkine-Hornung and Michael Rabinovich. Least-squares rigid motion using svd. Technical report, Department of Computer Science, ETH Zurich, 2017.

[26] Zhengyou Zhang. Iterative point matching for registration of free-form curves and surfaces. *International journal of computer vision*, 13:119–152, October 1994.

[27] Hans-Joachim Bohme Horst-Michael Gross Christof Schroter. Memory-efficient gridmaps in rao-blackwellized particle filters for slam using sonar range sensors. Technical report, Ilmenau Technical University, 98684 Ilmenau, Germany, 2007.

[28] Jeroen D. Hol, Thomas B. Schon, and Fredrik Gustafsson. On resampling algorithms for particle filters. Technical report, Department of Electrical Engineering. Linkoping University, 2006.

[29] G. Grisetti, C. Stachniss, and W. Burgard. Improved techniques for grid mapping with rao-blackwellized particle filters. *IEEE Transactions on Robotics*, 23(1):34–46, Feb 2007.

[30] W. Burgard, M. Moors, R. Simmons D. Fox, and S. Thrun. Collaborative multi-robot exploration. *IEEE International Conference on Robotics and Automation*, 2000.

[31] J.W. Fenwick, P.M. Newman, and J.J. Leonard. Cooperative concurrent mapping and localization. *IEEE International Conference on Robotics & Automation*, 2002.

[32] A. Howard, L.E. Parker, and G.S. Sukhatme. The sdr experience: Experiments with a large-scale heterogenous mobile robot team. *Proc. of the International Symposium on Experimental Robotics*, 2004.

[33] Lisa Gottesfeld Brown. A survey of image registration techniques. *ACM Comput. Surv.*, 24(4):325–376, December 1992.

[34] H. Foroosh, J.B. Zerubia, and M. Berthod. Extension of phase correlation to subpixel registration. *IEEE Transactions on Image Processing*, 2002.

[35] Szeliski Richard. *Computer Vision: Algorithms and Applications*. Springer London, 2011.

[36] Andreas Birk and Stefano Carpin. Merging occupancy grid maps from multiple robots. *IEEE Proceedings, special issue on Multi-Robot Systems*, 94(7):1384–1397, 2006.

[37] S. Carpin and G. Pillonetto. Robot motion planning using adaptive random walks. *IEEE International Conference on Robotics and Automation*, page 38093814, 2003.

[38] S. Carpin and G. Pillonetto. Planning for multi-robot exploration with multiple objective utility functions. *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2011.

[39] Kenneth I. Joy. Breshenhams algorithm. Technical report, Computer Science Department, University of California, Davis, 1999.

Appendix

See appended DVD.

1. Copy of Thesis
2. Original Source Code
3. New Developed Source Code
4. Code Documentation
5. Previous Work
6. User Manuals