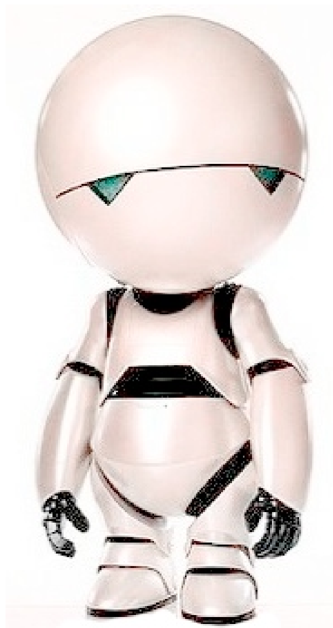
Localization and Mapping with Autonomous Robots

Benjamin Mader
Radboud University Nijmegen
Research number: 147 IK

May 16, 2011
Nijmegen, The Netherlands



Abstract

The field of robotics is becoming increasingly important in our society. Today, robots can be found in almost every aspect of our daily lives. Most of these robots are quite primitive and can only executed simple tasks. Some robots are more advanced and can be described as autonomous. So the question arises: which properties and skills does a robot need in order to be considered autonomous? The first thing that comes to mind is, that the robot must be able to move and perform other actions without direct external control information. Secondly, the robot needs a goal, as without a goal the robot does not know what to do and might perform erratic actions or non at all.

In order to perform movements without crashing into obstacles, a robot needs to see its environment. Another requirement for independent movement is a robots ability to find out where it is located in its current environment. These two skills enable a robot to circumnavigate directly visible obstacles and exhibit seemingly autonomous movements. If we want to let a robot navigate to a destination point, in other words supply it with a goal, the robot must be able to 'remember' the environment in order to plan a path from its position to the desired destination.

Additionally, sensors and actuators used by robots are usually very inaccurate and error prone. One can never rely on the correctness of measurements or the accurate execution of movement instructions. Therefore one must find a way to deal with this uncertainty during the localization and mapping processes.

So in order to construct an autonomous robot it needs to remember its environment, in other words: it has to generate a map. Therefore this activity is called *mapping*. It also must calculate its position relative to this map, this is called *localization*. This thesis will examine the theoretical foundations of these techniques and how they deal with the omnipresent problem of uncertainty. Additionally it will illustrate how to implement localization and mapping for a typical sensor equipped robot.
Keywords: Robot, localization, mapping, uncertainty, probability, belief

Contents

1 Introduction **5**
1.1 Chicken and Egg 6

2 USARSim **8**
2.1 USARSim Communication 9
2.1.1 Protocol 9
2.1.2 Toolbox 9

3 Localization **10**
3.1 Robot Localization Basics 13
3.1.1 Odometry 13
3.1.2 Belief 14
3.2 Particle Filter 15
3.2.1 The Basics 15
3.2.2 Particle Filter - A Nonparametric Bayes Filter Implementation 16
3.2.3 Odometry Motion Model 19
3.2.4 Particle Weights 21
3.2.5 Error Parameters 24

4 Mapping **27**
4.1 Challenges 27
4.2 Occupancy Grid Mapping 28
4.2.1 Log Odds Ratio 30
4.2.2 Inverse Laser Sensor Model 31
4.2.3 Learning Inverse Sensor Models 36

5 Implementation **37**
5.1 Start Scripts 37
5.2 Control Interface 38
5.3 Localization 39
5.3.1 run_particleFilter 40
5.3.2 Particle_Filter 41
5.3.3 sample_motion_model_odometry 43
5.3.4 weigh_Particle 44
5.4 Mapping 44
5.4.1 run_OGmapping 46
5.4.2 updateOGlaser 47
5.4.3 Bresenham 47

6 Discussion **48**
6.1 The autonomous robot 48
6.2 Localization 51
6.3 Mapping 53
6.4 Implementation 55

7 Acknowledgements **57**

References **58**

A USARSim Installation **59**
A.1 Unreal Tournament 2004 Installation 59
A.2 USARSim Installation 59
A.2.1 Windows 59
A.2.2 Linux 60
A.2.3 Mac OS X 60

List of Figures

1 Pop culture robots . . . 5
2 USARSim . . . 8
3 Probability Density Functions . . . 11
4 Gaussian PDFs . . . 12
5 Particle representation of a PDF . . . 16
6 Odometry Motion Approximation . . . 20
7 Sampling from the odometry motion model . . . 20
8 Occupancy grid map with measurement beams . . . 22
9 Importance factor calculation . . . 23
10 Particle filter behavior . . . 26
11 Occupancy grid map of an office . . . 29
12 Robot in office environment . . . 32
13 Visualized laser sensor measurements . . . 32
14 Bresenham's line algorithm . . . 34
15 Laser measurements embedded in occupancy grid . . . 35
16 Control interface . . . 38
17 Localization overview . . . 40
18 Mapping overview . . . 45
19 Fictional and non-fictional robots . . . 49
20 Global localization . . . 52
21 Comparison of Monte Carlo and grid-based localization techniques . . . 53
22 Inconsistencies in occupancy grid map . . . 54
23 Comparison of occupancy grid mapping techniques . . . 55

List of Tables

1 Bayes Filter 15
2 Particle Filter 17
3 Odometry Motion Model 21
4 Particle weighing 24
5 Occupancy Grid Mapping 30
6 Inverse Sensor Model 34

1 Introduction

> "A robot is an automatically guided machine which is able to do tasks on its own. Another common characteristic is that by its appearance or movements, a robot often conveys a sense that it has intent or agency of its own."[1]

The field of robotics is becoming increasingly important in our society. Today, robots can be found in almost every aspect of our daily lives, but most of the time robots are not recognized as such. Who would think that for example his automatic coffee maker is a robot? The reason for this is that when people think of robots, they think of robots as defined by pop culture, where we encounter humanoid robots like C3PO, Commander Data and even the infamous Terminator. Even when pop culture robots do not closely resemble human beings, they always possess some human attributes so that people do not view them strictly as cold and heartless machines.

Figure 1: Pop culture robots with increasingly humanoid anatomy[2]

Now let us take a look at some of the fundamental differences between a simple coffee maker and some of the previously mentioned, well known (although imaginary) robots. One of the main differences that comes to mind is the fact that a coffee maker cannot freely move around its environment. If we ask people why it cannot do that, most of them might say: Because the coffee machine has no wheels/legs to move around. And while this is of course true, it is not the whole story.

Suppose the coffee machine had some means of transportation, for example wheels, what else would it need to move in its environment? Firstly it needs software that can interact with the motors of the wheels. Now that the machine has wheels and software capable of operating them, it could theoretically move around. But one problem remains: how does the machine know where it is and where it can and cannot go, due to obstacles present in the environment? It does not, therefore it needs some kind of device that enables it to 'see' or recognize its surroundings in another way. So lets attach a sonar (sound navigation and ranging) scanner to the coffee

[1] http://en.wikipedia.org/wiki/Robot - 06.07.2010

[2] from left to right: Louie (Silent Running, 1972), R2D2 (Star Wars, 1977), B9 (Lost in Space, 1965), Cylon Centurion (Battlestar Galactica, 1978), T-800 (Terminator, 1984)

machine. This sensor emits sound waves and measures the time it takes for their reflections to return. The data of this sensor can be used to obtain accurate distance measurements. So now our coffee machine, through the use of a sonar scanner and software that is able to calculate distances from the scanners data, can 'see' its environment.

The question is, would people think of our enhanced coffee machine as a robot? I suppose most of them would, but think about what would happen if you turn the coffee machine on. The machine could move around through the use of its wheels, it might even be able to circumnavigate appearing obstacles if its movement software is able to access and process the measurements of the sonar scanner. This may look like autonomous and purposeful behavior to the layman and therefore the coffee maker could be considered a genuine robot. In reality, up to this point the robot is only able to see the directly visible part of its environment and react to it. But what about the parts of the environment which are not directly visible to the robot? Would it not be nice if the robot could 'remember' the obstacles it encountered and circumnavigated? This way the robot would be able to return to any of these obstacles directly, without looking for them again, all over the environment. An additional advantage of remembering the environment is, that it can help the robot to find out its current location within this environment. This activity is called *Localization* and it relies on sensor data as well as certain stored information about the environment in order to calculate the robot's position. This environmental information is best described as a map, this is the reason why gathering, processing and storing of this information is called *Mapping*.

The problems of Localization and Mapping are not easily solved. One thing further complicates the matter: *Uncertainty*. Localization and mapping both rely heavily on data from the robot's sensors and effecters/actuators (wheels, arms, legs, ...). The problem is that this data is inherently uncertain. No sensor and no actuator is 100% accurate, so in order to avoid calculation-errors we need to take this uncertainty into account. This means that we can calculate neither the robot's position nor the map of its environment in a deterministic way. We will need to make use of more complex, probabilistic techniques in order to obtain reliable results.

Localization and Mapping are two fundamental abilities any autonomous robot needs to possess in order to be able to efficiently navigate around its environment. But keep in mind that by simply implementing these two features, you will not automatically end up with a fully functional robot ready to execute its tasks. These two abilities form the basis for any further development of autonomous robots, so understanding and implementing them accurately and efficiently is of great importance to anyone wanting to work with autonomous robots.

In this thesis I will provide an insight into technical and mathematical foundations as well as detailed instructions that will illustrate how to implement selected localization and mapping algorithms for an autonomous sensor equipped robot using probabilistic techniques. The software development will be done in *Matlab*© and I will use the *Unified System for Automation and Robot Simulation* (USARSim) to provide virtual environments and simulate the robots.

1.1 Chicken and Egg

Before we dive into the details of localization algorithms and mapping techniques, let us take a look at a very basic property of the task at hand: the *chicken and egg* problem.
What is this problem all about? We want to solve two individual problems: *localization* and *mapping*. And this is exactly where we encounter the *chicken and egg* problem. In order to

execute the critical resampling step of the particle filter (see Table 2 and Section 3.2.4) a fairly accurate map of the environment has to be available. This means that we need to solve the mapping problem in order to solve the localization problem. On the other hand, in order to construct an accurate map of the environment, reliable information about the current pose is necessary (see Section 4). This implies that we need to solve the localization problem before we can begin to construct maps.

As it is now obvious why this is called a *chicken and egg* problem, we still need some kind of solution. There exists a technique which overcomes this problem and enables a robot to execute the two processes at the same time, as one builds on the results of the other. This technique is called *Simultaneous Localization and Mapping* (SLAM). Introducing SLAM is not the scope of this thesis and therefore we will make use of a 'cheat' to overcome the *chicken and egg* problem. We will assume that there is some kind of *oracle* that will provide the missing information (map or location). USARSim provides a so-called *Ground-Truth Sensor*. This sensor provides noise and error free pose information for the robot. From there on we can start constructing a map which can then be used by the particle filter for particle resampling. This way localization and mapping can be introduced without directly solving the SLAM problem.

2 USARSim

The Unified System for Automation and Robot Simulation is an open-source simulator for search and rescue robots. The simulator is based on the game *Unreal Tournament 2004* (UT2004) and makes use of its game and physics engine. The simulator provides the user with a small number of preconfigured robots and scenarios that can be used right away. Additionally the user can easily add new robots and/or configure the existing robots and their actuators/sensors to his liking.
As USARSim is based on Unreal Tournament, it can be run on every operating system that is supported by the game. Luckily UT2004 supports all major operating systems, these are Microsoft Windows, Mac OSX and Linux. USARSim, precompiled or as source code, can be downloaded from the USARSim Sourceforge page.[3]

Figure 2: USARSim, map: DM-TallTestWorld_250

USARSim offers simulation of robots and environments which are very close to reality. Simulated robots and sensors behave just like in real life, their data contains noise and errors and the communication interfaces are similar to their real world counterparts. The UT2004 engine used by USARSim is able to perform elaborate physics simulation which in turn leads to further increased realism in the simulation. These properties make USARSim a great platform to develop and test robot control software in a very comfortable and inexpensive way. Keep in mind though, that no simulation can simulate a 100% realistic environment. Therefore real life testing and verification/modification of any software developed and/or tested with USARSim will still be necessary.

For detailed installation instructions please consult Appendix A, while extensive information about USARSim can be found in the corresponding documentation [Wang 05].

[3]http://sourceforge.net/projects/usarsim

2.1 USARSim Communication

2.1.1 Protocol

USARSim uses a very straight forward communication protocol. All communication is done through plain text messages which are sent over a TCP connection. The program is reachable through the IP address of the computer it is running on and is listening for connections on port 3000. This standard port can be changed in the file `$UT2004/System/BotAPI.ini` in the section `[BotAPI.BotServer]`. Note that in USARSim, length is measured in meters, and angles in radians. For a listing of all possible messages please consult [Wang 05].

All messages sent by USARSim have the following form, which also messages sent to the program must satisfy:
`data_type {segment1} {segment2} ...`
The data_type and the first segment, as well as individual segments must be separated by one blank space. Each segment consists of one or more name-value pairs, names and values are also separated by one blank space.
`{Name1 Value1 Name2 Value2 ...}`
Blank spaces may not be used anywhere else in a message than in the above mentioned places! Two characters have to be appended to each message, *Carriage Return* followed by *Line Feed*, therefore `\r\n`. This character sequence indicates the end of the current message.

2.1.2 Toolbox

Communicating with USARSim can be tricky. There are a lot of messages to be sent and received while always ensuring up-to-date sensor data and lag free controls. Detailed information on this topic and an implemented communication program can be found in [Mader 10-1].

Implementing the communication protocol and program can be a time consuming process. Therefore we will use a freely available MatLab toolbox to communicate with USARSim. This toolbox is currently being actively developed at *Drexel University, Philadelphia* and a beta version can be downloaded from the university website.[4] Although currently only available as a beta version, the toolbox runs stably and offers sufficient functionality for this project.

The toolbox makes it very easy to add robots to the simulation, receive sensor readings and control the robot through simple MatLab commands. This works reliably and without any lag. A complete list of available commands can be found on the toolbox's download page.

Please note that in the current version of the toolbox (Beta 1.4), communication is limited to the local machine. This means that you have to run MatLab and USARSim on the same machine in order for them to communicate. Although it would be convenient to execute the simulation on a different machine than the control program, it poses no problem running them on the same machine, as long as it is powerful enough. The functionality to communicate with USARSim on remote systems will most likely be added in future versions of the toolbox.

[4] http://robotics.mem.drexel.edu/USAR/

3 Localization

The process of determining one's position in an environment is called *Localization*. All higher developed organisms have the ability to locate themselves within their surroundings. This ability seems to be 'hardwired' in their brains and happens mainly automatically or, in case of humans, subconsciously. We take it for granted to know where we are, therefore we do not think about the immense amounts of environmental information which has to be processed, stored, compared and finally used to derive the individual's real position, and all of this has to be done in *real time*.

Without going into to much detail, let us take a look at two intuitive ways to figure out one's location:

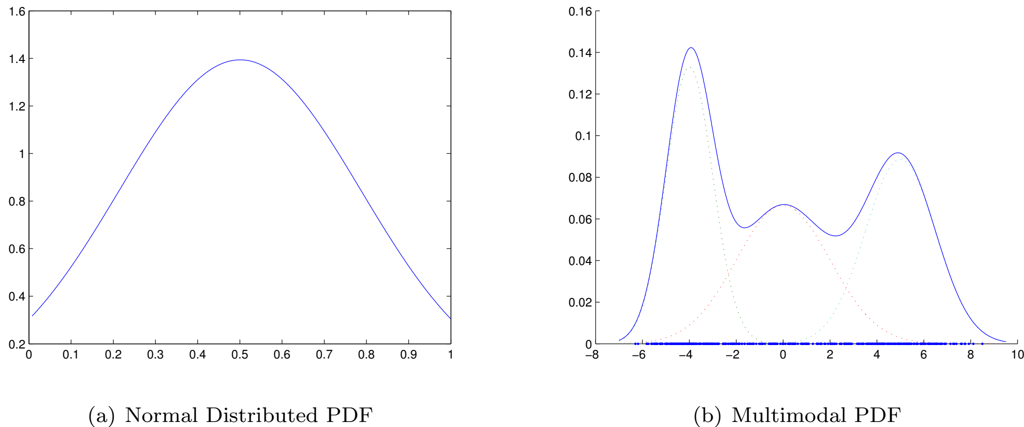
Odometry: This technique is based on a simple principle: If you know where you were and how you moved, then you know where you are! In other words: If you know your start position and remember every move you made since then, you can easily calculate your current position.

Sensing: This technique uses data from various sensory systems. The sensor data is compared to a description of the current environment. This comparison process enables us to determine the position relative to the stored environment description.

Both of these techniques have their own strengths and weaknesses, but there is one problem that is present in both approaches: *uncertainty*. Mobile robots contain various sensors and actuators to sense and manipulate the environment and change the robot's position. As it is almost impossible to build sensors and actuators which are error free, sensor measurements as well as movements will always be error prone. For example, if you instruct your robot to travel 1 meter in a straight line and measure the covered distance afterwards, the real covered distance will usually differ from the instructed distance. If the error were the same every time, it would be easy to take it into account. But as the error differs between measurements, you can never be sure how much error is present. The same is true for sensor measurements, for example distance measurements from sonar or laser scanners.

Somehow we have to deal with the uncertainty inherent in all measurements and control instructions, but how? Sensor measurements supply us with simple values, for example a distance, which already contain some error/uncertainty. The error is an attribute of the sensor that obtains the measurements, but it cannot be extracted from a single value. Therefore we take a series of measurements without moving the robot and calculate the error and other statistical data from them.
The same procedure can also be used for actuators. Instead of obtaining several measurements from the same position, we instruct the actuator to perform the same action several times and measure the results. We can then use this data to compute the necessary statistical properties of the actuator. Let us examine a simple example: Suppose we would like to identify the probabilistic properties of a certain robot's movement- apparatus. What we can do is instruct the robot to travel a certain distance and measure the true traveled distance afterwards. If we repeat this step several times, we can compute the average deviation from the instructed distance as well as corresponding mean and variance values.

Now that we know the properties of our robot's sensors and actuators, we can take uncertainty into account when working with their data. The straightforward way to do this would be to



(a) Normal Distributed PDF

(b) Multimodal PDF

Figure 3: Probability Density Functions

use a *probability density function* (PDF). There are many different types of PDFs and in this thesis we will make use of Gaussian distributions, also known as normal distributions. To model the uncertainty inherent in sensors and actuators, we construct a Gaussian PDF with the measurement value as its mean and its standard deviation defined by the measured sensor/actuator properties. So instead of relying on one value which we know is not absolutely correct, we have a whole range of possible values where some are more probable to be correct than others. A look at Figure 3a tells us that the measurements-mean, which represents the sensor-error corrected value measured by the sensor, is located at about 0.5. This is where the probability (of being the correct measurement) is the highest. To the left and right of the mean value, the probability drops significantly but it is still possible that one of these is the correct value which reflects the actual environment. How fast the probability drops is defined by the standard deviation which can be computed from consecutive measurements as described in the previous paragraph. Please see Figure 4 for an illustrated comparison of PDFs with varying parameters.

Apart from normal distributions, there is a plethora of other distributions to choose from. So what makes normal distributed PDFs more suitable for modeling sensor and actuator errors than other distribution types? First of all, the normal distribution is the most widely used and prominent probability distribution in the field of statistics. According to [Casella, Berger 02] there are several reasons for this:

Analytical tractability Gaussian distributions are very tractable analytically which means that many results, which involve this distribution, can be derived in explicit form.

Central limit theorem The normal distribution emerges from the central limit theorem. This theorem states that under normal conditions, the sum of many random variables is approximately normal distributed.

Bell shape The bell shape of the distribution makes it especially suited for modeling a great number of random variables which can be found in practice.

Please note that the benefits of the *analytical tractability* of Gaussian PDFs do not play a major role in this thesis. This is because in Section 3.2 we will introduce the so-called *particle filter*

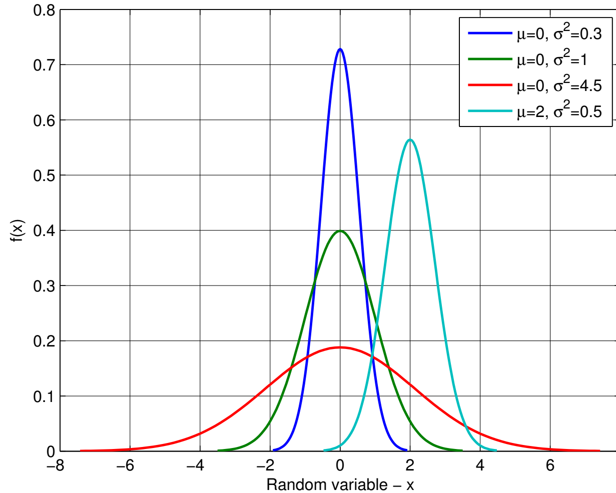


Figure 4: Gaussian PDFs with different variances σ^2 and means μ

which is able to reliably model *any* probability distribution.

Normal distributed PDFs (as in Figure 3a) provide a reliable way to model uncertainty, while their computational complexity is not too demanding and the resulting algorithms can be understood quite intuitively. The problem is, that normal distributed PDFs are not always the best way to model uncertainties in sensors/actuators. Often multiple properties of the sensor/actuator introduce uncertainty, which may be normal distributed or not, to the result. So in reality we will require non-normal distributed PDFs as well as multimodal PDFs (as in Figure 3b) and this leads to some drawbacks of the general PDF approach. First of all, the use of non-normal distributed or multi modal PDFs leads to algorithms which are not as easy to understand intuitively. Secondly the complexity of multimodal distributions, especially when used in a field as complex as robot localization, tends to be quite high. This complexity makes it difficult to accurately fit PDFs to the encountered real world uncertainty. Additionally, complex multi modal PDFs tend to complicate the process of 'inference', which means drawing conclusions from given facts.

In this thesis I will work with a technique which overcomes the disadvantages of the general PDF approach and is still able to model even the most complex PDFs. This technique is called the *Particle Filter*. The idea behind the particle filter is to approximate a PDF through a finite number of samples drawn from the PDF. These samples are called particles. Once a sufficient quantity of particles is used, this sampling approach has almost no apparent disadvantages when compared to directly working with PDFs. Instead it has the advantages of simplified complexity, intuitively comprehensible algorithms and the ability to approximate even the most complex PDFs.

Of course, the particle filter has some disadvantages of its own. The more particles are used to approximate a PDF, the more 'expensive' the calculations become in terms of computing time. Therefore it is important to carefully choose a quantity which leads to reasonably good approximation as well as sufficient speed. There is also no 100% reliable way to tell when a

particle distribution accurately describes the 'true' distribution it aims to approximate. Even though drawbacks exist, the particle filter's positive properties make it a very good choice when dealing with robot localization.

The particle filter's mode of operation and algorithms will be discussed in detail in Section 3.2.

3.1 Robot Localization Basics

Before we can begin to talk about localization techniques and algorithms, we need to take a step back and get familiar with some basic concepts and notions.

3.1.1 Odometry

The location of a robot is described by a pair of coordinates (x, y). These coordinates can either describe the robot's absolute location on the map or its location relative to a start position. In both cases the coordinate pair cannot be used to extract information about the robot's *orientation*, which describes the direction the robot is facing. Just like x and y, the orientation θ can be defined either absolute or relative and has to be added to the coordinate vector in order to describe all aspects of a robot's location. The resulting vector (x, y, θ) is called the *Pose* of the robot. This data structure contains all necessary information on the robot's current location, therefore virtually all localization algorithms will return a pose as their final result. Please note that such a pose (x, y, θ) allows only for two-dimensional localization. This might seem like a problem at first, but once a map of the environment is available, the z-coordinate can be easily recovered from it using the available x- and y-coordinates of the pose.

$$Pose = (x, y, \theta)^T \tag{1}$$

Now that we know how we can describe a robot's location and orientation, the question remains how to find out the current pose or, in other words, calculate the current values x, y and θ. Reliably computing a correct pose is no trivial task and providing a simple enough procedure to do this is exactly the goal of this thesis. Contrary to the complexity of computing the 'true' pose, an approximated or estimated pose is quite easy to compute. The way this is done is through so-called *odometry*.

> "Odometry is the use of data from moving sensors to estimate change in position over time. Odometry is used by some robots, whether they be legged or wheeled, to estimate (not determine) their position relative to a starting location. This method is sensitive to errors due to the integration of velocity measurements over time to give position estimates. Rapid and accurate data collection, equipment calibration, and processing are required in most cases for odometry to be used effectively." [5]

Suppose we use a wheeled robot. We know the circumference of the wheels as well as their individual positions relative to the center of the robot. If we equip the wheels with special sensors, so-called *Rotary Encoders*, then we are able to measure every wheel's rotation. As we now know the circumference and position of the individual wheels as well as their individual

[5] http://en.wikipedia.org/wiki/Odometry - 20.11.2010

amounts of rotation, we can easily calculate the robot's position relative to the start position. Not only does this technique work for traveling in a straight line, it works for every conceivable path of the robot.

Although it is quite easy to estimate the pose of a robot, relying strictly on odometry data for localization is not a good idea. The problem is that rotary encoder data contains a lot of noise and is very error prone, just think of the wheel slip on a slippery surface. Additionally, the way the pose is computed from wheel-spin data can only supply the correct result if the input data was absolutely correct. As we know that the input data, provided by the rotary encoders, most definitely contains noise and errors, the output pose will incorporate these flaws as well. As the robot continues to move, the errors accumulate and after a short while the calculated position will start to deviate a lot from the actual position of the robot. Therefore odometry is not suited to single-handedly solve the localization problem. But, as we will see in Section 3.2, odometry data can be used as input for more complex localization algorithms.

3.1.2 Belief

In a world of noisy sensors, error prone actuators and ever changing environmental conditions, *uncertainty* is the only real constant. Not only do we have to take into account the uncertainty inherent in the pose calculated either by odometry or a more advanced localization technique, but sometimes we even get several poses as output of a localization algorithm, each with a certain probability of being the correct pose.

What a robot knows about the state (e.g. its pose) of its environment, is reflected by its *belief*. The state of the environment can usually not be measured directly. Every robot most definitely has a pose relative to some coordinate system, but usually it is not capable to measure this pose directly and therefore does not 'know' its pose. Instead of measuring a state directly, the robot computes it from collected data by inference. Therefore we have to differentiate between two concepts: The true state present in the environment and the robot's *belief* in this specific state.

In the probabilistic approach, which this thesis is based on, beliefs are represented through conditional probability distributions which assign probabilities to any possible hypothesis with regard to the true state. The posterior probabilities over state variables, represented by the belief distribution, are dependent on available data. The belief over a state variable x at some time t is denoted by $bel(x_t)$. The belief is dependent on two variables or data sources. The first one is z which stands for the measurements of the robot. Additionally u describes the control information that the robot has received. The complete formula for the posterior is given in Equation (2).

$$bel(x_t) = p(x_t|z_{1:t}, u_{1:t}) \tag{2}$$

Equation (2) illustrates that the belief depends on all measurements and controls of the robot up to the moment of belief calculation. This is a quite undesirable property which can be overcome. It is possible to keep a limited history of measurements and controls to use for computation. In practice, this undesirable property is overcome by incorporating the previous belief in the calculation, using it to compute the current belief with just the most recent control and measurement. This works because all previous controls and measurements are already incorporated in the previous belief. Please consult the chapter *Recursive State Estimation* in [Thrun, Burgard, Fox 06] for comprehensive information on this topic.

3.2 Particle Filter

In this section I will provide detailed information on a nonparametric implementation of the Bayes filter, the so-called *particle filter.* Before engaging in the operating mode and algorithms of the particle filter, we will explore some basic properties of the underlying technique: the Bayes filter. As the particle filter is a nonparametric implementation of the Bayes filter, information on the basic concept will be useful when examining its nonparametric implementation.

3.2.1 The Basics

A Bayes filter is a probabilistic technique which allows us to recursively estimate an unknown probability density function, therefore this technique is also known as *recursive Bayesian estimation.* Bayes filters rely on mathematical models and continuous measurements to estimate the PDF.

> "A Bayes filter is an algorithm used in computer science for calculating the probabilities of multiple beliefs to allow a robot to infer its position and orientation. Essentially, Bayes filters allow robots to continuously update their most likely position within a coordinate system, based on the most recently acquired sensor data. This is a recursive algorithm. It consists of two parts: prediction and innovation. If the variables are linear and Gauss-distributed the Bayes filter becomes equal to the Kalman filter."[6]

Of course Bayes filters cannot only be used to estimate a robot's position and orientation (which together are called the *pose*), but any state variable. As this document deals with localization and mapping, the pose is the logical prime example of a state and will be used as such throughout the thesis.

The Bayes filter algorithm accomplishes the calculation of the *posterior* belief from the previous belief and the most current control and measurement data.

1: **Bayes_filter(**$bel(x_{t-1}), u_t, z_t$**):**
2: **for all** x_t **do**
3: $\overline{bel}(x_t) = \int p(x_t|u_t, x_{t-1})\ bel(x_{t-1})\ dx_{t-1}$
4: $bel(x_t) = \eta\ p(z_t|x_t)\ \overline{bel}(x_t)$
5: **end for**
6: **return** $bel(x_t)$

Table 1: Bayes filter algorithm, [Thrun, Burgard, Fox 06]

The basic algorithm of the Bayes filter is given in Table 1. The algorithm input consists of the belief at time $t-1$ ($bel(x_{t-1})$) and the most recent controls (u) and measurements (z) at time t. It computes the current belief $bel(x_t)$ from the previous one $bel(x_{t-1})$, which makes the Bayes Filter a *recursive* algorithm.

The Bayes filter is made up of two central steps. The so-called *control update* or *prediction* step can be found in Line 3. It uses the previous belief and most recent control u_t to calculate a

[6]http://en.wikipedia.org/wiki/Bayes_filter - 01.12.2010

belief over the state x_t. The belief obtained is the integral of the product of two probability distributions, $bel(x_{t-1})$ and the probability that control u_t introduces a change from x_{t-1} to x_t. In other words this step predicts the probability $p(x_t|u_t, x_{t-1})$ of reaching state x_t from x_{t-1} by executing the control u_t. Line 4 is called *measurement update* and it is quite obvious why. It multiplies the result of the prediction step ($\overline{bel}(x_t)$) with the probability of measurement z_t being observed at state x_t. You might ask yourself why there is an additional component (η) present in Line 4. The problem is, that the result of the calculation in Line 4 is generally not a valid probability as it may not integrate to 1. Therefore we introduce η as a normalization constant to ensure that the result $bel(x_t)$ will integrate to 1 and therefore is a probability.

As this is a recursive algorithm, we need an initial belief at time $t = 0$. There are 2 possible ways to initialize $bel(x_0)$:

If the value of x_0 is measurable or known, then we should initialize $bel(x_0)$ so that the probability of the correct x_0 is 1 but 0 everywhere else. If x_0 is unknown, then $bel(x_0)$ should be initialized as a uniform distribution over the whole domain of x_0, which means that every possible value of x_0 has the same probability.

The Bayes filter, as defined in Table 1, can only be used for very simple estimation problems. This is due to mathematical limitations in the algorithm. To be more precise, we need to be able to carry out the integration and the multiplication in Lines 3 and 4 of Table 1 in closed form. To be able to do that, we can restrict the complexity of the problem or work with finite state spaces in order to make the integral in Line 3 a finite sum. To overcome these and other limitations, we will now focus on a nonparametric Bayes filter implementation, the particle filter.

3.2.2 Particle Filter - A Nonparametric Bayes Filter Implementation

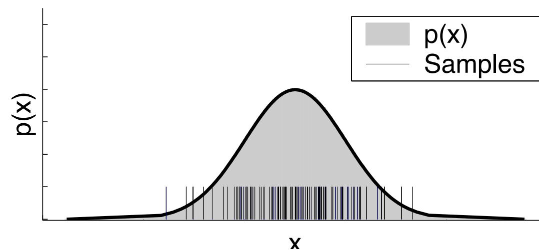


Figure 5: Particle representation of a PDF, [Thrun, Burgard, Fox 06]

As mentioned before, the particle filter is a nonparametric implementation of the Bayes filter. The basic idea behind this technique is to approximate the posterior distribution by a finite number of so-called *particles*, which are samples drawn from the same posterior $bel(x_t)$. See Figure 5 for an illustration of this technique. The advantage of this nonparametric approximation technique is that it is able to represent a much wider range of probability distributions than parametric techniques, for example Gaussian distributions.

$$x_t^{[m]} = (x, y, \theta)^T \tag{3}$$

Before we dive into the inner workings of the particle filter, let us take a closer look at the

particles used to approximate the posterior. The domain of this posterior is the robot's pose. Therefore the particles used to approximate this posterior have to be from this domain as well. The formal definition of particles can be found in Equation (3).

$$X_t := x_t^{[1]}, x_t^{[2]}, ..., x_t^{[M]} \tag{4}$$

The set of particles is defined in Equation (4). Each of the particles represents a hypothesis for the true state at time t. How many particles the set contains is denoted by M, which is usually quite a large number. As we want to approximate the belief $bel(x_t)$ through the use of these particles, the more particles we use (larger M) the better the approximation will be but at the same time more particles lead to longer computation times. Not all state hypotheses are equally likely to be included in X_t. Their likelihood of inclusion is proportional to their Bayes filter posterior, which is given in Equation (5).

$$x_t^{[m]} \sim p(x_t | z_{1:t}, u_{1:t}) \tag{5}$$

This posterior indicates how probable it is that x_t is the true state. As you can see, the resulting probability is dependent on all previous controls ($u_{1:t}$) and measurements ($z_{1:t}$). This has the consequence that, the more samples are located in a subregion of the space of all possible states, the more likely it is that the true state is also located in this region. To be precise, the property given in Equation (5) only holds if M asymptotically approaches ∞. Working with almost infinite sets would be computationally unfeasible, therefore we will use finite sets of particles. Finite sets of particles are drawn from slightly different distributions but in practice the differences are not noticeable as long as a sufficient number of particles is used.

Like all Bayes filters, the particle filter is a recursive technique. This means that the current belief $bel(x_t)$ is calculated from the previous belief $bel(x_{t-1})$. As the particle filter represents the belief through a set of particles, this naturally implies that the current set of particles X_t is computed from the previous set X_{t-1}. Additional to X_{t-1} the particle filter needs the current controls u_t and measurements z_t as input for its calculations. A basic algorithm for the particle filter can be found in Table 2.

```
1: Particle_filter(X_{t-1}, u_t, z_t):
2: X̄_t = X_t = ∅
3: for m = 1 to M do
4:    sample x_t^[m] ~ p(x_t | u_t, x_{t-1}^[m])
5:    w_t^[m] = p(z_t | x_t^[m])
6:    X̄_t = X̄_t + ⟨x_t^[m], w_t^[m]⟩
7: end for
8: for m = 1 to M do
9:    draw i with probability ∝ w_t^[i]
10:   add x_t^[i] to X_t
11: end for
12: return X_t
```

Table 2: Particle filter algorithm, [Thrun, Burgard, Fox 06]

The particle filter algorithm requires three input parameters to compute the current particle set:

the particle set X_{t-1} which was calculated in the previous time step, the current controls u_t and measurements z_t. The algorithm consists of two *for* loops which basically split the algorithm into two parts. At first an *extended* temporary particle set $\bar{X}_t$ is constructed. It is called extended because it contains not only particles but also their respective importance-factors or weights. After this set has been constructed, the algorithm iterates through all the particles in the input set X_{t-1} and uses each of them as a starting point to calculate a particle for the temporary set. The first part is explained in detail below:

1. In Line 4 a so-called hypothetical state is generated. As you can see, this state is dependent on the previous particle at the particular index m and on the controls the robot has received. Basically this line takes into account where the robot was before ($x_{t-1}^{[m]}$) and what it has done since then (u_t). Instead of deterministically calculating where the robot is now, uncertainty is taken into account by sampling from a probability distribution. Please refer to Section 3.2.3 where you can find detailed information on this step.

2. Line 5 calculates an *importance factor*, or *weight*, w for each particle. This importance factor equals the probability that the measurement z_t has been observed under state (particle) $x_t^{[m]}$, so naturally w can only be a value between, and including, 0 and 1. For more information on this step, please refer to Section 3.2.4.

3. In Line 6 the weighed particles are appended to the temporary extended particle set. Once all particles of X_{t-1} have been considered, the temporary set of weighed particles represents an approximation of the Bayes filter posterior $\overline{bel}(x_t)$.

After the first loop has been executed, the real point of the particle filter comes into play: *resampling* or *importance sampling*. This step, executed in Lines 8 to 11 in Table 2, generates the resulting particle set X_t which is the same size as $\bar{X}_t$. In Line 9 the algorithm draws (with replacement) M particles from the temporary set. The probability to draw a certain particle from the set is given by its *importance factor*. Particles are drawn with replacement, this means that it is possible to draw the same particle several times. This causes the resulting particle set to include several duplicate particles, but more importantly this step leads to omission of particles with a low *importance factor*. In short this means that the resulting set contains more particles with high weights and less particles with low weights. This means that less particles end up in regions with low posterior probability. This leads to the situation that the resulting particle set consists mostly of the particles with a high weight or, in other words, the most probable 'true' particles. The resampling step can be compared to the process of evolution where only the strongest and best are able to survive (*survival of the fittest*).

One topic which has not yet been discussed is the question of *particle initialization* or, in other words, what particles does the first set contain before the particle filter is executed the first time? Basically there are two options:

1. Initialize all particles with the known starting pose of the robot. This can be either $x_t^{[m]} = (0, 0, 0)^T$ if the basis for initialization is odometry information, or the absolute coordinates and rotation in the environment, for example $x_t^{[m]} = (-12.334, 5.329, 0.341)^T$

2. Draw random particles from the state space. A particle is a hypothesis of the true pose at time t. As, in the beginning, we know nothing about the location of the robot, it is only logical to randomly initialize particles. The first particle set will therefore contain particles that represent random poses at random locations. The particle filter's resampling

technique (Lines 8-11 in Table 2) will, little by little, eliminate the bad particles and keep the good ones. The only disadvantage to the first initialization technique is that it might take a little while until the inaccurate particles are removed from the set.

The particle filter algorithm is quite intuitive, but until now two lines are still not illustrated in full detail. It might not be absolutely clear how to calculate the hypothetical state in Line 4 and the importance factor in Line 5, therefore I will discuss them in detail in the next two sections.

3.2.3 Odometry Motion Model

In order to calculate the hypothetical state in Line 4 of Table 2, we need to know about the influence of control data on the state (in our case the pose) of the robot. There are many different models that can be used. In my opinion the most intuitive and easy to use model is the so-called *Odometry Motion Model*. This model uses odometry measurements instead of real control information to model the motion of the robot. Real control information would be data which directly controls the robot, for example instructions for wheel velocity or steering angles. It might be surprising that modeling robot motion with error-prone odometry data is usually more accurate than modeling it with, for example, velocity information. This is due to the fact that, when measured at small enough intervals, the difference between two consecutive odometry measurements is very close to the difference between the true poses in reality. However, one problem with the odometry model is that data can only be acquired in retrospect, after the robot has already moved. This property has no influence whatsoever on the usability of the odometry motion model for localization and mapping, but keep in mind that it makes this model unusable for accurate motion planning and control.

Before we can dive into the algorithm of the odometry motion model, we first need to define the format and properties of our control data. Remember that the correct pose of our robot at time t is defined by the random variable x_t. The robot's odometry estimates or approximates this pose. Due to the approximate nature of odometry calculations and the uncertainty inherent in all (odometry-)sensor measurements, we cannot define a deterministic transformation between the odometry pose and real world coordinates. Instead we use two consecutive odometry measurements, $\bar{x}_{t-1} = (\bar{x}, \bar{y}, \bar{\theta})^T$ and $\bar{x}_t = (\bar{x}', \bar{y}', \bar{\theta}')^T$, to estimate the robot's relative motion from true pose x_{t-1} to pose x_t. As indicated by the indices of x, this motion takes place in time interval $(t-1, t]$. The logic behind this estimation is, that the difference between the odometry poses $\bar{x}_{t-1}$ and $\bar{x}_t$ will be very close to the difference between the true poses x_{t-1} and x_t, especially if the time interval is very small. Therefore we can use this data to calculate an approximated x_t from a known, usually also approximated, x_{t-1}.

An important property of any motion defined by two coordinates from consecutive time steps is, that this motion can be approximated by two rotations and one translation. These three steps are illustrated in Figure 6. At first, we rotate (δ_{rot1}) the robot into the direction of the new coordinates, then it travels (δ_{trans}) in a straight line until the coordinates are reached. Finally the robot rotates again (δ_{rot2}) until it reaches the desired final heading. Every movement from one pose to another can be performed in this way and even if the movement of the robot was not performed by executing these three steps, then it can still be approximated by them. The odometry motion model sampling algorithm in Table 3 makes use of this property in order to estimate the real pose of the robot. Keep in mind that all rotations and translations of the robot are noisy, therefore we have to 'add' some uncertainty to our calculations. This is accomplished

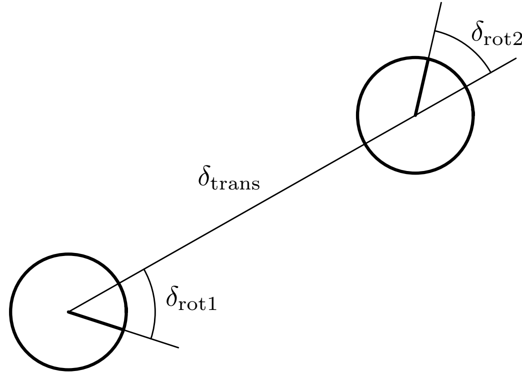


Figure 6: Odometry motion approximation, [Thrun, Burgard, Fox 06]

by the error parameters α_1 to α_4, the values of which have to be adjusted to the properties of the robot in question. See Section 3.2.5 for details about these parameters.

Figure 7 shows an example of a circular motion being approximated by several particles. The particles have been calculated by the algorithm in Table 3, making use of the principle illustrated in Figure 6 and the three sub-figures of Figure 7 show the effects of different values for translational and rotational error terms used to model the noise inherent in every robot motion. This should provide a good example of the influence of different error parameters on the particles. Figure 7a illustrates a particle distribution for typical error parameters. The form of the particle cloud shows that the *translational error* is lower than the *rotational error.* In the direction of translation, the particles are quite close to the actual position whereas they spread out considerably to the left and the right, caused by the larger rotational error. In Figure 7b the translational error is unusually large while the rotational error is abnormally low compared to the typical situation. In Figure 7c it is the rotational error which has been exaggerated while the translational error is exceptionally low.

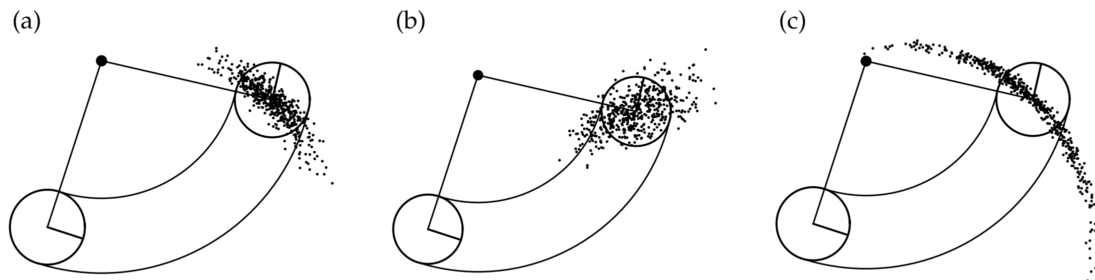


Figure 7: Sampling from the odometry motion model, [Thrun, Burgard, Fox 06]

The algorithm in Table 3 implements the functionality to sample from the distribution $p(x_t|u_t, x_{t-1})$, which is required in Line 4 of Table 2. It accepts control information $u_t = (\bar{x}_t, \bar{x}_{t-1})^T$, consisting of two consecutive odometry measurements ($\bar{x}_{t-1} = (\bar{x}, \bar{y}, \bar{\theta})$, $\bar{x}_t = (\bar{x}', \bar{y}', \bar{\theta}')$), and the pose at the previous time step $x_{t-1} = (x, y, \theta)^T$ as its input. The output of the algorithm is a random

pose x_t, distributed according to $p(x_t|u_t, x_{t-1})$ and calculated using the approximation principle presented in Figure 6. Every particle in Figure 7 represents one 'random' pose as calculated by the algorithm in Table 3.

```
1:  sample_motion_model_odometry(u_t, x_{t-1}):

2:  δ_rot1 = atan2(ȳ' − ȳ, x̄' − x̄) − θ̄
3:  δ_trans = sqrt((x̄ − x̄')^2 + (ȳ − ȳ')^2)
4:  δ_rot2 = θ̄' − θ̄ − δ_rot1

5:  δ̂_rot1 = δ_rot1 − sample(α1 δ_rot1^2 + α2 δ_trans^2)
6:  δ̂_trans = δ_trans − sample(α3 δ_trans^2 + α4 δ_rot1^2 + α4 δ_rot2^2)
7:  δ̂_rot2 = δ_rot2 − sample(α1 δ_rot2^2 + α2 δ_trans^2)

8:  x' = x + δ̂_trans cos(θ + δ̂_rot1)
9:  y' = y + δ̂_trans sin(θ + δ̂_rot1)
10: θ' = θ + δ̂_rot1 + δ̂_rot2

11: return  x_t = (x', y', θ')^T
```

$$\begin{array}{rl}
1: & \textbf{sample_motion_model_odometry}(u_t, x_{t-1})\textbf{:} \\
2: & \delta_{rot1} = \mathrm{atan2}(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \bar{\theta} \\
3: & \delta_{trans} = \sqrt{(\bar{x} - \bar{x}')^2 + (\bar{y} - \bar{y}')^2} \\
4: & \delta_{rot2} = \bar{\theta}' - \bar{\theta} - \delta_{rot1} \\
5: & \hat{\delta}_{rot1} = \delta_{rot1} - \textbf{sample}(\alpha_1 \delta_{rot1}^2 + \alpha_2 \delta_{trans}^2) \\
6: & \hat{\delta}_{trans} = \delta_{trans} - \textbf{sample}(\alpha_3 \delta_{trans}^2 + \alpha_4 \delta_{rot1}^2 + \alpha_4 \delta_{rot2}^2) \\
7: & \hat{\delta}_{rot2} = \delta_{rot2} - \textbf{sample}(\alpha_1 \delta_{rot2}^2 + \alpha_2 \delta_{trans}^2) \\
8: & x' = x + \hat{\delta}_{trans} \cos(\theta + \hat{\delta}_{rot1}) \\
9: & y' = y + \hat{\delta}_{trans} \sin(\theta + \hat{\delta}_{rot1}) \\
10: & \theta' = \theta + \hat{\delta}_{rot1} + \hat{\delta}_{rot2} \\
11: & \textbf{return} \;\; x_t = (x', y', \theta')^T
\end{array}$$

Table 3: Odometry motion model sampling algorithm, [Thrun, Burgard, Fox 06]

The sampling algorithm in Table 3 consists of 3 main parts:

1. In Lines 2 to 4, the three components necessary for the approximation of the actual movement, are calculated from the two consecutive odometry poses.

2. Lines 5 to 7 incorporate uncertainty into the calculations performed in the previous lines. They take the deterministically calculated values and add a random error. This random error is generated by the function **sample**(b^2). This function samples a random number from a zero-centered normal distribution with variance b^2 or, in other words, standard deviation b. As you can see in Lines 5 to 7, the value of b is dependent on the specified error parameters α_1 to α_4. Therefore the amount of randomly added error/noise corresponds to the noisiness of the robot's active components. Information on how to define these parameters correctly can be found in Section 3.2.5.

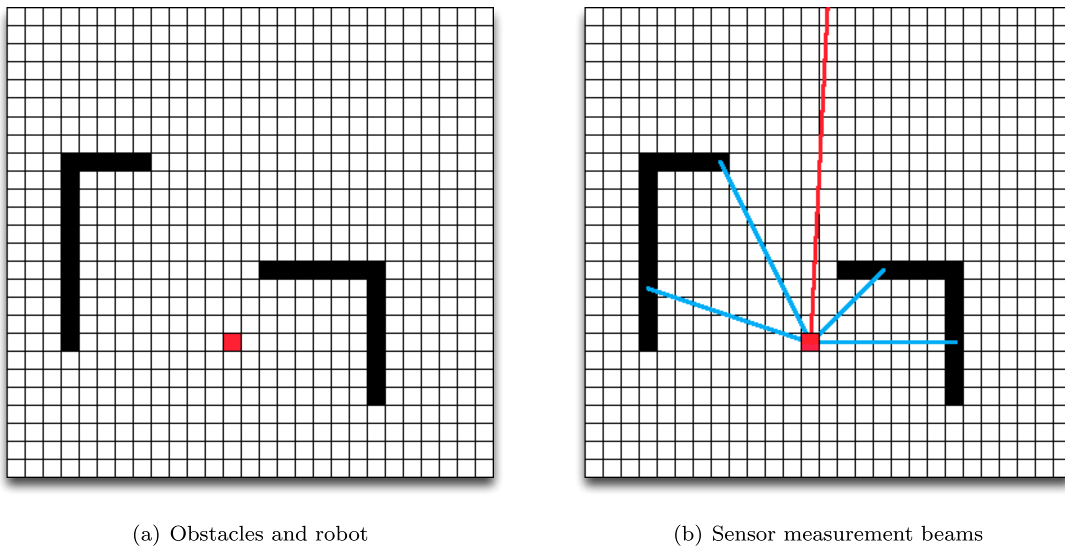
3. Lines 8 to 10 calculate the new pose by combining the previous pose, $x_{t-1} = (x, y, \theta)^T$ with the three error adjusted movement components. Line 11 defines the return value of the algorithm: the new pose.

3.2.4 Particle Weights

In order for the particle filter to execute the crucial *resampling* step, a weight or importance factor has to be calculated for every particle. This factor equals the probability of the measurement z_t being observed from the location defined by the particle/pose x_t, therefore w can only be a value between, and including, zero and one. Equation (6) formally defines this property of the importance factor:

$$w_t = p(z_t|x_t) \tag{6}$$

In order to calculate the importance factor of a certain particle we need a fairly accurate map. For now, we will assume that such a map is available. Please consult Section 1.1 for more information and justification of this decision. Please note that the maps discussed in this section are occupancy grid maps as described in Section 4.2, so it might be useful to consult this section if one has no prior knowledge about this type of map. Although the technique for calculating the importance factor does not depend on a certain kind of range sensor, the illustrations and explanations in this chapter are based on laser scanners as described in Section 4.2.2.

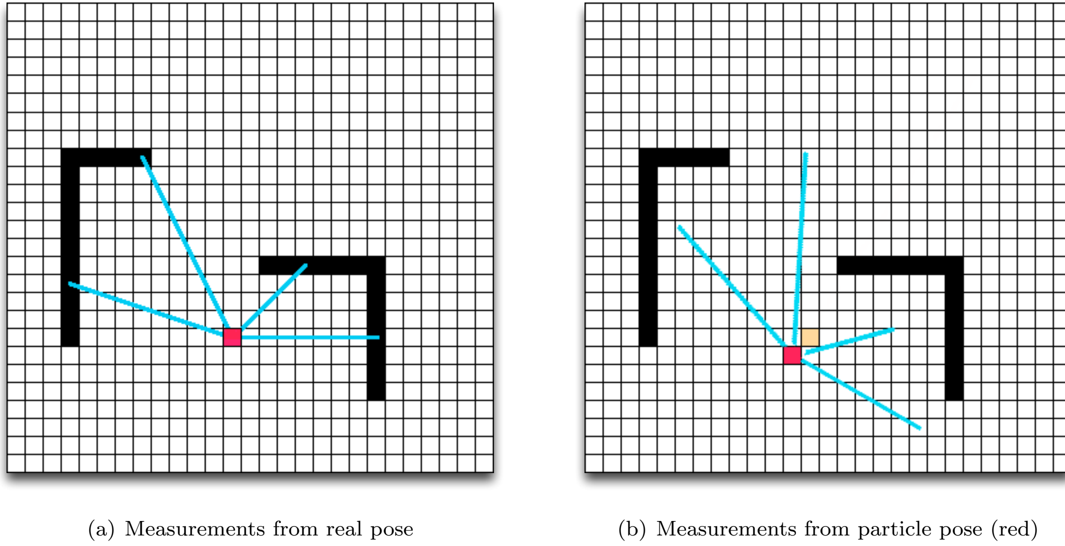


(a) Obstacles and robot

(b) Sensor measurement beams

Figure 8: Occupancy grid map with measurement beams

Figure 8a shows a simple occupancy grid map. The cell representing the *real* pose of the robot is marked in red while the black cells represent occupied space. Occupancy grid maps are defined so that the greyscale color value of a cell directly reflects the likelihood that this cell is occupied by an obstacle, more precisely dark cells have high occupational likelihoods while lightly colored cells exhibit a low likelihood of occupation. Figure 8b shows the same situation as 8a and additionally illustrates five laser scanner measurements. As you can see, four of those measurements (marked blue) hit an obstacle therefore correctly measuring the distance to it. The fifth measurement beam (red) does not hit an obstacle, which means that the reported distance will be the scanner's maximum range. Measurements like this, which do not hit an obstacle, must be found and excluded from the set of measurements before calculating the importance factor. The reason for this is simple: we want to rate a pose based on the measured obstacles. If a measurement did not hit an obstacle, it does not contain any additional information about the obstacles on the map, therefore we can safely discard them. In the special case, where the environment features big open spaces with no obstacles within the sensors maximum range, this method of weighing might not work satisfactory. In this case, all measurements would be discarded, making the range sensor useless. In this case other means would have to be found, for example analysis of pictures gathered by onboard cameras or sensors with a longer range.



(a) Measurements from real pose

(b) Measurements from particle pose (red)

Figure 9: Importance factor calculation

Figure 9a shows the robot's real pose with only valid measurements, the invalid ones have been removed from the set. This results in the situation that every laser measurement in the set specifies the distance to an *existing* obstacle. This set will then be used in cooperation with any arbitrary pose/particle to calculate the probability that these measurements were made from the pose specified by the particle. If the particle is very close to the *true* pose, then all or most of the measurements in the set will still be located in the same *correct* cell as if the real pose was used for calculation, basically identical to Figure 9a. On the other hand if the particle deviates a great deal from the true pose, then the projected measurements will be located in different cells, as illustrated in Figure 9b where the red cell marks the particle pose and the orange cell the real pose. Please keep in mind that poses that are very close together will mostly be located in the same cell of the occupancy grid as this grid is a discrete representation of the environment.

In almost every environment there is more free- than occupied space and from this we can reason that the probability is very high that, opposed to the true pose, not all measurements will be located in occupied cells if a deviated particle is used as their origin. So the average likelihood of the measurement cells in the case of an arbitrary particle tends to be lower than in the case of the true pose or a particle located close to this pose. Therefore we will use this average likelihood of the measurement cells as an indicator of the probability that a particle is *good* or not.

Now that we have established the basic technique, let us examine the associated algorithm in Table 4:

1. Line 2 calculates the grid cell locations of all single measurement values.

2. The loop in Line 3 iterates through all measurement cells and appends their occupational likelihood value to the set $\mathbf{M}_{z_t}$.

3. In Line 7 the average likelihood w is calculated while it is returned in Line 8.

1: **weigh_particle(x_t, z_t):**
2: $\mathbf{M}_{z_t} = calculateGridCells(x_t, z_t)$
3: **for all** cells $\mathbf{m}_i \in \mathbf{M}_{z_t}$ **do**
4: $w_i = occupationalLikelihood(\mathbf{m}_i)$
5: $\mathbf{M}_{z_t} = \mathbf{M}_{z_t} + \langle w_i \rangle$
6: **end for**
7: $w = average(\mathbf{M}_{z_t})$
8: **return** w

Table 4: Particle weighing algorithm

3.2.5 Error Parameters

Error parameters are used to incorporate noise and eventual errors of a robot's active components. These components, called actuators, are able to perform movements. Examples for actuators are wheels, grapplers and so on. This section will focus on the error parameters of a robot's movement system, in the majority of cases consisting of several actuators, for example wheels. Therefore the explanatory notes will be tailored to this field of application and especially the algorithm in Table 3. In this case we have to deal with two distinct kinds of errors.

- The *rotational error* denotes a robot's average deviation (in relation to a full rotation) from the desired amount of rotational movement. If you instruct a robot to rotate a certain amount and measure the actual amount it rotates, then you will find a difference between the two values. The average of this difference, calculated over many measurements, gives you an indication of how precise the robot is when executing rotational movements.
- The *translational error* denotes a robot's average deviation (in relation to a translation of 1 meter) from the desired amount of translational movement. We measure this error the same way as the rotational one. It indicates a robot's precision in executing straight line movements (translations).

As mentioned before, both error parameters have to reflect the amount of error in relation to either a full 360 degree rotation or, in case of the translational error, to a translation of 1 meter. Therefore these parameters can only be values between zero and one, which are equivalent to 0% and 100% of the performed movement.

The algorithm illustrated in Table 3, like many others in this thesis, is taken from the book *Probabilistic Robotics* ([Thrun, Burgard, Fox 06]). Unfortunately this book contains no explanation for the error parameters α_1 to α_4, nor are there any details on their individual roles. Therefore we have to take a look at the algorithm shown in Table 3 to find out more about them. As you can see in Lines 5 to 7 of said algorithm, α_1 and α_4 are only related to the rotational components, therefore we can safely assume that they model the *rotational error* of the robot. It is also obvious that α_2 and α_3 model the *translational error* of the robot.

After testing different error parameter values, some calculated from measurements and some chosen randomly, I can conclude that the precise value of the parameter is not extremely impor-

tant, as long as some error is present. Of course large error parameters lead to fast deviation of the particles from the true pose while small ones slow down this deviation. This difference can almost be neglected if the resampling process in the particle filter is active, as resampling removes the worst particles and keeps the good ones in the set. So even if you specify error terms that are way too large, the resampling step will take care of extremely deviated particles.

As the exact value of the error parameters does not have a huge effect on the functionality of the particle filter, and it seems to be better to use worst case error terms and let resampling handle the deviated particles, I decided to use the measured worst-case error terms. Therefore the parameters for the USARSim robot *P2AT*, the model used during this project, were chosen as follows:

- Rotational parameters: $\alpha_1 = \alpha_4 = 0.05$
- Translational parameters: $\alpha_2 = \alpha_3 = 0.1$

These values lead to a good behavior of the particle filter which can be observed if you execute the algorithm shown in Table 2 without the resampling step and monitor the output. See Figure 10 for an illustration of the particles generated using the specified error parameters.

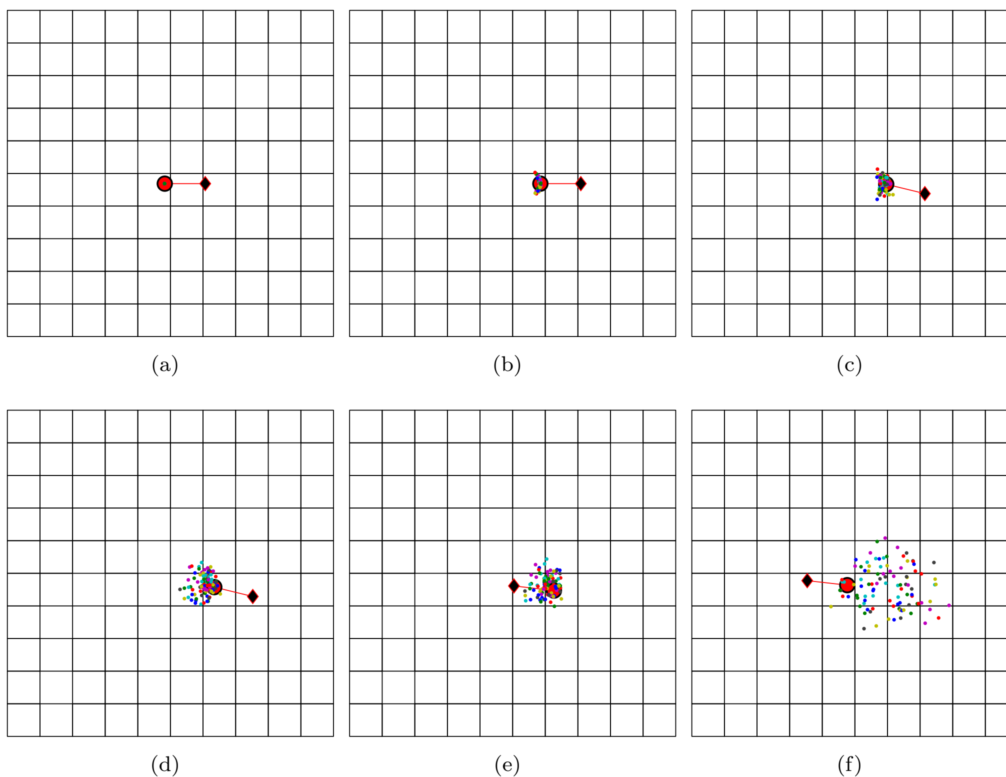


Figure 10: In each of the subfigures one can see the robots position, indicated by the big red circle, and its orientation, where the direction it is facing is indicated by the line with the rectangle at its end. The particles are indicated by a single dot. Note that only the location of the particles but not their orientation is illustrated in the subfigures.
Figure 10a shows the robot's position and orientation at the start position as returned by the Ground-Truth sensor. As all particles are initialized with this start position, all of them are located there and therefore there is only a single dot (particle) visible. In Figure 10b you can see that after the robot has traveled along a straight line, the particles begin to deviate from the robot's real position, in the direction of translation as well as to the sides of the robot in a circular fashion. This illustrates the influence of the error parameters. Figure 10c and d show the particles behavior after the robot has executed a small rotation to the right. While 10c shows the state directly after the rotation, 10d illustrates how the particles continue to follow their own trajectories, caused by the addition of 'random' noise to the angle of orientation of these particles. Figure 10e shows the state after the robot has rotated about 180 degrees and has travelled a short distance in the opposite direction as before. Figure 10f illustrates the final state after the robot has continued to travel in a constant direction.
Notice that in the end (Figure 10f) the particle cloud is spread out over a very big area, but some particles are still very close to the real position of the robot. These would be the particles that get reselected very often in the particle filters resampling step. The particles that are located very far from the robots real position have less chance of being re-selected for the output particle set and therefore would disappear over time. This also shows the importance of resampling in the particle filter, as without this step the 'incorrect' particles would stay in the set indefinitely and would have negative influence on the robot's belief.

4 Mapping

> "A map is a visual representation of an area - a symbolic depiction highlighting relationships between elements of that space such as objects, regions, and themes"[7]

Mapping is the process in which selected features of the environment are extracted, through sensor measurements, and stored in a data structure. This data structure is called a *map*. Such a map can represent the environment either three- or two-dimensionally. Although a three-dimensional map represents the environment with increased accuracy, two-dimensional maps are easier and more intuitive to generate and evaluate. Two-dimensional maps should be sufficiently accurate for most cases. But, for example, in environments where there are several navigable planes above each other (think of a skyscraper with its many levels), the use of a three-dimensional map can be beneficial.

In this thesis we will use two-dimensional maps, as the encountered environments can be accurately described by this type. To be more precise, I will make use of so-called *Occupancy Grid Maps*. For more information on this map type and the associated mapping technique, please consult Section 4.2.

4.1 Challenges

Generating maps with robots is not an easy problem to solve. This is mainly due to two properties:

Hypothesis Space: The hypothesis space for maps, in other words the space of all possible maps, is incredibly big. Even if we use discrete techniques, like an occupancy grid, to approximate this space, a map can easily consist of 10^15 variables or more. This property makes it difficult to calculate accurate probabilistic maps.

Chicken and Egg: Constructing a map of the robot's environment is relatively easy under the precondition that the robot's pose is known, as we will see in Section 4.2. It is also relatively easy to determine the robot's pose if a map of the environment is available, as we have seen in Section 3, especially 3.2.4. When neither a map exists nor the pose is known, that's when the process becomes tricky, as the robot has to construct a map and localize itself within this map at the same time. For now we will assume the poses to be known, due to the circumstances discussed in Section 1.1 .

Additionally to the properties mentioned above, other properties of the robot and the environment have an effect on the difficulty or 'hardness' of the respective mapping problem. [Thrun, Burgard, Fox 06] mentions and describes the four most important factors:

> **Size.** The larger the environment relative to the robot's perceptual range, the more difficult it is to acquire a map.
>
> **Noise in perception and actuation.** If robot sensors and actuators were noise-free, mapping would be a simple problem. The larger the noise, the more difficult the problem.

[7] http://en.wikipedia.org/wiki/Map - 01.02.2010

Perceptual ambiguity. The more frequently different places look alike, the more difficult it is to establish correspondence between different locations traversed at different points in time.

Cycles. Cycles in the environment are particularly difficult to map. If a robot just goes up and down a corridor, it can correct odometry errors incrementally when coming back. Cycles make robots return via different paths, and when closing a cycle the accumulated odometric error can be huge!

Overcoming these challenges is not an easy task. This is especially true for *perceptual ambiguity* and environmental *cycles*. As these two challenges are quite hard to overcome reliably, this thesis will focus on *size* and *noise*. Only once those obstacles have been overcome, can the other two be tackled.

4.2 Occupancy Grid Mapping

An occupancy grid is a type of map and in this section a way to construct such a map will be illustrated. The mathematical foundations in this section are based on [Thrun, Burgard, Fox 06] Most occupancy grid maps used in practice, as well as the ones that will be used in this thesis, are 2-D floor plan maps. These maps represent a slice of the Three-dimensional world, much like a constructional ground plan represents a single story of a skyscraper. Occupancy grids can be generalized to three dimensions, but the computational complexity is very high and the practical gains are not worth the expenses in most applications.

An occupancy grid map can be described as a rasterized data-structure, similar to a bitmap, where each cell/pixel possesses a certain likelihood of occupation. This value can range from zero to one and reflects the probability that the area represented by this cell is occupied. When visualizing an occupancy grid, the convention is to use greyscale images and let the occupational-likelihood define the greyscale value of the corresponding pixel. Pixels/cells with a high likelihood of occupation are darker than those with lower likelihoods. This allows easy and intuitive interpretation of such maps. Please refer to Figure 11 for a comparison between a ground plan[8] and the generated occupancy grid map.

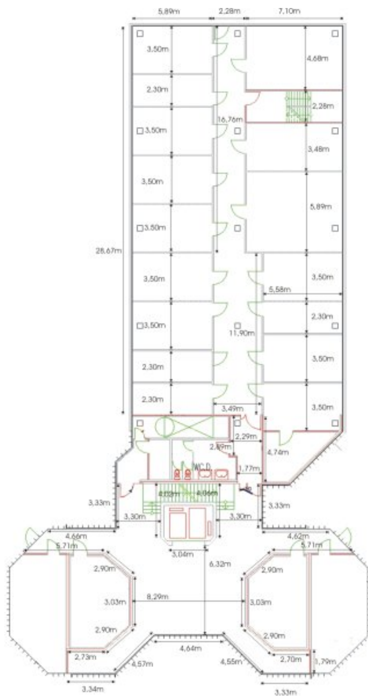
The goal of any occupancy grid mapping algorithm is to compute the posterior over maps depending on available data, as illustrated in Equation (7).

$$p(m \mid z_{1:t}, x_{1:t}) \tag{7}$$

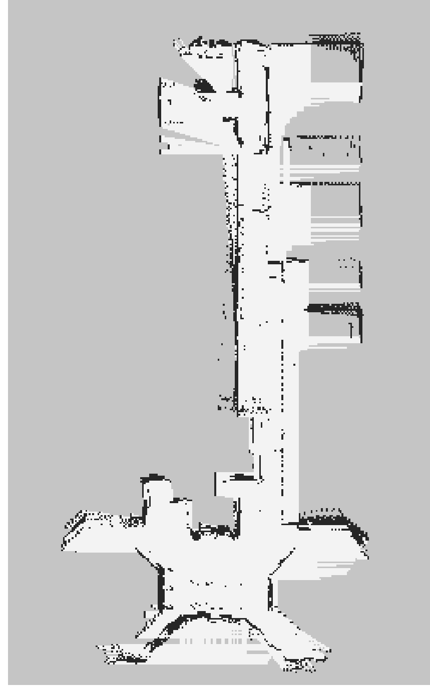
The variables used in this chapter follow the same convention as in previous chapters, m stands for the map while $z_{1:t}$ denominates the set of all measurements and $x_{1:t}$ the set of all of the robot's poses up to time t. The set of all poses $x_{1:t}$ can also be called the *path* of the robot. As the path of the robot is known, the controls $u_{1:t}$, which are responsible for the path, contain no additional useful information and are therefore omitted in this section.

Occupancy grid maps approximate the continuous space of locations. This is done by first partitioning the environment into finitely many grid cells, where $\mathbf{m}_i$ is the cell with index i.

[8]Ground plan and UT2004 map obtained from:
http://kos.informatik.uni-osnabrueck.de/download/UOSSim/UOSSim2004.html - 10.01.2011



(a) Ground plan[8]



(b) Map, partially explored

Figure 11: Comparison of ground plan and occupancy grid map of an office environment

$$m = \{\mathbf{m}_i\} \tag{8}$$

Each of these cells $\mathbf{m}_i$ has an *occupational probability* value $p(\mathbf{m}_i)$ attached to it. This value defines how probable it is that the part of the environment which is represented by this cell, is occupied by an obstacle. As with all probabilities this value can only range from 0 to 1.
The high-dimensionality of the posterior in Equation (7) is a problem. A sufficiently detailed occupancy grid map may consist of several thousand or even tens of thousands of individual cells. If we assume that each grid cell can only have one of two values (occupied or free) then the number of different maps that can be represented by a map with 10000 cells equals 2^{10000}. Therefore computing a posterior probability for every possible map is not a feasible approach.
As the occupancy grid map already partitions the continuous environment into separated smaller areas, we can break down the problem of estimating the whole map into many smaller problems. The new goal is to estimate the occupational probability of each individual cell.

$$p(\mathbf{m}_i \mid z_{1:t}, x_{1:t}) \tag{9}$$

This is convenient as it gets rid of the high-dimensional posterior present in Equation (7), but it also introduces a problem: Possible dependencies between neighboring cells cannot be represented so the posterior over a map is approximated as the product of the probabilities of all cells of this map.

$$p(m \mid z_{1:t}, x_{1:t}) = \prod_i p(\mathbf{m}_i \mid z_{1:t}, x_{1:t}) \tag{10}$$

Now that we have established the theoretical foundations of occupancy grids, it is time to take a look at the algorithm in Table 5 with which such maps can be generated. The basic functionality of this algorithm is quite simple: it looks at every cell and determines if this cell is in the perceptual field of the current measurement z_t, which means that z_t contains information about the occupancy of this cell. Cells that are in the perceptual field have their probability values updated by the result of the function **inverse_sensor_model** while the probability value of other cells remains unchanged. The function **inverse_sensor_model** implements the inverse measurement model $p(\mathbf{m}_i \mid z_t, x_t)$ and therefore calculates the likelihood of occupation of cell $\mathbf{m}_i$ dependent on the robot's current measurement data and pose. Please note that the function **inverse_sensor_model**, with special focus on laser sensors, will be closely examined in Section 4.2.2. Another noteworthy property of this algorithm is that it uses the log odds representation ($l_{t,i}$) to define the likelihood of occupancy. This enables the algorithm, to not only consider a finite number of previous occupational likelihoods, but all of them, in order to calculate the probability of occupation at the current time. Please consult Section 4.2.1 for detailed information on this representation.

```
1: Occupancy_grid_mapping({l_{t-1,i}}, x_t, z_t):
2: for all cells m_i do
3:    if m_i in the perceptual field of z_t then
4:       l_{t,i} = l_{t-1,i} + inverse_sensor_model(m_i, x_t, z_t) - l_0
5:    else
6:       l_{t,i} = l_{t-1,i}
7:    end if
8: end for
9: return {l_{t,i}}
```

Table 5: Basic occupancy grid mapping algorithm, [Thrun, Burgard, Fox 06]

4.2.1 Log Odds Ratio

By making use of the so-called *Log Odds Ratio* to update the occupational likelihoods of cells in an occupancy grid map, one is not confined to using a finite number of previous likelihoods in order to calculate the absolute value. By using the log odds representation, one can use the newly acquired occupational likelihood to update the previous value. This means that only one value, the current one, has to be stored. This stored value incorporates all earlier updates and therefore is a more complete representation of the environment than could be achieved by storing a big but finite number of previous values and calculating some kind of average.

The following formal definitions use the notation from Table 5 in order to make the explanations more intuitive. Table 5 is taken from [Thrun, Burgard, Fox 06], therefore most of the formal definitions in this chapter are also from this particular book.

$$log_odds_ratio(p(x)) = \log\left(\frac{p(x)}{1-p(x)}\right) \tag{11}$$

Equation (11) defines the log odds ratio. Now it should also become clear why it is called a ratio, because it defines the ratio between the *probability* (p) and the *complementary probability* ($1-p$) of the same event x. In order to calculate the probability of x from its log odds representation, Equation (12) can be used.

$$p(x) = 1 - \frac{1}{1 + \exp(x)} \tag{12}$$

Now that we know the basic formalisms of the log odds ratio, let us examine the definitions of the variables used in Table 5:

$$l_{t,i} = \log \frac{p(\mathbf{m}_i \mid z_{1:t}, x_{1:t})}{1 - p(\mathbf{m}_i \mid z_{1:t}, x_{1:t})} \tag{13}$$

$$l_0 = \log \frac{p(\mathbf{m}_i = 1)}{p(\mathbf{m}_i = 0)} = \log \frac{p(\mathbf{m}_i)}{1 - p(\mathbf{m}_i)} \tag{14}$$

$$\textbf{inverse_sensor_model}(\mathbf{m_i}, x_t, z_t) = \log \frac{p(\mathbf{m}_i \mid z_t, x_t)}{1 - p(\mathbf{m}_i \mid z_t, x_t)} \tag{15}$$

As illustrated in Equation (15), the value returned by the function $\textbf{inverse_sensor_model}(\mathbf{m_i}, x_t, z_t)$ in Table 5 must of course be the log odds representation of the calculated probability. Remember that the update functions in Lines 7 and 9 of Table 6, which basically implement Line 4 of Table 5, should of course also use the this representation when updating the occupational likelihoods of cells.

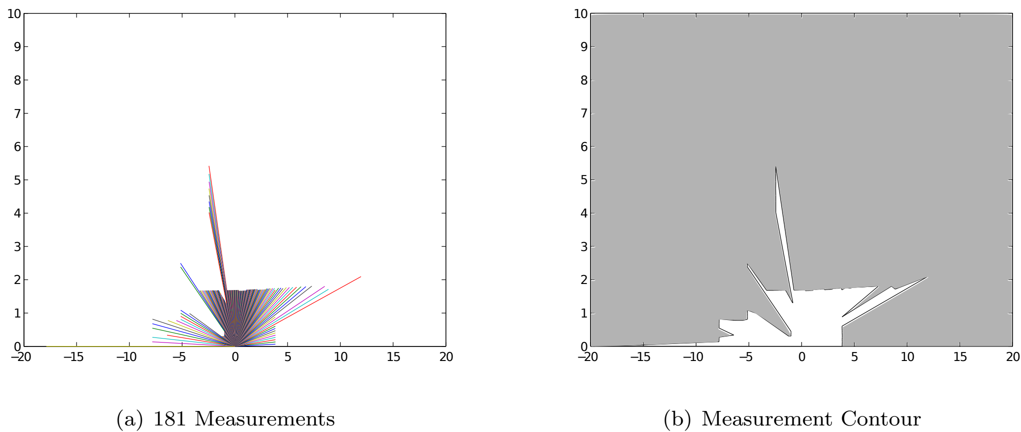
4.2.2 Inverse Laser Sensor Model

The function **inverse_sensor_model** in Line 4 of Table 5 has the task to calculate the occupational probability of a certain cell $\mathbf{m}_i$ dependent on the current measurements z_t and current pose x_t. Please note that because of the facts illustrated in Section 1.1, we will assume to have knowledge about the current pose of the robot. Conveniently USARSim, as opposed to reality, provides us with such an oracle in the form of the so-called 'Ground Truth Sensor'. This sensor can be queried by using the toolbox function *getGroundTruth* and it returns the precise pose of the robot free from any noise or error.

You might ask yourself why this function is called an *inverse* measurement model. Remember that the output of a range sensor consists of one or more single numerical values indicating the distance from the sensor to an obstacle. This value is of course dependent on the environment in which the measurement was taken, the robot's position within this environment and the way the sensor measures the distance. This means we can extract knowledge (the measurements) from the environment by using the sensor. This process also works the other way around (*inverse*), which means we can extract knowledge about the environment (the map) from the measurements through the sensor. Notice that in both cases the sensor is used to extract knowledge. In the first case it is used to conduct the measurements while its role in the second case is a little different. To be able to draw conclusions about the environment from sensor measurements, we must know exactly how the sensor works. The way the sensor works can then be modeled by software, this is called the *sensor model* or *measurement model.* This model does not only allow for simulation of the sensor, but as it defines *how* measurements arise, it can also be used to reconstruct the environment from these values. How this process works in detail will be illustrated in this section.

Figure 12: Robot in office environment

For this process to work reliably, the available sensor has to be modeled as precise and accurate as possible. For this thesis I will make use of a so-called *Laser Scanner*. This is a range sensor which emits a laser-beam, measures the time it takes the laser to reach an obstacle and uses it to calculate the distance to the obstacle. Laser scanners usually make use of more than one laser-beam to gain a broadened field-of-view (FOV). Our sensor uses 181 laser beams to take 181 measurements, angle-wise separated by 1°, which leads to a FOV of 180°. A visualization of such a measurement can be found in Figure 13a, where each measurement is represented by a line between its point of origin (sensor) and the measured distance factoring in the respective angle of the laser-beam. Please compare with Figure 12, which shows the environment and the robot's position at the time of the measurement.



(a) 181 Measurements

(b) Measurement Contour

Figure 13: Visualized laser sensor measurements, sensor-position = (0,0)

Figure 13a also illustrates the first step in processing the measurements: calculating the absolute world *coordinates* of the measurements. What information can we extract from a single distance value? The distance to an obstacle from the sensor. So if we know the world coordinates of the measurement, we know the coordinates of the obstacle. Additionally we know that the space between the sensor and the measurement is free from obstacles. This is due to the fact that laser scanners cast a ray of light to determine the distance to an obstacle. If the line that the light-ray has traveled were occupied at some point, then the measured distance would be lower, but as this is not the case, this space has to be free. This property of this sensor is illustrated in Figure 13b. Note that in this illustration, different colors indicate different likelihoods of occupation. Following the principle of occupancy grid maps darker areas are more likely to be occupied than light-colored areas.

Remember, that for each scan, a laser scanner returns many individual distance values. To calculate the coordinates of these distances relative to the sensor itself, we need to know only the angle, relative to the sensor, at which the measurement was taken. Provided the sensors vertical measurement angle is axis-parallel, we can compute the x- and x-coordinates by using the horizontal measurement angle in two simple formulas:

$$x = distance \times cos(angle) \tag{16}$$

$$y = distance \times sin(angle) \tag{17}$$

In order to compute the absolute coordinates of the measurements, we need to know where the sensor is mounted on the robot and in which direction it is facing relative to the robot. This information has to be combined with the current pose of the robot and the result is the absolute world-coordinate of each measurement. Now it is time to transfer the knowledge about measurement-coordinates and occupied/non-occupied space to the map.

Remember that an occupancy grid map is a finegrained grid of individual cells. These cells partition the continuous space of the environment into small independent areas. A value is attached to each cell, specifying the likelihood of this area being occupied by an obstacle.
In our case, a laser scanner is used to measure the distances, which it does by emitting a laser-beam and measuring how long it takes the laser to reach an obstacle. In a more abstract way, this means the sensor emits a straight line and the length of the line when its path crosses the first obstacle in its way, is the returned measurement distance.

If we break this behavior down to the absolute basics and keep in mind our goal of updating the map, this is what it boils down to: Connect two arbitrary cells of the occupancy grid map with a line. Luckily the wheel does not have to be reinvented, as there exists an algorithm which does exactly that: the *Bresenham Line Algorithm*[9] illustrated in Figure 14.

This algorithm was developed in 1962 by Jack Bresenham, who was a programmer at IBM at the time. He first published the algorithm in [Bresenham 65]. The algorithm operates on a two-dimensional rasterized data-structure, for example a computer display, occupancy grid map or any other two-dimensional array. It requires two pixels/coordinates as an input and connects these two coordinates with a line while minimizing the visual effect of so-called *Jaggies*. Not only is this algorithm easy to implement and computationally very fast, it also provides an accurate model of a straight line in a rasterized data structure and is therefore perfectly suited to model a laser scanner.

[9] http://en.wikipedia.org/wiki/Bresenham's_line_algorithm - 01.02.2011

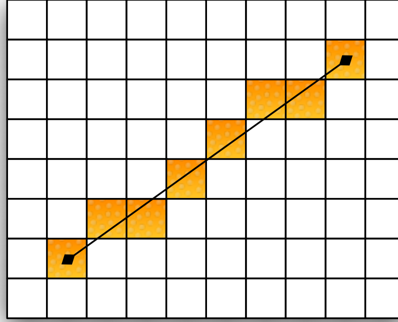


Figure 14: Bresenham's line algorithm

1: **inverse_laser_sensor_model**(x_t, z_t)**:**
2: $\mathbf{m}_{z_t} = calculateGridCell(z_t)$
3: $\mathbf{m}_{x'_t} = calculateSensorGridCell(x_t)$
4: $P_t = bresenham(\mathbf{m}_{x'_t}, \mathbf{m}_{z_t})$
5: $P_t = P_t \backslash \{\mathbf{m}_{z_t}\}$
6: **for all** cells $\mathbf{m}_i \in P_t$ **do**
7: $updateFree(\mathbf{m}_i)$
8: **end for**
9: $updateOccupied(\mathbf{m}_{z_t})$

Table 6: Inverse laser sensor model algorithm

Please note that in this algorithm, z_t denotes one single measurement value, therefore this algorithm has to be executed for all values of a laser sensor measurement. Please keep in mind that the algorithm in Table 6 does not strictly implement the inverse sensor model in Line 4 of Table 5. The function **inverse_sensor_model** in Line 4 of Table 5 calculates the occupational likelihood for a single cell while the algorithm in Table 6 calculates and updates all cells affected by a single laser beam at once. The difference in functionality is partly due to the fact that we use a laser scanner as opposed to the sonar scanners used in [Thrun, Burgard, Fox 06]. Also note that the functions in Lines 2, 3, 7 and 9 of Table 6 are very implementation specific and will therefore not be described here. For example implementations of these functions, please consult the source code accompanying this thesis.

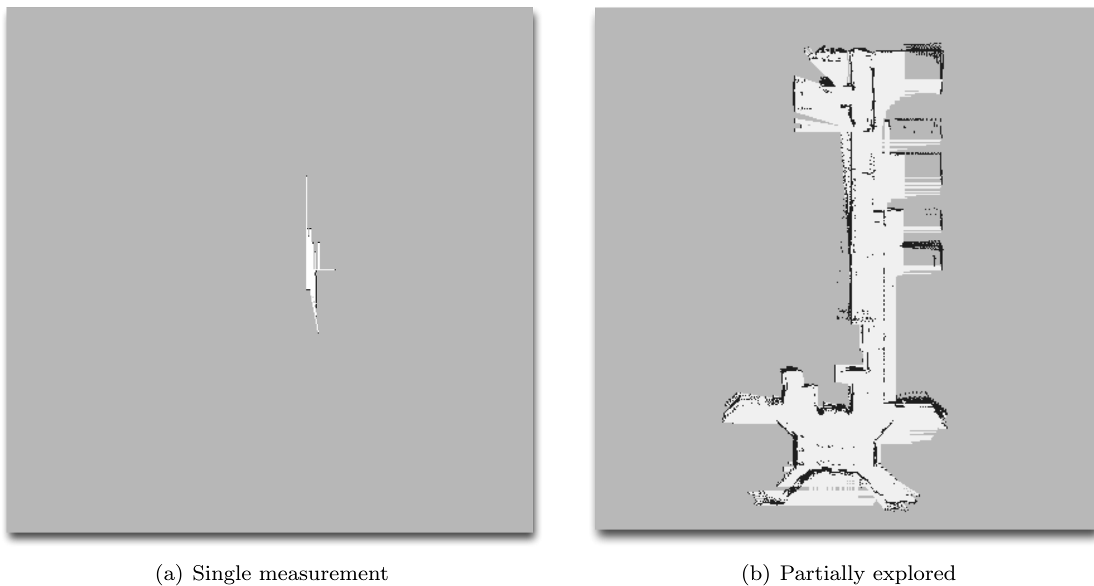
In order to update the map with knowledge from the measurements, the algorithm in Table 6 consists of three main parts:

1. In Lines 2 and 3 the grid cells representing the location of the measurement and the sensor are calculated. For z_t the containing cell is calculated directly from z_t's world-coordinates. For x_t the procedure becomes a little bit more complicated: As we know, x_t denotes the pose of the robot and the coordinates refer to the center of the robot. The

measurements, on the other hand, are obtained by a sensor which is usually not located exactly at the center of the robot. If we were to assume that x_t is the location from where the laser-beams are emitted, we could end up with the wrong starting cell for calculating the respective line/laser-beam. Therefore we need to take into account the offset between robot-center/angle and sensor-location/angle when calculating world-coordinates and grid cell of x_t.

2. Lines 4 and 5 model the laser-beam by calculating the cells of the occupancy grid which are affected by the laser-beam or, in other words, are necessary to display the line in the occupancy grid map between the cell $\mathbf{m}_{x'_t}$, representing the sensor location and the cell $\mathbf{m}_{z_t}$, representing the coordinates of measurement z_t.
   In Line 4 Bresenham's algorithm computes all cells necessary to connect the two input cells, the result is a set of cells including the start- and end-cell.
   In Line 5 the measurement cell $\mathbf{m}_{z_t}$ is removed from the set P_t. The reason for this is that a measurement in the area of a particular cell indicates that this cell is occupied, or else the laser-beam would not have hit an obstacle in this area.

3. Lines 6 to 9 form the final step: updating the occupational likelihood of each affected cell. The loop iterates through every cell in P_t and updates their likelihoods with the information that this cell is free. Line 9 makes it also clear why the cell $\mathbf{m}_{z_t}$ was previously removed from P_t: because it is the only cell which will be updated as being occupied. Please refer to Section 4.2.1 for details on updating cell likelihoods.



(a) Single measurement

(b) Partially explored

Figure 15: Laser measurements embedded in occupancy grid

The algorithm, if executed for all 181 measurement values returned by our laser sensor, produces a representation of a measurement's contour in the occupancy grid map. Figure 15a shows an occupancy grid map which was updated with the measurement illustrated in Figure 13. Once enough measurements have been taken at many different locations, the constructed map provides a good representation of the whole environment, as can bee seen in Figure 15b.

4.2.3 Learning Inverse Sensor Models

For the sake of completeness, I would like to mention that inverse measurement models can be modeled not only by implementing the sensor's attributes directly in software. It is also possible to derive such a model through the use of machine learning algorithms. This section will only provide a short introduction of how to learn measurement models, for more detailed information please consult Section 9.3 of [Thrun, Burgard, Fox 06].

Keep in mind that we want to compute $p(\mathbf{m_i}|x, z)$, the occupational likelihood of a single grid cell $\mathbf{m_i}$ depending on the robot's pose x and measurements z.

We could now use a *supervised machine learning algorithm* to train the software to approximate this expression. But for any training of learning algorithms, we need a data-set to train with. Such a set would consist of several value-triplets, each consisting of pose, measurement and correct occupational likelihood of a cell. Such triplets can be generated by executing four simple steps as described in [Thrun, Burgard, Fox 06]:

1. Sample a random map $m^{[k]}$. This map has to be sampled from $p(m)$, which can be represented by a collection of maps.

2. Sample a pose $x_t^{[k]}$ within the previously sampled map.

3. Sample a measurement $z_t^{[k]}$. This measurement sample has to be dependent on the map and the robot's pose within the map, therefore one samples from the distribution $p(z|x_t^{[k]}, m^{[k]})$. Please note that this sampling step uses a forward measurement model to simulate the measurements of the particular sensor.

4. Look up the *true* occupational likelihood value of the target cell $\mathbf{m_i}$ in m.

After these four steps have been executed, we end up with one of the desired triplets. In order to train the software, we use the pose x and the measurements z as the input of the learning algorithm. The occupational likelihood of $\mathbf{m_i}$, denoted as $occ(\mathbf{m_i})$, is the target result of the algorithm. After training a supervised learning algorithm with a sufficient amount of triplets, the software should be able to reliably approximate the expression $p(\mathbf{m}_i \mid z_{1:t}, x_{1:t})$ for any random triplet.

It is also possible to use real data acquired by the robot to train the software. In theory, this would result in more accurate training of the software as the measurement model used in Step 3 can only be an approximation of the real sensor. The big disadvantage to acquiring the data this way is, that it is more complex to accomplish. In order to collect training data with a real robot, the robot needs to operate in a known environment with a known map, additionally the used localization technique has to be very accurate. These requirements are not easy to satisfy, therefore many applications will use simulated training data, as illustrated above, for training as the expenses of using real data outweigh the advantages.

5 Implementation

Up until now, this thesis has focused on illustrating the theoretical background of certain localization and mapping techniques. However, this theoretical background could not have been compiled without concurrent implementation, testing and adaptation of the required algorithms. This section will illustrate the way in which the individual algorithms were encapsulated in functional units and implemented. Challenges that were overcome will be highlighted and interesting details of the implementation will be examined in order to provide an extensive overview of the practical implementation of the before mentioned localization and mapping techniques.

As mentioned before, the implementation was completed in the well known technical computing suite *Matlab*©. This application provides the benefit of being able to implement and test components very quickly and with minimum hassle. The availability of a toolbox, which handles the communication with USARSim, did not influence the decision to use *Matlab*© but came in handy nonetheless.

The implementation is split into two parts: *localization* and *mapping*. Each of these parts works in isolation and does not rely on the other, please consult Sections 5.3 and 5.4 for implementation details. Due to the straightforward nature of the problems to be tackled, the design of the software did not need to be very involved. Each of the two parts consists only of a couple of functions, no object orientation and no advanced design concepts were required. Please consult the files accompanying this thesis for the complete source code of this project.

5.1 Start Scripts

To begin execution of either *Localization* or *Mapping*, all that needs to be done is to make sure that USARSim is running on the system and execute the appropriate start script from within Matlab to kick off the desired process.

Localization: `ParticleFilterStartScript.m`

Mapping: `OGmappingStartScript.m`

These scripts are used to add a robot to the simulation and start either the localization- or mapping-process. As it is outlined in [Mader 10-1], each map used by USARSim has one or more recommended starting positions. By initializing the robot on one of these recommended positions has several benefits, the most important being: these spaces are guaranteed to not be occupied by any obstacles.

In order to find out the coordinates of the recommended start positions one has to send a message with the following content to USARSim: `GETSTARTPOSES`. USARSim will then respond by sending a message which contains all start poses of the current map.

The problem that presented itself, was that the toolbox does not offer any way to send a `GETSTARTPOSES` message, or any user defined messages for that matter, therefore a workaround had to be developed. I wrote a small and simple Matlab class called `NetworkInterface` which can connect to a TCP/IP socket, like the one provided by USARSim, and send and receive lines of plain text. In both start scripts this class is used to connect to USARSim, send `GETSTARTPOSES` and receive the response containing the start positions for the current map. After the response

has been received, the connection to USARSim is closed and `NetworkInterface` has fulfilled its purpose.

The next step is to extract the individual start positions from the message and store them for later use. Then the script constructs a dialog window and populates it with a list of available robots. This robot list is defined at the beginning of the script and contains only one robot at the moment. This robot is called *P2AT* and all implemented algorithms are tailored especially to this particular model. This means that robot specific information, such as sensor mount positions, are hard coded in the respective functions. Therefore it does not make a lot of sense to extend the list with additional robots at the current moment. After a robot has been selected, the user has to choose the robot's desired start position from a similar dialog. Of course the user can only choose one of the previously received start positions, thus making sure that the robot will be initialized in appropriate surroundings.

After the start position has been selected, the script uses the toolbox's `initializeRobot` function to add the robot to the simulation. Please note that the z-coordinate of the start position has to be adjusted according to the specific robot you want to use. The z-coordinate of the position relates to the center of the robot, so if you initialize it with the received value, the robot might end up submerged in the floor instead standing on its wheels on the floor. Therefore you have to adjust the start position z-coordinate by decreasing (USARSim uses an inverted Z-axis) it by half of the robot's height in order to ensure that the robot is initialized above the floor.

After the robot has been added to the simulation, the script starts the control interface for the robot as well as the respective functions to start either *Localization* or *Mapping*.

5.2 Control Interface

Figure 16: Robot control interface

In order to be able to execute localization and mapping algorithms, the robot needs to move

around the environment. As the robot has no will of its own own, we need to tell it what we want it to do. In advanced applications a robot might exhibit an intent of its own, but as this is not the case here, we will make use of a simple control interface to move the robot.

As you can see in Figure 16, the interface is fairly straightforward. The slider on the bottom adjusts the speed of the robot, as you can see it allows to adjustments in the range of $0 - 100\%$ of the robots maximum wheel spin velocity. The buttons labeled 'Forward' and 'Back' spin the wheels in the respective direction and the 'Stop' button stops all movement. The buttons 'Left' and 'Right' instruct the robot to turn to the indicated direction. Please note that the buttons 'Lights' and 'Flip' are non-functional at the moment. This is due to the same fact that lead to the need for the class `NetworkInterface` as explained in Section 5.1: the toolbox's inability to send arbitrary messages to USARSim. The toolbox currently offers no way to switch a robot's lights on and off or flip the robot to an upright position, which these, currently non-functional, buttons were intended to do. As I am sure the functionality to send arbitrary messages will be included in a future version of the toolbox, the unused buttons remain included in the interface.

The implementation of this interface consists of two files: `driveGui.fig` and `driveGui.m`. The file with extension `.fig` stores the visual appearance of the GUI including the location and visual appearance all of the buttons, sliders and other UI elements. The `.m` file on the other hand contains all executable code, such as callback functions for buttons and all other UI elements.

5.3 Localization

Figure 17 shows all components of the software as well as their organizational structure. Remember that each component is just a simple function, therefore arrows indicate that a certain function calls another function. For example, it is easy to see that `ParticleFilterStartScript` calls `DriveGui`, the robot control-interface, as well as the function `run_particleFilter`.

In short, localization works the following way:

1. `ParticleFilterStartScript` adds a new robot to the simulation, initializes the robot control-interface `DriveGUI` and starts the localization process by calling `run_particleFilter`.

2. `run_particleFilter` initializes the particle set on first execution. It also executes a loop, continuously gathering current odometry and measurement data in regular intervals. This data is then used in the function `Particle_Filter` which is called each time new data is available.

3. The function `Particle_Filter` implements the algorithm illustrated in Table 2 and calls the functions `sample_motion_model_odometry` (Table 3) to sample a pose and `weigh_particle` (Table 4) to assign an importance factor to the newly sampled pose. This is done for each particle and in the end a new set is constructed by drawing particles from the temporary set according to their importance factors.

4. The updated particle set is returned to `run_particleFilter` where it is visualized, as illustrated in Figure 10. Finally, we return to step 2 and reiterate.

The following subsections will closely examine the individual localization components. The focus

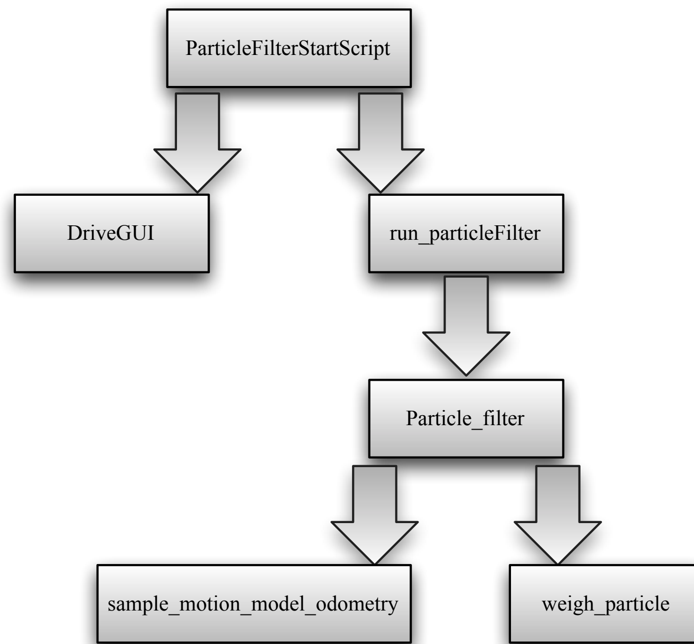


Figure 17: Localization implementation overview

will lie on the mode of operation while implementation problems and their respective solutions will be described as well as certain noteworthy code constructs.

5.3.1 run_particleFilter

After the start script `ParticleFilterStartScript` has initialized the robot on the chosen start position, it calls the function `run_particleFilter`. Its main task is to execute an infinite loop in which the robot's odometry data is obtained in regular intervals and used to update the particle set. The interval can be chosen by editing the first line of code (given in the code snippet below) within the loop and specifying the interval in terms of *seconds*. The time that the functions sleeps at the beginning of each loop iteration ensures that 2 consecutive odometry measurements are always separated by the same amount of time. But before loop execution can begin, there are other things that need to be taken care of. The detailed method of operation of this function will be examined in this section.

```
1 pause(0.2);
```

The first two instructions we encounter in the file `run_particleFilter.m` are depicted in the code section below, but what exactly do they do? The instruction in the first line is necessary to access the robot variable initialized by the toolbox. This variable can have any name, but in this project its name will always be 'rob'. This line searches in the Matlab workspace 'base', which is the standard Matlab workspace, for any variable with the name 'rob' and assigns it to

the functions internal variable rob. This is necessary because without this variable we cannot communicate with the robot. Another way would have been to pass the variable to the function as an argument. But as many functions need to communicate with the robot, it was chosen to store the respective data structure in the standard workspace so that any arbitrary function can access it when needed.

```
rob = evalin('base', 'rob');
psize = 100;
```

The second line of the code above defines the number of particles in the particle set. Of course, as this number increases, the more accurate the localization process becomes, but the time necessary for computation increases as well. Therefore it is crucial to experiment with different values to find the right balance between accuracy and computational complexity.

After these first preparations have been done, the function needs to initialize the particle set before its first loop iteration. Different ways to initialize this set have been introduced at the end of Section 3.2.2. In this implementation we initialize the set with absolute coordinates reported by the GroundTruth sensor. Alternatively one could also initialize the particles with start position coordinates, but using the GroundTruth sensor is more accurate. After the particles have been initialized, the function acquires odometry data from the robot as a starting point for the odometry motion model.

Please note that odometry data as well as GroundTruth data needs to be processed before it can be used for calculations. The problem is that the representation of θ differs between the two sensors. Typically angles in the radians format are defined in the interval $[0, 2\pi]$ with 0 representing 0° and $\pi/2\pi$ representing 180°/360°. The odometry sensor of our robot adheres to the USARSim internal representation of angles in the interval $[-\pi, \pi]$ complete with USASims inverted Z-axis. The GroundTruth sensor sticks to the standard angle representation but also needs adjustment because of USARSims inverted Z-axis. The two lines below take care of the normalization for odometry- and ground-truth data by executing the respective functions.

```
gtTheta0 = normalizeGroundTruthTheta(gt0.Orientation(3));
pose0(3) = normalizeOdometryTheta(pose0(3));
```

Now that everything has been prepared and the initial odometry data has been gathered, the function enters its main part which consists of an infinite loop. The first thing that happens is that the functions sleeps for a certain amount of time. After this waiting period another odometry data-set as well as measurement data is requested from the robot. The waiting period makes sure that two consecutive odometry data-sets are always separated by a certain amount of time. Then the function calls Particle_Filter and passes the particle set and current sensor measurement as well as current and previous odometry information to this function. After Particle_Filter returns the updated particle set, it is visualized and the current odometry-data is marked as the previous before the next iteration of the loop starts.

5.3.2 Particle_Filter

The function Particle_Filter implements the particle filter algorithm in Table 2. As the mode of operation of the particle filter was already examined in Section 3.2.2 it is not necessary to examine it again in this section. Therefore this section focuses on certain interesting

implementation details.

In order to assign an importance factor to particles as described in Section 3.2.4, a map of the current environment is required. This means that we need to load a previously generated map into memory. This and other preparations are handled by the four lines of code below.

```
og = load('OfficeMap.mat');
map = og.OGmap;
cellSize = og.cellSize;
weigh = 1;
```

In Line 1 a Matlab workspace containing a previously generated map is loaded from the hard drive. Lines 2 and 3 assign the map and the size of a cell in the map to their own variables. The cell-size is given in meters, so if a cell, which is a square, has a side length of 10 centimeters, `cellSize` will be 0.1. Please take care to always load the appropriate map for the current environment, as the weighing process will not work with a map which does not correspond to the current environment. Line 4 determines whether the weighing process for particles will be executed or not, where $weigh = 1$ leads to execution and $weigh = 0$ disables this process. After these preparations have been conducted, the function executes as defined in Table 2.

Another interesting part of function `Particle_Filter` is the implementation of Line 9 of Table 2: "draw i with probability $\propto w_t^{[i]}$". As described in Section 3.2.2, this means that particles are drawn from the temporary particle set and appended to the final set. They are drawn with replacement which means that a particle can be drawn more than one time. The probability of a certain particle to be drawn equals the particle's importance factor. The code below implements this functionality.

```
if (weigh)
    m=1;
    while (m<=length(Xt0))
        drawn = 0;
        while (drawn == 0)
            randParticle = round(rand * (length(Xt0) -1))+1;
            randNumber = rand;
            if (XtTemp{randParticle}{2} > randNumber)
                Xt{m} = XtTemp{randParticle}{1};
                drawn = 1;
            end
        end
        m=m+1;
    end
end
```

As you can see in Line 1, the whole section is only executed if weighing is activated. This is because of the fact, that when the particles are not assigned any weights, there is no point in drawing them with probabilities according to their weights. If there are no weights then every particle has the same probability of being drawn, and this does obviously not improve the final particle set.

Line 3 defines a loop where the number of iterations is defined by the size of the temporary particle set. In every iteration of the primary loop the secondary loop defined Line 5 is executed.

This loop handles the process of drawing with replacement, so let us examine it in detail:

1. In Line 6 a random integer in the interval $[1, size_of_particle_set]$ is generated. This integer, called `randParticle`, defines the particle which will be processed in this iteration of the loop.
2. Line 7 generates a random number between zero and one. This number defines the so-called 'target probability'.
3. Line 8 checks if the weight of the particle indicated by `randParticle` is higher than the target probability. If it is, then the secondary loop is completed and the particle is appended to the final particle set. If the weight is lower than the target probability, the loop starts again.

But how does this lead to the desired behavior that the probability of a particle to be drawn is dependent on the particles weight? The precondition for a particle to be appended to the final set, is that its weight must be higher than the randomly generated number. The higher the weight of a particle, the higher the chance that it is indeed higher than the random number. Therefore particles with high weights will be appended to the final set more often than particles with low weights. This leads to the desired property that the final set contains more 'good' particles, where 'good' is defined by a high weight, and therefore less 'bad' particles than the temporary set of the same size.

One disadvantage of this implementation is that it might be inefficient, especially in scenarios where there are a lot of particles and many of them have low weights. Then the loop might be executed many times before a particle is being successfully drawn from the set. In the scope of this thesis, this disadvantage did not manifest itself, even when testing the implementation with up to 1000 particles. Therefore a change was deemed unnecessary, but a more efficient way will be outlined in Section 6.4.

5.3.3 sample_motion_model_odometry

This function implements the algorithm **sample_motion_model_odometry** as given in Table 3. The implementation is very straightforward and matches the base algorithm almost completely. As the base algorithm was already examined in Section 3.2.3 it is not necessary to describe it again in this section.

One part of **sample_motion_model_odometry** worth mentioning deals with the error parameters associated with the robot's movement. These parameters are defined in the first few lines of the function, as presented in the code below.

```
1 alpha1 = 0.05; %rotational
2 alpha4 = 0.05; %rotational
3 alpha2 = 0.1 ; %translational
4 alpha3 = 0.1 ; %translational
```

Please consult Section 3.2.5 for detailed information about the specification of these error parameters.

5.3.4 weigh_Particle

This function implements the particle weighing technique as described in Section 3.2.4. Just like the function `sample_motion_model_odometry`, the implementation is very straightforward. `weigh_particle` is the function which directly implements the particle weighing algorithm defined in Table 4.

In order to accurately calculate the grid cells corresponding to a measurement value, we have to perform some preparatory calculations. Keep in mind that the location of the robot, as defined by a pose, always refers to the center of the robot. Usually sensors are not located at the dead center of the robot, so if we would calculate the measurement cells with an unprocessed pose, we might end up with the wrong results. Therefore we must take into account the offset of the sensor from the robot's center in order to ensure correct results. This action is performed at the beginning of the function as illustrated in the code snippet below.

```
1  sensor_pos_x=0.14399984;
2  sensor_pos_y=0;
3  sensor_pos_z=-0.0919999;
4  sensor_dir_x=0;
5  sensor_dir_y=0;
6  sensor_dir_z=0;
7  LaserX = particle(1)+sqrt(sensor_pos_x^2+sensor_pos_y^2)*cos(theta);
8  LaserY = particle(2)+sqrt(sensor_pos_x^2+sensor_pos_y^2)*sin(theta);
9  ocLaserPos(1) = floor(LaserX/cellSize);
10 ocLaserPos(2) = floor(LaserY/cellSize);
```

Lines 1-3 define the offset of the sensor from the robot's center. Please note that the z-coordinate is negative, this is because the Z-axis is inverted in USARSim. Lines 7 and 8 calculate the exact x- and y- coordinates of the sensor while Lines 9 and 10 calculate the grid cell coordinates corresponding to the sensors location. The sensor location's z-coordinate is not considered as we perform two dimensional mapping and therefore it has no influence on the two dimensional location of the sensor.

The sensor might also not face to the robot's front, therefore we might also take the angle-offset into account. As the laser scanner of the robot *P2AT*, which is used throughout this thesis, faces the robot's front exactly, as illustrated in Lines 4 to 6 of the above code snippet, the angle offset calculations have not been implemented in this function.

After these preparations are complete, the calculations can be executed as defined by the algorithm in Table 4.

5.4 Mapping

Figure 18 illustrates the components and their organizational structure as they are facilitated in the mapping implementation. The illustration follows the same principles as Figure 17 shown in Section 5.3.

The basic way the mapping process is performed is outlined below:

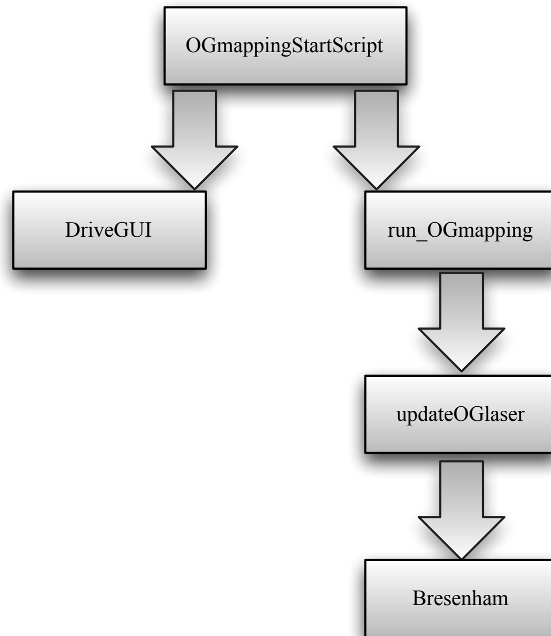


Figure 18: Mapping implementation overview

1. `OGmappingStartScript` adds a new robot to the simulation, initializes the robot control-interface `DriveGUI` and starts the mapping process by calling `run_OGmapping`.

2. On first execution `run_OGmapping` initializes the occupancy grid map as well as a second two-dimensional data structure used to store the log-odds representation of occupancy ($l_{t,i}$) for each cell. The current true pose (through GroundTruth sensor) and current measurements are gathered in regular intervals. `updateOGlaser` is called every time there is new data.

3. `updateOGlaser` implements the inverse sensor model described in Table 6 and updates the occupational likelihood of any cell affected by the measurement. Which cells are affected by a single measurement value is calculated by the function `Bresenham` as described in Section 4.2.2.

4. The updated map is returned to `run_OGmapping` where it is visualized. After that we continue with step 2 and reiterate.

The following subsections will closely examine the individual mapping components, focusing mainly on the mode of operation while describing implementation problems and challenges as well as their respective solutions.

5.4.1 run_OGmapping

Similar to the function `run_particleFilter`, the function `run_OGmapping` is responsible for executing an infinite loop in which data is gathered in regular intervalls and used to update the occupancy grid map of the environment.

In order to execute the before mentioned tasks, some preparatory steps have to be taken. More precisely, these steps deal with the initialization of the occupancy grid map and the accompanying log-odds data structure. The specific instructions can be found in the source code snippet below.

```
cellSize = 0.4;
OGmap = zeros(600);
OGmap(:) = 0.5;
logoddsmap = OGmap;
logoddsmap(:) = 0;
```

The instruction in Line 1 defines the lateral length of a cell in the occupancy grid map. The unit of measurement is *meters*, so 0.4 means that every cell represents an area of the environment equal to a square with a side length of 40 centimeters. In Line 2 the number of cells is defined. To be more precise, the map is initialized as a two dimensional array with 600 cells in each dimension. So together with the cell-size of 40 centimeters, this map can represent a square area of 240 by 240 meters. Please note that the map has to be big enough to represent the whole environment the robot can explore. Line 3 initializes the occupational likelihood of the whole map with 0.5 which means that there is a $50 : 50$ chance that the area represented by a cell is occupied by an obstacle. The value 0.5 was chosen because at this time we do not have any knowledge of the environment and therefore we cannot make any assumption if a cell is occupied or not.

Lines 4 and 5 deal with the initialization of the two dimensional data structure which will store the log-odds representation of the occupational likelihood of each grid cell. Naturally the dimensions of this data structure need to be equal to the used occupancy grid map which is why this is basically a copy of this map. Line 5 initializes the occupational likelihood values for all cells with 0. The reason this value was chosen is the same as in the initialization of the occupancy grid map. The difference in the chosen initialization value is down to the fact that the log-odds representation of $50 : 50$ is 0, as illustrated in Equation (18).

$$\log\left(\frac{0.5}{1-0.5}\right) = 0 \tag{18}$$

After these preparations have been completed, the function enters a loop which continuously updates the occupancy grid map. At first, the current laser sensor measurements are are obtained, as well as the current ‘true’ pose as reported by USARSim’s GroundTruth sensor. Then the function `updateOGlaser` is called and all necessary data is passed to it. This function returns the updated occupancy grid map as well as the accompanying log-odds map. Then the occupancy grid map is visualized and the function pauses for a certain amount of time before it enters the next iteration of the loop.

5.4.2 updateOGlaser

This function updates the occupancy grid map and the accompanying log-odds map with the information obtained from the most recent laser sensor measurements, based on the assumption that the given pose is correct. The correctness of the pose is ensured by obtaining it from the GroundTruth sensor present in USARSim. This happens in the function `run_OGmapping` as described in Section 5.4.1. The map is updated according to the inverse sensor model described in Section 4.2.2, which means that this function is an implementation of the algorithm given in Table 6.

Similar to the function `weigh_Particle`, described in Section 5.3.4, the first few lines of the function `updateOGlaser` calculate the exact position of the laser sensor which is the point of origin of all laser measurements. Afterwards the grid cell corresponding to the sensor's position is calculated.

After these preparations have been completed, the functions loops over all available measurements. A regular laser scanner returns 181 measurements, which amount to a field of view of 180°. For each measurement the following main steps are executed:

1. Check if the measurement is smaller or equal to the sensors maximum range. If it is not, it is disregarded, as it is definitely not a correct measurement.

2. Calculate the grid cell in which represents the location of the measurement.

3. Call the function `Bresenham` which implements Bresenham's line algorithm, as depicted in Figure 14, to calculate the cells which represent the line, as traveled by the laser beam, from the sensor's location to the measurement location.

4. Update the occupational likelihood of all cells calculated in the previous step. The cell of the measurement is marked as occupied while the rest is marked as unoccupied.

5.4.3 Bresenham

This function implements Bresenham's line algorithm as described in [Bresenham 65] and illustrated in Figure 14. It takes the $x-$ and $y-$coordinates of two cells and calculates the cells which represent a straight line between the two input cells. It returns an array which contains all calculated cells including the source- and destination-cell.

Please note that this function does *not* strictly implement the advanced version of this algorithm. The big advantage of the advanced version is that complex computational operations, such as multiplication, division and floating point operations, are not used and therefore the execution completes faster. The implementation in this function makes use of a single division, but the speed decrease will not be noticeable in the scope of this thesis. Noticeable slowdown could only occur in areas where occupancy grid maps of great size and unreasonably high resolution are used and the computational power of the executing machine is very low.

6 Discussion

Now that the theoretical and technical parts of this thesis have been attended to, it is time to take a step back and look at what has been achieved, what is still missing and where there is potential for improvements. The following sections will also relate the contents of this thesis to established as well as recent scientific publications. Section 6.1 will examine how much closer the described localization and mapping techniques bring us to the goal of a truly autonomous and seemingly *intelligent* robot. Sections 6.2 and 6.3 will deal with achievements and failures in the areas of localization and mapping while taking into account scientific concepts and discoveries within those areas. Finally, Section 6.4 will highlight the advantages/disadvantages of the chosen implementation of the respective techniques.

Now let us take a look at the general goal of this thesis: The particle filter as well as occupancy grid maps are well established techniques in the field of robotics and have been used, adapted and improved for some time. Even though these techniques are commonly used, obtaining detailed and in depth instructions for implementation is very hard. This is because the basic concepts can be implemented in a multitude of ways, dependent on the used hardware, field of application and many other factors. This was the motivation for this thesis, therefore the goal was the compilation of an easy to follow, in-depth instruction manual on the theoretical foundations and the direct implementation of localization and mapping algorithms. Furthermore this manual should be as abstract as possible so that it is applicable for many different robots in many areas of activity.

In retrospect I can conclude that the goal of the thesis has been achieved. Localization with particle filters and occupancy grid mapping were both discussed in detail, theoretically as well as implementation wise. Although only a single simulated robot model was used (P2AT) the instructions should be abstract enough to be applied to any robot which is equipped with an odometry- and laser-sensor. The detailed description of the implementation combined with the corresponding source code should make it easy enough to adapt it to any kind of robot. The use of Matlab as the development environment made sure that the implementation could be performed quite straightforward without having to take into account quirks and special requirements of specific programming languages and/or target platforms.

6.1 The autonomous robot

In the introduction, Section 1, it was established that when regular people think of robots, they think of very advanced, and mostly fictional, models like C3PO, R2D2 (Star Wars) and Commander Data (Star Trek), shown in Figure 19. Obviously Commander Data and C3PO exhibit more humanoid physical attributes than R2D2. Even so, people regularly characterize this little three legged fellow as *cute* and *lovable*, which indicates that they also see him as an autonomous, seemingly intelligent, living entity. What is the reason for this characterization? It certainly is not R2D2's physical appearance, as he looks more like a trashcan than a human being. R2D2's design does not even feature considerably more physical human attributes than a contemporary robot, for example the *P2AT* (Figure 19c[10]) whose simulated counterpart was used in this thesis. As the visual appearance of R2D2 is most likely not the reason for people to think of him as 'cute', the obvious explanation is that R2D2's 'cuteness' originates from his

[10]http://patrickshinzato.blogspot.com/ - 30.04.2011

(a) Commander Data (b) C3PO and R2D2 (c) P2AT[10]

Figure 19: Fictional (*a*,*b*) and non-fictional (*c*) robots

behavior.

Which specific behavioral attributes of R2D2 contribute to people seeing him more as a living being than a lifeless machine? There is the fact that this little robot moves around in a goal oriented, but somewhat erratic, manner. He is also able to relate his actions to the current environment, as he can anticipate reactions of the environment to his actions. Furthermore the little fellow can communicate through sound. Although not directly understandable by human beings, the intent of his messages is quite evident and his emotional status is clearly transported by the emitted beeping sound. Suppose one could change R2D2 so that he can not move and is completely stationary, or alternatively that he can move, but crashes into obstacles all the time. Would people still consider him 'cute' and therefore at least a little 'alive'? Maybe, but I suppose not as much as before. This indicates that intent-full and controlled movement plays a big role, but not the only one, in peoples conception of intent-full, goal oriented and seemingly intelligent behavior.

The concepts introduced in this thesis form the basis for achieving intent-full and controlled movement with an autonomous robot. After successfully implementing localization and mapping abilities on a robot platform, the robot in question possesses the most basic skills to sense and remember its environment and relates its position and movements to its surroundings. A robot which possesses these skills is still a far cry from being as lifelike as R2D2, but without localization and mapping one could not even dream of ever constructing a robot in his likeness.

Let us return to the coffee machine which was introduced in Section 1. If we would supply it with localization and mapping abilities, would people think of it as a fully fledged robot? Probably not, and there is a simple reason for that: localization and mapping do not cause intent-full behavior, but they constitute a foundation for it. These two basic techniques enable a developer to build on top of them. In order to construct a 'real' robot, a next step would be to extend, in a sense consolidate, localization and mapping so that they are not dependent on a predefined map or accurate and reliable pose information. This technique is called *Simultaneous Localization and Mapping* (SLAM) and can overcome the *chicken and egg* problem introduced in Section 1.1. This technique enables a robot to construct a map of its environment as well as locate itself within this map, all without a-priori knowledge about either the map of its location. SLAM enables a robot to navigate any known and unknown environment without the need for prior mapping or reliable position information such as the Global Positioning System (GPS). This makes a robot

suitable for many more applications than robots that rely on predefined maps or remote controls. The SLAM technique has been in use for some time, with the works of Smith and Cheeseman, [Smith, Cheeseman 86-1] and [Smith, Cheeseman 86-2], being important early contributions in this field.

But SLAM alone does not make our coffee machine have intent. To achieve something similar to intent, one has to go a step further and deal with *path planning and control*. This ability is essential if one does not want to remotely control every single action of the robot. Path planning and control enables the robot to navigate from its current position to a given destination. As the name suggests, path planning is used to plan a path, dependent on the current map of the environment, from the source to the destination. Path control is responsible to check if the robot stays on the defined path while executing the movement. If the robot deviates from the precomputed path, path control software can either bring it back to the path or compute a new one based on the robots current pose. Such a path is basically a long list of discrete motion instructions which, if executed one after another, should transport the robot to its destination along the planned path. Planning and control can not only be applied to a robot's transport system, but to any movable system on the robot, for example grappling arms. In general this is called *motion planning and control*, as it is applicable to any moveable part of the machine. According to Jean-Claude Latombe, motion planning and control are essential to enable truly autonomous robots. In [Latombe 91] he defines an autonomous robot as follows:

> "One of the ultimate goals in Robotics is to create *autonomous robots*. Such robots will accept high-level descriptions of tasks and will execute them without further human intervention. The input descriptions will specify *what* the user wants done rather than *how* to do it. The robots will be any kind of versatile mechanical device equipped with actuators and sensors under the control of a computing system."

So path planning and control enables us to construct an autonomous robot, but does this robot also have intent? It certainly possesses the intent to reach the defined destination point, but this destination was supplied by the user. Applied to our coffee maker, this would enable it to drive to a users position and make coffee for the user. So it is autonomous on two levels, moving to the destination and making coffee. The regular person would certainly describe the coffee maker as a robot, even if its intent-full behavior is based on someone else's instructions.

Giving robots their own intent has been the topic of artificial intelligence researchers all over the world. Where do we draw the line between intent supplied from outside and real internal intent? If we let a robot choose a random position as destination for its path, is the destination the robot's choice or not, as the instructions to do so were supplied from the outside? Answering this and other related questions seems to be more of a job for philosophers and psychologists than a computer scientist, therefore I will not speculate on this subject. The further developments in artificial intelligence improve, the closer mankind seems to get to its goal of being a creator of intelligence, but there is still a long way to go. The more we learn about artificial intelligence, the more the way we look at our own intelligence changes. Some have even speculated that the brain might be a computer and our mind is simply a program running on this computer. While some reject this notion, such as John R. Searle in [Searle 90] and [Searle 04], I myself have indicated, in [Mader 10-2], that it is theoretically possible to write a program that has the same abilities as the mind. This of course leads to a question with which I would like to conclude the discussion about artificial 'intelligence'. The question is, if such a program can be genuinely intelligent or if it simply mimics intelligence, as established in [Marr 77]:

> "The problem is that studies–particularly of natural language understanding, problem-solving, or the structure of memory–can easily degenerate into the writing of programs that do no more than mimic in an unenlightening way some small aspect of human performance."

If we look at the subjects discussed in this thesis, we can conclude that localization and mapping are very important foundations for any further work involving moving robots. The two techniques, by themselves, do not constitute an autonomous robot, but without these foundations the construction of an autonomous robot is almost impossible.

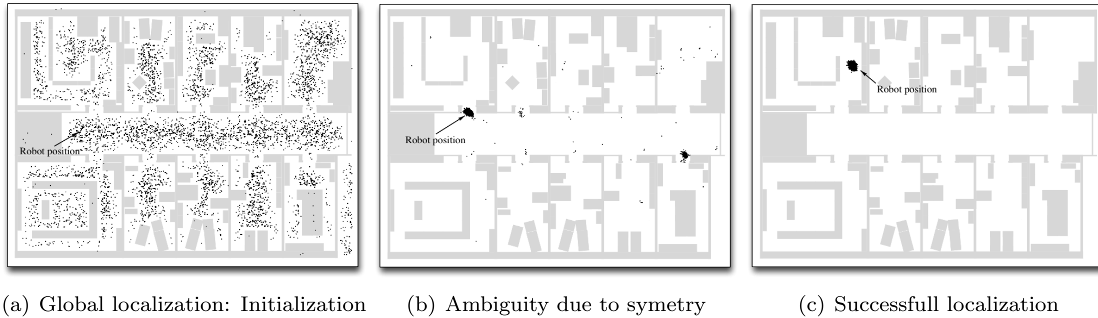
6.2 Localization

The first concept examined in this thesis is called *localization.* It deals with the problem of keeping track of one's position in an arbitrary environment. This activity relies on a robot's pose information $x_{1:t}$ as well as sensor data $z_{1:t}$ to estimate the 'real' pose of the robot. The pose can only be estimated and cannot be discretely computed as the uncertainty, which is inherent in every sensor and actuator, needs to be taken into account. Section 3 introduced a basic way to deal with this uncertainty through *probability density functions* (PDFs) while Section 3.1 introduced basic concepts which are vital before one can further examine the field of robot localization. In Section 3.2 the so-called particle filter was presented after examining the underlying Bayes filter.

The localization technique examined in this thesis can best be described as *position tracking*, as it initializes all particles at the known start position of the robot and then uses them to track the robots position. In our case, the start position is known as the robot is initialized on one of the recommended start positions provided by USARSim. If no such facility is available, $(0, 0, 0)$ can be assumed to be the start pose of the robot.

The described technique is not limited to position tracking but can also be used to solve the *global localization* problem, illustrated in Figure 20. This problem arises when one has no knowledge of where the robot's start position is located in relation to an environment. Then the particles would be initialized as a uniform distribution over all possible poses within the environment. The particle filter algorithm would then remove the unlikely particles in the course of time through *resampling.* After some time one would end up with most particles, in the optimal case all of them, distributed around the robot's real location.

The examined technique yields good results in a multitude of test environments. There is one environment type in which localization might not work satisfactory. These are big open planes in which no obstacles are present. This means that there is no sensor data which could show the distance to obstacles. This means that the particles can not be assigned an importance factor and the resampling step could not resample based on the weights of the particles. In case of such an environment global localization can not be performed reliably. Location tracking will still work but the particle filter will behave as if resampling was disabled and the localization would become increasingly inaccurate during time. This is not only a problem for the examined localization technique but for every such technique. Localization takes the uncertainty of robots into account and tries to counteract it by using sensor measurements. If no useful information can be extracted from said measurements, then the algorithm can not counteract the increasing uncertainty.



(a) Global localization: Initialization (b) Ambiguity due to symetry (c) Successfull localization

Figure 20: The global localization problem solved by Monte-Carlo-Localization, [Thrun, Burgard, Fox, Dellaert 99]

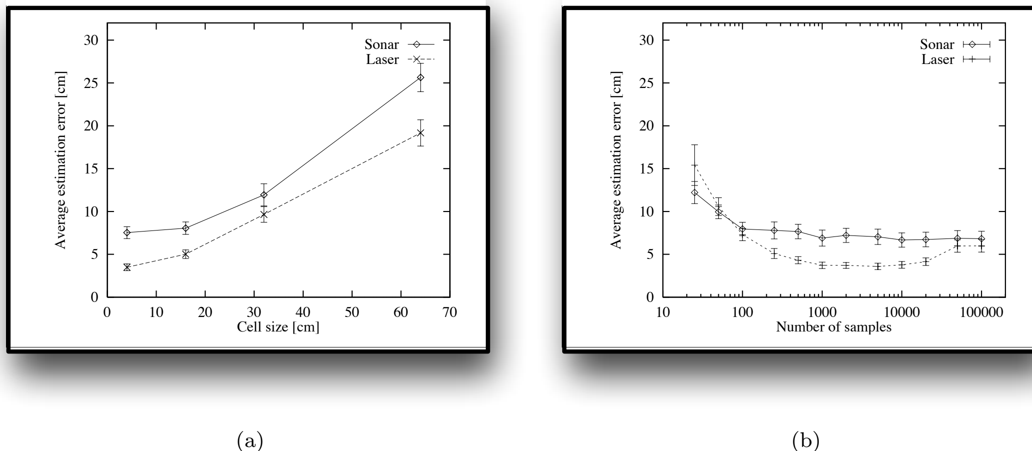
The localization technique described in this thesis, although it works sufficiently in most environments, is still a simplistic approach with lots of potential for optimization. [Thrun, Burgard, Fox, Dellaert 99] introduces an advanced localization technique called *Monte Carlo Localization* (MCL). Monte Carlo localization, through revised mathematical concepts, provides significantly increased accuracy for localization while dynamically adjusting the number of samples to the current situation and therefore reducing the computational complexity if possible. These two theoretical advantages were confirmed during real world testing in [Thrun, Burgard, Fox, Dellaert 99] and the results evaluated as follows:

1. MCL yields significantly more accurate localization results than the most accurate previous Markov localization algorithm, while consuming an order of magnitude less memory and computational resources. In some cases, MCL reliably localizes the robot whereas previous methods fail.

2. By and large, adaptive sampling performs equally well as MCL with fixed sample sets. In scenarios involving a large range of different uncertainties (global vs. local), however, adaptive sampling is superior to fixed sample sizes.

The dynamic adjustment of the sample size, or in other words the number of particles, depends on the weights of all particles. If the average weight is very high, then one can conclude that the localization process is currently very accurate and therefore one can reduce the number of particles. If the average weight of the particles drops, then one can conclude that the current accuracy is lower and therefore more particles will be used in order to compensate. This behavior reduces unnecessary computations while being as accurate as using a fixed sample size.

Figure 21 shows the result of a comparison between grid-based localization techniques and MCL in terms of accuracy. The localization technique used in this thesis is grid based as it uses an occupancy grid map to assign weights to particles and eventually resample the particle set. The illustrations show that the minimal error of MCL is not significantly lower than the one of grid-based techniques. However, to achieve a very low error with grid-based localization, one has to reduce the size of individual cells to unreasonably small values (< 5 cm as illustrated in Figure 21a). MCL exhibits a very different behavior. It's accuracy stays very much constant with varying numbers of samples (particles) once a sample size threshold has been exceeded. Figure 21b shows that the error of sonar-based MCL stays constant from sample sizes of 100 up to 100000 while laser-based MCL exhibits lower errors than sonar-based and achieves constant



(a) (b)

Figure 21: Accuracy of grid-based localization using different resolutions (a) and MCL (b) for different sample numbers (logarithmic scale), [Thrun, Burgard, Fox, Dellaert 99]

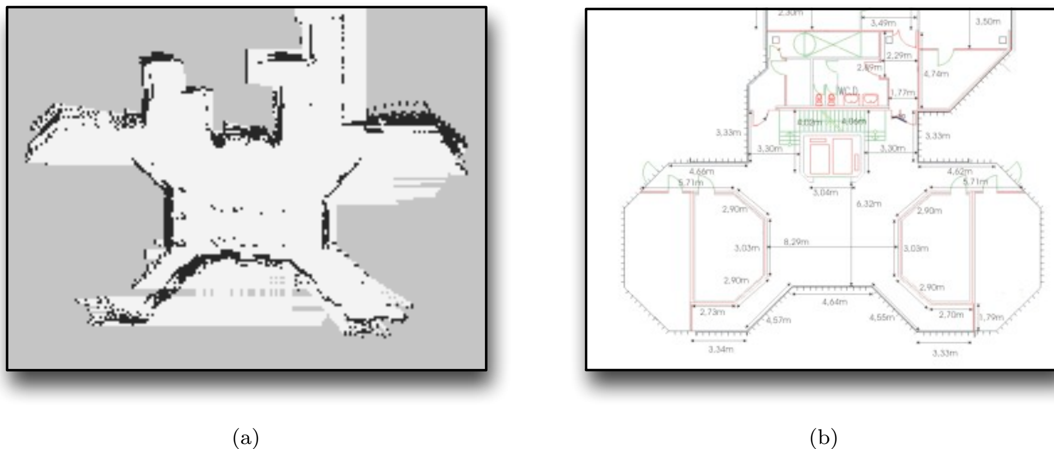
accuracy between 1000 and 10000 samples.

In the end one must admit that, when compared to Section 3, there are more accurate and computational less intensive localizations techniques in existence. These techniques usually facilitate some of the same concepts that were introduced in Section 3, as for example MCL also uses a particle filter. The localization technique examined in this thesis introduces fundamental concepts on which later localization algorithms improvements were built, therefore the understanding of those basic concepts is essential for anyone feeling the urge to dive deeper into the field of robot localization.

6.3 Mapping

The second concept which has been examined in this thesis is *mapping*, occupancy grid mapping to be precise. This mapping technique was developed at the end of the 1980s, with Alberto Elfes, [Elfes 87], being one of the pioneers. Occupancy grid mapping breaks down the problem of estimating the posterior probability over maps $p(m|z_{1:t}, x_{1:t})$ into smaller problems which are easier to solve. Basically the technique partitions the environment into a finite number of square cells and computes the probability of occupation $p(\mathbf{m_i}|z_{1:t}, x_{1:t})$ for each cell. As you can see, in both cases the resulting probability is dependent on $z_{1:t}$, which represents all of the robot's sensor measurements up to the current time, and $x_{1:t}$, which represents all of the robot's poses up to the current time.

How successful is the mapping technique described in Section 4.2 in constructing a map of the environment? If one re-examines Figure 11 and compares the floor plan of the area with the constructed map, then one will realize that the implemented technique, see Section 5.4, is able to construct a quite accurate map of the environment. Its accuracy becomes even more evident when the constructed map is used to assign weights to particles, as defined in Section 3.2.4. The assignment of a weight, or an importance factor, to a particle is based on a comparison of current



(a) (b)

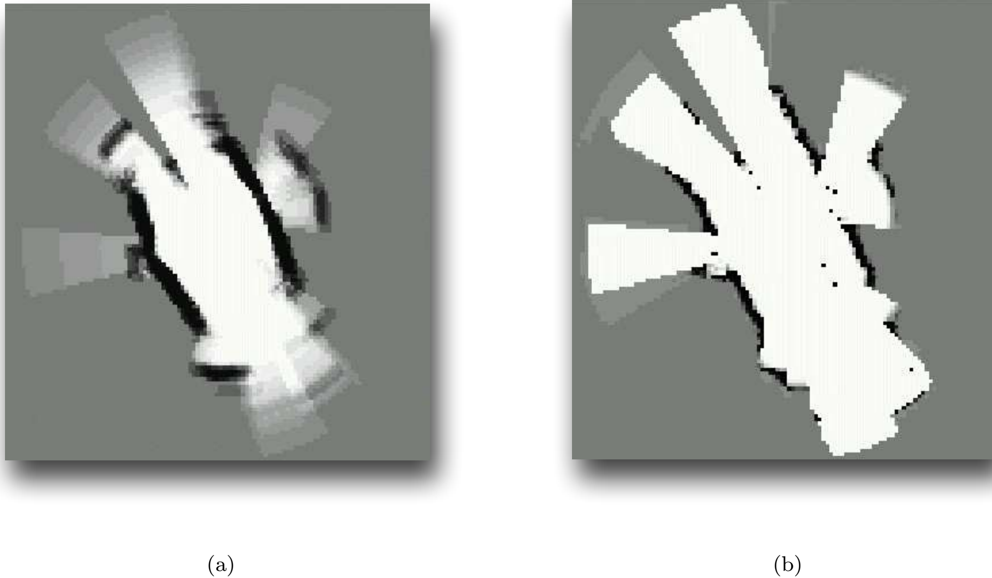
Figure 22: The occupancy grid map (*a*) describes unreachable space as free (right and bottom), compare the corresponding ground plan (*b*)

measurements to a given map. If we suppose that there are no calculation and/or sensor errors, then the success of the weighing process is dependent on the accuracy of the used map. In the performed tests, bad particles were consistently assigned low weights while good particles were consistently assigned high weights. This is a further argument for the accuracy of the technique in question.

As accurate as it is, the used technique is still quite simplistic when compared with advanced versions which have emerged in the past. Let us take a short look at the advanced occupancy grid mapping technique described in [Thrun 03], concentrating on highlighting the advantages it has over the one used in this thesis.

A well known problem of the mapping algorithm described in Section 4.2, as well as of simplistic approaches, is that they often produce maps which deviate from the present environment, as can be seen in Figure 22. This happens predominantly in environments which are very cluttered. A common example for a problematic situation is a moving robot passing by an open doorway. In such environments the inadequateness of the algorithm can manifest itself in the final map as such: open doorways appear as occupied space and therefore impassable although they are in fact not. The problem is that multiple measurements can interfere with one another and produce wrong results after being transcribed into the grid map. The basic problem lies not so much with the measurements as with the inverse sensor model, Section 4.2.2, used by the algorithm. It does not recognize dependencies between individual neighboring cells, which leads to the undesired behavior. The solution proposed in [Thrun 03] is to use a forward sensor model instead. This model takes cell dependencies into account and can therefore produce a more accurate map. See Figure 23 for a comparison between the simplistic and improved technique. Please note that the measurements were obtained from a sonar sensor which explains the measurements cones instead of the straight lines produced by a laser sensor. [Thrun 03] also mentions that laser sensors are less prone to suffer from the described problem of measurement interference due to the focused nature of the laser beam.

Comparing the resulting maps of the algorithm introduced in this thesis to those generated



(a) (b)

Figure 23: Comparison of regular occupancy grid mapping (a) and the advanced technique (b) introduced in [Thrun 03]. Both figures are taken from [Thrun 03]

by other simplistic approaches, one can conclude that the described technique offers accurate enough mapping capabilities for many applications. In case additional accuracy is needed for specific applications, I suggest to use a more advanced technique such as the one described in [Thrun 03].

6.4 Implementation

The implementation of the before mentioned localization- and mapping techniques was already discussed in Section 5, therefore this section will focus not on implementation details, but on possible improvements instead.

As discussed in Section 5, the implementation was very straightforward, implementing the algorithms introduced in the preceding sections. The first shortcoming of the current implementation comes to mind when one executes one of the start scripts. The following dialog boxes include a list of robots to choose from, but there is only one robot present in the list. This is the USARSim robot $P2AT$ and the reason for him being the only robot in the list is that its properties are hard coded within the program where needed. For example, the calculation of the sensor position based on the current pose depends on hard coded dimensions which correspond to $P2AT$. A welcome improvement would be to dynamically load these values depending on the chosen robot. The values for different robots could be stored in an XML file, a format for such an XML file was introduced in [Mader 10-1]. This would make it possible to easily provide the user with several robots to choose from while making it easy to add new robots.

The second potential improvement that comes to mind concerns the resampling step of the

particle filter. As discussed in Section 5.3.2, when the average weight of particles is very low the current implementation could execute a loop very often before a particle would be drawn from the temporary particle set in order to be added to the final set. This could be avoided through a change in the sampling process. Instead of drawing a target weight and then drawing particles until a particle's weight is higher than the target, one could add up the weights of all particles and draw a random number from the interval $[0, n]$, where $n = sum_of_weights$. As this interval is based on the sum of all particle weights, if one adds the weights in a specific order, one can assign a sub-interval to each particle. By looking up in which particle's sub-interval the randomly drawn value lies, one can determine which particle to add to the final set. The drawing process is dependent on the weight of a particle as the weight defines the size of the sub-interval. So the bigger a particle's weight is, the bigger it's sub-interval becomes and therefore the chance of being drawn is increased. The great advantage over the current implementation is, that a particle gets selected every time, therefore eliminating the problem in the current implementation.

Besides the before mentioned potential improvements, one can conclude that the implementation of the discussed localization- and mapping techniques runs stably and is computationally not too expensive. Although the implementation is simplistic, it shows an easily understandable way to implement localization and mapping in a basic form. If one wants to further investigate robot localization and mapping, then knowledge of the concepts introduced in this thesis is crucial.

7 Acknowledgements

First of all, I would like to take the time to thank my thesis supervisors at the Radboud University Nijmegen, Dr. Marcel van Gerven (Donders Institute for Brain, Cognition and Behaviour) and Dr. Perry Groot (Institute for Computing and Information Sciences), for their continuous assistance and sponsorship during the course of this project.

I would also like to thank Dr. Gerald Steinbauer from the Institute for Software-Technology (IST) at Graz University Of Technology for introducing me to the field of robotics and supervising my bachelor project.

Last but not least, I would like to thank my family, my friends and especially my girlfriend Anne for their support and endurance.

References

[Mader 10-1] Mader, B., *First Steps with USARSim*, 2010, Graz University of Technology, Austria

[Mader 10-2] Mader, B., *Software for the Brain*, 2010, Radboud University Nijmegen, The Netherlands

[Thrun, Burgard, Fox 06] Thrun S., Burgard W., Fox D., *Probabilistic Robotics*, 2006, Massachusetts Institute of Technology, USA

[Wang 05] Wang J., *USARSim - A Game-based Simulation of the NIST Reference Arenas*, 2005, University of Freiburg, Germany

[Searle 04] Searle J.R., *Is the Brain a Digital Computer?*, 2004, University of Southampton, UK

[Thrun 03] Thrun S., *Learning occupancy grid maps with forward sensor models*, 2003, Autonomous robots, Springer

[Thrun, Burgard, Fox, Dellaert 99] Thrun S., Burgard W., Fox D., Dellaert F., *Monte Carlo Localization: Efficient Position Estimation for Mobile Robots*, 1999, Proceedings of the National Conference on Artificial Intelligence (AAAI-99), Orlando, Florida, USA

[Casella, Berger 02] Casella G., Berger R., *Statistical inference (2nd edition)*, 2002, Duxbury

[Latombe 91] Latombe J.C., *Robot Motion Planning*, 1991, Kluwer Academic Publishers

[Searle 90] Searle J.R., *Is the Brain's Mind a computer program?*, 1990, Scientific American, January Issue p.26-31

[Elfes 87] Elfes A., *Using Occupancy Grids for Mobile Robot Perception and Navigation*, 1987, Carnegie Mellon University, USA

[Smith, Cheeseman 86-1] Smith R.C., Cheeseman P., *On the Representation and Estimation of Spatial Uncertainty*, 1986, The International Journal of Robotics Research, vol. 5 no. 4 p.56-68

[Smith, Cheeseman 86-2] Smith R.C., Cheeseman P., *Estimating Uncertain Spatial Relationships in Robotics*, 1986, Proceedings of the Second Annual Conference on Uncertainty in Artificial Intelligence, University of Pennsylvania, USA

[Marr 77] Marr D., *Artificial intelligence - A Personal View*, 1977, Massachusetts Institute of Technology, USA

[Bresenham 65] Bresenham J. E., *Algorithm for computer control of a digital plotter*, 1965, IBM Systems Journal, vol.4 p.25-30

A USARSim Installation

A.1 Unreal Tournament 2004 Installation

Installing UT2004 on Windows or Mac OSX is as simple as running the setup program and specifying the desired location of the program folder. Installation under Linux requires some additional steps which are detailed below:

Requirements:

- Unreal Tournament 2004 6CD Version with included Linux installer
- Activated and up to date 3D drivers for the graphics card
- libstdc++5

Start a Terminal and type `sudo bash` and type in your password when requested. Next type `export SETUP_CDROM=/media/cdrom` (you have to replace `/media/cdrom` with the appropriate path to your optical drive that contains the UT2004 Install CD).
Next you have to run the linux installer on the cd, type in the absolute path to the installer while you are in a directory not located on the optical drive, otherwise you will not be able to change the CDs when the installer requests it. So type `/media/cdrom/linux-installer.sh` and follow the instructions to install the game.
To simplify upcoming steps in case you have not installed UT2004 in your home folder we will change the owner of the UT2004 directory and all files in it. Type `chown -R PATH/ut2004` (change `PATH` to the path of your UT2004 installation). Download the latest UT2004 linux Patch (3369.2) and install it by copying the files to their respective locations in your UT2004 directory. You can now start the game by executing `PATH/ut2004/ut2004`

Potential Problems:
There is the possibility that your monitor does not support the standard resolution that UT2004 uses. To change the resolution from outside the game you have to open the file `~/.ut2004/System/ut2004.ini`, there you can specify your desired resolution under the section `[SDLDrv.SDLClient]`.

A.2 USARSim Installation

A.2.1 Windows

After installing UT2004 the installation of USARSim under Windows requires only 2 simple steps:

- Download the `.msi` installer version of USARSim
- Run the setup program

To run USARSim simply execute it from the start menu.

A.2.2 Linux

Download the newest USARSim version from the USARSim page on `www.sourceforge.com` and copy all the files to their specified locations in your UT2004 directory (from now on referred to as `$UT2004`).
Some files (*.uc) will need to be compiled before USARSim can be started, currently the compile process is only supported under Windows, therefore you will need to install UT2004 and USARSim on a Windows system as detailed in Section A.2.1. Then compile USARSim by executing the script `$UT2004/System/make.bat`, afterwards locate the compiled files (*.u) and copy them to the `$UT2004/System/` folder of your Linux installation of USARSim. Please note that the `/System/` directory with the customizable `.ini` files is stored at `~/.ut2004/System/`.
Download the maps you want to use from sourceforge.net and install them in `$UT2004/Maps/` and elevate their permissions to allow write- and execute-access..
To start USARSim execute the following command (`map_name` has to be changed to the name of the desired map and `$UT2004` to your UT2004 directory):

```
start $UT2004/ut2004 map_name?game=USARBot.USARDeathMatch?spectatoronly=1?
TimeLimit=0?quickstart=true -ini=usarsim.ini
```

A.2.3 Mac OS X

Installation and execution of USARSim is almost the same as on Linux. One difference is that the UT2004 root directory is the application package which is usually located in the applications folder. Unless you move it to another directory `$UT2004` should be:
`/Applications/Unreal Tournament 2004.app/`

Please note that additional maps you might want to add to USARSim have to be stored in `/Applications/Unreal Tournament 2004.app/Maps` and their permissions need to be changed to allow write- and execute-access. User-dependent configurations and files are located in a different directory than on Linux systems. OS X keeps this directory stored at the following location:
`~/Library/Application Support/Unreal Tournament 2004/System/`