FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Integrating probabilistic information of dynamic environment into maps for enhanced action planning

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Stephan Brugger
Matrikelnummer 0825250

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: o.Univ.-Prof. Dipl.-Ing. Dr.rer.nat Radu Grosu
Mitwirkung: Univ.-Ass. Dipl.-Ing. Oliver Höftberger

Wien, 20.01.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Integrating probabilistic information of dynamic environment into maps for enhanced action planning

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Stephan Brugger
Registration Number 0825250

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: o.Univ.-Prof. Dipl.-Ing. Dr.rer.nat Radu Grosu
Assistance: Univ.-Ass. Dipl.-Ing. Oliver Höftberger

Vienna, 20.01.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Stephan Brugger
Ochsenharing 12, 5163 Mattsee

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

First of all I want to thank the advisor of my thesis o.Univ.-Prof. Dipl.-Ing. Dr.rer.nat Radu Grosu for offering the possibility to do this thesis in such an interesting field of application and his good advices during the work.

Thanks to Univ.-Ass. Dipl.-Ing. Oliver Höftberger for many discussions and meetings, sharing his expertise in the field of robotics. He provided me with good advices and positive criticism whenever I needed them.

I also have to thank the stuff at the Department of Computer Engineering and the Vienna University of Technology for offering the possibility to do my experiments with a state of the art scientific robot.

Thanks to my girlfriend Kathrin for her patience and support during my work on this thesis, as well as the proof-reading. Sharing time with her brought the necessary balance during this thesis and she encouraged me in frustrating moments to keep making progress.

Furthermore I am very grateful to my father for proof-reading my thesis, his valuable comments and help to improve this work.

Thanks to all student colleagues who supported me during my studies and became friends during the last years.

Last but not least many thanks to my parents, who made my technical education in school as well as my studies at the Vienna University of Technology possible. They always supported me in anything during my whole lifetime and always trusted in me.

Abstract

Action planning of robots and autonomous vehicles is typically based on maps of their environment. These maps often only contain information about the static environment and ignore dynamic entities. Information about dynamic entities, like moving objects, in the environment can help to improve action planning in robotics.

The basics of autonomous navigation such as localisation, mapping and tracking are covered. State of the art algorithms for mapping such as particle filtering and Simultaneous Localization and Mapping (SLAM) are evaluated. An approach to gather probabilistic and dynamic information about the dynamic environment is introduced within this thesis. All entities within a map are probabilistic, containing a value about the probability of their existence and a change rate. Dynamic behaviour at certain locations will cause dynamic areas to be created. In addition these new information need to be saved somehow within the map. A technique to save this information, as well as to reuse existing maps and extend them, is presented. This thesis is all about gathering information, evaluating this information regarding its dynamic aspect, generating dynamic areas and store probabilistic and dynamic information in maps.

A software module, which is designed to gather and process change rates about the dynamic environment, was developed. This module in cooperation with other modules and a robotic software framework is able to perform dynamic mapping while autonomously driving to a certain destination. A small robot, equipped with a laser scanner which provides information about the environment, was used to experimentally examine the behaviour of the dynamic mapping software module. These experiments are evaluated and demonstrate the approach of generating maps with information about the dynamic environment.

Kurzfassung

Das Planen von Aktionen und Handlungen von Robotern und autonomen Fahrzeugen basiert oft auf Karten ihrer unmittelbaren Umgebung. Diese Karten enthalten oft nur Informationen betreffend der statischen Umgebung und ignorieren dynamische Aspekte. Jedoch können Informationen von dynamischen Aspekten, zumeist sich bewegende Objekte in der Umgebung, die Planung von Aktionen des Roboters verbessern.

Die Grundlagen von wesentlichen Themen im Bereich der Kartierung, wie beispielsweise Lokalisation, Kartenerstellung und das Verfolgen von Objekten, werden behandelt. Themenrelevante Algorithmen wie Particle Filtering und SLAM werden untersucht. Ein Ansatz zur Beschaffung und Verarbeitung der Informationen von Änderungsraten der dynamischen Umgebung wird in dieser Arbeit vorgestellt. Alle Objekte in einer Karte enthalten Werte bezüglich der Wahrscheinlichkeit ihrer Existenz und ihrer Änderungrate. Durch dynamisches Verhalten an bestimmten Positionen entstehen dynamische Zonen. Diese neuen Informationen müssen in geeigneter Weise in der Karte gespeichert werden. Daher wird sowohl eine Technik vorgestellt diese Informationen zu speichern, als auch bereits existierende Karten zu verwenden und diese zu erweitern. Der Hauptteil dieser Arbeit ist die Informationsbeschaffung, die Bewertung dieser Informationen bezüglich ihres dynamischen Aspekts, die Erstellung von dynamischen Zonen und die Speicherung dieser wahrscheinlichkeitsbehafteten und dynamischen Informationen in Karten.

Ein Softwaremodul, das designed wurde um Änderungsraten der dynamischen Umgebung zu beschaffen und zu verarbeiten, wurde entwickelt. Dieses Modul führt dynamische Kartenerzeugung in Mithilfe mit anderen Modulen und eines Roboter Softwareframeworks durch, während es autonom auf ein festgelegtes Ziel zusteuert. Ein kleiner, mit einem Laser Scanner welcher die Informationen der Umgebung liefert, bestückter Roboter wurde verwendet, um das Verhalten des dynamischen Kartenerzeugungsmodules experimentell zu testen. Diese Experimente wurden ausgewertet und zeigen, dass Karten mit Informationen der dynamischen Umgebung erfolgreich erzeugt werden können.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The first question which comes into my mind when we are talking about robots is why do we need robots at all. Humans are a long time on earth and did not have robots. Why shall we need them now? And yes we can live without them, but they can help us to make our life easier, smarter and maybe even funnier.

Robots can help us in dangerous environments, for instance in nuclear contaminated areas like the damaged Fukushima nuclear plant. They can perform measurements and simple tasks and can get closer to the nuclear reactor as humans are able to.

Another example is assisted living for handicapped people, who can't live without help. Robots could help them performing daily tasks like picking up things from the floor, cleaning or preparation of food. These actions a robot can do could be enough that a human can stay in his own home.

A third example is driving on highways, which is sometimes boring and just a routine task. This task can be done for us by an intelligent autonomous car. We will just have to tell the car, where we want to go and need not concentrate on the street any more but can do other things like reading a book.

There are many other cases, where robots respectively intelligent autonomous cars, which I also consider as robot[1], are able to help us in our daily life. For most of the currently available robotic applications humans play an important role in programming (e.g. industrial robots) or controlling the robot (e.g. rescue robots). Future robotic systems will incorporate more autonomy. In a majority of cases this requires that the robot:

1. *has knowledge about its environment* (a map)
2. *can localise itself within that environment* (finding the position of the robot in the map)

[1]According to the Oxford dictionary [39]: "*a machine capable of carrying out a complex series of actions automatically, especially one programmable by a computer*"

This thesis will concentrate on both of these problems.

1.2 Problem statement

Typically maps are static while our world is dynamic - humans walk around, doors are opened and closed, and cars drive around. Integrating information about the probability and change rate of dynamic entities in the map enables optimized action planning and localisation. For instance, the robot could choose its path based on the probability and the change rate of a door being open or closed. Another use case would be to determine free parking spaces in a city considering time, date and parking costs of a district and guiding cars to spots with a higher probability of a free parking lot. Cars equipped with sensors could maintain a globally shared map providing information about free parking spaces in a city. Beside the mentioned examples maps including information about dynamic entities could be beneficial for many other applications.

This work focusses on a conceptual model describing how and which information, gathered from the environment, can be used to calculate the probability and change rate of dynamic objects. Furthermore an algorithm for efficiently processing this information, storing and using it will be provided.

These concepts will be implemented as a module for Robot Operating System (ROS) [9], which allows to update dynamic regions within the map and store additional information to it. This module will handle the dynamic behaviour of the environment in contrast to the static maps currently provided. Dynamic objects are going to be evaluated and a probability of their presence as well as a change rate will be stored. Mapping and planning experiments will help to understand how this data can be used most efficiently.

1.3 Methodological approach

First of all the existing literature about autonomous robots and in particular mapping and planning algorithms are studied, and hereby the capabilities of current technology is analysed. These studies also include online lectures like [40], where the cutting edge technology for modern autonomous robotic systems is explained.

Robot Operating System (ROS) as fundamental system is well known and heavily used in scientific robotic projects, so there already exist different modules for common tasks like controlling the robot by a keyboard or simulating the robot. Hence different driving, controlling and collision avoidance modules and algorithms need to be evaluated, optimized and tested.

The main part of the thesis is the creation of maps containing dynamic information. Gathering information of dynamic objects and storing them with the map is the basis for making the decisions of the robot smarter. To obtain this, the existing Simultaneous Localization and Mapping (SLAM) algorithm needs to be analysed and adapted such that dynamic objects are indicated in the map. This adapted SLAM algorithm is implemented as ROS module. Experiments of different other innovative use cases of dynamic mapping with statistic background is also presented and evaluated in the thesis. The analysis of the results will help to improve the algorithm.

1.4 Structure of work

First of all basic concepts will be described in chapter 2 to avoid any ambiguity in the understanding of different terms and make the topic of this thesis clear. Afterwards the state of the art of map generation will be discussed in chapter 3. It will be shown, which types of maps can be distinguished, how localisation and tracking is done and which requirements exist for a robotic system to perform these tasks.

After that in chapter 4, the theoretical background of identification of dynamic areas in maps is discussed, as well as which problems need to be solved and how Robot Operating System (ROS) can support in robotics. In this chapter the dynamic mapping approach is described in detail. Additionally the main concepts and features of ROS will be shown, which play a central role in the implementation part.

In chapter 5 the development of the robotic software for mapping, finding dynamic areas within maps, import and export functionality of maps, live robot visualisation and others are going to be explained. The underlying software architecture of our implementation will be explained in detail as well as the algorithms, which were used. All ROS modules, required for the dynamic mapping process, and their integration in the framework will be discussed. The process of gathering information about non static objects in maps and especially the information sources to get this information are explained.

Chapter 6 contains information about how the implementation can be used, hence it is described, what needs to be done to start the dynamic mapping software. Additionally experiments with a robot running the dynamic mapping software demonstrates the usefulness of dynamic mapping.

Future work is discussed in chapter 7 as well as how the dynamic mapping software could be extended and used in the future. Some interesting application possibilities and concepts are also contained here.

Finally chapter 8 concludes this thesis.

CHAPTER 2

Basic Concepts

In this chapter basic concepts and the terminology used are described, which are required to understand the background and domain of the thesis and to avoid any misinterpretation. The domain of robotics is wide, hence an overview of the main concepts used in the context of this thesis is given.

2.1 Probability

The term *probability* is heavily used within this thesis. In [2] probability is defined as *"A number expressing the likelihood of the occurrence of a given event, especially a fraction expressing how many times the event will happen in a given number of trials."* Another definition of *probability* is given in [1]: *"measure of the chance of occurrence expressed as a number between 0 and 1, where 0 is impossibility and 1 is absolute certainty"*.

So the term probability denotes how likely it is that something happens. The *probability theory*, a subdomain of mathematics, defines how probabilities are calculated. Probabilities are used in very different fields of research. For instance in computing systems and especially in fault tolerant computing systems, probabilities are used to describe how likely it is that a system will fail. Engineers try to build systems with a very low probability that the system will fail.

Probabilities are also used in signal processing, where the receiver tries to reconstruct the sent signal out of the noisy received signal. Apart from that there are lots of other fields of research were probabilities are used. In Section 2.3 it is described how probabilities can be used within maps.

2.2 Change rate

Besides probability, the term *change rate* is heavily used within this thesis. We define the *change rate* as *"the frequency of the occurrence of a certain event"*. In fact the *change rate* is a frequency and hence the reciprocal value of a time interval.

For instance it is assumed that a door is opened and closed a few times within a certain amount of time. Then the *change rate* of a door state change is the amount of opening and closing door events divided by the amount of time ($change\ rate = \frac{number\ of\ events}{time}$).

The reciprocal value of the *change rate* is then the average time until a change occurs.

In Section 2.3 it is described how change rates can be used within maps.

2.3 Maps

The Oxford dictionary defines a map as *"a diagrammatic representation of an area of land or sea showing physical features"* [39]. But the term map can be interpreted differently in different domains. For instance city maps are used to find a way from one location to another, landscape maps have the purpose to show different landscapes like woods, fields or rivers. Weather maps show where it is cloudy and where the sun shines. For the purpose of the given topic the definition from the Oxford dictionary needs to be extended, because there exist also maps of areas, which are not part of land or sea. For instance floor maps are even more detailed as street maps and are used to build a house. Apart from that there exists a variety of different maps.

What all maps share is that they were made for a specific use case. Street maps, for example, are designed to guide a person through the streets and the traffic of a determined region. Thereby unnecessary information is omitted such that the user is not distracted.

The following terms are used within this thesis in the context of maps:

obstacle: Any item, where it is not possible to move or drive through it. (e.g. a wall, a desk, a human)

object: An obstacle which is moveable or can be moved. (e.g. a desk, a human, a car)

free area / free space: An area in the map, where are no obstacles.

known area: An area which is contained in the map and where the main characteristics are visible.

unknown area: An area which is not visible in the map. Either it is outside of the area, which is contained in the map, or the area is in the map and is surrounded by known area. So it is a black spot at the map.

discovering areas: An unknown area is changed into a known area by a map update.

pixel probability: The likelihood that an obstacle is located at a specific pixel on the map.

dynamic behaviour: The fact that a specific entity changes.

change rate: Denotes how often dynamic behaviour occurs in a map or parts of a map.

landmark: A specific point from the environment, which is used as orientation point for the mapping algorithm.

In robotics specific maps, containing the right type of information, are needed as well. These maps need to contain information about where the robot can drive and where the robot can not drive. The areas, where the robot must not drive, are marked by any form of obstacle existence in the map. Whenever the term map is used in this thesis, we mean such maps, which are designed for robotics and robot usage unless otherwise stated.

Whenever talking about maps for robotics, it must be distinguish between the following types of maps:

2.3.1 Static maps

The most common types of maps are static maps, hence they are created once and are changed very infrequently like a city map. Whenever there are updates or changes in the map, the whole map is regenerated. These maps are vital for environments, which change not very often or where it's not a big issue, when parts of the map are not up to date all the time.

2.3.2 Dynamic maps

There are environments, which change often and where it is important that maps are kept up to date. An example would be a weather map - it changes continuously during the day and we expect up to date weather maps presented in television. There are dynamic maps, where the amount of information that change is very high, as in weather maps. These type of dynamic maps are valid only for a specific point or interval of time, so they need to be updated very frequently. Another type of dynamic maps is, where information still changes, but slowly. In this type of maps the update rate can be lower.

The change rate of dynamic maps need not be the same for the whole map, but may differ between different parts of the map.

2.3.3 Probabilistic maps

Until now we have considered maps, where the existence of obstacles is binary, hence they exist or don't exist on a specific position in the map. But maps can be generated, where the existence of obstacles is probabilistic. Let's go back to the example of open and closed doors. We could calculate a probability of the door being opened or closed. When we come across the door four times and it is opened three times, we can assume a probability of 75 percent that the door is open. We can use this information for future action planning. In probabilistic maps we save this information. Another example of a probabilistic map would be the raining probability in a country. There are areas, where it's most likely to have rain on a specific day and others, where it's not.

2.4 Generating maps

As already mentioned a robot needs maps to perform its tasks, but where can we get a map from? First of all we can use existing maps from cities or countries, from buildings or rooms. In most situations we do not have a map, which is appropriate for our needs. So they often do not contain information about where it is possible for the robot to drive, and where not. From city maps we do not know exactly how broad a street is. Maps of buildings often do not contain the furniture or other obstacles. We also have to consider the data format of the map as the robot can only use a specified format of maps.

Another point is that there might not exist a map from a specific environment (e.g. a cave). In that case a map needs to be generated anyway.

So in most cases we won't be able to use existing maps, but generate them ourselves. For map generation some sort of sensor is needed. The most common ones for mapping are cameras and laser scanners.

2.4.1 Cameras

From cameras we can get a lot of information about the environment and often cameras are used to generate maps. When thinking about street maps, they are often generated from satellites or air crafts. Street maps can also be generated by driving through the streets, but this is not often the case.

With cameras we usually get two dimensional images. In the case of generating maps by a camera from the earth, we have the problem of the perspective.

When the camera is located on a robot it is impossible to take images from the top, like city maps are. Hence depth information has to be extracted from the image to know how far away objects in the image are. To gather this information sophisticated image processing algorithms are necessary. In fact it is possible to do mapping in such a way, but other techniques provide better results.

2.4.2 Laser Scanner

Different from the camera, a laser scanner was built to provide depth information. It measures the distance of objects by a laser beam, by measuring the duration of the sent out light until it was reflected at an obstacle and detected again by the robot. Most of the laser scanners operate in a two dimensional space, which means that they sense the environment at one specific height, if they are mounted horizontally. On a vehicle they are indeed often mounted horizontally. Most laser scanners are able to sense 180 to 270 degrees, which means they can't sense what is happening in the rear. With the laser scanner it is possible to create a map, which can be used without any image processing. The depth information provided by the laser scanner is used by mapping algorithms, which create maps by determining the current position and orientation of the laser scanner in the map and updating the map with the current laser scan. This mapping process is described in Section 3.2.

In general it is just measuring the distance between objects, which is sufficient to create maps of the environment. Such a map generated by a laser scanner can be seen in Figure 2.1. Here a part of the institute was mapped. The corridor and some parts of rooms can be seen. For instance on the left side of the corridor there are open doors, where the laser scanner only mapped a part of the rooms behind. This is a type of map, which contains information about where the robot can drive (white area), and where it can not drive (grey area).

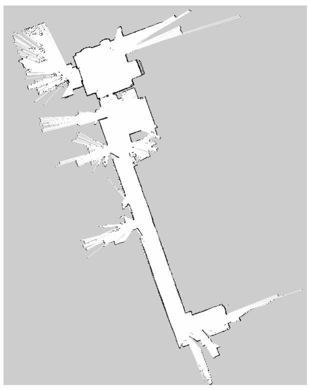


Figure 2.1: Map generated by a laser scanner

There also exist three dimensional laser scanners, but for our needs a two-dimensional map is sufficient.

2.4.3 Infrared Sensor

There are also other sensors available, which perform three-dimensional scanning of the environment, like the kinect [36]. Infrared light beams are sent out, and the reflected light is used to calculate the depth information. This makes it possible to capture a depth image. We won't go into further detail as we won't use this type of sensors in this thesis.

2.5 Map formats

For maps there are two possible formats to process and save them:

- Pixel maps (bitmaps)
- Vector graphics

The usage of vector graphics can have advantages, as the memory usage of the files for instance. But this is only a real advantage, if the files contain only a few elements and preferably only geometric forms as lines, arrows and ellipses for example. In case of obstacle existence, there is a lot of noise around obstacles caused by measurement errors of the laser scan. This has not only an impact on the geometric form of objects, but also on the value of the pixels. So in this case I think that vector graphics will not bring a significant memory reduction. Another reason why it is better to use pixel maps in this case is the possibility of filter usage. When an iteration over the image is performed and filtering is done, it is often easier with pixel maps instead of vector graphics.

In the used robotic framework module gmapping (see Section 5.4 for details) the map is stored as pixel map internally, which is another reason why pixel maps are preferred - no conversion is needed.

2.6 Localisation

In general by the term localisation *"a determination of the place where something is"* is meant according to [15]. Localisation in the context of this thesis means to determine the position of an object of interest on a map that models the environment of the object, see Figure 2.2. In fact in robotics one of the most important tasks is to localise the position of the robot with respect to its environment. Let's think about autonomous driving in robotics. This task is facilitated, if the robot knows, where obstacles are in its close environment and which path the robot can take without damaging itself or its environment.

Figure 2.2: Localisation

Localisation can be done with laser scanners in a given map by comparing the current measurement of the laser scan and a map. In fact the measurement is compared to different positions in the map. The point in the map, where the current measurement fits best, is most likely the current position of the robot. After another measurement the probability of that the robot is located at specific positions can decrease, while it increases on other positions. In that way the real position can be found. What needs to be considered too is the current movement of the robot. The direction and speed of the movement provide vital information for localisation. This can help to ignore specific possible positions in the map, where it would not be possible to perform such movements because of nearby obstacles. How localisation in robotics is done nowadays will be described in chapter 3.

2.7 Planning and Tracking

Planning and tracking are also very important topics in robotics, especially for autonomous driving.

2.7.1 Planning

The term planning has various meanings depending on the topic it is used in. In [15] it was defined as *"an act of formulating a program for a definite course of action"*. But all different meanings of planning share that they are assumptions or commitments how specific future actions should be done to reach a goal. In the topic of robotics the future movements of the robot are often meant by planning. Mostly the goal of autonomous driving in robotics is searching optimal paths from a current position to a destination and avoiding weird paths like in Figure

2.3. Sometimes it is not sure, if there even is a path from the current position to the target position. What also needs to be considered is, if all possible paths from a position to another one are known in advance. It is important to know, if an optimal path can be found with the current knowledge or if the known area needs to be extended first.

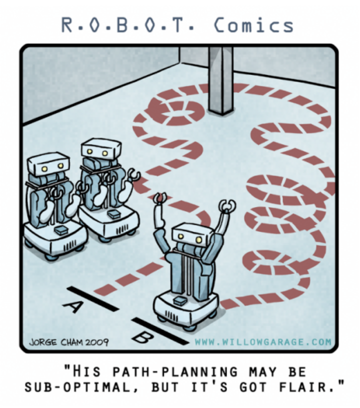


Figure 2.3: Planning [32]

The most common algorithms for finding optimal paths are Dijkstras algorithm and the A* algorithm, which is also based on the Dijkstras algorithm, but uses additional heuristics. Additional information about these algorithms can be found in [45]. Dynamic probabilistic maps can have an additional impact on finding the best path in specific situation. There might not be a single optimal path anymore, but different optimal paths depending on probabilistic areas, change rates of these areas and other parameters like time and date.

2.7.2 Tracking

A field of application, where tracking is done since decades is in radar systems of the flight control. In these systems air crafts are tracked. The term tracking was defined in the following way in *IEEE Standard Radar Definitions* [3]: *"The process of following a moving object or a variable input quantity. In radar, target tracking in angle, range, or Doppler frequency is accomplished by keeping a beam or angle cursor on the target angle, a range mark or gate on the delayed echo, or a narrowband filter on the signal frequency, respectively. Note: This process may be carried out manually or automatically for one or more of the above input quantities. The beam, range gate, or filter can be either centered on the input quantity or can be coarsely placed, with interpolation measurements providing accurate data to a computer that does the fine tracking."*

In robotics tracking is not always done with radar, but the goal is the same - determining the movement of other objects. In robotics not only the movement of the robot is important, but also the movement of other objects in the environment of the robot. When thinking about an autonomous car driving in a city, it is crucial to know the movement direction and speed of other vehicles. What objects are interesting for tracking depends on the use case. In road traffic other vehicles and pedestrians are the main targets for tracking.

A main point in tracking is, if the observer changes its position relatively to the tracked object over time. In radar stations this is sometimes the case, but they are often located at fixed positions on the earth and only the tracked air crafts move. But there are also mobile radar stations on big ships for instance. In robotics also the observer, a robot, moves while tracking other objects, which of course has an impact on the requirements for the tracking algorithm. For tracking enhanced methods exist. One of these methods is the use of Kalman filters, which are often used in robotics.

Kalman filters based methods predict the most probable location of a moving object by statistical methods and taking the results of the previous measurement into account. In fact a prediction on the current position of an object based on its previous position, movement direction and movement speed is calculated. Afterwards a new measurement is performed, followed by another prediction. This procedure is performed iteratively until we stop tracking. In [31] the Kalman filter is described in the following way.

For tracking a state vector $X_k = [x, y, v_x, v_y]$ and a measurement vector $Z_k = [x, y]^T$ are needed, where x and y are the positions of the object to be tracked and v_x and v_y are the velocities of the object. By the following set of equations the tracking is performed:

Motion equation:

$$X_k = F * X_{k-1} + W_k \tag{2.1}$$

Observation equation:

$$Z_k = H * X_k + V_k \tag{2.2}$$

Where W_k is a movement noise vector and V_k is a measurement noise vector, both with Gaussian distribution. F is a state transition matrix, H a measurement matrix.

The following prediction equations are used in [31]:

$$X'_k = F * X_{k-1} \tag{2.3}$$

$$P'_k = F * P_{k-1} * F^T + Q \tag{2.4}$$

The Kalman gain equation is:

$$K_k = P'_k * H^T * (H * P'_k * H^T + R)^{-1} \tag{2.5}$$

And the update equations:

$$X_k = X'_k + K_k * (Z_k - H * X'_k) \tag{2.6}$$

$$P_k = P'_k - K_k * H * P'_k \tag{2.7}$$

Where Q is the process noise covariance matrix and R the measurement noise covariance matrix.

So the key idea is to make predictions of the state (position and velocity) of the next step (X'_k and P'_k). These predictions are used to calculate new values of X_k and P_k.

Kalman filters can also be used for localisation by assuming that the objects in the environment are not moving and determining their consistent movement as movement of the robot instead. We can find the position of the robot relatively to them in this case. Most of the time other methods are needed together with the Kalman filter for localisation in robotics. The Kalman filter is able to localise in linear state models, but not in non-linear ones. In real world scenarios at least a part of the state model is non-linear very often. Another point is that the Kalman filter fail in re-localising after localisation failures, so their main field of application in robotics is tracking.

CHAPTER 3

State of the art

Driverless cars were a dream of engineers since decades and now we are not far away to also use them on the road. There is a huge amount of research going on and some prototypes already exist. The Defense Advanced Research Projects Agency (DARPA) Grand Challenge [19] is a competition, where driverless vehicles try to get to a goal throw the dessert. It is very popular in the scientific area and many universities participate. Because of this competition a lot of research was done in the field of artificial intelligence for robotics.

Positioning, Navigation and Tracking are very important tasks in the field of robotics and particle filters can help to accomplish these tasks [28]. Another important task in robotics is to generate a map from the environment for future (path) planning. A state of the art mapping approach is Simultaneous Localization and Mapping (SLAM) [22]. Particle filters are often used in SLAM algorithms, which draw a (probabilistic) map from the environment [34]. Hence a detailed look into particle filtering and SLAM is provided in this chapter.

3.1 Particle filtering

Nowadays a common method for localisation is to use particle filters (also known as monte carlo localisation) [21]. A prerequisite of the particle filter algorithm is a given map in which we want to localise the robot. In the literature there are different explanations of the algorithm as for instance in [21] [28]. In general they are all executed according to the following steps:

1. *Distribution (Initialization)*

   The idea of this algorithm is to assume that the robot is located everywhere in the map with the same probability at the beginning. Hence a certain amount of positions in the map is chosen randomly and uniformly, where N particles, which represent possible locations of the robot, are distributed.

2. *Measurement* Distance measurements of the robot are checked against the position and orientation of each particle, where the likelihood that the measurement fits for a specific

particle is stored. This process is also called to assign an importance weight to each particle.

3. *Resampling* Then an evaluation is done about which particles fit to the measurement and which not. This means to choose particles randomly, but with the likelihood of their importance weight. So particles with low importance weight are chosen more often. Each chosen particle is removed. The less M percent of all particles are chosen in this way and removed. Now there are fewer particles than before, so the missing M percent needs to be added again. This is done according to the likelihood of particles with a high importance weight nearby. So it is likely that new particles are generated nearby existing particles, which have a high importance weight.
4. *Movement (Prediction)* Now a movement with the robot is performed, the direction and speed of this movement is stored and also considered, when determining the new position at the next measurement step. This means that for each particle the movement of the robot is predicted. The particle filter algorithm is an iterative process, so after this step again step 2 is performed.

This procedure is done again and again until the most probable position of the robot in the map was found.

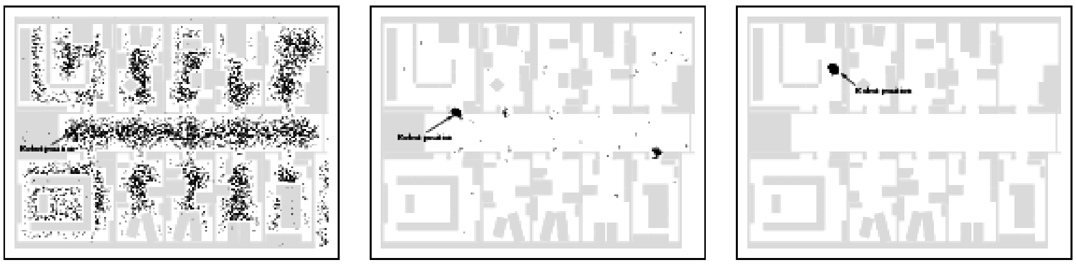


Figure 3.1: Localisation process over time [23]

At the beginning the particles are distributed uniformly over the map (see left figure of Figure 3.1), but with each iteration step the particles build some groups of particles, which are getting denser (see middle figure of Figure 3.1). After some iterations of the algorithm only one group of particles survives. The position of the robot is, with a very high likelihood, at the center of the particle cloud (see right figure of Figure 3.1). The robot position is visualised by an arrow at all three figures.

A disadvantage of the particle filter algorithm is the so called particle-depletion problem [34]. It could be that at an early stage particles with a high importance weight are removed at the re-sampling step, as it is a random process. If this is the case, it is not sure, if the particle cloud will find the correct position of the robot within the map. In most of the cases, where the particles with a high importance weight are removed, the algorithm still finds the robot position within the map, but it takes longer.

If the position of the robot within the map was lost by the particle filter algorithm because of erroneous measurements, it is possible to find the right position again after some time. This is because further robot movements can divide the particle cloud, which is caused by bad accordance between the distance measurements and the particle positions according to the Gaussian fault model. This fragmentation can hold on, until new clouds are built. After some while again one cloud is left and the robot position is found again.

3.1.1 Formal approach

The next step is to look at the formal approach of Monte Carlo Localisation (particle filtering), like described in [21]:

We define k as the current time stamp and $Z^k = \{\mathbf{z}_k, i = 1..k\}$ as all laser measurements until now. We only want to consider a two-dimensional space, hence we have a three-dimensional state vector $\mathbf{x} = [x, y, \theta]^T$. Where x and y are linear coordinates and θ the angle of the vector.

The control input for the motion control at time stamp k is denoted as $\mathbf{u}_k$.

The posterior density of the current state conditioned on all measurements is denoted by $p(\mathbf{x}_k|Z^k)$. We know that we use particles for localisation, so this set of N particles is denoted as $S_k = \{s_k^i, i = 1..N\}$.

The density $p(\mathbf{x}_k|Z^k)$ is represented by the N particles of S_k. Now the goal is to compute the set of particles S_k at each time stamp k, that is drawn from $p(\mathbf{x}_k|Z^k)$.

The algorithm is done in two steps where we start with $k = 0$ and a random sample $S_0 = \{s_0^i\}$ at the beginning:

prediction phase: : In this phase we start from S_{k-1}, which is the set of particles computed at the previous time stamp and apply the motion model to each particle by sampling from the density distribution:

```
for each particle $s_{k-1}^i$:
draw one sample $s_k'^i$ from $p(\mathbf{x}_k|s_{k-1}^i, \mathbf{u}_{k-1})$
```

Now we have a new set of particles S_k' which approximates a random sample from $p(\mathbf{x}_k|Z^{k-1})$. Until now we did not consider the new measurement at time k.

update phase: : Now we take the measurement z_k into account, and weight each sample in S_k' by $m_k^i = p(\mathbf{z}_k|s_k'^i)$. Then we obtain S_k by re-sampling:

```
for $j = 1..N$:
draw one $S_k$ sample $s_k^j$ from $\{s_k'^i, m_k^i\}$
```

With high probability the re-sampling selects samples that have a high likelihood m_k^i.

From now on these two phases are repeated recursively.

3.2 SLAM

In the previous sections mapping and localisation in robotics were discussed. But now some questions arise:

> How can localisation be done without a given map?
>
> How can a map be generated without localisation?

The answer is Simultaneous Localization and Mapping (SLAM). The localisation and mapping need to be performed simultaneously, if neither of them is given. This is an iterative process, where a map is built and in the next step a particle filter localisation is performed on this map. The probabilistic map is built from measured landmarks, where the probability denotes the likelihood that the landmark is at a certain position. Based on the location and orientation of the robot, found by the particle filter algorithm, the map is extended. Now another localisation step is done followed by another mapping step.

3.2.1 Formal definition of SLAM

In [37] the definition of the SLAM problem is given in the following way:

The SLAM problem can be described as Markov chain [1], where the position of the robot at time t is denoted as s_t. The pose consists of x, y and an orientation in a planar environment. The environment of the robot is assumed to consists of K immobile landmark points. The position of each of this landmarks is denoted by Θ_i for $i = 1, ..., K$. The set of all landmarks is denoted as Θ.

The robot sense landmarks to draw a map from its environment. The distance and angle of landmarks, relative to the robots position, are assumed to be measured. Such a measurement at time t is denoted as z_t. The control input for the motion control at time stamp t is denoted as u_t.

Now the definition of SLAM in [37] is given as: "*In short, SLAM is the problem of determining the locations of all landmarks* Θ *and robot poses* s_t *from measurements* $z^t = z_l, ..., z_t$ *and controls* $u^t = u_1, ..., u_t$*. In probabilistic terms, this is expressed by the following posterior:* $p(s^t, \Theta | z^t, u^t)$".

3.2.2 SLAM in detail

The main advantage of SLAM is that no a priori knowledge of the environment is needed. Hence SLAM can be started in an unknown area with unknown location. This makes it suitable for a wide range of applications, especially for unknown or dangerous areas. Example applications are planetary exploration, subsea missions or autonomous vehicles in nuclear contaminated areas.

To create a map, the SLAM algorithm is used, where the robot builds up a map from the environment and localises itself in this map while driving around. These maps are often dynamic, as areas are updated each time a robot comes across them if the environment has changed, they are expanded when the robot discovers new areas.

[1] According to the Oxford dictionary [39]: "*a stochastic model describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event*"

There are already different SLAM algorithms available [35] [30] [37], which where made for different use cases. In [35] they use Visual SLAM, which has the focus on accurate localisation and only use a sparse set of landmarks. In [30] the used sensor is no laser scanner, but a WiFi antenna. They introduced an approach to do indoor SLAM with information about the wireless signal strength. The SLAM algorithm in [37] uses a so called rao-blackwellized particle filter, which scales logarithmic in the number of landmarks and is hence perfect for large maps. A similar algorithm is used within this thesis. Apart from these different SLAM algorithms, there exist many more. Most of them use enhanced probabilistic methods. In [44] a comparison of different SLAM algorithms is presented.

The maps generated by SLAM algorithms are either binary or often probabilistic. Most of the SLAM algorithms work with probabilities of object existence in the environment. This means that each object in the environment gets a probability value, meaning how likely it is that it is indeed at the intended position. Sometimes inaccurate localisation, moving objects or inaccurate laser scan measurements lead to spurious objects. To avoid erroneous maps, SLAM algorithms use probabilities. Depending on how often objects are measured at certain positions the probability of an obstacle at this position changes. If an obstacle was detected in nearly all measurements, it is very likely that it really exists. If only in a few cases an obstacle was detected at a certain position, it is most likely that there is no obstacle at all and the measurements were erroneous. SLAM algorithms consider this and either output these probabilities of obstacle existence directly as probabilistic map or check the probabilistic values against a specific threshold and decide in that way, if there is an obstacle or not on a certain map position. In the second case the produced map is not probabilistic but binary.

3.2.3 SLAM in dynamic environments

There were also made first steps in using information about the dynamic environment. But in most cases only to increase the accuracy of localisation and not for gathering additional information about the dynamic behaviour or enhanced planning.

In [29] the focus lies on avoiding spurious measurements, where they try to find outliers in the measurement data. These outliers cannot be found by considering their distance to other data items, but need to be interpreted and transformed into a map first. The distance to other data items may be very large in case of small objects, but this does not mean necessarily that a measurement error occurred. A map is needed to identify individual measurements as outliers. By comparing a map with the same map after a new measurement changed the map, these errors can be found.

The approach can be summarized as follows: Start with an initial map obtained by the incremental mapping approach. Expectations are initialised with the prior probability, that a measurement is caused by a non-dynamic object. Given the resulting map and the robot position, new expectations (probabilities of reflection caused by a static object) are calculated. With these expectations a new map is generated. The overall process is performed until no improvement can be made (or a certain number of iterations is reached).

The whole algorithm is strictly based on statistic methods and is calculation intensive. The Expectation Maximation (EM) algorithm [38] is used to find data sets, which can not be explained by the rest of the data set. This is done by finding maximum likelihood estimates. The so found dynamic objects are ignored at the alignment and map update process. So no dynamic objects will influence the map in a negative way.

In [16] particle filters and Bayesian filters are used together to detect dynamic objects in maps. The paper is about concurrent localisation and environment state estimation. The particle filter is used for estimating the position of the robot, while the discrete environment variables are estimated by Bayesian filters. Those Bayesian filters are conditioned on the estimate of the robots path, so they are attached to different particles. Especially door states can be recognized by using this algorithm. The goal of the described method in this article was to localise the robot, even if door states change unexpectedly.

3.3 Is this enough?

To wrap it up, we now know how a robot generates maps and how it localises itself in its environment. We have an idea of how planning and tracking is performed. But is this all to let the robot perform enhanced tasks or do we need more information? Let's think about the following questions:

How can a robot know, where a lot of people are to be expected?

How can an intelligent car know, where free parking lots are?

How can the robot know, if there are doors in a room?

How can the robot know, if there are doors opened or closed?

I think dynamic maps are not enough. For all of these cases some kind of dynamic probabilistic maps can be the solution. In the following sections we will describe a concept to generate and save these dynamic and probabilistic maps.

CHAPTER 4

Dynamic Mapping Approach

4.1 Dynamic information

By dynamic information a type of information is meant, that is not static, but which often changes. For instance the amount of humans, walking on a street within a certain time range is dynamic information. When considering maps, the contained information is mostly static. When thinking of a city map, there are usually houses and streets, but no pedestrians or cars. So the static objects like houses and streets are contained in the map, but dynamic objects like pedestrians and cars are not included in the map.

These dynamic objects and their location, probability of existence on a certain position as well as the *change rate* (see Section 2) is dynamic information in the context of this thesis. In fact all object movements or changes in a map during the mapping process cause some dynamic information.

Within this thesis I try to find a way to automatically gather dynamic information and to save it in the map.

4.1.1 Probabilities of obstacles

The existence of many objects is probabilistic and not binary. Objects are not just there or not there at specific positions, but they are there with a certain probability. In maps often non probabilistic information about the environment is saved. In our specific case it is the intention to save the probability of obstacle existence and the change rate, which is dynamic information about the environment. Dynamic information is caused from moving objects or appearing respectively disappearing objects. From the probability of obstacles we can conclude, if an obstacle is very likely to be at a certain position, or if it is there only sometimes.

The probability of obstacle existence at an area a can be calculated in the following way. It is assumed that the amount of how often an area a was visited by the sensor is denoted by v_a. Additionally it is assumed that the amount of how often area a was detected as occupied is denoted by b_a. In detail v_a is increased every time area a is visible for the sensor. If the area is detected as occupied, which means that there is an obstacle at the area a, b_a is also increased. If the area a is detected as free, b_a stays the same.

The probability that area a is occupied can be calculated by:

$$p_a = \frac{b_a}{v_a} \tag{4.1}$$

If the laser scanner senses an other area, both values (v_a and b_a) stay the same. In this case the robot does not know what happens at the area, so it is assumed that it stayed the same. If an object was there at the beginning and suddenly removed, the probability of existence will decrease over time.

Assume a door stood closed for a long time during the observation and was opened suddenly. The probability value will change slowly because of the long observation period before (big value of v_a). When the same door was just observed for a few minutes (small value of v_a) and then opened, the probability changes faster, because there is no long history of observation. The probability, evolved over time has higher weight than probabilities of new or short observed areas.

No statement about how often obstacles move, respectively disappear or appear can be made with obstacle probability values only.

Probabilistic maps are generated by different available mapping modules. The described approach with v_a and b_a can be generalised for all areas within a map. The calculation of probabilistic information about the environment is performed by the given Robot Operating System (ROS) module gmapping (see Section 5.4). The approach with the calculation of the probability p out of v and b, is used by gmapping according to [27]. This probability value is saved in the map and is fed as initial value to the mapping algorithm, when the map is loaded.

4.1.2 Change rates of obstacles

Apart from obstacle probabilities the change rate is another very useful value. The change rate denotes how often obstacles move, respectively appear and disappear. It makes a difference if objects appear and disappear seldom or frequently. By measuring the time between object movements (appearing respectively disappearing), an average *time until change* can be calculated. The *time until change* is the reciprocal value of the change rate. Both terms, the *time until change* and the change rate will be used although only the *time until change* is stored.

For specific use cases this change rate (respectively *time until change*) may be even more important than the probability value. For example when searching for free parking lots the probability of if *it is free* is interesting, but what is at least equally important is the change rate. If cars park at a certain parking lot only for a short time in average, the chance is big to get the parking lot if the driver waits some time, even if the probability of this parking lot being free is very low.

The *time until change* between the last change of an obstacle (obstacle appearance, respectively obstacle disappearance) to the next change (obstacle disappearance, respectively obstacle appearance) is denoted as time value t_{until_change}.

As the *time until change* contains only information between two consecutive changes, an average *time until change* value can hold information of the previous *time until change* values.

For the *time until change* we decided not to use the standard average value over the whole history of an obstacle. Instead an *exponential moving average time until change* $t_{ema\_until\_change_n}$ is used. The weight factor is denoted by $weight$ and is a value between 0 and 1. We assume that the last value $t_{ema\_until\_change_{n-1}}$ was already calculated. At the first calculation $n = 0$. Hence we set $t_{ema\_until\_change_{n-1}} = 0$ and $weight = 0$. For every subsequent step the new change rate is calculated by:

$$t_{ema\_until\_change_n} = t_{ema\_until\_change_{n-1}} * weight + t_{until\_change_n} * (1 - weight) \quad (4.2)$$

If the *time until change* t_{until_change} is assumed to be measured in milliseconds [ms], the change rate (change frequency) in Hertz [Hz] can easily be calculated by:

$$f_{change} = \frac{1}{t_{until\_change}[s]} \quad (4.3)$$

4.1.3 Probability and change rate of obstacles

Calculating both, the probability value and the change rate of objects provide an useful amount of information about dynamic areas.

In probabilistic maps there is a lot of information contained, which is not visible at first sight. On the edges of objects, for instance, information about the measurement error respectively localisation accuracy can be gained. When the transition from free area to almost sure obstacle (still speaking about probabilities) is very sharp, the measurement and localisation are very accurate. If the transition is very blurred, meaning a big area where it is uncertain, if it is a free area or an obstacle, there might be measurement or localisation errors.

The change rate and probabilistic value together provide enough information to classify different objects. For example a high change rate and a high obstacle probability value can be caused from doors which are opened often and close automatically afterwards.

A low probability value and a low to middle change rate can be caused by humans walking in a certain area.

We will not consider classification of objects any deeper within this thesis, because this is an own field of research.

Beside other possible origins of dynamic areas we want to focus on dynamic areas caused by:

- door states
- walking humans
- small obstacles (e.g. boxes)

To sum it up, the first approach is to use probabilistic information as well as change rates about the environment in maps.

4.1.4 Dynamic area format

First of all a decision has to be made which shape a dynamic area can have. Should only rectangular dynamic areas be allowed, or is any polygon allowed? Only rectangular dynamic areas are too restrictive in my opinion and prevent us from saving information about the real shape of a dynamic area. So at least a polygon should be possible as dynamic area shape.

There are two possibilities to save a polygon. The first one is to save it like a vector graphic - only save the vertices of the polygon. In that case one global dynamic probability value for the whole dynamic area can be saved. The advantages of this method are the small memory capacity which is needed for large objects and that dynamic information only need to be stored once per dynamic area.

The other possibility is to save it as pixel map. In this case the dynamic area can have any shape. It is beneficial to save a dynamic value to each pixel instead of saving one dynamic value for the whole dynamic area. The advantages of this method are that every pixel has its own dynamic value, which makes it possible to distinguish between different areas with different dynamic values within one dynamic area. Another advantage is that it is better suitable for most filtering algorithms. Many filtering algorithms iterate over each pixel of a map and consider the neighbour pixels to change the current one. This is not possible with vector graphics.

Because of the own dynamic value of each pixel and the possibility of filter usage I decided to use the second method in this thesis. Additionally, the maps themselves are also saved as pixel maps instead of vector graphic which is discussed later on in Section 5.5. The disadvantage is the higher memory effort of saving dynamic areas.

4.1.5 Generation and update of maps

When considering a probabilistic map generated periodically from sensor data like it is provided by standard mapping modules, successive maps can be compared and additional dynamic information can be obtained in that way. For instance change rates can be calculated by considering changing probability values as will be shown in this section. When generating maps there are two possibilities which need to be distinguished:

New map on each update

Assume on each laser scan a new map is generated. We can compare different maps regarding known and unknown areas. Areas which are known in one map and unknown in the other map are most likely caused by dynamic behaviour. If this happens in consecutive maps, an object has appeared respectively disappeared, if the same area was scanned in both maps. With this information, *time until change* can be calculated directly by measuring the time between two such events.

One map on its own can not contain probabilistic values, because they evolve over time. At one specific laser scan, objects are just there or not, there is no probabilistic value in between possible.

The problem is that the robot can move between the generation of two maps, such that the origin and orientation of the two maps is not the same any more. This problem is handled internally in Simultaneous Localization and Mapping (SLAM) algorithms, where the new location is taken into account.

Changed map on each update

Most implementations of SLAM algorithms save a probabilistic map and merge each scanned map with this probabilistic map. Hence it is not possible to simply compare unknown, respectively known areas between the two maps to gain dynamic information. Furthermore they often just provide a changed probabilistic map after each update.

It is assumed that the map is updated each time a new laser scan was performed, which means that there is no new map from each laser scan, but that the laser scan changes and extends the existing map.

We decided to use the ROS module gmapping (see Section 5.4) which provides the probabilistic map, but no maps of single laser scans. That's the reason why the dynamic information need to be gained from comparing consecutive probabilistic maps.

4.1.6 Map comparison

As already discussed, the dynamic information needs to be obtained by comparing probabilistic maps. The term *old map* is used for the map before a new sensor measurement has changed the map, and the term *new map* denotes the map after the sensor measurement has already changed the map. Certainly the relevant cases are only those, where both (the new and the old) maps differ. The following cases of map comparison are interesting:

1. Gather dynamic information from an area, which is known in the new map, but unknown in the old map

2. Gather dynamic information from an area, which is unknown in the new map, but known in the old map

3. Gather dynamic information from an area, which is known in the new map and known in the old map, but the probabilities are different.

Of course it is impossible to gather any dynamic information for areas, which are unknown in both maps. The second point is not very obvious, because it is rather impossible to have a map, where all laser scan measurements until now are included and any area is known in the old map, but unknown in the new map. Until now we assumed that the map is only extended or changed after each new laser measurement. An area which is known once, will not be unknown in the next step. So how can we have an area, which is unknown in the new map, but known in the old map?

This can only be the case when a given map of the environment is compared to the map generated by a single laser scan. The given map may contain areas, which are not known in the generated one. Assume an export and import feature of maps exists. So a map can be saved to a file and can be imported later on. At the import procedure it can be the case to have an area, which is unknown in the laser generated map, but known in the imported map. Hence all three points can occur and will be discussed in detail.

Known in new map, unknown in old map

This is the case, if a new area was discovered within the new map, which was not discovered so far and hence not known in the old map.

At this point the area was detected the first time, so there was no previous dynamic behaviour measured. No dynamic change rate can be calculated because a reference value is missing. From now on the time is measured until the next dynamic behaviour occurs. Then the *time until change* can be calculated the first time with $n = 0$, $weight = 0$ and $t_{change_{n-1}}$ by Equation 4.2.

Unknown in new map, known in old map

As already described, it can only be the case that the old map contains known areas, which are unknown in the new map, if a given map was imported and considered as old map. If a known area in the old map and an unknown area at the same position in the new map were discovered, two cases are to be distinguished:

- The area was just not discovered yet in the new map
- The area is unknown because of an obstacle preventing to observe the area

In the first case no dynamic change rates need to be updated. The area was unknown until now. In the second case it depends on the last measurement of the area before the (old) map was exported. If there was no obstacle at the area, this would cause a change rate update now. If there was an obstacle at the last measurement of the area before the (old) map was exported, nothing changed. Hence no change rate update is required. The problem is to distinguish between these cases to determine, if a change rate needs to be updated or not. With the map only this can not be done. An additional variable, which holds the last measurement, need to be exported too. According to this variable, it can be distinguished between if an obstacle has appeared since the export of the map or if no obstacle has appeared. How this import and export is done in detail depends on the implementation and is described in Section 5.5.

Apart from the necessity to import different variables (see Section 5.5), the same approach like in the case *Known old map, known new map* which is described in the following section can be used. It is only necessary to provide specific information about the last laser scan observation before the export.

Known old map, known new map

In this section, the case where an area is known in the old map and where the same area is also known in the new map, is considered. If this is the case the following conditions can help to determine, if a pixel is count as dynamic one:

1. The probabilistic value over time has an extremum

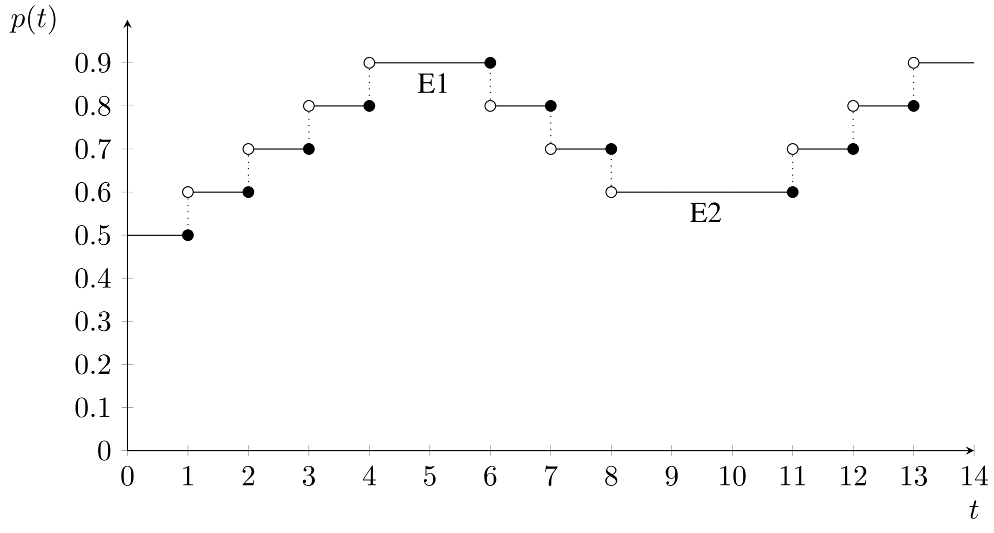


Figure 4.1: Probability value over time

2. The area around the current pixel is also known

The probabilistic value over time has an extremum

In consecutive probabilistic maps dynamic behaviour can only be calculated by extreme values. This is because at these points, the probability value changes its direction. Hence an object has appeared or disappeared.

Assume consecutive probabilistic maps generated by a SLAM algorithm over time. The function of the probabilistic value over time of a certain area, which we call *area 1*, is shown in Figure 4.1. For simplicity, the map update is assumed to be periodical with a period of 1 second. At *area 1* an obstacle was measured at the timestamps 1 to 5. The probability increased over time, because of obstacle existence. Then the object disappeared in the interval E1, where the probability value over time starts to decrease now, because no obstacle is measured any more at *area 1*. During the interval E2, the object was measured again at *area 1*, but the probability value stayed the same for one time step. This can happen because of Equation 4.1. But the probability again increases afterwards, because an obstacle was scanned at *area 1*.

At this example the dynamic behaviour took place at the extrema E1 and E2.

The dynamic event of object appearance can be detected by searching for extrema in the probability value over time. The same is true for disappearing obstacles. The only difference is the sign of the gradient of the first deviation of the probability value.

It is known that each pixel has its own probability value. Hence the *time until change* is also calculated for each pixel on its own. This value is increased every second until a dynamic event occurs again (where the *time until change* is set to 0). This dynamic event can happen for each pixel at an other point in time, so each pixel needs to have its own timer.

Known area around the pixel

This point avoids dynamic areas at the edges between the known area to the unknown area. At the edge of the known area measurement errors might be misinterpreted as dynamic areas. The change rate is not calculated at this areas. In addition, if a dynamic area is at the edge of the known space and big enough, it indeed will be recognized as dynamic. How far away from unknown areas dynamic areas are allowed, can be set by parameters. It is again dependent on the use case and the dynamic objects which should be observed to set this parameter accordingly.

These points facilitate the task to get paths, where humans walk most of the time. In corridors for example, there are areas, where humans walk around often. So if the robot drives there around, paths with the probabilities of how often humans walk on different paths will arise. Also door states as well as other dynamic behaviour can be observed by applying these rules.

4.1.7 Unknown areas

As already shown, unknown areas are important to evaluate dynamic areas, which will be discussed in more detail now. In the previous section, there was already shown how maps can be compared and that new areas contain a lot of information. But when converting obstacles into dynamic areas a few more things must be considered.

If the new area was discovered beside an obstacle, there is an unknown area left in the immediate environment (see Figure 4.2 and especially Figure 4.2b). Here the edges of a wall are discovered and the core of the wall is left as unknown area. This helps to avoid change rates respectively dynamic areas nearby unknown areas. There are changes in the probability value of obstacle edges caused by measurement errors. These should not lead to dynamic areas and are hence avoided.

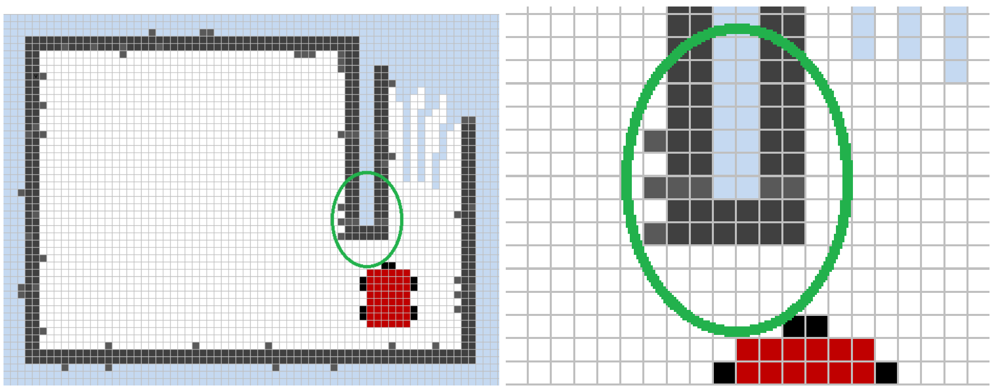

(a) (b)

Figure 4.2: Avoiding obstacle to dynamic area conversion

With this concept dynamic areas around a certain zone of unknown areas are avoided. What now needs to be discussed is the size of these areas, where no dynamic area is allowed. If the area is too small dynamic pixels might be generated in certain cases around unknown areas. If the area is too big, the generation of dynamic areas could be suppressed where they should appear. The size can be determined by some experiments and can also be configured using parameters.

4.1.8 Dilation filtering

Until now we did not consider neighbour pixels while processing dynamic pixels. A dilation filter changes the value of specific pixels and helps to expand shapes of an image, respectively map. Another point are small dynamic areas, which are caused by walking humans for instance. They are rather small, but if many humans walk the same paths in a room, dynamic areas are generated which together form paths of human walks. To fill holes within dynamic areas and merge near dynamic areas such a filter is a good choice. This is because a dilation filter iterates over the image and adds dynamic pixels besides other dynamic pixels in areas, where a lot of dynamic pixels are located. In Figure 4.3a a dynamic area, which was just generated, is shown. Holes in the dynamic area are caused by measurement errors. Specifically, the edges of the dynamic area are not very sharp. This is often the case when objects are removed and changed into a dynamic area. After a dilation algorithm is executed on the map, additional dynamic pixels are added and the map looks like shown in Figure 4.3b. Now there are sharper edges at the dynamic area and no more holes exist within the dynamic area.

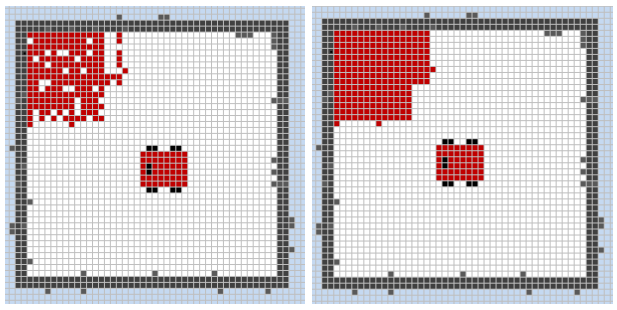

(a) (b)

Figure 4.3: Dynamic area before and after dilation filtering

The filtering is done for each known pixel where a rectangle around the pixel is considered. If there are dynamic pixels located in this rectangle at specific positions, the pixel in the middle of the rectangle is changed into a dynamic one. The detailed algorithm to perform this task is described in Section 5.5.5.

4.2 Generating object edges versus generating object surfaces

When considering moving objects, there are two possibilities of representing them as dynamic areas in the map:

- Generating dynamic areas only at object edges (see Figure 4.4a)
- Generating dynamic areas over the full area of the object (see Figure 4.4b)

Each possibility has its advantages and disadvantages. The question is, what is adequate for our needs?

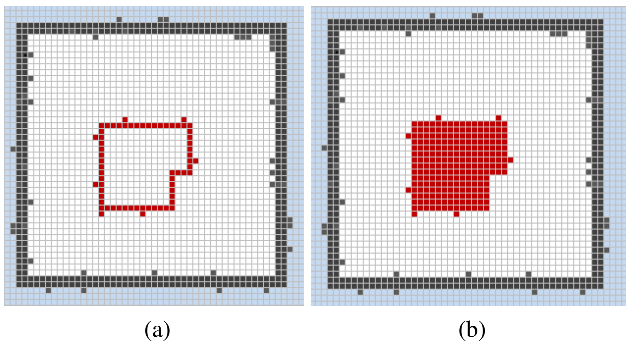

(a) (b)

Figure 4.4: Dynamic edges versus dynamic surface

When discovering the environment by laser scanners, there are only edges of objects detected, because the laser scanner can not see into, respectively through, obstacles. At the mapping process these edges are mapped and an unknown area at the center of the object is left.

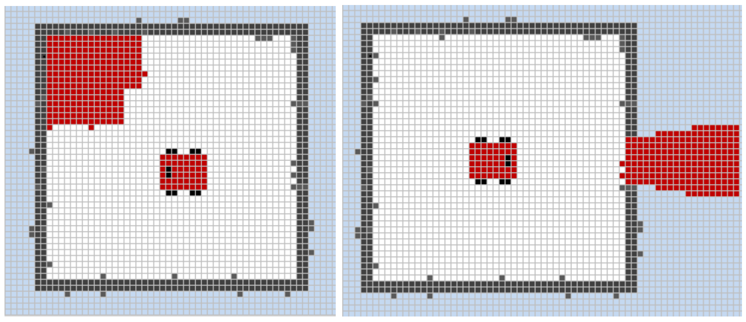
As already described the change rate calculation is performed on probability values. Hence these values are also generated on obstacle edges only.

The main advantage of considering edges only is that it is completely irrelevant to which object an edge belongs to. If an edge is recognized as dynamic, it can simply be changed into a dynamic area without considering other edges. While navigating the robot through its environment it is often the case that only a few edges of objects are detected or even visible. Otherwise the robot would have to drive around each object to scan it from all sides to generate the complete surface of it.

Additionally it is often the case that big objects, like the furniture in rooms, are located besides a wall, such that it is rather impossible to scan all sides of the object by laser scan. In this case it is difficult to distinguish the object from the wall and vice versa. This is only possible if the object is removed for at least one time. Then there is the chance to get an area like in Figure 4.5a, where the dynamic area is drawn as full surface of the object. By doing so the full space behind the object to the nearest unknown area is changed to a dynamic area.

If the same technique is used when a door is opened, the result would look like Figure 4.5b. The complete new area would be seen as dynamic one, but in fact only the area of the door itself should be a dynamic area. The rule for interpreting areas, which were discovered through a previous obstacle does not work in case of door opening. Because the dynamic area would reach until the unknown area.

What would be required is to check if the complete area around the intended dynamic area is already known prior to the generation of a dynamic area. By doing so the problem with opened



(a) (b)

Figure 4.5: Misleading dynamic surface

doors would be solved. But anyway the door itself has to be marked as dynamic area, because it shows dynamic behaviour.

Another case is to have two rooms which are both known areas and connected by a door. Let's assume the door is opened during the whole observation time and suddenly closed. Up to where should the dynamic area be drawn now - the whole room? Or just 10 cm behind the door? To be really sure how the surface of an object looks, a classification process is required. If the robot knows which parts of the map are doors, which are humans and which are furniture, then drawing the full surface of the object as dynamic area is possible. This classification can not be done by the laser scanner on its own. Additional information from other sensors like a camera is required.

When using only the edges of dynamic areas, these problems disappear without big additional disadvantages. For doors it is even the best case to only work with its edges. If edges are connected in the form of a polygon, they can easily be transformed to the surface of an object if this is necessary. If needed this step can even be done by other software modules and provided as service for other modules. Then either the original map with dynamic edges or the additionally provided map containing dynamic surfaces can be used.

Consequently, I made a decision to use dynamic object edges in this thesis.

4.3 Dynamic map based action planning

After the creation of maps, containing dynamic areas, they can be used for enhanced action planning. This offers a variety of new opportunities and use cases, where a few are described now.

4.3.1 Path planning

Robots can use dynamic probabilistic map information to drive carefully in dynamic areas, by reducing its speed for instance. Depending on the probability and change rate of the dynamic area the robot may avoid driving through the dynamic area because of higher collision probability with any other object, human or other robots. If the probability of an dynamic area is low, the robot may drive through the dynamic area, but precautionally reduces its speed.

Another opportunity is to use dynamic area information for path planning. When considering the map in Figure 4.6, it is assumed that the environment was already discovered by the robot. Between room A and room B a dynamic area was generated because of a door. The door was observed as closed with a probability of 0.4, and has an exponential moving average time until change of 120 seconds. The dynamic area between room B and room C is also caused by a door and has a probability of 0.3 that the door is closed and an exponential moving average time until change of 60 seconds. There is another door between room A and room C, with the dynamic area probability of 0.7 that the door is closed and an exponential moving average time until change of 30 seconds. Higher probability means a probability of an obstacle. Higher exponential moving average time until change means opening respectively closing the door seldom. The robot wants to drive from its current position to the green dot - its goal position. To do so there are two possibilities:

1. room A $\rightarrow$ room C
2. room A $\rightarrow$ room B $\rightarrow$ room C

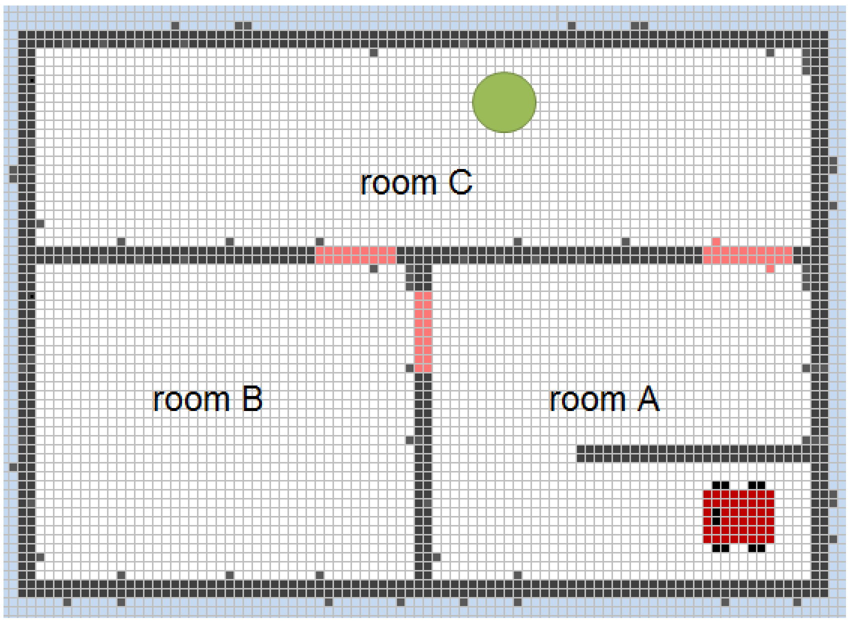


Figure 4.6: Enhanced path planning example

What can be done now is to integrate the dynamic areas, their probabilities and exponential moving average time until change for enhanced path planning. By assuming that the probabilities of all three dynamic areas are independent from each other, the probability that both doors room A to room B and room B to room C are open can be calculated.

We have to calculate with the probabilities for open doors, which is shown in Equation 4.4. The probability that the door from room A to room C is opened is calculated in Equation 4.5.

$$P_{opendoors}(A - B - C) = (1 - 0.4) * (1 - 0.3) = 0.42 \tag{4.4}$$

$$P_{opendoors}(A - C) = (1 - 0.7) = 0.3 \tag{4.5}$$

In case the probability is the only factor for path planning, the robot should choose the path through room B. Here the probability is higher to pass only open doors.

If the change rate (respectively the exponential moving average time until change) also is taken into account for path planning, the path A-C is the optimal one. The probability values are not far apart, but the change rates. The exponential moving average time until change from room A directly to room C is 30 seconds. Which means that the door is opened in 30 seconds in average if it is closed. So waiting is a good solution.

A more enhanced method could also reevaluate the possible paths after actually observing a closed or opened door. Considering dynamic area probabilities, change rates and possible paths leads to graph theoretical problems and can help to find optimal paths.

4.3.2 Road traffic

If dynamic probabilistic maps are applied to road traffic, there is also the possibility to use enhanced path planning. If the dynamic area change rate is high on some streets - meaning there is a lot of traffic, those streets can be avoided by the car navigation system. Crosswalks will also be discovered as dynamic areas, where autonomous cars could reduce their speed automatically or the driver could receive an information to pay attention when entering a dynamic area.

Another use case would be finding free parking lots. The probability of free parking lots can also be stored as dynamic areas in a map. Assuming a map, where besides the street parking lots are visualised, including the probability if they are free and their change rate. If such a map exists, a navigation software can lead the driver to an area, where the highest probability of free parking lots exists. The best parking lot again need not necessarily be the one with the highest probability that it is free, but one with a high change rate. Some parking lots are occupied nearly all the time. But if the average parking time is low (high change rate), this would also be an interesting parking lot for people how search one.

Of course such a map needs to be created once and then updated in certain intervals. This task could be performed by each car individually, but also by cars of the public service. Bin lorries drive around a whole city every week and regularly observes each parking lot. Hence if all bin lorries of a city are equipped with such a system, a dynamic map containing probabilities as well as change rates of parking lots can be generated and hold up to date.

This generated map can be offered to drivers over the internet and will simplify driving and parking in a city.

Considering the daytime or date

When considering parking lots it can also be useful to not only store one pair of probability and change rate value, but a set of these pairs. One pair probability and change rate value can be active in the morning, another in the afternoon and a third one in the evening and night. Because the probability and change rate value of free parking lots may be different in each of the timeslots. During the day there are a lot of free parking lots in residential districts, because a lot of people are at the work. In the evening these people drive home from work and park their car, then there are only a few free parking lots in this area, but a high change rate. In areas with high industry the behaviour of free parking lots is vice versa. In the night there are a lot of free parking lots, while there are only a few during the day.

Additionally the probabilities and change rates can be distinguished between weekday and weekend. If this is done, each pair of probability and change rate is only shown and updated during the time it is valid. This approach will lead to additional storage and update effort. But the gained additional information is very useful and will lead to more accurate probability and change rate values for specific points in time.

Enhanced action planning

When thinking forward into the future it would also be possible to store additional information in the map. A classification can be introduced, where each object is assigned to at least one group of classification. This classification can be made according to the size of dynamic areas, the average probability value or its change rate for instance. This will help to use dynamic areas for different use cases. Some features can only be performed for specific classes of dynamic areas. Examples of these classes are vehicles, parking lots, pedestrians, crosswalks and many more.

If such a classification is made, different classes can have different allowed speed values for robot driving through them. If an area was classified as crosswalk, it is reasonable to use a very low allowed speed value.

If a door was classified as door the robot knows from that information, that it is possible for it to drive through it, if the door is open.

4.4 Introducing the Robot Operating System (ROS)

Robot Operating System (ROS) is a framework for robotics. Its main tasks are to provide device drivers and a standard intercommunication channel between its modules. We will use ROS in the version *ROS Groovy Galapagos* [11], which was released in December 2012.

Other than its name pretends, it is not an Operating System in the usual sense, thus it does not provide memory management, interprocess communication, scheduling and all other things an operating system usually does. Instead ROS is installed on top of a real operating system like Ubuntu Linux, which is the preferred one according to [13]. ROS is an open source project, well documented [9] and heavily used for scientific robotic projects. The ROS framework comes with functionality that facilitates development for robots. One big advantage of ROS is that researchers can reuse existing ROS modules, which are available at the ROS community, and they can concentrate on specific topics in their scientific work, instead of building a robotic framework system first. To start, an own ROS module has to be written and the standard communication system of ROS to intercommunicate with other modules can be used.

ROS nodes (software modules) can be implemented in the following languages: C++, Python, Lisp, Java and Octave. Most of the modules available at the ROS community are written in C++, as it is well known and fast.

ROS itselves outlines three levels of concepts [12]:

- Filesystem level
- Computation Graph level
- Community level

These concepts will be shortly described for a better understanding of ROS.

4.4.1 Filesystem level

At this level the main concepts are:

Packages: A package is the main unit of how the software is organized in ROS. It contains runtime processes, configuration files and other files which logically belong together.

Stacks: A Stack is a collection of packages that perform a useful task together. So there arises some form of emergence if the modules work together. An example would be the navigation stack, where there are packages for mapping, path planning and robot control. These packages together can navigate the robot from a starting to a destination position.

Message types: These types define the data structure of messages sent in ROS.

Service types: These types define the data structure of service requests and responses in ROS.

4.4.2 Computation Graph level

The Computation Graph of ROS is a network of processes, which is processing data together. The elementary things are:

Nodes: The structure of ROS is very modular, where a node is the smallest entity - namely a process that performs computation.

Messages: Messages are structured and strictly typed data items which are defined in message definition files. Nodes communicate by passing messages to other nodes.

Topics: A topic is the name of a channel, where messages are sent over. Topics have publish / subscribe semantics (see Figure 4.7). That means a node can publish information on a specific topic - it sends data on a specific channel. Arbitrary many other nodes can subscribe to this topic - they listen on the channel to receive the sent data. There can be multiple subscriber on only one topic, but there can also be multiple publisher on one specific topic. The syntactical message structure on a topic is restricted by message types.

Services: As the topic concept is a many-to-many approach where the information flow is one-way, there are some situations where it is beneficial to have a request / response approach. Services are a request / response communication approach, where service messages exist for the request and the response to establish a strongly typed communication. An advantage of this communication paradigm is that the sender gets an information from the receiver to be sure, that the message was received successfully.

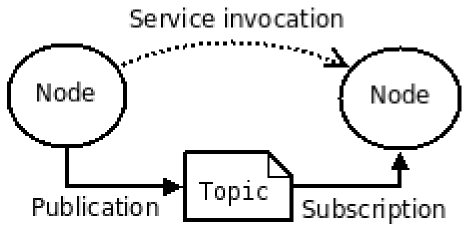


Figure 4.7: Communication concept of ROS [9]

The ROS master is the node, where the main roscore module was started, which handles the intercommunication between different nodes. The ROS master has a list of all topics and services and forwards this information to all nodes. Each topic and service is registered by the master, so new modules can be started and stopped at runtime. The connection between nodes is done directly, the master only provides lookup information. The underlying protocol for this communication is standard TCP/IP sockets. As will be shown later, this makes it possible to connect different hosts, hence the modules must not run on only one host. By host a computer or robot is meant.

4.4.3 Community level

The ROS community level provides ROS distributions, which are collections of stacks. It also provides a very good written wiki [10], which is the main documentation source of ROS.

ROS implements a few more concepts at these three different levels, but only the main concepts were covered here. For a full list of ROS concepts see [12].

4.4.4 Transformations

In ROS so called *transformations* are an important concept. To make it clear what they are and what they are there for, a general explanation is given.

In a robotic system there are often different movable parts as for example gripper arms. They all have their own coordinate system and a form of connection between them is needed. Here transformations come into account - they provide information to transfer the different coordinate systems into each other. This is needed such that points of different coordinate systems can be compared. These transformations are also important when determining where in a map the robot is located. The transformations are matrices consisting of linear coordinates and orientation coordinates (angles). A matrix transformation from the robot coordinate system to the map coordinate system is required. As we will see later I heavily use transformations, because there are many different coordinate systems required within this thesis.

We can distinguish between two types of transformations:

- static transformations
- dynamic transformations

Static transformations are independent of time, they are set once and not changed any more. The transformation matrix for a static transformation stays the same. When a point is transformed from a coordinate system into another it has to be multiplied with this constant transformation matrix. An example in ROS would be a transformation between the position of a robot bumper and the laser scanner of the robot. They are mounted at the robot and will not change their position during runtime. But nevertheless it is important to have a knowledge about the relative position of these elements and know the dimensions of the robot to avoid a crash while it is driving autonomously.

Dynamic transformations depend on the time. For instance the transformation between the robot and the map. This transformations will change each time the robot is driving and changing its position within the map. This means that the transformation matrix needs to be updated every time the robot performs a movement. If a point from the robot coordinate system has to be transformed to the map coordinate system it again needs to be multiplied by the dynamic transformation matrix. Time is a crucial aspect when talking about dynamic transformations, because we have to think about how often transformations have to be updated. If we update them seldom, our algorithms will run on inappropriate and old data, which influences there accuracy. When coming back to the example of the transformation between the robot and the map coordinate system, the robot position within the map will not be calculated right if an old transformation matrix is used. If the transformation matrices are updated very frequently we might have problems with the calculation speed and influence other modules running on the same processor and we can get in trouble with the capacities of the communication system. This can lead to utilize the full Central Processing Unit (CPU) capacity of the robot with only updating transformation matrices in the worst case. Another crucial point is that all different transformations have to be connected to each other.

In ROS transformations between specific topics need to be provided. For example different maps need not have the same origin and orientation. To use them together these different origins and orientations need to be compensated. Hence transformations are necessary to uniquely calculate the relation between any topic to any other topic which are both published by ROS. Here the time aspect comes into account again, the update frequencies along a path between two transformations are added up to get the global update frequency between two topics. The transformation tree of this project is described and shown in Section 5.9.

4.4.5 SLAM module: gmapping

As already mentioned a Simultaneous Localization and Mapping (SLAM) algorithm is used to get a map from the environment. Now the used algorithm is described - a rao-blackwellized particle filter [26]. How a particle filter works in principle was already described in Section 3.2, now a deeper look is done into the used mapping algorithm.

The problem with particle filtering is that the number of particles needs to be high in order to get good solutions. With this high number of particles the computational effort rises. If a low number of particles is used the algorithm may diverge and hence calculate wrong results. This is a difficult problem and to avoid this rao-blackwellization is used.

Besides this, rao-blackwellization scales logarithmic in the number of landmarks, which makes it possible to generate big maps and localise within them. At the mapping process rao-blackwellization can help to avoid wrong localisation and hence inaccurate maps.

The idea behind rao-blackwellization is to use a Kalman filter for the parts of the state model which are linear and to use the particle filter for the other, non linear parts of the state model. The Kalman filter is much better performing, but can not be used for non linear states. In a lot of practical applications only a few parts of the state space are not linear, so this brings a real effort.

The detailed algorithm is based on higher statistics and will not be described in detail. A detailed explanation can be found in [28].

CHAPTER 5

Implementation

In this chapter it is shown how the dynamic mapping approach can be implemented as part of the ROS framework. The goal is to implement a dynamic mapping module and use existing, as well as self developed ROS modules, which together perform the task of dynamic mapping on a robot.

5.1 Dynamic mapping in ROS

As Robot Operating System (ROS) is a widely accepted framework for mobile applications, many different ROS modules already exist that might be reused for implementing a mapping algorithm able to detect dynamic areas. Hence first of all available modules were tested and evaluated.

The following approaches have been evaluated for gathering dynamic information:

5.1.1 Integrating dynamic mapping algorithms in the SLAM module

If the Simultaneous Localization and Mapping (SLAM) module is used directly (in our case the ROS module gmapping), we have access to all the information we need. The way of how areas are recognized as occupied or free can be changed. An algorithm which marks specific areas as dynamic, where there is much movement, can be implemented. SLAM modules use data from the environment to create maps and often use the bitmap format for processing these maps. The SLAM module which is used in this thesis (gmapping module) is open source and publishes this bitmap. Hence it is possible to change the SLAM algorithm and influence the published map. For most of these issues access to the internal data structures of the gmapping module is needed.

But after a second look at this possibility some problems arise.

- The main one is that the internal state of the SLAM algorithm needs to be saved. When generating maps we want to build a part of the map today and another part another day. To accomplish this, it must be possible to save the map and load it back in the SLAM module later. Saving the internal state, such that the internal probabilities for each element of the map can be restored and variables containing the current position and orientation of the robot in the map, would be necessary.

- When a previously saved map is loaded again, it should not be required that the robot starts exactly at the position, where it stopped mapping the last time. It might be that the robot needs to change its position for any reason between two mapping processes. Not only that the robot is required to get back to the position where it stopped mapping the last time, it is also necessary to find the exact orientation the robot had last time. When the internal SLAM state is simply saved in that way, this would be a problem.

- If the logic for generating dynamic areas is directly written in the SLAM module, the SLAM module can only be changed with extra effort in the future. There might be more evolved SLAM modules, which are faster and more accurate, but when the logic is embedded in the gmapping module it will be difficult to upgrade to a more evolved SLAM module instead.

5.1.2 Using the map generated by the SLAM module

When the map, which was published by a SLAM module, is used, we do not have access to the internal variables of the SLAM module and can only find dynamic areas based on the map. Gmapping was not designed to reimport a map. It was not considered to find the robot position in a given map. If the map would be imported in gmapping directly, it is not possible to localise the robot within the map in the current implementation of gmapping. Without localisation the map can not be extended and changed. But what can be done is to change the behaviour of the gmapping module according to our needs. There are some configuration parameters, where different functions are enabled respectively disabled, and where the behaviour can be influenced by changing them. At the gmapping module, the map update frequency is one parameter, which is possible to be changed. By doing so, we can change how often a new map is published by the gmapping module. The big advantage of this approach is, that we get rid of two of the problems described above. Exporting only the map is sufficient to import it later on and proceed mapping. No internal state of the SLAM module needs to be exported. Gmapping starts with a new map and this map is merged with the imported map to obtain a combined map. It is possible to save the map and load it later to start gmapping again and it is now feasible to use another SLAM module in the future. Another reason to separate the map generation and the dynamic area finding is the possibility to run them on different computers for performance reasons.

Because of these reasons I will use this approach in this thesis.

5.2 System Overview

In Figure 5.1 the intercommunication between different Robot Operating System (ROS) modules is shown. The boxes are ROS modules respectively stacks and the arrows show the direction of the information flow. In most cases information is transferred over topics. As shown on the top left of the figure, information about the environment is gathered by the laser scanner and forwarded to the gmapping module, the dynamic_mapping module and the safety module. The gmapping module generates a probabilistic map out of the laser scan and publishes it at the *slam_map* topic. Depending on a parameter a *laser_map* topic can be published in addition. The dynamic_mapping module as well as the rviz module subscribe to the *slam_map* topic.

When the dynamic_mapping module is started it can either import a given map (which was exported by the dynamic_mapping module in the past) or start without a map. If a map was imported, this imported map and the slam_map are merged. This merged map is published as *map* topic (without dynamic areas) and published as *dyn_map* topic (containing dynamic areas). Both topics can be displayed by rviz. Rviz is a Graphical User Interface (GUI) for ROS, where different topics (e.g. maps) are displayed. The map topic is subscribed by the navigation stack, where it is used to perform autonomous driving and navigation. As visualised in Figure 5.15, the navigation stack also subscribes to other topics, which were ignored in Figure 5.1. In future work the navigation stack may also subscribe to the *dyn_map* topic and use the additional information of dynamic areas for optimized path planning. But at the moment we provide the legacy *map* topic for backward compatibility reasons. The *update_area* topic contains information about the area of the latest laser scan, which is used by the dynamic mapping module internally. But is is also published and can be visualised by rviz.

The navigation stack publishes the velocity and driving orientation at the *cmd_vel* topic. As an alternative the velocity and orientation command can also be published by the keyboard control module. In that case the navigation stack is not needed at all. The safety module checks the bumper state and only lets the *cmd_vel* pass, if it is valid (see Section 5.7.2). Additionally it can also check the laser scan and determine, if there are objects in front of the robot. Finally the *RosAria/cmd_vel* topic is published by the safety module, which is subscribed by the RosAria module, which actually controls the robot hardware.

Overall the described modules, transformation between different topics are published by different modules. Some modules publish different parts for the global transformation tree. The global transformation tree is described in Section 5.9.

5.2.1 Classification of modules

The modules shown in Figure 5.1 were not all developed for this thesis. Some of them already existed and some were developed. In Table 5.1 a classification is shown between modules, which were completely new developed in this thesis (first column), those which already existed before, but where changes in configuration files or little adaptation in the source code were made (second column), and those which already existed and were used without any changes (third column).

The detailed changes which were necessary for the modules in the second column are described in the respective sections.

5.2.2 Multiple machines

As described in Section 4.4, ROS modules can run on different hosts, because of their TCP/IP communication. This is an interesting opportunity, if the robot has limited processing power and a lot of ROS modules should run simultaneously. In case of dynamic mapping, the processing

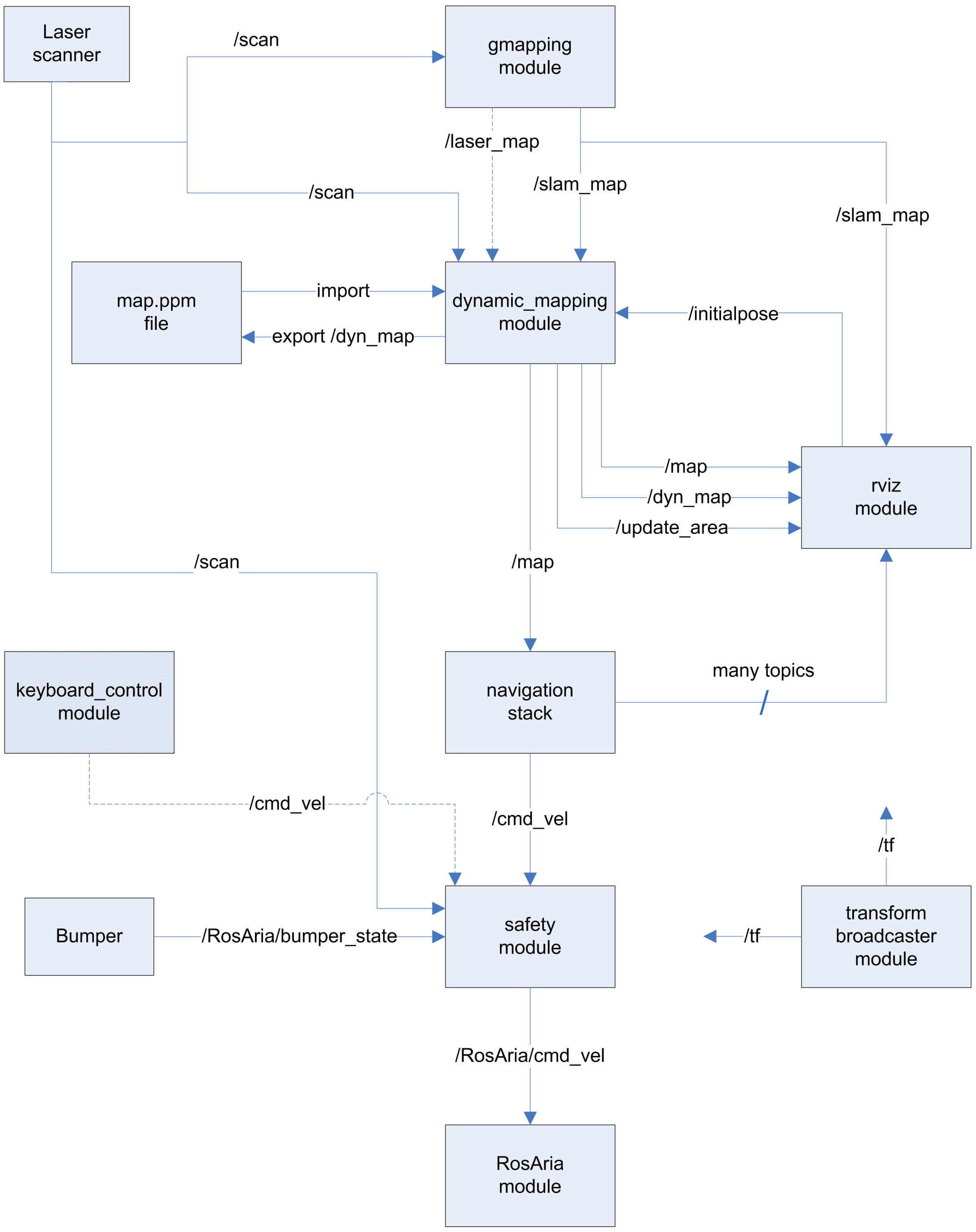


Figure 5.1: Software architecture / ROS intercommunication

Developed modules	Existing modules (changed)	Existing modules (unchanged)
dynamic mapping	laser scanner	RosAria
transform broadcaster	gmapping	
saftey	navigation stack	
keyboard control	rviz	

Table 5.1: Classification of existing and developed ROS modules

of maps is calculation intensive, so I decided to run these modules on a desktop host. The connection between the robot and the host PC was established over WLAN ad-hoc mode.

When ROS is used over multiple machines the following points must be considered (see [8]):

- The environment variable ROS_MASTER_URI must be set to the host, where the roscore module, the main ROS module which handles the intercommunication between different modules, is running.
- The /etc/hosts file of each host must contain the name and IP address of the other hosts.
- The environment variable ROS_IP must be set to the own IP address of each host.

5.3 ROSARIA

For accessing the hardware of the robot the package ROSARIA is used. The driver libraries for the robot are named ARIA and are written in C++. They allow to set velocity commands and read different sensor data from the robot hardware. ROSARIA is a wrapper module, which integrates these drivers into the Robot Operating System (ROS) framework.

The following topics are supported by ROSARIA:

- *cmd_vel:* Set the velocity and driving angle of the robot.
- *pose:* Read the current position of the robot (relative to its starting point).
- *bumper_state:* Read the bumper state, to get an information if the robot touches an object.
- *sonar:* Read the values of the sonar sensors.

An important point before starting this package is to set the *port* parameter to the device, where the communication between the robot hardware and the computer, located on the robot, takes place. In case of the robot P3AT this parameter was set to */dev/ttyS0*. For more information about ROSARIA see [7].

5.3.1 Laser driver

A laser scanner is mounted on top of the robot. This laser driver is not part of the ARIA driver library and hence not part of the ROSARIA wrapper. So there is another driver needed for this purpose. We decided to use the driver from here [33]. It is named LSM100, according to the name of the laser scanner. It processes the laser data and publishes a *scan* topic, which is used by the gmapping module.

The wrapper publishes an array of distance values within the *scan* topic containing 540 elements. These are caused by a scanning range of 270°and a resolution of 0.5°.

I changed the *host* parameter directly in the source code to 192.168.0.1 as it is the IP address of the computer within the robot and will not change at all. If this module should be used for other robots, this can also be changed by command line parameter. The module is called by:

```
rosrun LMS1xx LMS100 [_host:=192.168.0.1]
```

5.4 Gmapping

The gmapping [17] package is a Simultaneous Localization and Mapping (SLAM) package, which takes the laser scan data and generates a map out of it. It is a Robot Operating System (ROS) wrapper for the *openslam gmapping* [25] algorithm. As already described in Section 4.4.5 it is based on a rao-blackwellized particle filter. In Figure 5.2 a map generated by the gmapping algorithm is shown.

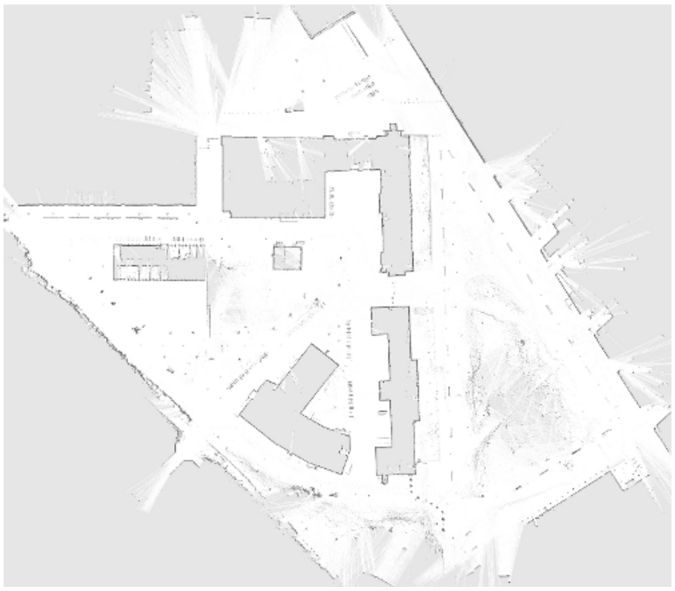


Figure 5.2: Gmapping generated map [17]

Gmapping comes with a lot of configuration parameters, the ones which were set are:

- Parameters to set the map size (*xmin, xmax, ymin, ymax*) (command line param) [1]
- The *delta* parameter to set the resolution of the map (command line param)
- The *maxUrange* parameter to set the maximum laser range distance, which should be used for mapping. Laser scans which are further away will not be used for mapping. (command line param)
- The *linearUpdate* parameter determines how far the robot must drive to trigger another laser scan. We set it to -1.0m (sourcecode)[2]
- The *angularUpdate* parameter determines how far the robot must rotate to trigger another laser scan. We set it to -1.0 radiants. (sourcecode)
- The *temporalUpdate* parameter determines how much time must go by to trigger another laser scan. This parameter was set to -1 as default, hence no temporalUpdate was made. We set it to 2s. (sourcecode)
- The *publishLaserMap* parameter determines if the laser_scan topic is published. If this parameter is not 0, the topic is published. Default value 1. (command line param)

These parameters have a big impact on the computational effort. As will be shown later it is necessary to iterate through the map in different processing steps. I did the experiments with a resolution of 0.02m and a map size of 20m x 20m. A new map update is triggered every 2 seconds. The *linearUpdate* and *angularUpdate* parameters where disabled to get a new map update in equidistant points in time.

The standard gmapping algorithm publishes its map on the *map* topic. This was changed to publish on the *slam_map* topic, the reason for this is that the navigation stack subscribes to the *map* topic and we would like to publish this at the dynamic mapping package.

We call this module by:

```
rosrun gmapping slam_gmapping xmin:=-10.0 xmax:=10.0 ymin:=-10.0 ymax:=10.0
                          delta:=0.02 maxUrange:=5
```

Of course we can add other command line parameters here or call it without any parameter, then all parameters use the default values.

5.4.1 Map format

The map which is published by the gmapping module is of the data type OccupancyGrid. It has a header with meta information about the map (e.g. the resolution) and besides the header there is an array of data contained in this data type. In this data array all the pixels of the map are saved, where each pixel has one of the following values:

-1: This pixel is unknown, hence the area in the map was not yet visited respectively discovered.

[1] These configuration parameters were changed as command line parameters.

[2] These configuration parameters are changed directly in the source code.

0: This pixel in the map is a known pixel with free space (no obstacle at all).

100: This pixel in the map is a known pixel with an obstacle.

The internal probability of each pixel to be an obstacle or not is compared to thresholds for determining, which are seen as obstacle and which as free. This is how we get the values 0 or 100. These maps are suitable for navigation and autonomous driving, but not for our needs.

The output of the SLAM gmapping algorithm was changed to publish not only 0 or 100 for known pixels, but probability values in between. The OccupancyGrid map type is designed to carry this information. The gmapping algorithm internally calculates with these probability values as described in Section 4.1.1, but does not output these values by default.

The algorithm was changed to publish probability values between 0 (free) and 100 (occupied) for known space and -1 for unknown space at the *slam_map* topic.

The probabilities of an object change slowly if the object was already scanned often by the laser scanner, as described in Section 4.1.1. This is because of the probability calculation, which is used by gmapping internally. The change rate calculation is based on the probability values from gmapping. If the probability changes slowly, we might miss dynamic behaviour. It could be that an object disappears and appears a view seconds later, but the probability did not change. In this case the change rate would also be left unchanged, which is incorrect.

To avoid this disadvantage of the gmapping probability calculation, an additional topic named *laser_map* was implemented. This topic contains a map, which contains only the map caused by one laser scan. It makes no sense to use probability values in this map, because with one scan there can only be binary values. Objects are there at certain positions in the map, or they are not there.

This map contains the following values:

0: This pixel in the map is a known pixel with free space, or an unknown pixel.

100: This pixel in the map is a known pixel with an obstacle.

5.5 Dynamic mapping

The dynamic mapping module is the main Robot Operating System (ROS) module within this project. It is the module which searches for dynamic areas in the *slam_map* topic. It is the module which publishes the *map* and *dyn_map* topics, it is the module which exports the dynamic map and it is the module which imports dynamic map files. All of the mentioned points are described in this chapter one after the other. Additional important aspects about this module are also explained.

5.5.1 Existing map processing in ROS

In ROS there exists the possibility of exporting the *map* topic into a file. This file is saved in the Portable Grey Map (PGM) format. The module which performs this task is named *map_server*. The PGM format is, as its name expects, a grey valued map. Each pixel of the map is saved in one byte. The values of this byte are in the range of 255 (white) to 0 (black). The values in between are different grey values. White corresponds to free space and black to an obstacle. The values in between are probabilities about the existence of an obstacle.

The *map_server* module can also do it the other way round, it can load a PGM image and publishes it as *map* topic. By doing so we can use this instead of gmapping as map provider for the navigation stack, if the whole map is already known and imported in this way.

The disadvantage is that it is not possible to generate a map by the gmapping module, export it with the map_server module to a PGM file, reimport it with the map_server later and start to expand the map by gmapping. Assume we have a big building and want to generate a map from it. When we generate a map of just a part of the building and export it, we do not have the chance to load the map again and change or even expand the map with gmapping later. We can only do two things, load the existing map and navigate the robot within it, or start over with the generation of a new map. Whenever we try to build a really big map we will fail to do so, because it is rather impossible to generate it in only one step without stopping the mapping. But why is this the case and why is there no solution for this so far? There exist a few problems which relate to the import and the wish to expand the map.

- The first problem is that we do not know, where the robot is located in the imported map. But to expand it we clearly have to know where it is. So the first action would be to locate the robot within the imported map.
- The second problem is that we have to merge the imported map and the map generated by gmapping. An idea which comes up when we think about this is to export the internal state of the gmapping algorithm instead of just exporting the map topic. This would be possible, but the robot position is somehow contained in the internal state, so starting mapping on arbitrary map positions again leads to changes in the internal state. So there would still be the problem to locate the robot when we import the internal state again, or do not move the robot between exporting and importing of the internal state. But this is also not possible every time and is an undesired solution. So the imported map and the new map provided by the gmapping module would have to be merged.

With these two main problems there comes a few more as will be clear later. These are the reasons why this has not be done until now. As we will see in the following section, I tried to find a solution for this and will explain the basic actions I had to do to handle these problems.

5.5.2 Introducing a new message type

Message types define the structure of the communication over topics as described in Section 4.4. The present message type for maps is named *OccupancyGrid* and consists of the following data types:

- Header
- nav_msg/MapMetaData
- int8[] data

In the Header type there are variables like the frame ID or the time stamp located. The MapMetaData contains meta data about the map, for instance the resolution and dimensions of the map. The data Section is one big array of data values each one with a value between -1 and 100, representing the probability of the existence of objects. Each pixel in the map corresponds to one element of the array, where the top left pixel of the map is the first element of the array and the bottom right pixel of the map is the last element of the array.

Now a new topic needs to be defined to also contain the dynamic area information. There are two possibilities to do so. The first one is to publish an additional topic, which contains the dynamic area information, the second one is to use the existing message type and extend it. I decided to implement the second one, because we might get into troubles when the two topics are not synchronized. Additionally the whole information about the dynamic map can be gained by only subscribing to one module. I took the *OccupancyGrid* message type and expanded it by an additional array called *dynamic*. In this array the *exponential moving average time until change* will be stored for each pixel of the map.

Additionally a global reference value for all *dynamic* values of the whole map is stored as *dynamic_time_max*. This value contains informations for the interpretation and visualisation of the *dynamic* values. When visualising the *dynamic* values as different red colors, this value can help as reference value, which is described in detail in Section 5.5.3.

The message file, from which the message type is constructed by ROS, is visualised in Listing 5.1. The *dyn_map* topic is of the type *DynamicGrid* and contains additional information in contrast to the former *map* topic. The *dynamic_mapping* module publishes both, the former *map* topic and the new *dyn_map* topic. The *map* topic is used by the navigation stack and the *dyn_map* topic for rviz and the map export.

5.5.3 Exporting and importing maps

As already mentioned in the introduction, we do not just want to expand existing maps, but also want to find dynamic areas within these maps. These dynamic areas should also be part of the exported map to make it possible to use them later, after importing the map file again. Until now grey valued maps were used as files for the export and import of maps in ROS. When information of dynamic areas should be included in these a problem arises. The information content of grey values is already used for the probability of the existence of obstacles. So we have to find a way to include additional information about dynamic areas. In general there are three possibilities to do so:

- Use vector graphics instead of pixel maps
- Use additional text files to add dynamic area information
- Use color maps

As already discussed in Section 2 I decided to use pixel maps.

As an alternative the files can be exported in grey values as described, and the information about dynamic areas can be saved in text files. We are free in choosing a format of how the information should be stored. We can choose any format which is reasonable. We can use a

```
# This represents a 2-D grid map, in which each cell represents the
  probability of
# occupancy.
# For each cell the dynamic value is also stored.

Header header

# MetaData for the map (e.g. resolution)
nav_msgs/MapMetaData info

# The reference value for dynamic area colorsiation
# This value will be the deepest red, all values above will also be deep red.
uint32 dynamic_time_max

# The map data, in row-major order, starting with (0,0).  Occupancy
# probabilities are in the range [0,100].  Unknown is -1.
int8[] data

# The exponential moving average time until change in milliseconds for each
  cell, starting with (0,0). max value 49 days
uint32[] dynamic
```

Listing 5.1: DynamicGrid.msg

value for each pixel, we can save the outer points of polygons or what ever. If we do so, we always have to import these two files again: the grey valued map file and the dynamic area text file. We have to make sure that only two compatible files are imported again.

Another point is that the dynamic areas are not visualised on the image file in that case, which would be nice to analyse them graphically.

The third possibility is to not only use grey valued maps, but use color maps. Here the grey values can be used for storing the information about the existence of obstacles and the red color can be used for storing dynamic areas. Three bytes Red Green Blue (RGB) color information needs to be saved instead of just one byte grey values. There is no need for the green and blue parts and there is an increase of the factor three of the image size. But the advantage is to see the dynamic areas directly in the image. The *exponential moving average time until change* needs to be stored to import it later on. But the exponential moving average time until change does not contain the time which past by since the last change. This information is stored in *dynamic_time*, so this information might also be useful at the import of the map. The third interesting variable to export is the *dynamic_trend*, which contains the actual sign of the deviation of the probability function over time (see Section 4.1.6). We decided that it should be possible to also export the *dynamic_trend*.

As a few variables should be exported, we decided to use structured YAML Ain't Markup Language (YAML) text files to save the dynamic information.

The *dynamic* value is exported always. If the *dynamic_time* and *dynamic_trend* should be exported is optional and can be set by the user.

What can also be set by the user is if the information about the location and *exponential moving average time until change* is additionally stored as red color in a color map file. This is only for visualisation purposes to get an overview of the location of dynamic areas, which is much better comprehensible by humans than text files. This is an optional feature, so the map can also be exported as grey valued map file without loosing information.

The extension from PGM grey value files to color files is the Portable Pixel Map (PPM) image format. Every pixel is saved in three bytes. One for the red color, one for the green color and one for the blue color part. As already mentioned, these files need three times more memory space than PGM files.

Exporting maps

Now that we know which files we are going to use (YAML and map file), we can take a look at the export procedure.

The dynamic map export module is called by:

```
rosrun dynamic_mapping map_export [-f filename] [parameters]
```

As default the map file is exported as *map.ppm*, a YAML file with meta information is also generated, named *map.yaml*. The default file name can be overwritten by command line parameter *-f filename*.

The following parameters can be set optionally for changing the behaviour of the map export:

grey_val: Export a grey valued pixel map without dynamic areas in red color if set to 1. Default value 0.

trend: Export the *dynamic_trend* variable from the dynamic mapping algorithm into the YAML file if set to 1. Default value 0.

time: Export the *dynamic_time* variable from the dynamic mapping algorithm into the YAML file if set to 1. Default value 0.

time: Export the *dynamic_time_max* variable from the dynamic mapping algorithm into the YAML file if set to 1. Default value 0.

If the command is called with the additional parameter *_grey_val:=1*, the map is exported in grey values without any dynamic areas. This may be interesting for existing tools, which work with these maps, as this feature makes the module backward compatible.

But now the focus lies on the export of dynamic maps. All necessary data is requested over a ROS service from the *dynamic mapping* module. The service definition is shown in Listing 5.2. The *init* variable is set by the service requesting map export and needs to be 1 to get any dynamic information at the response.

The response contains the probability informations in the *dynamic_map*, as well as the *dynamic_trend*, *dynamic_time* and *dynamic_time_max* (already described in Section 5.5.2).

At the exporting procedure we subscribe to the *dyn_map* topic, iterate over each point of the map and distinguish between the following cases:

```
# Get the necessary data to export the map
int8 init
---
# The dynamic map topic
dynamic_mapping/DynamicGrid dynamic_map

# The sign of the deviation of the probability function over time
int8[] dynamic_trend

# The time, which pasted by since the last change of a pixel
uint32[] dynamic_time

# The reference value for dynamic area colorsiation
# This value will be the deepest red, all values above will also be deep red.
uint32 dynamic_time_max
```

Listing 5.2: ExportMap.srv

- If the map point is unknown, export a light blue pixel $(R, G, B) = (200, 200, 255)$
- If the map point is known and the dynamic value is 0, export a pixel which has a grey value according to the *data* probability. This data probability is a percent value, hence between 0 and 100, where 100 is an obstacle. The color values are in the range from 0 to 255, where black (0) is an obstacle. The color value was calculated according to Equation 5.1 and the pixel was set to $(R, G, B) = (color_i, color_i, color_i)$.
- If the map point is known and the dynamic value at this certain position is greater than 0, a pixel which has a red value according to the *dynamic* probability is exported. The probability value of the *dynamic_map data* array needs also to be saved. The *dynamic_time_max* value is used to scale the dynamic value to get a red pixel as in Equation 5.2. The data value is decreased to support the red pixel (in the worst case it could be black instead). This is done as shown in Equation 5.3. With this steps we can be sure that the exported map has a red color at dynamic areas and that it is possible to restore the data value at the import of the image file. Each pixel is set to $(R, G, B) = (red_i, green_i, blue_i)$.

With these three cases it is possible to unambiguously restore the data value from the image at the import procedure. In Figure 5.3 an example export of a dynamic map is shown. In this figure the red dynamic areas are well visible.

$$color_i = ((100 - data_i) * \frac{255}{100} \tag{5.1}$$

$$red_i = min((100 - \frac{dynamic_i * 100}{dynamic\_time\_max}) * \frac{255}{100}, 255) \tag{5.2}$$

$$green_i = blue_i = \frac{((100 - data_i) * \frac{255}{100}}{5} \tag{5.3}$$

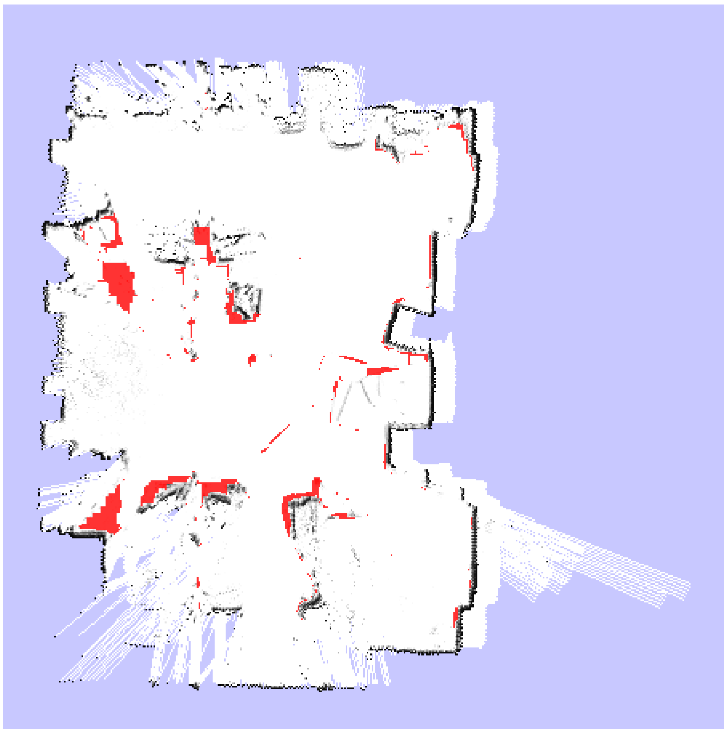


Figure 5.3: Dynamic PPM map file after export

As already mentioned it is also possible to export the file without its dynamic areas. This is done by adding *_grey_val:=1* to the command line call. If this is done, the information of one pixel is saved in just one byte and this is done by the following equation:

$$grey_i = \begin{cases} 0 & \text{if } data = -1 \\ 254 & \text{if } data = 0 \\ data + 100 & \text{otherwise} \end{cases} \tag{5.4}$$

With Equation 5.4 it is possible to unambiguously restore the data value at the import procedure. The different zones in the map file of Figure 5.4 are also visible, but there are no red dynamic areas in comparison to Figure 5.3.

In both cases (color map or grey valued map), the variables of Listing 5.2 are saved to the YAML text file (depending on the command line parameters) like an example YAML file in Listing 5.3 shows. As we can see not only the data generated by the *dynamic mapping* module is stored in the YAML file, but also detailed information about the map. This detailed information is produced by gmapping and contained in the MapMetaData Section of the *dyn_map* topic (see Listing 5.1). The *image* parameter contains the corresponding map file for example.

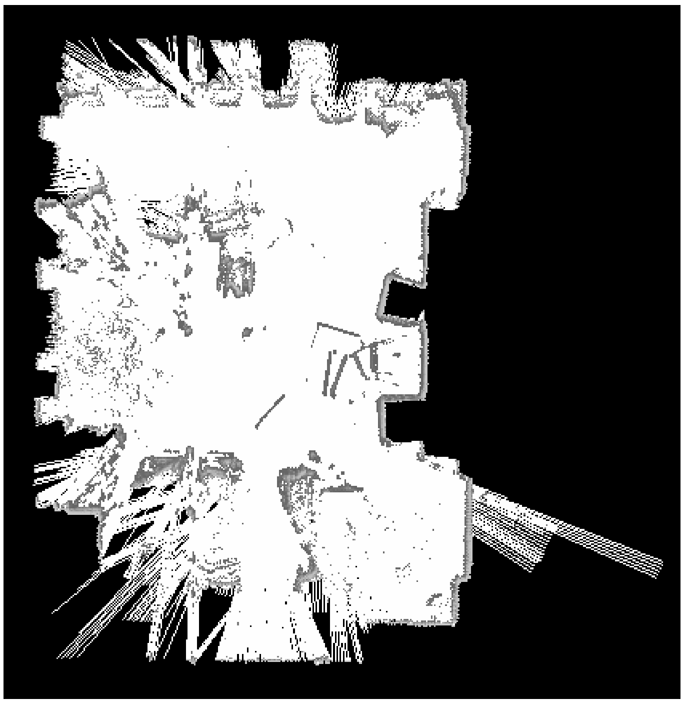


Figure 5.4: Grey valued PGM map file after export

```
image: map.ppm
resolution: 0.020000
origin: [-10.000000, -10.000000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
map_array_size: 10
dynamic: [0,5,0,10,10,10,15,0,0,0]
dynamic_trend: [-1,0,-1,0,0,1,1,-1,-1,-1]
dynamic_time: [0,5,0,5,5,0,10,0,0,0]
dynamic_time_max: 600
dynamic_map: true
```

Listing 5.3: map.yaml

Importing maps

The file import for grey valued PGM files already exists in the *map_server* module. I took this as a basis for the import procedure of PPM files. In general a PGM or PPM file is imported and stored as internal data structure in the *dynamic_mapping* module, where it waits for the appearance of the *slam_map* topic, published by the gmapping Simultaneous Localization and Mapping (SLAM) module, to be merged with it and then published as *dyn_map*. The dynamic area information of the YAML file need to be filtered out and the *dynamic* array of the

DynamicGrid message type has to be restored. If the YAML file contains *dynamic_trend* and *dynamic_time*, they are restored as well. Otherwise a default value is used instead.

At the PPM file import we iterate over each pixel of the PPM image and check, which color part is the dominating one. Then the reverse as at the export procedure is done and the values are reconstructed in that way.

$$data_i = \begin{cases} -1 & \text{if if blue color is dominating} \\ 100 - red_i * 5 * \frac{100}{255} & \text{if red color is dominating} \\ 100 - red_i * \frac{100}{255} & \text{otherwise} \end{cases} \tag{5.5}$$

With these equations the values of the *data* array of the *dyn_map* topic are restored.

The dynamic information is restored directly from the YAML file so the dynamic map is ready for publishing.

If a PGM file should be imported, then this can be done by importing it without any parameter settings. The file is automatically recognized as PGM file at the import procedure. The values of the data array are restored according to Equation 5.6.

$$data_i = \begin{cases} -1 & \text{if } greyvalue = 0 \\ 0 & \text{if } greyvalue = 254 \\ data - 100 & \text{otherwise} \end{cases} \tag{5.6}$$

What also need to be mentioned is that the YAML file contains information about the map width and height and is important to interpret the values of the image file. This file is named at the start of the dynamic mapping module, if a given map should be imported.

5.5.4 Merging algorithm

As described in the previous section the imported map (a given map to start with) and the slam map (map generated by gmapping) needs to be merged. As a prerequirement both maps need to have the same resolution, otherwise the map merge will fail. For merging the maps the knowledge of where the slam map has its origin, with respect to the imported map, is required. This is the same as to know at which position the robot starts in the imported map, because the origin of the slam map is at the same position, where the robot starts (as it is generated by the robot). So in general this is a localisation problem. We need to know where the robot starts with respect to the imported map, to be able to merge the maps.

This is a vital problem and there are three ways to handle this.

1. Trying to match the slam map and the imported map by some image processing algorithms. In detail trying to find the image position of the slam map within the imported map.

2. Performing a particle filter algorithm to locate the robot within the imported map

3. Knowing the position and set it correctly

The first possibility is difficult and calculation intensive. The problem is not only to match the images, but to know the orientation of the images. So if we do a straight forward approach we would have to check each orientation of the two images with each possible overlap and choose the most probable. I tried this, but there is no chance to do it in a few seconds. With big maps there is no chance to do it within minutes with the calculation power of the robot P3AT (without a Graphics Processing Unit (GPU)).

Two possibilities are left and either of them can be used as will be shown shortly. What I implemented is that the merging algorithm waits for an *initialpose* topic to be published. The content of this topic is interpreted as the initial position of the robot within the imported map. Now that we know the initial pose we can merge the maps.

Two things can be done, the first thing is to use any localisation algorithm and wait until the pose estimation within the imported map is good enough. If this is the case the initial position of the robot is published at the *initialpose* topic and the map merging starts.

The other possibility is to use the *2D pose estimation* button of rviz and set the location of the robot this way, if it is known. Rviz also publishes an *initialpose* topic in behind.

Before the maps are merged, the slam map needs to be rotated first. The orientation of the *initialpose* topic has to be considered and the slam map needs to be rotated according to its orientation. The image rotation algorithm is done in the following way. We have two map instances, the unrotated map and the rotated map. Both have the same size and resolution. We iterate over each pixel of the empty rotated map from the top left to the bottom right. For each pixel we calculate, which pixel from the unrotated map fits there.

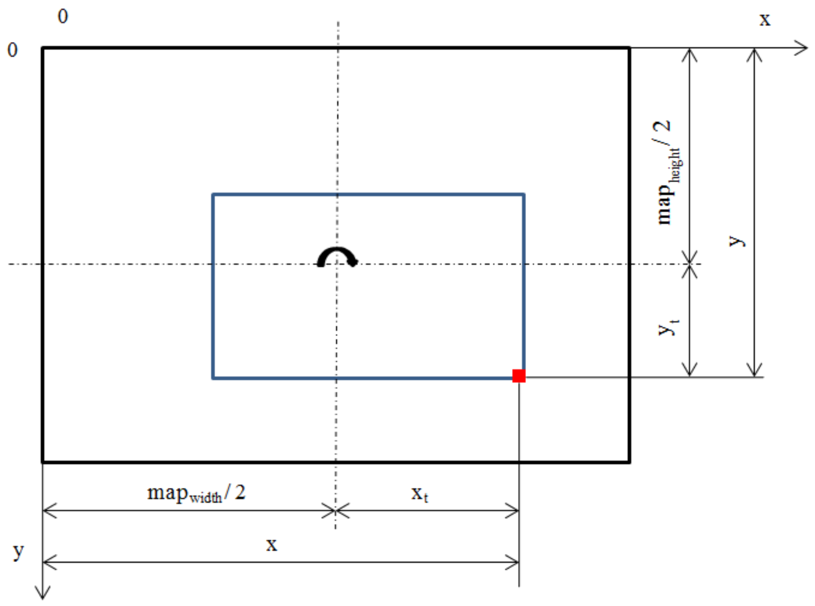


Figure 5.5: Map after rotation

In Figure 5.5 we assume to already have iterated over big parts of the map and have drawn the blue rectangle (which we assume as room in the map) until we reached the red pixel. We

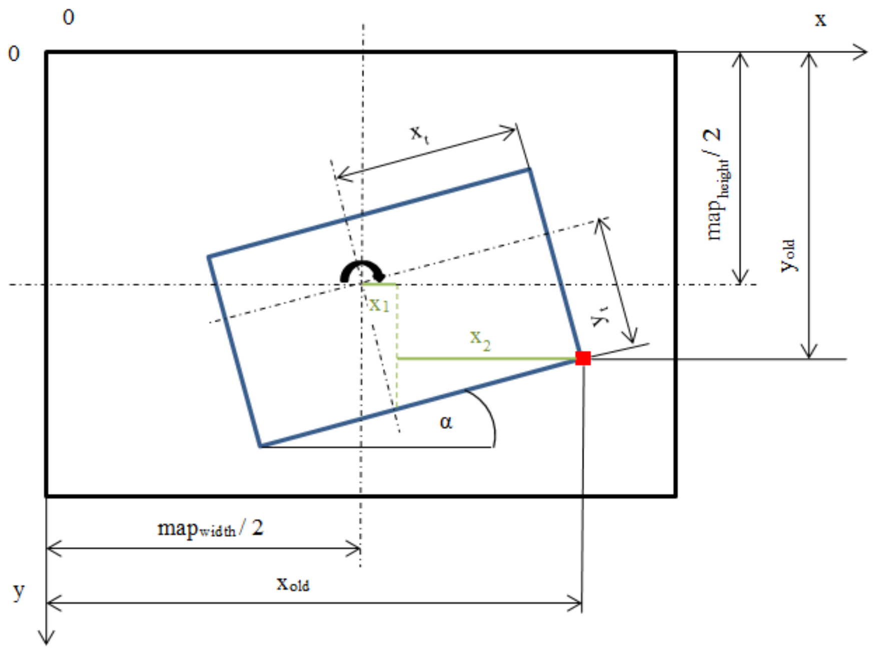


Figure 5.6: Map before rotation

show how the rotation is done for this specific pixel as an example. In addition this procedure can be used for every other pixel. In the iteration loop we are now at $map_{rotated}[x][y]$ and need to find the corresponding red pixel in the unrotated map as we can see in Figure 5.6. Here the red pixel is at the position $map_{unrotated}[x_{old}][y_{old}]$. What is left now is the calculation of x_{old} and y_{old}, given the rotation angle α.

This calculation is done by the following equations:

$$x_{old} = round(\overbrace{x_t * cos(\alpha)}^{x_1} + \overbrace{y_t * sin(\alpha)}^{x_2}) + \frac{map_{width}}{2} \tag{5.7}$$

$$y_{old} = round(y_t * cos(\alpha) - x_t * sin(\alpha)) + \frac{map_{height}}{2} \tag{5.8}$$

These equations follow directly from Figure 5.6, where x_1 and x_2 are visualised as in Equation 5.7. For y_{old} it is done in a similar way. The values $\frac{map_{width}}{2}$ and $\frac{map_{height}}{2}$ are needed because x_{old} and y_{old} are not calculated relative to the center of rotation, but absolute from the top left corner of the map.

What has to be considered is that at the corners of the map pixels will be lost during the rotation process. This can be avoided by taking a bigger map for the rotated image. In Figure 5.7 the two possibilities are shown. The orange map is the rotated one, while the black one is the unrotated map. I decided to use the method shown at the left, where the corners are cut away. Otherwise it would be necessary to handle a much bigger map with a lot of free space added. For our needs this method with cutting away edges is appropriate.

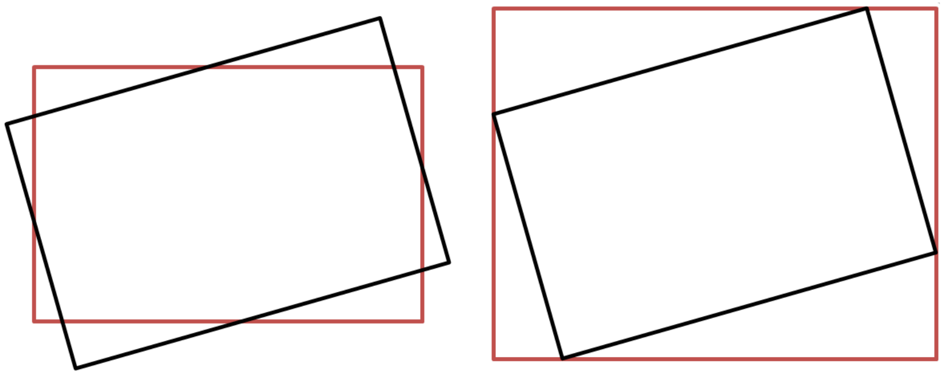


Figure 5.7: Corner cut at the rotation

Now the rotated slam map and the imported map need to be merged. We know where the origin of the slam map is with respect to the imported map from the *initialpose* topic. The problem now is to find the starting point and end point for the merging, because the slam map may not be contained in the imported map with the whole area, as visualised in Figure 5.8. Here the black rectangle is the slam map and the orange boarder the imported map. We see the point (x_{pos}, y_{pos}) of the imported map, where the slam map has its origin. And we can see the starting point of the slam map $(x_{start,s}, y_{start,s})$, as well as the end point of the slam map $(x_{end,s}, y_{end,s})$. The starting point of the imported map seems to be at the same position as the starting point of the slam map, but this is the point with respect to the imported (orange) map and is located at $(0, 0)$ in this case. Finally the endpoint of the imported map is located at $(x_{end,i}, y_{end,i})$. The equations for the starting- and endpoints of the imported map are shown in Equations 5.9 to 5.12 and the starting- and endpoints of the slam map in Equations 5.13 to 5.16. The light blue area of the slam map needs to be merged into the imported map.

$$x_{start,i} = max(x_{pos} - \frac{map_{width,i}}{2}, 0) \tag{5.9}$$

$$y_{start,i} = max(y_{pos} - \frac{map_{height,i}}{2}, 0) \tag{5.10}$$

$$x_{end,i} = min(x_{pos} + \frac{map_{width,s}}{2}, map_{width,i}) \tag{5.11}$$

$$y_{end,i} = min(y_{pos} + \frac{map_{height,s}}{2}, map_{height,i}) \tag{5.12}$$

$$x_{start,s} = max(map_{width,s} - x_{end,i}, 0) \tag{5.13}$$

$$y_{start,s} = max(map_{height,s} - y_{end,i}, 0) \tag{5.14}$$

$$x_{end,s} = min(x_{start,s} + window_{width}, map_{width,s}) \tag{5.15}$$

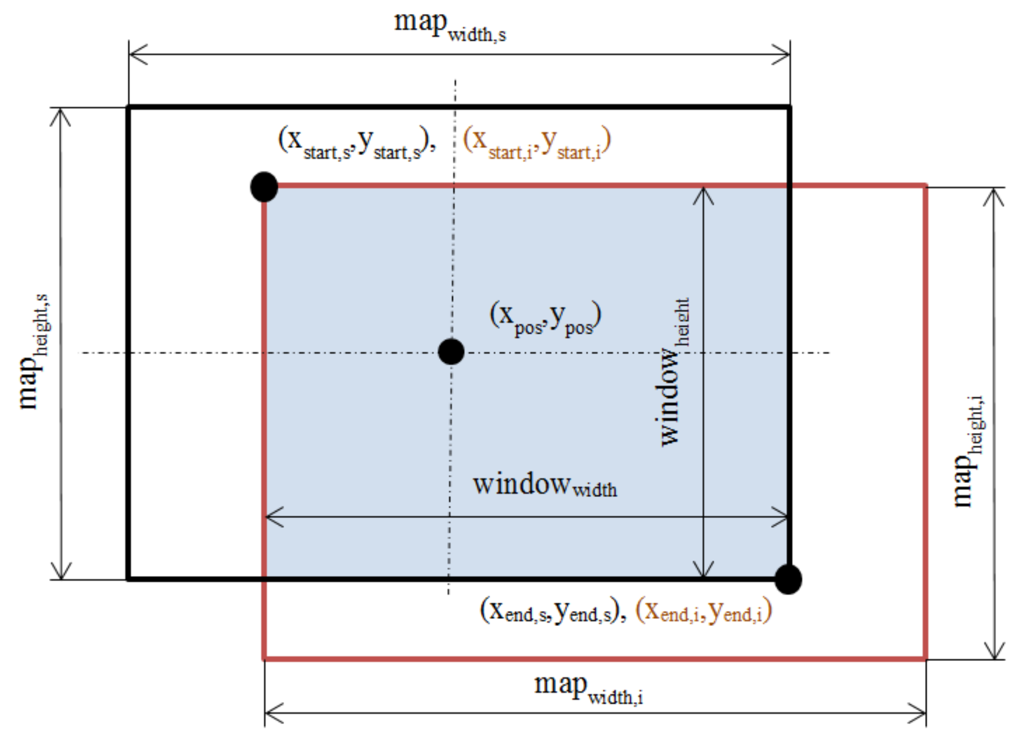


Figure 5.8: Relevant points for map merging

$$y_{end,s} = min(y_{start,s} + window_{height}, map_{height,s}) \tag{5.16}$$

The whole merging procedure is done in the following way. Merge each pixel of the imported map and the slam map into a new map called dynamic map. The data from the imported map is more trustful than the slam map, because the robot might have scanned objects a few times and not just once as in the slam map. The maps are merged by taking 80 percent of the value of the imported map and only 20 percent of the value of the slam map for each pixel, the proportion 80:20 between the two maps can be changed by command line parameter.

After that, the dynamic map is updated in a similar way whenever a new slam map was published. So the dynamic map always contains the current slam map data.

Now the merged dynamic map is published on the *dyn_map* topic for the first time. What is also done at the merging procedure is to find and update dynamic areas, but this is described in the next section.

5.5.5 Finding dynamic areas

Within this part the main topic is how dynamic areas can be found and how they are processed. As already described the *slam_map* topic is published by gmapping and used for determining dynamic areas. A chance to get this kind of information from maps is by comparing different maps generated at different time stamps. The *slam_map* topic already contains probabilistic information figured as values between 0 and 100. The gmapping algorithm saves probabilities for the existence of objects at certain positions. These probabilities are influenced by inaccurate laser scan measurements, moving objects or inaccurate localisation of the robot. Gmapping updates the probabilities each time the robot comes across and performs a new laser scan of a certain object. These probabilities can be used to perform filtering and calculate the change rate of moving objects. Additionally other types of information can be used too. The dynamic map contains information about all laser scans and all slam maps of the past. The last generated slam map and the dynamic map can be compared. The rules of the dynamic mapping approach discussed in Section 4.1.5 were implemented.

Gaining and saving change rates of obstacles

As discussed in Section 4.1.2, the change rate is saved as exponential moving average time until change for each pixel. The necessity of knowing extrema of the obstacle probability over time was also described.

Now these extrema need to be found, which is done in the following way. At the map merge each pixel of the slam map is compared to the corresponding pixel of the dynamic map. If they are different a *dynamic_trend* variable is changed. When considering pixel i and assuming the probability value of the slam map at position i is greater than the probability value of the dynamic map at position i, *dynamic_trend[i]* is set to 1. This means a rising probability trend. If the probability value at position i of the slam map is lower than the probability value at position i of the dynamic map, *dynamic_trend[i]* is set to 0. If the values are equally, *dynamic_trend[i]* is not changed.

What was stored now is an information of the sign of the gradient of the probability function over time.

What is done now, is to compare the current value of *dynamic_trend[i]* with the previous one. If they differ, an extrema was detected. In this case dynamic behaviour is assumed and the change rate is calculated. The *dynamic_trend[i]* is calculated in the following way, where $data_i$ is the probability value:

$$dynamic\_trend[i] = \begin{cases} -1 & \text{if } slam\_map\ data[i] = -1 \\ 1 & \text{if } slam\_map\ data[i] > dynamic\_map\ data[i] \\ 0 & \text{if } slam\_map\ data[i] < dynamic\_map\ data[i] \\ dynamic\_trend[i] & \text{otherwise} \end{cases} \tag{5.17}$$

The dynamic value itself is calculated in the following way. The first step is to measure the time between consecutive extreme values of the probability value for each pixel, by considering *data[i]*.

For every pixel a *dynamic_time[i]* variable is available, which contains the time since the last extreme value at data[i] over time in milliseconds.

So if an extreme value occurs in the *data[i]* over time, *dynamic_time[i]* contains change time. To get an exponential moving average time until change, the previous *dynamic[i]* value is also considered. Over the parameter *weight* can be set how much *dynamic_time[i]* counts with respect to the historical *dynamic[i]* time value. The default value is $weight = 0.8$.

If an extreme value was detected, the variable *dynamic[i]* is calculated in the following way:

$$dynamic[i] = \begin{cases} dynamic\_time[i] & \text{if } dynamic[i] = 0 \\ dynamic[i] * weight + & \\ dynamic\_time[i] * (1 - weight) & \text{otherwise} \end{cases} \tag{5.18}$$

The time variables are of the data type *uint32*. This makes it possible to store large values. In case of saving milliseconds the maximum value is about 49 days. This should be enough for determining dynamic areas.

Gaining and saving change rates of obstacles based on the laser_map topic

As already mentioned the probability values of objects, which where observed for a long time, change slowly. With this slowly change the problem is that an object may disappear and appear a few seconds later, but the probability did not change. The change rate is depending from the probability, so the change rate will also not change. This is incorrect behaviour and caused by the probability calculation in gmapping. Other SLAM modules for ROS may use other ways to calculate their probabilities, where this problem does not occur.

In case of gmapping we decided to use the *laser_map* topic as second opportunity to calculate the change rate. In this case the value of *dynamic_time[i]* is calculated in the following way.

Each *laser_map*$_n$ is compared to the previous *laser_map*$_{n-1}$. If an object was detected at *laser_map*$_{n-1}$, the value of *laser_map* at this position is 100. If the object disappeared, the *laser_map*$_n$ has the value 0 at this position. So the change rate can be calculated by comparing the *laser_map* topics obtained in different points in time. There are no extrema values necessary.

The problem is that the described comparison of *laser_map* topics can be influenced by laser measurement errors. The laser measurement might detects the same pixel in two consecutive scans one time at occupied and one time as free although no object change took place. This is the reason why we decided to not only compare the pixels with the same position in different *laser_maps*, but to also look at the neighbour pixels. So only if a pixel in the *laser_map* was detected and hence hs the value 100 and the same pixel and its neighbour pixels in the consecutive *laser_map* have the value 0, the change rate is updated.

Known slam map, unknown dynamic map

This is the case, if a new area within the slam map was discovered, which was not discovered so far and hence not known in the dynamic map.

There is no *dynamic_trend[i]* value available now, but it is set for the first time. As the area is unknown in the dynamic map, the *dynamic_map data[i]* value is -1. The *slam_map data[i]* value is greater than -1, because it is a known area now. Hence according to Equation 5.17 the *dynamic_trend[i]* is set to 1. But this can not be used for checking for extrema now, because no reference value is available as it is the first time *dynamic_trend[i]* is set.

Known slam map, known dynamic map

In this Section the case of a known area in the slam map and also knowing the same area in the dynamic map is considered. As already mentioned in Section 4.1.5, the same procedure is also used for the case *Unknown slam map, known dynamic map*. If this is the case the following points have to be fulfilled to count the area as dynamic one:

- The variable *dynamic* which contains the *exponential moving average time until change* is greater than 0 and less than a configurable upper bound (*dynamic_time_max*).
- The area around the current pixel is also known

A new laser scan is triggered every 2 seconds, hence a slam map is available ever 2 seconds. This causes a map comparison every 2 seconds. For each pixel, which is actually in the laser scan area, the *dynamic_time[i]* value is increased by 2000, because milliseconds are stored.

If an extreme value was detected, the *dynamic[i]* variables changes according to Equation 5.18

For every pixel, which is not within the laser scanned area, the *dynamic_time[i]* value is not changed, because we do not have any information about what happens there at the moment - so there can not be made any statement about the dynamic behaviour. The *exponential moving average time until change* should only be influenced if the relevant area is scanned. The *dynamic_time[i]* stands still for these pixels.

The second point avoids dynamic areas at the edges of the known area. Measurement errors misinterpreted as dynamic area should be prevented. The area around an unknown pixel, where no dynamic value is allowed, has the shape of a rectangle. The width and height are equal and can be set by command line parameter.

Dilation filtering

The dynamic areas were calculated pixel by pixel without any impact of the surrounding dynamic pixels until now, only the value of the same pixel in previous maps was considered. This can lead to holes in dynamic areas, which can also be caused by measurement errors. Another point are small dynamic areas, which are caused by human walking. They are very small but many, if humans walk the same paths in a room often.

To merge these small dynamic areas to bigger dynamic areas as well as to fill holes a dilation filtering is performed. This means that additional pixels are marked as dynamic area.

In detail this is done in the following way. We iterate over all pixels of the known dynamic map and do the following for each one. Assume we reached pixel *i* in Figure 5.9 of the dynamic map in our algorithm. Here we see the *dilate_size* parameter, which can be set by command line parameter (default value 7). In this example we use a *dilation_size* of 5, because it has no impact on the structure of the algorithm and is easier to understand. The *dilation_size* parameter is the width and height of the window for the dilation filter. What we do now is to check, if there is a dynamic pixel in the yellow area above the pixel *i*. If yes we save the position (respectively how far it is away from *i*) and call it *A*. We also search for a dynamic pixel in the yellow area below of *i*. If there is also a dynamic pixel, we call it *B* and set all pixels in between *A* and *B* to dynamic pixels with the dynamic value $dynamic = \frac{dynamic_A + dynamic_B}{2}$. A similar procedure is done with the orange pixels, here we simply change the orientation, but the algorithm stays the same in principle.

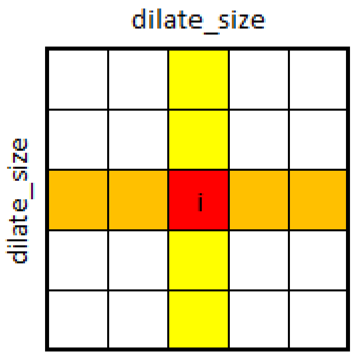


Figure 5.9: Dilation filter algorithm

Now one could think of doing this procedure also for diagonal neighbours of pixel *i*. But this is not really required, if there are pixels, which are diagonal neighbours it is likely to connect them with the shown algorithm, if there are a lot dynamic pixels around. In that step the pixel *i* is ignored in our algorithm. But at the next run of the algorithm there are now more neighbours dynamic pixels and now pixel *i* might also become a dynamic pixel.

This dilation algorithm is executed every time after the dynamic map and the new slam map were merged.

Calculating the laser scanned area

When calculating the exponential moving average time until change, only the area which is currently scanned by the laser scanner should be considered. Because it makes no sense to increase the time of an area of the map, which is not scanned by the laser scanner at the moment. We do not know, what happens at the whole map at a certain point in time, so its the best to leave the dynamic areas as they are. But at the current scanned area the dynamic areas are updated as seen before.

The calculation of the current laser scanned area is done in equidistant time intervals (set by the parameter *time_period*). Then the *dynamic_time[i]* of all pixels within the currently scanned area are increased by *time_period*. By default *time_period* is 5000 milliseconds, which means that every 5 seconds *dynamic_time[i]* of all relevant pixels is increased by 5000.

The problem is that we need to know, which area is currently scanned by the laser. This information can not be gained from the *slam_ map* topic, but is provided by the laser scan.

The laser scan topic consists of 540 measurements (270°measurement angle and 0.5°resolution). Each of this 540 elements has a value, which corresponds to the distance d_i in meters of the next object at a certain degree. For further calculation we need to determine the x_i and y_i values of the point out of the angle and the distance, see Figure 5.10.

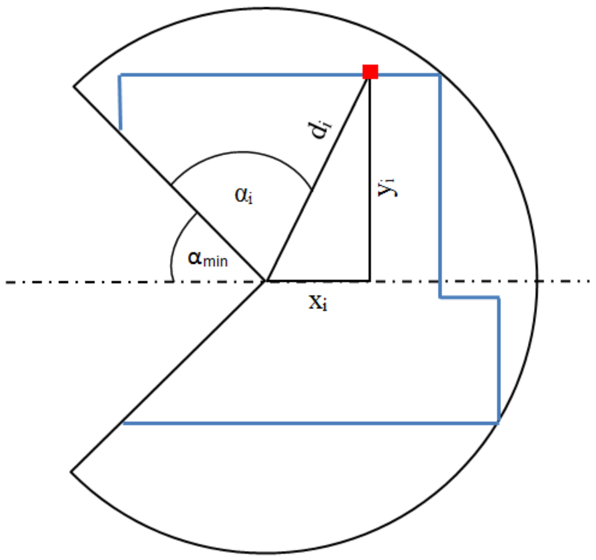


Figure 5.10: Laser scan area calculation

The blue area is the shape of actual laser scan, the black circle indicates that the laser scan is performed in 270°and shows the maximum scan range. Now the values of x_i and y_i can be calculated with the following equations i in $[0..539]$ and $\alpha_i = 0.5 * i$:

$$x_i = d_i * sin(\alpha_{min} + \alpha_i) \tag{5.19}$$

$$y_i = d_i * cos(\alpha_min + \alpha_i) \tag{5.20}$$

The points of the laser scan were calculated in (x, y) coordinates (Equations 5.19 and 5.20). Now each point is connected with its two neighbours (connect point x_i with x_{i-1} and x_{i+1}). The last point x_{max} is connected with $(0, 0)$ (robot position) and with x_0. By doing so we get a polygon with all points of the laser scan, extended by the point (0,0) which is the current position of the robot. This is necessary because the robot is unable to get laser scan data for the rear 90°. In Figure 5.11 the polygon is visualised in green color by the rviz polygon display plugin.

The calculation is done in the laser scan callback function, each time a new laser scan is available. After calculating the polygon, the polygon is published as *update_area* topic.

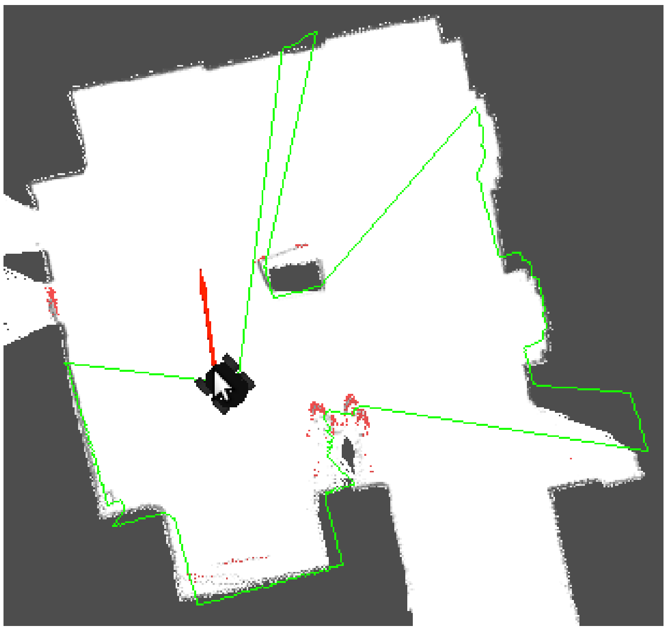


Figure 5.11: Green laser scan polygon

What is now left to do is to check for each known point of the dynamic map, if the point is inside the polygon or outside. If it is inside, the point is scanned by the laser scanner at the moment and its *dynamic_time* value is increased otherwise nothing is done.

To perform this task, the dynamic map needs to be transformed to the laser scan coordinate system. Then the points of the dynamic map coordinate system can be compared to the points of the laser scan coordinate system.

For determining if a point is inside the polygon or outside, an appropriate algorithm is used. The algorithm as well as a short description is shown in [18].

5.5.6 Dynamic mapping *main*

The issues discussed in the last sections were implemented in the dynamic mapping main file. The following concepts are contained:

- Import of a given map
- Merging slam map and dynamic map
- Finding dynamic areas
- Dilation filtering
- Calculating the laser scanned area

These features strongly depend on each other, such that it makes sense to concentrate them in one file. The whole module is called by:

```
rosrun dynamic_mapping main [_parameter1:=value _parameter2:=value ...]
```

In addition there are a few configuration possibilities, which can be set as command line parameters.

file_name: An existing map file can be imported at the beginning (PPM or PGM image). As default no file is imported and the dynamic mapping is performed based on the slam map only.

area_size: Determines how many neighbour pixels are considered, if a pixel could become a dynamic one. If it has unknown neighbour pixels it's on the edge of a known area and hence will not become dynamic. Default value 10.

map_weight: This factor shows the proportion between the weights of the dynamic map file and the slam map file at the merging process. The factor has to be between 0 and 1. $map_{weight} = \frac{dynamicmap}{slammap}$. A value near 0 causes a high impact of the slam map file on the merged file while a value near 1 causes a high impact of the dynamic map (respectively the imported map) file on the merged file. Default value 0.8

dilate_size: Size of the dilation window. Bigger values lead to more dilation around and between dynamic areas. Default value 7 pixels.

dynamic_weight: This factor shows the proportion between the weights of the *exponential moving average time until change* and the current *time until change*. The factor has to be between 0 and 1. A value near 0 causes a high impact of the *time until change* on the next *exponential moving average time until change* while a value near 1 causes a low impact on the change of *exponential moving average time until change*. Default value 0.8

dynamic_time_max: This factor denotes the reference value for *exponential moving average time until change* in milliseconds and is used for dynamic area visualisation at the map export and dynamic mapping rviz plugin. Default value 600000 milliseconds (10 minutes).

dynamic_remove_time: This is the maximum *exponential moving average time until change* an area is recognized as dynamic in milliseconds. Above this value the *exponential moving average time until change* will be reset to 0. Default 86400000 seconds (1 day)

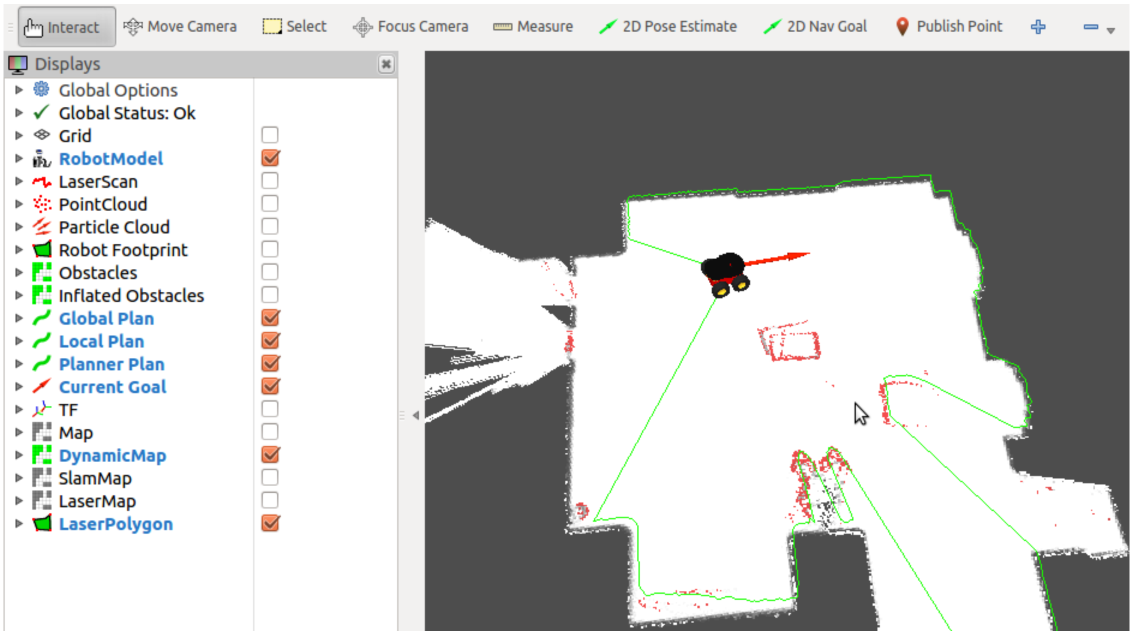
time_period: How often the *dynamic_time* of each pixel, which is within the currently laser scanned area, is increased in milliseconds. Default 5000 milliseconds.

hysteresis: Increasing this value can avoid the impact of measurement errors on the change of the *dynamic trend* nearby obstacles. Default 0.

laser_map: If the laser_map topic is published by gmapping and this value is not 0, the laser_map topic will be used for dynamic area calculation. Default 1.

5.6 Rviz

The Robot Operating System (ROS) package rviz [20] is a very useful tool for robot visualisation. The current map, the virtual robot model, its position in the map and other relevant information can be visualised graphically (see Figure 5.12).



Displays	
Global Options	
Global Status: Ok	
Grid	☐
RobotModel	☑
LaserScan	☐
PointCloud	☐
Particle Cloud	☐
Robot Footprint	☐
Obstacles	☐
Inflated Obstacles	☐
Global Plan	☑
Local Plan	☑
Planner Plan	☑
Current Goal	☑
TF	☐
Map	☐
DynamicMap	☑
SlamMap	☐
LaserMap	☐
LaserPolygon	☑

Figure 5.12: Rviz robot visualisation tool

Rviz subscribes to different topics, which are manually selectable by the user and visualises them. As each topic has an underlying message type, rviz has different plugins to visualise many different message types such as:

- Map
- Laser Scan
- Robot Position
- Image
- Pointcloud (e.g. for particles of Simultaneous Localization and Mapping (SLAM) algorithm)
- Transformation
- Polygon
- ...

If a topic should be visualised, a plugin which supports the underlying message type of the topic is chosen by the user. In addition different configuration parameters can be set. At [20] there are tutorials for rviz available.

A very useful feature is to display all transformations, which are currently published all over the ROS framework. By doing so one can see the different coordinate systems of topics as well as their distance and orientation relative to each other. If the transformations were configured wrong (e.g. they are not all connected or do not form a logical tree) an error will be displayed. This is also a good indicator, if the transformation tree was set up properly.

If we create an own message type (and an own topic), like we did for our dynamic map as will be shown later, there is of course no plugin available for this specific message type. So we have to implement our own plugin for rviz. For implementing such a plugin, tutorials are available for this issue at the ROS community.

5.6.1 DynamicMap plugin

I implemented a plugin for the *DynamicMap* message type to show not only the map, but also the dynamic areas, which were discovered so far. In detail this plugin subscribes to the *dyn_map* topic. The map is a grey valued map, where white indicates no obstacle and black indicates there is an obstacle. A grey value in between indicates that there is a certain probability that there might be an obstacle. The dynamic areas are shown in red color. They are also visualised regarding their *exponential moving average time until change* from lightred (low exponential moving average time until change of an obstacle at the dynamic area) to deepred (high exponential moving average time until change time of an obstacle at the dynamic area).

As reference value for the red colors, the variable *dynamic_time_max* is used. All *dynamic* values above this threshold are visualised as deep red.

For the grey valued map the *data* array of the *dyn_map* topic is used. For dynamic areas the different red values are caused by the *dynamic* array value, which is described in detail in Section 5.5. Of course the grey valued data probability in dynamic areas is lost because of the overlapping red color, but this is only for visualisation. Furthermore the *map* topic can also be visualised in rviz without dynamic areas instead. In fact both maps (*dyn_map* and *map*) can be displayed and are visualised as two half transparent layers. The map visualised by the dynamic map plugin is shown in Figure 5.12.

The *slam_map* topic, which is published by gmapping directly from laser scan data, can be visualised by the standard map plugin.

5.6.2 URDF model

The Unified Robot Description Format (URDF) [14] is an XML format, which represents a robot model. It contains various XML specifications for different parts of a robot as the housing, the wheels, the sensors and so on. There is also a URDF model for the robot P3AT available. It contains the main parts of the robot as virtual described model. Each part is described in its geometric form such that it can be visualised in tools like rviz. The *p2os_urdf* ROS module which contains this URDF file, publishes transformations between all parts and the *base_link* of the robot. The base_link is also defined in the URDF file and represents the position of the robot. It is used as standard transform of the robot and interacts with other transformations.

In Figure 5.13 the visualisation of the robot model in rviz can be seen. The laser scanner, the camera and the bumpers are not contained in this model.

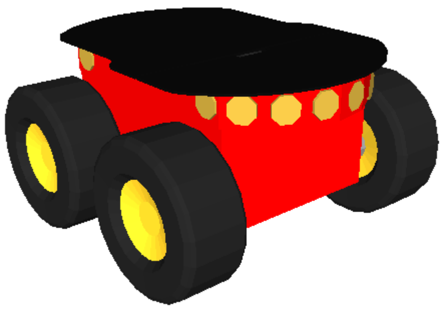


Figure 5.13: Rviz robot model visualisation

The URDF model is not only required for visualisation purposes, it is also important for navigation. When the robot navigates autonomously it has to know how big it is. Otherwise it could be that the robot collides with an object, if it does not take care of the distance between his *base_link* (in most cases its middle point) and his outer parts.

We used the *p2os_launch* ROS package, which contains the URDF files for the robot. It contains much more than just the URDF files, but only these were used.

5.6.3 Set up rviz for P3AT and Navigation Stack

For the ROS modules, display types respectively topics according to Table 5.2 were added. In this table all relevant topics are shown, such that we get a good overview of what is currently going on at the robot and its environment.

Rviz is not only able to subscribe to different topics to visualise them graphically, but it can also publish specific topics. A point and orientation can be selected in the map and an initialpose for the robot is set by doing so. This is done by clicking the *2D pose estimation* or *Set nav goal* button, then clicking at a position in the map and dragging a visualised arrow in the right direction to finally set the orientation.

The localisation algorithm of the navigation stack uses the information of the *2D pose estimation* to arrange the particles of the SLAM algorithm around this position and to use this as the most probable location of the robot. The user can not see, what happens in the background. An *initialpose* topic is published, which contains the position information. The navigation stack and other modules subscribe to this topic and use this information to update their internal robot position.

The *Set nav goal* button publishes a *move_base_simple/goal* topic to set a position as goal after clicking at a certain position in the map. Different modules again listen to this topic - in our case the navigation stack. It starts to find a path to the goal position right after setting a navigation goal position. The path and inflated obstacles are visualised. Inflated objects show an area, where the robot must not drive through with the middle line of its body (its *base_link*). To enter such a zone with one side or some wheels is allowed. This helps to avoid a crash into near objects, because of ignoring its full shape.

Display type (Plugin)	**Topic**
Fixed Frame	dyn_map
Robot Model	base_link
Laser Scan	scan
Point Cloud	RosAria/sonar
Particle Cloud	particlecloud
Robot Footprint	move_base/local_costmap/robot_footprint
Obstacles	move_base/local_costmap/obstacles
Inflated Obstacles[3]	move_base/local_costmap/inflated_obstacles
Global Plan	move_base/TrajectoryPlannerROS/global_plan
Local Plan	move_base/TrajectoryPlannerROS/local_plan
Planner Path	move_base/NavfnROS/plan
Current Goal	move_base/current_goal
TF	tf
Map	slam_map
DynamicMap	dyn_map
Map	map
Laser Polygon	update_area

Table 5.2: Rviz topic configuration for dynamic mapping and navigation stack

5.7 General driving package (p3at)

The robot needs to be controlled somehow regarding its driving actions, this module provides some general driving commands to perform this. It uses the sensors which are provided by ROSARIA and is performing velocity commands.

5.7.1 Keyboard control module

A module for controlling the robot with the keyboard was developed. It simply listens on key-down events at specific keys and publishes on the *cmd_vel* topic. With the cursor keys it is possible to control the movement direction. With the key *A* the speed is increased, with the key *S* the speed is decreased.

[3]Visualises the obstacles with clouds around them, where the robot must not cross with its base_link, but can enter with just a part like a wheel for instance.

This module is as simple as useful to control the robot. It was designed as alternative to the big and very calculation intensive navigation stack. One of those can be chosen according to the actual use case.

The module is called by:

```
rosrun p3at keyboard_control
```

5.7.2 Safety module

The idea of this module is to inhibit control commands, if they can not be safely performed. For instance it makes no sense, if the robot receives a drive forward command, while it already touches a wall at the front, which is indicated by the bumpers. So this module checks the bumpers and only forwards the control commands from the *cmd_vel* topic to the *RosAria/cmd_vel* topic, if no bumper is pressed (see Figure 5.14). It even does not only interrupt a forward driving command, but actually sends a command to stop the robot immediately in the case a bumper was activated.

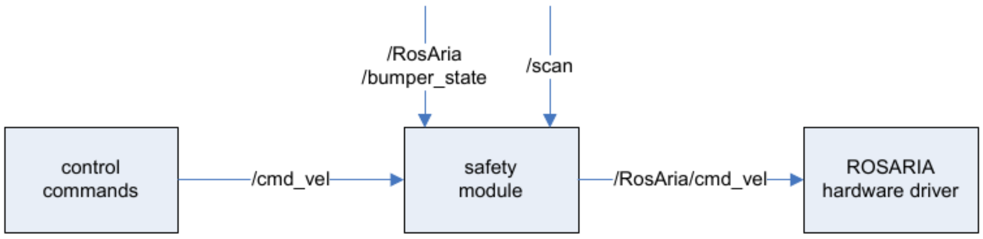


Figure 5.14: Safety approach

I also experimented with the laser scanner as additional safety module. An algorithm was implemented, which checks if an object is within 30cm in front of the robot and forbids to drive forward in that case. This works good, if the keyboard control module is used for controlling the robot, but there are some troubles when the navigation stack is used as control unit. Here the robot does autonomous driving and want to drive closer than 30cm to objects. We can't make the 30cm too small, because the robot will not be able to halt in front of an obstacle when it drives too fast. What was done is to automatically reduce the speed by multiplying the input speed value of the safety module with a slowdown factor (0.3 in this implementation) when the robot is close to obstacles. This speed reduction is active within 60 to 70cm distance of obstacles to the robot.

The laser scanner safety part of this module can be turned on by the command line parameter *_use_laser:=1*. The module is called by:

```
rosrun p3at safety_navigation [_use_laser:=1]
```

5.8 Navigation stack

As already mentioned in Section 4.4.1, a stack contains various packages and modules. They all work together to perform a global task. The navigation stack was made for autonomous navigation and driving.

In Figure 5.15 an overview of the navigation stack is shown. On the left side of the figure the transformations needed by the stack are shown. The stack subscribes to the *nav_msgs/Odometry* topic, which contains information about the current position and velocity of the robot.

On the top we can see that a goal position is needed for the robot to drive there. This can be done by rviz. The *Set nav goal* button publishes a *move_base_simple/goal* topic, which is used as goal position. On the right side of the figure can be seen, that a map is needed, as well as a laser scan and a point cloud. The map is published by the dynamic mapping module in this thesis as will be shown in Section 5.5. But the map can also be a static map, which needs to be imported prior to the start of the navigation stack. To run the navigation stack modules need to be started to provide these topics in order to use the navigation stack.

If all of these topics are provided, the navigation stack publishes a *cmd_vel* topic, which contains velocity and direction, and will bring the robot to the goal position.

Internally the navigation stack uses different algorithms and maps to find an optimal path from the current position to the goal.

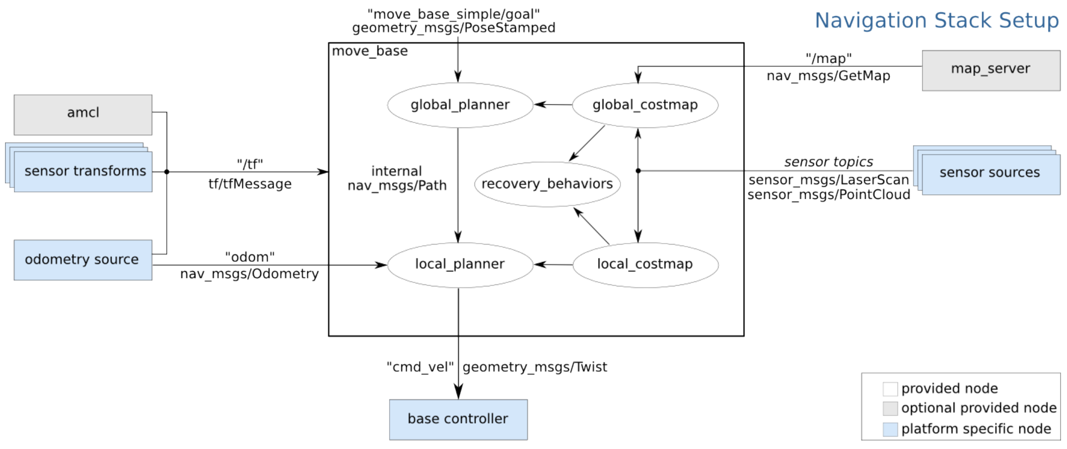


Figure 5.15: Navigation stack overview [6]

Although we will not look deeper into the internals of the navigation stack we have to discuss a few things in order to understand the changes we made in the navigation stack configuration and why they were necessary.

5.8.1 Global and local costmap

As can be seen in Figure 5.15, there is a global costmap and a local costmap. When the navigation stack is searching for optimal paths it generates these maps out of sensor data, and of course it uses these maps for path planning. These maps contain information about obstacles.

The global map is a costmap all over the known area of the map and used for long-term plans, the local costmap contains only the direct environment of the robot and is used for local planning as well as obstalce avoidance. The global planner, as a part of the navigation stack, calculates a global path by using the global costmap data and publishes this data. The local planner subscribes to this topic and uses this data, as well as the local costmap to find an optimal local path. There can be some objects around the robot, which were not visible in the global map, but appeared when the robot comes closer - the local planner has to handle this. Finally a velocity and movement direction is published on the *cmd_vel* topic.

If the robot has lost its current position in the map there are recovery behaviours available with the purpose to bring the robot back in place. In most cases the robot starts rotating until it finds its position again. By doing so it checks the laser scan against the map and searches for the most likely position and orientation.

More about the internal behaviour of the navigation stack is described in [5].

5.8.2 Changes for P3AT

To use the navigation stack with the robot P3AT and the dynamic mapping module, I had to make the following changes. I changed the paths of the *move_base.launch* file and the controller frequency. The controller frequency is an important configuration parameter, because if it is to high, problems with the performance of the whole Robot Operating System (ROS) framework are caused. If it is too low, the robot will not drive very smoothly, but roughly and the internal control loop will be too slow for accurate navigation.

If there is a static map used, which is not changed during runtime, the import of the static map file can be triggered directly at the *move_base.launch* file. We have a *map* topic available and also want to use changes of the map during runtime, so we do not need to import the map here. Hence we disabled this source code section.

The files *local_costmap_params.yaml* and *global_costmap_params.yaml* are located in the *p2os-master* module, but are used in the *move_base.launch* file. Here the *update_frequency* and *publish_frequency* parameters have been changed. As well as the controller frequency before, they also need to be set according to the amount of other modules running simultaneously and to ensure accuracy. In general it's a trade-off between speed and the desired accuracy. It is not recommended to set both values to arbitrary values, because they strongly depend from each other.

5.8.3 Transformation broadcaster

Publishing a transformation between the *map* topic and the *slam_map*, once the slam map and the imported map were merged, is also required. Otherwise other modules will fail to calculate transformations, as the navigation stack or the calculation of the *update_area* for instance. The transform broadcaster also subscribes to the *initialpose* topic and publishes the transform from *map* to *slam_map* after the initial pose was set. Before the initial pose was set we assume that the origins of the two maps are equal and publish the transformation data according to this.

What is also required is to publish the transform between the *dyn_map* topic and the *map* topic although it has the same origin and orientation. But this is important to get a connected transformation tree.

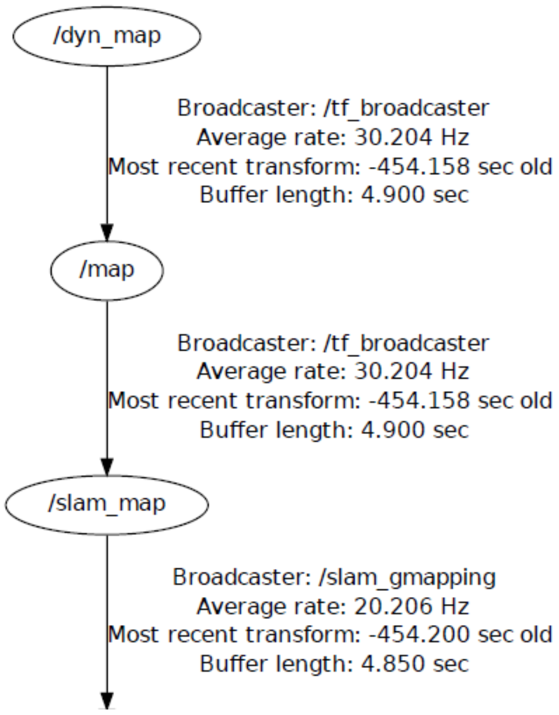

/dyn_map
Broadcaster: /tf_broadcaster
Average rate: 30.204 Hz
Most recent transform: -454.158 sec old
Buffer length: 4.900 sec
/map
Broadcaster: /tf_broadcaster
Average rate: 30.204 Hz
Most recent transform: -454.158 sec old
Buffer length: 4.900 sec
/slam_map
Broadcaster: /slam_gmapping
Average rate: 20.206 Hz
Most recent transform: -454.200 sec old
Buffer length: 4.850 sec

Figure 5.16: Transform published by the transformation broadcaster module

In Figure 5.16 the part of the transform tree, which is published by this module is shown.

A transform between the *base_link* (robot position) and the *update_area* is also published by the transform broadcaster module. This is required for transforming points of the map to the *update_area* topic, to check if the points are in or outside the update area polygon. This information is used to reset dynamic areas, see Section 5.5.5.

5.9 Global transformation tree

The global transformation tree is necessary to establish a hierarchy and dependency between Robot Operating System (ROS) topics. Transformations and the transformation tree are described in Section 4.4.4.

In Figure 5.17 the overall transformation tree of the project can be seen. The root node of the transformation tree is *dyn_map*. The *map* and *slam_map* topics are below, as shown in the previous section. Only these map topics are used within this project, the transformations between them are published by the transform broadcaster module of the dynamic mapping package.

The transform between *slam_map* and *odom*, which represents the moving of the robot, is published by the gmapping module. This module is responsible for the localisation, hence it knows the transform between them.

The transform between *odom* and *base_link* is published by the RosAria module. It performs the driving control of the robot and contains information about the robots base link.

From the *base_link* a lot of topics are derived from, but all of them except one represent physical parts of the robot. These are the top plate, the front right wheel or the front sonar for example. These transformations are published by the p2os-master package, which also contains the Unified Robot Description Format (URDF) file (see Section 5.6.2). These transformations are used by rviz to display the robot model and by the navigation stack to know how broad, long and heigh the robot is to avoid any crashes.

The only topic which is not the virtual model of a physical part of the robot is the *update_area*. It is used to calculate the area, which is relevant for dynamic areas update.

Building a complete and valid transformation tree is not trivial. The most difficult thing is not the connection of the different topics, but the time aspect. We have to make sure, that the different transforms are calculated often enough. Otherwise the internal control loops of different algorithms and modules will not work properly. But if the transform is calculated too often we will get in trouble with the overall performance. In that case most of the processing power is used to update the transformations and less time is left for executing algorithms. In our case a frequency of about 30Hz seems to be a good choice between these boarders.

The calculation effort is not too big and we can be sure to have up to date transforms. Even from the root node to a leave node the overall transform is recent enough, although the times of the individual transforms sums up.

The transform buffer is another important part, because the buffer only contains transforms of the last 15 seconds. If the overall transforms are not configured properly this buffer will overflow with a high probability. It's crucial to be sure the transforms are set properly, otherwise troubles will be inevitable. More information about transforms is shown in [43].

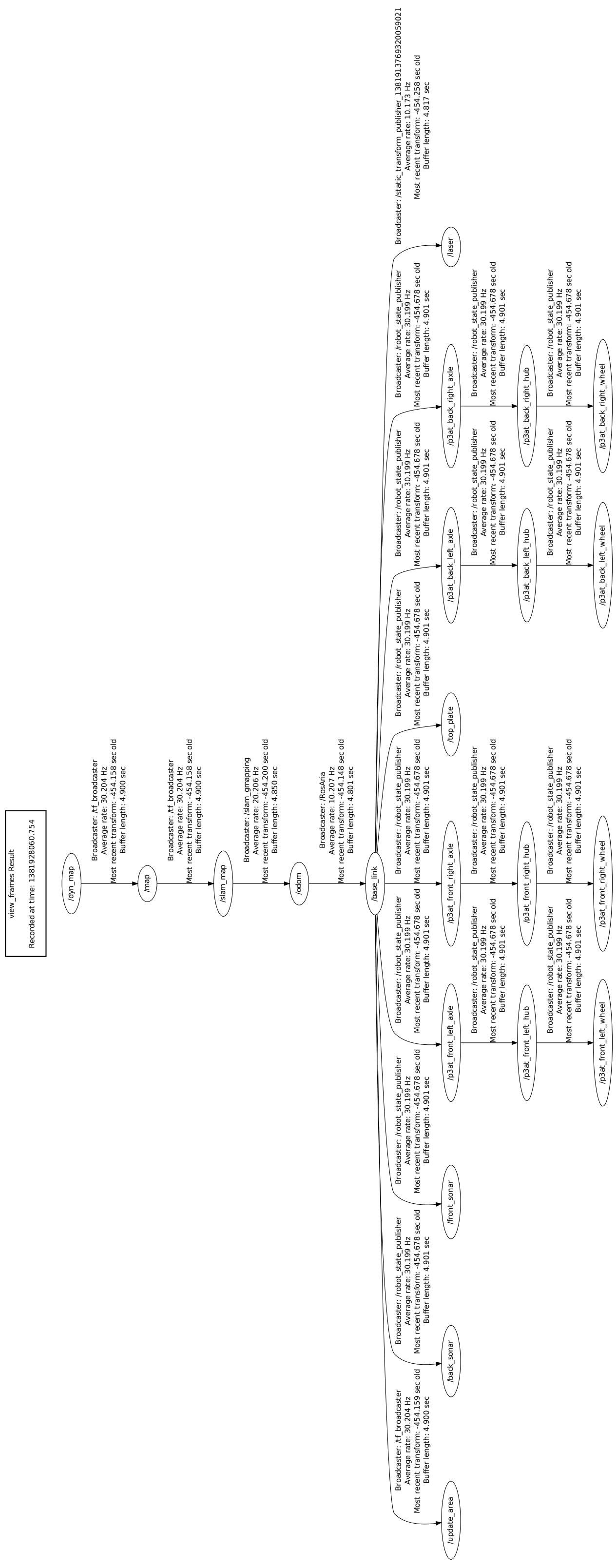


Figure 5.17: Transform tree of the whole ROS project

CHAPTER 6

Experiments and results

6.1 Robot P3AT

The robot used for experiments is a Pioneer 3AT from MobileRobots, which is an often used scientific robot.

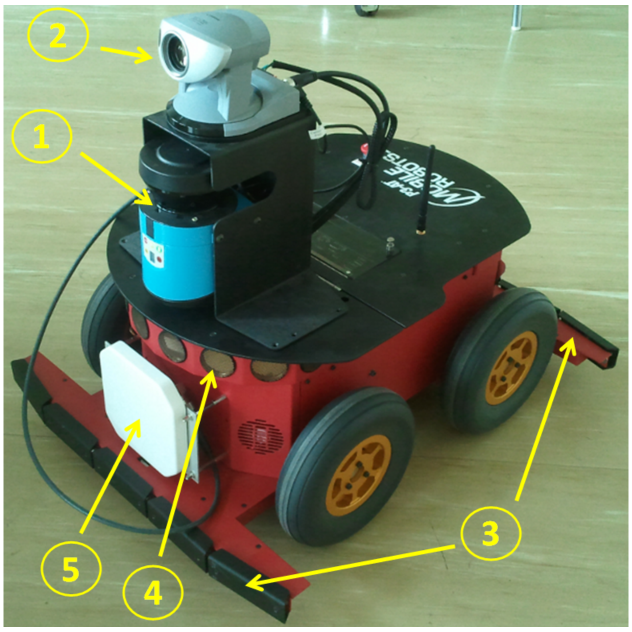


Figure 6.1: Robot P3AT was used for experiments

As we can see in Figure 6.1, the robot is equipped with the following sensors:

1. Laser Scanner LMS100

2. Camera Canon VC C50i
3. Front and rear bumpers
4. Front Sonar
5. RFID Scanner (white plate)

The following hardware and software is assembled respectively installed at the robot:

- Intel Core 2 Duo CPU 2.26GHz
- 2GB RAM
- WLAN Access Point
- Connectors for VGA and USB (for monitor screen, mouse and keyboard)
- Ubuntu 12.04 (Linux Kernel 3.5.0-32)
- Robot Operating System (ROS) Groovy Galapagos

The robot has a maximum speed of $1m/s$ and can rotate its wheels independently on each side of the robot. This makes it possible to drive a curve by reducing the rotation speed of the wheel at the inner side of the curve and/or increasing the rotation speed at the wheel at the outer side of the curve. Additionally it is possible to rotate without changing the position. This allows the robot to turn in small corridors.

For more information about the robot see [4].

6.1.1 Laser Scanner

The probably most important sensor for mapping is the laser scanner. Hence detailed information about it is presented. The laser scanner mounted on the robot is a SICK Laser Measurement System (LMS) 100. It is suitable for industrial needs in the field of object measurement, determining position or area monitoring. The technology applied is an optical measurement system, which consists of an infra-red laser and an infra-red sensor. The time of light of laser pulses is proportional to the distance of the reflecting object. The measurement is precise, in detail the statistical error is about 12mm. The maximum range of the sensor are 20m and the scanning is done over 270 degrees (see Figure 6.2). We configured the laser scanner with a resolution of $0.5°$, which leads to 540 laser measurement points.

See [24] for more details about the LMS100 laser scanner.

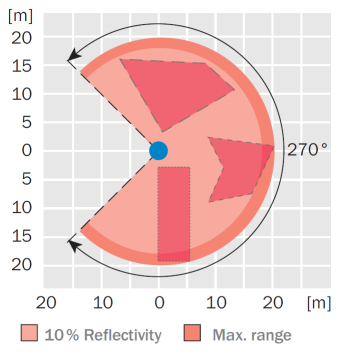


Figure 6.2: LMS100 scanning area [24]

Type	**Value**
Max. range	20 m / 18 m (at 10 % reflectivity [1])
Scanning angle	Max. 270°
Angular resolution	0.5°/0.25°adjustable
Scanning frequency	50 Hz / 25 Hz
Response time	20 ms / 40 ms
Statistical error (1 s)	Typical 12 mm

Table 6.1: Technical features of the SICK LMS100 laser scanner [24]

6.2 Compiling and running dynamic mapping

6.2.1 Catkin vs Rosmake

In the Robot Operating System (ROS) version I used a build system named *catkin* is used. With catkin, all modules are compiled by calling *catkin_make*. In each module a *CMakeLists.txt* and a *package.xml* file must exist. How *catkin* works in detail is described in [41].

In earlier versions of ROS, the build system *rosmake* was used. I also use some legacy modules in this project, which still use *rosmake*. Each module has to be compiled on its own with the command *rosmake*. How *rosmake* works in detail is described in [42].

Table 6.2 shows, which packages respectively modules use *catkin* and which *rosmake*.

6.2.2 Running dynamic mapping

I strongly recommend to run the different ROS modules on two different hosts like discussed in Section 5.2.2. The robot on its own will get performance problems otherwise and a second computer is needed anyway, if rviz is used. Because a screen is needed to visualize the rviz

[1]At least 10% of the out sent light need to be reflected to recognize an obstacle.

catkin	**rosmake**
dynamic_mapping	gmapping
p3at	LMS1xx
p2os-master	navigation stack
rviz	
RosAria	

Table 6.2: List of modules using catkin and rosmake

interface. As a requirement the two computers need to be connected over a WLAN connection. The following steps are needed to run the dynamic mapping including all necessary modules (the two hosts are named *robot* and *pc*):

1. Run *dynamic_mapping_robot.sh* on robot
2. Run *dynamic_mapping_pc.sh [-n] [-f filename]* on pc

Important is that we need to wait until *dynamic_mapping_robot.sh* has loaded all modules, thus the last module p2os_urdf was started before we run *dynamic_mapping_pc.sh* on pc.

The script *dynamic_mapping_robot.sh* loads the following modules in the given order:

1. roscore (ROS core module)
2. RosAria (ROS robot drivers)
3. LMS1xx (Laser Scanner driver)
4. gmapping (Simultaneous Localization and Mapping (SLAM) module)
5. tf (static transform base_link to laser)
6. transform_broadcaster (transform between maps and update_area)
7. safety_navigation (safety module)
8. p2os_urdf (robot model)

Most of the modules have configuration parameters, which can be used to change their behaviour. If this is required they can be added respectively changed directly in the script. This may be particularly interesting in case of the gmapping module, because there are various parameters which can be changed.

The script *dynamic_mapping_pc.sh* loads the following modules in the given order:

1. rviz (roboter visualisation)
2. dynamic_mapping (import, export, processing dynamic maps)
3. keyboard_control (control the robot by keyboard) [2]

[2]Keyboard control is started, if no argument was set

4. navigation stack (autonomous control) [3]

The user has to choose if the keyboard control module or the navigation stack should be used. Only one of them can be used simultaneously. The script *dynamic_mapping_robot.sh* is called without any parameters to start the keyboard control module or with parameter *-n* to start the navigation stack.

If a map was once exported and should be imported later again for further mapping, this import can be done by using the option *-f* and setting the desired file name of the map file. After this the robot location has to be set either by rviz *Set initial pose* feature or from an external module by publishing the robots location within the map on the *initialpose* topic. After the initial position of the robot was set, the map merging starts and dynamic areas are displayed respectively changed and generated.

If the dynamic mapping was finished the map can be exported by calling:

```
rosrun dynamic_mapping map_export [-f filename] [parameters4]
```

The file name can be chosen by using the option *-f* and the desired file name. The parameter *_grey_val:=1* exports the map without red colored dynamic areas as grey valued map.

6.3 Experiments

Dynamic mapping was tested on the basis of representative experiments. For these experiments the keyboard control module was used instead of the navigation stack, because the full functionality of the navigation stack is not required to test the dynamic mapping module. We want to test the dynamic mapping module and its intercommunication with other modules.

Apart from that, all modules which have been discussed so far, were used.

The experiments were performed in a room without any small objects (from the perspective of the laser scanner) like table bases, because with these localisation and mapping is difficult. They can be within the 0.5°angular resolution of the laser scanner. Depending on the distance and orientation of these objects to the robot, the laser scanner might recognize them or not. So they might be recognized by some scans and invisible for others. This can lead to the incorrect interpretation of dynamic object. The room chosen for the experiments is shown in Figure 6.3. On one side there are glass doors, on the other side there is an elevator and stairs beneath.

6.3.1 Experiment 1: Obstacle changes into dynamic area

In this first experiment we start dynamic mapping in a room, where an obstacle (a box) is located in the middle of the room. For this experiment the *slam_map* was used as information source for the change rate calculation, which means that the changing of the probability value over time was considerer for the change rate.

The robot starts next to the box. The first figure of the dynamic map is shown in Figure 6.4a. We can see a black line in the middle of the room, this is one edge of the box. Left to the edge

[3]Navigation stack is started by argument *-n*

[4]All available parameters are listed in Section 5.5.3

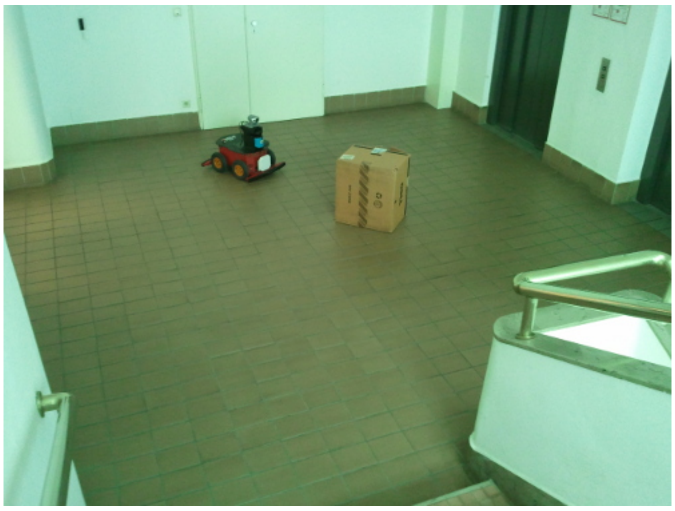


Figure 6.3: Room which was used for our experiments

there is an unknown area. Here the robot has its initial position. The robot is oriented facing the box, hence the unknown area is behind him.

We now drive the robot by keyboard control around the box once, which we see in Figures 6.4b and 6.4c. The full shape of the box can be seen now, where the edges are black - it is recognized as obstacle. The full box area is unknown space, because the robot is of course not able to look into the box.

In Figure 6.4d dynamic areas were generated for the first time. The dynamic area above the box is very small, it was caused by a small part protruding from the box. The dynamic area on the right of the figure is caused by me, sitting there and not keeping my feet calm.

Between Figure 6.4d and Figure 6.4e I removed the box from the middle of the room. This simulates an obstacle, which has gone after the robot rescans the area it was located in. Because of this, the unknown area caused by the box has changed to a known area in Figure 6.4e. We see small dynamic areas nearby the box, these were caused by my legs when I walked over to the box to grab and remove it. We also see some first dynamic areas raising at the edges of the box.

In Figure 6.4f the algorithm for changing an obstacle into a dynamic area has finished. The dilation filter already added some dynamic pixels and so the dynamic area is free of holes and shows the edges of the box now. This is exactly, what was expected here. The figures were produced by dynamic mapping export as discussed in Section 5.5.

6.3.2 Experiment 2: Free space changes into dynamic area

In this experiment we want to do it the other way round. We start with an empty room in Figure 6.5a. For this experiment again the *slam_map* was used as information source for the change rate calculation. There were no dynamic areas discovered so far, the robot drove around the box through the room to discover it. The current position of the robot within the map is on the very left of the known area.

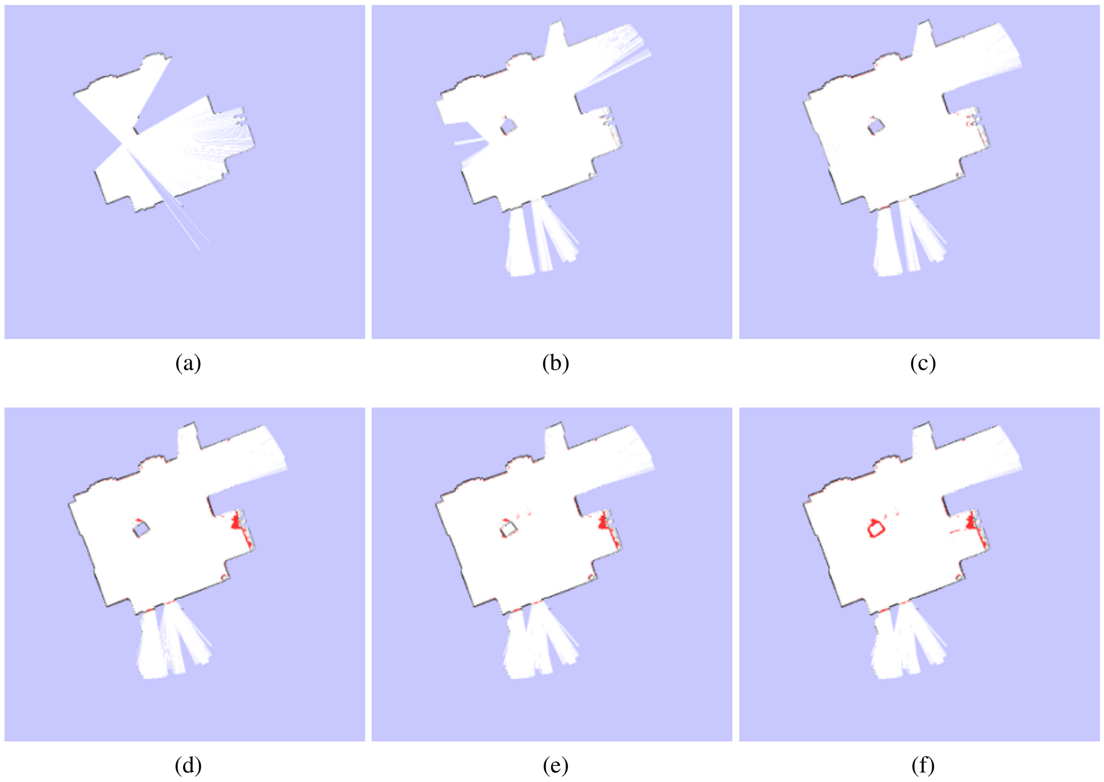


Figure 6.4: Experiment 1: Progress of dynamic area creation from obstacle

The white arms of known area, which reach far into the unknown area at the bottom left of the figures, are caused by a glass door. The other ones on the right were caused by an elevator keeping its door open for a few seconds.

After that, the same box as in the experiment before was positioned in the middle of the room.

In Figure 6.5b we see the rise of first dynamic areas. The one in the middle of the room is caused by one edge of the box. Now I started to drive around the box with the robot and discover the full shape of the box in that way.

In Figure 6.5c the process of creating the dynamic areas can be seen. By the way, the dynamic areas on the bottom of the known area are again caused by me, doing this experiment and moving my legs.

Finally the robot discovered the whole box as can be seen in Figure 6.5d. We nearly got the same shape of the box as in the previous experiment. At the edges of the box again a dynamic area was generated as expected.

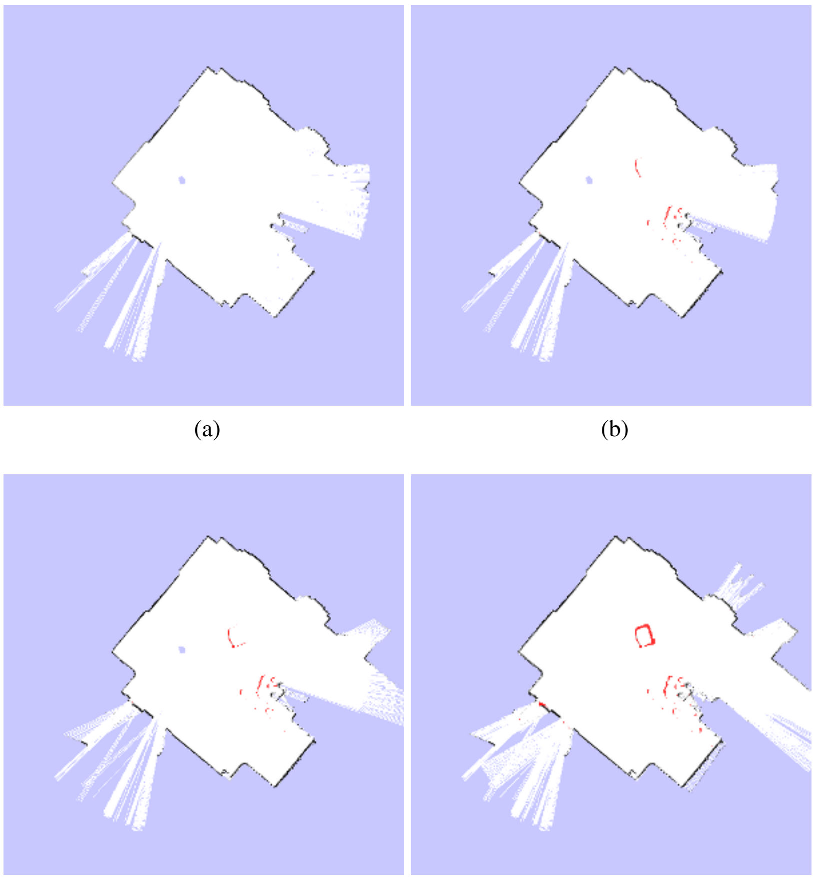

(a) (b) (c) (d)

Figure 6.5: Experiment 2: Progress of dynamic area creation from empty space

6.3.3 Experiment 3: Path generation from walking humans

During this experiment the position and orientation of the robot were not changed. The aim of this experiment was to see the creation of dynamic areas caused from walking human over time. For this experiment the *slam_map* was used again as information source for the change rate calculation.

Now the maps have not been exported like in the experiments before, but I made screen shots from the rviz visualization of the robot and its environment as can be seen in Figure 6.6. This is because in that way the update area can be seen - the current laser scan visualised as green polygon in the figure. The unknown area is not lightblue as in the exported map figures any more, but darkgrey instead.

I started with an empty room and without any movement in the room, see Figure 6.6a. After that I started walking on the same path over and over to simulate a path where humans walk often. The change rates of the pixels where I walked are calculated. In Figure 6.6b we already

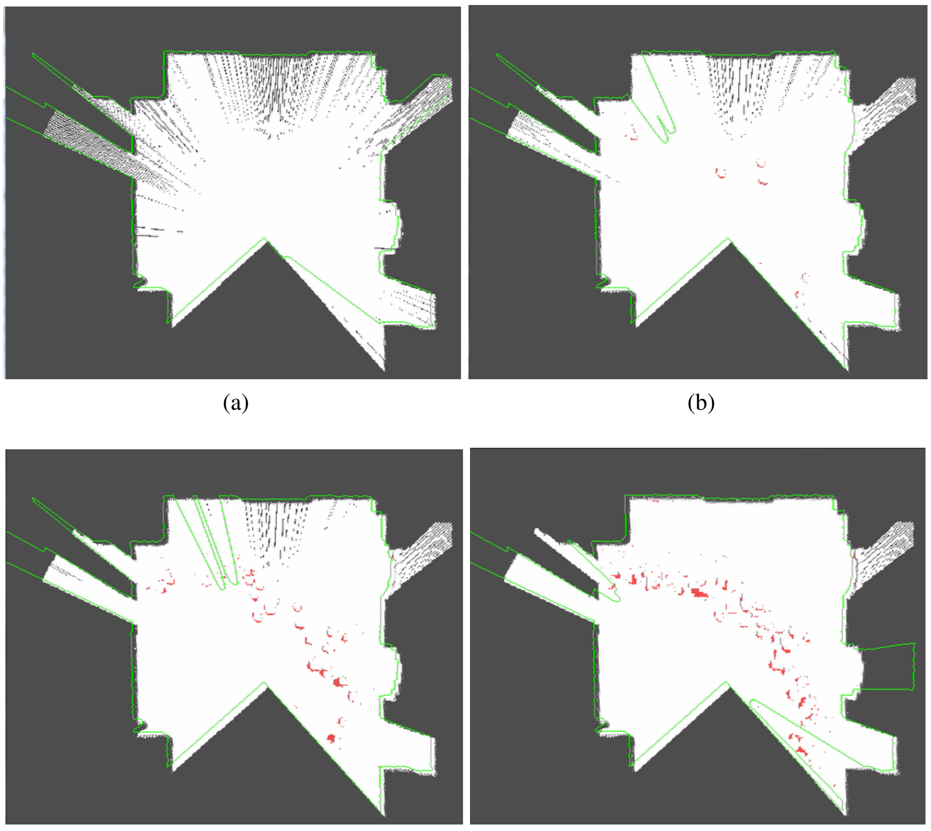

(a)
(b)
(c)
(d)

Figure 6.6: Experiment 3: Progress of dynamic area path creation

see small dynamic areas caused by humans legs. We also see that the green laser scan reaches into the known area on the top left of the room. The reason for this is that here I am actually walking. The two green shapes are caused by human legs.

After some while and a few walks on the same path in the room later, we get a map image like in Figure 6.6c. The amount of little dynamic areas, all nearly having the same shape, have increased over time. We also see the impact of a walking human on the green laser scan polygon again. As already described in Section 5.5.2, the red color intensity depends on the change rate and a reference value. There are no different red colors visible although the change rate between different dynamic areas are different. This is because the reference value for the intensity of the red color was set to 60 minutes. Hence the change rates of walking, which took only a view minutes, are too similar with respect to the reference value to cause different colors.

Finally after more time and still human walking on the path, many dynamic areas were generated as shown in Figure 6.6d. The path is distinctive, it leads from the bottom right to the top left of the room and vice versa. Some bigger dynamic areas can also be recognized, which were generated by dilation filtering. If we would have used a bigger value for the *dilate_size* parameter (than the default value), we would see much more connected dynamic areas. Then the path may be connected to one big dynamic area. But anyway, the generated path is well recognizable even with the default value for the *dilate_size* parameter. The aim to generate a map containing main walking paths of human was successful.

6.3.4 Experiment 4: Change rate of doors

In this experiment the change rates of two doors are considered. One door is opened and closed frequently (twice a minute), the other one is opened and closed infrequently (once every 2 minutes). For this experiment the *laser_map* topic was used as information source for the change rate calculation.

The exponential moving average time until change is published within the *dyn_map* topic for each pixel. To interpret it, a reference value is required, which is *dynamic_time_max*. This reference value is also contained in the *dyn_map* topic and was set to 10 minutes for this experiment.

The rviz plugin calculates the intensity of the red color of each pixel depending on the change rate with respect to the reference value *dynamic_time_max*.

A light red corresponds with a high change rate (low *exponential moving average time until change*) and a dark red with a low change rate (high *exponential moving average time until change*).

The experiment starts with the robot standing in the middle of the room, which can be seen in Figure 6.7a. The two doors are opened and closed according to their defined change rate (the first door twice a minute, the second door once every 2 minutes). In Figure 6.7b the fist door was opened, which caused the additional known area on the top left in the figure. The dynamic area does not have the size of the full shape of the door, because dynamic areas are avoided within a specified range from unknown area. This helps to avoid dynamic areas caused by measurement errors of the laser scanner. The door is opened by pushing, which means that it is located in the neighbour room when it is opened. This is the reason why there is no dynamic area at the position of the opened door.

After a minute a dynamic area with a light red color is visible in Figure 6.7c.

After two minutes, the second door is opened for the first time, which causes an additional known area at the top right of Figure 6.7d.

After five minutes, where the doors were opened and closed according to their defined change rate, the two dynamic areas have different color values. The first door still has a light red color, which stands for a high change rate. The second door has a dark red color, which stands for a low change rate.

This experiment showed, that the calculation of the change rate and its interpretation as red color value works.

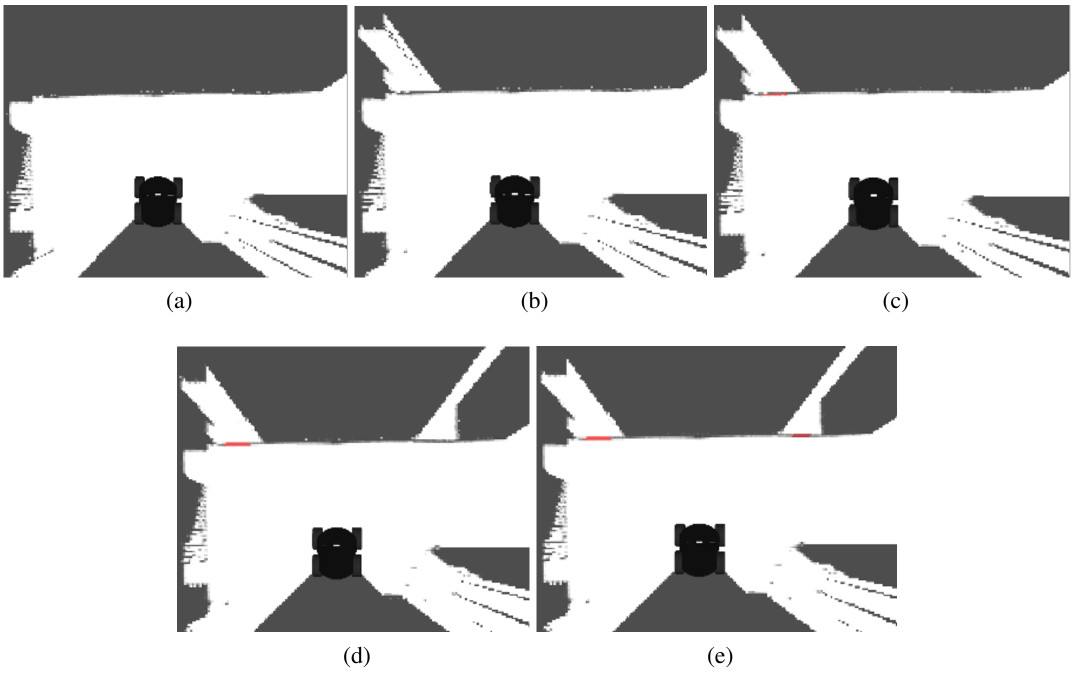

(a) (b) (c) (d) (e)

Figure 6.7: Experiment 4: Change rate of doors

CHAPTER 7

Future work

7.1 Dependency between dynamic areas

Until know there was no dependency between different dynamic areas considered. This would also be an interesting task and could bring additional information as well as increase the accuracy of the probability values and the change rates of dynamic areas.

Let's assume a door is opened and closed over and over. After some time there are two dynamic areas created. One is positioned at the door's closed position and the other one is generated at the door's open position. So there are two dynamic areas and in fact, they depend on each other. If the probability that the door is opened is 0.3, then this is the dynamic probability value of the dynamic area at the open door position. But from that the probability of the closed door dynamic area value can be calculated, which has to be $1 - 0.3$ which is 0.7. So the door's probability is 0.7 to be closed. The change rates of both dynamic areas need to be the same in this case. Of course this is only true if the door can only have two positions (opened and closed), but nothing in between. Otherwise the sum of all probabilities hast to be 1.

This way of argumentation is only valid if the object existence between these areas is complementary. This means that an object has to be located in one of these areas all the time, but it is not allowed to be in both areas simultaneously.

By exploring the dependency between these two dynamic areas additional information about the moving of objects can be gained. This can help for classifying objects. By knowing the dependency between the two dynamic areas of a door, it is easier possible to classify this object as door.

Another point is, that if such a dependency is known, the dynamic probability values of all dynamic areas, which are depending from each other, sum up to 1. This can be concluded directly from the complementary dynamic probability values. The change rate of complementary dynamic areas should be the same, because they have extrema on the same points in time in their probability functions. A minimum of the probability function for the one dynamic area is a maximum of the probability function for the other one. From that it is possible to validate the dynamic probability values as well as the change rates and correct them if this is necessary.

If errors occur during the dynamic mapping process this can be recognized by considering this correlation.

7.2 Extending the map by additional information

Another point for further research would be to examine which additional other information could be saved to the map for getting further benefits. Two exemplary cases from the context of road traffic are discussed briefly.

7.2.1 Dynamic area average speed detection

Considering the movement of objects and generating dynamic areas caused from that movement in the map, it would also be interesting how fast objects are in certain dynamic areas. Of course every object may move with a different velocity through the dynamic area, but knowing the average velocity will bring a lot of additional information with it.

Assume there are already dynamic areas with average velocities of objects moving there. Then it would be again easier possible to classify such dynamic areas. In road traffic crosswalks could be detected easier because the average velocity will be very low compared to the whole street. Maybe even each direction of movement could get its own average speed value. In case of crosswalks the movement direction of cars is oriented $\pm 90^\circ$ from the pedestrian walking directions. The cars will be much faster than the pedestrians. In consequence of the average speed values of these two participants in road traffic, crosswalks could be recognized automatically by intelligent cars. Having the underlying data also the percentile values can be calculated and used instead or additional to average values.

With the knowledge of the position of crosswalks an autonomous car can slow down and pay more attention for pedestrians to avoid accidents.

For autonomous cars also the average speed on different streets can be interesting. From that it could know if it is on a highway or in a city. According to this it can adjust its own speed accordingly. Of course this can not be the only information source for deciding the right speed value by the autonomous car, but an additional parameter.

7.2.2 Traffic light frequency detection

Saving dynamic information into the map, was limited to probability and change rate until now. But it can also be extended to other types of information. An interesting one is information about the traffic light frequency. If cameras detect the duration of green phases and yellow, respectively red phases it would be possible to save to every single crossroad the duration of the green phase or the frequency of two succeeding green phases.

If this information is provided for the driver of a car in any way, he can know in advance how long the traffic light will be green. With this information and the knowledge of how long the traffic light has been green until now, which we assume to be visible to the driver while getting closer, the driver can estimate how long it will still be green and adjust its speed. This information may also be useful for autonomous cars in the future.

CHAPTER 8

Conclusion

Within this thesis, the integration of information about the dynamic environment into maps was evaluated and performed. Different possibilities for gaining dynamic information were described. We have chosen to compare maps generated at different timestamps by a Simultaneous Localization and Mapping (SLAM) algorithm, which uses the laser scan data as input. The laser scanner was used as main information source regarding information about the environment. Dynamic areas including their probability of obstacle existence as well as their average change time are saved into maps. The dynamic areas of these maps are extended by a dilation filtering algorithm. The dynamic areas depend on change rates of dynamic behaviour. These change rates were calculated by considering extrema of the obstacle probability over time.

Different Robot Operating System (ROS) packages were developed to perform the overall task of dynamic mapping. The dynamic mapping itself as well as import and export possibilities of maps were implemented. Mapping can be started again based on already existing maps, only the starting position of the robot within the map needs to be defined. Transformations between different coordinate systems as well as the laser scan area were covered in order to understand the entire solution. The dynamic mapping was graphically visualised over self written plugins for rviz.

Experiments with the robot P3AT showed the behaviour of the implementation. They also showed that the dynamic mapping indeed works in case of furniture objects, opened and closed doors as well as in generating paths from human movement.

An outlook about additional ideas and possibilities regarding dynamic mapping was given. In general there are still many interesting issues for further research in the field of dynamic mapping in robotics.

List of Figures

List of Tables

Acronyms

ROS:	Robot Operating System
SLAM:	Simultaneous Localization and Mapping
URDF:	Unified Robot Description Format
PPM:	Portable Pixel Map
PGM:	Portable Grey Map
RGB:	Red Green Blue
CPU:	Central Processing Unit
EM:	Expectation Maximation
DARPA:	Defense Advanced Research Projects Agency
LMS:	Laser Measurement System
YAML:	YAML Ain't Markup Language
GPU:	Graphics Processing Unit
GUI:	Graphical User Interface

Bibliography

[1] Risk management – Vocabulary. *ISO Guide 73:2009, 3.6.1.4*, 2009.

[2] IEEE Recommended Practice for the Maintenance of Industrial and Commercial Power Systems. *IEEE Std 3007.2-2010*, pages 1–56, 2010.

[3] IEEE Std 686-1997. IEEE Standard Radar Definitions, 1997.

[4] Adept MobileRobots. http://mobilerobots.com/researchrobots/p3at.aspx. Accessed: 2013-03-05.

[5] Andreas Paepcke et al. http://wiki.ros.org/navigation. Accessed: 2013-26-06.

[6] Andreas Paepcke et al. http://wiki.ros.org/navigation/tutorials/robotsetup. Accessed: 2013-26-06.

[7] Andreas Paepcke et al. http://wiki.ros.org/rosaria. Accessed: 2013-21-09.

[8] Andreas Paepcke et al. http://wiki.ros.org/ros/tutorials/multiplemachines. Accessed: 2013-21-09.

[9] Andreas Paepcke et al. http://www.ros.org. Accessed: 2013-03-05.

[10] Andreas Paepcke et al. http://www.ros.org/wiki. Accessed: 2013-21-09.

[11] Andreas Paepcke et al. http://www.ros.org/wiki/groovy. Accessed: 2013-23-09.

[12] Andreas Paepcke et al. http://www.ros.org/wiki/ros/concepts. Accessed: 2013-22-06.

[13] Andreas Paepcke et al. http://www.ros.org/wiki/ros/installation. Accessed: 2013-30-06.

[14] Andreas Paepcke et al. http://www.ros.org/wiki/urdf. Accessed: 2013-23-09.

[15] AudioEnglish.org. http://www.audioenglish.org. Accessed: 2013-26-11.

[16] D. Avots, E. Lim, R. Thibaux, and S. Thrun. A probabilistic technique for simultaneous localization and door state estimation with mobile robots in dynamic environments. In *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*, volume 1, pages 521–526 vol.1, 2002.

[17] Brian Gerkey. http://www.ros.org/wiki/gmapping. Accessed: 2013-21-09.

[18] Darel Rex Finley. http://alienryderflex.com/polygon/. Accessed: 2013-14-10.

[19] DARPA. http://www.theroboticschallenge.org. Accessed: 2013-22-12.

[20] Dave Hershberger and David Gossow and Josh Faust. http://www.ros.org/wiki/rviz. Accessed: 2013-22-09.

[21] F. Dellaert, D. Fox, W. Burgard, and S. Thrun. Monte carlo localization for mobile robots. In *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, volume 2, pages 1322–1328 vol.2, 1999.

[22] M. W M G Dissanayake, P. Newman, S. Clark, H.F. Durrant-Whyte, and M. Csorba. A solution to the simultaneous localization and map building (SLAM) problem. *Robotics and Automation, IEEE Transactions on*, 17(3):229–241, 2001.

[23] FrankDellaert. http://www.cc.gatech.edu/ dellaert/assets/images/autogen/a_sonar.gif. Accessed: 2013-05-10.

[24] SICK AG · Germany. Laser measurement sensors of the lms100 product family. In *PRODUCT INFORMATION*, 2009.

[25] Giorgio Grisetti and Cyrill Stachniss and Wolfram Burgard; . http://openslam.org/gmapping.html. Accessed: 2013-21-09.

[26] G. Grisetti, C. Stachniss, and W. Burgard. Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters. *Robotics, IEEE Transactions on*, 23(1):34–46, 2007.

[27] Giorgio Grisetti. An Incomplete Scan Matching Tutorial. *gmapping software module*.

[28] F. Gustafsson, F. Gunnarsson, Niclas Bergman, U. Forssell, J. Jansson, R. Karlsson, and P.-J. Nordlund. Particle filters for positioning, navigation, and tracking. *Signal Processing, IEEE Transactions on*, 50(2):425–437, 2002.

[29] D. Hahnel, R. Triebel, W. Burgard, and S. Thrun. Map building with mobile robots in dynamic environments. In *Robotics and Automation, 2003. Proceedings. ICRA '03. IEEE International Conference on*, volume 2, pages 1557–1563 vol.2, 2003.

[30] J. Huang, D. Millman, M. Quigley, D. Stavens, S. Thrun, and A. Aggarwal. Efficient, generalized indoor WiFi GraphSLAM. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 1038–1043, 2011.

[31] Shengluan Huang and Jingxin Hong. Moving object tracking system based on camshift and Kalman filter. In *Consumer Electronics, Communications and Networks (CECNet), 2011 International Conference on*, pages 1423–1426, 2011.

[32] Jorge Cham. http://www.willowgarage.com. Accessed: 2013-24-06.

[33] Konrad Banachowicz. https://github.com/rcprg-ros-pkg/rcprg_laser_drivers. Accessed: 2013-21-09.

[34] Nosan Kwak, Beom-Hee Lee, and K. Yokoi. Result representation of Rao-Blackwellized particle filtering for SLAM. In *Control, Automation and Systems, 2008. ICCAS 2008. International Conference on*, pages 698–703, 2008.

[35] H. Lategahn, A. Geiger, and B. Kitt. Visual SLAM for autonomous ground vehicles. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 1732–1737, 2011.

[36] Microsoft. http://support.xbox.com/xbox-360/kinect/kinect-sensor-components. Accessed: 2013-11-08.

[37] M. Montemerlo and S. Thrun. Simultaneous localization and mapping with unknown data association using FastSLAM. In *Robotics and Automation, 2003. Proceedings. ICRA '03. IEEE International Conference on*, volume 2, pages 1985–1991 vol.2, 2003.

[38] T.K. Moon. The expectation-maximization algorithm. *Signal Processing Magazine, IEEE*, 13(6):47–60, 1996.

[39] Oxford dictionary. http://oxforddictionaries.com. Accessed: 2013-11-11.

[40] Sebastian Thrun. https://www.udacity.com/course/cs373. Accessed: 2013-03-05.

[41] Troy Straszheim and Morten Kjaergaard and Brian Gerkey and Dirk Thomas. http://www.ros.org/wiki/catkin. Accessed: 2013-05-10.

[42] Tully Foote. http://www.ros.org/wiki/rosmake. Accessed: 2013-05-10.

[43] Tully Foote and Eitan Marder-Eppstein and Wim Meeussen. http://wiki.ros.org/tf. Accessed: 2013-15-10.

[44] G. Tuna, K. Gulez, V.C. Gungor, and T. Veli Mumcu. Evaluations of different Simultaneous Localization and Mapping (SLAM) algorithms. In *IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society*, pages 2693–2698, 2012.

[45] Junfeng Yao, Chao Lin, Xiaobiao Xie, A.J. Wang, and Chih-Cheng Hung. Path Planning for Virtual Human Motion Using Improved A* Star Algorithm. In *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*, pages 1154–1158, 2010.