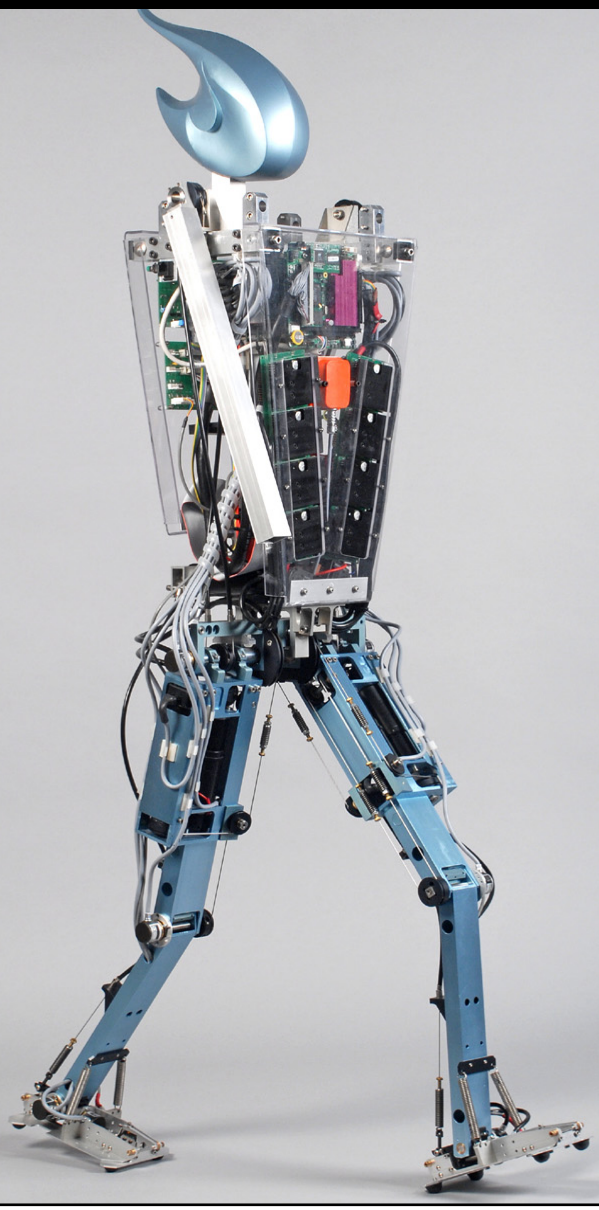
Gaining by forgetting

Towards long-term mobile robot autonomy in large scale environments using a novel hybrid metric-topological mapping system

C.J. Lekkerkerker

Master of Science Thesis



TU Delft Delft University of Technology

Delft Center for Systems and Control

Gaining by forgetting

Towards long-term mobile robot autonomy in large scale environments using a novel hybrid metric-topological mapping system

Master of Science Thesis

For the degree of Master of Science in Systems and Control at Delft University of Technology

C.J. Lekkerkerker

October 9, 2014

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

Delft University of Technology
Department of
Delft Center for Systems and Control (dcsc)

The undersigned hereby certify that they have read and recommend to the Faculty of Mechanical, Maritime and Materials Engineering (3mE) for acceptance a thesis entitled

Gaining by forgetting

by

C.J. Lekkerkerker

in partial fulfillment of the requirements for the degree of

Master of Science Systems and Control

Dated: October 9, 2014

Supervisor(s):

Dr. G.A.D. Lopes

Dr. Ing. M. Rudinac

Reader(s):

Prof. Dr. R. Babuška

Ir. A. Chandarr

Abstract

The emerging of mobile robots in everyday life scenarios, such as in the case of domestic care robots, is highly anticipated. Much research has been carried out to make robots more capable of performing tasks in our everyday environments. Despite major progress over the last decades, many hurdles are still to be taken. In the field of robotic mapping, which studies how robots can generate a map (internal representation) of their environment, modern Simultaneous Localization and Mapping (SLAM) methods allow robots to map their environment and be aware of where they are in that environment. Such maps can then be used for robot navigation, which allows a robot to travel from one place to another safely and autonomously. State of the art SLAM methods still show large limitations in their real world applicability. First and foremost, they are limited in the size of environment they can handle as maps grow inconsistent when environments get too large, or they cannot handle multi-story buildings for example because they are designed to only map in 2D. Performance even becomes significantly worse if one limits oneself to using affordable sensors. Secondly, modern SLAM algorithms still struggle with the tasks of building a map that is metrically consistent with the real world (that is, the map and a ground truth floor plan should align). Thirdly, the generated maps show obstacles (like walls), but do not give any other semantic details on them. For example, the map does not tell what places are rooms and what places are a corridor.

In this thesis, it is investigated how robotic mapping and robot navigation could benefit from a human inspired approach to these tasks. Humans do not create floor plans, but remember their environments in terms of concepts. These concepts are then linked in a relative way, and places are connected by fuzzy, relative defined connections. The relatively new study of semantic mapping aims at integrating these concepts (semantics) into robotic mapping. However, so far these systems have been built on top of a traditional SLAM method.

Parallel to this new development of semantic mapping, this thesis proposes an architecture, which we named LEMTOMap (Large Environment Metric TOpological Mapping system), that generates and handles maps in a relative way. It specifies mapping, localization and navigation in a way in which metric consistency of the map is no longer a requirement on a larger scale (e.g. that of a faculty building or larger).

The main contributions of this thesis are captured by the LEMTOMap architecture. LEMTOMap introduces a new topological mapping paradigm that allows the robot to generate a

map that is metrically consistent on a local scale, but does not require metric consistency on a larger scale. This way, the main challenge of modern SLAM - limiting metric inconsistency - is reduced to a challenge of subordinate importance. Additionally, a new grid map SLAM algorithm is introduced, named Rolling Window GMapping (RW-GMapping).

To verify the expected performance enhancements of the LEMTOMap system architecture, LEMTOMap has been partially implemented and tested in simulated experiments. The experiments confirm the main benefits of LEMTOMap, mostly in terms of improved overall time and space complexity.

The thesis concludes with a range of advices for future work. Part is aimed at the further implementation of LEMTOMap, and part at improving LEMTOMap beyond its current specification. Also, a performance issue of the original GMapping algorithm was detected and suggestions are made on how this should be improved.

Table of Contents

Preface

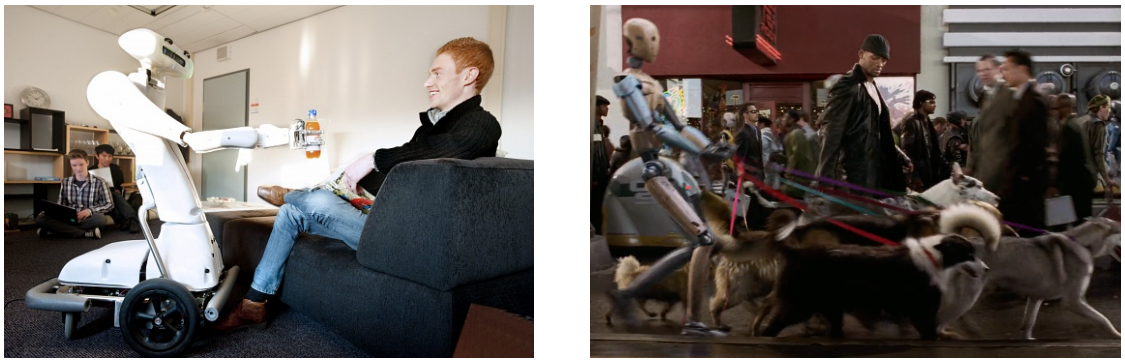
This thesis was performed at the Delft Center for Systems and Control (DCSC) in collaboration with BioMechanical Engineering (BMechE), which are both departments at the Delft University of Technology (TU Delft), faculty Mechanical, Maritime and Materials Engineering (3mE).

Chapter 1

Introduction

Mobile robots are becoming more and more versatile nowadays. As their capabilities rapidly expand and improve, an increasing amount of research is dedicated towards making mobile robots better capable of handling everyday human tasks, such as driving a car around in a city, go to the supermarket, cleaning the house, etc. Such mobile robots are receiving an increasing amount of attention, as commercially viable applications are on the horizon. Governments are willing to invest in them, for example to compensate for the lack of human resources in health care in a time of population aging. Mobile robots could be used as assistants to elderly people or people with a disability by clearing up dishes, doing groceries, cleaning the house, alarming when the person has fallen, etc. Applications are also seen in Search and Rescue (SAR) missions in case of disasters [Nüchter et al., 2006]. Robots can generally stand heat, radiation and poisonous gasses better than humans and they could for example be small enough to work their way through the ruins of a collapsed building. Another example of an autonomous mobile robot is a self-driving car. In September 2013, Tesla Motors announced that they plan to build a car that can drive 90% of travelled distance autonomously within 3 years. As a last example, robots are expected to become suitable for doing patrolling tasks.

Most of these tasks are currently possible to a limited level at best, and can often only be performed well under restricted circumstances (see Figure 1-1). The future of mobile robots as our companions is dependent on their ability to understand, interpret and represent their environment in an intelligent and human compatible manner. In this thesis, the focus is on how the perceived environment can be processed in a compact, robust and natural way, to facilitate long-term robot autonomy, including robust localization, navigation, and path planning of the robot, while also forming a basis for a more natural way of human-robot interaction (HRI). By *localization* it is meant that the robot knows where it is, by *navigation* and *path planning* it is meant that the robot knows where to go and how to get there. Interacting in terms as 'get me an apple from the kitchen' instead of a complete command including apple feature descriptions and locations, can be seen as an example of what is meant by natural HRI.
To enable navigation and path planning, a robot usually utilizes a mapping of its environment. For this research, it is considered that such a mapping is not provided a priori; the robot needs to build this mapping by itself, like humans do when encountering a new environment. Imagine



(a) (b)

Figure 1-1: Mobile robots can be used for all kinds of tasks. *(a)* Delft University of Technology personal robot Eva can do some simple tasks, such as bringing you a drink. *(b)* In the movie *I,Robot*, which plays in 2035, human-like robots help people with all kinds of tasks, such as walking the dogs. We still face many challenges before we get to this point in mobile robotics.

for example a person who starts a new job and gradually learns the layout of the office building over time. The two main capabilities involved are mapping and robot navigation, of which a short introduction is given in the next section.

1-1 Robotic mapping and robot navigation

Creating an internal representation of the robot's environment is called *robotic mapping*. Generally, such a map is stored as a floor plan. Map-learning is tightly related to *robot localization*. The combined challenge of mapping and localization is generally referred to as the Simultaneous Localization and Mapping (SLAM) problem. As the robot's sight is limited, it needs to move around to increase the size of the area that is mapped. Adding new data to the map needs an accurate localization of the robot, to make sure that new data is correctly inserted in the current map. However, accurate localization in an unadapted environment requires an accurate mapping. In addition to *odometry*[1], aligning the current perception of the environment to the map can significantly improve the accuracy of the localization (using odometry on its own for localization is generally not accurate enough and very prone to errors). To summarize: localization is needed to support mapping and mapping is needed to support localization. More information on the SLAM problem and its solutions can be found in Appendix A. Building the map requires exploration by the robot, which can be performed by autonomous exploration, or by manual teleoperation of the robot by a human controller.

Using the map, which in its most general sense should be seen as an internal representation that the robot has of its environment, the robot can perform *robot navigation* in that map. It can use the map, together with current sensor measurements, to navigate the robot to some goal within the map. To do so, it generates control commands for robot motion. The overall interplay of mapping and navigation is shown in Figure 1-2. Goals can be provided to the

[1]Odometry is the use of moving sensors to estimate change in position over time. In robotics, odometry is commonly based on rotary encoders that measure the number of wheel rotations. This way, change in pose can be estimated.

navigation module by for example an external source/user, or by the robot self, e.g. in case of autonomous exploration.

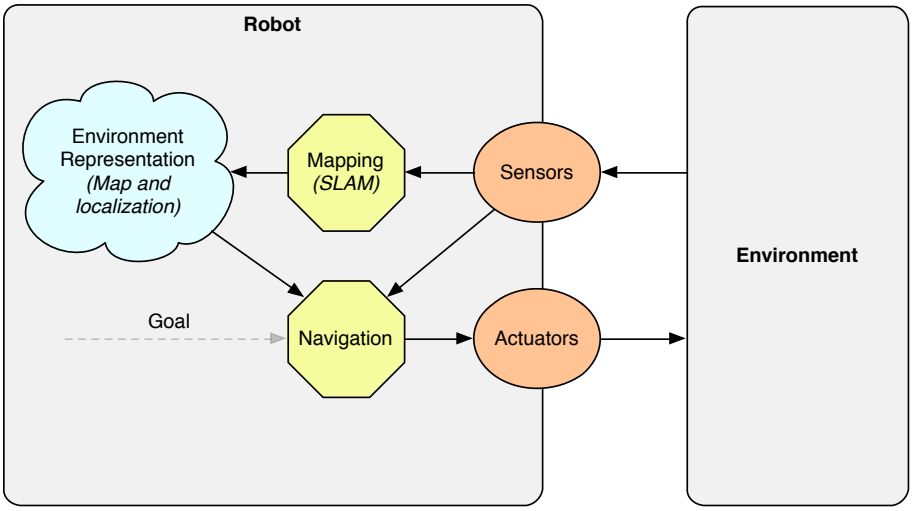


Figure 1-2: Roles of mapping and navigation in a mobile robot.

Popular state of the art SLAM algorithms suffer from several weaknesses and limitations, that limit applicability to situations involving long-term robot autonomy, large scale environments, and high-level robot behaviour. By long-term autonomy it is meant that he robot should by capable to operate without human intervention for several days or even weeks, months or years. By large scale environments, large multi-story buildings such as complete faculty building is meant, or even a full campus, city or country. By high-level robot behaviour, it is meant that a robot can handle complex tasks based on simple instructions only, such a handling the request to get someone a glass of water. In modern SLAM, map storage size grows very large for large environments, while maps only provide limited information about the environment. Most SLAM algorithms assume a static world, and consequently perform poor at handling environment dynamics such as moving people or seasonal changes. The maps are defined in one coordinate frame in which everything should be metrically consistent with the real world. As error accumulation in mapping and localization is virtually impossible to avoid in GPS denied scenarios (such as indoor scenarios), this goal becomes impossible to achieve for large scale GPS denied scenarios. This will eventually cause major inconsistencies in the map, which in its term significantly limits the usability of the map for navigation. All of these weaknesses and limitations will be discussed in more detail in Chapter 2.

Clearly, humans can deal with the challenge of mapping, localization, and navigation relatively well when compared to mobile robots. Humans deal with these challenges much differently from how it is currently solved in robotics. Humans do not try to create and store detailed floor plans or even 3D representations of their environment. Instead, humans create powerful abstractions, storing most information in terms of human concepts. For an office floor, a human would for example remember that there was a long corridor, with several offices bordering at that corridor, and an elevator at the end of the corridor. When a person needs to navigate somewhere, it knows how to get there without exactly knowing where it is, a person will just recognize places and know where to go left, right, through a door, etc. There

are two main qualities that humans show here which robots lack. Firstly, the ability of humans to abstract the perceived environment to *semantics*[2]. Semantic mapping is a relatively new approach to robotic mapping, that tries to achieve something similar for robots. More information on semantic mapping can be found at the end of Section 2-1, and in Appendix B. Secondly, humans store their representation of the world in a relative way; they do not have one coordinate frame in which all places they know are given a position.

In this thesis, it is investigated how robots could learn from humans to become more capable of dealing with large scale environments. Based on the findings, a new mapping and navigation system architecture will be proposed.

The main research question can be formulated as:

> *How can robots take inspiration from humans to perform better in mapping, localization and navigation for large scale, real world environments with the goal to act autonomously for long periods of time?*

1-2 Contribution of the thesis

In this thesis a new mapping, localization and navigation system architecture is introduced, named LEMTOMap, that enhances mobile robot performance in large-scale environments and forms a step towards long-term robot autonomy (which requires a robust mapping and navigation system). The LEMTOMap architecture is partially implemented as a software package for the Robot Operating System (ROS) and is available as open-source software[3]. The software package can be easily used on any robot that can provide odometry, laser range data, and can be controlled to move around by ROS.

The main novelties introduced in this thesis are captured by the LEMTOMap system architecture, and are summarized below.

1. A novel mapping, localization, and navigation architecture named LEMTOMap is introduced, which allows reliable mapping, localization and navigation in large scale environments while having relatively low time and space complexity.
2. As part of LEMTOMap, a novel hybrid topological and grid map SLAM system is introduced, that is robust against global metric inconsistencies.
3. A novel rolling window grid map SLAM algorithm is introduced.

Foremost, LEMTOMap provides a human-inspired approach to mapping, localization and navigation that allows it to perform at relatively low time and space complexity in large scale, gps-denied, indoor scenarios. LEMTOMap is compared to the current state of the art. The comparison is made against GMapping for mapping [Grisetti et al., 2007] and localization, and the Dynamic Windowing Approach for navigation [Fox et al., 1997].

[2]Semantics is 'the study of meaning', and is used in robotics and Artificial Intelligence (AI) to denote the usage of human concepts such as 'corridor', 'large room', 'book', 'door', etc. Semantics is about assigning meaning/labels to data.

[3]Available at https://github.com/koenlek/ros_lemtomap

Secondly, a novel hybrid topological and grid map SLAM system is introduced as part of LEMTOMap, which focusses on local metric consistency, without requiring global metric consistency. The topological part of the hybrid system is used for the full scale environment, and the grid map part is used only at a local scale, as it moves as a rolling window along with the robot. This way, the main challenge of modern SLAM - limiting metric inconsistency - is reduced to a challenge of subordinate importance.

Thirdly, as part of the LEMTOMap, a new grid map SLAM algorithm is introduced which can operate as a rolling window, which we named RW-GMapping (Rolling Window GMapping).

The system gains by forgetting, as suggested by the title of this thesis. The system forgets anything of the grid map that is not local. This, combined with the way the topological mapping is defined, allows the system to handle global metric inconsistencies in a robust way. Additionally, space and computational complexities are reduced. The space complexity of the overall map is greatly reduced when compared to popular state of the art occupancy grid mapping SLAM algorithms such as GMapping, as the hybrid map requires much less space to store. The time complexity of mapping, localization and navigation is reduced significantly, when compared to current state of the art. The abstraction to a topological map gains better robustness to metric inconsistencies and provides a way to store semantic knowledge in the nodes of the topological graph. Overall, the system is less resource demanding, allowing more resources to be available to demanding tasks such as computer vision and deriving environment semantics.

Part of the LEMTOMap is implemented in ROS, and part is left future work to implement. In addition to the future work stemming from the currently unimplemented part, a clear summary of additional future work suggestions is given in the Discussion chapter. LEMTOMap will enable other forms of novel functionality to be investigated in subsequent research.

1-3 Outline

The rest of this thesis will be organised as follows. Chapter 2 gives background information on the thesis. It will treat the topics of robotic mapping and robot navigation together with their related challenges/issues. It will also highlight the methods from these fields of research that have been used as a basis for LEMTOMap. Additionally it discusses how humans and robots compare when solving such tasks. Chapter 3 describes LEMTOMap. The partial implementation of LEMTOMap will be discussed in Chapter 4. In Chapter 5, experiments are performed to test the implemented system. The report concludes with Chapter 6 and 7, which summarizes the main conclusions and provide a discussion on future work respectively. At the end of the report, several Appendices, a Bibliography, Glossary, and an Index are provided. In the index, terms can be looked up when desired. Pages with the clearest definition are marked bold in the index. Words marked italic in the report can generally be found in the index.

Chapter 2

Background information

In this chapter, background information is provided on the topics of robotic mapping and robot navigation, together with their related challenges/issues. Additionally, for both topics, the methods that were eventually used to help create the novel system implementation described in Chapter 4 will be discussed. After that, section 2-5 discusses how humans and robots compare when solving mapping and navigation tasks. The information in this chapter is regarded essential knowledge for a proper understanding of the rest of the thesis.

The described used methods were designed to work with Robot Operating System (ROS), an open-source, community driven software project that provides all kinds of tools to develop and test robotic systems. An introduction to ROS can be found in Appendix C.

2-1 Robotic mapping overview

Robotic mapping is commonly referred to as Simultaneous Localization and Mapping (SLAM), on which a brief introduction is given in the next section. Several types of mapping can be used (individually or combined) to perform SLAM, including metric mapping, topological mapping and semantic mapping, which will be discussed in the three subsequent sections. A more detailed introduction to SLAM can be found in Appendix A.

Simultaneous Localization and Mapping (SLAM)

The Simultaneous Localization and Mapping problem is considered one of the most important perceptual challenges in robotics. To build a map of an environment, a robot needs to move around, as otherwise it will only perceive a small part of the environment. By moving around, it perceives more of the environment. Next, these perceptions need to be integrated into a map of the environment. This is where it becomes challenging. To integrate subsequent perceptions, the change in position of the robot between those perceptions needs to be known. Generally, it is hard to measure this change in position. Most commonly, odometry is used to get an

estimate of the change in position. But as odometry is prone to errors and is generally not very accurate, the map is often used to improve the estimate of the robots movement. So, for generating a map, the position (or localization) of the robot needs to be known, and for localizing the robot, the map needs to be known. Hence the name Simultaneous Localization and Mapping problem.

The main challenge is controlling the error accumulation that will inevitably happen in localization. This error accumulation will happen as the position estimate is based on an integration of data, without any way to measure the ground truth. This error accumulation is shown in Figure 2-1a. Two important issues arise from this. Firstly, maps become metrically inconsistent with the real world. A map built by a robot does not nicely overlap a true floor plan of a building floor. In general, errors accumulate more as the robot travels further from its starting point, so these inconsistencies become bigger the further the robot is from its starting point. Secondly, as a result, the robot will run into the challenge of closing loops. If the robot has travelled a circular path, e.g. four corridors that are connected in a rectangle, the robot will come back at a place where it has been before. Because of the error accumulation, the localization will likely not end up in the same place, and thus the map will also not nicely close. This is illustrated in Figure 2-1b. If a *loop is closed*, a correction can be back-propagated to correct map and localization. As such, loop closing can also be seen as a way to limit error accumulation. A possibility to limit error accumulation would be the usage of a GPS, which gives a way to measure the ground truth (albeit very noisy). Unfortunately, GPS cannot be used indoors. Hence, assuming that the robot keeps exploring more of its environment, it can be concluded that controlling localization accuracy is unstable by definition for indoor scenarios; the error accumulation can only be damped.

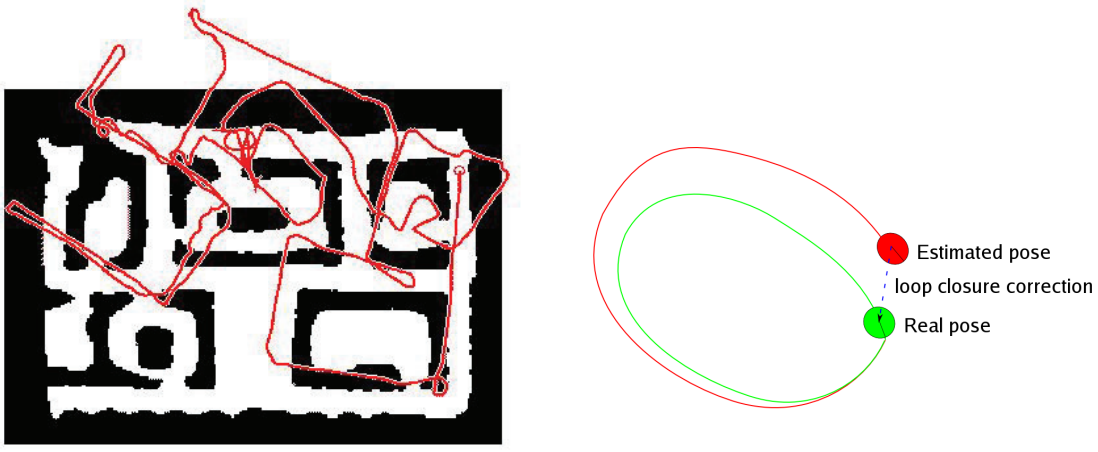
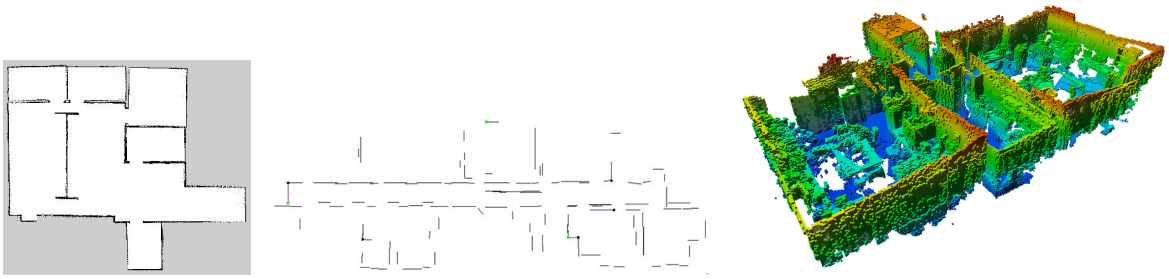


Figure 2-1: *(a)* Large localization errors can arise from small odometry errors, at the beginning a small rotational measurement error is made (lower right corner), resulting in big errors after translational movements. The black map represents the ground truth map. Figure is adopted from [Thrun, 2002]. *(b)* Localization errors cause the loop closing problem. If the correspondence is recognized, the accumulated localization and mapping errors can be corrected by back-propagating the correction. If errors get too big and correspondence is not recognized, it will inevitably result in faulty maps.

Next, different types of maps that can be used to store a robots internal representation of the environment will be discussed.

Metric mapping

A very common form of a metric map is the *occupancy grid map*, which is like a floor plan. An example is shown in Figure 2-2a. The map is a grid of cells, and each cell can be either marked occupied, free, or unknown. Often, values in between free and occupied are also possible to give a cell an arbitrary percentage certainty of being occupied or free. An occupancy grid map is called a *dense* map; the map covers the full space of the mapped part of the environment. A sparse map, is a map that only uses sparse features that together build the map. Figure 2-2b shows a sparse map that stores line features. This only needs to store a begin and end point of the line, and is thus much more compact than the occupancy grid map, which needs to store every pixel. Other sparse maps consist of points, or a combination of geometric shapes. Formerly, most maps were sparse, as dense mapping algorithms have much higher computational complexity. Nowadays, mainstream laptops have become powerful enough to generate dense maps in real time. As dense maps generally contain more information and detail, these have become the more popular type of map.
In addition to two dimensional maps, 3D maps exist as well. A dense example are maps build by OctoMap, see Figure 2-2c.



a **b** **c**

Figure 2-2: *(a)* A dense map. An occupancy grid map models a map as a grid which cells are occupied, free, or unknown. Black is occupied, white is free and grey is unknown (unexplored). Figure adopted from [Gallart del Burgo, 2013]. *(b)* A sparse map. The map only stores line. A line can be defined by a begin and end coordinated. Figure adopted from [Zender et al., 2008]. *(c)* OctoMap can generate 3D grid maps.

In most popular implementations of SLAM generating metric maps, failing to close loops result in maps that will contain serious glitches. Such glitches can cause problems when using the map to perform robot navigation. An example of such a map with glitches resulting from failed loop closure is shown in Figure 2-3. As new mapped parts will continue to map right on top of other parts of the map, the map will become unfit as a basis for path planning.

Metric maps are generally created using odometry data and laser range scanners. The latter can record perimeters of the current place seen from a robot (see Figure A-1 on page 80 for an example of such a recording).

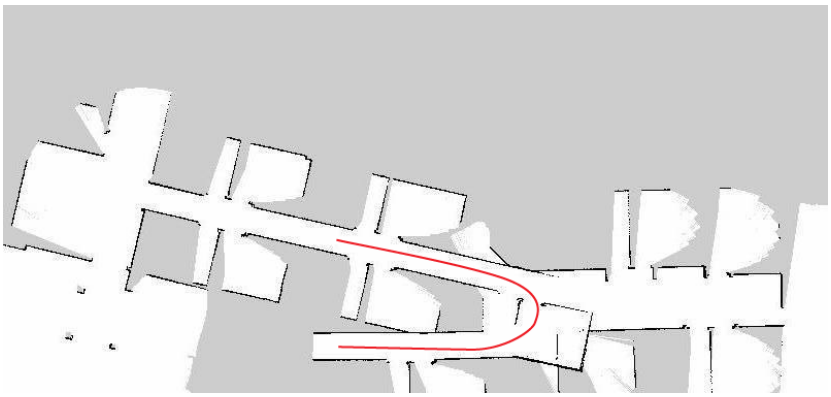
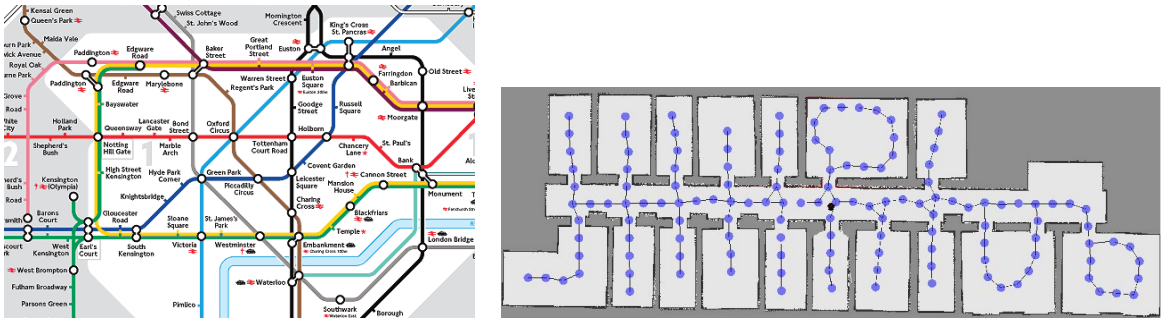


Figure 2-3: This map is the result of a failed loop closure. For navigation, the robot could for example plan the path shown by the red line. In fact, the corridor at the beginning of the red line and at the end, are the same corridor and they should overlap. Also notice the wall goes through the middle of a room at the curvature of the path. When continuing mapping, the glitches will become worse and worse.

Topological mapping

Topological maps consist of points connected by lines. These points are called *nodes* (or *vertices*) and the lines are called *edges*. A common example of a topological map is a subway map, such as the one in Figure 2-4a. Topological nodes are not necessarily defined in some world frame. That is, they do not need to have some position defined. However, for topological maps, nodes are often given an explicit location in the map. In the case of a subway map, the nodes only have a position that coarsely complies with the real world places they represent. Strictly speaking, edges do not necessarily mean that there is a *directly navigable* path between two nodes, it only defines that there is some relation between the nodes. Normally, it is safe to assume that an edge does imply direct navigability between the nodes. Figure 2-4b shows an example of a topological map used in robotics. It is shown on top of a metric map, as a reference. This is an example of a hybrid combination of two map types.



a **b**

Figure 2-4: *(a)* A subway map is a common example of a topological map. *(b)* A topological map on top of a metric (occupancy grid) map generated by a robot. Figure adopted from [Gallart del Burgo, 2013].

Basic topological maps contain less detail than metric maps (imagine the metric map in the

background of Figure 2-4b being removed; you would not know how the rooms look like for example). Because topological maps generally contain less detail, it is also harder for a robot to localize itself in the map, or to close loops. On the other side, nodes can be expanded with all kinds of information, you can store local sensor measurements such as laser scans in nodes, or other information such as semantic information. This can be used to overcome the previously mentioned disadvantages it has to metric maps.

Pose graphs Pose graphs are a special kind of topology. In pose graphs, robot poses are periodically stored in nodes. Between subsequent nodes, odometry based edges are added. Edges store what their length should be, so they actually define constraints between the nodes. In addition to odometry based edges, observation based edges are added; if the robot recognizes a place of another node, it adds an edge between the current node and that node which defines what the distance *should* be between both nodes. Consequently, observation based edges do *not* imply direct navigability between the nodes they connect. An example of this is shown in Figure A-11 (Appendix A, page 90). The resulting pose graph will not satisfy all the constraints, as localization errors result in inaccurate node locations. An error minimization algorithm is used to minimize the overall error, which solves the pose graph to a layout in which node poses are much closer to the places they correspond to in the real world. This is illustrated in Figure A-12 (Appendix A, page 91). In pose graph based SLAM, such pose graphs are generally optimized periodically.

Semantic mapping

Semantic mapping is a new kind of mapping that only has started to gain some more significant attention in the last 5 years or so (as opposed to over two decades of thorough research after general SLAM). A semantic map makes use of *semantics*. Semantics is 'the study of meaning' and is about assigning meaning to raw data. It involves abstraction to human concepts. In the case of mapping, that means that it involves marking portions of space as rooms, marking places as doorways, and recognizing objects and categorize them in some generic group, like 'this is a chair'. A more in depth introduction to semantic mapping is given in Appendix B.

In this thesis [Pronobis and Jensfelt, 2012] is regarded the leading work in state of the art semantic mapping. In the system presented in [Pronobis and Jensfelt, 2012], the robot combines multiple basic forms of knowledge about its environment to generate more sophisticated insights. These basic forms of knowledge are called *properties of space* in the paper, and include room geometry, room appearance, room topology, landmarks, and objects. The detection of doorways helps segmenting space into rooms. Laser scan measurements help determining the shape and size of a room. For example, if a place is elongated it becomes likely that it is a corridor. A small rectangular room might be a single office, while a large rectangular room can be a multi-person office. If it is a large rectangular room, with only one computer and a table with chairs around it, it is more likely a professors office. Room topology also contains information: a door at a kitchen is more likely to lead to a corridor than to another kitchen. Appearance is the overall look of a place; outdoor scenarios look very different from indoor scenarios. Global image descriptors can be used to capture and compare such appearances. Objects are generally detected using cameras or RGBD cameras (e.g. a Kinect), while other properties of space are found using other sensors. The overall system is multi-modal: it

combines several kinds of sensory information to build its overall representation of the environment. Believes about properties of space and higher semantics (such as room types) are defined as probability distributions and relations between them can be either probabilistic or deterministic. The overall representation of the environment created by the robot is captured in what the authors call the *conceptual graph*, which is solved by an approximate solver. The probability distributions in this graph relate on each other in both directions, e.g. it is more likely to find a stove in a kitchen, and when the robot has a strong belief that it is in a kitchen, it knows that there is a larger chance to find a stove in the room. An overview of the system is given in Figure B-2 (page 96).

The semantic map allows the robot to reason about its environment in a way similar to human common-sense. For example, if the robot is asked to get a glass of water, it can reason that it is probably best to go to the kitchen and search for a glass and a water tap there. With such reasoning, a semantic map can enable much more advanced forms of robotic behaviour. Also, it can support better human robot interaction, as the robot learns to deal with and think in same concepts as humans do.

The overall semantic map of [Pronobis and Jensfelt, 2012] is actually a combination of a sparse metric map (line segments representing walls), a topological map, and a semantic map (the conceptual graph which is linked to the topological map).

More information on [Pronobis and Jensfelt, 2012] is given in Appendix B.

2-2 Robotic mapping - used methods

In this section, the main mapping related methods that helped creating the implementation discussed in Chapter 4 are discussed.

2-2-1 Occupancy grid map SLAM - GMapping

GMapping was introduced in [Grisetti et al., 2007], and was originally made available as an open source implementation on openslam.org. Later, an implementation for ROS (essentially a wrapper) has been made available, which can be found at wiki.ros.org/gmapping. GMapping provides an occupancy grid map and localization within that map based on laser range scan measurements and odometry data. It uses a Rao-Blackwellized particle filter. In GMapping, each particle represents a possible trajectory of the robot. Every particle has its own grid map, which is constructed as if the poses of its trajectory are known with certainty. In other words: for each specific particle a map is constructed by fusing measurements as if there is no uncertainty about the poses out of which its trajectory is made of. To limit the overall computational complexity, it is useful to limit the number of required particles by using the particles efficiently. In GMapping, the probability distribution used to generate particles is further improved (compared to Fastslam [Montemerlo et al., 2003]) and resampling is improved to decrease the need for particles. It also introduces a two-step approach to the sampling which first samples based on odometry and then on scan-matching, which improves performance when it encounters multi-modal uncertainties, that can be encountered when closing loops. This drastically improves loop closing performance without adding much overhead, compared to its preceding state of the art algorithms. The GMapping system is able to run in real-time on basic hardware in general office floor environments[1]. The pseudocode description of the GMapping algorithm can be found in Appendix E, Algorithm E-1.

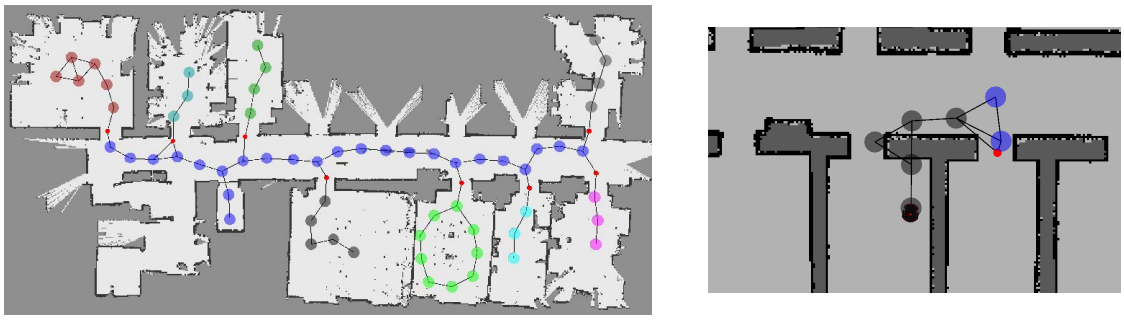
2-2-2 Topological mapping - ROCS-ROS

In [Gallart del Burgo, 2013], a partial implementation of ROCS (toolkit for RObots Comprehending Space) in ROS is described, called ROCS-ROS. The semantic mapping system by [Pronobis and Jensfelt, 2012], is based on the ROCS[2] toolkit. It must be said that ROCS-ROS (which was a masters thesis project) covers much less functionality than ROCS, and significantly differs on several components from ROCS. For example, ROCS-ROS uses dense SLAM (occupancy grid map build using GMapping), as opposed to the sparse SLAM implementation used in ROCS. Furthermore, it cannot perform object classification in real time, and it does not include the conceptual graph building/solving components. The topological mapping in ROCS-ROS creates a new node as soon as the closest node is more than $1m$ away. Edges are created between nodes if the distance is less than $1.3m$. The distance check is the only check for edge creation, which means that edges can also be created through walls, which is of course undesired. Node locations are defined in a global frame and locations are never

[1] To close loops reliably, it turns out to be advisable to use a fairly long range laser scanner (e.g. several tens of meters as used in the original article) which cost currently several thousands of Euros. This is clearly a big disadvantage when aiming to design a low cost system.

[2] Although ROCS itself seems to be available from https://github.com/pronobis/rocs/, the code on Github is incomplete and not functional currently. This has been confirmed by A. Pronobis.

updated. Consequently it will fail for large maps, as map updates can cause large shifts in parts of the map, resulting in nodes not being aligned with the map (nodes could end up on top of walls for example). ROCS-ROS also includes doorway detection to group nodes into areas (rooms). However, the topological mapping system assumes that a detected door is always a crossed door. In practice, this causes the robot to also create a new area when it simply travels close enough into the doorway and then returns, which is undesired as illustrated in Figure 2-5b (the performance according to the article itself is in Figure 2-5a). ROCS-ROS does not include any topological navigation.



a b

Figure 2-5: Node colors correspond to an area ID. *(a)* The performance of ROCS-ROS according to the article. Image adopted from the original thesis [Gallart del Burgo, 2013]. *(b)* The performance of ROCS-ROS reproduced. When exiting a doorway without passing it, a new area is created. When travelling close along walls, edges can go through walls.

2-2-3 Topological mapping - ROS topological_navigation software stack

The `topological_navigation` software stack is described in [Konolige et al., 2011], and was released in 2011 for the ROS Diamondback release, which is deprecated and unsupported nowadays. Unfortunately, after that it has never been updated to work with newer versions of ROS. After a considerable amount of changes to the original code, the software stack was usable under the still supported ROS Fuerte release. The mapping and navigation system of [Konolige et al., 2011] uses a collection of local occupancy grid maps. If the robot gets too close to the border of the current local grid map it switches to the neighbouring local grid map, or creates a new local grid map if no appropriate neighbour exists. A pose graph SLAM algorithm (see Appendix A) is used to minimize the overall mapping and localization errors. The local grid maps are fixed rigidly to some central node, which is part of this pose graph. New nodes are added to the pose graph if the distance to the closest node is larger than some fixed threshold. Edges are added based on odometry and scan-matching (see Appendix A-3). The navigation graph (the topology that will be used for robot navigation) is derived from the pose graph by searching for navigable edges. All nodes in a local grid map that have a relative distance within some threshold are checked for their navigability. To determine navigability, all obstacles are inflated by the robots radius and a straight line is drawn between the nodes. If it does not hit any inflated obstacle, the line is navigable. Navigational edges are not necessarily part of the set of pose graph edges. The stack also provides topological navigation, which will be discussed later in this chapter (Section 2-4-2). An example of a map build using the `topological_navigation` stack is shown in Figure 2-6.

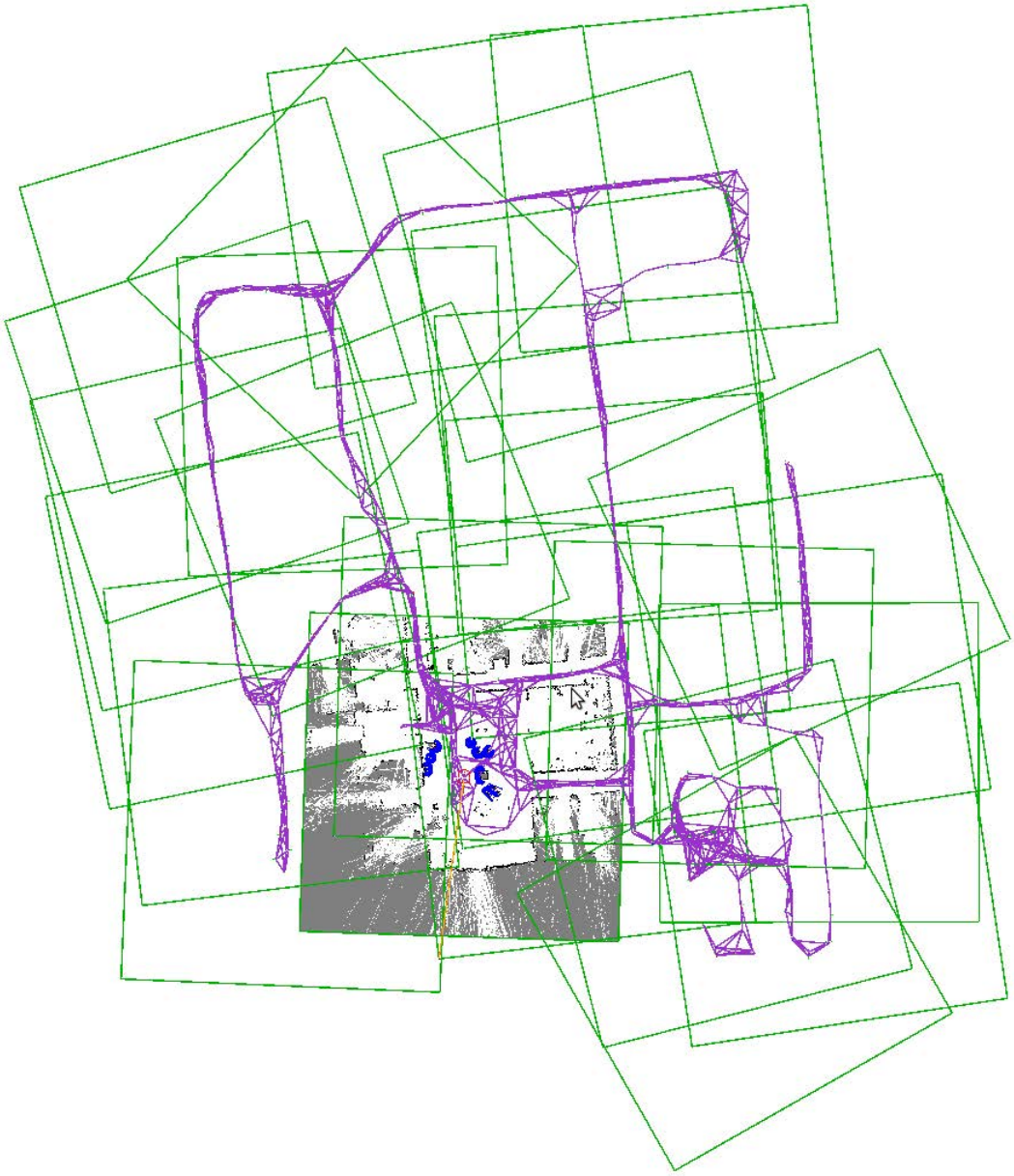


Figure 2-6: A map build using the ROS topological navigation stack. The green squares represent the outlines of the local grid maps, of which one is always active (currently at the bottom). The purple lines form the pose graph. The navigation graph is not shown. The inflated obstacles are blue, and the footprint outline of the robot is shown in red in the middle of the active grid map (hard to see). Figure adopted from [Konolige et al., 2011].

2-3 Robot navigation overview

Robot navigation is about how the robot should move around in its environment to get to some destination. An important part of robot navigation is *path planning*. If a robot wants to go to a certain point, it needs to plan a path that will safely get him there. Additionally, robot navigation covers the generation of the actual control signals that make the robot perform the required movements. For this section on robot navigation, the focus will be on path planning. In robot navigation, the term *navigability* is used to denote if it is possible to navigate the robot between two points. The term *direct navigability* can be used to indicate if a robot can directly go (i.e. in a straight line) from one point to another.

Metric path planning

Metric path planning is performed by finding a shortest or lowest cost path in a *costmap*. The most plain form of such a costmap is a gridmap in which free cells have some small value cost, and obstacle cells have infinite cost. Now, how does one calculate the path with the lowest cost in such a costmap? In ROS, Dijkstra's algorithm is used for this by default. An alternative implementation is based on the A* algorithm. These algorithms are so called *single source shortest path algorithms* and will be further discussed later in this chapter. Such algorithms operate on graphs that are built up out of vertices and edges. The costmap can be converted to a graph in which the grid cells become the vertices and the costs are assigned to the edges between these vertices. As such, the metric planning is effectively turned into a topological path planning by converting the costmaps into an undirected graph.

Topological path planning

For topological path planning, Dijkstra's algorithm and the A* algorithm can be used as well. When planning on a global level, a topology can greatly reduce the time required for calculating a global path, as it contains much less vertices and edges when compared to a costmap that is converted to a graph. Locally, you will always need to be able to avoid obstacles in metric space, so local planning can still be performed best in metric space, as described in the previous section. Therefore, topological navigation in robotics is normally in fact a hybrid approach: locally, paths are planned based on a metric costmap, while globally a topological map of the environment can be used.

As shortest path algorithms find the shortest route based on the costs defined in the edges, node positions do not matter for the path planning. Consequently, metric consistency of the node locations is also no requirement. The only requirement is that edge costs and connections are defined decently. The fact that topological path planning does not rely on a globally metrically consistent map, makes a big difference between metric and topological path planning.

Shortest path algorithms

In this section Dijkstra's algorithm and the A* search algorithm are discussed. Although these types of algorithms are called shortest path algorithms, they in fact calculate the path with

the lowest cost[3].

Dijkstra's algorithm Dijkstra's algorithm is an algorithm that finds the shortest path for a graph. The pseudocode description of the algorithm is shown in Appendix E, Algorithm E-2. The algorithm requires non-negative edge costs and a directed graph. As undirected graphs are a special kind of directed graphs (all edges simply need to be defined in both directions with same cost), it can also solve the shortest path problem for undirected graphs.

The algorithm will return two vectors: *dist* and *previous*. The *dist* vector contains the distance from the selected vertex (which is associated with the position in the vector) to the source vertex, while the *previous* vector tells what the previous vertex should be for the selected vertex. By picking an arbitrary destination vertex, one can find the shortest path by iteratively going through the *previous* vector. Please note that where ever the term distance is used in the pseudocode, any non-negative cost could be used.
Effectively, the algorithm simply explores all vertices neighbouring the current frontier of the explored part of the graph. At start this frontier consists of one vertex; the source. The surrounding vertices are 'explored', i.e. their cost and optimal preceding vertex is determined. The algorithm makes sure that the exploration is ordered such that the frontier vertices with the lowest total cost so far are explored first. This way, the amount of suboptimal explorations is limited. The ordering in Dijkstra's algorithm guarantees that the optimal shortest path is found eventually.

By default, the algorithm finds the shortest paths from the source vertex to any arbitrary other vertex in the graph. If desired, the algorithm can also be stopped as soon as the lowest cost path to an arbitrary destination vertex is reached.

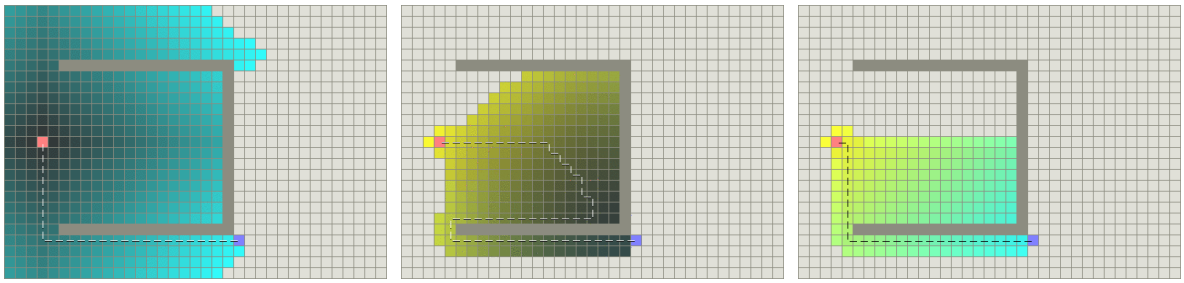
A* search algorithm The A* search algorithm uses heuristics to allow for more effective searching than Dijkstra's algorithm. Dijkstra's algorithm prioritizes the queue of to be explored vertices based on their cost (distance) from the source. Vertices with lowest cost are explored first. As a result, for an empty metric grid map, exploration progresses like a circular wave front around the starting vertex. A* improves the queue prioritization by prioritizing vertices that are more likely to lead towards the goal vertex. The plain heuristic that is often used when directly generating a graph from a grid map (as is done with metric path planning based on costmaps), is a estimated cost based on the distance between the current vertex and the destination.

The pseudocode description of the A* algorithm is shown in Appendix E, Algorithm E-3. The A* algorithm requires a destination vertex, otherwise it cannot utilize a heuristic to speed up the path search task. Like Dijkstra's Algorithm, A* returns a *dist* and *previous* vector. *dist* is sometimes also called g in the context of A*. The sum of the estimated remaining cost is determined by heuristic function h plus the current cost so far *dist* for the current vertex. This total estimate of the cost is generally denoted f. The heuristic function often takes the current vertex and destination vertex as inputs (e.g. when estimating remaining euclidean distance), but can be defined as any arbitrary function with any input if desired. As the algorithm searches for a path to a certain goal vertex, it is not guaranteed to succeed.

[3]Generally, the cost is defined to equal the distance between nodes, therefore the shortest path is de facto the same as the lowest cost path.

If the goal vertex is unreachable, it returns a failure. Dijkstra by default searches all vertices and will stop as soon as no reachable vertices are left.
As with Dijkstra's algorithm, the path can be reconstructed from the *previous* vector by iteratively backtracking the previous vertices starting from the goal vertex.

Comparison Although A* can potentially find the path in significantly less steps than Dijkstra's algorithm, in general it does not guarantee to find the optimal path, as opposed to Dijkstra's. Whether it will find an optimal path depends on the chosen heuristic and how the edges are weighted (by distance only or by other factors as well). In general, when planning in metric space, A* will outperform Dijkstra's in speed, but it does not guarantee optimal paths. Figure 2-7 shows a comparison of Dijkstra's, A* and Greedy Best-First Search. Greedy best-first search will not be discussed in detail, but is shown for comparison. It can be thought of as an algorithm that is fully driven by minimizing the distance between the current vertex and the goal vertex. A* can be regarded as an algorithm that combines the properties of Dijkstra's algorithm to minimize cost from source, and the greedy best-first search to try minimize the distance to the goal.



(a) **(b)** **(c)**

Figure 2-7: Examples of how the path finding problem is solved by three different algorithms. The dark gray shape is a wall, the coloured vertices are the explored vertices until the path was found and the dashed line is the found path. *(a)* shows the path found by Dijkstra's algorithm. *(b)* shows the greedy best-first search result, the path is clearly not optimal. *(c)* shows the A* result with a euclidean distance heuristic. Images adopted from Standford web page[4]

Dijkstra's Algorithm is in fact a special case of A*, the heuristic function is always 0 in Dijkstra's case. Dijkstra's does not explicitly define the $\mathcal{C}$ to track which vertices are evaluated already, as it is implicitly defined by $\mathcal{Q}$: $\mathcal{C}$ would equal the set of all vertices not in $\mathcal{Q}$.

[4]Images from: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html.

2-4 Robot navigation - used methods

In this section, the main navigation related methods that helped creating the implementation discussed in Chapter 4 are discussed.

2-4-1 Metric navigation - ROS navigation software stack

ROS provides an official 2D navigation stack, that is able to generate velocity commands for the mobile base of the robot, based on a goal pose. The ROS navigation stack is fairly flexible and offers a high degree of configurability.

Navigation in the ROS navigation software stack is performed using the `move_base` package, which generates velocity commands based on a goal pose and a global and local path planner. The path planners are loaded as plug-ins, which allows a relatively easy way to program a custom path planner plug-in. The path planners try to find a minimum cost path in their costmap. Each planner has its own costmap. Costs in costmaps are generated by combining cost layers, which are plug-ins, so this again allows for programming custom cost layers. Next, the working of `move_base` is described, based on a widely used configuration of `move_base`, which uses the `navfn` global planner and the `dwa_local_planner` local planner.

The global planner uses the global costmap, which uses a cost layer that is based on the current occupancy grid map as provided by SLAM (or pre-loading a known map). The obstacles in that map are marked as *lethal* cells in the costmap. Additionally, an obstacles layer adds additional obstacles based on current sensor measurements. To cover the robots own footprint size, the obstacle layer inflates all obstacles by a user specified radius, and marks theses cells as *inscribed* cells. Lethal and inscribed cells are forbidden terrain for the path planners. From the border of the inscribed cells, a user specified decay function decays the cost for a smooth transition from inscribed to free cells. The global costmap is fixed to the normal occupancy grid map, they have the same size and are defined in the same frame. The global path planner uses Dijkstra's algorithm to calculate the lowest cost path in the costmap.

The local planner refines the global path, by taking into account local obstacles and the physical limitations of the robot. The local planner will generate a local path and velocity commands, based on the global path and a local costmap. The `dwa_local_planner` uses the *Dynamic Window Approach*, as introduced in [Fox et al., 1997]. The basic steps of DWA are summarized as follows:

1. Sample velocities in the robot's control space ($\dot{x}$,$\dot{y}$,$\dot{\theta}$)
2. For each sample, perform forward simulation from the robot's current state to predict what would happen if the sampled velocity is applied for a short period of time.
3. Score each trajectory resulting from the forward simulation, using a metric that incorporates characteristics such as: proximity to obstacles, proximity to the goal, proximity to the global path, and speed. Discard illegal trajectories (those that collide with obstacles).
4. Pick the highest-scoring trajectory and send the associated velocity to the mobile base.

5. Repeat.

The local planner requires details about the robot such as its maximum speeds and accelerations, minimum speeds (as in real life, motors will block or start shocking below a certain speed), and parameters for the scoring function. The local costmap is limited in size and moves as a rolling window along with the robot. It only uses the obstacle layer and an inflation layer to inflate those obstacles. As the obstacle layer adds obstacles by measurements, the local planner can deal well with temporary obstacles, such as people.

Based on the global and local planner results, `move_base` broadcasts velocity commands that are executed by the ROS package that controls the mobile base of the robot. `move_base` also provides an interface to support recovery behaviour, which is triggered when the robot is stuck because `move_base` fails to produce a valid path (e.g. when the robot has come too close to a wall). Recovery is defined by plug-ins (that can be custom made too). Multiple plug-ins can be defined for the recovery behaviour; `move_base` will execute them in the order in which they are defined in the configuration, until `move_base` is able to produce a valid plan again.

2-4-2 Topological navigation - ROS topological_navigation stack

The `topological_navigation` stack contains a topological path planner, which uses the navigation graph to calculate the shortest route to a given goal node. The shortest topological path is calculated using Dijkstra's algorithm. Next, the last node of that path which is currently still in the current local grid map is passed to the metric `move_base` navigation module. The reasoning behind this is, that when the last possible node[5] is passed, the overall path will be shorter than when each node of the path is passed successively. Passing each path node successively could for example cause a detour in large open spaces or a swaying movement through a corridor (see for example Figure 3-9). The size of the local grid map, which is one of the parameters of the `topological_navigation` stack, is of large influence on this performance: (very) large local maps make the navigation less 'topological', while (very) small local maps still suffer from the issues of following a longer, suboptimal path caused by the generally slightly suboptimal path that is offered by the navigation graph.

The topological planner can detect when edges turn out to be blocked (for example if a door is closed or some object keeps blocking a route) for a period longer than some threshold. In that case, the edge is marked blocked temporarily (for a fixed duration), and the planner is triggered again to find a new topological route.

[5]Outside the current local grid map is impossible, as the metric planner does not know where obstacles are outside the current local map

2-5 Robot behaviour versus human behaviour

Nature has been an inspiration to many scientist in solving difficult problems found in all kinds of research, including robotics. In Artificial Intelligence (AI), machine learning is used to automatically learn optimal walking policies. Machine learning and techniques like neural networks found inspiration in what we know and understand from ourselves and nature. Another example is the Delft University of Technology dragonfly-like robot DelFly, whose flying techniques were largely found empirically; by inspecting nature.
Semantic mapping also takes inspiration from nature; by taking humans as an example. It is important however, to ask oneself to what extend mimicking nature is useful, and to what extend a robot should rather use skills that are unique or superior for robots when compared to humans. Clearly, it does not make much sense to strive for autonomous car navigation without GPS, simply because humans can do without as well. When considering how humans create mappings of their physical world, it is clear that humans do not use a strict global coordinate system. Everything seems to be defined relatively, sparsely, and semantically. Imagine yourself going to the train station from your house. It is often hard to accurately point the direction to the station from the place you start. Estimating the distance is perhaps even harder. Still, you would know how to get there, as you recognize places by salient details, such as objects and other properties of space. You know when to go left or right, and when to continue. Does this mean that robots should use such a fuzzy, relative coordinate system to store their maps as well? Maybe. It at least proves that it is possible and thus could be considered. For GPS denied places, such as indoor scenarios, building maps that are consistent in the global coordinate system has proven to be very challenging, as is discussed in Appendix A about SLAM. The importance of a map that is consistent in a global coordinate system mostly comes from the desire to close loops easily, by trying to limit the gap size of such loops. When searching for loop closures is not only driven by metric data, but also semantic data, the need for a metrically globally consistent map will likely reduce. Humans illustrate how powerful semantic data can be in mapping and navigating; humans recognize places based mostly on how they experience the place (in terms similar to *properties of space*, see description of Semantic mapping in this chapter), rather than where exactly they think they are. This is also why humans excel in a 'kidnapped robot' experiment (see Appendix A, page 87). Robots however, generally use metric tools like scan-matching.
Humans can only estimate distances, while robots can make more or less exact measurements. This is a quality that robots can use to help them perform as well as, or even better than humans; potentially reducing the need to have similarly advanced algorithms as humans have for pattern recognition, reasoning, understanding, etc. By utilizing robotic qualities that humans lack, robots can get by with less advanced artificial intelligence when compared to truly mimicking humans. Also, the overall cost in terms of effort (e.g. calculations and memory needed by an AI system, research required for the development of such a system) can be much lower. Therefore, excluding metrics completely or avoiding the use of GPS in outdoor scenarios, does not seem to be a logical step. Removing the need for a globally consistent metric map in indoor scenarios could be very interesting however. Metric data could still be used in relative descriptions, e.g. when defining a topological map, directions and distances between nodes could be defined metrically (together with a degree of uncertainty). This data can be very useful to deliver accuracy at a local level.

Another important question is to what extend robots should adapt to our environment, which

is primarily designed for humans, versus to what extend environments should be adapted to both accommodate humans and robots. When building a new elderly care building, you could place RFID tags everywhere and adapt that building such that care robots can easily do their job. Changing existing environments can be very costly however, and is in many cases an infeasible investment. An interesting example of changing an environment for the robot is for example a recent experiment by Volvo. Volvo glued small magnets on a road every 20cm, which their car used to accurately position itself on the road. In contrast to existing, visually based techniques, this technique works well under all weather conditions when the car simply needs to follow the road. The Swedish government has expressed serious interest in financing larger scale tests in the coming years. Nowadays, the car industry aims at designing cars that can drive an increasing amount of kilometres autonomously, while very well understanding the incredibly challenging task of making a car fully autonomous in our current road system. All in all, it can be concluded that the solution will likely be somewhere in the middle; in part our environments will be adapted to robots, and in part robots need to adapt to deal with our human focussed environments.

Chapter 3

Proposed hybrid metric-topological mapping system for large scale environments

In this chapter, a novel mapping system architecture will be introduced, which we call Large Environments Metric TOpological Mapping system (LEMTOMap). It is a hybrid combination of a metric and topological mapping system. The system aims to be lightweight for both mapping and navigation at larger scale environments (like a complete faculty building or even a full university campus) and aims to be suitable for long term robot autonomy.

In this chapter, first a basic overview of the LEMTOMap architecture is given. Next, in preparation of discussing LEMTOMap into more detail, an explanation of some frequently used terms will be given. In the subsequent sections, the topological mapping, topological navigation, and Rolling Window GMapping (RW-GMapping) subsystems will be discussed into more detail[1]. Next, the integration with a semantic mapping system is discussed. Finally, the main conclusions will be summarized.

3-1 Overview of the LEMTOMap architecture

The main structure of LEMTOMap and the data flows are shown in Figure 3-1. LEMTOMap consists of two main subsystems, a hybrid mapping subsystem and a hybrid navigation subsystem. Each of theses hybrid subsystems consist of a metric and a topological part. The mapping subsystem takes care of both localization and mapping. The navigation subsystem takes care of metric and topological navigation. The subsystems are called hybrid, as they combine metric and topological environment representations.

[1]Metric navigation is not discussed into more detail, as this is discussed clear enough in the 'Overview of the LEMTOMap architecture' Section

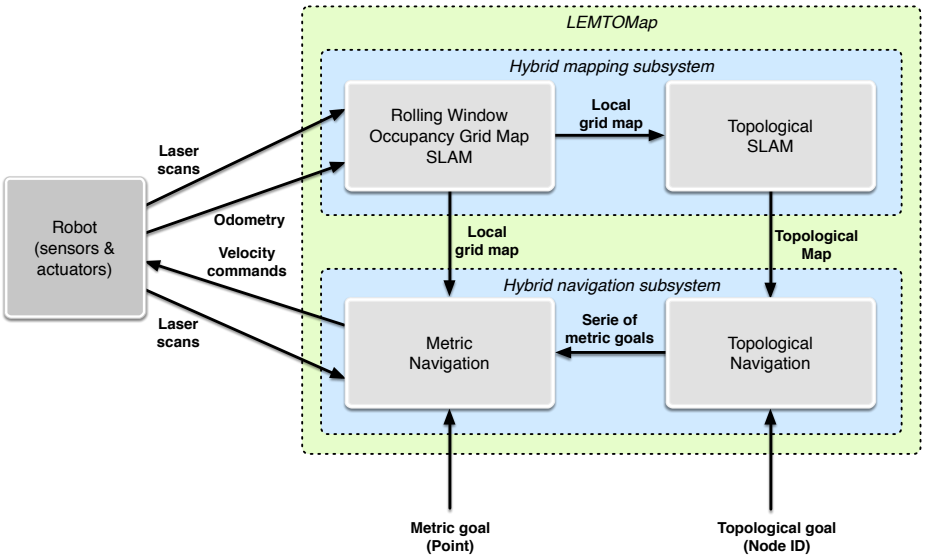


Figure 3-1: The main data components and data flows of LEMTOMap

The hybrid mapping subsystem is a combination of a rolling window occupancy grid map based SLAM algorithm (based on GMapping) and a topological SLAM algorithm on top of that. The rolling window grid map SLAM is used to generate a grid map in a limited region around the robot (e.g. a window of 20 by 20 meter), that moves along with the robot, such that the robot is always within the fixed-size window. On top of that a topological map is added, which is responsible for storing the environment on the global scale. The system is *robot-centric*, which means that the robot should be regarded the origin of its environment. Positions are defined relative to the robot, with the robot itself as the main coordinate frame[2]. Figure 3-2 shows an example of how such a map can look. The triangle in the lower corridor marks the robots pose estimate. The nodes are placed about every meter. There are many extra edges that are created, to minimize navigation path lengths, which will be further explained in Section 3-3. For this example the topological graph does not show metrical inconsistencies. However, in practice this will happen in larger scale mappings, as will be explained and shown in more detail later in this chapter (see e.g. Figure 3-5, and 3-8).

The key idea behind the hybrid approach of LEMTOMap, is that for operating in and interaction with your local environment, you need a somewhat detailed map of that environment. This is provided by our local grid map. However, on a global scale, details and *metric consistency* do not matter, as the main goal of your global map is navigation (for more explanation on metric consistency, see the 'Overview of terms' in Section 3-2). So for the global scale, our topological map suffices. As was explained in Chapter 2, for global path planning, topological path planning is less computationally complex and does not require any information on node locations, hence it does not require any metric consistency. This is similar to how humans deal with navigation. By lacking a large, global grid map, memory usage of the overall system can be significantly reduced (grid maps are in fact like bitmap image, in which each pixel, i.e. cell, needs to be stored). Also, the topological map may still have the risk of failing to

[2] *World-centric* SLAM on the other hand defines some origin point that is fixed to the world, and expresses the robots pose in terms of that world-fixed coordinate frame

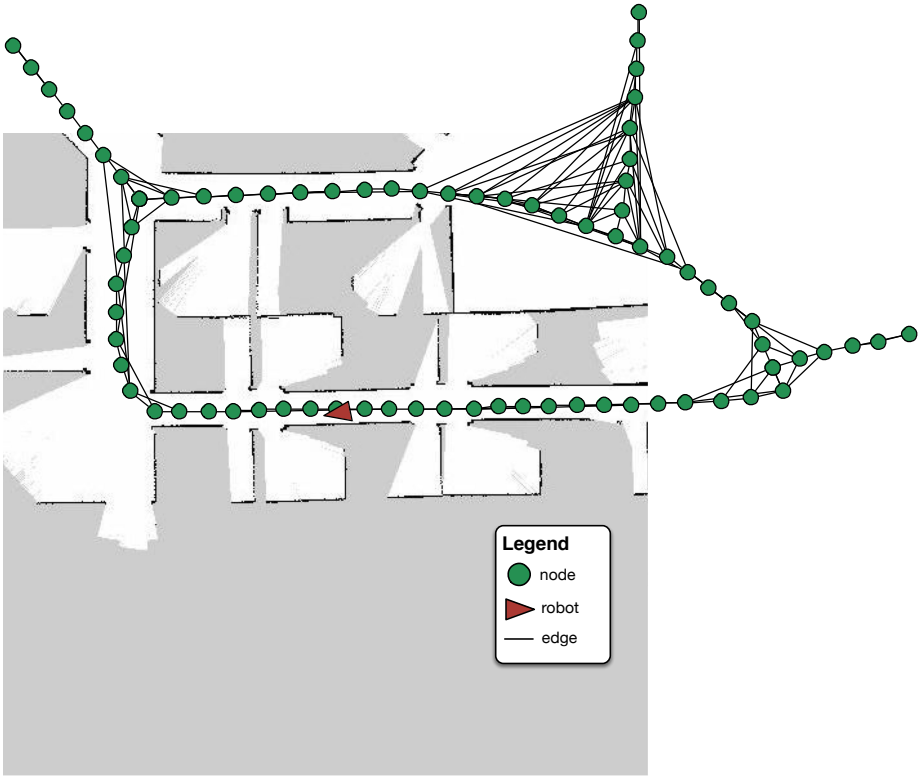


Figure 3-2: Example of how a map can look that is created using LEMTOMap. Everything shown is the robots internal representation of its environment, i.e. what the robot beliefs its environment is like. The map in the background is the rolling window grid map.

close loops, but consequences for navigation are not as disastrous as for a grid map SLAM algorithm (see Figure 2-3). Overall, LEMTOMap requires less memory, supports lightweight navigation on large scale, and is much more robust against the inevitable existence of metric inconsistencies.

General interplay of the subsystems In Figure 3-1, the robot supplies laser scans and odometry data to the grid mapping SLAM system. Based on the generated local grid map, the topological SLAM subsystem builds a topological map on top of that. When a topological goal is received by the topological navigation subsystem, it uses the topological map to find the shortest topological path. This topological path will then be passed as a series of metric goals to the metric navigation subsystem. It uses the local grid map and laser scans (to avoid collision with temporary and/or moving objects) to generate velocity commands for the robot. The metric navigation subsystem can also directly receive metric goals, if desired by the user. All four subsystems run asynchronously.

Rolling Window Occupancy Grid Map SLAM The local grid map serves several purposes. First of all, a grid map SLAM algorithm such as GMapping will significantly reduce localization errors when compared to a SLAM algorithm that completely relies on odometry

(especially errors introduced by rotational odometry errors). As a result of smaller localization errors, mapping errors will be reduced, resulting in less *metric inconsistency* between the map and the real world. Although metric consistency at global scale is not required for LEMTOMap, limiting metric inconsistencies is still useful (similar to how it is useful to humans to have a rough idea of where you more or less are in a metric sense). For example, the more limited metric inconsistencies are, the smaller the search space for loop closures can be made. The local grid map will help determine *direct navigability* between nodes of the topological map for edge creation (see Section 3-3). The local grid map will also be used by the metric navigation subsystem for local path planning. Overall, the local grid map gives the robot an accurate way to store how the local environment looks and to use that for local behaviour generation and for further processing to abstract features like local geometry, room segmentation, etc. The local grid map can be assumed to provide local metric consistency. As the robot moves along, parts that fall out of the rolling window grid map will be forgotten. All other relevant information should be stored in the nodes of the topological map, which serves as the storage system on the global scale.

Topological SLAM The topological map will be used to store all the relevant information about the environment for different kinds of long term usage, such as navigation. The topological graph will be used to perform global, topological path planning. Additionally, it can be used to enable advanced forms of loop closing. It is even possible to have multiple hypotheses simultaneously (one with, and one without loop closure). The topological map is aimed at performing under *metric inconsistencies*. Nodes of the topological graph do not have a location in some world level coordinate frame, instead, each node has its own *frame* and edges define the *transform* (de facto constraints) between those frames (see 'Over of terms', Section 3-2). Locally, the topological graph constraints defined by the edges will be solved such that it aligns to the local grid map, making it locally metrically consistent with the local grid map and the real world. Globally, solving the constraints is not necessary, as for global navigation it is not necessary to know the locations of nodes; the costs of edges suffice.

Topological & metric navigation In addition to the hybrid mapping system, a hybrid navigation subsystem forms the second main part of LEMTOMap. Metric navigation is well defined these days, and can be provided by any metric navigation algorithm that performs navigation based on a grid map. For LEMTOMap, a Dijkstra's Algorithm based metric navigation system is used, based on [Fox et al., 1997]. The topological navigation subsystem uses the topological graph to find the shortest path (using Dijkstra's Algorithm) of nodes to an arbitrary destination node. The nodes of this path are then passed successively to the metric navigation subsystem, which takes care of navigation at the local, metric level.

3-2 Overview of terms

Please note that several of these terms and definitions are defined by the author of this thesis and may not be found in other literature at all, or may be defined differently in other literature.

Frames, transforms and poses In robotics, *local coordinate frames* are often used to give objects or instances (like nodes) their own reference coordinate frame. In a multibody object like a robot arm for example, the arm can be regarded as a set of bodies connected by joints. Each body has its own coordinate frame, while transforms define how to transform a point defined in one frame to a point defined in another frame. A point in 3D space can be defined by an x, y, and z coordinate (as expressed in an arbitrary frame), while a transform between frames is defined by an x, y, z translation and a $roll, pitch, yaw$ rotation. This is in fact the relative pose of the target frame compared to the source frame.

Points are defined as vectors, while a transform can be applied to such a point by multiplying by a homogeneous transformation matrix. The homogeneous transformation matrix is a 4×4 matrix for 3D space and 3×3 for 2D space. Equation 3-1 to 3-5 shows how to transform a 2D point $\mathbf{p}^i$ defined in frame ψ_i to the same point $\mathbf{q}^j$ defined in frame ψ_j. $\mathbf{H}_i^j$ is the transformation matrix from frame ψ_i to ψ_j, which depends on the relative pose between the frames. The relative pose of frame ψ_j compared to ψ_i is defined by the pose vector $\left[x_i^j, y_i^j, \theta_i^j\right]^T$. The example situation is also shown Figure 3-3.

For LEMTOMap, transforms are used to define relative locations between the robot, nodes and the local grid map. As mentioned before, transforms are actually the same as relative poses. The terms (relative) pose and transform are hence used interchangeably in this thesis.

$$\bar{\mathbf{q}}^j = \mathbf{H}_i^j \bar{\mathbf{p}}^i \tag{3-1}$$

with (2D):

$$\mathbf{q}^j = [q_x, q_y]^T \tag{3-2}$$

$$\mathbf{p}^i = [p_x, p_y]^T \tag{3-3}$$

$$\bar{\mathbf{q}}^j = [q_x, q_y, 1]^T \tag{3-4}$$

$$\bar{\mathbf{p}}^i = [q_x, q_y, 1]^T \tag{3-5}$$

Topological localization Topological localization can be considered at two levels; the topological level and the metric level. At the topological level it is about determining what node of the topological graph the robot is currently at. The robot is always *at* some node. This node is what we call the *associated node*, as the robot is currently associated with that node. On the metric level, the robot always has some metric, relative position compared to this associated node. That is what is the metric part of topological localization. Topological localization as a whole is defined here to be the combination of the current associated node and the relative position of the robot to that node. Topological localization is illustrated in Figure 3-4.

Associated node The associated node is defined to be the node where the robot is currently at in the topological graph. At any point in time, the robot has always exactly one associated node. The associated node is illustrated in Figure 3-4.

Local/global metric consistency By *metric consistency* it is meant that a robots internal representation of an environment is metrically in accordance to the real world. In case of an

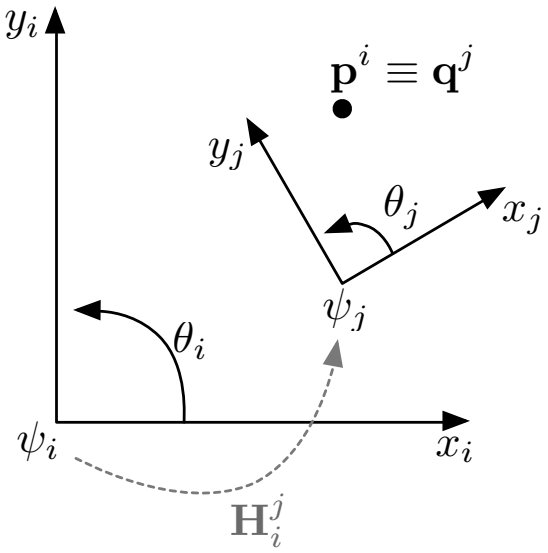


Figure 3-3: Using transforms, one can find the location of the point p^i (expressed in frame ψ_i) as a location q^j expressed in local coordinate frame ψ_j. The transition from frame ψ_i to ψ_j is defined by a homogeneous transformation matrix $\mathbf{H}_i^j$, which is based on the relative pose between the frames $\left[x_i^j, y_i^j, \theta_i^j\right]^T$

occupancy grid map, this would mean that the 'floor plan' built by the robot would map one to one to the true, exact floor plan. When such maps align well on a local scale, one could speak of a map that is *locally metric consistent*. Possibly, errors accumulate further from the robot, resulting in a map that is just locally metric consistent, and not *globally metric consistent*. See Figure 3-5 for an example. The transition from consistent to inconsistent is loosely defined in this thesis. A map is considered metrically consistent if it is consistent enough to be usable for navigation. We speak of local metric consistency as long as the map is locally usable for navigation. A requirement for being metrically consistent is that the map does not contain any map artefacts caused by failed loop closures that are so severe that they limit usability for navigation (such as in Figure 2-3).

Local graph solution The topological graph is allowed to become globally metrically inconsistent. At local scale, it should be metrically consistent however. To achieve this, the constraints that are defined by the edges of the graph need to be solved in such a way that local metric consistency is achieved. This is referred to as the *local graph solution* (or *local topology solution*). More explanation on this approach is given in Section 3-3.

Local grid map transform The *local grid map transform* defines the transform between the frame in which the metric map (i.e. the grid map) is defined and the frame of the current associated node. On node creation, nodes have some location relative to their metric environment (e.g. in the middle of the corridor), the local grid map transform makes sure that the metric map is properly aligned with the local topological graph solution. More explanation

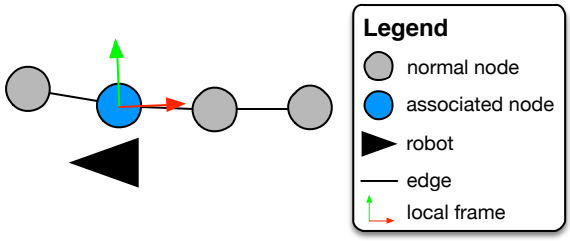


Figure 3-4: The topological localization at the topological level is represented by the associated node; the robot is currently topologically localized at the blue node. Topological localization at the metric level is the pose of the robot expressed in the local coordinate frame of this associated node.

on this approach is given in Section 3-3.

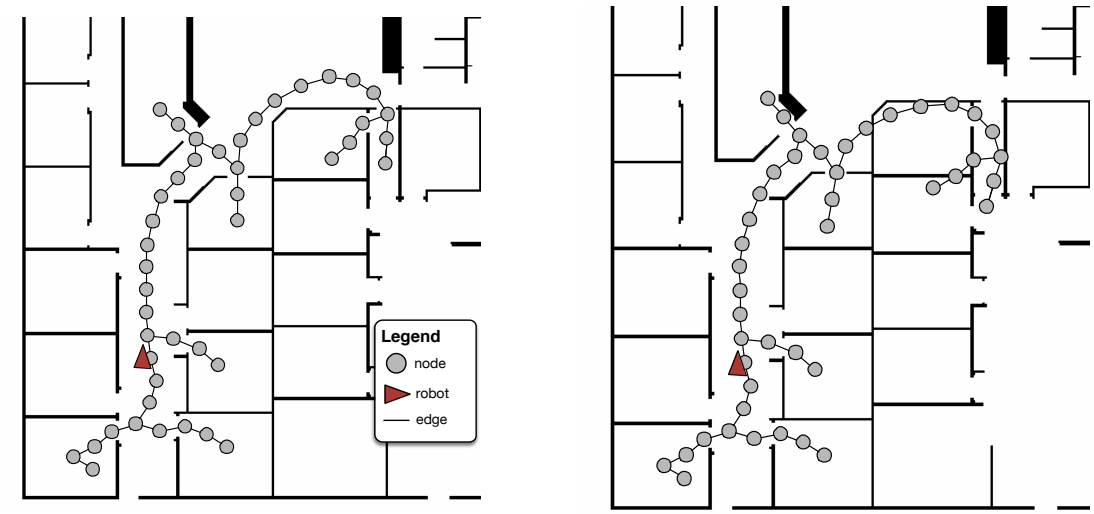
Navigability and direct navigability *Navigability* between two points means that it is possible for a robot to navigate between those points. We use the term *direct navigability* to denote that it is possible for a robot to navigate in a straight line between two points.

Rolling window A *rolling window* (or sliding window) map is a map that moves along with the robot. It has a certain fixed size and it shifts such that the robot is always inside the rolling window map. This is for example also used for the local costmap used by the metric navigation of the ROS `move_base` package, as discussed in Section 2-4-1. It can be assumed that data that slides out of the scope of the window will be deleted, while the new area sliding into the window will be initialized as unknown. More explanation on this approach is given in Section 3-4.

Topological loop closing If the topological mapping system recognizes that some place belongs to a certain node from the topological graph, a topological loop closure can be defined. This closure is defined by adding a constraint (edge) between the associated node and the recognized node. This is very similar to how constraints are added in pose graph SLAM (see Appendix Section A-3).

Topological distance By *topological distance*, the shortest topological distance between two nodes is meant.

Node neighbours *Node neighbours* are all nodes that are directly connected to another node, i.e. connected through one edge. These nodes are sometimes also named adjacent nodes. Similarly, adjacent edges are all edges directly connected to a node.



a: Global metric consistency

b: Local metric consistency only

Figure 3-5: The map in the background is the ground truth map. *(a)* is globally metrically consistent (and thus locally as well). *(b)* shows local metric consistency, but not globally. For this example, a topological map was used; obviously the same could be done for a grid map.

3-3 Topological SLAM

3-3-1 Map definition

The topological map is defined by a graph g, which is a tuple $g = (\mathcal{N}, \mathcal{E})$ that consists of a set of nodes $\mathcal{N} = \{n_1, ..., n_m\}$ and a set of edges $\mathcal{E} = \{e_1, ..., e_p\}$.

A node n_i is defined as the tuple:

$n_i = (N_{\text{id}}, a_{\text{id}}, l) \in \mathcal{N}_{\text{id}} \times \mathcal{A}_{\text{id}} \times \mathcal{L}$

The node ID $N_{\text{id}} \in \mathbb{N}$ is a unique, fixed ID of the node, i.e. it cannot be changed after node creation. If the node is deleted at some point, the ID is not allowed to be re-assigned at any time in the future during a new node creation.

The area ID $a_{\text{id}} \in \mathbb{N}$ is an ID that can be assigned to multiple nodes. Nodes with the same area ID belong to the same area, most commonly a room. Door detection can be used to segment space into areas (rooms) and thus to facilitate setting proper area IDs for the nodes. Area IDs can be updated at any time.

The localization data l is used to help the robot localize itself relatively to the node (topological localization at the metric level). It is needed for finding the local grid map transform and for performing topological loop closing (see Section 3-2). The type of localization data is not yet strictly defined for LEMTOMap. As an example, small local grid maps or sensor measurements can be used. Laser scan data can be used to perform scan-matching to find relative positions. Camera images can be used to recognize objects and/or appearance of previously visited places. Abstracted semantic properties can be used to aid or even fully perform topological loop closing and/or finding the local grid map transform. If the robot thinks that it is in a corridor, it can reduce the search space of nodes that are topological loop closing candidates to nodes that are (likely[3]) in a corridor. A downside of storing such localization data in the nodes is that it can significantly increase the required memory to store the topological graph g. Possibly, the (small) local grid maps or sensor measurements could be down-sampled to limit this effect. For topological localization and finding the local grid map transform this is likely no problem, as accuracy is not very important then.

Nodes can easily be expanded by other kinds of information that can be stored in them. Most importantly, abstracted semantic information can be stored in the node, such as the type of room, room geometry, objects detected, etc.

An edge e_i is defined as the tuple:

$e_i = (E_{\text{id}}, S_{\text{nid}}, T_{\text{nid}}, \mathbf{r}, \Sigma_{\mathbf{r}}, E_{\text{type}}) \in \mathcal{E}_{\text{id}} \times \mathcal{N}_{\text{id}} \times \mathcal{N}_{\text{id}} \times SE(2) \times \mathbb{R}^{3\times3} \times \mathcal{E}_{\text{type}}$

The edge ID $E_{\text{id}} \in \mathbb{N}$ is a unique, fixed ID of the edge, i.e. it cannot be changed after edge creation. If the edge is deleted at some point, the ID is not allowed to be re-assigned at any time in the future during a new edge creation.

The source node ID $S_{\text{nid}} \in \mathcal{N}_{\text{id}}$ and target node ID $T_{\text{nid}} \in \mathcal{N}_{\text{id}}$ define which nodes are connected by the edge

[3]See [Pronobis and Jensfelt, 2012] about the probabilistic semantic mapping approach by Pronobis. This is also described in Appendix B.

The vector $\mathbf{r} \in SE(2)$ defines the relative pose between the source and target node that are connected by the edge. As nodes do not define their own position in the world, these relative poses are the only way to find node locations relative to the robot. As the topological map should be interpreted in a robot-centric way, this means that the topological localization on the metric level will give the associated node some pose compare to the robot (consisting of an x,y, and θ coordinate). All adjacent edges will define the relative poses between the associated node and its neighbour nodes. This way, one can find the relative pose between any node and the robot. It is important to understand that this is only useful for nodes that are nearby (that is, within the range of the local graph solution), as the graph is globally not metric consistent. In other words, relative poses will give a decent estimate of node locations on a local scale. The relative pose defines a *constraint* between two nodes. The term constraint is especially used often in the context of pose graphs (see Section A-3) and topological loop closing. The relative pose implicitly defines the cost of the edge ($cost = \sqrt{r_x^2 + r_y^2}$).

The relative pose covariance $\Sigma_{\mathbf{r}} \in \mathbb{R}^{3\times3}$ is a covariance matrix that defines the uncertainty of the relative poses. Relative poses defined in loop closing edges are generally much less certain than relative poses based on odometry between the two nodes.

The E_{type} property is used to define the kind of edge. Three distinct types are used: odometry, near neighbour, and loop closing edges. When a new node is created, an odometry based edge is created between the new node and the previous associated node. Near neighbour edges are created between a new node and any node that is directly navigable from that node and is close enough in the shortest topological path length sense. Loop closing edges are used to close topological loops. Edge creation will be discussed in more detail in Section 3-3-2.

3-3-2 Subsystem operation

The main steps of how the topological mapping subsystem operates are summarized by the flow chart in Figure 3-6. First the general flow of the algorithm is treated, next some specific elements are discussed in more detail.

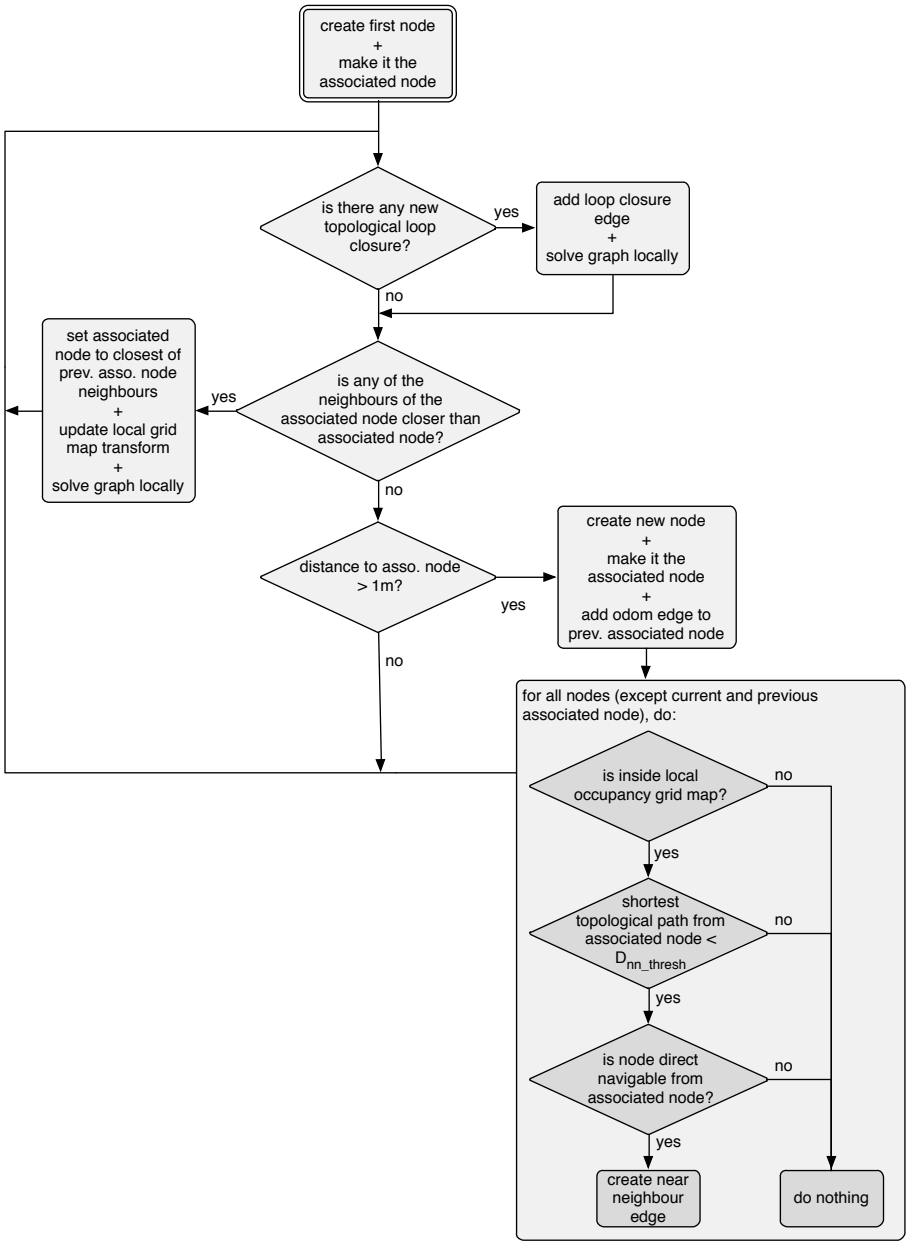


Figure 3-6: Flow chart of the topological mapping process.

General algorithm flow At the start of the algorithm, an initial node is created, which obviously automatically needs to become the associated node. After that, the main loop starts running.

At the start of the loop, an explicit topological loop closure check is performed to detect topological loop closures. If a closure is detected, a loop closure edge is added and the graph constraints need to be solved locally.

Next, based on the movement of the robot it is checked if the associated node is still the right one. If any of the neighbouring nodes of the associated node is closer to the robot, it should be set to be the new associated node. If the associated node is updated to be one of the neighbours, the local grid map transform should be updated and the graph should be solved locally.

If the associated node is not updated, creating a new node will be considered. If the associated node is further than 1 meter away, a new node will be created. This new node becomes the associated node and is connected to the previous associated node using an odometry edge. Additionally, the algorithm tries to create as many valid near neighbour edges as possible.

Explicit topological loop closing Explicit topological loop closing makes sure that loops in the topological graph are closed when possible. By topologically closing loops, topological paths generated for navigation can become significantly shorter, as loop closing edges can provide short cuts between the start and end node of the navigation task.

To close topological loops, an explicit check for topological loop closures is performed . Please note that small loops can already be *closed implicitly*[4] by the regular near neighbour edge creation process which is triggered on every new node creation. This explicit check for loop closures remains unspecified in this thesis, however many possibilities exist. The most brute-force approach would be to check for correspondence between the currently observed part of the environment and the localization data of all nodes. The search space of nodes could be limited in several ways, e.g. by only comparing to nodes that likely have the same room type as the associated node, or by limiting the search space to nodes that are within a certain topological distance. If a topological loop is detected (see Figure 3-7a,b), a loop closure edge is added, see the dotted line in Figure 3-7b.

In contrast to grid map SLAM, a failed loop closure cannot cause a navigation task to fail (Figure 2-3), although it can result in paths that are much longer than needed. Another advantage of this topological loop closing approach is that the system could potentially still close loops (that it initially failed to close) at any later stage in a straightforward way.

Local graph solving Figure 3-7c shows how a graph can be solved locally. In the case of the example, it is solved locally after a topological loop closure was performed in Figure 3-7b. Solving the graph constraints should be performed for all nodes up to some fixed threshold topological distance from the associated node. If the system is over-defined, the solving can be regarded a least squares problem which can be solved in a similar way as is done in pose

[4]Implicit loop closing means that the loop is closed by the algorithm itself, without requiring a special loop closure procedure. Explicit loop closing involves a special procedure that explicitly looks for possible loop closures first, and if a closure is found it updates the map to reflect the loop closure.

graph SLAM (e.g. using Gauss-Newton or Levenberg-Marquardt algorithm). As the set of nodes is limited, the solution of this problem can be found relatively fast compared to full scale pose-graph problems. By solving the graph locally, the robot is always surrounded by nodes that can help it navigate reliably on a local scale.

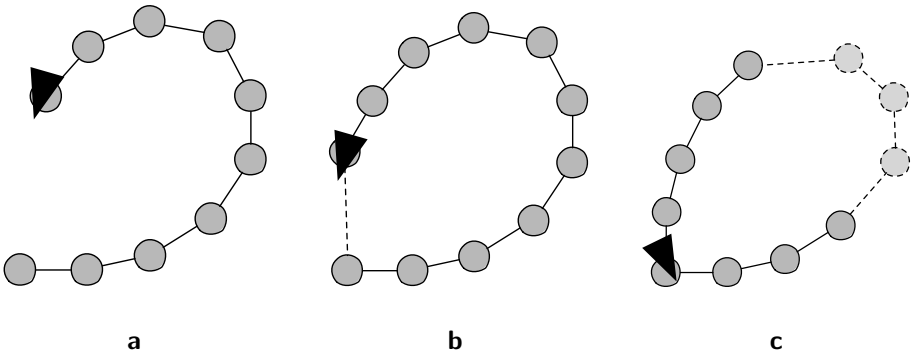
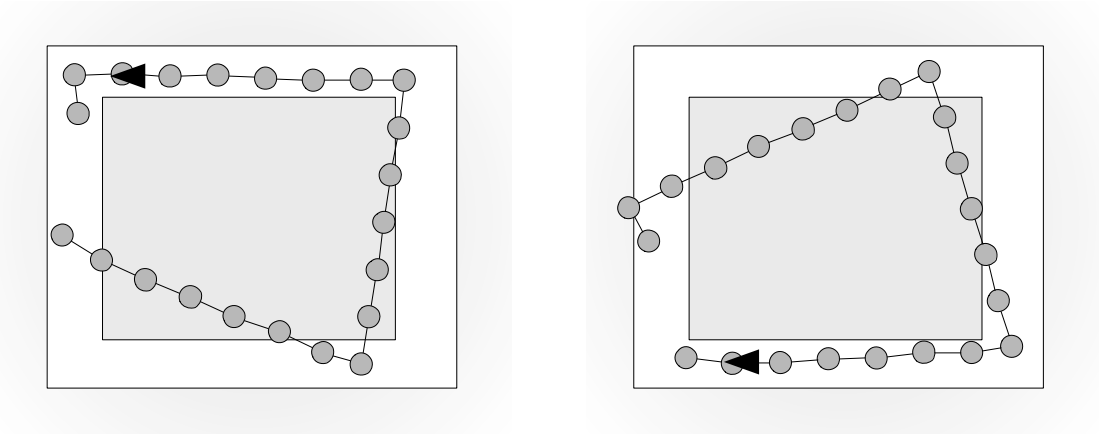


Figure 3-7: *(a)* The robot (black triangle) has travelled in a loop and nears detecting this loop. *(b)* The robot detects the loop. De dotted line shows the new loop closure edge which defines a constraint. As errors have accumulated, the nodes connected by the loop closure in (b) are visualized too far from each other. *(c)* The constraints (relative poses) defined by the edges are solved locally. The topological graph is now accurate locally, while further nodes and edges (dotted) do not matter and are not even considered.

Updating the local grid map transform If the associated node changes to some other node that already existed, the graph needs to be solved locally, and the local grid map transform needs to be updated. By updating it, the relative pose of the associated node is aligned with the local grid map. Consequently, in addition to having a locally solved graph, that graph is also aligned properly with the local grid map. The combination of local graph solving and updating the local grid map transform results in a local topology that can be used reliably for navigation purposes. By updating the local grid map transform, one also prevents that the topological map and local grid map start to diverge over time. Figure 3-8 shows an example of updating the local grid map transform.



a b

Figure 3-8: *(a)* While the robot (black triangle) has travelled three quarters of a circular corridor, errors in the topological graph have accumulated, resulting in a topological map that is not fully aligned with the ground truth map (the corridor and walls in the background represent the ground truth). *(b)* When the robot travels back, the local grid map transform is updated to make sure that nodes align properly with the environment locally. This way, the topological graph is stays usable locally. N.B. For the sake of simplicity, only odometry edges are shown.

Creating nodes & edges When a new node is created, it is automatically made the associated node and an odometry edge is created between the new node and the previous associated node. Thanks to the odometry edge, a new node is always connected to the graph, i.e. there is no risk of creating orphan nodes or a map that actually consists of more than one mutually unconnected graphs. After creating a node, the system will try to make as many near neighbour edges (see E_{type} definition in Section 3-3-1) as possible. As illustrated in Figure 3-9, the near neighbour edges serve to shorten path lengths for topological navigation. Additionally, depending on the D_{nn_thresh} threshold, they can implicitly close small loops. All nodes except the current and previous associated node are checked if they are inside the local occupancy grid map, if the topological distance is less than a certain threshold, and if they are directly navigable from the new node. If all these conditions are satisfied, a near neighbour edge is added. Near neighbour edges should only be created if there is direct navigability between the nodes. To check for direct navigability, a costmap based on the occupancy grid map is used. Hence, only nodes within the local grid map can be checked for direct navigability. As the topological graph is only accurate on a local scale, further nodes should be disregarded (see Figure 3-10). In general, setting the D_{nn_thresh} threshold equal to the distance up to which the topological graph was solved locally should work fine.

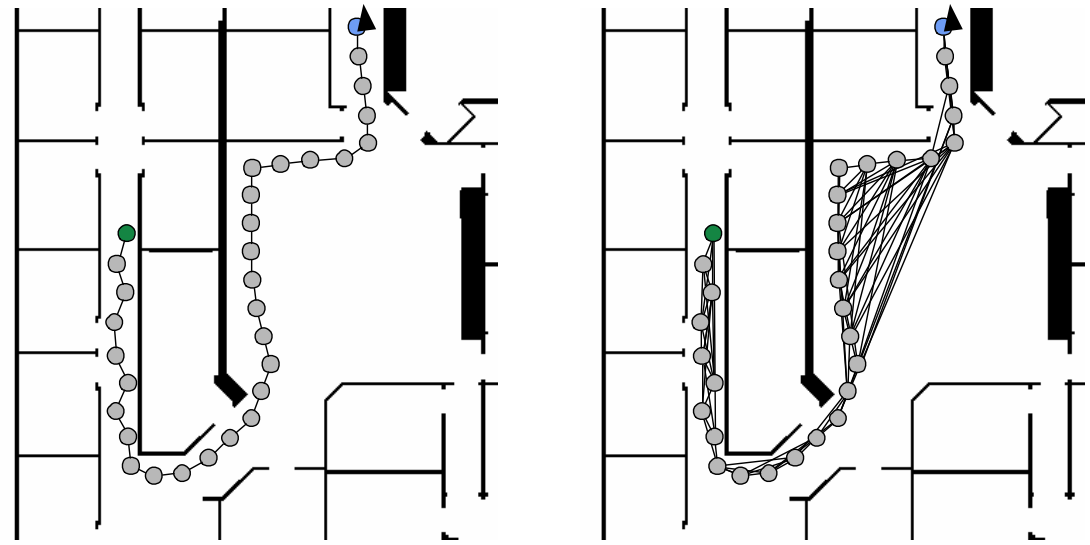


Figure 3-9: The map shown in the background is the ground truth map. If the robot (black triangle near upper right corner) needs to find to shortest topological path from the associated node (blue) to the green node in *(a)*, the path will be much longer than necessary. The additional near neighbour edges in *b* will allow the robot to find a much shorter topological path. Despite looking visually more cluttered, the topological graph in *(b)* is much more useful for topological navigation.

The associated node A key feature of the system is the associated node. The associated node enables a few important features. First of all, the associated node can be used to always create at least one edge when creating a new node. As the robot knows what node it came from, it can always know for sure that an edge between that node and the new node can be created. Without this odometry based edge, the method used to create near neighbour edges could otherwise fail to create any edge, resulting in an orphan node. Imagine for example a robot travelling close along a wall that takes a sharp turn as soon as it encounters a doorway. The new node that will be created at the other side of the wall can turn out to be not directly navigable in the original sense we defined it (the robot should be able to travel in a straight line between the nodes without running into an obstacle). We do know however, that the robot just travelled to this new node from its previous associated node, and thus we know that it can easily travel between the nodes. In case of topological navigation, the metric path planner takes care of obstacle avoidance and will plan a path between the nodes trough the doorway. Please note that the choice to create a new node as soon as the robot is more that one meter away from the associated node is also important to make sure that the topological mapping system works properly. If the threshold would be two meters for example, the robot could travel closely along a wall, make a sharp turn through a doorway and travel close to the wall again, without any nodes being created on the other side of the wall. For the system used in this theses, it is safe to assume that the center of the robot will always be more than half a meter from the center of a wall, resulting in a reliably working system.
Secondly, the associated node is used as the starting point for topological navigation. As becomes clear from Figure 3-10, one cannot simply take the node closest to the robot.

Lastly, the associated node enables the robot to perform in multi-level environments (within the limitations discussed previously) and to use a topological graph that would visualize cluttered (i.e. with overlapping parts) because of accumulated errors.

Only new nodes or neighbours of the current associated node can become the new associated node. The latter is important. Imagine for example what would happen if one would simply check for the closest node; in that case a node at the other side of a wall could invalidly become the associated node as well.

Multi-level environments The topological map is able to handle multi-level environments by the way it is defined. Consider Figure 3-10; there could be two reasons why no topological loop should be closed. Firstly, because the robot has travelled up a level through a staircase and thus is at a different place than the nodes that it seems to encounter when visualizing the topological graph. Secondly, it could be because errors have accumulated and therefore parts of the graph overlap on visualization, while they would not overlap when projected on the ground truth world. The latter is obviously not likely for a small loop as in this example figure, but is increasingly likely to occur when the mapped environment grows. Currently the near neighbour maximum distance threshold D_{nn_thresh} is used to mitigate the risk of creating near neighbour edges that are actually impossible to directly travel for the robot. Clearly, this can easily fail for multi-story buildings, as travelling up one level through a stair case can easily result in an edge being created between a node created when exiting the stair case and a node one level lower. Decreasing D_{nn_thresh} can decrease or even remove this risk, but will result in less near neighbour edges, which will result in longer paths when performing topological navigation (see Figure 3-9). To overcome these problems, a more advanced method to govern near neighbour edge creation is needed. The robot could for example try to track at which level it is and only create near neighbour edges between nodes that are at the same level.

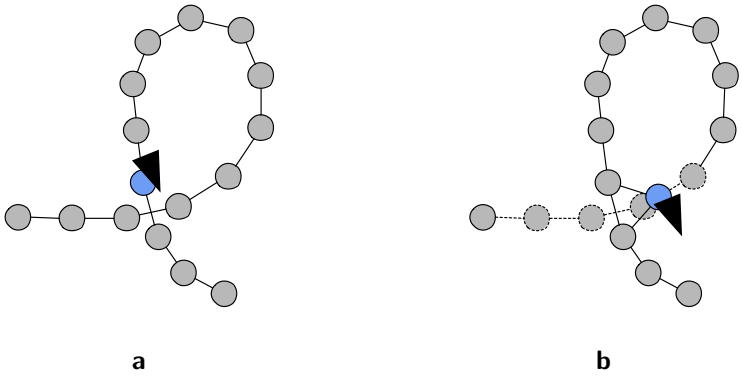


Figure 3-10: *(a)* Based on the topology, the robot seems to have travelled a circular path. The nodes that it encounters are further away than specified by D_{nn_thresh} (see Figure 3-6) and no topological loop closure was detected. Therefore, no edge is created that connects to the encountered nodes. Instead, a new node will be created when it continues, as shown in *(b)*. The blue node is the associated node. N.B. For the sake of simplicity, most near neighbour edges are not shown.

Area segmentation Lastly, area segmentation can be used to group nodes into areas. For an indoor environment, such areas are generally rooms. If the robot detects that it has passed a new doorway, a new area ID should be assigned to all nodes created after that. As soon as a loop is closed, the area IDs of the two connecting areas can be merged by assigning both the same area ID. With such an area segmentation of the topology, area based topological navigation can be performed. From the normal topological graph, which is a topology aimed at navigation, a more general 'area topology' can be extracted by grouping all nodes belonging to the same area in one node, such as is shown in respectively the right and middle image of Figure A-13 (Appendix A, page 92).

3-4 Rolling Window GMapping

Rolling Window GMapping (RW-GMapping) is an adaptation of the original GMapping algorithm as discussed in Section 2-2-1. The Rolling Rindow GMapping algorithm is described by the pseudocode of Algorithm E-4 (Appendix E).

RW-GMapping initializes all particles with a map size equal to a certain, user specified window size (e.g. 20 x 20 m). As soon as the grid map does not fit inside this window any more, the window is shifted such that the robot is in the middle again. Everything that was recorded that is outside of this window, is forgotten. As a result, RW-GMapping cannot create failed loop closures within is window, simply because its window is too small to generate a loop large enough to risk failing loop closure. Also, RW-GMapping only needs to store a fixed size map per particle. Which limits the amount of memory needed to maintain all the particles.

The choice has been made to only shift the window of the map when laser scans data is registered outside of the current window. This means that the window does not actually slide along with the robot all the time, but only gets updated when the robot starts mapping something outside of its window. If the window size is much larger than the range of the laser scanner, this means that the robot can travel far from the center of the window before an update is triggered which places the window with the robot in the center again. This is done for two reasons. Firstly, less frequent moving of the window causes less computational load. Secondly, moving it more frequently does not provide any value to the robot. New rows and columns added in the direction that the robot is moving will always be initialized as unknown, hence those rows and columns do not provide any value to the robot. On the other hand, old rows and columns removed from where the robot was coming from did possibly contain valuable information. Therefore, frequent updating actually decreases the overall informational value of the local occupancy grid map, making the current resizing strategy a better choice.

The reader might notice that GMapping is originally designed to be used in a world-centric way, rather than robot-centric. All particle poses describe a path leading back to $[0, 0, 0]^T$ (in the coordinate frame used by GMapping), and as particle paths diverge over time (with loop-closing as a way to limit this divergence), the origins of sliding window particle maps will also diverge over time. Despite being defined in a world-centric way, results can be interpreted and visualized in a robot centric way easily. Simply pick the best particle, and for any other particle take the difference between this particles robot pose estimate and the one of the best particle. Apply this difference to the full path of previous poses of the particle and apply the difference to shift the particle's map center. If one does this for all particles, the historical paths and maps per particle would visualize properly as if the system is defined in a robot-centric way.

As RW-GMapping does not need to close very large loops (loops larger than the scope of the rolling window is impossible to close), it requires less particles than normal GMapping. Only a few particles (e.g. 3 to 10) should suffice for a normal size window (e.g. 20 by 20 m).

3-5 Topological Navigation

Topological navigation enables navigation to an arbitrary goal node. Dijkstra's Algorithm will be used to find the shortest topological path from the associated node to the goal node. The path found consists of a series of node IDs that are passed successively to the metric navigation subsystem. As soon as an intermediate goal node becomes the associated node, the next node can be passed as a metric goal to the metric navigation subsystem[5].

In addition to node based topological navigation, area based topological navigation is also possible. One simply sends the robot an area ID as a navigation goal. Next, the system calculates the shortest topological paths for all nodes being part of that area. The node that has the shortest route is the best to pick and can be sent as a goal to the normal topological navigation subsystem (assuming that the user only cares about the robot getting to that area, without minding where the robot should be in that area).

3-6 Future integration with semantic mapping

LEMTOMap can very well be integrated with a semantic mapping system in the future. Pronobis has announced that he is working on a ROS version of the work presented in [Pronobis and Jensfelt, 2012], which is supposed to be released as open source soon. The topological map of LEMTOMap could serve as the storage backbone; all semantic details about places can likely be stored in the nodes in a straightforward way. This way, the semantic mapping system can benefit from the large scale environment capabilities of LEMTOMap, and a continued integration between both can result in a system that can smartly operate autonomously for long periods of time in large scale environments. LEMTOMap will replace the metric and topological SLAM systems used by the original semantic mapping system. Additionally, it will replace navigation partially, although a semantic planner ('go to my room', 'get me a glass of water') is still needed to find (a series of) topological goals.

3-7 Summary

LEMTOMap focusses on local metric consistency, while getting rid of the reliance on the global metric consistency assumption which is inherent to traditional SLAM methods. The system is robust against global metric inconsistencies, while traditional SLAM methods such as GMapping are likely to generate maps with severe artefacts if it fails to close a loop because of global metric inconsistencies. On a longer term, a stronger reliance on using semantics instead of metric consistency can be used to close loops, in a similar way to how humans close loops (i.e. by recognizing the semantics at places again, instead of estimating your position based on past movement). As a result, LEMTOMap is better suitable for large scale environments and consequently provides better possibilities for long term robot autonomy.
The system also provides resource efficient robot navigation through topological navigation based on the topological graph. The use of many near neighbour edges should limit the

[5] Alternatively, one could also choose to pass the next goal as soon as the current goal node is directly navigable from the robot.

performance decrease in terms of total path length travelled. The topological graph also has the possibility to handle multi-level environments, such as a multi-story building.
As the topological map is much more compact than an occupancy grid map, and the occupancy grid map only covers a limited region, the overall system will theoretically require a lot less more to store the map (both in working memory as on disk).

The main features of LEMTOMap allowing it to work as described above are the use of the local grid map transform, the local graph solving approach, the RW-GMapping algorithm, the associated node, and the creation of many near neighbour edges based on the rolling window occupancy grid map.

Disadvantages are that the overall system contains less detail than a full scale occupancy grid map (although adding semantic information to the nodes will add a lot of other valuable information, potentially being of much more value than a traditional grid map). The topological map is harder to 'read' for humans, as global inconsistencies and/or multi-level situations can cause the graph to look cluttered. Also, large amounts of near neighbour edges can make it look cluttered too. Furthermore, the topological graph does not clearly show shapes of rooms and layouts of buildings, make the graph much less intuitive to humans to interpret than the visualization of a grid map.
Another disadvantage is the fact that the topology currently only covers places where the robot physically has been, while an map generated by e.g. GMapping covers everything that has been seen by the laser range scanner. In [Pronobis and Jensfelt, 2012], hypothetical nodes (which are called placeholders in the paper) are used to deal with (and reason about) seen but unexplored space. Such hypothetical nodes seem like they can be a valuable addition to LEMTOMap.

The main challenges in implementing and further developing LEMTOMap lie in the topological loop closing, for which no detailed method is described yet in this chapter. Also the integration with semantic mapping and the possibilities arising from this combination remains largely undefined. Furthermore, a better, more advanced way to deal with the multi-level potential of the topological map should be worked out. A better alternative to the current D_{nn_thresh} is desired. Lastly, the current topological map consists of many nodes and edges. For very large scale scenarios, it might become interesting to try to limit the nodes and edges, either by periodically pruning or by limiting from the moment of creation directly. Nodes could for example only be placed on places of high entropy (e.g. large changes in the shapes described by the laser scan data, or large changes in camera images), or on places with high semantic significance (e.g. corners of corridors, at doorways, at objects, etc.).

Chapter 4

Implemented System

In this chapter, an implementation is given of the LEMTOMap architecture introduced in the previous chapter. The implementation is a partial implementation of LEMTOMap, due to time constraints. Therefore, we will refer to the implementation as LEMTOMap-V0.5.

LEMTOMap-V0.5 implements the main principle of the LEMTOMap architecture, namely the hybrid metric-topological map in which the metric map is a rolling window grid map. Using LEMTOMap-V0.5, the basic mapping, localization and navigation performance of LEMTOMap can be verified. Most notably, local graph solving is not implemented and maintaining the local grid map transform is partially implemented. The main consequence of lacking local graph solving is that the system is unable to close topological loops that bridge a discrepancy as shown in Figure 3-7. The partial implementation of maintaining the local grid map transform, has the consequence that knowledge of the actual robot pose is required to find and apply the right local grid map transform, which is needed to reliably map, localize and navigate at larger scale / longer periods of time.

LEMTOMap-V0.5 was implemented in C++, and is developed for usage with ROS in combination with the TurtleBot 2 robot[1], which is a small, driving robot of about half a meter high. Although primarily aimed at this robot, LEMTOMap-V0.5 should also be easy to use on any other laser range scanner equipped robot. The source code for LEMTOMap-V0.5 is available as open source at https://github.com/koenlek/ros_lemtomap.

The main differences between LEMTOMap-V0.5 and LEMTOMap will be discussed in the next section. Then, a short overview of the overall LEMTOMap-V0.5 system is given. In the section after that, the main components of LEMTOMap-V0.5 (topological SLAM, topological navigation, and RW-GMapping) will be discussed. Finally, the main steps towards a full implementation of LEMTOMap are discussed.

[1]More details on the TurtleBot are given in the Experiments chapter (Chapter 5).

4-1 Differences between LEMTOMap-V0.5 and LEMTOMap

The main differences are the following:

- Local graph solving is not implemented.
- Maintaining the local grid map transform is not fully implemented yet. A temporary solution is implemented that can provide the right transform in simulation when provided with the ground truth robot pose.
- Edges do not define relative poses between nodes, instead nodes carry their own pose relative to a global coordinate system.
- Door detection and area segmentation are not yet implemented. Consequently, area based navigation is also not implemented.
- Topological loop closing is not yet implemented.

The main reason behind these difference is a lack of sufficient time to implement them, while some are in part caused by difficulties faced when one would implement such a feature in ROS. The differences will be discussed in more detail in the sections about the corresponding component of LEMTOMap-V0.5.

The main consequences are that the current implementation is only applicable in real life to a limited extend. Also, the system cannot solve the graph constraints locally, making it unable to close topological loops that bridge a discrepancy as shown in Figure 3-7. Overall, there is no support yet for explicit topological loop closing. Lastly, area based navigation is not yet possible.

4-2 Overview of LEMTOMap-V0.5

LEMTOMap-V0.5 consists of four parts, topological SLAM, metric SLAM (RW-GMapping), a topological navigation subsystem, and a metric navigation subsystem. These subsystems are connected in the same way as LEMTOMap, as shown in Figure 3-1. For the metric navigation, the `move_base` ROS package is used. The other three parts were implemented for this thesis.

Figure 4-1 shows an example of how a map build using LEMTOMap-V0.5 looks. In the example a path from associated node 30 was planned to node 1. Because of to the near neighbour edges, intermediate nodes such as 2 and 3 are skipped in the plan. The slightly rotated gray square is the rolling window occupancy grid map. Later in this chapter, in the section on RW-GMapping, it will be explained what causes the grid map at node 21, and 20 to be marked ‘unknown’.

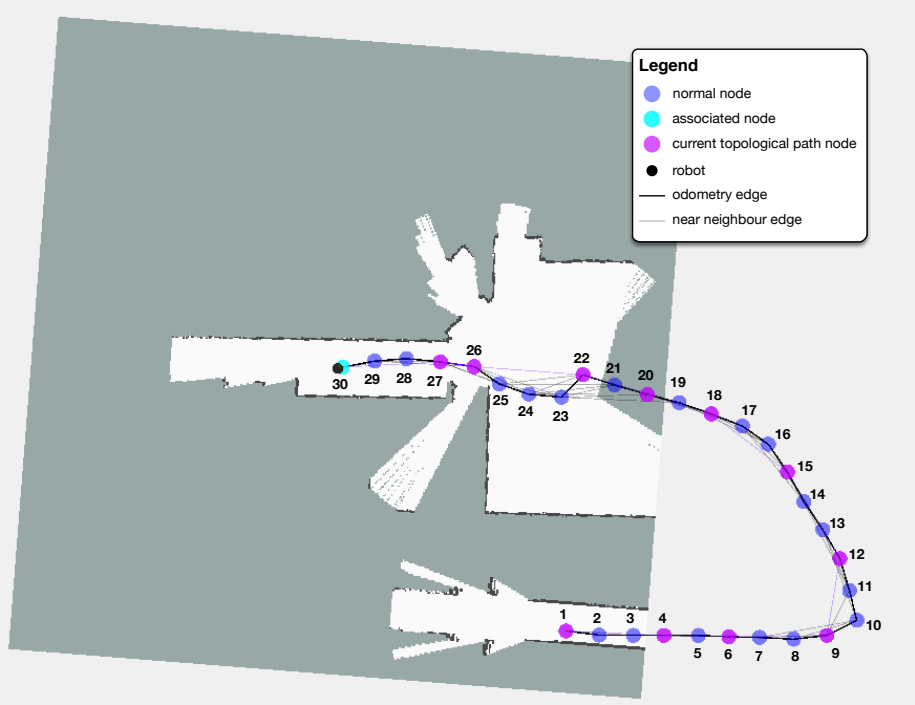


Figure 4-1: A small example map build using LEMTOMap-V0.5. Nodes are annotated with their IDs for explanatory purposes. Currently, a topological path is planned from the robot to node 1 (which resulted in a path formed by nodes 30, 27, 26, 22, 20, 18, 15, 12, 9, 6, 4, and 1 respectively). The slightly rotated gray square is the rolling window occupancy grid map.

4-3 Components of LEMTOMap-V0.5

4-3-1 Topological SLAM

Nodes and edges The topological mapping subsystem implementation has slightly different nodes and edges from the ones defined for LEMTOMap in Chapter 3. For the graph $g = (\mathcal{N}, \mathcal{E})$, consisting of nodes $\mathcal{N} = \{n_1, ..., n_m\}$ and edges $\mathcal{E} = \{e_1, ..., e_p\}$, the tuple definitions are as follows in LEMTOMap-V0.5:

$$\begin{aligned} n_i &= (N_{\text{ID}}, a_{\text{ID}}, \mathbf{s}, \mathbf{P}_{\text{map}}, \mathbf{D}_{\text{map}}, \mathbf{n}_{\text{adj}}, \mathbf{e}_{\text{adj}}, t_{\text{dijkstra}}) \\ &\in \mathcal{N}_{\text{ID}} \times \mathcal{A}_{\text{ID}} \times SE(2) \times \mathcal{F}_{\text{pred}} \times \mathcal{F}_{\text{dist}} \times \mathcal{N}_{\text{ID}}^{j(i)} \times \mathcal{E}_{\text{ID}}^{j(i)} \times \mathbb{R} \\ e_i &= (E_{\text{ID}}, S_{\text{nid}}, T_{\text{nid}}, c, E_{\text{type}}) \in \mathcal{E}_{\text{ID}} \times \mathcal{N}_{\text{ID}} \times \mathcal{N}_{\text{ID}} \times \mathbb{R}^+ \times \mathcal{E}_{\text{type}} \end{aligned}$$

The relative pose has moved from the edge to the node, and the pose uncertainty is not specified any more. The node pose $\mathbf{s} \in SE(2)$ is defined in a global coordinate frame. The used coordinate frame is the `topo_map` frame, on which more will be explained in the paragraph on coordinate frames. As one global frame is used to define node poses, the relative poses are now just called poses. The pose uncertainty in LEMTOMap is used to solve the local graph

constraints. As local graph constraint solving is not part of LEMTOMap-V0.5, there is no need yet to include such an uncertainty in LEMTOMap-V0.5.

The edges have gained a cost property c, which specifies the cost of traversing that edge. This cost is implicitly also defined by the relative pose between the two nodes it connects.

Nodes do not yet store localization data, which nodes should provide to enable the topological mapping system to find the local grid map transform. This data is not yet added, as maintaining the local grid map transform is not fully implemented in LEMTOMap-V0.5.

Nodes carry five extra properties; a predecessor map ($\mathbf{P}_{\text{map}}$), a distance map ($\mathbf{D}_{\text{map}}$), an adjacent nodes vector ($\mathbf{n}_{\text{adj}}$), an adjacent edges vector ($\mathbf{e}_{\text{adj}}$), and an update time stamp (t_{dijkstra}), which are used to store the solutions from Dijkstra's shortest path algorithm.

The node and edge tuples are defined in C++ classes, which can be straightforwardly expanded with additional properties. This way, future integration with semantic mapping should be possible in a relatively easy way.

Dijkstra's Algorithm related node details In LEMTOMap-V0.5, nodes store their own solutions from Dijkstra's algorithm. Dijkstra's algorithm finds the shortest path to any other node for some specific node. The solution for node n_i is stored in the four properties $\mathbf{P}_{\text{map}}$, $\mathbf{D}_{\text{map}}$, $\mathbf{n}_{\text{adj}}$, and $\mathbf{e}_{\text{adj}}$, respectively the predecessor map, distance map, adjacent nodes vector, and adjacent edges vector. The time of when the solution was calculated is stored in t_{dijkstra}. The term map here means that it maps a node ID to a distance or a predecessor, so this term should not be confused with a map as created in robotic mapping.

The distance map of n_i is used to store a mapping from a node ID to the shortest topological distance from that node to the node n_i as described in Section 2-3 (the *dist* vector in Algorithm E-2, page 113). $\mathbf{P}_{\text{map}}$ is in the function space of functions mapping node IDs to node IDs, defined as $\mathcal{F}_{\text{pred}} = \{P_{\text{map}} : \mathcal{N}_{\text{ID}} \rightarrow \mathcal{N}_{\text{ID}}\}$. In practice $\mathbf{P}_{\text{map}}$ can be stored in a matrix that is $(m-1) \times 2$, with m being the number of nodes, the left column containing the source node IDs, and the right column containing the preceding node IDs.

The predecessor map stores a mapping from one node ID to another node ID, which can be used to construct the shortest topological path through an iterative process (the *previous* vector in Algorithm E-2, page 113). $\mathbf{D}_{\text{map}}$ is in the function space of functions mapping node IDs to distances, defined as $\mathcal{F}_{\text{dist}} = \{D_{\text{map}} : \mathcal{N}_{\text{ID}} \rightarrow d\}$. In practice $\mathbf{D}_{\text{map}}$ can be stored in a matrix that is $(m-1) \times 2$, with m being the number of nodes, the left column containing node IDs, and the right column containing distances.

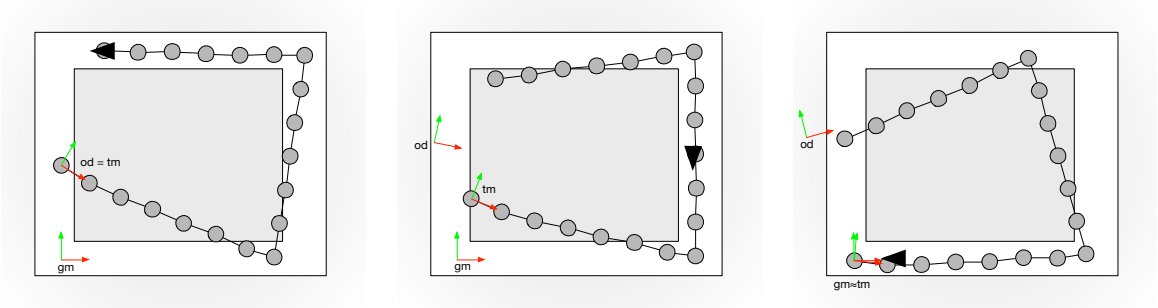
The adjacent nodes and adjacent edges vector respectively contain the node neighbours and the edges connecting to those neighbours. These size of these vectors is equal to the number of neighbours j, which depends on the specific node i.

The topological mapping subsystem keeps track of when the last change took place that could invalidate previous Dijkstra's solutions. Any newly added node or edge makes all per node stored Dijkstra's solutions likely invalid. Therefore, if any of these details is requested and the node's t_{dijkstra} is older than last of such a change, the details will get updated automatically. By storing the details in the nodes, and only generating them if requested and outdated, it is prevented that Dijkstra's Algorithm runs more than necessary.

Local graph solving The topological mapping component of LEMTOMap-V0.5 does not yet support local graph solving. This is related to the fact that nodes explicitly define their pose in the `topo_map` frame of LEMTOMap-V0.5, instead of that it is defined implicitly by relative poses defined in the edges (as in the original LEMTOMap architecture). As node poses are defined explicitly in a global frame, there is nothing to solve. Consequently, edges cannot bridge metric inconsistency gaps such as in Figure 3-7. The kind of loop closing shown in that example requires solving the graph locally, as it involves edges that define constraints that are not metrically consistent with the real local world until the constraints are solved locally. In theory, LEMTOMap-V0.5 will work perfectly without local graph solving if there is no metric inconsistency ($D_{\text{nn\_tresh}}$ of Figure 3-6 can be infinite) at all, or if $D_{\text{nn\_tresh}} = 0$. In practice, the actual allowable setting of $D_{\text{nn\_tresh}}$ is a trade-off between the metric consistency of topological graph and $D_{\text{nn\_tresh}}$. The higher the metric inconsistency, the more limited the maximum implicit loop closure becomes.

Coordinate frames In LEMTOMap-V0.5, there are three important frames at the map level: `odom`, `grid_map`, and `topo_map`. These frames are illustrated in Figure 4-2. To make the example more easy to understand, it is assumed that RW-GMapping performs perfect mapping and localization, while odometry is noisy and graph nodes are placed in `topo_map` based on pure odometry. The `grid_map` and `odom` frames are placed where the robot begins, and then track the place where the robot has started according to RW-GMapping and odometry respectively. As RW-GMapping performs perfectly, the `grid_map` origin will remain at the place where the robot has truly started. As can be seen in the leftmost figure, after travelling from the lower left corner, the odometry frame has developed a translation and rotation error compared to the grid map frame. As the robot has been creating new nodes all the time, the local grid map transform has needed an update (See flow chart of Figure 3-6), and thus, under our example definition, the `odom` and `topo_map` frames are still the same. When the robot starts to travel back (middle and right image), the local grid map transform makes sure that the the topological nodes, robot, and grid map align properly locally. As odometry errors will continue to accumulate, the odometry frame will continue to have an ever changing relative pose to the grid map frame. The local grid map transform takes care of setting the right relative pose of the topological map frame compared to the grid map frame. When the robot is back at its starting position, the grid map and topological map frames should be more or less the same, thanks to the local grid map transform.
To summarize, the relations are as follows: RW-GMapping is used to keep track of the relative pose of `grid_map` to `odom`. The local grid map transform, as part of the topological mapping system, is used to keep track of the relative pose of `topo_map` to `grid_map`. As RW-GMapping tries to improve the localization based on odometry only, the `grid_map` frame can be considered as a frame which provides an improved version of the odometry (changes in location in the `grid_map` frame over time should be more accurate that changes in location in the `odom` frame).

Please note that this just an example to explain the relations of the frames. Its perfect RW-GMapping assumption is not representative nor required for LEMTOMap and LEMTOMap-V0.5. In LEMTOMap and LEMTOMap-V0.5, nodes are not placed based on movement according to odometry (i.e. in the odometry frame), but on movement according to the RW-GMapping algorithm, which is used to provide more accurate localization (i.e. in the grid map frame).



a **b** **c**

Figure 4-2: Relations between the `odom` (od), `grid_map` (gm), and `topo_map` (tm) coordinate frames. For this example, grid mapping works perfectly, and the nodes of the topological map are placed based on odometry. *N.B. Because of these assumptions, this example does not reflect the actual working of either LEMTOMap-V0.5, or LEMTOMap.*

Local grid map transform The local grid map transform is partially implemented currently. LEMTOMap-V0.5 can handle a local grid map transform properly, but it can not yet derive what is the proper grid map transform to use (which should be done by finding the relative pose between the associated node and the grid map, based on the localization data stored in that associated node, see Chapter 3). The proper grid map transform can be derived if one has access to the ground truth pose of the robot, which is of course not possible in real-life, but is possible in simulation. Currently, as a temporary solution, solving the local grid map transform using the ground truth pose is implemented in LEMTOMap-V0.5. Consequently, LEMTOMap-V0.5 is currently primarily aimed at usage in simulation. However, real-life application will also work as long as the RW-GMapping grid map is non-rolling (e.g. it is set to be larger than the area explored by the robot.

Save and load maps Topological mapping in LEMTOMap-V0.5 can save topological maps, and load them to be used later.

Explicit topological loop closing Topological mapping does not yet support explicit topological loop closing. Implicitly, loops can be closed using the near neighbour edges, as explained in Chapter 3).

Area segmentation Topological mapping in LEMTOMap-V0.5 does not yet support door detection, and consequently does not support area segmentation (grouping nodes into areas). As a result, area based navigation could not yet be implemented in LEMTOMap-V0.5.

4-3-2 Topological Navigation

Topological navigation in LEMTOMap-V0.5 implements nearly all functionality of the topological navigation defined in the LEMTOMap architecture. The main difference is that the LEMTOMap-V0.5 implementation does not support area based navigation, as topological mapping does not yet support area segmentation.

As shown in Figure 4-1, the topological path found for a topological goal is marked purple.

4-3-3 RW-GMapping

The implementation of RW-GMapping for LEMTOMap-V0.5 slightly differs from how it is described in the original LEMTOMap architecture. This is mainly caused by how maps are stored in the particles in the default GMapping implementation, on which our RW-GMapping implementation is based. In the default GMapping implementation, each particle stores a series of historical poses. Each of those historical poses is linked with a laser range measurements vector taken at that point. GMapping gives each particle a weight, the particle with the highest weight can be considered the best particle, as it contains the most likely pose and map estimate. When a particle is the best particle, the map needs to be generated for visualization, which triggers a process that builds the map based on the full history of poses and their scan measurements. In other words; particles do not explicitly store maps, but store historical poses linked to measurements that together implicitly define the particle's map. Consequently, making the particles have 'rolling window' maps as defined in Chapter 3 is not as simple as cropping a map. Our solution is to clear the measurements coupled to all historical poses that are outside the rolling window. So, as soon as a historical pose moves outside of the current local grid map, its scan measurements are cleared. Scan measurements from that historical pose can (and often will) have measured points that are inside the current local grid map. The simple reason that this can occur is, that if the robots historical pose was outside the local grid map, it could have looked in the direction of the local grid map and thus scan measurements have registered parts that were within the local grid map. This fact can cause places near the border of the local grid map to become marked unknown. This can be observed in Figure 4-1, at node 20, and 21. As the robot moved more west, the sliding window made a shift. After that, the historical poses (not shown) near node 19 have fallen outside of the window. Their measurements are deleted and apparently those measurements were the only ones covering the space under nodes 20 and 21. So now, this space is marked unknown. Overall, this implementation causes RW-GMapping to have sliding windows that do not nicely 'crop' the map on the edge of the window, but it does properly forget what it had seen beyond the border of the window. So, despite not visualizing as nicely as proposed in LEMTOMap, the implementation of RW-GMapping in LEMTOMap-V0.5 does the job effectively too.

Please note that the window size should be set to be at least twice as large as the maximum range of laser scanner, as otherwise it can occur that RW-GMapping will resize the map every cycle of the algorithms loop.

4-4 Summary

LEMTOMap-V0.5 gives a first, basic implementation of the full proposed LEMTOMap architecture. Several features are not implemented, including local graph solving, door detection, area segmentation, and explicit topological loop closing. Additionally, the local grid map transform is not implemented properly, as it relies on a workaround that requires perfect odometry. This is left future work. LEMTOMap-V0.5 allows to test the main principle of LEMTOMap, namely the hybrid-topological map in which the metric map acts as a sliding

window. LEMTOMap-V0.5 can hence be used to show that this principle can be used for reliable mapping, localization and navigation in large environments, while reducing overall time and space complexity. LEMTOMap-V0.5 forms a starting point to test the main concepts of the LEMTOMap architecture. These test are performed in the next chapter.

Chapter 5

Experiments

In this chapter, LEMTOMap-V0.5 as described in Chapter 4 is benchmarked in a series of experiments. First, a short introduction of the experimental setup is given, then simulated experiments are shown. Next, real world application of LEMTOMap-V0.5 is shortly discussed.

In the experiments, we are going to evaluate the mapping, localization and navigation performance of LEMTOMap-V0.5 and compare that to a baseline system which is based on the current state of the art. We are going to compare qualitative results, such as how well the maps are usable for navigation, as well as quantitative results, such as the time and space complexity of both systems.

Using the experiments, we want to show that LEMTOMap-V0.5 has lower time and space complexity compared to current state of the art, when used in large mapping and navigation tasks, without compromising on mapping and navigation performance. We will perform experiments on both small scale and large scale environments (see Figure 5-2).

5-1 General experimental setup

Platform description

For our experiments, we made use of a TurtleBot 2 robot (see Figure 5-1a). The TurtleBot consists of a base that can be controlled to drive around, and that provides several sensors, including a gyroscope, odometry, bumping sensors, cliff sensors, and wheel drop sensors. On top of the base a structure is placed which allows to store a laptop (which is needed to control the robot), and gives the possibility to attach actuators and sensors to the robot. By default, the TurtleBot comes with a Kinect RGBD sensor, which provides 3D range information. The overall robot is fairly small, being about 36cm in diameter and 42cm in height.

The TurtleBot can be controlled using any laptop running ROS. The driver to control the base provides ROS with gyro enhanced odometry, resulting in very accurate odometry. Although the TurtleBot is quite limited in overall features, it forms an inexpensive way (less than

2000USD) to perform experiments in robotics in real life, especially it is very suitable for mapping and navigation related research. The largest downsides of the TurtleBot are its limited field of sight (the Kinect has a viewing angle of 43° vertical, 57° horizontal, with a range of 0.8 to 4 meter[1]), the lack of a laser range sensor, and its limited ability to drive over bumps (it can only drive over objects lower than 12mm high). The latter causes it to be unable to pass most of the doorsteps frequently encountered in buildings.

For our experiments, we added a Hokuyo URG-04LX-UG01 laser range sensor to the TurtleBot, which has a range of about 5.6m and covers an angle of 240°. The field of view of the Hokuyo is limited from ±120° to ±90°, as it would otherwise see its own poles, which will confuse scan-matching algorithms used in SLAM.

The Hokuyo is added to give the robot access to accurate, wide angle range data. Also, the Hokuyo has a somewhat larger maximum range than the Kinect. Especially the wide viewing angle is important for grid map SLAM algorithms such as (RW)-GMapping to perform well, as they heavily rely on laser scan-matching algorithms.

As the Hokuyo came into the bottom of the field of sight of the Kinect, we moved the Kinect forward, such that it looks over the Hokuyo. Our adapted version of the TurtleBot is shown in Figure 5-1b.

Figure 5-1: *(a)* The original TurtleBot 2. *(b)* Our TurtleBot 2, with laser range scanner. *(b)* Our TurtleBot 2, simulated.

Sensor description

In our experiments, we make use of three different sensors: the gyroscope, the odometry sensor, and the Hokuyo laser range scanner. As the ROS TurtleBot base driver automatically integrates the gyroscope data with the odometry data, we only need to interface with the odometry and laser range data provided by ROS.

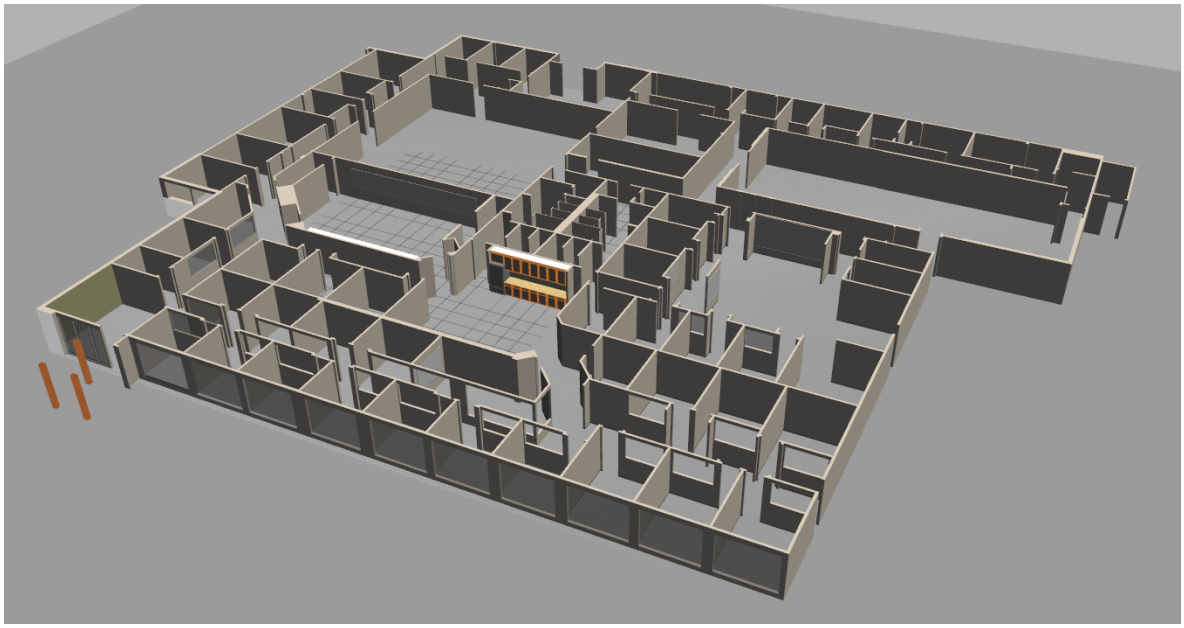
[1] See: http://msdn.microsoft.com/en-us/library/jj131033.aspx and http://msdn.microsoft.com/en-us/library/hh438998.aspx

- Gyroscope (part of the base of the TurtleBot) - The ROS TurtleBot base driver automatically integrates the gyroscope data with the odometry data to provide enhanced odometry accuracy.
- Odometry (part of the base of the TurtleBot) - The wheels register 11.5 ticks per driven mm. This, together with the integration with the gyroscope for rotational accuracy, delivers very accurate odometry. The odometry is used to estimate the change in pose of the TurtleBot.
- Hokuyo URG-04LX-UG01 laser range scanner - Range of 20 to 5600mm at 240°. Accuracy ±30mm up to range of 1000mm, ±3% of range if range is > 1000mm. The sensor has an angular resolution of 0.352°. The laser range sensor is used for building a grid map using the RW-GMapping algorithm, and for local obstacle avoidance in the metric navigation subsystem.

Robot and world simulation

All experiments were performed with a computer simulated robot in a computer simulated environment. Simulations offer several advantages over real world experiments. First of all, the environment we need to test on needs to be large (e.g. a full faculty building), which makes testing in real life impractical. Additionally, the TurtleBot can not drive over door thresholds, and can not open doors. This makes testing on a large scale infeasible within the TU Delft campus.

The TurtleBot is simulated in Gazebo; a robot simulator than can simulate the robot with its actuators and sensors in a simulated world. Gazebo includes dynamics simulations using physics engines and aims at generating realistic data, including sensor noise. The simulated sensors were configured to resemble the properties of the sensors used by our real TurtleBot. Simulations are run in real-time.
For the small scale simulations, a building with one floor is simulated (see Figure 5-2a). There are no static nor dynamic objects (such as persons), and no doors are part of the simulations. To generate an environment of larger scale (similar to the size of a faculty building), the building floor is placed several times next to each other, as shown in Figure 5-2b. The simulated robot is shown in Figure 5-1c.



a

b

Figure 5-2: *(a)* The small scale simulated world. *(b)* The large scale world, created by placing the small scale simulated world multiple times next to each other.

Processing power description

For the experiments, a Dell Precision M4600 workstation laptop was used with an Intel 2.5GHz Core i7-2860QM quad-core, 16GB of working memory, and an Nvidia Quadro 2000M GPU. The laptop runs Ubuntu 14.04 and ROS Indigo. The laptop was both used for simulated experiments as well as for controlling the TurtleBot in real world application. In simulated experiments, the laptop both ran the Gazebo simulator (to simulate the robot and world) and the LEMTOMap-V0.5 system.

Measuring resource usage

In the experiments, the time and space complexity of the algorithms is evaluated. To provide a way to measure time and space complexity, the usage of system resources is measured. Each of the subsystems (metric mapping, topological mapping, metric navigation, and topological navigation) runs in its own executable. In our experiments, the resources used by these executables is measured.

Three different types of resource usage are monitored.

- Memory - Used to measure space complexity. The memory[2] used by the process is a measure to see how much space (memory) is needed to run the implementation of the algorithm.
- CPU time - Used to measure time complexity. CPU time used by a process is the total time of CPU cycles claimed by a process. The CPU time spent by a process thus represents how many CPU calculations/operations were needed to run the implementation of the algorithm.
- CPU utilization (or equivalently: CPU load) - Used to measure time complexity. The CPU utilization is in fact the time derivative of the CPU time, and is often expressed as a percentage. The CPU utilization is the fraction of a CPU's capacity that is used by a process. Hence, the CPU utilization is in fact the time derivative of the CPU time usage. CPU utilization generally switches at a very high frequency, so it is normal that such measurements tend to look noisy.

In the context of this report, a thread is what is meant by a CPU. This means that on our 4 core system, which has two threads per core, the available capacity is 800% of CPU utilization, and 8 seconds CPU time is available per second of real world time.

To make sure that benchmarks give consistent results, dynamic CPU clock speed techniques (Intel SpeedStep™ and TurboBoost™) are disabled. The total size used on disk to store a map is the last kind of resource usage that is measured.

[2]Different definitions of memory usage of a process exist. For our experiments, we measure the *resident memory* used by the process. This is the memory that the process is actually occupying in the computers RAM memory.

5-2 Simulated experiments

The simulated experiments are divided in two parts. In the first part mapping is evaluated. In the second part navigation is evaluated in maps built using the mapping experiments.

5-2-1 Mapping using LEMTOMap-V0.5

Using the mapping experiments, we want to show the gains of LEMTOMap-V0.5 in terms of time and space complexity when compared to current state of the art mapping and navigation.

Approach

To benchmark the mapping process, LEMTOMap-V0.5 will be compared with a baseline comparison system, which forms the metric equivalent of LEMTOMap-V0.5.

First, without any mapping or navigation service running, the robot was driven around manually in the simulated world. The odometry and laser scan data of this drive was recorded. Two different recordings were made, one drive of about 3 minutes in the default world and one of about 11 minutes in the large world. The robot was driven around continuously at a constant speed.

The recorded sensor data was applied as an input to both LEMTOMap-V0.5 and the baseline system. By using this procedure, experiments are repeatable in a consistent way, making results reproducible. Each experiment was repeated 3 times and averaged to limit the effect of stochastic processes.

As a baseline comparison for the mapping experiments, we use GMapping as the metric SLAM algorithm, and `move_base` as the metric navigation subsystem. GMapping and our own RW-GMapping use the same settings, which are based on the default settings for the TurtleBot. The only exception is the number of particles. RW-GMapping for LEMTOMap-V0.5 uses 5 particles[3], while GMapping for the baseline system uses 80 particles (default setting for the TurtleBot). For GMapping, we use an initial map size of 20 by 20m. The same size is used as the rolling window size in case of RW-GMapping. `move_base` provides metric navigation for both LEMTOMap-V0.5 and the baseline system. For `move_base`, the same settings will be used both for LEMTOMap-V0.5 and the baseline system. LEMTOMap-V0.5 consists of four subsystems, RW-GMapping, Metric Navigation (`move_base`), Topological Mapping, and Topological Navigation. The baseline system consists of two subsystem, GMapping, and Metric Navigation (`move_base`).

Measurements

In the mapping experiment, the CPU time, CPU utilization and memory used by all subsystem processes are measured, to give insight in the time and space complexity of each subsystems. Additionally, the cumulative resources used by LEMTOMap-V0.5 and the baseline system will be compared. The cumulative resources used are found by adding up the resources used by

[3] 5 particles is sufficient for the small scale maps RW-GMapping needs to maintain, as explained in Chapter 4

the individual subsystems. Lastly, the maps generated by LEMTOMap-V0.5 and the baseline system will be stored to disk, such that they can be loaded for later usage. The total size on disk of both maps will be compared.

Space complexity is expected to be lower for RW-GMapping than GMapping, as less particles are used, and smaller maps are contained in the algorithm. As RW-GMapping has particles with fixed size maps, the memory usage is expected to be near to constant. This assumes, that memory usage of (RW)-GMapping mainly driven by the size of particles. For the same reason, space complexity is expected to be lower or even near constant for metric navigation, as the costmaps maintained by metric navigation for LEMTOMap-V0.5 will be smaller in size (the global costmap size is equal to the size of the occupancy grid map of (RW)-GMapping). In a cumulative comparison, the subsystems of the topological map and topological navigation will add memory to the overall system space complexity, but overall it is expected that the other space complexity reductions will easily compensate for this.

Overall time complexity is expected to be lower as well. The time complexity of RW-GMapping decreases compared to GMapping for the same reasons as why space complexity is expected to reduce. Possibly, the time complexity of metric navigation will reduce slightly as well, as it needs to maintain smaller costmaps.

When regarding space complexity in terms of storage size on disk, the LEMTOMap-V0.5 map is expected to require much less space then the baseline map. LEMTOMap-V0.5 will only need to store the topological map, while baseline needs to store a full size occupancy grid map, which stores to disk like a bitmap image (each pixel is value is stored explicitly).

Results

The small scale mapping runtime resource usage results are shown in Figure 5-3, 5-4, and 5-5. The large scale mapping runtime resource usage results are shown in Figure 5-6, 5-7, and 5-8. The size of the maps stored on disk are shown in Table 5-1.

From Figure 5-3 and 5-6 it can be concluded that LEMTOMap-V0.5 indeed has an overall lower time and space complexity than the baseline system, except for a higher time complexity after about 2/3 of the large mapping experiment, which will come back to later. First, we will look at the results into more detail in Figure 5-4 and 5-7.

RW-GMapping in LEMTOMap-V0.5 uses much less memory than GMapping in the baseline system. This is because less particles are used an maps are smaller. It can be observed that the memory usage of GMapping for the baseline system stays flat at several points, most notably in the end. This is because the was not resized during these periods. At first, the robot mostly explores and the mapped area is expanded regularly. In the end, the robot mainly stays within the window currently covered by the map and thus memory usage stays flat. When closely looking at RW-GMapping for LEMTOMap-V0.5, it becomes clear that memory usage keeps on growing slightly most of the time. Perhaps more than one would expect. Further inspection reveals that memory usage increases every time the sliding window is shifted. Possibly, a bug exists in the memory deallocation of the data carried by the particles. Further inspection with for example memory profiling software is needed to verify this. As the memory usage is very low overall compared to normal GMapping, this is of minor priority.

Metric navigation uses more or less the same amount of memory in both LEMTOMap-V0.5 and the baseline system. It was expected that memory usage would be significantly lower for LEMTOMap-V0.5. For the large scale experiment, a slight improvement can be noticed, but much less than expected. To our knowledge, no other rolling window grid mapping algorithms exist other than our own RW-GMapping. It is hence safe to assume that `move_base` was not designed for handling a shifting map. Possibly, a memory deallocation bug causes the observed memory usage. As the impact on overall memory usage is low, and as `move_base` is developed by the open source ROS community, we would like to leave fixing this as future work of minor priority.

Topological navigation shows low and constant memory usage. Which was expected, as no costmaps need to be maintained.

Topological mapping shows a more or less linear increase in size of the memory usage. As the robot travels continuously, the graph size grows more or less continuously as well. The memory usage thus seems to grow linearly with the size of the graph, which was expected.

When looking at the time complexity of RW-GMapping (LEMTOMap-V0.5) and GMapping (baseline system) in terms of CPU time usage and CPU load, it can be observed that GMapping gives a very high load for the baseline experiments. This is mainly due to the amount of particles. The load stays more or less flat with the map size.

Metric navigation has more or less the same time complexity for both LEMTOMap-V0.5 and the baseline system.

Topological navigation runs idle, as the mapping experiment does not use topological navigation. Hence, only a constant amount of CPU time is used per cycle of the program, resulting in a linear increase in CPU time usage.

Topological mapping shows a more or less linear increase in CPU load, and hence, a quadratic increase in CPU time usage. Further inspection shows that the CPU load scales more or less linearly with the topological graph size . That is, if the robot stops moving or moves along the existing graph (i.e. no new edges or nodes are created), the load stays constant. This behaviour shows that some part of the algorithm takes more time to complete as the graph grows. This is unexpected behaviour, and must be caused by a bug in the implementation. Solving the bug can be done in a straightforward manner. CPU time usage can be timed within a program. This can be used to time CPU usage of all the functions within the implementation. Reading the results of this timings should quickly reveal the cause of this behaviour, and a fix can be applied to solve the bug. Solving this bug is regarded the first priority in future work. This bug is also what causes the cumulative load of LEMTOMap-V0.5 to eventually become larger than that of the baseline system in the large scale mapping experiment (Figure 5-6).

The size to store maps to disk presented in Table 5-1 shows a large improvement. About 97% less memory is needed to store a map created using LEMTOMap-V0.5. This could for example be used to store maps of locations when they are not needed. A robot for example does not need to have the map of the supermarket in mind when it is currently at the university.

The maps generated in the experiments are shown in Figure 5-9a, 5-9b, 5-10, and 5-11 . Figure 5-11 shows that the baseline system was unable to properly map the environment. Part of the map has become unusable for navigation.

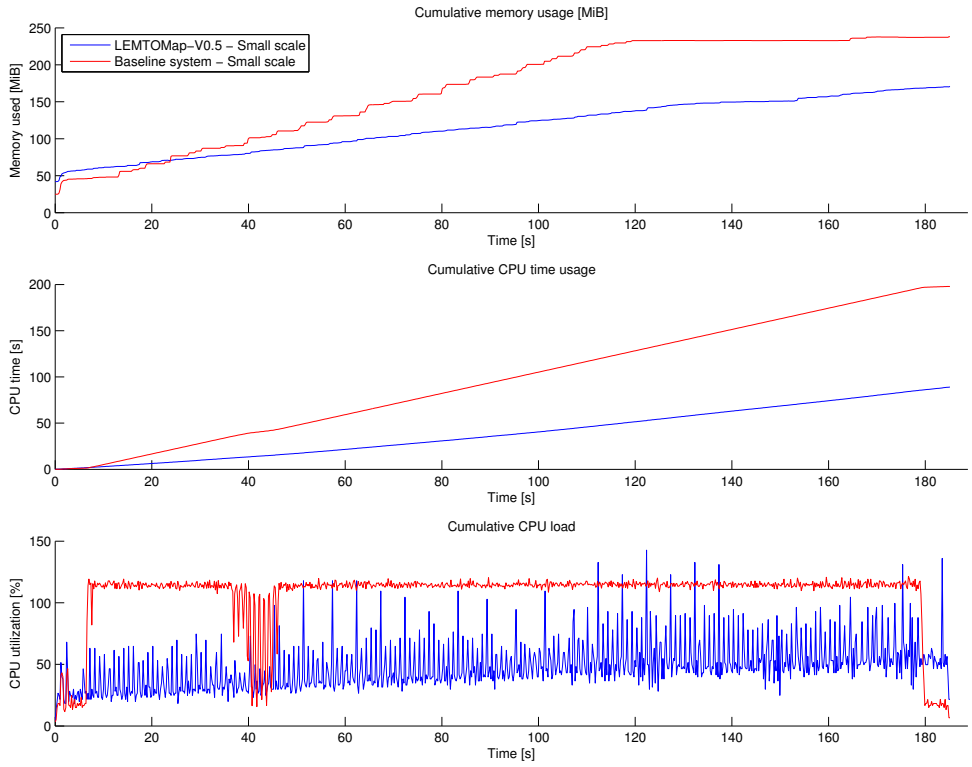
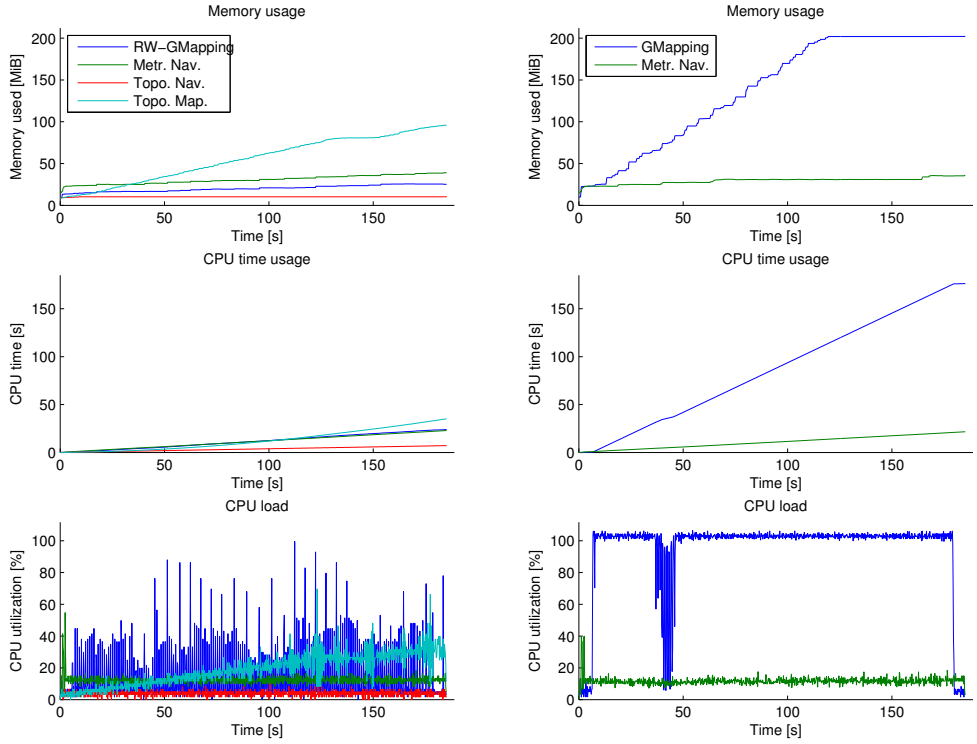


Figure 5-3: *Small scale mapping experiment results.* Cumulative resources used by LEMTOMap-V0.5 versus the baseline system. *N.B.* $1\text{MiB} = 1024^2\text{B}$, $1\text{MB} = 1000^2\text{B}$

	LEMTOMap-V0.5	Baseline	Difference [%]
Size [KB] (small scale experiment)	24	710	−96.6%
Size [KB] (large scale experiment)	72	2671	−97.3%

Table 5-1: Size of the maps generated in the mapping experiments, stored on disk.



LEMTOMap-V0.5 **Baseline system**

Figure 5-4: *Small scale mapping experiment results.* Resources used per process by LEMTOMap-V0.5 versus the baseline system. The left column shows LEMTOMap-V0.5, the right column shows the baseline system. Left and right y-axes use the same scale for easier comparison.

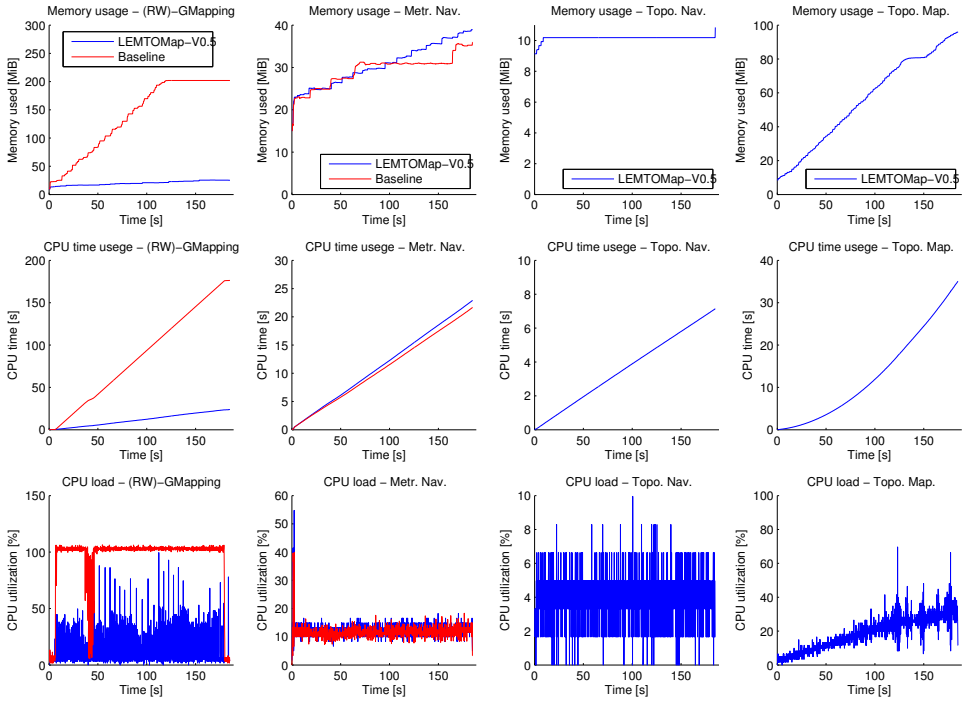


Figure 5-5: *Small scale mapping experiment results.* Resources used per process by LEMTOMap-V0.5 versus the baseline system. Please note that y-axes use different scales.

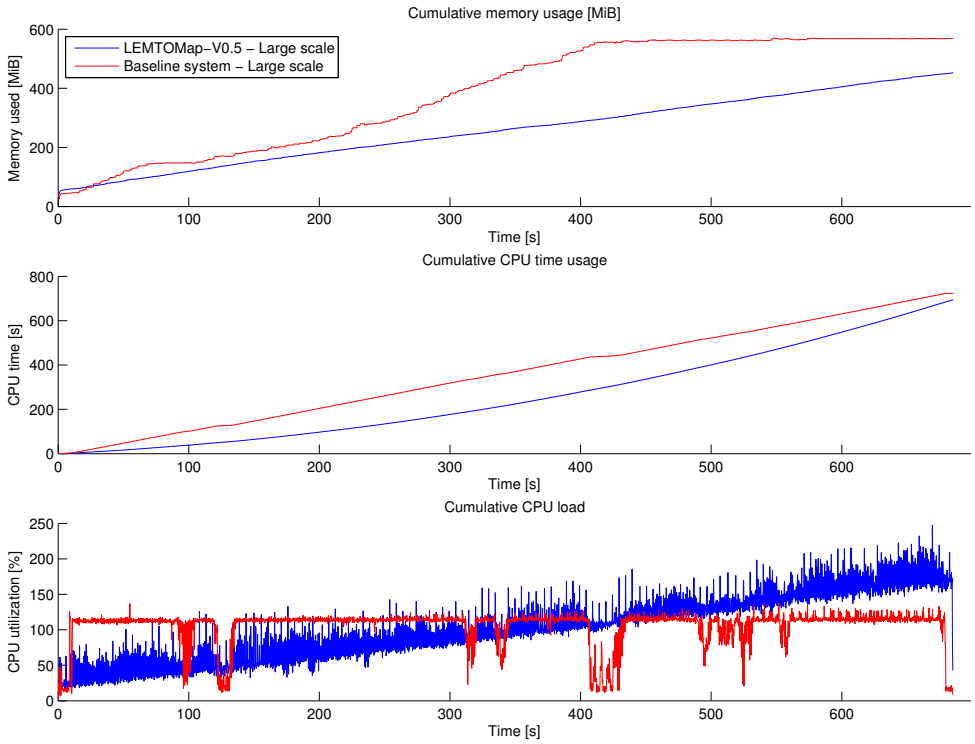
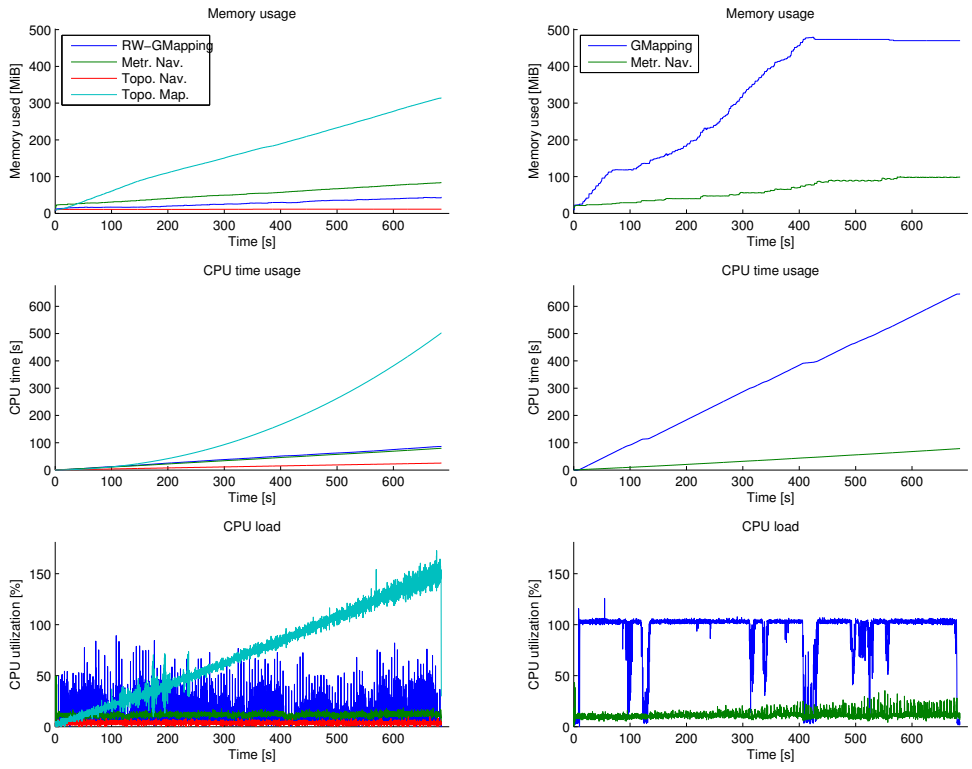


Figure 5-6: *Large scale mapping experiment results.* Cumulative resources used by LEMTOMap-V0.5 versus the baseline system.



LEMTOMap-V0.5 **Baseline system**

Figure 5-7: *Large scale mapping experiment results.* Resources used per process by LEMTOMap-V0.5 versus the baseline system. The left column shows LEMTOMap-V0.5, the right column shows the baseline system. Left and right y-axes use the same scale for easier comparison.

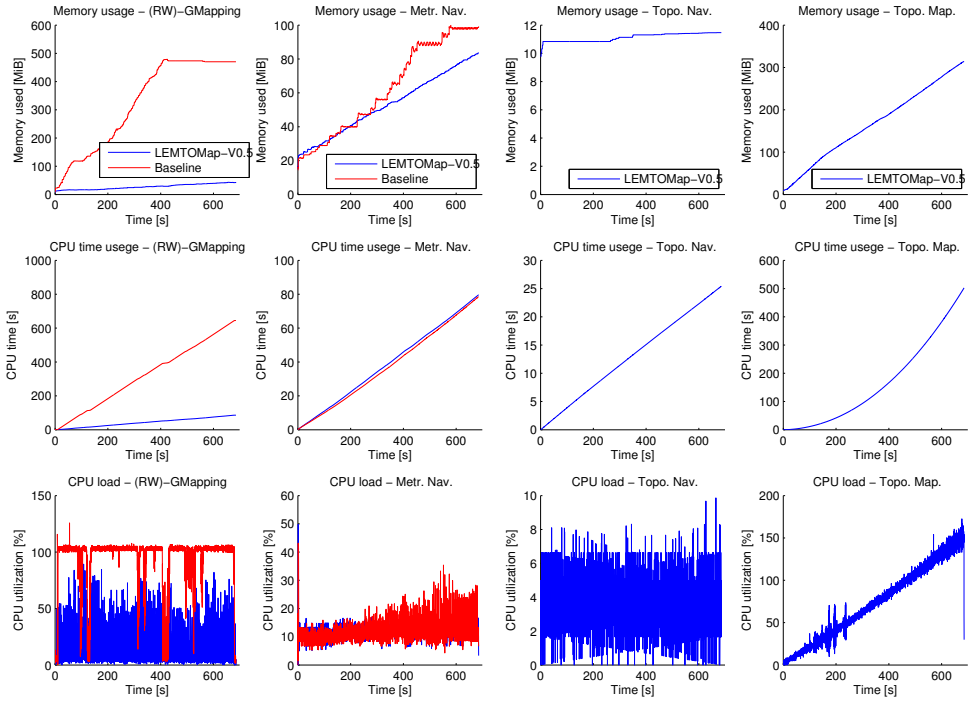
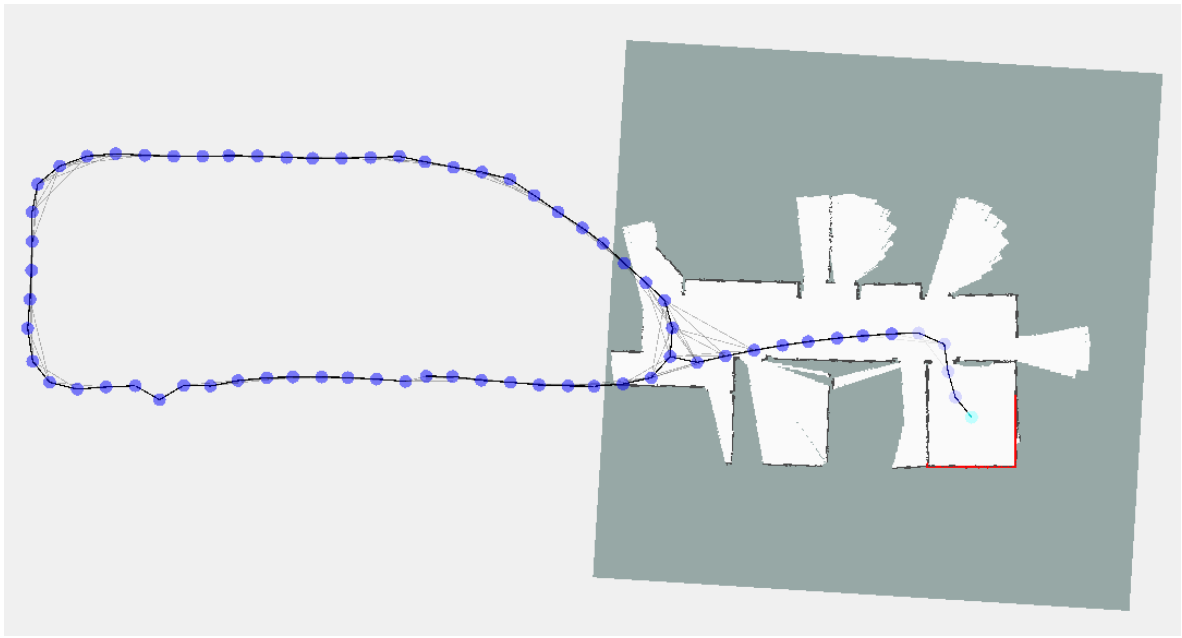
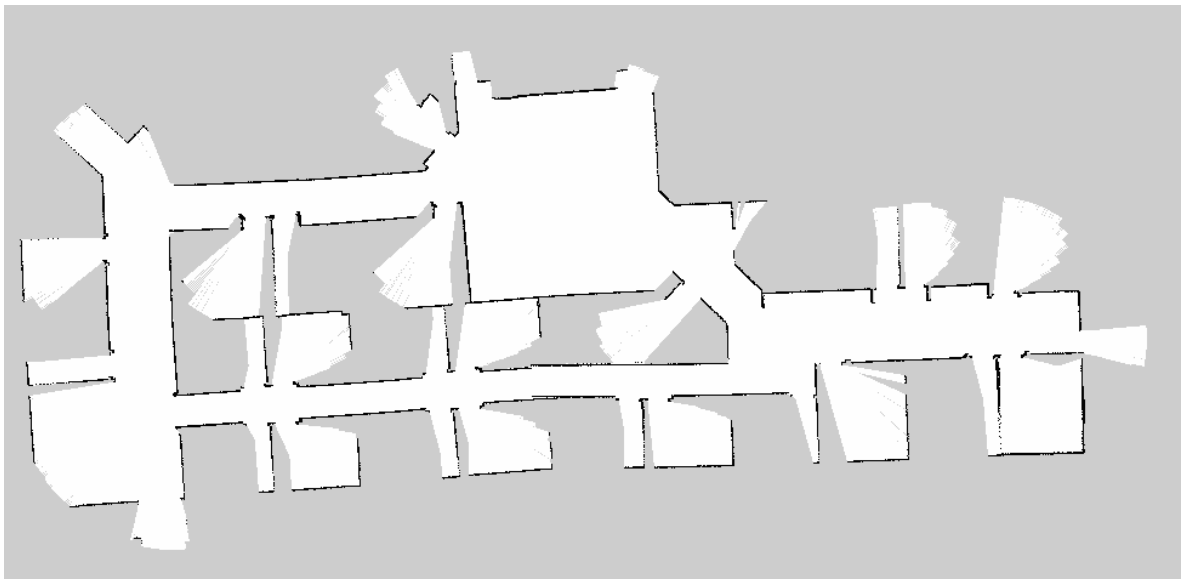


Figure 5-8: *Large scale mapping experiment results.* Resources used per process by LEMTOMap-V0.5 versus the baseline system. Please note that y-axes use different scales.



a



b

Figure 5-9: Maps constructed in the small scale mapping experiment. *(a)* LEMTOMap-V0.5. *(b)* baseline system.

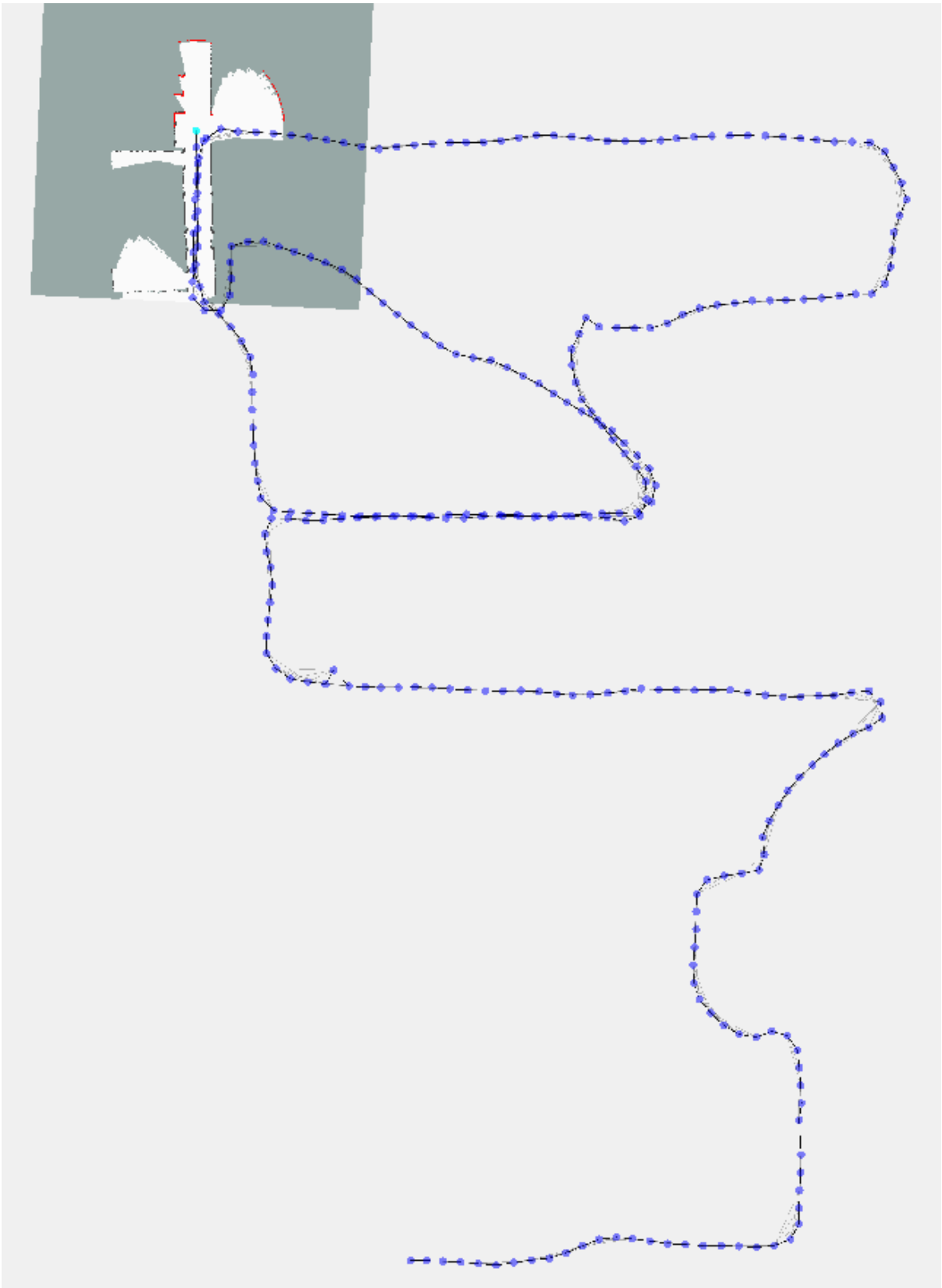


Figure 5-10: Maps constructed in the large scale mapping experiment: LEMTOMap-V0.5.

Figure 5-11: Maps constructed in the large scale mapping experiment: baseline system. The red line is a realistic path that a metric planner could come up with when planning between the black dots. This path is impossible to travel however, resulting in failing metric navigation.

5-2-2 Navigation using LEMTOMap-V0.5

In the navigation experiments, we want to show that our topological navigation has lower time complexity than current state of the art metric navigation. As navigation relies on a proper map and proper localization, we also use these experiments to show that the map and localization generated by LEMTOMap-V0.5 works for navigation.

Approach

We will have a series of navigation tasks performed by LEMTOMap-V0.5 and a baseline comparison system. Two different tasks need to be performed, one with a short trajectory (about 40 meters long) in a small environment, and one with a long trajectory (about 170 meters long) in a large environment. Each trajectory is travelled four times by the robot, as the path is travelled forward and backward, two times. By repeating the same trajectory multiple times, results can be averaged to limit the effect of stochastic processes. Figure 5-12 shows the short being executed by both LEMTOMap-V0.5 and the baseline system.

For the baseline system, we have used the AMCL (Adaptive Monte Carlo Localization) package for ROS, combined with metric navigation provided by the `move_base` package. AMCL can perform localization in a predefined map. The used predefined map is the map that was built in by the baseline system from the mapping experiments of this chapter, i.e. it was built using GMapping.

For LEMTOMap-V0.5, the same parameters were used as in the mapping experiments.

Measurements

In the navigation experiments, we will measure the CPU time it takes the planner to find a global path after receiving a navigation goal, to measure the time complexity of the initial global path planning task. In the case of LEMTOMap-V0.5, that is the time it takes the topological navigation subsystem to find a topological path to the target node. In the case of the baseline system, that is the time it takes the metric planner to find a path to the metric location of that target node. Additionally, the overall time complexity of the full execution of the navigation task is compared, to evaluate if the overall navigation requires less computations. This is measured by measuring the total CPU time spent by the planners. For LEMTOMap-V0.5 that is the CPU time spent by topological planner plus the metric planner, and for the baseline system that is the time spent by the metric planner only.

Furthermore, we measure the total time (in real world time) and distance it takes to reach the goal. We measure this to compare if LEMTOMap-V0.5 can perform navigation as time and distance efficient as pure metric navigation can.

In terms of time complexity, we expect the LEMTOMap-V0.5 to need less CPU time to find a global path. The total CPU time spent in the full navigation task will also likely reduce, as the metric planner will need to solve a lot of small tasks (navigate to each next node of the topological path), rather than one large task. It is expected that these small tasks cumulatively have lower time complexity.

Time to goal will likely slightly increase, as the abstraction of a topological graph can never deliver a path as optimal as a path derived from a metric map. For the same reason, it is also expected that the travelled distance will slightly increase.

Results

Table 5-2 summarizes the results of the navigation experiments.

The CPU time needed to find a global path is indeed lower for LEMTOMap-V0.5. However, even in the large experiment it takes only a couple of milliseconds to find a path. Hence, at this scale, the initial global path planning is of limited influence on overall performance.

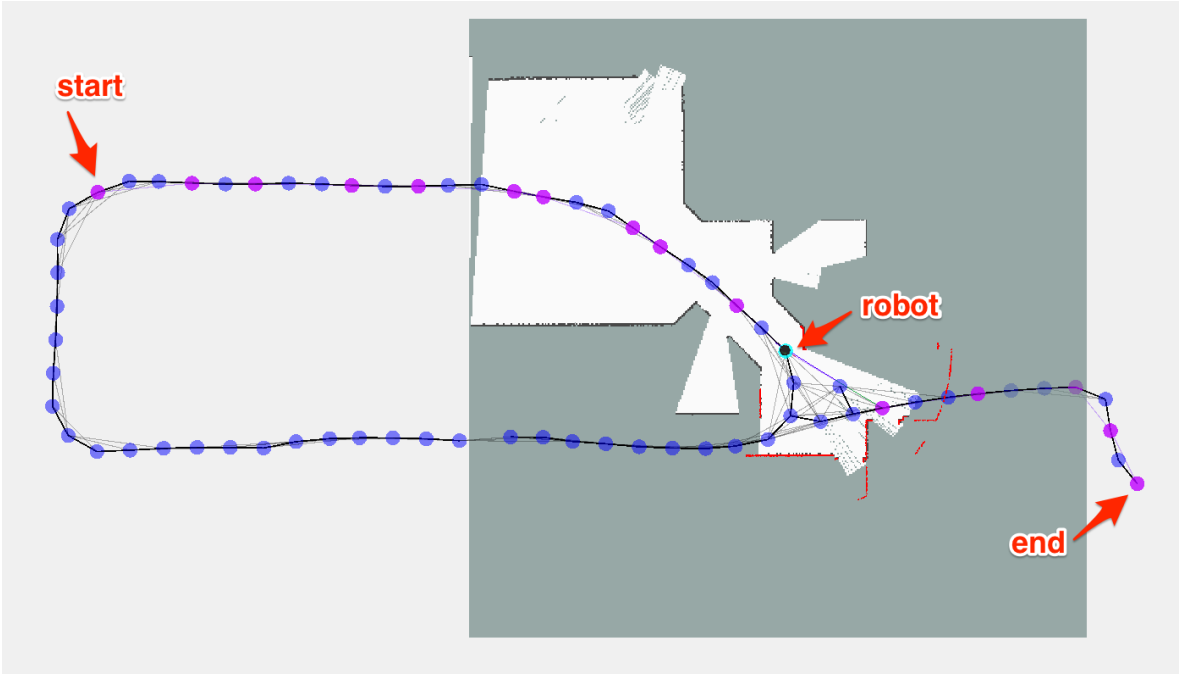
The total CPU time spent show a slight increase in the small scale experiment and a decrease in the large scale experiment. The increase in the small scale experiment is likely caused by the fact that the added CPU time of the topological planner can not be compensated by the reductions in the metric planner. A way to further decrease the total CPU time spent would be to integrate the topological and metric planner. Currently, much work is done double, as the topological planner triggers a series of metric navigation tasks, which uses both it's global and local planner for each of these tasks. Clearly, this could be simplified, because from the definition of the LEMTOMap-V0.5 map, it is known that direct navigability is possible between any subsequent nodes of a topological path. An suggestion to improve this in future work would be to turn the topological navigation subsystem into a global path planner plugin for `move_base`.

The time to reach the goal shows a more than slight increase. Further inspection shows that the robot often slows down after a new node is passed to the metric planner. When a new goal node is passed to the metric planner, it cancels the current goal and starts handling the new goal. In this transitionary period, the robot temporarily receives no locomotion commands. This results in a (often hardly observable) stammering motion, as shown in Figure 5-13. This is a limitation of the current implementation, and can for example be solved by turning the topological navigation subsystem into a global planner plugin for `move_base`.

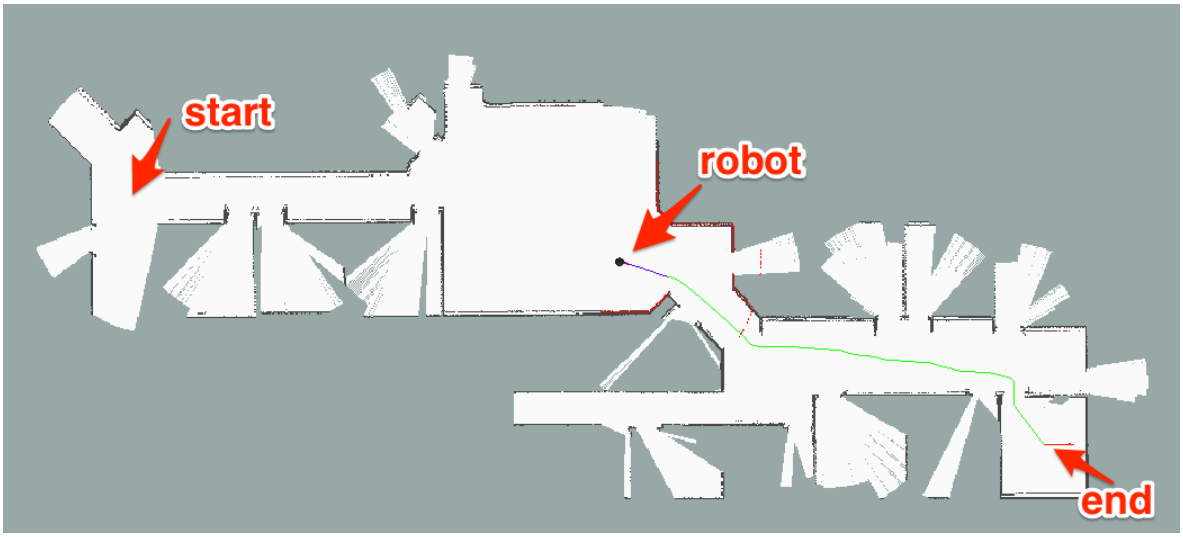
The distance to reach the goal merely shows a slight increase when comparing LEMTOMap-V0.5 with the baseline system. This is expected behaviour and can be considered an acceptable toll of topological navigation.

	Short trajectory			Long trajectory		
	LEMTO-Map	Baseline	Δ	LEMTO-Map	Baseline	Δ
CPU time to gl. plan [ms]	17.74	24.29	−27.0%	24.19	97.62	−75.2%
Total CPU time spent [s]	18.91	16.01	+18.1%	74.48	89.08	−16.4%
Time to reach goal [s]	88.66	82.90	+6.9%	375.25	341.27	+10.0%
Distance travelled [m]	38.81	38.25	+1.5%	171.56	167.86	+2.2%

Table 5-2



a



b

Figure 5-12: The trajectory for the short experiment. Both have exactly the same starting point and end point and navigate the trajectory up and down twice. *(a)* Navigation using LEMTOMap-V0.5. The purple nodes are the nodes part of the topological path, the cyan node (at the robot) is the associated node. *(b)* Navigation using the baseline system. The global plan is shown in green, the local in blue.

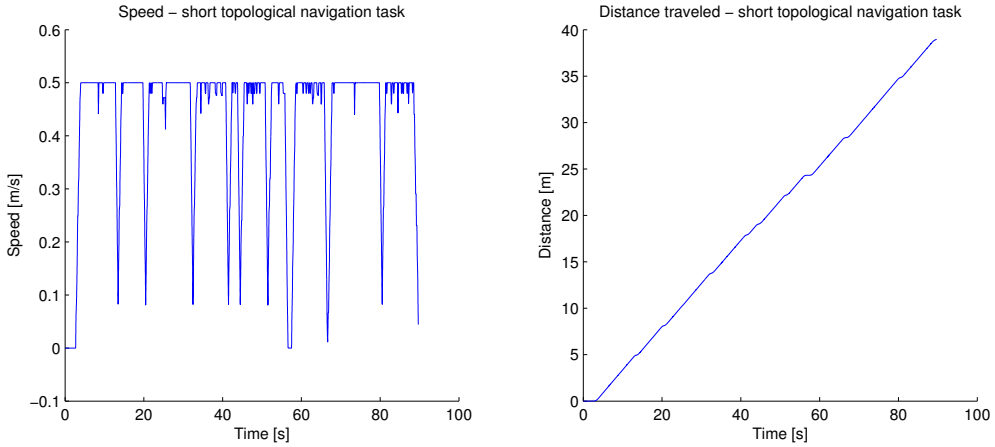


Figure 5-13: Stammering motion of the robot when performing topological navigation. In the right graph, one can observe the slight staircase effect this causes in the development of the travelled distance over time.

5-3 Real world application

LEMTOMap-V0.5 has been run on the TurtleBot in real life as well. Topological mapping and navigation was performed in a small indoor environment at the Delft University of Technology 3mE faculty. The system works as expected, although some problems were caused by poor obstacle avoidance. The limited (planar) field of view of the laser scanner causes the robot to overlook many obstacles. For example, the robot does only see the pole of an office chair, but does not see the base of it. It then can plan paths that collide the robot with the chair base. Unfortunately, in our lively office, this happened quite frequently. A major improvement can be made by incorporating the Kinect of the TurtleBot in the local costmap creation, as the Kinect's field of view is a 3D subset of 3D space, rather than planar (2D subset of 2D space) as is the case with the Hokuyo laser range scanner. The point cloud generated by the Kinect could be used to add (temporary) objects to the local costmap, which will significantly improve performance of object avoidance, and hence of real world navigation.

Chapter 6

Conclusions

In this thesis, it was investigated how robots can take inspiration from humans to perform better in mapping, localization and navigation for large scale, real world environments with the goal to act autonomously for long periods of time. A system architecture was proposed that aims at satisfying those goals, dubbed the Large Environments Metric TOpological Mapping system (LEMTOMap). A first implementation of LEMTOMap was given, which has been tested for qualitative and quantitative performance in a series of simulated tests. In the scope of this thesis, it was only possibly to partially implement LEMTOMap, leaving several of its aspects untested. The partial implementation shows in our experiments that reductions in terms of time and space complexity can indeed be made using LEMTOMap when comparing to a baseline system based on the current state of the art.

As the partial implementation confirms the main benefits, we firmly believe that this radically new approach to robotic mapping and navigation will be able to prove more of its benefits once it is fully implemented. At its core, the thesis shows an example of how to deal with the mapping and navigation tasks in a more human-like way. That is, without the reliance on a map that is globally metric consistent. The novel, hybrid mapping approach taken in LEMTOMap offers interesting new ways to deal with long-standing SLAM problems like the metric consistency problem and the loop closing problem, which will need to be further explored in continued research.

The main novelties introduced in this thesis are captured by the LEMTOMap system architecture. Foremost, a novel hybrid topological and grid map SLAM system is introduced as part of LEMTOMap, which focusses on local metric consistency, without requiring global metric consistency. The topological part of the hybrid system is used for the full scale environment, and the grid map part is used only at a local scale, as it moves along with the robot as a rolling window. As part of the novel mapping system, a new grid map SLAM algorithm is introduced which can operate as a rolling window, which we named RW-GMapping.

The title *Gaining by forgetting* can be justified by a series of (expected) benefits. The system forgets anything of the grid map that is not local. This, combined with the way the topological mapping is defined, allows the system to handle global metric inconsistencies in a robust

way. The most important properties are abstracted to a topological map. As this map is more compact, and easier to maintain, the overall space and time complexity are reduced. Abstraction gains robustness against for example failing loop closure when navigating, and knowledge about for example the room topology. Overall, more system resources will be available for other resource demanding tasks such as computer vision and the derivation of semantics.

Chapter 7

Discussion and future work

In this chapter, the presented work is discussed, and recommendations are made for future work. The chapter is divided in two parts. First, the steps towards a LEMTOMap-V1.0, which would be a full implementation of the LEMTOMap architecture, are discussed. Then, ideas are offered for what could be researched after that to continue the development of a mapping system that is able to deal with large environments autonomously for long periods of time.

7-1 Towards LEMTOMap-V1.0

7-1-1 Implement the rest of LEMTOMap functionality

A method to find a proper local grid map transform should be implemented first. This also requires that the nodes will need to start storing some kind of localization data, as explained in Section 3-3-1. The most straightforward approach would be to store a small local map at the node, or some sensor signature (such as a laser range measurement). This will have a large effect on the storage required to store the map (both in working memory as when stored to disk). To limit this effect, down-sampling the localization data could be considered. This might be possible, as only a coarse alignment of the associated node with the local grid map is needed.

Next, LEMTOMap-V0.5 should be updated to store relative node poses in edges, rather than poses in nodes defined in some global frame. When this is done, implementation of local graph solving becomes possible. For implementing the local graph solving, one could take inspiration from graph-based SLAM algorithms. As the graph constraints only needs to be solved on a local scale, the problem can be dramatically scaled down by pruning off all nodes that are further than some maximum topological distance threshold from the associated node.

When local graph solving is implemented, explicit topological loop closing becomes possible. Various options to achieve this are discussed in Chapter 3.

Door detection, and subsequently area segmentation based on door detection could be implemented apart from the items mentioned before. Once topological mapping provides area

segmentation, area based topological navigation can be implemented in a straightforward way, as described in Chapter 3.

7-1-2 Optimize implementation

In the experiments chapter (Chapter 5), several suggestions are made to optimize the current LEMTOMap-V0.5 implementation, to further reduce time and space complexity. Most importantly, the time complexity bug of the topological mapping subsystem needs to be fixed. A clear approach to solving the bug has been described in the experiments chapter already. Furthermore, memory usage by RW-GMapping can possibly be further reduced, as explained in the experiments chapter. Further inspection using a memory profiler is advised to determine what share of the memory usage in RW-GMapping is caused by what part of the implementation.

7-1-3 Further improvements

Currently, GMapping is used as a basis for our rolling window grid map SLAM algorithm. However, other, less complicated SLAM algorithms could maybe suffice as a basis for a rolling window grid map SLAM algorithm as well. A disadvantage of GMapping is that a switch in what is regarded the best particle can cause a jump in the robots pose estimate as defined in the `grid_map` frame. Consequently, when comparing positions in different frames, it is needed to check time stamps of these poses to make sure no poses are compared based on pose estimates of different particles . This makes dealing with the RW-GMapping algorithm perhaps somewhat unnecessarily complicated. As the rolling window grid map SLAM algorithm will not need to close large loops, there is no need for an advanced SLAM algorithm that aims at delivering high overall metric consistency. Unfortunately, setting GMapping to only use one particle is no proper option, as GMapping's particles pose estimates are exploratory. That is, GMapping consciously introduces randomness in a particles node estimates to generate a set of variate particles.

Another issue we found in GMapping is its poor performance when used with a laser range scanner with limited maximum range (see Appendix D). As we strive to use affordable hardware, this forms a notable issue. It would be better if the scan-matcher of GMapping was improved such that the score depends on the variety of line feature directions contained in the observations of the laser scanner (see Appendix D).

For real life applications, better obstacle avoidance is needed, as the current laser scanner overlooks many obstacles due to its flat field of sight. The Kinect of the TurtleBot can likely be incorporated in a straight forward way to improve this obstacle avoidance.

For navigation, the topological navigation can possibly be integrated with the `move_base` navigation package of ROS. The topological planner can possibly be designed to be used as a global planner plugin for `move_base`. This would reduce the overall complexity of topological navigation tasks, and could possibly overcome the 'stammering' movement issue currently seen in topological navigation.

Another approach would be to not use `move_base` at all, as it was originally not designed to be used as it is used in LEMTOMap (that is, as a companion to topological navigation that

turns a topological plan into metric navigation, operating on a sliding window grid map). Possibly, a modified fork of `move_base` is needed to provide better topological navigation performance and lower time and space complexity.

7-2 Beyond LEMTOMap-V1.0

Topological mapping In future, it could be investigated how the number of nodes could be limited. Perhaps, nodes can be pruned. Eventually, nodes should only remain at places of semantic significance, such as corners of corridors, at doors, or at places where objects were detected.

To increase the space described by the topological map to places that not only have been visited, but just have been observed, hypothetical nodes can be added to the map, similar to the placeholders described in [Pronobis and Jensfelt, 2012].

Topological mapping can be designed to include a probabilistic form of loop closing. In such a case, multiple hypotheses should be maintained. If a place is recognized that has a significant likeliness of being a topological loop closure, but not significant enough for the robot to know for sure, the robot can start keeping track of multiple hypotheses. It keeps track of the non-loop closing map and the loop closing map. If in subsequent travelling more evidence of the loop closing is found (e.g. the corridor is very similar, with doors at the same places and a corner in the corridor is found at the expected place), the robot can decide to make the loop closure definite. This is similar to how humans sometimes maintain multiple hypotheses about their localization.

For true, long term autonomy, it is important that a robot can robustly map its environment, robust localize itself in that map, and robustly navigate using that map. As first impressions/recordings of a place are not always valid for longer periods of time (due to inaccurate recordings or changes to the environment), the data captured by the map needs to be updated over time to provide a basis for true long term robot autonomy. Hence, LEMTOMap needs to be expanded to have an updating strategy for its topological graph. This is an interesting topic for future research.

Topological navigation For topological navigation, it can be considered to include temporary blocking of edges, as described in [Konolige et al., 2011]. The topological navigation described in that paper is capable of dealing with edges that are temporarily non-navigable (e.g. when a door is closed and the robot is not capable of opening a door). When a robot runs into a non-navigable edge during topological navigation, the edge is temporarily marked blocked (removed from the topological graph). Next, the topological path planner tries to find a new path and if it can find one, it executes it.

7-2-1 Combine with semantic mapping

LEMTOMap could eventually be combined with semantic mapping as provided by e.g. the semantic mapping system presented in [Pronobis and Jensfelt, 2012][1], or for example from the STRANDS project[2]. The STRANDS project is a collaboration between multiple universities and aims to develop intelligent robots that can operate autonomously in dynamic human environments for long periods of time (in the order of months). Several of the researchers involved in STRANDS were also involved in previous, prestigious semantic mapping and cognitive robotics projects such as CogX. The work of [Pronobis and Jensfelt, 2012] is an evolution of a part of the CogX project. During the CogX project, there has also been collaboration between the authors of [Pronobis and Jensfelt, 2012] and people involved in STRANDS. As STRANDS is in active development, available as open source on GitHub, and is developed for ROS, it is definitely a project to keep an eye on.

An integration between semantic mapping and LEMTOMap can prove beneficial to each others performance. LEMTOMap can enable better ability to perform on large scale for long periods of time, while semantic knowledge about the environment can for example aid topological loop closing[3]. After all, humans recognize that they are at some place from its observations at that place, much more than that they have kept track of their location and know that they are at the same place again based on that. In robotics, a shift towards this recognizance (semantics) based place recognition / loop closing would be an interesting topic for further research. Semantics can help to better understand and recognize places and objects (such as doors), which can be beneficial to the mapping, localization and navigation tasks of LEMTOMap. Using semantics, the robot can reason about unexplored space. Upon a rolling window shift of the local grid map, the robot could for example estimate what the new, unexplored space will be like.

7-3 Future implications of LEMTOMap in real life

In terms of real life implications of LEMTOMap, we would hope to see LEMTOMap to be applied in affordable care robots. An elderly care robot using (an evolution of) LEMTOMap can learn to know and understand the house of the person it is caring for. This could in term help solving the global ageing issue, and the shortages of people providing home and elderly care to the ones who need it.

[1] Pronobis announced to be working on a ROS version of it that will be released 'soon'

[2] See http://strands-project.eu/, and https://github.com/strands-project.

[3] A semantic map can for example provide a pruned search tree for topological loop closing; when the robot is in a corridor, only consider other corridors for loop closing.

Appendix A

Introduction to SLAM (Simultaneous Localization and Mapping)

A classical problem in the field of mobile robotics is the Simultaneous Localization and Mapping (SLAM) problem. It is considered one of the most important perceptual problems in mobile robotics. To navigate through an environment, a mobile robot usually requires a mapping of this environment, like a floor plan or a road map. It is considered an important skill for robots to build such a map autonomously, such that humans do not need to create and provide one a priori. Similarly, humans can learn to remember new environments like buildings or cities autonomously. Consider the mapping of an indoor environment by a robot, e.g. an office floor. Generally, mobile robots use laser range scanners to determine the perimeter of their direct environment, such as shown in figure A-1, but there are also robots that use a Kinect, stereo cameras, a plain camera or an integrated combination of those to map their environment.

To generate a full map of the office floor, the robot will also need to move around to explore more of it. This is where the mapping turns into a problem. Moving the robot around delivers new data, which needs to be merged with the old mapping to extend it. To properly merge measurements such as laser scans, the relative positions of where the scans are taken needs to be known, i.e. one needs to know the change in position of the robot. The position of the robot is generally called its *pose* and it consists of its location and orientation (x, y, and θ for a 2D map). When the exact change in location is known, the merging of the measurements is straightforward. However, the change in location needs to be estimated, for example using odometry. This pose estimation is called *localization*. If an exact mapping is known, much more accurate pose estimation would be possible by matching measurements with the mapping. Hence, estimating the joint probability distribution of the map and localization given measurements like laser range sensors and odometry is called the Simultaneous Localization and Mapping problem. The SLAM problem's main challenges lie in sensor noise, the correspondence problem, the high dimensionality of the problem, environment dynamics and the requirement of robotic exploration [Thrun, 2002]. The correspondence problem is about how to match data, e.g. aligning subsequent laser scans (which is often referred to as *scan-*

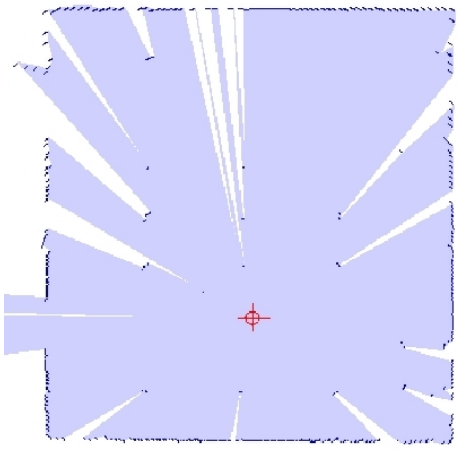
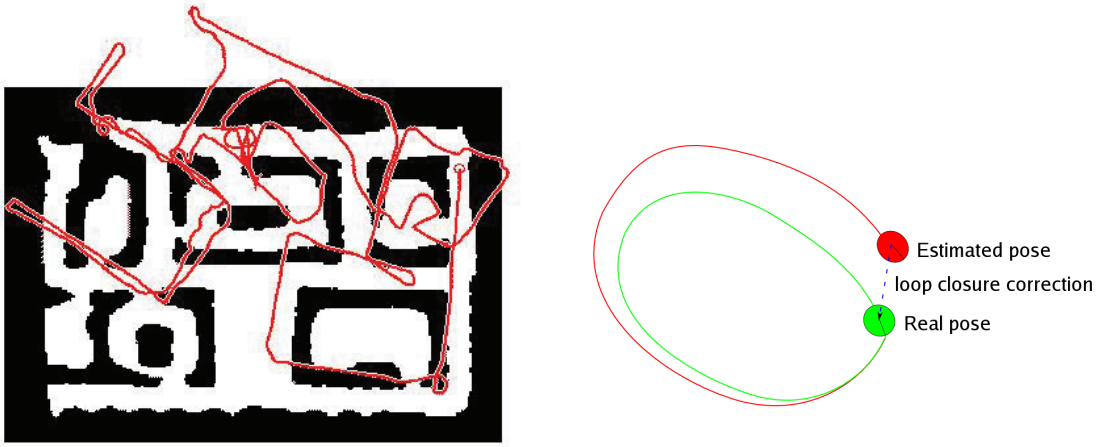


Figure A-1: Top view of the room perimeter measured using a 360° laser range scanner. The robots pose is depicted by the red crosshair. White is unknown space, blue is determined free space and black is where blockings are detected. Measurement noise and blocking objects like table legs cause the limited view at several points. This example shows a mostly closed room, possibly there are one or two doors at the left side of the room.

matching). As SLAM is generally an error minimization problem, large maps result a very high dimensional error minimizing problem, making it computationally intensive. To reduce the complexity of the problem, a static world is often assumed, which results in poor algorithm performance in case of environment dynamics (both short term, such as moving humans as long term dynamics such as moved furniture or seasonal changes). A whole different challenge is the challenge of how to let a robot explore the environment; using remote operation by a human (called *teleoperation*), or by autonomous exploration? Autonomous exploration is a whole different challenge which is often researched separately from SLAM. Sensor noise and the correspondence problem result in state estimation errors (the map and pose together are often regarded the state of the system). As Figure A-2a shows, a small odometry error in estimation the robots rotation can result in large pose estimation errors over time. Accumulating errors in pose and map estimation cause the loop closing problem, which is illustrated in Figure A-2b. The challenge is to recognize correspondence with enough certainty to close the loop, without mixing up plain similarity and true correspondence. GPS and digital compasses can limit the accumulation of such errors, but both have their drawbacks. GPS only works outdoor and has limited accuracy, while digital compasses are sensitive to disturbances by magnetic fields.

Although SLAM is defined as a problem, the abbreviation is also used to name the algorithms solving this problem. There are many kinds of algorithms; some come from the field of Augmented Reality (AR) such as [Klein and Murray, 2007, Newcombe et al., 2011b, Newcombe et al., 2011a], while most come from the field of mobile robotics (e.g. [Cheeseman et al., 1987, Montemerlo et al., 2002, Montemerlo et al., 2003, Grisetti et al., 2007, Grisetti et al., 2010, Kohlbrecher et al., 2011]). In AR, SLAM is generally used to draw augmentations such as objects in an image stream. For example, to draw and control a virtual action figure on a real desk, the 6 Degrees of Freedom (DOF) localization (x, y, z, $roll$, $pitch$, yaw) of the desk surface needs to be known, and the changes in position of the camera (e.g. a hand held



a b

Figure A-2: *(a)* Large localization errors can arise from small odometry errors, at the beginning a small rotational measurement error is made (lower right corner), resulting in big errors after translational movements. The black map represents the ground truth map. Figure is adopted from [Thrun, 2002]. *(b)* Localization errors cause the loop closing problem. If the correspondence is recognized, the accumulated localization and mapping errors can be corrected by back-propagating the correction. If errors get too big and correspondence is not recognized, it will inevitably result in faulty maps.

smartphone) need to be known to correctly draw that augmentation in the image stream. Therefore, AR is mostly focussed on generating 3D maps and 6DOF localization. This comes at the price of high computational complexity and thus map sizes are limited, which is often acceptable for AR: virtual objects often only need to be drawn in a limited field of view. For similar reasons, AR based SLAM is generally not suitable for loop closing. In mobile robotics, often 2D maps and 6DOF localizations are used, as they suffice and are less computationally expensive. A popular method to define the map is an *occupancy grid map* (often just called grid map). A 2D example is shown in Figure A-3. Occupancy grid maps are mostly popular for 2D maps, but 3D applications also exist. Occupancy grid maps are an example of a dense map: the full space is described. The detail of this description depends on de size of the grid cells, i.e. the resolution of the map. Sparse maps on the other hand, give a sparse description of space, for example by defining landmarks that represent objects, doors, lines and/or planes. Sparse maps generally contain less detail than dense maps, but their lower dimensionality are beneficial to the computational complexity.

For 2D maps, simple planar laser scanners suffice. For 3D mapping, 3D spatial structure can be measured using a Kinect, stereo cameras, spherical laser range scanners or by using Structure from Motion (SfM) algorithms when using a plain camera. Laser range scanners vary in their angle of view, some capture 180°, while some even do 360°.

Often, differences exist between SLAM algorithms depending on whether they are aimed at indoor or outdoor situations and whether they are aimed at land-based, airborne or underwater applications.

In this thesis, the focus will be on indoor, land-based robots using SLAM algorithms from the mobile robots field of research.

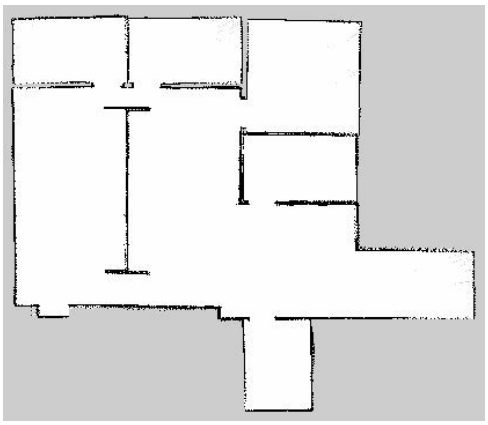


Figure A-3: An occupancy grid map models a map as a grid which cells, occupied, free, or unknown. Black is occupied, white is free and grey is unknown (unexplored). Figure is adopted from [Gallart del Burgo, 2013].

One of the most fundamental approaches to SLAM are based on Bayes filters, which will be explained in the next section.

A-1 Bayes filter based SLAM

SLAM is about estimating the joint probability of the map and pose based on measurements. The Bayes filter is a general probabilistic approach for estimating an unknown probability distribution in a recursive manner. Equation A-1 and A-2 show the general Bayes rule and the Bayes filter, which extends the Bayes rule to be used recursively, i.e. the probability distributions can be updated based on new information such as new measurements.

$$p(\mathbf{x}|\mathbf{z}) = \eta p(\mathbf{z}|\mathbf{x})p(\mathbf{x}) \qquad \text{(Bayes Rule)} \qquad \text{(A-1)}$$

$$p(\mathbf{x}_t|\mathbf{z}_t) = \underbrace{\underbrace{\eta p(\mathbf{z}_t|\mathbf{x}_t)}_{\text{update term}} \underbrace{\int p(\mathbf{x}_t|\mathbf{x}_{t-1})p(\mathbf{x}_{t-1}|\mathbf{z}_{t-1})d\mathbf{x}_{t-1}}_{\text{predict step}}}_{\text{update step}} \qquad \text{(Bayes Filter)} \qquad \text{(A-2)}$$

Here $\mathbf{x}$ is the state vector of the system, $\mathbf{z}$ are sensor measurements (such as laser range scans), η is a normalizer that is necessary to ensure that the left hand side of Bayes rule is indeed a valid probability distribution. The t subscript means that the value of the variable is evaluated at time instance t. $p(\mathbf{x}_t|\mathbf{z}_t)$ is a posterior probability, i.e. the chance that the state is $\mathbf{x}_t$ when you know $\mathbf{z}_t$. The filter first makes a prediction of what the next state will be ($p(\mathbf{x})$ in the normal Bayes rule), this is called the predict step. Then this initial prediction is updated using the measurements, the multiplication of this update term with the initial prediction is called the update step. Figure A-4 illustrates these steps. $p(\mathbf{z}_t|\mathbf{x}_t)$ and $p(\mathbf{x}_t|\mathbf{x}_{t-1})$ are called generative probabilities, as they are probabilities that generate a new probability $p(\mathbf{x}_t|\mathbf{z}_t)$ from the old one $p(\mathbf{x}_{t-1}|\mathbf{z}_{t-1})$. An important property of the Bayes filter is the Markov Assumption, which postulates that past and future data are independent if one knows the current state

$\mathbf{x}_t$[Thrun et al., 2005]. In other words: the prediction of the next state $\mathbf{x}_t$ only depends on data available at the previous state $\mathbf{x}_{t-1}$. As the estimation is recursive, the previous state $\mathbf{x}_{t-1}$ depends on the state before that: $\mathbf{x}_{t-2}$, and so on. Therefore, the conditional probability to know the current systems state is sometimes also defined $p(\mathbf{x}_t|\mathbf{z}_{1:t})$, where $\mathbf{z}_{1:t}$ is a short notation all $\mathbf{z}$ from 1 to t, that is[1]:

$$\mathbf{z}_{1:t} \equiv \{\mathbf{z}_1, \mathbf{z}_2, ..., \mathbf{z}_{t-1}, \mathbf{z}_t\} \tag{A-3}$$

Equation A-2 then becomes:

$$p(\mathbf{x}_t|\mathbf{z}_{1:t}) = \eta p(\mathbf{z}_t|\mathbf{x}_t) \int p(\mathbf{x}_t|\mathbf{x}_{t-1}) p(\mathbf{x}_{t-1}|\mathbf{z}_{1:t-1}) d\mathbf{x}_{t-1} \tag{A-4}$$

This notation might seem to be a violation of the Markov assumption, but it does not, as it only emphasizes that the current estimation is based on all previous estimations recursively and thus is indirectly based on all previous measurements. At each update step, however, only the measurements of the last step and the previous conditional probability are required, thereby honouring the Markov assumption. The $p(\mathbf{x}_t|\mathbf{z}_{1st})$ notation is used here to comply with other literature (e.g. [Thrun, 2002, Thrun et al., 2005, Stachniss, 2013]).

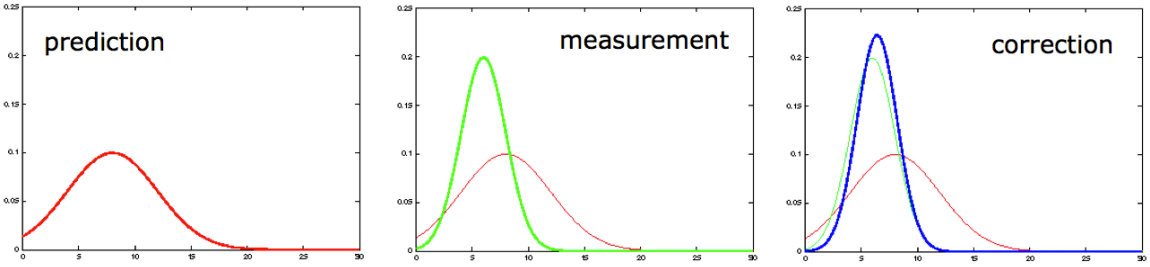


Figure A-4: The recursive steps of a Bayes filter. From left to right: first we predict the new state, then we measure it and finally we update our estimation by combining both probability distributions into one new, weighted probability distribution (here called correction). Figures adopted from [Stachniss, 2013].

When applying the Bayes filter to SLAM, Equation A-2 can be rewritten to Equation A-5[2].

$$p(\mathbf{s}_t, \mathbf{m}|\mathbf{z}_{1:t}, \mathbf{u}_{1:t}) = \eta p(\mathbf{z}_t|\mathbf{s}_t, \mathbf{m}) \int p(\mathbf{s}_t|\mathbf{u}_t, \mathbf{s}_{t-1}) p(\mathbf{s}_{t-1}, \mathbf{m}|\mathbf{z}_{1:t-1}, \mathbf{u}_{1:t-1}) d\mathbf{s}_{t-1} \tag{A-5}$$

$$bel(\mathbf{s}_t, \mathbf{m}) = \eta p(\mathbf{z}_t|\mathbf{s}_t, \mathbf{m}) \overline{bel}(\mathbf{s}_t, \mathbf{m}) \tag{A-6}$$

The systems state has been split in the pose $\mathbf{s}_t$ and the map $\mathbf{m}$, which is assumed to be static and thus lacks the t subscript. $\mathbf{u}$ represents the odometry information, while $\mathbf{z}$ covers the other sensor measurements. The conditional probability $p(\mathbf{s}_t, \mathbf{m}|\mathbf{z}_{1:t}, \mathbf{u}_{1:t})$ is also called

[1] In some literature a superscript is used instead of this notation, i.e. $\mathbf{x}_{1:t} \equiv \mathbf{x}^t$

[2] For the derivation of this rewrite, see [Thrun, 2002]

the systems belief about the map and pose $bel(\mathbf{s}_t, \mathbf{m})$. And the prediction of this belief $(\int p(\mathbf{s}_t|\mathbf{u}_t, \mathbf{s}_{t-1})p(\mathbf{s}_{t-1}, \mathbf{m}|\mathbf{z}_{1:t-1}, \mathbf{u}_{1:t-1})d\mathbf{s}_{t-1})$ is denoted $\overline{bel}(\mathbf{s}_t, \mathbf{m})$. There are two generative probabilities that need to be determined before the Bayes filter can be used. $p(\mathbf{z}_t|\mathbf{s}_t, \mathbf{m})$ is often called the *perceptual model* (also called observation model or measurement model), as it predicts what the next sensor measurements will be based on the current estimated pose and map. $p(\mathbf{s}_t|\mathbf{u}_t, \mathbf{s}_{t-1})$ is often called the *motion model* (also called transition model), as it predicts the new pose based on the control inputs and the old pose. A schematic representation of Bayes filter based SLAM is shown in Figure A-5. In some cases the complete history of poses is searched for. This is sometimes called the full SLAM problem and is show in Equation A-7.

$$p(\mathbf{s}_{0:t}, \mathbf{m}|\mathbf{z}_{1:t}, \mathbf{u}_{1:t}) = \eta p(\mathbf{z}_t|\mathbf{s}_t, \mathbf{m}) \int p(\mathbf{s}_t|\mathbf{u}_t, \mathbf{s}_{t-1}) p(\mathbf{s}_{0:t-1}, \mathbf{m}|\mathbf{z}_{1:t-1}, \mathbf{u}_{1:t-1}) d\mathbf{s}_{t-1} \quad \text{(A-7)}$$

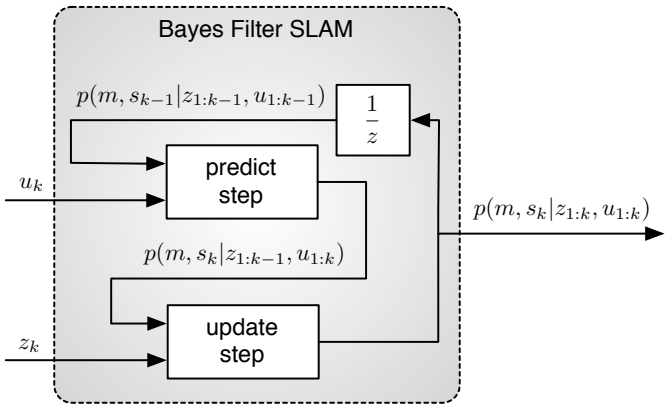


Figure A-5: Bayes Filter based SLAM model. $1/z$ means a 1 time step delay. The $1/z$ blocks can have some initial value for model initialization.

Bayes filters are a generic filter design, in practice more specific instantiations of the Bayes filters are required to actually implement the filter in a SLAM algorithm. Most well known are the Kalman filter (KF). SLAM algorithms are often based on KFs and variants like the Extended Kalman filter (EKF), Unscented Kalman filter (UKF), Information Filter (IF), Extended Information Filter (EIF), and Sparse Extended Information Filter (SEIF). Kalman filters cannot handle nonlinearities by default, which is partly solved by their Extended and Unscented variants. The Information variants use an information form instead of the normal canonical form to describe the system, which results in more efficient updates, but slower predictions. The Sparse EIF form reduces the EIFs complexity by sparsification. A schematic representation of KF based SLAM is shown in Figure A-6.

EKF-SLAM [Cheeseman et al., 1987] has been a very popular state of the art SLAM implementation for many years. EKFs uses a nonlinear system model combined with linearization to better deal with real world nonlinearities. The equations are shown in Equation A-8 to A-16.

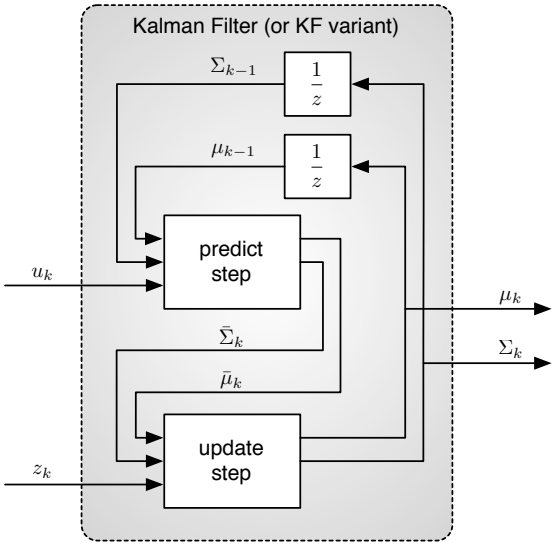


Figure A-6: Kalman Filter based SLAM model. This model also covers variants such as EKF, UKF, and IF variants (for IF variants μ and $\mathbf{\Sigma}$ should be replaced by ξ and $\mathbf{\Omega}$).

nonlinear model:

$$\mathbf{x}_t = g(\mathbf{u}_t, \mathbf{x}_{t-1}) + \epsilon_t \tag{A-8}$$

$$\mathbf{z}_t = h(\mathbf{x}_t) + \delta_t \tag{A-9}$$

predict step:

$$\bar{\mu}_t = g(\mathbf{u}_t, \mu_{t-1}) \tag{A-10}$$

$$\bar{\mathbf{\Sigma}}_t = \mathbf{G}_t \mathbf{\Sigma}_{t-1} \mathbf{G}_t^T + \mathbf{R}_t \tag{A-11}$$

update step:

$$\mathbf{K}_t = \bar{\mathbf{\Sigma}}_t \mathbf{H}_t^T \left(\mathbf{H}_t \bar{\mathbf{\Sigma}}_t \mathbf{H}_t^T + \mathbf{Q}_t \right)^{-1} \tag{A-12}$$

$$\mu_t = \bar{\mu}_t + \mathbf{K}_t \left(\mathbf{z}_t - h(\bar{\mu}_t) \right) \tag{A-13}$$

$$\mathbf{\Sigma}_t = \left(\mathbf{I} - \mathbf{K}_t \mathbf{H}_t \right) \bar{\mathbf{\Sigma}}_t \tag{A-14}$$

linearizations of g and h: (A-15)

$$\mathbf{G}_t = \frac{\partial g(\mathbf{u}_t, \mu_{t-1})}{\partial \mathbf{x}_{t-1}} \quad \mathbf{H}_t = \frac{\partial h(\bar{\mu}_{t-1})}{\partial \mathbf{x}_t} \tag{A-16}$$

The KF takes μ_{t-1}, $\mathbf{\Sigma}_{t-1}$, $\mathbf{z}_t$ and $\mathbf{u}_t$ as inputs, and has output μ_t and $\mathbf{\Sigma}_t$. μ is a vector that contains the estimated means of the system state, while $\mathbf{\Sigma}$ represents the accompanying covariance matrix. $\mathbf{Q}_t$ and $\mathbf{R}_t$ respectively describe the the measurement noise and the motion noise. $\mathbf{I}$ is the identity matrix. The state vector $\mathbf{x}_t$ combines the pose and map. For a 2D

mapping algorithm, usually the state vectors first 3 positions describe the pose (x, y, θ), complemented by an arbitrary number of (x, y) pairs of landmark positions that describe the map. ϵ_t and δ_t respectively represent the process noise (motion uncertainty) and measurement noise (measurement uncertainty). Also note that the matrices ($\mathbf{G}_t$, $\mathbf{H}_t$, etc.) and the nonlinear function g and h depend on the time step t, as they are updated as more information and more landmarks are added.

In addition to their limited capability to deal with nonlinearities, Kalman filters have another important drawback; they assume uncertainties to be Gaussian. The effects of this limitation is explained in the next section, which discusses Particle Filters as a solution to this problem.

A-2 Particle filters

Figure A-7 shows two examples of how a localization uncertainty develops over movement of a robot. Both examples start with a Guassian (i.e. bell-shaped) localization uncertainty. Translational movement results in a new Gaussian localization uncertainty, while a combination of rotation and translation results in a non-Gaussian localization uncertainty.

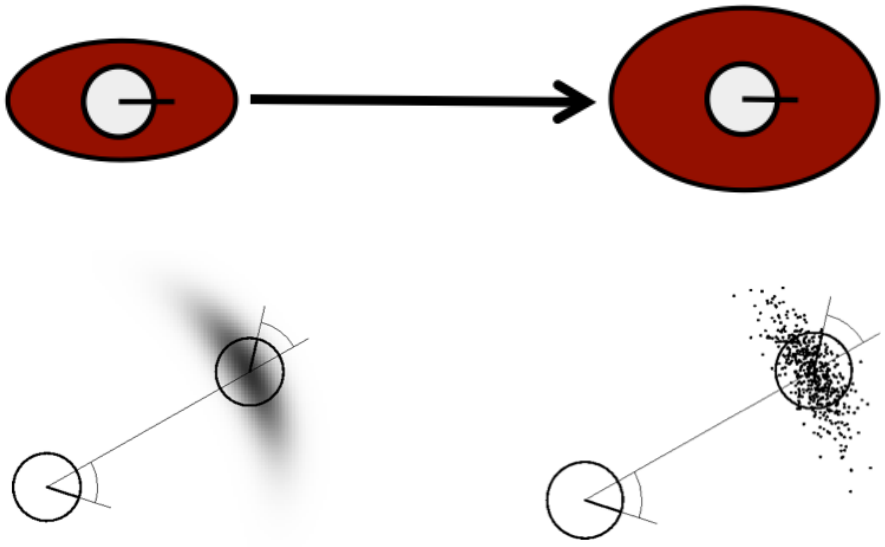


Figure A-7: *(Upper figure)* For plain translational motion Gaussian distributions suffice. The uncertainty is marked by the brown oval (think of it as the top view on a three dimensional uncertainty bell-shape). After movement, the uncertainty has grown, but can still be presented by the brown oval. *(Lower figure)* In this case, the robot rotates a bit counterclockwise, then moves forward and finally rotates a bit counterclockwise again. As a consequence of the rotational uncertainty, the final position uncertainty is not Gaussian any more. The right figure shows an example of poses one could get after a couple of experiments, i.e. samples. Both figures are adopted from [Stachniss, 2013].

Particle filters (PFs) do not use an algebraic function like a Gaussian to describe the uncertainty, but a collection of *particles* (*samples*) that together give a good representation of the

actual uncertainty. The more samples you draw, the more accurate the representation, but also the more computations are required to apply the filter.

A particle filter generates samples using a model (such as in Equation A-8, A-9), and initializes these samples by giving them a probability based on some initial proposal. These samples are then compared with a target distribution that follows from measurements, after which their weights will be updated. These steps are similar to the predict, measure and update steps in a Bayes filter. The steps are illustrated in Figure A-8. A set of N samples is defined by $\mathcal{X} = \left\{ \left\langle x^{[i]}, w^{[i]} \right\rangle \right\}_{i=1,\ldots,N}$. Where x are the sample values and w are the sample weights.

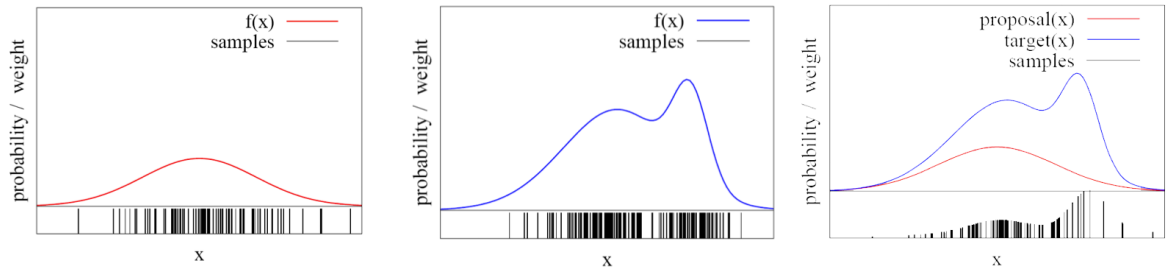


Figure A-8: *Left (prediction step)*: From this prediction samples are drawn in a random way. These samples are denoted by the stripes, which heights denote the sample probability. Please note that the 'wide' stripes are actually normal samples that are very close to each other. *Middle (measurement step)*: The distribution that follows from the measurement model, also with samples drawn. *Right (update step)*: The reweighed samples, that describe the final belief of the state. Figures adopted from [Stachniss, 2013].

For localization, each particle can be seen as a pose hypothesis. The left image in Figure A-9 shows a case where the map is already known, but the pose of the robot is unknown. Therefore, a set of samples is equally distributed among the space of possible poses (i.e. the proposed distribution is flat over the full space of possible poses). Based on odometry and measurements, the weights of the samples are updated. To make sure that the particle filter can deliver accuracy where needed, without requiring an enormous amount of samples, some of the less likely samples are *resampled*; they are reinitialized at more likely locations. This is shown in the right image of Figure A-9. At the same time, resampling can be used to make sure that the less likely places stay covered by at least some samples, to reduce the risk of settling on a suboptimal localization. This example illustrates another big advantage of Particle filters over Kalman based filters: they can represent multi-modal[3] probability distributions. That means that they can represent distributions with multiple 'peaks', i.e. multiple regions of likely hypotheses. Multi-modal distributions of pose estimates can (temporarily) develop when considering closing loops: two concentrations around pose hypothesis can exist for the scenario with and without the closed loop. They are also useful to solve the so called *kidnapped robot problem*: in which the robot experiences a large change in pose without having access to the odometry and measurement data recording this change.

An important advantages of particle filters is that they can handle nonlinear models and non-Gaussian uncertainty distributions with much more accuracy than EKFs (or KF variants),

[3]Multi-modal in the context of uncertainty distributions should not be confused with multi-modal sensor usage, as will be later used in the context of semantic mapping. The first is about probability distributions that can have multiple 'peaks' indicating points of increased likeliness, while the latter is about utilizing multiple sensory modalities.

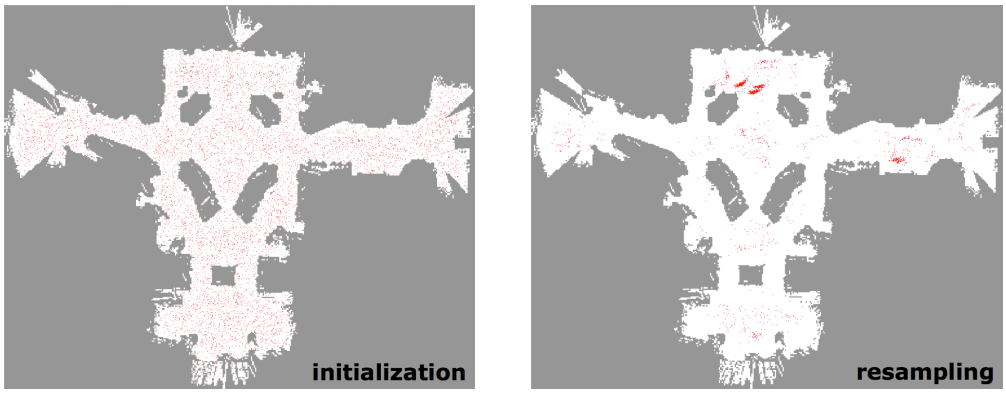


Figure A-9: *Left:* Initial sample distribution for determining robot location in a known map. Initially nothing is known about the robots location. *Right:* After a few motion updates and measurement updates from the robot, resampling could lead to such a sample distribution. Figures adopted from [Stachniss, 2013].

resulting in more accurate estimations $bel(\mathbf{x})$. A major disadvantage is that particle filters are not suitable for solving high dimensional problems. Therefore, they are not suitable for estimating maps, but are suitable for estimating poses. This advocates for using the PF for localization, while using some other filter (such as EKF) to deal with the mapping. This requires the SLAM problem to be split in a localization and a mapping part. This is done through a factorization method called Rao-Blackwellization. The factorization is shown in Equation A-18. Algorithms such as FastSLAM 1 and 2 [Montemerlo et al., 2002, Montemerlo et al., 2003] and GMapping [Grisetti et al., 2007] do this. GMapping has become very popular, as it can generate detailed occupancy grid maps, as opposed to FastSLAM which generates a sparse map of landmarks.

$$p(\mathbf{s}_{0:t}, \mathbf{m}|\mathbf{z}_{1:t}, \mathbf{u}_{1:t}) = p(\mathbf{s}_{0:t}, \mathbf{l}_{1:K}|\mathbf{z}_{1:t}, \mathbf{u}_{1:t}) \tag{A-17}$$

$$\begin{aligned} p(\mathbf{s}_{0:t}, \mathbf{l}_{1:K}|\mathbf{z}_{1:t}, \mathbf{u}_{1:t}) &= p(\mathbf{s}_{0:t}|\mathbf{z}_{1:t}, \mathbf{u}_{1:t}) p(\mathbf{l}_{1:K}|\mathbf{s}_{0:t}, \mathbf{z}_{1:t}) \\ &= p(\mathbf{s}_{0:t}|\mathbf{z}_{1:t}, \mathbf{u}_{1:t}) \prod_{i=1}^{K} p(\mathbf{l}_i|\mathbf{s}_{0:t}, \mathbf{z}_{1:t}) \end{aligned} \tag{A-18}$$

Equation A-17 rewrites the map of the full SLAM problem of Equation A-7 as a set of K landmarks $\mathbf{l}$. Equation A-18 splits this using Rao-Blackwellization in a localization part $p(\mathbf{s}_{0:t}|\mathbf{z}_{1:t}, \mathbf{u}_{1:t})$, which can be solved using a Particle filter, and a mapping part $p(\mathbf{l}_{1:K}|\mathbf{s}_{0:t}, \mathbf{z}_{1:t})$, which can be solved using for example an Extended Kalman filter.

A-3 Topological maps

Topological maps are made up out of *nodes* (sometimes called *vertices*) and *edges*. An example of a topological graph is a subway map, as shown in Figure A-10. The stations are the nodes and the connections in between are called edges. In the most general sense, a topology is a set of nodes, and a set of edges describing connectivity between those nodes. In the case

of a topological map, nodes define places and edges define direct *navigability* between those places, where direct navigability means that the user (e.g. robot) can directly navigate from one node to the other node without taking a route that leads along other nodes. Topological maps are not necessarily bound to Euclidean space; a topological map describing navigability of a two story parking garage could be projected to a $\mathbb{R}^2$ plane for example, as the edges preserve the right description of navigable connectivity (you can only get there by travelling via the nodes at the ramp that changes levels). Nodes could even completely lose any form of Euclidean coordinate, as long as the nodes and edges together describe properly how to get from one node to the other.

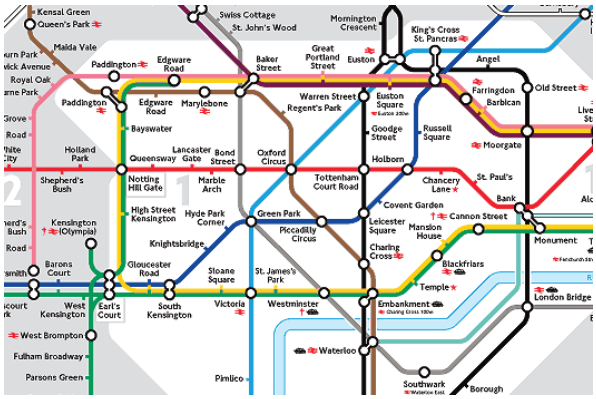


Figure A-10: A subway map is a common example of a topological map.

In SLAM, topologies are used in different ways, depending on the SLAM algorithm. Often, algorithms derive a topology from the metric map and localization (simultaneously or subsequently). The resulting maps are hybrid metric/topological maps. In these cases, the topology generation is actually an extra step after the actual SLAM process, instead of contributing to the SLAM process itself.

Pose graph based SLAM *Pose graph* based SLAM is an example where topologies are used in a constructive manner to the actual mapping process. A robot generates nodes at a predetermined distance interval (e.g. every meter) and adds spatial constraints between those nodes. The constrains specify what the distance between the nodes should be, together with the level of uncertainty for the constraint. In addition to these constraints that are based on odometry, constraints can also be added based on observations. Figure A-11a shows an example of such a pose graph. Figure A-11 b illustrates how observational constraints can be added. The resulting pose graph will generally have more constraints than can be satisfied. Periodically, pose graph optimization will be performed. Least square error minimization algorithms such as Levenberg-Marquardt or Gauss-Newton are used to calculate the node poses that minimize the constraint errors. An example of this is shown in Figure A-12. In pose graph based SLAM, generally first a graph is constructed in a front-end, which contains unaligned data. Periodically, this data is optimized by pose graph optimization in the back-end, which feeds the updated graph back to the front-end. Often, graph based SLAM uses a *lazy* SLAM front-end [Thrun et al., 2005]; it builds a map by accumulating all measurements along a trajectory that is only based on odometry. This is in contrast to *proactive* SLAM like Bayes filter or Particle filter based SLAM, which directly use measurements to correct the

localization and mapping. As *lazy* SLAM accumulates all data, much data overlaps other data. Consequently the maps grow much larger than proactively built maps in terms of required storage. Please note that the graph optimization in the back-end can greatly reduce the map size again periodically.

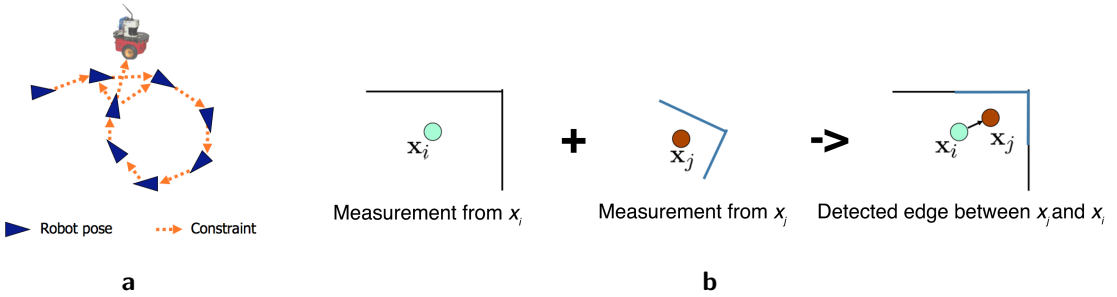


Figure A-11: *(a)* A pose graph. The poses are saved as *nodes*, which also contain observations. The *edges* are spatial constraints between these *nodes*. Edges are therefore either based on odometry or observations. Figure adopted from [Stachniss, 2013]. *(b)* Detect edges based on observations: Left is the environment observed from node $\mathbf{x}_i$, middle is the environment observed from node $\mathbf{x}_j$. By scan-matching both observations, the relative location of the nodes can be determined. A constraint is added together with its uncertainty (right). Figures adapted from [Stachniss, 2013].

Uses of topological maps Topological maps form an abstraction of full metric maps. They can for example be used to facilitate path planning for robotic navigation, of which more is discussed in Section 2-3. As discussed, they can also be used to facilitate the mapping process itself, as is the case in pose graph optimization.

Hierarchies and levels of abstraction Topological maps can be created at different levels of abstraction. This is illustrated in Figure A-13. The different levels of abstractions can be organized in a hierarchy. At the top you will find the most global abstraction, such as at the level of a university campus as shown in the left topological map of Figure A-13, which can be used for global path planning. After that, more detailed topological maps can be used to work out that plan into more detail. The right topological map in Figure A-13 shows nodes that were generated every meter or so, while the edges show whether there is a directly navigable connectivity between the nodes. The colors show how these nodes can be grouped to give the level of abstraction given in the middle graph.

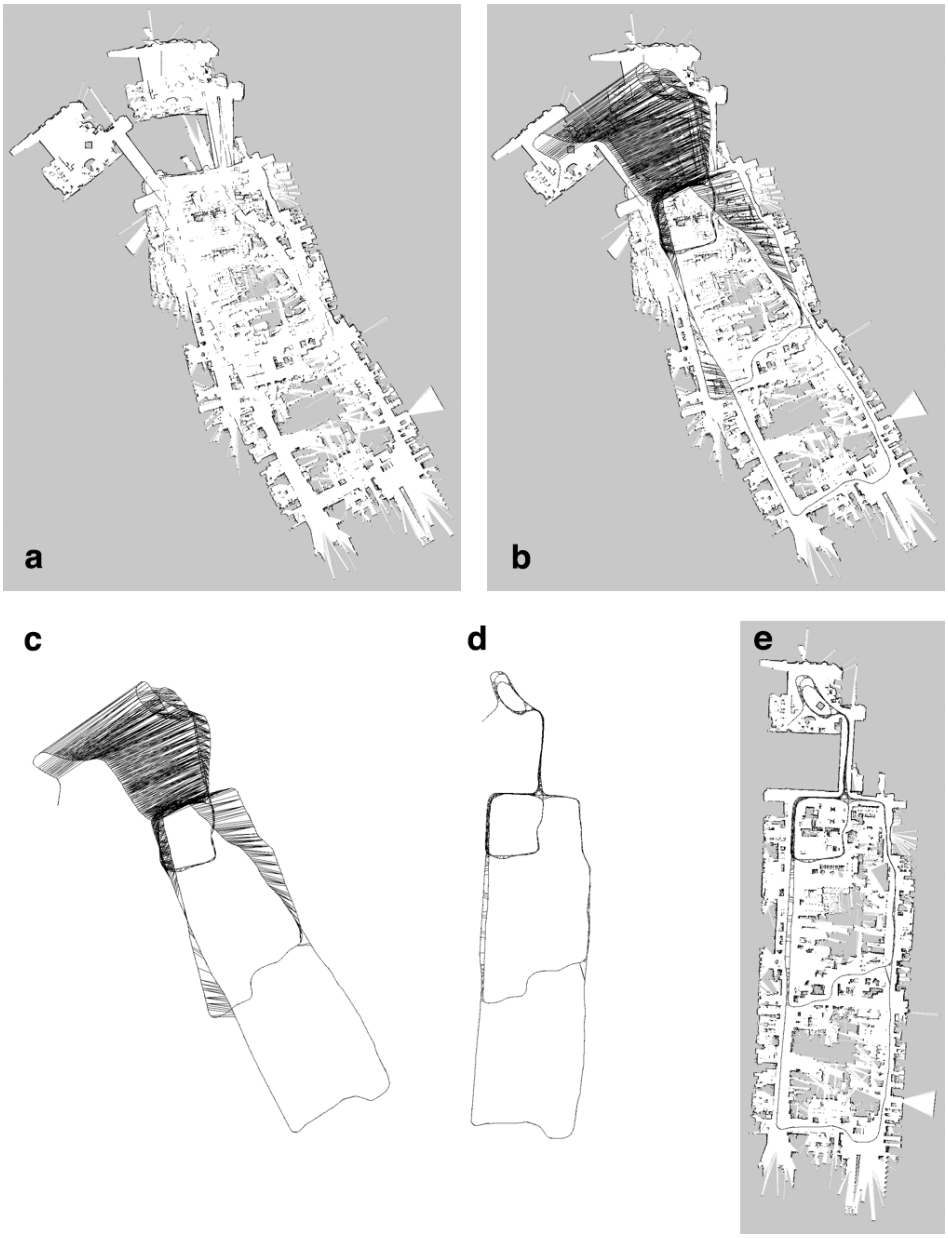


Figure A-12: *(a)* shows a map of unaligned grid map data that a robot has perceived during its trajectory. *(b)* shows its trajectory along with the constrains in this map. *(c)* shows the trajectory without the map. *(d)* shows the trajectory after minimizing constraint errors. *(e)* show the new map, which mapping data has aligned as a result of the constraint error minimization. Figures adapted from [Stachniss, 2013].

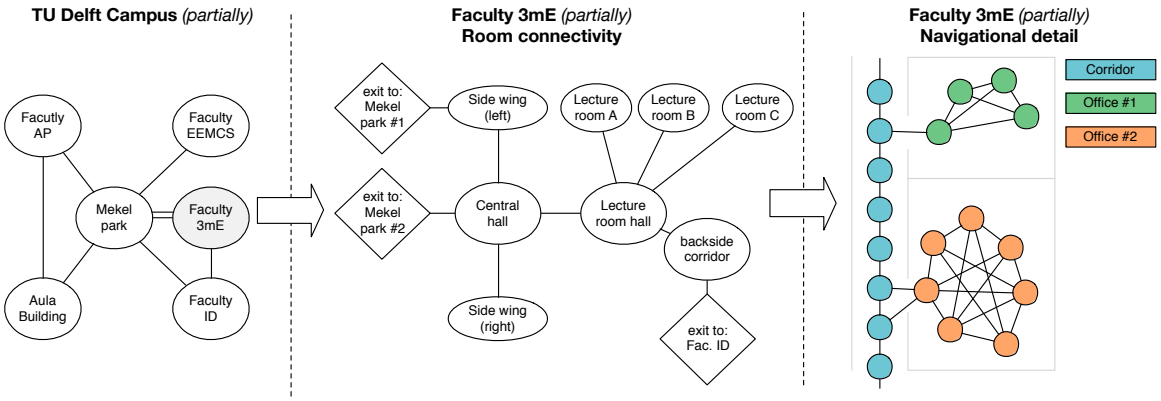


Figure A-13: *Left:* A topological map is used to describe connectivity between buildings at the Delft University of Technology campus. In this case a double edge shows that there are two connections between 3mE and the Mekelpark. *Middle:* A more detailed topological map describes connectivity of rooms in 3mE. *Right:* A more detailed topological map allows the robot to plan a path for navigation through the building. The gray walls are shown for the readers reference, but are not needed as part of the topological map: the edges define if a path between nodes is navigable.

Appendix B

Introduction to Semantic Mapping

Semantic mapping is a form of robotic mapping that aims to map the environment using semantics. Formally, *semantics* is the study of meaning. In the context of robotics, it generally means that human concepts are involved in the mapping. Higher level abstractions allow robots to use human concepts such as the notion of a *room*, a *door*, a *street*, and objects such as *books*, a *cup*, etc. Such abstractions allow for more natural human-robot interaction (HRI), but can also be used to enable overall more compact and more robust mapping and localization. When such an abstraction is descriptive enough to use it for robotic navigation, it can take away the need to store an (resource expensive) occupancy grid map of the environment. This can lead to a more compact mapping. Additionally, the awareness of the environment in terms of human concepts can give the robot a deeper understanding of the environment. This includes understanding that objects such as furniture can be new or moved when revisiting a place, while walls and doors are very unlikely to move or to be new. Therefore, an unexpected wall should trigger serious doubt about the localization, while moved furniture should not, or only to a lesser extend. Mapping in terms of concepts can also help to recognize places through seasonal changes. For example: trees will lose their leafs in the winter. With this knowledge, recognizing the same place throughout seasons becomes more feasible for a robot. Clearly, abstractions can increase a robots capability to deal with environmental dynamics. Also, the need for detailed quantifications largely disappears in semantic mapping. For example: Often, it is enough to know if an office is a *small*, *average-sized* or *large* office. The exact shape and color of a water tap do not matter, the robot should only know that it is a water tap, what it can be used for and how to use it. Also, the exact location of a room does not matter, as long as the robot knows how to get there.

Figure B-1 show an example illustration of a semantic map.

Semantic mapping is largely inspired on how humans deal with mapping their environment (more on this will be discussed in Section 2-5). It can potentially break with many of the existing characteristics of SLAM, such as the need to have a map that is globally consistent in metric space. Loop closing could be performed in semantic space only for example: by recognizing that two places have the same semantic abstraction. Semantics can be used to limit the size of the search tree (do not search for matches between an indoor place and

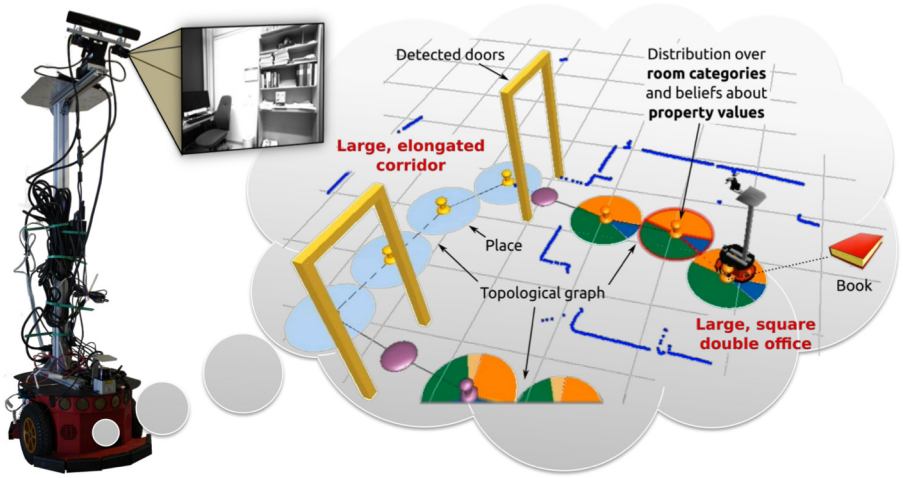


Figure B-1: An illustration of a semantic map generated using the semantic mapping system introduced in [Pronobis and Jensfelt, 2012]. Figure is adopted from [Pronobis and Jensfelt, 2012].

any outdoor places, or between a corridor and a large room, or between rooms at the 3mE TU Delft faculty and rooms at some other building). This is very much different from the geometric constraints that are usually applied in SLAM to search for correspondences (e.g. when searching for loop closures).

The deeper understanding that robots can get from semantic mapping can have many other advantages. Common-sense can help reasoning about the environment, in a similar way as humans do. This will be discussed later in this section about semantic mapping. Also, all kinds of properties can be learned over time: what is salient about a place, what is of semantic significance? A name plate on a door is small, but is of much more semantic significance that a cup on a desk. A table is less likely to move than a chair. When semantic mappings are available, robots can start learning these kind of properties to better understand and more robustly deal with their environment.

Semantic mapping is a relatively new field of research, and only a couple of years ago the interest in it has shown its first significant growth. Clear standards and definitions do not exist yet. Therefore, for this chapter, the work of Pronobis et al. as presented in [Pronobis and Jensfelt, 2012] will be mostly used as a base. We regard the work of Pronobis as defining the current state of the art in semantic mapping.

B-1 Properties of space

It has become clear that a semantic understanding of the environment can be very valuable for a robot. Now how can a robot determine such semantics? Semantic properties are not directly measurable; they are encoded in the environment. In [Pronobis and Jensfelt, 2012], the notion of *properties of space* is used to describe a set of measurable properties that can be used to find the semantic properties. Properties of space are for example the geometry of a room, appearance of a room, objects in a room, landmarks such as doors and walls, and the topology of a building floor for example. Geometry and appearance can also be considered

on other levels, such as at the level of objects. Geometry can be split in size and shape, e.g. a *large*, *square* room. Landmarks such as doors and walls can help segment space into *rooms*, the geometry and appearance of a room, and the objects inside a room can help determine the *room type*, such as a kitchen, corridor, office, or living room. Topologies can help the reasoning process as well: it is less likely that you will find a kitchen next to a kitchen and it is likely to find mostly offices along a corridor in an office building.

B-1-1 Measuring properties of space

It depends on the kind of property of space how it is measured. A rooms geometry can be determined using a laser range scanner, while its appearance can be measured using a plain camera. Objects can be 'measured' using any object detector, which in term are often based on local appearance, local features, and/or local geometry. The topology can be extracted from the room segmentations made, which in term can be based on landmarks as walls and doors. Walls and doors can be detected by laser range scanners, RGB-D cameras (cameras that assign a depth value to image pixels, such as a Kinect camera), and/or plain cameras. An important step towards abstraction is classification of such properties of space. Using classification, properties of space can be categorized in a limited set of semantic categories. For example, a room can be either *large*, *medium*, or *small*, its shape can be either *rectangular*, *square*, or *elongated*. An object should be classified as a *chair*, all cabinets as *cabinet*, and not simply *object1*, *object2*, etc. When looking at humans, humans do not know such properties exactly, hence the idea that such classification in categories can also be used in robotic semantic mapping. Additionally, humans do not know all properties for sure. Hence, in [Pronobis and Jensfelt, 2012] the idea is put forward to not just assign categories to instances of properties of space, but assign a probability distribution among the categories. In such a way, the robot can for example be 80% sure the room is large, 19% that it is medium, and 1% that it is small.

B-1-2 Integrating properties of space

The next step is to integrate properties of space to derive higher order concepts such as the room type. Such a *semantic label* for a room, as it is sometimes called, is based on different kinds of sensory data (geometry, appearance) that are integrated. This is referred to as *multi-modal* sensory data. More on the actual integration of this data is discussed in the next section.

B-2 Conceptual mapping and reasoning

Semantically categorized properties of space form a basic conceptual understanding of the space. These concepts can be related and integrated to derive more insights. For example, detecting a stove should increase the likeliness of the room being a kitchen, and detecting chairs make it likely that the room is for example a dining room or office, but is less likely to be a corridor. Also, when the robot knows that practically every room has a door, being in a room can make the robot reason that there must be a door somewhere without even having detected a door yet.

All the detected instances of concepts can be related with each other in a conceptual map. Relations can be either deterministic (e.g. a cup is an object) or probabilistic (e.g. a couch is 70% likely to be in a living room), directed (e.g. a cup is an object) or undirected (e.g. a office is x likely to be directly connected to another office). Relations can be predefined (e.g. a cup is an object), acquired directly (e.g. room1 is large), or inferred (e.g. this is a room, so there must be a door). Figure B-2 shows how this is integrated into the semantic mapping system of Pronobis et al. [Pronobis and Jensfelt, 2012]. Please note the difference between rectangles (concepts) and ovals (instances). The conceptual map is in the conceptual layer, while the sensory layer takes care of the sensor measurements, the place layer maintains a topological map, and the categorical layer contains categorization models to categorize the acquired properties of space.

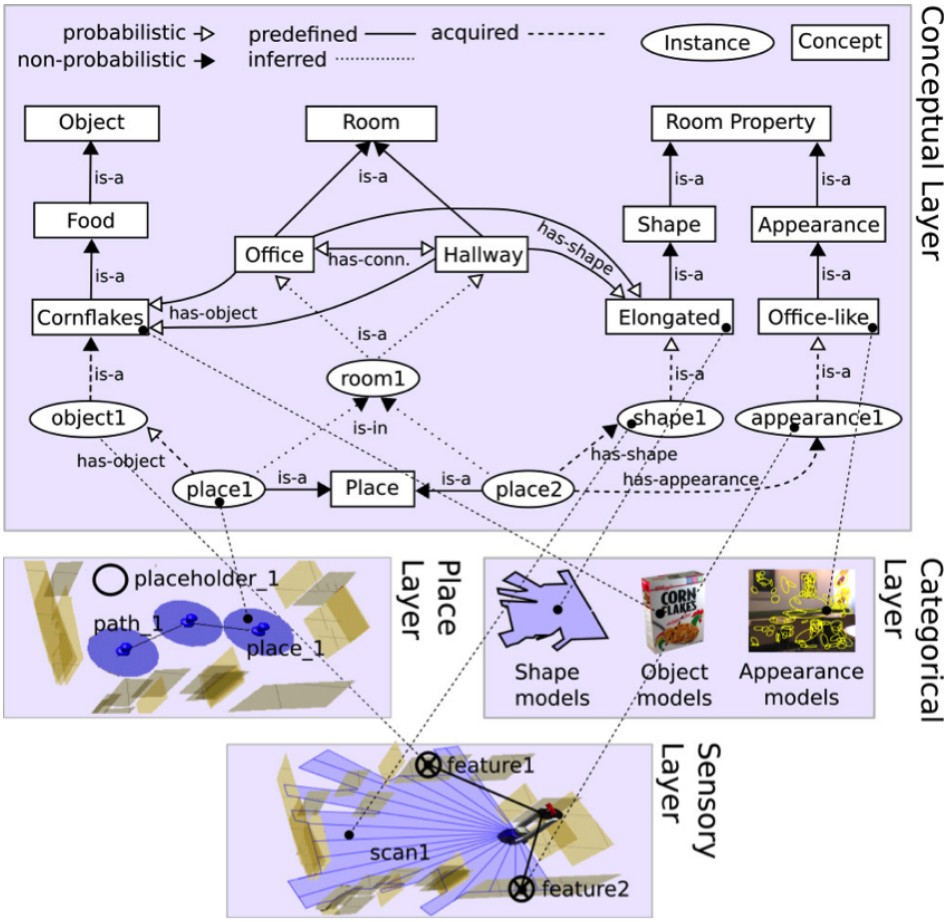


Figure B-2: The semantic mapping system presented in [Pronobis and Jensfelt, 2012], visualized as different layers that work together. Figure adopted from [Pronobis and Jensfelt, 2012].

The relations between the concepts shown in the rectangular boxes, together form a set of basic knowledge that the robot should have. Such relations are usually predefined. Such basic knowledge is like common-sense for humans. Ontologies are a common way to define such knowledge.

B-2-1 Ontologies

Ontologies define common-sense knowledge in the form of relations. An example of this is shown in Figure B-3. Such relations can defined to be deterministic or probabilistic. In Figure B-2, part of such an ontology is shown in the conceptual layer.

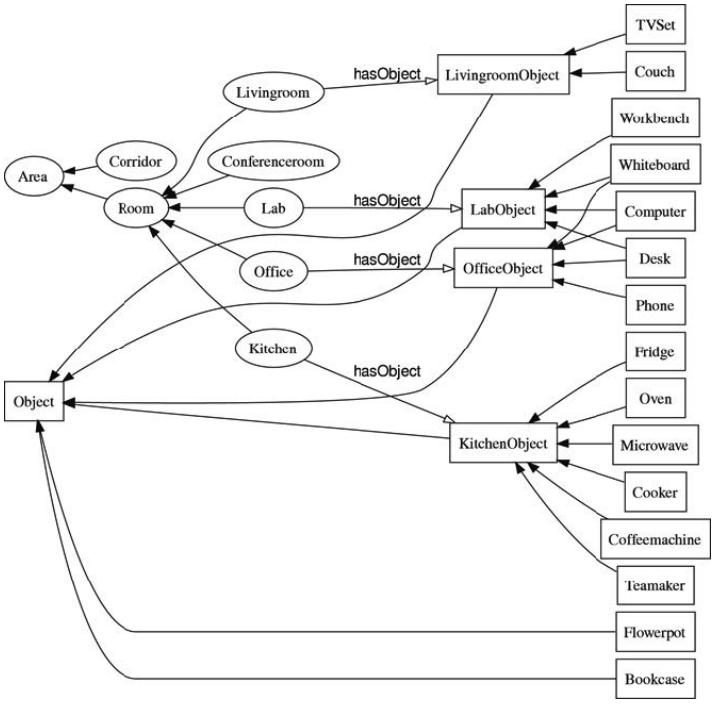


Figure B-3: An example of an ontology. In this figure blocks are objects and ellipses are areas. Please note that this is different from the use in other figures. Solid arrows show 'is-a' relations, open arrows show 'has-a' relations. Figure adopted from [Zender et al., 2008].

B-2-2 Common-sense reasoning

An important property of semantic mapping is that the awareness of concepts, combined with ontologies, allows the robot to mimic human common-sense reasoning. An example is for example if you ask a robot to cook pancakes. It can reason by itself that it can look up a recipe for cooking pancakes on the internet. Then it can reason by itself where to check in the kitchen for all the ingredients and execute the steps to make pancakes as described. The recipe should not need to contain detailed instructions such as 'go to the fridge, open it, grab the milk, put it on the counter. Now look for a bowl', etc. Also, a robot could try to answer the question: 'how many chairs do you see in this picture?' for Figure B-4, and might even understand what chairs can be regarded doubtful.

B-3 Probabilistic Graphical Models

The actual reasoning and inference is performed in the conceptual map, which is defined and solved as a Probabilistic Graphical Model (PGM). Different classes of PGMs exist, all

Figure B-4: "How many chairs are in this room?"; A question that can not be answered without common-sense. Figure adopted from [Bülthoff and Bülthoff, 2003].

with their own algorithms to solve them. Probabilistic Graphical Models exist of vertices (or nodes) and edges. The graph can thus be regarded to be a special kind of topology. Graphs that contain no loops at all are called *acyclic* graphs, while those that contain loops (that is, there is a vertex from which we can make a loop that ends at the same vertex again, without visiting any of the vertices multiple times) are called *cyclic graphs*. In PGMs a vertex v_i represents a stochastic variable X_i, so the set of vertices $\mathcal{V}$ represents the set of stochastic variables $\mathcal{X} = \{X_1, ..., X_N\}$. The edges $\mathcal{E}$ define the probabilistic interaction between the variables. Such edges can be either directed or undirected. The main types of PGMs are discussed below and shown in Figure B-5.

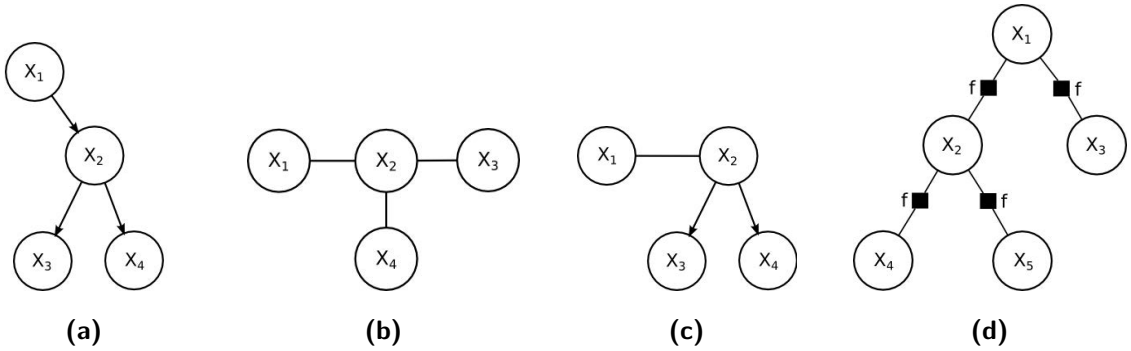


Figure B-5: *(a)* shows a directed graph, *(b)* an undirected graph, *(c)* a partially directed graph, and *(d)* a factor graph.

Directed Graphical Models Directed graphical models only contain directed edges, which represent conditional independence properties and causal relations. Edges point from parent to child. When they contain no cycles, these directed acyclic graphs are also called Bayesian networks.

Undirected Graphical Models Undirected Graphical Models are also called Markov random fields. Edges define associative relations and do not indicate causality. If the undirected graph is acyclic, it is sometimes also called a *tree*.

Partially Directed Graphical Models Partially directed graphs allow both directed and undirected edges. Edges can thus define causal, associative, and conditional independence relations. If there are no directed cycles, you call it a *chain graph*. The conceptual map from [Pronobis and Jensfelt, 2012] can be represented by such a chain graph.

Factor Graphs Factor graphs are used to perform belief propagation on PGMs. The PGMs discussed above can be converted to a Factor Graph, which can then be solved using the *belief propagation* algorithm. For graphs that contain cycles, *loopy belief propagation* can be used to approximate the solution of the cyclic factor graph.
Factor graphs exist of two types of vertices: stochastic variable vertices and factor vertices. Factor vertices define factors, which are functions that define the relations between the connected stochastic variables. In a factor graph, edges only define connectivity, while factors define the actual relation. Edges always connect nodes of different type (i.e. a stochastic variable always connects to a factor and vice versa).

Appendix C

Introduction to ROS (Robot Operating System)

Robot Operating System (ROS) is used in this thesis as the *robotic middleware*. It is used to develop and test implementations of the mapping system discussed in Chapter 4. According to the ROS website[1], "ROS is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behaviour across a wide variety of robotic platforms". ROS is a completely open-source, community driven project that is used by many universities, companies and hobbyists worldwide. ROS needs to run on top of an existing Operating System (OS), and is primarily targeted at Ubuntu. The ROS system is very modular; the minimum set of packages is formed by the ROS base installation. It can be extended by any ROS packages as desired. A common set of packages is captured in the ROS full desktop install, which contains of the base and a set of commonly used packages such as packages for simulation and visualization. In addition to the official Ubuntu releases, experimental releases exist for many other systems including OS X, Windows, Raspberry Pi, and embedded systems through OpenEmbedded. ROS can be run in a distributed way over multiple computers, communicating over TCP. Official support is also provided for several robots such as the PR2 and Turtlebot, which are controlled by a computer running Ubuntu.

ROS provides hardware abstraction, low-level device control, implementations of commonly used functionality, communication between processes, and package management. Packages provide functionality such as controlling movement of the robot, generating odometry information, reading and processing sensor data from e.g. a Kinect, camera or laser range sensor, keeping track of a robots joint configuration, etc. Packages can also provide more high level functionality such as SLAM implementations, object recognition, 3D simulation, compatibility layers to enable the use projects like Point Cloud Library (PCL) and Open Computer Vision (OpenCV), etc.

The basis for ROS was laid at Stanford University, where the robotic middleware Switchyard was developed in 2007. Switchyard was further developed under the name ROS by the company Willow Garage (a robotics research institute/incubator). ROS saw is first official release, ROS 1.0, in January 2010. Since then, many releases followed. For this thesis, the 8th official release named Indigo Igloo is used, released in July 2014.

[1] ROS.org

There are several alternatives to ROS. Similar to ROS is the CoSy Architecture Schema Toolkit (CAST) combined with the Boxes And Lines Toolkit (BALT), both introduced in [Hawes et al., 2007b][2]. These form an implementation of the architecture proposed by the CoSy Architecture Schema (CAS)[Hawes et al., 2007a]. The CAS Unified Robotics Environment (CURE) is a more high level set of tools on top of CAST, providing tools such as SLAM. The overall system provided by CAST, BALT and CURE combined was used in the CogX project, and in the semantic mapping system introduced in [Pronobis et al., 2010]. It was probably also used by [Pronobis and Jensfelt, 2012], as this has been developed in conjunction with the CogX[3] and CoSy[4] projects.
Other robotic middleware includes YARP, MIRO, MARIE and OpenRDK, of which only YARP seems to be still in active development.

In the last years, ROS has quickly grown in popularity and an increasing number of projects, companies, and universities seem to be using it. For example, the ambitious STRANDS project, which is in many ways a successor to the CogX project, is using it. Also, the ROCS toolkit by Pronobis which forms the basis of the work presented in [Pronobis and Jensfelt, 2012], is expected to be released soon for ROS[5].

ROS can use the Player/Stage, Gazebo and MORSE projects for (multi)robot simulation. Gazebo is capable of simulating full 3D worlds including physics and usually only a few adaptations are needed to run a real world ROS system in simulation.

C-1 ROS concepts

A good overview of the ROS concepts can be found on the ROS website[6]. ROS divides these concepts in three different levels: the ROS filesystem level, the ROS computation graph level, and the ROS community level.

C-1-1 ROS filesystem level

ROS uses various concepts at your local filesystem level, including:

Packages At the level of the computers local filesystem, ROS organizes functionality in *packages*. Packages contain files that together give the package functionality, such as executables (called ROS *nodes*), source code, cmake files, ROS *message* type definitions, ROS *service* type definitions, roslaunch files, configuration files, etc. Packages contain a manifest file (package.xml) that provide all the metadata about the package.

[2] This article also provides a good insight in the design challenges and trade-offs that are part of designing robotic middleware in general.
[3] http://cogx.eu
[4] http://www.cs.bham.ac.uk/research/projects/cosy
[5] Based on e-mail contact with Andrzej Pronobis. Please note that the code at is the code of a master students thesis and is only a basic partial implementation of ROCS for ROS; it does not provide the same functionality and quality as the system presented in [Pronobis and Jensfelt, 2012]
[6] wiki.ros.org/ROS/Concepts

Metapackages Metapackages are empty packages that only have dependencies on a set of other packages. Metapackages are used to collect a set of packages that together provide some functionality. By installing a metapackage, all relevant packages will get installed, similar to how this is sometimes done by apt-get.

Workspaces and overlaying ROS installs to your `/opt` folder. The Indigo release for example installs to `/opt/ros/indigo`. The ROS packages that are installed under this folder should not be edited by the end user. Instead, the end user should create a so called *workspace*, usually in your home folder. This workspace *overlays* the packages in `/opt/ros`: you can create your own packages here, add packages (manually or through a VCS) from other people, or check out original ROS packages from Github for example to make your own changes to it. An overlay, such as your workspace, has a higher precedence than everything that it overlays (e.g. `/opt/ros/indigo`). That is, you can install the `gmapping` package through apt-get (which will end under `opt`), and later add the Github version to your own workspace and make some changes to the code. ROS will now ignore the `gmapping` package under `opt`, as it gives priority to the version in your own workspace. You can even chain multiple workspaces using overlaying. Additionally, you can install multiple ROS releases at the same time. Switching releases or workspaces is as easy as sourcing a bash script (which you usually make part of your `.bashrc` file). Although multiple workspaces can be active at the same time, only one ROS release can be used at the same time.

Launch files An important kind of file are `.launch` files, which are executed by *roslaunch*. These XML based files specify ROS nodes (see next Section) that should be launched, together with parameters. The files can also include (import) other launch files, and can control many advanced settings such as remapping topics, interpreting passed arguments (through command line or launch file inclusions), etc. Launch files form a convenient way to start ROS functionality and are found in many packages to ease the use of the packages. By inclusion of other launch files, a launch file can be a very powerful tool that allows to start up a full, complex system.

C-1-2 ROS computation graph level

The ROS computation graph is the peer-to-peer network of processes that together process all data. Figure C-1 shows the structure of such a graph.

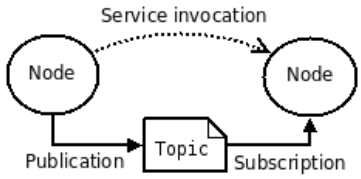


Figure C-1: Basic structure of a ROS computation graph

Master The ROS master is the process that governs the network and provides name registration and lookup to the rest of the processes.

Nodes Nodes[7] are the processes of the graph. The `gmapping` package for example contains a `gmapping` node (executable). Packages can contain multiple nodes. Nodes communicate with other nodes through messages and services, and can be configured using parameters. Nodes are generally programmed in C++ or Python. LISP is also official supported, while experimental support for Java and several others is available as well.

Parameters ROS works with global and private parameters. Private parameters are passed to nodes when they are launched and are only accessible to that node. Global parameters are stored in the parameter server and are accessible by all nodes at any time. Global parameter names are of the format `/<parent1>/<parent2>/<name>`, e.g. `/gmapping/map_update_interval`. A name can consist of zero or more parents.

Messages Messages are used for communication between nodes. Messages are simple data structures (similar to C structs), existing of typed fields. Messages can be exchanged through topics or services.

Topics Nodes can *publish* messages to topics. Other nodes can *subscribe* to such topics. A topic will always contain messages of only one type. Topics can receive messages from multiple publishing nodes and can have multiple subscribing nodes. Topics are named in the same way as global parameters. An example is a SLAM node that publishes an occupancy grid map as a message to the `/map` topic. A navigation node can subscribe to `/map` to use this map for planning a path.

Services Services work on a *request* / *reply* basis, whereas topics work on a *publish* / *subcribe* basis. A node (*called* the client) can send a request as a message to a service. The node that provides this service then responds with a reply message. Nodes that provide a service provide such a service under a name, which is named like a global parameter. Services always have only one node offering the service, and it can only serve one client at a time.

Other tools ROS can record message data and play it back later using the command line tool `rosbag`. This is especially useful for collecting real-world sensor measurements and playing that data back again later for development and testing.
Another important tool is the command line tool `rqt`, which offers a set of various GUIs. A very useful one is the *rqtgraph*, which generates a graphical representation of all the nodes and topics of the currently running system.
Lastly, the `rviz` node (which is part of the `rviz` package) can be used to visualize a variety of data. One can add all sorts of *windows* to enable visualization of certain types of data. For example, one can add a window showing the map generated by SLAM, a window that shows the robot, a window that shows laser scan data (reading `sensor_msgs/LaserScan` messages on the `/scan` topic), a window showing a planned path for navigation, etc. All windows are combined in one 2D or 3D view.

[7]ROS nodes should not be confused with topological nodes.

C-1-3 ROS community level

ROS is strongly driven by its community. The *wiki* provides a lot of documentation, including installation instructions, beginner tutorials and documentation for packages. ROS Answers (answers.ros.org) is a Q&A site for asking ROS related questions and can prove very useful from time to time. Furthermore, ROS provides all kinds of tools and documentation to stimulate contributions by the community. There are tools to ease the creation/generation of documentation, tools to generate package wiki pages, to automatically test and update package builds, there is a bug ticket system, there is a ROS Enhancements Proposals (REPs) system, there are tools and guidelines to use Github (the officially preferred VCS), etc.

C-2 ROS evolvement

ROS has been in active development since its initial release. The fast evolvement has brought many changes, including major changes that break existing code build on top of it. Recently, the switch to a new buildsystem has been carried out for example. The old buildsystem called *rosbuild* has been deprecated in favour of the new *catkin* buildsystem, which aims to be more conventional than rosbuild. Although both buildsystems can coexist and can be used in a mixed manner to a certain extend, using older rosbuild open source releases brings many challenges in practice. All official packages are currently fully *catkinized*[8], as the migration is dubbed. However, a large part of the packages is build and maintained by the community, and not all packages keep receiving maintenance. This has introduced difficulties in using older open source projects. The fast development of ROS has also resulted in other API changes that break code for older packages. Luckily, each new ROS release comes with a migration guide[9], which generally properly documents all necessary steps to update your own and non-official packages manually. Despite all efforts, the rapid changes result in documentation not always being up to date, including official documentation. This can turn out to be challenging from time to time. However, during the course of this thesis, many improvements and efforts were noticed to continue the professionalization. This is aimed at improving tutorials, documentation quality and API stability. Additionally, all official packages and many others have their own wiki.ros.org page which also refers to the repository where the code is hosted (Github for official packages). With some experience, reading this code can often give clarity about any undocumented changes. It can be expected that the ROS project will further continue to mature over the coming years.

[8] For more information, see wiki.ros.org/catkin/migratin_from_rosbuild and wiki.ros.org/catkin/conceptual_overview

[9] See for example the Hydro migration guide: wiki.ros.org/hydro/Migration

Appendix D

Issues encountered using GMapping with limited range laser scanners

GMapping performs remarkably poor when used with a relatively short range laser scanner (such as ours, at 5.6m maximum range). There are two main reasons for this, which are both related to the scan-matcher used by GMapping. Both issues also apply to RW-GMapping.

First of all, the shorter range scanners often do not see the end of the corridor or other features in the corridor. Hence, it frequently occurs that the robot sees very little features or even no features at all (large open spaces). In such occasions, the scan-matcher can often match the current measurements equally well for several different poses. For example: the robot is in a featureless corridor and measures two walls only. The scan matcher can match these scans equally well for hypothetical poses in a straight line trough that corridor. GMapping seems to simply pick the match with the highest score (the scan-matcher scores several samples of a estimated probability distribution for the pose). As a result, in featureless corridors or large open spaces, the pose estimate of the robot can make large jumps. In our simulated experiments, we have seen this to cause a serious decrease in the performance of GMapping; significant errors were introduced in the maps. To deal with this, in the original GMapping implementation (that is, the general implementation, not the ROS version) there is an option to set a minimum scan-matcher score. If the score of the scan-matcher is lower than this threshold, the new pose estimate will be based on the motion model (i.e. odometry) only, rather than on a combination of the motion model and sensor data (i.e. scan matcher). Scan-matcher scores are always positive and the default minimum score is 0, making the threshold ineffective in default configuration. This parameter, however, was initially not configurable in the ROS version of GMapping. I have made a patch to make this parameter configurable (through the `minimumScore` parameter), and this patch has currently been incorporated in the official ROS GMapping release (http://wiki.ros.org/gmapping). It is also already being used by the default TurtleBot navigation packages (http://wiki.ros.org/turtlebot_navigation/).

I belief that in addition to the `minimumScore` parameter, a better way of improving the performance for limited range scanners is possible. The scoring by the scan-matcher does not seem to correct for the diversity of the measurements. I would say that a scan match of a featureless

corridor should get a much lower score than a match of a corner of a corridor, because the first leaves us with great uncertainty in one direction. To further improve GMapping, I would suggest that a weighing factor is applied for the diversity of the scan data.

$$score = \mu \cdot score \tag{D-1}$$

This factor μ should range from e.g. 0 to 1, 0 for completely mono-directional, and 1 for multi-directional. By mono-directional I mean that data points of the scan measurement all align in one direction (like in a featureless corridor). Possibly, line detection can be applied tot the measurements and from this detection a line orientation histogram could be created. The diversity of the histogram determines the hight of μ.

Overall, further investigation of how the GMapping implementation exactly works is advisable, but from my practical experience I have the strong feeling that the suggested improvement can lead to a significant performance increase of one of today's most popular SLAM algorithms.

Appendix E

Pseudocode

E-1 GMapping

The GMapping algorithm is shown as pseudocode in Algorithm E-1[1]. Each particle (sample) s is defined as a tuple $s = (\mathbf{x}, w, \mathbf{M})$. The tuple contains a pose vector $\mathbf{x}$, a scalar weight w, and map matrix $\mathbf{M}$. The notation $s_t^{(i)}$ is used to denote the particle index i and the time step t. As GMapping is a 2D mapping algorithm, the pose $\mathbf{x}$ consists of x, y, and θ coordinates. The map is defined $\mathbf{M} \in \mathbb{R}^{m \times n}$, with values that define whether a grid cell is free (white), occupied (black), somewhere in between free and occupied (to cover uncertainty), or unknown (gray). The values 0, 100, and -1 are generally used for free, occupied and unknown. A laser scan measurement $\mathbf{z}$ is a vector, where each index in the vector represents a certain angle from the full range of angles that the sensor scans, and the value represents the measured distance at that angle. The odometry $\mathbf{u}$ is defined as changes in coordinate x, y, and θ, relative to the local coordinate frame of the robot (with x pointing forward, y left, z up). The pose compounding operator $\oplus$, as described in [Lu and Milios, 1997], performs adding such an odometry vector to a pose (their elements cannot be summed directly, as they are defined in different frames). To find the full path of the particle, one simply needs to collect all historical poses $\mathbf{x}$ from particle $s^{(i)}$. To control the risk of particle depletion (the risk of resampling decent particles, and thereby losing their 'memory'), resampling is only performed when N_{eff} drops below a threshold T_{resamp}. Normally $T_{resamp} = N/2$, where N is the total number of particles. N_{eff} is a measure for how well the overall accuracy of the current set of particles is. As each particle can be regarded a hypothesis of the robot pose, particles can handle multi-modal hypotheses. Two or more different likely pose hypotheses can exist when closing a loop for example. After a loop is closed, N_{eff} generally drops significantly. This is because only the few particles that were able to close the loop stay useful, while the others tend to get lost (due to problems arising in scan-matching). The drop of N_{eff} triggers resampling, which delivers a fresh, useful set of particles. The pose sampling based on combined odometry and measurements, which

[1]Please note that $\mathbf{u}_{t-1}$ is used in some literature (including [Grisetti et al., 2007]) to denote the most recent odometry. In this thesis, $\mathbf{u}_t$ is used.

is summarized by line 14 and 15 here, is one of the main novelties in GMapping and is more extensively described in [Grisetti et al., 2007].

Algorithm E-1. Default ROS implementation of GMapping SLAM algorithm, adapted from [Grisetti et al., 2007]

Require: (once)

T_{dist}, T_{angle}, T_{resamp}, thresholds

N, number of particles

Several other parameters, such as the initial map size, parameters for the scan-matcher, etc.

Require: (every cycle)

$\mathbf{z}_t$, the most recent laser scan

$\mathbf{u}_t$, the most recent odometry measurement

Ensure: $\mathcal{S}_t$, the new set of samples (contains all particles, and thus also the most likely localization and map)

1: **loop**
2: $\langle u_{x,t}, u_{y,t}, u_{\theta,t} \rangle \leftarrow \mathbf{u}_t$
3: **if** $\left(\sqrt{u_{x,t}^2 + u_{y,t}^2} > T_{dist} \;||\; u_{\theta,t} > T_{angle}\right)$ **then**
4: $\mathcal{S}_t = \{\}$
5: **for all** $s_{t-1}^{(i)} \in \mathcal{S}_{t-1}$ **do**
6: $\left\langle \mathbf{x}_{t-1}^{(i)}, w_{t-1}^{(i)}, \mathbf{M}_{t-1}^{(i)} \right\rangle \leftarrow s_{t-1}^{(i)}$
7: $\mathbf{x}_t'^{(i)} = \mathbf{x}_{t-1}^{(i)} \oplus \mathbf{u}_t$
8: // scan-matching
9: $\hat{\mathbf{x}}_t^{(i)} = \text{argmax}_{\mathbf{x}}\, p(\mathbf{x} \mid \mathbf{M}_{t-1}^{(i)}, \mathbf{z}_t, \mathbf{x}_t'^{(i)})$

Algorithm E-1. Default ROS implementation of GMapping SLAM algorithm (continued)

10: **if** $\hat{\mathbf{x}}_t^{(i)} = \text{failure}$ **then**
11: $\mathbf{x}_t^{(i)} \sim p(\mathbf{x}_t \mid \mathbf{x}_{t-1}^{(i)}, \mathbf{u}_t)$
12: $w_t^{(i)} = w_{t-1}^{(i)} \cdot p(\mathbf{z}_t \mid \mathbf{M}_{t-1}^{(i)}, \mathbf{x}_t^{(i)})$
13: **else**
14: $\mathbf{x}_t^{(i)} \sim p(\mathbf{x}_t \mid \mathbf{M}_{t-1}^{(i)}, \mathbf{x}_{t-1}^{(i)}, \mathbf{z}_t, \mathbf{u}_t)$
15: $w_t^{(i)} = w_{t-1}^{(i)} \cdot p(\mathbf{z}_t \mid \mathbf{M}_{t-1}^{(i)}, \mathbf{x}_{t-1}^{(i)}, \mathbf{u}_t)$
16: **end if**
17: // update map
18: $\mathbf{M}_t^{(i)} = \text{integrateScan}(\mathbf{M}_{t-1}^{(i)}, \mathbf{x}_t^{(i)}, \mathbf{z}_t)$
19: // update sample set
20: $s_t^{(i)} \leftarrow \left\langle \mathbf{x}_t^{(i)}, w_t^{(i)}, \mathbf{M}_t^{(i)} \right\rangle$
21: $\mathcal{S}_t = \mathcal{S}_t \cup s_t^{(i)}$
22: **end for**
23: // resample if needed
24: $N_{eff} = \frac{1}{\sum_{i=1}^{N} \left(\tilde{w}_t^{(i)} \right)^2}$
25: **if** $N_{eff} < T_{resamp}$ **then**
26: $\mathcal{S}_t = \text{resample}\,(\mathcal{S}_t)$
27: **end if**
28: $t = t + 1$
29: **end if**
30: **end loop**

The algorithm can be summarized by the following steps. When applicable, steps are performed for each particle i.

1. Initialize the algorithm. All particle poses initialize at $[0, 0, 0]^T$, and the occupancy grid map is initialized at -1, i.e. unknown. The initial size of the map is an input requirement and should be specified in meters. *(Pseudocode line* **??-??***)*

2. Run the update loop of the algorithm every time the pose has changed more than a certain angle (default 25°) or more than a certain distance (default $0.5m$) since last update *(Line 3)*

3. Based on the odometry $\mathbf{u}_t$, make an initial guess of the new pose $\mathbf{x}_t'^{(i)}$. *(Line 7)*

4. Scan-matching is performed on $\mathbf{M}_{t-1}^{(i)}$, starting from $\mathbf{x}_t'^{(i)}$ and in a limited region around $\mathbf{x}_t'^{(i)}$. Based on scan-matching a new estimate $\hat{\mathbf{x}}_t^{(i)}$ is generated *(line 9)*. If scan-matching fails, sample a new pose and weight based on odometry only (similar to FastSLAM [Montemerlo et al., 2003]) and continue at step 6 *(line 10-12)*.

5. A new pose estimate is drawn from the probability distribution based on odometry, measurements, the previous pose, and the previous map. This distribution involves

generating a Gaussian distribution based on samples drawn in a limited region around the scan-matching estimate $\hat{\mathbf{x}}_t^{(i)}$. The particle weight $w_t^{(i)}$ is updated accordingly. This step is described in more detail in [Grisetti et al., 2007]. *(Line 14-15)*

6. Using a scan-integrator, calculate the new map $\mathbf{M}_t^{(i)}$ based on $\mathbf{M}_{t-1}^{(i)}$, $\mathbf{x}_t^{(i)}$, and $\mathbf{z}_t$. *(Line 18)*

7. Perform resampling if N_{eff} drops below the threshold T_{resamp}. For calculating N_{eff}, the particle weights $w_t^{(i)}$ are first normalized. The normalized weights are denoted $\tilde{w}_t^{(i)}$. The resampling is perf *(Line 23-27)*. Continue at step 2.

The function `integrateScan` adds the new scan to the map (assuming known poses). This function will also take care of resizing the map matrix $\mathbf{M}$. If a scan reaches outside the area currently covered by $\mathbf{M}$, the matrix will be increased in size to fit the measurements. All cells not covered by the scan will be initialized as unknown (-1).

E-2 Dijkstra's Algorithm

By default, Dijkstra's algorithm finds the shortest paths from the source vertex to any arbitrary other vertex in the graph. When one is only interested in the shortest path to a specific target vertex, the algorithm can be stopped as soon as it reaches line 10 with u equal to the target vertex.

Algorithm E-2. Dijkstra's shortest path algorithm

Require: A source vertex $v_{src} \in \mathcal{V}$, and a directed graph $g = (\mathcal{V}, \mathcal{E}, \mathcal{W})$, with a set of vertices $\mathcal{V}$, edges $\mathcal{E}$, and non-negative edge weights $\mathcal{W}$.

```
1: function DIJKSTRA(g, v_src)
2:     for each v ∈ V do
3:         dist[v] := ∞
4:         previous[v] := undefined
5:     end for
6:     dist[v_src] := 0
7:     Q := V
8:     while Q ≠ {} do
9:         u ← v ∈ Q with smallest distance in dist[]
10:        remove u from Q
11:        if dist[u] = ∞ then
12:            break
13:        end if
14:        for each neighbour v of u do
15:            tentative_dist := dist[u] + dist_between(u, v)   ▷ dist_between is defined by the weight w of the edge e corresponding with vertices u, v.
16:            if tentative_dist < dist[v] then
17:                dist[v] := tentative_dist
18:                previous[v] := u
19:            end if
20:        end for
21:    end while
22:    return dist[], previous[]
23: end function
```

E-3 A* shortest path algorithm

Algorithm E-3. A* shortest path algorithm

Require: A source vertex $v_{src} \in \mathcal{V}$, a destination vertex $v_{goal} \in \mathcal{V}$ and a graph $g = (\mathcal{V}, \mathcal{E}, \mathcal{W})$, with a set of vertices $\mathcal{V}$, edges $\mathcal{E}$, and non-negative edge weights $\mathcal{W}$. h is the heuristic function.

1: **function** ASTAR(g, v_{src}, v_{goal}, h)
2: $\mathcal{C} := \{\}$ ▷ Closed set $\mathcal{C}$ is the set of evaluated vertices
3: $\mathcal{Q} := v_{src}$ ▷ Set of tentative vertices to be evaluated, initially containing v_{src} only
4: $previous[] := undefined$
5: $dist[v_{src}] := 0$
6: $f[v_{src}] := dist[v_{src}] + h(v_{src}, v_{goal})$
7: **while** $\mathcal{Q} \neq \{\}$ **do**
8: $u \leftarrow v \in \mathcal{Q}$ with the lowest value in $f[]$
9: **if** $u = v_{goal}$ **then**
10: **return** $dist[], previous[]$
11: **end if**
12: remove u from $\mathcal{Q}$
13: add u to $\mathcal{C}$
14: **for** each neighbour v of u **do**
15: **if** $v \in \mathcal{C}$ **then**
16: **continue**
17: **end if**
18: $tentative\_dist := dist[u] + dist\_between(u, v)$ ▷ $dist\_between$ is defined by the weight w of the edge e corresponding with vertices u, v.
19: **if** $v \notin \mathcal{Q}$ or $tentative\_dist < dist[v]$ **then**
20: $previous[v] := u$
21: $dist[v] := tentative\_dist$
22: $f[v] := dist[v] + h(v, v_{goal})$
23: **if** $v \notin \mathcal{Q}$ **then**
24: add v to $\mathcal{Q}$
25: **end if**
26: **end if**
27: **end for**
28: **end while**
29: **return** failure
30: **end function**

E-4 RW-GMapping

Please refer to the description of the normal GMapping algorithm (Section E-1) for a general explanation of all the parameters.

When a new scan is integrated into a particles map (line 18), the `integrateScan` function can cause the map to grow in size. This will happen when a scan falls partially or fully outside of the region covered by the old map $\mathbf{M}_{t-1}^{(i)}$ of the particle. The `integrateScan` function, supplied by the original GMapping implementation, will automatically take care of this and will initialize all other unknown cells of the grown map as unknown. If it is detected that the map has grown in size after performing `integrateScan`, the `resizeMap` function is triggered. The robot pose according to the particle will be used as the middle point of the map and the map will be resized to a size of W by W meter with the particle's estimated robot pose as the origin. Again any unknown cells are initialized as unknown. As each particle has its own pose estimate and own map, map resizes need to be performed per particle. It is possible that some particles do not need to resize while others do. The functions `height` and `width` are used to get the height and width of the map (in meters). The `resizeMap` function will always resize the map by an integer amount of grid cell columns/rows, i.e. an integer amount of columns and/or rows is removed and an integer amount of rows and/or columns is added. This way, the values of grid cells never need to be interpolated as they always can simply shift a certain number of rows and/or columns in the matrix. As a result, the window is generally not exactly covering x_{min} to x_{max} and y_{min} to y_{max}, and the robot is generally not exactly in the middle of the window right after a shift.

Algorithm E-4. RW-GMapping algorithm

Require: (once)
T_{dist}, T_{angle}, T_{resamp}, thresholds
N, number of particles
W, size of the rolling window in meters
Several other parameters, including parameters for the scan-matcher

Require: (every cycle)
$\mathbf{z}_t$, the most recent laser scan
$\mathbf{u}_t$, the most recent odometry measurement

Ensure: $\mathcal{S}_t$, the new set of samples (contains all particles, and thus also the most likely localization and map)

1: **loop**
2: $\langle u_{x,t}, u_{y,t}, u_{\theta,t} \rangle \leftarrow \mathbf{u}_t$
3: **if** $\left(\sqrt{u_{x,t}^2 + u_{y,t}^2} > T_{dist} \;||\; u_{\theta,t} > T_{angle}\right)$ **then**
4: $\mathcal{S}_t = \{\}$
5: **for all** $s_{t-1}^{(i)} \in \mathcal{S}_{t-1}$ **do**
6: $\left\langle \mathbf{x}_{t-1}^{(i)}, w_{t-1}^{(i)}, \mathbf{M}_{t-1}^{(i)} \right\rangle \leftarrow s_{t-1}^{(i)}$
7: $\mathbf{x}_t'^{(i)} = \mathbf{x}_{t-1}^{(i)} \oplus \mathbf{u}_t$
8: // scan-matching
9: $\hat{\mathbf{x}}_t^{(i)} = \text{argmax}_{\mathbf{x}}\, p(\mathbf{x} \mid \mathbf{M}_{t-1}^{(i)}, \mathbf{z}_t, \mathbf{x}_t'^{(i)})$

Algorithm E-4. RW-GMapping algorithm (continued)

10: **if** $\hat{\mathbf{x}}_t^{(i)} = \text{failure}$ **then**
11: $\mathbf{x}_t^{(i)} \sim p(\mathbf{x}_t \mid \mathbf{x}_{t-1}^{(i)}, \mathbf{u}_t)$
12: $w_t^{(i)} = w_{t-1}^{(i)} \cdot p(\mathbf{z}_t \mid \mathbf{M}_{t-1}^{(i)}, \mathbf{x}_t^{(i)})$
13: **else**
14: $\mathbf{x}_t^{(i)} \sim p(\mathbf{x}_t \mid \mathbf{M}_{t-1}^{(i)}, \mathbf{x}_{t-1}^{(i)}, \mathbf{z}_t, \mathbf{u}_t)$
15: $w_t^{(i)} = w_{t-1}^{(i)} \cdot p(\mathbf{z}_t \mid \mathbf{M}_{t-1}^{(i)}, \mathbf{x}_{t-1}^{(i)}, \mathbf{u}_t)$
16: **end if**
17: // update map
18: $\mathbf{M}_t^{(i)} = \text{integrateScan}(\mathbf{M}_{t-1}^{(i)}, \mathbf{x}_t^{(i)}, \mathbf{z}_t)$
19: // resize map if needed
20: **if** $\left(\text{height}(\mathbf{M}_t^{(i)}) > W \;||\; \text{width}(\mathbf{M}_t^{(i)}) > W\right)$ **then**
21: $\left\langle \mathbf{x}_{x,t}^{(i)}, \mathbf{x}_{y,t}^{(i)}, \mathbf{x}_{\theta,t}^{(i)} \right\rangle \leftarrow \mathbf{x}_t^{(i)}$
22: $x_{min} = -W/2 + \mathbf{x}_{x,t}^{(i)}$
23: $y_{min} = -W/2 + \mathbf{x}_{y,t}^{(i)}$
24: $x_{max} = W/2 + \mathbf{x}_{x,t}^{(i)}$
25: $y_{max} = W/2 + \mathbf{x}_{y,t}^{(i)}$
26: $\mathbf{M}_t^{(i)} = \text{resizeMap}(\mathbf{M}_t^{(i)}, x_{min}, y_{min}, x_{max}, y_{max})$
27: **end if**
28: // update sample set
29: $s_t^{(i)} \leftarrow \left\langle \mathbf{x}_t^{(i)}, w_t^{(i)}, \mathbf{M}_t^{(i)} \right\rangle$
30: $\mathcal{S}_t = \mathcal{S}_t \cup s_t^{(i)}$
31: **end for**
32: // resample if needed
33: $N_{eff} = \dfrac{1}{\sum_{i=1}^{N} \left(\tilde{w}_t^{(i)}\right)^2}$
34: **if** $N_{eff} < T_{resamp}$ **then**
35: $\mathcal{S}_t = \text{resample}\,(\mathcal{S}_t)$
36: **end if**
37: $t = t + 1$
38: **end if**
39: **end loop**

Bibliography

[Bülthoff and Bülthoff, 2003] Bülthoff, I. and Bülthoff, H. H. (2003). Image-based recognition of biological motion, scenes, and objects. *Analytic and holistic processes in the perception of faces, objects, and scenes*, pages 146–176.

[Cheeseman et al., 1987] Cheeseman, R. S. M. S. P., Smith, R., and Self, M. (1987). A stochastic map for uncertain spatial relationships. In *4th International Symposium on Robotic Research*, pages 467–474.

[Fox et al., 1997] Fox, D., Burgard, W., and Thrun, S. (1997). The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, 4(1):23–33.

[Gallart del Burgo, 2013] Gallart del Burgo, X. (2013). Semantic mapping in ros. Master's thesis, KTH Royal Institute of Technology, Stockholm, Sweden.

[Grisetti et al., 2010] Grisetti, G., Kummerle, R., Stachniss, C., Frese, U., and Hertzberg, C. (2010). Hierarchical optimization on manifolds for online 2d and 3d mapping. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 273–278. IEEE.

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

[Hawes et al., 2007a] Hawes, N., Sloman, A., Wyatt, J., Zillich, M., Jacobsson, H., Kruijff, G.-J. M., Brenner, M., Berginc, G., and Skocaj, D. (2007a). Towards an integrated robot with multiple cognitive functions. In *AAAI*, volume 7, pages 1548–1553.

[Hawes et al., 2007b] Hawes, N., Zillich, M., and Wyatt, J. (2007b). Balt & cast: Middleware for cognitive robotics.

[Klein and Murray, 2007] Klein, G. and Murray, D. (2007). Parallel tracking and mapping for small AR workspaces. In *Proc. 6th IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR'07)*, Nara, Japan.

[Kohlbrecher et al., 2011] Kohlbrecher, S., Meyer, J., von Stryk, O., and Klingauf, U. (2011). A flexible and scalable SLAM system with full 3d motion estimation. In *Proc. IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*. IEEE.

[Konolige et al., 2011] Konolige, K., Marder-Eppstein, E., and Marthi, B. (2011). Navigation in hybrid metric-topological maps. In *Proceedings of ICRA*, Shanghai.

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

[Montemerlo et al., 2002] Montemerlo, M., Thrun, S., Koller, D., and Wegbreit, B. (2002). FastSLAM: A factored solution to the simultaneous localization and mapping problem. In *AAAI/IAAI*, pages 593–598.

[Montemerlo et al., 2003] Montemerlo, M., Thrun, S., Koller, D., and Wegbreit, B. (2003). FastSLAM 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges. In *International Joint Conference on Artificial Intelligence*, volume 18, pages 1151–1156. Lawrence Erlbaum Associates LTD.

[Newcombe et al., 2011a] Newcombe, R. A., Davison, A. J., Izadi, S., Hilliges, O., Molyneaux, D., Kim, D., Kohli, P., Shotton, J., Hodges, S., and Fitzgibbon, A. (2011a). KinectFusion: Real-time dense surface mapping and tracking. In *Proc. IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR'11)*, Basel, Switzerland.

[Newcombe et al., 2011b] Newcombe, R. A., Lovegrove, S. J., and Davison, A. J. (2011b). DTAM: Dense tracking and mapping in real-time. In *Proc. of the International Conference on Computer Vision (ICCV)*, Barcelona, Spain.

[Nüchter et al., 2006] Nüchter, A., Wulf, O., Lingemann, K., Hertzberg, J., Wagner, B., and Surmann, H. (2006). 3d mapping with semantic knowledge. In *RoboCup 2005: Robot Soccer World Cup IX*, pages 335–346. Springer.

[Pronobis and Jensfelt, 2012] Pronobis, A. and Jensfelt, P. (2012). Large-scale semantic mapping and reasoning with heterogeneous modalities. In *Proceedings of the 2012 IEEE International Conference on Robotics and Automation (ICRA'12)*, Saint Paul, MN, USA.

[Pronobis et al., 2010] Pronobis, A., Mozos, O. M., Caputo, B., and Jensfelt, P. (2010). Multi-modal semantic place classification. *The International Journal of Robotics Research (IJRR), Special Issue on Robotic Vision*, 29(2-3):298–320.

[Stachniss, 2013] Stachniss, C. (2012/2013). SLAM course - slides and lecture recordings. http://ais.informatik.uni-freiburg.de/teaching/ws12/mapping/.

[Thrun, 2002] Thrun, S. (2002). Robotic mapping: A survey. In Lakemeyer, G. and Nebel, B., editors, *Exploring Artificial Intelligence in the New Millenium*. Morgan Kaufmann.

[Thrun et al., 2005] Thrun, S., Burgard, W., Fox, D., et al. (2005). *Probabilistic robotics*, volume 1. MIT press Cambridge.

[Zender et al., 2008] Zender, H., Martínez Mozos, O., Jensfelt, P., Kruijff, G.-J., and Burgard, W. (2008). Conceptual spatial representations for indoor mobile robots. *Robotics and Autonomous Systems*, 56(6):493–502.

Glossary

List of Acronyms

3mE	Mechanical, Maritime and Materials Engineering
AI	Artificial Intelligence
AR	Augmented Reality
BF	Bayes filter
BMechE	BioMechanical Engineering
DOF	Degrees of Freedom
DWA	Dynamic Window Approach
DCSC	Delft Center for Systems and Control
EKF	Extended Kalman filter
GUI	graphical user interface
HRI	human-robot interaction
KF	Kalman filter
LEMTOMap	Large Environments Metric TOpological Mapping system
PF	Particle filter
ROS	Robot Operating System
SAR	Search and Rescue
SfM	Structure from Motion
SLAM	Simultaneous Localization and Mapping
TU Delft	Delft University of Technology

UKF Unscented Kalman filter

VCS version control system

Index

V

W