On 2D SLAM for Large Indoor Spaces - A Polygon-Based Solution

Johann Dichtl

Numéro d'ordre : 2019MTLD0006 **École doctorale SPI Lille**

IMT Lille Douai
École Mines-Télécom
IMT-Université de Lille

Université de Lille 1 SCIENCES ET TECHNOLOGIES

THÈSE

présentée en vue
d'obtenir le garde de

Docteur

Discipline: Informatique et Applications

par

Johann DICHTL

DOCTORAT DE L'UNIVERSITE DE LILLE
DELIVERE PAR IMT LILLE DOUAI

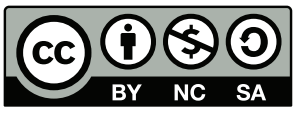
On 2D SLAM for Large Indoor Spaces

A Polygon-based Solution

Soutenue le 2 juillet 2019 devant le jury d'examen:

Présidente :	Prof. Ouiddad Labbani
Rapporteurs :	Ouiddad Labbani – Professeur – Université de Limoges
	Mikal Ziane – Maître de Conférences HDR – Université Paris Descartes (Paris 5)
Examinateur :	Laetitia Matignon – Maître de Conférences – Université Claude Bernard Lyon 1
Directeur :	Bouraqadi Noury – Professeur – IMT Lille Douai
Co-Encadrant de thèse :	Fabresse Luc – Professeur – IMT Lille
	Lozenguez Guillaume – Maître-Assistant – IMT Lille Douai

Département IA, IMT Lille Douai
Ecole Doctorale SPI 072 (Univ. Lille, Univ. Artois, ULCO, Univ. Polyt. Jauts de Fr., Centrale Lille, IMT Lille Douai)



Acknowledgments

Firstly, I would like to express my sincere gratitude to my supervisor, Prof. Noury Bouraqadi, as well as to Prof. Luc Fabresse and Dr. Guillaume Lozenguez for their continuous support of my Ph.D study and related research. From start to finish they were always supportive and helpful and made this thesis possible. Their comments, insights and feedback were always welcomed and I learned a lot from all of them.

I would also like to thank my colleagues and friends, in particular Khelifa Baizid, Pablo Tesone, Pau Segnovia, and Xuan Sang Le. They were equally supportive and altogether pleasant to have around. In this regard, special thanks to Pablo for his helpful comments whenever I was stuck with a Pharo-related problems. His experience in that manner was invaluable and help greatly with the software implementations of my work.

Deep thanks also go to my family for their constant support. They are one of the reasons why I was able to get as far as I did. With their help and patience, I had all the support I could dream of.

Lastly my thanks goes to the *CPER DATA* project (supported by Région Hauts de France, and the French state), and the *DataScience* project (co-financed by European Union with the financial support of European Regional Development Fund (ERDF), French State and French Region of Hauts-de-France). Both provided funding that was used for my work at the IMT.

Abstract

Indoor SLAM and exploration is an important topic in robotics. Most solutions today work with a 2D grid representation as map model, both for the internal data format and for the output of the algorithm. While this is convenient in several ways, it also brings its own limitations, in particular because of the memory requirements of this map format.

In this thesis we introduce PolyMap, a 2D map format aimed at indoor mapping, and PolySLAM, a SLAM algorithm that produces PolyMaps. Our PolyMap format utilizes polygons built from vectors to model the environment, and as such this is a special case of vector-based SLAM algorithms. The PolyMap format leads to approximation of laser scan points with line segments, effectively reducing sensor noise by averaging over multiple points. We also provide an algorithm that creates topological graph for Navigation tasks from the PolyMap.

Our PolySLAM algorithm uses keyframes based on polygons to create a global map in the PolyMap format. Each keyframe itself is a small PolyMap that depicts the local environment of the robot. With accurate keyframes, we are able to create maps of the environment that allow us to localize the robot with a high accuracy. This in return provides better alignment for our keyframes, and translates to good results when integrating keyframes into the map and an accurate pose estimate. In the end, we create maps that are consistent and usable for navigation and path planning without employing global optimization techniques.

In experiments, we evaluate the performance of our PolyMap format, the PolySLAM algorithm, and the topological graph that we create from PolyMaps. We confirm that our PolyMap format has advantages over the more popular occupancy grid and feature-based map formats. With multiple experiments, based on real world data and simulations, we find that this SLAM algorithm is showing good results on medium-sized maps despite the lack of global optimization. Experiments in regard of navigation on PolyMaps (with the help of topological graphs) also hold positive results.

Keywords: Indoor Mapping, VectorMaps, SLAM, PolySLAM, Navigation

Contents

Introduction

Chapter 1

Contents

1.1 Context

Mobile robots rely heavily on accurate representations of the environment (often referred to as *maps*) to fulfill their tasks. Maps are often restricted to 2D as a direct result of the capabilities of the sensors mounted on the robot. Creating an accurate representation of the environment comes down to the act of *Simultaneous Localization and Mapping* (SLAM). Inside buildings, GPS signals are too weak to be used to localize robots. Hence we face a so-called Chicken-and-Egg-Problem, as Localization requires a map, and Map Building requires the current location. This bootstrapping problem has been tackled from different directions, which result in a few major techniques to solve the problem (namely *Kalman Filters*, *Particle Filters* and *Graph Optimization*, all explained in Chapter 2). The most commonly used 2D map format is the occupancy grid, where cells in a grid depict a small area as either obstacle, free space, or unknown space. Another noteworthy map format is the *feature-based* map format, which holds only selected key elements of the environments (e.g. corners, or trees) and their relative spatial relationship in the map format. Lastly, we also have the *vector-based* map format, where the environment is represented by line segments (referred to as *vectors*).

One key area of interest to us is multi-robot autonomous exploration. In this scenario, two or more robots are exploring an unknown area while simultaneously mapping it, ideally creating a complete map that can be used by humans and robots alike. To keep track which areas are not yet fully explored, *frontiers* are employed to define the transitions from explored to unexplored space. These frontiers are then used to prioritize areas of interest and in case of multi-robot exploration to assign different areas to different robots to explore.

While robots are exploring the environment they continuously update their local map of the environment. The updated map will then be shared over wireless connection with human operators/supervisors, and in case of

multi-robot collaboration it will also be shared between robots. In both cases, occupancy grids are not a very efficient format because they require a lot of bandwidth, limiting the frequency of map exchanges.

Robots must be able to autonomously navigate the environment to achieve tasks such as exploration by computing paths from the current position to a target location. In general, path computation can be done on a topological graph which is computed from the map. Ideally we want such a graph to be sparse, for fast computation, and yet dense enough to have nearby nodes at all important areas of the environment.

Finally, one aspect to consider is the map quality and in tandem with that the precision on localization. Both aspects are intertwined, as a high quality map enables better localization, and a precise estimate of the robots position (and therefore travel trajectory) allows map updates with well aligned sensor data.

1.2 Problem Statement

In the context of autonomous exploration, we identify four map format characteristics that are important to for the performance in mapping, localization, navigation, and map exchange in general.

1. **Closed Maps**
   An important criteria for maps is to allow to distinguish between explored and unexplored space. Occupancy grids accomplish this by marking every grid cell that has been explored as free or occupied and leaving every other cell marked as unexplored. Feature-based maps don't support this at all, and are therefore unsuitable for exploration tasks on their own. Most vector-based maps are open as well, making it impossible for path planner to distinguish unexplored areas from already explored space.

2. **Frontiers**
   A frontier defines the transition from free space into unexplored space in the absence of obstacles. One limitation is the support (or lack of) frontiers by the map format chosen. Occupancy grids don't support frontiers, despite being a *closed* map format. However the map format allows (at the cost of additional computations) to extract frontiers. Naturally, if the map is updated, the computations to extract frontiers need to be repeated as well. The other two map formats mentioned (feature-based and vector-based) don't have support for frontiers at all, which is a major drawback for exploration tasks.

3. **Memory Footprint**
   Another limitation is the memory footprint of the map. Here the situation is the opposite, with feature-based maps and vector-based maps being at advantage by showing small memory footprints, while occupancy grids are large and cumbersome to deal with. Since mobile robots typically can share live data with humans and other robots only via wireless connections, bandwidth is a rather limited resource.
4. **Topological Graphs**
   To have a robot navigate on a map beyond its field of vision (limited by the sensor range and obstacles), we need a topological map. This is achieved by building a topological graph. In such a graph areas in the map being represented by nodes in the graph. Two nodes are connected by an edge if there is a direct path (e.g. by unobstructed line of sight) between the two locations linked to the nodes. The ability to build a topological graph from the map (or directly use the map as a topological graph) is crucial for path planning, the core task of autonomous exploration. Occupancy grids can be interpreted as high density topological graphs with implicit edges. As such, they are not particularly efficient but get the job done at the cost of a relatively high computation time. Neither feature based map formats nor vector-based maps support topological graphs on their own.
5. **Visualization**
   The last requirement for map formats is visualization for human use. Even if the robots are exploring the area autonomously, the resulting map is typically inspected by humans and used to perform further tasks. Occupancy grids are easy to understand by humans. Being able to see the transition from explored to unexplored space is a welcome feature here as well. Feature-based map formats are in general much harder to understand by humans due their sparse nature. Last, Vector-based map formats are also easy to read by humans. However unless they are closed, it's not always possible to see where the explored space ends and unexplored space starts.

Map format	Frontiers	Memory Footprint	Closed	Topol. Graphs	Visualization
Grid Maps	implicit	high	yes	yes	good
Feature-based	no	low	no	no	limited
Vector-based	no	low	no	no	good

Table 1.1: Overview of relevant map format properties.

As we can see in Table 1.1, none of the map format presented fulfills all

criteria. The closest match is the occupancy grid format, however the high memory footprint means that this map format doesn't scale well with increase of map size.

1.3 Contributions

As presented in the previous section, among the current map formats available none fulfills all criteria required in the context of autonomous exploration. Our main contribution is a full robotic stack of a SLAM (PolySLAM) and a navigation algorithm, based on a compact polygon-based map format (PolyMap).

PolyMap satisfies the five map format requirements (presented in Section 1.2). PolyMap is inspired by vector-based map formats with the following additional properties:

1. **Directed Vectors**
   We set the direction of the vectors in such a way, that the "left" side of the vector is explored/traversable space, and the other side is unexplored space. This is helpful in merging polygons and allows PolySLAM to perform backface culling.
2. **Simple Polygons**
   All vectors must be part of a simple polygon. This property ensures that all explored space is always inside a polygon and as a consequence we have a closed map with well defined transitions between explored and unexplored space. In addition this also allows us to build topological graphs as explained later.
3. **Vector Types**
   Defining vector types allows us to use vectors as frontiers. These frontier vectors allow us to differentiate between obstacles in frontiers when handling the polygons that make up our map. By adding the *sector* type as well, we also can split polygons into smaller polygons while maintaining all the above properties. This is useful when we want to handle only a part of the map but want to keep the polygon coherent, as is the case when we build our BSP-tree-based internal map representation.

PolyMap is also suitable for Navigation, thanks to an algorithm that creates topological graph from a PolyMap. The topological graph is sparse when compared to occupancy-grid-based topological graphs, and performs well in our experiments.

PolySLAM directly creates PolyMaps out of 2D laser scans. PolySLAM itself has two aspects that rely on our PolyMap format that sets it apart from other common SLAM implementations:

1. **Point-To-Vector ICP**
   The classic ICP algorithm uses a point-to-point approach to create corresponding pairs of points. However since we already have vectors that approximate the shape of obstacles, we are able to associate points with vectors. Using the point and its projection point of the closest vector allows us to get good results with high accuracy in terms of localization as the end result of the ICP algorithm.
2. **Backface Culling**
   Since we have directed vectors, we can conclude that vectors whose normal vector are pointing away from the sensor center are no possible matching candidate for the ICP matching algorithm. Discarding such vectors reduces false positives in the matching process and overall improves the result of the ICP algorithm.

We show in experiments that our map format performs well on its own, by converting grid-based maps into PolyMaps. The converted maps have a significantly smaller memory footprint, while providing the same information about the environment with the additional benefit of explicit frontiers build into the maps.

Our implementation of PolySLAM performs well, despite the lack of global optimization. The localization is precise, and shows remarkable small drift even on long trajectories. Experiments (both simulated and in real-world setups) confirm that PolySLAM performs well in a variety of scenarios.

1.4 Thesis Outline

The rest of this dissertation is structured with the following chapters:

Chapter 2 introduces the state of the art in regard of the thesis topic, in particular 2D SLAM with vector-based maps.

Chapter 3 highlights our contributions and explains in detail our proposed map format PolyMap and SLAM algorithm PolySLAM.

Chapter 4 covers robotic navigation in combination with our PolySLAM format.

Chapter 5 is dedicated to our experiments. It shows the possibilities and limits of PolySLAM, both in simulations and in experiments with real robots in realistic environments.

Chapter 6 concludes this dissertation. It sums up the contributions of this thesis and presents several lines of future work.

2D SLAM FOR AUTONOMOUS EXPLORATION

Chapter 2

Contents

This chapter presents an overview of current techniques to achieve Simultaneous Localization and Mapping (SLAM) in the context of autonomous exploration. It first presents what a SLAM algorithm is in terms of input, output and its constitutive logical parts. Each following section then focuses on one logical part of SLAM and describes variations of current SLAM approaches. This chapter ends with a comparison of the main existing SLAM solutions.

2.1 What is SLAM?

A SLAM algorithm aims at providing a model of the environment (a map) and a pose (location and orientation) of the robot within the environment (see Figure 2.1). The input for the SLAM algorithm is sensor data. In case of 2D SLAM, this typically entails (2D) laser range finder, odometry (e.g. based on wheel encoders), and Inertial Measurement Units (IMU). The sensor data is reduced before further processing, often by only considering data of key positions (keyframes). The process of keyframe creation (i.e. when to collect data and when to ignore

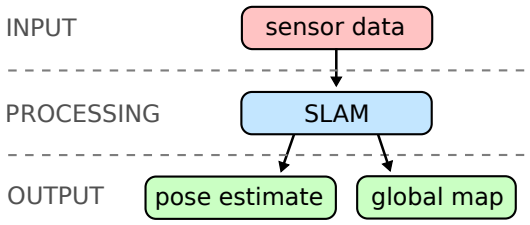


Figure 2.1: The basic task of a SLAM algorithm: take sensor input, create a map from it and provide a pose estimate.

it) typically is based on heuristics, e.g. by considering the distance traveled, or the overlap with the previous keyframe. This drastically reduces the computational load by limiting the data that needs to be further processed by the SLAM algorithm.

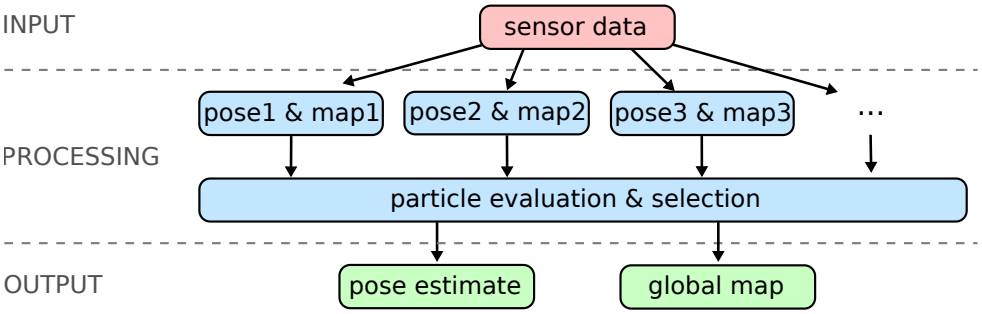


Figure 2.2: Simplified *Particle Filter* based SLAM algorithm.

To give a finer overview of SLAM, we will now present two different SLAM approaches. One approach to SLAM relies on particle filters [SGHB04] as illustrated in Figure 2.2. Multiple estimates about the pose and map are computed in parallel and independent from each other. The second processing stage evaluates the estimates and passes the best one to the output layer, making it available to the user.

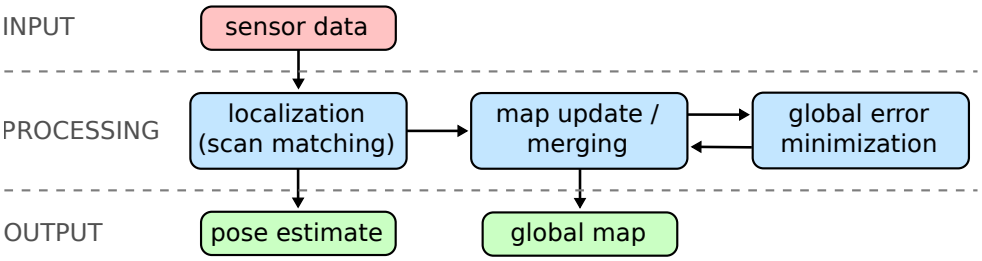


Figure 2.3: Simplified *Pose Graph Optimization* based SLAM algorithm.

In contrast to this, the Pose Graph Optimization [SP13a] approach shown in figure 2.3 first determines the pose estimate. This is achieved by using an initial guess (e.g. via the robots odometry) and applying scan matching techniques to refine the pose estimate. With this, the data is merged into the map, updating the global map of the environment. Periodically, the error in the map is minimized using pose graph optimization.

In this chapter we analyze 2D SLAM algorithms from multiple different perspectives. First, we look at the used map format. Second, we focus on localization of the robot within a map. Third, we explore how the map is constructed from a set of sensor data. Forth, we discuss how global error and sensor bias is dealt with via global optimization. Fifth, we discuss how SLAM algorithms impact navigation. And last, we compare different SLAM

algorithms.

Since our goal is 2D SLAM, we focus on sensors that either provide data in 2D euclidean space, or that can be easily converted into 2D euclidean space. As such, most 2D SLAM algorithms use laser range finders as their primary sensor, often supported by the robot's odometry and related sensors (e.g. IMU sensors). Whenever an algorithm relies on a specific sensor type, it will be mentioned in the respective section.

2.2 Map Formats

The purpose of a map varies from use case to use case. While, localization is always included in the context of SLAM, other popular uses for maps include visualization (e.g. for tele-operation) and navigation (e.g. for exploration). Therefore it is of no surprise, that different map formats emerged to satisfy different use cases. The two major classifications that are of interest to us are *metric* and *topological* map formats. Metric maps contain some form of coordinates that allow to measure distance, topological maps model connectivity between different locations. As shown in Figure 2.4, metric maps can be further split into three sub-groups: occupancy grids, feature-based, and geometric. On top of that, some map formats combine multiple characteristics (typically topological and one metric format), and are refereed to as hybrid maps.

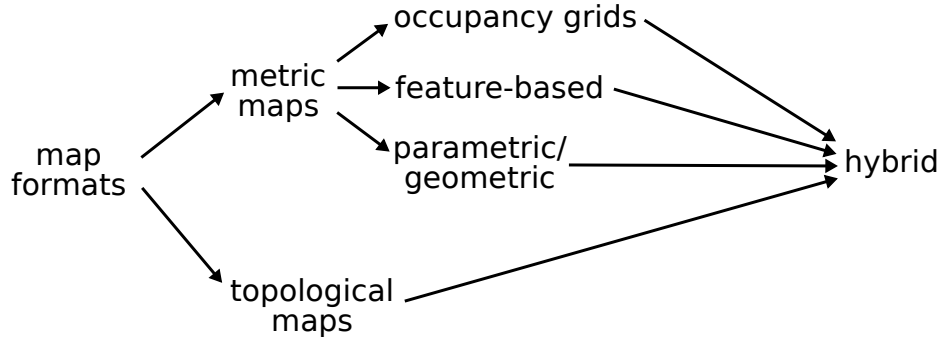


Figure 2.4: Map formats and their relationships with each other.

A map is modeling geometric information such as obstacles and free space, typically in coordinates relative to the robots starting position. For 2D maps, there exist three major metric map formats: *occupancy grids*, *feature-based*, and *parametric*.

Aside from metric data, it is often desirable to have also topological information embedded into the map. Topological maps provide a model that allows to efficiently compute which areas of the environment are connected with each other. This type of information is critical for path planning tasks.

Topological maps usually rely on a graph structure that can be operated on with algorithms such as A-star or RRT [Lav98].

Multiple 2D map formats have been standardized in [Gro16], including occupancy grids, feature-based maps, and topological maps. Feature-based maps and parametric maps are aggregated under the term *Geometric Map* within the standard, and assume that features hold metric information. We will continue to distinguish between the two because feature-based maps don't necessarily model obstacles.

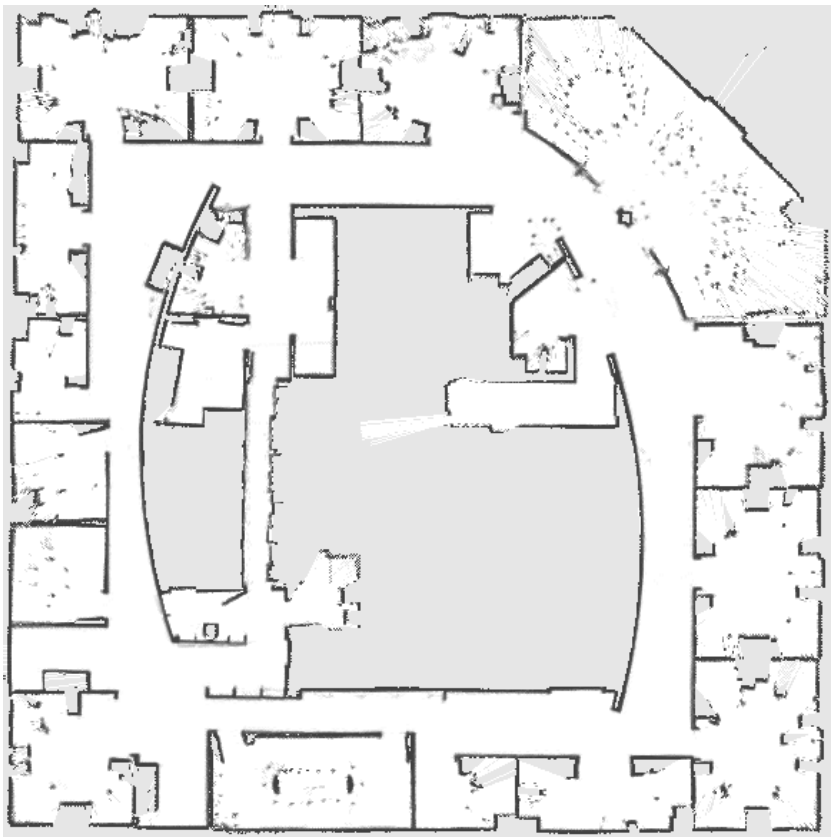


Figure 2.5: A 3-state occupancy grid created from the *Intel Research Lab* data set. Image and data set source: `http://www2.informatik.uni-freiburg.de/~stachnis/datasets.html`

2.2.1 Occupancy Grids

The dominating 2D map format is the *occupancy grid*, also called *grid map*. The occupancy grid format gained a lot of attention with the work of Grisetti et al. [SGHB04], which in return builds on *Monte Carlo Localization* that was introduced by Fox et al. [FBDT99]. It has since then become a de facto standard for 2D maps, and turned in 2016 into an official IEEE standard [Gro16]. Figure 2.5 shows an example of an occupancy grid.

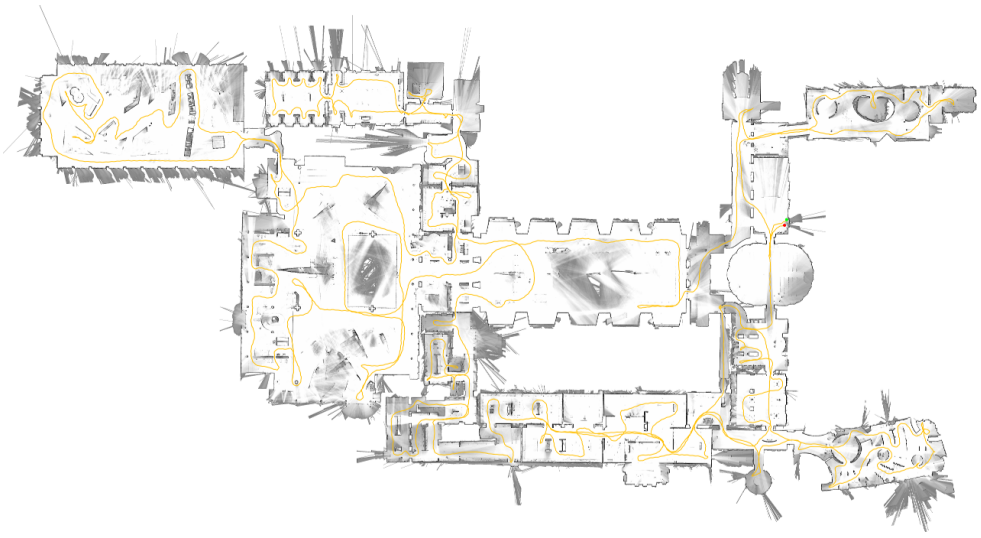


Figure 2.6: A probabilistic occupancy grid, showing a part of the *Deutsches Museum* in Munich. Image source: [HKRA16], data set source: [Goo16]

Formally, the occupancy grid is usually defined in a probabilistic way, where a grid of $M \times N$ with cells of size $\delta \times \delta$ that hold the estimated probability that the space represented by the respected cell is traversable or not. In early work by Moravec and Elfes [ME85], the grid cells hold values in the range $(-1, 1)$, with negative values depicting (probably) empty/free space and values greater than zero hinting occupied space. Later, Elfes [E$^+$90] switches to cell values in the range $(0, 1)$ with a value of 0.5 for unobserved cells to have a direct correspondence between the cell value and the probability that the cell is occupied. All cells are initially set to the value corresponding to unknown/uncertain, meaning they hold no bias towards a free or occupied state.

In practical solutions, each cell is typically quantized into three possible states: (1) traversable/free, (2) occupied, and (3) unknown/unexplored. In this context, *traversable* only means that the space has been observed (i.e. it is not unknown space) and is not occupied by an obstacle. It is quite possible that the space is not reachable by the robot, for example because of nearby obstacles blocking the path.

Figure 2.5 shows an example of a three-state grid map created with GMapping [Gera]. The state of a cell can only be traversable (white), occupied (black), or unknown (gray). This quantization reduces the required resources to create the map and distribute it in a network.

In contrast to this, figure 2.6 displays a probabilistic grid map. Here the cells can hold values between 1.0 and 0.0, based on how many laser beams were detecting an obstacle inside the cell. The figure reflects this by showing pixels in different shades of gray. However, white represents both traversable

space and unknown space.

The theoretical maximum accuracy of this map format depends on the resolution of the map, i.e. the size of the cells. Naturally, a higher resolution allows for a better approximation of reality, but also increases the required resources (CPU and memory) to create and use the map. The most common resolution of todays occupancy grids is 5cm, such as in Figures 2.5 and 2.6, which is the default grid size in most common implementations.

This format is easy to visualize for human use (as seen in figure 2.5 and 2.6), and can also be used for navigation purposes. This makes the map format very versatile. It is however relatively expensive in terms of CPU and memory requirements, which limits the maximum size of the created maps. The map resolution (i.e. how big is a single cell) influences both resources as well.

Topological information is embedded only implicitly, by treating the grid as a big graph. Here we have edges between two neighboring cells if and only if both cells are marked as traversable space. This results in a relatively large graph that is not optimized for navigation tasks. Therefore, occupancy grids are only used for navigation tasks in relatively small environments, e.g. in an office building.

2.2.2 Feature-based Maps

Feature- or *landmark-based* map format stores distinct features or landmarks of the environment (e.g. corners or artificial beacons) with their relative position. For example Castellanos et al. [CMNT99] uses line segments as features. Unlike the *occupancy grid*, this is a sparse representation of the environment. The advantages of this format stem from the sparse nature of the landmarks, typically translating into significantly smaller memory footprints and computation power. The disadvantage is that this format does not model the shape of obstacles, making path planning difficult to achieve. It is also harder to create visualization for human use, as shown by the example in figure 2.7. This map format is typically used in conjunction with Kalman-Filter-based SLAM (detailed in Section 2.4.1).

A way to overcome limitations of feature-based maps is to use them in conjunction with occupancy grids. For example Wurm et al. [WSG10] use occupancy grids indoors and feature-based mapping outdoors. This outperforms solutions that only use one of the two map formats. One drawback of this strategy is that the robot needs to be able to choose which map format to use for localization if the resulting estimates contradict each other. Wurm et al. use reinforcement learning to train their algorithm to deal with such contradictions.

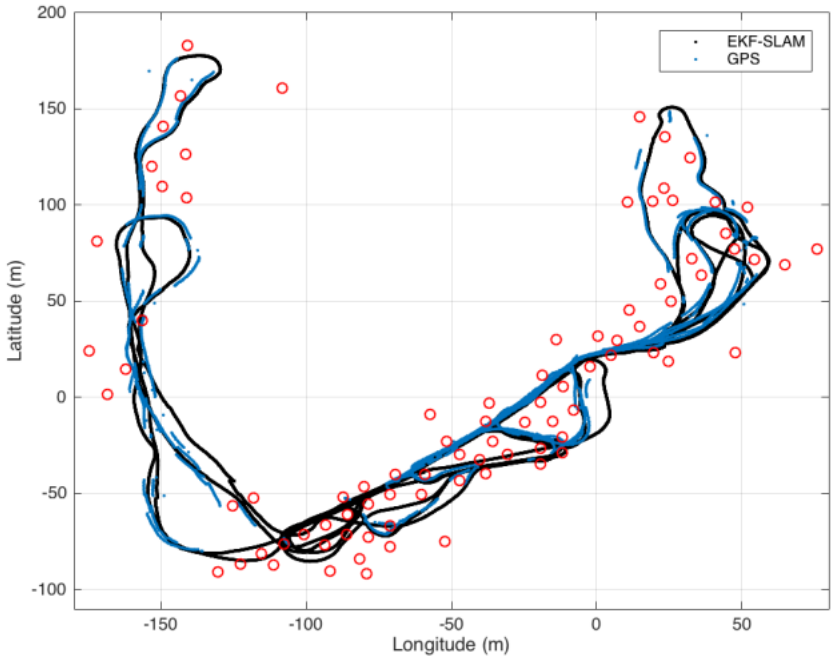


Figure 2.7: An example of a landmark-based map from the *Victoria Park* data set. The red circles show the detected landmarks, while the robot's estimated trajectory is displayed in black (EKF-SLAM) and blue (GPS). Image source: `https://jay.tech.blog/2017/03/26/ekf-slam/`

In contrast to this, Castellanos et al. [CMNT99] build a map with line segments, and ise their centers and orientations as landmarks. This allows to combine a feature-based format with geometric information about the environment. The landmarks can be used for an EKF-SLAM approach, while the geometric information provides a good representation of the obstacles detected by the sensors.

2.2.3 Parametric Maps

The third and currently least used map format of the three is a parametric representation of the environment. Parametric in this context refers to parametric curves, such as splines, bezier curves, and line segments. Among these, line segments are by far the most used. In the context of SLAM line segments are often also called vectors. In this thesis we treat these two terms as synonyms. This map format is classified in the IEEE standard [Gro16] as *geometric map*. The standard only supports vector-based maps and not the more general notation of parametric curves.

While curves find application in maps for humans, they don't find much use in 2D SLAM. The reason for this is, that curves can be reasonably well

approximated with a series of short line segments, and these line segments are significantly easier to handle for computations such as collision testing.

Line segment based maps model borders between traversable space and obstacles via line segments. This representation has been used since early robotic Localization and Mapping research [CL85, Cro85]. Compared with occupancy grids, line segment based maps are sparse in nature, since only the boundaries are explicitly modeled. The line segments can either form polygons (e.g. Zhang et al. [ZG00]), defining closed areas, or they can leave space open (e.g. [BR05, MP00, SBG02]). The former is better suited for navigation tasks, as one cannot accidentally exit explored space. The later is easier to create since we don't need to maintain closed shapes, and also makes it easier to merge multiple overlapping vector. Another optional feature for vector maps that we will make use of later is the explicit presence of frontiers. While typical grid maps don't model frontiers explicitly, they can still be detected by comparing cells with its immediate neighbors – a task that can be done even after the map has already been created. On vector maps, if frontiers are needed, they need to be created during the SLAM process. Vector-based maps are comparable to occupancy grids in terms of accuracy at representing the environment while the memory footprint is about an order of magnitude lower [BLFB16]. An example of a SLAM approach using line segments for mapping can be found at [LLW05].

Examples of vector maps are shown in figures 2.8. The first figure presents an *open* vector map – a map that is not fully enclosed by a boundary.

Vector maps do not contain topological information. However the sparse nature of the map representation reduces the required resource (in terms of CPU and memory) to create and maintain a topological map from the vector map [DLL+19].

The theoretical maximum accuracy of this map format does not depend on a predefined cell size, like it does with occupancy grids, but on the minimum size of the vectors used, as well as the type of environment and sensor accuracy. For example, most indoor environments can be well approximated with line segments, while outdoor environments are more challenging in this regard with more irregular shaped obstacles.

Our focus lies on this map format since it combines advantages of occupancy grids and feature-based maps. In particular, it is a lightweight map format, that allows good approximation of indoor environments and is suitable for navigation tasks.

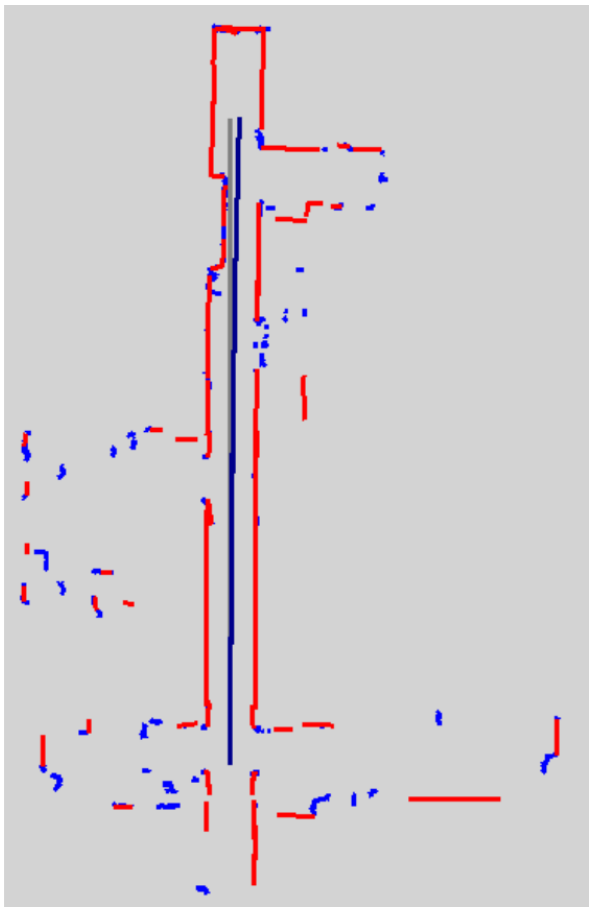


Figure 2.8: A small vector map created with edge-extraction from point clouds. Red edges are long enough to be used for scan matching, while blue edges are shorter and added to provide more details when visualizing but otherwise ignored. Source: [Jel15].

2.2.4 Topological Maps

While metric maps model obstacles and free space, topological maps model connectivity and reachability of the environment [KW94]. Their task is to provide a model of the environment that allows to efficiently compute which areas of the environment are connected with each other, and how to get from one area to another. Topological maps usually rely on a graph structure (called *topological graph*) that can be used with algorithms such as A^* [App66] or RRT [Lav98]. This is also reflected in the IEEE standard, which defines topological maps as sets of nodes and edges [Gro16].

In the context of SLAM, topological maps typically also contain metric information, which technically classifies them as a hybrid representation. We will still refer to them as topological maps if the focus of the format lies on the topological aspect.

Since the SLAM process typically works with consecutive robot poses,

building a pose graph on the fly is relatively easy. Such a pose graph already is a topological map, albeit in general an incomplete one.

Other ways to create a topological maps use the metric map to compute connectivity. This is done in a two step fashion: first select nodes, and second compute which nodes are connected with each other. Selecting nodes can be done in many different fashions. Popular methods are random sampling, Voronoi Diagrams [Thr98, BJK05], and map features such as corners. Determining which nodes are connected can be done with simple ray tracing. An underlaying data structure such as Binary Space Partioning Trees (BSP-Trees) [TN87] help to speed up the computation.

An example of a topological map is displayed in 2.9. The upper image shows the full graph, the lower image is zoomed in, enabling us to see the individual nodes and edges. Unlike in a pure trajectory-based graph, this graph includes loop closures, resulting in nodes with more then two edges.

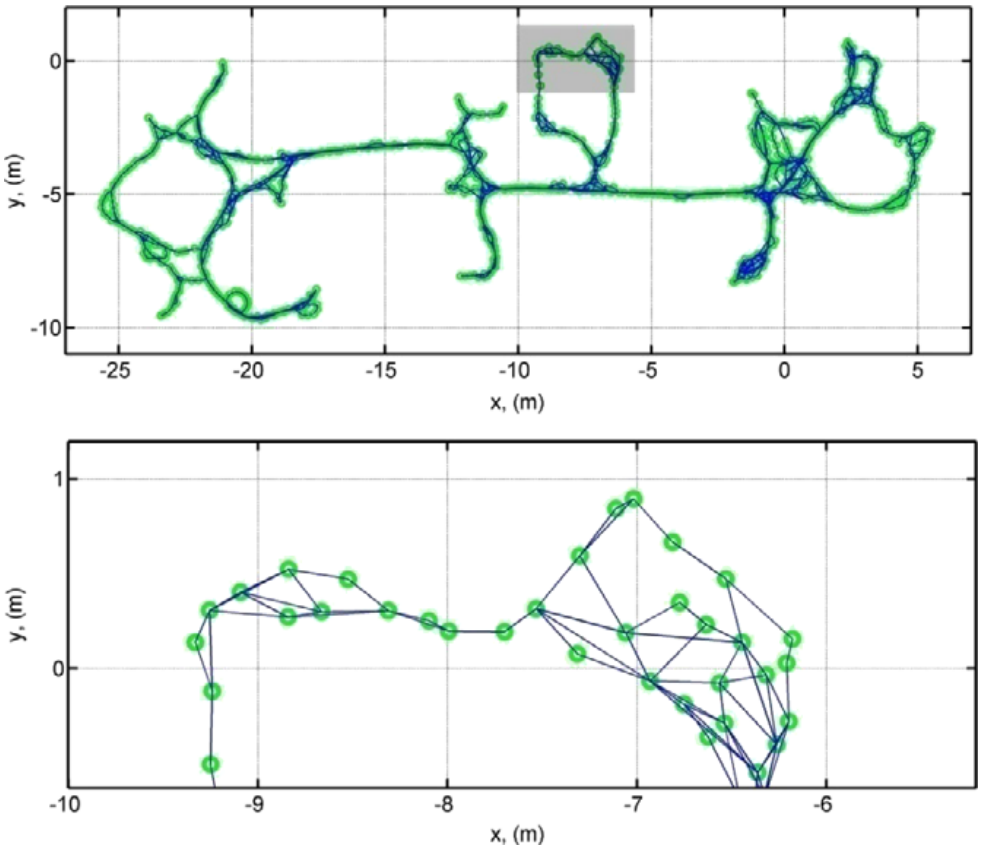


Figure 2.9: A topological map. The bottom image shows a zoomed-in section of the gray rectangle in the upper image. Image source: [MW10]

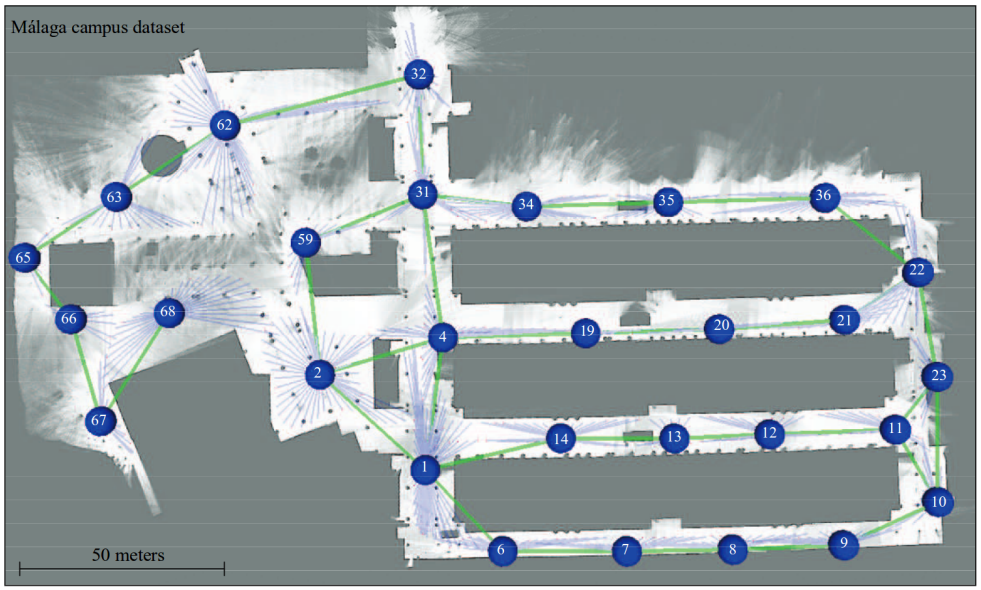


Figure 2.10: A topological-metric hybrid map of an outdoor environment. Image source: [BFMG08]

2.2.5 Hybrid Maps

Since SLAM tasks often require navigation capabilities in parallel, metric and topological maps are often created simultaneously. On some cases two separate maps are created [BGFM09], while in other cases a hybrid map containing both metric and topological information of the environment (e.g. [BFMG08]).

For example, [SD98] proposes a hybrid map model where a set local maps are organized in a topological way. The local maps are metric representations of the environment with its own local reference frame. Navigation tasks that don't exceed the scope of the local map are performed with the local map only, while those that are beyond the area of the local map rely on the topological structure that connects the local maps. Interestingly, the relative position of the local maps does not need to be very accurate to execute navigation tasks, and therefore a global optimization step is not necessary.

Similarly, Blanco et al. [BFMG07, BFMG08, BGFM09] create a global topological map, but the local metric maps overlap, covering the whole area. In their work both the metric and topological formal approach are formulated as a Bayesian inference problem. The focus does not lie on global optimization of the metric map as the topological map is used for global task planning. Instead only the relative pose of neighboring local maps are optimized without a global context. As a result, their approach is able to handle large scale maps, including a map create from a robot traveling for 2km. Figure 2.10 shows one such map of an outdoor environment. Each node is linked to a local map, the

green edges are part of the topological graph.

2.3 Localization

Localization in the context of robotics can be categorized into three type: metric, topological, and semantic. Metric localization is the task to provide a pose estimate on a metric map of the environment. Topological localization answers the question which vertex of a topological graph is associated with the current position of the robot. Semantic localization provides us with semantic information about the robots local environment, for example "the robot is in the kitchen right now".

The focus of our work lies on metric and topological localization. Obtaining the topological location of a robot is easier once the metric location has been determined, since topological nodes are generally associated with areas of the metric map. For example, if we have a hybrid map as described by Blanco et al. [BFMG08], the topological location follows simply by determining which local map the robot resides in.

2.3.1 Techniques

The most basic type of metric localization is *Dead Reckoning*. This technique takes sensor data such as wheel encoders and concatenates delta pose changes to provide an estimate of the current position. This works reasonable well for short distances, but since there is no feedback loop to take sensor noise and bias into account the estimation error grows quickly beyond acceptable limits. To overcome this problem, two major techniques for metric localization have emerged: *Particle Filters* and *Kalman Filters*. A third method, the *Pose Graph Optimization*, also helps with localization, but only within the context of SLAM, not in the broader sense of determining the robot pose on an already finished map.

Part of the difficulties with localization stems from the non-linearity of the robots motion model. When a robot moves , we typically have a translation and a rotation component to deal with. While the translation itself is linear in nature, when combined with the rotation we end up with a non-linear transformation. This means, that we no longer can describe the motion error model via a Gaussian distribution, since the probability distribution has a crescent shape. Figure 2.11 illustrates this. The top row shows the density of the probability function while the bottom row shows the pose of random samples. In the left column both translation and rotation contribute to the pose error, as we would expect if the robot motion consists of both translation and rotation. The center column shows an error that is mostly dominated by translation,

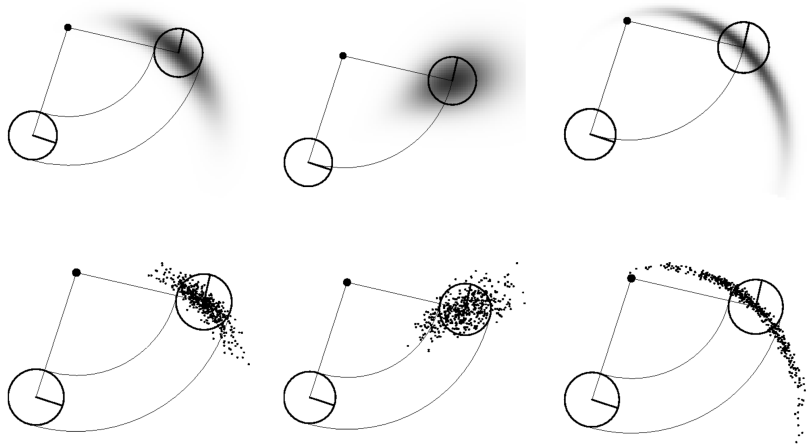


Figure 2.11: Motion model of a robot and the associated error as probability density (top row) and sampling based poses (bottom row). Image source: [BSBA11]

which occurs when the robot is moving forward or backward without rotating. The column at the right displays an error that is influence mostly by rotation, typically found when the robot rotates in place. Naturally, the actual shapes depend heavily on the circumstances (robot hardware, surface type, acceleration & speed, etc.). These errors add up as the robot traverses through the environment, as illustrated in 2.13.

Some robots have different motion models, for example rail-bound robots or CNC cutter. However this thesis focus on robots with the motion model mention above.

2.3.1.1 Particle Filters

Particle Filters work by keeping multiple pose estimates in parallel. Each pose estimate is stored in a *particle*. The particle's pose estimate is updated as the robot moves. Periodically all particles have a score computed, based on how well the sensor data fits the map with respect to the pose estimate. Particles with a "bad" score a discarded and replaced by new particles with a random pose. A key element in this technique is, to select the random poses in a way that the new particles are likely to receive a higher score the next time, for example via Sampling Importance Resampling (SIR) [GSB05]. That way, the pose estimates from the particles are expected to cluster around the true pose of the robot after a few iterations, which in typical applications translates into a few meters of traveling. The pose estimate of the robot can be either adopted from the particle with the highest score, or computed as a weighted average over multiple particles.

As a big breakthrough in robot localization, this principle has been intro-

duced by Fox et al. in [FBDT99] as *Monte Carlo Localization* (MCL). The concept of MCL does not require a special map format and works well with metric, parametric, or feature-based maps alike. Furthermore, the performance of the particle filter can be modified by increasing or decreasing the number of particles, making it easy to adjust to the available computational power as needed.

Fox et al. provided the implementation *Adaptive Monte Carlo Localization* (AMCL) that worked on grid maps and could keep track of the robot's location in (soft) real time. AMCL uses the principle of Markov Localization [SK95] in conjunction with particle filters. The idea is, to track multiple hypotheses about the robots motion and update these with the odometry sensor data. Figure 2.12 shows how the estimated robot poses – starting from the same position – diverge over time as the robot moves in the environment. Discarding estimates where the pose does not fit well with the laser range finder data and resampling near poses that remain is the key technique that makes this approach efficient. AMCL is part of the default ROS navigation stack [Gerb] and sill in use today.

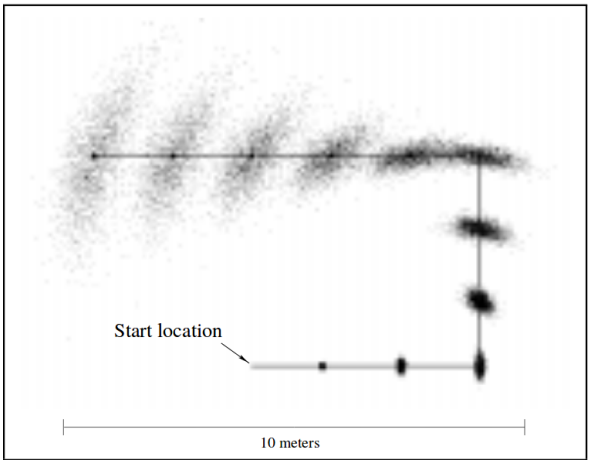


Figure 2.12: An illustration of sampling-based motion estimation. Image source: [FBDT99]

Figure 2.13 shows a particle filter in action. The top left image is right after initialization, and the particles are about equally distributed inside traversable space. The figure in the top right is a snapshot after the robot moved for about one meter. There are now two main clusters of particles, and some random particles around the map. One of the cluster is already around the robot's true position, while the second is not discarded yet due to local similarity of the environment. The third image shows the situation after the robot moved another two meters. Only one cluster of particles is left, and the robot's pose is close to its center.

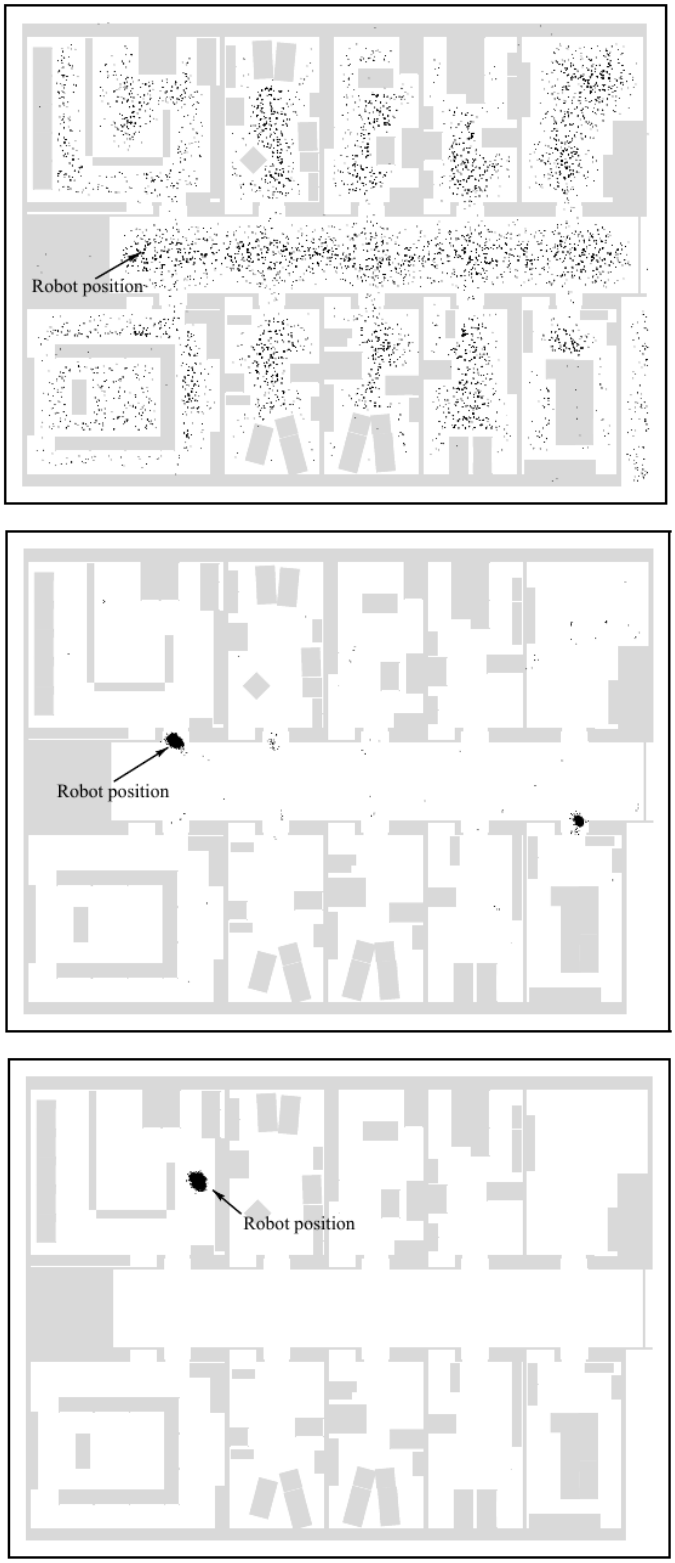


Figure 2.13: An example of localization with a particle filter. At first, the estimates for the robot pose are distributed all over the place (top left figure). After the robot moves around a bit, most estimates are discarded, and only two clusters of pose estimates remain (top right figure). As the robot keeps moving, only one cluster of pose estimates is left, with the true pose located near the center of the cluster (bottom figure). Image source: [FBDT99]

2.3.1.2 Kalman Filters

Kalman filters (KF) are based on the work of Kalman [K$^+$60]. They are filters that work as least square error optimizer. Localization based on Kalman Filters assume, that the uncertainty of the robot's position and the related sensor readings can be represented by a Gaussian distribution. And while these are clearly not Gaussian distributions, the approximation is good enough for our needs in regard of localization. To deal with the nonlinearity of the motion model, Extended Kalman Filters (EKF) [SSC90] and Unscented Kalman Filters (UKF) [WVDM00] are used to linearizes the motion estimate.

Localization itself relies on landmarks when using Kalman Filters. Observed Landmarks are tracked as the robot moves, and the position estimate of both the landmarks and the robot pose are updated with the Kalman Filter to minimize the square error. This method has the disadvantage, that the robot has to reliably sense and identify landmarks from its sensor data. The reason for this is, that landmarks from different keyframes are used to correct both the pose (and trajectory) of the robot as well as the pose of the landmarks. This method is also less scalable with the size of world in comparison to particle filters. Relying on matrix inversions, the KF algorithms have a computational complexity of $\mathcal{O}(n^{2.4})$ or worse, with n being the number of landmarks observed. There are however approaches that manage to reduce the computational load. For example, Gamage and Drummond [GD13] use dimension reduction techniques to deal with a high number of landmarks in their approach. Liu and Thrun [LT03] are using the Information Filter (a variant of the Kalman Filter, where the inverse of the matrix is used instead) to allow for matrix updates in constant time instead of $\mathcal{O}(n^2)$. Cadena and Neira [CN10] on the other hand are using a combined Kalman-Information filter to perform updates faster, taking advantage of both representations.

2.3.1.3 Pose Graph Optimization

Yet another technique to determine the robots pose within its environment is *pose graph optimization* [GKSB10]. Unlike Particle Filters and Kalman Filters, this method is only employed during the SLAM process, and not afterwards on an already constructed map. The concept of pose graph optimization is to build a graph from the robot's trajectory, and link nodes in the graph if they have sufficient sensor overlap. Constraints between connected nodes (in particular those from loop closures) are used in a least-square error minimization process to adjust the pose of the individual nodes and to minimize the global error imposed by edges of the graph. Edges in the graph denote scan matches transformations between keyframes that correspond to connected graph nodes. Libraries like g2o [KGS$^+$11] solve this optimization problem

iteratively, providing new pose estimates with a lower global square error. Pose Graph Optimization depends on edges from loop closures to reduce the global error. Improvements on trajectories without loop closures are rather limited. This method is sensitive to false positives and outliers, which can reduce the map quality significantly, even to the point where the map is no longer usable for its designed task.

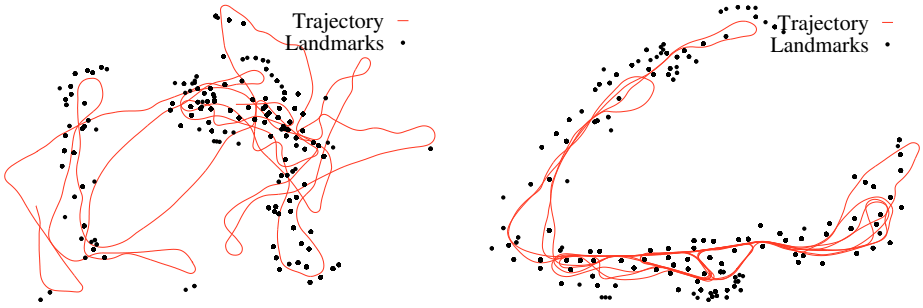


Figure 2.14: SLAM with the victory park data set. The left figure shows landmarks and the trajectory estimate with dead reckoning only. The right figure show the same data but after pose graph optimization has been applied. The optimized trajectory estimate is consistent and matches the actual robot trajectory closely. Image source: [KGS+11]

Figure 2.14 shows the results of pose graph optimization in comparison to no optimization at all. The results are similar to the trajectory estimate from EKF-SLAM that are displayed in 2.7, and are using the same data set. However, unlike EKF-SLAM, graph optimization does not rely on features.

2.3.1.4 Scan Matching

Scan matching is a special case in terms of Localization, as it is by itself not suitable to locate the robot without a good initial guess of the robot's pose. Instead, Scan Matching is used to refine the pose estimate that is provided by other means (which may be even just dead reckoning). Scan matching describes the task of aligning sensor data with the local or global map, typically in a least squared error method. This is realized by using the Iterative Closest Point (ICP) algorithm. It iteratively minimizes the error between points from the sensor scan and points (or vectors) from the map or keyframe(s). Multiple versions of the algorithm have been developed to improve certain characteristics [PCS15, RL01].

ICP works in two steps per iteration. The first step creates data associations between the two data sets. The second step computes a transformation that minimizes the square error for the associated data points. Figure 2.15 shows how two set of points are aligned with each other in three iterations

of the algorithm.

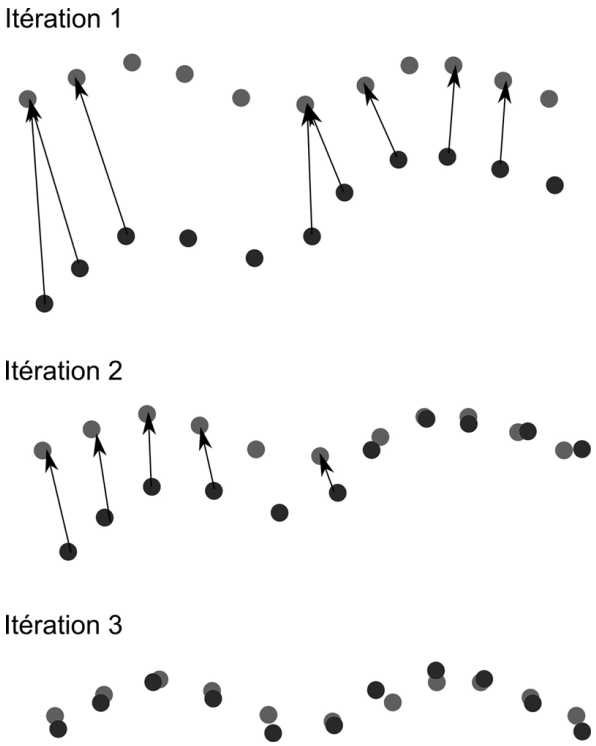


Figure 2.15: Iterative Closest Points algorithm illustrated. Image source: flash-informatique.epfl.ch/spip.php?article2581

As a non-linear optimization method, ICP can be stuck in local minima. It therefore requires "good" data associations, which in return is much easier to achieve if the two data sets already have a decent alignment, i.e. if we have a good initial guess. When aligning the current local map with the previous, the robot odometry is often already good enough for this. However for performing loop closures more care needs to be taken. Methods to improve data association and alignment quality include feature matching [SJH98] point-to-line matching [Cen08] and multi-resolution matching [JH03].

2.4 Mapping

Mapping in robotics refers to the creation of a digital representation of the environment that is usable by the robot, or by humans, or both. For this task we require accurate sensor data, including the position of the sensor(s) at the time of recording. In the context of this thesis, we focus on map formats that are usable by both robots and humans.

2.4.1 Techniques

Mapping typically relies on Localization for accurate pose estimates of the robot and by extension on the mounted sensors. However, mapping has to deal with error accumulation. On the one hand, accurate pose estimates are difficult to provide on incomplete maps. On the other hand, sensor data contains noise. Typically there are two different scopes of errors: local and global.

Local errors occur when estimating the pose delta between the last pose estimate and the current pose. This error can be reduced by utilizing localization techniques as discussed in 2.4.1, as well as by filtering odometry data to reduce noise.

Global errors emerge by accumulating local errors as the robot traverses through the environment. Noise and bias from local errors cannot be completely eliminated, despite best efforts. Global optimization techniques are employed to minimize these errors. These will be discussed in detail in section 2.5.

It is worth noting that many of the techniques below do not rely on a specific map format. Still, the most common map format in use is the occupancy grid. Most SLAM solutions provide only occupancy grids as output format, despite often using a very different internal representation of the environment.

Particle Filters. Being similar to localization-only particle filters in nature, this technique stores a separate map estimate in each particle. Due to the fact that the map and the robots trajectory are linked, the state space does not explode, and hence the number of particles required remains manageable. How this works is explained in more detail by Grisetti et al. [GSB05]. The algorithm uses SIR or similar resampling strategies to discard particles that score low and creates new particles that are likely to score better. Since particles that include (correct) loop closure tend to score better than those without, this technique also implicitly performs loop closures.

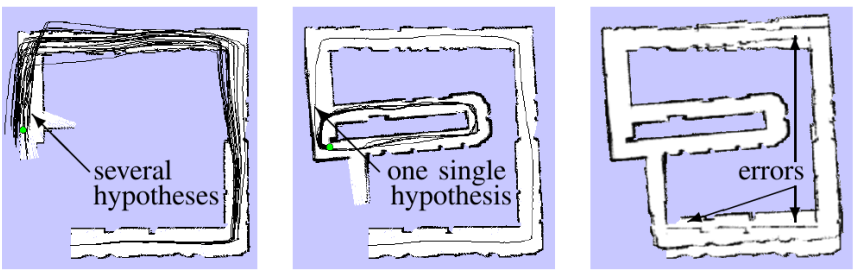


Figure 2.16: Loop closures with particle filters. The first loop closure leads to loss in particle diversity, resulting in failure to perform a second loop closure a bit later. Image source: [SGB05]

Potential problems with this technique lie in particle depletion and the difficulty to perform loop closure on long trajectories [SGB05]. This is illustrated in figure 2.16, which shows on an example how loop closures can cause problems later by discarding particles. In the example the robot starts in the bottom and moves counter-clockwise. The left image shows multiple trajectories, overlaid on top of the map from the top-scoring particle. The robot performs a loop in the center of the map, and a loop closure occurs after the robot re-enters a known part of the map. The center image shows how the diversity of particles is drastically reduced, leaving only particles that share the same history up until entering the loop. Upon further traveling, the robot passes his starting position, as shown in the right image. But due the earlier loop closure, we now have no particles that are able to match a second loop closure.

Kalman Filters. SLAM algorithms that are based on Kalman Filters, such as EKF-SLAM [SSC90] already create a map during localization. The reason for this is that these maps are feature based, and the localization process not only estimates the robot pose but also the location of all observed features. As a result, the position of the landmarks already form the map, and no extra steps are required. The main challenge for Kalman-Filter-based SLAM is identifying landmarks and how the required computation power rises as the map grows.

Composite Maps. The main problem when building large maps is to deal with accumulated errors that are unavoidable in long robot trajectories. Composite maps deal with this problem by creating many small maps instead of one single big map. Small maps only require short travel distances, reducing the possible error accumulation by a big margin. These small maps, often refereed to as *local maps*, are linked together in a graph-like structure. The local maps can overlap and represent a complete metric map, as for example in Cartographer [HKRA16], or have gaps between them as done in HMT-SLAM [BFMG08]. Often a global optimization step helps to keep the map consistent and reduces metric errors in the map.

Pose-Graph Algorithms. Pose-graph-based approaches to the topic record the trajectory of the robot, and link poses in a graph structure while imposing constraints on the graph. Key to this technique is to link poses when revisiting known areas of the environment, a process known as creating *loop closures* [SMD10, KGS+11]. An optimization process can then minimize the global error of the pose graph, and recreate the map with the new pose estimates of the robot's trajectory. There has also been a lot of research about

how to deal with outliers, for example [SP13b, LFP13].

2.4.2 Creating Occupancy Grids

GMapping. Grisetti et al. [GSB05,GSB07] introduced a SLAM solution based on particle Filters and provided an implementation called GMapping [Gera]. Localization in GMapping works very similar to AMCL, but GMapping is a complete SLAM application while AMCL only provides Localization. This adds the challenge to not only handle multiple pose estimates, but also keeping multiple map estimates in parallel.

The algorithm creates one global map per particle, but uses a hierarchical tree structure to store common parts of map estimates to save memory. That way, the memory requirements are feasible for the hardware to handle, even for a relatively high number of particles. The maps all share the same frame, which is the starting frame of the robot. Like all examples of occupancy grids in this section GMapping uses a fixed grid resolution, by default cells of 5x5cm. With its standard settings, GMapping works with relatively few particles: only 30. Modern hardware can handle significantly more however, shall the need arise.

Each particle has a score attached to it, representing how well (according to the algorithm) the map represents the environment. The particle with the currently highest score is chosen as output for services such as navigation tasks. Since the score of the particles changes over time, the map can change in significant ways if a new particle has the top score. This happens often when loop closures occur.

When the mapping process with GMapping is done, the resulting map can be used in conjunction with AMCL for future localization. While GMapping is running, it provides a pose estimate, eliminating the need for 3rd party Localization module.

Karto. The graph-based SLAM implementation Karto [Int, SRI10, SLA] creates an occupancy grid, similar to GMapping. The key feature of Karto is the graph-optimization process that acts as a global optimizer. Graph optimization is also used to provide the robot pose estimate. By actively closing loops, the algorithm builds a sparse graph of spacial restrictions between nearby keyframes. The graph structure is then optimized in an iterative process so that the global (square) error is minimized, smoothing out accumulated errors over multiple keyframes. To do so, the algorithm relies on correct loop closures, as false positives can cause an increase in the global error instead. According to Santos et al. [SPR13], the created maps are better then the compared SLAM algorithms in their paper (including GMapping and HectorSLAM), indicating that the localization also performs better.

Cartographer. A recent SLAM solution that uses Occupancy Grids for relatively large maps is Google Cartographer [HKRA16]. Localization with Cartographer [HKRA16] is a bit unusual in that it does not use a Particle Filter despite its popularity. Instead the algorithm uses its *local* SLAM stack (working with the local map) for initial pose estimate and periodically runs sparse pose graph optimization on the pose graph for a pose optimization. The pose estimate updates between the pose graph optimizations also utilize IMU sensor data when available.

The algorithm creates overlapping local maps (referred to as *submaps*) that are connected with a graph-like structure. Local maps are created from a few consecutive laser scans. While local maps are stored as occupancy grids, Cartographer puts emphasis into treating the grid as a probabilistic grid, in that it models the probability that a grid cell is occupied by an obstacle.

Global coherence between local maps is realized via pose graph optimization, which is performed regularly. For visualization and other tasks such as navigation, a single global occupancy grid is exported in regular intervals. Unlike many other SLAM algorithms, the output is not a three-state map but a "gray-scaled" map that allows grid cells to express the likelihood that the space is occupied.

HectorSLAM. HectorSLAM [KMvSK11], developed at the TU Darmstadt, is a robust SLAM algorithm. It combines multiple sensors (if available), doesn't rely on odometry (but uses it if available) and tracks the robots position in 2D and 3D simultaneously. Localization is based on scan matching, and additional estimates (e.g. by odometry and IMU sensors) are incorporated if provided by the robot.

The SLAM algorithm used by HectorSLAM uses multi-resolution scan matching and Kalman Filters to align keyframes with the global map. The low-resolution scan matching is more robust, and the resulting alignment is then used for the next higher resolution scan matching process. The highest-resolution global map is used as output, providing an occupancy grid for other modules to use. The multi-resolution approach allows to iteratively refine the pose estimate with the goal to provide optimal alignment when the new data is added to the highest-resolution map.

2.4.3 Creating Vector-based maps

TvSLAM. Chen et al. [CQW+17] describe a SLAM algorithm that creates vector maps. They focus on a home cleaning robot system without a laser range finder and instead rely on ultrasonic and infrared sensors for obstacle detection, as well as ultra-wide-band (UWB) receiver and wheel-encoder-based odometry readings for pose estimation. Localization does not rely on the map

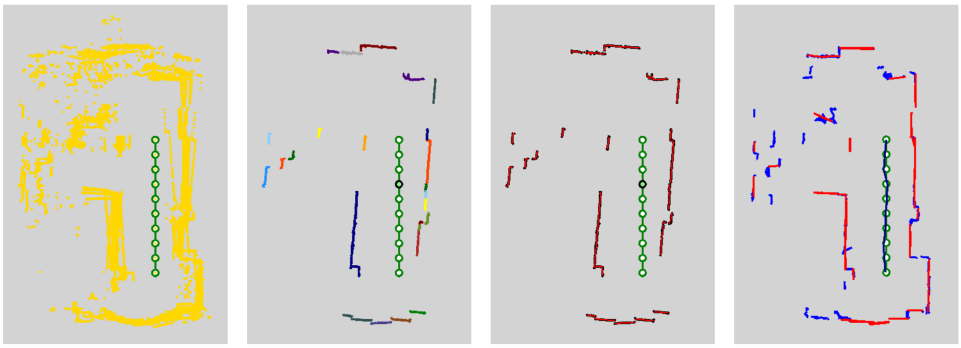
and instead is based on the UWB receiver and dead reckoning. The robot utilizes a wall-following exploration strategy and only collects data points when it turns. Consecutive data points are connected to form vectors, creating outlines of any obstacles. Unexplored areas remain when there are gaps in obstacles that are too small for the robot to enter, however they are not marked as frontiers and instead closed as obstacles. This makes sense for this use case, since the map is used for navigation only (and not for localization). This results in relatively long vectors and filters out most noise, but also ignores small details. The SLAM algorithm also adds topological information that are meant to help with navigation tasks.

VecSLAM. This algorithm by Sohn et al. [SK09] creates a single global map consisting of vectors. Vectors are not necessarily connected with each other, and consequently may contain gaps. Therefore the map itself is not closed, as the borders between explored and unexplored space are not well defined.

The vectors are created from laser scan data, using a sequential algorithm described in [SK08], building a new local map. This local map is aligned with the global map via scan matching, and vectors are merged with existing vector, using a recursive least square filter. Global map errors are reduced by utilizing loop closures.

SLAM algorithm by Jelinek. This SLAM algorithm described by Jelinek [Jel15] provides another open vector map without frontiers between explored and unknown space. Vectors are created from the sensor data via edge extraction. Scan matching with the global map is used to align new vectors. For this, only vectors in the global map that are longer than a given threshold are used, because shorter vectors are deemed to be not determined with enough precision. After this step new vectors are added to the global map. Whenever possible new vectors are merged with existing ones. New vectors that have no corresponding pair in the global map are added unchanged. Figure 2.17 shows the map building process. The left image shows an overlay of all raw data and the robot's trajectory. Next to it we can see a single scan what has been segmented into groups that are considered likely to belong to the same vector. The third image shows the same scan as the previous image, but now the points have been replaced by vectors. The last image shows the complete map, with red vectors being considered long enough for scan matching.

BS-SLAM. Pedraza et al. introduce BS-SLAM [PDM$^+$07], an EKF-based SLAM algorithm that models obstacle shapes with B-Splines and utilizes the control points of the splines as features for the Kalman Filter. B-splines



(a) Raw data (b) Segmentation (c) Line extraction (d) Complete map

Figure 2.17: Stages of the vector map building in the SLAM algorithm by Jelinek. From left to right: raw data, segmentation (showing a single keyframe), line extraction (showing a single keyframe), and complete map. Image source: [Jel15]

in this context are similar to vectors in that both are parametrized curves and are used to model the shape of the environment (i.e. the obstacles). However the splines are not necessarily connected with each other, resulting in general in open shapes. With no frontiers and no clear transition between explored and unexplored space, BS-SLAM is lacking a critical feature. An example map can be seen in Figure 2.18.

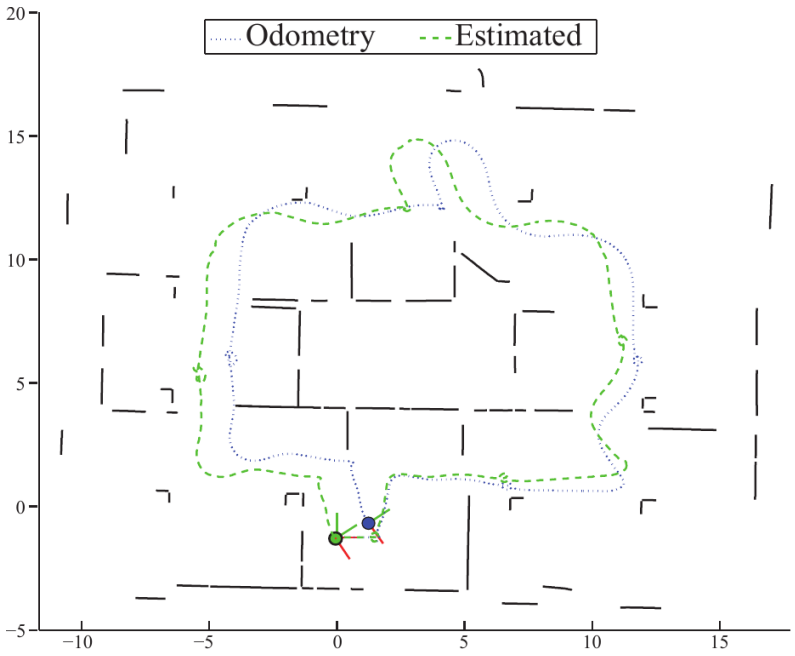


Figure 2.18: Vector map constructed with BS-SLAM. Image source: [PDM+07]

SLAM algorithm by Lakaemper et al. Lakaemper et al. [LLW05] use a vector-based map format for merging maps which may be created by different robots (including different laser scanners). Line segments are built to approximate the laser scan points, but details about this process are not mentioned in the paper. Instead the focus lies on map merging and detecting loop closures via shape similarity measure. Their approach does not rely on odometry. An example of their vector-based map can be seen in Figure 2.19 However like other examples, their map format does not include frontiers, and the shapes don't defined clear transitions between explored and unexplored space.

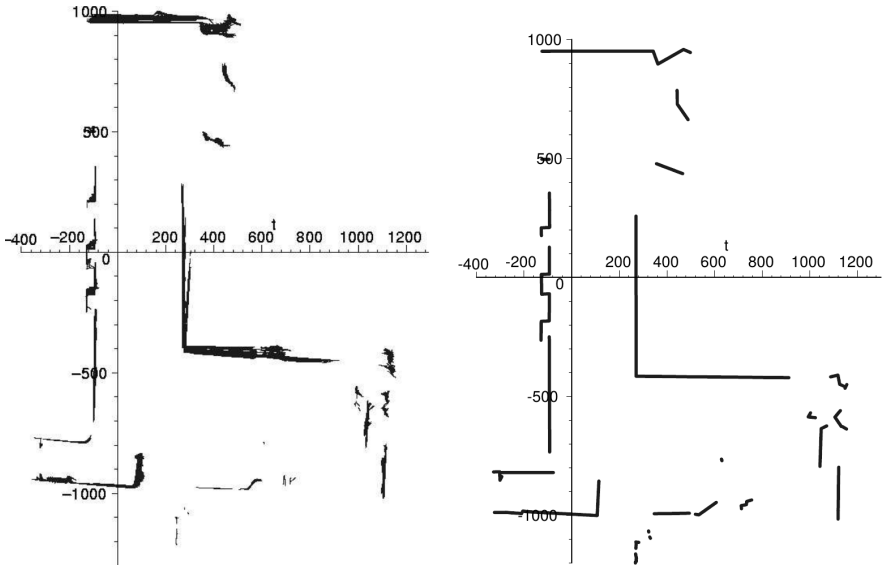


Figure 2.19: A vector map constructed with the algorithm introduced by Lakaemper et al. The left image contains line segments created from 400 laser scans. The right image is after merging of the lines. The unit size in the figures is cm. Image source: [LLW05]

SLAM algorithm by Elseberg et al.. Elseberg et al. [ECL10] are building the map from keyframes. Their keyframes are constructed from vectors without disclosing which line extraction method is used. Potential keyframes are rejected if they show too much overlap with previous keyframes, to reduce the computational load. Alignment between keyframes is computed by global pose graph optimization, utilizing the relative pose of overlapping keyframes obtained by scan matching. Once a global map has been created, clustered vectors (based on their similarity measure) are merged into average line segments that represent the replaced vectors. Finally inconsistent vectors (e.g. introduced by people walking nearby while the robot was recording) are removed from the global map. The map does not contain any frontiers,

and their approach does not scale well on larger data sets (large being 500+ keyframes according to the paper).

VectorAMCL. Hanten et al. [HBOZ] show that the principle of Adaptive Monte Carlo Localization (AMCL) can be used in conjunction with vector maps. They use a beam and likelihood field model as described by Thrun et al. [TBF05]. In simulations they outperform grid-based AMCL solutions in terms of *Absolute Trajectory Error* and *Relative Pose Error* (both translational and rotational errors). Experiments with real robots (utilizing CAD floor plans to create maps) also show improvements over conventional AMCL applications. In particular, they achieved higher accuracy, requires less memory & computation time, and expect to scale well in large scale environments. Strictly speaking, this is not a SLAM algorithm, as only the pose is provided as output.

2.5 Global Error Minimization

One of the biggest problems in early SLAM solutions was how to deal with error accumulation as the trajectory gets longer and the maps grow larger. The main source of error comes from sensor noise (including sensor bias), in particular the laser range finder, odometry, and (if present) the IMU. Other sources of error include dynamic elements in the environment (e.g. pedestrians, doors, other robots, etc.), approximations in mathematical models (e.g. linearization via extended/unscented kalman filters), limits of robot hardware (e.g. uneven robot wheels), discretization of sensor data, and many more.

Overall, there are two steps to minimize the global error. The first step is to reduce the local error. This comes down to estimate the robots pose as accurately as possible. Techniques involved in this step have been discussed in 2.3. In doing this, we reduce the global error bounds by a significant margin.

The second step is to reduce the global error by means of loop closures. Loop closures occur when a robot re-visits a place that is already part of the map. A successful loop closure introduces a restriction that links to poses in relation to each other. This ensures that (the affected part of) the map is coherent and reduces the global error. Loop closures can happen implicitly (for example in Particle Filters) or explicitly as in Kalman Filters and Graph Optimization.

2.5.1 Global Optimization with Kalman Filters

Global Optimization is a direct byproduct from the algorithm, as it minimizes the mean square error along all features and the robot pose. One chal-

lenge with this algorithm lies in feature association, in particular to recognize whether or not a feature is new or has already been observed in the past. In other words, the algorithm relies on loop closures.

Research in regard of global optimization with KF-based SLAM is less focused on further reducing the global error and more on increasing the computation speed by reducing the algorithms complexity class. While doing so (typically) increase the global error, this trade of is acceptable in exchange to allow the robot to work in larger environments.

2.5.2 Global Optimization with Particle Filters

As mentioned above, loop closing in particle filters happens implicitly by having particles that accidentally close a loop to score higher than those that fail to do so. On the one hand, this is convenient, because no further action is required by the algorithm. But the lack of actively finding loop closures is also limiting the ability to perform loop closures in the first place. The general problem in this regard is described by Stachniss et al. [SGB05]. The longer the trajectory before the robot revisits a location, the less likely it is for the particle filter to successfully perform a loop closure. This behavior can be reduced to some degree by parameter tuning, in particular by increasing the number of particles and reducing the threshold that causes premature particle rejection. Overall, particle filters can perform well, especially when loops can be closed early. But they perform badly when they fail to close loops after traveling for long distances.

2.5.3 Pose Graph based Global Optimization

In (pose) graph optimization, local maps (e.g. keyframes) are treated as nodes of a graph. Neighboring maps with a decent overlap are connected via edges in the graph. These edges contain the transformation from one local map to the other. As the graph grows (especially after loop closures) the transformations between local maps become increasingly contradictory in the global context. An iterative optimization algorithm (see [KGS+11] for examples) reduces the overall error by evening out transformation errors between multiple nodes and edges. The resulting graph (and corresponding map) is more consistent than before and the global error is minimized. Figure 2.20 shows this process. Subfigure (b) shows the pose estimate without graph optimization. In (c) the loop closure between pose 4 and pose 1 is detected and added to the pose graph. Finally in (d) the pose graph optimization has minimized the global error, resulting in an pose estimate close to the ground truth from (a).

Like other active loop closing algorithms, this technique is sensitive to

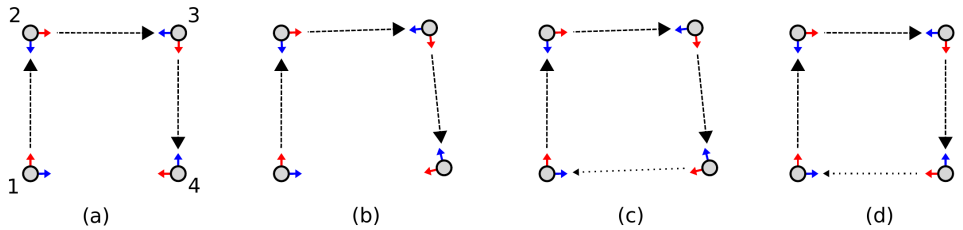


Figure 2.20: Pose graph optimization illustrated on a toy example. Figure (a) shows the ground truth of the robot trajectory. Figure (b) shows the pose estimate which includes a rotational bias. Figure (c) illustrates a detected loop closure between pose 4 and pose 1. Figure (d) is the result of graph optimization with the loop closure in effect.

false positives. Edges in the graph that add false restrictions can cause the algorithm to fail completely and greatly increase the global error. As a result some research focuses on dealing with outliers in the graph structure, for example Sünderhauf and Protzel [SP12]. Other research on Pose Graph Optimization works on reducing the computational load to deal with larger environments, e.g. by sparcification of the graph [HKL13].

2.6 Navigation

Navigation in the context of this thesis refers to the task of path finding a valid trajectory that leads the robot from position A to position B, with both positions provided via coordinates in the frame of a global map. Several strategies have been developed to deal with this, typically revolving around the capabilities of the robot, in particular the range, accuracy, and sensor resolution.

2.6.1 Reactive navigation

Reactive navigation computes the robot movement very quickly based on its current perception. Global knowledge beyond the robots sensor range (such as provided by a global map) is not used. Bug algorithms [MdCT18] for instance, guarantee to reach the target (in a static environment), assuming that the target is reachable at all. They rely only on short-range sensors such as tactile sensors or ultrasound sensors. The strategy involves “wall hugging”, that is moving along obstacles that are encountered on the robot’s trajectory. These strategies are fairly simple, which makes them suitable for autonomous robots that have minimal computation power. Furthermore, they don’t require neither extensive sensor setups nor any sort of mapping or localization. However the lack of global insight often leads to sub-optimal trajectories.

2.6.2 Grid-based Navigation

Several approaches are based on a regular decomposition of the environment into cells such as Occupancy Grid Maps to compute an efficient trajectory from the robot's position to a target area. For example, an approach based on potential field [BLL92] guides the robot with repulsive forces spread from the obstacles and an attractive force directed toward the target. Like other regular-cells-navigation approaches (as based on Markov Decision Process [FT07]), the action performed in a cell depends on an evaluation of neighboring cells. The trajectory planning requires several iterations on the complete map to converge to a stable configuration, driving a robot towards one and only one target. The resulting trajectory could be classified as optimal regarding the cell dimension. However, it requires computation that rapidly become unpractical in large environment or in multi-task and multi-robot scenario.

2.6.3 Heuristic approaches

To speed up computation, heuristic approaches compute paths on a graph (adjacent traversable cells) that is built from Occupancy Grid Maps produced by most SLAM algorithms. A textbook case is provided with the Robot Operating System (ROS) module *move_base*[1] [MEBF+10]. Some well-known path planning algorithms such as *Dijistra*, A^*, *Navfn*, *etc.* algorithms are employed on a graph generated from a Grid Map where obstacle shapes are inflated. This way, the approach returns the shortest path to a target position while the robot keep safe distances to obstacles.

These approaches based on converting Grid Maps usually lead to generate large and hard to process graphs making path planning slow. Approaches based on Rapidly exploring Random Tree [KL00] tackle this problem by performing the A^* algorithm while iteratively building a tree (instead of a graph). At each iteration, random movements toward the target position generate new reachable positions from the most promising ones. In case of a dead end (all random movements fail because of obstacles in the map), a backtrack mechanism is activated. The algorithm ends when the target is reached or when no more position in the tree can be extended.

2.6.4 Topology-based Navigation

In general, topology-based navigation (as proposed in [LAB+12]) relies on an hybrid approach based on a reactive and a deliberative architecture.

Deliberative modules control the local behavior on a high level, e.g. by activating or deactivating specific local behaviors, based on the current sit-

[1] `https://wiki.ros.org/move_base` - Author: Eitan Marder-Eppstein

uation. They are also responsible for representing the position of the robot and the path to the goal. As such, there task is to path planning on the level of the topological map itself Reactive modules are more focused on the local environment of the robot, and reaching nearby topological nodes. They are dealing with dynamic elements in the immediate area around the robot and how to avoid collisions with obstacles.

Ideally, the topological map has nodes in all key areas, while also minimizing the total amount of nodes for fast and efficient computation when performing path planning. In this context, nodes represent areas in the environment that are traversable by the robot, and edges in the graph connect two nodes if, and only if, they are reachable. The meaning of "reachable" varies from case to case, but typically encompasses, that a node can be reached (e.g. by a robot) by only providing the two nodes in question without relying on any other topological nodes. In some cases, this means that the two nodes must have a direct line of sight, in other cases there may be less restrictive demands to be satisfied.

Topological maps can be generated from metric maps such as occupancy grids as a *Voronoi* [Thr98] or a Visibility Graph [Wel85]. They can also be directly created without such intermediate formats and used for navigation, as shown for example by p [Kui00].

2.7 Comparison of 2D SLAM Techniques and Solutions

In the previous sections we took a closer look at the various SLAM techniques and algorithms. In this section, we compare the general techniques or selected algorithms against each other. There are some cases where we only compare the general techniques, because individual algorithms that are based on the respective technique all share the same characteristics in regard of the comparison criteria.

2.7.1 Comparison Criteria

When comparing different SLAM solutions for exploration, there are a number of criteria that we are interested in.

- **Localization Accuracy:** How accurate is the pose estimation within the created map. This depends on the map quality, the sensors, and the localization algorithm itself.
- **Mapping Accuracy:** Consistency and accuracy of the created map. This is also influenced by the map format, for example occupancy grids are bound to a fixed resolution.

- **Frontiers:** Whether or not the map contains frontiers that can be used for (autonomous) exploration. Frontiers can be included in the map in an explicit manner, or implied in an implicit way, or be completely absent.
- **Well Defined Borders:** Whether the the transition from traversable space into obstacles / unexplored space are well defined (closed map) or the borders between explored and unexplored space are not always clear. Occupancy grids are always closed, feature-based maps are never closed, some vector maps are closed.
- **Usable for Precise Human Navigation:** This is mostly about how well the map can be visualized for human use. In particular how well the shape of the environment is presented to human users.
- **Suitable for Autonomous Navigation** To satisfy this criteria we need to be able to create a topological graph from the map. Without such a graph we are unable to perform efficient global-scale path planning or determine if a target location is reachable at all from the robots current position.

2.7.2 Comparing Techniques and Map Formats

In this section, we compare the basic techniques and map formats with respect to the comparison criteria mentioned above.

2.7.2.1 Localization Accuracy

The following paragraphs evaluate the impact of different map formats and algorithms on the localization accuracy. Factors such as sensor quality and external influences are ignored, unless they have a significantly different impact between the compared approaches.

Occupancy Grids. While the quality of the map itself has a very big impact on the localization accuracy, modern SLAM algorithms can create fairly accurate occupancy grids of moderate size environments, for example inside buildings. This map format imposes a uniform and predefined resolution on the whole map. Theoretically the map resolution limits the possible accuracy for localization. However the most common resolution for grid maps (5x5cm) seems to be good enough for most real world scenarios. Ultimately the accuracy also highly depends on the localization method used. Kümmerle et al. [KSD+09] and Vincent et al. [VLE10] show that among the tested algorithms, pose-graph-based localization provides the highest accuracy. However, Röwekämper et al. [RST+12] show that Particle Filters can deliver good

results, even in dynamic environments. All tests in the cited papers were performed on occupancy grids with a 5x5cm resolution.

Feature-based Maps. Similar to other map formats, the quality of the map has a big impact. Hand placed beacons with known location are even used to provide a quasi-ground-truth source, as the error boundaries are typically not only known, but also fairly small and not subject to bias or drift. The same cannot be said about feature based maps that were built in a SLAM fashion.

The issue with feature-based maps is, that the higher the accuracy (of the map and the pose estimate) is, the more observable (and identifiable) landmarks are within range of the robot's sensors at any given time. However increasing the feature density shrinks the possible maximum area of the map, since the computational load largely depends on the total number of features in the map. As a result, there is a trade-off between map size, accuracy, and computational load that needs to be adjusted the robot's capabilities and the general scenario. The work of Paz et al. [PJTN07] shows how different EKF-based algorithms balance between the number of features and accuracy.

Vector Maps. While vector maps have the potential for high accuracy localization, there are only few modern implementations that create them. Overall, the quality of vector maps of reasonable large environments is inferior to occupancy grids or feature-based maps among vector maps introduced in papers that we found on the subject, such as [CQW+17, Jel15, SK09] and others. Though at least part of the reason for this is the lack of global optimization in many of the vector-based SLAM implementations.

However, given an accurate vector map of the environment, the expected accuracy of the pose estimate is higher than in other map formats, assuming that the environment can be reasonably well be approximated by vectors. The reason for this is, that the vectors are not bound to a fixed grid, allowing the vectors to approximate the environment better than occupancy grids. Compared to feature-based maps, vectors cover a larger area than landmarks, allowing the robot to use more sensor data for the localization process. This however is only an advantage if the environment can indeed be well approximated with vectors.

Topological Maps. Comparing topological maps to the other formats is difficult, because localization in topological maps often doesn't aim for precise pose estimates and is more about linking a graph node to a general area of the map. Therefore we will not look deeper into this map format in the context of localization or accuracy.

2.7.2.2 Frontiers

Map formats may specify frontiers to model the boundary between traversable space and unexplored space. Unlike borders, frontiers are not obstacles, but may be treated as such during navigation tasks, depending on the scenario and environment. Exploration greatly benefits from the presence of borders in the map, as it allows better planning and prioritization of different areas in the environment.

Occupancy Grids. The classic occupancy grid is initialized as unknown space. Cells are marked as free/occupied space as the robot traverses in the environment and provides sensor data. In maps like this frontiers are built implicitly and don't need to be formally added to the map format. The reason for this is, that frontiers exist whenever an unexplored cell in the grid is neighboring a free cell. Still, for the sake of faster computation during exploration tasks, frontiers may be explicitly added to the map and updated along with the normal SLAM process [Yam97].

Feature-based Maps. The nature of feature based maps leads to a complete absence of frontiers in the map format, and therefore in the maps itself. As a result, if frontiers are required, we need either a hybrid map format or a second map that includes frontiers. For example Tao et al. [THSW07] are using EKF-SLAM (witch uses a feature-based map) for the Mapping and Localization, but maintain an occupancy grid to compute frontiers.

Vector Maps. As mentioned earlier, most vector map formats that we found are open maps with no support for frontiers. And the only implementation that creates a closed vector map (Chen et al. [CQW+17]) uses a wall-following strategy for exploration and assumes that there are no unexplored areas left when finished, hence lacking any need to implement frontiers. Aside from special cases as above, a vector map that is closed (and doesn't assume that everything has been explored) should require frontiers to maintain its closed shape. We did not find any prior work where a vector-based SLAM algorithm creates maps with embedded frontiers.

2.7.3 Comparing Solutions

In this section, we discuss the different solutions that can provide very different map results as shown in Figure 2.21 depending on the techniques they use.

GMapping. One of the older and yet also one of the most popular SLAM implementation to this date is GMapping [Gera, GSB07]. It combines the idea of AMCL localization (as described in [FHL+03]) with mapping techniques and is one of the first demonstrations of effective loop closing in realistic scenarios.

To achieve this, GMapping employs several techniques. It adapts the use of particle filters from AMCL. In addition to a pose estimate, each particle also holds it own estimate of the environment, i.e. its own occupancy grid. By sharing common history between particles, the memory footprint per particle is reduced, which in return allows for more simultaneous particles to be used. The particles also perform loop closing, which results in consistent maps of the robots environment. The map itself is a three-state grid map, an example of a map created with GMapping is shown in figure 2.5.

Karto. The open source SLAM implementation Karto creates occupancy grids, similar to GMapping. While GMapping is using a Particle Filter for global optimization, Karto applies a pose graph optimization instead. According to Santos et al. [SPR13], Karto tends to produce better maps than GMapping. The resulting map is a three-state grid map, same as GMapping.

Cartographer. The newest SLAM algorithm in this list that produces occupancy grids is Cartographer [HKRA16]. It uses pose graph optimization to create consistent maps. While the results can be of high quality, even in large-scale environments, it relies heavily on its parameters being adjusted to the environment [LFBL18]. Internally, the algorithm handles multiple small maps instead of one single large global map. The maps are linked together in a graph structure. The map output of the algorithm is still a single large occupancy grid. However unlike GMapping and Karto, the maps produced by Cartographer are not three-state grid maps. Instead the cells contain values in the range of $[0, 1]$ (quantized into a byte with the range $[0, 255]$ for compatibility) with the value depicting the probability whether the cell is free or occupied. An example map from Cartographer is shown in Figure 2.6

VecSLAM. Sohn and Kim [SK09] introduced a vector-based solution called VecSLAM. They use a sequential segmentation algorithm described in [SK08] to create line segments from the point clouds. The keyframes are aligned with the map, and similar line segments are merged using a recursive least square filter for robustness. For the purpose of global optimization, a topological map (based on the robot's pose graph) is maintained parallel to the vector map. On detected loop closures a Weighted Error Distribution (WED) is used to minimize the error in the affected part of the topological map, and

the map is rebuild.The algorithm provides overall good results both in simulations and in an experiment with a real robots. However, the created map does not contain any frontiers, making it unsuitable for exploration tasks and of limited use for general navigation tasks.

SLAM algorithm by Elseberg et al. Elseberg et al. [ECL10] describe a vector-based SLAM algorithm that utilizes global optimization via pose-graph optimization. The algorithm first reduces the input data to keyframes, rejecting data that has to much overlap with the previous keyframe. In the next step global optimization improves the alignment of keyframes. Keyframes are then merged into the global map, and clusters of overlapping vectors are merged into fewer larger vectors. A final cleanup step removes inconsistent vectors. The resulting vector map (Figure 2.21 has a clean look to it, but contains small yet visible errors. The map does not support frontiers and consequently is not closed either.

SLAM algorithm by Jelinek. Another recent development comes from Jelinek [Jel15]. He describes a 2D SLAM solution with a focus on providing a good approximation of the environment, using a vector-based map format. The map is build up from keyframes. A keyframe holds a point cloud, on which an edge extraction algorithm is used to create a set of vectors that reflect the geometric structure of the environment. The first keyframe is used as the initial map, consecutive keyframes are merged to expand the map. In the process, overlapping and nearby vectors with similar alignment are merged, vectors that are shorter than a preset length are removed. Localization is based on dead reckoning (via inertial sensors) as initial guess and line fitting between the keyframe and the map afterwards. This approach is comparable simple since the edge extraction algorithm just looks for clusters of points that lie on a line. In return the whole process is relatively fast, and the implicit noise filtering from the line extraction is dealing with the sensor noise. A disadvantage is, that there is no distinction between traversable space and unexplored space, since only observed borders are stored in the map. As a result the map does not model frontiers, which would be required exploration tasks. However the map can still be used for localization since it contains metric information in the vectors. Navigation would be limited to paths already traveled, but was not considered in the paper. Another disadvantage is, that the line extraction algorithm needs sufficiently large straight surfaces, having it fail to approximate surfaces with a strong curvature. This work lacks global optimization. Examples can be seen in Figures 2.8 and 2.17.

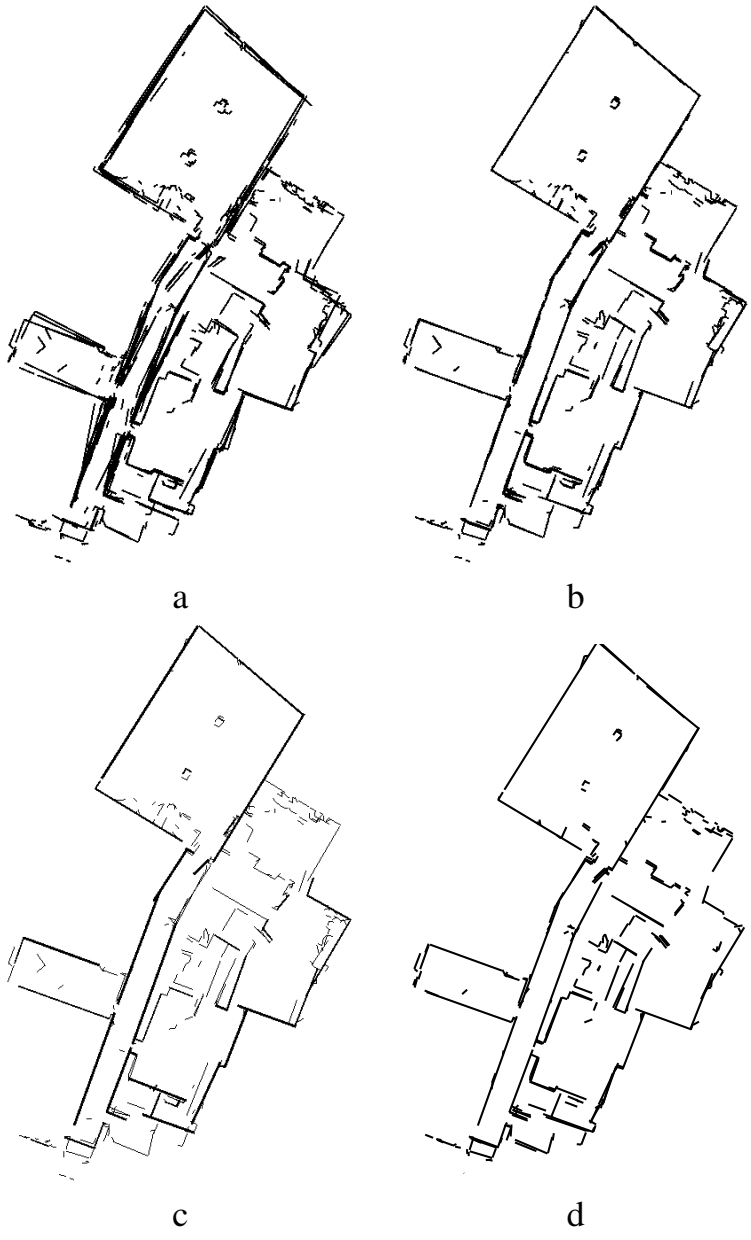


Figure 2.21: Example of a map created from the public data set *Freiburg 082*, using the SLAM algorithm by Elseberg et al. (a) shows aligned keyframes before global optimization, (b) after global optimization. In (c) we have the merged map and (d) shows the finished map. Image source: [ECL10].

TvSLAM. A recent example of vector-based maps can be found at [CQW+17]. Chen et al. are studying a home-cleaning-robot platform with very limit sensor range of 3m. The robot is equipped with 8 infrared range finders and a single ultrasound range finder. To compensate for the limited sensor range, they also use a UWB (ultra-wide-band) radio receiver. The robot also has wheel encoders and an IMU, which allow for dead reckoning localization. The combined use of dead reckoning and absolute position estimate via radio beacons allows the algorithm to determine its current pose well enough that global optimization is not needed. The exploration strategy is a simple wall following algorithm witch is not hindered by the short range of its infrared sensors. Intermediate map results are not discussed in the paper, and the finished result is assumed to cover the whole area, meaning that the map has no need for frontiers. Figure 2.22 shows a map created in a simulation.

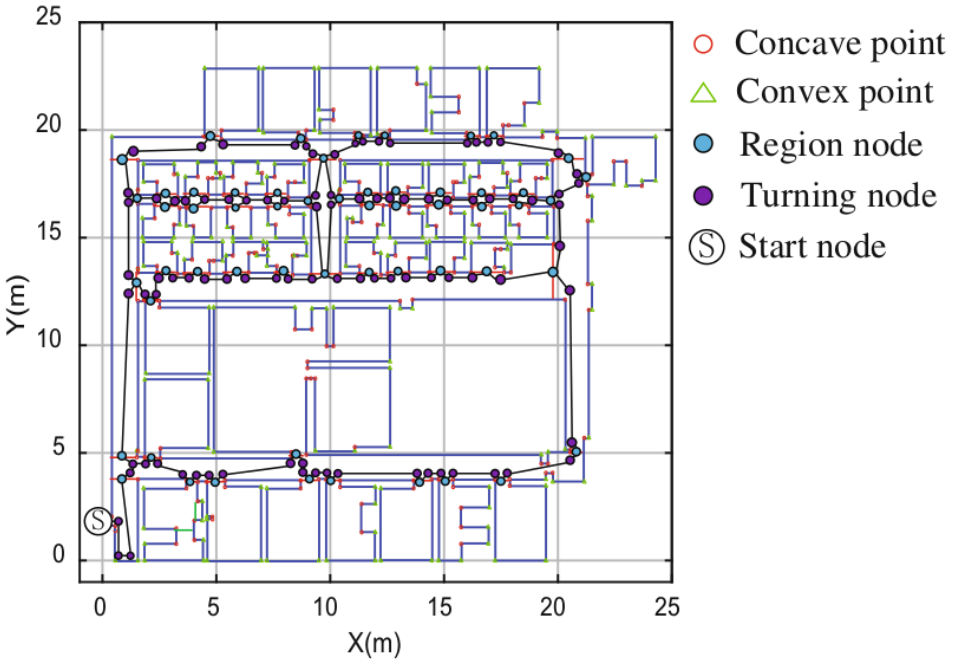


Figure 2.22: A vector map created with TvSLAM. Image source: [CQW+17]

[2]Map accuracy and Localization accuracy combined, since the two depend on each other.
[3]Frontiers are implicit in this map format and can be computed if needed.
[4]Results depend heavily on parameter tuning. We assume good parameters in this table.
[5]Not comparable to other results because no implementation is available to us test the algorithm on a common data set.
[6]Localization accuracy is great, but requires pre-installed radio beacons in the environment.

	Map format	Accuracy[2]	Frontiers	Closed	Visualiz.	Navigation
GMapping	grid	good	no[3]	yes	good	yes
Karto	grid	good	no[3]	yes	good	yes
Cartographer	grid	great[4]	no[3]	yes	good	yes
VecSLAM	vector	n/A[5]	no	no	good	no
Elseberg	vector	n/A[5]	no	no	good	no
Jelinek	vector	n/A[5]	no	no	good	no
TvSLAM	vector	n/A[5,6]	no	yes	good	no

Table 2.1: Overview of the discussed SLAM algorithms and their features or lack thereof.

2.7.4 Conclusion

Based on the comparison criteria in Section 2.7.1, summarized in Table 2.1, we have found no existing SLAM algorithm that satisfies all desired features. Occupancy grids, while widespread, quickly grow in size to a level where sharing them over a wireless connection becomes problematic. Their lack of explicit frontiers makes exploration harder and requires periodic recomputation of explicit frontiers necessary.

Feature-based maps are lightweight in comparison, but don't necessarily model the shape of obstacles well. Indeed, obstacles are often modeled as shapeless points in space. The feature-based map format also fails to mark transitions between explored and unexplored space, which is a major drawback for navigation and path planning.

The vector-based map formats are lightweight in nature, but none of the proposed formats we found were supporting frontiers. Furthermore, most implementations produce open maps. Only TvSLAM creates a closed map, but still does not support any frontiers.

2.8 Summary

The current state-of-the-art map formats and SLAM algorithms are not particularly well suited for multi-robot exploration as no map format ticks all marks on the requirements outlined by us. While among the three major map formats that we examined, each had its strong points, every single one also had aspects that were undesirable. Consequently, we did not find a SLAM algorithm that we consider well suited for multi-robot exploration tasks. This justifies our work on developing a new map format and SLAM algorithm with Navigation support, which we will introduce in the following chapters.

PolySLAM: A 2D Polygon-based SLAM Algorithm

Chapter 3

Contents

Introduction

Vector-based SLAM provides a lightweight map format like feature-based SLAM, but also models the shape of obstacles like grid-based SLAM approaches. Modeling the world with vectors also allows to reduce the impact of sensor noise by averaging over a subset of sensor data. As we have established in Chapter 2, vector-based map format are not common, and none of the formats known to us fulfill our requirements. In this chapter, we introduce PolySLAM, a novel vector-based SLAM that meats our requirements for our SLAM algorithm as outlined in Chapter 2.

PolySLAM relies on PolyMap, a polygon-based map format. In the following, we first introduce PolyMap, before discussing building blocks of the PolySLAM algorithm.

3.1 The PolyMap format

3.1.1 Model

PolyMap represents the environment using simple polygons. *Simple* polygons are non-self-intersecting closed polygons [Grü13]. Our polygons model only explored space, i.e. the inside of all polygons is traversable space. This

happens naturally in the PolySLAM algorithm, as the map is built up from keyframes that all share this characteristic by construction, and carries on as merging keyframes preserves this feature. All the space that is not modeled in PolyMap is unknown/unexplored space. Obstacles are created by (multiple) polygons enclosing the obstacle.

With this, a PolyMap M is defined as:

$$type: \quad T = \{obstacle, frontier, sector\} \tag{3.1}$$

$$vector: \quad V = \{(a, b, t) | a, b \in \mathbb{R}^2; a \neq b; t \in T\} \tag{3.2}$$

$$polygon: \quad P = \{v_i | i \geq 3; v_i \in V\} \tag{3.3}$$

$$PolyMap: \quad M = \{p_i | 1 \leq i \leq n; p_i \in P\} \tag{3.4}$$

with p_i being a *simple* polygon, and n being the number of polygons forming the map.

We define three types of line segments :

Obstacles: they represent the outline of obstacles.

Frontiers: they model the transition from free/traversable space to unknown/unexplored space.

Sectors: they delimit a traversable transition between the polygon the vector belongs to and another adjacent polygon.

Every line segment has exactly one type, but a polygon can contain line segments of different types. The direction of line segments is chosen, so that clockwise oriented polygons contain traversable space inside. Figure 3.1 shows an example of map containing two different types of line segments: red line segments represent obstacles and yellow line segments represent sectors.

Using directed line segment like this, has the advantage that it is easy to determine whether a given point is inside free or unexplored space; a property that feature-based and vector-based map formats are missing. For this, we only need to find the closest line segment to the point and look at which side of the polygon the point is located. Figure 3.3 illustrates this with two points: one is inside the explored area and the other is outside. This not only helps for visualization, but also allows localization with particle filters to discard particles that are inside obstacles or otherwise in unexplored space.

One characteristic of the PolyMap format is, that it only models explored space, i.e. the inside of all polygons is traversable space. This happens naturally in the PolySLAM algorithm, as the map is built up from keyframes that all share this characteristic by construction. Moreover, merging keyframes preserves this property. Being able to break up a polygon into smaller parts also has the advantage that partial maps of the environment (e.g. parts that

changed recently) are supported in this map format. This is advantages in situations where we need to share map updates with other robots or human operators, but have limited bandwidth available to do so.

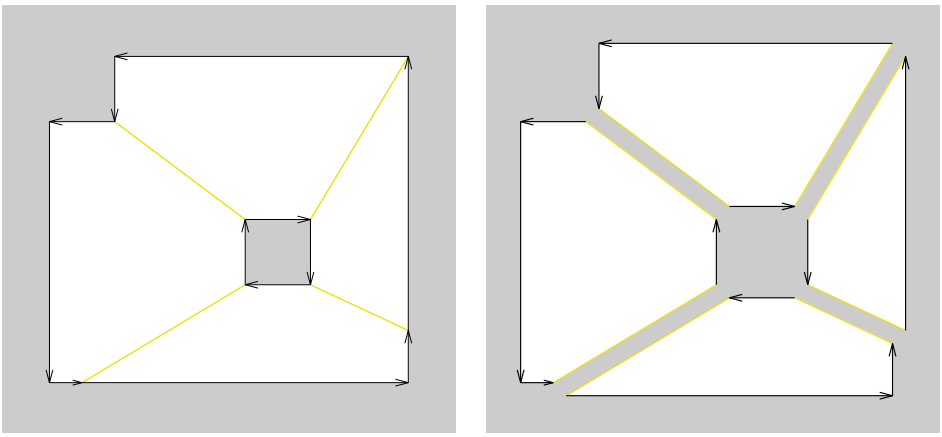


Figure 3.1: A PolyMap example. The left figure shows the map itself, the right figure has the polygons moved apart to show the four individual polygons that make up this map.

3.1.2 Evaluation of PolyMap

PolyMap does address all the requirements presented in Chapter 2. Being based on vectors grouped into polygons, a PolyMap has by definition the advantages of vector-based map formats. It is lightweight and can be easily visualized by humans. The remaining of this section first discusses how PolyMap meets the two other requirements and then how to build polygon-based maps.

Explicit Exploration Frontiers. The types of line segments of a polygon are either obstacles, frontiers, or sectors. Outside a polygon is unexplored space, while inside represents the explored space. Potentially traversable transitions from explored to unexplored space are represented by vectors of type frontier.

Support for Path Planning. PolyMaps are suitable for navigation because polygons separate traversable areas and unexplored ones. The sparse nature of this map format makes it easy to create a topological graph (explained in more detail in Chapter 4), for example by using visibility graphs, or random sampling points in traversable space [LaV06]. Such topological graphs are then suitable for navigation and path planning.

How to build a PolyMap. To validate our PolyMap format before implementing a compatible SLAM algorithm we had to build actual PolyMaps out of occupancy grids. Doing so can be achieved in several ways, for example by extending the approach of Baizid *et al.* [BLFB16] to form closed polygons. Another possibility would be to utilize a wall-following strategy to outline the borders of traversable space, creating closed polygons in the process. Yet another method is to create a vector for every transition from free space into non-free space, and build polygons from the collected vectors. We used this idea in our implementation () to build PolyMaps from grid maps. The pseudo-code for this is shown in Algorithm 1.

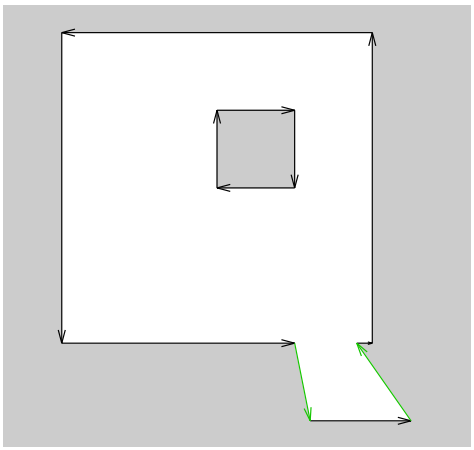


Figure 3.2: A map consisting of only simple polygons. Obstacles are black and frontiers are green.

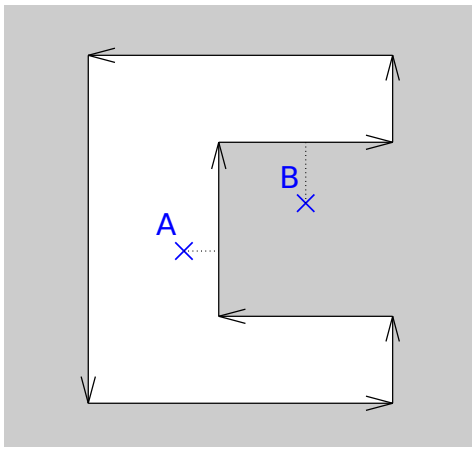


Figure 3.3: Two points (marked in blue), one inside free space, the other in unexplored space.

3.2 Overview of PolySLAM

In this section, we introduce our PolyMap-based SLAM algorithm PolySLAM. The algorithm, as depicted in Figure 3.4 can be split into 5 modules/stages: *data acquisition*, *data alignment*, *keyframe creation*, *refinement*, and *merging*. The *Data Acquisition* module creates and pre-aligns a *pointcloud* from *laser scan*, *odometry*. In *Data Alignment*, the pre-aligned pointcloud is aligned with the previously built global map. The alignment is performed via point-to-vector ICP and additionally provides an update to the robot pose estimate. The now aligned point cloud is converted into a polygon in the *Keyframe Creation* module, and then refined in the *Refinement* module to reduce the number of vectors. This polygon can now be merged into the global map in the *Merging* module, extending the robot's knowledge about its environment.

Algorithm 1: Creating a PolyMap from an Occupancy Grid

```
Data: occupancy grid map G
Result: PolyMap M
begin
    create dictionary D
    /* create vectors                                          */
    foreach cell c ∈ G where c is free space do
        foreach neighbor n of c where n is not free space do
            /* vector orientation is CCW with respect to
               c                                               */
            if n is obstacle then
                create vector v at border of c and n of type obstacle
            else if n is unexplored then
                create vector v at border of c and n of type frontier
            end
            add vector v to dictionary D with v start point as key
            in case of two vectors sharing the same start point, both are
             stored alongside
        end
    end
    /* create polygons                                         */
    while D is not empty do
        create empty polygon P
        take random vector v from D
        remove v from D
        w := v
        while w end point ≠ v start point do
            P add w
            w := D at key w end point
            remove w from D
        end
        M add P
    end
    /* aggregate vectors                                       */
    foreach polygon P ∈ M do
        foreach vector v ∈ P do
            w := vector in P where w start point = v end point
            if w orientation = v orientation and w type = v type then
                set v end point to w end point
                remove w from P
            end
        end
    end
end
```

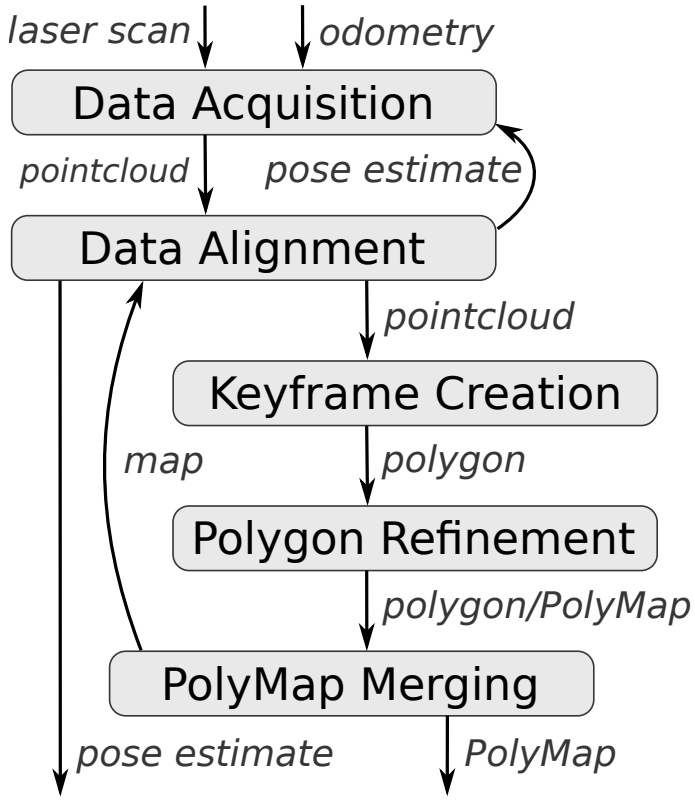


Figure 3.4: Overview of the PolySLAM algorithm pipeline. The input of the algorithm is at the top (laser scan & odometry), the output at the bottom (pose estimate & map). The different stages are highlighted in gray, with the arrows showing the data flow and the data types involved.

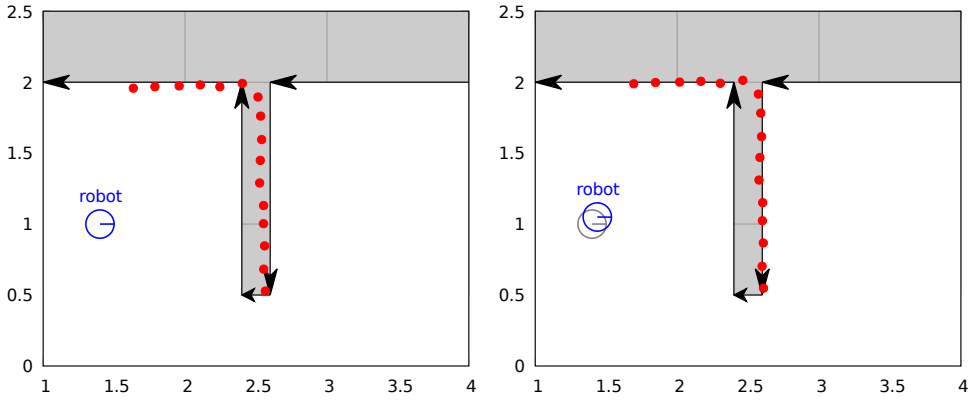
3.3 Data Acquisition and Alignment

We use a single laser sweep of the robot's laser range finder and the odometry reading associated with that time instance in the process of creating a keyframe. As an intermediate step, the algorithm produces a pre-aligned point cloud by converting the range readings from the laser scanner into a point cloud and transforming the pointcloud from the robots (or rather the sensor's) local frame into the global frame by utilizing the pose estimate of the last keyframe and the delta odometry since the last keyframe. In the next step we align the resulting pointcloud with the previously recorded obstacles (not frontiers) that are placed in the global map by using a point-to-vector Iterative Closest Point (ICP) algorithm. We only consider vectors that are of type obstacle *and* are facing towards the sensor center, practically performing backface culling[1]. That way our ICP algorithm wont try to align the points with for example the other side of a wall. While point-to-vector ICP (and point-to-plane

[1]Backface culling is a a term used in 3D rendering where graphical elements are not rendered if the normal vector is pointing away.

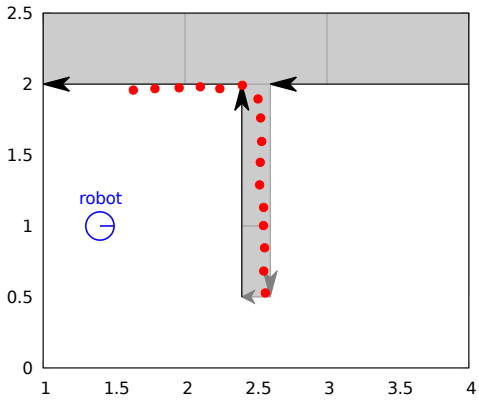
ICP in 3D) by itself is not new [RL01, Cen08], we have not found previous work of ICP that uses backface culling with 2D points/vectors.

The impact of backface culling is illustrate in Figure 3.5. The top left image show the original (pre-)alignment of the point cloud. In the top right image the point cloud is stuck in a local minima, and unable to get closer to the true pose of the sensor/robot. When backface culling is active, obstacles whose's normal is pointing away from the robot are no longer considered for matching (bottom left image). The bottom right image shows the alignment with backface culling, and has a much better alignment as a result. The pseudo code showing this is displayed in Algorithm 2.

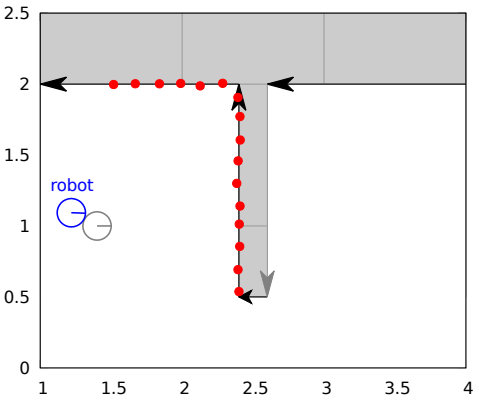


(a) initial setup

(b) result without backface culling



(c) setup with backface culling



(d) result with backface culling

Figure 3.5: Alignment of a laser scan with the global map. Image a) shows the initial pose of the robot (blue) and the laser scan end points (red). Image b) shows laser end points matched to the wrong side of the wall, caused by the ICP algorithm being stuck in a local minimum. Image c) shows the setup with backface culling active. Obstacles that are no longer considered for matching are colored gray. Image d) shows the point cloud aligned correctly.

Algorithm 2: Computing the transformation matrix TF that aligns the pointcloud pc with the map

Data: PolyMap: map; pointcloud: pc; sensor center S
Result: Transformation TF
begin
  3x3 transformation matrix: $TF := identity$
  `/* outer loop: icp iterations */`
  for $i := 1 \rightarrow$*maxIterations* **do**
    Create Collections A_1, A_2
    `/* inner loop: determine inliers and compute` ΔTF `*/`
    foreach *point* $p \in pc$ **do**
      $p' := TF \cdot p$
      `/* point-to-vector nearest neighbor search with backface culling */`
      find nearest obstacle $v \in map$ for point p' so that:
        corresponding point: c for $\{v, p'\}$ **and**
        angle between $\overrightarrow{S, c}$ and v is $\leq 90°$
      if *distance c to* $p' \leq$ *outlierThreshold* **then**
        add c to A_1, add p' to A_2
      end
    end
    compute centroid C_1 as average of A_1
    compute centroid C_2 as average of A_2
    $A'_1 := A_1 - C_1$; $A'_2 := A_2 - C_2$
    compute 2x2 matrix $H := [\sum A'_1, \sum A'_2]$
    compute singular value decomposition: $U \cdot S \cdot V := H$
    rotation matrix: $R := V \cdot U^T$
    translation vector $t := (R \cdot C_1) + C_2$
    ICP transformation step: $\Delta TF := \begin{bmatrix} R & t \\ 0\ 0 & 0 \end{bmatrix}$
    update TF: $TF := TF \cdot |DeltaTF$
  end
end

3.4 Creating Keyframes from Point Cloud

Keyframes represent the local environment of a robot at a single point in time (in practice a very short time interval) and each are modeled as a simple polygon. Creating the polygon for a keyframe is straightforward, with the three major steps shown in Figure 3.6. We compute the points of the aligned point cloud and use these to create the vectors that build the polygon. The last point from the aligned cloud is connected with the sensor center, and from there to the first point, forming a closed polygon. The vector types are determined by the length of the vector: those that are shorter than a predetermined threshold are considered obstacles, longer ones are frontiers. Vectors that are connected with the sensor center are always of type frontier. Invalid laser readings (e.g. NaN) are treated as close range readings (e.g. the robot radius) and result in frontiers as well. We do this, because we can safely assume that the space occupied by the robot is traversable, and this way we can mark more space as explored.

The resulting polygon is always *simple* and *closed*, its vectors model either frontiers or obstacles. This ensures that our starting map (the first keyframe) is also closed.

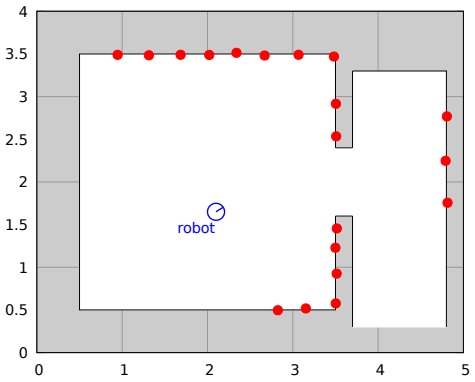
3.5 Polygon Refinement

Creating a raw keyframe results in a polygon with an unnecessary high amount of vectors. In particular in indoor environments, a lot of obstacles such as walls can be described by only a few yet relatively long vectors. To make use of this, we reduce the number of vectors in the polygons whenever possible. This is done by aggregating vectors that (1) all share the same type, (2) approximately lie on the same line, and (3) are connected with each other. The resulting polygon typically contains about an order of magnitude less vectors while still providing a good approximation of the environment, as shown in Figure 3.6. A side effect of this procedure is, that sensor noise is reduced as the newly created vectors average over the noise.

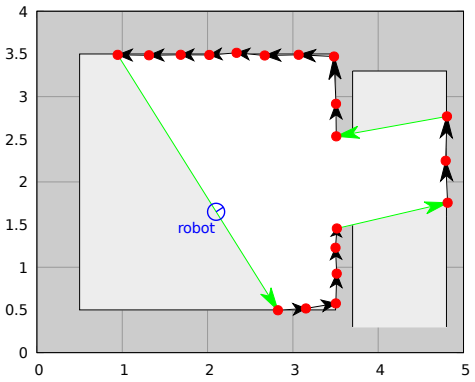
The simplification process can be split into three major steps:

1. Split the polygon's vectors into subsets called vector (sub-)chains.
2. Extract fitting lines from vector chains.
3. Replace every vector chain by a single vector based on fitting lines.

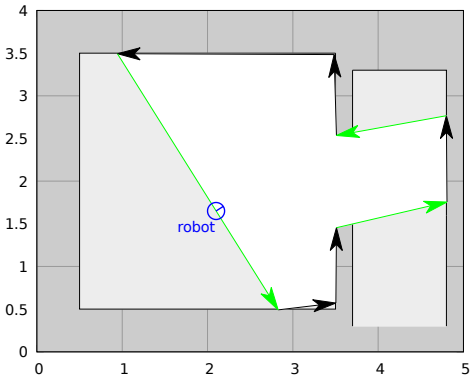
Step 1: Building the initial vector chains. A vector chain is a collection of vectors (at least one), where each vector (except the first) is connected to a single preceding and (except the last vector) a single following vector, i.e. the



(a) environment with the robot and its laser scan data



(b) the unrefined keyframe



(c) final keyframe after refinement

Figure 3.6: Creation of a keyframe. Image a) shows the robot in the test environment. The laser scan data is indicated with red dots. Image b) shows the aligned but unrefined keyframe, drawn on top of the environment. Image c) shows the final keyframe after the refinement step.

end point of one vector is the starting point of the next vector. All vectors in a vector chain share the same type.

To create vector chains from a polygon we create an empty vector chain and add a random vector from the polygon. We then add all vectors from the polygon that don't violate the definition of vector chains. If there are still vectors left in the polygon, we start a new vector chain with the vector that follows the last element in the previous vector chain and repeat the process until all vectors of the polygon are part of one and only one vector chain.

Step 2: Extracting fitting lines. The line fitting algorithm uses linear regression on vector chains to compute the line with the minimal square error in terms of sum over point-to-line square distance, with the start/end points of the vectors in the vector chain providing the points. We refer to the line that we compute this way as *original line*. Next we compute which vectors of the full-length vector chain are considered inliers in regard of the fitting line that we just created. For a vector to be considered as inlier, the distance of both its start and end point to the line must lie below a given threshold, otherwise we will treat the vector as an outlier. We then take the first vector of our vector chain and build a vector chain that only contains inliers and the first vector (even if the first vector would otherwise not an inlier). With this new vector chain we repeat the linear regression step, providing us with a new line that we call *refined line*.

We remove the last vector from the input vector chain and repeat the whole process of computing an original line and a refined line. This is done iteratively until our chain no longer contains any vectors. Each fitting result receives a score, computed according to the formula 3.9. The vector-chain/fitting line pair with the highest score is chosen. The vectors from the vector chain are removed from the input vector chain, which is then used as input for the next iteration, repeating the process until we have no vectors left.

Our scoring function is defined as:

$$inlier\ points: \quad P := \{p_i \in \mathbb{R}^2; p_i\ is\ inlier\} \tag{3.5}$$

$$inlier\ count: \quad n := |P| \tag{3.6}$$

$$average\ square\ error: \quad e := \frac{1}{n}\sum_{i=1}^{i\leq n} d_i^2 \tag{3.7}$$

$$count\ weight: \quad w := (\frac{n}{n+a})^b \tag{3.8}$$

$$score: \quad s := \frac{w}{e+c} \tag{3.9}$$

where d_i^2 is the square distance from point p_i to the refined line. Note, that the set of points P in 3.5 by the definition of sets does not contain any dupli-

cate points, which otherwise would typically be present, as the end point of one vector is also the starting point of the next vector in the chain.

There are three tuning parameters present: a, b, and c. The original scoring function had the tuning parameters set to $a = 3$, $b = 1$, and $c = 0$, which simplifies the formula. However we found during tests that the results could be improved when utilizing the current formula. The idea of the tuning parameters a and b are, to control how much we encourage a high inlier count even if it comes at an increase of the fitting error. Both are used to compute the count weight factor w, as defined in formula 3.8. All variants favor a higher inlier count, but some do so very quickly, and don't change much after the first ten or twenty inliers. Others don't change so drastically early on, and keep a flatter but more steady increase for a higher inliers count. Parameter b makes this trend even more drastic. This allows us to tune the algorithm to either favor a lower approximation error or a lower final vector count.

Parameter c has two tasks. First, it allows us to handle the case of zero error (which we always encounter when we have only a single inlier vector). And second it influences the weight that the average square error itself holds. The larger parameter c is set, the less influence the error has on the score, and consequently the number of inliers have a higher overall weight.

All the while our inlier threshold sets a hard boundary which vectors (and their respective points) can be considered inliers, so that even overly optimistic parameters for the scoring function don't lead to catastrophic simplifications of the environment. However as shown later in Chapter 5, a bad choice of parameters can lead to unsatisfying results non the less.

The impact of the scoring function and its parameters can be seen in Figure 3.7 and Figure 3.8. The figures show two test cases of noise-free toy examples that are refined with a variety of parameters. The top row in Figure 3.7 shows the unrefined input data of vectors, with the start/end points highlighted as blue dots. Right below in the following row we have the refined vector collection, utilizing our default parameter set for the scoring function. The remainder of the sub figures (including all sub figures in Figure 3.8) each use two of the default parameters and modify the third one. The inlier threshold has been set to a high value as to not interfere with the scoring itself for the sake of highlighting the scoring function.

The input data in the left columns resembles a corner case, that should be simplified to two perpendicular vectors. And indeed, all selected parameter sets lead to two vectors in the resulting refined vector chain.

The second input data, shown in the right columns, is a circle segment, representing a curved surface. Arguably the more interesting case, we now can see how the different parameters define a trade off between the number of resulting vectors and accuracy in approximating the original shape. In our

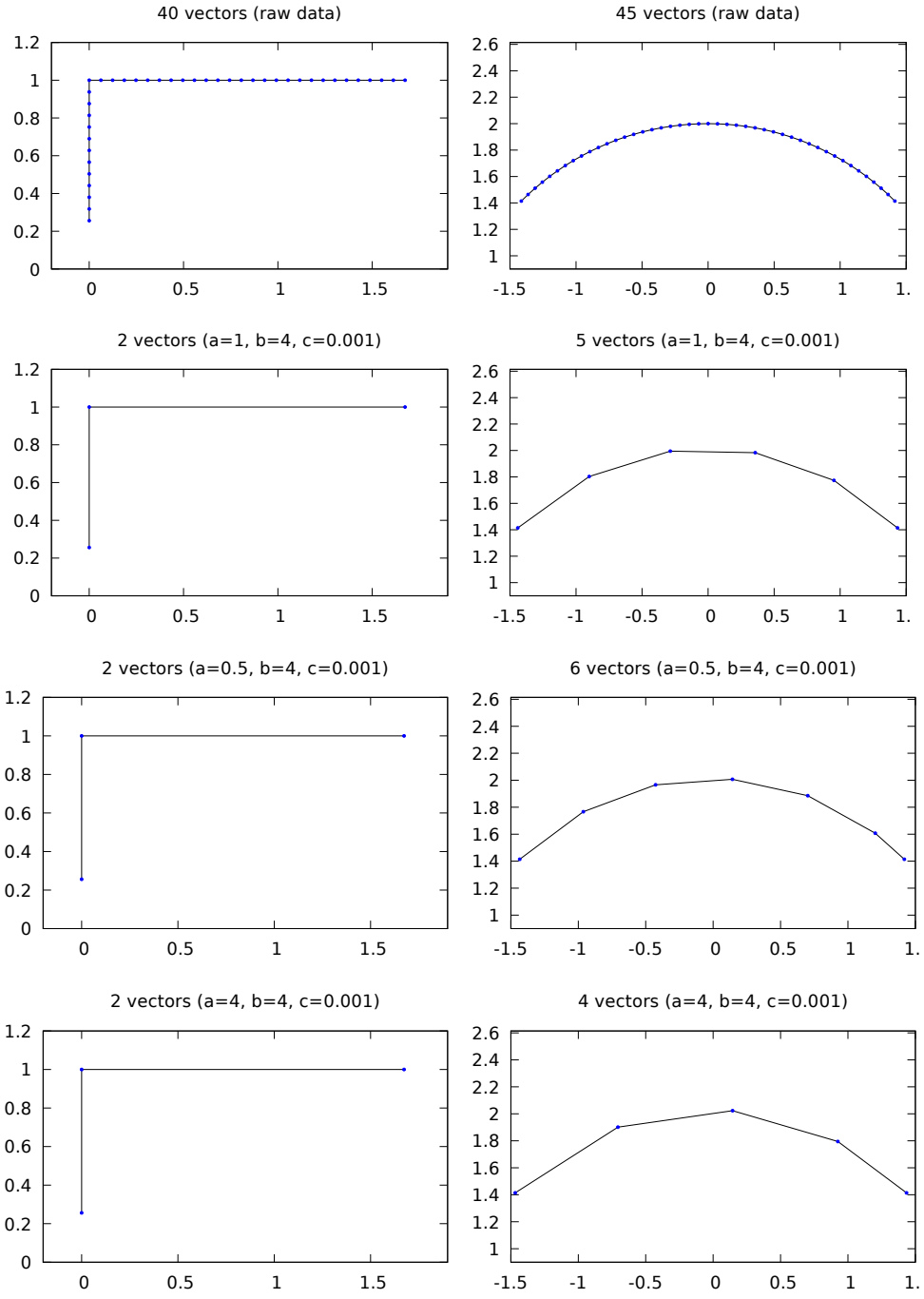


Figure 3.7: Corner & curve test case, part 1. The original data is shown in the top left row, consisting of 40 vectors (corner) and 45 vectors (curve). Each row shows the effect of a parameter set on the respective test case.

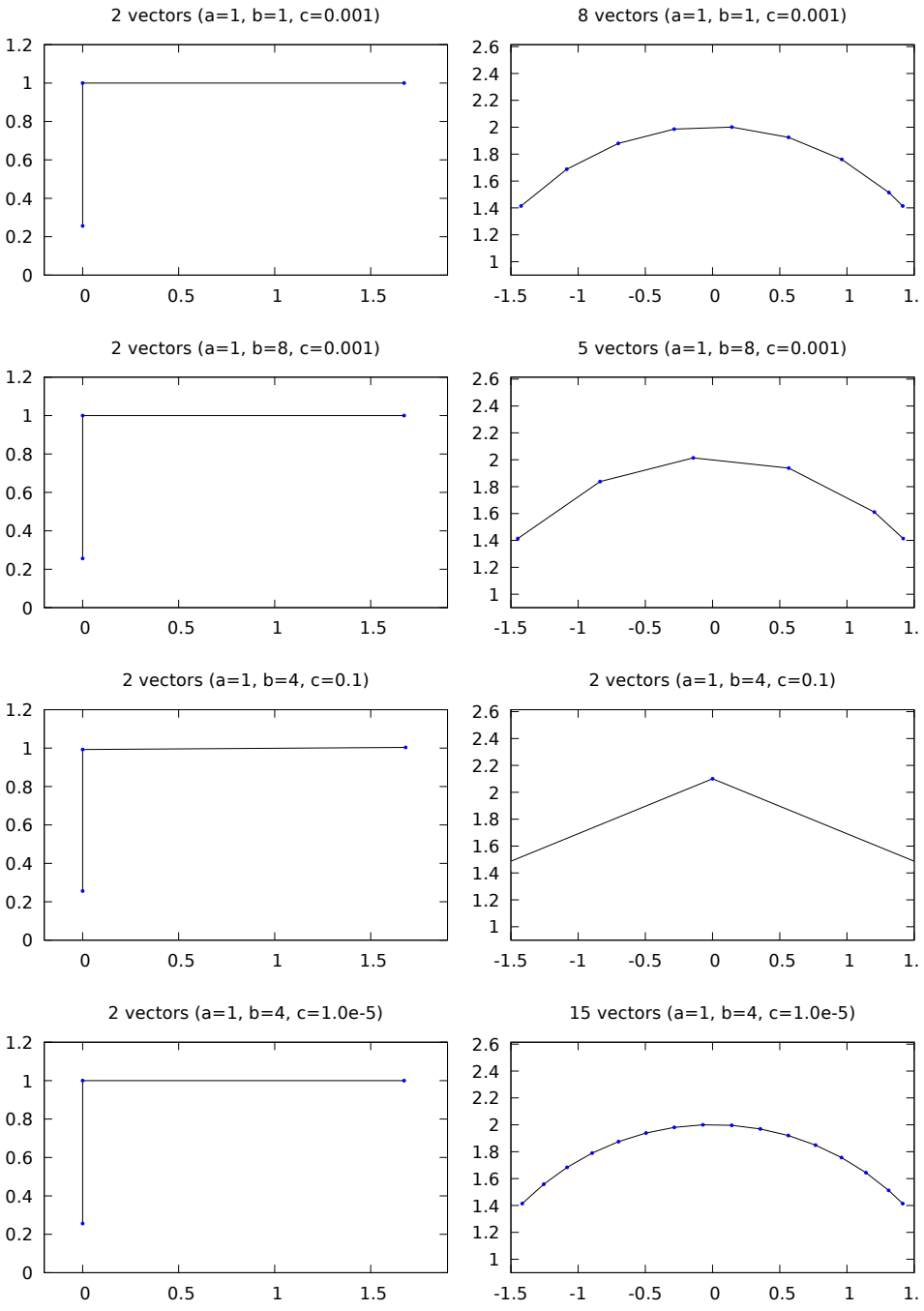


Figure 3.8: Corner & curve test case, part 2.

example, the original 45 vectors are reduced to 2 – 15 vectors.

Figures 3.9 shows the score for the first vector chain (with the very first vector being the right-most vector in both examples), in relation to the number of inliers. For most parameter sets of the example, the corner case data set has a distinct point where the score drops significantly. This point is reached when the inlier chain contains points that belong to the vertical line. In clear cases like this, a wide variety of parameters deliver good results and find the correct cut-off length for the inlier chain. In fact, only the second to last parameter set overshoots and includes a single point that does not lie on the horizontal line.

Figure 3.10 shows the same parameter sets, applied to the curve data set. The rise and fall of the score over the number of inliers is in general more smooth and now shows clearly how some parameters emphasize either a higher accuracy or larger inlier count.

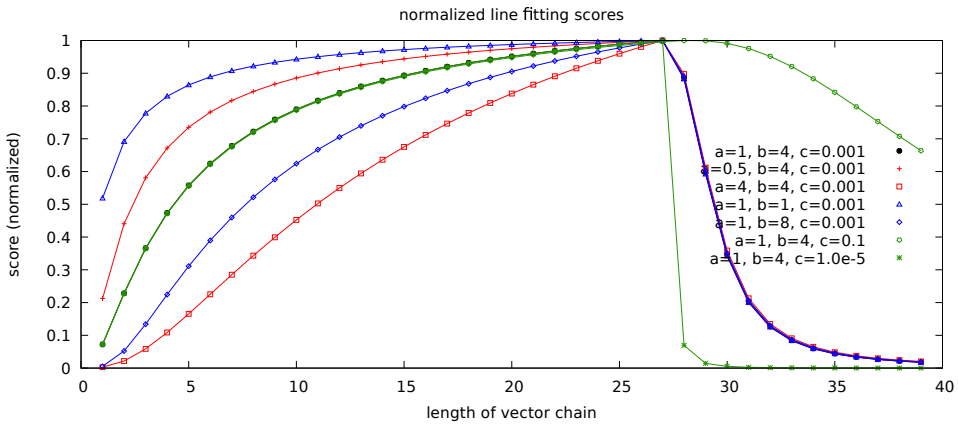


Figure 3.9: Line Fitter score in relation to the number of inliers for the corner example. The first parameter set (marked in black dots) holds our recommended default parameters

The process is visualized in Figure 3.11. The top image shows a set of vectors, starting with a frontier on the right, followed by ten obstacle-vectors and finally another frontier. The line-fitting algorithm starts with aggregating the vectors into vector chains (three chains In the example: a single vector frontier, one chain of ten obstacle-typed vectors and another single frontier vector chain). First, a line is computed from a linear regression over the start/end points of the chain (*original* line in the second image). Next the algorithm builds a sub-chain (which must include the first vector) of inliers and repeats the line fitting process once, creating the *refined* line. The process of creating *original/refined* lines is repeated, recursively using a half-length vector chain (the result visualized in the third image). The algorithm is applied on all vector chains of the original raw keyframe and ends with a collection of (sub-

Algorithm 3: Line Fitting for Vector Chains

Data: vector chain C; outlier threshold T
Result: line fitting result $R := \{score, line\}$
begin
  create empty collection of fitting results: $AllResults$
  while C *size* ≥ 1 **do**
    create point collection P_1 from start/end points of vectors in C
    compute linear regression line L_1 for points P_1
    create empty vector chain C_1, add first vector from C
    for $i \leftarrow 2$ *to* C *size* **do**
      $v := C$ at: i
      if v *distance to* $L_1 \geq$ *threshold* T **then**
        break
      end
      add v to C_1
    end
    create point collection P_2 from start/end points of vectors in C_1
    compute linear regression line L_2 for points P_2
    create empty vector chain C_2, add first vector from C_1
    for $i \leftarrow 2$ *to* C_1 *size* **do**
      $v := C_1$ at: i
      if v *distance to* $L_2 \geq$ *threshold* T **then**
        break
      end
      add v to C_2
    end
    create empty vector chain C_3, add first vector from C_2
    compute fitting score S for L_2 and C_2
    add $\{S, L_2\}$ to $AllResults$
    remove last vector from C
  end
  select highest scoring result R from $AllResults$
  return R
end

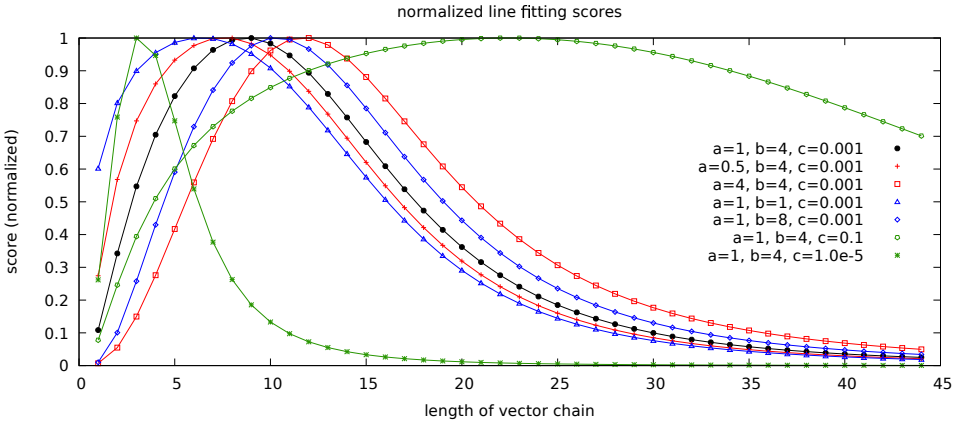
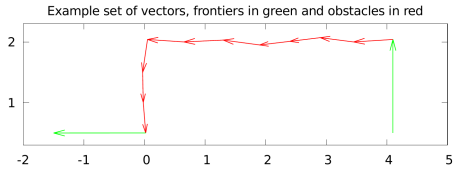


Figure 3.10: Line Fitter score in relation to the number of inliers for the curve example. The first parameter set (marked in black dots) holds our recommended default parameters

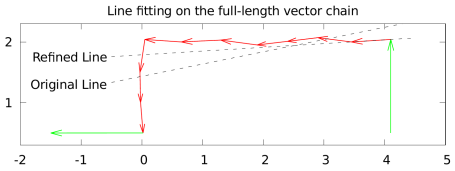
)vector-chains and their fitting lines (fourth image). The next step consists in defining the new vectors aggregating each vector-chain as shown in the last image. The pseudo-code is summarized in Algorithm 3.

Step 3: Building new vectors for a simplified polygon. Once we have sets of vector chains (the highest scoring vector sub-chains) linked with approximating lines, we need to create actual vectors to replace each vector chain with a single new vector. To compute the start/end points of a new vector, we determine the intersection points between the line for the current vector chain, and the lines for the previous/next vector chain. However, if the lines meet at a flat angle then the intersection point is not numerical stable. An example where the intersection point is placed significantly off from the expected locatiom is illustrated in Figure 3.12. This appears mostly with consecutive vector chains from different types which could theoretically be parallel or collinear.

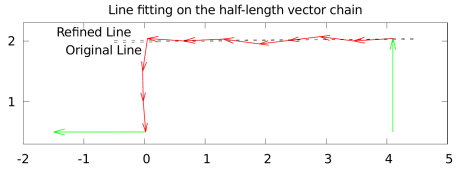
Therefore, whenever the lines meet at an angle of less then 15 degrees, we instead compute the projection points of the start point of the first vector and the end point of the last vector with the two lines involved. If the two points are closer than a given threshold (1cm in our implementation) then we use the average of the two points instead of the intersection point. If the distance is equal or greater than the threshold, then we use the projection points instead of the intersection point, and add an additional vector that connects the two projection points. When all vectors have been created, they form a new polygon that approximates the old polygon, but typically contains significantly less vectors. This process may remove small details in the map, but also averages over the noise from the laser sensor.



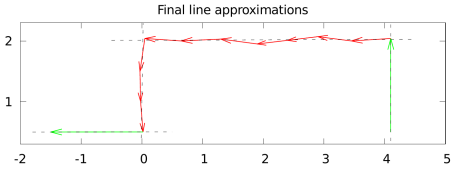
(a) Starting condition with three vector chains



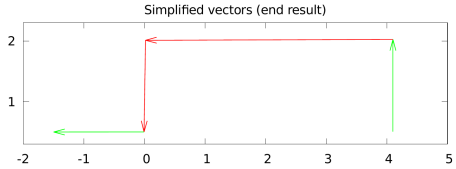
(b) Line fitting on full-length chain



(c) Line fitting on half-length chain



(d) Best-scoring lines from the line fitting process



(e) Resulting simplified vectors

Figure 3.11: Stages of the line-fitting process. The vector chain that is refined is colored red. The two neighboring chains are colored green.

3.6 Level of Detail & Parameter Tuning

During the polygon simplification process, we have the inlier threshold parameter that determines which points/vectors are considered an inlier when fitting a line with a vector chain. Vectors whose starting or end points have a distance greater than the inlier threshold are considered outliers. One way to chose this parameter is to look at the accuracy of the sensor and choose a threshold that will include for example 99% of the samples. For the laser scanner used in our simulations, we have a Gaussian error model with zero mean and a standard deviation of 0.01m. That means that we can expect 99.7% of all samples to lie within +/- 0.03m of the true position (approximately 2 outliers out of the 720 samples per laser sweep). If the environment does not include any challengingly shaped obstacles, this generally leads to good results and a low vector count.

However, if the environment contains curved obstacles or fine details need to be preserved, looking at the sensor specifications might not be enough. A looser threshold leads to a more aggressive aggregation of vectors and therefor a polygon with less vectors at the cost of a potential loss of

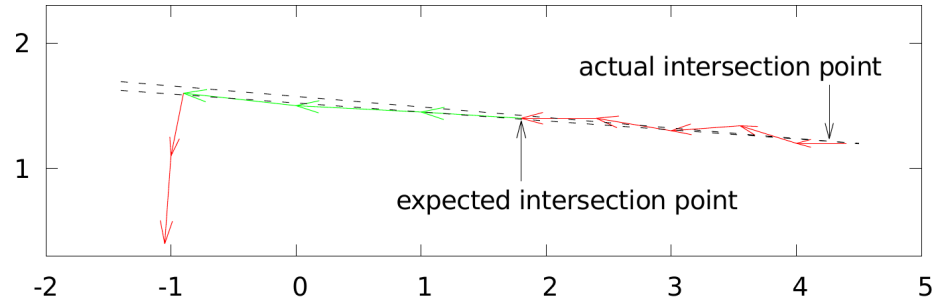


Figure 3.12: Example of two lines intersecting on a flat angle with an inappropriate intersection point.

details in the map and a worse approximation of curved obstacles. A tighter threshold on the other hand preserves more details but leads to a higher vector count in the simplified polygon. This is illustrated in Figure 3.13, where a starting polygon of 721 vectors has been simplified with different inlier thresholds.

3.7 PolyMap Merging

An important task in a SLAM algorithm is to integrate new data into the global map. In our case, this means to merge an aligned keyframe with the global map. In order to improve the speed of the merging process, we store the map in a Binary Space Partitioning Tree (BSP-Tree). A BSP-Tree is a data structure, where each node holds a hyperplane (i.e. a line in 2D) that partitions the space in half, while the leaves hold objects (e.g. polygons). The left child is associated with one side of the hyperplane, the right child with the other side. Objects (e.g. polygons) that are inserted into the tree are handed down, depending on which side of the hyperplane they are located. If an object intersects with a hyperplane then it is cut into pieces so that each piece lies on only one side of the hyperplane. In our implementation the leaves hold at most one polygon each.

When a new keyframe is merged with the map, the polygon that represents the keyframe is added to the BSP-tree. There it is split into convex polygons, which eventually end up in a leaf of the tree. If the leaf in not empty, we now only have to merge two convex polygons. If this results in a non-convex shape, the new polygon is split into convex shapes and the tree grows appropriately.

An empty BSP-Tree is represented by an empty leaf. If a polygon is inserted into an empty leaf, we first check if the polygon is convex. If so, the polygon is added to the leaf. If the polygon is concave, we need to create a line to split the polygon into smaller pieces. The hyperplane/line must be chosen

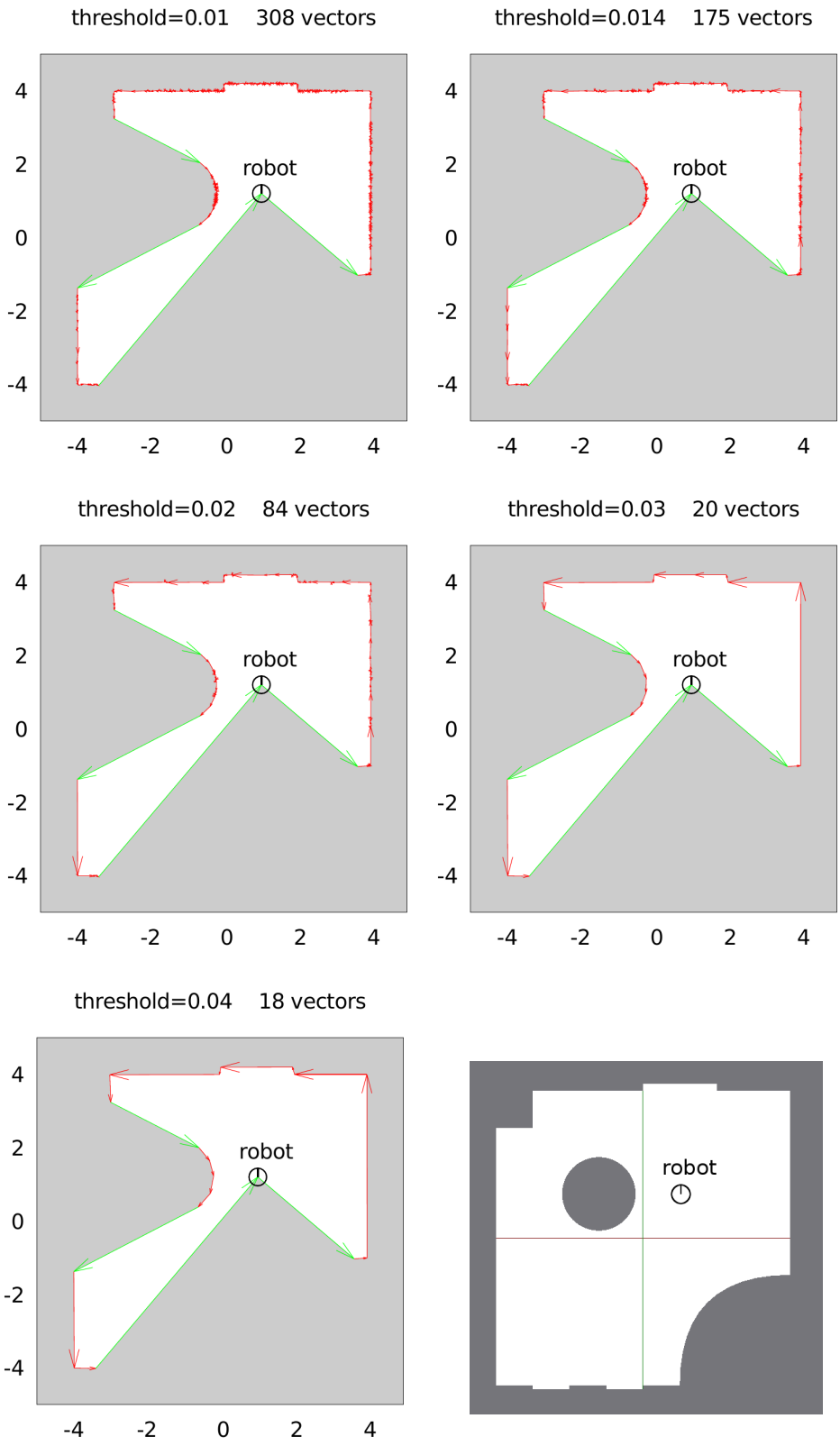


Figure 3.13: The same keyframe with different inlier threshold for the simplification process. The last image shows the ground truth map.

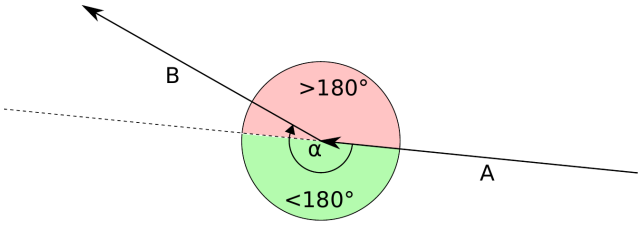


Figure 3.14: The angle between vector A and vector B. Angles over $180°$ cause a polygon to be concave. Note, that intersecting *vectors* have angles from $0°$ to $360°$, while *lines* only intersect at angles in the range of $0°$ to $180°$.

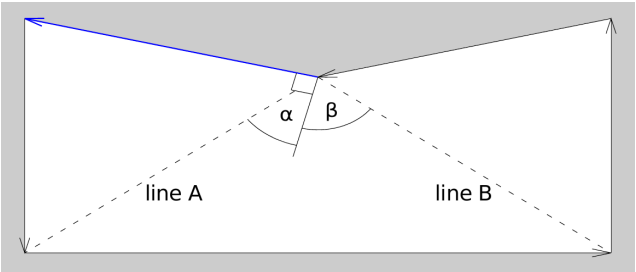


Figure 3.15: Two possible splitting-lines for the polygon. Line A has a sharper angle against the normal of the blue vector, and hence is chosen as the line to split the polygon.

in a way that, by repeating the strategy, we will eventually end up with convex polygons. A trivial way to do this is to split the polygon into triangles, since they are convex by construction. However this increases the number of polygons unnecessarily. We instead first select all vectors whose angle α with the previous vector is greater then $180°$ (see Figure 3.14). These vectors are causing the polygon to be non-convex, since in convex polygons all angles are less than or equal to $180°$. From the selected vectors we take the starting point and create a temporary line for each other point in the polygon (except the two direct neighbors), by connecting them with the starting point. This is displayed in Figure 3.15, where the vector in blue has an angle greater than $180°$ with its predecessor. We then check which of the temporary lines has most points one the "wrong" side and select this one as our new hyperplane/line. This will create two or more polygons that contain less line segments with an angle α greater than 180 degree, guaranteeing us that we eventually end up with only convex polygons after finite repetition. The new line is used to create a node that replaces the leaf. Two new leaves are created for the node, one for each side of the line. The polygons are then handed down to the new leaves, where the process repeats until all polygons are convex and stored in a leaf.

If a polygon is inserted into a non-empty leaf, we need to create a hyperplane/line. However, the algorithm to find a line that we used for empty

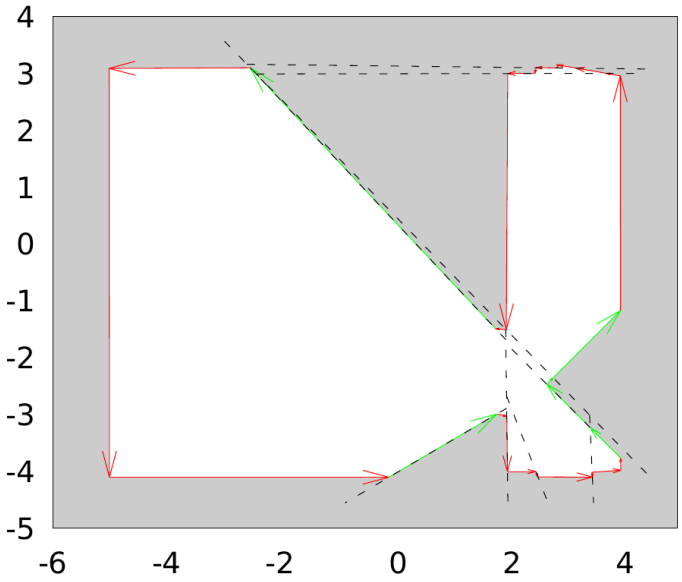


Figure 3.16: Map stored in a BSP-tree. The hyperplanes are displayed as black dashed lines.

leaves can fail in this scenario, since both polygons could already be convex. Therefor we instead start by treating every vector of the original polygon (the one already stored in the leaf) as a line and check how many corners of the other polygon lie on the "wrong" side of the line (the side where the original polygon is not located). We chose the line with the most points on the "wrong" side. If there are no lines that have any points on the "wrong" side, than the polygon is fully enclosed by the original polygon. If the polygon contains no obstacle-typed vectors, then we discard the polygon immediately. Otherwise we repeat the process with the role of the two polygons reversed. If this still bears no result, we can ignore the polygon. Otherwise we chose the line with the most points on the "wrong" side. We replace the leaf with a new node, split the polygons on the new line, and hand down the pieces to the node's children.

The merging process can be seen in Figure 3.17, where the top row shows the keyframes and the bottom row the corresponding global map. The map at t=1 contains more vectors than keyframe #1, because it is already stored in the BSP-tree and therefor partitioned into convex polygons (sector borders are not shown in the figure). Figure 3.16 shows the map at t=1 with the hyperplanes of the BSP-tree visible.

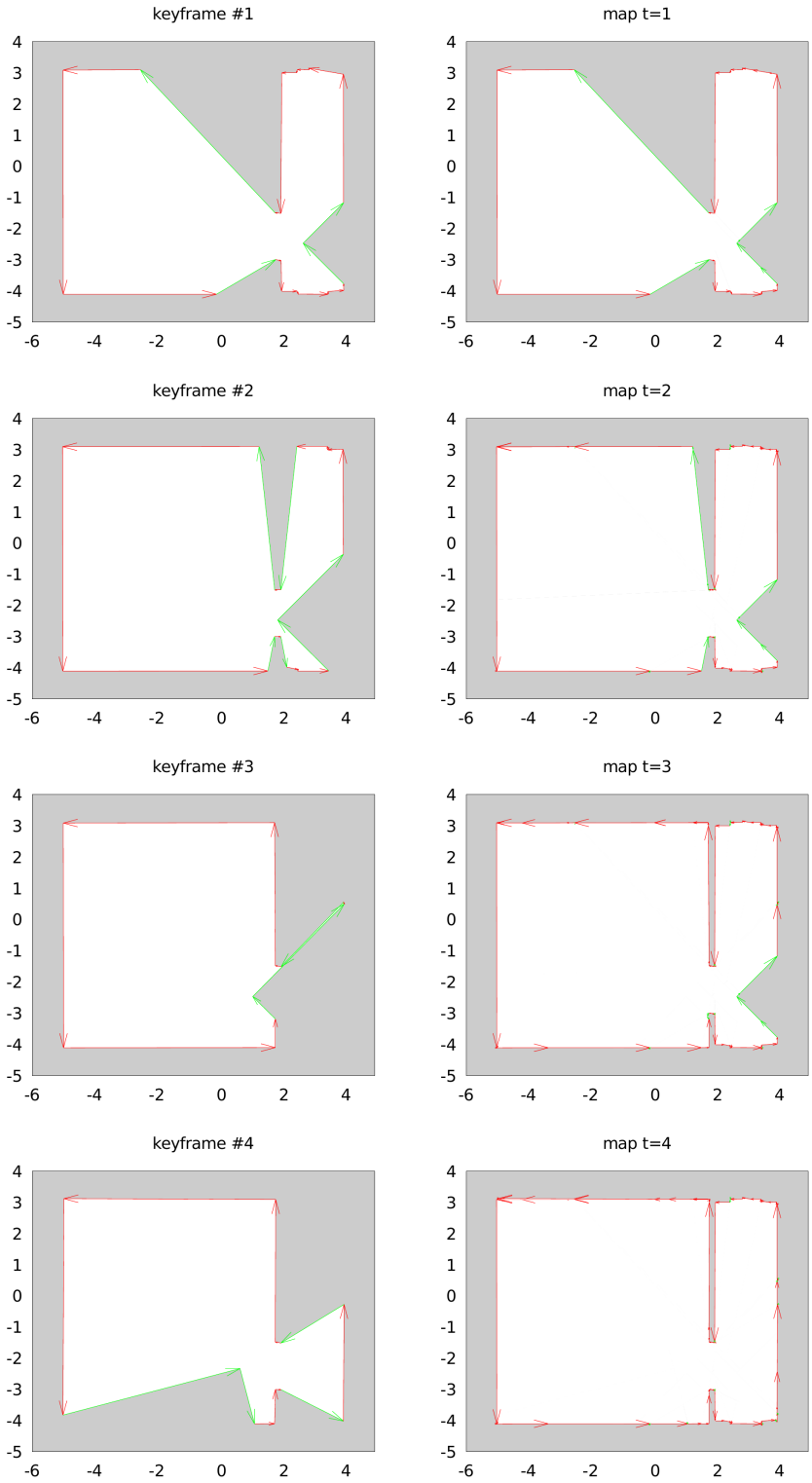


Figure 3.17: Map merging of aligned keyframes. The left column shows three keyframes, the right column shows the map as it grows with each keyframe incorporated.

3.8 Summary

Our PolySLAM algorithm creates PolyMaps from laser scans. This chapter presented the challenges of creating accurate PolyMaps. In particular, we modified the ICP algorithm to support point-to-vector matching, and made use of backface culling to reduce the chance to get stuck in local minima. Furthermore, our polygon refinement technique reduces the impact of measurement noise by averaging over multiple points. The refinement process also reduces the number of vectors per keyframe, reducing the overall size of the global map. With the scoring function and its parameters, we are able to control the trade off between the detail that we want to preserve and how much sensor noise we have to deal with. These techniques allow us to create consistent maps despite the lack of global optimization.

In following chapter will present how to realize navigation on the PolyMap format. For this we will build a topological graph which allows for efficient path planning within the explored space.

PolyMap-Based Navigation

Chapter 4

Contents

Introduction

Having a map of the environment, even if incomplete, enables us to plan further movement of the robot. This can be done fully autonomously, i.e. without any human interaction, or with the assistance of a human operator who marks a target area on the map. In both cases we are provided with the task to find a viable path from the robots current position to a target area. This task is commonly referred to as *Navigation* in robotics.

Navigation in this context typically focuses on two scales: *local* and *global*. Global navigation is based on the global map of the environment, and relies on a topological graph to compute a path from the robot's current position to a target area. Local navigation on the other hand primarily relies on the live data from the robot's sensors to compute a viable path in the presence of obstacles that are not necessarily part of the map (e.g. dynamic obstacles such as pedestrians). Our focus in this section lies on global navigation, in particular the computation of a topological graph from a PolyMap. In such a graph, each polygon of the map is represented as a single vertex of the graph, while edges model neighboring polygons that are reachable via an overlapping sector-typed vector from both polygons.

4.1 Numerical Problems to Consider

When faced with the task to build a topological graph, we start of with a PolyMap, which is essentially a set of polygons. From there, we need to de-

termine which polygons share overlapping vectors of type *sector* as these are the only (explored) places where the robot can traverse through. While this looks like an easy task at first glance, there are a few details that make this more tricky than expected. The first problem is, that we cannot just look for an identical vector with start and end points reversed, since vectors may overlap without sharing any common start or end points. This is illustrated in Figure 4.1 in which the opposing horizontal vectors either share the start or end point, but never both. Note, that in this example the vertical sector borders do share both start and end points.

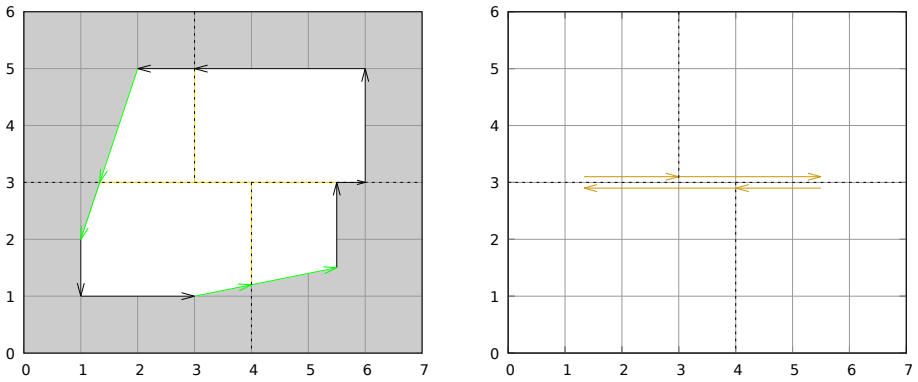


Figure 4.1: An example of sector borders partial overlapping with vectors on the other side of the hyperplane. The left figure shows the map embedded into a BSP tree with three hyperplanes splitting up the map. The right figure highlights the overlapping vectors in question.

The second problem is, that computing whether two given lines (extending two vector that we want to test) are collinear is numerically challenging. For example, the nature of how computers store float point numbers makes it highly likely, that two lines/vectors that should be considered collinear have a slightly different orientation. And for similar reasons, it is possible that for example the start or end point of a vector does not exactly lie on a line that is collinear to the vector. Adding a tolerance to similarity tests on the other hand provides us with the problem, that we may encounter false positives when testing for overlapping vectors. We circumvent this problem by using the BSP-tree structure that created the sector-typed vectors in the first place, as explained in the next section.

4.2 Formal Definition of the Topological Graph

We formally define a topological graph as follows:

$$node: \quad N = \{(p, n_1, ..., n_k) | p \in P; n_i \in N\} \tag{4.1}$$

$$top.\ graph: \quad G = \{n_i | 1 \geq i \geq S; v_i \in N\} \tag{4.2}$$

A node N represents a polygon P (as defined in Definition 3.3) in the graph, as well as a list of all neighboring nodes that are connected to this node. That means, that we don't have to explicitly define edges between nodes, since this information is already embedded in the node itself. The topological graph itself is defined a set of nodes.

This notation is carried over in our Pharo implementation, where we use a *Dictionary* to store the nodes. The dictionary utilizes the polygon as a key when storing the node. This allows us fast access to the node when we have the corresponding polygon.

4.3 Building a Topological Graph from a BSP-Tree

When building a PolyMap live, a BSP-Tree is used to store the polygons of the map, as explained in 3.7. By construction, every sector-typed vector is adjacent to a hyperplane (i.e. a line) in the BSP-Tree. Therefore, instead of considering all vectors of the map, we can narrow down potentially overlapping vectors to those that lie on the same hyperplane. The structure of the BSP-Tree makes this convenient and relatively fast, assuming that the tree itself is balanced. Once we have collected all vectors that lie on the same hyperplane, we compute which ones overlap by a simple one-dimensional search, comparing the position of the start/end points of the vectors expressed by their position on the hyperplane by the signed distance to a reference point on the hyperplane. Now we can add edges to the graph for each polygon that is sharing an overlapping vector with a polygon on the other side of the hyperplane. This is done by adding the corresponding neighbors to the node that represents the polygon in the graph (and creating new nodes if the respective polygons are not already represented in the graph). This process creates a topological graph based on the PolyMap, which is considerably more sparse than a graph based on a conventional grid map.

Algorithm 4 shows the whole process with pseudo code. The first outer loop is collecting all vector-polygon pairs, where a sector-typed vector of the polygon is touching a hyperplane. These pairs are stored in a dictionary, where each hyperplane is acting as a key, referring to a collection of vector-polygon pairs. The second outer loop iterates over all entries (i.e. collections) in the dictionary. Each collection holds all polygons that share the

same hyperplane, together with the vector that is collinear with that hyperplane. Finding overlapping vectors within each collection provides us with all (paired) polygons that are connected by a sector border and hence connected in the topological graph as well.

4.4 Path Planning on a Topological Graph

Once a topological graph has been built it can be used to compute the path to a target location in the environment. The first step is to determine the current location of the robot in the topological graph. For this we select the polygon that contains the center of the robot. This can be done efficiently since we have the map stored in a BSP-Tree which allows us to just walk down the tree on whichever side of the hyperplanes the robot's center is located. The polygon can then be used to identify the corresponding node in the topological graph. This will act as our starting node for a conventional $A^\star$ or $D^\star light$ graph search. For the final result of our graph search, we add a temporary node that connects the robot's current location with the first node from our graph search result. Similar, another temporary node is added at the end to connect the last node from the search result to the actual goal.

A simple example is illustrated in Figure 4.2, which shows navigation on a small map. The robot is located at the bottom part of the map, labeled 'start'. The target area is in the top right corner of the map, labeled 'goal'. With the map's BSP-Tree, it is partitioned into smaller polygons, indicated by the yellow lines. Each leaf in the BSP-Tree is linked to a node in the graph via the polygon that both share. In this example the topological graph (colored pink) contains eleven nodes. With an $A^\star$ graph search we are able to find a valid path from the robot's position to the target area, as shown in the right figure.

4.5 Using Grid Partitioning on the PolyMap

By construction, a PolyMap tends to have a lot of long and narrow polygons that are a result of incremental map updates as the robot moves in the environment. However for navigation purposes we would prefer polygons that are not to large in size, since we want the center of the polygon to be representative for the area that is covered by the polygon. Our solution is to impose a grid structure on the map. All polygons that span over a grid line are split on the respective line, and the new polygons are treated separately. This breaks up all large polygons into smaller polygons and provides a more even distribution of nodes in the topological graph as a result. Compared to conventional grid maps, our cell dimension is relatively large: 1 by 1 meter by default. Each occupied cell contains a BSP-Tree, like the original PolyMap

Algorithm 4: Building a topological graph from a BSP Tree

```
Data: BSP-Tree bspTree
Result: top. graph graph
begin
    Dictionary dict := {hyperplane →
     (collection of (vector, polygon))}
    foreach leaf in bspTree do
        bsp − node := leaf
        polygon := leaf polygon
        while bsp − node has parent do
            bsp − node := bsp − node parent
            line := bsp − node hyperplane
            foreach vector v in polygon do
                if v is collinear with line then
                    add line → {v; polygon} to dict
                end
            end
        end
    end
    foreach (hyperplane → (collection of (vector, polygon))) in dict do
        foreach entry {v; p} in (collection of (vector, polygon)) do
            foreach entry {v2; p2} in (collection of (vector, polygon))
             do
                if v direction = −1 ∗ v2 direction then
                    if v overlaps with v2 then
                        find or create node N for polygon p
                        find or create node N2 for polygon p2
                        add neighbor N2 to node N
                        add neighbor N to node N2
                    end
                end
            end
        end
    end
end
```

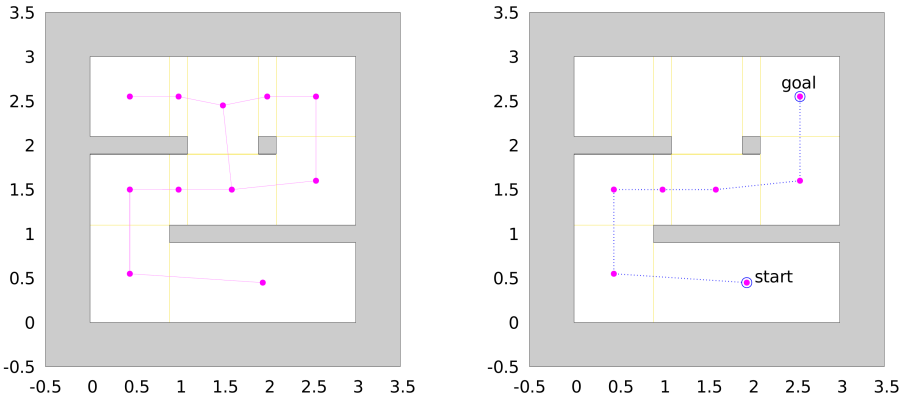


Figure 4.2: Navigation example on a small map. The left figure shows the map portioned into smaller areas, with the topological graph overlayed. The right figure shows the computed path that leads from the start location to the target area.

format, but now limited in its dimension to the size of the grid cell. This setup allows us to quickly access polygons, first by determining which grid cell we need, and then by searching within the BSP tree of the cell. Lookup of the cell typically is done in $O(1)$ time complexity (like a 2D table lookup), and finding elements in the BSP tree (if balanced) takes $log(N)$ time, with N being the number of elements in the tree. It also removes any worries about unbalanced BSP-Trees, since the individual trees are now much smaller, so that even a degenerated tree will not pose a problem performance-wise. In Figure 4.3 we can see the effect of the grid overlay in a section of the *Zigzag* map, which acts as a long and wide corridor.

With a map M as defined in 3.4, we formally define the GridPolyMap as follows:

$$cell: \quad C = \{(r, s) | r \in M; s \in \mathbb{R}^2_{>} 0\} \tag{4.3}$$

$$GridPolyMap: \quad G = \{m_{i,j} | i, j \in \mathbb{N}; m_{i,j} \in C\} \tag{4.4}$$

The GridPolyMap G is a two dimensional grid where each entry contains a cell C. A cell in return contains a map M, stored in a BSP-Tree, and its two dimensional size s. The cells are typically quadratic, but any rectangular size can be used. Similar, the cells are not required to all share the same size (but have to share the same size on the borders that they share). For convenience sake however, we assume that all cells share the same size in both dimensions. The GridPolyMap enables very fast spacial lookup since addressing a cell in a grid can be done in constant time, and finding an element in a BSP-Tree in linear time in respect to its depth, which in logarithmic to the number of entries (if balanced).

An example of a grid overlay with a topological graph can be seen in Figure 4.4. There we have a cell size of 1m by 1m. Close to the walls we have the typical structure of many small polygons that result from merging keyframes that contain some noise. The cells that don't contain any obstacles or frontiers on the other hand contain fewer but larger polygons. By separating parts of the maps into individual cells we prevent the larger polygons to be split into smaller ones when the merging process introduces new hyperplanes, because the hyperplanes in one cell do not affect any other cells.

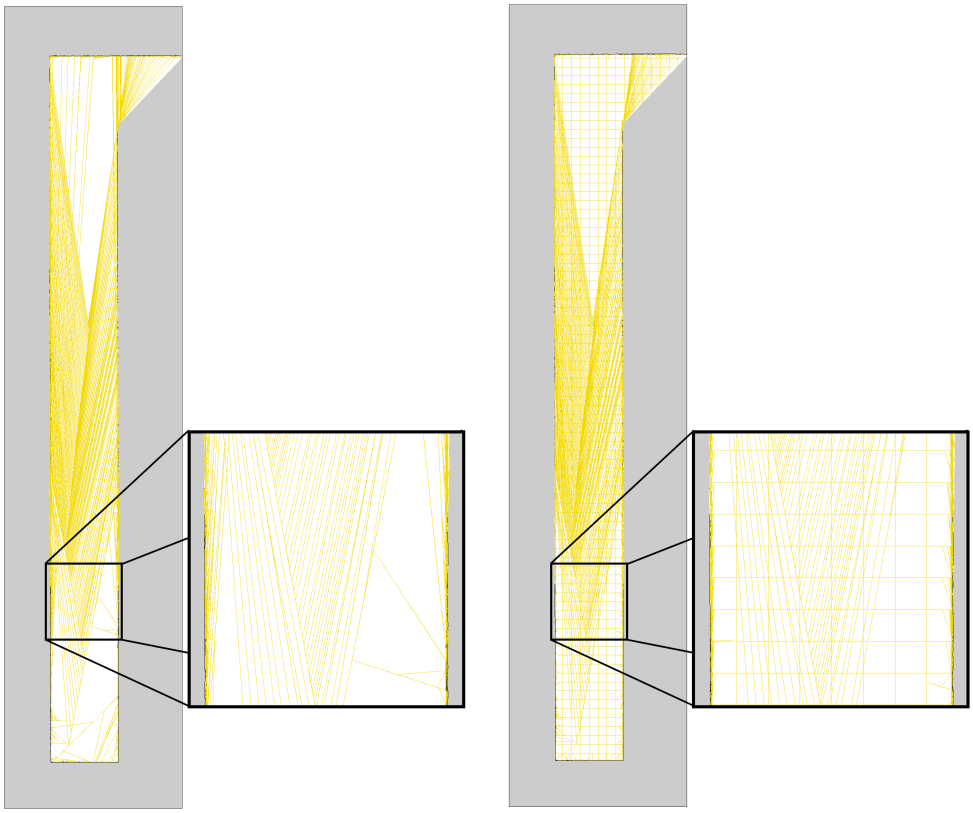


Figure 4.3: A direct comparison between a PolyMap and a GridPolyMap. Both maps have been created from the same keyframes. The grid overlay on the right map reduces the maximum size of the polygons, and restricts hyperplanes from the obstacles to their respective grid cells.

Simplifying Cell Content. If a grid cell contains only polygons that are made up entirely of section-type vectors, then by construction the cell is completely filled by the polygons and contains neither frontiers nor obstacles. Hence in that case we replace all polygons by a simple square polygon with the dimensions of the cell. In our experiments this reduced the total number of polygons by about 10%. Figure 4.5 shows an example of this on small section of the *Zigzag* map. Note, that grid cells that contain frontiers or obstacles re-

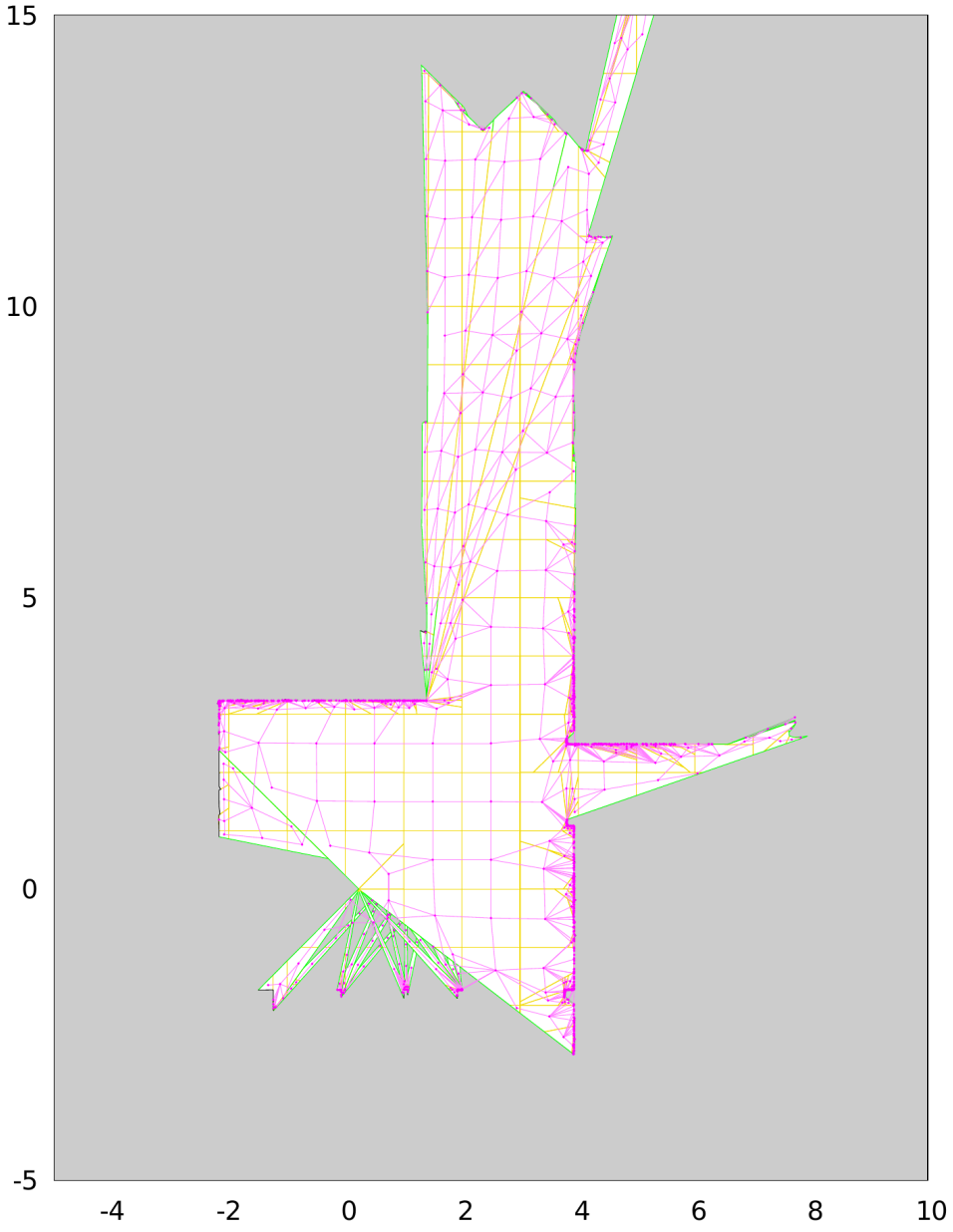


Figure 4.4: A map section and its topological graph. The grid structure leads to an even distribution of nodes in the large open areas of the map. The graph in this example contains 1385 nodes.

main unchanged with this algorithm.

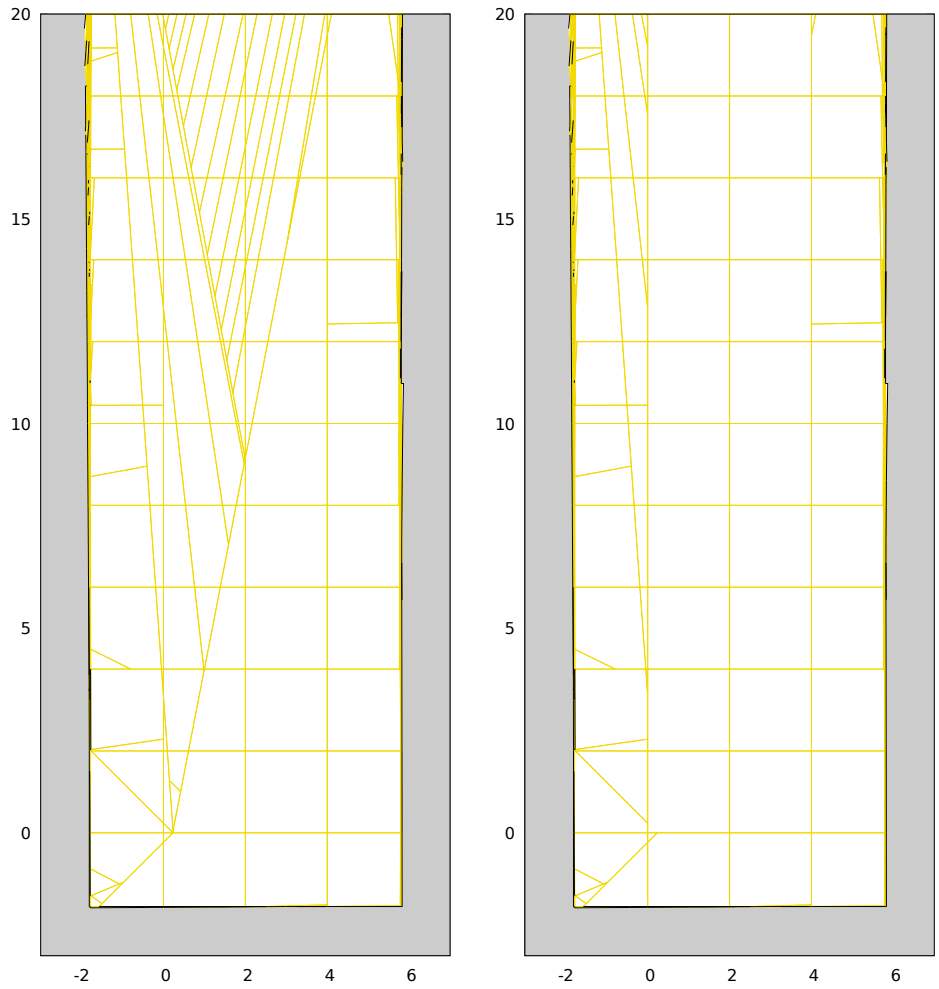


Figure 4.5: The *Zigzag* map with a grid overlay. The left figure shows the map before the simplification process, the right one afterwards. Cells that contain obstacles are not modified by the algorithm, and hence still contain multiple polygons that contain only sector-type vectors.

4.6 Removing Inaccessible Nodes from the Graph

When the size of the robot is known, we can take the robot footprint into account to remove nodes from the graph that are not accessible by the robot. As a quick heuristic we remove all nodes from the graph where the polygons center point is closer to an obstacle than the radius of the robot. Since we typically experience a lot of small polygons around obstacles, this reduces the size of the graph by a significant amount. In fact, the number of nodes and edges is reduced by about two orders of magnitude in our tests. An example can be seen in Figure 4.6, where the number of nodes in the graph drops from

Map Format		full graph	pruned
PolyMap		60786	676
GridPolyMap (grid size in meter)	0.5	78972	1837
	1.0	63279	885
	2.0	62427	688
	4.0	62289	670

Table 4.1: Size of the topological graph (node count) for various map configurations.

62427 to 688. The robot radius used in this example was 0.25m.

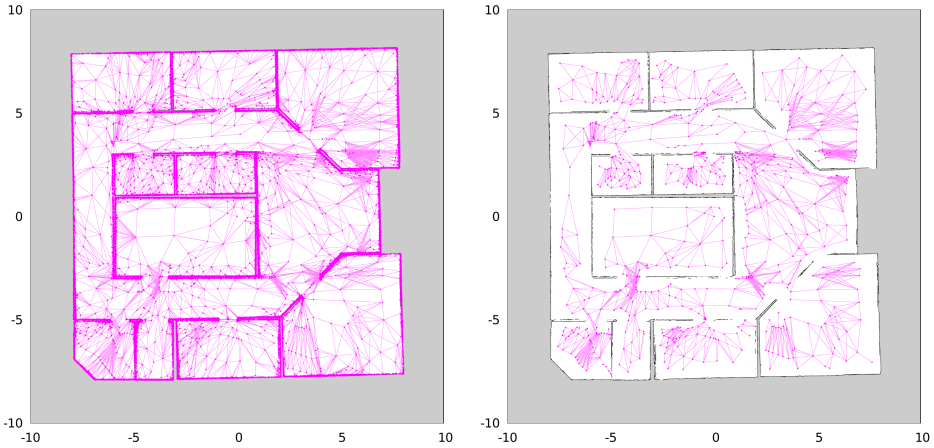


Figure 4.6: The left figure shows the full topological graph of the map, drawn on top of the actual map. The right figure shows the pruned graph with all inaccessible nodes removed.

To test the impact of our optimizations, we created a topological graph for a few of maps. The map has been created with our PolySLAM algorithm, using data collected from a simulated environment (the *office* environment). Table 4.1 shows the results with and without pruning the topological graph. We can see a significant reduction of the graph size in all cases. We also notice, that the grid overlay increases the number of nodes by only a moderate amount.

4.7 Comparison with Occupancy Grid based Navigation

It is natural to compare the performance of our navigation approach with a similar implementation based on occupancy grid maps. To accomplish this, we are creating a topological graph by treating every free cell in the occu-

pancy grid like a square polygon and build a topological map based on this. When creating the graph for the occupancy grid, we consider the 8 closest neighbors for each cell, allowing for horizontal, vertical, and diagonal edges.

In our example we created a topological graph from an occupancy grid representing an office environment (16m by 16m in size) and compared the performance. The occupancy grid has been inflated by 0.25m to make the results comparable to our pruned graph of the GridPolyMap, which also removed nodes that are closer than 0.25m to any obstacle. Table 4.2 and Figure 4.7 are showing the results of our experiment. With 72475 nodes, the graph computed from the occupancy grid is relatively large with respect of the environment. In comparison, the graph for the GridPolyMap only contains 688 nodes – two orders of magnitude less. Similarly, the A^* search requires significantly less iterations to reach the target on our GridPolyMap-derived topological graph. However, while the path for the occupancy grid contains more nodes, the actual distance is shorter. The reason for this is that the path for the GridPolyMap is not as close to the obstacles, and contains more zigzag-like movements.

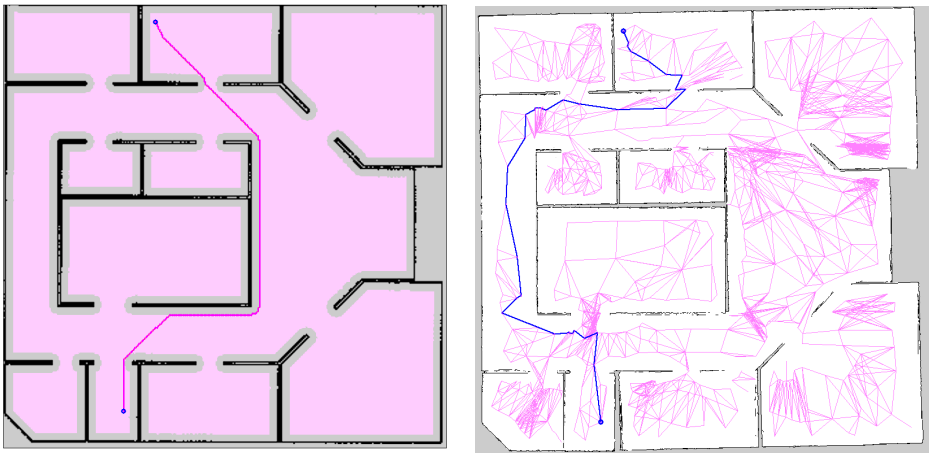


Figure 4.7: Navigation on an Occupancy Grid in comparison to a GridPolyMap. The left image shows the resulting path on the occupancy grid, the right image on a GridPolyMap. The pink background in the left image visualizes the traversable area after inflating all obstacles. The pink lines in the right image represent the edges of the topological graph.

4.8 Summary

This chapter presented one of our key contributions: Navigation on PolyMaps. We showed how to construct a topological graph from a PolyMap, and how to prune it to reduce the total number of nodes in the graph. We also introduced a grid overlay to avoid overly long polygons, and an optimization

Graph source	node count	path node count	path length
Occupancy Grid	72475	344	19.45m
GridPolyMap	688	39	25.06m

Table 4.2: Comparison of the topological graph created from an occupancy grid and created from a GridPolyMap. The computed path in visualized in Figure 4.7.

to reduce the number of polygons in the map. In direct comparison to topological graphs from occupancy grids, our topological graph has significantly less nodes.

In the next chapter we will evaluate our algorithms in a series of experiments. There we will test both the SLAM algorithm and our Navigation stack on data obtains from simulations, from public data sets, and from robots in our own labs.

Experiments

Chapter 5

Contents

Introduction

We performed experiments to evaluate the performance of our PolyMap format and PolySLAM algorithm. We used multiple sources of data sets for the experiments to cover a wide array of environments and setups. One source is a collection of public data sets that is available at [ODS]. These data sets have been widely used to evaluate state-of-the-art SLAM algorithms.

Another source of data for our experience comes from the Gazebo simulator. We used a set of benchmark maps used in previous work by Le et al. [LFBL18]. In addition a few more maps were used to test specific cases, such as the maps from Section 5.6, since the original benchmark maps had no curved surfaces. Using the simulator has the advantage that we have access to ground truth (both the robots trajectory and the environment's geometry). However a simulator never perfectly represents reality, so results have to always been seen in the corresponding context and taken with a grain of salt. Still, simulations are a relatively quick and cheap way and they allow us to run the experiments in a very controlled manner repeatedly and reproducibly.

The final data source was created by using our own robot, shown in Figure 5.1. We tele-operated the robot in our building, recording all sensor readings for later processing. The robot, a Turtlebot2[1], was equipped with a *Hokuyo's UTM-30LX* laser range finder. The sensor has an opening arc of $270°$, with a resolution of $0.15°$ angular resolution, a range from 100mm to and a range of 3000mm with a resolution of $\pm 30mm$ and performs 40 scan

[1] https://www.turtlebot.com/turtlebot2/

sweeps per second. The trajectory was chosen so that there is an overlap in the explored area towards the end of traveling, allowing us to roughly estimate the pose error by looking at the resulting map. The environment had its own challenges such as corridors that exceed the maximum range of the laser range finder, and a curved wall section. All data sources used can be found at our website [Pds] or at [ODS].

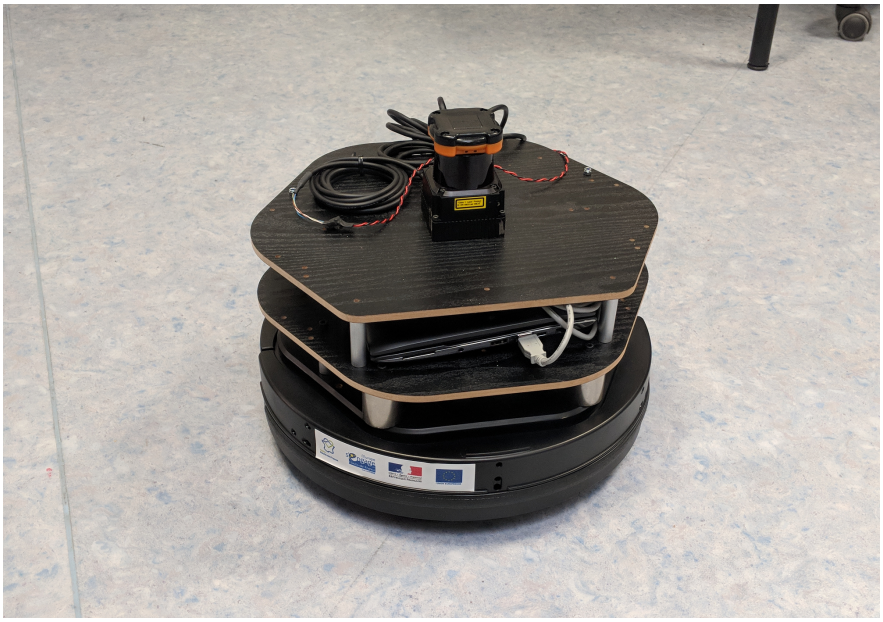


Figure 5.1: The Turtlebot2 used in our experiments. The laser range finder is mounted in the center of the top plate, a small laptop computer is located between the plates.

5.1 Metrics

The results from our experiments were evaluated using different criteria and metrics. The first step is visual confirmation that the map is consistent and depicts the environment correctly. Since this is difficult to put into numbers, we also used the map quality metrics *normalized error* [SPR13] (NE) and *Structure Similarity* [WBSS04] (SSIM) index. NE is computed from the sum of the distance between occupied cell of the ground-truth to the nearest one in the built map. Lower NE indicates better results, identical maps reaching a value of zero. SSIM evaluates local similarity by measuring cell intensities in a floating window, higher values indicating a better match with 1.0 being the highest possible score.

For the experiments from the simulator (where ground truth is available) we also computed the length of the trajectory and the maximum translation/angular errors of the pose estimates. Lastly, for the polygon simplifi-

cation process, we counted the number of vectors in test keyframes and once again performed visual inspection of the results to evaluate the quality of the results.

5.2 PolyMap Memory Sizes

Our first experiment was designed to test the memory footprint performance of the PolyMap format itself without relying on PolySLAM. For this, we took occupancy grids and converted them into PolyMaps by applying Algorithm 1 from Chapter 3 which creates a PolyMap in three steps:

1. create a vector (frontier or obstacle) for every transition from free space to non-free space,
2. connect neighboring vectors, creating polygons
3. simplify the polygons by merging connected collinear vectors of the the same type

It is noteworthy, that the conversion algorithm does not try to optimize the output with diagonal vectors, and only creates vectors that are parallel to the x- and y-axis. Still, the converted map contains all the details of the original map, and it is possible to recreate the original map from the PolyMap. As a post-processing step we also removed *isolated* areas, i.e. polygons that consist of only frontiers.

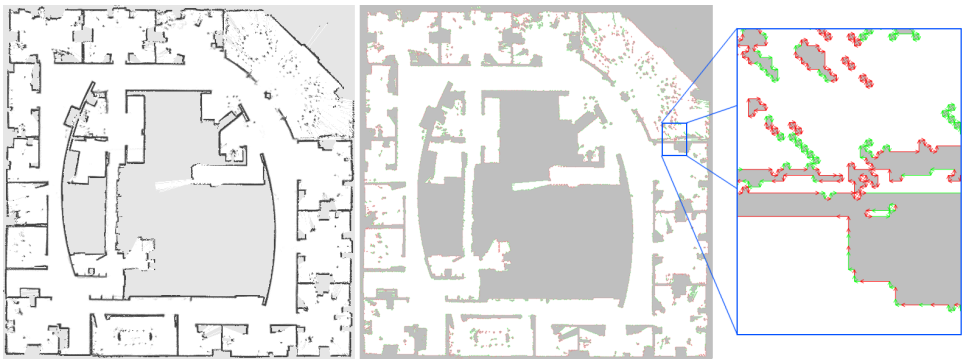


Figure 5.2: Occupancy grid (left), full converted PolyMap (middle) and zoom in the PolyMap (right)

Figure 5.2 shows the result on applying our algorithm to convert a grid map of the Intel Research Lab data set[2] to a PolyMap. Obstacles are shown with red vectors and frontiers with green vectors. The resulting PolyMap in this example consists of 829 polygons, with a total of 18054 vectors. The right-most figure is a zoomed-in section of the PolyMap, displaying individual vectors. Interestingly, the size of the full PolyMap is approximately the

[2]http://ais.informatik.uni-freiburg.de/slamevaluation/datasets.php

size of the compressed grid map. In this particular example, the grid map (in PNG format) takes about 100kB, and the uncompressed PolyMap requires about 160kB. Compressed, the PolyMap shrinks even more to approximately 44kB (ZIP[3]) and 20kB (7z[4]). Nevertheless, this algorithm only creates either horizontal or vertical vectors. Further optimization that allows vectors of any direction could result in even smaller memory footprints of the maps. Such optimizations will be considered in future work, though our PolySLAM implementation is not affected by this limit in the first place.

5.3 Simulation Setup

We report on a lot of different simulations in the following sections to assess our contributions. This section presents our common setup to all of our experiments unless mentioned otherwise. Simulations where conducted by using the Gazebo simulator [Gaz]. The simulated robot was equipped with a laser range finder (30m range, 270° opening angle, 720 scan points per sweep) with zero-mean Gaussian noise modeled after our Hokuyo laser range finder. Furthermore the robot odometry is published with noise (zero-mean Gaussian noise with default parameters) and as ground truth without noise. The odometry reading with noise was used by the SLAM algorithm, the ground truth reading were used to compute the pose errors.

The maps used are *Loop*, *Cross*, *Zigzag*, *Maze*, and *Willow garage* (W. G) as found in Le et al. [LFBL18]. The maps are about 80×80 meters (except for Willow Garage which is 55×45). PolySLAM was used offline on the collected data sets.

5.3.1 Loop Environment

The *Loop* environment is designed to test the loop closure capability after a long trajectory with a relatively feature-less surrounding. With the overall lack of features and the length of the hallways exceeding the maximum range of the laser range finders by about a factor of two, relying on odometry readings is mandatory here. At the same time, odometry alone is not enough for an accurate construction of the map, as drift accumulates over time. Start position for the experiments was the bottom left corner of the map.

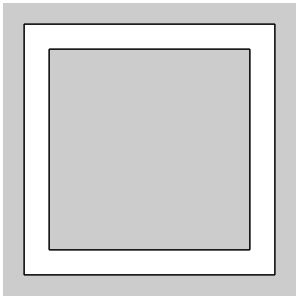


Figure 5.3: The *Loop* map (80x80m)

[3]ISO/IEC 21320-1:2015, https://www.iso.org/standard/60101.html
[4]https://www.7-zip.org/

5.3.2 Cross Environment

The *Cross* environment naturally creates several loops that need to be traversed if the whole map has to be explored. Algorithms that support loop closures have ample opportunities to reduce the global error as the robot creates one loop after another. Despite being of the same size as the Loop map, the traversable area is larger by about 50%. The hallways are still longer than the maximum laser range, but less dramatically so than in the *Loop* environment. In our experiments we had the robot start in the center of the map, at the 4-way crossway.

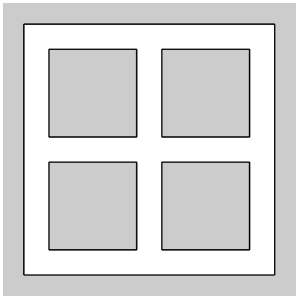


Figure 5.4: The *Cross* map (80x80m)

5.3.3 Zigzag Environment

The *Zigzag* environment is challenging for its long corridors that makes it difficult to estimate the distance traveled. Furthermore, the hallways are relatively close together, so that it is possible to accidentally align point clouds with the wrong side of the wall. The length of the hallways exceeds the laser maximum range, similar to the *Loop* environment. This is the only map in our simulation setup that does not promote loop closures. The robot starts in the bottom left corner and ends in the top right corner.

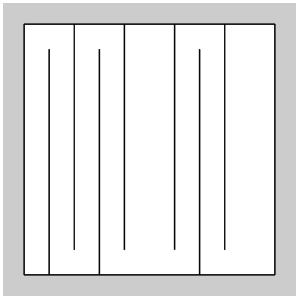


Figure 5.5: The *Zigzag* map (80x80m)

5.3.4 Maze Environment

The *Maze* environment is cluttered with walls and requires a relatively long trajectory to cover the whole area. Similar to the Zigzag map, the relatively thin walls can cause misalignments between keyframes. While there are still sections that exceed the laser's range, we almost always have some structure nearby to compensate for odometry drift. The start position of the robot is close to the center of the map, slightly to the right.

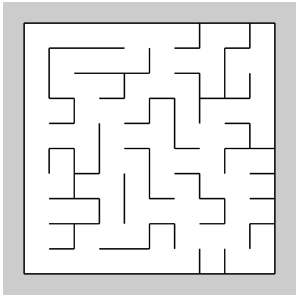


Figure 5.6: The *Maze* map (80x80m)

5.3.5 Willow Garage Environment

The *Willow Garage* environment is the smallest and yet most complex of our test environments. It depicts a classic indoor environment of an office with multiple small room of similar size and shape. This is also the only map that has an open environment with no outer map borders. However we limited our exploration to indoor only and did not leave the building. The robot's start position is in the bottom left, close to the entrance of the building.

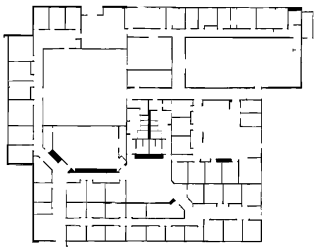


Figure 5.7: The *Willow Garage* map (55x45m)

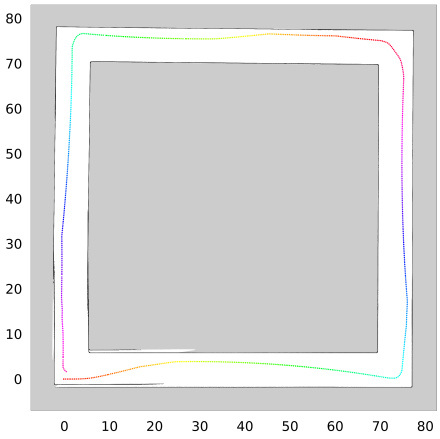
5.4 Simulation Results

A first visual inspection shows that all maps created with PolySLAM (see Figure 5.8) are consistent and usable for navigation and path planning. On closer look we can find small misalignments in every map, but not on a level that would impair our ability to use the map with autonomous or tele-operated robots. Table 5.1 shows the total distance traveled by the robot for each map, as well as the maximum errors in translation and rotation. The last column shows the maximum error as percentage of the total distance traveled by the robot. We notice, that the highest errors (relative to the distance traveled) are found on the Loop and Cross maps. The most likely reason for this is the fact that both maps have corridors that exceed the maximum sensor range, leaving only the left and right walls for localization corrections.

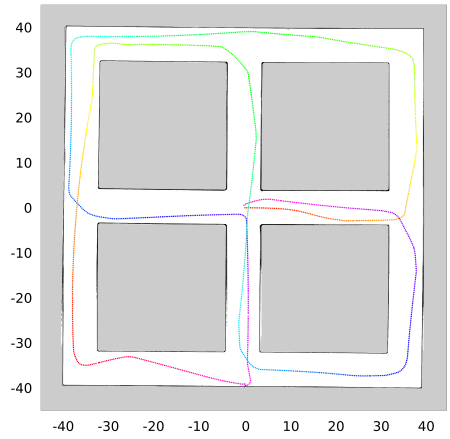
Table 5.2 shows an overview of *normalized error* (NE) and *Structure Similarity* (SSIM) index. For this, the constructed PolyMaps were converted to grid maps (with a 5x5cm grid size) and aligned to the ground truth by hand to compute the metrics. Figure 5.8 shows the resulting PolyMaps, including the estimated trajectory of the robot.

With PolySLAM currently not having any active loop closure techniques employed, we rely on our accurate pose estimate to reduce drift to a minimum. As such some maps are more challenging for PolySLAM than others.

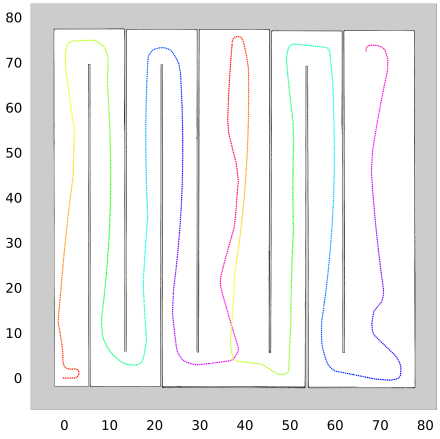
The PolyMap of the *Loop* environment, as show in Figure 5.8a, we have a visible misalignment around the starting/end point of the trajectory (bottom left), but still a relatively good overlap. However we notice that there is very little angular drift present, a testament PolySLAM's capabilities. Navigation is certainly possible with this result, both autonomous and with teleoperation. The NE value is worse for PolySLAM than the results from gmap-



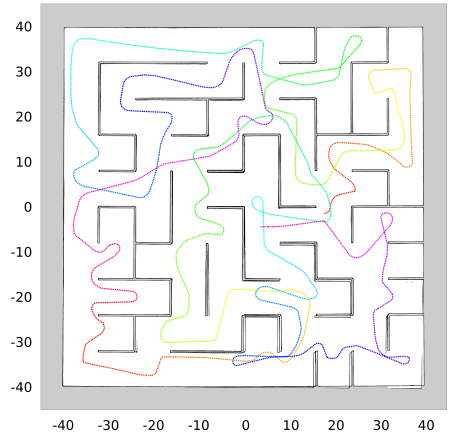
(a) The *Loop* map.



(b) The *Cross* map.



(c) The *Zigzag* map.



(d) The *Maze* map.

(e) The *Willow Garage* map.

Figure 5.8: PolyMaps of the five environments/maps that were used in the Gazebo simulations.

Map	size (meter)	distance traveled	max. error (transl. / rot.)	max trans. error by total distance
Loop	80x80	294m	1.25m / 0.031°	0.43%
Cross	80x80	576m	0.91m / 0.038°	0.16%
Zigzag	80x80	729m	0.54m / 0.019°	0.074%
Maze	80x80	946m	0.59m / 0.049°	0.062%
Willow Garage	55x45	951m	0.18m / 0.022°	0.019%

Table 5.1: Simulation results: distance traveled and maximum pose error for each map.

Map	NE PolyMap	NE gmapping	NE karto	SSIM PolyMap	SSIM gmapping	SSIM karto
Loop	131.9	73.7	20.4	0.92	0.92	0.94
Cross	72.6	33.8	41.6	0.89	0.90	0.90
Zigzag	20	87.2	18.3	0.88	0.89	0.89
Maze	5.9	6.9	8.2	0.89	0.90	0.90
W. G.	11.7	4.2	5.3	0.77	0.88	0.82

Table 5.2: Simulation results: normalized error (NE, smaller is better) and structure similarity (SSIM, larger is better).

ping and Karto, but the SSIM metric places all three very close together.

The *Cross* map (Figure 5.8b) shows significant less misalignments than the *Loop* map. We attribute this to the shorter hallway segments, which in return result in less drift accumulation before we can revisit already explored areas. Overall, PolySLAM shows again good results and a consistent map with only small alignment errors, despite the long trajectory of 576m.

The *Zigzag* PolyMap map, shown in Figure 5.8c holds no surprises. PolySLAM created a consistent map with a visible but overall not significant drift in the trajectory. We also once again notice that the map contains very little angular drift.

The *Maze* environment is more complex than the previous three maps. Among all maps tested in our simulations, PolySLAM experienced the highest angular error ($0.049°$) on this map. While there are some misalignments in the lower right part of the map, the result is still a consistent map that is suitable for navigation, as seen in Figure 5.8d.

The last of the benchmark maps is the *Willow Garage* environment. Despite having the longest trajectory in our simulations, we achieved very good results with a maximum translation error of 0.18m over a distance of 951m, all without any global optimization. The maximum angular error is also quite

low with $0.022°$, the second lowest result among the five PolyMaps. The full map and trajectory can be seen in Figure 5.8e.

5.5 Backface Culling

One of our key elements in our SLAM algorithm is backface culling, a technique explained in Section 3.3. The impact of backface culling largely depends on the parameters for ICP (e.g. the outlier threshold) and the characteristics of the environment. In particular walls that are explored from both sides (e.g. the wall between two offices) are easy to confuse without backface culling. In environments like *Zigzag* and *Maze* with multiple thin wall, SLAM algorithms are prone to mismatching, while this is not the case with the *Loop* or *Cross* maps.

We run PolySLAM with backface culling active/inactive for a variety of ICP outlier start thresholds. Table 5.3 shows the maximum pose errors for the maps Zigzag and Maze. In all cases the results with backface culling active are better. In some cases the differences are significant.

	max. error for ICP outlier threshold:		
Map: Zigzag	0.2	0.6	1.0
w/ backface culling	0.50m / 0.034°	1.25m / 0.033°	1.58m / 0.034°
w/o backface culling	0.55m / 0.033°	1.40m / 0.034°	1.58m / 0.035°
	max. error for ICP outlier threshold:		
Map: Maze	0.2	0.6	1.0
w/ backface culling	0.69m / 0.049°	1.40m / 0.055°	1.63m / 0.062°
w/o backface culling	0.70m / 0.049°	1.45m / 0.056°	1.84m / 0.060°

Table 5.3: Impact of backface culling on pose error, tested with three different ICP outlier thresholds.

The general pattern is, that backface culling has a bigger impact when our ICP outlier start threshold is larger. This is no surprise, since a larger threshold means, that larger distances between points from the point cloud and vectors in the map are still acceptable for matching. However, the outlier threshold cannot simply be set to a small value without consequences. In particular, if the odometry provided by the robot is not very reliable, we depend on a relatively large outlier threshold during the initial iterations of ICP to align the point cloud correctly. Hence, the ability to deliver good matching even with large outlier thresholds is certainly a positive characteristic that increases the robustness of our algorithm.

We also tested the impact of backface culling on our self-recorded real-world data set. Not having ground truth available, we have to resort to visual

inspection to evaluate the results. In this experiment we created PolyMaps with a set of different ICP outlier thresholds, ranging from 0.2 to 1.0. The results with backface culling active are either equal or better than the maps creates without backface culling. Figure 5.9 shows a zoomed-in part of the PolyMap, created with the exact same settings, only with backface culling active in the left image and disabled for the right. The difference in map quality is most prominent around the connecting corridor, where the misalignments cause the corridor to stick partially into the wall.

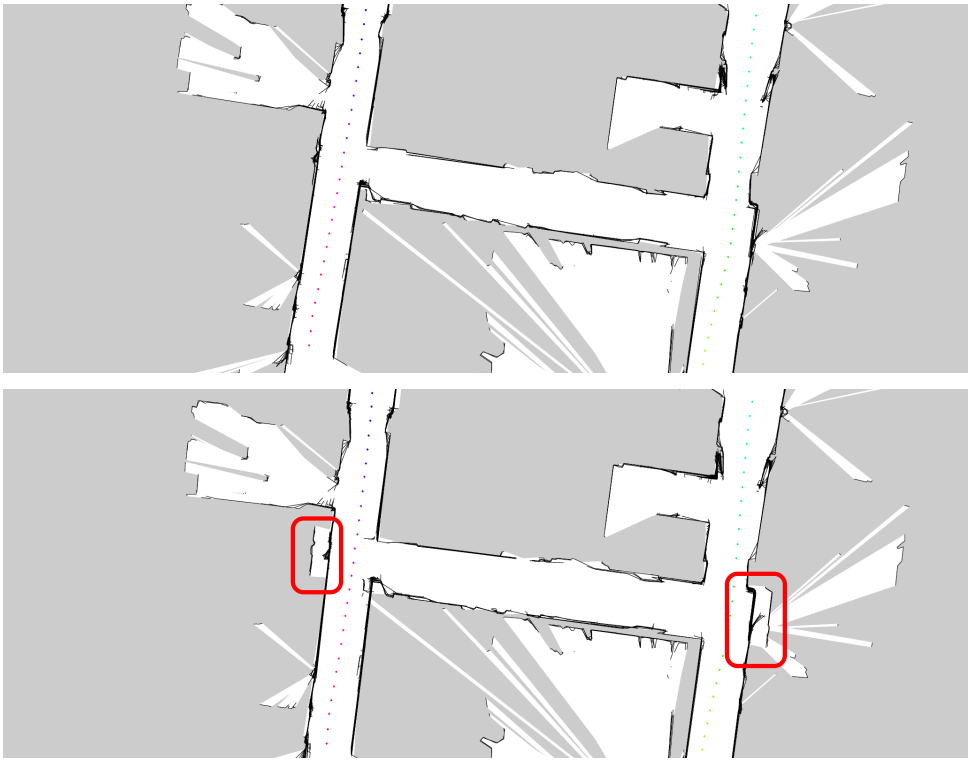


Figure 5.9: A comparison between running PolySLAM with (top) and without backface culling (bottom) on a data set of our office floor. The figures are zoomed in to the connecting hallway, where we can see the biggest difference between the two maps. Two red rectangles highlight spots where the errors are easy to see.

5.6 Polygon Simplifier Parameter Tuning

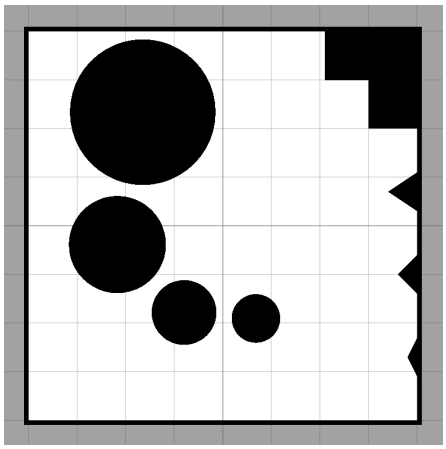
As explained in Chapter 3, the polygon simplification algorithm has a few parameter that can be adjusted to enhance the result quality. In this regard, there are two aspects that we want to optimize for. First, we want to approximate the environment as precisely as we can, and second we want to do so with as few polygons as possible. All the while we have to deal with sensor

noise which makes this task more difficult.

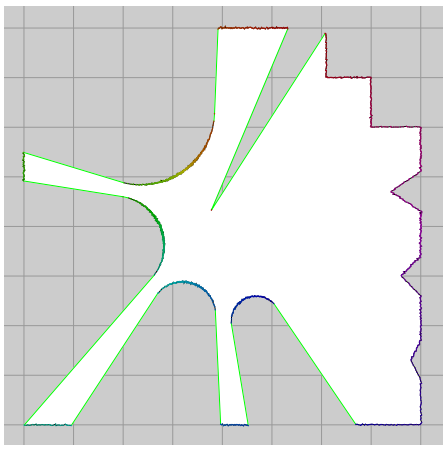
In total we have four parameters that influence the polygon simplification algorithm. The first parameter is the inlier threshold, which specifies which vectors are considered inliers when an approximating line is provided. This parameter must be chosen in accordance to the sensor noise that we have to expect.

The other three parameters (named a, b, and c) are used for computing a line fitting score. They influence on whether we prefer longer vectors at the potential loss of detail, or shorter vectors that may overfit the sensor data. All three parameters do so in a different way, which we look into in following experiments.

As test setup we have created a small 8x8 meters test environment for the simulator that contains a mix of flat and curved surfaces, shown in Figure 5.10a. The robot was equipped with a laser range finder that provided a 350° opening angle (to provide a wide field of view while leaving a blind spot behind the robot) and angular resolution of 0.25°, giving us a total of 1401 measured points per keyframe. The sensor noise model is the same zero-mean Gaussian noise as in all simulations in this chapter. We did chose a keyframe that has the robot slightly off-center, and includes all curved surfaces as well as plenty of flat surfaces. Figure 5.10b shows the keyframe as a polygon before the simplification process. Since the keyframe has a very high resolution in terms of measurement points, we additionally repeat the tests with reduced point density.



(a) The test map used for the experiments. The grid shown has a 1m spacing, the map itself is 8x8 meters in size.



(b) The original keyframe before simplification. The polygon contains 1702 vectors. Each measured point is drawn in a different color.

Figure 5.10: PolyMaps of the five environments/maps that were used in the Gazebo simulations.

5.6.1 Inlier Threshold Parameter

The inlier threshold is a parameter that allows us to reject points/vectors if they are located too far away from the line approximation. This is a common technique for outlier rejection used in a wide variety of fields. In this experiment we look at how the parameter influences the vector count and the approximation quality of the polygon simplification process. We found, that values larger then 0.4m don't improve the result and (depending on the other parameters) can worsen the result. In our experiments, we fixed the parameters $a = 1.0$, $b = 4$, and $c = 0.0001$, because they provided good results in average. We experimentally determined those values on different maps, both in simulations and with real robots.

Figures 5.11 and 5.12 show zoomed-in sections of the map (Figure 5.10) for four different values of the threshold in the range of 0.01m to 0.04m. The scan points that form the edges of the original polygon are overlayed to give a better impression how well the vectors approximate the original scan data.

Visual inspection of the resulting polygons shows that a tight value for the threshold causes trouble with the sensor noise. While the vector count is still noticeable reduced, the quality of the approximation leaves room for improvement. This can be observed in Figure 5.11 where we have a lot of zigzag patterns caused by the approximations sticking too close to noisy measurements.

A more moderate threshold delivers presentable results, but may occasionally fail to deal with the noise, causing small "bumps" in the approximations. A large threshold simply means that we don't reject vectors easily and that the algorithm relies largely on the scoring parameters. The problem with that is, that the algorithm relies on outlier rejection for the refinement steps, and without this, we test and score less line approximations. A large threshold also increases the risk to ignore finer details that would otherwise be preserved in the process.

In Table 5.4 we can see the sizes (i.e. vector count) of the resulting polygons with respect to the inlier threshold. The original polygon in this experiment had 1402 vectors. Again, the results also depend on the scoring parameters, which remain unchanged during this experiment. Based on both visual inspection and the vector count, we find that an inlier threshold in the range of 0.03m to 0.04m delivers good result in combination with the sensor (and its noise model) from our data set.

5.6.2 Line-Fitter Scoring Parameters

The scoring of the line fitting algorithm relies on three parameters (referred to as a, b, and c), as described in Chapter 3. We conducted a series of exper-

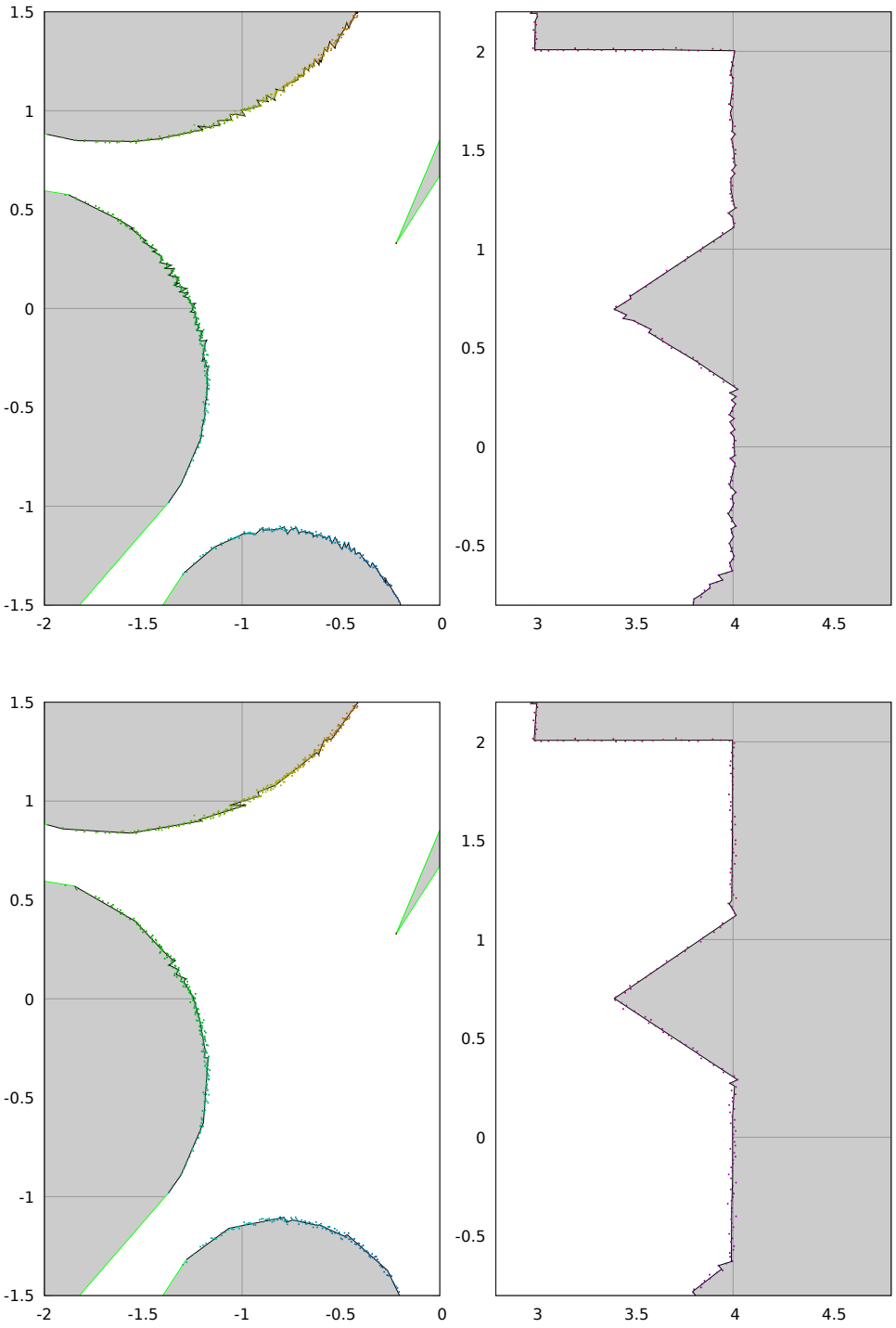


Figure 5.11: Polygon simplification with an inlier threshold of 0.01m (top) and 0.02 (bottom). Despite a noticeable reduction in vectors, we still have a significantly jagged border.

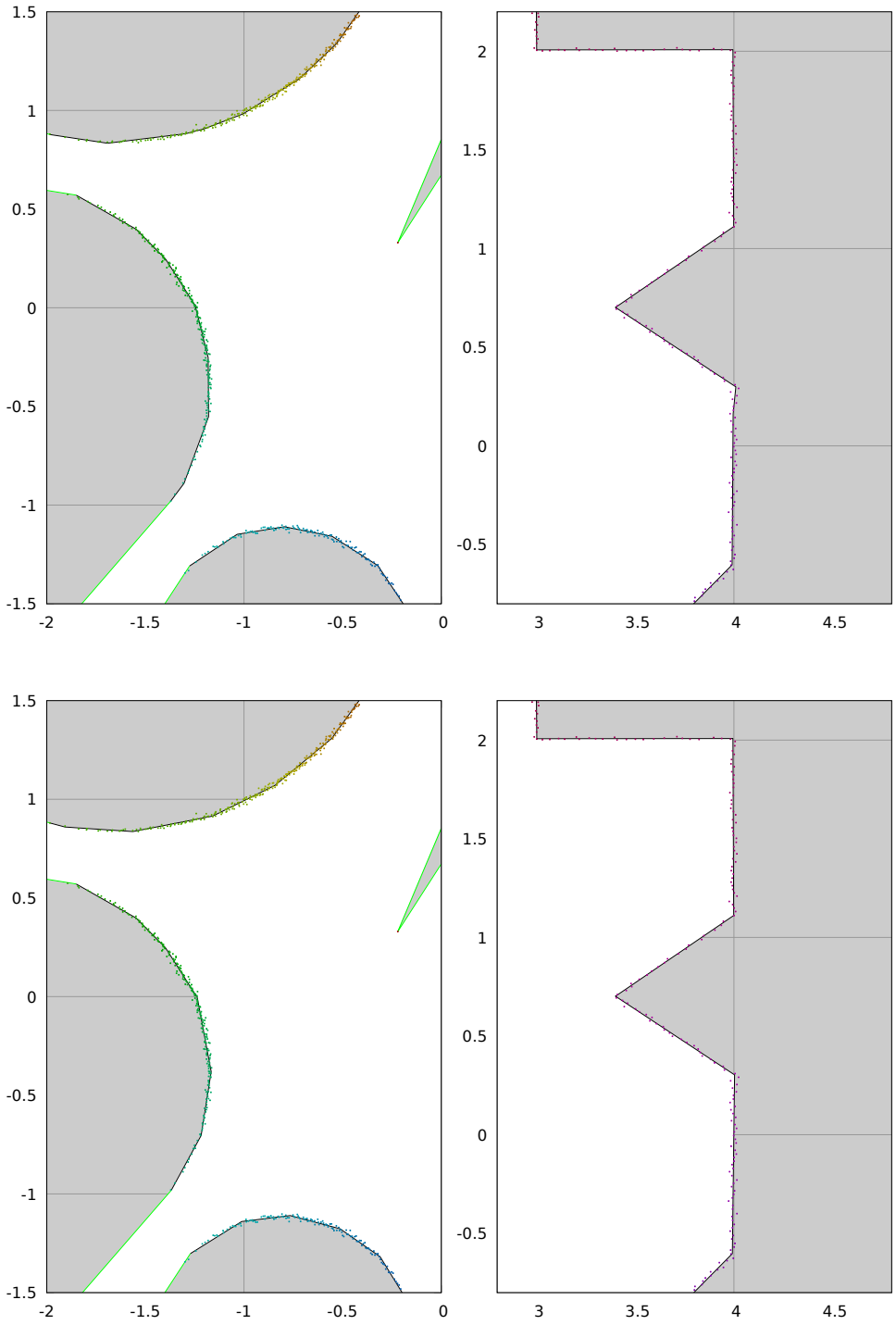


Figure 5.12: Polygon simplification with an inlier threshold of 0.03m (top) and 0.04 (bottom). We no longer have jagged vectors and the surfaces are well approximated.

Threshold	0.005	0.01	0.015	0.02	0.025	0.03	0.035	0.04	0.08
Vector count	791	411	205	114	72	58	54	54	58

Table 5.4: Vector count after the simplification process in relation to the inlier threshold

iments to deduct what set of parameters delivers a good approximation of the environment while keeping the total vector count low. The experiments show, that a good set of parameters for our data sets is $a = 1$, $b = 4$, and $c = 0.001$. In the following we present results that show the influence of the individual parameters by varying a single parameter while leaving the other parameters unchanged. Same as with the inlier threshold, we look in particular at the vector count and the approximation quality.

Table 5.5 shows the vector count for different values of parameter a. The table shows, that the trend is for higher values of a to result in a lower vector count. This is not surprising, as the scoring value with respect to the inlier count approaches 1 much slower, meaning that a higher inlier count is rewarded more that when compared with lower values of parameter a. Overall, this parameter can be used to influence the level of detail (i.e. the resulting number of vectors), but only to some extend. Increasingly large value cause less and less change in the result. A value of $a = 1$ appears to be a good middle ground and works reasonable well for a variety of values for parameters b and c.

Parameter a	0.2	0.5	1.0	2.0	4.0	8.0
Vector count	75	61	58	54	53	51

Table 5.5: Vector count after the simplification process in relation to the scoring parameter a. The remaining scoring parameters were set as $b = 4$ and $c = 0.001$, the inlier threshold was set to 0.03.

Parameter b also allows us to influence how much value we put in higher inlier counts. We can put the emphasis more on lower inlier counter, for example with $a = 0.1$ and $b = 8$ or encourage longer vectors with more inliers with $a = 10$ and $b = 1$. In Figure 5.14 we can see two examples.

Parameter b	0.5	1.0	2.0	4.0	8.0	16.0
Vector count	100	71	62	58	54	52

Table 5.6: Vector count after the simplification process in relation to the scoring parameter b. The remaining scoring parameters were set as $a = 1$ and $c = 0.001$, the inlier threshold was set to 0.03

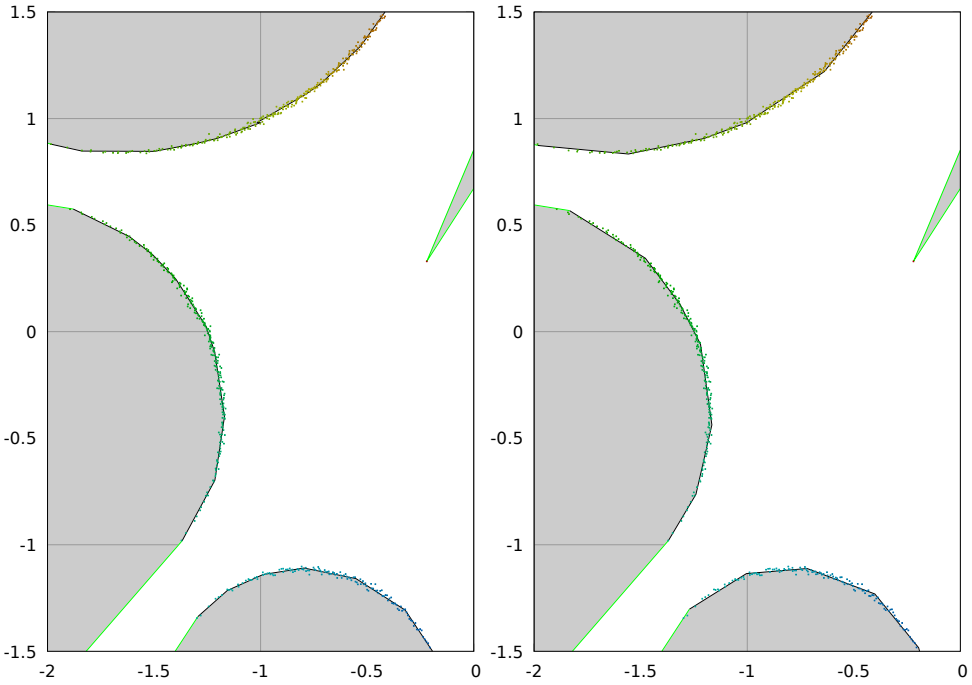


Figure 5.13: Polygon simplification with $a = 0.2$ (left) and $a = 8$ (right). The result on the left requires more vectors, but also provides a slightly better approximation. The remaining scoring parameters were set as $b = 4$ and $c = 0.001$, the inlier threshold was set to 0.03.

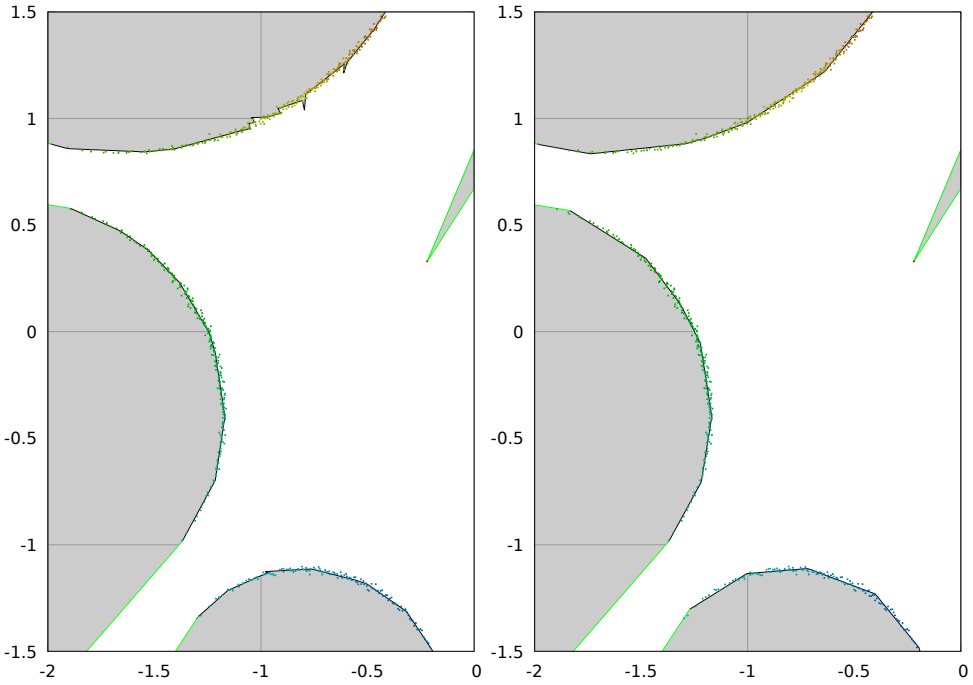


Figure 5.14: Polygon simplification with $b = 0.5$ (left) and $b = 16$ (right). The result on the left is overfitting in some areas, causing a jagged line instead of a smooth curvature. The right polygon has no signs of overfitting, but the approximations are a bit crude due to the low number of vectors available.

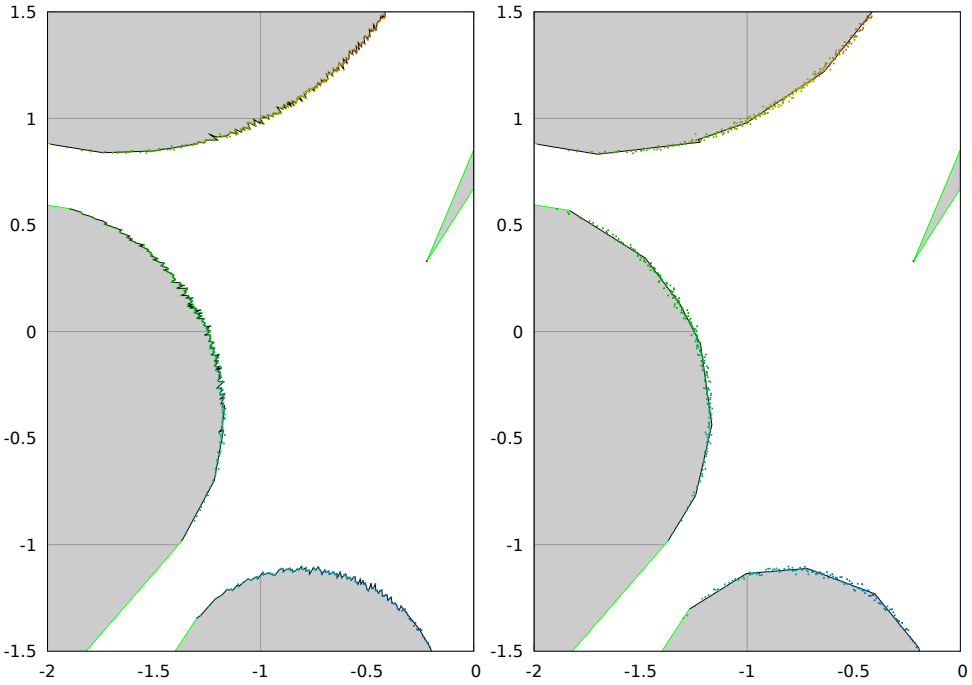


Figure 5.15: Polygon simplification with $c = 0.000001$ (left) and $c = 1$ (right).

Scoring parameter c only modifies the average square error of the inliers. The importance of this parameter becomes quickly apparent when we test values close to zero (zero itself being not an option due possible division by zero that can result then). The algorithm cannot deal with noise well and overfits in many places, causing a distinct zigzag pattern. A large value with respect to the average square error on the other hand results in the error itself to be mostly ignored, and the line fitting process relies heavily on the inlier threshold instead. Figure 5.15 shows two examples, with the zigzag pattern clearly visible in the left part.

Parameter c	0.000001	0.00001	0.0001	0.001	0.01	0.1	1.0
Vector count	893	257	77	58	51	52	52

Table 5.7: Vector count after the simplification process in relation to the scoring parameter c The remaining scoring parameters were set as $a = 1$ and $b = 4$, the inlier threshold was set to 0.03

5.7 Grid Overlay for Vector Maps

Enforcing a grid structure limits the size of the polygons, making it easier to handle them. We tested this grid overlay on a couple of maps from the

simulation data sets. For this we created maps with PolySLAM with three settings: a) without the grid overlay, b) with a 1x1m grid overlay, and c) with a 2x2 grid overlay. Table 5.8 shows the results of the experiment. We note, that applying the grid overlay does not change the number of polygons in a map in a significant way.

Map	no overlay	1mx1m overlay	2mx2m overlay
Loop	209585	225216	214121
Cross	429876	449331	320989
Zigzag	469110	481840	470333

Table 5.8: Impact of grid overlay on polygon count.

5.8 Experiments with data sets from real robots

While simulations have their advantages, in particular regarding ground truth and repeatability, we ultimately want to be able to work in real environments with actual hardware. To this end, we need to evaluate the performance of PolyMap and PolySLAM with data that has been collected with real robots. Given that we are targeting indoor SLAM, we limit ourself to data sets that depict indoor settings, in particular office buildings. We also have some diversity in robot hardware, both in the robots themselves and the laser range finders used.

5.8.1 Intel Research Lab

We decided to perform part of our experiments with public data sets. For this we used PolySLAM on the *Intel Research Lab* public data set [ODS]. Figure 5.16 shows the resulting PolyMap and the trajectory of the robot. The trajectory contains multiple loops, including two big loops around the building. The laser range finder only has an arc of $180°$, making the task more challenging then the other experiments in this paper. The algorithm was applied to the first 2000 keyframes, as soon after misalignments would build up beyond acceptable limits. Still we achieved a consistent map usable by both human and robot localization and navigation.

5.8.2 IMT Lille Douai Lab

Our first own experiments with a real robot was utilizing a Turtlebot2 [Tur] that had a Hokuyo utm-30LX laser range finder [Fin] mounted (30m range, $270°$ with $0.25°$ resolution, 1081 samples per sweep). The robot was teleoperated in the building of our department. The recorded data was used to

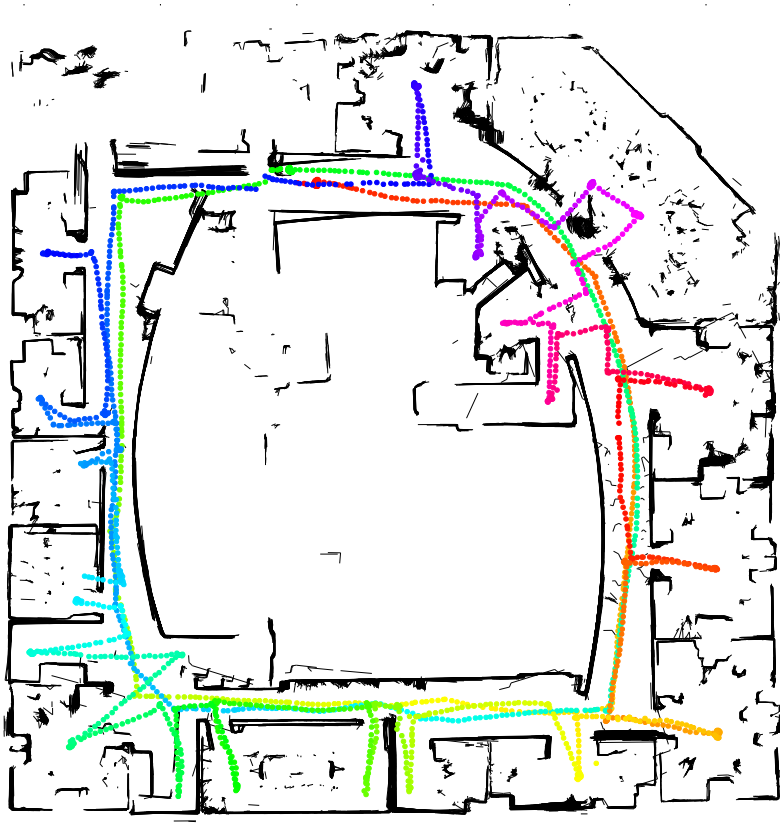
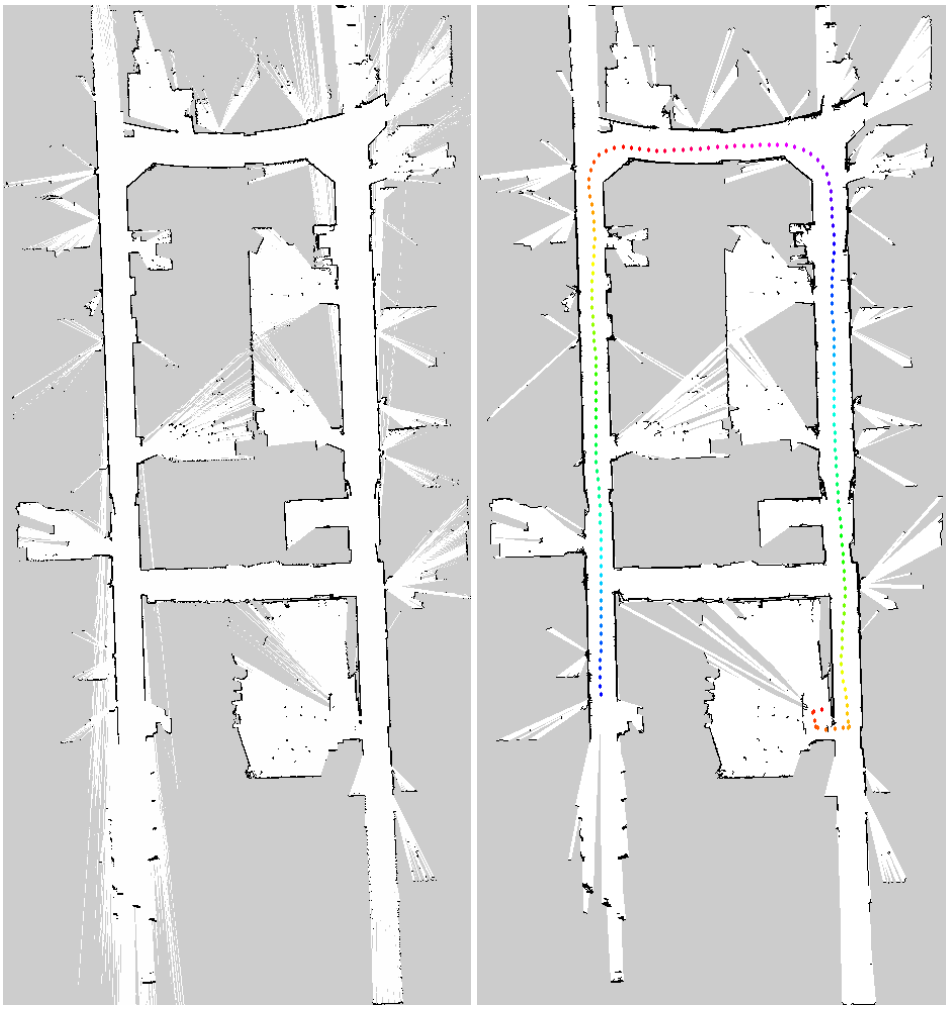


Figure 5.16: PolyMap of the Intel Research Lab. The estimated robot trajectory is rainbow colored for better visibility of overlapping parts of the trajectory.

create a grid-map with Karto [SLA] and a PolyMap with PolySLAM. The robot's trajectory was u-shaped. There is a connecting corridor visible that provides us with a rough estimate about how large the pose estimation drift is by looking for misalignments. By this criteria, our computed map shows no significant errors.



(a) grid-map, created with karto. (b) grid-map, created with PolySLAM.

Figure 5.17: Two grid-maps created from the same data set. Figure b) also shows the robot's trajectory.

Figure 5.17 shows the map from Karto in comparison to the PolyMap that has been converted to a grid-map. Both maps are consistent and show a good approximation of the environment. The robot traveled an estimated distance of 69m, and was able to line up the connecting hallway with almost no visual misalignment. The PolyMap consists of 66296 polygons, build from 265434 vectors. The PolyMap in the figure shows the estimated trajectory of the robot, with the starting position in a room in the bottom right.

5.8.3 Inria Lab

The second experiment uses data collected by Xuan Sang Le using the same robot as in 5.8.2. The experiment was conducted at the Inria research laboratory in Lille. The robot was tele-operated and explored most of the floor, resulting in a longer and more complex trajectory than the previous experiment. The trajectory is shown in Figure 5.18, where every dot corresponds to a keyframe. The thick "knots" along the trajectory are places where the robot was rotating in place to get a better view into rooms adjacent to the hallway. The robot's start and end position are about the same. They are located in the lower left area of the map. The estimated total length of the trajectory is 222m. The map shows some misalignment when the robots re-enters already explored areas, in particular in the lower left area. The misalignment is estimated to be less than 0.5 meters by looking at the generated map. Ground truth for a better error estimate is unfortunately not available. Overall this PolyMap is consistent and usable by both humans and robots alike and shows considerable good alignment when taking the length of the trajectory into account.

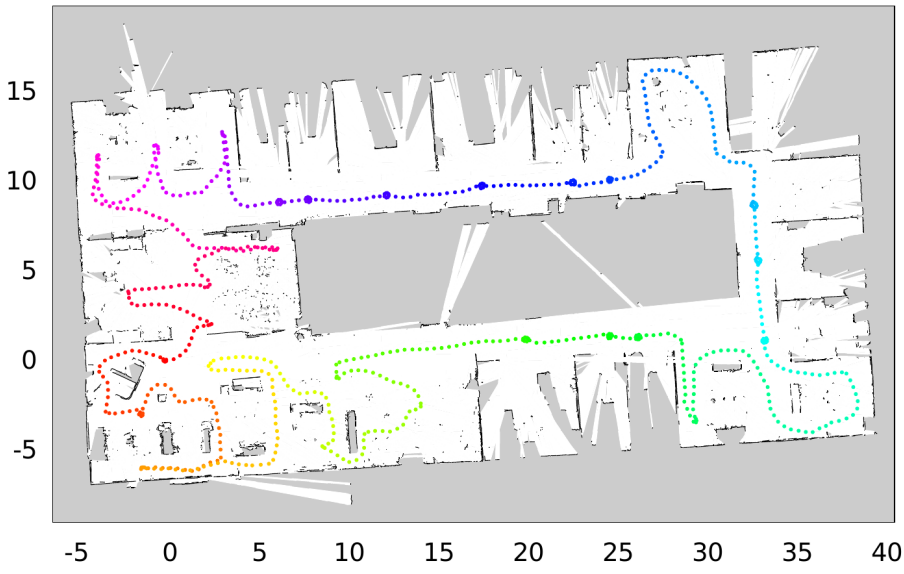


Figure 5.18: PolyMap created from a data set that has been created with a Turtlebot2 at the Inria research laboratory by Xuan Sang Le. The robot started in the lower left corner and move counter-clock-wise in one big loop around the floor, with small detours to explore rooms and other details.

5.9 Summary

In this chapter we conducted several experiments to test the performance of different aspects of our PolySLAM algorithm. We confirmed, that the PolyMap map format is able to produce maps with a smaller memory footprint. We run the PolySLAM algorithm on a variety of environments, from simulation to real world data. The PolyMaps create with PolySLAM are consistent and in terms of the provided NE and SSIM metrics comparable to other state-of-the-art algorithms. The lack of global optimization restricts the capabilities of PolySLAM, but much less so than would be expected. We also test our Navigation stack and find that the topological graph from PolyMaps is in general sparser and thous more lightweight than counterparts created from occupancy grids. Overall, we are satisfied with the results from the experiments.

In the next chapter we will summarize this thesis and the related work, including papers that have been published as result of our progress. We will also take an outlook on potential future work on PolyMap and PolySLAM.

Conclusion

Chapter 6

Contents

6.1 Summary

In this thesis we explore a bottleneck in multi-robot exploration – the high network bandwidth requirements for the required map exchanges. We first take a close look at the current state-of-the-art in regards of 2D SLAM and 2D map formats and find that no existing approach completely satisfies our needs in regard of memory footprint, usability for navigation, and visualization for human use.

To fill this gap we introduce PolyMap, a new vector-based 2D map format, and show that it meets all criteria outlined by us. Key features of our PolyMap format are that the explored/traversable space is fully enclosed by vectors (a useful property for navigation), and the presence of explicit frontiers by means of defining different types of vectors. One type (the sector border) is used to split up polygons, allowing us to partion the map into smaller sections.

Localization is based on dead reckoning with a correction step utilizing an ICP variant. Our ICP implementation makes us of the vector orientation to discard vectors with a normal vector that is pointing away from the sensor center – a technique that we refer to as *backface culling*.

With the PolyMap map format and a Localization technique, we introduce our PolySLAM algorithm that produces PolyMaps. The SLAM algorithm creates keyframes from the sensor data. Special care is taken to reduce the size of keyframes by simplifying polygons to approximate the obstacles in the environment. This simplification process also reduces the influence of sensor noise by use of line fitting techniques on the point cloud delivered by the robots laser range finder. The different tunable parameters of the polygon simplification process are also explained. Experiments on a benchmark map highlight the effect of different parameter values on the vector count of the simplified polygon.

The global map is constructed by the aggregation of local PolyMaps (i.e. keyframes) which are previously aligned by our Localization algorithm. To

accelerate merging process, the global PolyMap is structured with a BSP-tree of convex polygons paving the navigable space. Such maps are compact and suitable for navigation since traversable area is known. Our experiments on different types of maps both in simulation and on real data sets show very consistent maps despite PolyMap does not use any global optimization techniques yet such as loop closure. Pose estimates also show little drift if the environment is structured enough to allow the algorithm to compensate any measurement errors from odometry readings. This is achieved by our point-to-vector approach in pose alignment in combination with our line-fitting technique in the polygon simplification process.

We finally show that the PolyMap format is suitable for navigation. For this, we create a topological graph from the PolyMap while it is embedded in a BSP-tree. We cull any nodes from our topological graph that are not reachable and end up with a relatively sparse graph that allows typical graph search algorithms to perform much faster than on a graph based on a regular grid map.

6.2 Published Papers

Our work lead to several papers, focusing on the topics *Mapping*, *SLAM*, and *Navigation*. By the time of this writing, two of the papers have been presented, while the third has been accepted for presentation in September 2019.

ICIRA 2018. Johann Dichtl, Luc Fabresse, Guillaume Lozenguez, and Noury Bouraqadi; PolyMap: A 2D Polygon-based Map format for Multi-Robot Autonomous Indoor Localization and Mapping; *International Conference on Intelligent Robotics and Applications*; 2018; Springer
This paper introduces our PolyMap format, and compares it to other state-of-the-art map formats. It shows, that the PolyMap format allows to accurately model the environment, while being considerably more lightweight than occupancy grids. At the same time PolyMaps allow to model frontiers explicitly, which is another advantage in the context of multi-robot exploration.

ICARSC 2019. Johann Dichtl, Xuan Sang Le, Luc Fabresse, Guillaume Lozenguez, and Noury Bouraqadi; PolySLAM: A 2D Polygon-based SLAM Algorithm; *International Conference on Autonomous Robot Systems and Competitions*; 2019; IEEE
In this paper we introduces the PolySLAM algorithm. It shows, that the PolyMap format can be utilized in SLAM directly, without creating an occupancy grid as an intermediate step. Furthermore, the paper highlights that the SLAM algorithm experiences very little drift, both in translation and

rotation. As a result, PolySLAM creates consistent maps, despite the lack of global optimization in its current implementation. The paper was awarded with a *Best Paper Award* in the category *Industrial Robot*.

IntelliSys 2019. Johann Dichtl, Xuan Sang Le, Luc Fabresse, Guillaume Lozenguez, and Noury Bouraqadi; Robot Navigation With PolyMap, a Polygon-based Map Format; *Intelligent Systems Conference; Proceedings of SAI Intelligent Systems Conference*; 2019; Springer
This paper focuses on Navigation with PolyMaps. By creating a sparse topological graph from a PolyMap, we have a lightweight and fast way to perform path planning in large environments. When compared to occupancy grids, we see a clear advantage in using the PolyMap format for Navigation tasks.

6.3 Future work

While the PolySLAM implementation shows good result, there are several limitations that leave room for improvement. The most obvious missing feature at the moment is global optimization. Weather a pose-graph-based approach, or a Rao-Blackwellized Particle Filter (RBPF), adding global optimization should further improve the map quality, especially on large maps with opportunities for loop closures.

Hand-in-hand with this is the topic of cross-keyframe polygon simplification which is a different kind of global optimization. This would greatly reduce the number of polygons/vectors and help speed up the computation as the map grows. It would also open the door to preserve fine details in the map that would otherwise be hard to notice due the sensor noise.

The polygon simplification process could also benefit from a dynamic outlier threshold and adaptive parameter tuning, for example by taking the point density into account. This would also eliminate the risk of choosing unfitting parameters by unexperienced robot operators and enables the robot to operate optimal in environments that otherwise would ask for different parameters in different areas, such as an environment that combines indoor and outdoor sections.

Similar, our ICP algorithm could improve in performance if an adaptive outlier threshold would be implemented. And a better pose estimate would improve the overall map quality as well.

Last but not least, looking back at our original motivation, autonomous multi-robot exploration is another future goal. While having a single robot creating PolyMaps is a good result, our aim is to have a fleet of collaborating robots perform efficient autonomous indoor exploration, creating a shared map of the whole environment.

Bibliography

[App66] J Math Anal Appi. A formal basis for the heuristic determination of minimum cost paths. 1966. 15

[BFMG07] Jose-Luis Blanco, Juan-Antonio Fernández-Madrigal, and Javier Gonzalez. A new approach for large-scale localization and mapping: Hybrid metric-topological slam. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 2061–2067. IEEE, 2007. 17

[BFMG08] Jose-Luis Blanco, Juan-Antonio Fernández-Madrigal, and Javier Gonzalez. Toward a unified bayesian approach to hybrid metric–topological slam. *IEEE Transactions on Robotics*, 24(2):259–270, 2008. 17, 18, 26

[BGFM09] José-Luis Blanco, Javier González, and J-A Fernández-Madrigal. Subjective local maps for hybrid metric-topological slam. *Robotics and Autonomous Systems*, 57(1):64–74, 2009. 17

[BJK05] Patrick Beeson, Nicholas K Jong, and Benjamin Kuipers. Towards autonomous topological place detection using the extended voronoi graph. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 4373–4379. IEEE, 2005. 16

[BLFB16] Khelifa Baizid, Guillaume Lozenguez, Luc Fabresse, and Noury Bouraqadi. Vector Maps: A Lightweight and Accurate Map Format for Multi-robot Systems. In *Intelligent Robotics and Applications: 9th International Conference, ICIRA 2016*, pages 418–429. Springer International Publishing, 2016. 14, 48

[BLL92] J. Barraquand, B. Langlois, and J.-C. Latombe. Numerical potential field techniques for robot path planning. *IEEE Transactions on Systems, Man and Cybernetics*, 22:224–241, 1992. 35

[BR05] Emma Brunskill and Nicholas Roy. Slam using incremental probabilistic pca and dimensionality reduction. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 342–347. IEEE, 2005. 14

[BSBA11] Wolfram Burgard, Cyrill Stachniss, Maren Bennewitz, and Kai Arras, 2011. 19

[Cen08] Andrea Censi. An icp variant using a point-to-line metric. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 19–25. IEEE, 2008. 24, 51

[CL85] Raja Chatila and Jean-Paul Laumond. Position referencing and consistent world modeling for mobile robots. In *Robotics and Automation. Proceedings. 1985 IEEE International Conference on*, volume 2, pages 138–145. IEEE, 1985. 14

[CMNT99] Jose A Castellanos, JMM Montiel, José Neira, and Juan D Tardós. The spmap: A probabilistic framework for simultaneous localization and map building. *IEEE Transactions on Robotics and Automation*, 15(5):948–952, 1999. 12, 13

[CN10] César Cadena and José Neira. Slam in o (logn) with the combined kalman-information filter. *Robotics and Autonomous Systems*, 58(11):1207–1219, 2010. 22

[CQW+17] Yongfu Chen, Chunlei Qu, Qifu Wang, Zhiyong Jin, Mengzhu Shen, and Jiaqi Shen. Tvslam: An efficient topological-vector based slam algorithm for home cleaning robots. In *International Conference on Intelligent Robotics and Applications*, pages 166–178. Springer, 2017. 28, 38, 39, 43

[Cro85] J Crowley. Navigation for an intelligent mobile robot. *IEEE Journal on Robotics and Automation*, 1(1):31–41, 1985. 14

[DLL+19] Johann Dichtl, Xuan Sang Le, Guillaume Lozenguez, Luc Fabresse, and Noury Bouraqadi. Robot navigation with polymap, a polygon-based map format. In *Proceedings of SAI Intelligent Systems Conference*. Springer, 2019. 14

[E+90] Alberto Elfes et al. Occupancy grids: A stochastic spatial representation for active robot perception. In *Proceedings of the Sixth Conference on Uncertainty in AI*, volume 2929, page 6, 1990. 11

[ECL10] Jan Elseberg, Ross T Creed, and Rolf Lakaemper. A line segment based system for 2d global mapping. In *2010 IEEE International Conference on Robotics and Automation*, pages 3924–3931. IEEE, 2010. 31, 41, 42

[FBDT99] Dieter Fox, Wolfram Burgard, Frank Dellaert, and Sebastian Thrun. Monte carlo localization: Efficient position estimation for mobile robots. In *Proc. of the National Conference on Artificial Intelligence*, volume 1999, pages 2–2, 1999. 10, 20, 21

[FHL+03] Dieter Fox, Jeffrey Hightower, Lin Liao, Dirk Schulz, and Gaetano Borriello. Bayesian filtering for location estimation. *IEEE pervasive computing*, 2(3):24–33, 2003. 40

[Fin] Hokuyo Laser Range Finder. https://www.hokuyo-aut.jp/search/single.php?serial=169. 99

[FT07] Amalia F. Foka and Panos E. Trahanias. Real-time hierarchical POMDPs for autonomous robot navigation. *Robotics and Autonomous Systems*, 55(7):561–571, 2007. 35

[Gaz] Gazebo. http://gazebosim.org/. 84

[GD13] Dinesh Gamage and Tom Drummond. Reduced dimensionality extended kalman filter for slam. In *BMVC*, 2013. 22

[Gera] Brian Gerkey. Ros gmapping node. 11, 27, 40

[Gerb] Brian P. Gerkey. 20

[GKSB10] Giorgio Grisetti, Rainer Kummerle, Cyrill Stachniss, and Wolfram Burgard. A tutorial on graph-based SLAM. *IEEE Intelligent Transportation Systems Magazine*, 2(4):31–43, 2010. 22

[Goo16] Google, 2016. 11

[Gro16] IEEE RAS Map Data Representation Working Group. Ieee standard for robot map data representation for navigation, sponsor: Ieee robotics and automation society. June 2016. 10, 13, 15

[Grü13] Branko Grünbaum. *Convex Polytopes*, volume 221. Springer Science & Business Media, 2013. 45

[GSB05] G. Grisetti, C. Stachniss, and W. Burgard. Improving Grid-based SLAM with Rao-Blackwellized Particle Filters by Adaptive Proposals and Selective Resampling. In *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 2443–2448, 2005. 19, 25, 27

[GSB07] G. Grisetti, C. Stachniss, and W. Burgard. Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters. *IEEE Transactions on Robotics*, 23(1):34–46, Feb. 2007. 27, 40

[HBOZ] Richard Hanten, Sebastian Buck, Sebastian Otte, and Andreas Zell. Vector-AMCL: Vector based Adaptive Monte Carlo Localization for Indoor Maps. 32

[HKL13] Guoquan Huang, Michael Kaess, and John J Leonard. Consistent sparsification for graph optimization. In *Mobile Robots (ECMR), 2013 European Conference on*, pages 150–157. IEEE, 2013. 34

[HKRA16] Wolfgang Hess, Damon Kohler, Holger Rapp, and Daniel Andor. Real-time loop closure in 2d lidar slam. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1271–1278, 2016. 11, 26, 28, 40

[Int] SRI International. 27

[Jel15] Ales Jelinek. Vector maps in mobile robotics. *Acta Polytechnica CTU Proceedings*, 2(2):22–28, 2015. 15, 29, 30, 38, 41

[JH03] Timothée Jost and Heinz Hugli. A multi-resolution icp with heuristic closest point search for fast and robust 3d registration of range images. In *3-D Digital Imaging and Modeling, 2003. 3DIM 2003. Proceedings. Fourth International Conference on*, pages 427–433. IEEE, 2003. 24

[K+60] Rudolph Emil Kalman et al. A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, 82(1):35–45, 1960. 22

[KGS+11] Rainer Kümmerle, Giorgio Grisetti, Hauke Strasdat, Kurt Konolige, and Wolfram Burgard. g 2 o: A general framework for graph optimization. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 3607–3613. IEEE, 2011. 22, 23, 26, 33

[KL00] J.J. Kuffner and S.M. LaValle. RRT-connect: An efficient approach to single-query path planning. In *International Conference on Robotics and Automation*, volume 2, pages 995–1001, 2000. 35

[KMvSK11] S. Kohlbrecher, J. Meyer, O. von Stryk, and U. Klingauf. A flexible and scalable slam system with full 3d motion estimation. In *Proc. IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*. IEEE, November 2011. 28

[KSD+09] Rainer Kümmerle, Bastian Steder, Christian Dornhege, Michael Ruhnke, Giorgio Grisetti, Cyrill Stachniss, and Alexander Kleiner. On measuring the accuracy of slam algorithms. *Autonomous Robots*, 27(4):387, 2009. 37

[Kui00] Benjamin Kuipers. The spatial semantic hierarchy. *Artificial intelligence*, 119(1-2):191–233, 2000. 36

[KW94] David Kortenkamp and Terry Weymouth. Topological mapping for mobile robots using a combination of sonar and vision sensing. In *AAAI*, volume 94, pages 979–984, 1994. 15

[LAB+12] G. Lozenguez, L. Adouane, A. Beynier, P. Martinet, and A.-I. Mouaddib. Interleaving Planning and Control of Mobiles Robots in Urban Environments Using Road-Map. In *International Conference on Intelligent Autonomous Systems*, 2012. 35

[Lav98] Steven M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Technical report, 1998. 10, 15

[LaV06] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, New York, NY, USA, 2006. 47

[LFBL18] Xuan Sang Le, Luc Fabresse, Noury Bouraqadi, and Guillaume Lozenguez. Evaluation of out-of-the-box ros 2d slams for au-

tonomous exploration of unknown indoor environments. In *International Conference on Intelligent Robotics and Applications*, pages 283–296. Springer, 2018. 40, 81, 84

[LFP13] Gim Hee Lee, Friedrich Fraundorfer, and Marc Pollefeys. Robust pose-graph loop-closures with expectation-maximization. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 556–563. IEEE, 2013. 27

[LLW05] Rolf Lakaemper, Longin Jan Latecki, and Diedrich Wolter. Incremental multi-robot mapping. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3846–3851. IEEE, 2005. 14, 31

[LT03] Yufeng Liu and Sebastian Thrun. Results for outdoor-slam using sparse extended information filters. In *2003 IEEE International Conference on Robotics and Automation (Cat. No. 03CH37422)*, volume 1, pages 1227–1233. IEEE, 2003. 22

[MdCT18] Kimberly McGuire, Guido de Croon, and Karl Tuyls. A comparative study of bug algorithms for robot navigation. *CoRR*, abs/1808.05050, 2018. 34

[ME85] Hans Moravec and Alberto Elfes. High resolution maps from wide angle sonar. In *Proceedings. 1985 IEEE international conference on robotics and automation*, volume 2, pages 116–121. IEEE, 1985. 11

[MEBF+10] Eitan Marder-Eppstein, Eric Berger, Tully Foote, Brian Gerkey, and Kurt Konolige. The office marathon: Robust navigation in an indoor office environment. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 300–307. IEEE, 2010. 35

[MP00] Roman Mazl and Libor Preucil. Building a 2d environment map from laser range-finder data. In *Intelligent Vehicles Symposium, 2000. IV 2000. Proceedings of the IEEE*, pages 290–295. IEEE, 2000. 14

[MW10] Michael Milford and Gordon Wyeth. Hybrid robot control and slam for persistent navigation and mapping. *Robotics and Autonomous Systems*, 58(9):1096–1104, 2010. 16

[ODS] Uni Bonn Online Data Sets. http://www.ipb.uni-bonn.de/datasets/. 81, 82, 99

[PCS15] François Pomerleau, Francis Colas, and Roland Siegwart. A Review of Point Cloud Registration Algorithms for Mobile Robotics. *Found. Trends Robot*, 4(1):1–104, May 2015. 23

[PDM+07] Luis Pedraza, Gamini Dissanayake, Jaime Valls Miro, Diego Rodriguez-Losada, and Fernando Matia. Bs-slam: Shaping the world. In *Robotics: Science and Systems*, 2007. 29, 30

[Pds] Johann Dichtl PolySLAM data sets. http://car.imt-lille-douai.fr/polyslam/. 82

[PJTN07] Lina María Paz, Patric Jensfelt, Juan D Tardós, and José Neira. Ekf slam updates in o(n) with divide and conquer slam. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 1657–1663. IEEE, 2007. 38

[RL01] Szymon Rusinkiewicz and Marc Levoy. Efficient variants of the icp algorithm. In *3-D Digital Imaging and Modeling, 2001. Proceedings. Third International Conference on*, pages 145–152. IEEE, 2001. 23, 51

[RST+12] Jörg Röwekämper, Christoph Sprunk, Gian Diego Tipaldi, Cyrill Stachniss, Patrick Pfaff, and Wolfram Burgard. On the position accuracy of mobile robot localization based on particle filters combined with scan matching. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3158–3164. IEEE, 2012. 37

[SBG02] D. Schroter, M. Beetz, and J. S. Gutmann. RG mapping: learning compact and structured 2D line maps of indoor environments. In *Robot and Human Interactive Communication, 2002. Proceedings. 11th IEEE International Workshop on*, pages 282–287, 2002. 14

[SD98] S. Simhon and G. Dudek. A global topological map formed by local metric maps. In *International Conference on Intelligent Robots and Systems*, volume 3, pages 1708–1714, 1998. 17

[SGB05] Cyrill Stachniss, Giorgio Grisetti, and Wolfram Burgard. Recovering particle diversity in a rao-blackwellized particle filter for slam after actively closing loops. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 655–660. IEEE, 2005. 25, 26, 33

[SGHB04] Cyrill Stachniss, Giorgio Grisetti, Dirk Hähnel, and Wolfram Burgard. Improved rao-blackwellized mapping by adaptive sampling and active loop-closure. In *In Proc. of the Workshop on Self-Organization of AdaptiVE behavior (SOAVE)*, 2004. 8, 10

[SJH98] C Schutz, Timothée Jost, and H Hugli. Multi-feature matching algorithm for free-form 3d surface registration. In *Pattern Recognition, 1998. Proceedings. Fourteenth International Conference on*, volume 2, pages 982–984. IEEE, 1998. 24

[SK95] Reid Simmons and Sven Koenig. Probabilistic robot navigation in partially observable environments. In *IJCAI*, volume 95, pages 1080–1087, 1995. 20

[SK08] Hee Jin Sohn and Byung Kook Kim. An efficient localization algorithm based on vector matching for mobile robots using laser range finders. *Journal of Intelligent and Robotic Systems*, 51(4):461–488, 2008. 29, 40

[SK09] Hee Jin Sohn and Byung Kook Kim. VecSLAM: An Efficient Vector-Based SLAM Algorithm for Indoor Environments. *Journal of Intelligent and Robotic Systems*, 56(3):301–318, 2009. 29, 38, 40

[SLA] Karto SLAM. http://wiki.ros.org/slam_karto. 27, 101

[SMD10] Hauke Strasdat, JMM Montiel, and Andrew J Davison. Scale drift-aware large scale monocular slam. *Robotics: Science and Systems VI*, 2, 2010. 26

[SP12] Niko Sünderhauf and Peter Protzel. Switchable constraints for robust pose graph slam. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 1879–1884. IEEE, 2012. 34

[SP13a] Niko Sünderhauf and Peter Protzel. Switchable constraints vs. max-mixture models vs. rrr-a comparison of three approaches to robust pose graph slam. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 5198–5203. IEEE, 2013. 8

[SP13b] Niko Sünderhauf and Peter Protzel. Switchable constraints vs. max-mixture models vs. rrr-a comparison of three approaches to robust pose graph slam. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 5198–5203. IEEE, 2013. 27

[SPR13] Joao Machado Santos, David Portugal, and Rui P Rocha. An evaluation of 2d slam techniques available in robot operating system. In *Safety, Security, and Rescue Robotics (SSRR), 2013 IEEE International Symposium on*, pages 1–6. IEEE, 2013. 27, 40, 82

[SRI10] SRI International. Karto slam (http://www.ros.org/wiki/karto), 2010. 27

[SSC90] Randall Smith, Matthew Self, and Peter Cheeseman. Estimating uncertain spatial relationships in robotics. In *Autonomous robot vehicles*, pages 167–193. Springer, 1990. 22, 26

[TBF05] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. Probabilistic robotics (intelligent robotics and autonomous agents). 01 2005. 32

[Thr98] Sebastian Thrun. Learning metric-topological maps for indoor mobile robot navigation. *Artificial Intelligence*, 99(1):21–71, 1998. 16, 36

[THSW07] Tong Tao, Yalou Huang, Fengchi Sun, and Tingting Wang. Motion planning for slam based on frontier exploration. In *2007 International Conference on Mechatronics and Automation*, pages 2120–2125. IEEE, 2007. 39

[TN87] William C. Thibault and Bruce F. Naylor. Set operations on polyhedra using binary space partitioning trees. *SIGGRAPH Comput. Graph.*, 21(4):153–162, August 1987. 16

[Tur] Turtlebot2. https://www.turtlebot.com/turtlebot2/. 99

[VLE10] Regis Vincent, Benson Limketkai, and Michael Eriksen. Comparison of indoor robot localization techniques in the absence of gps. In *Detection and Sensing of Mines, Explosive Objects, and Obscured Targets XV*, volume 7664, page 76641Z. International Society for Optics and Photonics, 2010. 37

[WBSS04] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004. 82

[Wel85] Emo Welzl. Constructing the visibility graph for n-line segments in o (n2) time. *Information Processing Letters*, 20(4):167–171, 1985. 36

[WSG10] K.M. Wurm, C. Stachniss, and G. Grisetti. Bridging the Gap Between Feature- and Grid-based SLAM. *Robotics and Autonomous Systems*, 58(2):140–148, 2010. 12

[WVDM00] Eric A Wan and Rudolph Van Der Merwe. The unscented kalman filter for nonlinear estimation. In *Adaptive Systems for Signal Processing, Communications, and Control Symposium 2000. AS-SPCC. The IEEE 2000*, pages 153–158. Ieee, 2000. 22

[Yam97] Brian Yamauchi. A frontier-based approach for autonomous exploration. In *cira*, page 146. IEEE, 1997. 39

[ZG00] L. Zhang and B. K. Ghosh. Line segment based map building and localization using 2D laser rangefinder. In *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, volume 3, pages 2538–2543 vol.3, 2000. 14

SLAM 2d à base de polygones pour de grands espaces intérieur
Une solution à base de polygones

Le SLAM d'espaces intérieurs est un sujet important en robotique. La majorité des solutions actuelles se basent sur une carte sous forme de grille 2D. Bien que permettant de réaliser des cartographies satisfaisantes, cette solution admet des limites liées à la quantité importante de mémoire qu'elle requiert. Dans cette thèse, nous introduisons PolySLAM un algorithme de SLAM qui permet de produire des cartes vectorielles 2D à base de polygones.

Mots-clés: SLAM,Navigation,Exploration multi-robot,Robotique

On 2D SLAM for Large Indoor Spaces
A Polygon-based Solution

Indoor SLAM and exploration is an important topic in robotics. Most solutions today work with a 2D grid representation as map model, both for the internal data format and for the output of the algorithm. While this is convenient in several ways, it also brings its own limitations, in particular because of the memory requirements of this map format. In this thesis we introduce PolyMap, a 2D map format aimed at indoor mapping, and PolySLAM, a SLAM algorithm that produces PolyMaps.

Keywords: SLAM, Navigation, Multi-Robot Exploration, Robotics