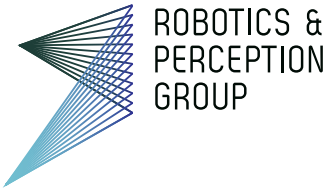
University of Zurich
Department of Informatics



Andreas Ziegler

A Representation for Exploration that is Robust to State Estimate Drift

Master Thesis

Robotics and Perception Group
University of Zurich

Supervision

Titus Cieslewski
Prof. Dr. Davide Scaramuzza
Prof. Dr. Roland Siegwart

April 2018

Contents

Abstract

Exploration is a fundamental task in the field of robotics. The goal is to build a map of a previously unknown environment. Two tasks have to be performed repeatedly for exploration: mapping the space the robot so far perceived; and planning where to go next. In this thesis we focus on the first task and the goal is to find a map representation that can deal with noisy state estimates. All so far presented map representations either assume perfect state estimates or need to rebuild the map after optimization.

To achieve this goal we build our representation with polygons. Our polygons represent the boundary between free known space and occupied space or unknown space and the inside of polygons is implicitly free space. We develop two approaches: The first approach builds a global map with polygons by continuously building the union of the polygon from the current field of view and the polygons of the so far explored space. The second approach works with polygons of the local field of view only. For every local polygon the frontiers, the obstacles and the free space is determined and the robot explores as long frontiers are present in the map.

In this thesis we propose a novel representation that can deal with noisy state estimates and does not need to rebuild the map or parts of it, e.g. after a loop closure. By experiments we show that we achieve full coverage of the area to explore with frontier-based exploration, using our proposed representation.

Nomenclature

Notation

$\mathbf{T}_{WB}$	coordinate transformation from frame B to frame W
$\mathbf{R}_{WB}$	orientation of B with respect to W
${}_W\mathbf{t}_{WB}$	translation of B with respect to W, expressed in coordinate system W

Scalars are written in lower case letters (a), vectors in lower case bold letters ($\mathbf{a}$) and matrices in upper case bold letters ($\mathbf{A}$).

Acronyms and Abbreviations

TSDF Truncated Signed Distance Field

NBV Next-best-view

FOV Field of View

SLAM Simultaneous Localisation and Mapping

Chapter 1

Introduction

Exploration is a fundamental task in the field of robotics. The goal is to build a map from a previously unknown environment [4]. To do this, two essential tasks have to be performed repeatedly: mapping the space the robot currently perceives; and planning where to go next [8]. There is already a large body of literature covering a variety of aspects of these two tasks, e.g. [21], [8], [12], [4], [11], [14] and [16]. In this work here, the focus is on the first task, building a map of the already explored environment. The goal is to find a representation for such a map, that is pose graph based and can deal with a pose graph exhibiting drift. This is in contrast to the so far proposed map representations that assume a perfect state estimation and therefore a drift-free pose graph. If this assumption does not hold, these representation will create an incorrect map.

1.1 Related Work

1.1.1 Map representations

Map representations can be divided into three different categories [18]:

- Metric representations:
  - Occupancy based representations
  - Feature based representations:
    - Geometric (lines, curves, planes)
    - Landmarks
- Topological representations
- Combinations / hybrid representations

Metric representations

Metric representations, as defined in [18], express spatial relations between basic entities implicitly by providing coordinates for each of the spatial objects within a single absolute coordinate system.

Metric representations have the advantage that the information they provide can directly be used for tasks such as planning or obstacle avoidance. To build a global map with a metric representation, the pose estimation and the free space detection must be good enough that the effects of uncertainty can be ignored [4]. Otherwise, the map can become inconsistent.

Recently OctoMap [11], an occupancy based representation has received a lot of attention. In OctoMap, the map is represented as a collection of occupancy probabilities stored over a voxel grid in a hierarchival octree structure. An example of such an OctoMap is shown in Figure 1.1.

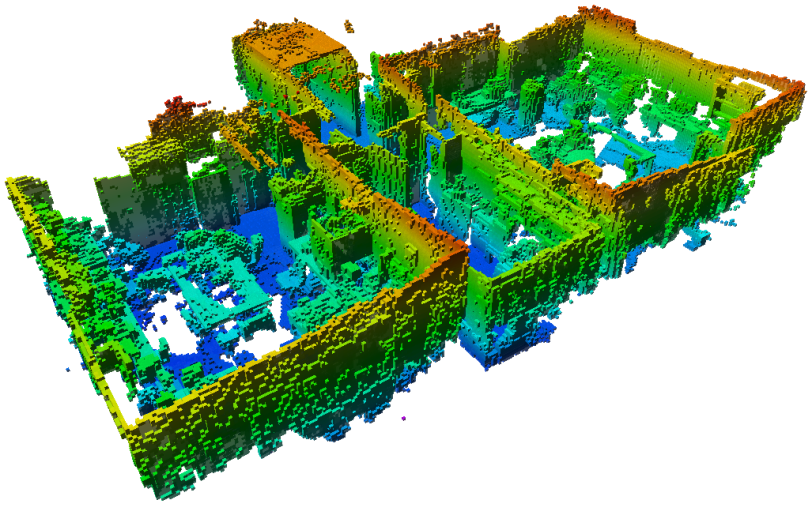


Figure 1.1: An example of an OctoMap (Figure taken from the ROS documentation http://wiki.ros.org/ccny_rgbd/keyframe_mapper).

Truncated Signed Distance Fields (TSDFs) are another metric representation, originally used as an implicit 3D volume representation for graphics, which became popular with KinectFusion [15]. A metric map representation that uses lines and curves as features to build a map was proposed in [8]. In this approach, the map is built with so called solid curves and free curves. Solid curves represent obstacles and free curves join each pair of successive solid curves.

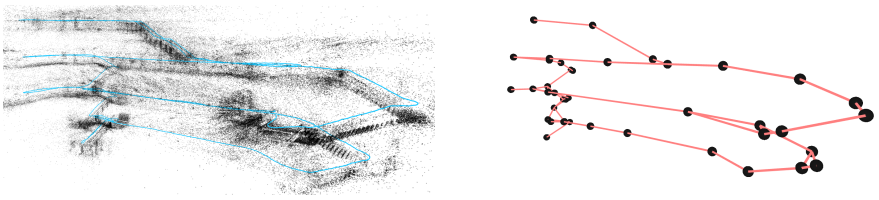
The limitation of metric based representations is, that they assume perfect pose estimation. If the pose estimation is noisy, the map can become inconsistent.

Topological representations

Topological representations express spatial relations between basic entities by explicitly stating that a certain relation holds between a certain set of objects [18].

Most topological representations are based on a graph with vertices and edges, where the vertices represent certain locations and the edges their relationship. Topological representations mostly do not contain very detailed information of the environment. This results in map representations that scale well with the size of the map (in terms of data size).

In [1], a topological map is built from a sparse feature-based map of a visual Simultaneous Localisation and Mapping (SLAM) system. An example is shown in Figure 1.2. This topological map is then used for global path planning and the authors show that their approach achieves similar performance as other systems but with significantly lower computation time and storage requirements.



(a) Sparse feature-based map of a SLAM system

(b) Topological map

Figure 1.2: An example of a topological map generated by Topomap (Figures taken from [1]).

Topological map representations alone are not so widely used in the robotics community. The reason for this is that topological map representations only allow a robot to perform actions in a more global scale, as details of the environment are not provided.

Hybrid representations

Most combinations of map representations are topological-metric representations, often called “topometric”. The advantage of such a hybrid approach is, that they inherit the advantages of both, the metric as well as the topological representation.

Most topometric representations build a local metric map for every vertex of the topological map. This way, each map’s uncertainties can be modeled with respect to its own local coordinate frame as explained in [2]. In contrast to metric only representations, this allows to build maps with noisy state estimates. The reason is, that in a local map the drift of the trajectory can be neglected as well as for the edge between two adjacent vertices of the topological map. There are several proposed approaches which all follow this idea, e.g. [2], [6], [17] and [14].

In the approach presented in [17], the global metric map is divided into submaps and a global topological graph is formed on the map. An example is shown in Figure 1.3.This allows recomputing only isolated areas of the map, whereas others can remain unchanged. The topological graph allows efficient path planning on the hybrid map.

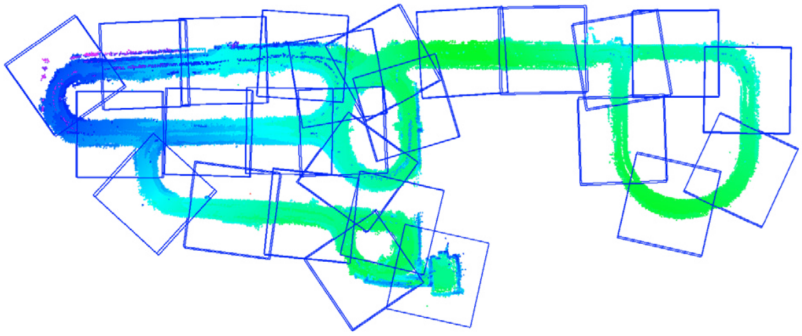


Figure 1.3: An example of a topometric map (Figure taken from [17]).

Limitations

The discussed related work all have their advantages and limitations.

Metric map representations cannot handle noisy state estimates and often the whole map needs to be rebuilt if a part of the map is distorted and needs to be corrected [8].

Topological representations, on the other hand, do not provide local details which are required for exploration.

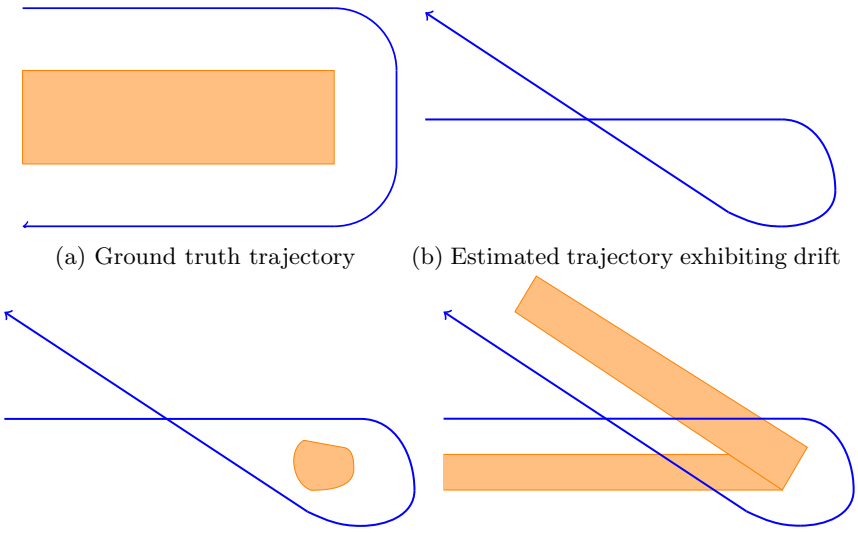
The ability of topometric representations to handle noisy state estimates well with local sub maps makes them an appealing representation. Some topometric representations however build local occupancy maps which are not well suited for deformations and must be rebuilt, in case the sub maps change [17].

Another disadvantage most map representations come with, is the fact that they do not directly represent frontiers but only free, occupied and unknown space. The direct representation of frontiers however is beneficial for frontier-based exploration.

Conclusion

In this thesis we want to develop a map representation which overcomes the limitations mentioned. The main goal will be to handle a pose graph exhibiting drift due to noisy state estimates. An example situation is illustrated in Figure 1.4, where the ground truth trajectory is shown in Figure 1.4a and the estimated trajectory is shown in Figure 1.4b. In a grid based representation the resulting map would look like the one shown in Figure 1.4c. But what we want is a representation as shown in Figure 1.4d which maps the environment

correctly with respect to the estimated trajectory. This situation is described in more detail in Section 2.7.1.



(a) Ground truth trajectory

(b) Estimated trajectory exhibiting drift

(c) With a metric, grid based representation

(d) Desired representation

Figure 1.4: (a) An example of a trajectory with an obstacle in the middle. (b) The estimated trajectory e.g. the output of a visual(-inertial) odometry system. (c) A failure case with a grid based map representation. (d) A correct map representation relative to the trajectory.

1.1.2 Path planning for exploration

In order to understand how to build a good representation, we need to understand how it is used. In our case, we want to use the representation for exploration. As mentioned in the introduction, in Chapter 1, for exploration, a robot has to perceive its environment and to plan its next move.

Therefore, in path planning for exploration, at every step the robot has to decide where to move next. Ideally, at the next location, the robot can perceive more of the previously unseen environment. There are two main planning approaches in the literature: Next-best-view (NBV) planner and frontier-based planner.

The first one tries to choose the next position in such a way that the perception is optimal and is known as the so called NBV problem. The NBV problem, known from the computer vision community, has already been studied since several decades [5], [13].

The other approach follows a more simple, but often also really efficient scheme. The strategy is to navigate to the closest so-called frontier. Frontiers are defined as the border between free space and unknown space. This approach is called frontier-based exploration and was introduced in [21].

Next-best-view (NBV) planner

Most NBV planners determine the next best view by sampling candidate points and evaluating their information gain with a utility function [8], [16]. The advantage of these approaches is, that the utility function can be adapted to different scenarios and can also contain additional constraints, e.g. the motion model of the robot. The disadvantage in practice is, that depending on the sampling strategy and the utility function, it can take time to evaluate the next best view point and this can result in a stop-and-go behavior in which the robot moves for a while, stops to perform the calculation and then moves again [4].

Frontier-based exploration planner

Frontier-based exploration planner are simpler in the way that they do not involve sampling of candidate points or the evaluation of a utility function. Because of this, frontier-based exploration planner do not allow for custom utility functions compared to NBV planners but are faster and as shown in [10], the frontier-based approach outperforms the NBV approach presented in [8].

Proof of coverage

We proof achieving full coverage with a frontier-based exploration approach by contradiction: We assume that the exploration is finished and there is still accessible unknown space left. If the unknown space is not accessible, e.g. if it is surrounded by obstacles, we do not consider it as unknown space as it is not possible for the robot to perceive this space. If there is unknown space that is accessible, it is adjacent to known free space, otherwise it would not be accessible. The boundary between known free and unknown space defines frontiers and if there are frontiers, the exploration is not finished which is a contradiction.

Conclusion

The work in [4] has shown, that frontier-based exploration is the favorable choice when working with multi-rotors. As our target platform are multi-rotors, we will use a frontier-based exploration approach for our work.

Chapter 2

Methodology

The methodology starts by motivating the later presented representation approaches. From the drawbacks of previous work, the required key properties for a representation are deduced. We then introduce the terminology and the polygon based representation needed for the explanation of the approaches. Afterwards we explain the two approaches and their building blocks.

2.1 Motivation

Previous approaches have their limitations as mentioned in Section 1.1.1. In this thesis we want to develop a new map representation which overcomes these limitations and is optimal to perform exploration.

The key properties for such a representation are the following:

- Robustness to drift: Capable of handling noisy state estimates
- Deformable: e.g. after a loop closure
- Frontier-based: Applicable to frontier-based exploration

2.2 Terminology

As explained in Chapter 1, the goal of exploration is to build a map from a previously unknown environment. As a first attempt, we reduce our work to 2D space. The approach, however, is extendable to 3D space. In a more formal language, $\mathcal{V} \in \mathbb{R}^2$ is a bounded subspace of $\mathbb{R}^2$ that represents the unknown environment we would like to explore. $\mathcal{V}$ consists of free and occupied space $\mathcal{V} = \mathcal{V}_{\text{free}} \cup \mathcal{V}_{\text{occ}}$. With a robot that can measure the free space around itself, we can explore $\mathcal{V}_{\text{free}}$. The Field of View (FOV) of the robot at pose $\mathtt{T}_{W,R}$ is denoted as $\mathcal{V}_{\text{fov}}(\mathtt{T}_{W,R}) \subset \mathcal{V}_{\text{free}}$. While a robot is moving on a trajectory $\mathtt{T}_{W,R}(t)$, it will measure $\mathcal{V}_{\text{fov}}$ at consecutive sampling times $t_0, t_1, \ldots$. The poses corresponding to the sampling times are denoted with $\mathtt{T}_{W,R_0}, \mathtt{T}_{W,R_1}, \ldots$

In the pose graph, the aforementioned poses model the vertices/nodes and the edge between two vertices represents a spatial constraint relating the two robot poses [9].The edges therefore represent the transformations $\mathtt{T}_{R_{k-1},R_k}$.

The so far explored space is the union of all the FOVs measured so far and is denoted as

$$\bar{\mathcal{V}}_{\text{free}} = \bigcup_k \mathcal{V}_{\text{fov}}(\mathtt{T}_{W,R_k}). \tag{2.1}$$

Exploration is thus complete if $\bar{\mathcal{V}}_{\text{free}} = \mathcal{V}_{\text{free}}$.

2.3 Polygon based representation

The proposed map representations are based on polygons which provide interesting characteristics. Polygons can represent complex environments at any degree of precision. In our case, polygons represent the boundary, $\partial\bar{\mathcal{V}}_{\text{free}}$, between free space and unknown or occupied space. Hence, $\bar{\mathcal{V}}_{\text{free}}$ is implicitly the inside of the polygons.

In Section 2.4 and Section 2.5 it is explained in depth how a polygon is build from depth measurements and how it is defined. The explanation how a polygon can be extended by building the union with another polygon is covered in Section 2.6.

2.4 Polygon construction from depth measurements

We use a RGBD camera as source to build the polygon of the robot's FOV. The RGBD camera gives us sampled depth measurements. In our 2D case, the sampled measurements are equally distributed within the FOV. Thus, the RGBD camera provides us with a sampled view of the environment in the FOV, as shown in Figure 2.1.

We build the polygon from these depth measurements by taking the position of the consecutive depth measurements as vertices, whereas we take the robot's position as first (and last) vertex. Hence, if there are n depth measurements, the polygon will have $n + 1$ vertices.

As the polygon represents the boundary between free space and unknown or occupied space as mentioned in Section 2.3, there are two kinds of polygon vertices. The vertices of the polygon are frontier vertices, if they were perceived at the sensor detection range, or obstacle vertices otherwise. In a more formal way, for $v_{\text{poly}} \in V_{\text{poly}}$, the set of frontier vertices F_v and the set of obstacle vertices O_v

$$\begin{aligned} &\|\mathbf{v} - \mathbf{r}\| = r_{\text{sensor}} \Rightarrow v_{\text{poly}} \in F_v, \\ &\|\mathbf{v} - \mathbf{r}\| < r_{\text{sensor}} \Rightarrow v_{\text{poly}} \in O_v, \\ &F_v \cup O_v = \emptyset \end{aligned} \tag{2.2}$$

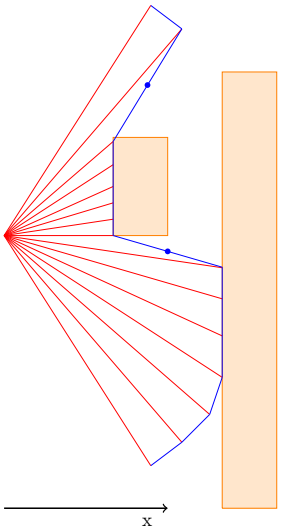


Figure 2.1: An example of a polygon in the local camera frame with occlusions and the inserted frontier vertices to account for these occlusions.

where V_{poly} is the set of polygon vertices, $\mathbf{v}$ is the position of the polygon vertex v_{poly} and corresponds to a depth measurement from $\mathbf{r}$, $\mathbf{r}$ is the position of the robot and r_{sensor} is the detection range of the depth sensor.

With the definition of the polygon vertices, the edges of the polygon can now be defined. An edge is either a frontier edge, if it connects two successive frontier vertices or a frontier vertex and an obstacle vertex. Otherwise, the edge is an obstacle edge. Again, in a more formal way, for $u_{\text{poly}}, v_{\text{poly}} \in V_{\text{poly}}$, the edge from u_{poly} to v_{poly}, $(u_{\text{poly}}, v_{\text{poly}})$, the set of frontier edges F_e and the set of obstacle edges O_e

$$\begin{aligned} (u_{\text{poly}}, v_{\text{poly}}) &\in F_e, \text{ if } u_{\text{poly}} \in F_v \cup v_{\text{poly}} \in F_v \\ (u_{\text{poly}}, v_{\text{poly}}) &\in O_e, \text{ otherwise} \\ F_e \cup O_e &= \emptyset. \end{aligned} \tag{2.3}$$

Before the polygon is build from the depth measurements, as explained before, some pre-processing is done.

As a first step, we check the depth measurements for occlusions. This is done by checking the distances between consecutive depth measurements

$$|p_{i,x} - p_{i+1,x}| > d_{\text{threshold}} \Rightarrow \exists \text{ occlusion}, \tag{2.4}$$

where $p_{i,x}$ is the x-coordinate of the ith depth measurement in the robot frame, $p_{i+1,x}$ is the x-coordinate of the $(i+1)$th depth measurement in the camera frame and $d_{\text{threshold}}$ defines the maximal distance between two consecutive depth measurements, to be considered as not separated by an occlusion. In case of an occlusion, an additional frontier polygon vertex is inserted between $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$, as visualized in Figure 2.1 with blue points.

In a second step, we subsample the depth measurements that are spatially close together to reduce the number of vertices saved for a polygon.

2.5 Polygon

A polygon is a directed ring graph $G_{\text{poly}} = (V_{\text{poly}}, E_{\text{poly}})$ of order n, where V_{poly} is a set with n elements called vertices (or nodes) and E_{poly} is a set of ordered pairs of vertices called edges ($E_{\text{poly}} \subseteq V_{\text{poly}} \times V_{\text{poly}}$). We call V_{poly} and E_{poly} the vertex set and edge set, respectively. For $u_{\text{poly}}, v_{\text{poly}} \in V_{\text{poly}}$, the ordered pair $(u_{\text{poly}}, v_{\text{poly}})$ denotes an edge from u_{poly} to v_{poly} [3].

As already explained in Section 2.3, the space inside a polygon is known free space, bounded by obstacles and frontiers. An example of such a polygon is shown in Figure 2.2.

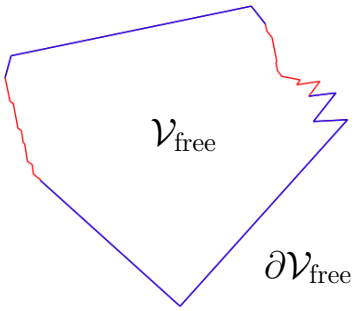


Figure 2.2: An example of a polygon with obstacle edges and frontier edges. The inside of the polygon represents known free space.

The data structure that handles the polygons is designed in the way, that an object of the Polygons class can contain multiple separate polygons. For this, an object of the polygons class has a list of polygon vertices and a list with the index of the start vertex of all the separate polygons, as shown in the UML diagram in Figure 2.4 and visualized in Figure 2.3. This way a Polygons object can perform operations on all the polygons it contains.

A polygon vertex is described by the following properties: the position of the vertex in the world frame, a boolean flag indicating if the vertex was observed at the maximum sensor range, indicating that it is a frontier vertex; the index of the next and the previous polygon vertex; the index of the corresponding pose graph vertex, from which the polygon vertex was observed; a boolean flag indicating, if it belongs to the current field of view and a boolean flag to indicate if the polygon vertex is accessible or not. With the information if vertices of the polygon belong to the current field of view, one can determine if the current field of view provides new information or not, and a planner can plan accordingly. The UML diagram of the Polygons and the PolygonVertex class is shown in Figure 2.4.

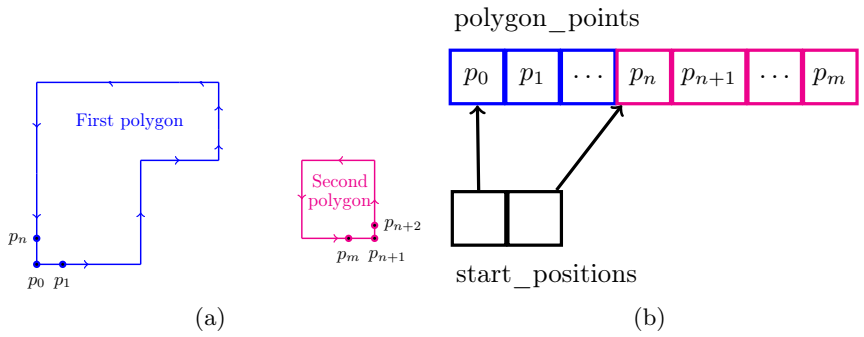


Figure 2.3: Visualization of the polygon data structure (For every polygon the index of the start vertex is saved in the list start_positions).

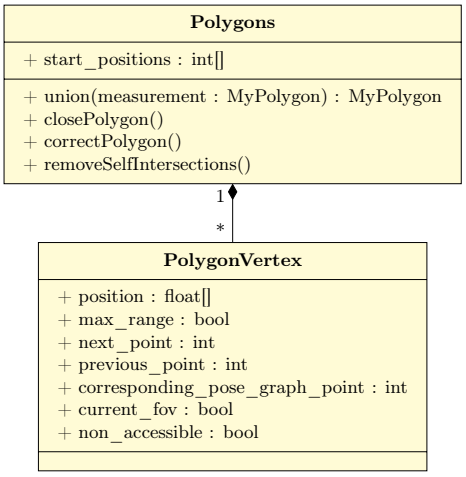


Figure 2.4: The UML diagram of the Polygons and PolygonVertex class.

2.6 Polygon union

In both proposed approaches, the union of two existing polygons is needed. The crucial point when building the union is, that the line segments of the two polygons are connected in a way that the outside and the inside of the polygons are consistent, as shown in Figure 2.6. To achieve this consistency, the cross product of the two line segments can be used to determine how the polygon points have to be connected to build the union.

The procedure to build the union starts by stacking the two Polygons objects together to form a new temporary Polygons object. The algorithm then searches for intersections, as explained in Section 2.6.1. In the next step, the polygon line segments are merged as described in Section 2.6.2. Afterwards, the polygons are separated and sorted according to the procedure in Section 2.6.3.

2.6.1 Searching intersections

To determine if there is an intersection and where, a 2D version of the algorithm "Intersection of two lines in three-spaces" presented in [7] was used.

Let each line be defined by two points $\mathbf{p}_{k,i}$ and $\mathbf{p}_{k,i+1}$ for $k = 1, 2$. Then the two lines can be expressed parametrically as $\mathbf{l}_1(t) = \mathbf{p}_{1,i} + t(\mathbf{p}_{1,i+1} - \mathbf{p}_{1,i})$ and $\mathbf{l}_2(s) = \mathbf{p}_{2,i} + s(\mathbf{p}_{2,i+1} - \mathbf{p}_{2,i})$ for $0 \leq t \leq 1$ and $0 \leq s \leq 1$. An intersection occurs if there exist a t and a s for which $\mathbf{l}_1(t) = \mathbf{l}_2(s)$.

To find intersections, the algorithm starts by going through all line segments of the first polygon $\mathbf{l}_{1,i}(t)$ for $i = 0, \ldots, m$, where m is the number of line segments the first polygon contains and looks for intersections with line segments of the second polygon $\mathbf{l}_{2,i}(t)$ for $i = 0, \ldots, n$, where n is the number of line segments the second polygon contains. The algorithm then differentiates the following cases:

1. $\nexists\, t$, $\nexists\, s$ for which $\mathbf{l}_{1,i}(t) = \mathbf{l}_{2,j}(s)\, \forall i, j$ (No intersection was found)
2. $\exists\, t$, $\exists\, s$, $\exists!\, i$, $\exists!\, j$ for which $\mathbf{l}_{1,i}(t) = \mathbf{l}_{2,j}(s)$ (One intersection was found)
3. $\exists\, t$, $\exists\, s$, $\exists!\, i$, $\exists!\, j_1$, $\exists!\, j_2$, $j_1 \neq j_2$ for which $\mathbf{l}_{1,i}(t) = \mathbf{l}_{2,j_{\{1,2\}}}(s)$ (Multiple intersections were found)

In case 1), the algorithm simply skips this line segment and proceeds with the next one. In case 2), an object with the required information to merge the line segments is created. The UML diagram of this class is shown in Figure 2.5. This object is added to a list to merge the line segments in a later step. The algorithm then proceeds with the next line. In case 3), the line segment is split up into multiple segments, such that there is only one intersection per line segment afterwards. After the additional points are added, the algorithm searches again for intersections and proceeds as in case 2).

2.6.2 Merging the line segments

This procedure adds the intersection point to the polygon vertices and connects the polygon vertices accordingly by setting the indices of the next and previous points of the corresponding polygon vertices.

To merge two line segments, the following information is needed:

- The first vertex of the polygon line segment of the first polygon ($\mathbf{p}_{1,1}$)
- The second vertex of the polygon line segment of the first polygon ($\mathbf{p}_{1,2}$)
- The first vertex of the polygon line segment of the second polygon ($\mathbf{p}_{2,1}$)
- The second vertex of the polygon line segment of the second polygon ($\mathbf{p}_{2,2}$)
- The intersection point ($\mathbf{i}$)
- The index of $\mathbf{p}_{1,1}$ in the polygon vertices list (self_point)

- The index of $\mathbf{p}_{2,1}$ in the polygon vertices list (other_point)

All this information was packed into an object of the MergeSegmentsInformation class, for every intersection found by the algorithm in the previous step. The UML diagram of the merge segments information class is shown in Figure 2.5.

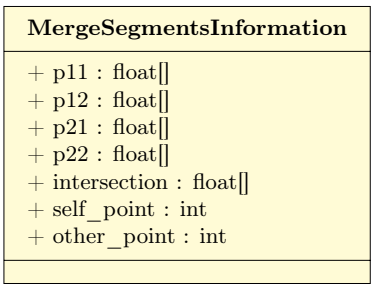


Figure 2.5: The UML diagram of the MergeSegmentsInformation class.

To merge the line segments, the algorithm begins by calculating the cross product c of the two line segments $\mathbf{l}_1$, $\mathbf{l}_2$ (one from the first polygon and one from the second polygon).

$$\begin{aligned} \mathbf{l}_1 &= \mathbf{p}_{1,2} - \mathbf{p}_{1,1} \\ \mathbf{l}_2 &= \mathbf{p}_{2,2} - \mathbf{p}_{2,1} \\ c &= \mathbf{l}_1 \times \mathbf{l}_2, \end{aligned} \tag{2.5}$$

where $\mathbf{p}_{1,1}$, $\mathbf{p}_{1,2}$ are the start respective the end point of the first line segment and $\mathbf{p}_{2,1}$, $\mathbf{p}_{2,2}$, the start respective the end point of the second line segment.

With the sign of the cross product, the point order can be determined.

$$\begin{aligned} &\mathbf{p}_{1,1} \rightarrow \mathbf{i} \rightarrow \mathbf{p}_{2,2}, \text{ if } c \geq 0 \\ &\mathbf{p}_{2,1} \rightarrow \mathbf{i} \rightarrow \mathbf{p}_{1,2}, \text{ if } c < 0, \end{aligned} \tag{2.6}$$

where $\mathbf{i}$ is the intersection point. Some examples of this line segment merge approach are shown in Figure 2.6.

2.6.3 Sorting polygons

After all the line segments are merged, some polygon vertices and edges do not longer belong to a valid polygon, as illustrated in Figure 2.7 with dotted edges.

As explained in Section 2.5, the data structure in which the polygons are saved has a list with all the polygon vertices and a list with the indices of the start vertex of every polygon it contains, visualized in Figure 2.3.

To sort the polygons, the separate polygons and the index of their starting vertex are determined and saved. Afterwards, a new list of polygon vertices is created, containing only the vertices which belong to a valid polygon and the indices of the start vertices are adjusted.

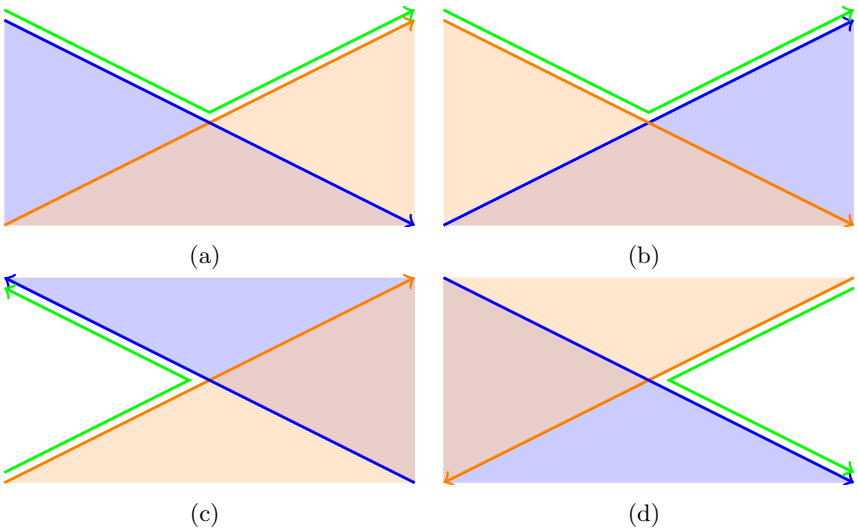


Figure 2.6: Some examples of the line segment merge approach (Line segment of the first polygon in blue, line segment of the second polygon in orange and the resulting line segment in green. The areas with background color represent the inside of the polygons whereas the area with white background represent the outside of the polygons).

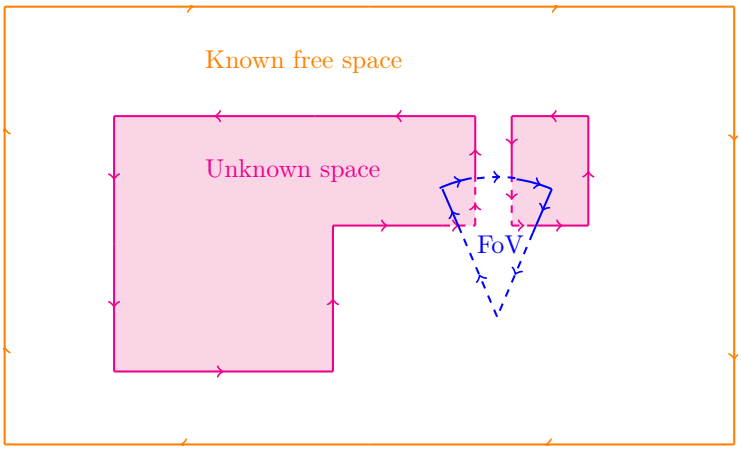


Figure 2.7: An example of separated polygons (Dotted edges are edges no longer belonging to a valid polygon).

To achieve this, for every detected intersection, the algorithm tries to traverse the polygon until it reaches the intersection vertex again or another already traversed vertex. The algorithm then adds the index of the intersection vertex to the list of the start vertex indices, if the polygon was fully traversed, or it continues with the next intersection, if the polygon was not fully traversed. This way, the index of the start vertex for every valid polygon is determined.

An example is shown in Figure 2.8. If we start to traverse the polygon from

the first intersection found (red circle), the first point we will revisit will be the first intersection point (where we started) and we will add this intersection point to the start vertices. If we afterwards start to traverse the polygon from the second intersection found (orange circle), we already start at a visited point and therefore do not add this intersection point to the start vertices.

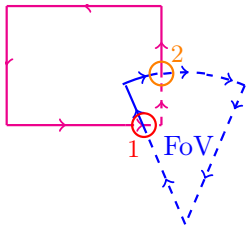


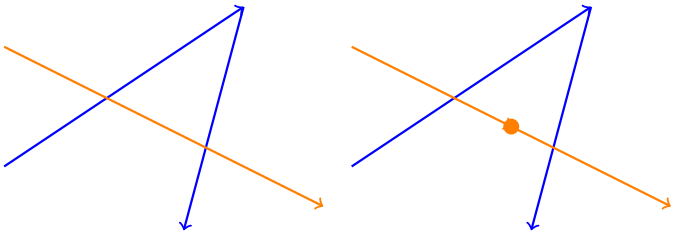
Figure 2.8: Traversing the polygon from the intersection points (The red circle is the first intersection found and the orange circle the second intersection found. The polygons are traversed in the direction of the arrows).

Now the index of every polygon start vertex is determined and saved, but the list with the polygon vertices may still contain some old polygon vertices, e.g. the dotted edges in Figure 2.8, which do not longer belong to any polygon. The next step is to remove these old vertices and to adjust the index of the polygon start vertices. The algorithm does this by traversing each polygon from the polygon start vertex and by building a new list of polygon vertices.

2.6.4 Special cases

Multiple intersections in a line segment of the second polygon

Multiple intersections in line segments of the first polygon are handled, as described in Section 2.6.1, by splitting up the intersecting line segment of the second polygon, as shown in Figure 2.9. Multiple intersections in line segments of the second polygon are handled in a similar way.



(a) Multiple intersections (b) Line segment split up

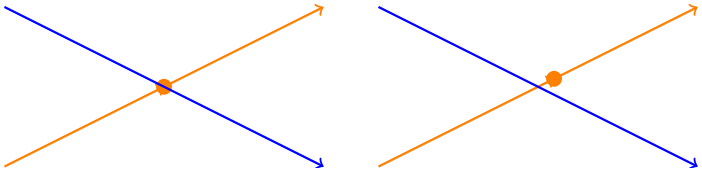
Figure 2.9: Handling of multiple intersection by splitting up the line segment into two line segments.

After all line segments of the first polygon are checked for intersections, we examine the stored MergeSegmentsInformation objects if two intersections have

the same start and end point in the second polygon. If this is the case, we add an additional polygon vertex in between the two intersections as shown in Figure 2.9b and adjust the start and end point of the MergeSegmentsInformation object.

Point on a line segment

It can happen that a polygon vertex is on the line segment of another polygon, as shown in Figure 2.10a. In this case, the algorithm finds two intersections: One where the polygon vertex on the line segment is the end point and one where it is the start point. As we need to detect only one intersection in this case, we move the polygon vertex slightly away from the line segment, as shown in Figure 2.10b. Although we modified a vertex of the polygon, the shape of the polygon does not change and the influence of this modification is negligible compared to the noise introduced by the state estimation.



(a) Polygon vertex on a line segment.

(b) Polygon vertex moved away from the line segment.

Figure 2.10: Handling of a polygon vertex on a line segment.

Parallel line segments

In case of parallel line segments, as shown in Figure 2.11, for one line segment two intersections will be found. In this case we also want to have only one intersection, similar to the case of a point on a line segment, described above. In this case we ignore the first intersection and only consider the second intersection.

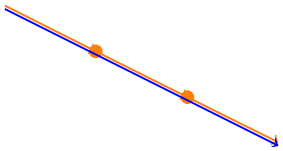


Figure 2.11: An example of parallel line segments.

2.7 First approach

In a first approach we tried to merge all field of view polygons into a global set of one outer and several inner polygons.

As explained in Section 2.2, the exploration is finished if $\bar{\mathcal{V}}_{\text{free}} = \mathcal{V}_{\text{free}}$, where $\bar{\mathcal{V}}_{\text{free}}$ is the so far explored space. The so far explored space is the union of all the FOVs measured so far, as stated in (2.1).

At every measurement, the latest polygons $\mathcal{P}_k(\mathtt{T}_{W,R_k})$ are built by taking the union of the polygons at the previous measurement $\mathcal{P}_{k-1}(\mathtt{T}_{W,R_{k-1}})$ and the current FOV

$$\mathcal{P}_k(\mathtt{T}_{W,R_k}) = \mathcal{P}_{k-1}(\mathtt{T}_{W,R_{k-1}}) \cup \mathcal{V}_{\text{fov}}(\mathtt{T}_{W,R_k}). \tag{2.7}$$

When building a map with polygons, there are two kinds of polygons: an outer polygon and inner polygons. The robot is located inside of the outer polygon and outside of the inner polygons. To distinguish these two types, we can use the fact that polygons have a direction. Polygons need to have a direction in order to distinguish between the inside and the outside of a polygon. The polygon built from the depth measurements is clock wise oriented and therefore the outer polygon as well, as the known space is inside. Inner polygons, on the other hand, are counter-clock wise oriented as the known space is located outside of them. An example is shown in Figure 2.12.

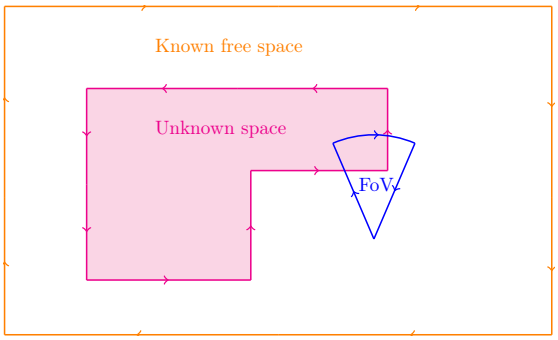
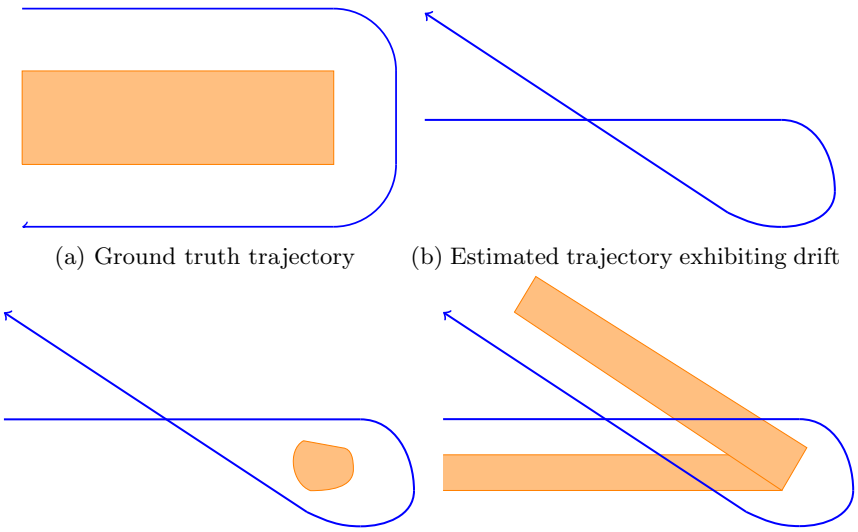


Figure 2.12: An example of an outer polygon in which there is known free space, an inner polygon with unknown space inside and the polygon of the FOV.

2.7.1 Active area

Visual(-inertial) odometry systems produce a pose graph exhibiting drift, as shown in Figure 2.13, and therefore polygon merges can only locally guarantee to be correct, where the drift is negligible.



(a) Ground truth trajectory

(b) Estimated trajectory exhibiting drift

(c) With a metric, grid based representation

(d) Desired representation

Figure 2.13: (a) An example of a trajectory with an obstacle in the middle. (b) The estimated trajectory e.g. the output of a visual(-inertial) odometry system. (c) A failure case with a grid based map representation. (d) A correct map representation relative to the trajectory.

To account for this, the concept of the active area is introduced. The active area is the local area within the drift is assumed to be negligible. Similar concepts were already used in [20] and [19]. In contrast to those approaches, we make use of the neighboring pose graph vertices in the pose graph to define the active area. We define the pose graph vertices of the active area as all the pose graph vertices maximum n edges away from the current pose graph vertex.

$$V_{\text{pg,active}} = \{v_{\text{pg},i} \mid \text{dist}(v_{\text{pg,current}}, v_{\text{pg},i}) \leq n\} \quad \forall v_{\text{pg},i} \in V_{\text{pg}}, \tag{2.8}$$

where $v_{\text{pg,current}}$ is the pose graph vertex, where the robot is currently located and $\text{dist}(v_{\text{pg},1}, v_{\text{pg},2})$ returns the minimal number of edges between $v_{\text{pg},1}$ and $v_{\text{pg},2}$. We set n empirically to a value which results in an active area which is neither too small nor too big. A detailed explanation will follow in Section 2.7.5. The polygon vertices of the active area are then all the polygon vertices that belong to a pose graph vertex of the active area

$$\begin{aligned} V_{\text{poly,active}} &= \bigcup_i V_{\text{poly,active},i} \\ v_{\text{pg,active},i} &\ni V_{\text{poly,active},i} \\ v_{\text{pg,active},i} &\in V_{\text{pg,active}}. \end{aligned} \tag{2.9}$$

Loop closures, introduced in Section 2.7.3, are implicitly handled as they result in an additional edge in the pose graph and thus connect two points in the pose graph.

To prevent the union of unrelated polygons which do overlap in space, due to drift in the trajectory, the union of two polygons is only performed if the intersections of the two polygons are in the active area. As a result, only polygons that are in a space where the drift is negligible are merged with the union operation. Polygons that appear to intersect, but do not belong to the same active area, are kept separate.

This leads to the following adaptations to determine if there is an intersection and where, introduced in Section 2.6.1. With the concept of the active area, the algorithm considers only line segments of the first and second polygon that belong to the active area. Thus, the line segments of the first polygon to check for intersections are $\mathbf{l}_{1,i}(t)$ if $v_{\text{poly},1,i} \in V_{\text{poly},active} \cap v_{\text{poly},1,i+1} \in V_{\text{poly},active}$, where $v_{\text{poly},1,i}$ and $v_{\text{poly},1,i+1}$ are the polygon vertex of the first respective second point of the line segment. The line segments of the second polygon to check for intersections are $\mathbf{l}_{2,i}(t)$ if $v_{\text{poly},2,i} \in V_{\text{poly},active} \cap v_{\text{poly},2,i+1} \in V_{\text{poly},active}$, where $v_{\text{poly},2,i}$ and $v_{\text{poly},2,i+1}$ are the polygon vertex of the first respective second point of the line segment.

2.7.2 Entry / exit intersections

As polygons are always closed, if a polygon line segment enters another polygon, there must also be a polygon line segment which exits the other polygon. Consequently, intersections always have to appear in pairs (entry and exit intersection). In some cases, only one intersection of a pair is found, due to the fact that only polygon line segments in the active area are considered while searching for intersections, as shown in Figure 2.14.

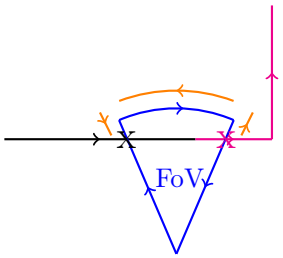


Figure 2.14: An example where an entry intersection is not in the active area (Edges in magenta belong to the active area, whereas black edges do not belong to the active area. The × represent intersections. As the left, black intersection does not belong to the active area, it is not detected during the search for intersections. The arrows on the polygon edges indicate the direction of the polygon, the orange arrows next to the polygon edges represent the direction in which the algorithm searches for the remaining entry intersection in this example).

To check if the associated intersection of an already found intersection is also already found or not, and has to be found, the algorithm proceeds as follows.

The algorithm first checks if the found intersection is an entry or exit intersection. This can again be determined with the sign of the cross product of the

two line segments $\mathbf{l}_1$ and $\mathbf{l}_2$ (one from the first polygon and one from the second polygon)

$$\begin{aligned} \mathbf{l}_1 &= \mathbf{p}_{1,2} - \mathbf{p}_{1,1} \\ \mathbf{l}_2 &= \mathbf{p}_{2,2} - \mathbf{p}_{2,1} \\ c &= \mathbf{l}_1 \times \mathbf{l}_2, \end{aligned} \tag{2.10}$$

where $\mathbf{p}_{1,1}$, $\mathbf{p}_{1,2}$ are the start respective the end point of the first line segment and $\mathbf{p}_{2,1}$, $\mathbf{p}_{2,2}$, the start respective the end point of the second line segment.

Then, with the sign of the cross product, it can be determined whether the intersection belongs to an entry or exit intersection.

$$\begin{aligned} &\text{entry intersection} \quad , \text{if } c \geq 0 \\ &\text{exit intersection} \quad , \text{if } c < 0 \end{aligned} \tag{2.11}$$

If the found intersection is an entry intersection, the algorithm follows the line segments after $\mathbf{l}_1$ and for every subsequent line segment $\mathbf{l}_{\text{sub}} = \mathbf{p}_{\text{sub},2} - \mathbf{p}_{\text{sub},1}$ it checks if an already found intersection exists where the start point of the first line segment $\mathbf{p}_{1,1}$ is equal to the start point of the current subsequent line segment $\mathbf{p}_{\text{sub},1}$ ($\mathbf{p}_{1,1} = \mathbf{p}_{\text{sub},1}$). If this is not the case, the algorithm checks if there is an intersection between the current subsequent line segment and the second polygon. The algorithm repeats these steps until an exit intersection is found.

Similarly, if the found intersection is an exit intersection, the algorithm follows the line segments before $\mathbf{l}_1$ and does the same checks described before for every preceding line segment until an entry intersection is found.

2.7.3 Loop closure

As discussed before, the union of two polygons is only built if both polygons belong to the active area.

To prevent mapping of already observed area, e.g. as it would happen in the example shown in Figure 2.15, SLAM systems perform a loop closure when a robot revisits a place and correct the pose graph accordingly.

After a loop closure, the trajectory is locally drift-free and the polygons can be corrected, given the new positions of the pose graph vertices which the SLAM system provides.

To correct the polygon after a loop closure, the position of the polygon vertices are corrected first. Then, some maintenance is needed to deal with inconsistent self-intersections. Afterwards, the union of the polygons is built with the extended active area.

Correction of the polygon vertices position

To correct the position of every vertex of the polygons, the algorithm iterates through all the polygon vertices and performs the correction for every vertex separately.

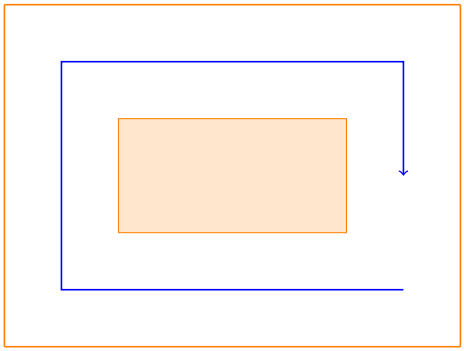


Figure 2.15: An example of an endless loop where a robot would map already observed space if there is no loop closures.

To calculate a corrected polygon vertex position, the algorithm uses the estimated position and orientation of the anchor point (the position of the corresponding pose graph vertex) and the true position and orientation of the same anchor point. It then calculates the corrected position according to

$$\begin{aligned} {}_W\mathbf{p}_{\text{p,corr}} = & {}_W\mathbf{p}_{\text{anchor, eal}} \\ & + {}_W\mathbf{R}_{WR,\text{eal}} {}_W\mathbf{R}^T_{WR,\text{ebl}} ({}_W\mathbf{p}_{p,\text{ebl}} - {}_W\mathbf{p}_{\text{anchor, ebl}}), \end{aligned} \tag{2.12}$$

where ${}_W\mathbf{p}_{\text{p,corr}}$ is the position of the corrected polygon vertex, ${}_W\mathbf{p}_{\text{anchor, eal}}$ is the estimated position of the anchor point after the loop closure, ${}_W\mathbf{R}_{WR,\text{eal}}$ is the estimated rotation from the robot frame to the world frame after the loop closure, ${}_W\mathbf{R}^T_{WR,\text{ebl}}$ is the estimated rotation from the world frame to the robot frame before the loop closure, ${}_W\mathbf{p}_{p,\text{ebl}}$ is the estimated position of the polygon vertex before the loop closure and ${}_W\mathbf{p}_{\text{anchor, ebl}}$ is the estimated position of the anchor point before the loop closure.

Polygon merging

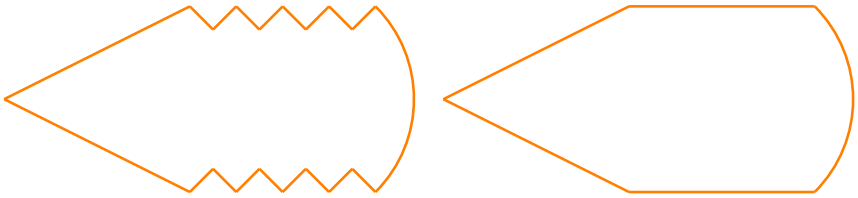
In polygon merging, overlapping parts of polygons are merged. The procedure to merge a polygon is similar to the process of building the union of two polygons, which was described in Section 2.6. The difference is, that instead of the second polygon only the first polygon is given but the two overlapping parts are treated like two polygons.

2.7.4 Some improvements

Neglecting the intersection points

While performing some experiments we observed, that adding the intersection points as mentioned in Section 2.6.2 leads to spiky polygons, as shown in Figure 2.16a. This can lead to a lot of self intersections after the polygon is corrected, as explained in Section 2.7.3. A way to get smoother polygons is by

neglecting the intersection point while merging the line segments, as explained in Section 2.6.2, shown in Figure 2.16b. In the current implementation, inserting the intersection point can be either deactivated or activated by a flag.



(a) (b)

Figure 2.16: Polygons with and without the intersection point inserted.

2.7.5 Problems of the first approach, partial solutions and failure cases

Self intersections

Self intersections can arise in multiple ways: if two parts of the same polygon, due to drift, overlap in space; if a polygon gets twisted with the correction of the polygons; or if already mapped space is perceived again because no loop closure was triggered.

While unwanted polygons that arise from overlapping polygon parts are prevented from merging, thanks to the concept of the active area, we need to find a way to resolve the other two cases.

Self intersections after polygon correction - twisted polygons

Self intersections can arise after the correction of the position of the polygon vertices when a polygon gets twisted, as shown in Figure 2.17. In twisted polygons, a self intersection leads to a change of the orientation in one part of the polygon. This can lead to a failure when a polygon union is built with the polygon of the FOV.

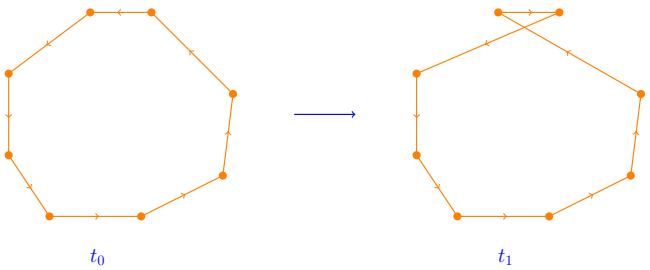
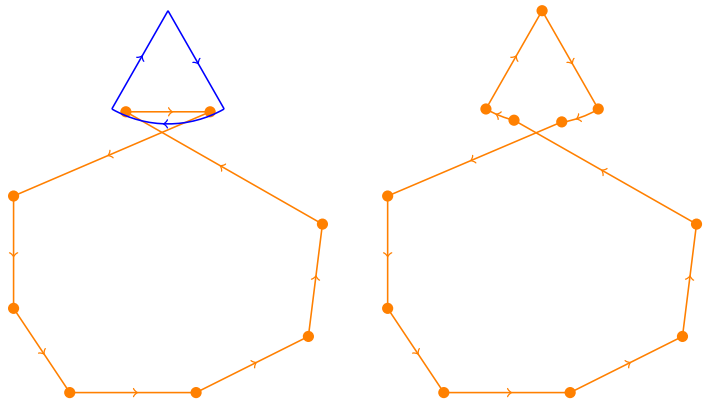


Figure 2.17: An example of a polygon that gets twisted after the correction of the polygon vertices.

A normal inner polygon is counter-clock wise oriented and the inside of the polygon represents unknown space (indicated by the arrows pointing to the unknown space in Figure 2.19a). An upended inner polygon, on the other hand, is clock wise oriented and the unknown space would be outside and the known space inside (indicated by the arrows in Figure 2.19c), which is wrong as the outside of the upended polygon was already explored and only the outer polygon should be clock wise oriented, as explained in Section 2.7. To check if a polygon is the outer polygon or an inner polygon, one can check if the robot is located within the polygon or not. When the union between an upended part of a polygon or an upended polygon and the polygon of the FOV is built, the union is wrong as it extends the polygon rather than decreasing it, as shown in Figure 2.18.



(a) (b)

Figure 2.18: An example of a union of the polygon of the FOV with the upended part of a polygon.

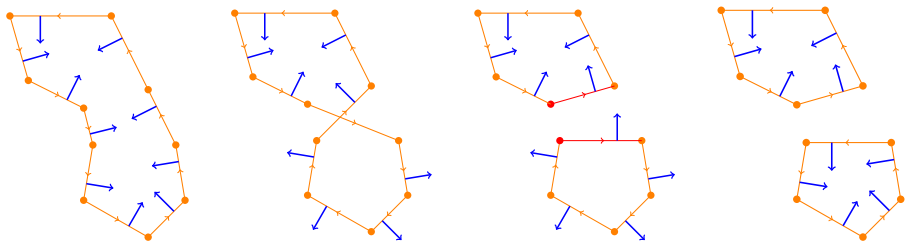
Resolving twisted polygons

We check for twisted polygons and resolve them right after they arise, which is after the correction of the position of the polygon vertices.

Resolving inner, twisted polygons is done in three steps: First, the polygons are searched for self intersections similar as explained in Section 2.6.1. Second, the polygons with a self intersection are split into two polygons. This is done by reconnecting the polygon points of the intersecting polygon line segments (red edges in Figure 2.19c). This results in two polygons, one counter-clock wise oriented and one upended, clock wise oriented polygon. In the third step, to correct the orientation of inner, upended polygons, their orientation gets reversed, as shown in Figure 2.19d.

Instead of reversing upended polygons it would also be possible to remove them to prevent polygon union failures. If the upended part of a twisted polygon represents already explored space, removing the upended part would be the right thing to do. If, on the other hand, the upended polygon part represents unexplored space, removing it would be wrong. As we can not distinguish this

two cases, we do not remove the upended polygon but reverse its orientation, which results in a normal inner polygon.



(a) Normal polygon (b) Self intersected polygon (c) Separated polygons (d) Separated, polygons with correct orientation

Figure 2.19: From a normal polygon to a polygon with a self intersection, how to separate the polygons and how to reverse the wrong oriented polygon.

Self intersection in double perceived space

In a scenario as the one shown in Figure 2.20a and Figure 2.20b, the robot perceives free space it already has perceived at a previous point of time. As the SLAM system has not yet triggered a loop closure, the active area contains only one side of the polygon, the FOV polygon intersects with (Figure 2.20a). Therefore, the polygon union is built as if the robot would perceive free unknown space and that leads to a self intersection (Figure 2.20b). A loop closure before, as shown in Figure 2.20c and Figure 2.20d, or after the polygon union, would resolve this situation. If, however, the SLAM system does not trigger a loop closure, we end up with an inner polygon part that is clock wise oriented. This can lead to a failure in a subsequent polygon union if we do not resolve this situation similar to the twisted polygons.

Resolving self intersections in double perceived space

To resolve self intersections in double perceived space, we proceed similar as we do to resolve twisted polygons. Before we perform a polygon union, we search in the active area of the intersected polygon for self intersections and then resolve them as described above. We restrict the search for self intersections to the active area, as only there the drift is negligible and we can guarantee consistent polygons when they are modified. While this fix works for many cases, the dependence on the size of the active area leads to a limitation as described next.

Failure cases

There are situations where the size of the active area is a critical factor and decides, if the resulting polygon is correct or not. We will describe two such scenarios, one in which the size of the active area should be as big as possible

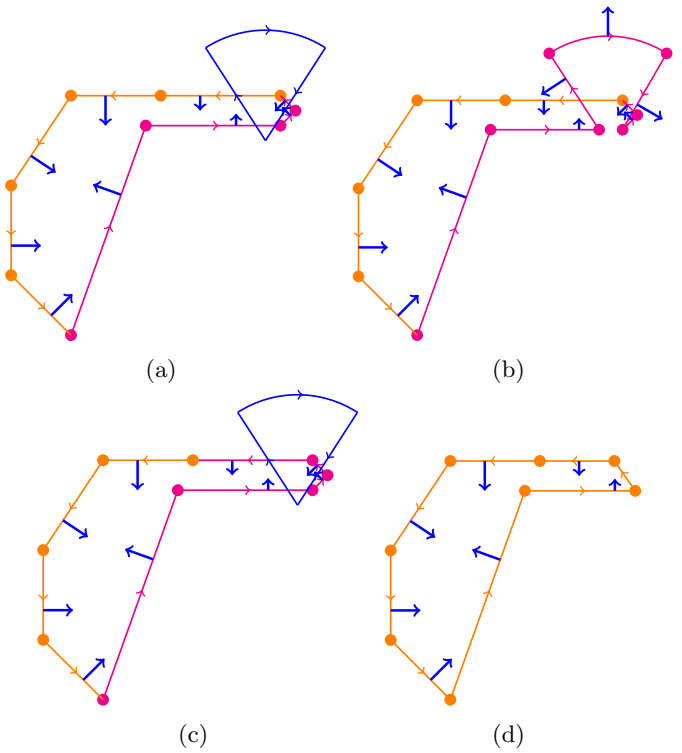


Figure 2.20: ((a) and (b)) An example of a polygon self intersection due to the absence of a loop closure and a too small active area. ((c) and (d)) An example where the size of the active area is fine (due to a loop closure) (The polygon edges of the active area are shown in magenta).

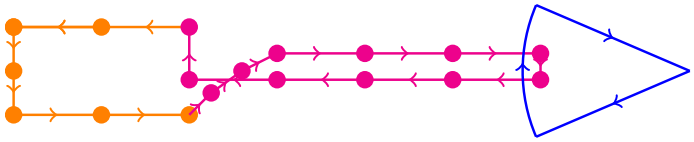
to achieve a correct polygon and one, in which the opposite is the case. These contradicting corner cases make the active area approach unsuitable for many use cases.

In scenarios, where there is a polygon self intersection, e.g. as illustrated in Figure 2.21a and Figure 2.21c, the size of the active area is critical. As explained in Section 2.7.5, if a polygon intersects with the polygon of the FOV, it will be searched for self intersections within the active area. If such a self intersection exists, the polygon will be split into two polygons and the upended polygon will be reversed, as shown in Figure 2.21b. Now the union of the polygon from the FOV and the existing polygon can be built without a problem.

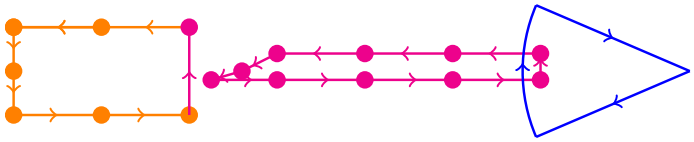
However, if the self intersection is not within the active area, as shown in Figure 2.21c, the self intersection will not be removed and the polygon of the FOV intersects with an upended part of the polygon. As discussed earlier, the union with a wrong oriented inner polygon leads to a wrong polygon and should be prevented.

In the above mentioned case (Figure 2.21c), the solution would be to increase the size of the active area, that it also contains the self intersection.

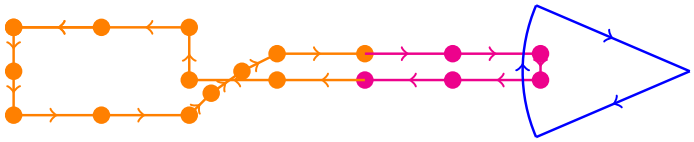
Another situation, where the size of the active area is critical, is illustrated in



(a) Self intersection within the active area.



(b) Self intersection removed and upended polygon reversed.



(c) Self intersection out of the active area.

Figure 2.21: Self intersection within and out of the active area in magenta.

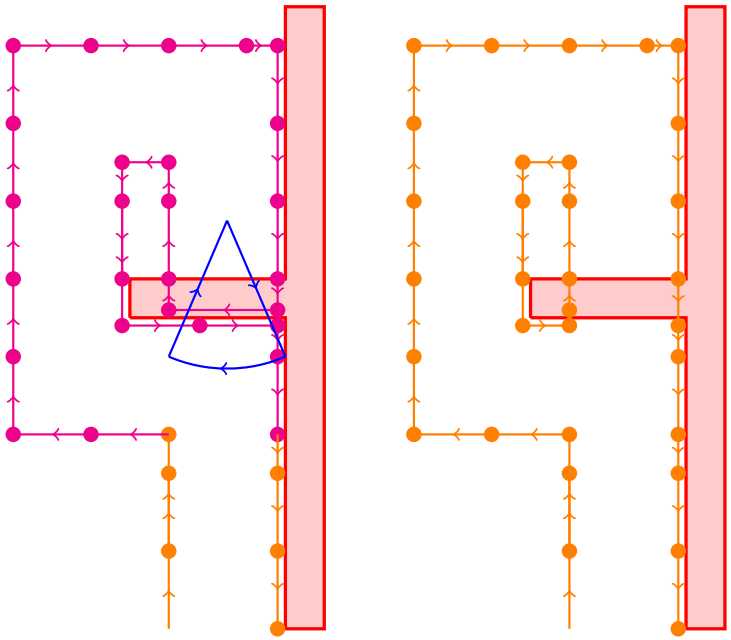
Figure 2.22. In this scenario, there is a small wall for which the polygon is located at a lower position as the wall is in reality, due to drift in the trajectory. The polygon of the FOV than overlaps with the part of the polygon on the other side of the wall. In case of a big active area, the wall will simply be removed as the polygon parts on both sides of the wall belong to the active area, illustrated in Figure 2.22a and Figure 2.22b. On the other hand, if the active area is small enough, that only the polygon part on one side of the wall belongs to the active area, the part of the polygon that represents the wall would not be removed.

These two scenarios demonstrate the problem of the size of the active area. For certain situations it will be too small, whereas for other situations it will be too big.

2.7.6 Achieving coverage

The discussed failure cases in Section 2.7.5 could lead to an unsuccessful exploration where there is still unexplored space left. The reason that the proof in Section 1.1.2 does not hold in such scenarios is the following: If a polygon gets removed due to a failure while building a polygon union, it can happen that frontiers are removed without further exploration.

Besides these failure cases, the robot knows where there are frontiers if there are any. If the used SLAM system allows the robot to move along the pose graph to a certain pose, the robot can reach the frontiers and the proof in Section 1.1.2 holds for this first approach.



(a) The polygon parts on both sides of the wall belong to the active area.

(b) The wall no longer exists after the polygon union.

Figure 2.22: An example of a too big active area.

2.8 Second approach

As the first approach introduced in Section 2.7 has some limitations, discussed in Section 2.7.5, we developed a second approach that does not have these problems.

The drawbacks of the active area approach come from the fact that we build a global map. In the new approach we do not build a global map but keep the polygon of the FOV of every pose separate.

2.8.1 Building the local polygons

As we not longer have a global polygon, and work with local polygons only, we have to introduce a new type of polygon vertices and edges, one that represents free space.

We proceed as follows to determine the type of the polygon vertices for the polygon of the current measurement. We start by taking a copy of the current polygon. Then we check if there are any intersections between the current polygon and the polygons of the directly connected pose graph points (Figure 2.23a and Figure 2.23f). If there are intersections, we resolve overlaps of these two polygons and proceed with the polygons of the next neighboring pose graph points (Figure 2.23b and Figure 2.23g) until there are no intersections

found anymore (Figure 2.23i). This process is illustrated in Figure 2.23 and Figure 2.24.

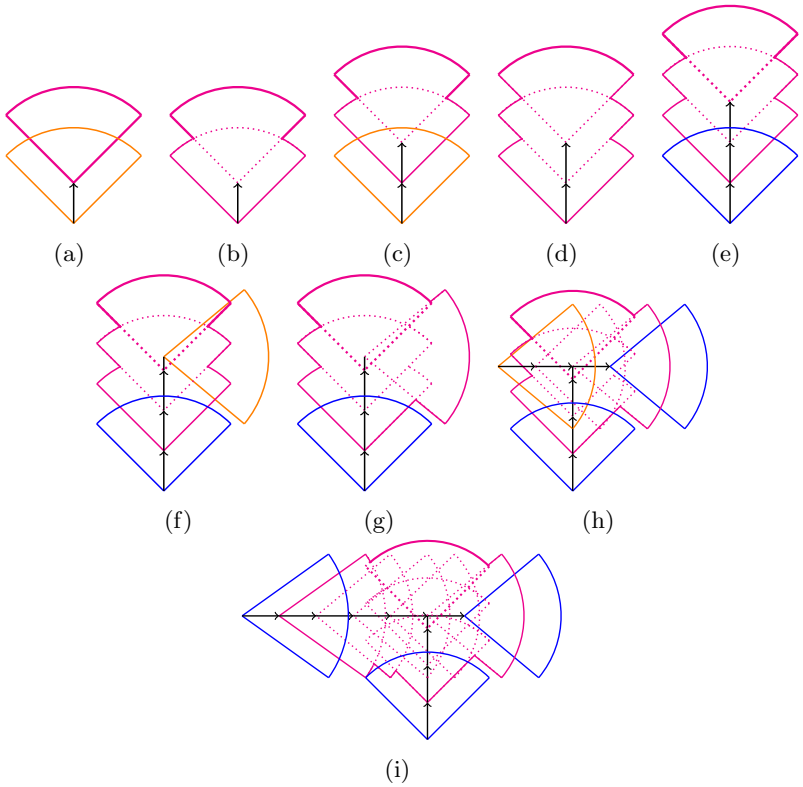


Figure 2.23: Steps to build the extended polygon: First five steps ((a) - (e)), following the pose graph in one direction. Following the pose graph in a second and third direction ((f) - (h)). Last step, no intersecting polygon left (i).

As every polygon is represented in the local frame of the pose at which the measurement was taken, we have to transform the polygons of the neighboring pose graph points into the frame of the current pose before we build the union.

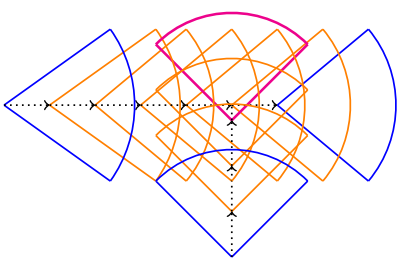
In a more formal way,

$$ {}_{R_0}\mathcal{P}_{\text{grown}} = \bigcup_i f\left(\mathcal{P}(\mathtt{T}_{W,R_i})\right) \mathtt{T}_{R_0,R_i} \mathcal{P}(\mathtt{T}_{W,R_i}), \tag{2.13} $$

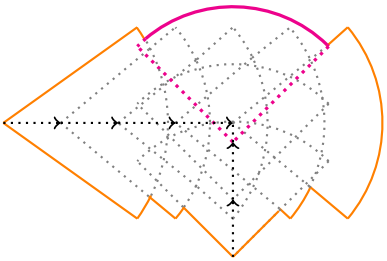
with

$$ f\left(\mathcal{P}(\mathtt{T}_{W,R_i})\right) = \begin{cases} 1, & \text{if } \mathcal{P}(\mathtt{T}_{W,R_j}) \cup \mathcal{P}(\mathtt{T}_{W,R_0}) \neq \emptyset, \text{ for } j = 0, \ldots, i \\ 0, & \text{otherwise} \end{cases} \tag{2.14} $$

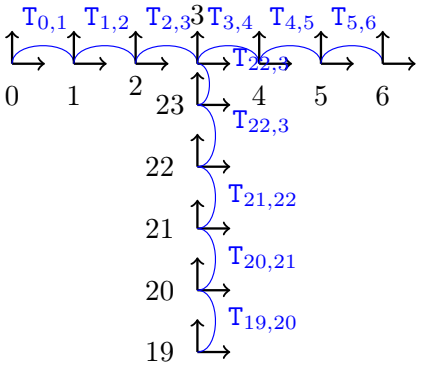
where $\mathcal{P}(\mathtt{T}_{W,R_i})$ is the local polygon at pose $\mathtt{T}_{W,R_i}$, $\mathcal{P}(\mathtt{T}_{W,R_0})$ is the local polygon of the current pose and $\mathtt{T}_{R_0,R_i}$ is the transformation form the frame of the ith pose into the frame of the current pose.



(a) Current polygon, neighboring polygons, intersecting with the current polygon and neighboring polygons, not intersecting with the polygon of the current FOV.



(b) Extended polygon and parts of the polygon of the current FOV (Dotted edges represent free space).



(c) Pose graph with the transformations between the pose graph points used to transform the local polygons into the frame of the current pose.

Figure 2.24: Building the extended polygon.

After we have built the union of all the intersecting polygons, we determine which of the polygon vertices of the local polygons are not contained in the extended polygon. We set the type for these polygon vertices to free space.

Again, in a more formal way,

$$\begin{aligned} v_{\text{poly},i} \in V_{\text{poly,local}} \cap v_{\text{poly},i} \notin V_{\text{poly,grown}} \Rightarrow v_{\text{poly},i} \in FR_{\text{poly}} \\ O_{\text{poly}} \cap F_{\text{poly}} \cap FR_{\text{poly}} = \emptyset \end{aligned} \tag{2.15}$$

where $V_{\text{poly,local}}$ is the set of polygon vertices of a local polygon, $V_{\text{poly,grown}}$ is the set of polygon vertices of the grown polygon and FR_{poly} is the set of polygon vertices, representing free space.

2.8.2 Loop closures

As we do not build a global map in the second approach, correcting the pose graph after a loop closure is not required and has not to be done. Without the

correction of the pose graph, it will keep the drift over the trajectories and we must deal with the virtual displacement occurring at loop closures in another way. For this, we introduce the so called loop closure edge in the pose graph that connects the virtual displaced poses and represents the relative transformation between them. An example of such a loop closure edge is shown in Figure 2.25.

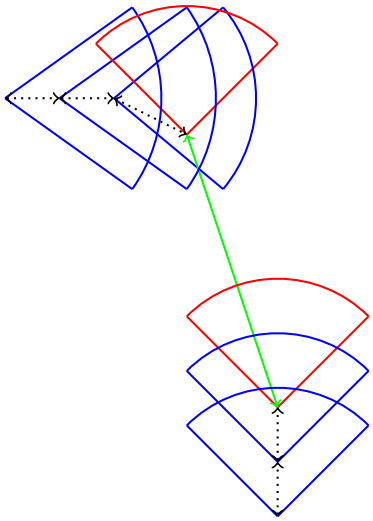


Figure 2.25: An example of a loop closure: The two red FOVs are the virtually displaced pose graph points (representing the same physical location). The black dotted pose graph edges are normal pose graph edges and the green pose graph edge is a loop closure edge.

2.8.3 Achieving coverage

For a frontier based exploration we need to know where there are frontiers and how we can reach them. With a local polygon for every pose in the pose graph, the robot knows where the frontiers are located if there are any present. If the used SLAM system allows the robot to move along the pose graph to certain pose, the robot is also able to reach the frontiers.

Under this condition, the proof in Section 1.1.2 is therefore applicable for this approach.

2.9 Rapid exploration

To perform exploration with the proposed representations, we adapted the rapid exploration approach introduced in [4].

The rapid exploration approach proposed in [4] is designed specifically for multi-rotor exploration at high speeds. The reactive behavior of the algorithm allows for fast incorporation of new information and results in efficient trajectories. Compared to classic frontier-based exploration, the approach can occasionally

exhibit a small increase in the total path length, but at the same time achieves smaller exploration times for the same maximum velocity constraint. To achieve the reactive behavior, the multi-rotor rapidly selects a goal frontier from its field of view. The goal frontier is selected in a way that minimizes the change in velocity. If there are no frontiers in the field of view, the algorithm switches to a classical frontier selection method.

To have the approach working with the proposed representation, the following adaptations were required: The frontier selection, discussed in Section 2.9.1, the accessibility check, explained in Section 2.9.2 and the classical frontier selection method, as detailed in Section 2.9.3.

2.9.1 Frontier selection

Instead of choosing the frontier that leads to a minimal amount of velocity deviation, we choose the frontier that leads to the minimal amount of orientation change. The resulting behavior is the same, but evaluating the orientation changes is simpler within our implementation.

2.9.2 Accessibility check of frontiers

In the original approach, presented in [4], OctoMap [11], an occupancy grid based representation was used. OctoMap builds a global map and provides the functionality to check a path for intersections with occupied cells. In contrast, our representation has the concept of the active area, introduced in Section 2.7.1, and only the position of obstacles in this active area is supposed to be known exactly or even only the current FOV in the second approach. Therefore, during exploration, only the obstacle points in the active area, respective the current FOV can be checked for intersections with the path. Additional to this, the current FOV is also checked not to head in the direction of an obstacle.

2.9.3 The classical frontier selection method

The classical frontier selection method in the original rapid exploration approach [4] searched the shortest path to any frontiers with the Dijkstra algorithm in the OctoMap [11]. As we do not assume to have access to a global map, we search for the closest frontier with a breadth-first search along the pose graph and then take the pose graph vertices which connect the robot position with the frontier position as waypoints. This way, we track back the pose graph or across loop closures until we reach the closest pose graph vertex that has a frontier vertex.

If a pose graph vertex is too close to obstacles, e.g. a wall, it can happen that the accessible check for a frontier vertex fails and thus accessing the frontier vertex from its corresponding pose graph vertex is not possible. For this case, the concept of the “normal point” was introduced. The “normal point” lies on the normal of the frontier line, in a certain distance away from the unexplored space. If a frontier vertex is not accessible via its corresponding pose graph vertex, the planner will instead try to access the frontier vertex via this “normal point”.

Chapter 3

Experiments

As there is no other exploration approach that is based on a representation, capable of handling noisy state estimates, we can only present some results of our approach and compare the exploration time to an exploration with a grid based representation.

3.1 Experimental setup

To perform our experiments, we implemented a simulated SLAM system. It is implemented in the Python programming language and uses the libraries *numpy* and *matplotlib*. The simulation loops through the three parts: mapping, path planning and plotting until the exploration is finished. One iteration takes in average $0.3s$. The noise of the state estimates is simulated with the help of a gaussian random number generator provided by the *numpy* library.

The simulated SLAM system has basic loop closure and localization functionality. Loop closures are triggered when the distance from the robot to a previous visited place is equal or less than the camera range. A loop closure is only triggered if the previous point is accessible form the current robot position to prevent loop closures over an obstacle. Compared to a real SLAM system with a place recognition module, our loop closure simulation is quite limited. However, for a minimal simulation environment to evaluate our map representation approach, it fulfills its purpose.

To simulate localization when the robot is moving through a previous visited area, we calculate the distances to the pose graph points and if the robot is enough close to a pose graph point (below a certain threshold) we set the robot's current position to the one of the pose graph point. Again, this simulated localization is quite limited compared to a real SLAM system, but suites the purpose of evaluating our map representation approach.

As mentioned, the implemented SLAM simulation environment used for the experiments has its limitations, e.g. no real place recognition. These limitations can lead to synthetic problems while traversing back the pose graph. As the

main focus of this thesis is the representation approach, we implemented a *teleport* function for the robot. With this, the robot can directly *teleport* to a pose graph point and does not have to move all the way back. To account for the time the robot would need to reach a pose graph point normally, we calculate the path between the pose graph points and determine the amount of time needed by assuming the robot could move at its maximal speed. As the robot moves in known area and should be able to localize itself while traversing the pose graph, the assumption to move at maximal speed is reasonable.

3.2 Evaluation method

We want to evaluate if it is possible to reach full coverage performing frontier-based exploration with our proposed map representation approach. For this, we overlay the map with a cell grid and perform exploration on this discretized map in parallel. During the exploration we mark grid cells as explored if a depth measurement ray (a ray from the robot position to the depth measurement) intersects with the grid cell, as shown in Figure 3.1.

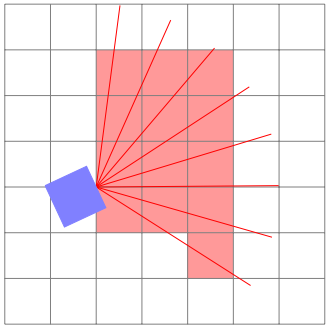
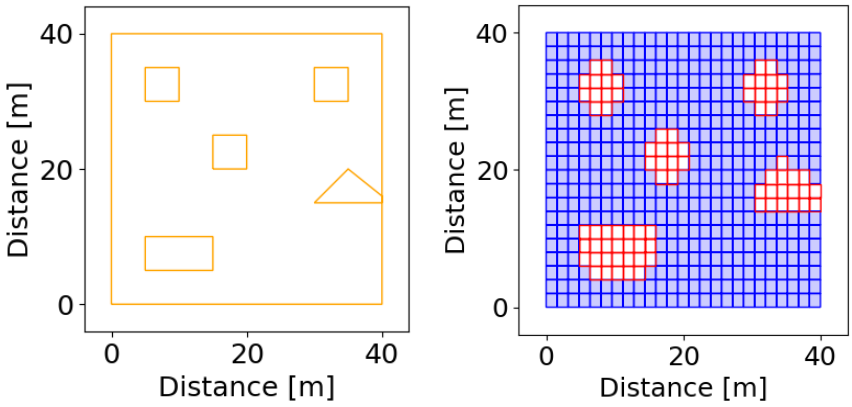


Figure 3.1: Grid cells in red are explored grid cells. Grid cells in white are unexplored grid cells. The lines in red are the rays from the robot to the depth measurements. The blue rectangle represents the robot.

For the ground truth, we evaluate the number of occupied grid cells, where we consider a grid cell as occupied if an edge of an obstacle intersects with the grid cell or if the cell is within an obstacle. An example of a map and the corresponding ground truth is shown in Figure 3.2. We then determine the number of free grid cells as $n_{\text{free}} = n_{\text{total}} - n_{\text{occ}}$, where n_{total} is the total number of grid cells, which we know from the creation of the grid and n_{occ} is the number of occupied grid cells.

After the exploration we compare the number of explored cells with the ground truth. In this way we can evaluate if the exploration achieved full coverage.



(a) Map with obstacles.

(b) Map with occupied cells in red and free cells in blue.

Figure 3.2: An example of a map and the corresponding ground truth (occupied and free cells).

3.3 Exploration time comparison

We want to compare the exploration time of a frontier-based exploration using our map representation to a baseline. As there are no other map representations fulfilling our key properties, we decided to compare with a grid-based metric map representation, as it is an often used representation.

This comparison is not entirely fair because grid-based map representations are discrete representations and our proposed representation is a continuous representation. We also use the ground truth poses for the state estimates in the explorations with the grid-based representation, whereas the exploration with our proposed representation has only the noisy state estimates described before. Nevertheless, this comparison should give us a general idea about the exploration with our proposed representation approach.

For the comparison, we record the time until all free grid cells are explored as well as the time until there are no more frontiers left. We than take this two times for the comparison.

3.4 Experiment output

During the experiments we visualize the following elements: A coverage plot that visualizes the unexplored, blue grid cells, and the explored, orange grid cells as well as the frontier and obstacle polygon vertices. A plot showing the estimated and true trajectories of the robot and a plot with the pose graph that also visualizes loop closure edges. An example of this visualization is shown in Figure 3.3.

Next to the above described visualization during the experiment, we also save the time stamps and the ratio of explored space for every time stamp. With

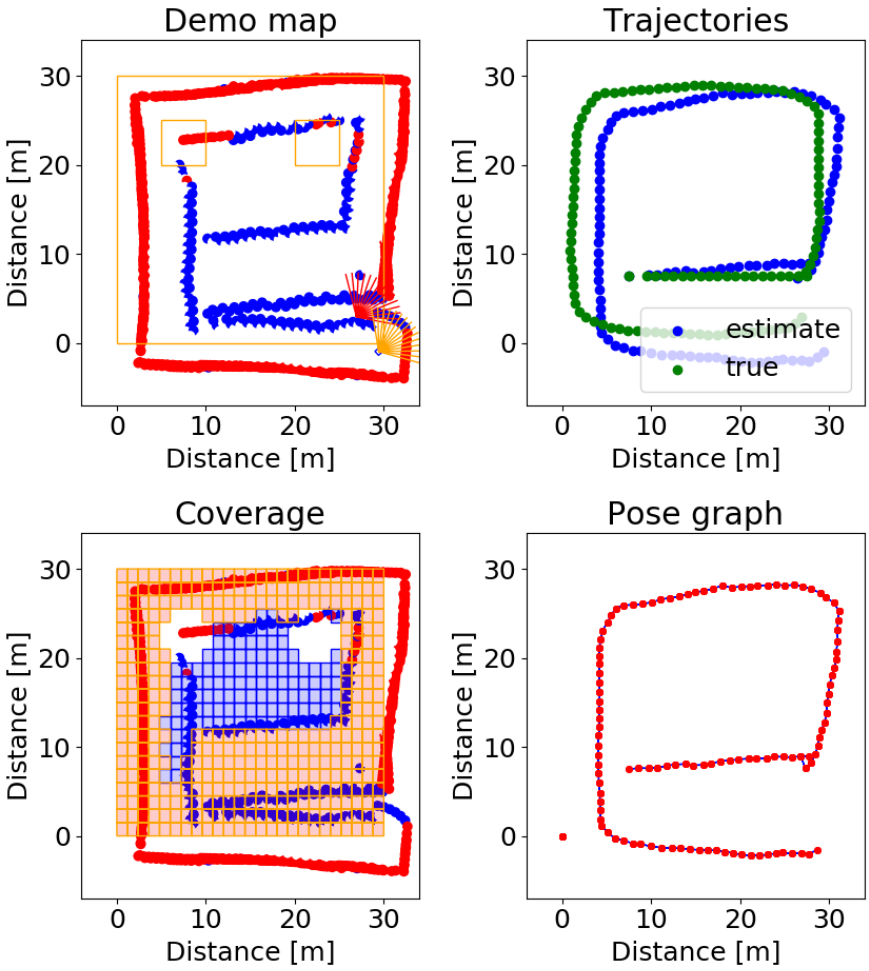
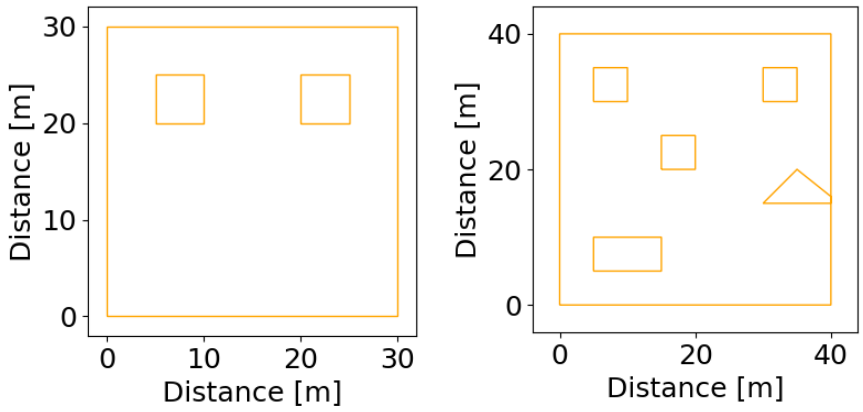


Figure 3.3: Experiment visualization: The demo map with the robot, its FOV, frontier and obstacle vertices; The true and estimated trajectories; The coverage plot with the explored cells in orange and the unexplored cells in blue as well as the frontier and obstacle vertices in blue respective red; The pose graph.

this data we can generate the coverage/time plots, as an example is shown in Figure 3.5, and can compare the exploration time.

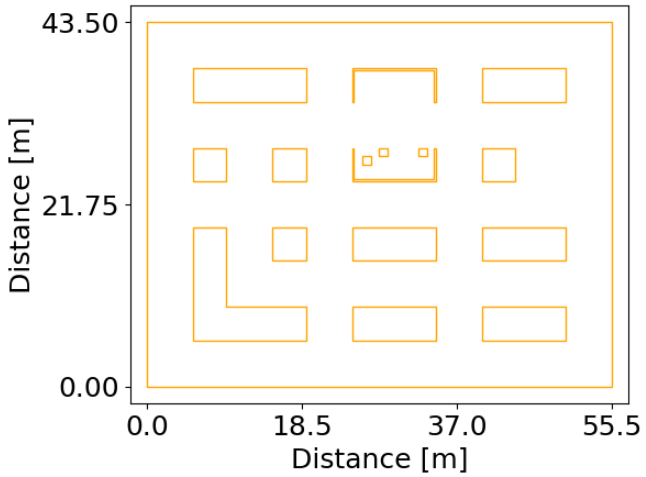
3.5 Experiments

We performed experiments on three different simulated maps, shown in Figure 3.4. While we created the first two maps on our own, the third map mimics *Scenario 2* from [12]. For all the experiments we used gaussian noise with mean $m_p = \begin{bmatrix}0\text{m} & 0\text{m}\end{bmatrix}^\mathsf{T}$ and standard deviation $\sigma_p = \begin{bmatrix}0.1\text{m} & 0.1\text{m}\end{bmatrix}^\mathsf{T}$ for the position estimates and gaussian noise with mean $m_o = 0\text{rad}$ and standard deviation $\sigma_o = 0.015\text{rad}$ for the orientation estimates.



(a) First map

(b) Second map



(c) Third map

Figure 3.4: The simulated maps.

3.5.1 First map

The coverage/time plot for the exploration with the proposed representation and the grid-based approach is shown in Figure 3.5. The explored map with the grid cells, the estimated and true trajectories, and the pose graph, are shown in Figure 3.6. With the proposed representation, the exploration takes 432.85s whereas with the grid-based representation it takes 280.21s.

3.5.2 Second map

The coverage/time plot for the exploration with the proposed representation and with the grid-based representation is shown in Figure 3.7. The explored map with the grid cells, the estimated and true trajectories, and the pose graph, are shown in Figure 3.8. With the proposed representation, the exploration takes 3030.37s whereas with the grid-based representation it takes 323.76s.

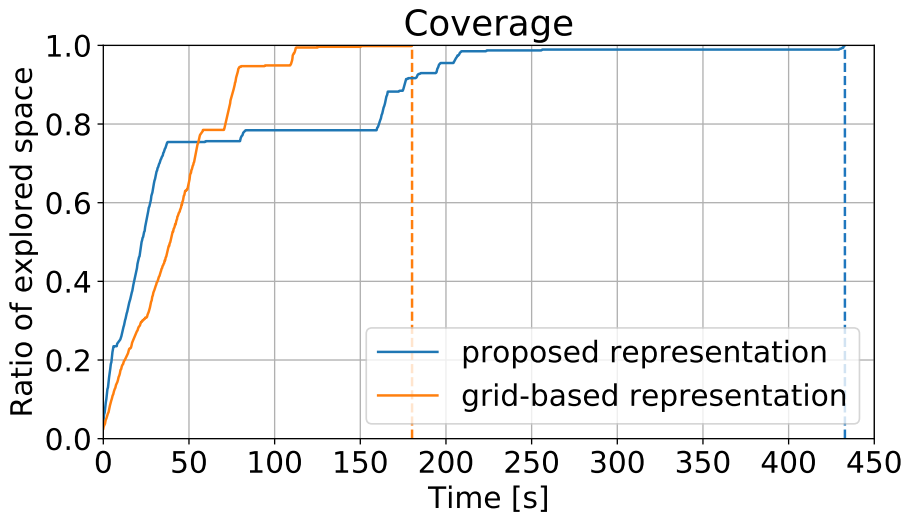


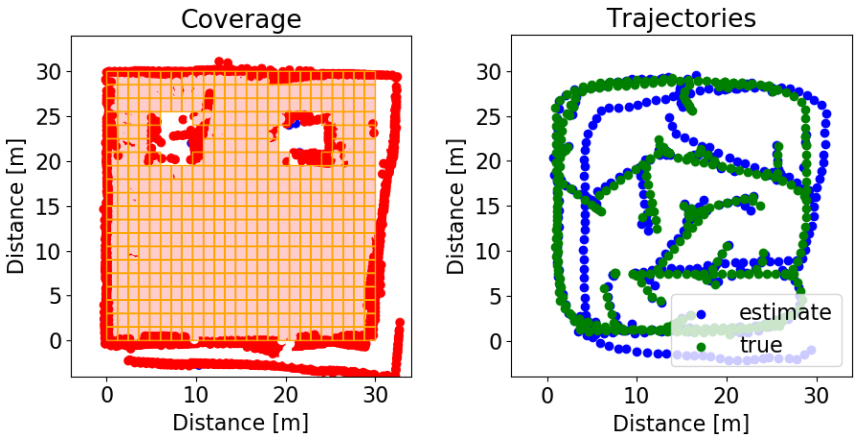
Figure 3.5: Coverage/time plot of the first map.

3.5.3 Third map

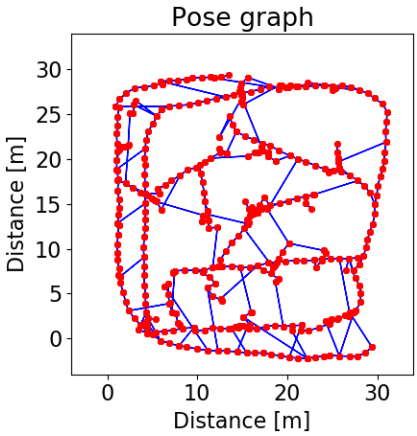
The coverage/time plot for the exploration with the proposed representation and the grid-based representation is shown in Figure 3.9. The explored map with the grid cells, the estimated and true trajectories, and the pose graph, are shown in Figure 3.10. With the proposed representation, the exploration takes $3166.36s$ whereas with the grid-based representation it takes $497.36s$.

3.5.4 Discussion

We have shown that the explorations on all maps with the proposed representation, as well as the grid-based representation, reached full coverage. We also compared the exploration time for both representations and showed that our proposed representation takes up to $9.36\times$ more time than the grid-based representation. The reason for this is, that if all grid cells are explored, there are still frontier polygon vertices located within explored grid cells, because the grid cells are a discrete (sampled) representation of the map and the polygons are a continuous representation. Another reason is, that the exploration with the grid-based representation uses the ground truth poses for the state estimates and not the noisy state estimates. A fair comparison is therefore not possible. Nevertheless, as we have performed the exploration with both representations and reached full coverage with both, we know that the exploration with our proposed representation in our experiments leads to full coverage.



(a) Coverage plot (b) True and estimated trajectories



(c) Pose graph

Figure 3.6: The grid cells, the trajectories and the pose graph at the end of the experiment of the first map.

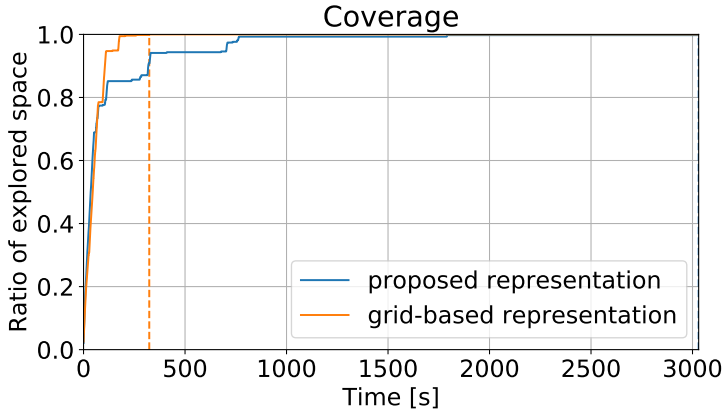
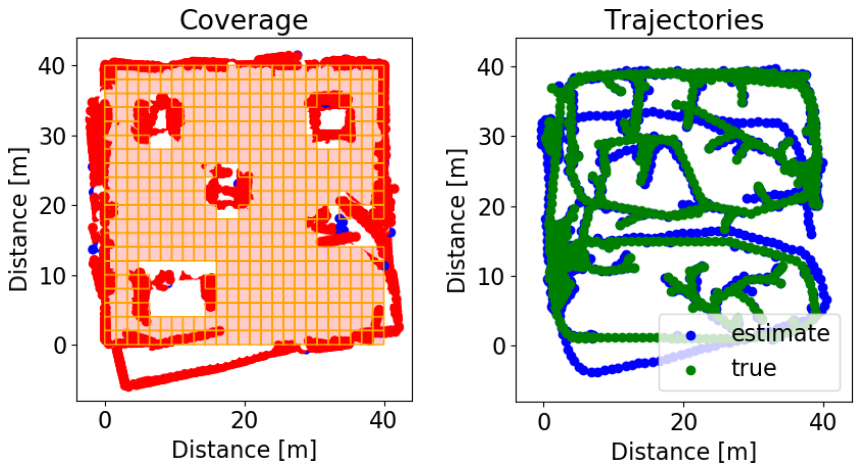
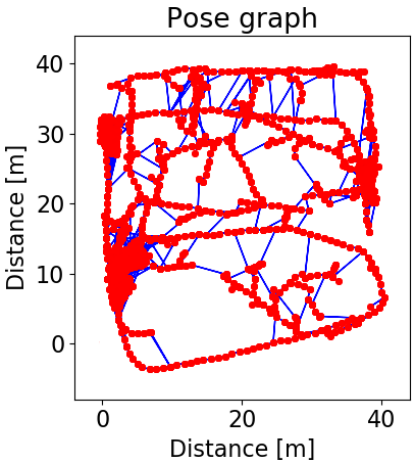


Figure 3.7: Coverage/time plot of the second map.



(a) Coverage plot

(b) True and estimated trajectories



(c) Pose graph

Figure 3.8: The grid cells, the trajectories and the pose graph at the end of the experiment of the second map.

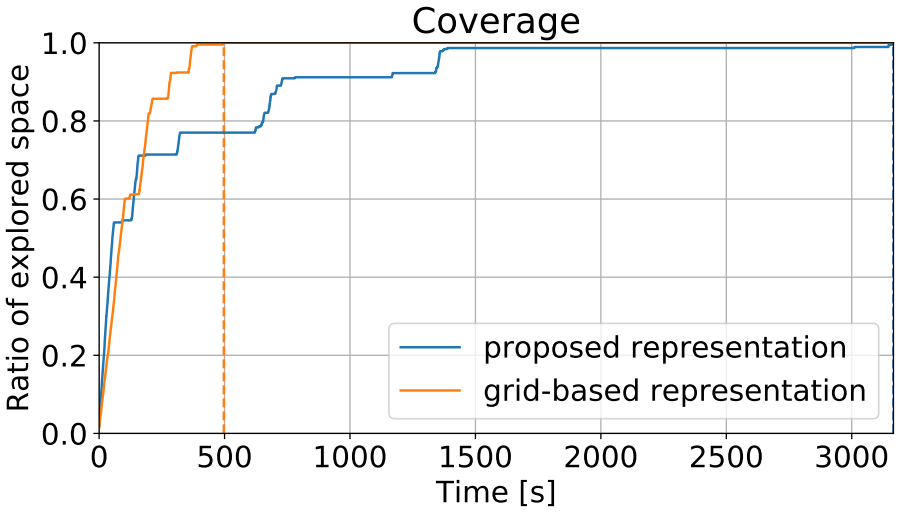
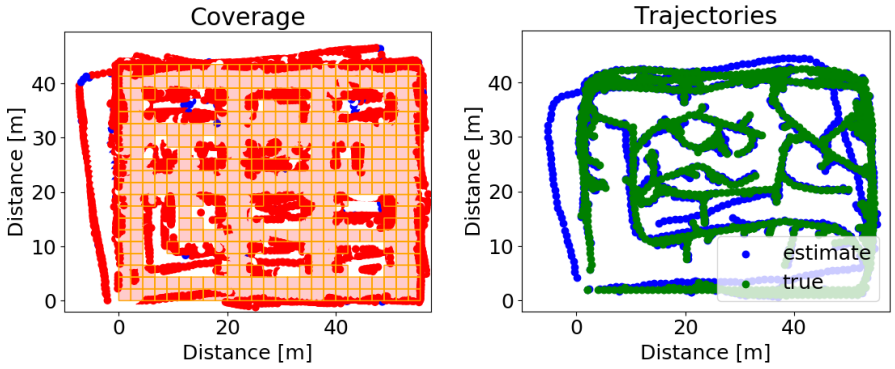
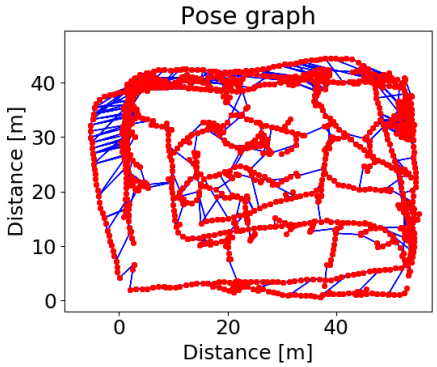


Figure 3.9: Coverage/time plot of the third map.



(a) Coverage plot (b) True and estimated trajectories



(c) Pose graph

Figure 3.10: The grid cells, the trajectories and the pose graph at the end of the experiment of the third map.

Chapter 4

Discussion

In this thesis, we developed a map representation that can deal with noisy state estimates, that can handle loop closures and is suitable for frontier based exploration. In both approaches, polygons are used for the representation. Polygons represent the boundary between known free space and obstacles or unknown space. The inside of polygons therefore implicitly represents free space.

4.1 First approach

In the first approach, the so called "active area" approach, we tried to build a global map with polygons. Each polygon vertices had a corresponding pose graph point. This allows to correct the polygons according to changes in the pose graph. The polygons are therefore deformable and the map can be adjusted in case of a correction of the pose graph, e.g. after a loop closure.

As we already discussed in Section 2.7.5, the active area approach has its limitations as we could not find a general way to resolve twisted polygons and as the size of the active area has to be large for certain scenarios and small for others.

4.2 Second approach

In the second approach, we do no longer build a global map, but work with the local polygon at each pose of the pose graph. Working with these local polygons has the advantage, that we do not have to deal with polygon self intersections which occurred in the first approach and therefore we do not need as complex maintenance for merging. Another advantage is, that the number of polygons we have to check for intersections with the polygon of the current FOV is bounded. If a polygon is n pose graph points away from the current pose graph point and does not intersect with the polygon of the current FOV, then a polygon $n + 1$ pose graph points away will not intersect either. As we

only work with relative transformations in the second approach, we do not rely on computationally expensive pose graph optimization after loop closures.

As shown in the experiments we presented in Section 3.5, our novel representation used with a frontier-based exploration approach, such as the one proposed in [4] leads to full coverage.

A disadvantage of the second approach is, that we do not get a global map implicitly. Another limitation could be the memory usage when the robot explores larger areas. As for every pose the polygon of the FOV is stored, this may lead to an extensive memory usage.

4.3 Conclusion

In this thesis we proposed a novel representation for frontier based exploration. In comparison with existing representation approaches, our representation can deal with noisy state estimates and does not have to be rebuilt after a change of the pose graph, e.g. after a loop closure. It is therefore more robust and computationally more economical.

4.4 Future Work

In this thesis, we demonstrated the proposed approach in 2D. An implementation of the approach in 3D would be the next thing to do. In 3D, polygons will be replaced by meshes and intersections of two meshes will be edges instead of points.

As mentioned, one disadvantage of the second approach is, that we do not get a global map implicitly. However, as we have the FOV of all the observations, one could create a global map out of this observations and the poses of the pose graph for example in post-processing. One possibility would be to create an occupancy grid map by ray casting every FOV.

The proposed approach is also applicable for scenarios with multiple robots. To realize such an extension to multiple agents, one has to determine what information needs to be shared across the agents, how and when. A second part of a multi robot extension that would need to be addressed is collaborative (distributed) planing, also an active field of research.

Bibliography

[1] Fabian Blöchliger, Marius Fehr, Marcin Dymczyk, Thomas Schneider, and Roland Siegwart. Topomap: Topological mapping and navigation based on visual slam maps. *arXiv preprint arXiv:1709.05533*, 2017.

[2] M. Bosse, P. Newman, J. Leonard, M. Soika, W. Feiten, and S. Teller. An Atlas framework for scalable mapping. *2003 IEEE International Conference on Robotics and Automation (Cat. No.03CH37422)*, 2(September):1899–1906, 2003.

[3] F. Bullo. *Lectures on Network Systems*. Version 0.96, 2018. With contributions by J. Cortes, F. Dorfler, and S. Martinez.

[4] Titus Cieslewski, Elia Kaufmann, and Davide Scaramuzza. Rapid Exploration with Multi-Rotors: A Frontier Selection Method for High Speed Flight. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017.

[5] C. Connolly. The determination of next best views. In *Proceedings. 1985 IEEE International Conference on Robotics and Automation*, volume 2, pages 432–435, Mar 1985.

[6] Paul Furgale and Timothy D. Barfoot. Visual teach and repeat for long-range rover autonomy. *Journal of Field Robotics*, 27(5):534–560, 2010.

[7] Ronald Goldman. Graphics gems. chapter Intersection of Two Lines in Three-space, pages 304–. Academic Press Professional, Inc., San Diego, CA, USA, 1990.

[8] H. H Gonzalez-Banos and J.-C. Latombe. Navigation Strategies for Exploring Indoor Environments. *The International Journal of Robotics Research*, 21(10-11):829–848, 2002.

[9] 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.

[10] D. Holz, N. Basilico, F. Amigoni, and S. Behnke. Evaluating the efficiency of frontier-based exploration strategies. In *ISR 2010 (41st International Symposium on Robotics) and ROBOTIK 2010 (6th German Conference on Robotics)*, pages 1–8, June 2010.

[11] Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. OctoMap: An efficient probabilistic 3D mapping framework based on octrees. *Autonomous Robots*, 34(3):189–206, 2013.

[12] Miguel Juliá, Arturo Gil, and Oscar Reinoso. A comparison of path planning strategies for autonomous exploration and mapping of unknown environments. *Autonomous Robots*, 33(4):427–444, 2012.

[13] J. Maver and R. Bajcsy. Occlusions as a guide for planning the next view. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(5):417–433, May 1993.

[14] Alexander Millane, Zachary Taylor, Helen Oleynikova, Juan I. Nieto, Roland Siegwart, and Cesar Cadena. TSDF manifolds: A scalable and consistent dense mapping approach. *CoRR*, abs/1710.07242, 2017.

[15] Richard A. Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohli, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. KinectFusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE International Symposium on Mixed and Augmented Reality, ISMAR 2011*, pages 127–136, 2011.

[16] Christos Papachristos, Shehryar Khattak, and Kostas Alexis. Uncertainty-aware receding horizon exploration and mapping using aerial robots. In *Proceedings - IEEE International Conference on Robotics and Automation*, pages 4568–4575, 2017.

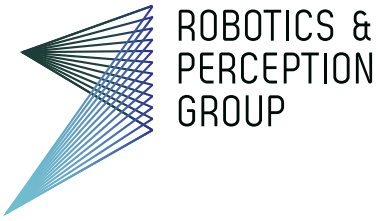
[17] Patrik Schmuck, Sebastian A. Scherer, and Andreas Zell. Hybrid Metric-Topological 3D Occupancy Grid Maps for Large-scale Mapping. *IFAC-PapersOnLine*, 49(15):230–235, 2016.

[18] Jan Oliver Wallgrün. *Hierarchical voronoi graphs: Spatial representation and reasoning for mobile robots*. 2010.

[19] Thomas Whelan, Michael Kaess, John J. Leonard, and John McDonald. Deformation-based loop closure for large scale dense RGB-D SLAM. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 548–555, 2013.

[20] Thomas Whelan, Stefan Leutenegger, Renato F Salas-moreno, Ben Glocker, and Andrew J Davison. ElasticFusion : Dense SLAM Without A Pose Graph. *Robotics: Science and Systems*, 2015(December), 2015.

[21] B. Yamauchi. A frontier-based approach for autonomous exploration. In *Computational Intelligence in Robotics and Automation, 1997. CIRA'97., Proceedings., 1997 IEEE International Symposium on*, pages 146–151, Jul 1997.



Title of work:

A Representation for Exploration that is Robust to State Estimate Drift

Thesis type and date:

Master Thesis, April 2018

Supervision:

Titus Cieslewski
Prof. Dr. Davide Scaramuzza
Prof. Dr. Roland Siegwart

Student:

Name:	Andreas Ziegler
E-mail:	anziegle@ethz.ch
Legi-Nr.:	09-253-048

Statement regarding plagiarism:

By signing this statement, I affirm that I have read the information notice on plagiarism, independently produced this paper, and adhered to the general practice of source citation in this subject-area.

Information notice on plagiarism:

`http://www.lehre.uzh.ch/plagiate/20110314_LK_Plagiarism.pdf`

Zurich, 8. 4. 2018: _____