POLITECNICO DI MILANO
MSc in Computer Science and Engineering
School of Industrial and Information Engineering

A system for automatically evaluating the quality of maps built by autonomous mobile robots

Supervisor: Prof. Francesco Amigoni
Co-Supervisor: Dr. Matteo Luperto

Master thesis by:
Matteo Pasina, ID 837816

Academic year 2017-2018

Abstract

The main contribution of this thesis is related to the research field of SLAM (Simultaneous Localization And Mapping), in which a mobile robot builds a map of an environment while simultaneously localizing itself in it. This problem is well known within the field of mobile robotics and has already been successfully addressed with several probabilistic algorithms that produce the map of the environment that the robot has explored. SLAM algorithms do not take into account the reliability of the produced map, i.e., how much the map reconstructed by the robot is similar to the actual map of the environment, called *ground truth*. Knowing, without running a SLAM algorithm, the quality of the map produced in a given environment would be of great advantage for research and industrial applications. For example, you might know in advance if the algorithm is expected to build a map that is accurate enough to allow the robot to complete its task. In order to have such knowledge, it would be necessary to know how an algorithm behaved in different environments in order to correlate its performance with the characteristics of the environment. With this thesis we want to move a step towards the goal of having a method to predict the quality of the map produced by a SLAM algorithm by presenting a system for the automatic evaluation of the maps. The entire system has been implemented using the Robot Operating System (ROS) framework.

Sommario

Questo lavoro di tesi si colloca nell'ambito dello SLAM (Simultaneous Localization And Mapping), termine che indica l'operazione con cui un robot mobile costruisce incrementalmente una mappa di un ambiente localizzandosi allo stesso tempo al suo interno. Possibili soluzioni al problema di SLAM sono state proposte con successo con diversi algoritmi probabilistici che producono la mappa dell'ambiente che il robot ha esplorato. La qualità di una mappa ottenuta mediante algoritmi di SLAM, ovvero quanto sia aderente all'ambiente che rappresenta, non è nota a priori. Riuscire a conoscere, senza eseguire l'algoritmo, la qualità della mappa prodotta in un determinato ambiente sarebbe di grande vantaggio per applicazioni di ricerca e industriali. Per esempio si potrebbe sapere in anticipo se l'algoritmo in questione costruisce una mappa abbastanza accurata per permettere al robot di portare a termine il suo compito. Per fare questo servirebbe conoscere come si comporta un algoritmo in diversi ambienti per poter correlare le sue prestazioni alle caratteristiche dell'ambiente. Con questa tesi si vuole compiere un passo verso l'obiettivo di avere un metodo per prevedere la qualità della mappa prodotta da un algoritmo SLAM, presentando un sistema di valutazione automatica delle mappe. Tutto il sistema è stato implementato usando il framework Robot Operating System (ROS).

Chapter 1

Introduction

Autonomous mobile robots, in order to complete a task, usually rely on a map representing the environment. This map is not always provided a priori. Sometimes, robots have to build a map of the environment while simultaneously determining their location within this map. This problem is called Simultaneous Localization and Mapping (SLAM) [11], or Concurrent Mapping and Localization (CML), and is a fundamental challenge of mobile robotics [42]. To address this issue a robot can rely on sensors, that can be of different types. These sensors receive data from the world in which the robot is located. For example, a laser range finder sends laser beams and measures the time taken by these beams to be reflected off the objects and returned to the range finder, in this way it can compute the distance of the objects. Another example is represented by the sensors that measure the movements, namely the odometry, of the robot. However, since the information collected by these sensors is not completely reliable, there is the need of SLAM algorithms. These algorithms try to find a correct estimate of the map while localizing the robot, within it. Different SLAM algorithms [14, 17, 41], with good results, have been developed to solve SLAM problems. However, to the best of our knowledge, there is not any standard way of assessing the quality of the map produced by these algorithms in an automated way.

The objective of this thesis is to build a system that evaluates the quality of a map built by a SLAM algorithm in an environment without the intervention of a human. Being paired with simulations, the proposed system allows to generate a large quantity of data on the performance of SLAM algorithms. With these data it will be possible to correlate the quality of the SLAM algorithm's map to some features of the environments, and, given a new environment, predict, based on its features, the quality of the map potentially produced by the algorithm.

In order to evaluate map quality we have to define what is map quality, so we study different approaches in the literature. Many of the approaches rely on the visual similarity between the map built by the SLAM algorithm and the ground truth map, and often a visual similarity check is performed by a human in a qualitative way. The approach that we select as reference is the one, developed by Kümmerle et al. [24], which assumes that if the trajectory reconstructed by the SLAM algorithm is similar to the ground truth trajectory, then the map is accurate. In particular, it computes the displacement between pair of poses on the trajectory estimated by the SLAM algorithm and those on the ground truth trajectory, that are the real poses that the robot assumes. However, the original method proposed in [24] has a human component in it: the ground truth trajectory file is built by a human who knows the structure of the environment. We build a system that generates in an automated way the ground truth trajectory file that has to be compared with the estimated trajectory produced by the SLAM algorithm. This method [24] have been proved to be reliable if a reliable ground truth path is available [23]. Thus, we use simulations to have access to the ground truth trajectory followed by the robot while navigating in the environment. Simulations have also other useful properties: first, we can repeat evaluations of different explorations, with the same conditions of the environment. Second, we can test the system on many different environments with little effort. Third, we can let the system run generating evaluation data for hours or days without logistic problems.

SLAM algorithms are influenced by different factors, that include environment, robot path, sensors, computing power, and the algorithm used. The SLAM algorithm evaluated is GMapping [1], that is based on a particle filter approach. The posterior probability is represented as a set of possible robot locations and maps (the *particles*) and is updated using the sensor input. The quality of the map depends on the number of particles used to represent the posterior probability. As the number of particles goes to infinity, nonparametric techniques tend to converge uniformly to the correct posterior [42]. A weight is associated with each particle, indicating how well the observations correspond to that particle. At the end of an iteration a new set of particles is sampled using the calculated weights. GMapping is a well known SLAM algorithm with generally good performance and reliability [34]. We try to evaluate a SLAM algorithm performance related to the characteristics of the explored environment.

A problem that we face is the autonomous exploration of the environments. Since we have to automatize the generation of data we must have a good navigation algorithm that explores the environment autonomously, getting stuck less possible. We develop the system using the ROS (Robot Operating System) framework [33], so we use the Navigation Stack provided. However, the Navigation Stack shows limits in the exploration, and some tuning of the parameters and code adjustments are necessary to improve its performance. The simulator used is Stage [13]. The implemented system launches all the processes needed for the exploration and mapping of an environment, and a Python script checks if the exploration is completed and, if so, it starts another exploration. In future, environments can be classified according to features and can be related to the quality of the map produced by the SLAM algorithms. This relation will result in a system able to predict the capacity of a SLAM algorithm to build a correct map in a new environment, only by knowing its relevant features.

The structure of this thesis is the following: in Chapter 2 we discuss a number of works that are related to ours and that were of inspiration for this study. In Chapter 3 we define the evaluation method and the problems we are dealing with in more detail. In Chapter 4 we introduce the way in which we generate the data for the evaluation of the map quality. In Chapter 5 we present the software architecture of the system. In particular, we first give a short overview of the ROS framework and the Stage simulator, over which we build and test our system, and then we give the details of the implementation of the architecture. In Chapter 6 we show the experiments we perform to test our system, along with the obtained results. In Chapter 7, we conclude by summarizing the purposes and the final evaluations of this thesis. Some suggestions for future works are also proposed.

Chapter 2

State of the art

In robotics, when the robot does not have access to a map of the environment, nor it has not access to its own poses Simultaneous Localization and Mapping (SLAM), problems might arise. SLAM, also known as Concurrent Mapping and Localization (CML), is one of the fundamental challenges of robotics, dealing with the necessity of building a map of the environment while simultaneously determining the location of the robot within this map [40]. To map an environment and localize itself, the robot relies on sensors, which gather data from the world but, since they are often erroneous, we cannot rely directly on them to get the robot position and the position of the obstacles in the world. SLAM algorithms try to solve this problem, through estimation of the map and the localization within it.

Many kinds of sensors can be used for mapping and localization. Range sensors are one of these and suffer from four different types of noise: small measurement noise, errors due to unexpected objects, errors due to failures to detect objects, and random unexplained noise. The first type of noise happens for the limited resolution of the sensors and atmospheric effects on the measurement signal. Errors due to unexpected objects are caused because the environment where the robot moves is dynamic, this means that objects detected can move, typical examples of moving objects are people that share the operational space of the robot. The error occurs when a object detected moves, then the obstacle map has to be updated, but this can require time (i.e., the next scan). Sometimes, obstacles are missed altogether. For example, this happens frequently with sonar sensors when measuring a surface at a steep angle. Failures also occur with laser range finders when sensing black, light-absorbing objects, or when measuring objects in bright light. Finally, range finders occasionally produce entirely unexplained measurements. For example, sonars often generate phantom readings when their signals bounce off walls, or when they are subject to cross-talk between different sensors. Another kind of sensors are the motion sensors that provide odometry. Odometry is commonly obtained by integrating wheel encoders information (distance traveled, angle turned) and most commercial robots make such integrated pose estimation available in periodic time intervals (e.g., every second). Also these sensors suffer from errors, for example drifting and slippage. These imperfect sensors make the use of SLAM algorithms necessary to estimate the map and the localization of the robot within this map.

Despite SLAM is by this time well known and well-defined (in planar indoor environments could be considered as an almost solved problem [45]) to the best of our knowledge there is not a standard evaluation methodology for SLAM algorithms, although many proposals have been discussed in the robot community, as we will illustrate in this Chapter. A quantitative assessment of the reliability of the map built and the localization estimated is of high interest for many reasons. First of all, it allows individual researchers to quantify the quality of their SLAM approach and to study the effects of system specific choices made, like different parameter values, in an objective way. Secondly, it allows researchers to rank the quality of their different approaches to determine scientific progress; similarly, it allows rankings within competitions like RoboCup.

The purpose of this chapter is to give an overview of the current state of the art of SLAM algorithms and to present the techniques used to evaluate the quality of these algorithms. Assessing the quality of maps in a simple, efficient and automated way is not trivial and represents an ongoing research topic [35]. In this chapter we also try to define from the literature what is usually intended as map quality.

2.1 SLAM algorithms

SLAM is concerned with the problem of building a map of an unknown environment by a mobile robot, while at the same time navigating the environment using the map. The term SLAM is, as stated, an acronym for Simultaneous Localization And Mapping. It was originally developed by Hugh Durrant-Whyte and John J. Leonard [27] based on earlier work by Smith, Self and Cheeseman [37]. Durrant-Whyte and Leonard originally termed it SMAL but it was later changed to give a better impact.

For mapping and localization there is the need of a representation of the environment, that is the *map*. Maps can be represented in many different ways. In robotics, the most popular representations are grid maps for two dimensions and point clouds or voxel maps for three dimensions. 2D maps were the earliest representation while 3D representations demand much more computing power and memory, but are becoming increasingly popular. Grid maps represent the area of the environment as a matrix. Generally each cell represents whether the portion of space it represents is occupied by an obstacle, free or unobserved. The size of the cells depends on the resolution of the grid. In probabilistic grid maps every cell is represented by a probability of being occupied or free, usually a value between 0 and 255 is used in order to occupy only one byte of memory. The map is initialized with the unobserved value between the minimum and maximum (128). Subsequent observations then update this value: the higher the probability of being occupied the higher the value and vice versa.

Voxel maps a 3D grid, where each cell is a cubic volume of equal size, to discretize the mapped area. The problem with this approach is the high memory consumption. Point clouds use the distance of the obstacles returned by range sensors, such as laser range finders or stereo cameras, to model the occupied space in the environment. The drawback is that free space and unknown areas are not modeled [18]. Also 2.5D maps can be used as a model of the environment. Typically, a 2D grid is used to store the measured height for each cell.

SLAM algorithms use the sensors to extract landmark points that are distinct, stationary features of the environment that can be recognized reliably. These landmarks are the basis for localization and mapping. Since the errors are not predictable, SLAM algorithms are based on probability: given the recorded observations of the range sensors and control inputs together with the initial state of the robot, SLAM algorithms compute the new landmark locations and the robot position. As stated in [11] the probability distribution

$$P(x_k, m | z_{0:k}, u_{0:k}, x_0) \tag{2.1}$$

has to be computed for all times k, where x_k is the state vector describing the location and orientation of the robot, m is the set of all landmarks, $z_{0:k}$ is the set of all landmark observations, $u_{0:k}$ is the history of control inputs and x_0 is the initial state of the robot. This is a general probability distribution for the SLAM problem, but it can be slightly different depending on the implementation. The process of updating the distribution can be divided in five steps: landmark extraction, data association, state estimation, state update and landmark update. There are many ways to solve each of these steps.

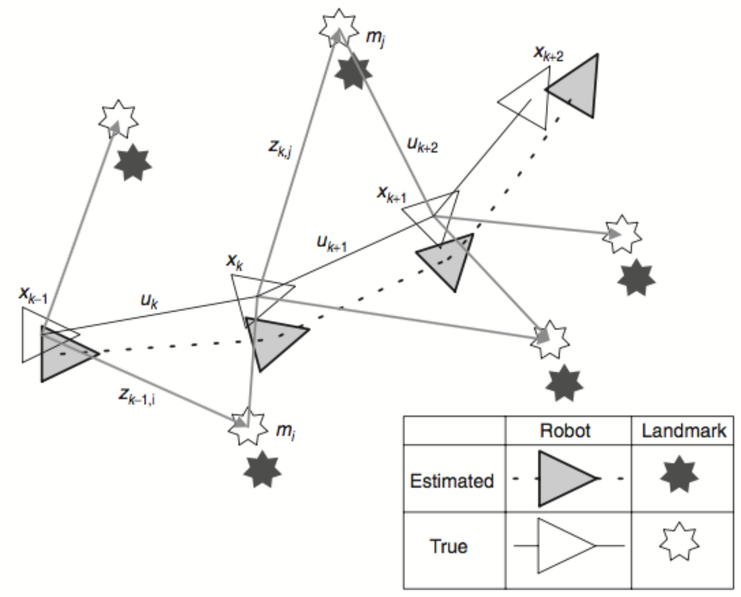


Figure 2.1: The essential SLAM problem. A simultaneous estimate of both robot and landmark locations is required. The true locations are never known or measured directly. Observations are made between true robot and landmark locations. Image from [11]

Since there is not a single way to solve this problem many SLAM algorithms have been proposed, the more relevant are based on:

- Extended Kalman filters;
- Expectation maximization;
- Particle filters;
- Graph-based approaches.

Historically, the earliest, and perhaps the most influential SLAM algorithm is based on the extended Kalman filter, or EKF. In a nutshell, the EKF SLAM algorithm applies the EKF to online SLAM using maximum likelihood data association. Using an EKF the estimates of the position of the landmarks and the robot are kept and updated if a loop is closed (a landmark is re-visited) [16]. EKF SLAM has indeed been applied successfully to a wide range of practical mapping problems, but a key limitation of it lies in the necessity to select appropriate landmarks. By doing so, most of the sensor data is usually discarded. Further, the quadratic update time of the EKF limits this algorithm to relatively scarce maps depending on the hardware computing capability.

Expectation Maximization is an optimization algorithm [41] based on landmarks, in which the robot motion and the perception are statistically processed. In this maximum likelihood estimation problem the location of landmarks and the robot position are estimated using two steps. The expectation step keeps the current map constant and calculates the probability distribution of past and current robot locations. Then the most likely map is computed in the maximization step. This is based on the estimation result of the expectation step. Through the alternation of these two steps SLAM is achieved, that results in a local maximum in the likelihood space. According to [6], it is computationally very extensive and cannot incrementally build maps.

Particle filters are nonparametric filters. These filters are recursive Bayesian filters using Monte Carlo simulations to keep track of the robot location [17], [15]. Nonparametric filters do not rely on a fixed functional form of the posterior, such as Gaussians. Instead, they approximate posteriors by a finite number of values, each roughly corresponding to a region in state space. The quality of the approximation depends on the number of values used to represent the posterior probability. As the number of values goes to infinity, nonparametric techniques tend to converge uniformly to the correct posterior [42]. The posterior probability is represented as a set of possible robot locations and maps (the *particles*) and is updated using the sensor input. A weight is associated with each particle, indicating how well the observations correspond to each other. At the end of the iteration a new set of particles is sampled using the calculated weights.
Particle filtering can be very inefficient in high-dimensional spaces. To make it treatable the use of a technique called Rao-Blackwellization is needed. A Rao-Backwellized Particle-Filter or RB-PF, estimates the joint posterior of SLAM using the following factorization:

$$P(x_{1:t}, m|z_{1:t}, u_{1:t-1}) = P(m|x_{1:t}, z_{1:t}) \cdot P(x_{1:t}|z_{1:t}, u_{1:t-1}) \qquad (2.2)$$

This factorization allows us to first estimate only the trajectory of the robot, and then to compute the map given that trajectory. Since the map strongly depends on the estimated pose of the robot, this approach offers an efficient computation.

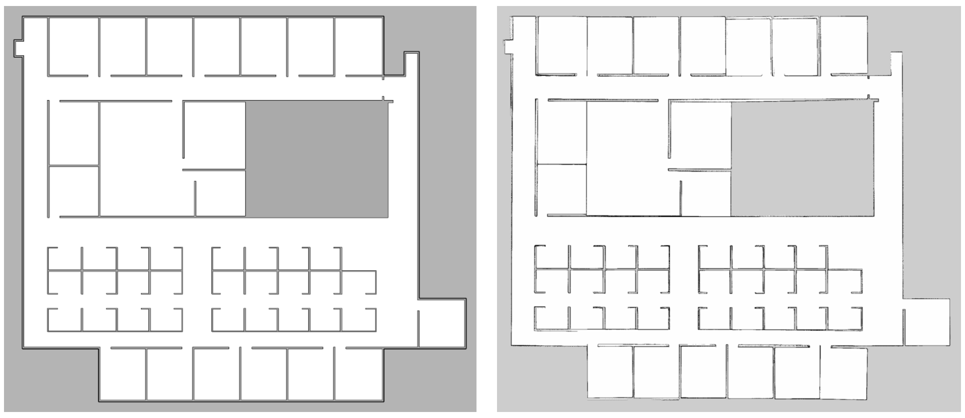


Figure 2.2: The ground truth image used by a simulator and the corresponding grid map built by GMapping

The most popular implementation of this kind of filters is GMapping [1]. It is available online at OpenSLAM.org[1], the C++ code is open source and there is also a Robot Operating System (ROS) implementation. It has several improvements to reduce the number of particles needed. The approach computes the proposal distribution taking into account also the most recent observations, first a scan-matcher (i.e., an algorithm that associates similar scans) is used to determine the meaningful area of the observation likelihood function. Then the sample is taken in that meaningful area. This helps to have a more accurate proposal distribution in the prediction step of the filter. Furthermore, the resampling step can potentially eliminate the correct particle. This effect is also known as the particle-depletion problem, or particle impoverishment. In [1] this step is improved using a adaptive resampling strategy, allowing to perform a resampling step only when needed, and in this way keeping a reasonable particle diversity. An example of the output map of GMapping is showed in Figure 2.2.

The last approach presented here is graph-based SLAM. In this kind of solution to the SLAM problem the sensor data is saved in nodes and the robot movement in the edges between the nodes [14]. The main idea behind graph-based SLAM methods is to use a graph to represent the problem in which every node in the graph corresponds to a pose of the robot during mapping and every edge between two nodes corresponds to a spatial constraint between them, the goal is to build the graph and find a node configuration that minimize the error introduced by the constraints.

[1]https://www.openslam.org/gmapping.html

2.2 Evaluation of the map quality

In the previous section we showed that SLAM is a problem that can be solved with different techniques. Every technique has pros and cons and understanding which works better is useful when it comes to choosing an algorithm. Unfortunately, since these algorithms are different, they provide different kinds of maps and also different kinds of errors. Having a single, consistent way of assessing the quality of the map is not a trivial task. In the area of grid-based mapping techniques, people often use visual inspection to compare maps with blueprints of buildings [24]. Map quality is not the only thing that matters in SLAM algorithms, and other ways to evaluate an algorithm consider different factors, for example the time in which the map is built and the computing load needed. The relative importance of the factors depends on the task that the robot has to achieve.

The derivation of meaningful quantitative assessments of map quality is desirable for many reasons. It can allow a ranking, which can be used in competitions like RoboCup. Also, in a work that tries to improve a SLAM algorithm, quantitative results can be compared to understand if a new feature enhances the quality. Another useful consequence of a quantitative measure is that it can allow researches to compare their systems and provide information about the development of the field as a whole.

A way of assessing the quality of a map is by using different attributes [36]. Those attributes can be measured separately and weighed according to the needs of the application [26]. Those attributes can include:

- Coverage: how much area was traversed/visited;
- Resolution quality: to what level/detail are features visible;
- Global accuracy: correctness of positions of features in the global reference frame;
- Relative accuracy: correctness of feature positions after correcting (the initial error of) the map reference frame;
- Local consistencies: correctness of positions of different local groups of features relative to each other;
- Brokenness: how often is the map broken, i.e., how many partitions of it are misaligned with respect to each other by rotational offsets [9].

Another point that doesn't allow a single evaluation of the algorithms is that the evaluation methodologies are often biased by hardware and settings.

Indeed the quality of a map produced by a robot is influenced by many different factors:

- Environment: sparse environments are difficult for most mapping approaches. SLAM algorithms perform better in structured datasets than unstructured ones, because features can be more easily identified [28];

- Robot path: the path that a robot took to gather the sensor data can contain different occurrencies of loops. A loop is closed when the robot returns in a location that it has already seen. Loop closing is the main way to reduce the accumulated error in the map [32];

- Sensors: the range, field of view, accuracy, and the position of the sensor on the robot are factors influencing mapping algorithms;

- Computing power: SLAM algorithms can be very computational intensive and computing power on mobile robots can be limited. Moreover process time is often restricted in real time applications because other tasks depend on mapping and localization, like path planning. To reduce the hardware load several limitations to the algorithms can be applied, for example use less scans from the range sensors or execute less frequently loop closing algorithms, at the cost of a probably worse performance;

- Algorithm: the mapping algorithm itself influences the map quality to a great extend.

After analyzing the factors that influence the evaluation of a map we had to find a method to determine the quality of this map in a quantitative way and not only by human visual inspection.

Here follows a review on the techniques for map evaluation proposed in the literature:

Yairi [46] proposed to use Least Mean Squares of Euclidean Distances (LMS-ED) to measure the distance between similar points from both maps. As well as any similar Euclidean neighborhood-based metric LMS-ED is expensive to compute. It is hence usually not applied to all cells of the occupancy grids, but only to a very limited sets of landmarks.

Varsadan, Birk and Pfingsthorn [43] use an image based approach. The similarity function defined in this work is the sum over all colors of the average Manhattan-distance of the pixels with color c in picture a to the nearest pixel with color c in picture $a^{'}$ and vice versa. In the context of occupancy grid maps, the color c simply corresponds to occupancy information. Unlike LMS-ED this metric can be computed very efficiently, namely in linear time. They tested it on four type of errors: salt and pepper noise, translation, rotation, and scaling. Maps with a very good geo-reference can be evaluated quite nicely, but slightly broken maps or errors in the initial frame of reference lead to bad values, although the map might look very nice.

Lakaemper and Adluru [25] use a comparison with a ground truth map. They get from the laser scans lines and rectangles of the environment with a method called Virtual Scans and align it with the ground truth. The alignment energy quantify the similarity of the two maps. This metric measures topological correctness and can also quantify the global correctness. It can maybe have problems with unstructured environments because it is based on lines and rectangles.

The metric of Wagan, Godil and Li [44] is a feature based approach comparing a map to a ground truth map. They focus on the structural details in the map, proposing a novel method to assess the map quality based on three separate algorithms, each corresponding to a different type of features found in the map. These are: Harris Corner Detector, Hough Transforms, and Scale Invariant Feature transform. These measures will give three values which can be used to assess the quality of the map in three different terms. The three algorithms are run on the map produced and on the ground truth map and then the ratio between the set of features is the metric assessing the quality. This system is only suitable for offline-measurement for the quality of the maps and, as mentioned in the paper, noise, jagged lines, and distortions pose problems to the feature detectors. This approach can thus only be applied to nearly perfect maps.

Pellenz and Paulus [31] also propose to use feature extraction. They use Speeded Up Robust Features (SURF) and extract rooms as features from the grid map. Then the translation vector and the rotation matrix that minimize the location error of the matched features are computed. Quality is computed as the average match error over features. This is a completely automatic way for evaluation. However, experimental results of the metric are not presented.

All the metrics presented till now are based on the comparison of the map produced by the SLAM algorithm with the ground truth, but there are other ways of judging SLAM algorithms. Every algorithm produces a trajectory that is the sequence of estimated poses, so if the ground truth trajectory is available the deviation can be computed. This kind of evaluation have the important property to be independent of the sensor setup of the robot, and this allows for example to compare laser-based systems with vision-based ones. Here the assumption is that correct localization is a sufficient indicator for a good map quality.

In [45] a system to evaluate the quality of a 3D map, obtained with Monte Carlo Localization (MCL) for matching 3D scans, with a map obtained from the land registry office has been developed. However the MCL output is checked by a human to be sure that the path computed is feasible as ground truth. The comparison between generated map and ground truth is carried out by computing the Euclidean distance and angle difference of each pose. Standard deviation and maximum error of the trajectory are provided for comparisons. Comparing the absolute error between two trajectories might not yield a meaningful assertion in all cases. This effect gets even stronger when the robot makes a small angular error especially at the beginning of the dataset (and when it does not return to this place again). Then large parts or the overall map are likely to be consistent since the error is only at the beginning of the path. However the output error given by this system will be huge because every pose after the initial error will be considered erroneous.

The more relevant metric of this type is the one introduced in [24] that is also the basis of our work. It is based on the deformation energy that is needed to transfer the estimate into the ground truth. Unlike in [45], the displacement from every pose of the real path is computed independently from the others. So if there is a error at the beginning of the mapping this is not carried on for the rest of the process. It will be described with more precision in Chapter 3. This method is an excellent metric for map evaluation if a ground truth path is available. It is used in [23] to test the performance of three teams in the RoboCupRescue Interleague Challenge 2009 and in [28] to compare different open source packages of visual SLAM.

Birk [9] computes one interesting attribute of maps that can be used for map evaluation: the level of brokenness. This value tries to capture the structural errors in grid maps, structural error here means a fault that affects the global spatial layout. The map is partitioned into regions that are locally consistent with ground truth. Brokenness measures the number of these regions not aligned with the reference map. It can be use to determine the structural quality of a map in a quantitative way.

At last, in [35] it is proposed a metric for topology graphs. From the 2D grid map the nodes and edges of the graphs are extracted to have a model of the structure of the environment. Based on a similarity metric on

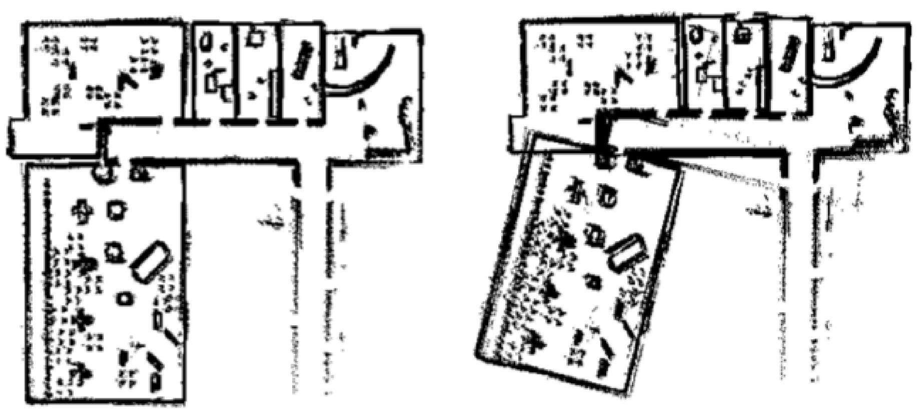


Figure 2.3: A typical example of "broken" map (right) and the same map without errors. Image from [9]

vertices in topology graphs, the vertices can be matched across maps and spatial (dis)similarities, and hence errors in the mapping, can be identified and measured. More precisely, the vertex-similarity is the basis to match the structures of topology graphs up to the identification of subgraph isomorphisms through wave-front propagation. This allows to determine map quality attributes up to very challenging structural elements like brokenness.

2.2.1 Comparision between SLAM algorithms

Another way to evaluate SLAM algorithms is to compare the algorithms in the same environment and see which performs better. This is an interesting method but the test cases have to be chosen accurately. In many cases the map in which the algorithms are tested is not very challenging and this can lead to misleading results. An algorithm that works well in a small room can fail to handle more large environments with respect to another that is less accurate but more efficient. Further, using few maps is not really meaningful because of the varying performance of SLAM algorithms, depending on the way they extract landmarks and associate data. This means that an algorithm can overcome the accuracy of another in a type of map but perform worse elsewhere.

In the work presented in [34] the available laser-based 2D SLAM algorithms already implemented in ROS are compared. The experiments are run both in simulated and real world environments. The algorithms tested are HectorSLAM, GMapping, KartoSLAM, CoreSLAM and LagoSLAM. The robot has been teleoperated by a human in all the experiments, the results showed that GMapping is robust in different environments and with a low computational complexity. The best results are provided by Karto and GMapping. The evaluation of the quality is given by a comparison of the ground map and the generated map. The distance from each occupied cell of the ground truth map to the nearest cell in the resulting map is determined computing the k-nearest neighbor cell. The sum of all distances obtained is then divided by the number of occupied cells in the ground truth map. The test were performed in 2 maps with low degree of complexity, so more investigation can be performed.

In [8] three different bearing-only SLAM algorithms are evaluated: Extended Kalman Filters, Incremental Maximum Likelihood and Rao-Blackwellized Particle Filters. The experiments were conducted on simulated environments with different noise, landmark density, and paths. To check how well an algorithm has performed two criteria are considered. The first one is robustness that is the percentage of successful runs: a run is considered successful when the localization and mapping errors are below two empirically chosen thresholds, however, it is not clear how the error is computed. The other criteria is efficiency that is the average amount of time in milliseconds that each method spends per step. Rao-Blackwellized PFs showed robustness with respect to the other two approaches. Also here the environment is not particularly complex: a simulated rectangle with low (5 landmarks) or high (100 landmarks) landmark density.

In [28] a comparision of different open source packages of visual SLAM algorithms is performed. The algorithms were tested on a variety of environments (including indoor, outdoor, and underwater) and vehicle types (including terrestrial, air-borne, marine surface, and underwater platforms), here all the hardware settings are well documented. The algorithms were evaluated using the provided datasets from each package using cross-validation (i.e., evaluating every algorithm on the datasets provided in the other packages) and in eight new datasets. In [28], the metric evaluation from [24] is the basis to compute the quality of the maps produced by the SLAM algorithms but without considering the rotational error. Human interaction is needed to control the ground truth data. Also track loss percentage, that is the ratio between the time in which the system is not localized and the total time of the dataset, and memory usage are considered. The results show that the SLAM algorithms evaluated perform better in structured datasets than in unstructured ones, because features can be more easily identified. Another result is that the algorithms perform well in their respective datasets and also in the datasets provided with the other packages. This is not the case in the new datasets were the performance are many times worse also with parameters tuned ad hoc.

Balaguer, Carpin, and Balakirsky [7] utilize the USARSim robot simulator and a real robot platform for comparing different open source SLAM approaches. They demonstrated that maps resulting from processing simulator data are very close to those resulting from real robot data. Hence, they concluded that the simulator engine could be used for systematically benchmarking different approaches to SLAM. However, it has also been shown that noise is often, but not always, Gaussian in the SLAM context [39]. Gaussian noise, however, is typically used in most simulation systems. In addition to that, [7] do not provide a quantitative measure for comparing generated maps with ground truth. As with many other approaches, their comparisons were carried out by visual inspection. In this paper it is also pointed out that there is no rigorous approach in comparing different performance in the robot community and that there is a lack of shared code: it is still too often observed that when a new project is started, certain tasks are coded again from scratch, rather than relying on existing libraries.

Sharing the data and the code is not so common in the community, but, as stated in [5], comparison concerns the capability not only of knowing what has been already done in the past, but also of comparing the new results with the old ones. This means that all the results have to be presented, good ones and bad ones, and they have to be well documented. It is not always the case in the field of SLAM evaluation where happens frequently that the parameters of the algorithms and the hardware are not specified. The main settings in which SLAM algorithms are evaluated are articles where a new metric is presented, comparisons on public datasets, or competitions like RoboCup. When a evaluation metric is proposed most of the times the environment is not particularly stressful for the system, like only one map not representing a real environment of use. Also some settings are selected because it is recognized (a posteriori) that they have brought to satisfactory results.
The competition settings are likely to generate additional noise due to differing hardware and software, furthermore the environments are not always real but designed ad hoc.
The public datasets are the best way to assess a good comparison and thanks to the ease of use and availability their use is common. For example Radish [19] and Rawseeds [3] are collections of recorded robot real-world runs. The drawback of these datasets is their poor variety: most are collected inside university campuses, university parking structures, research labs, researchers' offices, and university cafeterias. They represent only a very narrow and specific class of environments [5]. For example, a robot that has to

work in a complex environment, such as an old hospital, may face problems that are difficult to detect with tests performed in a modern office building because of the differences in the architecture. This can influence the selection of landmarks and hence bad landmarks can be chosen in environments with objects and obstacles that the algorithm has never seen before.

These limitations in achieving comparison and generalization don't allow to predict the performance of SLAM algorithms in new environments due to all the factors influencing map quality stated above. So their applicability in settings different from those in which they have been developed is not guaranteed. From these conditions follows this thesis. To the best of our knowledge, there are no works in literature that try to predict the performance of SLAM algorithms in a new environment, but an interesting work is [47]. The aim of the system described in the [47] is to predict the performance of autonomous robots measuring the complexity of the environment. The hypothesis is that by modeling the complexity of the environment in which a system will operate, it is possible to predict the system's performance. However, with this approach, the overall system, and not SLAM algorithms, are evaluated. To predict the quality of the map built by SLAM algorithms thus can be useful understand the structure of the environment, this can be related to semantic mapping, i.e., associate a high-level human-understandable label (like "office" or "corridor") to a portion of an environment [29]. Know how well SLAM algorithms operate in a type of environment can be used to predict the performance in another environment of the same type. But this need lots of data to generalize the results.

All the studies described so far provide a background that has been of inspiration for this thesis. The SLAM algorithm we evaluated is the ROS implementation of GMapping [1]. In almost all the approaches found in the literature a human intervention is performed to check the reliability of the map or to build the ground truth, this makes an automatization of the evaluation process impossible. In this work we try to completely remove the human to have the possibility to run several evaluations of the map quality in a fast and automated way. We use the metric by [24] to evaluate maps but we modify the generation of the ground truth to make it completely automatic. Using simulations, in particular the Stage simulator [13], we test GMapping with the same settings in many different environments to achieve generalization and have a better understanding of its ability to build maps.

Chapter 3

Problem setting

The objective of this thesis is to predict the quality of the map that a SLAM algorithm will build in an environment, before actually exploring it. In order to tackle this problem we think that the availability of an extended set of measured performance of a SLAM algorithm in different environments can allow us to generalize and thus predict its performance (i.e., the quality of map and localization output) in new environments, based on the environment's characteristics. The main problem faced in this thesis is how to evaluate, without any human interaction, the quality of the maps produced by a SLAM algorithm run on a dataset composed of buildings of different structure. As explained in Chapter 2 there is not a standard evaluation methodology of the quality of a map in a quantitative way. We chose to use the metric presented in [24] because it is considered a reliable evaluation methodology when the ground truth path is available [36]. In our solution we extensively evaluate the ROS implementation of GMapping by automatizing the generation of data of several maps and data representing the maps' quality as produced by GMapping. Generating this data with real robots is an expensive task, but the use of simulations is an accepted method in the community to test new systems [5]. Simulations have many advantages in our case. First of all, we have easily access to the ground truth path of the robot. Secondly we can repeat the same experiment with the same conditions (same starting point, exactly same environment, same sensors) multiple times without any further procedures. Finally, since we need lot of data, we can run simulations without human supervision for long time.

3.1 Evaluation method

We need a reliable way to asses the quality of the map produced by the SLAM algorithm. The output of a SLAM algorithm is a series of estimated poses of the robot, and a map of the environment. The evaluation approach we used, presented in Kümmerle et al. [24], relies on the evaluation of the displacement between the estimated poses and the corresponding ground truth poses.

The assumption it makes is that if the localization is good, then the mapping is also good. Let $x_{1:T}$ be the poses of the robot, estimated by the SLAM algorithm from time step 1 to T. Let $x^*_{1:T}$ be the ground truth poses of the robot, ideally the true locations. In planar environments a pose of a robot is summarized by three variables, these are its two-dimensional planar coordinates, x and y, relative to an external coordinate frame, along with its angular orientation θ. The pose at time t is described by the vector:

$$x_t = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}$$

A straightforward error metric could be defined as

$$\epsilon(x_{1:T}) = \sum_{t=1}^{T} (x_t \ominus x^*_t)^2 \tag{3.1}$$

where $x_t \ominus x^*_t$ is the relative transformation that moves the pose x^*_t onto x_t. Let $\delta_{i,j} = x_j \ominus x_i$ and accordingly $\delta^*_{i,j} = x^*_j \ominus x^*_i$. The mathematical operation that computes $\delta_{i,j}$ is the roto-translation that moves x_i to x_j. A roto-translation is the rigid body transformation that moves the pose at time i to the pose at time j. Since we are in an absolute reference frame from linear algebra we have:

$$\delta_{i,j} = M_i^{-1} \times M_j \tag{3.2}$$

where M_i^{-1} is the inverse of the matrix that moves the robot from the origin of the frame to the pose x_i, and M_j is the matrix that moves it from the origin to x_j, as shown in Figure 3.1.

M_j thus is composed by a rotation and then a translation, that in matrix notation is:

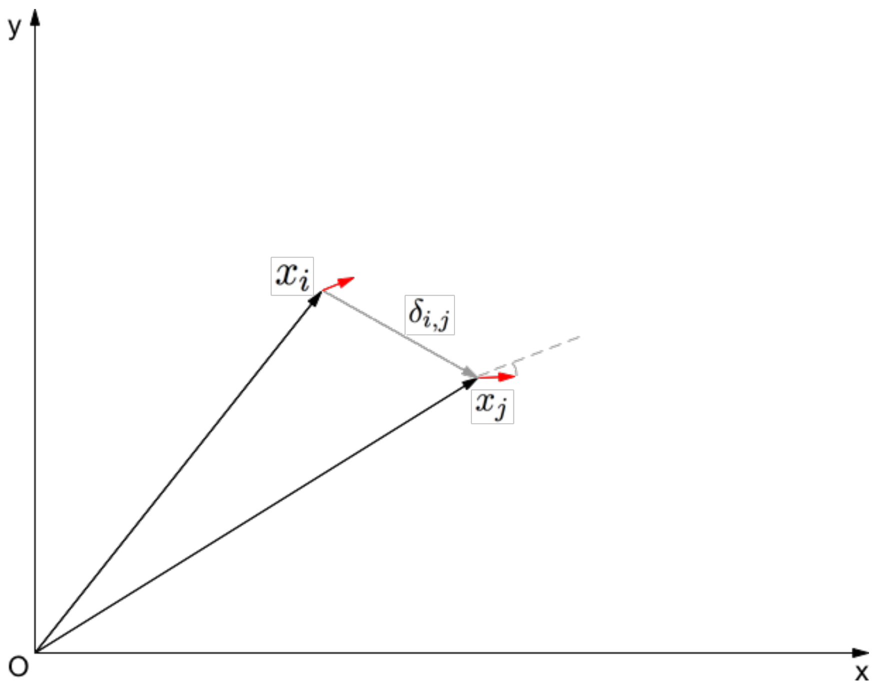


Figure 3.1: Representation of $\delta_{i,j} = x_j \ominus x_i$. The poses considered are x_j and x_i and the grey arrow represents $\delta_{i,j}$. The black arrows are the translations needed to move from the origin to the poses. The red arrows indicate the orientation of the robot. The dashed line is the orientation of pose x_i compared to x_j. The green line is the series of poses of the robot, namely the trajectory

$$M_j = \begin{bmatrix} 1 & 0 & 0 & x_j \\ 0 & 1 & 0 & y_j \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} cos(\theta_j) & -sin(\theta_j) & 0 & 0 \\ sin(\theta_j) & cos(\theta_j) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Equation 3.1 can be rewritten as

$$\epsilon(x_{1:T}) = \sum_{t=1}^{T}((x_1 \oplus \delta_{1,2} \oplus ... \oplus \delta_{t-1,t}) \ominus (x_1^* \oplus \delta_{1,2}^* \oplus ... \oplus \delta_{t-1,t}^*))^2 \tag{3.3}$$

where $\oplus$ is the inverse of $\ominus$. This means that every pose x_t is composed of the summation of the roto-translations from the origin (pose x_1) to pose x_t. Assume the robot makes a translational error of e during the first motion $\delta_{1,2} = \delta_{1,2}^* + e$, but perfectly estimates all other points in time $\delta_{t,t+1} = \delta_{t,t+1}^*$ for $t > 1$. Thus, the error according to (3.3), will be $T \cdot e$, since $\delta_{1,2}$ is contained in every pose estimate for $t > 1$. However, if we estimate the

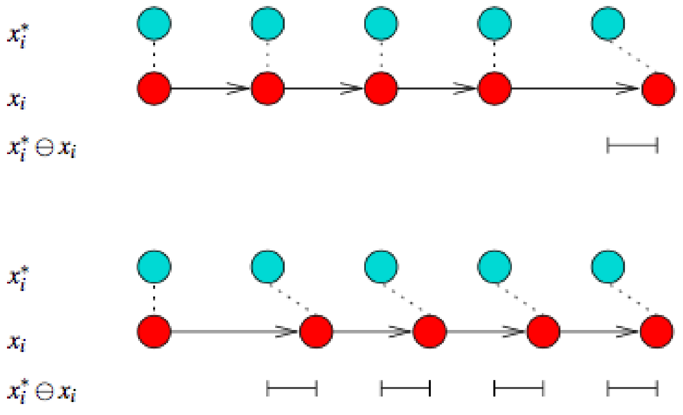


Figure 3.2: This figure illustrates a simple example where the metric in (3.1) fails. The light blue circles show the reference positions of the robot x_i^* while the dark red circles show the estimated positions of the robot x_i. The correspondence between the estimated locations and the ground truth is shown with dashed lines, and the direction of motion of the robot is highlighted with arrows. In the situation shown in the upper part, the robot makes a small mistake at the end of the path. This results in a small error. Conversely, in the situation illustrated on the bottom part of the figure the robot makes a small error of the same entity, but at the beginning of the travel, thus resulting in a much bigger global error. Image from [24]

trajectory backwards starting from x_T to x_1, or alternatively by shifting the whole map by e, we obtain an error of e only. To illustrate this, consider the example in Figure 3.2 in which a robot travels along a straight line.

Based on this experience, [24] proposes a measure that considers the deformation energy that is needed to transfer the estimate into the ground truth. This can be done by considering the nodes as masses and connections between them as springs. Thus this error metric is based on the relative displacement between robot poses. Instead of comparing x to x^* (in the global reference frame), the operation is based on δ and δ^* as

$$\epsilon(\delta) = \frac{1}{N} \sum_{i,j} trans(\delta_{i,j} \ominus \delta_{i,j}^*)^2 + rot(\delta_{i,j} \ominus \delta_{i,j}^*)^2 \tag{3.4}$$

where N is the number of relative displacements considered, the *relations* $\delta_{i,j}$, while $trans()$ and $rot()$ are used to separate, and weight, the translational and rotational components. Since every relation is independent from the others, the ϵ (or transformation energy), of the example in Fig 3.2, will be consistently estimated as the single rotational error, and the error will not change regardless of where the error occurs in the space, or in which order the data is processed. The error metric, however, leaves open which

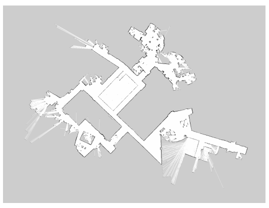
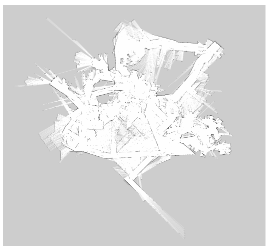
Output Map	Total Error	A Error	B Error	% B on total error
	226,950712	162,11416	64,836552	28,63
	1855,651358	126,020906	1729,630452	93,21

Table 3.1: In the table, 2 different runs of the same dataset are presented with the respective translational total errors. In the first row the map is coherent with the ground truth, while in the second it is completely different. It can be seen that the B relations have a large impact on the error when the map is bad

relations $\delta_{i,j}$ are included in the summation in Equation (3.4). Selecting the more relevant relations is the problem of the metric of Kümmerle et al..

In the paper [24] the relations are manually selected by the authors: besides the definition of the metric, Kümmerle et al. provide 5 different datasets with the corresponding sets of relations. These datasets S are composed by two types of relations: the first type of relations A are intended to measure the local consistency of the trajectory, comparing two poses i and j that are temporally close, considering as temporally close two poses with under a second of difference. The second type of relations B are intended to measure the topological consistency with long temporal displacements, that can be from time 0 to the end of the *run*. The term “run” indicates a single exploration of an environment by a real or simulated robot. The length of the runs reported in [24] is between 20 minutes and an hour and a half. B relations allow to verify if the SLAM algorithm has built a topologically consistent map.

There are some problems with the relations provided, in fact, after we analyzed the datasets, we find that the A relations barely affect the global error when bad maps are produced. From now on when a map is referenced as *good* (or *bad*) it means that is visually similar (or not) to the ground truth map. The reason why the A relations affect in a little measure the error is because the displacement of the predicted trajectory, with respect to the ground truth trajectory $\delta_{i,j} \ominus \delta^*_{i,j}$, with i, j close temporally, usually has not a high value. E.g., if the difference between i and j is 0.5 seconds, then, due to the relatively low velocity of the robot, the translational displacement can be in the order of centimeters and the rotational displacement in the order of few degrees. Furthermore, the error is normalized over the number of relations N. Instead, we found that the major contribution to the error is given by the second type of relations, namely the B relations, that, with large displacements, can have a high value and thus signal the errors of the SLAM algorithm. For example, let's assume we map a corridor and there is only a rotational error in the middle of it, so the first part and the last part of the corridor is right. Along the whole length of the corridor there are the A relations and there is a B relation that starts at the beginning of a corridor and ends at the end of it. Only few of the A relations will catch the error, and since the error can happen in more than one second, it will be divided in little pieces and normalized on the number of relations. Instead, the only B relation will catch the whole error because the end point produced by the SLAM algorithm is far away from the ground truth end point. Table 3.1 shows how the error of the A relations doesn't increases, actually in this case decreases, when the map is bad. However, the error due to the relations of type B is high.

Unluckily the relations are build by a human that knows the structure of the environment, and in the paper they state that "for a standard dataset with 1700 relations, it took an unexperienced user approximately four hours to extract the relative translations that then served as the input to the error calculation". This is unfeasible for our scope. But, unlike in [24], we simulate the robots and thus we can have ground truth data. Since we have the ground truth path of the robot we can easily generate the A relations like this: we take every 0.5 seconds the pose of the robot and we create a relation between t and $t + 0.5$, so $\delta^*_{t,t+0.5} = x^*_{t+0.5} \ominus x^*_t$. We can build a set of A relations on the whole path, but as we have seen they most likely don't contain enough information to detect the error. We will see it clearer in Chapter 4. Hence, in our work, we focused on an automated way to generate B relations.

3.2 Factors influencing SLAM algorithms

In the last section we saw that the main problem is how to evaluate the quality of a map in a automated way, but there are other factors to consider to have a reliable evaluation of the map. In this section we analyze a set of factors, pointed out in Chapter 2, that influence the quality of the map produced by the SLAM algorithm. In our analysis, we want to focus our attention on only one of such factors that influences the performance of GMapping, namely the environment, so we kept fixed as possible the others. The factors are:

- Algorithm;
- Robot path;
- Computing power;
- Sensors;
- Environment.

The SLAM algorithm that we use is GMapping, it is kept the same with the same settings for all the runs. The parameters settings for Gmapping that we used are for the most part the default ones of the ROS implementation. The ones we changed are: the particles and the angular update, we use 40 particles, instead of 30, and 0.25 radians to update the map, instead of 0.5 radians, to have a better performance in the building of the map. The other parameter changed is maxUrange, that is the max range of the laser sensor in meters, and we set it to 30.

Another factor that has to be considered is the robot path. For our work it can't be controlled manually, because, since we have to automatize the generation of data, and we can't teleoperate the robot in every environment. A solution can be to register the series of commands needed to explore all the environment, then run the SLAM algorithm on that path. However, this is not a flexible solution, because for every new environment inserted in the system a manual exploration have to be performed. This lead us to the implementation of an autonomous navigation and exploration system. It resulted, as we will see in Chapter 4, that tuning a good exploration method was a major problem, since the navigation relies on the localization provided by the SLAM algorithm. Further, the navigation system can crash the robot against a wall and make the SLAM algorithm fail. Differentiating the navigation errors from the ones of the SLAM algorithm is thus necessary.

The computing power is another factor that influences the runs. We run simulations on different machines to generate more data, and the results were slightly different. The performances of GMapping are affected by the computing load if it saturates the CPU. This, however, is not a problem that happens frequently since if the system is launched on a machine with a Core2 Quad Q9400@2.66Ghz and 4GB of memory the percentage of CPU used on average is around 40%.

Sensors are kept the same in every simulation. The Hokuyo laser ranger used in the simulations has a range from 0 to 30 meters, a field of view of 270 degrees and a definition of two samples for every degree. The odometry information is provided by the simulator and it has a error provided by a uniform random distribution. The error is set for x, y, and θ respectively as 0.01, 0.01, and 0.02. For each value, if the value specified is E, the actual measure is chosen at random in the range -E/2 to +E/2.

The last factor is the type of environment mapped. In our work we want to understand how well GMapping performs in different environments, so we need to test it in many environments with various structure. We use simulations, so we can map different kinds of buildings to obtain a good generalization easily, since we only need a 2D floorplan, in an image format, to have an explorable environment. The buildings explored have different number of rooms, dimensions, and presence (or not) of furnitures. We used two distinct datasets from universities and schools.

All these problems are addressed in the next chapters, where the proposed solutions and their implementation are presented.

Chapter 4

Data generation

In this chapter we present the solutions proposed to automatize the generation of the data. The first problem is to generate the ground truth relations for the evaluation metric in an automated way. We begin analyzing the data sets provided by Kümmerle et al.. There are 5 data sets, each of these is the log of a run in university building. The data sets contain the odometry and laser measurements of a real robot. Every run has its corresponding set of relations. In the experiments presented in [24] different SLAM algorithms are tested with these log files, and the quality of the path produced is measured. Since we have to generate automatically the relations files, we investigate the files provided. This is a piece of an original relations file:

```
90.239599 90.719599 0.422330 −0.000750 0.000000 0.000000 0.000000 0.017260
90.719599 91.321399 0.213430 −0.014750 0.000000 0.000000 0.000000 −0.405780
94.399600 94.759600 0.277100 0.063830 0.000000 0.000000 0.000000 0.097740
94.759600 95.239900 0.295310 0.033520 0.000000 0.000000 0.000000 0.059840
95.749899 96.469599 0.222950 0.050560 0.000000 0.000000 0.000000 0.125440
```

these values, from their documentation, indicate, respectively:

```
timestamp1 timestamp2 x y z roll pitch yaw
```

A single line corresponds to a relation: $\delta^*_{i,j} = x^*_j \ominus x^*_i$. The two poses i and j are identified by the two timestamps: `timestamp1`, t_1, and `timestamp2`, t_2. x, y, and yaw represents the relative roto-translation needed to move the pose from t_1 to t_2. x and y are provided in meters and yaw in radians. For example, to go from the pose at time 90.239599 to the pose at time 90.719599, first it has to be performed a rotation of 0.017260 radians, and then a translation of 0.422330 meters on the x axis and of -0.000750 on the y axis. Because we are in 2D environments z, $roll$, and $pitch$ have always value 0. The whole of these relations represents the ground truth. The evaluator software uses this file as follows: it takes as input this data set of relations and the estimated trajectory built by the SLAM algorithm, for every line of the relations file, it computes the roto-translation from t_1 to t_2 between the poses estimated by the SLAM algorithm, identified by x_1, y_1 and yaw_1. Then the evaluator determines the difference between the computed values, x_1, y_1 and yaw_1, and the corresponding values of the relations file, i.e., x, y, and yaw. The sum of these differences is the total error, and will be divided by N that is the number of relations. As mentioned in Chapter 3, the relations can be divided into A and B. The A relations are characterized by a short time difference from t_1 to t_2 and in all the data sets are the major part. The B relations are fewer in number, inserted by a human, and generally at the end of the file. After an experimental evaluation, that is presented in Chapter 6, we have seen that the A relations are not important for the detection of the quality of the map. Instead, the B relations reveal important informations.

4.1 Generation of B relations

Thus we need to build the B relations. We have to find a way to detect the errors on the whole length of the trajectory in an uniform way. The best way that we found is to generate a random set of $\delta_{i,j}$, with no bounds on i and j and big enough. After some experiments this method to generate the relations showed a good correlation between the error computed by the evaluator and the visual quality of the maps produced.

The first thing to do is define what is the value of N, the number of relations in the set. From statistics we know that, to generate a relevant sample of the population, we need the standard deviation σ of it. In our case we don't know it, so we sample 500 test relations to compute σ and then compute the number N of the population. We take 500 as number of test relations because we have tried different numbers experimentally, and we saw that after 500 the standard deviation begins to stabilize. The equation that we use to compute the population size is:

$$N = \left(\frac{z_{\frac{a}{2}}^{\sigma}}{E} \right)^2 \tag{4.1}$$

where:

- $z_{\frac{a}{2}}$ is known as the critical value, the positive z value that is at the vertical boundary for the area of $a/2$ in the right tail of the standard normal distribution. To have a 95% degree confidence this value is 1.96;

- σ is the sample standard deviation;
- E is the margin of error, the maximum difference between the observed sample mean and the true value of the population mean, our setting for the experiments is 0.05;
- N is the population size.

Once that we have the population size N we randomly take 2 poses from the ground truth trajectory N times. For each time, we compute the roto-translation, x, y, and yaw, needed to move the first pose, at time t_1, on the second pose, at time t_2. These values go to compose relations the file. In Table 4.1 an example of the error from automatic generated B relations is shown. Visually inspecting the maps, it can be seen that in the ones that are really different from the ground truth the mean error is in the order of tens of meters, in maps with small differences the mean error is in the order of few meters and in maps very similar to the ground truth the error is lower than 1 meter. Now we have a reliable way to generate the ground truth file that the metric evaluator uses, so we can evaluate the quality of the maps. Nevertheless, we can't explore autonomously the environments. We have to set up a stable navigation system that can handle different types of environments.

4.2 Autonomous exploration

We first tried a navigation package named nav2d [21] that allows to control a mobile robot within a planar environment. The main features are a purely reactive obstacle avoidance and a simple path planner. However, we found that this package works well only in simple environments (4-5 rooms without furniture) while in more cluttered or vast maps it gets easily stuck or lost.

Our choice went to the Navigation Stack of ROS. This is a set of processes that take information from odometry and sensor streams and output velocity commands to send to a mobile base, i.e., the robot. The Navigation Stack needs to be configured for the shape and dynamics of the robot to perform well. It uses two different planners for local and global path: the local planner uses Dynamic Window Approach (DWA) algorithm. DWA is proposed by Dieter Fox et al. in [12]. According to this paper, the goal of DWA is to produce (vx, vy, ω), that are the translational and rotational

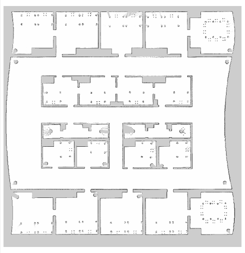
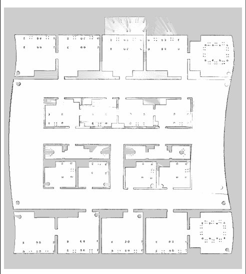
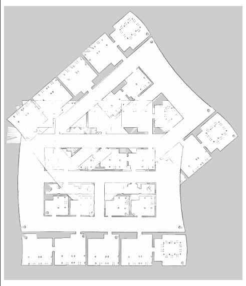
Output Map			
B Mean Error	T = 0.396124m R = 0.019070rad	T = 2.500798m R = 0.021366rad	T = 14.297833m R = 0.396734rad
B Std dev	T = 0.778331m R = 0.027167rad	T = 2.563980m R = 0.027465rad	T = 17.849961m R = 0.433986rad

Table 4.1: The table shows the errors provided by the B relations on three different runs of the same environment. The first run is visually right, the second has some errors in the rooms at the top, and the third has big mapping errors

velocity commands, which represents a trajectory that is optimal for the robot's location. ROS Wiki provides a summary of its implementation of this algorithm:

1. Discretely sample in the robot's control space (vx, vy, ω);
2. For each sampled velocity perform forward simulation from the robot's current state to predict what would happen if the sampled velocity were applied for some (short) period of time;
3. Evaluate (score) each trajectory resulting from the forward simulation, using a metric that incorporates characteristics such as: proximity to obstacles, proximity to the goal, proximity to the global path, and speed;
4. Discard illegal trajectories (those that collide with obstacles);
5. Pick the highest-scoring trajectory and send the associated velocity to the mobile base;
6. Repeat.

The first step is to sample velocities (vx, vy, ω) in the velocity space within the *dynamic window*. DWA will only consider velocities within a dynamic window, which is defined to be the set of velocity that is reachable within the next time interval given the current translational and rotational velocities and accelerations. The second and third steps are to simulate and evaluate the velocities using the objective function, which outputs trajectory score. The fourth step is basically obliterating velocities (i.e., kill off bad trajectories) that are not admissible. The fourth and fifth steps are easy to understand: take the current best velocity option and recompute.

The global planner of the Navigation Stack that we use is *navfn*, that provides a fast interpolated navigation function that can be used to create plans for a mobile base. The planner operates on a costmap to find a minimum cost plan from a start point to an end point in a grid. The navigation function is computed with Dijkstra's algorithm.

The Navigation Stack tuning will be described in more detail in Chapter 6. In our experiments it has shown good performance, but incomplete explorations are not completely avoided. Depending on the complexity of the environment, clutter, and path choice, the robot can get stuck.

For automatization, we need to know when an exploration is completed to start the next one. An exploration is considered completed when the environment is fully mapped. In our system there are three conditions that stop the exploration: the first one is to check if the Navigation Stack doesn't find new *frontiers* to explore, and so it considers the environment completely explored. The frontiers are points, or clusters of points, considered free or unexplored on the edge of the explored environment. The Navigation Stack keeps a list of all the frontiers and scores each on the basis of a function that considers the path that the robot needs to travel to reach the frontier. When the Navigation Stack doesn't find new frontiers is not always because the environment is completely explored, because the frontiers that it finds depend on it's settings and on the map. It can happen that an exploration is stopped early, but it can also happen the opposite, that is, new frontiers are found even if the environment is completely explored. Hence, the exploration can go on for much more time than needed. Another case that makes the first condition fail is if the robot gets stuck. The Navigation Stack will try to reach the goal forever, and so the exploration will never be stopped. In these situations the second condition triggers. Every time t_i the grid map built by the SLAM algorithm is taken and saved as a image, the gap between t_i and t_{i+1} is a parameter g. Starting from t_2, the last two grid maps saved, t_i and t_{i-1}, are compared pixel per pixel with Mean Squared Error (MSE), that is the sum of the squared differences between the pixels of the two images. If the difference is under a threshold e the exploration is stopped. The third condition stops the exploration after a time limit m. If the two previous conditions are not activated, most likely something is gone wrong with the exploration, or the environment is very big. Every condition have a parameter to be set: g, e, and m. In our experiments g was between 3 and 5 minutes, e under 5, and m between 1 hour and 2 hours.

4.3 Data sets

We used two different data sets for the exploration of the environments, one is the data set used in the paper [10], composed by 20 floor plans with and without furniture, hence 40 floor plans. They represent university buildings. The other one is a data set of 20 floor plans of schools without furniture. The environments are of various shapes, dimensions, and clutter, in order to test GMapping in many different situations. In the university data set, since we have also the furniture version of every environment, we can compare the performance of GMapping in the same environment with clear rooms or with rooms with clutter. The furniture present is for the most part composed of chairs, tables and desks, and the robot can go through the legs of the furniture because of its size. The schools data set instead has no furniture but different structures of the buildings. They have on average more rooms (30.1 against 26.1) since they have many classrooms. In many buildings there is a big room, that is the hall or the gym, that in the other data set is rarely present. In Figure 4.1 there is a sample of the environments of the first data set, while in Figure 4.2 there are all the school buildings.

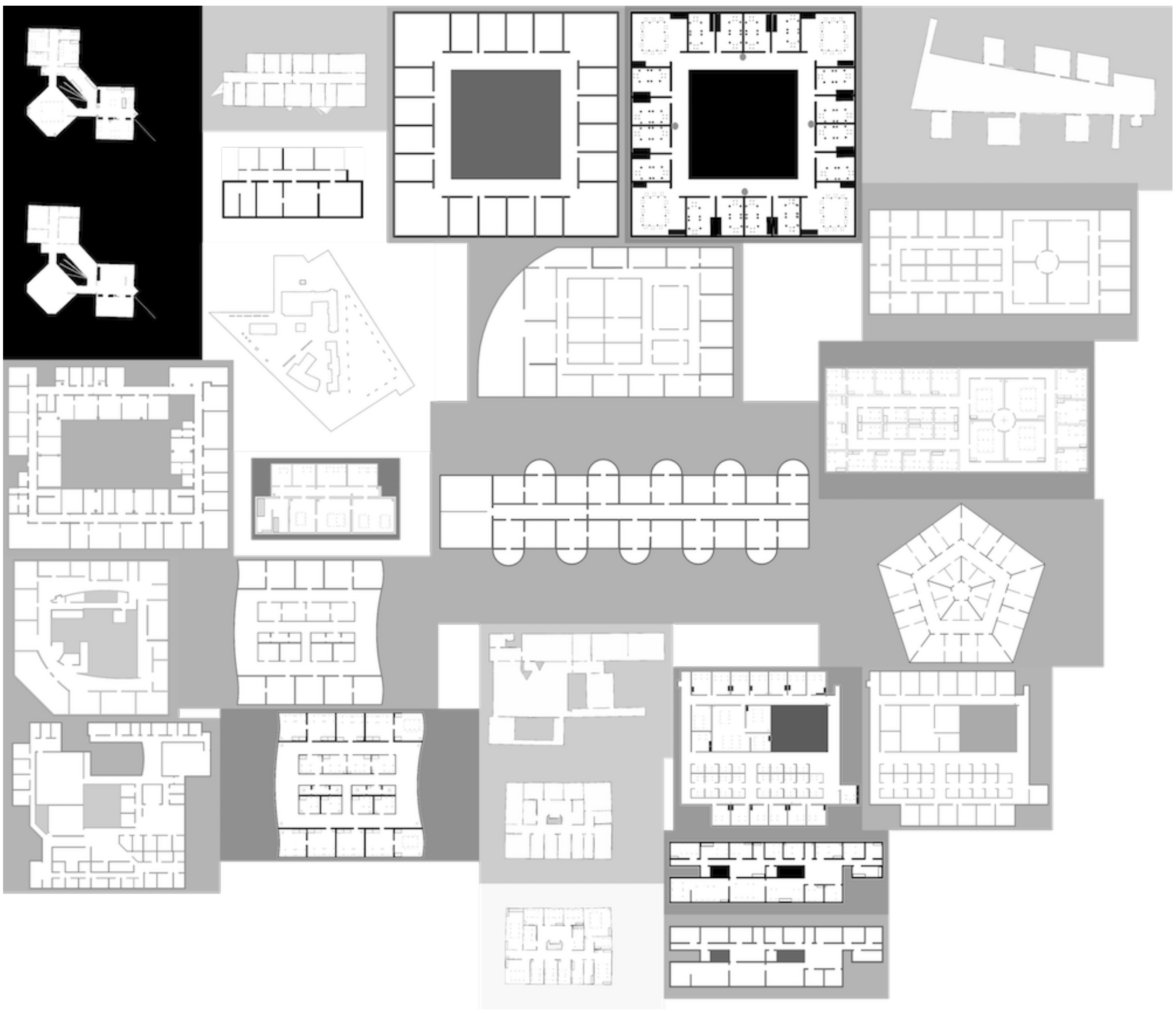


Figure 4.1: A sample of the environments contained in the first data set used, with furniture and without furniture. It can be seen the variety of the shape and structure

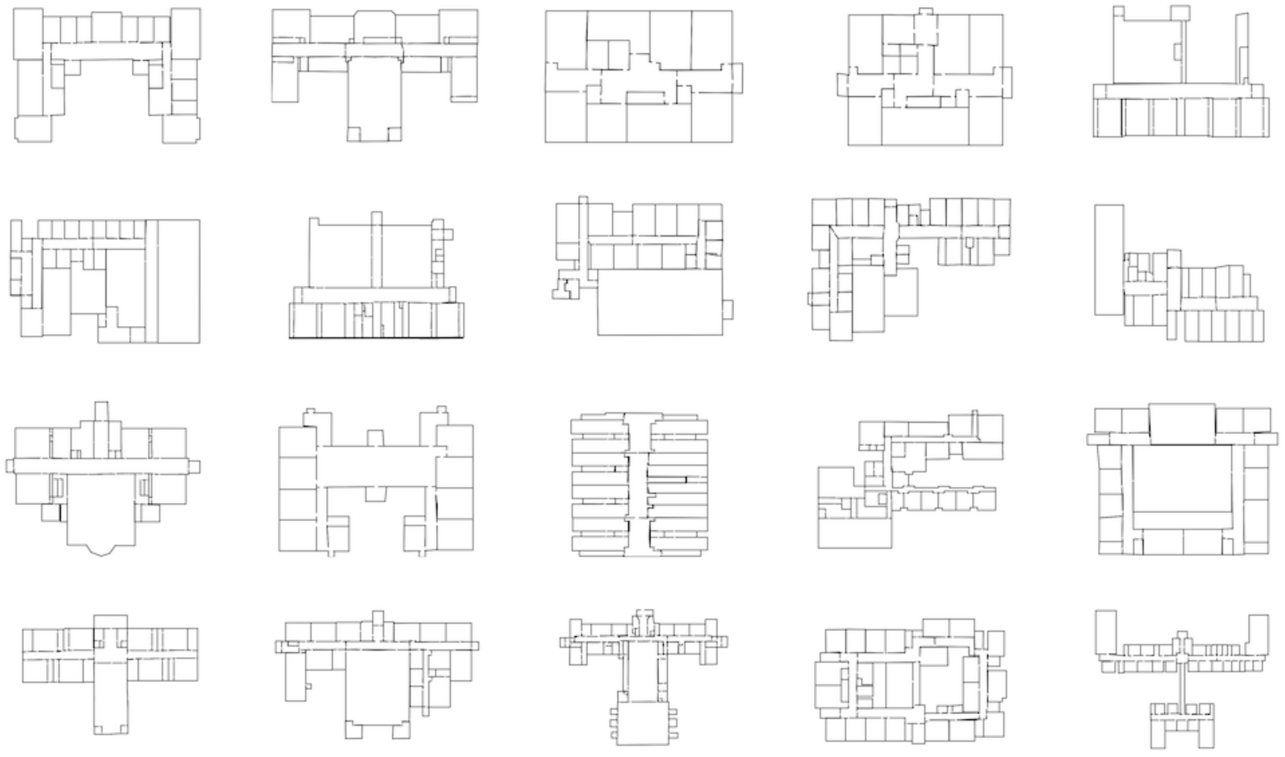


Figure 4.2: The environments contained in the second data set used. Schools have a different type of structure with respect to university offices

Chapter 5

System architecture

This chapter presents the software architecture designed to generate the data described in Chapter 4. In Section 5.1 we give an overview of the main characteristics of the Robot Operating System (ROS), on which we build and test our system. Then in Section 5.2 we explain the ROS nodes that compose the system to automatically explore an environment and collect the data, and, at last, in Section 5.3 we explain the Python script that launches multiple executions of the aforementioned system, builds the ground truth data, and computes the errors about the quality of the maps.

5.1 ROS: Robot Operating System

ROS is a distributed, flexible framework for writing robot software that offers a message exchanging interface providing inter-process communication, behaving like a middleware. As described in [33], the main components of the ROS middleware are nodes, messages, topics, and services.
Nodes are processes that perform computations. As ROS is designed to be modular, a system is typically composed by several nodes, that together compose a graph. For example, one node controls a laser range scanner, one node controls the robot's wheel motors, one node performs localization, one node performs path planning, one node provides a graphical view of the system, and so on. The use of nodes in ROS provides several benefits to the overall system. There is additional fault tolerance, as crashes are isolated to individual nodes. Code complexity is reduced in comparison to monolithic systems. Implementation details are also well hidden as the nodes expose a minimal API to the rest of the graph and alternative implementations, even in other programming languages, can easily be substituted. Nodes communicate with each other by passing messages in two different ways: through

topics or through services. The difference between the two ways of handling the message exchange is the following: when dealing with topics, a node sends a message by publishing it to the topic of interest, while a node that is interested in receiving the same kind of data subscribes to the same topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. On the contrary, services are defined by a pair of strictly typed messages, one for the request and one for the answer, and only one node can advertise a service. The introduction of the service mechanism overcomes the problem of synchronous transactions, not taken care by the publish-subscribe paradigm. The consequence of this architecture is that, once a node subscribes to a topic, it sees all the messages arriving on that topic, also those not directly meant for it. In both cases, the communication pattern used by ROS is a publish-subscribe one. In order for ROS nodes to be able to communicate, a ROS network must have its roscore running. The roscore is a collection of nodes and programs that are pre-requisites of a ROS-based system: it includes a ROS master, that provides naming and registration services to the rest of the nodes and allows them to locate each other, a ROS parameter server, initialized by the master and used to retrieve and store parameter values, and a rosout logging node. For our experiments we used the ROS distribution Kinetic Kame.

5.2 ROS nodes for exploration

The launch file starts the nodes in Figure 5.1. They perform the exploration of an environment and write the files for the subsequent evaluation: the ground truth trajectory from the simulator and the estimated one from the SLAM algorithm. The nodes are:

- *Stage*, the simulator;
- *Mapper*, the SLAM algorithm, that is GMapping in our case;
- *recorder*, that records ground truth data from Stage;
- *RVIZ*, that is the 3D visualization environment present in ROS. It shows the map built by GMapping, the current path of the robot, obstacle data, and sensor data;
- *listener*, that writes to a file the estimated poses of the robot;
- *laser_noise*, that creates some Gaussian noise on the simulated laser ranger, it is optional since the noise can be turned off;

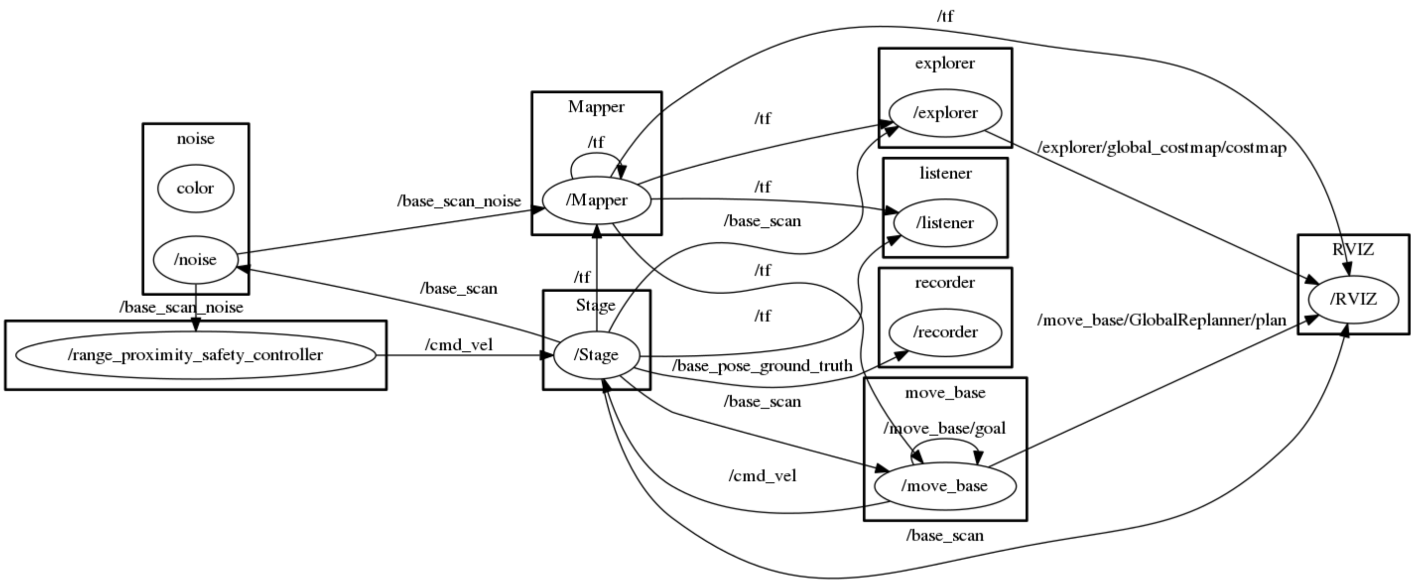


Figure 5.1: The ROS graph of the nodes involved in our system

- *move_base*, that sends the move commands to the robot, it is part of the Navigation Stack;
- *explorer*, that finds the goals to reach, also part of the Navigation Stack;
- *range_proximity_safety_controller*, that keeps the robot at a safe distance from the obstacles.

5.2.1 Stage

For simulation we have used Stage, a 2D simulator already implemented in ROS. It is described as a lightweight, highly configurable robot simulator that supports large populations of robots. Together with Stage there is Player, that is a robot device server that provides network transparent robot control and offers a combination of transparency, flexibility, and speed. Stage provides robots with different kinds of sensors operating in a two-dimensional bitmapped environment. The devices are accessed through Player, as if they were real hardware. Stage aims to be efficient and configurable, it can simulate tens or hundreds of robots on a desktop PC, and with behaviours similar to those of real robots. Stage runs on many UNIX-like platforms, and is released as Free Software under the GNU General Public License [4]. It is maintained at: http://playerstage.sf.net. Users have found

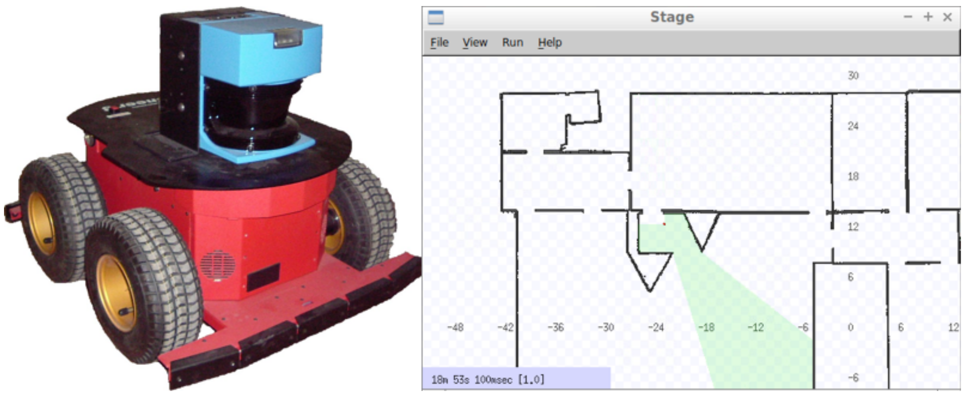


Figure 5.2: The P3AT real robot on the left and the Stage simulation on the right

that clients developed using Stage will work with little or no modification with the real robots and vice versa [30], that is a proof of its reliability.
In practice it provides a virtual world, shown in Figure 5.2 on the right, represented by a image that it takes as input, populated by a mobile robot, identified by a numerical id and equipped with sensors and actuators. The image of the environment that Stage uses is a simple matrix of 0*s* and 1*s*, with the 0*s* representing areas of the environment that are free from obstacles, and 1*s* representing the areas in which obstacles are present. We simulated a Pioneer 3-AT, shown in Figure 5.2 on the left, that is a highly versatile four wheel differential drive robotic platform [22], and the sensor that we mounted in the simulations is a Hokuyo UTM-30LX laser sensor with field of view of 270° and range up to 30 meters [2]. The input the system needs is a *world* file, that is a description of the world that Stage must simulate. It describes robots, sensors, actuators, moveable and immovable objects. The world file can also be used to control many aspects of the simulation engine, such as the scale, its speed, and fidelity.

5.2.2 Mapper

The Mapper node is the node that is concerned with the building of the map, namely, the SLAM algorithm. In our work we used GMapping, that is a SLAM algorithm based on a particle filter, in which each particle represents an individual map of the environment. The GMapping implementation available on ROS is a wrapper of the OpenSlam version [38], that is coded in C++. Basically, the algorithm requires time-stamped odometry (i.e., Transform messages) and time-stamped readings from the laser topics (i.e., LaserScan messages). It creates a 2D occupancy grid map from laser and pose data collected by a mobile robot subscribing to the topics:

```
tf (tf/tfMessage)
```

```
scan (sensor_msgs/LaserScan)
```

In ROS the laser messages are encoded in the LaserScan type, which contains all the information of a single scan from a planar laser range-finder. The definition of the message from docs.ros.org is:

```
LaserScan:
        std_msgs/Header header
        float32 angle_min
        float32 angle_max
        float32 angle_increment
        float32 time_increment
        float32 scan_time
        float32 range_min
        float32 range_max
        float32[] ranges
        float32[] intensities
```

The measurements are encoded in the `ranges` vector, while in the `header` there are the timestamp of the acquisition time of the first ray in the scan, and the ID of the scan. `angle_min` and `angle_max` define the field of view of the sensor, `angle_increment` its resolution, and `range_min` and `range_max` the minimum and maximum values of the readings.
`time_increment`, `scan_time`, and intensities are not relevant in our case. Odometry is contained in the tfMessage type, that is a vector of Transform-Stamped messages. These are Transform messages with ID and timestamp. The Transform messages represent the transform between two coordinate frames in free space. Transform is, in turn, composed by a Vector3 message, that represents the translation, and a Quaternion message, that represents the rotation. From docs.ros.org:

```
TransformStamped:
        ID
        Timestamp
        Transform:
                geometry_msgs/Vector3 translation:
                        float64 x
                        float64 y
                        float64 z
                geometry_msgs/Quaternion rotation:
                        float64 x
                        float64 y
```

```
                    float64 z
                    float64 w
```

The topics published by GMapping are:

```
map (nav_msgs/OccupancyGrid)
```

that is the map data, which is latched and updated periodically, and the correction of the pose of the robot in:

```
map → odom
```

the current estimate of the robot's pose within the map frame.
Since the SLAM algorithm is a node, it can be changed easily to build the map with a different algorithm, the only requirement is to publish the estimate of the robot's pose within the map frame.

5.2.3 recorder

All the ROS messages can be recorded in *bag* file format. We used them to log the run of the robot and build the relations file. The recorder node is the node that gets from Stage the ground truth pose every 0.1 seconds, in the global reference frame, and saves it in a .bag file. This will be used to extract the trajectory and compose the relation file.

5.2.4 RVIZ

RVIZ is a ROS tool to visualize the robot and its movements in the environment, the sensors data that it receives, and the occupancy grid built. As can be seen from the graph in Figure 5.1 it doesn't publish data to the other nodes, though we need it to see the robot behavior during the explorations and tune the Navigation Stack consequently.

5.2.5 listener

The listener node listens for the Transform messages published on

```
map → odom
```

that are the estimates of the robot's pose, which are then written into a file in a readable way for the evaluator. This is done using a CARMEN log file format, in which every row is composed as follows:

```
FLASER num_readings [range_readings] x y theta odom_x odom_y
        odom_theta timestamp
```

The data that the evaluator uses are composed by only the last 4 fields: odom_x, odom_y, odom_theta, and timestamp. These fields represent the pose estimated by GMapping in the absolute reference frame. Hence, the other fields are always 0. This is an example of a piece of file:

```
FLASER 0 0.0 0.0 0.0 −45.9153122597 15.4101963164 3.12809938018 4215.9
FLASER 0 0.0 0.0 0.0 −45.9153122597 15.4101963164 3.12809938018 4215.9
FLASER 0 0.0 0.0 0.0 −45.9254113403 15.4103325943 3.12809938018 4216.0
FLASER 0 0.0 0.0 0.0 −45.9254113403 15.4103325943 3.12809938018 4216.0
FLASER 0 0.0 0.0 0.0 −45.9355104208 15.4104688722 3.12809938018 4216.1
FLASER 0 0.0 0.0 0.0 −45.9355104208 15.4104688722 3.12809938018 4216.1
```

5.2.6 laser_noise

The laser noise node is a simple node that adds a Gaussian noise to the readings of the simulated laser sensor. It can be tuned to different levels of noise and it takes a parameter that is the standard deviation ("spread" or "width") of the Gaussian distribution.

The next three nodes automate the exploration of the environments, taking as input the odometry and the laser sensors information, and producing as output the velocity and direction commands to the robot to explore the environment. The first two are part of the Navigation Stack present in ROS and the last one is a safety controller that integrates the Navigation Stack.

5.2.7 move_base

The move_base node provides an implementation of an action that, given a goal in the world, will attempt to reach it with a mobile base. This node links together a global and a local planner to accomplish its global navigation task. As said in Chapter 4, we used *navfn* for the global planner and *DWA* for the local planner. The move_base node also maintains two costmaps: a costmap is a occupancy grid map with a value in every cell that represents if there is a obstacle with three values: free, occupied, or unknown. In ROS, the costmap is composed of static map layer, obstacle map layer, and inflation layer. The static map layer directly represents the static SLAM map provided to the Navigation Stack. Obstacle map layer includes 2D obstacles. Inflation layer is where obstacles are inflated to calculate cost for each 2D costmap cell. There is a global costmap, as well as a local costmap. Global costmap is generated by inflating the obstacles on the map provided to the Navigation Stack. Local costmap is generated by inflating obstacles detected by the robot's sensors in real time. The tuning of the parameters of the costmaps is crucial for a right navigation. The move_base node may optionally perform recovery behaviors when the robot perceives itself as stuck. The node will take the following actions to attempt to clear out space: first, obstacles outside of a user-specified region will be cleared from the robot's map. Next, if possible, the robot will perform an in-place rotation to clear out space. If this too fails, the robot will more aggressively clear its map, removing all obstacles outside of the rectangular region in which it can rotate in place. This will be followed by another in-place rotation. If all this fails, the robot will consider its goal infeasible and notify the user that it has aborted.

The DWA planner depends on the local costmap which provides obstacle information. The local planner takes the velocity samples in robot's control space, examines the trajectories represented by those velocity samples, and finally eliminates bad velocities (ones whose trajectory intersects with an obstacle). Each velocity sample is simulated as if it is applied to the robot for a fixed time interval, controlled by *sim time* parameter. We can think of sim time as the time allowed for the robot to move with the sampled velocities. DWA maximizes an objective function to obtain optimal velocity pairs. In ROS's implementation, the cost of the objective function is calculated like this:

$$cost = p_b * d_p + g_b * d_g + o_b * cost_o \tag{5.1}$$

where

- d_p is the distance (in meters) to the path from the endpoint of the trajectory;

- d_g is the distance (in meters) to the local goal from the endpoint of the trajectory;

- $cost_o$ is the maximum obstacle cost along the trajectory in obstacle cost (0-254);

- p_b is the weight for how much the local planner should stay close to the global path. A high value will make the local planner prefer trajectories on global path;

- g_b is the weight for how much the robot should attempt to reach the local goal, with whatever path;

- o_b is the weight for how much the robot should attempt to avoid obstacles. A high value for this parameter results in indecisive robot that stucks in place.

The objective is to get the lowest cost.

5.2.8 explorer

The explorer node has the task to find a proper goal for the move_base node. Its approach is based on the detection of *frontiers*, regions on the border between free known space and unexplored space. From any frontier, the robot can see into unexplored space and add the new observations to its map. By reaching each frontier, or determining that frontier to be inaccessible, the robot can build a map of every reachable location in the environment. Beside frontier detection, exploration strategies are available to select goal points. In particular the exploration strategies available are:

- Navigate to nearest frontier, based on the travel path;
- Navigate using auctioning with cluster selection using nearest selection (Kuhn-Munkres);
- Navigate to furthest frontier;
- Navigate to nearest frontier, based on Euclidean distance;
- Navigate to random frontier;
- Cluster frontiers, then navigate to nearest cluster using Euclidean distance;
- Cluster frontiers, then navigate to random cluster.

The strategy that worked better for us was nearest frontier, based on the travel path. Hence, when we talk about the exploration strategy from now on we intend this one.

5.2.9 range_proximity_safety_controller

The range_proximity_safety_controller node prevent the robot from running into walls or obstacles by publishing the *cmd_vel* topic, that is the topic that controls the robot velocity. When the robot is under a certain distance threshold to a obstacle, that we set at 0.2 meters, the node drives the robot backwards and to the left, if the perceived obstacle is on the right, or vice versa. The node is created by Charly Huang of the National Taiwan University [20].

There were problems with the navigation tuning, since the Navigation Stack in ROS is prone to errors, and the robot can crash during exploration. Tuning was a consistent part of the work to make the robot autonomously explore all the environments that it is given. A helpful guide is the ROS Navigation Tuning Guide by Kaiyu Zheng [48]. The parameters used are provided in the next chapter for completeness and to help future works.

5.2.10 ROS launch files

roslaunch is a tool for easily launching multiple ROS nodes, as well as setting parameters on the parameter server. roslaunch was designed to fit the ROS architecture through composition: at first, write a simple system, then, combine it with other simple systems to make a larger one. Hence, it is suitable for projects like ours. It also includes options to automatically respawn processes that have already terminated for an error, that is a nice feature if something goes wrong. roslaunch files use one or more XML configuration files (with the .launch extension) which sets the parameters and contains a list of nodes to launch. We used roslaunch for launching at the same time all the nodes of the system. Our roslaunch file takes as input:

- *worldfile*, the file for the simulator;
- *outputfile*, the file where to write the SLAM estimated trajectory;
- *bag*, the file where to record the ground truth trajectory.

5.3 Python script

The whole system is launched by a Python 2.7 script. Python interpreters are available for many operating systems, allowing Python code to run on a wide variety of systems. This makes our code portable on different platforms. Our script takes as input a folder that contains .world files. These files are used by the Stage simulator to simulate the environments. When the script is called it starts a launch file and thus a exploration of an environment. The launch file launches all the nodes needed for simulation and exploration previously described. While the exploration is going, the script saves the grid map produced by GMapping as a png, every g seconds. The maps are used to check whether the exploration of the map is completed and

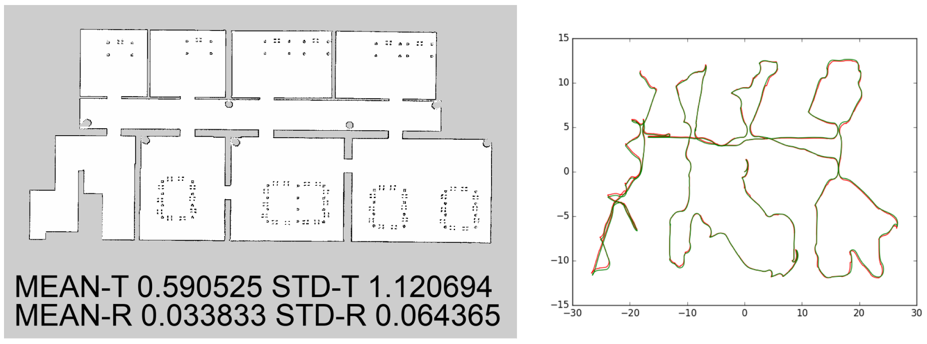


Figure 5.3: On the left there is a completed map with the respective translational mean error, rotational mean error, translational standard deviation, and rotational standard deviation. On the right the ground truth trajectory (green) and estimated (red) trajectory of the run that has built the map on the left

this is done by comparing the last two saved maps. This method to stop the exploration has its drawbacks: in large maps it can happen that the two last maps are the same, because the robot is moving in already explored areas, before entering new unknown areas. Another reason of undesired stop is that the robot can try to find a path and, hence, it remains still for more than g seconds, due to long computations. However, stuck problems are related most of the time to a bad tuning of the navigation algorithm, otherwise this method to stop the exploration works well. When the exploration is stopped we need to compute the error. The script first generates the relations file, then it calls the evaluator to generate the translational and rotational error files. To generate the relations file it takes the file written by the listener node and, as previously explained, it generates 500 random relations to compute the standard deviation σ, then, based on this value, it generates a relevant sample of relations, and writes them to a relations file. Since to understand the entity of the errors is useful for a human to visualize it, the script plots the ground truth trajectory and the one estimated by GMapping. Further, the last map saved is taken and the translational mean error, rotational mean error, translational standard deviation, and rotational standard deviation are printed on the map to have a visual confirm of the reliability of the error. Figure 5.3 shows these last two files.

Chapter 6

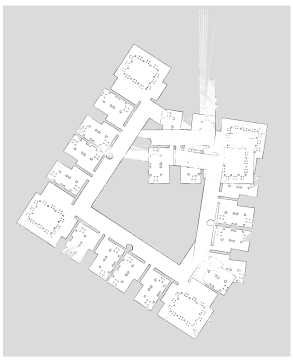
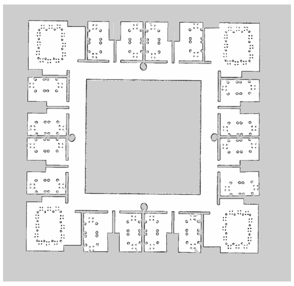
Experimental evaluation

This chapter presents the experiments performed to test the validity of our autonomous map quality evaluation system, at first discussing how we define the random model to generate the relations, then discussing the tuning of the autonomous exploration system, and at last showing the results of the evaluation of map quality produced on multiple runs of the data sets.

6.1 Random relations

The evaluator needs a reliable ground truth file, that is the relations file, to produce an error that measures the quality of the map. Further, we need an automatic method to generate it. The selected method to accomplish this task relies on a set of random generated relations. In order to arrive to this method, several methods have been tested previously. We will now explain the various methods tested and how we arrived to what we consider to be the most appropriate method to generate random relations. In other works that use the evaluation proposed by Kümmerle et al., like in the work of Quattrini Li et al. [28], only the A relations, are used. As said in Section 3.1 the A relations are the relations built by taking poses sequentially from the trajectory. Hence, our first try is to build these relations taking every 0.5 s subsequent poses of the robot and we compute the transformation (rotation and translation) needed to move the first pose on the second. Here is a piece of our A relations file:

```
25.8 26.3 0.115231457561 0.0145470435818 0.0 0.0 0.0 0.31068761832
26.3 26.8 0.108272621659 0.0118482917111 0.0 0.0 0.0 0.27195075041
26.8 27.3 0.0986139442487 0.00939971339177 0.0 0.0 0.0 0.24163496546
27.3 27.8 0.0883247476921 0.00737905211678 0.0 0.0 0.0 0.20121391734
```

Output Map		

A Mean Error	T = $0.007841m$
	R = $0.003744rad$
		T = $0.008280m$
		R = $0.004148rad$

A Std dev	T = $0.034581m$
	R = $0.015620rad$
		T = $0.012489m$
		R = $0.013806rad$

Table 6.1: The table shows the errors provided by the A relations on two different runs of the same environment, as can be seen the error doesn't detect that the left run has produced a bad map, since it has almost the same error of the good map

In [28] these kind of relations showed that they detect the error well, but the SLAM algorithms evaluated are visual SLAM approaches. Instead, we evaluate GMapping, that is a laser based approach, and on average it produces lower errors with respect to visual SLAM approaches. In fact, in our experiments, these relations influence the total error almost in the same way in every run, independently if the map produced is visually considered good or bad. It can be seen in Table 6.1: two runs in the same environment, which produced really different maps, produced an error based on the A relations very similar. This behavior is due to the fact that also in bad runs the local consistency of the map is mostly preserved, and since the error is the average of the errors produced by the relations, having many relations with low values produce a low error even in clearly bad maps. Figure 6.1 is an example of this. Even if there are big errors in the mapping this happens only for few poses, resulting in a good mapping of the remaining part of the environment, but shifted or rotated. This is why the brokenness metric has been proposed in [9]: as said in Section 2.2, brokenness measures how often the map broken, i.e., how many partitions of it are misaligned with respect to each other by rotational offsets. The A relations fail to detect this kind of failures of the mapping, and so the error produced by A relations will hardly have a high value.

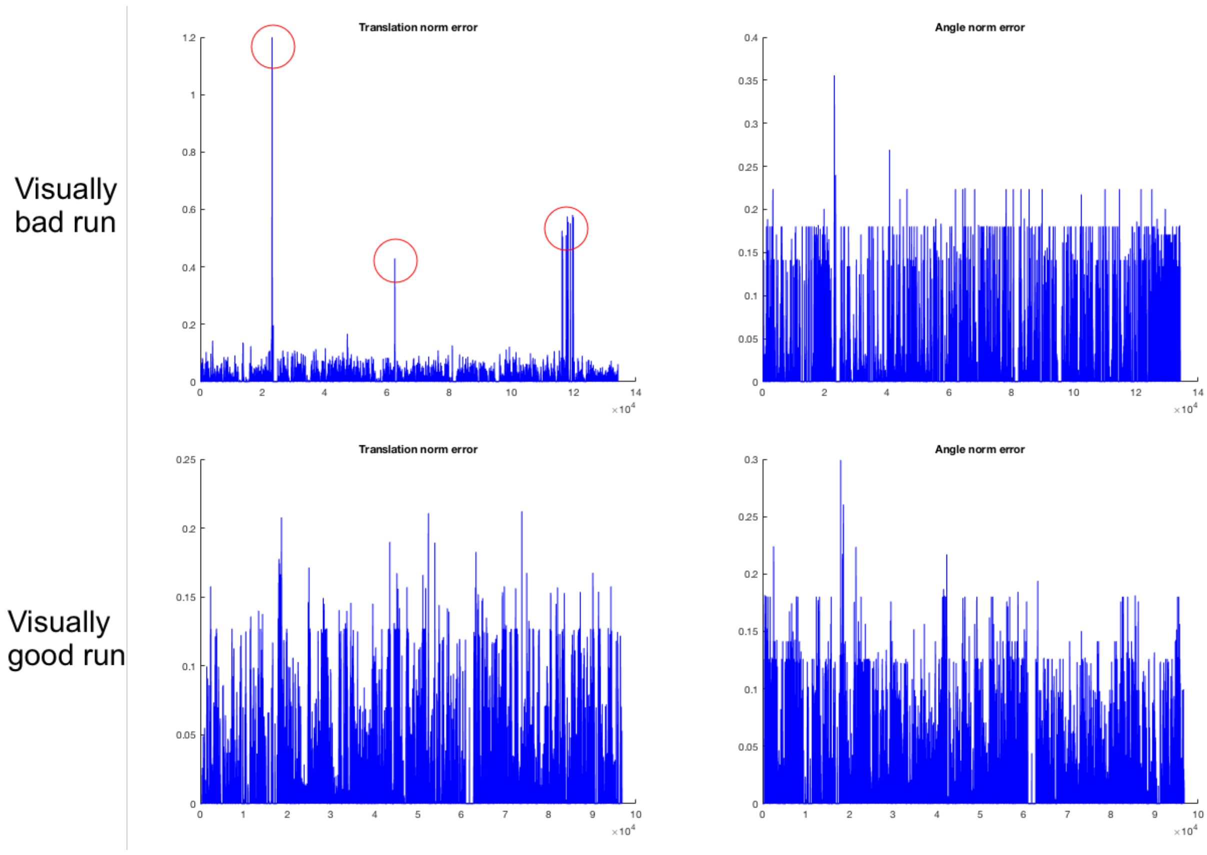


Figure 6.1: The four plots show the error produced by the A relations on the runs in Table 6.1. On the left there are the translational errors and on the right the rotational errors. In the plots about the rotational error there is not a sensible difference, while in the plots about the translational error there are some peaks (red circles) where the SLAM algorithm fails to localize the robot, but these errors are compensated by the other errors, resulting in a average translational error of 0.007841 for the visually bad map, and of 0.008280 for the visually good map

Hence we test another method, to create different kinds of relations at significant points:

- at the closure of a loop, L;
- after a displacement of x meters, M;
- after x seconds, S.

For the L relations we have to understand on the ground truth trajectory when there is the closure of a loop. A loop is considered a significant point because, for SLAM, loop closing is the main way to reduce the accumulated error in the map. We thought that when the robot returns in a point *near* to a previous visited point the loop can be considered closed. We have to define what near means for our scope. We used a parameter to define the Euclidean distance under which a point is considered near to another, in the experiments set at 0.2 meters. Also we define another parameter to check that the robot effectively returned near to a point and don't simply it stood still there, set to 0.5 meters. This parameter defines the Euclidean distance that the robot has to move from a point to consider it a possible return point for the closure of a loop. It can be noted that these parameters require the computing of several Euclidean distances for pairs of points, this results in a time complexity of this method in the order of $O(n^2)$, where n is the number of poses that can be in the order of hundreds of thousands. However, this kind of relations don't show a good effectiveness empirically, producing low values also when evaluating maps with big mapping errors. The M and S relations are easy to compute. For M, we compute the Euclidean distance from the point 0 until the x meters are reached, then the new point 0 is set as the current point. This is done for the whole length of the trajectory, with linear time complexity. For the S relations, it is even simpler, every x seconds a pose is taken, still with linear time complexity. The M and S relations produce a low number of relations and if there is a error in a point of the trajectory it is recognized by only one relation. This is a problem since, in this way, the value of the average error remains low. So we defined the *smart* relations, SM, that are all the poses in the trajectory that have a Euclidean distance that is more than m meters. For example, if m is set to 5 meters all the poses in the trajectory that have a Euclidean distance $d > 5$ are considered as relations. The computing time is in the order of $O(n^2)$, where n is the number of poses: to lighten the computing load we define another parameter t that is the temporal displacement between the poses considered in the trajectory. For example, if t is set to 2 seconds only the relations with 2 second of difference are considered: at $0, 2, 4, 6, 8, ...$

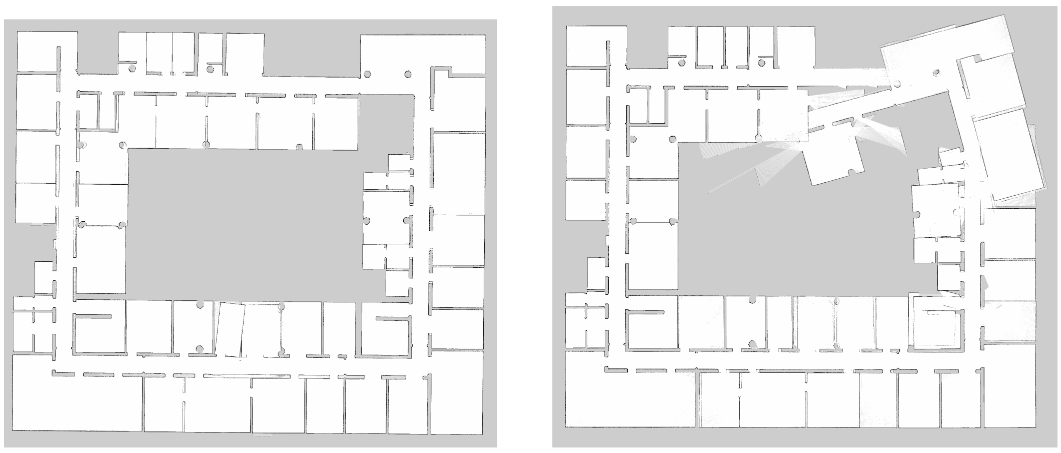


Figure 6.2: The maps produced by the runs to which the values of the following table refer. The visually bad map (bad) and the visually good map (G).

second instead of the standard displacement given by the simulator that is $0.1s$. With the SM relations, when a map is visually wrong the relations file produces a significantly higher error, similar to the one computed by the B relations introduced in the previous chapters. However, compared to the B relations, these relations are more complex to compute. The time needed to build a relations file it can be of several minutes on a desktop PC, with respect to few seconds of the B relations. Further, there are two parameters to tune by hand with respect to the single one (the dimension of the sample), that has a statistical motivation, of the B relations. In Table 6.2 the comparison of the different kinds of generated relations on the 2 runs in Figure 6.2 is shown. The experiments are made with different values of t and m. With lower values of t and m the computing complexity for the SM relations increases, but the Mean Translational Error is more similar to the B relations. The loop relations don't affect much the error even when the run is visually bad.

Data set	Temporal displacement t	Distance m	SM relations	B relations	Loops	MTE	Std deviation
bad	10	5	Yes	No	No	2.89	3.98
bad	5	2	Yes	No	No	3.27	4.01
bad	4	2	Yes	No	No	3.37	4.02
bad	2	2	Yes	No	No	3.47	4.04
bad	2	2	Yes	No	Yes	3.49	4.05
bad	-	-	No	Yes	No	3.42	4.01
bad	-	-	No	No	Yes	0.20	0.20
G	2	5	Yes	No	No	0.47	0.75
G	2	2	Yes	No	No	0.61	0.82
G	-	-	No	Yes	No	0.58	0.81

Table 6.2: In the table are shown the Mean Translational Error (MTE) and the standard deviation produced by different kinds of generated relations on a visually bad mapping run B and on a visually good mapping run G. The temporal displacement is in seconds and the distance in meters. As can be seen the loop relations don't affect much the error even when the run is visually bad. Instead the SM relations have values similar to the B relations in every run.

6.2 Autonomous exploration tuning

The ROS Navigation Stack has many different parameters that can be tuned to obtain a good performance. Here we show the most important in our experience. There are 4 files that contain the parameters to set:

- base_local_planner_params, the parameters that determine the behavior of the global and local planners;
- pioneer3at_costmap_common_params, the parameters common to the two costmaps: footprint of the robot, the obstacle layer, inflation layer, and static layer;
- pioneer3at_local_costmap_params, the parameters of the local costamp;
- pioneer3at_global_costmap_params, the parameters of the global costamp.

In our experiments we need a system working in real time, so the computing complexity can't be too high for the hardware used. Some of the parameters of the Navigation Stack are related to the computing power available, and setting them higher, to a certain extent, will provide a better

exploration of the environments. One of these parameter is max vel x: 0.85, that is the maximum velocity that the robot can reach, and is kept to a relatively low value. With higher values, the path that the local planner has to simulate is longer, so there are more possible trajectories to compute, but the time to compute them is the same, so it's more likely that the Navigation Stack will crash the robot since it has little time to decide where to go.
Another parameter that depends on the computing power is sim_time, this parameter indicates how long the trajectory is simulated beforehand. In our case it is set to 1.5 seconds.
The update frequencies of the local and global costmaps are strictly related to the computing power, and especially increasing the update frequency of the local costmap can have good effects on the path produced. These parameters are set as 10 updates per second in the local costmap and 2 per second in the global costmap.

The parameters for the local planner, namely DWA, to select the trajectory are:

```
pdist_scale: 0.6
gdist_scale: 0.8
occdist_scale: 0.05
```

These are the default parameters for pdist_scale, the weighting for how much the controller should stay close to the path it was given, and gdist_scale, the weighting for how much the controller should attempt to reach its local goal, while for occdist_scale, the weighting for how much the robot should attempt to avoid obstacles, we have raised its value to 0.05 from 0.01, because with the smaller value the robot passes too close to the obstacles and this can cause the robot to crash. We used meter_scoring: true, that means that instead of using the cells of the grid map as units, the distances computed by DWA are in meters.
Costmap parameters tuning is essential for the success of the planners. The costamp is composed of a static layer, a obstacle layer, and an inflation layer. The static layer is the map that GMapping, or the SLAM algorithm used, provides. The obstacle layer tracks the obstacles as read by the sensor data. Inflation layer is an optimization that adds new values around obstacles (i.e., inflates the obstacles) in order to make the costmap represent the configuration space of the robot. To have smooth paths we found that the following values for these two parameters in the inflation layer are good:

```
cost_scaling_factor: 7.5
inflation_radius: 2.25
```

cost_scaling_factor is inversely proportional to the cost of a cell. Setting it higher will make the decay curve of the values around the obstacles more steep. inflation_radius controls how far away the zero cost point is from the obstacle in meters.

ROS navigation has two recovery behaviors. These behaviors will be run when DWA fails to find a valid plan. After each behavior completes, DWA will attempt to build a plan. If planning is successful, DWA will continue the normal operations. Otherwise, the next recovery behavior in the list will be executed. The behaviors are: clear costmap recovery and rotate recovery. Clear costmap recovery is basically reverting the local costmap to have the same state as the global costmap removing all obstacles outside of the rectangular region in which it can rotate in place. The aggressive_clearance behavior will clear out to a distance of $4 * r$, where r is the circumscribed radius of the robot. Rotate recovery makes the robot rotate 360 degrees in place. Sometimes rotate recovery will hit an obstacle during the rotation and cause worse problems, so we don't use it. We also changed the behavior of the explorer node enabling the operate_with_goal_backoff parameter, this makes the robot navigate to a goal point which is close to (but not exactly at) the selected goal. This is helpful when the selected goal is too close to a wall. To help future works we provide all the parameters:

base_local_planner_params:

```
controller_frequency: 10.0
recovery_behavior_enabled: true
clearing_rotation_allowed: false
recovery_behaviors: [{name: aggressive_clearance,
          type: clear_costmap_recovery/ClearCostmapRecovery}]
aggressive_clearance:
     reset_distance: 0.0
max_replanning_tries: 3
planner_patience: 5.0
planner_frequency: 0.2
controller_patience: 15.0
TrajectoryPlannerROS:
     max_vel_x: 0.85
     min_vel_x: 0.1
     max_vel_theta: 0.7
     min_vel_theta: −0.7
     acc_lim_theta: 2.5
     acc_lim_x: 1.5
     acc_lim_y: 1.5
```

```
    min_in_place_rotational_vel: 0.4
    escape_vel: −0.1
    escape_reset_dist: 0.15
    escape_reset_theta: 0.15

    holonomic_robot: false −Since the P3AT is a differential robot,
                                      this parameter is kept to false
    yaw_goal_tolerance: 0.2
    xy_goal_tolerance: 0.3
    latch_xy_goal_tolerance: false
    pdist_scale: 0.6
    gdist_scale: 0.8
    meter_scoring: true

    occdist_scale: 0.05
    oscillation_reset_dist: 0.25
    oscillation_reset_timeout: 10.0
    prune_plan: false

    sim_time: 1.5
    sim_granularity: 0.025
    vx_samples: 10
    vtheta_samples: 20

    dwa: true
GlobalReplanner:
    old_navfn_behavior: true
    allow_unknown: true
    track_unknown_space: true
    default_tolerance: 0.0
```

pioneer3at_costmap_common_params:

```
footprint: [ [−0.2,−0.2], [0.2, −0.2], [0.2, 0.2], [−0.2,0.2] ]
footprint_padding: 0.0

obstacle_layer:
    enabled: true
    max_obstacle_height: 0.6
    min_obstacle_height: 0.0
    obstacle_range: 30.0
    raytrace_range: 30.0
    observation_sources: base_scan
    base_scan: {data_type: LaserScan, sensor_frame: base_laser_link,
                topic: /base_scan, marking: true, clearing: true,
                    max_obstacle_height: 0.6, min_obstacle_height: 0.0,
                    expected_update_rate: 0.4}

inflation_layer:
    enabled: true
    cost_scaling_factor: 7.5
    inflation_radius: 2.25

static_layer:
    enabled: true
    map_topic: /map
    subscribe_to_updates: true
```

pioneer3at_local_costmap_params:

```
local_costmap:
   global_frame: odom
   robot_base_frame: base_link
   update_frequency: 10
   publish_frequency: 0.2
   static_map: false
   rolling_window: true
   width: 6.0
   height: 6.0
   resolution: 0.02
   transform_tolerance: 0.15
   track_unknown_space: true
   unknown_cost_value: 255
```

pioneer3at_global_costmap_params:

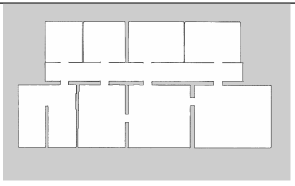
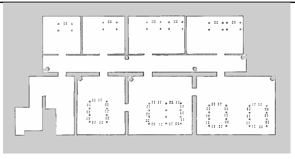
```
global_costmap:
   global_frame: map
   robot_base_frame: base_link
   update_frequency: 2
   publish_frequency: 0.1
   static_map: true
   transform_tolerance: 0.25
   track_unknown_space: true
   unknown_cost_value: 255
```

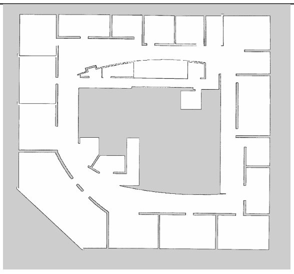
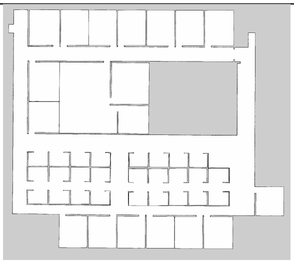
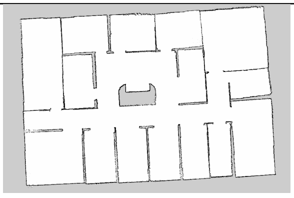
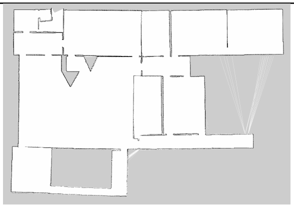
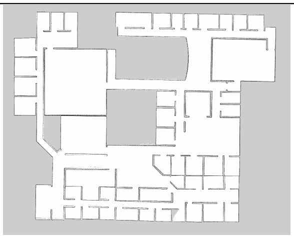
6.3 Quality of the maps

The results of the experiments that we present in this section are the results of the runs of the system on the data sets presented in Chapter 4. All the experiments were run with the parameters of the Navigation Stack mentioned in the previous section, on the Stage simulator with the P3AT robot equipped with a Hokuyo laser sensor of 270 degrees of field of view and 30 meters of range. The SLAM algorithms used is GMapping, set with 40 particles.

6.3.1 Success rate

We run the explorations of the data sets in Table 6.3 on a machine with a Core2 Quad Q9400@2.66Ghz and 4GB of RAM. A map is considered as successfully mapped if the exploration has produced a map visually similar to the ground truth map. There are two ways in which a run can not have success: if GMapping fails or if the Navigation Stack fails. The explorations have a good success rate thanks to the high reliability of GMapping and to the good settings of the Navigation Stack. In fact, changing the parameters of the Navigation Stack can have high influence on the success of a exploration. Most of the explorations that don't succeed are due to navigation problems, i.e., the robot hits a wall, rather than to the SLAM algorithm. The environments with lower number of successes are generally the ones that visually have a more complex structure, with many rooms and corridors.

Environment	Success	Map
freiburg52	10	
freiburg52_furnitures	10	

lab_intel	9	
lab_a	8	
lab_c	10	
lab_c_furnitures	10	
lab_d	9	
lab_f	6	

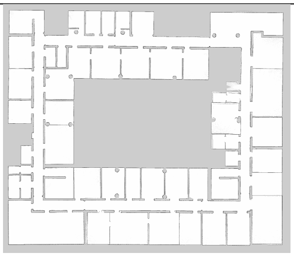
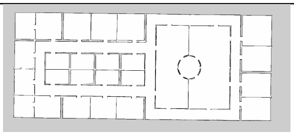
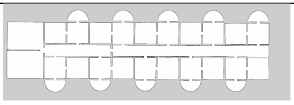
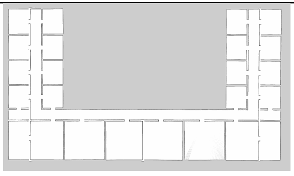
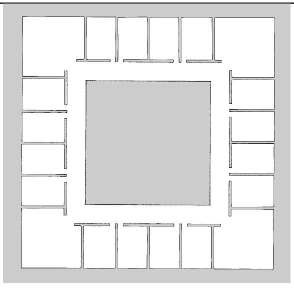
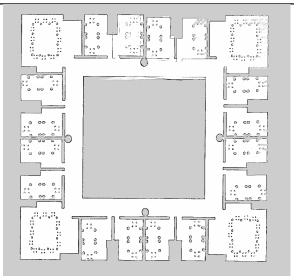
NLB	7	
office_b	8	
office_c	10	
office_f	6	
office_h	10	
office_h_furnitures	7	

Table 6.3: The table shows a sample of the environments with its name, the number of successful runs on 10 explorations, and on the rightmost column a complete map

6.3.2 Furniture

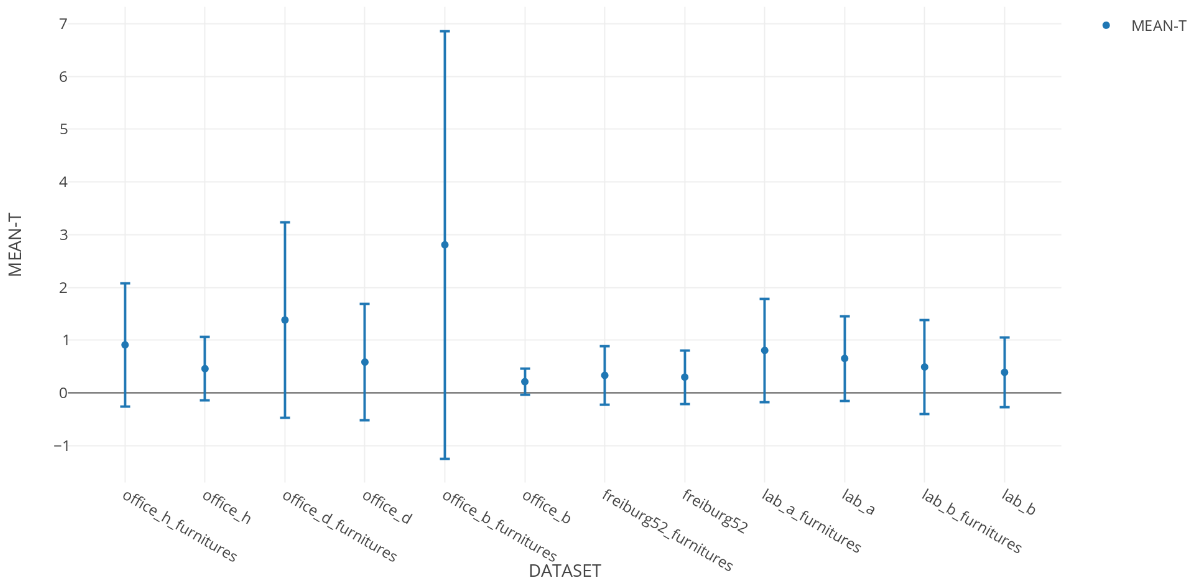


Figure 6.3: The error bars represents the average mean translational error and the corresponding average standard deviation of 10 runs in environments with and without furniture. In every environment the presence of furniture increases the error computed by the evaluator.

The height of the simulated robot is 0.5 meters, therefore the part of furniture detected by the laser range sensor at this height are the one represented in the 2d environments. Legs are the only part detected of tables and chairs and these are represented as points. Instead, wardrobes and cabinets are represented as rectangles. In Figure 6.3 the same environments with and without furniture are compared. As it can be seen, the average error increases when the same environment presents furniture. In some environments, like `freiburg52` and `lab_b`, that are two small environments, the difference between the errors with and without furniture is limited. Instead, for example, in `office_b` the difference is large. It is a much bigger environment with more rooms. With furniture it's easier to make errors during the exploration, and the sum of these errors eventually makes a bad map. The presence of furniture also increases the possibility for the robot to get stuck, since the legs of tables and chairs are obstacles that increase the clutter of the environment.

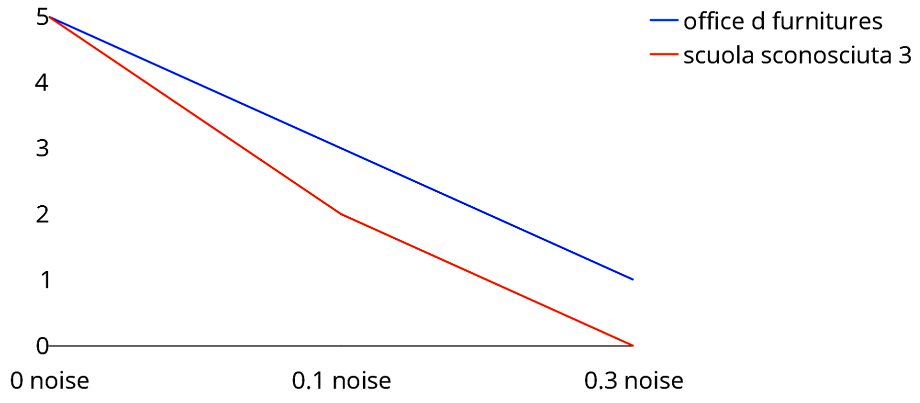


Figure 6.4: The number of succesfull explorations in 2 different environments with standard deviation of the Gaussian distribution set to 0, 0.1, and 0.3.

6.3.3 Noise

We run experiments introducing Gaussian noise on the laser range sensor with a node. The `laser_noise` presented in Section 5.2.6 takes a parameter n that is the standard deviation ("spread" or "width") of the Gaussian distribution. The presence of noise influences the successful exploration of an environment to a great extent as shown in the following figures. We run 5 explorations with n set at 0, 0.1, and 0.3 in two environments: "office d furnitures" and "scuola sconosciuta 3". The success of the explorations decreases if the noise increases: with the parameter set at 0.3 the system has not been able to conclude successfully an exploration of "scuola sconosciuta 3" and only one of "office d furnitures". This is shown in Figure 6.4, while the average translational and rotational error of the runs is plotted in Figure 6.5 and Figure 6.6.

The error even in successful runs is higher with a higher noise, due to the difficulty of GMapping to correctly estimate the position of the robot with bad readings of the range sensor. This behaviour is desirable because the output map is visually less accurate than the one without noise, as can be seen in Figure 6.7.

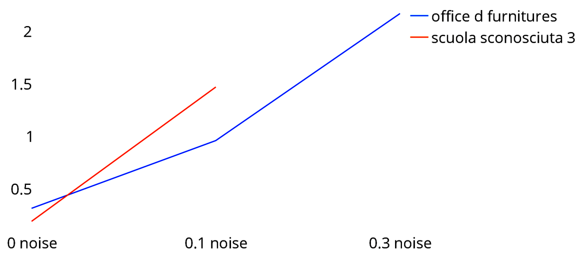


Figure 6.5: Translational error

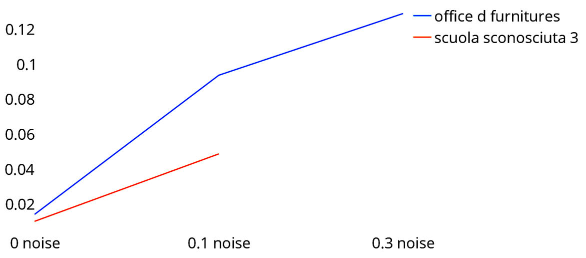


Figure 6.6: Rotational error

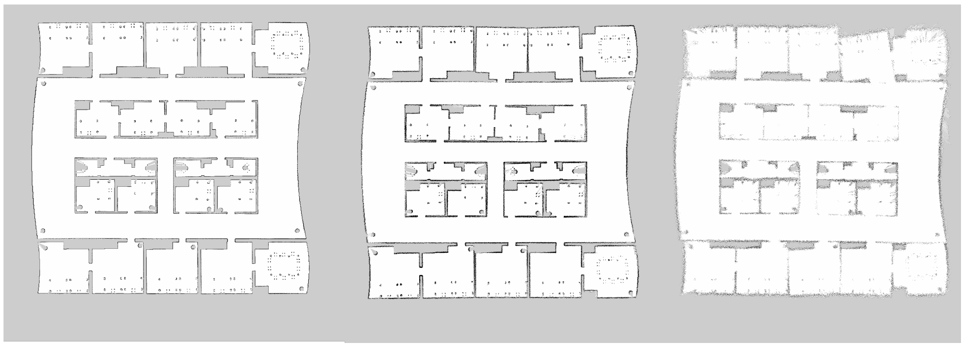


Figure 6.7: The leftmost map is built without noise, the central map with 0.1 Gaussian noise, and the rightmost one with 0.3 Gaussian noise.

6.3.4 Errors

In the following Tables, 6.4 and 6.5, and in the Figures 6.8 and 6.9, the errors on 20 successful runs are showed. The two tables, for every environment, show:

- the mean error, that is the average error of the errors produced by the relations;
- the standard deviation, that is the standard deviation of the errors produced by the relations;
- the min error, that is the minimum single error produced by a relation;
- the max error, that is the minimum single error produced by a relation;
- the number of relations in the relations file.

There is not a big difference between the errors of the environments of the data set of Bormann [10] and the schools data set, even though the schools have almost in every case a lower average error. From the the Figures 6.8 and 6.9 it can be noted that the mean error, the standard deviation, and the max error follow the same trend. The translational errors and rotational errors do not appear to have a strong relation, only the lab_f environment has both a high translational error and a high rotational error. lab_f is an environment with many rooms and it's exploration runs are longer. In this environment (and also in others, i.e., NLB and lab_c) that there are high errors because, through the explorations, the poses estimated can be wrong with respect to the ground truth for a short time, but GMapping succeed to relocate the robot to the right pose and, since the run is long, the average error remains low. All the runs considered here are runs that have produced a visually good map so a large max error doesn't mean that the map produced is visually bad, but the map probably contains some imperfections although the structure of the environment is preserved. However, there is not a defined threshold on the error to identify if a map is visually good or not. Empirically we observed that if a run has a translation error above 1 meter or a rotational mean error above 0.03 radians, that are 1.72 degrees, it is visually bad. If the error is higher than 3 meters or higher than 0.07 radians, that are 4 degrees, the map is completely useless for every kind of task. The system works well in recognizing if a map is visually good or bad, but in some cases it doesn't complete the exploration. If an environment is not explored completely but the part explored is visually good, the mean error is low. This happens when the robot gets stuck and is related to the exploration algorithm. However, if the exploration is completed the mean translational and rotational error are indicative about the quality of the map.

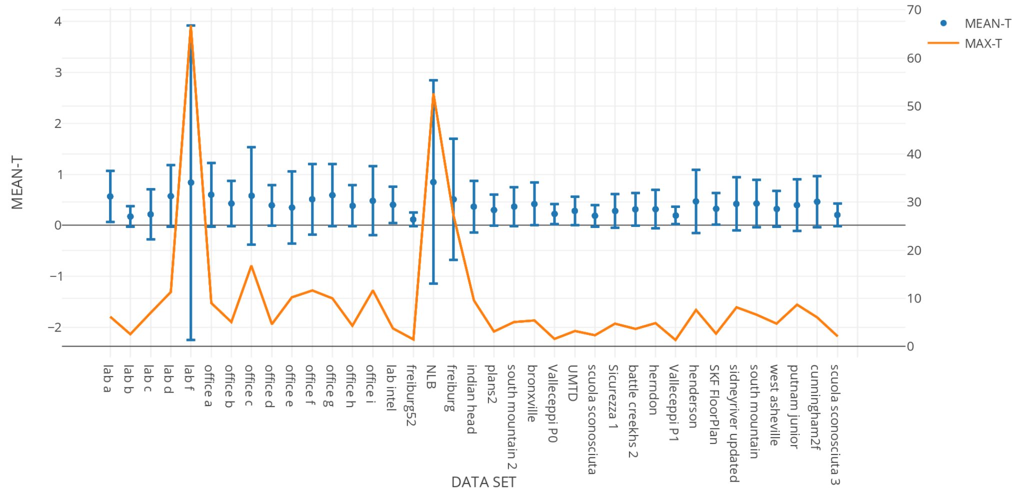


Figure 6.8: The translational errors of the runs. On the x axis there are the environments' names. The y axis values on the left are the mean average error that is depicted through the error bars, while the y axis values on the right are the max error that is depicted through the orange line. The max error follows the trend of the mean error.

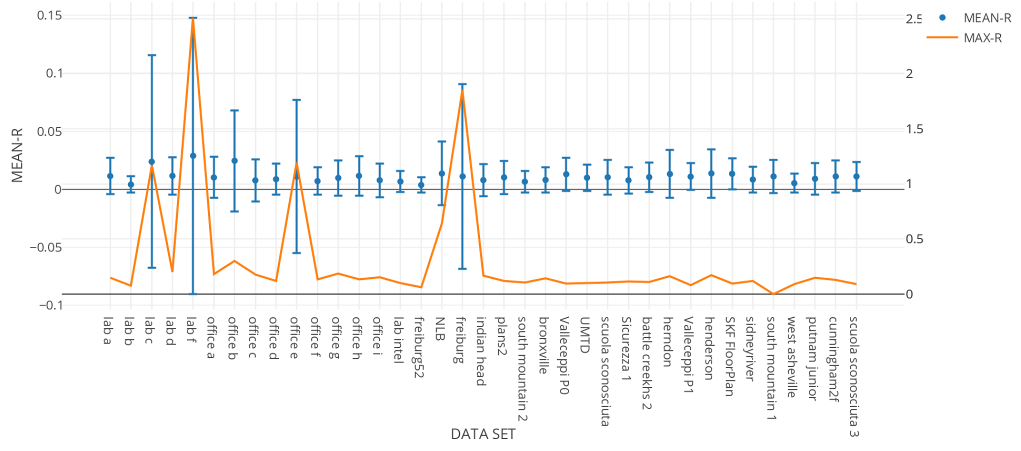


Figure 6.9: The rotational errors of the runs. These are not directly related to the translational errors: except for lab_f which has high errors in both evaluations, the environments with the higher rotational errors are not the ones with the higher translational error.

ENVIRONMENT	MEAN-T	STD-T	MIN-T	MAX-T	NUMMEASURES
lab_a	0.5674	0.5009	0.0013	6.1065	3712.6129
lab_b	0.1735	0.2018	0.0023	2.5153	627.0
lab_c	0.2161	0.4937	0.0021	7.0478	216.2222
lab_d	0.5735	0.6040	0.0017	11.2952	5412.125
lab_f	0.8397	3.0853	0.0005	66.8112	23210.3333
office_a	0.6006	0.6256	0.0033	9.0297	6328.1071
office_b	0.4286	0.4430	0.0008	5.0689	3028.0357
office_c	0.5803	0.9554	0.0004	16.8290	13016.5
office_d	0.3928	0.3987	0.0036	4.6523	2496.6764
office_e	0.3486	0.7096	0.0128	10.1825	332.8
office_f	0.5118	0.6906	0.0010	11.5920	8121.3478
office_g	0.5913	0.6128	0.0014	9.9545	9255.70588235
office_h	0.3828	0.4042	0.0014	4.2604	2459.1
office_i	0.4809	0.6758	0.0021	11.6891	8616.0
lab_intel	0.4032	0.3596	0.0026	3.8029	1983.4782
freiburg52	0.1148	0.1330	0.0032	1.4373	259.5
NLB	0.8480	1.9928	0.0018	52.6058	1629.0
freiburg	0.5104	1.1931	0.0050	27.2924	607.0
indian_head	0.3670	0.5022	0.0016	9.5708	3816.9756
plans2	0.3004	0.3041	0.0018	3.1101	1459.6785
south_mountain_school_2	0.3672	0.3802	0.0020	5.0151	2459.3571
bronxville	0.4189	0.4189	0.00	5.4262	2705.4
Valleceppi_P0	0.2250	0.1968	0.0012	1.6032	610.8
UMTD_School_Floorplan	0.2836	0.2829	0.0009	3.2270	1166.4516
scuola_sconosciuta	0.1881	0.2115	0.0016	2.3586	706.4444
Sicurezza_1	0.2814	0.3293	0.0010	4.6832	1935.8888
battle_creekhs_2	0.3139	0.3250	0.0012	3.5928	2150.9583
herndon	0.3165	0.3763	0.0010	4.8773	2531.7714
Valleceppi_P1	0.1928	0.1721	0.0031	1.3660	502.7317
henderson_high_school	0.4700	0.6208	0.0010	7.5559	8462.4
SKF_FloorPlan	0.3247	0.3052	0.0011	2.6430	1426.0238
sidneyriver_updated	0.4215	0.5207	0.0013	8.1608	3824.64
south_mountain_school_1	0.4280	0.4632	0.0013	6.5454	3092.1333
west_asheville	0.3215	0.3502	0.0024	4.7859	1754.4814
putnam_junior_high	0.3975	0.5037	0.0021	8.6455	5287.2142
cunningham2f	0.4638	0.4990	0.0015	6.0004	5838.0689
scuola_sconosciuta_3	0.2049	0.2185	0.0023	2.0616	687.8333

Table 6.4: The table shows the name of the environment, the average translational error, the translational standard deviation, the minimum translational error, the maximum translational error, and the number of measurements. All the values are the average on 20 successful explorations per environment and the errors are in meters

ENVIRONMENT	MEAN-R	STD-R	MIN-R	MAX-R	NUMMEASURES
lab_a	0.0114	0.0156	0.00	0.1471	3712.6129
lab_b	0.0041	0.0070	0.00	0.0778	627.0
lab_c	0.0238	0.0916	0.00	1.1689	216.2222
lab_d	0.0116	0.0162	0.00	0.1988	5412.125
lab_f	0.0289	0.1191	0.00	2.5101	23210.3333
office_a	0.0102	0.0177	0.00	0.1789	6328.1071
office_b	0.0246	0.0435	0.00	0.3008	3028.0357
office_c	0.0077	0.0182	0.00	0.1743	13016.5
office_d	0.0087	0.0134	0.00	0.1195	2496.6764
office_e	0.0110	0.0661	0.00	1.1958	332.8
office_f	0.0071	0.0117	0.00	0.1330	8121.3478
office_g	0.0098	0.0153	0.00	0.1870	6616.0588
office_h	0.0116	0.0171	0.00	0.1339	2459.1
office_i	0.0078	0.0146	0.00	0.1519	8616.0
lab_intel	0.0067	0.0089	0.00	0.0981	1983.4782
freiburg52	0.0037	0.0065	0.00	0.0592	259.5
NLB	0.0136	0.0274	0.00	0.6403	1629.0
freiburg	0.0111	0.0796	0.00	1.8554	607.0
indian_head	0.0079	0.0138	0.00	0.1651	3816.9756
plans2	0.0103	0.0142	0.00	0.1201	1459.6785
south_mountain_school_2	0.0066	0.0094	0.00	0.1059	2459.3571
bronxville	0.0081	0.0110	0.00	0.1421	2705.4
Valleceppi_P0	0.0130	0.0143	0.00	0.0933	610.8
UMTD_School_Floorplan	0.0100	0.0113	0.00	0.1030	1166.4516
scuola_sconosciuta	0.0104	0.0151	0.00	0.1062	706.4444
Sicurezza_1	0.0078	0.0114	0.00	0.1145	1935.8888
battle_creekhs_2	0.0105	0.0127	0.00	0.1088	2150.9583
herndon	0.0132	0.0207	0.00	0.1635	2531.7714
Valleceppi_P1	0.0109	0.0116	0.00	0.0815	502.7317
henderson_high_school	0.0137	0.0209	0.00	0.1694	8462.4
SKF_FloorPlan	0.0134	0.0135	0.00	0.0958	1426.0238
sidneyriver	0.0084	0.0111	0.00	0.1161	3824.64
south_mountain_school_1	0.0111	0.0144	0.00	0.1380	3092.1333
west_asheville	0.0053	0.0081	0.00	0.0886	1754.4814
putnam_junior_high	0.0091	0.0136	0.00	0.1487	5287.2142
cunningham2f	0.0111	0.0137	0.00	0.1275	5838.0689
scuola_sconosciuta_3	0.0111	0.0124	0.00	0.0900	687.8333

Table 6.5: The table shows the name of the environment, the average rotational error, the rotational standard deviation, the minimum rotational error, the maximum rotational error, and the number of measurements. All the values are the average on 20 successful explorations per environment and the errors are in radians

Chapter 7

Conclusions and future research directions

This thesis has focused on the evaluation, in an automated way, of the quality of the maps produced by a SLAM algorithm. Defining the quality of a map produced by SLAM algorithms in a quantitative way is a difficult task, since it is often the human visual examination that defines it. In this thesis a method has been proposed for automating the application of the evaluation metric proposed by Kümmerle et al. [24], based on the comparison with a ground truth file that represents the trajectory performed by the robot. The ground truth file, that contains a list of relative displacements between poses of the trajectory (the *relations*), was built manually on a limited set of examples in [24], and we tried different ways to automatically build it. The best results in our experiments were given by adding relations relative to random displacements on the whole trajectory. This kind of relations are fast to compute and produce an error that is a good indicator of the quality of the maps.

The experimental evaluation has been performed using ROS/Stage simulations. To perform explorations of different environments in an autonomous way, we tuned the Navigation Stack of ROS by setting its parameters. The system is built to generate data on the quality of the maps produced by GMapping autonomously, thus it is able to stop an exploration when it's completed and start another one, while generating the evaluation files. The results obtained denote that the performance of GMapping is stable during explorations of different structures and different environments. We noted that the performance of the SLAM algorithm depends on the path that the Navigation Stack produces, so a good tuning is necessary for the building of a qualitatively good map. A future work can take a look more in depth in this sense, studying the correlation between the path and the map produced by the SLAM algorithm.

In the future, several aspects can be investigated. The first is to predict the quality of the maps produced by the SLAM algorithms. This means finding relevant characteristics of the environments that can be correlated to the performance of the SLAM algorithms. A system capable of doing so can be of great importance in the industry and in research. Another interesting work is to analyze the performance of different SLAM algorithms with our approach, to have the data to predict the performance not only of GMapping, but also of other SLAM algorithms, based on different methods or sensors. With these data, predictions can be done also for different SLAM algorithms and these can be compared, before actually explore the environments, to know which algorithm is the most suitable for that specific environment. Finally, it would be necessary to validate our approach also on data set obtained from online use of a real robot.

Bibliography

[1] GMapping. `http://wiki.ros.org/gmapping`. Accessed: 2017-11-08.

[2] Hokuyo UTM-30LX. `https://www.hokuyo-aut.jp`. Accessed: 2017-11-08.

[3] TheRawseedsProject. `http://www.rawseeds.org`. Accessed: 2017-11-09.

[4] Gnu general public license. `http://www.gnu.org/licenses/gpl.html`, June 2007.

[5] Francesco Amigoni, Matteo Luperto, and Viola Schiaffonati. Toward generalization of experimental results for autonomous robots. *Robotics and Autonomous Systems*, volume(90):4–14, 2017.

[6] Josep Aulinas, Yvan Petillot, Joaquim Salvi, and Xavier Lladó. The SLAM Problem: A Survey. In *Proceedings of the Conference on Artificial Intelligence Research and Development: Proceedings of the International Conference of the Catalan Association for Artificial Intelligence(CCIA)*, pages 363–371, Amsterdam, The Netherlands, The Netherlands, 2008.

[7] Benjamin Balaguer, Stefano Carpin, and Stephen Balakirsky. Towards quantitative comparisons of robot algorithms: Experiences with slam in simulation and real world systems. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems(IROS) Workshop*, 2007.

[8] Kostas E Bekris, Max Glick, and Lydia E Kavraki. Evaluation of algorithms for bearing-only SLAM. In *Proceedings of the IEEE International Conference on Robotics and Automation(ICRA)*, pages 1937–1943, 2006.

[9] Andreas Birk. A quantitative assessment of structural errors in grid maps. *Autonomous Robots*, 28(2):187–196, 2010.

[10] Richard Bormann, Florian Jordan, Wenzhe Li, Joshua Hampp, and Martin Hägele. Room segmentation: Survey, implementation, and analysis. In *Proceedings of the IEEE International Conference on Robotics and Automation(ICRA)*, pages 1019–1026, 2016.

[11] Hugh Durrant-Whyte and Tim Bailey. Simultaneous localization and mapping: part I. *IEEE Robotics & Automation magazine*, 13(2):99–110, 2006.

[12] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, 4(1):23–33, 1997.

[13] Brian Gerkey, Richard T Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the International Conference on Advanced Robotics(ICAR)*, volume 1, pages 317–323, 2003.

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

[15] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. Improved techniques for grid mapping with Rao-Blackwellized particle filters. *IEEE Transactions on Robotics*, 23(1):34–46, 2007.

[16] Jochen S. Gutmann, Wolfram Burgard, Dieter Fox, and Kurt Konolige. An experimental comparison of localization methods. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems(IROS)*, volume 2, pages 736–743, 1998.

[17] Dirk Hahnel, Wolfram Burgard, Dieter Fox, and Sebastian Thrun. An efficient FastSLAM algorithm for generating maps of large-scale cyclic environments from raw laser range measurements. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems(IROS)*, volume 1, pages 206–211, 2003.

[18] 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.

[19] Andrew Howard. The Robotics Data Set Repository (Radish). `http://radish.sourceforge.net`, 2003. Accessed: 2017-11-09.

[20] Charly Huang. Range proximity safety controller. `https://github.com/Megacephalo/range_proximity_safety_controller`. Accessed: 2017-11-09.

[21] Sebastian Kasperski. Nav2d. `https://github.com/skasperski/navigation_2d`. Accessed: 2017-11-09.

[22] Sebastian Kasperski. Pioneer 3-AT. `http://www.mobilerobots.com`. Accessed: 2017-11-09.

[23] Alexander Kleiner, Chris Scrapper, and Adam Jacoff. RoboCup Rescue interleague challenge 2009: Bridging the gap between simulation and reality. In *Proceedings of the Performance Metrics for Intelligent Systems(PerMIS) Workshop*, pages 115–121, 2009.

[24] 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):387–407, 2009.

[25] Rolf Lakaemper and Nagesh Adluru. Using virtual scans for improved mapping and evaluation. *Autonomous Robots*, 27(4):431–448, 2009.

[26] David Charles Lee. *The map-building and exploration strategies of a simple sonar-equipped mobile robot: An experimental, quantitative evaluation.* 2003.

[27] John J Leonard and Hugh F Durrant-Whyte. Mobile robot localization by tracking geometric beacons. *IEEE Transactions on Robotics and Automation*, 7(3):376–382, 1991.

[28] Alberto Quattrini Li, Adem Coskun, Sean M. Doherty, Shervin Ghasemlou, Apoorv S. Jagtap, Sharmin Rahman, Akanksha Singh, Marios Xanthidis, Jason M. O'Kane, et al. Experimental comparison of open source vision based state estimation algorithms. In *Proceedings of the International Symposium on Experimental Robotics(ISER)*, pages 775–786, 2016.

[29] Matteo Luperto and Francesco Amigoni. Exploiting structural properties of buildings towards general semantic mapping systems. In *Intelligent Autonomous Systems 13*, pages 375–387. 2016.

[30] Alexei A Makarenko, Stefan B Williams, Frederic Bourgault, and Hugh F Durrant-Whyte. An experiment in integrated exploration. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems(IROS)*, volume 1, pages 534–539, 2002.

[31] Johannes Pellenz and Dietrich Paulus. Mapping and map scoring at the RoboCup Rescue competition. In *Proceedings of the Robotics: Science and Systems (RSS) Workshop*, volume 46, pages 47–66, 2008.

[32] Max Pfingsthorn, Bayu Slamet, and Arnoud Visser. *A Scalable Hybrid Multi-robot SLAM Method for Highly Detailed Maps*, pages 457–464. 2008.

[33] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *Proceedings of the International Conference on Robotics and Automation(ICRA) Workshop*, volume 3, page 5, 2009.

[34] Joao Machado Santos, David Portugal, and Rui P Rocha. An evaluation of 2D SLAM techniques available in robot operating system. In *Proceedings of the IEEE International Symposium on Safety, Security, and Rescue Robotics(SSRR)*, pages 1–6, 2013.

[35] Sören Schwertfeger and Andreas Birk. Evaluation of map quality by matching and scoring high-level, topological map structures. In *Proceedings of the IEEE International Conference on Robotics and Automation(ICRA)*, pages 2221–2226, 2013.

[36] Sören Schwertfeger, Adam Jacoff, Chris Scrapper, Johannes Pellenz, and Alexander Kleiner. Evaluation of maps using fixed shapes: The fiducial map metric. In *Proceedings of the Performance Metrics for Intelligent Systems(PerMIS) Workshop*, pages 339–346, 2010.

[37] Randall Smith, Matthew Self, and Peter Cheeseman. Estimating uncertain spatial relationships in robotics. In *Autonomous Robot Vehicles*, pages 167–193. 1990.

[38] Cyrill Stachniss, Udo Frese, and Giorgio Grisetti. OpenSLAM.org give your algorithm to the community. `http://www.openslam.org`, 2007. Accessed: 2017-11-09.

[39] Cyrill Stachniss, Giorgio Grisetti, Wolfram Burgard, and Nicholas Roy. Analyzing Gaussian proposal distributions for mapping with Rao-Blackwellized particle filters. In *Proceedings of the IEEE/RSJ Inter-*

national Conference on Intelligent Robots and Systems(IROS), pages 3485–3490, 2007.

[40] Sebastian Thrun. Robotic mapping: A survey. *Exploring artificial intelligence in the new millennium*, 1:1–35, 2002.

[41] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. A probabilistic approach to concurrent mapping and localization for mobile robots. *Autonomous Robots*, 5(3-4):253–271, 1998.

[42] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics*. 2005.

[43] Ioana Varsadan, Andreas Birk, and Max Pfingsthorn. Determining map quality through an image similarity metric. In *Proceedings of the Robot Soccer World Cup*, pages 355–365, 2008.

[44] Asim Imdad Wagan, Afzal Godil, and Xiaolan Li. Map quality assessment. In *Proceedings of the Performance Metrics for Intelligent Systems(PerMIS) Workshop*, pages 278–282, 2008.

[45] Oliver Wulf, Andreas Nüchter, Joachim Hertzberg, and Bernardo Wagner. Benchmarking urban six-degree-of-freedom simultaneous localization and mapping. *Journal of Field Robotics*, 25(3):148–163, 2008.

[46] Takehisa Yairi. Covisibility-based map learning method for mobile robots. In *Proceedings of the Pacific Rim International Conference on Artificial Intelligence(PRICAI)*, pages 703–712, 2004.

[47] Stuart Young, Thomas Mazzuchi, and Shahram Sarkani. A Framework for Predicting Future System Performance in Autonomous Unmanned Ground Vehicles. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 47(7):1192–1206, 2017.

[48] Kaiyu Zheng. Ros navigation tuning guide. *arXiv preprint arXiv:1706.09068*, 2017.