Politecnico di Milano
Department of Electronics, Information and Bioengineering
Doctoral Programme In Information Technology

Methodologies for Benchmarking of Robot Tasks and Systems

Doctoral Dissertation of:
Enrico Piazza

Supervisors:
Prof. Matteo Matteucci, Prof. Pedro U. Lima

Tutor:
Prof. Francesco Amigoni

The Chair of the Doctoral Program:
Prof. Luigi Piroddi

2022 – 33rd Cycle

Abstract

WHEN the performance of robot functionalities and robot software components is evaluated, functionalities and software components are usually assumed independent from characteristics of the robot system and environment in which they operate. However, these aspects influence the performance, e.g., the performance of a software component implementing a robot functionality depends on the robot system configuration, such as which sensors are used, the sensor properties, or the robot platform kinematics, characteristics of the environment where the robot operates, and the component configuration parameters. This thesis proposes a benchmarking methodology which models the impact of the characteristics of the robot system and its environment on the performance of functionalities and their implementation as software components. However, measuring the performance of a software component for every combination of the variables which influence the performance would be untractable. To make the problem tractable, we propose to sample a relatively small number of combinations, conduct experiments for each of them, and from these results estimate a statistical model of the software component performance, which we call component performance model. To study the performance dependency between components, we build component performance models for multiple functionalities of a robot system. A performance model allows the comparison of different components implementing the same functionality to determine the best one to be used in a given setting and its optimal configuration. Moreover, the performance models enable us to predict the performance of a robot system given the performance models of its components. Two case studies illustrate application of this methodology to extract performance models: a first case study about benchmarking the Simultaneous Localization and Mapping (SLAM) functionality and the second case study focusing on an autonomous navigation system composed of a localization component and a navigation component.

Contents

CHAPTER 1

Introduction

Benchmarking can be defined as a standardized procedure to measure the performance of a software/hardware solution against a reference performance. Several areas of research require benchmarking to assess the value of new results against some reference performance. An example of wide-spread use of benchmarking in the fields of machine learning, robotics and computer vision research are publicly available datasets against which algorithms are compared and ranked [11, 16, 28].

Benchmarking, to be a useful tool in science and engineering, requires the application of the concepts of reproducibility and repeatability. A scientific work is reproducible when other researchers are able to independently reproduce the results of an experiment by using an equivalent setup and following the same experimental techniques. Repeatability is a quality of the scientific results and it describes the ability to confirm the outcome of the experiments through systematically repeated trials [2].

Some sub-fields of robotics research and engineering are affected by a lack of reproducibility and repeatability, as highlighted in the works of Bonsignorio *et al.* [5, 6], due to the complexity of hardware and software architectures of the robot systems and the environments in which the systems or methods are tested, as well as methodological obstacles such as the need for specific test equipment.

Important developments have happened in the last decade concerning the availability of development frameworks. In the academic sphere, the most important is the ROS framework [35], which among other aspects, provides access to a database of software packages developed by the robotics community which implement robot functionalities with standardized communication interfaces. Examples of the most common functionalities are navigation, arm motion planning, localization, and Simultaneous Localization And Mapping (SLAM). Regardless of a specific framework, benchmarking is needed to assess the quality of the software packages and the methods used in the robot

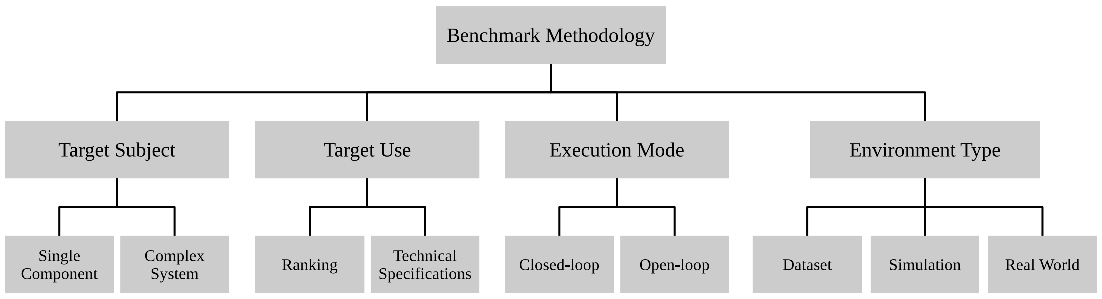


Figure 1.1: *Taxonomy of benchmarking methodologies.*

systems. Some form of evaluation is usually performed in the literature presenting the methods, but these usually do not provide a comprehensive evaluation or a comparison with the competing methods. Some literature works propose benchmarks of the most popular methods or software packages for a specific robot functionality to evaluate their strengths and weaknesses and rank them by their performance [18,41,45]. Some other works estimate the effect of the complexity or difficulty of the task on the performance of robot software and methods [2,10,18,21,29].

In these benchmarking works, the performance of the methods is evaluated in isolation from other components of a robot system. Benchmarking of methods implementing a specific functionality can provide a comparison of their performance, but the performance of one method is often dependent on the performance of other methods used in the robot, as well as properties of the robotic platform and characteristics of the environment in which the robot operates. When benchmarking methods which are meant to work in complex robot systems, these aspects must be included in the methodology in order to obtain a comprehensive evaluation. A few works proposed methodologies to evaluate the effect of single components on the performance of the system, such as [3,4,12,13,26,27], but these methodologies have not been demonstrated in practice.

Studying how to benchmark complex systems rather than single functionalities can also improve the use of benchmarking in the development process. Being able to evaluate the overall performance of the robot system requires to evaluate the performance of each component and understand its impact on the system. Even when building a system by integrating components provided by third parties, choosing which components to use requires not only information on their performance, but also knowledge on how they interact. Therefore, the developer of a component that wants to document the performance of their software, would need to evaluate how the performance of a component depends on properties of the robot platform and characteristics of the environment.

The objective of this thesis is to introduce a methodology for benchmarking of autonomous robot systems which enables characterizing the impact of the performance of the components of a system on the performance of the system itself, the performance dependency between software components, and supporting good experimental methodology.

Benchmark framework	Environment type	Execution mode	Target Subject	Target Use
Proposed methodology	any	any	complex system	technical specification
Plug and Bench	any	any	complex system	technical specification
ERL Consumer Service Robots	real world	closed-loop	complex system	rank
KITTI Vision Benchmark Suite	dataset	open-loop	single component	rank
Pascal VOC	dataset	open-loop	single component	rank
COCO	dataset	open-loop	single component	rank
OpenAI Gym leaderboard	simulation	closed-loop	single component	rank

Table 1.1: *Notable examples of benchmark methodologies and classification with respect to our taxonomy.*

1.1 Benchmarking Methodologies: A Taxonomy

In this section we present a taxonomy of benchmarking methodologies from the scientific literature. Since our methodology can be categorized as a specific type of benchmarking, it is useful to list the differences between established and emerging benchmarking methodologies. In our taxonomy, we define the following fundamental aspects of benchmarking methodologies: target subject, target use, execution mode, and environment type. In Fig. 1.1 we show the taxonomy graphically. In Tab. 1.1 we list some notable examples and their classification.

The **target subject** describes the subject of the benchmark. **Single component** benchmarks aim at measuring the performance of an algorithm, method or software component in isolation. Whereas **complex system** benchmarking methodologies aim at measuring the performance of a subject structured as a system and its components. Implying the methodology evaluates the effect of the performance of one component on other components, or the effect of features of the robot system (e.g., sensors or kinematic) on the performance of the components under evaluation.

Object detection and object classification benchmarks, such as [11, 16, 28], are generally single component benchmarks, since they evaluate different algorithms with one or more performance metrics, and the algorithms are executed by themselves, rather than in a system. An example of complex system benchmark methodology is the one applied in the European Robotics League (ERL) competition [4, 27], in which the challenges are framed as benchmark experiments. Two types of benchmarks are organized: functionality benchmarks, which measure the performance of a single functionality, and task benchmarks, which measure the robot system's overall performance in a complex task. Another example of complex system benchmark methodology is presented in Plug and Bench [12, 13], which proposes to integrate performance evaluation into the model-driven software development framework of RobMoSys[1]. The idea put forward in the work is to provide a way to estimate the performance of a robot system from performance data of its software components, allowing a robot software developer to estimate the performance of the system at design time, i.e., before actually deploying the software to a real robot.

The **target use** categorizes how the benchmark results are used. The target use can be ranking or technical specifications. **Ranking** implies the results of the benchmarks are used to compare different subjects with each other or with a reference. The output

[1]`https://robmosys.eu/`

is a ranking of subjects ordered by one (or more) performance metrics. Results are compared through leaderboards, e.g., to publish research results and compare them to the state of the art. The **technical specifications** category implies the result of the benchmark is a higher level description of the performance. A technical specification provides its user with the ability to predict the performance as a function of relevant parameters such as characteristics of the environment or the robot system.

Object detection and object classification benchmarks, such as [11, 16, 28], are examples of methodologies to produce rankings based on different performance metrics. The Plug and Bench methodology [3, 12, 13] is an example of technical specification, in fact, we derive the term technical specification from the work of [3]. The idea behind the methodology is to enable the developer of a robot system to use the results of benchmarking to predict, at design time, the performance of software components from: characteristics of the robot system (e.g., which sensors are used and their properties, kinematic type of the robot platform, etc); characteristics of the output or behavior of software components on which the benchmarked component depends; and characteristics of the environment.

The **execution mode** categorizes the benchmark into the open-loop and closed-loop categories. A benchmark execution is considered to be in **closed-loop** if the subject is able to act on the environment, affecting subsequent action decisions, **open-loop** otherwise.

The **environment type** describes how the subject of the benchmark receives its input. The environment type can be a dataset, a simulation, or the real world. **Dataset** benchmarks rely on pre-recorded data that is fed to the subject. Dataset benchmarks naturally support the implementation of open-loop benchmarks, since the state of the environment is captured in each dataset and can not be changed by the behavior of the subject. In **simulation** benchmarks, the environment is simulated, allowing the subject to influence state of the environment and allowing us to set specific initial conditions. A simulation benchmark naturally fits with the implementation of closed-loop benchmarks, although open-loop benchmarks can derive advantages from a simulated environment, where the state can be controlled exactly and is fully observable, allowing to generate a considerable amount of dataset-based equivalent environments. **Real world** benchmarks are implemented in a real environment. Compared to simulation benchmarks, reality benchmarks allow higher fidelity in the data fed to the subject of the benchmark and avoid approximations that may be required for the modelling of the simulated environment. On the other hand, executing experiments with real robots in a real environment makes replicating exact conditions and full observability very hard if not impossible.

Examples of dataset benchmarks are the object detection and classification benchmarks [11, 16, 28], which publish datasets of images and annotations used to train the machine learning algorithms and evaluate their performance. The benchmark methodology presented in [46] uses a photorealistic simulation of 3D building interiors to train and evaluate the performance of visual navigation algorithms. Additionally, the methodology takes advantage of the physics simulation to evaluate the ability to complete the navigation task by interacting with the objects in the environment. An interesting cross-over between the simulation and dataset categories can be found in object detection datasets [1] obtained by synthesizing datasets from real background images to which high fidelity rendering of the objects from 3D models are added. An example of real world benchmark methodology is presented in [43], which evaluates the performance of a navigation component in two real environments for extended periods of time. A second example of real world benchmark methodology is the ERL competition [4, 27], which organizes the competition's events in various testbeds built to realistically replicate domestic and industrial environments.

1.2 Contributions

Our contributions consists in producing a review of benchmarking methodologies and a taxonomy to classify them, the formalization of the concept of performance modelling of robot software components and robot systems, the development of a software framework which supports producing benchmarks following the proposed methodology, and lastly, in producing two case studies which demonstrate the proposed methodology.

In more details, we produce a taxonomy of benchmarking methodologies which classifies them based on four key characteristics. We then review and classify works from literature which are relevant to benchmarking in robotics. We developed our methodology by formalizing the concept of performance model. We define and formalize the entities relevant to the performance model: the robot system, the environment, the experimental protocol, and the structure of the data. We describe how these entities are used to build the performance model of software components, and how these are composed in order to apply the methodology to a complex robot system. We developed a software framework supporting our methodology which allows us to define a set of parameters used to automatically run simulation experiments for different system and environment configurations and automatically collect the data required to build the performance models. The software framework has been developed iteratively while producing performance models for different robot functionalities, improving its generality and adding tools to aid the benchmark developer in its use.

We produced two case studies to demonstrate the proposed methodology, the first on performance modelling of SLAM methods, and the second on performance modelling of an autonomous navigation system composed by multiple components. We developed simulation models of two robots. One was obtained by modifying a pre-existing simulation model, and one was developed from scratch. Additionally, we developed a tool to create simulation environments from grid-maps, which allows to quickly add environments to the experimental setup.

The first case study, developed concurrently to the performance modelling software framework, required us to set up a robot system using the ROS framework which allowed us to run experiments and collect the experimental data. We developed the metrics needed to measure the performance of the SLAM methods, some of which were taken from scientific publications and adapted to our needs, while some were developed from the ground up. We then developed different ways to measure characteristics of the environmet, one of which was suitable to be used in the case study. Finally, using the data from the experiments we were able to produce a performance model for three SLAM methods. The proposed methodology and the case study were published to the IEEE Robotics and Automation Letters (IEEE RAL) journal [34].

We developed the second case study to demonstrate the composition of performance models for a complex robot system. The system, implemented in the ROS2 framework, includes a localization component and a navigation stack, which is itself a set of components. In order to evaluate the performance of both the localization and the navigation components, we developed and adapted the two robot simulation models and added more environments to the set already used in the first case study. We developed the metrics needed to measure the performance of the navigation and localization components.

1.3 Thesis Organization

In Chapter 1, we introduce the reader on the concept of benchmarking methodologies in robotics. We briefly describe different classes of methodologies, from commonly used but too simple for our purpose, to methodologies which aim at benchmarking complex systems but still not tested in practice. In Sec. 1.1, we propose a taxonomy of benchmarking methodologies. The taxonomy classifies methodologies with respect to four key aspects, allowing us to classify existing methodologies and define the aspects of the methodology we propose.

In Chapter 2, we review the scientific literature regarding benchmarking methodology. Firstly, we review works about experimental methodology, which is a fundamental aspect of scientific research, and even more so of benchmarking. Then we review works of benchmarking methodologies of specific areas of interest for the thesis: object detection and classification, 2D SLAM an localization, and autonomous navigation. We classify each benchmarking methodology within our taxonomy, highlighting their key aspects.

In Chapter 3, we introduce the proposed methodology. We define our objectives with respect to our taxonomy and motivate them by describing in more details how the proposed methodology improves and enables the evaluation of robot systems and components in research and engineering, including practical examples. In Sec. 3.1, we formalize the concept of performance model. In Sec. 3.2, we describe the software framework developed to support our methodology and how it is used to run experiments and produce the performance models. Finally, in Sec. 3.3, we describe and provide examples for the composition of performance models, which allows us to study complex robot systems.

In Chapter 4, we describe two use cases which we use to demonstrate our methodology. In the first case study, in Sec. 4.1, we show how to produce performance models for a single functionality, SLAM, and in the second case study, in Sec. 4.2, we demonstrate how we produce composed performance models for a complex system.

In Chapter 5, we analyse the experimental results obtained in the case studies, showing how to make use of the performance models and which decisions are supported by the results.

In Chapter 6, we draw our conclusions and propose possible future directions of our work.

CHAPTER 2

Background and Literature Review

2.1 Experimental Methodology

Experimental methodology is a fundamental aspect of research but lagging behind in robotics research if compared with other disciplines such as physics or medicine. Bonsignorio *et al.* [5, 6] highlighted the importance of repeatability and reproducibility of research, to support industrial adoption of new solutions and allow research groups to build on previous results. The authors point to some of the main difficulties in reproducing robotics experiments such as the high variability in hardware and software architectures of the robot systems, the high variability of the environments in which the systems or methods are tested, the need for specific test equipment to conduct the experiments, etc. Bonsignorio *et al.* also point to methodological issues affecting the reproducibility of research in robotics, such as the lack of experimental protocols and the lack of assessment of the statistical significance for the results. Additionally, they assert the need to provide the data and sufficient description of the hardware and software used in the evaluation of the system or method, the parameters, the environment, the task, as well as a precise description of the evaluation metrics and criteria. Bonsignorio and other researchers from specific fields participated to a Special Interest Group on Good Experimental Methodology and Benchmarking (SIG GEM) and produced a set of guidelines [7] regarding the review of experimental research in robotics and some of its sub-areas (e.g., SLAM, Motion Control, Obstacle Avoidance, etc).

Some works, such as [24,25], provide tools to support the complete development, reproduction and modification process of robot systems aiming to facilitate reproducibility of robotics experiments and provide a systematic approach to aggregate and link information on publications, version of component releases, systems, datasets, and so on.

Relatively recent developments in computer engineering tools and data sharing platforms can be used to alleviate the difficulty of replication of scientific results in robotics. Containerization tools such as Docker[1] greatly reduce the effort needed to run software setups developed by researchers and can ensure that a software setup can be run even when some of the software packages stop being supported or become available.

Code and data sharing platforms, such as GitHub[2] and Zenodo[3], allow researchers to share code and scientific data. Specifically, the data sharing service Zenodo allows to share data, even in large volumes, with the promise that it will remain available for an unlimited time, citeable using Digital Object Identifiers (DOI), and versioned. GitHub has a similar promise for open source code. The availability of datasets is not always to be taken for granted. For instance, the original data of the Radish dataset [19], which we used in our work, is not available from the original source anymore. The data used in our work has been found in alternative sources which republished the original dataset[4].

2.2 Benchmarking Object Detection and Classification Methods

The *KITTI Vision Benchmark Suite*[5] [16], *Pascal VOC*[6] [11] and *COCO*[7] [28] frameworks provide multiple datasets of images and ground truth data for computer/robot vision tasks such as object detection, visual odometry, etc. We classify these framework as *single component* with respect to our taxonomy since each benchmark assesses the performance of methods with no connection to a complex system. These frameworks provide a public leaderboard ranking algorithms by multiple performance metrics. In particular, in COCO the leaderboard shows the average performance of each metric for each method, allowing to identify the best performing method for each metric. In Pascal VOC, the leaderboard only reports one metric, average precision, for each class of the dataset. This allows to identify the best performing method for specific classes or the best performing method overall. In KITTY, specifically for the object detection tasks, the leaderboard reports the performance divided in three levels of detection difficulty, the detection time, and information on the computation device used (i.e., CPU/GPU, number of cores, frequency). The detection difficulty levels (i.e., easy, moderate, and hard), are based on the size, occlusion and truncation of the objects in the images.

Huang *et al.* [20] measure the trade-off between speed, memory and accuracy for different object detection Convolutional Neural Network (CNN) architectures with different architecture parameters. The approach is not meant to benchmark object detection methods *per se*, but provides a more complete characterization of the performance than common practice for object detection benchmark methodologies such as [8,11,28], where the dependency between detection speed, memory requirements, and detection accuracy is not reported in such level of details. In their work, many CNN architectures are built with different combinations of parameters: feature extractor, number of proposals, output stride, loss function, and input size. Some of these combinations, but not all, correspond to architectures that have been used in state-of-the-art object detec-

[1]`https://www.docker.com/`. Accessed on 2022-09-27.
[2]`https://github.com/`. Accessed on 2022-09-27.
[3]`https://zenodo.org/`. Accessed on 2022-09-27.
[4]`https://www.ipb.uni-bonn.de/datasets/`,
`http://www2.informatik.uni-freiburg.de/~stachnis/datasets.html`. Accessed on 2022-09-27.
[5]`http://cvlibs.net/datasets/kitti`. Accessed on 2022-09-27.
[6]`http://host.robots.ox.ac.uk/leaderboard`. Accessed on 2022-09-27.
[7]`http://cocodataset.org/#detection-leaderboard`. Accessed on 2022-09-27.

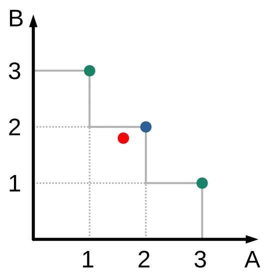
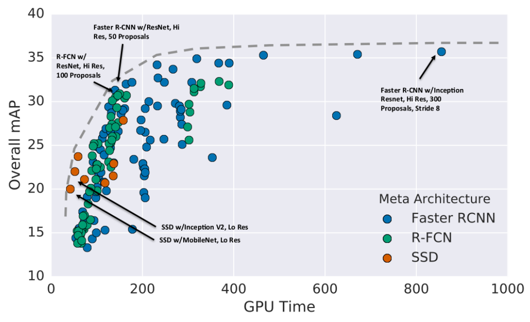
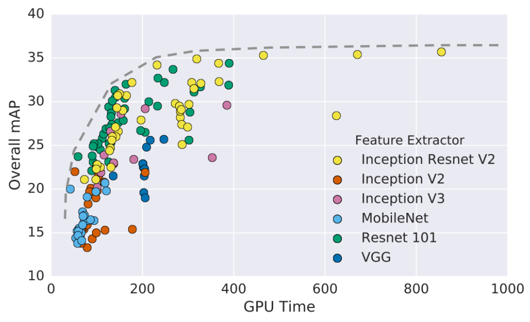


Figure 2.1: *Simplified example of trade-off curve (gray continuous line) for two metrics, A and B (higher values are better). Green dots have the best performance for the metrics A and B. The blue dot represents a performance which is not best in any metric, but can still be a good compromise. The red dot is not a good compromise, since its performance is worse than the blue dot for all metrics.*



(a)



(b)

Figure 2.2: *Reprinted from [20]. Overall mAP (higher is better) vs GPU time (lower is better) colored by (a) meta-architecture and (b) feature extractor. Each (meta-architecture, feature extractor) pair can correspond to multiple points on this plot due to further parameter changes.*

tors. The results allow the authors to find the trade-off curve, which identifies the set of architectures with competitive performance when all metrics are considered. These are the architectures which are not outperformed for every single metric by another architecture, and are thus a valid compromise. See Fig. 2.1 for a simplified example of the trade-off curve, and Fig. 2.2 for an example of the trade-off curves presented in [20]. The architectures which do not produce the best performance for any metric, but are still good compromises, would not be highlighted in a ranking based on single metrics.

2.3 Benchmarking Complete Robot Systems

The following two works present methodologies which tackle the problem of evaluating the performance of software components in relation to each other in the context of complete robot systems, although they do not produce practical developments. Competitions such as RoboCup[8], RoCKIn and the European Robotics League (ERL) aim at evaluating entire robot systems. The ERL competition [4,27] specifically aims at integrating a benchmarking methodology with a robot competition. To do so, two types of challenges are organized: functionality benchmarks and task benchmarks. The functionality benchmarks evaluate the performance of single functionalities: object perception, navigation, people perception, person following, grasping and manipulation. These benchmarks are executed in isolation to measure the performance of the robot system without interference from the performance of other functionalities. Task benchmarks evaluate the performance of the integrated robot system executing tasks that require multiple functionalities. The evaluation is done in testbeds that replicate an apartment or factory layout, with all its environmental aspects, like walls, windows. The methodology proposes to quantify the impact of each functionality on the performance of the robot system in each task using the scores obtained in each benchmark challenge.

These competitions also have some limits. The competition setting, indeed, offers limited repeatability as the robot systems (hardware and software) are frequently changing, even within a single competition event. Because of the limited repeatability, it is too complex to study the relationship between the performance of a complete robot system, the performance of the system's functionalities and the characteristics of the environment. We classify this methodology as *complex system* since the competition provides both system-level and functionality-level benchmarks in order to assess the integration quality of the components. With respect to our taxonomy, we classify the methodology's subject type as complex system since both single components and the entire robot system are evaluated, and the methodology proposes to analyse how the functionalities affect the overall system performance. We classify the target use as ranking since the competition results are published as a ranking at the end of each season, for each functionality and task benchmark. The environment type is real world and the execution mode is closed-loop.

The methodology presented in Plug and Bench [3,12,13] developed a formalism that integrates benchmarking methodology with the Model Driven Software Development (MDSD) concept of software component in the RobMoSys ecosystem[9]. The work proposes to develop benchmark procedures and performance metrics for each component

[8]`https://www.robocup.org/`. Accessed on 2022-02-02.
[9]`https://robmosys.eu`

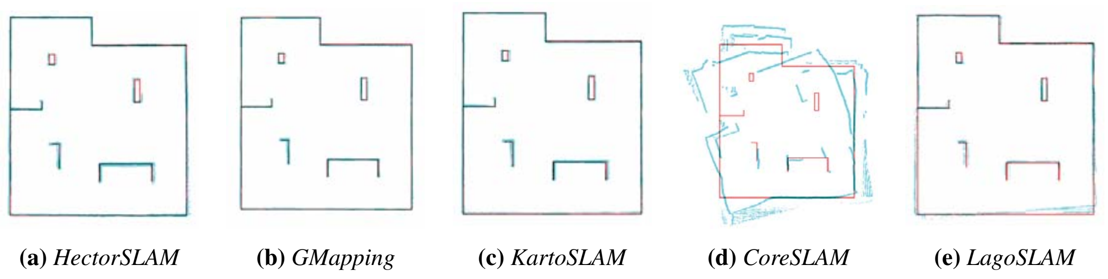

(a) *HectorSLAM* **(b)** *GMapping* **(c)** *KartoSLAM* **(d)** *CoreSLAM* **(e)** *LagoSLAM*

Figure 2.3: *Reprinted from [41]. Maps obtained through simulation in the MRL arena environment. Red represents the ground truth and blue represents the final map.*

available in the framework's library. This would be possible thanks to the standardized interfaces of each component in the software framework used in RobMoSys. The benchmark of each component would be run by the developer of the software component. The developer of the robot system, which integrates software components produced by third parties, can than use the benchmark results of each component to estimate its performance given characteristics of the robot platform and characteristics of the environment. Additionally, the authors propose as future work the possibility of composing the benchmark results of different components in order to estimate the performance of the whole system at design time. They suggest this is possible provided that:

- Each component of the robot system has been benchmarked;
- The performance of each component can be fully characterized by information regarding the robotic platform and environment;
- The performance of each component which depends on other components can be fully characterized by information regarding those components.

We classify the methodology's target subject as complex system since the methodology is centered on the idea of predicting the performance of software components taking into consideration their dependence. The target use is technical specification since the results of the benchmark the software components are conceived to allow predicting the performance from characteristics of the robot system, other components, and environment. The environment type and execution mode can be any since the formalism specifies how the benchmarking process is integrated into the RobMoSys framework, rather than specifying a specific protocol.

2.4 Benchmarking 2D SLAM and Localization Methods

The work of [41] benchmarks five SLAM algorithms in two simulated environments and one real world environment. The performance is evaluated with two metrics: map accuracy, which computes the difference between the ground truth map and the one estimated by the methods, as shown in Fig. 2.3, and CPU load. We classify the methodology's subject type: as single components since the performance of the SLAM methods is computed in isolation from other components or features of a robot system which may influence their performance. The methodology's target use is ranking, since the benchmarked methods are compared by the two performance metrics. The environment type is simulation, since the work is applied to a simulation and the real world.

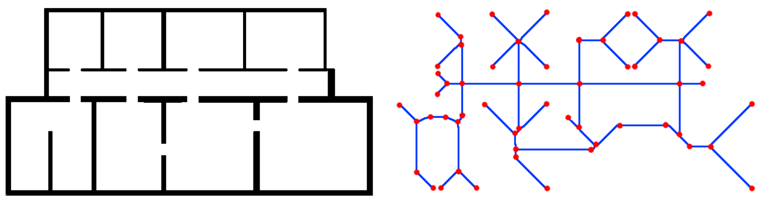


Figure 2.4: *Reprint from [29]. A floor plan (left) and the corresponding Voronoi graph (right; cells are red and blue pixels)*

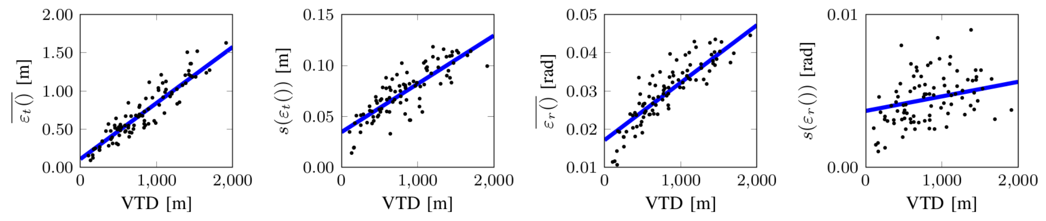


Figure 2.5: *Reprint from [29]. The regression lines representing the models that correlates the Voronoi Traversal Distance (VTD) with the four components of the localization error (black dots are values relative to individual environments in E) for maps obtained using the GMapping SLAM algorithm.*

Although the environment type could also be classified as dataset since the SLAM methods are fed the odometry and LIDAR sensor data which was recorded in the real and simulated environments. For the same reason, the execution mode can be classified as either open-loop or closed-loop.

The works of [2, 29] have introduced a methodology that exploits simulations to generate a large number of test data on which SLAM algorithms are automatically evaluated. The authors apply their methodology to the SLAM methods GMapping and KARTO. The simulated robot has realistic characteristics, such as noisy odometry readings and a LIDAR sensor with properties corresponding to a commercial product. A large number of heterogeneous environments is used in their experiments. From the collected data, the authors are able to extract a statistical relationship between the localization error and different geometrical features of the environment. In Fig. 2.4 we report the Voronoi graph computed on the grid-map of the environment which the authors use to compute geometrical features of the environment, such as the Voronoi Traversal Distance (VTD), which is the distance of the path of a robot traversing the graph in order to visit every node. In Fig. 2.5 we report the data and statistical model obtained in [29] regarding the GMapping SLAM method. The localization error is measured with four performance metrics measuring the mean and variance of the translation and rotation relative localization error. The effect of the properties of the sensors on the performance are not investigated by their work. With respect to our taxonomy, we classify the methodology's subject type as single component since the performance of the SLAM methods is computed in isolation from other components or features of a robot system which may influence their performance. We classify the target use as technical specification since the performance of the SLAM methods is evaluated against different features of the environment. The environment type is simulation, since the work is applied to a simulation, although in principle the methodology can also be applied to experiments executed with a real robot and from datasets since the SLAM method only requires the odometry and LIDAR data, in fact the authors include experiments from datasets recorded on real robots to validate the simulation results. For the same reason, the execution mode can be both open-loop or closed-loop.

2.5 Benchmarking Autonomous Navigation Methods

The following works tackled the problem of measuring the performance of navigation algorithms in relation to characteristics of the environment. No work showed the effect of other components of a system on the performance of navigation methods.

An evaluation framework for measuring the performance of a navigation stack for autonomous robots in relation to a measure of the environment's complexity is presented in [10]. The complexity of an environment is computed as the average of the shortest path distance computed between every pair of accessible locations in the environment. The authors show that the complexity of the environment has a considerable impact on the performance and it is, therefore, useful to consider this relationship when evaluating the performance of navigation methods. Three environments and six start/-goal positions (two start/goal positions for each environment) are used to evaluate the relation between performance and environment complexity. Only one navigation software is evaluated, the Husky ROS navigation stack [10]. Although the work proposes a complexity metric rather than a benchmarking methodology, we can still classify the proposed methodology with respect to our taxonomy. The subject type is a single component, since a navigation stack is evaluated as a monolithic component. The target use could be classified as technical specification since the approach could evaluate the impact of the environment complexity on the performance of different navigation components. Although this would require to compare the dependency for multiple navigation components. The approach can be applied to a simulation or to a real world environment, so we can classify the methodology's environment type as simulation and real world, and the execution mode as closed-loop.

In the work of [21], a local planner based on Deep Reinforcement Learning (DRL) and three other local planners available in the ROS framework (DWA, TEB, and MPC) are evaluated in different environments. The difficulty of the navigation task is set by varying the number and velocity of dynamic obstacles, representing people walking in the environments. The results quantify the effect of the dynamic obstacles on the performance of each local planner. The localization and odometry readings of the robot are assumed to be ideal, meaning that the ground truth pose of the robot is provided to the navigation methods, rather than computing an estimated pose with a localization method. With respect to our taxonomy, we can classify the methodology's subject type as single components, since the effect of the localization on the performance of the local planners is ignored. The methodology's target use is technical specification, since the methodology allows to estimate the dependency of the performance on the difficulty of the navigation task caused by the dynamic obstacles. The methodology relies on simulation to apply specific properties to the dynamic obstacles, therefore the environment type is simulation, and the execution mode is closed-loop.

The work of [18] presents a benchmarking tool for sampling-based global planners. The environments used to evaluate the global planners are generated as indoor

[10] `http://www.clearpathrobotics.com/assets/guides/kinetic/husky/HuskyGmapping.html`. Accessed on 2022-09-26.

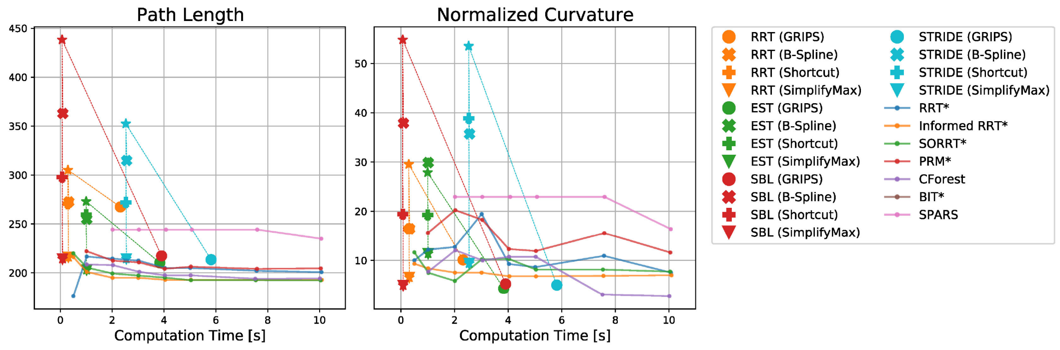


Figure 2.6: *Reprint from [18]. Path length and normalized curvature of different combinations of the feasible motion-planning algorithms and post-smoothing algorithms compared against the asymptotically (near) optimal motion-planning algorithms. Both performance metrics are reported with different computation time limits. The initial paths of the feasible motion-planning algorithms are marked with ★, and the paths of the post-smoothing algorithms are marked with • for GRIPS, × for B-Spline, + for Shortcut and ▼ for SimplifyMax. The paths of the asymptotically (near) optimal motion-planning algorithms are solid lines marked with ·.*

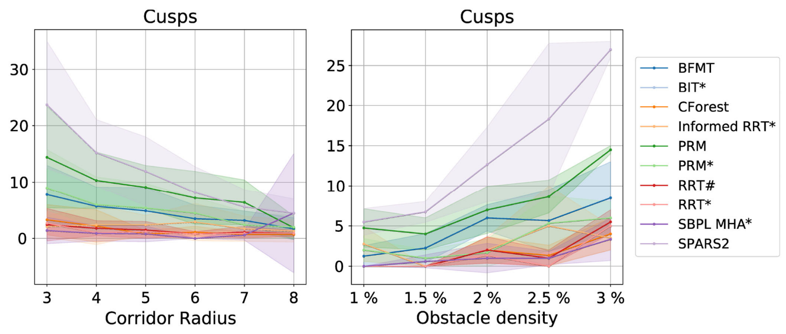


Figure 2.7: *Reprint from [18]. Number of cusps for different global planners with a computation time limit of 15 seconds each in 5 random indoor-like environments with desired minimum corridor widths from 3 to 8 cells in increments of 1 cell (left) and 5 random outdoor-like environments with obstacle density ranging from 1.0% to 3.0% in increments of 0.5% (right).*

or outdoor with two parameters which set the difficulty of the global planning: minimum corridor width and obstacle density. The environment difficulty is related to the performance measured as the number of cusps present in the generated plan, see Fig. 2.7. The trade-off curves for some performance metrics are shown in the paper for different combinations of motion planning algorithms, post-smoothing algorithms, and computation time limits, as shown in Fig. 2.6. A limitation of this work is that only static environments are considered. With respect to our taxonomy, we can classify the methodology's subject type as single component, since no interaction with other components of a robot system is considered. The methodology's target use is technical specification since the performance dependency on the difficulty of the environments is estimated for different planning algorithms. The data used to evaluate the algorithms consists in grid-based maps, therefore we can classify the methodology's environment type as dataset, and the execution mode as open-loop.

CHAPTER 3

Performance Modelling Methodology

We can frame the objective of this thesis with respect to the taxonomy described in Sec. 1.1 into a benchmarking methodology that falls into the categories of technical specifications and complex system benchmarking. Complex system benchmarking methodologies aim at measuring the performance of a subject structured as a system and its components. Such methodology evaluates the effect of the performance of one component on other components, or the effect of features of the robot system on the performance of the components under evaluation. Technical specifications benchmark methodologies provide the ability to predict the performance as a function of relevant parameters such as characteristics of the robot system and characteristics of the environment.

Taking into consideration the characteristics of the system and environment allows us to determine what influences the performance of the component and the conditions necessary to maximize its performance, or more realistically, to obtain a sufficient performance given constraints on the robot system such as its cost. As a practical example, the performance of a 2D SLAM component depends on the information produced by the odometry and LIDAR sensors. As we will show in Sec. 5.1.1 (Fig. 5.2a and 5.2b), we find that the performance falls off when using a LIDAR sensor with field of view and range lower than certain values. This information is key to choose a sensor with characteristics which allow the SLAM component to maintain a sufficient performance, and at the same time, minimizing the cost of the system by considering the diminishing returns of a more expensive sensor that only produces a small performance increase.

When comparing the performance of different components, a technical specification methodology allows us to identify in which conditions one component performs better than the others. For instance, in Sec. 5.1.1 (Fig. 5.2b), we find that when using a LIDAR sensor with low field of view, the SLAM component with the best localization performance is GMapping, and when using a LIDAR sensor with high field of view, the best localization performance is obtained by SLAM Toolbox.

Benchmarking a robot system by assessing the performance of each component with independent single component benchmarks would not capture the gain or loss of performance of the system due to the interaction between components. Components can interact directly, when one component relies on information generated by another component, or indirectly, when a component is affected by the actions of another component. The performance of a component which relies on information generated by another component, may - in general - be affected by some characteristics of this information. A practical example of this occurrence, which we will show in Sec. 5.2.2 (Fig. 5.19), is found in a robot system which includes a localization component and a local planning component. The local planning component is responsible for following a path by sending motion commands while avoiding obstacles. The local planning component relies on the estimated position and orientation of the robot in the environment, which is provided by the localization component. We find that, in some cases, the probability of success of the local planning component is affected by the accuracy of the estimation error of the localization component.

The study of the performance of multiple components can also provide a more significant performance measure. For example, a mapping component can be benchmarked with a general performance metric measuring the similarity between the estimated map and the ground truth [41]. This measure may generally tell the quality of the estimated map, but if the map is utilized by another component, the quality of the map should be judged by how well it supports the execution of the other component. In the work of [23], the authors point out that a map may be locally or globally consistent. A globally consistent map will achieve a higher performance when measuring the similarity of the estimated map with the ground truth map. A map which is locally consistent but not globally consistent, will achieve a lower performance with the similarity metric, but it may still be adequate for a navigation component if the topology of the environment is preserved.

By applying the technical specification and complex system benchmarking, we postulate that the overall system performance can be measured in relation to the performance of its components and characteristics of the environment. The result of unifying the technical specification and complex system benchmarking is what we call *component performance model*: a statistical model which allows us to predict the performance of the software components from features of the system and the environment. In robotics research, such methodology allows us to evaluate comprehensively the methods against the state-of-the-art, and in engineering, it can be used as a tool to improve the development of robot systems.

3.1 Formal Definition

Let us define the terminology of functionality and component. A *functionality* identifies a set of abilities required by the robot system to work. The definition of what constitutes a specific functionality is arbitrary. Benchmarking of robot systems can and should take advantage of the componentization that comes into existence from the development of the software packages in the robot frameworks. Since we base our work on the ROS framework, we make the definition of specific functionalities coincide with the structure that came into existence in the ROS framework. For example, the most basic and common feature of a mobile autonomous robot is the ability to localize itself and navigate in an environment, which we divide into the following (non exhaustive) set of functionalities: localization, SLAM (Simultaneous Localization And Mapping), global planning, and local planning. These functionalities include different abilities that may be considered separate in different fields of research or even different software frameworks. For instance, the local planning functionality may be split into further sets of abilities such as motion planning, motion control and obstacle avoidance. We define a *software component* as the software implementation of a method or algorithm which provides a specific functionality. For instance, in the ROS framework, different software packages are available to provide the localization functionality, such as the AMCL and the SLAM Toolbox software packages.

We define a *component performance model* as a set of statistical models able to predict the value of the software component's performance metrics from the characteristics of the system in which the component is used, the characteristics of the environment in which the system is deployed, and the configuration of the component itself.

We call *system features* those features representing characteristics of the robotic hardware and software system that affect the performance of a software component. For instance, the performance of a local planner may depend on the quality of the localization as computed by a localization or SLAM component, which in turn, depends on the number of beams of a LIDAR sensor. In this case, the system features would be: quality of the localization and number of LIDAR sensor beams.

In our approach, we call *environment features* those values representing some characteristic of the environment that we can compute and which affect the performance of the component. For instance, the performance of a Simultaneous Localization and Mapping (SLAM) method depends on the environment's clutterness, size, or richness in corners. While the motion commands of a local planner are affected by the obstacles surrounding the robot, the tightness of the passages, and the dynamics of the obstacles. In this case, the environment features would include: environment clutterness, mean or minimum width of the passages, velocity and number of dynamic obstacles.

While developing performance models for different functionalities we identified the parts of our methodology that apply generally. These are represented as entities and relationships in the diagram in Fig. 3.1. The entities are represented by rectangles. The relationship between entities is represented by edges. The cardinality of a relationship from entity A to entity B, labeled as $n : mR$, indicates that A has relationship R with m B entities, and entity B has relationship R with n A entities. The symbol "*" in the edge labels indicates a number greater or equal to 0. This diagram can be used as a meta-model, meaning that it can be used as the definition of the diagrams describing specific performance models. The entities of the meta-model are classes, while in the diagram of a specific performance model the entities are instances. Examples of specific performance model diagrams will be shown in Sec. 4.1 and Sec. 4.2.

The entities of the meta-model can be divided in two groups, those representing the protocol and those representing the data, which is generated using the protocol. The following are the entities representing the protocol:

- an *experimental setup*, consisting in the implementation of the *experimental sys-*

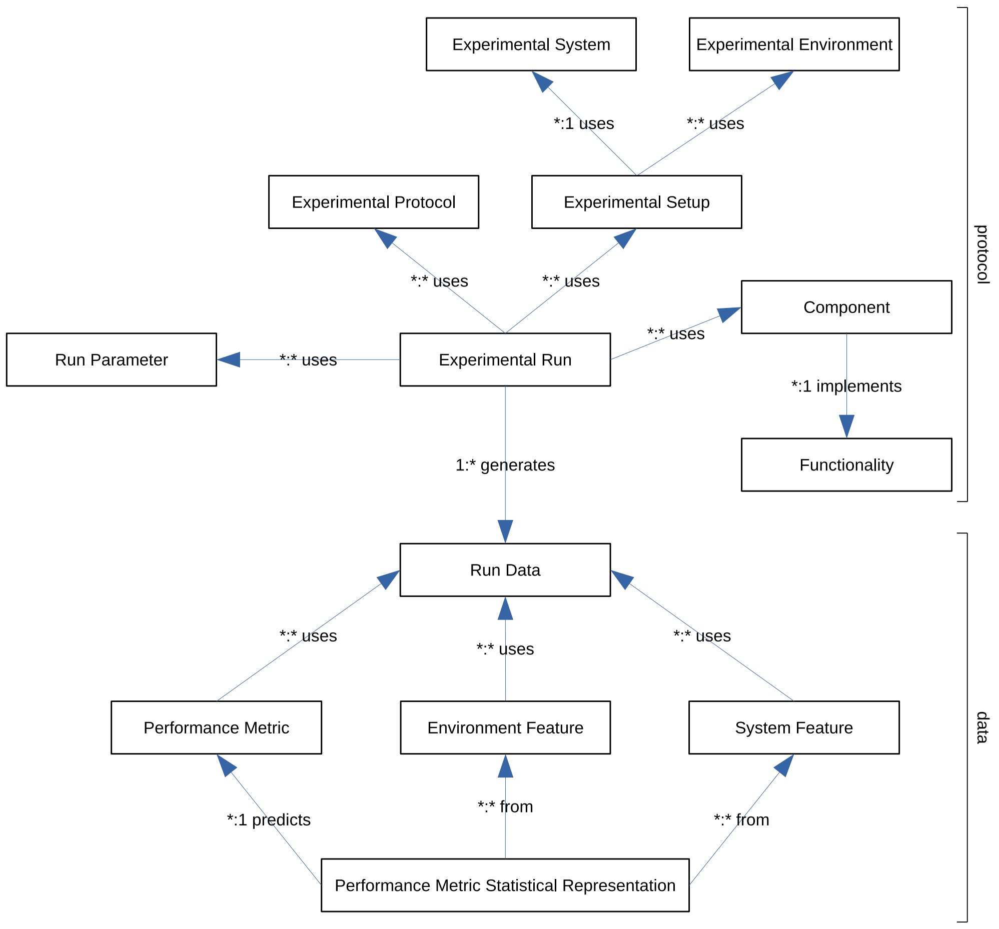


Figure 3.1: *The meta-model representing the relationships between the entities (rectangles) constituting a performance model. The relationship between entities is represented by edges.*

tem and *experimental environments*;

- an *experimental protocol*, which identifies the actions executed by the system and the behavior of the environment;
- a set of *run parameters*, which control aspects of the experiments that are relevant to the performance of the component;

The following are the entities representing the data of the methodology:

- a set of *run data*, which are collected during the experiments and used to compute the performance metrics and features;
- a set of *environment and system features*, which are quantifiable characteristics of the environment and system that might influence the overall performance;
- a set of *performance metrics*, which provide a measure of the performance of the component that can be used to compare different implementations;
- a set of *performance metric statistical representations*, each of which allows to predict the value of a performance metric from the relevant features.

In the following paragraphs, we describe each entity in more detail and provide examples.

Experimental Setup The *experimental setup* describes how we collect the data regarding the component we wish to model and it manages the execution of experiments. One *experimental system* and one or more *experimental environments* are used in the setup to simulate the conditions in which the component is deployed in the real scenario. The experimental system represents the software residing in the robot system. Multiple experimental environments may be used by an experimental setup to test a variety of environmental conditions. The experimental environment may be a real setting, based on a simulation or based on a dataset. While experiments conducted in the real world could in some cases provide more realistic performance measurements, the use of datasets or simulation enables to execute a high number of experiments, which makes it possible to collect more data. Additionally, the use of datasets and simulation simplifies the collection of ground truth information, which in some cases may not be at all possible or too expensive in a real setting. For instance, continuously and accurately measuring the pose of a robot in a large indoor environment would require an expensive motion capture system, while in simulation it comes for free. The experimental setup of [4] uses multiple real world environments, implemented in different testbeds. The pose of the robot is measured using a visual marker fixed to the robot platform and a motion capture system using cameras to observe every location of the environment. An example of experimental setup is the one used in the first use case demonstrating our methodology, in Sec. 4.1. The experimental setup is used to evaluate different SLAM components by running them in a robot system which visits every location of multiple simulated environments. The experimental environments are simulated buildings represented by 3D models. The experimental system includes the navigation software that commands the simulated robot platform in order to visit every location of the environment and the sensors providing the information to the component under evaluation.

Experimental Protocol The experimental protocol defines how the environment behaves (e.g., defining the dynamics of an environment with moving obstacles) and how the experimental system interacts with the component and with the environment. The protocol should ensure that the component is faced with realistic events and behaviors. Since the experimental protocol affects the performance of the component, it is important to provide a description that allows the user of the performance model to evaluate if the results are compatible with their use of the component. Using our first use case as example (Sec. 4.1), the experimental protocol consists in the procedure used to automatically generate a list of locations of the environment which are visited in a specific order. Additionally, the experimental protocol defines the conditions for the termination of the experiment.

Experimental Run and Run Parameters We organize the experiment execution in runs. We define an *experimental run* as an instance of the experimental system and experimental environment configured with a specific set of *run parameters*. The run parameters, to which we assign different values in each run, are used to set every aspect of the experimental setup which may produce a change in performance of the software components under evaluation. For instance, when evaluating a component that relies on information from a sensor, which has different properties affecting the quality of the information, we would use a run parameter for each relevant property of the sensor and assign to them different values that could be realistically found in a real robot system. Other examples of run parameters, which we will use in the demonstration of the proposed methodology, are the type of kinematic model of the robot platform, kinematic constraints related to each kinematic model, specific properties of multiple sensors of the robot, and run parameters which affect the configuration of the components themselves. Run parameters are also used to set lower level aspects of each run. If we are interested in evaluating multiple components implementing the same functionality and in the same experimental setup, we would use a run parameter to select which of multiple components are used in the experimental run. If we test multiple components in the same experimental setup, we use a run parameter for each functionality for which we have multiple component choices. If multiple experimental environments are available, we use a run parameter to select which one to use in each run. Additionally, we can repeat multiple times the execution of runs with the same configuration of the experimental environment and experimental system by using a run parameter that does cause changes in them, which can be useful to test the variability of the results in the same conditions. Since we do not yet know the effect of each experimental setup configuration generated by the run parameters on the performance, we must, in principle, execute a run for all combinations of run parameter values. This would result in a large number of runs: with n run parameters to which we assign m values each, the number of combinations to run is m^n, so we need to reduce the number of run parameters and values. To do this, we can execute different sets of runs with limited number of run parameters and/or limited number of values, observe the effect on the performance, and select which run parameters and values are actually significant. Additionally, we can start with a small set of run parameters values, apply the whole methodology end-to-end, observe whether the results provide sufficient details, and iteratively add more values to each run parameter as needed.

Run Data The *run data* is data collected during the execution of the runs, and is an intermediate step in the computation of the performance metrics, the system features, and the environment features relative to each run. The run data contains all the information necessary to compute the metrics and features, such as the values of the run parameters used to setup the experimental system and environment, events that happened during the execution of the run, and the state of the environment and system through time. This allows to make changes to the implementation of the performance metrics and the algorithms that compute system features and environment features, even after the run data has been collected. Collecting the run data also allows us to implement additional metrics and features from the collected data. For instance, to compute the localization error of a localization component, which estimates the position and orientation of the robot with respect to the environment's frame of reference, we collect two run data. The first run data contains the ground truth position and orientation of the robot at different times. In case of a simulated environment, this data is provided by the simulation software every time the state of the simulation is updated. The second run data contains the position and orientation estimated by the localization component, and the time at which it is provided by the localization component. These two run data can then be compared to compute the error of the estimation with respect to the ground truth information throughout the execution of the run.

Environment and System Features The *environment features* and *system features* are values representing some characteristic of the environment, robotic hardware and software system that affect the performance of the software components. The system features and environment features are computed from the run data. The features may correspond to run parameters that define characteristics of the system/environment relevant to the performance of the component, or they may be computed with algorithms from information regarding the environment and information available to the component during the execution of the run. environment features and system features in our methodology are essential to find the statistical relationship between the performance of the components and the characteristics of the system and environment. This is essential to comprehensively evaluate the software components of robot systems that we are most interested in benchmarking, i.e., the components developed as a general solution to complex problems, such as SLAM, trajectory planning, object detection, and so on. These components have to be adaptable to a variety of conditions, as they most likely will face environments and systems which present different degrees of difficulty. We postulate it is possible to describe the difficulty of the problem solved by the component in terms of environment features and system features. The difficulty of the problem solved by methods and algorithms is not always explicitly taken into account, even when it could provide a more thorough evaluation, such as the works of [11,28,41]. A few methodologies study the difficulty of the problem and use it to categorize the performance in different classes, such as [16], in which the performance of object detection methods is measured for three levels of detection difficulty based on the size, occlusion and truncation of the objects in the images; the work of [10], in which the difficulty of the navigation problem is estimated for three environments; and the work of [45], in which local planners are evaluated in different environments that are designed to pose different levels of difficulty. And finally, a few methodologies propose to study the relationship between the performance and the difficulty of the problem or characteristics of the environment, such as the work of [21], which tests different local planners against environments with dynamic obstacles producing different levels of difficulty and shows that different local planners perform best in more or less dynamic environments; and the work of [29], which produces statistical models of the relationship between the performance of SLAM algorithms and characteristics of the environmet in which they operate. In our methodology, we exploit these features via a statistical model, which allows us to study how a component performance is affected by them and to find what are the operational limits of a component while giving us information on how to change the system and environment to improve its performance.

Performance Metrics *Performance metrics* measure the performance of the software components from the run data collected during the experimental runs. The measurement of the performance is then used to evaluate whether each performance metric is affected by environment features and system features, and in such case, to build a statistical model that allows us to predict the performance from the features. In general, multiple performance metrics are required to provide a comprehensive evaluation of a robotics software component. Providing a statistical model able to predict the performance for a multitude of aspects allows us to maximize an objective function that weights each performance metric based on our needs. Different users of a component may care about different aspects of its performance. For instance, a mobile robot motion planning component may produce paths that maintain a higher safety distance between the robot platform and surrounding obstacles, but these paths will be longer, take more time to execute, and result in lower velocity. When the component is used in different applications, one performance metric may be considered more important than another: in an environment shared with humans, safety may be prioritized, while in a more controlled and predictable environment, efficiently executing the task may be the primary concern. An example of the usefulness of evaluating methods or algorithms with multiple metrics is the work of [20], which measures the trade-off between three performance metrics, speed, memory and accuracy, for different object detection CNN architectures, allowing us to identify the architectures with competitive performance when all metrics are considered, rather than the architectures which perform best for each metric by itself.

Performance Metric Statistical Representation We are interested in producing statistical models predicting each performance metric of a component from features of the system and the environment. The *performance metric statistical representation* is a statistical model predicting a specific performance metric from certain environment features and system features, and for specific conditions. Each performance metric can then be predicted by multiple *performance metric statistical representations* using different statistical modelling techniques, from different sets of features and for different conditions. The performance model of a component contains multiple performance metric statistical representations. Multiple statistical representations of the same performance metric may be useful when different representations have different advantages. For example, a performance metric which depends on multiple features can be represented with a set of univariate statistical models, each predicting the performance from a single feature, or it can be represented with a multivariate statistical model which predicts the performance from all the features at once. The univariate representations allow us to have an overview of the dependency from each feature which is easier to visualize and study, while the multivariate statistical model allows us to actually predict the performance given the value of the features, but the predicted values are more difficult to visualize and interpret due to the multidimensional nature of the data. Additionally, we may want to provide statistical representations of a performance metric computed in different conditions, which is equivalent to produce the statistical models only considering specific ranges of values for each feature. For example, a system feature may determine the type of robot platform and different components implementing the same functionality may only be compatible with some of these platforms. In such case, it is convenient to produce a different statistical representation which considers each platform separately, allowing us, for each platform, to compare the performance of the components and find which component performs best. At the same time, it is also convenient to compare the performance of each component when using different platforms and determine the best platform given the component.

3.2 Performance Modelling Software Framework

To support our methodology, we developed the performance modelling software framework, which automates the execution of the experiments, the collection of the data, and the computation of the performance metrics and the features used to produce the performance models. Since we want to evaluate the performance of software components over an extensive number of run parameter combinations, manually managing the execution of the experiments would be unfeasible, therefore we automate the execution of runs and the collection of data. The automation of the experiments also allows us to make our work more reproducible by packaging in software containers, such as Docker containers[1], the data and software needed to re-execute all the experiments and, in principle, obtain equivalent results.

The software framework is organized as a pipeline, shown in Fig. 3.2. A set of run parameters and their values are used to create the list of combinations of run parameter values by the *grid executor*. Each combination of run parameter values is passed to the *run script*, which uses them to configure the experimental setup and execute the experimental run. The experimental setup contains the experimental system, the experimental environment and one or multiple components we want to evaluate. The experimental system includes all the software necessary to execute the run: the *run data logger*, which collects the run data; the *supervisor*, which coordinates the execution of the run; and the *accessory software*, which fulfills the functions of a robot system needed to evaluate the components. For instance, in order to evaluate a SLAM or localization component, the accessory software includes a navigation stack which controls the robot motion and the software needed to provide the information from the sensors. The *performance and features computation script* computes the metrics and features from the run data for all runs that have been completed. This script can be executed even while runs are being executed, allowing us to compute partial results for the statistical models while experiments are still in progress. The statistical models and other information

[1]`https://www.docker.com/resources/what-container/`. Accessed on 2022-10-13

run	param_1	param_2	param_3
1	1	A	
2	1	B	
3	2	A	
4	2	B	
5	1		0.1
6	1		0.2
7	2		0.1
8	2		0.2

Table 3.1: *List of combinations produced by the grid executor from the parameters in Listing 3.1.*

regarding the experiments are computed using Jupyter notebooks[2]. The time-sequence diagram in Fig. 3.3 shows the sequence of interactions between the parts of the software framework throughout the execution of multiple runs. In the following paragraphs, we provide more details on the parts of the software framework.

Grid Executor Because of the non-linearity of the performance in function of the run parameters, we need to test exhaustively the system for all combinations of run parameter values. The grid executor takes a configuration file containing a list of run parameters and respective list of values, which we call *run parameter grid*, and generates the list of all parameter-value combinations. For instance, in Listing 3.1 we have three run parameters, param_1, param_2, and param_3, with two values each. The grid executor produces the 8 combinations of run parameter values in Table 3.1. Each row of the table corresponds to the run parameters used to configure an experimental run. Additionally, the grid executor takes care of only running the runs with parameter combinations that have not been already executed. This allows to add parameters and values to the input file and run the grid executor to extend the tested combinations.

```
combinatorial_parameters: [
  {
    param_1: [1, 2],
    param_2: [A, B],
  },
  {
    param_1: [1, 2],
    param_3: [0.1, 0.2],
  },
]
```

Listing 3.1: *Example of run parameter grid, which is used as input of the grid executor.*

Run Script The grid executor passes a set of run parameters and their value to the run script. The run script has two functions: configuring the run and launching the experiment. The run is configured by creating a *run directory* in which all configuration files (run configuration in Fig 3.3) and information relative to the run are stored (including, for instance, the run data). All information needed to configure the run is stored in the run directory, in this way it is always possible to determine how the system was configured and replicate the run in the same conditions. Additionally, the list of all software

[2]`https://jupyter.org/`. Accessed on 2022-10-13.

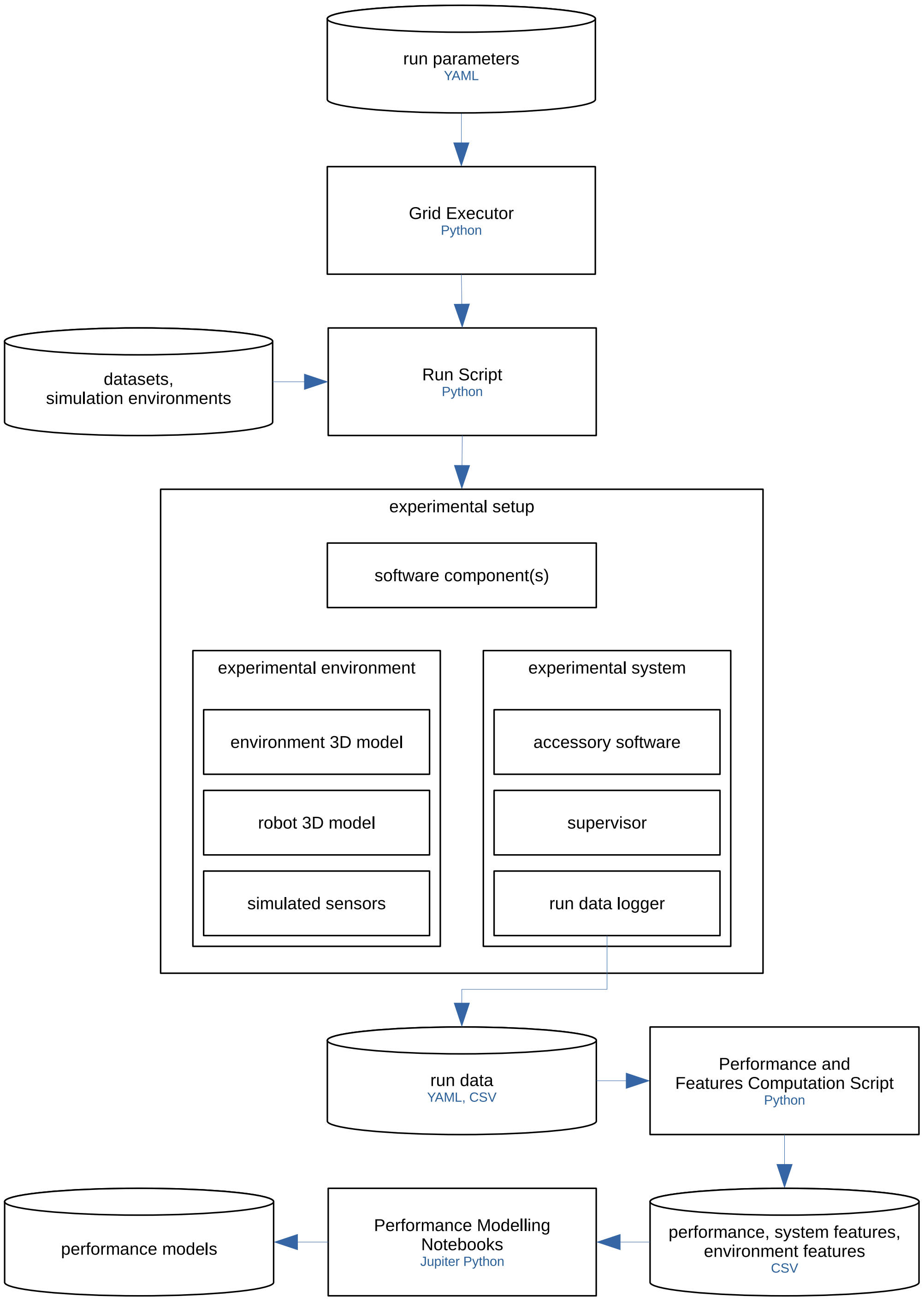


Figure 3.2: *Overview of the performance modelling software framework, developed to support our methodology by automating the execution of the experiments, the collection of data, and the computation of the performance metrics, environment features, and system features.*

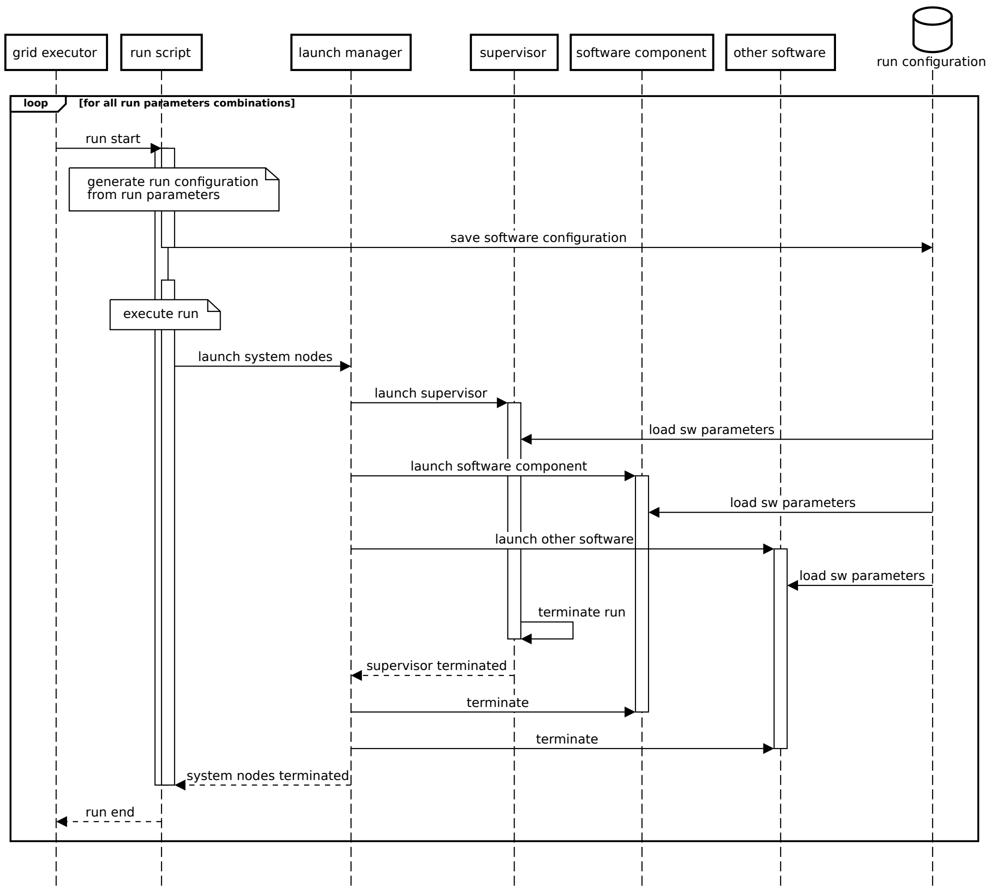


Figure 3.3: *Time-sequence diagram describing the sequence of events in a session of experiments as implemented in our software framework.*

packages installed in the system or used in the run is included in the run directory, allowing to keep track of which software versions are used in each run. The run is executed by launching the required software packages, namely, the software component being tested, the simulation software if the run is executed in simulation, or the dataset software if the run is executed from a dataset, the supervisor and data logger, and any other software needed to implement the experimental system. The run is considered terminated when the supervisor terminates.

Supervisor The supervisor is the part of the experimental setup which coordinates the software component and other software packages of the experimental setup in order to implement the experimental protocol. This includes sending commands to the software component and deciding when to terminate the run.

Run Data Logger The data logger writes the run data to files, which are saved in the run directory. The run data is all the information required to compute the performance, system features and environment features data relative to the run. The run data is raw information, such as events that occurred during the run, the trajectory of the robot, ground truth information from the simulator or dataset, etc.

Experimental Environment The experimental environment can be implemented as a simulator, a dataset or a real-world setup. In the case of simulation, the experimental environment consists of the implementation of the surrounding of the robot system. A simulation can provide different representations of the environment based on the fidelity required to accurately measure the performance of the software components. For instance, the environment used to evaluate a robot that autonomously navigates in a building using a LIDAR sensor may be represented with a 3D model of a building constituted by a flat floor and flat walls, while a robot which navigates relying on images from a camera sensor requires a more sophisticated simulation, such as the one proposed in the work of [46]. The interface between the environment and the robot system consists of the physical simulation of the robot and the simulated sensors, which can also provide different degrees of realism. The physical model of the robot is represented in the simulation by a 3D model, including geometric and kinematic information, actuator models, and sensor models. The simulated sensor models are implemented by algorithms which replicate the measurements and information that would be generated by the sensors in a real robot system, and can approximate the errors of real sensor measurements. Similarly, the actuators are represented by algorithms simulating the physical behavior of the commands sent by the robot system and can approximate limitations present in a real robot system.

3.3 Performance Model Composition

We define performance model composition as joining component performance models to obtain a performance model of a complex system. This allows us to evaluate multiple components in a complex robot system, to analyze the dependency of the performance between components, and to predict the performance of a complex system given the choice of which component is used for each of its functionalities.

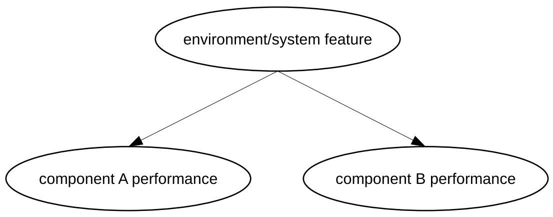


Figure 3.4: *Bayes network in which the performance of two components are conditionally independent.*

Different aspects can influence the performance of a component in a robot system. The performance of a component can depend on the characteristics of the information generated by other components and from the actions executed by other components. We use performance metrics to characterize any aspect of a component A that can influence another component B. For the component performance model of B, these metrics are considered system features.

We identify three general cases of how we can model the dependency between the performance metrics of one component with respect to another. The dependency of the performance metrics on other performance metrics, system features, and environment features is represented using Bayesian networks [33], where the performance metrics and system/environment features are variables and their conditional dependency is represented by directed edges. We propose examples for these general cases with two components, with one performance metric each, but the same reasoning can be applied to each pair of performance metrics of any two components in a complex system, and to components with more than one performance metric.

Case 1: the performance of two components, A and B, are correlated, but they are conditionally independent given the environment features and system features that are common to the two component performance models. This case is shown as a Bayes network in Fig. 3.4.

Case 2: the performance of component B depends on the performance of component A. This case is shown as a Bayes network in Fig. 3.5a, where the performance of component A depends on a system feature or environment feature, and the performance of component B depends on the performance of component A.

Case 3: component A depends on some environment/system feature and the performance of component B depends on both the performance of component A and the same system/environment features on which the performance of component A depends. This case is shown as a Bayes network in Fig. 3.5b.

When evaluating a complex system, often we are able to choose between different components implementing a functionality and we can evaluate the performance of these components with the same set of metrics. We propose three examples for general cases of how we can model the dependency in a system in which two components, A_1 and A_2, implement a functionality F_A, that influences the performance of component B. In Fig. 3.6, we show a Bayes network representing each example. In these examples, the performance of the components A_1 and A_2 depends from an environment/system feature and the performance of the component B depends on the performance of the component used to implement the functionality F_A. What differentiates these cases is how we model the dependency of the performance of B on the performance metrics of F_A.

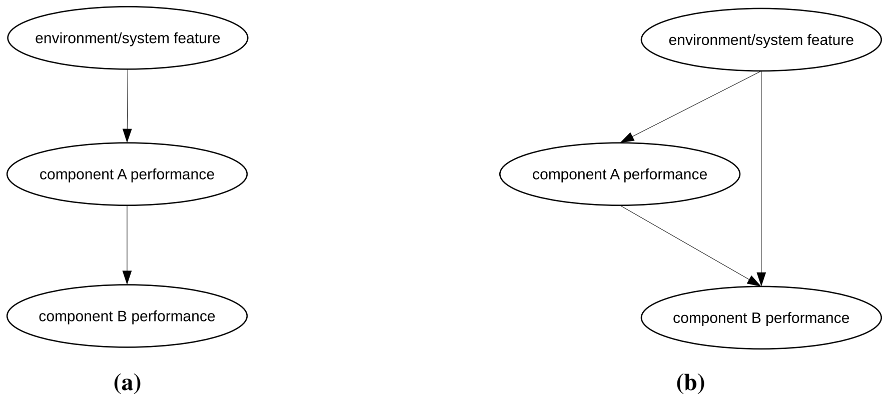


Figure 3.5: *Bayes networks in which the performance of component B is correlated with the performance of component A.*

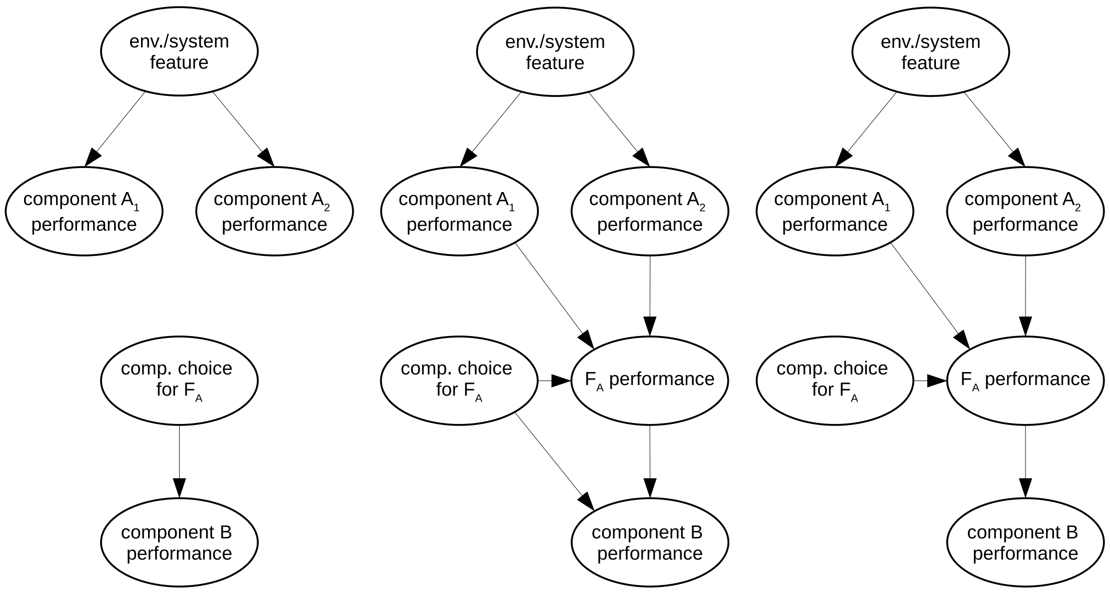
Case 1: we are not able to characterize the dependence of component B on the performance of the components A_1 and A_2, therefore, we can only predict the performance of component B from a variable identifying which component is used to implement the functionality F_A, named *component choice for* F_A in the Bayes network. This variable is considered a system feature for the performance model of component B. This is shown in the Bayes network in Fig.3.6a. Although we are not able to study how the performance of A_1 and A_2 affects the performance of B in detail, we can determine the performance of component B given the choice of component for F_A.

Case 2: we are able to partially characterize the dependence of component B on the performance of the components A_1 and A_2, and the prediction is most accurate with the knowledge of which component is used to implement the functionality F_A. This case is shown as a Bayes network in Fig. 3.6b. In this case, we can predict the performance of component B from the performance of the components A_1 and A_2, unified in the variable *functionality* F_A *performance* in the Bayes network, but using the variable identifying which component is used to implement the functionality F_A (variable *component choice for* F_A in the Bayes network), we can improve the prediction of the performance of component B. The improvement of the prediction is due to the performance metric of B only partially characterizing the dependence on the performance of F_A.

Case 3: we are able to fully characterize the dependence of component B on the performance of the components A_1 and A_2, meaning that we can predict the performance of component B from the performance of the component implementing functionality F_A, regardless of which component, A_1 or A_2 in our example, implements the functionality F_A. This is shown in the Bayes network in Fig.3.6c. If we are able to completely characterize the dependency of B on the component implementing F_A, then we can build the performance models of A_1, A_2, and B separately, then use the composed performance model to predict their performance without the need to build the actual composed system.

3.3.1 Synthetic Example of Composition

To clarify the composition of component performance models we synthesize two example datasets representing simple systems. The first example, shown in Fig. 3.7, rep-



(a) *Bayes network in which the performance of component B depends on which component is used to implement a functionally F_A.*

(b) *Bayes network in which the performance of component B depends on which component is used to implement a functionally F_A and the performance of the component implementing the functionality F_A.*

(c) *Bayes network in which the performance of component B depends on the performance of the component implementing the functionality F_A.*

Figure 3.6: *Bayes networks representing different levels of characterization of the dependency of the performance of a software component B on the performance of the multiple software components which implement the same functionality F_A.*

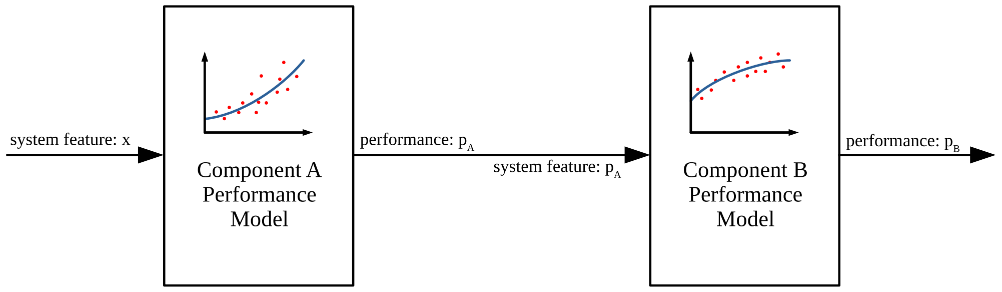


Figure 3.7: *Example of composition of two performance models. The component A performance model has one system feature, x, and a performance metric, p_A. The component B performance model has one system feature, p_A, and a performance metric, p_B.*

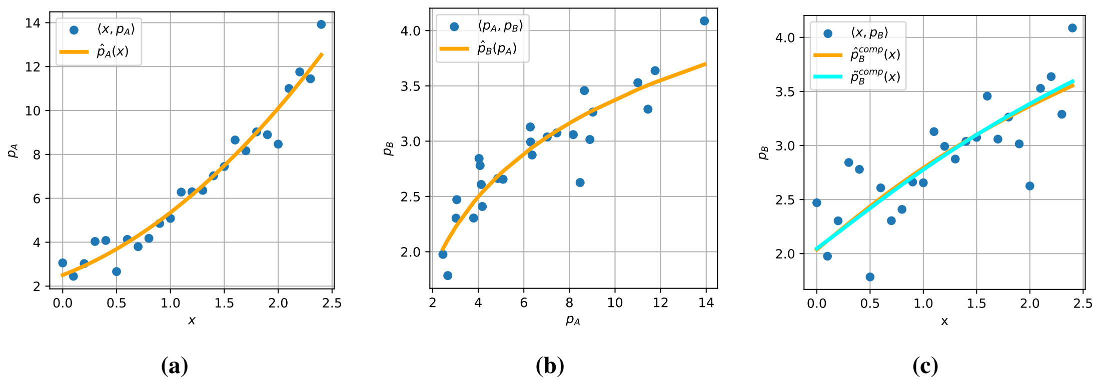


Figure 3.8: *Example of composition of two performance metrics. In (a) the $\langle x, p_A \rangle$ datapoints and the estimated function $\hat{p}_A(x)$. In (b) the $\langle p_A, p_B \rangle$ datapoints and the estimated function $\hat{p}_B(p_A)$. In (c) the $\langle x, p_B \rangle$ datapoints, the estimated function $\hat{P}_B^{comp}(x)$, and the composed function $\tilde{p}_B^{comp}(x)$.*

resents a system with two components, A and B. The performance of component A, named p_A, depends from a system feature, named x. The performance of component B, named p_B, depends from the performance of component A, therefore we treat the performance p_A as a system feature for component B.

We generate a set of datapoints, shown in Fig. 3.8, for x, p_A and p_B using Eq. 3.1 and Eq. 3.2, where $\mathcal{N}(\mu, \sigma)$ is the normal distribution with mean μ and standard deviation σ. These datapoints, in a real performance model, would be the measurements of the system feature and the performance metrics collected in different experiments.

$$p_A(x) = 1 + (1+x)^2 + \varepsilon_A, \quad \varepsilon_A \sim \mathcal{N}(0.0, 0.6) \tag{3.1}$$

$$p_B(p_A) = log(p_A(x)) + 1 + \varepsilon_B, \quad \varepsilon_B \sim \mathcal{N}(0.0, 0.2) \tag{3.2}$$

Since we know, in this example, the true relationship between x, p_A and p_B, it is easy to fit a generalized linear model estimator [31] to the data and obtain the estimated functions $\hat{p}_A(.)$ and $\hat{p}_B(.)$. This is done by fitting the linear regression $\hat{p}_A = \beta_0 + \beta_1 f_A(x)$, where $f_A(x)$ is the function that transforms the feature x to make $\hat{p}_A$ linear with respect to x, and $\hat{p}_B = \beta_2 + \beta_3 f_B(p_A)$, where $f_B(p_A)$ is the function that transforms the feature p_A to make $\hat{p}_B$ linear with respect to p_A.

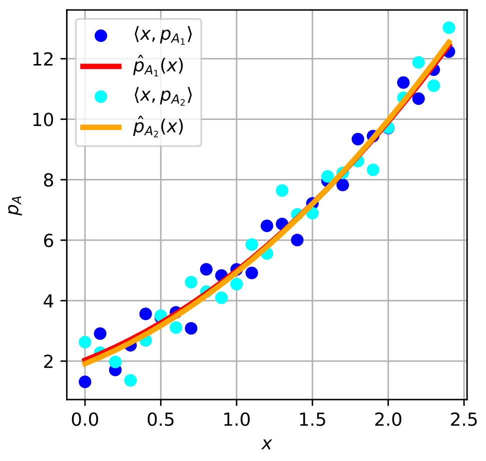
The data and estimated function for $\hat{p}_A(.)$ are shown in Fig. 3.8a, and the data and estimated function for $\hat{p}_B(.)$ are shown in Fig. 3.8b. Additionally we can produce a function estimating the performance p_B from x with one of two following approaches. Either we collect the data from the same experiments, then we can directly estimate the function $\hat{p}_B^{comp}(x)$ with a generalized linear model estimator from the dataset, shown in orange in Fig. 3.8c, or we collect the data for component A and component B in two separate sets of experiments, then we can obtain the function $\tilde{p}_B^{comp}(x) = \hat{p}_A(.) \circ \hat{p}_B(.) = \hat{p}_B(\hat{p}_A(x))$ by composing the estimated functions.

We propose a second example based on a synthesized dataset representing a system with two functionalities: functionality F_A, which can be implemented by two components, A_1 and A_2, and functionality F_B, which can be implemented by two components B_1 and B_2. The performance of components A_1 and A_2, named p_{A_1} and p_{A_2}, depend from a system feature, named x. The performance of components A_1 and A_2 is measured with the same performance metric. The performance of components B_1 and B_2, named p_{B_1} and p_{B_2}, depend from the performance of the component implementing the functionality F_A. We assume that the performance of the components A_1 and A_2 completely characterized the dependency with the performance of components B_1 and B_2, therefore we treat the performance of the components implementing the functionality F_A as the same performance, regardless of which components implements the functionality, and we name this performance p_A, which is used as a system feature for components B_1 and B_2.

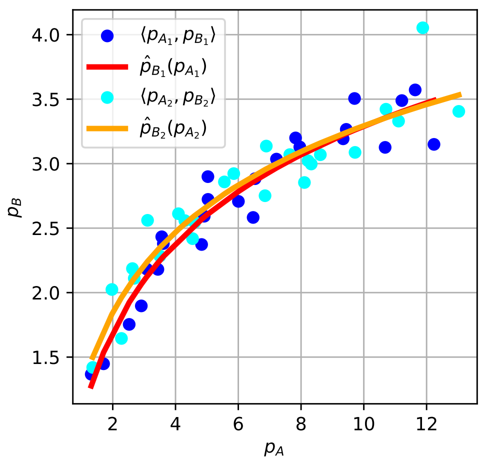
We generate a set of datapoints, shown in Fig. 3.9, for x, p_{A_1}, p_{A_2}, p_{B_1}, and p_{B_2} using the same equations as the first example, Eq. 3.1 and Eq. 3.2. We fit multiple generalized linear model estimators to the data and obtain the estimated functions $\hat{p}_{A_1}(.)$ and $\hat{p}_{A_2}(.)$ for the functionality F_A, shown in Fig. 3.9a, and the estimated functions $\hat{p}_{B_1}(.)$ and $\hat{p}_{B_2}(.)$ for the functionality F_B, shown in Fig. 3.9b. The data is generated for two separate robot systems. Robot system S_1, with components A_1 and B_1, and robot system S_2, with components A_2 and B_2.

We then compose the resulting models of the two functionalities, obtaining the functions predicting the performance p_B from the feature x. In Fig. 3.9c are shown the functions $\tilde{p}_{B_i}^{comp}(x) = \hat{p}_{A_i}(.) \circ \hat{p}_{B_i}(.) = \hat{p}_{B_i}(\hat{p}_{A_i}(x))$ obtained by composing the estimated functions of components A_1 and B_1 ($i = 1$), and components A_2 and B_2 ($i = 2$). In Fig. 3.9d are shown the functions $\tilde{p}_{B_i}^{cross}(x) = \hat{p}_{A_j}(.) \circ \hat{p}_{B_i}(.) = \hat{p}_{B_i}(\hat{p}_{A_j}(x))$ obtained by composing the estimated functions of components A_2 and B_1 ($i = 1, j = 2$), and components A_1 and B_2 ($i = 2, j = 1$). Because the performance of the components implementing each functionality are equivalent, the resulting functions $\tilde{p}_{B_i}^{comp}(x)$ and $\tilde{p}_{B_i}^{cross}(x)$ are also equivalent. The small difference between them is due to the random noise.

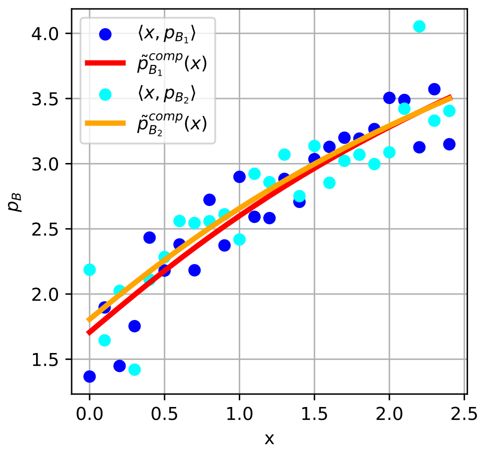
Producing performance models for a complex system may be unfeasible since the number of run parameter combinations explodes when the run parameters necessary to evaluate each component are joined. Collecting the data in separate experiments allows us to simplify the evaluation of a complex system by dividing the system in sub-systems which are tractable, and then predicting the performance of the complete system by composing the performance models. Producing performance models from separate experiments is a difficult problem to solve. When evaluating the performance of a component B, which depends on the performance of component A, the experimental setup for component B requires the information or behavior equivalent to the one which would have been produced by the component A in the same conditions (i.e., with



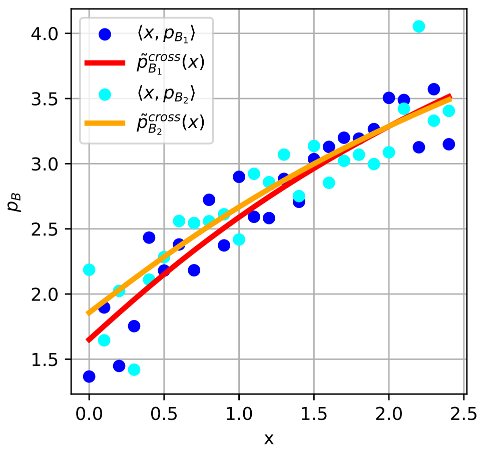
(a)



(b)



(c)



(d)

Figure 3.9: *Example of composition in a system with multiple components implementing two functionalities. In (a), the $\langle x, p_{A_1} \rangle$ and $\langle x, p_{A_2} \rangle$ datapoints and the estimated functions $\hat{p}_{A_1}(x)$ and $\hat{p}_{A_2}(x)$. In (b), the $\langle p_{A_1}, p_{B_1} \rangle$ and $\langle p_{A_2}, p_{B_2} \rangle$ datapoints and the estimated functions $\hat{p}_{B_1}(p_{A_1})$ and $\hat{p}_{B_2}(p_{A_2})$. In (c), the $\langle x, p_{B_1} \rangle$ and $\langle x, p_{B_2} \rangle$ datapoints and the estimated functions $\tilde{p}_{B_1}^{comp}(x)$ and $\tilde{p}_{B_2}^{comp}(x)$. In (d), the $\langle x, p_{B_1} \rangle$ and $\langle x, p_{B_2} \rangle$ datapoints and the estimated functions $\tilde{p}_{B_1}^{cross}(x)$ and $\tilde{p}_{B_2}^{cross}(x)$.*

equivalent system features and environment features). In principle, it may be possible to design an experimental setup for component B in which we synthesize an equivalent information or behavior that substitutes the presence of component A, but this requires knowledge on how the components A and B interact, which may require us to have studied their dependency in the same experimental setup in the first place.

CHAPTER 4

Component Performance Modelling

In this chapter, we describe two use cases which we use to demonstrate our methodology. In the first use case, we show how to produce performance models for a single functionality by evaluating the performance of three Simultaneous Localization and Mapping (SLAM) components and their dependency on different features of the system and a feature of the environment. In the second use case, we demonstrate how we produce a composed performance model for a complex system which includes three functionalities needed for autonomous navigation: local planning, global planning and localization. We evaluate three local planning components, two global planning components, and one localization component. In the two use cases, we describe the experimental setup and the experimental protocol which allows us to evaluate different software components, the run parameters used to configure the experimental setup, and the run data which we collect from the experiments. Finally, we describe how we compute the environment features and the performance metrics used to evaluate the performance of the software components and their dependency from features of the environment, features of the robot system, and the dependency between the performance of different components used in the same system. The resulting performance models obtained with our methodology will be shown in Chapter 5.

4.1 SLAM Component Performance Modelling

In this section, we present a use case of the proposed performance modelling methodology applied to the SLAM functionality. The experiments consist of a navigation task which visits every location of various indoor environments. The robot system has two sensors, a LIDAR sensor and an odometry sensor. The SLAM component uses information from these sensors to simultaneously estimate the pose of the robot and map

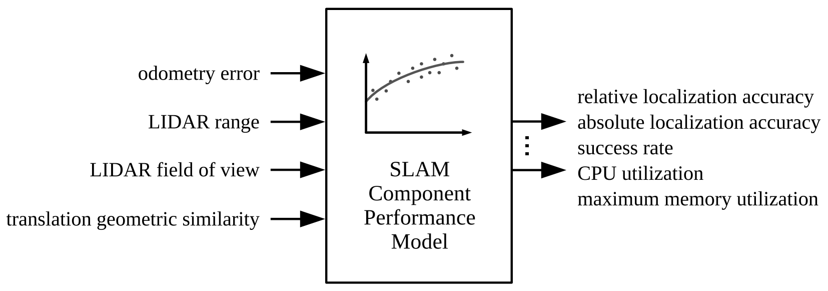

odometry error
LIDAR range
LIDAR field of view
translation geometric similarity
SLAM Component Performance Model
relative localization accuracy
absolute localization accuracy
success rate
CPU utilization
maximum memory utilization

Figure 4.1: *High level view of the SLAM performance models. Various features (odometry error, LIDAR range, LIDAR field of view, and translation geometric similarity) are used to predict multiple performance metrics of the SLAM components.*

of the environment. The experimental setup, described in more details in the following section, includes all the software needed to simulate the environment and to support and coordinate the execution of the run, such as the navigation stack software and the software which sends commands to the navigation stack in order to navigate through every location of the environmet. A high-level view of the performance models is shown in Fig. 4.1, highlighting the environment features and system features from which we predict the performance metrics.

We evaluate three SLAM components, GMapping[1] [17] (a combination of particle filter for the localization and Extended Kalman Filters for the map landmarks), and two graph-SLAM-based methods: SLAM Toolbox[2] [30] and Hector SLAM[3] [22]. All software components estimate a map of the environment and the current pose of the robot from 2D LIDAR sensor readings, while only GMapping and SLAM Toolbox use the odometry information. The SLAM performance model allows us to predict the performance of the SLAM component from three system features relative to the LIDAR and odometry sensors and an environment feature which quantifies the amount of information provided by the geometry of the environment as encoded in the LIDAR measurements.

The diagram, split in three parts and shown in Fig. 4.2-4.4, describes the entities that are part of the SLAM performance model and their relationship. The first part of the diagram, shown in Fig. 4.2, lists the run parameters used to configure the experimental setup in each run, the experimental environments, the components evaluated and the functionality they implement, which in this case, is the single functionality being evaluated in the experimental setup. Note that, although the experimental setup includes multiple SLAM components, they are never used at the same time, but rather in different runs. The second part of the diagram, shown in Fig. 4.3, lists the run data collected during the experiments, the performance metrics, the system features and the environment features included in the performance model. The relationships, represented by edges, in this part of the diagram specify which run data is used to obtain each metric and feature. The third part of the diagram, shown in Fig. 4.4, lists the performance metric statistical representations included in the performance model, which performance metric they predict, and which features are used to predict the performance metric.

[1]`http://wiki.ros.org/gmapping`. Accessed on 2022-10-14.
[2]`http://wiki.ros.org/slam_toolbox`. Accessed on 2022-10-14.
[3]`http://wiki.ros.org/hector_slam`. Accessed on 2022-10-14.

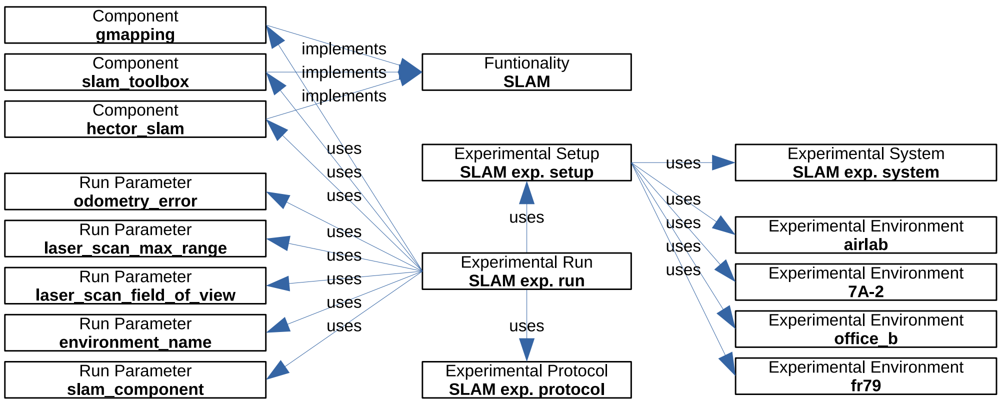

Component gmapping
Component slam_toolbox
Component hector_slam
implements
implements
implements
Funtionality SLAM
Run Parameter odometry_error
Run Parameter laser_scan_max_range
Run Parameter laser_scan_field_of_view
Run Parameter environment_name
Run Parameter slam_component
uses
Experimental Setup SLAM exp. setup
Experimental Run SLAM exp. run
Experimental Protocol SLAM exp. protocol
Experimental System SLAM exp. system
Experimental Environment airlab
Experimental Environment 7A-2
Experimental Environment office_b
Experimental Environment fr79

Figure 4.2: *Part 1 of 3 of the SLAM performance model diagram, listing the entities used in the performance model: the experimental setup, the experimental system, the experimental environments, the components evaluated in the experimental setup, the functionality implemented by the components, and the run parameters used to configure the experimental setup.*

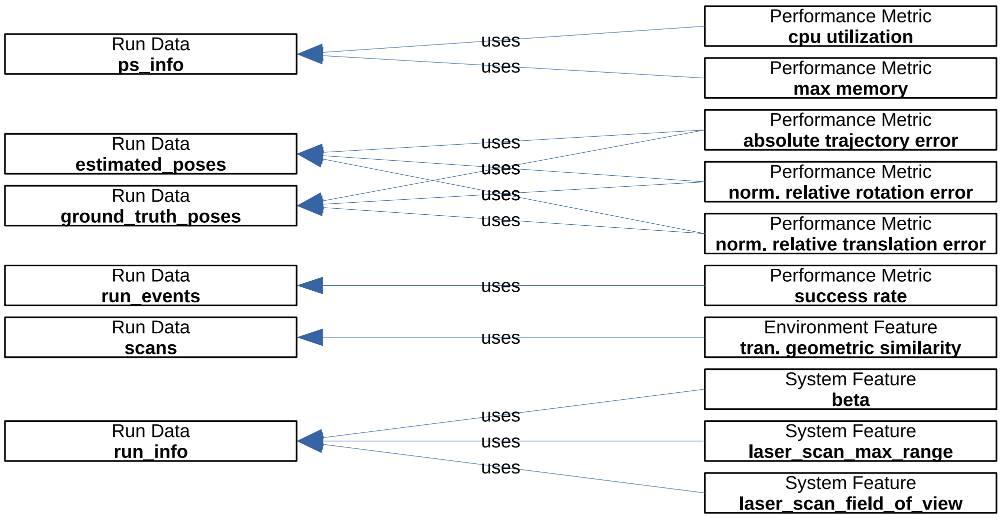

Run Data ps_info
Run Data estimated_poses
Run Data ground_truth_poses
Run Data run_events
Run Data scans
Run Data run_info
uses
Performance Metric cpu utilization
Performance Metric max memory
Performance Metric absolute trajectory error
Performance Metric norm. relative rotation error
Performance Metric norm. relative translation error
Performance Metric success rate
Environment Feature tran. geometric similarity
System Feature beta
System Feature laser_scan_max_range
System Feature laser_scan_field_of_view

Figure 4.3: *Part 2 of 3 of the SLAM performance model diagram, listing the entities used in the performance model: the run data, the performance metrics, the environment features and the system features. The relationships in this diagram show which run data is used to compute each metric and feature.*

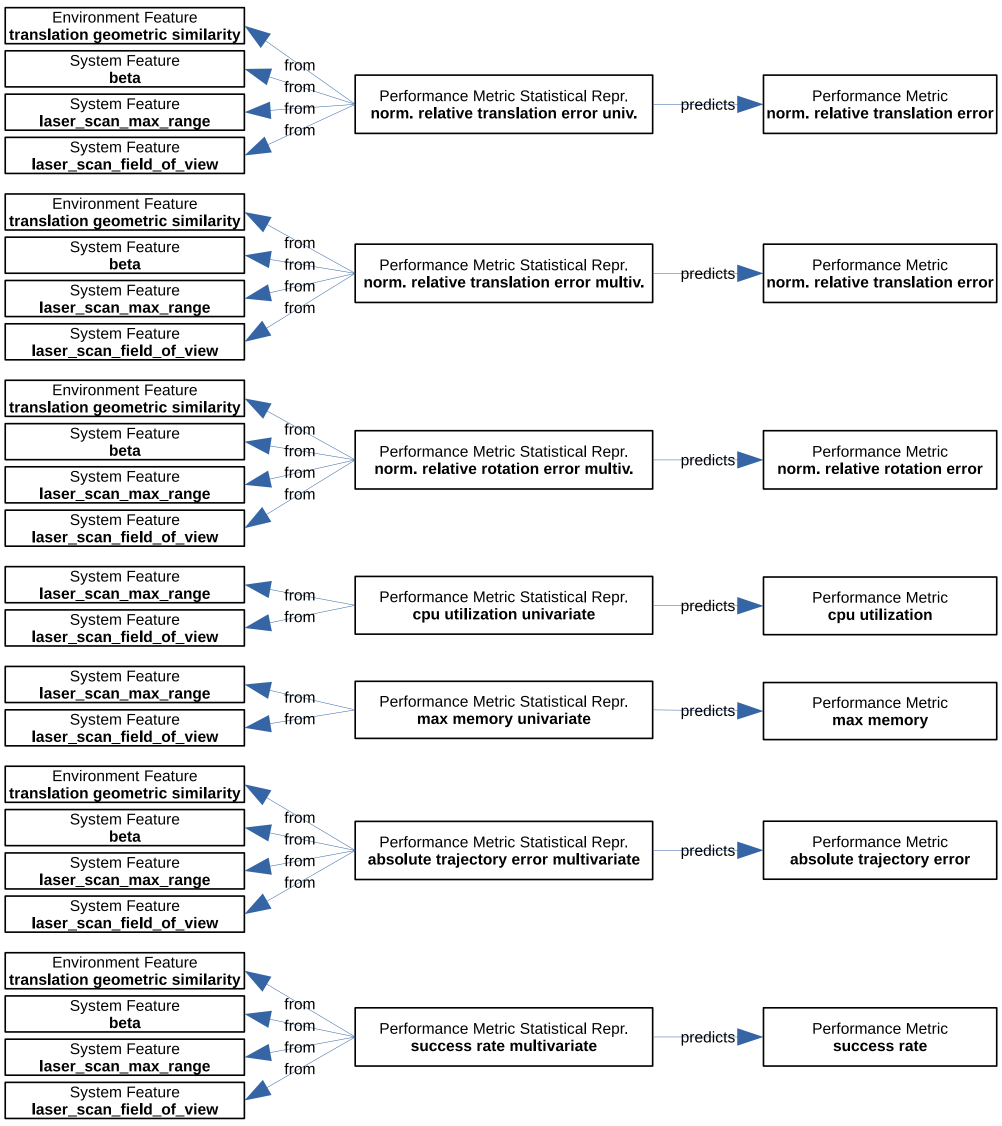

Environment Feature translation geometric similarity
System Feature beta
System Feature laser_scan_max_range
System Feature laser_scan_field_of_view
from
Performance Metric Statistical Repr. norm. relative translation error univ.
predicts
Performance Metric norm. relative translation error
Performance Metric Statistical Repr. norm. relative translation error multiv.
Performance Metric Statistical Repr. norm. relative rotation error multiv.
Performance Metric norm. relative rotation error
Performance Metric Statistical Repr. cpu utilization univariate
Performance Metric cpu utilization
Performance Metric Statistical Repr. max memory univariate
Performance Metric max memory
Performance Metric Statistical Repr. absolute trajectory error multivariate
Performance Metric absolute trajectory error
Performance Metric Statistical Repr. success rate multivariate
Performance Metric success rate

Figure 4.4: *Part 3 of 3 of the SLAM performance model diagram, listing the entities used in the performance model: the performance metrics, the system features, the environment features, and the performance metric statistical representations. The relationships in this diagram show which performance metric is predicted by each statistical representation, and from which features they are predicted.*

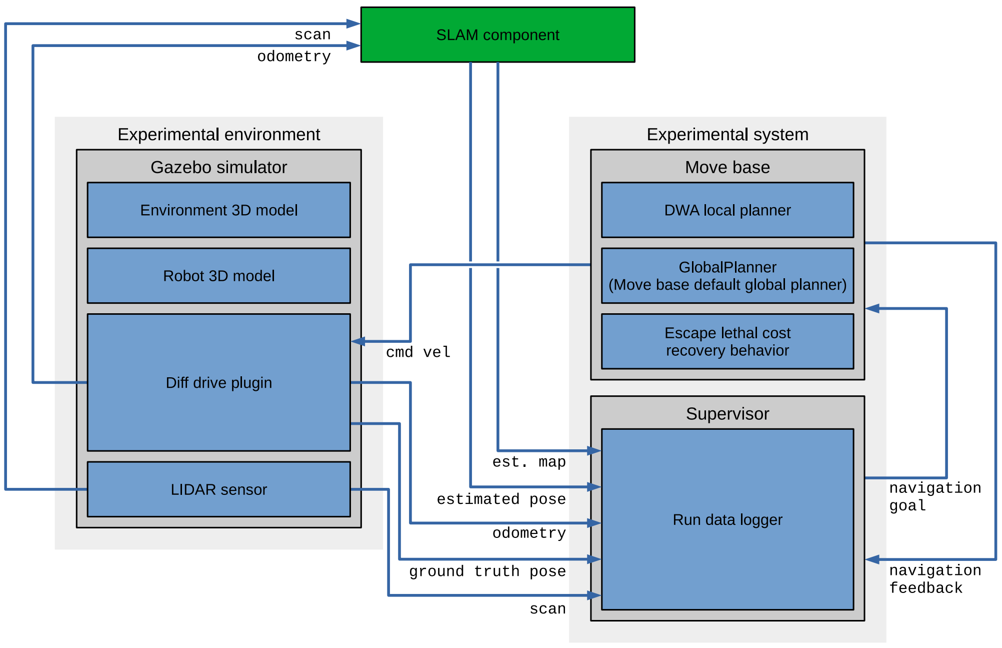


Figure 4.5: *The software used to implement the experimental setup. The SLAM component is highlighted in green. The edges of the graph represent the communication between the software.*

In the following sections we describe in details the experimental setup, the experimental protocol, the run parameters used to configure the experimental setup, the run data which we collect from the experiments, and finally the environment feature and the performance metrics that we will later use to evaluate the SLAM components.

4.1.1 Experimental Setup

The experimental setup needed to execute the experiments provides the environment and system software of a robot that autonomously navigates and visits every location of a building. The experimental setup is divided in an experimental environment and an experimental system. The experimental environment includes the 3D model of a building, the 3D model of the robot, and the plugins implementing the sensors and actuators of the robot. The experimental system includes the *Move base* navigation stack and the supervisor. A diagram of the setup is shown in Fig. 4.5.

The navigation stack ensures that the robot navigates towards navigation goals sent by the supervisor, and provides feedback to the supervisor to communicate that the goal was reached or that the navigation has failed. The navigation stack relies on ground truth information to avoid any interference by the performance of the SLAM component, which would otherwise require us to treat the navigation stack software as additional software components, but in this case we are not yet interested in evaluating a complex system. The navigation stack utilizes three plugins: the DWA local planner, the ROS global planner named *GlobalPlanner*, and a recovery behavior named *escape lethal cost*.

The recovery behavior was developed on purpose for this experimental setup to increase the reliability of the navigation. The DWA local planner often maneuvers the robot closer than a safety distance from the obstacles, causing it to stop to avoid a

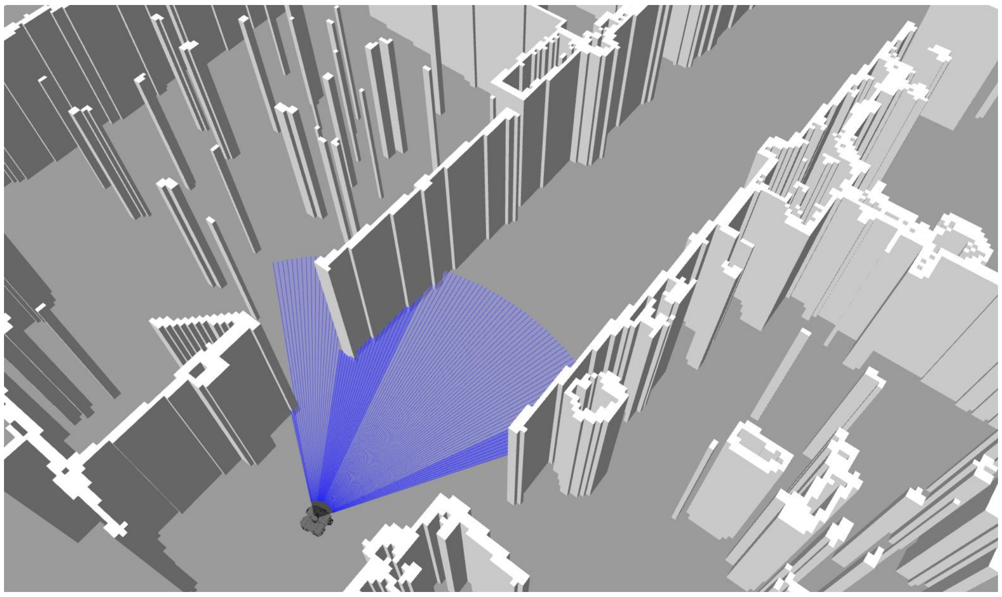


Figure 4.6: *View of the Gazebo simulation. The 3D model generated from the fr079 grid-map and the Turtlebot robot. The blue lines correspond to the LIDAR beams.*

potential collision. In this state, the local planner is not able to resume the navigation by itself. The recovery behavior, which is activated by the navigation stack when the local planner can not make progress, consists of rotating the robot in place away from the nearby obstacles and then slowly moving forward until the robot is far enough to resume the navigation.

The supervisor is in charge of sending commands to the navigation stack in order to visit every location of the environment, deciding when to stop the run, and collecting the run data.

The experimental environment is implemented using the Gazebo simulator[4]. Gazebo is sufficient to simulate an environment with the features that impact the performance of the SLAM components being evaluated. A more complex simulator, such as the ones able to provide a photorealistic representation of the environment, would not improve the quality of the simulation in our setup since in this case we evaluate software components that only rely on information from a 2D LIDAR sensor and wheel odometry. The simulation consists of a 3D representation of a physical environment and a 3D robot model, see Fig. 4.6.

We craft the 3D environments by extruding the 2D gridmaps in Fig. 4.7. Three environments (airlab, 7A-2, office_b) represent empty buildings of varying sizes generated from gridmaps from the dataset presented by [2]. One environment, named fr079, is generated from a gridmap obtained by running GMapping on data from the radish dataset[5] [19], which was collected in real buildings. These environments have different characteristics: airlab is small, office_b presents repetitive patterns on a global scale, 7A-2 is made of very large and empty rooms, fr079 represents a realistic scenario for highly cluttered indoor space.

[4]`https://www.gazebosim.org/`. Accessed on 2022-02-02.

[5]The data was provided by Cyrill Stachniss, and is available at `http://ais.informatik.uni-freiburg.de/slamevaluation/datasets.php`. Accessed on 2022-02-02.

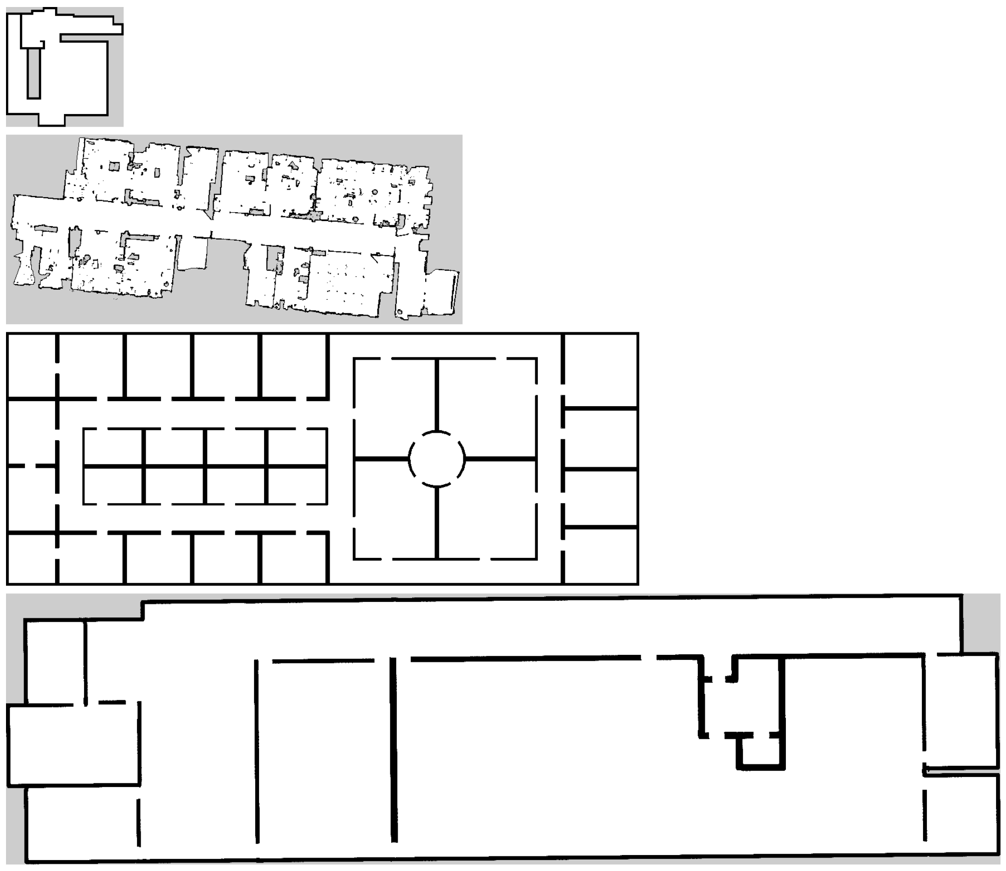


Figure 4.7: *The gridmaps of the experimental environments (top to bottom): airlab, fr079, office_b, 7A-2. All the gridmaps have the same scale, 0.05m per pixel.*

The modeled robot is the Turtlebot 3 Waffle[6], a robot platform widely used both in simulation and the real world. The LIDAR sensor is simulated with the default Gazebo plugin. The measurement noise is implemented as Gaussian noise added to each range measurement independently from the range value (constant standard deviation of 0.01m and zero mean). The angular increment between beams is 1 degree. The measurement frequency is set to 10Hz. The *maximum range* and *field of view* are assumed, for the sake of simplicity, as the system features of the LIDAR sensor affecting the performance of the SLAM methods under evaluation, therefore we run experiments with different values which are common in commercial products, i.e., the maximum range is set to 3.5, 8, 15,and 30 meters, and the field of view is set to 90, 180, 270, and 360 degrees.

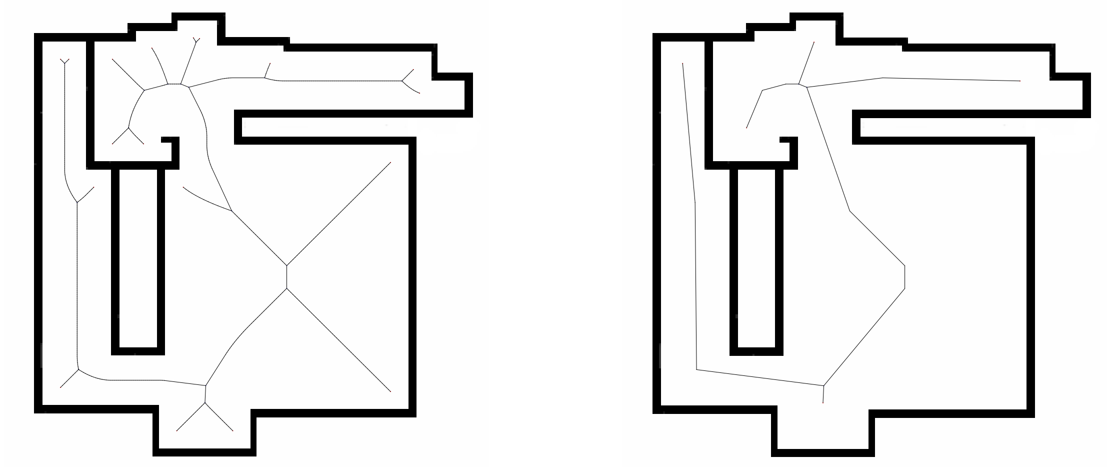
The odometry data is computed with a version of the differential drive Gazebo plugin that we modified to simulate an odometry sensor affected by a rotational and translational Gaussian noise proportional to the amount of rotation and translation performed between each odometry measurement. The standard deviation of the translation and rotation odometry error is $\beta\delta_\rho$ and $\beta\delta_\theta$ respectively, where δ_ρ and δ_θ are the ground truth translation and rotation of the odometry reading. The amount of odometry noise has a significant effect on the performance of the two SLAM methods, therefore we consider it a system feature, and run experiments with a range of values: $\beta = (0.0, 0.5, 1.0, 1.5, 2.0)$. This noise model is inspired by the one described in [44].

4.1.2 Experimental Protocol

The navigation procedure used to acquire the data to build our performance model is based on a traversal path which is computed from the ground truth gridmap and which visits every location of the environment accessible to the robot. The traversal path is obtained from the graph G_1 (Fig. 4.8a) obtained by computing a Delaunay triangulation based on the occupied cells of the gridmap and the corresponding Voronoi graph. G_1 is reduced to a simpler graph G_2 by keeping only the nodes connected to one or three edges, and then by removing leaf nodes (see Fig. 4.8b).

The traversal path is then obtained from the graph G_2 by randomly selecting an initial vertex from the graph G_2 then the next vertex is selected as the vertex of graph G_2 with the shortest distance to the current vertex among the vertices that have not been included in the traversal path. The distance is computed with respect to the graph G_1. The procedure terminates when all vertices of graph G_2 are added to the traversal path. A further simplification of the path is made by discarding every vertex closer than 2m to the previous before building the traversal path to speed up the execution of the run and simplify the navigation task. The traversal path is traversed in both directions, allowing the SLAM components to produce complete gridmaps even with LIDAR sensors having a field of view smaller than 180 degrees. Two criteria determine the termination of the run: the completion of the navigation on the traversal path, and a timeout of 7200s (2 hours). No experiment has been terminated by timeout.

[6]`https://github.com/ROBOTIS-GIT/turtlebot3_simulations/tree/melodic-devel/turtlebot3_gazebo/models/turtlebot3_waffle`. Accessed on 2022-02-02.



(a) *Voronoi graph G_1* **(b)** *Reduced graph G_2*

Figure 4.8: *Graphs used to compute a path that visits every location of an environment.*

4.1.3 Run Parameters

Five run parameters are used to produce the experiments, *odometry error*, *laser scan max range*, *laser scan field of view*, *environment name*, and *slam component*. Additionally, each combination of parameters is repeated 5 times to account for variability. The run parameter *slam component* is used to select between the three SLAM algorithms that come packaged in the ROS framework: GMapping, SLAM Toolbox, and Hector SLAM. The run parameter *environment name* is used to select which of the simulated environments is used in each run.

The run parameter *odometry error* is a vector describing the four components of odometric error, similarly to the odometry error model in [44]: *beta_1* controls the amount of odometry rotation error caused by the effective rotation, *beta_2* controls the amount of odometry rotation error caused by the effective translation, *beta_3* controls the amount of odometry translation error caused by the effective translation, and *beta_4* controls the amount of odometry translation error caused by the effective rotation. Although in a real robot the odometric error would manifest in all four components, to reduce the number of run parameters combinations, only *beta_1* and *beta_3* are used and set to the same value, which we call *beta*, and is also used as a system feature in the performance model. The run parameters *laser scan max range* and *laser scan field of view* are used to set the maximum range and field of view of the LIDAR sensor. These run parameters limit the quality of the sensor information which is available to the SLAM component, and are also used as system features in the performance model.

In order to reduce the number of parameters combinations while still testing many values of odometry and LIDAR parameters, two parameter grids are used. Using two grids allows us to obtain data with sufficient resolution for odometry error and the LIDAR parameter values, with fewer parameter combinations than the same grid with all parameter values. The number of parameters combinations is computed as the product of the number of values of *slam component*, *environment name*, *odometry error*, *laser scan max range*, and *laser scan field of view*. The number of runs is the number of parameters combinations multiplied by the number of repetitions of each combination.

```
combinatorial_parameters: [
  {
    slam_component: [gmapping, slam_toolbox, hector_slam],
    environment_name: [airlab, 7A-2, office_b, fr79],
    odometry_error: [[0.0, 0.0, 0.0, 0.0], [2.0, 0.0, 2.0, 0.0]],
    laser_scan_max_range: [3.5, 8.0, 15.0, 30.0],
    laser_scan_fov: [90, 180, 270, 359],
  },
]
```

Listing 4.1: *Run parameter grid used in the SLAM performance model, with more values for LIDAR parameters.*

```
combinatorial_parameters: [
  {
    slam_component: [gmapping, slam_toolbox, hector_slam],
    environment_name: [airlab, 7A-2, office_b, fr79],
    odometry_error: [
        [0.5, 0.0, 0.5, 0.0],
        [1.0, 0.0, 1.0, 0.0],
        [1.5, 0.0, 1.5, 0.0]
    ],
    laser_scan_max_range: [8.0, 30.0],
    laser_scan_fov: [180, 359],
  },
]
```

Listing 4.2: *Run parameter grid used in the SLAM performance model, with more values for odometry parameters .*

In the run parameter grid in Listing 4.1, four values are used for the *laser scan max range* and *laser scan field of view* run parameters, and only two values are used for the *odometry error* run parameter, resulting in 384 parameters combinations and 1920 runs.

In the run parameter grid in Listing 4.2, only two values are used for the *laser scan max range* and *laser scan field of view* run parameters, and three additional values are used for the *odometry error* run parameter, resulting in 144 parameters combinations and 720 runs.

The two run parameter grids can be visualized in Table 4.1, in which for simplicity only the *laser scan max range* and *odometry error* run parameters are present. Using two smaller grids, the sum of the number of parameters combination is 528 (2640 runs), while a single grid with all parameter values would generate 960 parameters combinations (4800 runs).

4.1.4 Run Data

The following run data are collected in each run:

- run_info: this run data stores the value of the run parameters with which the run is configured.
- run_events: this run data is a series of events identified by a name and a timestamp. The relevant events are: run_start identifies when the run has started, tar-

		odometry error				
		0.0	0.5	1.0	1.5	2.0
	3.5	G1				G1
laser scan	8	G1	G2	G2	G2	G1
max range	15	G1				G1
	30	G1	G2	G2	G2	G1

Table 4.1: *G1 are combinations from grid 1 (Listing 4.1), G2 are combinations from grid 2 (Listing 4.2).*

get_pose_set indicates that the supervisor successfully sent a navigation goal to the navigation stack, waypoint_timeout indicates that the navigation stack did not reach the goal within a timeout, target_pose_reached indicates that the navigation stack reported that the goal was reached, target_pose_not_reached would indicate that the navigation goal reported that it failed to reach the navigation goal, run_timeout indicates the run has reached the timeout, run_completed indicates the run is finished, supervisor_finished indicates the supervisor is about to terminate.

- scans: this run data stores a series of laser scan readings, including information on the time of generation of the data from the sensor, the characteristics of the sensor (i.e., field of view, minimum and maximum range, and number of beams), and the list of range measurements.
- estimated_poses: this run data contains a series of the poses of the robot as estimated by the SLAM component, including the time. The pose is sampled at 10Hz.
- ground_truth_poses: this run data contains a series of the ground truth poses of the robot as published by the simulator, including the time. The pose is sampled every time the simulator publishes the information, which is around 100Hz.
- ps_info: this run data contains a series of snapshots of all the experimental environment and system processes running on the machine, obtained with the *psutil* Python library[7]. This allows to retrieve information about the processes such as memory and CPU utilization. The snapshots are collected every 10 seconds.

4.1.5 Environment Features

The performance of the SLAM and localization methods that use LIDAR measurements in their estimation process depends on the information provided by the geometry of the environment surrounding the sensor. At one extreme, an environment may have geometric features such that localization can be obtained with LIDAR information alone, at the opposite extreme, an environment may be geometrically ambiguous resulting in the LIDAR information being almost useless.

[7] `https://psutil.readthedocs.io/en/latest/#psutil.Process`. Accessed on 2022-07-01.

We measure the amount of information provided by a LIDAR measurement with an environment feature we call *environment geometric similarity*[8]. The geometric similarity is a measure of the uncertainty in the estimation of a small translation and rotation based on the information encoded by a LIDAR sensor. The feature ranges from 0 for complete certainty to 1 for complete uncertainty. We named this feature geometric similarity because it measures the self-similarity of the environment with respect to translation and rotation at a specific position and orientation within the environment.

The metric is computed employing the real-time scan matching method presented in [32], which computes an approximation of the probability $p(x|z,m)$, (where z is the LIDAR information, x is the pose of the robot and m is the gridmap information). The original method in [32] is used to solve the scan-matching problem, which consists in matching the LIDAR information z to the gridmap m within a small window $x+\Delta x$ in order to find the translation and rotation Δx. We are interested in the probability $p(\Delta x|z,m_z)$, where m_z is the gridmap information equivalent to the LIDAR information z itself (i.e., the probability of a scan-match of the surrounding environment with itself). We neglect the pose in the global reference frame x since we represent all data in the robot frame of reference. The window Δx used to compute the feature is $(0 \pm 1m, 0 \pm 1m, 0 \pm 11deg)$. The translation geometric similarity is computed as $tgs = 1 - v_{min}/v_{max}$, where v_{min}, v_{max} are the smaller and larger eigenvalues of the translation covariance matrix $Cov[\Delta x|z,m_z]$.

In our experiments, we aim at finding the relation between the environment feature *translation geometric similarity* relative to a trajectory and the performance metrics *normalized translation and rotation relative errors* (see Sec. 4.1.6) along the same trajectory. To relate the geometric similarity to a trajectory we average the value of this feature sampled at different poses within the trajectory.

4.1.6 SLAM Performance Metrics

The output of a SLAM method are the robot poses and the map. Another important aspect of these components is the computational resources used during the execution, i.e., the average CPU and maximum memory utilization. To measure the pose accuracy we use three metrics: the *normalized translation relative error* ε_T^{norm}, the *normalized rotation relative error* ε_R^{norm} and the *absolute trajectory error*. The normalized translation relative error ε_T^{norm} and the normalized rotation relative error ε_R^{norm} measure the amount of translation and rotation error that is accumulated in the estimation of the robot pose by the SLAM component between successive waypoints (w_a, w_b) of the traversal path,

[8]The code of the implementation can be found at `https://github.com/AIRLab-POLIMI/cartographer/blob/geometric-similarity-dev/cartographer/geometric_similarity/geometric_similarity.cc`. Accessed on 2022-10-14.

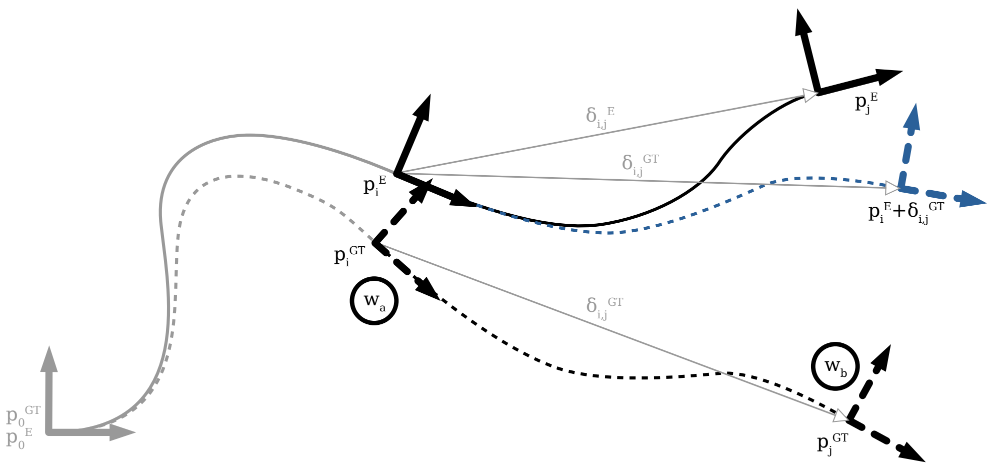


Figure 4.9: *Example of estimated trajectory (continuous black line), ground truth trajectory (dashed black line), ground truth trajectory with zeroed previously accumulated error (dashed blue line), waypoints a and b.*

as visualized in Fig. 4.9. Their values are computed as

$$\varepsilon_T^{norm}(i,j) = \frac{||trans(\delta_{i,j}^{E} \ominus \delta_{i,j}^{GT})||}{l_{i,j}} \tag{4.1}$$

$$\varepsilon_R^{norm}(i,j) = \frac{|rot(\delta_{i,j}^{E} \ominus \delta_{i,j}^{GT})|}{l_{i,j}} \tag{4.2}$$

$$l_{i,j} = \sum_{k=i}^{j-1} ||trans(p_{k+1}^{GT}, p_k^{GT})|| \tag{4.3}$$

$$ATE = \frac{1}{N}\sum_{k=0}^{N-1} ||trans(p_k^{E}, p_k^{GT})|| \tag{4.4}$$

where i,j are the indices of the poses closest in time to waypoints w_a and w_b respectively, $\delta_{i,j}^{E}$ and $\delta_{i,j}^{GT}$ are the homogeneous transformations from the i-th to the j-th estimated pose and ground truth pose respectively, p_k^{GT} are ground truth poses. $\ominus$ is the inverse of the transformation composition operator. $trans$ is the translation component of an homogeneous transformation or the translation between poses. rot is the rotation component of an homogeneous transformation. The relative errors disregard the error accumulated before the waypoint w_a, and it only considers the error accumulated from a waypoint to the next one. We normalize these metrics with the length of the trajectory $l_{i,j}$ followed by the robot between the waypoints to decrease the dependence of the performance on the length of the trajectory.

The *Absolute Trajectory Error (ATE)* is computed as Eq. 4.4. Note that this metric is computed for a whole run, rather than for each pair of waypoints. To evaluate the probability of success of the SLAM process, we establish an arbitrary threshold for the absolute trajectory error of 100m. Using this threshold we define the performance metric *success rate*. When the SLAM process fails, the values of the ATE may become meaningless or too large to compute the mean, thus when aggregating the data of the absolute trajectory error we only consider the data under the threshold and represent

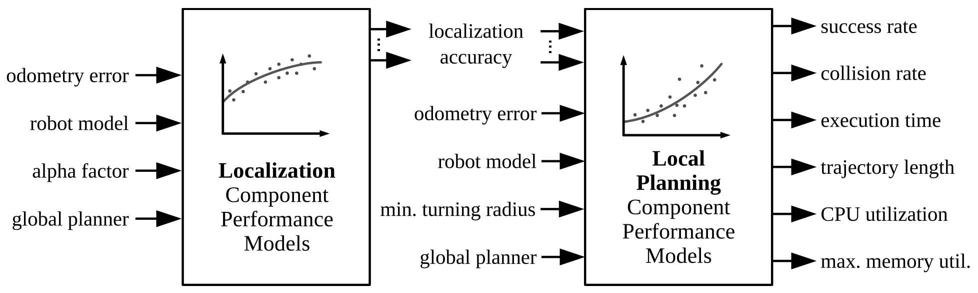


Figure 4.10: *High level view of the Localization and Local Planning performance models. Various features (odometry error, robot model, alpha factor and global planner) are used to predict the performance of the localization components. These performance metrics, which are used as system features, are used to predict the performance of the local planning components, together with additional system features (odometry error, robot model, minimum turning radius, and global planner).*

the success rate as a separate metric.

We compute the *CPU utilization* metric as the total user and system CPU time accumulated by the ROS node implementing the SLAM component divided by the duration of the run. This metric is dependent on the processor hardware and can only be used to compare results obtained on the same machine or with the same hardware. We run all experiments on the same machine. The *max memory* metric measures the maximum amount of memory allocated to the ROS node implementing the SLAM component during a run in MiB (2^{20} Bytes). The allocated memory is counted as Unique Set Size, i.e., the memory which is unique to the ROS node's processes and which would be freed if those processes were terminated.

4.2 Localization and Local Planning Performance Modelling

In this section, we present a use case of the proposed performance modelling methodology applied to a complex system with three functionalities: localization, local planning and global planning. We build a composed performance model by designing an experimental setup in which the components implementing the three functionalities are evaluated simultaneously, therefore obtaining a performance model for multiple functionalities as well. A high-level view of the performance models is shown in Fig. 4.10, highlighting the system features and the performance metrics used in the performance models, and the use of performance metrics of one functionality (i.e., localization) as system features of performance models of other functionalities (i.e., local planning).

The experiments consists of a navigation task from one location to another in various indoor environments. The robot system has two sensors, a LIDAR sensor and an odometry sensor. The localization component uses information from the LIDAR and odometry sensors to estimate the pose of the robot in the environment. The local planner component sends motion commands to the robot platform in order to follow the path generated by the global planner and reach the goal location. The experimental setup, which will be described in more details in the following section, includes all the software needed to simulate the environment and to support and coordinate the execution of the run, such as the navigation stack software which coordinates the communication between the local planner and the global planner, and the software which sends requests to the navigation stack in order to navigate to a specific location.

We evaluate one component implementing the localization functionality, AMCL[9] [14]. The localization performance model allows us to predict the performance of the localization component from two system features, *beta* and *AMCL alpha factor*. The system feature *beta* represents the amount of error of the odometric sensor, specifying the amount of rotation and translation drift resulting from the motion of the robot. The system feature *AMCL alpha factor* specifies the over- or under-estimation of the odometry error provided to AMCL. The performance of the localization component is evaluated with multiple metrics which measure the error of the estimated pose, and are additionally used as system features to evaluate the dependency of the local planning component.

We evaluate three local planning components, TEB (Timed Elastic Band)[10] [36–40], DWB (based on Dynamic Window Approach)[11] [15] and RPP (Regulated Pure Pursuit)[12]. The local planning performance model allows us to predict the performance of the local planner from the system feature *beta* and from the system features characterizing the localization error.

We use two global planning components in the experimental setup: the NavFN global planner[13], using wavefront Dijkstra planning, and the SMAC global planner[14] [9], an SE2 Hybrid-A* using a Reeds-Shepp motion model. Although characteristics of the global plan may influence the performance of the local planning and localization components, to simplify the modelling problem and because the available global planners have a limited amount of configuration, we do not produce a global planning performance model, instead, we only represent which global planner is used in each run as a system feature and show the effect of using each global planning component on the performance of the other components.

The diagram, split in five parts and shown in Fig. 4.11-4.15, is an instance of the meta-model for the composed performance model, and describes the entities that are part of the performance model and their relationship. The first part of the diagram, shown in Fig. 4.11, lists the run parameters used to configure the experimental setup in each run, the experimental environments, the components evaluated and which functionality they implement. Note that, although the experimental setup includes multiple components implementing the same functionality, they are never used at the same time, but rather in different runs, as specified by the *localization component*, *local planner component*, and *global planner component* run parameters. The second and third parts of the diagram, shown in Fig. 4.12 and Fig. 4.13, list the run data collected during the experiments, the performance metrics, the system features and the environment features included in the performance model. The relationships, represented by edges, in this part of the diagram specify which run data is used to obtain each metric and feature. The entities used to evaluate the localization component are shown in Fig. 4.12 and the entities used to evaluate the local planning components are shown in Fig. 4.13. The fourth and fifth parts of the diagram, shown in Fig. 4.14 and Fig. 4.15, list the per-

[9]`https://index.ros.org/p/nav2_amcl/#foxy`. Accessed on 2022-10-13.

[10]`https://index.ros.org/r/teb_local_planner/#foxy`. Accessed on 2022-10-13.

[11]`https://index.ros.org/p/nav2_dwb_controller/github-ros-planning-navigation2/#foxy`. Accessed on 2022-10-13.

[12]`https://index.ros.org/p/nav2_regulated_pure_pursuit_controller/#foxy`. Accessed on 2022-10-13.

[13]`https://index.ros.org/p/nav2_navfn_planner/#foxy`. Accessed on 2022-10-13.

[14]`https://index.ros.org/p/smac_planner/#foxy`. Accessed on 2022-10-13.

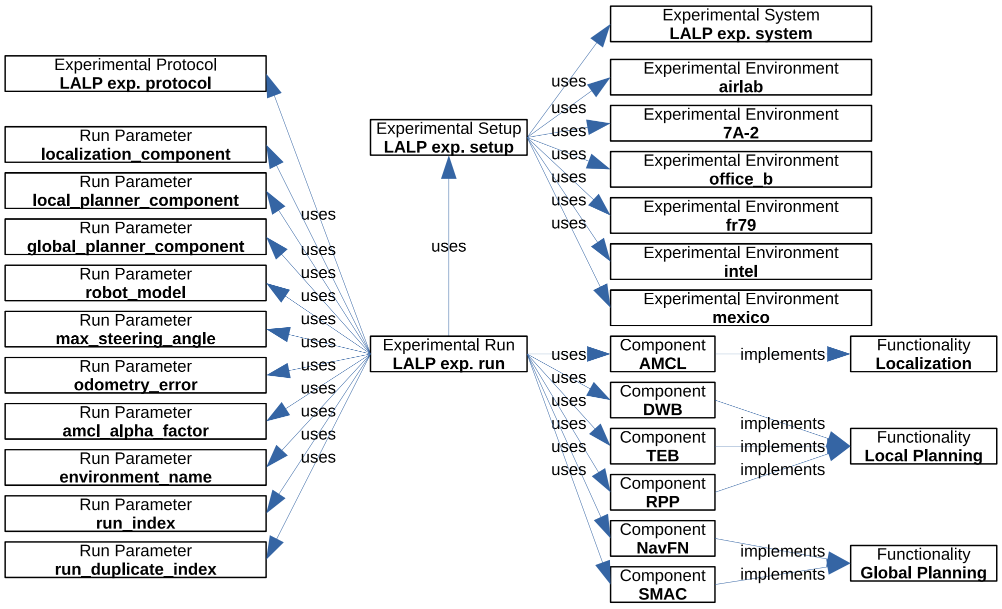


Figure 4.11: *Part 1 of 5 of the Localization And Local Planning (LALP) performance model diagram, listing the entities used in the performance model: the experimental setup, the experimental system, the experimental environments, the components evaluated in the experimental setup, the functionalities implemented by each component, and the run parameters used to configure the experimental setup.*

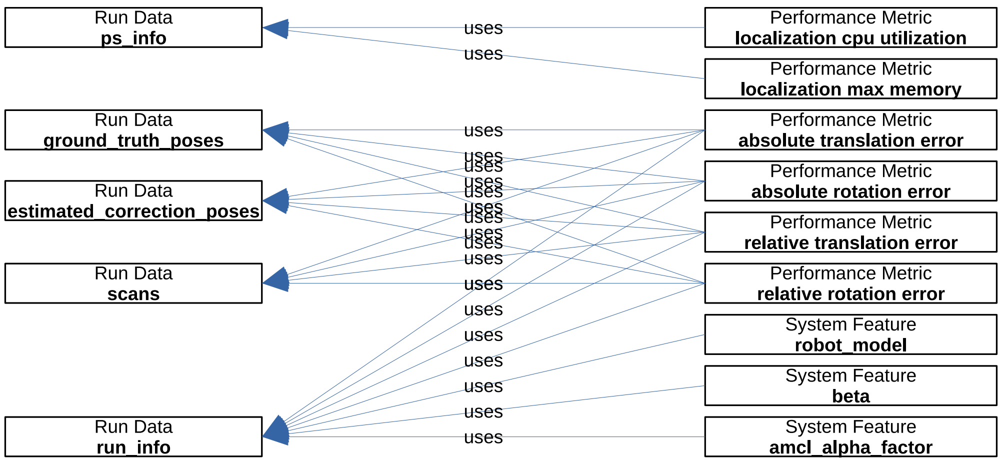


Figure 4.12: *Part 2 of 5 of the Localization And Local Planning (LALP) performance model diagram, listing the entities used in the performance model with regard to the localization performance metrics and features: the run data, the performance metrics, the environment features and the system features. The relationships in this diagram show which run data is used to compute each metric and feature.*

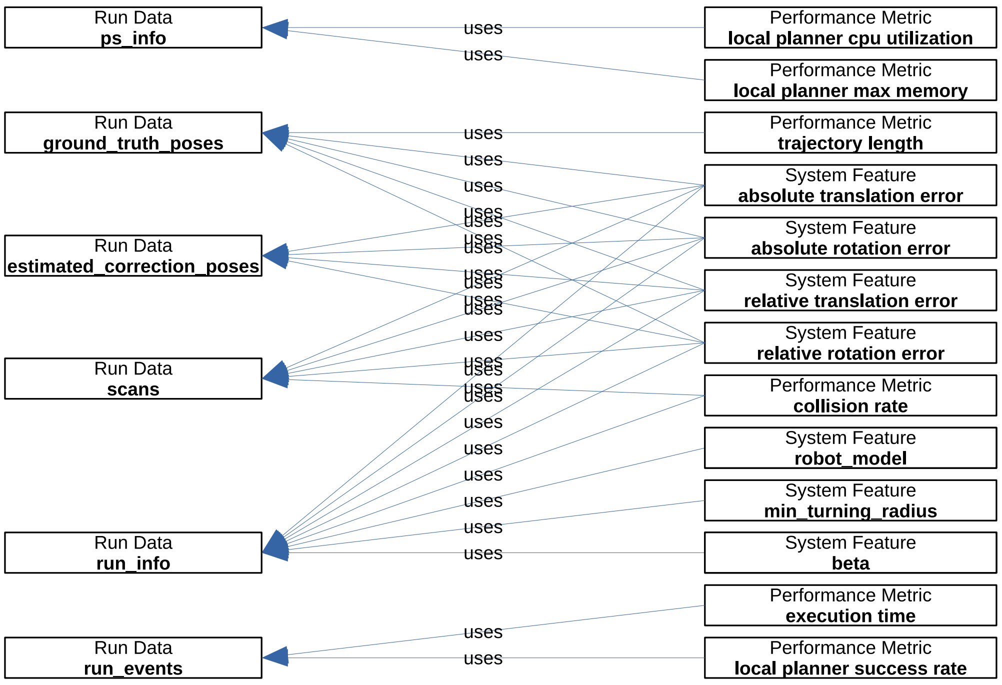


Figure 4.13: *Part 3 of 5 of the Localization And Local Planning (LALP) performance model diagram, listing the entities used in the performance model with regard to the local planning performance metrics and features: the run data, the performance metrics, the environment features and the system features. The relationships in this diagram show which run data is used to compute each metric and feature.*

formance metric statistical representations included in the performance model, which performance metric they predict, and which features are used to predict the performance metric. The entities used to evaluate the localization component are shown in Fig. 4.14 and the entities used to evaluate the local planning components are shown in Fig. 4.15.

In the following sections we describe in details the experimental setup, the experimental protocol, the run parameters used to configure the experimental setup, the run data which we collect from the experiments, and finally the performance metrics we use to evaluate the localization component and the local planning components.

4.2.1 Experimental Setup

The experimental setup used to evaluate the localization and local planning components is very similar to the one used in the SLAM use case, although there are some differences, which we will point out while we describe it.

The environments used to conduct the experiments are implemented using the Gazebo simulator, like in the SLAM use case, although we developed and use more environments in this use case. As in the SLAM use case, Gazebo is sufficient to simulate an environment with the features that impact the performance of the local planning and localization components, which only rely on information from a 2D LIDAR sensor, wheel odometry and 2D environment information, such as the gridmap used by the localization component to estimate the pose of the robot. The simulation consists of a 3D

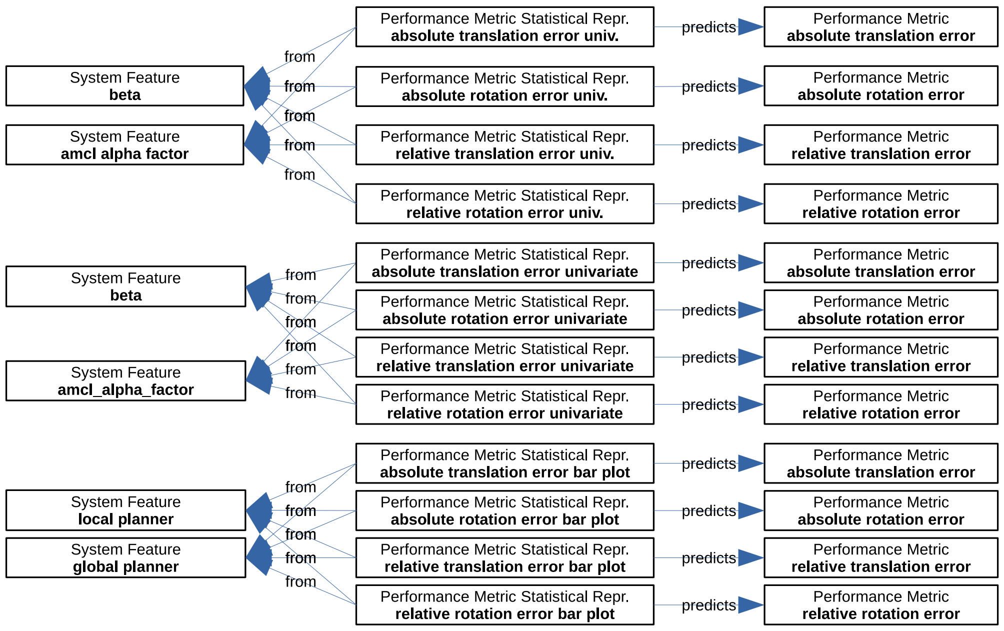

System Feature
beta
System Feature
amcl alpha factor
from
Performance Metric Statistical Repr.
absolute translation error univ.
Performance Metric Statistical Repr.
absolute rotation error univ.
Performance Metric Statistical Repr.
relative translation error univ.
Performance Metric Statistical Repr.
relative rotation error univ.
predicts
Performance Metric
absolute translation error
Performance Metric
absolute rotation error
Performance Metric
relative translation error
Performance Metric
relative rotation error
System Feature
amcl_alpha_factor
Performance Metric Statistical Repr.
absolute translation error univariate
Performance Metric Statistical Repr.
absolute rotation error univariate
Performance Metric Statistical Repr.
relative translation error univariate
Performance Metric Statistical Repr.
relative rotation error univariate
System Feature
local planner
System Feature
global planner
Performance Metric Statistical Repr.
absolute translation error bar plot
Performance Metric Statistical Repr.
absolute rotation error bar plot
Performance Metric Statistical Repr.
relative translation error bar plot
Performance Metric Statistical Repr.
relative rotation error bar plot

Figure 4.14: *Part 4 of 5 of the Localization And Local Planning (LALP) performance model diagram, listing the entities used in the performance model with regard to the localization performance metrics and features: the performance metrics, the system features, the environment features, and the performance metric statistical representations. The relationships in this diagram show which performance metric is predicted by each statistical representation, and from which features they are predicted.*

representation of a physical environment and a 3D robot model. The Gazebo models of the robot and environments are made available for experiment replication[15].

As in the SLAM use case, we craft the 3D environments by extruding the 2D gridmaps in Fig. 4.16. Some of these environment gridmaps are equivalent to the SLAM use case: airlab, 7A-2, office_b, fr079, while the gridmaps intel and mexico are only used in this use case. Three environments (fr079, intel, mexico) are generated from a gridmap obtained by running GMapping on real data from the radish dataset[16] [19], as was done in the SLAM use case. The additional environments, intel and mexico, both represent realistic scenarios indoor spaces, although mexico is a large, mostly uncluttered environment, while intel is a highly cluttered environment with tight passages.

The robots modeled are the Turtlebot 3 Waffle, a robot platform widely used both in simulation and real world and the Hunter 2.0[17] produced by AgileX Robotics, an Ackermann steering platform. The constraints of the Ackermann kinematic model, consisting of the maximum angle that can be applied to the steering wheels, increase the difficulty of the navigation. In order to evaluate the effect of the kinematic constraints on the performance of the local and global planners, we use the system feature *min turning radius*, derived from the maximum steering angle of the robot and its wheelbase.

Both robots have two sensors: a LIDAR and a differential odometry sensor. The

[15]`https://github.com/AIRLab-POLIMI/performance_modelling_test_datasets/tree/local-planning-devel`

[16]The data was provided by Cyrill Stachniss, Dirk Hähnel, and Nick Roy, and is available at `http://ais.informatik.uni-freiburg.de/slamevaluation/datasets.php` and `https://dspace.mit.edu/handle/1721.1/62236`. Accessed on 2022-07-29.

[17]`https://global.agilex.ai/products/hunter-2-0`. Accessed on 2022-07-26.

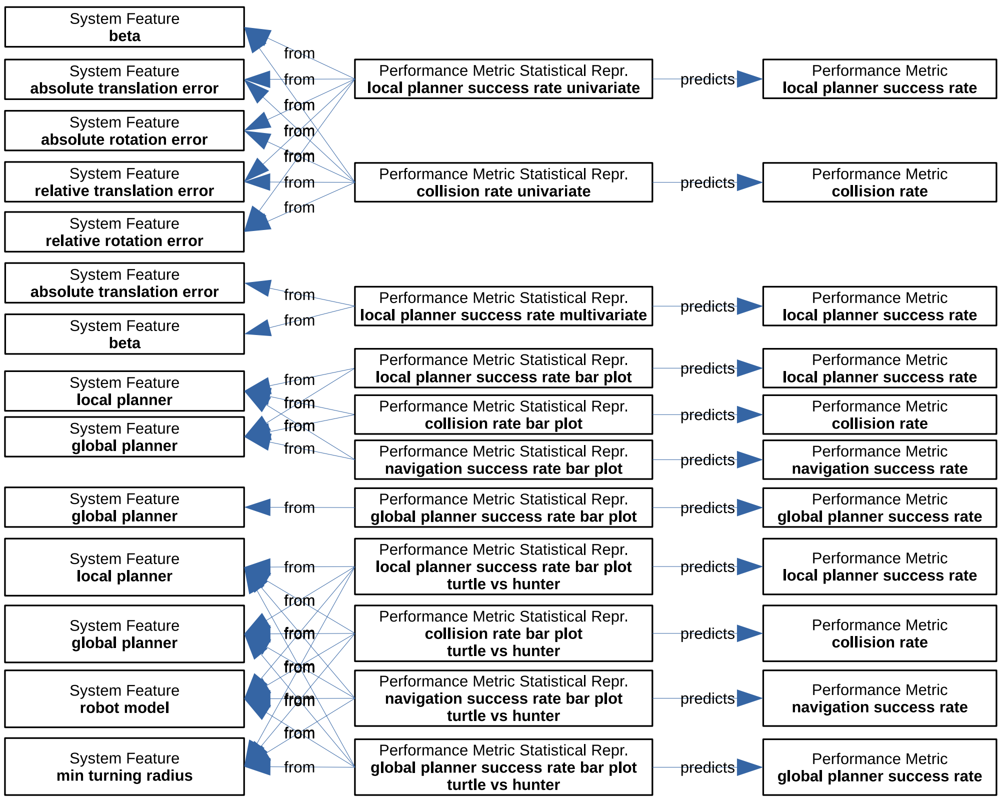

System Feature beta
System Feature absolute translation error
System Feature absolute rotation error
System Feature relative translation error
System Feature relative rotation error
from
Performance Metric Statistical Repr. local planner success rate univariate
Performance Metric Statistical Repr. collision rate univariate
predicts
Performance Metric local planner success rate
Performance Metric collision rate
System Feature absolute translation error
System Feature beta
Performance Metric Statistical Repr. local planner success rate multivariate
System Feature local planner
System Feature global planner
Performance Metric Statistical Repr. local planner success rate bar plot
Performance Metric Statistical Repr. collision rate bar plot
Performance Metric Statistical Repr. navigation success rate bar plot
Performance Metric navigation success rate
Performance Metric Statistical Repr. global planner success rate bar plot
Performance Metric global planner success rate
System Feature robot model
System Feature min turning radius
Performance Metric Statistical Repr. local planner success rate bar plot turtle vs hunter
Performance Metric Statistical Repr. collision rate bar plot turtle vs hunter
Performance Metric Statistical Repr. navigation success rate bar plot turtle vs hunter
Performance Metric Statistical Repr. global planner success rate bar plot turtle vs hunter

Figure 4.15: *Part 5 of 5 of the Localization And Local Planning (LALP) performance model diagram, listing the entities used in the performance model with regard to the local planner performance metrics and features: the performance metrics, the system features, the environment features, and the performance metric statistical representations. The relationships in this diagram show which performance metric is predicted by each statistical representation, and from which features they are predicted.*

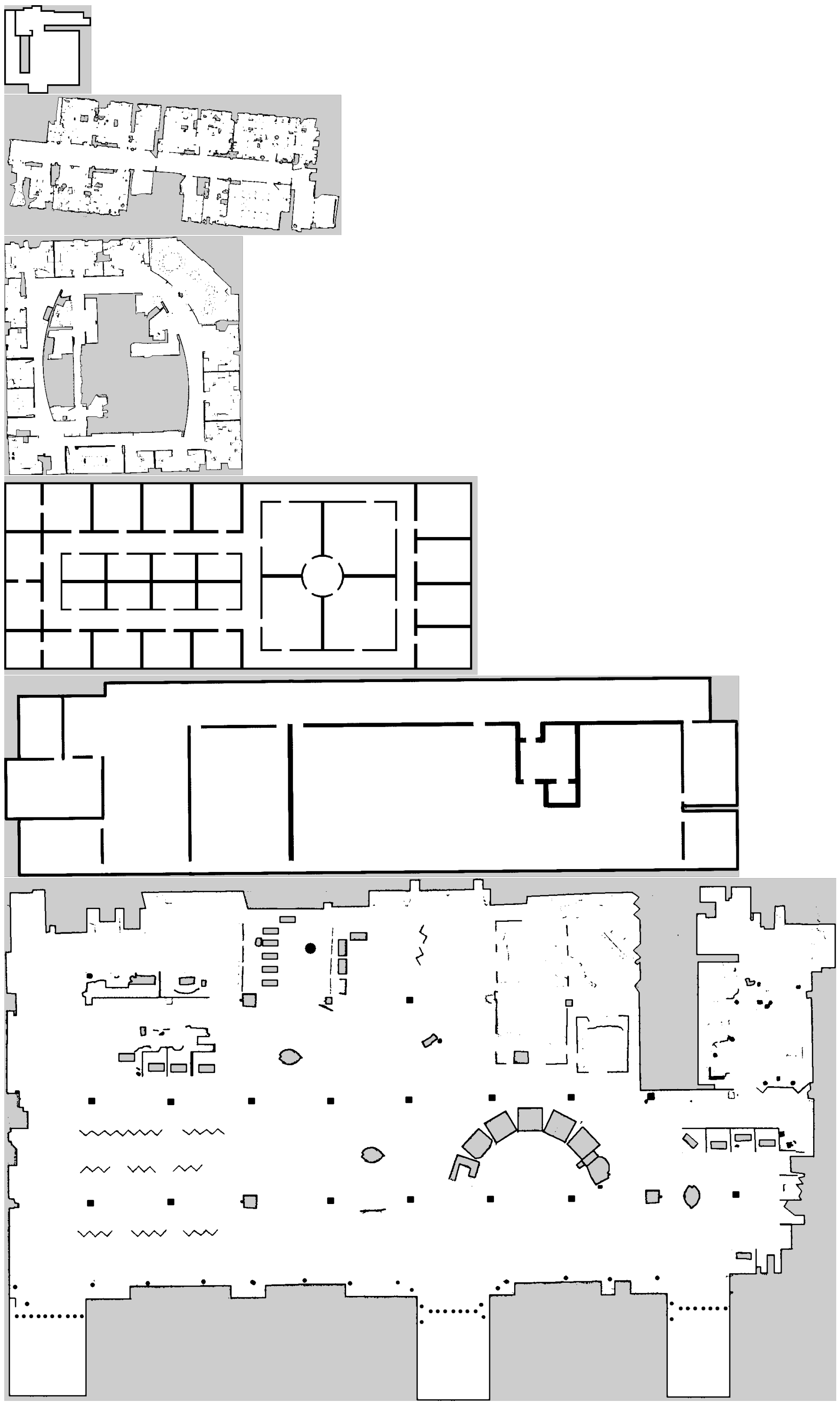


Figure 4.16: *The gridmaps of the experimental environments (top to bottom): airlab, fr079, intel, office_b, 7A-2, mexico. All gridmaps have the same scale, 0.05m per pixel.*

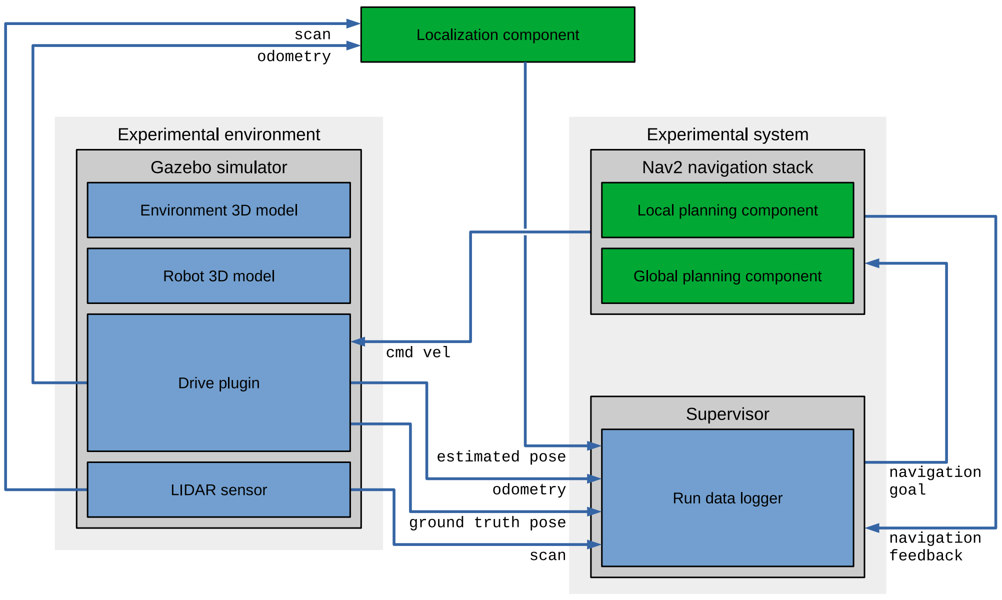


Figure 4.17: *The software used to implement the experimental setup. The software components being evaluated are highlighted in green. The edges of the graph represent the communication between the software.*

LIDAR sensor is simulated with the default Gazebo plugin. The measurement noise is implemented as Gaussian noise added to each range measurement independently from the range value (constant standard deviation of 0.01m and zero mean). The angular increment between beams is 1 degree. The measurement frequency is set to 10Hz.

The odometry data is computed from the differential drive[18] and Ackermann drive[19] Gazebo plugins that we modified to simulate an odometry sensor affected by a rotational and translational drift. The amount of odometry drift β has a significant effect on the performance, therefore we consider the odometry error a system feature and run experiments with multiple values.

The navigation stack used in the experimental system, Nav2[20], is composed by multiple parts, three of which are important in our setup: the local planner (called controller in the Nav2 terminology), the global planner (called planner in the Nav2 terminology) and the BT navigator, which receives navigation requests from the supervisor and coordinates the parts of the navigation stack based on a behavior tree.

The diagram in Fig. 4.17 illustrates the parts of the experimental system and experimental environment, the software components, and the communication between them.

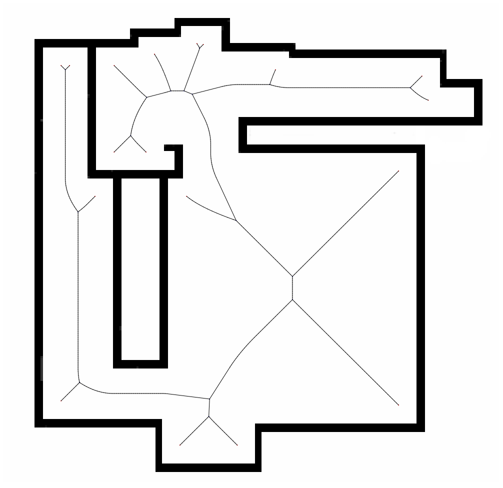
4.2.2 Experimental Protocol

The navigation procedure used to acquire the data to build our performance model consists in navigating between a start pose and a goal pose. The start pose is manually selected in each environment. The goal pose is picked using the run parameter *run*

[18]`https://github.com/AIRLab-POLIMI/gazebo_ros_pkgs/blob/foxy/gazebo_plugins/src/gazebo_ros_diff_drive.cpp`. Accessed on 2022-07-29.

[19]`https://github.com/AIRLab-POLIMI/gazebo_ros_pkgs/blob/foxy/gazebo_plugins/src/gazebo_ros_ackermann_drive.cpp`. Accessed on 2022-07-29.

[20]`https://navigation.ros.org/`. Accessed on 2022-07-29.


(a) *Voronoi graph.*

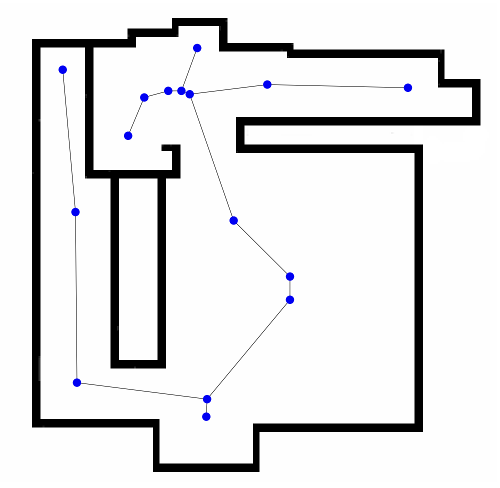

(b) *Graph obtained from the Voronoi graph (Fig. 4.18a). The vertices (blue dots) are used as navigation goal positions.*

Figure 4.18: *Graph used to compute the set of navigation goals.*

index from the set of navigation locations which are automatically generated from the 2D gridmap. The orientation of the goal pose is randomly sampled between 0 and τ. The set of navigation locations is the set of vertices of the graph in Fig. 4.18. The graph is obtained by computing a Delanuay triangulation based on the occupied cells of the gridmap and the corresponding Voronoi graph shown in Fig. 4.18a, which is then reduced to a simpler graph by keeping only the nodes connected to one or three edges, and then by removing leaf nodes.

The run is terminated when the BT navigator terminates the execution of the navigation request, which can be a success or failure to reach the goal pose, or when a timeout occurs.

4.2.3 Run Parameters

The run parameters used have been chosen to test three local planning components and the localization component in different environments, with different sensors and different kinematic models. The set of components to use in the run are selected with three run parameters: *localization component*, *local planner component*, and *global planner component*.

The parameter *robot model* specifies which robot is used in the run, therefore specifying the kinematic model. The Turtlebot robot is a differential drive and the Hunter robot is an Ackermann drive. The *max steering angle* parameter specifies the maximum angle of the steering wheels of the robot. This parameter can also be specified for the Turtlebot robot and it is set to 90 degrees, which is the equivalent value for a differential drive robot.

The *odometry error* run parameter is a vector describing the four components of odometric error, although in this setup the implementation is different from the setup of the SLAM experiments. Rather than computing a random error, the odometry sensors are affected by a constant drift. The error caused by drift is specified for these four components: *beta_1* controls the amount of odometry rotation error caused by the effective rotation, *beta_2* controls the amount of odometry rotation error caused by the effective translation, *beta_3* controls the amount of odometry translation error caused by the effective translation, and *beta_4* controls the amount of odometry translation error caused by the effective rotation. Although in a real robot the odometric error would manifest in all four components, to reduce the number of run parameters combinations, only *beta 1* and *beta 3* are used and set to the same value, which we call *beta*. This run parameter has been chosen to limit the quality of the sensor information which is available to the localization component.

When the localization component is set to AMCL and the odometry error is greater than 0, the run parameter *AMCL alpha factor* is used to specify the alpha parameters of the AMCL component. The alpha parameters of AMCL specify the expected process noise in the rotation and translation of the odometry estimate from rotation and translation. Specifically, *alpha 1* is the expected process noise in odometry's rotation estimate from rotation, *alpha 2* is the expected process noise in odometry's rotation estimate from translation, *alpha 3* is the expected process noise in odometry's translation estimate from translation, *alpha 4* is the expected process noise in odometry's translation estimate from rotation. The component parameter *alpha factor* specifies the value of the parameters $alpha_{1..4}$ of AMCL[21] in relation to the actual amount of odometry error *beta*: $alpha_i = beta_i \times alpha\_factor$. When the localization component is set to AMCL and the odometry error is 0, the alpha parameters are set to a small value, but greater than 0. AMCL would not work with alpha parameters set to 0.

Environment name is used to select which of the available environments is used in each run. *Run index* is used to select the goal pose from the list of positions generated for each environment. *Run replication index* is used to create replication of the experiments and it does not cause any change in the experimental setup.

Some parameters are not valid in some combinations, therefore it is necessary to specify three pairs of parameter grids. Specifically, *max steering angle* is only specified when the *local planner component* parameter is set to TEB, The parameter *AMCL alpha factor* can only be specified when the odometry error is greater than 0. When the odometry error is set to 0, the alpha parameters of AMCL will be set to a fixed value. The run parameter grids in Listing 4.3 produce the experiments for the Hunter robot using the TEB local planner and the SMAC global planner. The run parameter grids in Listing 4.4 produce the experiments for the Turtlebot robot using the TEB local planner and both NavFN and SMAC global planners. The run parameter grids in Listing 4.5 produce the experiments for the Turtlebot robot using the DWB and RPP local planners, and the NavFN global planner.

4.2.4 Run Data

The following run data are collected in each run:

- run_info: this run data stores the value of the run parameters with which the run is configured.
- run_events: this run data is a series of events identified by a name and a timestamp. The relevant events are: run_start identifies when the run has started, navi-

[21] `https://navigation.ros.org/configuration/packages/configuring-amcl.html`

```
run_duplicate_index: &id003 [1, 2]
run_index: &id001 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
environment_name: &id002 [7A-2, airlab, fr079, intel, mexico, office_b]
combinatorial_parameters: [
  {
    localization_component: [amcl],
    local_planner_component: [teb],
    global_planner_component: [smac],
    robot_model: [hunter2],
    max_steering_angle_deg: [20, 40],
    odometry_error: [[0.0, 0.0, 0.0, 0.0]],
    environment_name: *id002,
    run_index: *id001,
    run_duplicate_index: *id003,
  },
  {
    localization_component: [amcl],
    local_planner_component: [teb],
    global_planner_component: [smac],
    robot_model: [hunter2],
    max_steering_angle_deg: [20, 40],
    odometry_error: [[0.02, 0.0, 0.02, 0.0], [0.05, 0.0, 0.05, 0.0], [0.1, 0.0, 0.1,
    0.0]],
    amcl_alpha_factor: [0.5, 0.75, 1.0, 1.5, 2.0],
    environment_name: *id002,
    run_index: *id001,
    run_duplicate_index: *id003,
  },
]
```

Listing 4.3: *Run parameter grid used in the Localization and Local Planning performance model, for the experiments with the Hunter 2 robot.*

gation_goal_sent indicates that the supervisor successfully sent a navigation goal to the navigation stack, target_pose_reached indicates that the navigation to the goal pose was successful, navigation_failed indicates that the navigation stack failed to reach the goal pose, run_timeout indicates the run has has terminated because of timeout, run_completed indicates the run was finished, supervisor_finished indicates the supervisor was about to terminate.

- scans: this run data stores a series of laser scan readings, including information on the time of generation of the data from the sensor, the characteristics of the sensor (i.e., field of view, minimum and maximum range, and number of beams), and the list of range measurements.
- estimated_poses: this run data contains a series of the poses of the robot as estimated by the localization component, including the time. The pose is sampled at 10Hz.
- estimated_correction_poses: this run data contains a series of the poses of the robot as estimated by the localization component, including the time. The poses are added to the data when received by the localization component after updating its estimate.
- ground_truth_poses: this run data contains a series of the ground truth poses of the robot as published by the simulator, including the time. The pose is sampled every time the simulator publishes the information, which is around 100Hz.

```
run_duplicate_index: &id003 [1, 2]
run_index: &id001 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
environment_name: &id002 [7A-2, airlab, fr079, intel, mexico, office_b]
combinatorial_parameters: [
  {
    localization_component: [amcl],
    local_planner_component: [teb],
    global_planner_component: [navfn, smac],
    robot_model: [turtlebot3_waffle_performance_modelling],
    max_steering_angle_deg: [90],
    odometry_error: [[0.0, 0.0, 0.0, 0.0]],
    environment_name: *id002,
    run_index: *id001,
    run_duplicate_index: *id003,
  },
  {
    localization_component: [amcl],
    local_planner_component: [teb],
    global_planner_component: [navfn, smac],
    robot_model: [turtlebot3_waffle_performance_modelling],
    max_steering_angle_deg: [90],
    odometry_error: [[0.02, 0.0, 0.02, 0.0], [0.05, 0.0, 0.05, 0.0], [0.1, 0.0, 0.1,
    0.0]],
    amcl_alpha_factor: [0.5, 1.0, 2.0],
    environment_name: *id002,
    run_index: *id001,
    run_duplicate_index: *id003,
  },
]
```

Listing 4.4: *Run parameter grid used in the Localization and Local Planning performance model, for the experiments with the Turtlebot robot and the TEB local planner.*

- ps_info: this run data contains a series of snapshots of all the experimental environment and system processes running on the machine, obtained with the *psutil* Python library[22]. This allows to retrieve information about the processes such as memory and CPU utilization. The snapshots are collected at 1Hz.

4.2.5 Localization Performance Metrics

The output of a localization method is the estimated robot pose, which is communicated to the other components in the system by updating a transform between the map and odom frames. We characterize the localization component using the following metrics: the *translation relative error* ε_T, the *rotation relative error* ε_R, the *absolute translation error* and the *absolute rotation error*.

The *translation relative error* ε_T and the *rotation relative error* ε_R measure the mean translation and rotation relative errors that are accumulated in the estimation of the

[22]`https://psutil.readthedocs.io/en/latest/#psutil.Process`. Accessed on 2022-07-01.

```
run_duplicate_index: &id003 [1, 2]
run_index: &id001 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
environment_name: &id002 [7A-2, airlab, fr079, intel, mexico, office_b]
combinatorial_parameters: [
  {
    localization_component: [amcl],
    local_planner_component: [dwb, rpp],
    global_planner_component: [navfn],
    robot_model: [turtlebot3_waffle_performance_modelling],
    odometry_error: [[0.0, 0.0, 0.0, 0.0]],
    environment_name: *id002,
    run_index: *id001,
    run_duplicate_index: *id003,
  },
  {
    localization_component: [amcl],
    local_planner_component: [dwb, rpp],
    global_planner_component: [navfn],
    robot_model: [turtlebot3_waffle_performance_modelling],
    odometry_error: [[0.02, 0.0, 0.02, 0.0], [0.05, 0.0, 0.05, 0.0], [0.1, 0.0, 0.1,
    0.0]],
    amcl_alpha_factor: [0.5, 1.0, 2.0],
    environment_name: *id002,
    run_index: *id001,
    run_duplicate_index: *id003,
  },
]
```

Listing 4.5: *Run parameter grid used in the Localization and Local Planning performance model, for the experiments with the Turtlebot robot and the DWB and RPP local planners.*

robot pose between estimation updates. The metrics are computed as

$$\varepsilon_T = mean_{i,j} ||trans(\delta^E_{i,j} \ominus \delta^{GT}_{i,j})|| \tag{4.5}$$

$$l_{i,j} = \sum_{k=i}^{j-1} ||trans(p^{GT}_{k+1} \ominus p^{GT}_k)|| \tag{4.6}$$

$$\varepsilon_R = mean_{i,j} |rot(\delta^E_{i,j} \ominus \delta^{GT}_{i,j})| \tag{4.7}$$

$$r_{i,j} = \sum_{k=i}^{j-1} |rot(p^{GT}_{k+1} \ominus p^{GT}_k)| \tag{4.8}$$

$$ATE = \frac{1}{N} \sum_{k=0}^{N-1} ||trans(p^E_k \ominus p^{GT}_k)|| \tag{4.9}$$

$$ARE = \frac{1}{N} \sum_{k=0}^{N-1} |rot(p^E_k \ominus p^{GT}_k)| \tag{4.10}$$

i, j are time indices for the estimated poses and ground truth poses from the run data. These two series of poses are sampled at different times, therefore, to have matching indices, the ground truth series is interpolated in time. U is the set of pairs of consecutive indices of the estimated poses, i.e., $\{(0,1),(1,2),...\}$. $\delta^E_{i,j}$ and $\delta^{GT}_{i,j}$ are the homogeneous transformations from the i-th to the j-th estimated pose and ground truth pose respectively. p^{GT}_k are ground truth poses. $\ominus$ is the inverse of the transformation composition operator. $trans$ is the translation component of an homogeneous transformation or the translation between poses. rot is the rotation component of an homogeneous transformation. The relative errors disregard the error accumulated before the estimation update w_a, and it only considers the error accumulated between estimation updates.

The *Absolute Translation Error (ATE)* and *Absolute Rotation Error (ARE)* are computed as in Eq. 4.9 and 4.10.

4.2.6 Local Planning Performance Metrics

The metric *navigation success rate* indicates whether the navigation was successful. The navigation is considered successful if the navigation stack has reached the navigation goal pose within some tolerance, and if there was no collision between the robot and the environment. The collision is detected when any point corresponding to the LIDAR measurements fall within the footprint of the robot at any time during the run. The footprint of the robot is the geometrical representation of the shape of the robot for the local and global planners. Two additional metrics are derived from the *navigation success rate*, the *global planner success rate* and the *local planner success rate*. The *global planner success rate* indicates whether the global planner was able to produce a global plan that the local planner could attempt to follow. The *local planner success rate* indicates whether the navigation was successful, and the metric is only defined for the runs in which the global planner was successful.

$$l_{run} = \sum_{k=0}^{N-1} ||trans(p_{k+1}^{GT}, p_k^{GT})|| \tag{4.11}$$

$$L^{norm} = \frac{l_{run}}{l_{shortest}} \tag{4.12}$$

The *normalized trajectory length*, L^{norm} (Eq. 4.12), is computed for each run, by dividing the length of the trajectory taken by the robot (Eq. 4.11) by the shortest trajectory length among the runs in which the navigation was successful (i.e., success rate = 1), and that were executed in the same environment and with the same start and goal positions ($l_{shortest}$ in Eq. 4.12). The lowest value of this metric is 1 for the runs with successful navigation. Runs with unsuccessful navigation may produce a value lower than one since the run may terminate before reaching the goal position. The value of p_k^{GT} is the $k-th$ ground truth position recorded in the *ground_truth_poses* run data.

$$t_{run} = t_{end} - t_{start} \tag{4.13}$$

$$T^{norm} = \frac{t_{run}}{t_{shortest}} \tag{4.14}$$

The *normalized execution time*, T^{norm} (Eq. 4.14), is computed for each run, by dividing the execution time (Eq. 4.13) by the shortest execution time among the runs in which the navigation was successful (i.e., success rate = 1), and that were executed in the same environment and with the same start and goal positions ($t_{shortest}$ in Eq. 4.14). The lowest value of this metric is 1 for the runs with successful navigation. Runs with unsuccessful navigation may produce a value lower than one since the run may terminate before reaching the goal position. The value of t_{start} corresponds to the time of the event *navigation_goal_accepted*, which corresponds to the time that the navigation stack acknowledges that the navigation request has been accepted. The value of t_{end} is the time of the event *navigation_succeeded* or event *navigation_failed*. One of these two events must always be present in the *run_events* run data. These events correspond to the navigation stack reporting that the navigation goal was reached or that the navigation has failed.

Another important aspect of these components is the computational resources used during the execution, i.e., the average CPU and maximum memory utilization. We compute the *CPU utilization* metric as the total user and system CPU time accumulated by the ROS node implementing the local planner component divided by the duration of the run. This metric is dependent on the processor hardware and can only be used to compare results obtained on the same machine or with the same hardware. We run all experiments on the same machine.

The *max memory* metric measures the maximum amount of memory allocated to the ROS node implementing the component during a run in MiB (2^{20} Bytes). The allocated memory is counted as Proportional Set Size, i.e., the memory which is unique to the ROS node's processes and which would be freed if those processes were terminated, plus the amount of memory shared with other processes divided by the number of processes using it.

CHAPTER 5

Experimental Results

In this chapter, we present the results of the two case studies used to demonstrate our methodology. We will describe how we use statistical models to predict the performance metrics from the system and environment features and to evaluate the effects of the performance of one component on other components. We will analyse these results, showing how to make use of the information provided by the performance models and which decisions are supported by the results.

5.1 SLAM Performance Model Results

In this section, we present the performance models relating to five metrics presented in Sec. 4.1.6. The results are only valid for the conditions in which we evaluate the performance, and our analysis is limited to the aspects of the performance that have been measured with the performance metrics included in our work. Other aspects may be considered to decide which component performs best depending on its use. For instance, the quality of the map produced by the SLAM component during or at the end of the exploration, which can be measured in different ways [41,42]. Characteristics of the methods or their implementation that are not strictly related to a measurable performance may also be important in deciding which component best suits an application, such as the data type used to represent the map or the ability to resume the SLAM process.

Our performance models aim at representing the stochastic relationship between the system features, the environment features and the performance of the SLAM components. In particular, in our setting we have four values for *laser scan max range* (3.5, 8.0, 15.0, 30.0) in meters, four values for *laser scan fov deg* (90, 180, 270, 359) and five values for the odometry noise β (0.0, 0.5, 1.0, 1.5, 2.0). The values of *translation* *geometric similarity* are discretized into five values, so that 0 corresponds to the lowest value of translation geometric similarity and 4 corresponds to the highest possible value. We execute 2,640 runs from which we obtain 285,410 datapoints corresponding to trajectories between successive waypoints in the four environments. The results for the performance models, each corresponding to a performance metric, are presented in the following sections.

The data used to compute the performance models is made available online[1]. The experiments are executed with 4 threads of a Intel® Xeon® CPU E5-2687W v4 and 252GB of RAM memory on Ubuntu 18.04. The code used to implement and run the experiments, and the data for robots and environments is made available in our repository[2].

5.1.1 Normalized Relative Translation Error Metrics

We have fitted a set of generalized linear regression univariate models predicting the performance from each individual feature. The features *laser scan field of view* and *laser scan max range* are used in the linear model after computing their inverse (i.e., $1/feature$) and we fit a second degree polynomial model with interaction variables for max range, field of view, and beta features, while we fit a fourth degree polynomial model with interaction variables for the translation geometric similarity feature. Each plot in Fig. 5.2 shows the measured performance data and the prediction of the univariate models. Note that the legend of these plots, and all following plots, is reported in Fig. 5.1. The univariate models allow us to visualize the dependency of the component performance for each system and environment feature. For instance, from the model in Fig. 5.2a, we can tell that the performance of SLAM Toolbox degrades with a laser scan max range lower than 8 meters, and a laser scan max range equal or greater than 8 meters will not significantly increase its performance. From the model in Fig. 5.2b, we can tell the dependency of SLAM Toolbox on the laser scan field of view is greater than GMapping, and that GMapping is able to perform better than SLAM Toolbox with a laser scan field of view lower than 180 degrees. From the model in Fig. 5.2c, we can tell the dependency of GMapping on the odometry noise beta is greater than SLAM Toolbox, and that SLAM Toolbox is able to perform better than GMapping with very high and very low beta values. Finally, from the model in Fig. 5.2d, we can tell Hector SLAM suffers from performance degradation with high translation geometric similarity, which can be explained by the fact that Hector SLAM only relies on the LIDAR sensor and does not use the odometry information.

A drawback of these univariate models is that the performance refers to a system with average features. For example SLAM Toolbox always outperforms GMapping if we only consider the laser scan max range (Fig. 5.2a), nevertheless GMapping can outperform SLAM Toolbox when we consider the laser scan field of view (Fig. 5.2b). To get a better insight into these aspects we can use a multivariate generalized linear model comprising all our features.

Also with the multivariate model, features *laser scan field of view* and *laser scan max range* are first inverted to be used in the model. We used in this case a second

[1]`https://doi.org/10.5281/zenodo.5482936`. Accessed on 2022-02-02.

[2]`https://github.com/AIRLab-POLIMI/slam_performance_modelling/tree/melodic-devel`. Accessed on 2022-02-02.

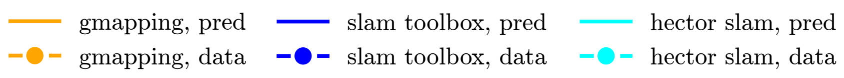


Figure 5.1: *Legend for all performance model plots: measured performance data (points connected by dashed lines) and prediction of the models (continuous lines) for GMapping (orange), SLAM Toolbox (blue) and Hector SLAM (cyan).*

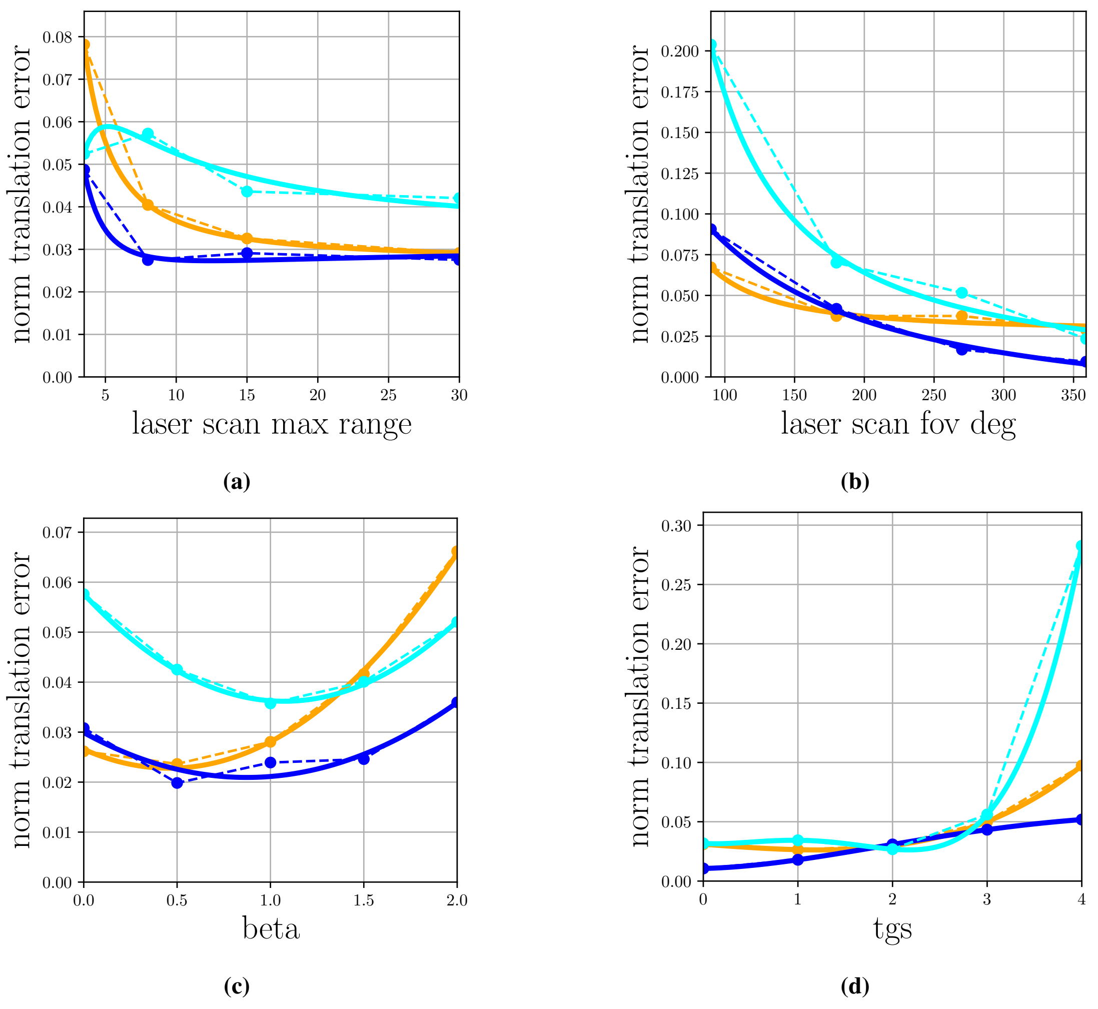


Figure 5.2: *Normalized relative translation error univariate performance model. Legend in Fig. 5.1. Lower values are better.*

degree polynomial model with interaction variables for all features, as shown in Eq. 5.1, where $\hat{y}$ is the predicted value, $\beta_{i_1 i_2 i_3 i_4}$ are the regression coefficients, and $x_1 ... 4$ are the four features. In Fig. 5.3 four of the 100 plots that can be generated to compare the measured performance data and the prediction of the models are shown. These models allow us to visualize the dependency of the component performance from one system or environment feature (laser scan max range in the four examples in Fig. 5.3), while the rest of the features are fixed to specific values. Fixing a set of features allows us to compare the predicted performance of the components for a system and environment pair characterized by those features.

$$
\begin{aligned}
\hat{y} = {} & \beta_{0000} + \beta_{1000}x_1 + \beta_{0100}x_2 + \beta_{0010}x_3 + \beta_{0001}x_4 \\
& + \beta_{2000}x_1^2 + \beta_{0200}x_2^2 + \beta_{0020}x_3^2 + \beta_{0002}x_4^2 \\
& + \beta_{1100}x_1x_2 + \beta_{1010}x_1x_3 + \beta_{1001}x_1x_4 \\
& + \beta_{0101}x_2x_4 + \beta_{0110}x_2x_3 + \beta_{0011}x_3x_4
\end{aligned}
\tag{5.1}
$$

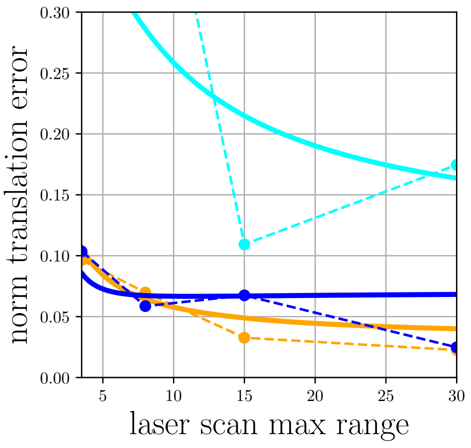
With the multivariate model, we can now tell when SLAM Toolbox actually outperforms GMapping. In Fig. 5.3a we fix the laser scan field of view to 180 degrees, the odometry noise beta to 0 (ideal odometry sensor) and the translation geometric similarity of the environment to 4 (the highest value). In this setup, GMapping outperforms SLAM Toolbox when the laser scan max range is greater than 8 meters, and they perform similarly otherwise. By increasing the odometry noise beta to 2 (Fig. 5.3b), SLAM Toolbox now outperforms GMapping for every value of the laser scan max range. By upgrading the laser scan field of view from 180 to 359 degrees (Fig. 5.3c), the gap in performance between SLAM Toolbox and GMapping increases. By decreasing the translation geometric similarity from 4 to 3 (Fig. 5.3d), the performance of both components increases for all values of laser scan max range. Hector SLAM can outperform the other components in certain conditions when evaluated with the normalized relative metrics, but as discussed later, it is always outperformed when considering the success rate metric.

5.1.2 Other Performance Metrics

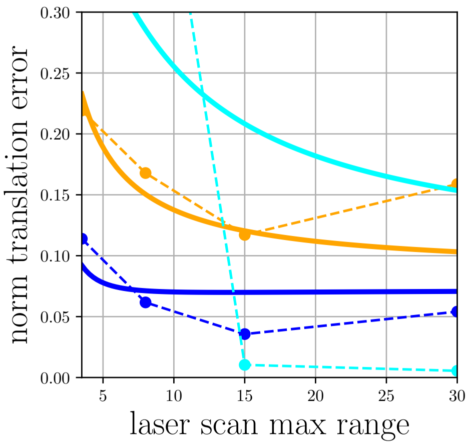
In Fig. 5.4 we can see the performance measured with the *Normalized relative rotation error* multivariate model for the same combinations of feature values as in Sec. 5.1.1. Note that the discrepancy between prediction and the data is due to the noise in multiple dimensions since this is also a multivariate model. SLAM Toolbox outperforms GMapping in most cases.

Fig. 5.5 shows two plots for the *success rate*, indicating the probability of the absolute error being above a threshold during the run. The threshold is chosen as 100m, by observing that above this value the SLAM component has definitely produced an invalid map. Not all maps with lower absolute error are valid, but this metric still allows us to compare the probability of failure for the components. GMapping and SLAM Toolbox have a negligible number of runs in which the absolute error is above the threshold, while for Hector SLAM, the values are significantly worse.

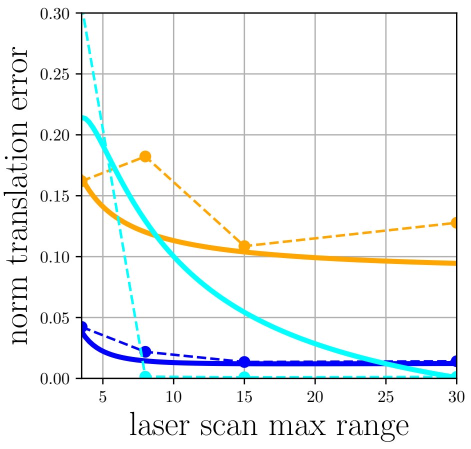
The *Absolute Trajectory Error model* allows us to evaluate the performance of the components relative to the complete exploration task. In Fig. 5.6 two plots from the model in which SLAM Toolbox outperforms Gmapping and Hector SLAM are shown.



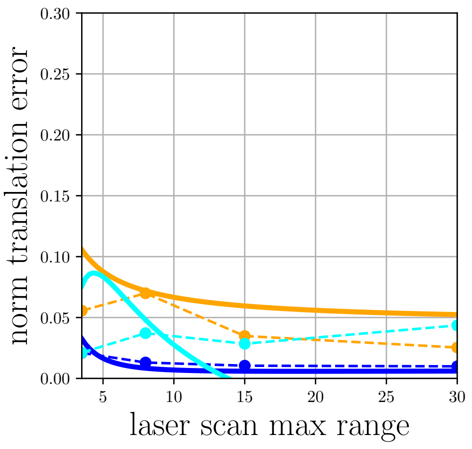
(a) *fov=180deg, beta=0, tgs=4*



(b) *fov=180deg, beta=2, tgs=4*

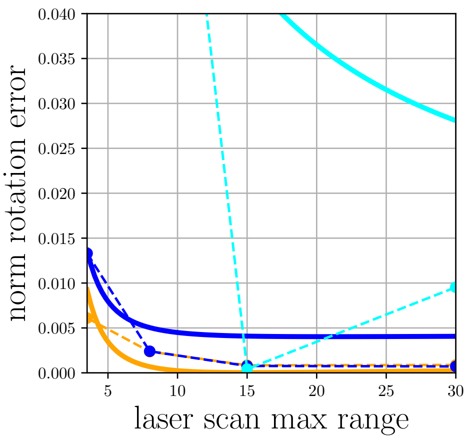


(c) *fov=359deg, beta=2, tgs=4*

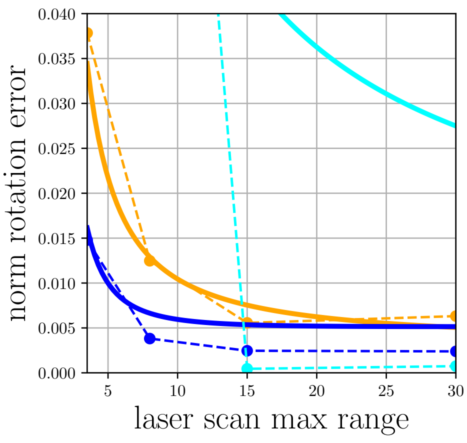


(d) *fov=359deg, beta=2, tgs=3*

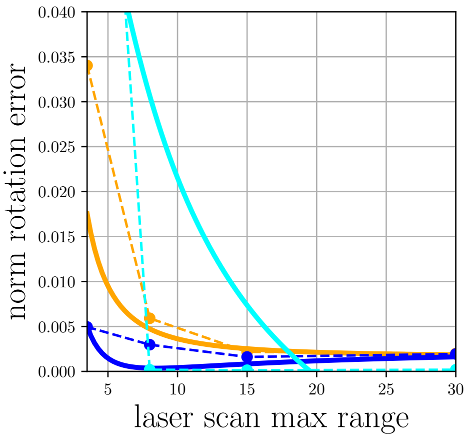
Figure 5.3: *Normalized relative translation error multivariate performance models. Legend in Fig. 5.1.*



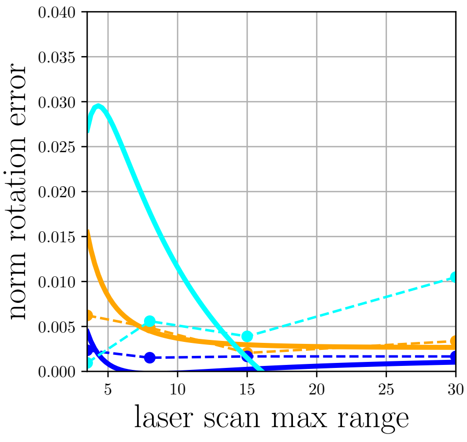
(a) *fov=180deg, beta=0, tgs=4*



(b) *fov=180deg, beta=2, tgs=4*

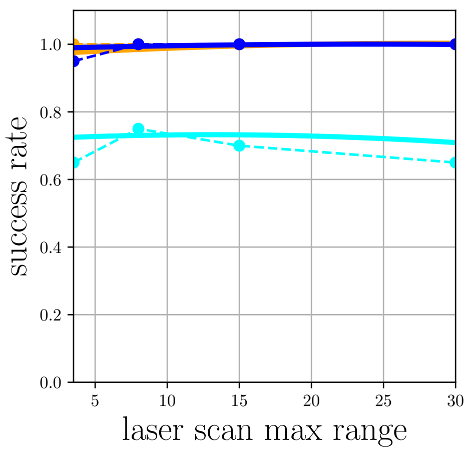


(c) *fov=359deg, beta=2, tgs=4*

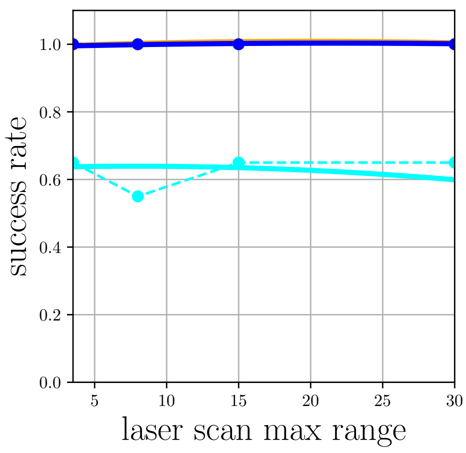


(d) *fov=359deg, beta=2, tgs=3*

Figure 5.4: *Normalized relative rotation error multivariate performance models. Legend in Fig. 5.1. Lower values are better.*

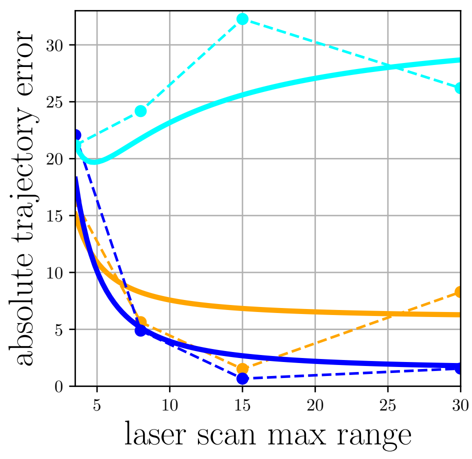


(a) *fov=180deg, beta=2*

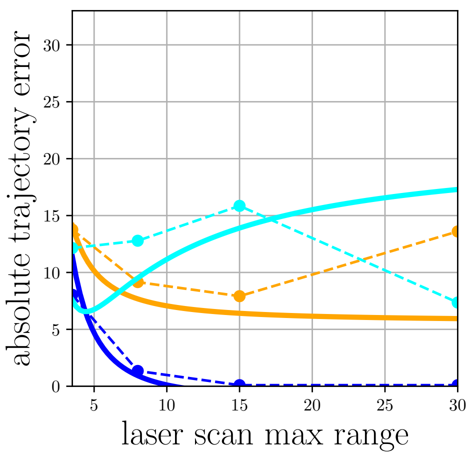


(b) *fov=359deg, beta=2*

Figure 5.5: *Success rate multivariate performance models. Legend in Fig. 5.1. Higher values are better.*



(a) *fov=180deg, beta=2*



(b) *fov=359deg, beta=2*

Figure 5.6: *Absolute trajectory error multivariate performance models. Legend in Fig. 5.1. Lower values are better.*

Note that the performance of Hector SLAM appears to be comparable with the other components, but the low success rate means that many runs incurred in a failure to produce a map with sufficient absolute error, and are therefore not included in these results. For this reason, this model and the success rate model must be evaluated jointly. The difference in the performance of Hector SLAM is attributable to the exclusive use of LIDAR information and not odometry, which increases the likelihood of a failure when the LIDAR information alone is not sufficient.

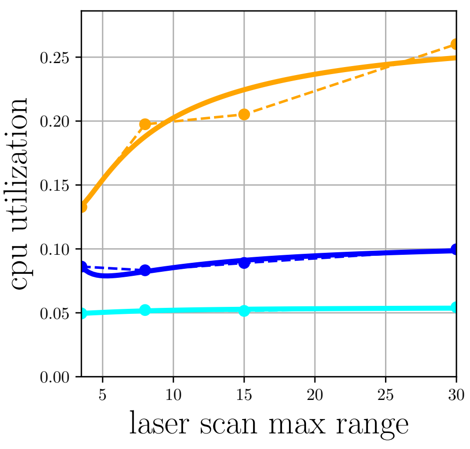
In Fig. 5.7 are shown the *CPU utilization* and *max memory* univariate models. Although we compute the multivariate model here as well, the univariate model allows us to provide an overview of the computational performance of the components. Hector SLAM utilizes fewer CPU resources than SLAM Toolbox, which performs better than GMapping. In Hector SLAM the map size has to be set to a predefined value before running the software, which explains the fixed and higher use of memory.

5.2 Localization and Local Planning Performance Model Results

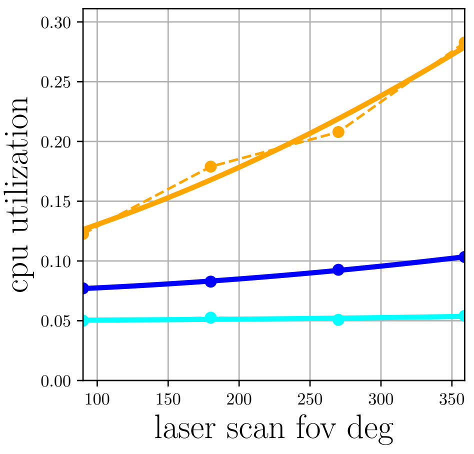
In this section, we present the performance models relating to the localization performance metrics (see Sec. 4.2.5 and Fig. 4.2.6). The results are only valid for the conditions in which we evaluate the performance, and our analysis is limited to the aspects of the performance that have been measured with the performance metrics included in our work.

Our performance models aim at representing the stochastic relationship between the system features, the performance of the localization and local planning components. The system features we set from run parameters are the odometry error, *beta*, and AMCL's alpha parameter, *amcl alpha factor*. Additionally, the *local planner* and *global planner* features are used to identify which local and global planner are used in each experiment.

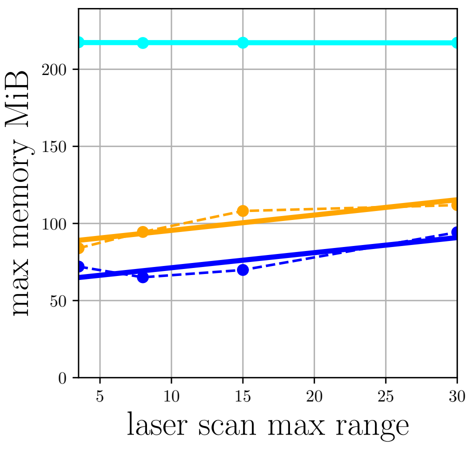
We execute 11531 runs, 11062 with the Turtlebot robot and 469 with the Hunter robot. The accumulated navigation time during the experiments is 226 hours (9.4 days),



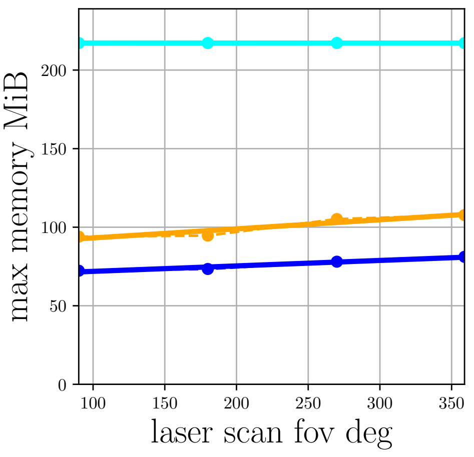
(a)



(b)

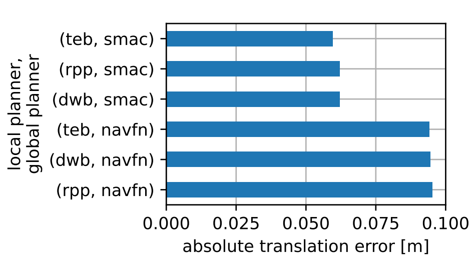


(c)

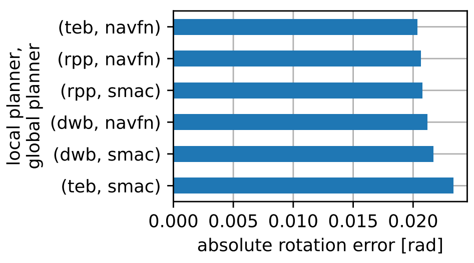


(d)

Figure 5.7: *CPU utilization and max memory univariate performance models. Legend in Fig. 5.1. Lower values are better.*

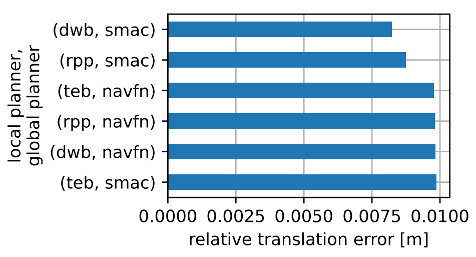


(a)

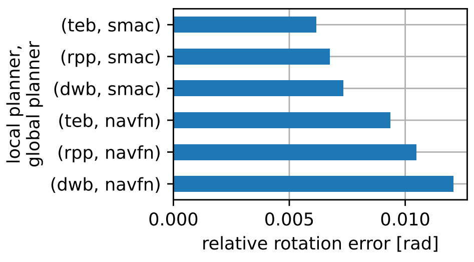


(b)

Figure 5.8: *Overall absolute translation and rotation error from the runs with the Turtlebot robot (lower is better).*



(a)



(b)

Figure 5.9: *Overall relative translation and rotation error from the runs with the Turtlebot robot (lower is better).*

202 with the Turtlebot robot and 24 with the Hunter robot. The experiments are executed with 16 threads of an AMD Ryzen™ 7 3700X CPU and 32GB of RAM memory on Ubuntu 20.04. The code used to implement and run the experiments, and the data for robots and environments is made available in our repository[3].

5.2.1 Localization Performance Model Results

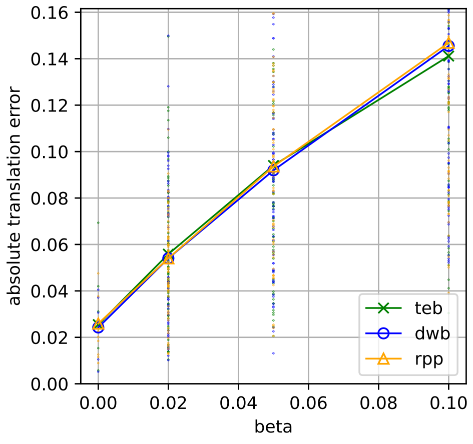
Fig. 5.8 and Fig. 5.9 show the overall localization error for each combination of local planner and global planner using the Turtlebot robot.

Fig. 5.8 shows the overall absolute translation and rotation error. There is a dependency between the performance of the localization component, specifically the absolute translation error, and which global planner is used. The average absolute translation error is 0.095 meters when using the NavFn global planner and 0.062 meters when using the SMAC global planner. The average absolute rotation error varies relatively little, between 0.02 and 0.023 radians.

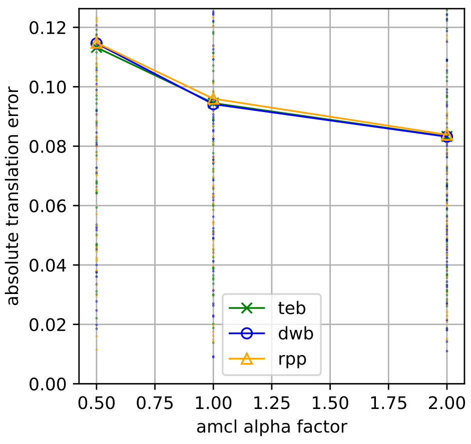
Fig. 5.9 shows the overall relative translation and rotation error. The average relative rotation error is 0.011 radians when using the NavFn global planner and 0.007 when using the SMAC global planner. The average relative translation error varies relatively little, between 0.008 and 0.010 meters, with the NavFn global planner causing the lowest overall localization performance.

Although the SMAC global planner causes lower absolute translation and relative

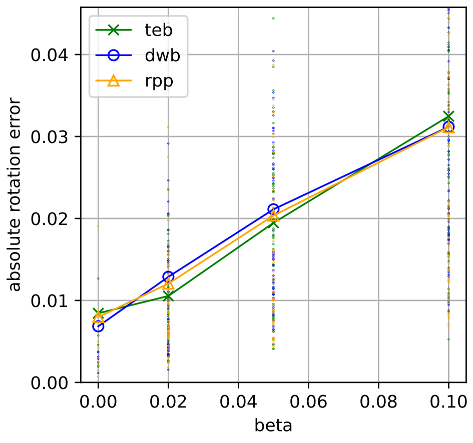
[3]`https://github.com/AIRLab-POLIMI/local_planning_performance_modelling/tree/foxy-devel`. Accessed on 2022-09-13.



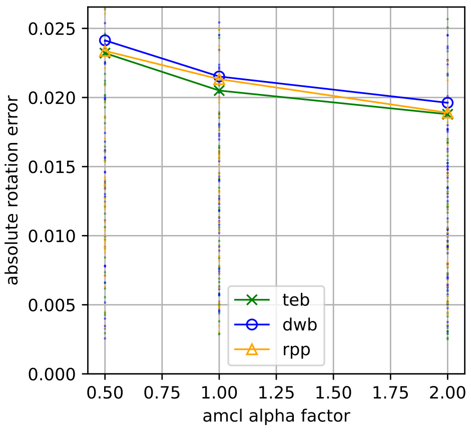
(a)



(b)



(c)



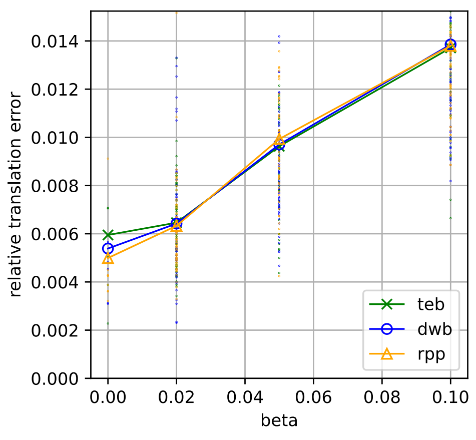
(d)

Figure 5.10: *Localization performance data for absolute error metrics (lower values are better).*

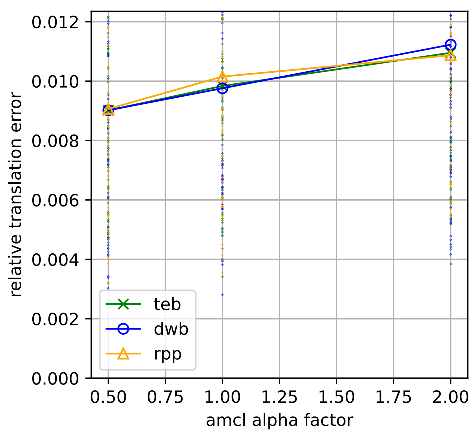
rotation errors, as shown later in Sec. 5.2.2, it also causes the probability of successfully reaching the navigation goal to be much lower, making it a poor choice in most realistic cases.

Fig. 5.10 and Fig. 5.11 show the localization error metrics resulting from different values of odometry error *beta* and *amcl alpha factor*. Only the data obtained using the Turtlebot robot and the NavFn global planner are considered. These plots are obtained by computing the mean of the performance metrics for each value of the system feature. The values corresponding to different local planners (i.e., TEB, DWB, RPP) are shown in different colors. The datapoints are also shown as a scatter plot.

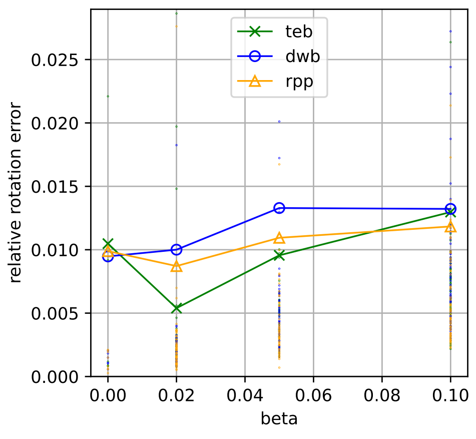
The absolute error is greatly influenced by the odometry error, as shown in Fig. 5.10a and Fig. 5.10c. Note that the absolute error metrics measure the error when the localization component updates the estimate of the robot pose, and not while the robot pose is integrating the odometry error. Therefore, the error measured by the metrics indicates that the localization component is not able to completely correct the odometry error. As shown in Fig. 5.10b and Fig. 5.10d, the absolute translation and rotation errors are greater for values of *amcl alpha factor* lower than 1, indicating that, in the case



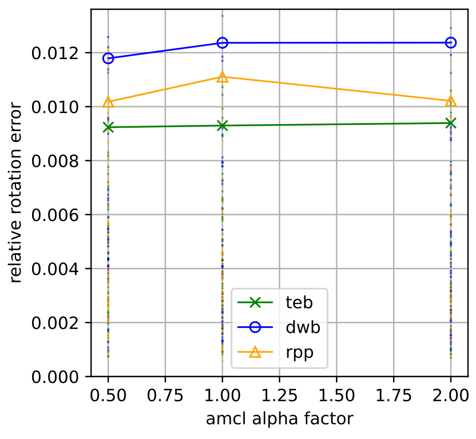
(a)



(b)



(c)



(d)

Figure 5.11: *Localization performance data for relative error metrics (lower values are better).*

of AMCL, to achieve a lower absolute error, it is better to over-estimate the odometry error rather than under-estimating it.

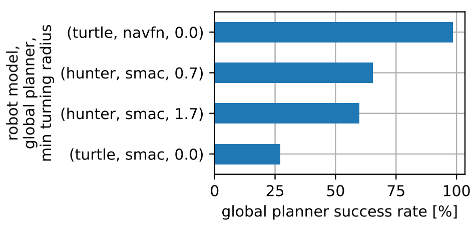
In Fig. 5.11a and Fig. 5.11c, we can see the relative errors are also affected by the odometry error. As shown in Fig. 5.11b, the relative translation error tends to increase when the alpha parameters are over-estimated, which is the opposite behavior when compared to the absolute translation error. The relative rotation error, as shown in Fig. 5.11d, is not affected by the alpha parameters. The relative rotation errors, as opposed to the absolute rotation errors, are affected by which local planner is used. In particular, the TEB local planner causes lower relative rotation errors.

5.2.2 Local Planning Performance Model Results

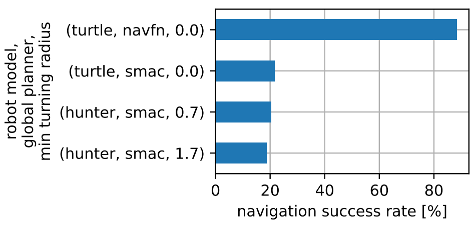
Fig. 5.12 shows the global planner success rate and the navigation success rate, the local planner success rate and collision rate using the two robot platforms, identified by the system feature *robot model*, the Turtlebot robot and the Hunter robot. These platforms have different kinematic models, the turtlebot uses a differential drive kinematic model, while the Hunter robot uses an Ackermann kinematic model. Among the evaluated local planning methods, as implemented in the ROS2 software framework, only the TEB local planner is able to control a robot with Ackermann kinematic constraints. Similarly for the global planning methods, as implemented in the ROS2 software framework, only the SMAC global planner is able to plan a path which enforces Ackermann kinematic constraints. For this reason, while comparing the performance across kinematic models, we only consider the runs with the TEB local planner even for the Turtlebot robot, and later, we will compare the performance of all local planning components using the same kinematic model. The Hunter robot is evaluated for two values of the system feature *min turning radius*, computed from the robot's wheelbase and its maximum steering angle (set by the run parameter *max steering angle*). The global planner success rate and local planner success rate are higher with the Turtlebot robot, as we can see in Fig. 5.12a and Fig. 5.12c respectively, which is to be expected since the additional constraints of the Ackermann kinematic model make the planning and navigation more difficult. The performance of the SMAC global planner results surprisingly low with the Turtlebot robot, although configured with a *min turning radius* of 0 meters, which in principle should constitute an easier problem. This could be due to a problem related to effectively using primitives representing a rotation in place (i.e., having a 0 meters turning radius). The collision rate, shown in Fig. 5.12d, shows that the SMAC global planner allows the TEB local planner to avoid collisions when using the Turtlebot robot, although this is at the cost of lower local planner and global planner success rates, meaning that the probability of obtaining a global plan and being able to reach the goal following it will be inferior with respect to using the Turtlebot robot with the NavFN global planner, or the Hunter robot. Finally, in Fig. 5.12b, we can assess the overall navigation success rate, which considers both the global planner and local planner success rate, which in turn includes the collision rate.

Fig. 5.13, Fig. 5.14 and Fig. 5.15 show the overall local planning and global planning performance for each combination of local planner and global planner, but only using the Turtlebot robot.

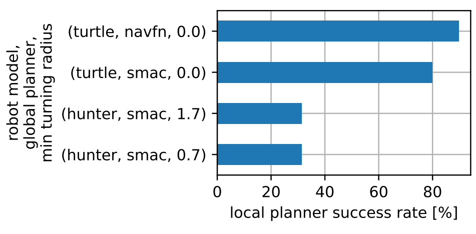
Fig. 5.13 shows the global planners success rate. The SMAC global planner has a substantial failure rate, with only a 28% probability of producing a global plan, while



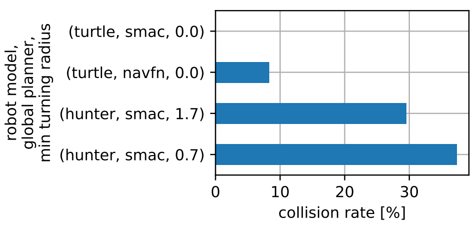
(a)



(b)



(c)



(d)

Figure 5.12: *On top, the overall global planner success rate (higher is better) and navigation success rate from the runs with the TEB local planner, comparing the Turtlebot and Hunter robots. On the bottom, the overall local planner success rate and collision rate (lower is better) from the runs with the TEB local planner in which the global planning was successful, comparing the Turtlebot and Hunter robots.*

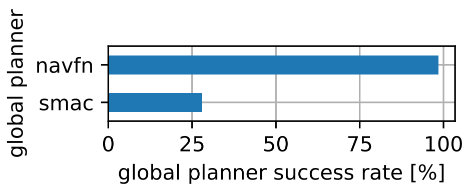
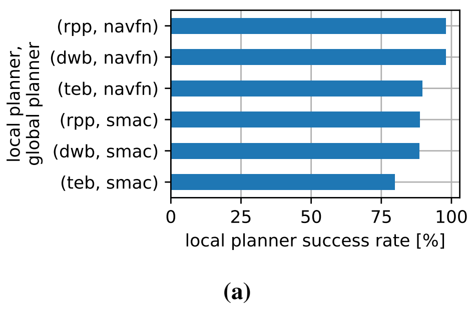


Figure 5.13: *Overall global planner success rate (higher is better) from the runs with the Turtlebot robot.*

the NavFn global planner can produce a global plan with a probability of 98.5%. This means that, in conditions similar to our experiments, the SMAC global planner would not be feasible in a real system if we value its reliability above its efficiency. On the other hand, contrary to the NavFn global planner, the SMAC global planner allows to produce plans with kinematic constraints for an Ackermann robots.

Fig. 5.14a shows the local planner success rate, i.e., the navigation success rate of the runs in which the global planner produced a plan which the local planner could attempt to use. The SMAC global planner reduces the success rate of all local planners. In particular, the TEB local planner achieves a lower success rate compared to the RPP and DWB local planners. The highest performance, 97.1%, is achieved by the RPP and DWB local planners, with the NavFn global planner. The lowest performance, 79.9%, is achieved by the TEB local planner, with the SMAC global planner.

Fig. 5.14b shows the collision rate. The TEB local planner causes the lowest and highest collision rate when paired with the SMAC and NavFn global planners respectively. Meaning that, although the SMAC global planner only produces a global plan in 28% of the runs, when it does, the plan allows TEB to navigate with no collision.



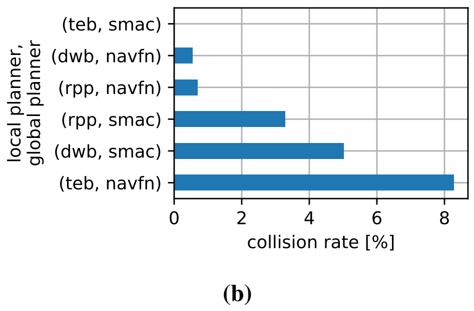


Figure 5.14: *Overall local planner success rate (higher is better) and collision rate (lower is better) from the runs with the Turtlebot robot, in which the global planning was successful.*

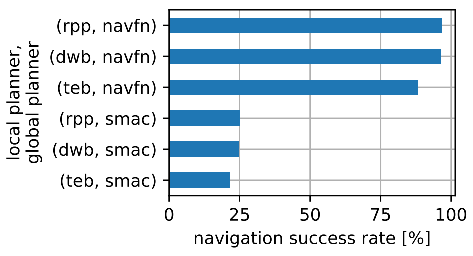


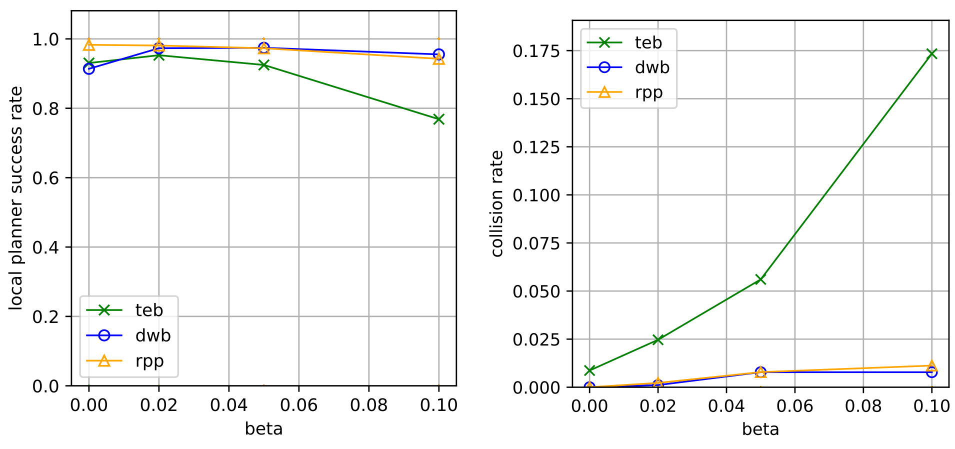
Figure 5.15: *Overall navigation success rate (higher is better) from the runs with the Turtlebot robot.*

Fig. 5.15 shows the overall navigation success rate. This performance metric allows us to choose the best combination of local and global planner, with respect to the probability of successfully reaching the navigation goal without collisions. The best results, 96.6% and 96.5%, are obtained by using the NavFn global planner, and the RPP and DWB local planners respectively. The worst result, 21.7%, is obtained by using the TEB local planner and the SMAC global planner.

Fig. 5.16 shows the local planner success rate and collision rate metrics resulting from different values of odometry error *beta* (along the abscissa of each plot). Only the data obtained using the Turtlebot robot, the NavFn global planner and a successful global planning are considered. These plots, as before, are obtained by computing the mean of the performance metrics for each value of the system feature. The values corresponding to different local planners (i.e., TEB, DWB, RPP) are shown in different colors. The datapoints are not shown in these plots, because the values of success rate and collision rate are either 0 or 1 in each run, so the points would be overlapping.

The success rate, shown in Fig. 5.16a, is only affected by the odometry error when the TEB local planner is used. The RPP and DWB local planners are relatively unaffected by the odometry error. In particular, the success rate of TEB drops from around 0.9 (i.e., 90%) to 0.8 (i.e., 80%), when the odometry error increases from 0 to 0.1. The collision rate, shown in Fig. 5.16b, is also only affected by the odometry error when the TEB local planner is used. TEB's collision rate increases from around 0.01 (i.e., 1%) to 0.175 (i.e., 17.5%), when the odometry error increases from 0 to 0.1.

In Fig. 5.17 and Fig. 5.18 the local planner success rate and collision rate are computed against different values of localization errors (along the abscissa of each plot), and for the three local planners (in different colors). Only the runs obtained using the Turtlebot robot, the NavFn global planner and in which the global planning is success-

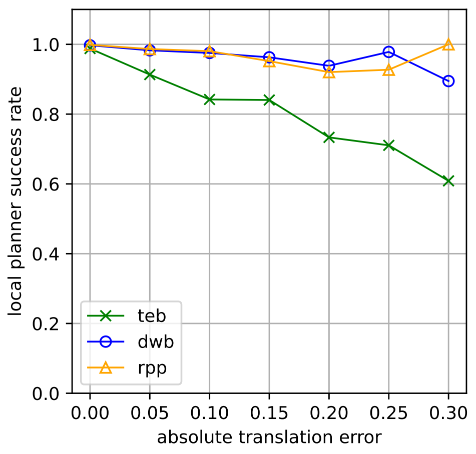


(a) **(b)**

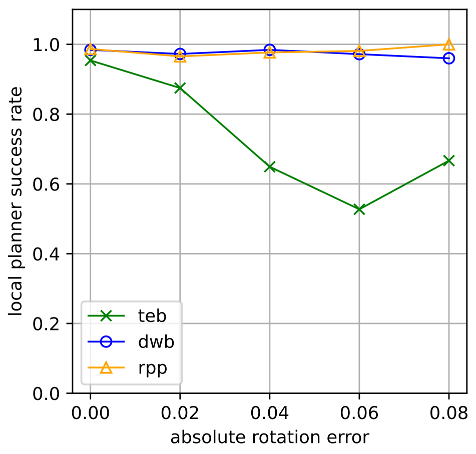
Figure 5.16: *Local planner success rate (higher values are better) and collision rate (lower values are better) against different values of odometry error, beta. Each local planner is plotted in a different color.*

ful are considered. Similarly to the dependency of these metrics on the system feature *beta* (in Fig. 5.16), the performance of the TEB local planner is degraded when the absolute and relative translation errors increase. The dependency on the absolute and relative rotation errors is not as clear since the local planner success rate and collision rate improve up to some localization error value, but then degrade before improving again.

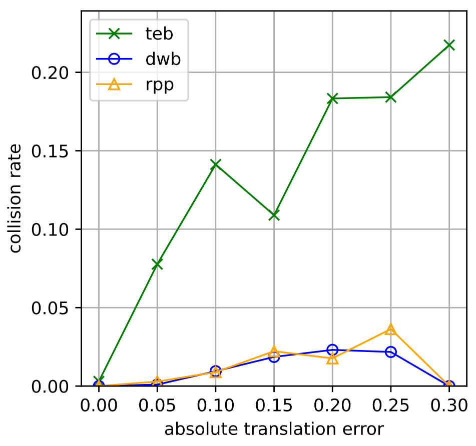
The localization error metrics are themselves strongly correlated with the odometry error *beta*. Fig. 5.19 shows the logistic regression prediction of the TEB local planner success rate from the absolute translation error for each value of *beta*. We can observe that, even within runs with the same odometry error *beta*, the TEB local planner success rate still presents a downward trends with respect to the absolute translation error, meaning that the performance depends from both the odometry error *beta* and the absolute translation error. In Fig. 5.20, we show the Bayes network describing the dependency of the TEB local planner success rate on the feature *beta* and on the absolute translation error, which is itself dependent on the feature *beta*.



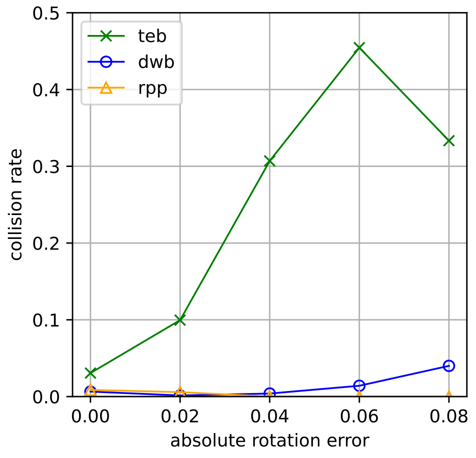
(a)



(b)



(c)



(d)

Figure 5.17: *Local planner success rate (higher values are better) and collision rate (lower values are better) against different values of absolute localization error metrics. Each local planner is plotted in a different color.*

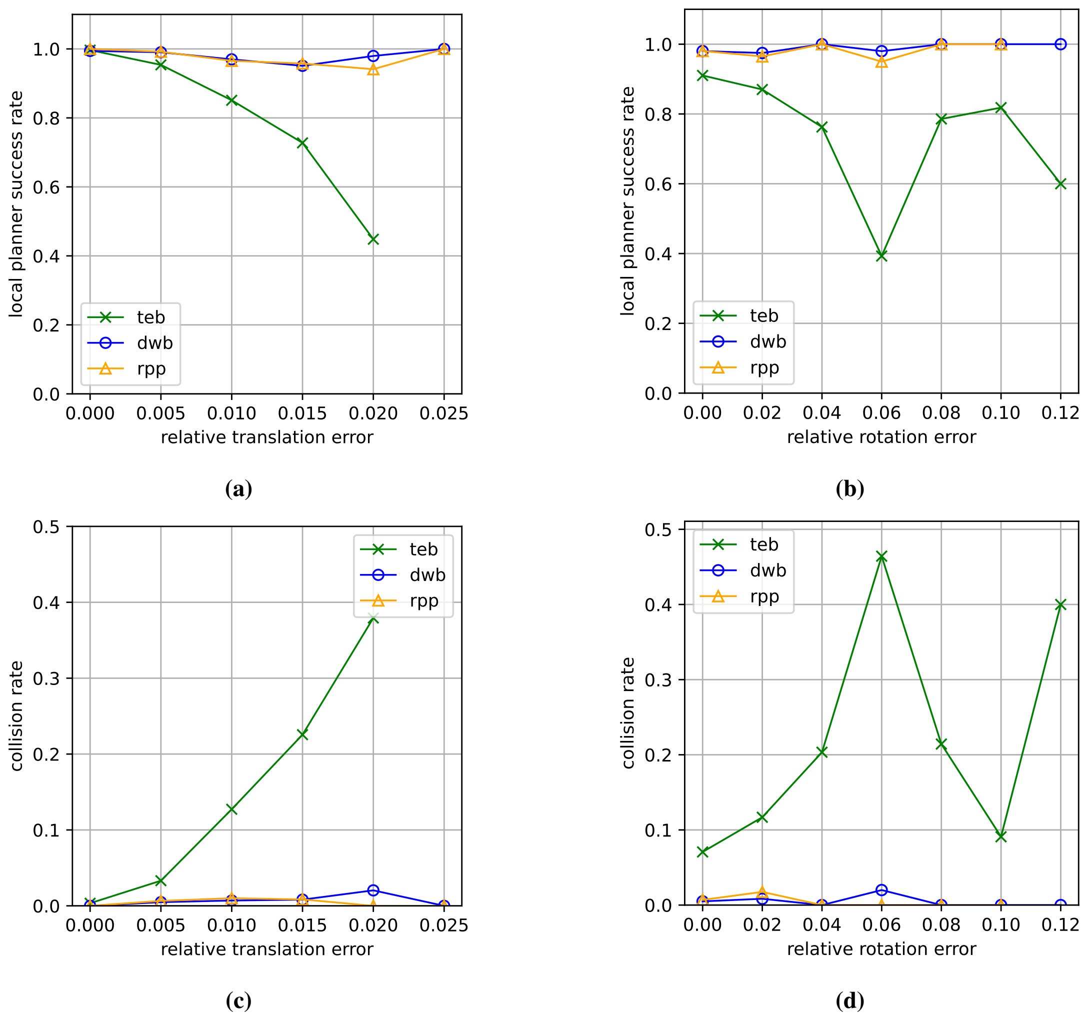


Figure 5.18: *Local planner success rate (higher values are better) and collision rate (lower values are better) against different values of relative localization error metrics. Each local planner is plotted in a different color.*

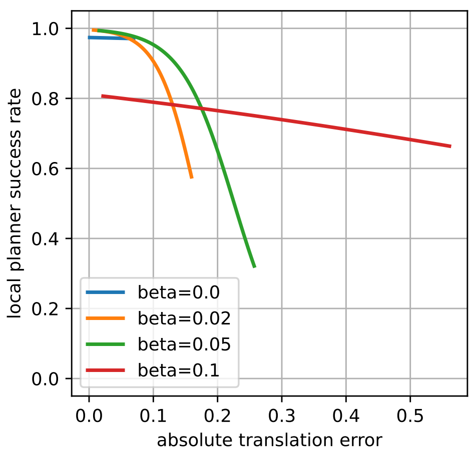


Figure 5.19: *Logistic regression prediction of the TEB local planner success rate from the absolute translation error (along the abscissa), for different values of odometry error beta (in different colors).*

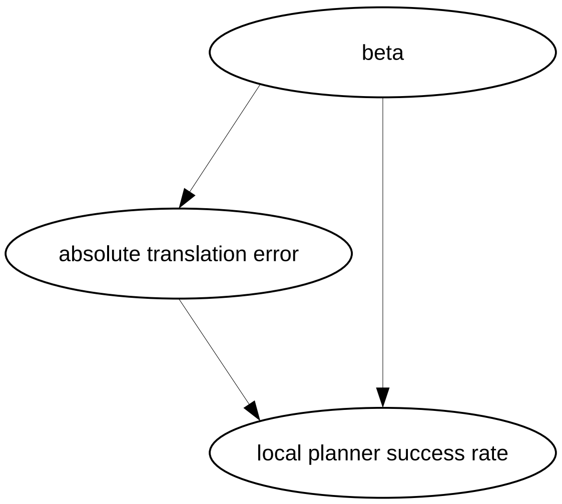


Figure 5.20: *Bayes network in which the performance of the local planner component TEB, local planner success rate, depends from one of the performance metrics of the localization component, absolute translation error, and a system feature, beta.*

CHAPTER 6

Conclusions and Future Work

We presented a general methodology to produce performance models of robotics software components allowing to predict the performance from characteristics of the environment and the system in which they operate. We developed a software framework supporting the proposed methodology by allowing to automatically execute experiments and collect the data. Finally, we presented two case studies of our methodology applied to three SLAM components and a complex system capable of autonomous navigation. The performance models built in the case studies allowed us to inspect the dependency of the performance metrics from each of the analyzed features and to evaluate the dependency of the performance between components.

Future works in the context of the proposed methodology include adding more software components to the performance models of the existing case studies, and producing more performance models for additional functionalities and more complex systems:

- The performance model on autonomous navigation could be expanded by implementing dynamic obstacles and by including additional local planning components, as in the work of [21], which would additionally allow us to replicate and validate their results.
- Performance modelling could be applied to visual SLAM methods. Producing performance models for visual SLAM would be interesting in itself, and it would allow us to compare the performance of visual SLAM components using cameras with the performance of 2D SLAM components using LIDAR sensors.
- Performance modelling could be applied to a robot system for autonomous exploration of an unknown environment, consisting in estimating the map of the environment by using a SLAM component while navigating towards the boundaries of the known environment until the environment is completely explored. In

such a system we can expect the dependency between the SLAM component, navigation components, and the component planning the destination to explore to be inter-dependent, making such system an interesting subject. More complex robot systems could be studied, such as manipulating objects with a robotic arm, for example for grasping, transportation and placement of the objects. This task requires functionalities to localize the objects, positioning the robot in their proximity, building a representation of the environment suitable for motion planning and obstacle avoidance, and executing the arm motion.

A future evolution of the proposed methodology is to study the feasibility of building component performance models obtained by evaluating subsets of the system's components, while still evaluating which components affect each other and model their dependence. This may reduce the number of run parameter combinations needed to obtain a comprehensive evaluation of the entire system and make the problem tractable even for extensive systems. If this was possible, the composition of these performance models would allow us to compose the performance models and predict the performance of an extensive robot system without the need to actually build it and evaluate it in its entirety. This opens the possibility of estimating the performance of an extensive robot system at design-time as envisioned in the Plug and Bench methodology [3].

Real-world experiments can in general provide a better evaluation of the performance, on the other hand, it can be very expensive and time consuming to execute runs with real robots. Therefore, studying how to minimize the number of runs while still obtaining sufficiently representative statistical models is important. Simulation allows us to continuously collect data for very extended periods of time, even experiments which require a real-time simulation can be executed in parallel to speed up the collection of data. Two approaches could be applied to tackle the problem, the first to limit the number of runs, and the second to use both results of simulated and real experiments and gain from the respective benefits.

- The choice of run parameters and their values in our case studies followed an approach similar to brute force: we defined a set of run parameters and for each of them, a list of values, then, we executed a run for each combination of run parameter values (i.e., the Cartesian product of the sets of values of each run parameter). Instead, by automating the choice of run parameter values, it could be possible to minimize the number of runs while obtaining equally or more accurate statistical models. This could be obtained by manually producing a set of coarse statistical models, then applying a method which iteratively refines the statistical models by choosing which run parameter values to use for the next run.

- Simulation could be used to get a relatively approximated performance model which provides the information on how the functionalities/components interact in the system and what range of run parameter values provide interesting results. With this information, a limited number of real-world experiments can be carried out to validate the performance models obtained in simulation, and if a non-negligible difference is found, to calibrate the uncertainty of the simulation results. In this way, it could be possible to exploit the advantages of simulation with the improved confidence of real-world results. For instance, Amigoni *et al.* [2, 29] have validated their extensive simulation results with a small set of real-world experiments.

Bibliography

[1] Hassan Abu Alhaija, Siva Karthik Mustikovela, Lars Mescheder, Andreas Geiger, and Carsten Rother. Augmented Reality Meets Deep Learning for Car Instance Segmentation in Urban Scenes. In *British machine vision conference*, volume 1, page 2, 2017.

[2] Francesco Amigoni, Valerio Castelli, and Matteo Luperto. Improving Repeatability of Experiments by Automatic Evaluation of SLAM Algorithms. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7237–7243. IEEE, 2018.

[3] Gianluca Bardaro, Mohamed El-Shamouly, Giulio Fontana, Ramez Awad, and Matteo Matteucci. Toward Model-Based Benchmarking of Robot Components. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1682–1687. IEEE, 2019.

[4] Meysam Basiri, Enrico Piazza, Matteo Matteucci, and Pedro U. Lima. Benchmarking Functionalities of Domestic Service Robots Through Scientific Competitions. *KI-Künstliche Intelligenz*, 33(4):357–367, 2019.

[5] Fabio Bonsignorio and Angel P Del Pobil. Toward Replicable and Measurable Robotics Research [from the Guest Editors]. *IEEE Robotics & Automation Magazine*, 22(3):32–35, 2015.

[6] Fabio Bonsignorio, John Hallam, and A Del Pobil. Defining the Requisites of a Replicable Robotics Experiment. In *RSS2009 Workshop on Good Experimental Methodologies in Robotics*, 2009.

[7] Fabio Bonsignorio, John Hallam, and Angel Del Pobil. Special Interest Group on Good Experimental Methodology — GEM Guidelines, 2007.

[8] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A Large-Scale Hierarchical Image Database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[9] Dmitri Dolgov, Sebastian Thrun, Michael Montemerlo, and James Diebel. Practical Search Techniques in Path Planning for Autonomous Driving. *Ann Arbor*, 1001(48105):18–80, 2008.

[10] Gabriele Ermacora, Daniele Sartori, Manfredi Rovasenda, Ling Pei, and Wenxian Yu. An Evaluation Framework to Assess Autonomous Navigation Linked to Environment Complexity. In *2020 IEEE International Conference on Mechatronics and Automation (ICMA)*, pages 1803–1810. IEEE, 2020.

[11] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The Pascal Visual Object Classes (VOC) Challenge. *International journal of computer vision*, 88(2):303–338, 2010.

[12] Giulio Fontana and Matteo Matteucci. Methodological Foundations for the Benchmark Metamodel. `https://www.dropbox.com/s/kirlrvb90pxgs8e/foundations_benchmark.pdf?dl=0`, 2018. Accessed: 2019-11-23.

[13] Giulio Fontana, Matteo Matteucci, Gianluca Bardaro, Mohamed El-Shamouty, and Ramez Awad. Benchmark Models - Deliverable D3. `https://www.dropbox.com/sh/4gx7vb0eimg5l7j/AAAM0qVQfyFOVPoycUaODxq1a?dl=0`, 2018. Accessed: 2019-05-03.

[14] Dieter Fox. KLD-Sampling: Adaptive Particle Filters. *Advances in neural information processing systems*, 14, 2001.

[15] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. The Dynamic Window Approach to Collision Avoidance. *IEEE Robotics & Automation Magazine*, 4(1):23–33, 1997.

[16] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision Meets Robotics: The KITTI Dataset. *The International Journal of Robotics Research*, 32(11):1231–1237, 2013.

[17] 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.

[18] Eric Heiden, Luigi Palmieri, Leonard Bruns, Kai O Arras, Gaurav S Sukhatme, and Sven Koenig. Bench-MR: A Motion Planning Benchmark for Wheeled Mobile Robots. *IEEE Robotics and Automation Letters*, 6(3):4536–4543, 2021.

[19] Andrew Howard and Nicholas Roy. The Robotics Data Set Repository (Radish). `http://radish.sourceforge.net/`, 2003.

[20] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, et al. Speed/Accuracy Trade-Offs for Modern Convolutional Object Detectors. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7310–7311, 2017.

[21] Linh Kästner, Teham Buiyan, Lei Jiao, Tuan Anh Le, Xinlin Zhao, Zhengcheng Shen, and Jens Lambrecht. Arena-Rosnav: Towards Deployment of Deep-Reinforcement-Learning-Based Obstacle Avoidance into Conventional Autonomous Navigation Systems. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6456–6463. IEEE, 2021.

[22] S. Kohlbrecher, J. Meyer, O. von Stryk, and U. Klingauf. A Flexible and Scalable SLAM System with Full 3D Motion Estimation. In *Proc. IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*. IEEE, November 2011.

[23] Rainer Kümmerle, Bastian Steder, Christian Dornhege, Michael Ruhnke, Giorgio Grisetti, Cyrill Stachniss, and Alexander Kleiner. On Measuring the Accuracy of SLAM Algorithms. *Autonomous Robots*, 27(4):387, 2009.

[24] Florian Lier, Phillip Lücking, Joshua de Leeuw, Sven Wachsmuth, Selma Šabanović, and Robert Goldstone. Can We Reproduce it? Toward the Implementation of Good Experimental Methodology in Interdisciplinary Robotics Research. In *ICRA 2017 Workshop on Reproducible Research in Robotics: Current Status and Road Ahead*, 2017.

[25] Florian Lier, Johannes Wienke, Arne Nordmann, Sven Wachsmuth, and Sebastian Wrede. The Cognitive Interaction Toolkit–Improving Reproducibility of Robotic Systems Experiments. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 400–411. Springer, 2014.

[26] Pedro U. Lima. A Probabilistic Approach to Benchmarking and Performance Evaluation of Robot Systems. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7231–7236. IEEE, 2018.

[27] Pedro U. Lima, Daniele Nardi, Gerhard K. Kraetzschmar, Rainer Bischoff, and Matteo Matteucci. Rockin and the European Robotics League: Building on Robocup Best Practices to Promote Robot Competitions in Europe. In *Robot World Cup*, pages 181–192. Springer, 2016.

[28] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft COCO: Common Objects in Context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[29] Matteo Luperto, Valerio Castelli, and Francesco Amigoni. Predicting Performance of SLAM Algorithms. https://arxiv.org/abs/2109.02329, 2021.

[30] Steve Macenski and Ivona Jambrecic. SLAM Toolbox: SLAM for the Dynamic World. *Journal of Open Source Software*, 6(61):2783, 2021.

[31] John Ashworth Nelder and Robert WM Wedderburn. Generalized Linear Models. *Journal of the Royal Statistical Society: Series A (General)*, 135(3):370–384, 1972.

[32] Edwin B. Olson. Real-Time Correlative Scan Matching. In *2009 IEEE International Conference on Robotics and Automation*, pages 4387–4393. IEEE, 2009.

[33] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems*, volume 88. Elsevier, 2014.

[34] Enrico Piazza, Pedro U Lima, and Matteo Matteucci. Performance Models in Robotics With a Use Case on SLAM. *IEEE Robotics and Automation Letters*, 7(2):4646–4653, 2022.

[35] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. ROS: an Open-Source Robot Operating System. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[36] Christoph Rösmann, Wendelin Feiten, Thomas Wösch, Frank Hoffmann, and Torsten Bertram. Trajectory Modification Considering Dynamic Constraints of Autonomous Robots. In *ROBOTIK 2012; 7th German Conference on Robotics*, pages 1–6. VDE, 2012.

[37] Christoph Rösmann, Wendelin Feiten, Thomas Wösch, Frank Hoffmann, and Torsten Bertram. Efficient Trajectory Optimization Using a Sparse Model. In *2013 European Conference on Mobile Robots*, pages 138–143. IEEE, 2013.

[38] Christoph Rösmann, Frank Hoffmann, and Torsten Bertram. Planning of Multiple Robot Trajectories in Distinctive Topologies. In *2015 European Conference on Mobile Robots (ECMR)*, pages 1–6. IEEE, 2015.

[39] Christoph Rösmann, Frank Hoffmann, and Torsten Bertram. Integrated Online Trajectory Planning and Optimization in Distinctive Topologies. *Robotics and Autonomous Systems*, 88:142–153, 2017.

[40] Christoph Rösmann, Frank Hoffmann, and Torsten Bertram. Kinodynamic Trajectory Optimization and Control for Car-Like Robots. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5681–5686. IEEE, 2017.

[41] Joao Machado Santos, David Portugal, and Rui P Rocha. An Evaluation of 2D SLAM Techniques Available in Robot Operating System. In *2013 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, pages 1–6. IEEE, 2013.

[42] Sören Schwertfeger and Andreas Birk. Evaluation of Map Quality by Matching and Scoring High-Level, Topological Map Structures. In *2013 IEEE international conference on robotics and automation*, pages 2221–2226. IEEE, 2013.

[43] Christoph Sprunk, Jörg Röwekämper, Gershon Parent, Luciano Spinello, Gian Diego Tipaldi, Wolfram Burgard, and Mihai Jalobeanu. An Experimental Protocol for Benchmarking Robotic Indoor Navigation. In *Experimental Robotics*, pages 487–504. Springer, 2016.

[44] S. Thrun, W. Burgard, D. Fox, and R.C. Arkin. *Probabilistic Robotics*, chapter 5.4, page 132. Intelligent Robotics and Autonomous Agents series. MIT Press, 2005.

[45] Jian Wen, Xuebo Zhang, Qingchen Bi, Zhangchao Pan, Yanghe Feng, Jing Yuan, and Yongchun Fang. MRPB 1.0: A Unified Benchmark for the Evaluation of Mobile Robot Local Planning Approaches. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 8238–8244. IEEE, 2021.

[46] Fei Xia, William B Shen, Chengshu Li, Priya Kasimbeg, Micael Edmond Tchapmi, Alexander Toshev, Roberto Martín-Martín, and Silvio Savarese. Interactive Gibson Benchmark: A benchmark for Interactive Navigation in Cluttered Environments. *IEEE Robotics and Automation Letters*, 5(2):713–720, 2020.