

Universidade Federal de Santa Catarina
Centro de Blumenau
Departamento de Engenharia de
Controle e Automação e Computação



Brunno Vanelli

Comparison and benchmarking for SLAM in mobile robots

Blumenau
2019

Brunno Vanelli

Comparison and benchmarking for SLAM in mobile robots

Final paper submitted in partial fulfillment of the requirements for the degree of BEng. in Automation and Control Engineering of the Universidade Federal de Santa Catarina.

Advisor: Prof. Dr. Marcelo Roberto Petry

Universidade Federal de Santa Catarina
Centro de Blumenau
Departamento de Engenharia de
Controle e Automação e Computação

Blumenau
2019

Brunno Vanelli

Comparison and benchmarking for SLAM in mobile robots

Final paper submitted in partial fulfillment of the requirements for the degree of BEng. in Automation and Control Engineering of the Universidade Federal de Santa Catarina.

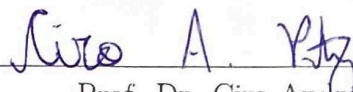
Examination Committee



Prof. Dr. Carlos Roberto Moratelli
Universidade Federal de Santa Catarina



Prof. Dr. Leonardo Mejia Rincon
Universidade Federal de Santa Catarina



Prof. Dr. Ciro Andre Pitz
Universidade Federal de Santa Catarina

Blumenau, June 11, 2019

Don't panic.

Acknowledgements

It is always hard to find words to acknowledge all the people that were and are important during all these years. If I happen to forget anyone, Sir or Madam, truly your forgiveness I implore.

I want to first thank my parents Augustinho and Isabel, and my sister Jaqueline, for supporting me for all these years I've been an unproductive member of society (hopefully that will end soon).

To my close friends for sharing laughs and eventually food during 5 long years. I'll quickly address the five that went through pretty much everything in university with me. Stephan, with an unbeatable amount of side pockets that serve no particular purpose. André, who has a very hard time distinguishing colors. Juliano, who graduated with honors with his astonishing 8,83 score. Guilherme, my distant cousin and finally Rômulo, who probably spend more time in the bus back home than in the university.

To my friend Gabriela for all unlicensed psychological support provided to each other.

To all my professors at UFSC who dedicate their lives to transmit their knowledge forward. A special thanks to professor Marcelo Roberto Petry who oriented me during this dissertation even when far away.

To my supervisors and colleagues at Fraunhofer IPA that during my internship allowed me to learn so much about robotics and got me fascinated on the field.

To all the people I met in the weirdest of circumstances in Stuttgart area and around the world, making my stay a lot better and giving me the opportunity to learn many different cultures. To the survival machine that showed me not all genes are selfish.

And finally, to the open-source community, without whom this work wouldn't be possible. In their free time, they coded the operating system I was working on, the framework that I was working on, the tools and used and even helped me solve the bugs in my code I didn't even know I had.

"Did you ever hear the tragedy of Darth Plagueis the Wise?

I thought not. It's not a story the Jedi would tell you.

It's a Sith legend.

Darth Plagueis was a Dark Lord of the Sith, so powerful and so wise he could use the

Force to influence the midichlorians to create life...

He had such a knowledge of the dark side that he could even keep the ones he cared about

from dying.

The dark side of the Force is a pathway to many abilities some consider to be unnatural.

He became so powerful... the only thing he was afraid of was losing his power, which

eventually, of course, he did.

Unfortunately, he taught his apprentice everything he knew, then his apprentice killed

him in his sleep.

It's ironic he could save others from death, but not himself."

(Sheev Palpatine)

Resumo

A robótica está presente na indústria há décadas, mas a adoção de robôs trabalhando em estreita colaboração com os seres humanos ainda é um desafio. Embora muito tenha sido desenvolvido no campo dos robôs assistivos, eles ainda são incipientes por causa de toda a tecnologia necessária para interagir com os usuários de maneira significativa. Este trabalho de conclusão de curso tem como objetivo discutir uma tarefa específica em robôs móveis chamada SLAM, ou Mapeamento e Localização Simultânea. Essa tarefa compreende a capacidade do robô para mapear ambientes desconhecidos sem informações prévias. Uma estrutura é proposta para analisar metodologicamente o resultado de diferentes algoritmos de SLAM. O estudo de caso será apresentado usando o Care-o-bot, o robô assistivo desenvolvido na Fraunhofer IPA. Dados de sensores de varrimento a laser e odometria são utilizados, e as reconstruções resultantes dos algoritmos mais populares disponíveis no framework ROS (*Robot Operating System*), como Gmapping, Hector, Karto e Cartographer, serão apresentados e comparadas. As métricas de erro quadrático médio e erro de deslocamento serão calculadas para cada algoritmo, bem como os cálculos propostos para distorção do mapa e uso da CPU e da memória. Os resultados mostram boas métricas para Gmapping e Cartographer, escolhas populares na comunidade ROS, com o Cartographer tendo os mapas mais precisos. Hector e Karto são opções alternativas para dispositivos com menor poder de computação, já que podem consumir muito menos CPU nas configurações padrão, além de fornecer boa localização.

Palavras-Chave: 1. SLAM 2. Gmapping 3. Hector 4. Karto 5. Cartographer

Abstract

Robotics has been present in industry for decades now, but the adoption of robots working closely to humans is still challenging. Although much has been developed in the field of assistive robots, they are still incipient because of all the technology required to interact with users in a meaningful way. This paper aims at discussing a specific task in mobile robots, SLAM, or Simultaneous Localization and Mapping. It comprises the ability of the robot to map unknown environments while having no previous information. A framework is proposed to methodologically analyse the mapping results for different SLAM algorithms. The case study will be presented using Care-o-bot, the assistive robot developed at Fraunhofer IPA. Data from laser scanners and odometry is used, and the resulting reconstruction from the most popular algorithms available on ROS (Robot Operating System) will be presented and benchmarked, namely Gmapping, Hector, Karto and Cartographer. Comparisons on mean square error and displacement error will be calculated for each algorithm, as well as proposed calculations for map distortion and CPU and Memory usage. The results show good stats for Gmapping and Cartographer, some of the most popular choices in the ROS community, Cartographer having the most accurate maps. Hector and Karto seem alternative options for devices with lower computing power, as they can consume far lower CPU on default settings, as well as providing good localization.

Keywords: 1. SLAM 2. Gmapping 3. Hector 4. Karto 5. Cartographer

List of figures

Figure 1 – Robots per thousand workers in the industry from 1995 to 2014.	13
Figure 2 – Examples of Assistive and Social robots.	15
Figure 3 – Evolution of Care-o-bot.	16
Figure 4 – Localization stack flowchart.	17
Figure 5 – Topic initialization.	20
Figure 6 – Gazebo simulation suite.	24
Figure 7 – STL Link files.	25
Figure 8 – COB 4 full robot with both manipulators.	28
Figure 9 – LWA4P extended arm.	29
Figure 10 – Dependency graph of <code>cob_simulation</code> with depth 3.	30
Figure 11 – Laser Scanner.	34
Figure 12 – Forward kinematics of mobile robot.	35
Figure 13 – Steps when building an occupancy grid.	37
Figure 14 – Illustration of occupancy grid for a single sensor.	37
Figure 15 – Occupancy grid algorithm for multiple sensors proposed by Elfes.	38
Figure 16 – High-level System overview of Cartographer.	41
Figure 17 – Selected maps for testing.	44
Figure 18 – Representation of metrics taken.	44
Figure 19 – Occupancy grid representation of ground truth and SLAM generated map.	46
Figure 20 – Point cloud representation of ground truth and SLAM generated map.	46
Figure 21 – Alignment of point clouds from Figure 20.	47
Figure 22 – Scripted map generation for Gazebo.	49
Figure 23 – Running the automated parser node with Gmapping.	56
Figure 24 – Results of mapping for first map.	59
Figure 25 – Results of mapping for second map.	60
Figure 26 – Results of mapping for third map.	61
Figure 27 – Incorrect mapping from Gmapping on test 2.	62
Figure 28 – CPU and Memory usage for Gmapping running map test 2.	63
Figure 29 – CPU and Memory usage for Hector running map test 2.	64
Figure 30 – CPU and Memory usage for Karto running map test 2.	64
Figure 31 – CPU and Memory usage for Cartographer running map test 2.	65

List of tables

Table 1 – Base command API.	31
Table 2 – Torso and head command API.	31
Table 3 – Arms and grippers command API.	31
Table 4 – Laser Scan API.	32
Table 5 – Cameras API.	32
Table 6 – Miscellaneous API.	32
Table 7 – Data collected for the first map (lower is better).	59
Table 8 – Data collected for the second map (lower is better).	60
Table 9 – Data collected for the third map (lower is better).	61
Table 10 – Results of running ICP over maps (lower is better).	62
Table 11 – Results of free space mapping accuracy (lower is better).	63

Acronyms

API	<i>Application Program Interface</i>
COB	<i>Care-o-bot</i>
DART	<i>Dynamic Animation and Robotics Toolkit</i>
DOF	<i>Degree of Freedom</i>
ICP	<i>Iterative closest point</i>
IMU	<i>Inertial measurement unit</i>
IPA	<i>Institut für Produktionstechnik und Automatisierung</i>
LIDAR	<i>Light Detection And Ranging</i>
ODE	<i>Open Dynamics Engine</i>
ROS	<i>Robot Operating System</i>
RPC	<i>Remote Procedure Call</i>
SDF	<i>Simulation Description Format</i>
SLAM	<i>Simultaneous localization and mapping</i>
UFSC	<i>Universidade Federal de Santa Catarina</i>
URDF	<i>Unified Robot Description Format</i>
XML	<i>Extensible Markup Language</i>

Table of contents

1	INTRODUCTION	13
1.1	Objectives	18
1.2	Structure	18
2	BACKGROUND THEORY	19
2.1	Robot operating system	19
2.1.1	Packages	20
2.1.2	Topics	20
2.1.3	Services	21
2.1.4	Message types	21
2.2	Transformations	22
2.3	Simulation	23
2.3.1	URDF	24
2.3.2	SDF	25
2.3.3	Physics engines	25
2.3.4	Sensors and actuators	26
2.4	Care-o-bot	27
2.4.1	Base	27
2.4.2	Torso	28
2.4.3	Arms	28
2.4.4	Head	29
2.4.5	Package organization	29
2.4.6	Basic API	30
3	SLAM	33
3.1	Sensors	33
3.2	Localizing the robot	34
3.2.1	Wheel odometry	34
3.2.2	Laser odometry	35
3.3	The localization and mapping problem	36
3.4	ROS SLAM algorithms	38
3.4.1	Gmapping	38
3.4.2	Hector	39
3.4.3	Karto	39
3.4.4	Cartographer	40

3.5	Evaluating SLAM performance	40
3.6	Proposed evaluation techniques	42
3.7	Experimental maps	43
3.8	Pose metrics	44
3.9	Map alignment metric	46
3.10	Free space metric	47
4	EXPERIMENTS AND RESULTS	49
4.1	Building an accurate map	49
4.2	Setting up the SLAM algorithms	50
4.2.1	Gmapping	51
4.2.2	Hector	51
4.2.3	Karto	52
4.2.4	Cartographer	52
4.3	Collecting data	54
4.4	Running the automated reconstruction	55
4.5	Parsing the data	56
4.6	Exporting the map	57
4.7	Results	58
5	CONCLUSIONS	66
5.1	Final considerations	66
5.2	Future work	67
5.3	Contributions	67
	REFERENCES	68

1 Introduction

The robot development in the last century was able to revolutionize the industry by providing flexibility, reliability and quality to the production line, since robots often can perform tasks faster, more accurately and with fewer errors than humans. Human labor that was often repetitive and risky was steadily replaced by robotic labor, with precise movements and increasing automation over time, as shown on Figure 1.

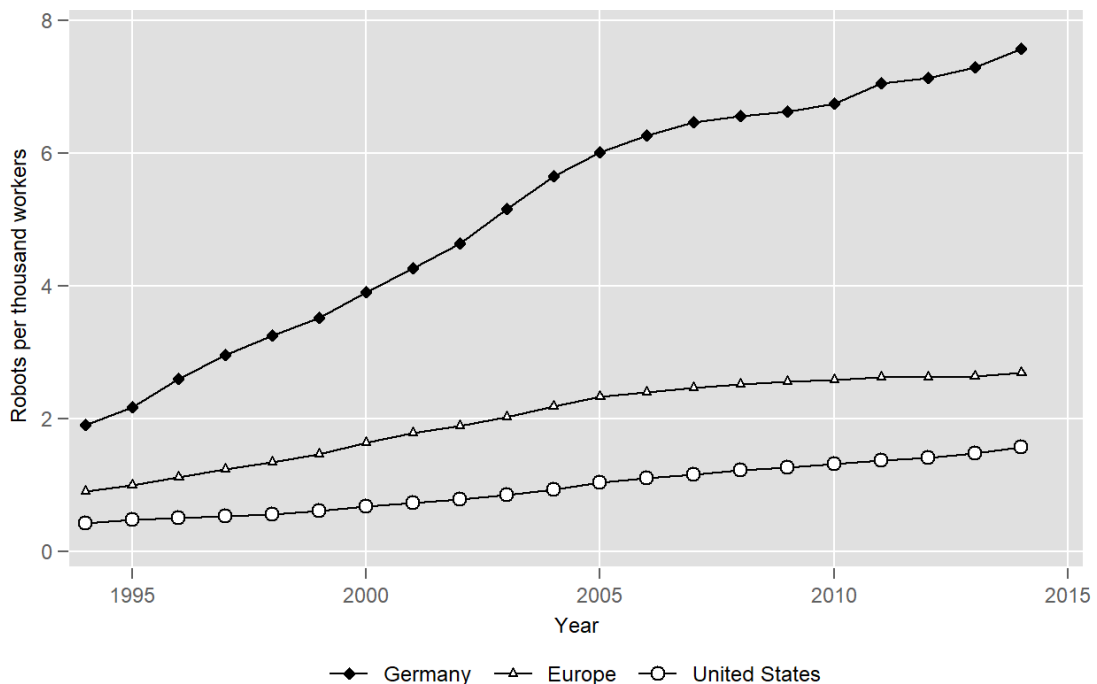


Figure 1 – Robots per thousand workers in the industry from 1995 to 2014 [1].

Even though robots are increasingly present in the industry, using robots to assist persons in everyday life might be more challenging. Many authors propose their use in places like schools, hospitals, and homes, including wheelchair robots, companion robots, manipulator arms for the psychically disabled and elderly populations, etc [2]. However, the dynamic and heterogeneous environment, the energy constraints and the safety are still issues in domestic use robotics.

According to Feil [2], assistive robots are defined as ones that give support to a human user, whereas socially interactive robots are the ones that merely can interact with humans in the form of gestures, speech, etc. The intersection between these two groups is called socially assistive robots, not focusing on the interaction itself but using it as a mechanism to provide aid or assistance.

Fong [3] defines 8 traits in socially interactive robots:

- Embodiment: defined as the body capabilities of the robot, the morphology, and

design to deal with the ambient. Social robots design is often taken into consideration because the user must be comfortable engaging with the robot. Also, the number of sensors and actuators the robot has will expand its capabilities.

- **Emotion:** complements the embodiment in the robots by interacting in a social context. Integrating emotion in robots can create empathy and make people treat them like they treat other humans. Emotions can be displayed both as an expression, in the form of moving lips, eyebrows, eyes, and LEDs, as well as in the form of speech, in voice tone, loudness, and pitch. Body language also takes place in full-body robots.
- **Dialogue:** Creating a meaningful dialogue between two or more parties is hard even in the form of low-level dialogue. Natural language processing are still features under development and remain a great challenge to robots. Robots can also use dialog in a non-verbal way, communicating in gestures and facial display, in addition to displaying emotions.
- **Personality:** in order to correlate with users, robots must develop personality traits that will distinguish them from other robots.
- **Human-oriented perception:** to interact, robots must perceive the environment as humans do. In social situations, this includes people tracking, speech recognition, gesture recognition, and facial perception.
- **User modeling:** in addition to design and perception, they must act based on people personality, learning the user's personality to create a model on how to react.
- **Socially situated learning:** continuously learning for improving communication or acquiring new skills is essential, whether by teaching or imitation.
- **Intentionality:** lastly, humans must feel that the robot has a purpose and acts rationally. This can be achieved by demonstrating goal-directed behaviors or demonstrating attention to key objects in the scenario.

Feil goes further and defines the socially assistive robot with additional properties relative to Fong's definition:

- **User Populations:** defines the characteristics of the user, like age, impairment, and need. He categorizes the user populations as elderly, individuals with physical impairments, individuals in convalescent care, individuals with cognitive disorders and students.
- **Task:** the author cites as task examples tutoring, physical therapy, daily life assistance, and emotional expression.

- Sophistication of interaction: the robot must develop beyond the emotional feature described by Fong, evolving into more complex reciprocal interactions with the user.
- Role of the assistive robot: the role of the assistive robot must reflect into its appearance and behavior, affecting both embodiment and personality, depending on the task and nature of the interaction.

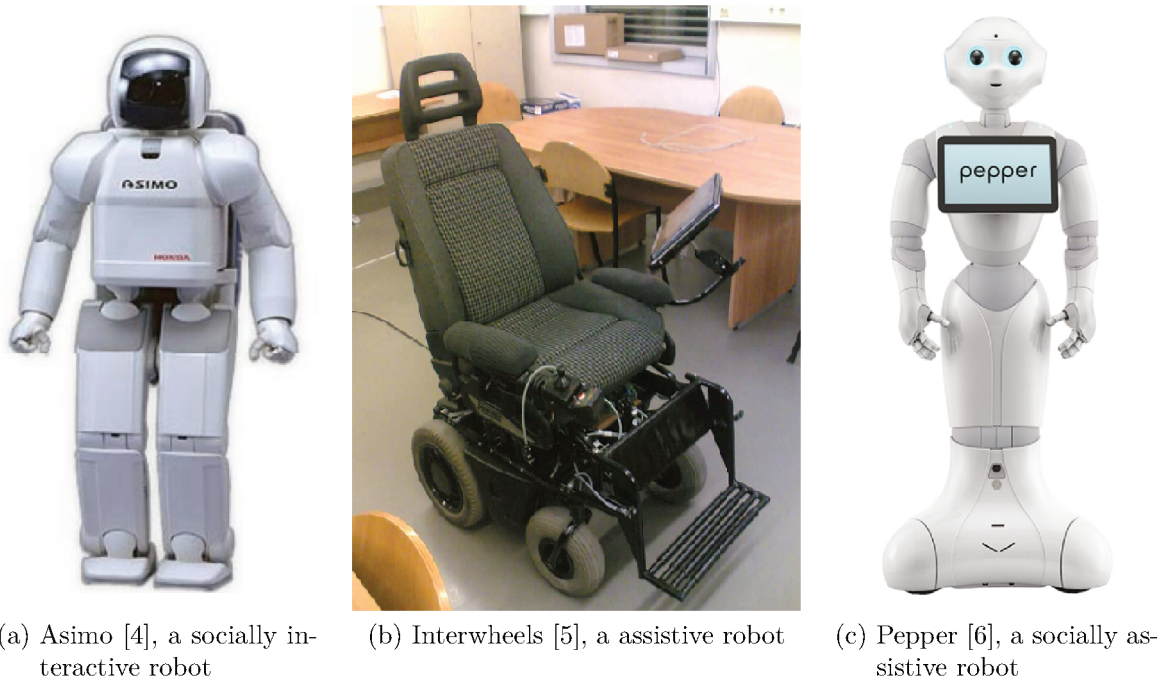


Figure 2 – Examples of Assistive and Social robots.

One of the most famous socially interactive robots developed is ASIMO, shown on Figure 2a, a human-like robot developed by the Japanese company Honda in 2000. It featured a pair of legs and many features and sensors that would later be integrated into other assistive robots. Its main task was to meet visitors in the meeting room and to guide them once the manager confirms their identity. It was able to perform navigation, obstacle avoidance and respond to voice and gesture calls [4]. Even though the robot can interact with people, it does not qualify as a socially assistive robot (the first version, at the time) as defined by Feil because it couldn't fit in the user populations described. The robot was later improved to have greater grasping capabilities and a more robust sensor set.

Intelwheels is an example of an assistive robot, as seen in Figure 2b. It can assist handicapped people operating an electric wheelchair, providing not only user operation with collision avoidance but autonomous navigation. It can help elders and people with physical impairments, but the user input is one-way: it can recognize voice and facial expressions but won't interact back with the user [5].

Pepper, seen on on Figure 2c, was developed to fit into the category of both emotional expression and tutoring. It was aimed at the concept of the robot learning together with children using the robot’s display to teach English, characterizing it as a socially assistive robot. A remote human teacher would help the process by orienting the classes. Instead of just showing the contents in the screen, creating boredom, the robot engaged in playful activities with the kids, including telling them to search for a specific object in the room or repeating gestures with the robot, compelling the children to participate actively [6].

The Care-o-bot, or COB, for short, is described as a robotic home assistant aimed at helping people with mobility impairments in their daily lives. The target group includes elders, disabled, people with health conditions and with movement restraints. Its tasks include setting the table, carrying objects like books and drinks around, dealing with medication, helping the patients standing up, as well as serving as a companion to the person. The robot can also do other tasks usually performed by a nurses and doctors, like monitoring the patient with conditions in their daily routines, reminding them to take medication and calling emergency in case of incidents [7].



Figure 3 – Evolution of Care-o-bot.

The Care-o-bot evolution throughout the years can be seen on Figure 3. The first prototype was built in 1997, but it didn’t pack many capabilities as the technology was very limited. It was soon followed by Care-o-bot II, in 2002, equipped with a manipulator, two cameras, laser scanners, and a hand-held control panel. Version II presented great improvements in navigation, computer vision, and manipulation, but was still rudimentary, having problems in low light conditions and dynamic environments (the 3D scans could only be run once because of processing constraints) and was not recommended for inexperienced users [7].

The third generation came in 2008, using a better 7DOF (Degrees of Freedom) arm and 3 finger gripper with tactile feedback. It also became a lot more user-friendly, applying the concepts of embodiment and presenting a less bulky body with smoother surfaces and

less visible mechanical parts. The robot body was divided into a working side (in the back, where the manipulator stands) and a serving side (on the front, to interact with users) [8].

The fourth generation was presented in 2015 and focused heavily on emotion design. It was developed to appear familiar and sociable, and avoid the uncanny valley [9], where human-like robots can cause strangeness or even fear. It featured a multi-modal user interface capable of displaying facial expressions with a minimalistic pair of eyes to display a wide range of emotions. The spherical joints in the torso and head allow more agility, the body is smaller and more efficient and more sensors and 3D cameras were installed.

One of the many challenges of assistive robots is navigating the environment. As they need to reach places in the house like the kitchen or the bedroom, they sometimes need to be aware of how to navigate through the rooms and corridors to reach the final destination. The approach described in Siegwart, Nourbakhsh and Scaramuzza[10] for this problem is running algorithms in different layers: perception, to extract meaningful data from the environment using available sensors; localization, to determine where it is relative to the environment; cognition, in order to take action given the inputs; and motion control, in order to output the right commands to the motors [10]. Figure 4 shows us the guidelines to build a localization stack in a mobile robot.

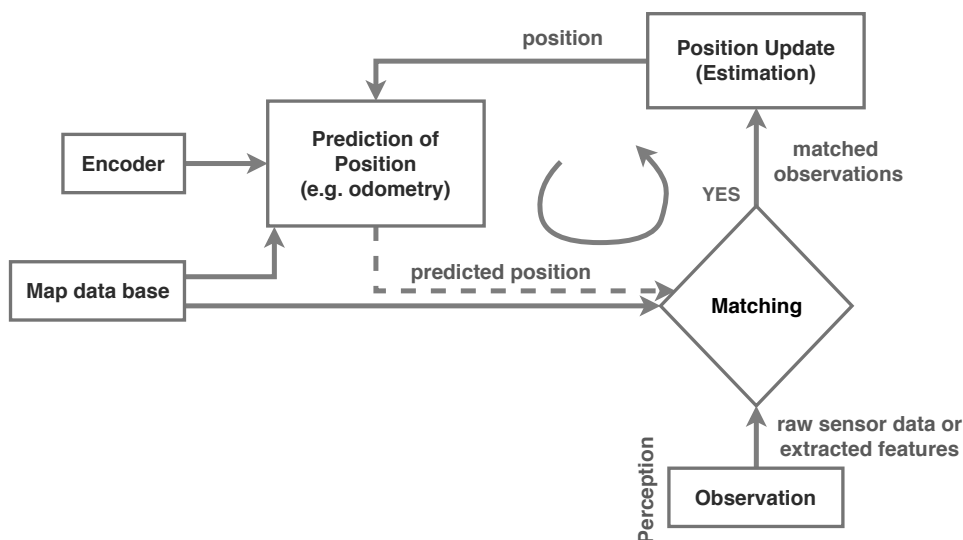


Figure 4 – Localization stack flowchart [10].

As we can see, it is necessary to sense the state of the robot and the world. This is done through encoders in the wheels, GPS, or other forms of localization, in the form of odometry, or in observations, using 3D cameras, LIDAR (Light Detection And Ranging), Sonar (Sound Navigation and Ranging), etc. This data is fed to a matching algorithm, together with a map database, resulting in an estimated position update, that will be used as feedback for the next iteration.

However, this approach is only as accurate as the map is. Many algorithms aim to solve

the mapping problem accurately, but there is no standard way of telling how accurate each algorithm is because there is no agreement on which benchmarks to use. The difficulties in comparing visual data and the absence of the ground truth impose challenges when comparing maps generated by different algorithms [11].

1.1 Objectives

The goal of this paper is to propose a framework to methodically compare and evaluate accuracy of mapping algorithms. We will test the proposed framework against the most popular algorithms available in the open repositories, namely Gmapping, Hector, Karto and Cartographer, using data from simulated Care-o-bot. The results will be then discussed and compared against what is currently present in the literature about mapping benchmarking.

The following tasks are specific objectives:

- Revision of literature for SLAM comparison metrics.
- Implement an algorithm for the generation of maps based on the ground-truth representation.
- Simulation of proposed SLAM algorithms in the Care-o-bot robot.
- Comparison of results using proposed metrics.
- Publish all test data and algorithms in an open-source repository.

1.2 Structure

This work is structured as follows. Chapter 2 shows the background theory in order to understand how the tests were performed. It includes an introduction to the Robot Operating System (ROS) used in all experiments, as well as an introduction to COB hardware. Chapter 3 explains the mapping problem into more detail and gives an introduction to how each tested algorithm works. This chapter also presents a review of the literature to come up with good metrics and an algorithm approach for comparison. A custom way of generating accurate maps is proposed and simulated using Care-o-bot. The resulting laser scan and odometry data is exported into public datasets, so that they can be reused for further testing or even for different algorithms not tested in this dissertation. Chapter 4 shows how the experiments are set, the results obtained using the proposed metrics, and a brief discussion. Finally, Chapter 5 presents conclusions and discusses future work in this topic.

2 Background Theory

2.1 Robot operating system

There are many problems when developing robot applications, especially because of the complexity of those systems. As more and more functionality is added to the robot, the code base becomes cluttered with intricate dependencies and entangled libraries. ROS (Robot Operating System) is not an operating system per se, but a framework that allows coders to readily develop and test solutions with modularity and code reusability in mind. It was built in an agnostic package system that allows integration with many packages available from the ROS Open-source community, a lot of them implementing support libraries and proof-of-concept algorithms, as well as core infrastructure.

The main aspects of ROS are [12]:

- Peer-to-peer: even though the ROS framework relies on a master or namespace as a lookup mechanism, the communication is established between peers, avoiding unnecessary routing through slow links when the recipient is on the same subnet.
- Tools-based: instead of building an intricate framework, ROS relies on a set of tools written to perform specific tasks, including various tools for compilation, tap data stream, data plotting, configuration, documentation generation, etc. Custom tools can even be written by the user in the form of new packages.
- Multi-lingual: since communication between nodes relies only on XML-RPC, they can be implemented in any language, either by explicitly writing the full library that interacts with the ROS Core or building a wrapper for the ROS C++ library.
- Thin: many robots implementations have parts of the code that could be reused in another project if they weren't so entangled with all existing code. ROS proposes an architecture where the code is separated into packages that hold no dependency on ROS. All packages can be built individually using CMake, different from the traditional software paradigm where one CMake file builds the entire project.
- Free and Open-source: ROS source code is publicly available and released under the BSD License, allowing you to copy, modify and redistribute the source code, including for commercial purposes.

The execution of ROS is separated into smaller pieces of code that do particular tasks, called nodes. Each time ROS Core is launched, a collection nodes that are required for execution are spawned: the ROS master, that provides support for the registration of

all subsequent nodes, the ROS parameter server, to register parameters during execution time, and the `roscout` node, for logging purposes. Once the core is launched, every other node can be spawned.

2.1.1 Packages

In order to better build robotic systems, ROS adopts a packaged architecture, making every subsystem of the application separate from all the others. Every package can contain new nodes, libraries, configuration or even datasets.

The main advantage of this approach is making the code more organized in its own subsystems, that specialize in doing a specific task well so that other packages can use this functionality. Packages can also be written independently of language, as long as it's supported by ROS.

2.1.2 Topics

In order for nodes to communicate, they interact by publishing messages in topics. Topics are anonymous buses where each node can publish messages following the message type standards. Each node can then subscribe to topics that are relevant to them and act upon data captured on the topic. Message can even be recorded and played back to support applications that will need the information later in time.

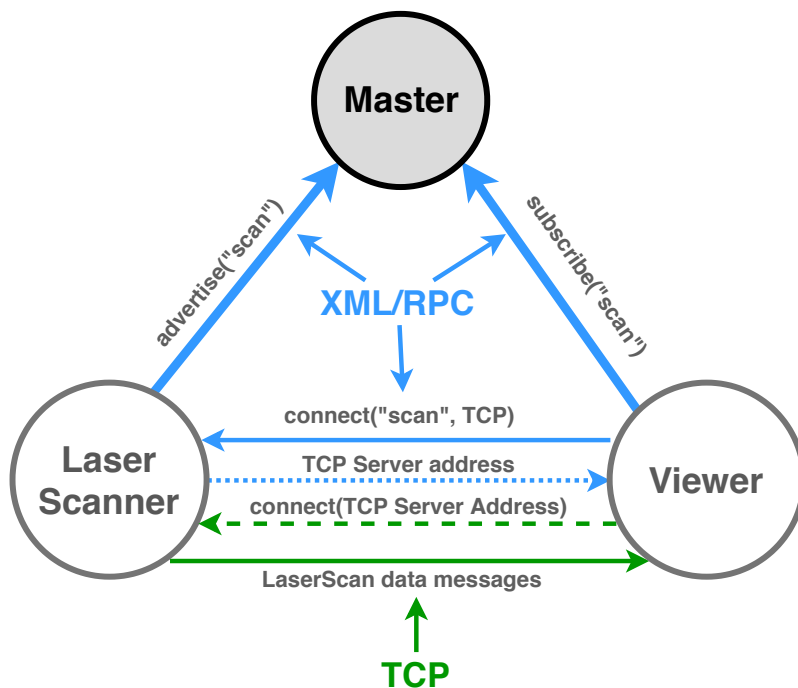


Figure 5 – Topic initialization [13].

The topics are implemented in the XML-RPC protocol, using a master node in order to provide name resolution. In order for a publisher to connect to a subscriber, they do

the following steps [13]:

1. Subscriber register with the master the topics it will be listening to.
2. Publisher register with the master the topics it will be publishing to.
3. Master informs the subscriber of a new publisher.
4. Subscriber requests a topic connection with the Publisher and negotiates a transport protocol.
5. Subscriber connects using the selected protocol.

The process described above is illustrated in Figure 5. It's important to notice that, once the connection is established, the communication is maintained peer-to-peer, using former protocols like TCPROS, built over TCP, and UDPROS, built over UDP. This not only provides faster communication inside the same network by not requiring the messages to be relayed through the master but also enables communication through available networks of the Internet protocol suite, including 802.11X wireless transmission.

The topic configuration also enables true agnostic packages, since the communication between them will be done using a standardized communication medium, loosely coupling the packages and making them easily maintainable. Debugging can also be done using command-line tools that wiretap this medium and display the information exchanged by nodes.

2.1.3 Services

The topic communication can be very useful in many-to-many communication but lacks support when sending messages or commands that require a response. When a reply is needed, it is a better practice to use services instead of topics. This is especially true for tasks that need a lot of computing power but only need to be executed once in a while, so instead of calculating it in every iteration, the service can be run just when requested and return data to the caller. The request is usually done in a similar way to Remote Procedure Calls (RPCs) in programming languages.

2.1.4 Message types

Since nodes need to understand each other, they talk following pre-defined message standards, and the message files themselves are packages that define the content of each message. To avoid confusion, each topic is initialized using a pre-defined message type and every node has to conform to it. Let's take a look at the message definition `geometry_msgs/Twist.msg`, that is used in navigation to define linear and angular velocity for the joints:

```
1 Vector3  linear
2 Vector3  angular
```

Note that the `Vector3` is not a base type message, but is another type of message defined at `geometry_msgs/Vector3.msg`:

```
1 float64 x
2 float64 y
3 float64 z
```

The variable types that cannot be expanded into other definitions are called primitive types. The primitive types are:

- `bool`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`
- `int64`
- `uint64`
- `float32`
- `float64`
- `string`
- `time`
- `duration`

Message types for services are done in a similar way, but since services support response, the message will consist of two individual messages. If we look at a standard definition found in `std_srvs/SetBool.srv`:

```
1 bool data
2 ——
3 bool success
4 string message
```

This service is used for setting a boolean variable to *true* or *false*, that can be activating or deactivating an actuator. The response consists of a boolean that tells if the operation was successful and a message for better description in case of error.

2.2 Transformations

Transformations, or tfs, for short, are the way ROS deals with coordinates frames in space. As the poses for each link in the robot might change over time, it keeps track of these changes and provides tools to assist the user to do transformations with the data.

Suppose a particular robot has a reference fixed frame `/world` in the origin. The center of the robot is another frame called `base_link`. The transformation `/world -> /base_link` would indicate the position of the robot relative to the world.

Now, let's say the robot has a laser scanner sensor `/laser_link` relative to `/base_link`, as it is fixed on the robot. The laser makes a measurement and this result is relative to

the laser position. Let's call the result `/result`. The result is then related to `/world` by a long chain of transformations.

```
/world -> /base_link -> /laser_link -> /result
```

It can be tricky to transform the `/result` back to `/world` coordinates. The `tf` package gives the user an easy way of setting up a listener on the `/tf` topic that will listen to the published transforms and do the required transformations between coordinate frames, so the user can ask directly for the transform `/world -> result` instead of transforming the data himself.

2.3 Simulation

Since deploying a test robot at every code change is costly and time consuming and ROS only provides the tools to develop a robot system solution, there is a need for offline simulation of the robots, avoiding having to test every configuration on physical hardware. In these cases, it can be helpful to set up a rigid body simulation. The normal workflow for setting up a simulation is [14]:

- Isolating the important variables in the physical process.
- Model the physical system behavior using equations.
- Find a method to solve the equations, given the inputs and the initial state of the system.
- Write a computer program that can do that simulation.
- Simulate and benchmark the results against the physical system.

Repeating all these five steps for every physical system will lead to the best results, but can be time-consuming and difficult because all the variables involved. Since the possibilities for a robot system are often limited, physics engines or physics SDKs (source development kits) were developed to aid simulation.

While the real world has a lot of complexity, rigid body dynamics can be simplified in rigid bodies, joints, collisions, friction and springs, elements that are important to the process. The physical model can be reduced to the laws of motion, using mass, velocity, acceleration, and force as simulation variables. Since the model and the solution to these problems are well known, generalized solvers can be written that compute the simulation at each instant of time, or time steps.

Many simulators implement this generalized solvers, as well as render the 3D models to show to the user, including game engines like Unity and USARSim (based on Unreal

engine), commercial solutions like Microsoft Robot Studio, Webots and MATLAB, and open-source projects like Gazebo [15]. Since Gazebo adopts the mentality of the ROS project of being open and free, it became widely used in the community, and as a result better developed over time.

Gazebo provides ROS with a framework to simulate and benchmark the robot or even a group of robots accurately. Figure 6 shows the Gazebo simulation for a supported robot [16].

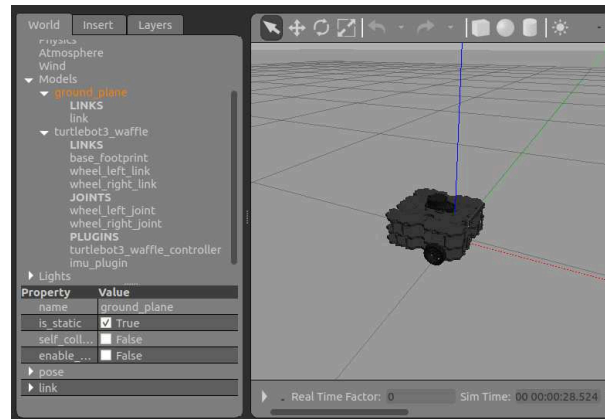


Figure 6 – Gazebo simulation suite.

The Gazebo simulator offers:

- Dynamics simulation using multiple physics engines.
- Advanced 3D graphics for high-quality rendering.
- Sensors and noise, to generate reliable sensor data, compatible with real-world sensors.
- Robot models, including ones from the community.

All the robot description is made using a URDF (Universal Robotic Description Format) file or an SDF (Simulation Description Format) file. The URDF file is an XML file describing all elements of the robot. Even though it's called "Universal", it lacks some of the features like parallel linkages, friction, etc. To get around these issues, a new model called SDF was developed specifically for use in Gazebo, while the URDF was maintained for backward compatibility. Every time a URDF file is loaded, it is converted by Gazebo to an SDF equivalent.

2.3.1 URDF

URDF files starts describing each link of the robot and it's respective inertia. Figure 6 show an example of the robot Turtlebot3 Waffle [17] consisting of three links:

`base_footprint`, `wheel_left_link` and `wheel_right_link`. They are loaded from STL or Collada files included in the project. Figure 7 show the STL renders for the robot shown on Figure 6.

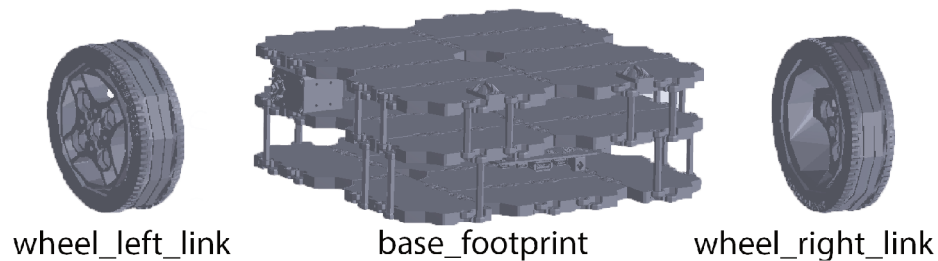


Figure 7 – STL Link files.

The inertia is described by its inertial parameters, namely the mass, center of mass and moment of inertia matrix. Additionally, collision box that will aid collision testing in simulation can be specified. The user can then describe the rest of the robot, including the joints that will hold links together. Two basic joints can be seen, `wheel_left_joint` and `wheel_right_joint`, that hold all three links together. Finally, the plugins will give the robot the simulation functionality, adding IMUs, Laser Scanners, Cameras, etc.

2.3.2 SDF

The SDF is an improvement of the URDF descriptor format that supports more functionality. It not only supports all the physical descriptions shown in the URDF model but also features:

- World information, from lightning and gravity to complex information like magnetic fields, winds, and atmosphere (temperature and pressure).
- Scene information, like ambient, background, sky, fog and shadows.
- Multiple robots descriptors.
- Choosing a physics engine.

2.3.3 Physics engines

In order to simulate the physical conditions of the robot and the environment, Gazebo supports four different physics engines: ODE, Bullet, Simbody, and DART. At the start of each simulation, the user can select the desired physics engine changing the startup flags for the program, or configure it on the SDF file.

ODE or Open Dynamics Engine is an engine for simulating articulated rigid body structures. It features a stable integrator, meaning that numeric errors may not grow

out of control. Because of that, the simulator drops physical accuracy in favor of speed, stability and robustness [18].

Bullet is a Python implementation of physics simulation for robotics, games, and visual effects that provides forward and inverse dynamics and kinematics, as well as built-in collision detection. Bullet differentiates itself by being easy to use and provides integration to machine learning frameworks like TensorFlow [19].

Simbody is a multibody simulator focused on biomedical research. It was developed to better suit simulation scenarios where engines like ODE may not converge to correct results due to its lack of fidelity. It is used for neuromuscular, prosthetic, and biomolecular simulation, as well as design and control of humanoid robots [20].

DART or Dynamic Animation and Robotics Toolkit is another rigid body simulation that distinguishes itself due to its accuracy and stability. The main purpose of this simulator is to provide full access to internal kinematic and dynamic quantities [21].

All four engines are used to tackle different problems. By varying the time step, for instance, you might obtain better results in your simulation with one of the engines. Usually, when choosing the simulation engine, the relationship between the number of iterations, error, and speed must be taken into account. Some of the simulations require greater precision, while others may require real-time performance. For a more in-depth comparison of the four engines see [22].

2.3.4 Sensors and actuators

Gazebo also allows simulating sensors and actuators that will interact with the world and send data back to ROS packages. Sensors can include cameras, laser scanners, contact sensors, IMU, RFID, Sonar, Magnetometer. Actuators can use the ROS control library to actuate joints. The sensors are described in the SDF or URDF files. The example below shows an example of a camera description.

```
1 <gazebo reference="camera_link">
2   <sensor type="camera" name="cameral">
3     <update_rate>30.0</update_rate>
4     <camera name="head">
5       <horizontal_fov>1.3962634</horizontal_fov>
6       <image>
7         <width>800</width>
8         <height>800</height>
9         <format>R8G8B8</format>
10      </image>
11      ...
12    </camera>
13    <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
14      <alwaysOn>true</alwaysOn>
15      <updateRate>0.0</updateRate>
```

```
16     <cameraName>robot/cam1</cameraName>
17     <imageTopicName>image_raw</imageTopicName>
18     <cameraInfoTopicName>camera_info</cameraInfoTopicName>
19     <frameName>camera_link</frameName>
20     <hackBaseline>0.07</hackBaseline>
21     <distortionK1>0.0</distortionK1>
22     <distortionK2>0.0</distortionK2>
23     <distortionK3>0.0</distortionK3>
24     <distortionT1>0.0</distortionT1>
25     <distortionT2>0.0</distortionT2>
26   </plugin>
27 </sensor>
28 </gazebo>
```

Listing 2.1 – Exemple of a URDF file for a camera description.

The controller is loaded from a plugin called `libgazebo_ros_camera.so` and uses as parameters which topic the camera will correspond to and the intrinsic parameters of the lenses, update rate and sensor type, as well as the link the camera will be attached to.

There are many pre-built plugins available at Gazebo library, including Cameras (mono and stereo), Kinect, Laser Scanner, Force and IMU sensors, Differential Drive, Skid Steering Drive, as well as templates to write your own dedicated plugin.

2.4 Care-o-bot

Care-o-bot is a project for a mobile assistive robot that is modular, developed and maintained by Fraunhofer IPA. COB's fourth version, Figure 8, was designed not only to provide researchers a reliable mobile base, but also to aid research on human-robot interaction and social behavior [23]. It is composed mainly by a mobile base, a torso, and a head.

2.4.1 Base

The base features three steerable wheels used for moving the robot on the ground. Because the modularity, the wheels can be configured to use Ackermann kinematics, moving forward and backward and rotating on the vertical axis, but also Omnidirectional kinematics, allowing the robot to move in every direction. The maximum speed supported is 1,1 m/s. It is also equipped with three 2D laser scanners with 360° coverage for object detection and safety, and the battery pack to power the robot and the control panel.

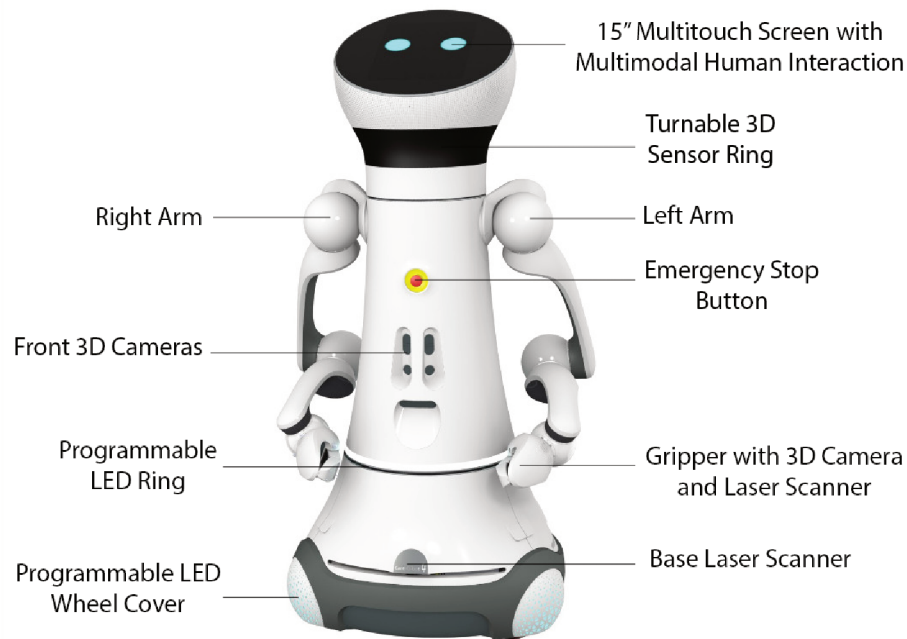


Figure 8 – COB 4 full robot with both manipulators [24].

2.4.2 Torso

The torso is linked with the base, and can be configured to be fixed (the torso doesn't move), to use a Pan joint (allowing for a full rotation) or using a spherical joint (providing 3DOF). It can have two optional arms with 7DOF, each with a robotic hand with two fingers (2 DOF hand). The torso also features 3D cameras on the front and in the back.

2.4.3 Arms

Arms are composed of three SCHUNK Powerball ERB modules and one PRL+ 100 module, a rotary actuator, in the shoulder. The arm is a variation of the commercially available Schunk arm called LWA4P, which has 6 DOF and is composed of three Powerball ERB, with 2 DOF each. The last degree of freedom is provided by the PRL+, allowing one rotation around the shoulder and totaling 7 DOF. Figure 9 shows the full arm in simulation. The manipulator's finger was developed by Schunk specifically for this robot.

The manipulator also has a 3D camera and laser pointer for object recognition and picking, as well as LEDs for illumination. The torso also provides 3D cameras and sensors for computer vision activities that cover up the front of the robot and an LED Ring for signaling and an emergency stop.

The LWA4P arm was designed to operate at low power at battery-dependent devices and is capable of lifting a maximum payload of 6kg. Each joint in the Powerball is capable of rotating to $\pm 170^\circ$, at a maximum speed of $72^\circ/\text{s}$, and the overall repeat accuracy of the arm is ± 0.15 mm.



Figure 9 – LWA4P extended arm.

2.4.4 Head

The head is linked with the torso, allowing for both a pan joint or a spherical joint, and contain the human interface to interact with the user, including the sound system, microphone, a touch screen display and optional camera for face recognition.

2.4.5 Package organization

COB is built around ROS and combines different sets of packages for different purposes. Since the robot support different configurations (manipulators, joints, mobile bases), some packages are optional. Figure 10 shows the dependency graph for the first three layers of packages. Notice that the arrangement is quite intricate even showing only the packages written exclusively for COB (not showing other ROS packages used on COB).

The COB core consist of the following packages and its dependencies:

- `cob_msgs`: Robot-specific Messages, representing state information like battery status, etc.
- `cob_srvs`: Robot-specific Services.
- `cob_description`: Robot URDF models for different COB configurations (only base, base with fixed torso, base with actuated torso, etc).
- `cob_bringup`: machine configuration, including all scripts and dependencies required to run COB, both in simulation and real hardware.

COB also features high-level capabilities, some of them being:

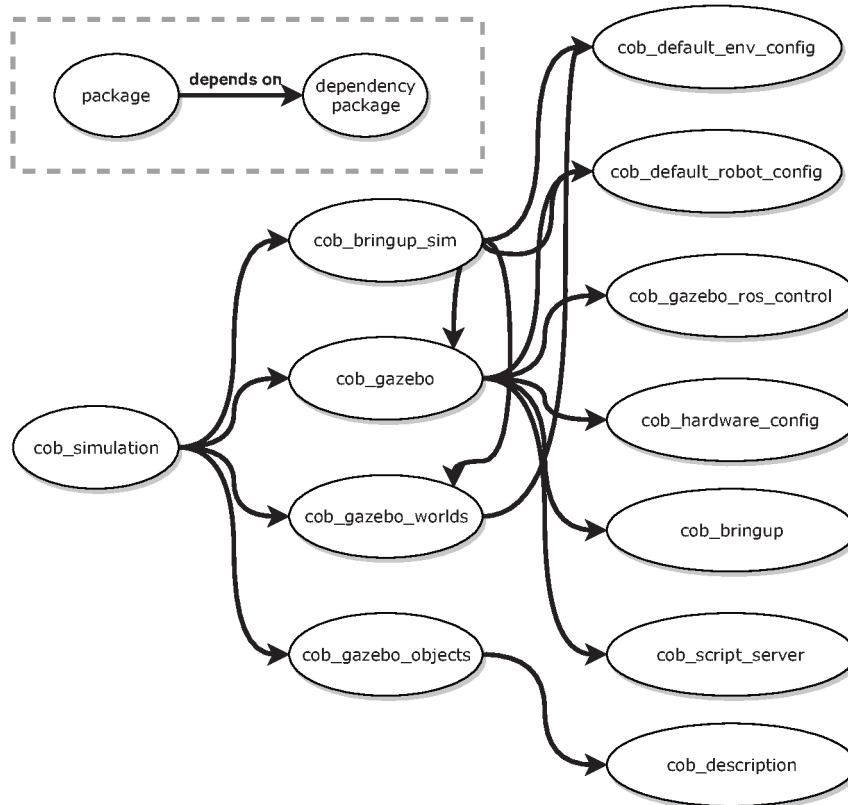


Figure 10 – Dependency graph of `cob_simulation` with depth 3.

- `cob_command_tools`: high-level utilities command tools, including API for commonly used movements, control dashboard, teleoperation and status monitoring.
- `cob_driver`: plugins interfacing motors, LEDs, sound system, cameras, batteries and even the facial expressions for the robot, and providing their data in the form of topics and services.
- `cob_navigation`: provides tools for robot navigation, including creation of maps, navigation with/without collision avoidance and navigation in dynamic environments.
- `cob_object_perception`, `cob_people_perception`, `cob_environment_perception`: computer vision libraries used for perception of the environment.
- `cob_manipulation`: manipulator related package, including inverse kinematics, arm motion planning and collision avoidance.

2.4.6 Basic API

Most of the capabilities of COB are exposed in the form of topics and services, forming a higher level API and sparing the user from interacting with low-level sensors and actuators.

The actuators move the three wheels on the base. Since the base kinematics is already calculated by the node, you only need to publish a `geometry_msgs/Twist` type message with linear and angular velocities and the drivers will take care of the rest. The topic names can be seen on Table 1.

Table 1 – Base command API.

Topic Name	Message Type	Information
<code>/base/twist_mux</code> <code>/command_navigation</code>	<code>geometry_msgs/Twist</code>	Velocity topics related to navigation
<code>/base/twist_mux</code> <code>/command_safe</code>	<code>geometry_msgs/Twist</code>	Velocity topics related to teleoperation with collision checking and smoothing
<code>/base/twist_controller</code> <code>/command</code>	<code>geometry_msgs/Twist</code>	Low level Velocity topics for control purposes

Since the torso and the head only have one joint, they receive a set of points forming a trajectory to follow, given by the `control_msgs/FollowJointTrajectory` service message type. This service returns if the robot was able to perform the trajectory, as well as the error in the path. The structure can be seen on Table 2.

Table 2 – Torso and head command API.

Topic Name	Message Type	Information
<code>/torso/joint_trajectory_controller</code> <code>/follow_joint_trajectory</code>	<code>control_msgs/FollowJointTrajectory</code>	Trajectory to move the torso
<code>/head/joint_trajectory_controller</code> <code>/follow_joint_trajectory</code>	<code>control_msgs/FollowJointTrajectory</code>	Trajectory to move the head

The arms, the grippers and the sensor ring are similar to the torso and head, as they act as a service and receive the same message type. Their API is shown on Table 3.

Table 3 – Arms and grippers command API.

Topic Name	Message Type	Information
<code>/arm_left/joint_trajectory_controller</code> <code>/follow_joint_trajectory</code> <code>/gripper_left</code>	<code>control_msgs/FollowJointTrajectory</code>	Trajectory to move the left arm
<code>/joint_trajectory_controller</code> <code>/follow_joint_trajectory</code>	<code>control_msgs/FollowJointTrajectory</code>	Trajectory to move the left gripper
<code>/arm_right/joint_trajectory_controller</code> <code>/follow_joint_trajectory</code> <code>/gripper_right</code>	<code>control_msgs/FollowJointTrajectory</code>	Trajectory to move the right arm
<code>/joint_trajectory_controller</code> <code>/follow_joint_trajectory</code> <code>/sensorring</code>	<code>control_msgs/FollowJointTrajectory</code>	Trajectory to move the right gripper
<code>/joint_trajectory_controller</code> <code>/follow_joint_trajectory</code>	<code>control_msgs/FollowJointTrajectory</code>	Trajectory to move the sensor ring ring

For the sensors, there are three lasers scans that cover the entire circumference of the robot, shown on Table 4. Even though they are separate entities, there are nodes that transform the three different measurements into a single one for easier use later on.

Table 4 – Laser Scan API.

Topic Name	Message Type	Information
/base_laser_front/scan	sensor_msgs/LaserScan	Front laser scan
/base_laser_left/scan	sensor_msgs/LaserScan	Left laser scan
/base_laser_right/scan	sensor_msgs/LaserScan	Right laser scan
/scan_unified	sensor_msgs/LaserScan	Unified laser scan

There are also cameras in the torso, head and in the sensor ring that collect both raw images and a point cloud representation that includes the distance of each point to the focal point of the camera. Their respective topics can be seen on Table 5.

Table 5 – Cameras API.

Topic Name	Message Type	Information
/torso_cam3d_left/rgb/image_raw	sensor_msgs/Image	Color image of the left torso camera
/torso_cam3d_left/depth_registered/points	sensor_msgs/PointCloud2	Depth data from torso left camera
/torso_cam3d_right/rgb/image_raw	sensor_msgs/Image	Color image of the right torso camera
/torso_cam3d_right/depth_registered/points	sensor_msgs/PointCloud2	Depth data from torso right camera
/torso_cam3d_right/rgb/image_raw	sensor_msgs/Image	Color image of the down torso camera
/torso_cam3d_down/depth_registered/points	sensor_msgs/PointCloud2	Depth data from torso down camera
/sensorring_cam3d_front/depth/points	sensor_msgs/PointCloud2	Depth data from sensor ring camera
/sensorring_cam3d_back/rgb/image_raw	sensor_msgs/Image	Color image of the back sensor ring camera
/torso_cam3d_down/depth_registered/points	sensor_msgs/PointCloud2	Depth data from back sensor ring camera
/head_cam3d/rgb/image_raw	sensor_msgs/Image	Color image of the head camera

Finally, there are also other miscellaneous topics to control the lights and text-to-speak output and publish robot information, shown on Table 6.

Table 6 – Miscellaneous API.

Topic Name	Message Type	Information
/joy	sensor_msgs/Joy	Input commands of joystick
/sound/say	cob_sound/Say	Text for text-to-speak output
/light_base/set_light	cob_light/SetLightMode	Command for base lights
/light_torso/set_light	cob_light/SetLightMode	Command for torso lights
/light_torso/set_light	cob_light/SetLightMode	Command for torso lights
/emergency_stop_state	cob_msgs/EmergencyStopState	Laser and button stop information.
/power_state	cob_msgs/PowerState	Battery information.

3 SLAM

One of the first challenges in indoor navigation for assistive robots is actually finding where they are. Robot localization is fundamental to not only know where the robot needs to go, but also find a path to avoid obstacles.

Mapping is an essential tool in this matter, and it connects the areas of *concurrent mapping* and *localization problem*. In themselves, both problems are relatively easy and well understood: mapping an environment knowing the localization and localizing the robot having the map in hands are simple tasks. However, the combination of those two problems is hard to solve [25].

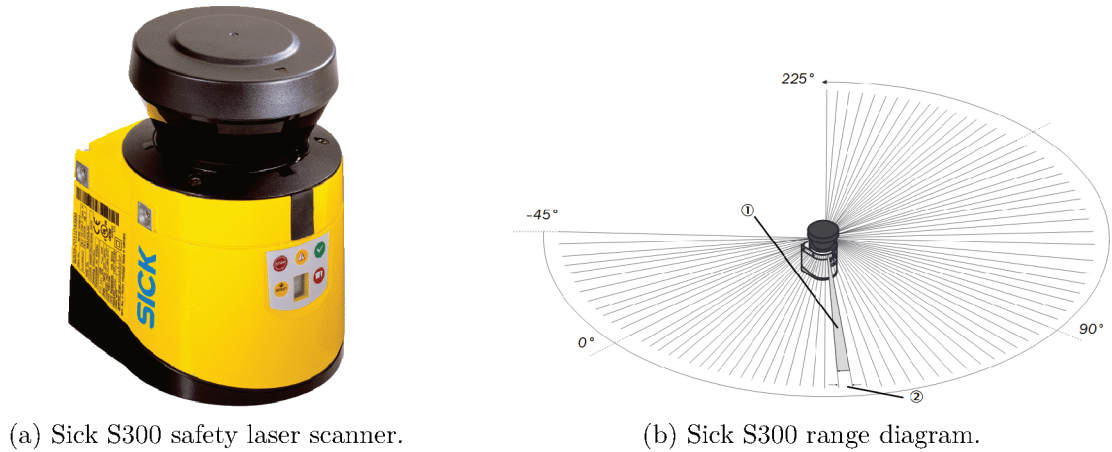
Many SLAM (Simultaneous Localization and Map Building) algorithms emerged to try and solve these problems, using different approaches and a range of sensors to do the task. They also combine data from different sensors to provide higher precision. They often rely on assumptions about the system's nature. Some of the algorithms assume that the noise in the sensors can be modeled as a Gaussian distribution and that the movement of the robot is linear and apply Kalman Filters to predict the current state, based on the last state and the odometry data.

3.1 Sensors

The laser scanner (Figure 11a) is the most popular way to capture data about the environment. They work by sending beams of light in a direction and measuring the time it takes for the light to travel back and forth between the object and the sensor, either directly, by measuring time of flight, or indirectly, by measuring the phase-shift [26]. As you want the information to be gathered on more than a single point, the scanner can rotate a mirror or even the whole sensor to gather measurements from all directions, as shown on Figure 11b. The LIDARs that do not contain moving parts are called solid state LIDARs.

In the time of flight measurement, a stopwatch is started when the beam of light is emitted and stopped when it hits the sensor, giving a measurement of time t . Since the speed of light c is well known, the calculations are very straight forward, as shown on Equation (3.1). This form of measurement depends heavily on the quality of construction, as noise (electronics, radiation in the room) and timing (stopwatch precision, pulse detection) can lead to significant errors in the final result, and averaging and filtering help gather more reliable data [10].

$$D = \frac{c \cdot t}{2} \tag{3.1}$$



(a) Sick S300 safety laser scanner.

(b) Sick S300 range diagram.

Figure 11 – Laser Scanner [27].

The second way of calculating the distance is by using phase-shift techniques, assuming there will be a difference in phase between the beam of light emitted and received. This varies according to the frequency and time traveled according to

$$\phi = \omega \cdot t \quad (3.2)$$

where ϕ is the phase-shift, t the time traveled and ω the angular frequency of the wave. Isolating t and substituting in Equation (3.1), we can derive the Equation (3.3) that dictates the distance based on frequency. This technique also requires more complex signal processing structures like a heterodyne for good measurements [10].

$$D = \frac{1}{2} \frac{c \cdot \phi}{\omega} = \frac{1}{4\pi} \frac{c \cdot \phi}{f} \quad (3.3)$$

3.2 Localizing the robot

3.2.1 Wheel odometry

Wheel odometry is one of the most simple ways to determine the position of the robot based on the starting point. Considering a flat surface, simply taking the turns made by each wheel and the steering actions, it is possible to estimate the path the robot took by applying the forward kinematics of the base.

Considering the robot from Figure 12, the position according to the global coordinate system can be represented by $\xi_I = [x, y, \theta]$ and the velocity $\dot{\xi} = [\dot{x}, \dot{y}, \dot{\theta}]$. The velocity model for the robot coordinate system can be written as [10]:

$$\dot{\xi}_R = \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = R(\theta) \cdot \dot{\xi}_I \quad (3.4)$$

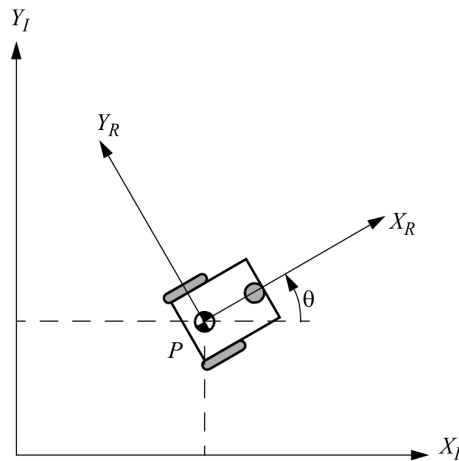


Figure 12 – Forward kinematics of mobile robot [10].

It is easy to calculate the inverse problem using the inverse matrix $R^{-1}(\theta)$, converting back from the robot coordinate frame to the global frame. The only missing parameter is how the wheel kinematics translates to the $\dot{\xi}_R$ robot velocity, and that will depend on the robot wheel arrangement (wheel type, number of wheels, radius) and the robot dimensions.

The main disadvantages of wheel odometry are that the robot is limited to flat terrain, and even then slippery and small changes to forward kinematics (i.e. the radius of the wheel changes slightly) can accumulate error over time, as there is not a suitable way to correct the error without other sources of information.

3.2.2 Laser odometry

Laser odometry bases itself on Laser Scanner data to localize the robot. Instead of using odometry to watch how the robot moves in the environment, it uses the collected data to see how the environment moves in relation to the robot.

One approach to this calculation would be taking two consecutive laser scanners and comparing them. It is assumed that the environment is mostly static so not much should change from one set of data to the other. The transformation that minimizes the distance between points from the two sets is then considered to be the movement of the robot in that time slot [28].

As the small movements are integrated over time, this approach has the same flaws as wheel odometry, with error accumulating over time. However, the robot can improve accuracy by keeping a laser scanner measurement history and repeating the minimization between many poses. It can also take advantage if it revisits a point in the past where the measurement was more accurate, as it can use data from that measurement for comparison.

3.3 The localization and mapping problem

The localization process can be expressed mathematically as follows [29]. Since all the measurements are discrete and supposing we want the localization of the robot through time, the set of poses over time can be represented as:

$$X_t = \{x_0, x_1, \dots, x_t\} \quad (3.5)$$

The map can also be represented by a variable M as follows. Notice also that the map is considered to be time-invariant in this case, and only depends on n which is the number of features in the world.

$$M = \{m_0, m_1, \dots, m_{n-1}\} \quad (3.6)$$

In order to evaluate both variables X_t and M , it is necessary to have some idea on how the robot is interacting with the world. This can be for instance the measurements of robot odometry (wheel or laser odometry) or IMU data. The set of these measurements is defined as:

$$U_t = \{u_0, u_1, \dots, u_t\} \quad (3.7)$$

To also build the map, the robot will need the set of observations of the world, that can come as measurements from 3D cameras, LIDAR, sonar, etc. This set of measurements is defined as:

$$Z_t = \{z_0, z_1, \dots, z_t\} \quad (3.8)$$

Since every measurement is noisy, the position can only be represented as a *belief*, where the belief is the probability that the robot is in a position given the set of inputs and measurements. This belief depends on the set of observations from the past. This can be represented as:

$$bel(x_t, M) = p(x_t, M | u_{1,t}, z_{1,t}) \quad (3.9)$$

There are three main approaches to solve this problem and calculate the belief: Extended Kalman Filters, Particle Filters, and Graph-based.

One of the concerns of mobile robots is how to build a map that is compatible with the environment and represents obstacles properly. While in some applications it is possible to have a pre-compiled map from the environment using the floor plan as a reference, those can be obsolete when dealing with highly dynamic environments or when objects get in the way. Even in static environments, there is a need to compensate for faulty or noise sensors, errors in the localization, and the pre-compiled maps should only be used as complementary information. One of the techniques that emerged to solve these problems,

and later was adopted as the core of many SLAM algorithms, is the occupancy grid [30] representation.

The occupancy grid is a form of representing obstacles in 2D or 3D where each cell on the grid stores the probabilistic information of that area. This is especially useful because it provides a comprehensive way to fuse sensor data, based on probability, instead of out-of-the-box algorithms that require fine tuning to work.

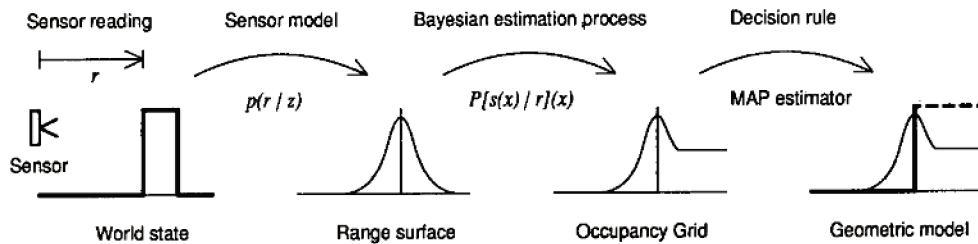


Figure 13 – Steps when building an occupancy grid [30].

According to Figure 13, the first step when building a sensor grid is to get the sensor reading. The next step is build a sensor model $p(r|z)$. Then, the Bayesian estimation is applied, based on all the observations before and current observations, to update the map. Finally, the world model is obtained using an estimator such as *maximum a posteriori* (MAP). All those steps are done by the SLAM algorithms using different techniques. Even with default tuning, those algorithms are good enough to work on most scenarios.

Naturally, the obstacle cell is labeled as occupied, with probability 1. All the cells until this one are labeled as empty, with probability 0. The unknown cells are labeled unknown, with probability 0.5. Figure 14 show the graphical representation of this process.

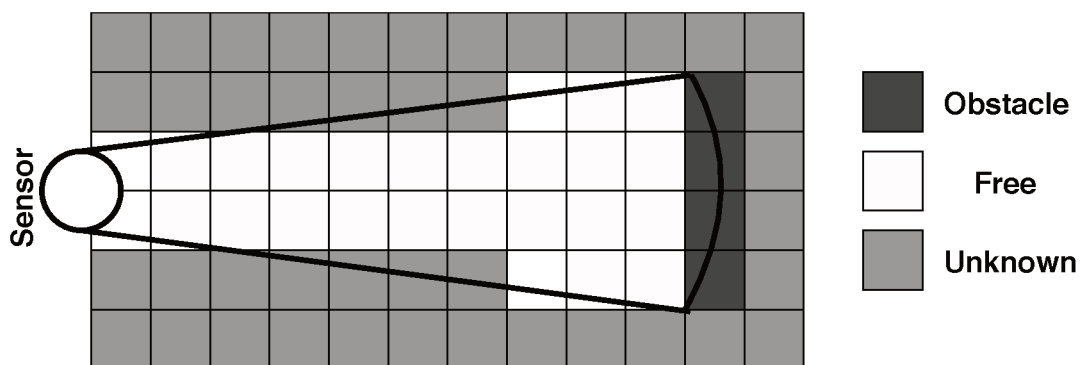


Figure 14 – Illustration of occupancy grid for a single sensor.

The process is iterative and, as measurements from different points of view and different sensors grow, the maps become more and more complete, as shown on the occupancy grid proposed by Elfes[30] seen on Figure 15. Data fusion between sensors can also be done to build a more accurate global map.

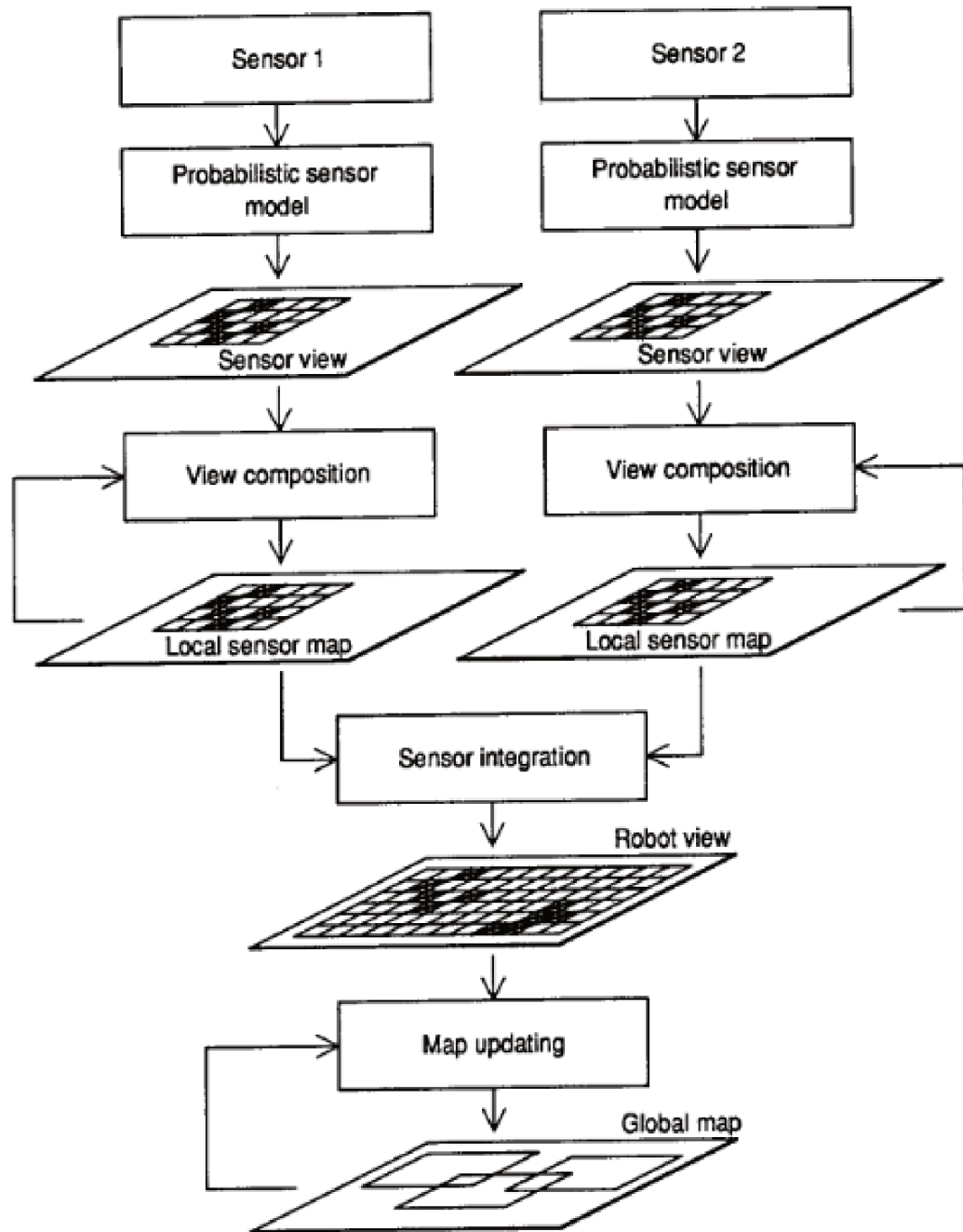


Figure 15 – Occupancy grid algorithm for multiple sensors proposed by Elfes[30].

3.4 ROS SLAM algorithms

3.4.1 Gmapping

Gmapping is by far the most famous SLAM algorithm in robotics for a number of reasons. It was originally proposed by Doucet *et al.*[31] in 2000 as a way of combining particle filter algorithms with Rao-Blackwellized techniques. The main advantage of their approach is dealing with non-gaussian distributions that Kalman filters cannot deal with without rough linearization, as well as being less complex to implement.

In [32], Grisetti proposed improvements to Doucet *et al.*[31] techniques in order to reduce complexity and make the resampling step better. It works by first reducing the number of particles needed to be stored combining the current laser scan observation and odometry information, contrary to previous approaches where only odometry was used, reducing the estimation error and getting a more refined map. The second technique is using adaptative resampling, meaning that resampling only has to be done when needed, reducing the problem with particle depletion, when only a few particles are in high-probability areas and a lot of particles represent pretty old and unreliable information.

3.4.2 Hector

Hector SLAM first emerged in 2013 to solve the very specific problem of mapping in uneven terrains. It was aimed to be used in rescue robots that have to be robust enough to drive through ramps and obstacles and still be able to estimate the trajectory and map without reliable odometry information. Instead of trying to filter data to only include useful data, Hector pose estimation completely drops odometry data in favor of laser scanner data. It instead uses a fast LIDAR data scan-matching to estimate odometry. In order to estimate 6DOF when moving in uneven terrain, the algorithm also needs an IMU device, and optional localization devices like GPS, barometers, and accelerometers. All the data is then fused using an Extended Kalman Filter (EKF), and not including odometry information is a simple way to exclude all errors caused by wheel spin, drift or slippery ground [33].

3.4.3 Karto

Karto is a graph-based SLAM algorithm developed by Karto Robotics and made open-source in 2010. Graph-based SLAM, proposed initially by Lu and Milios[34], works by organizing the robot pose information into a graph and then optimizing it to make it more consistent and minimize an error function. While the construction of the graph is heavily sensor dependent, optimizing the graph is computationally expensive and the reason why Graph-based SLAM took so long to become popular [35].

The graph is constructed representing every pose for the robot as a node in the graph. Every robot movement is then represented as an edge in the graph that connects the poses, this data in the edge usually being the odometry information. If the robot comes back to a known position, the algorithm does the loop closure and connects the graph to a previously known node. Since odometry is not reliable, it is corrected using the laser scanner measurements. The scan observation in both poses is then matched to calculate the virtual transformation that should map one measurement optimally to the other. Let $z_{i,j}$ be the odometry information between poses i and j and $\hat{z}_{i,j}$ be the expected

measurement, the error $e_{i,j}$ can be calculated simply subtracting one from the other, i.e.,

$$e_{i,j}(x_i, x_j) = z_{i,j} - \hat{z}_{i,j} \quad (3.10)$$

The goal is to ultimately build a function $F(x)$ using the log-likelihood strategy:

$$F(x) = \sum_{\langle i,j \rangle \in \mathcal{C}} e_{i,j}^T \Omega_{i,j} e_{i,j} \quad (3.11)$$

so that it can be minimized by the optimization algorithm for x^* , i.e.,

$$x^* = \operatorname{argmin}_x F(x). \quad (3.12)$$

In other words, the question is what is the optimal path that minimizes the observed error, given the distribution Ω and the measurements z and \hat{z} . The optimization is the main problem when dealing with graph-based SLAM, and many different approaches exist to solve this problem. Karto uses Sparse Pose Adjustment, which takes advantage of the sparsity on the large matrices required to solve the optimization problem [36].

3.4.4 Cartographer

Cartographer is a fairly recent SLAM technique developed by Google. The concept behind the algorithm can be seen on Figure 16. The main idea is separating SLAM into two different problems: local SLAM and Global SLAM. The main objective is not having to deal with big maps or representations while mapping a new area. Instead, a submap is created for the local area and updated every new scan. Every scan is also tested against the submap using a Ceres-based scan matcher, to do pose optimization.

The idea of having submaps is that it is only built using a few scans, meaning that the estimate should be very close to reality. As the submaps grow larger, so does the error, meaning that at every few scans a new submap is started. The Global SLAM thread will then have a collection of submaps to compute the whole map running loop closure [37].

3.5 Evaluating SLAM performance

There is a lot of debate on how to evaluate SLAM performance. Since there are plenty of algorithms using different techniques, each one of them having their own set of parameters, there is a need to evaluate them and tell which one does better in each scenario. A lot of this evaluation is done visually, assisted by a human that tells whether the occupancy grid is adequate considering the building floor plans. But as SLAM algorithms get more precise, it is difficult to draw conclusions just from the appearance itself. Additionally, there is the problem of not having the floor plans for publicly available datasets, making it harder to compare between methods [39].

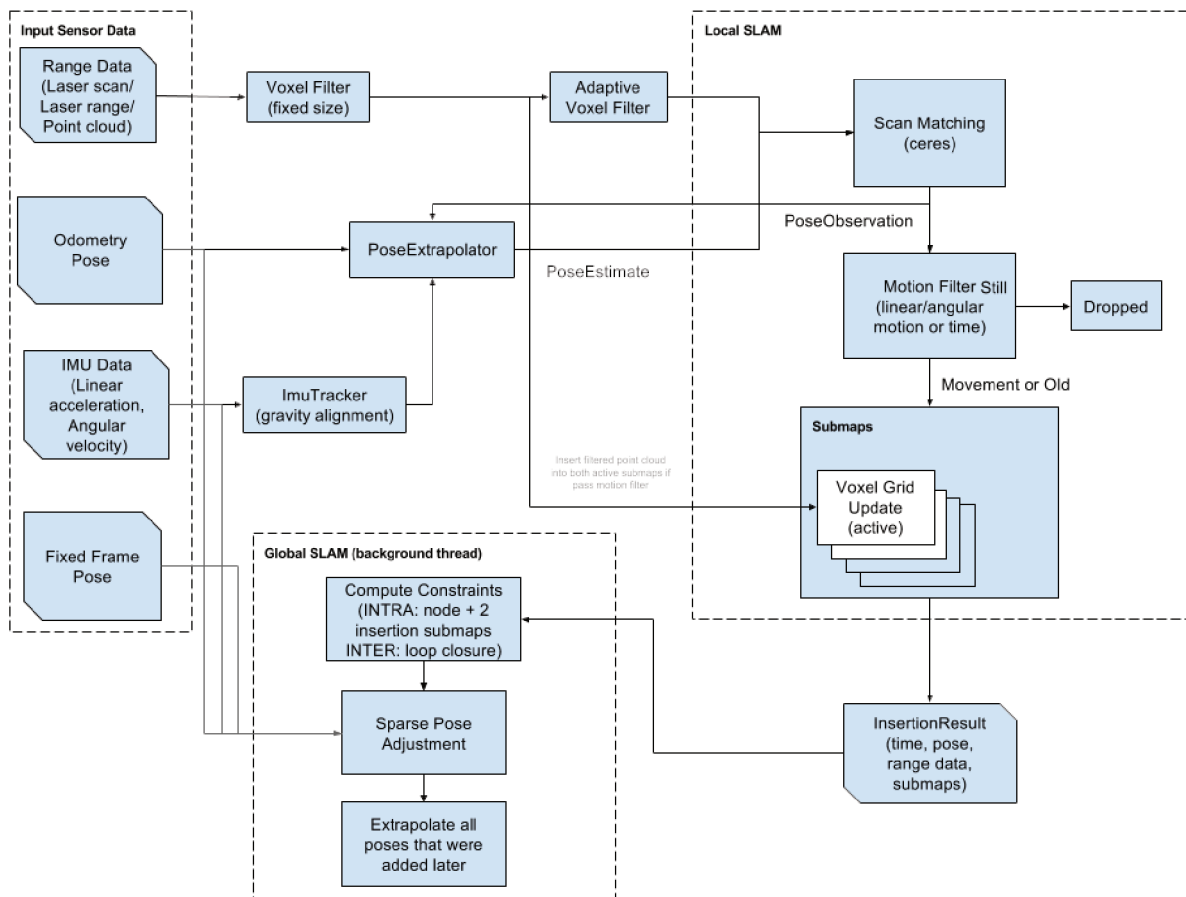


Figure 16 – High-level System overview of Cartographer [38].

According to Amigoni, Gasparini and Gini[11], the following issues have to be addressed when comparing SLAM:

- The dataset must be publicly available, examples being MIT Killian Court or the Intel Research Lab dataset [39].
- All algorithm parameters have to be indicated.
- The behavior for the algorithm has to be tested with different parameters in order to evaluate robustness.
- The dataset must include a closed-loop test, where the robot runs around an environment and comes back to the same place, to test against algorithm divergence.
- The ground truth must be used to evaluate results when available.
- Bad mapping results must be shown.

The most natural way of analyzing the poses is the squared error from the pose estimates against the ground truth. By calculating the distance from one to another and adding over time, we can get a good metric for the pose error. Considering the x_i as the

pose calculated by the SLAM and x_i^* the ground truth pose and N the total number of poses, we can derive the squared error $\epsilon(x)$ as follows:

$$\epsilon(x) = \frac{1}{N} \sum_{i=1}^N (x_i - x_i^*)^2. \quad (3.13)$$

Kümmerle *et al.*[39] proposes a framework to analyze mapping accuracy that uses visual inspection in order to estimate the relations between the robot poses and the environment. The estimation is then compared to the SLAM results and the final metric is the "deformation energy" required to transform the mapped result into ground truth. In other words, each of the N_c displacements $\delta_{i,j}$ is compared against the ground truth $\delta_{i,j}^*$ using $\epsilon(\delta)$ given by

$$\epsilon(\delta) = \frac{1}{N_c} \sum_{i,j} \text{trans}(\delta_{i,j} \ominus \delta_{i,j}^*)^2 + \text{rot}(\delta_{i,j} \ominus \delta_{i,j}^*)^2 \quad (3.14)$$

for a set of (i, j) pairs. The displacement $\delta_{i,j}$ is simply calculated by the transformation from a local measurement between two known poses, from pose x_i to pose x_j , using a distance function like the Euclidian Distance. Evaluating the displacement instead of the global position is great because it makes the evaluation resilient to small errors at the start of mapping that would impact every subsequent global position, even when the mapping in the next steps is done correctly. Since the ground-truth displacement is not available, this implementation relies on the fact that the relation between two poses can be calculated using the laser scanner, each pose later evaluated by a human. It also relies on a good enough initial guess, also human assisted. The authors also assume just evaluating the poses without evaluating the resulting map is enough for SLAM benchmarking. While this holds true for most cases, it is still very hard to infer global performance, as global displacements (large enough distance between i and j) also carry the problem of accumulating human error, as each measurement is supervised.

Santos, Portugal and Rocha[40] propose a more in-depth comparison with the publicly available SLAM algorithms that run on ROS. The authors ran both noise-free simulation and real-life experiments with the scenarios and analyzed. The maps collected were binarized and aligned and error metric was defined as the normalized sum of distances from all pixels in the resulting map to its nearest neighbour in the ground-truth map. The error metrics were analyzed, also evaluating the CPU usage for each algorithm. The authors, however, didn't provide extensive information on how the maps were aligned, very crucial since the fit has to be optimal in order to do an adequate comparison.

3.6 Proposed evaluation techniques

In order to better evaluate the algorithms, general guidelines will be respected:

- Algorithms available in ROS will be used, to ensure every algorithm is publicly available for testing.
- Different maps will be tested to ensure no algorithm is favored. In each run, each algorithm will receive the same working data in the form as ROS bags.
- The scenario will be simulated in a map carefully generated using a public tool, to make sure everyone can generate the same testing data.
- Maps will test the ability for the algorithm to do accurate mapping, accurate localization, and loop-closure. Different metrics will be used to ensure all these three aspects of each map are analyzed.
- Each algorithm will be run using their default configuration. The goal is to test how each algorithm performs in the general case, instead of finding the optimal configuration that gives us the best results for a specific test.

The manuscripts discussed in Section 3.5 already give us a good indication of what to aim for in a comparison algorithm. The first metric chosen is the one described in [39], as it best describes local error in the form of displacements when analyzing the trajectory of the robot. Since the ground-truth data is now available and doesn't have to be inferred by a human operator, the process can be done autonomously and we can ensure the data perfectly matches the environment. For comparison, we are also going to include the squared error metrics proposed in Equation (3.13).

The second comparison method chosen is the one demonstrated in [40], only that now the approach for lining up the maps and calculating the error metric will be fully described. This approach will help to analyze the quality of the generated map regarding the placement of walls and objects, including their orientation and the amount of noise.

The third metric will focus on analyzing the modeled empty space of each algorithm, to see if the area of the generated room matches the area of the map. This is useful in combination with the last algorithm to see how good is the scale on each map.

3.7 Experimental maps

The maps have to be designed to include the scenarios encountered in real life. Three maps were designed and can be seen in Figure 17. The selection aims at testing three key elements in SLAM: loop closure, scale and localization.

The first map can be seen on Figure 17a. It is meant to test the loop closure capabilities of each algorithm since the robot will have to do the full circle, come back to the same point, and then connect the two pathways. The robot starts at the A position, goes into

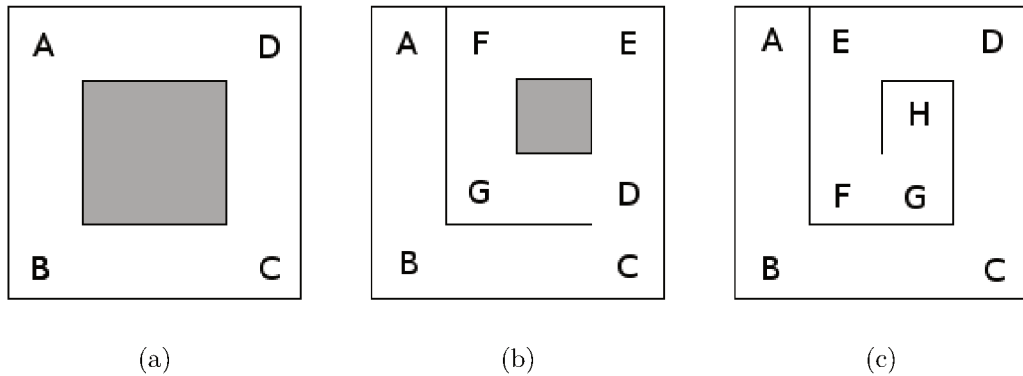


Figure 17 – Selected maps for testing.

a straight line into B, then into a straight line into C. It then spins once around itself, goes to D and finally comes back to A.

The second map seen on Figure 17b. It is a more complex version of the first map. It will test the capability of the algorithm to maintain scale between different rooms, as the corridor on the left and the loop on the right are separated by walls. The robot starts at A and follows the sequence ABCDEFG. It then returns to the home position through D, totalling ABCDEFGDCBA.

The third map, shown on Figure 17c, will test how the localization performs without revisiting positions, as the robot goes to the center of the loop without previous information and then comes back. The sequence followed is ABCDEFGHGFEDCBA.

3.8 Pose metrics

Section 3.6 presented the metrics that will be used to evaluate the algorithms. Figure 18 illustrates some of the measurements that are taken between two estimated poses x_i and x_j , in green, and two ground-truth poses, x_i^* and x_j^* , in red.

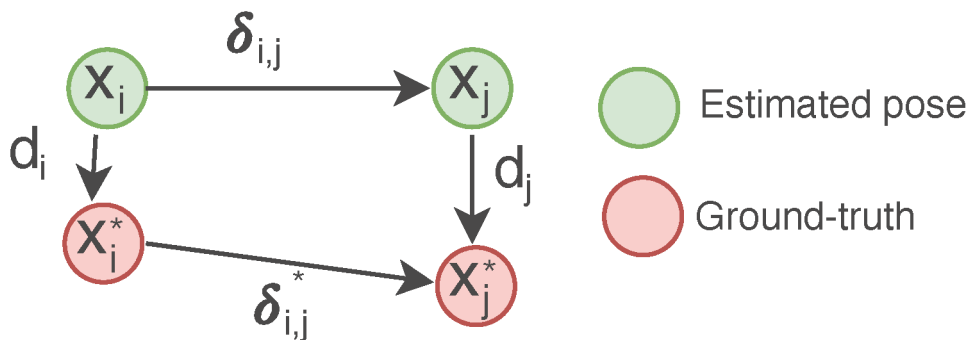


Figure 18 – Representation of metrics taken.

Recalling Section 3.2, the robot position relative to the global reference frame can be represented by two coordinates for the Cartesian position and one for the orientation. Let's define $x_i = [x_i^c, y_i^c, \theta_i]$ and $x_j = [x_j^c, y_j^c, \theta_j]$.

The displacements $\delta_{i,j}$ and $\delta_{i,j}^*$ are calculated by taking the relative transformation between two poses i and j :

$$\delta_{i,j} = x_j - x_i = \begin{bmatrix} x_j^c - x_i^c \\ y_j^c - y_i^c \\ \theta_j - \theta_i \end{bmatrix}. \quad (3.15)$$

The goal is to calculate the contributions $trans(\delta_{i,j} \ominus \delta_{i,j}^*)$ and $rot(\delta_{i,j} \ominus \delta_{i,j}^*)$, respectively the linear displacement and the angular displacement. As mentioned, these transformations can be calculated using the Euclidian distance as following:

$$trans(\delta_{i,j} \ominus \delta_{i,j}^*)^2 = ((x_j^c - x_i^c) - (x_j^{c*} - x_i^{c*}))^2 + ((y_j^c - y_i^c) - (y_j^{c*} - y_i^{c*}))^2, \quad (3.16)$$

$$rot(\delta_{i,j} \ominus \delta_{i,j}^*)^2 = ((\theta_j - \theta_i) - (\theta_j^* - \theta_i^*))^2. \quad (3.17)$$

Kümmerle *et al.*[39] proposes to select the pairs (i, j) from the dataset using scan matching evaluated by a human operator. Since our dataset have the ground-truth, all possible displacements can be evaluated. Given a set of N poses, the linear displacement can then be represented by:

$$\text{Linear displacement} = \frac{1}{N^2} \sum_{i=1}^N \sum_{j=1}^N trans(\delta_{i,j} \ominus \delta_{i,j}^*)^2 \quad (3.18)$$

and the angular displacement by:

$$\text{Angular displacement} = \frac{1}{N^2} \sum_{i=1}^N \sum_{j=1}^N rot(\delta_{i,j} \ominus \delta_{i,j}^*)^2 \quad (3.19)$$

The squared error is easier to calculate, as it relates only to the current pose. We can define the same separate metrics for the translation and rotation from the estimated pose to the ground-truth pose:

$$trans(d_i)^2 = (x_i^{c*} - x_i^c)^2 + (y_i^{c*} - y_i^c)^2 \quad (3.20)$$

$$rot(d_i)^2 = (\theta_i - \theta_i^*)^2. \quad (3.21)$$

The individual pose errors are then summed across the trajectory and normalized according to the following equations:

$$\text{Linear squared error} = \sum_{i=1}^N \frac{1}{N} trans(d_i)^2, \quad (3.22)$$

$$\text{Angular squared error} = \sum_{i=1}^N \frac{1}{N} \text{rot}(d_i)^2. \quad (3.23)$$

3.9 Map alignment metric

Once the map is saved, we have both the reference map and the generated map, as shown in Figure 19. We can then run an algorithm to align the maps properly, as suggested by Santos, Portugal and Rocha[40].

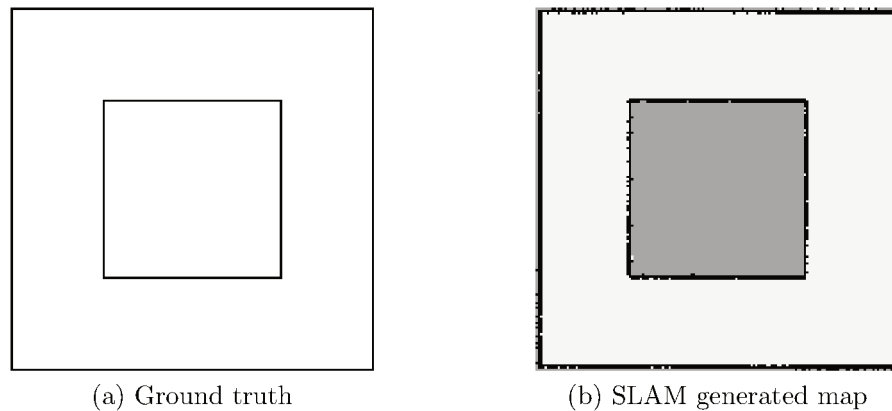


Figure 19 – Occupancy grid representation of ground truth and SLAM generated map.

First, both maps are imported as a point cloud, each pixel representing a point in space, as shown in Figure 20. As you can probably tell, the image is twisted 90 degrees counter-clockwise when being read by the algorithm. This happens because the coordinate system for images is not the same as the Cartesian coordinate system.

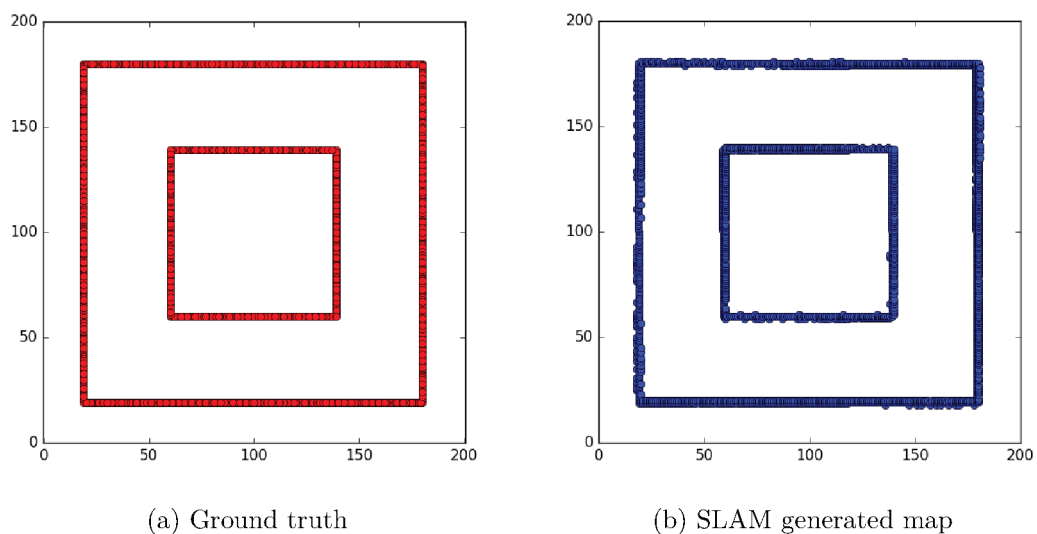


Figure 20 – Point cloud representation of ground truth and SLAM generated map.

ICP, or Iterative Closest Point, is a state-of-the-art way of aligning 3D meshes. It works by finding the transformation that best aligns two distinct point clouds minimizing the error distance between them. For more information about ICP, refer to Besl and McKay[41].

The ICP algorithm used is the one provided by Flannigan[42]. It will calculate the transformation that best aligns the two maps shown in Figure 20. The aligned point cloud can be seen on Figure 21. To do that, we simply call the node launcher with the desired map and algorithm.

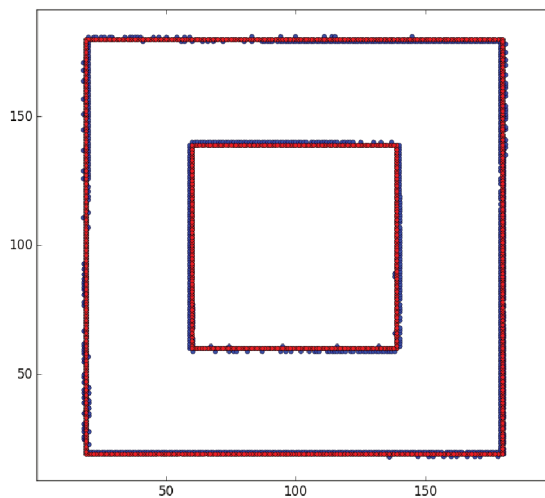


Figure 21 – Alignment of point clouds from Figure 20.

With the maps aligned, we then use the following equation

$$\epsilon(\text{map}) = \frac{1}{P} \sum_{i=1}^P \text{dist}(p_i - p_i^*(p_i))^2 \quad (3.24)$$

to calculate the error metric, where P is the number of points in Figure 20b, p_i represents one point in the data set and p_i^* is the nearest neighbour of p_i in the dataset shown on Figure 20a. All the measurements are in pixels.

3.10 Free space metric

As important as placing the walls in the correct spots, metric that will be checked using ICP, is modeling free space correctly, where the robot can move. This is done by each algorithm by setting the pixels that are free as white. Those white pixels can be counted and checked against the free space in the original map. Since we only want a measurement of scale, we can use the following formula, where w_o is the number of white

pixels in the original image and w_m is the number of pixels in the map generated by SLAM:

$$\epsilon(\text{space}) = \frac{w_o - w_m}{w_o} \times 100. \quad (3.25)$$

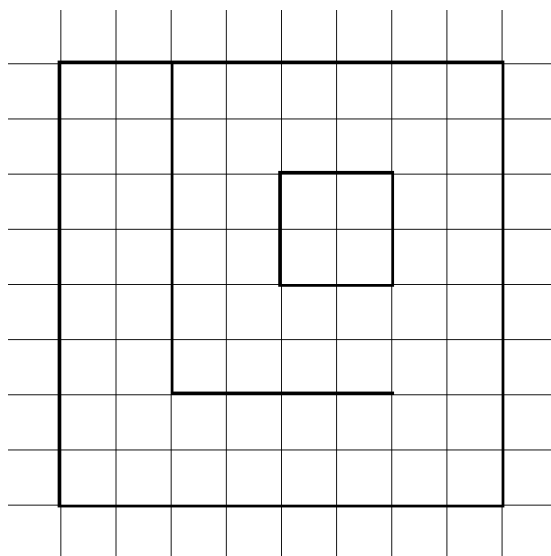
The signal of the result will also give indications about the mapping. If it is positive, it means that the original map has more white pixels than the mapped result, meaning the SLAM was more conservative and mapped less space than available. If it's negative, the algorithm actually mapped space that is not there, meaning the navigation layer will, later on, have to deal with this problem.

4 Experiments and Results

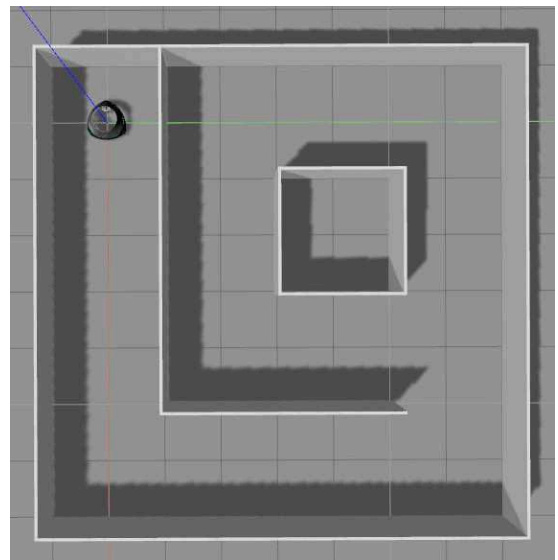
4.1 Building an accurate map

It is very important to build accurate maps for SLAM testing, as the resulting map has to be compared against the ground truth and any inaccuracies might lead to different results. For that, a map generation script has been written to generate accurate Gazebo SDF descriptions of the desired map.

To generate a map, we begin with the model we want for the map to be generated with a small image file. As an example, Figure 22a is 200x200 pixels. We then draw black pixels that will represent the walls in the generated map. After running the script to generate the map, the resulting map can be seen on Figure 22b.



(a) Representation of map in an image editor.



(b) Generated map in Gazebo.

Figure 22 – Scripted map generation for Gazebo.

The main body for the SDF world description can be seen on Listing 4.1. The only information that has to be filled is the `{robot_start_pose}` that will define where the robot will start within the map. The `{content}` tag will then contain a list of walls that compose the test scenario.

```

1 <?xml version='1.0'?>
2 <sdf version='1.6 '>
3   <model name='autogenerated '>
4     <pose frame=''>{robot_start_pose} 0 0 0 0</pose>
5
6     {content}

```

```

7
8     <static>1</static>
9     </model>
10 </sdf>

```

Listing 4.1 – SDF Header.

Each wall can be represented by the XML shown in Listing 4.2. Each will have a unique link name guaranteed by an increasing integer called `{link_number}`. The `{size}` and `{height}` are the dimensions of the wall. Finally, the `{position}` and `{orientation}` represent the location of the wall center in the world, relative to `{robot_start_pose}`.

```

1     <link name='Wall_{link_number}'>
2         <collision name='Wall_{link_number}_Collision'>
3             <geometry>
4                 <box>
5                     <size>{size} {height}</size>
6                 </box>
7             </geometry>
8             <pose frame=''>0 0 0 0 0 0</pose>
9         </collision>
10        <visual name='Wall_{link_number}_Visual'>
11            <pose frame=''>0 0 0 0 0 0</pose>
12            <geometry>
13                <box>
14                    <size>{size} {height}</size>
15                </box>
16            </geometry>
17            <material>
18                <script>
19                    <uri>file://media/materials/scripts/gazebo.material</uri>
20                    <name>Gazebo/Grey</name>
21                </script>
22                <ambient>1 1 1 1</ambient>
23            </material>
24            </visual>
25            <pose frame=''>{position} 0 0 0 {orientation}</pose>
26        </link>

```

Listing 4.2 – SDF for a single wall.

4.2 Setting up the SLAM algorithms

The following configuration was used for each of the tested algorithms. If not specified, the default configuration is being used. Cartographer is the only algorithm that also requires an external Lua file for configuration. All algorithms require the same hardware setup: a source of odometry, in this case a transformation from the robot fixed

frame `odom_combined` to `base_link` and the laser scanner data, represented by the topic `scan_unified`, that combines the data from all three lasers scanners on the robot base.

Every algorithm cycle then publishes the calculated map into the `/map` topic and a correction of odometry using a transformation from `map` to `odom_combined`. This transformation is a small displacement between the two frames to account for errors in the odometry for the robot that accumulates over time and can be reduced by taking the pose corrected by SLAM.

4.2.1 Gmapping

To start Gmapping, we call the launch file shown in Listing 4.3. It starts the node `slam_gmapping` from package `gmapping`. The laser scan topic is remapped to match the robot topic name with the `remap` tag and the odometry frame is provided as `odom_combined`. The `map_update_interval` and number of particles are kept in the default configuration. The `xmin`, `xmax`, `ymin` and `ymax` are the initial size of the resulting map in meters and won't impact the results, as they are automatically increased if the map needs to be bigger. The `delta` is the map resolution and is kept at the default value of 0.05 pixels/meter.

```

1 <launch>
2   <node pkg="gmapping" type="slam_gmapping" name="slam_mapping" output="
3     screen">
4     <remap from="scan" to="scan_unified"/>
5     <param name="odom_frame" type="string" value="odom_combined"/>
6     <param name="map_update_interval" value="5.0"/>
7     <param name="particles" value="30"/>
8     <param name="xmin" value="-8"/>
9     <param name="ymin" value="-8"/>
10    <param name="xmax" value="8"/>
11    <param name="ymax" value="8"/>
12    <param name="delta" value="0.05"/> <!-- map_resolution -->
13  </node>
14 </launch>

```

Listing 4.3 – Gmapping launch file.

4.2.2 Hector

The Hector mapping node is started using the launch script in Listing 4.4, starting the node `hector_mapping` from package `hector_mapping`. We first remap the laser scan to the right topic and use the adequate frame names for the map, base link and odometry link. The `pub_map_odom_transform` is set to `True` to publish the transform from `map` to `odom_combined`. The `laser_min_dist` is set to the minimum value registered for the COB laser scanners.

```
1 <launch>
2   <node pkg="hector_mapping" type="hector_mapping" name="slam_mapping"
3     output="screen">
4     <remap from="scan" to="scan_unified" />
5     <param name="map_frame" value="map" />
6     <param name="base_frame" value="base_link" />
7     <param name="odom_frame" value="odom_combined" />
8     <param name="pub_map_odom_transform" value="true" />
9     <param name="laser_min_dist" value="0.05">
10  </node>
</launch>
```

Listing 4.4 – Hector launch file.

4.2.3 Karto

The Karto mapping node is started using the launch script in Listing 4.5, starting the node `slam_karto` from package `slam_karto`. We only have to set the scan and odometry frames. The `map_update_interval` and `resolution` are set to the same values as Gmapping.

```
1 <launch>
2   <node pkg="slam_karto" type="slam_karto" name="slam_mapping" output="
3     screen">
4     <remap from="scan" to="scan_unified" />
5     <param name="odom_frame" value="odom_combined" />
6     <param name="map_update_interval" value="5" />
7     <param name="resolution" value="0.05" />
8   </node>
</launch>
```

Listing 4.5 – Karto launch file.

4.2.4 Cartographer

The Cartographer node requires a lot of configuration compared to the other SLAM algorithms. Two separate nodes have to be called at start, as seen in Listing 4.6: the `cartographer_node` and `cartographer_occupancy_grid_node`, both from package `cartographer_ros`.

The main Cartographer node does all the sub-map generation and takes as parameters a Lua file with the algorithm configuration, shown in Listing 4.7. The configuration file uses all the default parameters available in Cartographer example files (in file `backpack_2d.lua`), with the exception that the IMU was disabled, as COB doesn't have one. The frames were set accordingly and the option `provide_odom_frame` was set to

true to get the map to odometry transform during execution. The laser scan was changed from multi-echo laser scan to laser scan.

The second node is the occupancy grid node, that reads data from the sub-map list and republishes into the `/map` topic as a standard Occupancy Grid message from ROS. The resolution is also set to 0.05 pixels/meter.

```

1 <launch>
2   <!-- Arguments -->
3   <arg name="configuration_basename" default="cartographer.lua" />
4
5   <!-- cartographer_node -->
6   <node pkg="cartographer_ros" type="cartographer_node" name="slam_mapping"
7         args="-configuration_directory $(find cob_bringup_sim)/launch
8             -configuration_basename $(arg configuration_basename)"
9         output="screen">
10    <remap from="scan" to="scan_unified" />
11  </node>
12
13  <!-- cartographer_occupancy_grid_node -->
14  <node pkg="cartographer_ros" type="cartographer_occupancy_grid_node"
15        name="cartographer_occupancy_grid_node"
16        args="-resolution 0.05" />
17 </launch>

```

Listing 4.6 – Cartographer launch file.

```

1 include "map_builder.lua"
2 include "trajectory_builder.lua"
3
4 options = {
5   map_builder = MAP_BUILDER,
6   trajectory_builder = TRAJECTORY_BUILDER,
7   map_frame = "map",
8   tracking_frame = "base_link",
9   published_frame = "base_link",
10  odom_frame = "odom_combined",
11  provide_odom_frame = true,
12  use_odometry = false,
13  num_laser_scans = 1,
14  num_multi_echo_laser_scans = 0,
15  num_subdivisions_per_laser_scan = 10,
16  num_point_clouds = 0,
17  lookup_transform_timeout_sec = 0.2,
18  submap_publish_period_sec = 0.3,
19  pose_publish_period_sec = 5e-3,
20  trajectory_publish_period_sec = 30e-3,
21  rangefinder_sampling_ratio = 1.,
22  odometry_sampling_ratio = 1.,

```

```
23   imu_sampling_ratio = 1.,
24 }
25
26 MAP_BUILDER.use_trajectory_builder_2d = true
27 TRAJECTORY_BUILDER_2D.num_accumulated_range_data = 10
28 TRAJECTORY_BUILDER_2D.use_imu_data = false
29
30 return options
```

Listing 4.7 – Cartographer Lua configuration.

4.3 Collecting data

For data collection, we first start the robot using the following command in the command line:

```
roslaunch cob_bringup_sim robot.launch robot:=cob4-9 robot_env:=test1
```

We are using `cob4-9` to avoid having to load unnecessary parts of the robot like the arms or the cameras. We then launch the controller to be able to drive the robot around using the keyboard using the following command:

```
roslaunch cob_teleop teleop_keyboard.launch
```

Finally, the data is recorded using the `rosbag` tool. We obviously need the `/tf`, `/tf_static` and `/scan_unified` for the SLAM algorithms. The topic named `/base_pose_ground_truth` is the ground truth data and will be later on used for comparison between algorithms.

The `/base/twist_controller/command` and `/base/odometry_controller/odometry` are respectively the commands given by the keyboard and the calculated odometry after the command has been executed by the robot, and are recorded in case the bag files need to be re-executed. The following command

```
rosbag record /base/odometry_controller/odometry
           /base/twist_controller/command
           /base_pose_ground_truth
           /scan_unified
           /tf
           /tf_static
```

records the data being published in these topics into a file that can be played at will and work as the real robot is sending the scans. This ensures every algorithm will get the same working data in the comparisons.

4.4 Running the automated reconstruction

All the steps required for data parsing by the SLAM algorithms were automated to ensure minimal human interaction is required. Once the data has been collected in the previous step, it can be played using the launch file shown in Listing 4.8.

First, we set the parameters and arguments required. The `use_sim_time` ensures the clock will be used from the bag file, to avoid inconsistencies with time. The robot is set to `cob4-9` because this model will be uploaded for visualization purposes, and it's not required to run the SLAM. We then select the bag file and the algorithm to run together.

The launch file then calls the mapping algorithm, which will launch one of the files described on Section 4.2. Finally, we call the `roslaunch play` node that will play back data already collected to the algorithm, providing clock with the option `-clock` and with a delay `-d 5` of 5 seconds to ensure all nodes are initialized before replaying data.

The last include calls for the RVIZ visualization if requested, as shown on Figure 23.

```

1 <launch>
2
3 <!-- First set up sim time -->
4 <param name="use_sim_time" value="true" />
5
6 <!-- define arguments -->
7 <arg name="robot" default="cob4-9" />
8 <arg name="bag" default="test1" />
9 <arg name="slam" default="gmapping" />
10
11 <!-- Call mapping -->
12 <include file="$(find cob_bringup_sim)/launch/cob_$(arg slam).xml" />
13
14 <!-- Play bag data with clock -->
15 <node pkg="roslaunch" type="play" name="player" output="screen" args="--
    clock -q -d 5 $(find cob_bringup_sim)/bags/$(arg bag).bag"/>
16
17 <!-- Show visualization if requested -->
18 <arg name="rviz" default="false" />
19 <group if="$(arg rviz)">
20   <include file="$(find cob_bringup_sim)/launch/visualization.launch" >
21     <arg name="robot" value="$(arg robot)" />
22   </include>
23 </group>
24 </launch>

```

Listing 4.8 – Automated data parser.

Then, the automated reconstruction can be called passing as parameter the bag file that was saved and the SLAM algorithm to execute on that bag of data using the following command:


```
roslaunch cob_bringup_sim parse_data.launch
          rviz:=true bag:=test1 slam:=gmapping
```

And that will not only launch the bag data `test1` running Gmapping but also launch a visualization tool to see progress, as shown on Figure 23. The point cloud data resulting from the laser scanners (in red) will be feed to the algorithms and the resulting map (in gray) will be published on the `map` topic. Every algorithm also publishes the transform from `map` to `odom_combined`.

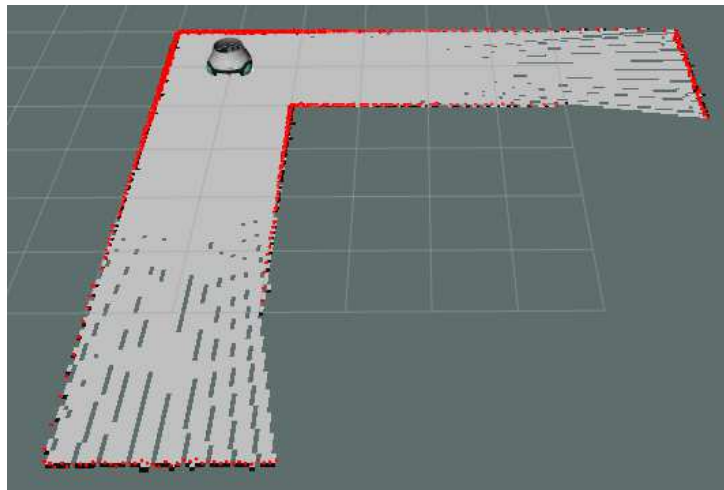


Figure 23 – Running the automated parser node with Gmapping.

4.5 Parsing the data

The data parser is a Python node that will calculate the required metrics. It runs alongside the SLAM algorithm and constantly imports its information listening to the topics and computer resource information. When shut down, the node outputs the desired graphs and metrics. It can be called using the command:

```
roslaunch cob_bringup_sim pipeline.py
```

The node keeps executing the following tasks in parallel:

- Republishes the ground truth (`/base_pose_ground_truth`, as described in Section 4.3) from the ground truth topic and republishes it as a `tf`, since it's easier to do transformations on.
- Reads the `/tf` topic and stores a new pose into a pose history whenever the SLAM pose is updated, as well as the ground truth pose at that point in time.
- Collects data from CPU Usage and Memory Usage with an interval of 0.1 seconds using the `psutil` library [43].

When the node is shut down, the following tasks are executed:

- The pose history is plotted alongside the ground truth.
- The squared pose error is calculated according to Section 3.8.
- The displacement pose error is calculated according to Section 3.8.
- The CPU and Memory usage are plotted over time.
- The summary is generated including the average CPU usage, average Memory usage, translation displacement error, rotation displacement error, translation squared error, rotational squared error.

4.6 Exporting the map

The map can be exported using the `map_saver` node from the package `map_server`. To export the map, simply call the node with the option `-f` and the map name, as in the following command:

```
roslaunch map_server map_saver -f map_name
```

Since Cartographer uses a different approach for generating submaps, the data has to be saved as a `.pbstream` first to generate the full map. To generate the map, we first have to tell the node to finish the trajectory calling the `/finish_trajectory` service. Then, we export the `.pbstream` file and use it to generate the map. The following sequence of commands represent this process:

```
rosservice call /finish_trajectory 0
rosservice call /write_state "filename: '${HOME}/file.pbstream'"
roslaunch cartographer_ros cartographer_pbstream_to_ros_map
  -pbstream_filename ${HOME}/file.pbstream
```

Because the resulting image for the map doesn't always have the correct dimensions (approximately the same as the ground truth map), we have to crop the empty gray areas before running the map comparisons. We also want to convert from `.pgm` saved automatically to `.png` that the algorithm expects. To do that, we simply call the conversion function from ImageMagick, using the `-trim` options to trim the gray borders, using the following command:

```
convert -rotate 90 map_name.pgm -trim map_name.png
```

Why the image is rotated 90 degrees is explained in Section 3.9. After exporting the map, we can calculate the ICP metric using the following command:

```
roslaunch cob_bringup_sim icp_map_comparison.launch
    map:=test1 slam:=gmapping
```

4.7 Results

All the mapping was done in the same machine equipped with an Intel Core i5-4430@3.00 GHz and 8 GB DDR3 memory. All the CPU measurements reflect the usage relative to a single core usage, meaning that values higher than 100% represent the usage of more than one core at a time. The memory measurements are USS or "Unique Set Size", which is the amount of memory that would be freed if the process was terminated. The Gmapping version used was 1.3.10 [44], the Hector version used was 0.3.5 [45], the Karto version used was 0.7.3 [46] and the Cartographer version used was 0.3.0 [47].

The mapping results for the three test maps designed on Section 3.7 can be seen on Figure 24, for the test map 1, Figure 25, for the test map 2 and Figure 26 for the test map 3. The respective data collected during execution can be seen on Table 7, Table 8 and Table 9.

At visual inspection, we can see that for the first map (Figure 24), Karto Slam and Cartographer perform better, as the noise in the walls is lower. They look straight and sharp, as opposed to Gmapping and Hector, where the walls look noisy. If we inspect the results of Table 7, we can see that this reflects in Karto having the lowest localization error between all algorithms for this map for all metrics. Even though the Hector reconstruction is not as great, it scores second place in localization error, followed by Cartographer and finally Gmapping, although Cartographer is better at poses and Gmapping is better at angles.

In the second map, Figure 25, the results are quite the opposite. Hector shows the best pose estimate in displacement, but Gmapping overcomes in squared error. Karto remains with good results but Cartographer lags behind. We can actually see why looking at the map, as Cartographer's map is tilted relative to the others. This error of orientation at the start was probably what made Cartographer perform worse in the localization.

In terms of average CPU and Memory usage, the values remained constant throughout the tests. Gmapping shows the highest usage of CPU among all algorithms, consuming almost double of Cartographer, in second place. Karto and Hector show low consumption of CPU, with Hector being the lowest. In terms of memory, Hector jumps ahead in all tests, followed by Gmapping, Hector and finally Cartographer. It is important to notice that despite having low CPU and memory footprint, we are only taking into consideration the SLAM node for Cartographer, and not the obstacle grid node nor the offline tasks executed by the `.pbstream` conversion.

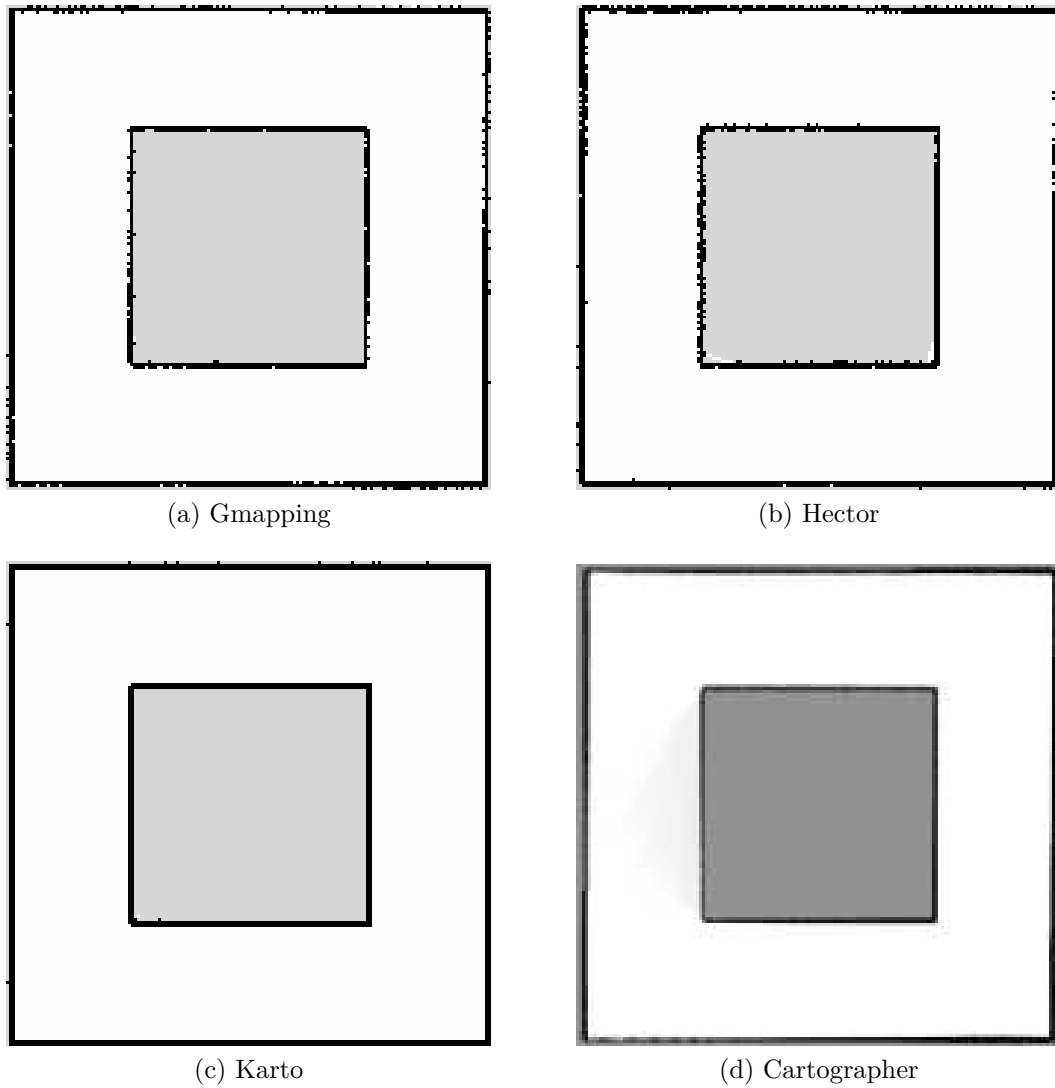


Figure 24 – Results of mapping for first map.

	Gmapping	Hector	Karto	Cartographer
Linear displacement	0.0010116	0.00051379	0.00014934	0.00095104
Angular displacement	4.0385e-05	2.3272e-05	1.2402e-05	0.00010114
Linear squared error	0.0018224	0.00058361	0.00024294	0.00086193
Angular squared error	2.0899e-05	1.6251e-05	6.9566e-06	6.3385e-05
CPU (%)	14.36	3.67	4.74	7.13
Memory (MB)	19.56	26.39	13.18	12.52

Table 7 – Data collected for the first map (lower is better).

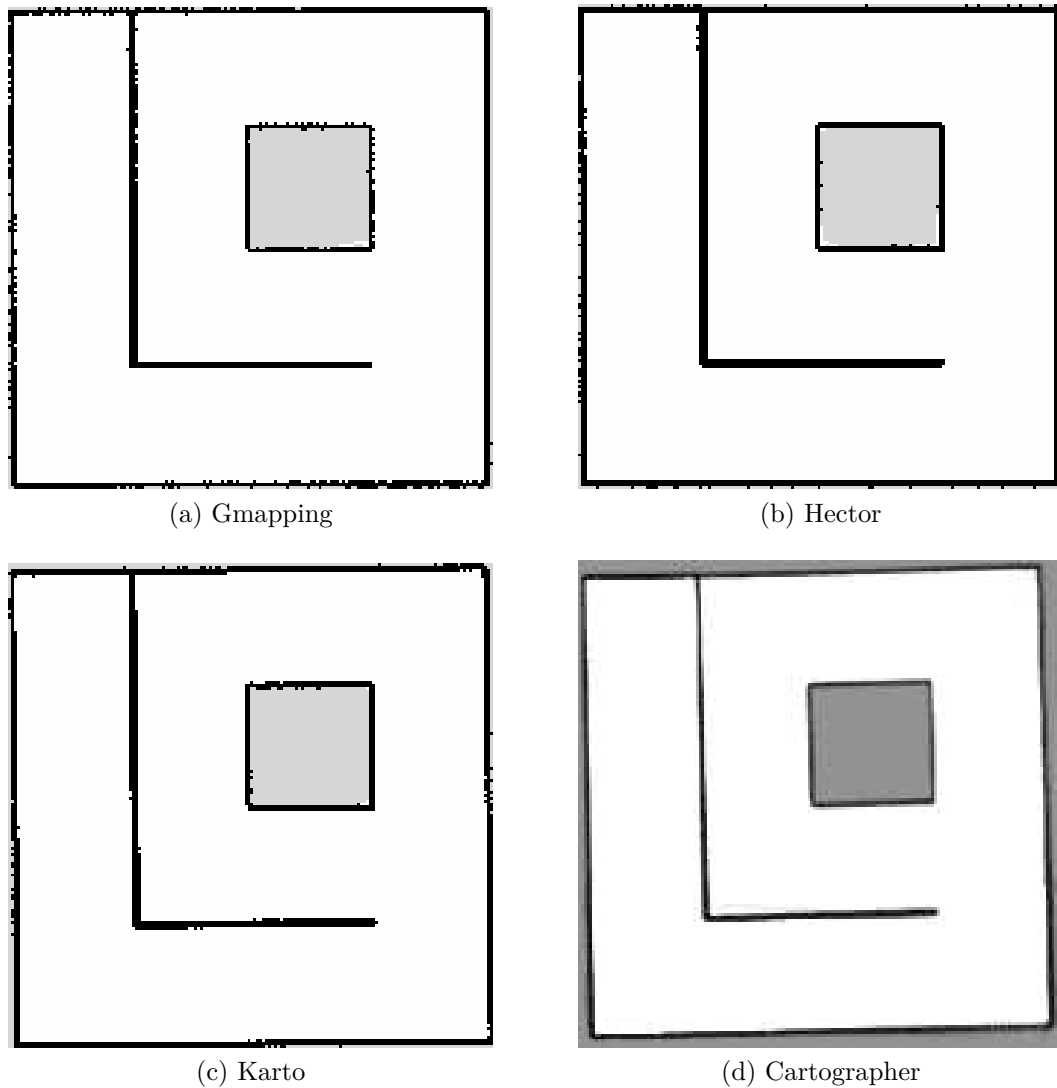


Figure 25 – Results of mapping for second map.

	Gmapping	Hector	Karto	Cartographer
Linear displacement	0.00037009	0.00024439	0.0031410	0.013768
Angular displacement	3.6161e-05	2.6309e-05	1.0607e-05	4.4936e-05
Linear squared error	0.00038672	0.0010135	0.0035834	0.013529
Angular squared error	0.00038672	2.5932e-05	0.00016662	0.00060306
CPU (%)	11.38	3.75	4.72	6.56
Memory (MB)	19.11	26.50	14.45	12.33

Table 8 – Data collected for the second map (lower is better).

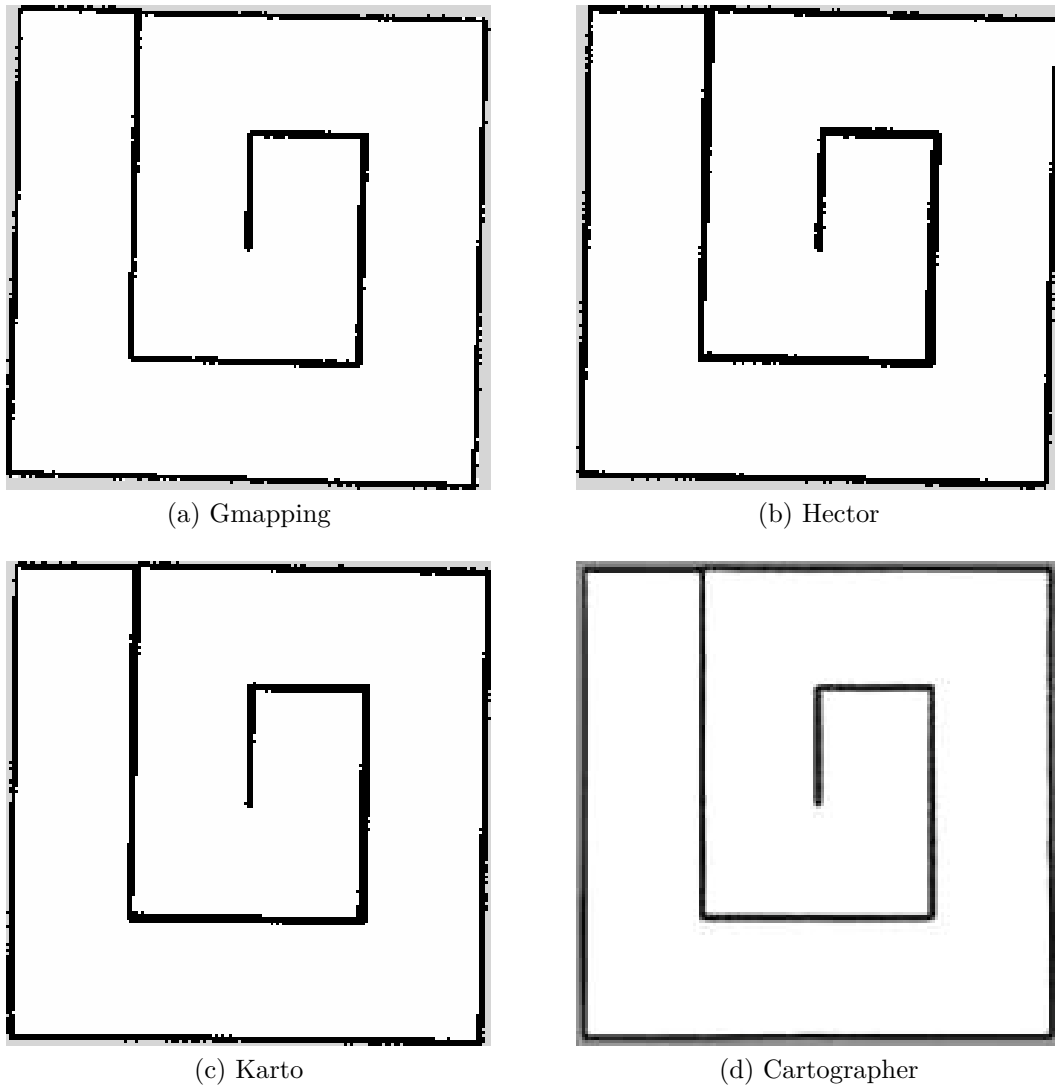


Figure 26 – Results of mapping for third map.

	Gmapping	Hector	Karto	Cartographer
Linear displacement	0.015545	0.013457	0.0050207	0.0012753
Angular displacement	6.0892e-05	9.0999e-05	5.0616e-05	4.7758e-05
Linear squared error	0.022001	0.015299	0.0054754	0.0026521
Angular squared error	0.00055478	0.00080006	0.00033509	2.8089e-05
CPU (%)	12.04	3.73	4.99	6.30
Memory (MB)	19.88	26.32	15.88	14.45

Table 9 – Data collected for the third map (lower is better).

In the third map, Cartographer outperforms every other algorithm in every metric, resulting in a much better map, followed by Karto, and finally Gmapping and Hector with very similar results. It is clear from analyzing the pose error from all the algorithm runs that this method is not giving a good way of measuring accuracy, at least for this test case.

The results of running the ICP matcher with the algorithms can be seen on Table 10. The best algorithm in all cases was Cartographer, scoring lowest. This means two things: most of the walls were placed in the correct spot and the noise is low. In the second place, Gmapping could outperform Hector and Karto, justifying the wide adoption of Gmapping in the robotic world, as it is far easier to set up than Cartographer. Hector and Karto were tied in the last position in this test, as Hector was better at the first map, Karto was better in the second map and the results are approximately the same in the third map.

	Gmapping	Hector	Karto	Cartographer
Test 1	0.46144	0.61088	0.75229	0.34752
Test 2	0.55829	0.76593	0.62868	0.51959
Test 3	0.64751	0.78693	0.75315	0.41721

Table 10 – Results of running ICP over maps (lower is better).

When modeling free space, most of the algorithms show the same result, with Gmapping showing slightly better results than the other. One of the problems Gmapping faces, though, is mapping places inside the walls where it has no information about, as shown on Figure 27. This results in more free space being shown than normal, which might indicate why Gmapping has a lower score. All the algorithms have results with a positive sign, meaning that the less free space was mapped than in fact exists, which is good, as discussed in Section 3.10.

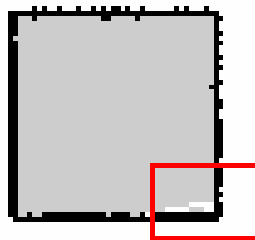


Figure 27 – Incorrect mapping from Gmapping on test 2.

	Gmapping	Hector	Karto	Cartographer
Test 1 (%)	3.71	4.15	4.98	4.91
Test 2 (%)	3.68	4.80	4.61	6.09
Test 3 (%)	3.90	5.24	5.02	5.79

Table 11 – Results of free space mapping accuracy (lower is better).

We are going to analyze the CPU and Memory profiles for each algorithm. To avoid generating too many graphics, only the profile for the second map will be shown because it features both localization and loop closure. Figures 28 to 31 show both the CPU usage and RAM Memory used over time. The x axis represents the number of samples, and since the acquisition frequency was 10 Hz, every 500 samples are equivalent to 50 seconds.

Figure 28 shows how Gmapping performs when running. It is possible to see that the CPU peaks at a value of more than 100% every few iterations, and most of the time it stays around 10%. This peak every few iterations is the result of each laser scan being processed, and the effort seems to stay constant until the end, peaking around the same value. The memory grows in small incremental steps as the map is being constructed, and stops at around 1000 samples, where the map is already near the final state and only a few bits of new information are added.

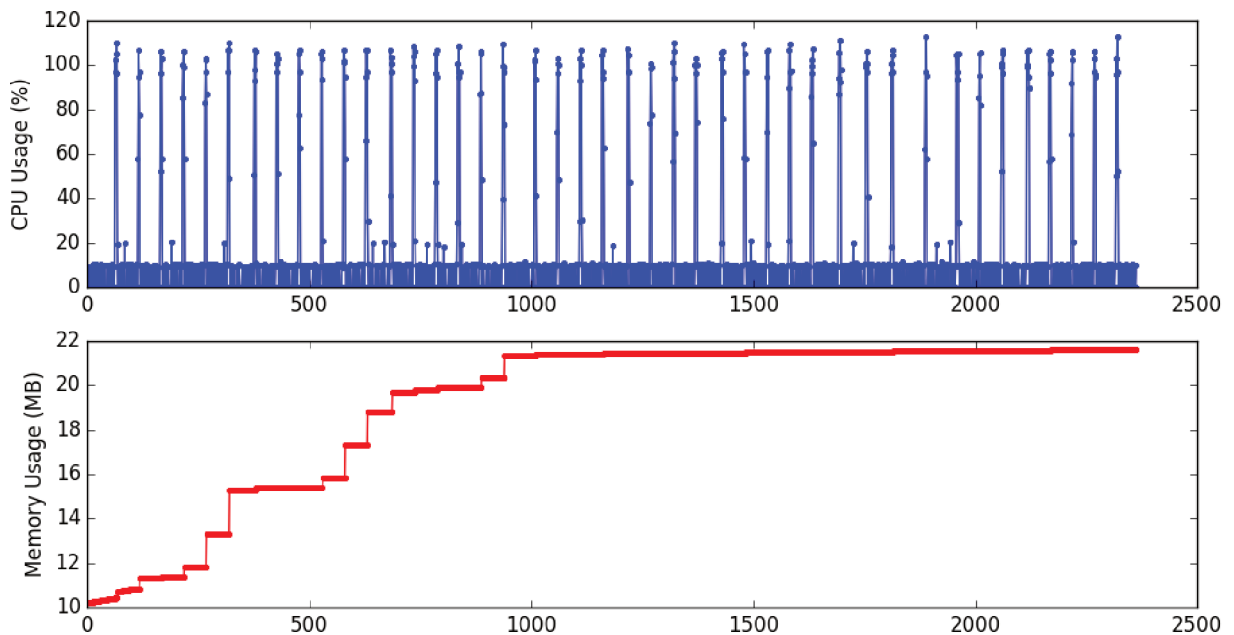


Figure 28 – CPU and Memory usage for Gmapping running map test 2.

As already mentioned, Hector performs better on CPU and worse on memory. Figure 29 shows peaks around 20 % and around 10 %. Most of the time, the algorithm stays idle, to account for the average CPU usage of around 4 %. Memory consumption, in this case, is high from the start and stays the same until the end. One of the reasons might be because it allocates the whole map right from the start, instead of allocating memory

for a small map and increasing as new areas are discovered.

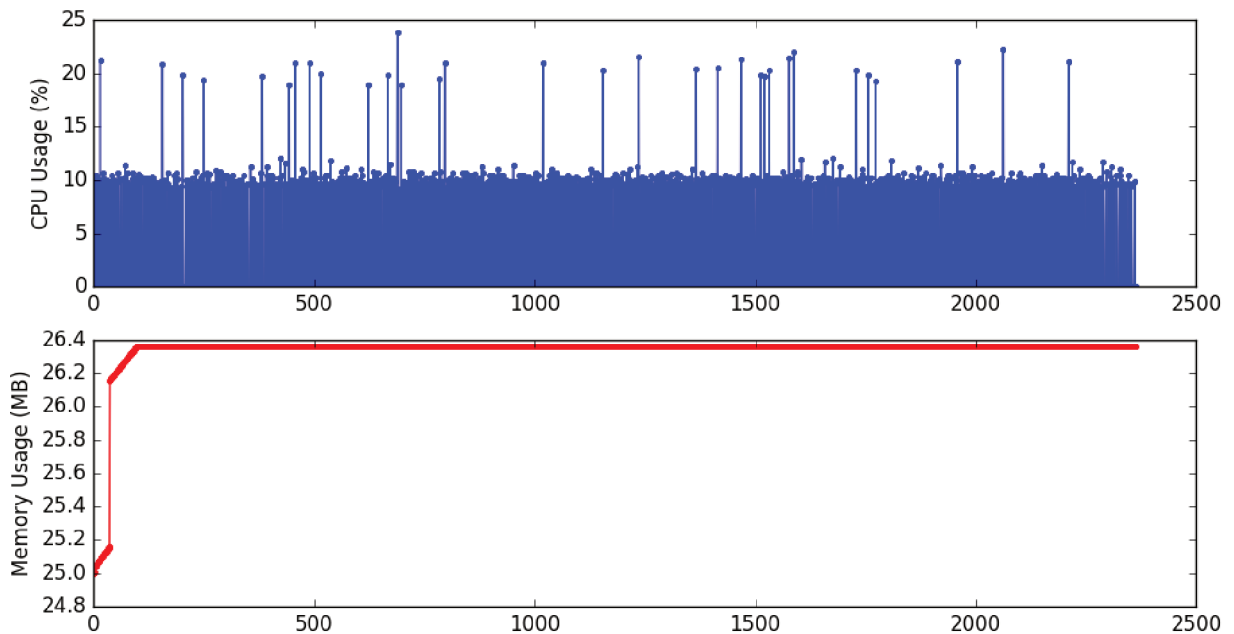


Figure 29 – CPU and Memory usage for Hector running map test 2.

The results for Karto can be seen on Figure 30. For CPU, we can see that the peaks become higher as the algorithm is executed for longer. This is expected as the problem of optimizing a pose-graph becomes harder as more nodes are added to the graph. The memory seems to grow linearly as time progresses, even though no more information is being added after some point in time.

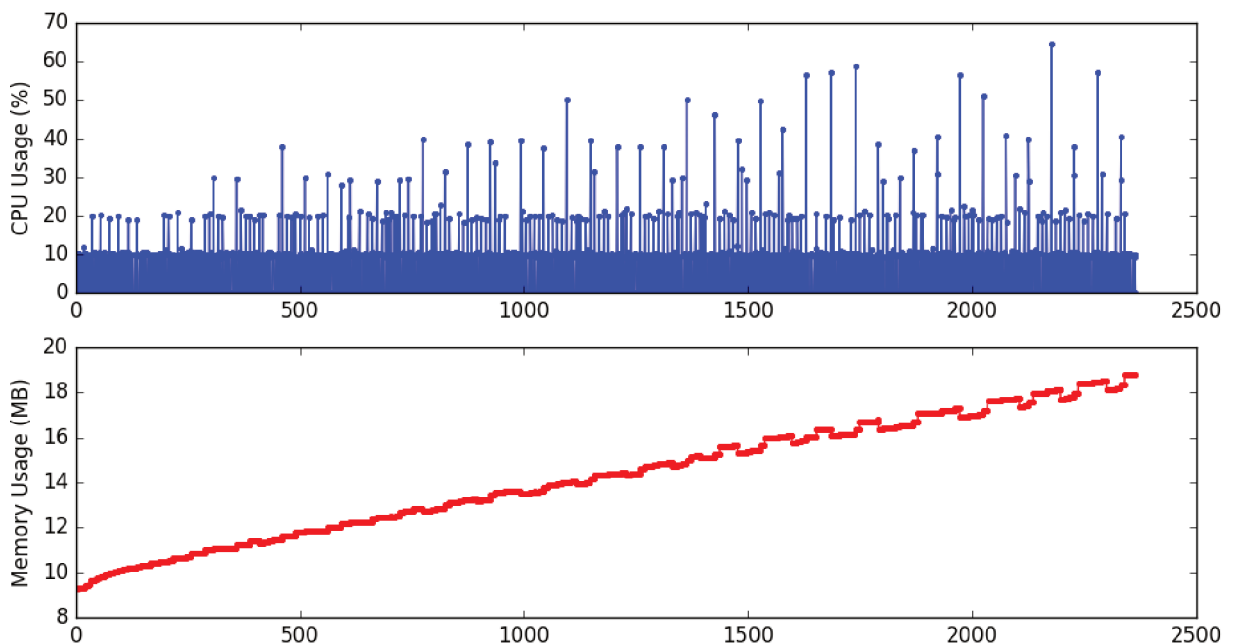


Figure 30 – CPU and Memory usage for Karto running map test 2.

Finally, the results for Cartographer can be seen on Figure 31. It behaves similar to

Gmapping, although with lower peaks, with the memory growing linearly like Karto, with small bumps every 1000 samples.

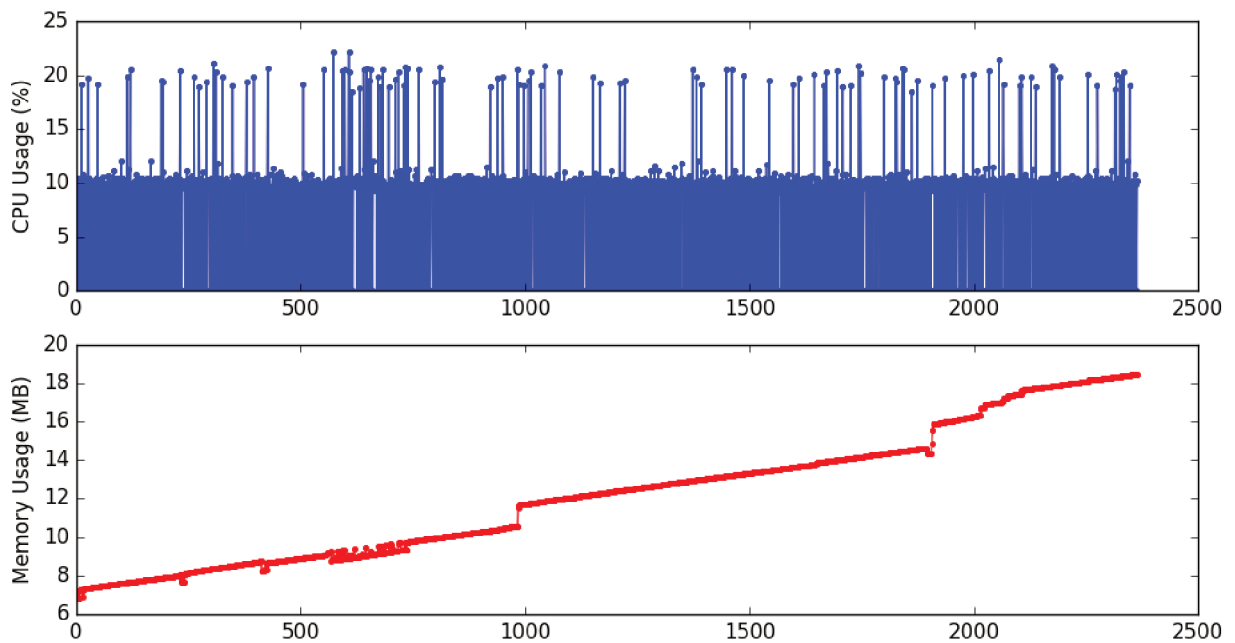


Figure 31 – CPU and Memory usage for Cartographer running map test 2.

In terms of results, the temporal analysis of the graphs brings additional information to the table. All the algorithms show consumption around 10%, differentiating on the peaks and the amount they stay idle. Gmapping was the most intensive on CPU, especially because of all the peaks. Hector and Karto stayed with low consumption and Cartographer were very modest considering its complexity.

5 Conclusions

5.1 Final considerations

This dissertation presented a framework for methodologically analysing SLAM results for different algorithms in an assistive robot environment. The framework was tested using a simulated version of Care-o-bot, developed by Fraunhofer IPA. First, the problem was stated, emphasizing why mapping is a big challenge in mobile robots. The current limitations of mapping benchmarks were discussed. The main aspects of ROS and COB were introduced to give the reader a good understanding of their concepts. The SLAM problem was detailed and some methodologies were proposed, based on the literature review about the subject. Tools were developed to aid the process of comparison, including a map ground-truth generator and a data parser. Finally, four algorithms (Gmapping, Hector, Karto and Cartographer) were benchmarked and tested against the proposed framework.

The tests can give us some insight about the metrics chosen to represent the accuracy of each algorithm. The displacement error and the squared error were consistent with each other. Those measurements were also coincident with the visual quality of the final map.

The CPU and memory metrics were important to analyse the footprint of the robot in each scenario, as well as giving an indication on how the algorithm would behave with continued execution. Even though the modern dedicated hardware found current robots like Care-o-bot is quite powerful, their importance comes with the fact that the lower processing power would require less energy, important in mobile robot depending on batteries, and allow for more utilities to be executed in the same processor, possibly reducing the number of processors needed.

Comparison through ICP and free-space were by far the most important, comparing directly the result of mapping to the exact ground-truth. The ICP comparison gives us a quantitative metric to compare the maps, that shows exactly by how much the walls are out of place and how much noise there is in the reconstruction. The free space comparison shows how well scaled the map, as shorter walls and corridors would increase this metric.

Overall, the chosen algorithms performed very well on all the tests performed. Cartographer scored best in the map accuracy, what is expected considering that it was built to outperform the popular algorithms at the time. It is important to say that few of the possibilities of Cartographer were explored as it also supports 3D SLAM using a point cloud. Gmapping showed consistent results, justifying its wide adoption in the robotic world. Hector didn't perform as well, but it was released to be effective in uneven terrain

in rescue robots, conditions not present in the scenario proposed. Since the original robot tested with Hector had a very limited processor, the modest CPU usage is very fit for its purpose. Karto, tied with Hector, showed good enough results, but its lack of documentation and no recent updates are strong points, and since only a portion of Karto is released to the public, the majority of its development is kept closed source code.

5.2 Future work

Since the goal was to have small and objective tests, not all aspects of a reliable SLAM algorithm were covered. The aspect of robustness, for instance, was not explored in this dissertation. It is important to say that these benchmarks only contemplate 2D SLAM. There is a wide selection of algorithms that perform 3D SLAM, one of them being Cartographer, that could be analyzed, as the 3D SLAM can be converted into 2D SLAM for comparison.

Another aspect not tested here is how the algorithms perform using different configurations. Cartographer offers a lot of configurations regarding the scan matching, pose optimization, filters and even the option to tune Ceres separately. Gmapping offers the option to tune the number of particles and the resampling threshold.

Other tests might include adding bigger maps, to see how well algorithms perform. The aspect of large empty areas also needs to be tested, as algorithms that do not depend on odometry wouldn't have a reference to follow. The same happens in long corridors or featureless environments.

Another interesting metric would be fault tolerance. How each algorithm performs with a noisy sensor or when the odometry has errors (drift, collision or even kidnapped robot) is very important when designing a fault tolerant localization system.

5.3 Contributions

The main contribution of this dissertation is the tools for generation and comparison of maps. As the purpose of the proposed techniques is to have reproducible tests, all the code used is available at the following repository on Github:

<<https://github.com/bvanelli/TCC>>

References

- 1 DAUTH, W. *et al.* German robots—the impact of industrial robots on workers. 2017.
- 2 FEIL-SEIFER, D.; MATARIC, M. J. Defining socially assistive robotics. In: IEEE. *Rehabilitation Robotics, 2005. ICORR 2005. 9th International Conference on.* [S.l.], 2005. p. 465–468.
- 3 FONG, T.; NOURBAKHSI, I.; DAUTENHAHN, K. A survey of socially interactive robots. *Robotics and autonomous systems*, Elsevier, v. 42, n. 3-4, p. 143–166, 2003.
- 4 SAKAGAMI, Y. *et al.* The intelligent asimo: System overview and integration. In: IEEE. *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on.* [S.l.], 2002. v. 3, p. 2478–2483.
- 5 BRAGA, R. A. *et al.* Intellwheels—a development platform for intelligent wheelchairs for disabled people. In: *ICINCO 2008: PROCEEDINGS OF THE FIFTH INTERNATIONAL CONFERENCE ON INFORMATICS IN CONTROL, AUTOMATION AND ROBOTICS, VOL RA-2: ROBOTICS AND AUTOMATION, VOL 2.* [S.l.: s.n.], 2008.
- 6 TANAKA, F. *et al.* Pepper learns together with children: Development of an educational application. In: IEEE. *Humanoid Robots (Humanoids), 2015 IEEE-RAS 15th International Conference on.* [S.l.], 2015. p. 270–275.
- 7 GRAF, B.; HANS, M.; SCHRAFT, R. D. Care-o-bot ii—development of a next generation robotic home assistant. *Autonomous robots*, Springer, v. 16, n. 2, p. 193–205, 2004.
- 8 GRAF, B.; PARLITZ, C.; HÄGELE, M. Robotic home assistant care-o-bot® 3 product vision and innovation platform. In: SPRINGER. *International Conference on Human-Computer Interaction.* [S.l.], 2009. p. 312–320.
- 9 MORI, M.; MACDORMAN, K. F.; KAGEKI, N. The uncanny valley [from the field]. *IEEE Robotics & Automation Magazine*, IEEE, v. 19, n. 2, p. 98–100, 2012.
- 10 SIEGWART, R.; NOURBAKHSI, I. R.; SCARAMUZZA, D. *Introduction to autonomous mobile robots.* [S.l.]: MIT press, 2011.
- 11 AMIGONI, F.; GASPARINI, S.; GINI, M. Good experimental methodologies for robotic mapping: A proposal. In: IEEE. *Robotics and Automation, 2007 IEEE International Conference on.* [S.l.], 2007. p. 4176–4181.
- 12 QUIGLEY, M. *et al.* Ros: an open-source robot operating system. In: KOBE, JAPAN. *ICRA workshop on open source software.* [S.l.], 2009. v. 3, n. 3.2, p. 5.
- 13 ROS Wiki. *Technical Overview.* [S.l.]: ROS, 2014, Revision 36. <<http://wiki.ros.org/ROS/Technical%20Overview>>.
- 14 SMITH, R. *Dynamic Simulations: A whirlwind tour.* 2004. <<http://ode.org/slides/parc/dynamics.pdf>>.

- 15 CRAIGHEAD, J. *et al.* A survey of commercial & open source unmanned vehicle simulators. In: IEEE. *Robotics and Automation, 2007 IEEE International Conference on.* [S.l.], 2007. p. 852–857.
- 16 KOENIG, N. P.; HOWARD, A. Design and use paradigms for gazebo, an open-source multi-robot simulator. In: CITESEER. *IROS.* [S.l.], 2004. v. 4, p. 2149–2154.
- 17 ROBOTIS-GIT. *turtlebot3_simulations.* [S.l.]: GitHub, 2019, commit a6f8e14. <https://github.com/ROBOTIS-GIT/turtlebot3_simulations>.
- 18 SMITH, R. *et al.* Open dynamics engine. 2005.
- 19 COUMANS, E.; BAI, Y. *PyBullet, a Python module for physics simulation for games, robotics and machine learning.* 2016–2018. <<http://pybullet.org>>.
- 20 SHERMAN, M. A.; SETH, A.; DELP, S. L. Simbody: multibody dynamics for biomedical research. *Procedia Iutam, Elsevier*, v. 2, p. 241–261, 2011.
- 21 LEE, J. *et al.* Dart: Dynamic animation and robotics toolkit. *The Journal of Open Source Software*, v. 3, n. 22, p. 500, 2018.
- 22 PETERS, S.; HSU, J. Comparison of rigid body dynamic simulators for robotic simulation in gazebo. *Open Source Robotics Foundation. Available at http://www.osrfoundation.org/wordpress2/wp-content/uploads/2015/04/roscon2014_scpeters.pdf. Accessed September*, v. 8, p. 2016, 2014.
- 23 KITTMANN, R. *et al.* Let me introduce myself: I am care-o-bot 4, a gentleman robot. In: DIEFENBACH, S.; HENZE, N.; PIELOT, M. (Ed.). *Mensch und Computer 2015 – Proceedings.* Berlin: De Gruyter Oldenbourg, 2015. p. 223–232.
- 24 Mojin-Robotics. *Care-o-bot 4.* [S.l.]: Mojin-Robotics, 2019. <<https://www.care-o-bot-4.de/>>.
- 25 THRUN, S.; BURGARD, W.; FOX, D. A real-time algorithm for mobile robot mapping with applications to multi-robot and 3d mapping. In: IEEE. *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on.* [S.l.], 2000. v. 1, p. 321–328.
- 26 AMANN, M.-C. *et al.* Laser ranging: a critical review of unusual techniques for distance measurement. *Optical engineering*, International Society for Optics and Photonics, v. 40, n. 1, p. 10–20, 2001.
- 27 SICK AG. *SICK S300 Safety Laser Scanner Operating Instructions.* [S.l.], 2016.
- 28 JAIMEZ, M.; MONROY, J.; GONZALEZ-JIMENEZ, J. Planar odometry from a radial laser scanner. a range flow-based approach. In: *IEEE International Conference on Robotics and Automation (ICRA).* [s.n.], 2016. p. 4479–4485. Disponível em: <<http://mapir.isa.uma.es/mapirwebsite/index.php/mapir-downloads/papers/217>>.
- 29 THRUN, S.; BURGARD, W.; FOX, D. *Probabilistic robotics.* [S.l.]: MIT press, 2005.
- 30 ELFES, A. Using occupancy grids for mobile robot perception and navigation. *Computer, IEEE*, n. 6, p. 46–57, 1989.

- 31 DOUCET, A. *et al.* Rao-blackwellised particle filtering for dynamic bayesian networks. In: MORGAN KAUFMANN PUBLISHERS INC. *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence.* [S.l.], 2000. p. 176–183.
- 32 GRISETTI, G.; STACHNISS, C.; BURGARD, W. Improved techniques for grid mapping with rao-blackwellized particle filters. *IEEE transactions on Robotics*, IEEE, v. 23, n. 1, p. 34–46, 2007.
- 33 KOHLBRECHER, S. *et al.* A flexible and scalable slam system with full 3d motion estimation. In: IEEE. *Safety, Security, and Rescue Robotics (SSRR), 2011 IEEE International Symposium on.* [S.l.], 2011. p. 155–160.
- 34 LU, F.; MILIOS, E. Globally consistent range scan alignment for environment mapping. *Autonomous robots*, Springer, v. 4, n. 4, p. 333–349, 1997.
- 35 GRISETTI, G. *et al.* A tutorial on graph-based slam. *IEEE Intelligent Transportation Systems Magazine*, IEEE, v. 2, n. 4, p. 31–43, 2010.
- 36 KONOLIGE, K. *et al.* Efficient sparse pose adjustment for 2d mapping. In: IEEE. *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on.* [S.l.], 2010. p. 22–29.
- 37 HESS, W. *et al.* Real-time loop closure in 2d lidar slam. In: *2016 IEEE International Conference on Robotics and Automation (ICRA).* [S.l.: s.n.], 2016. p. 1271–1278.
- 38 Cartographer Documentation. *Cartographer.* [S.l.]: Google, 2018, Revision f73758e3. <<https://google-cartographer.readthedocs.io>>.
- 39 KÜMMERLE, R. *et al.* On measuring the accuracy of slam algorithms. *Autonomous Robots*, Springer, v. 27, n. 4, p. 387, 2009.
- 40 SANTOS, J. M.; PORTUGAL, D.; ROCHA, R. P. An evaluation of 2d slam techniques available in robot operating system. In: IEEE. *Safety, Security, and Rescue Robotics (SSRR), 2013 IEEE International Symposium on.* [S.l.], 2013. p. 1–6.
- 41 BESL, P. J.; MCKAY, N. D. Method for registration of 3-d shapes. In: INTERNATIONAL SOCIETY FOR OPTICS AND PHOTONICS. *Sensor Fusion IV: Control Paradigms and Data Structures.* [S.l.], 1992. v. 1611, p. 586–607.
- 42 FLANNIGAN, C. *icp.* [S.l.]: GitHub, 2019, commit 167cc4a. <<https://github.com/ClayFlannigan/icp>>.
- 43 RODOLA, G. *psutil - Cross-platform lib for process and system monitoring in Python.* [S.l.]: PyPI, 2019, version 5.6.2. <<https://pypi.org/project/psutil/>>.
- 44 ros-perception. *slam_gmapping.* [S.l.]: GitHub, 2018, commit 5a707e0. <https://github.com/ros-perception/slam_gmapping>.
- 45 tu-darmstadt-ros-pkg. *slam_karto.* [S.l.]: GitHub, 2016, commit 5717906. <https://github.com/tu-darmstadt-ros-pkg/hector_slam>.
- 46 ros-perception. *slam_karto.* [S.l.]: GitHub, 2016, commit 5d31abf. <https://github.com/ros-perception/slam_karto>.

47 googlecartographer. *cartographer_ros*. [S.l.]: GitHub, 2017, commit 42d82cb. <https://github.com/googlecartographer/cartographer_ros>.