FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Optimal Wheelchair Multi-LiDAR Placement for Indoor SLAM

Paulo Rúben Alves Silva

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Professor Luís Paulo Gonçalves Reis

Co-Supervisor: Doutor Eurico Farinha Pedrosa

July 26, 2021

Resumo

Com o objectivo de atingir navegação autónoma, têm sido desenvolvidos protótipos de cadeiras de rodas inteligentes que tiram partido de recentes desenvolvimentos no campo da robótica móvel. A implementação deste tipo de navegação, juntamente com métodos alternativos de controlo, como por exemplo, comandos de voz e sinais cerebrais, expande a utilização deste tipo de equipamento a pessoas que não são capazes de operar uma cadeira de rodas manualmente ou através de controladores padrão. Neste contexto, o projecto Intellwheels 2.0 aparece como uma plataforma de desenvolvimento de cadeira de rodas inteligentes com o fim de ser facilmente adaptada a qualquer cadeira de rodas eléctrica comercial e ajudar qualquer pessoa com capacidades cognitivas e/ou locomotoras limitadas, implementando diferentes métodos de controlo e navegação autónoma.

Uma das técnicas adaptadas do campo da robótica móvel está relacionada com o problema SLAM, que é definido pela utilização de sensores como lasers e câmeras, instalados num robô, para realizar um processo de localização e mapeamento simultâneo. Os algoritmos SLAM são, normalmente, baseados em métodos probabilísticos e matemáticos, e distinguem-se entre si não só pelos métodos que utilizam mas também pelo mapa que produzem. A forma como estas soluções são implementadas reflecte-se, normalmente, no desempenho dos seus processos de localização e mapeamento que, por sua vez, se reflecte nas capacidades autónomas de um robô.

Portanto, fazendo parte do projecto Intellwheels 2.0, uma parte desta dissertação foi dedicada a explorar a integração de várias soluções populares de "open-source" 2D-SLAM, num modelo de cadeira de rodas, e a analisá-las sob múltiplos cenários de navegação diferentes. O seu desempenho foi analisado com a utilização de um número de métricas, uma centrada no erro local gerado entre o verdadeiro movimento relativo da cadeira de rodas e o movimento relativo estimado pela solução SLAM e outra baseada no erro entre a verdadeira posição global e a posição global estimada pela SLAM da cadeira de rodas. Além disso, a drenagem computacional de cada uma das soluções, durante cada experiência, foi analisada. A partir dos resultados obtidos, relativamente a esta fase da dissertação, as soluções SLAM baseadas no uso de filtros de partículas ofereceram o melhor desempenho, sob os cenários de navegação em que foram testadas.

A eficiência destas técnicas depende da combinação de robô, ambiente e sensores aos quais é aplicada. Por isso, um aspecto importante é a colocação de sensores no próprio robô. Com isto em mente, na segunda fase desta dissertação, foi desenvolvido um algoritmo de optimização, baseado num algoritmo de optimização "steespest hill-climber". O principal objectivo aqui foi encontrar a posição/configuração, num modelo de cadeira de rodas, de um ou vários sensores tipo LiDAR, a fim de maximizar o desempenho de uma das soluções SLAM utilizadas na fase anterior. Com os resultados obtidos, foi possível concluir sobre uma série de configurações diferentes que alcançam um desempenho óptimo SLAM.

Abstract

In order to achieve autonomous navigation, there have been developed intelligent wheelchairs prototypes that take advantage of recent developments in the mobile robotics field. The implementation of this type of navigation, along with alternative control methods such as, voice commands and brain signals, expands the usage of this equipment to people who are not capable of operating a wheelchair manually or through standard controllers. In this context, the Intellwheels 2.0 project appears as an intelligent wheelchair platform that strives to be capable of being easily adapted to any commercial electric wheelchair, aiding any person with impairing cognitive and or locomotive abilities by implementing different methods of inputs and autonomous navigation.

One of the techniques adapted is related to the SLAM problem, which is defined by the usage of sensors like lasers and cameras, installed in a robot, to perform a process of simultaneous localization and mapping. SLAM algorithms are, normally, based on probabilistic methods and mathematics, and are distinguished between themselves not only by the methods they use but also by the map they produce. The way these solutions are implemented is usually reflected in their localization and mapping performance which, in turn, is reflected on the autonomous capabilities of a robot.

Therefore, being part of the Intellwheels 2.0 project, a portion of this dissertation was dedicated to exploring the integration of a number of popular open-source 2D-SLAM solutions, on a wheelchair model, and analyzing them under multiple different navigation scenarios. Their performance was analysed by way of a number of metrics, one focused on the local error generated between the true relative motion of the wheelchair and the relative motion perceived by the SLAM solution and another based on the error between the true global position and the SLAM perceived global position of the wheelchair. Furthermore, the computational resources consumption of each of the solutions, during each experiment, was analysed. From the results gathered, regarding this phase of the dissertation, the particle-filter based SLAM solutions offered the better performance, under the navigation scenarios they were tested in.

The efficiency of these techniques is dependent on the combination of robot, environment, and sensors to which it is applied to. So, an important aspect is the placement of sensors on the robot itself. With this in mind, in the second phase of this thesis, an optimization algorithm, based on a steepest-ascent hillclimber optimization algorithm, was designed. The main goal here was to find the optimal sensor placement/configuration, on a wheelchair model, of one or multiple LiDAR type sensors in order to maximize the performance of one of the SLAM solutions used in the previous phase. With the results obtained, it was possible to conclude upon a number of different configurations that achieve optimal SLAM performance.

Acknowledgements

I would like to first thank Prof. Luís Paulo Reis for the help given throughout this dissertation and for providing me with the oportunity to work on the Intellwheels 2.0 project.

I would also like to thank Dr. Eurico Pedrosa for the vast knowledge he shared during this dissertation and for going above and beyond in helping me complete it.

Finally, I express immense gratitude to my family and to all my close friends who have provided me with joy and laughter through these years of study in the city of Porto. This past year (COVID) has made it very obvious how much I require these people around me and this work would certainly have not been completed without them, so, again, a deep thank you to all of you.

Paulo Silva

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

Brian W. Kernighan

Contents

List of Figures

List of Tables

Abreviaturas e Símbolos

IW	Intelligent Wheelchair
PW	Power Wheelchair
SLAM	Simultaneous Localization and Mapping
iSAM	Incremental Smoothing and Mapping
RBPF	Rao-Blackwellized Filter
SIS	Sequential Important Sampling
IMU	Inertial Measurement Unit
IEEE	Institute of Electrical and Electronics Engineers
RANSAC	Random Sample Consensus
LIDAR	Light Detection And Ranging
ROS	Robotic Operating System
VO	Visual Odometry
EKF	Extended Kalman Filter
RGB	Red Green Blue
VINS	Visual Inertial Navigation System
RTAB	Real-Time Appearance-Based Mapping
GPS	Global Positioning System
LIAAC	Laboratório de Inteligência Artificial e Ciência de Computadores
FEUP	Faculdade de Engenharia da Universidade do Porto
RD	Research and Development
MR	Mixed Reality
XML	Extensible Markup Language
SDF	Simulation Description Format
URDF	Unified Robot Description Format
YAML	Yet Another Markup Language
API	Application Programming Interface
LaMa	Localization and Mapping
IRIS	Intelligence Robotics and Systems
FOV	Field of View
CPU	Central Processing Unit
RAM	Random Access Memory
PF	Particle Filter
GB	Gigabytes
APP	Application

Chapter 1

Introduction

1.1 Motivation

Currently, it is estimated that about 10% to 15% of the world population (1 billion people), experience some kind of physical disability [9]. In this number, about 10% of all disabled people, experience lower limb dysfunctions and require a wheelchair to move around [10]. These can be caused by factors such as aging, accidents, wars, and various medical conditions and diseases that affect a person's movement such as multiple sclerosis, cerebral palsy, ataxia, dystonia, etc.

Today's society is increasingly worried about developing methods and technologies that return autonomous and independent behavior to disabled and/or elderly citizens and, even though the classic wheelchair and power wheelchairs have partially fulfilled this need, the increasing development of autonomous technologies - in the last decade not only have computers and sensors gotten faster but they have become cheaper and smaller - allowed for the introduction of a more advanced type of wheelchair, the Intelligent Wheelchair (IW).

This equipment should fulfill the following needs: Localization, safe navigation avoiding obstacles, intelligent interface - easy and adaptable to the user - and communication with other devices. These needs can be included in one major domain in the autonomous vehicles industry, specifically, autonomous navigation. Over the last few decades, IW prototypes have managed to implement a number of autonomous navigation capabilities, ranging from simple obstacle avoidance systems to much more complex path planning technologies. The most recent prototypes have made use of simultaneous localization and mapping (SLAM) solutions fully expanding IW's capabilities to full autonomous navigation systems. Like a big chunk of autonomous navigation technologies, SLAM solutions, specifically 2D SLAM, are reliant on data gathered by laser scan rangers to perform their mapping and localization processes, which is itself influenced by both the capabilities and the placement of said sensors. The latter is especially important in robotic systems where the available footprint for placing the sensors is limited by either the robot frame or the robot's purpose.

This dissertation is then focused on studying and comparing different open-source 2D SLAM solutions by using a benchmarking metric that should then be applicable in a standard optimiza-

tion algorithm focused on maximizing the performance of said SLAM solutions, by searching for the optimal placement/configuration of the laser scan rangers that are used, in an IW frame. The framework developed to achieve this is part of the Intellwheels 2.0 project, which is a platform for developing IW prototypes and researching multiple topics surrounding this concept.

1.2 Context

The Intellwheels 2.0 project, an iteration of the previous project, Intellwheels 1.0, has been carried out by researchers from the Universities of Porto and Aveiro, alongside three additional companies with experience in the areas of medical informatics, simulation and serious games, and assistive equipment, with emphasis on wheelchairs. This project will follow its previous iteration by consolidating its respective innovative ideas and developing four fully functional products:

- IW framework/kit, which allows for the transformation of different types of commercial wheelchairs into IWs, with minimum hardware changes, reduced costs, and low visual and ergonomic impact. It will include all hardware and software, specific to the robotics area, to solve the problems of sensing, mapping, localization, control, navigation, and planning on a wheelchair.
- A realistic virtual reality IW simulator, with a 3D interface, that includes three instructional serious games with the intent to train the user on how to control the wheelchair.
- A totally configurable interface that is adaptable to the special needs of the user.
- Complete IW prototypes with the above features included.

1.3 Goals and Contributions

While there has been much research done on comparing different SLAM solutions in different scenarios, there does not seem to be any publicly available investigations regarding the maximization of the performance of said solutions by altering the placement of the sensors used to perform SLAM. With this in mind, this dissertation strives to offer a step in the exploration of this topic, with a few goals being set:

- Gather knowledge of the SLAM concept, the different types of solutions, and the problems associated with it, along with a general study on the different IW prototypes that employ this concept.
- Develop a simulation environment that includes:
  - Multiple SLAM solutions integrated into the previously developed wheelchair model and navigation stack.

  - An odometry noise model that adds sufficient noise to distinguish the performance between the different SLAM solutions.
  - Distinct indoor environments that are identifiable by the robotics community.
  - A benchmarking metric capable of accurately representing SLAM performance.
  - A steepest-ascent hillclimber adapted to the simulation environment created that tries to maximize SLAM performance by changing laser scan rangers configurations and/or placement around the wheelchair model.

- Compare the 2D SLAM solutions in the indoor environments developed using the benchmarking tool.

- Apply the optimization algorithm considering different variables regarding the placement/configurations of the sensor(s).

1.4 Document Structure

The first chapter corresponds to chapter 2, and it comprises of the description of the navigation problem that will be tackled in this dissertation, namely, what actually defines this problem both theoretically and mathematically, and why it is important to solve the problem. It contains some examples of the state-of-the-art solutions to the SLAM problem and a short taxonomy of the issues that still remain unsolved. Additionally, research on optimal sensor placement techniques, applied in similar fields, is offered, specifically, the application and adaptation of optimization algorithms. The following chapter, chapter 3, offers a definition of the intelligent wheelchair concept along with a description of state-of-the-art prototypes developed in the market, focusing on the ones that have employed SLAM algorithms. A detailed description of the project Intellwheels is also included. In chapter 4 the simulation enviroment developed is described in detail while in chapter 5 the results of the experiments made are displayed and discussed.

Finally, chapter 6 states the conclusions and contributions made during the development of the dissertation and offers a quick view into the possible future work and improvements on the research conducted.

Chapter 2

Simultaneous Localization and Mapping

In the modern era, intelligent mobile robots are often used to autonomously achieve various tasks like material transportation, surveillance duties, space exploration, etc. To accomplish this, a robot must be able to freely navigate dynamic and/or static environments.

For a robot to correctly perform this navigation, it must be able to localize itself in an environment. Consequently, to achieve localization, a map of the environment must be provided. If the robot has access to a prior map, the localization problem, usually, becomes much simpler by taking advantage of commonly used sensors such as wheel encoders and IR sensors, to keep track of its movement and compare its visual measurements to the pre-existing map. However, if no prior map is provided, a robot must then be able to explore the environment and execute a process of mapping, using visual sensors (sonar sensors, LIDAR sensors, cameras). The issue here is that, for a robot to perform mapping it needs to know its position, and for a robot to perform localization it needs access to a map. In the robotics community, this problem is referenced to as Simultaneous Localization and Mapping or, SLAM. As the name suggests, it deals with methods that are used in the autonomous robots industry that allow them to perform mapping while simultaneously localizing themselves within said map.

2.1 SLAM and Applications

Localization is one of the main pillars supporting robot navigation. Robot localization often involves identifying the absolute position of a robot with respect to its target environment. The environment highly conditions the localization methods that are able to be used. If one could attach an accurate GPS (global positioning system) sensor to a robot, most of the localization problem would be solved. However, as we know, modern GPSs can only be used outdoors and, even when used outside, they still may have errors in the range of several meters. To mitigate this, a robot is often fitted with sensors that offer it some kind of perception of the world and of the state of the robot. Sensors like wheel encoders, IMUs, and compasses supply the robot with information

about its movement while laser range finders, cameras, and other visual sensors, offer insight into the surrounding environment.

Typically, if a map is provided, the robot can estimate its position, in reference to the map, by using information about its own movement and then performing corrections on this positioning by comparing its perception with the given map (figure 2.1) This is typically done with methods such as landmark Kalman Filter Localization or Monte Carlo Localization [1].

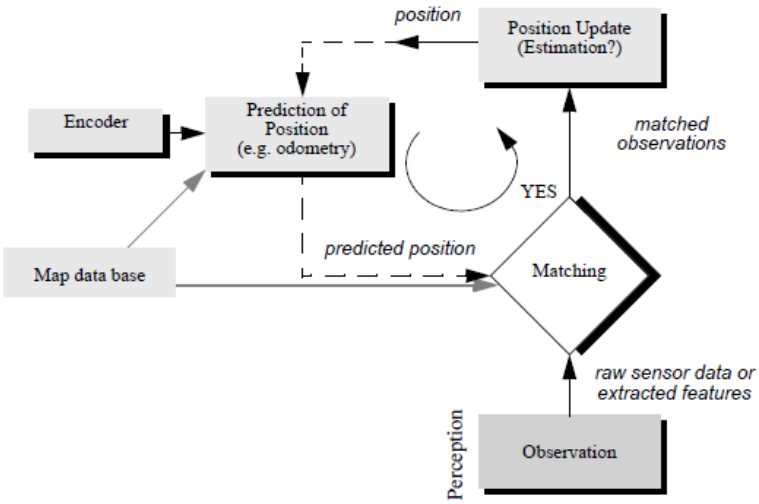


Figure 2.1: Common robot localization approach [1]

However, if no prior map is provided the robot must be able to build one. This process is called mapping. Mapping is a discipline related to computer vision and it stands for the ability to construct a map using the robot's perception of a given environment. Mapping can either be done in 2D using, for example, an IR sensor or, in 3D, using cameras or 3D IR sensors. Additionally, there are several ways to represent this map, which are further discussed in the following sections. This mapping cannot be separated from the localization process. If we can't locate our robot while mapping is being performed, it would be impossible to correlate different observations made at different locations.

These two concepts, localization and mapping, are deeply connected, which is why SLAM is such a complicated problem to solve.

Finding a solution to the SLAM problem is an important basis for extending the variety of robotic applications. Currently, mobile robots are already being used in many different situations, respectively:

- At home: vacuum cleaners, lawnmower;
- Airborne: Surveillance with aerial vehicles;
- Underwater: Reef monitoring;

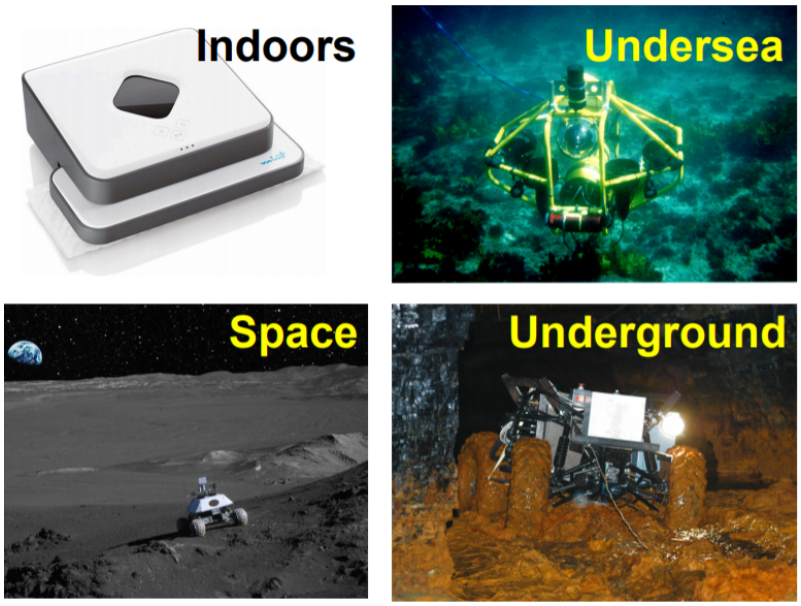


Figure 2.2: SLAM application Examples [2]

- Underground: Mine exploration;
- Space: Terrain mapping and exploration;

As a very simple example, we can look at early models of autonomous vacuum cleaners. They would only employ sensors to avoid obstacles or inaccessible areas like stairs while moving through an area at random, within a certain amount of time, in hopes of guaranteeing they have cleaned the whole area they've been assigned to [11]. While this is a solution for this problem, it is very limited in its success as a lot of power is wasted since the robot would often end up visiting already cleaned areas due to the inability to track its own position. Currently, modern models are now able to employ SLAM technologies giving the robot the ability to map rooms and keep track of its position in said rooms making them more efficient and truly making them autonomous in the completion of their task [12].

Through this example it is possible to see why the solution to the SLAM problem is seen in the mobile robotics community as the "holy grail" [3] and, while it has been solved in a vast number of different set environments, there still remain many questions, such as how to develop a more general SLAM solution and also in how to build detailed and semantic rich maps.

2.2 Mathematical Basis

Usually, in a SLAM problem, we have a robot that is equipped with a combination of proprioceptive (visual, laser, etc.) and exteroceptive sensors (wheel encoders, IMUs, etc) and, using these, we try to estimate the robot's pose (position and orientation) and build a representation of its surrounding environment [3]. Let's consider a mobile robot, equipped with a laser sensor and

wheel encoders, moving through an environment and taking measures of surrounding landmarks - typically walls and other day-to-day objects. In this situation, we want to know the position of our robot (equation 2.1) and the map of the surrounding environment (equation 2.2), given observations made by our IR sensor (equation 2.3) and the movement of our robot returned by the wheel encoders (equation 2.4).

Mathematically, we can define the following sets that include, until time constant t, the vector pose of the robot, x_t, the vector control actions applied to the robot, u_t, the vector containing locations of the ith landmarks, m_i, and, finally, the vector containing the observations of the ith landmark at time t, z_t.

Wanted

- Path of the robot

$$X_{0:t} = \{x_0, x_1, ..., x_t\} = \{X_{0:t-1}, X_t\} \tag{2.1}$$

- Map of the landmarks

$$m = \{m_1, m_2, ..., m_n\} \tag{2.2}$$

Given

- Landmark observations

$$Z_{0:t} = \{z_0, z_1, ..., z_t\} = \{Z_{0:t-1}, Z_t\} \tag{2.3}$$

- Robot's control inputs

$$U_{0:t} = \{u_0, u_1, ..., u_t\} = \{U_{0:t-1}, U_t\} \tag{2.4}$$

Most onboard sensors used in the mobile robotics industry have some kind of accumulative error associated with their measurements so we must not assume we know the exact position of the elements and, instead, make an estimation of these. With these errors, uncertainty has to be associated with every measurement and every robot position. To represent this, in figure 2.3 we can see that every robot position (triangle) x and landmark position (star) m is represented by two figures. To name a few of these error sources, they can be caused by wheel slippage and poor calibration of kinematic models due to inaccurate measurement of the robot's characteristics (distance between wheels, the diameter of wheels, etc) affecting the measurements taken by encoders [13], noise populated readings in lasers [14], camera lens distortion in visual sensors [15] and many others. Therefore, to perform these estimations we usually employ a number of probabilistic methods.

Probabilistic Framework

First introduced in the 1986 IEEE Robotics and Automation Conference, the description of the SLAM problem with probabilistic tools is used in most types of SLAM algorithm developed.

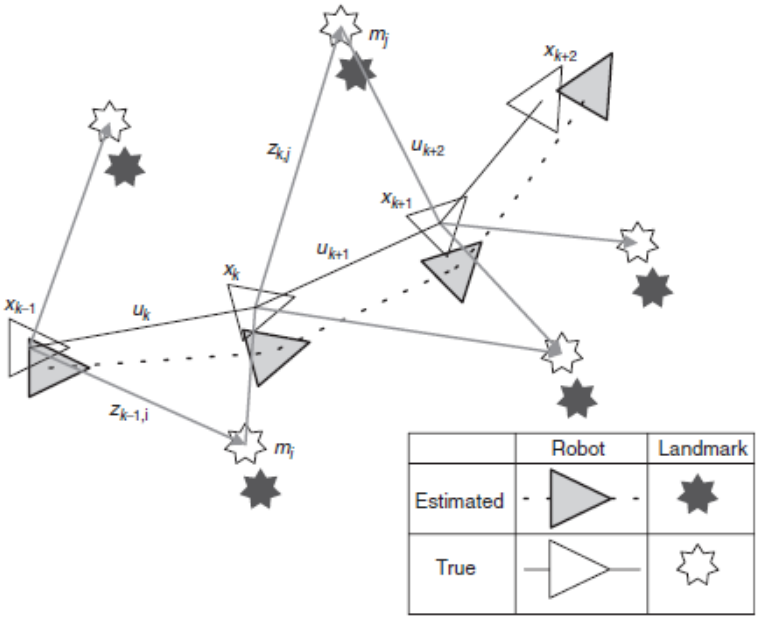


Figure 2.3: Essential SLAM problem as described in [3]

In the probabilistic form, solving SLAM now consists of estimating the posterior probability of the state of the robot x and the map of the landmarks m given all the sensor measurements $z_{0:t}$, all the control inputs $u_{0:t}$ and the initial position of the robot, x_0 at every time constant t.

$$p(x, m|z_{0:t}, u_{0:t}, x_0) \tag{2.5}$$

Note here that no mention was made of when the robot pose is calculated. This is because we must first distinguish between two types of SLAM, full SLAM, and online SLAM. Full SLAM computes the whole trajectory of the robot, $x_{0:t}$, along with the map of the environment.

$$p(x_{0:t}, m|z_{0:t}, u_{0:t}, x_0) \tag{2.6}$$

On the other hand, online SLAM is a subset of the above solution as it only computes the posterior of the robot's current pose along with the map.

$$p(x_t, m|z_{0:t}, u_{0:t}, x_0) \tag{2.7}$$

This is done by recursively integrating, $tth-1$ number of times, the data from the full SLAM approach, which allows the discarding of past pose information of the robot.

$$p(x_t, m|z_{0:t}, u_{0:t}, x_0) = \int\int \ldots \int p(x_{0:t}, m|z_{0:t}, u_{0:t}, x_0) dx_0, dx_1 \ldots dx_{t-1} \tag{2.8}$$

In any case, to compute these probabilities we need to define the motion and observation

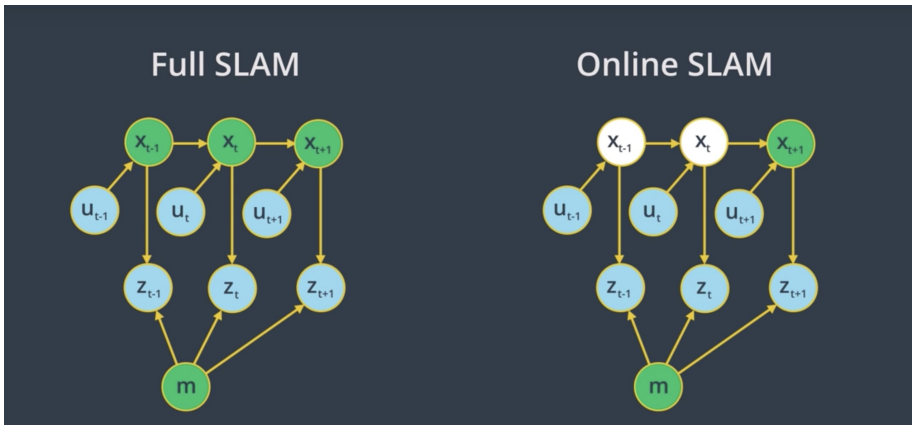


Figure 2.4: Full SLAM vs Online SLAM

models of our robot of choice since both of these depend on its on-board sensors. The motion model of a robot is the probability distribution of the current position x_t given the past position x_{t-1} and the control input u_t.

$$p(x_t|x_{t-1},u_t) \tag{2.9}$$

Similarly, the observation model is the probability distribution of making an observation z_t given the current robot position x_t and the state of the map m.

$$p(z_t|x_t,m) \tag{2.10}$$

For a more detailed explanation of how these probabilistic methods work please refer to [16].

2.3 Common SLAM approaches

Other than the distinction made on 2.2 between full SLAM and online SLAM there are many subsets of these approaches to the SLAM problem. Usually, on a general level, we can differentiate between SLAM algorithms by the way that they solve the posterior probabilistic problem but also by how they represent the solution, specifically, by how the map is built and presented. As theorized in [17], the metric representation that is chosen for SLAM is very important as it impacts many other surrounding aspects such as long-term navigation, physical interaction with the environment, and human-robot interaction.

Generally, mapping in the SLAM context is either done in 2D or 3D. The most prominent representations are *feature-based maps* and *occupancy grids* and they are also by far the most mature representations having recently been released a standardization for these two by the *IEEE RAS Map Data Representation Working Group*. Aside from them, *graph-based* portrayals, *topological mapping* and *semantic mapping* are types of mapping still in their infancy, in terms of research, but are quickly becoming state-of-the-art solutions. The concept of mapping is later discussed in section 2.4.3.

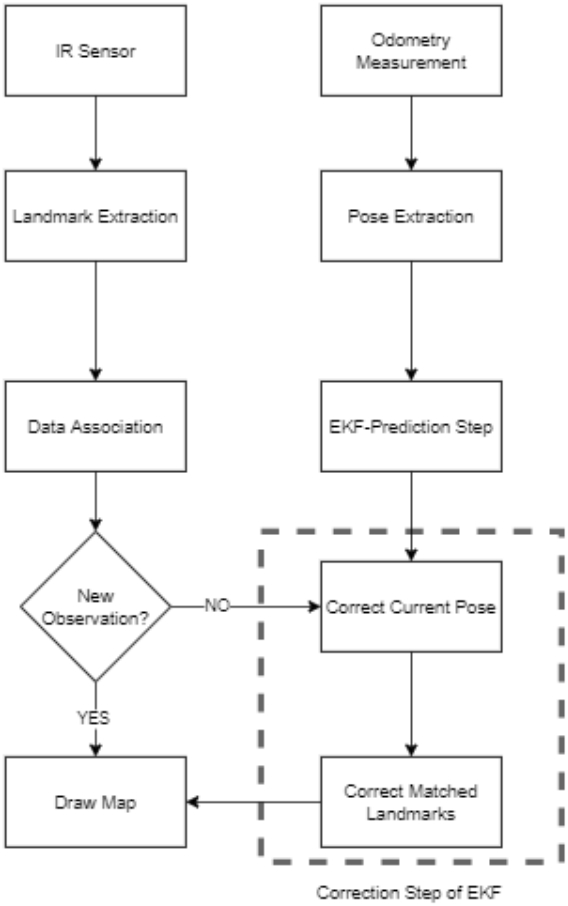


Figure 2.5: EKF-SLAM flowchart

2.3.1 Feature-Based SLAM

Feature-based SLAM or, as it also is recognized, *landmark-based mapping*, is by far the most widespread approach to mapping in the mobile robotics industry, and it is based on using sensor measurements to identify landmarks and build a map with the location of said landmarks.

2.3.1.1 EKF-SLAM

As mentioned in 2.2 there is always an error associated with the measurements taken by our robot's sensors. In a situation where we have access to our robot's odometry along with sensor measurements of the surrounding environment, we may implement an algorithm that recursively makes corrections to the robot's state space based on comparing the measurements by these two types of sensors. In this context, the usage of the Extended Kalman Filter in a SLAM system, is one of the most common solutions and it works by implementing a recursive process of prediction (robot moves) and correction (robot measures). In figure 2.5 we have a simple flowchart describing the whole process.

Whenever we receive a measurement from the odometry sensors we make a prediction about our state space, as per equations 2.11 and 2.12, where $f(x_{t-1}, u_t)$ and $h(x_t, m)$, stand as our motion model and observation model, respectively, and w_t and v_t are our, previously mentioned, additive errors which, in this context, are represented by zero-mean Gaussian noises related to each sensor.

$$P(x_t|x_{t-1}, u_t) \iff x_t = f(x_{t-1}, u_t) + w_t \tag{2.11}$$

$$P(z_t|x_t, m) \iff z_t = h(x_t, m) + v_t \tag{2.12}$$

On the other hand, whenever we receive measurements about the environment (in the case of this flowchart, from an IR sensor), we check to see if the landmark measurement has already been mapped or not. If it hasn't, we use this observation to update the map, but if it has, we use it to correct our current state space of the robot. This step of checking if the surrounding environment has been mapped or not is called, *Data Association* and stands as one of the most important phases in any SLAM algorithm, later described in section 2.4.1.1.

Overall, the EKF-SLAM solution has been deeply studied and applied resulting in similar benefits and problems as the standard EKF solutions to navigation. Since EKF-SLAM employs linearized models of nonlinear motions and observation models, this solution can still lead to unexpected results. On top of that, the EFK correction step requires that all landmarks be updated whenever there is a new observation. As expected, this leads to a massive computational requirements growth as the landmarks keep increasing in number, having already been a lot of effort put in to develop efficient variants that avoid this problem [18]. Finally, this solution typically has problems with incorrect association of observations to already existing landmarks. EKF-SLAM is especially vulnerable to this problem since it employs *Stochastic Mapping*, which provides metrically accurate navigation but, since it incorporates no other information other than distance measurements, *data association* becomes particularly hard [19].

2.3.2 Grid-Based SLAM

As the name suggests, grid-based SLAM divides the created map into small individual areas, usually squares. These squares can either be filled, empty, or partially filled, representing the state of occupancy of the grid in that location. Comparatively, grid-based algorithms tend to require a higher computational effort than the landmark-based approaches mentioned in the previous section, however, they are able to represent arbitrary locations and provide detailed representations, while feature-based mapping methods are preferred because of their lower-resource requirements [20]. One of the most common algorithms developed using Grid-Based SLAM is GMapping and it uses a concept known as *particle filtering*.

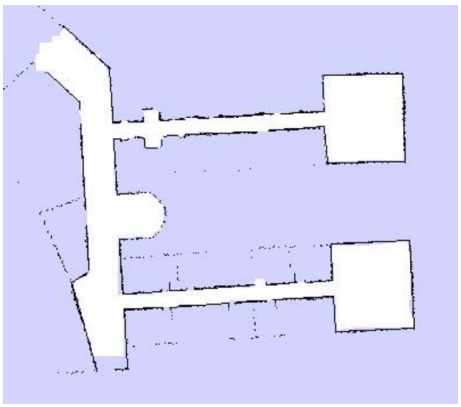


Figure 2.6: Grid-based SLAM mapping

2.3.2.1 GMappping

Instead of focusing on improving the performance of EKF, GMapping takes a different approach to the recursive probabilistic problem. The main basis of the technology is particle filtering, and it consists of the estimation of internal states in dynamic systems from observations made, considering random perturbations present in sensors and in the environment itself. The advantage here is that these perturbations are not considered to be exclusively Gaussian.

GMapping uses an improved adaptation of the standard particle filter, the *Rao-Blackwellized Filter* (RBPF) [21]. The usage of RBPF allows for the splitting of the aforementioned joint state posterior probability into two separate posterior probabilities, one regarding the path of the robot and another regarding the map, as seen in equation 2.13.

$$p(x_{0:t}, m|z_{0:t}, u_{0:t}) = p(x_{0:t}|z_{0:t}, u_{0:t})p(m|z_{0:t}, u_{0:t}) \tag{2.13}$$

Thus, the essential structure of GMapping, is an RBPF state, where the trajectory of the robot is represented by weighted particles and the map is computed analytically. Our joint state then becomes represented by the following set:

$$\{w_t^{(i)}, x_{0:t}^{(i)}, P(m|x_{0:t}^{(i)}, z_{0:t}^{(i)}\}_i^N \tag{2.14}$$

Each particle has its own map - in this case, a grid-based map - predicted using the robot's measurements $z_{0:t}$ and the trajectory $x_{0:t}$ corresponding to each particle. The actual particle filtering algorithm is based on the popular *Sequential Important Sampling* (SIS) [22]. At each time step t, particles are drawn from a proposal distribution $\pi(x_t|x_{0:t-1}, z_{0:t})$ which is an approximation of the actual distribution $P(x_t|x_{0:t-1}, z_{0:T})$. The samples are then given importance weights according to the error between these two distributions. This error increases over time which will increase the variation between existing sample weights, where the particles that have the least discrepancy will have the higher the weights and vice-versa. A resampling step then occurs reinstating the uniform sampling.

The general form of an RBPF particle for SLAM is divided into 4 steps and works as follows [3]. Assuming that at a time $t-1$ our joint state is represented by $\{w_{t-1}^{(i)}, x_{0:t-1}^{(i)}, P(m|x_{0:t-1}^{(i)}, z_{0:t-1}^{(i)}\}_i^N$.

1. *Sampling*: Obtain our set of particles $\{x_t\}^i$ by sampling from the distribution:

$$x_t^{(i)} \sim \pi(x_t|x_{0:t-1}^{(i)}, z_{0:t}, u_t) \tag{2.15}$$

2. *Importance Weighting*: Weight samples according to the importance function:

$$w_t^{(i)} = w_{t-1}^{(i)} \frac{P(z_t|x_{0:t}^{(i)}, z_{0:t-1})P(x_t^{(i)}|x_{t-1}^{(i)}, u_t)}{\pi(x_t|x_{0:t-1}^{(i)}, z_{0:t}, u_t)} \tag{2.16}$$

   where the numerator is both the observation and motion model, respectively.

3. *Resampling*: Done by selecting particles, with replacement, from the set $\{x^{(i)}, w^{(i)}\}_{i=0,\ldots,N}$, including their associated maps, where the probability of each particle being picked is proportional to the their weight $w_t^{(i)}$. The selected take on a uniform weight, $w_t^{(i)} = 1/N$.

4. *Map Estimation:* For each particle $x_t^{(i)}$, perform an update of the map $m_t^{(i)}$ according to $p(m_t^{(i)}|x_{0:t}^{(i)}, z_{0:t})$.

It should be noted that when to perform the resampling step is still an open problem, not only for this algorithm but for most others that use the RBPF filter. Some implementations of GMapping resample at every time-step, others after a fixed number of time-steps, and others once their weight variance exceeds a threshold. We can see that each variation looks to decrease the number of resampling steps which consequently reduces the computational effort of the algorithm. Another big problem for particle filtering based SLAM encompasses the number of particles that are used in each sampling cycle. Typically, A high number of particles will result in better performance, however, it will also lead to higher computational requirements. To diminish this effect, GMapping and many other RBPF algorithms, introduce an additional action before the filtering process called scan-matching, which is a method that can efficiently estimate the rigid transformation of a robot between two poses [23]. This allows the algorithm to take into account the last reading when creating a new particle, and to have a much more precise estimate of the evolution of the system decreasing the number of particles created.

2.3.3 Graph-Based SLAM

This method revolves around building a graph whose nodes represent the robot's poses and the landmarks observed. These nodes are then connected through constraints that are either sensor measurements or odometry commands. However, as discussed before, these measurements are often populated with errors so, once a final graph is built, we try to find the best configuration of nodes that is consistent with the measurements taken, which involves solving a complex error minimization problem [24]. Thus, graph-based SLAM can be divided into two tasks:

1. *Graph-Construction:* Building the graph from the raw sensor measurements;
2. *Graph-Optimization:* Determining the most likely configuration of the poses given the edges of the graph.

This approach is actually quite old, having been introduced in 1997 [25]. Despite this, it took several years for this formulation to become popular due to the high complexity of the minimization problem that this solution is based on. With recent developments in the sparse linear algebra field, the problem at hand has been optimized to the point where graph-based SLAM methods have undergone a renaissance, and currently belong to the state-of-the-art techniques with respect to speed and accuracy.

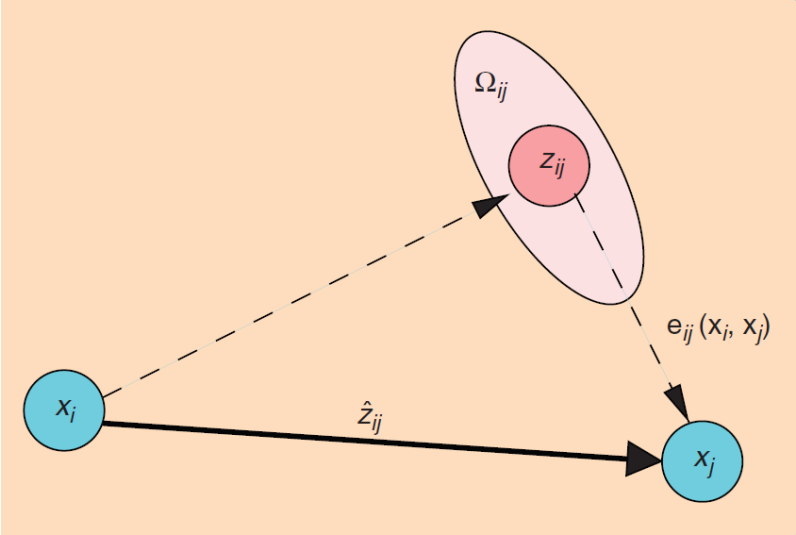


Figure 2.7: Graph-based SLAM

Let $x = (x_1, ..., x_t)^{(t)}$ be a vector of our robot poses and z_{ij} and Ω_{ij} be our mean and information matrix of our virtual measurements, computed by our observation model $p(z_t, x_t, m)$, between node i and j, respectively. These virtual measurements are seen as a possible transformations that makes the observation acquired at time i maximally overlap with the observation at time j. Additionally, let $\bar{z}(x_i, x_j)$ be the prediction of the virtual measurement given the nodes x_i and x_j, the log likelihood l_{ij} of a measurement z_{ij} will be:

$$l_{ij} \propto [z_{ij} - \bar{z}(x_i, x_j)]^t \Omega_{ij} [z_{ij} - \bar{z}_{ij}(x_i, x_j)] \tag{2.17}$$

With this in mind, $e(x_i, x_j, z_{ij})$ will be the function that computes the error between the expected observation $\bar{z_{ij}}$ and the real observation z_{ij} gathered by the robot. Finally, the goal of this SLAM approach is to find the configuration of nodes x^* that minimizes the negative log-likelihood $F(x)$ of all the observations.

$$F(x) = \sum_{(i,j)} e_{ij}^t \Omega_{ij} e_{ij} \tag{2.18}$$

Therefore, we are looking to compute the following equation:

$$x^* = \arg\min_x F(x) \tag{2.19}$$

What differentiates many of the graph-based SLAM algorithms is the technique used to minimize the function described in equation 2.18. As an example, we have GraphSLAM [26] that applies variable elimination techniques to reduce the dimension of the problem and the work done in iSAM [27], that presents an online version of this method, exploiting partial reorderings to compute the sparse factorization.

2.4 Open Problems

The question, "is SLAM solved?" is often asked within the robotics community. This is quite a difficult question to answer as SLAM has become a very broad topic and it is used in a great number of different robot/environment/performance combinations and while, for some of these, there have been developed quite satisfactory solutions, there still remain many combinations that require a substantial amount of research.

For instance, as mentioned in section 2.1, the problem regarding a small vision-based slow-moving robot in a home environment has largely been solved. Another example of a solved problem is NASA's Mars Rovers, which are, used outside of a home environment. On the other hand, when we flip this combination, namely, a fast vision-based robot in a highly dynamic environment, current SLAM algorithms still lead to undesirable results. In this section, we will distinguish some of the issues that are still prevalent in the SLAM world, namely, open problems dealing with the *Robustness*, *Scalability*, and *Map Representation* of modern SLAM technologies.

2.4.1 Robustness

There are two main aspects that contribute to the fragility of a SLAM system, particularly, the hardware associated with it or the algorithm used. The first usually comes from inaccurate sensor measurements and hardware degradation, while, the second, is an outcome of limitations in the mathematical methods used to design the algorithm. Addressing these issues is key to designing a long-term robust SLAM system that no longer needs to make strict assumptions about the environment and can fully rely on its on-board sensor capabilities.

2.4.1.1 Data Association

One of the main sources of algorithmic failures is data association. As mentioned in section 2.3.1.1, this is a prevalent phase in all SLAM algorithms, where sensor measurements are matched with previous measurements taken. The presence of unmodeled dynamics in an environment, say, different weather conditions, and also the false assumption that the world is always static makes data association especially hard, as these variables often lead to incorrect associations or the rejection of accurate measurements.

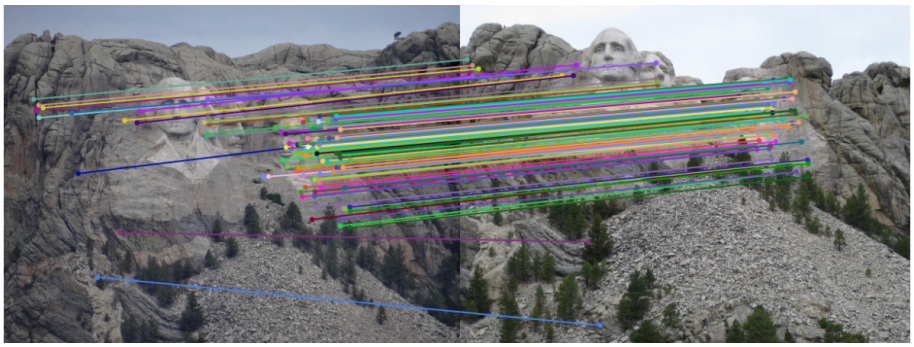


Figure 2.8: 3D points matching

Data association from a short-term perspective is the easiest to tackle. This is the case for visual odometry, where a robot estimates its velocity through the association of data (pictures) acquired from a visual sensor in a short time frame. If the camera has a high enough frame-rate compared to the dynamics of the robot we can easily track corresponding features between frames [28] resulting in reliable tracking. Unfortunately, data association in a long-term scenario is much more challenging, involving the process of loop closure which in itself is divided into two main phases, *detection* and *validation*. Regarding detection, previous methods of detecting features in current data and simply matching them with previous features have become quite inefficient. Currently, Bag-of-Words models [29] are used to avoid this previous unruliness as they enable the quantification of feature spaces which provides more efficient searches. Beyond this, these models have been further optimized [30, 31] to be able to deal with environmental variations such as illumination. The loop closure validation phase consists of supplementary measures that are taken to guarantee the quality of the loop closure. Again, considering visual-based systems, RANSAC algorithms are commonly used for geometric verification and outlier rejection [28].

Despite this effort, it is still unavoidable that wrong loop closures occur. This can severely compromise our map estimate and, in order to deal with this problem, recent lines of research have proposed various techniques [32, 33]. Other than data association there still remain a few open problems requiring further research [17]:

Failsafe SLAM and recovery: Because most SLAM techniques are based on iterative optimization two main issues are established. First, the outlier rejection outcome is highly reliant on the initial guess provided to the iterative process and second, the inclusion of a single outlier heavily degrades the estimation capability of a SLAM algorithm. Ideally, our system should be aware that these failures might occur and be able to recover from the inclusion of these outliers. Currently, there are no SLAM implementations that deal with this.

Hardware failure detection: The failure and degradation of our hardware due to aging, malfunctions, and malpractice is undoubtedly unavoidable and results in inaccurate measurements which obviously affect our SLAM performance. Naturally, questions of how our system can autonomously detect this degradation are crucial in the development of long-term reliable SLAM technologies.

Deformable maps: As mentioned before, most SLAM methods work under the assumption that the environment surrounding the system is static and rigid. However, as we know, the real world is non-rigid and dynamic. An ideal SLAM solution should be able to identify these dynamics and be able to generate "all-terrain" maps in real-time.

Automatic Parameter Tuning: This is a crucial step for a SLAM system to be able to adapt to arbitrary scenarios since most of these require extensive parameter tuning to achieve satisfiable performance in a given scenario. To achieve true autonomous behavior, the SLAM system should be able to perform this tuning automatically.

2.4.2 Scalability

As mentioned before, the SLAM problem has been successfully solved for most indoor applications, however, in many other endeavors, robots must be able to operate for a prolonged period of time over areas of a larger scale. Accordingly, the information gathered by our SLAM system. represented, in the case of a graph-based SLAM system explained in section 2.3.3, by the number of nodes and edges, can grow unbounded due to the long-term exploration cycles associated with these prolonged scenarios. Since the computational time and memory available is bounded to our robot's resources, it's important to design SLAM methods that also fixate the computational and memory requirements within the bounds set by the hardware being used.

The most recent methods aimed at reducing the scope of this problem, still with a factor-graph based approach in mind, focus on reducing the complexity of factor graph optimization using *sparsification methods* that are mostly focused on decreasing the number of nodes and edges added to the system. Other more non-conventional methods like *parallel SLAM* split the computation and memory effort among multiple processors. These approaches are often mentioned as *sub mapping algorithms*. An example of this is [34], which proposes a hierarchy of sub-maps. Specifically, whenever an observation is acquired at the highest level then only the sub-maps that are affected by this change are altered. Another solution to mapping a large scale environment is to use multiple robots doing SLAM, dividing the scenario into smaller areas, each one of them being mapped by a different robot.

Despite these efforts to reduce the complexity of SLAM systems, there still remain some unexplored areas when it comes to this issue [17], namely:

Map Representation: How to efficiently store a map, even when there no are no memory constraints, for example, when data is stored in the cloud, is still a vastly unexplored concept. In this vein, another idea that remains uncharted is when and how to decide to erase data that is no longer representative of the environment in order to decrease memory requirements.

Resource Constraints: Another problem to consider is how to adapt existing SLAM technologies to hardware that is limited in its computational and memory resources.

Distributed mapping: This last scalability related problem is applied to the case of when we are using multiple robots employing SLAM. While approaches for outlier rejection, as mentioned in the previous section, have been proposed for the case of a single robot, there has not been much research tackling a system composed of multiple robots.

2.4.3 Map Representation

As said in section 2.3, there are many ways to model the geometry of the environment built by a SLAM algorithm and since this mapping often has effects on other robotic fields, it's important to discuss the many different approaches that exist and the problems still associated with them and this concept in general.

2.4.4 Metric Map Models

In section 2.3.1 it was already mentioned one of the most popular types of mapping, *landmark-based mapping*. The biggest issue with this type of mapping is that it assumes that all the landmarks are distinguishable from each other which is not always the case. Contrary to landmark-based representations, *dense representations*, attempt to provide high-resolution models of the environment in 3D, through raw representations of 3D data gathered by cameras, namely, point clouds. In comparison with the previous mapping, this depiction is better suited for obstacle avoidance. Furthermore, we have *spatial dense representations* extending the usage of point clouds to actually represent surfaces and volumes of the objects in the environment through boundary representations that define 3D objects in regards to their surface boundary [35, 36].

As per identified in [17] there are areas that remain unexplored regarding metric models, specifically:

High Level, expressive representations: The previous point cloud-based models mentioned are very wasteful regarding memory conservation and, while they provide an accurate model of the surrounding environment, they offer no high-level understanding of the environment, that is, they make no distinction between types of environments say a room versus a corridor. This type of knowledge would lead to many advantages as it would allow for easier loop closure and data association, new tools for compact representations, and integration with existing map tools.

Optimal and Automatic Representation: Similar to the concept of automating tuning told in section 2.4.1, developing a method to autonomously decide the better representation to use considering the properties of the surrounding space would be highly beneficial.

2.4.5 Semantic Map Models

To mitigate some of the shortcomings that metric map models have, semantic map models, as the name suggests, work by associating semantic concepts to geometric entities in a robot's surroundings, for example. being able to identify a muddy road and giving the robot the ability to understand that crossing that road would affect its movement. The complexity associated with this type of mapping is directly dependent on the number of concepts we want to be able to identify and also the relationship between these (see figure 2.9).

Currently, there are 3 ways that this type of mapping has been implemented [17], namely, by simply building a map through SLAM, analyzing it, and associating semantic concepts to the map, for example, by running an offline classification algorithm on the metric map model [37]. Another way researched flips this previous idea and uses already existing semantic classes to

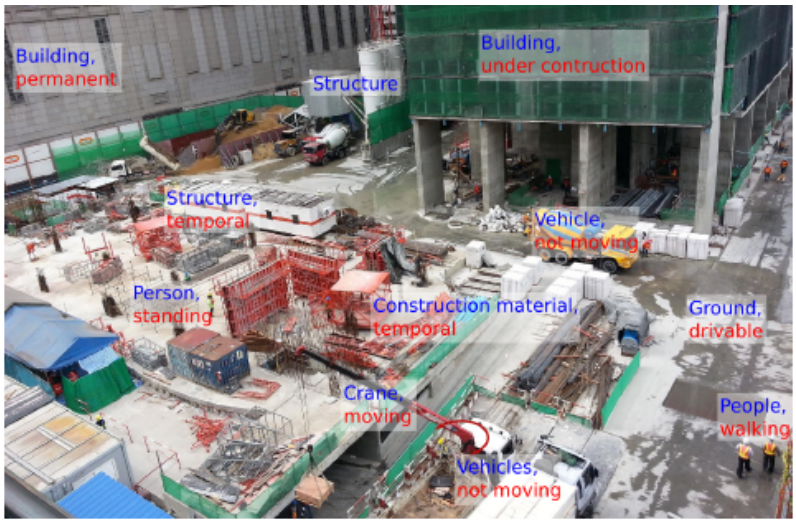


Figure 2.9: Semantic concepts examples

identify objects with prior knowledge of their geometry, during the actual mapping process [35]. Finally, we can actually use semantic inference during the mapping process by taking advantage of state-of-the-art object recognition algorithms used on cameras.

Since this type of mapping is still in its infancy not only development wise but research-wise, some open problems moving forward have been identified [17]:

Consistent semantic fusion: Incorporating semantic information with the metric information coming from measurements taken at differing time frames presents a big issue.

Adaption: The system should be able to not only work with previous existing semantic classes, but it should also be capable of building new semantic classes.

Semantic Based Reasoning: As humans we can use our understanding of semantic relationships to speed-up our reasoning and, similarly, introducing this ability into a SLAM system would also improve its mapping capabilities.

2.5 Optimal Sensor Placement

The performance of a SLAM algorithm is highly dependent on the perception of our environment, as the quality of the robot's visual coverage will reflect on the quality of the localization and mapping processes. This is influenced by the location and orientation, in respect to its robot model, of the visual sensors that are used to obtain the measurements that are then computed by the SLAM solutions. In turn, this will impact the efficiency of our robot's navigation. Dynamic environments are often populated with challenging geometry and visual occlusions, so it is essential to have a sensor network that is placed accordingly in order to maximize the performance and efficiency of the SLAM algorithm being used.

Some research has been done on developing algorithms that deal with the maximization of quality of visual coverage by simply altering the position of onboard sensors. In the self-driving automobile industry, sensors are used to implement advanced driver assistance systems

like adaptive cruise control and automatic parking. To improve these abilities, research has been done regarding the placement of the sensors used.

In the Colorado State University, researchers proposed the framework VESPA [4], which is able to explore the design space of sensor placement locations and orientations to find the optimal sensor configuration for 2 different vehicles. By defining a few metrics representative of the sensor's performance, for example, the position error of the automobile when performing automatic parking, a cost function was created that included these metrics. Then, several state space search algorithms were used to minimize this cost function, essentially turning the sensor placement optimization into a local minima problem. When comparing the results of this framework with a sensor configuration designed by a vehicle expert, the performance of the sensors was much more effective when using the VESPA framework.

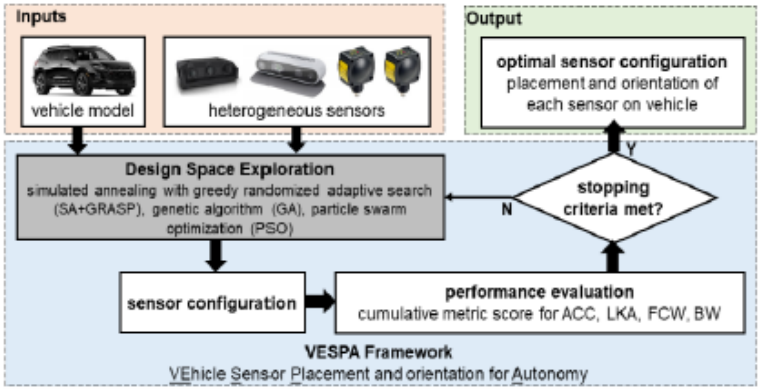


Figure 2.10: Overview of VESPA framework [4]

Similarly in [38], the optimal placement of 3 3D LIDAR sensors was found by the usage of a concept named LIDAR occupancy grid. This grid was used to represent the area that each LIDAR sensor was able to cover. The combination of the LIDAR occupancy grids of the 3 sensors quantifies the full perception that the vehicle has of the environment. In this way, the sensor optimization problem was turned into the maximization of the area that is covered by these sensors. To apply this maximization, a recursive method, based on the Genetic algorithm, also a state-space search algorithm, was successfully developed.

Interestingly, there has not been much, if any, at least publicly available, research on optimal sensor placement for SLAM systems. Since this is the case, this dissertation will look to apply these same concepts, specifically, turning sensor placement into an optimization problem with the objective of minimizing/maximizing a cost function that is reflective of SLAM performance and efficiency, by making alterations to the used sensors' configurations.

2.5.1 Optimization Problems

An optimization problem is nothing more than looking for the maximum/minimum value of a function. Say, a classical example of this is the traveling salesman problem. Imagine we have

a delivery company with a single delivery truck that needs to complete a fixed number of deliveries scattered around different cities. A typical optimization problem would be to minimize the delivery time by exploring the different combinations of cities we can visit until we find the one that minimizes said delivery time and, in turn, maximizes the delivery efficiency of the delivery company. In figure 2.11 we can see the representation of 2 possible solutions to a classic traveling salesman problem. The nodes represent the cities that must be completed while the arrows between the nodes represent the order in which the cities were visited with their respective time of travel. Usually, there are also a number of constraints that need to be considered. In this example, these could be some sort of fuel consumption limit, velocity limit, etc.

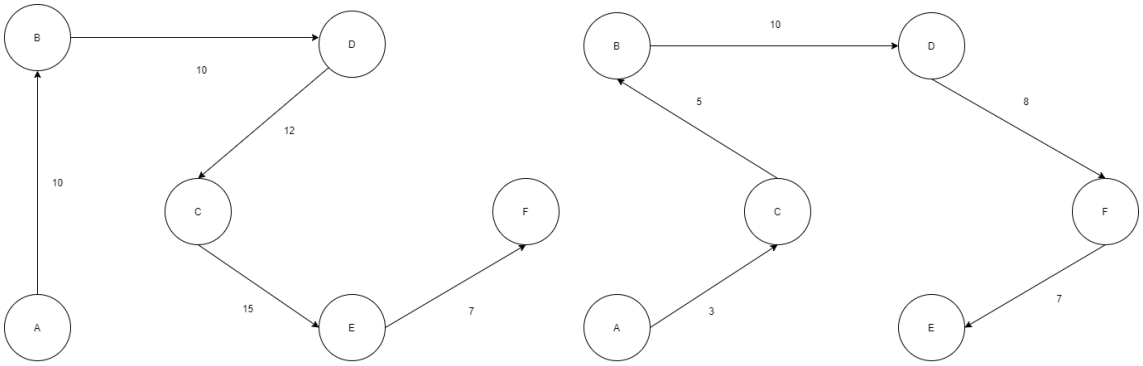


Figure 2.11: Classic salesman problem

Mathematically, we can define this problem the following way:

- Objective Function

$$f_0 : R^n \rightarrow R \tag{2.20}$$

- Controllable Variables (State Space)

$$x = \{x_0, x_1, ..., x_n\} \tag{2.21}$$

- Constraints (Heuristics)

$$g_I : R^n \rightarrow R : (I = 1, ..., n) \tag{2.22}$$

In the Traveling salesman problem, our objective function f_0 (also known as cost function, energy function, or fitness function) would represent the time of travel for the combination of visited cities, x, taking into account the full list of constraints g_n. To solve an optimization problem there are a number of techniques that can be used, each with its own drawbacks and advantages.

2.5.1.1 Hill-Climbing

Hill-climbing is one of the most popular and simple optimization techniques. Categorized as a greedy local search algorithm, hill-climbing is an iterative method that starts by selecting a

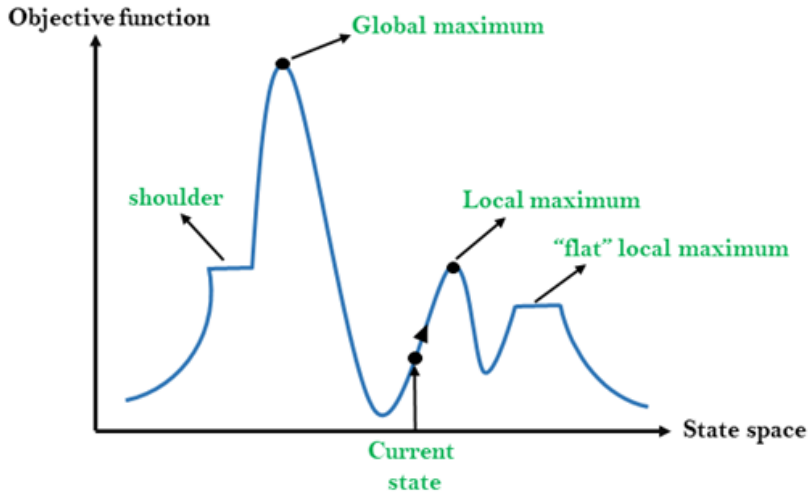


Figure 2.12: Objective function Vs state space

random solution to an optimization problem and proceeds by performing incremental changes to the referenced random solution until it finds a better performing one.

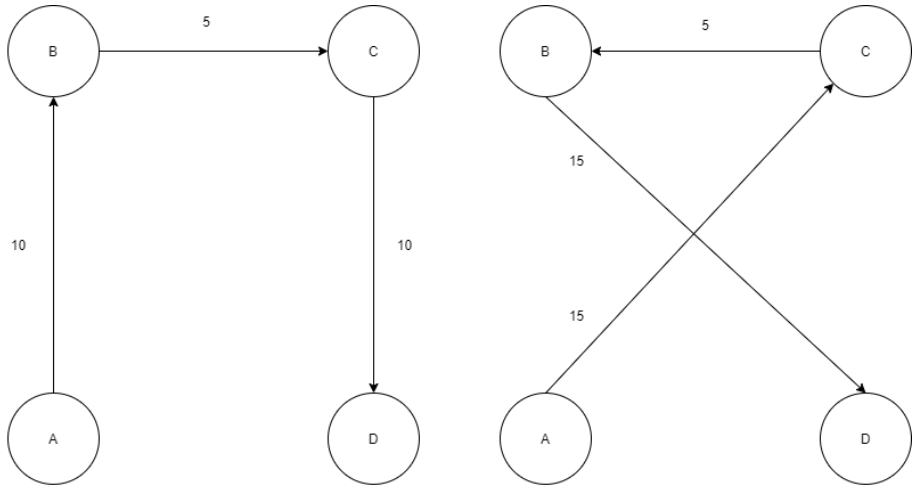


Figure 2.13: Travelling salesman state (left) and neighbor (right)

In the case of the traveling salesman problem, the hill-climber would select a random combination of deliveries, which would represent our *state*. Next, it should perform some kind of simple modification to the combination, for example, switching the order of visits between two cities. This modification to the current *state* is usually named as a *neighbor* of our current *state* (figure 2.13) and all of the possible neighbors represent our *neighborhood*. If said change leads to a better result then the algorithm selects this as its new state and performs a new modification. If none of the neighbors do offer a better solution then the algorithm will return the current solution as the best solution. This concept is demonstrated pseudo-code 1.

Algorithm 1 Steepest Ascent Hill-Climber Pseudo-Code

```
1:  function STEEPESTHILLCLIMBER(ObjectiveFunction, StateSpace)
2:      state = Generate random state within StateSpace
3:      while True do
4:          initial_solution = ObjectiveFunction(state)
5:          neighborhood = GetNeighborhood(state)
6:          while neighbor = neighborhood[i] != None do
7:              if ObjetiveFunction(neighbor) > ObjectiveFunction(state) then
8:                  best_solution = ObjectiveFunction(neighbor)
9:                  state = neighbor
10:             end if
11:         end while
12:         if best_solution == initial_solution then
13:             return state
14:         end if
15:     end while
16: end function
```

There are also different types of Hill-Climbing algorithms based on the *state space* and *neighborhood* that the algorithm can explore and evaluate, specifically:

1. **Basic Hill-Climber** - The algorithm goes over the *neighbors* of the *state* until it finds a better *solution*. Once it does, it changes *state* and creates a new *neighborhood*. This is the most basic of the hill-climbing algorithms to program. As it does not require a full exploration of all possible neighbors, it is faster than the Steepest Ascent Hill-Climber but it also reaches less optimal solutions.

2. **Steepest Ascent Hill-Climber** - Very much the same as the basic hill-climber but, instead of changing state immediately as it finds a better *solution*, all of the neighbor states are first explored and then the best solution is picked. Again if none of the neighbors present a better alternative to the initial solution, then the algorithm stops and returns its current solution.

3. **Stochastic Hill-Climber** - Similar to the steepest ascent hill-climber, this variation also explores the full *neighborhood*. However, it randomly picks between the states that represent an uphill movement, where, the steepest incline neighbor has the best chance to be picked.

Even with the existing variations of the hill-climber algorithm, there are still reasons to consider using completely different optimization algorithms. Looking at figure 2.12, it's possible to see a situation where all the variations of the hill-climber would actually fail at finding the best state of the cost function. Looking at the initial state in that figure, we see that any of the hill-climbing algorithms would eventually lead to the local maximum of the cost function and not its global maximum.

The hill-climber algorithm is highly dependent on the "landscape" of the cost function it is applied to. The higher the number of local maximums the harder it will be for the algorithm to find

the global maximum. A solution to this problem, applied specifically to hill-climber algorithms, is to run the algorithm multiple times starting at different locations.

2.5.1.2 Simulated Annealing

Simulated annealing is a probabilistic optimization algorithm inspired by the real-world, metallurgy industry technique of annealing. The main difference between this algorithm and a basic hill-climber is that, in simulated annealing, there is a chance that, during the convergence of the algorithm, worse solutions may be explored.

With the introduction of the *temperature* parameter, T, whenever the worst solution is found by the algorithm there is a probability, defined by the difference between the *neighbor* solution, s_n, and current solution, s_c divided by the current temperature (equation 2.23), that said the solution may be accepted.

$$P = e^{(s_n - s_c)/T} \tag{2.23}$$

Like its real-world counterpart, simulated annealing introduces a notion of *slow cooling* which, in this context, means that the *temperature* decreases over time (usually decreases every time a new *state* is explored) which, in turn, also decreases the chance of accepting a worse *solution* over time. This means that the chance of accepting worse solutions is considerably higher at the beginning of the run time of the algorithm than in the ending stages. In fact, it is near impossible to accept worse solutions once the *temperature* value is low enough.

Algorithm 2 Simulated Annealing Pseudo-Code

```
1:  function SIMULATEDANNEALING(ObjectiveFunction,StateSpace,Temperature,RefreshCondition)
2:      state = Generate random state within StateSpace
3:      neighborhood = GetNeighborhood(state)
4:      best_solution = ObjectiveFunction(state)
5:      while iterations <= 10 000 do
6:          Temperature = RefreshCondition(Temperature)
7:          neighbor = RandomNeighbor(Neighborhood)
8:          diff = ObjectiveFunction(neighbor) - ObjectiveFunction(state)
9:          prob = e^(diff/Temperature)
10:         if ObjetiveFunction(neighbor) > ObjectiveFunction(state) Or prob > random(0,1) then
11:             best_solution = ObjectiveFunction(neighbor)
12:             state = neighbor
13:             neighborhood = GetNeighborhood(state)
14:         end if
15:     end while
16: end function
```

This method is also widely used as, it is not only simple to implement but, is also known to consistently solve many practical problems by converging to approximate solutions of the global maximum/minimum of an objective function. The fact that it allows for the acceptance of worse conditions largely eliminates the previously mentioned issue of getting stuck in local

maximum/minimum, especially in the early stages of the algorithm run time, as the chance to accept worse solutions is significantly high.

Even so, there are some drawbacks to using simulated annealing, the main ones being that its convergence efficiency is highly dependent on how the exploration is defined, specifically, it depends on how the neighborhood is delimited and on the initial value of the *temperature* parameter and the decreasing condition associated with it. Furthermore, it is also a quite slow algorithm (loops can be over 10 000) which can become quite impractical in situations where the objective function may have a running time associated with it.

2.6 Summary

Over the last 30 years, the SLAM problem has seen great progress which has lead to the development of a plethora of different techniques used to solve it. By employing a number of probabilistic tools, state-of-the-art sensors and, sensor measurement processing mechanisms, the SLAM research community has been able to increasingly improve the autonomous navigation capabilities of mobile robots. Regardless, there is no absolute solution that solves every issue as the problems that SLAM presents are still highly dependent on the combination of robot/environment/performance.

On a mathematical basis, state-of-the-art SLAM implementations like graph-based implementations have increased the reliability of SLAM in large-scale environments while grid-based implementations like GMapping are still very popular due to their robustness and low computational requirements. Beyond this, research on map representation of SLAM algorithms has made new strides by incorporating more advanced features other than just metric measurements. The ability to categorize and differentiate between mapped rooms offers high-level expressive representations. Also, the association of semantic concepts to objects looks to extend the understanding of the environment and offer semantic-based reasoning to navigation.

Optimization algorithms are actively used to facilitate and solve sensor placement problems in various industries. In the automobile industry, by modeling objective functions that are representative of technologies that are dependent on information gathered by exteroceptive sensors like smart parking and obstacle avoidance, researchers have been able to reach interesting sensor placement solutions that are typically better than human-designed ones. Similarly, since SLAM systems are also dependent on the data gathered by these type of sensors, if one could model an objective function that translates SLAM performance to the placement of sensors in a robot, an optimization algorithm, like a hill-climber, could be used to explore the effect of different sensor positions on a SLAM system.

With this in mind, this dissertation will look to use typically used SLAM evaluation metrics to compare a number of SLAM solutions. That metric will then be used as the objective function of an optimization algorithm with the objective of maximizing the performance of said SLAM solutions.

Chapter 3

Intelligent Wheelchairs

A recent study from the World Bank Group [9] found that about 10% to 15% of the world population suffer from some kind of physical ability. Additionally, 10% of this previous number includes people that endure a lower limb dysfunction preventing self-sufficient locomotion which, ultimately, requires the users to use a wheelchair [10] to fulfill this need.

A wheelchair is, in its very basic shape, a chair with wheels (figure 3.1) that is powered manually. The use of this mobility aid has been traced back to ancient Greek culture, however, it is considered that the first manual wheelchair was made in Spain in 1595 [39]. Since then, there has been the introduction of the Power Wheelchair (PW). This type of wheelchair usually includes battery-powered motors that provide self-propellant movement (figure 3.2). Furthermore, users are able to operate this movement through the usage of standard controllers such as hand joysticks, chin joysticks, sip-n-puff, and head joysticks. This extends the utilization of this equipment to users that were previously unable to operate a manual wheelchair. Despite these additions, PWs still show a lack of maneuverability in daily tasks and also remain inoperable to people who lack the motor control to operate a standard controller, such as users who suffer from arthritis or motor neuron degenerative diseases.

To accommodate this need, the concept has further evolved with the introduction of the Intelligent Wheelchair (IW) [40]. This type of wheelchair (figure 3.3) adds to the capabilities of a standard PW, taking advantage of state-of-the-art developments in standard robotics technologies. The adaption of these technologies in this market will provide automation to the navigation and control features of regular PWs, thus reducing the physical, perceptual, and cognitive requirements to operate this instrument. Typically, this is achieved through the addition of sensors connected to a computer that is able to properly process the measurements taken by the sensors and generate commands to the wheelchair.

This field has suffered a lot of research, as explored in the following sections, which has allowed for SLAM techniques to be applied to these types of equipment, helping in the need to automate their navigational capabilities.

Figure 3.1: Manual wheelchair

Figure 3.2: Power wheelchair

Figure 3.3: Intellwheels' intelligent wheelchair

3.1 Common Characteristics

Most IW prototypes use a PW as a basis for implementation [40, 5, 41, 42], so it's important to name the characteristics [39, 40] that differentiate an IW from a PW.

- ***Autonomous Navigation:*** The ability to safely navigate an environment avoiding obstacles. This extends from simple obstacle avoidance algorithms to much more complex methods employing the previously explained SLAM algorithms, and path planning algorithms. However complex the navigation technique used, it usually calls for the marriage of state-of-the-art proprioceptive and exteroceptive sensors.

- ***Intelligent Interface:*** As the IW looks to accommodate heavily physically impaired users, its interface must be easy to use and adaptable to the type of disabilities that the user may suffer from. Therefore, there may be the need to incorporate alternative controllers such as fingertip control, brain-computer interfaces (BCI), touch screens, etc;

- ***Device Communication:*** The ability to communicate with other devices such as automatic doors and other IWs.

3.1.1 Operating Modes

As mentioned, this type of wheelchair must be adaptable. Not only must it be able to satisfy constraints imposed by the user's disease but also by their surrounding environment. As an example, regarding the latter, a user may find themselves in a fixed environment, say their home, and, ideally, the wheelchair should be able to map and locate itself in the environment and switch to an operation mode that allows for fully autonomous navigation through the surroundings. On the other hand, if the user is just temporarily passing through an environment, a semi-autonomous mode of navigation would be more appropriate. Per [40], these operating modes include but are not restricted to:

- ***Machine Learning:*** Some IWs employ the use machine learning algorithms to detect obstacles and plan trajectories;
- ***Following:*** The ability to follow human or animal companions;
- ***Localization and Mapping:*** The main topic of this dissertation. This is achieved by using measurements obtained from proprioceptive and exteroceptive sensors and applying them to SLAM algorithms;
- ***Navigational Assistance:*** Semi-autonomous mode that provides features such as collision detecting, path-planners, and prompters.

Each operating mode mentioned has advantages and disadvantages depending on the combination of environment and user needs it is applied to.

3.2 Developed Prototypes

The production of functioning IW prototypes dates back to about 30 years. Earlier models were originally equipped with rudimentary and costly hardware and software which offered very limited navigation and control techniques. With recent developments in the mobile robotics community, the technology used to develop IWs has progressed into using cheaper hardware and extensively more complex software, in particular, SLAM methods.

3.2.1 Early Models

The first-ever IW was proposed by Madarasz [39] in 1986. This model was equipped with a micro-computer, a digital camera, and an ultra-sound scanner with the intent to offer obstacle avoidance. In 1995 the KISS institute introduced the Tin Man I [43] (figure 3.6) which was a low-cost robot wheelchair equipped with an IR sensor, motor encoders, contact sensors, sonar range finders, and a compass that managed to provide limited autonomous navigation. In particular, this chair was able to navigate through corridors and hallways and perform obstacle avoidance. The

Figure 3.4: FRIEND prototype

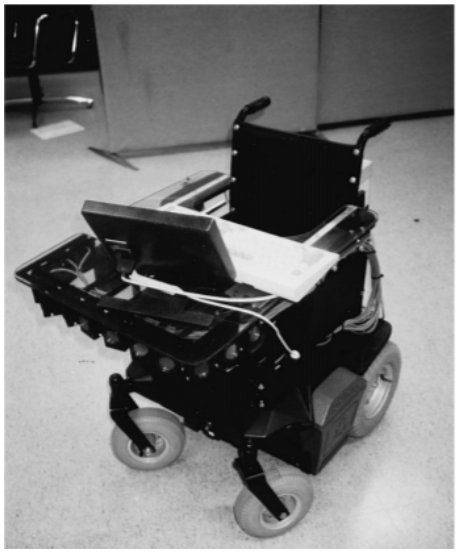


Figure 3.5: NavChair prototype

Figure 3.6: Tin Man I prototype

Tin Man II, developed 3 years later, was a cleaned-up version of this original chair adding a mechanical joystick interface. Also in 1995, the NavChair [44] (figure 3.5) was introduced as an assistive wheelchair navigation system. Equipped with 12 proximity sonar sensors, and an interface composed of a power modulator and a joystick, this chair was also able to navigate corridors and doorways, but, additionally, possessed a follow wall mode. The FRIEND [45] (figure 3.4) robotic wheelchair, developed in 1999, is equipped with a MANUS robot arm. Both the wheelchair navigation and the robotic arm are controlled by the user through voice commands. This feature extends the usage range of the former intelligent wheelchair prototypes as its alternative control method is adaptable to users that are unable to use their upper limbs.

3.2.2 Recent SLAM-based Models

All of the following IW projects, including the one associated with this dissertation, are implemented using the Robotic Operating System (ROS) and make use of some sort of SLAM solution. ROS [46] is an open-source framework used to write robot software. It is based on services, topics,

messages, and nodes. Nodes communicate between each other through messages, topics are published and subscribed by nodes and services are, essentially, a pair of messages. This is a widely used tool in the robotics community due to its compatibility and modularity.

Equipped with the visual sensor Microsoft's Kinect V2, wheel encoders, and an on-board computer, researchers at the University of Toronto [5] have built a relatively cheap and portable IW framework focusing on a particular case of wheelchair navigation, the traversal of narrow doorways. The system architecture uses the open-source RGB-D SLAM ROS package. RGB-D SLAM is a graph-based SLAM approach capable of building 3D maps using images taken by Kinect-style cameras. In this case, the 3D map was converted to a 2D map using the ROS package *PointCloudToLaserScan*.

Additionally, while this SLAM algorithm also offers visual odometry (VO), the authors opted to use the odometry generated from the wheel encoders, since only one camera was used and the environment often consisted of areas with insufficient visual features to properly perform VO. Given that only one camera was used, during the exploration of the environment, the wheelchair would often have to perform 360º rotations in the corners of a room to properly perform data association. To fix this issue a rear-facing camera is a possible solution although, the processing of the data would lead to higher computational requirements, which would increase the cost of the IW.

Figure 3.7: IW developed in [5]

Similarly, [41] developed an IW using identical hardware and open-source software. Here, the developers used the Percipio RGB-D camera, only in this case, VO from the camera was used. Regarding the SLAM algorithm, they opted for the previously described open-source ROS package of GMapping (section 2.3.2.1). A mobile APP interface was also developed for the control of the wheelchair which included the functions of remote control through a virtual joystick, visualization of map and position of the wheelchair, and access to color images taken by the camera. The camera used was found to be better suited than the standard Kinect camera as it is smaller in size and can be powered through a USB connection, being more suitable for integration in mobile robots.

The research conducted in [6] is aimed at comparing different open-source SLAM algorithms applied to an IW. The hardware kit added to the original power wheelchair was composed of wheel encoders, a Kinect camera, a SLAMTEC RPLIDAR-A3-LiDAR, and an IMU unit connected to a RaspberryPi model b+ placed as seen in figure 3.8. The following open-source SLAM algorithms were used:

- ***GMapping:*** Already explained in section 2.3.2.1;
- ***HectorSLAM:*** A LIDAR-based 2D mapping SLAM algorithm based on a Gauss-Newton approach for scan matching;

- ***RTAB MAP:*** Similar to the RGB-D SLAM approach, Real-Time Appearance-Based Mapping is an RGB-D, Stereo, and LIDAR graph-based SLAM approach;
- ***RBG-D SLAM:*** Already explained in this same section;
- ***VINS-Mono:*** Visual Inertial Navigation System - Monocular is a SLAM algorithm based on visual-inertial odometry.

According to the tests performed in [6], it was found that, among the visual SLAM methods used, VINS-Mono was computationally more expensive than the other two and, at higher speeds (over 1 m/s) RTAB MAP was found to have more errors than the others. Regarding the laser only SLAM approaches HectorSLAM and GMapping, the latter proved to build more accurate maps and had less error in its pose estimate. Although HectorSLAM was found to be a pretty light algorithm it was bound to not be very optimal since it was using no odometry at all to locate the wheelchair.

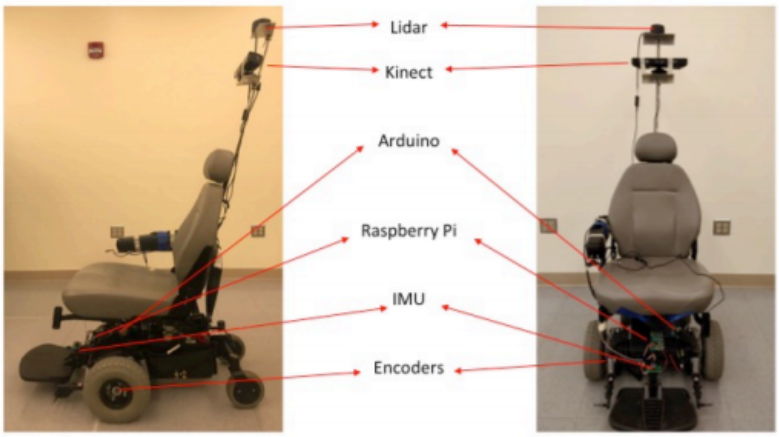


Figure 3.8: IW sensor placement in [6]

3.2.2.1 Vulcan - Intelligent Robotic Wheelchair [42]

The Vulcan robotic wheelchair is an adaptive IW framework currently being used as a research platform at the Michigan State University. This project is focused on the study and development of assistive navigational technologies for IWs targeted to a campus-like environment, which includes indoor and outdoor areas connected through paths and hallways. The framework in question, called Hybrid Spatial Semantic Hierarchy, has 3 different objectives:

- Map natural human environments through computer-vision;
- Adapt this mapping to natural language voice commands;
- Develop effective human-robot interaction that maximizes the above.

With this in mind, the first wheelchair model developed in this project stood as a hybrid of laser and vision-based models capable of building 2D metric maps that incorporated information about potential hazards in the environment [47]. To achieve this, a grid-based SLAM algorithm using two 2D LIDAR lasers was used to perform the metric mapping while a stereo camera was used to obtain 3D information about the environment, allowing for the incorporation of hazard assessments into the map itself.

Further work on this project has upgraded the mapping performed to include semantic classification [48] (discussed in section 2.4.5) allowing the robot to make distinctions between different types of rooms. It has also included the addition of dynamic elements of the environment (e.g people) into its map. The most recent research done, studies the inclusion of "socially-aware navigation" [49] into the autonomous navigation feature of the wheelchair.

3.2.2.2 Intellwheels

Intellwheels originated in the Faculty of Engineering of the University of Porto in 2007 and is an ongoing research project aimed at developing a platform used for the production of intelligent wheelchairs [50]. In 2015 a new iteration of this project was introduced, Intellwheels 2.0, with the intent to use the concepts developed in the first iteration to produce fully functioning products. These products include an IW framework/kit, a realistic IW simulator, and a multi-modal IW control interface.

Intellwheels 1.0

The concept behind the project is not to just develop an IW wheelchair but to create a dynamic platform that facilitates the development of IWs. To achieve this, regarding software, the platform uses a multi-agent system (MAS) paradigm. On the other hand, addressing the hardware aspect, a generic hardware framework is used that is designed to fit in most powered wheelchairs.

The multi-agent system enables the easy integration of a wide range of sensors, actuators, user interfaces, navigation technologies, path planning techniques, and cooperation methodologies by dividing these features into separate modules as presented in figure 3.9.

Figure 3.9: Different Intellwheels modules [7]

Furthermore, these modules belong to the different agents of the MAS. The agents communicate with each other following the communication standards set by the Foundation for Intelligent Physical Agents (FIPA) (figure 3.10).

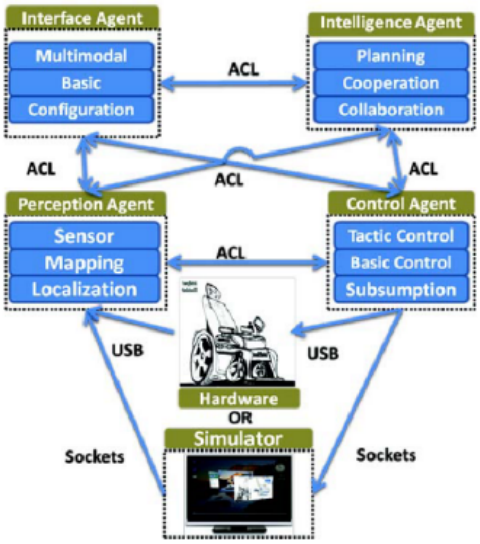


Figure 3.10: Intellwheels MAS architecture [7]

Regarding the simulator module, a mixed reality (MR) simulator was developed using an adaptation of the "Cyber-Mouse" simulator. The simulator produced is able to create a virtual world to easily and inexpensively run experiments with developed IWs. Additionally, it also enables the communication between simulated agents and real-world agents (for example, a real wheelchair agent communicating with a virtual wheelchair agent or virtual door agents). This MR concept stretches the Intellwheels simulator's capabilities beyond testing algorithms as, through its usage, we can evaluate the performance of a real IW in dynamic simulated environments (figure 3.11). Since a lot of the work done in this dissertation will be done using simulators, additional existing simulators are explored in section 3.3. Later, this module was upgraded to IntelSim, a simulator developed based on the existing USARSim 3D simulator.

In 2011, Intellwheels presented a functioning prototype of an IW (figure 3.3) with the following integrated sensor hardware module installed on a commercial power wheelchair. A U-shaped bar with a set of 8 ultrasound sensors and 12 infrared sensors, 2 wheel encoders, and 2 webcams, one directed at the floor and the other one at the user's face (to read facial expressions). Through the module, this wheelchair was capable of performing semi-autonomous actions [50]. Using probabilistic models for the odometry motion model the wheelchair was able to perform active localization with a prior map. Consequently, path planning through A* algorithms was also achieved.

Intellwheels 2.0

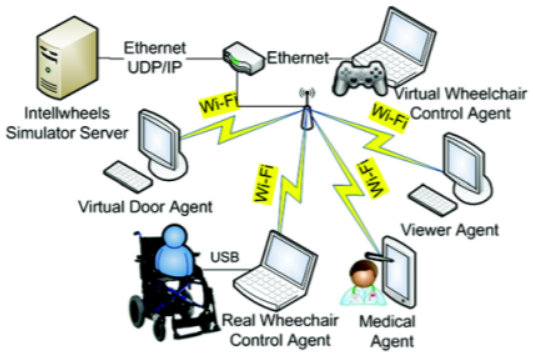


Figure 3.11: Intellwheels MR simulator architecture [7]

As mentioned, this next iteration of the Intellwheels project is focused on developing fully functional products. Specifically, it aims to produce the following:

- A dynamic low-cost IW framework/kit that is able to transform different types of powered wheelchairs into intelligent wheelchairs with minimum hardware changes and low visual and ergonomic impact. It will include all hardware and software, specific to the robotics area, to solve the problems of sensing, mapping, localization, control, navigation, and planning on a wheelchair.
- A realistic IW simulator, with a 3D interface, virtual reality that includes three instructional serious games with the intent to train the user on how to control the wheelchair.
- A totally configurable interface that is adaptable to the special needs of the user.
- Complete IW prototypes with the above features included.

This dissertation is included in the first item above. This framework will work to build upon the semi-autonomous navigation features of the previous Intellwheels iteration IW prototype by implementing state-of-the-art hardware and SLAM software. This will allow the wheelchair to perform SLAM in unknown dynamic environments eliminating the previous need for prior maps. This feature, together with a path planning algorithm and a configurable interface could extend the navigation capabilities of an IW to a near fully autonomous state.

3.3 Multi-Robot Simulators

Robotic systems are often difficult and costly to deploy in the real world due to their complexity, so researchers often look to perform development and validation of real-world technologies on virtually simulated environments. Similarly, most of the work done during this dissertation will be done in a simulated robotic environment. As later discussed in this document, performing

the experiments done during this project in a real-life wheelchair would lead to very high time-consuming efforts which do not satisfy the time restrictions set on a regular dissertation. Therefore, choosing an appropriate and realistic environment is an important task. All of the following simulators can be fully integrated with ROS.

Gazebo [51]

Gazebo is an open-source 3D multi-robot simulator, programmed in C++, that relies on the Open Dynamics Engine and the Object-Oriented Graphics Rendering Engine to provide simulated 3D robots and environments. Among other characteristics, the simulated objects in the world have friction and mass. Access to multiple shapes along with different joints is also available making it possible to design simulated robots. Gazebo is by far the most used simulator in the robotics community [52].

USARSim [53]

The Unified System for Automation and Robot Simulation (USARSim) is a 3D simulator based on the Unreal Tournament game engine. USARSim was originally developed to simulate multiple robots in search and rescue missions. It supports sound sensors, touch sensors, lasers, odometry, and cameras. Additionally, the Intellwheels simulator IntelSim is also adapted from USARSim.

MORSE [52]

The Modular Open Robots Simulation Engine (MORSE) is based on the open-source project Blender, a 3D game engine. It supports any 3D model, so any 3D model can be imported for use. MORSE operates from a command line and it was implemented using Python.

Webots [54]

Webots is also an open-source simulator used to simulate, model, and program mobile robots in 3D environments. The usage of the ODE library allows it to simulate rigid body dynamics and associate attributes to objects such as mass, shape, and texture. It supports a wide range of programming languages to build robots, specifically, Java, C++, C, Python, and MATLAB. It is also compatible with different sensors frequently used by the robotics community such as light sensors, proximity sensors, GPS, touch sensors, lasers, and accelerometers.

3.4 Summary

The evolution of wheelchairs over the years along with state-of-the-art technologies in modern robotics has allowed for the introduction of the Intelligent Wheelchair. Previous wheelchair mod-

els have provided people with lower-limb dysfunctions with locomotion by offering either manual or powered control of wheelchairs. Most intelligent wheelchairs aim to expand upon this, by offering the user alternative control methods and equipping powered wheelchairs with autonomous navigation methods. To achieve this, a large number of prototypes developed have used SLAM technologies to deal with the mapping and localization problems of their navigation.

These prototypes have taken advantage of the commonly used robotics software ROS for their implementations. Specifically, using ROS, there are many open-source packages like GMapping, RGB-D SLAM, RTAB MAP, and VINS-Mono that have been used to implement SLAM with limited computational efforts and sensors. During this dissertation, we will evaluate the performance of similar open-source SLAM packages, in the Intellwheels project context. To perform this testing, there are many 3D simulators available, like gazebo and webots, that are able to replicate the wheelchair and the sensors models onto different simulated environments, which allow researchers to perform testing in a cheap and convenient way.

Chapter 4

Simulation Environment

In the previous chapter, it was mentioned that researchers in the robotics industry often perform initial research and tests under simulated environments because of the shortcomings that come with deploying robotic systems in the real world (monetary costs and elevated time consumption). As said, the work done in this dissertation follows in the same footsteps. Therefore, this chapter is dedicated to the detailing of the elements present in the simulated environment that was used. Some of the work used, like the wheelchair model and the navigation stack, was developed in the previous research done along the Intellwheels project, while, on the other hand, elements such as the odometry noise model, SLAM evaluation metrics, world models, and optimization algorithms were all fully developed during the course of this project.

Both the wheelchair model and the navigation stack were implemented using the previously mentioned middleware, ROS. So, all the framework developed in this project is also fully compatible with ROS.

Overall, the whole project is divided into various ROS packages containing each of the elements used, some of them newly developed, like the odometry noise model, optimization loop, and benchmarking tools, while others were either existing open-source ROS packages, like all of the SLAM solutions used, or packages developed in previous iterations of the project, like the wheelchair description and the navigation stack. Nonetheless, ROS packages are able to include dependencies from different packages so all the framework used here was condensed down to two different ROS packages, namely:

- ***intellwheels_project*** - This is the main package of the project and it is used to launch all of the required tools since it has dependencies on all of the other used packages. Furthermore, it also contains the developed scripts to the nodes used for the benchmarking metric, the optimization loop, and the noisy odometry model.

- ***intellwheels_desc*** - This package contains the wheelchair description used to spawn the wheelchair and all of its additional elements like the laser sensors used and the navigation stack.

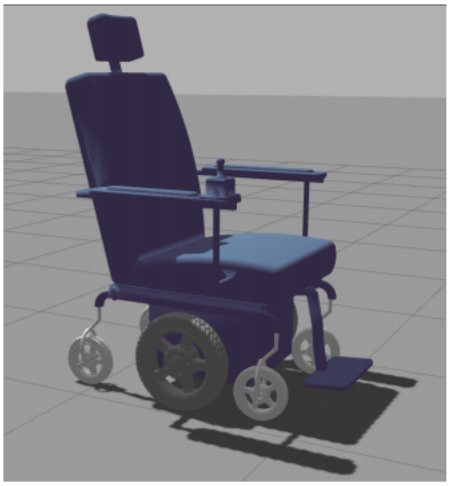


Figure 4.1: Gazebo wheelchair model

Figure 4.2: Real-world wheelchair model

4.1 Intelligent Wheelchair Model

Developed by [55], the model in figure 4.1 is a 6-wheeled motorized wheelchair model inspired by the real-world wheelchair presented in figure 4.2. While obviously not being a 1:1 representation of its real-world inspiration, the wheelchair model is still able to faithfully depict the actual footprint of a wheelchair. Furthermore, a definitive wheelchair model has yet to be associated with the project, so it make sense to conduct research on a "standard" wheelchair model since, more specific models, may have additional "quirks" that may restrict some portion of the research.

This initial prototype was created in the Simulation Description Format (SDF) using the Blender software along with the Phobos[1] plugin. Models using this format are both compatible with ROS and the Gazebo multi-robot simulator mentioned in 3.3. An SDF model is a modified Extensive Markup Language (XML) format that is able to describe objects and environments for robot simulators, visualization, and control. When used to describe a robot, SDF is mainly composed of three tags.

- ***Links*** - These represent every visual element that is seen in the robot model such as the arms rests, wheels, chair, etc. Usually, but not necessarily, each link has an underlying collision, inertial, and mesh tag.
- ***Joints*** - The *joint* component creates connections between links by establishing parent-child relationships. This connection makes it possible to see how a child link moves relative to its parent link.
- ***Plugins*** - This final element allows for the integration of external components such as driver plugins and sensor plugins that offer additional functionalities to robot models. It is also compatible with a number of tools in ROS.

[1] https://github.com/dfki-ric/phobos

Even though SDF offers quite an effective compatibility with Gazebo and ROS, there still remain some functionalities that are unmet with the usage of this format. Specifically, ROS' rviz[2] visualization tool, which enables the visual interpretation of most data published in a ROS framework, is only able to work with robot models in the Unified Robot Description Format (URDF), which, similarly to SDF, is also a modified XML format. It also follows the same principles as the SDF with the usage of *links*, *joints* and *plugins*.

The visualization of data in a ROS framework is quite useful as it enables for a much more convenient debugging process which will become quite important once the SLAM solutions are implemented and working, as it will be possible to actually see the localization and mapping processes taking place in real-time. So, since Gazebo is also compatible with URDF, researchers in the Intellwheels project, have made an effort to convert the wheelchair model description from SDF to URDF. In addition, the URDF files were also converted to the Xacro[3] format.

Xacro is a scripting mechanism that allows more modularity and code re-use when defining a URDF model. It also is compatible with the usage of math constants, equations, and loading YAML files as dictionary data structures. This last aptitude will become quite useful in the optimization portion of the project explained further in section 4.5.1.

Furthermore, we can divide the Gazebo program into two separate sub-programs, *gzserver* and *gzclient*. Respectively, the first is basically the core of the program, where the communications and physics engine are started and maintained, while the second is the visualization tool of Gazebo. The Gazebo visualization tool consumes considerably more computational resources than rviz, as *gzclient* renders all of the 3D models present in the simulation. Since we only need *gzserver* to create and handle the actual physics simulation, *gzclient* was replaced altogether by rviz, reducing the overall computational consumption of the simulation environment. Gazebo allows the physics engine to be sped up/slowed down, however, it is dependent on the computational resources available. For reasons further discussed in section 5.2, it is advantageous to reduce the computational of the simulation environment as minimum as possible.

4.1.1 Odometry Noise Modelling

The real wheelchair moves as a differential drive robot with motors placed in the center of two wheels. To replicate this in the wheelchair model, the *libgazebo_ros_diff_drive.so* plugin is used. There are two important features which this plugin offers. It controls the actual movement of the wheelchair by using all of the data published to the ROS topic /cmd_vel, which contains information about the linear and angular velocity that the wheelchair should take. This data can either be published manually by the user, by some sort of controller, or by using a navigation stack that is able to translate world goals to /cmd_vel topics. More importantly, the plugin is also able to simulate the behavior of odometry sensors by publishing to the TF tree (figure 4.3), the transform between the odometry frame and the wheelchair base frame. This transform is what ROS-based SLAM solutions use to get odometry data. Unfortunately, the transform that is

[2] http://wiki.ros.org/rviz/UserGuide
[3] http://wiki.ros.org/xacro

published by the plugin includes next to no noise and offers no way to add any. This fact stands as an issue because, as explained in chapter 2, most SLAM systems rely on both odometry and visual sensors to perform localization and mapping. If one of these sensors is 100% correct, the SLAM algorithm will end up making all localization predictions based on that sensor data, which, since there is noise to deal with, will also be 100% correct.

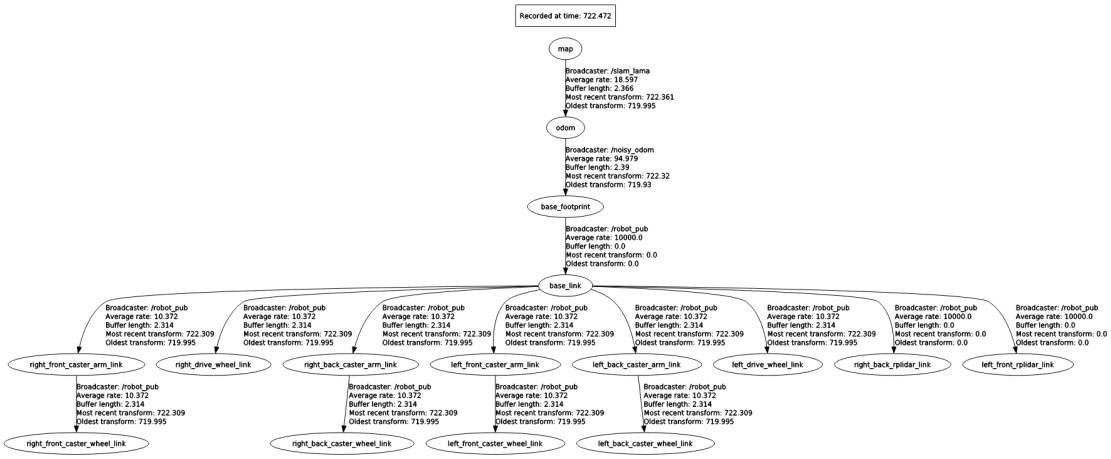


Figure 4.3: Wheelchair model TF tree

The nonexistence of noise would make it next to impossible to make any distinctions between the SLAM systems' performance since all of them are quite capable of using accurate odometry data to perform exact localization predictions. To deal with this issue, the *noisy_odom* node, contained in the script *noisy_odom.py* was created. This node takes the ground truth information of the robot location, adds noise to it, and replaces the original odometry to robot base transform. This is possible, as the differential drive plugin allows the original transform not to be published, while still maintaining its other functionalities. The original transform is then replaced by the noisy odometry data coming from the *noisy_odom* node by using the *tf2_ros* python API, which has methods included in it for publishing transforms to the ROS TF tree.

By adding the plugin *libgazebo_ros_p3d.so*, which publishes the exact location of the robot to topic /ground_truth/state, at a rate of 100 Hz, the displacement, (dx, dy), and rotation, Δrot, is calculated between the current position, $rpos_t$, and the past position, $rpos_{t-1}$. This displacement/rotation is then multiplied by a scaling factor, α_1, which alters the magnitude of the displacement/rotation along with addition of Gaussian noise, $G(0,\sigma)$, that is also subject to the magnitude of displacement/rotation between t and $t-1$.

$$rpos_{t-1}^{-1} \cdot rpos_t = \begin{bmatrix} \cos(rot_{t-1}) & -\sin(rot_{t-1}) & x_{t-1} \\ \sin(rot_{t-1}) & \cos(rot_{t-1}) & y_{t-1} \\ 0 & 0 & 1 \end{bmatrix}^{-1} \cdot \begin{bmatrix} \cos(rot_t) & -\sin(rot_t) & x_t \\ \sin(rot_t) & \cos(rot_t) & y_t \\ 0 & 0 & 1 \end{bmatrix} \tag{4.1}$$

$$rpos_{t-1}^{-1} \cdot rpos_t = \begin{bmatrix} \cos(\Delta rot) & -\sin(\Delta rot) & dx \\ \sin(\Delta rot) & \cos(\Delta rot) & dy \\ 0 & 0 & 1 \end{bmatrix} \tag{4.2}$$

$$\begin{bmatrix} noisy\_dx \\ noisy\_dy \\ noisy\_\Delta rot \end{bmatrix} = \alpha_1 \times \begin{bmatrix} dx \\ dy \\ \Delta rot \end{bmatrix} + \begin{bmatrix} g(0,\sigma_x) \\ g(0,\sigma_y) \\ g(0,\sigma_{rot}) \end{bmatrix} \tag{4.3}$$

$$\begin{bmatrix} \sigma_x \\ \sigma_y \\ \sigma_{rot} \end{bmatrix} = \begin{bmatrix} a_1|dx| + 0.3a_1|dy| + a_2|\Delta rot| \\ a_1|dy| + 0.3a_1|dx| + a_2|\Delta rot| \\ a_3|\Delta rot| + a_4\sqrt{dx+dy} \end{bmatrix} \tag{4.4}$$

As seen in equations 4.3 and 4.4, both the scaling factor α_1 and the standard deviations related to the Gaussian noise make it so the actual noise that is added is always related to the amount of displacement/rotation that has occurred between the time frame of t and $t-1$. Additionally, the set of variables (a1,a2,a3,a4), dictate the effect that those displacement/rotations have on the normal distribution that is sampled to generate the Gaussian noise.

After several tests were performed, with a range of values for (α_1,a1,a2,a3,a4), it was found that the combination (0.75,0.2,0.2,0.2,0.2) creates a considerable amount of noise, at least, enough to facilitate the distinction between the different SLAM systems that were tested. In figure 4.4 we can see the difference between the ground truth path of the wheelchair and the path obtained from the noisy odometry. From the starting position (x,y) = (0,0) it is possible to see that the ground truth and noisy odometry data remain similar for some distance. However, from (20,0) onwards, the odometry data becomes completely unreliable, on a long-term scale, which, in a real-life scenario, is expected.

In sum, all the experiments were performed with the following odometry noise model:

$$\begin{bmatrix} noisy\_dx \\ noisy\_dy \\ noisy\_\Delta rot \end{bmatrix} = 0.75 \times \begin{bmatrix} dx \\ dy \\ \Delta rot \end{bmatrix} + \begin{bmatrix} g(0,\sigma_x) \\ g(0,\sigma_y) \\ g(0,\sigma_{rot}) \end{bmatrix} \tag{4.5}$$

$$\begin{bmatrix} \sigma_x \\ \sigma_y \\ \sigma_{rot} \end{bmatrix} = \begin{bmatrix} 0.2|dx| + 0.06|dy| + 0.2|\Delta rot| \\ 0.2|dy| + 0.06|dx| + 0.2|\Delta rot| \\ 0.2|\Delta rot| + 0.2\sqrt{dx+dy} \end{bmatrix} \tag{4.6}$$

4.1.2 Sensors Used

As stated previously, SLAM systems depend on the information obtained from both exteroceptive sensors and interoceptive sensors. In section 4.1.1, it was explained how the odometry sensors and odometry data are modeled in our simulated environment. Regarding the exteroceptive information in our model, all information will come from a single (or multiple) infrared sensor visually modeled after the RPLidar A3 (figure 4.5).

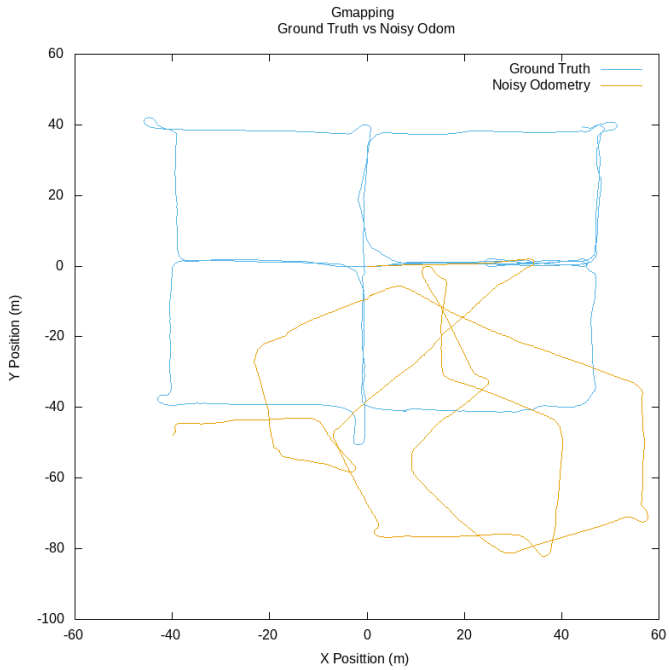


Figure 4.4: Ground truth Vs noisy odometry example

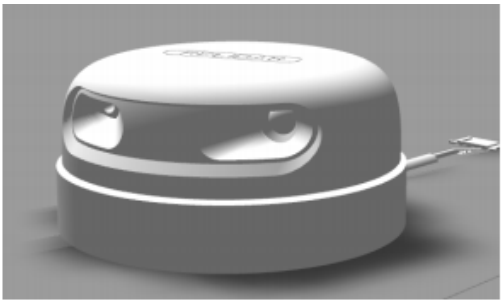


Figure 4.5: RPLiDAR sensor model

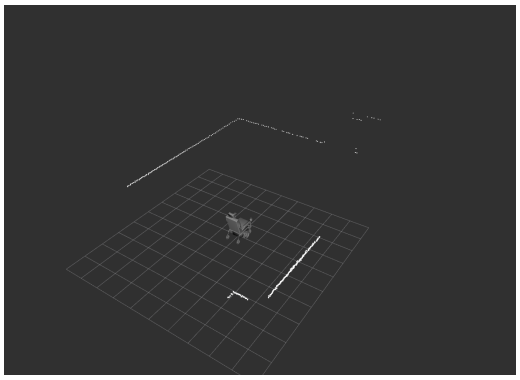


Figure 4.6: Laser scan rviz representation

Again, Gazebo already has available plugins that are able to model the behavior of general laser sensors. The plugin *libgazebo_ros_laser.so* has the ability to spawn and simulate customizable 2D laser sensors into Gazebo simulations and to publish the laser scan obtained from the sensors to ROS topics. The SLAM solutions subscribe to the topic to gather all of the laser scan data.

Like the rest of the wheelchair model, the sensor model is also defined within URDF files containing the *link* and *joint* parameters specifying the position and orientation of the sensor in relation to the wheelchair model. The customization aspect comes in the form of various changeable tags that are related to the standard characteristics of 2D laser sensors, specifically.

- ***Min/Max Range*** - Defines the distance interval that the lasers are able to cover.

- ***Field of View*** - The field of view of the sensors in radians.
- ***Update Rate*** - The rate of the messages published to the specified ROS topic.
- ***Resolution*** - The distance between the emitted laser scans.
- ***Gaussian Noise*** - Models the Gaussian noise that is added to the laser scan figures. Both the mean and the standard deviation can be changed in this parameter.

Most of the sensor configurations will be changed during the experiments conducted, however, the Gaussian noise will remain defined as $(\mu, \sigma) = (0.0, 0.03)$, which means that all the laser scan data gathered will have an added Gaussian noise with a mean of 0 and a standard deviation of 0.03.

In reality, even though the sensors are visually modeled after the RPLIDAR A3, at no point will any of the sensors used have any of the capabilities of its real-life counterpart. The usage of this visual model was done for convenience sake, as, in this case, the visual component of the sensor has no drawbacks on the experiments made.

Additionally, at some point during the experiments 2 laser sensors will be used, which publish their data to different ROS topics. This stands as an issue as most SLAM ROS packages only take into account a single topic. To solve this, whenever multiple laser sensors were required, the *ira_laser_tools* ROS package, developed by [56], was used to merge multiple laser scan topics into a single one.

4.2 World Models

To analyze the SLAM systems, 3 different buildings were modeled using Gazebo's building editor. These models were created using real-world laser scan data, made available by a consortium of robotics researchers, in the radish robotics repository [57], who have deployed their own robotic systems into these environments. The goal in using this data is that that the results gathered in these experiments are transparent enough to be analyzed alongside other research, as these specific datasets are commonly used and recognized by the SLAM community [8, 58, 59].

The Univ. of Freiburg Building 079 (figure 4.9) is the smallest of the worlds, at a size of 76x28 meters, being mostly composed of small rooms connected by a long corridor in the middle. On the other hand, the largest world is the ACES building (figure 4.8), at a size of 115x106 meters. Finally, at 57x58 meters, Seattle's Intel Research Lab (figure 4.7) is a "middle-ground" between the two previous models in respect to the overall size.

Each of these worlds has interesting discrepancies that distinguish them from one another which, in turn, will allow the comparison between different SLAM systems under different conditions. For example, the ACES building, theoretically, should be the hardest world to perform SLAM in, as most of the map is composed of extremely long corridors with very limited connections between them. This should not only make the loop closing processes of SLAM difficult but also, traveling long corridors is a known nuisance for most SLAM systems as, if the distance

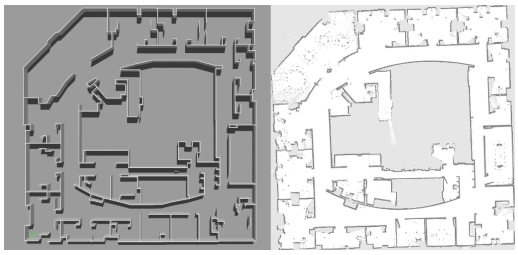


Figure 4.7: Intel Research Lab model (left) & dataset (right)

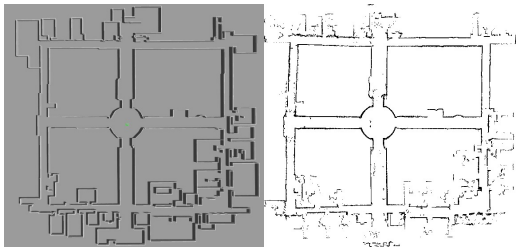


Figure 4.8: ACES Building model (left) & dataset (right)

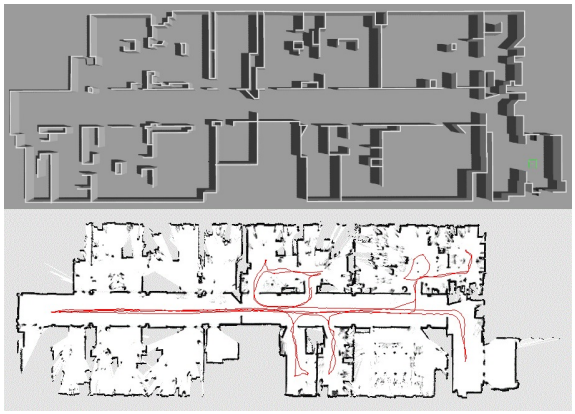


Figure 4.9: Univ. of Freiburg Building 079 model (top) & dataset (bottom)

covered by the exteroceptive sensors does not contain the end of the corridor (which would typically contain a perpendicular wall to the robot traveling orientation), the algorithm relies mostly on the odometry information which is not dependable for long spans of time. In contrast, the Univ. of Freiburg building should stand as the opposite, as the world is considerably smaller and a large number of the rooms are connected, which should make for a much smoother mapping and localization process. The Intel research lab has a considerable amount of different rooms, however, there is no common connection between them, in fact, the world is really just a long corridor folded up into a square, which, since it is not as big as the ACES building nor as small as Frei079, it should stand as good "middle-ground" between the other two buildings.

Additionally, a small room, similar to one of the rooms in the Univ. of Freiburg model, with a size of 20x8 meters, was modeled (figure 4.10) to be used in the sensor placement optimization phase of the dissertation. The point in using such a small world in this specific case is to keep the run-time of the optimization algorithm as low as possible while also still applying the SLAM solution to a real-world scenario.

4.3 2D SLAM Algorithms

All of this projects' framework is implemented using ROS, so, consequently, all of the SLAM algorithms that are compared must have a ROS wrapper. Given these requirements, there are

Figure 4.10: Small room model

many SLAM projects implemented in ROS. This section offers a brief description of each of the algorithms (all of them being 2D SLAM solutions) that were picked to be compared.

A. ***HectorSLAM***[4] - HectorSLAM is a solution developed with the intent to be used in Urban Search and Rescue (USAR) missions. To achieve this, the authors of [60], offer a light online SLAM solution capable of learning occupancy grid maps, by taking advantage of the high update rate and low distance measurement noise of modern LIDARs and by using a fast approximation of map gradients and a multi-resolution grid. Pose estimation is based on the optimization of the alignment of beam endpoints with the current map, while, to solve scan matching, a Gaussian-Newton method is used to find the rigid transformation that matches the laser beams with the map. This solution is meant to be used in environments where hard loop closures are not present so it should work best in small indoor environments or in an outdoor setting.

B. ***KartoSLAM***[5] - KartoSLAM is a graph-based SLAM solution. As explained in section 2.3.3, these types of SLAM solutions revolve around building a graph whose nodes represent the robot's positions and landmarks observed and the connections between these nodes are either odometry data or laser sensor measurements. Solving SLAM then turns into a graph optimization problem to determine the most likely configuration of the poses given the edges of the graph. In this case, KartoSLAM uses the Sparse Pose Adjustment (SPA) method for both scan-matching and loop-closure. In graph-based SLAM fashion, the bigger the environment the higher the number of nodes, which in turn, increases the process' computational requirements. It should be noted, that the available ROS implementation of this solution is hard-coded to not accept laser scans with a higher range of measurement of 12 meters.

C. ***SLAM Toolbox***[6] - SLAM Toolbox is also a graph-based SLAM, in fact, developed on top of a highly modified version of KartoSLAM. This solution was designed to work in large and dynamic spaces such as large retails stores and warehouses without any additional help from a user. As stated in [61], a lot of modifications were made to the legacy version of

[4] http://wiki.ros.org/hector_slam
[5] http://wiki.ros.org/slam_karto
[6] https://github.com/SteveMacenski/slam_toolbox

KartoSLAM, specifically, scan matching was sped-up by enabling multi-threaded processing, SPA optimization was replaced with Google Ceres, providing faster and more flexible optimization settings. Additionally, serial and deserialization support was added for saving and reloading functionalities between mapping sessions. Finally, processing modes were added along with a K-D tree-search to process measurements.

D. ***Gmapping***[7] - Gmapping, thoroughly described in section 2.3.2.1, is a particle filter based SLAM solution. Even though it the most widely used open-source SLAM package, it is known to require some tuning to be able to work properly, especially in large-scale environments. It is also a very high demanding solution in terms of computational resources.

E. ***LaMa Online SLAM***[8] - The Localization and Mapping (LaMA) library, developed at the Intelligence Robotics and Systems (IRIS) of the University of Aveiro, provides two solutions to the SLAM problem. The LaMa Online SLAM [59] version offers a fast scan matching approach to robot localization supported by a continuous likelihood field paired with a Sparse-Dense Mapping (SDM) framework [62] used for efficient implementation of 3D volumetric grids. Its main feature is efficiency, boasting very low computationally effort and low memory usage when possible.

F. ***LaMA Particle Filter SLAM*** - The second version, LaMa Particle Filter SLAM, is an extension of the previous version. Like Gmapping, it offers a SLAM solution based on the Rao-Blackwellized particle filter with the support of a similar fast scanning method of the Online SLAM version used for pose refinement and a flexible space management data structure. Additionally, by taking advantage of independence between particles its computational efficiency is further improved through multi-threading [63].

Note that all of these solutions are heavily modifiable by changing the parameters associated with their ROS implementations. These can go from the number of particles that are used in the RBPF-based SLAM solutions, to the max range of the laser readings that are allowed to be used in the solutions. As further detailed in 5.1, some of these parameters were changed in order to keep all of these solutions as close as possible in terms of acting conditions.

4.3.1 Benchmarking Metrics

Benchmarking SLAM systems is, unfortunately, not a straightforward task as all of these solutions are affected by the customizable parameters available in their implementations along with the computational resources available for usage. An example of this is in the multi-threading option that some of these solutions offer. To solutions that use this approach, such as the LaMa and SLAM Toolbox, the number of threads available will have a considerable impact on the performance of these implementations whereas, on others, like Gmapping, not so much. This to say, that

[7] http://wiki.ros.org/gmapping
[8] https://github.com/iris-ua/iris_lama_ros

not only the SLAM solutions are often heavily customizable, their performance is also reliant on the system they are operated from.

On top of that, even though there have been multiple solutions proposed to solving the SLAM problem, there still hasn't been developed a gold standard on how to compare these solutions. In the community of feature-based SLAM techniques, researchers often present benchmarking results based on the euclidean distance between the estimated landmark location and its true location. On the other hand, in the area of grid-based implementations, the most common method is to use visual inspection to compare the maps generated by the SLAM solutions to the original blueprint of the environment. However, both of these methods have their drawbacks as they both rely on either a global appreciation of the map or subjective inference, respectively.

This idea is exemplified in figure 4.11. In (a) we have the original topology of a given map, which is just a long corridor with multiple doors on each side. In (b) we have the position where the measurements were taken by the robot and in (c) we have the actual map estimate returned by a SLAM algorithm. Even though the quality of the map (c) has obviously decreased in regards to the map in (a), the original topology of the map, that is, the correct length of the corridor the same number of doors on each side, is still accurately depicted. This means that even with the reduction of quality in the map estimate, which these two previous metrics would reflect, it is actually still usable for robot navigation.

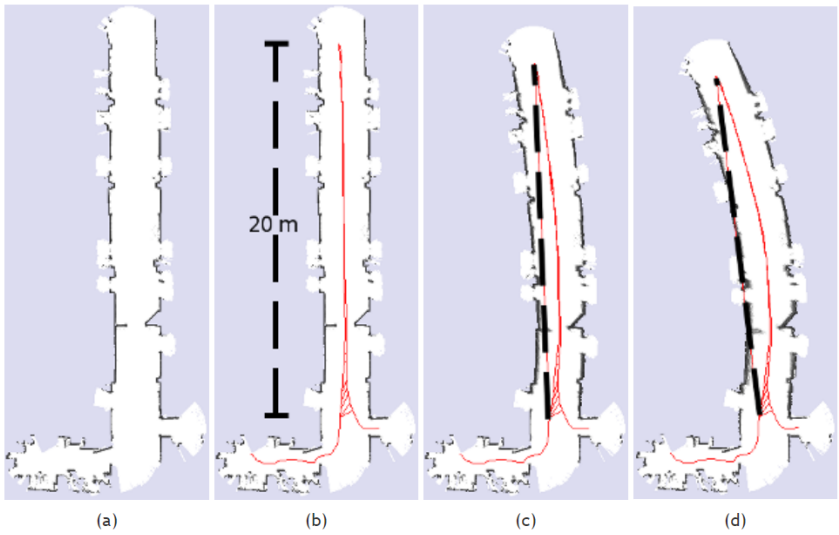


Figure 4.11: Relative motion error effect [8]

In an effort to mitigate this issue, we will be comparing the SLAM solutions based on a number of different metrics. The first one will be applied by using the *SLAM Benchmarking* Tool. Researchers in [8] believe they have developed a metric that is able to objectively represent the performance of different SLAM systems, regardless of the type of techniques they use, by basing the performance of the SLAM system on the error between the robot local poses perceived by the SLAM solution and the actual robot poses (ground truth). Specifically, this benchmark is based on comparing the relative ground truth motions, $\delta^*_{i,j}$, between the robot poses i and j and the relative

motion perceived by our SLAM algorithm, $\delta_{i,j}$. Thus, equation 4.7 renders the benchmark, where N is the number of relative motions (relations) that occurred within a given time frame.

$$\varepsilon(\delta) = \frac{1}{N}\sum_{i,j}(\delta_{i,j} - \delta^*_{i,j}) \tag{4.7}$$

This relative motion can actually be split into the translation and rotation that may have happened between i and j.

$$\varepsilon(\delta) = \frac{1}{N}\sum_{i,j}(trans(\delta_{i,j} - \delta^*_{i,j}) + rot(\delta_{i,j} - \delta^*_{i,j})) \tag{4.8}$$

The usage of this tool requires two input files. A SLAM log file containing all of the robot poses returned by our SLAM solution and a relations file that includes the ground truth relative translations and rotations. To achieve this, the node *slam_benchmarking*, contained in the *slam_benchmarking.py* python script, was created to calculate the relative ground-truth motion of the wheelchair and to monitor all the robot poses returned by SLAM. The robot poses are recorded by obtaining the transform between the map frame and the odometry frame, seen in figure 4.3, which is published by the SLAM solution chosen. For the ground truth relations, similar to what was done for adding the noise into the odometry data in equation 4.3, the *libgazebo_ros_p3d.so* plugin is used to obtain the robot poses i and j which are then used to calculate the displacement and rotation between the two robot poses. Also, the relations recorded are based on the relative motion that occurs every 0.5 seconds.

However, as will be seen when discussing the results, this metric also has its drawbacks, especially, in large maps. since this metric focuses on errors occurring locally. Over time, the motion errors may accumulate and cause inaccurate results in both the mapping and localization results of SLAM.

So, another metric was introduced based on a global reference. Specifically, this benchmarking technique is based on the error, $\varepsilon(r)$, between the ground truth position of the wheelchair in relation to a global frame, $r_{t,t}$, the map frame, and the robot position estimate returned by the SLAM solution, $r_{s,t}$, at time t, rendered by equation 4.9. Since these errors are based on a global reference, the accumulated error over time will be taken into account.

$$\varepsilon(r) = \frac{1}{N}\sum_{t=0}^{t}|r_{t,t} - r_{s,t}| \tag{4.9}$$

$$\varepsilon(r) = \frac{1}{N}\sum_{t=0}^{t}(x_{t,t} - x_{s,t}, y_{t,t} - y_{s,t})| \tag{4.10}$$

The node responsible for calculating this error is also the aforementioned *slam_benchmarking* node contained in the *slam_benchmarking.py* python script. Similar to how it performs to generate the benchmarking files for the *SLAM Benchmarking* tool, it keeps track of all the ground truth robot positions returned by the *libgazebo_ros_p3d.so* ROS plugin and the robot SLAM estimated positions, returned by the map frame to odometry frame transform. The positions recorded along

with the difference between the SLAM estimate and the ground truth positions are then saved in the *compare_positions.txt* file which is then parsed by the same python script using the pandas' python library. Using this library the mean of the error between the two saved positions is calculated and used as the final benchmarking metric.

It should be noted that the transform between the map and robot frame only monitors the SLAM perceived poses in real-time. Full-SLAM solutions, which is the case for all of the solutions considered here, except for LaMa 2D and Hector SLAM, when loop closing, have the ability to perform adjustments on both the perceived global path of the robot and the map recorded based on the optimization occurring in the loop closing phases. Unfortunately, both Karto SLAM and Gmapping do not publish this information to ROS topics, and, in an effort to maintain the analysis performed equal across all of the solutions, it was chosen to not include these adjustments into the recorded data. Since this metric uses the transform named, the errors that are generated do not consider the adjustments to the robot poses made in the past. Essentially, the metric analyses all of the SLAM solutions as Online-SLAM solutions, so both metrics will not reflect the adjustments that may occur on the Full-SLAM solutions. As such, the maps will also be analyzed visually.

Finally, the computational resources consumed by these solutions, during the deployment, will also be evaluated using the *psrecord*[9] python package, which allows the monitoring of CPU (%) and RAM consumption of given Linux programs.

In sum, 2 metric SLAM benchmarking metrics are used to compare the performance of the SLAM solutions, one based on the relative motion perceived by the SLAM solutions and another based on the global error of the robot position also perceived by the SLAM solution. The fact that there is access to the ground truth robot positions makes the usage of these benchmarking metrics quite simple, which would not be the case if these experiments were done in a real-life scenario, where the true robot positions would most likely need to be tracked "by hand".

4.4 Path Planning

Having implemented the wheelchair model along with the SLAM system responsible for our robot's mapping and localization needs, our wheelchair is now ready to perceive the world models described in 4.2. However, there still remains the issue of how it will intelligently navigate its environment. To do this, a modified version of the ROS 2D Navigation Stack, contained in the *move_base* node, developed by researchers[10] in the Intellwheels project, was used.

The navigation stack is fairly simple on a conceptual level. As can be seen in figure 4.12, it receives a navigation goal (like most data in ROS, this is sent through a topic, specifically to the */move_base/simple_goal* topic) and uses data from the odometry source, exteroceptive sensors, and the global and local cost maps to create a path to the navigation goal. Finally, to command the robot to follow the path the *move_base* node publishes velocity commands to the */cmd_vel* topic.

[9] https://pypi.org/project/psrecord/
[10] https://github.com/siferati

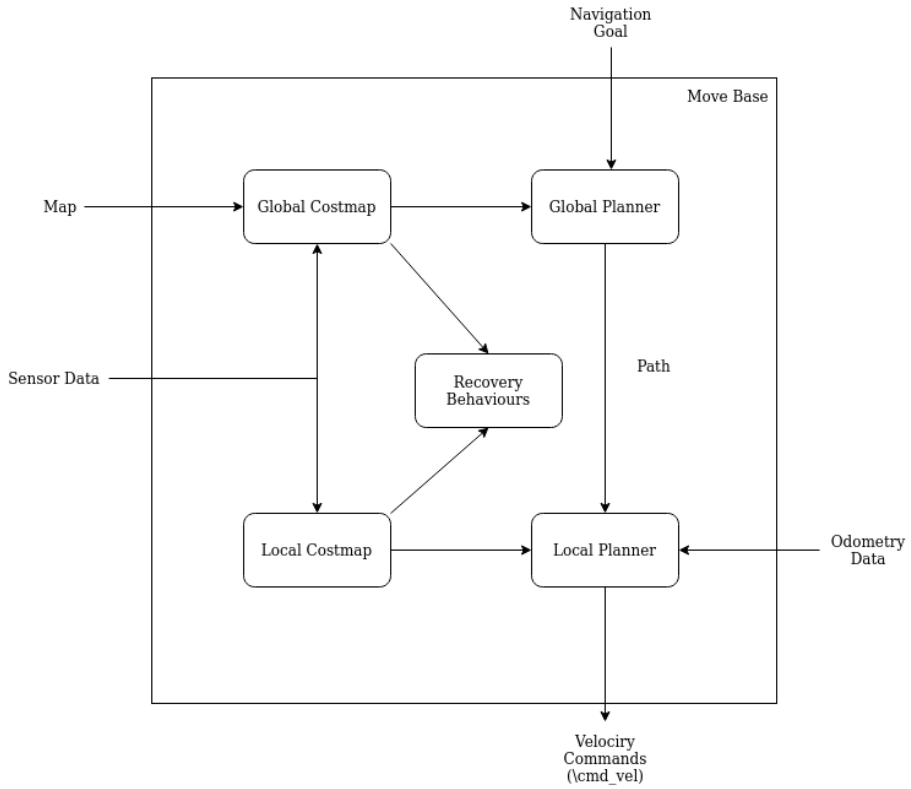


Figure 4.12: Navigation stack diagram

4.4.1 Cost maps

In a nutshell, cost maps are created by using sensor and map data (coming from the SLAM system used) to generate, in this case, a 2D occupancy grid that contains the cost (difficulty) of traversing different areas of a given map. For example, in the case of a ground robot working on rough terrain, the cost map would be a 2D cost map with lower values where the ground is flat and safe and the higher cost values would be associated with areas where the ground is rough/sloping. In the case of the modeled environments used in this project, the only obstacles available in the environments are the walls created in the worlds, so the higher values here are associated with the exact position of the walls and they decrease the further we are from those walls. The values held in a cost map are usually abstract and don't directly represent any measurement of the world, they are simply used to guide a route planning algorithm to find efficient and safe routes across the ground. An example of a cost map of the aforementioned small room world is in figure 4.13, where the colors surrounding the buildings represent the level of cost decreasing the further away we are from the walls.

4.4.2 Navigation Planners

The planners shown in 4.12 calculate the best path to the goal provided, taking into account the values of the cost map. There are two types of planners, global and local planners.

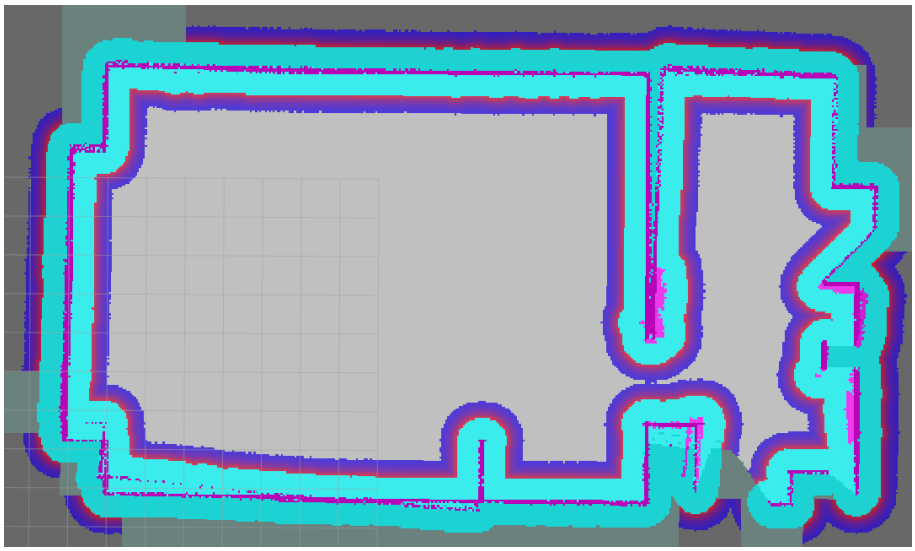


Figure 4.13: Cost map of small room world model

4.4.2.1 Global Planner

The global planner is what actually creates the specific plan from the current position of the wheelchair to the navigation goal that is given. The global planner developed uses the ROS *global_planner* package, which enables the application of either Dijkstra's algorithm or A* on the created 2D occupancy grid cost maps to find the minimum cost plan from the start position to the end goal. In this case, the algorithm used is Dijkstra's.

4.4.2.2 Local Planner

Given the plan created by the global planner, the local planner is what generates the velocity commands to the wheelchair in order to keep it on the path. The local planner used here is the *teb_local_planner*[11] ROS package. Using this package, the initial trajectory created by the global planner is optimized during run time in order to minimize the trajectory execution time, separation from obstacles, and compliance with the max angular and linear velocities that are set.

4.4.2.3 Recovery Behaviours

Finally, the last element in a navigation stack is the recovery behavior. Whenever the wheelchair is perceived as being stuck or cannot obtain a path to the desired location the /move_base package allows for the implementation of recovery behaviors that try to fix the situation before actually just quitting on the objective. There can also be multiple recovery behaviors that are activated sequentially. In the implementation used, whenever the wheelchair is stuck it performs multiple 360º rotations, If this does not work the navigation simply gives up on the goal and waits for a new one.

[11] http://wiki.ros.org/teb_local_planner

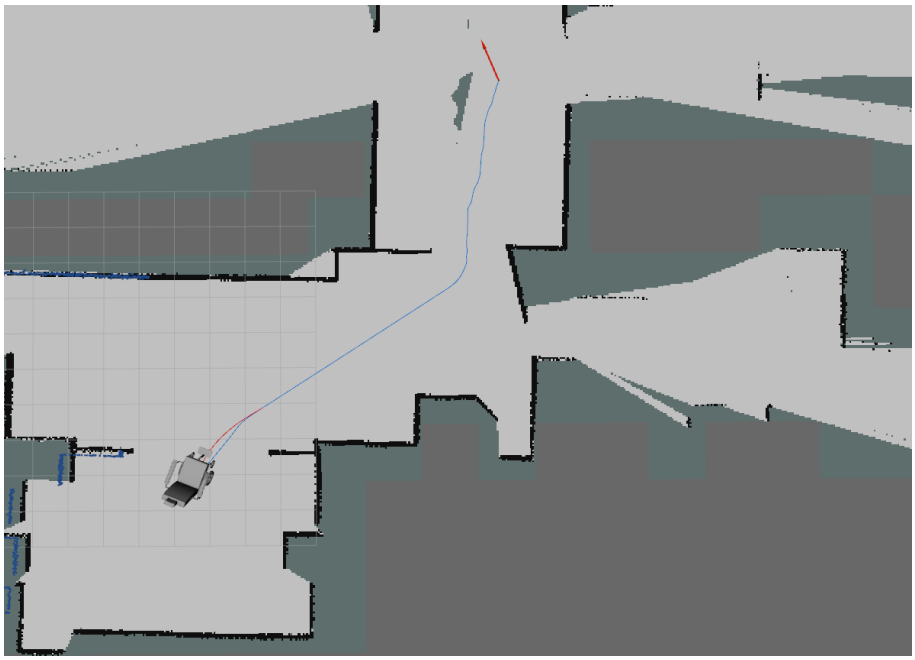


Figure 4.14: Navigation planners - local (red) & global (blue)

Again, the navigation stack used was already implemented by researchers in the Intellwheels project and it is fully available in this Github[12] repository.

4.4.3 Goal Creation

With the navigation stack properly set up, all that is required for the robot to navigate the environment is a goal. In some of the experiments conducted, the exploration ROS package *explorer_lite* was used to automatically generate goals based on a greedy frontier logic.

4.4.3.1 Explorer Lite

The *explorer_lite*[13] ROS package provides frontier based automated exploration of environments. With the package implemented in a robotic system, it makes use of the map information coming from the SLAM system to greedily explore the environment until no frontiers are found. This stands as the sister package to the *explorer* ROS package, but unlike the former, *explorer_lite* makes use of either the existing cost map of the navigation stack or SLAM map data, which makes it much easier to configure and more efficient (lighter on resources). In this case, the package was configured to take in the data coming from the SLAM system.

As we can see in figure 4.15, the package creates frontiers (blue lines), which is the line between the unknown area (dark grey area) and the already mapped areas (lighter grey). Based on the length of these frontier lines, the package generates goal messages to be read by the *move_base* node which, in turn, will generate a path from the position of the wheelchair to the goal and control

[12] https://github.com/siferati/intellwheels_nav
[13] http://wiki.ros.org/explore_lite

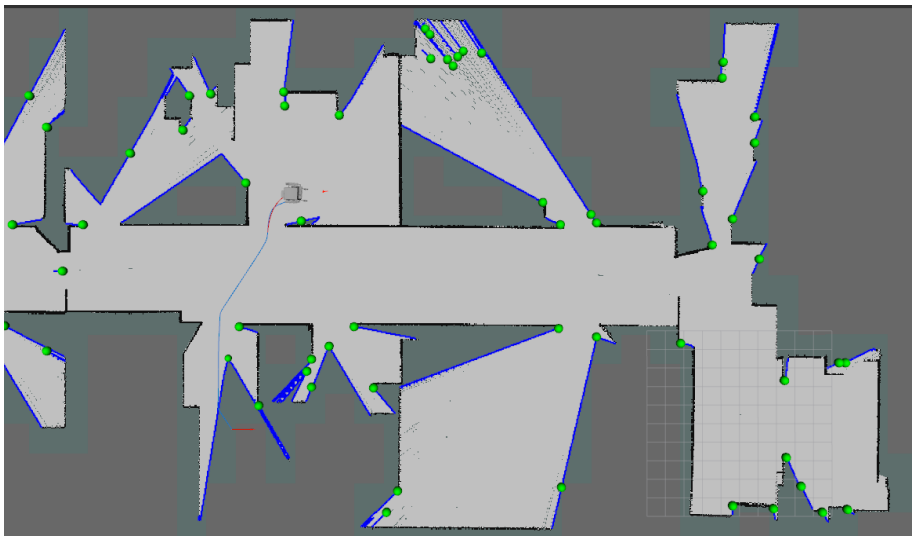


Figure 4.15: Frontier-based exploration example

said wheelchair until it reaches that position. Once that frontier has been explored the package will generate another goal based on the next biggest frontier on the map. This architecture is shown in figure 4.16.

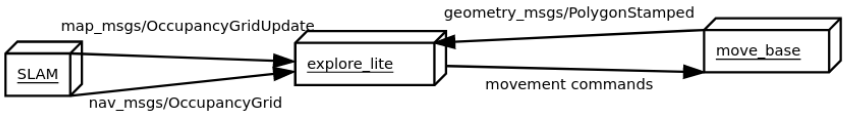


Figure 4.16: Explorer lite architecture

4.5 Optimization Process

In the second and final phase of this project, a steepest-ascent hill-climber was applied to optimize the sensor placement of one and two laser sensors in order to maximize the performance of one of the SLAM solutions picked from the previous phase. The objective function used is a modified version of the two previous SLAM benchmarking metrics developed.

To do this, it was required to develop and adapt the optimization algorithm to the Gazebo and ROS configurations, meaning that, since our objective function will be dependent on the data acquired (relative motion and global position) from the wheelchair completing a designed course in a created environment in Gazebo, each time the objective function is called, the actual optimization algorithm has to wait for the circuit to be completed. So, the *loop_master* node, included in the *loop_master.py* python script, was created and it's responsible for controlling each step of the optimization algorithm put in place. Additionally, exploring the state space is also not just a matter of simply changing variables like the position, orientation, and configuration of the sensors since they actually have to be changed directly in the wheelchair description files. The full optimization process is shown in pseudo-code 3.

Algorithm 3 Sensor Placement Steepest Ascent Hill-Climber Pseudo-Code

```
1: function SENSORPLACEMENTOPTIMIZER(ObjectiveFunction, StateSpace)
2:     rand_times = 0
3:     while rand_times < 80 do
4:         if rand_times != 0 then
5:             SaveHillclimbRun(visited_positions)
6:         end if
7:         rand_times += 1
8:         ClearPreviousRun(visited_positions)
9:         state = Generate random state within StateSpace
10:        while True do
11:            if NOT CheckVisitedPositions(visited_positions,state) == True then
12:                UpdateSensor(state)
13:                initial_bench = ObjectiveFunction(state)
14:                UpdateVisitedStates(visited_positions,state,initial_bench)
15:            else
16:                initial_bench = GetVisitedSolution(state)
17:            end if
18:            best_bench = initial_bench
19:            neighborhood = GetNeighborhood(state)
20:            while neighbor = neighborhood[i] != None do
21:                UpdateSensor(neighbor)
22:                neighbor_bench = ObjectiveFunction(neighbor)
23:                UpdateVisitedState(visited_positions,neighbor,neighbor_bench)
24:                if neighbor_bench < best_bench then
25:                    state = neighbor
26:                    best_bench = neighbor_bench
27:                end if
28:            end while
29:            if best_bench == initial_bench then
30:                break
31:            end if
32:        end while
33:    end while
34: end function
```

The code shown here is a modification of pseudo-code 1, designed to adapt to the previously discussed simulation conditions. First, a random state, contained in the limits of the variables set in table 4.1, is initiated. Depending on the state space, further discussed in sub-section 4.5.1, this can be the position and orientation of one or multiple sensors with various configurations (max range and FOV). Now, a new procedure was added to the optimization algorithm. Instead of risking the chance that the *state space* exploration may lead to already visited positions, the functions *Check-VisitedPositions()*, *GetVisitedPositions()*, *UpdateVisitedPositions()* were created to keep track of already visited states, saving each of them in the dictionary data structure, *visited_positions*. This was done to minimize the run-time of each iteration of the hill-climbing algorithm. So, if the *state* has not yet been visited then, the objective function is called.

The performance of the SLAM algorithm chosen will be our objective function, which is rendered in equations 4.11 and it stands as the weighted sum of our two previously implemented metrics, the error between the relative motion estimate, $\delta^*_{i,j}$ and the ground truth relative motion, $\delta_{i,j}$ plus the error between the global position estimate, $r_{s,t}$ and the ground truth position, $r_{t,t}$, given the position/orientation/configuration of 1 or 2 sensors, defined by the variable *state*. Naturally, as we want to maximize the performance of the SLAM system we want to find the *state* that minimizes both of these errors.

$$\varepsilon(state) = \frac{1}{2}\sum_{i,j} trans(\delta_{i,j} - \delta^*_{i,j}) + \frac{1}{2}\sum_{t=0}^{N} r_{t,t} - r_{s,t} \tag{4.11}$$

Again, this objective function can only be called after the wheelchair has completed its predefined course and gathered all of the required data, so the function itself ends up being responsible for an array of additional functionalities in the ROS framework. Specifically, once the objective function is called its first order of business is to initiate the Gazebo world and spawn the wheelchair model along with all of the required nodes responsible for the circuit to be completed and monitored. With everything spawned, the objective function will block the main algorithm until the circuit has ended and the *slam_benchmarking* node has created the file with the required data for the metric computation. Once all the data is gathered, the metric is calculated and the *state*, along with its corresponding metric is saved onto the *visited_positions* dictionary. On the other hand, if the *state* has already been visited then the *initial_bench* variable is updated with the corresponding *state* benchmark, using the *GetVisitedSolutions* function.

Following this, the *neighborhood* is obtained with the *GetNeighborhood* function, which should generate all of the available neighbors of the current *state* that are included in the defined state space.

4.5.1 State Space

In this case, the state space is defined as the possible positions/orientations that the sensor can take along with all of the possible configurations, considering a number of restrictions. Specifically, the variables that are included in the state space are the x,y positions of the sensor, its

orientation, their covered field of view, and finally, the max range of their laser readings (table 4.1).

Perimeter(m)	**Orientation(º)**	**FOV(º)**	**Range(m)**
[0.00, 0.72]	[-90, 90]	[40,180]	[4,20]
]0.72, 1.61]	[0,180]	[40,180]	[4,20]
]1.61, 2.33]	[90, 270]	[40,180]	[4,20]
]2.33, 3.22]	[180,360]	[40,180]	[4,20]

Table 4.1: Sensor placement state space

The positions that are tested all belong to a rectangular perimeter with a length of 3.22 meters defined around the wheelchair just above the wheels (figures 4.17 and 4.18), so, really, the x and y variables actually become one, the perimeter variable. The position of the sensor also causes some restrictions on the other variables, specifically, the orientation of the sensor, since, for example, it would not make sense to have a sensor placed on the front of the wheelchair but have it pointing backward. In figure 4.18 we have an example of a possible sensor position/configuration. The sensor is sitting in the perimeter position 1.20 meters with a 90º orientation, a field of view of 180º, and a max range of 20 meters.

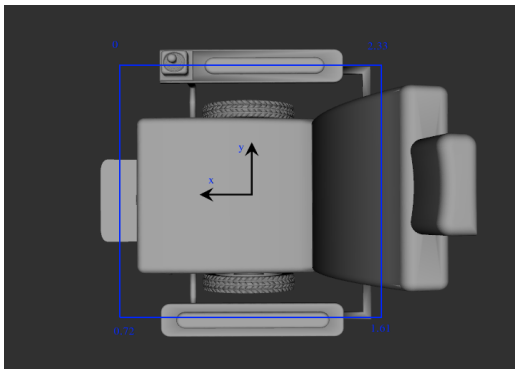


Figure 4.17: Sensor placement wheelchair perimeter

Figure 4.18: Sensor side position (1.30,90,180,20)

The *GetNeighborhood()* function returns all of the possible neighbors of the *state*. Each *neighbor* is defined as a step-sized positive or negative iteration of one of the variables in the current *state*. For example, a possible *neighbor* of the (1.30,90,180,20) *state* would be (1.40,90,180,20) as it stands as a positive iteration to the perimeter variable. So, since we have 4 changeable variables each with the possibility to offer 2 neighbors ($\pm$ *step_size*) there are, most of the time, 8

possible neighbors if we are using only 1 sensor. In the case of 2 sensors, this number doubles to 16 possible neighbors as the number of sensor variables also doubles.

$$neighbor = state \pm step\_size \tag{4.12}$$

Variables	Step Size
Perimeter(m)	± 0.1
Orientation(º)	± 10
FOV(º)	± 10
Range(m)	± 2

Table 4.2: Sensor placement state space step size

Note that each the time the objective function is called, the tags associated with these sensor parameters in the description files of the wheelchair have to be changed. So, the *update_sensor* function is always called before the objective function. This function exploits the fact that the xacro URDF type files can import dictionary data structures from Yet Another Markup Language (YAML) files. Instead, of parsing the main configuration files each time a sensor parameter is changed, the *update_sensor* simply creates a new *sensor_config.yaml* file with a dictionary containing all of these sensor parameters. So each time the wheelchair is spawned, the configuration files import the sensor parameters from the *sensor_config.yaml*.

Finally, the objective function is called for all the possible neighbors, and their solutions are compared with the *initial_bench*. Since this is the steepest-ascent version of a hill-climber algorithm, all of the neighbors are explored before moving on to a new *state*. If one of the neighbors has a lower cost value (in this case, the aim is to minimize the objective function), then that *neighbor* becomes the new current *state*. This iteration of the hill-climber continues until *initial_bench* is the minimum value found among the neighbors. When this happens, this solution is saved as the best solution found in this iteration of the hill-climber.

As mentioned in section 2.5.1.1, the hill-climber algorithm has a chance to get stuck on local minima, so to mitigate this, *loop_master* initiates a new hill-climber iteration each time one has ended it until it has ran at least 80 times.

4.6 Summary

Conducting experiments under a real-life scenario can be quite a costly operation, both in terms of monetary and time costs. This is true in the context of this dissertation, especially since an optimization algorithm is employed to recursively change the positions, orientations, and configurations of multiple sensors in a wheelchair model. With the number of variables considered, it simply would not make sense to perform this optimization in the real wheelchair. So, taking advantage of previous research done in the Intellwheels project, a simulated environment was designed with the purpose of performing the planned experiments while keeping the aforementioned

costs to a minimum and also maintaining a decent degree of fidelity to the hardships that a robotic system may encounter in a real-life scenario.

First, a number of 2D-SLAM algorithms were implemented, into the existing wheelchair model and navigation stack, along with a frontier-based exploration package used for goal generation. To analyze these algorithms a number of metrics based on relative motion errors and global position errors were also implemented.

To accentuate the differences between the SLAM solutions' performances an odometry noise model was added, as the original differential driver plugin did not add enough noise to the odometry data published, for the SLAM solutions to generate any errors. Furthermore, a number of distinguishable world models, designed using laser scan data gathered on real-life environments, were created in order to analyze the performance of the given SLAM systems, under different conditions.

Lastly, to perform the actual sensor placement optimization, a steepest hill-climbing algorithm, using the benchmarking metrics as the objective function, was adapted to the ROS framework.

Chapter 5

Experiments and Results

Given the simulation environment properly set up, along with the required tools needed to monitor the performance of the SLAM solutions, it is now possible to discuss the experiment procedures and results. The objective of these experiments was to use the benchmarking metric developed to compare and analyze the performance of the mentioned 2D SLAM solutions in distinct navigation scenarios, and to find the optimal place/configuration of sensors that maximizes the performance of said solutions by employing a steepest ascent hillclimber algorithm. For reference, all of the experiments detailed were conducted in an Acer Aspire V5-591G-55T2 laptop with an Intel Core i5-6300HQ CPU @ 2.30GHz processor and 8GB of RAM running on Linux's Ubuntu 18.04.

This chapter is then divided into two sections, each of them dedicated to each of the experiments, detailing both how they were conducted and discussing their results.

5.1 2D SLAM Analysis

To compare the SLAM solutions between each other, the first thing done was to adjust the most relevant parameters of each of these solutions to be as close as possible, namely, the max range of the laser scan readings, and the accumulated translation, and rotational motions required before each scan processing procedure takes place. In this vein, all of the SLAM solutions were adjusted to only take laser readings that have no more than 18 meters of range, in the two smallest worlds, and then changed to 25 meters for the biggest world. The accumulated translations and rotations, before scans are processed, were adjusted for 0.5 meters and 0.25 radians, respectively, for each of the SLAM solutions. Additionally, for the particle filter SLAM solutions, LaMa Particle Filter, and Gmapping, the number of particles was adjusted to 30. Regarding the actual wheelchair, it will move at a max linear velocity of 0.4 m/s and an angular velocity of 0.3 m/s.

In this phase only one sensor was used, supported by a FOV of 180º and a max range of either 18 meters or 25 meters, depending on the testing environemnt. Additionally, it was placed in the front of the wheelchair pointing in the same direction as the forward movement direction of the wheelchair as shown in figure 5.1.

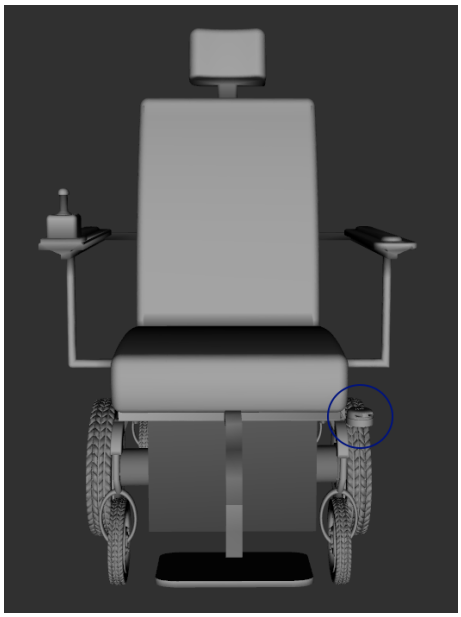


Figure 5.1: Sensor position for SLAM comparison

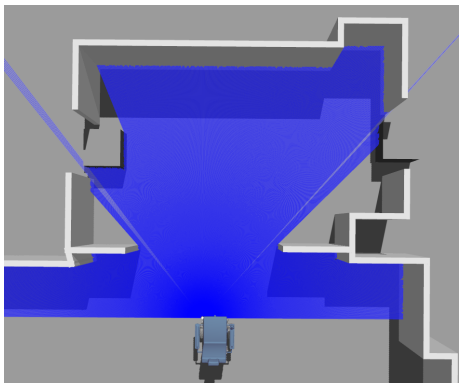


Figure 5.2: Sensor scan example for SLAM comparison

As explained in section 4.3.1, three different benchmarking metrics will be used. The first, based on the relative motion of the robot (translation and rotation error), should be able to interpret if the SLAM solution is able to faithfully maintain the correct topology of the environment. The metric based on the global position errors between the SLAM solution and the ground truth should, in this case, represent the accuracy of both the localization and mapping processes in a less forgiving way as the quality of the map will be taken into account. The last metric will be based on the computational resources consumed by the SLAM solution during its run-time.

With the solutions used adjusted, they were applied to map and explore the three main worlds models discussed in section 4.2, where two different types of navigation scenarios were used, one based on the exploration package *explorer_lite* and another based on teleoperation of the wheelchair model.

5.1.1 Exploration Circuits

In the first case, the exploration package does not guarantee the exact same experiment circumstances to be performed, since it generates goals based on the highest value frontier, which in turn is obtained from the map generated by the SLAM solution. So, in this case, goals generated from the exploration package may be different which will also cause the path generated by the navigation stack to also be different. Even though the same experiment conditions will not be maintained between the SLAM solutions (since their navigation goals will likely be different), this is still an interesting experiment to conduct since this exploration phase is a very likely real life scenario.

As an example, let's imagine a setting where the wheelchair robotic system is deployed in a hospital for transportation of patients that are unable to sufficiently operate the wheelchair.

The first other of business, before actually performing this navigation, would be to deploy the wheelchair with an exploration algorithm combined with a SLAM solution and let it explore and map the hospital environment. With this example, it is possible to see why analyzing the performance of SLAM solutions under this scenario could be useful.

The wheelchair system with the different SLAM solutions and the exploration package were then deployed into the 3 different worlds. Also, again, in an effort to maintain experiment circumstances for all of the SLAM solutions tested, each exploration cycle was limited to a certain time-frame, meaning once the robot exploration begins it has a limited time-frame to perform said exploration. So, the smallest of worlds, the Univ. of Freiburg Building 79 was restricted to 20 min, the second largest one, the Intel's Research Lab, to 30 minutes, and, at last, the largest, ACES Building, at 40 min.

In figures 5.3, 5.4 and 5.5 we can see the actual path completed by the wheelchair compared with the path completed perceived by the SLAM solution and the path returned solely by the odometry data. With this, it's possible to see if there is a difference between both paths and also conclude that the odometry noise model was successfully implemented as it is unreliable in long spans of time.

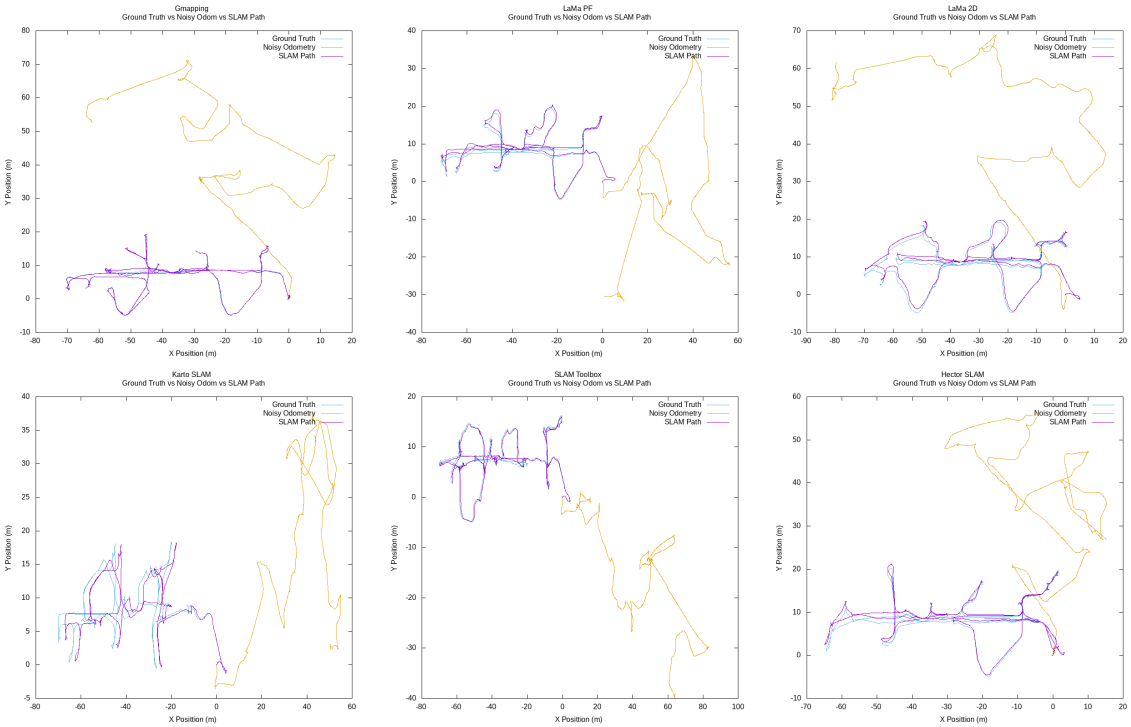


Figure 5.3: SLAM Vs ground truth Vs odometry paths in Frei079 during exploration circuit

5.1.1.1 Relative Motion Metric

We can see in table 5.1, that the metric based on relative motion, stays pretty consistent across all of the SLAM solutions, except for Hector SLAM in the ACES building model. Apart from

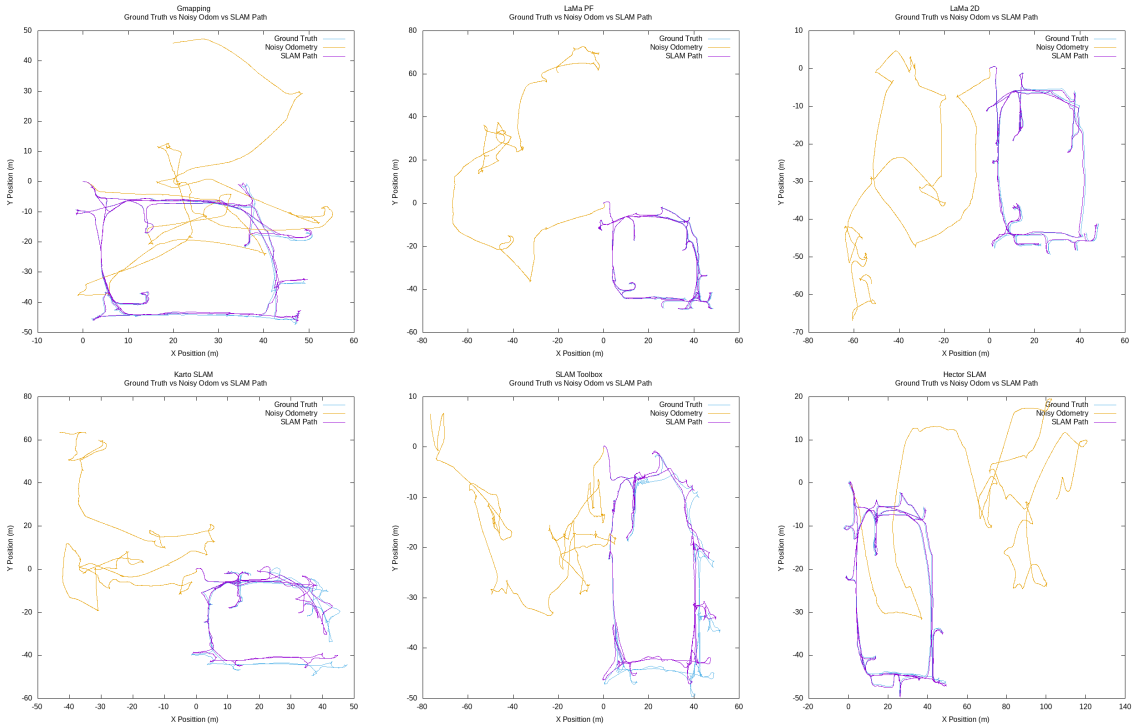


Figure 5.4: SLAM Vs ground truth Vs odometry paths in Intel during exploration circuit

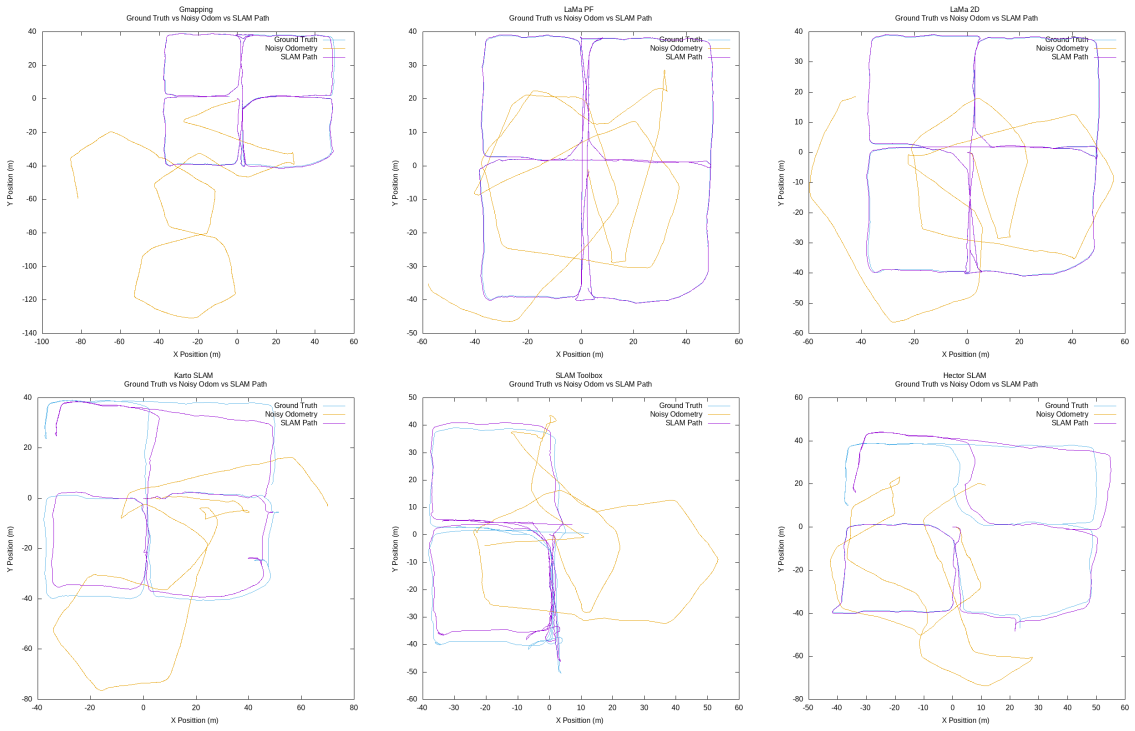


Figure 5.5: SLAM Vs ground truth Vs odometry paths in ACES during exploration circuit

this outlier, this means that all solutions should be able to keep up with the relative motion that occurs in the wheelchair with a mean error value in the low centimeters range, in the case of the

translation error, and in the low radians range (all mean rotational errors are below 0.010). The translation errors also seem to get higher across all of the solutions when comparing the results from the ACES building with the other two. This is most likely due to the original topology of the world being mostly composed of extremely long corridors which could be taking its toll on the performance of the scan matching processes of the SLAM solutions.

As can be seen in figures, 5.3, 5.4 and 5.5 even though, in some cases, there is clearly some displacement between the ground truth path and the SLAM path, the length of both paths stays mostly the same. However, this metric may not be fully representing the performance of the solutions for a number of reasons. Since it focuses on local errors that occurred, it provides no data about the fact that the SLAM solutions may or may have not been able correct the accumulated error that is being produced in each motion. This could cause inconsistencies in the generated map that may inhibit its later usage for navigation.

Translation Error (m)	Gmapping	LaMa PF	LaMa 2D	Karto SLAM	SLAM Toolbox	Hector SLAM
Frei079	0.04 ± 0.02	0.04 ± 0.02	0.04 ± 0.02	0.03 ± 0.02	0.04 ± 0.02	0.04 ± 0.02
Intel	0.04 ± 0.02	0.04 ± 0.02	0.04 ± 0.02	0.04 ± 0.02	0.04 ± 0.02	0.04 ± 0.02
Aces	0.05 ± 0.01	0.07 ± 0.01	0.05 ± 0.01	0.05 ± 0.01	0.05 ± 0.02	0.11 ± 0.15
Rotational Error (rad)						
Frei079	0.06 ± 0.08	0.06 ± 0.08	0.07 ± 0.09	0.07 ± 0.09	0.09 ± 0.16	0.06 ± 0.08
Intel	0.06 ± 0.08	0.07 ± 0.09	0.07 ± 0.09	0.07 ± 0.09	0.09 ± 0.12	0.05 ± 0.07
Aces	0.04 ± 0.05	0.03 ± 0.06	0.05 ± 0.03	0.04 ± 0.06	0.04 ± 0.06	0.04 ± 0.06

Table 5.1: Relative motion metric results of the exploration circuits

5.1.1.2 Global Position Metric

In the errors generated based on the the global position metric, which are displayed in table 5.2, there are some much more clear discrepancies among the performance results of each solution.

X-Axis Position Error (m)	Gmapping	LaMa PF	LaMa 2D	Karto SLAM	SLAM Toolbox	Hector SLAM
Frei079	**0.11 ± 0.10**	0.28 ± 0.16	0.50 ± 0.19	2.19 ± 0.81	0.33 ± 0.19	0.12 ± 0.09
Intel	0.27 ± 0.23	**0.26 ± 0.23**	0.51 ± 0.31	1.55 ± 1.13	1.12 ± 0.72	0.29 ± 0.24
Aces	0.33 ± 0.37	**0.087 ± 0.095**	0.14 ± 0.23	2.3 ± 1.5	0.95 ± 1.2	2.6 ± 2.4
Y-Axis Position Error (m)						
Frei079	**0.10 ± 0.12**	0.68 ± 0.36	0.70 ± 0.41	0.11 ± 0.08	0.24 ± 0.18	0.53 ± 0.40
Intel	0.40 ± 0.34	**0.14 ± 0.10**	0.23 ± 0.16	1.30 ± 1.20	1.40 ± 0.85	0.23 ± 0.17
Aces	0.25 ± 0.32	**0.092 ± 0.09**	0.14 ± 0.11	2.0 ± 2.01	2.8 ± 1.11	2.0 ± 2.0

Table 5.2: Global position metric results of the exploration circuits

Looking at the smallest of the worlds, the better performing solution was Gmapping, although, except for Karto SLAM, the remaining solutions were all very interchangeable. The high error values produced in the case of Karto SLAM appear, in the quality of the map produced, as overlapping walls, as we can see in figure 5.7. Although this inconsistency exists, it could be argued that the map is more than sufficient to be used in navigation.

In the following world, Intel's research lab, LaMa's particle filter version was the better performing solution, while Karto SLAM and SLAM Toolbox offer the worst results. Again, these

Figure 5.6: Gmapping map generated in Frei079 during exploration circuit

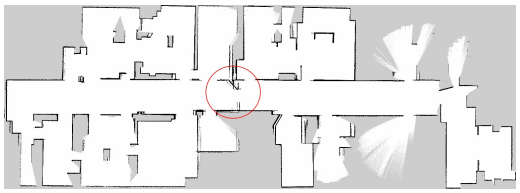
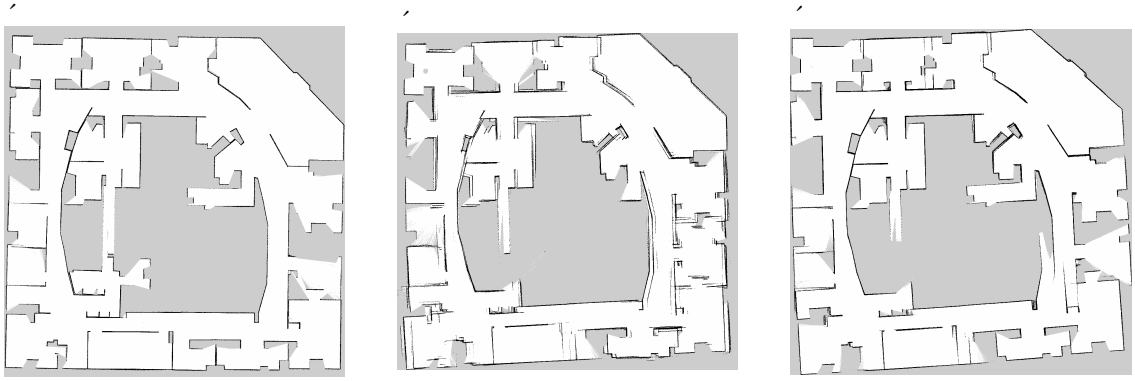


Figure 5.7: Karto SLAM map generated in Frei079 during exploration circuit

high values are both translated in the difference between the global path taken (seen in figure 5.4) but also through the quality of the maps generated. Looking at the maps in both figures 5.8b and 5.8c it is also possible to see some of the apparent drawbacks of using this metric. By comparing these two maps with the one generated by LaMa's particle filter, in figure 5.8a, we can see that the two worst ones seem to be slightly rotated. This is one of the issues of using a performance metric based on a global fixated frame. While from a global perspective that rotation represents a feasible decrease in quality of the mapping and localization, it could be argued that, if that was the only inconsistency with the map, both maps could still be used to perform navigation.

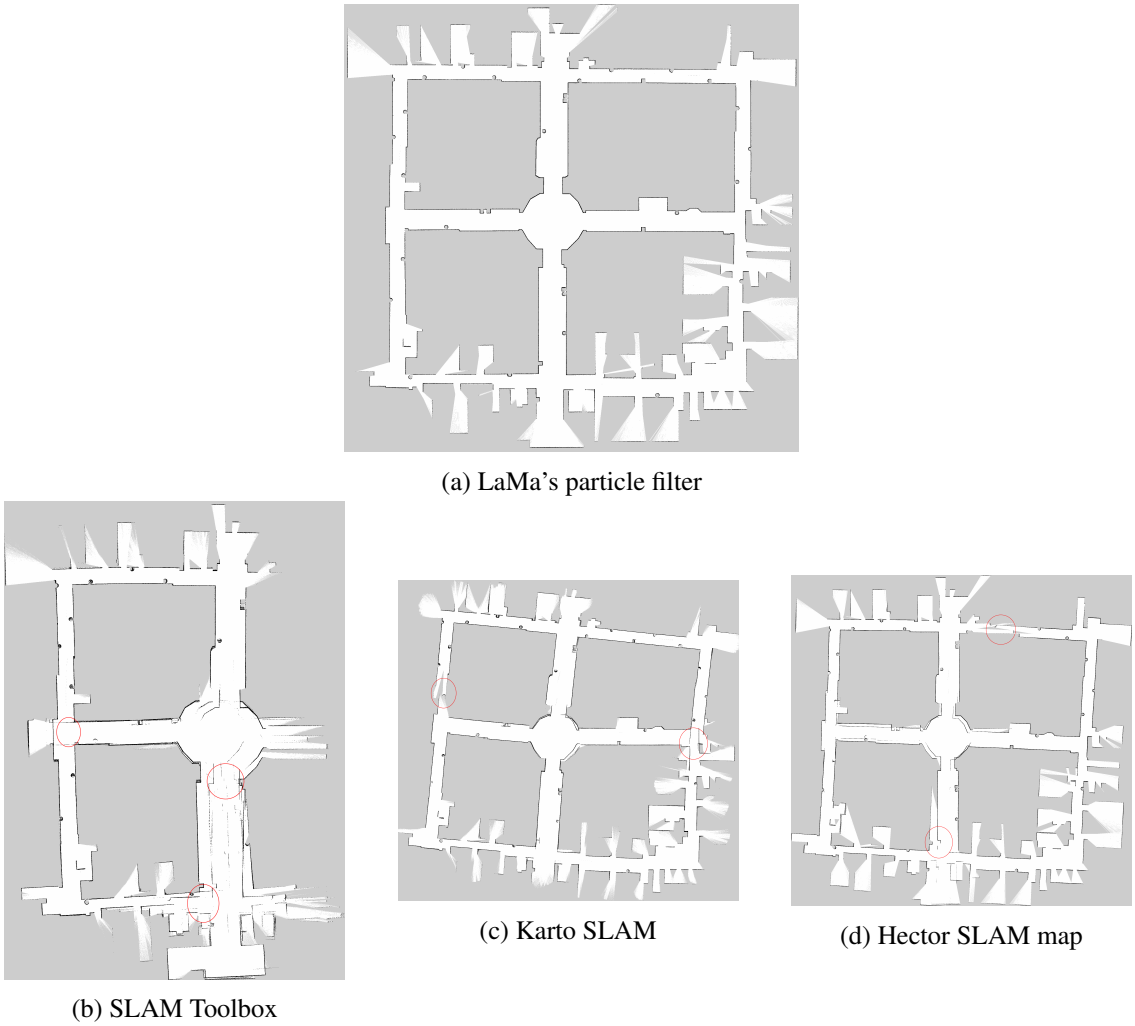


(a) LaMa's particle filter (b) SLAM Toolbox (c) Karto SLAM

Figure 5.8: Intel maps generated during exploration circuit

Regardless of the quality of the map, all of the SLAM solutions, up until now, were able to fully explore and map the worlds models, to an acceptable degree. However, in the last and largest world, the ACES building, some of the SLAM solutions fell short of this capability. In this last model, only Gmapping, LaMa PF, and LaMa 2D were able to faithfully explore and create an acceptable map, with LaMa's particle version having the better results. In the case of the last remaining solutions, the quality of the maps being generated during run-time ended being poor to the point that the environments were unable to be fully explored, as seen in the following figures 5.9b, 5.9c and 5.9d. From the mapping errors seen in the figures, the overlapping walls included in the maps have made it impossible for the wheelchair to correctly traverse the map as both the exploration algorithm and the navigation stack make use of this map to employ proper navigation. In the case of the graph-SLAM solutions, Karto SLAM, and SLAM toolbox, this is most likely due to increased need in loop closing of the solutions, which in this environment, the opportunity to

perform this process is greatly reduced due to the long corridors and lack of connection between them. Hector SLAM, offers no loop closing solution so the accumulated error gathered in said corridors likely caused this end result and its scan matching may not be as reliable as the other Online SLAM solution, LaMa 2D.



(a) LaMa's particle filter

(b) SLAM Toolbox

(c) Karto SLAM

(d) Hector SLAM map

Figure 5.9: ACES maps generated during exploration circuit

From these results, both particle-filter-based solutions seem to be the more robust of the ones that were picked. Consistently, Karto SLAM was the worst-performing solution across all of the exploration phases, most likely due to its hard-coded limitation, of only accepting sensor readings below the length of 12 meters. In actuality, both graph-based solutions do not seem good choices in a scenario like this, as they are both reliant on loop closure for graph optimization. The exploration algorithm used does not take this into account, since it is solely focused on exploring the areas of the map with a higher value frontier which may not include a navigation path that actually has loop-closing opportunities. Regarding the Online SLAM solutions, both have similar performances on the smallest maps, however, Hector SLAM fell short on mapping the biggest.

5.1.2 ROSBag Circuits

In this second navigation experiment, the performance of the SLAM solutions is tested in a much more controlled scenario, where the wheelchair was teleoperated using the *teleop_twist_keyboard*[1] ROS package. In this case, each of the environments was explored "by hand" while all the ROS topics regarding the odometry, ground truth, and laser scan data were saved into ROSbags[2]. ROSbag is a set of tools that enables the recording and the playing back of data stored in ROS topics. By using this tool we are able to replay the ROSbags gathered using the different SLAM solutions which, in a nutshell, enables all of the SLAM solutions to be tested using the exact same data. Also, by using a predefined route controlled by the user, the dependency on the map generated during navigation for the navigation itself is negated, which will likely make it easier for the solutions to perform loop closing in a case where the map generated may inhibit that (which is what happened in the last environment in the exploration circuits). Essentially, we are performing what is called offline SLAM. The respective ground truth, odometry, and SLAM paths are shown in figures 5.10, 5.11 and 5.12. As seen, the ground truth and odometry data are exactly the same for each of the testing worlds.

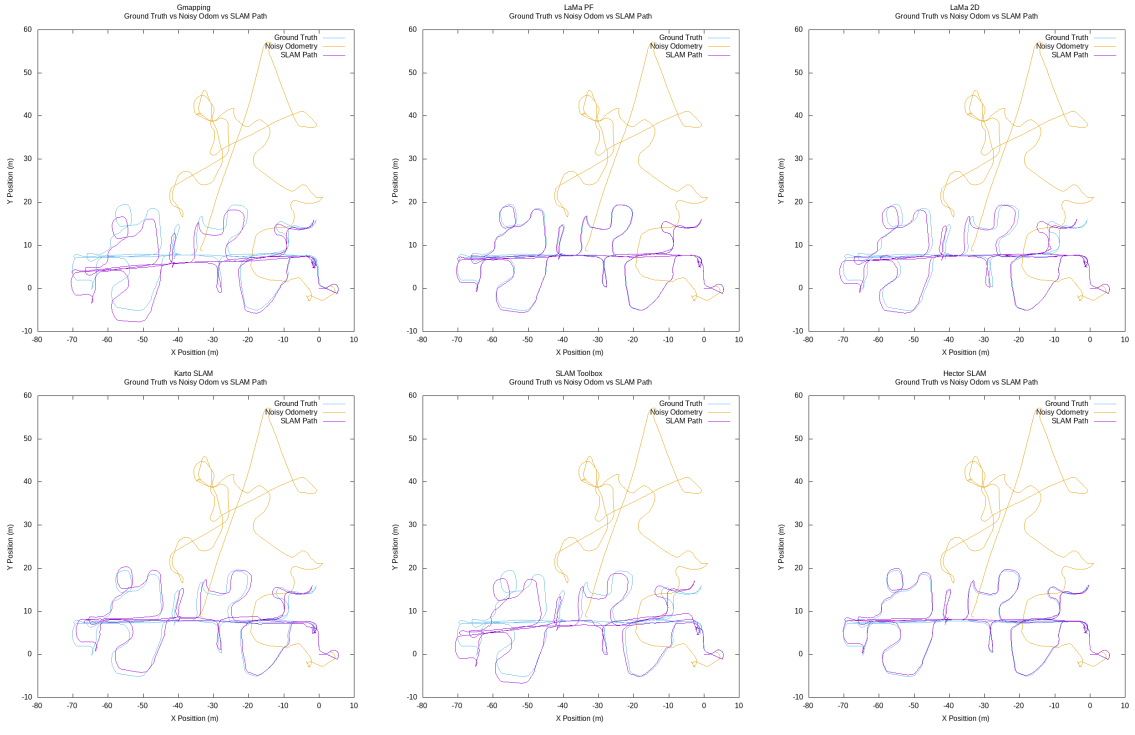


Figure 5.10: SLAM Vs ground truth Vs odometry paths in Frei079 during ROSbag circuit

[1] http://wiki.ros.org/teleop_twist_keyboard
[2] http://wiki.ros.org/rosbag

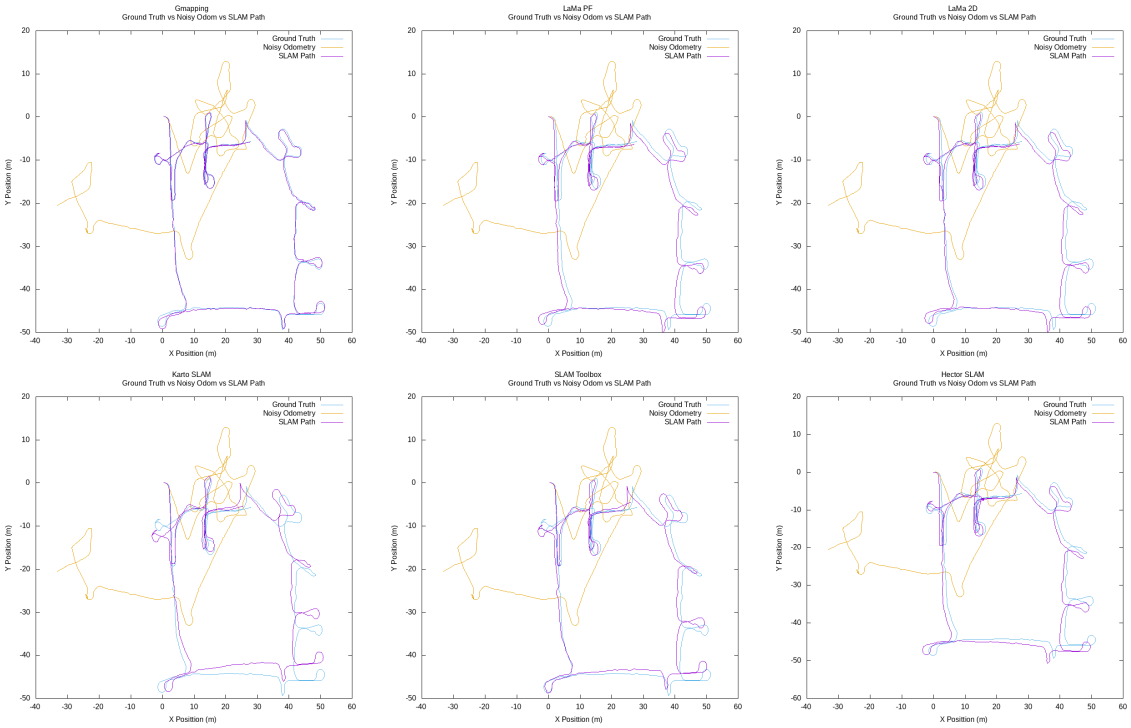


Figure 5.11: SLAM Vs ground truth Vs odometry paths in Intel during ROSbag circuit

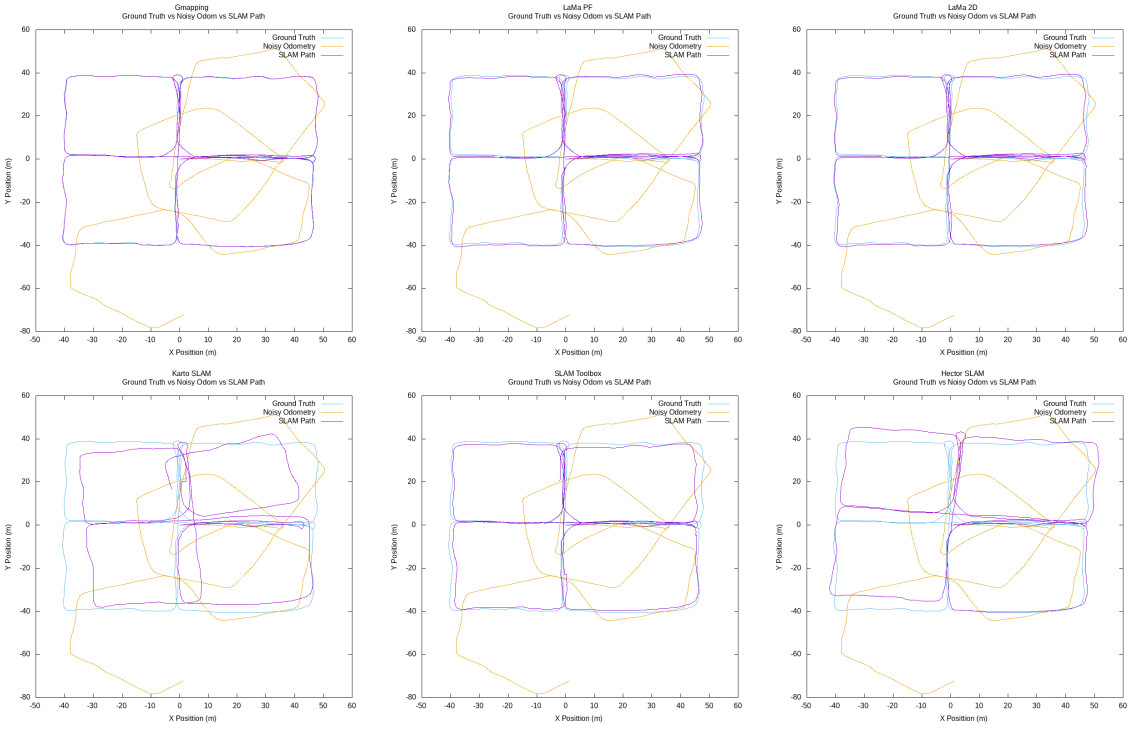


Figure 5.12: SLAM Vs ground truth Vs odometry paths in ACES during ROSbag circuit

5.1.2.1 Relative Motion Metric

The results obtained from this metric shows us mostly the same as the other experiment as all of the solutions have similar results in each of the environments tested. By using the same data we are able to conclude here that all of the solutions show similar performance in predictions made about the relative motion of the wheelchair. This similarity in the results both across the ROSbag and the exploration circuits could also be suggesting that the noise added in both the odometry data and laser scan data is not sufficient to see any difference between the local SLAM solution's motion predictions.

Translation Error (m)	Gmapping	LaMa PF	LaMa 2D	Karto SLAM	SLAM Toolbox	Hector SLAM
Frei079	0.05 ± 0.02	0.05 ± 0.01	0.05 ± 0.01	0.05 ± 0.01	0.05 ± 0.01	0.05 ± 0.02
Intel	0.05 ± 0.01	0.05 ± 0.01	0.05 ± 0.02	0.05 ± 0.01	0.05 ± 0.01	0.05 ± 0.02
Aces	0.05 ± 0.02	0.05 ± 0.01	0.05 ± 0.01	0.05 ± 0.01	0.05 ± 0.01	0.05 ± 0.02
Rotational Error (rad)						
Frei079	0.05 ± 0.09	0.05 ± 0.09	0.05 ± 0.09	0.05 ± 0.09	0.05 ± 0.09	0.05 ± 0.09
Intel	0.06 ± 0.09	0.06 ± 0.09	0.06 ± 0.09	0.06 ± 0.09	0.06 ± 0.09	0.06 ± 0.09
ACES	0.01 ± 0.03	0.02 ± 0.03	0.02 ± 0.03	0.01 ± 0.03	0.01 ± 0.03	0.01 ± 0.03

Table 5.3: Relative motion metric results of the ROSbag circuits

5.1.2.2 Global Position Metric

In table 5.4 it is possible to see the results of the global metric in the ROSbag circuits.

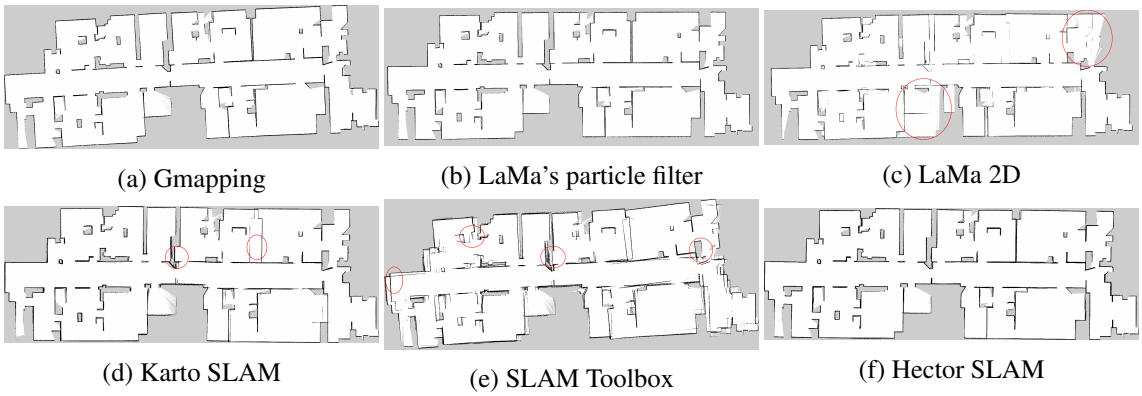
X-Axis Position Error (m)	Gmapping	LaMa PF	LaMa 2D	Karto SLAM	SLAM Toolbox	Hector SLAM
Frei079	0.49 ± 0.3	0.16 ± 0.11	1.7 ± 2.3	1.1 ± 1.1	0.94 ± 0.61	**0.11 ± 0.08**
Intel	**0.24 ± 0.14**	1.1 ± 0.37	0.96 ± 0.38	0.98 ± 0.75	0.95 ± 0.57	1.13 ± 1.11
Aces	**0.24 ± 0.24**	0.47 ± 0.37	0.47 ± 0.37	4.4 ± 3.4	0.9 ± 0.72	1.7 ± 1.3
Y-Axis Position Error (m)						
Frei079	1.7 ± 1.0	**0.21 ± 0.20**	0.27 ± 0.27	0.48 ± 0.37	1.1 ± 0.84	0.3 ± 0.17
Intel	**0.25 ± 0.22**	0.49 ± 0.33	0.46 ± 0.28	1.5 ± 1.24	0.68 ± 0.58	0.73 ± 0.51
Aces	**0.18 ± 0.14**	0.55 ± 0.40	0.55 ± 0.4	2.6 ± 2.3	0.63 ± 0.51	3.1 ± 2.4

Table 5.4: Global position metric results of the ROSbag circuits

In the first world, most of the SLAM solutions obtained sufficient results, with Hector SLAM having less error in the x-axis position and LaMa's PF in the y-axis. In the case of the worst-performing SLAMs using this metric, we can see that there is either a slight bend in the map generated and/or multiple overlapping walls in all of remaining maps presented in figure 5.13.

Even though our metric reflects these inconsistencies, it could be argued that all of the maps shown here could still be used for navigation.

In the second world, the best performing solution was Gmapping. However, in this case, looking at the quality of the maps generated in figure 5.14, contrary to what is suggested in the metric, both Karto SLAM and LaMa particle filter have managed to produce maps without any obvious shortcomings. Again, this reflects on another drawback of our benchmarking tool, which

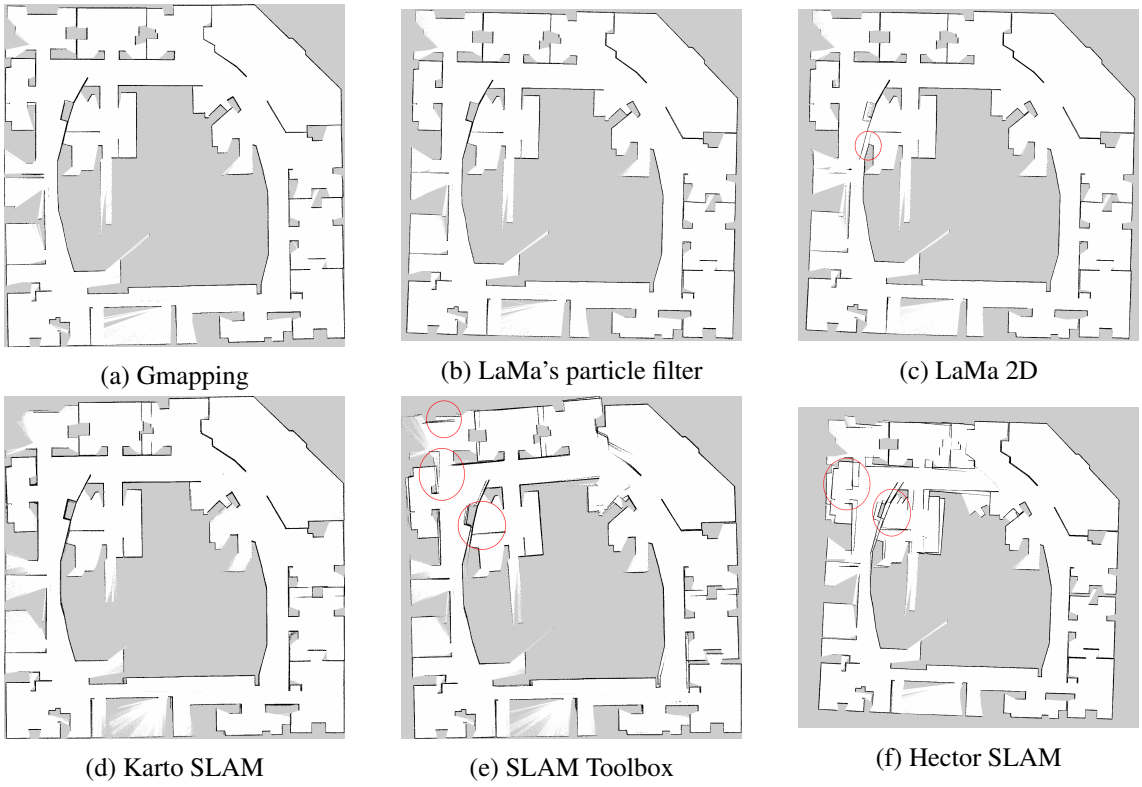


(a) Gmapping (b) LaMa's particle filter (c) LaMa 2D

(d) Karto SLAM (e) SLAM Toolbox (f) Hector SLAM

Figure 5.13: Frei079 maps generated during ROSbag circuit

is evaluating these solutions as they were all Online SLAM algorithms. What most likely happend is that Karto SLAM and LaMa's PF were able to perform loop closing and correctly adjust their final map and path.



(a) Gmapping (b) LaMa's particle filter (c) LaMa 2D

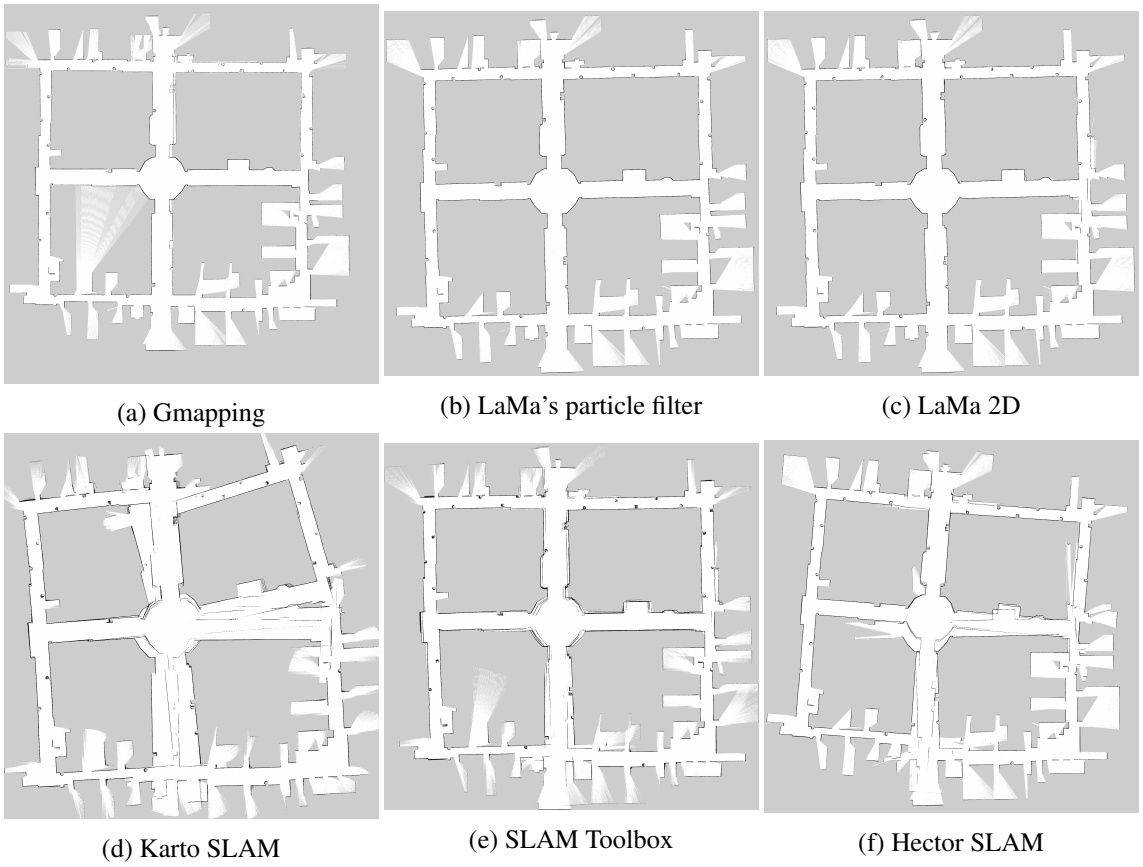
(d) Karto SLAM (e) SLAM Toolbox (f) Hector SLAM

Figure 5.14: Intel maps generated during ROSbag circuit

Finally, in the last and biggest world, the results suggest that the best solution is Gmapping, however, LaMa's PF solution seems to have produced the best map, as seen in figure 5.15. Both Karto SLAM and Hector SLAM, did not come close to producing an accurate map and deviated immensely from the original path when navigating the environment.

From results gathered in the ROSbag circuits, the particle-filter-based SLAM solutions still

remain the more robust solutions. Among the graph-based ones, they have had a slight increase in performance most likely due to the increase in loop closing opportunities. Regarding the Online SLAM solutions, LaMa 2D also remained the better performing one of the two.



(a) Gmapping (b) LaMa's particle filter (c) LaMa 2D

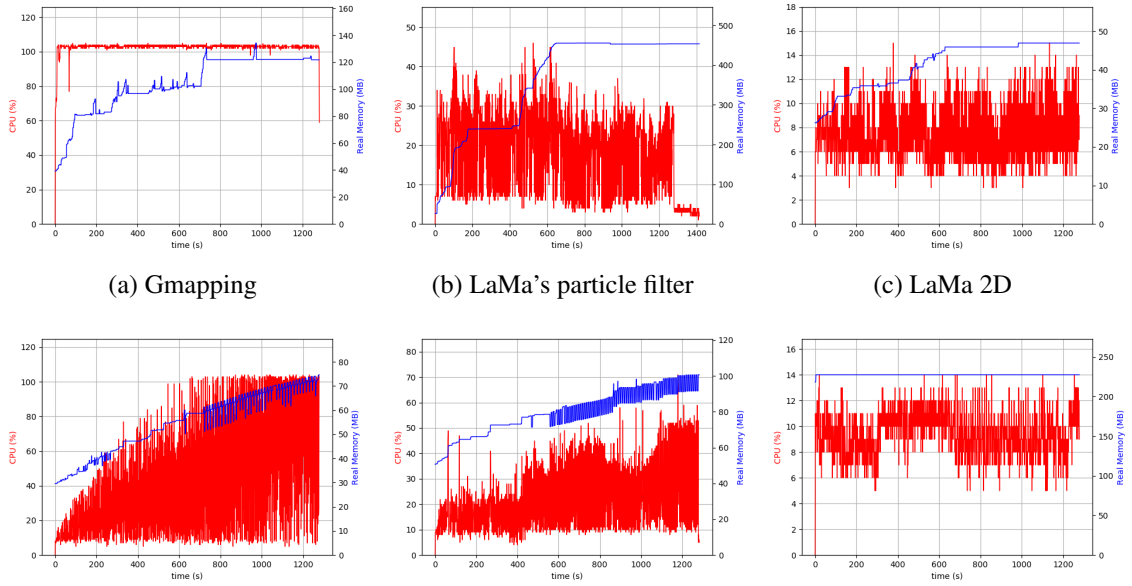
(d) Karto SLAM (e) SLAM Toolbox (f) Hector SLAM

Figure 5.15: ACES maps generated during ROSbag circuit

5.1.3 Computational Resources Consumption

The computational resources spent, in terms of CPU (%) consumption and RAM megabytes consumption, of all of the SLAM solutions for each of the models tested in, are presented in figures 5.16, 5.17 and 5.18, where the x-axis represents the time-frame of the recorded consumption and the y-axis displays both the CPU's percentage consumption (left side) and the RAM usage shown in megabytes (right side). Also, for convenience sake, the results shown here are all gathered during the ROSbag circuits, as the ones gathered in the exploration circuits lead to the same conclusion.

In the case of Frei079's circuit, regarding the CPU (%) usage, it is clear that the Online-SLAM options, LaMa 2D and Hector SLAM, are the least expensive, with similar results, whereas, both Karto SLAM and Gmapping are the most expensive. To distinguish the best solutions, in this case, it is possible to see that LaMa 2D uses much less RAM than Hector, which makes it the least economical solution overall. Analyzing both the graph-mapping solutions we can also see that,



(a) Gmapping (b) LaMa's particle filter (c) LaMa 2D

(d) Karto SLAM (e) SLAM Toolbox (f) Hector SLAM

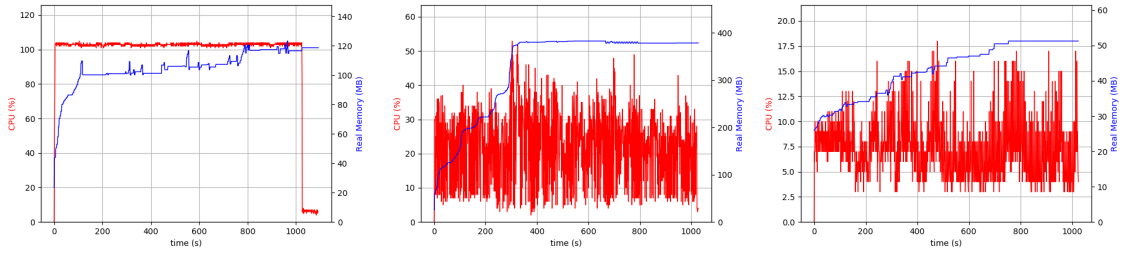
Figure 5.16: Computational resources used by SLAM solutions in Frei079

as time goes on, they increase their performance cost (almost to a linear pattern). This is most likely due to the increase in the number of nodes, as more data is gathered, in the graph, which requires a higher computational power to optimize and also leads to a higher memory requirement as the data saved increases. For the particle filters we can see there is a trade-off between the RAM consumption and CPU usage for both solutions as Gmapping makes complete usage of the CPU power, while LaMa PF trades this for higher usage of the memory capabilities.

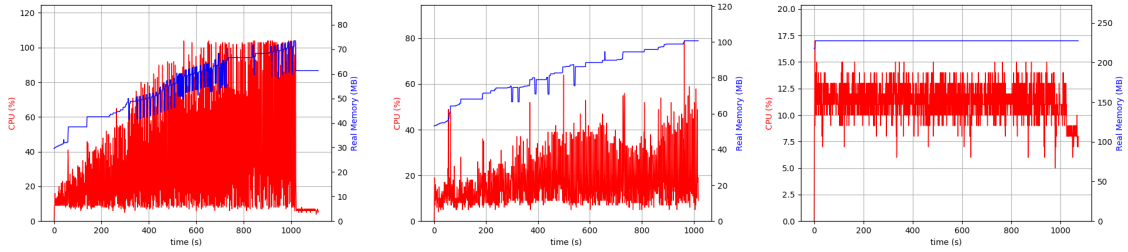
In Intel's circuit, the results gathered display mostly the same conclusions as in the last case, where LaMa 2D continues as the least consuming across all of the solutions.

Lastly, in the ACES circuit, again we see the same trends as in the previous models, except with some increase in the memory consumption as the world is much bigger than the other two. However, LaMa 2D still comes as the best option.

In sum, gathering all of the results from the previous metrics, we can see that both particle filters consistently go "toe-to-toe" regarding the two initial metrics, which makes them both robust solutions for performing SLAM. Picking between these two would end coming down to the computational resources that are available for usage, where LaMa's PF trades more RAM consumption for less CPU (%), while, Gmapping works the other way around. Following that, the graph-based solutions are inconsistent in their performance metrics as the maps generated in the small worlds tend to have discrepancies and, in the case of the exploration circuits, they were unable to map all of the environments, specifically, the ACES' Building. Regarding the least economical solutions, LaMa 2D and Hector SLAM, the former proves to be the most consistent solution of the two.

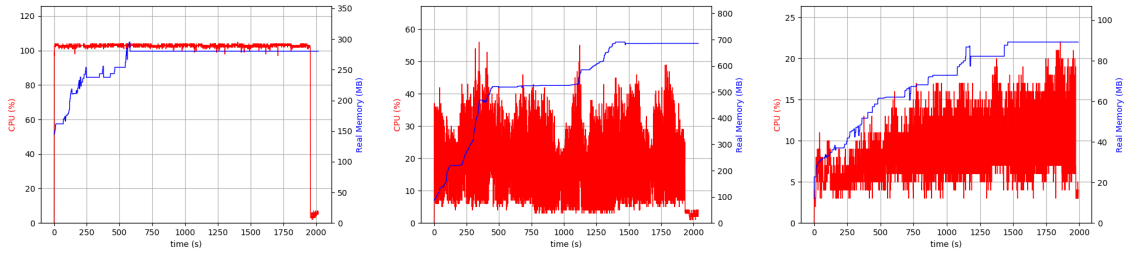


(a) Gmapping (b) LaMa's particle filter (c) LaMa 2D

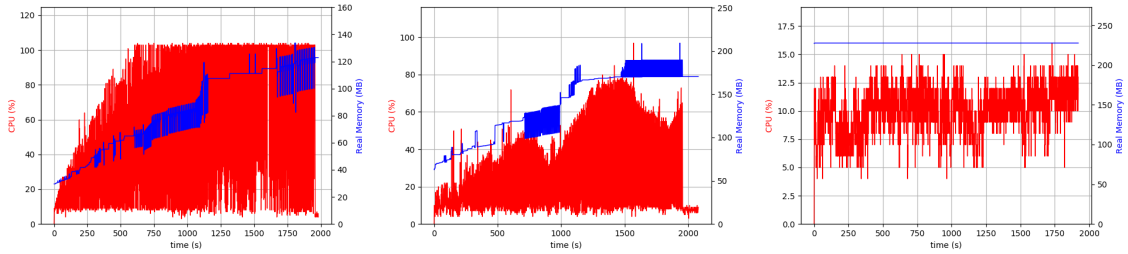


(d) Karto SLAM (e) SLAM Toolbox (f) Hector SLAM

Figure 5.17: Computational resources used by SLAM solutions in Intel



(a) Gmapping (b) LaMa's particle filter (c) LaMa 2D



(d) Karto SLAM (e) SLAM Toolbox (f) Hector SLAM

Figure 5.18: Computational resources used by SLAM solutions in ACES

5.2 Optimal Sensor Placement

In the second and final phase of the project, a number of experiments were conducted to find the optimal sensor positioning/configuration that maximizes the performance of a given SLAM solution picked based on the results obtained in the previous experiments.

One of the most important variables in this phase is the computational power available to conduct the experiments. As mentioned before, the objective function is obtained from the data gathered after the wheelchair has completed some sort of navigation circuit, so each time the function is called, there is a time cost associated with the completion of said circuit. If this cost is too high, the optimization algorithm would simply take too long to obtain a sufficient number of results.

Since these experiments were conducted using Gazebo and ROS, it is possible to speed up the physics engine of the Gazebo simulator to increase the speed of the navigation circuit. However, this increase in speed is reliant on the processing power available in the machine running the simulation. With this said, a number of measures were taken to increase this speed as much as possible, namely:

- The least consuming SLAM solution was picked, specifically, LaMa 2D. This is also advantageous since the metrics developed consider the SLAM solutions to be Online SLAM, which LaMa 2D originally is.
- The world model in 4.10 was used as the testing environment. Since it is such a short environment, the processing power consumed for loading the world should be reduced in comparison with the bigger worlds. Also, even though the SLAM solution picked was not the best from the results gathered in the previous phase, it should still produce good results in such a small environment.
- Instead of using some sort of goal-oriented algorithm coupled with the navigation stack, a follow line algorithm was created to control the navigation of the wheelchair.

Regarding the last point, the *move_back_forth* node was created, contained in the *move_back_forth.py* python script, which stands as a simple follow-line algorithm that creates a predefined circuit by controlling the wheelchair movement by publishing to the *cmd_vel* ROS topic. So, for all the experiments performed in this phase, the wheelchair performs the circuit shown in figure 5.19. With these modifications, it was possible to increase the simulation speed to be 1.5x faster than in real-time, making each objective function call last around 30 seconds.

Apart from this, there also a number of things to consider associated with the objective function results. SLAM solutions are inherently stochastic processes, which means that even when performing the exact same course with the exact sensor configuration, the performance of the solution may not be exactly the same. As an example, we can see the fluctuation between the values in figure 5.20 which compares the metric results of repeated iterations of the navigation circuit with 2 different sensor configurations, one with a FOV of 120º and another with a FOV of 130º. The fluctuation seen in the metric results of both configurations, causes both to alternate between being the best result multiple times. Unfortunately, this further adds to the chance that the hillclimber will get stuck in local minimums, instead of trending to the best sensor positions. This was also taken into account when choosing the SLAM solution to use. Since we are using an Online SLAM solution this should decrease the number of probabilistic predictions in the SLAM

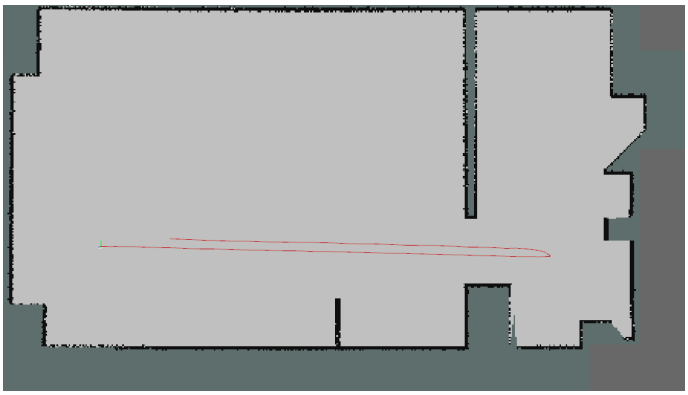


Figure 5.19: Optimization loop circuit (Red Line)

process, compared with graph-based or particle filter-based solutions. Also, in an effort to make the distinction between the performance of each configuration more obvious, the original metric changed from the sum of the mean of the errors in the metrics to the accumulation of errors (equation 4.11).

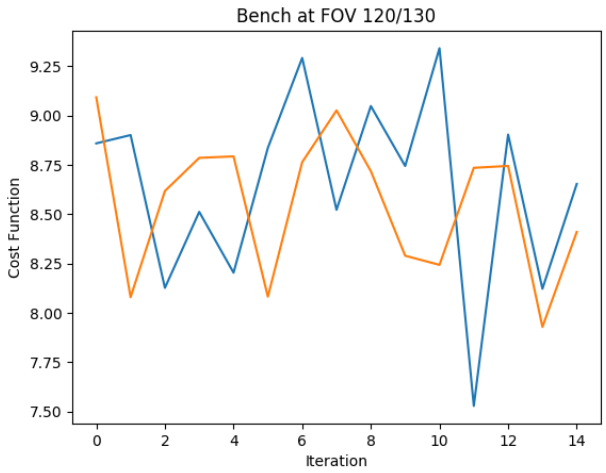


Figure 5.20: Benchmark metric results fluctuation between hillclimber iterations

With these points considered, a number of different experiments were conducted using the optimization algorithm described in section 4.5. The difference between the experiments is the state space that is considered when performing the optimization. The number of variables considered is incrementally increased, starting by only considering the state space as the perimeter around the wheelchair with different fields of view and finishing by considering all of the possible places in the perimeter around the wheelchair with 2 sensors with different configurations (both FOV and the max range). The thought here is to analyze the effect that each variable has on the performance of the SLAM solution. Again, all of these experiments are based on the completion of the circuit in figure 5.19 with odometry model described in section 4.1.1. For future reference, in figures 5.21 and 5.22 we have the path comparisons and the map produced by a configuration

that returned a metric result of 10.1 and 14.1, respectively.

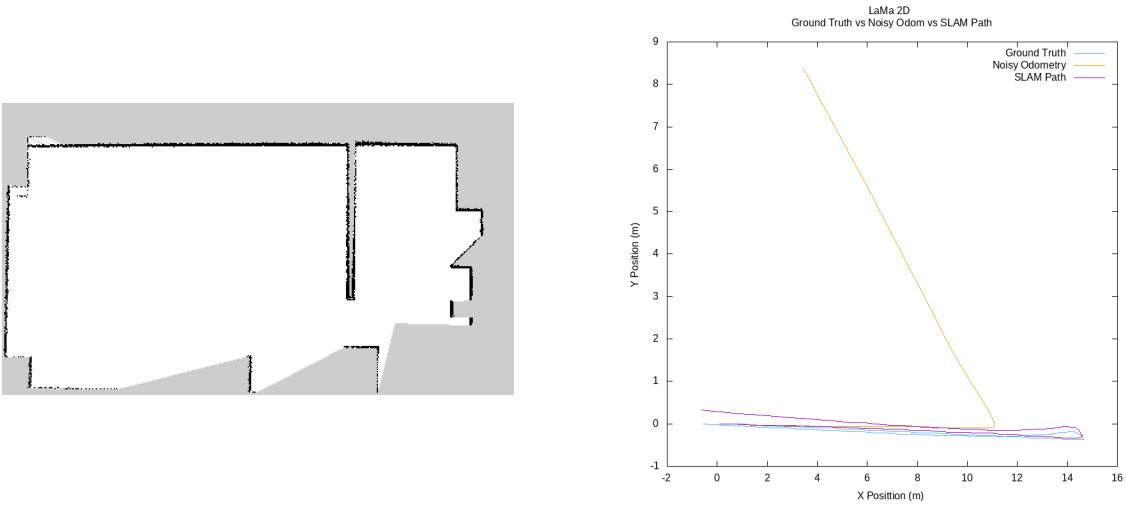


Figure 5.21: Optimization circuit generated map (left figure) & path comparisons (right figure) with benchmarking result of 10.1

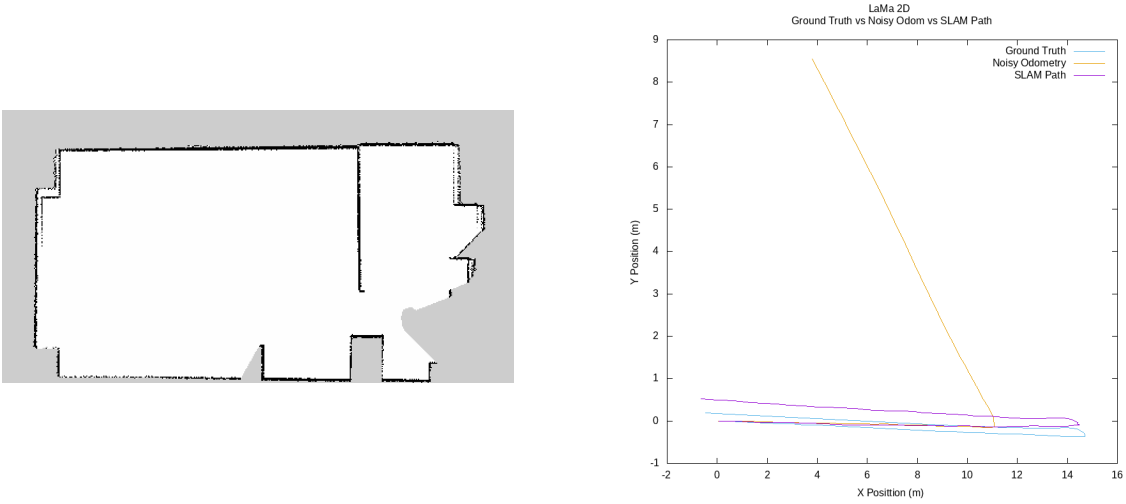


Figure 5.22: Optimization circuit generated map (left figure) & path comparisons (right figure) with benchmarking result of 14.2

Based on these results any sensor configuration that achieves a value below 13 will be considered a good sensor placement performance.

5.2.1 Use Case 1: Single Sensor, Perimeter, FOV & Orientation

In the first experiment conducted, the optimization algorithm was applied to the state space shown in table 4.1 with only 1 sensor and with the range variable restricted to its maximum range of 20 meters. In figure 5.23 we can see all of the local minimums achieved by the different runs of the hill-climber algorithm, ordered from the highest cost function value to the lowest, where

the x-axis represents the hill-climber run and y-axis represents the minimum cost function value found in that run.

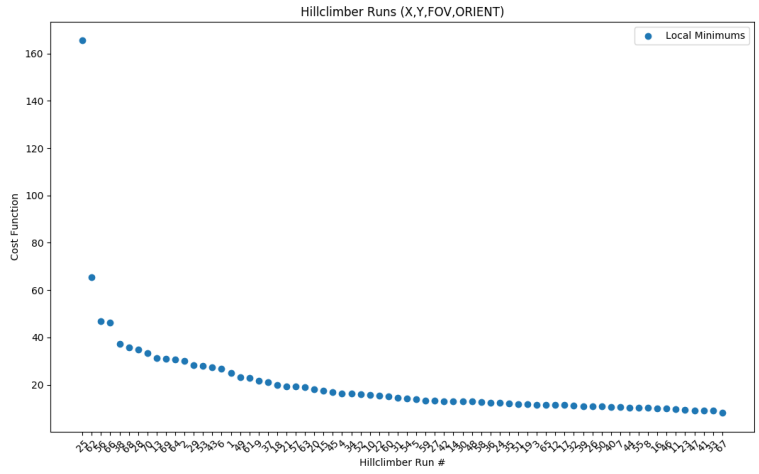


Figure 5.23: All hillclimber results from use case 1

From this plotting it's possible to see that from 70 different hillclimber runs, there are 27 sensor configurations (38.5%) that fall below the 13 meters threshold. All of the sensor configurations below 13 meters are shown in the next figure, where the black box represents the wheelchair model and the triangle the front and center of the model. The blue triangles are the sensors, each with an orientation associated with them (where the triangle is pointing). The FOV of the sensor is the number next to the sensor position/configuration.

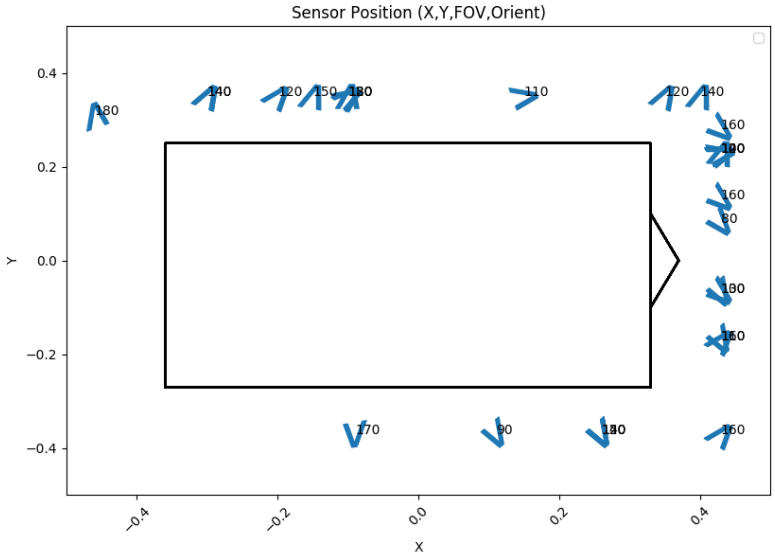


Figure 5.24: Sensor configurations with a benchmarking metric result below 13 in use case 1

There is a concentration of a large part of the sensors in the front of the wheelchair with an orientation and FOV sufficient to always cover the direction of the movement of the wheelchair. Even the sensors that ended up on the sides of the wheelchair seem to either be pointing in the direction of the wheelchair movement or have a FOV sufficient to cover it. In the small world model considered, the sensors being pointed in the direction of movement of the wheelchair with a range of 20 meters, means that at all times, the SLAM is able to get data from a wall that is perpendicular to its direction of movement. This should make the scan matching process easier and, in turn, its SLAM estimate more accurate. Furthermore, pretty much all of the sensors configurations have a FOV above 100º. In figures 5.25 we see the top 4 positions of these hillclimber runs. The legend of each plot represents the configuration plus the result (FOV,Range=Cost Function Result). All of them are pointed in the direction of wheelchair movement and have FOVs higher than or equal to 130º.

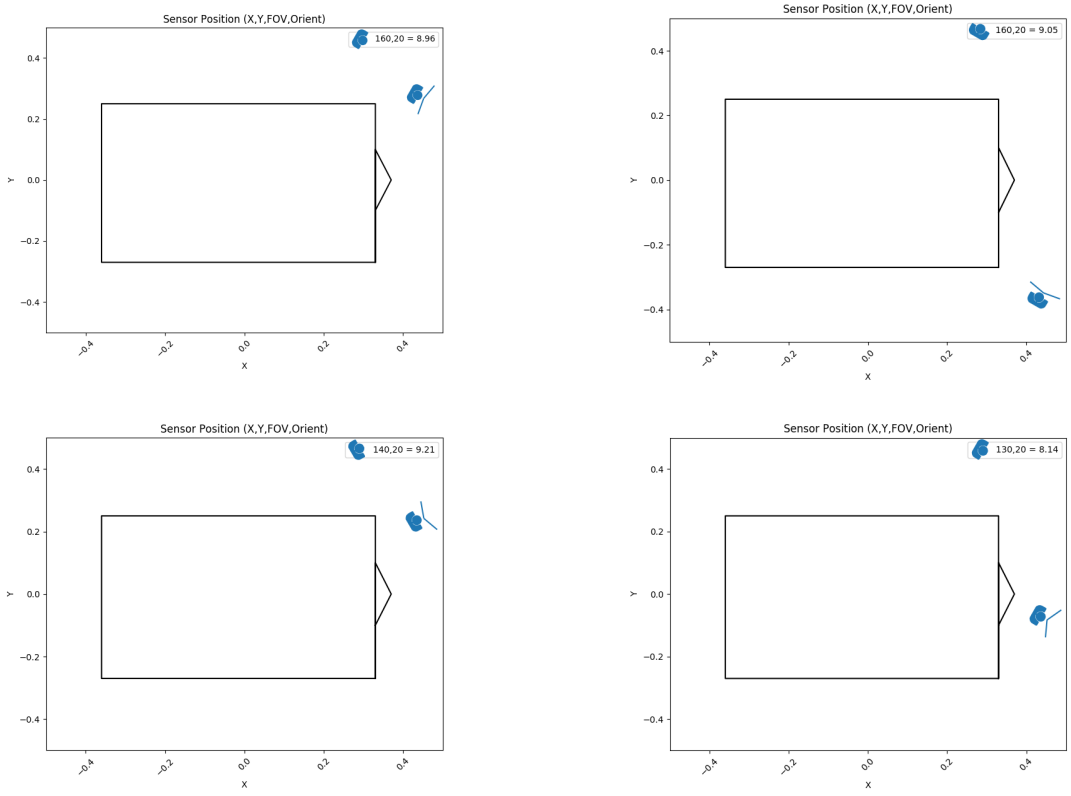


Figure 5.25: Top 4 sensor positions/configurations in use case 1

5.2.2 Use Case 2: Dual Sensors, Perimeter, FOV & Orientation

In the second use case scenario, the same variables as the previous scenario were used, however, a second sensor was added with the same state space possibilities. All of the results from the different hillclimber iterations are present in figure 5.26.

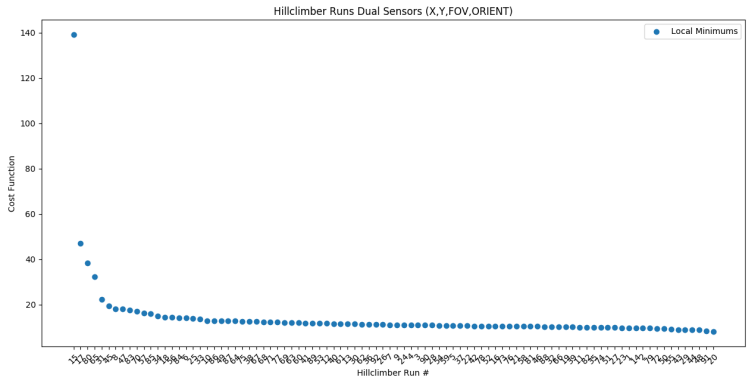


Figure 5.26: All hillclimber results from use case 2

From 95 different hillclimber runs, there are 73 sensor configurations (77%) that have achieved a cost function below 13 meters. Since it would become quite cumbersome to display all 73 positions, in figure 5.27 we have all of the sensor configurations that achieved values below a cost function of 10 meters. Both plots in the figure show the pair of sensors of these configurations where each colored sensor has a matching pair on the second plot.

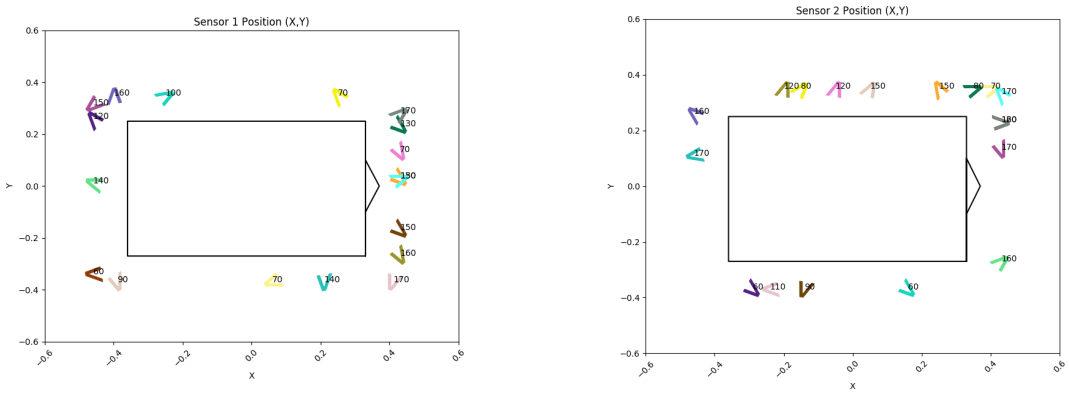


Figure 5.27: Sensor configurations with a benchmarking metric result below 10 in use case 2

Looking at both plots, it is clear that there is a trend of each pair of sensors diverting their orientation in order to maximize the area covered by the laser scans. Also, some sensors have started appearing pointing in the opposite direction of movement of the wheelchair (the back of the chair), which is further shown in figure 5.28 where the best sensor positions/configurations are displayed.

From the top results, we can also see an interesting configuration in the last plot that diverts from the trend of previous results pointing in the direction of the movement of the wheelchair while still managing to be one of the top results. Another interesting remark is that the cost function values of both the single and dual sensor are all in the same range, which suggests that there are single sensor positions/configurations that offer similar results to the best dual sensor posi-

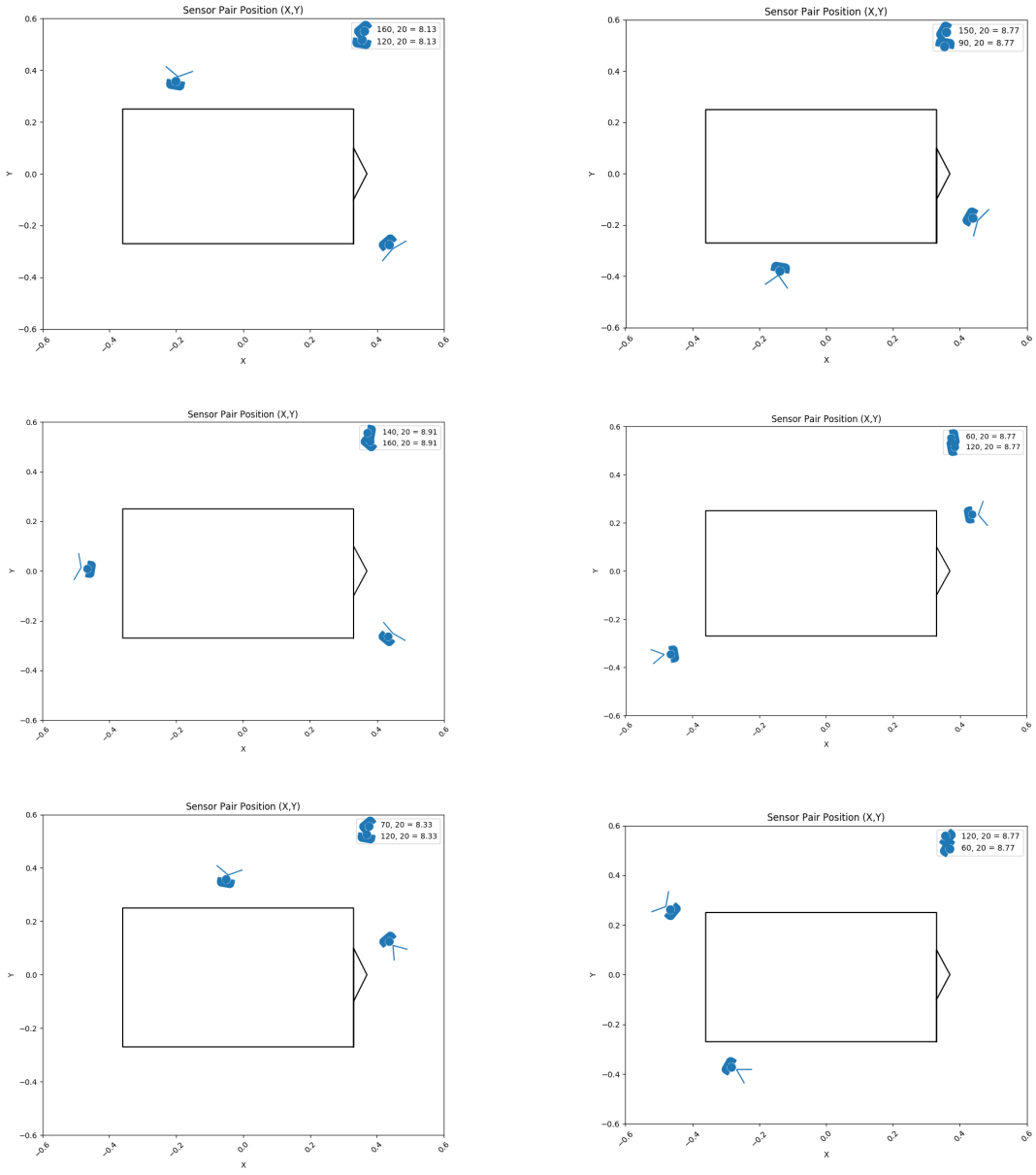


Figure 5.28: Top 6 sensor configurations in use case 2

tion/configurations. However with a similar number of hillclimber runs for both single and dual sensor positions/configurations, there are many more results below the set threshold of 13 meters in the case of the 2 sensors than with just 1 sensor, which implies that is easier to achieve acceptable SLAM performance using double sensors configurations then when using a single sensor.

5.2.3 Use Case 3: Single Sensor, Perimeter, Limited FOV & Orientation

In this next use case, the FOV was limited to a max range of 120º instead of 180º. The point is to check if it is possible to achieve the same SLAM performance up until now, with a lower

quality of sensors (lower FOV). From the 110 different hillclimber runs in figure 5.29, there were 24 sensor configurations (22%) falling below the set 13 meters threshold.

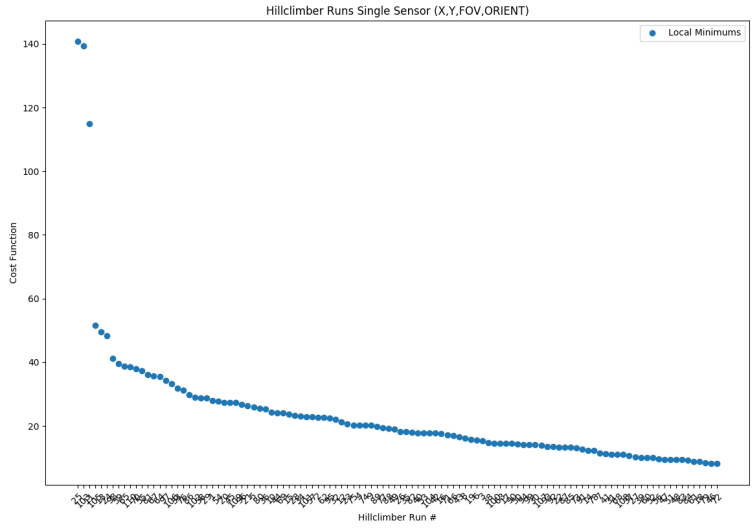


Figure 5.29: All hillclimber results from use case 3

In figure 5.30 the same trend in the orientation and position of the sensors continues. The top results in figure 5.31 also show that, in this experimental setting, that it is possible to achieve the same SLAM performance with sensors with a lower FOV coverage.

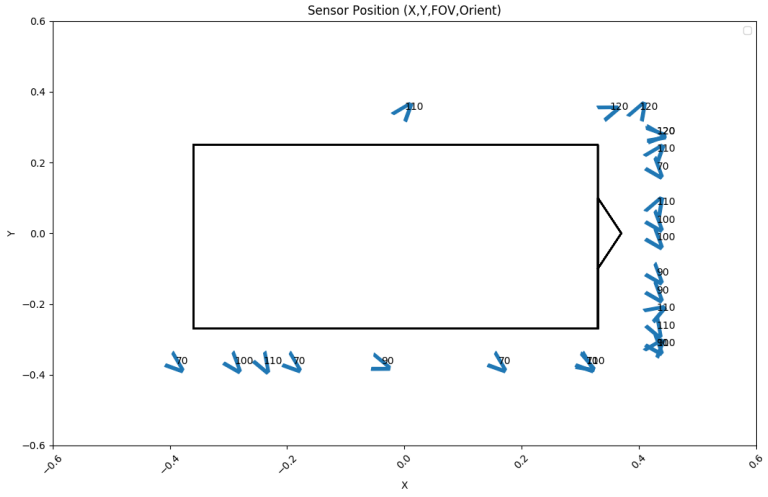


Figure 5.30: Sensor configurations with a benchmarking metric result below 13 in use case 3

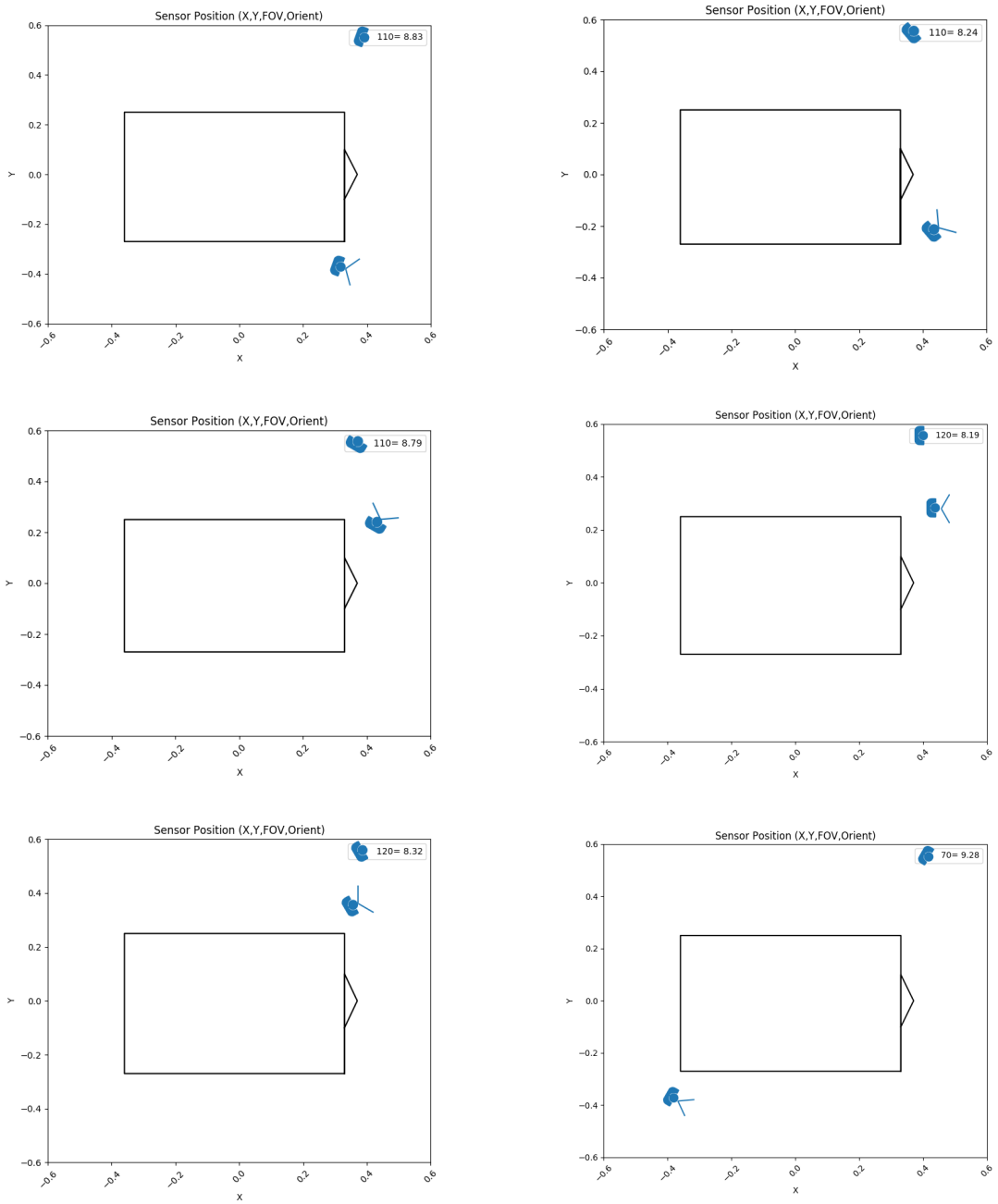


Figure 5.31: Top 6 sensor configurations in use case 3

5.2.4 Use Case 4: Double Sensors, Perimeter, Limited FOV & Orientation

In this use case, again, a second sensor is added to the state space of the previous use case. From 95 different hillclimber tests, all of the final results are displayed in figure 5.32 having 46 sensor configurations (48.2%) achieved results below the 13-meter metric threshold.

Comparing the best results in this use case with the previous, the sensors continue to divert their orientations to cover a bigger area, and the performance compared with the performance in

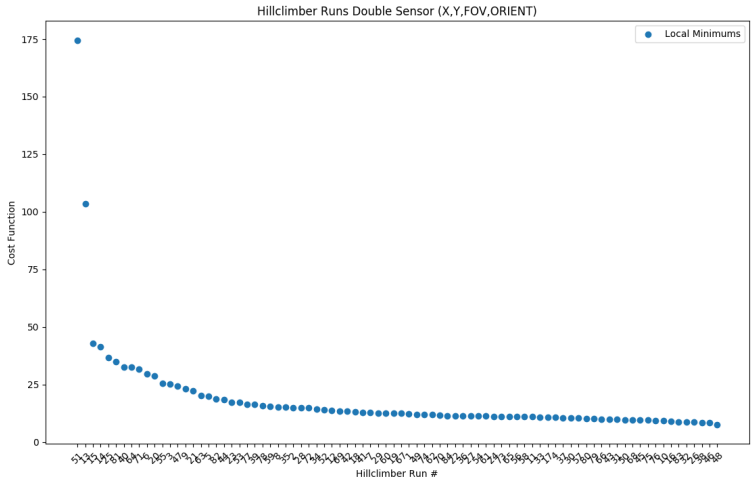


Figure 5.32: All hillclimber results from use case 4

the use cases of a single sensor does not seem to be any better.

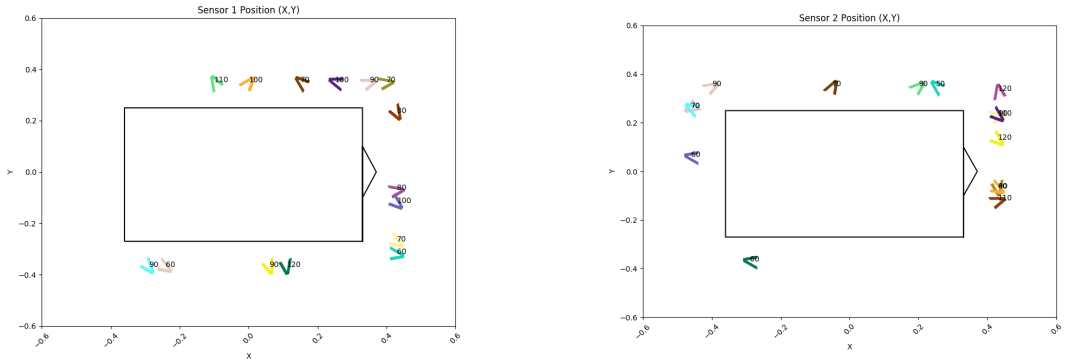


Figure 5.33: Sensor configurations with a benchmarking metric result below 10 in use case 4

5.2.5 Use Case 5: Single Sensor, Perimeter, FOV, Orientation & Range

In the final experiments, the variable of the max range is included in the state space with an interval of [4,20]. The size of the testing world is 20x8, so the max possible range was restricted to 20 meters, which represents a range that, at all places in the world, should be able to cover the whole environment. In figure 5.35, the final results of all the 78 different hillclimber runs are presented, having landed 26 sensor positions/configurations (33.3%) below the 13 meters threshold.

We can see the sensor configurations falling below the set threshold, in figure 5.36 (the range of the sensor is now next to the FOV). The trend of the sensors pointing in the direction of the wheelchair movement continues. As far as the max range of the laser scans, most sensors have at least a 12-meter max range, which makes it so, if the sensor is pointing in the direction of

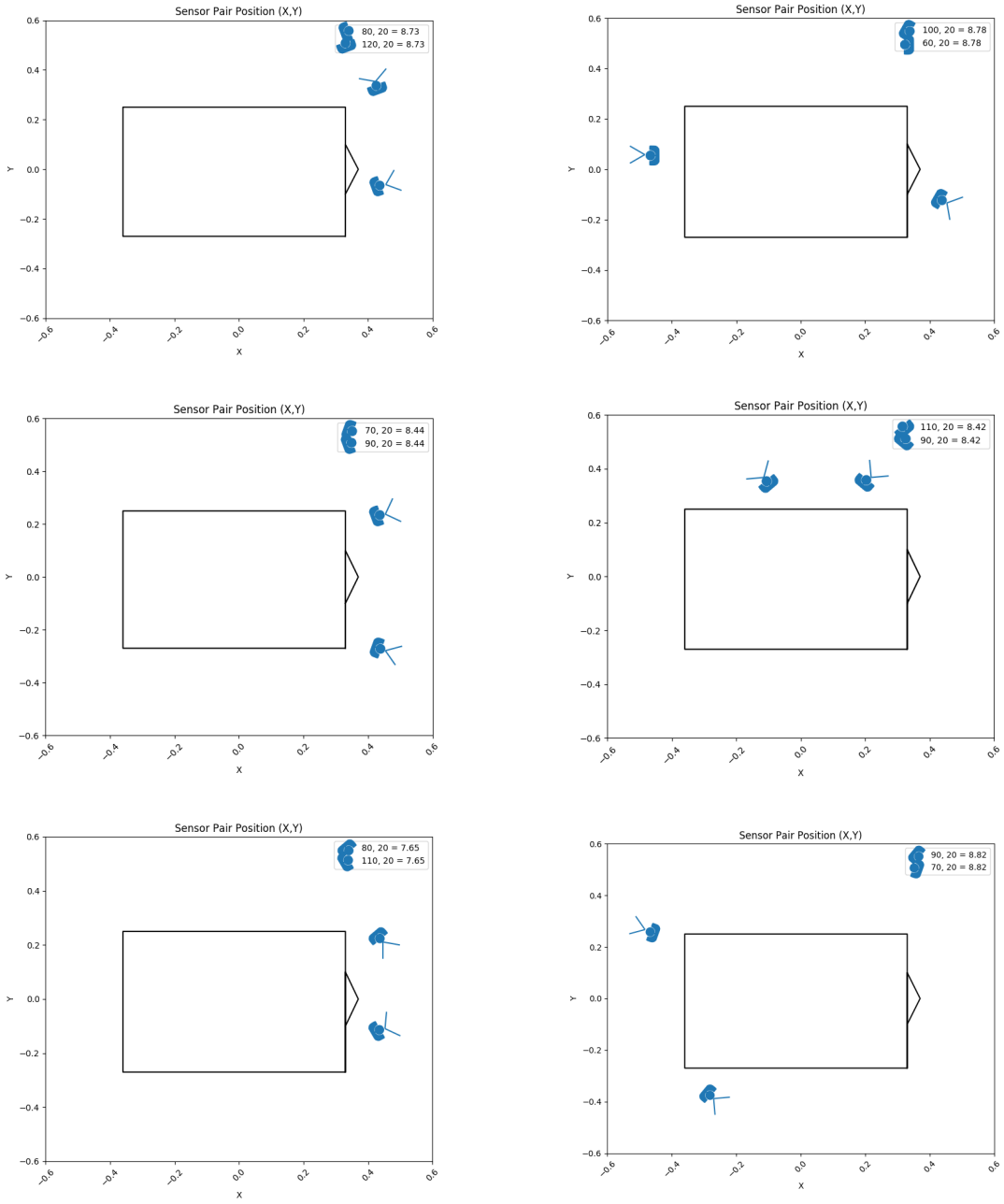


Figure 5.34: Top 6 sensor configurations in use case 4

the wheelchair movement, the SLAM solution has data on an obstacle opposing the direction of movement on most of the circuit duration.

From the best sensor configurations in plotted in figure 5.37 all the same trends are seen. We can also see that 2 of the best sensor positions/configurations manage to be in the top with only a 12-meter max range laser scan readings.

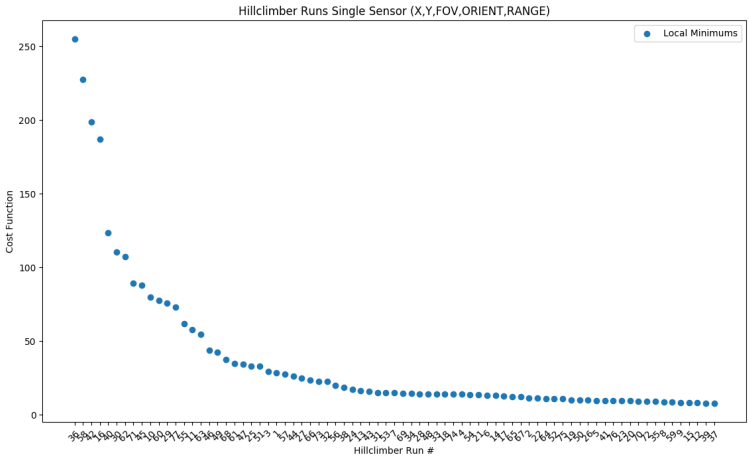


Figure 5.35: All hillclimber results from use case 5

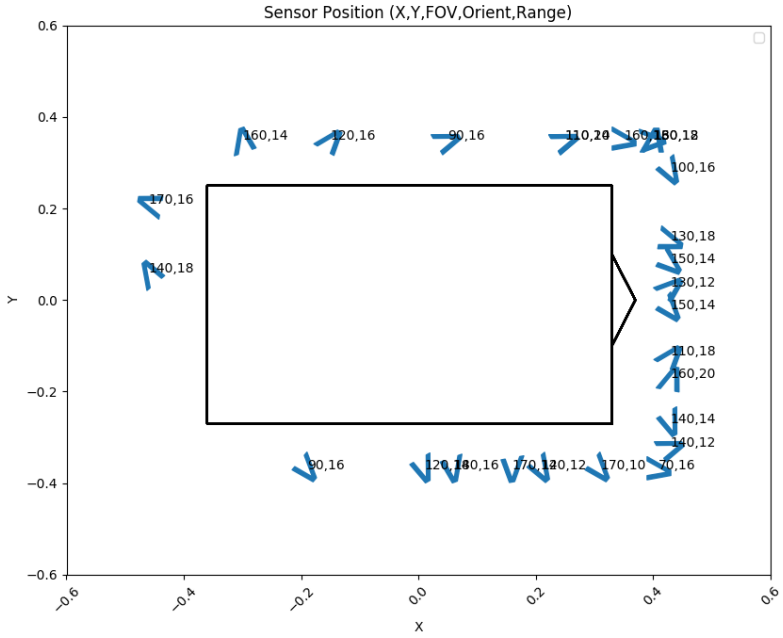


Figure 5.36: Sensor configurations with a benchmarking metric result below 13 in use case 5

5.2.6 Use Case 6: Dual Sensors, Perimeter, FOV, Orientation & Range

Again, the same variables as the previous scenario were used, however, a second sensor was added with the same state space possibilities. All of the results from the different hillclimber iterations are present in figure 5.38.

From the 81 ordered results, there are 52 results that are below the 13 meters threshold. In figure 5.39 we can see all of the results below the threshold of 10 meters. The range of the sensors isn't displayed in this plot as the figure would be quite hard to read. Much of the same seen in use

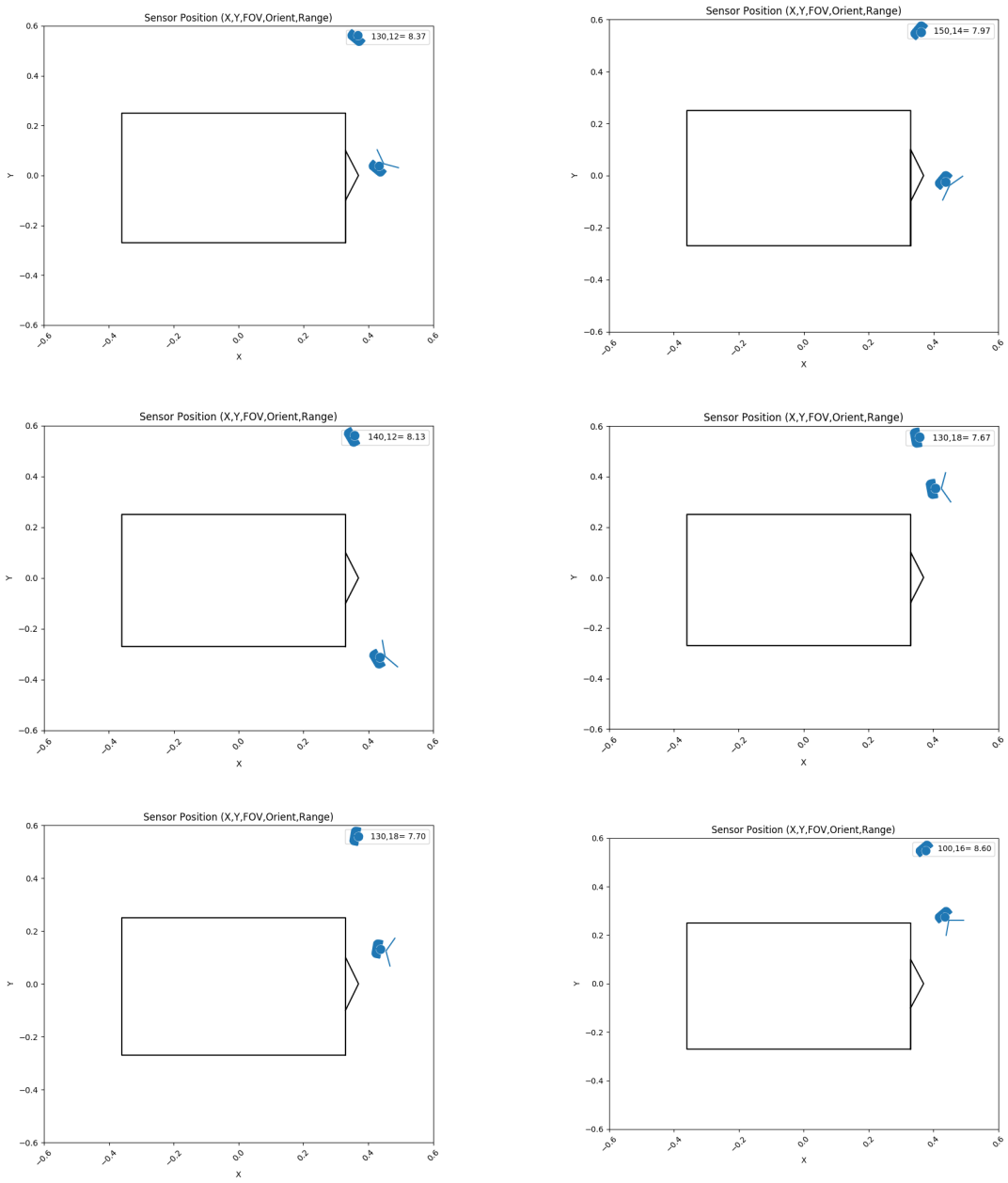


Figure 5.37: Top 6 sensor configurations in use case 5

case 2 can be seen here with the sensors having opposing orientations in an attempt to maximize the area covered.

From the best positions in figure 5.40, most of the sensor pairs shown seem to have a difference in the range of the sensor. This, coupled with the fact that there does not seem to be any difference between the final results in the use cases where one or two sensors are used suggests that there is no significant difference in SLAM performance between using either one or two sensors if one of them is able to sufficiently cover the environment (enough FOV and range).

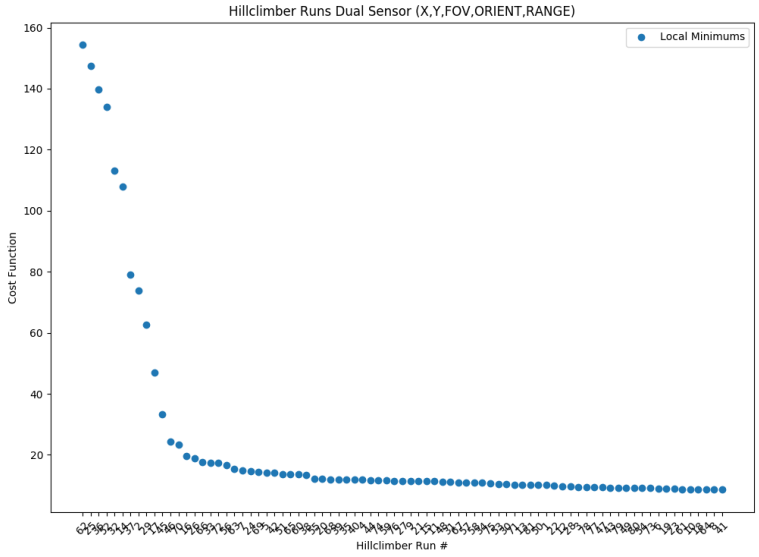


Figure 5.38: All hillclimber results from use case 6

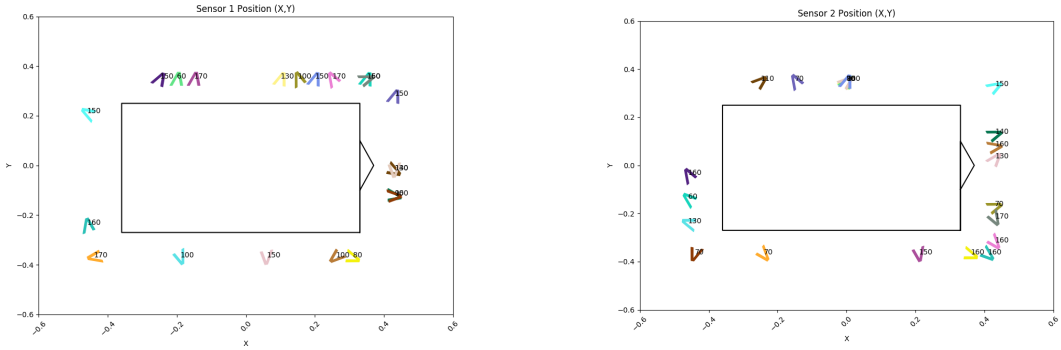


Figure 5.39: Sensor configurations with a benchmarking metric result below 10 in use case 6

From all the results gathered in the phase of the project, it is possible to see that, for the navigation scenario proposed, the best possible single sensor position/configuration is able to achieve similar performance to when using double sensors. Also, it is possible to reduce the quality of the sensor, specifically, the FOV covered and its max range, and still be able to achieve a SLAM performance that could rival the best sensor position/configuration that was previously perceived to be the best.

These two findings point to both a possible cost reduction in the sensor equipment of intelligent wheelchairs, as the possibility of the usage of lower-performing sensors has opened. Also, the different possible sensor positions offer interesting alternatives in the physical placement of the sensor. This is an important point to consider, as the space available for the placement of sensors in powered wheelchairs can be limited by the actual structure of the wheelchair and the user of the wheelchair.

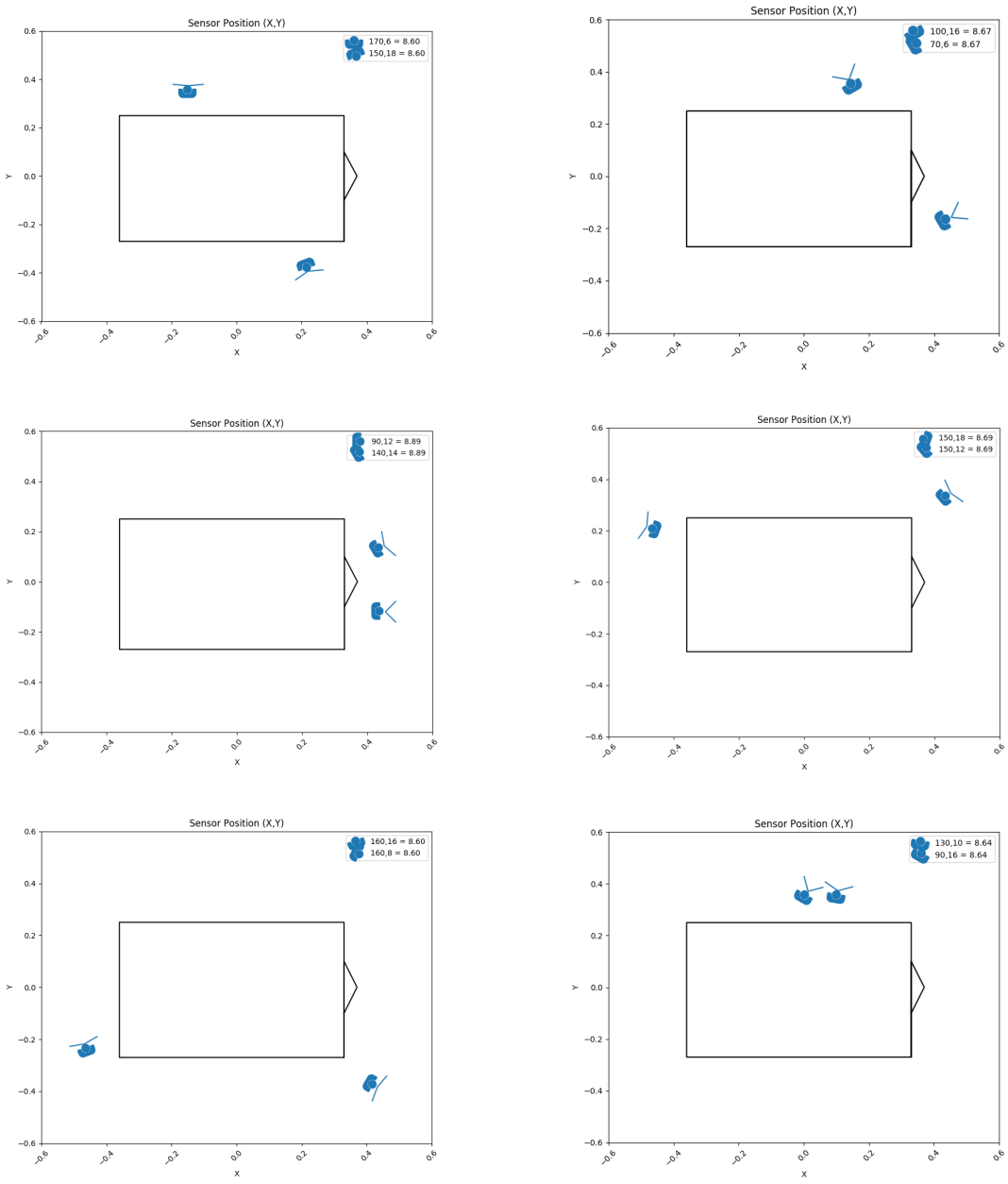


Figure 5.40: Top 6 sensor configurations in use case 6

5.3 Summary

From the results gathered in this first phase of the experiments it was found that the particle-filter-based SLAM solutions showed better performance on both the exploration circuits and the ROSbag circuits. However, if the scenario is appropriate, namely if the deployment environment is small enough, and the computational resources are limited, the Online SLAM solution LaMa 2D is the best option. Regarding the graph-based solutions, there does not seem to be any advantage

in using them since they end up consuming similar computational resources as the particle-filter-based ones but are much more inconsistent regarding their performance.

In the phase of the next experiments, the results gathered offer a number of different conclusions. There are multiple sensor positions/configurations that were able to achieve similar optimal SLAM performance. These results are important since the available sensor footprint, in a real-life situation, could possibly be much more restrictive than state space considered. Furthermore, if the placement and orientation are correct it is possible to reduce configurations of the sensor (FOV and range) to values that may have not been so obvious. This is also important, as a reduction in the capabilities of the sensor opens a possibility in the reduction of the overall cost of the equipment required to implement an intelligent wheelchair.

Chapter 6

Conclusion

Throughout this document, the SLAM problem, along with some of its common solutions and existing problems, was explained and reviewed. Even though this topic has had much research over the years, the solutions that have been developed are heavily dependent on the combination of environment, robot, and hardware it is applied to. Moreover, these solutions are also dependent on the perception of the environment offered by the sensors, so it's important that the sensors are placed in the robot in a manner that maximizes the performance of the SLAM solution that is used. This reflects how broad and complex SLAM is, as there are situations where this problem has been solved and others where more research is still required. Similar to what has been done in the automobile industry, where the proper placement of LiDAR scanners was considered in order to optimize the performance of self-driving mechanisms, the possibility of looking at sensor placement as an optimization problem applied to SLAM performance was explored.

Intelligent wheelchairs are mobility aid instruments that make use of common mobile robotics software and hardware to augment the capabilities of power wheelchairs. These aim to provide alternative control methods and either fully autonomous or semi-autonomous mobility by installing adaptable kits, composed of a number of sensors connected to a computer, in existing power wheelchair models. As expressed in section 3.2.2, there have been several implementations of SLAM algorithms in IW prototypes, making use of open-source ROS SLAM packages like RGB-D SLAM, GMapping, ORB-SLAM, etc, that have achieved satisfactory results.

The research conducted during this dissertation was divided into 2 separate phases. The first phase was dedicated to comparing a number of 2D SLAM algorithms, in the context of the Intellwheels 2.0 project, by using a comparative process, where, the selected SLAM implementations were deployed in 2 different types of navigation scenarios, each applied to 3 different worlds. Additionally, the computational resources consumption of each of the solutions was also evaluated. From the results gathered, it seems the particle-filter-based solutions still remain among the most robust for all of the navigation scenarios tested, with LaMa offering considerable savings in its processing power. On the other hand, the remaining solutions still offered sufficient performance for scenarios where the loop-closing opportunities were more abundant. Furthermore, in the smaller environments and the ROSbag circuits, the Online SLAM solutions offered the best

performance-computational resources consumption ratio.

For the next experiment phase, where an adapted steepest hillclimber optimization algorithm was used to find the best sensor placement/configuration that maximizes SLAM performance, the number of computational resources that could be diverted to speeding the simulation physics was important. So, since the testing scenario in this phase was quite simple, the more economical of the SLAM solutions, LaMa 2D, was used. Based on all the different variables considered it was found that, under the simulation conditions offered, a number of different existing positions/configurations manage to maintain similar optimal results. This abundance in different positions may be interesting as further restrictions are added to the possible sensor footprint. Also, by choosing certain sensor positions/orientations it was found that there does not seem to be any advantage in using sensors over the 110º FOV range, either by adding extra sensors or by using more complex sensors.

6.1 Contributions

Overall, in this dissertation, a comparison process for both 2D SLAM algorithms and different sensor placements/configurations was developed, in the context of the Intellwheels 2.0 project, with its main contributions being:

- A better understanding of the SLAM problem, its main available solutions, and its possible further developments.
- The implementation of an intelligent wheelchair system capable of performing SLAM using various ROS-based 2D SLAM solutions.
- The creation of simulation environments that are able to explore the performance of said 2D SLAM solutions under different navigation scenarios.
- The comparison of different SLAM solutions under a benchmarking tool based on their relative motion and global localization error.
- The application of an optimization algorithm aimed at maximizing the performance of the chosen SLAM algorithm, LaMa 2D, by finding the optimal sensor placement/configuration of one or multiple laser scan rangers.

6.2 Future Work

While performing experiments under simulated environments is convenient, it should be noted that, even with the effort put into adding noise into both exteroceptive and interoceptive sensors and with the different indoor models created, these testing scenarios are still far from completely emulating real-life conditions. In turn, the performance of the SLAM algorithms may be inaccurately portrayed which also reflects on the results given by the optimization algorithm, especially

in this case, since the environment used was quite simple and small. Additionally, some shortcomings were also identified in both the benchmarking metric used and the optimization algorithm architecture so it would be of note to attempt this optimization using different benchmarking metrics and more complex optimization algorithms architectures, such as genetic algorithms or simulated annealing. The full scope of possible sensor positions/configurations was also not used, due to time constraints. In sum, it is possible to identify a number of improvements and different testing scenarios that could be applied to the existing framework, specifically:

- Consider the full scope of possible sensor positions/configurations (laser resolution, z-axis, etc).
- Add further restrictions to the state space. For example, in a real-life scenario, the front of the wheelchair would be much more restricted by the legs of the wheelchair user.
- Attempt the same optimization algorithm using a different cost function and different environments.
- Adapt more complex optimization algorithms such as simulated annealing, genetic algorithms, tabu search algorithms, etc.
- Even though it would be far too cumbersome to use an optimization algorithm in a real-life scenario, it would still be interesting to check if the best positions/configurations obtained in the simulated environment, reflect the same results in a real-life scenario.

References

[1] Roland Siegwart, Illah Reza Nourbakhsh, and Davide Scaramuzza. *Introduction to autonomous mobile robots*. MIT press, 2011.

[2] Gian Diego Tipaldi and Wolfram Burgard. Robot Mapping Introduction to Robot Mapping What is Robot Mapping ?

[3] Hugh Durrant-Whyte and Tim Bailey. Simultaneous localization and mapping: Part I. *IEEE Robotics and Automation Magazine*, 13(2):99–108, 2006.

[4] Joydeep Dey, Wesley Taylor, and Sudeep Pasricha. Vespa: A framework for optimizing heterogeneous sensor placement and orientation for autonomous vehicles. *IEEE Consumer Electronics Magazine*, 2020.

[5] Maya Burhanpurkar, Mathieu Labbé, Charlie Guan, François Michaud, and Jonathan Kelly. Cheap or robust? the practical realization of self-driving wheelchair technology. In *2017 International Conference on Rehabilitation Robotics (ICORR)*, pages 1079–1086. IEEE, 2017.

[6] A. Juneja, L. Bhandari, H. Mohammadbagherpoor, A. Singh, and E. Grant. A comparative study of slam algorithms for indoor navigation of autonomous wheelchairs. In *2019 IEEE International Conference on Cyborg and Bionic Systems (CBS)*, pages 261–266, 2019. doi:10.1109/CBS46900.2019.9114512.

[7] Rodrigo Antonio Marques Braga, Marcelo Petry BEng, Luís Paulo Reis, and Antonio Paulo Moreira. Intellwheels: Modular development platform for intelligent wheelchairs. *Journal of Rehabilitation Research & Development*, 48(9), 2011.

[8] Wolfram Burgard, Cyrill Stachniss, Giorgio Grisetti, Bastian Steder, Rainer Kümmerle, Christian Dornhege, Michael Ruhnke, Alexander Kleiner, and Juan D. Tardós. A comparison of SLAM algorithms based on a graph of relations. *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2009*, pages 2089–2095, 2009. doi:10.1109/IROS.2009.5354691.

[9] Disability Inclusion and Accountability Framework. page 80, 2018. URL: http://documents.worldbank.org/curated/en/437451528442789278/Disability-inclusion-and-accountability-framework.

[10] Norman a Jacobs and Sarah Sheldon. Report of a Consensus Conference on Wheelchairs for Developing Countries Bengaluru , India Edited By. (November):1–318, 2006. URL: http://www.who.int/disabilities/technology/WCGconcensusconf/en/.

[11] Abhishek Pandey, Anirudh Kaushik, Amit Kumar Jha, and Girish Kapse. A Technological Survey on Autonomous Home Cleaning Robots. *International Journal of Scientific and Research Publications*, 4(4):1–7, 2014. URL: www.ijsrp.org.

[12] T B Asafa, T M Afonja, E A Olaniyan, and H O Alade. Development of a vacuum cleaner robot. *Alexandria Engineering Journal*, 57(4):2911–2920, 2018. URL: https://doi.org/10.1016/j.aej.2018.07.005, doi:10.1016/j.aej.2018.07.005.

[13] De Engenharia Eletrot. Evolution of Odometry Calibration Methods for Ground Mobile Robots. pages 294–299. doi:10.1109/icarsc49921.2020.9096154.

[14] Filipe André Sousa Barbosa and Doutor Adriano Carvalho. FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO Controlo de Tração em Veículos Elétricos MESTRADO INTEGRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES MAJOR: AUTOMAÇÃO. 2013. URL: https://paginas.fe.up.pt/{~}ee08326/wp-content/uploads/2013/03/PDI-Relatorio-FInal{_}V3-Final.pdf.

[15] Fei Dai, Youyi Feng, and Ryan Hough. Photogrammetric error sources and impacts on modeling and surveying in construction engineering applications. *Visualization in Engineering*, 2(1):1–14, 2014. doi:10.1186/2213-7459-2-2.

[16] Sebastian Thrun, Wolfram Burgard, Dieter Fox, et al. Probabilistic robotics, vol. 1, 2005.

[17] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, Jose Neira, Ian Reid, and John J. Leonard. Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age. *IEEE Transactions on Robotics*, 32(6):1309–1332, 2016. arXiv:1606.05830, doi:10.1109/TRO.2016.2624754.

[18] J. E. Guivant and E. M. Nebot. Optimization of the simultaneous localization and map-building algorithm for real-time implementation. *IEEE Transactions on Robotics and Automation*, 17(3):242–257, 2001. doi:10.1109/70.938382.

[19] Guido Zunino and Henrik I Christensen. Navigation in realistic environments. In *9th Intl. Symp. on Intelligent Robotic Systems, Toulouse, France*, 2001.

[20] Giorgio Grisettiyz, Cyrill Stachniss, and Wolfram Burgard. Improving grid-based slam with rao-blackwellized particle filters by adaptive proposals and selective resampling. In *Proceedings of the 2005 IEEE international conference on robotics and automation*, pages 2432–2437. IEEE, 2005.

[21] Arnaud Doucet, Nando De Freitas, Kevin Murphy, and Stuart Russell. Rao-blackwellised particle filtering for dynamic bayesian networks. *arXiv preprint arXiv:1301.3853*, 2013.

[22] Arnaud Doucet. On sequential simulation-based methods for bayesian filtering. 1998. doi:doi=10.1.1.361.4361.

[23] . *Scan Matching and SLAM for Mobile Robot in Indoor Environment*. PhD thesis, , 2016.

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

[25] Feng Lu and Evangelos Milios. Globally consistent range scan alignment for environment mapping. *Autonomous robots*, 4(4):333–349, 1997.

[26] Sebastian Thrun and Michael Montemerlo. The graph slam algorithm with applications to large-scale mapping of urban structures. *The International Journal of Robotics Research*, 25(5-6):403–429, 2006.

[27] Michael Kaess, Ananth Ranganathan, and Frank Dellaert. isam: Incremental smoothing and mapping. *IEEE Transactions on Robotics*, 24(6):1365–1378, 2008.

[28] Davide Scaramuzza and Friedrich Fraundorfer. Visual odometry [tutorial]. *IEEE robotics & automation magazine*, 18(4):80–92, 2011.

[29] Josef Sivic and Andrew Zisserman. Video google: A text retrieval approach to object matching in videos. In *null*, page 1470. IEEE, 2003.

[30] Winston Churchill and Paul Newman. Experience-based navigation for long-term localisation. *The International Journal of Robotics Research*, 32(14):1645–1661, 2013.

[31] Michael J Milford and Gordon F Wyeth. Seqslam: Visual route-based navigation for sunny summer days and stormy winter nights. In *2012 IEEE International Conference on Robotics and Automation*, pages 1643–1649. IEEE, 2012.

[32] Yasir Latif, César Cadena, and José Neira. Robust loop closing over time for pose graph slam. *The International Journal of Robotics Research*, 32(14):1611–1626, 2013.

[33] Luca Carlone, Andrea Censi, and Frank Dellaert. Selecting good measurements via 1 relaxation: A convex approach for robust estimation over graphs. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2667–2674. IEEE, 2014.

[34] Giorgio Grisetti, Rainer Kümmerle, Cyrill Stachniss, Udo Frese, and Christoph Hertzberg. Hierarchical optimization on manifolds for online 2d and 3d mapping. In *2010 IEEE International Conference on Robotics and Automation*, pages 273–278. IEEE, 2010.

[35] Javier Civera, Dorian Gálvez-López, Luis Riazuelo, Juan D Tardós, and Jose Maria Martinez Montiel. Towards semantic slam using a monocular camera. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1277–1284. IEEE, 2011.

[36] Michael Kaess. Simultaneous localization and mapping with infinite planes. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4605–4611. IEEE, 2015.

[37] Kevin Lai, Liefeng Bo, and Dieter Fox. Unsupervised feature learning for 3d scene labeling. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3050–3057. IEEE, 2014.

[38] Tae-Hyeong Kim and Tae-Hyoung Park. Placement optimization of multiple lidar sensors for autonomous vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 21(5):2139–2145, 2019.

[39] Brígida Mónica Faria, Luís Paulo Reis, and Nuno Lau. A survey on intelligent wheelchair prototypes and simulators. In *New Perspectives in Information Systems and Technologies, Volume 1*, pages 545–557. Springer, 2014.

[40] Richard C Simpson. Smart wheelchairs: A literature review. *Journal of rehabilitation research & development*, 42(4), 2005.

[41] Zhengang Li, Yong Xiong, and Lei Zhou. Ros-based indoor autonomous exploration and navigation wheelchair. In *2017 10th International Symposium on Computational Intelligence and Design (ISCID)*, volume 2, pages 132–135. IEEE, 2017.

[42] The Intelligent Wheelchair Project. URL: https://web.eecs.umich.edu/{~}kuipers/research/wheelchair/.

[43] Takashi Gomi and Ann Griffith. Developing intelligent wheelchairs for the handicapped. In *Assistive Technology and Artificial Intelligence*, pages 150–178. Springer, 1998.

[44] Simon P Levine, David A Bell, Lincoln A Jaros, Richard C Simpson, Yoram Koren, and Johann Borenstein. The navchair assistive wheelchair navigation system. *IEEE transactions on rehabilitation engineering*, 7(4):443–451, 1999.

[45] Christian Martens, Nils Ruchel, Oliver Lang, Oleg Ivlev, and Axel Graser. A friend for assisting handicapped people. *IEEE Robotics & Automation Magazine*, 8(1):57–65, 2001.

[46] ROS.org | Powering the world's robots. https://www.ros.org/ visited 2021-01-28.

[47] Aniket Murarka, Joseph Modayil, and Benjamin Kuipers. Building local safety maps for a wheelchair robot using vision and lasers. In *The 3rd Canadian Conference on Computer and Robot Vision (CRV'06)*, pages 25–25. IEEE, 2006.

[48] Collin Johnson. *Topological mapping and navigation in real-world environments*. PhD thesis, 2018.

[49] Collin Johnson and Benjamin Kuipers. Socially-aware navigation using topological maps and social norm learning. In *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society*, pages 151–157, 2018.

[50] Rodrigo AM Braga, Marcelo Petry, Antonio Paulo Moreira, and Luis Paulo Reis. Concept and design of the intellwheels platform for developing intelligent wheelchairs. In *Informatics in control, automation and robotics*, pages 191–203. Springer, 2009.

[51] Gazebo. http://gazebosim.org/, visited 2021-01-28.

[52] Farzan M Noori, David Portugal, Rui P Rocha, and Micael S Couceiro. On 3d simulators for multi-robot systems in ros: Morse or gazebo? In *2017 IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*, pages 19–24. IEEE, 2017.

[53] Stephen Balakirsky and Zeid Kootbally. Usarsim/ros: A combined framework for robotic control and simulation. In *International Symposium on Flexible Automation*, volume 45110, pages 101–108. American Society of Mechanical Engineers, 2012.

[54] Webots: robot simulator. https://cyberbotics.com/, visited 2021-01-28.

[55] Ana Beatriz Cruz, Armando Sousa, and Luis Paulo Reis. Controller for real and simulated wheelchair with a multimodal interface using gazebo and ros. *2020 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, 2020. doi:10.1109/icarsc49921.2020.9096195.

[56] Augusto Luis Ballardini, Simone Fontana, Axel Furlan, and Domenico G. Sorrenti. ira_laser_tools: a ros laserscan manipulation toolbox, 2014. arXiv:1411.1086.

[57] Andrew Howard and Nicholas Roy. The robotics data set repository (radish), 2003. http://radish.sourceforge.net/, visited 2021-01-28.

[58] Rainer Kümmerle, Bastian Steder, Christian Dornhege, Michael Ruhnke, Giorgio Grisetti, Cyrill Stachniss, and Alexander Kleiner. On measuring the accuracy of SLAM algorithms. *Autonomous Robots*, 27(4):387–407, 2009. doi:10.1007/s10514-009-9155-6.

[59] Eurico Pedrosa, Artur Pereira, and Nuno Lau. A Non-Linear Least Squares Approach to SLAM using a Dynamic Likelihood Field. *Journal of Intelligent and Robotic Systems: Theory and Applications*, 93(3-4):519–532, 2019. doi:10.1007/s10846-017-0763-7.

[60] Stefan Kohlbrecher, Oskar Von Stryk, Johannes Meyer, and Uwe Klingauf. A flexible and scalable SLAM system with full 3D motion estimation. *9th IEEE International Symposium on Safety, Security, and Rescue Robotics, SSRR 2011*, pages 155–160, 2011. doi:10.1109/SSRR.2011.6106777.

[61] Steve Macenski and Ivona Jambrecic. SLAM Toolbox: SLAM for the dynamic world. *Journal of Open Source Software*, 6(61):2783, 2021. doi:10.21105/joss.02783.

[62] Eurico Pedrosa, Artur Pereira, and Nuno Lau. A sparse-dense approach for efficient grid mapping. In *2018 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, pages 136–141, 2018. doi:10.1109/ICARSC.2018.8374173.

[63] Eurico Pedrosa, Artur Pereira, and Nuno Lau. Fast grid slam based on particle filter with scan matching and multithreading. In *2020 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, pages 194–199, 2020. doi:10.1109/ICARSC49921.2020.9096191.