POLITECNICO DI TORINO

Dipartimento di Automatica e Informatica

Master of Science Degree in Mechatronic Engineering

Master's Degree Thesis

Architectures and Methods for Autonomous Indoor Localization and Mapping

Supervisors:

Prof. Marcello Chiaberge

Candidate:

Luca Bini

December 2018

Ringraziamenti

Innanzitutto, un ringraziamento speciale va rivolto ai miei genitori, che mi hanno dato l'opportunità di studiare a Torino e mi hanno sostenuto fino all'ultimo senza alcuna esitazione. Senza di loro non sarei dove sono ora e non avrei conosciuto le persone fantastiche che incontrato qua. Colgo quindi ancora l'occasione per ringraziarli di cuore.

Un altro ringraziamento speciale va ad Alessia, che mi è stata vicino in tutti questi giorni, che per la maggior parte del suo tempo si è presa cura di me e ha sopportato le mie follie. La ringrazio per avermi aiutato senza esitare nonostante ciò di cui mi occupassi non facesse parte del suo ambito. La ringrazio anche solo per quei giorni in cui se ne stava ferma ad aspettare che io prendessi una pausa per abbracciarmi.

Un ringraziamento va a ''Gli studiamici'', con cui ho condiviso la gran parte del mio percorso universitario, in classe e fuori. Sono le persone che mi hanno spinto a partecipare a competizioni di robotica a Roma e al 31° piano del grattacielo dell'Intesa Sanpaolo, facendomi scoprire la mia passione in quest'ambito. Grazie perché mi hanno dedicato parte del loro tempo a migliorare il mio lavoro di tesi, alle volte facendo le nottate con me.

Un ringraziamento è dedicato al professor Marcello Chiaberge per avermi dato la possibilità di lavorare su questo progetto di tesi e per avermi aiutato a migliorare. Ringrazio anche i compagni del LIM con cui ho condiviso quest'ultima parte del mio percorso e Luca, che mi ha seguito fino alla fine anche quando rendevo meno.

Ringrazio il mio coinquilino Alessandro che mi è stato vicino e mi ha aiutato ogni volta che poteva, con cui ho iniziato e concluso la mia carriera universitaria da Udine fino a Torino. Ringrazio inoltre tutti gli amici che ho incontrato in questa nuova città e non mi hanno mai fatto pentire di esserci venuto.

Ringrazio mio cugino Mattia che mi ha aiutato a migliorare il mio inglese, nonostante il poco tempo libero a sua disposizione, e per tutte le volte che mi ha chiamato per farmi scendere la tensione.

Ringrazio infine tutti i miei amici che sono in Friuli, che quando torno a casa mi fanno dimenticare di tutti i miei problemi e mi ricaricano le batterie. Ringrazio il mio migliore amico Marco, che è venuto fino a Torino, oltre che a farsi una vacanza, a trovarmi e a spronarmi a dare il meglio di me.

Abstract

In this thesis work the problem of mapping an indoor environment is handled, using a wheeled robot that autonomously moves and builds the map without the intervention of a human operator.

The robot chosen for this task is TurtleBot3, developed to be compatible with ROS (Robot Operating System), the meta operating system used in this research. Through ROS it is possible to acquire data received from TurtleBot3 laser sensor in "real-time", process them using a Python program and then accomplish tasks like localization, mapping and navigation. In this way the robot can move avoiding obstacles and reporting them into a 2D top-view map of the considered building.

The thesis arises from an opportunity provided by the PIC4SeR, PoliTO Interdepartmental Centre for Service Robotics. The research is conducted at the Mechatronic Laboratory LIM (Laboratorio Interdisciplinare di Meccatronica) of the Politecnico di Torino, in a team of students. Thus, in chapter 1 an introduction of the whole work is performed, introducing the environment of the project and the allocation of tasks within the team.

In chapter 2 the software environment of ROS and the hardware environment of the TurtleBot3 are analysed. In chapter 3 the probabilistic theory needed to get into the thesis context is studied. Hence, an introduction to probabilistic filters is accomplished, followed by a research on Bayes filter and the probabilistic algorithms that arise from this latter.

From the various available algorithms, the Occupancy Grid Mapping is adopted for the realization of the map. In chapter 4, the reader can look at the path from the choice of the probabilistic filter to the development of the Python program that carries out the build of the map. The mapping program divides the environment in a fixed-number of fixed-dimension cells and assigns to each cell a value, stating the probability that the cell is occupied by an obstacle or not.

Finally, in chapter 5 results are presented. A comparison is accomplished between the developed mapping algorithm and Gmapping, a common algorithm for the

build of 2D maps using a laser scanner. Some problems arise with the robot localization and these also affect the reliability of the map.

Sommario

In questa tesi è stato affrontato il problema di mappare un ambiente interno, utilizzando un robot su ruote che autonomamente si muove e costruisce la mappa senza l'intervento di un operatore umano.

Il robot scelto per questo compito è il TurtleBot3, sviluppato per essere compatibile con ROS (Robot Operating System), il meta sistema operativo usato in questa ricerca. Attraverso ROS è possibile acquisire i dati ricevuti dal sensore laser del TurtleBot3 in ''tempo-reale'', processarli utilizzando un programma Python e quindi svolgere determinati compiti come la localizzazione, la mappatura e la navigazione. In questo modo il robot può muoversi evitando gli ostacoli e riportandoli in una mappa 2D vista dall'alto dell'edificio considerato.

Questa tesi nasce da un'opportunità offerta dal PIC4SeR, PoliTO Interdepartmental Centre for Service Robotics. La ricerca è condotta al laboratorio di meccatronica LIM (Laboratorio Interdisciplinare di Meccatronica) del Politecnico di Torino, in un team di studenti. Quindi, nel capitolo 1 viene eseguita un'introduzione dell'intero lavoro di tesi, presentando l'ambiente del progetto e la suddivisione dei compiti all'interno del team.

Nel capitolo 2 l'ambiente software di ROS e l'ambiente hardware del TurtleBot3 vengono analizzati. Nel capitolo 3 è studiata la teoria probabilistica necessaria ad entrare nel contesto della tesi. Quindi, viene eseguita un'introduzione sui filtri probabilistici, seguita da una ricerca sul filtro di Bayes e sugli algoritmi che nascono da quest'ultimo.

Tra i vari algoritmi disponibili, la Mappatura a Griglia di Occupazione viene adottata per la realizzazione della mappa. Nel capitolo 4, il lettore può osservare il percorso dalla scelta del filtro probabilistico fino allo sviluppo del programma Python che porta a termine la costruzione della mappa. Il programma di mappatura divide l'ambiente in un numero fisso di celle di fissa grandezza e assegna ad ognuna un valore, il quale attesta la probabilità che la cella sia occupata da un ostacolo o meno.

Infine, nel capitolo 5 vengono presentati i risultati. Viene eseguito un confronto tra l'algoritmo di mappatura sviluppato e Gmapping, un comune algoritmo per la costruzione di mappe 2D utilizzando i dati provenienti da un scannerizzatore laser. Alcuni problemi sorgono per quanto riguarda la localizzazione del robot e questi compromettono anche la credibilità della mappa.

Index

1 Introduction

The scope of this work is the implementation of an algorithm to solve the Simultaneous Localization and Mapping (SLAM) problem for indoor environment, without the human handling.

This task is accomplished using a wheeled robot named TurtleBot3 Burger. This robot must be able to navigate into an area avoiding all the obstacles and, at the same time, to report them into a map. For the realization of the map, the robot must track its position and orientation in time, with respect to an initial fixed reference frame. It must be localized into the built map. Without localization, the robot cannot associate the relative measurements gathered by its sensors to the fixed reference frame of the map. Then, the map cannot be built. At the same time, the robot cannot localize itself without a map where it can be located. This is the chicken-and-egg problem. Localization and mapping processes must proceed simultaneously and cannot exist without each other.

The thesis arises from a project of the PIC4SeR, PoliTO Interdepartmental Centre for Service Robotics. The program for the autonomous navigation has been developed by Lorenzo Galtarossa, another student of Politecnico di Torino. Instead, the mapping algorithm is the object of this work. The thesis will focus on the mapping purpose, but it is also necessary to perform the robot localization.

For the localization task, the wheel encoders are used to take odometry data. From these data position and orientation of the Burger are evaluated. Since only encoders are used to localize the robot, the developed algorithm can be executed only in indoor environment. On a flat floor, encoders are quite reliable. If the Burger is placed on the ground, when it deals with holes or a rough terrain, the encoders get an excessively faulty odometry and the localization is lost.

The mapping task is accomplished by means of the 360° laser scanner of the Burger. Laser data sets are collected and processed to build a top-view 2D map of

the environment. Map must report all the obstacles detected by the laser sensor. Sometimes the robot detects moving objects and walking people. Random moving obstacles must be avoided and removed by the map.

The presented program, that carries out the Simultaneous Localization and Mapping task, is a probabilistic filter algorithm. This algorithm is developed using the Python language.

The achieved mapping algorithm is an open-source code. In the following chapters, all the needed theoretical and practical concepts are provided to the reader, to allow him to interact with and implement the program.

2 Software and Hardware

The software and hardware environment, in which the thesis research is carried out, is introduced in the following sections. As for the software, the thesis project has been conducted on the ROS environment. As hardware for the thesis work a robot called TurtleBot3 has been employed. The third section consists in a brief guide to set up the working environment. An introduction to the useful and used commands is accomplished to provide to the reader a complete insight into the used tools.

2.1 ROS

The thesis work is developed in a software environment called ROS (Robotic Operating System). The readers that already know ROS, can skip to the next section, while now a brief introduction on ROS will be carried out.

Like its name suggests, ROS is a meta-operating system. It is not a real operating system like Linux or Mac OS X (ROS has to be installed in the current operating system), but it has low level device control, implementation of commonly used functionalities, package management, share of messages between processes and it works with tools and libraries to obtain, write, build and run codes across multiple computers, like an operating system. Its goal is to improve and simplify the programming and the use of robots, creating a framework of processes that are loosely coupled at runtime. Through ROS, both the programmer and the user can manage a wide amount of data that a remote computer, one or more robots and a Cloud server on internet (if exploited) need to share with each other, making use of an internet connection. The interaction with a Cloud is possible but it is not topic of this work, so it will be left aside.

A ROS runtime framework is a network where every single process is a base unit, called *node*. In this way, every task is performed by its own node. For example, to handle a wheeled robot: there will be a node to manage the laser scanner, another node for the camera, another one for the wheels control, etc. So, there is a node for every requested task, but who handles all the nodes? How do nodes share their data with each other?

To answer the first question a special node must be introduced, the *ROS Master* node. This is the first node that must be launched, usually from the remote computer and not from the onboard robot computer. The reason is that remote computer has in general a faster processor, but above all, onboard computer must be as free as possible to not slow the robot functionalities. Indeed, the latter must control all the processes for the robot motion and sensors operation. Moreover, this node has to be unique. There must be only one Master node for ROS network. Every other node, when started up, automatically registers itself on the Master. Thus, the ROS Master holds the task to manage all the nodes and their behaviours, which information a node must send and which one it has to read. For the sake of simplicity, the reader can understand better this structure from the following figures.

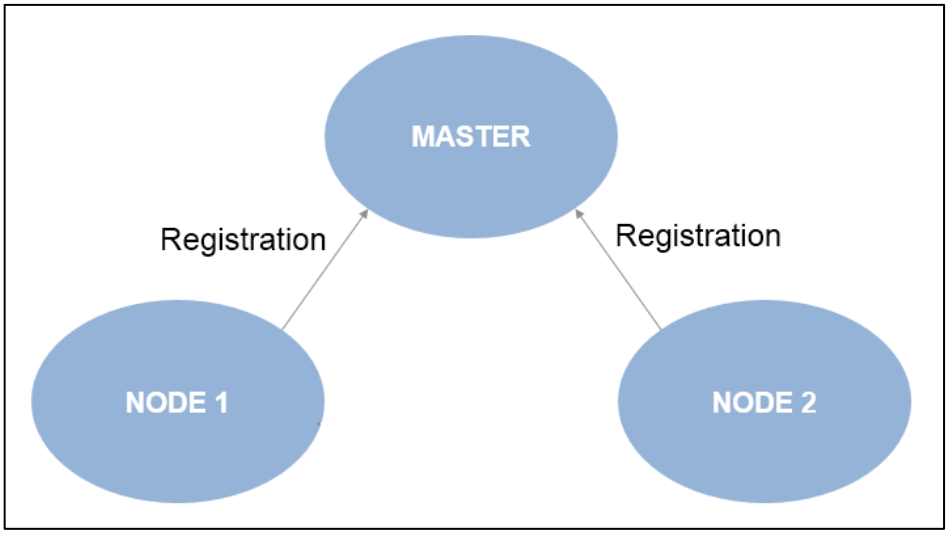


Figure 2.1

The missing part concerns how nodes share data with each other. They do it in two ways: through *topics* or *services*. In the first case, nodes exchange information publishing and subscribing messages on topics. It is like a notice-board, where each node can post a message of a defined type or it can read a message of that type posted by another node. For each topic, a node can only publish or subscribe to it at the same time, not both, but there is a potentially infinite number of nodes that can publish or subscribe to this topic. In the case of services, there is a server-client model, where a node registers a service and then, any other node can ask for it and get a reply. With this process, there is a mutual exchange of information, because the request has the possibility to hold some data too, contrary to topics system. For the purposes of this work, the best solution for data sharing between nodes is the use of topics.

As a practical example, it is considered a robot that moves somewhere in some way, equipped with a laser scanner in such a way that the user is advised, on its laptop, when the robot comes up against an obstacle. The program that manages this behaviour is run on the remote computer, as usual, to leave the robot computer free from laborious calculations. How does ROS handle this task? Firstly, the Master node is launched from the remote PC, then a node is started up by the robot onboard computer, it registers to the Master and publishes data coming from the laser sensor to a topic, that can be called "/advisor". Finally, a node is started from the remote pc and it registers to the Master too, but this time it subscribes to the topic /advisor. This node, when reads data from /advisor, stamps, on the screen of the remote computer, an alert message with some information about the distance of the robot from the obstacle detected. The *Figure 2.2* displays this structure.

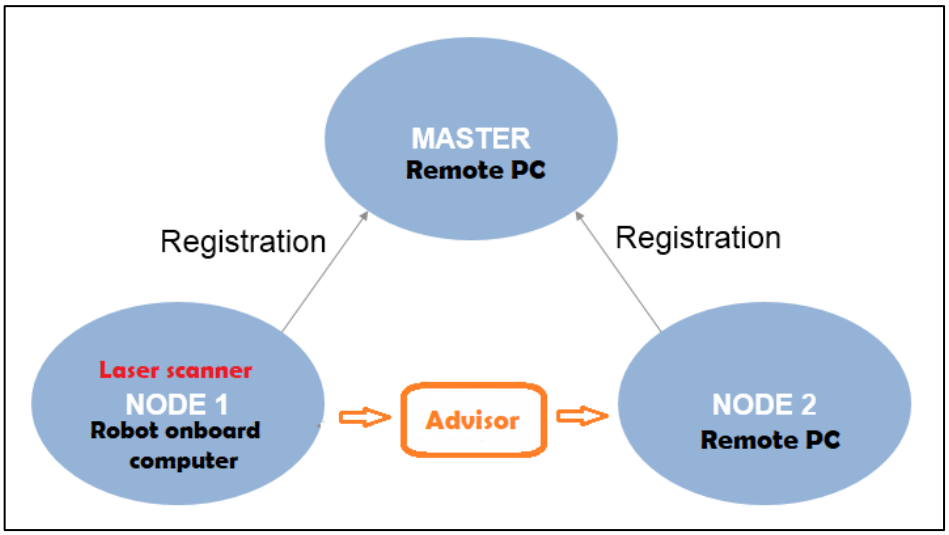


Figure 2.2

It has been said that nodes are the base units of a ROS framework. Every single node is a piece of a complex program and it must perform one of the various tasks. Thus, a single node is a program written in C++ or python language and there may be more nodes into the same ROS system, running at the same time, using a different programming language. This possibility is a strength of ROS, that can match these diverse codes. The user can start up a node individually or several of them, using a launch file. This type of file is an XML code, a markup language similar to HTML, that, when launched by the user, starts up nodes with the possibility of setting parameters, remapping topics to which the nodes publish or subscribe, changing the nodes names used in the ROS network and other minor features. All the C++, python, launch files and other ones needed to set up parameters and environments, like a simulated world for a robot, are collected into ROS packages. When a package is built on a computer, three folders appears into the package: the *src* one contains the source code and its files can be edited, the *build* and *devel* ones must not be modified because are only used by the system at build and run time.

To check the ROS tree made by nodes, topics and their connections, the *rqt_graph* tool is used. This tool allows to get a graphical display of the data flow in the ROS system. Then, rqt_graph is useful for checking and debugging the nodes codes,

observing the correctness of links and, in case, which nodes are excluded from the tree.

A simple exemplifying case is given by the ROS tree shown below in *Figure 2.3*. There are 3 nodes: the Master, a node (called for example "/talker") publishing a fixed string, every 10 seconds, on a topic (named "/chatter") and another node (named "/listener") subscribed to the same topic.

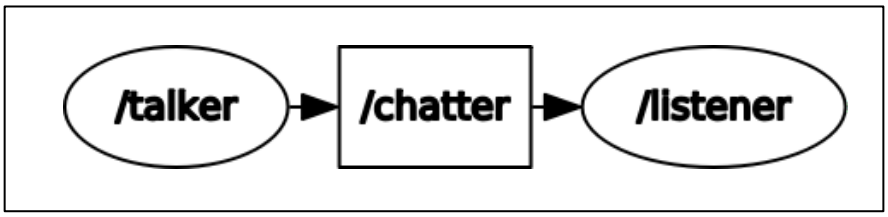


Figure 2.3

It can be noted that, in a rqt_graph, the nodes are represented as ovals, topics as rectangles and the arrows, linking nodes and topics, point out if data flow from a node to a topic (publication) or vice versa (subscription). In this example the /talker publishes any fixed sentence, for example "Hello!", after every 10 seconds and the /listener reads the string, whenever available, and it does nothing else.

2.2 TurtleBot3

Regarding the hardware environment, the devices employed in this work are a remote computer like a laptop, that acts as a control centre because of its faster processor, and a TurtleBot3, a two-wheeled robot manufactured by ROBOTIS. The latter is an open source robot, both from the hardware and software point of view, born to be programmed in the ROS environment. Indeed, the TurtleBot series (TurtleBot, TurtleBot2 and TurtleBot3) is the standard platform for students and developers that have their first approach with ROS. TurtleBots are built with the purpose of teaching how to program a device in ROS, so there is a wide

documentation on this subject on the websites of ROBOTIS[4] and ROS Wiki[1]. For these reasons, the most recent TurtleBot3 has been selected to work in the ROS system.

The third version of TurtleBot is available on the market in three models. All of them are composed of three or four support plates of injection molded plastic arranged one on top of the other, supported from thin metal cylinders. The material used for the plates allows to achieve a cheaper and, at the same time, open source hardware device. In fact, it is possible to download the 3D CAD model of the plats and to produce them with a 3D printer.

In the figure below the three TurtleBot3 models are shown.

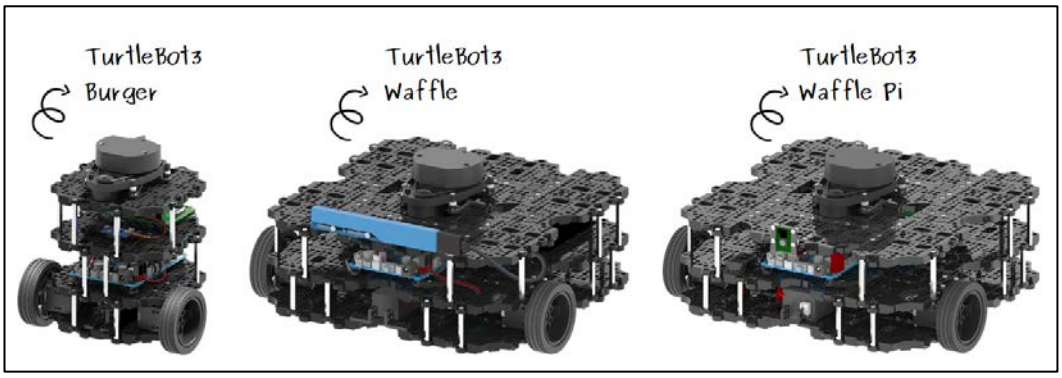


Figure 2.4

Each TurtleBot3 has a single-board computer (SBC), two electric motors joint to two wheels, one ball caster for the Burger model and two ball casters for the Waffle and Waffle Pi models, one LIPO battery of 11.1V and 1,800mAh, a controller board (OpenCR1.0) to which the SBC, the motors and the battery are connected and finally some movement and detection sensors. Regarding the latter, all three models are fitted with three-axis gyros, accelerometers and magnetometers, wheel encoders and one 360 Laser Distance Sensor LDS-01, connected to the SBC through a small interface board USB2LDS. These are the parts common to the three TurtleBot3 models, now the differences will be shown.

Considering *Figure 2.4*, the first model on the left is the Burger one, more compact in base dimensions but taller than the other two, 138 x 178 x 192 (L x W x H, mm). It is composed of four support layers, on which there are: on the bottom one two Dynamixel XL430-W250 motors and the LIPO battery, on the lower middle one the OpenCR1.0, on the higher middle one a Raspberry Pi 3 Model B, which acts as SBC, and the USB2LDS and on the top layer the Lidar LDS-01. From this arrangement derives the name Burger. The other two models are the Waffle and the Waffle Pi. They are very similar in structure: 281 x 306 x 141 (L x W x H, mm) and three support plates. On the bottom layer there are two Dynamixel XM430-W210 motors and the LIPO battery, on the middle one the OpenCR1.0, the SBC and the USB2LDS are located. The SBC mounted on Waffle is an Intel® Joule™ 570x, whereas the SBC of the Waffle Pi is a Raspberry Pi 3 Model B, like in the Burger model. On the top layer there is the Lidar, like on the Burger, and in addition a camera, an Intel® RealSense™ R200 for the Waffle and a Raspberry Pi Camera Module v2.1 for the Waffle Pi. These are the structures of the three TurtleBot3 models. For a better understanding of the hardware part of these robots, now the individual hardware devices are analysed one at a time.

Figure 2.5

The SBC of the Burger and the Waffle Pi is the Raspberry Pi 3 Model B. It is an open-source single-board computer equipped with an Ethernet port and Wi-Fi

connection. It has one CPU 64-bit quad-core ARM Cortex-A53 at 1.2 GHz and a RAM of 1 Gigabyte shared with GPU. The most common operative systems officially released by the manufacturer are based on GNU/Linux, but the board can support other operative systems, like Windows 10 IoT.

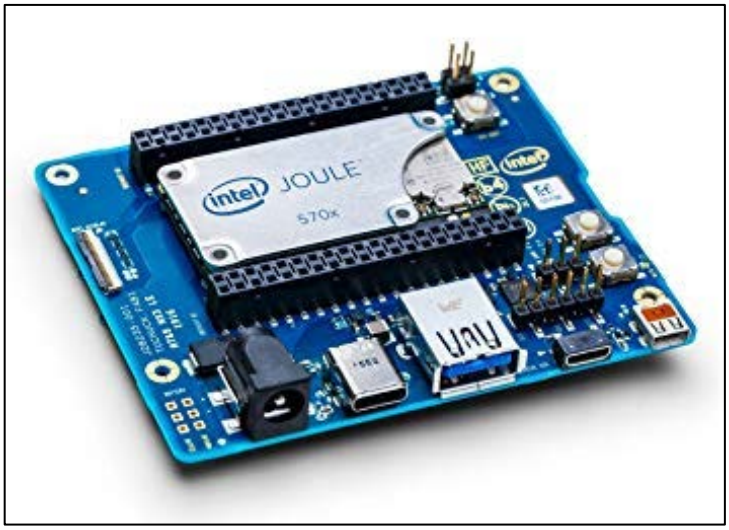


Figure 2.6

The SBC mounted on the Waffle, instead, is an Intel® Joule™ 570x. Its Computer Processing Unit is a 64-bit quad-core Intel® Atom™ T5700 at 1.7 GHz with burst up to 2.4 GHz. This single-board computer has a 4 GB LPDDR4 RAM and a Wi-Fi connection. The operative system released from the manufacturer is Linux. It is made to support the Intel® RealSense™ cameras and their libraries. In fact, the Waffle has an Intel® RealSense™ R200 camera mounted on itself.

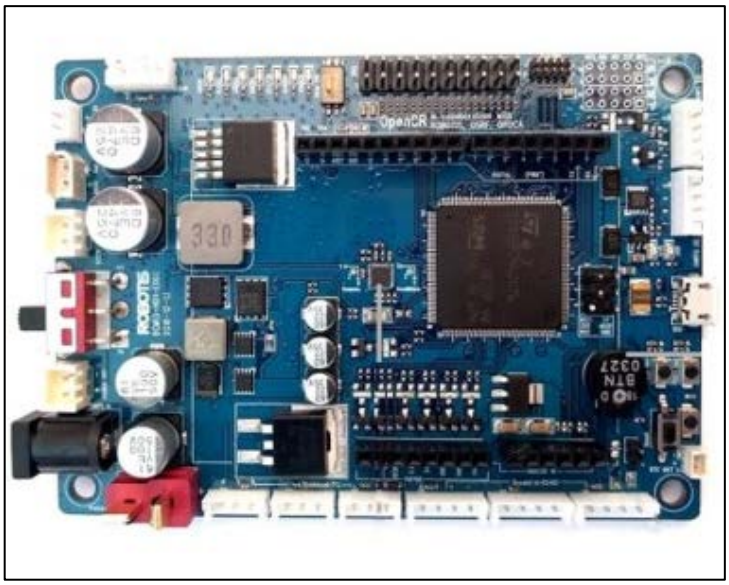


Figure 2.7

As mentioned earlier, the various SBCs are connected to the battery and the motors through the OpenCR1.0. OpenCR stands for ‘‘Open-source Control module for ROS’’, it is a controller board developed for ROS embedded systems, supplying open-source hardware and software. The main chip of the OpenCR1.0 is based on a 32-bit ARM Cortex-M7. Arduino IDE is the software used to manage the board. The OpenCR1.0 has the 3-axis gyroscope, 3-axis accelerometer and 3-axis magnetometer mentioned a few paragraphs above. Thanks to these sensors, the board can provide an ‘‘IMU’’ measure, available on ROS as messages published on the topic */imu*. A message of this type holds the measures of ‘‘orientation’’, ‘‘orientation_covariance’’, ‘‘angular_velocity’’, ‘‘angular_velocity_covariance’’, ‘‘linear_acceleration’’ and ‘‘linear_acceleration_covariance’’.

Figure 2.8

Take the distance and visual sensors into consideration. All three models of TurtleBot3 has on top of their structure a 360 Laser Distance Sensor LDS-01. It is a 360-degrees 2D laser scanner with a resolution of 1 degree, that detects obstacles in a plane parallel to the ground plane. Thus, it rotates at 5 Hz frequency, taking one distance measure every 1 degree for a total of 360 measures per round and performing 5 complete rotations per second. So, the LDS-01 can take a measure for each its relative position every 0.2 seconds. The distance range is 120 mm – 3500 mm, with distance accuracy of ± 15 mm from 120 mm to 499 mm and ± 5.0% from 500 mm to 3500 mm. Instead, the distance precision is ± 10 mm from 120 mm to 499 mm and ± 3.5% from 500 mm to 3500 mm.

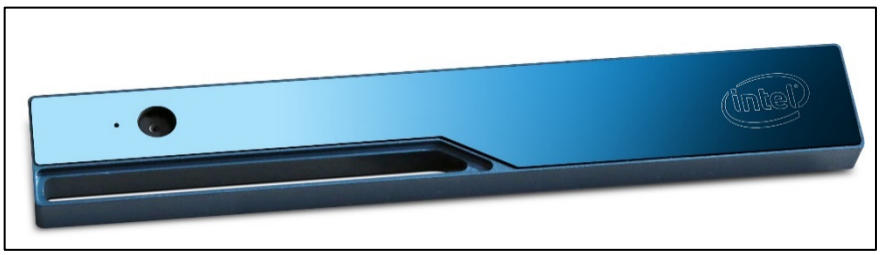


Figure 2.9

The camera mounted on Waffle is the Intel® RealSense™ R200, compatible with the SBC of the robot, the Intel® Joule™ 570x. This is a 3D RGB-Depth camera, that can provide color, depth and infrared video streams. With this device it is possible to carry out object, gesture and scene recognition thanks to the depth sensor. To analyze the device technical specifications, the color and depth sensors must be considered separately. The RGB camera has a video resolution of 1920 x 1280 and 2M, works on 30 fps and has (70±2)° and (43±2)° of horizontal and vertical field of view respectively. The infrared system is composed from an IR laser projector and two IR cameras, right and left. This IR depth system has a resolution of 640 x 480 VGA, works on 60 fps and has a horizontal/vertical field of view of (59±5)°/(46±5)°. The overall of the camera range in nominally between 30 cm and 400 cm.

Figure 2.10

The camera used for the Waffle Pi is the Raspberry Pi Camera Module v2.1. This camera is perfectly compatible with the Raspberry Pi 3 B used as onboard computer. It is an RGB camera that can take high-definition videos with its 8 megapixels. It is mounted on this TurtleBot3 model because it is not very expensive and easy to use for who is approaching to the ROS environment. The sensor resolution is 3280 x 2464 pixels, the video modes are 1080p at 30Hz, 720p at 60 Hz and 640x480p at 90 Hz. Then 90 fps is the maximum frame rate. Lastly,

the camera has 62.2° of horizontal and 48.8° of vertical field of view and the focus range, given from the onboard fixed focus lens, is 1 meter to infinity.

This was the last noteworthy hardware device mounted on TurtleBots3. Now the processes performed in this thesis work, starting from the operating system and software installation on the various used devices, are analyzed.

2.3 ROS and TurtleBot3 configuration

This thesis has been developed using a laptop as remote computer and the Turtlebots3 employed are the Burger and Waffle models. The ROS version recommended to be run on TurtleBots3 is ROS Kinetic Kame, released the May 23rd, 2016. The Kinetic version is primarily target for the Linux Ubuntu 16.04 LTS release, but it is supported by other Linux systems, Mac Os X, Android and Windows at various degrees. So, Ubuntu 16.04 LTS with ROS Kinetic has been installed in the three devices.

On ROBOTIS e-Manual website[4], under the section "Setup→PC Setup" there is a "Download link" to download the ISO file of Ubuntu 16.04 for the laptop. Instead, under the section "Setup→SBC Setup" there are links to download the ISO file of Ubuntu MATE 16.04 for both the Raspberry Pi 3 and Intel® Joule™ 570x boards. The MATE version of Ubuntu is very similar to the original one, but developed for SBCs boards. In order to install Ubuntu 16.04 and Ubuntu MATE 16.04 a USB flash drive is needed, in which the Ubuntu image file is put, creating a bootable Ubuntu USB stick. To make this type of bootable sticks there is a tutorial on the official website of Ubuntu[8] for each of the operating systems Microsoft Windows, Mac OS X and Linux. When the bootable Ubuntu stick is ready, Ubuntu can be installed on devices. Considering the laptop PC, the installation starts inserting the USB flash drive and entering the system-specific boot menu. From this menu it is possible to select the USB device to boot and start installing Ubuntu with the desired settings. The installation procedure for the Turtlebot3 Burger with the Raspberry Pi 3 is the same as just seen. The only difference is the use of a USB flash

drive with the ISO of Ubuntu MATE 16.04. For the Turtlebot3 Waffle the installation of Ubuntu MATE is more complicated because of the need for a second USB drive with Ubuntu Core 16 image for the Intel® Joule™. Moreover, the board may require having its BIOS updated to version #193. For the Ubuntu installation on Waffle, it is better to refer to the web page of ROBOTIS[4], section "Setup→SBC Setup→Intel Joule 570x", linked to the Ubuntu official website[9].

Once Ubuntu is been installed and configured on the different devices, the procedure to install ROS Kinetic is available on ROS Wiki website[10]. It is important to pay attention to the initially required setups and, when asked, the "Desktop-Full" installation is recommended, to add all the available packages for ROS, robot tools and robot simulators. When the environment setup and the installation of dependencies for the build of packages are completed, it is necessary to return to the previous ROBOTIS page. At this point, the following step is the installation of "Dependent Packages" for TurtleBot3 control both from the remote PC and from the TurtleBots3 themselves. For the latter there is a further step for the removal of some packages needed only for the laptop, not the onboard SBC. Thus, the packages are built. For the TurtleBots3 there is an extra command to launch to allow the use of USB port for the OpenCR1.0 without acquiring root permission.

The last step before using TurtleBots3 with ROS is the "Network Configuration", always described in the e-Manual of ROBOTIS website. The ROS framework is based on an internet network, so each device must be configured to robustly connect itself with the same ROS structure. Then a static IP address is needed for each device, so that each one has a permanent number assigned to it. For this reason, it is used a private Wi-Fi internet network, where the laptop and the TurtleBots3 are connected. There is the possibility to employ more devices connected to different private Wi-Fi networks, but it is out of the scope of this work. Then, always using the same Wi-Fi network, the IP addresses of the laptop and of the TurtleBots3 are found using the command

```
$ ifconfig
```

from a terminal window of each computer. The searched IP is the number tagged as "inet addr" into the "wlp2s0" section. When the IPs are been identified, the *.bashrc* file located in the *Home* folder can be modified. This is a fundamental step. In the .bashrc file two lines must be added at the end of the document:

export ROS_MASTER_URI=http://<remote_PC_IP>:11311

export ROS_HOSTNAME=<current_PC_IP>

Both in the laptop and in the Turtlebots3 the ROS_MASTER_URI carries the IP of the laptop, because this parameter identifies the IP where the current device searches for the ROS Mater node. On the contrary, the ROS_HOSTNAME has the IP of the working device, because the parameter identifies the current device to be connected to the ROS Master before indicated. This step closes the installation and setup of the laptop and the TurtleBots3.

The TurtleBot3 Waffle Pi, not considered here, has the same SBC Raspberry Pi 3 of the Burger model, so it can be handled in the same way for the Ubuntu and ROS installation. Once Ubuntu MATE 16.04 and ROS Kinetic Kame have been set up, since there are no differences in the use of the same operating system on a Turtlebots3 with Raspberry Pi 3 or Intel® Joule™ 570x, from now on the discussion will be focused only on the Burger model.

Now the ROS system can be started. The laptop and the SBC of the TurtleBot3 Burger are powered on. The robot is in this moment connected to a screen, a mouse and a keyboard, like a desktop PC. The first action to be completed is the start-up of the ROS Master node to which all the other nodes have to register. Then, from the laptop a first terminal window is opened and the command

$ roscore

is launched. This command stays active until the node is shut down, so the window is unusable for other actions and can be minimized. It is time to start the TurtleBot3 functionalities, bringing up the basic packages. For this purpose, they are launched from a terminal window of the robot:

$ roslaunch turtlebot3_bringup turtlebot3_robot.launch

The first word identifies the start of a *.launch* file on ROS, ‘‘turtlebot3_bringup’’ is the ROS packages where this file is located and ‘‘turtlebot3_robot.launch’’ is the file to launch. If the user opens this file, he discovers that it starts up other two .launch files. They are ‘‘turtlebot3_core.launch’’ and ‘‘turtlebot3_lidar.launch’’. The first is the program that starts all the nodes relative to the motion control of the robot wheels and the measurements of IMU and encoders sensors. The second program starts the node of the Lidar sensor. Using a Waffle or a Waffle Pi the commands

$ roslaunch turtlebot3_bringup turtlebot3_realsense.launch

$ roslaunch turtlebot3_bringup turtlebot3_rpicamera.launch

must be launched respectively on a second terminal of the robot SBC, to start the nodes of Intel® RealSense™ R200 and Raspberry Pi cameras. Once these launches are completed, screen, mouse and keyboard can be removed from the robot and this can be placed on the floor. The effective control of the robot takes place through the laptop. Then, to connect the remote computer to the TurtleBot3 via ROS, a second terminal is opened on the laptop and the commands

$ export TURTLEBOT3_MODEL=burger

$ roslaunch turtlebot3_bringup turtlebot3_remote.launch

are launched. The first line is used by the system to identify the model of the TurtleBot3 to which the laptop must be connected. Working with the Burger model, the TURTLEBOT3_MODEL parameter is set to ‘‘burger’’. All the .launch files seen so far are necessary for the operation of the ROS working environment, thus the terminals, where they have been started, cannot be closed until the end of the robot activity.

At any time, it is possible to check all the nodes and the topics active in the ROS framework. To see them another terminal is opened from the laptop, then several commands can be used.

$ rosnode list

This shows the list of the active nodes. If the user wants to check some other parameter like the frequency at which a node posts or reads something from a topic, he can write “rosnode” and, instead of “list”, push twice the TAB button to make appear a list of possible related commands. These are useful for handling nodes in ROS:

- $ rosnode info /“node_name”: displays on screen information about a node, with publications and subscriptions;
- $ rosnode kill /“node_name”: kills a node process disactivating that node;
- $ rosnode list: shows the list of active nodes;
- $ rosnode machine /“machine_name”: shows the list of nodes running on a specific machine;
- $ rosnode ping /“node_name”: pings a node repeatedly;
- $ rosnode cleanup: deletes the registration of any node that cannot be immediately contacted.

To show the list of the topics used by the nodes, the following command must be launched:

$ rostopic list

In the same way, if “rostopic” is written in the terminal and TAB is double pressed, the possible commands used to check a specific topic appears.

- $ rostopic bw /“topic_name”: displays on screen the bandwidth used by a topic;
- $ rostopic delay /“topic_name”: displays on screen the delay for a topic which has a header;
- $ rostopic echo /“topic_name”: displays on screen messages published by a topic;
- $ rostopic find “msg_type”: finds topics filtering them according to the message type;
- $ rostopic hz /“topic_name”: displays on screen the publishing rate of a topic;

- $ rostopic info /“topic_name”: displays on screen information about a topic;
- $ rostopic list: shows the list of active topics;
- $ rostopic pub /“topic_name” “msg_type” “msg”: publishes the message “msg” on a topic;
- $ rostopic type /“topic_name”: displays on screen the type of a topic.

To show the complete ROS structure tree, the tool called *rqt_graph* is used. This has been presented in the first subchapter. It displays all the nodes and topics active in that moment, using arrows to represent a publication or a subscription to a topic for posting and reading messages. After the bringing up of the TurtleBot3 Burger, the ROS tree of nodes and topics sketched by rqt_graph is depicted in *Figure 2.11*.

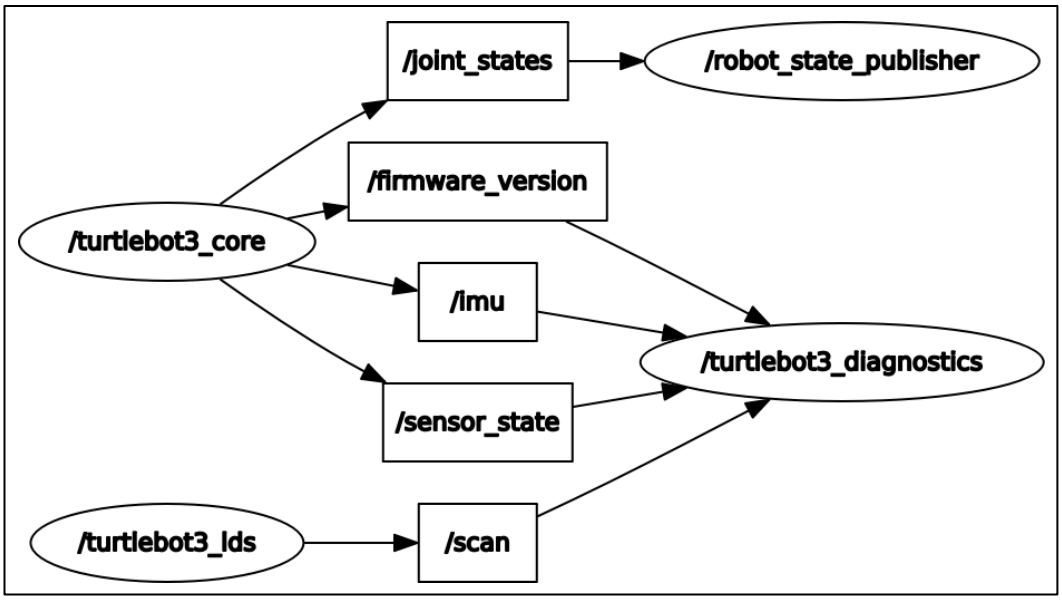


Figure 2.11

Now the Burger is ready to run any compatible package or script. There are various ROS packages available for the SLAM purpose, some developed from researches but still ongoing projects, some completed, tested and perfectly functioning. An example of the latter is the tool named *Gmapping*, the most common ROS package aimed at building a map. Gmapping makes a progressive 2D map of the

environment as the robot moves in it. The map can be visualized on the screen of the laptop in real time, during the building, running the viewer tool *rviz* in a new terminal window. The sensor used by Gmapping package is the Lidar of the Burger. In *Figure 2.12* there is a screenshot of the system running Gmapping and rviz, during a mapping process. In this case the robot moves controlled by the user through the keyboard of the laptop, using a ROS package named *teleop*.

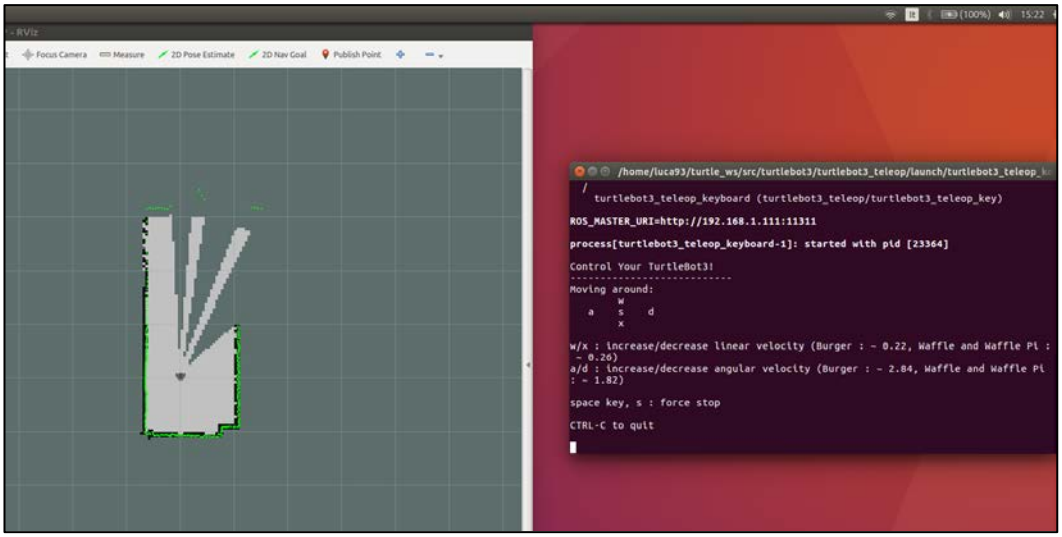


Figure 2.12

In the example shown in this figure, a limitation of Gmapping tool is revealed. In order to map the whole environment, the robot must be moved by the user, since it is not autonomous. To implement some kind of autonomous navigation algorithm, it may be necessary to access the map in image format, for selecting the next goal to reach. This is very hard using the map created by Gmapping. Thus, it is necessary to work with a mapping program that provides an easily accessible map, in order to guarantee an efficient autonomous navigation. Therefore, it was chosen to build a progressive 2D map using a program written in Python language, through the 360° laser sensor mounted on TurtleBot3. Since this work consists in writing a mapping program from scratch, it has been decided to build a 2D map, for a simpler implementation and debug. Switching to 3D mapping can be

considered as a future implementation, achievable using data coming from cameras of Waffle and Waffle Pi TurtleBot3 models.

Thus, the first step is to study the probabilistic filters theory, needed for selecting and employing the best filter suitable for building a reliable map.

3 Probabilistic filters

This chapter aims to provide the reader with a higher level of awareness about the probabilistic theory. Probabilistic filters are introduced from the general to the particular. Firstly, the discussion enters into the detail of the Bayes filter, the most common basic filter. From this, the two main macro groups arises: Gaussian filters and Non-parametric filters.

3.1 A general view on probabilistic filters

When studying a robot that navigates in the real world, a probabilistic approach is needed. A probabilistic model is necessary to partially compensate for the uncertainty in robot perception and action.

Every sensor and actuator hold a certain grade of uncertainty. A fixed Lidar can detect the same fixed object at different ranges, in different time instants. This behaviour can be experienced positioning the TurtleBot3 Burger in front of a straight wall at a distance of 0.5 metres, hence checking data taken from the 360° Lidar and sent to the */scan* topic. The results of this test are reported in the table in Appendix A, where only the first measure of each vector data, corresponding to the angle 0°, are considered. The real distance from the wall is around 0.5 metres, because of the error given by the manual positioning of the robot. How far is the TurtleBot3 from the wall then? There are three more frequent distances in the set of data, not one. These and the other less frequent measures belong to the uncertainty associated to sensor detection, instead the "0.0 m" data represents a failure of the device.

Similarly to uncertainty related to the laser sensor, the other sensors and the wheel motors of a robot have their degree of precision. Thus, each sensor holds its own accuracy and precision and its measurements must have their weight of

reliability. The error in robot sensing can be caused directly from sensors used for objects detection, or indirectly from other upstream sensors and actuators.

As it has been shown through the Lidar of the Burger, the environment perceived from the robot can be different from the real one. Probabilistic algorithms could represent a valid compensation to the various degree of uncertainty given from information received by the robot. A good rule of thumb is the following:

> "A robot that carries a notion of its own uncertainty and that acts accordingly is superior to one that does not." [11]

A probabilistic approach works better than a not-probabilistic one in not completely known environments and it is necessary in unknown, unstructured and complex areas. Estimation problems like the famous "kidnapped robot problem", in which robot must recover from localization failure, or other tasks like mapping a huge facility, need the same type of approach for being carried out. A probabilistic algorithm is more robust and it can deal with sensors noise, sensors limitation, dynamism of the environment and so on.

The goal of this work is the building of a 2D map using a 360° laser sensor. A map is a scan of the environment around the robot, represented by obstacles and free spaces. A problem arises suddenly when objects are detected by the Lidar like series of points. An obtained point may not match with the real position of the found object, but this latter can be located in an area around that point, with an uncertainty diameter smaller according to the accuracy of the laser sensor. Then the object can be at the measured distance, closer or further from the laser device. The detected point located one degree after the previous one, belonging to the same object, has the same issue. Hence it is difficult perfectly recreating the environment in a map, because even a straight line is collected like a 2D point cloud, making hardly to understand its real direction. This is a first reason for implementing a probabilistic algorithm, that takes into account the uncertainty arising from the point cloud. Relying blindly on measurements of the Lidar brings to an inaccurate map. Let us consider a wall. If the same point of it is detected at a certain distance and one second later at another closer distance, the wall

represented in the map will be thicker than a line, since this logic still stands for all points of the wall. In this case the map is faulty and using an algorithm to obtain a midline, that represents the wall, can be dangerous, because the robot could hit it.

A second reason for which the implementation of a probabilistic approach is a better solution, is the possible presence of moving objects and persons. If the robot has to map an office with people working, getting around, moving objects or with other robots in movement, the built map must not include them. For this purpose, a probabilistic algorithm can easily ensure that dynamic obstacles are not reported, however providing identification and avoidance.

In the studied context, a probabilistic approach is implemented like a probabilistic filter. This filter makes quantities like sensor measurements, controls, robot states and map states to be modelled as random variables. These latter follow probabilistic laws and are derived from other random variables, such as those modelling sensor data. Using a probabilistic approach, a variable does not assume one single value of a set of possible values, named *sample space*, but at each point of this space a relative likelihood is associated. The function that accomplishes this association is a *probability density function*.

There has been talked about robot and map states. A *state* of a system is a collection of variables which fully describe that system. For the robot state, these variables can be position, orientation, velocity, angular velocity, acceleration, etc. The map has its state too, that represents its content, its information according to the used mapping approach. A state can be *static* if its variables do not change value in time and it is *dynamic*, on the contrary, when the variable values change over time. In a probabilistic environment, states are usually dynamic. Robot state changes within the time and during the navigation. Map state changes when detecting walking people or when an object is moved.

Another important characteristic for a state is the *completeness*. A state can be considered complete when the knowledge of past states, past robot controls and past taken measurements does not add any additional information, compared to

the present state, for the prediction of the future next state. Then, in the case of complete state, the past and the future states are independent.

Working in a real environment and not in a simulated one, it is important to consider not a deterministic future, but a *stochastic* one. In the real world everything can happen, a robot controlled to move always straight ahead can step on an unexpected pebble and deviate its direction. Or during a mapping process a person can get into the visual field of the robot and erroneous measurements bring the map state to be incorrect. Thus, the best solution is to work with a stochastic model describing the future.

A stochastic process where the future of the process can be predicted based only on the present state, as well as knowing its full history, is considered to be a *Markov chain*, or *Markov process*. The property that a stochastic process has to satisfy to be a Markov chain is named Markov property, whose concepts are similar to those of the completeness property. Therefore, a complete state modelled in a stochastic environment is a Markov chain.

Any Markov chain has a state space that can be continuous or discrete. An example of continuous state space is that formed by a robot pose with its position and orientation, hence continuous variables. An example of discrete space is any binary state variable that monitors the functionality of an internal device of the robot, like a sensor. If a state contains both continuous and discrete variables, the state space is hybrid. A Markov process has an index set too. In most cases this index represents time, that can be continuous or discrete. Time is continuous as is well known, but in robotics problems it is considered discrete. This is due to the fact that, to update the robot or the map states, some algorithms have to be executed on the onboard robot PC or on a remote PC and they take a little quantity of time. Thus, a dead time is introduced, in which these states cannot be updated. Moreover, any sensor takes its samples at a certain frequency, introducing another forced dead time between an available measurement and the next one. Therefore, in the studied context states will have a discrete time index. The update rate of the states depends on frequency at which sensors work and time needed to run any required algorithm.

To sum up, in the studied robotics problem of mapping an indoor environment with a wheeled robot, two states are needed: robot state and map state. Ideally, these states must be complete, that is information carried by the present state should be sufficient to predict the future next state, with the knowledge of the next control and measurements. Indeed, a complete state for any realistic robot system cannot exist, because this state should carry any type of information inherent to the robot and the environment, like content of its computer memory, position of every not detectable pebble or trash, etc. Due to the huge amount of information real states are irremediably incomplete. However, from a practical point of view, a very small state is adequate for predicting the future in an appropriate manner. Thus, this state can be considered complete. Every snag can be partially or totally overcome by the stochastic environment model. The evaluation of the next state is obtained with stochastic probabilistic laws, so these complete states can be considered Markov chains. Dealt states have a continuous or a hybrid state space and are evaluated in a discrete time.

The last fundamental concept in this introduction is that of *belief*. It arises from the need to distinguish the real state from the internal state of the robot. A robot has its knowledge about itself and its environment, that can be different from the real one. Let us consider a robot placed into a room, after a specific motion the robot sees itself in a specific location, specified in the pose variable. This measurement can be momentary wrong and so it differs from the real one. Pose, such us other state variables, is not directly measurable, but it is obtained from sensor data and previous state through some algorithms. Hence it is easy that the internal state of the robot does not coincide with the real one. Therefore, in a robotic system there is a true state and the belief with regards to that state, the one known by the robot.

Considering a state variable x_t, the belief of this variable is $bel(x_t)$. The belief allocates a probability value to each possible hypothesis regarding the real state. For the computation of the belief, the most common families of algorithm derive from the Bayes filter algorithm. Therefore, first the Bayes filter will be studied and

then, one of the algorithms that arise from this filter will be selected and examined.

3.2 Bayes filters

The Bayes filter is an approach for the estimation of unknown probability density functions, like the random state of dynamic systems. It is a probabilistic tool that works on complete states modelled in discrete time. By definition, such states can be inferred from themselves evaluated at the previous time step, depending on the current measurements. Because of that, this method is recursive and the Bayes filter approach is also named Recursive Bayesian estimation. Time can be continuous, however, as mentioned above, it is always handled like it was discrete, when referring to robot states.

In robotics problems, one seeks to estimate the probability density function, or the belief, of the state based on all available real-time information, which is usually the received sensors data. Thus, an estimation is needed every time the robot gets a set of measurements. In this scenario a recursive filter, and in particular the Bayes filter, is a suitable solution.

A Bayes filter process is divided into two phases, prediction and update. In prediction a *prior* probability density function of the state is computed using the system model, before incorporating the current measurements. Since this model is usually affected by unknown disturbances, the unpredictable noise makes this prior faulty. Then, during the update phase, the latest data coming from sensors are implemented in the system to correct the prior. The result is the achievement of the *posterior* probability density function of the state. In the next time step a new prior will be obtained from this latter posterior, and so on.

To be recursive, the Bayes filter needs to associate a probability density function to the initial state, when measurements have not yet been performed. If the initial state of the system is known, the probability density function is a distribution centred in the correct value with zero probability anywhere else. Instead, if the

system starts from an unknown initial state, a uniform distribution over the domain of the state can be used. These two cases are the most common, but there may be cases in which state is partially known and it is expressed by a not uniform distribution.

Let us consider a practical example of implementation of the Bayes filter algorithm: the robot localization problem. The goal is to obtain the belief of the robot state $bel(x_t)$ using the control and measurement data. The state consists of the robot pose and the initial state is known. The map on which the robot must be localized is given and static. As mentioned above, the filter is divided in two steps. In the prediction step the prior belief distribution $\underline{bel(x_t)}$ is evaluated from the previous posterior belief $bel(x_{t-1})$ and the control data u_t. Then in the update step, from the prior $\underline{bel(x_t)}$ and the current measurements z_t the posterior distribution $bel(x_t)$ is obtained. The algorithm just discussed is shown in the following table.

1	**Bayes_filter_algorithm**($bel(x_{t-1})$, u_t, z_t):
2	for all x_t do
3	$\underline{bel(x_t)} = \int p(x_t \mid u_t, x_{t-1})\, bel(x_{t-1})\, dx$
4	$bel(x_t) = \eta\, p(z_t \mid x_t)\, \underline{bel(x_t)}$
5	endfor
6	return $bel(x_t)$

Table 3.1

In line 2 the control u_t is processed to estimate the variation in pose of the robot from the previous belief of the pose $bel(x_{t-1})$. Specifically, $\underline{bel(x_t)}$ is the belief assigned to the state x_t at time t, computed integrating the product of the belief distribution at time t-1, $bel(x_{t-1})$, and the probability that control u_t leads to transition from x_{t-1} to x_t.

At line 4 the posterior belief $bel(x_t)$ is achieved incorporating the measurement data. The prior belief $\underline{bel(x_t)}$ just accomplished is multiplied by the probability that the measurement z_t have been observed from x_t and by η, a normalization factor. With this latter, an integration of the resulting product is led to 1, like a probability

distribution. bel(x_t) is the result returned by the Bayes filter algorithm, given the posterior belief of the previous time step, bel(x_{t-1}), and the current control and measurement data, u_t and z_t.

Let us introduce a case where the state is binary (x = 0 or x = 1), that is discrete. The objective is again the estimation of the belief of the state, processing current control data and measurements. Firstly, considering the full ignorance of the initial state, an equal probability is assigned to each of the two possible states:

$bel(x_0=0) = 0.5$

$bel(x_0=1) = 0.5$

Furthermore, the control u_t is an action for the interaction with the state. Let us assume that action is aimed at bringing the state to 1. Then, the transition probabilities considering this u_t are:

$p(x_t=1 \mid u_t, x_{t-1}=1) = 1$

$p(x_t=0 \mid u_t, x_{t-1}=1) = 0$

$p(x_t=1 \mid u_t, x_{t-1}=0) = p_{u,high}$

$p(x_t=0 \mid u_t, x_{t-1}=0) = p_{u,low}$

The first two probabilities are valid in most of cases where the action u_t, that brings the state from 0 to 1, is different from the one that brings the state from 1 to 0. Instead, $p_{u,high}$ is usually a high probability value like 0.9, if the actuator works correctly, and $p_{u,low}$ is hence its complementary value to 1 ($p_{u,low} = 0.1$). Indeed, taken a specific previous state ($x_{t-1} = 1$ or $x_{t-1} = 0$) and executing the action u_t, the integration over the state space of the computed probabilities of the state $p(x_t)$

must lead to 1. Working in a discrete state space the integral becomes a finite sum, so the following equations must be verified:

$p(x_t=1 \mid u_t, x_{t-1}=1) + p(x_t=0 \mid u_t, x_{t-1}=1) = 1$

$p(x_t=1 \mid u_t, x_{t-1}=0) + p(x_t=0 \mid u_t, x_{t-1}=0) = 1$

Even the sensors are noisy, so their measurements have a certain uncertainty degree. Given a state value, the sensor can detect it or not. The probability that sensors get a correct measurement added to the probability that they do not get it, must be equal to 1 for each of the two states.

$p(z_t=1 \mid x_t=1) = p_{z1,1}$

$p(z_t=0 \mid x_t=1) = p_{z0,1}$

with: $p_{z1,1} + p_{z0,1} = 1$

and

$p(z_t=0 \mid x_t=0) = p_{z0,0}$

$p(z_t=1 \mid x_t=0) = p_{z1,0}$

with: $p_{z0,0} + p_{z1,0} = 1$

Now it is possible to evaluate recursively the belief of the binary state of the system, by means of the Bayes filter algorithm shown in *Table 3.1*. For example,

suppose to evaluate the belief at time t = 1, with the control u_1 considered until now and receiving a sensor data $z_1 = 1$. The prior belief is obtained by way of the equation in line 3 of the *Table 3.1*, substituting the integral with a sum.

$$\underline{bel(x_1)} = p(x_1 \mid u_1, x_0=0)\, bel(x_0=0) + p(x_1 \mid u_1, x_0=1)\, bel(x_0=1)$$

Introducing numerical values for each probability (considering $p_{u,high} = 0.9$ and $p_{u,low} = 0.1$), the computed believes are:

$$\underline{bel(x_1=1)} = p(x_1=1 \mid u_1, x_0=0)\, bel(x_0=0) + p(x_1=1 \mid u_1, x_0=1)\, bel(x_0=1) =$$

$$= p_{u,high} \cdot 0.5 + 1 \cdot 0.5 =$$

$$= 0.9 \cdot 0.5 + 1 \cdot 0.5 =$$

$$= 0.95$$

$$\underline{bel(x_1=0)} = p(x_1=0 \mid u_1, x_0=0)\, bel(x_0=0) + p(x_1=0 \mid u_1, x_0=1)\, bel(x_0=1)$$

$$= p_{u,low} \cdot 0.5 + 0 \cdot 0.5 =$$

$$= 0.1 \cdot 0.5 + 0 \cdot 0.5 =$$

$$= 0.05$$

Then, the posterior belief is calculated from the prior using the equation in line 4 of the *Table 3.1*.

$$bel(x_1) = \eta\, p(z_1=1 \mid x_1)\, \underline{bel(x_1)}$$

Suppose the probabilities related to the measurement correctness are the following:

$p_{z1,1} = 0.6$ $p_{z0,1} = 0.4$ $p_{z0,0} = 0.7$ $p_{z1,0} = 0.3$

Thus, the posterior believes are obtained:

$bel(x_1=1) = \eta\, p(z_1=1 \mid x_1=1)\, \underline{bel(x_1=1)}$

$= \eta\, p_{z1,1} \cdot 0.95 =$

$= \eta\, 0.6 \cdot 0.95 =$

$= \eta\, 0.57$

$bel(x_1=0) = \eta\, p(z_1=1 \mid x_1=0)\, \underline{bel(x_1=0)}$

$= \eta\, p_{z1,0} \cdot 0.05 =$

$= \eta\, 0.3 \cdot 0.05 =$

$= \eta\, 0.015$

Since the two posteriors must be complementary to 1, the normalization factor can be calculated.

$\eta\, 0.57 + \eta\, 0.015 = 1$

$\eta = (0.57 + 0.015)^{-1} = 1.7094$

Then, the two posteriors are:

$bel(x_1=1) = 0.9744$

$bel(x_1=0) = 0.0256$

As evidenced, the incorporation of sensor data improves the belief precision. This example will help the reader to better understand calculations performed in the next sections, about an algorithm that arises from the Bayes filter, named occupancy grid mapping.

The Bayes filter is the base for the more important existing probabilistic filters. In the following sections parametric filters, like Kalman filter, and non-parametric filters, like grid-based filter, are introduced.

3.3 Gaussian filters

After observing the Bayes filter from a general point of view, considering a generic continuous or discrete state space, it is needed to step a little further into the concepts. The main goal is still the research of a probabilistic approach suitable to build the map of a building.

The probabilistic approaches that arises from the Bayes filter can be assigned to two broad categories: the parametric and the non-parametric filters.

The former are better known as *Gaussian filters* and are implemented working on continuous state spaces. In Gaussian filters the probability density functions of the believes are considered to be Gaussian distributions. They are, then, multivariate normal distribution of the form:

$$p(x) = \det(2\pi\Sigma)^{-\frac{1}{2}} \exp\{-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)\}$$

The distribution over the variable x is characterized by means of two parameters, μ and Σ. μ is the mean vector and has the same dimensionality of the state x. Σ is the covariance matrix, a square, symmetric and positive semi-definite matrix with dimensionality equal to that of the state x squared. Some Gaussian algorithms, like Kalman filter, handle directly mean and covariance, other methods works on other two parameters, that are functions of the first two. Implementing a Gaussian approach, all the estimated believes are Gaussian distributions. In parallel, the measurement model must allow for extracting a measurement probability $p(z_t \mid x_t)$ of Gaussian form. The Gaussian filter has the goal to evaluate the posterior belief of the state and this latter is always a Gaussian distribution, because of its computation. Since a Gaussian can be described by two parameters, the goal of a Gaussian filter turns into the research of these ones. For this reason, these filters are named "parametric". Discussing Gaussian filters, the term belief is set aside and the prior and the posterior refers directly to the state, of which the characterizing parameters are sought.

3.3.1 **Kalman filter**

The best known Gaussian filter is the *Kalman filter*. As mentioned above, it is used to evaluate the value of the mean and the covariance of the posterior, at each time step. Implementing a Kalman filter an additional assumption is needed: all the probability density functions have to be linear functions of its arguments plus a Gaussian noise. This means that probability functions $p(x_t \mid u_t, x_{t-1})$ and $p(z_t \mid x_t)$ of the Bayes filter algorithm become linear.

In the prediction step the prior is computed like:

$$\underline{x_t} = A_t x_{t-1} + B_t u_t + \delta_t \qquad (3.1)$$

$\underline{x_t}$, x_{t-1} and u_t are state and control vectors, each one with its dimension. A_t and B_t are the matrices that linearize the function. Finally, δ_t is the Gaussian noise vector,

which models the randomness in the state transition from x_{t-1} to $\underline{x_t}$. δ_t must have the same dimension of $\underline{x_t}$ and is composed by a mean value equal to zero and a covariance D_t. Therefore, the mean of the prior is achieved by means of the equation (3.1), substituting the mean values of x_{t-1}, the posterior of the previous time step, and of δ_t, equal to zero. The covariance of the prior is derived from the same equation, considering the covariances of x_{t-1} and δ_t (u_t has not a covariance matrix). Then the prior is always a Gaussian distribution with a new mean vector $\underline{\mu_t}$ and a new covariance matrix $\underline{\Sigma_t}$.

In the update step, the measurement probability $p(z_t \mid x_t)$ must also to be a Gaussian function linear in its arguments, with an additional Gaussian noise.

$$z_t = C_t \underline{x_t} + \gamma_t$$

γ_t is the Gaussian noise vector and models the noise of sensor measurement. It has the same dimensionality of z_t. Its mean value is zero and is covariance matrix is Q_t.

The Kalman filter algorithm is shown in *Table 3.2*.

1	**Kalman_filter_algorithm**(μ_{t-1}, Σ_{t-1}, u_t, z_t):
2	$\underline{\mu_t} = A_t \mu_{t-1} + B_t u_t$
3	$\underline{\Sigma_t} = A_t \Sigma_{t-1} A_t^T + D_t$
4	$K_t = \underline{\Sigma_t} C_t^T (C_t \underline{\Sigma_t} C_t^T + Q_t)^{-1}$
5	$\mu_t = \underline{\mu_t} + K_t (z_t - C_t \underline{\mu_t})$
6	$\Sigma_t = (I - K_t C_t) \underline{\Sigma_t}$
7	return μ_t, Σ_t

Table 3.2

The mean $\underline{\mu_t}$ and the covariance $\underline{\Sigma_t}$ of the prior state of the system are computed from the previous posterior and the control u_t, at line 2 and 3. These two

calculations are accomplished once at a time, considering separately the mean and the covariance of the involved arguments. As mentioned above, in line 2 $\underline{\mu_t}$ is calculated from the equation (3.1), substituting μ_{t-1} to x_{t-1} and 0 to δ_t. In parallel, in line 3, $\underline{\Sigma_t}$ is obtained using the covariances of x_{t-1} and δ_t, Σ_{t-1} and D_t respectively.

The mean and the covariance of the posterior are computed in the update step at line 4, 5 and 6 of the Kalman filter algorithm. K_t is a variable called Kalman gain. It estimates the degree of the impact of the new measurements on the current state. The posterior mean μt is calculated at line 5, in which the prior mean $\underline{\mu_t}$ is corrected by means of the deviation of the actual measurement z_t from the mean predicted one ($z_t - C_t\ \underline{\mu_t}$). This deviation is weighted by the Kalman gain K_t. The posterior covariance is obtained at line 6. Σ_t is computed adjusting the prior covariance $\underline{\Sigma_t}$ for the Kalman gain K_t.

In Kalman filter algorithm the initial state x_0 must be represented by a Gaussian distribution. So, the initial state probability is:

$$p(x_0) = \det(2\pi\Sigma_0)^{-\frac{1}{2}} \exp\{-\frac{1}{2}\,(x_0-\mu_0)^T\,\Sigma_0^{-1}\,(x_0-\mu_0)\}$$

with known mean and covariance parameters, μ_0 and Σ_0. This type of initial state is needed to provide that the next state probability is again a Gaussian distribution. This is an important requirement for the implementation of the algorithm. Indeed, the Kalman filter is not suitable for the localization problem of "lost robot" or other problems which imply an unknown initial state.

3.3.2 Information filter

Another probabilistic filter belonging to the family of Gaussian ones is the Information filter. It can be considered the dual of the Kalman filter. It has the same requirements and the same steps of the Kalman. The difference stands in the choice of the describing parameters of the Gaussian distributions. In the

Kalman filter a probability density is parameterized by the mean vector μ and the covariance matrix Σ. The Information filter makes use of an information vector ξ and an information matrix Ω. Both these parameters can be easily evaluated from the Kalman ones. The information vector is equal to:

$$\xi = \Sigma^{-1} \mu$$

The information matrix, also named precision matrix, is equal to:

$$\Omega = \Sigma^{-1}$$

The Information filter algorithm is shown in the following *Table 3.3.*

1	**Information_filter_algorithm**(ξ_{t-1}, t-1, u_t, z_t):
2	$\underline{\Omega_t} = (A_t \Omega_{t-1}^{-1} A_t^T + D_t)^{-1}$
3	$\underline{\xi_t} = \underline{\Omega_t} (A_t \Omega_{t-1}^{-1} \xi_{t-1} + B_t u_t)$
4	$\Omega_t = C_t^T Q_t^{-1} C_t + \underline{\Omega_t}$
5	$\xi_t = C_t^T Q_t^{-1} z_t + \underline{\xi_t}$
6	return ξ_t, Ω_t

Table 3.3

In the Kalman filter the update step is the most time and computing expensive of the two, because of the inversion matrix calculation. Instead, in the Information filter, the most time and computing consuming step becomes the prediction one. For this reason, the Information filter is the dual of the Kalman. At every step it is possible to move from a representation to the other, transforming the parameters.

3.3.3 Extended Kalman filter

In these two filters the assumption of linear probability functions, with added Gaussian noise, is not reflected in most of the practical robotics problems. For example, a robot that follows a circular trajectory cannot have a linear probability state transition $p(x_t \mid u_t, x_{t-1})$. In order to overcome this problem, non-linear versions of these algorithm are introduced: the Extended Kalman filter (also called EKF) and the Extended Information filter (or EIF). Since the two filters are one the dual of the other, only the Extended Kalman filter is taken in consideration here.

The Extended Kalman filter approximates the state distributions by Gaussians and the non-linear state transition and measurements functions by means of linearization. The non-linear functions transform Gaussians in distributions of other shape, interrupting the recursion of Gaussian states. Linearization approximates a non-linear function by a linear one, tangent to the first at a specific point. Working on normal distributions, a logical choice of this tangent point is the mean of the Gaussian implied in the process. As a result, the linearized probability function leads to a new Gaussian distribution, after the state transition or the measurement update. This recursively ensures to achieve a Gaussian posterior state at each time step.

In the Extended Kalman filter the linearization is accomplished by means of a Taylor expansion of the first order. Then, a first order Taylor expansion is the following:

$$f(u_t, x_{t-1}) \approx f(u_t, x_{t-1,0}) + f'(u_t, x_{t-1,0}) (x_{t-1} - x_{t-1,0})$$

with

$$f'(u_t, x_{t-1,0}) = \partial f(u_t, x_{t-1,0}) / \partial x_{t-1,0}$$

As mentioned above, the linearization of the non-linear function is evaluated at the mean of the Gaussian distribution x_t. The Taylor expansion becomes:

$$f(u_t, x_{t-1}) \approx f(u_t, \mu_{t-1}) + f'(u_t, \mu_{t-1}) (x_{t-1} - \mu_{t-1})$$

Let us assume the state transition probability function processing the control information is equal to the non-linear function $g(u_t, x_{t-1})$. Then, the update probability function processing the measurement data is equal to $h(\underline{x_t})$. The first derivatives $g'(u_t, \mu_{t-1})$ and $h'(\underline{\mu_t})$ are shortened in G_t and H_t. The first Taylor expansions are:

$$g(u_t, x_{t-1}) \approx g(u_t, \mu_{t-1}) + G_t (x_{t-1} - \mu_{t-1})$$

and

$$h(\underline{x_t}) \approx h(\underline{\mu_t}) + H_t (\underline{x_t} - \underline{\mu_t})$$

With the linearization, the prior and the posterior can be approximated to Gaussian distributions:

$$p(x_t \mid u_t, x_{t-1}) \approx \det(2\pi D_t)^{-\frac{1}{2}} \exp\{ -\frac{1}{2} [x_t - g(u_t, \mu_{t-1}) - G_t(x_{t-1} - \mu_{t-1})]^T R_t^{-1} [x_t - g(u_t, \mu_{t-1}) - G_t(x_{t-1} - \mu_{t-1})]\}$$

$$p(z_t \mid x_t) \approx \det(2\pi Q_t)^{-\frac{1}{2}} \exp\{ -\frac{1}{2} [z_t - h(\mu_t) - H_t(x_t - \mu_t)]^T Q_t^{-1} [z_t - h(\mu_t) - H_t(x_t - \mu_t)]\}$$

The Extended Kalman filter algorithm is shown in *Table 3.4*.

1	**Extended_Kalman_filter_algorithm**(μ_{t-1}, Σ_{t-1}, u_t, z_t):
2	$\underline{\mu_t} = g(u_t, \mu_{t-1})$
3	$\underline{\Sigma_t} = G_t \Sigma_{t-1} G_t + D_t$
4	$K_t = \underline{\Sigma_t} H_t^T (H_t \underline{\Sigma_t} H_t^T + Q_t)^{-1}$
5	$\mu_t = \underline{\mu_t} + K_t (z_t - h(\underline{\mu_t}))$
6	$\Sigma_t = (I - K_t H_t) \underline{\Sigma_t}$
7	return μ_t, Σ_t

Table 3.4

This algorithm is similar to the Kalman filter one, thus its analysis is avoided.

3.3.4 Mapping with Gaussian filters

The Gaussian filters just seen can be easily deployed for the localization of a robot, given a fixed map. In case the map is not given and/or not fixed, it has to be built and updated using the measurement data, collected by the robot from the environment. The probabilistic filter structure is still the same. Development of a mapping algorithm using a Gaussian filter adds some computations during the measurement update phase. The robot must build the map and, at the same time, localize itself into this map. Then, during the prediction step, the robot pose relating to the map is estimated. Therefore, sensor data are processed. Obstacles already present in the map before the measurements are used to correct the predicted pose. New obstacles are added to the map. Moreover, the map itself is updated. If an obstacle previously seen has been moved, or at the previous step a walking person has been detected, the map must be updated and corrected too.

The map saved into the robot can have an its own state, separated from the robot state, or can belong to a larger system state. This latter includes both the robot state and the map state. Anyway, the estimation state procedure is the same, although the computation is a little different.

As mentioned, the Kalman filter, and therefore also its Extended version, can be used only in continuous state space. Under this condition the map state is usually a vector with a dynamic dimension, that increases every time a new obstacle is detected and added to the map. This vector can also have a large fixed dimension and be initially filled with a value indicating the absence of feature. When a new obstacle is detected from the laser scanner of the robot, it is represented like a point series. Using a suitable algorithm, a feature extraction is accomplished. Now from series of points, the robot has a series of features. At each feature is usually associated three parameters: two for the feature position on the map, x_f and y_f, and one for the signature, s_f. This latter parameter is used to distinguish a feature from another during the update step of the map. The map built with this method is continuous and, if plotted, it is a top-view 2D map. An example of continuous map can be seen in the following *Figure 3.1.*

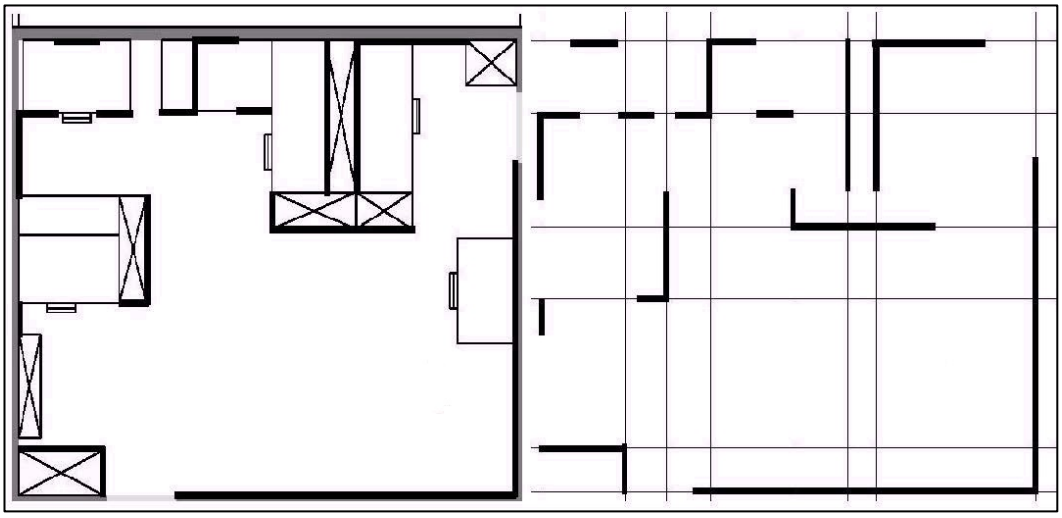


Figure 3.1

The feature extraction algorithm is fundamental for the obstacle detection. There exist algorithms for the extraction of simple geometric primitives, like straight lines and corners, and others for the extraction of polygons. The more elaborated the type of feature is, the more complex and computational consuming the algorithm is.

Gaussian filters have advantages and disadvantages. They are computationally efficient and fast to process. These characteristics are ensured by the state evaluation by means of Gaussian distributions. On the other hand, a normal distribution is unimodal, i.e. it has a single maximum. In localization problems, this translates to having a single area and a single maximum point where the robot could be located. There is not the possibility to track more distinct areas and maximums at the same time. Then, the Extended Kalman filter is not suitable for localization problem in fully or partial given maps with lost initial position. Some extensions of this filter overcome this problem using a mixture of Gaussian densities, like the Multi-Hypothesis Extended Kalman filter. This implies a higher computational complexity and a greater process time consumption. Another limitation is the impossibility of Gaussian filter to work on discrete state space and on hybrid state space. As mentioned above, this group of filters only works on continuous variables, approximating them by normal distributions.

3.4 Non-parametric filters

Non-parametric filters are another important group of filters. They not require a fixed function form for the posterior state distribution or the measurement probability model, like Gaussian filters. Non-parametric filters have a finite number of multiple hypothesis, to each of which corresponds an area in the continuous state space. More areas of the state space are then considered in time. Some non-parametric techniques rely on approximating posterior distributions taking some samples from it, others are based on decomposition of the state space. Therefore, these filters can be used with both continuous, discrete and hybrid state spaces.

The posterior is characterised by a variable number of parameters. Let us reconsider the distinction between real state and belief of that state. Increasing the number of the parameters of a posterior belief, this latter converges to the true state. Updating a higher number of parameters, the computational cost and complexity can easily grow. This price is greater compared to an Extended Kalman

filter. But the number of employed parameters can be adapted to the complexity of the posterior function. Some techniques can adapt this number online and are named *adaptive*. If they adapt the number of parameters based on the computational resources available for the posterior estimation, they are named resource-adaptive.

3.4.1 Particle filters

Two non-parametric filters are discussed in this thesis: the Particle filter and the family of Histogram filters. The Particle filter represents the posterior belief by a set of random points, or samples, taken by the posterior distribution of the state space. This approach has the goal to approximate the posterior with the best degree of reliability, like Gaussian filters do. The difference is that the form of the original distributions has not to fulfil any requirement, so the Particle filter can be applied to many more cases. Moreover, the approximation is more reliable depending on the number of considered samples. This filter can adapt the number of samples, and then the reliability, basing on the computational resource available, so it belongs to the class of resource-adaptive methods.

During each time step, the particle filter collects a number of samples that are named *particles*. The set of samples taken at time t can be denoted as:

$$\chi_t = x_t^{[1]},\ x_t^{[2]},\ x_t^{[3]},\ \ldots\ ,\ x_t^{[Ns]}$$

Ns is the number of samples. It can be fixed or variable, depending on time, computational resource availability or other parameters. Each particle $x_t^{[n]}$ ($1 \leq n \leq Ns$) is a distinct hypothesis about the real state considered at time t. If an area of the state space is densely populated by particles, then the true state has a high likelihood to belong to that area. A mentioned above, the samples are taken randomly, but they are weighted and selected before being included in the particle

set. The probability for a sample to be annexed is proportional to the posterior belief of that state, estimated by means of the Bayes filter algorithm.

$x_t^{[n]} \approx p(x_t^{[n]} \mid x_{t-1}^{[m]}, u_t, z_t)$

The Particle filter is a recursive technique, like the other methods arisen from Bayes filter. The set of particles at time t, χ_t, is evaluated from the previous set χ_{t-1}, depending from the current controls u_t and measurements z_t. m identifies the corresponding particle of the set χ_{t-1} from which the new particle $x_t^{[n]}$ is evaluated. In general, $1 \leq m \leq Ms$, with $Ms \neq Ns$. The basic variant of the Particle filter considers a fixed number of particles, thus a fixed value of Ns in time. This basic variant is shown in *Table 3.5*.

1	**Particle_filter_algorithm**(χ_{t-1}, u_t, z_t):
2	$\underline{\chi_t} = \chi_t = \varnothing$
3	for m = 1 to M do
4	sample $x_t^{[m]} \approx p(x_t \mid ut, x_{t-1}^{[m]})$
5	$w_t^{[m]} = p(z_t \mid x_t^{[m]})$
6	$\underline{\chi_t} = \underline{\chi_t} + \langle x_t^{[m]}, w_t^{[m]} \rangle$
7	endfor
8	for m = 1 to M do
9	draw i with probability $\propto w_t^{[i]}$
10	add $x_t^{[i]}$ to X_t
11	endfor
12	return χ_t

Table 3.5

In line 2, the particle set χ_t and a temporary particle set $\underline{\chi_t}$ are initialized to be empty.

From line 3 to 7, the temporary set $\underline{\chi_t}$ is filled. At line 4, each particle $x_{t-1}^{[n]}$ is updated processing the control information u_t. This process involves the

resolution of the probability $p(x_t^{[n]} \mid u_t, x_{t-1}^{[n]})$, not always easy to implement to certain probability distribution. However, from this equation the Ns particles obtained represents the prior belief $\underline{bel(x_t)}$. The measurement data are incorporated at line 5, where they are processed to achieve the weighting parameters of the Ns particles. These parameters are also named importance factors $w_t^{[n]}$. Importance factors are computed using the probability function of the measurement update step of the Bayes filter, $p(z_t \mid x_t^{[n]})$. Multiplying the importance factor $w_t^{[n]}$ and the corresponding particle $x_t^{[n]}$ obtained at the previous line, a weighted particle is achieved, that represents the posterior belief $bel(x_t)$ of that particle. At line 6 each particle $x_t^{[n]}$ with the corresponding importance factor $w_t^{[n]}$ is added to the momentary particle set $\underline{\chi_t}$.

From line 8 to 11, the final particle set χ_t is achieved. After computing the various particles $x_t^{[n]}$, they must be selected to build the new particle set. This procedure is named *resampling*. This term comes from the complete replacement of the previous particles into the particle set. Indeed, at line 2, the particle set χ_t is cleaned out and, in this step, it is refilled. The particles selecting method is based on relevance given from the weighting factors. The particles are drawn randomly from the temporary set $\underline{\chi_t}$ and, when extracted, they are inserted into the particle set χ_t. However, each particle in $\underline{\chi_t}$ is accompanied by its own importance factor, that weights the particle likelihood to be randomly drawn. The new particle set χ_t will be formed by many duplicates of those samples with a high importance weight. Because of this, the particles with a low value of the information factor will be easily discarded and lost.

After the first for cycle (lines 3 to 7), the distribution of particles $x_t^{[n]}$ present in the momentary particle set $\underline{\chi_t}$ can be associated to the prior belief $\underline{bel(x_t)}$ of a Bayes filter. After the resampling step (lines 8 to 11), the distribution of particles $x_t^{[n]}$ in the final particle set χ_t can be associated to the posterior belief $bel(x_t)$. Indeed, this latter distribution comes from a random selection weighted by an importance factor. This factor is computed by means of the probability $p(z_t \mid x_t^{[n]})$. Therefore, the particle distribution returned from the Particle filter algorithm can be associated to the Bayes posterior belief distribution $bel(x_t) = \eta\, p(z_t \mid x_t).\, \underline{bel(x_t)}$.

There exist many variations of the proposed Particle filter. The difference from one filter to the other usually stands in the resampling step. The above explained resampling method is named Importance sampling. Another version of the algorithm never resamples and update the importance factor multiplying it recursively, with an initial value equal to 1. Other algorithms differ in resampling frequency. A too frequent resampling can lead to a loss of diversity in the particle set. A too infrequent resampling can lead to a loss of samples in low probability areas. Then, various Particle filters play around this problem and develop different sampling algorithms.

There are two other main problems connected with the Particle filter. The former stays in the generation of particles. Particles are initially produced by the previous particles of the set χ_{t-1} and the control data u_t. Measurements are not included in this procedure. So, using a robot with an inaccurate motion and very accurate sensors leads to the generation of a set of particles which cannot match with the measures of the following step. This leads to an inefficient filter. The second problem arises using a high-dimensional state space. Increasing the dimension of the state space, the likelihood of not finding a sample near the true state increases. Thus, in these cases it is necessary to consider a larger number of samples. The same problem can be translated in a lower-dimensional state space with a too little number of samples. If the available samples are not enough a solution is to force the introduction of random ones drawn by a second algorithm. At this point the request is to find a suitable number of samples to approximate the posterior distribution. This number depends on the dimensionality of the state space and the degree of uncertainty of the filter distributions.

3.4.2 Histogram filters

Another type of non-parametric filters are the Histogram filters. Histogram filters works on discrete state spaces with a finite number of possible state values. In order to process a continuous state space, the filter must decompose it in a discrete space. A discretized space has many regions, each one characterized by a single cumulative probability value. The posterior is represented by a histogram.

Every region is a constant bar of the histogram, characterized by its value of probability regarding the posterior belief.

In discrete spaces with a finite number of possible state values, the posterior belief is a discrete probability distribution, that allocates a probability value to all the possible states. This distribution is given by the Bayes filter algorithm used for discrete cases. This algorithm has been described at the section 3.2 and is reported in the following *Table 3.6*.

1	**Discrete_Bayes_filter_algorithm**(bel(x_{t-1}), u_t, z_t):
2	for all n do
3	$\underline{bel(x_t=x_n)} = \sum_i p(x_t=x_n \mid u_t, x_{t-1}=x_i)\ bel(x_{t-1}=x_i)$
4	$bel(x_t=x_n) = \eta\ p(z_t \mid x_t=x_n)\ \underline{bel(x_t=x_n)}$
5	endfor
6	return bel(x_t)

Table 3.6

The control u_t and the measurements z_t, performed at each time step, recursively update the posterior belief bel(x_t) and, therefore, recursively update the probability associated to every state of the state space. If control and measurements do not involve a state, its probability value does not change.

Continuous state spaces must be decomposed. This decomposition, or discretization, translates a continuous space in a continuous series of regions of that space. These regions can be denoted as $x_{1,t}$, $x_{2,t}$, $x_{3,t}$, … , $x_{N,t}$. Each region is convex and does not intersect another region. If the state space is not finite, it has to be truncated. As mentioned above, the Histogram filter works on discrete finite spaces, i.e. with a finite number of possible states.

Regions of a discretized state space are usually cells and the whole space appears as a grid. For this reason, this filter is also called grid-based. The cells dimension is named *granularity* of the grid, or of the decomposition. The granularity distinguishes the quality of a Histogram filter from another one. A high granularity,

hence a small cell dimension, provides a great accuracy of the filter, but increases the computational cost. Then the cell dimension is a variable parameter, basing on requirements and processing limitations. In high-dimensional or considerably extensive state spaces, a larger number of cells is needed, at the same granularity. But increasing too much the number of cells means the achievement of an unsustainable computational cost. Otherwise, the dimension of the cells can be increased, but in too wide state spaces, this option leads to an inaccurate posterior distribution. Therefore, a Histogram filter has a limitation on the state space dimension, given from the available computational resources and available memory.

The decomposition of the continuous state space can be *fixed* or *adaptive*. A fixed cell decomposition builds a grid of cells with a predefined dimension. It is easy to implement but requires an advance rough idea of the granularity needed. Moreover, a waste of computational resources may occur for the process of a high number of cells in low probability region. On the other hand, a more inaccurate posterior approximation may be accomplished in regions with a concentrated high probability. An adaptive cell decomposition uses a generally bigger size of cells and develops a further decomposition of the area in regions with a higher probability value. The decomposition is recursively pursued, proportionally to the probability value, until a certain resolution. This process must be accomplished online in real time and involves an additional computational cost. An example of fixed and adaptive decomposition is shown in the following *Figure 3.2*.

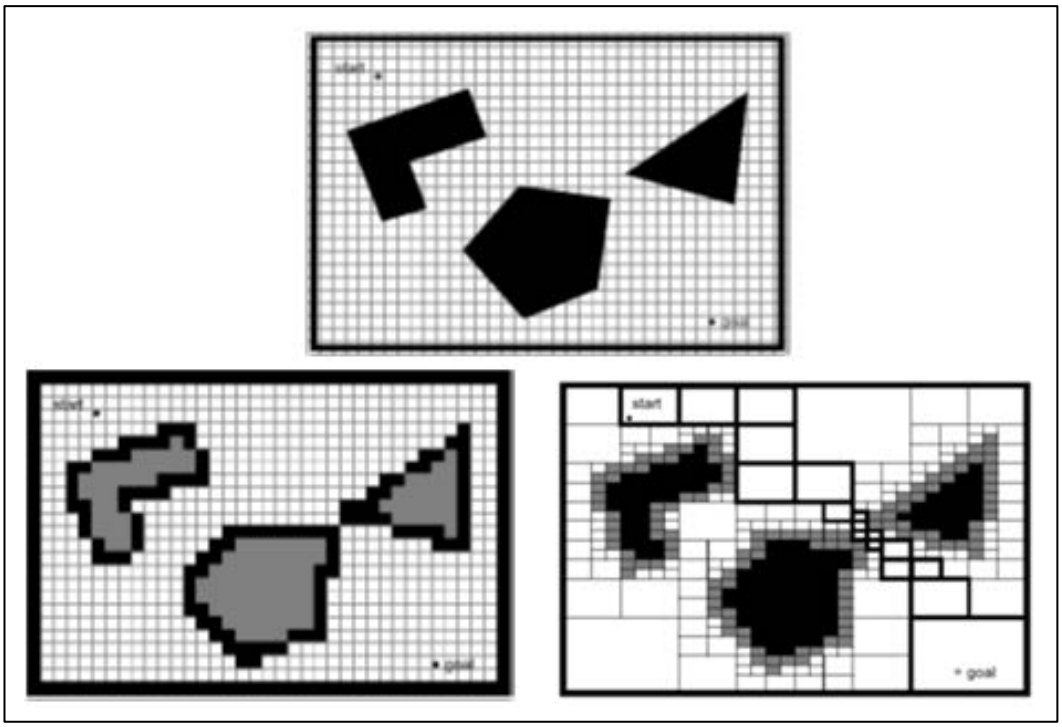


Figure 3.2

The probability value given to a cell is the cumulative probability value of the whole cell area. The resulting probability assigned to a cell is uniform and corresponds to a bar of the posterior belief histogram. Considering N cells, $x_{1,t}$, $x_{2,t}$, $x_{3,t}$, … , $x_{N,t}$, this uniform probability can be obtained by means of the following equation:

$$p(x_t) = \frac{p_{n,t}}{|x_{n,t}|}$$

where $p_{n,t}$ is the probability of the n-cell ($1 \leq n \leq N$) and $|x_{n,t}|$ is the volume of that region. The probability value is usually associated to the centre of the cell. If the cell is small enough, this association is better defined and the posterior distribution acquires a better accuracy.

3.4.3 Mapping with Histogram filters

Mapping with a Histogram filter presents some advantages and some limitations. The state space usually represents the real environment. This is a continuous space, then it is decomposed in a grid. Each cell of the grid represents a region of the environment. Each cell is associated to a binary variable, that specify its state. The cell can be free, thus navigable, or occupied by an obstacle, that the robot must avoid. A cell cannot be half occupied and half free. When this latter case arises, the algorithm must consider the cell in one of the two possibilities. The greater the granularity of the grid, the more this problem becomes irrelevant. Because of the method used, when the Histogram filter is used for the mapping purpose, the filter technique is named *Occupancy Grid Mapping*. Essentially, the map state space is the robot representation of the environment to map. This continuous state space is decomposed in a finite grid. Each cell of the grid is represented by its middle point and this point is a binary map state variable. The value of each map state variable is usually represented in the Boolean notation: 0 stands for free cell, 1 for occupied cell. The posterior belief of the map state is always a histogram. Each bar of the histogram is associated to the middle point of a cell, thus to a binary map state variable, and represents the belief of the robot regard the state of that cell, by means of a probability value.

It is easy to implement respect to a Particle filter and avoids the sampling problems. As mentioned before about Histogram filters, wide areas, then wide state spaces, are hard to process, because of the high storage capacity required. Also a Particle filter may present some problems mapping large environments. A high number of parameters and samples are required to achieve an accurate posterior belief. In the outdoor case a Gaussian filter works better. A Gaussian filter builds a map recognising new obstacles, translating them in features and finally adding them to the map state. Free spaces are not inserted in the map state, so they do not fill the memory. This means that sparse environments are easier to map. On the contrary, areas full of obstacles are hard to map with a Gaussian filter. The filter must be very accurate to distinguish different objects and associate the already-seen ones. This involves a higher computational effort. Problems with a Gaussian filter arise more easily in indoor environments, that are usually not

sparse. Instead, the Occupancy Grid Mapping handles every cell of the grid in the same way, verifying if it is occupied or not. This filter is not affected by the density of objects found, because it only updates the probability value associated to determined cells. Therefore, the Occupancy Grid Mapping is limited by the extension of the environment, Gaussian filters by the quantity of faced obstacles.

4 Occupancy Grid Mapping development

The mapping algorithm is developed using the Python language. In the first section the probabilistic filter used for the mapping purpose is selected. Thus, it is described in detail. In the second section, the inputs needed to the selected algorithm and the obtained outputs are analyzed. In the third section the achieved code is explained, one part at a time.

4.1 Choice of the mapping filter

The mapping technique used in this thesis work is the Occupancy Grid Mapping. It is easy to implement and has a lower processing effort. It does not rely on a feature detector, like a Gaussian filter, that involves an additional computational cost. Moreover, it is hard to develop a robust algorithm for a correct feature detection. A laser range sensor detects a series of points. These points are not usually in their real position but are located in a cloud around it. In case of straight-line extraction algorithms, the lower accuracy is achieved during the computation of the direction of a line. If the algorithm is not robust, a straight wall may be represented by a series of straight broken lines with different direction. A mapping algorithm that avoids the feature detection, does not deal with this problem and is easier to develop. Instead, the Occupancy Grid Mapping algorithm has the disadvantage of a larger storage capacity requirement, but this problem becomes significant only when the environment to map is huge or the dimension of the cells is too small. The goal of the thesis is the development of a mapping algorithm for indoor environments. Consequently, the problem of storage requirement will not be faced, as shown later on.

After having selected a mapping filter, a feasible type of map must be chosen. Working with an Occupancy Grid Mapping method, the map must be discrete and finite. The environment is then discretized and truncated. The map state is represented by a grid of square cells. The dimension of the grid can be fixed or can increase during the robot navigation. In this latter case, the map has a starting little dimension and when the robot moves, new cells are added to the grid, i.e. rows and columns are added to the map state. Also the dimension of the cells can be fixed or dynamically change in time. Indeed, the decomposition of the environment can be fixed or adaptive, as mentioned in the previous chapter. A mapping algorithm usually works along with other algorithms that perform other tasks. A grid with fixed size and fixed resolution is a choice easily feasible with these other algorithms. This is the case experienced in the lab at Politecnico di Torino, where the developed Occupancy Grid Mapping has to be run with an algorithm for the autonomous navigation of the robot. In this case, the navigation algorithm expects a map matrix of a given size, where the resolution is known a priori. Grid size and cells dimension are then set before running the mapping algorithm.

In summary, the mapping algorithm that will be developed is an Occupancy Grid Mapping. The discretized environment is a grid with predefined size. Cells of the grid have the same dimension and fixed resolution. Each cell is represented by its middle point and is associated to a binary state value. This value is 1 if the cell is occupied and 0 if it is free. The belief of the robot about every cell value is a matrix of probability values stored in the state belief $bel(x_t)$. This belief is computed by means of the Discrete Bayes filter algorithm, stated in *Table 3.6* of section 3.4.2. The algorithm has the two steps characterising every Bayes filter, prediction and update. These steps are executed for all the cells, recursively at each considered time.

In the prediction step the control data u_t are processed to estimate the prior belief $\underline{bel(x_t)}$ of the system. Since the map is not influenced by the movement of the robot, only the pose belief is involved in this step. The map belief in the prior is still the same of the previous posterior. In the update step, measurements z_t are

used to correct the belief of pose and update the belief of map. A single system state can be considered, formed by the pose vector and the map matrix. Alternatively, the robot state, with its pose, and the map state, with the environment representation, can be considered separately. Let us denote the robot state at time t x_t and the map state m_t. Their believes are respectively bel(x_t) and bel(m_t).

Let us consider the recursively estimation of the robot belief bel(x_t). The robot state consists of its pose related to the constructed map. Thus, the robot state space is continuous. In this thesis work the robot state is not represented by a probability function, then it is not computed by means of probabilistic filter. Instead, the belief about the robot pose is given with a probability of 100%, related to a single point of the map. The localization of the robot is carried out recursively processing the control data u_t. At each time step the information u_t provides a displacement measure with respect to the previous robot pose. This measure is implemented to the previous pose to achieve the new one, by means of a recursively addition. The updating process of the robot state uses a motion model to compute the belief of the pose. A motion model can be odometry based or velocity based. Robots equipped with wheel encoders implement the odometry based model. Robots without encoders use the velocity model. TurtleBot3 used in this thesis is provided with an encoder for each wheel. So, the control data u_t processed are the measurements coming from the encoders. These measurements are better examined later on. The absence of a probabilistic computation for the robot state makes the process cost lighter, keeping the robot state and the map state updated even in conjunction with other algorithms.

Let us consider now the estimation of the map belief bel(m_t). Given the discrete Bayes filter algorithm in TAB 2.5, a form readapted for the implementation of the Occupancy Grid Mapping is presented below:

1	**Discrete_Bayes_filter_algorithm_for_OGM**(bel(m_{t-1}), u_t, z_t):
2	for all n do
3	$\underline{bel(m_{n,t})} = \sum_i p(m_{n,t}=1 \mid u_t, m_{n,t-1}=m_i)\ bel(m_{n,t-1}=m_i)$
4	$bel(m_{n,t}) = \eta\ p(z_t \mid m_{n,t}=1)\ \underline{bel(m_{n,t}=1)}$
5	endfor
6	return bel(m_t)

Table 4.1

As can be seen, the posterior belief obtained from this filter indicates the probability of the N cells of the grid being occupied by an obstacle or not. Then, the notation bel(m_t) stands for bel(m_t=1). Since the state is binary, the bel(m_t=0) can be calculated by a straightforward subtraction:

$$bel(m_t=0) = 1 - bel(m_t=1) \tag{4.1}$$

Since the map state is a matrix, its belief is also a matrix. So, in equation (4.1), the "1" is a full-1 matrix of the same dimension of bel(m_t).

Therefore, considering the cell n, with 1 < n < N, its probability of occupation is evaluated, i.e. the probability that the state of the cell n is equal to 1. This probability value ranges between 0 and 1. It will be 0 when the cell is occupied with a probability of 0%, i.e. free with a 100% probability. Instead, the probability value will be 1 if the cell is occupied with a 100% probability. In practice, the probability values will never be precisely 0 or 1. This is due to the nature of the recursive probabilistic calculation and to the initial state of the map. Indeed, at time t_0 = 0, when no measurement has yet been received from the sensors, the maximum uncertainty value $p(m_{n,t0} = 0.5)$ is assigned to each cell of the map.

During the prediction step the control data u_t, that is the measurements of the encoders, are processed. As mentioned above, they do not directly affect the map belief. Indeed, the data u_t are used exclusively for the calculation of the robot pose. Therefore, it can be said that the current prior belief calculated in line 3 is equal to the previous posterior belief:

$$\underline{bel(m_{n,t})} = \sum_i p(m_{n,t}=1 \mid u_t, m_{n,t-1}=m_i)\, bel(m_{n,t-1}=m_i) =$$
$$= \sum_i p(m_{n,t}=1 \mid m_{n,t-1}=m_i)\, bel(m_{n,t-1}=m_i) =$$
$$= bel(m_{n,t-1}=m_i) \qquad (4.2)$$

The notation m_i represents one of the two possible values of the binary state: 1 or 0. So, the sum over the index i computes the probability that $m_{n,t}$ is 1 processing the control u_t, considering the two possible starting cases $m_{n,t-1} = 0$ and $m_{n,t-1} = 1$. When the control does not influence this probability, the previous belief cannot change and the prior is equal to the previous posterior (equation (4.2)).

The update step is carried out in line 4 of the *Table 4.1*:

$$bel(m_{n,t}) = \eta\, p(z_t \mid m_{n,t}=1)\, \underline{bel(m_{n,t}=1)} \qquad (4.3)$$

The standard approach uses this equation to update the map state depending from the measurement data z_t. The output is a probabilistic value from 0 to 1, which must be truncated at a certain number of digits after the decimal point. There exists another method to represent the occupancy likelihood of a cell. This method arises from the same root of equation (4.3). The equation (4.3) derives from the probability that a state takes place at time t, giving the whole set of measurements $z_{1:t}$, gathered from time t = 1 to the current time t. A generic entry of the map state matrix, $m_{n,t}$, and its likelihood to be equal to 1 are taken in consideration. This probability can be rewritten like:

$$bel(m_{n,t}) = p(m_{n,t}=1 \mid z_{1:t})$$

The Bayes rule and the assumption of complete state can be implemented to this equation.

$$p(m_{n,t}=1 \mid z_{1:t}) = \frac{p(z_t \mid m_{n,t}=1, z_{1:t-1})\, p(m_{n,t}=1 \mid z_{1:t-1})}{p(z_t \mid z_{1:t-1})} \quad \text{Bayes rule on } m_{n,t}$$

$$= \frac{p(z_t \mid m_{n,t}=1)\, p(m_{n,t}=1 \mid z_{1:t-1})}{p(z_t \mid z_{1:t-1})} \quad \text{complete state}$$

$$= \frac{p(m_{n,t}=1 \mid z_t)\, p(z_t)\, p(m_{n,t}=1 \mid z_{1:t-1})}{p(m_{n,t}=1)\, p(z_t \mid z_{1:t-1})} \quad \text{Bayes rule on } z_t$$

The same procedure is accomplished for the opposite event $m_{n,t} = 0$.

$$p(m_{n,t}=0 \mid z_{1:t}) = \frac{p(m_{n,t}=0 \mid z_t)\, p(z_t)\, p(m_{n,t}=0 \mid z_{1:t-1})}{p(m_{n,t}=0)\, p(z_t \mid z_{1:t-1})}$$

The ratio of both probabilities is computed:

$$\frac{p(m_{n,t}=1 \mid z_{1:t})}{p(m_{n,t}=0 \mid z_{1:t})} = \frac{p(m_{n,t}=1 \mid z_t)\, p(m_{n,t}=1 \mid z_{1:t-1})\, p(m_{n,t}=0)}{p(m_{n,t}=0 \mid z_t)\, p(m_{n,t}=0 \mid z_{1:t-1})\, p(m_{n,t}=1)}$$

Probabilities $p(m_{n,t}=1 \mid z_{1:t})$ and $p(m_{n,t}=0 \mid z_{1:t})$ are complementary to 1. Then $p(m_{n,t}=0 \mid z_{1:t})$ can be written like 1 - $p(m_{n,t}=1 \mid z_{1:t})$. The ratio becomes:

$$\frac{p(m_{n,t}=1 \mid z_{1:t})}{p(m_{n,t}=0 \mid z_{1:t})} = \frac{p(m_{n,t}=1 \mid z_t)}{p(m_{n,t}=0 \mid z_t)} \; \frac{p(m_{n,t}=1 \mid z_{1:t-1})}{p(m_{n,t}=0 \mid z_{1:t-1})} \; \frac{p(m_{n,t}=0)}{p(m_{n,t}=1)} \qquad (4.4)$$

Every term of this equation is divided for its complementary. The logarithm of the ratio of probabilities assigned to opposite binary events is named *log odds ratio*. The log odds ratio of a generic state x is the l(x) function:

$$l(x) = \log\left(\frac{p(x)}{1 - p(x)}\right)$$

The posterior belief of the map state computed for the generic cell n, bel($m_{n,t}$), can be represented in log odds form:

$$l(m_{n,t}) = \log(\frac{bel(m_{n,t})}{1 - bel(m_{n,t})})$$

The log odds ratio of the posterior map belief can be evaluated from the equation (4.4), introducing a logarithm. Therefore, the multiplication of ratios becomes a sum of logarithms of those ratios.

$$\begin{aligned} l(m_{n,t} = 1 \mid z_{1:t}) &= \log\left(\frac{bel(m_{n,t} = 1 \mid z_{1:t})}{1 - bel(m_{n,t} = 1 \mid z_{1:t})}\right) \\ &= \log\left(\frac{p(m_{n,t} = 1 \mid z_t)}{1 - p(m_{n,t} = 1 \mid z_t)}\right) + \log\left(\frac{p(m_{n,t} = 1 \mid z_{1:t-1})}{1 - p(m_{n,t} = 1 \mid z_{1:t-1})}\right) \\ &+ \log\left(\frac{1 - p(m_{n,t} = 1)}{p(m_{n,t} = 1)}\right) \\ &= l(m_{n,t} = 1 \mid z_t) + l(m_{n,t} = 1 \mid z_{1:t-1}) - l(m_{n,t} = 1) \end{aligned}$$

A shorter form of this equation is introduced substituting $l(m_{n,t} \mid z_{1:t})$ with $l_{n,t}$. As in the case of the belief bel($m_{n,t}$), the notation $l(m_{n,t})$ implies $m_{n,t} = 1$.

$$l_{n,t} = l(m_{n,t} \mid z_t) + l_{n,t-1} - l_0 \qquad (4.5)$$

The first log odds ratio $l_{n,t}$ represents the posterior belief about the occupancy of the cell n at time t ($m_{n,t} = 1$). It is computed by means of the sum of three logarithms. $l(m_{n,t} \mid z_t)$ is the inverse measurement model. An inverse measurement function $p(m_t \mid z_t)$ is preferable to the forward measurement function $p(z_t \mid m_t)$, since the first is easier to implement. The log odds ratio $l_{n,t-1}$ represents the previous posterior belief of the n cell. The log odds ratio l_0

computes the belief about the occupancy of the cell without any past or present knowledge. This is the starting scenario, before any control and measurement. When the robot must map a completely unknown environment, each cell belief is initialized with 0.5, the maximum uncertainty value. This is the most common situation. In this case the log odds ratio l_0 is the same for every cell of the grid at each time step and is computed as follows.

$$l_0 = \log\left(\frac{p(m_{n,t=0})}{1 - p(m_{n,t=0})}\right) = \log\left(\frac{0,5}{1 - 0,5}\right) = \log(1) = 0$$

Then, using a log odds form of belief in a starting fully unknown environment, all the elements of the map belief matrix are initialized with 0.5. These elements represent the log odds believes of distinct cells of the map. At each time step, each element is updated by means of the equation (4.5), along with the measurement data z_t. This corresponds to the update step of a discrete Bayes filter. Being $l_0 = 0$, the equation can be rewritten:

$$l_{n,t} = l_{n,t-1} + l(m_{n,t} \mid z_t) \qquad (4.6)$$

In log odds form, the update is accomplished by a simple sum. Moreover, each element of the map belief matrix assumes a value from -∞ to +∞. This representation introduces an important advantage. It avoids problems of truncation, especially for probability values closer to 0 or 1 (values that cannot be reached). The higher above 0 the probability value, the higher the belief that the cell is occupied. The lower below 0 the probability value, the higher the belief that the cell is free. A threshold must be set. If the cell belief is greater than this positive threshold, the cell is considered occupied with enough certainty. If the cell belief is smaller than the negative threshold, the cell is supposed to be free. When the belief of the cell is a value that ranges between the negative threshold and the

positive threshold, the cell state is uncertain. Therefore, the belief values of these cells cannot assign a state with enough confidence.

4.2 Input/ Output analysis

The Occupancy Grid Mapping needs some inputs and returns other outputs. Inputs are given from the TurtleBot3 sensors, over the ROS framework. Let us go into detail about this topic. The mapping algorithm needs two measurements to work, odometry and laser scan. Odometry specifies position and orientation of the robot with respect to a coordinate frame. This latter is positioned at the starting point of the robot. Odometry is updated recursively at each measurement step, adding the shift in position or the change in orientation detected from the encoders. Encoders are incremental sensors that detect the movement by means of the wheel rotation. The TurtleBot3 has two encoders, one for each wheel. Raw data coming from encoders are elaborated from the robot onboard computer, to achieve the odometry measure. Odometry is then posted on a ROS topic. From this topic data can be read from the remote PC, where the mapping algorithm runs. ROS network is needed to share the measurements from the robot PC and the remote PC, through the Wi-Fi.

Laser scan is a set of 360 measurements taken from the Lidar. Working at a certain frequency, the 360° laser sensor mounted on TurtleBot3 gathers a set of range measurements around it. The 360° Lidar is a rotating laser. Rotation velocity defines the frequency of measurements achievement. During a single round, the sensor sends a laser ray at each degree. Based on returning time of each ray, distance from an obstacle is computed for that degree. This means that at each time step Lidar obtains a set of 360 measurements, each one shifted from the previous one by 1°. Even laser scans are published by the TurtleBot3 on a ROS topic. Sent data are organized in an order vector of 360 elements. The "0" element corresponds to the distance detected at degree 0, the "1" element to the detection at degree 1 and so on, until the "359" element.

All data shared via ROS by the TurtleBot3 can be acquired from a device connected to ROS framework. The list of active topics, where data are published, can be carried out running on a terminal window the command:

$ rostopic list

Here all devices connected to ROS can publish on different topics. Relevant topics for the thesis purpose are */odom* and */scan*. To observe the structure of published data, the command "rostopic echo /odom" must be launched from the terminal. The topic /odom is considered. Odometry data represent the pose of the robot. They are divided into position and orientation. Position is a 3-dimensional variable, therefore it is divided in its three components x,y and z. Orientation is represented in the quaternion form, then it is composed by four components x,y,x and w. In the /odom topic, position and orientation are collected in the variable vectors *pose.pose.position* and *pose.pose.orientation*, respectively.

Laser scan is a vector with dimension equal to 360. Command "rostopic echo /scan" is used to see how the vector of laser measurements is published. Laser scan data are published under the variable vector *ranges*.

During a mapping process execution, the developed program must enter the relevant topics and read published data. To accomplish this, the knowledge of the exact data structure of the ROS topics is needed. The submitted method can be used for discovering this structure.

Odometry and laser scan are necessary to update the robot state and the map state. Odometry is used to localize the robot. It is published on topic /odom at a sampling rate of about 30 Hz. Thus, robot can localize itself into the created map with that frequency. Laser scan is used to detect objects around the robot. Laser data are collected into a vector and published on topic /scan with a frequency of about 5 Hz. This sampling rate is limited by the sensor rotational speed. Laser scans alone cannot build a map. They are series of points taken around the robot, but

the robot must be localized. Then odometry is used to identify the robot pose in a map, previously built and initialized with the maximum uncertainty degree. At start up, a system reference frame is created, to which robot pose and map refer. When the robot moves, odometry tracks its position in the map and selects which cells of the map are involved in the update step of the Occupancy Grid Mapping algorithm. Then laser data set is used to update the belief of the selected cells.

The working frequency of sensors is very important and must be analyzed. Odometry is published with a frequency of 30 Hz, laser data with a frequency of 5 Hz. This means that whenever a laser scan is gathered, robot pose has been already updated a few times. The limiting sensor of the system is the laser scanner. When programs are developed to run on this system, their working frequency must be evaluated in comparison with the Lidar sampling rate. Scanning at 5 Hz implies that obstacles are identified every 0.2 seconds. Then, to not waste a set of laser data, every external cycle process must terminate in 0.2 seconds. A mapping algorithm with an updating cycle frequency lower than 5 Hz loses some sets of laser scans. For example, the case in which a mapping algorithm processes a data set in 0.3 seconds is examined. At time t = 0 a laser data set is published and the mapping algorithm is executed. At time t = 0.2 another laser scan is ready, but the mapping algorithm is busy. At time t = 0.3 the algorithm becomes free, but cannot process data set published 0.1 seconds earlier, because the current pose of the robot does not coincide with the previous one. Updating the map with these data translates the occupation of some cells from their real position to a faulty one, based on the movement of the robot in the last 0.1 seconds. Then the robot has to wait 0.1 seconds for the next laser scan. A slow mapping process inevitably implies a loss of some laser data sets.

The output of the developed Occupancy Grid Mapping is the map belief matrix. This matrix has a pre-set dimension, i.e. a fixed number of elements. Each element represents the occupancy belief of a cell of the map. The position of the elements in the matrix corresponds to the position of the cells in the map. As mentioned above, the log odds form is used to state the map belief. Commonly the environment to map is unknown, so at start up all the elements of the map belief

have a 0 value. During robot recon, cell believes are updated and can assume a value from -∞ to +∞.

When the developed mapping program is asked to show the built map, the map matrix is analysed and plotted. The created image looks like an ordered set of coloured points, representing each element of the matrix. Each point is located based on the position of the element in the matrix and is associated to the centre of a cell of the grid map. The color of each point is assigned with respect to the associated belief value of that cell. There are three different colours. Black describes a cell occupied by an obstacle. White is assigned to a free cell. Finally, blue is associated to unknown cells. Two thresholds are used to establish probability values over which a cell changes its believed state, and then its colour. Whenever the user wants, he can ask the algorithm to process data relative to the map and plot it. During the plot process the robot can continue to navigate and update its map. The printed map will contain all the updates accomplished before the plotting request.

4.3 The developed algorithm

The thesis purpose is the development of an algorithm for the mapping of an indoor environment. Localization is intrinsically connected to the mapping goal, then it is carried out, but not improved. The Occupancy Grid Mapping is the type of algorithm chosen to develop. The log odds ratio is the form of representation for the belief of map state. Odometry and laser scan data are the inputs of the algorithm. The map belief matrix and its plot are the outputs. Carried out these starting decisions and considerations, the mapping algorithm can be developed. The first version of the program is analyzed in this section. In the next one, performed improvements and observed problems about the algorithm are examined.

Mapping and localization algorithm is developed in a script written in python programming language. Let us have a more detailed look at the program. The full

python script can be found in the Appendix B, but analyzed parts are reported here.

4.3.1 Imports

At the start-up all the necessary imports are performed.

```
2       #Imports:
3       import time
4       import threading
5       import math
6       import numpy
7       from tf.transformations import euler_from_quaternion
8       import matplotlib.pyplot as plt
9
10      import rospy
11      from sensor_msgs.msg import LaserScan
12      from nav_msgs.msg import Odometry
13      from geometry_msgs.msg import Twist
14      from std_msgs.msg import String
```

Table 4.2

The first block imports all the needed structural and mathematical modules and functions. *time* is a common module in python with useful functions like *time.sleep(x)*, that makes the current process wait for x seconds. *threading* module is used to run more processes simultaneously within the same program. It is employed when a part of the program must wait for any kind of input, while another part must be processed. *math* module makes available many mathematical functions defined by the C standard. Thus, algebraic functions, elementary transcendental functions, like exponential, trigonometric, logarithmic and hyperbolic ones, function for the conversion between radians and degrees, and so on. *numpy* module introduces useful functions for matrices initialization and handling. From the *tf.transformations* module, the *euler_from_quaternion* function is imported. Since the input orientation measurement is taken from

/odom topic in quaternion form, it must be converted into Euler representation. Euler angles are straightforward to be processed by the Occupancy Grid Mapping program. *matplotlib.pyplot* is the module that operates on mathematical plots. It is imported as *plt*, then it is invoked typing "plt".

Imports in second block are required to introduce python functions related to the ROS environment. *rospy* module is needed to handle operations on ROS topics from a python script. Topics have some data posted, by means of published ROS messages. When the node created by the program wants to publish or read a message from a topic, it must declare the type of the involved message. In lines from 11 to 14, four types of message are imported. These messages will be examined later on.

After having imported the needed modules the program can run. Firstly, the mapping program must initialize a node.

```
17      #Start up node of the program:
18      rospy.init_node('my_slam')
```

Table 4.3

The new node *my_slam* registers on the Master node. A node must be associated to the developed program to make it interact with the ROS framework. Master node can connect registered nodes to each other and make them communicate by means of topics. Then, my_slam node is linked to the required topics, stated in the following parts of the program.

4.3.2 Parameters setting

Another initial step is the set-up of the parameters used by the program.

```
21      #Set up parameters:
22      global grid_map, my_pose_x,my_pose_y,my_pose_theta, n_plots,plots,
        res, lo1,lo0,pz1m1,pz0m0, semaph
23      grid_map=numpy.zeros((5000,5000))
24      my_pose_x=[]
25      my_pose_y=[]
26      my_pose_theta=[]
27      n_plots=0
28      plots=[]
29      res=0.01
30      pm1z1=991/1000.0
31      pm0z0=(1000-6)/1000.0
32      lo1=math.log(pm1z1/(1- pm1z1))
33      lo0=math.log(pm0z0/(1- pm0z0))
34      semaph=0
```

Table 4.4

Some parameters are fixed, others are settable. At line 22 all the parameters are declared as global variables. In this way, they can be recalled into functions of the script maintaining their set value. A more in-depth analysis is carried out on this fundamental step. The first variable is *grid_map*, i.e. the belief matrix of the grid map. This is also the output variable of the mapping algorithm. Two important settable variables are stated here: the dimension of the grid map and the a priori knowledge of the map. The environment to map is usually completely unknown to the robot, so each cell belief has a value equal to 0 (map belief is represented by means of log odds ratios).

The dimension of the matrix is a settable variable, defined by the user depending on the granularity of the map and the extension of the environment. The variable used to set the resolution of each cell is *res*. Values assigned to grid_map dimension and res must be a trade-off based on the area to map. A smaller building allows to implement a lower resolution with a consecutive higher map matrix dimension. Lower the res value, higher the map resolution. A limit to the res value reduction is the time cost of the mapping process. It is harder to assign a laser measurement to the correct cell rather than to the next one, when they

are too little. A higher computation effort is needed. Additional computations are necessary if many cells result unseen between two laser rays. A laser ray is shifted from the previous one by 1° and some cells can be in this cone. If the cell dimension is too low, too many cells are not involved in laser scanning. Then, an additional algorithm must evaluate if the unseen cells can be considered occupied or free, depending from their position with respect to the nearby observed cells. A good value of resolution for the map is 1 centimetre. The building mapped in the thesis project is the LIM lab of Politecnico di Torino, where this work has been performed. A map size suitable for this environment is a square area of 50 x 50 metres.

In order to choose an appropriate dimension for the map belief matrix, correlation between the matrix and the built map is analyzed in the following. The developed map is a top-view 2D grid, thus it can be represented in a Cartesian plane. Entries of the matrix can also be depicted in a Cartesian plane, basing on their position into the matrix. Let us consider the first quadrant of the plane, with positive axes values. Entry $m_{0,0}$ of the map belief matrix coincides with the origin of the Cartesian plane. The first row of the matrix, with entries $m_{0,n}$ (n = 0, 1 ,2 ,3 ,4 ,5 …), represents the Cartesian horizontal axis x. The first column, with entries $m_{n,0}$ (n = 0, 1 ,2 ,3 ,4 ,5 …), represents the Cartesian vertical axis y. The resolution of the map indicates the distance between two consecutive entries, of the same row or column, in the Cartesian plot. If every matrix entry is plotted on this Cartesian plane and straight lines are drawn over the matrix rows and columns, the plot shown in *Figure 4.1* is obtained.

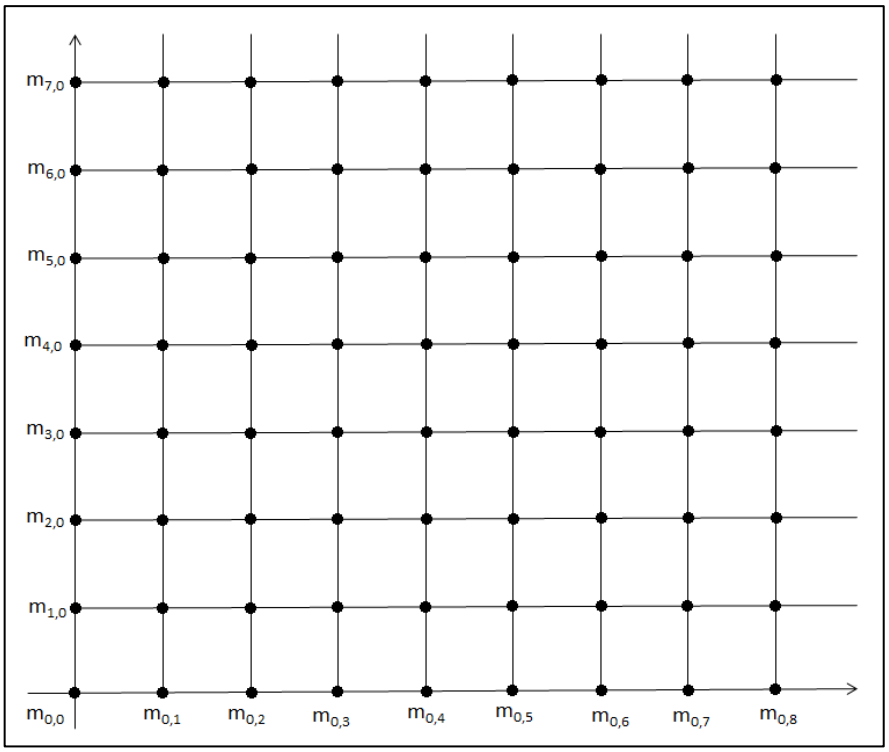


Figure 4.1

Each matrix entry is positioned at the intersection of two lines. An entry represents the occupancy belief of a cell of the map. This probability value is associated to the centre of that cell. For this reason, lines intersection of *Figure 4.1* are related to the centres of the cells of the grid map. The central point of each cell is 1 centimetre far from the middle points of neighbouring cells. All the cells have the same extension, so the grid map can be plotted considering a square area around each intersection.

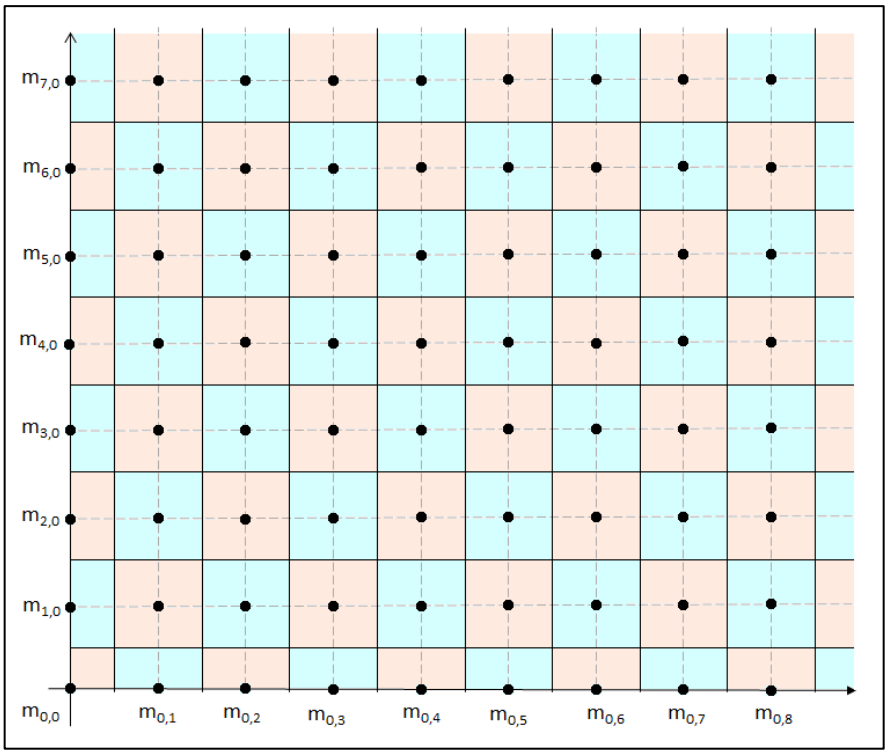


Figure 4.2

In *Figure 4.2* a grid map is depicted, showing its correlation with the matrix of map belief. The occupancy of cells is not considered yet. Dashed straight lines are the same of the previous graph in *Figure 4.1*. They represent rows and columns of the map matrix. Continuous straight lines define the edge of cell regions.

Once the size of the map area is established, the matrix dimension is equal to that size evaluated considering the resolution as unit of measurement. As already mentioned, the needed map size is 50 x 50 meters and the resolution chosen is 1 centimetre. The dimension of the grid matrix is 5000 x 5000. This matrix is initialized at line 23 of the mapping script by means of the numpy module. At the start-up, it is a 5000-by-5000 zero matrix.

The next three initialized parameters are *my_pose_x*, *my_pose_y* and *my_pose_theta*. They are three vectors that store the pose values registered during mapping execution. When the movement of a wheeled robot is constrained on a two-dimensional plane, it has three degrees of freedom. Robot can translate along the two axes defining the plane or rotate along an axis perpendicular to the

plane. Then only the three mentioned parameters are sufficient to describe the robot pose. All the past history of the pose values is used when the map is plotted to also print the path performed by the robot. If this path representation is useless, the three pose parameters can be associated to three values that are updated each time, without storing all the past pose. They are initialized like three empty vectors at the start-up. They cannot be set.

n_plots and *plots* are not settable parameters needed for the map printing in python, when the pyplot module is used. To plot more map plots during a mapping session, every map must be built, saved and printed on a new figure, until the stop of the program. In the map plotting function of the script, parameter plots is a list where different printed maps are saved in different locations. It is initialized like an empty list. Parameter n_plots is a counting variable needed for the association between a new map plot and the next empty element of the plots list. It is incremented by 1 every time a map is printed. n_plots is initialized with 0.

The next parameter, set at line 29, is res. It represents the resolution of the map grid. Its value is chosen by the user depending on a series of factors described above. In this thesis it is chosen equal to 1 centimetre. Odometry position and laser scan measurements taken from ROS topics are represented in metres. Since mapping program works on indexes of the map matrix to update the entries, all distance measurements must be transformed in centimetres. Parameter res is employed with this goal. Data coming from TurtleBot3 sensors are divided by res to represent them in centimetres. Therefore, res is initialized with 0.01.

From line 30 to 33, there are the parameters used in computation of the inverse measurement model. The occupation belief of a generic n cell, in log odds form, is stated by the equation (4.6), reported here:

$$l_{n,t} = l_{n,t-1} + l(m_{n,t} \mid z_t)$$

The belief is achieved adding to its previous value a probability value given by the inverse measurement model. It evaluates the probability of a state, given a measurement about it, $p(m_t \mid z_t)$. This probability value depends on the accuracy of the laser sensor used to obtain the measure. For testing the 360° Lidar accuracy, the TurtleBot3 Burger is placed at 0.5 metres from a wall and 1000 sets of laser measurements are taken. From each set, only the measure corresponding to the laser ray casted at 0° is considered. These data are reported in the Appendix A.

Three values appear more often: 0.5, 0.500999987125 and 0.501999974251. There is not only one right value, due to not perfect precision of the Lidar. Other values can be considered wrong and their presence reduces the accuracy of the sensor. Some of them are random values around the possible right one. Instead, measurement 0.0 represents a failure in the functioning of sensor.

The probability $p(m_t \mid z_t)$ can be evaluated by means of the ratio between the number of correct measurements and the total. Number of recurrences of the three more frequent values is 991. The total number of measurements is 1000. The probability value obtained by the inverse measurement model is stored in parameter pm1z1.

$pm1z1 = p(m_t=1 \mid z_t=1) = 991/1000.0$

pm1z1 represents the probability that a cell is occupied when it is seen occupied by the laser sensor. It is a degree of correctness of Lidar when facing an obstacle.

Parameter pm0z0 is its dual. It represents the degree of correctness of the sensor when there are not obstacles in front of it. Other 1000 sets of measurements are accomplished as before, but this time TurtleBot3 is facing a free hallway. Once again only measures relative to the laser ray casted at 0° are considered. Results are not shown, since almost all the values are equal to “inf”. There are not inaccurate values, but faulty measurements still occur. inf” value means that the distance detected in front of the robot is higher than Lidar maximum range, even

if some exceptions can arise. From the 1000 laser measurements, there are six 0.0 values, as before related to the wrong functioning of the sensor. Parameter pm0z0 is computed like the ratio between correct detections and total ones.

$pm0z0 = p(m_t=0 \mid z_t=0) = (1000-6)/1000.0$

Belief is represented in log odds form, then pm1z1 and pm0z0 must be transformed in log odds ratios. pm1z1 is divided by its opposite binary case, i.e. the probability that the detected cell has state 0 even if it is measured 1. This last probability is pm0z1 and is equal to (1-pm1z1). Log odds ratio of pm1z1 is represented by the parameter *lo1*, computed at line 32. Log odds ratio of pm0z0, *lo0*, is calculated in the same way at line 33. Their values are computed explicitly below:

$lo1 = \log(pm1z1/(1-pm1z1)) \approx 4.7015$

$lo0 = \log(pm0z0/(1-pm0z0)) \approx 5.1100$

During the update of the map, belief of the cells involved in the measurement process is evaluated. Parameter lo1 is added to the previous belief of cells detected occupied. Parameter lo0 is subtracted to the belief of cells detected free.

The last parameter initialized here is *semaph*. It is not settable and is equal to 0 at the start-up of the program. Its function is to block more updates of the map with the same laser scan data. This blocking parameter will be better analyzed later on.

4.3.3 The “main” and the structure of the program

The rest of the program can be decomposed in some macro groups. From line 37 to 53 there are three mathematical functions, invoked during the program. From line 56 to 74 there are the callback functions, run in response to messages

published on a subscribed ROS topic. From line 77 to 173 there is the principal function, that updates the map belief matrix depending on odometry and laser scan input data. From line 176 to 196 there are the threading classes that handle the execution of more processes at the same time. From line 199 to 215 there is the function that plots the map belief matrix into a grid map. In the end, from line 218 to 232 there is the main function of the python program. Each part of the script is presented in order of processing and analyzed. When the program starts, it imports specified external module and functions, sets up all the parameters and then runs the main function. Therefore, the main function is now considered.

```
218    #Main
219    if __name__=='__main__':
220        grid_map_plt=numpy.zeros((5000,5000))
221        global pub
222        pub=rospy.Publisher('/a_topic',String,queue_size=10)
223        sub=Subscribers()
224        sub.start()
225        time.sleep(1)
226        mpu=MapUp()
227        mpu.start()
228        while True:
229            print_map=raw_input("Do you want to plot the map? y/n ")
230            if print_map=='y':
231                grid_map_plt=grid_map
232                PlotMap(grid_map_plt)
```

Table 4.5

Let us look in detail at the various pieces of the main.

```
220        grid_map_plt=numpy.zeros((5000,5000))
```

This line initializes a second map matrix identical to that of the parameters setting. Values assigned to the entries are not important. The dimension of *grid_map_plt*

must be the same of grid_map. This new matrix is used at line 231 to decouple the map matrix continuously updating from the matrix that must be plotted when requested by the user.

```
221     global pub
222     pub=rospy.Publisher('/a_topic',String,queue_size=10)
```

Variable *pub* is initialized like global, so that it can be used by another function of the program. To the variable pub is associated a publisher. From now on, the node started by the program, my_slam, can publish messages on a specific topic. The first argument, */a_topic*, is the name of the topic where the node publishes. If a topic with this name is not present yet, it is created. The argument *String* defines the type of posted messages. The message type must be included in the imports at the beginning of the script (line 14). String messages allow to share vectors of characters. The *queue_size* states how many messages can be stored in the outgoing queue. When the ROS messages that must be published are sent faster than the capacity of the system to publish them to selected topic, the outgoing queue is filled. When a new message overcomes this limit, the older stored message is deleted.

```
223     sub=Subscribers()
224     sub.start()
```

A variable named *sub* is associated to the class *Subscribers*. The class is initialized and waits. In line 224, when the command *start()* is sent, the function *run(self)* of the class associated is executed. This class is used to create a threading process, i.e. a simultaneous process that works in parallel to the main one. This new process subscribes my_slam node to the topics that provide the inputs for the MapUpdater function.

```
225     time.sleep(1)
226     mpu=MapUp()
227     mpu.start()
```

The same procedure holds for the class *MapUp*. At line 226 it is initialized and at following line it is started. However a time.sleep() function is introduced. This function stops the execution of the program for a time equal to the value indicated by its argument, evaluated in seconds. Class MapUp creates another threading process that keeps MapUpdater function active, selecting cases in which the map belief must not be updated. Since the MapUdater function needs measurements published on topics /odom and /scan and these data are received only when the class Subscribers has run, a delay of 1 second is introduced between the starting of these two classes. In this way, MapUpdater function runs when node my_slam has subscribed to the required topics and the first messages have already been received.

```
228     while True:
229         print_map=raw_input("Do you want to plot the map? y/n ")
230         if print _map=='y':
231             grid_map_plt=grid_map
232             PlotMap(grid_map_plt)
```

The main function ends with an infinite loop started at line 228. It stops when the program is shut down. In the terminal window, the program asks to the user if he wants to plot the grid map, then waits for an input by the terminal. When “y” is typed and entered, lines 231 and 232 are executed. If other characters are typed, program shows again the message of line 229 on terminal and waits for another input. If the user wants to plot the map, he sends the “y” command. Program enters the if statement. Firstly, matrix grid_map, that contains the map updated until that moment, is copied into the matrix grid_map_plt. Then, the function PlotMap is executed for printing map matrix like an image. The argument of this function is grid_map_plot. Then the plotted map is not grid_map but its copy.

Matrix grid_map can be continuously updated, without influencing the printing process. Plotted map is the copy of the map saved at the moment of the request. As a result, variables grid_map, continuously updated, and grid_map_plt, processed to print the grid map, are decoupled. The image of the map is available when the user wants without stopping the mapping process.

4.3.4 Subscribers

The main process is blocked into an infinite loop. Simultaneously, other two processes are run by two threading classes. The first executed is reported below in *Table 4.6.*

```
188   #Set up subscribers to ROS topics:
189   class Subscribers(threading.Thread):
190     def __init__(self):
191       threading.Thread.__init__(self)
192     def run(self):
193       subLs=rospy.Subscriber('/scan', LaserScan, callbackLs)
194       subOd=rospy.Subscriber('/odom', Odometry, callbackOd)
196       rospy.spin()
```

Table 4.6

Line 195 is omitted because it belongs to an improvement developed later. It will be introduced in the next chapter. The first lines define the thread class and initialize it. Lines from 192 to 196 are the commands executed when the class is run. In lines 193 and 194 two subscribers are set. Each one has three arguments. The first is the name of the topic to which node my_slam subscribes. The second argument indicates the type of message read from that topic. Like it has been said for the publisher, the message type must be added to the import statements at the beginning of the script (lines 11 and 12). The third argument specifies a callback function. This callback is defined in the script and is executed every time a new message is posted on the specified topic. At line 196, *rospy.spin()* is a command of the ROS environment that keeps on hold the process. This threading

process does not terminate or restart. It does nothing but keeps active the two subscribers. Program node must subscribe to two topics: /scan and /odom. Data shared by these topics are the input arguments for the MapUpdater function.

Let us consider the two callback functions executed when a LaserScan message or an Odometry message are published respectively on /scan or /odom topics. When a laser scan is posted, the callbackLs function is run:

```
56      #Callback running when new LaserScan message published:
57      def callbackLs(msg):
58          global rngs_scan,semaph
59          rngs_scan=msg.ranges
60          semaph=1
```

Table 4.7

The argument of a callback is the message that invokes it. In this case this argument is named "msg" but can be associated to any variable. The name of the argument is used to recall the LaserScan message into the callbackLs function. At the beginning of the function block, two variables are declared like global. As a result, the function can modify their value and share it with the whole program. Then in the variable *rngs_scan* is stored the vector with the 360 measurements gathered by the Lidar. This vector is brought by the LaserScan message msg. As shown in the previous section, the variable of the message that holds the laser values is ranges. So rngs_scan is set equal to msg.ranges. This procedure is fundamental to lighten the processing load of the callback. If the callback is still busy when the next LaserScan message is published, a failure occurs and laser data are no more updated. This holds for every callback function. For this reason, coming laser data are stored into another variable that is processed by another function.

In the last line of the callbackLs the semaph value is changed. The variable semaph is initialized with 0 in the parameters setting section. As mentioned above, it is a blocking parameter for the map update. When it is equal to 0 the MapUpdater

cannot be executed, otherwise, when its value is 1, the map update is performed. The variable semaph is necessary. When the robot navigates, odometry and laser scans data are sent to the ROS network and shared with the Occupancy Grid Mapping program. The odometry information is published with a frequency of around 30 Hz, while laser scans are published with a frequency of around 5 Hz. Then, from a laser data and the next one, more odometry messages are posted on ROS. When a laser scan is available, MapUpdater is executed along with the most recent laser and odometry data. As mentioned before, the map update process must be accomplished before a new laser data is posted, to not waste these data. Then, in a loop cycle, MapUpdater is restarted once a cycle is completed, but now previous laser data with the current odometry are processed. This may lead to a wrong association between cells and their values. semaph variable prevents this error. At the beginning semaph =0 and the MapUpdater function is blocked, since there are not input data to process. When a laser scan is published on /scan topic, semaph is set to 1 in the callbackLs function. MapUpdater can be executed, updating the map matrix with the current values of laser data and odometry. When this process is completed, the semaph value must be set again to 0 to block another consecutive execution of MapUpdater. It waits until a new laser scan is published on /scan and, therefore, callbackLs sets semaph to 1.

When a message is posted on /odom topic, the callbackOd is run:

```
63      #Callback running when new Odometry message published:
64      def callbackOd(msg):
65          global my_pose_x,my_pose_y,my_pose_theta
66          my_pose_x.append((-msg.pose.pose.position.y+10.000)/res)
67          my_pose_y.append((msg.pose.pose.position.x+10.000)/res)
68          my_pose_theta.append(math.degrees(get_yaw_from_quaternion(
        msg.pose.pose.orientation))+90)
```

Table 4.7

Like in the previous callbackLs, message published on /odom topic is the argument of the callbackOd function. Three variables are declared "global" and store the useful pose values that must be processed in the MapUpdater function. Thus, callbackOd terminates and waits for another Odometry message. Pose variables stored from the argument message correspond to the three degrees of freedom of the robot. Let us consider a 3-dimensional reference frame xyz and TurtleBot3 moving on plane xy. Translations of the robot along x and y directions are saved in my_pose_x and my_pose_y. Rotation of the robot around z axis is stored in my_pose_theta vector. Since the measurements are published in metres unit, they must be transformed in centimetres, dividing them for the parameter res.

Odometry data published on /odom are computed from the TurtleBot3 onboard PC, based on the measurements of the wheel encoders and considering the robot reference frame. Map has another reference frame different from the robot one. Odometry data must be translated to the map reference frame to be processed from the MapUpdater function.

Robot and map reference frames are shown in *Figure 4.3*:

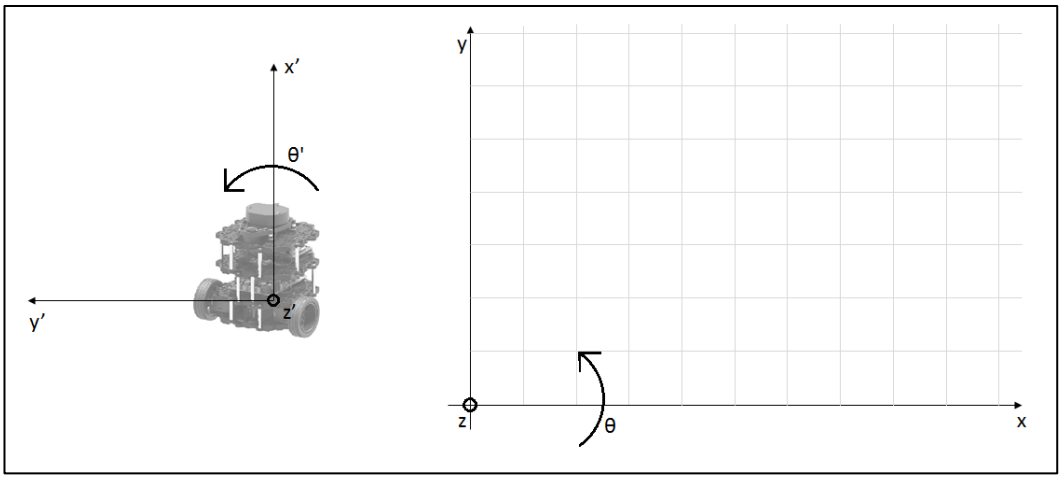


Figure 4.3

At the start-up of the program, robot and map reference frames coincide. Their z axis is perpendicular to the ground plane, exiting from it. If the robot is moved

straight ahead, in the map representation it follows the x axis. Instead, a robot that moves forward without turning is usually desired to be represented by a translation along the direction of y axis. This provides a better understanding of the map and of the path accomplished by the robot, for the user who generally knows only the initial pose of the robot. Thus, the desired reference frame orientation of the robot is indicated in *Figure 4.3*. It is obtained by means of a 90° rotation of the reference frame around the z axis. The rotation matrix is the following:

$$R = \begin{bmatrix} \cos(90°) & -\sin(90°) \\ \sin(90°) & \cos(90°) \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

Then, the transformed reference frame of the robot is:

x’ = -y

y’ = x

z’ = z

Rotations θ of the robot around the z axis, when turning, must be evaluated in the new reference. Since the reference frame x’y’z’ is rotated by 90° around the axis z’ with respect to the reference frame xyz, rotations θ’ of the robot are computed adding 90° to the measured ones, θ.

$\theta' = \theta + 90°$

Odometry message read from /odom topic has the pose orientation represented as quaternions. A function must be introduced to transform quaternions in the Euler angles: roll, pitch and yaw. Since only rotation around the axis z is needed, a

function is introduced in the script, to accomplish the angles transformation and the selection of the yaw.

```
37    #Function computing the yaw angle from quaternion:
38    def get_yaw_from_quaternion(quaternion):
39      quaternion_list=[quaternion.x, quaternion.y, quaternion.z,
      quaternion.w]
40      (roll,pitch,yaw)=euler_from_quaternion(quaternion_list)
41      return yaw
```

Table 4.8

At line 39 a list is created to have a suitable input for the transformation function. At line 40 an external function is used to accomplish the transformation from quaternions to Euler angles. This function is imported at line 7. Then at line 41 only the yaw is returned.

Returning on callbackOd of *Table 4.7*, line 68 computes the orientation of the robot in the x'y' plane:

```
68      my_pose_theta.append(math.degrees(get_yaw_from_quaternion(
      msg.pose.pose.orientation))+90)
```

Orientation data flow is the following. Orientation of the robot is given as quaternions from the variable *msg.pose.pose.orientation*. This variable is processed by the function get_yaw_from_quaternion, stated at line 37. This function returns the yaw angles in radians. math.degrees transforms the angle in degrees. To the obtained angle are summed 90°, representing the transformation from θ, evaluated in the xy plane, to θ' evaluated in x'y' plane. This final angle represents the orientation of the robot in the x'y' plane and is added to my_pose_theta list

At this point, robot reference frame and map reference frame have the same origin and the same axis z. For an optimal map update the robot reference frame must be translated. Indeed, the Occupancy Grid Mapping can update only the matrix entries with rows and columns from 0 to 5000. This matrix corresponds to a square area in the first quadrant of the Cartesian plane. If the robot is positioned in the origin of the map reference frame, it can update only the cells located into the first quadrant. To build a complete map, the robot and the visual area of its sensors must be entirely included, at each time step, into the map.

A roughly idea of the form of the building to map is needed, to predict the maximum extension of the place towards each possible direction. This is the case when the external dimensions of a building are known but the indoor environment must be mapped. For example, the lab, where the developed Occupancy Grid Mapping program is tested, extends only in forward and right directions, from the starting point. Then, the robot is positioned near the bottom left corner of the map. The origin of the robot reference frame is translated at point (1000, 1000) of the map. To perform this translation, each odometry position measurement, coming from the robot, is summed at 10 metres. It can be seen at lines 66 and 67 of the callbackOd function.

4.3.5 Map update function

The other threading class of the program handles the update of the map belief matrix. The class is reported in the following table:

```
176   #Thread class to continuously update map:
177   class MapUp(threading.Thread):
178       def __init__(self):
179           threading.Thread.__init__(self)
180       def run(self):
181           while True:
182               if semaph==1:
183                   rngs_data=rngs_scan
```

```
184             act_pose={"x":my_pose_x[-1], "y":my_pose_y[-1],
        "theta":my_pose_theta[-1]}
185             MapUpdater(rngs_data,act_pose)
```

Table 4.9

In the first lines the class is initialized. When it is run, an infinite loop is performed, defined by the while statement. The process waits for semaph = 1. When laser data is published and the callbackLs sets the blocking variable semaph to 1, lines from 183 to 185 are executed. In these lines, two new variables are introduced. In these variables are stored the input data for MapUpdater function. These data are the current measurements of odometry, stated with respect to the robot reference frame x'y'z', and the current laser scan. At line 183, rngs_scan updated in the callbackLs is stored in rngs_data. At line 184, the last values of the robot pose vectors are saved into the dictionary act_pose. In line 185, the MapUpdater function is run.

The MapUpdater function receives as inputs the current data of laser scan and odometry. Depending on these data, it can update the occupancy belief value of the cells of the map. The occupancy belief of the cells is described in log odds form. Then, when a laser ray identifies an object or a free space, the cells involved are updated, according to the following formula:

$$l_{n,t} = l_{n,t-1} + l(m_{n,t} \mid z_t)$$

The occupancy cell belief is recursively updated adding to their previous belief value a fixed parameter. This fixed parameter is computed from the inverse measurement model. Using a TurtleBot3 Burger, this model leads to getting two parameters, lo1 and lo0, as demonstrated in the previous section. Cases of their use are specified during the function explanation. The MapUpdater function is shown in *Table 4.10*, but while its parts are analyzed, they are reported again, to guarantee a better understanding.

```
77      #MAP BELIEF UPDATE ALGORITHM:
78      def MapUpdater(rngs_data,act_pose):
79          global grid_map,semaph
80          for i in xrange(360):
81              #faulty measure
82              if rngs_data[i] == 0.0:
83                  continue
84              if (i+act_pose["theta"]>=-45 and i+act_pose["theta"]<45) or
        (i+act_pose["theta"]>=315 and i+act_pose["theta"]<405):
85                  #free space
86                  if rngs_data[i] > 3.5:
87                      cnt=1000
88                      ob_dist=350
89                      ind_x=int(round(ob_dist*math.cos(math.radians(
        i+act_pose["theta"]))+act_pose["x"]))
90                      ind_y=int(round(ob_dist*math.sin(math.radians(
        i+act_pose["theta"]))+act_pose["y"]))
91                      grid_map[ind_x,ind_y]-=lo0
92                  #obstacle detected
93                  else:
94                      ob_dist=round(rngs_data[i],3)/res
95                      ind_x=int(round(ob_dist*math.cos(math.radians(
        i+act_pose["theta"]))+act_pose["x"]))
96                      ind_y=int(round(ob_dist*math.sin(math.radians(
        i+act_pose["theta"]))+act_pose["y"]))
97                      grid_map[ind_x,ind_y]+=lo1
98                      cnt=ind_x-int(round(act_pose["x"]))
99                  for x in xrange(int(round(act_pose["x"])),ind_x):
100                     y=int(round(yFromLine(act_pose["y"],act_pose["x"],
        ind_y,ind_x,x)))
101                     if cnt<2:
102                         grid_map[x,y]-=lo1
103                     else:
104                         grid_map[x,y]-=lo0
105                     cnt-=1
106             elif (i+act_pose["theta"]>=45 and i+act_pose["theta"]<135) or
        (i+act_pose["theta"]>=405 and i+act_pose["theta"]<495):
107                 #free space
108                 if rngs_data[i] > 3.5:
109                     cnt=1000
110                     ob_dist=350
```

```
111                 ind_x=int(round(ob_dist*math.cos(math.radians(
    i+act_pose["theta"]))+act_pose["x"]))
112                 ind_y=int(round(ob_dist*math.sin(math.radians(
    i+act_pose["theta"]))+act_pose["y"]))
113                 grid_map[ind_x,ind_y]-=lo0
114             #obstacle detected
115             else:
116                 ob_dist=round(rngs_data[i],3)/res
117                 ind_x=int(round(ob_dist*math.cos(math.radians(
    i+act_pose["theta"]))+act_pose["x"]))
118                 ind_y=int(round(ob_dist*math.sin(math.radians(
    i+act_pose["theta"]))+act_pose["y"]))
119                 grid_map[ind_x,ind_y]+=lo1
120                 cnt=ind_y-int(round(act_pose["y"]))
121             for y in xrange(int(round(act_pose["y"])),ind_y):
122                 x=int(round(xFromLine(act_pose["y"],act_pose["x"],
    ind_y,ind_x,y)))
123                 if cnt<2:
124                     grid_map[x,y]-=lo1
125                 else:
126                     grid_map[x,y]-=lo0
127                 cnt-=1
128         elif (i+act_pose["theta"]>=135 and i+act_pose["theta"]<225) or
    (i+act_pose["theta"]>=495 and i+act_pose["theta"]<585):
129             #free space
130             if rngs_data[i] > 3.5:
131                 cnt=1000
132                 ob_dist=350
133                 ind_x=int(round(ob_dist*math.cos(math.radians(
    i+act_pose["theta"]))+act_pose["x"]))
134                 ind_y=int(round(ob_dist*math.sin(math.radians(
    i+act_pose["theta"]))+act_pose["y"]))
135                 grid_map[ind_x,ind_y]-=lo0
136             #obstacle detected
137             else:
138                 ob_dist=round(rngs_data[i],3)/res
139                 ind_x=int(round(ob_dist*math.cos(math.radians(
    i+act_pose["theta"]))+act_pose["x"]))
140                 ind_y=int(round(ob_dist*math.sin(math.radians(
    i+act_pose["theta"]))+act_pose["y"]))
141                 grid_map[ind_x,ind_y]+=lo1
```

```
142                 cnt=0
143             for x in xrange(ind_x+1,int(round(act_pose["x"]))+1):
144                 y=int(round(yFromLine(act_pose["y"],act_pose["x"],
    ind_y,ind_x,x)))
145                 if cnt<2:
146                     grid_map[x,y]-=lo1
147                 else:
148                     grid_map[x,y]-=lo0
149                 cnt+=1
150         elif (i+act_pose["theta"]>=225 and i+act_pose["theta"]<315) or
    (i+act_pose["theta"]>=-90 and i+act_pose["theta"]<-45) or
    (i+act_pose["theta"]>=585 and i+act_pose["theta"]<=630):
151             #free space
152             if rngs_data[i] > 3.5:
153                 cnt=1000
154                 ob_dist=350
155                 ind_x=int(round(ob_dist*math.cos(math.radians(
    i+act_pose["theta"]))+act_pose["x"]))
156                 ind_y=int(round(ob_dist*math.sin(math.radians(
    i+act_pose["theta"]))+act_pose["y"]))
157                 grid_map[ind_x,ind_y]-=lo0
158             #obstacle detected
159             else:
160                 ob_dist=round(rngs_data[i],3)/res
161                 ind_x=int(round(ob_dist*math.cos(math.radians(
    i+act_pose["theta"]))+act_pose["x"]))
162                 ind_y=int(round(ob_dist*math.sin(math.radians(
    i+act_pose["theta"]))+act_pose["y"]))
163                 grid_map[ind_x,ind_y]+=lo1
164                 cnt=0
165             for y in xrange(ind_y+1,int(round(act_pose["y"]))+1):
166                 x=int(round(xFromLine(act_pose["y"],act_pose["x"],
    ind_y,ind_x,y)))
167                 if cnt<2:
168                     grid_map[x,y]-=lo1
169                 else:
170                     grid_map[x,y]-=lo0
171                 cnt+=1
172         pub.publish('TOOOPIIIIIC!!!')
173         semaph=0
```

Table 4.10

In the first line of the function, two variables are declared as global, i.e. they can be modified by MapUpdater and be seen from the rest of the program.

```
79          global grid_map, semaph
```

The first variable is grid_map, the output of the function. It is not returned at the end of the function, but it is read from the main function, when the user wants to plot the map. The second variable is semaph, that must be set to 0 when the grid_map has been updated. Moreover, it must be read from the class MapUp. In this way another execution of MapUpdater with the same laser data is avoided.

```
80          for i in xrange(360):
81              #faulty measure
82              if rngs_data[i] == 0.0:
83                  continue
```

The updating function starts. The update of the map belief takes place every time a laser ray carries out a measurement on the real environment. Every time a set of laser data is available, this set includes 360 measurements, each one gathered by a laser ray at each degree around the robot. So, 360 distance measurements are considered individually and processed, by means of the for cycle at line 80.

From now on, the function considers only one laser measurement relative to a specific angle. A first distinction is accomplished at the following line. If the reported laser measurement is equal to 0.0, a sensor error has occurred and the measurement relative to that angle is faulty. Then, the function skips the update relative to that measurement and the for cycle starts again considering i+1.

If the laser measure, relative to angle i, is different from 0.0 the function goes on.

```
84          if (i+act_pose["theta"]>=-45 and i+act_pose["theta"]<45) or
        (i+act_pose["theta"]>=315 and i+act_pose["theta"]<405):
…       …
…       …
106         elif (i+act_pose["theta"]>=45 and i+act_pose["theta"]<135) or
        (i+act_pose["theta"]>=405 and i+act_pose["theta"]<495):
…       …
…       …
128         elif (i+act_pose["theta"]>=135 and i+act_pose["theta"]<225) or
        (i+act_pose["theta"]>=495 and i+act_pose["theta"]<585):
…       …
…       …
150         elif (i+act_pose["theta"]>=225 and i+act_pose["theta"]<315) or
        (i+act_pose["theta"]>=-90 and i+act_pose["theta"]<-45) or
        (i+act_pose["theta"]>=585 and i+act_pose["theta"]<=630):
…       …
…       …
```

An if statement divides the Cartesian plane around the robot in four regions. Depending on the region in which the laser ray ends up, one of the four if-blocks is executed. The division, performed on the area around the robot, has the goal to optimize the cells selection during the belief update. In this phase the selector must decide which cells are hit or traversed by the considered laser ray. These four regions are shown in *Figure 4.4*. At each region corresponds the number of the script line that identifies it.

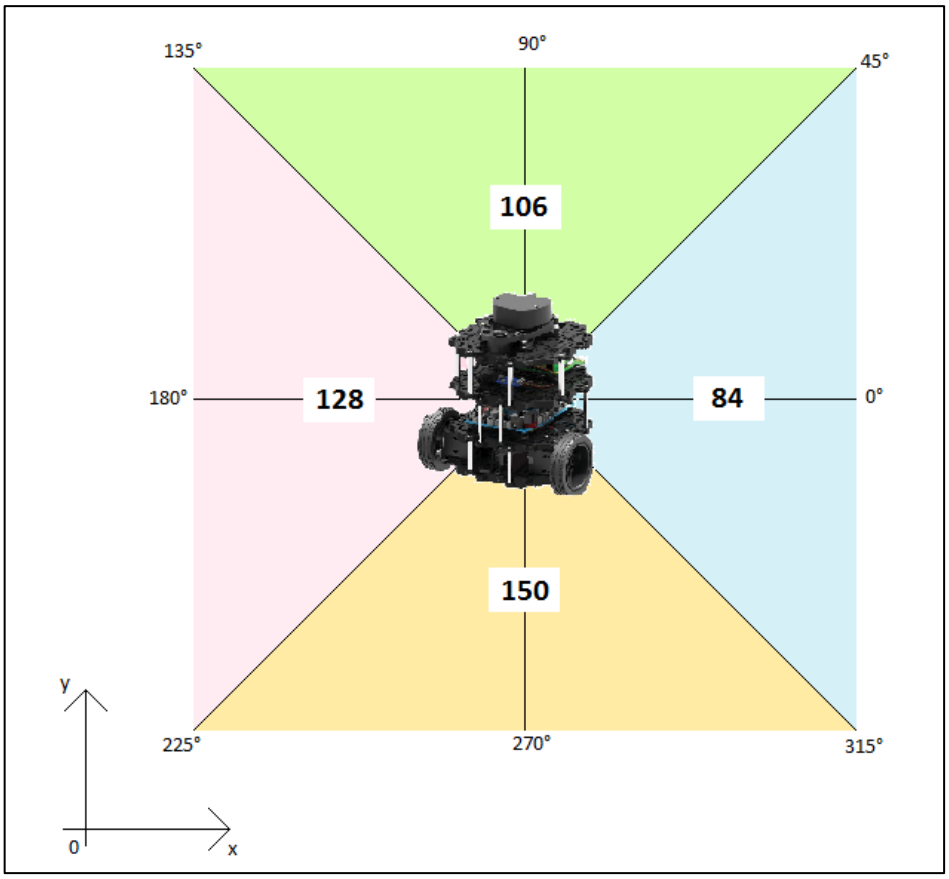


Figure 4.4

The region is selected based on the point where the measurement is gathered by the laser scanner. Then, this point is found by means of the angle of the considered laser ray, with respect to the map reference frame. The angle of the considered laser ray, at i-iteration of the for-cycle of line 80, evaluated in the map reference frame, is equal to the angle i of the ray with respect to the robot reference frame plus the orientation of the robot considered in the map reference frame (i+act_pose[“theta”]). This sum may have a value between -90° and 630°. Indeed, i ranges between 0° and 360°, while pose[“theta”] ranges from -90° to 270°. All possible cases are reported in the shown if statement.

Let us consider the first block, from line 84 to 105. There are two identified cases, based on the maximum range of the Lidar (3.5 metres). The first case is the detection of free space, so no obstacles are detected along the direction of the laser ray, until 3.5 metre. If an obstacle is detected at a higher distance, it is

considered out of range. All the cells involved in the measurement must modify their belief values. The second case is the obstacle detection. In this case, the last cell hit by the laser ray must arise its occupancy belief, while the previous ones traversed by the ray are considered free. They must decrease their occupancy belief.

In the following part, the program processes only the last cell relative to these two cases (lines 85 to 98). From line 99 to 105, all the other involved cells are considered free space, since traversed by the laser ray.

```
85              #free space
86              if rngs_data[i] > 3.5:
87                  cnt=1000
88                  ob_dist=350
89                  ind_x=int(round(ob_dist*math.cos(math.radians(
        i+act_pose["theta"]))+act_pose["x"]))
90                  ind_y=int(round(ob_dist*math.sin(math.radians(
        i+act_pose["theta"]))+act_pose["y"]))
91                  grid_map[ind_x,ind_y]-=lo0
```

When an obstacle is detected at a distance higher than 3.5 metres or is not detected (rngs_data[i]=inf), the last cell of the visual range is considered free. The parameter *cnt* of line 87 is explained later on. At line 88, the variable *ob_dist* stores the maximum distance reached by the laser ray, considered in centimetres. In this case ob_dist is equal to 350. The selection of the cell corresponding to the point reached by the laser ray at 3.5 meters is achieved in lines 89 and 90. A cell is represented by an entry of the map matrix grid_map. The problem of the cell identification in the map is transposed to indices identification relative to the entry associated to that cell. The indices of the entries of the map matrix are integer positive numbers (0 included). Each integer index corresponds to a distance value on the x and y axes of the grid map, evaluated in centimetres from the origin. Thus, the indices are evaluated locating the point reached by the laser ray on the grid

map. The x and y components of this point are reported on the Cartesian axes. Indices are the integer values closest to the found components.

At line 89 index x is evaluated. The x component of the measured point is obtained by a sum of vectors and then the decomposition of the resulting vector. It is better explained in the following *Figure 4.5*.

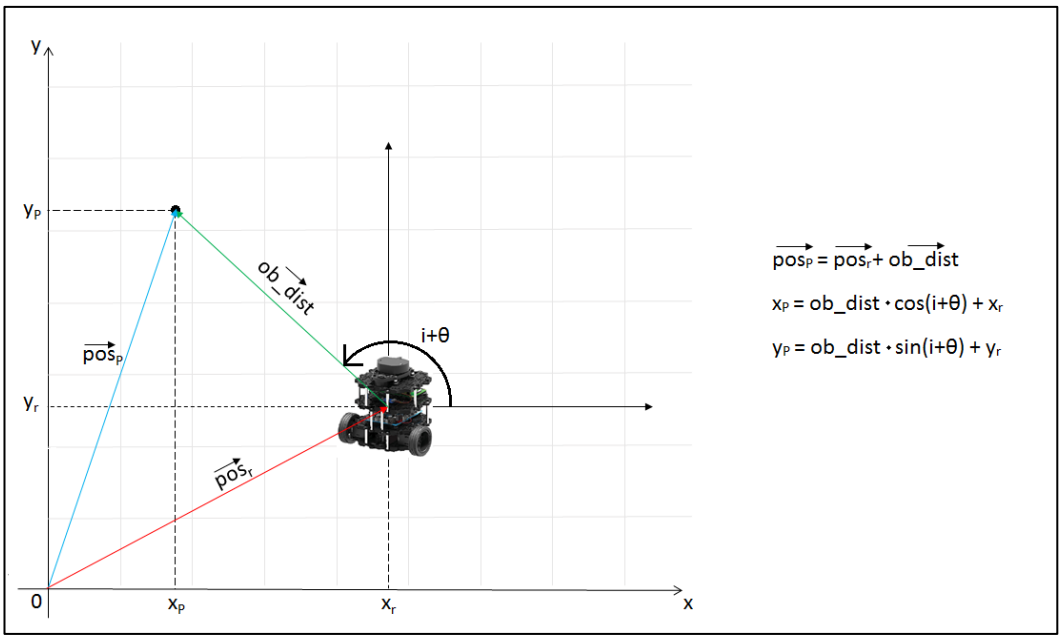


Figure 4.5

The vector of the measured point in the map reference frame is obtained summing the vector representing the point in the robot reference frame and the vector representing the robot in the map reference frame. Once the resulting vector has been decomposed, the x component is rounded and then transformed into an integer value. This integer represents the index x of the measured point, stored in variable *ind_x.* The y component is also rounded and transformed into an integer value. This value represents the index y and is saved in the variable *ind_y*.

At line 91, the value of the grid_map entry, with indices ind_x and ind_y just found, is updated. The belief in log odds form is updated subtracting the parameter lo0.

Indeed, lo0 represents the probability of correctness when a free space is detected by the Lidar. The system increases its belief regarding the vacancy of that cell.

```
92              #obstacle detected
93              else:
94                  ob_dist=round(rngs_data[i],3)/res
95                  ind_x=int(round(ob_dist*math.cos(math.radians(
        i+act_pose["theta"]))+act_pose["x"]))
96                  ind_y=int(round(ob_dist*math.sin(math.radians(
        i+act_pose["theta"]))+act_pose["y"]))
97                  grid_map[ind_x,ind_y]+=lo1
98                  cnt=ind_x-int(round(act_pose["x"]))
```

When an obstacle is detected by the considered laser ray, the variable ob_dist stores the measurement value, rounded and transformed into centimetres. The indices computation is the same of the free space case, only the ob_dist differs. Once ind_x and ind_y have been evaluated, the corresponding matrix entry is updated. In this case occupancy belief is increased, adding the lo1 parameter. lo1 represents the probability of correctness in the measurement, when an obstacle is detected. At line 98 the variable cnt is computed, but it will be analyzed in the next part of the MapUpdater function.

All the cells traversed by the laser ray are now updated. Firstly, an entry must be associated to each cell. Given a line in a grid space, this line must be represented by a set of cells. The Bresenham's line algorithm is used to reach this goal. It consists in increasing the discrete x or y value from the start point of a line to the end point. The choice of increasing the x or the y depends on the inclination of the line. Then, for each discrete value x or y the corresponding y or x is evaluated, by the line equation. The resulting value is rounded to the closest discrete value. When a line must be represented by set of cells, the result is a sequence of cells. They have an index always changing and an index computed by the Bresenham algorithm. An example of a rasterized line is shown in *Figure 4.6*.

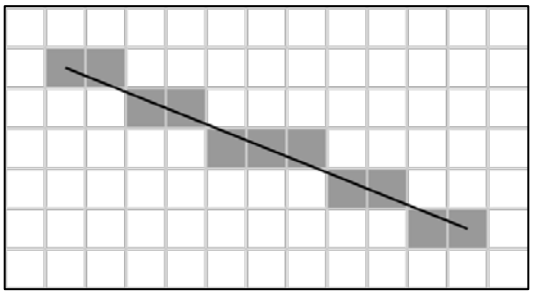


Figure 4.6

The Bresenham algorithm is implemented as two functions into the script. One increases the x, the other the y.

```
44      #Bresenham's line algorithm returning y with x increasing:
45      def yFromLine(y1,x1,y0,x0,x):
46          y=(y1-y0)/(x1-x0)*(x-x0)+y0
47      return y
48
49
50      #Bresenham's line algorithm returning x with y increasing:
51      def xFromLine(y1,x1,y0,x0,y):
52          x=(x1-x0)/(y1-y0)*(y-y0)+x0
53      return x
```

Table 4.11

MapUpdater function is considered again. The selection of the cells traversed by the laser ray is similar to the representation of a line in a grid space by a sequence of cells. The division of the visual area of the robot, shown in *Figure 4.4*, is needed to optimize the rasterization of the laser ray in the grid map. This rasterization is acomplished by means of the Bresenham's line algorithm. The part of the script is the following:

```
99              for x in xrange(int(round(act_pose["x"])),ind_x):
```

```
100             y=int(round(yFromLine(act_pose["y"],act_pose["x"],
    ind_y,ind_x,x)))
101             if cnt<2:
102                 grid_map[x,y]-=lo1
103             else:
104                 grid_map[x,y]-=lo0
105             cnt-=1
```

The region considered is the area 84 in *Figure 4.4*. The laser ray has a slope ranging from -45° to 45°. The Bresenham algorithm is hence implemented increasing the integer value of the index x (line 99). The starting index represents the actual position x of the robot rounded to the closest index x. The closing one is the index ind_x, obtained from the last if statement. Considering these two limit values, the position x of the robot must be involved in the update step of its occupancy belief as a free space. If the robot moves onto that cell, its occupancy belief must be considerably lowered. Indeed, considering that the cell, where the robot is positioned, is involved in this step for every laser ray i, its occupancy belief decreases very much. Instead, the cell corresponding to the ind_x has already been evaluated in the previous if statement and must not be included in the for cycle. At line 100 the y index corresponding to the increasing x is evaluated, by means of the function stated at line 45. Iteratively, cells constituting the rasterized line of the laser ray are found.

For each cell, the occupancy belief must be updated. The for cycle considers only the free cells traversed by the laser ray, not the hit ones. Then, in this cycle, all the believes are updated subtracting to their previous value the parameter lo0. Because of the inaccuracy of the Lidar when facing an obstacle, the object detected in a cell can be located in another closer position of the real environment. For this reason, the occupancy belief updating of the traversed cells may be differentiated, when these cells are close to a detected obstacle.

The purpose of the cnt parameter is to differentiate the traversed cells. If the laser ray has detected an obstacle, cnt variable is set in line 98. Its value provides the number of cells located between the obstacle and the robot. A for cycle is started

at line 99, where depending on the value of cnt, the position of the considered cell is checked. Cells can be located between the robot pose and the final point reached from the laser ray. cnt is set equal to the total distance. The traversed cell considered is the one containing the TurtleBot3. cnt is greater than 2, in general, so lo0 is subtracted at its belief. At the end of the for cycle, the value of cnt is decreased by 1 and the next cell is considered. When cnt value is less than 2, the corresponding cell is updated using no more lo0, but lo1. lo1 is subtracted to the previous occupancy belief of the cell, that is near to a cell detected occupied. If the occupancy measurement is faulty, the error is overcome by the next laser scan, since when an obstacle is detected to the belief is added +lo1. The subtraction of lo0 represents a higher confidence about the state of the cell. When this state is less reliable, lo1 is used.

In the case of free space detected, no distinctions are accomplished in the update of traversed cells. At line 87, cnt variable is set to 1000, a high value that cannot be decreased below 2 in the for cycle of line 99.

When the laser ray ends up in another region (depicted in *Figure 4.4*), the procedure is the same with some adjustments. When the Bresenham's line filter is implemented for the regions 106 and 150, y is increased by 1 at each step and x is computed. In regions 128 and 150 when the for cycle is executed to update the belief of traversed cells, the variable increased at each step ranges from the position of the obstacle to the position of the robot, since the for-statement needs an increasing parameter. Requirements mentioned above still holds. Moreover, in these regions the cnt variable is set to 0 when an obstacle is detected. Therefore, at the end of the for cycle updating traversed cells, cnt is increased by 1.

```
172         pub.publish('TOOOPIIIIIC!!!')
173         semaph=0
```

The last two lines of MapUpdater are executed when the update of the map belief is completed. The first line publishes a String message on the topic /a_topic. It is

used to check the working frequency of the function, by means of the ROS framework. The second line sets the blocking variable semaph to 0. Then, MapUpdater cannot be run again, until a new laser scan is published by the TurtleBot3.

4.3.6 Map plotting function

While the threading processes work to update the map belief matrix, the main function asks the user on the command window if he wants to plot the map. When "y" is digited, PlotMap function is run. Its argument is the copy of the map matrix, updated until the moment in which the user asks for the print. The code of the function is reported in the following *Table 4.12*:

```
199    #Map plotting function:
200    def PlotMap(mpl):
201        global n_plots,plots
202        now_pose={"x":my_pose_x, "y":my_pose_y}
203        axis_x_ob=[]
204        axis_y_ob=[]
205        plots.append(plt.figure(n_plots))
206        occup_threshold=3
207        for i in xrange(len(mpl)):
208            for j in xrange(len(mpl)):
209                if mpl[i][j] > occup_threshold:
210                    axis_x_ob.append([i])
211                    axis_y_ob.append([j])
212        plt.plot(axis_x_ob,axis_y_ob,'k.')
213        plt.plot(now_pose["x"],now_pose["y"],'r-')
214        plots[-1].show()
215        n_plots+=1
```

Table 4.12

The matrix shared as argument is named *mpl* within the function. At line 201, two variables are introduced like global ones, n_plots and plots. As mentioned before, n_plots is a count variable to keep in memory the number of stored maps and to

distinguish different map figures. plots is the list where the plotted maps are saved. At line 202, the whole position vectors are stored into a dictionary. All the past values of position are registered by the robot and then can be printed into the map. At lines 203 and 204 two variables are initialized as empty vectors each time this function is executed. They are used to store the points that must be plotted in the current map. At line 205 a new figure is created, identified by the count variable n_plots. In this new figure the map will be printed.

At line 206, the parameter *occup_threshold* is set. This is the value with which the belief value of every cell is compared. Cells with a belief greater than occup_threshold ($bel(m_{n,t}) > 3$) are considered occupied. Cells with a belief less than -occ_threshold ($bel(m_{n,t}) < -3$) are considered free. Cells with a belief that ranges between the negative and the positive value of threshold ($-3 < bel(m_{n,t}) < 3$) are considered unknown. The parameter occup_threshold is chosen by the user. It can be set depending on the accuracy of the used sensors.

At lines 207 and 208, two for-cycles consider all the cells of the grid map by means of the entries of the map belief matrix. For each entry, at line 209 an if statement selects only the cells with a belief value greater than the positive threshold. The x and y components of the position of the selected cells are stored in the axis_x_ob and axis_y_ob vectors respectively. The for cycles end. At line 212, points stored in the two vectors just seen are plotted on the new figure created at line 205. The argument ‘k.’ indicates that these points are plotted as black points. The printed map is an occupancy grid map. The real environment is represented by a grid with fixed cells. When a cell is considered occupied a black point is plotted in the middle of the cell. The map is then represented by a matrix of points, each of which represents a region of the environment. The smaller each region is, the more accurate the map is.

In line 213, all the history of the positions of the robot is plotted on the same figure. Argument ‘r-‘ plots this history like a red line. The figure appears on the screen by means of the command in line 214, that shows the last element of the list plots. At the following line, the n_plots counter is increased by 1 and the

function is completed. The main function restarts its loop and asks on screen if the user wants to print the map again.

5 Conclusions

In this chapter the results are shown. In the first section the developed mapping program is run along with the autonomous navigation algorithm. The realized maps are analyzed and improved. The mapping program is then compared to Gmapping, another common mapping algorithm working with the same sensor data. In the last section, future implementations are presented.

5.1 Autonomous mapping and navigation

The Occupancy Grid Mapping program is developed to map an indoor environment and is set for working on the Turtlebot3 Burger. The full script is named *OccGridMapp*. It is reported in the Appendix of the thesis. This program accomplishes the SLAM, Simultaneous Localization and Mapping. To build the map of building without the human interaction, the Burger must move autonomously. For this goal, the Occupancy Grid Mapping is run alongside a program for the robot autonomous navigation. This program has been provided by Lorenzo Galtarossa, another student of Politecnico di Torino working on autonomous navigation at LIM.

TurtleBot3 Burger is started. The ROS Master node is launched by the laptop with the command "roscore". After this, bringup is executed from both the laptop and the Burger. Now all the needed nodes are active and publish messages on ROS topics, about the state of the robot and sensor measurements. Developed mapping program *OccGridMapp.py* and autonomous navigation program *autonomous_nav.py* can be started. autonomous_nav is run. On the terminal window it asks to the user if he wants the robot goes ahead or stops, typing "g" or "s" respectively. If another character is typed, the question is remade.

OccGridMapp is also executed and, on its terminal window, the program asks the user if he wants to plot the actual map.

“g” is typed and the Burger starts to move. The LIM hallway is mapped. The starting point is the bottom left corner of the lab. While moving, the Burger robot avoids obstacles and walking persons. In the meantime, the mapping algorithm registers all the detected obstacles and free spaces into its grid map. Walking persons and moving objects are detected and the map is updated with their detection. However, when they shifted position, the mapping program empties the map from their presence. Every second the map is updated around 5 times. If a leg of a moving person is detected in 2 consecutive time steps, after other 3 detections the cell believed occupied becomes free. The moving obstacle is eliminated from the map within 1 second.

The map plotted by the OccGridMapp program, using a TurtleBot3 Burger to gather the inputs and moving it with the autonomous_nav algorithm, is shown in *Figure 5.1*.

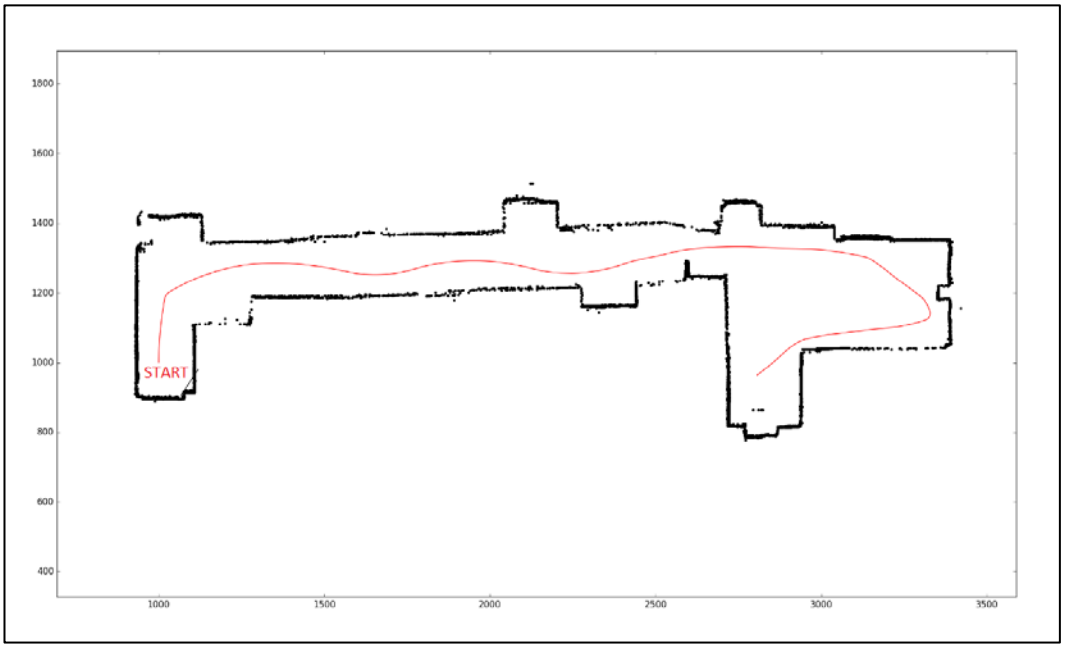


Figure 5.1: LIM hallway map realized by OccGridMapp

As mentioned in the previous chapter, the grid map is plotted like a matrix of points, centred in the middle of each cell. Each point represents the occupancy belief of the corresponding cell. Within this code, only the cells considered occupied are plotted, i.e. cells with an occupancy value greater than the positive occupancy threshold. These cells are printed as black points. Therefore, the map is a grid map filled by black points, representing little regions of the environment considered occupied. The unknown or free cells are not shown. If the complete map is needed, representing all the 5000 x 5000 cells of the grid with their associated believed state (occupied, free or unknown), the PlotMap function must be modified. At line 199 of the OccGridMapp program, PlotMap is substituted by the function in the following *Table 5.1*.

```
#Map plotting function:
def PlotMap(mpl):
   global n_plots,plots
   now_pose={"x":my_pose_x, "y":my_pose_y}
   axis_x_un=[]
   axis_y_un=[]
   axis_x_ob=[]
   axis_y_ob=[]
   axis_x_fr=[]
   axis_y_fr=[]
   plots.append(plt.figure(n_plots))
   occup_threshold=3
   for i in xrange(len(mpl)):
      for j in xrange(len(mpl)):
         if mpl[i][j] > occup_threshold:
            axis_x_ob.append([i])
            axis_y_ob.append([j])
         elif mpl[i][j] < -occup_threshold:
            axis_x_fr.append([i])
            axis_y_fr.append([j])
         else:
            axis_x_un.append([i])
            axis_y_un.append([j])
   plt.plot(axis_x_ob,axis_y_ob,'k.')
```

```
    plt.plot(axis_x_fr,axis_y_fr,'w.')
    plt.plot(axis_x_un,axis_y_un,'b.')
    plt.plot(now_pose["x"],now_pose["y"],'r-')
    plots[-1].show()
    n_plots+=1
```

Table 5.1

Vectors *axis_x_un*, *axis_y_un*, *axis_x_fr* and *axis_y_fr* are added to the function. In these vectors are stored cells whose belief value considers the corresponding cell unknown or free. In terms of code, at these new vectors are appended the indices of the entries representing an unknown or a free cell. At the end of the for-cycles, all the cells of the map are assigned to one of the three couple of vectors. These are: two vectors containing the indices of occupied cells, two vectors for the indices of free cells and two vectors containing the indices of unknown cells. After this, once at a time, each vector is processed and all the cells are plotted into the map. The cells are represented through their middle point. This point is black if the cell is considered occupied, white if considered free and blue if unknown, i.e. its state is too uncertain to be assigned to the other two cases.

The obtained map, printing all the cells of the grid basing on their associated occupancy belief, is shown in *Figure 5.2*. Executing another test, the Burger is moved in the same hallway with the same autonomous navigation algorithm.

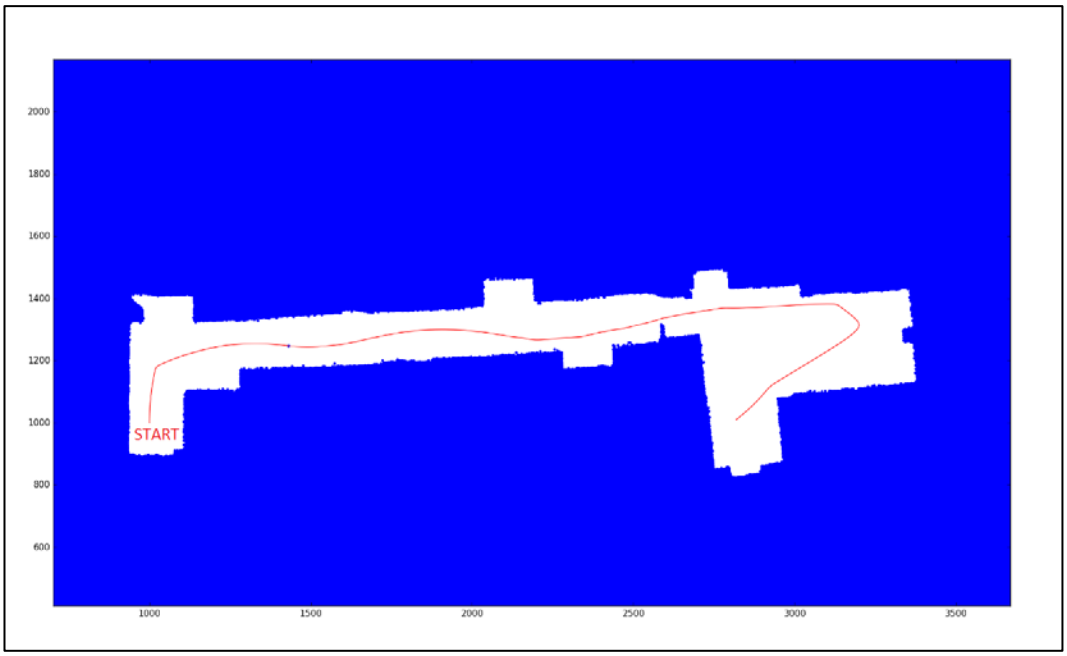


Figure 5.2: LIM hallway map realized by OccGridMapp, representing all cells

The map is the same, only representing all the stored occupancy believes regarding the cells. In the following, only the obstacles are shown, then the black points. This OccGridMapp program is run as shown in the Appendix B. In this way, it is easier to check the reliability and the precision of the map. Uncertain and free cells are still present in the map matrix and are updated by the mapping algorithm, but the output image shows only the occupied ones.

Let us consider the map printed in *Figure 5.1*. Some points of the walls are not been plotted. This occurs when some measurements are faulty and the following ones must correct those errors. Alternatively, the Burger, moving too fast and taking one measurement for each degree, may lose some points of the map. The missed cells can be updated when the robot comes back. When a completed map is needed at the first transition, an optimization can be carried out at the plotting time. Once the map is plotted but not shown yet, extra lines are added to connect closer points. The new PlotMap function is the following:

```
#Map plotting function:
def PlotMap(mpl):
  global n_plots,plots
  now_pose={"x":my_pose_x, "y":my_pose_y}
  axis_x_ob=[]
  axis_y_ob=[]
  plots.append(plt.figure(n_plots))
  occup_threshold=3
  for i in xrange(len(mpl)):
    for j in xrange(len(mpl)):
      if mpl[i][j] > occup_threshold:
        axis_x_ob.append([i])
        axis_y_ob.append([j])
  plt.plot(axis_x_ob,axis_y_ob,'k.')
  #Optimization algorithm
  for i in xrange(len(mpl)):
    for j in xrange(len(mpl)):
      if mpl[i][j] > occup_threshold:
        for n in xrange(1,6):
          for m in xrange(1,6):
            if mpl[i-n][j-m]>occup_threshold and mpl[i-n+1][j-
m+1]<=occup_threshold:
              plt.plot([i,i-n], [j,j-m], 'k-')
        for n in xrange(1,6):
          for m in xrange(1,6):
            if mpl[i+n][j+m]>occup_threshold and mpl[i+n-1][j+m-
1]<=occup_threshold:
              plt.plot([i,i+n], [j,j+m], 'k-')
        for n in xrange(1,6):
          for m in xrange(1,6):
            if mpl[i-n][j+m]>occup_threshold and mpl[i-n+1][j-m-
1]<=occup_threshold:
              plt.plot([i,i-n], [j,j+m], 'k-')
        for n in xrange(1,6):
          for m in xrange(1,6):
            if mpl[i+n][j-m]>occup_threshold and mpl[i-n-1][j-
m+1]<=occup_threshold:
              plt.plot([i,i+n], [j,j-m], 'k-')
        for n in xrange(1,51):
          if mpl[i-n][j]>occup_threshold and mpl[i-1][j]<=occup_threshold:
            plt.plot([i,i-n], [j,j], 'k-')
```

```
                if mpl[i+n][j]>occup_threshold and mpl[i+1][j]<=occup_threshold:
                    plt.plot([i,i+n], [j,j], 'k-')
                if mpl[i][j-n]>occup_threshold and mpl[i][j-1]<=occup_threshold:
                    plt.plot([i,i], [j,j-n], 'k-')
                if mpl[i][j+n]>occup_threshold and mpl[i][j+1]<=occup_threshold:
                    plt.plot([i,i], [j,j+n], 'k-')
    plt.plot(now_pose["x"],now_pose["y"],'r-')
    plots[-1].show()
    n_plots+=1
```

Table 5.2

The section *Optimization algorithm* is inserted in the function. At the first lines, only the occupied cells are selected, one by one, and checked. In this check, a 10 x 10 area around the cell is considered. If in this area another occupied cell is found, a line is printed between the two, if they are disconnected. The area is divided in four regions, likely a Cartesian plane with the origin located in the middle point of the checked cell. Four groups of for-cycles look into each region for a cell with an occupancy belief greater than 3, the occupancy threshold. When another occupied cell is found in a region, the cell immediately close, in the direction of the middle cell, is checked. If it is occupied, the optimization is useless. If it is free, the two occupied cells are disconnected in a very short space and, then, a line is drawn. Cells located in the horizontal and the vertical lines crossing the middle cell, are not checked here. Another for cycle is used. The procedure is the same.

Possible robot paths are not wasted, by means of this map optimization. The cell dimension is 1 x 1 centimetres, so the considered area is 10 x 10 centimetres. Since the Burger is wide 17,8 centimetres, it cannot pass through the lines drawn in this region.

In the tested context, an optimization using a 10 x 10 centimetres checking area has not been sufficient for some horizontal and vertical lines. For this reason, in this case, along the horizontal and vertical directions cells with a maximum distance of 50 centimetres from the middle one are considered. This modification

can be carried out when from the map it is clear that narrow passages are not wasted.

The optimized map is shown in Figure 5.3.

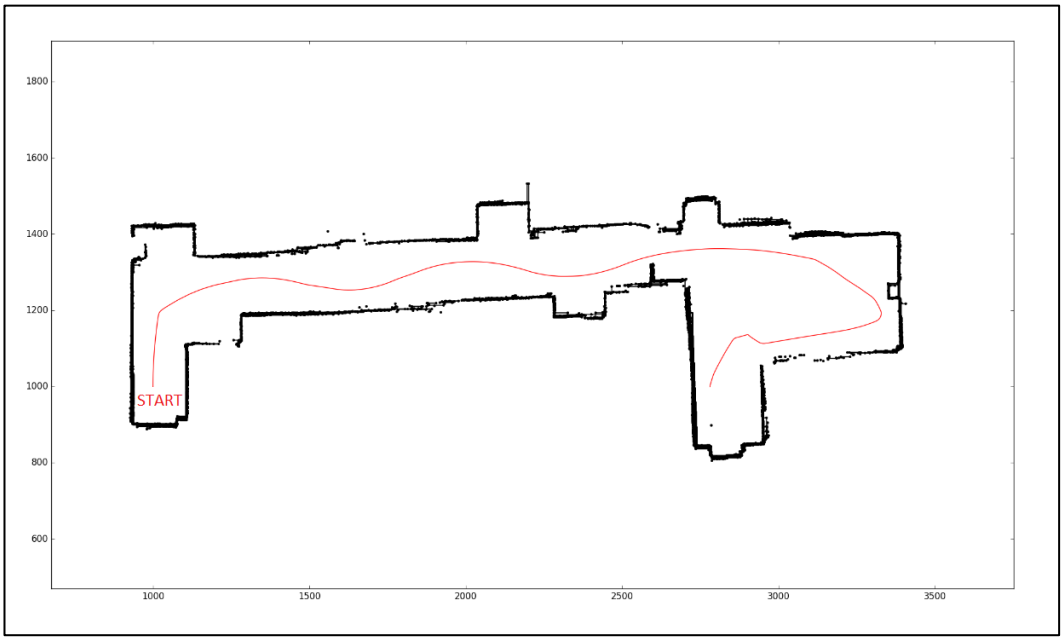


Figure 5.3: LIM hallway map realized by OccGridMapp with Optimization

Many free cells between points of the same wall are filled. Others do not. The same test is accomplished in a room of the LIM. The Burger is placed in a fixed starting point and the autonomous navigation and mapping of the area are executed. The autonomous_nav algorithm moves the robot around the room. At the same time the environment is mapped by means of the OccGridMapp program. The resulting map is shown in *Figure 5.4*.

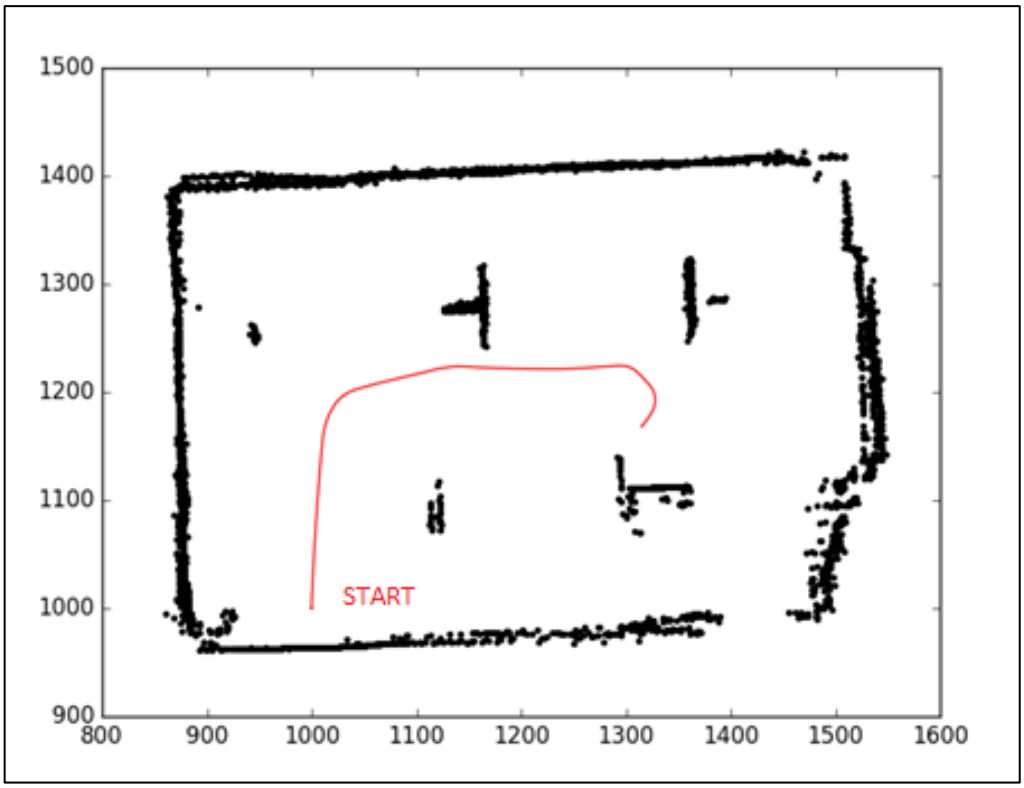


Figure 5.4: LIM room map realized by OccGridMapp

A second test is carried out in the same conditions, implementing the optimization algorithm within the OccGridMapp. The map is reported in *Figure 5.5*.

Figure 5.5: LIM room map realized by OccGridMapp with Optimization

At the corners the optimization is clear. In the left bottom corner, the trash can is visible in *Figure 5.4*. Because of its closeness to the wall, that does not allow the passage of the Burger, in *Figure 5.5* the trash can is depicted like an extension of the wall.

5.2 Comparison with Gmapping

A very common algorithm used to build a 2D map using a 360° laser scanner is Gmapping. It is the method implemented in the e-Manual website of ROBOTIS [4] to solve the Simultaneous Localization and Mapping problem. Gmapping is a particle filter algorithm. Each particle represents a possible grid map. Depending on the most recent odometry and laser scan data, it assigns a probability to the particles. It is run with the viewer tool rviz, to show the build of the map in real time and save it as an image. This tool and the intrinsic computations of a particle filter are obviously a higher computational cost, with respect to an Occupancy Grid

Mapping. If the execution of rviz is disabled, the grid map can be found on a dedicated topic in matrix form. When the user wants to print the map after a certain path or time, without the execution of rviz during mapping, another program must be run. The latter plots the map found on the assigned topic.

In Gmapping the frequency of the map update can be changed. The default value is 0.5Hz. Then the update of the map is accomplished within 2 seconds. The nominal maximum operating frequency is equal to 2 Hz, an update every 0.5 seconds. The map of the hallway of the LIM is built by means of Gmapping and the autonomous navigation algorithm used until now. Two maps are realized setting the update frequency to 0.5 Hz and to 2 Hz. Resulting maps are shown in the figure below.

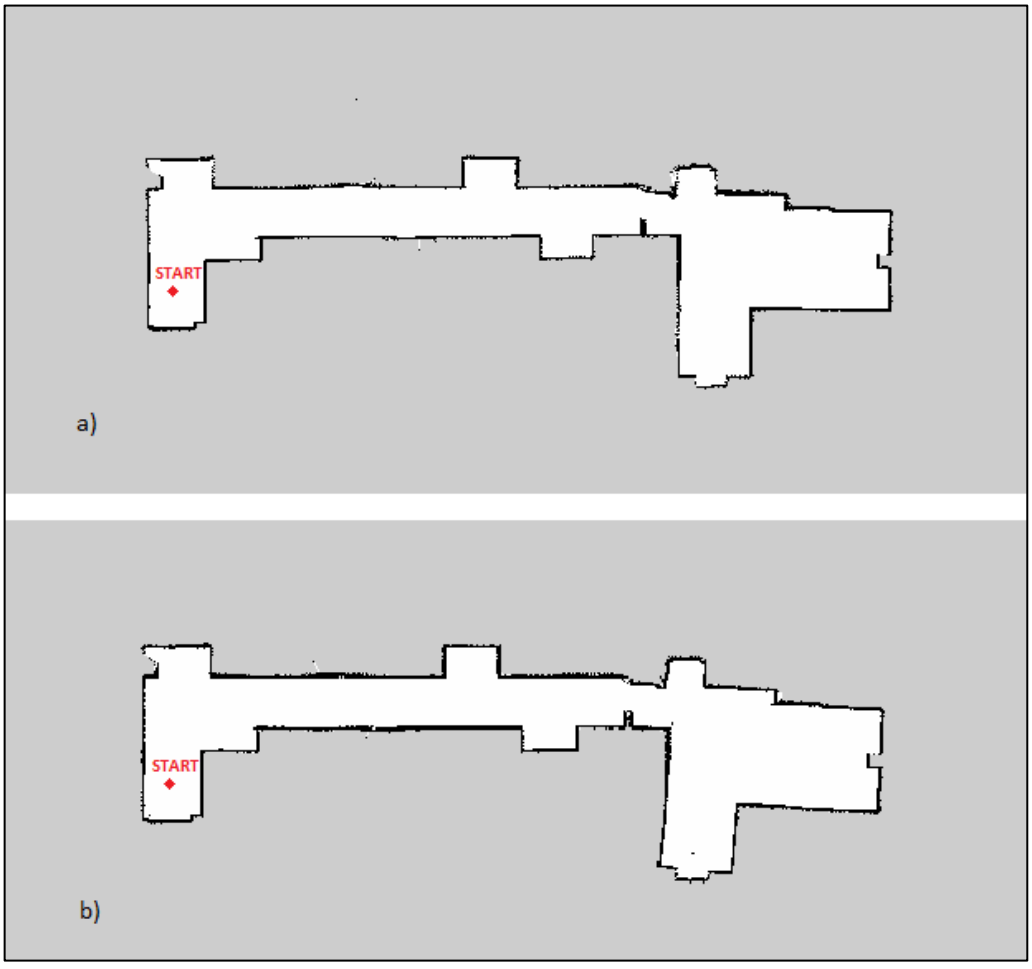


Figure 5.6: a) LIM hallway map realized by Gmapping at 0.5 H

b) LIM hallway map realized by Gmapping at 2 Hz

As the reader can see, increasing the update frequency the Gmapping algorithm starts to lose its odometry reliability and the map rotates at the end of the path. As mentioned in the previous chapter, a mapping algorithm should update its map with a frequency higher than the frequency of coming laser data. When working along with a second program, it may be necessary that the two programs, running one after the other, terminates their cycles between two laser data publications. Gmapping has an update frequency (maximum 2 Hz) lower than the publication frequency of the laser scans (about 5 Hz). The operating frequency is set as high as possible, then 2 Hz. The map considered for the comparison with the developed mapping algorithm is the *Figure 5.6 b)*.

While the particle filter assigns a probability to more possible grid maps, the Occupancy Grid Mapping assigns a probability to each cell of the map. The developed algorithm is executed with a lower computation effort, than the running of Gmapping alongside a viewer tool. Moreover, the map is shown when requested, without the execution of a secondary application or viewer.

The comparison between the printed maps obtained with Gmapping plus rviz and the mapping algorithm developed in this thesis work are reported in the following figures. The comparison is accomplished both in the hallway and the room of the LIM, previously seen in this chapter. Each map is realized with a different robot run.

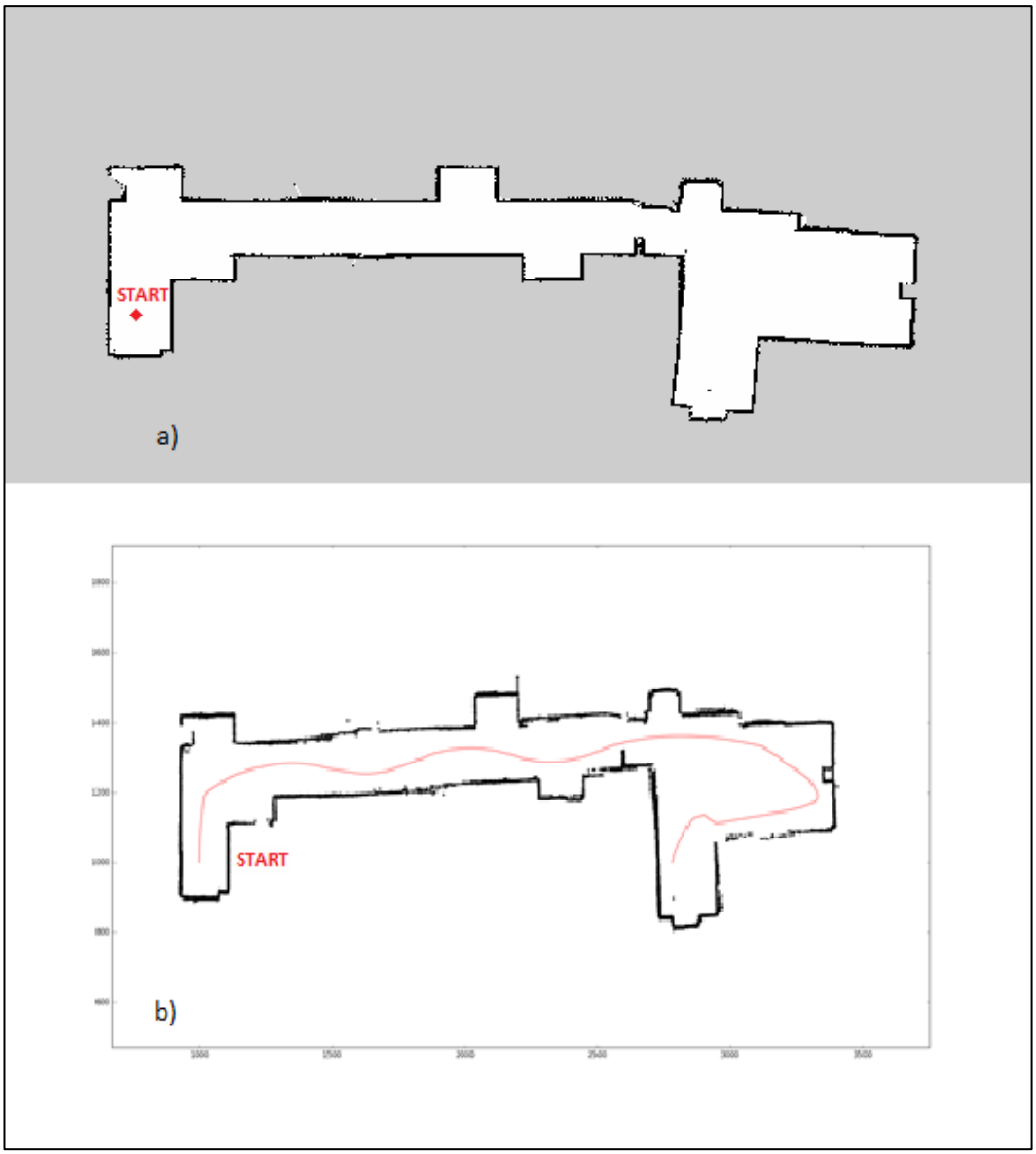


Figure 5.7: a) LIM hallway map realized by Gmapping at 2 Hz

b) LIM hallway map realized by OccGridMapp with Optimization

Figure 5.8: a) LIM room map realized by Gmapping at 2 Hz

b) LIM room map realized by OccGridMapp wit Optimization

The comparable parameters that show the better and the worse aspects of a method with respect to the other are the weight of the resulting map (in kilobytes), the mapping update rate and the reliability of the map. The resulting parameters relative to the two maps are compared in the following table:

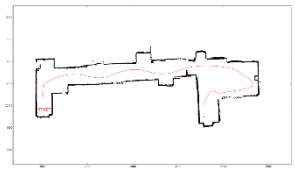
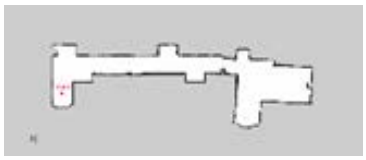
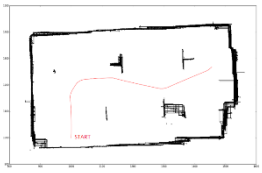
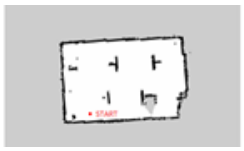
Parameters		*OccGridMapp*	*Gmapping*
Weight [KB]	Hall	81.8	312
	Room	99.7	144
Update rate [Hz]		6.600 ÷ 18.188	1.243 ÷ 1.679
Reliability	Hall		
	Room		

Table 5.3

The hallway has size of around 7 x 25 metres. The room around 4.5 x 6.5 metres. The weight of the hallway map printed by OccGridMapp is lower than the one obtained by Gmapping and rviz. Increasing the dimension of the mapped area, the weight of the map plotted by OccGridMapp is not increased. It depends on the number of visible plotted points, basing on the zoom chosen by the user. On the contrary, the map plotted by Gmapping has a weight greater than double of before. The map of the hallway realized with the Occupancy Grid Mapping algorithm is about four times lighter than the one built with Gmapping. Thus, the weight of a map printed by means of Gmapping and rviz is higher and quickly arises, not depending on zoom used. The map is saved by means of a ROS command from the terminal.

The second considered parameter is the update rate, i.e. the frequency at which the algorithms update their map belief. This parameter can be obtained using the ROS command “rostopic hz /topic_name”. Gmapping publishes its map every time this is updated, using the topic /map. OccGridMapp publish a string message on topic /a_topic every time the map is updated. To check their update rates, names of these topics are to be substituted in ROS command above. This command

returns a continuous print, on the terminal window, of the average frequencies at which messages are published in the considered topic. These values change in time. Then, during a mapping process, it can be possible to extract the minimum and the maximum values of the map update rate.

The frequency at which laser scans are posted on the /scan topic is about 5 Hz. To not waste any laser data set, the minimum update rate of the mapping algorithm must be at least 5 Hz. OccGridMapp has a minimum value of 6.600 Hz, so it can use every laser scan to update its map. Moreover, during the mapping process, the map update rate is usually around $8 - 10$ Hz, allowing other processes to be performed in series to it, before the following laser scan publication. Instead, the maximum map update rate of Gmapping is lower than 2 Hz. Results shows that Gmapping updates its map every 3, 4 or 5 laser measurents, depending on its momentary update rate. Then 2, 3 or 4 laser scan sets are losed.

The comparison about the reliability is carried out examining the map images of *Figure 5.7* and *Figure 5.8*. Maps realized by OccGridMapp are not perfectly optimized and some holes are still present. Gmapping is well optimized and eliminates these imperfections. Gmapping is sensitive to the increasing of the update frequency of the map. Odometry is lost and the map results deformed at the end of the robot path. Although the OccGridMapp works with a higher update rate, the odometry is not lost in the same way and the map is not deformed like the Gmapping one.

Moreover, from the map created with OccGridMapp the spatial quantities can be roughly derived, since at each unit of the plotted Cartesian plane corresponds 1 centimetre.

In summary, maps realized by Gmapping plus rviz have a greater reliability if working at low frequencies, with respect to maps built by OccGridMapp. If the working frequency of Gmapping is increased, the map is more reliable for the absence of holes in walls and, at the same time, less reliable because the map is deformed after a certain path. This deformation is not seen during the mapping of the room. The mapping algorithm developed in this thesis project plots a map with

a lower memory cost and is computationally faster. This is a fundamental characteristic needed for the accomplishment of certain tasks.

5.3 Future works

There are two improvements that can be implemented at the developed mapping algorithm. If the map must eliminate all the faulty and not seen holes in the walls at the first robot passage, a better optimization algorithm can be studied and implemented. The second and more important one is the improvement of the odometry data.

The robot localization task is assigned to the odometry values, given from the Burger through the ROS topic /odom. Odometry measurements have a low cumulative error that is not deleted and can only grow in time. This error is significant during a turn. The mapping algorithm, reported in Appendix B, partially overcomes this problem stopping to update the map during bends. Some lines of codes are implemented to the program presented in chapter 4. In the threading class MapUp another blocking condition is added to handle the map updating process.

```
176    #Thread class to continuously update map:
177    class MapUp(threading.Thread):
178        def __init__(self):
179            threading.Thread.__init__(self)
180        def run(self):
181            while True:
182                if ang_vel<=0.2 and ang_vel>=-0.2 and semaph==1:
183                    rngs_data=rngs_scan
184                    act_pose={"x":my_pose_x[-1], "y":my_pose_y[-1],
       "theta":my_pose_theta[-1]}
185                    MapUpdater(rngs_data,act_pose)
```

Table 5.4

In the if statement at line 182, a condition is added to start the current update of the map. The angular velocity of the robot must be lower than 0.2 rad/s and higher than -0.2 rad/s. To check the angular velocity of the robot another subscriber is initialized in the class Subscribers. Then, another callback function is created, *callbackVl*. In this function the angular velocity of the robot around the z axis is stored in a global variable named *ang_vel*, that is invoked by the class MapUp as shown. These implementations can be checked in the OccGridMapp script, reported in the Appendix B.

Let us consider again the mapping process of the hallway of the LIM. The starting part of the map is reported here. The map is realized without using the blocking condition about the angular velocity of the robot, stated at line 182.

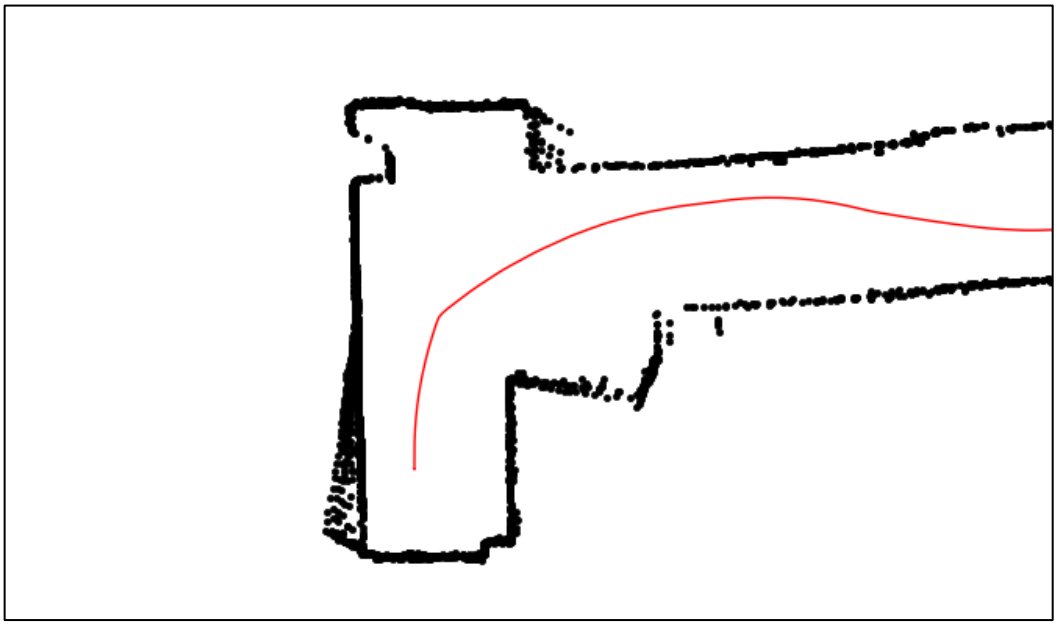


Figure 5.9: detail of map realized by OccGridMapp without blocking condition based on angular velocity

Using the blocking condition, the result is the following shown in *Figure 5.10*.

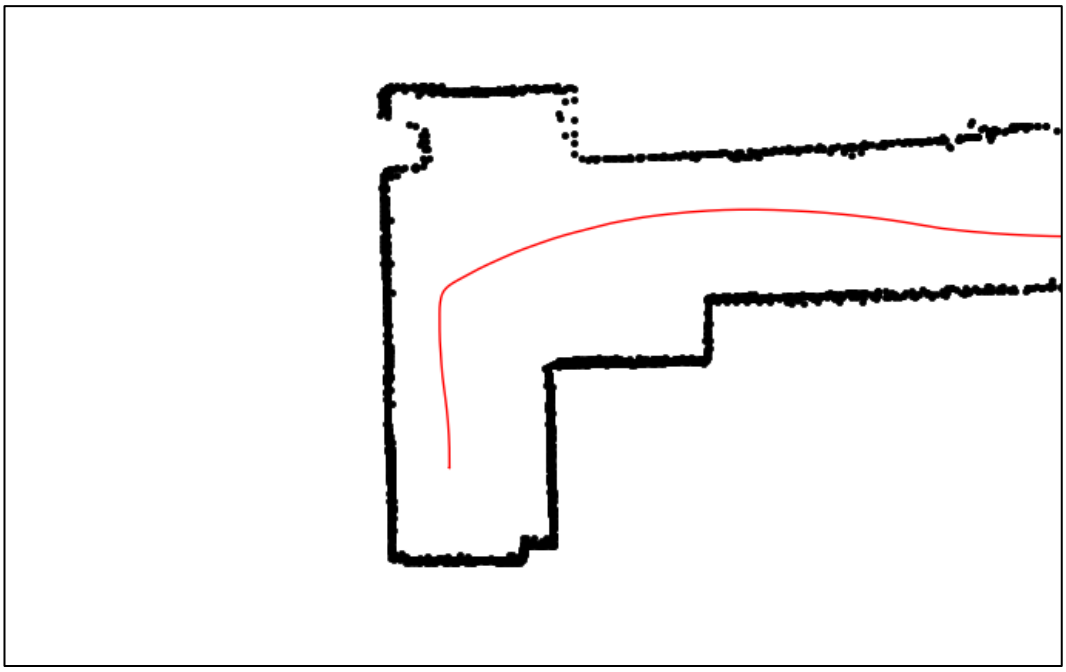


Figure 5.10: detail of map realized by OccGridMapp with blocking condition based on angular velocity

The error in the map is eliminated, but the error in the odometry is not. Then, moving, the Burger accumulates the error about its localization in the real environment. As a consequence, the following cells update, depending on this faulty odometry, can lead to a faulty belief of the map.

This error becomes evident when the robot accomplishes a 360° bend and returns in the already seen part of the map. This case is shown in *Figure 5.11*. The mapping test, in which the map of *Figure 5.1* has been realized, is continued and the Burger turns back in the same hallway.

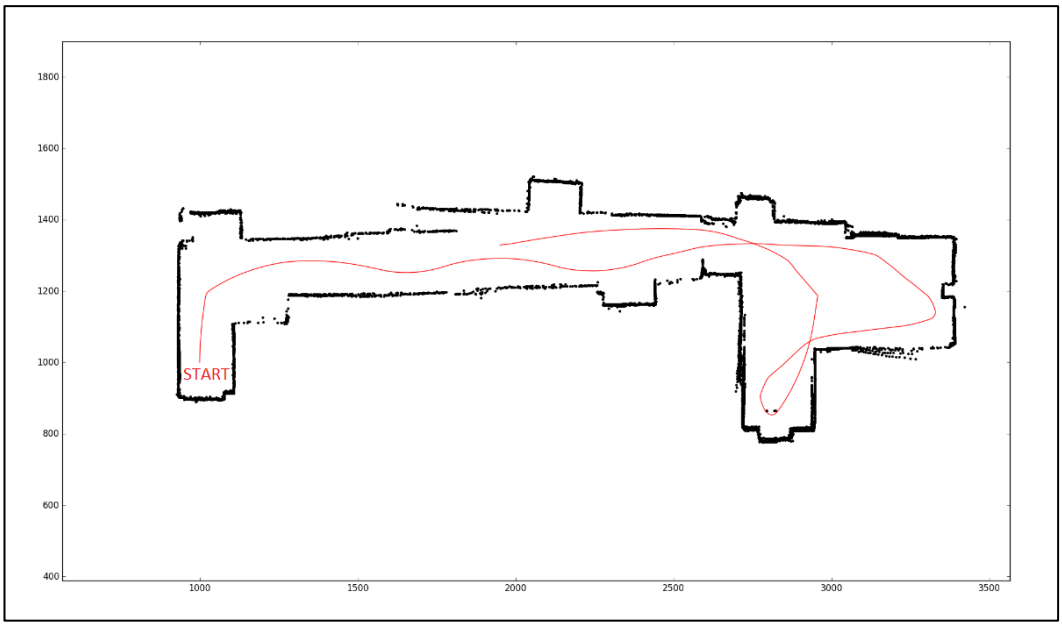


Figure 5.11: LIM hallway map realized by OccGridMapp turning back

The odometry is lost and the robot creates a second path on the map, overlapping the first one. This case is not always verified, i.e. the robot can turn back without overwriting the previous path. In any case, the odometry is easily lost.

A future work can be the implementation of a feedback system for the improvement of the odometry, like the data fusion of odometry, imu measurements and visual odometry.

Appendix A:

In this Appendix is presented the table of measurements performed to test the reliability of the 360° Lidar mounted on TurtleBot3. The robot is manually placed at 0.5 metres from a wall, in front of it, and 1000 sets of laser measurements are collected from the ROS topic /scan. From each set, only the measure corresponding to the laser ray casted at 0° is considered.

There is not only one right value. Due to the imprecision of the laser, the more frequent values, found in a set of data, can be considered "correct". In this case there are three correct values: 0.5, 0.500999987125 and 0.501999974251. Other values must be considered wrong measurements or failures of the sensor. The wrong measures are random values around the correct measurements. Instead, data 0.0 represent a failure in the functioning of sensor. There cannot exist a 0.0 measurement, because this value represents the point at the centre of the Lidar.

Laser detection of an obstacle located at 0.5 metres from the Burger				
0.5	0.5	0.500999987125	0.5	0.5
0.500999987125	0.5	0.0	0.5	0.5
0.5	0.500999987125	0.5	0.500999987125	0.5
0.500999987125	0.500999987125	0.5	0.5	0.500999987125
0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.500999987125	0.5	0.5
0.5	0.500999987125	0.5	0.5	0.5
0.5	0.5	0.5	0.500999987125	0.500999987125
0.5	0.500999987125	0.5	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.5	0.5	0.5
0.500999987125	0.5	0.5	0.500999987125	0.5
0.5	0.5	0.500999987125	0.5	0.500999987125
0.5	0.5	0.5	0.500999987125	0.5
0.5	0.0	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.500999987125
0.5	0.5	0.5	0.5	0.500999987125
0.500999987125	0.5	0.500999987125	0.5	0.500999987125
0.500999987125	0.5	0.500999987125	0.500999987125	0.5
0.500999987125	0.500999987125	0.5	0.500999987125	0.5

0.500999987125	0.5	0.500999987125	0.500999987125	0.5
0.5	0.5	0.500999987125	0.500999987125	0.5
0.5	0.5	0.500999987125	0.5	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.5	0.500999987125
0.500999987125	0.5	0.5	0.500999987125	0.500999987125
0.5	0.5	0.500999987125	0.500999987125	0.500999987125
0.5	0.500999987125	0.500999987125	0.5	0.5
0.5	0.500999987125	0.5	0.5	0.5
0.5	0.5	0.500999987125	0.5	0.5
0.5	0.5	0.500999987125	0.5	0.500999987125
0.5	0.5	0.5	0.500999987125	0.5
0.5	0.500999987125	0.5	0.5	0.5
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.5	0.5	0.5
0.5	0.500999987125	0.500999987125	0.500999987125	0.5
0.5	0.500999987125	0.5	0.500999987125	0.500999987125
0.5	0.500999987125	0.500999987125	0.5	0.5
0.5	0.5	0.500999987125	0.500999987125	0.5
0.0	0.5	0.5	0.500999987125	0.5
0.500999987125	0.500999987125	0.500999987125	0.5	0.500999987125
0.5	0.500999987125	0.5	0.5	0.500999987125
0.5	0.5	0.500999987125	0.500999987125	0.500999987125
0.5	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.501999974251	0.5	0.5	0.500999987125	0.5
0.5	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.5	0.5	0.5	0.500999987125
0.5	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.499000012875	0.5	0.5	0.5	0.500999987125
0.500999987125	0.500999987125	0.5	0.500999987125	0.500999987125
0.5	0.500999987125	0.5	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.5	0.5	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.5	0.500999987125
0.500999987125	0.5	0.500999987125	0.500999987125	0.5
0.500999987125	0.500999987125	0.5	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.5	0.500999987125	0.5
0.500999987125	0.5	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.5	0.500999987125	0.5	0.500999987125
0.5	0.500999987125	0.5	0.5	0.500999987125
0.500999987125	0.501999974251	0.5	0.500999987125	0.500999987125
0.5	0.500999987125	0.500999987125	0.500999987125	0.5
0.500999987125	0.5	0.500999987125	0.0	0.500999987125
0.5	0.500999987125	0.5	0.500999987125	0.5
0.500999987125	0.5	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.5	0.500999987125	0.5
0.500999987125	0.500999987125	0.5	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.5	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.5
0.5	0.500999987125	0.500999987125	0.5	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.5	0.5
0.500999987125	0.500999987125	0.500999987125	0.5	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.5	0.5	0.5	0.500999987125	0.500999987125

0.5	0.500999987125	0.5	0.500999987125	0.5
0.5	0.500999987125	0.500999987125	0.500999987125	0.5
0.500999987125	0.500999987125	0.500999987125	0.501999974251	0.5
0.500999987125	0.500999987125	0.500999987125	0.5	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.5	0.500999987125
0.0	0.5	0.5	0.500999987125	0.5
0.5	0.500999987125	0.5	0.500999987125	0.500999987125
0.500999987125	0.5	0.500999987125	0.5	0.5
0.500999987125	0.500999987125	0.500999987125	0.5	0.5
0.500999987125	0.500999987125	0.500999987125	0.5	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.5	0.500999987125	0.500999987125
0.500999987125	0.5	0.5	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.5	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.5
0.5	0.500999987125	0.500999987125	0.500999987125	0.5
0.5	0.500999987125	0.500999987125	0.5	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.5	0.5
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.5
0.500999987125	0.5	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.5	0.500999987125	0.500999987125	0.500999987125
0.5	0.500999987125	0.5	0.500999987125	0.500999987125
0.500999987125	0.5	0.500999987125	0.5	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.5	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.5	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.5	0.5	0.500999987125
0.500999987125	0.5	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.5	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.5	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.5	0.501999974251	0.5	0.500999987125
0.500999987125	0.5	0.5	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.5	0.5	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.5	0.500999987125	0.500999987125	0.500999987125
0.5	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.5	0.500999987125	0.500999987125	0.500999987125	0.5
0.500999987125	0.5	0.500999987125	0.500999987125	0.5
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.5	0.500999987125
0.500999987125	0.500999987125	0.5	0.500999987125	0.5
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.5	0.500999987125	0.500999987125	0.500999987125
0.5	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.5	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.5	0.501999974251	0.500999987125	0.500999987125	0.5
0.0	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.5
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.5

0.5	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.5	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.5
0.500999987125	0.500999987125	0.5	0.5	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.5
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.501999974251	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.5	0.500999987125
0.5	0.500999987125	0.5	0.500999987125	0.500999987125
0.500999987125	0.5	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.5	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.5	0.5
0.500999987125	0.500999987125	0.500999987125	0.5	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.0	0.500999987125	0.500999987125	0.500999987125	0.5
0.500999987125	0.500999987125	0.5	0.500999987125	0.5
0.5	0.5	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.5
0.500999987125	0.500999987125	0.5	0.500999987125	0.5
0.500999987125	0.500999987125	0.5	0.5	0.500999987125
0.500999987125	0.500999987125	0.5	0.5	0.5
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.501999974251
0.5	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.5	0.500999987125
0.500999987125	0.5	0.500999987125	0.500999987125	0.5
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.5	0.5	0.5	0.500999987125
0.500999987125	0.500999987125	0.5	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.5	0.5	0.500999987125	0.500999987125	0.5
0.5	0.500999987125	0.500999987125	0.500999987125	0.5
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.5	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.5	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.501999974251	0.500999987125	0.500999987125	0.500999987125	0.501999974251
0.500999987125	0.500999987125	0.500999987125	0.501999974251	0.500999987125
0.500999987125	0.500999987125	0.0	0.500999987125	0.500999987125
0.5	0.5	0.5	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.5	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.5	0.500999987125
0.501999974251	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.501999974251	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125

0.500999987125	0.500999987125	0.500999987125	0.5	0.500999987125
0.500999987125	0.500999987125	0.501999974251	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.5	0.500999987125
0.500999987125	0.500999987125	0.5	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.5	0.501999974251	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.501999974251	0.500999987125	0.500999987125	0.500999987125	0.501999974251
0.5	0.501999974251	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.5	0.501999974251
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.5	0.500999987125	0.5	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.500999987125
0.500999987125	0.500999987125	0.500999987125	0.500999987125	0.501999974251

Appendix B:

The code developed during the thesis work is reported here. It is a Python program that performs the Simultaneous Localization and Mapping (SLAM). The algorithm receives odometry and laser scan data as inputs. It returns a plotted grid map as output. The matrix representing the grid map is available is stored in the variable *grid_map*.

The localization relies on the odometry measurements coming from the wheel encoders. The build and the continuously update of the map are accomplished using data gathered from the Lidar. A function for the plotting of the map like an image has been implemented.

OccGridMapp.py

```
1   #!/usr/bin/env python
2   #Imports:
3   import time
4   import threading
5   import math
6   import numpy
7   from tf.transformations import euler_from_quaternion
8   import matplotlib.pyplot as plt
9
10  import rospy
11  from sensor_msgs.msg import LaserScan
12  from nav_msgs.msg import Odometry
13  from geometry_msgs.msg import Twist
14  from std_msgs.msg import String
15
16
17  #Start up node of the program:
18  rospy.init_node('my_slam')
19
20
```

```
21    #Set up parameters:
22    global grid_map, my_pose_x,my_pose_y,my_pose_theta,
      n_plots,plots, res,lo1,lo0,pz1m1,pz0m0, semaph
23    grid_map=numpy.zeros((5000,5000))
24    my_pose_x=[]
25    my_pose_y=[]
26    my_pose_theta=[]
27    n_plots=0
28    plots=[]
29    res=0.01
30    pm1z1=991/1000.0
31    pm0z0=(1000-6)/1000.0
32    lo1=math.log(pm1z1/(1-pm1z1))
33    lo0=math.log(pm0z0/(1-pm0z0))
34    semaph=0
35
36
37    #Function computing the yaw angle from quaternion:
38    def get_yaw_from_quaternion(quaternion):
39       quaternion_list=[quaternion.x, quaternion.y, quaternion.z,
      quaternion.w]
40       (roll,pitch,yaw)=euler_from_quaternion(quaternion_list)
41       return yaw
42
43
44    #Bresenham's line algorithm returning y with x increasing:
45    def yFromLine(y1,x1,y0,x0,x):
46       y=(y1-y0)/(x1-x0)*(x-x0)+y0
47       return y
48
49
50    #Bresenham's line algorithm returning x with y increasing:
51    def xFromLine(y1,x1,y0,x0,y):
52       x=(x1-x0)/(y1-y0)*(y-y0)+x0
53       return x
54
55
56    #Callback running when new LaserScan message published:
57    def callbackLs(msg):
58       global rngs_scan,semaph
59       rngs_scan=msg.ranges
```

```
60          semaph=1
61
62
63      #Callback running when new Odometry message published:
64      def callbackOd(msg):
65          global my_pose_x,my_pose_y,my_pose_theta
66          my_pose_x.append((-msg.pose.pose.position.y+10.000)/res)
67          my_pose_y.append((msg.pose.pose.position.x+10.000)/res)
68          my_pose_theta.append(math.degrees(get_yaw_from_quaternion(
        msg.pose.pose.orientation))+90)
69
70
71      #Callback running when new Velocity message published:
72      def callbackVl(msg):
73          global ang_vel
74          ang_vel=msg.angular.z
75
76
77      #MAP BELIEF UPDATE ALGORITHM:
78      def MapUpdater(rngs_data,act_pose):
79          global grid_map, semaph
80          for i in xrange(360):
81              #faulty measure
82              if rngs_data[i] == 0.0:
83                  continue
84              if (i+act_pose["theta"]>=-45 and i+act_pose["theta"]<45) or
        (i+act_pose["theta"]>=315 and i+act_pose["theta"]<405):
85                  #free space
86                  if rngs_data[i] > 3.5:
87                      cnt=1000
88                      ob_dist=350
89                      ind_x=int(round(ob_dist*math.cos(math.radians(
        i+act_pose["theta"]))+act_pose["x"]))
90                      ind_y=int(round(ob_dist*math.sin(math.radians(
        i+act_pose["theta"]))+act_pose["y"]))
91                      grid_map[ind_x,ind_y]-=lo0
92                  #obstacle detected
93                  else:
94                      ob_dist=round(rngs_data[i],3)/res
95                      ind_x=int(round(ob_dist*math.cos(math.radians(
        i+act_pose["theta"]))+act_pose["x"]))
```

```
96                  ind_y=int(round(ob_dist*math.sin(math.radians(
      i+act_pose["theta"]))+act_pose["y"]))
97                  grid_map[ind_x,ind_y]+=lo1
98                  cnt=ind_x-int(round(act_pose["x"]))
99              for x in xrange(int(round(act_pose["x"])),ind_x):
100                 y=int(round(yFromLine(act_pose["y"],act_pose["x"],
      ind_y,ind_x,x)))
101                 if cnt<2:
102                     grid_map[x,y]-=lo1
103                 else:
104                     grid_map[x,y]-=lo0
105                 cnt-=1
106         elif (i+act_pose["theta"]>=45 and i+act_pose["theta"]<135) or
      (i+act_pose["theta"]>=405 and i+act_pose["theta"]<495):
107             #free space
108             if rngs_data[i] > 3.5:
109                 cnt=1000
110                 ob_dist=350
111                 ind_x=int(round(ob_dist*math.cos(math.radians(
      i+act_pose["theta"]))+act_pose["x"]))
112                 ind_y=int(round(ob_dist*math.sin(math.radians(
      i+act_pose["theta"]))+act_pose["y"]))
113                 grid_map[ind_x,ind_y]-=lo0
114             #obstacle detected
115             else:
116                 ob_dist=round(rngs_data[i],3)/res
117                 ind_x=int(round(ob_dist*math.cos(math.radians(
      i+act_pose["theta"]))+act_pose["x"]))
118                 ind_y=int(round(ob_dist*math.sin(math.radians(
      i+act_pose["theta"]))+act_pose["y"]))
119                 grid_map[ind_x,ind_y]+=lo1
120                 cnt=ind_y-int(round(act_pose["y"]))
121             for y in xrange(int(round(act_pose["y"])),ind_y):
122                 x=int(round(xFromLine(act_pose["y"],act_pose["x"],
      ind_y,ind_x,y)))
123                 if cnt<2:
124                     grid_map[x,y]-=lo1
125                 else:
126                     grid_map[x,y]-=lo0
127                 cnt-=1
128         elif (i+act_pose["theta"]>=135 and i+act_pose["theta"]<225) or
```

```
        (i+act_pose["theta"]>=495 and i+act_pose["theta"]<585):
129             #free space
130             if rngs_data[i] > 3.5:
131                 cnt=1000
132                 ob_dist=350
133                 ind_x=int(round(ob_dist*math.cos(math.radians(
        i+act_pose["theta"]))+act_pose["x"]))
134                 ind_y=int(round(ob_dist*math.sin(math.radians(
        i+act_pose["theta"]))+act_pose["y"]))
135                 grid_map[ind_x,ind_y]-=lo0
136             #obstacle detected
137             else:
138                 ob_dist=round(rngs_data[i],3)/res
139                 ind_x=int(round(ob_dist*math.cos(math.radians(
        i+act_pose["theta"]))+act_pose["x"]))
140                 ind_y=int(round(ob_dist*math.sin(math.radians(
        i+act_pose["theta"]))+act_pose["y"]))
141                 grid_map[ind_x,ind_y]+=lo1
142                 cnt=0
143             for x in xrange(ind_x+1,int(round(act_pose["x"]))+1):
144                 y=int(round(yFromLine(act_pose["y"],act_pose["x"],
        ind_y,ind_x,x)))
145                 if cnt<2:
146                     grid_map[x,y]-=lo1
147                 else:
148                     grid_map[x,y]-=lo0
149                 cnt+=1
150         elif (i+act_pose["theta"]>=225 and i+act_pose["theta"]<315) or
        (i+act_pose["theta"]>=-90 and i+act_pose["theta"]<-45) or
        (i+act_pose["theta"]>=585 and i+act_pose["theta"]<=630):
151             #free space
152             if rngs_data[i] > 3.5:
153                 cnt=1000
154                 ob_dist=350
155                 ind_x=int(round(ob_dist*math.cos(math.radians(
        i+act_pose["theta"]))+act_pose["x"]))
156                 ind_y=int(round(ob_dist*math.sin(math.radians(
        i+act_pose["theta"]))+act_pose["y"]))
157                 grid_map[ind_x,ind_y]-=lo0
158             #obstacle detected
159             else:
```

```
160                 ob_dist=round(rngs_data[i],3)/res
161                 ind_x=int(round(ob_dist*math.cos(math.radians(
        i+act_pose["theta"]))+act_pose["x"]))
162                 ind_y=int(round(ob_dist*math.sin(math.radians(
        i+act_pose["theta"]))+act_pose["y"]))
163                 grid_map[ind_x,ind_y]+=lo1
164                 cnt=0
165               for y in xrange(ind_y+1,int(round(act_pose["y"]))+1):
166                 x=int(round(xFromLine(act_pose["y"],act_pose["x"],
        ind_y,ind_x,y)))
167                 if cnt<2:
168                   grid_map[x,y]-=lo1
169                 else:
170                   grid_map[x,y]-=lo0
171                 cnt+=1
172       pub.publish('TOOOPIIIIIC!!!')
173       semaph=0
174
175
176   #Thread class to continuously update map:
177   class MapUp(threading.Thread):
178       def __init__(self):
179           threading.Thread.__init__(self)
180       def run(self):
181           while True:
182               if ang_vel<=0.2 and ang_vel>=-0.2 and semaph==1:
183                   rngs_data=rngs_scan
184                   act_pose={"x":my_pose_x[-1], "y":my_pose_y[-1],
        "theta":my_pose_theta[-1]}
185                   MapUpdater(rngs_data,act_pose)
186
187
188   #Set up subscribers to ROS topics:
189   class Subscribers(threading.Thread):
190       def __init__(self):
191           threading.Thread.__init__(self)
192       def run(self):
193           subLs=rospy.Subscriber('/scan', LaserScan, callbackLs)
194           subOd=rospy.Subscriber('/odom', Odometry, callbackOd)
195           subVl=rospy.Subscriber('/cmd_vel', Twist, callbackVl)
196           rospy.spin()
```

```
197
198
199     #Map plotting function:
200     def PlotMap(mpl):
201         global n_plots,plots
202         now_pose={"x":my_pose_x, "y":my_pose_y}
203         axis_x_ob=[]
204         axis_y_ob=[]
205         plots.append(plt.figure(n_plots))
206         occup_threshold=3
207         for i in xrange(len(mpl)):
208             for j in xrange(len(mpl)):
209                 if mpl[i][j] > occup_threshold:
210                     axis_x_ob.append([i])
211                     axis_y_ob.append([j])
212         plt.plot(axis_x_ob,axis_y_ob,'k.')
213         plt.plot(now_pose["x"],now_pose["y"],'r-')
214         plots[-1].show()
215         n_plots+=1
216
217
218     #Main
219     if __name__=='__main__':
220         grid_map_plt=numpy.zeros((5000,5000))
221         global pub
222         pub=rospy.Publisher('/a_topic',String,queue_size=10)
223         sub=Subscribers()
224         sub.start()
225         time.sleep(1)
226         mpu=MapUp()
227         mpu.start()
228         while True:
229             print_map=raw_input("Do you want to plot the map? y/n ")
230             if print_map=='y':
231                 grid_map_plt=grid_map
232                 PlotMap(grid_map_plt)
```

Bibliography

[1] "*ROS*" [Online]. Available: http://wiki.ros.org/. [Accessed: 05-Dec-2018].

[2] "*ROS introduction*" [Online]. Available: https://husarion.com/tutorials/ros-tutorials/1-ros-introduction/. [Accessed: 05-Dec-2018].

[3] "*Programming for Robotics* - ROS", Online course, [Online]. Available: http://www.rsl.ethz.ch/education-students/lectures/ros.html. [Accessed: 05-Dec-2018].

[4] "*TurteBot3 – ROBOTIS e-Manual*" [Online]. Available: http://emanual.robotis.com/docs/en/platform/turtlebot3/overview/. [Accessed: 05-Dec-2018].

[5] "*Raspberry Pi*" [Online]. Available: https://en.wikipedia.org/wiki/Raspberry_Pi. [Accessed: 05-Dec-2018].

[6] "*Intel Atom T5700 @ 1.70GHz*" [Online]. Available: https://www.cpubenchmark.net/cpu.php?cpu=Intel+Atom+T5700+%40+1.70GHz&id=3293. [Accessed: 05-Dec-2018].

[7] "*Raspberry Pi Camera Module*" [Online]. Available: https://www.raspberrypi.org/documentation/hardware/camera/README.md. [Accessed: 05-Dec-2018].

[8] "*Bootable Ubuntu USB stick*" [Online]. Available: https://tutorials.ubuntu.com/tutorial/tutorial-create-a-usb-stick-on-windows#0. [Accessed: 05-Dec-2018].

[9] "*Intel Joule*" [Online]. Available: https://developer.ubuntu.com/core/get-started/intel-joule. [Accessed: 05-Dec-2018].

[10] "*Ubuntu install of ROS Kinetic*" [Online]. Available: http://wiki.ros.org/kinetic/Installation/Ubuntu. [Accessed: 05-Dec-2018].

[11] Sebastian Thrun, Dieter Fox and Wolfram Burgard, "*Probabilistic Robotics*", 1999-2000

[12] Roland Siegwart and Illah R. Nourbakhsh, "*Introduction to Autonomous Mobile Robots*", The MIT Press Cambridge, Massachusetts, 2004.

[13] Julien Diard, Pierre Bessière, and Emmanuel Mazer, "*A survey of probabilistic models, using the Bayesian Programming methodology as a unifying framework*", 2003.

[14] Simo Särkkä, "*Bayesian Filtering and Smoothing*", Cambridge University Press, 2013.

[15] "*The Python Standard Library*" [Online]. Available: https://docs.python.org/2/library/. [Accessed: 05-Dec-2018].

[16] "*Bresenham's line algorithm*." [Online]. Available: https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm. [Accessed: 05-Dec-2018].

[17] "*The Bresenham Line-Drawing Algorithm*." [Online]. Available: https://www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html. [Accessed: 05-Dec-2018].

[18] "*OpenSLAM: GMapping*." [Online]. Available: https://openslam-org.github.io/gmapping.html. [Accessed: 05-Dec-2018].

[19] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard, "*Improved Techniques for Grid Mapping with Rao-Blackwellized Particle Filters*", IEEE Transactions on Robotics, Volume 23, pages 34-46, 2007

[20] Carol Fairchild, Dr.Thomas L. Harman, "*ROS Robotics By Example*", Packt Publishing Ltd., Birmingham, 2017

[21] M. Sanjeev Arulampalam, Simon Maskell, Neil Gordon, and Tim Clapp, "*A Tutorial on Particle Filters for On-line Non-linear/Non-Gaussian Bayesian Tracking*", IEEE, 2002

[22] Zhe Chen, "*Bayesian Filtering: From Kalman Filters to Particle Filters, and Beyond*", A Journal of Theoretical and Applied Statistics, January 2003.