



UNIVERSITÀ DEGLI STUDI DI PADOVA

TESI DI LAUREA MAGISTRALE

NAVIGATION AND GRASPING
WITH A MOBILE MANIPULATOR:
FROM SIMULATION TO EXPERIMENTAL RESULTS

Laureando: IOVINO MATTEO
Matricola: 1128617

Relatore: Prof.ssa VALCHER MARIA ELENA

Tutore Industriale: PhD. Dr. FALCO PIETRO

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA DELL'AUTOMAZIONE

8 Ottobre 2018

ANNO ACCADEMICO 2017-2018

*To my family,
for they know what this meant for me
as I know what this meant for them.*

Abstract

Cobot is the name for collaborative robots. This kind of robot is intended to work in close contact with the human being and to collaborate, by increasing the production rate and by reducing the human onerous tasks, in terms of repetitiveness and precision. At the state of the art, Cobots are often fixed on a support platform, static in their workstation.

The aim of this thesis is, hence, to explore, test and validate navigation algorithms for a holonomic mobile robot and in a second moment, to study its behavior with a Cobot mounted on it, in a pick-move-place application.

To this purpose, the first part of the thesis addresses the mobile navigation, while the second part the mobile manipulation.

Concerning mobile robotics, in the first place, a theoretical background is given and the kinematic model of a holonomic robot is derived. Then, the problem of simultaneous localization and mapping (SLAM) is addressed, i.e. how the robot is able to build a map while localizing itself. Finally, a dedicated chapter will explain the algorithms responsible for exploration and navigation: planners, exploration of frontiers and Monte Carlo localization.

Once the necessary theoretical background has been given, these algorithms will be tested both in simulation and in practice on a real robot.

In the second part, some theoretical knowledge about manipulators is given and also the kinematic model of the Cobot is derived, together with the algorithm used for a collision free trajectory planning.

To conclude, the results of the complete task are shown, first of all in simulation and then on the real robotic system.

Acknowledgements

I would first like to thank my industrial tutors, the PhD Dr. Pietro Falco and the Eng. Jon Tjerngren. They entrusted me the accomplishment of a an ambitious project and supported me with all their competences and capabilities throughout the whole duration of the work. Additionally, I want to acknowledge the EU project ROSIN [1], which this master thesis was a part of. ROSIN is an European project that is part of the Research and Innovation programme Horizon 2020.

I would also like to thank the Mechanical Systems & Mechatronics team of the ABB Corporate Research Center in Västerås, and in particular the manager Jonas Larsson, for having accepted me in the team and for giving me all the equipment I asked for, included the robots I worked with. Moreover, I want to thank Gabriella Grefberg-Kvist, as well, for the help provided in the bureaucratic matters and others, which made my permanence in Västerås and in ABB very pleasant.

I now want to thank Hugo Bouvier-Lambert, who worked with me in achieving what showed in this thesis. By working alone I would not have been able to arrive so far.

I must express my very profound gratitude to my thesis advisor the Prof. Maria Elena Valcher of the Department of Information Engineering of the University of Padua. She read this work with a severe criticism, which made the thesis improve in fluency and clarity. Her support dates back to two years ago, because she is the responsible for the TIME project I carried out, a Double Degree program in collaboration with the Ecole Centrale of Nantes in France. Without her active help in facing all the difficulties we encountered, I would improbably have learned Robotics during the last year in Nantes and this thesis would have been about something else.

Equivalently, I must thank as well Dr. Luisa Bortolini, responsible for the TIME project in the International Office of the University of Padua. Her diplomacy and competences were a key factor during the two years in Nantes.

My last thanks go to my parents and relatives. The support and encouragement they provided me throughout all my years of study were fundamental for the path I took. Moreover, in the last two years, they made me feel as if I had never left home.

Contents

Abstract	i
Acknowledgements	iii
Contents	v
List of Figures	ix
1 Introduction	1
1.1 Definition of the problem	1
1.2 Content of the thesis	2
I Navigation	5
2 ABB RobotStudio	7
2.1 Introduction to ABB's RobotStudio	7
2.2 RobotStudio Smart Component	9
2.3 RobotStudio Add-In	11
2.3.1 TCP/IP Communication protocol	11
2.3.2 UDP/IP Communication protocol	11
2.3.3 Google Protocol Buffers	12
3 Mobile Robotics: theoretical background and kinematic modeling	15
3.1 Preliminary definitions and concepts	15
3.1.1 Wheeled Mobile Robots, robot pose, twist	15
3.1.2 Degrees of freedom, degrees of mobility	17
3.1.3 Holonomic, nonholonomic, redundant and omnidirectional robots	17
3.2 Types of wheels	17
3.2.1 The general wheel assembly	18
3.2.2 The fixed wheel	18
3.2.3 The Mecanum (Swedish) wheel	19
3.3 Kinematic model of a four mecanum wheeled mobile robot	20
4 Simultaneous Localization and Mapping	23
4.1 Theory of probability: concepts and notations used	23
4.2 Monte Carlo Localization	25
4.3 Occupancy Grid Mapping	28
4.4 The SLAM algorithm	32
4.4.1 Feature-based FastSLAM	33
4.4.2 Grid-based FastSLAM	34
5 Navigation	37
5.1 Path Planning	37
5.1.1 The global planner	38
5.1.2 The local planner	41
5.1.3 Frontier Exploration	52
5.2 Navigating a given map	54
5.2.1 Odometry Motion Model	55

5.2.2	Augmented MCL	57
6	Simulating and Testing Ridgeback with ROS	59
6.1	Robot Operating System	59
6.2	Simulation set up	60
6.2.1	The simulation environment	61
6.3	ROS Navigation Stack	62
6.3.1	Robot setup	62
6.3.2	Navigation Stack setup	63
6.4	Parametrization	63
6.4.1	Move Base	63
6.4.2	The Global Planner	64
6.4.3	The Local Planner	64
6.4.4	Cost maps	66
6.4.5	Gmapping	67
6.4.6	Frontier Exploration	68
6.5	Launching the Exploration	69
6.6	Navigation of the built map	72
6.7	Accuracy and Robustness tests	74
6.8	Conclusions and encountered problems	82
II	Mobile Manipulation	85
7	Introduction to Mobile Manipulation	87
8	Manipulators: theoretical background and kinematic modeling	91
8.1	Homogeneous Transformation	91
8.2	Kinematic Chains	92
8.3	The Denavit-Hartenberg convention	95
8.4	Kinematic model of the IRB-14000 robot	97
9	Trajectory Planning for Manipulators	99
9.1	Probabilistic Planning	99
9.1.1	PRM Method	99
9.1.2	RRT Method	101
9.1.3	RRT-Connect	102
10	Simulating and Testing YuMi with ROS	105
10.1	ROS MoveIt!	106
10.2	Simulating manipulation tasks in MoveIt!	107
10.3	Combining Navigation and Manipulation	109
10.3.1	Offset Planner for the Ridgeback	109
10.3.2	The Station Traveling Node	111
10.4	Conclusions and future work	112
	Appendices	117
A	ABB IRB 14000 YuMi Datasheet	119
B	Clearpath Ridgeback Datasheet	123
C	Hokuyo UST-10LX Laser Sensor Datasheet	124

D	<i>Add-In</i> source codes	131
D.1	C# code of the Main Class	131
D.2	C# code of the TCP Server	137
D.3	C# code of the UDP Client	139
D.4	XML code of the Ribbon	140
D.5	XML code of the Command Bar Control	141
D.6	XML code of the Command Help Text	141
E	Basic Filters	143
E.1	Bayes Filter	143
E.2	Gaussian Filters	143
E.2.1	The Kalman Filter	144
E.2.2	The Extended Kalman Filter	145
E.3	The Particle Filter	147
F	Ridgeback Simulation source codes	149
F.1	Planners	149
F.1.1	Planner_Mapping.yaml	149
F.1.2	Planner_Navigation.yaml	150
F.2	Costmap_Common.yaml	151
F.3	Gmapping.yaml	152
F.4	Frontier Exploration.yaml	153
F.5	Amcl.yaml	153
F.6	Exploration.launch	155
F.7	Navigation.launch	155
F.8	Gazebo.launch	156
	Bibliography	157

List of Figures

1.1	The two robots used in the project.	1
2.1	ABB RobotStudio: overall view of a <i>Solution</i>	7
2.2	ABB RobotStudio: overall view of a <i>Solution</i> in <i>Simulation</i> mode.	8
2.3	The <i>Ridgeback</i> mobile platform model.	9
2.4	The <i>Smart Component</i> Design view.	10
2.5	The scheme of the communication environment.	11
2.6	The <i>Add-In</i> buttons.	12
2.7	The Help Text of the <i>Add-In</i> buttons.	13
2.8	The Settings window of the <i>Add-In</i>	13
3.1	Scheme of the ideal wheel-ground contact (source: [30]).	16
3.2	Parametrization of the robot pose and the frame of the wheel (source: [30]).	16
3.3	Parametrization of the general wheel assembly (source: [30]).	18
3.4	Parametrization of the fixed wheel (source: [30]).	18
3.5	Parametrization of the mecanum wheel.	19
3.6	Wheels' command inputs vs. robot displacement.	19
3.7	mWMR parametrization.	20
3.8	Parameter table of a 4 mWMR.	20
4.1	Monte Carlo particle filter behavior in localizing the robot (source: [34]).	27
4.2	Scheme of the parametrization of a laser range finder.	29
4.3	Scheme of the assignment of the occupancy value to a cell.	30
4.4	Map obtained with Occupancy Grid Mapping (right) from raw sensor data (left) based on odometry only (source: [34]).	31
4.5	Feature-based FastSLAM: particle set and algorithm (source: [34]).	33
4.6	In grid-based FastSLAM each particle builds a map (source: [34]).	34
5.1	Pixel connectivity scheme.	38
5.2	Breadth-first expansion in Dijkstra algorithm.	39
5.3	Best-first expansion in A* algorithm.	40
5.4	Dynamic Window Approach.	42
5.5	Parameters of the cost function for DWA with the detail for goal_dist.	43
5.6	Inflation scheme to compute the occupancy distance cost.	43
5.7	Scheme of the operating principle of the DWA Local Planner.	44
5.8	Operating principle of bubbles-based Elastic Bands.	46
5.9	Bubbles behavior in consequence of a moving obstacle.	47
5.10	A case of failure for Elastic Bands: the circle approximating the robot is larger than the door width.	47
5.11	Two half-ellipses behavior for the Holonomic-DWA.	48
5.12	New goal distance for the Holonomic-DWA and the “ <i>local minima</i> ” case.	49
5.13	Scheme of the criteria for grouping frontier regions (dashed line) in frontiers (black square).	53
5.14	Cases when the frontier exploration fails.	53
5.15	The definitive behavior of the frontier exploration algorithm.	54
5.16	Odometry model.	55
5.17	Uncertainties in the robot pose due to motion noise.	56
5.18	Comparison between Augmented and standard MCL (source: [34]).	58

6.1	ROS logo.	59
6.2	ROS operating principle scheme.	60
6.3	Gazebo and RViz logos.	61
6.4	ROS Navigation Stack overview.	62
6.5	Evolution of the cost function of the DWA local planner.	65
6.6	Initialization of the simulation environment: Gazebo (left) and RViz (right).	69
6.7	The three layered cost maps of the CobotLab.	69
6.8	Three instants of the exploration task of the CobotLab in simulation.	70
6.9	Resulting map after SLAM gmapping task.	70
6.10	CobotLab in ABB Corporate Research Center.	71
6.11	Three views of the Ridgeback robot.	71
6.12	Exploration of the real CobotLab.	71
6.13	Simulated exploration task of a two rooms environment.	72
6.14	Convergence of the AMCL algorithm in estimating the robot pose.	73
6.15	AMCL algorithm fails in estimating the robot pose.	73
6.16	Chosen features for the map accuracy test.	74
6.17	Graphs for the three measures to evaluate the map accuracy.	75
6.18	Resume of the measurements for the map accuracy.	75
6.19	Scheme of the measures for the Movement Accuracy Test.	76
6.20	Scheme of the measures for the Goal Accuracy Test.	77
6.21	Results of the Goal Accuracy test.	79
6.22	Resume of the measurements for the map accuracy.	79
6.23	Percentage of success in the localization task.	80
6.24	Characteristics of the two laptops used for the simulation.	84
7.1	Example of Collaborative Mobile Manipulators.	89
8.1	Frame transformations in the representation of a point.	91
8.2	Schematic representation of an open kinematic chain.	93
8.3	Schematic representation of a closed kinematic chain.	93
8.4	Schematic representation a <i>prismatic</i> (left) and a <i>revolute</i> (right) joint.	94
8.5	Parametrization of the classic Denavit-Hartenberg convention.	95
8.6	Denavit-Hartenberg parameters table.	96
8.7	DH table for the arm of the YuMi robot.	97
8.7	The links, joints and main frames of the YuMi robot.	98
9.1	Scheme of the operating principle of the PRM Method for path planning.	100
9.2	Scheme of the operating principle of the RRT-Connect Method for path planning.	102
10.1	Robot Web Services library.	105
10.2	URDF model of the mobile manipulator Ridgeback-YuMi and its collision model (right).	106
10.3	MoveIt! logo.	106
10.4	MoveIt! system architecture.	106
10.5	Planning request for the robot arms in MoveIt!.	107
10.6	Main phases of the simulated pick&place task.	108
10.7	Main frames of the kinematic chain from the Ridgeback base to the YuMi end-effector.	109
10.8	Scheme of the robot traveling within workstations to accomplish the pick&place task.	111
10.9	Three views of the real mobile manipulator.	112

10.10	Standard gripper (on the left) compared to the special ones used (on the right).	113
10.11	Scheme of the final communication setup of the mobile manipulator.	113
10.12	Scheme of a fixed workstation with features to retrieve the task frame.	114
E.1	Errors due to linearization (source: [34]).	146
E.2	Samples of a Gaussian through a nonlinear function (source: [34]).	147

List of Algorithms and Source Codes

2.1	C# Smart Component	10
2.2	Google Protobuf for CmdVel	12
2.3	Google Protobuf for RobotPose	12
4.1	MCL algorithm, pseudo code	26
4.2	Occupancy grid algorithm, pseudo code	29
4.3	Inverse sensor model function, pseudo code	30
4.4	Grid-based FastSLAM algorithm, pseudo code	34
5.1	Dijkstra algorithm, pseudo code	39
5.2	A* algorithm, pseudo code	40
5.3	DWA Local Planner, pseudo code	45
5.4	Revisited-DWA, pseudo code	50
5.4	Revisited-DWA, pseudo code (<i>continued</i>)	51
5.5	isFrontierCell, pseudo code	54
5.6	Odometry Motion Model Sample, pseudo code	56
5.7	Odometry Motion Model Sample with Map, pseudo code	57
5.8	Augmented MCL algorithm, pseudo code	58
6.1	Move Base parametrization	63
6.2	Global Planner parametrization	64
6.3	Local Planner parametrization (exploration case)	64
6.4	Common Cost Map parametrization	66
6.5	Global Cost Map parametrization	66
6.6	Local Cost Map parametrization	66
6.7	Gmapping parametrization	67
6.8	Frontier Exploration parametrization	68
6.9	Augmented MCL parametrization	72
6.10	Local Planner parametrization for Goal Accuracy Test	78
9.1	PRM algorithm: learning phase, pseudo code	100
9.2	PRM algorithm: query phase, pseudo code	101
9.3	RRT algorithm, pseudo code	102
9.4	RRT-Connect algorithm, pseudo code	103
10.1	Offset Planner, pseudo code	110
E.1	Bayes Filter algorithm, pseudo code	143
E.2	Kalman Filter algorithm, pseudo code	144
E.3	Extended Kalman Filter algorithm, pseudo code	146
E.4	Particle Filter algorithm, pseudo code	148

Introduction

This aim of this first chapter is to introduce the reader to the thesis work done at the ABB Corporate Research Center in Västerås, Sweden, under the leading guide and technical supervision of the Eng. Jon Tjerngren and the PhD Dr. Pietro Falco. The content of the thesis will be presented, chapter by chapter, starting from the definition of the problem and the aim of the project.

1.1 Definition of the problem

Nowadays it is quite well-known that coffee is the fuel of an engineer. It is also well-known that engineers are capable of working within short time constraints and nothing makes them more efficient than dealing with last minute tasks.

In light of that, why loosing time to go and take a coffee when one can prototype a robot capable of doing it?

This project was born to try to answer this question, and even though ABB is the most advanced producer and designer of Manipulator Robots, the Group has never pushed the research in autonomous navigation that far.

The main idea is hence to install a suitable manipulator robot on a mobile platform, capable of navigating autonomously in a room, and also from one room to another, and driving the manipulator to the most suitable position to make it accomplish the picking task.

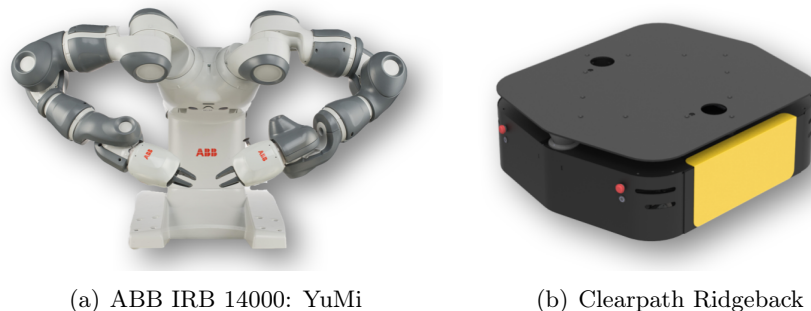


Figure 1.1: The two robots used in the project.

The manipulator chosen for this task is not a common manipulator, but a so called *collaborative robot*, namely a robot designed to work in close contact with the human operator, helping him/her in performing whatsoever task. The specific robot is the ABB IRB-14000, also known as YuMi (which stands for “you and me”, Figure 1.1a) [12]. Even though YuMi will be better introduced and detailed in due course, its datasheet can be found in Appendix A.

The mobile platform is the Clearpath Ridgeback (Figure 1.1b). This mobile robot has a sufficient payload to carry the YuMi and it is endowed with four Mecanum wheels that allow it to move in each direction [13]. This robot is responsible for building a digital map of the environment it moves in, and for the localization task, as detailed in Chapters 4 and 5. The datasheet of the Ridgeback is reported in Appendix B.

Robotics is a branch of engineering, the result of the synergy of multiple disciplines, such as mechanical, control and electrical engineering and computer science, to cite some. For reasons that range from personal skills and competences, to the time available to accomplish this work, the defined problem will only be faced from a computer science point of view, i.e. by describing the algorithms used to realize the various tasks. However, for the two robots, the kinematic model will be derived, and for the algorithms, a mathematical model will be provided.

1.2 Content of the thesis

This thesis is organized in a way that follows step by step the work done. However, to make it suitable also for those who are not directly involved in robotics, some theoretical definitions and formalizations are provided as well, when necessary. Therefore, the thesis is organized so that the necessary theoretical background is provided before presenting its implementation both in simulations and in practice on the real robot. Moreover, the thesis is split in two parts: the first one is dedicated to the mobile platform Ridgeback, while the second one to the manipulator robot YuMi and its interaction with the mobile platform.

In Chapter 2, the first step of the project will be presented, namely the attempt of establishing a connection between the Robot Operating System (ROS), the most used tool to communicate and operate with robots, and ABB RobotStudio, the best program to simulate ABB robots.

In Chapter 3, the most important concepts about mobile robotics will be defined and the kinematic model of a 4 Mecanum Wheeled Mobile Robot, a class of robots the Ridgeback is part of, will be derived. Also, Mecanum Wheels will be introduced, as they are a particular and uncommon type of wheel.

Chapters 4 and 5 will be dedicated to the real core of mobile robotics: the algorithms that allow the robot to build and navigate a map, together with the localization. Therefore, in Chapter 4, the so called *Simultaneous Localization and Mapping* algorithms will be discussed. In Chapter 5, instead, the algorithms that deal with path planning and localization in a built map will be presented.

In Chapter 6, the work done on the Ridgeback, using ROS, will be detailed. Hence, ROS will be introduced, as well, together with a description of how the theory of the two previous chapters is used. A first part of the work, namely the analysis of the ROS packages that better suit the robot, was done in simulation. Once the obtained results were satisfying, the final choice for the packages was also tested on the real robot, and a final parametrization set was found. In this chapter, the behavior of the robot, while performing navigation and map building tasks, will be discussed in detail. Then, a test protocol for the mobile platform will be designed, to attempt to measure the accuracy and the robustness of the various tasks. Finally, the chapter will end with a description of the problems that have been faced and how they were solved. This chapter will close the first part of the thesis.

The second part of the thesis will be dedicated to the mobile manipulation.

In Chapter 7, mobile manipulators will be defined and an attempt to state their usage in industrial environments will be made. To this extent, a set of components with which the mobile manipulator should be equipped will be described.

Chapter 8 will be entirely dedicated to robot manipulators. In the first part of the chapter, the homogeneous transformation will be introduced as a tool to derive the kinematic model of a robot arm. Finally, the kinematic model of the YuMi robot will be derived.

In Chapter 9, the main algorithms used for trajectory planning for manipulators will be presented and detailed.

Chapter 10 will be the last chapter of the thesis. Its first part will be dedicated to the implementation of the theory of the two previous chapter, to accomplish a pick&place task with YuMi, in simulation. The trajectory planning for the manipulator is realized with a ROS plug-in called MoveIt!, so it will be introduced as well. Finally, the complete mobile manipulator robot obtained by mounting YuMi on the Ridgeback will be described, together with the accomplishment of a complete task involving both navigation and simulation. The chapter will end with the last conclusions and critics of the work, together with some proposal of how to continue the mobile manipulator project.

Part I

Navigation

ABB RobotStudio

The aim of this chapter is to present the work that has been done with the ABB's RobotStudio software, whose library can be found in the RobotStudio Developer Center [16].

2.1 Introduction to ABB's RobotStudio

ABB's RobotStudio is a powerful off-line programming software that allows to simulate ABB's Manipulator Robots by letting one perform tasks such as training, programming, and optimization without disturbing production. RobotStudio is built on the ABB Virtual Controller, a virtual simulation of the real controller that is used on the real robots. This allows to perform very realistic simulations, because the same code and configurations that are used in simulation can be copied in the real robotic system [14].

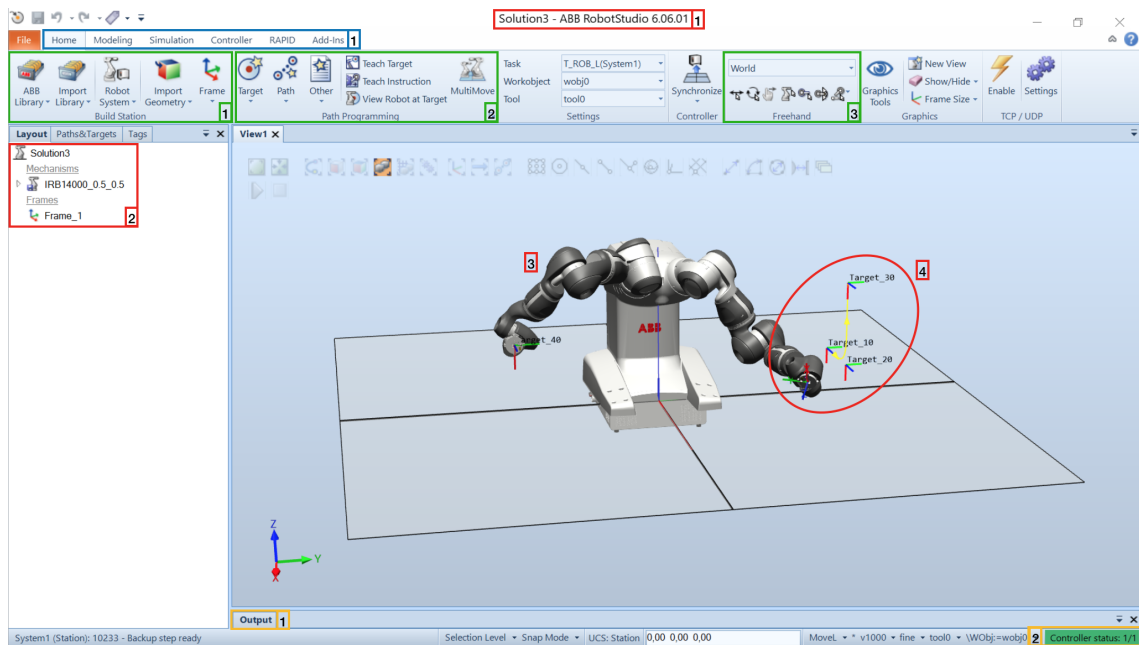


Figure 2.1: ABB RobotStudio: overall view of a *Solution*.

Figure 2.1 is intended to introduce this software to the reader, in order to make what follows more understandable.

Once RobotStudio is installed, the first thing one needs to do is to create his/her own project, which is called therefore a *Solution* (1). In a *Solution* one can set up his/her own work *Station*, by creating or importing every kind of ABB Robot or solid shape, to simulate walls, cages, obstacles, lifters, and so on (1).

When all the modeled elements are placed in the desired place in the *Station*, a useful menu allows the user to navigate through each element (2).

RobotStudio provides a library (*ABB Library*, 1) with all the ABB Robots already modeled and therefore ready to use (3). Each robot can be imported with its own controller, which is fundamental to make the robot perform a task.

In the *Home* menu (1) one can hence make the robot move, either by defining multiple

targets and a path to unite them (2, 4), or by moving the arms in the so called *Freehand* mode (3).

In the latter case, one can modify the arm(s) configuration by selecting an end effector (the last joint of the robot, loosely speaking: its “hand”) position (namely a Cartesian position, which means that it is identified by three translation coordinates $[x, y, z]$ and three rotation coordinates $[\alpha, \beta, \gamma]$ with respect to the Origin of a reference frame), or by acting directly on the arms by changing the joint values. The robot arm configuration will be better detailed further in this thesis.

Finally, in the bottom bar there is the *Logger Outputs*, where debug messages and simulation outcomes are displayed (1), and the *Controller Status*, which indicates the availability of the robot controller (2), (see Figure 2.1).

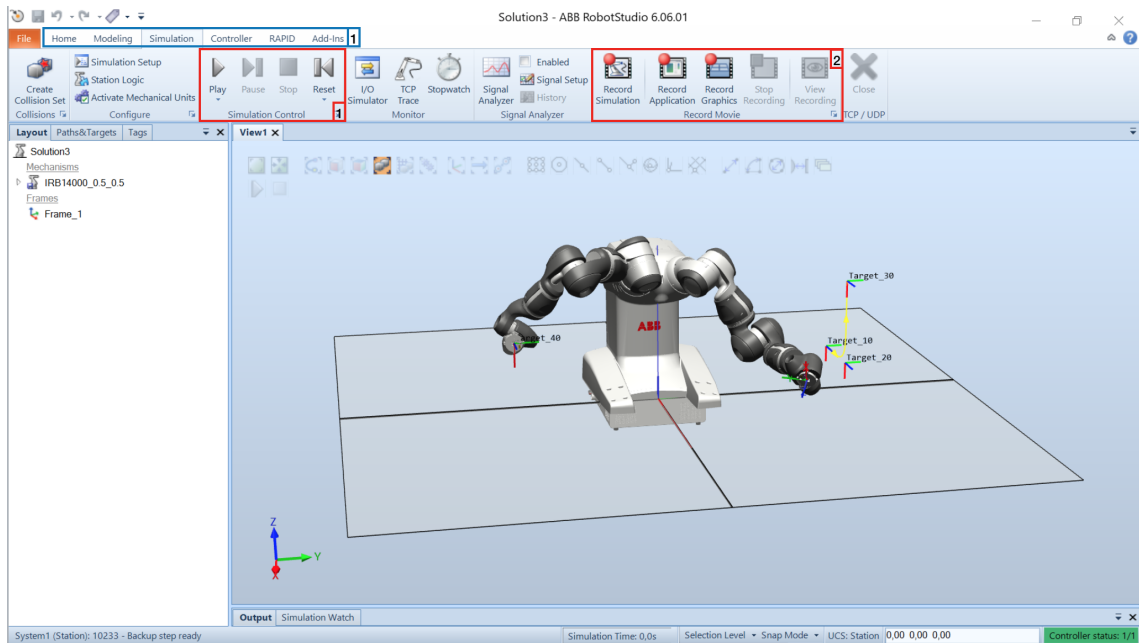


Figure 2.2: ABB RobotStudio: overall view of a *Solution* in *Simulation* mode.

When the *Station* is ready, the *Simulation* mode (Figure 2.2, 1) allows to watch the robot performing the task given through the *Simulation Control* buttons (1). Finally, if the outcome is satisfying, it is also possible to record the video of the simulation (2).

The user can interact with this software in three different ways:

1. by using the buttons, as showed above;
2. by programming the robot through a *RAPID* program (*RAPID* menu in (1), since presenting this programming language is not the purpose of this thesis, the User Manual can be found in [15]);
3. by changing some features or adding some plug-in, using the programming language *C#* (written in Windows Visual Studio, given that also RobotStudio runs under Windows).

2.2 RobotStudio Smart Component

According to the RobotStudio Developer Center, a *Smart Component* is a RobotStudio object with behavior implemented by some custom code and/or by the aggregation of other default components. Combining more Smart Components is also possible. A Smart Component is represented by the class `SmartComponent` [18][19].

Sometimes, in setting up a simulation environment, it is necessary to create such a component, in order to build unique parts with a desired behavior and interaction with the other elements of the *Station*.

This has been the case, in the first part of the work. Since the manipulator already existed in the *ABB Library*, the mobile platform had to be created.

The mobile platform was intended to simulate the real Ridgeback platform, which means that it had to:

1. receive some velocity command inputs (namely two linear velocities $[\dot{x}, \dot{y}]$ and the steering velocity $\dot{\theta}$);
2. move according to these command inputs;
3. return its position, (namely the pose $[x, y, \theta]^T$).

As a first step, the CAD model of each piece of the platform was imported and assembled, in order to obtain the model in Figure 2.3.

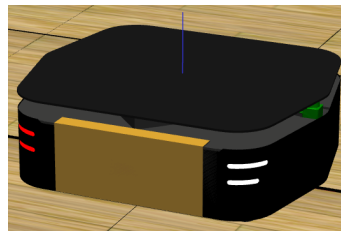


Figure 2.3: The *Ridgeback* mobile platform model.

As a second step, the *Smart Component* was created and the model included in it. The *Compose* window opens (Figure 2.4, 1) to let the user personalize the *Smart Component* with different kinds of tools, such as sensors and movers (that act like motors), or physical properties and behaviors, or also logical operators.

Then, it is possible to personalize further these elements by switching to the *Design* window (2).

This window shows all the characteristics and properties of each *Smart Component*. It shows also that the *Smart Component* behavior can be modified by acting on the *I/O Signals* and on the *Properties*.

It is essential to understand that in each *Smart Component*, the *Properties* can communicate with, and hence modify, only other *Properties* and the same applies to *I/O Signals*. For instance, the Ridgeback platform *Smart Component* that has been created is provided with one mover for each velocity input, but since the velocity is a *Property*, each mover requires a converter to translate an *Input Signal* into a *Property*. For example, the *Smart Component* model of the Ridgeback is endowed with three movers (one for each linear velocity along x and y and another for the angular velocity). Since the velocity is a *Property*, to modify it (for example to change its value from 0 to a chosen one) with an *I/O Signal* it is necessary to use a proper converter, which belongs to the logical operators. According to the needs specified above, the platform is also provided with a position sensor for the localization.

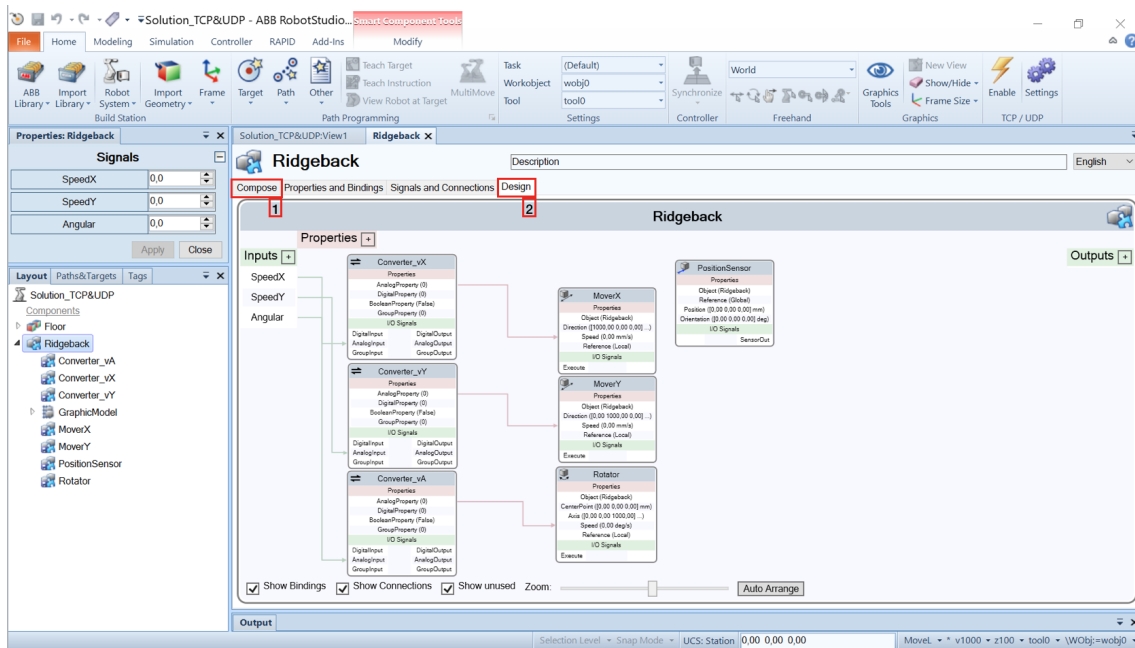


Figure 2.4: The *Smart Component* Design view.

Once the platform is equipped with all the desired components, it is still to be defined how an external input can modify the *Smart Component* behavior: only a C# program can deal with it. The snippet of code in Listing 2.1 is an extract of the complete code (entirely reported in Appendix D.1) for the *Add-In* presented in the next section. The code shows how to get access to the *Smart Component* and how to deal with the *Properties* and the *I/O Signals*.

To conclude, now that the *Smart Component* is ready, a new tool is required to ensure a wireless communication between computers and to allow the robot to receive the command and to send the position: the RobotStudio *Add-In*, which is the subject of the following section.

Listing 2.1: C# Smart Component

```

// Station
private static Station station;
// Robot platform
private static SmartComponent robot;
5 // Sensor
private static SmartComponent positionSensor;

robot = station.GraphicComponents["Ridgeback"] as SmartComponent;
// senso assignment
10 positionSensor = robot.GraphicComponents["PositionSensor"] as SmartComponent;
// for each simulation tick check for another command
Simulator.Tick += new EventHandler(tickEvent);

static void tickEvent(object sender, EventArgs e)
15 {
    station = Project.ActiveProject as Station;
    // ask for a velocity command
    CmdVel cmdVel = TCPCom.TCPGetCmdVel();

    // updates the smart component with the received command
    robot.IOSignals["SpeedX"].Value = cmdVel.LinearX;
    robot.IOSignals["SpeedY"].Value = cmdVel.LinearY;
    // the angular velocity is sent in radiants/s but in RobotStudio is in degrees/s
    robot.IOSignals["Angular"].Value = cmdVel.Angular * 180 / Math.PI;

    // reading of the position from the sensor
    Vector3 position = Vector3.Parse(positionSensor.Properties["Position"].Value.
    ToString());

```



```

30 Vector3 orientation = Vector3.Parse(positionSensor.Properties["Orientation"].Value.
    ToString());
    // state vector to send (the orientation is sent in deg/s)
    RobotPose state = new RobotPose
    {
        X = position.x,
        Y = position.y,
        Theta = orientation.z * Math.PI / 180
35 };
    }

```

2.3 RobotStudio Add-In

A RobotStudio *Add-In* is a plug-in that allows to introduce custom features not provided in the standard version of the RobotStudio software.

An *Add-In* comes in the form of one or multiple buttons that can be added to the main bar, called **Ribbon**, also modifiable by changing names, icons and help texts [20].

In the case of the robot *SmartComponent* the *Add-In* implemented a way to allow wireless communication between a first computer, where the robot is simulated in ROS (the Robot Operating System which is the subject of the Chapter 6), and a second one in which the RobotStudio simulation is running (Figure 2.5).

This communication is ensured by a TCP/IP protocol, when the message is sent from ROS to RobotStudio, and by a UDP/IP protocol in the other way around, while the message encoding is handled by Google Protocol Buffers. The three of them are briefly introduced, before presenting the *Add-In* itself.

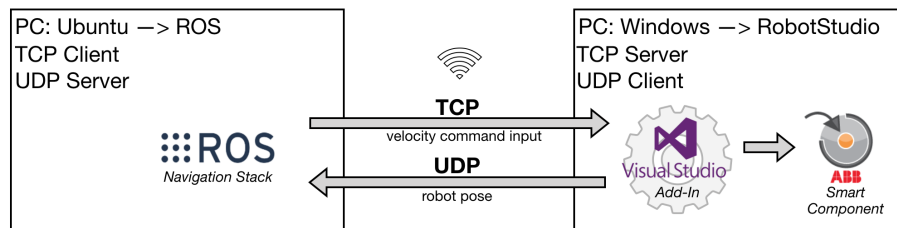


Figure 2.5: The scheme of the communication environment.

2.3.1 TCP/IP Communication protocol

The Transmission Control Protocol (TCP) is one of the main protocols of the Internet Protocol (IP) suite. TCP allows to send streams of octets, i.e. bytes, with the property of being reliable, ordered, and error-checked, between applications running on hosts communicating via an IP network [22].

Thanks to its reliability, this kind of communication protocol has been selected to ensure the transmission from ROS to RobotStudio. Indeed in controlling a robot, the command input needs to be transmitted properly and error-less.

2.3.2 UDP/IP Communication protocol

In computer networking, the User Datagram Protocol (UDP) is another IP suite protocol. Through UDP computer applications can send messages, referred to as datagrams, to other hosts on IP network. Unlike TCP, in UDP there is no guarantee of delivery, ordering, or duplicate protection [25].

The UDP was the protocol chosen for the task because it is faster than TCP and if a pose message is not delivered, the behavior of the robot is not compromised.

2.3.3 Google Protocol Buffers

Protocol buffers are Google’s language-neutral and platform-neutral libraries for serializing structured data. Once the data structure is defined, a special automatically generated source code allows to easily use the structured data for sending/receiving applications among different programming languages [27].

The specification of the structure information to serialize is defined by protocol buffer message types in `.proto` files. Each protocol buffer message is a small logical record of information, containing a series of name-value pairs [28].

According to the Google Protocol Buffer Language Guide [29], two types of protocol buffer messages have been defined, one for the velocity command input (`CmdVel` in `protobuf 2.2`) transmitted via TCP, and the other for the robot pose (`RobotPose` in `protobuf 2.3`) transmitted via UDP.

Listing 2.2: Google Protobuf for CmdVel

```
syntax = "proto3";

message CmdVel {
  double linear_x = 1;
  double linear_y = 2;
  double angular = 3;
}
```

Listing 2.3: Google Protobuf for RobotPose

```
syntax = "proto3";

message Robotpose {
  double x = 1;
  double y = 2;
  double theta = 3;
}
```

Without entering into the details, once Google Protobuf is installed, it provides an executable file that accepts `.proto` files and generates the corresponding `.cs` (the extension of C#). In the `.cs` file, the `CmdVel` and `RobotPose` classes are defined, together with the constructor and some standard methods (for example: `get()`, `toString()`).

Finally, these are the classes that are used in the codes for the *Add-In* (Appendix D).

Concerning the *Add-In*, the intention is not to bore the reader with a detailed analysis of the code itself, which is entirely reported in Appendix D, but to focus the dissertation on the graphical outcomes that are far more tangible and meaningful. Moreover, a code analysis will be performed later in the thesis, when the algorithms used for the ROS simulation will be presented.

The *Add-In* consists of three buttons, two of them placed in the Ribbon of the *Home* bar (already visible in Figure 2.1) and the other one placed in the Ribbon of the *Simulation* bar (already visible in Figure 2.2): the first button has the shape of a lightning and it enables the TCP/UDP communication, the second one has the shape of a gear and allows the user to change some parameters, and the last one has the shape of an “X” and stops the communication.

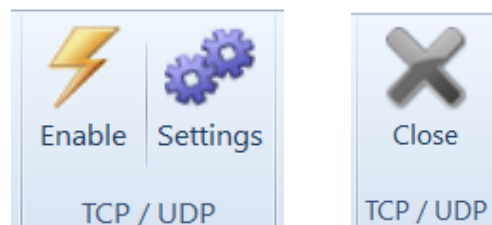


Figure 2.6: The *Add-In* buttons.

All the buttons belong to the same group that is “creatively” called “*TCP/UDP*” (Figure 2.6). The positioning of the buttons can be done in the `Ribbon.xml` file that is entirely reported in the Appendix D.4. On the other hand, the behavior of each button, coded in the

Main class (Appendix D.1, by the methods `RegisterCommand`, `button_UpdateCommandUI`, `button_ExecuteCommand`), becomes effective only once the buttons have been registered in the `CommandBarButton.xml` file (Appendix D.5).

By positioning the mouse on each button, one makes the help text come out. The help text can be personalized in the `CommandHelpTextButton.xml`, the choice made is showed in Figure 2.7.

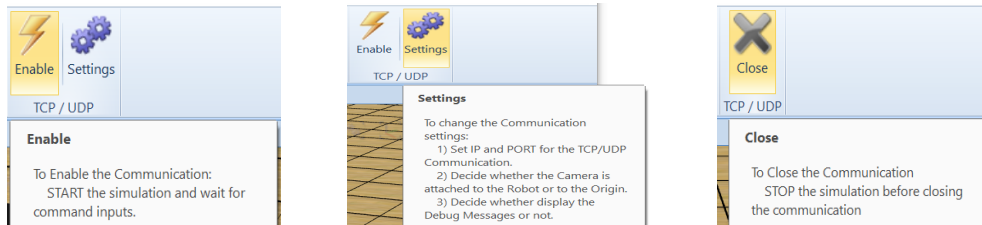


Figure 2.7: The Help Text of the *Add-In* buttons.

Finally, when selecting the Settings button, a personalized window pops up (Figure 2.8), and allows the user to manually set the following features:

- the **Server IP** and the **Port Number** of the TCP Server;
- the **Server IP** and the **Port Number** of the UDP Server;
- the **Camera** behavior (see the corresponding class at [21]): if the option is checked, the *Camera* will be fixed at the origin of the axes, if not it will follow the platform;
- the displaying of **Debug** messages in the corresponding bar. In case this option is checked, information concerning the status of the connection (IP and Port Number) and the details of the messages exchanged (the command received and the pose sent) will be displayed in the *Output* bar.

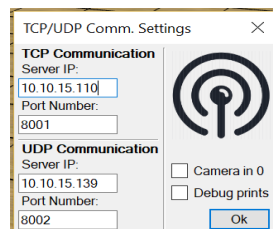


Figure 2.8: The Settings window of the *Add-In*.

Again, the corresponding code is in the `Main` class.

This kind of work has been carried on both with RobotStudio and ROS. The ROS part was developed by a colleague and, hence, it is not presented here.

The work done in RobotStudio was intended to explore the possibilities of the *Smart Component* and to what extent it is possible to communicate with ROS for robot simulations. The result of this investigation is that, at the state of the art, the functionalities available in the *Smart Component* are not sufficient to cover all the complex tasks that intervene in navigation. This is because it has no sensors capable of retrieving informations about the environment, as required by both localization and mapping.

Concerning the *Add-In*, it works as intended and in general it seems to be quite a powerful concept, as one can extend the functions already provided in Robot Studio to achieve special targets.

Mobile Robotics: theoretical background and kinematic modeling

The main goal of this chapter is to provide the readers with some preliminary background about mobile robotics, in order to help a better understanding of the content of the next chapters. Moreover, the platform used for this project is an unconventional mobile robot, a case that is usually not detailed in textbooks.

Some general definitions will be given and finally the mathematical model for the Clearpath Ridgeback robot will be derived.

Most of the content of this chapter is taken from the notes of the course of Prof. Gaëtan Garcia, *Wheeled Mobile Robotics* [30], except for the sections concerning the mecanum wheels.

3.1 Preliminary definitions and concepts

3.1.1 Wheeled Mobile Robots, robot pose, twist

Mobile robots have been mentioned earlier in this thesis, without defining them properly. This will hence be the first definition of this section. All these concepts can be found in [30].

Definition 3.1.1 (Wheeled Mobile Robot). *A wheeled mobile robot is a robot capable of locomotion on a surface solely through the actuation of wheel assemblies mounted on the robot and in contact with the surface.*

Definition 3.1.2 (Wheel assembly). *A wheel assembly is a mechanical device that allows the wheel to rotate around its spin axis. In addition, the device may also allow the wheel to rotate around one orientation axis perpendicular to the spin axis and to the ground.*

Some additional hypotheses about the environment need to be introduced:

- the robot moves on a flat horizontal plane;
- the plane is a rigid, non deformable body;

about the robot:

- a mobile robot is composed of a rigid body called *platform*, to which the wheel assemblies are attached;
- the robot has at least three wheel assemblies, because the minimal number of not aligned wheels that ensures the robot stability is three;

and about the wheels and wheel-ground contact:

- there is a single point of contact between a wheel and the rolling plane;
- the wheels are always in contact with the ground and perpendicular to the rolling plane;
- the motion of the wheels is a pure rolling motion, i.e. the point of the wheel instantaneously in contact with the ground has zero velocity (Figure 3.1). Three properties derive from the pure rolling condition:
 - o the tangential speed v_t is zero: no slipping.
 - o the normal speed v_n is zero: no skidding.

- rotational slipping is allowed, meaning that ω_z is non zero when the robot turns.

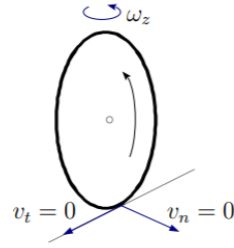


Figure 3.1: Scheme of the ideal wheel-ground contact (source: [30]).

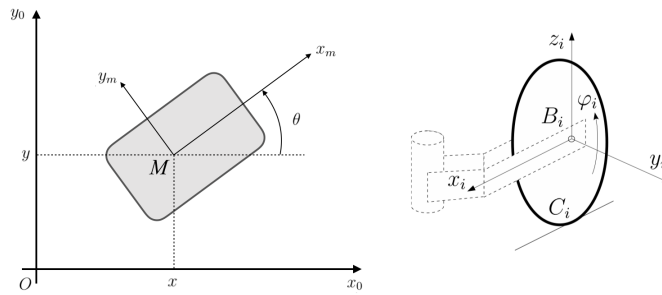


Figure 3.2: Parametrization of the robot pose and the frame of the wheel (source: [30]).

With reference to Figure 3.2, a mobile robot moving on a plane is identified by its position. The position of the robot is given with respect to a fixed frame $R_0(O, x_0, y_0, z_0)$ attached to the plane on which the robot moves, considering z_0 perpendicular to the plane and directed upward. The robot is endowed with a frame as well, called robot frame $R_m(M, x_m, y_m, z_m)$. The *pose* of the robot is instead the vector composed of the position and orientation, denoted by ${}^0\xi$, where the left exponent refers to the frame in which the pose is expressed. If not otherwise indicated, the pose of the robot will be expressed as it follows:

$$\xi = {}^0\xi = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \quad (3.1.1)$$

The orientation θ of the robot is called *heading*. This vector spans the *configuration space* \mathcal{C} , i.e. the space of all possible configurations of the robot.

For each wheel, an associated frame is defined. The frame associated with the i^{th} wheel is denoted by R_i and its origin B_i is the center of the wheel. The axes are oriented as shown in Figure 3.2 (on the right), so that the wheel spins with an angle φ_i around y_i . C_i is the point of the wheel in contact with the ground.

From the pose of the robot the following is derived:

Definition 3.1.3 (Twist). *The twist represents the instantaneous motion of a rigid body as an angular speed around an axis and a linear velocity perpendicular to this axis.*

For a mobile robot, the twist of the platform with respect to the frame R_0 is the time derivative of the pose:

$${}^0\dot{\xi} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix}. \quad (3.1.2)$$

The twist referred to the frame R_m can be obtained by multiplying the twist defined above with a rotation matrix:

$${}^0\dot{\xi} = {}^0\Omega_m {}^m\dot{\xi}, \quad \text{with: } {}^0\Omega_m = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}. \quad (3.1.3)$$

Remark. *Despite of its form, ${}^0\Omega_m$ is not a homogeneous transformation on $SE(2)$ (as they are defined in Chapter 8). This notation will only be used when it will be necessary to change the frame in which the twist is expressed.*

3.1.2 Degrees of freedom, degrees of mobility

Definition 3.1.4 (Degrees of Freedom). *In physics, the degree of freedom (DOF) of a mechanical system is the number of independent parameters that define its configuration [31].*

For example, the position and orientation of a rigid body in space is defined by three components of translation and three components of rotation, which means that it has 6 degrees of freedom.

In the case of a mobile robot, the number of degrees of freedom reduces to 3 because there are three constraints: the gravity force anchors the robot to the ground and hence there is no motion along the z axis and no rotations either around the x or around the y axis.

Definition 3.1.5 (Degrees of Mobility). *For a robot, the degrees of mobility δ_m are equal to the number of twist components (v_x, v_y, ω) that its wheel configuration allows to independently generate, while maintaining pure rolling. For a planar motion the maximum is hence three.*

3.1.3 Holonomic, nonholonomic, redundant and omnidirectional robots

Definition 3.1.6. *A robot is **holonomic** if all the kinematic constraints that it is subjected to can be put in the form:*

$$\frac{dh_i(\mathbf{q})}{dt} = \frac{\partial h_i(\mathbf{q})}{\partial \mathbf{q}} \dot{\mathbf{q}} = 0 \quad i = 1, \dots, k < n. \quad (3.1.4)$$

*The variables \mathbf{q}_i are the generalized coordinates, while the functions $h_i : \mathcal{C} \mapsto \mathbb{R}$ are of class \mathcal{C}^∞ (smooth) and independent. When a robot system contains constraints that cannot be expressed in this form, the robot is said to be **nonholonomic** [32].*

In simpler terms, a robot is called holonomic when the number of degrees of mobility is equal to the dimension of the configuration space.

An example of a nonholonomic system is a car. A car has three degrees of freedom, i.e. its position in $[x, y]$ and its orientation. However, there are only two controllable degrees of freedom which are acceleration (or braking) and turning angle of the wheels.

To conclude, in the case of a wheeled mobile robot with 3 DOF, omnidirectional has the same meaning as holonomic.

3.2 Types of wheels

In this section the parametrization of a general wheel assembly is presented, then the Fixed Wheel is introduced, to conclude with the Mecanum (or Swedish) Wheel, which is of great interest as it is used by the Ridgeback. This will lead to the derivation of the kinematic model in the next section.

3.2.1 The general wheel assembly

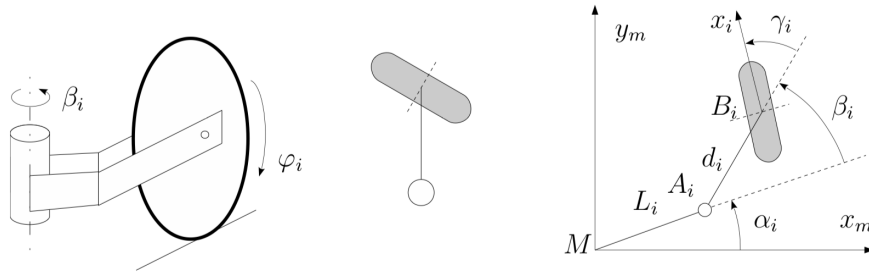


Figure 3.3: Parametrization of the general wheel assembly (source: [30]).

In Figure 3.3 the 3D view of the wheel assembly, together with its symbol and parametrization, is shown. The parameters of the i -th wheel can be identified as follows:

- L is the distance between the center of the robot M and the pivot of the wheel assembly A ;
- α is the angle formed by the segment \overline{MA} with the robot frame, i.e. the angle between the robot frame and the wheel assembly;
- d is the distance between the pivot of the wheel assembly and the wheel center B ;
- β is the angle formed by the segment \overline{AB} with the robot frame, i.e. the angle between the wheel assembly and the wheel center;
- γ is the orientation of the wheel.

The only parameters which vary with respect to the robot motion are β and the spin angle of the wheel φ . From this general wheel assembly, the conventional wheels (namely fixed, steering and castor wheels) are obtained by setting to zero certain parameters. Only the fixed wheel is detailed, as the others are beyond the scope of this thesis.

3.2.2 The fixed wheel

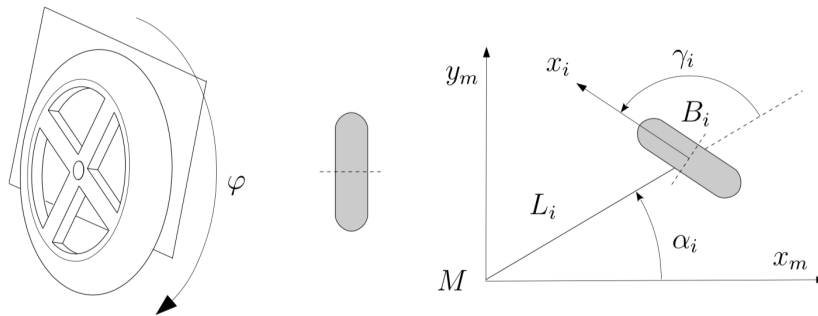


Figure 3.4: Parametrization of the fixed wheel (source: [30]).

In a fixed wheel (Figure 3.4) the spin axis y_i is fixed with respect to the mobile robot frame R_m . It is completely defined by the position of the origin B_i of the frame R_m and the orientation of x_i . The fixed wheel is derived from the general wheel assembly by setting $A_i = B_i$, $d_i = 0$ and $\beta_i = 0$.

3.2.3 The Mecanum (Swedish) wheel

Note: this section is taken from [33].

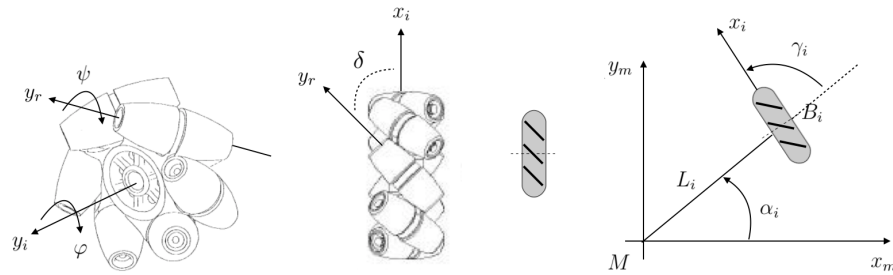


Figure 3.5: Parametrization of the mecanum wheel.

A Swedish wheel consists of a fixed standard wheel with passive rollers attached to the wheel circumference. The Mecanum wheel is a type of Swedish wheel where the angle between the passive-roller rotation axis and the wheel plane is $\delta = 45^\circ$ (Figure 3.5). The Mecanum wheel was invented in 1973 by Bengt Ilon, an engineer of the Swedish company Mecanum AB. This configuration allows in place rotation with small ground friction and low driving torque.

Usually (and this is the case for the Ridgeback) mobile robots using Mecanum wheels are designed with four wheels (in the X configuration) to provide agile mobility in any direction without changing their orientation, they are thus omni-directional robots (Figure 3.6).

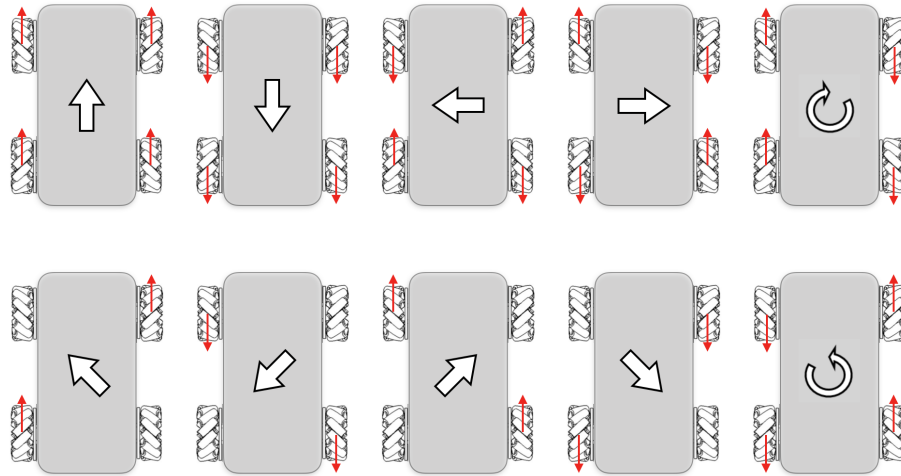


Figure 3.6: Wheels' command inputs vs. robot displacement.

Since a planar robot consisting of four Mecanum wheels has only 3 DOF (Section 3.1.2), it has one redundant degree of freedom. Concerning the parametrization of a mecanum wheel, it is the same as that of a fixed wheel except for the following differences:

- the wheel rotation φ is around the wheel axis y_i , while the roller rotation ψ is around the axis y_r ;
- the angle between the roller axis and the wheel plane is δ ;
- the radius of the fixed wheel part is R while the radius of the roller is r , which means that the distance between the wheel pivot B_i and the contact point C_i is $R + r$.
- the pivot of each roller is B' .

3.3 Kinematic model of a four mecanum wheeled mobile robot

To derive the kinematic model of a four mecanum wheeled mobile robot (mWMR), the expression of the velocity of the point C_i of the wheel in contact with the ground is derived, in the first place, then the equation is solved by imposing the pure rolling constraint. The constraint is expressed by imposing that the skidding and slipping components of the velocity are zero. The parameters that appear in the equations are those of the Figures 3.2, 3.3 and 3.5.

The velocity of point C for a general wheel is related to:

- the velocity and rotation speed of the robot platform;
- the orientation speed of the wheel $\dot{\beta}$;
- the spinning speed of the wheel $\dot{\varphi}$.

Other geometric parameters are constant, and hence do not contribute to the velocity of C . For an mWMR also the spinning velocity of the roller $\dot{\psi}$ has to be taken into account.

For each wheel, the velocity of the contact point is:

$$\vec{V}_c = \vec{V}_M + \dot{\theta} \vec{z}_m \times \vec{MC} + \dot{\beta} \vec{z}_m \times \vec{AC} + \dot{\varphi} \vec{y}_i \times \vec{BC} + \dot{\psi} \vec{y}_r \times \vec{B'C} \quad (3.3.1)$$

Before proceeding with solving the equation, it is necessary to provide a schematic representation of an mWMR.

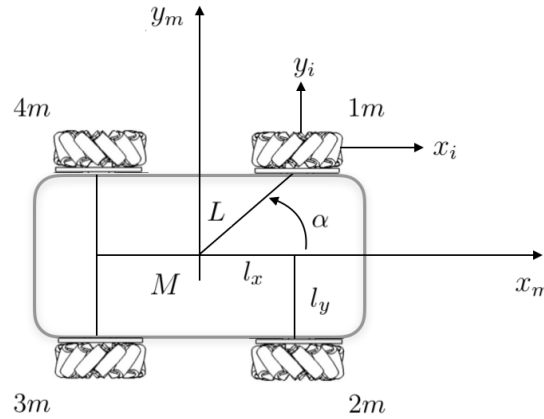


Figure 3.7: mWMR parametrization.

The parametrization of the mWMR can be resumed in a parameter table, as the one in Figure 3.8.

Wheel	L	α	d	β	γ	δ	φ	ψ
1m	L	α	0	0	$-\alpha$	$\pi/4$	φ_1	ψ_1
2m	L	$-\alpha$	0	0	α	$-\pi/4$	φ_2	ψ_2
3m	L	$\alpha + \pi$	0	0	$-\alpha - \pi$	$\pi/4$	φ_3	ψ_3
4m	L	$-\alpha + \pi$	0	0	$\alpha - \pi$	$-\pi/4$	φ_4	ψ_4

Figure 3.8: Parameter table of a 4 mWMR.

Moreover, for practical purposes, two others parameters are defined:

$$\begin{aligned} l_x &= L \cos \alpha \\ l_y &= L \sin \alpha \end{aligned} \quad (3.3.2)$$

Taking into account the given parametrization, the fact that the velocity along z is zero, and the pure rolling constraint (no slipping and no skidding), equation (3.3.1) for the i -th wheel can be rewritten as follows:

$$\begin{aligned} 0 &= v_x - (L \sin \alpha_i) \dot{\theta} - (R + r) \dot{\varphi}_i - (r \cos \delta_i) \dot{\psi}_i \\ 0 &= v_y + (L \cos \alpha_i) \dot{\theta} - (r \sin \delta_i) \dot{\psi}_i \end{aligned} \quad (3.3.3)$$

By deriving it for each wheel and using the parametrization in Table 3.8 and in (3.3.2), it follows:

$$\begin{aligned} 1m : & \begin{cases} v_x - l_y \dot{\theta} - (R + r) \dot{\varphi}_1 - \frac{\sqrt{2}}{2} r \dot{\psi}_1 = 0 \\ v_y + l_x \dot{\theta} - \frac{\sqrt{2}}{2} r \dot{\psi}_1 = 0 \end{cases} \\ 2m : & \begin{cases} v_x + l_y \dot{\theta} - (R + r) \dot{\varphi}_2 - \frac{\sqrt{2}}{2} r \dot{\psi}_1 = 0 \\ v_y + l_x \dot{\theta} + \frac{\sqrt{2}}{2} r \dot{\psi}_2 = 0 \end{cases} \\ 3m : & \begin{cases} v_x + l_y \dot{\theta} - (R + r) \dot{\varphi}_3 - \frac{\sqrt{2}}{2} r \dot{\psi}_1 = 0 \\ v_y - l_x \dot{\theta} - \frac{\sqrt{2}}{2} r \dot{\psi}_3 = 0 \end{cases} \\ 4m : & \begin{cases} v_x - l_y \dot{\theta} - (R + r) \dot{\varphi}_4 - \frac{\sqrt{2}}{2} r \dot{\psi}_1 = 0 \\ v_y - l_x \dot{\theta} + \frac{\sqrt{2}}{2} r \dot{\psi}_4 = 0 \end{cases} \end{aligned} \quad (3.3.4)$$

Now, for each equation in (3.3.4), the term $\dot{\psi}_i$ can be simplified. This operation needs to be done because the rollers are passive and hence their velocity cannot be controlled. It follows:

$$\begin{aligned} 1m : & v_x - v_y - (l_x + l_y) \dot{\theta} - (R + r) \dot{\varphi}_1 = 0 \\ 2m : & v_x + v_y + (l_x + l_y) \dot{\theta} - (R + r) \dot{\varphi}_2 = 0 \\ 3m : & v_x - v_y + (l_x + l_y) \dot{\theta} - (R + r) \dot{\varphi}_3 = 0 \\ 4m : & v_x + v_y - (l_x + l_y) \dot{\theta} - (R + r) \dot{\varphi}_4 = 0 \end{aligned} \quad (3.3.5)$$

The equations (3.3.5) can be rewritten to obtain the holonomic condition (3.1.4) and, hence, to understand why the Ridgeback is a holonomic robot. By combining the equations according to the expression $1m + 2m - 3m - 4m$, it follows:

$$\frac{\partial h_i(\mathbf{q})}{\partial \mathbf{q}} \dot{\mathbf{q}} = -(R + r) \begin{bmatrix} 1 & 1 & -1 & -1 \end{bmatrix} \begin{bmatrix} \dot{\varphi}_1 \\ \dot{\varphi}_2 \\ \dot{\varphi}_3 \\ \dot{\varphi}_4 \end{bmatrix} = 0 \quad (3.3.6)$$

Moreover, equation (3.3.6) allows to obtain one of the $\dot{\varphi}_i$ as a function of the other three, which means that only three spin velocities are linearly independent. From this consideration it follows that, for the Ridgeback, three independent wheel speed configurations allow to generate the whole twist vector $\dot{\xi}$ and, hence, the degree of mobility δ_m of the robot is three.

To conclude, two models can be derived.

The equations in (3.3.5) can be presented in the form of the relationship:

$$\dot{\varphi} = J \cdot {}^m \dot{\xi} \quad (3.3.7)$$

where:

$$\dot{\varphi} = \begin{bmatrix} \dot{\varphi}_1 \\ \dot{\varphi}_2 \\ \dot{\varphi}_3 \\ \dot{\varphi}_4 \end{bmatrix}, \quad J = \frac{1}{(R + r)} \begin{bmatrix} 1 & -1 & -(l_x + l_y) \\ 1 & 1 & (l_x + l_y) \\ 1 & -1 & (l_x + l_y) \\ 1 & 1 & -(l_x + l_y) \end{bmatrix}, \quad {}^m \dot{\xi} = \begin{bmatrix} v_x \\ v_y \\ \dot{\theta} \end{bmatrix}. \quad (3.3.8)$$

The other model is the direct (or forward) kinematic model, which is the model used by the odometry, where the velocity of the robot is derived from the velocity of the wheels. This model can be derived from the inverse one by inverting the Jacobian matrix J . Since J is a rectangular matrix, to obtain the inverse one can use the Moore-Penrose pseudoinverse:

$${}^m \dot{\xi} = J_{od} \cdot \dot{\varphi}, \quad J_{od} = (J^T J)^{-1} J^T = \frac{(R+r)}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & -1 & 1 \\ \frac{-1}{l_x+l_y} & \frac{1}{l_x+l_y} & \frac{1}{l_x+l_y} & \frac{-1}{l_x+l_y} \end{bmatrix}. \quad (3.3.9)$$

Simultaneous Localization and Mapping

In order to complete a task, a mobile robot needs to know the environment it is moving in. This environment is called “map” and the process of building up a map is called mapping. During the mapping process, the robot moves in an unknown area and combining its position, given by odometry and localization algorithms, with the data provided by the laser sensors, it is capable of building a 2D occupancy grid that indicates both the free space and the obstacles.

The algorithms that allow this kind of process are called SLAM (Simultaneous Localization and Mapping) algorithms and are the subject of this chapter, which entirely refers to the chapters 1, 8, 9 and 13 of the book *Probabilistic Robotics* [34].

4.1 Theory of probability: concepts and notations used

The aim of this section is to introduce the concepts and notations from probability theory used to explain the subject of this chapter. The reader is assumed to be familiar with the very basics of probability theory.

First of all, one may wonder why probability intervenes in robotics matters. The reason is that robots move in a physical world that presents many sources of uncertainty. The environment is unpredictable and its interaction with the robot is made possible by the knowledge of the data measured by robot sensors. Sensors, in turn, have a perception that is limited by their range and resolution, and they are also subjected to noise, so that the measurements are perturbed in unpredictable ways.

The motors that actuate the robot are unpredictable, as well, because the control can be noisy and there might be mechanic failures.

Finally, most of the times, the real information one needs (for example the location of the robot in a map) is not even directly measurable and has to be inferred.

Probability theory is, hence, necessary to cope with these uncertainties in robot actuation and perception by deriving a suitable probabilistic model.

The first tool that is presented allows to compute the *conditional* probability, namely the probability an event has, when conditioned by another event:

$$p(x|y) = \frac{p(x, y)}{p(y)} \quad (4.1.1)$$

From this the *Theorem of total probability* follows:

$$p(x) = \sum_y p(x|y)p(y) \quad (\text{discrete})$$

$$p(x) = \int p(x|y)p(y)dy \quad (\text{continuous}) \quad (4.1.2)$$

In light of this, the important *Bayes' rule* can also be determined:

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)} = \frac{p(y|x)p(x)}{\sum_{x'} p(y|x')p(x')} \quad (\text{discrete})$$

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)} = \frac{p(y|x)p(x)}{\int p(y|x')p(x')dx'} \quad (\text{continuous}) \quad (4.1.3)$$

If, as it is written above, x is the quantity to infer from y , then $p(x)$ is called the *prior probability distribution* and y is the *data*. The conditional probability $p(x|y)$ is hence the *posterior probability distribution* over the random variable X , where x is an event, i.e. a specific value that X may assume.

Finally, the rules defined above can also be conditioned on multiple random variables. So, for example, the Bayes' rule (4.1.3) can be rewritten as:

$$p(x|y, z) = \frac{p(y|x, z)p(x|z)}{p(y|z)} \quad (4.1.4)$$

Finally, if a variable y carries no information about a variable x , given that the value of a third variable z is known, one has the so called *conditional independence*:

$$p(x, y|z) = p(x|z)p(y|z) \quad (4.1.5)$$

Other notations used in [34] are the following:

- time is discrete, which means that the events take place at discrete time steps $t = 0, 1, 2, \dots$;
- x_t is the *state* of the robot and the environment at time t . One can think of the state as the collection of all the characteristics of the robot and of the environment that can impact the future (pose and velocity of the robot, location of the obstacles, . . .);
- z_t is the *measurement data* at time t , i.e. the information about the state of the environment. If measurements are taken in the time interval from an instant t_1 to an instant t_2 , the notation is $z_{t_1:t_2}$;
- u_t is the *control data*, corresponding to the change of state in the time interval $(t-1; t]$. Similarly to the previous case, sequences of control data are denoted by $u_{t_1:t_2}$.

It is important to notice that both control and measurement data are usually perturbed by some noise. Since the evolution of state and measurement is governed by probabilistic laws, the probability distribution that generates x_t can be written as:

$$p(x_t|x_{0:t-1}, z_{1:t-1}, u_{1:t}). \quad (4.1.6)$$

However, if the state is *complete*, i.e. the knowledge of past states, measurements or controls does not provide additional information to help predicting the future, the probability distribution becomes:

$$p(x_t|x_{0:t-1}, z_{1:t-1}, u_{1:t}) = p(x_t|x_{t-1}, u_t), \quad (4.1.7)$$

because the knowledge of x_{t-1} includes the one of the controls and measurement data up to the time instant $t-1$. Finally, the initial state distribution is $p(x_0)$.

The last key concept reported in *Probabilistic Robotics* is that of *belief*. A belief is the knowledge the robot has at disposal about the environment. For example, one can measure the state of the robot externally, but the robot itself cannot and it must infer it from data: this is hence the difference between the true state and the belief of it. From a probabilistic point of view, beliefs are represented as conditional probability distributions, i.e. posterior probabilities over state variables conditioned on the available data:

$$bel(x_t) = p(x_t|z_{1:t}, u_{1:t}). \quad (4.1.8)$$

These are all the tools that are used to explain the mathematical formalization of the algorithms that are implemented in practice to make a robot build a map while localizing itself in it.

All along this chapter, other basic filters, namely Bayes, Kalman, Extended Kalman and Particle filters, are mentioned. This is because the algorithms described in this chapter are derived from them. For the sake of completeness, they are all reported, explained and commented in the Appendix E.

4.2 Monte Carlo Localization

Localization is the problem of estimating the pose of the robot *in relation to a known map* of the environment.

There exist three types of localization problems that differ in the type of knowledge available initially and during the task:

- *Local localization*: also called *position tracking*, it assumes that the initial pose of the robot is known with little noise. So the uncertainty is local and near the true value of the robot pose;
- *Global localization*: in the global localization problem, the initial pose of the robot is unknown;
- *Kidnapped Robot problem*: in this case, during the localization task, the robot can disappear and be teleported to some other location.

Historically, one of the most powerful and important filters is Kalman filter and, in the case of mobile robotics, its extended version (EKF). In mobile robotics the EKF is applied to a non linear system describing the evolution of the model and the equations for the measurements, both with some additive noises characterized by their covariance matrices. The EKF is a robust filter, but it has three limitations:

1. the main hypothesis it relies upon is that the added noises and the initial belief $bel(x_0)$ are Gaussian (the noises have also a zero mean);
2. since an initial belief is needed, the EKF cannot solve the global localization problem;
3. the linearization of the filter is obtained by Taylor's Expansion, which returns an approximated linear version of the non linear system, as detailed in Appendix E.2.2;
4. the map used for robot localization must be feature-based, for example, using point landmarks. These features represent the measurements the filter uses to estimate the position of the robot.

The Monte Carlo Localization (MCL) algorithm is a good alternative to the EKF because it can process raw sensor measurements and, due to its non-parametric nature, it gets rid of the uni-modal distribution assumption of the EKF. In fact, the `measurement_model` algorithm, that models the range finder sensor, returns a probability $p(z_t|x_t, m)$ (where m is the map) that is a mixture of Gaussians, exponential and discrete distributions.

The MCL algorithm (pseudo code 4.1) is based on a particle filter, whose main drawback is the computational time, which grows exponentially with the number of particles. In this algorithm the key idea is to represent the belief $bel(x_t)$ by means of a set of random state samples, called *particles*, drawn from the belief itself.

In the MCL algorithm, the belief $bel(x_t)$ is represented by a set of M particles $\mathcal{X}_t = \{x_t^{[1]}, \dots, x_t^{[M]}\}$. In Line 6, each particle is sampled from the motion model of the robot (algorithm 5.6 that will be detailed in Section 5.2.1), i.e. from the transition distribution $p(x_t|u_t, x_{t-1})$. The initial belief $bel(x_0)$ is obtained by randomly generating M samples from the distribution $p(x_0)$ and by assigning uniformly to each particle the importance weight M^{-1} . Then, in Line 7, the importance weight w_t is computed for each particle. The importance weight is used to assign weights to the particles by incorporating the measurements: $w_t = p(z_t|x_t, m)$. This means that only the ones that better represent

Algorithm 4.1 MCL algorithm, pseudo code

```

1: function MCL( $\mathcal{X}_{t-1}, u_t, z_t, m$ )
2:
3:    $\bar{\mathcal{X}}_t, \mathcal{X}_t \leftarrow \emptyset$ 
4:
5:   for  $m = 1$  to  $M$  do
6:      $x_t^{[m]} \leftarrow \text{sample\_motion\_model}(u_t, x_{t-1}^{[m]})$  // sampling
7:      $w_t^{[m]} \leftarrow \text{measurement\_model}(z_t, x_t^{[m]}, m)$  // compute the importance weight
8:      $\bar{\mathcal{X}}_t \leftarrow \bar{\mathcal{X}}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$ 
9:
10:    // resampling process
11:    for  $m = 1$  to  $M$  do
12:      draw  $i$  with probability  $\propto w_t^{[i]}$ 
13:      add  $x_t^{[i]}$  to  $\mathcal{X}_t$ 
14:
15:  return  $\mathcal{X}_t$ 

```

the belief $bel(x_t)$ survive to subsequent loops. In Line 8 the weighted particles update a temporary particle set $\bar{\mathcal{X}}_t$.

Lines 11 to 13 realize the so called *importance sampling*: the particle set \mathcal{X}_t is populated by particles that are drawn from the set $\bar{\mathcal{X}}_t$ with a probability proportional to their weight: the particles with lower weights tend to disappear.

Figure 4.1 shows how the MCL works in the case of an uni-dimensional hallway. In this example, a robot can move only along the x axis and it uses its sensor to detect features and, hence, localize itself in the map. The features are the three doors placed in the corridor. Being a localization problem, the map is given to the robot, which tries to guess its position $bel(x_t)$ from the observations.

- (a) the initial global uncertainty is obtained by generating M identical samples of the pose space: each sample has the same importance weight;
- (b) as soon as the door is sensed by the robot, the filter updates the importance weights (the height of each bar): the particles in correspondence of a door have a higher weight. The temporary particle set $\bar{\mathcal{X}}_t$ contains such weighted particles;
- (c) the particle set \mathcal{X}_t is now updated by taking into account the resampling process and by integrating the robot motion. The particles of this set have been drawn from the temporary set with a probability proportional to their weight: particles corresponding to the more likely position of the robot have been drawn multiple times, while particles representing unlikely positions might have disappeared;
- (d) the new measurement updates the importance weights of the particles. Now, thanks to this operation the particles with highest weights are centered on the second door, which indicates also the most likely location of the robot;
- (e) with each iteration of the filter, the estimation of the robot location becomes more and more precise and representative of the real one.

The main drawback of this algorithm, as it is presented, is that it cannot verify if its pose is incorrect. Indeed, it can happen that it discards accidentally the particles near to the correct pose during the resampling process. This problem can be solved by implementing a heuristic behavior in the filter: during the resampling step, some particles are randomly added. These particles are hence based on an estimate of the localization performance. This version of the MCL algorithm is called Augmented MCL and it will be discussed in the next chapter.

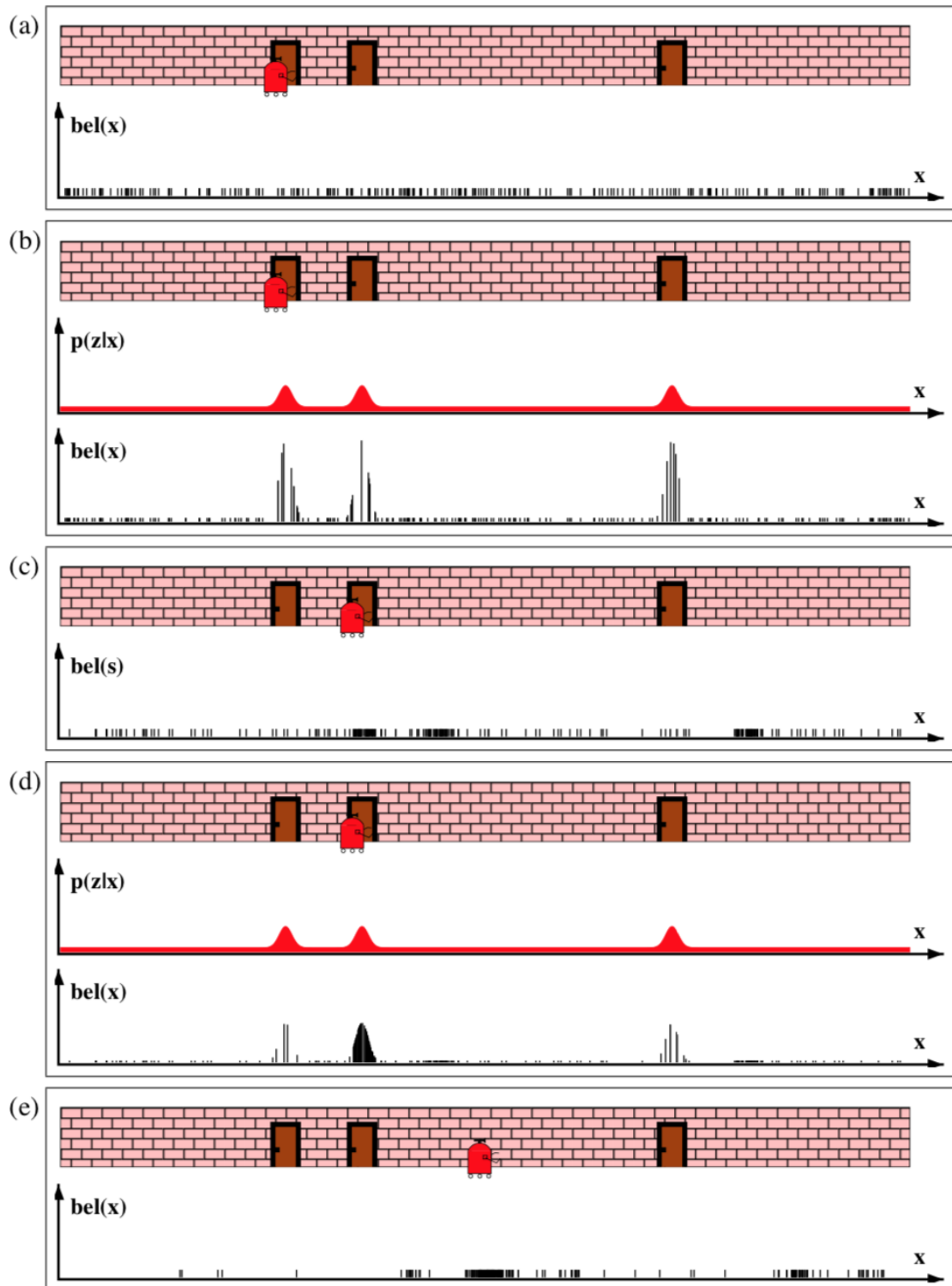


Figure 4.1: Monte Carlo particle filter behavior in localizing the robot (source: [34]).

4.3 Occupancy Grid Mapping

Mapping is the problem of generating the map of the environment the robot is in from noisy and uncertain measurement data and *under the assumption that the robot pose is known*.

The algorithms that deal with this kind of problems are called mapping algorithms. These algorithms can be based on features, i.e. recognizable objects in the environment, or on occupancy grids, where the map is represented as an evenly spaced grid. In this thesis, only the grid-based approach will be discussed.

Each grid can correspond to an obstacle or to a free space, a binary value is assigned accordingly. Occupancy grid mapping algorithms compute approximate posterior estimates for these random variables.

The posterior over maps given the data is

$$p(m|z_{1:t}, x_{1:t}), \quad (4.3.1)$$

where m is the map, $z_{1:t}$ the set of all measurements up to time t , and $x_{1:t}$ is the path of the robot, i.e. the sequence of all its poses.

An occupancy grid map partitions the space into grid cells:

$$m = \{\mathbf{m}_i\}, \quad (4.3.2)$$

where \mathbf{m}_i is the grid cell with index i . Each \mathbf{m}_i is a binary occupancy value, that specifies whether a cell is occupied (“1”) or free (“0”). The notation $p(\mathbf{m}_i)$ represents the probability that a grid cell is occupied.

Since the number of the grids describing a map is in the order of the tens of thousands, this grid partition can represent around $2^{10.000}$ maps, and calculating the posterior (4.3.1) for each single map is not feasible.

The algorithm, hence, simplifies the problem by estimating separately

$$p(\mathbf{m}_i|z_{1:t}, x_{1:t}), \quad (4.3.3)$$

for each grid cell \mathbf{m}_i . It follows that the posterior over maps is approximated by the product of its marginals:

$$p(m|z_{1:t}, x_{1:t}) = \prod_i p(\mathbf{m}_i|z_{1:t}, x_{1:t}). \quad (4.3.4)$$

The laser range finder

A map is built from sensor measurements. In particular, for the Ridgeback a laser range finder sensor is used, so, to understand the occupancy grid algorithm, the parametrization of such a sensor needs to be introduced.

A laser range finder emits a 2-D signal and records the echo once the signal bounces on an obstacle. The signal emitted is a focused light beam. The measurement process is modeled by the conditional probability density $p(z_t|x_t, m)$, which allows to cope with the uncertainties in the sensor model. The parameters that define the measurement beam are the measurement range z_t expressed in meters and defined in the interval $[0; z_{max}]$, the opening angle of the cone β and the resolution ρ both expressed in degrees (Figure 4.2). The existence of a resolution parameter is justified by the fact that range finders generate entire scans of ranges, i.e. the measurement z_t is carried out by N individual beams, one every ρ degrees, so it follows:

$$z_t = \{z_t^1, \dots, z_t^N\}, \quad (4.3.5)$$

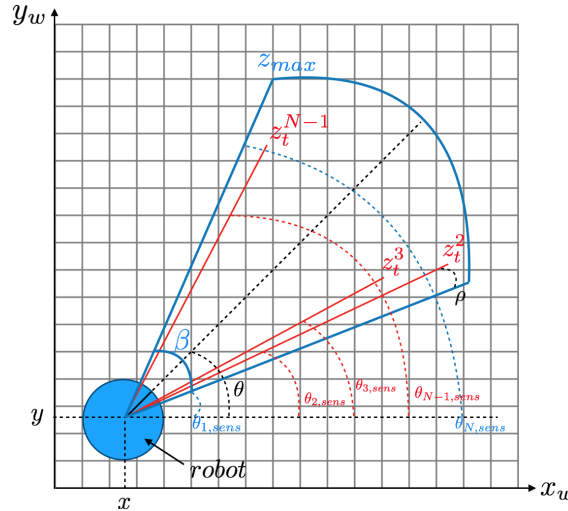


Figure 4.2: Scheme of the parametrization of a laser range finder.

and z_t^k , with $k = 1, \dots, N$ refers to a single measurement.

The model of the range sensor finder is far more complicated, but other details are of no use to understand the following algorithms.

The occupancy grid algorithm can be thought of as an adapted version of the *binary Bayes' filter*:

Algorithm 4.2 Occupancy grid algorithm, pseudo code

```

1: function OCCUPANCY_GRID_MAPPING( $\{l_{t-1,i}\}, x_t, z_t$ )
2:
3:   for all cells  $\mathbf{m}_i$  do
4:     if  $\mathbf{m}_i$  is in the sensor cone of the measurement  $z_t$  then
5:        $l_{t,i} \leftarrow l_{t-1,i} + \text{inverse\_sensor\_model}(\mathbf{m}_i, x_t, z_t) - l_0$ 
6:     else
7:        $l_{t,i} \leftarrow l_{t-1,i}$ 
8:
9:   return  $\{l_{t,i}\}$ 

```

To represent the occupancy l , a *log odds* notation is used, which prevents numerical instabilities for probabilities near to zero or one:

$$l_{t,i} = \log \frac{p(\mathbf{m}_i | z_{1:t}, x_{1:t})}{1 - p(\mathbf{m}_i | z_{1:t}, x_{1:t})}. \quad (4.3.6)$$

Algorithm 4.2 loops through all the cells and updates only those interested by the measurement z_t . For example, in the case of a laser sensor, all the cells that fall into the cone of the sensor. For these cells, the occupancy value is updated according to the `inverse_sensor_model` function:

$$\text{inverse_sensor_model}(\mathbf{m}_i, x_t, z_t) = \log \frac{p(\mathbf{m}_i | z_t, x_t)}{1 - p(\mathbf{m}_i | z_t, x_t)}. \quad (4.3.7)$$

Finally, l_0 is the constant prior occupancy:

$$l_0 = \log \frac{p(\mathbf{m}_i = 1)}{p(\mathbf{m}_i = 0)} = \log \frac{p(\mathbf{m}_i)}{1 - p(\mathbf{m}_i)}. \quad (4.3.8)$$

A way to implement the inverse measurement model is the following Algorithm 4.3.

Algorithm 4.3 Inverse sensor model function, pseudo code

```

1: function INVERSE_SENSOR_MODEL( $\mathbf{m}_i, x_t, z_t$ )
2:
3:    $x_i, y_i \leftarrow$  center of  $\mathbf{m}_i$ 
4:    $r \leftarrow \sqrt{(x_i - x)^2 + (y_i - y)^2}$  // range of the center of mass of the cell
5:    $\phi \leftarrow \text{atan2}(y_i - y, x_i - x) - \theta$  // bearing of the center of mass of the cell
6:    $k \leftarrow \arg \min_j |\phi - \theta_{j, \text{sens}}|$  // index of the center of mass of the cell
7:
8:   if  $r > \min(z_{\max}, z_t^k + \alpha/2)$  or  $|\phi - \theta_{k, \text{sens}}| > \beta/2$  then
9:     return  $l_0$ 
10:  else if  $z_t^k < z_{\max}$  and  $|r - z_t^k| < \alpha/2$  then
11:    return  $l_{\text{occ}}$ 
12:  else if  $r \leq z_t^k$  then
13:    return  $l_{\text{free}}$ 

```

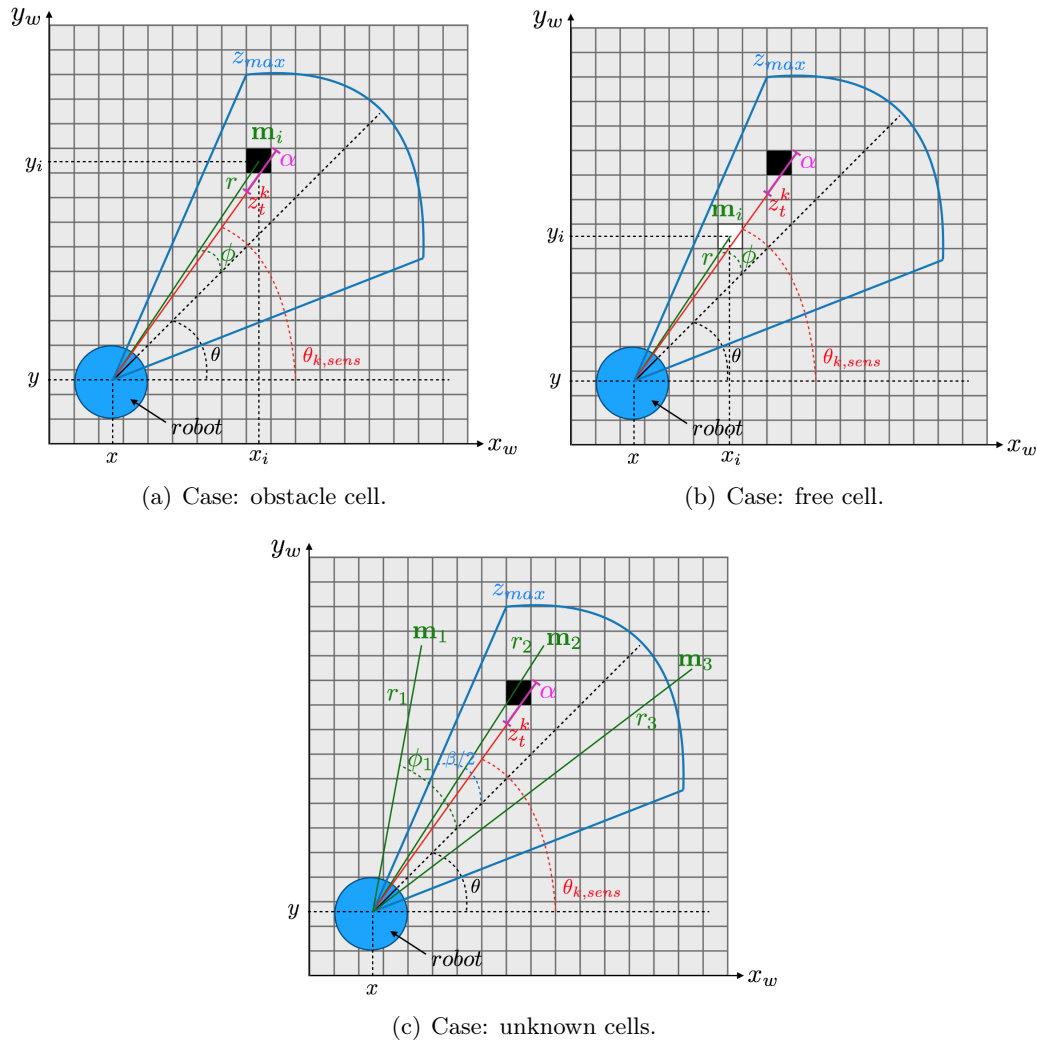


Figure 4.3: Scheme of the assignment of the occupancy value to a cell.

Algorithm 4.3 works under the assumptions that the robot pose is given by $x_t = [x, y, \theta]^T$ and that the sensor is centered in the robot. In the first place, in Line 3, the center of the cell \mathbf{m}_i is calculated. Then, the distance r , from the center of the cell to the position of the robot, is computed in Line 4. This distance is the range of the cell. In Line 5, the orientation of such a distance with respect to the orientation of the robot is computed. This value allows to retrieve the index k of the laser beam z_t^k that hit the cell. In fact,

in Line 6 this value is obtained as the index of the beam z_t^k that minimizes the angular difference between the heading of the cell ϕ and the heading of the beam z_t^k , expressed by $\theta_{k,sens}$. Once the laser beam that hit the cell has been identified, its range z_t^k is augmented by a value $\alpha/2$, where α is the obstacle thickness, and an occupancy value is assigned to the cell according to the following conditions:

- if the cell range r is greater than the minimum between the beam range z_t^k augmented by $\alpha/2$ and the maximum range of the sensor z_{max} , or if the cell is outside the scanning cone (Line 8), then the algorithm returns the prior l_0 (unknown zone, the gray cells in Figure 4.3c);
- if the range beam that hits the cell is lower than the maximum range and the range of the cell is within $\pm\alpha/2$ of the detected range z_t^k (Line 10), the algorithm returns $l_{occ} > l_0$, the black cell in Figure 4.3a;
- finally, if the range of the cell is shorter than the detected range z_t^k by more than $\alpha/2$ (Line 12), the algorithm returns $l_{free} < l_0$, the white cell in Figure 4.3b.

The occupancy grid algorithm presented in this section relies on the strong decomposition of equation (4.3.4). This assumption is intended to simplify the presentation of the occupancy grid mapping, but in practice more advanced derivations of this algorithm are used.

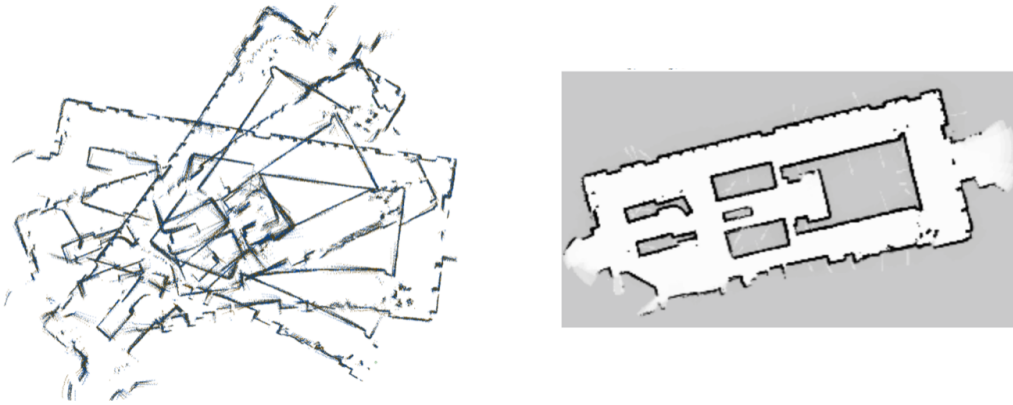


Figure 4.4: Map obtained with Occupancy Grid Mapping (right) from raw sensor data (left) based on odometry only (source: [34]).

4.4 The SLAM algorithm

The Simultaneous Localization and Mapping problem occurs when a robot knows neither the map of the environment nor its pose, it is only aware of the measurements $z_{1:t}$ and the command inputs $u_{1:t}$. Thanks to SLAM algorithms, the robot is capable of building the map of the environment while estimating its location with respect to the map.

The SLAM problem may come in two different forms, according to the posterior to estimate. If the probability to estimate is the posterior over the present pose, i.e. the pose at time t , together with the map,

$$p(x_t, m | z_{1:t}, u_{1:t}), \quad (4.4.1)$$

the problem is known as *online SLAM problem*. On the other hand, if the posterior is calculated over the entire path $x_{1:t}$, together with the map,

$$p(x_{1:t}, m | z_{1:t}, u_{1:t}), \quad (4.4.2)$$

one deals with the *full SLAM problem*.

Note that the online SLAM problem can be derived from its full version by integrating the poses:

$$p(x_t, m | z_{1:t}, u_{1:t}) = \int \int \cdots \int p(x_{1:t}, m | z_{1:t}, u_{1:t}) dx_1 dx_2 \dots dx_{t-1} \quad (4.4.3)$$

For convenience, one can consider a *combined state vector* that includes both the robot pose and the map:

$$y_t = \begin{pmatrix} x_t \\ m \end{pmatrix}, \quad (4.4.4)$$

so that the posterior (4.4.2) can be rewritten as

$$p(y_{1:t} | z_{1:t}, u_{1:t}). \quad (4.4.5)$$

In this section a *particle filter* approach to SLAM is presented.

In particular, another version of the particle filter can be applied: the *Rao-Blackwellized particle filter*, which uses a mixture of particles and Gaussians to represent variables. A SLAM algorithm that relies on this version of the particle filters takes the name of *FastSLAM*.

In FastSLAM, particle filters are used to estimate the robot path, so, for each particle, the individual map errors are conditionally independent. Thanks to this property, the mapping problem can be factorized and addressed separately for each feature of the map. Hence, each feature location is estimated by an EKF.

To conclude this brief introduction, the main features of the FastSLAM algorithm are:

1. it uses particle filters and thus it can handle non-linear robot motion models without approximating them with linearization;
2. it solves both the full and the online SLAM problems because it calculates the full path posterior that makes feature locations independent, but *de facto* it estimates one pose at-a-time.

There exist two main different versions of the FastSLAM algorithm: the first one is feature-based, while the other is grid-based and it uses some of the results reported in Section 4.3.

4.4.1 Feature-based FastSLAM

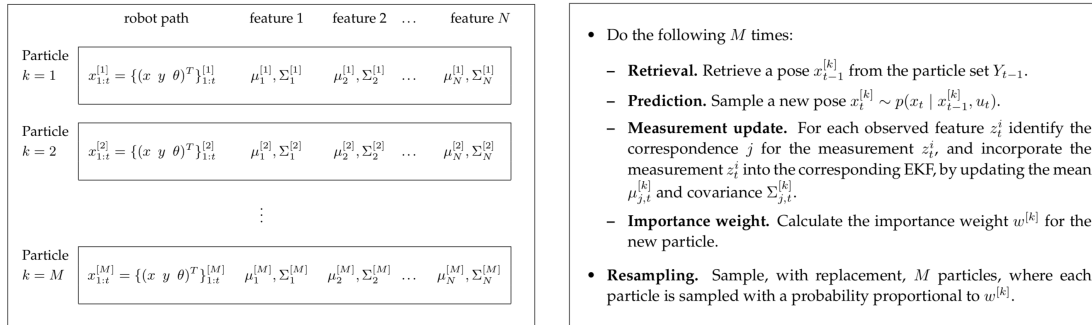


Figure 4.5: Feature-based FastSLAM: particle set and algorithm (source: [34]).

The feature-based version of the FastSLAM algorithm relies on the feature-based measurement model. In this model features f are extracted from measurements, $f(z_t)$, an approach that enormously reduces the computational complexity. However, additional specific algorithms for feature extraction and recognition are required.

In feature-based FastSLAM, particles contain an estimation of the robot pose $x_t^{[k]}$ and a set of Kalman filters for each feature m_j in the map. Each Kalman filter is used to estimate the location of a feature in the map, identified by its first and second moment, $\mu_{j,t}^{[k]}$ and $\Sigma_{j,t}^{[k]}$, for the k^{th} particle. The algorithm, as every particle filter algorithm, retrieves the particle at time $t - 1$ and samples the new pose at time t . Then, whenever a new feature is observed, it updates the Extended Kalman Filters. Finally, it updates the importance weight of the particle, which is used in the resampling process.

The feature-based representation of the map allows to factorize the posterior (4.4.5):

$$p(y_{1:t}|z_{1:t}, u_{1:t}, c_{1:t}) = p(x_{1:t}|z_{1:t}, u_{1:t}, c_{1:t}) \prod_{n=1}^N p(m_n|x_{1:t}, z_{1:t}, c_{1:t}), \quad (4.4.6)$$

where $c_{1:t}$ is the *correspondence variable* between the feature observed and the real feature in the map, variable that helps identifying the observed feature.

The true nature of the algorithm lies in this factorization because the posterior over robot paths $p(x_{1:t}|z_{1:t}, u_{1:t}, c_{1:t})$ is computed by a particle filter, while each posterior $p(m_n|x_{1:t}, z_{1:t}, c_{1:t})$ is handled by a dedicated EKF. At this point, the posterior is thus factorized in $N + 1$ products but, since the N Kalman estimations are performed for each one of the M particles, the actual number of filters is $MN + 1$.

A schematic representation of the particle set, together with the basic steps of the algorithm, is proposed in Figure 4.5.

4.4.2 Grid-based FastSLAM

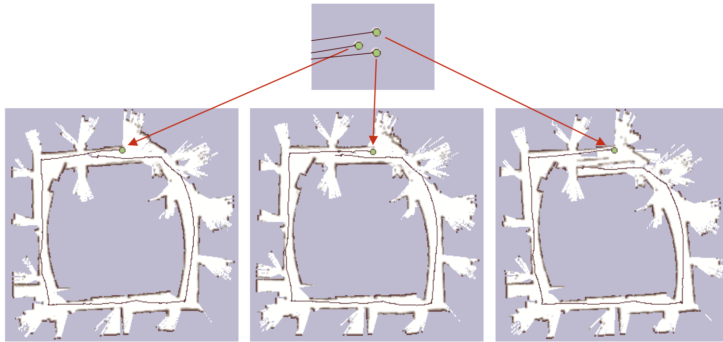


Figure 4.6: In grid-based FastSLAM each particle builds a map (source: [34]).

This derivation of the FastSLAM algorithm does not use feature-based maps, hence, it does not require EKFs to estimate the feature localization. On the contrary, it merges the Monte Carlo Localization (pseudo code 4.1) with the Occupancy Grid Mapping (pseudo code 4.2):

Algorithm 4.4 Grid-based FastSLAM algorithm, pseudo code

```

1: function FASTSLAM_OCCUPANCY_GRID( $\mathcal{X}_{t-1}, u_t, z_t$ )
2:
3:    $\bar{\mathcal{X}}_t, \mathcal{X}_t \leftarrow \emptyset$ 
4:
5:   for  $k = 1$  to  $M$  do
6:      $x_t^{[k]} \leftarrow \text{sample\_motion\_model}(u_t, x_{t-1}^{[k]})$ 
7:      $w_t^{[k]} \leftarrow \text{measurement\_model\_map}(z_t, x_t^{[k]}, m_{t-1}^{[k]})$ 
8:      $m_t^{[k]} \leftarrow \text{updated\_occupancy\_grid}(z_t, x_t^{[k]}, m_{t-1}^{[k]})$ 
9:
10:     $\bar{\mathcal{X}}_t \leftarrow \bar{\mathcal{X}}_t + \langle x_t^{[k]}, w_t^{[k]}, m_t^{[k]} \rangle$ 
11:
12:    for  $k = 1$  to  $M$  do
13:      draw  $i$  with probability  $\propto w_t^{[i]}$ 
14:      add  $\langle x_t^{[i]}, m_t^{[i]} \rangle$  to  $\mathcal{X}_t$ 
15:
16:  return  $\mathcal{X}_t$ 

```

The functions used above for the FastSLAM are not exactly the same as the ones used in Monte Carlo and Occupancy Grid. In particular:

- the `sample_motion_model` function computes the sample $x_t^{[k]}$ after integrating the motion model, i.e. the effect of the input u_t on the previous sample $x_{t-1}^{[k]}$. This means that it computes the posterior $p(x_t | x_{t-1}^{[k]}, u_t)$;
- the `measurement_model_map` function expresses the importance weight w_t of the k -th particle through the likelihood $p(z_t | x_t^{[k]}, m_{t-1}^{[k]})$ of the measurement z_t , given the pose x_t represented by the same particle and the map m_{t-1} computed via the previous measurement and the trajectory covered so far by this particle;
- the `updated_occupancy_grid` function uses the pose of the k -th particle, the map associated to it and the measurement to compute a new occupancy grid.

In this algorithm, each particle is responsible for building its own map (Figure 4.6); in the end, only the map corresponding to the particle with the highest importance weight is taken as outcome.

Before ending this section, a more efficient implementation of grid-based SLAM is described. It is also the one used in the simulation. This technique is described in [35].

In the algorithm described in pseudo code 4.4, the observation likelihood $p(z_t|x_t^{[k]}, m_{t-1}^{[k]})$ most of the times is peaked, i.e. it presents a very high likelihood in a narrow interval, expressed by $L^{[k]}$, around the real robot pose and a very low likelihood elsewhere. The interval can be formally defined as:

$$L^{[k]} = \left\{ x \mid p(z_t|m_{t-1}^{[k]}, x) > \varepsilon \right\}, \quad (4.4.7)$$

where ε is a threshold value that identifies the likelihood peak.

Given that this likelihood is used to weight the particles, a high number of particles is required to be sure to cover the interval. This means that a high number of pose samples $x_t^{[k]}$ has to be sampled from the motion model $p(x_t|x_{t-1}^{[k]}, u_t)$.

Moreover, the motion model distribution is a smooth function, so, given that a high number of particle is already required, the distribution to sample can be improved so that it presents the same peak as the observation likelihood. This can be done by integrating, in the motion model, the most recent measurement available, to obtain

$$p(x_t|m_{t-1}^{[k]}, x_{t-1}^{[k]}, z_t, u_t). \quad (4.4.8)$$

This new improved distribution allows to sample the motion model only around the pose $x_t^{[k]}$ that corresponds to that likelihood peak of the sensor. With this operation, less meaningful regions of the likelihood are ignored. Thus, the required amount of samples and, consequently, of computational time, is substantially reduced. The new samples set is obtained by taking $K \ll M$ samples $x_t^{[i]}$, $i = 0, \dots, K$.

To efficiently draw the new samples, the improved distribution is approximated by a Gaussian $\mathcal{N}(\mu_t^{[i]}, \Sigma_t^{[i]})$, where $\mu_t^{[i]}$ and $\Sigma_t^{[i]}$ are computed for each particle $x_t^{[i]}$.

Finally, also the weights $w_t^{[i]}$ are computed for this reduced set of particles.

To understand the efficiency of such an improvement, the same environment of Figure 4.6 has been mapped both in [34] and [35] (the *Intel Research Lab* in Seattle). The Lab has a size of $28m \times 28m$, and with the base *FastSLAM* algorithm, the map has been obtained with around 500 particles and a resolution of $10cm$, while with the improved technique only 15 particles have been used, for a $1cm$ resolution!

To conclude, the main advantage of the feature-based *FastSLAM* approach is that one can control the computational complexity by changing both the size M of the particle set and the number N of features describing the map, which are localized with an EKF. On the other hand, the grid-based approach takes advantage of both Monte Carlo Localization and Occupancy Grid Mapping by using particle filters. Thus, a grid-based approach is generally preferred because it requires neither these features to be defined, nor dedicated feature recognition algorithms. It is, hence, more portable because it can model any type of environment.

Finally, an improved technique for grid-based *FastSLAM* has been briefly discussed. This technique allows to drastically reduce the number of particles, required by the standard approach, by relying on Gaussian distributions to approximate the posterior only around its maximum.

It is this last version that is used, both in simulation and in practice, to build a map with Ridgeback.

Navigation

The previous chapter showed how the robot is able to create a map while localizing itself. The aim of this chapter is, therefore, to understand how the robot can generate a path to navigate an environment, while avoiding obstacles. The tools used to generate trajectories are presented, together with showing how the mapping process can be done autonomously, i.e. without manually controlling the robot.

The algorithms used for the path planning are included in the so called *global planner* and *local planner*, while the method used for the autonomous mapping is called *frontier exploration*: these are the topics of the first part of this chapter.

These algorithms are not only used for SLAM, but also to navigate a map once it has been built. In such a case, they are supported by an improved version of the Monte Carlo algorithm presented in Chapter 4, namely the *Augmented Monte Carlo Localization* (AMCL), which is considered in the second part of this chapter.

The first part of this chapter refers to [36], while the second part to [34] again.

5.1 Path Planning

Path planning addresses the problem of creating a trajectory the robot has to follow, from its position to a goal, while avoiding obstacles. In this process, two tools are used:

- a global planner, that is responsible for finding the optimal trajectory that connects a starting point to a final one;
- a local planner that is responsible for telling the robot how to behave, and hence which command should be sent to the wheels, in order to adhere to the global planner.

In simpler words, while the global planner searches for the shortest path in a digital representation of the known map (a graph or a grid), the local planner states *where* the robot can go and *how*. Moreover, if the robot is doing autonomous SLAM, another algorithm is needed to tell the robot which goal to reach in order to explore a map: the frontier exploration algorithm. Since finding a goal is part of path planning, the frontier exploration is presented in this section.

The path planning problem is defined in the *Workspace* \mathcal{W} , which is an Euclidean space: \mathbb{R}^2 or \mathbb{R}^3 . The workspace may contain some obstacles that are indicated with the set \mathcal{O} , the i -th obstacle defined in the workspace is indicated with $\mathcal{W}\mathcal{O}_i$.

Thus, the free workspace is:

$$\mathcal{W}_{free} = \mathcal{W} - \bigcup_i \mathcal{W}\mathcal{O}_i \quad (5.1.1)$$

However, path planning lives in the *Configuration space* \mathcal{Q} ¹, where a configuration q specifies the robot position with respect to a fixed frame. For example in the case of the Ridgeback, its configuration is its pose: $q = \xi = [x, y, \theta]^T$.

¹The configuration space may not be an Euclidean space. Indeed, for manipulators it is described by topological manifolds (*ex*: a torus for a 2R manipulator).

Equivalently to the workspace, in the configuration space the following can be identified:

$$\begin{aligned} \text{configuration space obstacle: } \mathcal{QO}_i &= \{q \mid \mathcal{R}(q) \cap \mathcal{WO}_i \neq \emptyset\} \\ \text{free configuration space: } \mathcal{Q}_{free} &= \mathcal{Q} - \bigcup_i \mathcal{QO}_i \end{aligned} \quad (5.1.2)$$

where $\mathcal{R}(q)$ is the space occupied by the robot in configuration q .

With all these elements the path planning problem can be better formulated. It searches for a \mathcal{C}^0 function $\tau : [0, 1] \rightarrow \mathcal{Q}_{free}$ with boundary constraints:

- $\tau(0) = q_{start}$;
- $\tau(1) = q_{goal}$.

If this function is parametrized by time t , $\tau(t)$ is called *trajectory*. Moreover, if also velocities and accelerations need to be computed, τ must be at least \mathcal{C}^2 .

To find such a function is the task of the global planner.

5.1.1 The global planner

A global planner is an algorithm that computes the best trajectory the robot has to follow, from an initial to a target position. In SLAM, as the robot moves, it collects data about the surrounding environment with its sensors and it uses them to build a map. The global planner uses the available map to compute the optimal trajectory, taking into account the path length and the obstacle avoidance. If the end point is in an unknown area, the global planner simply takes as a trajectory the straight line that connects the target point to its closest known point.

In 2D occupancy grid maps, the two most used algorithms for path planning are Dijkstra and A*. Both of them will be, hereafter, presented as implemented in weighted graphs, because this is how they are usually explained in the literature. In the adaptation to grid maps, each node can be seen as a cell, but the search for adjacent cells is made in the 4-connected or 8-connected neighborhood, as shown in Figure 5.1, where the starting cell is the central one and its adjacent cells the red ones.

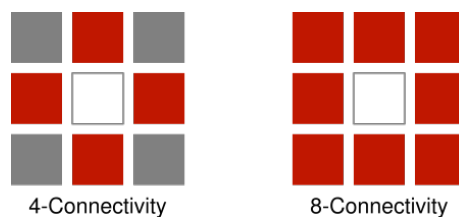


Figure 5.1: Pixel connectivity scheme.

Let $G = (V, E)$ be an undirected graph, where V is the set of vertices of the graph and E the set of the edges, each of them defined as a pair of vertices (because the edge is graphically represented as an arc connecting two vertices). A cost function $c : E \rightarrow R$, that maps each edge into a real cost value (here a distance cost is used), is also associated to the graph. It follows that:

- the cost of the path p is $\text{dist}[p] = \sum_i c(v_{i-1}, v_i)$, i.e. the cost of a path is equal to the sum of the costs of its constituent edges;
- the minimum cost (or the distance) between two vertices u and v , denoted by $\text{length}(u, v)$, is the minimum cost of a path from u to v , and it takes the value $+\infty$ if it does not exist.

Dijkstra Algorithm

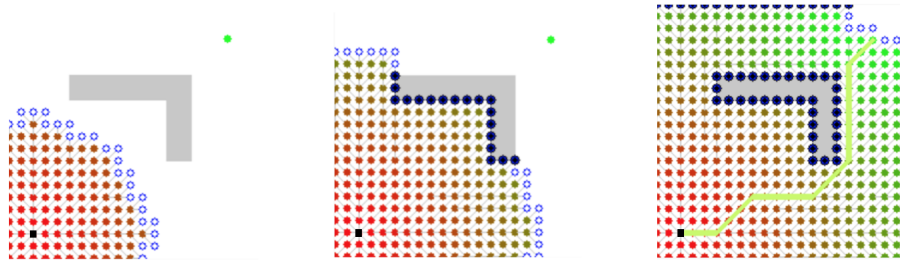


Figure 5.2: Breadth-first expansion in Dijkstra algorithm.

Dijkstra algorithm is a breadth-first-search algorithm operating on graphs (Figure 5.2). Breadth-first means that the search expansion is made in all possible directions, without favoring one in particular; it finds the optimal path from a single source vertex to all the others [37].

The Dijkstra algorithm takes as its input the graph G and the source s , and it can be written as in Algorithm 5.1.

Algorithm 5.1 Dijkstra algorithm, pseudo code

```

1: function DIJKSTRA( $G, s$ )
2:
3:   create vertex set  $Q$ 
4:
5:   for all vertex  $v$  in  $V$  do
6:      $\text{dist}[v] \leftarrow \infty$ 
7:      $\text{prev}[v] \leftarrow \text{undefined}$            // previous node that has an optimal path from s
8:     add  $v$  to  $Q$ 
9:
10:   $\text{dist}[s] \leftarrow 0$                        // the distance from s to s is 0
11:
12:  while  $Q$  not empty do
13:     $u \leftarrow$  vertex in  $Q$  with min  $\text{dist}[u]$ 
14:    remove  $u$  from  $Q$ 
15:
16:    for all each neighbor  $v$  of  $u$  do
17:       $\text{alt} \leftarrow \text{dist}[u] + \text{length}(u, v)$ 
18:      if  $\text{alt} < \text{dist}[v]$  then           // then the path found is better
19:         $\text{dist}[v] \leftarrow \text{alt}$ 
20:         $\text{prev}[v] \leftarrow u$ 
21:
22:  return  $\text{dist}[\ ]$ ,  $\text{prev}[\ ]$ 

```

Finally, to obtain the shortest path from target to goal, it is necessary to build a new sequence containing all the vertices in $\text{prev}[\]$ but in the reverse order.

A* Algorithm

Unlike Dijkstra's algorithm, A* is a best-first search algorithm (Figure 5.3). Best-first means that the nodes of a graph are explored in the direction of the most promising vertex, according to a specified rule. The rule is a heuristic evaluation function $f(v)$ that depends on the vertex v . The heuristic function returns a cost that is the result of a trade off between optimality and computation time.

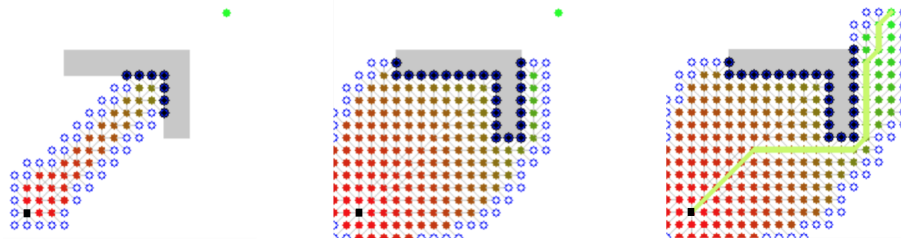


Figure 5.3: Best-first expansion in A* algorithm.

Thus, A* algorithm 5.2 selects the path that minimizes the evaluation function:

$$f(v) = g(v) + h^*(v) \quad (5.1.3)$$

where $g(v)$ is the cost of the path from the source s to the vertex v and $h^*(v)$ is a heuristic function that estimates the cost of the cheapest path from s to v . Moreover, by setting $g(v) = 0$ one can obtain a cost function only based on the heuristic, namely a *greedy* best-first search, while by setting $h^*(v) = 0$ one gets the Dijkstra's algorithm back.

Algorithm 5.2 A* algorithm, pseudo code

```

1: function A*(start, goal)
2:
3:   closeSet  $\leftarrow$   $\emptyset$ 
4:   add start to openSet
5:
6:   came_from  $\leftarrow$  empty map // the map of navigated nodes
7:
8:    $g(\text{start}) \leftarrow 0$  // cost from start along best known path
9:    $f(\text{start}) \leftarrow g(\text{start}) + h^*(\text{start}, \text{goal})$  // estimated total cost from start to goal
10:
11:  while openSet is not empty do
12:    current  $\leftarrow$  node in openSet with  $\min(f(\cdot))$ 
13:    if current = goal then
14:      return reconstruct_path(came_from, goal)
15:
16:    remove current from openSet
17:    add current to closeSet
18:    for all neighbor in neighbor_nodes(current) do
19:      if neighbor in closeSet then
20:        continue
21:      tentative_g  $\leftarrow g(\text{current}) + \text{dist}(\text{current}, \text{neighbor})$ 
22:
23:      if neighbor not in openSet or tentative_g <  $g(\text{neighbor})$  then
24:        came_from(neighbor)  $\leftarrow$  current
25:         $g(\text{neighbor}) \leftarrow$  tentative_g
26:         $f(\text{neighbor}) \leftarrow g(\text{neighbor}) + h^*(\text{neighbor}, \text{goal})$ 
27:        if neighbor not in openSet then
28:          add neighbor to openSet
29:    return failure
30:
31: function RECONSTRUCT_PATH(came_from, current)
32:
33:   total_path  $\leftarrow$  [current]
34:   while current in came_from do
35:     current  $\leftarrow$  came_from(current)
36:     total_path.append(current)
37:   return total_path

```

To find an optimal solution, when it exists, the heuristic $h^*(v)$ must be *admissible*:

$$\forall v \in V, h^*(v) \leq h(v) \quad (5.1.4)$$

that is, it never overestimates the actual cost $h(v)$ from v to the goal.

The two most used heuristics are:

- Euclidean distance:

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{\sum_{i=1}^n (q_i - p_i)^2} \quad (5.1.5)$$

- Manhattan distance:

$$d_1(\mathbf{p}, \mathbf{q}) = \|\mathbf{p} - \mathbf{q}\|_1 = \sum_{i=1}^n |p_i - q_i| \quad (5.1.6)$$

The A* algorithm 5.2 uses two lists: `openSet` and `closeSet`. The former contains vertices that have been visited but not expanded (i.e. the successor has not been visited yet), while the latter stores visited and expanded vertices. The algorithm returns either a failure or the path from the start to the goal.

5.1.2 The local planner

The local planner creates a feasible trajectory, from a kinematic point of view, from a starting point to a goal location. Since the planner works locally, these two locations are the center of the robot and a point which is within some meters. In between these two points the planner evaluates a cost function, from points belonging to a grid-based local cost map. The local cost map is a grid map of fixed dimension, centered in the robot position (x, y) in the global map. This function computes the cost of navigating through the grids, taking into account the occupancy value of the grids, the velocities of the robot and the distances from the global plan and from the goal.

Then, a controller uses the information about the cost function to determine the command twist $([v_x, v_y, \dot{\theta}]^T)$ to be sent to the robot.

The local planners that have been tested for the simulation are four: the first two are based on the Dynamic Window Approach (DWA) algorithm [38], while the third one implements the Elastic Bands (Eband) [42]. Finally, the last one is another version of the DWA, implemented by us. In this section, all these algorithms are presented and compared by analysing their qualities and their flaws.

Trajectory Planner

In the first place, it is necessary to understand what the Dynamic Window Approach is, and how does it work. Its standard version (presented in [38]) was designed for non-holonomic robots, which, due to their wheel configuration, can only develop the velocities along x and around θ , so that their twist vector is $\dot{\xi} = [v, \omega]^T$.

At each iteration, DWA executes the following steps:

1. **Search Space:** the algorithm computes the space of all possible velocities by taking into account:
 - **Circular Trajectories:** resulting trajectories from the velocity couple (v, ω) .
 - **Admissible Velocities:** for obstacle avoidance, only admissible velocities, i.e. “safe”, are taken into account. The set of admissible velocities represents the

velocities a robot can assume while being able to come to a stop without causing a collision. It is given by:

$$V_a = \{(v, \omega) | v \leq \sqrt{2 \text{dist}(v, \omega) \dot{v}_b} \wedge \omega \leq \sqrt{2 \text{dist}(v, \omega) \dot{\omega}_b}\} \quad (5.1.7)$$

where the pair $(\dot{v}_b, \dot{\omega}_b)$ represents the braking accelerations and the term $\text{dist}(v, \omega)$ is the distance of the closest obstacle on the specific (v, ω) trajectory.

- **Dynamic Window:** the dynamic window set V_d is the set of all the velocities resulting from a uniform acceleration motion, given the accelerations $(\dot{v}, \dot{\omega})$ and the initial velocity (v_c, ω_c) , i.e. the velocity of the robot :

$$V_d = \{(v, \omega) | v \in [v_a - \dot{v}t; v_a + \dot{v}t] \wedge \omega \in [\omega_a - \dot{\omega}t; \omega_a + \dot{\omega}t]\} \quad (5.1.8)$$

Thus, upon indicating with V_s the set of all possible velocities, the resulting search space is given by:

$$V_r = V_s \cap V_a \cap V_d \quad (5.1.9)$$

In Figure 5.4 all the previous sets are represented:

- the external rectangle is the overall search space;
- the set of admissible velocities is the light gray area, i.e. the intersection between the search space and the dark gray areas, which, in turn, represent the velocities to discard to avoid collisions;
- the Dynamic Window, that takes into account the accelerations, is the white rectangle;
- the resulting set V_r is the intersection of the other three sets.

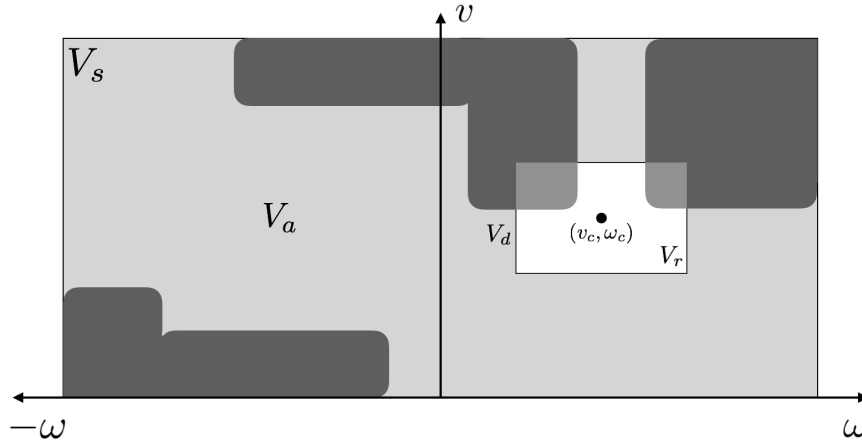


Figure 5.4: Dynamic Window Approach.

2. **Optimization:** the algorithm, for each trajectory, minimizes the cost function with respect to the current pose:

$$G(v, \omega) = \min(\alpha \cdot \text{heading}(v, \omega) + \beta \cdot \text{occ_dist}(v, \omega) + \gamma \cdot \text{path_dist}(v, \omega) + \delta \cdot \text{goal_dist}(v, \omega)) \quad (5.1.10)$$

The four functions in equation (5.1.10) are computed with respect to the pose resulting by applying the trajectory (v, ω) and they are detailed hereafter:

- **Heading:** is the orientation of the robot with respect to the local goal location. The local goal is the goal of the local planner, identified as the intersection of the local cost map with the global planner trajectory. It is, hence, given by $\pi - \theta$, where θ is the angle between the target and the resulting heading of the trajectory (v, ω) (see Figure 5.5).
- **Occupancy Distance:** occ_dist is the cost value of the robot position with respect to the nearest obstacle. This value is calculated as depicted in Figure 5.6 [39]. In Figure 5.5 the inscribed radius is reported in the cost map to form the light blue area, while the circumscribed radius is represented by the dark blue zone.
- **Path Distance:** is the distance from the robot pose obtained once (v, ω) is applied, to the global planner trajectory (“ pd ” in Figure 5.5).
- **Goal Distance:** is the distance in cells from the robot pose obtained once (v, ω) is applied, to the local goal, taking into account the obstacles (“ gd ” in Figure 5.5, with the respective detailed scheme). This means that if the local goal is behind an obstacle, the goal distance value represents the distance, in cells, of the shortest path to the local goal that avoids obstacles.

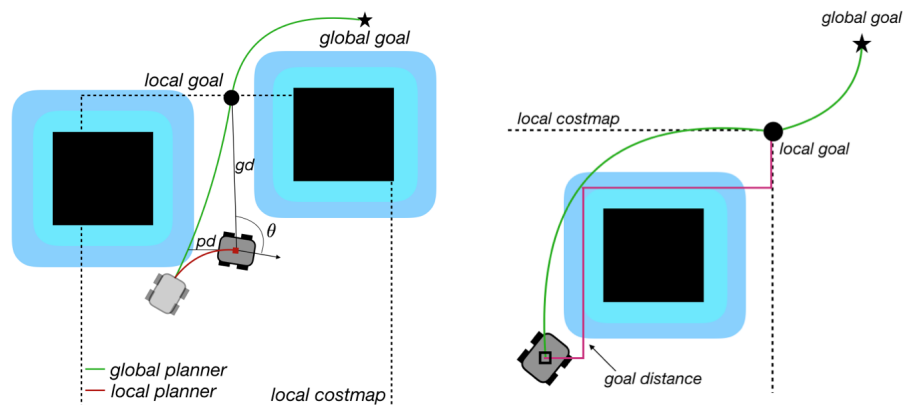


Figure 5.5: Parameters of the cost function for DWA with the detail for `goal_dist`.

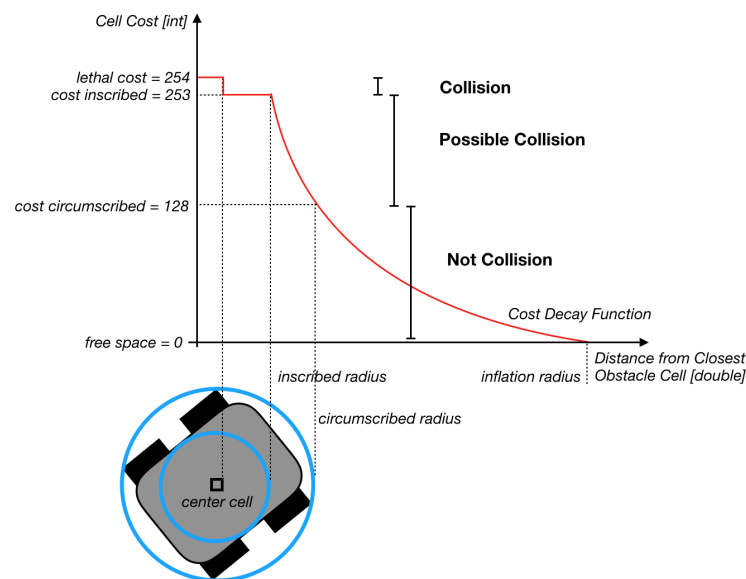


Figure 5.6: Inflation scheme to compute the occupancy distance cost.

The Trajectory planner [40] is the standard local planner provided in ROS, which implements the DWA algorithm. It allows to choose, among other options, how many samples for v and ω have to be considered to compute the admissible velocities and, also, whether the used robot is holonomic or not. If the robot is omnidirectional, in the algorithm two samples are added that take into account the velocities along y . The resulting planner is not satisfactory, because a non-holonomic behavior is still largely preferred. Moreover, it considers neither a maximal nor a minimal translation velocity: if a v_y term is considered, the module of the linear velocity is computed consequently, with the effect that the robot moves faster in diagonal than along a straight line. This is not a desirable behavior.

Another flaw of this planner is that it does not search for trajectories with a negative velocity, avoiding the robot to move backward. On the contrary, this backward movement is deployed as an escape maneuver (when the other trajectories are not feasible), regardless of the presence of obstacles. This resulted in the robot hitting the wall in most of the simulations.

DWA Local Planner

This local planner implements the DWA algorithm as the Trajectory Planner, but it differs in the way the trajectories are computed and it can also be used for holonomic robots [41]. Its pseudo-code is partially reported in Algorithm 5.3: the real algorithm is more complicated than this one, and only a simplified version of the main idea used to compute trajectories is proposed. The code can be integrated with the schemes in Figure 5.7.

The algorithm, in the first place, samples the set V_d according to the parameters of the configuration file (bullets in Figure 5.7), then tests whether the samples respect the velocity limits (conserving the green bullets and canceling out the red ones, Figure 5.7), specified in the same file as well. Once each trajectory sample is tested, its relative cost is computed and only the best score is retained.

The Dynamic Window is computed with the `sim_period`: the duration of the controller loop (i.e. the inverse of the `controller_frequency` parameter presented in 6.4.3).

Note that in Figure 5.7 two different cases for the sample test are proposed: in the right one, values greater than the maximal velocity in translation max_V are removed, while in the left one, samples smaller than both the maximal velocity in translation min_V and rotation min_omega are removed.

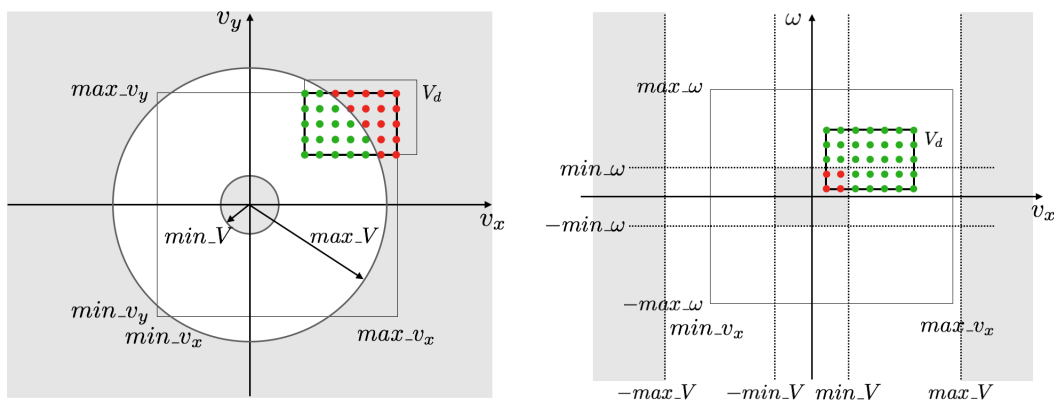


Figure 5.7: Scheme of the operating principle of the DWA Local Planner.

Thank to the simplicity of its computation, the DWA Local Planner generates smooth trajectories in a short period of time.

Algorithm 5.3 DWA Local Planner, pseudo code

```

1: function FIND_TRAJECTORY(robot.pose, robot.vel)
2:
3:   vel_samples  $\leftarrow$  initialize(robot.pose, robot.vel, vx_samples, vy_samples,  $\omega$ _samples)
4:
5:   best_traj  $\leftarrow$  null
6:   for i = 0 to vel_samples.size do
7:     vel_tested  $\leftarrow$  vel_samples[i]
8:     current_traj  $\leftarrow$  generate_trajectory(robot.pose, robot.vel, vel_tested)
9:     if current_traj better than best_traj then
10:       best_traj  $\leftarrow$  current_traj
11:
12:   return best_traj
13:
14: function INITIALIZE(robot.pose, robot.vel, vx_samples, vy_samples,  $\omega$ _samples)
15:
16:   // compute Dynamic Window velocities taking into account accelerations
17:   max_vx  $\leftarrow$  min(max_vx, robot_vx + robot_accx  $\times$  sim_period)
18:   max_vy  $\leftarrow$  min(max_vy, robot_vy + robot_accy  $\times$  sim_period)
19:   max_ $\omega$   $\leftarrow$  min(max_ $\omega$ , robot_ $\omega$  + robot_acc $\theta$   $\times$  sim_period)
20:   min_vx  $\leftarrow$  max(min_vx, robot_vx - robot_accx  $\times$  sim_period)
21:   min_vy  $\leftarrow$  max(min_vy, robot_vy - robot_accy  $\times$  sim_period)
22:   min_ $\omega$   $\leftarrow$  max(min_ $\omega$ , robot_ $\omega$  - robot_acc $\theta$   $\times$  sim_period)
23:
24:   // sampling interval
25:    $\delta$ _vx  $\leftarrow$  (max_vx - min_vx) / (vx_samples - 1)
26:    $\delta$ _vy  $\leftarrow$  (max_vy - min_vy) / (vy_samples - 1)
27:    $\delta$ _ $\omega$   $\leftarrow$  (max_ $\omega$  - min_ $\omega$ ) / ( $\omega$ _samples - 1)
28:
29:   vx_samp  $\leftarrow$  min_vx // first tested value for vx
30:   for ix = 0 to vx_samples do // loop through all x velocities
31:     vy_samp  $\leftarrow$  min_vy // first tested value for vy
32:     for iy = 0 to vy_samples do // loop through all y velocities
33:        $\omega$ _samp  $\leftarrow$  min_ $\omega$  // first tested value for  $\omega$ 
34:       for i $\theta$  = 0 to  $\omega$ _samples do // loop through all  $\theta$  velocities
35:         v_samp[0]  $\leftarrow$  vx_samp
36:         v_samp[1]  $\leftarrow$  vy_samp
37:         v_samp[2]  $\leftarrow$   $\omega$ _samp
38:         vel_samples.push(v_samp)
39:         // update velocity samples and close loops
40:          $\omega$ _samp  $\leftarrow$   $\omega$ _samp +  $\delta$ _ $\omega$ 
41:         vy_samp  $\leftarrow$  vy_samp +  $\delta$ _vy
42:         vx_samp  $\leftarrow$  vx_samp +  $\delta$ _vx
43:
44:   return vel_samples
45:
46: function GENERATE_TRAJECTORY(robot.pose, robot.vel, vel_tested)
47:
48:   v  $\leftarrow$   $\sqrt{\text{vel\_tested}[0]^2 + \text{vel\_tested}[1]^2}$  // compute linear velocity module
49:    $\varepsilon$   $\leftarrow$  10-4 // velocity bias
50:
51:   // test if the tested velocity respects the limits
52:   if v -  $\varepsilon$  > max_V then
53:     return invalid_traj
54:   if v +  $\varepsilon$  < min_V and |vel_tested[2]| +  $\varepsilon$  < min_ $\omega$  then
55:     return invalid_traj
56:   traj.vel  $\leftarrow$  vel_tested
57:   // evaluate the cost function for this trajectory
58:   traj.cost  $\leftarrow$  compute_cost(robot.pose, robot.vel, traj)
59:   return traj

```

Elastic Bands

An elastic band is a deformable collision-free path: the path originally created by a global planner is deformed in real time to a smoother and shorter path that ensures obstacle avoidance. The deformation is granted by an internal contraction force and an external repulsive force. The former is responsible for shortening the path, while the latter pushes the path far from obstacles.

The most famous implementation of such elastic bands is based on bubbles.

First, the \mathcal{Q}_{free} space of the robot is sampled in many local subsets, called bubbles, obtained by measuring the local freedom of the robot at the specific configuration.

$$B(q_c) = \{q : D(q_c - q) < \rho(q_c)\} \quad (5.1.11)$$

The bubble $B(\cdot)$ at the current configuration q_c is the set of all the configurations contained in a sphere of center q_c and radius the distance function $D(\cdot)$, bounded by $\rho(q_c)$. The function $\rho(q_c)$ computes the minimum distance of the obstacles from the center of the sphere, while the function $D(\cdot)$ computes the maximum distance that can be traveled by any point of the robot while it moves from configurations q to q' :

$$D(\Delta q) = \sqrt{\Delta q_x^2 + \Delta q_y^2} + r_{max} |\Delta q_\theta| \quad (5.1.12)$$

where r_{max} is the distance from the center of the robot to its farthest point.

To ensure that all the path, spanned by bubbles, is collision free, the bubbles at consecutive configuration samples must overlap (Figure 5.8).

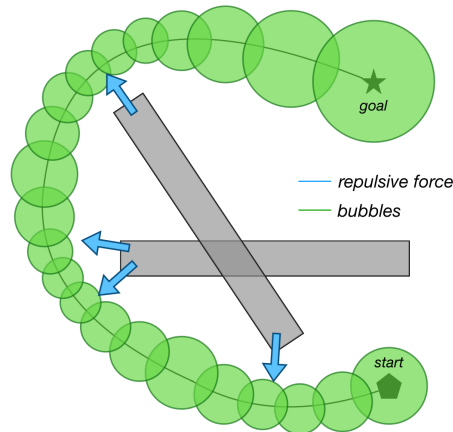


Figure 5.8: Operating principle of bubbles-based Elastic Bands.

Moreover, the bubbles are deformed according to their distance from the obstacles. If the planned trajectory makes the robot pass near an obstacle, the distance $\rho(q_c)$ decreases and, as a consequence, the resulting bubbles are smaller. In this case, given that consecutive bubbles must overlap, more bubbles will be generated. On the contrary, the bubbles will be bigger and lower in number when the obstacles are far (Figure 5.9).

The behavior of the bubbles allows to generate very smooth trajectories.

The simplicity of the computation of the bubbles is also the main drawback of this approach. Indeed, the robot shape is approximated as a circle of radius r_{max} . Thus, the approximation does not consider the orientation of the robot, so, even if this does not cause problems in obstacle avoidance, the robot fails in *passing through* obstacles: cross a door, pass between two obstacles, . . . , as showed in Figure 5.10.

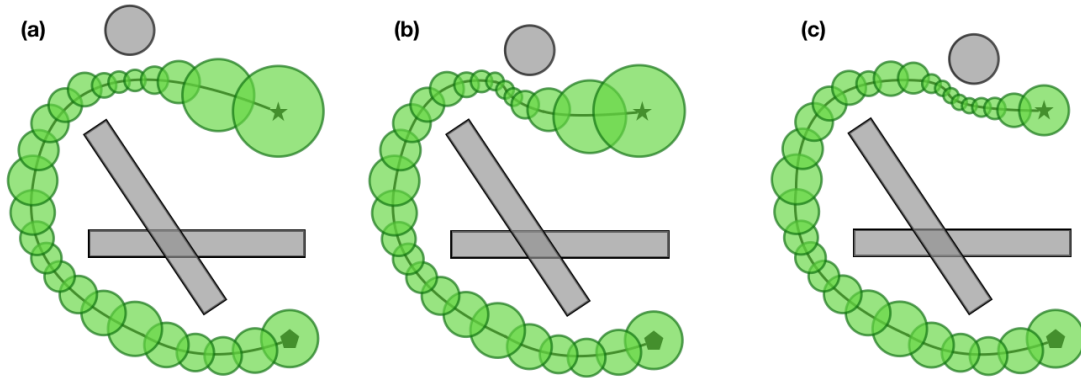


Figure 5.9: Bubbles behavior in consequence of a moving obstacle.

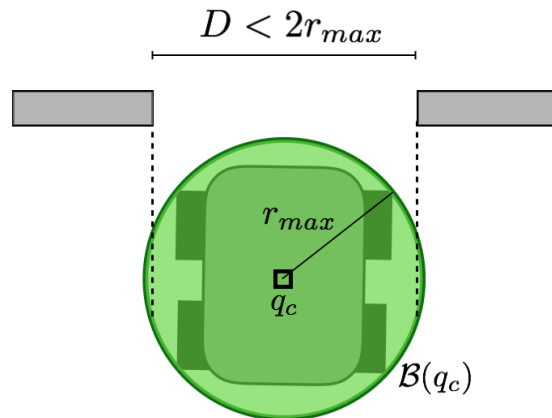


Figure 5.10: A case of failure for Elastic Bands: the circle approximating the robot is larger than the door width.

By relaxing the repulsion force to make the robot pass through a door, one can dramatically compromise the obstacle avoidance task and make the robot crash.

To summarize, both the Trajectory Planner and the EBand Local Planner presented some problems in simulation:

- Trajectory Planner is optimized for nonholonomic robots and the implemented extension for holonomic ones tests only two trajectories along the y axis, which is not satisfactory. The required behavior for such robots is to take into account motions along x and y . Moreover, the escaping maneuver may result in a collision if there is an obstacle behind the robot;
- EBand computes the bubbles by approximating the robot with a circle of radius r_{max} . This approximation has an unpleasant outcome: a robot that can pass through a door by moving on a straight line, with EBand cannot if the radius of the circle is greater than the width of the door.

To solve such problems, we decided to modify the more promising Trajectory Planner. Indeed, it seemed easier for us to implement a y displacement than to modify the obstacle avoidance behavior. For simplicity, our version of DWA is named Revisited-DWA.

One may wonder why we spent time to make our own local planner. The answer is both honest and sad: we found the improved version, DWA Local Planner, only when running some tests with the real robot. We have been blind enough not to see it, while searching for ROS packages implementing the Navigation Stack. Moreover, the Trajectory Planner

was set by default as local planner in the Ridgeback Navigation packages available in the tutorials.

Revisited DWA

To adapt the DWA algorithm, in Trajectory Planner, to our needs, we modified:

- the search space both by adding the v_y behavior and by changing the shape of the admissible velocities space V_a to an elliptical form;
- the way the goal distance is computed in the cost function.

The first change of adding the v_y behavior is justified by the fact that the Ridgeback is a holonomic mobile robot and hence we must be able to take advantage of this during local planning. To do that, we changed the source code of the `base_local_planner` by adding the bounds on the y velocity, acceleration and samples.

Our main contribution is, however, the modification in the shape of the admissible velocities space.

With the original rectangular shape², the robot exhibits the undesired feature of moving faster in diagonal than along the x (or y) direction only. We thought then that it would have been a more reasonable behavior for the robot to reach the limit velocity on the whole boundary of the window. In such a case we would have a circle centered in $(v_x = 0, v_y = 0)$. However, considering that:

- the negative v_x velocity lower than the positive one: $|max_v_x| > |min_v_x|$;
- the v_y velocity lower than v_x velocity: $|max_v_y| < |max_v_x|$;
- the module of the maximum and minimum for v_y equivalent (there is no reason why the robot should go faster to the right than to the left): $min_v_y = -max_v_y$.

the behavior finally chosen for the robot originates two half-ellipses for the (v_x, v_y) admissible set (Figure 5.11).

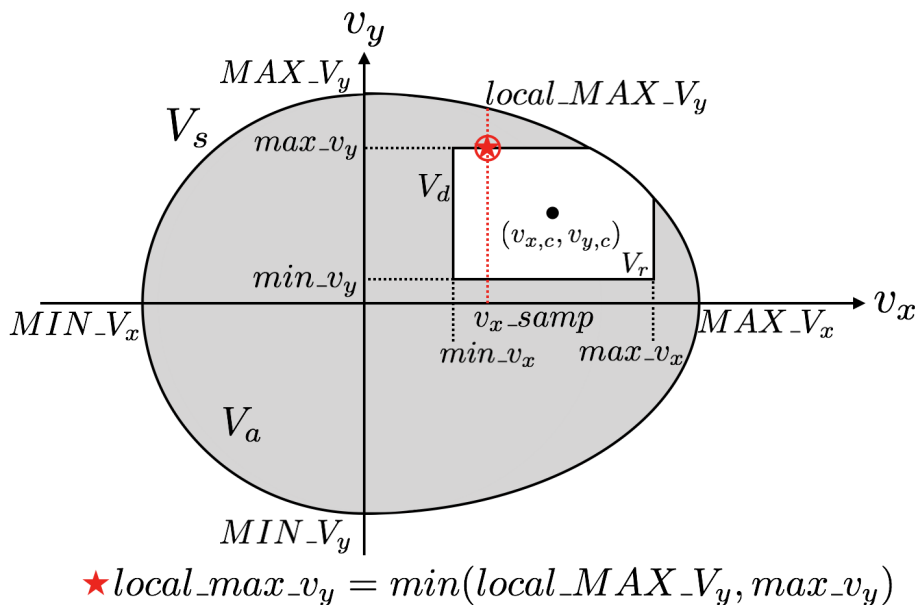


Figure 5.11: Two half-ellipses behavior for the Holonomic-DWA.

²Note that in the original algorithm, the space of all velocities is 2D, one dimension for $v = v_x$ and the other for $\omega = v_\theta$, by adding the velocity along y , the space is hence 3D. Only the 2D space (v_x, v_y) will be considered, because the θ dimension has not been modified.

We changed also the way the goal distance cost is computed, by considering the distance from the local goal *along* the global plan. In such a way, we assigned a lower cost to the cells that belong to the global plan, “forcing” the local planner to adhere to it as much as possible (left scheme in Figure 5.12). This is because the local goal lies on the boundaries of the local cost map, which means that it can be located behind an obstacle. In such a case the computed cost of the goal, in its original version, may originate what in the literature are called “*local minima*”, i.e., points, different from the goal, corresponding to which the cost function returns a local minimum. The outcome is that the local planner will command the robot to reach such points, with the final result of getting stuck (right scheme in Figure 5.12)³.

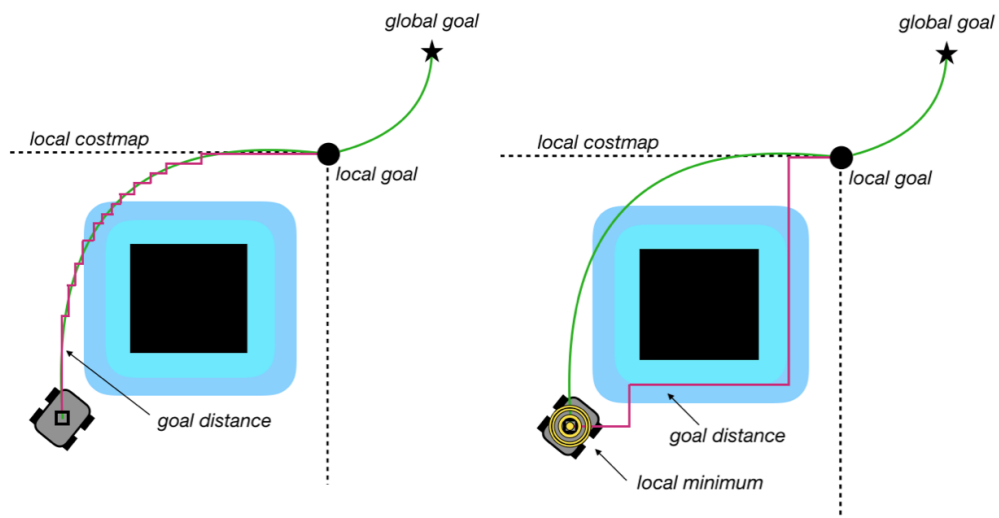


Figure 5.12: New goal distance for the Holonomic-DWA and the “*local minima*” case.

According to the notation of Figure 5.11, the pseudo code of the Holonomic-DWA can be written as in Algorithm 5.4. It is very important to understand it, so a few lines have been dedicated to explain it better, step by step:

- i*) from a configuration file (presented in Chapter 6) the maximum values for the velocities along x and y are retrieved: $\text{MAX_MIN_}V_x$, $\text{MAX_MIN_}V_y$;
- ii*) the velocity bounds for the dynamic window V_d , centered in the robot actual velocity $(v_{x,c}, v_{y,c})$, are computed taking into account these configurations velocities, the admissible velocities V_a (which allow the robot to stop in the goal position or in front of obstacles: Lines from 7 to 12 in Algorithm 5.4) and the accelerations of the robot (Lines from 27 to 32);
- iii*) the interval of velocities of the dynamic window is sampled according to the parameters $v_x, v_y_samples$ specified in the same configuration file as well. Each velocity sample is called v_x, v_y_samp ;
- iv*) for each sample two local maximal velocities along y are computed: the first one ($\text{local_MAX_}V_y$) is the value in the ellipses perimeter, while the second one ($\text{local_max_}v_y$) is the minimum between $\text{local_MAX_}V_y$ and the y bound of the dynamic window ($\text{max_}v_y$). The value $\text{local_max_}v_y$ is then taken as bound for the y component of the velocities in the set V_r (Lines from 45 to 51);
- v*) the instructions from *i*) to *ii*) are executed also for ω .

³This is true at least in theory, as we have not been able to simulate a failure case.

Algorithm 5.4 Revisited-DWA, pseudo code

```

1: function CREATE_TRAJECTORIES(robot.pose, robot.vel, robot.acc)
2:
3:   // initialization of each velocity value ( $v_x, v_y, \omega$ )
4:   max_velocities  $\leftarrow$  MAX_velocities
5:
6:   // velocity values that allow the robot to stop in the goal
7:   if goal_position is valid then
8:     goal_distance  $\leftarrow$   $\sqrt{(\text{goal}_x - \text{robot}_x)^2 + (\text{goal}_y - \text{robot}_y)^2}$ 
9:     max_vx  $\leftarrow$  min(MAX_Vx, goal_distance/sim_time)
10:    min_vx  $\leftarrow$  min(max(MIN_Vx, -goal_distance/sim_time), goal_distance/sim_time)
11:    max_vy  $\leftarrow$  min(MAX_Vy, goal_distance/sim_time)
12:    min_vy  $\leftarrow$  -MAX_Vy
13:
14:   // escaping maneuver: test with slower velocities and more samples
15:   if escaping then
16:     max_vx  $\leftarrow$  max_vx/2
17:     min_vx  $\leftarrow$  -max_vx
18:     max_vy  $\leftarrow$  max_vy
19:     min_vy  $\leftarrow$  -max_vy
20:     max_omega  $\leftarrow$  max_omega/2
21:
22:     vx_samples  $\leftarrow$  vx_samples $\times$ 2
23:     vy_samples  $\leftarrow$  vy_samples $\times$ 2
24:     omega_samples  $\leftarrow$  omega_samples $\times$ 2
25:
26:   // compute Dynamic Window velocities taking into account accelerations
27:   max_vx  $\leftarrow$  max(min(max_vx, robot_vx + robot_accx  $\times$  sim_period), min_vx)
28:   min_vx  $\leftarrow$  max(min_vx, robot_vx - robot_accx  $\times$  sim_period)
29:   max_vy  $\leftarrow$  max(min(max_vy, robot_vy + robot_accy  $\times$  sim_period), min_vy)
30:   min_vy  $\leftarrow$  max(min_vy, robot_vy - robot_accy  $\times$  sim_period)
31:   max_omega  $\leftarrow$  min(max_omega, robot_omega + robot_acc_theta  $\times$  sim_period)
32:   min_omega  $\leftarrow$  max(min_omega, robot_omega - robot_acc_theta  $\times$  sim_period)
33:
34:   // initialize best and comparative trajectory
35:   best_traj  $\leftarrow$  null
36:   comp_traj  $\leftarrow$  null

```

Algorithm 5.4 Revisited-DWA, pseudo code (*continued*)

```

37: // TRAJECTORY COMPUTATION: begin
38: // sampling interval
39:  $\delta_{v_x} \leftarrow (\max_{v_x} - \min_{v_x}) / (v_x\_samples - 1)$ 
40:  $\delta_{\omega} \leftarrow (\max_{\omega} - \min_{\omega}) / (\omega\_samples - 1)$ 
41:
42:  $v_x\_samp \leftarrow \min_{v_x}$  // first tested value for  $v_x$ 
43: for  $i_x = 0$  to  $v_x\_samples$  do // loop through all  $x$  velocities
44: // global  $y$  maximum of the ellipses with respect to the  $v_x$  sample
45: if  $v_x\_samp \geq 0$  then
46:  $local\_MAX_{V_y} \leftarrow MAX_{V_y} \times \sqrt{1 - (v_x\_samp / MAX_{V_x})^2}$ 
47: else
48:  $local\_MAX_{V_y} \leftarrow MAX_{V_y} \times \sqrt{1 - (v_x\_samp / MIN_{V_x})^2}$ 
49: // local  $y$  max and min of the dynamic window with respect to the  $v_x$  sample
50:  $local\_max_{v_y} \leftarrow \min(MAX_{V_y}, \max_{v_y})$ 
51:  $local\_min_{v_y} \leftarrow \max(-MAX_{V_y}, \min_{v_y})$ 
52:
53:  $\delta_{v_y} \leftarrow (local\_max_{v_y} - local\_min_{v_y}) / (v_y\_samples - 1)$ 
54:  $v_y\_samp \leftarrow local\_min_{v_y}$ 
55: for  $i_y = 0$  to  $v_y\_samples$  do // loop through all  $y$  velocities
56:  $\omega\_samp \leftarrow \min_{\omega}$ 
57: for  $i_{\theta} = 0$  to  $\omega\_samples$  do // loop through all  $\omega$  velocities
58: // if the tested velocity is too small, ignore it
59: if  $|v_x\_samp| \leq MAX_{V_x} / 100$  and  $|v_y\_samp| \leq MAX_{V_y} / 100$  and  $|\omega\_samp| \leq$ 
 $MAX_{\omega} / 100$  then
60: continue
61:
62: // generate the trajectory with these velocity values
63:  $comp\_traj \leftarrow generate\_trajectory(robot.pose, robot.vel, robot.acc, v_x\_samp, v_y\_samp, \omega\_samp,$ 
 $comp\_traj)$ 
64:
65: // compare the trajectories and take the best one
66: if  $comp\_traj$  better than  $best\_traj$  then
67:  $best\_traj \leftarrow comp\_traj$ 
68:
69: // update velocity samples and close loops
70:  $\omega\_samp \leftarrow \omega\_samp + \delta_{\omega}$ 
71:  $v_y\_samp \leftarrow v_y\_samp + \delta_{v_y}$ 
72:  $v_x\_samp \leftarrow v_x\_samp + \delta_{v_x}$ 
73: // TRAJECTORY COMPUTATION: end
74:
75: return  $best\_traj$ 

```

Particular attention needs to be paid to the variables `sim_time` and `sim_period`: the former is the duration in seconds of the forward simulation, while the latter, as previously described, is the duration of the controller loop. The first one is used to slow the velocities when approaching to the goal, while the second one is involved in the computation of the Dynamic Window.

To sum up, we preserve the number of velocity samples, in the (x, y) plane, but we shrink the velocity set in the form of two ellipses by computing in every loop the local limits for v_y . The DWA Local Planner, instead, calculates every time the same number of samples and then neglects those that exceed some chosen limits.

In terms of performance, since we found the DWA Local Planner, we did not try to optimize our solution, by removing from the Trajectory Planner all the unused functions and computations. The consequence was that, occasionally, the robot presented some jerks in the motion. These jerks are justified by the fact that the robot waits for the command input for some time, but if the control loop exceeds this time value, the robot stops. An-

other main difference between the DWA Local Planner and our version is the possibility for the former to visualize the cost functions.

5.1.3 Frontier Exploration

The frontier exploration is a frontier-based approach for autonomous exploration [43]. A frontier is defined as a region on the boundaries between known and unknown space. This approach is used because in most of the mobile robotics applications, the environment is mapped by the human before letting the robot navigate it autonomously. This is done, for example, by tele-operating the robot either with a controller (for PC games or console) or the computer keyboard. The human control in tele-operation is both an advantage, because it is the human who decides where to move the robot, and a disadvantage because, from a technical point of view, it requires a camera to let the human see what the robot sees. Moreover, tele-operation requires continuous control, while this approach, once it is launched, it is autonomous from the beginning to the end.

In [43], by the term *exploration* it is, hence, meant to move through an unknown environment while building the map for possible navigation applications.

The mathematical idea that lies behind this approach is formulated in [34] as well, where *exploration* addresses the problem of maximizing the cumulative information the robot has about each grid cell of a grid map. In the case of frontier-based exploration, the problem is simplified as a cell is either explored or not and hence the robot is pushed to move towards the unexplored area. Also, as previously discussed, an explored cell can be either free or an obstacle.

Once the user manually selects an area to explore, the detection of frontiers is made through three steps:

- i*) all the free cells adjacent to an unknown cell are labeled as frontier edge cells;
- ii*) adjacent edge cells are grouped into frontier regions;
- iii*) if these frontier regions contain more than a certain amount (that the user can impose) of edge cells, they are marked as frontiers.

The criteria to group the frontier regions are described below and depicted in Figure 5.13:

- *centroid*: the geometrical center of the edges in a frontier regions is chosen as a frontier (whether it is in an unexplored area or not);
- *middle*: the closest edge to the robot (with respect to the Euclidean distance) in the frontier region is regarded as frontier⁴;
- *closest*: the closest edge to the robot (with respect to δ -connected neighborhood) in the frontier region is regarded as frontier (from a graphical point of view there is no visible difference between closest and middle).

Once these frontiers are built, the algorithm selects the closest one to the robot and sends it to the global planner as a goal. Once it is reached, or after a fixed amount of time, the frontiers are updated and the global planner is assigned a new goal. The algorithm, together with the exploration and hence the mapping process, stops when all the selected area is explored.

Changes have been made also in the frontier exploration code regarding:

1. how far a frontier should be from the robot;
2. how far a frontier should be from an obstacle.

When the robot spawns in a map (when the algorithms are launched and the robot initialized), the area beneath it is marked as unexplored. As a result, if the selected area for

⁴This denomination is quite inconsistent with its effect, but there is no error: it is the used one.

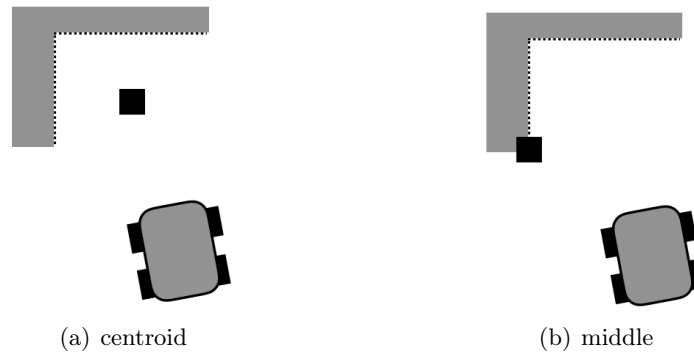


Figure 5.13: Scheme of the criteria for grouping frontier regions (dashed line) in frontiers (black square).

exploration includes the robot position (and it usually does), this area is hence a frontier and also the closest to the robot (Figure 5.14a). The problem is that the robot cannot explore this area, so it gets stuck and does not move at all.

To solve this problem, we prevented the algorithm from considering the area including the robot footprint as a candidate frontier region. The area to exclude is a circle whose radius is chosen by the user and centered in the robot position. To be effective the circle has to be bigger than the robot.

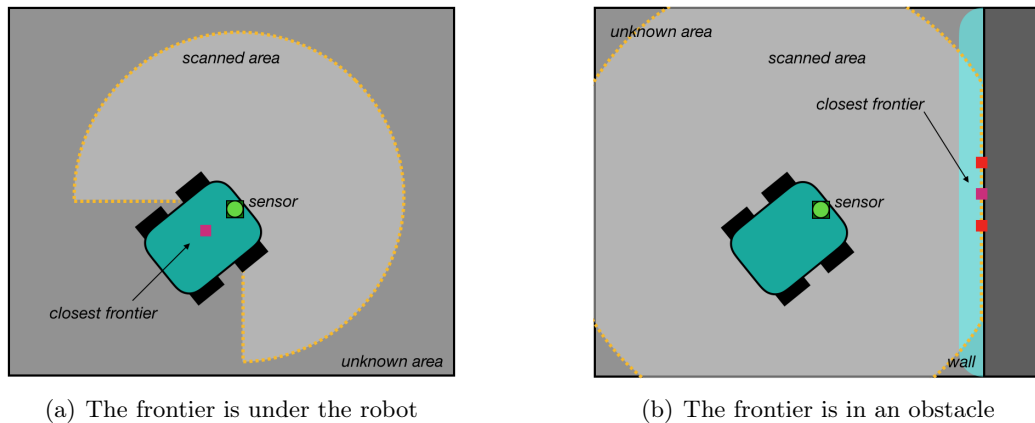


Figure 5.14: Cases when the frontier exploration fails.

Concerning the second point, sometimes because of the probabilistic model of both the sensor and the map, there are discontinuities in the obstacle space that are hence marked as unexplored cells. As a consequence, some of them were marked as frontiers but, being inside an obstacle, the global planner could not reach them. Thus without a global plan, the exploration failed (Figure 5.14b).

To solve this problem, we changed the way frontier edges are chosen: to be an edge, a cell does not have to be within a chosen distance from a occupied cell.

A scheme of the resulting behavior is depicted in Figure 5.15, while the respective pseudo code is reported in algorithm 5.5, that is responsible for deciding whether the input cell is a candidate frontier or not.

In algorithm 5.5, we added the two tests, TEST 1 and 2, which are based on the parameters `min_frontier_dist` and `min_frontier_clear_dist` defined in the parametrization file presented in Section 6.4.6.

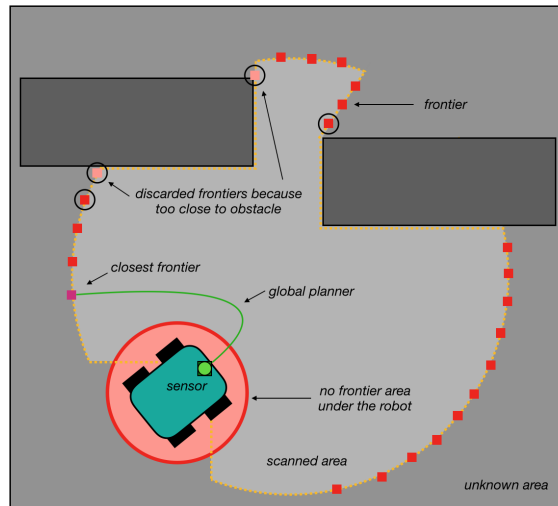


Figure 5.15: The definitive behavior of the frontier exploration algorithm.

Algorithm 5.5 isFrontierCell, pseudo code

```

1: function IS_FRONTIER_CELL(cell)
2:
3:   // test if the cell is unknown or already a frontier
4:   if map(cell) is unknown or frontier_cell then
5:     return false
6:
7:   // transform the cell index in a map point
8:   sizex ← costmap.sizex
9:   mx ← cell%sizex
10:  my ← cell/sizex
11:
12:  // transform the map point in a world point
13:  wx, wy ← null
14:  costmap.mapToWorld(mx, my, wx, wy)
15:
16:  // TEST 1: frontier far from robot of a circle of radius min_frontier_dist
17:  dist_robot ← √((robot.x - wx)2 + (robot.y - wy)2)
18:  if dist_robot < min_frontier_dist then
19:    return false
20:
21:  // TEST 2: frontier far from walls in a neighborhood of radius min_frontier_clear_dist
22:  dist_wall ← min_frontier_clear_dist/costmap.resolution
23:  for all idx in nhooDist(idx, dist_wall) do
24:    if map(idx) is obstacle then
25:      return false
26:
27:  // TEST 3: a frontier cell should have at least one free cell in 4 connected neighborhood
28:  for all idx in nhoo4(idx) do
29:    if map(idx) is free_space then
30:      return true // thanks to the previous tests the frontier is now accepted
31:  return false

```

5.2 Navigating a given map

To navigate a given map, the robot uses the sensor measurements and odometry motion to localize itself in the map and hence recover its position. Then, these data are filtered with the Augmented Monte Carlo filter, an extension of the filter presented in Section 4.2.

5.2.1 Odometry Motion Model

Odometry is obtained by integrating the data of the wheel encoder in order to estimate the change of robot position over time. Since odometry is only available once the robot has moved, the information provided can be used for localization but not for planning.

The odometry motion model uses the relative motion information which is provided by robot internal odometry: if the robot moves from x_{t-1} to x_t in the time interval $(t-1, t]$, the odometry measures a motion from $\bar{x}_{t-1} = [\bar{x}, \bar{y}, \bar{\theta}]^T$ to $\bar{x}_t = [\bar{x}', \bar{y}', \bar{\theta}']^T$.

The bar notation indicates that the measurement provided by the odometry is made with respect to a reference frame internal to the robot. The transformation between the robot internal frame and the world frame is only an estimate and hence the difference between \bar{x}_{t-1} and \bar{x}_t can only be used as an estimator for the difference of the true poses x_{t-1} and x_t .

The motion information is hence:

$$u_t = \begin{bmatrix} \bar{x}_{t-1} \\ \bar{x}_t \end{bmatrix} \quad (5.2.1)$$

To retrieve the odometry, this motion has to be split in the sequence of a translation (called δ_{trans}) and a rotation (called δ_{rot}) starting from the estimate initial heading (called $\delta_{bearing}$), as depicted in Figure 5.16a. Thus, each u_t is parametrized by a unique sequence vector $[\delta_{rot}, \delta_{trans}, \delta_{bearing}]^T$.

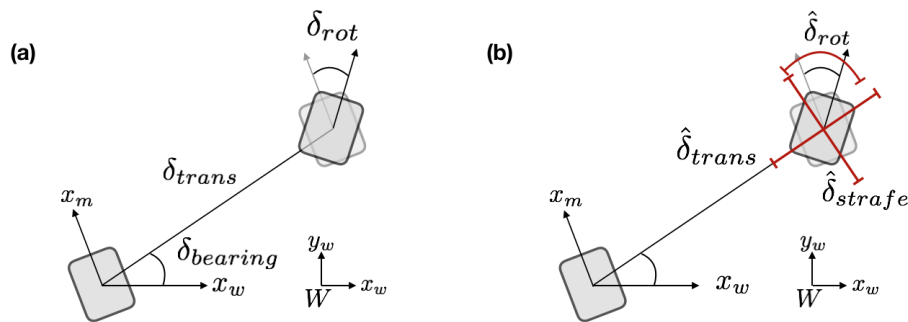


Figure 5.16: Odometry model.

A further hypothesis adopted by the odometry motion model is that each one of these three parameters is corrupted by independent noise.

The pseudo code 5.6 for the Odometry Motion Model Sample algorithm computes a sample from $p(x_t|u_t, x_{t-1})$, which is hence used in localization based on particle filters. The returned sample is nothing but a random guess, distributed according to $p(x_t|u_t, x_{t-1})$. This model is not provided in [34] because it is an adapted version for omnidirectional robots, which are not treated in the book.

The estimate of the particle representing the pose x_t is calculated from x_{t-1} by integrating the Gaussian errors $\hat{\delta}_{trans}$, $\hat{\delta}_{strafe}$ and $\hat{\delta}_{rot}$ as schematically represented in Figure 5.16b.

The variables α_1 to α_5 are robot parameters that specify the noise affecting the motion. The effects of changing these parameters are presented in Figure 5.17: in (a) the parameters have typical values, while in (b), (c) and (d) the errors in translation, strafe and rotations respectively are surprisingly large.

Finally, by taking into account also the map m , the model probability becomes $p(x_t|u_t, x_{t-1}, m)$. Hence, by adapting the algorithm 5.6, the model computes the likelihood that a robot in a map m reaches the pose x_t from x_{t-1} via the motion u_t . The hypothesis that the distance

Algorithm 5.6 Odometry Motion Model Sample, pseudo code

```

1: function SAMPLE_MOTION_MODEL_ODOMETRY( $u_t, x_{t-1}$ )
2:
3:    $\delta_{trans} \leftarrow \sqrt{(\bar{x} - \bar{x}')^2 + (\bar{y} - \bar{y}')^2}$ 
4:    $\delta_{rot} \leftarrow \bar{\theta}' - \bar{\theta}$ 
5:    $\delta_{bearing} \leftarrow \text{atan2}(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \bar{\theta} + \theta$ 
6:
7:   // standard deviations
8:    $\sigma_{trans} \leftarrow \sqrt{\alpha_3 \delta_{trans}^2 + \alpha_1 \delta_{rot}^2}$ 
9:    $\sigma_{rot} \leftarrow \sqrt{\alpha_2 \delta_{trans}^2 + \alpha_4 \delta_{rot}^2}$ 
10:   $\sigma_{strafe} \leftarrow \sqrt{\alpha_5 \delta_{trans}^2 + \alpha_1 \delta_{rot}^2}$ 
11:
12:  // estimate with normal distribution
13:   $\hat{\delta}_{trans} \leftarrow \delta_{trans} + \text{gaussian}(\sigma_{trans})$ 
14:   $\hat{\delta}_{rot} \leftarrow \delta_{rot} + \text{gaussian}(\sigma_{rot})$ 
15:   $\hat{\delta}_{strafe} \leftarrow 0 + \text{gaussian}(\sigma_{strafe})$ 
16:
17:  // compute a new particle
18:   $x' \leftarrow x + \hat{\delta}_{trans} \cos(\delta_{bearing}) + \hat{\delta}_{strafe} \sin(\delta_{bearing})$ 
19:   $y' \leftarrow y + \hat{\delta}_{trans} \sin(\delta_{bearing}) - \hat{\delta}_{strafe} \cos(\delta_{bearing})$ 
20:   $\theta' \leftarrow \theta + \hat{\delta}_{rot}$ 
21:
22:  return  $x_t = [x', y', \theta']^T$ 

```

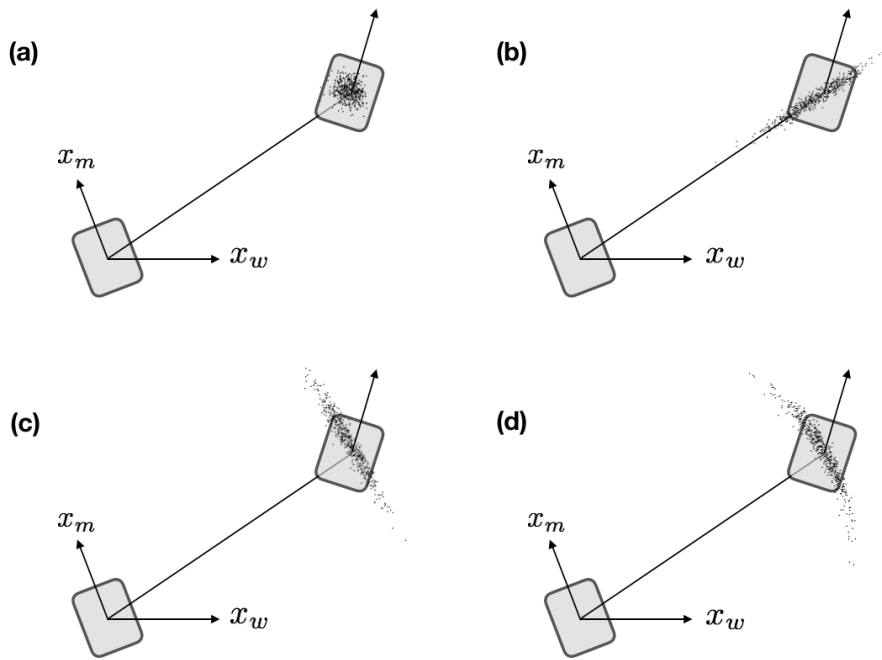


Figure 5.17: Uncertainties in the robot pose due to motion noise.

x_{t-1} and x_t is small allows to factorize the posterior:

$$p(x_t | u_t, x_{t-1}, m) = \eta \frac{p(x_t | u_t, x_{t-1}) p(x_t | m)}{p(x_t)} \quad (5.2.2)$$

In that form, the posterior is given by the normalized product between the odometry motion model estimate and the likelihood of the pose in the map. The pseudo code for the odometry motion model including a map is reported in Algorithm 5.7.

To summarize, the reason why this motion model is not used in FastSLAM is because it is a random computation, which means that each time this algorithm is executed, it randomly returns a pose x_t distributed according to $p(x_t | u_t, x_{t-1})$. On the other hand, the motion

Algorithm 5.7 Odometry Motion Model Sample with Map, pseudo code

```

1: function SAMPLE_MOTION_MODEL_MAP( $u_t, x_{t-1}$ )
2:
3:   do
4:      $x_t \leftarrow$  sample_motion_model_odometry( $u_t, x_{t-1}$ )
5:      $\pi \leftarrow p(x_t|m)$ 
6:   until  $\pi > 0$ 
7:
8:   return  $\langle x_t, \pi \rangle$ 

```

model used in FastSLAM is based on EKF, which uses a deterministic odometry model to compute the pose and update its mean and covariance. In this case the computation is deterministic, which means that the outcomes are the same when the calculation is repeated.

These are the results used for localization as it is presented in the next section.

5.2.2 Augmented MCL

As commented in Section 4.2, this augmented version is able to recover the position of the robot after a localization failure. This problem is solved with a heuristic: new particles are randomly added to the particle set. Mathematically, this responds to the so called *kidnapped robot problem*: it is assumed that the robot can be kidnapped with a small probability, which consequently generates new random samples.

These samples are added proportionally to an importance factor that approximates the probability of sensor measurements:

$$\frac{1}{M} \sum_{m=1}^M w_t^{[m]} \approx p(z_t | z_{1:t-1}, u_{1:t}, m) \quad (5.2.3)$$

Adding too many particles makes the localization task impossible, while adding too few, has no substantial effect on the performance of the algorithm. So, the estimate is smoothed in two different time steps, originating a short-term and a long-term average. To control these averages, two weights are introduced: ω_{slow} and ω_{fast} . They are both exponential filters of the weights over a long time and a short time, respectively. The decay rates of the filters are controlled by α_{slow} and α_{fast} .

The Augmented MCL algorithm can be written in pseudo code 5.8 and it behaves for the most part like its basic version. The main differences are:

- in Line 10 the empirical probability of the sensor measurement is calculated;
- in Line 12 and 13 short-term and long-term averages are updated considering their decay rates: $0 \leq \alpha_{slow} \ll \alpha_{fast}$;
- in Line 16, in the resampling process, a new sample is added with probability $\max\{0.0, 1.0 - w_{fast}/w_{slow}\}$. In such a way, if the measurement likelihood suddenly degenerates, the number of added random samples increases.

A comparison with the standard MCL algorithm is presented in Figure 5.18.

To conclude, in this chapter the algorithms adopted for exploration and navigation in grid-based maps have been presented. As far as the former is concerned, the Frontier Exploration algorithm computes, based on the actual knowledge of the map and the localization of the robot (SLAM), goal points that the robot has to reach to maximize the knowledge of the map. To move, the robot uses a global and a local planner. The global

Algorithm 5.8 Augmented MCL algorithm, pseudo code

```

1: function AUGMENTED_MCL( $\mathcal{X}_{t-1}, u_t, z_t, m$ )
2:
3:   static  $w_{slow}, w_{fast}$ 
4:    $\bar{\mathcal{X}}_t, \mathcal{X}_t \leftarrow \emptyset$ 
5:
6:   for  $m = 1$  to  $M$  do
7:      $x_t^{[m]} \leftarrow \text{sample\_motion\_model}(u_t, x_{t-1}^{[m]})$ 
8:      $w_t^{[m]} \leftarrow \text{measurement\_model}(z_t, x_t^{[m]}, m)$ 
9:      $\bar{\mathcal{X}}_t \leftarrow \bar{\mathcal{X}}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$ 
10:     $w_{avg} \leftarrow w_{avg} + \frac{1}{M} w_t^{[m]}$ 
11:
12:     $w_{slow} \leftarrow w_{slow} + \alpha_{slow}(w_{avg} - w_{slow})$ 
13:     $w_{fast} \leftarrow w_{fast} + \alpha_{fast}(w_{avg} - w_{fast})$ 
14:
15:    for  $m = 1$  to  $M$  do
16:      with probability  $\max\{0.0, 1.0 - w_{fast}/w_{slow}\}$  do
17:        add random pose to  $\mathcal{X}_t$ 
18:      else
19:        draw  $i$  with probability  $\propto w_t^{[i]}$ 
20:        add  $x_t^{[i]}$  to  $\mathcal{X}_t$ 
21:
22:    return  $\mathcal{X}_t$ 

```

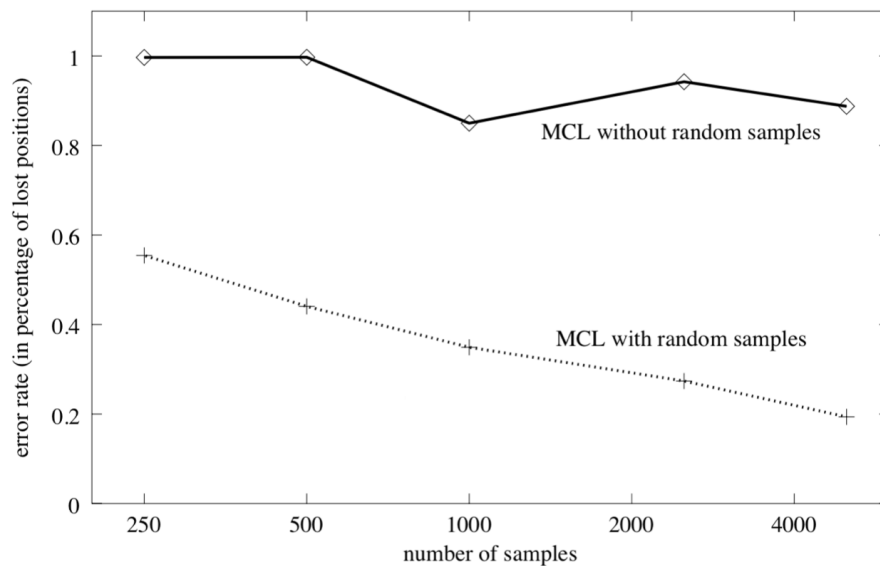


Figure 5.18: Comparison between Augmented and standard MCL (source: [34]).

planner uses Dijkstra’s algorithm to compute the best path, in terms of length and obstacle avoidance, from the robot position to the goal, while the local planner exploits the Dynamic Window Approach to find the best trajectories that the robot has to follow locally.

As far as the navigation task is concerned, it is supposed that a map is given to the robot. In such a case, the Augmented Monte Carlo particle filter is responsible for localizing the robot in the map, using the odometry model. Differently from the standard Monte Carlo method, the augmented version is able to recover from localization failures, by randomly adding new particles to the particle set. The number of particles added is proportional to an importance factor that approximates the probability of sensor measurements. This improvement reduces the error rate of the robot localization.

Simulating and Testing Ridgeback with ROS

Note: the work presented in this chapter was done in close collaboration with Hugo Bouvier Lambert, as we both worked on the same project at the same time.

Simulation is an important step to validate the algorithms before testing them in practice on a real robot. In this chapter, the behavior of the algorithms discussed in Chapters 4 and 5 and the results obtained, both in simulation and in practice, will be showed, together with the problems faced.

The robot simulations and applications are most of the times made with ROS, which is the subject of the next section.

6.1 Robot Operating System



Figure 6.1: ROS logo.

Robot Operating System (ROS) is robotics middleware: an open-source, meta-operating system that provides tools for communicating with robots, allowing to control them and to handle the interchange of informations and data [44].

ROS is organized in three levels, also called concepts: the Filesystem level, the Computation Graph level, and the Community level. Below, some concise descriptions of ROS concepts from [45] are reported.

Filesystem level. The filesystem level concepts include ROS resources that are available in a computer, among them, the most important are the **Packages**. Packages are the main tool to organize and store software. The most important features contained in packages are ROS runtime processes (nodes), libraries or configuration files.

Computation Graph level. The Computation Graph is the peer-to-peer network of ROS processes that are sharing and elaborating data together. There are several Computation Graph concepts of ROS, all of them providing data to the Graph in different ways. The most important are:

- **Nodes.** Nodes are processes that perform computation. A robot control system usually comprises many nodes, each of them specialized in a given task (for example in controlling the wheels of the robot, in localization, in path planning). A ROS node is written in **C++** or **Python**.
- **Master.** The ROS Master provides an organized structure of dependencies among nodes and other concepts in Computation Graph. Without the Master, nodes would not be able to find each other, exchange messages, or invoke services.

- **Messages.** Nodes communicate with each other by exchanging messages. A message is simply a data structure, comprising typed fields.
- **Topics.** Messages are routed via a transport system with publish/subscribe semantics, proper of a middleware. This semantic messaging pattern is used for asynchronous communication among processes, i.e. the nodes. A node that sends a message is called *publisher*, while the receiver is called *subscriber*. Publishers do not send messages directly to the subscribers but they use, instead, a message broker, called *topic*. The topic performs a filtering action, in the first place, where it selects only the messages to forward to the subscribers, then it also performs store and forward functions to route the messages from publishers to subscribers. The topic name is used to identify the content of the message. A node that is interested in a certain kind of data will *subscribe* to the appropriate topic and hence retrieve the data. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics.
- **Services.** A Service provides the node request/reply interactions, which are often required in a distributed system. Services are defined by a pair of message structures: one for the request and one for the reply.

Schematically, the operating principle of ROS can be summarized as shown in Figure 6.2.

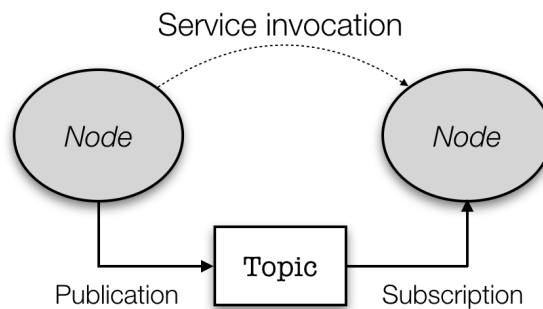


Figure 6.2: ROS operating principle scheme.

Community level. The ROS Community Level concepts are ROS resources that enable separate communities to exchange software and knowledge. Among them, **ROS Wiki** [46] is the main forum for documenting information about ROS. Anyone can sign up for an account and contribute with their own documentation, provide corrections or updates, write tutorials, and more.

6.2 Simulation set up

To set up a mobile robotics simulation in ROS, one needs several components:

- one or more simulation programs;
- one or more simulated robots that integrate the ROS Navigation Stack;
- one or more launch files that specify the nodes to launch, together with their parameter configurations.

Following the previous order, each of these components will be presented later in this chapter.

6.2.1 The simulation environment

Since a robot has to move in an environment, a map has to be provided to some simulator. The simulators chosen to set up this simulation are Gazebo and RViz (Figure 6.3).

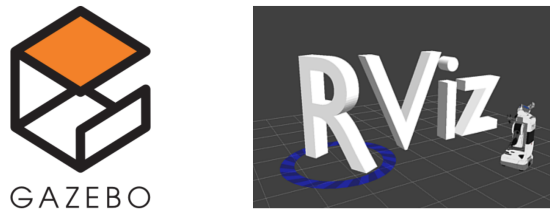


Figure 6.3: Gazebo and RViz logos.

Gazebo is a 3D dynamic simulator with the ability of accurately and efficiently simulating populations of robots in complex indoor and outdoor environments [47]. Gazebo offers physics simulation at a high degree of fidelity, a suite of sensors, and interfaces for both users and programs. It is, thus, in charge of simulating the robot and map models and of transmitting the odometry of the robot and the sensor data. Gazebo replaces the real robot and environment, which means that when the real robot is used, Gazebo is not.

Gazebo was used to design the simulation map, namely one of the multiple Fika Room (a cafeteria, a room with tables and coffee machines where one can have lunches and coffee breaks) at ABB Corporate Research Center, or at the CobotLab where the experiments were carried out. Once the map is designed, the simulated model of the Ridgeback has to be imported. The package for simulating the Ridgeback in Gazebo can be downloaded from Git Hub, by writing in the shell prompt the following command:

```
$ git clone https://github.com/ridgeback/ridgeback_simulator.git
```

RViz (ROS visualization) is a 3D visualizer for displaying sensor data and state information from ROS. Using RViz it is possible to visualize the Ridgeback current configuration on a virtual model of the robot. It is also possible to display live representations of sensor values coming over ROS Topics. For example, it recovers odometry and sensor data published by Gazebo. The package for simulating the Ridgeback in RViz can be downloaded from Git Hub, by writing in the shell prompt the following command:

```
$ git clone https://github.com/ridgeback/ridgeback_desktop.git
```

Finally, it is also necessary to download the main package for Ridgeback, the main ROS implementation of the robot, where the type of messages that are sent, the controller, and the navigation algorithms are described, and the `urdf` (Universal Robot Description Format) description of the robot is given: the standard ROS XML representation of the robot model (kinematics, dynamics, sensors).

```
$ git clone https://github.com/ridgeback/ridgeback.git
```

6.3 ROS Navigation Stack

This section is taken from the ROS Navigation Stack Wiki [48].

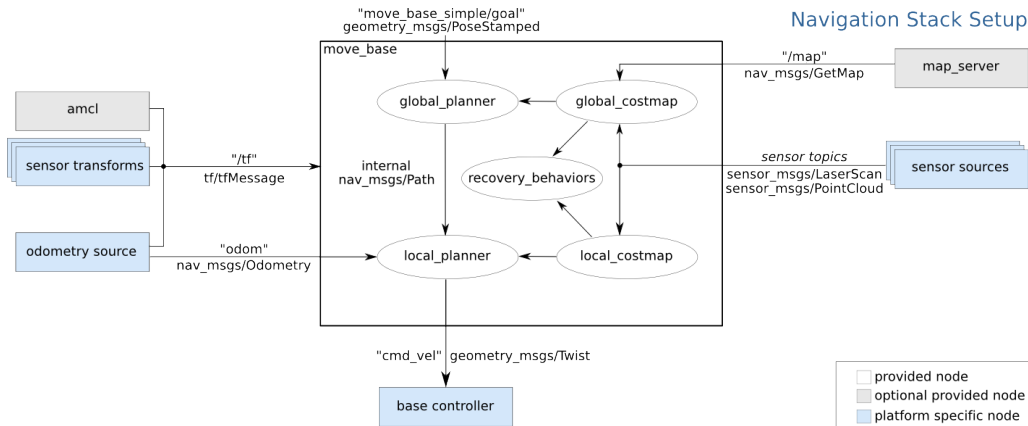


Figure 6.4: ROS Navigation Stack overview.

The Navigation Stack assumes that the robot is configured in a particular way in order to run. The diagram above (Figure 6.4) gives an overview of this configuration. Both the required white components and the optional gray ones are already implemented, while the blue features must be created for each robot platform. The Navigation Stack can be downloaded with:

```
$ sudo apt-get install ros-kinetic-navigation
```

or:

```
$ git clone https://github.com/ros-planning/navigation.git
```

6.3.1 Robot setup

Transform Configuration (sensor transforms) The navigation stack requires that the robot publishes information about the relationships between coordinate frames using the topic `/tf`.

Sensor Information (sensor sources) The navigation stack uses information from sensors to avoid obstacles in the surrounding environment; it assumes that these sensors are publishing either `sensor_msgs/LaserScan` or `sensor_msgs/PointCloud` messages over ROS.

Odometry Information (odometry source) The navigation stack requires that odometry information is published using the topic `/tf` and the `nav_msgs/Odometry` message.

Base Controller (base controller) The navigation stack assumes that a trajectory planner can send velocity commands using a `geometry_msgs/Twist` message, in the base coordinate frame of the robot, on the “`cmd_vel`” topic. This means that there must be a node subscribing to the “`cmd_vel`” topic that is capable of taking $[v_x, v_y, \omega]^T$ (`cmd_vel.linear.x`, `cmd_vel.linear.y`, `cmd_vel.angular.z`) velocities and converting them into motor commands to be sent to a mobile base.

Mapping (map_server) The navigation stack does not require a map to operate. One can therefore distinguish two separate cases:

1. the robot is *building* the map (mapping process);
2. the robot is *navigating* the map.

In the first case, the robot does not know the map, therefore no map is given to the Navigation Stack. In this mapping process, the robot uses its laser sensor to detect the obstacles and to build the map (SLAM).

In the second case, the robot has already built a map (or a map has been given), and it is asked to navigate in it to accomplish other tasks (for example, to go to some defined place in the map).

6.3.2 Navigation Stack setup

Cost Map configuration (local_costmap & global_costmap) The navigation stack uses two cost maps to store information about obstacles in the world. One is used for global planning and the other is used for local planning, both for obstacle avoidance. The configuration parameters for both the cost maps are written in three different YAML files: `costmap_common.yaml`, `costmap_global.yaml` and `costmap_local.yaml`.

These files will be better detailed later.

The planners and the cost maps, grouped under the name `move_base` in Figure 6.4, have a specific parametrization set, which is the subject of the next section.

6.4 Parametrization

In this section the parametrization made for the simulation will be presented. Each of the elements presented in Chapter 5 has a dedicated parameter set which is written in a `.yaml` file, in order to make the remapping from ROS easier. While the greatest part of them was available by default, a couple was added in order to be able to modify part of the code. The parameters presented have been obtained by making the robot complete the task of exploring the CobotLab in simulation, in the first place; then, they have been tested on the real robot and therefore updated. Finally, they have been tested again through simulations to verify their correctness.

Only the parameters that are described and detailed are reported, the others are listed in their corresponding `.yaml` file in Appendix F.

6.4.1 Move Base

Move Base includes a list of parameters describing the general behavior of the planners and the controller [49], the most important of them are reported in snippet 6.1, while the complete set is in `planner.yaml` in Appendix F.1.

Listing 6.1: Move Base parametrization

```

controller_frequency: 5.0 #The rate in Hz at which to run the control loop and send
    velocity commands to the base.
controller_patience: 15.0 #How long the controller will wait in seconds without
    receiving a valid control before space-clearing operations are performed.

planner_frequency: 2.0 #The rate in Hz at which to run the global planning loop. If the
    frequency is set to 0.0, the global planner will only run when a new goal is
    received or the local planner reports that its path is blocked.
5 planner_patience: 5.0 #How long the planner will wait in seconds in an attempt to find a
    valid plan before space-clearing operations are performed.

```

The frequency depends on the calculation power of the computer and on the quality of the wireless communication. The chosen values are, up to now, the best compromise found, to have a good motion and no computation loops missed for controller and planner. Increasing or decreasing these values results in a jerk motion for different reasons: firstly, because the time is not enough to complete the computations, secondly, because the controller ends its task and it has to wait for the following one.

Concerning the `controller/planner_patience`, they determine how long the controller or the planner have to wait before restarting the computations, when the robot is stuck.

6.4.2 The Global Planner

The global planner used is called `NavFn` and it provides a fast interpolated navigation function. As specified above, `NavFn` needs a cost map to find a minimum cost path [50]. The navigation function is computed with Dijkstra's algorithm presented in Section 5.1.1.

Its parametrization is written in the planners file, fully reported in Appendix F.1. In this file all the parameters for the Navigation Stack `move_base`, `global_planner` and `local_planner` are stored.

The parametrization of the global planner is very simple and, therefore, it does not allow so much freedom, as other algorithms in ROS do. The truly useful parameter is `allow_unknown`, which allows the planner to compute a path even when it ends in an unexplored zone.

Listing 6.2: Global Planner parametrization

```
NavfnROS:
  allow_unknown: true #Specifies whether or not to allow navfn to create plans that
                    traverse unknown space.
  default_tolerance: 0.8 #A tolerance on the goal point for the planner.
  visualize_potential: false #Specifies whether or not to visualize the potential area
                           computed by navfn via a PointCloud2
```

The global planner `NavFn` implements only the Dijkstra's algorithm and, since it worked well, no additional time was spent to find other solutions that use A^* .

6.4.3 The Local Planner

The parametrization of the local planner [41] is far more important, complicate and delicate than the one for the global planner. Moreover, before finding the best set of parameters for the Ridgeback, also other planners provided by ROS were tested, as stated in Section 5.1.2. The parametrization of the local planner is reported in the `.yaml` snippet 6.3. There are to versions of this parameter set, to make the local planner behave differently in the mapping process with respect to the navigation (as detailed in Appendix F.1).

Listing 6.3: Local Planner parametrization (exploration case)

```
DWAPlannerROS:
#http://wiki.ros.org/dwa_local_planner

 5  acc_lim_x: 0.5
    acc_lim_y: 0.5
    acc_lim_th: 2.0

10  max_trans_vel: 0.55
    min_trans_vel: 0.1

    max_vel_x: 0.4
    min_vel_x: -0.1

15  max_vel_y: 0.1
    min_vel_y: -0.1
```

```

max_rot_vel: 1.0
min_rot_vel: 0.4

yaw_goal_tolerance: 0.157 #pi/20
xy_goal_tolerance: 0.08
latch_xy_goal_tolerance: false

vx_samples: 5
vy_samples: 5
vth_samples: 10

path_distance_bias: 32.0
goal_distance_bias: 24.0
occdist_scale: 0.4

publish_cost_grid_pc: true

```

As it can be seen, the velocities and accelerations of the robot are defined in the snippet of code 6.3, within line 4 and 19 (and they are used in the Dynamic Window Approach, Figure 5.7). The values for `min/max_trans_vel` and `min/max_rot_vel` define the limits for the linear and rotational velocities, as explained in Section 5.1.2 and reported in Figure 5.7. The negative velocity in x is lower, in module, than the positive one, because a backward trajectory is preferred to be slower, for safety reasons. In the case of the velocity in y , it does not make sense to consider two different values for the *max* and the *min*, because, in reality, one does not favor a movement to the right with respect to the one to the left. The key of the flexibility of this planner is in the definition of the velocities: it can be adapted to both holonomic and non-holonomic robots by imposing y -velocities, which will be null in the second case.

The `yaw/xy_goal_tolerance` defines the tolerance of the goal point for both position and orientation, while `latch_xy_goal_tolerance` defines the behavior of the robot once the goal is reached: if `false`, once the robot achieves the goal, it will try to execute small rotations to respect the tolerances, while, if `true`, once the robot reaches the goal it will only rotate in place, even if it ends up outside the goal tolerance while it is doing so.

The parameters `vx_samples`, `vy_samples` and `vth_samples` determine how many velocity samples the planner will generate in the forward simulation.

Then, there are the parameters intended to weight the planner behaviors, as discussed in the Section 5.1.2, dedicated to the local planner.

To conclude, `publish_cost_grid_pc` allows to display the cost function in RViz, as shown in Figure 6.5. The complete list of parameters is reported in Appendix F.1.

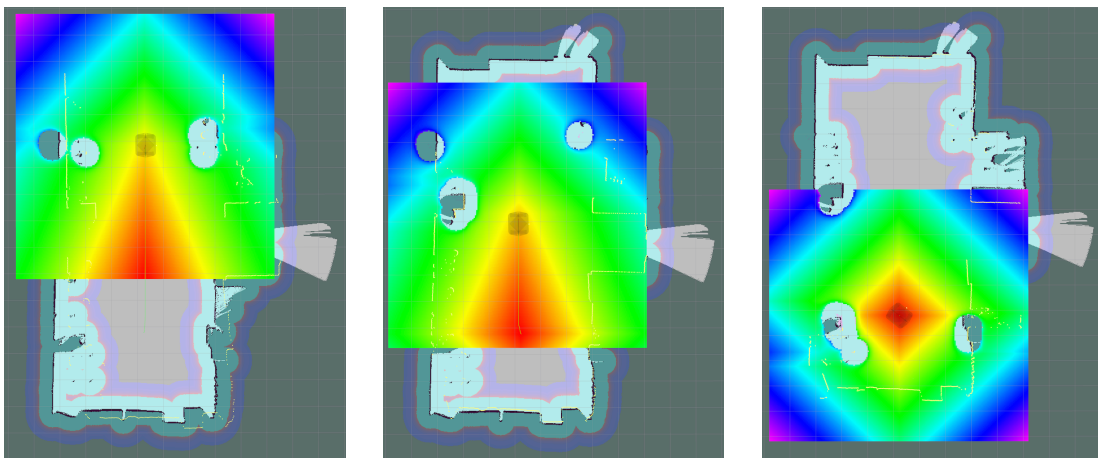


Figure 6.5: Evolution of the cost function of the DWA local planner.

6.4.4 Cost maps

The cost maps parametrization [39] is very important, because cost maps transmit to the planner useful data for obstacle avoidance. The common parameters for both global and local cost maps are included in the same `.yaml` file called `costmap_common` (snippet 6.4), while dedicated ones are stored in other files, `costmap_global` (snippet 6.5) and `costmap_local` (snippet 6.6), respectively. The complete file is reported in Appendix F.2.

Listing 6.4: Common Cost Map parametrization

```

map_type: costmap

footprint: [[0.48, -0.40], [0.48, 0.40], [-0.48, 0.40], [-0.48, -0.40]]
footprint_padding: 0.1
5
global_frame: map
robot_base_frame: base_link
update_frequency: 2.0
publish_frequency: 2.0
10
inflation:
  inflation_radius: 1.0 #bigger than circumscribed_radius defined in costmap_local

obstacle:
15  observation_sources: "scan scan_r"

```

The `footprint` indicates the robot shape as a vector of points, while `footprint_padding` is its uncertainty; they are both expressed in meters.

Two important parameters are the `update_frequency` and the `publish_frequency`. They specify the frequency with which the map is updated and published in the relative topic, and they have to be finely tuned in order not to miss update map loops. If this occurs, frontiers and obstacles are not updated, and the robot can stop or, even worse, keep proceeding, unaware of the presence of an obstacle.

Finally, the plugins allow to define other specific parameters. For example, with `inflation` one can choose the dimension of the area around the obstacles, which is taken into account for obstacle avoidance, as in Figure 5.6, discussed in Section 5.1.2. With `obstacle` one can consider also data coming from the rear sensor of the robot, by adding it to the `obstacle_sources`. However, this extensions works for the navigation but not for the exploration.

Listing 6.5: Global Cost Map parametrization

```

global_costmap:
  width: 50.0
  height: 50.0
  origin_x: -25.0
  origin_y: -25.0
  static_map: true
  rolling_window: false
  resolution: 0.02
5
plugins:
  - {name: static, type: "costmap_2d::StaticLayer"}
  - {name: obstacle, type: "costmap_2d::ObstacleLayer"}
  - {name: inflation, type: "costmap_2d::InflationLayer"}
10

```

Listing 6.6: Local Cost Map parametrization

```

local_costmap:
  width: 10.0
  height: 10.0
  static_map: false
  rolling_window: true
  resolution: 0.05
5

```



```

10 plugins:
    - {name: obstacle, type: "costmap_2d::ObstacleLayer"}
    - {name: inflation, type: "costmap_2d::InflationLayer"}

```

The global cost map dimension is a rough estimate of the area the robot moves in, it does limit, though, the zone where one can impose a goal. In any case, the dimension of the global cost map does not limit the navigation, because the map is updated dynamically whenever the robot exceeds the limits. The dimension of the local cost map, instead, is more important, because all the computations of trajectories and cost functions are done within its limits (Figure 5.5). The global cost map is static, while the local one is not: it is centered on the robot (as the parameter `rolling_window` indicates). The `resolution` parameter can differ from one map to another, in particular, the one for the global cost map is equivalent to the one for the map acquired via grid mapping. In any case the resolution indicates the size of each cell in the map, in meters.

To conclude, the `plugins` allow to add layers to the cost maps and are crucial for the avoidance of both static and dynamic obstacles: the `plugins` must be kept in the order specified in the snippet 6.5, because they are treated as a list by the code (“*first in first out*”). The *static layer* retrieves data from the static map, the *obstacle layer* updates the map with new laser scans and the *inflation layer* generates the *inflation* area around the obstacles. If, for example, `inflation` is put before `obstacle`, only the obstacles already existing in the static map will be inflated, any dynamic obstacle will not, avoiding the global planner to compute another path for the robot.

6.4.5 Gmapping

The `gmapping` (grid mapping) file contains a list of parameters that allow to modify the behavior of the SLAM algorithm discussed in Chapter 4 [51]. Also in this case, the dimension of the map should be equal to the real environment. For `gmapping`, the size of the map does limit the selectable area for the exploration: if one wants to explore all the environment at once, the map size should be regulated accordingly.

Listing 6.7: Gmapping parametrization

```

map_update_interval: 5.0 #How long (in seconds) between updates to the map. Lowering
    this number updates the occupancy grid more often, at the expense of greater
    computational load.

maxUrange: 5.0 #The maximum usable range of the laser. A beam is cropped to this value.
maxRange: 10.0 #The maximum range of the sensor.

5 particles: 10 #Number of particles in the filter.

# Initial map size (in metres)
xmin: -25.0
10 ymin: -25.0
xmax: 25.0
ymax: 25.0

# Processing parameters (resolution of the map)
15 delta: 0.02 #Resolution of the map (in metres per occupancy grid block).

```

The parameter `map_update_interval` indicates the updating period of the occupancy grid. The `maxRange` is the real range of the sensor, as one can see in the datasheet in Appendix C. However, within some meters, due to the sensor resolution step, some discontinuities appear in the scanned area and they are responsible for creating a lot of frontier regions, with the consequence of increasing the computation time and reducing the efficiency. By setting `maxUrange`, one can prevent this behavior.

The parameter `particles` allows to determine the number of particles used to build the map. The choice is justified by what stated in Section 4.4.2, knowing that CobotLab has

a size of $14m \times 7m$ and that the map has a resolution `delta` of $2cm$. The parametrized map size (`xmin, ...`) has to be greater than the real one.

6.4.6 Frontier Exploration

As largely discussed in Section 5.1.3, in the parametrization file 6.8 there are some parameters that have been added (with respect to those declared in [52]) to test the efficiency of our modifications and to adapt them to our needs.

Listing 6.8: Frontier Exploration parametrization

```

frequency: 2.0 #Frequency with which to reprocess the costmap for next frontier goal. If
0.0, only ask for new frontier when last frontier was reached via move_base. Higher
frequencies submit move_base goals more often and create 'smoother' exploration.

explore_costmap:
  track_unknown_space: true

5
plugins:
- {name: external,          type: "costmap_2d::StaticLayer"}
- {name: explore_boundary,  type: "frontier_exploration::BoundedExploreLayer"}

10
explore_boundary:
  resize_to_boundary: false
  frontier_travel_point: middle #When outputting pose of next frontier via ~
  get_next_frontier, define the geometric property of frontier to output as pose.
  position. Available- closest point to robot, middle point of frontier, centroid (
  cartesian average) of all frontier points.

  explore_clear_space: false #set to false for gmapping, true if re-exploring a known
  area

15
  min_frontier_dist: 1.0 #minimal distance from the robot position for frontier points
  to be valid
  min_frontier_size: 7 #minimum number of frontier cells to be considered as frontier
  point
  min_frontier_clear_dist: 0.4 #minimum distance of a frontier point to the closest
  obstacle

```

In the first place, the ideal value for frontier updating (`frequency`) would be 0, i.e. it should be updated once it is reached by the robot, but there could be two problems:

1. some frontiers are not reachable, so they must be updated to avoid the planner failure;
2. some bugs when running the programs can happen, so it is a good choice to refresh the frontiers.

The parameter `track_unknown_space` allows to decide whether the cell should take 2 values (obstacle or free), when it is set to false, instead of 3 (obstacle, free or unknown), if true. As for the case of the cost maps, there is a specific plugin to modify the parameters of the exploration task. There, one can find the previously discussed `frontier_travel_point`, i.e. how frontier regions are grouped in frontiers.

Finally,

- `min_frontier_dist` allows to choose the radius of the circular area that prevents a frontier from being under the robot;
- `min_frontier_clear_dist` allows to choose the radius of the circular area around a candidate frontier cell, which has to be free from obstacles;
- `min_frontier_size` is the size of the frontier regions that will be grouped in a frontier.

The first two are the parameters we added to the modified version of the code.

6.5 Launching the Exploration

To launch the exploration, three *launch file* are needed: the first one specifies the ROS nodes involved in the grid-based SLAM, together with their configuration files, the second one specifies the Gazebo map to launch and the third allows to view the robot in RViz. The launch files used for the simulation are reported in Appendix F.6 and F.8 (the third one is not reported because it is a default launch file in the Ridgeback packages). Thus, on the one side Gazebo is launched and the simulated model of the map and the robot are generated, on the other side also RViz starts and it displays the cost maps, the robot model and the sensor scans, as depicted in Figure 6.6.

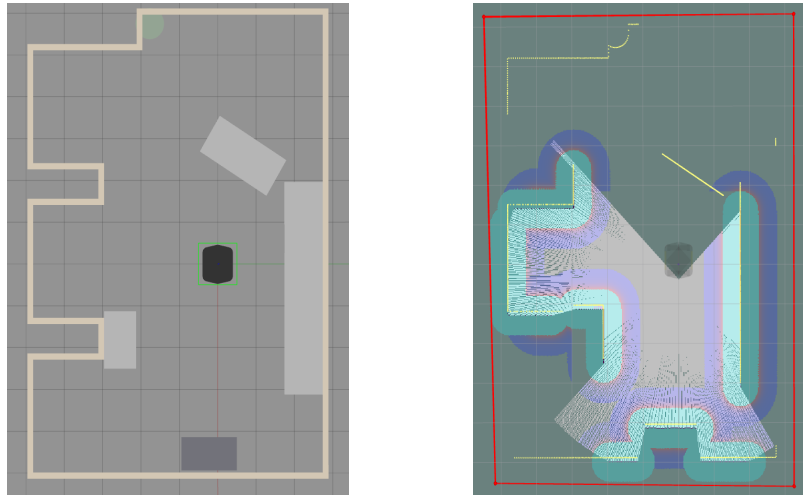


Figure 6.6: Initialization of the simulation environment: Gazebo (left) and RViz (right).

As discussed above, in the image on the right, one can see the discontinuities in the scanned area, while the scan limits are represented by the dotted yellow line. To start the exploration, it is necessary to publish some points in RViz (with a specific tool), in order to draw a polygon, like the red one in Figure 6.6 on the right. This polygon delimits the area the robot will explore. With RViz, one can also display the three layered cost maps (Figure 6.7): static (where the cells are free, obstacles or unknown), global (with different costs for obstacles) and local (equal to the global ones, but centered in the robot).

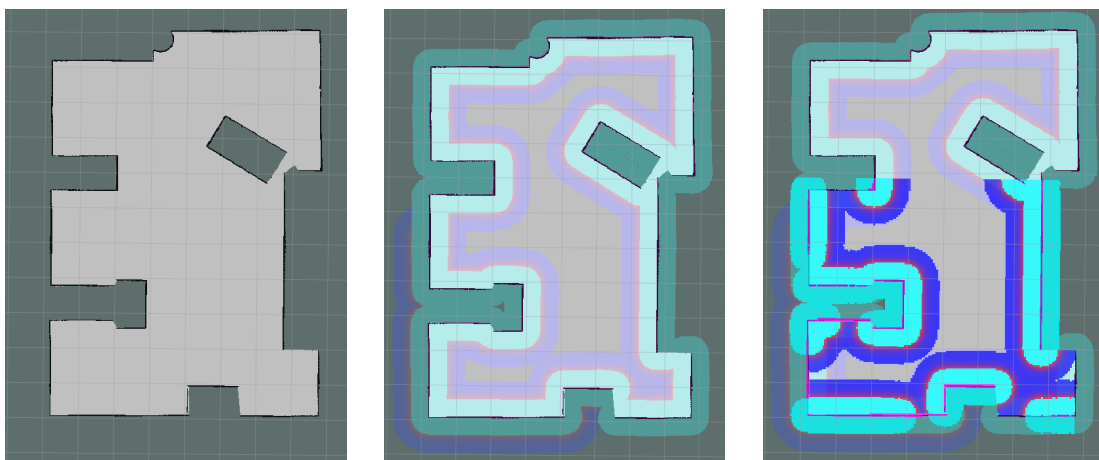


Figure 6.7: The three layered cost maps of the CobotLab.

In Figure 6.8, the global planner (green line) finds a path from the robot to the closest frontier (magenta square), among the other ones (red squares). The local planner (red line) creates local trajectories based on the velocities of the robot. At the beginning, multiple frontiers are found, while towards the end only a few remain. This behavior is due to the discontinuities of the sensor beam, which appear within some meters. They create small connected groups of frontier cells, later grouped in frontiers.

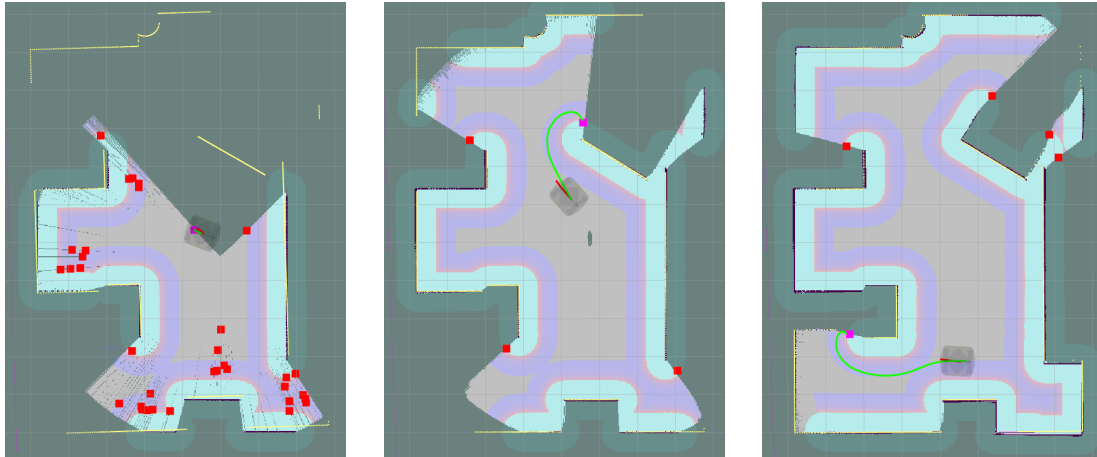


Figure 6.8: Three instants of the exploration task of the CobotLab in simulation.

To be sure that the exploration task ended successfully, the following message has to appear in the terminal:

```
[ WARN] [ ... ]: Finished exploring room
```

Finally, the map built in simulation looks like the one in Figure 6.9.

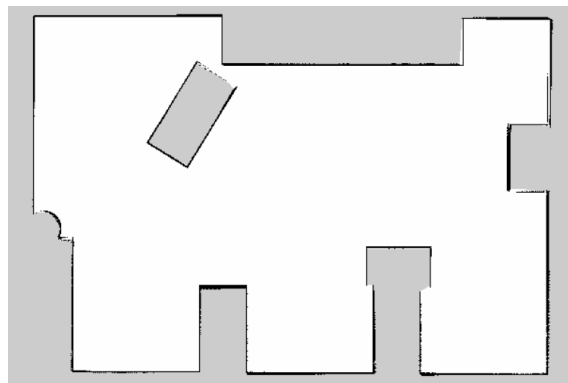


Figure 6.9: Resulting map after SLAM gmapping task.

The real CobotLab differs a little from the environment we built in Gazebo, and it is depicted in Figure 6.10, also with the robot (shown in Figure 6.11 as well) ready to explore it. The real robot is controlled through the Wi-Fi of the laboratory. In order to do that, the ROS Master has to be exported from the laptop to the computer of the robot. Moreover, now that there is the real robot, Gazebo is no longer necessary.

The results of the exploration and the built map are reported in Figure 6.12. During the mapping process, dynamical obstacles, like a person walking in the area scanned by the robot, are detected but not included in the map. This behavior is due to the SLAM

algorithm: the particles associated to dynamical obstacles do not survive subsequent re-sampling steps.



Figure 6.10: CobotLab in ABB Corporate Research Center.



Figure 6.11: Three views of the Ridgeback robot.

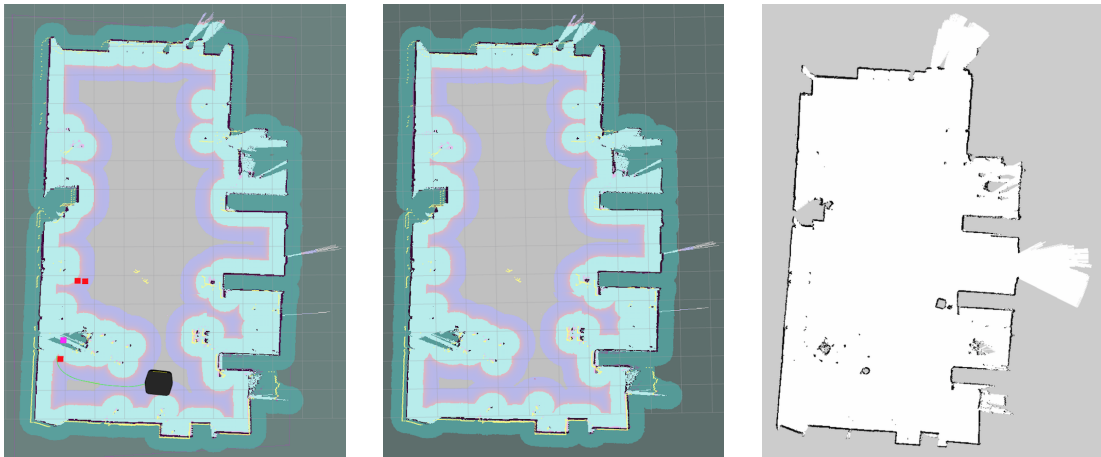


Figure 6.12: Exploration of the real CobotLab.

Finally, these algorithms were also tested in the more complex task of exploring two rooms connected by a corridor. However, so far, this has only been tried in simulation (Figure 6.13). To design a corridor can be quite tricky, because one has to include some discontinuities in the wall. Indeed, the robot has a tendency to lose itself in long, straight and flat environments; this is because it does not find anything useful to use as a reference. These results have been obtained with the same parameter set as for the CobotLab, except for the map size.

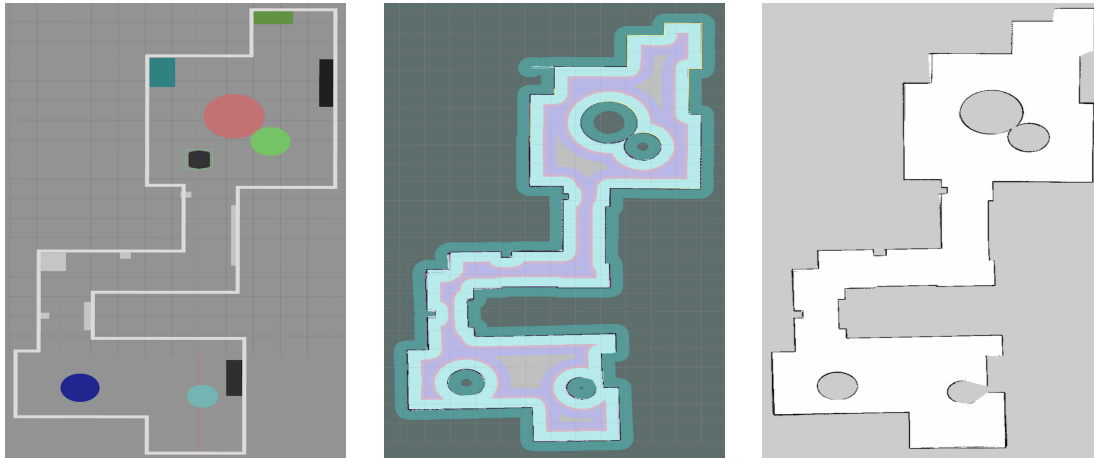


Figure 6.13: Simulated exploration task of a two rooms environment.

6.6 Navigation of the built map

Once the map is built, the navigation task can be executed. A dedicated launch file (in Appendix F.7) specifies the map to navigate and the node executing the AMCL, together with the nodes required by the Navigation Stack (global and local planners) and cost maps. Also the AMCL node comes in with a parametrization file [53], entirely reported in Appendix F.5. Snippet 6.9 shows only the most significant parameters that will be discussed.

Listing 6.9: Augmented MCL parametrization

```

min_particles: 100
max_particles: 500

# Odometry motion model
5 odom_model_type: omni-corrected
odom_alpha1: 0.010
odom_alpha2: 0.010
odom_alpha3: 0.020
odom_alpha4: 0.010
10 odom_alpha5: 0.006

# Exponential decay rate for the slow average weight filter, used in deciding when to
  recover by adding random poses. A good value might be 0.001.
recovery_alpha_slow: 0.001
# Exponential decay rate for the fast average weight filter, used in deciding when to
  recover by adding random poses. A good value might be 0.1.
15 recovery_alpha_fast: 0.1
# Initial pose mean
initial_pose_x: 0.0
initial_pose_y: 0.0
initial_pose_a: 0.0
20 initial_cov_xx: 25.0 #5m
initial_cov_yy: 25.0 #5m
initial_cov_aa: 40.0 #2pi rad

```

This parametrization set is very easy to understand, especially compared to the AMCL algorithm discussed in Sections 5.2.1 and 5.2.2. Indeed, `min/max_particles` allows to set limits on the number of elements in the particle set \mathcal{X}_t of the filter, while `recovery_alpha_slow/fast` are the values of the exponential decay rate that controls the addition of random particles, it holds: $0 \leq \alpha_{slow} \ll \alpha_{fast}$. Then `odom_model_type` is set to `omni-corrected` to adhere to the code 5.6¹. These parameters were left to their default values because the overall

¹Indeed, this parametrization refers to the algorithm in [34], reported in Section 5.2.

performance was really satisfactory and it would have been a waste of time to tune them differently. What was changed, though, was the initial covariance of the robot, because we wanted to make the algorithm converge, even in case of a completely unknown initial position.

Once the navigation starts, a number of particles, defined by the imposed limits and representing the estimated robot pose, are displayed according to the covariance values (the black arrows in Figures 6.14 and 6.15). Then one can choose between estimating the robot pose with the RViz tool `PoseEstimate`, or moving the robot manually. With the first option, the AMCL is forced to converge to the given pose; since this pose is likely to be close to the real one, the algorithm converges quickly. By moving the robot, the algorithm will converge autonomously after some steps, thanks to the measurement data it collects during the motion. However, in this case, the algorithm may fail, due to the fact that the map presents symmetries and there are locations with similar features, features that may lead to confusion.

The first behavior is depicted in Figure 6.14, while the second in Figure 6.15, where the algorithm fails in identifying the robot pose, as the scanned shape (in yellow) does not correspond to the map walls (in black).

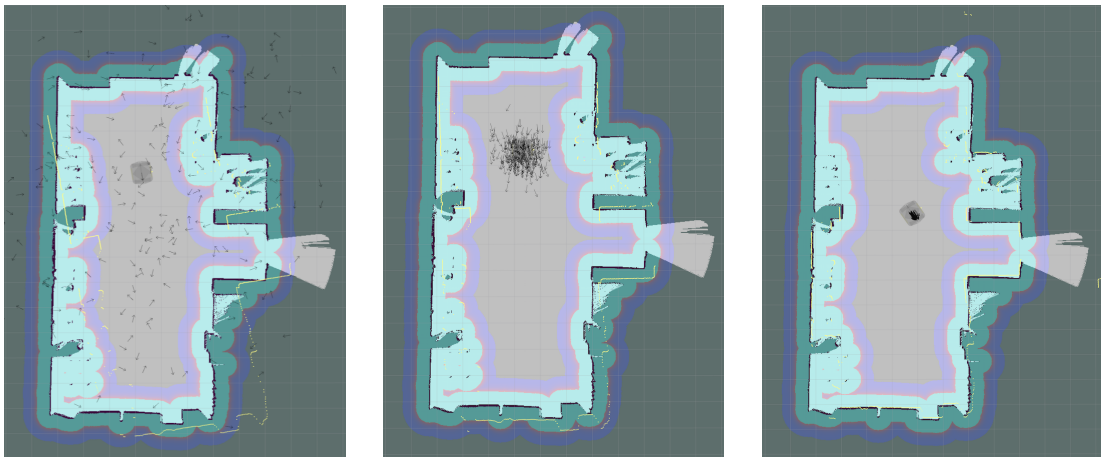


Figure 6.14: Convergence of the AMCL algorithm in estimating the robot pose.

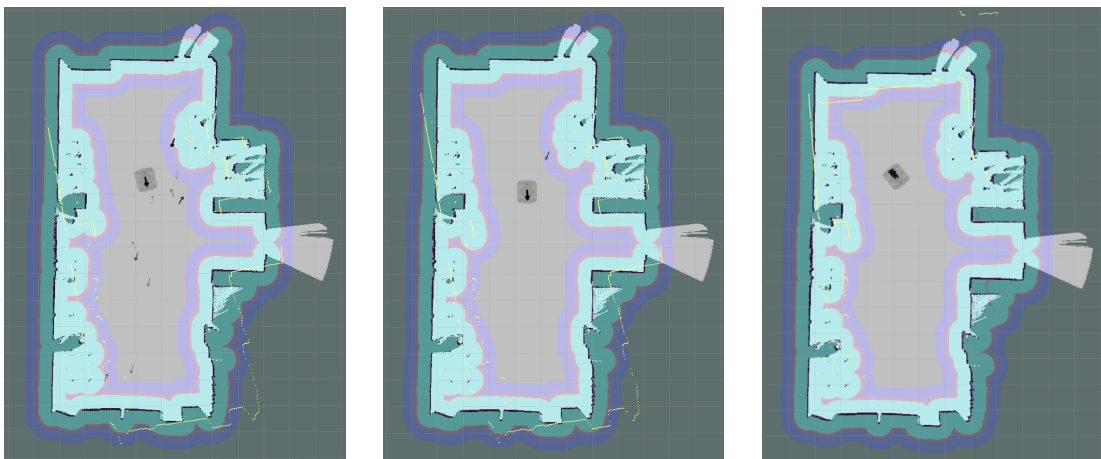


Figure 6.15: AMCL algorithm fails in estimating the robot pose.

During the navigation, also the cost maps are launched, so dynamical obstacles detected by the laser appear in the map and, thank to the *inflation* layer, the robot is capable of avoiding collisions and of replanning.

6.7 Accuracy and Robustness tests

The purpose of having a mobile platform is to autonomously move a manipulator from a place, i.e. a workstation, to another, to allow it to perform tasks in different locations. Therefore, it is important that the mobile platform reaches each workstation with a sufficient precision, for example a short distance from the imposed goal, and robustness, i.e. it reaches the goal with a high probability of success. To this goal, a test protocol to measure accuracy and robustness in different conditions was designed, as reported below.

Accuracy Tests

1) Map Accuracy:

- *measurement*: difference between the dimensions of the digital and real map;
- *test procedure*: measurement of the same characteristic features of the environment for each built map.

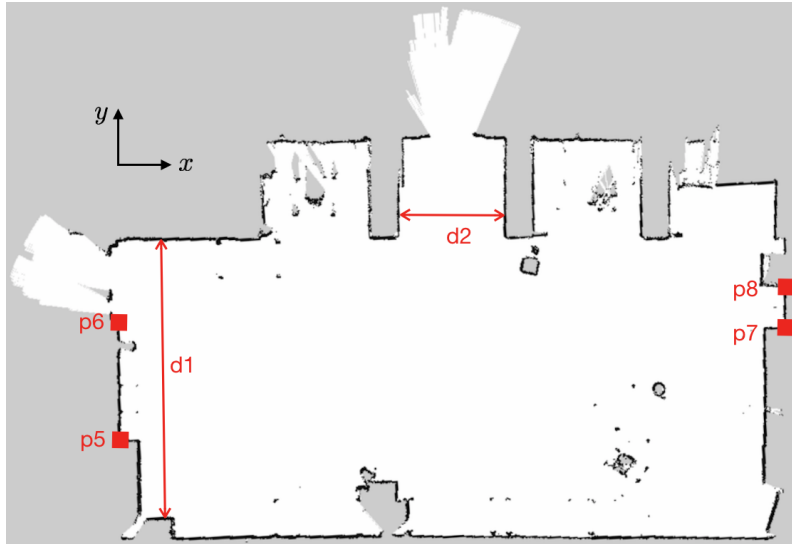


Figure 6.16: Chosen features for the map accuracy test.

To evaluate the map accuracy, two distances $d1$ and $d2$ were measured, as shown in Figure 6.16: the angle drawn by the two segments $\overline{p5p6}$ ($w1$) and $\overline{p7p8}$ ($w2$) was computed:

$$\alpha_w = \alpha_{w2} - \alpha_{w1}$$

$$\alpha_{wi} = \arctan 2 \left(\frac{\Delta x}{\Delta y} \right) = f(\Delta x, \Delta y) \quad (6.7.1)$$

The point coordinates are first measured in pixels, by opening the map in the program GIMP Image Editor, with an uncertainty of one pixel, then they are converted into map coordinates, through the following relation:

$$\begin{aligned} map_x &= pixel_x \times resolution + origin_x \\ map_y &= pixel_y \times resolution + origin_y \end{aligned} \quad (6.7.2)$$

where origin_x/y are the parameters of the global cost map, used here to make the origin of the map coincide with the spawning point of the robot. One can notice that with this procedure it is not possible to be more precise than the resolution of the map.

Then the uncertainties of each measurement were also derived:

$$\begin{aligned} u(\Delta x) &= u(\Delta y) = \sqrt{2} \cdot \text{resolution} \\ u(\alpha_w) &= \sqrt{u(\alpha_{w1})^2 + u(\alpha_{w2})^2} \\ u(\alpha_{wi}) &= \sqrt{\left[\left| \frac{\partial f}{\partial \Delta x} \right| u(\Delta x) \right]^2 + \left[\left| \frac{\partial f}{\partial \Delta y} \right| u(\Delta y) \right]^2} = \frac{u(\Delta x)}{\sqrt{(\Delta x)^2 + (\Delta y)^2}} \end{aligned} \quad (6.7.3)$$

Each measurement was repeated five times with a map resolution of 2cm (hence for five different maps), twice with a resolution of 1cm and only once with a resolution of 5cm (expressed as “TAKE Nb.” in the abscissa of the graphs of Figure 6.17). The outcomes are plotted in the graphs of Figure 6.17, where the uncertainties increase when lowering the resolution, as expected.

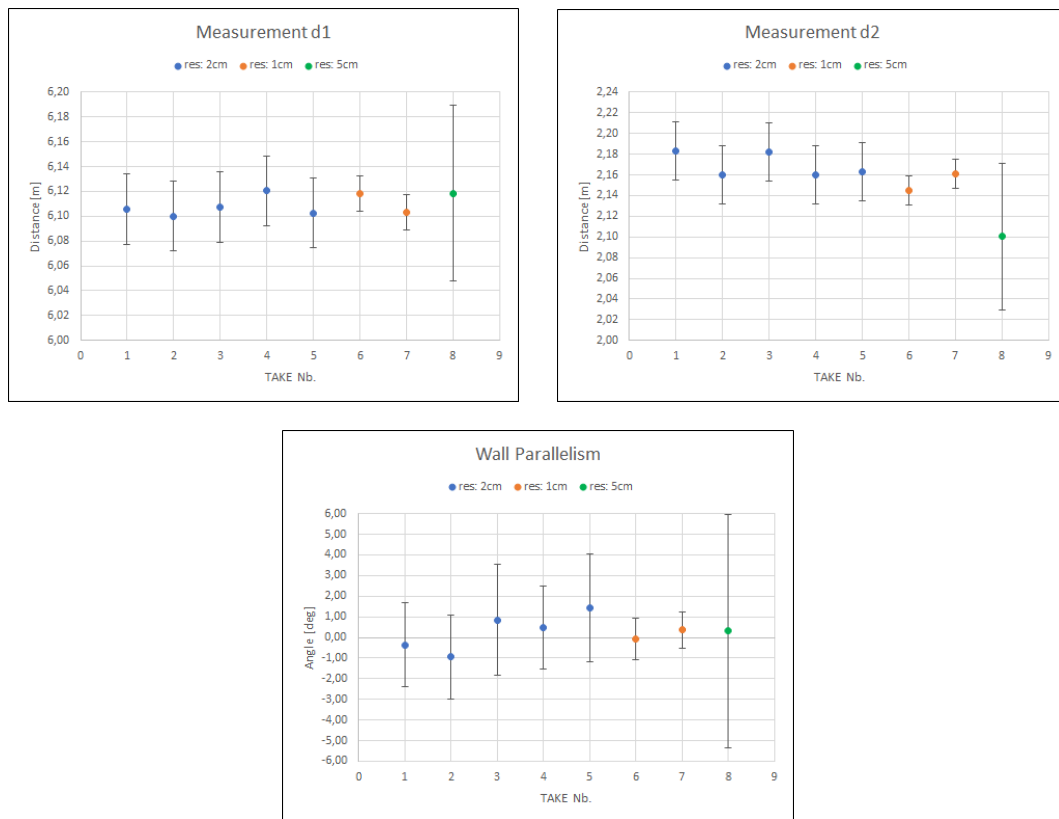


Figure 6.17: Graphs for the three measures to evaluate the map accuracy.

Measure	Mean	Std. Dev.	Real Value
$d1$	$6,102m$	$0,0202m$	$6,120m$
$d2$	$2,157m$	$0,0259m$	$2,185m$
α	$0,251^\circ$	$0,733^\circ$	$\simeq 0^\circ$

Figure 6.18: Resume of the measurements for the map accuracy.

2) Movement Accuracy:

- *measurement*: difference between planned and executed trajectory;
- *test procedure*: to publish a constant velocity command to obtain a circular trajectory and to measure the error between the end pose and the start one after a fixed number of loops.

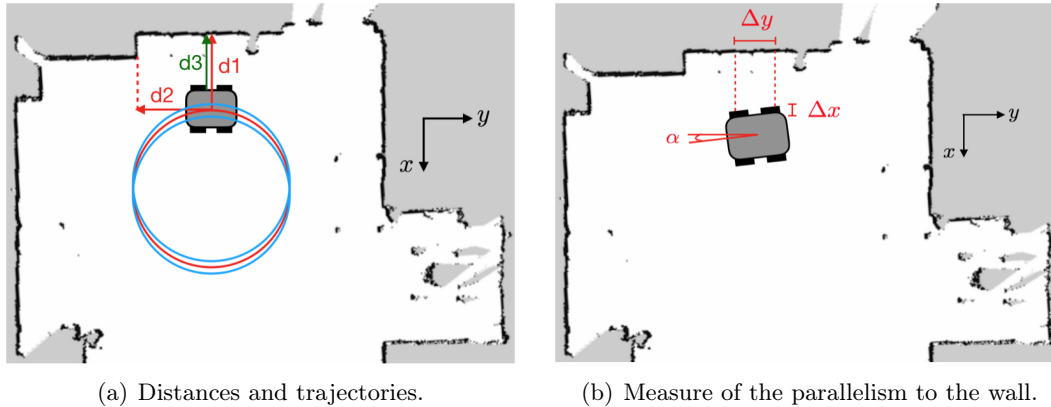


Figure 6.19: Scheme of the measures for the Movement Accuracy Test.

The purpose of this test is to measure the precision of the odometry, as it is done in [54]. Odometry generates two types of error, systematic and nonsystematic. The errors that are vehicle specific belong to the first category, such as unequal wheel diameters, degradation of the tires materials (rubber or urethane), differences between the wheel design and the actual manufacture, tires contact with the floor which is an area and not a point. The errors caused by the environment the robot navigates, such as wheel slippage, belong to the second category.

Only systematic errors can be corrected to improve odometry accuracy. The calibration of these errors is done by the authors in [54] with a square path track. We preferred a circular path because it is easier to implement.

To realize this experiment, the robot was placed as parallel as possible to a wall, as shown in Figure 6.19, and this position was marked as the reference starting point: $d1 = 1.30m$, $d2 = 1.30m$, $\alpha = 0.51^\circ$. Then, a velocity command message `geometry_msgs/Twist` was published on the topic `/cmd_vel` with a frequency of $10Hz$, and it was checked whether the produced trajectory was consistent with the input. We define it as an *open loop* test, because the robot relies only on the odometry to follow the velocity command. It differs from a *closed loop* approach, where, instead, the input is a trajectory imposed by a planner and the robot uses both odometry and localization to follow it.

The test in *closed loop* was not realized, because it would have required to design a specific planner for it, a planner that, instead of finding a path from a starting position to a goal, generates fixed trajectories, such as circles, straight lines, and so on.

A circular trajectory of radius $R = 1m$ (the red circle in Figure 6.19a) was imposed and, since $v_x = \omega R$, the command $v_x = 0, 1m/s$, $\omega = 0, 1rad/s$ was published. For a rigorous experiment, one could publish the command, then stop the robot after one loop, which means after $t = 2\pi/\omega = 62, 8s$, and finally measure the reached position and compare it to the starting one.

However, the test was highly influenced by the Wi-Fi communication between the robot and the computer, which produced a not negligible delay, thus falsifying the measurement. Thus, the displacement along $d3$ in Figure 6.19a of the robot side, at the passage of the robot on the starting position, was instead measured. To better understand this problem: after one loop the reached position was $d1 = 1.305m$, $d2 = 1.085m$ with an orientation of

$\alpha = 0.51^\circ$. To obtain more observations, the number of loops was augmented to five and the velocity command was increased to $v_x = 0,4m/s$, $\omega = 0,4rad/s$:

```
$ rostopic pub -r 10 /cmd_vel geometry_msgs/Twist
"linear:
  x: 0.4
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.4"
```

To take the measurements, two tapes have been placed along the circumference of the circle, corresponding to an angle of $\simeq 0$ radians and one of $\simeq \pi$ radians. The measured offset along $d3$ oscillated from $1cm$ to $3cm$ (schematically represented by the blue circles in Figure 6.19a). Since it was not possible to stop the robot in a precise way, and the measurements have been taken during the motion, these values are reliable only in terms of order of magnitude.

3) Goal Accuracy:

- *measurement*: difference between given goal and reached goal;
- *test procedure*: to publish a goal point and to measure accuracy in goal reaching using environment features as reference for the measurement (which are, hence, biased by map accuracy).

4) Localization Accuracy:

- *measurement*: difference between estimated pose and real pose;
- *test procedure*: same procedure as goal accuracy but estimated and real pose are compared.

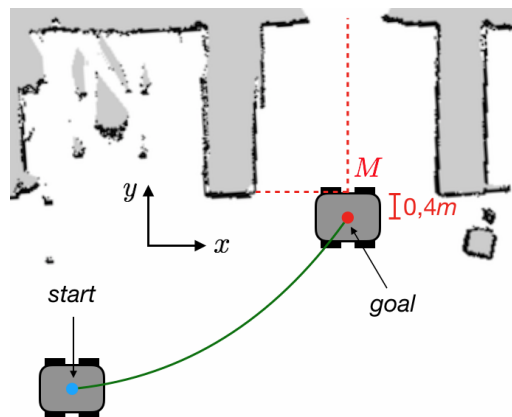


Figure 6.20: Scheme of the measures for the Goal Accuracy Test.

Regarding this experiment, the authors in [55] have equipped a $10m \times 10m$ room with two goniometric sensors for perceiving the robot and two beacons to be detected by the robot. The localization algorithm estimates ten poses of the robot along a circular trajectory and then they are compared with those of the room sensors. Since this kind of equipment was not available, another test was designed.

The goal point for the robot, chosen to measure the accuracy, was approximatively the middle point at the end of the entrance hall of the CobotLab, the M point shown in Figure 6.20, computed in the built map. Measuring the goal sent to the robot is tricky,

because it is a point reached by the center of the robot, so it is actually *under* the robot. What it was done instead, was to translate the goal along y by half the width of the robot, namely $0.4m$, so that the measured position M could be reached by the *side* of the robot. To this extent, the goal tolerance and the number of samples were set for the local planner, as it is reported in the snippet 6.10.

Listing 6.10: Local Planner parametrization for Goal Accuracy Test

```
DWAPlannerROS:
  yaw_goal_tolerance: 0.03142
  xy_goal_tolerance: 0.03
  latch_xy_goal_tolerance: true
  vx_samples: 6
  vy_samples: 7
  vth_samples: 20
```

The first step of the test was to localize the robot in the map, then it was moved from a starting position, the origin of the map, to the goal point described above.

The goal is a `geometry_msgs/PoseStamped` message, published in the topic `/move_base_simple/goal`. The message specifies the position in Cartesian coordinates and the orientation in quaternions.

The message regarding the starting position is the following:

```
$ rostopic pub /move_base_simple/goal geometry_msgs/PoseStamped
"header:
  seq: 6
  stamp:
    secs: 1529585344
    nsecs: 286991328
  frame_id: "map"
pose:
  position:
    x: 0.0
    y: 0.0
    z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 0.0
    w: 0.1"
```

The message for the measurement position, instead, is written as it follows:

```
$ rostopic pub /move_base_simple/goal geometry_msgs/PoseStamped
"header:
  seq: 6
  stamp:
    secs: 1529585344
    nsecs: 286991328
  frame_id: "map"
pose:
  position:
    x: 3.75
    y: 2.72
    z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 0.0124997
    w: 0.9999219"
```

This path start-goal was repeated six times and the result is reported in the graph of Figure 6.21.

Together with measuring the reached position, also the estimated position was checked,

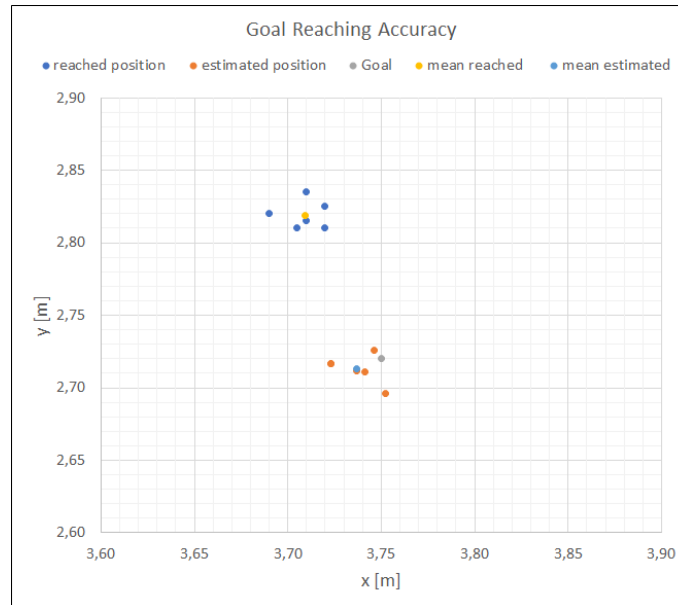


Figure 6.21: Results of the Goal Accuracy test.

to evaluate the localization accuracy. What it can be seen in Figure 6.21 is that the estimated position is very close to the goal sent to the robot. This is because the robot uses the AMCL to navigate the map. So it can be concluded that the robot achieves a good precision in reaching the goal repeatedly.

Moreover, it is clear that the best way to obtain a good goal accuracy is to set the goal from the estimated pose of the robot, once the desired pose is reached, rather than measuring it from environment features. In such a way, indeed, the measure is no more affected by the map precision.

To support such conclusions, in Figure 6.22, two mean values, together with the standard deviation, are reported. The first value specifies how far, in average, the reached positions are both from their mean and to the goal. The second, instead, specifies how far, in average, the estimated positions are both from their mean and the goal.

Measure	<i>reached position</i>		<i>estimated position</i>	
	Mean	Std. Dev.	Mean	Std. Dev.
Dist. to Mean	0,0126m	0,0135m	0,0122m	0,0141m
Dist. to Goal	0,108m	0,108m	0,0189m	0,020m

Figure 6.22: Resume of the measurements for the map accuracy.

Repeatability and Robustness Tests

1) Map Building:

- *measurement*: percentage of success in exploration and duration of the task;
- *test procedure*: to run a fixed number of exploration tasks.

This test was realized by repeating five times the exploration task, with a map resolution of 2cm. The task always succeeded, with an average time of 65,4s and a standard deviation of 9,81s.

At this point, in light of the experiments done so far, we can conclude that building a map with a resolution of 2cm is the best compromise between the computational time (not measured, though, but substantial in practice) and the desired precision.

2) **Localization:**

- *measurement*: percentage of success in localization and duration of the task, with unknown starting pose;
- *test procedure*: to run a fixed number of localization tasks with high uncertainty on the starting pose (the robot is moved manually).

3) **Localization:**

- *measurement*: percentage of success in localization and duration of the task, with starting pose known approximately;
- *test procedure*: to run a fixed number of localization tasks with low uncertainty on the starting pose (the robot is moved manually).

The unknown starting pose refers to the *global localization problem* defined in Section 4.2. In this case, when the AMCL is launched, the robot is unaware of its location, which can be anywhere in the map. The robot performs the localization either by reaching goals sent with the RViz tool `2DNavGoal`, or by a manual control. It differs from an approximately known starting pose (*local localization problem*), because in such a case, the pose of the robot is estimated to be close to the real one, via the RViz tool `PoseEstimate`.

The experiment was run five times in each of the following conditions:

- no `PoseEstimate`, robot moved via controller;
- no `PoseEstimate`, robot moved via `2DNavGoal`;
- `PoseEstimate` first, then robot moved via `2DNavGoal`.

Two different probabilities of success/failure were computed, with the discriminant of the use of a `PoseEstimate`. The first measure is the success in localizing the robot, when the AMCL algorithm converges the first time. The second one quantifies the algorithm capability of correctly localizing the robot, even after the first convergence. The result is reported in Figure 6.23.

Measure	PoseEstimate	
	No	Yes
first convergence	20%	100%
final localization	60%	100%

Figure 6.23: Percentage of success in the localization task.

We believe that the cases where the final localization failed were caused by the symmetries of the CobotLab: in fact, most of the time the robot pose converged to an acceptable Cartesian position, but with an orientation that differs by π radians from the real one, so with the map upside down.

We also wondered whether the algorithm was able to solve the “kidnapped robot problem” introduced in Section 5.2.2. The answer is difficult to find because with a little robot it is possible to move it from a place to another, simulating a kidnapping. With a $140kg$ robot it is more difficult to reproduce this experiment. What could be done is to give the robot a false `PoseEstimate` and check if the algorithm succeeds in re-localizing the robot.

4) **Navigation:**

- *measurement*: ability to navigate in dynamic environment;
- *test procedure*: to add new obstacles and to move them with respect to the built map, check obstacle avoidance and path replanning.

5) **Navigation:**

- *measurement*: ability to continue the task after being blocked by obstacles;
- *test procedure*: to block the robot to avoid its motion and free it after a certain time and check path replanning.

Thanks to the layer dynamic explained in Section 6.4.4, the robot is always capable of avoiding obstacles and of replanning. There is nothing more to say about this issue.

To conclude this section, the performed measurements are nothing but a raw indication of the precision of the robot and the algorithms involved in navigation. This is due to the objective difficulty in finding reference points in the real environment and in transferring them on the digital map. All the measurements, indeed, are biased by the precision of the map, by the precision with which the points in this map are computed from the pixels and by the measurements taken in the real environment. Even though the uncertainties have been estimated, one does not have to take the outcomes of the experiments as final. Moreover, the estimation of the uncertainties relies on the hypothesis that the error of the measurement of a pixel coordinate is of one pixel, which is reasonable but also questionable. This is also why these tests have been repeated a limited number of times.

Since this test protocol was designed to try to tentatively estimate the accuracy and the robustness, the outcomes are quite satisfactory for this purpose. If more precise measurements are required, a new and more rigorous test protocol needs to be designed.

The robustness tests, though, are more reliable because they are easy to reproduce and they do not require precise measurements.

Finally, one might notice that the measurements have been taken with the Ridgeback only, i.e. without any charge mounted on it. It is possible that, when the YuMi robot will be attached, the resultant robot system will lose some precision due to the changed inertia.

6.8 Conclusions and encountered problems

To conclude this first part of the thesis, the principal subjects are briefly surveyed.

The *Add-In*

Concerning the RobotStudio *Add-In*, it was done with the intention of using RobotStudio as a simulation environment for both the robots. After a second thought, however, we realized that RobotStudio was very restrictive because, up to now, it is not meant for navigation. Even if the movement can be computed by an external computer, RobotStudio should provide all the laser data, which are essential for mapping, localization and collision avoidance.

This approach, however, highlighted more problems than solutions, which is why it was abandoned and the navigation was performed in ROS only. If ABB will decide to design its own mobile robot, then it would need to design dedicated tools in RobotStudio and this work would be of some use.

The Mobile Robot

The mobile robot arrived in Sweden in poor packaging condition, and the yellow panel on the right hand side was deformed due to a hit. This caused the laser to detect an obstacle, which was therefore internal to the robot. To solve this problem this panel had to be removed.

Another problem related to the Ridgeback concerned the *IMU* (Inertial Measurement Unit) sensor mounted on it. The *IMU* combines accelerometers, gyroscopes and magnetometers and it is used to compute the accelerations along x and y and the angular velocity around z . These values are then used to correct the odometry. The problem was that, when the robot was not moving, the *IMU* returned non null values, e.g. $1m/s^2$ for the x acceleration, thus compromising the localization. This problem was solved by by-passing the *IMU*, modifying the file `control.yaml` stored in `ridgeback_control/config` by imposing \dot{z} , $\dot{\varphi}$, $\dot{\theta}$ to false (meaning that the corresponding measure is not used) in the matrix:

$$imu0_config = [x \ y \ z \ \varphi \ \theta \ \psi \ \dot{x} \ \dot{y} \ \dot{z} \ \dot{\varphi} \ \dot{\theta} \ \dot{\psi} \ \ddot{x} \ \ddot{y} \ \ddot{z}] \quad (6.8.1)$$

where φ = roll, θ = pitch and ψ = yaw. The other values are not used by the IMU. With this new configuration only the measures coming from the odometry are used, which, by the way, are sufficiently precise.

Concerning the sensors of the mobile robot, we would also like to share the consideration made in [43] about robot sensors.

The most used sensors are sonars and lasers. The formers use a sound pulse that hits the surrounding surfaces and bounces back to the sensor so that the data can be acquired. This sonar allows, hence, to scan a 3D environment. The main problem arises when a flat surface is hit, in which case the pulse does not bounce back toward the sensor: either no data is acquired or the sensed object appears farther than it really is (as reflections in mirrors). This problem was solved by the author in [43] by combining both sonars and lasers.

On the other hand, lasers operate on a 2D plane, which is very restrictive, as discussed below. An alternative would be to use 3D laser range-finder, but they are too large, too expensive (around 4 thousand dollars) and too power-hungry to be available for mobile robotics applications.

Ridgeback mounts two HOKUYO UST-10LX laser sensors (see datasheet in Appendix C), one on the front and one on the back of the platform at approximatively 29cm from the

ground. Even though these lasers are very precise, they scan a 2D plane, which means that every obstacle smaller than $29cm$ (for example, the wheeled basis for chairs and the wheeled support of some white boards that were present in the CobotLab) is not scanned and hence it is a potential source of collision (if these supports are wider than the inscribed radius of the detected part).

Another problem is that lasers pass through glass walls, i.e. glass obstacles are ignored. This was a major problem in CRC because this kind of wall is massively used as separator from the corridor and the workstations. Thus, opaque tapes will have to be put on the glasses, at the height of the sensor, to make them recognizable, in order to let the robot move without damaging the environment.

Moreover, Ridgeback has been bought to put YuMi on it: with such lasers, there might be objects in the environment that are not scanned but that could be a potential obstacle for YuMi.

To conclude, at the state of the art, the environment must be designed for the Ridgeback by removing any potential and not sensed danger.

As stated, Ridgeback mounts two lasers and we have not been able to use both by merging their data, so far. Indeed, having a 270° scanning area on the front and on the back, means to obtain a sensed zone that surrounds the platform. The algorithm used for the exploration (grid mapping) does not consider measurements coming from two sensors. A future work could be to modify the algorithm to accept also data from the rear sensor. In such a case, for the overlapping areas, the worst and the most conservative outcome is taken, i.e. the one with the highest probability.

Communication and Computation Power

The results obtained with the parametrization presented above, have been quite satisfactory, even if there have been two bottle necks that affected the performance of the robot:

- the Wi-Fi network of the CobotLab that, not being dedicated to the robot, caused some delays in the communication;
- the limited hardware components mounted on the laptop used for the simulation, i.e. the processor, the ram and the graphic card.

As a result, lots of computation loops were missed both by the controller and the map update, which caused some jerks in the robot motion.

Since communicating via Wi-Fi allows to have only one central process unit controlling all the robots (imagine to have a fleet of mobile robots or mobile manipulators), the first problem will be solved only when mobile robots will be used in ABB industrial applications, and then there will probably be a dedicated network. For the sake of the thesis work, the problem was partially solved by using a Ethernet connection instead, with the consequent annoying problem of being obliged to place the computer on the robot itself. Just to give an idea of the problem extent, by *pinging* the IP of the robot from the computer, the connection velocity oscillated from $200ms$ to $500ms$ using the Wi-Fi and around $0,1ms$ using Ethernet.

The computer problem was solved by using a gaming PC, whose characteristics are compared with the normal laptop in Figure 6.24.

The effects were a substantial reduction of missed loops (for the same parameter configuration) and hence a smoother motion. A quantitative indicator was given by the frame rate of RViz, that moved from an oscillating range $[3; 11]fps$ to a stable value of $31fps$.

PC	MSI GT73VR	HP ZBook
Memory	32 GB	16 GB
Processor	2,70 GHz, IntelCore i7	2,70 GHz, IntelCore i7
Graphics	GeForce GTX	NVE6

Figure 6.24: Characteristics of the two laptops used for the simulation.

To conclude, most of the modifications to the codes have been made only once they had been tested on the real robot. Indeed, most of the times, it is difficult in simulated environments to create *ad hoc* cases to make the simulation fail. Hence, during the simulation, the choice of the packages to use and the major modifications are made, while in the real experiment the algorithms are refined, together with the parametrization.

Part II

Mobile Manipulation

Introduction to Mobile Manipulation

The aim of this chapter is to introduce and define the mobile manipulator. The industrial context and the needs that have pushed the robotic research to investigate this particular kind of robot are presented in [56]. Moreover, an extended set of hardware requirements, to handle most of the industrial tasks, is proposed. The setup is detailed in [57] and in [58].

Robots are widely used in today's industry, to perform dangerous, repetitive and burdensome tasks. Moreover, robot-based industry increases the quality of the product and the speed of the manufacturing, it improves work conditions and allows an efficient use of resources. However, the needs of the industry have evolved simultaneously with the progress in technology and the demands of the market. In the last ten years, due to the globalization of markets, the trade instability and the shift from mass production to customized production, the industrial robots had to evolve consequently. In the early years of the 21st century, the industrial robots were rather inflexible, because they were only dedicated to one task and without the possibility of moving from their workstation. Thus, the production systems were either fully automated, and hence efficient in high production volumes but not flexible, or strictly manual, and hence very flexible but less cost-efficient. In recent years, the necessity to have both flexibility and efficiency in terms of production and cost has raised. A solution to this necessity is the autonomous industrial mobile manipulator, i.e. a mobile manufacturing assistant in industrial environment, with application in logistics, assistive and service areas. This technology combines the motion capabilities of the mobile robots with the manipulation abilities, extending the potentials of industrial robots applications [56].

A conventional mobile manipulator consists of a manipulator robot mounted on a mobile platform and it is normally provided with vision and tooling systems (Figure 7.1). This combined robot is designed to be capable of working in close contact and interactively with a human worker, in various applications.

In the industrial case, an operator has to deal with many objects, geometrically different and normally heavy to carry. In an ideal case, a mobile manipulator should be endowed with a flexible gripping technology, a payload of $20kg$, a workspace of $1,8m$ and a 24-hour energy supply. Moreover, the manipulator has to be lightweight and little enough to be mounted on the mobile platform and not to damage it. However, the greatest part of the manipulators, that meet the requirements in terms of weight and dimension, are limited in payload and workspace [57].

Concerning the hardware, the mobile platform must have a sufficiently large surface to allow the manipulator to be built on it. Also the payload is important, because the mobile robot has to carry both the manipulator and its charge. The manipulator itself, as already stated, should have a suitable payload and workspace for the tasks it is meant to accomplish. So does the tool that is attached to the end-effector of the manipulator.

Since the work environment is intrinsically unpredictable, the robot has to be endowed with different sensors. The objective is to perceive, with a sufficient precision, both the environment surrounding the robot and the manipulation surface on the robot itself. Regarding the first objective, the potentials and limitations of the 2D laser range finders have already been discussed, so they should be combined with cameras, or other sensors, to

extend the perception field of the mobile platform to the third dimension.

Additional sensors could be placed on the end-effector of the arm, for manipulation tasks. The hand camera of the manipulator allows to gain precision in manipulating objects, since, by analyzing the images, it is possible to correct the trajectory of the end-effector. Depending on the manipulator used, it can be also possible to mount another camera on the support of the manipulator, for example on the torso in the case of YuMi.

Authors in [58] suggest also to put the cameras in a pole, mounted on the platform, but independent from the manipulator (as in Figure 7.1c). This camera could be used to perform different tasks, such as modeling the environment, recognizing an object and referencing the mobile platform to the workstation.

The IT hardware has to provide enough network bandwidth and computational power to process the sensors data and run the algorithms. Moreover, if the wireless network is not dedicated, all the computations have to be done on CPU embedded in the mobile manipulator.

Finally, the mobile manipulator must have a human robot interface. Indeed, in the first place, it allows to easily train the robot to do a specified task, similarly to what happens with human workers. The second, but not less important reason, is that human workers need to see and understand what the robot is currently doing in order to predict its behavior and act consequently.

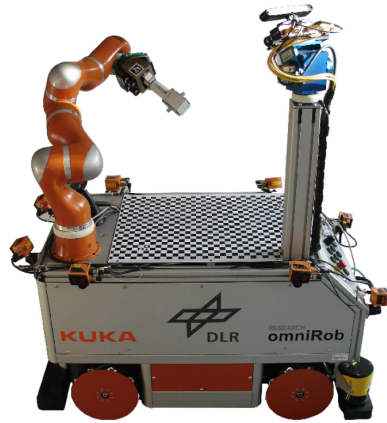
To conclude, the robot system must be programmable also by human workers with no knowledge of robotic engineering, or a knowledge limited to the task they have to program the robot for. So, the interface has to be intuitive and simple to use.



(a) UMan, *University of Massachusetts*, 2008



(b) MM-500 UR, *Neobotix*, 2011



(c) OmniRob II, *KUKA*, 2011



(d) RB-Kairos, *Robotnik*, 2017

Figure 7.1: Example of Collaborative Mobile Manipulators.

Manipulators: theoretical background and kinematic modeling

This chapter provides some preliminary information about manipulator robots. Some general definitions will be given, to conclude with the derivation of the kinematic model for the ABB IRB-14000 (YuMi) robot.

This chapter mostly relies on [32], but also on [59].

8.1 Homogeneous Transformation

In this section the *homogeneous transformation matrix* is derived. It allows to describe the relation between two different frames in which a point in the space is represented.

However, it is necessary, in the first place, to define a frame and to understand its importance in the robotics case.

Definition 8.1.1 (Coordinate frame). *A coordinate frame i , expressed as \mathcal{F}_i , is defined by a point, i.e. the origin O_i , and three mutual orthonormal basis vectors, the triad $(\hat{x}_i, \hat{y}_i, \hat{z}_i)$, that are all fixed within a particular body [60].*

To simplify the notation, the frame \mathcal{F}_i will be identified by the notation $O_i-x_iy_iz_i$, where the triad of vectors is mutually orthogonal.

The concept of frame is important because it allows to describe the pose and the displacement of a body. Since it is always possible to assign a frame to a body, the frames allow to express the pose and the displacement of a body relative to another body. In robotics this allows to define a task for the robot, in terms of the position to reach, but also how to reach it. For example, if the task is the grasping of an object, it is possible to describe the object to grasp with respect to a world fixed frame. Then, if also the frames attached to the base of the robot, and to all the robot joints, are known, it is possible to assign to each joint a position to reach, so that the task can be accomplished. This can be done thanks to the transformation between frames, i.e. the homogeneous transformations, which will be detailed hereafter.

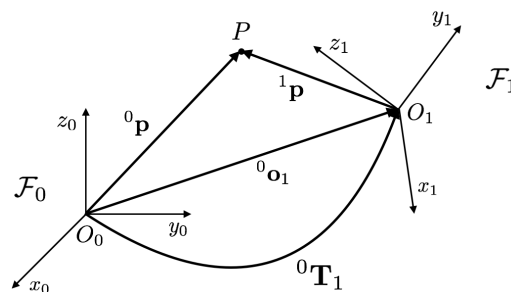


Figure 8.1: Frame transformations in the representation of a point.

Let a point P be represented by the vector ${}^0\mathbf{p}$, with respect to the origin of a Frame \mathcal{F}_0 , described by the origin O_0 and the axes x_0, y_0, z_0 (for brevity $O_0-x_0y_0z_0$), as depicted in Figure 8.1. The same point can be equivalently expressed by the vector ${}^1\mathbf{p}$, with respect to another Frame \mathcal{F}_1 , described by $O_1-x_1y_1z_1$, which has been obtained by translating and

rotating \mathcal{F}_0 .

Summing up, to express the position of point P with respect to \mathcal{F}_0 , it can be written:

$${}^0\mathbf{p} = {}^0\mathbf{o}_1 + {}^0\mathbf{R}_1 {}^1\mathbf{p}, \quad (8.1.1)$$

where ${}^0\mathbf{o}_1$ is the vector describing the translation of O_1 with respect to O_0 , and ${}^0\mathbf{R}_1$ the rotation matrix of \mathcal{F}_1 with respect to \mathcal{F}_0 . To make the notation clear, the generic transformation ${}^i\mathbf{T}_j$ is the necessary transformation to change the reference from the Frame \mathcal{F}_i to the Frame \mathcal{F}_j .

To describe, in a more compact way, the point coordinates with respect to two different frames, the *homogeneous representation* of the vector \mathbf{p} can be introduced:

$$\tilde{\mathbf{p}} = \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix}. \quad (8.1.2)$$

This allows to rewrite equation (8.1.1) as it follows:

$${}^0\tilde{\mathbf{p}} = {}^0\mathbf{T}_1 {}^1\tilde{\mathbf{p}}, \quad (8.1.3)$$

where

$${}^0\mathbf{T}_1 = \begin{bmatrix} {}^0\mathbf{R}_1 & {}^0\mathbf{o}_1 \\ \mathbf{0}_3^T & 1 \end{bmatrix} \quad (8.1.4)$$

is the *homogeneous transformation matrix* of \mathcal{F}_1 with respect to \mathcal{F}_0 .

From a topological point of view, this matrix belongs to the *special Euclidean group* $SE(3) = \mathbb{R}^3 \times SO(3)$, because it is derived from a translation ${}^0\mathbf{o}_1 \in \mathbb{R}^3$ and from a rotation matrix ${}^0\mathbf{R}_1 \in SO(3)$.

The inverse transformation is:

$${}^1\tilde{\mathbf{p}} = {}^1\mathbf{T}_0 {}^0\tilde{\mathbf{p}} = ({}^0\mathbf{T}_1^{-1}) {}^0\tilde{\mathbf{p}}, \quad (8.1.5)$$

where, due the orthogonality property ${}^j\mathbf{R}_i = ({}^i\mathbf{R}_j)^{-1} = ({}^i\mathbf{R}_j)^T$,

$${}^1\mathbf{T}_0 = \begin{bmatrix} {}^0\mathbf{R}_1^T & -{}^0\mathbf{R}_1^T {}^0\mathbf{o}_1 \\ \mathbf{0}_3^T & 1 \end{bmatrix} = \begin{bmatrix} {}^1\mathbf{R}_0 & -{}^1\mathbf{R}_0 {}^0\mathbf{o}_1 \\ \mathbf{0}_3^T & 1 \end{bmatrix}. \quad (8.1.6)$$

Attention must be paid to the orthogonal property, because in general it does not hold in $SE(3)$:

$$\mathbf{T}^{-1} \neq \mathbf{T}^T. \quad (8.1.7)$$

Finally, the relation between two frames can be extended to n frames:

$${}^0\tilde{\mathbf{p}} = {}^0\mathbf{T}_1 {}^1\mathbf{T}_2 \dots {}^{n-1}\mathbf{T}_n {}^n\tilde{\mathbf{p}}. \quad (8.1.8)$$

8.2 Kinematic Chains

A manipulator is defined by a series of rigid bodies, called *links*, connected by kinematic pairs, called *joints*. The ensemble constitutes a kinematic chain. One end of the chain is attached to a base (that, in case of mobile manipulators, is indeed not fixed), and the corresponding link is fixed and is called *base link*. The other end is connected to a tool performing the manipulator task. This tool is normally referred to as *end-effector* and it can be a gripper, a robotic hand, a nozzle for painting, and so on, depending on the task the robot is designed for.

The kinematic chain can be either open or closed, depending on its structure.

Open chain

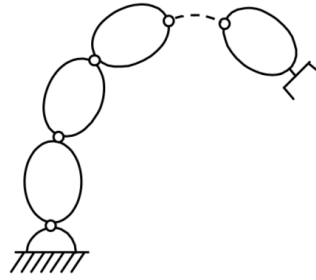


Figure 8.2: Schematic representation of an open kinematic chain.

A kinematic chain is said to be *open* when the sequence of links, connecting the base to the end-effector, is unique. Manipulators having an open kinematic chain are called *serial* (Figure 8.2). Manipulators of this type are the most commonly used.

Closed chain

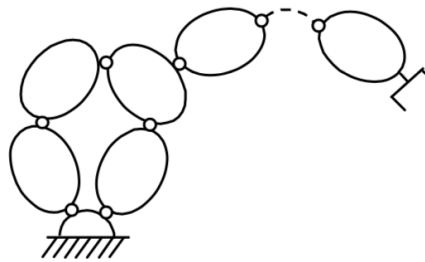


Figure 8.3: Schematic representation of a closed kinematic chain.

A kinematic chain is said to be *closed* when the sequence of links, connecting the base to the end-effector, forms a loop. Manipulators having a closed kinematic chain are called *parallel* (Figure 8.3). This type of manipulator is more rigid and, for this reason, more precise. They will not be discussed further in the thesis.

Each body of the chain is defined by its *pose*, while the *posture* of the manipulator is uniquely obtained by combining the poses of all the bodies. Equivalently, a manipulator can be described by the state of each of its joints. Each joint represents a degree of freedom¹, and it is associated with a *joint variable*. The configuration of all joints spans the so called *joint space*, which defines the DOF of the robot, while the end-effector lives in the *task space*.

Type of joints

The simplest joints are of two types, *prismatic* or *revolute*. These joints limit the mobility of the links they are connecting to one DOF:

- **Prismatic:** the motion between the connected links is reduced to a translation along the common axis, thus the relative position of the links is given by their distance

¹This is true for the basic type of joints only, i.e. *prismatic* or *revolute*. For example, a *spherical* joint has 3 DOF, but it can be obtained by three consecutive revolute joints.

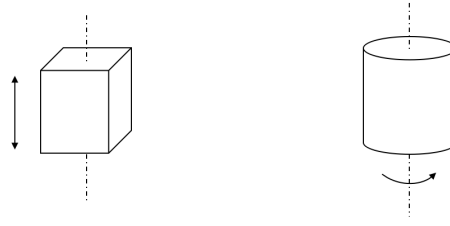


Figure 8.4: Schematic representation a *prismatic* (left) and a *revolute* (right) joint.

along the axis. This joint is denoted by a “*P*” and is schematically represented in Figure 8.4 (left).

- **Revolute:** the motion between the connected links is reduced to a rotation around the common axis, thus the relative position of the links is given by the angle with the axis. This joint is denoted by “*R*” and is schematically represented in Figure 8.4 (right).

In light of the above, a manipulator is said to be *redundant* if the robot has more DOF than the end-effector, i.e. the dimension of the joint space is higher than the dimension of the task space.

For example, YuMi is a 7R redundant robot because it has 7 links, so 7 DOF, but the end effector lives in the task space so it has only 6 dimensions. The *direct* kinematic model of a manipulator can be derived by computing the pose of the end effector as a function of these joint variables. In other words, the objective is to compute the homogeneous transformation matrix from the base frame to the end-effector frame:

$${}^b\mathbf{T}_e(\mathbf{q}) = \begin{bmatrix} {}^b\mathbf{n}_e(\mathbf{q}) & {}^b\mathbf{s}_e(\mathbf{q}) & {}^b\mathbf{a}_e(\mathbf{q}) & \mathbf{p}_e^b(\mathbf{q}) \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad (8.2.1)$$

where \mathbf{q} is the vector of joint variables, of dimension $(n \times 1)$ with n the number of joints; \mathbf{n}_e , \mathbf{s}_e , \mathbf{a}_e are the unit vectors defining the end-effector frame and \mathbf{p}_e is the position vector identifying the origin of the end-effector frame with respect to the origin of the base frame. When the end-effector is a gripper, its origin is the center of the gripper, \mathbf{a}_e is in the direction pointing to the object (*approach*), \mathbf{s}_e is in the *sliding* plane of the “*fingers*” and normal to \mathbf{a}_e , \mathbf{n}_e is *normal* to the two others. Assuming that the manipulator has n DOF, the direct kinematic model can be described by:

$${}^b\mathbf{T}_e(\mathbf{q}) = {}^b\mathbf{T}_0^0 \mathbf{T}_n(\mathbf{q})^n \mathbf{T}_e = {}^b\mathbf{T}_0^0 \mathbf{T}_1(q_1)^1 \mathbf{T}_2(q_2) \dots {}^{n-1} \mathbf{T}_n(q_n)^n \mathbf{T}_e \quad (8.2.2)$$

where ${}^b\mathbf{T}_0$ and ${}^n\mathbf{T}_e$ are two constant homogeneous transformations between frames and, hence, they do not depend on \mathbf{q} .

The *inverse* kinematic model, on the contrary, is derived by computing the joint configuration starting from the pose of the end-effector.

While equation (8.2.2) allows to uniquely determine the pose of the end-effector, once the transformations are known, the inverse kinematic problem is far more complex for these reasons:

- the equations that intervene are in general non linear;
- multiple solutions may exist;
- infinite solutions may exist in the case of a redundant manipulator;
- there can be no admissible solutions.

8.3 The Denavit-Hartenberg convention

To compute the direct kinematic model it is necessary to define the relative pose of two consecutive links. This can be done by attaching a frame to each link and computing the coordinate transformations between them. As it has already been said in the first part of the chapter, it is possible to arbitrarily attach the frames to a body, but this can lead to complex computations. The *Denavit-Hartenberg convention* (DH) proposes to set some rules for the definition of the link frames, in order to simplify the computations and to make the derivation of the direct kinematic model uniform for all the manipulators.

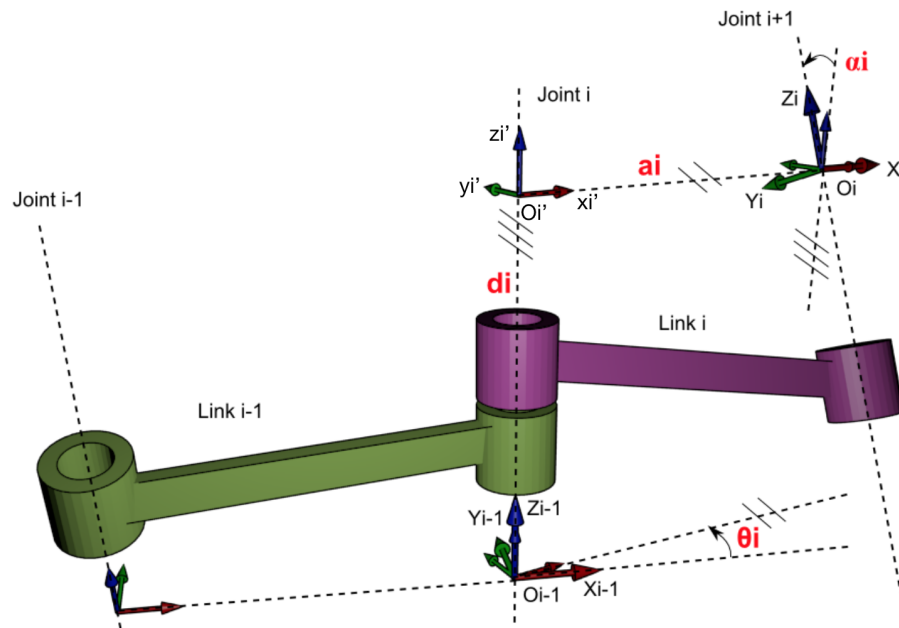


Figure 8.5: Parametrization of the classic Denavit-Hartenberg convention.

As schematically represented in Figure 8.5, the axis of the i -th joint connects the link $i - 1$ to the link i . The frame \mathcal{F}_i attached to the link i is defined as it follows:

- the axis z_i is chosen along the axis of the joint $i + 1$;
- the intersections of the axis z_i and z_{i-1} with their common normal identify the two origins O_i , for z_i , and O_{i-1} , for z_{i-1} ;
- the axis x_i is chosen along the common normal to z_i and z_{i-1} with direction from joint i to joint $i + 1$;
- the axis y_i completes the right-handed triad.

The definition of the link frame provided by this convention is non unique in the following cases:

- for the frame \mathcal{F}_0 , only z_0 is defined, so O_0 and x_0 can be arbitrarily chosen;
- for the frame \mathcal{F}_n , z_n is not uniquely defined because there is no joint $n + 1$, the axis x_{n-1} instead has to be normal to axis z_{n-1} . If joint n is revolute, z_n is chosen to be aligned with z_{n-1} ;
- when z_i and z_{i-1} are parallel, their common normal is not unique;
- when z_i and z_{i-1} intersect, the direction of x_i is arbitrarily chosen;
- when joint i is prismatic, the direction of z_{i-1} is arbitrarily chosen.

Now that the frames \mathcal{F}_i and \mathcal{F}_{i-1} have been identified, the position and orientation of the former with respect to the latter are described by the following parametrization:

- a_i : the distance between O_i and $O_{i'}$;
- d_i : the coordinate of $O_{i'}$ along z_{i-1} : this is also the variable of the prismatic joint i ;
- α_i : the angle between z_{i-1} and z_i around x_i , which is positive when the rotation is counter-clockwise;
- θ_i : the angle between x_{i-1} and x_i around z_{i-1} , which is positive when the rotation is counter-clockwise: this is also the variable of the revolute joint i .

This leads to two homogeneous transformation matrix. The first one expresses the transformation between the frames \mathcal{F}_{i-1} and $\mathcal{F}_{i'}$, obtained with a translation of d_i along z_{i-1} and a rotation of θ_i around the same axis:

$${}^{i-1}\mathbf{T}_{i'} = \begin{bmatrix} c_{\theta_i} & -s_{\theta_i} & 0 & 0 \\ s_{\theta_i} & c_{\theta_i} & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad (8.3.1)$$

where $c_\theta = \cos(\theta)$ and $c_{\theta_1\theta_2\dots\theta_n} = \cos(\theta_1 + \theta_2 + \dots + \theta_n)$, and equivalently for the sine. The second transformation is obtained from $\mathcal{F}_{i'}$, with a translation of a_i along $x_{i'}$ and a rotation of α_i about the same axis, to obtain \mathcal{F}_i :

$${}^{i'}\mathbf{T}_i = \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & c_{\alpha_i} & -s_{\alpha_i} & 0 \\ 0 & s_{\alpha_i} & c_{\alpha_i} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (8.3.2)$$

The complete transformation between the frames \mathcal{F}_{i-1} and \mathcal{F}_i is obtained by multiplying the two matrices:

$${}^{i-1}\mathbf{T}_i(q_i) = {}^{i-1}\mathbf{T}_{i'} {}^{i'}\mathbf{T}_i = \begin{bmatrix} c_{\theta_i} & -s_{\theta_i}c_{\alpha_i} & s_{\theta_i}s_{\alpha_i} & a_ics_{\theta_i} \\ s_{\theta_i} & c_{\theta_i}c_{\alpha_i} & -c_{\theta_i}s_{\alpha_i} & a_ics_{\theta_i} \\ 0 & s_{\alpha_i} & c_{\alpha_i} & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (8.3.3)$$

The transformation matrix (8.3.3) is a function only of the joint variable q_i , i.e. θ_i for a revolute joint or d_i for a prismatic joint.

To conclude, a manipulator is schematically parametrized by a table listing all the Denavit-Hartenberg parameters of each link, like the one in Figure 8.6. The transformation matrices of the direct kinematic model that appears in (8.2.2) can be computed from the one in (8.3.3), based on the parameters of the table. In the next section, the table for the IRB-14000 YuMi robot will be given.

Link	a_i	α_i	d_i	θ_i
\vdots	\vdots	\vdots	\vdots	\vdots
j	a_j	α_j	d_j	θ_j
\vdots	\vdots	\vdots	\vdots	\vdots

Figure 8.6: Denavit-Hartenberg parameters table.

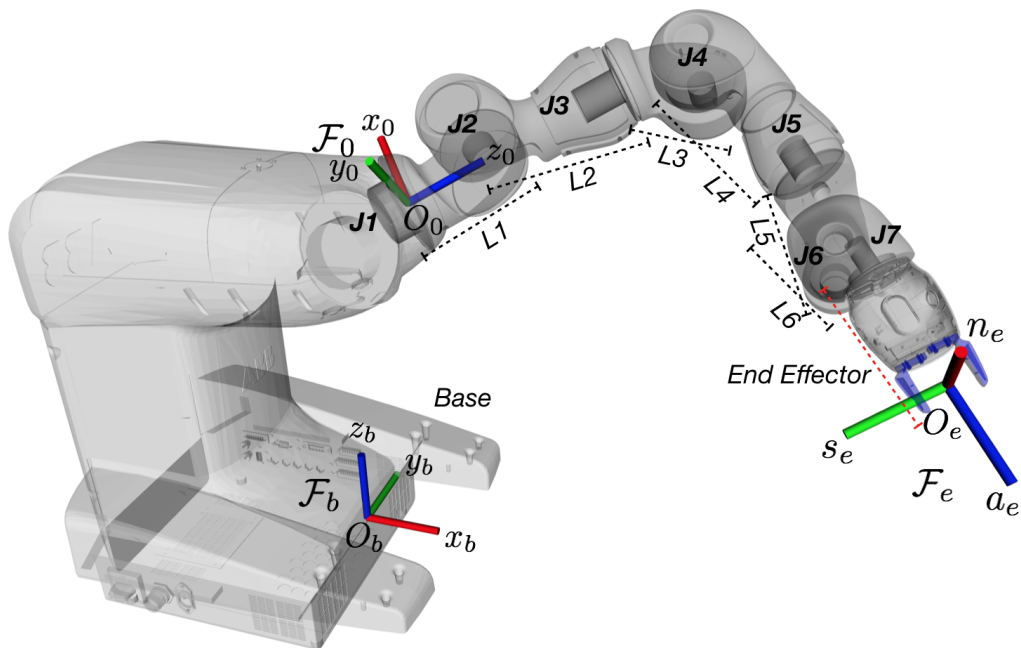
8.4 Kinematic model of the IRB-14000 robot

The DH table 8.7 for the arm of the YuMi robot is derived from the basis of the arm, i.e. the link 1, to the link 7, i.e. the end-effector, considering that the transformations from the robot base to the arm is constant.

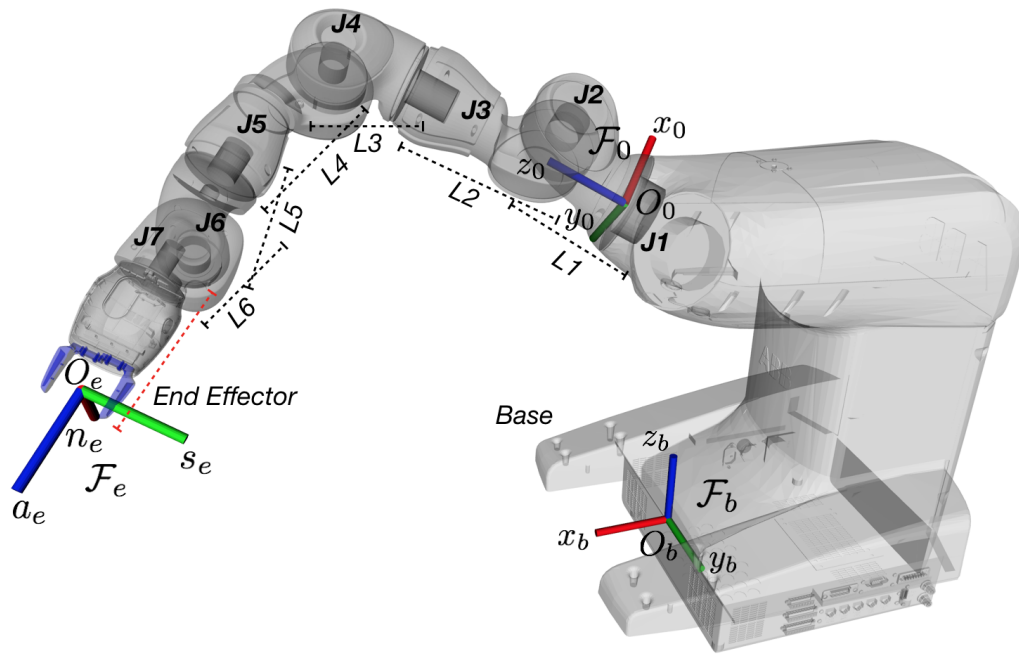
Link	a_i	α_i	d_i	θ_i
1	0	0	0	θ_1
2	0,03	$\pi/2$	0	θ_2
3	0,027	$-\pi/2$	0	θ_3
4	0,03	$\pi/2$	0,2515	θ_4
5	0,0405	$-\pi/2$	0	θ_5
6	0,0405	$-\pi/2$	0,265	θ_6
7	0,027	$-\pi/2$	0	θ_7

Figure 8.7: DH table for the arm of the YuMi robot.

In Figure 8.7, instead, the main frames of the robot arm, together with the links and the joints, are depicted, for both left and right arm.



(a) Left Arm.



(b) Right Arm.

Figure 8.7: The links, joints and main frames of the YuMi robot.

Trajectory Planning for Manipulators

The aim of this chapter is to recall the main topics of Section 5.1 and to adapt them to the case of robots manipulator, in order to present, at the end of this section, the planner used for the YuMi.

As already stated, the path planning algorithms represent the robot pose as a point \mathbf{q} in the *Configuration space* \mathcal{Q} , which is in general more complex than a Euclidean space. For the YuMi robot, the configuration space is of dimension 7 and the configuration vector is defined by the seven joint variables θ_i , with $i = 1, \dots, 7$. The images of the obstacles in the configuration space define the *configuration space obstacle* \mathcal{QO} , while its complement is the *free configuration space* $\mathcal{Q}_{free} = \mathcal{Q} - \mathcal{QO}$.

The path planning algorithm is hence responsible for finding a trajectory that is entirely contained in \mathcal{Q}_{free} .

The algorithms discussed in this chapter belong to the category of probabilistic planners, which are discussed in the next section.

9.1 Probabilistic Planning

Probabilistic planners rely on sample-based methods, where a roadmap is built from a finite set of configurations belonging to \mathcal{Q}_{free} .

Definition 9.1.1 (Roadmap). *A roadmap $\mathcal{R} \subset \mathcal{Q}_{free}$ is a network of paths that describe the connectivity of \mathcal{Q}_{free} [32].*

For each iteration of the planner algorithm, a new configuration is sampled. If the sample causes a collision between the robot and the workspace obstacles, it is discarded, while, in the opposite case, it is added to the roadmap and connected to the other configurations. This kind of planner is more efficient than deterministic approaches if the configuration space is of high dimension. In this section, the two most famous probabilistic planners will be discussed, to conclude with a derivation of one of them, which is then used for planning the pick&place trajectory for the YuMi.

9.1.1 PRM Method

This section is based on [32] and [61].

The *Probabilistic Roadmap* (PRM) method represents the roadmap as a graph $\mathcal{R} = (V, E)$ where the vertices are the set of robot configurations, belonging to the free configuration space, while the edges are simple local paths connecting two configurations.

The method operates in two phases: the *learning* phase, detailed in pseudo code 9.1, and the *query* phase, presented in pseudo code 9.2. In the learning phase the roadmap is built by generating, for each iteration, a random configuration \mathbf{q}_{rand} from a uniform distribution. The generated sample is checked not to cause collisions, then, if so, it is added to the roadmap and connected (if possible) to near configurations in the roadmap, labeled \mathbf{q}_{near} . The near configurations are k nodes within a certain distance δ from \mathbf{q}_{rand} (in Figure 9.1a, the blue dotted line represents the case when a configuration is distant from another configuration more than δ). The distance between two configurations is a derivation of the

Euclidean distance in the configuration space. The connection between the configurations \mathbf{q}_{rand} and \mathbf{q}_{near} is made through a local path generated by a *local planner* Δ .

A standard approach for the local planner is to generate a rectilinear path in \mathcal{Q} between the two configurations and to sample it with a chosen resolution. Then, each sample of the straight path is tested for collision and, finally, the two configurations result connected in the roadmap if the whole path is collision-free (in Figure 9.1a, the red dotted line represents the case when the path causes a collision).

By linking the configurations, connected components are created in the roadmap.

The algorithm stops generating configurations when the number of nodes in the roadmap reaches a maximum threshold n .

Algorithm 9.1 PRM algorithm: learning phase, pseudo code

```

1: function BUILD_PRM( $n, k$ )
2:    $V \leftarrow \emptyset$ 
3:    $E \leftarrow \emptyset$ 
4:   while  $|V| < n$  do // number of nodes lower than  $n$ 
5:     do
6:        $\mathbf{q}_{rand} \leftarrow \text{random\_config}()$ 
7:       until  $\mathbf{q}_{rand} \in \mathcal{Q}_{free}$ 
8:       add  $\mathbf{q}_{rand}$  in  $V$ 
9:   for all  $\mathbf{q}_{rand} \in V$  do
10:     $N_q \leftarrow$  the  $k$  closest neighbors of  $\mathbf{q}_{rand}$  in  $V$  within  $\delta$ 
11:    for all  $\mathbf{q}_{near} \in N_q$  do
12:      if  $(\mathbf{q}_{rand}, \mathbf{q}_{near}) \notin E$  and  $\Delta(\mathbf{q}_{rand}, \mathbf{q}_{near}) \in \mathcal{Q}_{free}$  then
13:        add  $(\mathbf{q}_{rand}, \mathbf{q}_{near})$  in  $E$ 
14:   return  $\mathcal{R} = (V, E)$ 

```

In the query phase, the algorithm tries to connect, through a local path, both \mathbf{q}_{start} and \mathbf{q}_{goal} to the roadmap. Then, the method executes a graph search to find a connected sequence of edges, i.e. a path P , to connect \mathbf{q}_{start} to \mathbf{q}_{goal} (Figure 9.1b).

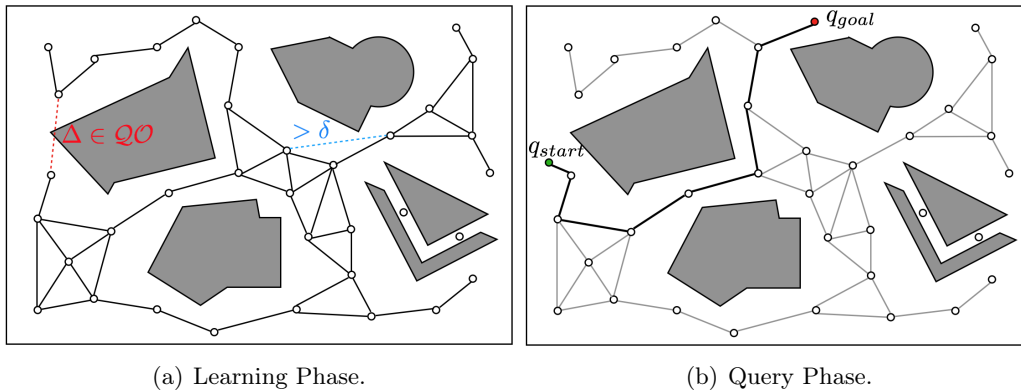


Figure 9.1: Scheme of the operating principle of the PRM Method for path planning.

The query phase is not executed once the learning phase is finished, but it is possible to alternate them. For instance, if the algorithm is not able to connect the initial and the goal configurations, it proceeds iterating and it continues to add new configurations.

The main drawback of the method is that the algorithm is only *probabilistically complete*, i.e. if the solution exists, the probability of finding it tends to 1 as the execution time tends to infinity. So, the algorithm may run indefinitely. One critical case is the presence of narrow passages because the probability of generating samples in such regions is low. This is due to the fact that using random distributions, the probability of placing a sample

Algorithm 9.2 PRM algorithm: query phase, pseudo code

```

1: function QUERY_PRM( $\mathbf{q}_{start}, \mathbf{q}_{goal}, k, \mathcal{R} = (V, E)$ )
2:
3:    $N_{q_{start}} \leftarrow$  the  $k$  closest neighbors of  $\mathbf{q}_{start}$  in  $V$  within  $\delta$ 
4:    $N_{q_{goal}} \leftarrow$  the  $k$  closest neighbors of  $\mathbf{q}_{goal}$  in  $V$  within  $\delta$ 
5:   add  $\mathbf{q}_{start}$  and  $\mathbf{q}_{goal}$  in  $V$ 
6:
7:    $\mathbf{q}_{near} \leftarrow$  closest neighbor of  $\mathbf{q}_{start}$  in  $N_{q_{start}}$ 
8:   do
9:     if  $\Delta(\mathbf{q}_{start}, \mathbf{q}_{near}) \in \mathcal{Q}_{free}$  then
10:      add  $(\mathbf{q}_{start}, \mathbf{q}_{near})$  in  $E$ 
11:     else
12:       $\mathbf{q}_{near} \leftarrow$  next closest neighbor of  $\mathbf{q}_{start}$  in  $N_{q_{start}}$ 
13:   until  $\mathbf{q}_{start}$  connected or no more  $\mathbf{q}_{near}$  in  $N_{q_{start}}$ 
14:
15:    $\mathbf{q}_{near} \leftarrow$  closest neighbor of  $\mathbf{q}_{goal}$  in  $N_{q_{goal}}$ 
16:   do
17:     if  $\Delta(\mathbf{q}_{goal}, \mathbf{q}_{near}) \in \mathcal{Q}_{free}$  then
18:      add  $(\mathbf{q}_{goal}, \mathbf{q}_{near})$  in  $E$ 
19:     else
20:       $\mathbf{q}_{near} \leftarrow$  next closest neighbor of  $\mathbf{q}_{goal}$  in  $N_{q_{goal}}$ 
21:   until  $\mathbf{q}_{goal}$  connected or no more  $\mathbf{q}_{near}$  in  $N_{q_{goal}}$ 
22:
23:    $P \leftarrow$  shortest_path( $\mathbf{q}_{start}, \mathbf{q}_{goal}, \mathcal{R}$ )
24:   if  $P \neq \emptyset$  then
25:     return  $P$ 
26:   else
27:     return failure

```

in a certain region is proportional to the region volume.

On the other end, the main advantages are the velocity in finding a solution in standard cases and the simplicity of implementation, in fact PRM does not need to represent \mathcal{Q} . To conclude, this method is said to be *multiple-query* because of the presence of the learning phase: it preprocesses the information about the configuration space and the configurations and store it in the data structure roadmap, to make the next planning queries faster.

9.1.2 RRT Method

This method differs from the previous one because it does not use a roadmap to represent the connectivity of the space \mathcal{Q}_{free} , but it tries to explore only the subset of the free configuration space that is relevant for the resolution of the current path planning problem. So, this method is even faster than the PRM. This kind of execution procedure is said to be *single-query*, i.e. the single-query path planning task computes a path from \mathbf{q}_{start} to \mathbf{q}_{goal} without performing the learning phase [32].

Unlike PRM, the Rapidly-exploring Random Tree method (RRT) is implemented with trees. A tree basically consists of a data structure containing an initial node, called root, and recursively defined subtrees. Each node of the tree, except for the root, has only one parent node and none or several children; if a node has no child, it is called a leaf.

The standard algorithm for the RRT method is described in [62] and reported in pseudo code 9.3. It performs one loop where, for each iteration, it tries to expand the tree, rooted in \mathbf{q}_{start} , by adding a new configuration \mathbf{q}_{new} . The function performing the expansion of the tree is called **EXTEND**. This function receives as inputs the tree to expand and a configuration \mathbf{q}_{rand} randomly generated. Then, the function searches the tree for the nearest configuration \mathbf{q}_{near} to \mathbf{q}_{rand} , with respect to a metric defined in \mathcal{Q} . Finally, the function **NEW_CONFIG** performs an extension motion, of fixed distance step ε , toward \mathbf{q}_{rand} . The configuration reached with such a step is called \mathbf{q}_{new} (Figure 9.2a: the expansion

function is also used in both RRT and RRT-Connect methods). This expansion has three outcomes:

1. if $\mathbf{q}_{new} = \mathbf{q}_{rand}$, i.e. the distance from \mathbf{q}_{near} to the random configuration is lower than ε , the function returns *Reached* and \mathbf{q}_{rand} is added;
2. if $\mathbf{q}_{new} \neq \mathbf{q}_{rand}$, the function returns *Advanced* and \mathbf{q}_{new} is added;
3. if $\mathbf{q}_{new} \notin \mathcal{Q}_{free}$, the function returns *Trapped* and \mathbf{q}_{new} is rejected;

Algorithm 9.3 RRT algorithm, pseudo code

```

1: function BUILD_RRT( $\mathbf{q}_{start}$ )
2:    $\mathcal{T}$ .init( $\mathbf{q}_{start}$ )
3:   for  $k = 1$  to  $K$  do
4:      $\mathbf{q}_{rand} \leftarrow$  random_config()
5:     extend( $\mathcal{T}$ ,  $\mathbf{q}_{rand}$ )
6:   return  $\mathcal{T}$ 
7:
8: function EXTEND( $\mathcal{T}$ ,  $\mathbf{q}$ )
9:    $\mathbf{q}_{near} \leftarrow$  nearest_neighbor( $\mathcal{T}$ ,  $\mathbf{q}$ )
10:  if new_config( $\mathbf{q}$ ,  $\mathbf{q}_{near}$ ,  $\mathbf{q}_{new}$ ) then
11:     $\mathcal{T}$ .add_vertex( $\mathbf{q}_{new}$ )
12:     $\mathcal{T}$ .add_edge( $\mathbf{q}_{near}$ ,  $\mathbf{q}_{new}$ )
13:    if  $\mathbf{q}_{new} = \mathbf{q}$  then
14:      return reached
15:    else
16:      return advanced
17:  return trapped
  
```

Moreover, during the tree expansion, it is checked whether the goal can be reached or not. If it is the case, the goal is added as a node and the algorithm stops, if not, another iteration is performed.

9.1.3 RRT-Connect

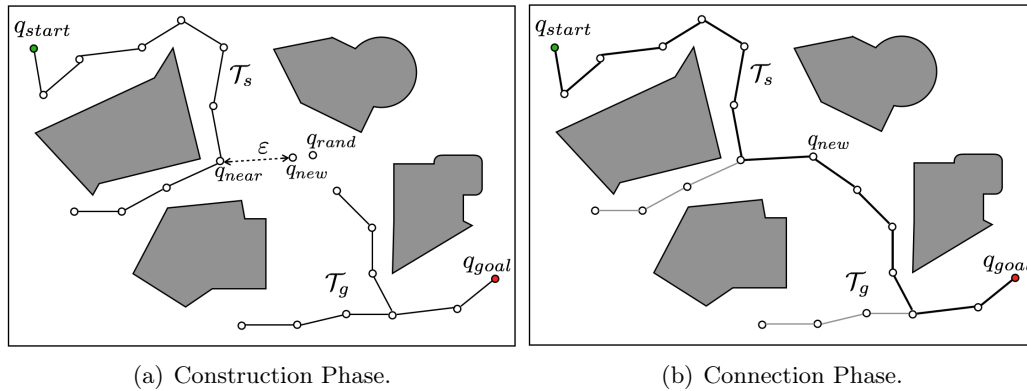


Figure 9.2: Scheme of the operating principle of the RRT-Connect Method for path planning.

This method, also called bidirectional-RRT, is proposed both in [62] and [32], and detailed in pseudo code 9.4. The name is justified by the fact that the path planning task is performed by expanding two trees \mathcal{T}_s and \mathcal{T}_g , rooted in \mathbf{q}_{start} and \mathbf{q}_{goal} respectively. For each iteration, one tree is expanded by performing the EXTEND function, of the standard RRT, until the configuration \mathbf{q}_{rand} is added or the next suitable configuration for the expansion lies in an obstacle (Figure 9.2a). Then, the algorithm makes an attempt to

connect the two trees with the greedy heuristic **CONNECT**: the configuration \mathbf{q}_{rand} recently added to one tree is used as reference for the expansion of the other tree. If a connection is established, the path from \mathbf{q}_{start} to \mathbf{q}_{goal} is returned (Figure 9.2b), otherwise, the trees are swapped and another iteration is performed with the reversed roles for the trees.

Algorithm 9.4 RRT-Connect algorithm, pseudo code

```

1: function RRT_CONNECT_PLANNER( $\mathbf{q}_{start}, \mathbf{q}_{goal}$ )
2:    $\mathcal{T}_s$ .init( $\mathbf{q}_{start}$ )
3:    $\mathcal{T}_g$ .init( $\mathbf{q}_{goal}$ )
4:   for  $k = 1$  to  $K$  do
5:      $\mathbf{q}_{rand} \leftarrow$  random_config()
6:     if not (extend( $\mathcal{T}_s, \mathbf{q}_{rand}$ )=trapped) then
7:       if (connect( $\mathcal{T}_g, \mathbf{q}_{rand}$ )=reached) then
8:         return path( $\mathcal{T}_s, \mathcal{T}_g$ )
9:     swap( $\mathcal{T}_s, \mathcal{T}_g$ )
10:  return failure
11:
12: function CONNECT( $\mathcal{T}, \mathbf{q}$ )
13:  do
14:     $S \leftarrow$  extend( $\mathcal{T}, \mathbf{q}$ )
15:  until not  $S$ =advanced
16:  return  $S$ 

```

To conclude, the bidirectional RRT method is faster than the standard one, which, in turn, is faster than PRM because it skips the learning phase. However, PRM is preferred when the manipulator has to perform the same task repeatedly and in the same obstacle scene (i.e. a fixed workstation). In such a case the learning phase is executed only the first time and the roadmap is defined once for good, allowing to perform subsequent queries faster. On the other hand, RRT methods are preferred for single-query path planning.

Simulating and Testing YuMi with ROS

The aim of this last chapter is to detail the application of the planning algorithms to the robot manipulator both in simulation and in practice, as it was done for the mobile platform. Initially, only YuMi is used, to understand how to perform a pick&place task, then, it will cooperate and interact with Ridgeback to accomplish an objective different from the one presented in the introductory chapter. The robot will move from a starting position to a table with a workstation, then it will perform a manipulation task and finally return to the initial position.

Chapter 2 presented RobotStudio as the software environment where ABB robots are programmed and simulated. Ridgeback, on the other hand, is simulated in ROS, as detailed in Chapter 6. So, it is necessary to have some kind of tool to interface programs written in C++, the ROS language, with RobotStudio and its programming language, i.e. RAPID. This tool is called Robot Web Services (RWS) and it is a C++ library that allows to interface C++ programs with the controller of ABB robots, with the possibility of exchanging I/O signals, RAPID codes and Configuration files, through HTTP and TCP communications (Figure 10.1) [63].

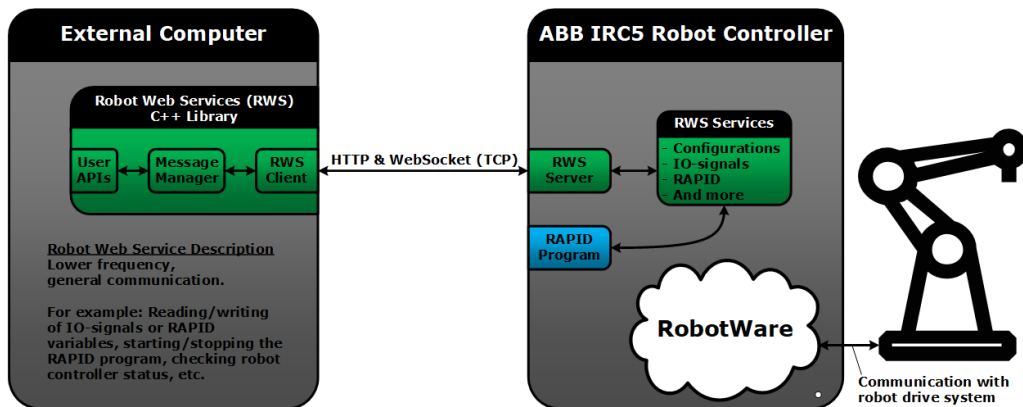


Figure 10.1: Robot Web Services library.

Once the two robots communicate, the ROS algorithms need a unified model of the mobile manipulator, in the form of an `urdf` file, as support for the simulations and the displaying of the robot in RViz. By including, in one single file, the already existing `urdf` models of the Ridgeback and the YuMi and by adding the models of the two pedestals, it was finally possible to obtain the mobile manipulator in Figure 10.2.

The last needed component is a software implementing the algorithms presented in Chapter 9, to compute and simulate trajectories for pick&place tasks. The software in charge of doing it is called MoveIt! and it is available in ROS. This will be the subject of the next section.

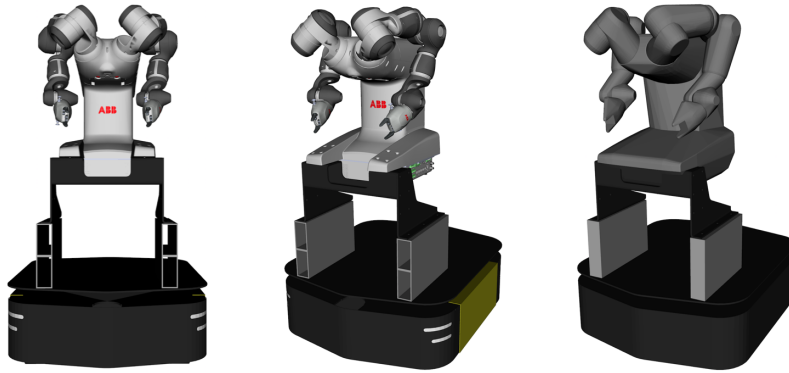


Figure 10.2: URDF model of the mobile manipulator Ridgeback-YuMi and its collision model (right).

10.1 ROS MoveIt!



Figure 10.3: MoveIt! logo.

MoveIt! is a software implementing algorithms for motion planning, manipulation, 3D perception, kinematics, control and navigation [64]. Here, it is used to realize the computations for the manipulator robot. As it was done for the Ros Navigation Stack in Section 6.3, the MoveIt! system architecture [65] sketched in Figure 10.4 will be presented hereafter.

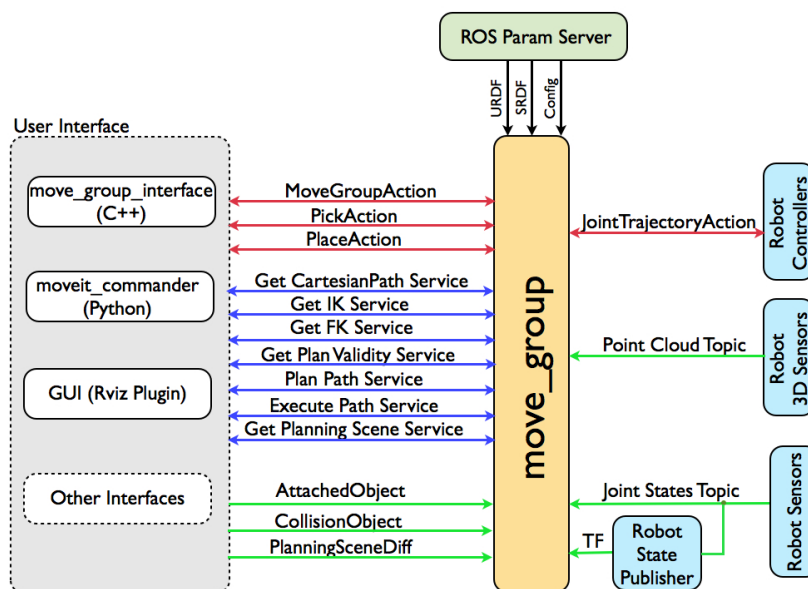


Figure 10.4: MoveIt! system architecture.

The most important node provided by MoveIt! is `move_group`, which gathers and handles information and data, to make ROS action and services at one's disposal.

User Interface MoveIt! provides a C++ package to allow the user to interact with the `move_group` node by exploiting pre-defined actions to move a robot arm or to command a pick or a place action. Moreover, in the user interface there are useful plugins for RViz that allow the user to act directly on the robot by changing the configuration of the end-effector of the robot arm and then to query path planning and execution (Figure 10.5). Finally, MoveIt! provides some interfaces that allow to build a planning scene, i.e. the representation of the world and the current state of the robot, containing some objects for the robot to pick as well as obstacles to avoid (Figure 10.6).

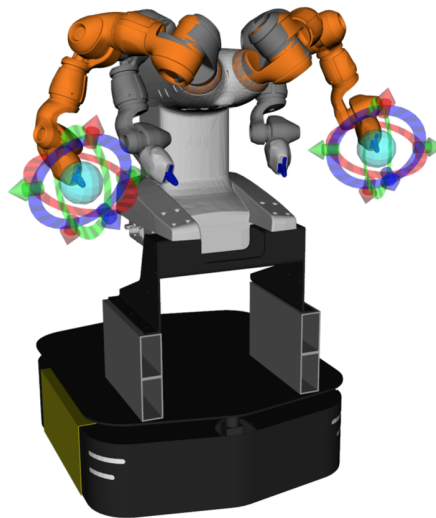


Figure 10.5: Planning request for the robot arms in MoveIt!.

Robot setup MoveIt!, via the `move_group` node, gathers informations about the robot and the task scene. The joint state information are retrieved through the topic `/joint_states`, where the configurations of the robot joints are stored. Then, the node uses the TF library in ROS to monitor the transformations provided by the robot, in order to estimate the pose of the robot and carry out the computations for the planning. Finally, the `move_group` node communicates with the controllers on the robot so that it is possible to command the robot joints to follow a trajectory.

Motion Planning The motion plan request can be done both in joint and task space, so that it is possible to command each joint separately or the end-effector position only. In the computation of the plan, the collision are checked with external objects and also within the robot links. For pick&place tasks, the object to pick can be attached to the robot, so it is taken into account and tested for collisions in the placing motion. To personalize the plan request, it is possible to set some more constraints, for example to restrict the position and orientation of a joint during the motion.

10.2 Simulating manipulation tasks in MoveIt!

To set up the pick&place task, the following hypothesis were made:

- the position of the object to pick is known;
- the task is pick&place only: no navigation is involved;
- the controllers of the arms of the simulated robot are supposed to be ideal, i.e. once the trajectory has been checked to be collision free, the controllers realize the command perfectly.

Then, a node was created to build the scene with a table, a coffee machine and a cylinder simulating the coffee cup, as it is showed in the pick&place tutorial [66] and depicted in Figure 10.6. The node computes the trajectory once the following features have been specified:

- the picking pose of the end-effector chosen for the grasping, specifying also the behavior in approach and retirement;
- the placing pose of the end-effector chosen for the grasping, specifying also the behavior in approach and retirement;
- the object to grasp and its position with respect to a reference frame;
- the planner to use, chosen from the list available in [67], which belongs to the Open Motion Planning Library [68].

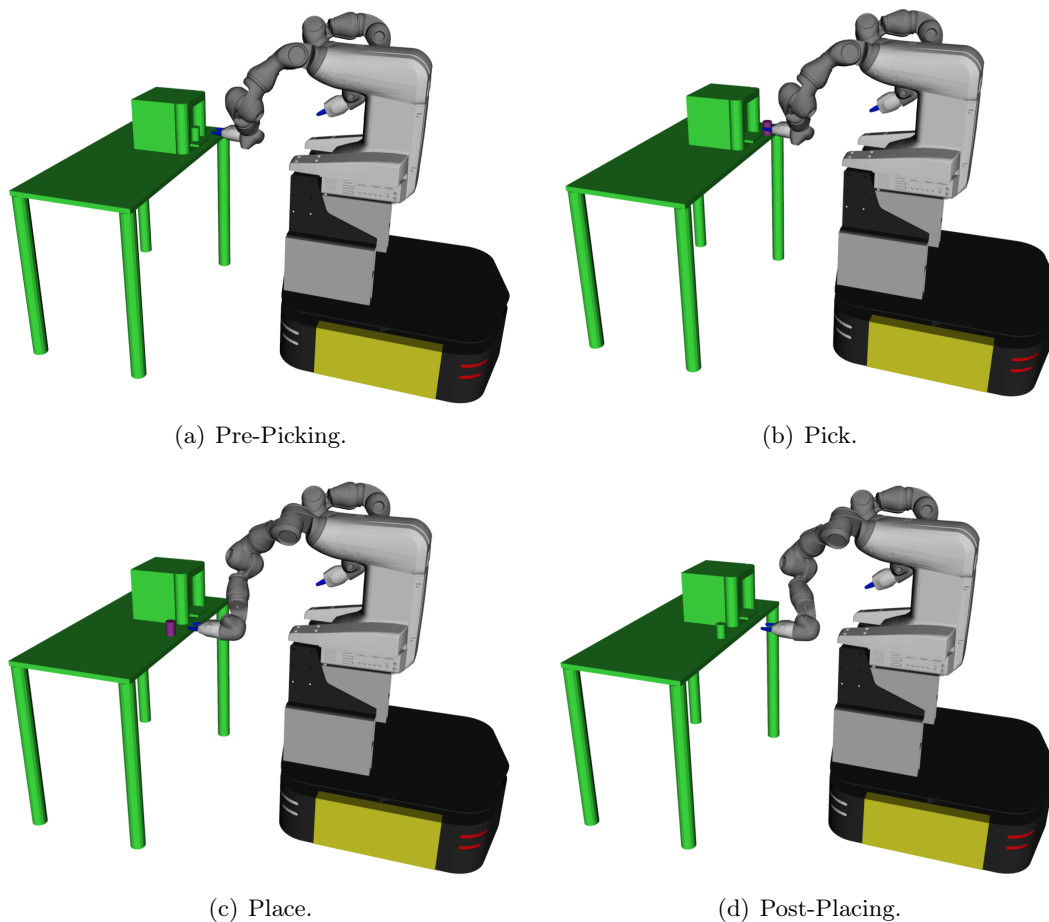


Figure 10.6: Main phases of the simulated pick&place task.

In Figure 10.6, the main phases of the simulation are represented:

- a pre-picking pose for the end-effector is set in front of the object to grasp: the green cylinder which simulates a coffee cup (Figure 10.6a);
- the picking movement is realized and the object is colored in purple because it becomes part of the robot arm and it is taken into account to compute a collision free path (Figure 10.6b). Moreover, the grippers are commanded to close;
- the object is placed in a defined position and the grippers are commanded to open (Figure 10.6c);
- a post-placing pose is given to the end-effector and the object is no longer considered as being part of the arm (Figure 10.6d).

For the simulation, we specified “yumi_base_link” as reference frame for the position of the object to grasp, corresponding to “YuMi Base” in Figure 10.7, and as planner the RRT-Connect, detailed in Section 9.1.3.

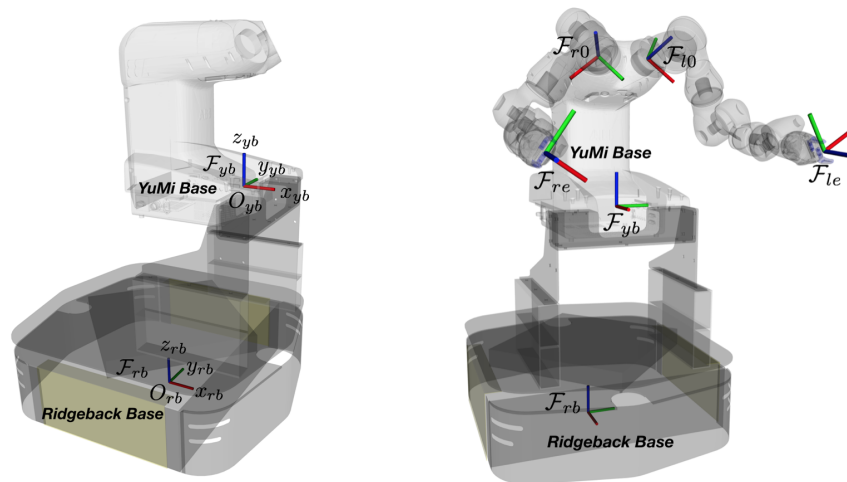


Figure 10.7: Main frames of the kinematic chain from the Ridgeback base to the YuMi end-effector.

The simulation qualitatively works as intended, indeed the planner computes a collision free trajectory for the robotic arm, allowing to achieve the pick and then the place of a cylindrical object, whose position is known. However, it is far from a real case where the navigation has to be taken into account and the controllers are not ideal. Moreover, in a real application, the object to grasp is not defined with respect to a frame of the robot, but rather with respect to the fixed frame of the map. Finally, using an ABB robot as manipulator, a node implementing RWS is needed, to translate the trajectory planned by MoveIt! into RAPID code.

The first of these modifications concerns the behavior of the global planner, as it is described hereafter.

10.3 Combining Navigation and Manipulation

Up to now, the global planner of the mobile platform computes a collision free plan from a starting point to a goal, points set with respect to the Ridgeback base frame (Figure 10.7), located in the center of the robot base. So, if the goal is a position of an object to pick, two problems may rise:

1. if the object is on a table, the robot cannot reach it because the goal is on an obstacle;
2. if the object is in an allowed position, the robot can reach it but the manipulator may not be able to perform the picking task.

For these reasons, the goal position needs to be set in an ideal location for the YuMi to have access to it. Since no cameras are available to set the goal using object recognition, the behavior of the global planner had to be modified, as it is detailed in the next section.

10.3.1 Offset Planner for the Ridgeback

The desired behavior for the global planner of the mobile platform is to make the robot arrive within a certain offset from the chosen goal, which is why we named this planner “Offset Planner”.

Offset Planner is derived from `GlobalPlanner`, a class of global planners like `NavFn`. Unlike

the latter, the former does implement A* but it does not consider the goal tolerance. So, if with NavFn it is possible to set a goal *on* an obstacle to make the robot reach the nearest allowed position to that obstacle, with GlobalPlanner it is not, which is why NavFn was preferred for all the applications that concerned the Ridgeback *alone*.

The function `makePlan` of the `GlobalPlanner` class computes a global plan using A*.

The Offset Planner computes the best goal position among the possible ones belonging to a circle, whose radius is equal to an offset specified in a configuration file. To test all the cells in the circumference, the algorithm computes an angle step that depends on the resolution of the map. Then, for each of these cells, the algorithm retrieves the occupancy cost value, to discard the positions corresponding to obstacle cells. The loop stops either as soon as it finds a position with cost 0 (no obstacle), or when all the positions are checked and the one with the lowest cost is returned.

Once the goal position has been identified, also the goal orientation is changed in order to make the robot face the object.

Algorithm 10.1 Offset Planner, pseudo code

```

1: function OFFSETPLANNER::MAKEPLAN(start, goal, plan)
2:
3:   offset_goal ← goal
4:   // return if there is no valid offset
5:   if offset ≤ costmap.getResolution() then
6:     return GlobalPlanner::makePlan(start, offset_goal, plan)
7:   // goal in map coordinates
8:   mx_goal, my_goal ← null
9:   costmap.mapToWorld(goalx, goaly, mx_goal, my_goal)
10:
11:   angle_step ← costmap.getResolution()/offset // to test all cells
12:   best_mx ← mx_goal
13:   best_my ← my_goal
14:   best_cost ← 255
15:   map_offset ← offset/costmap.getResolution()
16:
17:   // search for the best point given the goal and the offset to apply
18:   for angle = 0 to 2π do
19:     mx ← mx_goal + map_offset × cos(angle)
20:     my ← my_goal + map_offset × sin(angle)
21:     if mx < 0 or my < 0 or mx > costmap.getMapSizex() or my > costmap.getMapSizey() then
22:       continue
23:     cost ← costmap.getCost(mx, my)
24:     if cost < best_cost then
25:       best_cost ← cost
26:       best_mx ← mx
27:       best_my ← my
28:       if best_cost = 0 then // the best position has been found
29:         break
30:     angle ← angle + angle_step
31:
32:   costmap.mapToWorld(best_mx, best_my, offset_goalx, offset_goaly)
33:   // set the orientation (in quaternion) so that the robot faces the goal
34:   heading ← atan2(goaly - offset_goaly, goalx - offset_goalx)
35:   offset_goalz ← sin(heading/2)
36:   offset_goalw ← cos(heading/2)
37:
38:   return GlobalPlanner::makePlan(start, offset_goal, plan)

```

10.3.2 The Station Traveling Node

The planner described in the above section was used when conceiving the coffee making application. The complete task is sketched in Figure 10.8 and it can be described as it follows:

1. the mobile manipulator starts from an initial position that can simulate, for example, a charging station;
2. the mobile manipulator navigates the environment towards the first workstation, where a coffee machine and a cup are located;
3. the robot grasps the coffee cup and place it in the coffee machine, with an arm, then, with the other arm, it pushes a button on the coffee machine to make the coffee;
4. the mobile manipulator grasps the cup and places it on another workstation;
5. the robot finally comes back to the charging station.

To perform this complete task, a node called “*station traveling*” was written. The node processes the station coordinates, defined in a `.yaml` file, in order to set them as a goal for the global planner. The node loop works as a state machine, alternating the navigation mode with the manipulation mode. This interaction is realized with I/O signals, sent from the Ridgeback to YuMi and vice-versa. In the manipulation mode, a task is defined according to the reached station. The planning is realized with MoveIt! and then translated, via the node itself, into RAPID code. Since the obstacle avoidance of the Ridgeback does not take into account the volume occupied by its charge, the YuMi must be in the most compact configuration possible, and it does not have to move during the navigation.

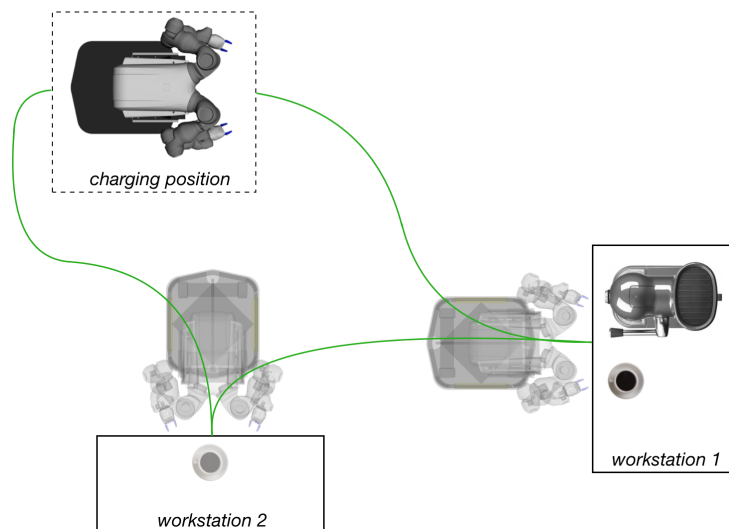


Figure 10.8: Scheme of the robot traveling within workstations to accomplish the pick&place task.

The realization of this application was soon abandoned for two reasons:

1. other projects involving the mobile manipulator overlapped with this one, leaving not enough time to complete the development of the MoveIt! part of the node;
2. the application itself is intrinsically destined to fail. Indeed, without a vision system, there is no movement correction when approaching the object. Since the planning is computed under the hypothesis of known position for the object to pick, the precision of the mobile platform (2cm in the best case) is not enough to grant a planning success.

Yet, a complete application was finally realized and it is described in the next section.

10.4 Conclusions and future work



Figure 10.9: Three views of the real mobile manipulator.

The mobile manipulator used for this thesis work, consists of two robots: the mobile platform Ridgeback from Clearpath and the dual arm manipulator IRB-14000 (YuMi) from ABB (Figure 10.9).

The mobile platform is endowed with four Mecanum wheels for the motion, which allow the robot an omnidirectional displacement, and two laser range finders for the sensing. The mobile robot is responsible, in the first place, of building a map of the environment to navigate, then, of using the built map to plan collision free trajectories to bring the manipulator from one workstation to another. The process of building a map is made by using Simultaneous Localization and Mapping (SLAM) algorithms, which use a modified version of the particle filter to create a grid representation of the environment. Moreover, this process is fully autonomous thanks to a frontier-based exploration, where unexplored zones of the map are set as goal to explore for the robot. The trajectories for the robot motion are computed by a global planner, which calculates a collision free path from the starting point to the goal, and by the local planner, which tells the robot how to behave locally, i.e. it computes a velocity trajectory to cover small fractions of the global path, taking into account the accelerations, the distance from the goal and the path, the distance from the obstacles.

Once the map is available, the mobile robot uses the Augmented Monte-Carlo Localization (AMCL) algorithm to retrieve its position inside the map and then to perform the navigation. The AMCL uses particle filters to solve the global localization problem, i.e. the algorithm is able to retrieve the robot position during navigation, starting from an unknown initial position.

Concerning the manipulator, the IRB-14000 is endowed with two arms with 7 degrees of freedom each and the end-effector is a particular 3D printed gripper, designed to grasp objects with a cylindric shape (Figure 10.10). The trajectory planning for the manipulator is implemented with a Rapidly-exploring Random Tree (RRT) algorithm, which builds two trees, one rooted in the starting position and the other one rooted in the goal, and expands them by connecting randomly generated configurations.

The algorithms used for the mobile manipulator are written in C++ and interfaced with the Robot Operating System (ROS), which is an open source middleware conceived for



Figure 10.10: Standard gripper (on the left) compared to the special ones used (on the right).

robotic applications. The interaction of the two robots is made via I/O signals, that allow the robots to communicate and to switch from navigation mode to manipulation mode. The commands to the robot are sent from an external computer. With the introduction, in the mobile manipulator, of a Wi-Fi router, it is now possible to restore the wireless communication between the laptop and the robot (Figure 10.11). Nonetheless, connection losses still occur.

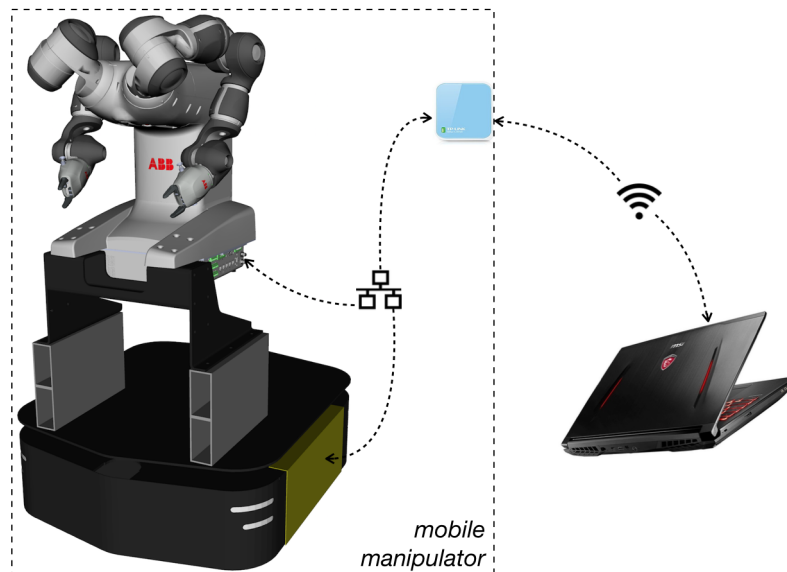


Figure 10.11: Scheme of the final communication setup of the mobile manipulator.

With reference to the mobile manipulators described in Chapter 7, the one used for this work presents some differences. The limitations of the mobile platform used here, have been already discussed in Section 6.8 and, to summarize, they concern the 2D scan area of the sensors that is not sufficient to grant obstacle avoidance for the whole mobile manipulator. A future work could be, in the first place, to implement the discussed navigation algorithms to deal with both front and rear sensors, then, to extend the sensing area of the robot by adding other sensors to scan in the third dimension, too. On the other hand, this mobile robot has a good payload and a sufficient maneuver precision, thanks to the Mecanum wheels.

Concerning the manipulator, it is clear that it is not powerful enough to be used in heavy objects manipulation tasks, but it is designed for small part assembly in an industrial context, such as electronic components. However, it has been chosen for research purposes and, being a dual arm manipulator, it can perform a broad number of tasks that go beyond the industrial needs. For example, the complete mobile manipulator has been used in simulating a hospital environment where it was dedicated to the preparation of medical samples, as detailed hereafter.



Figure 10.12: Scheme of a fixed workstation with features to retrieve the task frame.

For the simulation of a hospital environment, a table in the CobotLab was equipped as shown in Figure 10.12, to become the workstation for the mobile manipulator. The demo is composed by different parts:

1. the robot starts from an initial position, i.e. the charging station. It waits for a signal, coming from the feeder (the object on the right in Figure 10.12), to start the navigation: the signal indicates whether there are samples or not, indeed the robot does not have to go to the workstation if there are no samples to manipulate;
2. the mobile manipulator navigates the CobotLab, avoiding static obstacles and also dynamic ones if they occur, to reach the workstation in Figure 10.12. Once the station is reached, the mobile platform sends a signal to the YuMi robot allowing it to start the manipulation task;
3. the robot uses the hand cameras (Figure 10.10, right) to take a picture of the gear icons in Figure 10.12 and, consequently, to retrieve the frame of the fixed workstation to compute the planning accordingly;
4. once there are no more samples to manipulate in the feeder, YuMi stops its task, puts its arms in a compact position (the one in Figure 10.9) and sends a signal to the Ridgeback to start the following navigation task that takes the mobile manipulator back to the charging station.

For this application, the manipulation task, from the object recognition itself, is completely coded in RobotStudio in advance, i.e. if the workstation slightly changes, the task fails. Also, being coded in RobotStudio, the planner is not probabilistic. Finally, there is no obstacle collision, with the exception of the features of the workstation.

Having already tested the efficiency of the mobile platform in obstacle avoidance, this mobile manipulator could be exploited in environments populated by the human being, because the robot arms are not sufficiently powerful to cause some harm. Another limitation of the approach used here is that the planners for the manipulation tasks are probabilistic, which means that they are not repeatable because every time the task is queried, the outcome may be different. Industrial applications have to be the most repeatable possible, to lower uncertainties and failures in the execution. On the other hand, probabilistic planners are more flexible in obstacle avoidance, especially if visual sensors are available.

Moreover, the hardware at our disposal does limit the application. Up to now, an external laptop does all the computations and communicates with the mobile manipulator through a Wi-Fi router. Then, the communication among the two robots is wired. In the ideal case, it would be better to run all the computations in a powerful computer embedded in the mobile manipulator itself, and to leave to the external laptop only the sending of the commands. Communication losses, indeed, easily end up in collisions.

To conclude, the main difference between the industrial mobile manipulators and this one, is, for the latter, the lack of a vision system. This is simply because the six months that were dedicated to this work were not enough to provide the robot with a full operating vision system. The camera would be useful both for the mobile platform and for the manipulator. In the first case, cameras would be used for sensing the environment and improve obstacle avoidance and precision in goal reaching. Moreover, they could change the way the goal is sent to the robot. Up to now, the goal is a pre-determined position, in the map frame, which has to be registered in advance. With a camera, one could send as a goal, the model of an object, commanding the robot to reach the real one by exploiting object recognition. Moreover, the navigation and SLAM algorithms could be changed into their feature-based version, where the camera is used to recognize the key features in the map. For the mobile manipulator, cameras would be used for pick&place tasks, where the position of the object to pick is unknown a priori. In that case, the robot recognizes the object to pick and, with the camera model, it is able to compute the necessary transformations to grasp it. This kind of approach is known under the name of *visual servoing*, where the relative pose of the object to grasp with respect to the gripper, acquired by the camera, is used as feedback for the control system in charge of computing the trajectories. The natural continuation of this work is, hence, to implement a vision system for the mobile manipulator, which will allow to relax the hypothesis made in Section 10.2.

When the Dr. Pietro Falco presented us the project and the goals the team had to accomplish, the workload seemed overwhelming. The project is, indeed, really ambitious and the potential set of algorithms that could be tested with a mobile manipulator is incredibly wide. Later on, Dr. Falco proposed to limit the work only to the navigation, but the curiosity and the scientific thirst of knowledge pushed us to obtain the results that were presented in this last chapter.

We soon realized that it was impossible to integrate also the vision system, but I believe that the work accomplished is quite satisfactory. Moreover, the awareness of the skills acquired and the subjects learned is priceless, and these factors contributed to make me do a step forward in the robotics domain and its understanding.

Appendices

ROBOTICS

YuMi® creating an automated future together. You and me.



YuMi heralds a new era of robotic co-workers which are able to work side-by-side on the same tasks as humans while still ensuring the safety of those around it.

Collaboration

YuMi is the first truly collaborative dual armed robot, designed for a world in which humans and robots work together. Intentionally designed to resemble its human counterparts to be friendly and non-threatening with a compact, dual-arm body which requires no more space than a human workstation.

YuMi harnesses the enormous potential of human-robot collaboration in small parts assembly. YuMi offers manufacturers a transformational new solution, the first dual arm robot purpose-built for the small parts space: inherently safe, extremely accurate.

Each magnesium arm flexes on seven axes to mimic human-like movements with spatial efficiency. The robot was specifically designed to meet the flexible and agile production needs required by the consumer electronics industry.

Thanks to its dual-arms, flexible hands, universal parts feeding system, camera-based part location and state-of-the-art motion control, YuMi has equal application in any small parts assembly environment.

Redefining safety

YuMi represents a dramatic shift in how industrial robots operate.

At its core, YuMi has the DNA of safety. Much like a human arm has a skeleton covered with muscles that provide padding, YuMi has a lightweight yet rigid magnesium skeleton covered with a floating plastic casing wrapped in soft padding. This arrangement absorbs the force of any unexpected impacts to a very high degree. Like the human arm, YuMi has no pinch points so that sensitive ancillary parts cannot be crushed between two opposing surfaces as the axes open and close.

If YuMi senses an unexpected impact, such as a collision with a coworker, it can pause its motion within milliseconds, and the motion can be restarted again as easily as pressing play on a remote control. Additionally, the robot can rapidly diagnose changes in its environment and, if necessary, register the overload, shutting down YuMi's motion within milliseconds to prevent injury. When this is combined with the floating padding, safety for a human coworker is drastically increased. Even with these inherent safety features, YuMi is incredibly precise and fast, returning to the same point in space over and over again to within 0.02 mm accuracy and moving at a maximum velocity of 1,500 mm/sec.

This revolutionary integration of speed, agility and sensory advantage ensures the safety of human coworkers on production lines and in fabricating cells.

Innovative technology by design

In addition to being a global leader in the manufacture of industrial robots, ABB Robotics also develops software and manufactures hardware, peripheral equipment, process equipment and modular manufacturing cells. This “total solution” concept is evident in YuMi’s breakthrough design.

While designed specifically for the electronics industry, YuMi is also well suited to other small parts environments, including the manufacture of watches, toys and automotive components. These end-markets have been changing faster than the process improvements they demand – until now.

Features

- The fifth-generation, integrated IRC5 controller with TrueMove and QuickMove™ motion control technology commands accuracy, speed, cycle-time, programmability and synchronization with external devices.
- I/O interfaces include Ethernet IP, Profibus, USB ports, DeviceNet™, communication port, emergency stop and air-to-hands. YuMi accepts a wide range of HMI devices including ABB’s teach pendant, industrial displays, commercially available tablets and smartphones.
- The 100-240 volt power supply plugs into any power socket for worldwide versatility.

Benefits

- Can operate equally effectively side-by-side or face-to-face with human coworkers.
- Servo grippers (the “hands”) include options for built-in cameras
- Real-time algorithms set a collision-free path for each arm customized for the required task.
- Padding protects coworkers in high-risk areas by absorbing force if contact is made.
- If the robot encounters an unexpected object – even a slight contact with a coworker – it can pause its motion within milliseconds, and the motion can be restarted again as easily as pressing play on a remote control.
- Pinch points have been eliminated or minimized to an acceptable level between moving parts, and between moving and stationary parts.

Specification			
Robot Version	Reach	Payload	Armload
IRB 14000-0.5/0.5	559 m	500 g	No armloads

Features	
Integrated signal and power supply	24V Ethernet or 4 Signals
Integrated air supply	1 per Arm on tool Flange (4 Bar)
Integrated ethernet	One 100/10 Base-TX ethernet port/per arm
Position repeatability	0.02
Robot mounting	Table
Degree of protection	IP30
Controllers	Integrated

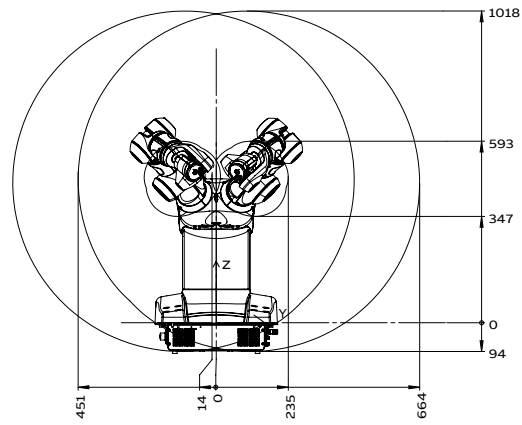
Safety specification	
Functional safety	PL b Cat B

Movement		
Axis movement	Working range	Maximum speed
Axis 1 rotation	-168.5° to 168.5°	180°/s
Axis 2 arm	-143.5° to 43.5°	180°/s
Axis 3 arm	-123.5° to 80.0°	180°/s
Axis 4 wrist	-290.0° to 290.0°	400°/s
Axis 5 bend	-88.0° to 138.0°	400°/s
Axis 6 turn	-229.0° to 229.0°	400°/s
Axis 7 rotation	-168.5° to 168.5°	180°/s

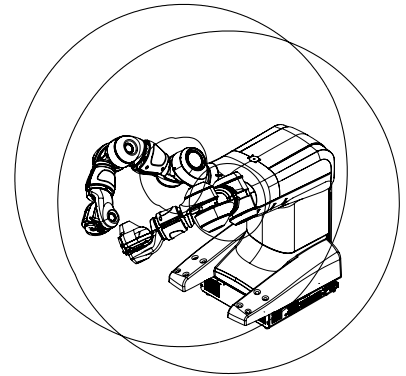
Physical order of the axis is 1,2,7,3,4,5,6

Performance	
0.5 kg picking cycle	
25* 300 * 25mm	0.86s
Max TCP Velocity	1.5 m/s
Max TCP Acceleration	11 m/s*s
Acceleration time 0-1m/s	0.12s

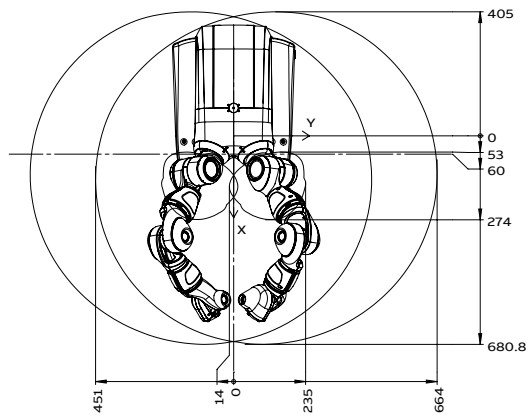
Physical	
Total bottom	399mm * 496mm
Toes	399mm * 134MM
Weight	38kg



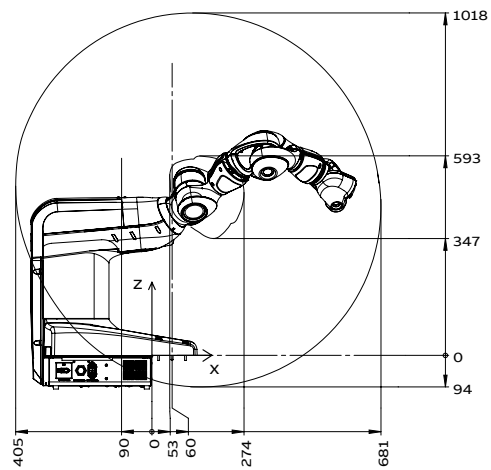
IRB 14000-0.5/0.5 Front view



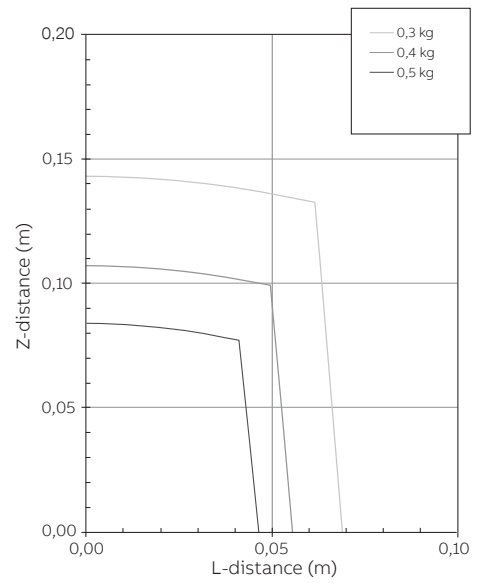
IRB 14000-0.5/0.5 Isometric view



IRB 14000-0.5/0.5 Top view



IRB 14000-0.5/0.5 Side view



Load range

—
ABB Engineering (Shanghai) Ltd.
No. 4528 Kangxin Highway
Pudong New District
201319, Shanghai, China
Phone: +86 21 6105 6666

abb.com/robotics

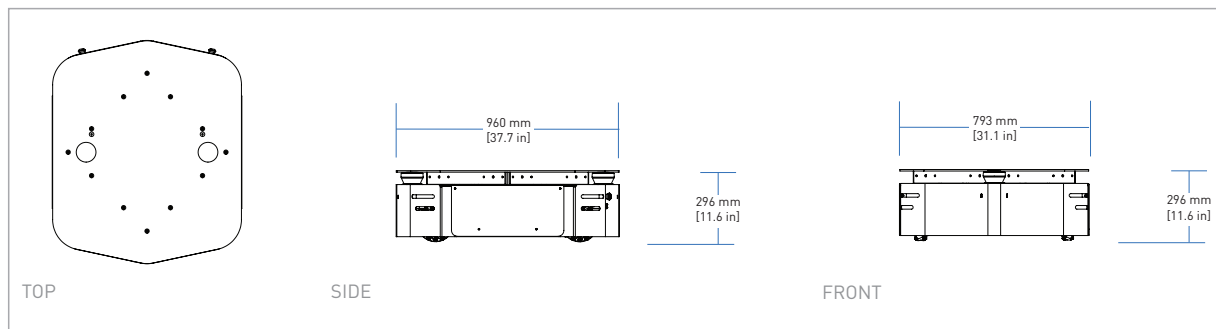
—
We reserve the right to make technical changes or modify the contents of this document without prior notice. With regard to purchase orders, the agreed particulars shall prevail. ABB AG does not accept any responsibility whatsoever for potential errors or possible lack of information in this document.

We reserve all rights in this document and in the subject matter and illustrations contained therein. Any reproduction, disclosure to third parties or utilization of its contents – in whole or in parts – is forbidden without prior written consent of ABB AG. Copyright© 2017 ABB
All rights reserved

ROB0317EN 14.08.2017

RIDGEBACK™

Omnidirectional Development Platform



TECHNICAL SPECIFICATIONS

SIZE AND WEIGHT	
EXTERNAL DIMENSIONS	960 x 793 x 296 mm (L x W x H)
WEIGHT	135 kg
OBSTACLE CLEARANCE	18 mm
SPEED AND PERFORMANCE	
MAX PAYLOAD	100 kg
MAX SPEED	1.1 m/s
DRIVE CONFIGURATION	4 Independently driven omni-directional wheels
OPERATING ENVIRONMENT	Indoor
BATTERY AND POWER SYSTEM	
BATTERY CHEMISTRY	Marine Grade AGM Sealed Lead Acid
CAPACITY	24 V 100 Ah
OPERATING TIME	15 hours with max payload
CHARGE TIME	8 Hours approx
USER POWER	5 V, 12 V, 24 VDC (fused at 10A each), optional 120 VAC
POWER	800 W Typical Use, Shore Power Available
INTERFACING AND COMMUNICATION	
CONTROL MODES	Kinematic control (forward, sideways, rotation), individual wheel velocities
FEEDBACK	Encoders, onboard IMU
COMMUNICATION	Ethernet, USB 3.0, RS 232
INCLUDED ACCESSORIES	Onboard computer, laser, gyroscope
DRIVERS AND APIs	ROS Indigo, Gazebo, navigation support, MoveIt!

Contact us today for pricing and a free 30 minute technical assessment: 1-800-301-3863

Date: 2016.06.20

Scanning Laser Range Finder Smart-URG mini UST-10LX Specification

CE
RoHS

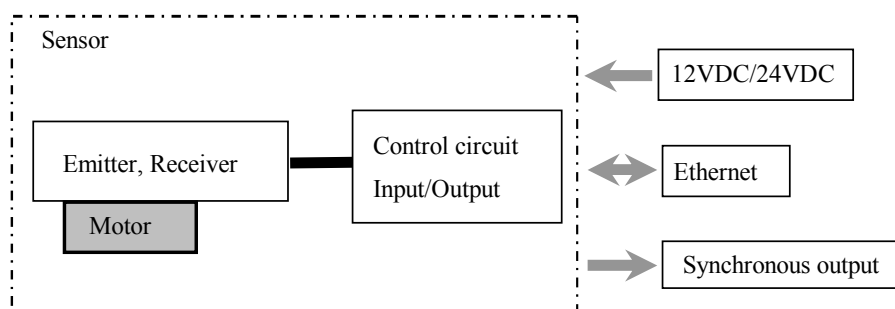
$\Delta \times 3$	Improved document quality.			3,4,6	2016.06.20	<i>Y.Kamioka</i>	RS-00629
$\Delta \times 1$	Product code no deleted.			1	2015.03.02	<i>T.Kasahara</i>	RS-00554
Symbol	Amended Reason			Pages	Date	Amended by	Ref.No
Approved by	Checked by	Drawn by	Designed by	Title	UST-10LX Specification		
<i>T.Kamitani</i>	<i>M.Maeda</i>	<i>Y.Kamioka</i>	<i>A.Yamamoto</i>	Drawing No.	C-42-04077		1/6

1. General

This sensor uses a laser source to scan 270° field of view. Positions of objects in the range are calculated with step angle and distance. Sensor outputs these data through communication channel.

2. Structure

2-1. Structure diagram

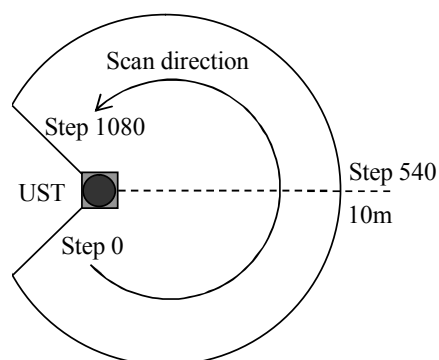


2-2. Laser scanning image

Measurement steps 1081

Detection angle 270°

Angular resolution 0.25°



3. Important notes

- (1) This sensor is not a safety device/tool.
- (2) This sensor cannot be used for human body detection as per the machinery directives.
- (3) Hokuyo products are not developed and manufactured for the use in weapons, equipments or related technologies intended for destroying human lives or causing mass destruction. If such possibilities or usages are revealed, the sales of Hokuyo products to those customers might be halted by the laws of Japan such as Foreign Exchange Law, Foreign Trade Law or Export Trade control order. In addition, Hokuyo products are for the purpose of maintaining the global peace and security in accordance with the above law of Japan.

Title	UST-10LX Specification	Drawing No	C-42-04077	2/6
-------	------------------------	------------	------------	-----

4. Specifications

Product name	Scanning Laser Range Finder
Model	UST-10LX
Supply voltage	12VDC/24VDC (Operation range 10 to 30V ripple within 10%)
Supply current \hat{I}	150mA or less (When using DC24V) (during start up 450mA is necessary.)
Light source	Laser semiconductor (905nm) Laser class 1 (IEC60825-1:2007)
Detection range	0.06m to 10m (white Kent sheet) 0.06m to 4m (diffuse reflectance 10%) Max. detection distance : 30m
Accuracy	$\pm 40\text{mm}$ (*1)
Repeated accuracy	$\sigma < 30\text{mm}$ (*1)
Scan angle	270°
Scan speed	25ms (Motor speed 2400rpm)
Angular resolution	0.25°
Start up time	Within 10 sec (start up time differs if malfunction is detected during start up)
Input	IP reset input, photo-coupler input (current 4mA at ON)
Output	Synchronous Output, photo coupler open collector output 30VDC 50mA MAX.
Interface	Ethernet 100BASE-TX
LED display	Power supply LED display (Blue): Blinks during start up and malfunction state.
Surrounding intensity	Less than 15,000lx Note : Avoid direct sunlight or other illumination sources as it may cause sensor malfunction
Ambient temperature and humidity	-10°C to +50°C, below 85%RH (without dew, frost)
Storage temperature and humidity	-30°C to +70°C, below 85%RH (without dew, frost)
Vibration resistance	10 to 55Hz double amplitude of 1.5mm for 2hrs in each X, Y, and Z direction 55 to 200Hz 98m/s^2 sweep of 2min for 1hr in each X,Y and Z direction
Vibration resistance (Operating)	55 to 150Hz 19.6m/s^2 sweep of 2min for 30min in each X,Y and Z direction
Shock resistance	196m/s^2 (20G) X,Y and Z direction 10 times.
EMC standards	(EMI) EN61326-1:2013 EN55011:2009 + A1:2010 (EMS) EN61326-1:2013 EN61000-4-2:2009 EN61000-4-3:2006 + A1:2008 + A2:2010 EN61000-4-4:2012 EN61000-4-6:2009 EN61000-4-8:2010
Protective Structure	IP65
Weight	130g (Excluding cable)
Material	Front case: Polycarbonate, Rear case: Aluminum
Dimensions (W×D×H)	50×50×70mm (sensor only)

(*1) Under the factory standard testing conditions using white Kent sheet.

Title	UST-10LX Specification	Drawing No	C-42-04077	3/6
-------	------------------------	------------	------------	-----

5. Measurement Data

Distance Value (x)	Meaning
$x < 21$	Output numerical number "4" as Measurement error
$21 \leq x \leq 30000$	Valid distance [mm]
$x > 30000$	Output numerical number "65533" as Measurement error (object does not exist or object has low reflectivity)

6. Connection

6-1. Power source, I/O cable

Cable length: 1000mm Flying lead cable (AWG28)

Color	Signal
Red	COM Input +
Gray	COM Output -
Light Blue	IP Reset Input
Orange	Synchronous Output
Brown	+VIN (12VDC/24VDC)
Blue	-VIN

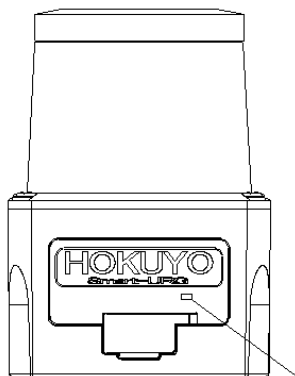
Note: Direction of Inputs and Outputs are mentioned from the sensor's side.

6-2. Ethernet cable

Cable length: 300mm

Color	Signal
Blue	TX+
White	TX-
Orange	RX+
Yellow	RX-

7. LED display



Power supply display
(Blinks during start up and malfunction state)

Title

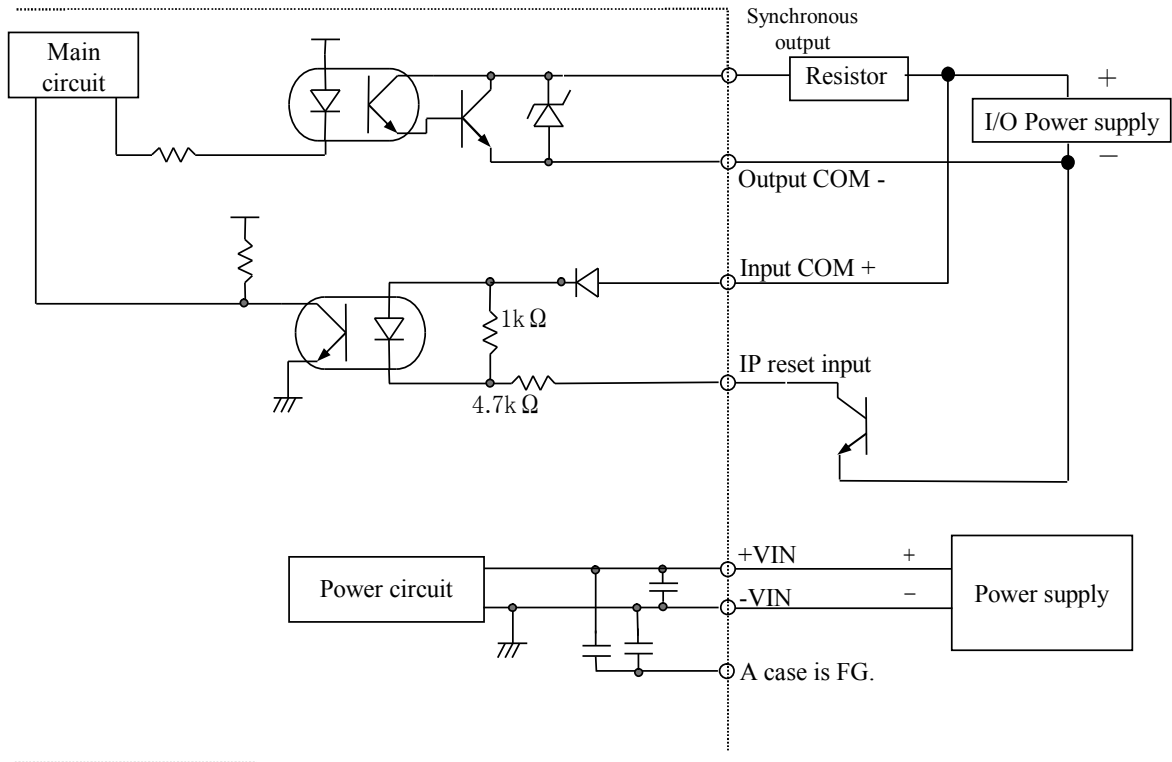
UST-10LX Specification

Drawing No

C-42-04077

4/6

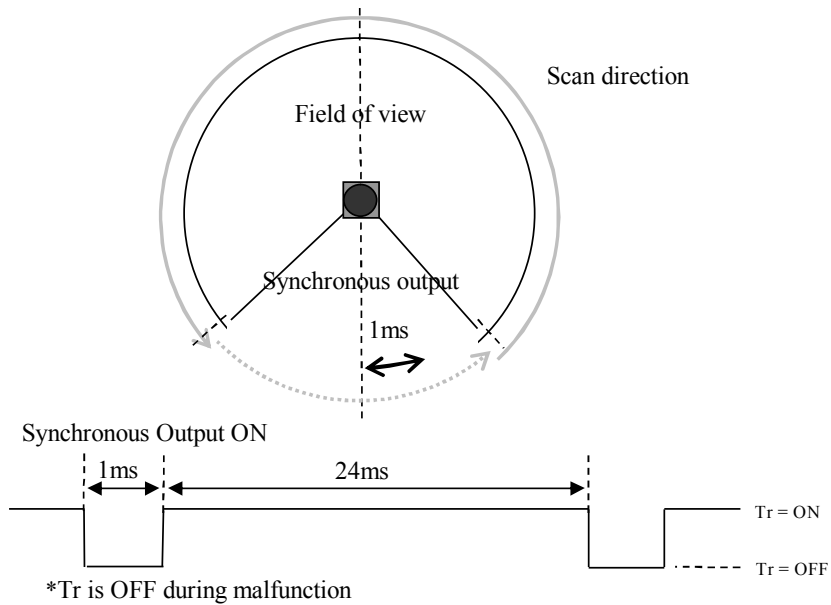
8. Output circuit



9. Control signal

9-1. Synchronous output

1 pulse is approximately 1ms. Output signal synchronization timing chart is shown as below.




Title

UST-10LX Specification

Drawing No

C-42-04077

5 / 6

10. Ethernet Setting 

1. The setting value is as below.

IP Initial value :192.168.0.10

Port number :10940

2. About Initialization of IP address

To reset IP address to the factory default value, please provide power to input circuit (see Section 8) and connect IP RESET LINE to COM- for more than 2 sec.

After IP RESET LINE is disconnected from COM- or opened, the sensor LED blinks and the sensor start to reboot.

11. Cautions for operation

This sensor uses high speed processing components that generate heat during operation.

The heat is concentrated at the bottom of the unit. When mounting, please attach the bottom of the unit to a good heat sink. A 200mm x 200mm x 2mm aluminum plate is recommended as a heat sink.

If multiple sensors are installed side by side, a sensor might mistake the laser pulses of other units as its own and the detection error occurs. When it happens, usually the error lasts for one or two steps of measurement. Please use software filters to handle this type of error.

Title	UST-10LX Specification	Drawing No	C-42-04077	6/6
-------	------------------------	------------	------------	-----

Add-In source codes

D.1 C# code of the Main Class

```

using System;
using System.Windows.Forms;
using System.Drawing;

5 using ABB.Robotics.Math;
using ABB.Robotics.RobotStudio;
using ABB.Robotics.RobotStudio.Environment;
using ABB.Robotics.RobotStudio.Stations;
using ABB.Robotics.RobotStudio.Stations.Forms;

10 // To activate this Add-In you have to copy Addin_test2.rsaddin to the Add-In directory
//
// typically C:\Program Files (x86)\Common Files\ABB Industrial IT\Robotics IT\
// RobotStudio\AddIns

namespace Addin_test2
15 {
    public class Main
    {
        // ***** PUBLIC VARIABLES *****
        // Default settings for the prompt
        public static PromptAnswer settings = new PromptAnswer("10.10.15.110", "8001", "
20 10.10.15.139", "8002", false, false);

        // ***** PRIVATE VARIABLES *****
        // Create a Camera
        private static Camera camera = new Camera();
        // Station
        private static Station station;
        // Time
        private static DateTime last_position_time;
        // Robot platform
        private static SmartComponent robot;
        // Sensor
        private static SmartComponent positionSensor;

        // This is the entry point which will be called when the Add-in is loaded
        public static void AddinMain()
        {
            // Handle events from controls defined in CommandBarControls.xml
            RegisterCommand("Addin_test2.StartButton");
            RegisterCommand("Addin_test2.Settings");
            RegisterCommand("Addin_test2.CloseButton");

            station = Project.ActiveProject as Station;
            // If there is no active project...
            if (station == null)
            {
                // ...create a new one.
                station = new Station();
                station.ActiveTask = station.DefaultTask;
                Project.ActiveProject = station;
            }

            // time initialisation
            last_position_time = DateTime.Now;
            // robot assignment
            robot = station.GraphicComponents["Ridgeback"] as SmartComponent;
            // sensor assignment
            positionSensor = robot.GraphicComponents["PositionSensor"] as SmartComponent
;

            // for each simulation tick check for another command
            Simulator.Tick += new EventHandler(tickEvent);
60 }

```

```

static void tickEvent(object sender, EventArgs e)
{
65     station = Project.ActiveProject as Station;

    // ***** TCP *****
    if (TCPCom.initialized)
    {
70         // ask the TCP for a velocity command
        CmdVel cmdVel = TCPCom.TCPGetCmdVel();

        // Print the command in the RobotStudio Logger
        if (settings.use_debug_printouts)
75         {
            Logger.AddMessage(new LogMessage("TCP Communication receives this
command input: [" + cmdVel.LinearX.ToString() + ", " + cmdVel.LinearY.ToString() + "
, " + cmdVel.Angular.ToString() + "]"));
        }

        // updates the smart component with the received command
        robot.IOSignals["SpeedX"].Value = cmdVel.LinearX;
        robot.IOSignals["SpeedY"].Value = cmdVel.LinearY;
        // the angular velocity is sent in radiants/s but in RobotStudio is in
degrees/s
        robot.IOSignals["Angular"].Value = cmdVel.Angular * 180 / Math.PI;
    }
85

    // ***** UDP *****
    if ((DateTime.Now - last_position_time).TotalSeconds > 0.02)
    {
        // Debug.WriteLine(positionSensor.Properties["Position"].GetType().
ToString());
        Vector3 position = Vector3.Parse(positionSensor.Properties["Position"].
Value.ToString());
        // Debug.WriteLine(positionSensor.Properties["Orientation"].GetType().
ToString());
        Vector3 orientation = Vector3.Parse(positionSensor.Properties["
Orientation"].Value.ToString());
        // state vector to send via UDP (the orientation is sent in deg/s)
        RobotPose state = new RobotPose
95         {
            X = position.x,
            Y = position.y,
            Theta = orientation.z * Math.PI / 180
        };
100

        // send the state vector via the UDP connection
        UDPCom.UDPsendPose(state);
        // time update
        last_position_time = DateTime.Now;
        // Print the command in the RobotStudio Logger
        if (settings.use_debug_printouts)
105         {
            Logger.AddMessage(new LogMessage("UDP Communication sends this robot
pose: [" + state.X.ToString() + ", " + state.Y.ToString() + ", " + state.Theta.
ToString() + "]"));
        }
110     }
}

static void RegisterCommand(string id)
{
115     var button = CommandBarButton.FromID(id);
    button.UpdateCommandUI += button_UpdateCommandUI;
    button.ExecuteCommand += button_ExecuteCommand;
}

static void button_UpdateCommandUI(object sender, UpdateCommandUIEventArgs e)
{
120     switch (e.Id)
    {
        case "Addin_test2.StartButton":
        case "Addin_test2.Settings":
        case "Addin_test2.CloseButton":
125             e.Enabled = true;
            break;
    }
}

```

```
130     }
    static void button_ExecuteCommand(object sender, ExecuteCommandEventArgs e)
    {
135         switch (e.Id)
        {
            case "Addin_test2.StartButton":
                {
                    // Initialise the TCP communication
                    TCPCom.InitTCP();
140                    // Initialise the UDP communication
                    UDPCom.InitUDP();

                    // Show and activate the add-in tab
                    RibbonTab ribbonTab = UIEnvironment.RibbonTabs["Simulation"];
145                    ribbonTab.Visible = true;
                    UIEnvironment.ActiveRibbonTab = ribbonTab;
                }
                break;
            case "Addin_test2.Settings":
                {
                    PromptForInput();
                    // Set the Camera properties
                    SetCamera(station.GraphicComponents["Ridgeback"]);
155                }
                break;
            case "Addin_test2.CloseButton":
                {
                    // Close the communications
                    TCPCom.CloseTCP();
160                    Logger.AddMessage(new LogMessage("TCP Communication closed.));
                    //UDPCom.CloseUDP();
                    Logger.AddMessage(new LogMessage("UDP Communication closed.));
                    // return to the Home tab
                    RibbonTab ribbonTab = UIEnvironment.RibbonTabs["Home"];
165                    ribbonTab.Visible = true;
                    UIEnvironment.ActiveRibbonTab = ribbonTab;
                }
                break;
        }
    }

170     static void SetCamera(GraphicComponent component)
    {
175         try
        {
            station = Station.ActiveStation;

            // Set the camera to view from a specific point
            camera.LookFrom = new Vector3(0, -15, 5);
180

            // Add it to the stations camera collection
            station.Cameras.Add(camera);

            // Sync the current view to the camera view
            GraphicControl.ActiveGraphicControl.SyncCamera(camera, true, 0);
185

            // Use the camera as the true camera
            // This make sure that the constraints on the movement kicks in
            GraphicControl.ActiveGraphicControl.Camera = camera;

            // Keep the direction in which the view of the box fixed
            camera.FollowBehavior = FollowObjectBehavior.FixDirection;

            // Follow the object
            if(settings.fixed_camera == false)
195             {
                camera.FollowObject = component;
            }
            else
            {
200                // Fix the camera to this point
                camera.LockRotate = 0.0;
                camera.LockTranslate = 0.0;
                camera.LockZoom = 0.0;
            }
205        }
    }
}
```

```

        catch
        {
            Project.UndoContext.CancelUndoStep(CancelUndoStepType.Rollback);
            throw;
        }
    }

    //-----
    // User Input Related
    //-----
    public class PromptAnswer
    {
        public TCPCom.UserInput tcp;
        public UDPCom.UserInput udp;
        public bool fixed_camera;
        public bool use_debug_printouts;

        // Empty constructor
        public PromptAnswer()
        {
            tcp = new TCPCom.UserInput();
            udp = new UDPCom.UserInput();
            use_debug_printouts = false;
            fixed_camera = false;
        }

        // Constructor
        public PromptAnswer(string tcp_server, string tcp_port, string udp_server,
            string udp_port, bool fixed_camera, bool use_debug_printouts)
        {
            tcp = new TCPCom.UserInput(tcp_server, tcp_port);
            udp = new UDPCom.UserInput(udp_server, udp_port);
            this.fixed_camera = fixed_camera;
            this.use_debug_printouts = use_debug_printouts;
        }

        public override string ToString()
        {
            return tcp.ToString() + ", " + udp.ToString() + ", Camera: " +
                fixed_camera + ", Debug: " + use_debug_printouts;
        }

        public bool Validate()
        {
            return tcp.Validate() && udp.Validate();
        }
    }

    public static class Prompt
    {
        public static PromptAnswer ShowDialog(string caption)
        {
            Form prompt = new Form()
            {
                Width = 300,
                Height = 300,
                FormBorderStyle = FormBorderStyle.FixedDialog,
                Text = caption,
                StartPosition = FormStartPosition.CenterScreen,
                AutoSize = true,
                MaximizeBox = false,
                MinimizeBox = false
            };

            int x = 0;
            int y = 0;
            int y_offset = 0;
            int header_offset = 17;

            //-----
            // TCP Communication
            //-----
            x = 5;
            y = 3;
            Label tcp_header_label = new Label() { AutoSize = true, Text = "TCP
Communication" };

```

```

tcp_header_label.Font = new Font(SystemFonts.DefaultFont.FontFamily,
SystemFonts.DefaultFont.Size, FontStyle.Bold);
tcp_header_label.Location = new Point(x, y);

285     x = 5;
        y = 7;
        y_offset = 25;
        Label tcp_server_label = new Label() { AutoSize = true, Text = "Server
IP:" };
        Label tcp_port_label = new Label() { AutoSize = true, Text = "Port
Number:" };
tcp_server_label.Location = new Point(x, y + 0 * y_offset +
header_offset);
290     tcp_port_label.Location = new Point(x, y + 2 * y_offset + header_offset)
;

        x = 5;
        y = 5;
        TextBox tcp_server_textbox = new TextBox() { Width = 120, Text =
settings.tcp.server };
295     TextBox tcp_port_textbox = new TextBox() { Width = 120, Text = settings.
tcp.port };
        tcp_server_textbox.Location = new Point(x, y + 1 * y_offset +
header_offset);
        tcp_port_textbox.Location = new Point(x, y + 3 * y_offset +
header_offset);

300     x = 5;
        y = 130;
        Label tcp_separator_1 = new Label() { AutoSize = false, Height = 2,
Width = 170, BorderStyle = BorderStyle.Fixed3D };
        tcp_separator_1.Location = new Point(x, y);

305     prompt.Controls.Add(tcp_header_label);
        prompt.Controls.Add(tcp_server_label);
        prompt.Controls.Add(tcp_port_label);
        prompt.Controls.Add(tcp_server_textbox);
        prompt.Controls.Add(tcp_port_textbox);
        prompt.Controls.Add(tcp_separator_1);

310     //-----
        // UDP Communication
        //-----

315     int y_udp_offset = 130;
        x = 5;
        y = 3 + y_udp_offset;
        Label udp_header_label = new Label() { AutoSize = true, Text = "UDP
Communication" };
        udp_header_label.Font = new Font(SystemFonts.DefaultFont.FontFamily,
SystemFonts.DefaultFont.Size, FontStyle.Bold);
320     udp_header_label.Location = new Point(x, y);

        y = 7 + y_udp_offset;
        y_offset = 25;
        Label udp_server_label = new Label() { AutoSize = true, Text = "Server
IP:" };
        Label udp_port_label = new Label() { AutoSize = true, Text = "Port
Number:" };
325     udp_server_label.Location = new Point(x, y + 0 * y_offset +
header_offset);
        udp_port_label.Location = new Point(x, y + 2 * y_offset + header_offset)
;

        x = 5;
        y = 5 + y_udp_offset;
330     TextBox udp_server_textbox = new TextBox() { Width = 120, Text =
settings.udp.server };
        TextBox udp_port_textbox = new TextBox() { Width = 120, Text = settings.
udp.port };
        udp_server_textbox.Location = new Point(x, y + 1 * y_offset +
header_offset);
        udp_port_textbox.Location = new Point(x, y + 3 * y_offset +
header_offset);

335     prompt.Controls.Add(udp_header_label);
        prompt.Controls.Add(udp_server_label);
        prompt.Controls.Add(udp_port_label);

```

```

prompt.Controls.Add(udp_server_textbox);
prompt.Controls.Add(udp_port_textbox);
340
//-----
// Extra: IMG / Camera Settings / Debug Messages
//-----
x = 180;
345 y = 5;
Label separator = new Label() { AutoSize = false, Height = 250, Width =
3, BorderStyle = BorderStyle.Fixed3D };
separator.Location = new Point(x, y);

x = 190;
350 y = 5 + y_udp_offset + 15;
Button confirmation = new Button() { Text = "Ok", AutoSize = true,
DialogResult = DialogResult.OK };
confirmation.Location = new Point(x + 45, y + 3 * y_offset - 1);
confirmation.Click += (sender, e) => { prompt.Close(); };

355 y = 7 + y_udp_offset + 5;
CheckBox use_debug = new CheckBox() { AutoSize = true, Text = "Debug
prints", Checked = settings.use_debug_printouts };
use_debug.Location = new Point(x, y + 2 * y_offset);

y = 7 + y_udp_offset;
360 CheckBox camera = new CheckBox() { AutoSize = true, Text = "Camera in 0"
, Checked = settings.fixed_camera };
camera.Location = new Point(x, y + 1 * y_offset);

x = 190;
y = 10;
365 PictureBox picture = new PictureBox()
{
    Image = Image.FromFile(@"C:\Users\xmaiov\source\repos\Addin_test2\
Addin_test2\antenna3.PNG"),
    SizeMode = PictureBoxSizeMode.StretchImage,
    Height = 120,
370 Width = 120
};
picture.Location = new Point(x, y);

prompt.Controls.Add(separator);
375 prompt.Controls.Add(confirmation);
prompt.Controls.Add(use_debug);
prompt.Controls.Add(camera);
prompt.Controls.Add(picture);
prompt.AcceptButton = confirmation;
380

//-----
// Collect input
//-----

385 // In here it collects the inputs written in the prompt by the user
return prompt.ShowDialog() == DialogResult.OK ? new PromptAnswer(
tcp_server_textbox.Text, tcp_port_textbox.Text, udp_server_textbox.Text,
udp_port_textbox.Text, camera.Checked, use_debug.Checked):new PromptAnswer();
}
}

390 private static void PromptForInput()
{
    PromptAnswer temp = Prompt.ShowDialog("TCP/UDP Comm. Settings");

    395 if (temp.Validate())
    {
        settings = temp;
        Logger.AddMessage(new LogMessage("User input: " + settings.ToString()));
    }
}

400 }
}

```

D.2 C# code of the TCP Server

The main idea of this code has been found in [24].

The class documentation is available in [23].

```

// TCP communication
/*  Server Program  */

using System;
using System.Diagnostics;
5 using System.Net;
using System.Net.Sockets;

10 using ABB.Robotics.RobotStudio;

public class TCPCom
{
    // ***** PRIVATE VARIABLES *****
15 private static Socket s;
private static TcpListener listener;
private static CmdVel cmdVel;
private static CmdVel cmd0;
private static DateTime last_update_time;
20 private static byte[] byte_msg = new byte[400];

    // ***** PUBLIC VARIABLES *****
public static bool initialized = false;

25 private static void TCPStartCallback(IAsyncResult ar)
{
    s = listener.EndAcceptSocket(ar);
    Debug.WriteLine("TCP Connection accepted from " + s.RemoteEndPoint);
    // Print a message in the RobotStudio Logger
30 Logger.AddMessage(new LogMessage("TCP Communication initialised: waiting for the
command inputs.));
    initialized = true;
}

public static bool InitTCP()
35 {
    cmd0 = new CmdVel
    {
        LinearX = 0.0,
        LinearY = 0.0,
40 Angular = 0.0
    };
    cmdVel = cmd0;

    last_update_time = DateTime.Now;

45 try
    {
        IPAddress ipAd = IPAddress.Parse(Addin_test2.Main.settings.tcp.server);
        // IPAddress ipAd = IPAddress.Parse("10.10.15.110");
        // the local IP address can by found via the console --> ipconfig
        // use local m/c IP address, and use the same in the client

        /* Initializes the Listener */
50 listener = new TcpListener(ipAd, Int32.Parse(Addin_test2.Main.settings.tcp.
port));
        // listener = new TcpListener(ipAd, 8001);

        /* Start Listeneting at the specified port */
        listener.Start();

60 Debug.WriteLine("The TCP server " + Addin_test2.Main.settings.tcp.server + "
is running at port " + Addin_test2.Main.settings.tcp.port);
        Debug.WriteLine("Waiting for TCP Client.");

        // accepts a pending connection request
        object state = new object();

65 // s = listener.AcceptSocket();
        listener.BeginAcceptSocket(TCPStartCallback, state);
    }
}

```

```

    return true;
70 }
    catch (Exception e)
    {
        Debug.WriteLine("TCP Error: " + e.StackTrace);
        return false;
75 }
}

public static CmdVel TCPGetCmdVel()
80 {
    try
    {
        while (s.Available != 0)
        {
            // receive data from a bound socket into a receive buffer
85 int k = s.Receive(byte_msg);
            //Debug.WriteLine("Recieved: ");
            //for (int i = 0; i < byte_msg.Length; i++)
            //    Debug.WriteLine(byte_msg[i]);

            // create a new CmdVel object and parse the input data
90 cmdVel = CmdVel.Parser.ParseFrom(byte_msg, 0, k);

            last_update_time = DateTime.Now;
        }
95 if ((DateTime.Now - last_update_time).TotalSeconds > 0.25)
        {
            if (Addin_test2.Main.settings.use_debug_printouts)
            {
                Logger.AddMessage(new LogMessage("TCP Command timeout reached, set
100 to zero."));
            }
            cmdVel = cmd0;
        }
    }
    catch (Exception e)
105 {
        Debug.WriteLine("TCP Error: " + e.StackTrace);
    }
    return cmdVel;
}

110 public static void CloseTCP()
    {
        /* clean up */
        s.Close();
115 listener.Stop();
        initialized = false;
        Debug.WriteLine("TCP Communication Interrupted.");
    }

120 //-----
// TCP USER INPUT
//-----
public class UserInput
125 {
    public string server; // I.e. IP address or hostname.
    public string port;

    public UserInput()
    {
130         server = "";
        port = "";
    }
    public UserInput(string server, string port)
    {
135         this.server = server;
        this.port = port;
    }

    public bool Validate()
140 {
        uint temp;
        bool ok = !(server.Equals("") || port.Equals(""));
        if (ok)
        {

```



```

145         ok = uint.TryParse(port, out temp);
        }
        return ok;
    }

150    override public string ToString()
    {
        return "TCP Communication [" + server + ", " + port + "]";
    }

155    public string GetURL()
    {
        return (port.Equals("80") ? ("http://" + server + "/" ) : ("http://" + server
+ ":" + port + "/" ));
    }
}
160 }

```

D.3 C# code of the UDP Client

The class documentation is available in [26].

```

// UDP communication
/* Client Program */

using System;
5 using System.Diagnostics;
using System.Net;
using System.Net.Sockets;
using Google.Protobuf;

10 public class UDPCom
{
    // ***** PRIVATE VARIABLES *****
    private static UdpClient udpClient;

15    public static bool InitUDP ()
    {
        // This constructor arbitrarily assigns the local port number.
        udpClient = new UdpClient ();

20        try
        {
            String ipAd = Addin_test2.Main.settings.udp.server;
            IPEndPoint endPoint = new IPEndPoint (IPAddress.Parse(ipAd), Int32.Parse(
Addin_test2.Main.settings.udp.port));
25            // IPEndPoint endPoint = new IPEndPoint (IPAddress.Parse("10.10.15.139"),
8002);
            udpClient.Connect (endPoint);
            Debug.WriteLine("The UDP server " + Addin_test2.Main.settings.udp.server + "
is running at port " + Addin_test2.Main.settings.udp.port);
            Debug.WriteLine("Waiting for UDP Server.");
            return true;

30        }

        catch(Exception e)
        {
            Debug.WriteLine("UDP Error: " + e.ToString());
            return false;

35        }
    }

    public static void UDPsendPose(RobotPose pose)
40    {
        // Sends a message to the host to which you have connected.
        byte[] msg = new byte[pose.CalculateSize()];
        CodedOutputStream output = new CodedOutputStream (msg);
        pose.WriteTo(output);
45        //for (int i = 0; i < msg.Length; i++)
        //    Debug.WriteLine(msg[i]);

        udpClient.Send(msg, msg.Length);
        // Debug.WriteLine("UDP Message sent.");
    }
}

```

```

50     }

    public static void CloseUDP ()
    {
        /* clean up */
55     udpClient.Close();
        Debug.WriteLine("UDP Communication Interrupted.");
    }

    //-----
60     // UDP USER INPUT
    //-----
    public class UserInput
    {
        public string server; // I.e. IP address or hostname.
65     public string port;

        public UserInput ()
        {
70         server = "";
            port = "";
        }
        public UserInput(string server, string port)
        {
75         this.server = server;
            this.port = port;
        }

        public bool Validate()
        {
80         uint temp;
            bool ok = !(server.Equals("") || port.Equals(""));
            if (ok)
            {
85                 ok = uint.TryParse(port, out temp);
            }
            return ok;
        }

        override public string ToString()
90     {
            return "UDP Communication [" + server + ", " + port + "];"
        }

        public string GetURL()
95     {
            return (port.Equals("80") ? ("http://" + server + "/" ) : ("http://" + server
+ ":" + port + "/" ));
        }
    }
}

```

D.4 XML code of the Ribbon

```

<?xml version="1.0" encoding="utf-8" ?>
<Ribbon xmlns="urn:abb-robotics-robotstudio-ribbon"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:abb-robotics-ribbon file:///C:\Program%20Files%20(x86)\
5     ABB%20Industrial%20IT\Robotics%20IT\SDK\RobotStudio%20SDK%206.06\Ribbon.xsd">

    <Tabs>
        <!-- Add a button to an existing ribbon tab -->
        <Tab id="Home">
            <!-- It is possible to add a button to an existing group, but note that standard
10         groups may change -->
            <!-- In this sample we create a new group -->
            <Group id="Addin_test2" caption="TCP / UDP">
                <Control id="Addin_test2.StartButton" layout="Large"/>
                <Separator/>
                <Control id="Addin_test2.Settings" layout="Large"/>
15         </Group>
        </Tab>
        <Tab id="Simulation">

```

```

    <!-- It is possible to add a button to an existing group, but note that standard
    groups may change -->
    <!-- In this sample we create a new group -->
20 <Group id="Addin_test2" caption="TCP / UDP">
      <Control id="Addin_test2.CloseButton" layout="Large"/>
    </Group>
  </Tab>
</Tabs>
25 </Ribbon>

```

D.5 XML code of the Command Bar Control

```

<?xml version="1.0" encoding="utf-8" ?>
<CommandBarControls xmlns="urn:abb-robotics-robotstudio-commandbarcontrols"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xsi:schemaLocation="urn:abb-robotics-robotstudio-commandbarcontrols
5   file:///C:\Program%20Files%20(x86)\ABB%20Industrial%20IT\Robotics%20IT\SDK\
    RobotStudio%20SDK%206.06\CommandBarControls.xsd">

  <!-- Controls defined by this add-in -->

  <!-- The Start button has default_Enabled set to true which means it can always be
  clicked. -->
  <!-- When it is clicked the add-in will be loaded and the ExecuteCommand handler will
  be called. -->
10 <CommandBarButton id="Addin_test2.StartButton" caption=" Enable " defaultEnabled="true"
  "/>
  <CommandBarButton id="Addin_test2.Settings" caption=" Settings " defaultEnabled="true"
  />
  <CommandBarButton id="Addin_test2.CloseButton" caption=" Close "/>
</CommandBarControls>

```

D.6 XML code of the Command Help Text

```

<?xml version="1.0" encoding="utf-8"?>
<Controls xmlns="urn:abb-robotics-robotstudio-controlhelptexts"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="urn:abb-robotics-robotstudio-controlhelptexts file:///C:\
5   Program%20Files%20(x86)\ABB%20Industrial%20IT\Robotics%20IT\SDK\RobotStudio%20SDK
    %206.06\ControlHelpTexts.xsd">

  <!-- Help texts for controls specified in Command_Bar_Controls -->

  <Control id="Addin_test2.StartButton">
    To Enable the Communication:
    START the simulation and wait for command inputs.
10 </Control>
  <Control id="Addin_test2.Settings">To change the Communication settings:
    1) Set IP and PORT for the TCP/UDP Communication.
    2) Decide whether the Camera is attached to the Robot or to the Origin.
15 </Control>
    3) Decide whether display the Debug Messages or not.
  <Control id="Addin_test2.CloseButton">
    To Close the Communication
    STOP the simulation before closing the communication
20 </Control>
</Controls>

```


Basic Filters

This appendix contains the pseudo-code of three main filters, namely Bayes, Extended Kalman and Particle filters. All those algorithms are taken from [34].

E.1 Bayes Filter

Before presenting the Bayes filter, one more probability concept is needed. The notion of *belief* has been already introduced in Section 4.1 as an abbreviation for the posterior

$$bel(x_t) = p(x_t | z_{1:t}, u_{1:t}). \quad (\text{E.1.1})$$

In such a case the belief is computed just after taking into account the measurement z_t . Sometimes one may need to calculate the posterior before incorporating z_t .

Thus, such a posterior takes the name of *prediction* because it predicts the state at the time instant t based on the previous state posterior, and it is denoted as follows:

$$\overline{bel}(x_t) = p(x_t | z_{1:t-1}, u_{1:t}). \quad (\text{E.1.2})$$

Hence, the computation of $bel(x_t)$ from $\overline{bel}(x_t)$ is called *measurement update*.

Both beliefs are used in the general algorithm for Bayes filtering:

Algorithm E.1 Bayes Filter algorithm, pseudo code

```

1: function BAYES_FILTER( $bel(x_t), u_t, z_t$ )
2:
3:   for all  $x_t$  do
4:      $\overline{bel}(x_t) \leftarrow \int p(x_t | u_t, x_{t-1}) bel(x_{t-1}) dx_{t-1}$ 
5:      $bel(x_t) \leftarrow \eta p(z_t | x_t) \overline{bel}(x_t)$ 
6:
7:   return  $bel(x_t)$ 

```

Where in Line 5 a normalization factor η has been used because the product may not integrate to 1 and hence $bel(x_t)$ may not be a probability.

Moreover the belief is calculated recursively and hence an initial belief $bel(x_0)$ at time $t = 0$ is also needed as boundary condition.

The main issue of the Bayes filter is that the integral in Line 4 cannot be computed in closed form and hence it needs to be approximated.

On the base of the approximation done, the other filters (Kalman, Extended Kalman and Particle) are derived.

E.2 Gaussian Filters

Gaussian Filters are based on the hypothesis that beliefs are represented by multivariate normal distributions:

$$p(x) = \det(2\pi\Sigma)^{\frac{1}{2}} \exp\{-\frac{1}{2}(x - \mu)^T \Sigma^{-1} (x - \mu)\}, \quad (\text{E.2.1})$$

where μ and Σ are respectively the first (mean) and second (covariance) moments of the probability distribution.

E.2.1 The Kalman Filter

The Kalman filter was invented by Swerling (1958) and Kalman (1960) as a technique for filtering and prediction in linear Gaussian systems:

$$\begin{cases} x_t = A_t x_{t-1} + B_t u_t + \varepsilon_t \\ z_t = C_t x_t + \delta_t \end{cases} \quad (\text{E.2.2})$$

In the first equation, the state equation, A_t is a square matrix of dimension $n \times n$ where n is the size of the state vector x_t ; B_t is also a matrix but of dimension $n \times m$ where m is the size of control vector u_t . The term ε_t in the state equation is a Gaussian random vector which models the uncertainties in the state evolution. It is also called WGN, which stands for White Gaussian Noise, because its mean is zero, while its covariance is denoted by R_t . The state equation is derived from the state transition probability, which is Gaussian as well:

$$p(x_t | u_t, x_{t-1}) = \det(2\pi R_t)^{\frac{1}{2}} \exp\left\{-\frac{1}{2}(x_t - A_t x_{t-1} - B_t u_t)^T R_t^{-1} (x_t - A_t x_{t-1} - B_t u_t)\right\}. \quad (\text{E.2.3})$$

In the second equation, the measurement equation, C_t is a matrix of dimension $k \times n$ where k is the size of the measurement vector z_t . The term δ_t in the measurement equation is a Gaussian random vector which models the measurement noise. It is still a WGN but with covariance Q_t . The measurement equation is hence derived from the measurement probability:

$$p(z_t | x_t) = \det(2\pi Q_t)^{\frac{1}{2}} \exp\left\{-\frac{1}{2}(z_t - C_t x_t)^T Q_t^{-1} (z_t - C_t x_t)\right\}. \quad (\text{E.2.4})$$

Finally, also the initial belief must be a multivariate normal distribution:

$$\text{bel}(x_0) = p(x_0) = \det(2\pi \Sigma_0)^{\frac{1}{2}} \exp\left\{-\frac{1}{2}(x_0 - \mu_0)^T \Sigma_0^{-1} (x_0 - \mu_0)\right\}. \quad (\text{E.2.5})$$

Having both the state and measurement probability represented as linear functions in their arguments with added WGN, and also a normally distributed initial belief, ensure that the posterior $\text{bel}(x_t)$ is always Gaussian.

Thus, the Kalman Filter Algorithm can be written as follows:

Algorithm E.2 Kalman Filter algorithm, pseudo code

```

1: function KALMAN_FILTER( $\mu_{t-1}, \Sigma_{t-1}, u_t, z_t$ )
2:
3:    $\bar{\mu}_t \leftarrow A_t \mu_{t-1} + B_t u_t$ 
4:    $\bar{\Sigma}_t \leftarrow A_t \Sigma_{t-1} A_t^T + R_t$ 
5:
6:    $K_t \leftarrow \bar{\Sigma}_t C_t^T (C_t \bar{\Sigma}_t C_t^T + Q_t)^{-1}$ 
7:    $\mu_t \leftarrow \bar{\mu}_t + K_t (z_t - C_t \bar{\mu}_t)$ 
8:    $\Sigma_t \leftarrow (I - K_t C_t) \bar{\Sigma}_t$ 
9:
10:  return  $\mu_t, \Sigma_t$ 

```

In the pseudo code E.2, beliefs are represented by their first and second moments. The algorithm hence takes as input the belief at time $t - 1$, together with the control input and the measurement at time t . One can easily notice the likelihood to the Bayes filter reported in the pseudo code E.1 earlier above:

- in Lines 3 and 4 the belief $\bar{\text{bel}}(x_t)$ is obtained from the belief $\text{bel}(x_{t-1})$ by incorporating the control u_t ;

- in Lines 7 and 8 belief $\overline{bel}(x_t)$ is used to compute the belief $bel(x_t)$ by taking into account the measurement z_t .

The difference is made by the so called *Kalman gain* computed in Line 6. This gain states how the measurement will be incorporated in the new state estimate and indeed is used in Line 7 and 8 to manipulate the relation between the belief $\overline{bel}(x_t)$ and the measurement z_t .

The main success of this filter and the reasons of its employment in many applications, lies in its simplicity and its computational efficiency. The main drawback of this filter is the limitation to linear Gaussian systems.

E.2.2 The Extended Kalman Filter

The linearity assumption of the Kalman filter derives from the property of Gaussian distribution which states that a linear combination of Gaussian random variables is still a Gaussian random variable. However in practice, the equations describing the state evolution and the measurement update are rarely linear. By relaxing the linearity assumption, the Kalman filter can be modified in an extended version, which takes the name of *Extended Kalman Filter*.

The system of equation (E.2.2) can be generalized as follows:

$$\begin{cases} x_t = g(x_{t-1}, u_t) + \varepsilon_t \\ z_t = h(x_t) + \delta_t \end{cases} \quad (\text{E.2.6})$$

where g and h are two nonlinear functions.

The problem now is that the property of the linear combination of Gaussian variables is lost; hence, if for example X is a random variable, $Y = g(X)$ is not.

By consequence, what the EKF does is to compute an *approximation* to the true belief. The approximation done is nothing but a linearization, through a (first order) Taylor expansion, of the functions g (around μ_{t-1}) and h (around $\bar{\mu}_t$):

$$\begin{aligned} g(x_{t-1}, u_t) &\approx g(\mu_{t-1}, u_t) + G_t(x_{t-1} - \mu_{t-1}) \\ h(x_t) &\approx h(\bar{\mu}_t) + H_t(x_t - \bar{\mu}_t) \end{aligned} \quad (\text{E.2.7})$$

where:

$$\begin{aligned} G_t &= \left. \frac{\partial g(x_{t-1}, u_t)}{\partial x_{t-1}} \right|_{x_{t-1}=\mu_{t-1}} \\ H_t &= \left. \frac{\partial h(x_t)}{\partial x_t} \right|_{x_t=\bar{\mu}_t} \end{aligned} \quad (\text{E.2.8})$$

From that, the following approximated probabilities follow:

$$p(x_t | u_t, x_{t-1}) \approx \det(2\pi R_t)^{\frac{1}{2}} \exp\left\{-\frac{1}{2}[x_t - g(\mu_{t-1}, u_t) - G_t(x_{t-1} - \mu_{t-1})]^T R_t^{-1}[x_t - g(\mu_{t-1}, u_t) - G_t(x_{t-1} - \mu_{t-1})]\right\}, \quad (\text{E.2.9})$$

$$p(z_t | x_t) \approx \det(2\pi Q_t)^{\frac{1}{2}} \exp\left\{-\frac{1}{2}[z_t - h(\bar{\mu}_t) - H_t(x_t - \bar{\mu}_t)]^T Q_t^{-1}[z_t - h(\bar{\mu}_t) - H_t(x_t - \bar{\mu}_t)]\right\}. \quad (\text{E.2.10})$$

The EKF algorithm can be written as follows:

Algorithm E.3 Extended Kalman Filter algorithm, pseudo code

```

1: function EXTENDED_KALMAN_FILTER( $\mu_{t-1}, \Sigma_{t-1}, u_t, z_t$ )
2:
3:    $\bar{\mu}_t \leftarrow g(\mu_{t-1}, u_t)$ 
4:    $\bar{\Sigma}_t \leftarrow G_t \Sigma_{t-1} G_t^T + R_t$ 
5:
6:    $K_t \leftarrow \bar{\Sigma}_t H_t^T (H_t \bar{\Sigma}_t H_t^T + Q_t)^{-1}$ 
7:    $\mu_t \leftarrow \bar{\mu}_t + K_t (z_t - h(\bar{\mu}_t))$ 
8:    $\Sigma_t \leftarrow (I - K_t H_t) \bar{\Sigma}_t$ 
9:
10:  return  $\mu_t, \Sigma_t$ 

```

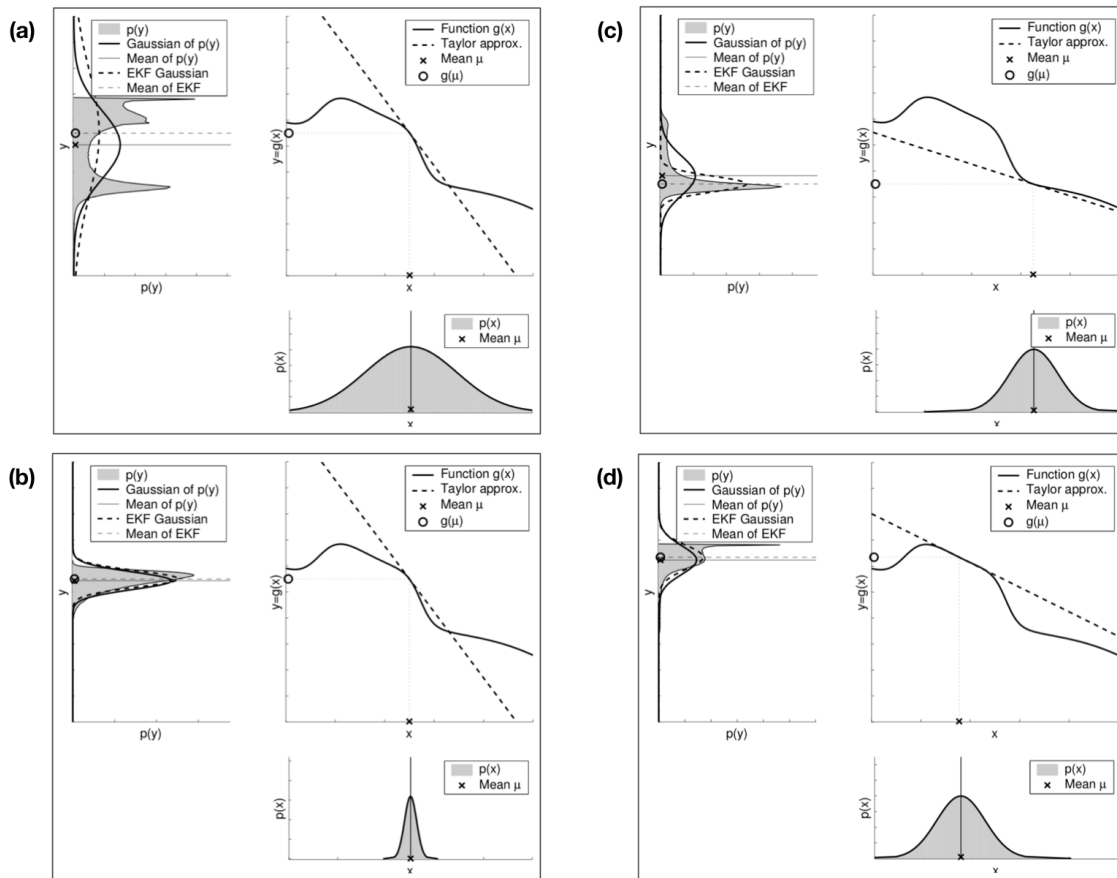


Figure E.1: Errors due to linearization (source: [34]).

The Extended Kalman Filter has the same strengths of its linear version, namely the simplicity and computational efficiency.

However its main limitation is in fact the linearization because its goodness depends on the degree of uncertainty of the Gaussian random variable (covariance, in Figure E.1 (a) and (b)) and on the degree of local nonlinearity of the functions to approximate (in Figure E.1 (c) and (d)):

- in Figure E.1 both Gaussians $p(x)$ in (a) and (b) have the same mean and are passed through the same nonlinear function g . Given that the Gaussian a) has an higher covariance, the resulting random variable $p(y)$ has a more distorted density (compare the Gaussian extracted from the density in solid line to the one obtained by the EKF in dashed line);

- in Figure E.1 (c) and (d) now both Gaussian have the same covariance but different mean, and are passed again through the same function. Thanks to the difference in the mean, the linearization is computed around two different points, which showily changes the Gaussian obtained via EKF (dashed lines).

E.3 The Particle Filter

The Particle Filter belongs to the family of nonparametric filters, where the posterior is not represented by a fixed functional form such as the Gaussian but instead it is approximated by a finite number of parameters.

The main idea of this filter is to represent the posterior $bel(x_t)$ by random state samples drawn from the posterior itself, called *particles* (Figure E.2). All the M drawn particles belong to the same set \mathcal{X}_t :

$$\mathcal{X}_t = x_t^{[1]}, x_t^{[2]}, \dots, x_t^{[M]}. \quad (\text{E.3.1})$$

Each particle $x_t^{[m]}$ ($1 \leq m \leq M$) is an instance of the state at time t , i.e. a guess at the value of the true state at time t .

Thus, each guess shall be proportional to $bel(x_t)$:

$$x_t^{[m]} \sim p(x_t | z_{1:t}, u_{1:t}). \quad (\text{E.3.2})$$

Hence, the subregion of the state space most populated by samples has the highest likelihood to be representative of the true state.

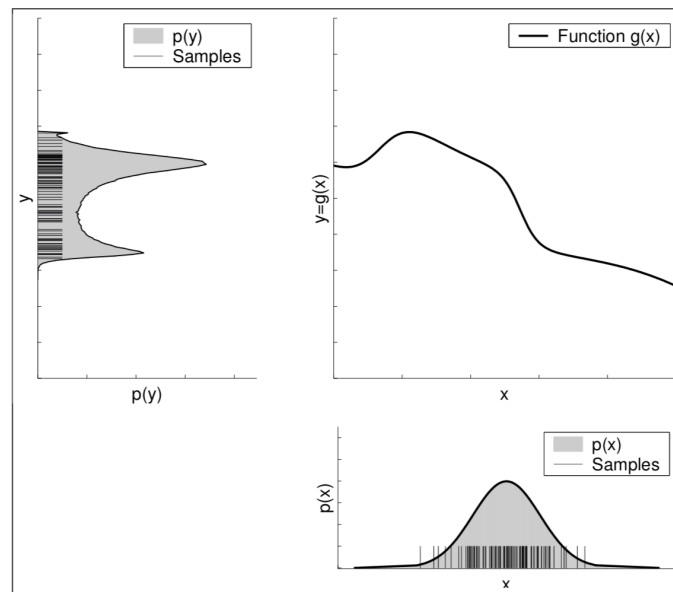


Figure E.2: Samples of a Gaussian through a nonlinear function (source: [34]).

Similarly to the other Bayes filters presented above, the belief at time t , represented by the particle set \mathcal{X}_t , is computed recursively from the one at time $t-1$, the set \mathcal{X}_{t-1} , which is the input of the algorithm below, together with the control u_t and the measurement z_t .

Algorithm E.4 Particle Filter algorithm, pseudo code

```

1: function PARTICLE_FILTER( $\mathcal{X}_{t-1}, u_t, z_t, m$ )
2:
3:    $\bar{\mathcal{X}}_t, \mathcal{X}_t \leftarrow \emptyset$ 
4:
5:   for  $m = 1$  to  $M$  do
6:     sample  $x_t^{[m]} \sim p(x_t|u_t, x_{t-1}^{[m]})$ 
7:      $w_t^{[m]} \leftarrow p(z_t|x_t^{[m]})$ 
8:      $\bar{\mathcal{X}}_t \leftarrow \bar{\mathcal{X}}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$ 
9:
10:  for  $m = 1$  to  $M$  do
11:    draw  $i$  with probability  $\propto w_t^{[i]}$ 
12:    add  $x_t^{[i]}$  to  $\mathcal{X}_t$ 
13:
14:  return  $\mathcal{X}_t$ 

```

The pseudo code E.4 can be explained as follows:

- in Line 3 both the set representing the belief $bel(x_t)$ and $\overline{bel}(x_t)$ are constructed and initialized to the empty set;
- in Line 6 the m -th sample is generated from the distribution $p(x_t|u_t, x_{t-1})$, which is why \mathcal{X}_{t-1} is needed;
- in Line 7 for the m -th particle, the *importance factor* $w_t^{[m]}$ is calculated. The importance factor can be interpreted as the weight of each particle;
- in Line 8 the pair particle/weight is inserted in the set representing $\overline{bel}(x_t)$;
- the key part of the algorithm happens in Lines from 10 to 14, where M particles are drawn from the set $\bar{\mathcal{X}}_t$ with a probability proportional to their weight, to populate the set \mathcal{X}_t . This process is called *resampling* or *importance sampling*. One may also notice that, since the particles are drawn by incorporating the importance factor, in the resulting set \mathcal{X}_t many duplicates can be found corresponding to higher weights. As a consequence, particles with higher weights survive and are duplicated in the resampling process, while particles with lower weights are more likely to disappear.

The efficiency of this filter lies in not making parametric assumptions on the posterior density, which allows it to represent also complex multimodal beliefs. If this property has on one hand made the fortune of this filter, on the other it increases considerably the computational complexity, which is exponential with the number of particles. Fortunately the number of particles can be adapted.

Ridgeback Simulation source codes

This appendix contains the `.yaml` parametrization files and the `.xml` launch files that have been used in simulating and testing the Ridgeback (Chapter 6).

F.1 Planners

This section contains the parametrization file for the planners, both in the case of mapping and navigation.

F.1.1 Planner_Mapping.yaml

```

##### MOVE BASE #####
# http://wiki.ros.org/move_base
5
shutdown_costmaps: false

controller_frequency: 5.0 #The rate in Hz at which to run the control loop and send
  velocity commands to the base.
controller_patience: 15.0 #How long the controller will wait in seconds without
  receiving a valid control before space-clearing operations are performed.
10
planner_frequency: 2.0 #The rate in Hz at which to run the global planning loop. If the
  frequency is set to 0.0, the global planner will only run when a new goal is
  received or the local planner reports that its path is blocked.
planner_patience: 5.0 #How long the planner will wait in seconds in an attempt to find a
  valid plan before space-clearing operations are performed.

oscillation_timeout: 0.0 #How long in seconds to allow for oscillation before executing
  recovery behaviors. A value of 0.0 corresponds to an infinite timeout.
15
oscillation_distance: 0.5 #How far in meters the robot must move to be considered not to
  be oscillating. Moving this far resets the timer counting up to the ~
  oscillation_timeout

recovery_behavior_enabled: true #Whether or not to enable the move_base recovery
  behaviors to attempt to clear out space.
clearing_rotation_allowed: true #Determines whether or not the robot will attempt an in-
  place rotation when attempting to clear out space.
20
##### GLOBAL PLANNER #####

NavfnROS:
# http://wiki.ros.org/navfn
25
  allow_unknown: true #Specifies whether or not to allow navfn to create plans that
  traverse unknown space.
  default_tolerance: 0.8 #A tolerance on the goal point for the planner.
  visualize_potential: false #Specifies whether or not to visualize the potential area
  computed by navfn via a PointCloud2

##### LOCAL PLANNER #####
30
DWAPlanerROS:

  acc_lim_x: 0.5
  acc_lim_y: 0.5
35  acc_lim_th: 2.0

  max_trans_vel: 0.55
  min_trans_vel: 0.1

40  max_vel_x: 0.55
  min_vel_x: -0.1

```

```

max_vel_y: 0.1
min_vel_y: -0.1
45
max_rot_vel: 1.0
min_rot_vel: 0.4

yaw_goal_tolerance: 0.157
xy_goal_tolerance: 0.08
latch_xy_goal_tolerance: false
50

vx_samples: 5
vy_samples: 5
vth_samples: 10
55

path_distance_bias: 32.0
goal_distance_bias: 24.0
occdist_scale: 0.4
forward_point_distance: 0.325
60

publish_cost_grid_pc: true
publish_traj_pc: false

```

F.1.2 Planner_Navigation.yaml

```

##### MOVE BASE #####
# http://wiki.ros.org/move_base
5 shutdown_costmaps: false

controller_frequency: 5.0 #The rate in Hz at which to run the control loop and send
  velocity commands to the base.
controller_patience: 15.0 #How long the controller will wait in seconds without
  receiving a valid control before space-clearing operations are performed.
10 planner_frequency: 2.0 #The rate in Hz at which to run the global planning loop. If the
  frequency is set to 0.0, the global planner will only run when a new goal is
  received or the local planner reports that its path is blocked.
planner_patience: 5.0 #How long the planner will wait in seconds in an attempt to find a
  valid plan before space-clearing operations are performed.

oscillation_timeout: 0.0 #How long in seconds to allow for oscillation before executing
  recovery behaviors. A value of 0.0 corresponds to an infinite timeout.
oscillation_distance: 0.5 #How far in meters the robot must move to be considered not to
  be oscillating. Moving this far resets the timer counting up to the ~
  oscillation_timeout
15
recovery_behavior_enabled: true #Whether or not to enable the move_base recovery
  behaviors to attempt to clear out space.
clearing_rotation_allowed: true #Determines whether or not the robot will attempt an in-
  place rotation when attempting to clear out space.

##### GLOBAL PLANNER #####
20 NavfnROS:
# http://wiki.ros.org/navfn

  allow_unknown: false #Specifies whether or not to allow navfn to create plans that
    traverse unknown space.
25  default_tolerance: 0.8 #A tolerance on the goal point for the planner.
  visualize_potential: false #Specifies whether or not to visualize the potential area
    computed by navfn via a PointCloud2

OffsetPlanner:
30  allow_unknown: false #Specifies whether or not to allow navfn to create plans that
    traverse unknown space.
  default_tolerance: 0.8 #A tolerance on the goal point for the planner.
  publish_potential: false
  use_dijkstra: false
35  offset: 1.0

```

```

GlobalPlanner:
    allow_unknown: false #Specifies whether or not to allow navfn to create plans that
                        traverse unknown space.
40    default_tolerance: 0.8 #A tolerance on the goal point for the planner.
    publish_potential: false
    use_dijkstra: false

45 ##### LOCAL PLANNER #####

DWAPlannerROS:
    acc_lim_x: 0.1
50    acc_lim_y: 0.1
    acc_lim_th: 0.07

    max_trans_vel: 0.4
    min_trans_vel: 0.01

55    max_vel_x: 0.4
    min_vel_x: -0.1

    max_vel_y: 0.3
60    min_vel_y: -0.3

    max_rot_vel: 0.3
    min_rot_vel: 0.01

65    yaw_goal_tolerance: 0.0785
    xy_goal_tolerance: 0.03
    latch_xy_goal_tolerance: true

    vx_samples: 8
70    vy_samples: 5
    vth_samples: 13

    path_distance_bias: 32.0
    goal_distance_bias: 24.0
75    occdist_scale: 0.1
    forward_point_distance: 0.2

    publish_cost_grid_pc: true
    publish_traj_pc: false

```

F.2 Costmap_Common.yaml

```

##### COMMON COSTMAP #####
#http://wiki.ros.org/costmap_2d

map_type: costmap
5
footprint: [[0.48, -0.40], [0.48, 0.40], [-0.48, 0.40], [-0.48, -0.40]]
footprint_padding: 0.1

global_frame: map
10 robot_base_frame: base_link
transform_tolerance: 5.0
update_frequency: 2.0
publish_frequency: 2.0

15 obstacle_range: 2.5
raytrace_range: 3.0

#layer definitions
20 static:
    map_topic: /map
    subscribe_to_updates: true

inflation:
25 inflation_radius: 0.8 #bigger than circumscribed_radius defined in costmap_local

```

```

obstacle:
  observation_sources: "scan scan_r"
  scan: {sensor_frame: front_laser, data_type: LaserScan, topic: front/scan, marking:
    true, clearing: true, min_obstacle_height: -2.0, max_obstacle_height: 2.0,
    obstacle_range: 2.5, raytrace_range: 3.0}
30  scan_r: {sensor_frame: rear_laser, data_type: LaserScan, topic: rear/scan, marking:
    true, clearing: true, min_obstacle_height: -2.0, max_obstacle_height: 2.0,
    obstacle_range: 2.5, raytrace_range: 3.0}
  track_unknown_space: false

```

F.3 Gmapping.yaml

```

##### GMAPPING #####
#http://wiki.ros.org/gmapping

odom_frame: "odom"
5 base_frame: "base_link"
  map_frame: "map"

throttle_scans: 1 #Process 1 out of every this many scans.

10 map_update_interval: 5.0 #How long (in seconds) between updates to the map. Lowering
    this number updates the occupancy grid more often, at the expense of greater
    computational load.

maxUrange: 5.0 #The maximum usable range of the laser. A beam is cropped to this value.
maxRange: 10.0 #The maximum range of the sensor. If regions with no obstacles within the
    range of the sensor should appear as free space in the map, set maxUrange < maximum
    range of the real sensor <= maxRange.

15 sigma: 0.05 #The sigma used by the greedy endpoint matching.
kernelSize: 1 #The kernel in which to look for a correspondence.
lstep: 0.05 #The optimization step in translation.
astep: 0.05 #The optimization step in rotation
iterations: 5 #The number of iterations of the scanmatcher.
20 lsigma: 0.075 #The sigma of a beam used for likelihood computation.
ogain: 3.0 #Gain to be used while evaluating the likelihood, for smoothing the
    resampling effects.
lskip: 0 #Number of beams to skip in each scan.
minimumScore: 0.0 #Minimum score for considering the outcome of the scan matching good.

25 srr: 0.01 #Odometry error in translation as a function of translation.
srt: 0.02 #Odometry error in translation as a function of rotation.
str: 0.01 #Odometry error in rotation as a function of translation.
stt: 0.02 #Odometry error in rotation as a function of rotation.

30 linearUpdate: 0.1 #Process a scan each time the robot translates this far.
angularUpdate: 0.05 #Process a scan each time the robot rotates this far.
temporalUpdate: -1.0 #Process a scan if the last scan processed is older than the update
    time in seconds. A value less than zero will turn time based updates off.
resampleThreshold: 0.5 #The Neff based resampling threshold.
particles: 10 #Number of particles in the filter.

35 # Initial map size (in metres)
xmin: -25.0
ymin: -25.0
xmax: 25.0
40 ymax: 25.0

# Processing parameters (resolution of the map)
delta: 0.02 #Resolution of the map (in metres per occupancy grid block).
l1samplerange: 0.01 #Translational sampling range for the likelihood.
45 l1samplestep: 0.01 #Translational sampling step for the likelihood.
lasamplerange: 0.005 #Angular sampling range for the likelihood.
lasamplestep: 0.005 #Angular sampling step for the likelihood.

transform_publish_period: 0.05 #How long (in seconds) between transform publications.

```

F.4 Frontier Exploration.yaml

```

##### FRONTIER EXPLORATION #####
#http://wiki.ros.org/frontier_exploration

frequency: 2.0 #Frequency with which to reprocess the costmap for next frontier goal. If
0.0, only ask for new frontier when last frontier was reached via move_base. Higher
frequencies submit move_base goals more often and create 'smoother' exploration.
5 goal_aliasing: 0.6 #When frequency > 0.0, ~goal_aliasing is the required distance delta
between the last goal and a new goal, before the new goal is submitted to move_base.
It is safe to set to anywhere within sensor_range/2 > ~goal_aliasing > 0.0, and
this parameter will reduce the amount of redundant goals sent during 'smooth'
exploration.

explore_costmap:
  track_unknown_space: true
  rolling_window: false
10

plugins:
- {name: external,          type: "costmap_2d::StaticLayer"}
- {name: explore_boundary, type: "frontier_exploration::BoundedExploreLayer"}

15 explore_boundary:
  resize_to_boundary: false
  frontier_travel_point: middle #When outputting pose of next frontier via ~
get_next_frontier, define the geometric property of frontier to output as pose.
position. Available- closest point to robot, middle point of frontier, centroid (
cartesian average) of all frontier points.

  explore_clear_space: false #set to false for gmapping, true if re-exploring a known
area
20

  min_frontier_dist: 1.0 #minimal distance from the robot position for frontier points
to be valid
  min_frontier_size: 7 #minimum number of frontier cells to be considered as frontier
point
  min_frontier_clear_dist: 0.4 #minimum distance of a frontier point to the closest
obstacle

```

F.5 Amcl.yaml

```

##### AUGMENTED MONTE CARLO LOCALIZATION #####
#http://wiki.ros.org/amcl

use_map_topic: false
5

gui_publish_rate: -1.0
laser_max_beams: 720
laser_min_range: 0.1
laser_max_range: 30.0
10 min_particles: 100
max_particles: 500
# Maximum error between the true distribution and the estimated distribution.
kld_err: 0.05
kld_z: 0.99
15

# Odometry motion model
odom_model_type: omni-corrected
odom_alpha1: 0.010
odom_alpha2: 0.010
20 odom_alpha3: 0.020
odom_alpha4: 0.010
odom_alpha5: 0.006

# Beam Model for Range Finder
25 laser_z_hit: 0.5
laser_z_short: 0.05
laser_z_max: 0.05
laser_z_rand: 0.5
laser_sigma_hit: 0.2
30 laser_lambda_short: 0.1
laser_model_type: likelihood_field

```

```
# Maximum distance to do obstacle inflation on map, for use in likelihood_field model.
laser_likelihood_max_dist: 2.0
35 # Translational movement required before performing a filter update.
update_min_d: 0.1
# Rotational movement required before performing a filter update.
update_min_a: 0.314
odom_frame_id: odom
40 base_frame_id: base_link
global_frame_id: map
# Number of filter updates required before resampling.
resample_interval: 1
# Increase tolerance because the computer can get quite busy
45 transform_tolerance: 1.0
# Exponential decay rate for the slow average weight filter, used in deciding when to
  recover by adding random poses. A good value might be 0.001.
recovery_alpha_slow: 0.001
# Exponential decay rate for the fast average weight filter, used in deciding when to
  recover by adding random poses. A good value might be 0.1.
recovery_alpha_fast: 0.1
50 # Initial pose mean
initial_pose_x: 0.0
initial_pose_y: 0.0
initial_pose_a: 0.0
initial_cov_xx: 25.0 #5m
55 initial_cov_yy: 25.0 #5m
initial_cov_aa: 40.0 #2pi rad

# When set to true, AMCL will only use the first map it subscribes to, rather than
  updating each time a new one is received.
first_map_only: false
```


F.6 Exploration.launch

```

<!-- LAUNCH FILE FOR EXPLORATION -->

<launch>

5   <!-- Run gmapping -->
   <arg name="scan_topic" default="front/scan" />

   <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping" output="screen">
     <roscpp param file="$(find ridgeback_config)/config/gmapping.yaml" command="load"/>
10    <remap from="scan" to="$(arg scan_topic)"/>
   </node>

   <!-- Run Move Base -->
15   <node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen"
     >

     <roscpp param file="$(find ridgeback_config)/config/costmap_common.yaml" command="load"
       ns="global_costmap" />
     <roscpp param file="$(find ridgeback_config)/config/costmap_common.yaml" command="load"
       ns="local_costmap" />

20    <roscpp param file="$(find ridgeback_config)/config/costmap_local.yaml" command="load" /
     >
     <roscpp param file="$(find ridgeback_config)/config/costmap_global.yaml" command="load"
       />

     <roscpp param file="$(find ridgeback_config)/config/planner.yaml" command="load" />

25    <remap from="odom" to="odometry/filtered" />
   </node>

   <!-- Run Frontier Exploration -->
   <node pkg="frontier_exploration" type="explore_client" name="explore_client" output="
     screen"/>

30   <node pkg="frontier_exploration" type="explore_server" name="explore_server" output="
     screen">

     <roscpp param file="$(find ridgeback_config)/config/costmap_common.yaml" command="load"
       ns="explore_costmap" />
     <roscpp param file="$(find ridgeback_config)/config/costmap_exploration.yaml" command="
       load" />

35   </node>
</launch>

```

F.7 Navigation.launch

```

<!-- LAUNCH FILE FOR NAVIGATION -->

<launch>

5   <!-- Run the map server -->
   <arg name="map_file" default="$(find ridgeback_config)/map/cobotLab_map.yaml"/>
   <node name="map_server" pkg="map_server" type="map_server" args="$(arg map_file)" />

   <!-- Run AMCL -->
10   <arg name="scan_topic" default="front/scan" />

   <node pkg="amcl" type="amcl" name="amcl">
     <roscpp param file="$(find ridgeback_config)/config/amcl.yaml" command="load" ns=""/>
     <remap from="scan" to="$(arg scan_topic)"/>
15   </node>

   <!-- Run Move Base -->
   <node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen"
     >

```

```

20 <rosparam file="$(find ridgeback_config)/config/costmap_common.yaml" command="load"
    ns="global_costmap" />
    <rosparam file="$(find ridgeback_config)/config/costmap_common.yaml" command="load"
    ns="local_costmap" />

    <rosparam file="$(find ridgeback_config)/config/costmap_local.yaml" command="load" />
    >
    <rosparam file="$(find ridgeback_config)/config/costmap_global.yaml" command="load"
    />
25
    <rosparam file="$(find ridgeback_config)/config/planner.yaml" command="load" />

    <remap from="odom" to="odometry/filtered" />
    </node>
30 </launch>

```

F.8 Gazebo.launch

```

<!-- LAUNCH FILE FOR GAZEBO SIMULATOR -->

<launch>
  <arg name="use_sim_time" default="true" />
  <arg name="gui" default="true" />
  <arg name="headless" default="false" />
  <arg name="world_name" default="/home/xmaiov/Documents/gazebo_worlds/CobotLab.world" />
  >

  <!-- Configuration of Ridgeback. See ridgeback_description for details. -->
  <arg name="config" default="$(optenv RIDGEBACK_CONFIG base)" />

  <!-- Launch Gazebo with the specified world -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="debug" value="0" />
    <arg name="gui" value="$(arg gui)" />
    <arg name="use_sim_time" value="$(arg use_sim_time)" />
    <arg name="headless" value="$(arg headless)" />
    <arg name="world_name" value="$(arg world_name)" />
    <arg name="paused" value="false"/>
  </include>

  <!-- Load Ridgeback description, controllers, and teleop nodes -->
  <include file="$(find ridgeback_description)/launch/description.launch">
    <arg name="config" value="$(arg config)" />
  </include>
  <include file="$(find ridgeback_control)/launch/control.launch" />
  <include file="$(find ridgeback_control)/launch/teleop.launch">
    <arg name="joystick" value="false"/>
  </include>

  <rosparam param="/gazebo_ros_control/pid_gains">
    front_left_wheel:
      p: 1
      i: 0.1
      d: 0
    front_right_wheel:
      p: 1
      i: 0.1
      d: 0
    rear_left_wheel:
      p: 1
      i: 0.1
      d: 0
    rear_right_wheel:
      p: 1
      i: 0.1
      d: 0
  </rosparam>

  <!-- Spawn Ridgeback -->
  <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model"
    args="-urdf -model ridgeback -param robot_description -x 0 -y 0 -z 0.1" />
</launch>

```

Bibliography

- [1] EU Project ROSIN:
<http://rosin-project.eu/>
- [2] ABB in brief:
<http://new.abb.com/about/abb-in-brief>
- [3] ABB Group:
https://en.wikipedia.org/wiki/ABB_Group
- [4] History of ABB:
<http://new.abb.com/about/abb-in-brief/history>
- [5] ABB, Power Grids Division:
<http://new.abb.com/about/our-businesses/power-grids-division>
- [6] ABB, Electrification Products:
<http://new.abb.com/about/our-businesses/electrification-products-division>
- [7] ABB, Discrete Automation and Motion:
<http://new.abb.com/ie/about/our-businesses/discrete-automation-and-motion-division>
- [8] ABB, Process Automation:
<http://new.abb.com/ie/about/our-businesses/process-automation-division>
- [9] ABB FIA Formula E:
http://new.abb.com/formula-e?itm_source=ABBCOM&itm_medium=banner_cta&itm_campaign=abbformulae&itm_content=hp_racetrack_slider
- [10] ABB Corporate Research Center:
<http://new.abb.com/about/technology/corporate-research-centers>
- [11] ABB Corporate Research Center in Sweden:
<http://new.abb.com/about/technology/corporate-research-centers/corporate-research-center-sweden>
- [12] ABB's IRB 14000 YuMi:
<http://new.abb.com/products/robotics/industrial-robots/yumi>
- [13] Clearpath Ridgeback:
<https://www.clearpathrobotics.com/ridgeback-indoor-robot-platform/>
- [14] RobotStudio Introduction:
<http://new.abb.com/products/robotics/robotstudio>
- [15] RobotStudio RAPID User Manual:
https://library.e.abb.com/public/688894b98123f87bc1257cc50044e809/Technical%20reference%20manual_RAPID_3HAC16581-1_revJ_en.pdf
- [16] RobotStudio *Developer Center*:
<http://developercenter.robotstudio.com/blobproxy/devcenter/RobotStudio/html/a63094bb-1e05-4d31-8686-1437eb34519c.htm>

- [17] RobotStudio *Namespaces*:
<http://developercenter.robotstudio.com/blobproxy/devcenter/RobotStudio/html/e6b0bdc1-06ee-3c59-1052-5fcb92beb85e.htm>
- [18] RobotStudio *Smart Component*:
<http://developercenter.robotstudio.com/blobproxy/devcenter/RobotStudio/html/1159c9a8-da11-4d6e-b707-73ad4193d2b1.htm>
- [19] RobotStudio *Smart Component Class*:
<http://developercenter.robotstudio.com/blobproxy/devcenter/RobotStudio/html/a80fa180-48d3-28ce-1fab-8d0539112ac1.htm>
- [20] RobotStudio *Add-In*:
<http://developercenter.robotstudio.com/blobproxy/devcenter/RobotStudio/html/b3473c40-97bc-4484-b566-e7840da963fb.htm>
- [21] RobotStudio *Camera Class*:
<http://developercenter.robotstudio.com/blobproxy/devcenter/RobotStudio/html/80aae580-13d1-3b17-b3bb-fd0d38a5a094.htm>
- [22] TCP Communication Protocol:
https://en.wikipedia.org/wiki/Transmission_Control_Protocol
- [23] TCP C# Server class:
[https://msdn.microsoft.com/en-us/library/system.net.sockets.tcplistener\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.net.sockets.tcplistener(v=vs.110).aspx)
- [24] TCP Client Server in C#:
<https://www.codeproject.com/Articles/1415/Introduction-to-TCP-client-server-in-C>
- [25] UDP Communication Protocol:
https://en.wikipedia.org/wiki/User_Datagram_Protocol
- [26] UDP C# Client class:
[https://msdn.microsoft.com/en-us/library/system.net.sockets.udplclient\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.net.sockets.udplclient(v=vs.110).aspx)
- [27] Google Protocol Buffers:
<https://developers.google.com/protocol-buffers/>
- [28] Google Protocol Buffers, Developer Guide:
<https://developers.google.com/protocol-buffers/docs/overview>
- [29] Google Protocol Buffers, Language Guide:
<https://developers.google.com/protocol-buffers/docs/proto>
- [30] Gaëtan Garcia, *Wheeled Mobile Robotics, Kinematic Modelling*, Ecole Centrale de Nantes, 2014.
- [31] Degrees of Freedom:
[https://en.wikipedia.org/wiki/Degrees_of_freedom_\(mechanics\)](https://en.wikipedia.org/wiki/Degrees_of_freedom_(mechanics))
- [32] Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, Giuseppe Oriolo, *“Robotics - Modelling, Planning and Control”*, ©Springer-Verlag London Limited, 2010.

- [33] Lih-Chang Lin, Hao-Yin Shih, “*Modeling and Adaptive Control of an Omni-Mecanum-Wheeled Robot*”, article in *Intelligent Control and Automation*, Vol. 4, No. 2, ©Pleiades Publishing, 2015.
- [34] Sebastian Thrun, Wolfram Burgard, Dieter Fox, “*Probabilistic Robotics*”, ©Massachusetts Institute of Technology, 2006.
- [35] Giorgio Grisetti, Cyrill Stachniss, Wolfram Burgard, “*Improved Techniques for Grid Mapping with Rao-Blackwellized Particle Filter*”, IEEE Transactions on Robotics, Volume 23, 2007.
- [36] Howie Choset, Kevin M. Lynch, Seth Hutchinson, George A. Kantor, Wolfram Burgard, Lydia E. Kavraki, Sebastian Thrun, “*Principles of Robot Motion: Theory, Algorithms, and Implementations*”, ©Massachusetts Institute of Technology, 2005.
- [37] Fadzli S. Abdullah, Sani I. Abdulkadir, Mokhairi Makhtar, Azrul A. Jamal, “*Robotic Indoor Path Planning using Dijkstra’s Algorithm with Multi-Layer Dictionaries*”, 2015.
- [38] Dieter Fox, Wolfram Burgard, Sebastian Thrun, “*The Dynamic Window Approach to Collision Avoidance*”, Published in *IEEE Robotics & Automation Magazine* (Volume: 4, Issue: 1, Mar 1997).
- [39] ROS Costmap 2D:
http://wiki.ros.org/costmap_2d
- [40] ROS Trajectory Planner Package:
http://wiki.ros.org/base_local_planner
- [41] ROS DWA Local Planner Package:
http://wiki.ros.org/dwa_local_planner
- [42] Sean Quinlan, Oussama Kathib, “*Elastic Bands: Connecting Path Planning and Control*”, Stanford University.
- [43] Brian Yamauchi, “*A Frontier-Based Approach for Autonomous Exploration*”, Navy Center for Applied Research in Artificial Intelligence, Naval Research Laboratory, Washington DC.
- [44] ROS Introduction:
<http://wiki.ros.org/ROS/Introduction>
- [45] ROS Concepts:
<http://wiki.ros.org/ROS/Concepts>
- [46] ROS Wiki:
<http://wiki.ros.org/Documentation>
- [47] Gazebo Simulator:
<http://gazebosim.org>
- [48] ROS Navigation Stack:
<http://wiki.ros.org/navigation/Tutorials/RobotSetup>
- [49] ROS Move Base:
http://wiki.ros.org/move_base

- [50] ROS NavFn Package:
<http://wiki.ros.org/navfn>
- [51] ROS Gmapping Package:
<http://wiki.ros.org/gmapping>
- [52] ROS Frontier Exploration Package:
http://wiki.ros.org/frontier_exploration
- [53] ROS Augmented Monte Carlo Localization Package:
<http://wiki.ros.org/amcl>
- [54] Changbae Jung, Woojin Chung, “*Design of Test Traks for Odometry Calibration of Wheeled Mobile Robots*”, International Journal of Advanced Robotic Systems, Vol. 8, No. 4, 2011.
- [55] Etienne Colle, Simon Galerne, “*A robust set approach for mobile robot localization in ambient environment*”, Autonomous Robots, Springer Verlag, 2018.
- [56] Bøgh Simon, Hvilsoj Mads, Kristiansen Morten, Madsen Ole, “*Autonomous Industrial Mobile Manipulation (AIMM): from Research to Industry*”, in Proceedings of the 42nd International Symposium on Robotics VDE Verlag GMBH, 2011.
- [57] Stefanie Angerer, Christoph Strassmair, Max Staehr, Maren Roettenbacher, Neil M. Robertson, “*Give me a hand - The Potential of Mobile Assistive Robots in Automotive Logistic and Assembly Applications*”, in IEEE International Conference on Technologies for Practical Robot Applications (TePRA), 2012.
- [58] Andreas Dömel, Smion Kriegel, Michael Kaßecker, Manuel Brucker, Tim Bodenmüller, Michael Suppa, “*Toward fully autonomous mobile manipulation for industrial environments*”, in International Journal of Advanced Robotic System, 2017.
- [59] Wisama Kalhil, “*Modeling and Control of Manipulators*”, course-book at Ecole Centrale de Nantes.
- [60] Bruno Siciliano, Oussama Khatib, “*Springer Handbook of Robotics*”, ©Springer-Verlag Berlin Heidelberg, 2016.
- [61] Lydia E. Kavraki, Petr Švestka, Jean-Claude Latombe, Mark H. Overmars “*Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces*”, in IEEE Transactions on Robotics and Automation, Vol. 12, No. 4, 1996.
- [62] James J. Kuffner, Steven M. LaValle “*RRT-Connect: An Efficient Approach to Single-Query Path Planning*”, in IEEE International Conference on Robotics and Automation, 2000.
- [63] Robot Web Services C++ library:
https://github.com/ros-industrial/abb_librws
- [64] Moveit! Home Page:
<https://moveit.ros.org/>
- [65] Moveit! System Architecture and Concepts:
<https://moveit.ros.org/documentation/concepts/>
- [66] Moveit! Pick&Place Tutorial:
http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/pick_place/pick_place_tutorial.html

- [67] Moveit! - list of planners:
<http://ompl.kavrakilab.org/planners.html>
- [68] Moveit! - The Open Motion Planning Library:
<http://ompl.kavrakilab.org/index.html>