POLITECNICO DI MILANO
Master of Science in Automation and Control Engineering
Dipartimento di Elettronica, Informazione e Bioingegneria

POLITECNICO
MILANO 1863

DYNAMIC MOTION PLANNING FOR A TETHERED INSPECTION ROVER

Supervisor: Prof. Fredy Orlando Ruiz Palacios
Co-Supervisor: Prof. Lorenzo Mario Fagiano

Master Thesis by:
Pedro Henrique G dos Santos, Matr. 912800

Academic Year 2020–2021

Al SIGNORE, *da cui deriva tutta la scienza.*

Acknowledgements

I am grateful to my parents, Ana and Fabio, whose tireless support and encouragement made possible the completion of my wildest endeavours. Words will never be enough thank them for what they sacrificed for me. I would also like to thank my brother and best friend, Paulo Victor, a constant source of inspiration and good example in my life. Although now we are equal in academic degree, I shall always see him as my master. I am particularly grateful to Nidhi, my treasure. Her patience and faith in me were invaluable and vital to this accomplishment. A supporting pillar and calming breeze in difficult times, I hope to repay her with love and support always.

I would like to give special thanks to Prof. Ruiz Palacios and Prof. Fagiano for the precious guidance provided throughout this thesis. Their patience, kindness and willingness to help shall be remembered. I extend my thanks to professors Maria Helena Baccar, Max Suell Dutra, Felipe Sass, José Santisteban and many others, who along my path shared their knowledge and contributed indirectly to this achievement. I am also grateful to all my friends, each one of them contributing in their own particular way to this accomplishment. I am blessed by their quantity and quality.

Above all, I thank the Lord, who made everything possible. I was never alone, for His presence was always with me. I felt His care through every person I know and met and His provision, mercy and love always followed me and gave me strength to endure every storm and rejoice in every sunny day. My heart shall always be grateful. To Him, all the glory and honour.

Abstract

Mobile robots for inspection purposes have been increasingly studied, developed and perfected to achieve new possibilities and face new challenges. In this context, a particular challenging scenario is the motion planning for autonomous mobile robots with movement related constraints. The scope of this thesis is to develop a motion planning routine capable of managing constraints arising from the presence of a tether cable on a robot, more specifically a rover designed for subfloor inspection. The dynamic motion planning RRT^X algorithm was implemented with the herein developed geometric approach, followed by improvements in the system structure and communication to increase the autonomy of the rover. Simulations, performance comparisons and experimental tests were conducted to investigate the functionality, efficiency and reliability of the developed method, with results indicating an operating time reduction of 90% if compared to the previous algorithm, a success rate of effectively 100% and an improved communication system remarkably efficacious, especially with the implementation of the devices integration by shared environment. The outcomes signal the achievement of a fast, efficient and most of all promising solution for the dynamic motion planning for tethered robots with constraints.

Sommario

I robot mobili per l'ispezione sono stati sempre più studiati, sviluppati e perfezionati per raggiungere nuove possibilità e affrontare nuove sfide. In questo contesto, uno scenario particolarmente impegnativo è la pianificazione del movimento dei robot mobili autonomi con restrizioni del movimento. Lo scopo di questa tesi è sviluppare una routine di pianificazione del movimento in grado di gestire le restrizioni derivanti dalla presenza di un cavo di tethering su un robot, nello specifico un rover progettato per l'ispezione del sottopavimento. L'algoritmo di pianificazione dinamica del movimento RRT^X è stato implementato con l'approccio geometrico qui sviluppato, seguito da miglioramenti nella struttura del sistema e nella comunicazione per aumentare l'autonomia del rover. Simulazioni, confronti prestazionali e prove sperimentali sono state condotte per valutare la funzionalità, l'efficienza e l'affidabilità del metodo sviluppato. I risultati mostrano una riduzione del tempo operativo del 90% rispetto all'algoritmo precedente, un tasso di successo effettivo del 100% e un migliore sistema di comunicazione notevolmente efficace, soprattutto grazie all'implementazione dell'integrazione dei dispositivi per ambiente condiviso. I risultati segnalano il raggiungimento di una soluzione veloce, efficiente e soprattutto promettente per la pianificazione dinamica del movimento dei robot allacciati con restrizioni.

Contents

Acknowledgements **III**

Abstract **V**

Sommario **VII**

1 Introduction **1**

1.1 General Overview 2

1.2 Thesis Outline 5

2 Tethered Motion Planning Literature Review **7**

2.1 Problem Statement 8

2.2 State of the Art 9

2.3 Methodology 12

3 Review of the Rover **13**

3.1 Context of the Project 14

3.2 Hardware Overview 16

3.2.1 Mechanical Components 16

3.2.2 Electronic Components 18

3.2.3 Sensors 19

3.3 Software Overview 20

3.3.1 Arduino 20

3.3.2 Raspberry Pi 21

3.3.3 PC 22

3.4 Results 23

4 Path-Planning Program Development 25

4.1 RRT^X Path-Planning Dynamic Algorithm 26

4.1.1 RRT^X Main Characteristics 27

4.1.2 RRT^X Program Development 30

4.2 Constraint Problem 33

4.2.1 Geometric Approach Implementation 35

4.3 Final Program Overview 46

4.4 Simulation Results and Performance Analysis 51

4.4.1 Simulations 52

4.4.2 RRT^X Performance Comparison 70

5 Device Integration and Experimental Results 75

5.1 Device Integration 76

5.1.1 Separated Environments Operation 76

5.1.2 Raspberry Pi Single Operation 77

5.1.3 Shared Environment Operation 78

5.2 Experimental Tests 81

5.2.1 Problems Encountered 81

5.2.2 Operation Assessment 89

6 Conclusion 93

6.1 Future Developments 94

Bibliography 97

A Performance Comparison 105

A.1 Test Description 105

A.1.1 First Map 107
A.1.2 Second Map 108
A.1.3 Third Map 110
A.1.4 Conclusion 111

List of Figures

1.1 Pioneer, a tethered rover designed to remove debris from the Chernobyl disaster. 3
1.2 The inspection rover targeted herein. 4

2.1 Venn diagram representing the scope of the thesis. 8
2.2 Simplified model of an extreme terrain. 9

3.1 The rover and its internal hardware. 14
3.2 SAE in Amatrice. 15
3.3 Subfloor made of wood with wooden structure. 15
3.4 Scheme of the hardware connection. 17
3.5 Software hierarchy. 21
3.6 Illustration of the angle evaluation. 22
3.7 Flowchart of the adapted RRT* algorithm. 23
3.8 Picture taken by the rover during inspection test. 24

4.1 Rapidly-exploring Random Tree operation. 28
4.2 Example of a RRT^X execution with starting point at $(15, 2.5)$ and goal point at $(25, 40)$. 31
4.3 Example of a wrong branch deletion. 33
4.4 Sketch of the geometric approach for the constraint problem. 35
4.5 Example of the alternate path implementation. 37
4.6 Example of the tree re-span implementation. 38

4.7 Trivial example of a self-intersecting polygon. 42
4.8 An example of an obtained self-intersecting polygon incurring in a false positive result from the geometric approach. 43
4.9 The new polygon, that is the concave-hull, of the trajectory shown in Figure 4.8. 45
4.10 Example of a multi-goal execution with five goals and geometric approach to constraint problem. 47
4.11 Example of a feasible trajectory obtained in two steps, *i.e.* using the map partition strategy. 50
4.12 Simplified flowchart of the main and core routines. 51
4.13 Example of a successful execution with five goals. 53
4.14 Single goal execution with random goal point. 54
4.15 Violin plot of the time for 100 single random goal executions. 54
4.16 Single goal execution with goal point at $(32, 42)$. 55
4.17 Violin plot of the simple cost for 100 single fixed goal executions. 55
4.18 Violin plot of the time for 100 single fixed goal executions. . . 56
4.19 Four goals execution with random goal points. 57
4.20 Four goals execution with random goal points and two-steps execution. 57
4.21 Success rate of four goals execution with random goal points. 58
4.22 Violin plot of the time for 100 four random goals executions. 59
4.23 Four goals execution with fixed goal points. 59
4.24 Success rate of four goals execution with fixed goal points. . . 60
4.25 Violin plot of the simple cost for 100 four fixed goals executions. 61
4.26 Violin plot of the time for 100 four fixed goals executions. . . 61
4.27 Nine goals executions with random goal points. 62
4.28 Success rate of nine goals execution with random goal points. 63
4.29 Violin plot of the time for 100 nine random goals executions. 65
4.30 Nine goals executions with fixed goal points. 66

4.31 Success rate of nine goals execution with fixed goal points. . . 67
4.32 Violin plot of the time for 100 nine fixed goals executions. . . 68
4.33 Violin plot of the simple cost for 100 nine fixed goals executions. 68
4.34 Bar graph of the average execution time per number of goals. 69
4.35 The maps used for the performance analysis. 71
4.36 Violin plots of the iteration time for RRT^X and RRT* considering the three maps used. 73

5.1 Simplified scheme of the Separated Environments Operation. 77
5.2 Simplified scheme of the Raspberry Pi Single Operation. . . . 78
5.3 Simplified scheme of the Shared Environment Operation. . . . 80
5.4 Experiments area. 82
5.5 Sketch of the solution proposed. 84
5.6 Arduino to driver signal analysis. 85
5.7 Displacement error illustrated. 86
5.8 Straight movement experiment showing the displacement error. 87
5.9 Driver relative to the left motor. 88
5.10 Example of an experiment sample for a straight movement. . 90
5.11 Example of an experiment sample for a two goals movement. 92

A.1 First map used for the performance analysis. 106
A.2 Second map used for the performance analysis. 106
A.3 Third map used for the performance analysis. 107
A.4 Iteration time over number of iterations for the first map. . . 108
A.5 Iteration time over number of iterations for the second map. . 109
A.6 Iteration time over number of iterations for the third map. . . 110

Chapter 1

Introduction

Mobile Robotics, although recently developed if compared to other fields of study, has a longstanding existence in the imaginary of mankind. From the automaton and golem of the Greek and Hebrew myths, passing by the automated humanoid of Leonardo da Vinci and the *robota* of Karl Kapek, to the middle of the 20th century, when the imagination begun to come to fruition as a result of the great technological advances made and motivated by the Space Race [1]. Since then, mobile robots have expanded their presence to many other fields of work besides the space and neighbouring planets, such as deep seas, caves, skies, buildings and streets, covering a vast range of applications, for instance rescue operations, logistics, professional and personal service, military use, exploration or inspection. The latter particularly, the inspection robots, have been extensively studied and developed in the past decades mostly because of their important role in assuring safety and efficiency on the execution of challenging tasks. Herein the research progresses as a mobile robot designed for inspection is improved to handle dynamic environments in an autonomous, reliable and efficient manner. It is to be hoped that the progress here made may in some way contribute to such significant field of research and help future developments in the area.

1.1 General Overview

Since any material or structure is naturally susceptible to deterioration and prone to failure, long-lasting structures require maintenance to assure the proper operation, delivery of expected results and safety for any living creature in contact with it. Such applies for example to industrial machinery, power-plants, transmission lines, oil pipes, tunnel-structures and buildings. With the development of more reliable mobile robots, inspection robots capable of handling such tasks and fully or partially substituting human inspections became a reality.

The field of inspection robots is per se a diverse area of study and development, aggregating robots of all shapes and sizes that ultimately share with each-other the main purpose of substituting a human being in a task that for a given reason is impossible or inconvenient to be performed by humans. Several are the reasons to support the necessity for the tasks to be performed by a mobile robot, but amidst them it can be highlighted the necessity for inspection in environments that present potential harm to human beings or that are inaccessible and the requirement of performing repetitive time-demanding assignments that need reliability, *i.e.* a methodical inspection aiming to avoid human caused errors [2].

As examples of the former, it can be pointed: inspections in mining sites [3, 4], nuclear power-plants [5] and transmission power lines [6], due risk of explosions and gas leaks, radiation or lethal high voltages, as well as inspections in oil extraction equipment underseas [7, 8] and pipelines for oils, gases and chemicals [9, 10], which occasionally can be done by humans but more often than not are located in depths that render the access inconvenient at minimum or are highly expensive to be executed by humans. For the latter, it can be given as an example: structural inspections in large structures, such as in ship hulls [11], aircraft surfaces [12] and civil structures [13, 14].

Concerning inspection robots, it is not unusual for such robots to have a cable, formally known in the literature as umbilical cord or tether cable, physically connecting the robot to its centre of operation, where the user generally is located and from where the robot is deployed. The tether provides power to the robots, enables a fast and reliable exchange of data and allows the user to retrieve the robot in case of malfunction, which can play a major safety role depending on the application [7]. As an example, [5] proposes a tethered service rover equipped to clean debris resultant from the Chernobyl disaster, mostly motivated by another unsuccessful attempt with the cordless German robot MF-2 that stopped working after seven minutes of contact with extremely high radiation, remaining among the debris until it was removed in a special operation. Figure 1.1 presents a blueprint and a 3D render of the robot Pioneer, introduced in [5].

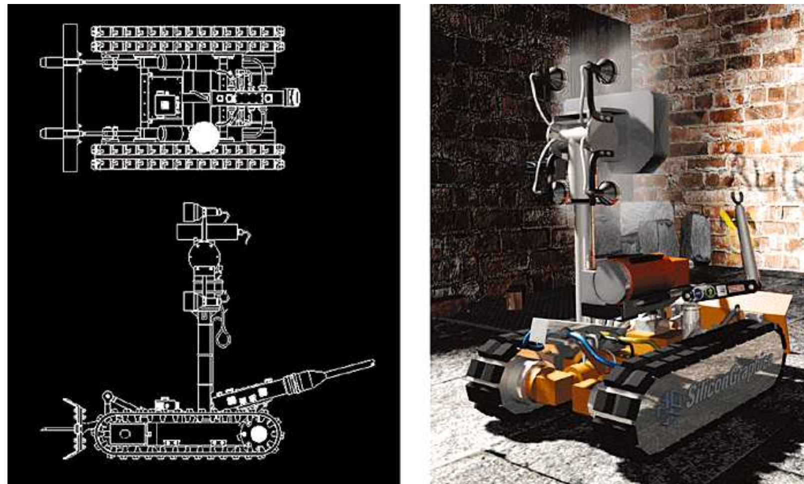

Figure 1.1: Pioneer, a tethered rover designed to remove debris from the Chernobyl disaster. Extracted from [5].

In order to stress the importance and convenience of the tether cable for some applications, it is worth to point that robots presented in [4, 5, 7, 8, 9, 10, 11, 12, 13] have all some sort of tether cable attached to them. Nevertheless, the presence of a umbilical cord brings with itself a set of problems,

especially concerning the cable dynamics and motion of the robots. Such problems are enhanced when the robot must move autonomously, *i.e.* a robot not tele-operated, requiring from the motion planning of the robot to take into consideration the tether cable, as is the case herein presented.

The rover, object of the present work, is a tethered inspection rover designed for a tight space unpractical for human inspection, the subfloor of houses in Amatrice, Italy. The rover was first introduced in [15], which presented the prototyping and full development of the hardware of the rover along with a preliminary version of the motion planning with the constraint consideration. A review of the project and specifications of the rover in addition to the description of the context and motivation behind its project can be read in Chapter 3 of the present work. The aforementioned rover can be seen in its upper view in the Figure 1.2.

Figure 1.2: The inspection rover targeted herein.

The scope of the present work is therefore the development and implementation of a suitable, fast and reliable method of path-planning for the rover presented in [15]. The path-planning method must be able to not only respond in adequate manner to the constraints imposed by the presence of the tether but also substantially improve the speed and functionality of the

motion planning of the rover such that the rover becomes capable of handling environments containing unexpected obstacles not previously mapped.

1.2 Thesis Outline

The current work is structured as follows: Chapter 2 presents in greater detail the problem being addressed, followed by a concise review of the state of the art and lastly by the methodology used for the research and development of the work. Chapter 3 proceeds to present a review of the project of the rover, developed and presented in [15]. In this chapter, the motivation and project specifications of the rover are briefly introduced, succeeded by an overview of the hardware and software developed, and at last the results obtained and future developments left.

Chapter 4 then shows the new motion planning algorithm chosen, as well as the constraint approach development. The chapter also presents an overview of the final version of the algorithm, currently running on the rover, and describes the simulations and performance tests carried out, with emphasis on the simulation results. Thereafter, Chapter 5 addresses the different modes of interaction between the PC and the Raspberry Pi, and the implementation tests performed with the rover. The results are then presented and briefly discussed. The last chapter, Chapter 6, concludes the work by providing an overview of the whole development, analysing the results obtained and introducing the future prospects. Finally, on the Appendix A, data regarding the performance comparison realised is presented.

Chapter 2

Tethered Motion Planning Literature Review

The field of motion planning has been heavily studied since even before the development of affordable mobile robots, due to its significance also to industrial robotics [16]. Aiming at attaining more efficient, reliable and fast algorithms, new methods are in constant development and well established ones are under a persistent perfecting. Furthermore, the frontier of technology is ceaselessly pushed as new and more complex applications demand new approaches for the motion planning problem. Such is the case of the motion planning for tethered inspection robots in challenging environments with regard to the space of operation.

In this context, a brief review of the literature concerning motion planning for tethered robots is presented as follows. A proper statement of the problem targeted by the current work is shown prior to it, with the objective of guiding the understanding of the motivations and requirements that conducted the development of the here presented motion planning algorithm.

2.1 Problem Statement

The problem addressed consists of implementing a reliable motion planning algorithm that first and foremost is capable of handling the fact that there are unknown obstacles inside the subfloor gap to which the rover was designed to inspect, as it can be followed in Chapter 3. The rover must, therefore, be able to re-plan its trajectory at any given moment during the mission. Secondly, the path computed must take into consideration the presence of the tether and its interaction with the obstacles when defining which path constitute a feasible one. The constraint imposed by the tether must be managed in such manner that in does not interfere with the dynamic path-planning.

To summarise, the motion planning problem of the rover is a dynamic path-planning problem with constraints related to the feasibility of path as a consequence of the presence of a tether cable. Figure 2.1 shows graphically the scope as a Venn diagram containing the two main areas tackled, with the present thesis being represented by the star in the intersection of the two.

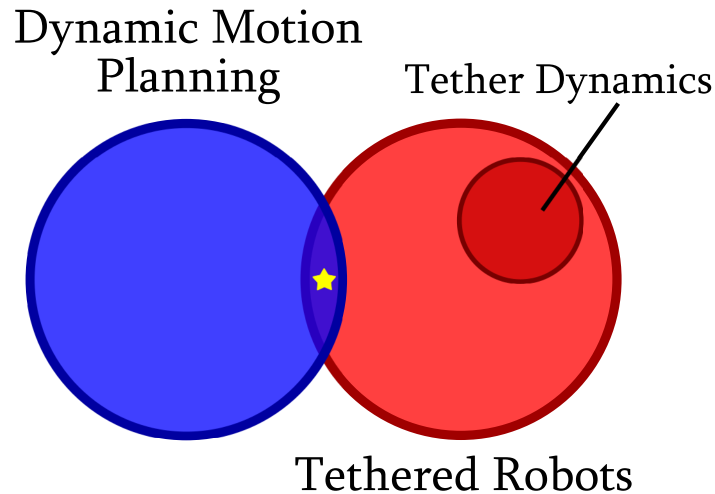

Figure 2.1: Venn diagram representing the scope of the thesis.

2.2 State of the Art

The literature describing dynamic motion planning for tethered robots is rather limited. In this respect, it is worth to point primarily the work from [17], which focus on planning the trajectory of the tethered extreme extra-planetary terrain exploration rover Axel, from NASA, considering a map not fully known and using a motion planning based on an A^* heuristic search algorithm. The application, however, considers a single goal mission on a steep plane, where the anchor point for the rover is in the upper part and the goal in the lower part of the plane, as a way to simulate a crater, analogous to the ones on the Moon and in Mars. Once the goal is reached, the rover is reeled back to its anchor position. Although the idea of reeling the cable during the operation is appealing, the fact that the specificities of the mission fulfilled by Axel and by the rover centre of this work are quite different, another approach had to be taken. Figure 2.2, taken from [17], shows the type of environment for which the rover and method aforementioned were developed.

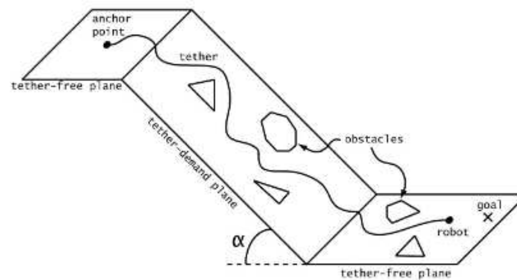

Figure 2.2: Simplified model of an extreme terrain. Extracted from [17].

As a matter of fact, for the static path-planning with a tether situation, most of the works reviewed tackle the tether interaction with the obstacles present in a map in a point-to-point linear approach, that is without considering closed trajectories. That is the case of [18] for example, which presents a space discretisation algorithm to find the feasible single goal shortest tra-

jectories for robots with constraints imposed by cables in a planar map with static obstacles. In [19], a multi-goal scenario is briefly considered, but the main goal is regarding the length constraint imposed by the tether in that context rather than assuring a trajectory that would not incur in entanglement of the cable.

Another line of study focuses on the tether dynamics while the robot is moving. Albeit many works, including the work here presented, take as an assumption the use of an ideal frictionless cable, that does not correspond to reality and cannot be always neglected. Another often disregarded behaviour concerns the curve characteristic of the cable and also its movement as a consequence of the movement of the rover to which it is attached. Taking these characteristics into consideration tends to harshly increase the complexity of the problem being approached and whether or not they can be neglected depends on the scope of the work.

As examples of works that focus on the behaviour of the cable, it can be pointed [20], where a method to approximate the shape and configuration of the cable in proposed, considering a scenario similar to [17] in which the rover or the anchor point is equipped with a reel. Additionally, [21] performs a substantial approach of possible behaviours of the tether on a planar mobile robot while proposing a cell decomposition method to generate the configuration space based on what the author calls cable events. Such analysis of the dynamics of the cables, although relevant to the present thesis, does not fall within its scope.

Regarding the dynamic path-planning for mobile robots, independently of whether they have an umbilical cord or not, the topic is still in active development and is often considered one of the most challenging topics in mobile robotics. Such is the case of [22], which uses probabilistic representation for the prediction of dynamic environments, computing the probabilistic risk of collision and planning the trajectories accordingly. In other words,

the probabilistic position of the obstacles based on the available data concerning their behaviour is used to compute paths with a lower probability of coming across a new obstacle. Therefore, a scenario where the obstacles are positioned with a determined dispersion is recommended for this method, as well as the information about their behaviour, such as for example an environment with a flux of pedestrians.

In [23], a potential field method equipped to handle dynamic environments is presented, where both relative position and velocity of the planar mobile robot in respect to the goal and the obstacles are considered in the planning. Moreover, [24] presents a method called Partial Motion Planning, focusing on the real-time constraint imposed by the motion in highly dynamic environments, where the obstacles may also be moving. Successive random trees are created and spanned for a predetermined period of time while the rover executes its mission traversing the ICS-free path computed in the previous step, with ICS standing for inevitable collision states. As an example of an optimisation-based method, it can be pointed [25], which uses spline parametrisation in combination with B-spline relaxations and a receding horizon approach to compute smooth paths in highly dynamic environments for both holonomic and nonholonomic mobile robots.

The optimal motion planning method CHOMP, or Covariant Hamiltonian Optimisation for Motion Planning, presented in [26], although not explicitly equipped to operate in dynamic environments, is said to be suited for many real-world planning queries, quickly converging to a locally optimal trajectory, and to excel in obstacle handling. Combined with methods to identify unknown obstacles using the hardware available in the rover, an adaptation of this method could lead to an adequate dynamic environment response. As exposed in Chapter 4 however, the RRT^X method [27] was found better suited considering the specificities of the problem at hand and a geometry based approach was developed to include the tether constraint in

the motion planning algorithm. A more complete description of the RRT^X algorithm can also be observed in Chapter 4.

2.3 Methodology

The workflow of the present thesis includes firstly the thorough review of the previous work, that is [15], where the rover was initially proposed and developed. Subsequently, a review of the literature was performed considering the defined problem addressed and scope of this work. Once the computational approach to be taken for the dynamic motion planning was determined, the implementation in a computer program was realised. Thereafter, regarding the constraints imposed by the presence of the tether, the approach used to deal with the constraint was developed and incorporated in the motion planning.

Afterwards, simulations were performed to assess the quality and performance response of the method created. Comprising the test phase were also comparative performance tests with a method taken as a staple reference in the literature and, subsequently, experimental tests with the rover embedding the developed method, to evaluate the response of the method in the real-case scenario. Hardware adaptations in the embedded systems of the rover were also made as they were needed in this phase. Finally, the results were analysed and the documentation concerning the thesis project realised was written.

Chapter 3

Review of the Rover

The robot, centre of the present thesis and first presented in [15], is a differential drive tethered rover designed to inspect structures characterised by their difficult access due to limited dimensions. The rover is equipped with sensors to provide data about the structure being examined, further analysed to assess the condition of the section inspected, as well as a camera, whose image is constantly shared with the PC of the user linked with the rover via the umbilical cable, and sensors to aid the motion and localisation of the rover.

Concerning the control of the rover, it can be tele-operated or move autonomously from user provided initial instructions. When tele-operated, referred in [15] as manual mode, it behoves the user to cautiously plan the trajectory to be traversed such that the tether cable does not entwine by encircling a column of the building. For the autonomous operation, focused herein, the system of the rover must be responsible for choosing a path that does not incur in the cable being entangled in the columns. Figure 3.1 displays the rover with its upper part opened, making visible its internal hardware.

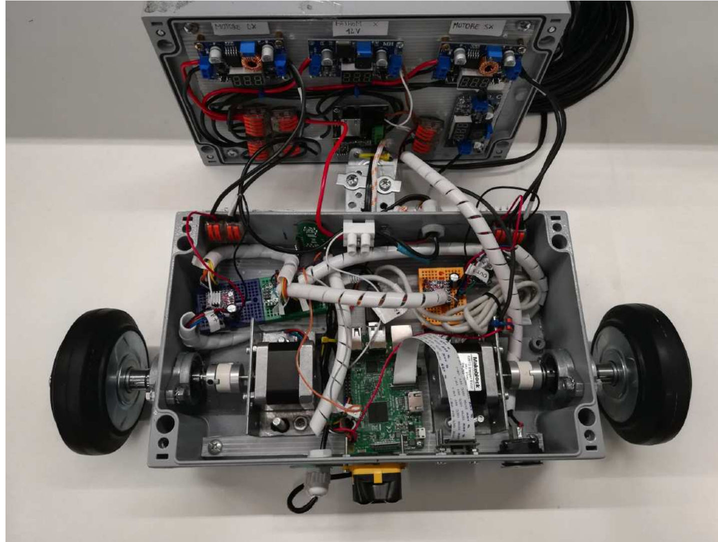

Figure 3.1: The rover and its internal hardware. Extracted from [15].

3.1 Context of the Project

The motivation behind the development of the rover comes from the necessity of inspection of the subfloors in houses built in the Italian region of Amatrice, in the Central Apennines, after the earthquake that struck the region in the year of 2016. More specifically, the rover was designed to inspect the subfloor of the *Soluzioni Abitative d'Emergenza* (*transl.* Emergency Housing Solutions), or SAE, built to house the survivors of the earthquake who lost their homes or were told by the authorities to evacuate their houses due to risk of structural collapse after the seismic event. SAE are single-family container-like housing units, available in three different sizes of 40, 60 and 80 m^2, subdivided into rooms. In total, 3702 SAE were planned to be built in the region of Amatrice [28]. Figure 3.2 shows one of the SAE built in Amatrice, in the front, while in the back the remains of a collapsed building from the earthquake can be seen.

As mentioned before, the structure of the houses being investigated are the subfloors. By subfloors, it must be understood the layer of material,

Figure 3.2: SAE in Amatrice. Extracted from [28].

in this case wood, serving as the house pavement and standing above the ground floor where the house is built upon by a structure of beams and columns. Figure 3.3 presents an example of subfloor as found in the SAE. As it can be observed, the use of subfloors generate a tight gap between the wood structure and the ground of generally a few centimetres. In the SAE case, these structures began to present infiltration problems.

Figure 3.3: Subfloor made of wood with wooden structure. Extracted from [29].

Although the inspection and identification of infiltration points can be

made with static devices, such method is very limited, as pointed by [15], not producing the desired results. In this context, it is necessary a means of inspecting the narrow gap between the subfloor and the ground in a non-invasive manner, to identify possible infiltration spots and apply the appropriate measures before bigger sections or the whole floor are completely compromised. The inspection rover is developed therefore as a solution to the aforementioned necessity.

3.2 Hardware Overview

The rover developed by [15] was designed respecting the spatial requirements of the environment to be inspected, the most severe one being naturally the height of the rover. The subfloor gap has a height of approximately 12 centimetres, so obligatorily the height of the rover must be below this limit. Figure 3.4 provides a scheme containing all the hardware of the rover, mechanical and electronic, as well as their connections. The subsequent sections address more specifically, but briefly, each hardware component involved in the operation of the rover.

3.2.1 Mechanical Components

Chassis

The chassis of the rover is characterised by a box with protection grade IP67, which although heavier than the plastic ABS alternative, has a stronger structure. The dimensions of the box are $28cm$ of length, $17cm$ of width and $9cm$ of height, therefore under the dimension limits. Holes were made to allow the passage of cables and for the wheel axis to be attached.

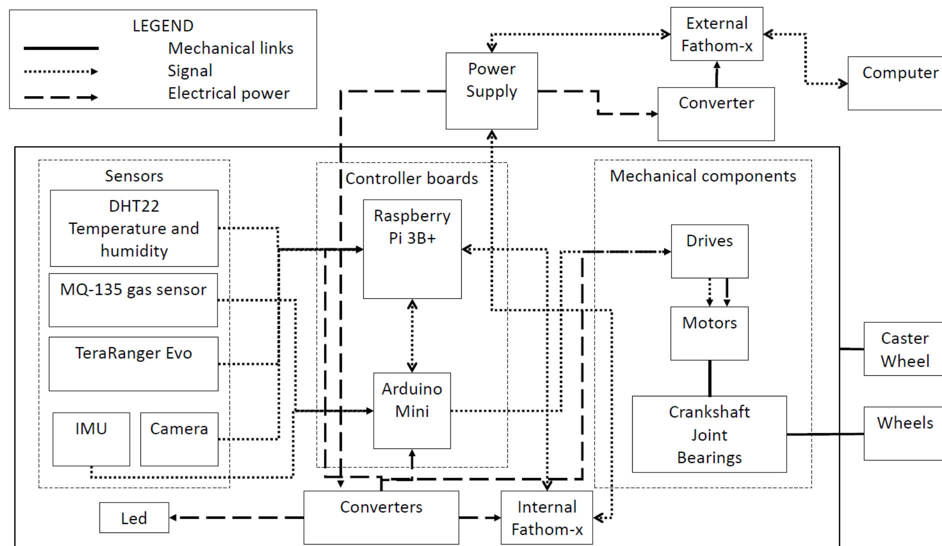

Figure 3.4: Scheme of the hardware connection. Extracted from [15].

Wheels

Regarding the wheels, the model chosen was the RMNA100, with a diameter of 10cm and made of rubber. Thus, the overall height of the rover remains under the limit required. A caster well was also added on the rear part of the rover to provide stability, totalising three wheels on the rover. As seen in Chapter 5, the RMNA100 wheels proved to be inadequate for the application intended upon testing.

Motor

The motor used to move the rover are NEMA17 bipolar stepper motors. Presenting a holding torque of 0.4Nm and 1.8° of rotation per step, or 200 steps per rotation, this motor presents a satisfactory precision and, if a slipless movement is assured, the counting of the steps can be used to determine or validate the localisation of the rover. This topic is as well further discussed in Chapter 5.

3.2.2 Electronic Components

Driver

The driver implemented was the DRV8825, after the driver L298N proved to be unsatisfactory due to overheating. According to [15], the DRV8825, although fragile, also presented a better performance for continuous movements. It is also worth to remark that a capacitor is used in the power supply of the driver to avoid damage due to destructive LC voltage spikes. Each driver is connected to the Arduino board by three signal wires, for step control, direction and motor activation, and connected to the motor by four wires, corresponding to the two pairs of windings present in the motor.

Communication

The two Fathom-X boards were embedded in the rover and attached to the PC to enable the communication between the Raspberry Pi and the PC over the power line. The board, intended for applications with umbilical cords, transfer Ethernet over power at 80 Mbps over two wires and supports cables of up to 300 metres. Inside the rover, the Ethernet cable being used was a flat one, later changed for a round Ethernet cable during the development of the present work due to the latter providing not only a better transfer rate but also, and more importantly, higher durability.

Power Supply and DC-DC Converter

The power required by the rover is provided by a 36V DC power source, *i.e.* an AC-DC converter, which through five LM2596 DC-DC converters provides the right voltages to all the electronic components in the rover. More especifically, starting from the power source, downstream is the first Fathom-X board, connected via Ethernet cable to the PC and via power cable to the rover, powered by 12V through a converter. Inside the rover,

another converter powers the second Fathom-X, connected with the Raspberry Pi, with 12V, and two dedicated converters power the drives, and consequently the motors, with 10V. A last converter powers the Raspberry Pi and the Arduino with the required 5V.

3.2.3 Sensors

Temperature and Humidity

The DHT22 is a broadly used temperature and humidity sensor compatible with Arduino and Raspberry Pi. The sensor allegedly is able to read humidity levels for 0% to 100% with at most 5% of inaccuracy, and temperatures from $-40°$ to $80°C$. During the field tests performed by [15], the accuracy of the sensor was verified to be harshly different from the nominal value.

Gas Sensor

The MQ135 gas sensor used provides a part per million count regarding the gases present in the air, in spite of that it is not capable of distinguishing the gases being read. The sensor is directly connected to the Arduino board.

Raspberry Pi Camera

This camera is intended to be used with a Raspberry Pi, so little to no shortcoming was expected from its implementation. The quality of the camera image and its frame rate, especially when used with the Raspberry Pi 4 instead of the Raspberry Pi 3, greatly increases the functionality and inspection capabilities of the rover.

Inertia Measurement Unit

The BNO055 IMU sensor is a powerful and versatile sensor, being able to provide information about the linear acceleration, angular velocity, magnetometer measurements and temperature of the rover in both raw and

processed forms. Currently, it is being used to acquire the angular position of the rover, assuring a precise value for its orientation. The sensor is connected with the Arduino board through I^2C protocol.

LiDAR

An important sensor in the operational framework of the rover is the TeraRanger Evo 60m Time-of-Flight distance sensor. The TeraRanger Evo 60m sensor is robust, versatile and accurate, providing to the Raspberry Pi to which it is connected the distance value of the obstacle or wall immediately in front of the rover, with a range that varies from 0.5 to 60 metres, yet also depending on conditions such as lighting and reflectiveness of materials. As explained in greater detail in Chapter 5, the use of the LiDAR sensor was changed, from being use to estimate the linear distance traversed by the rover to identify unknown obstacles on the path being followed by the rover and enable the transmission of a subsequent instruction from the Raspberry Pi to the Arduino board.

3.3 Software Overview

The control structure embedded in the rover is divided in two levels, the low-level headed by the Arduino board and the high-level headed by the Raspberry Pi. The PC, also part of the high-level section, interacts with and controls the Raspberry Pi, being also pertinent to include in the software overview. Figure 3.5 depicts a scheme of the hierarchy of the software in the control system of the rover.

3.3.1 Arduino

The Arduino routine is responsible for the direct activation and movement of the motors, in addition to acquire data from the MQ135 and BNO055

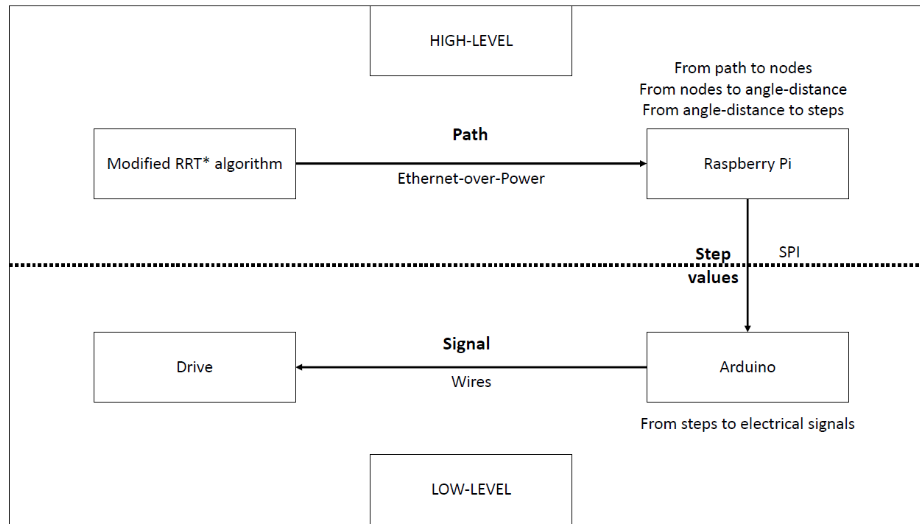

Figure 3.5: Software hierarchy. Extracted from [15].

sensors. The board, connected to the Raspberry Pi via SPI communication, receives messages which are concatenated in a buffer, that in turn is read and distributed in pertinent variables used in the movement routines. The messages received are read by interrupt routines, independent of whether or not the program is executing the movement routines at a given moment. This fact, for example, allows the movement to be halted at any moment by the transmission of the appropriate instruction.

3.3.2 Raspberry Pi

The Raspberry Pi autonomous routine essentially reads the list containing the points composing the trajectory computed and converts them into instructions, which are then sent to the Arduino to move the motors. Furthermore, the routine manages the camera and computes the orientation of the rover, whenever a rotation is necessary, and read the measurements from the TeraRanger Evo 60m and from the DHT22. For the manual mode, the main difference concerns the routines to obtain from the user the input instructions to be sent to the Arduino in order to move the rover, that is

the Raspberry Pi program acts as a mediator in this scenario between the Arduino controlling the motors and the user providing the commands.

3.3.3 PC

The motion planning routine presented in [15] is an adapted version of the RRT* path-planning algorithm. The RRT* algorithm is briefly explained in Chapter 4. The adaptation consists in the constraint approach developed to handle the presence of a tether on the rover. To do so, for every path found, successive points are evaluated with the purpose of investigating if any of the known obstacles is surrounded by the trajectory in such a way that the sum of the relative angles of the points is greater than 180°. Figure 3.6 shows an illustration of the idea behind the angle computation and evaluation.

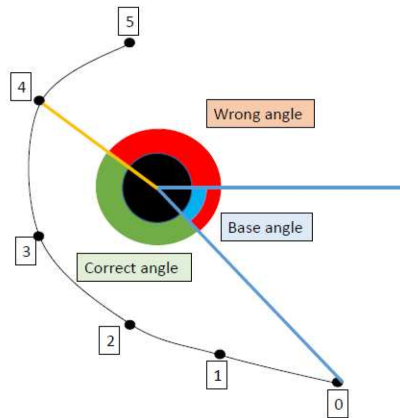

Figure 3.6: Illustration of the angle evaluation. Extracted from [15].

Whenever a violation is found, the first point to characterise the violation is transformed into an auxiliary obstacle and the path is recomputed. Figure 3.7 presents the flowchart of the adapted RRT* algorithm, where the above-mentioned processed can be observed. The main disadvantage of such approach is briefly addressed in the introduction of Chapter 4, while a concise review of the dynamic environment problem is made. Finally, the high

number of angle computations and comparisons and the fact that the algorithm may be subjected to several executions prior to achieving a feasible result constraint-wise render the algorithm considerably slow, with execution times of 5 minutes on the PC and 15 minutes on the Raspberry Pi for a point-to-point trajectory. Such times are impractical, especially considering a dynamic environment that may possibly require multiple executions during the mission, *i.e.* after the rover is already deployed.

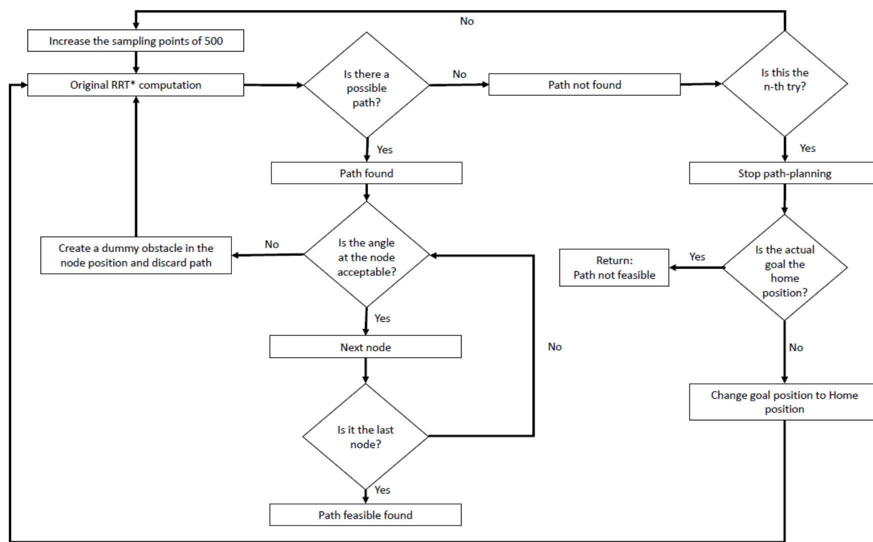

Figure 3.7: Flowchart of the adapted RRT algorithm. Extracted from [15].*

3.4 Results

Concerning the results of the development of the rover, [15] highlights the construction of a rover that meets the requirements of the project, with functional integrated system, data acquisition and two modes of operation, manual and autonomous, as well as a path-planning algorithm equipped to handle the constraints imposed by the tether. It is mentioned, however, that the rover exhibits a slight misplacement in respect to the expected position when performing tests with the autonomous mode, and also that

the cable interferes with the proper rotation movement of the rover. These comments later on play a significant role in the test phase of the present work, as observed in Chapter 5. More importantly, it was verified that the subfloor gap contains unexpected obstacles, as can be observed from Figure 3.8, that due to not being considered in the initial motion planning approach gravely hinder the autonomous operation of the rover. This finding was the main drive for the research and development of the motion planner algorithm capable of handling unexpected obstacles and suited for the rover here presented.

Figure 3.8: Picture taken by the rover during the inspection test. High humidity in the subfloor and unexpected obstacles can be observed. Adapted from [15].

Chapter 4

Path-Planning Program Development

As stated in Chapter 2, the motion planning problem of the rover can be summarised as a dynamic path-planning problem with constraints related to the path. The vast majority of existing motion planning algorithms were developed to tackle static environments, that is, environments where the map used as the base for the path-planning does not change along the execution of the designated path, whichever algorithm was used to compute it, or in other words, all the obstacles are static. Although not every algorithm is prepared to deal with dynamic environments, given that static environments are simply easier to manage planning-wise being therefore the stepping stone for the development of the motion planning field of work, real environments are rarely static [30].

Environments with possible unknown existing obstacles, such as the real environment of this project, can also be considered as dynamic environments. In other words, an obstacle that is not present in the map and is suddenly identified by the sensors of the mobile robot and inserted in the map as an obstacle is similar in behaviour to a previously existing dynamic obstacle, especially to one with random movement. In this context,

a path-planning algorithm must be found such as to treat the dynamic environment scenario but also considering the constraints introduced by the tether, as previously described in Chapter 2.

However, motion planning for a tethered planar mobile robot is a highly not-trivial problem, as states [19], even for static environments, at least on an optimal level. The combination of the two non-trivial problems, still undergoing research and development, *i.e.* the search for optimal or approximately optimal dynamic motion planning algorithms and the search for algorithms able to deal with constraints like the ones imposed by tethers in an optimal way, results in a challenging situation, especially optimality-wise. The search for optimality can end in a somewhat "short blanket" situation, where either optimality or approximated optimality can be achieved for the path-planning situation or for the constraint situation, but not simultaneously for both.

The course of action initially chosen and later followed in the present thesis was to first tackle the dynamic planning problem, and to further adapt the algorithm to somehow manage the tether constraint on a satisfactory manner, operation and computational-wise. Considering the worst case scenario, where such adaptation does not meet a computational adequate solution, a solution as the one applied to the previously developed static algorithm, described in Chapter 3, can be applied even though it is not optimal [30], that is, for every path found that violates the constraints, the motion planning algorithm must be executed again, but with an altered map containing inserted auxiliary obstacles.

4.1 RRT^X Path-Planning Dynamic Algorithm

According to [31], motion planning methods can be categorised into five main different types, and none of these types is able to manage every existing motion planning problem. The motion planners in [31] are categorised as

complete methods, grid methods, sampling methods, virtual potential fields methods and nonlinear optimisation methods. As summarised by [32], some of the best path-planning algorithms can be classified in two categories, both also present in the categorisation proposed by [31], which are optimisation techniques and sampling methods.

Two methods were initially considered as candidates for the dynamic path-planning algorithm based on their main features and scope of action: the CHOMP [26], an optimisation method that would possibly require adaptations, and the RRT^X [27, 33]. Upon a comparison of their advantages, the RRT^X was chosen as the algorithm to be implemented and further on modified to handle the constraints of the rover. It is worth noting that the RRT^X method falls in the sampling methods category. A sampling method was chosen based on the already vast utilisation of such methods for motion planning in mobile robots, as RRT and RRT* methods serve as a solid reference for satisfactory asymptotically optimal results.

Also it is worth to remark that, as presented in chapter 3, the previous path-planning algorithm present in the rover was a modified version of the RRT*, so choosing another sampling method would provide not only a continuity to the development of the motion planning algorithm of the rover, but also render feasible a more straightforward comparison between the two algorithms performance. Furthermore, regarding the advantages of RRT^X in respect to other sampling method algorithms, the RRT^X as presented in [27] has a similar computational efficiency as methods suited to handle static environments while still being able to handle highly dynamic environments.

4.1.1 RRT^X Main Characteristics

The very core of the RRT^X path-planning method is naturally the same as the RRT, the basic rapid-exploring random tree method. Briefly summarising, the RRT method builds a tree composed by nodes, each one be-

ing a configuration $q \sim (x, y, \theta)$ of the robot to which the motion is being planned. Customarily, the starting node, *i.e.* the root of the tree, is the goal node. The consecutive nodes are created by randomly sampling configuration points q_{rand} on the already known map and taking as new configuration points q_{new} to become nodes in the tree, those within a determined expand distance d_{exp} between the nearest node in the tree q_{near} and the random point q_{rand}.

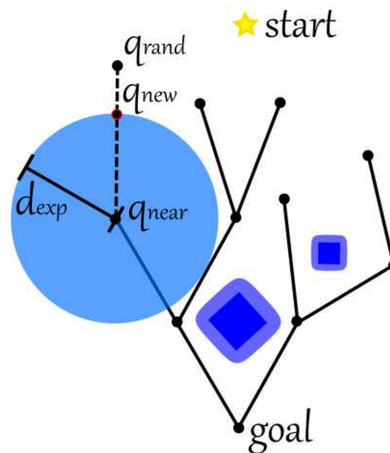

Figure 4.1: Rapidly-exploring Random Tree operation.

If the new node q_{new} is not located within and its path does not cross a static obstacle, then it is finally attached to the tree and a new random configuration is assigned. When a fixed number of iterations, a fixed time or when the tree reaches the starting point of the robot, the algorithm stops and the least costly path is chosen [34]. The follow-up method, the RRT* [35], rewires the tree during the execution of the algorithm every time there is shortest path between two nodes, hence avoiding loops and fostering less costly paths [31].

The RRT^X method shares the same basic operation as the RRT and RRT*, but distinguishes itself due to its ability to handle dynamic environments by erasing the branches of its tree that are affected by the presence

of a dynamic obstacle, being therefore a dynamic path-planning algorithm. Moreover, with other sampling methods of the rapidly-exploring random tree family of methods, the RRT$^{\#}$ [36, 37] and the SPRT [38], the RRT^X shares the idea of maintaining a consistent tree throughout the execution of the algorithm. According to [36], a tree is fully consistent if for every node v, the cost-to-goal $g(v)$ is equal to the look-ahead estimate of the cost-to-goal, defined as $lmc(v)$. A fully consistent tree guarantees that the topology obtained for the tree after the algorithm execution contains the best cost-to-goal for each and every node.

Instead of using the same metric of full consistency for the trees, the RRT^X builds trees that are ϵ-consistent, *i.e.* the consistency of the tree is kept under an arbitrarily defined value ϵ. The formal definition of ϵ-consistent can be observed in the inequality 4.1.

$$g(v) - lmc(v) < \epsilon \tag{4.1}$$

Where the look-ahead cost-to-goal for a given node v, $lmc(v)$, is defined as:

$$lmc(v) = min_{u \in N^+(v)} d_\pi(v,u) + lmc(u) \tag{4.2}$$

Being u a node belonging to $N^+(v)$, the set of outgoing neighbour nodes of the node v, and $d_\pi(v,u)$ the length of the feasible trajectory $\pi(v,u)$, which in the present case is the Euclidean distance between nodes v and u, given the differential dynamics of the rover.

Furthermore, concerning the computational cost of the RRT^X method, its most remarkable advantage lies in the fact that the RRT^X has an amortised computational cost equal to the amortised computational cost of the RRT and RRT* methods, *i.e.* $\Theta(\log n)$ with n being the number of nodes in the tree. The amortised analysis consists of computing the average of the running time per operation of a sequence of operations, considering their

respective worst-case [39]. Such analysis is particularly recommended for algorithms that have expensive operations with rare occurrences that somehow limit in number or cost the expensive operations that may occur in the future [40], which is a precise description of the RRT^X algorithm.

According to [27], two strategies related to the rewiring cascades present in the algorithm are responsible for the amortised cost of $\Theta(\log n)$, faster than the RRT$^\#$ method, with $\Theta(\log^2 n)$. Briefly summarising, the two strategies comprehend stopping rewiring cascades once ϵ-consistency is attained and managing node connectivity information in a more efficient manner. These approaches reduce the RRT^X iteration time when compared with RRT$^\#$ and the cost propagation when compared with RRT*, while asymptotic convergence to the optimal solution is maintained.

4.1.2 RRT^X Program Development

Although there was a version of the RRT^X algorithm available in the programming language Julia [41], the available computer program was considerably outdated and hence, not operational, demanding thorough review of it and replacement of all the outdated syntax as well as deprecated functions. Using an outdated version instead of the current Julia version available on the other hand could lead to future problems and an inescapable need of rewriting and adapting the program to an updated version later on.

As a consequence it was decided to write an operating routine of the algorithm taking as base the algorithm depicted in [27]. The programming language of choice was Python [42], due to it being well-established in the scientific community as well as in the industry and more importantly, due to Python being the native language of the Raspbian OS running in the Raspberry Pi embedded in the rover, whose integration with the other systems of the rover, *i.e.* the Arduino board and the TeraRanger LiDAR sensor, was already built and operational. Changing the motion planning algorithm

to another programming language would therefore imply in adaptations to integrate both the motion planning and the routine being used to communicate with the Arduino board, responsible for moving the motors of the rover.

The version of RRT^X written on Python for this thesis used as structural basis the RRT and RRT* routines provided by [43] and available as part of the open source software project PythonRobotics. The use of such routines provided a solid operational base upon which the RRT^X algorithm was written as a computer program itself, with the adaptations of the common basic routines present in the RRT and RRT* programs, such as steering and rewiring, necessary to fit the procedure of the algorithm, and the inclusion of the many routines that compose the characteristic operation of the RRT^X, such as the look-ahead cost-to-goal and ϵ-consistency related routines.

The same RRT* provided by [43] was used with the purpose of the performance comparison with the finished version of RRT^X, as presented afterwards in this chapter. Finally, an example of an execution of the algorithm with the respective tree computed can be observed in Figure 4.2.

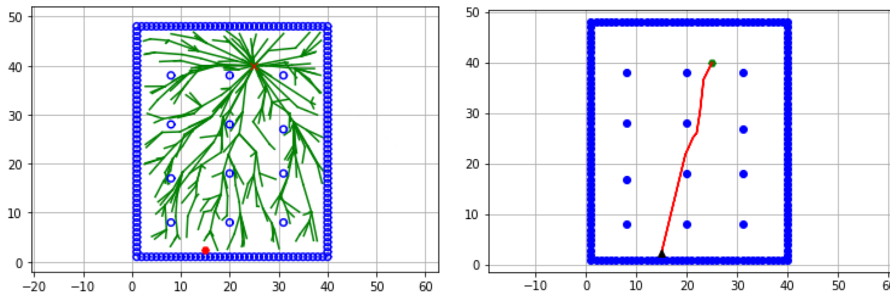

Figure 4.2: Example of a RRT^X execution with starting point at $(15, 2.5)$ *and goal point at* $(25, 40)$.

Problems Encountered

During the development of the RRT^X routine a few problems emerged. Initially, there were problems with the spanning of the tree generated by the algorithm, solved with adjustments in the parameters used and in the rewiring and steering functions. The second remarkable problem to emerge concerns the branch deletion once a new obstacle is identified, *i.e.* concerning the disconnection and removal of all the nodes affected by the new obstacle inserted in the map, characterising the dynamic environment.

The branch deletion routines were apparently generating orphan nodes not detected and deleted by those routines. By orphan node, it must be understood any node whose connection with its parent node was severed by the obstacle introduced in the map and the child nodes of the referred orphan node, which will become orphan nodes themselves once their parent node is deleted for being an orphan node. Through investigation of such occurrences, which were happening frequently but not on every execution, it was verified the existence of ghost node parents, that is nodes that were parents of other nodes, but did not figure in the node list of the tree generated. Such where the nodes serving as roots for the non deleted orphan branches.

Figure 4.3 depicts an example of the aforementioned problem. Static obstacles are presented as blue circles, nodes as blue dots, the dynamic obstacle as a red circle and the orphan branch marked with red rectangle. After the introduction of the obstacle in red, all the branches affected by the obstacle should have been erased, but as evident in the figure, all the orphan nodes were properly erased except for the seven orphan nodes inside the red rectangle. These nodes are all child nodes of a ghost node parent.

To solve the problem, the deletion routines were rewritten and changes were made to reinforce the erasure and removal of every node considered orphan and their respective child nodes from the node list. Apart from that,

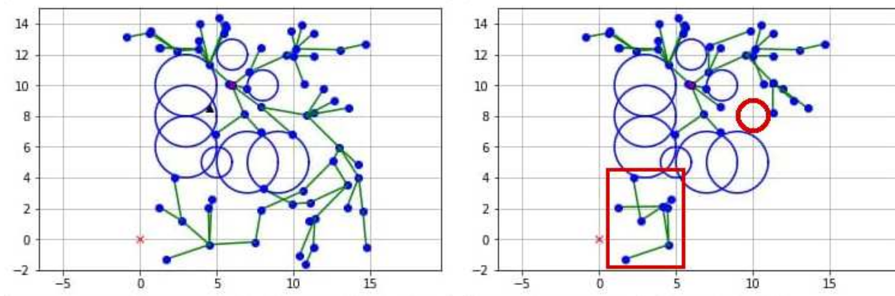

Figure 4.3: Example of a wrong branch deletion.

a routine was created to run through the node list, identify ghost nodes and fix them by reassigning them to the correspondent node in the list, and thus reinforcing parent-child relationship. This routine was also later on used to reassure consistent proper relationships when the tree is reloaded.

The cause for the existence of ghost nodes was not completely tracked down, remaining as possible reason for the problem irregularities when handling the nodes and node addresses by Python. Through the relationship check routine, it was also possible to perceive that such irregularities indeed happen, although very rarely, even when the program is running for a static environment, rendering necessary the scan and subsequent fixing of the node list. However, it is not possible to run the relationship check routine every iteration, given that it is computational costly to the algorithm, as shown in the performance comparison section, causing it to have an exponentially increasing computational time.

4.2 Constraint Problem

As presented in chapter 3, the constraint problem imposed by the presence of a tether on the rover and by a map with columns and random obstacles was solved in [15] through the computation of the rotation angle of the obtained paths around the known obstacles present in the map. Upon a rotation angle greater than 180° around a known obstacle, an auxiliary obstacle would be

placed in a node position in the vicinity of the known obstacle, in such a way that the recomputed path would not encircle the obstacle again. Albeit functional, this approach had a high computational toll, remarkably due to the extensive angle computations made and due to the fact that each time a path was unfeasible, the whole tree was being recomputed, which as already mentioned in the present chapter and stated by [30], is a non-optimal way of dealing with dynamic environments.

In this context, the solution found and proposed in the present work is to approach the constraint problem from a geometrical perspective. The main idea of the geometrical approach starts from the condition that the rover will start and end its path at the same point, that is the entrance point from which the rover will access the environment to which the rover was designed to inspect, in this case the house subfloors in Amatrice, is the same as the exit point. Such condition is naturally aligned with the purpose and design of the rover, given that any tethered inspection vehicle is expected to return to its base point, *i.e.* the origin of the tether, after its operation. From this condition, it is trivial to verify that any operation, feasible or not, will be a closed path.

The closed path formed may then as well be seen as a polygon, which in turn can be defined as a set of straight line segments that together bound a closed plane figure [44]. The obtained polygon can be used to identify whether or not there is a obstacle encircled by the trajectory of the rover, and consequently, by its tether. That derives from the fact that whenever the trajectory of the rover encircles an obstacle, *i.e.* an unfeasible trajectory, be it locally or considering the full closed path, the coordinates of the problematic obstacle will be invariably contained inside the boundaries of the obtained polygon.

The main idea of the geometric approach can be then summarised as follows: a feasible path is any path whose polygon obtained from the closed

trajectory of the rover has no obstacles within its boundaries.

During the conception of the geometrical approach, the existence of self-intersecting polygons was overlooked. However, during the implementation phase, the problems brought by these particular kind of polygons emerged, demanding adaptions on the written routine so the working principle of the geometrical approach was maintained. Figure 4.4 shows a rough visual sketch of the geometric approach idea, that is for the trajectory in red to be feasible, the chequered area must not include any of the black dots. The trajectory and polygon is represented in 4.4 by the line in red, the goals by the squares in blue, the columns of the building, that is the known obstacles, by the black dots, the area inside the polygon by the chequered pattern and the initial and final point by the green triangle.

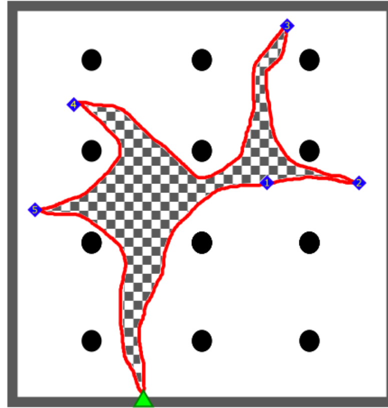

Figure 4.4: Sketch of the geometric approach for the constraint problem.

4.2.1 Geometric Approach Implementation

The implementation of the geometric approach lies heavily on the use of the Python package Shapely [45], developed to manipulate and analyse planar geometric objects regardless of data formats or coordinate systems. From a set of ordered points it is possible through Shapely to create a geometric object of the class polygon. With the polygon obtained, all that remains is

to perform the spatial analysis, namely the point-in-polygon query, of the object.

To solve the well-known and studied point-in-polygon computational problem, addressed for example in [46, 47], Shapely makes use of binary predicates, *i.e.* functions that point whether a condition is *true* or *false*, to assess topological relationships between geometrical objects. In the present case, the geometric objects are the polygon representing the trajectory and the points representing the obstacles, and the relationship of belonging being assessed is expressed by the binary predicate *within*, provided by Shapely. Alternatively, the binary predicate *contains* could have been used, in that case evaluating whether or not the polygon contains the points instead of whether the point is within the boundaries of the polygon [48].

It is important to remark that, although not trivial, the point-in-polygon query performed by Shapely is highly computationally efficient, later verified for example by the computation of hundreds of queries in far less than a second during the executions of the RRT^X algorithm with the developed constraint approach. The feasibility of the geometric approach depended heavily on how efficient the point-in-polygon query, henceforward referred in the present work as polygon check, was handled by Python using the Shapely package. In that context, the computational feasibility requirement of the geometric approach was satisfied.

With the polygon check operational, the rest of the constraint approach was built around it. In other words, by having a reliable way to separate feasible and unfeasible trajectories, it was possible to built routines such that a feasible solution is typically achieved, *i.e.* impossibility of finding a feasible trajectory for a determined set of goal points happens scarcely, and no unfeasible trajectory is ever provided as a result of the program. Such routines, explained in more detail subsequently, make use of the tree created and utilised to find the route connecting the starting point and the

goal point, and of the ability of the tree of cutting its own branches and re-spanning again, provided by the RRT^X algorithm having in mind dynamic environments.

The first characteristic, the tree structure, is used to obtain more possible routes that connect starting and goal points. The RRT^X algorithm, as do the other sampling method algorithms, returns the most cost efficient path connecting starting and goal points considering the randomly built tree obtained in a particular execution of the algorithm, however by no means such path is the only trajectory that connects the two points. With a properly spanned tree, which is deeply related to the parameters chosen for the algorithm execution, there is a suitable number of alternative not necessarily cost efficient possible paths that can be used, without having to re-span the tree, to experiment new path arrangements in search of compositions that form feasible trajectories according to the polygon check. The idea is represented graphically by the Figure 4.5.

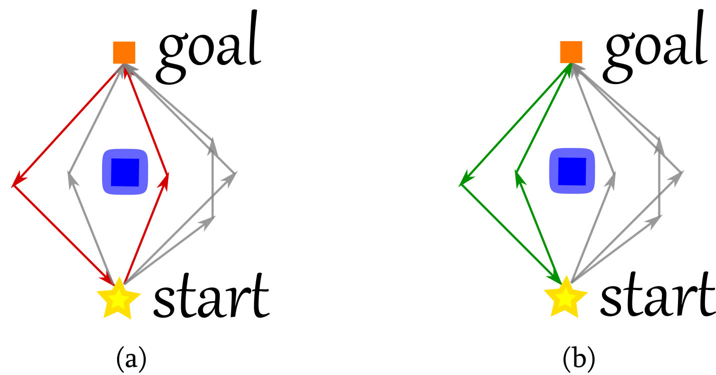

Figure 4.5: Example of the alternate path implementation. (a) Unfeasible trajectory in red with alternative possible paths in grey. (b) Feasible trajectory in green using a suitable alternative possible path, even though it is not the most cost effective one.

The second characteristic, the ability to re-span the tree, is used when no arrangement with the available possible paths produces a feasible trajectory.

In that scenario, the course of action taken was to place a virtual obstacle on top of the nearest node to the problematic obstacle, that being the obstacle inside the polygon. The RRT^X therefore interprets the virtual obstacle as a new obstacle and following its dynamic environment approach, erases the branches severed by the new obstacle, re-spanning the tree around the obstacle. Nevertheless, the obstacle does not exist in reality being only an auxiliary resource for the purpose of the constraint approach, leading to it being erased right after the execution of the tree re-spanning. As a result, a new path to the goal is found, as well as new possible paths to be tested in case the first keeps encircling the obstacle and rendering the path unfeasible. Regarding the developed algorithm for the virtual obstacle placement, it is worth to remark that it cannot be place atop or considerably close to starting or goal nodes, otherwise the tree will not be able to span from the starting point or to reach and connect with the goal point. The idea of the tree re-span implementation can be seen in Figure 4.6.

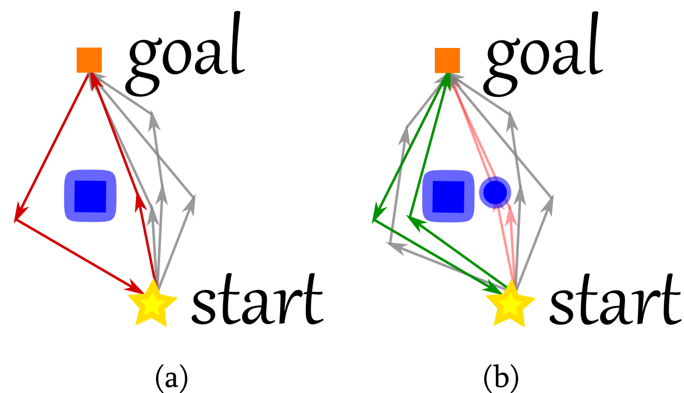

Figure 4.6: Example of the tree re-span implementation. (a) Unfeasible trajectory in red with alternative possible paths in grey. (b) Feasible trajectory in green using a suitable path computed after the placement of the virtual obstacle atop the closest node from the problematic obstacle, the blue square.

The foregoing processes were organised and implemented in the form of

an integrated algorithm, which is presented in pseudocode in Algorithm 1.

As it can be seen, the constraint approach is composed by five main processes. They are the identification of the problematic sub-paths, based on the already identified problematic obstacles, the search for alternative paths that eliminate one or more of the problematic obstacles from inside the polygon, re-spanning of the tree, alternative path search in reverse order of the problematic paths and ultimately in shuffled order. The maximum number of attempts and condition for re-spanning the tree can be adjusted arbitrarily and, for all the results presented henceforth, a maximum number of two attempts and the condition of only re-spanning the tree in the first attempt were chosen, since they presented a good compromise between performance and execution time.

In Algorithm 2 the implementation of the possible paths solution briefly described previously and displayed in Figure 4.5 is shown.

Since more that one feasible alternative sub-path can be found, the whole list of possible paths is investigated and the one with the lower simple cost is taken, except when one of the possible paths attempted results in the whole trajectory being feasible, corresponding to the third row of the algorithm, to which the constraint approach is terminated and the trajectory found registered, for the sake of time efficiency of the algorithm.

Algorithm 3 depicts the second solution method where virtual obstacles, or V_{obs} in the algorithm pseudocode, are used to guide the tree to a desirable region, as can also be observed in Figure 4.6.

The algorithm loads the old tree for the specific problematic sub-path being examined and re-spans its tree with the virtual obstacles in place, attaining a new sub-path and new possible sub-paths to be tested, if the new sub-path is still inadequate.

Once the testing of the performance of the constraint approach started, it was verified that the process of generating the polygon, that is the geometric

Algorithm 1: Constraint Approach

```
1  while attempt < max_number do
2      foreach sub_path ∈ path_list do
3          Obs_local, V_obs ← Id_Problem_Paths(sub_path);
4          if Possible_Path_Sol(possible_paths) ≠ True then
5              prob_path ← prob_path ∪ {sub_path};
6      foreach sub_path ∈ prob_path do
7          if Possible_Path_Sol(possible_paths) = True then
8              prob_path ← prob_path\{sub_path};
9          if prob_path = ∅ then
10             return path_list;
11     if first attempt then
12         foreach sub_path ∈ prob_path do
13             path_list ← ReSpanningTree(sub_path, tree, V_obs);
14             problem_obs ← PolyCheck(path_list, Obs_coord);
15             if problem_obs = ∅ then
16                 return path_list;
17             if Possible_Path_Sol(possible_paths) = True then
18                 prob_path ← prob_path\{sub_path};
19     foreach sub_path ∈ reverse(prob_path) do
20         if Possible_Path_Sol(possible_paths) = True then
21             prob_path ← prob_path\{sub_path};
22         if prob_path = ∅ then
23             return path_list;
24     foreach sub_path ∈ shuffle(prob_path) do
25         if Possible_Path_Sol(possible_paths) = True then
26             prob_path ← prob_path\{sub_path};
27         if prob_path = ∅ then
28             return path_list;
29 return path_list;
```

Algorithm 2: Possible Paths Solution

```
1  foreach possible_path ∈ possible_path_list do
2      problem_obs ← PolyCheck(possible_path, Obs_coord);
3      if problem_obs = ∅ then
4          path_list ← (path_list\{sub_path}) ∪ {possible_path};
5          return True;
6      if Obs_local = ∅ then
7          cost ← get_cost(possible_path);
8          if cost < min_cost then
9              min_cost ← cost;
10             aux_path = possible_path;
11 if aux_path ≠ ∅ then
12     path_list ← (path_list\{sub_path}) ∪ {aux_path};
13     return True;
14 return False;
```

Algorithm 3: Re-Spanning Tree

```
1 path_new, node_list, possible_paths ← RePlan(goal_list, tree, V_obs);
2 path_list ← (path_list\{sub_path}) ∪ {path_new};
3 pp_list ← (pp_list\{possible_paths_old}) ∪ possible_paths;
4 trees ← (trees\{tree_old}) ∪ node_list;
5 Obs_local ← Id_Problem_Paths(sub_path);
6 return path_list, possible_paths_list;
```

object corresponding to a polygon for Shapely, was faulty. More specifically, the polygon was created in such a way that depending on the trajectory used to generate it, there could be holes inside of the polygons and, if there was an obstacle inside these holes, they would not be recognised as problematic ones. That is, due to these unwanted holes the constraint approach is prone to result in false positive cases. Such generated faulty polygons are classified as self-intersecting polygons [44].

Self-Intersecting Polygons

As the name indicates, self-intersecting polygons compose a category of polygons whose edges intersect each-other at least once. As previously exposed, depending on the number and form of the intersections, non-filled areas, or simply putting holes, may be present inside the confines of the polygon, as can be observed for example in the Figure 4.7. Polygons such as the one in the figure lead to false positive results whenever an obstacle is located inside one of these non-filled areas, like the non-coloured area in the centre of the triangle in Figure 4.7.

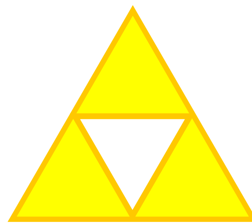

Figure 4.7: Trivial example of a self-intersecting polygon.

Upon brief investigation it was verified that considering the set of points used to build the polygon, independently of whether or not it would result in a self-intersecting polygon, it was necessary to obtain its respective concave hull and with it proceed with the geometric approach. The concave hull comprehends the external hull characterised by a set of points, which in opposition to the more common convex hull, may contain internal angles

superior to 180°. The use of the concave hull as the polygon utilised in the geometric approach instead of merely the polygon built by the edges composing the computed trajectory would eliminate every non-filled area inside the polygon, if the polygon was a self-intersecting one and such area was present. An example of an occurrence of a false positive from a self-intersecting polygon built from a planned trajectory can be observed in Figure 4.8, where the area in cyan was the one considered as part of the polygon, or the filled area, by the geometric approach while the area in white containing the obstacle was interpreted as not being part of the polygon, *i.e.* the non-filled area.

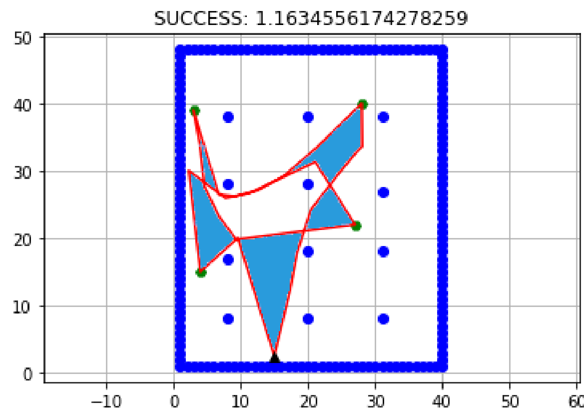

Figure 4.8: An example of an obtained self-intersecting polygon incurring in a false positive result from the geometric approach.

Initially, an attempt was made to apply the alpha-shape method [49] for creating a concave-hull to the set of points available, but the results were not satisfactory. The alpha-shape algorithms used were heavily dependent on the parameters used, especially the α parameter that names the method, and thus the fact that the same algorithm must be used for a vast variety of different polygons both in shape and size rendered it unusable, even when an algorithm to optimally define such α value was applied. The results would exclude or not consider points in such a way that the final polygon obtained

had a different hull if compared with the shape of the original trajectory, including nonexistent areas or erasing existing ones. Notwithstanding the undesired results achieved, the algorithm would take a considerable amount of time to execute, which, given the high number of polygon checks required for a single planned trajectory, would also preclude the use of alpha-shape for the geometric approach due to the computational time constraint.

Given the unreliability of the alpha-shape algorithms for the present application, a solution within the Shapely library was then investigated. Upon a series of successive operations of merger between the straight lines composing the trajectory, creation of a set of sub-polygons and finally merger of the sub-polygons into one single polygon, the intended result was ultimately achieved. The computational feasibility of the new process to construct the polygon was examined and, despite containing more steps and transitioning between different geometrical objects within the Shapely library, the new process presented no considerable toll in the computational time in respect to the previous faulty version. The working algorithm currently in use to built the polygon from the planned trajectory and with it perform the geometric approach can be accessed in Algorithm 4. Apart from the third and sixth rows, every function seen in the algorithm is part of the Shapely library and their formal definitions, parameters and examples can be verified in [45].

Figure 4.9 shows the aforementioned algorithm being applied to the same trajectory presented in Figure 4.8 and, as it can be seen, the concave-hull was successfully extracted from the trajectory, eliminating the possibility of false positives.

At last, the geometric approach for the constraint problem imposed by the tether presented itself as a feasible and suitable solution. The approach correctly identifies feasible and unfeasible paths and within an adequate computational time, does not impact expressively the overall time of the

Algorithm 4: Polygon Creation

```
1  lines ← MultiLineString(path_list);
2  line_unary_union ← unary_union(lines);
3  if type(line_unary_union) = 'LineString' then
4  |   line_list_merge ← line_unary_union;
5  else
6  |   line_list ← list(line_unary_union);
7  |   line_list_merge ← linemerge(line_list);
8  result ← polygonize_full(line_list_merge);
9  polygon ← unary_union(result);
10 return polygon;
```

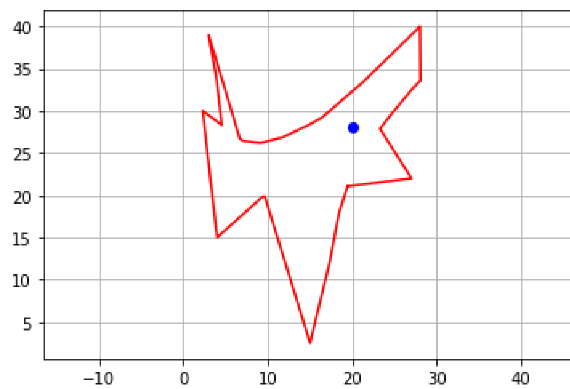

Figure 4.9: The new polygon, that is the concave-hull, of the trajectory shown in Figure 4.8.

whole motion planning algorithm.

4.3 Final Program Overview

In the current section, the final version of the motion planing routine with all the adaptations and additions made due to the constraint problem and to the integration between routines is presented with emphasis in such changes. The first remarkable modification was the addition of routines to allow the algorithm to work with multiple goals, essential for creating the closed trajectory and further perform the polygon check. In case only one goal is inserted, the routine computes the path to the goal and automatically mirrors the result to add the returning path. Moreover, since during the tests it was verified that repeated nodes in the path list introduce at best a delay in the operation, successive repeated nodes are currently simplified to just one node.

When more than one goal is initially set prior to the execution, the goals are entered in a list. The goals in the list are then sorted by their Euclidean distance in respect to the rover and each individual sub-path is computed, as well as their respective trees and possible paths stored. For the computation of the sub-paths, starting by the set starting node and the first goal in the goal list, every successive sub-path is characterised by the starting point being the last goal point used and the goal point being the consecutive goal point in the list. After the last goal in the list is processed, another sub-path is computed having as stating point the last goal in the list and as goal the original starting point of the rover. In doing so, considering a multi-goal situation, the trajectory is closed and the final position of the rover is its first position.

Thus, for a number n of goals, there will be precisely $n + 1$ sub-paths to be computed and further on processed by the geometric approach. An example of a multi-goal execution can be observed in Figure 4.10.

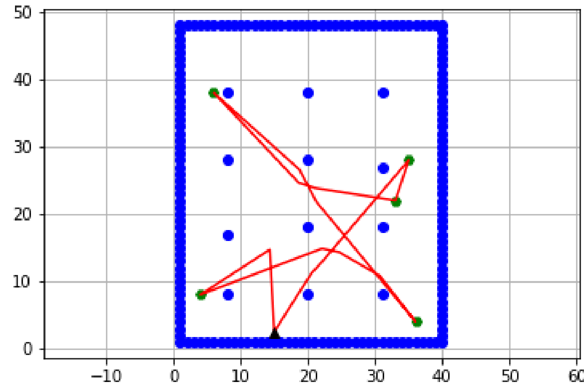

Figure 4.10: Example of a multi-goal execution with five goals and geometric approach to constraint problem.

In terms of program structure, a main routine is responsible for obtaining the goals, sorting them if necessary and calling the core routine, responsible for the planning and constraint approach. It is worth to remark that, in case the constraint approach fails to find a feasible path, a last attempt is made before the user is notified about the failure of the execution. During simulations, it was observed that most situations where no feasible path was found generally involved a high number of goals, usually horizontally distant from each-other. In this context, the final attempt, hereinafter referred as two-steps solution, consists in separating the goals horizontally, creating two sub-divisions of the map, here called west and east. The whole core routine is then re-applied, now on the two separated lists of goals generated. The main routine can be observed in Algorithm 5.

In the main routine, firstly, the goals are sorted and the core routine is executed. The resulting trajectory is then processed and, if a feasible path is not found, the map is divided into two sections, as it can be verified in the fourth row of Algorithm 5. The core routine is then called two times in succession, to compute the trajectory of the two closed trajectories created. In the fifteenth row the successive repeated nodes are simplified to just one,

Algorithm 5: Main Routine

```
1  goal_list ← sort(goal_list);
2  path_list, result ← Core(goal_list, obstacle_list);
3  if result = False then
4      West_goals, East_goals ← geo_sort(goal_list);
5      W_path_list, W_result ← Core(West_goal_list, obstacle_list);
6      E_path_list, E_result ← Core(East_goal_list, obstacle_list);
7      path_list ← W_path_list ∪ E_path_list;
8      result ← True (Two-Steps);
9      if W_result = False then
10         path_list ← E_path_list;
11     if E_result = False then
12         path_list ← W_path_list;
13     if W_result = False & E_result = False then
14         result ← False;
15 path_list ← remove_repeated_nodes(path_list);
16 create_txt_file(path_list);
17 return path_list, trees;
```

due to the issue previously discussed in the present chapter. It must be remarked that the result is only considered a failure, or a complete failure, if both west and east trajectories cannot be computed for determined list of goals, as it can be seen in rows 13 and 14.

As for the core routine, it can be observed in Algorithm 6.

Algorithm 6: Core Routine

```
1  if size(goal_list) = 1 then
2  |   back_path ← reverse(goal_list);
3  |   goal_list ← goal_list ∪ {back_path};
4  path, node_list, possible_paths ← Plan(goal_list, obstacle_list);
5  if path = ∅ then
6  |   Error: exit
7  problem_obs ← PolyCheck(path_list, Obs_coord);
8  if problem_obs ≠ ∅ then
9  |   path_list, prob_paths, problem_obs ← Constraint_Approach()
10 result ← True;
11 if problem_obs ≠ ∅ then
12 |   result ← False;
13 return path_list, result;
```

The core routine essentially executes the planning of all sub-paths and applies the constraint approach to the closed trajectory found. If one of the sub-paths is not feasible, *i.e.* a start-to-goal path cannot be found, the execution halts and the user is invited to review the selected goals and to try again. In case the first closed trajectory found already succeeds in the polygon check, that is it is feasible, naturally the constraint approach is not executed. Conversely when the constraint approach has to be performed, if after the execution there are still obstacles inside the formed polygon, in other words the search for a feasible trajectory failed, the algorithm outputs

a negative result, which as seen in Algorithm 5, is the trigger to initiate the two-steps solution. Otherwise, if the constraint approach is successful, the path and a positive result variable are shared with the main routine at the end of the algorithm.

Figure 4.11 shows an example were the feasible trajectory was obtained making use of the aforementioned final attempt. One list of goals comprehends all the nodes west of the 20 metres mark, or the centre column of obstacles, while the second list comprehends the remaining nodes to the east. Since the two now separated goal lists share the same starting point, from the operational point of view it is not an issue to separate the two trajectories and yet confine them in the same execution, or mission, of the rover. The result would be similar to two independent missions.

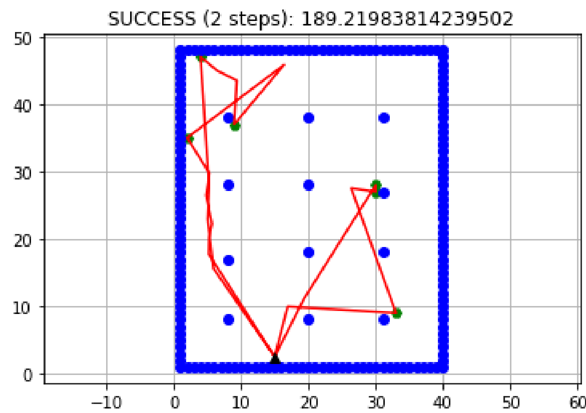

Figure 4.11: Example of a feasible trajectory obtained in two steps, i.e. using the map partition strategy.

In case after all the attempts no feasible trajectory is found, the user will be notified and advised to either try again or to choose different goals. Since the trees for every sub-path are random and the success of the geometric approach depends on finding a combination of sub-paths that does not encircle an obstacle, it can occur that a combination is not found, even though it might exist in the specific execution. Although the routines were written to

prioritise promising sub-paths and increase the number of combinations attempted, given the great number of possible paths per sub-path computed, to try every possible combination would be computational demanding and impractical. Among the additions to foster the look for a feasible combination, it is the consecutive reshuffling of the order of the sub-paths being examined, as it can be observed in Algorithm 1.

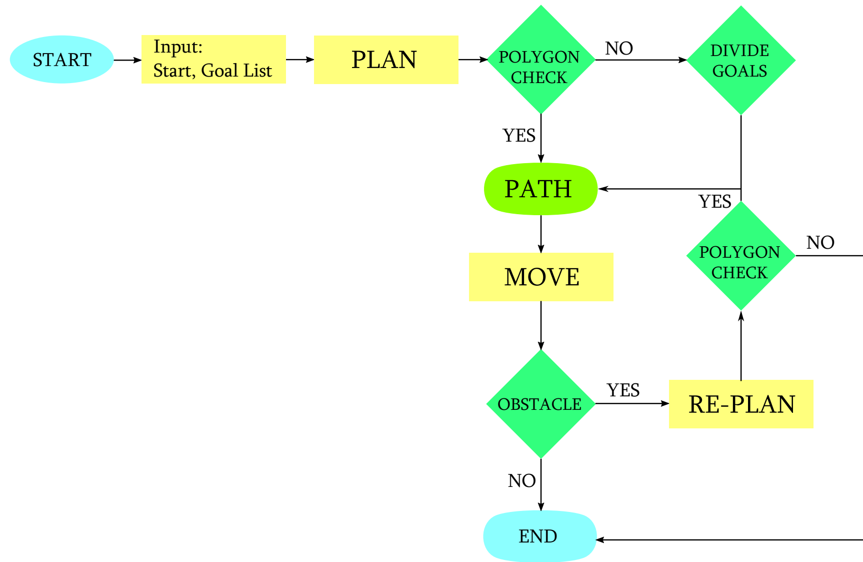

Figure 4.12: Simplified flowchart of the main and core routines.

4.4 Simulation Results and Performance Analysis

Following the development of the motion planning for dynamic environments with support to cable-imposed constraints, simulations were made to assess first and foremost the validity of the method, *i.e.* if the herein proposed method is effectively working for an adequate range of situations and, more importantly, if it is reliable. Subsequent tests evaluated the performance of the method as well as compared the RRT^X algorithm with the RRT* in terms also of performance for different scenarios.

Naturally, through the different phases of the development of the al-

gorithm and its routines, countless simulations were made to examine its execution and thus implement corrections and improvements. The major two examples of the aforesaid were described anteriorly in the present chapter, concerning the branch deletion problem and the self-intersecting polygons problem. The tests presented in the following sections, however, were performed using the up-to-date version of the program. Throughout the tests, especially the simulations, repeatability was prioritised in order to obtain robust data expressing the average behaviour of the method for a wide range of situations, and with such data, evaluate the performance and draft the possible limits of operation of the rover with respect to the planning.

4.4.1 Simulations

For the first test after the completion of the algorithm, complete trajectories for randomised goal points were computed. The number of goals initially ranged from one to five goals, or two to six sub-paths, later expanded to nine goals, or ten sub-paths, when it was verified that the program was able to handle such number without a drastic loss of its performance. As the number of goals increases, naturally the difficulty of the task increases accordingly, which was perceived through a higher number of two-steps solutions in comparison to the executions with less goals. From that, it can be taken that the limit for operation is not necessarily of nine goals, but the value was taken as the maximum for the present tests due to presenting a high density of points, considering the size of the map used, a considerable level of difficulty for the motion planning algorithm and due to characterising the round number of ten sub-paths in the trajectory.

More importantly, during the tests absolutely no case of false positive was verified, confirming that the corrected version of the geometric approach is indeed reliable. Regarding the map utilised, for comparison purposes the same map applied in [15] was also used for the tests, being it the map

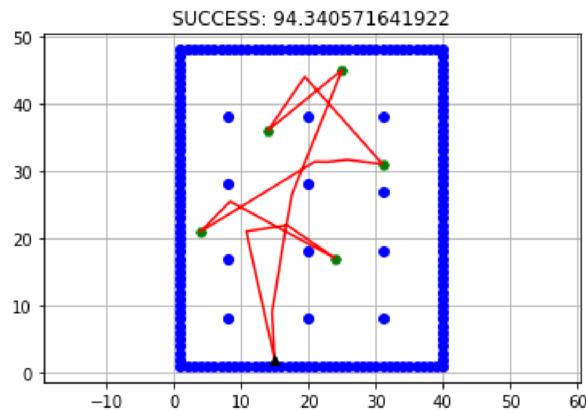

Figure 4.13: Example of a successful execution with five goals.

representing the subfloor of a house with dimensions 38×48 metres and 12 columns distributed in four rows of three columns each. The map can be observed in Figure 4.13, which displays an example of a simulation with a successful result. On the top of the figure it can be seen the status of the result as well as the time taken to compute the path in seconds, in this case 94.34 seconds approximately, or 1 minute and 34 seconds.

To better grasp the behaviour of the algorithm, successive tests were made with one, four and nine goals. In these simulations, the success rate, computational time and simple cost, *i.e.* the length the rover would traverse with the computed trajectory, were registered over 100 samples. The results can be observed in the following sections.

Single Goal Trajectory

The first batch of simulations was performed with a single goal, randomly positioned in the free area of the map. Since the return path from the goal position, that is from goal to start, is simply the mirrored points from start to goal, there is no need for the constraint approach and thus, such routines are not activated. Consequently, the average time of execution reflects the average time of execution per sub-path for the motion planning algorithm,

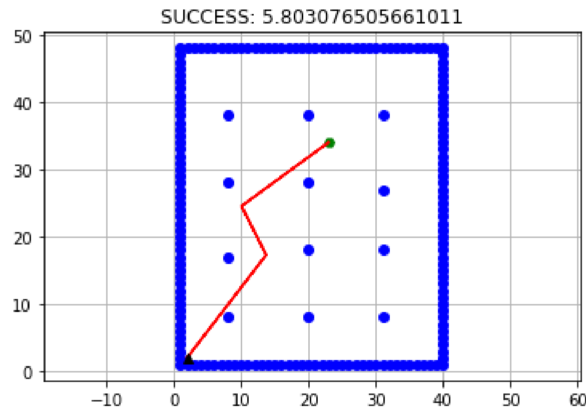

Figure 4.14: Single goal execution with random goal point.

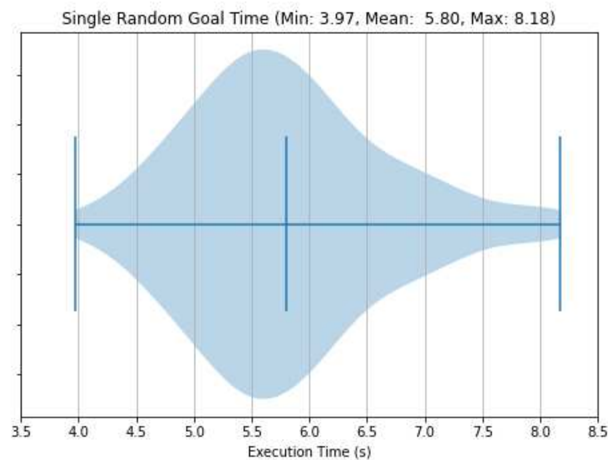

Figure 4.15: Violin plot of the time for 100 single random goal executions.

not considering the constraint approach. Figure 4.14 shows a single goal execution, with the goal being decided randomly.

As it can be seen in the violin plot of Figure 4.15, the average time of execution is approximately 5.8 seconds. As aforementioned, this corresponds to the general average time of a sub-path computation. From the violin plot in Figure 4.15, it can be seen that the time does not have a great variance, oscillating for this set of samples from 3.97 to 8.18 seconds.

The second group of single goal simulations were made with a fixed goal

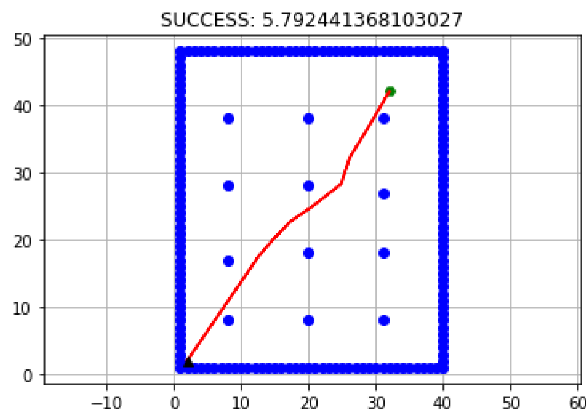

Figure 4.16: Single goal execution with goal point at $(32, 42)$.

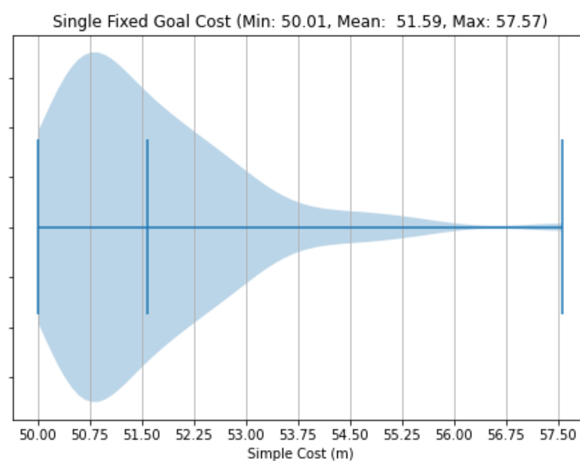

Figure 4.17: Violin plot of the simple cost for 100 single fixed goal executions.

located at $(32, 42)$, being 50 metres apart from the start point, again defined at $(2, 2)$. The average time of execution was approximately 6.04 seconds, therefore in accordance with the previous experiments.

The average simple cost of the path is approximately 51.59 metres, which shows that within a certain range of tolerance, the algorithm finds feasible trajectories close to the unfeasible optimal one, in this case a 50 metre straight line form the start point to the goal.

The Figure 4.17 displays the violin plot presenting the average value of

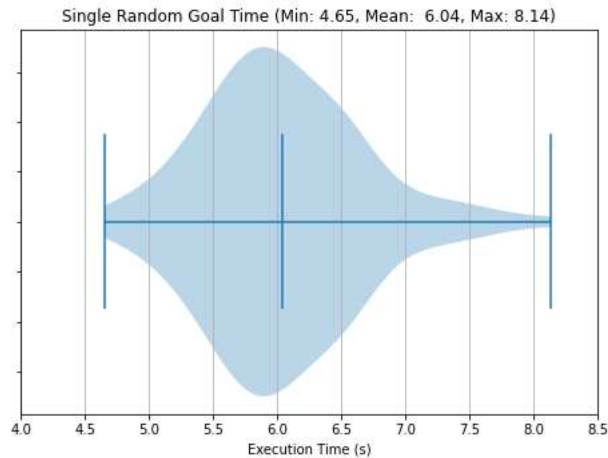

Figure 4.18: Violin plot of the time for 100 single fixed goal executions.

the cost and its distribution. Considering the minimum value of 50.01 and the maximum value of 57.57, the cost does not present a great variation for the single goal example and its distribution is higher near the minimum value, as expected from an approximately optimal algorithm. Figure 4.18 shows the violin plot for the execution time referent to the same samples. Compared to Figure 4.15, the behaviour presented in Figure 4.18 is consistently similar.

Four Goals Trajectory

From two goals on, the constraint approach routines start being applied. Therefore, the first set of tests here presented to consider the behaviour of the geometric approach developed is the simulation with four goals, or five sub-paths. First, four random goals were positioned in the free space of the map. In case a feasible path was not found, as described in more detail in the previous section, the map was equally divided in two partitions, and the sub-paths and geometric approach recomputed. Figure 4.19 depicts an example of an execution were a feasible path was found without the need of map partition.

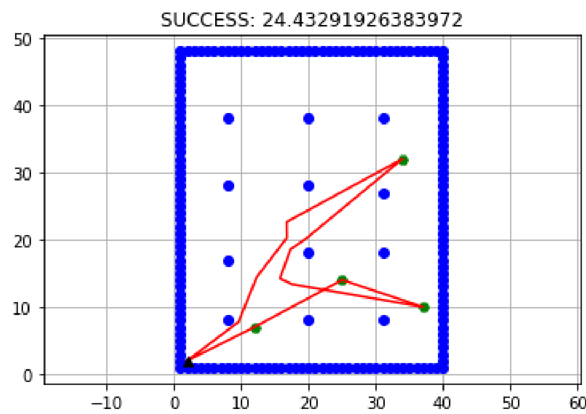

Figure 4.19: Four goals execution with random goal points.

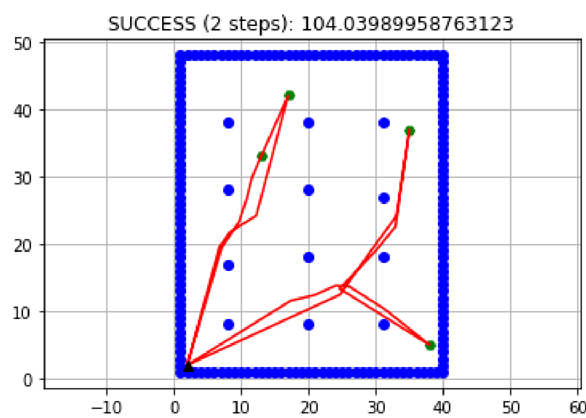

Figure 4.20: Four goals execution with random goal points and two-steps execution.

Conversely, Figure 4.20 presents an example were the map partition was necessary and ultimately guaranteed a feasible result. As stated before, this approach is herein referred as two-steps, given that it is characterised by the second and last attempt, or step, to find a feasible trajectory in the algorithm, as well as it produces two distinct feasible trajectories, notwithstanding within the same mission.

Regarding the success rate of the algorithm, as can be seen in Figure 4.21, 80% of the executions resulted in success without requiring the two-steps solution, 20% were successful making use of the two-steps solution and

Figure 4.21: Success rate of four goals execution with random goal points.

no execution resulted in failure, that is in no feasible path being found.

The violin plot of the execution time displays an interesting characteristic, which is the minimum value for the time equals approximately the number of sub-paths computed times the shortest times for a single goal execution. That is a reflex of the executions where a feasible path was either found with the need for the measures applied in the geometric approach, *i.e.* the first path found was already feasible, or a feasible path was quickly found amidst the available sub-paths and no further search or re-span of the tree was necessary. The aforementioned violin plot can be seen in Figure 4.22, where the minimum time computed was 24.11 seconds, the maximum 153.56 seconds and the average, 53.29 seconds.

The second group of simulations performed with four goals was characterised by four fixed goals instead of random ones, although the four goals used were selected randomly prior to the first execution. The goals utilised were located at $(4, 19)$, $(21, 2)$, $(17, 20)$ and $(33, 44)$, as can be seen in Figure 4.23 through an example of a successful execution.

The success rate can be observed in Figure 4.24, with 89% of success results and 11% of two-steps results. Such results strongly show the random characteristic of both the path-planning and the geometric approach, *i.e.*

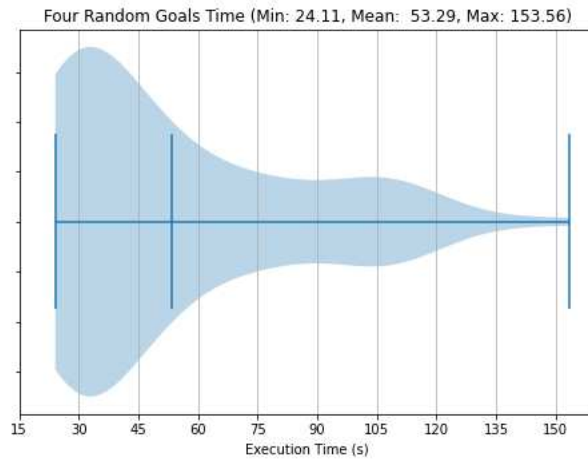

Figure 4.22: Violin plot of the time for 100 four random goals executions.

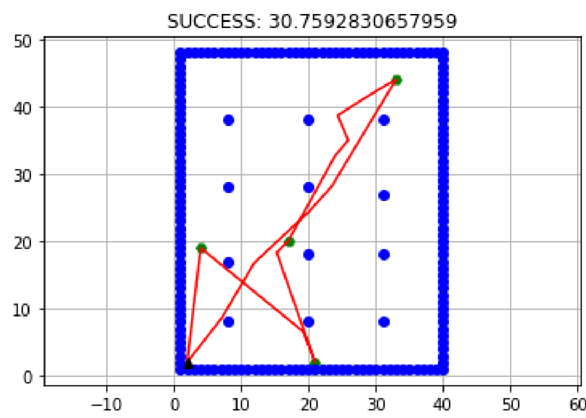

Figure 4.23: Four goals execution with fixed goal points.

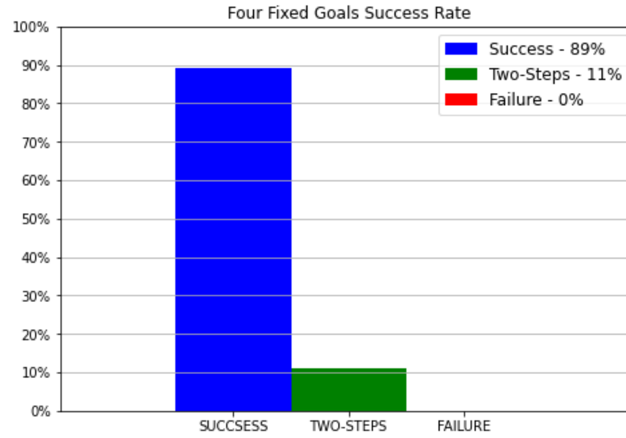

Figure 4.24: Success rate of four goals execution with fixed goal points.

for the same four goals 11% of the samples were not able to find a result on the first attempt of the algorithm, resorting to the two-steps one. Hence, whether the feasibility of the trajectory for a given number of goals can be attained or not does not necessarily depend, or rather it does not depend only, on the locations of the goals, even though their position indeed play a major role in determining the difficulty of the task imposed and thus influence in the success to two-steps ratio.

Although for the case of four goals it is not trivial to derive the optimal trajectory, even if it is unfeasible, the violin plot of Figure 4.25 is useful to stress once more the approximately optimal behaviour of the algorithm, as it can be observed by the higher distribution of results near the lower extremity of the plot, *i.e.* the minimum value registered.

Regarding the execution time, Figure 4.26 shows that the results for this configuration of goals heavily tend to shorter times of execution. The success rate of the samples and the distribution of results presented in the plot of Figure 4.26 corroborate to the perception that the goals picked configure a low difficult task, for example.

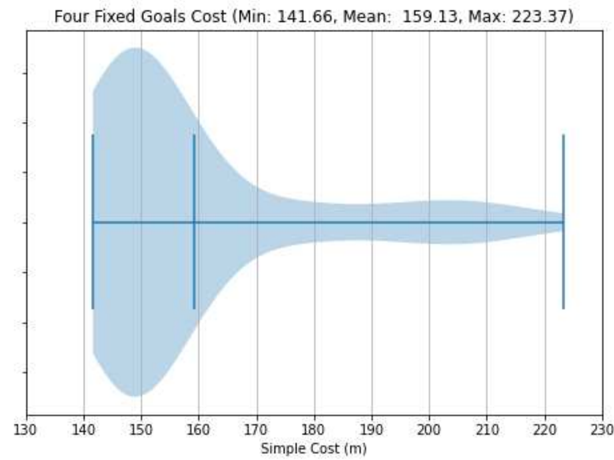

Figure 4.25: Violin plot of the simple cost for 100 four fixed goals executions.

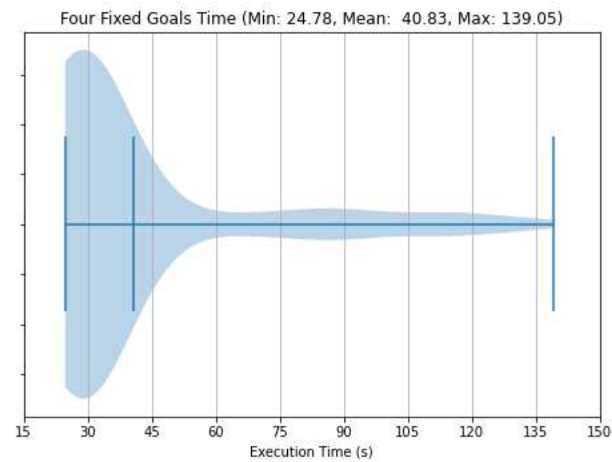

Figure 4.26: Violin plot of the time for 100 four fixed goals executions.

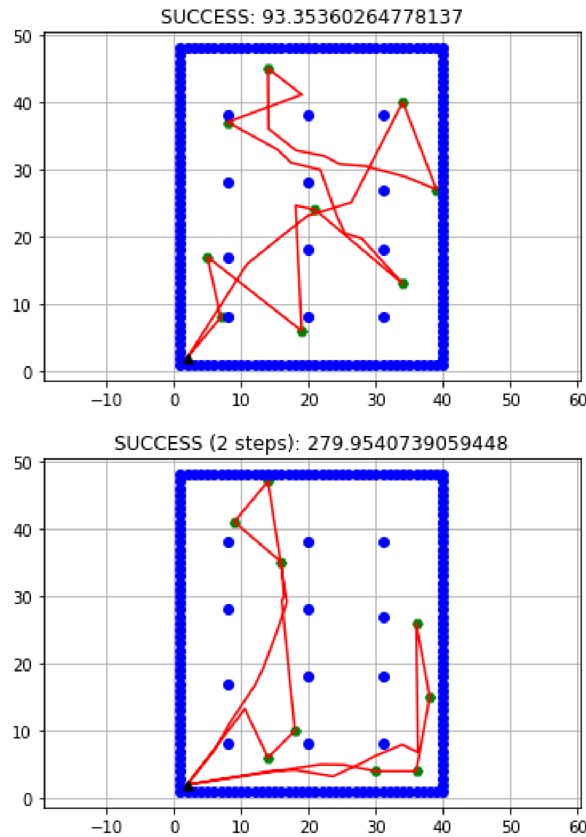

Figure 4.27: Nine goals executions with random goal points.

Nine Goals Trajectory

The last test group to be simulated was characterised by nine goals, defined randomly in the first set of 100 test and fixed in the second set. Computing ten sub-paths is notably more demanding for the geometric approach, since a greater area is generally covered, more obstacles are prone to initially be located inside the polygon area and consequently more available sub-paths must be analysed and matched and, in case no feasible path is found, be re-planned according to the strategy presented in this chapter. Figure 4.27 show respectively an example of an execution where the feasible trajectory was found in the first attempt and an example with the two-steps approach, both for randomly defined goals.

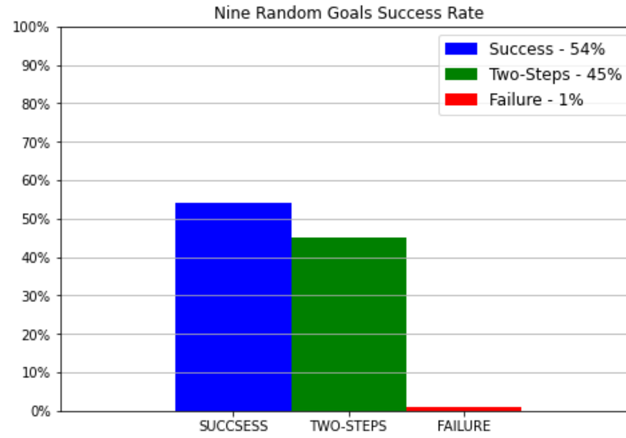

Figure 4.28: Success rate of nine goals execution with random goal points.

As it can be observed from the bar graph in Figure 4.28, the success rate, 54%, and two-steps solution rate, 45%, are more balanced, due to the higher complexity of the tasks. Moreover, differently from the previous simulations here presented, it was verified a 1% rate of failure, which means that no feasible path was found, after both attempts.

Regarding failures, two remarks are significant. The first concerning the two-steps solution, that can naturally result in one of the sub-trajectories feasible and another unfeasible. Such cases in the present set of simulations comprehend approximately 35.56% of the two-steps results shown, the absolute majority corresponding to feasible trajectories for the west goals only, or the goals located in the left half of the plane, on account of the start point being located in such partition, at $(2, 2)$. To verify such assertion, a second set of simulations was made with the same conditions, except for the starting goal, placed instead at $(20, 2)$. The simulations showed 69% of direct successes and 31% of two-steps solutions, with no complete or partial failures registered, which stresses the influence of the starting position over the results obtained. The average and maximum execution times also presented a slight difference, with a reduction of 30 seconds on each. It is also important to highlight that the partial failure does not mean that the

remaining goals characterise an unfeasible goal list, just that it must be recomputed, preferably as a single mission, or broken down in more missions with less goals.

The second remark concerns the complete failure, which similarly to the aforementioned failure in the two-steps does not imply that the goals to which the algorithm fail to find a feasible trajectory are unfeasible goals. Due to the difficulty level and the intrinsic randomness of the process, a simple recomputation can lead to a feasible trajectory, either in a single or two-steps solution, or if the failure persists, the mission can be divided in simpler missions and recomputed, since it is notable that the level of complexity swiftly decreases with the number of sub-paths to be computed.

With respect to the execution time, the same behaviour observed in the previous simulations regarding the minimum time values registered is verified. As the violin graph in Figure 4.29 exposes, the minimum computation time, 51.54 seconds, is close to the average of a singe path computation times the number of sub-paths computed, in this case ten. The overall behaviour, however, is slightly different, since due to the increase in the complexity, the distribution is more widespread, or in other words, more executions take a longer time to find a feasible trajectory than in comparison with the previous simulations with one and four goals.

The last simulations performed were nine goals executions considering fixed goals randomly defined beforehand. The goals were positioned at $(8, 13)$, $(5, 24)$, $(28, 2)$, $(31, 15)$, $(16, 31)$, $(27, 23)$, $(6, 36)$, $(35, 12)$ and $(31, 36)$, as can be seen in Figure 4.30. The first example in the figure presents a direct success, the second a two-steps solution and the third, another two-steps solution except that in this case achieved within a longer time and with a different and more complex trajectory as a result, in order to once more highlight the randomness characteristic of the results.

As it can be better verified on the nine fixed goals simulation results in

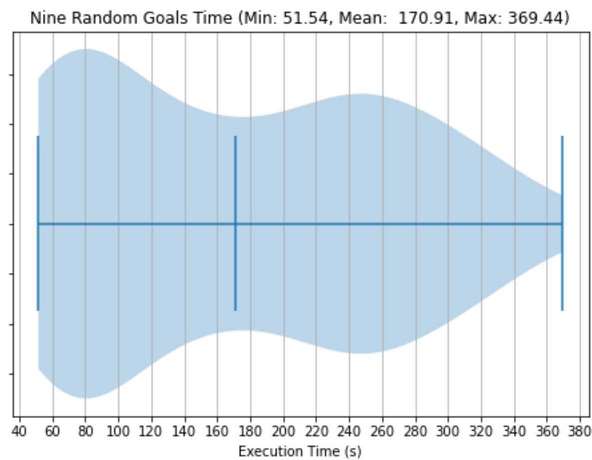

Figure 4.29: Violin plot of the time for 100 nine random goals executions.

Figure 4.31, the difficulty of the task highly depends on the location of the goals, so in respect to the random goals simulations, the fixed goals simulations may present either a higher number of two-steps results or of success results depending on the difficulty inherent to the goals chosen. Naturally, harder tasks also rend the algorithm more prone to failure results, as already discussed.

For this particular set of goals, the predominant results were two-steps solutions, totalling 83% of the results, against 17% of direct successes and no complete failures. Concerning the 83% of two-steps solutions, approximately 9.64% presented a partial failure, that is one of the two sub-trajectories, mostly the east and farthest from the starting point one, resulted as unfeasible and hence failed.

The violin plot in Figure 4.32 reinforces the idea of a demanding task. Differently from the previous simulations, most of the execution here are located near the upper extreme of the graph, indicating that the majority of the computations took a considerable amount of time and involved searching for possible alternate paths, re-span of the random tree for many sub-paths and the repetition of these procedures in case the two-steps routine is executed.

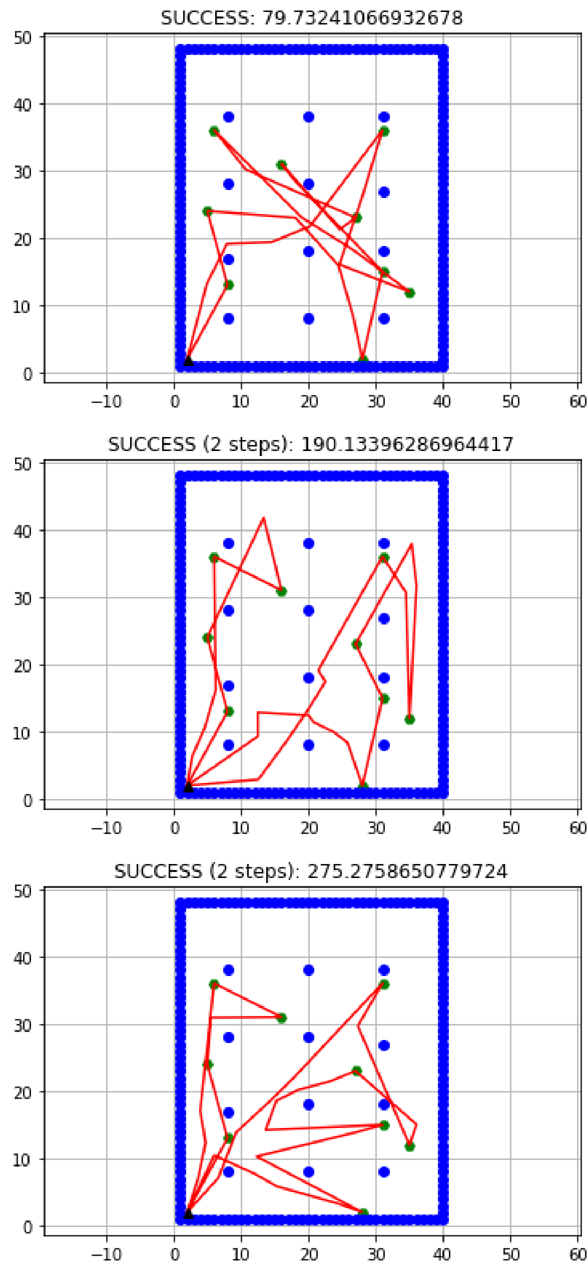

Figure 4.30: Nine goals executions with fixed goal points.

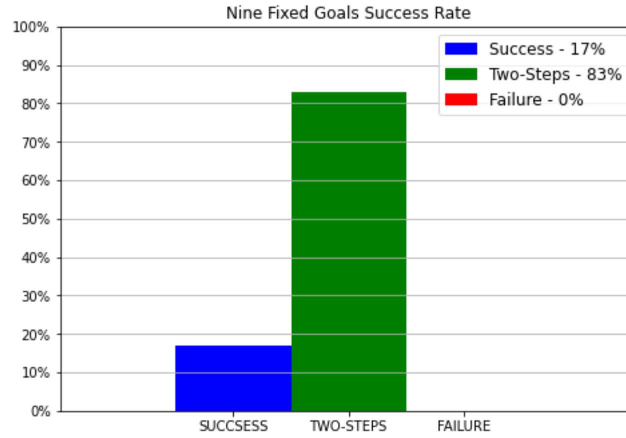

Figure 4.31: Success rate of nine goals execution with fixed goal points.

The average of the execution time nevertheless is still under 5 minutes, being in this case 238.78 seconds.

The violin plot for the simple cost of the simulations exhibits a particularly interesting characteristic, as it can be verified in Figure 4.33. It can be observed two distinct distributions for the cost, the lower one concerning the success on the first attempt and the second and more expressive one, regarding the two-steps results, which inherently are characterised by longer paths to be traversed and consequently higher simple costs.

Considerations and Remarks

The foremost and main consideration to be done relates to the overall execution time. It is reported in [15] that the average time of the modified RRT* algorithm implemented in the rover and described in Chapter 3 is around 5 minutes per sub-path computed. Therefore, for a four goals scenario, for example, the modified RRT* would take around 25 minutes to find a feasible path. Comparing such execution time with the results herein presented, considering the worst cases registered, *i.e.* the maximum execution times, the RRT^X with geometric approach represents an expressive reduction of

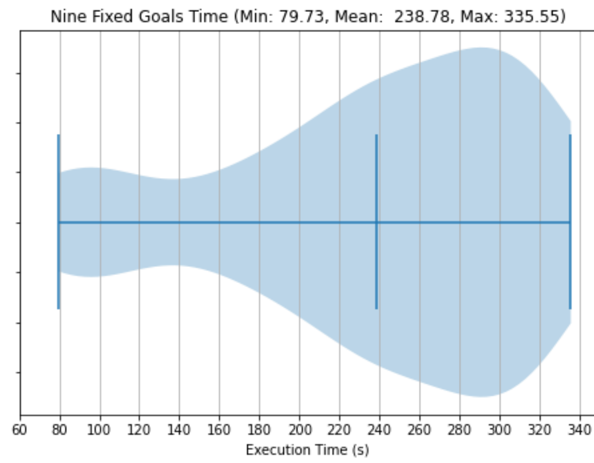

Figure 4.32: Violin plot of the time for 100 nine fixed goals executions.

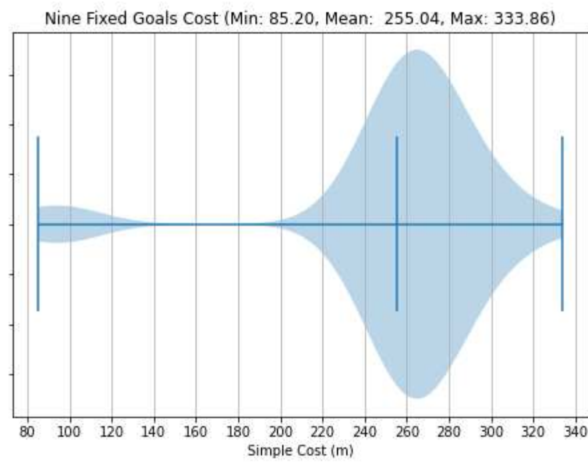

Figure 4.33: Violin plot of the simple cost for 100 nine fixed goals executions.

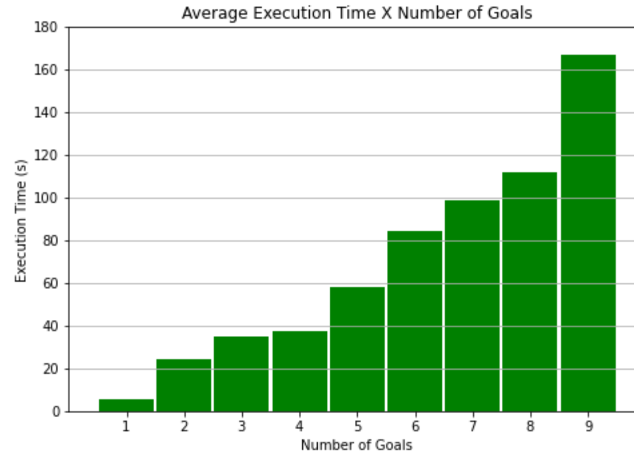

Figure 4.34: Bar graph of the average execution time per number of goals.

90% in the time used for planning the trajectory, rendering it well suited to deal with both dynamic environments and the cable constraints, at least with respect to time.

The bar graph in Figure 4.34 provides a general panorama of the execution time over different numbers of goals, rendering possible to observe the growth tendency of the execution time as a function of the main operational parameter. The data being shown corresponds to the average execution time for each number of goals, from one to nine, for 25 samples of random goal executions.

Another important remark is related to the routines equipped to handle the dynamic environment. Although their performance was not explicitly analysed in the present section, the fact that these routines are used as part of the geometric approach provided their evaluation. The re-span operation managed throughout the simulations to add new obstacles and successfully erase branches, regrowing the three around the imposed virtual obstacles in search for alternative routes to avoid the existing problematic obstacles. The execution time of a re-span operation varies from around 8 to 15 seconds, with outliers close to 20 seconds being registered on rarer occasions. Such

is the execution time expected and verified when performing experimental tests with unexpected obstacles being avoided by the rover.

Last remark involves the Raspberry Pi execution of the algorithm. Given simulations and experiments, the Raspberry Pi 4 installed in the rover to substitute the previous hardware presented an adequate execution time, albeit slower than the simulations on the PC, as expected. The average computational time on the Raspberry Pi is around 1.5 times the computational times here presented, that is for the worst cases where the algorithm took around 6 minutes and 30 seconds to find a feasible trajectory, the Raspberry Pi execution takes around 10 minutes, for example. Such characteristic only affects the Raspberry Pi single operation mode presented in the Chapter 5, since this mode of operation is the only one where the planning is computed on the Raspberry Pi and not on the PC. More strict limits of operation can be determined for this mode of operation in case it is necessary to assure an execution time close to 5 minutes, for instance.

4.4.2 RRT^X Performance Comparison

In order to attest the performance of the RRT^X algorithm, base of the method herein presented, in comparison to the staple RRT* algorithm, base of the previous method, a series of simple tests were made. An additional motivation was also the comparison made in [27], where it is said that the amortised computational cost of the RRT^X is similar to those of the RRT and RRT* algorithm, notwithstanding the tests here performed did not intend to evaluate the amortised cost.

The tests were characterised by simple point-to-point path computations in different maps, where the overall time and the time per iteration were registered and compared. With such data, it is possible to evaluate how the growth of the tree affects the execution time of the algorithm, which consequently aids in defining execution parameters for the operation. Moreover,

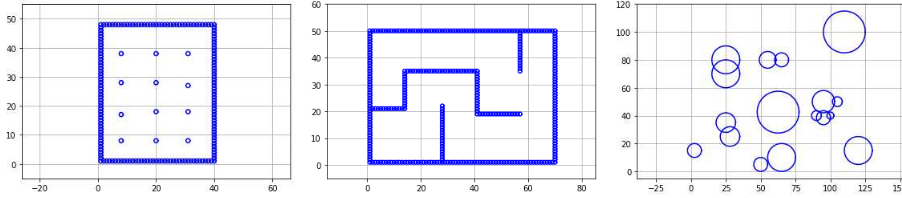

Figure 4.35: The maps used for the performance analysis.

the simple cost of the solutions found and the number of nodes computed for a fixed number of iterations were also registered as a mean to better assess the behaviour of each algorithm.

The maps used were based in [50], where reference maps are presented with the specific purpose of motion planning algorithm testing. Three maps were used, the original map representing the subfloor where the rover must operate, a simplified labyrinth and an open map with randomly positioned obstacles with different sizes. For each map, a fixed number of iterations was defined and 100 samples were collected from each algorithm. Afterwards, the results were compared. Figure 4.35 shows the three maps used for the tests in the performance analysis. The same maps can be observed in greater detail in Appendix A.

Initially, the RRT^X exhibited a highly exponential tendency of growth in computational time as the number of iterations, and thus the tree would grow as well, while the RRT* displayed a linear pattern of growth, as expected. Upon close inspection, it was observed that one of the routines to reinforce tree integrity, mentioned previously, was causing the exponential behaviour due to being called and scanning the growing tree during each iteration. The structure of the algorithm was then reviewed and fixed to not incur in overly expensive computational operations, while still reinforcing the integrity of the tree.

After the repetition of the tests, the RRT^X program written for the present work still presented an exponential growth, nonetheless significantly

diminished compared to the prior version, in such a way that for a range of iterations the performance response of both algorithms is considerably close. Overall, however, the RRT* still presented a quicker response in respect to computational time and a linear growth with respect to the number of iterations. Regarding the other data collected, the behaviour of the two algorithms, as expected from two approximately optimal methods, is rather similar.

Figure 4.36 displays three violin plots corresponding to the comparison between the iteration time of the RRT^X and the RRT* motion planning algorithms for the three aforementioned maps. As it can be observed, the iteration time for the RRT^X algorithm presented a higher value on average if compared to the one of RRT* across the three graphs. However its distribution lies towards the lower extremity of the graphs, which along with the general low iteration times observed, grant a feasible and similar operation with respect to RRT* for a certain maximum number of iterations.

For the sake of brevity, more graphs containing the data and results here described can be observed in Appendix A.

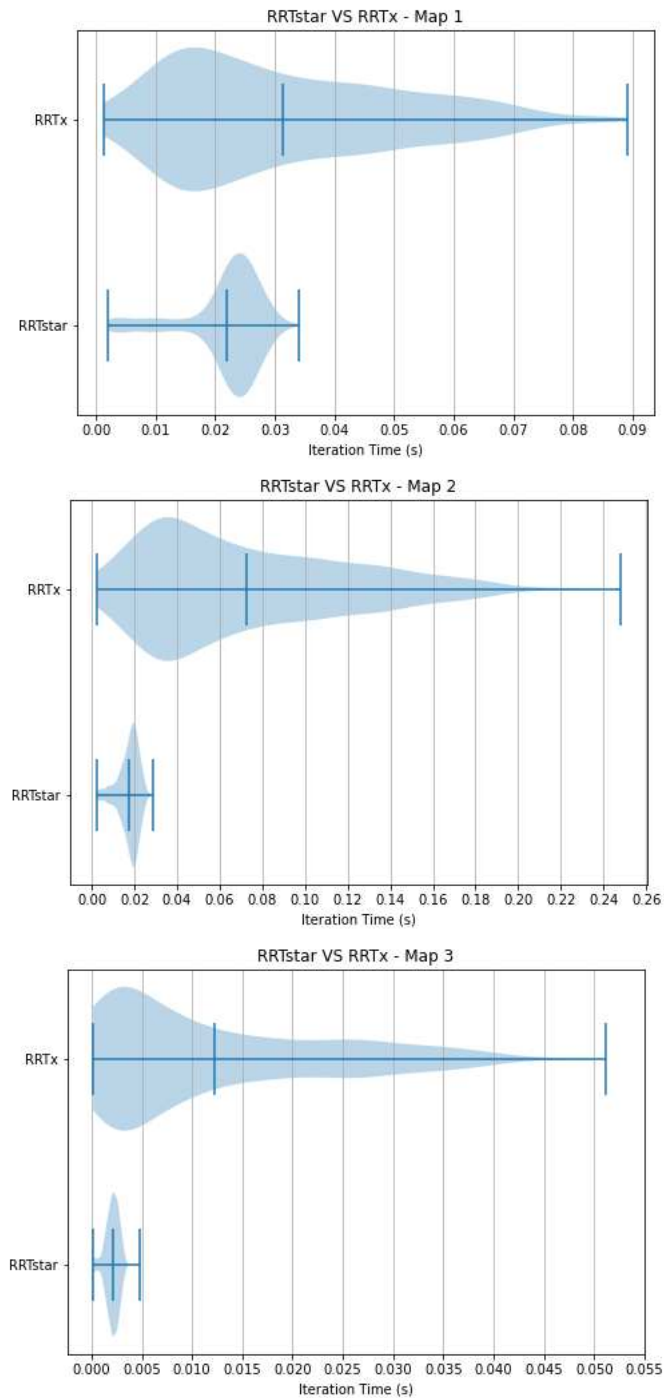

Figure 4.36: Violin plots of the iteration time for RRT^X and RRT considering the three maps used.*

Chapter 5

Device Integration and Experimental Results

As presented in [15] and reviewed in Chapter 3, the previous operation structure of the rover implied in the motion planning algorithm being executed once on the PC, then transferring the resultant path via Ethernet cable to the Raspberry Pi embedded in the rover who would in turn read the nodes composing the path and transfer them via *SPI* communication to the Arduino board controlling the stepper motors. Therefore, it was not required from the previous operation structure to quickly and constantly exchange information between the PC and the Raspberry Pi, or in other words between the planning routine and the movement routine. Apart from the video of the screen capture of the Raspberry Pi being shared with the PC, the only data being exchanged was a single text file transferred manually, *i.e.* with the direct intervention of the user, through the VNC software.

The current operation structure, however, demands that data be frequently exchanged between the PC and the Raspberry Pi in both directions, apart as mentioned before from the screen sharing. Not only is necessary that the trajectory be shared with the movement program, but also that the position of the rover and obstacles must be shared with the motion plan-

ning routine so that the path be recomputed according to RRT^X dynamic environment obstacle routines. Preferably, such communication should be performed without any addition of hardware and in the most automated way possible, aiming at reducing the role of the user in coordinating the movement of the rover and thus allowing the user to focus on the supervising of the operation and inspection of the structures through the camera of the rover.

5.1 Device Integration

In this context, three modes of operation were defined for the rover:

- The first being similar to the previous operation structure, where the user has to manually transfer the files between the PC and the Raspberry Pi.
- The second mode of operation using only the Raspberry Pi to perform both the planning and the movement and, therefore, not demanding any transference of files.
- And the third mode creating a shared environment between both devices such that, similarly to the second mode of operation, the transference of files is not needed.

5.1.1 Separated Environments Operation

The first mode of operation is characterised by the two devices, the PC and the Raspberry Pi, operating separately in its respective environments. All the exchanges of data between the two devices must be mediated by the user via the VNC software transfer option. Although the transfer time by itself is irrelevant, it indeed takes a short but relevant amount of time for the user to select, transfer the data files and execute once more the routines.

Furthermore it is worth to also remark that the whole process is more prone to errors.

Regarding the files being transferred, apart from the text file previously mentioned containing the trajectory, the file being exchanged with the PC with the purpose of re-planning the path in case of an unknown obstacle was saved as a *.npz* file. A *.npz* file is a binary file that can be written and read by the NumPy scientific Python library [51] and is capable of storing several NumPy array structures. Although having the limitation of only being read with the use of the NumPy library, the *.npz* files are lighter and faster to read, as well as reinforce data structure consistency between different programs communicating to each-other.

In the context of the rover operation, the *.npz* file contains three arrays, which are: the position of the rover when an unknown obstacle is found; the location of the obstacle in the map; the last and next node with respect to the position of the rover in the trajectory being followed. In the Figure 5.1 it is possible to observe a simplified scheme of the above-mentioned mode of operation.

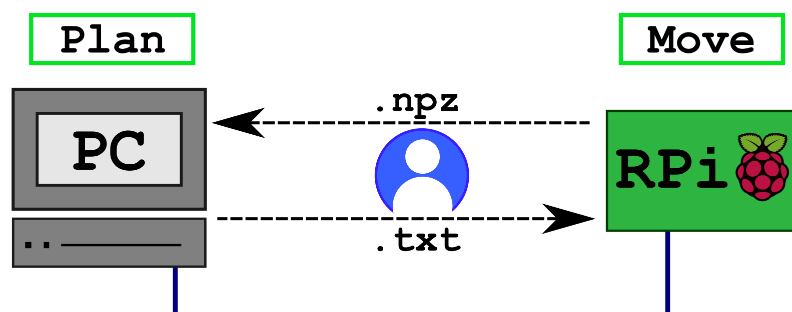

Figure 5.1: Simplified scheme of the Separated Environments Operation.

5.1.2 Raspberry Pi Single Operation

The second mode of operation, here called Raspberry Pi single operation, is mainly characterised by the execution of both the planning and movement

routines inside the Raspberry Pi, delegating to the PC only the monitoring of the video captured by the camera of the rover through VNC screen sharing. This mode of operation was made possible only because the present planning routine, as shown in the algorithm performance analysis in Chapter 4, has a feasible computational time even when executed on the Raspberry Pi, which has a reduced processing power. However, it is worth to stress that the Raspberry Pi embedded in the rover was upgraded, from version 3 model B+, as presented in [15], to version 4 model B.

For the integration of the planning and the moving routines, a new routine was written merging the two operations into one single continuous execution. Thus, all the data exchange is made inside the same program and no file transfer is needed. Compared to the shared environment approach there are no relevant advantages, but in case transferring is preferred to be avoided or the rover must to be run blind, that is, without constant screen sharing with the PC during the movement phase, the current mode of operation presents itself as a viable option. Figure 5.2 shows the simplified scheme of the Raspberry Pi single operation mode.

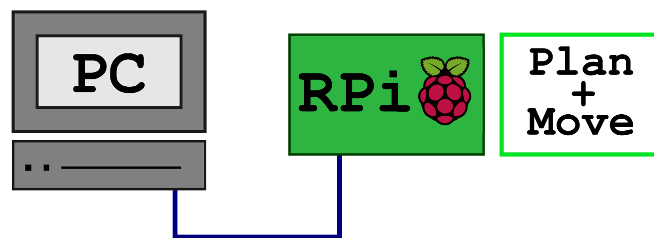

Figure 5.2: Simplified scheme of the Raspberry Pi Single Operation.

5.1.3 Shared Environment Operation

The third and last mode of operation is the most advantageous of the three of them. It allows the user to operate on both devices as one, since all the routines and files in this mode of operation are located in the same storage

unit, *i.e.* the shared environment. Effectively, the storage unit being used is the Raspberry Pi memory but through Samba [52], a Raspberry Pi implementation of the Service Message Block protocol targeted to file sharing over a network [53], any file on a predefined folder can be simultaneously accessed and edited by a device using Windows or Mac OS and with the proper credentials. It worth to remark that, on both operational systems, the connection with Raspberry via SMB protocol is native, *i.e.* no additional software is required.

Making use of Samba therefore, the routines were adapted in such a way that they could operate without the intervention of the user. A shared folder was created where both the files for the motion planning and movement of the rover were to be allocated. Such programs produce, similarly to the first mode of operation, a *.txt* and a *.npz* file whenever a new path was generated or the movement of the rover started respectively. The two routines also wait in stand-by mode until a new *.npz* or *.txt* is generated, respectively when an unexpected obstacle or the end of the trajectory is reached and when the path is re-planned after encountering an obstacle.

This mode of operation, in similar fashion to the second mode of operation here presented, requires only for the user to introduce the initial parameters of the execution, such as start point, goal points and initial attitude and to execute the routines in their respective devices. After both programs are started, the planning being executed on the PC and the movement one being executed on the Raspberry Pi, the rest of the operation proceeds autonomously until the end of the trajectory is reached, including the possibility of unexpected obstacles. Figure 5.3 presents a simplified scheme of the shared environment mode of operation.

There were two main concerning points regarding the shared environment operation. The first was whether using the Ethernet cable for the PC-Raspberry Pi communication via Samba would interfere with the VNC

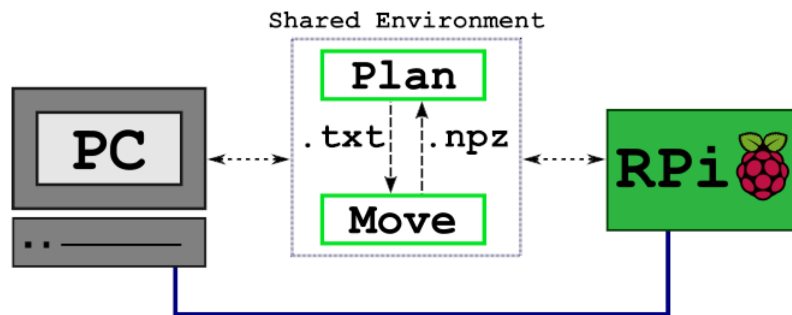

Figure 5.3: Simplified scheme of the Shared Environment Operation.

software being used to share the screen and control the Raspberry Pi through the PC. The second, whether synchronisation problems could occur while two programs are generating files, standing-by and reading the newly generated files, as errors could occur in case one routine tries to read a file while it is being re-written, or if it happens to read an outdated version of a file and derive wrong instructions from it.

The initial operation tests quickly showed that the first possible problem was not verified, that is there was no problem in using the Ethernet cable for both the VNC screen sharing and control function and the Samba shared environment. The second concerning point was discarded after further operation tests, when it was verified that the rover had no synchronisation problems whenever a *.npz* or *.txt* was generated and read. Since both files are light, their writing speed is fast enough to not incur is misreads. Moreover, modifications were made prior to the tests such that every time a file is read, it is immediately erased, in this manner avoiding accumulation of outdated files. In other words, at any given moment there will be either one or no *.txt* and *.npz* in the shared folder.

Finally, experimental tests were conducted to investigate the behaviour of the method and the three modes of operation in a real-case scenario. The tests contributed to validate the work developed, but also to acquire information regarding necessary future improvements and developments.

5.2 Experimental Tests

The experimental tests constitute an important part of the process of validation and analysis of the method developed. With the experiments, it was expected to evaluate the response of the method in the real case scenario, analyse its actual feasibility and the issues derived from the real application, possibly neglected during development and simulations. More specifically, the capability of the rover of moving in a dynamic environment, especially considering time constraints, and its interaction with the PC according to the modes of operation defined in the previous section, as well as how the rover would physically interact with the tether, given that problems in this aspect are indeed mentioned by [15].

The area utilised for the experiments was a hall with approximately 13 metres of length, 7 metres of width and granite floor. The hall had also two access points to the electrical network in the floor, initially considered for the role of obstacles in the simulations, for the sake convenience. A simplified map of the area as well as a picture taken during one of the experiments can be observed in Figure 5.4. The rover can also be observed in the lower right corner.

5.2.1 Problems Encountered

With the start of the experimental tests, it became evident that some of the previous problems introduced in [15] considerably hindered the capability of the rover of moving properly and hence performing the tasks designated. Additionally, new problems not previously mentioned were verified. In a similar fashion, these problems would affect in some degree the proper operation of the rover and prevent it from achieving the desired results. The main problems encountered and worth mentioning are the error in attitude processing, the overheating of the Raspberry Pi, the manoeuvres involving the rover passing over its cable, and the most grave one, the angular and

Figure 5.4: Experiments area.

linear errors in the displacement of the rover.

The attitude processing issue caused the rover to misinterpret its own attitude for certain angle configurations. The most remarkable one was the 180° rotation, *i.e.* when the next point on the list containing the nodes of the trajectory required an exact 180° rotation, to which the rover would interpret as a 0° rotation, that is no rotation, and move ahead instead of revolving and moving in the opposite direction. Similar situations would occur for rotation movements such 90° and −90° rotations. The routine being used to generate the instructions sent to the Arduino board was composed, rotation-wise, by a series of comparison statements to define the attitude of the rover based on the nodes in the path list and, upon review, it was found that the comparisons did not include a range of situations, such as the ones above-mentioned. After the correction of such comparisons to include all possible attitude situations, the problem was considered and effectively verified as solved. It is worth to remark however that the problem described exposes the fact that albeit the rover has an IMU sensor, no feedback is provided to the Raspberry Pi regarding odometry or the attitude of the rover.

The overheating issue was verified after around two hours of continuous

or nearly continuous operation. After the Raspberry Pi indicated overheating through an icon in the screen being shared, the movement of the rover, most probably due to severed communication with the Arduino board, becomes either erratic or ceases completely. To mitigate the problem, as already expressed, the Raspberry Pi was substituted for a newer model containing also heat dissipators and a fan. The power being fed to the system was also corrected through an adjustment in the DC-DC converter connected to the Raspberry Pi, providing $5V$, since the Raspberry Pi was also showing the alert sign for low power.

Regarding more general grievous problems, *i.e.* the tether related issues and the movement inconsistency, no definitive solution was found and implemented, although for the former a solution was proposed and for the later some were attempted, while the source of the problems was being investigated. The tether related problems, whereas having a single cause, can be divided into two, the rotation over the cable and the crossing of the cable while performing a linear movement. The referred cause for both problems is the fact that the rover cannot cross, or pass over, the cable, likely due to a combination of low traction with inadequate wheels for the task. When striving to do so, the wheels start to slip and the rover moves incorrectly, losing its proper sense of position, since the localisation of the rover is performed by the computation of its odometry, without any feedback from the sensors.

The problem of the rotation over the tether, previously pointed by [15], is characterised by the rover encountering its own tether while attempting to perform a rotation. One possible solution would be to attach a tail-like structure in the rear part of the rover without increasing the radius of the imaginary circle circumscribing the rover, such that the tether is always kept at some distance from the wheels during rotation. This idea might be seen graphically in the Figure 5.5. Notwithstanding the proposed solution might

not eradicate the problem, it might drastically reduce its occurrence and impact. As for the experiments, during every rotation of the rover, the user would hold the tether above the rover, so it would not interfere.

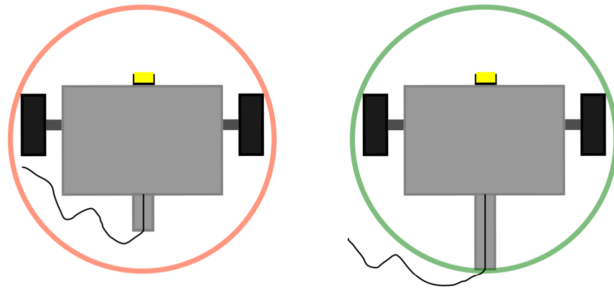

Figure 5.5: Sketch of the solution proposed.

One of the greatest impediments to the proper testing of the algorithm developed was the fact that during linear movement, the rover presents itself as not able to cross over its own tether. As it can be seen from figures previously shown in the previous chapter, the feasible trajectories required for the rover to pass over its own cable, typically many times per mission. Moreover, [15] describes that upon testing in floors with more dirt or sandier grounds, the rover displays slippage, which as already mentioned is rather problematic. The first and foremost solution to be attempted is to change the wheels of the rover to off-road wheels that offer a higher grip. If such solution fails to extinguish the problem, a more powerful motor must be sought.

Finally, the issue that more severely affected the performance of the rover was the deviation in the movement of the rover due to the inconsistent operation of the motors. The initial lurch, as described by [15], is caused by the asynchronous activation of the motors, however the experiments point to more reasons that most probably combine themselves to cause such undesired yaw. One of the possible additional factors is of mechanical order, being it an uneven resistance to rotation in the wheels, *i.e.* one of the wheels requiring less effort from the motor to be spun than the other. Another pos-

sible factor might be related to faulty wiring or other electrical disturbances. The signal sent from the Arduino board to the driver was verified, but no anomaly was found, as can be observed in Figure 5.6 where the driver is receiving pulses, measured by an oscilloscope, corresponding to a rotation of the left wheel at a constant speed of 10 steps per second.

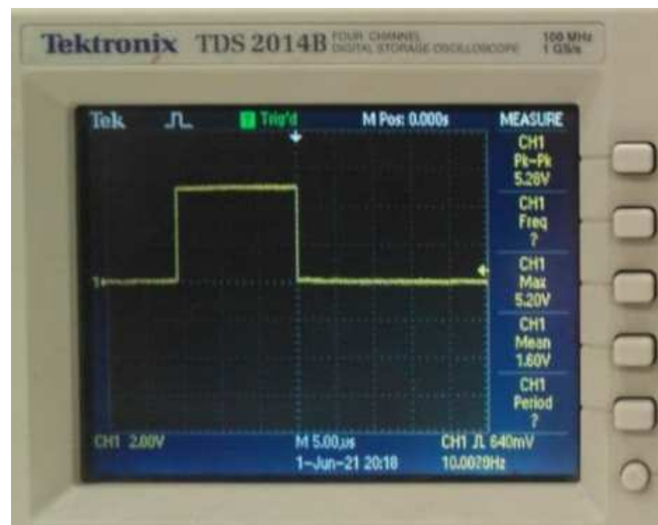

Figure 5.6: Arduino to driver signal analysis.

To aggravate the situation, a swerve was also observed occasionally at the end of a point-to-point trajectory, which increases the attitude error of the rover, as well as a general linear deviation, that is the rover was not moving linearly the amount requested. The average angular displacement measured is around 2.5°, which matches the value presented by [15], nevertheless the angular displacement proven not to be negligible as previously proposed, despite it being small. Even if the other two forms of displacement were not present, the accumulation of small errors on every point-to-point move would incur in drastic errors in the final position and even in the obstacle avoidance, considering that every trajectory consists of many point-to-point segments. With the inclusion of all perceived forms of displacement error, the movement of the rover is rendered completely unreliable. Figure 5.7

shows an illustration of the movement of the rover considering in the upper part only the initial lurch of 2.5°, in the figure increased to 5° for better visibility, and in the lower part all the displacement errors observed.

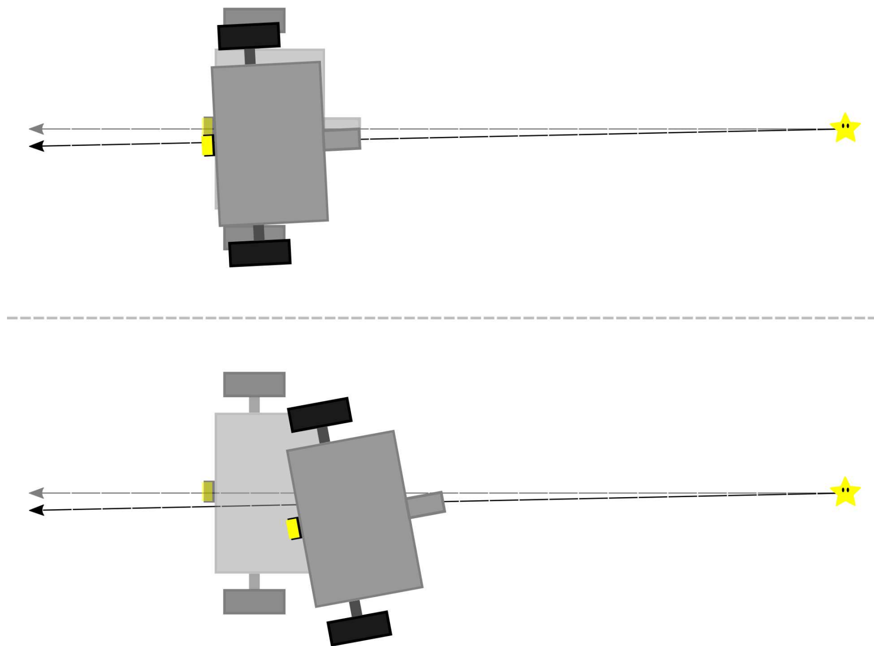

Figure 5.7: Displacement error illustrated.

It is worth to remark that the rover exhibited, during the experiments, an average linear misplacement of minus 20 centimetres. Figure 5.8 exhibits a picture taken after a three metres straight movement experiment. The orange dot marks the precise position where the rover should have ended considering its starting point, and the orange dashed line demonstrates the trajectory that ideally should have been followed.

Among the attempts of solution for the problem, it can be remarked the inspection and alteration of the low-level program, the alteration of the odometry system used and the inspection of the power supply and conditions of operation of the driver, as already partially mentioned. Of these attempts, the only one to show visible results was the alteration of the odometry. The

Figure 5.8: Straight movement experiment showing the displacement error.

low-level with alternated activation and deactivation of the motors and with simultaneous control of the motors did not present any difference concerning the behaviour of the rover. In the same fashion, the driver investigation lead to no useful results, except from the realisation that in certain occasions the driver seems to have overheated, due to the melting marks on the breadboard. Some of the melting marks can be see in Figure 5.9.

The alteration of the odometry was made by using the step counter, intrinsic to the Arduino stepper library being used, to control how much the motor should move, instead of the Time-of-Flight TeraRanger sensor previously being used for the odometry, which albeit considerably precise, is prone to small variations and errors, especially if considered the data acquisition update rate in the Raspberry Pi routine. The TeraRanger sensor is currently being used therefore to assess whether the rover finished the movement sequence, so another movement instruction can be sent to the Arduino. This is done by comparing two successive measures of the sensor

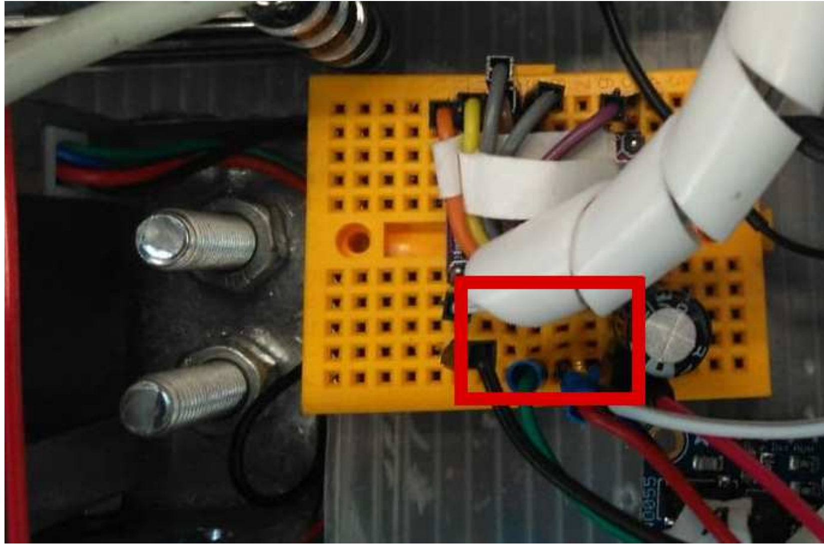

Figure 5.9: Driver relative to the left motor.

within an interval of around 2.5 seconds and if the measures are equal, given a small threshold, the rover is considered as stopped and the subsequent instruction is sent, if it is the case. The same routine is responsible for checking the path ahead for unexpected obstacles, however with a higher update rate, with a check every 0.1 second. Whenever the sensor registers that its lower limit of 0.5 metre was reached, the rover stops and the new obstacle ahead is added to the obstacle list, followed by the appropriated re-plan routines of the RRT^X method.

The odometry being performed by Arduino in conjunction with the TeraRanger sensor culminated to more precise linear movements, yet the initial and final lurch are still present. The use of the sensor for obstacle identification also proved to work satisfactorily, however it requires improvements especially regarding the adequate estimation of the size of the obstacle, since for bigger obstacles the approach presents risks of collision due to undersizing of the obstacle in the perspective of the routine.

5.2.2 Operation Assessment

Notwithstanding all the aforementioned issues, simple tests were made to verify the operation of the method as a whole and its interaction with the other systems, as well as the overall system integration in the context of the three modes of operation. Primarily, for simple straight trajectories, that is a single goal whose goal lies directly ahead of the starting goal, the movement and connection of the devices for the separated and shared environment modes of operation were analysed. It was observed that successive repeated goals in the path list introduce a delay in the operation, given that the rover has to read the node and process the information that no movement or rotation should be performed. Successive repeated nodes occur because, as expressed in Chapter 4, the starting point of a sub-path in the motion planning is defined as the goal of the previous sub-path computed, even for single goals. Thus the path output of the algorithm was adapted to remove such repeated goals, rendering the operation more smooth. For the tests, a single goal task was then comprised by three goals, the starting point, the goal point and lastly the starting point once more. Furthermore, the goals were positioned always around three metres from the start point, to limit the deviations and facilitate the experiments.

Figure 5.10 presents an example of the referred experiments, with the following path re-planning performed after the unexpected obstacle was identified.

The separated mode of operation, apart from the inconvenience of transferring the files manually, worked as expected. The shared environment mode, a novelty for the rover, also responded superbly, guaranteeing a fully automated operation with the supervision of the user and no signs of synchrony problems. After evaluating the behaviour of the method for simple single goal trajectories, a small box-like object was placed on a position between the starting point and the goal point, which for both modes con-

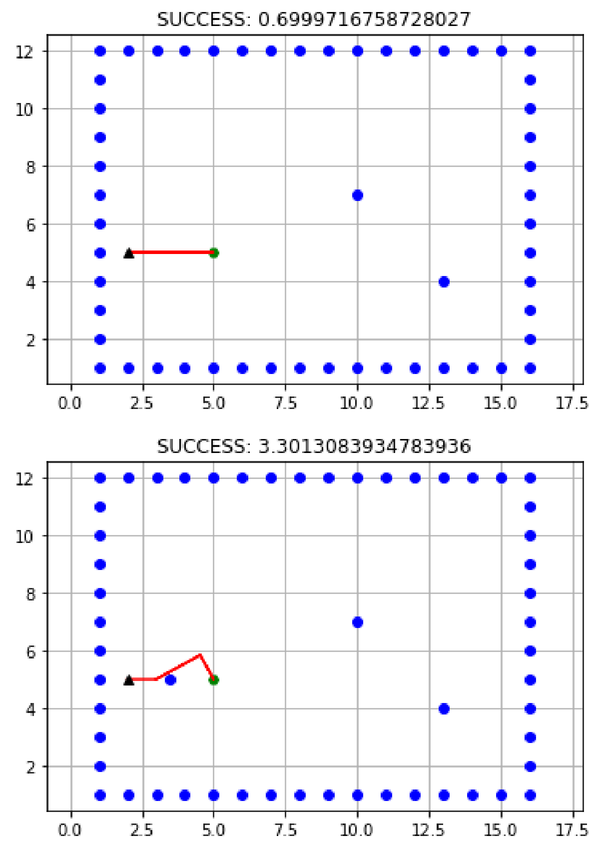

Figure 5.10: Example of an experiment sample for a straight movement.

sidered produced satisfactory results, *i.e.* the obstacle was recognised, processed by the RRT^X with constraint approach and thereafter the new path was processed by the Raspberry Pi, which proceeded to send the new movement instructions to the Arduino board. Since a single goal was being computed and later re-planned, the whole process took seconds only, in accordance to the results previously presented in Chapter 4. Subsequently, the Raspberry Pi single mode of operation was tested under the same circumstances, presenting thus similar successful results, with the only difference of a slightly higher time of operation, which for a small number of goals is a negligible difference.

The last test was similar to the previous one, with the exception that one goal was added, totalling two goals forming a triangular trajectory. The three modes of operation were able to plan the path, start the movement, identify the obstacle, update the trajectory and resume the movement towards the next goal and after it return to the starting point. The whole mission elapsed in an acceptable time. Although the results were overall positive for the experimental tests, more demanding experiments are absolutely necessary, not merely to validate the method developed in circumstances more similar to the real-case scenario but as well to foster the perfecting of the method with the aim of achieving more robustness and reliability.

In Figure 5.11 it is possible to observe an experiment sample with two goals forming a triangular trajectory, instead of a straight movement. Similarly to the previous example, the path is re-planned after the unexpected obstacle was correctly identified.

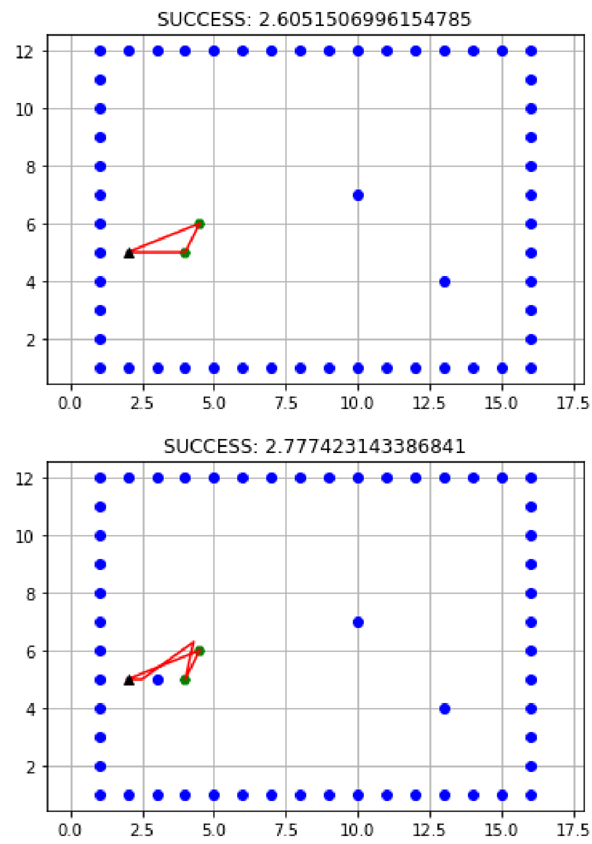

Figure 5.11: Example of an experiment sample for a two goals movement.

Chapter 6

Conclusion

The fields of mobile robotics for dynamic environments and tethered mobile robotics are both challenging areas, object of countless researches in the latest years, but the intersection of the two topics remains scarcely explored. In this context, the present thesis aimed in developing an algorithm capable of managing both situations, the dynamic environment and the tether imposed constraints on the mobile robot. The former was tackled by the existing dynamic motion planning algorithm RRT^X, written in Python and adapted for the purposes of the thesis, and the latter was approached by a method herein developed which makes use of intuitive notions and a geoprocessing Python library to quickly evaluate the feasibility of computed paths. The entirety of the work completed was intended for the specific implementation in a tethered rover developed for infiltration inspection in the subfloors of SAE units built in Amatrice, Italy.

Throughout the development of the work, many hindrances were encountered, from the implementation of the motion planning to the constraint approach to the experimental tests performed. Nevertheless, such hindrances were by-passed, fixed or adapted such that a reliable and efficient method for the dynamic planning of a tethered rover was achieved. Simulations were accomplished to assess the performance of the algorithm

on a range of operation scenarios and the execution time, success rate and cost of the solutions were evaluated.

The data collected point to a fast algorithm suited for the dynamic environment to which the rover was developed, with a success rate of effectively 100% and a reduction in execution time of 90% if compared with the previous algorithm implemented. Such progress guarantees also a higher limit of operation with respect the number of goals, once time ceases to be a hard constraint on the operation. Albeit the maximum number of goals utilised in the simulations presented in this thesis was set to nine goals for convenience, two more than applied by the previous algorithm, it does does not reflect the maximum limit of the method, remaining to be defined by the time and complexity constraints of the mission stipulated by the user.

The communication system of the rover has also been improved, henceforth including more modes of operation, with emphasis on the shared environment mode, and therefore expanding the possibilities of the rover regarding its functionalities. Principally, it must be highlighted the significant improvement of accomplishing a method capable of not only managing the tether constraints efficiently, but also equipped to handle dynamic environments, not contemplated in the previous algorithm. The experimental tests however were not enough to analyse thoroughly the performance of the method in the real-case scenario, being only capable, due to the limitations imposed by the movement issues of the rover, to verify if the algorithm is operational and test the different modes of device integration.

6.1 Future Developments

The main future development of the current work is concerning the hardware and low-level software of the rover. As presented in Chapter 5, many were the problems faced regarding the movement of the rover, being required therefore a full review and improvement of the hardware and low-level soft-

ware such that problems such as the displacement errors, lack of motor traction, inadequate wheels and cable crossing when rotating cease to exist and the rover succeed to move smoothly as necessary. The sensors and wiring of the rover must also be contemplated in the review, such that the rover can be better equipped for the inspection to which it is destined.

To conclude, the final outcome attained is a functional algorithm prepared to and efficient in addressing dynamic environments with tether related constraints. Modifications must be made in the rover and more experimental tests performed to render the rover fully prepared to be safely deployed in real missions.

Bibliography

[1] Bob Williams. An introduction to robotics. *Mech. and Cont. of Robotic Manipulators*, 2014.

[2] Andreas Kroll. A survey on mobile robots for industrial inspection. In *Proceedings of the Int. Conf. on Intelligent Autonomous Systems IAS10, Baden-Baden, Germany*, pages 406–414, 2008.

[3] R Zimroz, M Hutter, M Mistry, P Stefaniak, K Walas, and J Wodecki. Why should inspection robots be used in deep underground mines? In *Proceedings of the 27th International Symposium on Mine Planning and Equipment Selection-MPES 2018*, pages 497–507. Springer, 2019.

[4] Leszek Kasprzyczak, Stanisław Trenczek, and Maciej Cader. Robot for monitoring hazardous environments as a mechatronic product. *Journal of Automation Mobile Robotics and Intelligent Systems*, 6:57–64, 2012.

[5] Jeffrey Abouaf. Trial by fire: teleoperated robot targets chernobyl. *IEEE Computer Graphics and Applications*, 18(4):10–14, 1998.

[6] Jaka Katrasnik, Franjo Pernus, and Bostjan Likar. A survey of mobile robots for distribution power line inspection. *IEEE Transactions on power delivery*, 25(1):485–493, 2009.

[7] Aleksandar Lazinica. *Mobile robots: towards new applications*. 2006.

[8] Serdar Soylu, Alison A Proctor, Ron P Podhorodeski, Colin Bradley, and Bradley J Buckham. Precise trajectory control for an inspection class rov. *Ocean Engineering*, 111:508–523, 2016.

[9] Hyouk Ryeol Choi and Se-gon Roh. *In-pipe robot with active steering capability for moving inside of pipelines.* INTECH Open Access Publisher, 2007.

[10] John Faber Archila Diaz and Max Suell Dutra. Pipelines inspection robots; robos para inspecao de linhas de servico. 2008.

[11] Cristiano Zacarias Ferreira, Gerson Yuri Cagnani Conte, Juan Pablo Julca Avila, Renato Coelho Pereira, and Thiago Morais Ceratti Ribeiro. Underwater robotic vehicle for ship hull inspection: control system architecture. In *22nd International Congress of Mechanical Engineering (COBEM 2013)*, volume 6, pages 1231–1241, 2013.

[12] Mel Siegel and Priyan Gunatilake. Remote inspection technologies for aircraft skin inspection. In *Proceedings of the 1997 IEEE Workshop on Emergent Technologies and Virtual Systems for Instrumentation and Measurement, Niagara Falls, CANADA*, pages 79–78, 1997.

[13] Bing Lam Luk, Alexandar Djordjevic, Shiu Kit Tso, and King Pui Liu. Development of a range of robot and automation prototypes for service applications. In *Cutting Edge Robotics*. IntechOpen, 2005.

[14] Carlos Balaguer, Antonio Gimenez, and Alberto Jardón. Climbing robotsâ mobility for inspection and maintenance of 3d complex environments. *Autonomous Robots*, 18(2):157–169, 2005.

[15] Angelo Marchesi. Design, prototyping and testing of a tethered rover with advanced path planning for building inspection. 2019.

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

[17] Melissa M Tanner, Joel W Burdick, and Issa AD Nesnas. Online motion planning for tethered robots in extreme terrain. In *2013 IEEE International Conference on Robotics and Automation*, pages 5557–5564. IEEE, 2013.

[18] Oren Salzman and Dan Halperin. Optimal motion planning for a tethered robot: Efficient preprocessing for fast shortest paths queries. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4161–4166. IEEE, 2015.

[19] Soonkyum Kim, Subhrajit Bhattacharya, and Vijay Kumar. Path planning for a tethered mobile robot. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1132–1139. IEEE, 2014.

[20] Michael A Sebok and Herbert G Tanner. On the hybrid kinematics of tethered mobile robots. In *2019 American Control Conference (ACC)*, pages 25–30. IEEE, 2019.

[21] Reza HosseiniTeshnizi. *Motion Planning for a Tethered Mobile Robot.* PhD thesis, 2015.

[22] Chiara Fulgenzi, Anne Spalanzani, Christian Laugier, and Christopher Tay. Risk based motion planning and navigation in uncertain dynamic environment. 2010.

[23] Shuzhi Sam Ge and Yun J Cui. Dynamic motion planning for mobile robots using potential field method. *Autonomous robots*, 13(3):207–222, 2002.

[24] Stéphane Petti and Thierry Fraichard. Safe motion planning in dynamic environments. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2210–2215. IEEE, 2005.

[25] Tim Mercy, Ruben Van Parys, and Goele Pipeleers. Spline-based motion planning for autonomous guided vehicles in a dynamic environment. *IEEE Transactions on Control Systems Technology*, 26(6):2182–2189, 2017.

[26] Nathan Ratliff, Matt Zucker, J Andrew Bagnell, and Siddhartha Srinivasa. CHOMP: Gradient optimization techniques for efficient motion planning. In *2009 IEEE International Conference on Robotics and Automation*, pages 489–494. IEEE, 2009.

[27] Michael Otte and Emilio Frazzoli. RRT^X: Asymptotically optimal single-query sampling-based motion planning with quick replanning. *The International Journal of Robotics Research*, 35(7):797–822, 2016.

[28] Giulia Barra, Alberto Marzo, Serena Olcuire, and Davide Olori. Emidio di treviri, uno sguardo critico sulla gestione del dopo-terremoto dellâappennino centrale, tra movimenti centrifughi e la (ri) costruzione di nuove vocazioni territoriali.

[29] Floor Levelling Systems Ltd. Subfloor technical: floor levelling systems ltd.: raised flooring system, 2021. `https://www.flslimited.com/subfloor/subfloor-technical/`.

[30] Devin Connell and Hung Manh La. Dynamic path planning and replanning for mobile robots using rrt. In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 1429–1434. IEEE, 2017.

[31] Kevin M.. Lynch and Frank Chongwoo Park. *Modern Robotics: Mechanics, Planning, and Control*. Cambridge University Press, 2017.

[32] Néstor García, Raúl Suárez, and Jan Rosell. Hg-rrt*: Human-guided optimal random trees for motion planning. In *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*, pages 1–7. IEEE, 2015.

[33] Michael Otte and Emilio Frazzoli. RRT^X: Real-time motion planning/replanning for environments with unpredictable obstacles. In *Algorithmic Foundations of Robotics XI*, pages 461–478. Springer, 2015.

[34] Peter Corke. *Robotics, vision and control: fundamental algorithms in MATLAB® second, completely revised*, volume 118. Springer, 2017.

[35] Sertac Karaman and Emilio Frazzoli. Incremental sampling-based algorithms for optimal motion planning. *Robotics Science and Systems VI*, 104(2), 2010.

[36] Oktay Arslan and Panagiotis Tsiotras. The role of vertex consistency in sampling-based algorithms for optimal motion planning. *arXiv preprint arXiv:1204.6453*, 2012.

[37] Oktay Arslan and Panagiotis Tsiotras. Use of relaxation methods in sampling-based algorithms for optimal motion planning. In *2013 IEEE International Conference on Robotics and Automation*, pages 2421–2428. IEEE, 2013.

[38] Michael Wilson Otte. *Any-com multi-robot path planning*. PhD thesis, University of Colorado at Boulder, 2011.

[39] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[40] Rebecca Fiebrink. Amortized analysis explained. 2007.

[41] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[42] Python Software Foundation. Python. `https://docs.python.org`.

[43] Atsushi Sakai, Daniel Ingram, Joseph Dinius, Karan Chawla, Antonin Raffin, and Alexis Paques. Pythonrobotics: a python code collection of robotics algorithms. *arXiv preprint arXiv:1808.10703*, 2018.

[44] Eric W Weisstein. Polygon. From MathWorld – A Wolfram Web Resource. `https://mathworld.wolfram.com/Polygon.html`.

[45] Sean Gillies. The Shapely User Manual. *Shapely 1.7.1 Documentation, Readthedocs.io*, sep 2020.

[46] Stig Nordbeck and Bengt Rystedt. Computer cartography point-in-polygon programs. *BIT Numerical Mathematics*, 7(1):39–64, 1967.

[47] Ivan E Sutherland, Robert F Sproull, and Robert A Schumacker. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys (CSUR)*, 6(1):1–55, 1974.

[48] Henrikki Tenkanen. Point in Polygon & Intersect, 2018. `https://automating-gis-processes.github.io/CSC18/lessons/L4/point-in-polygon.html`.

[49] Herbert Edelsbrunner, David Kirkpatrick, and Raimund Seidel. On the shape of a set of points in the plane. *IEEE Transactions on information theory*, 29(4):551–559, 1983.

[50] Jingyu Kim and Sang Hyo Arman Woo. Reference test maps for path planning algorithm test. *International Journal of Control, Automation and Systems*, 16(1):397–401, 2018.

[51] Eli Bressert. Scipy and numpy: an overview for developers. 2012.

[52] Raspberrypi.org. Raspberry Pi Documetation - Remote Access, 2021. `https://www.raspberrypi.org/documentation/remote-access/samba.md`.

[53] Microsoft. Microsoft SMB Protocol and CIFS Protocol Overview. `https://docs.microsoft.com/en-us/windows/win32/fileio/microsoft-smb-protocol-and-cifs-protocol-overview`.

Appendix A

Performance Comparison

In the present appendix, more details regarding the comparison carried out between the RRT^X and the RRT* are provided. More precisely the iteration time comparison between the two algorithms is deepened in respect to the brief discussion made in Chapter 4 and the specifications of the samples collected is described. The performance comparison served its purpose of better understanding the behaviour of the algorithm chosen for motion planning, investigating its viability and defining its internal operational parameters.

A.1 Test Description

As stated in Chapter 4, three maps were used to perform the point-to-point, *i.e.* single goal, executions used to evaluate the iteration time and cost efficiency of the RRT^X and RRT* algorithms. The first map utilised was the same map applied for the simulations of the routines developed herein, representing the subfloor for which the rover was designed to inspect. The second and third maps were derived from [50], which proposes ideal maps to test the performance of motion planning algorithms. The former was drawn as a simple labyrinth map and the latter, an open map with bigger dimensions in comparison with the first two and multiple different sized

obstacles. It can be observed in Figure A.1, Figure A.2 and Figure A.3 respectively the three maps here referred.

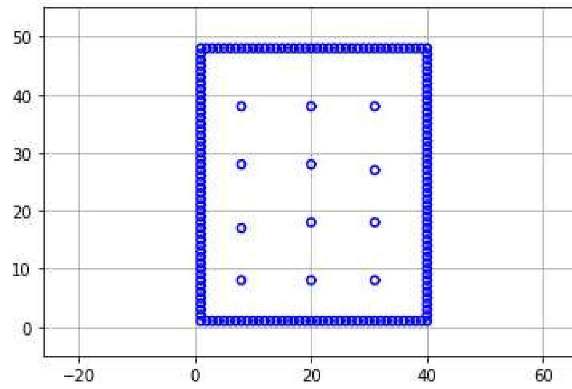

Figure A.1: First map used for the performance analysis.

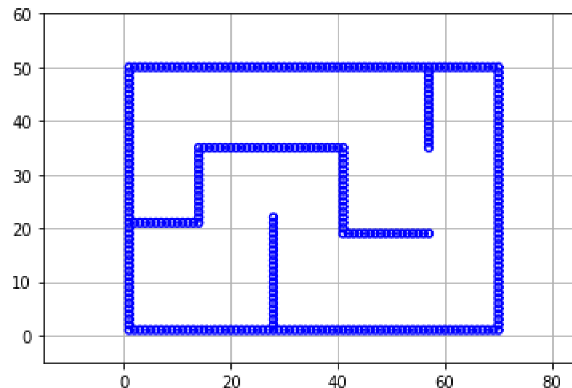

Figure A.2: Second map used for the performance analysis.

Regarding the execution parameters, as for the simulations in Chapter 4, 100 samples were collected for each algorithm and map. Start and goal positions were kept constant throughout the simulations, as well as operational parameters such as expand distance and maximum number of iterations. Since the parameters employed by both algorithm are very similar yet not equal, trivial adaptations were necessary. Remarkably, the radius used in the listing of the near nodes to a particular node in the tree, and also for the culling neighbours process in the RRT^X, is an arbitrary value in the RRT*

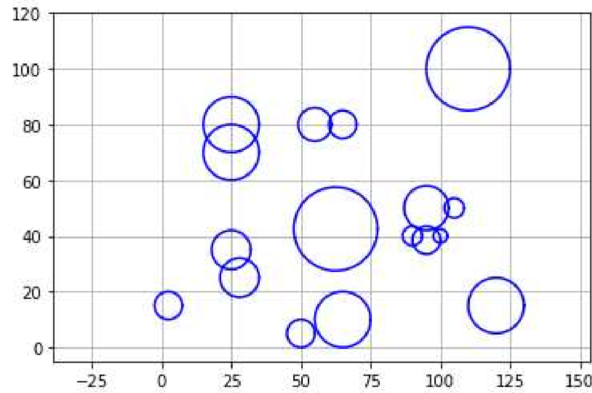

Figure A.3: Third map used for the performance analysis.

routine while in the RRT^X it is computed through the Lebesgue measure and the estimated available free space in the map. When properly set, nevertheless, the values virtually coincide.

Another relevant parameter is the number of iterations, crucial to investigate the iteration time. For the first and third maps, the algorithms were configured to execute 1000 iterations, which is notably beyond the necessary to find successful path in the referred maps, yet ideal to evaluate the growth, if there is any, of the computational time while the tree spans and increases in number of nodes. For the second map, due to it being slightly more demanding for both algorithms, the maximum number of iterations was set to 1500 iterations. Furthermore, the success rate for this map was also registered, while for the other two maps this measurement was proven unnecessary due to a 100% success rate by both algorithms.

A.1.1 First Map

The first map, being the one that simulates the work environment of the rover, was the most significant one in terms of analysis. The starting point utilised in the map by both algorithms was placed at $(15, 2.5)$, while the goal at $(25, 40)$. As expected, especially considering the high number of it-

erations and overgrowth of both trees, the average cost of the paths found is approximately the same and considerably close to the optimal solution, with a error margin of less than 10%, as was also observed in the simulations presented in Chapter 4. Regarding the overall computational time taken to finish the 1000 iterations, the average for the RRT^X corresponded to approximately 31.51 seconds, while the RRT* registered 22.04 seconds. The most expressive result can however be observed in Figure A.4, which represents through plotted points the average time employed to compute each one of the 1000 iterations, considering the 100 samples executed. Such graph complement the first graph shown in Figure 4.34.

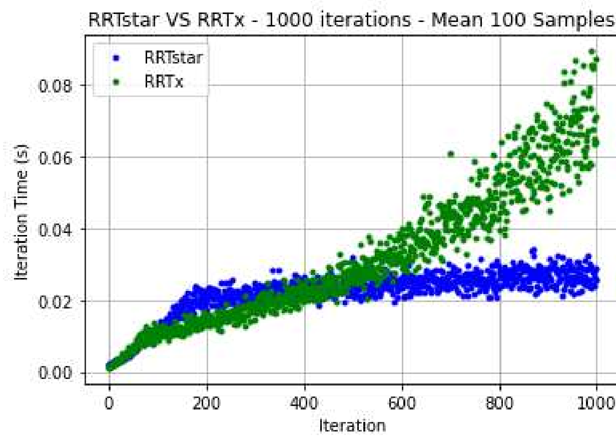

Figure A.4: Iteration time over number of iterations for the first map.

As it can be observed by Figure A.4, the RRT^X algorithm in green, presents a slight exponential tendency of growth, while the RRT* presents a linear, almost constant, tendency. Below 700 iterations, nonetheless, the behaviour of the two algorithm can be considered substantially close.

A.1.2 Second Map

The second map, as already stated, presented itself as a more difficult task for both algorithms. Upon gathering and analysing the data, it was verified

that, for the parameters set for both algorithms, only 44.62% of the RRT^X executions were successful, against 46.18% of the RRT* executions, which once more indicates a fairly similar behaviour by the two motion planning algorithms. The costs were also similar, with the RRT^X only marginally more efficient, reinforcing the characteristic of approximate optimality observed on both algorithms in the previous test. Concerning the overall time, the RRT^X in the present test displayed a significantly greater computational time, of nearly two minutes, while the average time for the RRT* remained close to 30 seconds. Figure A.5 shows the plotted points regarding the iteration time for second map test.

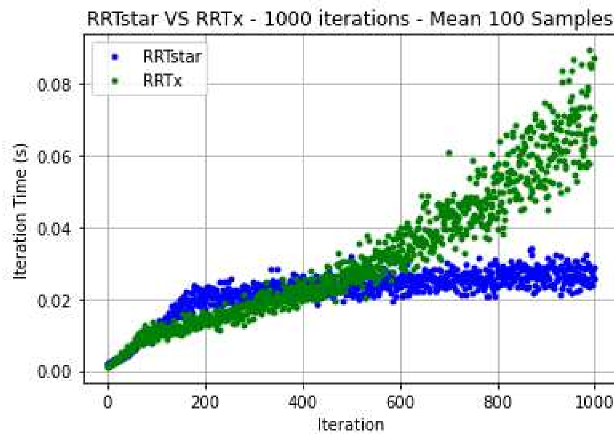

Figure A.5: Iteration time over number of iterations for the second map.

It is worth to note that only the executions where both algorithms were successful were registered and used for the data analysis, *i.e.* all the data utilised for the computation of the average values of cost, overall time and iteration time refers to 100 successful executions, obtained out of 511 attempts. Regarding the behaviour observed in the graph, the discrepancy between the RRT^X and the RRT* is considerably more pronounced, albeit below 600 iterations the behaviours are relatively close.

A.1.3 Third Map

Finally, the last test performed applied the map presented in Figure A.3. The obstacles in the map were generated randomly, however the same map was employed for every sample and to both algorithms. The general results obtained from the test were comparable to the ones obtained from the test with the first map, except that the computational time for the RRT^X in the present example is, on average, six times greater than the one from RRT*. The two of them are, however, significantly small, with RRT^X presenting an average time of 12.24 seconds while the RRT* presents 2.14 seconds. The time per iteration for this test can be observed in Figure A.6.

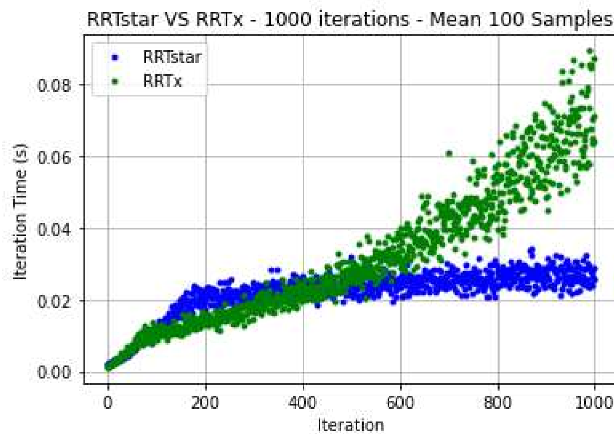

Figure A.6: Iteration time over number of iterations for the third map.

From the Figure A.6, it can be observed that the iteration time values for both algorithms are moderately smaller than the ones presented in Figure A.4 and Figure A.5. Moreover, the difference between the two tendencies is more accentuated, insomuch that maximum value to which the behaviours of the two algorithms is sufficiently close is smaller, around 500 iterations.

A.1.4 Conclusion

Considering the results attained with the tests and the analysis performed, it can be concluded that the behaviour of the RRT^X algorithm used in this thesis, despite being different from, *i.e.* less computationally efficient than, the RRT*, is sufficiently close to it under a certain limit of operation, of about 600 iterations. This limit was applied to the final version of the code and is implicit in the results presented in Chapter 4. Since a description of the algorithm complexity compared with the RRT* supported by data is not available on [27], only theoretical and qualitative descriptions, it remains uncertain whether the exponentiality of the results is a consequence of any internal routine written for this specific thesis but not present in the original algorithm, as it was the tree integrity enforcer routine previously removed, mentioned in Section 4.4.2. Further investigation in this sense may be necessary for a better understanding and improvement of the algorithm, with the goal of achieving a more robust, efficient and reliable dynamic motion planning routine.