Nuno Miguel Pereira Muchaxo

Master in Electrical and Computer Engineering at Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia

UAV Navigation System for Prescribed Fires

Dissertação para obtenção do Grau de Mestre em

Engenharia Eletrotécnica e de Computadores

Orientador: Professor, José António Barata de Oliveira, NOVA University of Lisbon

Co-orientador: Investigador, Francisco Cardoso Marques, NOVA University of Lisbon

Júri

Presidente: Doutora Ana Inês da Silva Oliveira
Arguentes: Doutor João Almeida das Rosas
Vogais: Mestre Francisco Cardoso Marques

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

November, 2020

UAV Navigation System for Prescribed Fires

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Acknowledgements

I would like to express the deepest appreciation to my adviser, professor José António Barata de Oliveira, co-adviser, professor Francisco Antero Cardoso Marques and everyone in the RICS group. Thank you for helping me on such a challenging task, as well as, help me to improve on a professional level. Without you, it had not been possible to perfect this dissertation at a very good level.

I would also like to show special gratitude to my family who has always been there to support me throughout all these years, which were some of the most challenging years of my life. Thank you, Francisco Muchaxo, Cristina Pereira and João Muchaxo for always being there by my side.

In addition, I would also like to show special gratitude for my best friends and everyone who helped me throughout this master's degree. Without you, it would not have been possible to finish it.
Thank you.

Abstract

Since the beginning of mankind, a lot of fires have happened and have taken millions of lives, whether they were human or animal lives.
On average, there are about twenty thousand forest fires annually in the world and the burnt area is one per thousand of the total forest area on Earth. In the last years, there were a lot of big fires such as the fires in Pedrogão Grande, Portugal, the SoCal fires in the US coast, the big fire in the Amazon Forest in Brazil and the bush fires in Australia, later 2019.

When fires take such dimensions, they can also cause several environmental and health problems. These problems can be damage to millions of hectares of forest resources, the evacuation of thousands of people, burning of homes and devastation of infrastructures. When a big fire starts, the priority is the rapid rescue of lives and then, the attempt to control the fire. In these scenarios, autonomous robots are a very good assistance because they can help in the rescue missions and monitoring the fire. These autonomous robots include the unmanned aerial vehicle, or commonly called the UAV.

This dissertation begins with an intensive research on the work that has already been done relative to this subject. It will then continue with the testing of different simulators and see which better fits for this type of work. With this, it will be implemented a simulation that can represent fires and has physics for test purposes, in order to test without causing any material damage in the real world.

After the simulation part is done, algorithm testing and bench marking are expected, in order to compare different algorithms and see which are the best for this type of applications. If everything goes according to plan, in the end, it is expected to have an autonomous navigation system for UAVs to navigate through burnt areas and wildfires to monitor the development of these.

Keywords: Forest Fire, Unmanned Aerial Vehicle, Navigation, Planner

Resumo

Desde o início da humanidade muitos incêndios têm acontecido e têm levado milhões de vidas, quer estas sejam humanas ou animais. Em média, no planeta, existem cerca de vinte mil incêndios florestais anualmente e a área queimada é um por mil da área total de florestas do mundo e na última década, houveram grandes incêndios. Alguns destes são os de Pedrogrão Grande, em Portugal, os incêndios no sul da Califórnia, na costa dos EUA, o incêndio que deflagrou na floresta Amazónia, no Brasil e os incêndios na Austrália, no final de 2019.

Quando os incêndios assumem estas dimensões, podem vir a causar vários problemas ambientais e de saúde. Estes problemas podem ser danos a milhões de hectares de recursos florestais, a evacuação de milhares de pessoas e podem haver habitações e infraestruturas ardidas.
Quando um grande incêndio começa, a primeira prioridade é o resgate rápido e de seguida a tentativa de controlar o incêndio. Nestes cenários, robôs autónomos são uma excelente assistência. Estes robôs incluem o veículo aéreo não tripulado, o UAV.

Esta dissertação começa com uma intensa pesquisa sobre o trabalho já realizado em relação a este tema. De seguida, vários testes irão ser realizados para testar diferentes simuladores e ver qual melhor se adapta ao trabalho que se irá realizar. Com isto, será implementada uma simulação que consiga representar um incêndio e suporte várias fisícas do mundo real.

Após a secção da simulação estar concluída, espera-se vários testes de algoritmos e comparação entre eles, para ver qual o que se adequa melhor a este tipo de situações.
Se tudo correr conforme planeado, é esperado no final desta dissertação ter-se um sistema de navegação autónoma para UAVs percorrem áreas florestais e ser possível monitorizar incêndios.

Palavras-chave: Incêndio Florestal, Veículo aéreo não tripulado, Navegação, Planeador

Contents

List of Figures

List of Tables

Acronyms

ACO	Ant Colony Optimization.
AI	Artificial Intelligence.
AirSim	Aerial Informatics and Robotics Simulation.
API	Application Programming Interface.
CPU	Central Processing Unit.
DDRT	Dynamic Domain for Rapidly exploring Random Trees.
DWA	Dynamic Window Approach.
ENU Frame	East, North, Up Frame.
FCU	Flight Controller Unit.
GIS	Geographic Information System.
GPS	Global Positioning System.
GUI	Graphical User Interface.
IMU	Inertial Measurement Unit.
Lidar	Light Detection and Raging.
MOOS	Mission Oriented Operating Suite.
MRS	Microsoft Robotic Studio.
NED Frame	North, East, Down Frame.
OMPL	Open Motion Planning Library.
OS	Operating System.

PGM	Portable Gray Map.
PID	Proportional Integral Derivative.
PNG	Portable Network Graphics.
PRM	Probabilistic Road Map.
PSO	Particle Swarm Optimization.
ROS	Robotic Operative System.
RPCSC	Remote Procedure Call Style Communication.
RPG	Role Playing Game.
RRT	Rapidly exploring Random Trees.
SA	Simulated Annealing.
SAA	Sense and Avoid.
SRDF	Semantic Robot Description Format.
UAS	Unmaned Aircraft System.
UAV	Unmaned Aerial Vehicle.
UE4	Unreal Engine 4.
URDF	Unified Robot Description Format.
UTM	Universal Transverse Mercator.
XML	Extensible Markup Language.
YAML	YAML Ain't Markup Language.

Chapter 1

Introduction

This is the first chapter of this dissertation. Initially it is given the framework and the research problem that allowed develop this work and write this dissertation.

The objective and the reason why this subject was chosen are also in this chapter. In the end there is the dissertation structure that shows the reader how this work is divided and a brief explanation of each chapter is given.

1.1 Framework and Research Problem

When fires take big dimensions, they can also cause several environmental and health problems such as damage to millions of hectares of forest resources, the evacuation of thousands of people, burning homes and the devastation of infrastructures. When a big fire starts, the priority is a rapid rescue and then the attempt to control the fire [1].
In these scenarios, autonomous robots are a very good assistance. These autonomous robots include the unmanned aerial vehicle, also commonly called the UAV.

Nowadays, UAVs have been used to perform a lot of missions, such as exploration of the moon and Mars [2], surveillance in areas affected by war [3], to predict meteorological conditions [4], among others.
More recently, these aircrafts have also been developed for rescue missions in earthquakes, tsunamis and forest fires, as they have the advantage of rapid maneuver and simple maintenance [5].

This dissertation presents a UAV navigation system to monitor forest fires that is able to provide a better navigation solution at low altitudes (above tree lines).
UAVs have been recently tested in several fire monitoring missions [1, 6], however in all of these examples a human operator remotely controls them and the UAV must be in their line of sight which also involves a great danger for the operator.

It is not new what UAVs can do in the event of a natural disaster. However, most of the systems are remotely controlled and don't have paths planned for the UAVs to cover. The problem with path planning is finding a collision free path in an environment with obstacles. As it is known, there are already algorithms that can plan paths in a world. However, planning paths in a world with static or dynamic objects is different. Calculating this path is a complicated optimization problem, but this will allow the aircraft to get to its final destination or cover its path without hitting any obstacle.
Despite some differences, most of the algorithms are based on the same general approaches as these algorithms create a road map, or they decompose the path in cells and it is created a potential field for the UAV to cover.

In this dissertation, an autonomous navigation system for UAVs to cover areas without hitting any obstacles or going into restricted areas will be developed. The main objective is for the UAV to be able to autonomously navigate through the borders of wildfires and burnt areas in order for the fire to be monitored.

1.2 Motivation

Due to the increasing fires all around the globe and with the aim to achieve a better world, I applied for this dissertation.

The motivation behind this dissertation is the development and integration of a control system available for UAVs to monitor forest fires. This system is to be used under hazardous and demanding situations for Human beings, such as Search and Rescue Operations, surveillance and forest fire monitoring.

Another motivation to choose this dissertation is my interest in this subject. The fact that it was possible to join a personal interest with my master thesis definitely improved my knowledge in this field. Another reason, it is knowing that this dissertation will help populations and others worldwide and hopefully try to minimize the harm that forest fires can cause.

Unfortunately, and due to the pandemic, real life tests were not possible to perform. Various tests were scheduled with the Fire Department in order to test the algorithms presented in the navigation system however, these had to be postponed.

1.3 Objective

The main objective of this dissertation is to develop an autonomous navigation system available for UAVs that can monitor forest fires and help in difficult and hazardous situations such as prescribed fires and rekindle operations.

In the past months, it was developed an autonomous navigation system based on path planning algorithms for UAVs to use and cover a path, avoiding possible obstacles while also taking into account the micro climatic alterations that occur during a forest fire. This path can be given by an operator for the UAV to cover and it must be able to autonomously navigate through the borders of wildfires and burnt areas in order for the fire to be monitored.

1.4 Dissertation Structure

This dissertation is structured into seven chapters. Each one of these chapters represents a big section of this dissertation and they are described here.

- **Chapter 1:** An introduction to this work with a framework and the research problem to this dissertation are presented in this chapter. Motivation, an objective and a structure are also given;
- **Chapter 2:** This chapter is dedicated to the supporting concepts and all the scientific work that is already done related to this dissertation;
- **Chapter 3:** The simulators used are covered in this chapter. It is provided an overall understanding of the software and what is needed to do the necessary tests related to the autonomous navigation system;
- **Chapter 4:** This chapter is dedicated to the system architecture. It is explained more in depth the integration done with all the simulators and how everything is connected;
- **Chapter 5:** This chapter is dedicated to the autonomous navigation system. The parameters of the costmaps and the planners used are explained in this chapter as well as, how the UAV localization is performed.
- **Chapter 6:** Dedicated to the experimental results. These are reviewed and discussed;
- **Chapter 7:** It is given a conclusion and possible future developments and research topics related to this subject.

Chapter 2

Scientific and Related Work

This chapter presents the scientific work and the supporting concepts that are fundamental for the writing of this dissertation. This section sums up all the work that has been done relative to this subject. It is divided in sub sections accordingly to different matters.

The first section is relative to the navigation of robots and UAVs, the second section explains what path planning is and its algorithms. Here, 2D planning and 3D planning are addressed. The following section is dedicated to explaining how different altitudes can influence path planning and the last section addresses how path planning can be applied to natural disasters.

2.1 Robots and UAVs - Navigation

An Unmanned Aerial Vehicle or UAV, is an aircraft without a human pilot on board and UAVs are a component of an Unmanned Aircraft System (UAS). This UAS includes an aircraft, a ground-based controller and a system of communications between the two. The flight of UAVs may operate with various degrees of autonomy: either under remote control by a human operator or autonomously by on-board computers. This dissertation is focused on UAVs autonomously operated.

Compared to aircrafts where human assistance is needed, UAVs were originally used for missions where humans would be in some sort of danger. While they originated mostly in military applications, their use is rapidly expanding to commercial, scientific, recreational, agricultural, and other applications, such as policing and surveillance, product deliveries, aerial photography, among others, as it is referenced in [7]. In this chapter it is discussed about various types of navigation and Behaviour-based Architectures.

Starting of with the control and navigation, there are various types of UAVs. In this dissertation the work is done with drones or multi-rotors, but there are also fixed-wing, rotary-wing and flapping-wing UAVs. For the UAV to be able to locate itself and avoid obstacles, tasks needed to be created. These tasks are vision based and can be separated in 4 categories:

- Visual Navigation;
- Vision-based Control;
- Vision-based Tracking and Guidance;
- Vision-based Sense-and-Avoid (SAA).

Visual Navigation is the task that determines the aircraft's position and orientation. Therefore, sensors are required for the CPU in the aircraft to measure its state and sense the flight environment. Nowadays most of the aircrafts have IMUs (Inertial Measurement Unit) and a GPS (Global Positioning Systems) to better correct its position and the most recent ones also start to have a vision system. Having sensors all around it, if the GPS signal is low, the aircraft can better estimate where it will be and keep a better track of its positioning, as it is possible to see in figure 2.1.

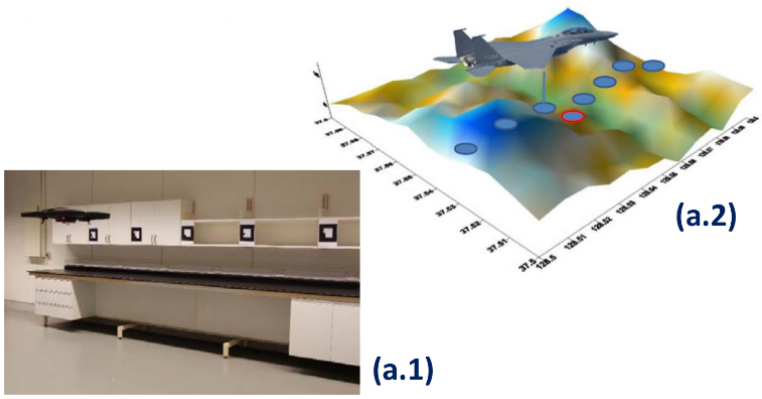

Figure 2.1: Visual Navigation on an aircraft [8]. The multi rotor is inside a room, therefore it doesn't have GPS. It is using visual navigation to locate where other objects are.

The next task is referred as Control Task, also vision-based. In this task the aircraft's position and orientation are controlled based on the information captured by sensors and then processed by algorithms. This technique was first used in 1990 and since then, several solutions have been proposed in order to address operations such as the aircraft's stabilization and to maintain a certain altitude or directional speed, as it is represented in figure 2.2.

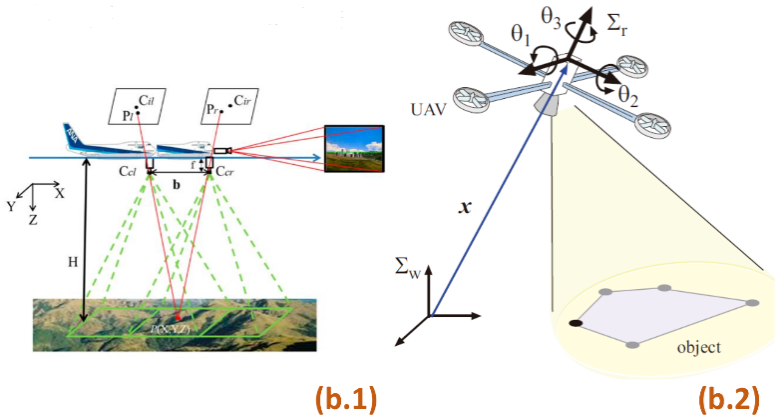

Figure 2.2: Vision-based control task [8]. The position and orientation of the UAV are controlled based on the information captured by sensors.

Another vision-based task is the Tracking/ Guidance based on its sensors. In this task, the aircraft is designed to perform a flight based on relative navigation with respect to a target that may be standing still or moving. This is defined by a series of visual references or features. In this task, represented in figure 2.3, the system must be able to detect a visual reference or a referenced target in order to perform its flight.

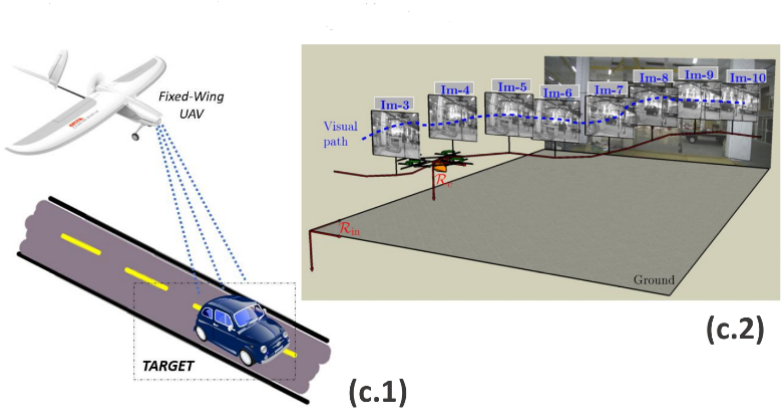

Figure 2.3: Vision-based Tracking/ Guidance [8]. It is shown an aircraft locking on to a target and following it, in this case a car.

The last vision-based task in [8] is the Sense-and-Avoid Task or SAA Task. This is a completely autonomous navigation task that requires the ability to detect and avoid obstacles. This task in particularly requires multiple cameras and sensors spread all across the aircraft to detect possible collisions with objects, as it is possible to see in figure 2.4.

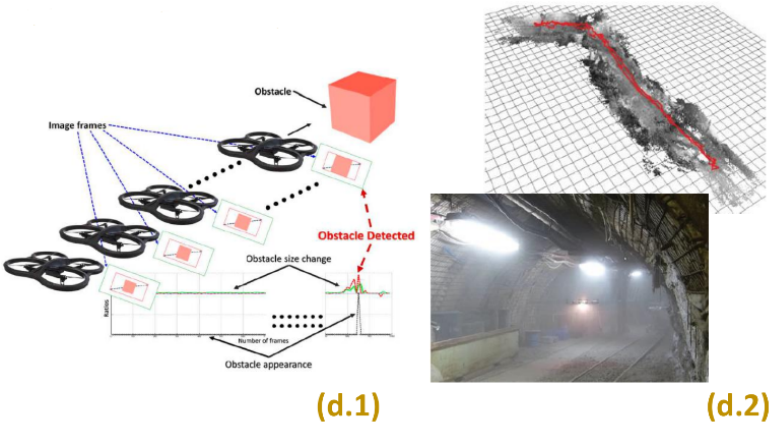

Figure 2.4: Vision-based Sense-and-Avoid [8]. The UAV is using the SAA task where it can navigate autonomously and it has the ability to detect and avoid obstacles.

After this overview of the various navigation tasks and their influence on UAVs, let's now move on to the different architectures. Starting of with Behaviour-based architectures, this concept started from the conventional artificial intelligence as combination with automation. In this architecture the control function is componentialized, which means it is distributed to a variety of function items. In a traditional control problem, the signal is generated by a control law in response to some sort of feedback from the object or the environment, the process is seen as a whole.

In behaviour-based architectures the signals and their various responses are created in terms of behaviours. The control process is defined by its control functions, being each one, a single behaviour. A behaviour is a function between the sensing input and the action output internally and a segment of the signal externally [9]. A behaviour is generally related to a set of parameters, usually representing the state of the agent, the objective or the performance. Behaviours can be classified in three categories:

- Control law based behaviours
- Signal based behaviours
- User operation behaviours

Control law based behaviours can be executed by a traditional control principle. On the other hand, signal based behaviours are generated by certain control law as a response to a sensor's input. The executor plays the controller role and the mechanism is in closed loop. At last, User operation behaviour is based on an experienced segment of signal. The executor directly generates a period of signal based on an experience that is stimulated by some event.

2.2 Path Planning and its Algorithms

This section focus on Path Planning algorithms. This is the main focus of this dissertation as different algorithms can influence a lot on how autonomous aircrafts behave.

2.2.1 Path Planning and Trajectory Planning

Let's start by addressing path planning and explaining what it is. UAV path planning refers to the optimal trajectory between Point A and Point B. The main purpose of a optimized path plan is to find a safe flight path for the UAV between two points. This flight path needs to be covered with the minimum energy consumption for the UAV's mission to be completed optimally. In most of the situations, the flight path is getting from point A to point B, but the essence of path planning is to find the work-space according to an optimization criteria. Examples are minimum working cost, shortest covered path, shortest covered time, among others.

In [10], mathematically, path planning is the function $\delta : [0, T] \rightarrow R^3$ of bounded variation, where $\delta(0) = X_{init}$ and $\delta(T) = X_{goal}$, if there is a process Φ, that can guarantee $\delta(\tau) \in W_{free}$ for all $\tau \in [0, T]$, then δ is the Path Planning.

Now, mathematically, the optimal path planning has a different explanation. Given the following path planning problem $(X_{init}, X_{goal}, W_{free})$ and the following cost function $c : \sum \rightarrow R \geq 0$, where $\sum$ is the sum of all paths. If the condition to find a path is plausible δ' and $c(\delta') = min\{c(\delta)\}$, where δ is the set of all feasible paths, then δ' is the path planning and Φ' is the optimal path planning.

To find the optimal path for the UAV, a lot of work and studies have been done. There are many algorithms and these fall under many categories. In figure 2.5 it is possible to see the path planning research span of this dissertation. All of the existent path planning algorithms are not addressed, as there are too many, in this dissertation it will be just covered the ones that are useful and important for this dissertation.

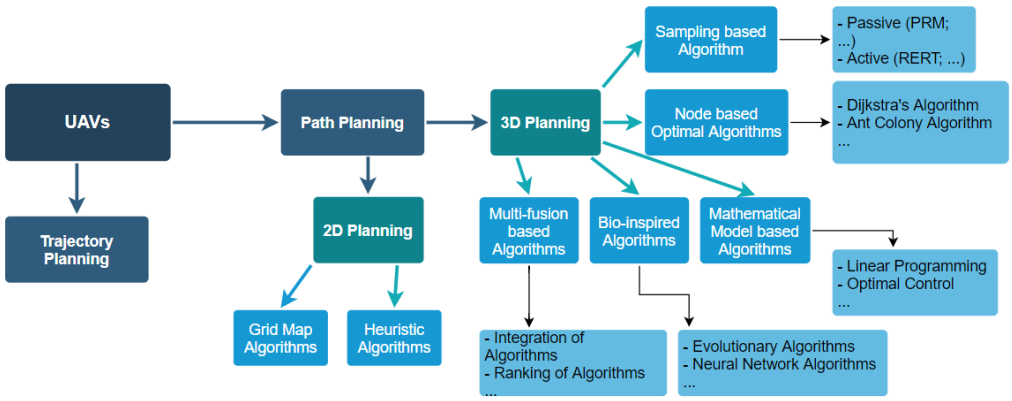

Figure 2.5: Path Planning research span diagram;

In the beginning of the diagram in figure 2.5, it is possible to see a separation between Trajectory Planning and Path Planning. The distinction between the two is never addressed properly in other related papers so, in this dissertation its difference will be properly explained.

Above, it is explained what path planning is, and trajectory planning although it may sound the same it is very different from path planning. There are several written papers that bring up the discussion between these two terms, [11] and both are intimately related.

As the term path planning is already explained, trajectory planning refers to finding a solution from the UAV's path planning and determining the best way to move the aircraft along its path. This path can either be continuous, curve or even discrete with line segments. The trajectory of the path can be described mathematically as a twice-differentiable polynomial and trajectory planning consists of finding a smooth and continuous trajectory for the UAV to move along the planned path.

2.2.2 2D Path Planning

Now, that the definitions of trajectory and path planning are explained, the categories under this type of planning can be analyzed and explained.
As it is possible to see in Fig.4, Path Planning is sub-divided in 2D Planning and 3D Planning. 2D Planning is the first to be explained and there are three algorithms based on grid maps (DWA, Probability Map and Grassfire) and two algorithms based on heuristic functions (Simulated Annealing or SA algorithm and A* algorithm).

There are plenty of 2D Algorithms, nevertheless these five were the ones where most part of the work was concluded and are the most important in 2D Planning, as they show better results.
Next, it is given a brief explanation on how these algorithms work.

Starting with the DWA algorithm or Dynamic Window Approach algorithm, the idea of this algorithm is to create discrete samples of the robot's velocity. For each of these samples a simulation is performed for the current position of the robot to try to predict what would happen if the robot would move with the help of that sample. For each path or trajectory, a score is created and evaluated. This score takes into consideration characteristics such as the proximity to the goal, to obstacles, its speed, among others, so the algorithm can pick the trajectory with the highest score. Knowing this, the robot ends up moving in the selected trajectory.

Continuing with what is a Probability Map, [12], at the beginning of a search each node has probability zero of having a detected target in its area and the probability of a target to be in any given area is given by an initial probability map. A good method for planning a search while measuring its effectiveness is doing a discrete search. This generates a probability grid containing N discrete nodes. With N nodes, it is easier to approximate the probability distribution in the searched area.

The probability map is generated considering many factors. These factors are for example the last seen position and the probable target behaviour. As mentioned above, the probability grid is the probability map that is stored and updated with the factors mentioned. This grid records whether or not all of the possible target locations have been searched. To update the map it is required access to both the camera calibration and the UAV telemetry.

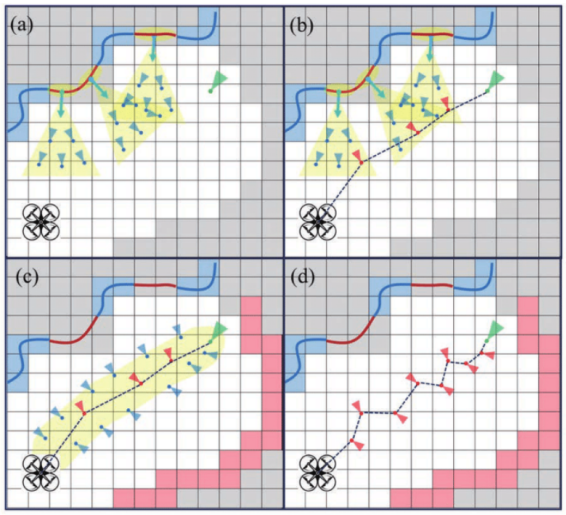

Figure 2.6: Algorithm based on PRM [13]. The UAV is calculating the cost of each cell as it is making its way to the final destination.

Moving on to the next example of an algorithm based on grid maps is Grassfire. Again, the objective of this path planning algorithm is to find the shortest path from one point to another. There are innumerous methods to do this, however a lot of these methods don't guarantee they will find the shortest or the quickest path and a lot of them are methods of trial and error.

One method that is guaranteed to find the shortest path is the breadth-first search, which explores evenly out from the 'start' cell until the 'end' cell. One of the simplest types of this search algorithm is the Grassfire algorithm, also referred as a fast wavefront expansion. The main idea behind this algorithm is to mark each node (cell) with its distance from the start cell.

As this algorithm runs, it finds new adjacent cells at the same distance (or depth) as the previous one, in which it only advances to a different distance, if only all nearby cells are explored.

Next comes the algorithms of 2D planning, based on heuristic functions. These are named Simulated Annealing or SA Algorithm and A* algorithm. There are plenty of 2D Algorithms based on heuristic functions however, these two were most used in the 2D Planning.

Starting with the SA algorithm, [14], this is an optimization algorithm that can process cost functions with various degrees of non-linearities, discontinuities and stochasticity. This algorithm can process arbitrary boundary conditions and constraints imposed on its cost functions.

Almost all of the path planning algorithms are based on grid map searching. However, there are some optimization problems that are not easily solved by the traditional approaches, as they are performed in a two dimensional space and the minimizing cost function has a complex topology, in which may present a local minimum.

This is where the SA algorithm enters and has an advantage comparing to others. In this algorithm the path can be represented as poly-line, a bézier curve and a spline interpolated curve.

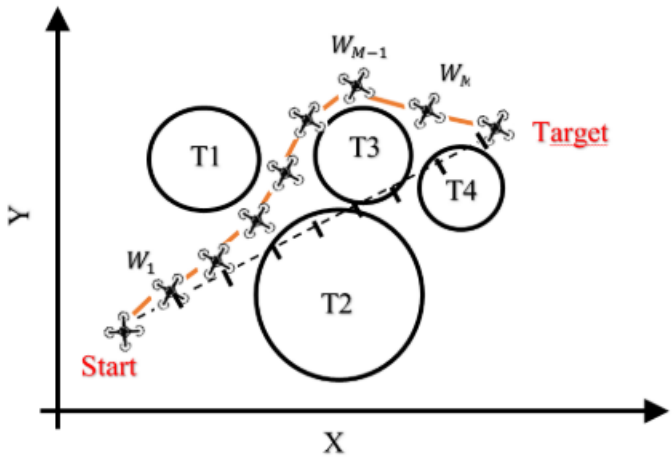

Figure 2.7: 2D Path Planning Algorithm [15]. It is possible to see the path chosen by the UAV to avoid the obstacles (T1, T2, T3 and T4) and get to its final destination in a 2D world.

As it is concluded in [14] all these three representations reach a global minimum. This algorithm implements a loop, to be more specific an annealing loop that explores the minimum, the maximum and average cost for a given temperature in a cost map.

Another 2D path planning algorithm is the A* algorithm [16], this algorithm is commonly used in engineering and it is possible to see it in action, in figure 2.7.

Summing it up, first, this algorithm calculates the cost of all nodes that can be reached from the current position node (the node where the UAV is). With this, the algorithm evaluates the point that has been searched through the heuristic information and then chooses the node with the minimum cost as the new expanding node (next UAV's location). The algorithm then repeats this process until one of the expanding nodes is the target point, or the final position of the UAV.

2.2.3 3D Path Planning

Now that 2D path planning algorithms are explained lets move on the diagram to the 3D Path Planning category. 2D path planning algorithms are not able to deal with complex 3D environments, where there are a lot of structures and uncertainties. Therefore 3D path planning algorithms are needed nowadays, especially in complex environments such as cities, forests and caves, as it is depicted in figure 2.8.

Figure 2.8: 3D Complex Environments (a forest, a cave and a city) [10];

Path Planning in 3D environments has a great potential, but it is expected the difficulties to increase when comparing to 2D path planning. It is much more difficult because of the dynamic and kinematic constraints these ambients have to offer. Accordingly, in order to plan a collision free path through rough environments mathematic tools are used to model these constraints and store a lot of data.

As it is possible to see from the diagram presented in figure 2.5 there are a lot of algorithms under 3D Planning, such as:

- Sampling based Algorithms;
- Node based Optimal Algorithms;
- Mathematical Model based Algorithms;
- Bio-inspired Algorithms;
- Multi-fusion based Algorithms.

As all of these algorithms are under 3D Path Planning, a brief explanation will be given to all of them. For covering this subject a lot of papers were researched, but the one that better explains it and covers this matter is [10].

Starting with **Sampling based algorithms**, this method needs to know in advance the work-space where the UAV will operate. As the name of this algorithm suggests, it samples the environment as a set of nodes. Next, it maps the environment and searches randomly to find an optimal path.

Sampling based Algorithms can be divided in two sub-categories, Passive and Active. Passive are algorithms based on Probabilistic Road maps (PRM). These maps generate a road map from the start node to the goal node. With this, a set of paths are created, resulting in a combination of search algorithms needed to fulfil the task.

On the other hand, Active algorithms can form a skeleton to reach the objective, all by its own processing procedure. One example of this algorithm is Rapidly-exploring Random Trees. In [10], there are a lot of algorithms in which cannot independent generate a single path, as though they are classified as passive.

Examples are 3D Voronoi [17], Rapidly-exploring Random Graph [18], PRM, K-PRM, S-PRM [19], Visibility Graphs, Corridor Map, among others. Examples of the Active category are RRT, Dynamic Domain RRT(DDRRT) [20], RRT-Star(RRT*) [21], Artificial Potential Field, among others.

Now, moving on to **Node based Optimal Algorithms**, from the name it is possible to understand how this algorithm works. They generate a path based on a set of nodes. Giving a more in depth look of Node based Optimal algorithms, they share the same property as the Sampling based ones. They search through a set of nodes on a graph or a map. These types of algorithms share a special form of dynamic programming. When the graph or map is built, they define a cost function and after that is when they search each node to find the path with the minimum cost.

Examples of this type of algorithms are the Dijkstra's algorithm [16, 22], Theta* [23], Lazy Theta* [24], Lifelong Planning A* (LPA) [25], among others.

Next in the diagram, is the **Mathematical Model based Algorithms**. This type of algorithm includes for example, Linear Programming and Optimal Control. Both of these methods model the environment considering kinematic and dynamic constraints. With this, the cost function is bounded with all the equations required to achieve an optimal solution. The figure 2.9 shows a diagram where is possible to see the process of a Mathematical model based algorithm.

There a lot of examples of this type of algorithms. Some are Mixed-Integer Linear Programming [26], Binary Linear Programming [27], Non-linear Programming [28, 29] and all of these have one thing in common. These algorithms consider all of the available factors and then the cost function is defined based on the current selection until an optimal path is found.

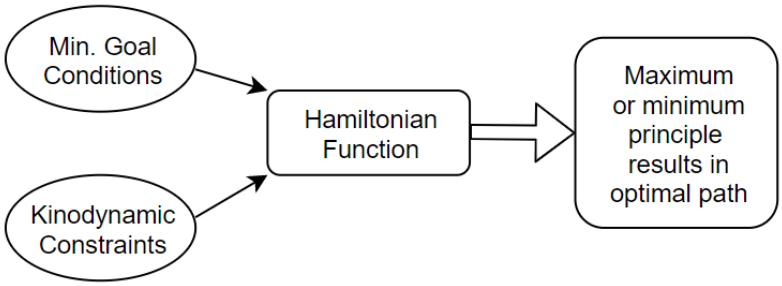

Figure 2.9: Procedure of Basic Mathematic model based problem [10]. With circular shaped forms it is possible to see the inputs of the Hamiltonian Function, in which will result in an optimal path.

Next, are the **Bio-inspired Algorithms**. As the name references, these algorithms are 'inspired' in biological behaviour. Instead of giving more importance into building a complex model of the environment, these algorithms focus on the searching method.

According to [10], Bio-inspired algorithms can be divided into Evolutionary and Neural Network algorithms, because of the fact that they do a deep analyze at different levels. Evolutionary algorithms are composed by many other algorithms such as genetic [30], Particle swarm optimization or PSO Algorithms [31], memetic algorithms [32] or Ant Colony Optimization algorithms [33].

All of these start by selecting individuals randomly and assigning them to be the first generation. Then, they take into consideration the goal, the environment and other constraints and the planner of each algorithm evaluates the fitness of each individual. The following step is choosing a set of individuals as parents for the next generations. The individuals that have a better chance to be chosen are the ones that have a better fitness. Finally, the end step is the mutation and crossover step. In this step the parents reproduce, creating the child individuals with a better fitness than the previous generation. After all these steps, one individual is selected, the one with the best fitness, which is the optimal path. In the figure 2.10 it is possible to see a diagram of this process.

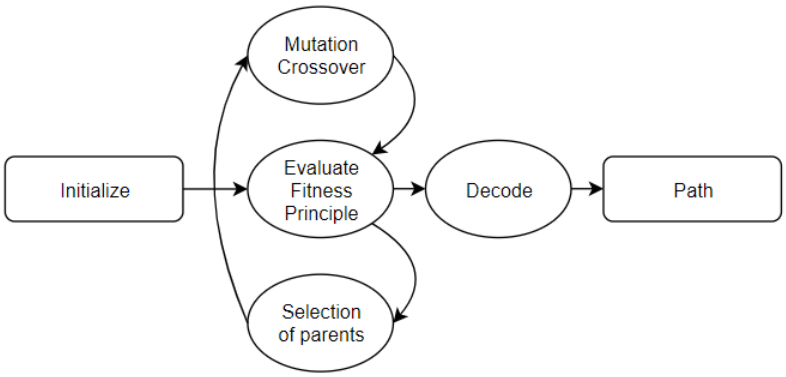

Figure 2.10: Evolution process for a Bio-inspired Algorithm [10]. With a rectangular form it is possible to see the beginning and the end of the process and with a circular form are the middle steps.

As mentioned above, there are also Neural Network algorithms. The approach these take is to generate a dynamic environment (a landscape) for the neural activities to occur. This environment also has unsearched areas that attract the UAV. A shunting equation is defined to guarantee that the positive UAV's activity can spread to this unsearched environment. With this equation, the negative activity tries to only stay local.

Bio-inspired algorithms can be very good in choosing the individuals, but because of the crossover step being random, it may cause a problem of premature convergence in which may end up compromising all the individuals.

The last type of algorithm under this category is the **Multi-fusion based Algorithms**. It is normal for 3D path planning algorithms to integrate various algorithms in order to obtain better results and better optimal paths.

Artificial potential field algorithms, without the navigation part or other methods, tend to drop into a local minimum. Another type of algorithm explained above, Probabilistic road maps, also cannot generate an optimal single path, working just by themselves.

With this, by combining several algorithms together, it is possible to achieve a better result, thus a better optimal path. This type of algorithm can deal with problems that a single one cannot. An example is referenced in [34]. He used a 3D grid to represent the environment and a 3D PRM algorithm to form a road map in an obstacle free space. With these, he also used the node based algorithm A* to achieve an optimal path.

These multi-fusion based algorithms can be split into two categories. One called Integration of Algorithms, as it is formed by integrating several path planning algorithms in order to work simultaneously to find an optimal path. The other category is called Algorithm Ranking. In this category, when one algorithm does its part and ends, another one assumes the control immediately. This category is composed of several path planning algorithms.

2.3 Different altitudes in Path Planning

This chapter is focused on how different altitudes, distinguished as low and high influence Path Planning. This is a very important chapter for this dissertation as different altitudes can influence a lot on how the autonomous aircraft behaves.

Although applied to crops and precision agriculture, this section will follow a similar structure as found in [35], as an amazing job has been done covering this subject.

As it is mentioned above, in this dissertation it is distinguished two different altitudes, one being low altitude and the other being high. Low is when the autonomous aircraft is below the treetops, high altitude is when the aircraft is above the treetops. In the figure 2.11, it is possible to see two different flight altitudes.

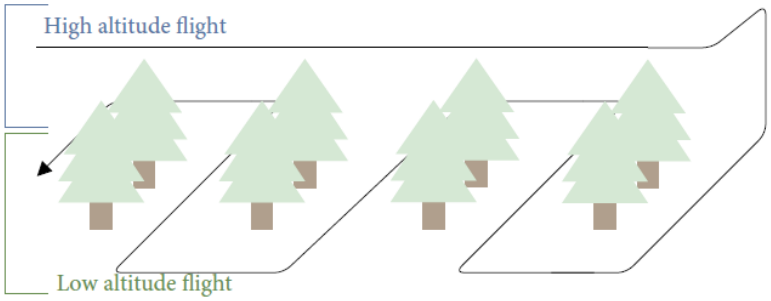

Figure 2.11: UAV paths can include flights at low and high altitudes. [35]

High altitude is an obstacle free environment as so, for covering big forests or crops a zig-zag pattern for example, can be applied. With this pattern the UAV can cover the whole area safely and with precision. Prior to high altitude, for the low altitude path planning, the best way for the autonomous aircraft to travel is to construct paths between the treetops to avoid possible obstacles and tree branches.

For the low altitude path planning, following [36], the algorithms are categorized in:

- Classical exact cellular decomposition
- Morse-based cellular decomposition
- Landmark-based topological coverage
- Grid-based methods

Starting with Classical exact cellular decomposition, this category was created to deal with polygonal shaped forms or obstacles, in which are very few in a real world environment. Alternatively Morse based methods also have one flaw. If the obstacles are in a parallel line to the propagating Morse functions, which is very common in trees, the algorithms cannot identify them, as it is explained in [37].

The last two categories, Landmark-based topological and grid-based methods are a better approach to solve this problem. As explained in the last chapter, grid-based methods work with a fixed grid map where they can operate. Landmark-based methods rely on algorithms with a dynamic slice free/ occupied decomposition.

Concerning the path planning part, as explained in the previous chapter, there are a lot of algorithms. However the algorithms that work the best with these kind of applications are the algorithm Rapidly-exploring Random Trees (RRT), the A*, the DWA algorithm and its derived methods. RRT is one of the best given its lower computational load and A* is also good given its own algorithm.

For the high altitude coverage, a location in the world can be specified with GPS coordinates to form a shape to better create the grid map.

2.4 Path Planning applied to forest fires and natural disasters

In this chapter it is applied path planning to situations that can happen in the real world, such as floods, forest fires or other natural disasters.

There are already a lot of UAV-based sensing systems to handle and support aid in these type of situations. An example of one, is cited in [1], because in addition to using multi rotors it also uses fixed wing UAVs. The multi rotors are being controlled manually and operate as a bridge between earth communication and a cloud based processing unit, in a way that various UAVs can fly in pre-programmed paths and collect information from stationary sensors.

Nonetheless, this method has a flaw, as it is not well suited for quick reactions, or life-threatening events such as wildfires or floods. In this type of natural disasters a quick response is needed also, using multi rotors with direct communication to the ground consumes a lot of the multi rotor's battery. In these scenarios using autonomous multi rotors with the capability of independent path planning is a much more efficient solution.

Facing this, a network of multiple UAVs and multi rotors can be utilized to cooperatively perform tasks and cover a wide and highly dynamic operation field.

The paper [1] also proposes a solution to the system referenced above. It is called 'Leader-Follower coalition formation for UAV-based wildfire monitoring' and it is represented in the figure 2.12.

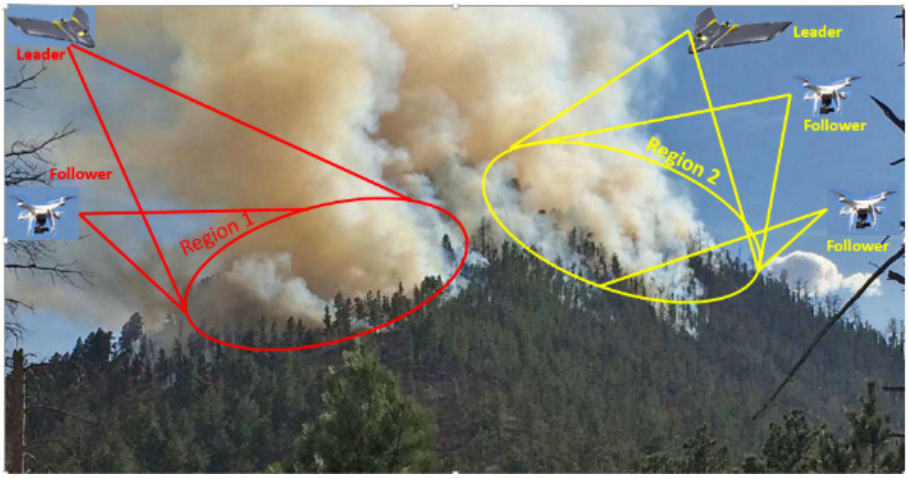

Figure 2.12: This figure shows the Leader and Follower system, where it uses a system of UAVs and multi rotors [1].

The main objective of this system is forming optimal UAV coalitions in a distributed way to cover a wide area of a wildfire. As the multi rotors have a limited communication range and cannot send their information to a ground station, there are also in the skies a set of fixed wings UAVs with sensing and imaging capabilities.

With this, the fixed wing UAVs can fly at higher altitudes and quickly search a wide area and the multi rotors can hover at low altitudes to collect data with much better resolution. Fixed wings UAVs have better flight capabilities and higher computation process, these will serve as the leader. The multi rotors will be the respective followers and will perform the video and recording tasks.

Depending on the scenarios faced it is possible to consider a different number and type of UAVs and multi rotors. The UAVs can be in a stand-by mode or parked in charging pads located near the forest and if a fire starts, the UAVs can fly towards the area and provide an initial fire map. With this, the multi rotors can enter in action and provide a fire profile. Can estimate the spread rate, flame length and the intensity with a thermal mapping of the fire.

Moving now to the multi rotor's equipment it can be attached different cameras and sensors in order to identify different things. It is possible to have:

- Visible camera: A common camera. This camera can identify the forest fire front expansion, new fire spots, damage to infrastructures, location of people, among others;
- Infrared camera: Also known as thermal camera. Can detect people and animals at night, in foggy situations or locate fire spots;
- Radar: Measures the reflection of electromagnetic waves. Can detect people and animals under trees and foliage;
- Chemical Sensors: Can detect toxic chemical compounds, smoke and the air quality;
- LIDAR: Device that measure the reflection of a laser pulse and can detect emissions and see trough the fire smoke;
- Environmental sensors: Can provide meteorological data such as temperature, humidity, wind speed and its direction, among others.

Depending on the situation it is possible to have a multitude of multi rotors with different cameras or sensors to identify and look for different things.

Looking now to the algorithms that can be applied to these type of situations, one can say that a way is to decompose path planning into two steps, as it is depicted in [38]. First, a polygonal path must be generated from the Voronoi graph by applying Dijkstra's algorithm, in which is possible to obtain the same results if a road map and A* search algorithm are applied. Then, the second step is to refine a navigable path considering the UAV's maneuverability constraints.

Chapter 3

Supporting Tools

This chapter covers all the software that is essential for this work and dissertation. It is explained why ROS is used and why this work is done with certain tools such as RViz. It is also covered why the AirSim simulator from Unreal Engine 4 is used, as it initially may not seem essential, but in the long term, it will prove to be very useful.

The last section of this chapter is dedicated to the software used to obtain images from burnt areas in the world, the FlamMap. It is explained how FlamMap is used and how essential it is for this dissertation.

3.1 ROS

The first section of this chapter is related to ROS or Robotic Operating System and it is explained what it is and what is its primary objective in this dissertation.
ROS is an open source system for robots. This system provides services that someone would expect from an operating system because ROS includes hardware abstraction, package management, low level device control and message circulation between processes. Beyond this, it also includes or provides tools and libraries for building and running code across multiple platforms. Other similar platforms are Orocos [39], Orca [40], MOOS [41] and Microsoft Robotics Studio [42], however this dissertation is done using ROS because in my master's degree I already had subjects where I worked with this software and RICS, the research group that helped throughout this year already has projects and robots based on this framework [43] and [35].

The ROS runtime is a peer-to-peer network of processes distributed across multiple machines that are communicating using the ROS communication infrastructure. This system has several communication styles, including synchronous RPC-style communication, asynchronous streaming of data over topics and the possibility to store data on a parameter server. This platform's objective is not to be a framework with the most features but instead trying to support code in robotics development and research. For greater detail, please consider reading the ROS web page [44].

ROS is structured to have code processing blocks, called ROS nodes [45] and these nodes are responsible for tasks. Each node is responsible for one task and these can exchange information. This information is transmitted through messages, which are data structures and can have various types and are in charge of the communication between the nodes [46]. These messages are published in topics that allow nodes to subscribe and view its content. This means that in order for different nodes to communicate, they need to be subscribed to a topic and in order to advertise a message to one, it must publish a message to it.
The following image illustrates this communication, figure 3.1 shows a form of communication between two nodes, a publisher and a subscriber.

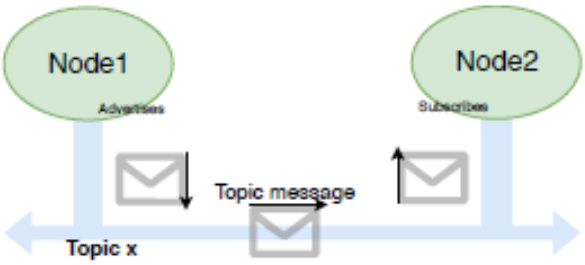

Figure 3.1: Representation of the communication between ROS nodes. [35]

ROS also allows services, which are a synchronous private way to exchange data. These services can be advertised by a node and accessed by others and these nodes are called clients and can request processes to other nodes.
In this dissertation, ROS is used as the platform to support the autonomous navigation, which will be explained later.

3.2 RViz Software

RViz is a ROS graphical interface that allows the user to visualize its robot's state and a lot more information, using plugins for different available topics [47].
In this section are only referenced the plugins used in this dissertation.
One important plugin is the *Global Options*, where the *Fixed frame* and *Frame rate* parameters are. These parameters allow the user, respectively to highlight the frame used as a reference for all the others and the frequency in which the 3D view is updated.
Another essential plugin is the *TF* plugin, which allows the user to see the position and the orientation of all the frames that compose the TF tree. TF is a ROS package that allows the user to keep track of multiple coordinate frames over time. This package maintains the relationship between the frames in the form of a tree structure. [48]

Some important parameters are *Show names*, *Show axes* and *Update interval*. These allow the user to respectively enable/ disable the name of the links, the axes and set the update time in seconds. *Robot Model* is another plugin that is worth mentioning as this allows the user to visualize the robot model according to its description from the URDF model. A URDF model is a package that contains C++ parsers for the Unified Robot Description Format (URDF). These, are files with an XML format dedicated to representing the model of the robot [49].
Some key parameters are the *Visual enabled* and the *Collision enabled* to, respectively enable or disable the 3D model and the *Collision box*.
At last, there is the *Grid* plugin. As the name suggests, this plugin allows the user to visualize a grid that is usually associated with the floor plane.
Some parameters are the *Reference frame* and the *Cell size*. These allow the user to choose the reference for the grid coordinates and the dimensions, in meters, of each grid cell, respectively.

In the figure that follows, an RViz window is displayed.

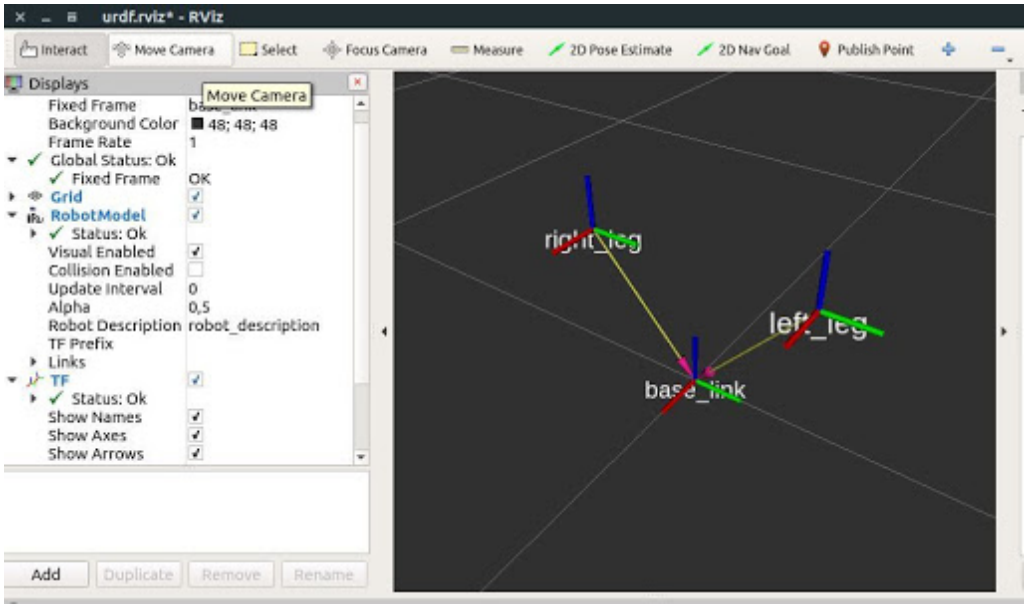

Figure 3.2: Representation of some RViz panels. [50]

In Figure 3.2 it is possible to see in the *Displays* window (left side) some plugins the user opened. On the main window is where the model and the environment are depicted.

For this dissertation, RViz is mainly used to visualize the planned paths, the environment construction and it is used as a tool for debugging and monitoring the model.

3.3 MoveIt!

MoveIt! is a software which aims to manipulate robotic arms. This tool tries to provide the user with the latest advances in motion planning, control, manipulation, 3D perception, among others [51].
This platform is open-source, easy-to-use and it is developed for advanced robotics applications like navigation and manipulation control in a 3D environment.

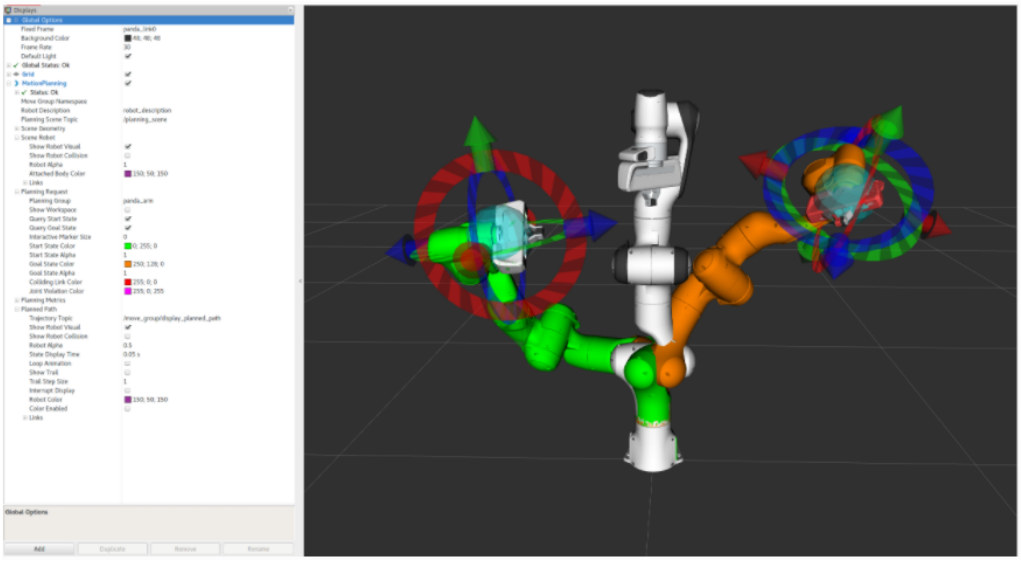

Figure 3.3: Representation of MoveIt! in RViz. [52]

Not taking the full advantage of this software, as it is mainly used to provide an obstacle free trajectory from the current position to the destination and constant monitoring of the robot model trajectory in a 3D monitoring system. This software also allows the user to configure a robot by adding components responsible for interacting with the robot actuators, also known as the robot joint controllers.

To configure MoveIt! for a robot, in this case, the UAV Matrice 100, some settings were required. The description of the robot is necessary, also known as the URDF of the robot. With this, the software is able to associate each joint value to the virtual representation of the robot. Another requirement by MoveIt! is the state of the robot, which can be represented by the model's joint states, which are the angles between each robot arm link.

To interact with MoveIt! it is provided an RViz plugin to allow some functionalities like selecting the desire positions for the robot's arms.
Another way to interact with this software is by using a C++ class called MoveGroupClassInterface, in which the communication with MoveIt! is simplified and some methods are provided to define the robot, the planning types and other functionalities.

In figure 3.4 it is depicted a basic representation of how a user can interact with RViz and how MoveIt! interacts with the RViz software and a robotic arm.

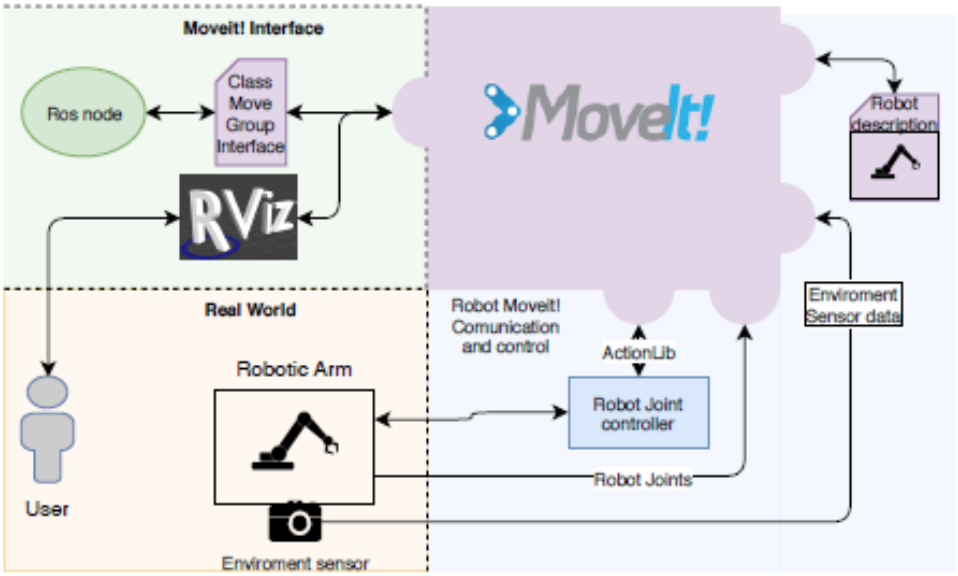

Figure 3.4: MoveIt! interaction with the RViz software and the robot. [35]

3.4 Unreal Engine and AirSim

This chapter is dedicated to two softwares that were essential for the conclusion of this dissertation. One of them being the platform Unreal Engine and the other being the simulator AirSim.
Starting with the Unreal Engine, this is a game engine developed by the Epic Games company. In 1998, Unreal Engine was initially created with the purpose of being a first person shooter game. However, with the development of the Epic Games company, nowadays Unreal Engine is much more than that. Unreal has been used in a variety of other genres, including platforms, fighting games, RPGs, among others.
Written in C++, the Unreal Engine features a lot of portability and supports a wide range of other platforms [53].

The latest release and the one used in this dissertation is Unreal Engine 4, which was launched in 2014 and since 2015 it is free to download, with the source code available on GitHub and on its main website [54]. Epic Games allows for its commercial use but asks developers for 5% of revenues from sales.

Comparing to Unreal Engine, there was also another option, the game engine called Unity.

Unity is a cross platform game engine developed by Unity Technologies, first announced and released in June 2005. As of 2020, the engine has been extended to support more than 25 platforms. Unity is very common in mobile games and can create 2D and 3D games as well as games with virtual and augmented reality. More recently, Unity has been adopted by automotive, architecture and engineering industries to create and run various kinds of simulations [55].

Now, that some information is given about both engines, it is answered why Unreal Engine 4 is used instead of Unity. First, Unreal Engine already has some integration with the ROS software and a flying simulator was already built for Unreal. Since this was not the main focus of this dissertation, it would be easier and time would not have to be spent creating a simulator.
Furthermore, Unreal Engine 4 is a simulator that is vastly used throughout these types of applications and RICS, being a group that works a lot with robots and UAVs, already uses this software.

Figure 3.5: Representation of a world in Unreal Engine 4. [56]

Depicted in figure 3.5 it is possible to see the main window of Unreal Engine 4. Below the main window is the *Content Browser*. In this window, the user can browse for props to place in the world and can edit them. On the right it is the *World Outliner*, where the user can see all the props that are shown in the main window. The last window, below the *World Outliner* is the *Details* window where the user, for each prop can choose its rotation, its placement and even give them mobility, among others options.

The next part of this dissertation is about the simulator used within Unreal Engine, called AirSim. AirSim (Aerial Informatics and Robotics Simulation) is an open source platform for aerial and ground vehicles. This simulator was developed by Microsoft and it is built on Unreal Engine 4 with its main purposes being AI research, deep learning and computer vision algorithms for autonomous vehicles [57].

AirSim provides twelve kilometers of roads, twenty city blocks and many worlds for the user to test its algorithms. APIs are also provided for the user to retrieve data and control the vehicles in an independent platform and are available in the following programming languages C++, C#, Python and Java. This platform also supports hardware-in-the-loop with steering wheels and flight controllers, such as the PX4 software.

As it is said above and one of the reasons why Unreal Engine 4 and AirSim were chosen to work with, was because some integration with the ROS software was already done. Another reason is that AirSim has other features that other simulators did not have. Some of these features are the manual drive, the programmatic control and the computer vision mode. Starting with the manual drive, AirSim gives the user the option to plug in an RC remote control to manually control the UAV in the simulator. As the UAV requires much more keys than the standard arrow keys (used in a ground vehicle), in this way, it is much easier to control it. The programmatic control allows the user to interact with the vehicle using APIs and with these, it is possible to retrieve images, get the state of the vehicle, control it and so on. The next feature is also very useful, the Computer Vision mode. In this mode, it is possible not to have the vehicle or the physics engine active, plus the keyboard can be used to go around the scene and collect images and other data. In figure 3.6 it is possible to see a quadrotor in a city environment. It is also possible to see a thermal view, a real time segmentation of the objects and the UAV's view.

Figure 3.6: UAV in a city environment. Besides seeing the UAV in the world, it is possible to see three views, a thermal, the one with the segmentation of the objects and the UAV's view. [58]

In order to get the AirSim Simulator to work with ROS, another installation is required. Following [59], it is possible to install the package "airsim_ros_pkgs", which basically is a ROS wrapper over the AirSim C++ Client Library.
When built correctly, the ROS wrapper is composed of two ROS nodes. The first one, being the wrapper over AirSim's multi rotor C++ client library and the second one being a simple PD position controller, which it will not be used.

In order to communicate with ROS and the RViz software within these two nodes, publishers, subscribers and services were created. As the second node is not used, are only explained publishers, subscribers and services for the first node.
Some important publishers are the `/airsim_node/quadrotor/global_gps`, `/airsim_node/quadrotor/odom_local_ned` and the `/airsim_node/quadrotor/image/camera`. Respectively these three publishers give the current GPS coordinates of the UAV in the AirSim, the odometry in the frame and the image from the multi rotor's camera.
Moving on to the subscribers, there are two very important. These are `/airsim_node/vel_cmd_body_frame` and the `/gimbal_angle_quat_cmd`, which respectively give the speed of the moving vehicle and the gimbal set point in quaternion.
As mentioned above, there are also services and two important ones are the `/airsim_node/quadrotor/land` and the `/airsim_node/quadrotor/takeoff`, which respectively land and lift off the UAV.

Within AirSim, there are a lot of environments or worlds. The one used in this dissertation is the Blocks environment [60], mainly because it is lighter for the computational unit being used.

In figure 3.7 it is possible to see the UAV in the Blocks environment. There is no sky and there are few props but it is enough to test and run the algorithms. It is also possible to see three views, a thermal, the one with the segmentation of the objects and the UAV's view.

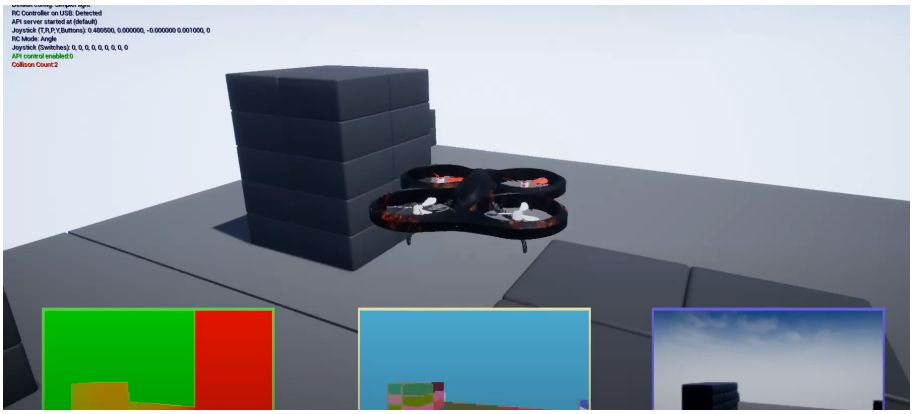

Figure 3.7: UAV in the Blocks environment. Besides seeing the UAV in the Blocks environment, it is possible to see three views, a thermal, the one with the segmentation of the objects and the drone's view.

3.5 FlamMap

This section is about the FlamMap software.
FlamMap is a fire analysis application that runs in Windows OS. This software can simulate fire behaviour characteristics, like the spread rate, flame length, the fire's intensity and so on. This application can also simulate the fire growth and the spread of the fire and its direction if some constants are given correctly, like for example, some environmental conditions, weather and the needed fuel moisture [61].

Getting more in depth, this tool describes fire behaviour for constant environmental conditions as it is said above and this is done by calculating the fire behaviour for each pixel within the landscape file. These fire behaviour calculations include as variables the surface fire spread, the flame length and the crown fire activity, which is the fire that spreads from treetop to treetop.
Recently, FlamMap included another algorithm in its software called FARSITE. This algorithm also allows FlamMap to calculate the dead fuel moisture and its conditioning in each pixel based on slope, shading, elevation and weather.
However, FlamMap does its calculation as if the environmental conditions remained constant and it does not simulate temporal variations in the fire behaviour caused by the weather, which is something they want to implement in the next versions.

In the following image it is depicted the canopy view of an area in North America. The greener the area, the most dense the vegetation is.

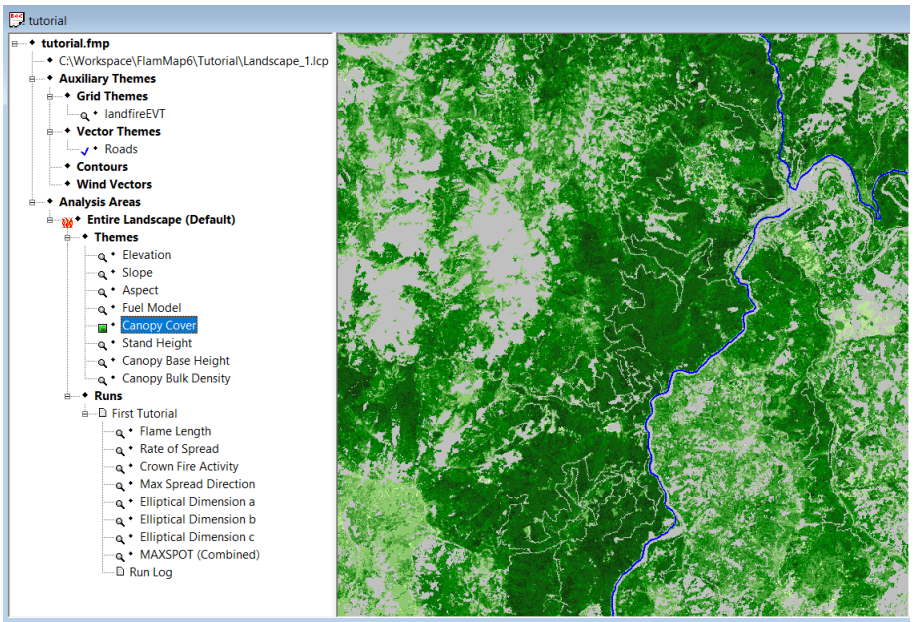

Figure 3.8: Example of a view in FlamMap program. In the left tab it is possible to see the various options, themes and outputs presented by the FlamMap. According to the output selected, this is presented in the image. The theme selected is the canopy view.

In figure 3.8 it is also possible to see other option themes that FlamMap offers, such as the slope, fuel model and elevation of the terrain.

Following [61], the way FlamMap works is by creating a variety of vectors and raster maps of potential fire behaviour characteristics, like spread rate and flame length.
The difference between a vector map and the raster map is that the vector map uses points and line segments to identify locations, while the raster map uses a series of cells to represent locations.

With both of these models, FlamMap can create various outputs based for example, on the flame length or the spread rate over an entire landscape. These maps, can then be viewed in FlamMap or exported to an image format or for later use in GIS (Geographic Information System).
Below, in figure 3.9 the output *Flame length* is selected. The red areas were the areas where the flames were more significant, as opposed to the yellow areas where they were smaller. In the image there is also a blue line representing the river and a green area near the blue line representing a zone the fire did not hit.

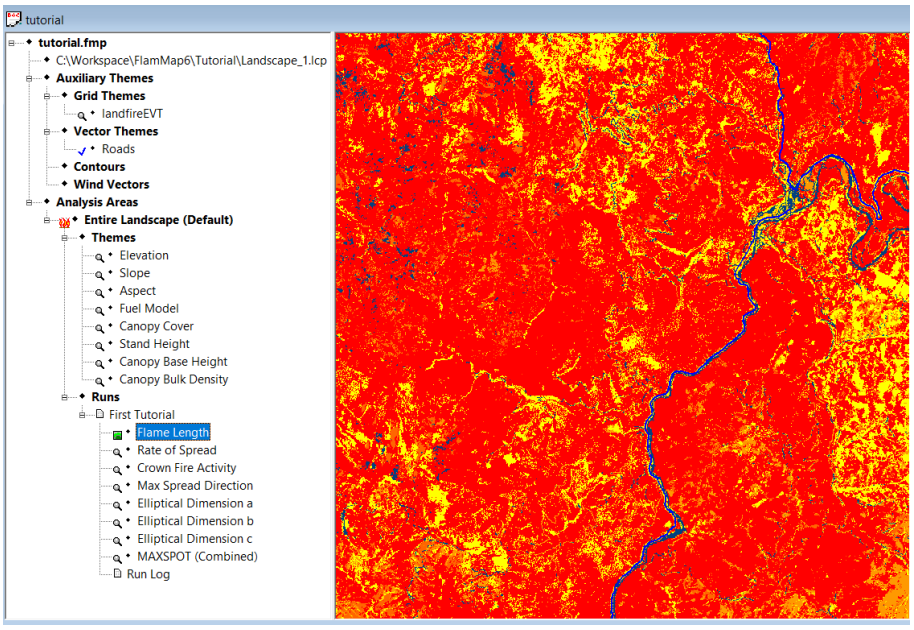

Figure 3.9: Example of the Flame Length output in FlamMap. In the left tab it is possible to see the various options, themes and outputs presented by the FlamMap. According to the output selected, this is presented in the image. The output selected is respective to the Flame Length of the fire.

Chapter 4

System Architecture

This chapter mainly focuses on the systems' description. In the first section of this chapter, an introduction is given about why this was first done using simulators and how data is acquired in the virtual simulations. It is also presented the system requirements and the sensors needed in order for the simulation to work.

Following, it is given a description of the UAV used and finally, it is explained all the integration done with the UAV and how the grid maps are created using the FlamMap software. It is also mentioned how the communication is done between different softwares.

4.1 Simulation Approach

When developing mobile robots, it is crucial not to test the implementations done in real life and with the robot that will be used. In an initial phase, as these implementations are just tests, some may go poorly and end up causing damage to the robot, to a person or can even cause property damage. This is why it is important to test in a controlled environment and by controlled environment it is meant using and taking benefit of virtual simulators. Simulating the software developed offers a quick and easy way to test new implementations done to the code that will act in real life.
With simulations, the developer has the ability to execute and test various simulations, as well as build virtual environments that represent the real world. In these virtual worlds, the developer can also debug its code and test its rigidity. However, they have a downside.

These simulations are restricted to the processing and graphical power of the computer. This means that there is a limit and a compromise between the complexity of the simulation and the processing power of the unit being used.
Another critical aspect is the time invested in representing the real world in the virtual simulation. Trying to create a more realistic world and adding physics like gravity, collisions or wind force can take a lot of time and sometimes, these can be difficult tasks as it is required to know a lot of the simulator's code in order to optimize it so the computer can handle such computations and calculus in the simulations.

Although it is very hard for the developer to 100% represent the real world in the simulation because of all the complex computations, it is possible to simplify some physics and interactions in order to maintain a reliable simulation output.
In this dissertation, a virtual simulation is necessary due to the robot's value being used in the harsh conditions of the forest fires. The objective of these simulators is to provide a more controlled environment for developing and testing before addressing the real world.

4.2 System Requirements

In this dissertation, a virtual simulation is built to help with the development of the navigation system in the UAV. The main requirements of the simulator in order to give a reliable output are depicted below. After the list that follows, a brief explanation of every single one is given.

- Availability of the equipment;
- Availability of the sensors;
- Simple physics engine;
- Communication with the software being developed;
- Low performance impact;
- Able to perform autonomous navigation with the equipment.

Starting with the first point, it had to be possible to integrate the equipment in the simulation, in this case, the UAV DJI Matrice 100. However, this isn't something already available, if the developer wanted a similar model he had to create the schematics, the meshes and the URDF, in order to be available in the RViz software.
For this, the model's schematics were found online and from there, one was able to create the URDF profile of the UAV using the software MoveIt Setup Assistant [62]. How the UAV's description was done will be better explained in the next section.

The sensors' availability in the simulations also played a significant role when choosing the platform. Knowing this, the AirSim simulator already has implemented GPS and IMU sensors, to later on, if the developer desires, they can locate the UAV and know its position. Another important sensor that was used was the laser scan sensor in RViz. This also allowed to perform the autonomous navigation with the UAV as it was possible to identify the areas where the drone could not go and navigate within its limits.
As the main purpose of these simulations were to look very similar to real life, the simulator in use already needed to have a physics engine where the computer being used to test these could handle it in terms of performance. For this, the AirSim already has options to integrate gravity, collision boxes and other effects the developer may want and they are relatively light for the specifications of the computer being used.

Another key feature is being able to communicate between both softwares, RViz and AirSim. As it is explained in section 3.4, with the AirSim ROS package, it is possible. As ROS is a low performance software, the processing unit in the UAV is able to run it and it also serves as a valuable feature in the development of the autonomous navigation. The last point of the list above is being able to perform autonomous navigation with the UAV and for this the RViz allowed the user to perform a series of tests to prove that the equipment can navigate autonomously throughout the limits and borders of the areas that were on fire.

4.3 System Description

This section is relative to the description of the system in this dissertation. Here, are described the interactions between the UAV, its components and a station computer.
In the figure depicted below, figure 4.1 it is possible to see how the connection is made between the UAV, its sensors, the software and a computer station.

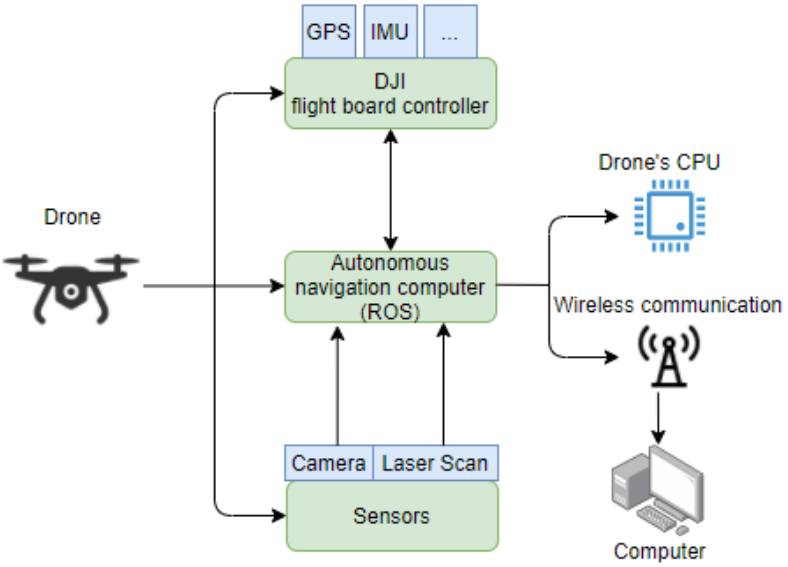

Figure 4.1: Interactions between the UAV's sensors (in blue), its components and software (in green) and a computer.

Starting with the *DJI flight board controller* or FCU (Flight Controller Unit), this is a micro-controller that runs software that has the main functionality of controlling the UAV. This unit is responsible for all the controllers for the stabilization and movement of the UAV and it also provides data such as, its position, acceleration and velocity.
This FCU is composed of various sensors. However, the most important ones for this dissertation are the GPS and IMU sensors. The GPS or Global Positioning Systems uses satellites to calculate the position of the UAV in the world. The IMU sensors or Inertial Measurement Unit, uses a combination of sensors like accelerometers and gyroscopes to determinate the robot's behaviour. It provides data like linear acceleration, orientation and the velocity of the robot. This controller is represented by the environment on the AirSim simulator.

Following the schematics above, next, is the *Autonomous navigation computer*. This box represents the software that enables the autonomous navigation of the UAV. For this, there are two possible processing units. One is the processing unit on the UAV, the CPU that acts independently from any external connection. The other is a ground station, or the *Computer* as it is represented in figure 4.1, which can be connected wirellessly to the machine. This computer must have an Ubuntu or Linux operating System with ROS installed.

The *Autonomous navigation computer* also depends or needs input from the *Sensors*, in this case, the camera and the laser scan that are represented by the RViz software. These sensors gather information from the map that is exported from the FlamMap software. The connection with the FCU also provides the system with the UAV precise location in the world, due to the GPS. The distance and path that has traveled since the beginning or lift off is tracked by the IMU sensors. The camera and the laser scan provide information about its surroundings. These sensors detect nearby obstacles and most important the burnt areas and the tree tops that serve as guidance to the UAV. The autonomous navigation software is developed using the ROS environment, which will be explained in another section.

4.4 UAV Description

This Chapter is about how the description of the UAV was built using the software MoveIt!. This chapter is not about what MoveIt! is or how does it work, that is described in section 3.3. Creating a UAV description in the MoveIt! software is a bit different than creating a robotic arm. Here, the primary node used is the move_group node. As it is shown in figure 4.2 this node merges all the other external nodes in order to provide a group of ROS actions and services for the users and developers.

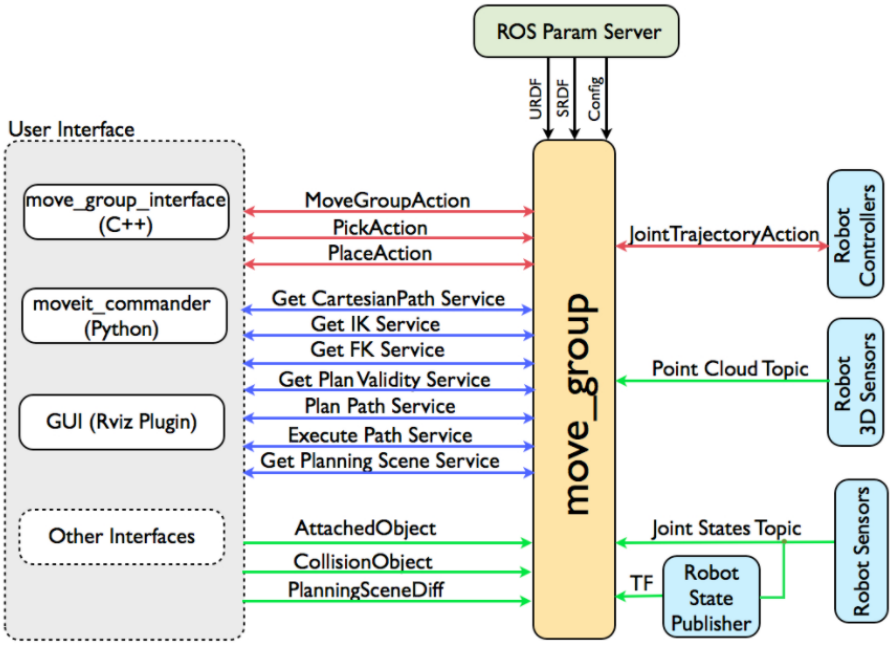

Figure 4.2: System Architecture with the integration of the move_group node. The User Interface is represented by a gray color, the move_group node by a yellow color, the ROS Parameter Server in green and in a blue are represented the sensors of the robot. [63]

When creating the description of a robot in this software the concept is to define various groups of joints and other elements that can perform the moving actions of a robot arm, for example. Each joint has its own characteristics and can interact with objects within its scene, that is defined by motion planning algorithms.

MoveIt! provides three interfaces for easier access to the actions and services the move_group node has. The *move_group_interface* provides the user with a C++ interface, the *moveit_command* supports python and the Motion Planning RViz plugin gives the user a GUI interface. How the move_group node communicates with the UAV is by using several ROS topics and actions. The UAV's current state is received by the node and information such as the position and orientation of the UAV are given to the `/joint_states` topic. Consequently, in order to broadcast the state of the UAV, it is necessary to launch the node `/joint_state_publisher`, as it is depicted on the right side of the figure 4.2. Also, using the ROS TF library, the move_group node receives constant information about the UAV's pose. The transform between the UAV's base frame and the world or map frame is also provided by this TF.

Finally, the ROS Param Server is used by the move_group node to get all the configuration. This includes the description, the URDF, the SRDF of the UAV and other information relative to its configuration like for example the joint limits and some other variables related to motion planning. The URDF (Unified Robot Description Format) is an XML file that describes all elements of the robot and the SRDF (Semantic Robot Description Format) is intended to represent information about the robot that is not in the URDF file.

The MoveIt! interface also provides the developer access to motion planning libraries, that can be accessed through ROS actions and services. If the user wants, they can initiate a motion plan request by selecting the desired goal in RViz and the motion planner calculates the trajectory it will take to get the UAV to the destination. The move_group node will create the desired trajectory and during the path it will detect any obstacles identified by the sensors and avoid them. This is all done without violating the velocity and acceleration constraints of the UAV model.

4.4.1 MoveIt! Setup Assistant

This section is dedicated to the MoveIt! Setup Assistant. This software is used to configure UAVs within the MoveIt! framework and generates the SRDF as well as other configuration files necessary. In this chapter it is described the process and how the UAV is created to later on be used in RViz.

The first thing to do is to locate on the computer the description of the model the developer wants to use, the type of the file is a .xacro. Once it is loaded into the MoveIt! Assistant the next step is to generate the self collision boxes. The default self collision generator searches for pairs of links in the description files of the UAV that can be safely disabled. This will decrease the planning processing time dramatically. The assistant will then present the results, identifying the links and its possible collisions. As it is shown in picture 4.3 all of these links are adjacent to each other and they can all be disabled.

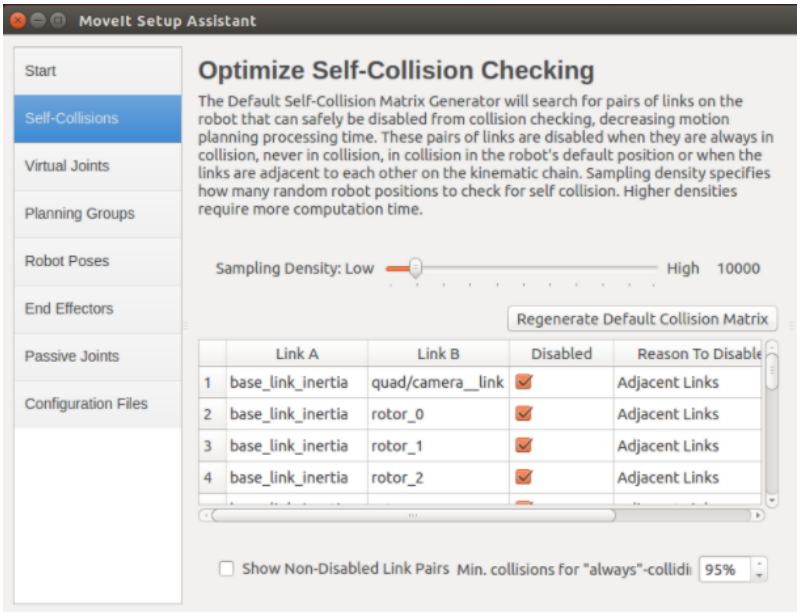

Figure 4.3: MoveIt! Setup Assistant: Self Collisions tab.

Moving on to the Virtual Joints tab, these joints are used in order for the world to know the position of the UAV. It is supposed to be defined only one virtual joint, attaching the UAV's *base_link* to the *world_frame*. The objective of this joint is to represent the base of the robot in a Cartesian plane. Since the UAV is a multi degree of freedom object, the joint type must be floating. The next tab of this setup is the Planning Groups tab. This tab is responsible for describing different parts of the UAV. As in this dissertation, the UAV is used as a single object, there is only one planning group.

Next, there is the Passive Joints tab. However, since the UAV used does not have any special operations happening internally required to create other joints that can move freely from the UAV this process can be skipped. The last tab is responsible for creating the configuration files that enable the developer to create the respective package needed to include the robot in RViz and MoveIt!.

Beyond this, the setup assistant also creates a launch folder in order to launch the robot in the RViz software. Some files that are important referencing are the *quad.srdf*, *ompl_planning.yaml*, *kinematics.yaml* and *joint_limits.yaml*. Respectively, the srdf file is where it is represented the semantic information relative to the UAV.
The *ompl_planning.yaml* is where the planning is described and the *kinematics.yaml* is where the kinematic solver is specified. At last, the *joint_limits.yaml* file gives the developer some additional information about the joints in the respective planning groups that aren't stated in the URDF file. This allows for the controller to established realistic trajectories for the UAV.

As this setup is finished, the *joint_limits.yaml* file must be updated. The information that should be added are the acceleration and the velocity limits for the UAV. Besides making the trajectories more realistic, it also makes the planning processes faster by eliminating a set of other possible trajectories.
As a final result of this setup, the developer should obtain the UAV depicted in figure 4.4 and 4.5.

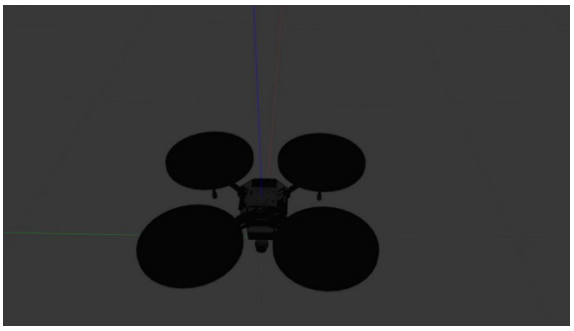

Figure 4.4: Top view of the UAV model DJI Matrice 100.

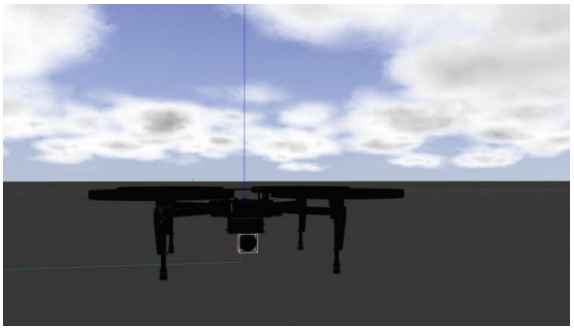

Figure 4.5: Front view of the UAV model DJI Matrice 100.

4.4.2 Physics of the Model

The platform Unreal Engine 4 and the software AirSim already offer a suitable environment for the UAV. The virtual world created within Unreal Engine already offers the developer an internal physics engine with its own physics laws and all the aspects like the weight of the robot, its inertia and all the respective dynamics. As this simulator is just used to put things into perspective, the model used in AirSim may differ a bit from the model in the RViz software, the DJI Matrice 100.
Within Unreal Engine, it is assumed the weight of the UAV is distributed and geometrically centered. The UAV is represented as an object and it has its own collision boxes.

Following, it is explained the mechanics of the propellers of a UAV. It is common for UAVs to have counter rotating propellers, this is because the rotational acceleration of the propellers produces force in one direction. If all propellers were to rotate in the same direction, the torque produced would be applied vertically to the center of the UAV's mass, thus creating an infinite spin and eventually causing the UAV to lose control and crash. It is for this reason that multi rotor UAVs are built with counter rotating propellers. In figure 4.6 is displayed this example.

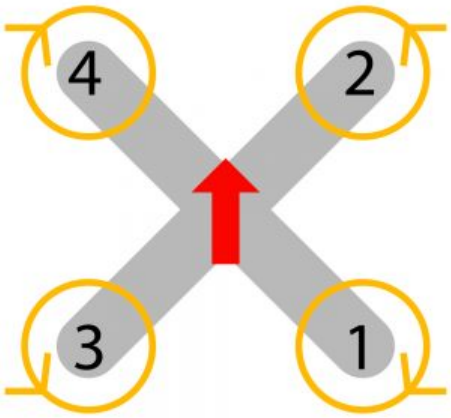

Figure 4.6: Orientation of the propellers on the model used. [64]

The torque applied to the center of mass of the UAV can be represented by the following equation: $\tau = K(F1 + F4 - F2 - F3)$ where τ means the total torque applied, K is a constant that represents the sum of each propeller lifting force and F1, F2, F3 and F4, respectively represent the lifting force produced by each individual motor.
Following Newton's third law of physics, for every applied force, there's an equal and opposite reaction force to it, meaning that an upward torque is produced by motors 1 and 4 producing a clockwise motion. The same law can be applied to motors 2 and 3, producing a counter clockwise motion.

It is based on this principle that UAVs are able to fly and navigate. For example, this is how the yaw is controlled, by applying more or less torque on these sets of propellers. Yaw is the angle produced by the vertical axis Uz', as it is depicted in figure 4.7.

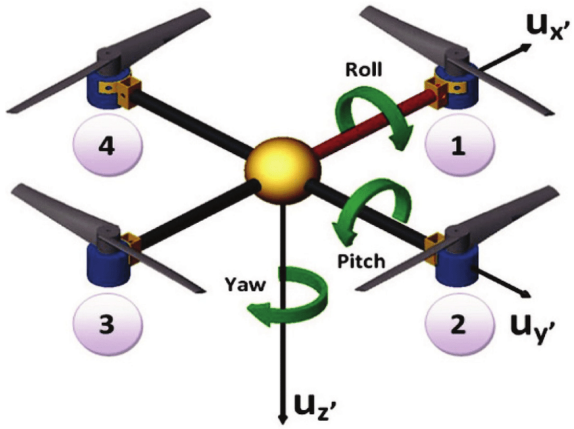

Figure 4.7: Perspective view of a UAV and its axis. [65]

By rotating its yaw, the UAV will rotate on its center axis however, this is not how the UAV navigates in the world. In order to navigate in the environment the user has to apply changes to the pitch and roll angles. These angles are formed by rotating around the horizontal axis of the UAV.
Sending commands to apply more torque or force to one side of the UAV will make changes in the orientation, making the UAV tilt to one side, therefore increasing its speed in that axis.

4.5 Grid Maps

This section is relative to grid maps, what they are and their purpose in this dissertation. How FlamMap works has already been explained in section 3.5, in this chapter only the various outputs of FlamMap and how to include them on the RViz software will be covered.
As it is stated in [66], a grid map is a network of evenly spaced horizontal and vertical lines used to identify locations on a map. Giving an example, someone can pick a location on a map and divide it into a certain number of columns and rows, thus creating a grid. A grid can also display locations projecting coordinates on a map.
There are many examples of grid maps, for example, below in figure 4.8 there is a map divided into grids. There are five columns vertically, from the letter A to the letter E and horizontally, there are five rows, from number 1 to 5.

Figure 4.8: Grid map divided into five columns (A-E) and five rows (1-5). [66]

Another essential point of grid maps are graticules, which are parallel lines showing the latitude and longitude on earth. These graticules can be used on a grid map to show a location in coordinates, with degrees of longitude and latitude.
With this, it is possible to combine both maps displaying grids and graticules and it is possible to add another type of measured grid called UTM or Universal Transverse Mercator to later on display more precise information.

As it is explained in section 3.5, FlamMap can have various outputs, it has multiple themes, these being Elevation, slope, aspect, canopy, among others and after inserting some realistic inputs, the user is presented with some output files.
As it is depicted in figure 4.9 the developer can add some files and characteristics to the simulation in order to make it more real. For example, fuel moisture is the amount of water in a fuel, expressed as a percent of the dry weight of that same fuel, as it is explained in [67]. The fuel moisture content in percentage is determined by dividing the difference between the wet and dry weights by the dry weight and multiplying by 100 in order to have a percentage. Files with information relative to fuel moisture can be added to the simulation.

The developer can also include information relative to the wind. The direction can be chosen, if it is blowing up or downhill and the speed and wind azimuth (direction of the wind measured in degrees) can also be chosen. Characteristics of the canopy can also be selected according to the preferences of the user. The foliar moisture content can be selected and the user can choose the calculation method for the crown fire activity. In this simulation, the method chosen is the Scott and Reinhardt method.

Below it is also possible to choose specific settings for the fuel moisture. The user can select a conditioning period however, in this simulation, fixed fuel moisture settings will be used.

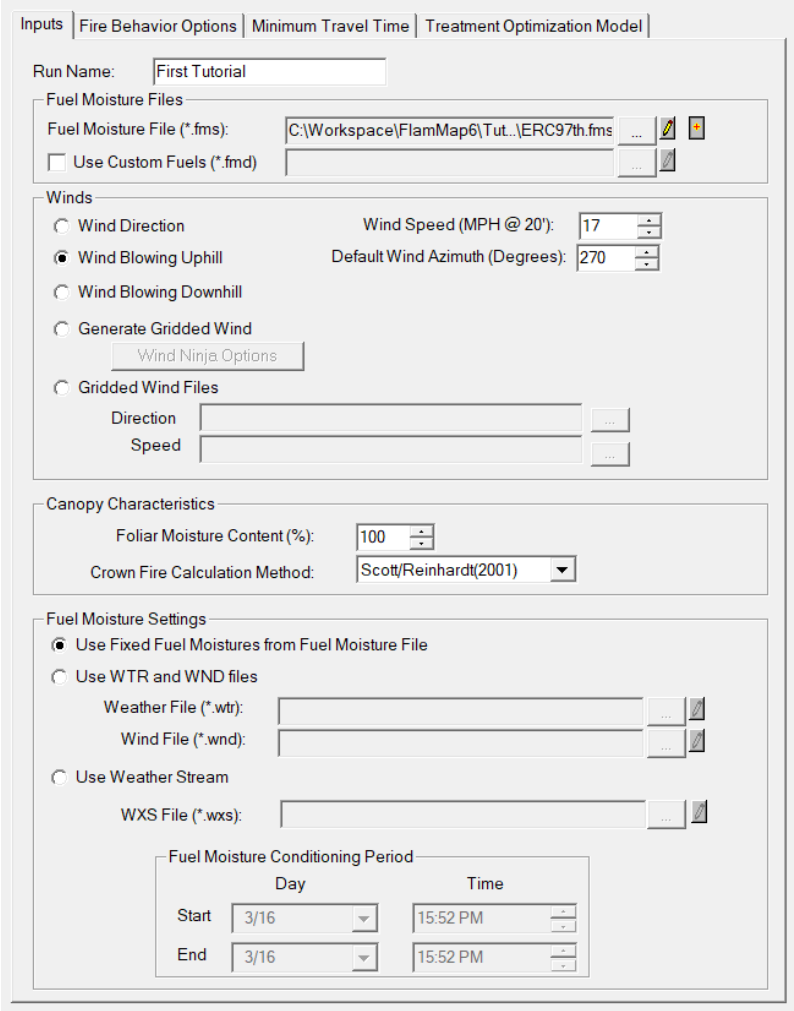

Figure 4.9: Inputs necessary to run a demonstration on FlamMap.

After applying all these changes to the simulation, the developer is presented with a series of outputs. These outputs range from flame length to crown fire activity passing trough the spread rate and fire spread direction.
According to the developer's needs, these outputs can later on be exported into different types of files, to be opened in different softwares. The most important types of files these outputs can be exported to are ASCII grid files and images in the .png format. In this dissertation, the exportation type used is .png, as it is depicted in figure 4.10. Later on, the file will be imported to the RViz software.

Crown fire is a forest fire that spreads from tree top to tree top and according to [68] it can be segmented into four sections, these being:

- **Non burnable**: Portions of landscape that cannot burn, such as, bodies of water or non-vegetated areas;
- **Surface Fire**: A fire that burns surface debris, like dead trees and leaves;
- **Passive Crown Fire**: A fire that spreads on tree tops in which a group of trees becomes fuel for the fire and ignites. These trees can increase the spread rate.
- **Active Crown Fire**: Fire that spreads both in tree tops and in the surface. The burning areas spread at the same rate.

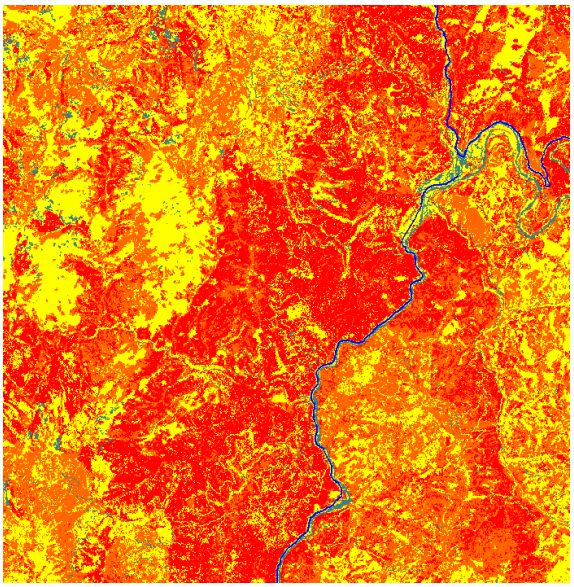

Figure 4.10: Output Crown Fire Activity in .png format. The more red the area is, the more it suffered from crown fire activity.

Once the image file is saved, it is possible to import it into the RViz simulation with some more steps that will be explained in section 5.1.4.

Chapter 5

Autonomous Two Dimensional Navigation

This chapter is dedicated to the two dimensional autonomous navigation of the robot. In the next section a brief explanation of 2D navigation is given and it is also shown its purpose in this dissertation. Some settings and the map plane in the RViz software are also taken into consideration in the next section.

Following, is the section where it is explained how it is possible to pinpoint the robot using the AirSim simulator and the navigation package, within ROS. This is a very important section of this chapter because, without this, it would be impossible for the robot to know its precise localization in the world, whether it is in a virtual or real world. The last section of this chapter is dedicated to the planners used within the 2D Navigation and it is explained how it is possible for the robot to avoid certain areas and how the tracking is done.

5.1 Navigation Stack Setup

This section is about the two dimensional Navigation Stack in ROS. This Navigation Stack is no more than a package that receives data and other information from odometry and sensors. It receives a goal given by the developer or user and it is responsible for sending velocity commands to the robot in order for it to reach its final destination.
Giving a general overview of this stack, on a conceptual level it is fairly simple. The requirements necessary in order to run these packages are that the robot must be able to have ROS running, in order to receive and publish sensor data, using the correct message types within this software. This stack also needs to be able to establish the communication between the robot and other frames in order for the stack to know the location of the robot and for example, where the floor or ground is relative to its position.

For this and other important frames for this stack, a TF Tree or Transform Tree is also necessary. This allows ROS to establish the connection between all possible frames for a better understanding and co-relation between all systems.

In order for this Navigation Stack to work, some hardware requirements are also necessary. A robot with a laser is required in order for the Navigation Stack to know the robot's surroundings and also, the preferable size form of the robot must be a square or circular. This is because the navigation package was developed for robots that do not have arbitrary shapes and sizes. Although it is possible, this stack may not take full advantage of its calculations in some situations, for example when a large rectangular robot passes through a narrow space like a doorway.

In order for this package to work, the Navigation Stack assumes that the robot is already configured so that it can move around.
Below, in figure 5.1, it is depicted a diagram that shows this configuration. In the diagram, it is possible to observe three different colored boxes. Starting with the blue ones, these are components that must be created by the developer for each robot platform to work. The gray boxes are optional requirements that are already implemented in the Navigation Stack, but it is up to the developer to use them in the Navigation Stack. For example, for this dissertation to work, the *map_server* component needs to be created, however the amcl component was not required to do the autonomous navigation on the UAV. Lastly, the white boxes are components that need to be implemented in the Navigation Stack. The developer needs to change some settings and parameters accordingly in order to make the autonomous navigation. All of this, will have its own section, section 5.3.

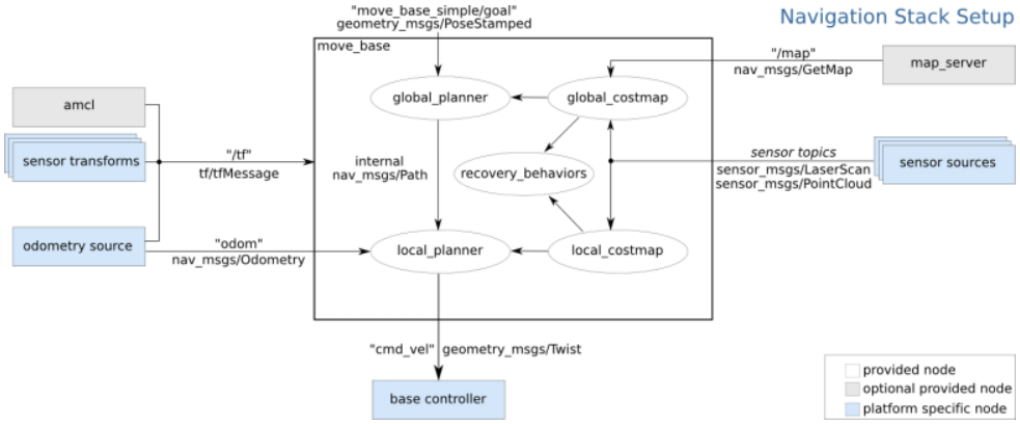

Figure 5.1: Schematics of the Navigation Stack Setup. [69]

Connecting these boxes it is possible to see different message types, for example *cmd_vel* under *geometry_msgs/Twist*, connecting the *base controller* box or *odom* under *nav_msgs/Odometry*, connecting the *odometry source* box.
These message types and their content will be explained in section 5.1.1. For now, it is just explained each box and what is their purpose on the Navigation Stack.

As it is referenced above, it is expected for the developer to install an updated version of ROS on the robot being used.
Starting with the *sensor transform* box, it is required for the Navigation Stack that the robot publishes the necessary information about all the coordination frames and their relationships using Transforms.
Following, is the *odometry source* box. This part of the Navigation Stack is responsible for acquiring all the information related to the robot's odometry using Transforms and sending data through the message type *nav_msgs/Odometry*.
Next is the *base_controller* box. This component is crucial because it is in charge of sending velocity commands to the robot in order for it to move. The data is sent using the topic *cmd_vel* under the message type *geometry_msgs/Twist*. If data is being sent to this topic, it means the robot needs to have a node subscribing to the *cmd_vel* topic. This topic must be capable of receiving linear velocities and angular velocities. The UAV's node available to receive this information is the `/airsim_node/vel_cmd_body_frame` that has the similar type of message. Once this node receives the velocity commands, the data is converted into motor commands and the UAV is able to move in the desired direction with the desired acceleration and speed.

On the right side of the table it is possible to see the *sensor sources* box. In order for the UAV to avoid obstacles, the Navigation Stack receives information from various sensors. The stack assumes the data these sensors receive are being published to the following sensor topics, `sensor_msgs/LaserScan` or `sensor_msgs/PointCloud`.
Although it is not required by the Navigation Stack, the map server is an optional parameter. This component was necessary on this dissertation and will have its own dedicated section, section 5.1.4.

Finally, in the middle, it is possible to see the *move_base* box with two planners, two costmaps and one recovery behaviour. The *move_base* group node provides a ROS interface for the developer to configure and interact with the Navigation Stack on the robot. This component of the Navigation Stack is vital on this dissertation as it is responsible for providing the autonomous navigation of the UAV and has a whole section dedicated to it, section 5.3.

5.1.1 Sensor Information

This sub-section is dedicated to sensor streams and how messages work over the ROS framework. In order for the Navigation Stack to work properly and safely, it is essential to know how to publish data from sensors over ROS correctly. If in any case, the Navigation Stack does not receive information properly from the required sensors, the robot will be driving blindly and most likely, will end up hitting obstacles.
The Navigation Stack can acquire information from a whole range of sensors, as long as the data is sent correctly. Examples of these sensors are lasers, sonars, cameras, infrared sensors, among others.

In this dissertation, there is only one required sensor, a laser sensor. The way this sensor works is by emitting a visible laser light towards a target or object. When hitting the obstacle, the laser light is then reflected diffusely. There is also a receiver lens on the sensor that focuses on that reflected light, which creates a spot or an obstacle on the image. In the next paragraphs it is explained how to send and how the Navigation Stack accepts data published by the *sensor_msgs/LaserScan* and *sensor_msgs/PointCLoud* message types.

To start, both of these message types contain a Transform frame and a time dependent information, in a way it is possible to standardize how this information is sent, all messages within ROS contain a Header type.
This Header type contains three fields, the *seq* field, the *stamp* field and the *frame_id* field.
The *seq* field corresponds to an identifier, that as the messages are sent from the publisher, it automatically increases.
The *stamp* field is responsible for storing time information that is associated with data in the message. Giving an example, in a LaserScan message, this field corresponds to the time at which the scan was taken.
At last, the *frame_id* field stores the Transform frame information associated with the message sent.
As long as the data coming from the sensor can be formatted into this message type, the Navigation Stack can receive information and later on work with it in order to simulate a real robot with these sensors in a virtual simulation.

5.1.2 Odometry Information

This sub-section is relative to the odometry information, what is its message's content and how it can be published to the Navigation Stack. In order for this stack to know and determine the robot's location in the world and link various data to a static map Transforms are used. Although Transforms are essential to give some information to the Navigation Stack, they do not provide any information related to the UAV's velocity. It is because of this, the Navigation Stack requires any odometry source publishes both the Transform with some information and a message from the type *nav_msgs/Odometry* with data relative to the velocity and its content.
Next, it is described what parameters are sent in a message with the odometry type.
This message type is composed of a Header with a respective ID that is supposed to be the parent coordinate frame. It also has a child coordinate frame and another two message types within. These are the pose and the twist messages, that respectively correspond to the position of the robot in the odometric frame and the robot's velocity in the child coordinate frame.

5.1.3 Transform Configuration

As it is mentioned above, the Navigation Stack requires the Transform software library in order to work at its best. A transform tree tries to define, at an abstract level, the translation and rotation between different coordinate frames. For example, in the figure below, figure 5.2 it is possible to see the UAV DJI M100, which has a mobile base with a laser mounted below. There are two coordinate frames, one called *base_link* corresponding to the center of the robot and the other called *base_laser* corresponding to the center point of the laser mount. For a better explanation, let's assume the laser is receiving data from its sensors in the form of distances, in other words, the frame *base_laser* contains data. If the developer wants to take this data and use it to help the robot move and avoid obstacles, a connection is needed from the frame *base_laser* to the frame *base_link*.

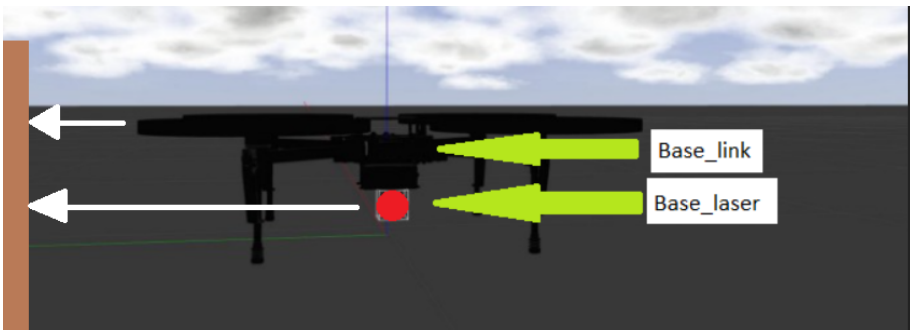

Figure 5.2: The UAV DJI M100 with a laser mounted below. Its frames are also depicted, these being the *base_link* frame in the gravitational center of the robot and the *base_laser* frame in the center of the laser mount.

As it is depicted in the figure 5.2, the laser is five centimeters below the center of the robot. When specifying the relation between these two coordinate frames, the developer needs to consider this distance. In order to get data from the *base_laser* frame to the *base_link* frame the developer must apply the following translation (X:0.0m, Y:0.0m, Z:0.05m). On the other hand, to transfer data in the opposite direction, the following translation (X:0.0m, Y:0.0m, Z:-0.05m) must be applied.
In order for the ROS Navigation Stack to know these translations, this information must be defined and stored in the Transform Tree. Each node on this tree represents a coordinate frame and each node's end represents a coordination frame. In this case, the *base_link* frame is called the parent frame, and the *base_laser* is denominated the child frame.
Applied to this dissertation, the Transform Tree has more than two coordination frames. Below, in figure 5.3 it is possible to see a diagram of the Transform Tree.

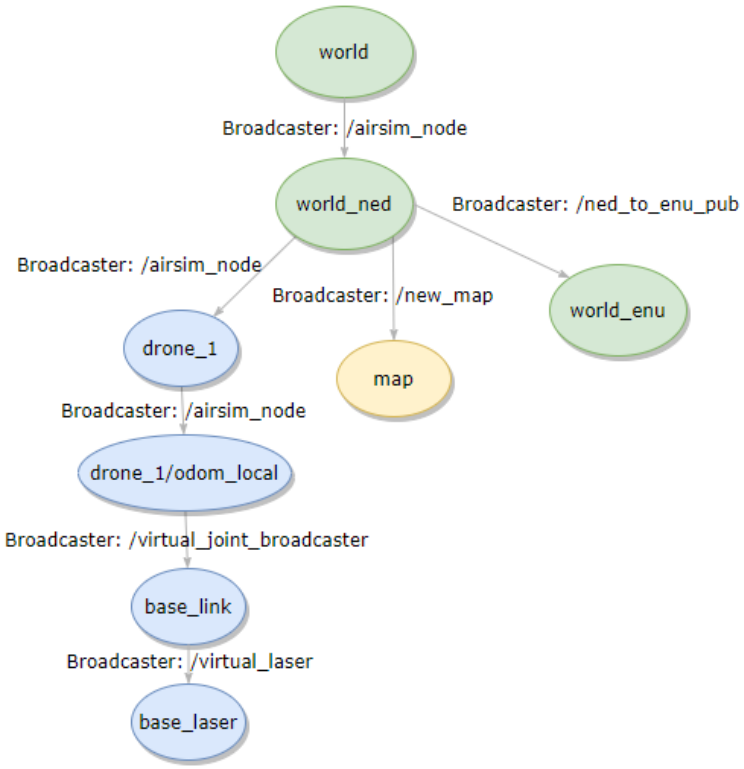

Figure 5.3: Diagram of the Transform Tree used. In blue it is possible to see the UAV's frames, in green the world's frames and in yellow the frame responsible for the map.

As it is depicted in the diagram, the Transform Tree or TF Tree, starts with the *world* frame and from there, it has many other branches. This coordinate frame is the origin and it is designed to allow the interaction between various robots and maps, if the developer wants it. In this case, only one map and one UAV are used, but it is possible to add multiple UAVs and maps if desired. This first frame, is usually static, which means it does not move in space or time and always has the same position, with its broadcaster being the node `/airsim_node`.

The following frame, is the *world_ned* frame, that has three child frames, the *drone_1*, *map* and *world_enu* frame. Although it may seem the same, there is a reason why these two world frames are included (ned and enu). The difference between these two frames is its referential. For short range Cartesian representations, it is mostly used the ENU convention, where the X is East, Y is North and Z is Up. For outdoor systems, it is most desirable to work under the NED convention, where X is North, Y is East and Z is Down. In this case, the following child frames all come down from the *world_ned* frame, which means this is the referential used throughout this dissertation. Following [70], has more information about these two referential and the standard units of measure and coordinate conventions.

Next, on the TF Tree, it is the *map* frame with the broadcaster `/new_map`. Typically this coordinate frame is a world fixed frame, which means it does not move in correlation to the *world* frame. The *map*, usually is a long term global reference and in this dissertation has the same origin as the *drone_1 odom_local* frame.
Following, the next frame, is the UAV's frame, with the name *drone_1*. This, has the GPS coordinates where the UAV starts in the AirSim simulator. In the map depicted in the RViz software, this corresponds to the set of coordinates [0.0, 0.0, 0.0], which means its origin. The child frame of the frame *drone_1* is the frame *drone_1 odom_local*. This coordinate frame is also a world fixed frame. In a typical setup a localization component constantly re-computes the robot pose in which is saved in this frame, these computations are based on an odometry source. This frame is useful as a short term local reference due to its accuracy, however as the UAV goes further away from its initial point, some drifts may occur leading to this frame being poor for long term reference. To avoid this, the UAV always knows its position according to the GPS in the AirSim simulator, which will have its dedicated chapter's section.

The last pair of coordination frames are the *base_link* frame and the *base_laser* frame. The *base_link* frame, usually is attached to the robot base, in this case the base of the UAV and represents the UAV as one single object. If the developer wants to attach a laser, a camera or other components and wants to represent it as a new coordination frame, in order to have its own independent movement, the parent frame will be the *base_link* frame and the child frame will be the laser or the new component frame, in this case is the *base_laser* frame.
In figure 5.4 it is possible to see in the RViz software, the UAV, the map and three coordination frames represented.

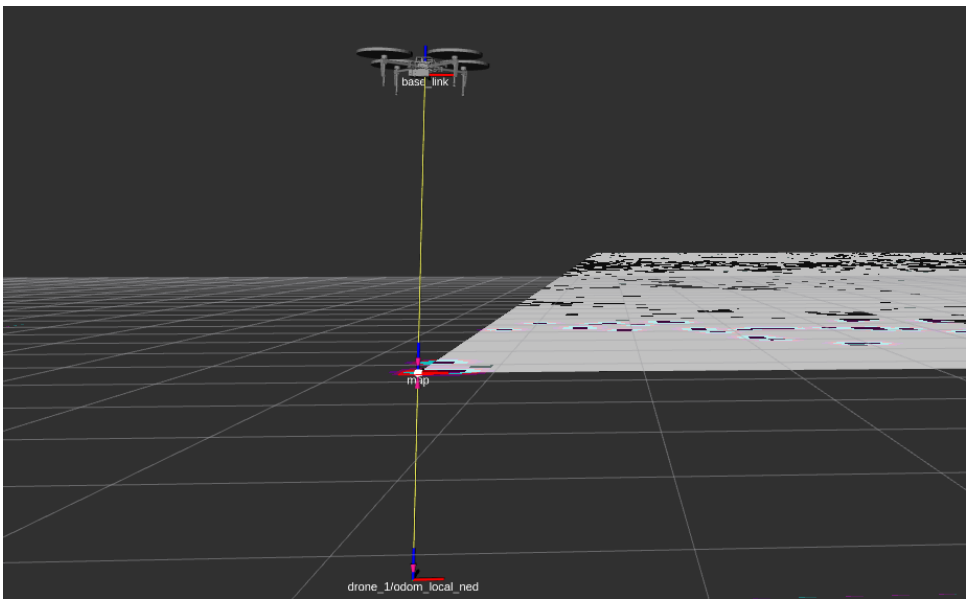

Figure 5.4: UAV with a view of three frames. It is possible to see the *drone_1 odom_local* frame, the *map* frame and the *base_link* frame.

5.1.4 Building the Map Environment

This section is dedicated to explaining how the map is built in RViz and its various parameters. In order for the UAV to navigate with the help of the Navigation Stack, a static map must be present. The addition of a static map aids the Navigation Stack to better locate the UAV and it can receive commands to navigate to a specific location within the map.

After saving the image, the first step is to convert the .png file to a .pgm file, also known as Portable Gray Map file. Next, a .yaml file must be created in order to launch when the simulation starts, but first, some parameters need to be established in this file. These parameters are the image, resolution, origin and occupied and free thresh values.

Starting with the image parameter, this is the path to the .pgm image file the developer wants to open. The resolution field, is the resolution of the map in meters per pixels, in this dissertation the value is set to 0.5. Following, is the origin parameter that represents the two dimensional pose of the lower left pixel in the map, in this case it is [0.0, 0.0, 0.0]. The next parameters are the occupied thresh and the free thresh. Occupied thresh is set to 0.55 and is the field that considers the occupancy of a pixel, values greater than this one consider the pixel as occupied. On the opposite there is the parameter free thresh, set to 0.54 that considers a pixel free if the occupancy probability is less than this value.

With this set, when the RViz starts, a map will appear in the simulation, as it is depicted in the figure 5.5, where it is possible to see the UAV and the map in a gray scale.

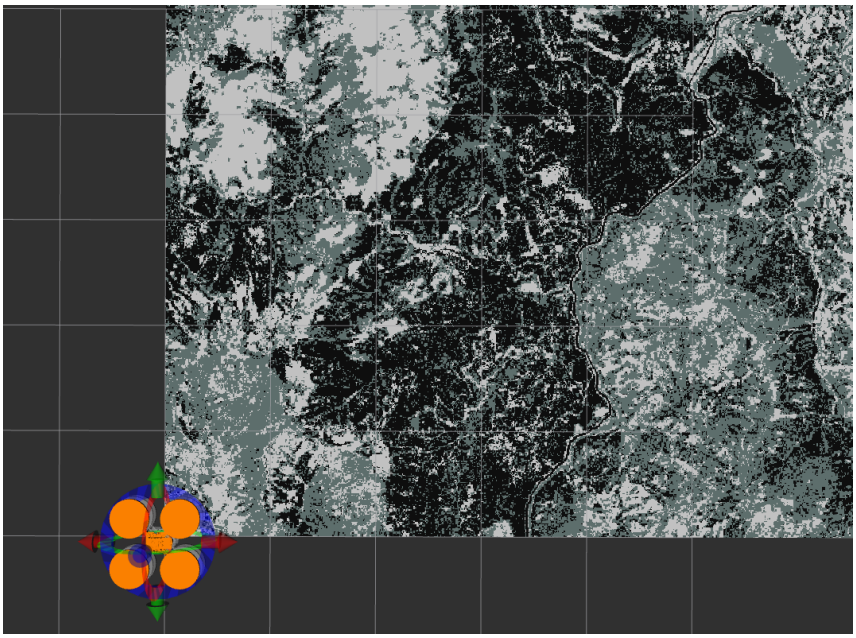

Figure 5.5: Output of FlamMap in the RViz software. It is possible to see the static map and the UAV in an orange color.

5.2 Robot Localization

This section is dedicated to the robot localization. It is addressed how it is implemented and how it works. The robot localization package is a collection of estimation nodes, each one being an implementation of a state estimator for robots moving in two and three dimensional space. In order for the developer to integrate the robot with GPS data, this package also provides the node *navsat_transform_node*.
All of the estimation nodes in this robot localization package share similar features. Some of them are:

- Being able to fuse any number of sensors, meaning the nodes do not restrict the number of input sources. For example, if the robot has multiple IMUs or various sources of odometry information, this package can support all of them;
- Can support multiple message types within ROS. Some examples are `nav_msgs/Odometry` and `sensor_msgs/Imu`;
- Has sensor input customization, which means it is able to exclude data from sensors independently;
- It is possible to do a continuous estimation. As soon as the package receives any input from the sensors, it starts estimating. However, if there are long periods of time where no data is received, the filter will continue to estimate the robot's state.

In order to integrate the GPS data, the *navsat_transform_node*, within the Navigation package, transforms the GPS data from the AirSim simulator into a frame that is consistent with the initial position and orientation in the world frame, inside the RViz software. For this to work, some inputs are required. The *navsat_transform_node* requires three sources of information and these three are the UAV's current pose (position and orientation) in the world frame, an heading with an earth-reference and geographic coordinates expressed in latitude, longitude and altitude. This data can be obtained through the following topics:

- GPS coordinates can be sent in the message type `sensor_msgs/NavSatFix`;
- The heading with the earth reference is in the message type `sensor_msgs/Imu`;
- The message containing the position and orientation of the UAV is in the message type `nav_msgs/Odometry`.

To configure the *navsat_transform_node* some parameters must be changed. First, a launch file must be created in order to launch this node so the communication can start. As this dissertation already needs two simulators and various data in order to work, to keep it better organized, the launch command that launches the localization was put into the main launch file of this project, the quad_2dnav.launch file. With this, when the developer launches the main file with the UAV schematics, the visualization in the RViz and the planning, the GPS navigation is launched as well.
The localization launch file is where some parameters need to be changed. In order for everything to communicate, the topics `/imu/data`, `/gps/fix` and `/odometry/filter`, must be remapped respectively to, `/airsim_node/drone_1/imu/Imu_transformed`, `/airsim_node/drone_1/global_gps_transformed` and `/quad/ground_truth/odometry`.
With this, the navigation package can receive data and information from other topics in other simulators and applications in order to merge them.

5.2.1 AirSim Integration

This chapter's section is relative to how the integration between the AirSim simulator, the navigation package and the representation on the RViz was done. The primary use of the AirSim simulator in this dissertation is to give a precise location using the GPS system within the simulator. With this, python scripts were created in order to send and receive information relative to the GPS data and to be able to perform the communication between both software. The two airsim topics that send the desired data are the `/airsim_node/drone_1/global_gps` and the `/airsim_node/drone_1/imu/Imu` however, both of these topics have some information missing within each message.

As it was previously described, topics can have different message types, in this case, the *global_gps* topic has messages from the type *sensor_msgs/NavSatFix* and the *Imu* topic has messages from the type *sensor_msgs/Imu*.
In order for this dissertation to work, the data from the GPS and IMU sensors has to be published to the Navigation stack to later on, be subscribed by the RViz software.
To accomplish this, two subscribers were created in order to get the data and messages from the GPS and IMU to be published in the Navigation Stack.
However, some information was missing in both messages. In the GPS message being sent, the header with the respective frame ID, the covariance matrix and its respective type were missing. In the IMU message that was being published, information relative to the covariance matrix was also missing. This information was the orientation, the angular velocity and the linear acceleration.

Covariance matrices are used a lot in the robotics world [71]. These, are a way of describing the existent relation between various variables and are a tool for estimating the possible error in a numerical value and predicting this same value.

Respectively to the GPS message, the frame ID added to the header was *drone_1* because the message is relative to the UAV with this name. Regarding the covariance matrix, a matrix with the value 0.01 in the diagonal and 10000 in the other values of the matrix was added and the type is *1*, which indicates it is an approximated covariance and not the exact covariance.
Relative to the IMU message, the same covariance matrix was added to the orientation, angular velocity and linear acceleration matrices. After completing the missing data in both messages, they can then be published to the Navigation Stack.

In order for the UAV in the RViz software and the UAV in the AirSim simulator to be connected, some changes were also performed. As it is pretended for both UAVs to only act as one, their velocity and acceleration also need to be connected. With this, when the UAV from the RViz software travels from point A to point B, the UAV in the AirSim simulator will also travel from point A to point B with the same speed, acceleration and vice-versa. Connecting both UAVs also means the UAV in the RViz software has the same GPS coordinates as the UAV in the AirSim simulator, which is ideal. For this connection to happen the topics containing all data regarding the UAV's speed and acceleration must be subscribed and then published in order for both UAVs to act as one. Whether the user sends a UAV or the other to a specific location, the significant other must go too.

This was done by subscribing the topic `/cmd_vel`, which contains the information relative to the velocity and acceleration of the UAV in the RViz simulation. After receiving data from the topic, it has to be modified in order to publish it to the topic in the AirSim simulator. The data has to be transformed because these two topics have different message types. After putting all the useful data in the same type as the AirSim topic, the data can then be published, for the `/airsim_node/drone_1/vel_cmd_body_frame` to subscribe.

5.3 Autonomous Navigation

This section is relative to the autonomous navigation system of the UAV. In this first section, an explanation of the node responsible for this system is given. Following, there are two sub-sections relative to the costmaps and the planners used in this dissertation.

In the figure 5.1, with the schematics of the Navigation Stack setup, the main white box or the *move_base* box (where are the planners and the costmaps) is what allows the autonomous navigation system. Properly configured, when running the move_base node on a robot the outcome should be the attempt for the robot to reach the desired goal with a certain or specific tolerance given by the user. If there are no dynamic obstacles in the environment, this node will help the UAV get to the desired objective specified by the user.
If it cannot, it will give the information to the user that the robot failed its mission. This node, the move_base node, is also capable of performing recovery behaviours if the robot finds itself stuck.
The recovery process is displayed in figure 5.6.

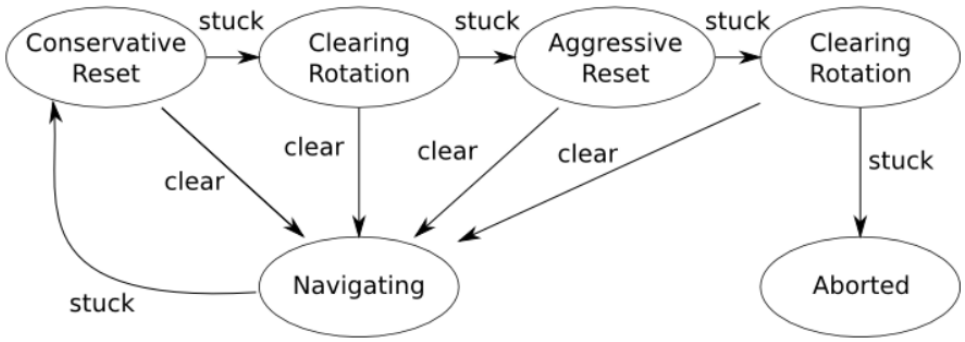

Figure 5.6: Diagram of the move_base node recovery behaviour. [72]

Giving a brief explanation on how this is done, first the user specifies a closer region to the UAV with a certain radius. Obstacles outside this region will be cleared from the robot's map. Next, within this region, the robot will perform a rotation to see what sides are clear. If the robot finds a better route, it will then proceed to take that route and continue the navigation. However, if this step fails, the robot will try again to clear its map, this time more aggressively. After this, another clearing rotation will be followed. If none of these steps work, the robot will consider itself stuck and will notify the user that its mission has been aborted.

This node also provides the user an implementation of the *SimpleActionServer* which consists in an action library that can take goals in the form of the message type *geometry_msgs PoseStamped*. The way this Action Library works is by providing communication between the *ActionClient* and the *ActionServer* via the *ROS Action Protocol*, which is a protocol built on top of ROS messages. Both of these servers provide a simple API for users to request or execute goals.

In order for the server to communicate with the client some messages need to be defined which is done using an action specification. These messages are the *Goal*, *Feedback* and *Result* and both the client and the server communicate through this.
The Goal message is responsible for containing the information relative to the end point or the final position of the UAV. The message type is *PoseStamped* and contains the information of where the robot should move to, in the virtual world.
The Feedback message is a way to tell the ActionClient about the robot's progress when reaching the goal, it sends the current pose of the robot along the path it has to follow.
At last, the Result is the message sent from the ActionServer to the ActionClient when the goal is completed. This message is only sent one time during the execution of the goal as opposed to the Feedback message.
Now, that the move_base node is explained and clarified, the next sub-section is relative to the Cost Maps.

5.3.1 2D Cost Maps

This section provides a more in depth look at the costmaps configuration used in this dissertation. Before explaining all the parameters and their configuration, a costmap is a grid map where each cell is assigned a specific cost. Costs with a greater value mean a smaller distance between the robot and the obstacle. By receiving information and data from the UAV's sensors, a 2D occupancy grid is built around the UAV.

The costmap package within the move_base node provides the user with a configurable structure that is able to keep information relative to where the UAV should or should not navigate through in the static map. This structure is called the occupancy grid. In this dissertation, a static map is provided to the Navigation Stack, with this, it is possible for the costmap package to either mark an obstacle (save information in the costmap) or clear an obstacle (remove information from the costmap). Marking operations are nothing more than an index in an array that can change the cost of each individual cell. On the other hand, clearing operations are operations that consist in ray tracing through a grid for each observation reported.

It is known that each cell in the costmap can have 255 different cost values however, the structure this package uses is only capable of representing three values. Basically, all of the cells in this structure can either be unknown, occupied or free and each one of these status has a special cost value assigned to it. For example, rows or columns with a certain number of occupied cells are assigned with a lethal obstacle cost however, if a group of cells has a certain number of free cells, they are marked with a free space cost.

Relative to the number of times the map updates, this is directly influenced by the parameter *update_frequency*, that is given in Hertz, Hz. Each cycle, data from various sensors comes in and the marking and clearing operations are performed. These operations are then projected into the static map for the user to see in the RViz software.
In figure 5.7 it is possible to see a graphic that helps explaining the inflation process. Inflation is the process of propagating cost values out from the occupied cells that decrease with the distance.

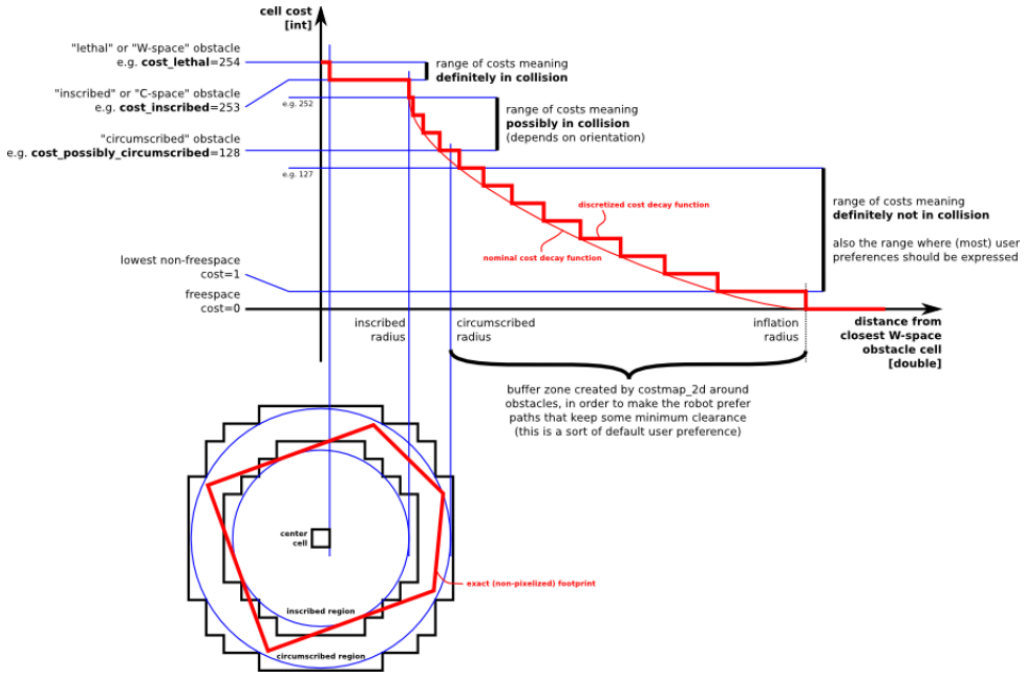

Figure 5.7: Graphic relative to the inflation parameter. On the Y axis is the cell cost and on the X axis is the distance from the obstacle cell. As the distance from the obstacle cell increases, the cell cost decreases. [73]

There are five different types of costmap values:

- **Lethal**: This cost means that there is an actual obstacle in the respective cell, so if the center of the robot would be in this cell, a collision would happen;
- **Inscribed**: This cost means that the cost cell is less than the robot radius. If the robot continues in this direction, it will definitely hit something, causing a collision;
- **Possibly circumscribed**: This cost is a similar cost to the inscribed cost, however it uses the robot's radius as a cutoff distance;
- **Freespace**: This cost is zero, and means that is a free cell and the robot can take this path if desired;
- **Unknown**: Means that the cell has no information relative to its cost.

It should be noted that all the other costs are assigned to values between the circumscribed and free space. In this dissertation, there are three files regarding costmaps and these are:

- *costmap_common_params* file;
- *global_costmap_params* file;
- *local_costmap_params* file.

These are the files where the settings are adjusted and the parameters are added in order to define the costmaps of this work. The *costmap_common_params* file is where information about the world's obstacles is stored. This file also contains information regarding the sensor's topics.
In this file are the parameters *obstacle_range* and *raytrace_range*. These two parameters are thresholds relative to the obstacle information that is put into the costmaps. The first parameter sets the maximum range sensor reading, resulting in an obstacle being put into the costmaps. For example, if this parameter is set to five meters, the information is only updated to the map within a five meter distance of the UAV. The other parameter determines the range to which the algorithm ray traces free space. If this value is set for two meters for example, this means that the UAV will try to clear space in front of it up to two meters away.
Next on the file, are the *footprint* and the *inflation_radius* parameters. As the name suggests, the first parameter is the area of the footprint of the robot. In this case, the robot is a UAV, so it is assumed a squared shape footprint. The *inflation_radius* parameter is the maximum distance from obstacles at which a cost should occur. For example, if this value is set to 0.2 meters, the algorithm will consider paths that stay 0.2 meters or more away from any given obstacle.

Here, are also assigned the parameters *robot_base_frame*, *transform_tolerance*, *update* and *publish* frequency. The first parameter is relative to the robot's frame, in this case is called *drone_1*. The second and third parameters are the frequency at which the algorithm and the RViz software update, respectively. At last, the *transform_tolerance* parameter is the tolerance in seconds, at which the transform is expecting any result or feedback. If it does not get one inside this time window, it is possible the robot will get stuck.
Last, in this file, are the *Layer Definitions*. This is were the sensors are depicted and other frames necessary for the costmaps. The *map* topic is here and it is enabled in order to receive updates from the algorithms. The *Observation_sources* parameter defines the list of sensors that will provide information to the costmap and these sensors are also depicted with the respective frame, data type and the topic on which they provide such information.

Next it is the file *global_costmap_params*. This file is where the configuration options for the global costmap are stored. The parameters that come first in this file are the *global_frame* and *robot_base_frame* that respectively, define the coordinate frame where the costmap will run, in this case, is the *map* frame and the frame that is the reference for the base of the robot, *drone_1*.
Next, are the parameters *update_frequency*, *static_map* and *rolling_window*. The first parameter is already explained above, the second is relative to the use of a static map and is set to true. Finally, the *rolling_window* parameter is set to false as the simulation map is static. If it was not, this parameter should be set to true because it keeps the robot in the center of the area of the costmap as it moves in the world.

The last file related to costmaps is the *local_costmap_params* file. Here, some parameters are the same as in the file above, the ones that are different are the *width*, *height* and *resolution* parameters. As it is intended, these parameters define the width and height of the map in meters and the resolution in meters per cell.

In figure 5.8 it is depicted a map where the burnt areas are represented by the black cells and the cells with a gray color are the areas where the UAV can safely navigate. On the bottom left corner, it is possible to see the UAV with the local costmap being displayed. All the black cells are identified by the costmap by having a blue color around them and if the UAV wants to navigate pass through these cells, it has to avoid them and find a path around them.

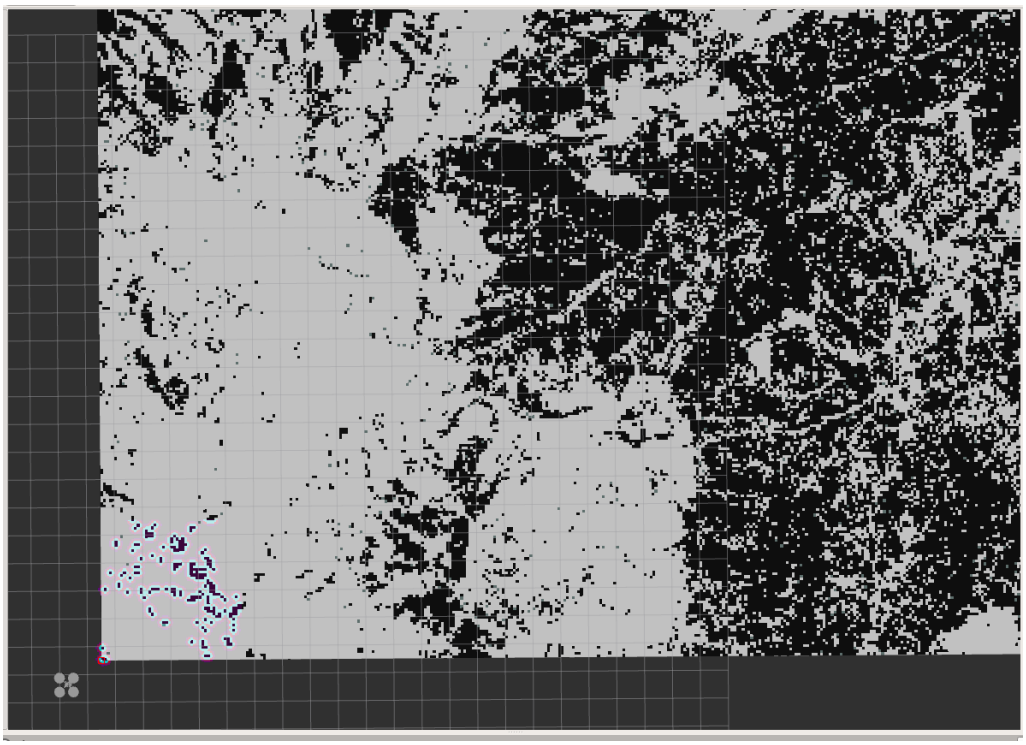

Figure 5.8: Figure where the local costmap is displayed. The black areas are the restricted areas, or areas the UAV must avoid. Around the UAV are the local costmaps that border line the black areas with a blue color.

5.3.2 2D Planners

This section is relative to the planners used, and it is composed of one file, named *base_local_planner_params* that has all the parameters needed for the planning algorithms. The planners used can be divided in two types of planners. The planner used globally, and the planner used locally. The global planner used is the navfn, as it is depicted in the file. This planner uses an algorithm called Dijkstra and the main objective of this planner is to find a global path with the minimum cost between the two given points, these points being where the UAV is and where its goal is.
The parameters in this planner that have been changed from their default values are the *allow_unknown* that specifies whether or not to allow the planner to create plans that transverse the unknown space, which is set to true and the *cost_factor* and *neutral_cost*. These last two parameters, together, determine the quality of the planned path calculated by the algorithm. For example, if the robot goes around objects with a sharper turn or if it tries to keep a greater safe distance from them, etc, this is all accounted in these two parameters.

The other planner, that is used locally is called the Dynamic Window Approach or DWA algorithm. An explanation more in depth is given in [74] however, a summary of the implementation of this algorithm is as follows:

- 1. Discretely sample the robot's control space (dx, dy, $dtheta$);
- 2. For each sampled velocity, this algorithm will perform a forward simulation to predict what would happen if the velocity were to be applied;
- 3. Evaluate by scoring each trajectory resulting from the simulations;
- 4. Pick trajectory with the better scoring and send the associated velocity to the robot;
- 5. Repeat the process.

Besides the steps shown above, this algorithm also maximizes an objective that depends on three things. These are the progress of the target, the respective clearance from obstacles and the forward velocity in order to produce the optimal velocity pair (Vx, Vy). This algorithm, the DWA planner, depends a lot on the local costmap, which provides useful information regarding the nearest obstacles.
Knowing this, it is crucial to tune the parameters for this planner in order to have an optimal behaviour. There are a lot of parameters related to this planner and they are all depicted in the file.

The parameters are grouped into several categories. The first category in the file is relative to the robot. Here are parameters that concern the robot's velocity and acceleration and its turning circle. The next category is called Goal Tolerance and here are set three parameters that relate to the controller's tolerance when achieving a goal.
The category that follows is relative to the speed samples mentioned above. It is here where they are adjusted and the parameter *sim_time* also falls under this category. It is referent to the time the algorithm takes when calculating a new simulation.
Next is the Trajectory Scoring category. This is one of the most important categories, because the parameters that are adjusted here have a direct influence on the objective function. This function relies on three components, these being, the progress to the goal, the clearance from obstacles and the forward velocity. The function's cost is calculated by summing all of the components and the objective is to get the lowest cost.
The last category is called Oscillation Prevention and the parameters in here are responsible for keeping the robot steady if it is oscillating. The oscillations occur for example, if the robot needs to pass through a narrow zone (in between two obstacles) and the local planner is producing paths leading to two opposite directions. If the robot oscillates too much, the algorithm will assume it is stuck and therefore, will start the recovery behaviour.
Depicted below, in figure 5.9 is an example of what is explained above.

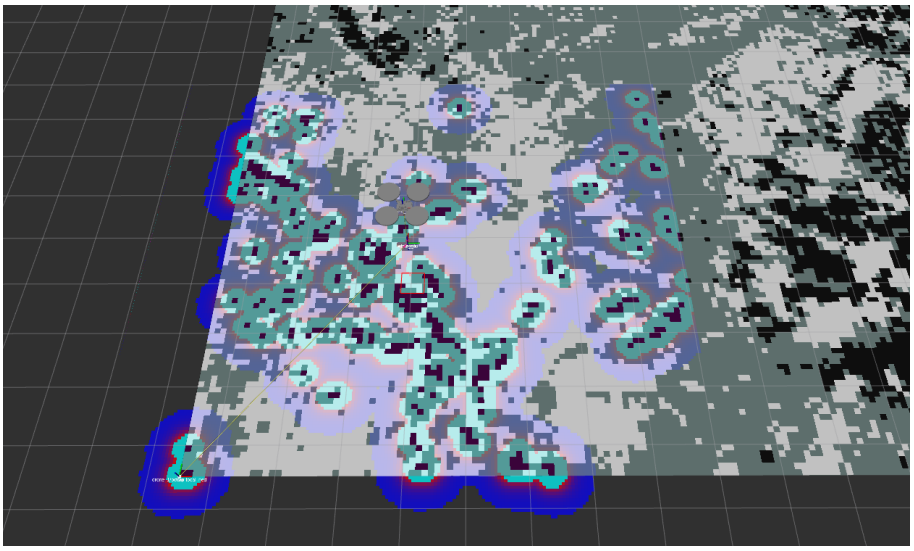

Figure 5.9: Figure where the local costmap is displayed. It is possible to see a portion of the map in a gray scale and the outlines in a blue color represent the borders of the areas where the UAV cannot go. Also in gray, in the middle of the figure, it is possible to see the UAV.

The robot is crossing some burnt areas that are being identified by the local planner. It is possible to see the UAV in a gray color, in the center of the figure and directly below its is footprint.

Chapter 6

Experimental Results

This chapter is relative to the experimental results. Here, are presented the virtual simulations or experiments that are able to demonstrate the behaviour of the autonomous navigation system. This chapter is divided in two sections. The first section is dedicated to explaining how to set up the demonstration and the second section gives the experimental results.

6.1 Simulation Setup

In this section it is explained how to setup everything in order to obtain optimal results from the virtual simulations. To set up everything in order to work, some requirements are necessary. These are a computer with Linux Operative System, the framework ROS with the respective package dependencies installed in order to allow the communication, as well as the RViz software, the Robot Localization Package and the move_base package to allow the autonomous navigation. Unreal Engine 4 with the AirSim simulator is also necessary, along with the description of the UAV DJI M100.

As it is mentioned above, only one file is needed to launch all the packages responsible for the autonomous navigation. However, before launching this file, it is required to launch other nodes or systems to make sure everything is communicating.
First, the user must launch the simulator Unreal Engine 4, where they can choose a wide range of worlds or environments for the UAV. In this case, the environment chosen was the *Blocks* environment.

After UE4 is launched and the simulation starts, in order for the communication between different simulations to be established, it is required to launch the airsim node, *roslaunch airsim_ros_pkgs airsim_node.launch*. This is followed by typing another four commands to connect both UAVs and to be able to transfer data and information between them.
These commands are *rosrun armdrone.py*, *rosrun cov_gps.py*, *rosrun cov_imu.py* and *rosrun airsim_sub.py*. All of these files are scripts that will be running in the background and will handle the data being sent between both UAVs.
With the first command, the user can lift off the UAV in the AirSim Simulator by clicking on a key, inside the terminal window. The second and third commands are responsible for providing the missing information relative to the GPS, the covariance and the IMU of the UAV. With this information completed, both UAVs will be perfectly in-sync.
Finally, the last command that will run the last required script is to enable the velocity commands on both UAVs. With this, it is possible to move the UAV in one software and the other UAV (in the other software) will also acquire the same velocity commands, therefore going to the same location and acquiring the same GPS position.
After all these scripts are launched the user can go ahead and launch the main file, *roslaunch quad_2dnav.launch*, which will start the RViz software, the map, the node with the planning and costmap algorithms, among others.

In this paragraph a more detailed explanation of this launch file is given.
In ROS, launch files are very common to use. These files provide the developer a convenient way to start multiple nodes as well as, initialize other parameters and requirements necessary for the software to work. In this file, the first three nodes that start when this file is launched are the *joint_state_publisher* node, which publishes in the Transform Tree the UAV's links, the laser that is virtually created in order to detect obstacles on the path and the static map is also started with the information regarding the cost cells.
The other nodes in the file are depicted as two groups. The nodes for the Navigation Stack and all its files for the costmaps and planners are a group and the other group has the nodes that publish and subscribe information relative to the Transform Tree in order to keep it connected.

6.2 Simulation Results

In this section the results of various simulations are presented. During these last months of finishing the dissertation, many tests were done in order for the autonomous navigation system to perform at 100%.
However, not all of them are presented below, in the table 6.1. The results that are presented here, are the ones that better demonstrate how the system works. These tests differ from each other. Gradually the difficulty of the runs is increased by choosing paths that are difficult to perform and take longer for the UAV to finish.

As it is possible to see, the runs displayed in the table 6.1 vary from tests where the initial position and the final position of the UAV are 200 meters apart to tests where these positions are almost 4 kilometers apart. The longest test performed is relative to the run nº8, where the UAV covered an almost circular area and traveled nearly 4 kilometers in just 17 minutes and 15 seconds with an average speed of 3.7 meters per second or 13.3 kilometers per hour.

After the table, figures of some runs are included for a better understanding of the system. This allows to better explain how the system works and reference some of its flaws or where the system may have less success.
In the table 6.1 it is possible to see ten runs performed by the autonomous navigation system. Each test or run is composed by the distance the UAV traveled, by the time it took for the UAV to reach its final position and by the average speed at which the UAV performed the run.

Run	Distance (m)	Time (mm:ss)	Average Speed (m/s)	Notes
1	222	00:49	4.5	Goal reached
2	287	01:18	3.6	Goal reached
3	219	00:59	3.7	Goal reached
4	957	04:39	3.4	Goal reached
5	159	00:44	3.6	Rec. mode
6	462	01:34	4.9	Goal reached
7	2121	11.18	3.1	Goal reached
8	3900	17:15	3.7	Goal reached
9	330	01:41	3.2	Goal reached
10	1200	05:39	3.5	Goal reached

Table 6.1: Table where are depicted different tests performed to the system. For each test performed there is the distance covered in meters, the time it took for the UAV to complete it and its average speed in meters per second. The last column refers to any additional notes.

In figures 6.1 to 6.5 it is possible to see represented by a green line, the path assigned for the UAV, this line represents the global path chosen by the user.
The initial position is where the UAV is at the moment and at the end of the line it is the goal or the final position of the UAV.
The line in blue represents the local path of the UAV. The algorithm is always taking into consideration its distance to the green line, its speed and orientation as well as the distance to potential obstacles it may find in the area. Knowing this, the local path is calculated by the DWA algorithm, while the global path is calculated by navfn algorithm. Represented by the pink color are the obstacles or the burnt areas the UAV cannot go through and has to navigate around them. The global path is always calculated in order to avoid these areas and travel along their border.

Around the UAV, in a square shaped form, there is a point cloud. This represents the proximity the UAV is to the desired goal. In the middle and the area around the green line are warmer colors, that lead to the goal and in the extremes of the square are colder colors representing areas that are further away from the desired objective.
In figure 6.1 it is possible to see the path assigned for the UAV. In this test only a straight line was set for the UAV to follow. It has a distance of approximately 200 meters and it took the UAV 49 seconds to reach the desired position. In table 6.1 it is possible to see the average speed of the UAV and in this run is one of the highest, being 4.5 meters per second. The reason why this run is one with the highest average speeds is because the path chosen is in a straight line and around are few areas where the UAV cannot go, therefore the algorithm calculated the best route and since it was safe, it applied almost the maximum speed for the UAV to achieve the goal in the fastest time.

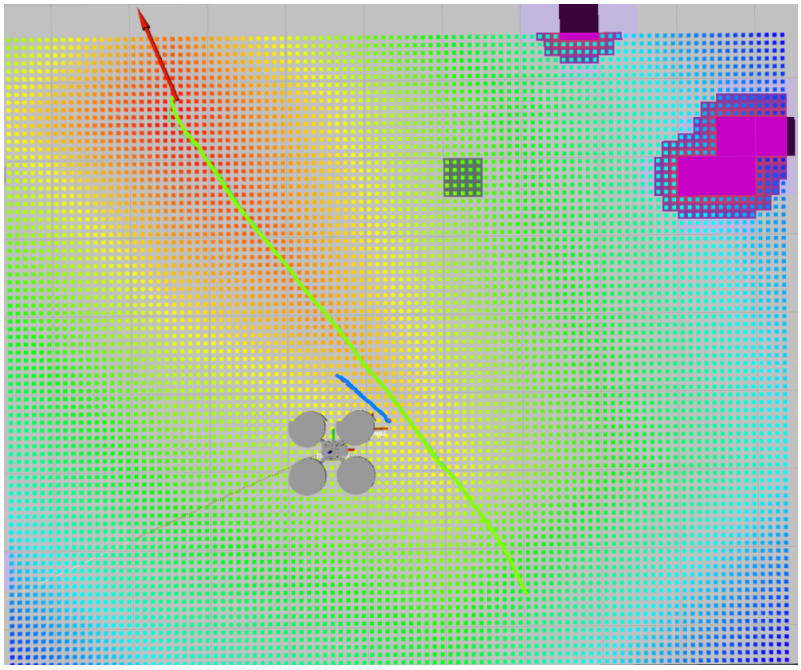

Figure 6.1: Figure where the run nº1 is displayed. It is possible to see the global path represented by the green line and the local path represented by the blue line. The red arrow is the desired final position for the UAV. In a black color it is possible to see the restricted areas, areas which the UAV must avoid and in a pink color are the areas in sensor range.

Another test performed that is a good example to explain how the algorithm works is the run nº3. It is possible to see in figure 6.2, that this run is a little bit more complex, as the algorithm has more to take into consideration.
First, when the user sets the goal, the algorithm in charge for the global path has to calculate a safe route for the UAV to reach the desired destination avoiding the burnt areas, or the areas in the pink color.
After this, the DWA algorithm in charge for the local path needs to take into consideration the speed of the UAV, the distance it is to the areas as well as if it is on track or not (on top of the green line). With this said, in this run the UAV traveled a distance of 219 meters in 59 seconds, with an average speed of 3.7 meters per second or 13.3 kilometers per hour.

Comparing to the run nº1 it is possible to see this path is far more complex. Although the distance is almost the same, the results show that it took the UAV 10 more seconds to reach the goal. The average speed on which the UAV did this path is also lower as this path has more curves and obstacles for the algorithm to compute.
Once more, represented by the point cloud, it is possible to see that the warmer colors like red indicate the path the UAV should follow to reach its desired position.
Nevertheless, the UAV performed this run successfully and without any problem.

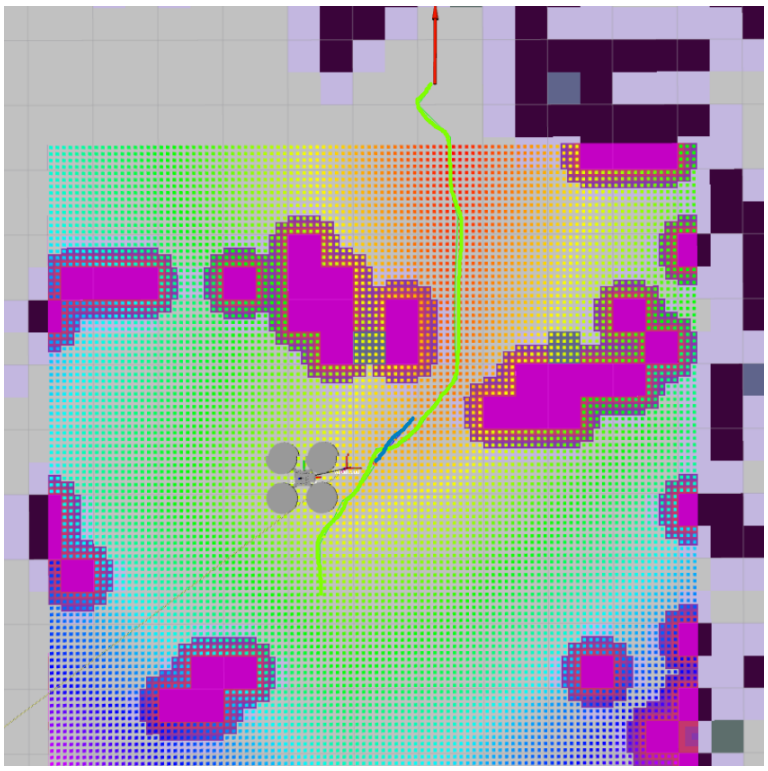

Figure 6.2: Figure where the run nº3 is displayed. It is possible to see the global path represented by the green line and the local path represented by the blue line. The red arrow is the desired final position for the UAV. In a black color it is possible to see the restricted areas, areas which the UAV must avoid and in a pink color are the areas in sensor range.

In the figure that follows, figure 6.3 it is possible to see another run performed by the UAV with the help of the autonomous navigation system.
As the run represented in figure 6.2 this one has approximately the same difficulty level. It has a lot of areas for the algorithm to process and the path created needs to be precise in order for the UAV to avoid these areas and navigate around them or in its limits. This test has a distance of 159 meters and took the UAV 44 seconds to reach its final position. As it is possible to see, in the moment the image was taken the local path of the UAV is about to collide with a restricted area. Right after the blue line (local path) crossed the pink hit boxes a recovery behaviour was initiated.
The algorithm processed the space it had around and the UAV started going backward to avoid the restricted area. This only happens when the path is too narrow and the velocity samples are not enough for the algorithm to calculate more path possibilities. Knowing this, the UAV ends up taking the path chosen and it only knows it isn't a feasible path when it has already hit the restricted area and the recovery behaviour has to act. After this it went forward again, but this time was closer to the global path (the green line) and ended up reaching the desired goal without any more problems.

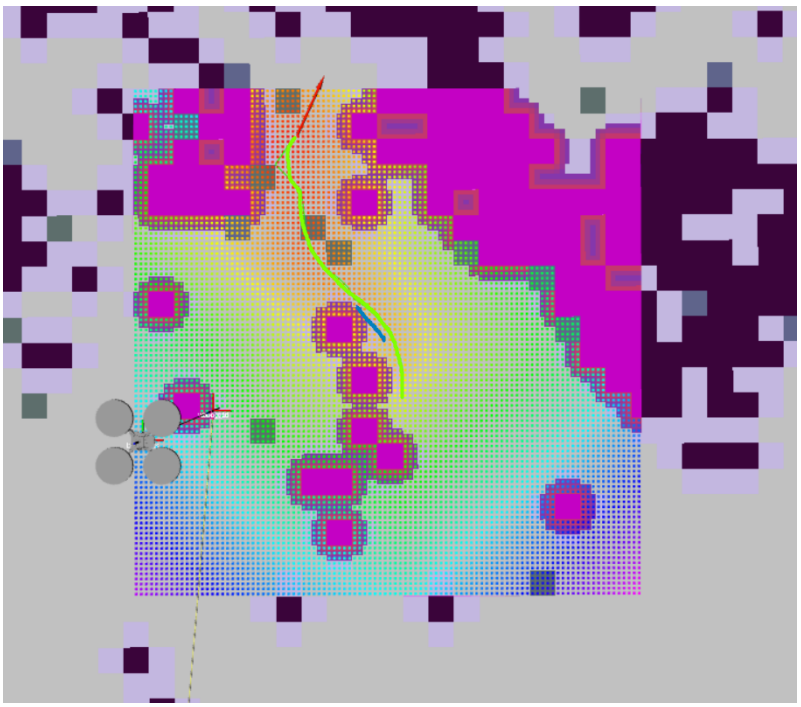

Figure 6.3: Figure where the run nº5 is displayed. It is possible to see the global path represented by the green line and the local path represented by the blue line. The red arrow is the desired final position for the UAV. In a black color it is possible to see the restricted areas, areas which the UAV must avoid and in a pink color are the areas in sensor range.

The next two tests were runs that were performed in order for the UAV to cover a bigger area and therefore test the integrity of the autonomous navigation system. In the next figure, figure 6.4, it is possible to see the path the UAV follows in order to reach the desired goal. Some way points were set in order for the UAV to follow and go through the designated path. As it is displayed in the table, in the run nº7, the UAV did more than 2 kilometers and took 11 minutes and 18 seconds to reach its final destination. The UAV performed this test successfully and not once had to enter the recovery mode.
In figure 6.4 it is possible to see the UAV starts its path by going north and circles around small areas and goes to the left. In the end of this test, the UAV performs two tight turns around areas that are on fire and travels within both limits. In none of these turns did the UAV get stuck and had to enter the recovery mode.

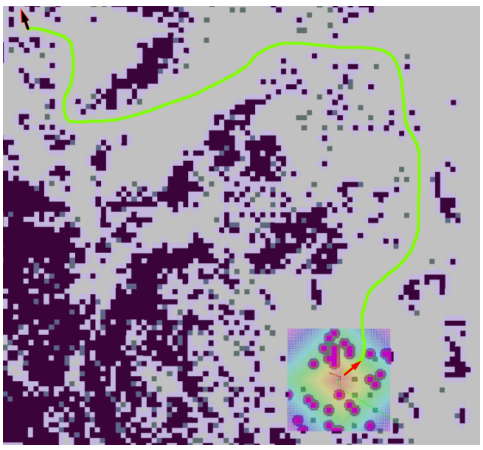

Figure 6.4: Figure where the run nº7 is displayed. It is possible to see the global path represented by the green line and. The red arrow is the initial position and the black arrow, the final position. In a black color it is possible to see the restricted areas, areas which the UAV must avoid and in a pink color are the areas in sensor range.

In the last test displayed, in figure 6.5 the UAV performed a path that is similar to a circular path around an area. The UAV started in the red arrow and ended in the black arrow. This path has almost 4 kilometers and it took the UAV 17 minutes and 15 seconds with an average speed of 3.7 meters per second to perform it. A series of way points were set in the map and the UAV traveled through these paths successfully. Some paths that should be highlighted are the path around the burnt area on top of the figure and the path in the area below where the drone started. In both of these areas the path is very narrow and the drone had to travel within the limits of these areas without going on top of them. Once again, there was no need for the recovery behaviour to enter in action and the UAV performed this run flawlessly.

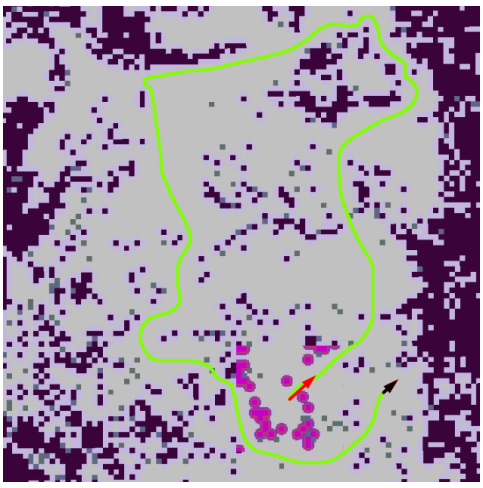

Figure 6.5: Figure where the run nº8 is displayed. It is possible to see the global path represented by the green line. The red arrow is the initial position and the black arrow, the final position. In a black color it is possible to see the restricted areas, areas which the UAV must avoid and in a pink color are the areas in sensor range.

Chapter 7

Conclusion and Future Work

This chapter is divided into two sections. One is related to the conclusion of this dissertation, the results presented are taken into consideration and are discussed.
The other section is relative to future work. Possible future developments and the continuity of this work are discussed and what would be the next ideal steps in order to have a real and functional autonomous navigation system.

7.1 Conclusion

In order for a better understanding of this dissertation, the conclusion's section is divided into two sub-sections, these being the Simulators section and the Autonomous Navigation section. In the first sub-section within 7.1 a conclusion related to the simulators used is given and in the second sub-section is provided a conclusion related to the autonomous navigation system and its algorithms.

In the beginning of this dissertation, a work plan was created to better split the different tasks and establish a certain time to do them. The tasks that were set in the beginning were all performed except one. Unfortunately, it wasn't possible to do the field tests or the real life simulations on the autonomous navigation system. Real life tests were scheduled with the Fire Department to start controlled fires in order to test the algorithms of this dissertation, however, due to the pandemic, these tests were postponed and it was not possible to test this system in real life.

7.1.1 Simulators

Since the beginning of this dissertation, a lot of papers were read. When the part for testing simulators began, a lot of uncertainty was presented. Although it wasn't new, the best choice would be to use the ROS framework and the RViz software, the graphical simulator was still missing. Once more, a lot of research was done and the AirSim simulator seemed to be and it still is the best choice in my perspective. It is very stable when running in machines with low processing power units and if the user wants, they can increase the graphics for a better image to improve the quality and for the user to be in a more immersive environment. Another point that led to choose this simulator was the opportunity to have more worlds or environments available. Besides this, the simulator already has some integration done with the ROS environment and this meant the work could be focused on developing the autonomous navigation system.

What followed was trying to set up a simulation where a fire would be included or the user could set an input from a forest fire and then test the UAV in a controlled environment or a virtual simulation to see and adjust its behaviour accordingly.
For this, the FlamMap was used. Here, the developer could create a map and being dependent from the output chosen, could then test the behaviour of the autonomous navigation system on the UAV in a controlled environment.
After having both simulators working and the map with the desired output included in the simulation, real conditions were added to better simulate what it would be in real life. Gravity, momentum and collisions were added to the AirSim simulator to better demonstrate the conditions the UAV would be in.

7.1.2 Autonomous Navigation

After both simulators were working and connected with the correspondent topics, the next step in this dissertation would be to include an autonomous navigation system for the UAV. A lot of research related to the planners and its algorithms was done. However, quickly became evident the best planners to use would be the NavfnROS planner for the global path and the DWA planner for the local path planning. Knowing this, a flexible system for the UAV to navigate through a difficult environment was presented.
As it is explained in chapter 5 this navigation system was done using the operating system ROS and mainly the package move_base as a way to allow and modify the path planning of the UAV.

After a lot of testing and changing parameters for the UAV's autonomous system to be at 100%, the main objective of this dissertation, to create an autonomous navigation system for UAVs to prevent forest fires was accomplished.

A lot of runs were performed to test the integrity of this system and although in chapter 6 only ten were presented, the results give a good example of the overall performance of the system.
In the eight runs performed (run nº1 to nº6, nº9 and nº10) and the two tests that were done with a lot of way points (run nº7 and nº8) in only one run (run nº5) did the system needed to start the recovery mode for the UAV to adjust its position accordingly and create a new path to the desired goal.
However, although the system entered in this mode, these results only show that the autonomous navigation system is prepared for these kinds of adversities and it is able to recover from them with success.

As it is said above, there was only one task missing, that unfortunately was not possible to do, due to the pandemic. This was to do field tests and test these planners in a real world environment with controlled fires set by the fire department.

7.2 Future Work

This section is dedicated to possible future work of this dissertation.
Initially, the main goal of this dissertation was to develop a three dimensional autonomous navigation system for the UAV DJI Matrice 100, however after some simulations done in the virtual world, doubts started to appear because the computational power unit on both the computer used and on the UAV could not handle such system.
In order to avoid this, it was changed to a two dimensional system in order to be lighter for both processor units and the user, would be the one to set an altitude for the UAV. Having said this, an area where this dissertation can improve is by trying new simulations with the three dimensional system in order for the UAV to move in a three axis world.

Related to the simulators, some things can also be improved. Despite being successful in the environment used and the static map where the algorithm was running, one can improve by adding more worlds and more simulated environments to test the algorithm of the autonomous navigational system. On the AirSim simulator, the graphics can also be improved in order to give a better experience for the user.
Concerning the navigational system, some improvements can also be done, as this system was only perfected in virtual simulations. From the best path chosen for the goal to the recovery behaviours imposed if the sensors detected an object some things can be improved when the UAV goes to testing in the real world.

The way this system was designed was to be an iterative process that keeps on improving with experience and the more environments the algorithm has, the better. The system is already implemented, functioning and working however, if necessary it is just some parameters that need to be adjusted in order for the system to better embrace new worlds and environments.

The algorithm for this type of applications was the most suitable one however, there are other algorithms that are able to enrich the overall system performance and the user experience and this is a point that can also be improved in the future.

Bibliography

[1] F. Afghah, A. Razi, J. Chakareski, and J. Ashdown. "Wildfire Monitoring in Remote Areas using Autonomous Unmanned Aerial Vehicles." In: (2019). arXiv: 1905.00492. URL: http://arxiv.org/abs/1905.00492.

[2] *Mars Helicopter - NASA Mars*. URL: https://mars.nasa.gov/technology/helicopter/ (visited on 11/26/2020).

[3] *Suspected drone strikes kill 12 civilians in Yemen — The Bureau of Investigative Journalism*. URL: https://www.thebureauinvestigates.com/stories/2012-05-15/suspected-drone-strikes-kill-12-civilians-in-yemen (visited on 11/26/2020).

[4] *Neuigkeiten | Meteomatics*. URL: https://wp.meteomatics.com/neuigkeiten/page/2/ (visited on 11/26/2020).

[5] S. Karma, E. Zorba, G. C. Pallis, G. Statheropoulos, I. Balta, K. Mikedi, J. Vamvakari, A. Pappa, M. Chalaris, G. Xanthopoulos, and M. Statheropoulos. "Use of unmanned vehicles in search and rescue operations in forest fires: Advantages and limitations observed in a field trial." In: *International Journal of Disaster Risk Reduction* 13 (2015), pp. 307–312. ISSN: 22124209. DOI: 10.1016/j.ijdrr.2015.07.009.

[6] C. Yuan, Y. Zhang, and Z. Liu. "A survey on technologies for automatic forest fire monitoring, detection, and fighting using unmanned aerial vehicles and remote sensing techniques." In: *Canadian Journal of Forest Research* 45.7 (2015), pp. 783–792. ISSN: 12086037. DOI: 10.1139/cjfr-2014-0347.

[7] P. Kardasz and J. Doskocz. "Drones and Possibilities of Their Using." In: *Journal of Civil & Environmental Engineering* 6.3 (2016). ISSN: 2165-784X. DOI: 10.4172/2165-784x.1000233.

[8] L. M. Belmonte, R. Morales, and A. Fernández-Caballero. "Computer vision in autonomous unmanned aerial vehicles-A systematic mapping study." In: *Applied Sciences (Switzerland)* 9.15 (2019), pp. 1–34. ISSN: 20763417. DOI: 10.3390/app9153196.

[9] M. Dong and Z. Sun. "A behavior-based architecture for unmanned aerial vehicles." In: *IEEE International Symposium on Intelligent Control - Proceedings* (2004), pp. 149–155. DOI: 10.1109/isic.2004.1387674.

[10] L. Yang, J. Qi, J. Xiao, and X. Yong. "A literature review of UAV 3D path planning." In: *Proceedings of the World Congress on Intelligent Control and Automation (WCICA)*. Vol. 2015-March. March. Institute of Electrical and Electronics Engineers Inc., 2015, pp. 2376–2381. ISBN: 9781479958252. DOI: 10.1109/WCICA.2014.7053093.

[11] F. Schøler. *3D Path Planning for Autonomous Aerial Vehicles in Constrained Spaces*. 2012, p. 144. ISBN: 9788792328731. URL: http://forskningsbasen.deff.dk/Share.external?sp=Sc6793ef3-be6c-43a3-bed4-d561bc1f8cd0{\&}sp=Saau.

[12] S. R. Hansen, T. W. McLain, and M. A. Goodrich. "Probabilistic searching using a small unmanned aerial vehicle." In: *Collection of Technical Papers - 2007 AIAA InfoTech at Aerospace Conference* 1 (2007), pp. 287–302. DOI: 10.2514/6.2007-2740.

[13] S. Song and S. Jo. "Surface-Based Exploration for Autonomous 3D Modeling." In: *Proceedings - IEEE International Conference on Robotics and Automation* May 2018 (2018), pp. 4319–4326. ISSN: 10504729. DOI: 10.1109/ICRA.2018.8460862.

[14] M. D. S. G. Tsuzuki, T. D. C. Martins, and F. K. Takase. "Robot path planning using simulated annealing." In: *IFAC Proceedings Volumes (IFAC-PapersOnline)* 12.PART 1 (2006), pp. 175–180. ISSN: 14746670. DOI: 10.3182/20060517-3-fr-2903.00105.

[15] S. Ghambari, J. Lepagnot, L. Jourdan, and L. Idoumghar. "A comparative study of meta-heuristic algorithms for solving UAV path planning." In: *Proceedings of the 2018 IEEE Symposium Series on Computational Intelligence, SSCI 2018* November (2019), pp. 174–181. DOI: 10.1109/SSCI.2018.8628807.

[16] Z. F. He and L. Zhao. "The comparison of four UAV path planning algorithms based on geometry search algorithm." In: *Proceedings - 9th International Conference on Intelligent Human-Machine Systems and Cybernetics, IHMSC 2017*. Vol. 2. Institute of Electrical and Electronics Engineers Inc., 2017, pp. 33–36. ISBN: 9781538630228. DOI: 10.1109/IHMSC.2017.123.

[17] Y. Cho, D. Kim, and D.-S. K. Kim. "Topology representation for the Voronoi diagram of 3D spheres." In: *International Journal of CAD/CAM* 5.1 (2009). URL: http://www.ijcc.org/ojs/index.php/ijcc/article/view/38{\%}0Ahttp://www.ijcc.org/ojs/index.php/ijcc/article/viewFile/38/34.

[18] S. Karaman and E. Frazzoli. "Incremental sampling-based algorithms for optimal motion planning." In: *Robotics: Science and Systems* 6 (2011), pp. 267–274. ISSN: 2330765X. DOI: 10.7551/mitpress/9123.003.0038. arXiv: 1105.1186.

[19] S. M. LaValle. "Planning algorithms." In: *Planning Algorithms* 9780521862059 (2006), pp. 1–826. DOI: 10.1017/CBO9780511546877.

[20] K. Yang and S. Sukkarieh. "3D smooth path planning for a UAV in cluttered natural environments." In: *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS* (2008), pp. 794–800. DOI: 10.1109/IROS.2008.4650637.

[21] R. Dutoit, D. Moines, M. Lyle, and M. Holt. "UAV Collision Avoidance Using RRT * and LOS Maximization Technical Report # CSSE12 - 03." In: ().

[22] I. A. Musliman, A. A. Rahman, and V. Coors. "Implementing 3D Network Analysis in 3D-GIS." In: *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* 37 (2008), pp. 913–918. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.150.8084{\&}rep=rep1{\&}type=pdf.

[23] L. De Filippis, G. Guglieri, and F. Quagliotti. "Path planning strategies for UAVS in 3D environments." In: *Journal of Intelligent and Robotic Systems: Theory and Applications* 65.1-4 (2012), pp. 247–264. ISSN: 09210296. DOI: 10.1007/s10846-011-9568-2.

[24] A. Nash, S. Koenig, and C. Tovey. "Lazy Theta*: Any-angle path planning and path length analysis in 3D." In: *Proceedings of the 3rd Annual Symposium on Combinatorial Search, SoCS 2010* (2010), pp. 153–154.

[25] S. Koenig and M. Likhachev. "Fast replanning for navigation in unknown terrain." In: *IEEE Transactions on Robotics* 21.3 (2005), pp. 354–363. ISSN: 15523098. DOI: 10.1109/TRO.2004.838026.

[26] C. Schumacher and P. Chandler. "UAV Task Assignment with Timing Constraints via Mixed-Integer Linear Programming American Institute of Aeronautics and Astronautics." In: September (2004), p. 6410.

[27] E Masehian and G Habibi. "Robot Path Planning in 3D Space Using Binary Integer Programming." In: *International Journal of Mechanical Systems Science and Engineering* 1.5 (2007), pp. 1255–1260.

[28] A. Chamseddine, Y. Zhang, C. A. Rabbath, C. Join, and D. Theilliol. "Flatness-based trajectory planning/replanning for a quadrotor unmanned aerial vehicle." In: *IEEE Transactions on Aerospace and Electronic Systems* 48.4 (2012), pp. 2832–2847. ISSN: 00189251. DOI: 10.1109/TAES.2012.6324664.

[29] F. Borrelli, D. Subramanian, A. U. Raghunathan, and L. T. Biegler. "MILP and NLP techniques for centralized trajectory planning of multiple unmanned air vehicles." In: *Proceedings of the American Control Conference* 2006 (2006), pp. 5763–5768. ISSN: 07431619. DOI: 10.1109/acc.2006.1657644.

[30] O Al, H Jeong, B. H. Koo, and C. G. Lee. "Mössbauer spectra of MnFe." In: 4 (2010), pp. 1129–1132. DOI: 10.1007/s11771.

[31] J. L. Foo, J. Knutzon, V. Kalivarapu, J. Oliver, and E. Winer. "Path planning of unmanned aerial vehicles using B-splines and particle swarm optimization." In: *Journal of Aerospace Computing, Information and Communication* 6.4 (2009), pp. 271–290. ISSN: 15429423. DOI: 10.2514/1.36917.

[32] J. Botzheim, Y. Toda, and N. Kubota. "Bacterial memetic algorithm for offline path planning of mobile robots." In: *Memetic Computing* 4.1 (2012), pp. 73–86. ISSN: 18659284. DOI: 10.1007/s12293-012-0076-0.

[33] C. T. Cheng, K. Fallahi, H. Leung, and C. K. Tse. "Cooperative path planner for UAVs using ACO algorithm with gaussian distribution functions." In: *Proceedings - IEEE International Symposium on Circuits and Systems* (2009), pp. 173–176. ISSN: 02714310. DOI: 10.1109/ISCAS.2009.5117713.

[34] F. Yan, Y. S. Liu, and J. Z. Xiao. "Path planning in complex 3D environments using a probabilistic roadmap method." In: *International Journal of Automation and Computing* 10.6 (2013), pp. 525–533. ISSN: 14768186. DOI: 10.1007/s11633-013-0750-9.

[35] J. António Barata de Oliveira, F. Antero Cardoso Marques, J. Presidente, D. Paulo da Costa Luís da Fonseca Pinto Arguentes, and D. João Almeida das Rosas. *Luìs Ricardo Baptista do Ó Aerial Navigation for active perception in Precision Agriculture*. Tech. rep. 2018. URL: https://github.com/joaomlourenco/novathesis.

[36] E. Galceran and M. Carreras. "A survey on coverage path planning for robotics." In: *Robotics and Autonomous Systems* 61.12 (2013), pp. 1258–1276. ISSN: 09218890. DOI: 10.1016/j.robot.2013.09.004.

[37] J Milnor. "Morse Theory: Annals of mathematics studies." In: (1963).

[38] M. Jun and R. D. Andrea. *PATH PLANNING FOR UNMANNED AERIAL VEHICLES IN UNCERTAIN AND ADVERSARIAL ENVIRONMENTS* *. Tech. rep.

[39] *The Orocos Project – Smarter control in robotics & automation!* URL: https://orocos.org/ (visited on 11/26/2020).

[40] *ORCA Robotics - Home*. URL: https://orcahub.org/ (visited on 11/26/2020).

[41] *MOOS : Main - Home Page browse*. URL: http://www.robots.ox.ac.uk/{~}mobile/MOOS/wiki/pmwiki.php/Main/HomePage (visited on 11/26/2020).

[42] *Download Microsoft Robotics Developer Studio 4 from Official Microsoft Download Center*. URL: https://www.microsoft.com/en-us/download/details.aspx?id=29081 (visited on 11/26/2020).

[43] *RICS Research Group*. URL: https://rics.uninova.pt/ (visited on 11/26/2020).

[44] *ROS/Introduction - ROS Wiki*. URL: http://wiki.ros.org/ROS/Introduction (visited on 08/17/2020).

[45] *An Introduction to Robot Operating System (ROS) - Technical Articles*. URL: https://www.allaboutcircuits.com/technical-articles/an-introduction-to-robot-operating-system-ros (visited on 08/17/2020).

[46] *Messages - ROS Wiki*. URL: http://wiki.ros.org/Messages (visited on 08/17/2020).

[47] *ROS - Data display with Rviz | Stereolabs*. URL: https://www.stereolabs.com/docs/ros/rviz/ (visited on 08/18/2020).

[48] *tf - ROS Wiki*. URL: http://wiki.ros.org/tf (visited on 08/18/2020).

[49] *urdf - ROS Wiki*. URL: http://wiki.ros.org/urdf (visited on 08/18/2020).

[50] *rviz doesn't show any shape - ROS Answers: Open Source Q&A Forum*. URL: https://answers.ros.org/question/271357/rviz-doesnt-show-any-shape/ (visited on 08/18/2020).

[51] S. Chitta. "Moveit!: An introduction." In: *Studies in Computational Intelligence* 625 (2016), pp. 3–27. ISSN: 1860949X. DOI: 10.1007/978-3-319-26054-9_1. URL: https://link.springer.com/chapter/10.1007/978-3-319-26054-9{_}1.

[52] *MoveIt Quickstart in RViz — moveit_tutorials Melodic documentation*. URL: https://ros-planning.github.io/moveit{_}tutorials/doc/quickstart{_}in{_}rviz/quickstart{_}in{_}rviz{_}tutorial.html (visited on 08/18/2020).

[53] *The most powerful real-time 3D creation platform - Unreal Engine*. URL: www.unrealengine.com/en-US/ (visited on 08/19/2020).

[54] *Download - Unreal Engine*. URL: https://www.unrealengine.com/en-US/download (visited on 11/27/2020).

[55] *Professional 2D, 3D, VR, & AR software for cross-platform development of games and mobile apps. - Unity Store*. URL: https://unity.com/ (visited on 08/19/2020).

[56] *Announcing Our Collaboration with Epic Games to Create Cesium for Unreal Engine | cesium.com*. URL: https://cesium.com/blog/2020/06/04/cesium-for-unreal-engine/ (visited on 08/20/2020).

[57] *Home - AirSim*. URL: https://microsoft.github.io/AirSim/ (visited on 08/20/2020).

[58] *Aerial Informatics and Robotics Platform - Microsoft Research*. URL: https://www.microsoft.com/en-us/research/project/aerial-informatics-robotics-platform/ (visited on 08/20/2020).

[59] *ROS: AirSim ROS Wrapper - AirSim*. URL: https://microsoft.github.io/AirSim/airsim{_}ros{_}pkgs/ (visited on 08/20/2020).

[60] *Blocks Environment - AirSim*. URL: https://microsoft.github.io/AirSim/unreal{_}blocks/ (visited on 08/20/2020).

[61] *FlamMap | Fire, Fuel, and Smoke Science Program*. URL: https://www.firelab.org/project/flammap (visited on 08/21/2020).

[62] *Implementing DJI M100 Emulator in ROS-Gazebo for HITL | by Tahsincan Kose | Medium*. URL: https://medium.com/@tahsincankose/implementing-dji-m100-emulator-in-ros-gazebo-for-hitl-e79f4bb077f6 (visited on 11/29/2020).

[63] *Concepts | MoveIt*. URL: https://moveit.ros.org/documentation/concepts/ (visited on 09/22/2020).

[64] *How to Setup Reversed Propellers | GetFPV Learn*. URL: https://www.getfpv.com/learn/fpv-essentials/how-to-setup-reversed-propellers/ (visited on 09/24/2020).

[65] *Yaw, pitch and roll rotations of a Quadcopter. | Download Scientific Diagram*. URL: https://www.researchgate.net/figure/Yaw-pitch-and-roll-rotations-of-a-Quadcopter{_}fig1{_}280573614 (visited on 09/24/2020).

[66] *What are grids and graticules?—Help | ArcGIS for Desktop*. URL: https://desktop.arcgis.com/en/arcmap/10.3/map/page-layouts/what-are-grids-and-graticules-.htm (visited on 09/25/2020).

[67] *S-290 Unit 10: Fuel Moisture*. URL: http://stream1.cmatc.cn/pub/comet/FireWeather/S290Unit10FuelMoisture/comet/fire/s290/unit10/print.htm (visited on 09/26/2020).

[68] *NWCG - crownfire*. URL: https://www.nwcg.gov/publications/pms437/crown-fire/active-crown-fire-behavior (visited on 09/27/2020).

[69] *navigation/Tutorials/RobotSetup - ROS Wiki*. URL: http://wiki.ros.org/navigation/Tutorials/RobotSetup (visited on 10/16/2020).

[70] *REP 103 – Standard Units of Measure and Coordinate Conventions (ROS.org)*. URL: https://www.ros.org/reps/rep-0103.html (visited on 10/19/2020).

[71] R. Smith, M. Self, and P. Cheeseman. "Estimating Uncertain Spatial Relationships in Robotics." In: *Autonomous Robot Vehicles*. Springer New York, 1990, pp. 167–193. DOI: 10.1007/978-1-4613-8997-2_14. URL: https://link.springer.com/chapter/10.1007/978-1-4613-8997-2{_}14.

[72] *move_base - ROS Wiki*. URL: http://wiki.ros.org/move{_}base (visited on 10/27/2020).

[73] *costmap_2d - ROS Wiki*. URL: http://wiki.ros.org/costmap{_}2d (visited on 10/28/2020).

[74] *base local planner - ROS Wiki*. URL: http://http://wiki.ros.org/base_local_planner (visited on 10/29/2020).