POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

A Model Predictive Sample-Based Trajectory Planning Strategy for UAS

Supervisors

Prof. Alessandro RIZZO

Dr. Stefano PRIMATESTA

Candidate

Alessandro PAGLIANO

December 2020

Summary

Trajectory planning is an essential element in robotics applications. The quality of the planned trajectory strongly influences the robot behavior, particularly when an autonomous robot operates in complex and structured environments.
This thesis focuses on the study, deployment and testing of a trajectory planning algorithm for autonomous Unmanned Aerial Vehicles (UAVs), exploiting the Model Predictive Control (MPC) theory and the Rapidly-Exploring Random Tree "Star" (RRT*) algorithm. The developed logic makes use of the RRT* algorithm to explore the search space and, then, construct an incremental optimal tree to connect a given start and a goal pose in the search space. During the exploration phase, whenever two states are attempted to be connected, the deployed Model Predictive Control logic computes a "cost-to-go" cost of moving between two adjacent states in the graph by predicting the motion of the UAV, exploiting its dynamic model. At the end of the search phase, RRT* returns the asymptotically optimal path in the graph with the lowest cost. The resulting path consists in a sequence of states connected through the optimal motion computed by the MPC logic.
One of the drawback of the implemented logic is its complexity due to the MPC optimization that requires huge computational resources and time. Hence, some assumptions and heuristics are deployed for improving the quality of the algorithm. Finally, simulation results in realistic environments validate the implemented approach, proving how the proposed trajectory planner is able to compute an effective trajectory to be executed by the UAV.
This thesis is carried out within the activities of the Amazon Research Award "From Shortest to Safest Path Navigation: An AI-Powered Framework for Risk-Aware Autonomous Navigation of UASes" granted to Prof. Rizzo.

Acknowledgements

Last months have been the toughest of my life, because of the pandemic and the several difficulties I encountered. Nevertheless, I have never stopped smiling, thanks to all the people around me.
First of all, I want say thanks to my family for all the sacrifices they did for me and for the values and love they have always shared with me. Particularly, I want to spend some words for my sister. You are the most important person in my life, the person that, with her simplicity, always manages to pull a smile out of my face. Please, do not change, because, even if you are young, you are an example to follow. Thanks to my great-grandparents, that I know they are happy for me. I want to say that I gave my all to finish this thesis as soon as possible, in order to share this joy with Grammy, but unfortunately I didn't make it. What I want to say is that, if someone asked me to imagine the perfect family, I would not be able to imagine a better family than the one I have.
Thanks to all my friends that are always willing to help me and that always encourage me in what I do. Thanks to all my university friends with whom I shared beautiful and though moments, spending entire days at the Politecnico. Thank you all my friends for all the moments you shared with me, hoping to see you again and again in this life.
I want to thank Professor Alessandro Rizzo for having given me the opportunity of developing this thesis. A really big thanks to my supervisor Stefano Primatesta that has always helped me, replying to my stressful messages and supporting me at every stage of this project.
During this course of study I had beautiful experiences that I will never forget. Therefore, I dedicate this thesis to all the people I love; thank you for having accompanied me on this beautiful journey. I will be forever grateful to all of you.

Alessandro

Table of Contents

List of Tables

List of Figures

Acronyms

ACO
Ant Colony Optimization

MPC
Model Predictive Control

OMPL
Open Motion Planning Library

PRM
Probabilistic Roadmap

ROS
Robot Operating System

RRT
Rapidly-Exploring Random Tree

RRT*
Rapidly-Exploring Random Tree "Star"

SITL
Software In The Loop

UAS
Unmanned Aerial System

UAV
Unmanned Aerial Vehicle

Chapter 1

Introduction

Some of the most challenging issues in the Robotics field are related to motion planning. New generation robots are suppose to work with high speed, low consumption movements, in order to perform high number of actions for long periods of time. For the already mentioned reasons, motion planning plays a fundamental role in this sense. In fact, the quality of the planning strongly influences the quality of the actions performed by robots. When robots execute harsh maneuvers, it is necessary to avoid to damage their components because of the excessive forces and accelerations necessary for performing desired movements, so these precautions are intrinsically defined by the planner.
A popular motion planning problem often used as an example to explain a practical motion planning is the so called Piano Mover's Problem [1], where a piano is supposed to be moved in a known house, from a room to another, without hitting obstacles and walls. Going deeper inside the motion planning theory, it is possible to distinguish between path planning and trajectory planning, two central topics in the Robotics industry, particularly when automation is requested.
Differences between these two planning theories need to be presented. Path planning algorithms are able to return a continuous path between two given robot configurations; trajectory planning algorithms, instead, return a continuous path between two robot configurations marked with a timing law. In this sense, trajectory planning algorithms affect robot kinematics and dynamics. As properly presented in [2], path planning algorithms are usually divided according to the methodologies used for generating the geometric path:

- Roadmap Techniques;
- Cell Decomposition Algorithms;
- Artificial Potential Methods.

Trajectory planning algorithms are named on the basis of the function that the algorithm tries to minimize:

- Minimum Time;
- Minimum Energy;
- Minimum Jerk.

Hybrid algorithms optimizing multiple functions are presented in literature.

Historically, as presented in [3], two fields have contributed to trajectory or motion planning methods, particularly with Unmanned Aerial Systems (UASes): robotics, and dynamics and control; the former focuses on computational issues and robot control, while the latter focuses on the dynamic behavior and more specific aspects of trajectory performance. Concerning the UAV case, the motion planning represents a real challenging problem because of the different nature of this systems with respect to the common mobile robots or manipulators. Qualities characteristic to UAVs include non-trivial dynamics, three-dimensional environments, disturbances and uncertainties in state knowledge [3].
In this thesis, a different kind of trajectory planner is proposed.
The idea is to develop an hybrid approach inspired to [4], in which the Closed-Loop Rapidly-Exploring Random Tree algorithm is proposed. The basic idea is to evaluate the dynamical feasibility of the trajectory by running forward a simulation of the closed-loop system, consisting of the vehicle model and the controller. In this way, even the presence of obstacles can be easily handled by the planner, as well as dynamic and kinematic limitations. For the mentioned reasons, the attention is focused on applying the Model Predictive Control logic together with a State of the Art planning algorithm in order to a-priori evaluate the trajectory feasibility.
This thesis is carried out within the activities of the Amazon Research Award "From Shortest to Safest Path Navigation: An AI-Powered Framework for Risk-Aware Autonomous Navigation of UASes" granted to Prof. Rizzo.

1.1 State of the Art

This section is devoted to the presentation and comparison of some algorithms used in robotics for accomplishing the task of finding a feasible path between an initial and a final position in a search space.
It is important to precise that not all the algorithms in the literature are mentioned in this Section, but only some of the most curious and famous.
For each algorithm presented below, it is tried to explain how it works, how it has been used or how it may be used for accomplishing UAV trajectory planning queries, highlighting advantages and disadvantages of each approach.

At the end of this section, one algorithm is chosen as candidate algorithm for the thesis.

1.1.1 Dijkstra's Algorithm

The Dijkstra's Algorithm is probably the oldest algorithm for finding the shortest path between vertices in a graph. This algorithm is able to find a path depending on edges cost, that can be considered as a "cost-to-go" from a starting node to a goal one.

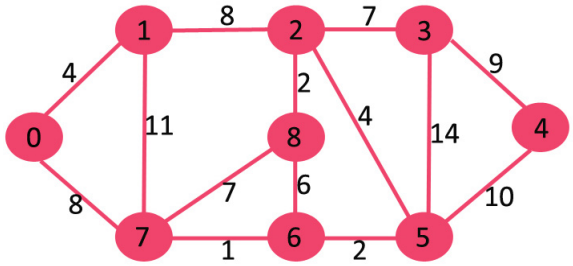


Figure 1.1: Dijkstra's Algorithm grid example.[1]

Explanation

The aim of the algorithm is to find the shortest path between any two vertices in a graph. Given a set of vertices, Dijkstra's Algorithm first reports the shortest distance from a previous vertex (i.e. the lowest cost for connecting the current vertex to another vertex). In this way, it is possible to easily find out the lowest cost path for reaching the goal vertex. Defined a starting vertex, the algorithm iteratively repeats the following phases until all vertices are visited:

- Visit the nearest unvisited vertex to the start one;
- Examine the current vertex unvisited neighbour vertices;
- Calculate the distance (cost) of each neighbour from the starting vertex;
- Update the shortest distance whenever it is shorter than the known distance;
- Update the previous vertex for each of the updated distances;
- Mark the current vertex as visited;
- Move to the next unvisited vertex.

[1] Image courtesy from Geeksforgeeks website.

Limitations

Even if it is very fast and computationally simple, Dijkstra's Algorithm wastes time by performing a "blind" research, computing unnecessary calculations. Moreover, Dijkstra's Algorithm is able to return the global optimal solution with respect to a quantifiable variable, like length or another cost. However, in several problems, multivariable functions are asked to be minimized. Because of these impediments, this algorithm is modified by adding some heuristics to speed up the algorithm.

1.1.2 A* Search Algorithm

A* Search Algorithm is one of the most popular technique used in path finding and graph traversals because of its simplicity and its quick solution finding capability. It is an extension of the Dijkstra's Algorithm that allows to add some heuristics for the path selection.

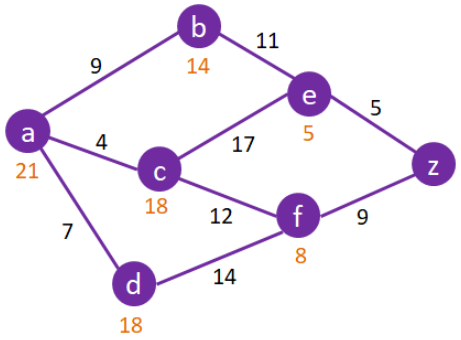


Figure 1.2: A* Search Algorithm grid example.[2]

Explanation

Consider a square grid having many obstacles. It is desired to plan the optimal trajectory from a starting cell to a target one as quick as possible. A* Search Algorithm, at each computational step, picks a node x computing the cost $f(x)$, equal the sum of two functions, $g(x)$ and $h(x)$. At each iteration step, the node/cell with the lowest $f(x)$ value is chosen.
$g(x)$ and $h(x)$ values are defined as follow:

g(x): The cost to move from the starting point to a given cell on the grid, following the path generated to get there, the same used in the Dijkstra's Algorithm.

[2]Image courtesy from 101computing website.

h(x): The estimated cost to move from a given node on the grid to the final destination. In order to guarantee the optimal solution in the graph, the heuristic must be admissible, i.e. it must never overestimate the cost of moving toward the goal node.

The main advantage of this algorithm is that the distances used as a criterion can be accepted, modified or another distance can be added. This allows a wide range of modifications of this basic principle, so that time, energy consumption or safety can be also included in function *f(x)*. Moreover, because of the introduced heuristic, world information can be taken into account [5].

Limitations

A* Search Algorithm doesn't produce always the shortest path because it heavily relies on heuristics/approximations to calculate *h(x)*. Moreover, as explained in [6], this algorithm provides a solution that is a sequence of vertices to be followed in order to move from a starting position to a target one.

1.1.3 Genetic Algorithm

Inspired by Charles Darwin's Evolutionary theory, Genetic Algorithm is an algorithm that reflects the process of natural selection, where fittest individuals are selected for reproduction, in order to produce progeny of the next generation.

Explanation

The process of natural selection starts with the selection of fittest individuals from a population. The results of this population are offspring that inherit parents' characteristics. Better fitness parents produce better offspring with higher chances to survive. The process iteratively repeats until the best offspring is found. Same iterative approach can be used for finding best problem solutions.
Genetic algorithm can be described with five phases[3]:

Initial Population: The process begins with a set of individuals which is called a Population. Each member of this Population is a solution to the problem. An individual is characterized by a set of parameters (variables) known as Genes. A set of Genes is joined into a string to form a Chromosome (solution). Often, Genes are described by alphanumerical characters.

[3]Definition courtesy from Towardsdatascience website.

Fitness Function: The fitness function represents a way for quantifying the quality of a solution, relative to the problem to be solved. Thanks to this function, each individual receives a score, called fitness score. Higher the fitness score, higher the probability of an individual to be chosen for the future generations.

Selection: The idea of selection phase is to select the fittest individuals and let them pass their Genes to the next generation. Depending on their fitness scores, two pairs of individuals, called parents, are selected.

Crossover: The crossover phase is very significant. Parents pair Genes are mixed until a predefined crossover point is reached. This point is randomly chosen, with the aim of increasing the probability of finding a solution to the problem.

Mutation: New generated offspring Genes can be conditioned by mutation with a low probability. This implies that some of the bits in the bit string can be flipped.

The algorithm ends when it results in saturation, i.e. new generations are very similar to parents. In this way, a solution to the problem is found.
A three-dimensional trajectory planning based on genetic algorithm, as reported in [7], is done by randomly generating a route from the initial three-dimensional position $(x_i,\ y_i,\ z_i)$ to the final one $(x_f,\ y_f,\ z_f)$, and finding out an optimized trajectory by minimizing the sum of the Euclidean distances between couples of points in the 3D space.

Limitations

This method is efficient for trajectory planning where the obstacles are static. Moreover, it is really effective in minimizing the distance from a start to a goal, but it is not able consider dynamic and kinematic constraints of the UAV.
Moreover, as shown in [8], a successful solution depends on the success in encoding or representing Chromosomes.

1.1.4 Ant Colony Optimization

Ant Colony Optimization algorithm (ACO) belongs to the so-called swarm intelligence, a relatively new approach to problem-solving that takes inspiration from the social behaviors of insects and of other animals.[4]
In particular, ACO is inspired by the behavior of some ant species; these ants

[4]Definition courtesy from Towardsdatascience website.

deposit pheromone on the ground in order to mark some favorable path that should be followed by other members of the colony. Similar mechanism is used for solving optimization problems.

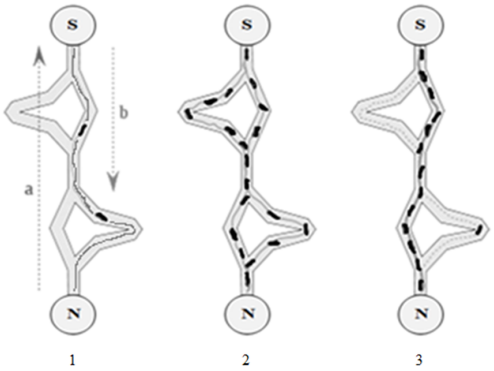


Figure 1.3: ACO path example.[5]

Explanation

This algorithm is introduced based on the foraging behavior of ants for seeking a path between their colony and food source. While searching, ants roaming around their colonies. While moving, they deposit an organic compound called pheromone on the ground. Ants communicate with each other via pheromone trails. When an ant finds some amount of food, it carries as much as it can carry. When returning to the colony, it deposits pheromone on the paths based on the quantity and quality of the discovered food. Ant can smell pheromone, so, other ants can follow that path. Higher the pheromone level, higher the probability of choosing that path and the more ants follow the path, higher the amount of pheromone deposited on the pathway, that results in higher probability of choosing it as preferential route by other ants.

Limitations

As properly explained in [9], ACO can fall into local minima during the path research, so it is not popular to be used for trajectory planning, while it is widely used for problems optimization.

[5]Image courtesy from Towardsdatascience website.

1.1.5 Probabilistic Roadmap Planner

The Probabilistic Roadmap Planner (PRM) is a motion planner used in Robotics for determining a path between a starting robot configuration and a final one, while avoiding obstacles and collisions.

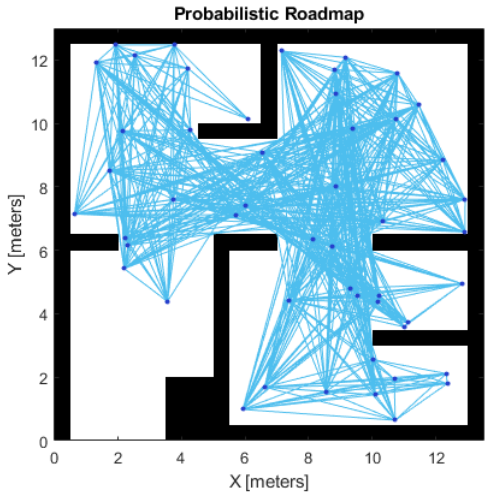


Figure 1.4: PRM graph example.[6]

Explanation

The concept behind PRM is to take random samples from the robot configuration space, ensuring that they are in the free space and, then, trying to connect them to nearby configurations. Then, starting and goal configurations are added in, and a graph search algorithm is applied to the resulting graph in order to determine a path between the starting and goal configurations. PRM works by uniformly sampling the free space, trying to connect the samples to form a roadmap of the free space. PRM has two main phases:

Construction Phase: In this phase, the graph is created by adding random configurations and trying to connect them to some neighbor ones. Whenever the graph is dense enough, this part is completed;

Query Phase: In this phase, start and goal are added to the graph, and the shortest length roadmap is found.

[6]Image courtesy from Mathworks website.

Limitations

Because of the random nature of the samples, it is not sure that a solution is found, particularly inside environments with narrow passages. Furthermore, this algorithm is provably probabilistically complete: it means that the probability of finding a solution increases with the increasing number of samples.

1.1.6 Rapidly-Exploring Random Tree

Rapidly-Exploring Random Tree (RRT) is an algorithm used for the path-planning that attempts to construct a tree for exploring the robot space and finding out a suitable path for connecting a start and a goal position. This algorithm is really popular nowadays because of its capability of considering robot kinematic and dynamic constraints.

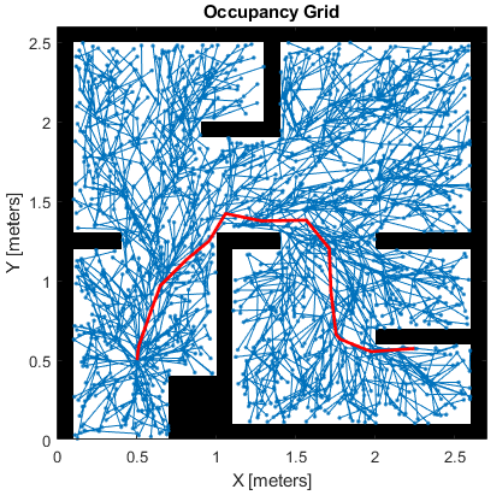


Figure 1.5: RRT graph and trajectory (in red) example.[7]

Explanation

After having defined a start and a goal position in the robot free space, a set of samples is generated. Commonly, a single tree is constructed starting from the initial position. At each sample, the sampled state is attempted to be connected to the nearest state in tree. If the connection is feasible, the sample is added to the tree. Whenever the nearest sample in the tree is farther to the sampled state

[7]Image courtesy from Mathworks website.

than a predefined distance called growth factor, this sample is substituted with another one on the same line of the original one, but at a distance equal to the growth factor. The probability of finding a solution depends on the number of samples, as well as on the sampling strategy. Higher the number of samples, higher the probability to find a path. In the same way, it is possible to grow two trees starting from initial and final state, with the aim to try to connect them, finding a suitable path with a higher probability; this variation is called Bidirectional RRT.

Limitations

Even if the probability of finding a solution is high also with a small number of samples, the obtained solution is not supposed to be optimal. In fact, RRT is not able to find a path that minimizes the distance between start and goal.

1.1.7 Rapidly-Exploring Random Tree "Star"

Similar to RRT, Rapidly-Exploring Random Tree "Star" (RRT*) is a path-planning algorithm that attempts to connect a starting and a goal state by constructing an optimal tree made of random samples. RRT* is an incremental sample-based algorithm which finds an initial path very quickly and later optimizes the path as the execution takes place [10].

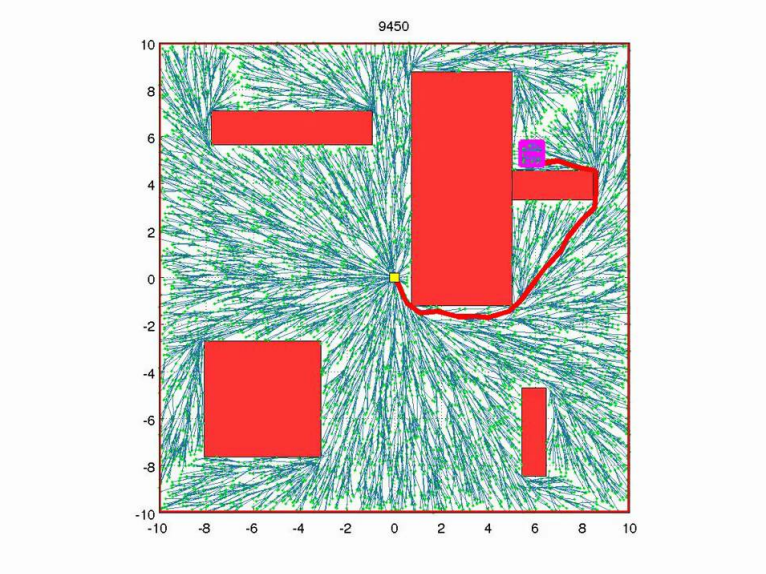


Figure 1.6: RRT* graph and trajectory (in red) example.

Explanation

The sampling theory used by RRT* is the same used by RRT: robot configurations are sampled in the robot free space. The main difference is the introduction of a minimization function. RRT* algorithm adds to the grown tree robot configurations whenever connection to parent node is collision free and with a minimum cost. In fact, new sample nodes are not connected to the nearest node, but to the node with the minimum cost. For each added node, the cost of the connection, defined by a cost function *c(t)*, is computed. Each node of the graph is marked with a cost relative to the connection to the parent node. In this way, for each new connection, the overall cost of the graph (i.e. the summation of each node cost) is updated. The algorithm ends after a predefined solve time or after a predefined number of nodes are sampled, and the returned graph is the one joining the start and the goal pose with the lowest cost according to the cost function $c(t)$.

Limitations

The limit of this algorithm is that, ideally, it converges to the optimal solution with an infinite number of iterations, meaning that the convergence to the optimal requires an infinite amount of time. However, this algorithm is able to provide a sub-optimal path solution in a limited period of time. For the mentioned reasons, together with the fact of being able to consider dynamic and kinematic constraints, RRT* is used as path-finding algorithm in this thesis, with an ad-hoc extension that allows to characterise the path with a timing law.

1.2 Outline of the Thesis

This thesis is structured as follows. **Chapter 2** presents the Model Predictive Control theory, the Open Motion Planning Library, the Robotic Operating System, as well as the Unmanned Aerial Vehicle model and notations adopted for developing this thesis, presenting some specificities. Of particular importance is **Chapter 3**, devoted to the deep explanation of the implemented logic; the chapter contains some code fragments used for setting the Model Predictive Control tool and better understanding the whole planner algorithm. In **Chapter 4**, simulations and testing results are presented; the effect of the different algorithm parameters on the final solution are treated, comparing graphical representations of the computed trajectories. Moreover, it also contains a simulation using SITL and Gazebo, proving that the planned trajectory can be really handled and performed by an Unmanned Aerial Vehicle. The thesis ends with **Chapter 5**, containing the overall analysis of the proposed algorithm, highlighting strengths and limitations of the implemented trajectory planner, as well as possible improvements.

Chapter 2

Background

2.1 Robot Operating System (ROS)

Robotic industry is quickly expanding; robots are rapidly becoming usual actors in every manufacturing sector. In order to accomplish different tasks in different working fields, different kinds of robots can be made of very dissimilar hardware, making the programming phase quite trivial. Moreover, being robots more and more complicate and smart, researchers and developers have to manage a big amount of codes from driver-level software to higher level like perception and artificial intelligence. In addiction, different programming languages can be used for designing a single robotic software, depending on developers' preferences. For the purpose of making the programming easier and more immediate, allowing robot developers to focus only on software, neglecting hardware-code integration, ROS has been created. Robot Operating System, known worldwide by the acronym ROS, is an open-source, meta-operating system for robots. As precisely expressed in [11], ROS is not a traditional operating system, addicted to process management and scheduling; indeed, it does not provide only hardware abstraction, low-level device control, tools and libraries for obtaining, building, writing, and running code across multiple computers, but, in addition, ROS provides structured communication layers above the host operating systems of a heterogeneous compute cluster. The philosophical goals of ROS can be summarized as:[1]

Peer-to-Peer: ROS-built systems processes are connected at run-time in a peer-to-peer topology. This characteristic allows system processes running on a single host or on multiple hosts to communicate each other via UDP protocols. The main actor in this communication is the Master process, that enables

[1]Guide courtesy from ROS website.

each component to synchronously or asynchronously communicate with any other, naming the different Nodes, making them able to be found;

Multi-Lingual: ROS is a language-neutral operating system. In this way, depending on the programming language preferences of each programmer, different languages can be used for the software design. ROS currently supports four languages: C++, Octave, Python and LISP. However, if necessary, support to other languages can be added by wrapping existing libraries. To support cross-language development, ROS uses a simple, language-neutral interface definition language (IDL) to describe the messages sent between modules;

Tool-Based: With the aim of avoiding monolithic kernel architecture, ROS is implemented with a microkernel structure where system components are made and run by a set of small tools. Using a decentralized run-time environment, the whole system is more robust and flexible;

Thin: ROS environment is designed so that code written for this platform can be used on other software frameworks. This is possible thanks to the lightness of ROS. In order to fully exploit this characteristic, the drivers and algorithms development should be done using standalone libraries, without ROS dependencies. The ROS build system performs modular builds inside the source code tree;

Free and Open-Source: ROS has been developed with the aim of being open-source. In fact, the entire ROS source code is public. The ROS distribution is performed under the terms of BSD license, allowing commercial and non-commercial project development.

2.1.1 ROS Resources Hierarchy

In this section, the ROS resources hierarchy on the storage medium is presented.

Packages: Packages are the key units in which ROS software is organized. Packages can carry different entities, like ROS run-time processes (Nodes), a ROS-dependent library, data sets, configuration files, or anything else that is usefully organized together. Packages aim to provide their useful functionalities in an easy-to-consume manner, in order to exploit software reusability;

Metapackages : Metapackages are particular Packages used for representing a group of related other packages for creating high level library. They are not intended for installing files or for containing tests, code, files and other items generally present in the packages;

Package Manifests: The package manifest, named package.xml, is an XML file that must be included with any catkin-compliant package root folder. This package is important because it is used for declaring different package properties like package name, version number, authors, maintainers, as well as dependencies on other catkin packages;

Message Type: In order to make ROS tools able to easily perform the automatic generation of the source code in different target languages, data values published by ROS Nodes are described through a simplified messages description language. Message files are marked with ".msg" extension and are composed of two parts: fields, that are the data sent inside of the message, and constants, that define useful values for interpreting those fields;

Services Type: Similarly to message types, ROS Service types are described in ROS using a simplified service description language. This is necessary for building directly upon the ROS ".msg" format to enable request/response communication between Nodes.

2.1.2 ROS Computation Graph Level

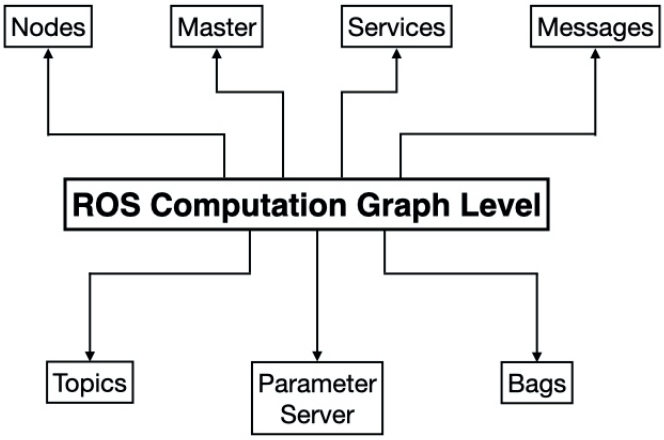


Figure 2.1: ROS Computation Graph Network.

In ROS, several processes compute data together in a peer-to-peer network, named ROS Computational Graph Level. The processes belonging to this network are:

ROS Nodes: Nodes are executable files inside ROS Packages. Nodes exchange information through message passing, using Topics for identifying message contents. In order to maintain the modularity principle on which ROS is based, a single robot process, like a control system, can be composed of several Nodes. In this way, errors can be restricted to a single Node, improving code readability, debugging and decreasing the probability of error spread up to the entire system.

Master: The ROS Master is the main actor of the Computation Graph for the Nodes peer-to-peer communication. Its main action is to store Topics and Services registration information for ROS Nodes. In this way, Master allows the ROS Nodes to locate each other, enabling peer-to-peer communication and direct Nodes connections. ROS Master provides an XMLRPC-based API, called by ROS client libraries, such as roscpp and rospy, for storing and retrieving information.

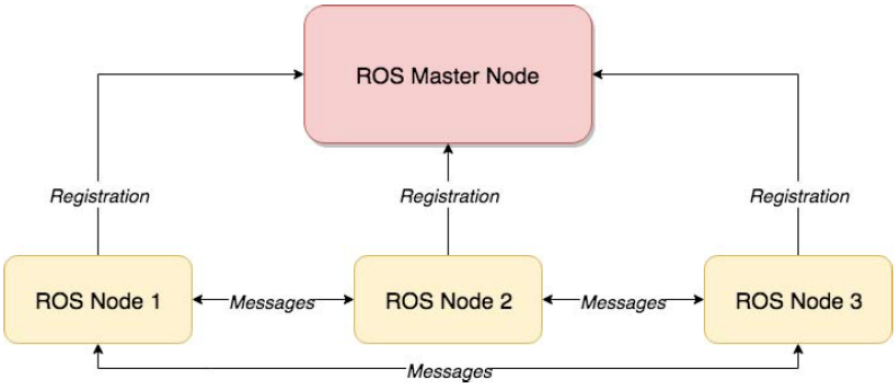


Figure 2.2: ROS Nodes - ROS Master relationship.[2]

Parameter Server: Parameter Server is a shared and multi-variable dictionary provided by ROS Master and accessible via network APIs. This server is used by system Nodes for storing and retrieving parameters at runtime. Thanks to its implementation on XMLRPC, Parameter Server API can be accessed via XMLRPC libraries.

[2]Image courtesy from Introduction to ROS, Clearpath Robotics, 2015.

Messages: ROS provides a message passing interface for Nodes communication. Messages are digital data structure, whose primitive types are integer, floating point, Boolean and array. Nested structures and arrays are allowed too.

Topics: ROS supports a publish-subscribe message pattern. Using this kind of semantics, providers and consumers of information are decoupled from each other, while they are logically connected through Topics. This asynchronous communication protocol is used for decoupling ROS Nodes, making them independent each other, easily maintainable and deployable. Topics are named logical channel used for sending or retrieving information. In fact, publisher Nodes publish data with a certain Topic, while all the Nodes that are subscribed to the same Topic receive those data. Different subscribers can subscribe to several single Topics, in the same way, different publishers can publish information on different Topics. In ROS, each Topic is strongly typed by the ROS message type used to publish to it and Nodes can only receive messages with a matching type. In this way, Nodes are able to understand messages arriving from a Topic they are subscribed to.

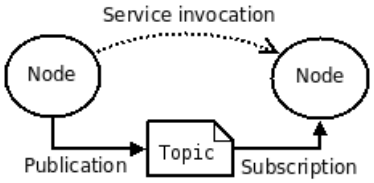


Figure 2.3: ROS Nodes - ROS Topic interaction.[3]

Services: Even if the asynchronous communication provided by publish-subscribe paradigm is a flexible message pattern, sometimes it is necessary to have a synchronous communication paradigm. For accomplishing this function, ROS provides a request-response communication pattern where Services work similarly to Topics in publish-subscribe pattern. Services are defined by a pair of messages, one for the request and one for the response. Services are offered by provider ROS Node under a string name, ROS Node client calls the Service by sending the request message and awaiting the response.

Bags: Bags are important ROS tools used for storing data published on a defined Topic. A Bag file, in fact, subscribes to desired ROS Topics and stores message data in the order they are received.

[3]Image courtesy from ROS website.

2.2 Open Motion Planning Library (OMPL)

Concerning the implementation of the motion planner for the thesis, it is decided to use a library able to interact with ROS. This library, better known as Open Motion Planning Library (OMPL), implements the basic primitives of sampling-based motion planning and allows users to modify the planners' settings, or use them as they are.
Sampling based motion planners, like PRM-based and RRT-based algorithms, are known for employing samples of the robot free state space and trying to connect them, via collision free paths that respect robot constraints, in order to reach a goal position, defined a starting one. The way the samples are connected differs from algorithm to algorithm. For more details, refer to Section 1.1. Most sampling-based methods provide probabilistic completeness, that means that the probability of finding a solution increases as the number of samples increases.

2.2.1 Problem Statement Definition

In this section, some terminologies, useful for better understanding the following parts, are presented.

Workspace: The workspace is the physical space where the robot operates in;

State Space: The state space is the parameter space where the robot operates in; it means that this space is the set of all the possible configurations that the robot is able to assume in the relative workspace;

Free State Space: The free state space is the obstacles-free subset of the state space;

Obstacle Space: The obstacle space is the subset of the state space occupied by obstacles;

Path: A path is a continuous sequence of states in the state space. A collision free path is a path made only by free state space elements.

2.2.2 OMPL Foundations

OMPL sampling-based motion planners are made of several components for solving planning queries. Thanks to the C++ based architecture of OMPL, these components are made by classes, where each class implements a single actor in the trajectory planning:

StateSampler: This class implements methods for uniform and Gaussian sampling robot configurations in the available state space;

NearestNeighbors: NearestNeighbors is a class used for searching nearest neighbours between samples in the desired state space;

StateValidityChecker: As previously mentioned, samples come from the entire robot state space, so it is necessary to check if the sampled states are valid or not. In fact, there could be some robot configurations that collide with obstacles, while others could not satisfy robot constraints. While all the previously mentioned classes are already available in OMPL, this class has to be defined by programmer and a callback to this method has to be provided by the user to the planner in order to detect and discard undesired states;

MotionValidator: MotionValidator is a class used for checking if the motion of the robot from a state to another is valid. Some motions could hit obstacles or not respect robot constraints, so it is necessary that a class returns whether the motion between a state to another is invalid or not;

OptimizationObjective: This class implements an abstract function used for defining an optimization objective to be used by the motion planner to search for an optimal solution. In fact, several motion planners like RRT* aim to solve the planning query by minimize a function. This class contains a function able to return costs that will be taken in account for the final path definition. For example, minimum path length could be achieved by implementing functions able to return cost proportional to the length of the considered path: longer the path, higher the cost;

ProblemDefinition: ProblemDefinition is a class used for specifying the desired motion query that is intended to be solved. This class needs as input parameters the start and the goal robot configurations (region surrounding particular states can be passed too) and the optimization objective to be met, if any.

Class hierarchy is reported in Figure 2.4.
One of the main advantages of using OMPL is the fact that it is object oriented; because of its nature, it is possible to inherit already existing components or create newer ones. Moreover, it is not necessary to define all the objects that compose OMPL architecture since for general motion queries most of the objects can be used in their default implementation.

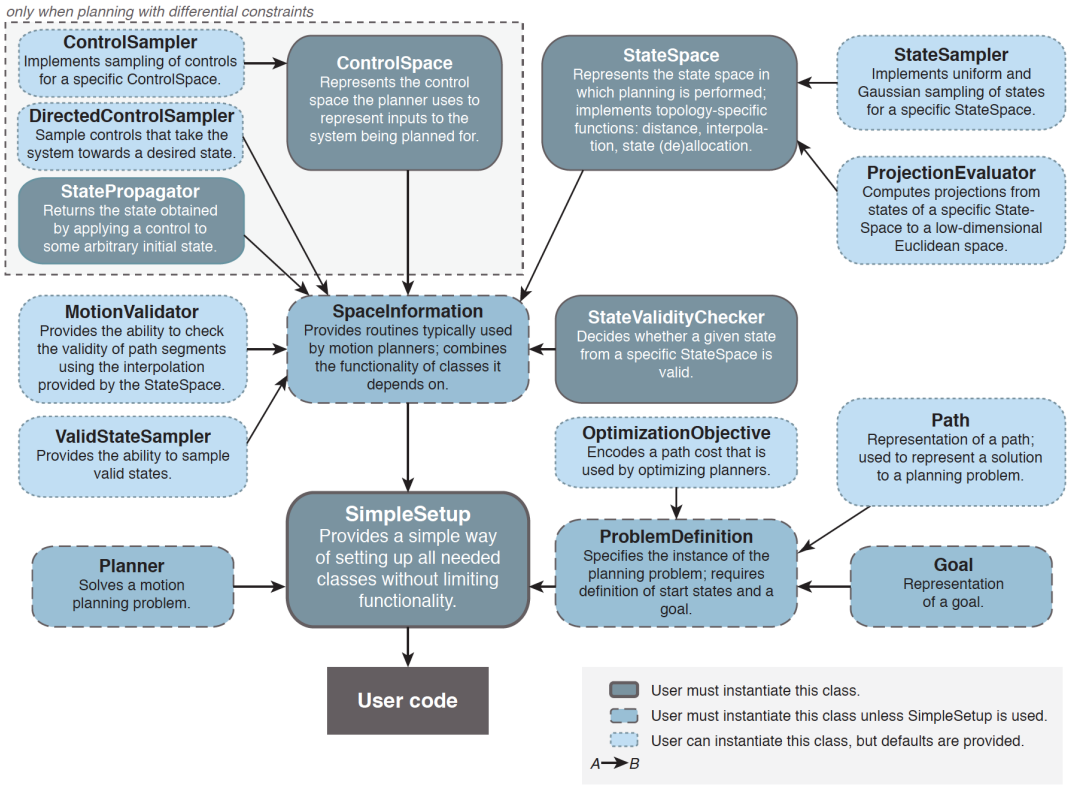


Figure 2.4: OMPL high level components hierarchy.[4]

[4]Image courtesy from OMPL website.

2.3 Model Predictive Control (MPC)

Modern industrial evolution is making control theory in the presence of constraints and optimization objectives more and more challenging. In the last 40 years Model Predictive Control (MPC) is become a milestone in the control theory and in industrial control applications for its capability of stabilizing plant including soft or hard constraints directly in the control input calculation.

2.3.1 MPC Theory

As widely presented in [12], MPC aims at solving constrained control problems with optimization demands. The need of having constrained input is really common nowadays, just think about actuators limitations and energy consumption reduction. The input signals limitations can be directly handled inside classical control approaches by saturating these signals whenever a critical value is reached. However, input saturation causes the feedback control system to become non-linear. Beside that, exceeding input bounds leads unexpected system behaviours such as bad time response, low performances and instability. Moreover, in the majority of cases, input limitations are checked a posteriori. The fact that, in nowadays control problems, output signals limitations are often required demonstrates the need of changing strategy, in particular when approaching Multiple Input Multiple Output (MIMO) systems.
MPC is a tactical constrained control, meaning that constraints are included from the beginning of the control input computation. The theory behind MPC is based on the prediction of the future behaviour of a system. In fact, starting from a mathematical system model and defining desired signal limitations, it is possible to track desired output values by generating prediction of the future output of the system in the defined time horizon. After that, a sequence of state and input values over the time horizon are calculated by minimizing an objective function, always considering the predefined constraints. Finally, adopting the Receding Horizon theory, only the first step input in the calculated input sequence is applied to the system. The process repeats measuring current output, predicting system behaviour and applying input signal until the desired output values are reached.
It is necessary to better explain the MPC theory for understanding the way this powerful tool solves control problem. For understanding the Receding Horizon principle, Linear Quadratic finite horizon optimal control must be introduced.

The discrete time finite horizon LQ design procedure considers the minimization of a cost function J of the input and state over a finite time horizon:

$$\min_{U(k|k)} J\Big(x(k|k), U(k|k)\Big) \tag{2.1}$$

where J is the cost function, $x(k|k)$ the state vector at time k and $U(k|k) = [u(k|k)\ u(k+1|k)\ \dots\ u(k+H_P-1)]^T$ is the control sequence computed at time k over the *prediction horizon* H_P, subjected to the following model constraint:

$$x(k+1) = Ax(k) + Bu(k) \qquad x(k) \in \mathbb{R}^n,\ u(k) \in \mathbb{R}^p \tag{2.2}$$

with $A \in \mathbb{R}^{n\times n}$ the state matrix and $B \in \mathbb{R}^{n\times p}$ the input matrix.
The optimal input sequence, marked with $U^*(k|k)$, is the input argument that minimizes the cost function J:

$$U^*(k|k) = arg \min_{U(k|k)} J\Big(x(k|k), U(k|k)\Big) \tag{2.3}$$

This input sequence is computed optimizing the predicted state response, given the system information at time k:

$$x(k+1|k),\ x(k+2|k),\ \dots,\ x(k+H_P|k)$$

On the basis of the measured state $x(k|k) = x(k)$ at time k, in fact, exploiting system model, it is possible to calculate the i^{th}-step ahead prediction:

$$x(k+i|k) = A^i x(k|k) + A^{i-1}Bu(k|k) + A^{i-2}Bu(k+1|k) + \cdots + Bu(k+i-1|k) \tag{2.4}$$

In order to take into account input saturation, the following constraints are added in the optimization problem:

$$\begin{gathered} u_{min} \le u(k|k) \le u_{max} \\ u_{min} \le u(k+1|k) \le u_{max} \\ \vdots \\ u_{min} \le u(k+H_P-1|k) \le u_{max} \end{gathered} \tag{2.5}$$

where $u_{min}, u_{max} \in \mathbb{R}^p$ are the vectors containing the saturation values for the p input of the considered system.
These input saturation constraints can be expressed as a set of linear inequalities in the variable $U(k|k)$.
The application of the minimizing sequence $U^*(k|k)$ causes an open-loop control strategy, so that, in order to address this problem, the Receding Horizon (RH) principle is applied.

The RH principle recursively performs the following actions:

- Get the state $x(k) = x(k|k)$;
- Solve the Quadratic Problem optimization related to the $U(k|k)$ and compute the minimizer $U^*(k|k) = [u^*(k|k)\ u^*(k+1|k)\ \dots\ u^*(k+H_P-1)]^T$;
- Apply the present control input $u(k) = u^*(k|k)$ to the system and discard the other optimized input;
- Repeat the procedure for the next time instant.

Recalling that cost function J depends on $x(k|k) = x(k)$ only, it can be shown that the Receding Horizon implicitly defines a nonlinear time invariant static state feedback control law of the form:

$$u(k) = K(x(k)) \tag{2.6}$$

After this introduction, it possible to present the Model Predictive Control (MPC) methodology. MPC is a control strategy that exploits a dynamical model of the plant to predict the future behaviour of the variables of interest to compute an optimal control action. MPC control input is computed by solving at each sampling time k the following Quadratic Problem:

$$\min_{U(k|k)} \sum_{i=0}^{H_P-1} \Big(x^T(k+1|k)Qx(k+1|k)+u^T(k+1|k)Ru(k+1|k)\Big)+x^T(H_P|k)Sx(H_P|k)$$

$$U(k|k) = [u(k|k)\ u(k+1|k)\ \dots\ u(k+H_P-1)]^T \tag{2.7}$$

$$\text{so that: } \begin{cases} x(k+1) = Ax(k) + Bu(k) \\ u_{min} \leq u(k+i|k) \leq u_{max}, i = 0, \dots, H_P - 1 \end{cases}$$

According to Receding Horizon principle, only the first element of the minimizer $u(k) = u^*(k|k)$ is applied to the system, and the process repeats iteratively.
In the following, some MPC advantages are recapped:

- MPC is a control procedure that allows to control Single Input Single Output as well as Multiple Input Multiple Output systems;
- It can be applied on linear and non-linear systems, providing high performances even in the presence of highly complex dynamics, instability and non-minimum phase systems;
- It allows to handle soft and hard input and output constraints directly in the control problem, providing better control performances;

- By using an Observer and a Kalman filter it is possible to estimate disturbances, system unmeasurable states and include them directly inside the control problem.

On the other hand, MPC presents several disadvantages that must be taken into account:

- Even if it is a really effective approach, MPC may require more sophisticated derivation for designing the control procedure than classical control approaches;
- MPC strongly relies on the plant mathematical model used for the states prediction. Most of the cases where MPC results infeasible are due to modelling errors, neglected disturbances or dynamics. For these reasons, it is very important to use a precise system model, in order to carefully take into account system dynamics;
- The biggest limit of the MPC approach is the computational effort that computers spend in order to solve the control problems. Moreover, better system performances can be obtained with longer prediction horizon, but higher the prediction horizon, higher the number of degrees of freedom in the optimization, so the problem complexity. A possible solution to this problem is reducing the number of variables to be optimized over a shorter time horizon named *control horizon* H_C. In this way, the system is predicted over the entire prediction horizon, but only the first H_C input variables are optimized, while the remaining $H_P - H_C$ control values may be set in different ways, reducing computational effort.

2.4 Multi-Rotor System Notation

As happened for most of the technological innovations upon the last centuries, Unmanned Aerial Vehicles are a technology developed for war purposes. Starting from early 19^{th}, big interest has been spent on the deployment of aerial vehicles flying without pilot. This kind of technology allows to inspect hostile unknown environments without any kind of risk as well as transporting goods through risky zones without jeopardizing human life [13]. Modern technologies have allowed to improve quality and performance of UAVs while reducing their size, so, because of the already mentioned characteristics, big attention is paying on UAVs. Referring to Civil UAVs, acronym that refers to unmanned aircraft that can be either autonomously guided or remotely controlled by a pilot and have the ability of performing a variety of missions, it is possible to split them in two categories:

- Multicopters;

- Fixed-wing.

The main difference between these two categories of UAVs is the mechanics adopted for generating lift, that influences the way wings and rotors are arranged in the UAV body. After this short introduction and before presenting the UAV model that is used in this thesis, it is necessary to explain several notations and terms that are used in the following parts.
It is important to precise that the developed project does not focus on the UAV model, while focusing on the trajectory planning strategy. Anyway, it is decided to use a multi-rotor system model. For the following explanations, a quadcopter drone will be considered. Moreover, the following considerations will be treated for a three-dimensional frame, while this thesis is developed in a two-dimensional one. Following the approach used in [14], the vehicle position and speed are defined with respect to a world inertial frame $\mathbb{W}$. Defined a vehicle body frame $\mathbb{B}$, the position vector is computed as the relative position of the drone body frame $\mathbb{B}$ origin, fixed in the vehicle center of mass, expressed in the world inertial frame $\mathbb{W}$. Both frames are *right-handed reference frames*. The model speed vector is defined as the derivative of the position vector in the given instant of time. Geometrically speaking, given a point in the 3D space, the following relationship can be defined:

$$p_B = \begin{pmatrix} x_B \\ y_B \\ z_B \end{pmatrix} = v_{AB} \cdot t + p_A = \begin{pmatrix} v_{AB,x} \\ v_{AB,y} \\ v_{AB,z} \end{pmatrix} \cdot t + \begin{pmatrix} x_A \\ y_A \\ z_A \end{pmatrix} \tag{2.8}$$

Being the system working in discrete time, for a matter of notation and completeness, the previous equation can be rewritten as follows:

$$p(t+1) = v(t) \cdot T_S + p(t) \tag{2.9}$$

where $p(t+1)$ and $p(t)$ are the positions at time $t+1$ and t, $v(t)$ is the vehicle speed at time t, and T_S is the sampling time.
UAV orientation in the space is described with Cardan angles convention, a particular representation of the Euler angles.

Remembering that the vehicle is described in a three-dimensional right-handed reference frame, it is possible to name the rotations as follows:

Roll ϕ: Rotation around the drone longitudinal axis;

Pitch θ: Rotation around the drone transverse axis;

Yaw ψ: Rotation around the drone vertical axis passing for its center of mass.

These rotations are considered in the vehicle body frame, so, in order to correctly translate information from the inertial world frame to the vehicle body frame it is

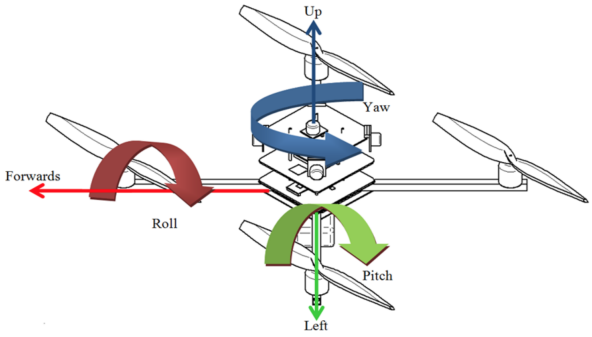


Figure 2.5: Quadcopter axes and movements description.

necessary to perform rotations.
Geometrically speaking, these transformations are performed by pre-multiplying the drone body frame orientations for the rotation matrices. This discussion will not be done in this part, but is presented in the part related to the linearization, precisely at Equation 2.19.
It is now time for explaining how the UAV moves and how some maneuvers are performed.
Quadcopters are made by four rotors arranged in the drone body forming a square. Two out of four rotors spin counterclockwise, while the others spin clockwise, with the rotors on the same diagonal spinning in the same direction. This arrangement is made for partially compensating drone internal forces. If the four rotors spin at the same rate, null angular speed results around the rotation axes. As a result, changing the angular speeds of the vehicle rotors causes resulting torques given by the decompensation of accelerations, making the UAV able to perform the desired maneuvers.
In the following, the most common drone maneuvers are listed:

Hovering: The drone hovering represents the condition of stationary flight at a certain altitude, meaning that the drone remains at a certain altitude without angle variations. This condition is obtained by making the four rotors spinning at the same rate, with each rotor compensating 1/4 of the drone weight;

Move Up and Move Down: In order to increase or decrease the altitude of the drone, the spins of the four rotors change identically, augmenting the four rotors rotational speeds in case of drone ascent maneuver, and decreasing them in case of drone descent maneuver;

Roll: Roll maneuvers are performed by increasing or decreasing the rotational speeds of the rotors of lateral axis (i.e. the perpendicular to the drone forward

direction). It is important to point out that, in order to not affect drone yaw, the rotor on the same axis are coupled in the sense that whenever the speed of a drone rotor is augmented for performing the desired maneuver, the one of the opposite rotor is decreased;

Pitch: Similarly to how the roll maneuver is done, the pitch is performed by increasing or decreasing the rotational speeds of the rotors of longitudinal axis (i.e. the drone forward direction);

Yaw: This movement is performed by increasing the rotational speeds of a couple of opposite rotors, while decreasing the ones of the other two rotors. In this way, an overall imbalance of moments causes a rotation around the vertical axis.

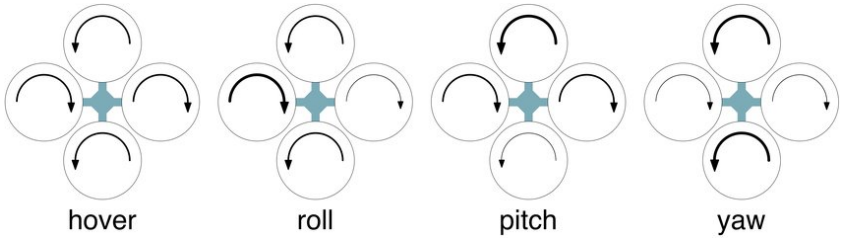


Figure 2.6: Quadcopter maneuvers description [15].

2.5 Multi-Rotor System Model

In this section, a simplified linear model of multi-rotor system is presented. The current model, studied by [14], is used for a model-based control to achieve trajectory tracking. The presented model describes a 6DoF multi-rotor system. Its body frame $\mathbb{B}$ is described with respect to a fixed world frame $\mathbb{W}$, with p the position of the origin of the vehicle body frame $\mathbb{B}$ with respect to the inertial world frame $\mathbb{W}$ described in $\mathbb{W}$ and R the rotation matrix of $\mathbb{B}$ in frame $\mathbb{W}$ expressed in $\mathbb{W}$. UAV angles are defined with respect to its body frame and they are marked with the Greek letters ϕ, θ and ψ, denoting roll, pitch and yaw angles respectively. The used non-linear drone model is presented below:

$$\dot{p}(t) = v(t) \tag{2.10}$$

$$\dot{v}(t) = R(\phi, \theta, \psi) \begin{pmatrix} 0 \\ 0 \\ T \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ -g \end{pmatrix} - \begin{pmatrix} A_x & 0 & 0 \\ 0 & A_y & 0 \\ 0 & 0 & A_z \end{pmatrix} v(t) + d(t) \tag{2.11}$$

$$\dot{\phi}(t) = \frac{1}{\tau_\phi}(K_\phi \phi_d(t) - \phi(t)) \tag{2.12}$$

$$\dot{\theta}(t) = \frac{1}{\tau_\theta}(K_\theta \theta_d(t) - \theta(t)) \tag{2.13}$$

where the vehicle speed v is expressed as the derivative of the position vector p, T is the mass normalized thrust, g represents the gravitational acceleration, A_x,A_y,A_z are the mass normalized drag coefficients, d is the external disturbance vector, while $\tau_\phi, K_\phi, \tau_\theta, K_\theta$ are the time constants and gains of the inner-loop behavior for roll angle and pitch angle respectively, and $R(\phi, \theta, \psi)$ is the rotation matrix from $\mathbb{W}$ to $\mathbb{B}$ presented in Equation 2.14 (cosine and sine are marked with c and s letters).

$$R(\phi,\theta,\psi) = \begin{pmatrix} c(\psi)c(\theta) & c(\psi)s(\phi)s(\theta) - c(\phi)s(\psi) & s(\phi)s(\psi) + c(\phi)c(\psi)s(\theta) \\ c(\theta)s(\psi) & c(\phi)c(\psi) + s(\phi)s(\psi)s(\theta) & c(\phi)s(\psi)s(\theta) - c(\psi)s(\phi) \\ -s(\theta) & c(\theta)s(\phi) & c(\phi)c(\theta) \end{pmatrix} \tag{2.14}$$

2.5.1 Linearization and Discretization

For the purpose of this thesis, the vehicle model is approximated around its hovering conditions, where small attitude angle variations are assumed and vehicle heading is null, so vehicle longitudinal axis is aligned with the inertial frame x axis. With the already presented conditions, the state space representation of the system can be rewritten as follows:

$$\dot{x}(t) = A_c x(t) + B_c u(t) + B_{d,c} d(t), \tag{2.15}$$

where A_c is the state matrix defined as:

$$A_c = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -A_x & 0 & 0 & g & 0 \\ 0 & 0 & 0 & 0 & -A_y & 0 & 0 & -g \\ 0 & 0 & 0 & 0 & 0 & -A_z & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{\tau_\phi} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{\tau_\theta} \end{pmatrix}$$

B_c is the input matrix equal to:

$$B_c = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \\ \frac{K_\phi}{\tau_\phi} & 0 & 0 \\ 0 & \frac{K_\theta}{\tau_\theta} & 0 \end{pmatrix}$$

$B_{d,c}$ is the disturbance matrix:

$$B_{d,c} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

x is the state vector $[p^T, v^T, {}^{\mathbb{W}}\phi, {}^{\mathbb{W}}\theta]^T$, u is the input vector equal to $[{}^{\mathbb{W}}\phi_d, {}^{\mathbb{W}}\theta_d, T]^T$ and d is the disturbance vector equal to $[d_x, d_y, d_z]^T$. Entities marked with subscript c are continuous time models.
Being the controller implemented in discrete time, system dynamics is discretized as follows:

$$A = e^{A_c T_s} \tag{2.16}$$

$$B = \int_0^{T_s} B_c d\tau \tag{2.17}$$

$$B_d = \int_0^{T_s} B_{d,c} d\tau \tag{2.18}$$

where T_s is the *prediction sampling time*.
It is important to point out that, in this linearization, the attitude is marked in the world inertial frame $\mathbb{W}$ for removing the yaw angle ψ dependence from the model. ${}^{\mathbb{W}}\phi_d$ and ${}^{\mathbb{W}}\theta_d$ control actions are computed in the world frame, so they have to be converted to body frame corresponding angles by performing a rotation around the z axis.

$$\begin{pmatrix} \phi_d \\ \theta_d \end{pmatrix} = \begin{pmatrix} \cos\psi & \sin\psi \\ -\sin\psi & \cos\psi \end{pmatrix} \begin{pmatrix} {}^{\mathbb{W}}\phi_d \\ {}^{\mathbb{W}}\theta_d \end{pmatrix} \tag{2.19}$$

As advised by Kamel, Stastny, Alexis and Siegwart, control input should undergo a feed-forward compensation before being applied to the system, in order to compensate coupling and achieve better tracking performances [14].

Chapter 3

Software Implementation

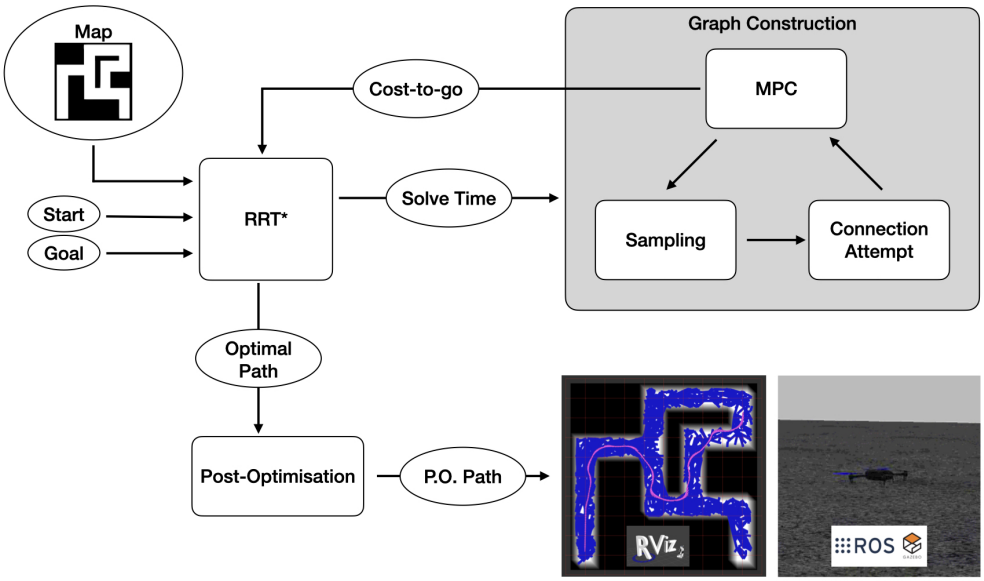


Figure 3.1: Algorithm logic scheme.

This section focuses on the explanation of the proposed trajectory planner. The implemented logic is developed in C++ programming language because of its compatibility with ROS and OMPL. For this thesis, OMPL is used to implement the planning algorithm to plan the UAV trajectory. As explained in Section 1.1, RRT* algorithm is chosen as the candidate planner for its characteristic of cost minimization. Exploiting OMPL object oriented nature, a newer class for computing a trajectory optimization with MPC approach is created. This function is set as default optimization objective for the path computation.

In fact, whenever a new state is sampled as candidate child node in the chosen state space, the node and its parent node are passed to the optimization class that computes the trajectory for moving from the parent node to the sampled child. If the sampled child satisfies specific characteristics, both states are transformed in ROS poses and the MPC implemented logic is called for predicting the intermediate poses and computing the relative cost of moving. If child pose does not match particular characteristics, the class returns an infinite "cost-to-go", marking the trajectory as unfeasible.
After the defined search time, the sub-optimal path (i.e. the path with the lowest cost) is chosen as candidate path and then, exploiting again the MPC approach, an optimized trajectory is returned. The proposed algorithm contains function for simplifying the resulting sub-optimal path.
This thesis is developed using a two-dimensional state space, configuring OMPL for sampling 2D poses in the the *Special Euclidean Group* with dimension two (*SE(2)*). The implemented software is able to detect the presence of obstacles during the path cost computation by exploiting the predictive capability of the MPC tool.

3.1 Code Details

In this part, the implemented code is deeply described, highlighting its structure, its functioning and its peculiarities.
Several code fragments are presented, trying to make the explanation of the proposed trajectory planning algorithm clearer.

3.1.1 RRT* Algorithm

The first component that needs to be presented is the RRT* planner. In order to exploit the tools offered by OMPL, a class inheriting most of its components is created. As properly presented in Section 2.2, OMPL contains default functions for performing the trajectory planning. These functions are designed for being customized according to the programmer's needs.
First of all, it is necessary to present the motion planning problem.
Consider a system dynamically described by the following dynamic reported in Equation 2.2 of the form: $\dot{x}(t) = f(x(t), u(t))$, where $x(t) \in X$ is the state vector at time t with $X \subseteq \mathbb{R}^8$ the state space, $u(t) \in U$ is the input vector at time t with $U \subseteq \mathbb{R}^3$ the input space. Let X_{obs} indicate the obstacle region, $X_{free} = X \backslash X_{obs}$ the obstacle-free space and $X_{goal} \subset X$ the goal region. The motion planning problem is to find a control input set $u : [0, T] \rightarrow U$, that allows to obtain a feasible path $x(t) \in X_{free}$ for $t \in [0, T]$ from an initial state to a goal region $x(T) \in X_{goal}$ [16], satisfying system dynamics.
The already explained logic represents the basis of the RRT algorithm; RRT*

contains an important improvement for the path computation, that is the research of an optimal path with respect to a given cost function $c(t)$. The optimal motion planning problem targets the minimization of a cost function $c(t)$ for the feasible path computation. In fact, each admissible trajectory is tagged with a real number representing the relative cost for performing the desired motion. In solving the optimal motion planning problem, the RRT* algorithm builds and maintains a tree $G = (V; E)$ comprised of a vertex set V of state from X_{free} connected by directed edges $E \subseteq V \times V$[16].
In the following, a set of basic procedures is listed for introducing the RRT* logic[17]:

Sampling : The first function to be introduced is the one devoted to the sampling of states in the space. This function randomly samples a state z_{rand} in the robotic free space X_{free};

MPC Logic: $motionCost\ X \times X \rightarrow \mathbb{R}^+$ is the function that solves the optimization problem defined in the MPC logic, given an initial and a final state, and returns the cost of the optimized trajectory. If an obstacle is encountered during the optimal trajectory computation or the final state does not satisfy certain characteristics, this function returns an infinite cost, marking the trajectory as unfeasible;

Nearest Neighbor: Given a state $z \in X$ and a tree $G = (V; E)$, $v = Nearest(G, z)$ is the function used for returning the nearest node in the tree G, in terms of Euclidean distance, to the passed state z;

Collision Check: $ObstacleFree(x)$ function checks whether a path $x : [0, T] \rightarrow X$ lies within the obstacle-free region of state space. The previous sentence means that each state of the resulting path belongs to the obstacles free state space X_{free}: $x(t) \in X_{free}, \forall t \in [0, T]$;

Distance Computation: A function called $computeDistance(x_1, x_2)$, $X \times X \rightarrow \mathbb{R}^+$ is the function that returns the Euclidean distance between two states in the state space.

Node Insertion: $InsertNode(z_{current}, z_{new}, G)$ is the function that adds the new sampled node z_{new} to V and creates an edge to connect the new state to the current node $z_{current} \in V$ as its parent. The already generated edge is added to E and the cost of the trajectory for reaching the newer state is added to the cost of the whole path up to z_{new}. Whenever the trajectory for joining a state has an infinite cost, the corresponding node is discarded.

The functioning of the RRT* algorithm is the same of RRT, with the only difference that whenever a node is added as vertex, the *"cost-to-go"* from the start

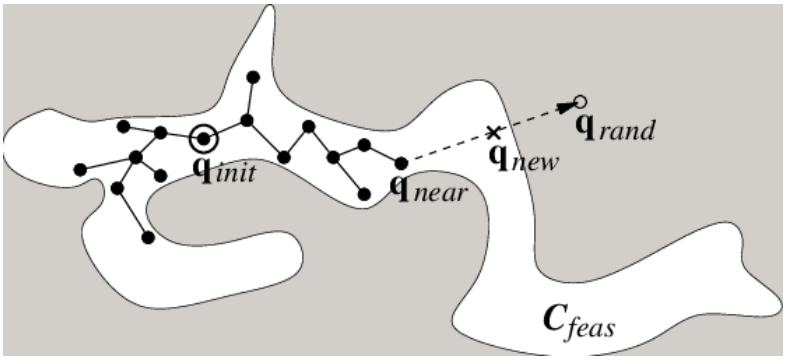


Figure 3.2: RRT* expansion phase [18].

state up to the added node z_{new} is increased by the cost of the trajectory for reaching z_{new} form its lowest cost near state in the vertex space V. In fact, it is important to highlight the choice of the parent made by the algorithm. The parent of a sampled state, according to RRT logic, is the nearest node in the graph, instead, RRT* chooses the lowest cost parent according to the defined cost function, from a set of near parent nodes. Initially, the sample is attempted to be connected to its nearest vertex in the graph in the circular area of radius equal to ρ around its nearest node, with ρ the *Planner Range*. Samples are performed in the whole robot free space, but, whenever a state is randomly sampled in this space, if the sampled state is farther to its nearest node than ρ, the parent nearest node connection is attempted with a state on the same line joining the sampled node and its parent, but at a distance equal to ρ.
Then, being Z_{near} the set of near nodes to the sampled one, for each node z_{near} in this set, the motion cost between it and the sample is computed. Discovered the near parent with the lowest cost of moving, the sample is connected to it and a rewiring action is performed. For each sample $z_{near} \in Z_{near}$ in the vicinity of z_{new}, a connection check is done to see whether reaching z_{near} via z_{new} would achieve lower cost than doing so view its current parent [16]. If this connection reduces the total cost associated with z_{near}, the tree is modified to make z_{new} the parent of z_{near}.
An example of this state connection is shown in Figure 3.2.

The algorithm logic is presented in the following:

Algorithm 1: G=(V,E)← RRT*(z_{init})

```
G ← InitializeThree();
G ← InsertNode(∅,z_init,G);
for i = 1 to i = N do
    z_rand ← Sample(i);
    z_nearest ← Nearest(G,z_rand);
    if ObstacleFree(x_new) then
        Z_near ← Near(G,z_new,|V|);
        z_min ← ChooseParent(Z_near,z_nearest,z_new,x_new);
        G ← InsertNode(z_min,z_new,G);
        G ← ReWire(G,Z_near,z_min,z_new);
return G
```

Algorithm 2: z_{min} ← ChooseParent(Z_{near},$z_{nearest}$,x_{new})

```
z_min ← z_nearest;
c_min ← motionCost(z_nearest)+c(x_new);
for z_near ∈ Z_near do
    (x',u')←motionCost(z_near, z_new);
    if ObstacleFree(x') and (x'(G')=z_new or i<iter_max) then
        c'=motionCost(z_near, z_new)+c(x');
        if c'<motionCost(z_near, z_new) and c' < c_min then
            z_min ← z_near;
            c_min ← c';
return z_min
```

Algorithm 3: G ←ReWire(G,Z_{near},z_{min},z_{min})

```
for z_near ∈ Z_near \ z_min do
    (x',u',G')← motionCost(z_near, z_new);
    if ObstacleFree(x') and x'(G') = z_near and
     motionCost(z_near, z_new)+c(x')< motionCost(z_near, z_new) then
        G ←Reconnect(z_new,z_near,G);
return G
```

Even if the drone model allows to perform a three-dimensional trajectory planning, the thesis project is developed for planning in a two-dimensional space. Being more precise, the code works in 3D because UAV states and references exploit all the model characteristics, but data related to z coordinate are set to zero (i.e.

reference z position and reference z speed).
Because of the fact that the drone model does not need roll and pitch references, being the planning performed in 2D, the used OMPL state space is *SE(2)*.
SE(2) is the Special Euclidean Group of dimension 2; in general *SE(n)* is a topological space and it is homeomorphic to $\mathbb{R}^n \times SO(n)$, with $\mathbb{R}^n$ the Euclidean space of dimension n and $SO(n)$ the rotation group of all the rotations about the origin of n-dimensional Euclidean space.
With the library configured for sampling in this state space, each sample contains two data describing state position in the x-y plan (i.e. abscissa and ordinate coordinates) and one describing the state orientation in the same space.
In order to properly exploit the characteristics of the OMPL sampled states, it is decided to use two-dimensional poses in the x-y plan. For doing that, ROS *Pose2D* message type is used. This data structure, contained in the ROS geometry message work-space, allows to construct objects made by three attributes:

- $float64$ x position;
- $float64$ y position;
- $float64$ θ orientation (i.e. drone yaw).

OMPL sampled states need to be translated from their native data structure to the ROS *Pose2D* one. For doing that, a function, named *SE2ToROSPose2D*, is developed. It exploits OMPL *SE(2)* state space methods that are automatically able to transform states characteristics into double quantities, saved in the *Pose2D* attributes:

Listing 3.1: SE2ToROSPose2D function code.

```
1  void OmplPlanner::SE2ToROSPose2D(const ompl::base::State*
   ompl_state, geometry_msgs::Pose2D& pose2D)
2  {
3      pose2D.x = ompl_state->as<ompl::base::SE2StateSpace::
   StateType>()->getX();
4      pose2D.y = ompl_state->as<ompl::base::SE3StateSpace::
   StateType>()->getY();
5      pose2D.theta = ompl_state->as<ompl::base::SE2StateSpace::
   StateType>()->getYaw();
6      return;
7  }
```

Explaining and showing the translation form OMPL states to ROS *Pose2D* is important because it is the first action performed by *MPCOptimizationObjective* class.

3.1.2 MPCOptimizationObjective

The novel feature of the proposed algorithm that differs from those presented in the literature is the path cost computation, that exploits the MPC logic. For accomplishing this task, a class named *MPCOptimizationObjective* is created. This class, sets as default class for performing the RRT* optimization through OMPL default *setOptimizationObjective* function, contains some methods needed for properly solving the MPC optimization problem and for generating the desired trajectory. OMPL automatically recalls two default functions for evaluating a cost in the motion query. These functions must be defined in the optimization class, and are named *stateCost* and *motionCost*. The former, evaluates the cost of a state, while the latter evaluates the cost for moving from am initial state to a final one arbitrarily defined in the state space. *stateCost* function does not perform any computation and it always returns a null cost; for the purpose of this thesis, in fact, it is not necessary to compute the cost of a single state, but it is necessary to compute the cost of moving from a certain pose to another one in the two-dimensional plan.
Particular attention is paid to the implementation of the *motionCost* function devoted to compute the cost for approaching a final pose from an initial one, exploiting the MPC logic. Moreover, this function is also exploited to define the optimal trajectory for moving between two poses. Before presenting the functions created for properly setting the MPC for the thesis purposes, it is mandatory to introduce an important tool used for generating the MPC desired code.
As treated in Section 2.3, the biggest limit of the MPC approach is the computational effort requested for solving the optimization problem, and higher the problem complexity (i.e. the number of optimization variables), higher the computational effort. This limit causes big time delays that are undesired in sample-based motion planner. In fact, higher the speed of the cost computation, higher the number of samples that the planner is able to analyze in a fixed amount of time and, concerning the RRT* algorithm, higher the probability of obtaining a better solution.
For the already mentioned problems, it is decided to use CVXGEN for the MPC code generation.
CVXGEN stands for Code Generation for Convex optimization. It is an online interface used for generating fast custom code for small, QP-representable convex optimization problems. As treated in [19], CVXGEN interface allows to describe the problem to be optimized with a simple and powerful language, to automatically create library-free C code for custom, high-speed solver that can be directly downloaded from the CVXGEN website.
Additionally, for same optimization problems, the solution provided by CVXGEN is twelve to one thousand-times faster than the solution offered by the most common optimizers [20] (further details at Appendix C).

Before showing the CVXGEN description of the problem, the optimization problem is presented. The MPC problem is defined by inspiring to [14], where a MPC-based trajectory tracking is developed using CVXGEN.
Assuming a disturbance free system model (i.e. $d(t) = 0 \ \forall t \in [0, H_P]$), the optimization problem can be defined as:

$$\min_{U,X} \Big(\sum_{k=0}^{H_P-1} (x_k - x_{ref,k})^T Q_x (x_k - x_{ref,k}) + (u_k - u_{k-1})^T R_\Delta (u_k - u_{k-1}) \Big) + \\ + (x_{H_P} - x_{ref,H_P})^T Q_{final} (x_{H_P} - x_{ref,H_P}) \tag{3.1}$$

subject to Equation 2.15:

$$\begin{aligned} &x_{k+1} = Ax_k + Bu_k; \\ &u_k \in U; \\ &x_0 = x(t_0) \end{aligned} \tag{3.2}$$

where H_P is the prediction horizon, U is the input space $U = [u_0 \ u_1 \ \dots \ u_{H_P-1}]^T$ with $u_k \in \mathbb{R}^3$ for $k = [0, \dots, H_P - 1]$, X is the state space $X = [x_0 \ x_1 \ \dots \ x_{H_P}]^T$ with $x_k \in \mathbb{R}^8$ for $k = [0, \dots, H_P]$, $X_{ref} = [x_0 \ x_1 \ \dots \ x_{H_P}]^T$ with $x_k \in \mathbb{R}^8$ for $k = [0, \dots, H_P]$, Q_x is the penalty on the state error, R_Δ is a penalty on the control change rate and Q_{final} is the terminal state error penalty. The computation of the terminal cost matrix Q_{final} is done by solving the Algebraic Riccati Equation iteratively [21].
analyzing the introduced objective function, it is clear that control input rate Δu_k and state error Δx_k are penalized. In this way, smooth control input can be obtained, as well as oscillations prevention.
After this small presentation, it is now possible to present the structure of the *motionCost* function. This function receives as input from the OMPL planner two states that are immediately transformed in ROS *Pose2D* after the MPC parameters initialization.
In order to properly fill the drone model matrices, the optimization function ones and the reference state vectors, the *initializeParameters* function is called. This function is called when *MPCOptimizationObjective* class is constructed. The initialization phase is quite long, so, being the MPC tool parameters fixed for the entire RRT* graph construction phase, it is performed only once inside the *MPCOptimizationObjective* class constructor. *initializeParameters* contains a ROS node handler that makes the function able to read parameters form a launch file (the test.launch file used in this thesis is detailed in Appendix B). Data read from this file are saved into global variables. A particular string value that is saved through *initializeParameters* is the desired state space. OMPL class and *MPCOptimizationObjective* are designed for sampling states even in other state

spaces, but the reference trajectory is only computed for a x-y plan, being z position and z speed assumed to be equal to zero.
In the following, the way state matrix A and input matrix B are filled is presented:

Listing 3.2: Drone model matrices definition.

```
1     Eigen::MatrixXd A_continuous_time(StateSize, StateSize);
2     A_continuous_time = Eigen::MatrixXd::Zero(StateSize, StateSize);
3     Eigen::MatrixXd B_continuous_time(StateSize, InputSize);
4     B_continuous_time = Eigen::MatrixXd::Zero(StateSize, InputSize);
5
6     Eigen::Matrix<double, StateSize, StateSize> model_A_;
7     Eigen::Matrix<double, StateSize, InputSize> model_B_;
8
9     std::vector<double> drag_coefficients_;
10
11    const double kGravity = 9.8066;
12    drag_coefficients_.push_back(0.01);
13    drag_coefficients_.push_back(0.01);
14    drag_coefficients_.push_back(0.0);
15    A_continuous_time(0, 3) = 1;
16    A_continuous_time(1, 4) = 1;
17    A_continuous_time(2, 5) = 1;
18    A_continuous_time(3, 3) = -drag_coefficients_.at(0);
19    A_continuous_time(3, 7) = kGravity;
20    A_continuous_time(4, 4) = -drag_coefficients_.at(1);
21    A_continuous_time(4, 6) = -kGravity;
22    A_continuous_time(5, 5) = -drag_coefficients_.at(2);
23    A_continuous_time(6, 6) = -1.0 / roll_time_constant_;
24    A_continuous_time(7, 7) = -1.0 / pitch_time_constant_;
25
26    model_A_ = (prediction_sampling_time_ * A_continuous_time).exp();
27
28    Eigen::MatrixXd integral_exp_A;
29    integral_exp_A = Eigen::MatrixXd::Zero(StateSize, StateSize);
30    const int count_integral_A = 100;
31
32    B_continuous_time(5, 2) = 1.0;
33    B_continuous_time(6, 0) = roll_gain_ / roll_time_constant_;
34    B_continuous_time(7, 1) = pitch_gain_ / pitch_time_constant_;
35    for (int i = 0; i < count_integral_A; i++)
36    {
37        integral_exp_A += (A_continuous_time *
      prediction_sampling_time_ * i / count_integral_A).exp()*
      prediction_sampling_time_ / count_integral_A;
38    }
39    model_B_ = integral_exp_A * B_continuous_time;
```

For computing B matrix in discrete time, it is necessary to compute the integral calculation presented in Section 2.5.1, so, being integral calculation in C++ quite

trivial, an incremental approach is adopted.
Objective function matrices are initialize as follows:

Listing 3.3: MPC optimization objective matrices definition.

```
1     Eigen::Vector3d q_position_(q1,q2,q3);
2     Eigen::Vector3d q_velocity_(q4,q5,q6);
3     Eigen::Vector2d q_attitude_(q7,q8);
4
5     Eigen::Vector3d r_command_(35.0, 35.0, 2.0);
6     Eigen::Vector3d r_delta_command_(0.3,0.3,0.0025);
7
8     Eigen::Matrix<double, StateSize, StateSize> Q;
9     Eigen::Matrix<double, StateSize, StateSize> Q_final;
10    Eigen::Matrix<double, InputSize, InputSize> R;
11    Eigen::Matrix<double, InputSize, InputSize> R_delta;
12
13    Q.setZero();
14    Q_final.setZero();
15    R.setZero();
16    R_delta.setZero();
17
18    Q.block(0, 0, 3, 3) = q_position_.asDiagonal();
19    Q.block(3, 3, 3, 3) = q_velocity_.asDiagonal();
20    Q.block(6, 6, 2, 2) = q_attitude_.asDiagonal();
21    R = r_command_.asDiagonal();
22    R_delta = r_delta_command_.asDiagonal();
23    Q_final = Q;
24    for (int i = 0; i < 1000; i++)
25    {
26        Eigen::MatrixXd temp = (model_B_.transpose() * Q_final *
      model_B_ + R);
27        Q_final = model_A_.transpose() * Q_final * model_A_ - (
      model_A_.transpose() * Q_final * model_B_) * temp.inverse()* (
      model_B_.transpose() * Q_final * model_A_) + Q;
28    }
```

All the matrices in the implemented code are defined as Eigen matrices of double quantities, but CVXGEN corresponding parameters must be passed as attributes of an object in vector form. For doing that, *cast* operations are performed:

Listing 3.4: Matrices cast operations.

```
1     Eigen::Map<Eigen::MatrixXd>(const_cast<double*>(params.Q),
      StateSize, StateSize) = Q;
2     Eigen::Map<Eigen::MatrixXd>(const_cast<double*>(params.Q_final),
      StateSize, StateSize) = Q_final;
3     Eigen::Map<Eigen::MatrixXd>(const_cast<double*>(params.R_delta),
      InputSize, InputSize) = R_delta * (1.0 / 0.01 * 0.01);
4
```

```
5     Eigen::Map<Eigen::MatrixXd>(const_cast<double*>(params.A),
      StateSize, StateSize) = model_A_;
6     Eigen::Map<Eigen::MatrixXd>(const_cast<double*>(params.B),
      StateSize, InputSize) = model_B_;
```

After having initialized the necessary parameters and having transformed the sampled states in ROS *Pose2D*, the samples evaluation can be performed.
Points sampled in the plan portion defined by the perpendicular to the initial pose yaw angle in the opposite direction to the initial pose orientation are marked with an infinite cost. The same happens for the final sampled poses with orientations offset more than sixty degrees with respect to the line joining initial and final poses. This samples selection is done to exclude non-promising samples and, then, avoiding to spend time computing unnecessary MPC problem solutions, considering that it is a-priori known that initial and final poses with opposite directions may cause bad trajectories in terms of cost. This is a sort of heuristic used to speed-up the algorithm.
For doing that, a function inside MPCOptimizationObjective class named *motionCostHeuristic2D* is implemented. This function is unprecedented named heuristic. "Heuristics are problem-solving methods that use shortcuts to produce good-enough solutions given a limited time frame or deadline. Heuristics are flexibility techniques for quick decisions, particularly when working with complex data", this is the definition given by James Chen on Investopedia website, so our function aims to speed-up the optimal solution computation, avoiding to evaluate non-optimal trajectories, but it indirectly increases the quality of the computed solution; in fact, *motionCostHeuristic2D* allows to sample and evaluate an higher number of possible "good trajectories" states, while neglecting those causing "bad trajectories". The result of this function is a Boolean variable sets as *true* whenever a goal pose does not satisfy predefined characteristics, and cost computation is immediately stopped returning an infinite cost; otherwise, if *false*, the MPC solver, the input limits, the initial position and the reference states are set and the optimization problem solution is computed.
It is decided not to limit the states, being the sampled child nodes quite close to parent nodes, so short motions are supposed to be predicted by the MPC.
Concerning the setting of the solver characteristics and of the limits, two functions are created:

Listing 3.5: setSolver function code.

```
1     set_defaults();
2     setup_indexing();
3     //settings.max_iters = 10;
4     //settings.eps = pow(10,-5);
5     //settings.resid_tol = pow(10,-3);
6     int dont_print_solver_output{0};
```

```
7     settings.verbose = dont_print_solver_output;
```

Listing 3.6: setLimits function code.

```
1     params.u_max[0] = roll_limit_;
2     params.u_max[1] = pitch_limit_;
3     params.u_max[2] = thrust_max_;
4
5     params.u_min[0] = - roll_limit_;
6     params.u_min[1] = - pitch_limit_;
7     params.u_min[2] = thrust_min_;
```

setSolver function performs a default configuration of the MPC, but the desired CVXGEN generated solver parameters could be customized. *settings.eps* is used for setting the solver duality gap for returning the MPC problem solution: solver will not declare a problem converged until the duality gap is known to be bounded by *eps*; *settings.resid_tol* command sets the residue tolerance value for returning the solver solution: solver will not declare a problem converged until the norm of the equality and inequality residuals are both less than *resid_tol*. *settings.max_iters* is used for setting the maximum number of iterations the MPC solver is allowed to run before returning a solution if *eps* and *resid_tol* are not reached. *settings.verbose* is used for making the solver able to output or not output information about each iteration, including residual norms, duality gap bounds and step sizes. By properly adjusting the setting parameters of the MPC solver, solution computation can be speeded-up considerably (further information available on CVXGEN website). For the purpose of this thesis, it is decided to keep these parameters at their default values. *setLimits* function defines the input values bounds. Limits are set according to [14].
Of particular interest is the reference trajectory $x_{ref,k}$ used by the MPC optimization with $k \in [0, H_P]$, defined by line-circumference interpolations, with an iterative process. Given an initial and a final pose, defined a line joining the two poses, interceptions line-circumferences centred in the initial pose are calculated, by increasing the circumference radius iteration after iteration, finding out newer reference positions farther to the initial pose. When the interpolated pose is near to the final pose, the successive references are calculated with interceptions circumferences centred in the start pose-final speed direction line, along the speed direction.
The reference speeds are set by multiplying the speed module, constant and a-priori defined, for sine and cosine of an angle that could be the joining line inclination angle, in case of reference points on the poses joining line, or the final pose orientation in the other case.
Setting as initial states the initial position and as initial speed the speed vector with predefined module and direction given by the initial pose orientation, the

optimization problem is solved iteratively. In fact, instead of saving the whole trajectory computed by MPC tool, only the first optimized state $x_1 \in \mathbb{R}^8$ is saved as newer initial position. The MPC approach needs the measured state quantities for iteratively adjusting the control input set and achieving the desired references; in the absence of real measurement from the UAV, being this phase totally theoretical and devoted to the motion cost computation for finding the optimal path, it is decided to use the simulated state of the previous MPC solving phase. The optimization problem is solved since the *Euclidean Distance* (for further details refer to Appendix D) between the first optimized state and the final pose is greater than a predefined threshold, since the *Euclidean Distance* between the first optimized states and the final one at the previous iteration is bigger than the distance at the current iteration (i.e. the distance to the goal is decreased), or since a predefined maximum number of iterations is not reached. The reference trajectory is recomputed at each iteration with the optimized state returned by the MPC logic.
An additional characteristic introduced by the motion cost computation function is the capability of stopping the trajectory optimization whenever an optimized state collides an obstacle. The trajectory planning is performed on a predefined map with fixed obstacles in the robotic space. The obstacle avoidance capability of the project may be guaranteed by the RRT* algorithm, that samples states only in the UAV free space, neglecting states sampled on or in the neighbourhood of obstacles. However, it may happen that the optimized trajectory joining two states in the free space collides with an obstacle.
Because of the already explained fact, a logic for stopping the trajectory is implemented in the *motionCost* function.

Listing 3.7: Obstacles detection function code.

```
1     initial_pose2D.x= vars.x_1[0];
2     initial_pose2D.y= vars.x_1[1];
3     initial_pose2D.theta=atan2(vars.x_1[4],vars.x_1[3]);
4
5     double temp_cost = getCost(initial_pose2D);
6     if (temp_cost < 0.0 || temp_cost >= 100)
7     {
8         return(Cost(infiniteCost()));
9      }
```

motionCost returns an infinite cost whenever a double value, returned by a function named *getCost*, does not lay between predefined bounds.

In the following, the *getCost* function is presented:

Listing 3.8: getCost function code.

```
1  double ompl::base::MPCOptimizationObjective::getCost(
   geometry_msgs::Pose2D pose) const
2  {
3    int cell_x = (int)(- pose.x + origin_x_) / resolution_;
4    int cell_y = (int)(- pose.y + origin_y_) / resolution_;
5    float temp_cost = (*data_)(cell_x, cell_y);
6
7    return (double)temp_cost;
8  }
```

Whenever the goal neighbourhood is reached, the optimization objective returns a cost. In order to test the influence of the cost function on the quality of the solution path, three different functions for computing the cost, named *computeCost1*, *computeCost2* and *computeCost3*, are implemented. The resulting cost computed by *motionCost* is the sum of the weighted costs returned by each single cost function. By changing these weights, the algorithm has different behaviours highlighted in Section 4.
Concerning *computeCost1*, this function is designed for returning the sum of the distances between each pair of poses of the predicted trajectory in meter. *computeCost2* is designed for returning the total rotation performed by the UAV in the predicted trajectory. *computeCost3* calculates a cost using the cost function of the optimization problem (3.1) with the control input and the state of the predicted trajectory. It is important to specify that this cost computation is done whenever a new state is sampled and attempted to be inserted in the graph.
In order to properly calculate these costs, at each iteration, the computed first input vector u_0, the first reference state $x_{ref,1}$ the first optimized state x_1 are saved into three matrices, one for each signal, with predefined dimensions equal to the input or state vectors dimensions and to the maximum number of the iteration that the MPC is allowed to be computed for a single couple of initial and final poses. These matrices are initialized and set null; then, they are filled with the computed values, iteration after iteration. In this way, the cost can be easily calculated performing a product between the optimized quantities and the optimization matrices.

Listing 3.9: Reference state, optimized state and input matrices initialization and filling, with *it* the number of iterations of the MPC logic.

```
1  Eigen::Matrix<double, 200, StateSize> matrix_x_temp;
2  Eigen::Matrix<double, 200, StateSize> matrix_x_ref_temp;
3  Eigen::Matrix<double, 200, StateSize> matrix_u_temp;
4  matrix_x_temp.setZero();
5  matrix_x_ref_temp.setZero();
6  matrix_u_temp.setZero();
7
```

```
8       for(unsigned j=0;j<StateSize;j++)
9       {
10          matrix_x_ref_temp(it,j)=params.x_ss_1[j];
11          matrix_x_temp(it,j)=vars.x_1[j];
12          matrix_u_temp(it,j)=vars.u_0[j];
13      }
```

Listing 3.10: computeCost1 function code.

```
1       double ompl::base::MPCOptimizationObjective::computeCost1(Eigen::
        MatrixXd matrix_x,Eigen::MatrixXd matrix_x_ref,Eigen::MatrixXd
        matrix_u, int it) const
2       {
3           double c1=0.0;
4           geometry_msgs::Pose2D pose1,pose2;
5
6           for (size_t i = 1; i < it; i++) {
7               pose1.x=matrix_x(i-1,0);
8               pose1.y=matrix_x(i-1,1);
9               pose2.x=matrix_x(i,0);
10              pose2.y=matrix_x(i,1);
11              c1+=computeDistance2D(pose1,pose2);
12           }
13          return(c1);
14      }
```

Listing 3.11: computeCost2 function code.

```
1       double ompl::base::MPCOptimizationObjective::computeCost2(Eigen::
        MatrixXd matrix_x,Eigen::MatrixXd matrix_x_ref,Eigen::MatrixXd
        matrix_u, int it) const
2       {
3           double c2=0.0,angle_diff=0.0;
4           for (size_t i = 1; i < it; i++) {
5
6
7               angle_diff=std::abs(atan2(matrix_x(i,4),matrix_x(i,3))-
        atan2(matrix_x(i-1,4),matrix_x(i-1,3)));
8               if(angle_diff>M_PI)
9               angle_diff=(2*M_PI-angle_diff);
10              c2+=180/M_PI*angle_diff;
11          }
12          return(c2);
13      }
```

As already explained, the *computeCost3* cost computation is done by performing a summation of matrices products.
This operation can be described in C++ language by nested *for* loops performing

matrices cells product. These products are added to an incremental double variable, representing the cost.

Listing 3.12: computeCost3 function code fragment.

```
1     double cost=0.0;
2     for(unsigned i=1;i<it;i++)
3     {
4         for (unsigned j=0;j<InputSize;j++)
5         {
6             cost=cost+pow((matrix_u(i,j)-matrix_u(i-1,j)),2)*R_delta(
      j,j);
7         }
8     }
9
10    for(unsigned i=0;i<it;i++)
11    {
12        for (unsigned j=0;j<StateSize;j++)
13        {
14            cost=cost+pow((matrix_x(i,j)-matrix_x_ref(i,j)),2)*Q(j,j)
      ;
15        }
16    }
17    for (unsigned j=0;j<StateSize;j++)
18    {
19      cost=cost+pow((matrix_x(it-1,j)-matrix_x_ref(it-1,j)),2)*
      Q_final(j,j);
20    }
21    return(cost);
```

This cost computation process is performed many times for each sampled state of RRT*, so it is logical to expect that the cost computation is continuously performed until the solve time. If, after this period of time, a path joining the desired start and goal poses in the two-dimensional plan is found, it is the one that minimizes the "cost-to-go" function for reaching the desired pose in the space, according to the motion cost definition. Being the MPC optimization computationally expensive, the algorithm computes few states in the defined search time, so the path will not be the global optimal one, while the one with the lowest cost between the evaluated paths for reaching the goal. This limit is always present in RRT* logic; in fact, the global optimal solution is obtained only when search time approaches infinite. The limitation is emphasized in the deployed algorithm because of the time needed for evaluating a single trajectory through MPC approach.
After the computation of the path using RRT*, the returned path consists in a sequence of motions defined by the MPC. However, these motions are not perfectly aligned due to uncertainties in the MPC optimization. In order to have a continuous trajectory between the start and the goal pose, the MPC optimization is used to define the whole trajectory.

Differently of what is done for the motion cost computation, the trajectory computation function, named *trajectoryGenerator*, introduces a little modification with respect to the previous function. These function receives as inputs two states that are successively transformed into ROS *Pose2D* data for performing the desired operations, and a vector of ROS *Pose* elements. The MPC logic is initialized in the same way it is done for the *motionCost* method; the only difference is that, at each iteration successive the first one, the initial pose to be connected to the corresponding goal pose is the one calculated by the MPC logic at the previous iteration. In this way, possible orientation errors as well as position errors are corrected by the MPC tool, returning a continuous final path. *trajectoryGenerator* and *motionCost* are structured in the same way; for a single couple of poses, MPC problem is solved since the *Euclidean distance* between the first optimized state and the goal pose is greater than a predefined quantity or since a predefined maximum number of iterations is not reached.
However, at the end of each optimization phase, the first optimized pose $x_1 \in \mathbb{R}^8$ is saved into the vector of poses passed as input. Trajectory generation is performed after that the lowest cost path is obtained. The sets of states passed by the planner are assumed to be trajectory feasible, being the unfeasible nodes discarded in the cost computation phase. After that each of the couples is passed to *trajectoryGenerator*, the optimized trajectory and the whole tree constructed by this algorithm are published on the corresponding Topics and visualized with RViz tool. An example of the obtained trajectory is presented in Figure 3.3

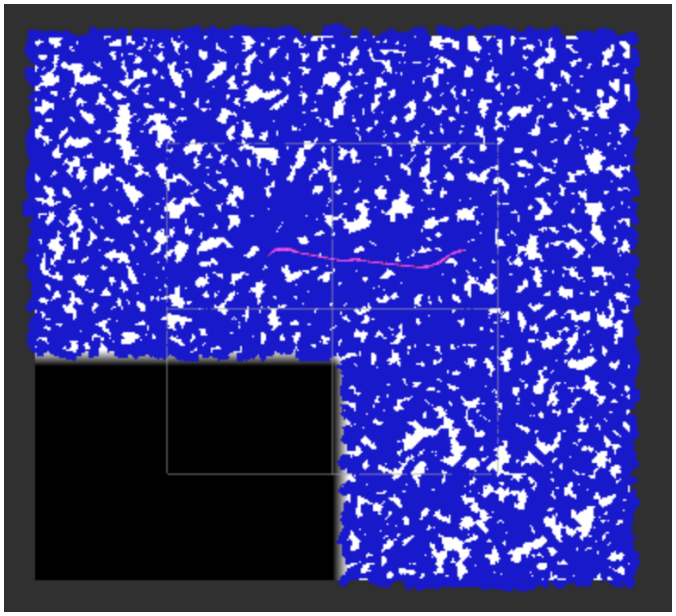


Figure 3.3: Example of an algorithm solution.

The search tree is marked with a blue colour, while the optimized trajectory is represented by a set of pink arrows. As clearly shown in Figure 3.3, the algorithm samples states only in the robotic free space. The constructed tree never enters the obstacle region, highlighted with the black colour, and it never exceeds the map outline.
Another important aspect to be highlighted is the fact that the computed trajectory is not straight. This is due to the random orientation of the sampled states. The algorithm computes the cost for reaching a state that could have any orientation in the space. It is important to remember that wrong samples in terms of orientation are discarded, but those with an acceptable orientation are always taken into consideration, even if they are not optimal for reaching the goal pose of the motion query. These states are marked with an higher cost because of their distance from the desired reference, designed as a straight line, and for their longer distance and bigger overall rotation. By increasing the number of samples, the probability of sampling lower cost states is higher, but it is not sure that the solution obtained in the given search time is perfectly straight. It could happen that an intermediate state in the graph has an high cost, so its trajectory is not "optimal", but in the given solve time, no other solution with lower cost is found.
It must not be forgotten that, even if the final graph contains high cost nodes, the resulting trajectory is supposed to be kinematically and dynamically "optimal" for the UAV, being computed by an MPC that exploits the vehicle dynamic model with constraints in terms of UAV input signals. Being more precise, the resulting trajectory is always the optimal one between any couple of states in the resulting graph.
For all the mentioned reasons, timing is a crucial topic for this algorithm. The quality of the resulting trajectory strongly depends on the sampled states orientations.
With the aim of improving the quality of the planner, some simplifications are done exploiting default OMPL functions. In particular, *reduceVertices* function is applied after the optimal path computation and before the MPC trajectory generation. This function attempts to remove vertices from the passed path while keeping it valid by creating connection between non-consecutive way-points for "short-cutting" the path when possible. Other simplifications are offered by OMPL, but it is important to say that the simplifications compromise the quality of the algorithm; some nodes, that are optimally oriented, may be deleted for achieving a simpler resulting path, but not optimal in terms of trajectory regularity. Anyway, simulation results are reported in the Section 4.

3.2 ROS-PX4 Interface

This work is tested with a realistic simulation that allows to show the behaviour of the UAS in a realistic environment. In order to do that, PX4 is used.
PX4 is an open source flight control software for drones and other unmanned vehicles [1], offering a flexible set of tools for drone developers to share technologies to create tailored solutions for drone applications. PX4 provides a standard to deliver drone hardware support and software stack and supports Software In the Loop (SITL) simulation with flight stack running on computer.
SITL allows to test an implemented code directly into the mathematical simulation that contains the models of the Physical System. Thus and so, it is possible to test the current code even without the target physical hardware, performing faster simulation in terms of time.

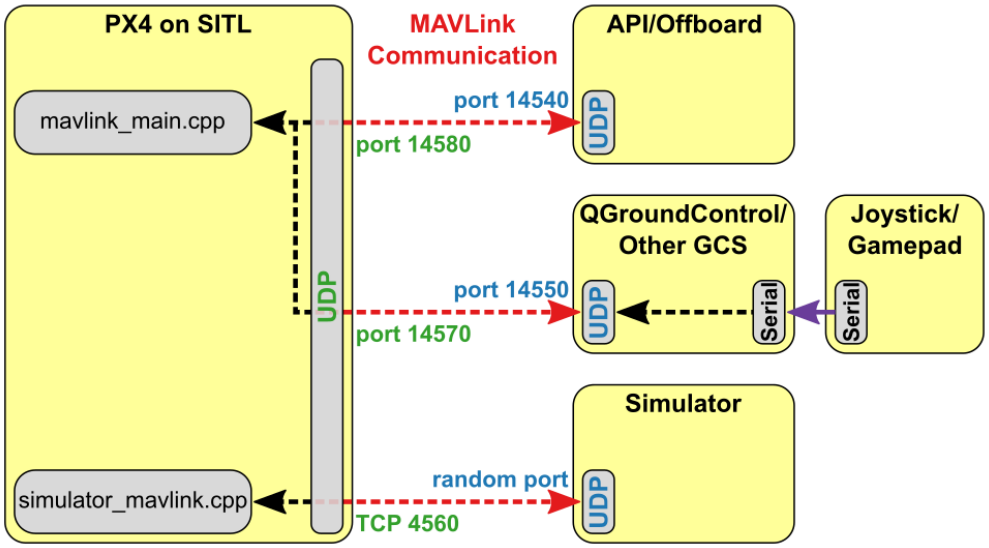


Figure 3.4: PX4 SITL Simulation Environment.[2]

It is decided to use PX4 for the final simulation because of its capability of interfacing with ROS. In fact, PX4 communicates with external frameworks using the MAVLink communication protocol [22]. In order to enable the communication between PX4 and ROS, the ROS community developed the *mavros* package, that converts ROS messages into MAVLink messages, and the opposite.
In particular, as showed in Figure 3.4, UDP port 14540 allows the communication

[1]Definition courtesy from PX4 Autopilot website.

[2]Image courtesy from PX4 Autopilot website.

with ROS. Exploiting the *mavros* Topics and the ROS services offered by MAVLink, it is possible to publish missions described by a set of points in the space on PX4 to perform realistic simulations. In order to publish trajectory states with a proper syntax on the *mavros* corresponding Topic, a ROS Node is implemented.
This node, named *drone_node* contains a ROS Service and a ROS Subscriber. The subscriber subscribes to the optimized trajectory Topic and a callback to that Topic is implemented for making the node able to react to any trajectory publication.
It is decided to publish data in the form of UAV *Waypoints list*. MAVLink exploits a *mavros/mission/waypoints* Topic for receiving data of this type. Each element of this list is a *Waypoints* message containing attributes for position and other parameters to be passed to PX4 for configuring the autopilot (for further information referring to Appendix E). Callback function is intended for translating ROS *Pose* in *Waypoints*. *Waypoints* are set for performing UAV arming, takeoff, trajectory following and landing, depending on the passed pose set.
Mavlink *Waypoints* interface needs global GPS coordinates for working properly, so local positions in meter are translated into GPS data. The function that performs this operation translates Cartesian coordinates in GPS ones referring to a fixed origin position.

Chapter 4

Simulation and Testing

This section is devoted to the presentation of the environment used for simulating and testing the proposed trajectory planning algorithm.
As mentioned in Section 2.1, this thesis is based on ROS framework. The whole developed framework consists in ROS Nodes, ROS Services and ROS Topics interacting each other exploiting the decentralised ROS structure for having an algorithm that can be realistically tested and, eventually, easily implemented on existing UAVs.
Trajectory planning algorithm can be launched through *roslaunch* command from Ubuntu terminal. Automatically, RViz (ROS Visualization Tool) is executed and start and goal poses can be chosen on the map and, then, they are sent as messages on a ROS Topic to the Node that provides the trajectory planner. The implemented algorithm is able to read those poses, when available, directly from their Topic and start the trajectory planning.
After the search time, RRT* algorithm returns the optimal path in the constructed graph, successively passed to the MPC tool for computing the optimized trajectory, and the grown tree. RRT* tree and MPC optimized trajectory are published on their ROS Topics for being visualized on RViz. ROS Visualization Tool allows to analyze the resulting planned path on the desired map. RRT* algorithm is configured for reading the map size and sampling the free space defined by its dimensions.
For the timing limitations introduced by the MPC logic, the proposed algorithm is designed for working locally, so small map tests are performed. Nevertheless, this algorithm works quite well even with large maps, but the quality of the planner reduces with the growth of the map size.

4.1 Simulation Hardware

The proposed trajectory planning algorithm is tested on a PC with the following features:

Operating System: Ubuntu 18.04.5 LTS;

CPU: Intel® Core™ i5-4670K CPU @ 3.40GHz × 4;

GPU: GeForce GTX 950;

RAM: 8GB;

ROS Version: Melodic 1.14.9.

4.2 Parameter Optimization

This section is devoted to discover the best set of parameters defined in the algorithm in terms of solution quality and planning efficiency.
The influence of the following parameters is analyzed:

- Cost Function weights;
- UAV Speed Module;
- Solve Time.

For the whole set of tests the trajectory planner always optimizes the motion between the same start and goal poses in the two-dimensional space. The poses are set for forcing the planner to discover a path trying to avoid a fixed obstacle; so, it is decided to use a map of size $20m \times 20m$, showed in Figure 4.1, with a L shape obstacle occupying the bottom-left and central part of the map in order to simulate an harsh local environment. The planner, in fact, has to avoid the obstacle while trying to pass as close as possible to it.
Each subsection aims to analyze the influence of a single parameter for finding out its value that maximizes the quality of the planner. Initially, the test parameters are "randomly" set, but at the end of each subsection, a good value is found and the following tests are performed with the previously found set of parameters. In this way, at the end of this phase, the optimal configuration is obtained.
Moreover, the influence of the path simplification before the final MPC optimization is shown. This comparison is done only for the best possible set of already analyzed values (i.e. only the candidate parameter sets). Initial parameters are shown in Table 4.1.
It is important to point out that *Cost1* is the weight cost relative to the trajectory

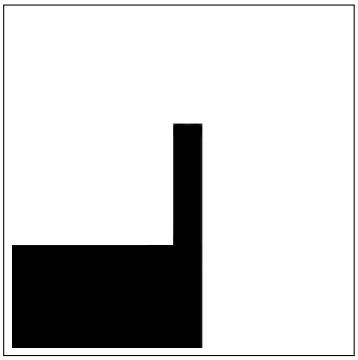


Figure 4.1: Algorithm parameters test map.

length in meters, *Cost2* is the weight cost of the total rotation in degrees necessary for performing the motion, while *Cost3* is the weight of the cost obtained computing the optimization function used by the MPC, solved with the trajectory set of state, reference and input.
Before proceeding with testing, it is necessary to introduce two parameters that have not been taken into consideration yet, but that are really important for obtaining good planning results. The first one is the *Planner Range*; it is the radius of the circle centered in the states in which RRT* connects newer states; whenever a sample is randomly sampled at a distance greater than the *Planner Range*, it is substituted with a newer sample along the line joining the initial sample, but *Planner Range* far from its nearest node in the graph. This radius decreases with the increasing of the sampled states, as properly explained by Frazzoli and Karaman in [17], to converge to the sub-optimal path solution, so, for this reason, it is fixed equal to five meters, but decreases while the planning proceeds. Another important parameter to introduce is the *Planner Goal Bias*, that represents the probability of sampling the goal state in the whole *Solve Time*. Its value is fixed equal to 5%, meaning the goal state is sampled five out of one hundred times during the graph construction phase; higher value are counterproductive in terms of planner quality.
For the way the proposed algorithm is designed, RRT* *Solve Time* does not represent the total time it searches for a solution. In fact, for analyzing the evolution of the graph constructed by the planner, the solving phase is divided in ten phases with the same duration equal to the *Solve Time*. At the end of each phase, the cost of the path up to that time is printed for seeing it diminishes cycle after cycle.
Because of the probabilistic nature of RRT* algorithm, each test is performed five times and the different path costs and path lengths are reported on a single label, while only the most significant trajectory out of five is figured out (i.e. the trajectory that best represents the limitations or advantages introduced by the

analyzed set of parameters).

4.2.1 UAV Speed Module

In this subsection, the influence of the *UAV Speed Module* on the final solution is analyzed. In order to recap the concept, the UAV trajectory is designed for being done at constant speed equal to *UAV Speed Module* value. Depending on the UAV orientation, the reference speed contributions in the space are changed. Because of the input constraints, *UAV Speed Module* represents an important parameter for the planner.

Test 1

The first test is performed with the parameters set labelled at Table 4.1.

Parameter Set	
Cost1	1.0
Cost2	1.0
Cost3	0.1
Prediction Sampling Time	0.1s
Planner Range	5m
Solve Time	6s
Total Solve Time	60s
UAV Speed Module	3.0m/s

Table 4.1: Initial parameters set.

This value is too high for the imposed input constraints, so bad trajectories are expected.

Simulation Results		
Simulation Number	Path Cost	Path Length
First Simulation	1287.763	46.380m
Second Simulation	1250.149	42.837m
Third Simulation	1276.167	42.800m
Fourth Simulation	1155.827	54.519m
Fifth Simulation	1307.899	40.382m
Average	1255.561	45.384m

Table 4.2: UAV Speed Module test 1: simulation results.

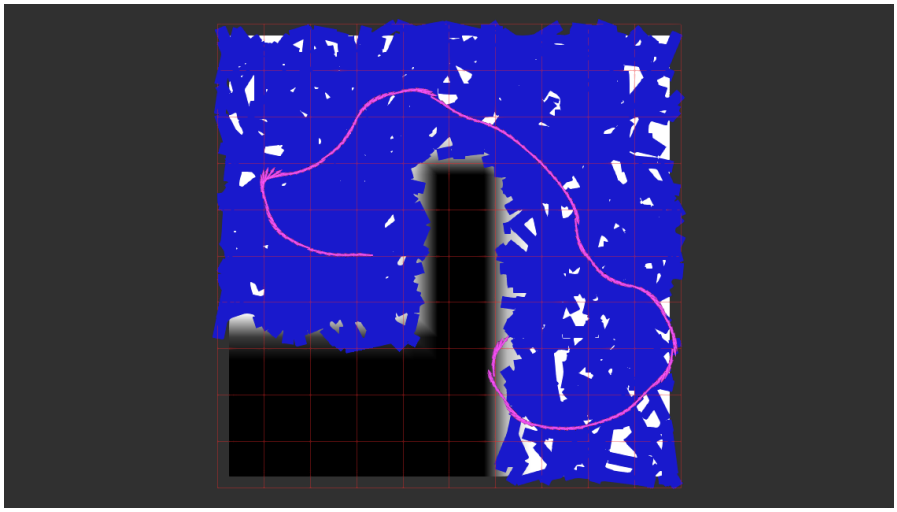


Figure 4.2: UAV Speed Module test 1: third simulation trajectory.

As clearly shown in Figure 4.2, this configuration of the planner is not effective. The *UAV Speed Module*, assumed constant for the whole motion, is too high with respect to the imposed input constraints: the goal pose orientation is never obtained and long trajectories are planned. Adopting a set of constraints that is optimal for the used trajectory tracking logic on which the designed MPC tool is based, it is decided to decrease the speed of the UAV during the planning instead of changing the input constraints. Another possible approach is adding state limits directly in the MPC optimization problem, but this addiction increases the complexity of the problem, slowing down the solution computation and the overall algorithm performances.

Test 2

For the previously explained reasons, this test is performed by decreasing the *UAV Speed Module* from 3.0m/s to 2.0m/s.

The quality of the planner increases a lot with the new *UAV Speed Module*; the costs of the new trajectories are less than half of those obtained from the previous parameter set, and as a consequence, the total lengths of the computed paths are reduced. Decreasing again *UAV Speed Module* is counterproductive because the planner has to guarantee good motion performances, so lower values of speed are not taken into consideration.
The trajectories computed by the proposed algorithm in this test set are quite good; however, they are quite far from the optimal; the length of the path is high and several useless curves are performed.

Parameter Set	
Cost1	1.0
Cost2	1.0
Cost3	0.1
Prediction Sampling Time	0.1s
Planner Range	5m
Solve Time	6s
Total Solve Time	60s
UAV Speed Module	2.0m/s

Table 4.3: UAV Speed Module test 2: parameters.

Simulation Results		
Simulation Number	Path Cost	Path Length
First Simulation	570.945	24.079m
Second Simulation	799.445	28.477m
Third Simulation	565.249	29.308m
Fourth Simulation	690.262	24.998m
Fifth Simulation	572.070	25.230m
Average	639.594	26.418m

Table 4.4: UAV Speed Module test 2: simulation results.

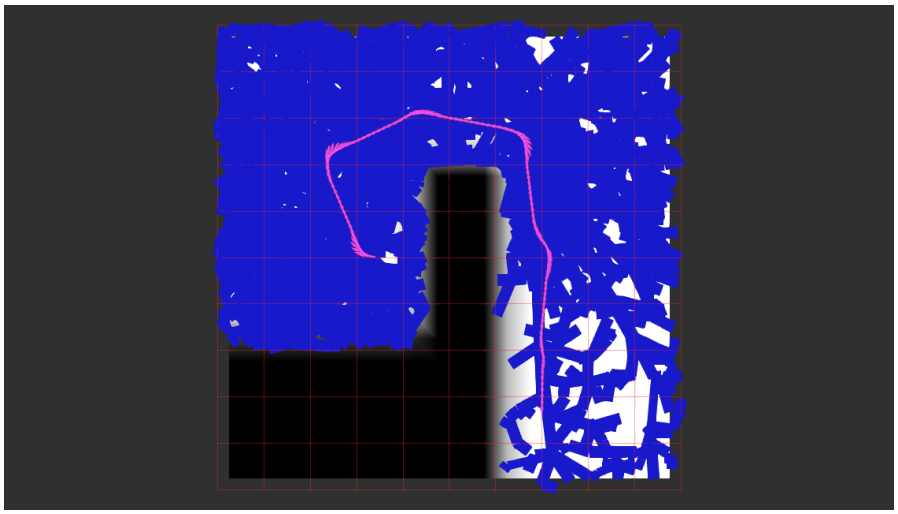


Figure 4.3: UAV Speed Module test 2: fifth simulation trajectory.

4.2.2 RRT* Cost Function Weights

This test section aims to find the best combination of MPC cost function weights. The trajectory planner is designed for minimizing the length of the final solution while diminishing the rotation of the drone, taking into consideration input variations and state errors with respect to the reference trajectory.
For performing straight trajectories, UAV model needs a constant set of input, so null input variation. In the same way, if UAV is correctly aligned with the reference trajectory, its contribution to the cost is negligible. However, it could happen that a low cost trajectory with respect to input variation and state error is long. So, it is necessary to take into consideration the length of the path properly weighted with respect to the MPC optimization function.

Test 1

For the previously explained reasons, taking into consideration the rotation of the UAV seems useless. Indeed, the presence of this cost contribution is ineffective for the optimal solution computation. Sharp corners are marked with high cost in terms of MPC function because of the smooth reaction due to input limitations that causes "slow" maneuvers for riding the corner; so, higher the number of rotations in the trajectory, higher the cost in terms of MPC function due to state errors with respect to the reference as well as the input variations.
So, the first test is done with a null weight of the cost function that takes into consideration the UAV rotation during the motion.

Parameters Set	
Cost1	1.0
Cost2	0.0
Cost3	0.1
Prediction Sampling time	0.1s
Planner Range	5m
Solve Time	6s
Total Solve Time	60s
UAV Speed Module	2.0m/s

Table 4.5: Cost function weights test 1: parameters.

The performances of the algorithm following the first modification of the weights look quite interesting, as clearly shown in Figure 4.4. The generated trajectories are smooth, but another limit is met: being the MPC relative cost dominant with respect to the length cost, it could happen, and it is clearly observable in Table 4.6,

Simulation Results		
Simulation Number	Path Cost	Path length
First Simulation	281.131	28.495m
Second Simulation	277.422	20.413m
Third Simulation	336.692	22.967m
Fourth Simulation	275.649	26.753m
Fifth Simulation	327.581	25.707m
Average	299.695	24.867m

Table 4.6: Cost function weights test 1: simulation results.

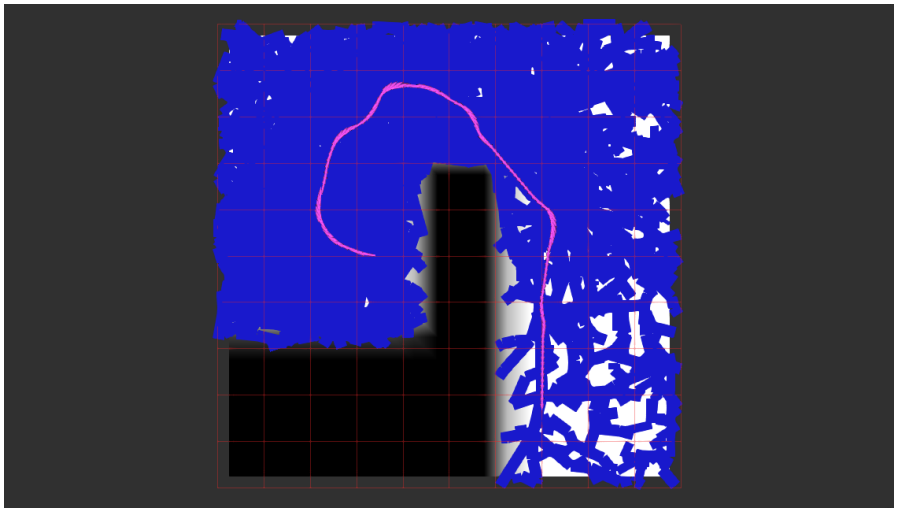


Figure 4.4: Cost function weights test 1: fourth simulation trajectory.

that the lowest cost trajectory is the longest one, being the quality of the motion the predominant parameter in the optimization problem.

Test 2

In this test, the best relative weights between the MPC optimization function and the path length in the cost computation are discussed.
For giving bigger importance to the path length, its cost is augmented of ten times with respect to the previous test sets. Having the weighted MPC cost a dimension in the order of hundreds, multiplying by 10 the path makes the two costs comparable for the final cost value. Higher values in terms of cost are expected, but they do not have to be compared with the ones of the previous test sets: another cost function is introduced whenever the relative weights in the cost function change.
The new test parameters set is showed in Table 4.7

Parameters Set	
Cost1	10.0
Cost2	0.0
Cost3	0.1
Prediction Sampling time	0.1s
Planner Range	5m
Solve Time	6s
Total Solve Time	60s
UAV Speed Module	2.0m/s

Table 4.7: Cost function weights test 2: parameters.

Simulation Results		
Simulation Number	Path Cost	Path length
First Simulation	543.601	23.201m
Second Simulation	509.132	22.436m
Third Simulation	528.185	23.662m
Fourth Simulation	422.044	23.590m
Fifth Simulation	454.620	22.418m
Average	491.516	23.061m

Table 4.8: Cost function weights test 2: simulation results.

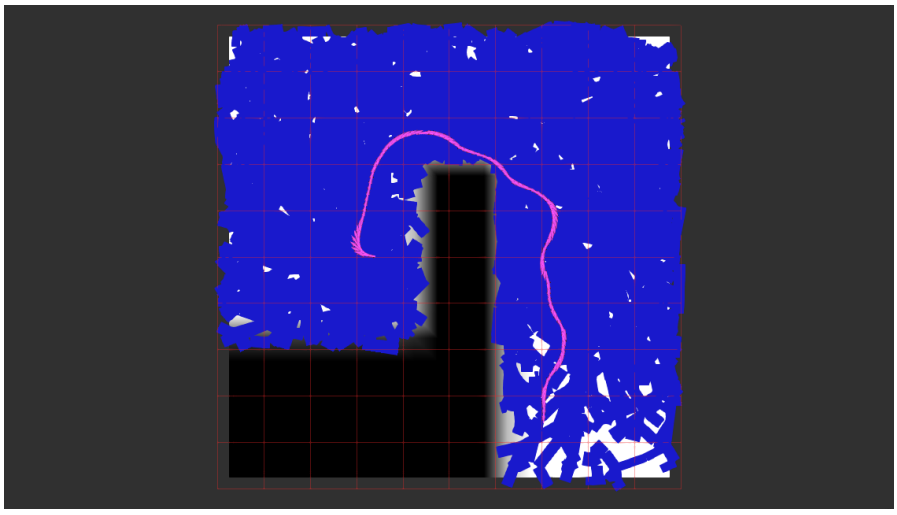


Figure 4.5: Cost function weights test 2: second simulation trajectory.

At the end of this test it is clear that, with the current parameter set, the algorithm discovers shorter good shape trajectories. The aspect that could be

improved is the quality of the shape; however, it is important to highlight that, increasing the *Solve Time*, the solution converges to the expected shortest path. So, the current weights are kept, and the quality of the solution is analyzed with shorter and longer *Solve Time*.

4.2.3 Solve Time

Discovered a good trade-off between the different cost function weights, the timing performances of the algorithm are tested in this section. It is clear that, with longer *Solve Time*, shorter and less expensive paths are expected, because of the convergence of RRT* to the optimal path with an higher number of samples. On the other side, the behaviour of the algorithm with short *Solve Time* is unknown.

Test 1

This test set aims to demonstrate that, doubling the RRT* *Solve Time*, the trajectories discovered by the planning algorithm converges to the optimal in terms of cost function. The cost weights remain unchanged with respect to the previous test set, while the only parameter that is changed is the *Solve Time*, that becomes of 12.0s.

Initial Parameters	
Cost1	10.0
Cost2	0.0
Cost3	0.1
Prediction Sampling time	0.1s
Planner Range	5.0m
Solve Time	12.0s
Total Solve Time	120.0s
UAV Speed Module	2.0m/s

Table 4.9: Solve Time test 1: parameters.

This test set does not present unexpected results. With higher *Solve Time* value, the probability of obtaining an optimal solution is higher. For the whole test set, in fact, low cost and short trajectories are planned, showing the convergence of the resulting path to the optimal one with the time increasing.

Simulation Results		
Simulation Number	Path Cost	Path length
First Simulation	503.885	21.204m
Second Simulation	470.008	23.184m
Third Simulation	421.392	22.485m
Fourth Simulation	449.071	21.206m
Fifth Simulation	433.744	22.388m
Average	455.62	22.093m

Table 4.10: Solve Time test 1: simulation results.

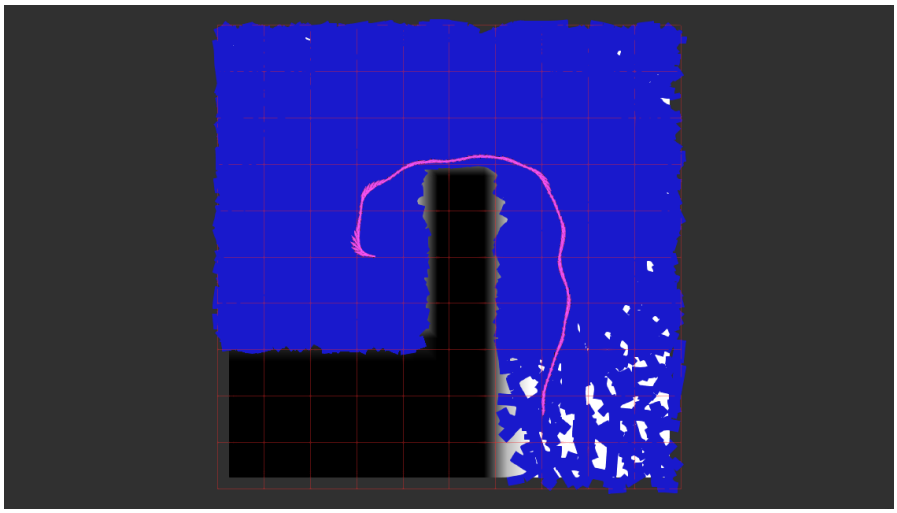


Figure 4.6: Solve Time test 1: fourth simulation trajectory.

Test 2

In order to test the effectiveness of the proposed algorithm, the second test of this section shows the trajectory-find capability of the implemented logic in small amount of time. The algorithm configuration parameters of this test are reported in Table 4.11.

By inspecting the results labelled in Table 4.12, it could be seen that the used set of cost function weights allows to plan good trajectories even with few *Solve Time*. The overall length and cost of the motion paths are similar to the ones obtained in the previous test set. This fact demonstrates that, with the proper trade-off of the algorithm parameters, good solutions can be found even in small amount of time. Moreover, until now, the effects of the path simplifier have not been taken into account, so it is interesting to understand if this kind of simplification could be useful in terms of final solution.

Initial Parameters	
Cost1	10.0
Cost2	0.0
Cost3	0.1
Prediction Sampling time	0.1s
Planner Range	5m
Solve Time	1s
Total Solve Time	10s
UAV Speed Module	2.0m/s

Table 4.11: Solve Time test 2: parameters.

Simulation Results		
Simulation Number	Path Cost	Path length
First Simulation	433.859	22.936m
Second Simulation	421.449	23.607m
Third Simulation	635.237	27.800m
Fourth Simulation	570.987	24.222m
Fifth Simulation	452.389	25.166m
Average	502.784	24.746m

Table 4.12: Solve Time test 2: simulation results.

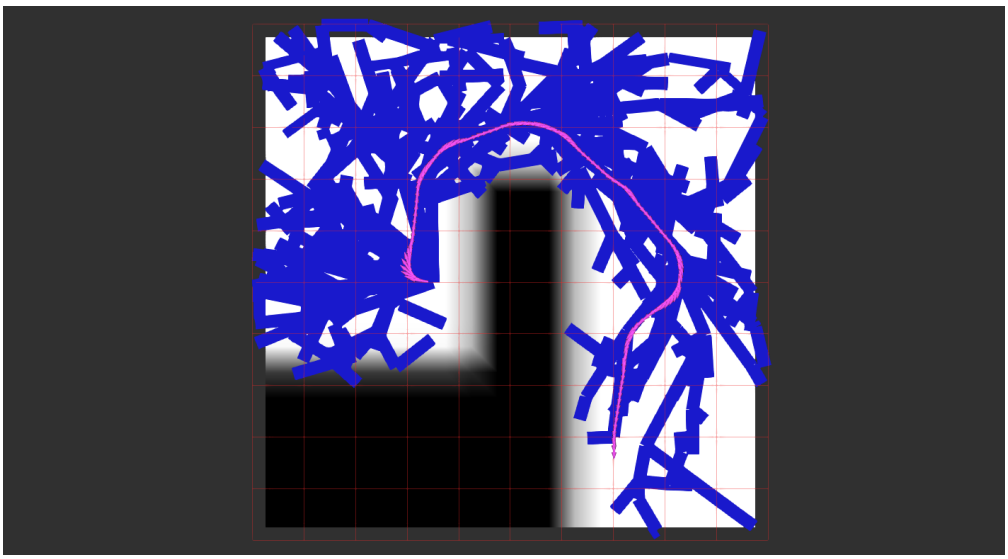


Figure 4.7: Solve Time test 2: fifth simulation trajectory.

Test 3 with Path Simplifier

This subsection is devoted to the testing of the default OMPL path simplifier applied to the optimal trajectory in the same conditions analyzed in the previous test set. This test aims to demonstrate the effectiveness of the implemented trajectory planning strategy in short time with a path simplification. It is important to anticipate that the path simplifier does not affect the cost computation, being the simplification a post-processing action applied to an already found path. On the contrary, this simplification affects the path length that is calculated after the path simplification through the MPC logic. The used set of data is labelled at Table 4.11; no changes are done with respect to the previous test parameters set.

Simulation Results		
Simulation Number	Path Cost	Path length
First Simulation	430.465	22.973m
Second Simulation	528.295	26.021m
Third Simulation	562.702	22.536m
Fourth Simulation	535.964	22.402m
Fifth Simulation	580.709	22.154m
Average	527.627	23.217m

Table 4.13: Solve Time test 3: simulation results with path simplifier.

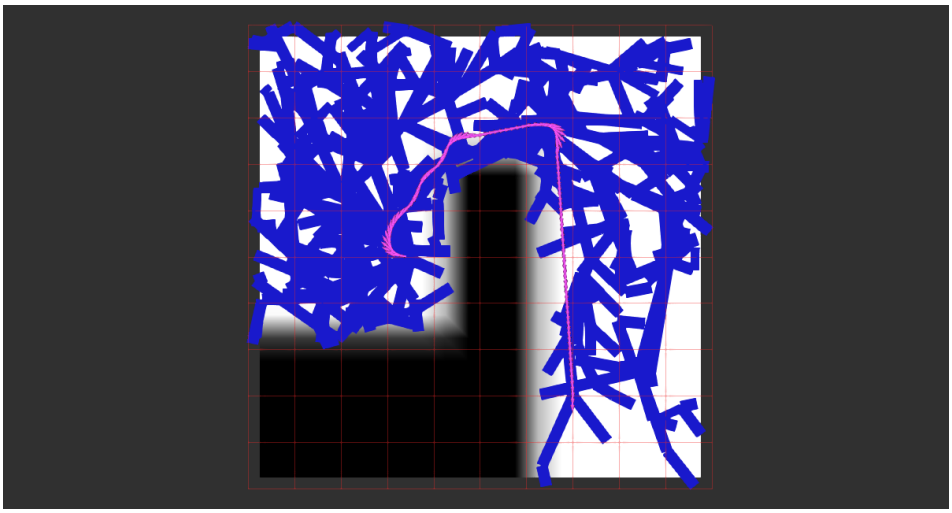


Figure 4.8: Solve Time test 3: third simulation trajectory.

As clearly shown in Table 4.13, the main advantage of applying a path simplification in the presence of few samples in the search space is the fact that shorter

paths can be obtained, being longer trajectories "cut" by the path simplifier. In fact, it could happen that, with few samples, the algorithm returns a long trajectory; the probabilistic nature of the algorithm does not ensure to discover always a smooth and short path in a small search time. Removing non-optimum vertices in terms of trajectory length solves this problem. Moreover, RRT* discovered path is passed to the MPC logic for computing the final trajectory, ensuring a good shape final solution. However, even if this solution looks pretty good for solving local motion queries, it is decided to plan for one minute, and, then, applying the path simplifier.

4.2.4 Final Configuration with Path Simplifier

This section is devoted to test the parameter set that is the candidate final set for the proposed logic. Parameters values are shown in Table 4.14.

Parameters Set	
Cost1	10.0
Cost2	0.0
Cost3	0.1
Prediction Sampling time	0.1s
Planner Range	5m
Solve Time	6s
Total Solve Time	60s
UAV Speed Module	2.0m/s

Table 4.14: Final configuration test: parameters.

Simulation Results		
Simulation Number	Path Cost	Path length
First Simulation	534.707	21.191m
Second Simulation	544.864	22.346m
Third Simulation	543.272	20.680m
Fourth Simulation	470.754	22.212m
Fifth Simulation	505.283	20.935m
Average	519.776	21.473m

Table 4.15: Final configuration test: simulation results.

The application of the path simplifier to a good discovered path does not produce excessive changes in terms of quality. Nevertheless, applying a path simplification is helpful for reducing the length of the final trajectory when the algorithm discovers relative high cost solutions. Because of the probabilistic nature of the algorithm,

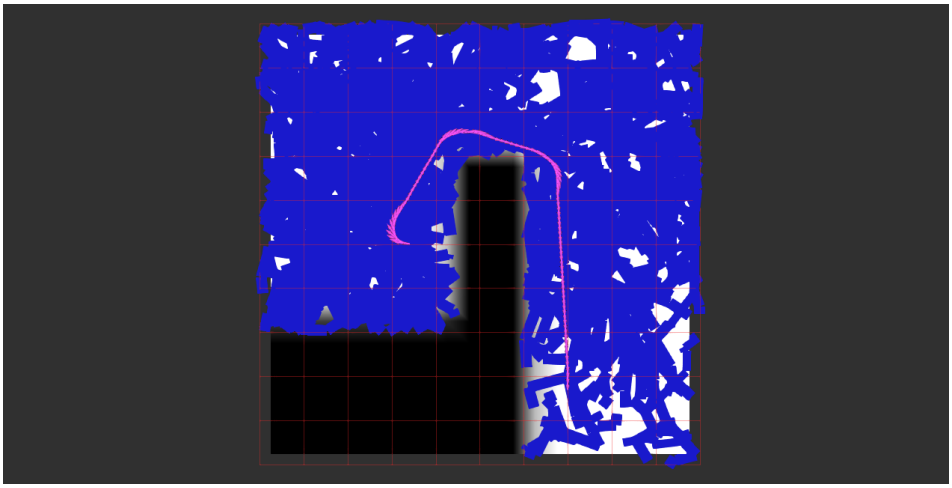


Figure 4.9: Final configuration test: fifth simulation trajectory.

the final solution strongly depends on the quality of the sampled states, so, in order to force the final trajectory to be straight, the use of path simplifier is appreciable in this sense.
Summarising, after this first testing phase, the overall best solution with a *Solve Time* of sixty seconds, where the rotation cost is neglected and the MPC cost and the length cost are comparable in terms of relative values, is the one analyzed in this last section. In addiction, for the already explained reasons, the path simplification is kept valid for planning the final trajectory.

4.3 Test in Different Maps

After having discovered the best algorithm configuration, labelled at Table 4.14, it is interesting to test it in different maps in order to understand if the set of parameters tagged as the best one is really effective in any kind of environment.
For the following tests, the graphical dimension of the constructed graph is decreased with respect to the previous test sets for better appreciating the proposed figures. All the maps used in this section have dimension of $20m \times 20m$.

4.3.1 Narrow and Constrained Environment, First Map

In this section, the behaviour of the planner in a narrow and constrained environment is tested. This simulation is quite trivial because of the fact that the algorithm is forced to sample states with proper orientations in a small passage; in fact, wrong sampled poses angles cause unfeasible trajectories. The map used in this test is reported at Figure 4.10.

The analyses performed in this section are quite different from those of the previous test sets.
Here, the convergence of the cost function to the optimal is showed with a chart where each of its points represents the average path cost over the five tests in a precise instant of time. This chart is important for figuring out the diminishing of the path cost over time, proving that the algorithm tries to minimize the defined cost function. Moreover, another chart is introduced; it shows the average over the five tests of the number of samples in the constructed graph in a precise time instant in order to show that the addiction of vertices to the graph constructed by RRT* slows down while time passing. So, each point in the charts represents the average over the five tests of the costs and of the number of samples in that precise time instant.
It could happen that, being the analyzed map quite big, the saturation of the vertices number is not reached in the imposed *Solve Time*.

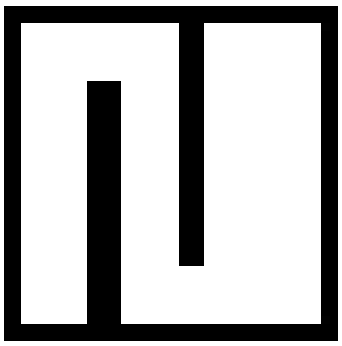
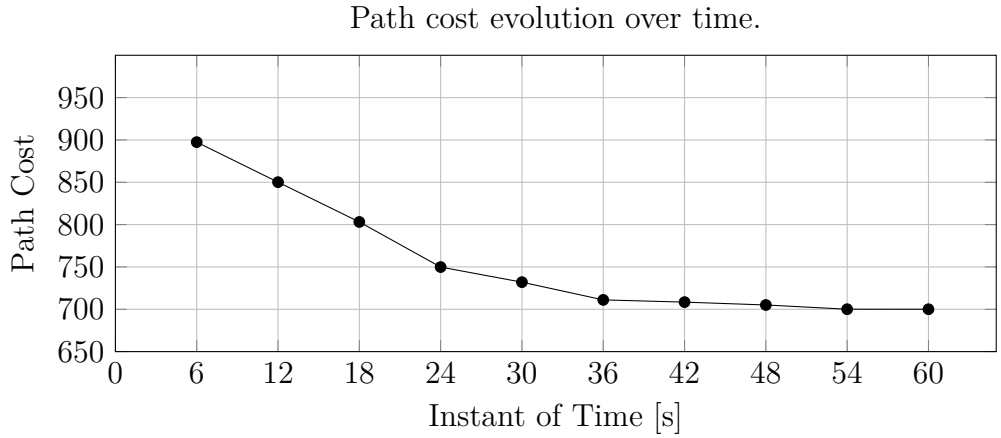
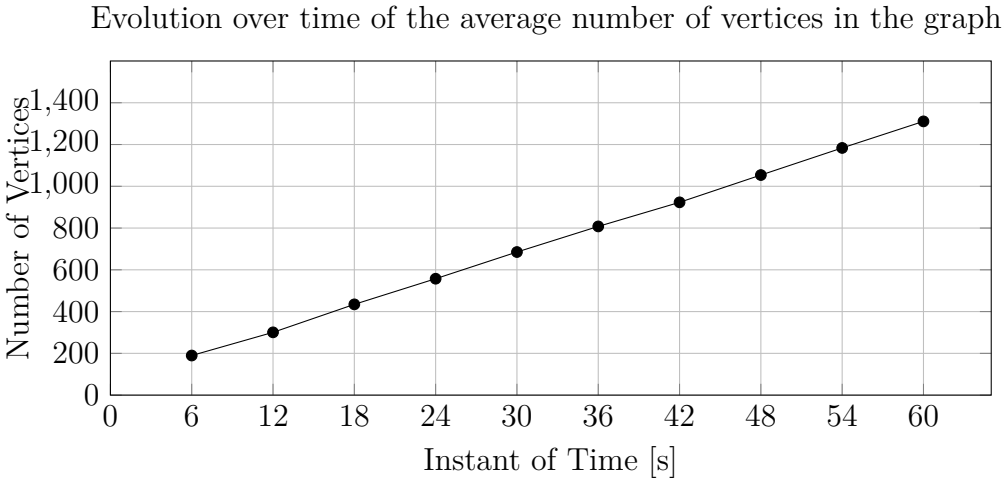


Figure 4.10: Narrow and constrained environment, first map.





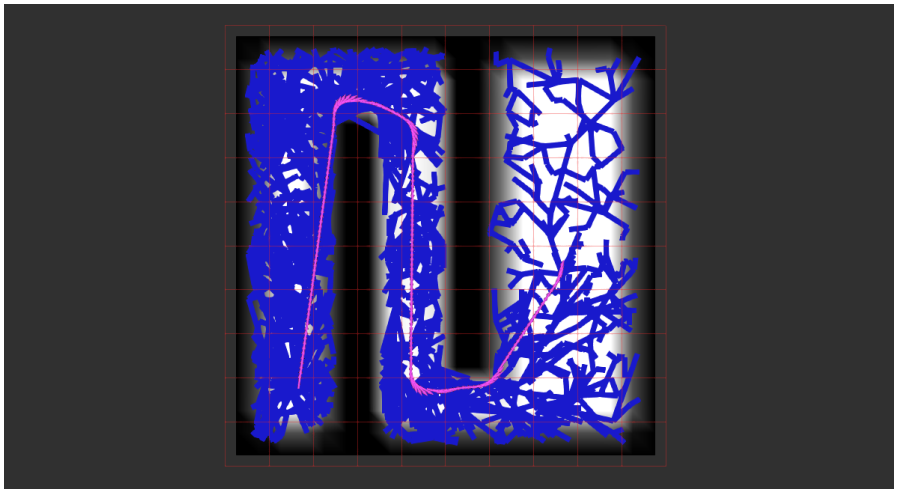


Figure 4.11: Example of solution in the first narrow and constrained environment.

Nothing unexpected is observed in this test; as clearly shown in Figure 4.11, the goal pose is reached while properly moving inside the narrow passage.

4.3.2 Narrow and Constrained Environment, Second Map

Similar to the previous test, this test is performed in a narrow and constrained environment. However, this test introduces a new obstacle in the path computation clearly showed at Figure 4.12; the presence of multiple passages for reaching the same point stresses the algorithm, that has to choice between two pathways for reaching the goal. This map, in fact, is designed for having two different paths of more or less the same length between the start and the goal. So, depending on the

orientation of the goal, the algorithm may return the best route for reaching that desired pose in the space.

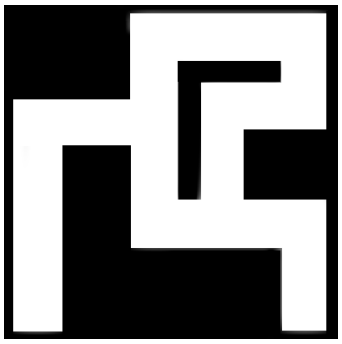
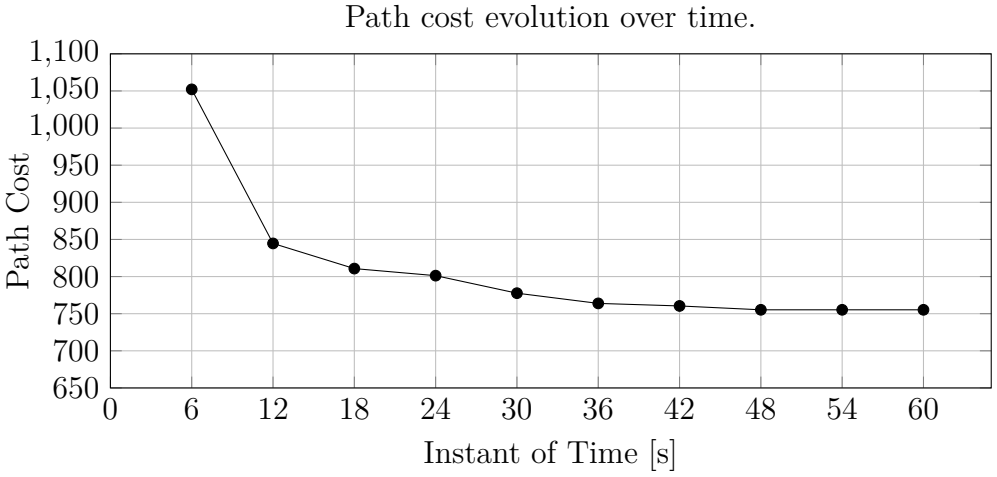
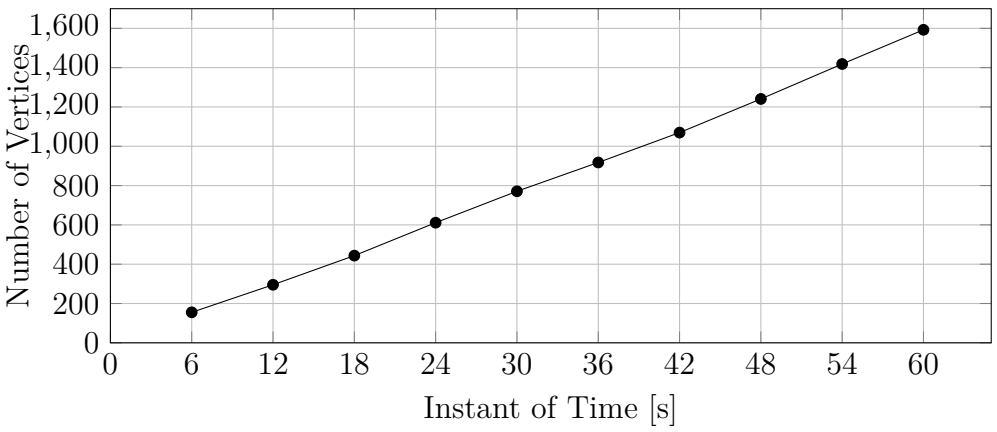


Figure 4.12: Narrow and constrained environment, second map.





The algorithm behaves exactly as expected. As can be seen in Figure 4.13, the

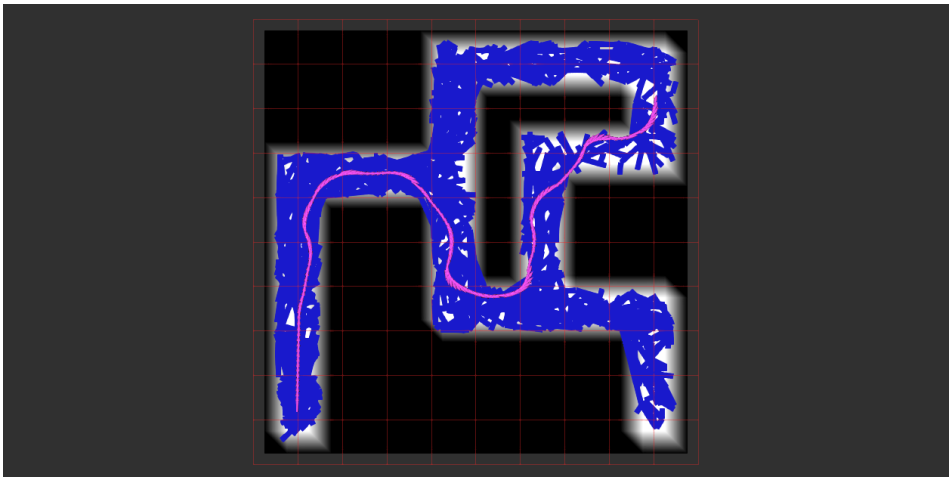


Figure 4.13: Example of solution in the second narrow and constrained environment.

logic privileges the trajectories that approach the final pose in the right direction, and not those passing from the other pathway that need a final rotation.

4.3.3 Empty Environment

An interesting test of the proposed algorithm performances is to plan a trajectory in an empty environment. In the absence of obstacles, the planner is supposed to plan straight trajectories, being designed for achieving the shortest path with the smallest possible input variation. In order to properly perform this test, an empty map is used. The evaluations for this test are done with respect to path length. Multiple simulations are necessary because of the probabilistic nature of the algorithm and, for each test, the path cost, the number of states as well as the path length are reported. Start and goal are 22.63 meters away in the sense of Euclidean distance, so the same length of the planned path is expected.

As clearly shown in Table 4.16, regardless the path cost and the number of samples in the constructed graph, the distance returned by the algorithm is always the same. In this sense, the path simplifier action is really effective; at the end of the path computation, the useless path vertices are removed. It is clear that, for this test, the only useful states are the start and the goal, being the trajectory supposed to be straight, so the logic directly joints the start and the goal through a straight line following the reference.
The solutions provided by the algorithm for problems of this kind could be optimized: the planner wastes a lot of time searching for a good path by joining the sampled states, while the best solution is to try to directly connect the start and the goal

Simulation Results			
Simulation Number	Path Cost	Path Length	Number of Vertices
First Simulation	407.838	22.63m	1680
Second Simulation	493.952	22.63m	1837
Third Simulation	422.915	22.63m	1877
Fourth Simulation	434.476	22.63m	1856
Fifth Simulation	430.796	22.63m	1800
Average	437.995	22.63m	1810

Table 4.16: Empty environment: simulation results.

Figure 4.14: Example of solution in the empty environment.

state if there are not obstacles along the straight line joining them.

4.3.4 Two Obstacles Avoidance

Another interesting test that is useful to be performed is the test in a map with two obstacles. This test aims to demonstrate the effectiveness of the planner in open environments where the path is imposed. In fact, in the map used for this test, reported at Figure 4.15, the motion of the UAV is forced to be through two obstacles after having turned around one of them. Experimental results considering the same data as before are plotted in the following.

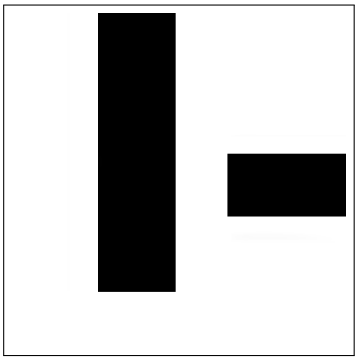
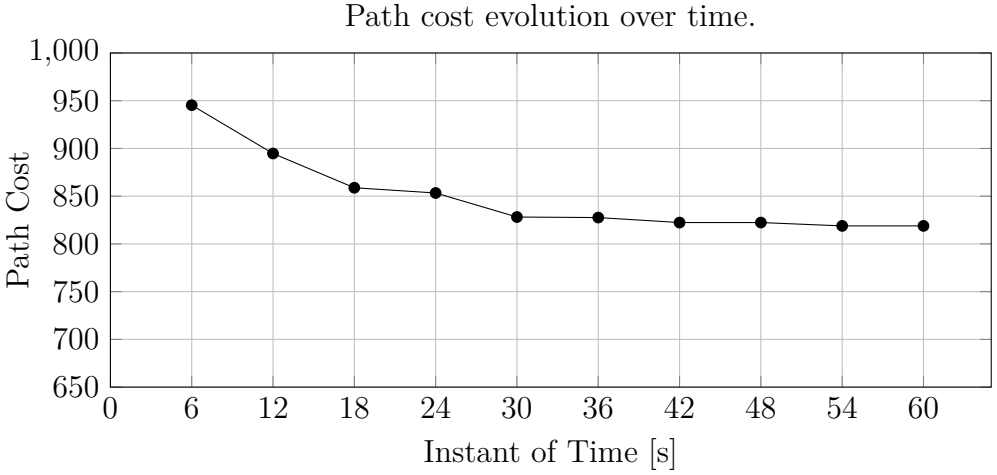
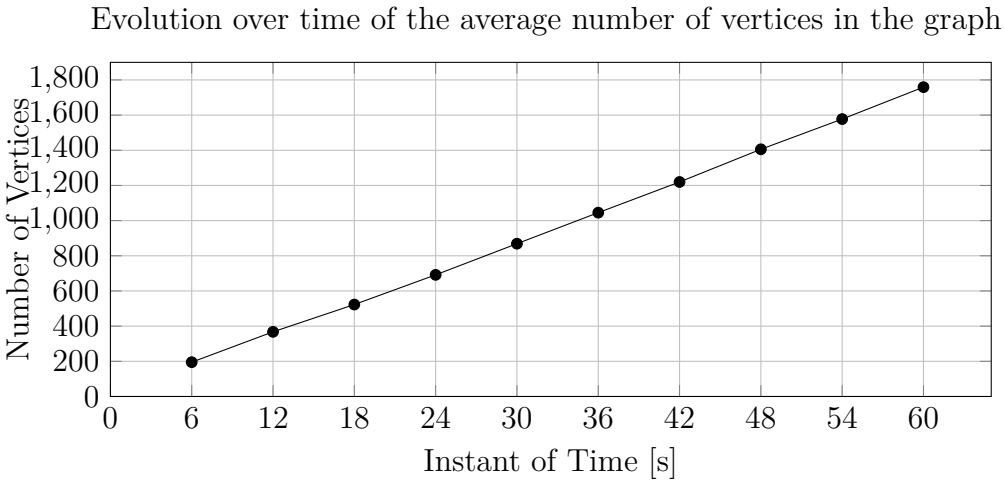


Figure 4.15: Two obstacles avoidance test map.





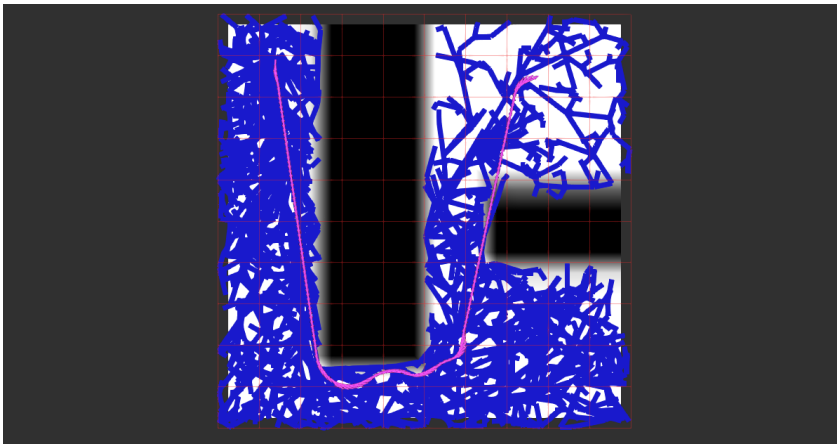


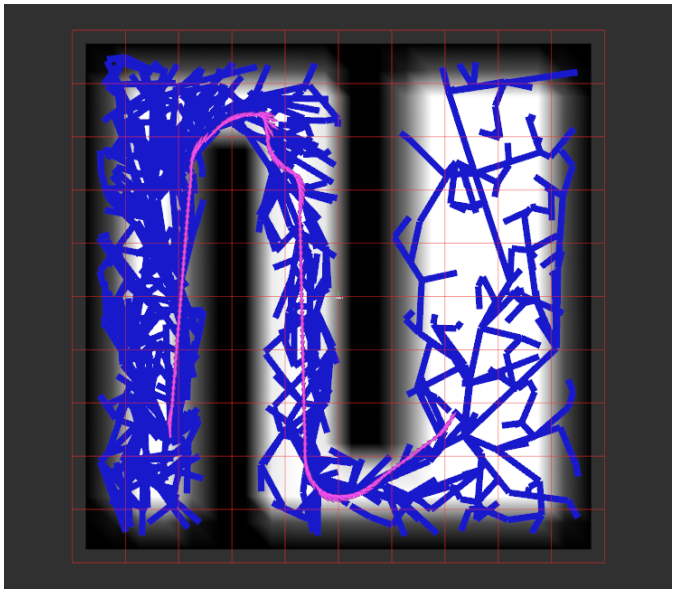
Figure 4.16: Example of a solution in the two obstacles avoidance test map.

This test does not present peculiarities to be mentioned. The algorithm behaves exactly as expected, trying to minimize the distance for approaching the goal while avoiding the obstacles encountered during the motion. Analyzing Figure 4.16, it is possible to appreciate the work of the path simplifier; when no turning action is requested, the simplifier directly joins states for obtaining a "short-cut" of the path. In this way, perfectly straight trajectories can be obtained. For the path sections close to a corner, it is possible to see that no simplification in terms of motion is done and the trajectory is that planned by the RRT* algorithm exploiting the UAV dynamic model.

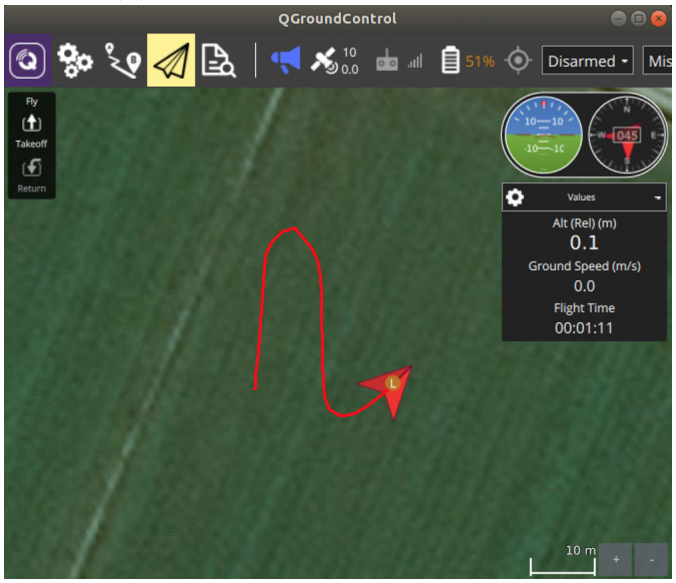
4.4 SITL Testing

The quality of the planner is graphically analyzed in the previous sections, but the effectiveness of its solution with a real UAV is not tested. This section is totally devoted to the Software In The Loop testing of the implemented algorithm, where the computed trajectory is passed by PX4 autopilot for understanding if it could be handled and executed by a real drone. This test is done for validating the implemented trajectory planner. In fact, after its computation, the optimal trajectory is published as message on a predefined Topic; an implemented ROS Node, that works as middleware in the communication between the planner and the autopilot, reads this trajectory and translates it as a *Waypoint list*. Finally, the translated trajectory is passed to the autopilot, that performs the desired motion showing the behaviour of a UAV in a three-dimensional world simulated on Gazebo and represented on the QGroundControl application.
For better visualizing the resulting path, the trajectory as a *Waypoint list* is scaled, doubling its size with respect to the one visualized on RViz. As clearly shown in

Figure 4.17b, reporting an example of SITL testing, the trajectory planned by the proposed logic can be handled and performed by PX4 autopilot, resulting in the vary same path computed by the algorithm. In this way, it is possible to state that the implemented trajectory planner can be used for real UAVs.



(a) Trajectory represented on RViz



(b) Trajectory from QGroundControl

Figure 4.17: Comparison between the trajectory planned on RViz and the one performed by PX4 autopilot for the SITL testing.

4.5 Limitations and Possible Solutions

Even if the proposed logic seems working fine, it presents some limitations.
The first highlighted limit is the speed of the UAV. As properly analyzed at Section 4.2.1, this algorithm is really effective when planning with low speed. At high speed, the imposed input constraints do not allow to perform agile maneuvers, worsening the quality of the final trajectory. A possible solution to this problem is to compute the optimal speed components during the planning procedure, but this addiction increases the overall problem complexity, slowing down the whole algorithm.
However, the biggest limit of the implemented logic is the time requested for the computation of the MPC problem solution. For analyzing a single state connection, the prediction and optimization process repeats iteratively for a number of times that depends on the distance between the states, but that is, in the worst cases, equal to 25 iterations, given the current Prediction Sampling Time, the UAV Speed Module and the Planner Range. The RRT* algorithm recalls the MPC tool whenever a newer state is attempted to be added as vertex into the graph, so it is clear that these recursive MCP solution computations reduce the number of sampled states. Needing an average time of 0.01 seconds for computing each solution, the delay introduced by this logic forces its computation to be done offline. In fact, because of its timing limitations, this software is not designed for working in real-time. A possible way to make it able to work in real-time applications is to optimize the algorithm, exploiting multi-core hardware for paralleling the planner processes to speed it up.
Indeed, the presence of obstacles is a-priori known; a map with fixed obstacles is read by the trajectory planner, that sets the proper sampling bounds and starts the trajectory computation. The obstacles that can be detected by the proposed logic are the ones already present on the map when it is read by the algorithm and not possible obstacles occurring when the UAV is moving. This planner does not take into account the presence of low level hardware in the motion planning phase, like sensors and cameras, that are useful for the obstacle avoidance; it only focuses on discovering a path between a start and a goal pose in a two-dimensional map, optimizing the motion in a-priori known environment.
The error that is encountered during the different test phases concerns the final pose orientation in the returned trajectory. An example of this error is showed in Figure 4.18. For the way this logic is designed, the algorithm stops planning whenever a neighbourhood of the goal is reached; possible orientation differences between the planned final goal and the desired goal are never taken into consideration. This choice is due to the fact that the algorithm is designed for working locally, planning short partial trajectories; so, the possible final pose errors are adjusted at the beginning of the successive motion.

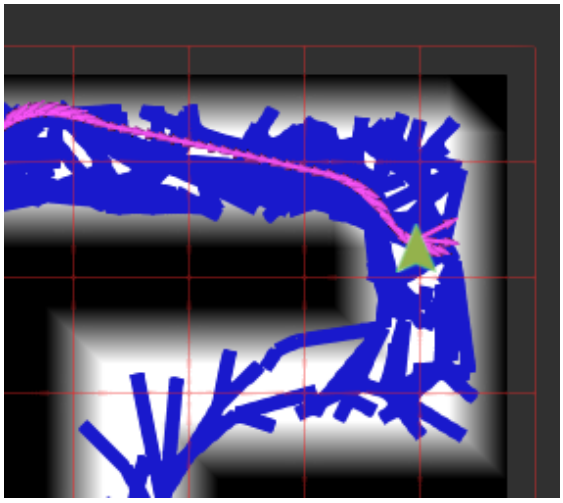


Figure 4.18: Example of the orientation error. The green arrow represents the goal pose that is not reached in terms of heading.

This problem could be solved by introducing a cost contribution in the cost function that evaluates the final orientation error; in this way, not only the length of the path, the input variation and the state error with respect to the reference are minimized, but also the final pose orientation error.
Another limitation of this work is the capability of planning only into a two-dimensional space. As already explained, the drone model, RRT* algorithm and the MPC logic are able to work even in a three-dimensional environment, but the reference trajectory is computed in 2D.
In fact, the difficulty of the interpolation phase grows considerably with the increasing of the number of dimensions. Referring to the proposed work, a sphere-line interpolation is the natural evolution in 3D world of the reference adopted for this planner.

Chapter 5

Conclusions

The main contribution of this thesis focuses on the development of a trajectory planning algorithm. The proposed logic, based on the RRT* planning algorithm, provides good performances in terms of quality of the final solution.
The MPC logic developed for accomplishing the desired tasks is really useful for the path computation; even if this prediction is really expensive in terms of computer resources and time, exploiting the system model, each sampled state is inserted as vertex in the constructed graph and labelled with a cost only if the predicted motion for reaching it is feasible. In this way, the resulting optimal path is composed by states whose connections are previously checked by the MPC logic. The application of the MPC tool after the optimal path computation performed by the RRT* algorithm allows to obtain a resulting smooth and continuous trajectory that is optimal for the UAV, being calculated on the basis of its dynamical model together with the imposed input constraints.
As already explained in Section 4.5, this trajectory planner is designed for working locally and performing small motions; this logic is supposed to be run several times for performing small sequential moves between the desired starts and goals, so, even the presence of small final orientation errors with respect to the desired goal poses are accepted, being completely compensated by the successive trajectory.
Furthermore, the SITL testing phase shows that this planning algorithm can be applied to real UAV. The computed trajectories are not only marked as a sequence of states in the space with a given speed, but they are also marked with a set of input that could be passed to a real UAV. It is important to point out that the input sets returned by the algorithm need some arrangements for being applied in a real world; gravity and accelerations need to be compensate for achieving precise trajectories.

The proposed planner could be attempted to be improved by adopting some modifications:

- The first possible improvement refers to the space dimension. This thesis project is designed for working in a two-dimensional space; so the variables sampled by the RRT* algorithm in each sample are three, two positions and one orientation in the space. Changing the state space from *SE(2)* to *SE(3)* allows to extend the planning capabilities to the three-dimensional space, but it means adding one positional contribution and two orientations at each sample data-structure, increasing considerably the complexity of the algorithm. Moreover, the presence of a third positional coordinate makes a bigger amount of samples necessary for obtaining a good final trajectory, so timing limitations have to be taken into consideration in order to provide planning effectiveness;

- Another possible improvement is the use of Dubins path for the reference trajectories definition. This different kind of approach, that could be used even in *SE(3)* state space, allows to obtain better curvatures with respect to the line-circumference interpolation. However, this different reference is not sure to be better than the implemented interpolation technique; close start and goal states with very different orientations that are attempted to be joined in the space may need long curvatures for approaching the final orientation.

- Concerning the time performances of the planner, it is possible to change the MPC logic. Instead of returning only the first state calculated by the MPC, it is possible to return more than one state from each optimization cycle. In this way, the number of solving iterations can be considerably decreased, making the algorithm able to sample an higher number of states. On the contrary, the quality of the planning may decrease and this different approach cannot be considered an MPC anymore.

- The last improvement that is proposed is about the possibility of improving the quality of the planning procedure by inserting disturbances directly inside the UAV state space representation. By using an Extended Kalman Filter (EKF), disturbances prediction can be done, increasing the overall quality of the planner. Even if the estimation phase is quite heavy in terms of computational effort, it is the best possible way of modelling the UAV dynamics.

In conclusion, the proposed planner represents an efficient way of planning the UAV trajectory inside an unknown two-dimensional environment. The use of the MPC logic together with the RRT* algorithm turned out to be a good planning strategy, even with its timing limitations. This technology presents several limitations and possible improvements that need to be considered step-by-step during the developing phase for preserving the planning effectiveness.

Appendix A

CVXGEN Code

```
dimensions

  m = 3 # dimension of inputs.
  nx = 8 # dimension of state vector.
  T = 10 # prediction horizon -1.

end

parameters

  A ( nx , nx ) # dynamics matrix.
  B ( nx , m) # transfer (input) matrix.
  Q ( nx , nx ) psd # state cost , positive semidifined.
  Q_final ( nx , nx ) psd # final state penalty, positive semidifined.
  R (m,m) psd # input penalty, positive semidifined.
  R_delta (m,m) psd # delta inputpenalty, positive semidifined.
  x[0] (nx) # initial state .
  u_prev (m) # previous input applied to the system.
  u_max (m) # input amplitude limit.
  u_min (m) # input amplitude limit.
  x_ss[t] (nx), t=0..T+1 # reference state.

end

variables

  x[t] ( nx ), t=1..T+1   # state.
  u[t] (m), t=0..T  # input.

end

minimize

  quad(x[0] - x_ss[0], Q) + quad(u[0]-u_prev,R_delta) + sum[t=1..T]( quad(x[t]-x_ss[t],Q) +
  + quad(u[t] - u[t-1],R_delta) ) + quad(x[T+1]-x_ss[T+1],Q_final)

subject to

  x[t+1]==A*x[t]+B*u[t], t=0..T
  u_min <= u[t] <= u_max, t=0..T

end
```

Appendix B

text.launch File

Listing B.1: text.launch file content.

```
<launch>
<!--
  <node pkg="tf" type="static_transform_publisher" name="fake_tf"
    args="0 0 0 0 0 0 1 world map 20" />
-->
    <node name="risk_aware_pp" pkg="risk_aware_pp" type="
    risk_aware_pp" output="screen">
    <node name="risk_aware_pp" pkg="risk_aware_pp" type="
    risk_aware_pp" output="screen">
    <param name="solve_time" type="double" value="5.0" />
    <param name="planner_type" type="string" value="MPC_RRTstar" />
    <param name="planner_goal_bias" type="double" value="0.1" />
    <param name="planner_range" type="double" value="5.0" /> <!-- 30
    -->
    <param name="relative_validity_check_resolution" type="double"
    value="0.01" />
    <param name="start_flight_altitude" type="double" value="0.0" />
    <param name="goal_flight_altitude" type="double" value="0.0" />
    <param name="roll_limit" type="double" value="0.436332" />
    <param name="pitch_limit" type="double" value="0.43633" />
    <param name="thrust_max" type="double" value="10.1934" />
    <param name="thrust_min" type="double" value="-4.80" />
    <param name="roll_time_constant" type="double" value="0.25" />
    <param name="pitch_time_constant" type="double" value="0.255" />
    <param name="prediction_sampling_time" type="double" value="0.1"
    />
    <param name="roll_gain" type="double" value="0.9" />
    <param name="pitch_gain" type="double" value="0.9" />
    <param name="q1" type="double" value="40.0" />
    <param name="q2" type="double" value="40.0" />
    <param name="q3" type="double" value="60.0" />
```

```
26      <param name="q4" type="double" value="20.0" />
27      <param name="q5" type="double" value="20.0" />
28      <param name="q6" type="double" value="25.0" />
29      <param name="q7" type="double" value="0.0" />
30      <param name="q8" type="double" value="0.0" />
31      <param name="p1" type="double" value="100.0" />
32      <param name="p2" type="double" value="100.0" />
33      <param name="p3" type="double" value="1.0" />
34      <param name="p4" type="double" value="1.0" />
35      <param name="p5" type="double" value="1.0" />
36      <param name="p6" type="double" value="1.0" />
37      <param name="p7" type="double" value="0.0" />
38      <param name="p8" type="double" value="0.0" />
39      <param name="dist1" type="double" value="0.15" />
40      <param name="dist2" type="double" value="0.35" />
41      <param name="cost1" type="double" value="2.0" />
42      <param name="cost2" type="double" value="57.3248" /> //"57.3248"
43      <param name="cost3" type="double" value="0.0" />
44      <param name="speed_module" type="double" value="1.5" />
45      <param name="angle_difference" type="double" value="1.57" />
46      <param name="num_of_iter" type="int" value="200" />
47      <param name="state" type="string" value="SE2" />
48      <param name="map_file" value="$(find risk_aware_pp)/config/map/
        mappa_thesis.png" 5/>
49    </node>
50    <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
        risk_aware_pp)/config/rviz/grid_map.rviz"/>
51  </launch>
```

Appendix C

CVXGEN Statistics

Figure C.1 shows the time needed by several optimization methods for solving the same Model Predictive Control problem presented in [19], where m is the number of input, n is the number of state variables and T is the prediction horizon.

	Size (m, n, T)		
	Small (2, 3, 10)	Medium (3, 5, 10)	Large (4, 8, 20)
CVX and SeDuMi	870 ms	880 ms	1.6 s
Scalar parameters	41	105	249
Variables, original	55	88	252
Variables, transformed	77	121	336
Equalities, transformed	33	55	168
Inequalities, transformed	66	99	252
KKT matrix dimension	242	374	1008
KKT matrix nonzeros	552	1025	3568
KKT factor fill-in	1.30	1.44	1.60
Code size	337 kB	622 kB	2370 kB
Generation time, i7	4.3 s	13 s	200 s
Compilation time, i7	3.6 s	9.4 s	41 s
Binary size, i7	175 kB	351 kB	1445 kB
CVXGEN, i7	85 µs	230 µs	970 µs
CVXGEN, Atom	1.7 ms	3.3 ms	13 ms
Maximum iterations required, 99.9%	13	13	12
Maximum iterations required, all	23	24	24

Figure C.1: Computation time for finding the solution to the same problem by different solvers.

Appendix D

Euclidean Distance

The Euclidean distance between two points is the length of the straight line joining the points.
In Cartesian coordinates, given two points $\mathbf{p} = [p_1, p_2, \ldots, p_n]$ and $\mathbf{q} = [q_1, q_2, \ldots, q_n]$ belonging to the *Euclidean n-space*, the Euclidean distance of the points can be computed exploiting the *Pythagorean theorem*:

$$d(p, q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2}$$

For the purpose of this thesis, being developed in 2-D space, the previous formula can be re-written for $n = 2$:

$$d(p, q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2}$$

Appendix E

mavros_msgs/Waypoint Message

Listing E.1: Waypoint.msg structure

```
# Waypoint.msg
#
# ROS representation of MAVLink MISSION_ITEM
# See mavlink documentation

# see enum MAV_FRAME
uint8 frame
uint8 FRAME_GLOBAL = 0
uint8 FRAME_LOCAL_NED = 1
uint8 FRAME_MISSION = 2
uint8 FRAME_GLOBAL_REL_ALT = 3
uint8 FRAME_LOCAL_ENU = 4

# see enum MAV_CMD and CommandCode.msg
uint16 command

bool is_current
bool autocontinue
# meaning of this params described in enum MAV_CMD
float32 param1
float32 param2
float32 param3
float32 param4
float64 x_lat
float64 y_long
float64 z_alt
```

Bibliography

[1] Jacob T Schwartz and Micha Sharir. «On the "piano movers'" problem I. The case of a two-dimensional rigid polygonal body moving amidst polygonal barriers». In: *Communications on pure and applied mathematics* 36.3 (1983), pp. 345–398 (cit. on p. 1).

[2] Alessandro Gasparetto, Paolo Boscariol, Albano Lanzutti, and Renato Vidoni. «Path Planning and Trajectory Planning Algorithms: A General Overview». In: *Mechanisms and Machine Science* 29 (Mar. 2015), pp. 3–27. DOI: 10.1007/978-3-319-14705-5_1 (cit. on p. 1).

[3] Chad Goerzen, Zhaodan Kong, and Bernard Mettler. «A survey of motion planning algorithms from the perspective of autonomous UAV guidance». In: *Journal of Intelligent and Robotic Systems* 57.1-4 (2010), p. 65 (cit. on p. 2).

[4] Yoshiaki Kuwata, Gaston A Fiore, Justin Teo, Emilio Frazzoli, and Jonathan P How. «Motion planning for urban driving using RRT». In: *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2008, pp. 1681–1686 (cit. on p. 2).

[5] František Duchoň, Andrej Babinec, Martin Kajan, Peter Beňo, Martin Florek, Tomáš Fico, and Ladislav Jurišica. «Path planning with modified a star algorithm for a mobile robot». In: *Procedia Engineering* 96 (2014), pp. 59–69 (cit. on p. 5).

[6] Emilio Frazzoli, Munther A Dahleh, and Eric Feron. «Real-time motion planning for agile autonomous vehicles». In: *Journal of guidance, control, and dynamics* 25.1 (2002), pp. 116–129 (cit. on p. 5).

[7] Reagan L Galvez, Elmer P Dadios, and Argel A Bandala. «Path planning for quadrotor UAV using genetic algorithm». In: *2014 International Conference on Humanoid, Nanotechnology, Information Technology, Communication and Control, Environment and Management (HNICEM)*. IEEE. 2014, pp. 1–6 (cit. on p. 6).

[8] Safaa H Shwail and Alia Karim. «Probabilistic roadmap, A*, and GA for proposed decoupled multi-robot path planning». In: *Iraqi Journal of Applied Physics* 10.2 (2014), pp. 3–9 (cit. on p. 6).

[9] Guanjun Ma, Haibin Duan, and Senqi Liu. «Improved ant colony algorithm for global optimal trajectory planning of UAV under complex environment.» In: *IJCSA* 4.3 (2007), pp. 57–68 (cit. on p. 7).

[10] Sertac Karaman and Emilio Frazzoli. «Sampling-based algorithms for optimal motion planning». In: *The international journal of robotics research* 30.7 (2011), pp. 846–894 (cit. on p. 10).

[11] Morgan Quigley, Josh Faust, Tully Foote, and Jeremy Leibs. «ROS: an open-source Robot Operating System». In: (cit. on p. 13).

[12] Yu-Geng XI, Dewei Li, and Shu Lin. «Model Predictive Control — Status and Challenges». In: *Acta Automatica Sinica* 39 (Mar. 2013), pp. 222–236. DOI: 10.1016/S1874-1029(13)60024-5 (cit. on p. 21).

[13] John F Keane and Stephen S Carr. «A brief history of early unmanned aircraft». In: *Johns Hopkins APL Technical Digest* 32.3 (2013), pp. 558–571 (cit. on p. 24).

[14] Mina Kamel, Thomas Stastny, Kostas Alexis, and Roland Siegwart. «Model predictive control for trajectory tracking of unmanned aerial vehicles using robot operating system». In: *Robot operating system (ROS)*. Springer, 2017, pp. 3–39 (cit. on pp. 25, 27, 30, 39, 43).

[15] Jung Cheon, Han Kyoohyung, Seong-Min Hong, Junsoo Kim, Suseong Kim, Hoseong Seo, Hyungbo Shim, and Yongsoo Song. «Toward a Secure Drone System: Flying with Real-time Homomorphic Authenticated Encryption». In: *IEEE Access* PP (Mar. 2018), pp. 1–1. DOI: 10.1109/ACCESS.2018.2819189 (cit. on p. 27).

[16] Sertac Karaman, Matthew R Walter, Alejandro Perez, Emilio Frazzoli, and Seth Teller. «Anytime motion planning using the RRT». In: *2011 IEEE International Conference on Robotics and Automation*. IEEE. 2011, pp. 1478–1483 (cit. on pp. 33–35).

[17] Sertac Karaman and Emilio Frazzoli. «Optimal kinodynamic motion planning using incremental sampling-based methods». In: *49th IEEE conference on decision and control (CDC)*. IEEE. 2010, pp. 7681–7687 (cit. on pp. 34, 55).

[18] Juan Cortés, Léonard Jaillet, and Thierry Siméon. «Disassembly Path Planning for Complex Articulated Objects». In: *Robotics, IEEE Transactions on* 24 (May 2008), pp. 475–481. DOI: 10.1109/TRO.2008.915464 (cit. on p. 35).

[19] Jacob Mattingley and Stephen Boyd. «CVXGEN: A code generator for embedded convex optimization». In: *Optimization and Engineering* 13.1 (2012), pp. 1–27 (cit. on pp. 38, 85).

[20] Michael Grant and Stephen Boyd. *CVX: Matlab software for disciplined convex programming, version 2.1*. 2014 (cit. on p. 38).

[21] Francesco Borrelli, Alberto Bemporad, and Manfred Morari. *Predictive control for linear and hybrid systems*. Cambridge University Press, 2017 (cit. on p. 39).

[22] Joseph A Marty. *Vulnerability analysis of the mavlink protocol for command and control of unmanned aircraft*. Tech. rep. AIR FORCE INSTITUTE OF TECHNOLOGY WRIGHT-PATTERSON AFB OH GRADUATE SCHOOL OF ..., 2013 (cit. on p. 50).