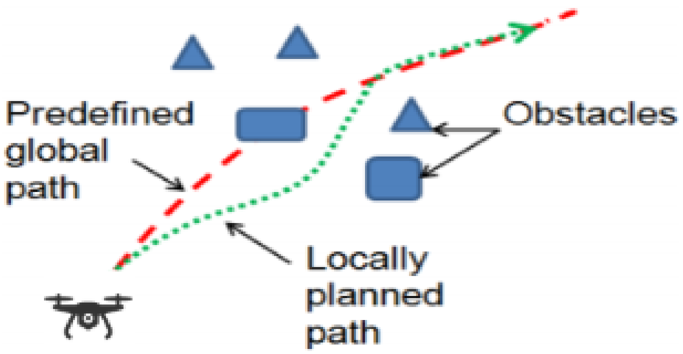


Real-Time Onboard Path Planning for Quadrotors

Hélder Gonçalves Espírito Santo Cristóvão

Thesis to obtain the Master of Science Degree in

Aerospace Engineering

Supervisor: Prof. Afzal Suleman

Examination Committee

Chairperson: Prof. Paulo Jorge Coelho Ramalho Oliveira
Supervisor: Prof. Afzal Suleman
Member of the Committee: Prof. Rodrigo Martins de Matos Ventura

June 2021

I dedicate my dissertation work to my family and friends, those who are here and the one's who have left for a better place.

Acknowledgments

Firstly, I wish to thank my supervisor Dr.Suleman for the singular experience of working abroad in Victoria during the completion of my thesis, his full and always available support, assistance and guidance throughout the entire process. I would like to thank my fellow portuguese colleagues with whom I spent my stay for their company specially through the rough times. I would also thank the University of Victoria Centre for Aerospace Research for the resources provided and the entire personnel for their assistance. and continued support. Finally I would also extend my thanks to my family and friends back home for helping me make this journey from the beginning.

Resumo

O trabalho desenvolvido nesta tese tem como foco o desenvolvimento de uma plataforma que possa ser empregue a bordo de um UAV e lhe forneça as capacidades necessárias para que um consiga deslocar-se da partida até ao destino final sem ser perturbado por obstáculos estáticos e/ou dinâmicos que apareçam no seu percurso.

A abordagem seguida é composta por um algoritmo de planeamento de uma trajectória em tempo real que se encontra dividido em duas componentes: uma fase de planeamento Offline/Pré-voo e uma fase de replaneamento Online(em tempo real).

Na primeira fase, as trajectórias são geradas num ambiente conhecido *a priori* através de uma combinação de um algoritmo de amostragem (RRT-COnnect e Informed RRT* são utilizados) que gera um caminho que é posteriormente convertido numa trajectória suave por meio de um método de otimização baseado na minimização da quarta derivada da posição ("snap").

Na segunda fase, a trajectória inicial é corrigida em tempo real perante situações de potencial colisão com obstáculos dinâmicos. A correção é baseada numa trajectória de transição que segue o método utilizado na fase offline com os ajustes necessários para poder ser realizada em tempo real.

Os ambientes onde os algoritmos atuam foram modelados como "Octomaps".

De modo a testar os algoritmos densolvidos, foram feitas simulações não-físicas e físicas (ditas simulações "Software-in.the-loop").

Keywords: a bordo, replaneamento, RRT-Connect, Informed RRT*, suave, Snap, Octomap, Software-in-the-Loop

Abstract

The work done in this thesis is focused on the development of a framework that empowers an UAV with the capability of going from a start to a destination while simultaneously avoiding static and dynamic obstacles during its course. The framework was developed with the intention of running onboard of an UAV the associated computational limitations.

The framework is composed of a real-time trajectory planning algorithm that is split into a two-step approach: an Offline/Pre-Flight Path Planning and an Online/Real Time Path Replanning.

In the first step techniques are implemented to generate trajectories in a known static environment. The proposed solution makes use of sampling-based motion planning algorithms (both the RRT-Connect and the Informed RRT* are used) to find an initial feasible path that's then feed into an optimization method which turns it into a feasible trajectory. This optimization method seeks to find smooth trajectories by minimizing the 4th derivative of position (Snap).

The second step consists of a real time avoidance module which allows the UAV to avoid dynamic obstacles in its course by means of a local generation of a transitioning trajectory around them. A pragmatic approach is used to tackle this problem that leverages the quick generation of trajectories provided by the aforementioned optimization method.

The algorithm was developed under the assumption that the UAV surroundings are modeled in the form of an Octomap.

To test and validate the capabilities of the developed framework, simple simulations were designed and Software-in-the-loop tests were carried out using PX4 and Gazebo as simulation tools.

Keywords: onboard, RRT-Connect, Informed RRT*, Replanning, smooth, Snap, Octomap, Software-in-the-Loop, PX4, Gazebo

Contents

List of Tables

List of Figures

Nomenclature

Greek symbols

ω	Angular velocity.
ϕ , θ , ψ	Roll, Pitch and Yaw angles.
σ	Optimal path planning function.
Θ	Vector of Euler angles.

Roman symbols

A	Mapping matrix.
C	Configuration space.
c	Polynomial coefficients.
d	Derivatives vector.
f	Path planning function.
g	Trajectory planning function.
J	Quadratic cost function.
o	Obstacle.
$P(x)$	Polynomial function.
Q	Hessian matrix.
W	Workspace.
x, y, z	Cartesian components.
s	Robot state vector.
p, v, a, j, sn	Position, Velocity, Acceleration, Jerk and Snap.
q	Configuration vector.

Subscripts

B	Body fixed frame.
free	Free condition.
max	Hard constraints.
max	Maximum value condition.
occupied	Occupied condition.
soft	Soft constraints.
W	World fixed frame.

Superscripts

T	Transpose.

Glossary

UAV, Unmanned Aerial Vehicle (commonly known as drone) is defined as ”an aircraft without a human pilot on board”.

S&A, Sense and Avoid consists of the systems necessary for sense, detect and avoidance of obstacles in autonomous navigation.

PRM, Probabilistic Roadmap Methods use randomly sample nodes from the search space and try to find the shortest sequence of collision free configurations between the start and goal states using Dijkstra algorithm.

RRT, Rapid Exploring Random Trees are an example of sampling based algorithms by building a tree from samples drawn randomly from the search space. The search in the space is inherently biased towards the goal state.

PF, Potential fields model the world by a force field with obstacles acting as repulsive poles and defined goals as attractive poles therefore there is an attractive force that increases towards the goal and repulsive forces that increase towards obstacles.

MDP, Markov decision process is a discrete-time dynamic stochastic control process where the outcome of a system depends on the current state and an action.

MA, Manoeuvre Automata are a collection of maneuver generation techniques that work by creating a continuous trajectory over predefined waypoints.

DC, Dubins Curve are a set of lines that satisfy motion constraints by a combining curvature arcs and/or straight line.

VO, Velocity Obstacle based methods are a common approach to dynamic obstacle avoidance and are based on the concept of the possible velocities of a robot that would put it in collision with an obstacle (static or dynamic) at a future point in time.

MPC, Model predictive control is an example of an optimization-based algorithm using a nonlinear control technique that uses a dynamic model of a plant and past control history to optimize future states over a defined time period and subject to a cost function, therefore the trajectory optimization problem is solved through feedback control.

MILP, Mixed Integer Linear Programming is a type of optimization problem in which all formulated constraints and objective function are linear and some design variables are restricted to discrete/integer values.

FCL, Flexible Collision Library is a library for performing collision checking and distance computation between geometric models.

OMPL, Open Motion Planning Library is an open source library containing many state-of-the-art sampling-based motion planning algorithms.

SITL Software-In-The-Loop simulates mathematical models and provides engineers with virtual simulation environments for developing and testing the control of systems without the need of hardware.

ROS, Robot Operating System is an open-source software framework for robotic and drone development.

Chapter 1

Introduction

Figure 1.1: Artists rendition of air mobility in a metropolitan area in the future. Taken directly from [1].

An Unmanned Aerial Vehicle (UAV or commonly known as *drone*) is defined as "*an aircraft without a human pilot on board*". According to International Civil Aviation Organization (ICAO) an UAV can be classified according to its degree of automation, meaning it can be non-autonomous if remotely piloted or autonomous otherwise. In reality, non-autonomous UAVs might have some autonomous capability such as a return-to-base functionality for example. Nevertheless, the recent ascension of UAV hardware as a mean of fulfilling the requirements of civilian, commercial, military and aerospace applications (due to the versatility they offer), along with its projected growth in the next decade [2] means these aircraft's are increasingly becoming more robust, reliable and above all more affordable.
On pair with this, we have also experienced recently a fast growth in smaller and powerful computational processors, the advent of more precise and accurate sensors which paired with the development of efficient and powerful Artificial Intelligence (AI) algorithms [3] have opened an entire world of possibilities within multiple fields of technology, specifically the latter which aims to solve complex problems whose solutions would simply not be able to be achieved otherwise. Autonomous navigation stands out as an example of a field which has seen significant research and advancements.
Companies like Amazon [4] have begun taking advantage of incorporating autonomous capabilities in drones by offering autonomous delivery service using drones. Worldwide, governments are starting to

take advantage of autonomous drones in military operations [5]. Agriculture is a field which can greatly benefited from these autonomous capabilities [6] due to the simplistic nature of the operations performed which go from for example a fly-by above a field with the objective to collect images or even disseminate seeds.

1.1 Context and Motivation

Most current UAVs employed in other activities require at all times a pilot that is responsible for ensuring its safe navigation and compliance with the rules of the air. The shift from having a Human operator to a complete independent, reliable and robust autonomous framework represents the final leap in harnessing the full potential that UAVs could offer. However, this is not yet possible to achieve but progressively empowering UAVs with more autonomous capabilities would lead to a decrease in both operational cost and operator workload while potentially improving its safety in the event of loss of communications. It is also true that if the autonomous system fails to meet rigorous operational standards, this would have the exact opposite effect and susceptibility to failure would increase. One area of particular interest in the future for this consists in tackling the arising problem of Urban Air Mobility (an artist depiction can be seen in figure 1.1), that pertains to aerial urban transportation systems. It is estimated that in 2030, 60% of the world's population will live in cities [7] which will pose an immense problem regarding personal and commercial ground transportation within these urban environments due to infrastructure congestion. To achieve this, there is an immediate basic need of having guidelines which rule UAV urban navigation. While this is a topic that still requires immense development, there are already entities which have tried to tackle this such as the European Union through the European Aviation Safety Agency (EASA) that recently released a set of regulations for different UAV operations according to its perceived level of risk:

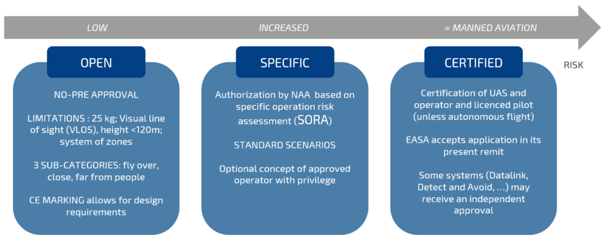
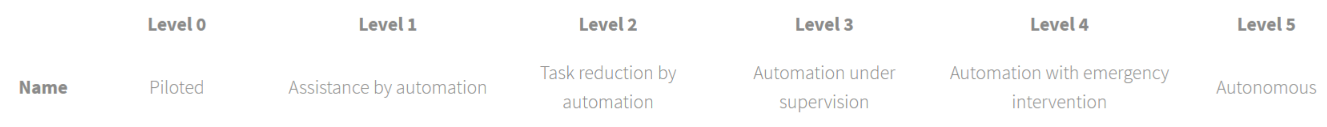

LOW
INCREASED
= MANNED AVIATION
RISK
OPEN
NO-PRE APPROVAL
LIMITATIONS : 25 kg; Visual line of sight (VLOS), height <120m; system of zones
3 SUB-CATEGORIES: fly over, close, far from people
CE MARKING allows for design requirements
SPECIFIC
Authorization by NAA based on specific operation risk assessment (SORA)
STANDARD SCENARIOS
Optional concept of approved operator with privilege
CERTIFIED
Certification of UAS and operator and licenced pilot (unless autonomous flight)
EASA accepts application in its present remit
Some systems (Datalink, Detect and Avoid, ...) may receive an independent approval

Figure 1.2: UAV regulations. Taken directly from [8].

.

In parallel, classifying an UAV as fully autonomous or not is inaccurate since as mentioned before there are examples of non-autonomous UAVs that posses some autonomous capabilities.

Six levels have recently been defined by JARUS (Joint Authorities For Rulemaking Of Unmanned Systems) to characterise the levels of autonomy of these operations:



	Level 0	Level 1	Level 2	Level 3	Level 4	Level 5
Name	Piloted	Assistance by automation	Task reduction by automation	Automation under supervision	Automation with emergency intervention	Autonomous

Figure 1.3: Automation Levels. Taken directly from [8].

Nonetheless, to navigate autonomously in a certain environment, any system, whether a car or in this instance an UAV, requires among other competences:

- Self-Localization , capability of determining its own position and orientation within a certain frame of reference which is achieved through equipped sensors such as Inertial Measurements Units (IMUs) and the Global Positioning System (GPS). In indoor spaces where GPS based navigation is not possible, motion based localization is used, where the Vicon Capture System that consists of a set of cameras stands out as a method of obtaining a reliable [9] estimation of the UAVs position within the environment. Other approaches to GPS denied operations have been made such as the use of phone sensors onboard small UAVs as a localization method has been explored in [10].

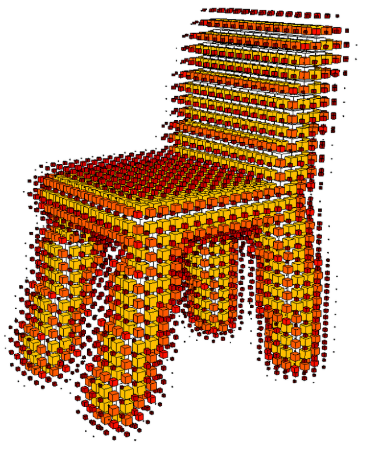


Figure 1.4: 3D voxel model of a chair. Taken directly from [11]

- Map-Building, capability of representing the map of the environment through information acquired by vision based and motion sensors such as depth cameras. A common approach is to use a grid of cubic volumes of equal size (voxels) to discretize the mapped area or using point clouds to model the occupied space obtained from range sensors. An upgrade to the first approach was realized in 2010 with the introduction of Octomaps [12], a probabilistic, flexible and compact 3D map representation based on octrees;

- Path Planning, for creating the necessary path configurations that allow hypothetical movement from a start point to a goal destination. It is common to try to attain the most optimum path in terms of fuel consumption, smoothness of the trajectory or mission-based specific goals such as scanning an area during a surveillance operation. It is a multi-layered subject where many variables are at play such as planning in environments where the obstacles are static or in dynamic environments where their position can change at any moment. Related to this there is also the question posed of having full access to the environment when planning in a known space but it is possible that the UAV must plan its navigation considering the lack of full information about the environment meaning it might need to create alternative paths if new information about the environment renders the current flying configurations unfeasible, which can happen in partially known or even unknow environments. There has been extensive work done in this field ranging from the development of lightweight algorithms capable of running onboard UAVs on low powered processors such as in [13], where an algorithm based on ant colony optimization (ACO) and artificial potential field was applied to a dynamic path planning problem with experimental results showing that a smoother path could be obtained compared to other similar algorithms and [14], where a hybrid genetic algorithm was developed and tested on literature benchmark maps with positive results to more complex algorithms whose effectiveness can only be achieved when running on ground stations such as in [15], a real-time path planning method was proposed by combing the improved Lyapunov Guidance Vector Field (LGVF), the Interfered Fluid Dynamical System (IFDS) and the strategy of varying receding-horizon optimization from Model Predictive Control (MPC) and [16] , where a real-time motion planning framework for kinodynamic robots was developed and tested on a quadrotor in an indoor environment;

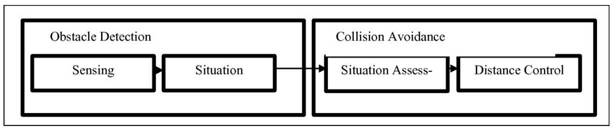


Figure 1.5: General Ilustration of Sense and Avoid Systems. Taken directly from [17]

- Sense and Avoid (S&A) technology, necessary for preventing collisions with dynamic obstacles. There is a need for automatically sense, detect and avoid obstacles in autonomous navigation, in particular in highly congested environments. This SAA systems must be reliable, robust and affordable for mass implementation. There are several components involved in these systems but they can be narrowed down to sensors, responsible for collecting information of the environment along the flying path, detection algorithms needed to process that information and avoidance protocols that can harness that information and avoid collisions along the course. A popular approach to to SAA is the use of electro optical sensors combined with radar/infrared sensor/thermal imaging/motion detector due to its smaller size, high refresh scan rate and high image resolution. However they can not estimate obstacle range and sometimes generate false detections. The recent integration of the LIDAR (Light detection and ranging), whose detection of objects is based on the calculation of the time necessary from light to travel to and from the obstacle has proven to be capable of achieving high precision and resolution in the measurement of distances and in particular when combining the information acquired through several onboard sensors in a process known as sensor fusion ilustrates the effectiveness of the LIDAR. There are several examples in Literature as in [18], [19] and [20] where 3D LIDAR was combined with a vision camera for detecting and identifying objects.

One additional major factor that cannot be overlooked when moving in outdoor environments is weather uncertainty which can act as a major bottleneck for a non-robust UAVs or operations which do not consider these conditions. Both rain and wind state can strongly determine the solution strategy for the UAV mission planning since they impact not only fuel consumption but also UAV control by causing it to drift in a certain direction. Work tailored to this constraints as been done in [21], where path planning of a fleet of delivery UAVs is adaptable to weather forecast changes, wind aware path planning of a quadcopter was developed in [22] and also [23] where it was tested in real UAV hardware at a wind speed of 5m/s and achieved significantly better results when compared to a non-constrained approach.

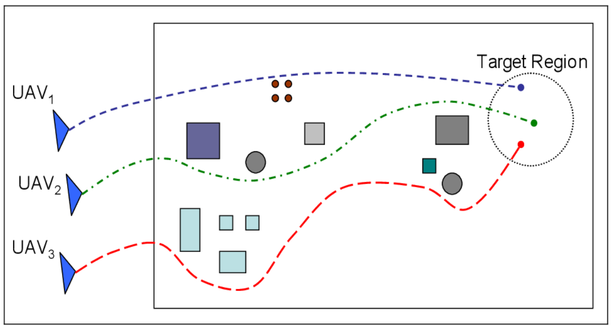


Figure 1.6: Multi-UAV Optimal Path Planning scenario. Taken directly from [24]

Until now, autonomous navigation was only mentioned in the context of a single UAV but depending on the need of the mission, we may want have autonomous navigation of a fleet of UAVs. This adds even more complexity to an already extense topic with the rise of challenging problems such as fleet formation control and now considering the problem of obstacle avoidance amongst the multiple fleet ”individuals”. There is already some work that tries to find solutions to this problem such as in [25] where a distributed trajectory generation strategy is proposed to control a group of UAVs, in [26] where a framework for multi-UAV path planning in GPS enabled areas showing that the developed method is effective and computational time is small for a large number of UAVs.

In summation, to move from one point to another without external Human intervention, an UAV must know its location as well as sense the environment around whether that would be static or dynamic objects and be empowered with methods of generating collision-free feasible trajectories.

As we can see there are several variables at play in achieving the aforementioned objective of having a fully autonomous UAV. It requires a global effort in the many fields of science and technology as well as synergy between the many companies, institutions and governments.

1.2 Objectives and Contributions

The complex nature of this problem is why the work that was carried out throughout this thesis was targeted not considering the full scope of the problem but reduced to target some integral parts of it. In this thesis a focus was placed on path planning and dynamic obstacle avoidance but it also touches briefly on the need of an accurate estimation of the world around. What is aimed to be accomplished here is the development of a framework that empowers an UAV with the capability of going from a start to a destination while simultaneously avoiding static and dynamic obstacles during its course.

To that effect there was the intended goal was divided in 3 major sub-goals:

- Develop a path planning algorithm capable of generating feasible trajectories for a generic quadrotor;
- Develop a collision avoidance method that allows safe avoidance of dynamic obstacles;
- Evaluate the performance and Validate the algorithms use through appropriate literature simulation environments;

The main contribution of this thesis is the special emphasis placed on developing a real-time path planning framework capable of running in real-time on less powerful computers that can fit onboard of a smaller UAV where payload weight is a severe bottleneck.

1.3 Thesis Outline

The thesis structure is organized as follows:

- **Chapter 1** consists of the topic that was studied, the context and motivation behind the choice and the objectives it aims to fulfill;
- **Chapter 2** goes further into the theoretical requirements of the proposed work and presents an overview of current state of the art implementations;
- **Chapter 3** outlines the Path Planning problem and presents a clear overview of the developed framework in the thesis. Several simulations related to the performance of the algorithms are shown;
- **Chapter 4** reveals the performance of the framework in physics simulations;
- **Chapter 5** presents overall conclusions on the developed framework and possible future work to be done;

Chapter 2

Background

Previously the several underlying topics involved in autonomous navigation were presented in a general way and the topics concerning the subject of this thesis were highlighted. This chapter contains a deeper overview of the relevant topics as well as literature review on state of the art works related to the subject of real-time path planning. Since the path planning is applied to a quadrotor, this chapter starts with a general overview on quadrotor dynamics and its importance in the overall autonoumous navigation problem. Next, the relevant topics of map building, path planning and dynamic obstacle avoidance will be covered in the context of the proposed work, identifying the several existing methods, associated drawbacks and advantages as well as providing literature review on state of the art work done using these methods.

2.1 Quadrotor Dynamics and Control

Figure 2.1: Quadcopter DJI Mavic Mini. Taken directly from [27]

Definition 2.1.1 (Quadrotor)**.** A type of UAV that consists of two pairs of counter-rotating rotors and propellers, located at the vertex of a square frame. It is capable to perform vertical take-off and landings (VTOL), similar to the typical helicopters. As shown in the figure above.

Understanding quadrotor dynamics plays a huge role in many steps of achieving autonomous navigation. These dynamics include several characteristics like nonlinearities, underactuation, parametric and non parametric uncertainties, measurement noise, among others which make the task of trajectory tracking more difficult.

Definition 2.1.2 (Underactuated Mechanical System - UMS)**.** A system is said to be an UMS if has more degrees of freedom (DOF) to be controlled than the number of independently controlled actuators exerting force or torque onto the system.

This property is of particular importance since it means that an UMS cannot follow an arbitrary trajectory.

The equations of motion and energy of a mechanical system are given by the Euler-Lagrange formulation by the following equations:

$$\frac{d}{dt}(\frac{\partial L}{\partial \dot{q}}) - \frac{\partial L}{\partial q} = F(q)u \tag{2.1}$$

$$L = T - V \tag{2.2}$$

where T and V represent the kinetic and potential energy of the system, respectively, $q \in \mathbb{R}^n$ is the configuration vector, $u \in \mathbb{R}^m$ is the actuator input vector and $F(q) \in \mathbb{R}^{n \times m}$ is a non-square matrix of external input on the system.

Which can be rewritten as:

$$D(q)q + C(q, \dot{q})\dot{q} + G(q) = F(q)u \tag{2.3}$$

where $D(q) \in \mathbb{R}^{n \times n}$ is the inertia matrix, $C(q, \dot{q}) \in \mathbb{R}^n$ represents both the centrifugal and coriolis terms, and G(q) denotes the gravity. If the rank of the input matrix, $rankF(q)$, is lower than the dimension of the configuration vector, $dim(q)$, the system is said to be underactuated.

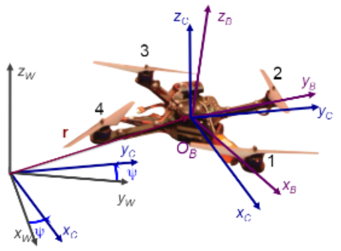


Figure 2.2: Quadrotor model body frame (subscript B) and earth fixed frame (subscript W). Taken directly from [28]

To describe the motion of a quadrotor we define two reference frames: earth inertial reference denoted by (W-frame) and body frame of reference denoted by (B-frame). The W-frame is an inertial right-hand reference symbolized by three axis (x_W, y_W, z_W). The Body frame with axis (x_B, y_B, z_B) is attached to the body of the quadrotor and has its origin at the center of gravity of the quadrotor. r is the position vector (x,y,z) and Θ the 3 Euler or roll, pitch and yaw angles (ϕ, θ, ψ) that correspond to the orientation of the body frame with respect to the earth fixed frame. The only external force applied to the quadrotor is the force of gravity, g, which is not represented but it is orientation is the reverse to the orientation of the z_W axis.

Using the previous notions, the rigid body equations as shown in [28] can be written as:

$$
\begin{aligned}
&\boldsymbol{q} = \begin{bmatrix} \boldsymbol{\xi} & \boldsymbol{\Theta} \end{bmatrix} = \begin{bmatrix} x & y & z & \phi & \theta & \psi \end{bmatrix}\\
&D = \begin{bmatrix} m \cdot \boldsymbol{I}_{3\times 3} & 0_{3\times 3} \\ 0_{3\times 3} & \boldsymbol{J} \end{bmatrix}\\
&\boldsymbol{C}(q,\dot{q}) = \begin{bmatrix} 0_{3\times 1} \\ \boldsymbol{\omega} \times \boldsymbol{J}\boldsymbol{\omega} \end{bmatrix}\\
&\boldsymbol{G}(q) = \begin{bmatrix} -g \cdot \boldsymbol{e}_3 \\ 0_{3\times 1} \end{bmatrix}\\
&\boldsymbol{F}(q) = \begin{bmatrix} \boldsymbol{R} \cdot e_3 & 0_{3\times 3} \\ 0_{3\times 1} & \boldsymbol{I}_{3\times 3} \end{bmatrix}\\
&u = \begin{bmatrix} T_z \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ -l & 0 & l & 0 \\ 0 & l & 0 & -l \\ d & -d & d & -d \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{bmatrix}\\
&\qquad f_i = k \cdot w_i^2
\end{aligned}
$$

Where (m) is the quadrotor mass, (J) is the quadrotor inertia matrix with respect to B-frame, ω is the angular velocity with respect to the B-frame, $(e_3) = [0,0,1]^T$ is the third vector of the canonical basis of $\mathbb{R}^3$, $[T_z, T_\phi, T_\theta, T_\psi]$ are the inputs of the quadrotor representing the collective force, roll torque, pitch torque, yaw torque, respectively, (ω_i) is the speed of the ith motor, (k) is the thrust factor, (d) is the drag factor, (l) is the distance between the center of the quadrotor and the center of a propeller, (R) is the rotation matrix needed to map the orientation of a vector from B-frame to W-frame.
Since the rank of the input matrix ($rankF(q) = 4$) is lower than the dimension of the configuration vector ($dim(q) = 6$) the system is said to be underactuated and cannot follow an arbitrary trajectory.

However, in [28] the authors prove that a quadrotor is a differential flat system and the inputs of the system can be written as functions of the derivatives of the position.

This property can be defined as:

Definition 2.1.3 (Diferentially Flat System)**.** A system in which we can find a set of outputs (equal in number to the number of inputs) such that we can express all states and inputs in terms of those outputs and their derivatives.

In a formal way this means that in a non-linear system such as the quadrotor system:

$$\dot{q} = f(q,u) \tag{2.4}$$

$$j = h(q) \tag{2.5}$$

where q represents the configuration vector and u, the control input, is differentially flat if it is possible to find outputs k:

$$k = \zeta(q,u,\dot{u},....,z^l) \tag{2.6}$$

such that,

$$q = q(k,\dot{k},....,k^l) \tag{2.7}$$

$$u = u(k,\dot{k},....,k^l) \tag{2.8}$$

q are the tracking outputs and k the flat outputs.

The authors of [28] choose as flat outputs:

$$q = (x, y, z, \psi) \tag{2.9}$$

and proved they could be written as q and it is derivatives. This result is of particular importance since the method utilized in computing feasible trajectories in this work makes use of it as it will be detailed ahead.

2.2 Map Building

In order to navigate between different parts of the environment, mobile robots need a map of the environment, either being able to construct it during flight or using an already built map. Map building is the branch of autonomous navigation concerned with providing methods to achieve this.

UAVs in particular require a 3D model of their surroundings since unlike other ground robots they are able to move in the air. This computational representation of the environment is of particular importance in dense and cluttered environments where this is an immediate need for a high degree of precision in UAV maneuverability. There is also the problem concerning how to acquire and process the information of the environment to build these high fidelity models. Both problems are deeply interlinked and there has been extensive research made in the past.

Robots can acquire information for map building in two ways:

- idiothetic which correlates to internal kinesthetic information such as tracking the velocity evolution throughout the path;
- allothetic that translate to external visual information acquired through robot's sensors such as camera, Lidar or depth sensors.

The most used representation methods are direct, topological and grid-based or metric.

In the **direct representation** measurements from the environment are directly used to build a representation of it by aggregating them (similar to a cloud of points) without considering its characteristics. This direct use of point cloud is used in [29] but presents some downsides such as the inability to model free space and unknown areas, account for sensor noise (there is a need for high precision sensors), model dynamic objects as well as complex and irregular shaped objects also cannot be dealt with. It computationally costly since the increase in the number of measurements from the environment severely acts as a bottleneck.

Topological representation uses a graph-like structure, where nodes usually correspond to places and edges correspond to paths between the places. This representation can be used efficiently for path planning and localization in large-scale environments.

Grid-based representation stores the geometric properties of the environment allowing for accurate and more refined descriptions. They can be derived from grid-based approximations or geometric primitives. Occupancy grid model is an example in which maps are discretized in a grid composed of cells that contain an occupancy probability value. This allows the environment to be fully classified, whether occupied by an obstacle, free of obstacle, or not mapped. As examples we have 2.5D maps or Elevation maps similar to a basic 2D occupancy grid but now each cell stores the measured height instead of the occupancy probability. They are used to map non-flat surfaces. Multi-level surface maps [30] allows to store multiple surfaces in each cell of the grid overcoming the disadvantages of elevation maps of not being able to represent vertical structures or multiple levels. A natural evolution is a 3D representation of the surrounding space that in which the space is partitioned into a 3D matrix of cubic voxels. However to efficiently store this multi-resolution maps there was a need for a specific data structure to be developed. Octrees [31] provided the solution for this problem. In this hierarchical data structure, each node stores the information of a cubic voxel in the occupancy grid. Each voxel is subdivided into eight smaller voxels until a given minimum voxel size is reached and this minimum voxel size determines the resolution of the octree. In this method if a certain measured volume within an environment is occupied then the node corresponding to that volume is set to occupied. Free volumes can be represented as free nodes and any uninitialized volume or subvolume is set as unknown. This allows for a reduction in the number of subvolumes of the environment that actually need to be explicitly represented in the octree since for example if all the children of a node are occupied they can be pruned. However, octree based approaches pose additional problems such as map update, overconfidence and compression.

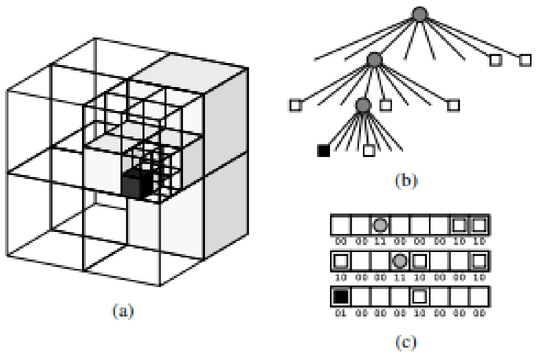


Figure 2.3: Example of an octree storing free (shaded white) and occupied (black) cells (a), the corresponding tree representation (b), and the corresponding bitstream for compact storage in a file (c). The complete octree structure can be stored using only six bytes, 2 bits per child of a node.

Octomaps [12] implement a 3D occupancy grid mapping approach based on octrees. It performs a probabilistic occupancy estimation to keep the map updatable and capable of dealing with sensor noise. A lossless compression method is applied to keep the maps compact in memory. Due to its adaptive nature and reliability they have been widely featured in several literature works as the chosen environment modelling tool such as in [32] and [33] with results corroborating the effectiveness of the proposed method.

2.3 Path Planning

In this section we present a more extensive overview on the path planning problem, highlighting the advantages and drawbacks of the several existing methods, in particular of those employed on this work.

Path planning can be defined as the collection of techniques through which it possible to generate a collision-free path between a start and goal state. In a formal manner, it can be viewed as a search in a configuration space, C, in which each $q \in C$ represents the position and/or orientation of a robot in the environment. Free space, C_{free} , contains all possible configurations that avoid collisions with obstacles.

In [34] path planning is classified in terms of it overall reachability as global or local.

In **Global Planning** we use *a priori* information about the environment to find the best collision-free path that contains the start and goal states.

In **Local Planning** we recalculate the initial plan to avoid possible collisions if there is new information about the environment (whether that is a new static obstacle or a dynamic one). This type of planning is also known as trajectory or manoeuvre generation.

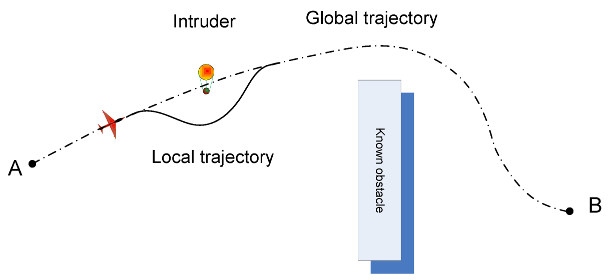


Figure 2.4: Local path (green) generated to correct collision in tracking of global path (red). Taken directly from [35].

These two techniques are commonly applied together as is an example the work in [36], where in the first stage the objective is to find a collision free feasible path from start to goal that respects kinematic constraints such as velocity and acceleration, while in the local planning step path smoothing is applied to deal with dynamic constrains. This idea of cooperation between these two methods is of particular importance to the work carried out here as it will be clear ahead.

2.3.1 Global Planning

There are two main global path planning techniques which are Graph Search and Biologically inspired algorithms. In **Graph Search** algorithms the search space is represented by a set of nodes on a weighted graph. The robots configuration are the graph nodes with the edges between each node being the cost of moving between the nodes (occupied space can be represented by nodes with higher cost).

One widely used example of this are **Probabilistic Roadmaps (PRMs)** in which the first stage is randomly sampling nodes from the search space and tries to find the shortest sequence of collision-free configurations between the start and goal states using Dijkstra algorithm (the process is illustrated in the figure below). This algorithm divides the nodes in the weighted graph into two groups, the first contain the nodes with the shortest path determined and the second contains nodes not explored which are gradually added to the first according to the order of increasing length of the path until all are added and the algorithm stops. During the sampling stage, the algorithm identifies in a heuristic approach difficult regions in the free space and adds more configurations around those areas, increasing the density in that region which helps particularly in cluttered environments. This method performs well in high-dimensional state spaces but fall short planning when the environment is not known a priori. It a probabilisticaly complete algorithm when considering that failure rate is zero if the number of sampled nodes in the graph is infinite. There are several versions of this method such as the Lazy PRM [37] where the validity of an edge is only checked if it part of a candidate solution.

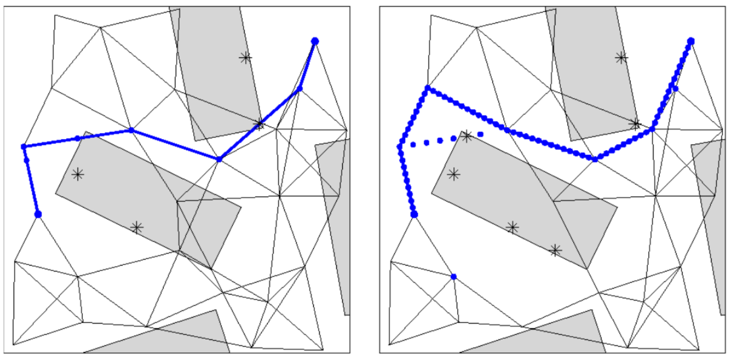


Figure 2.5: After a few iterations a sequence of feasible nodes is found. Collision checking reveals it unfeasible and it removed from the graph. Eventually a collision free path is found. Taken directly from [38].

Another algorithm used is the **A*** algorithm [39] which is based on the Dijkstra's algorithm but the cost of moving between nodes is now given not only by the motion cost of reaching its position but also by an heuristic cost which is a cost estimation of reaching the goal node from that node. The process chooses the node with the minimum added cost reducing the number of processed cells in an environment as long as the chosen heuristic is admissible. The major disadvantage of this method is that it might require a lot of memory usage and large computation times for larger maps where thousands of states have to be stored. **Dynamic A*** [40] allows for application in dynamic environments by allowing the edges cost to be dynamically increased or decreased.

Rapid Exploring Random Tree (RRT) [41] is one of the most used methods in literature which works in a similar way to PRM. Unlike PRM where a collision-free graph is computed beforehand, here a graph is initialized only with the initial state as a node and at each step a random sample from the search space is attempted to be connected to the nearest node on the graph. If the connection is successful, meaning that the path between nodes is collision free and the sampled node was part of the free space then the node is added to the graph and the process repeats itself until the goal state is added to the graph. There is a distance limit which is called a growth factor in the step of adding a sampled node to the graph. Aside from this, the search in the space is biased towards the goal state, the more biased the greedier the algorithm becomes. Just like PRM it probabilistically complete and does not guarantee asymptotic optimality of the generated path. The notion of optimality is with respect to a specified optimization objective which commonly is path length. Since its development, many extensions of this algorithm have been proposed in order to improve upon it. One of the first approaches was the **RRT-Connect** [42], which expands on the basic RRT by initializing two graphs, one beginning at the start node and the other at the goal node and it grows both graphs until it possible to connect them. The main advantage behind this method is a faster computation time when compared to the basic RRT. **RRT*** [43] was developed in 2010 and while it shares most basic principles of operation with basic RRT but unlike it, strives for asymptotic optimally (visible in the figure below) by applying two different concepts which are called near neighbor search and rewiring tree operations. The first finds the best parent node for the new node to be added to the graph, while the second feature locally optimizes every node in the graph allowing the algorithm to converge asymptotically (provided the number of iterations is sufficient). However, despite allowing for a minimum-cost path when compared to the basic RRT, there is a bigger cost of execution time especially in large environments.

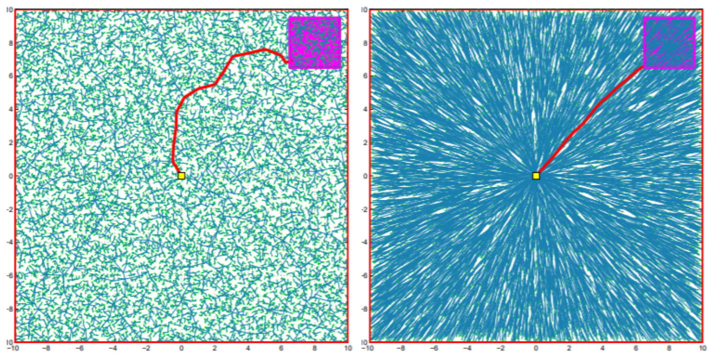


Figure 2.6: Comparison between the length of paths generated by RRT and RRT*. Taken directly from [44].

To improve the convergence rate of this method, in [45] a new version was proposed called **Informed RRT*** where an heuristic informed search strategy is implemented in RRT* in order to shrink the planning problem to subsets after finding an initial solution. This process is detailed in figure 2.7. Compared to RRT*, it shows a significant better convergence rate. This sampling based method has been applied successfully in a UAV path planning system for dynamic obstacle avoidance in a cluttered 3D environment [46], where the authors took advantage of the fast convergence of this method (relative to the basic

RRT*) coupled with real-time feedback of the UAV and obstacles poses obtained from motion tracking cameras and showed the ability to handle multiple obstacles in real-time.

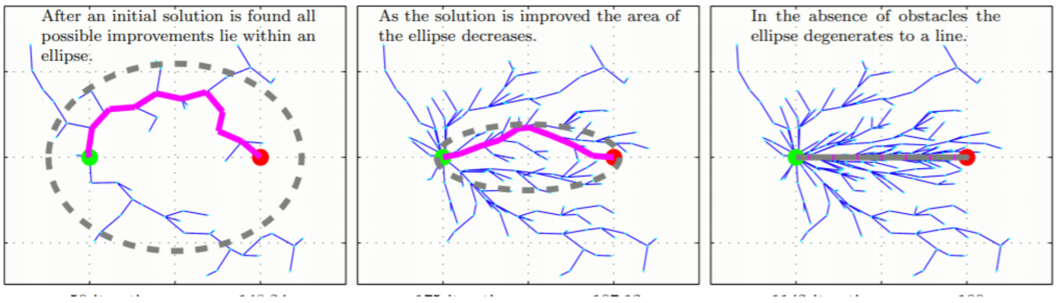

After an initial solution is found all possible improvements lie within an ellipse.
As the solution is improved the area of the ellipse decreases.
In the absence of obstacles the ellipse degenerates to a line.

Figure 2.7: Informed RRT* converging to within machine zero of the optimum in the absence of obstacles, from left to right the number of iterations is increased showing the reduction in size of the sampling domain. Taken directly from [47].

Another advancement has been made upon the RRT* to turn this method fully capable of being directly applied to dynamic environments. The **Real-Time RRT* (RT-RRT*)** [48] was developed in 2015 with that in mind and allows this functionality by an online rewiring of the path which can be summarized in the figure below.

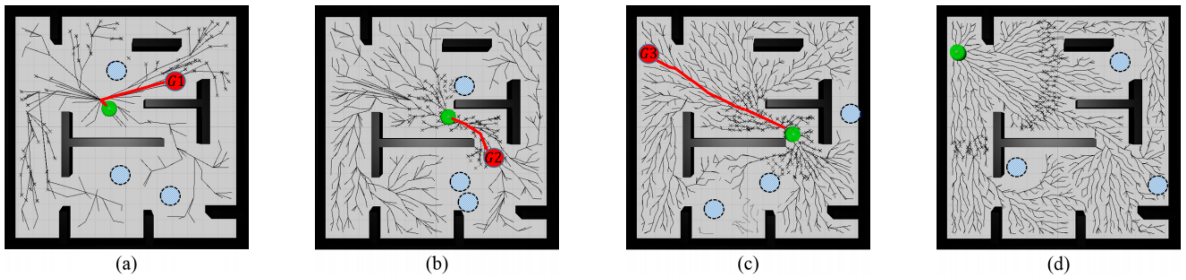


Figure 2.8: Blue circles denote dynamic obstacles, green circle is the autonomous robot, red line represents the path to the goal point which is corresponds to the red circle. The black lines denote the paths store in the tree. (a) a path to the goal point is found. (b) and (c) the goal point is changed during movement. In both cases since the tree root moves with the agent and the tree covers most of the environment, the paths to the changed goals are returned quickly. In (d) the agent has reached the goal point and rewiring of the nodes based on the current location of the tree root has generated the minimum length paths to others nodes of the tree. The crossed points denote the Rewiring Circle. Taken directly from [48].

Despite the added benefits, it requires a high computational capacity since it is required that the tree is stored fully at each instance in time and it only works in bounded environments and the rewiring is only efficient in smaller environments.

Biologically inspired algorithms that mimic biological behaviour have been applied to path planning problems. They can be divided in **evolutionary algorithms** and **application of neural networks algorithms**. The first tries to mimic genetic systems by specifying an initial population of vehicle paths and applying evolutionary processes (i.e mutation, propagation among others) to them. A cost function is used to evaluate against vehicle goals and constraints. According to their performance, paths can be kept or discarded from the population which leads to paths who are more compliant to the objectives.

Ant Colony Optimization (ACO) [49] is one example of an evolutionary algorithm that based on the behaviour of ants when trying to find a path between the colony and food sources. The basis behind ACO algorithm is illustrated in the figure below.

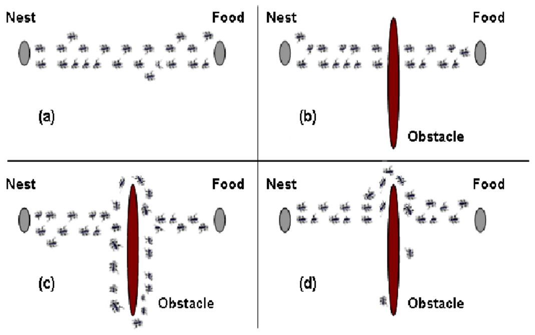


Figure 2.9: Natural behaviour of an ant colony. In (a) ants follow a path from nest to the food source; (b) An obstacle is placed on the path and ants choose with equal probability to turn left or right; (c) more pheromones are placed on the short path; (d) most ants have chosen the shortest path. Taken directly from [50].

These algorithms have been tested in uav path planning in [51] where the method was applied to a minimum time search for a target involving a fleet of UAVs and in [52] where one of the main drawbacks of the method related to premature sub-optimal convergence was targeted by expanding the algorithm to deal with cooperation between multiple colonies trying to find an optimal solution. This approach wields better convergence and reduces the probability of an UAV being trapped in the *local minima* when compared to the single ACO method. However, the time to converge is still an uncertainty as well as computation time has shown itself prohibitive making unsuitable for online applications.

A **Neural Network (NN)** can be summarized in the following figure:

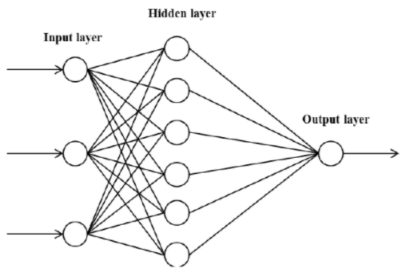


Figure 2.10: Basic Neural Network structure. Taken directly from [53].

The most basic neural network contains three layers of interconnected nodes (input, hidden and output layer) where each node is called a *perceptron* and receives an input similar to a multiple linear regression and maps it into a certain value through an activation function. The idea behind this is to use this network to find underlying relationships between data in a similar way to which our brain operates. An example could be we have a set of images each containing a dog of a certain race and we want to identify in each image which race is present, in this case we feed the image to the neural network that has to be able to pick the certain race while ignoring all the remaining. This is a simple example but this is the basic logic behind neural networks. Recently due to advancements in deep learning, neural networks have been applied to path planning problems such as in [54] where a method called Deep-Sarsa was used to tackle the problem. The proposed algorithm (Deep-Sarsa) is based on a combination between the Depth-First Search (a graph searching algorithm) and Sarsa (a delayed reinforcement learning algorithm). This method gains information and rewards from the environment and helps UAV to avoid dynamic obstacles as well as finds a path to a target based on a deep neural network. Tested in a ROS-Gazebo environment and it showed impressive results when compared to other algorithms, presenting itself as the first time autonomous navigation was achieved using this neural network based method.

A state of the art Convolutional Neural Network (CNN) model was employed to solve the autonomous navigation of an UAV, equiped with a monocular camera, in a previously unknown indoor environment [55]. The neural network uses the video feed from the camera as an input in order to decide the appropriate maneuver. The training of the neural network is done over a dataset of various images from indoor corridor environments and the authors achieved efficient results in indoor corridor scenarios.

The topic of neural networks applied to autonomous navigation is still at the early stages of development but it has shown great promise for the future.

2.3.2 Local Planning

It was mentioned earlier in this chapter that local planning could be seen effectively as a complement to global planning since it deals with dynamic obstacle avoidance and also planning in changing environments whether that is because the robot is learning about it as it moves or due to changes in the environment itself. We also alluded to the possibility where local planning was employed to correct locally previously global paths to account for dynamic constraints. Due to the nature of the methodology employed in this work, we will mostly refer to local path planning in the context of the first application of this type of planning since a large majority of obstacle avoidance methods already take into account other feasibility constraints.

Potential field (PF) based algorithms have been applied in local planning as part of reactive obstacle avoidance. These algorithms are called **Artificial Potential Field (APF)** [56]. Potential fields model the world by a force field with obstacles acting as repulsive poles and defined goals as attractive poles therefore there is an attractive force that increases towards the goal and repulsive forces that increase towards obstacles. In APF the robot is modeled as a charged particle moving through this world where these potential fields can ”drive” the robot away from static and dynamic obstacles. These algorithms are low in computational requirements and provide smooth paths while not offering guarantees of optimality or completeness since one of the main drawbacks is that they tend to become trapped in *local minima*, preventing the robot from reaching its goal.

Nonetheless they have been widely used for obstacle avoidance problems involving UAVs such as in [58] and [59]. Many works have tried to improve upon the initial concept of APF. One example is [60] where an improved APF algorithm was proposed for autonomous obstacle avoidance planning of a quadrotor. The novelty introduced in this approach is that the algorithm can now calculate in real-time

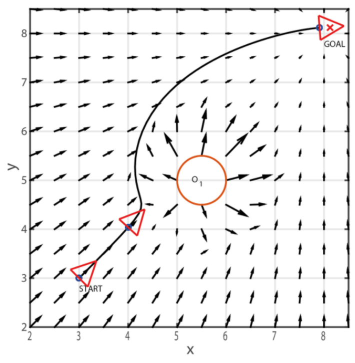


Figure 2.11: APF environment model. The arrows pointing towards the goal simulate an attractive force while the arrows starting inside the obstacle simulate a repulsive force. Taken directly from [57]

the different potential field at each instance. The authors showed through simulation results that the mitigated many of the shortcomings of the basic APF such as the *local minima* issue.

A Markov decision process (MDP) is a discrete-time dynamic stochastic control process where the outcome of a system depends on the current state and an action. Formally it can be defined by a 4 tuple (S, A, P_a, R_a):

- S is the state space;
- A is the action space (set of actions available from state s);
- $P_a(s, s')$ is the probability of transitioning from state s at time t to state s' at time t+1 having taken action a;
- $R_a(s, s')$ is the expected immediate reward due to transitioning from state s at time t to state s' at time t+1 having taken action a;

In a MDP, the objective is to select the policy π (a function that specifies the action $\pi(s)$ that the decision maker will choose when in state s), also known as optimal policy, that maximizes a function of the cumulative rewards over a potentially infinite horizon.

In a formal way, this can be written as:

$$E[\sum_{t=0}^{+\infty} \gamma^t R_{at}(s_t, s_{t+1})] \tag{2.10}$$

γ is the discount factor and its value is defined between 0 and 1. The lower the value the more it benefits the taking of earlier actions. A MDP might gave multiple optimal policies.
This method can be applied to dynamic obstacle avoidance problem since it can be formulated as a MDP by including in the state s the positions and velocities of dynamic obstacles (and obviously the UAV itself) in order to find the optimal policy which could be in this particular problem aerial maneuvers to avoid the obstacles or a feasible collision-free trajectory. It depends entirely on how the problem would be formulated in a MDP environment. MDPs are solved computationally through dynamic programming algorithms which makes the real-time application of these techniques an issue when dealing with large

state spaces. MDP has been employed in the dynamic obstacle avoidance problem in [61] where it was used for optimal decision making when faced with a possible dynamic intruder during periods of communication latency between a remote operator and the UAV. The dynamic avoidance problem was also modeled as a MDP in [62] in which this method coupled with a deep reinforcement learning technique allowed for optimal decision strategies.

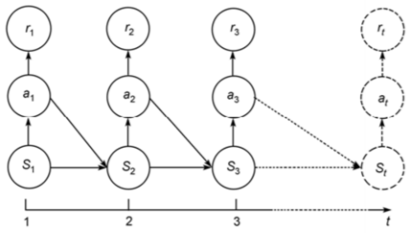


Figure 2.12: Diagram of a MDP. Taken directly from [63]

Maneuvre Automata (MA) are a collection of maneuver generation techniques that work by creating a continuous trajectory over predefined waypoints. They are generated offline and can be either trim primitives, if the maneuver is of constant velocity and turn rates comply with dynamic constraints, or motion primitives when the maneuver is feasible within UAV kinodynamic constraints. the main benefit behind this primitives is that they can be combined through a trim primitive of finite length [] whose own length can be null. This means that its possible to generate a smooth trajectory around an incoming object by concatenating this primitives through a collision-free set of waypoints. While not prevalent in current literature, maneuver based approaches to obstacle avoidance were explored in [64], where primitive maneuvers were integrated with flight controllers to produce dynamically feasible trajectories in low dimensional search space with the intent of being applicable to cluttered environments.

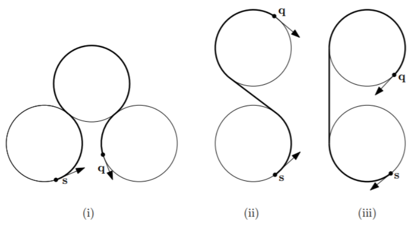


Figure 2.13: Types of Dubins Curves. Taken directly from [65]

Similarly to MA, **Dubins Curves (DC)** are another method to generate smooth trajectories. Unlike MA, this technique generates optimal geometric paths between two points (where at each we assume tangents) under a curvature constraint. The constructed paths resut of a combination of curves of maximum curvature and/or straight lines tangential to the curves. This also works under the assumption that the robot can transition between these curves and straight lines effectively using external guidance or some sort of control method. However, Dubins Curves do not account for obstacles and might produce paths with abrupt curvature discontinuities and direction turns. Nonetheless they can be used in conjunction with other methodologies to solve the collision avoidance problem. Furthermore, they are computationally inexpensive and relaxing the steering constraint might mitigate the problem associated with discontinuities. In [66], RRT was combined with 3D Dubins Curve for path planning, where at each step the branch of the tree is generated along Dubins Curve. Collision is checked in real-time and the tree is updated accordingly until the goal is reached. The authors say the algorithm was able to be employed in real time to generate reliable paths.

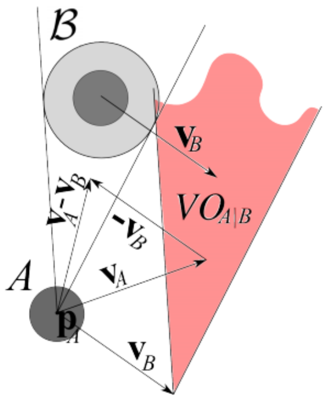


Figure 2.14: The velocity obstacle $V_{OA|B}$ for robot A relative to dynamic obstacle B. The cone of velocities that would lead to a collision in the static case is translated by V_B. Since the velocity v_A is inside the velocity obstacle it leads to a collision. Taken directly from [67]

Velocity Obstacle (VO) based methods are a common approach to dynamic obstacle avoidance and are based on the concept of the possible velocities of a robot that would put it in collision with an obstacle (static or dynamic) at a future point in time. Although, initially developed for legged robots, they have been applied to UAVs due to being a ”light” method in terms of computation which means it is able to be employed for real-time avoidance. In [67], VO is defined geometrically for two moving disk-shaped robots A and B (of radius r_a and r_b, respectively) as $VO_{A|B}$, the velocity of A induced by B for which A would be in collision with B at some point in the future. The Minkowsky sum β can be defined as the disk centered at B with radius equal to the sum of r_a and r_b. A collision cone C can be defined if B is static as the set of rays shot from A that intersect the boundary of β, that represents the velocities for A that lead to a collision with B. To obtain the actual VO we translate C by the velocity of B, V_b.

Formally this can be written as:

$$VO_{A|B} = \{v|\exists t > 0 :: p_a + t(V - V_b) \in \beta\} \tag{2.11}$$

p_a denotes the center of disk A.

From this it would result in a set of possible velocities that would put the robot out of collision. However, not all of the possible velocities may be dynamically feasible so they must be checked with the admissible velocities of the robot that respect the kinematic and dynamic constraints. Since it is a very straightforward approach, it has been employed to deal with dynamic obstacle avoidance in several works such as in [68] where the conflict situation with obstacles is represented by avoidance planes to aid the maneuver decision, in [69] the standard procedure of Sense, Detect and Avoid (SDA) is replaced by Sense and Avoid with VO by substituting the detection part with a probability of collision map based on obstacle’s position and velocity (proving itself effective in emulating SDA methodology) and [70], VO is applied to UAVs with limited sensing capability (VO requires knowledge of obstacles shape, size, position and velocity) by deriving the necessary obstacle parameters from sensor readings.

The obstacle avoidance problem can be formulated as an **optimization problem**. In a formal way an optimization problem can be defined as:

$$\begin{aligned} &\underset{x\in\mathbb{R}^n}{\text{minimize}} && f(x) \\ &\underset{m\geq 0, p\geq 0}{\text{subject to}} && g_i(x) \leq 0,\ i = 1, \ldots, m \quad \text{and} \quad h_j(x) = 0,\ j = 1, \ldots, p \end{aligned} \tag{2.12}$$

where f is the objective function to be minimized, $g_i(x)$ inequality constraints and $h_j(x)$ equality constraints. For m = p = 0 it is called an unconstrained optimization problem. The solution to this problem is the n-variable vector $x_{solution}$, part of the domain of the n-variable design vector x, that minimizes the f and respects all imposed constraints. This solution can be **global** if its it leads to the absolute minimum value of f for all domain of x or **local** if this minimum value is relative only to a certain neighborhood of the solution $x_{solution}$. This characteristic that depends upon the convexity of the constraints, more specifically it is entirely dependent on how the constraints are formulated. In case of convex optimization, there is only one optimal solution which is global or there is no feasible solution. Otherwise, in non-convex optimization, we might have several local optimal solutions and determining if there is a single global solution (or no solution) among the multiple solutions is computationally expensive.

In a general form, the UAV obstacle avoidance problem can be formulated as a constraint in an **optimization problem** such as the idea of ensuring at all times a minimum distance from an obstacle (static or dynamic). This constraint could be posed as $p_{UAV} - p_{obstacle} \geq d$ where each term represents the position of the UAV, the position of an obstacle and the minimum defined distance (this constraint would have to be scaled for all possible obstacles and for all time instants). A commonly used method associated with the previous formulation is the use of signed distance which is defined as the distance between a given point in space and the boundary of an obstacle (only applicable for convex obstacles). Its value is negative if the point is inside the obstacle, null if it lays on the surface or positive it is outside.

Generally, the avoidance problem is only a part of the construction of an optimization problem since UAV model dynamics and kinematics (maximum velocity, acceleration, attitude rate among others) are also constraints to have into account when generating trajectories. Therefore, this is a type of planning that can be applied as an overall solution to the path planning problem. We can still make a distinction between global planning and local planning within the method but the generated path is a solution simultaneously to both levels of planning. With this being said whenever we talk about a different application of this method we will be more focused on the formulation of the obstacle avoidance, without disregarding the full scope of the method.

Model predictive control (MPC) is an example of an optimization-based algorithm. It consists of a nonlinear control technique that uses a dynamic model of a plant and past control history to optimize future states (which correspond to all the variables like position, velocity, acceleration and attitude that are used to describe the robot system) over a defined time period and subject to a cost function, therefore the trajectory optimization problem is solved through feedback control. Although initially developed for chemical processes with slow dynamics, in robotic systems, this method allows for the use of system dynamics and kinematics to generate feasible trajectories. For an UAV, the dynamic model corresponds to the UAV and the response to external and internal forces, while the cost function is generally a combination of constraints like distance, velocity or acceleration to be minimized. A recent use of this method have been made in [71], where the dynamics of multiple obstacles are anticipated through the use of active set algorithms that only consider obstacles that affect the UAV, reducing the computational burden of the algorithm. The authors also employed orientable ellipsoids to model the obstacles in order to improve the smoothness of avoidance maneuvers. Results in two real-time implementations showed the algorithms capability to avoid dynamic obstacles. In a similar way in [72], the authors also employ a method to predict future trajectories of obstacles and these are fed into the MPC as an additional constraint. The developed framework showed efficient results in tests, delivering fast and computationally stable solutions in dynamic environments.

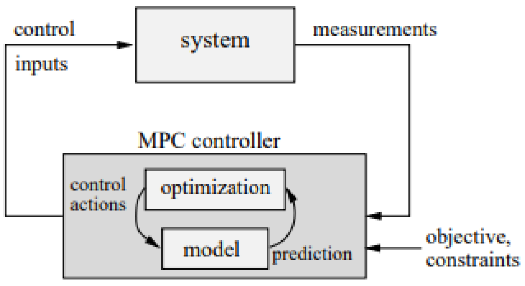


Figure 2.15: Model Predictive Control illustration. Taken directly from [63]

The main challenge of optimization based planning deals with the formulation of the collision avoidance constraints. Moreover, these constraints are known to be non-convex and therefore a heavy burden in computational efficiency. One approach uses a formulation based on integer values. It is called **Mixed Integer Linear Programming (MILP)** in which all formulated constraints and objective function are linear and some design variables are restricted to discrete/integer values. This reasoning falls behind the idea of taking the path planning problem as a linear optimal control problem.

In [73], the path planning problem is modelled as a MILP and the collision avoidance is incorporated as a constraint, specifically a minimum distance constraint and also to ensure additional safety it was also defined a safety margin around the obstacles. It has to be said however an approximation through linear constraints is flawed since it is difficult to establish the approximation error.

Due to this drawback, there have been attempts to develop techniques to allow for improving effiency in solving non-linear non-convex optimization problems of UAVs while adequately representing the context which UAV dynamics pose.

In [74], path planning is modeled as a non-linear non-convex optimization problem. Non-convex constraints are then approximated by convex parts in a series of sequential convex programming problems in order to have quicker convergence and stability. This proposed algorithm achieves global convergence to a Karush-Kuhn-Tucker (KKT) point of the original non-convex problem. A different approach is taken in [75] by implementing a lossless convexification method into the non-convex programming problem. By doing so, the authors are able to improve efficiency in finding an optimal solution to the problem without having the implicit need of solving the non-convex programming problem directly. A recent method developed in [76] turned non-differentiable collision avoidance constraints into smooth nonlinear constraints. This formulation is applicable to general obstacles (static or dynamic) that can be represented as the union of convex sets. By doing so the authors avoid introducing approximations. The authors show trough numerical experiments that the developed formulation allowed for real-time optimization-based planning.

2.4 Remarks

In this section, we presented an overarching analysis on the subjects deeply connected to autonomous navigation. As stated previously, it is a challenging task that requires nonetheless a path planning strategy and a collision avoidance method. A distinction was between global and local planning with the latter being more closely associated to the collision avoidance necessity. Several techniques within each path planning component were illustrated as well as the benefits and drawbacks associated. Due to the scope of this thesis, the research on each topic was kept concise. With this being said, the focus was put on RRT derived algorithms and optimization-based methods to solve the real-time implementation of autonomous capabilities on a quadrotor. The next chapter contains an overview of the proposed solution to the initially outlined problem.

Chapter 3

Implementation

This chapter starts with an overall description of the proposed framework to solve the path planning problem and it is followed by the actual implementation of each of the framework's components as well as the interactions between them that allow the framework to function as a whole system. There will be a clear reasoning behind each choice made. As stated in the thesis objectives, the developed framework must be capable of running onboard an UAV (particularly a generic quadrotor) and empower this system to navigate autonomously in a dynamic environment where collisions with both static and dynamic obstacles must be avoided. The generated path must be feasible in the context of quadrotor dynamics (it is a necessity to have a good tracking of the trajectory using low-level controllers) and strive when possible for global optimality (evaluated in terms of distance). For that the agent (quadrotor) needs to find the optimal sequence of actions to take in order to travel from a source to a given destination.

The framework and accompanied simulations were built from open source material and software available (code implemented in C++ and tested on the Linux environment).

3.1 Proposed Solution

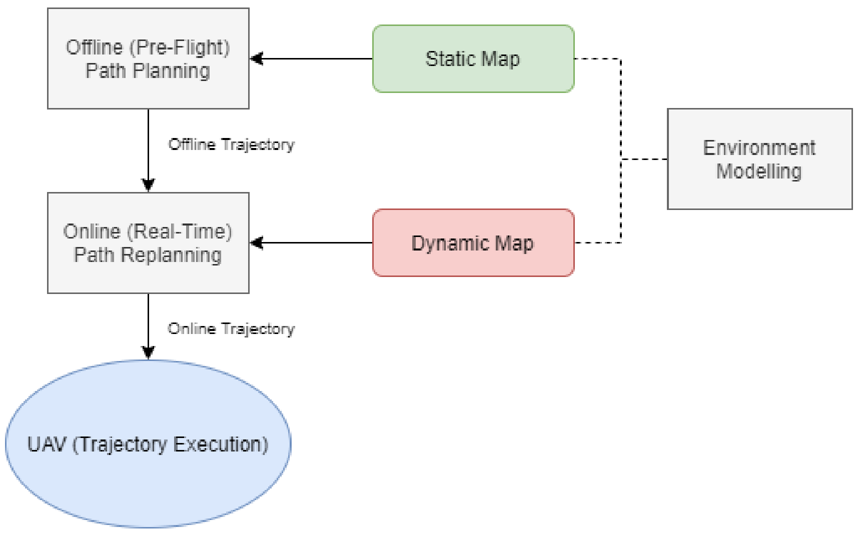

Offline (Pre-Flight) Path Planning
Static Map
Offline Trajectory
Environment Modelling
Online (Real-Time) Path Replanning
Dynamic Map
Online Trajectory
UAV (Trajectory Execution)

Figure 3.1: Simplified architecture of proposed implementation

This sequence of adequate actions comprises the developed framework. It consists of a two-phased planning approach in accordance to the division presented in the previous chapter. To that effect we have:

- **Offline/Pre-Flight Path Planning** In the first phase, from the start position and with full knowledge of the current state of the environment, a global path planning strategy is employed to generate a feasible collision-free trajectory to lead the UAV to the chosen destination. This trajectory is generated under an optimality guarantee of distance.

- **Online/Real-time Path Replanning** From the instance in which the UAV starts tracking the desired trajectory, the dynamic environment is checked periodically for possible collisions with moving obstacles. In case of a collision, replanning protocols are activated in order to generate a new collision-free trajectory towards the destination. At this point the optimality guarantee is sacrificed to ensure collision avoidance.

The offline path planning is not time constrained since it is executed before the mission starts and the UAV takes off. This effectively means that the first generated trajectory should try to come as close as possible to an optimal solution. To that effect a combination of a sampling based method with an optimization technique based on the work done in [77] that seeks to minimize the fourth derivative of position (snap) is employed. The used sampler is the Informed RRT* and generates an optimal collision-free path in terms of length from start to goal. This path is then converted into a trajectory by feeding the path waypoints to an algorithm based on the above mentioned technique. The generated polynomial trajectories are smooth and show continuous motion between the different UAV poses. The algorithm uses an unconstrained quadratic program to solve the optimization problem and allows for efficient real-time solutions.

So far, path and trajectory have been used interchangeably but an important distinction has to be made. Path planning is related to the techniques that allow the generation of a geometric path from start to goal states while Trajectory planning are the collection of algorithms that allow the conversion of this geometric path by providing the additional information on how to transverse it with respect to time. This is generally done by associating a velocity and/or acceleration to each of the waypoints in the path. The first part is where the sampling based methods work while the optimization deals with the actual trajectory generation, which contains the information (position, velocity, acceleration, time, among others) that the UAV effectively requires to navigate the environment. We will provide a more formal description of these elements in this chapter.

Regarding the online path replanning, as mentioned due to the need of a quick response to dynamic changes in the environment, the optimality guarantee is sacrificed. The previous trajectory is discarded since with the avoidance maneuvers render the initial trajectory's optimality invalid. This is a routine whose job is check at all times for current trajectory feasibility and implement the necessary adjustments to maintain the UAV safety in flight. This component of the framework is only applicable locally within a certain distance of the UAV as to simulate the range of the equipped sensors. As to respect the mission defined requirements and to harness the benefits of the previously mentioned optimization method, this method is based on a reactive approach that replaces the previous optimal sampler with a much faster sampling approach based on the RRT Connect and keeps the optimization method while employing the changes necessary to achieve real-time avoidance. It must be added that at each instance in time the moving obstacle is viewed effectively by the framework as a static obstacle, which requires the active replanning to be as time efficient as possible to account for the need of multiple instances of trajectory generation.

To better illustrate the proposed framework a simple example was drawn up representing an autonomous mission of an UAV from a start to a goal destination. Two major situations are brought up when dealing with moving obstacles.

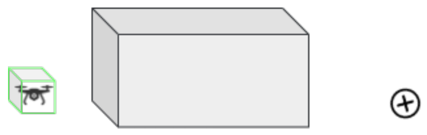
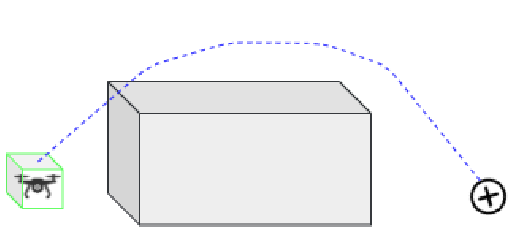


Figure 3.2: The UAV at the start position, a static obstacle and a cross circled mark representing the goal destination.

In figure 3.2, the UAV (bounded by a green cube) is at its start position and the goal destination is represented by the cross circled mark.

From that position, a collision-free trajectory is generated for the first time and the UAV starts flying across the intended path (figure 3.3). This is where the offline planning takes place, providing an initial trajectory for the UAV to follow.

At this point, the UAV detects an intruder in its surroundings (it is in line of sight of its equipped sensors). For simplicity, the intruder is also an UAV (bounded by a purple cube). The intruder is not perceived as an immediate threat by the UAV and so continues its course. Two distinct situations may


(a) Trajectory Generation

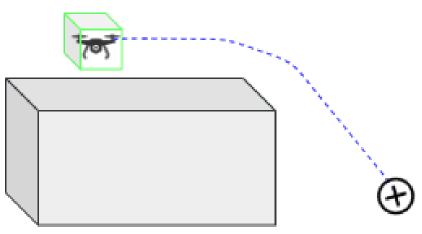

(b) Trajectory tracking

Figure 3.3: In (a), a first global trajectory is generated which is then followed by the UAV as shown in (b).

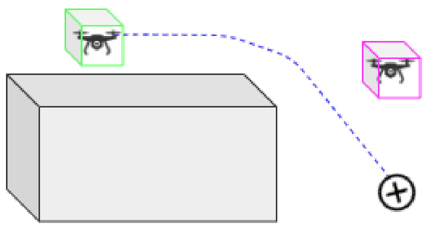
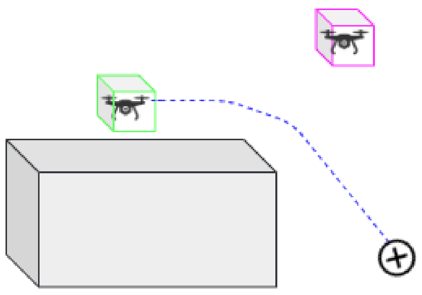


Figure 3.4: Detection of an Intruder represented by an UAV bounded by a purple cube

arise:

- the intruder stays out of collision course with the UAV and remains an **inactive threat**;
- the intruder enters in collision course with the UAV and becomes an **active threat**;

In the first case, the UAV continues to monitor the location of the intruder as long as its in line of sight. If at any point the intruder enters in collision course with the UAV, we are presented with the second case. Collision Avoidance protocols are activated and a new trajectory is generated from a point prior to the predicted collision in the original trajectory. This encompasses the online replanning which provides alternatives trajectories during flight to ensure UAV safety.


(a) The intruder keeps out of the UAV's planned trajectory

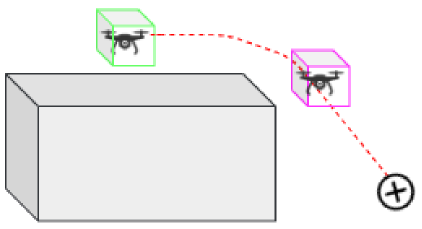
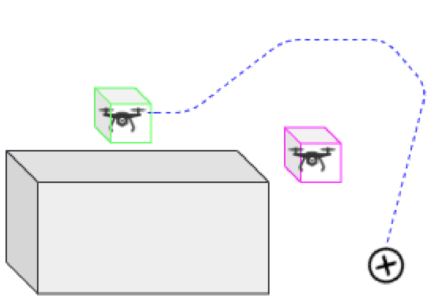

(b) The intruder enters collision in collision course with the UAV's planned trajectory

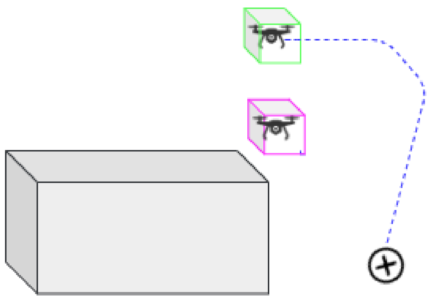
Figure 3.5: Possible situations when dealing with moving obstacles

When the avoidance protocols are activated, ideally a new collision-free trajectory is generated within a time frame that allows for safe avoidance.

It is possible, however, that this reaction to dynamic changes in the environment may not be feasible to avoid critical situations. An example is the turning around corners of static obstacles and suddenly encountering a moving obstacle immediately in its way. To prevent these situations, the concept of maximizing clearance around obstacles is explored. Another limitating factor regarding collision avoidance is



(a) A new collision-free trajectory is generated around the intruder



(b) The UAV continues on its way onto the goal destination

Figure 3.6: Collision Avoidance maneuver

the computation time of the solution in the optimization problem. While the decision on the optimization technique was based on the efficient results it wields, it is possible that this computation time might exceed time thresholds that make impossible to generate a trajectory in the necessary window of time to avoid an incoming obstacle. This problem is mitigated by applying a computation time threshold.

In a case of an imminent collision, a strategy is explored in situations where there is enough space clearance to safely apply repulsion forces to the UAV in order to replace the current avoidance protocol.

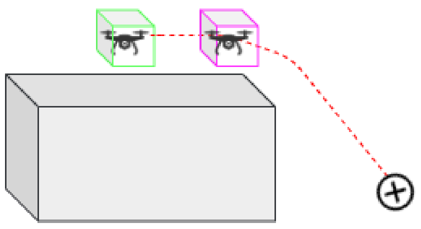


Figure 3.7: Illustration of an imminent fatal collision

From this overall description of the envisioned implementation, the several components of the framework will be thoroughly detailed sequentially. Before entering the inner workings of the implementation, a formal description of the problem will be presented to allow for an easier understanding of how the components fit together in the overall system.

3.2 Problem Formulation

Lets recover what was written about **configuration space** in the previous chapter and expand upon it. The location of a robot with *n*-links constitutes its configuration q which is part of the set of all possible configurations C or $q \in C$. In the case of a quadrotor and particularly in the context of the developed framework, its a rigid object moving on a three dimensional space which follows the rules of *Euclidean geometry*. Therefore, q is a tuple of three *cartesian* coordinates (x, y, z), representing the position of the center of mass of the quadrotor. The configuration space C can be formally written as:

$$C = \{x, y, z\} = \mathbb{R}^3 \tag{3.1}$$

Lets now define the **workspace** W as the subset of $\mathbb{R}^3$ where the quadrotor navigates. In this particular instance, the configuration space is identical to the workspace and consists as previously mentioned of a three dimensional cartesian space. Obstacles might be present in the workspace or $o_i \in W$. As opposed, W_{free} consists of the free space and illustrates the navigable regions of the discrete space and can be written as:

$$W_{free} = \{W \setminus o_i, \forall i\} \tag{3.2}$$

The subset of W occupied by the quadrotor is defined by $A = A(q)$. The subset of C occupied by obstacles is written as:

$$C_{occupied} = \{q \in C : A(q) \cap o_i \neq \emptyset, \forall i\} \tag{3.3}$$

With the previous information, we can define the collision-free configurations of the quadrotor as:

$$C_{free} := \{C \setminus C_{occupied}\} \tag{3.4}$$

The **path planning problem** can be defined within the configuration space as finding the collision-free path connecting the initial configuration (q_{start}) and goal configuration (q_{goal}). This can be written formally as a continuous function f where :

$$f : [0,1] \rightarrow C_{free} \tag{3.5}$$

subject to $f(0) = q_{start}, f(1) = q_{goal}$.
In the context of this thesis, this definition is related to the sampling based methods employed, both the Informed RRT* and the RRT Connect. In the context each methodology used we should make a distinction between optimal and non-optimal path planning. **Optimal Path Planning** can be defined by modifying the previous definition to include a cost function $c : \sum \rightarrow \mathbb{R}_{\geq 0}$ to minimize (in this case the cost is the length of the path). Formally we have:

$$\sigma : [0,1] \rightarrow C_{free} \ , \ c(\sigma *) = min\{c(\sigma) : \sigma \text{ is feasible}\} \tag{3.6}$$

On the other hand, the **trajectory planning problem** can be defined within the configuration space as finding the collision-free path connecting the initial configuration (q_{start}) and goal configuration (q_{goal}) with explicit parametrization of time. This can be written formally as a continuous function g where :

$$g : [0,1] \rightarrow C_{free} \tag{3.7}$$

$$[T_{start}, T_{goal}] \ni t \rightarrow \tau \in [0,1] : q(t) = \gamma(\tau) \in C_{free} \tag{3.8}$$

The time parametrization (t) implies that one or more considerations need to be made such as:

- **Velocity** of a motion, $\frac{d}{dt}q(t)$;
- **Acceleration** of a motion, $\frac{d^2}{dt^2}q(t)$;

- **Jerk** of a motion, $\frac{d^3}{dt^3}q(t)$;
- **Snap** of a motion, $\frac{d^4}{dt^4}q(t)$;

There are higher derivatives that determine the motion of a robot but due to the scope of the methodology followed here, we will focus only on the derivates up to and including **snap**. In a similar way to the path planning problem, this is where the optimization method inserts itself into by picking up the previous collision-free path and imbuing it with the necessary kinematic characteristics mentioned above to turn it into a flyable trajectory. There are constraints inherent to the kinematics that must be respected and to that effect the chosen methodology had that into account. The formal definition of the problems to tackle allows us to move on to the first integral part of the framework which deals with the representation in a computational manner of the physical environments.

3.3 Map Representation

As stated before map representation is a necessary component to be defined in the problem of autonomous navigation. We want to translate the physical world (ground, static and moving obstacles) into our configuration space so that the algorithms responsible for safely guiding the UAV between locations all operate under the same data structures. In the second chapter, we presented several representations of configuration spaces and the octomap representation was outlined as a reliable method which paired with the fact that it was available as an open-source project made the choice fall behind this representation.

3.3.1 Octomap

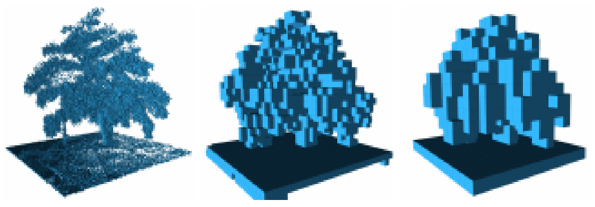


Figure 3.8: Multiple resolutions of the same map from a higher resolution on the left to a progressively lesser one on the right. Taken directly from [12]

An Octomap is an occupancy grid mapping approach to build 3D environments by dividing the space in cubic volumes of equal size called *voxels*. Octrees are used as the representation of space and allow each voxel to be recursively subdivided into eight voxels of smaller size therefore increasing the resolution of the grid by a factor of two with each subdivision. The compactness offered by octrees is leveraged by this approach by applying probabilistic modelling. Given a sensor measurement $z_{1:t}$, the probability of a leaf node n (corresponding to the smallest defined voxel subdivision) being occupied is estimated as:

$$P(n|z_{1:t}) = \left[1 + \frac{1 - P(n|z_t)}{P(n|z_t)} \frac{1 - P(n|z_{1:t-1})}{P(n|z_{1:t-1})} \frac{P(n)}{1 - P(n)}\right]^{-1} \tag{3.9}$$

where $P(n|zt)$ is specific to the sensor used.
The previous equation can be simplified assuming a uniform prior $P(n) = 0.5$ and by using the *logOdds (L)* notation (which allow for faster updates by reducing computation time),

$$L(n|z_{1:t}) = L(n|z_{1:t-1}) + L(n|z_t) \tag{3.10}$$

To keep the updatability of this framework a clamping update policy (citar) is enforced:

$$L(n|z_{1:t}) = max(min(L(n|z_{1:t-1}) + L(n|z_t), l_{max}), l_{min}) \tag{3.11}$$

with l_{max} and l_{min} representing the upper and lower bounds. The chosen values determine a tradeoff between compactness and map confidence. Choosing lower values of both parameters leads to a stronger compression while the opposite makes for finer grained map.

Besides updating leaf nodes, there is also the need to update the inner nodes. Two strategies can be used to determine the occupancy probability of these nodes based on the occupancy probability of each of its eight child nodes n_i, either by using the mean occupancy:

$$\bar{l}(n) = \frac{1}{8} \sum_{i=1}^{8} L(n_i) \tag{3.12}$$

or use a more conservative approach in the maximum occupancy:

$$\hat{l}(n) = \underset{i}{max} L(n_i) \tag{3.13}$$

By taking the maximum value that a child node exhibits it leads to a higher probability of taking an observed part of the environment as occupied thus being more suitable for robot navigation safety and computation efficiency. Tree compression is done when a node becomes stable (if its value $L(n_i)$ surpasses either threshold l_{max} or l_{min}) and can be assumed as free or occupied. If all the child nodes of a given parent node exhibit this stable state then they can be pruned leading to a reduction in the overall computational burden of this method. However, new measurements can lead to the reversion of pruned child nodes by regenerating them. One additional implementation to reduce the size of the octree in memory is the use of only one pointer per node (pointing to an array of eight pointers) instead of eight individual pointers per node, meaning that the array only is allocated when children of this node need to be initialized. Maps generated using this method can be store for posterior use.

The advantages of this method can be summarized by the following properties:

- **Full 3D modelling** - The map is able to model arbitrary environments (free, occupied and unknown areas) without prior assumptions about it.

- **Adaptability** - It is possible to update the map with new information or sensor readings due to the use of a probabilistic occupancy estimation which accounts for sensor noise and/or changes in the environment.

- **Flexibility** - The octree implementation allows for multi-resolution which means it can serve autonomous navigation missions where high precision is a lesser requirement and to that effect coarse maps would be used or the opposite for example navigating in cluttered environments which requires fine resolution maps.
- **Compactness** - The map is stored efficiently, both in memory and on disk and can be compressed for later usage.

The Octomap approach presents a reliable solution to the need of a map representation. This combined with the fact that there is an open source C++ library where this method is fully realized as well as seamlessly integration with the sampling methods employed and the prevalence of this method in literature were the deciding factors that made this the map representation used throughout the design of the framework.

3.3.2 Integration

In the scope of this thesis, the application of autonomous navigation is limited to partially known environments where the static obstacles' location are known *a priori* and dynamic obstacles detection is simulated within the range of hypothetical sensors equipped by the quadrotor. There were several methods available that allowed to build octomaps from pre-built 3D models such as a tool offered by a multirotor physics simulator [78]. The idea behind the feature took care of the entire generation of the octomap by creating a savable octomap file. We would need only to provide the 3D map to convert. However, due to conflicts that surged from incompatibilities detected between the Operating System and the provided software rendered this tool useless. Another way to generate an octomap was based on the implementation of a software simulation in which an UAV equipped with a RGB depth camera would move around in the environment and progressively construct the octomap through pre-built tools. This method was partially successful since there were a significant amount of crashes and bugs experienced when running the simulation that resulted in either the constructed octomap having incoherent levels of resolution or even total corruption of the generated file. Due to the drawbacks faced and since both of the previous methods both pull sensorial information from simulations and make use of the octomap provided library to construct the representations, a decision was made to go straight into the manipulation of these functions and data structures to construct the octomaps. It is necessary to add that this does not compromise the integrity of the chosen representation. There are two different situations in which octomap modelling is employed in the developed framework:

- **Static Map** contains the *a priori* knowledge of the environment before the quadrotor takes flight. Only the static obstacles are modelled;
- **Dynamic Map** is the designation of the Static Map when the quadrotor is moving. When an

intruder is within a defined range of the quadrotor, the octomap is updated periodically according to the position of the intruder.

The distinction made between the maps is made solely on a theoretical interpretation since we have effectively only one global map in memory which is updated as mentioned when required.

The obstacles in the environment are described by their boundaries, i.e. they are represented by defining the following boundaries:

$$\underset{x_1,x_2\in\mathbb{R}\ y_1,y_2\in\mathbb{R}\ z_1,z_2\in\mathbb{R}}{\text{subject to}} \qquad x_1 \leq x \leq x_2 \quad y_1 \leq y \leq y_2 \quad z_1 \leq z \leq z_2 \tag{3.14}$$

The following pseudo code represents the routine that allows the generation of the static map from a file containing the description of the environment.

Algorithm 1: Pseudo algorithm for building an octomap from a file containing a set of static obstacles

```
1  function build-octomap $(environment.txt)$;
   Input  : A text file containing the description of boundaries of obstacles of an environment
   Output: Octree object
2  $N \leftarrow$ number of obstacles;
3  $b_i \leftarrow$ boundaries of an obstacle;
4  $octree \leftarrow$ octree object;
5  $o_n \leftarrow$ octree node;
6  $octree \leftarrow initializeTree()$;
7  $N \leftarrow getNumberofObstacles(environmentFile)$;
8  for $i = 1...N$ do
9      $b_i \leftarrow getObstacle(environmentFile)$;
10     while $b_i \neq null$ do
11         $o_n \leftarrow$ getMeasure$(b_i)$;
12         $octree \leftarrow$ insertNode$(o_n)$;
13     end
14 end
```

This is a simple algorithm that as explained takes the boundaries of each obstacle in a description file, converts them to the appropriate measurements creating for each one the corresponding node and inserts them in the octree. Once the octree is completed, the map is saved under the corresponding octomap type and is ready to be used by the trajectory planning algorithms.

Apart from this, we have the dynamic map which must be updated whenever the intruder is at reach from the sensors in the quadrotor. This update can be translated by the following pseudo code which shares the same functioning as the previous one.

In this algorithm, the static map is already created and is updated when an intruder is detected. Several assumptions are made relative to the detection and tracking of the dynamic obstacle. It is assumed that the quadrotor is equipped with the necessary sensors to accomplish the detection and is able to continuously track it as long as it stays within range of the sensors. Moreover, to simplify the detection process the intruder is chosen as another similar quadrotor. Due to the inherent complexity of detection and tracking systems, the intruder can not be represented in a dynamic map with all its structural detail nor it would be beneficial for the planning process since as its going to be explained further on several

Algorithm 2: Pseudo algorithm for updating an octomap when an intruder is detected

```
1  function update-octomap ($octree$);
   Input  : The octree data structure that contains the current state of the environment
   Output: Octree object
2  $octree \leftarrow$ octree object;
3  $o_n \leftarrow$ octree node;
4  $intruderObject.position \leftarrow$ intruder position;
5  $intruderObject.dimensions \leftarrow$ intruder dimensions;
6  $bbox \leftarrow$ bounded box dimensions;
7  $bbox \leftarrow$ getDimensions($intruderObject.dimensions$);
8  while $bbox \neq null$ do
9      $o_n \leftarrow$ getupdate($bbox + intruderObject.position$);
10     $octree \leftarrow$ insertNode($o_n$);
11 end
```

safety margins will be employed to account for possible deviations in real world situations. To that effect, the intruder is represented using a bounded box around its center of mass. With this in mind the update process is done in a similar way to the construction of the static map, according to the size of the intruder a box is created around it with a certain dimension and given the boundaries of that box, we again create the necessary measurements and nodes, inserting them in the octree. At each time step, the previous created nodes are removed if the intruder moves to a different position and the process repeats itself until the quadrotor reaches its goal or the intruder falls out of sight relative to its sensors. This was only tested successfully for one intruder at a time and if we increase the number of active intruders, besides overloading the detection and tracking capabilities, updating the octomap with a large number of active intruders might become unfeasible. There is one additional factor that must be covered when dealing with octomaps and it pertains to the resolution of said representation. It was mentioned before that one of the main benefits of using octomaps was in being able to choose the resolution of the map depending on the nature of the requirements. While on the one hand we want a map as detailed as possible for a higher degree of certainty in our autonomous navigation, it is also true that increasing the resolution past a certain point results in a higher computational time for our planning algorithms since the higher the resolution, more nodes are available and therefore the more instances of collision checking are needed. It is also worth noting that past a certain resolution we do not have more relevant information for the planning algorithms. With this being said, the chosen resolution of the octomap was of $10\ cm$. This means that the smallest voxel of the octomap is of that size.

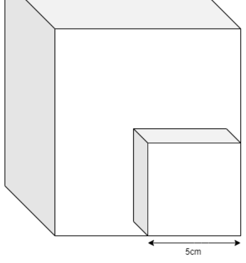


Figure 3.9: Voxel resolution

This decision tries to achieve the best tradeoff between resolution and the computation burden on the execution of algorithms. It was mentioned before that collision checking is a major bottleneck related to planning algorithms so a simple simulation in order to study the overall impact of changing resolution. The pseudo code of the collision checking simulation won't be detailed since it is going to be featured ahead when detailing the planning algorithms. For now let's just present the simulation as a common obstacle (e.g a box) and a point outside the box and the collision checking method employed is the one to be featured throughout the algorithm. In the simulation, the point is checked for collision against all of the nodes in the octree representing the octomap. While this may appear at first glance an unorthodox approach, it is a good tool to provide an appreciation on the impact of the octomap representation on the overall framework.

The following graph illustrates the decrease in performance when resolution increases.

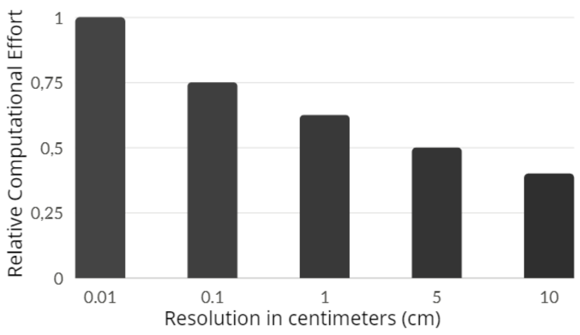


Figure 3.10: Collision Checking Simulation Results (averaged over 50 simulations).

The horizontal axis represents resolution of the octomap in cm, while in the vertical axis starting from the highest resolution tested ($0.01\ cm$), a relative measure of computational effort (time) which is equal to 1 as expected for this resolution. With the decrease in resolution to $0.1\ cm$ we have approximately a 25% drop in computational effort. This trend follows as expected with the decrease in resolution. At $10\ cm$ and even $5\ cm$ we have a good compromise between resolution and a substantial reduction in computational effort compared to the highest resolution. The choice fell on the first since apart from the slight decrease in computational effort when compared to the later, the scale difference does not carry significant benefits in transversing the environments. There is one additional reason for this resolution that while it does not reflect directly on the scope of this work, it affects robotic systems in which the entire environment is mapped as the robot is moving, having no *a priori* knowledge. In these situations, decreasing resolution is of extreme importance to allow a real time navigation.

3.4 Offline Pre-Flight Path Planning

The decision on the map representation allows us to move on to the first component of the framework which deals with the generation of a feasible trajectory that is able to guide the quadrotor from start to goal. This is as mentioned a non time constrained generation that is carried out before the quadrotor begins the flight. The following scheme illustrates the individual sub-components that make up this part of the framework and how they are interconnected. Again, each individual part here represented will be detailed thoroughly as the work unfolds. To avoid unnecessary repetition, let's assume the map representation is available from the previous sub-chapter.

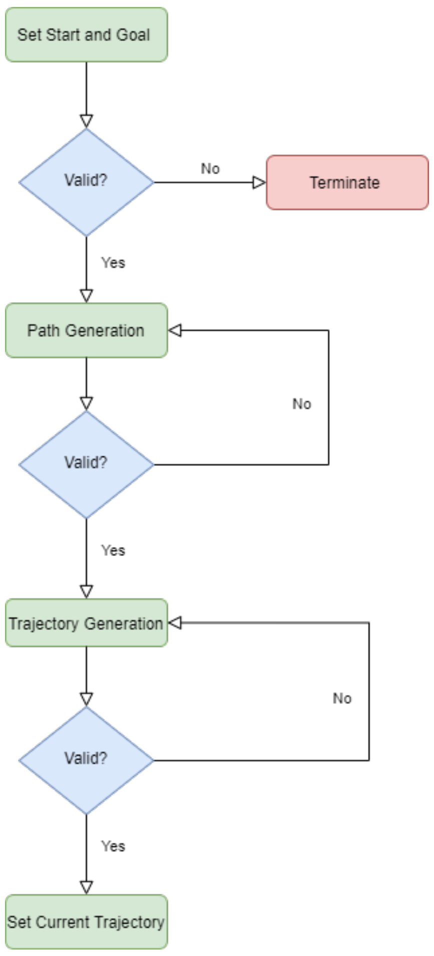


Figure 3.11: Offline Pre-Flight Path Planning proposed solution. This represents a rough overall description with each component being described in detail throughout this chapter

The flowchart represents a solution for the problem formulation described at the beginning of this chapter. A top down approach is now taken, starting by mentioning the **Set Start and Goal** as the simple routine that takes a desired start configuration $q_{start} = (x_{start}, y_{start}, z_{start})$ and goal configuration $q_{goal} = (x_{goal}, y_{goal}, z_{goal})$, defining them as mission parameters. There is as expected a validity checker whose main objective is to determine if both the path and trajectory configurations respect environment constraints, that is if they are not placed inside static obstacles and if they respect the mission parameters. This is done simply by employing a collision query for each one as well as checking the initial and final states. A particular method for collision checking is used and it will be described next when inserted in the context of valid path sampling.

For now let's focus on the next component of the proposed solution which deals with **Path Generation**.

3.4.1 Path Generation

This step is comprised of the methods that allow the generation of a collision-free path which is done in this work by a sampling based method based on RRT called Informed RRT*. Before describing the implementation behind this method, there is an important concept that needs to be explained related to how collision checking with obstacles is done.

Collision Checking

Sampling based methods work by sampling the configuration space and trying to connect the sampled configurations until a collision-free path is formed. Therefore, there is a need to have a collision checking method that is able to tell if a given sampled configuration is inside the free space or if on the other hand it is in the restricted space of obstacles. The algorithms needed to employ these methods are as mentioned before one of the most computational intense processes and in the context of this work there is an additional need of finding a specific collision checking method compatible with the octomap representation. The choice fell on the **Flexible Collision Library (FCL)** [79] whose implementation is available in [80]. This framework allows for accurate and efficient collision checking and proximity computation between different model data representations among which octrees are included. The traversal process it employs to achieve these goals is based on a three step approach [79]:

- Object representation : objects are represented in a hierarchical data structure to assure collision and proximity computation efficiency. From simple basic geometric shapes (such as spheres, cones, among others) to deformable models and more importantly in the context of this work, octrees. They are represented by a data structure called *CollisionObject*, consisting of a bounding volume hierarchy which can be one of four distinct types AABB (axis-aligned bounding boxes), OBB (Object-oriented bounding boxes), RSS (rectangle swept spheres) or k-DOP (discrete oriented polytope). This bounding volume representation allows for two different operations, **overlap** and **intersection** between bounding volumes of two objects. There are advantages and drawbacks to each representation, for example performing these operation with AABB is computationally less expensive when compared to OBB but the latter allows for a finer representation of the objects. When dealing with continuous collision checking between deformable models, kDOP and AABB prove to be better than OBB and RSS since the way in which they represent bounding volumes allows for quicker changes. There are many more examples of these tradeoffs between the four types but the authors choose to use AABB to model the nodes in octrees, mainly prioritizing efficiency over precision.

- Traversal node structure that stores all the necessary information to perform the collision or proximity computation between two bounding volume hierarchies and also decides on the appropriate traversal strategy to take. In the figure below, we can see the full structure of the traversal node.
- Hierarchy traversal begins after traversal node initialization by traversing the bounding volume test tree (BVTT) generated from the two bouding volumes. The complete traversal hierarchy can also be visualized in figure D.1 in Appendix D .

(It must be noted however that at the time [79] was developed, octomap inclusion was not yet possible but has since been added to the available library)

The collision checking module employed can be illustrated by the following pseudo-code and it serves two different instances of collision: between the quadrotor and the octomap or between the quadrotor and the intruder. Although different, due to the fact that the octomap is updated periodically with the information regarding intruders, there is no distinction in the implementation of this module between the two situations.

Algorithm 3: Pseudo algorithm for performing collision checking

```
1  function collisionChecking ($collisionObject1, collisionObject2$);
   Input   : Two collision objects to perform operations on
   Output: True if there is a collision or False otherwise
2  $collisionObject1 \leftarrow$ object 1;
3  $collisionObject2 \leftarrow$ object 2;
4  $intruderObject.position \leftarrow$ intruder position;
5  $intruderObject.dimensions \leftarrow$ intruder dimensions;
6  $bbox \leftarrow$ bounded box dimensions;
7  $bbox \leftarrow$ getDimensions($intruderObject.dimensions$);
8  $isCollisionResult \leftarrow$ collide($collisionObject1, collisionObject2$);
9  if isCollisionResult then
10     return $True$;
11 else
12     return $False$;
13 end
```

The distance computation is also employed and its implementation follows similar logic. However, this operation is used only to simulate the range detection of sensors when there are intruders nearby the quadrotor and therefore has no application other than serving this purpose.

Informed RRT*

The informed RRT* exhibits (regarding general aspects) a similar behaviour to the basic RRT algorithm which can be summed up by the following pseudo-code:

Algorithm 4: Pseudo algorithm for RRT

```
1  function RRTGrow ($(q_{start}, q_{goal})$);
   Input   : start configuration and goal configuration
   Output: tree structure $G = (V, E)$ containing list of vertex and edges
2  $V \leftarrow \{q_{start}\}$;
3  $E \leftarrow \phi$;
4  for iteration $= 1, 2, ...N$ do
5      $q_{rand} \leftarrow$ SampleFree$(i)$;
6      $q_{nearest} \leftarrow$ Nearest$(G = (V, E), q_{rand})$;
7      $q_{new} \leftarrow$ Steer$(q_{nearest}, q_{rand})$;
8      if CollisionFree$(q_{nearest}, q_{new})$ then
9          $V \leftarrow V \cup \{q_{new}\}$;
10         $E \leftarrow E \cup \{(q_{nearest}, q_{new})\}$;
11     end
12 end
13 return $G = (V, E)$
```

The algorithm starts by sampling $SampleFree()$ a random configuration q_{rand} from the configuration space. $Nearest(G = (V, E), q_{rand})$ finds the nearest configuration in V which is the configuration in that list that is closest to q_{rand} in distance and then $Steer(q_{nearest}, q_{rand})$ finds a new configuration that is closer to q_{rand} than to $q_{nearest}$. We designate this new configuration as q_{new} and this new configuration is the same as the initially random sampled configuration q_{rand} only when the new configuration is not further away than a certain distance threshold d from the nearest configuration $q_{nearest}$. If the space between the nearest configuration and the new configuration is collision free ($CollisionFree(q_{nearest}, q_{new})$) then the new configuration is inserted into V and the edge composed by the pair of $(q_{nearest}, q_{new})$ is added to the list of edges formed by E. The algorithm ceases when the goal configuration q_{goal} is added to V.

With the basis of the algorithm explained we can now move on to the explanation of the more complex algorithms derived from it. It is important to first analyse RRT* since the method used [47] represents an improvement upon it. Therefore, the two methods share most of the same algorithm between themselves and with the basic RRT which can be summed up for the Informed RRT* by the pseudo-code below. Since RRT* returns an optimal path, we will refer to the tree structure as T instead of G. The RRT* implementation corresponds to every line except 4, 7-8 (this second line in particular in the case of RRT* is the same as in the RRT) and 37-38 that correspond to the additional functionalities provided by the Informed RRT*. Let's first interpret the pseudo-code related to the RRT* since the changes introduced can be explained afterwards in this context without the need of fully explaining two individual algorithms. In a similar way to the basic RRT, a newly random configuration is sampled and we try to connect it to the nearest configuration in the graph. If the connection is successful, it is added to the vertex set V. Contrary to the previous algorithm, we attempt to connect this new vertex q_{new} to other vertexes already in V but limited to a defined region in space which corresponds to the space within a ball of radius r_{RRT*} (line 12). However, not all connections result in new edges being added since this method avoids adding ”redundant” edges by employing a rewiring of the stored tree. Before doing so, we must discuss some functions which are new to this algorithm: Parent : $V \to V$ is a function that maps a vertex $v \in V$ to the unique vertex $u \in V$ such that $(u, v) \in E$. For the root vertex of $\mathrm{T}, Parent(v_0) = v_0$; line$(q_1, q_2) : [0, s] \to \chi$ denotes the straight-line path from point $q_1 \in \mathbb{R}^n$ to point $q_2 \in \mathbb{R}^n$; the additive cost function Cost(v) = Cost(Parent(v)) $+ c \cdot$ Line(Parent$(v), v)$ whose value represents the total cost length of traversing from the root/initial vertex to the current vertex v. For the root vertex we have a null cost Cost$(q_{start}) = 0$.

In line 14 and 15, we first see the additive cost function that takes the value corresponding to the latest vertex added to the tree. From line 17 to line 24, the minimum-cost path is searched for within the set of near vertexes Q_{near}. Only the vertex from this set that results in the lowest possible cost compared to the others has its edge added to the tree (line 25). The rewiring happens from line 26 to 36, new edges are created from q_{near} to vertices in Q_{near}, if and only if the path through q_{new} has a lower cost than the path through the current parent. The edge linking the vertex to its current parent is deleted (line 32), to maintain the tree structure and the new edge is added (line 33). This way this method is guaranteed to find an optimal solution in terms of path length (the authors prove asymptotic optimality of the method in [44]).

Informed RRT* maintains the same qualities of this method while improving the convergence rate and the quality of the solution. To do that, it focuses on the search part of the algorithm, specifically implementing a direct sampling of the ellipsoidal heuristic which can be visualized below and corresponds among other changes in the RRT* algorithm to the definition of a specific sampling function shown here as algorithm 6.

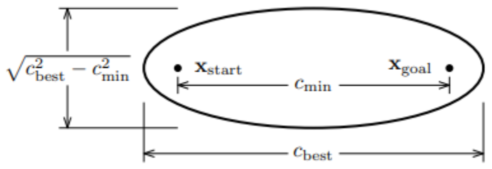

$\sqrt{c_{\mathrm{best}}^2 - c_{\mathrm{min}}^2}$
$\mathbf{x}_{\mathrm{start}}$
$\mathbf{x}_{\mathrm{goal}}$
c_{min}
c_{best}

Figure 3.12: Heuristic sampling domain represented for a two dimensional space ($\mathbb{R}^2$) problem seeking to minimize path length. Visually corresponds to an ellipse with the initial state, q_{start}, and the goal state, q_{goal} as focal points. The theoretical minimum cost between the two, c_{min}, and the cost of the best solution found to date, c_{best}. The eccentricity of the ellipse is given by the quotient between the two costs. Taken directly from [47]

When the algorithm runs through lines 7 and 8 for the first time the solution set is empty Q_{soln} and as convention we take its cost as infinite. Therefore looking at algorithm 6, sampling is performed as previously from an uniform distribution U (any other type of distribution is equally valid but the common choice falls on this type of distribution). When a configuration close to the goal configuration is sampled, the additional features of this algorithm are put in place. It is common to define this region as the space within a ball of radius r_{goal} centred at the goal configuration. In this case, the random sampled configuration is added to the solution set (lines 37-38) and in the next iteration we have a value different than infinity for the best current solution cost. This means that the sampling algorithm now uses the minimum of this list (different than infinity) and starts sampling within the ellipsoidal domain (as pictured in figure 3.13), that is the sampling domain is now restricted to a smaller subset of the original domain. Going through algorithm 6 we can see that line 2 to line 4 can be calculated once at the start of the problem since they define the minimum cost path from q_{start} to q_{goal} corresponding to a straight line that unites both states, define the q_{centre} that allows to define the ellipsoidal subspace as well as RotationToWorldFrame(q_{start}, q_{goal}) which returns a rotation matrix from the ellipsoid aligned frame to the world frame. In line 5, the radius of the N-dimensional ball is defined as the best current cost. This is done under the requirement of an admissible heuristic of which the Euclidean distance define by the cost is (the authors provide the necessary mathematical proof for the employment of this). From line 6 to line 10, we have the actual sampling from the ellipsoid subset created. This progressive reduction in the size of the sampling space greatly benefits the convergence of this method relative to the RRT*. It is worth noting however that there may be cases where the improvements achieved are incremental but for the large majority the first conclusion is prevalent.

Algorithm 5: Pseudo algorithm for Informed RRT*

```
1  function RRTGrow ($q_{start}, q_{goal}$);
   Input  : start configuration and goal configuration
   Output: tree structure $\mathrm{T} = (V, E)$ containing list of vertex and edges
2  $V \leftarrow \{q_{start}\}$;
3  $E \leftarrow \phi$;
4  $Q_{soln} \leftarrow \phi$;
5  $\mathrm{T} \leftarrow (V, E)$;
6  for iteration $= 1, 2, ...N$ do
7      $c_{best} \leftarrow \min_{q_{soln} \in Q_{soln}} \{\mathrm{Cost}(q_{soln}\}$;
8      $q_{rand} \leftarrow \mathrm{Sample}(q_{start}, q_{goal}, c_{best})$;
9      $q_{nearest} \leftarrow \mathrm{Nearest}(\mathrm{T}, q_{rand})$;
10     $q_{new} \leftarrow \mathrm{Steer}(q_{nearest}, q_{rand})$;
11     if CollisionFree$(q_{nearest}, q_{new})$ then
12         $V \leftarrow V \cup \{q_{new}\}$;
13         $Q_{near} \leftarrow \mathrm{Near}(\mathrm{T}, q_{new}, r_{\mathrm{RRT}^*})$;
14         $q_{min} \leftarrow q_{new}$;
15         $c_{min} \leftarrow \mathrm{Cost}(q_{min}) + c \cdot \mathrm{Line}(q_{nearest}, q_{new})$;
16         for $\forall q_{near} \in Q_{near}$ do
17             $c_{new} \leftarrow \mathrm{Cost}(q_{near}) + c \cdot \mathrm{Line}(q_{near}, q_{new})$;
18             if $c_{new} < c_{min}$ then
19                 if $CollisionFree(q_{near}, q_{new})$ then
20                     $q_{min} \leftarrow q_{near}$;
21                     $c_{min} \leftarrow c_{new}$;
22                 end
23             end
24         end
25         $E \leftarrow E \cup \{(q_{min}, q_{new})\}$;
26         for $\forall q_{near} \in Q_{near}$ do
27             $c_{near} \leftarrow \mathrm{Cost}(q_{near})$;
28             $c_{new} \leftarrow \mathrm{Cost}(q_{new}) + c \cdot \mathrm{Line}(q_{new}, q_{near})$;
29             if $c_{new} c_{near}$ then
30                 if $CollisionFree(q_{new}, q_{near})$ then
31                     $q_{parent} \leftarrow \mathrm{Parent}(q_{near})$;
32                     $E \leftarrow E \setminus \{(q_{parent}, q_{near})\}$;
33                     $E \leftarrow E \cup \{(q_{new}, q_{near})\}$;
34                 end
35             end
36         end
37         if $InGoalRegion(q_{new})$ then
38             $X_{soln} \leftarrow X_{soln} \cup \{q_{new}\}$;
39         end
40     end
41 end
42 return $\mathrm{T}$
```

Algorithm 6: Pseudo algorithm for for Sample function

```
1  function Sample ($q_{start}, q_{goal}, c_{best}$);
   Input  : start configuration , goal configuration and lowest cost so far
   Output: random sample
2  if $c_{max} < \infty$ then
3      $c_{min} \leftarrow \| q_{goal} - q_{start} \|$;
4      $q_{centre} \leftarrow (q_{start} + q_{goal})/2$;
5      $C \leftarrow$ RotationToWorldFrame$(q_{start}, q_{goal})$;
6      $r_1 \leftarrow c_{max}/2$;
7      $r_i \leftarrow (\sqrt{c_{max}^2 - c_{min}^2})/2$   subject to   $i = 2, ..., n$;
8      $L \leftarrow$ diag$(r_1, r_2, ..., r_n)$;
9      $q_{ball} \leftarrow$ SampleUnitNBall;
10     $q_{rand} \leftarrow (CLq_{ball} + q_{centre}) \cap Q$;
11 end
12 else if $c_{max} \to \infty$ then
13     $q_{rand} \sim U(Q)$;
14 end
15 return $q_{rand}$
```

In order to implement this method, the **Open Motion Planning Library** [81] was used. The library (written in C++) contains many state-of-the-art sampling-based motion planning algorithms among which the Informed RRT* and offers unparalleled computational efficiency when it comes to the execution time of sampling methods as well as a high degree of reliability which is needed to safely implement autonomous capabilities.

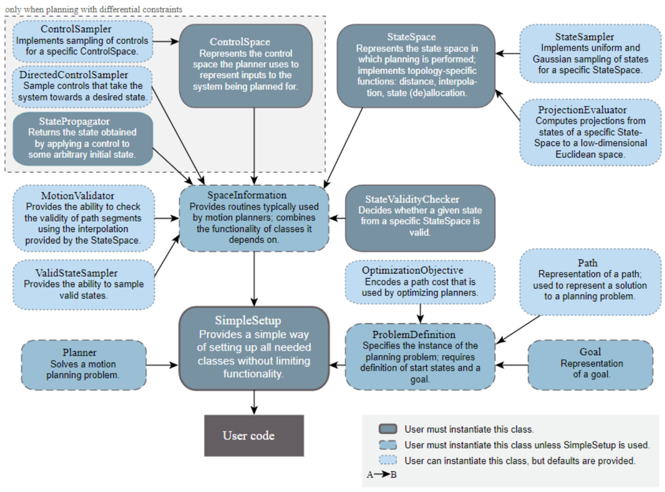
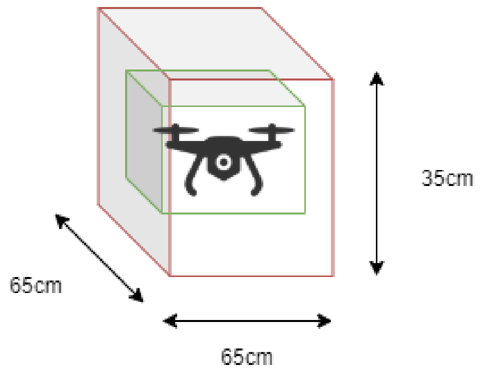

only when planning with differential constraints
ControlSampler
Implements sampling of controls for a specific ControlSpace.
DirectedControlSampler
Sample controls that take the system towards a desired state.
StatePropagator
Returns the state obtained by applying a control to some arbitrary initial state.
ControlSpace
Represents the control space the planner uses to represent inputs to the system being planned for.
StateSpace
Represents the state space in which planning is performed; implements topology-specific functions: distance, interpolation, state (de)allocation.
StateSampler
Implements uniform and Gaussian sampling of states for a specific StateSpace.
ProjectionEvaluator
Computes projections from states of a specific StateSpace to a low-dimensional Euclidean space.
MotionValidator
Provides the ability to check the validity of path segments using the interpolation provided by the StateSpace.
SpaceInformation
Provides routines typically used by motion planners; combines the functionality of classes it depends on.
StateValidityChecker
Decides whether a given state from a specific StateSpace is valid.
ValidStateSampler
Provides the ability to sample valid states.
OptimizationObjective
Encodes a path cost that is used by optimizing planners.
Path
Representation of a path; used to represent a solution to a planning problem.
SimpleSetup
Provides a simple way of setting up all needed classes without limiting functionality.
ProblemDefinition
Specifies the instance of the planning problem; requires definition of start states and a goal.
Planner
Solves a motion planning problem.
Goal
Representation of a goal.
User code
User must instantiate this class.
User must instantiate this class unless SimpleSetup is used.
User can instantiate this class, but defaults are provided.
A→B

Figure 3.13: Visual hierarchy of high-level components of OMPL

Next, several simulations using the Informed RRT* were designed to test the capabilities of the method in different 3D environments. The division was made in 3 distinct scenarios: a maze environment, a constrained object course and a more relaxed object course which can be visualized in the figures below (the environments are represented here using **Gazebo** software for better clarity). For each environment, the results were averaged over 50 simulations. Since both the RRT* and the Informed RRT* are methods which converge to an optimal solution with the increase in time, different thresholds were imposed to test for the pseudo-optimallity achieved. This evaluation metric was defined as a measure of how close to an optimal path was achieved in the defined amount of time. Formally it can be written as the quotient between the length of the path averaged over the tests and the minimum path length obtained from setting an unreasonably high time threshold for the given environment (greater than 30 minutes) : $\frac{\overline{PathLength}}{BestPathLength} \times 100\%$. It must be noted that while the applied metric can not be classified as giving an absolute measure of optimality, it comes as close to an optimallity guarantee as possible. In all simulations, the quadrotor was modeled as a box whose representation can be visualized below. The models dimensions are based on the **3DR Iris+** quadrotor, an autonomous commercial quadrotor with a compact and durable design.



(a) Framework quadrotor internal representation

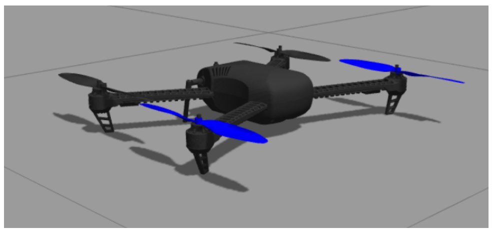

(b) Gazebo representation of 3DR Iris+

Figure 3.14: On the left, we can see two bounding boxes around the quadrotor. The smaller green box fits to the quadrotors dimensions but to perform collision checking when sampling points from the configuration space, the larger red representation with dimensions shown (from left to right length, width and height) is used to provide safety margins. On the right, the model of the 3DR Iris represented in Gazebo

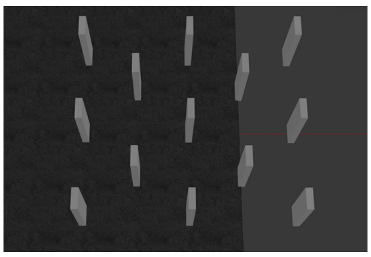


Figure 3.15: Maze environment (Pseudo-Optimallity of 15.2 meters)

In the first environment similar to an obstacle maze, although there are a considerable number of obstacles and overall map area size is small (15m width an 15m length), there is enough space between them which does not weigh as much both algorithms as if the obstacles were of a larger size. Comparing both methods along each timeout , we see that as expected the Informed RRT* achieves much better convergence and solutions.

Timeout	0.1s	0.5s	1s	5s
RRT*	24%	65%	97%	99%
Informed RRT*	54%	94%	99%	99%

Table 3.1: Pseudo-Optimality for each method with timeout variation (maze environment)

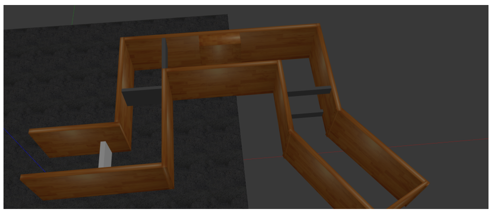


Figure 3.16: Constrained Obstacle Course (Pseudo-Optimallity of 24.8 meters)

The second environment and third environments are similar in the sense that both are corridors with narrow corners obstacles limiting the passage at certain points. The two major differences between them are map size, 25m width and 25m length for the second , 40m width and 40 length for the third and obstacle clearance which is much higher for the third environment (in the second environment obstacles impose narrow passages while in the third we the quadrotor has a lower degree of difficulty navigating between the corridors). These differences are reflected in the results for both methods. We can see that in both environments, RRT* can not converge to a solution in a reasonable amount of time and when it achieves a solution the degree of pseudo-optimality is very low (under 60%). The Informed RRT* on the other hand always converges even when the timeout is set to the lowest value tested (0.1s) at a significant cost for the solution (low pseudo-optimality). Increasing the timeout slightly wields significantly better results and the method achieves in both instances near perfect pseudo-optimality for an execution time of just 1s.

Timeout	0.1s	0.5s	1s	5s
RRT*	—	—	35%	60%
Informed RRT*	35%	61%	99%	99%

Table 3.2: Pseudo-Optimality for each method with timeout variation (Constrained Obstacle course)

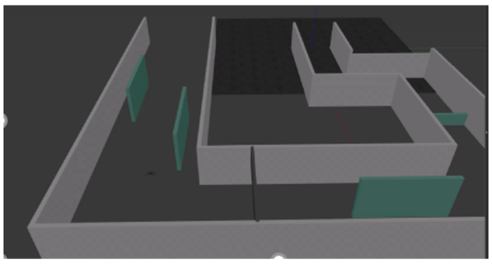


Figure 3.17: Relaxed Obstacle Course (Pseudo-Optimallity of 61.15 meters)

Timeout	0.1s	0.5s	1s	5s
RRT*	—	—	—	53%
Informed RRT*	36%	59%	95%	99%

Table 3.3: Pseudo-Optimality for each method with timeout variation (Relaxed Obstacle Course)

3.4.2 Trajectory Generation

Until now in the path planning solution presented for the offline component of the framework, there was no mention of the constraints quadrotor dynamics might impose in the curvature of the generated paths as well as kinematics. If we were to generate a path which can not be physically followed by a quadrotor, that would render the entire work done here useless. Having this into account, the next step is to go from the generated path to a flyable trajectory which must have among other requirements a high degree of smoothness. The optimization strategy followed was based on generating smooth flyable trajectories. In [28], the authors main objective was to generate smooth trajectories that required as minimum control effort as possible. This was made possible due to the concept of differential flatness introduced by the same authors in the work. As mentioned in chapter 2, the control inputs are represented as functions of some trajectory derivatives. These concepts play a crucial role in the optimization method followed here which is based on the work done in [77] where the authors develop a method that allows the generation of high-quality minimum-snap piece-wise polynomial trajectories based on jointly optimizing polynomial path segments. Through this method the need for computationally expensive kinodynamic (where kinematics and dynamic constraints restrict the available range of motion during sampling) path planning is eliminated. In [82], the previous work is further expanded to include non-linear optimization features. The authors turn the initial linear optimization problem as in [77] into a nonlinear optimization problem by modifying some optimization variables. The present work makes use of this latter extension of the method and it will detailed next as well as its integration on the framework.

Optimization

From the path planning component, we obtained a sequence of waypoints representing a path in the configuration space. Let's assume the following generic path.

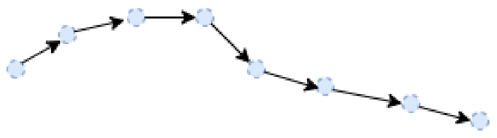


Figure 3.18: Generated path. The blue dotted circles are the vertexes and the black arrows represent the edges between them.

One of the main problems behind trajectory parametrization arises from considering the straight line representation from the generated path. In this case, smoothness would not be possible since there would be discontinuities in the trajectory's derivatives at each waypoint. To circumvent this, there are several candidate curves such as cubic polynomials (splines) or other higher order polynomials (quintic) that can be applied. Polynomials trajectories are computed efficiently as a solution to quadratic optimization problems where the cost function to minimize features some of the trajectory's derivatives. They can be be jointly optimized while maintaining continuity of the derivatives up to arbitrary order, therefore ensuring motion smoothness by preventing control input oscillations.

The main objective behind the optimization method is to find a smooth piece-wise polynomial trajectory that is anchored in the path waypoints and fits as close to the straight line between them as possible. Polynomials are expressions built from addition, multiplication and exponentiation (to non-negative integer power) of constants and symbols called variables. Formally, any polynomial P with n coefficients can be written as:

$$P(x) = \sum_{k=0}^{n-1} c_k x^k = c_{n-1}x^{n-1} + ... + c_2x^2 + c_1x + c_0 \tag{3.15}$$

where $c_0, c_1, c_2..., c_{n-1}$ are constants and x is the indeterminate or variable. The value of $n-1$ defines the order of the polynomial. This can also be written as the product between two vectors:

$$P(x) = x \cdot c \tag{3.16}$$

where $x = [1, x, x_2..., x_{n-1}]$ and $c = [c_0, c_1, c_2..., c_{n-1}]^T$.
A piece-wise polynomial trajectory consists of M continuous polynomial $P(x = t)$ segments where each individual polynomial is valid from $t = 0$ until the segment duration $t = T_{s,i}$ ($i = 1, 2, ..., M$ denotes the corresponding segment). Each segment is composed of D polynomials where D is the number of dimensions of our configuration space. We consider that the quadratic cost function J for each individual trajectory segment i in a given dimension d can be written as a combination of the polynomial and associated derivatives:

$$J_{i,d} = \int_0^{T_{s,i}} \sum_{j=0}^{n-1} (c_0)_{i,d} P_{i,d}(t)^2 + (c_1)_{i,d} P'_{i,d}(t)^2 + (c_2)_{i,d} P''_{i,d}(t)^2 + ... + (c_{n-1})_{i,d} P^{(n-1)}_{i,d}(t)^2 \, dt \tag{3.17}$$

Depending on the derivative we want to minimize throughout the trajectory, all derivatives coefficients up to that derivative and with higher order are set to zero. In this work, the authors choose to minimize the snap. Effectively we are left only with $J_{i,d} = \int_0^{T_{s,i}} (c_4)_{i,d} P''''_{i,d}(t)^2$. The result with any other choice of derivative to minimize is analogous. We can condense this cost by using algebraic notation:

$$J_{i,d} = c_{i,d}^T \cdot Q(T_{s,i}) \cdot c_{i,d} \tag{3.18}$$

where $c_{i,d}$ is the vector of coefficients of the polynomial valid for segment i in dimension d and $Q(T_{s,i})$ is an *Hessian* matrix which contains the values derived from differentiation of the square of the polynomial with respect to each of its coefficients. The values this matrix exhibits are constant over all dimensions for a given segment, thus eliminating the need to compute this matrix for as many times as there are dimensions.
The quadratic cost for the whole trajectory can be now written as the sum of the cost for each trajectory segment in each dimension:

$$J_{total} = \sum_{i=1}^{M} \sum_{d=1}^{D} J_{i,d} \tag{3.19}$$

With the objective function defined we can focus on the imposed constraints (velocity, acceleration, jerk, snap and higher order derivatives) related to polynomial optimization. For example, we can impose specific derivative values at certain endpoints to ensure that the system is moving at a certain velocity at that point. They can be formulated as follows:

$$A_{i,d}c_{i,d} = d_{i,d} \quad , \quad A_{i,d} = \begin{bmatrix} A(t=0) \\ A(t=T_{s,i}) \end{bmatrix}_{i,d} \quad , \quad d_{i,d} = \begin{bmatrix} d(t=0) \\ d(t=T_{s,i}) \end{bmatrix}_{i,d} \tag{3.20}$$

$A_{i,d}$ is a mapping matrix consisting of row vectors t and $\frac{d}{dt}t$ between the coefficients $c_{i,d}$ and derivatives $d_{i,d}$ at the start and endpoints of a polynomial segment. Similarly to $Q(T_{s,i})$, the mapping matrix values only depend on the segment time and are valid across multiple dimensions (from now on we'll omit the notation related to the different dimensions due to this).

There is one constraint that is always required even when specific constraints are not enforce. Derivative continuity between segments must be set through all the trajectory segments since a polynomial is infinitely derivable on its interval. However, for the sake of feasibility we can only guarantee these constraints up to a given order (in this method up to the chosen order of the derivative to minimize). Formally, this can be written as:

$$A_{T,i}p_i = A_{0,i+1}p_{i+1} \tag{3.21}$$

where the left side of the equation represents the constraints at the endpoint of segment i^{th} and the right side the same constraints at the beginning of the $i+1^{th}$ segment.
Given the total quadratic cost function J_{total} and all of the imposed constraints $A_{total}p_M = d_M$, this is an example of a constrained quadratic optimization problem. However, the authors in [77] reformulate this as an unconstrained quadratic optimization problem since obtaining a viable solution in a reasonable time-frame becomes unachievable due to the computational inefficiency when the number of segments increases or the chosen polynomial chosen are of higher order. Unlike the original optimization problem where there are specified constraints that must be enforced to reach a solution, in the reformulated unconstrained problem, the constraints are incorporated in the objective function as penalization terms. This compromise allows the problem to remain bounded by constraints (although "soft" instead of "hard") while reducing the computational time in finding a feasible solution. In this particular problem, polynomial coefficients are dropped as the decision variables in favor of the endpoint derivatives derived from the constraints. After obtaining the solution to the unconstrained problem, we still need to find the minimum-order polynomial that connects the waypoints which is done by inverting the constraints matrix. We start by replacing the constraints into the cost function:

$$J_{total} = \begin{bmatrix} d_1 \\ \vdots \\ d_M \end{bmatrix}^T \begin{bmatrix} A_1 & & \\ & \ddots & \\ & & A_M \end{bmatrix}^{-T} \begin{bmatrix} Q_1 & & \\ & \ddots & \\ & & Q_M \end{bmatrix} \begin{bmatrix} A_1 & & \\ & \ddots & \\ & & A_M \end{bmatrix}^{-1} \begin{bmatrix} d_1 \\ \vdots \\ d_M \end{bmatrix} \tag{3.22}$$

Since there was a distinction made regarding the two main types of constraints, the endpoint derivatives are split into two groups: fixed/specified derivatives (d_F) and free/unspecified derivatives (d_P).

This leads to a reformulation of the previous cost function (C is a matrix assembled of ones and zeros used to accomplish this):

$$J_{total} = \begin{bmatrix} d_F \\ d_P \end{bmatrix}^T CA^{-T}QA^{-1}C^T \begin{bmatrix} d_F \\ d_P \end{bmatrix} \tag{3.23}$$

In [82], a first improvement is made by exploiting the structure of the mapping matrix A for each segment in order to facilitate the inversion process required to solve the problem:

$$A(t=0) = \left[\tfrac{d^0}{dt^0}t(0)^T \quad \dots \quad \tfrac{d^{N/2-1}}{dt^{N/2-1}}t(0)^T\right]^T \tag{3.24}$$

$$A(T_{s,i}) = \left[\tfrac{d^0}{dt^0}t(T_{s,i})^T \quad \dots \quad \tfrac{d^{N/2-1}}{dt^{N/2-1}}t(T_{s,i})^T\right]^T \tag{3.25}$$

$$A^{-1} = \begin{bmatrix} A(t=0) \\ A(t=T_{s,i}) \end{bmatrix} = \begin{bmatrix} \sum & 0 \\ \Gamma & \Delta \end{bmatrix} \tag{3.26}$$

Stacking the mapping matrices A to form the joint optimization problem leads to $\sum$ being a diagonal matrix , Γ an upper diagonal matrix and Δ is the only non sparse matrix since only the constant parts of t and its derivatives are non-null as opposed to $A(t=0)$ where the segment time at the beginning is zero. The matrix is easily inverted since all the sub-matrices involved either contain fewer high powers of t like Δ or do not contain them at all $\sum$. Using the *Schur-Complement*, we obtain:

$$A = \begin{bmatrix} \sum^{-1} & 0 \\ -\Delta^{-1}\Gamma\sum^{-1} & \Delta^{-1} \end{bmatrix} \tag{3.27}$$

The initial formulation of the objective function attaches the assumption that segment times are known and fixed, only the endpoint derivatives are unknown variables. It is however important in the context to reduce as much as possible the total combined cost of the segment times since that is equivalent to reducing the duration of an autonomous mission. To do this, besides incorporating segment times $(T_{s,1}, T_{s,2}, ...T_{s,M})$ in the cost matrix as unknown variables, the authors of [82] also incorporate a new quadratic term that measures the cost of segment times in the initial cost function:

$$J_{new} = J_{total} + k_T \cdot (\sum_{i=1}^{M} T_{s,i})^2 \tag{3.28}$$

This new formulation aims to achieve the ideal trade-off between smoothness and trajectory duration. k_T is a parameter that controls this trade-off (higher values place a deeper importance in minimizing trajectory duration) and depends on the specific requirements of each individual case (it is always a good practice to keep a balance between both). One additional change brought by the reformulation is that unlike before where the optimization problem featured only linear constraints and the objective function was itself linear, the inclusion of the segment time constraint turns it into a non-linear problem since the cost matrix involves powers of t.

One particular constraint explored by the authors is related to imposing global constraints related to velocity or acceleration since in the real world a quadrotor (as well as any other mechanical system) is limited by the maximum thrust capacity of its rotors. These inequality constraints raise a problem since as explained we would have to guarantee that every point in the trajectory would not surpass the constraints and that would lead to an unfeasible amount of constraints in the problem. To circumvent this problem, the authors formulate these as soft constraints instead of hard constraints and adding an additional cost term in the objective function. They justify this change by empirically showing that even when they are formulated as hard constraints, they tend to not be enforced by the optimization methods while simultaneously exponentially increasing the required computation time. The soft cost is formulated as follows for an arbitrary maximum value of a given constraint x_{max}:

$$J_{soft} = exp(\frac{x_{max,actual} - x_{max}}{x_{max} \cdot \epsilon} \cdot k_s) \tag{3.29}$$

ϵ defines the acceptable deviation from the maximum and k_s is a constant that in a similar way to k_T defines the weight of the soft constraint on the overall cost. We can now write the full cost function to minimize in the approach followed in this work as:

$$J = J_{new} + J_{soft,velocity} + J_{soft,acceleration} \tag{3.30}$$

where a maximum velocity $v < v_{max}$ and maximum acceleration $a < a_{max}$ constraints were imposed. The use of this method was made possible through an implementation of both works available in [83]. The authors of this implementation are the one's who introduced the changes above described to the original optimization method. This implementation makes use of *NLopt* which is an open-source library that contains the necessary routines to solve (among others) unconstrained non-linear optimization problems. In particular, the algorithm used is adapted from BOBYQA [84] (Bounded Optimization BY Quadratic Approximation) and allows for derivative-free bound-constrained optimization through iteratively approximating quadratically the objective function. For the sake of simplicity the inner workings of this algorithm won't be detailed. The overall implementation will be briefly detailed next, coupled with some alterations made to the source code as well as some remarks to a possible change in the optimization goal.

Before starting the analysis of the algorithm below, there is need to define clearly the concept of trajectory presented at the beginning of this chapter, in particular the parametrization that characterizes it. Each state is our trajectory is composed by $s_i = [q(t_i), \frac{dq(t_i)}{dt}, \frac{d^2q(t_i)}{dt^2}, \frac{d^3q(t_i)}{dt^3}, \frac{d^4q(t_i)}{dt^4}]$ corresponding respectively to a position in the configuration space (p), the velocity (v), acceleration (a), jerk (j) and snap (sn). Of all these elements, only the first three will be used later to provide guidance to the quadrotor in real-world simulation tests as we will explain. However, the fact the latter components are not directly used by the system does not invalidate the benefits behind the optimization objective.

The algorithm starts when we have a path generated that is collision-free represented by the tree structure $T(V, E)$. Several parameters needed for the optimization must be set including the optimization derivative goal, the configuration space dimension, maximum velocity and acceleration state constraints.

From the path configuration, a list of ordered waypoints is created to serve as the basis for the joint polynomial optimization to pass through these. If there are any endpoint derivative constraints required at any waypoint they are also set during the creation of the list. Finally, before initiating the optimization algorithm, additional parameters must be set such as: the maximum number of iterations $maxiterations$ the algorithm has to achieve a solution and to subsequently improve upon it; f_{rel} which determines the stopping criteria of the optimization that is when a new solution represents an improvement in the cost function lower than a defined threshold relative to the previous solution, the optimization stops and it is considered that a global optimal solution has been achieved; x_{rel} has a similar effect to the previous parameter but the threshold is on the values of the optimization variables; $time_{pen}$ the parameter that controls the penalty associated with the segment time (in equation 3.28 it appears as k_T); $init_{step}$ value determines how much of the initial guess is taken as the initial step size; $tolerance$ consists of two values that measure the acceptable deviation from equality and inequality constraints. There is a timeout that was created to prevent the optimization from exceeding a defined time threshold. Since the algorithm used to solve the optimization problem works by iteratively improving a solution, it first needs an initial guess and for that we have to provide an estimate of the segment times between each waypoint.

Algorithm 7: Pseudo algorithm for generating a trajectory from a path configuration

```
1  function TrajectoryGeneration ($T(V, E)$);
   Input  : Path configuration
   Output: Trajectory configuration
2  $v_{list}$ ← list of ordered vertexes for optimization;
3  $sampledTrajectory$ ← list containing trajectory states;
4  $opt$ ← optimization object;
5  $trajectoryparameters$ ← structure containing necessary trajectory parameters for optimization;
6  $num_{states}$ ← getStateNumber($T(V, E)$);
7  $trajectoryparameters.dimension$ ← setSpaceDimension();
8  $trajectoryparameters.derivativeToOptimize$ ← setDerivative();
9  for $num_{states}$ do
10     $v_{list}$ ← getNextVertex($T(V, E)$);
11     if $newconstraint$ then
12         $v_{list}$ ← addConstraint($newconstraint$);
13     end
14 end
15 $trajectoryparameters.maxVelocity$ ← setMaxVelocity();
16 $trajectoryparameters.maxAcceleration$ ← setMaxAcceleration();
17 $trajectoryparameters.segmentTimes$ ← estimate($v_{list}, maxVelocity, maxAcceleration$);
18 $opt.optimizationparameters$ ← setupOpt($maxiterations, f_{rel}, x_{rel,time_{pen}}, init_{step}, tolerance$);
19 $opt.trajectoryparameters$ ← setupTraj($v_{list}, trajectoryparameters$);
20 while $!solutionNotFound$ do
21     $opt.solution$ ← optimize();
22 end
23 $trajectoryparameters.interval$ ← getTimeInterval()
24 while $!endofSolution$ do
25     $sampledTrajectory$ ← sample($opt.solution$);
26 end
27 return $sampledTrajectory$
```

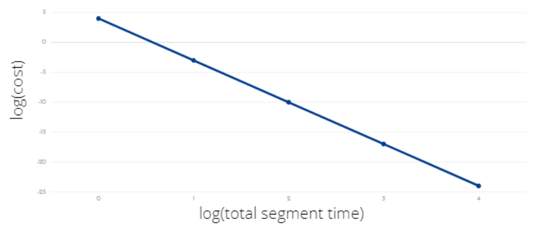
To do that the preferred method used in this thesis can be written as:

$$T_{s,i} = 2\frac{d_i}{v_{max}}\left(1 + \gamma\frac{v_{max}}{a_{max}} \cdot e^{2\frac{d_i}{v_{max}}}\right) \tag{3.31}$$

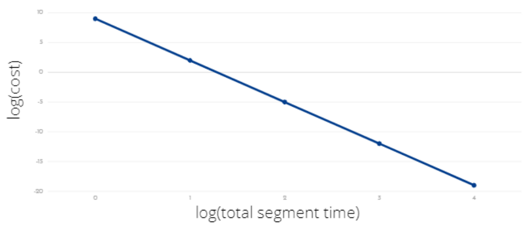
where d_i is the straight-line distance between waypoint at time t_i and the next waypoint at time t_{i+1} and γ is a constant with a defined value of six. This is not the common approach where we would assume velocity between waypoints to be calculated under the assumption of constant maximum acceleration and calculate segment times as the straight-line distance between waypoints divided by this velocity.

With this, we can start the optimization algorithm until a solution is found or until a defined timeout is met (this was added to the original code which did not present this important safety mechanism). The optimizer returns a theoretical infinite trajectory (containing as many states as we want) and therefore the last step is to sample consecutive equally time-spaced states to obtain a usable trajectory. The sampling interval is initially set to a low value to obtain sufficient states to perform periodic safety checks during flight. This is specially important to avoid sudden collisions with dynamic obstacles.

While in the initial stages of testing this implementation, an incoherence was found on the messaging forum in **Github** (where the code was pulled from) while searching for a better comprehension of the implementation. The cost associated with the segment times was chosen as a quadratic term $k_T(\sum_{i=1}^{M} T_{s,i})^2$ but it was suggested that such implementation could be improved by considering that the cost associated with segment times should be $k_T(\sum_{i=1}^{M} T_{s,i})^7$. To prove this, a path composed of 10 random waypoints was chosen to generate a trajectory from and another experience to corroborate the previous results was made for another path of 20 waypoints. The derivative chosen to minimize was switched between acceleration, jerk and snap with no difference in the outcome of the simulations. In order to see the expected behaviour of the total cost without influence of the segment times cost, parameter k_T was set to zero and the total segment time ($\sum_{i=1}^{M} T_{s,i}$) was fixed for each simulation. Several simulations were made and the total cost was plotted against different values of total segment time.



(a) 10 waypoint path



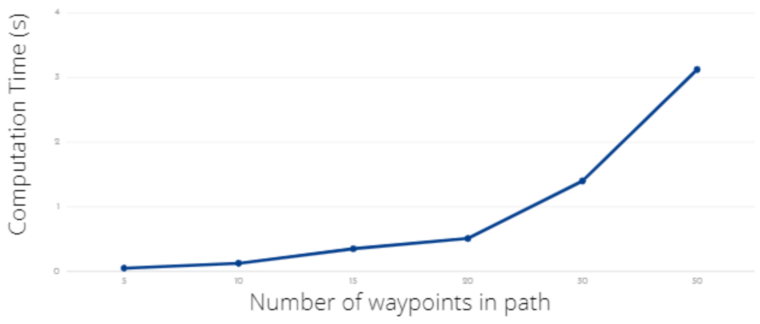
(b) 20 waypoint path

Figure 3.19: Both graphs represent the evolution of the total cost with the total segment time. A logarithmic scale was applied for better understanding.

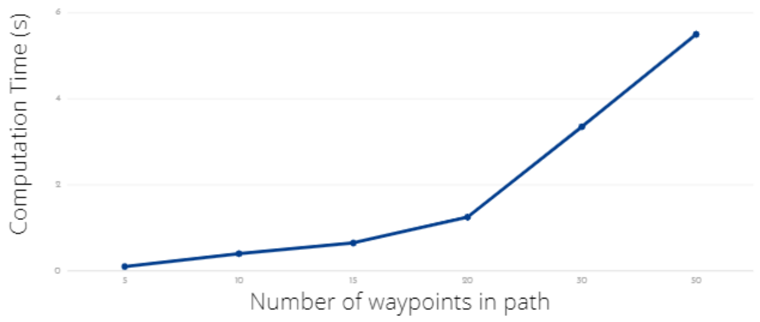
From the analysis of both graphs (we applied a logarithmic scale), we can see a clear tendency of the cost decreasing according to a linear polynomial of slope 7. This suggests that the non-log cost is a function of $k_T(\sum_{i=1}^{M} T_{s,i})^7$. This change was made after using the implementation and a small decrease in computational time as well as an improvement in solution quality was verified.

Jerk vs Snap

While the implementation used here was developed under the assumption of a minimum snap piecewise polynomial trajectory, it is also true that it allows for a seamless switch between minimizing the snap or any other derivative. Until now, despite mentioning that minimizing the total snap in a trajectory is directly related to the smoothness of the same trajectory (we equate smoothness to minimizing the control effort necessary for the quadrotor to follow a trajectory), we have not explored existing alternatives which consider that we could also achieve smooth polynomial trajectories by minimizing the third derivative of position, the jerk. It is strongly linked to the rotation speed [85] and the oscillations in acceleration of a quadrotor, both of which we want to keep as low as possible in a trajectory since they are part of the notion of smoothness. By considering the higher order derivative in the snap, we further add another level of smoothness to the trajectory by lowering the control input oscillations in a trajectory. This comes at a cost of additional constraints and variables to optimize, thus increasing the computational time of convergence to a solution. The impact of the increase in computational time is specially important in the context of a real-time trajectory generation method and it was important to study it before making a decision on the order of derivative to minimize. In a similar way to the previous simulations, the two different optimization goals were defined and for each one the number of waypoints in the path to optimize was varied.



(a) Jerk



(b) Snap

Figure 3.20: Both graphs represent the evolution in computation time in seconds with the number of waypoints. The results are averaged over 50 simulations for each.

As expected there is a noticeable difference in computation time between both implementations, with the snap case averaging computational times around twice as large as the ones for the jerk. We can also see the exponential jump in both cases when the number of waypoints increase which is additional to factor in since for the jerk the prohibitive range in computational time happens later. Nonetheless, even when the number of waypoints can be reduced in a particular case, this is something that can be mitigated by incrementally optimizing the path. Here, a decision was made to employ the first method only when the number of waypoints became prohibitive and using the snap minimization as the preferred method. (In experimental tests, when dealing with large or cluttered environments the number of waypoints never exceeded a prohibitive value)

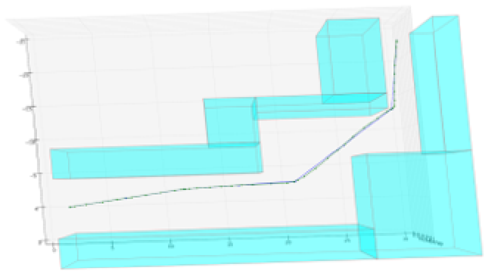
Trajectory Results

With all the associated problems taken care of, we are able to present the full result from the Offline Pre-Flight Path Planning. The following parameters were set:

Max Iterations	25000
Objective Function Threshold (f$_{rel}$)	5%
Optimization Variables Threshold (x$_{rel}$)	10%
Time penalty (k$_{T}$)	2000
Initial Step Size	10%
Tolerance (ϵ)	5%
Sampling Interval (Δt)	0.1s

Table 3.4: Optimal parameter values for the optimization

Several optimization parameters were tuned to apply for the large majority of situations such as the maximum number of iterations which was originally set at 10000 but in order to improve the quality of the solution it was subsequently increased until reaching 25000, that proved to be the best compromise between computational time and solution quality. Experimentally it was verified that further increasing the number of iterations did not wield any significant improvements. The values of the thresholds that define the stopping criteria of the optimization also contribute to the fact of not further increasing the number of iterations since at times they indirectly define the maximum number of iterations of the algorithm. The step size was kept at default value as well as the tolerance since they wield better results. k_T is as expected a major factor in the solution type and again we want a trajectory whose duration is as low as possible but also smooth, without steep changes in velocity and acceleration. Choosing a value between lower than 1000 lead to slow trajectories while values higher than 5000 originated the opposite with hard changes in motion, which is specially concerning in cluttered environments since trajectory tracking in real-world might be unfeasible in such conditions. A value of 2000 was chosen to prevent these issues. Despite the fact that the generated trajectory is a continuous function, in practical applications we have to consider that the changes between trajectory states happen at discrete time steps and therefore from this continuous function we sample equally time-spaced states with a sampling interval set to $0.1s$. This value was later adjusted when applying the framework in real-world simulation software. The choice of this parameters resulted in the following simulations where maximum velocity was kept at $10m/s$ and maximum acceleration at $8m/s^2$. To visualize the generated trajectories and environments, an auxiliary tool was employed. Although there are differences between the environments for the simulations, they follow more or less the same basic architecture.


(a) High

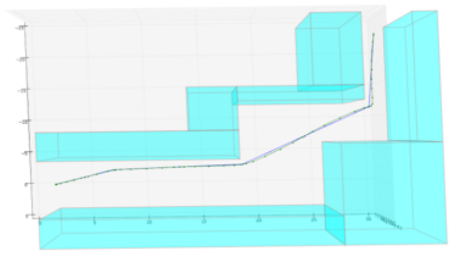

(b) Snap

Figure 3.21: In both graphs, the path is represented in blue (generated from the Informed RRT*) and the resulting trajectory (obtained from the optimization method) in green.

In both simulations, a $9th$ order polynomial that represents the trajectory fits tightly to the original straight-line segments that form the path. Small offsets can also be observed when turning corners which is to be expected. State constraints in velocity and acceleration are met. The only difference in these two simulations comes from the original path generated.

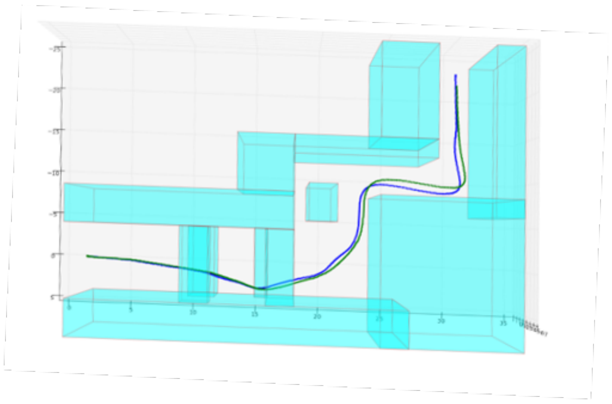


Figure 3.22: Two distinct trajectories resulting from the same path under identical parameters. (The path generated from the Informed RRT* is omitted)

This simulation serves to demonstrate the unpredictable nature of the optimization. While the parameters are the same as well as the original path, the resulting trajectories are slightly different.

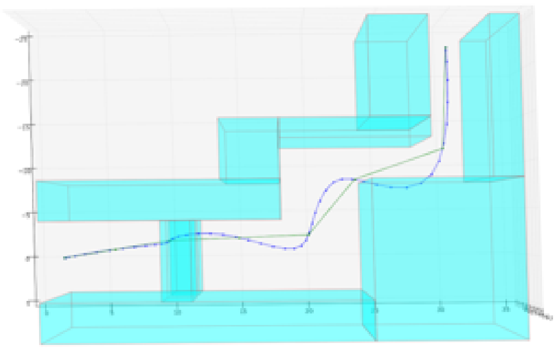
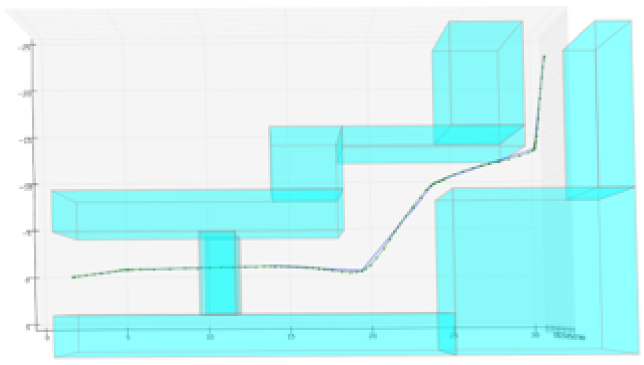
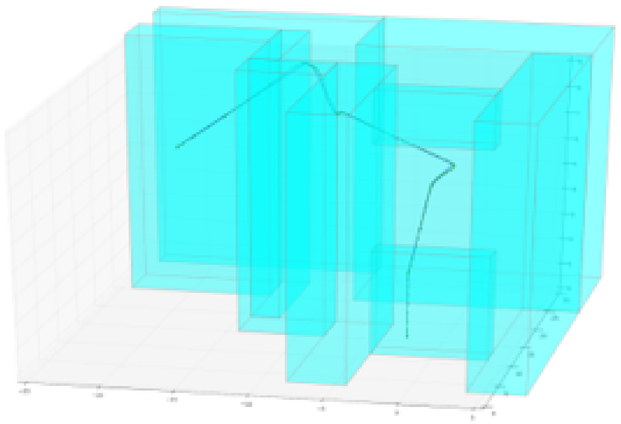


Figure 3.23: Generated trajectory in blue is in collision with an obstacle. The original path in green.

While the optimization tries to find the polynomial anchored to the waypoints, it might result in a trajectory that violates feasibility constraints by having some states of the trajectory placed directly inside obstacles. This happens particularly in highly congested environments if the number of waypoints of a path is too low. To prevent this problem from happening the number of segments in a path is processed after its generation. The length of each segment is checked to determine if its under a threshold (defined according to the total length of the path) and a waypoint is added to the middle of each segment that violates the threshold. The authors in [77] explored a solution to this problem based on adding waypoints to the middle of each segment after a trajectory is in collision with an obstacle but by doing so besides preemptively not avoiding a problem, by not having a more restrict criteria in adding waypoints to the path, they are unnecessarily increasing the computational time of the optimization process.



(a) Top view



(b) Isometric view

Figure 3.24: Corrected trajectory in blue is now feasible and shows a closer adherence to the original path in green.

By applying the previously explained process, the trajectory is now feasible. Another possible method would be to increase the safety distance to the obstacles in the original path by increasing the size of the bounding box (seen in figure 3.16) that represents the quadrotor. However, in cluttered environments that could easily lead to the inability of finding an initial path, though in environments with less obstacles it could be a viable solution.

3.5 Online Real-Time Path Replanning

At this point in the framework the quadrotor has a feasible trajectory that can be used to put it on course to its goal. As explained, despite having complete information on the location of static obstacles within the environment, due to the possible presence of dynamic obstacles a correction on the current trajectory might be needed. The following flowchart illustrates the integration of this in the framework.

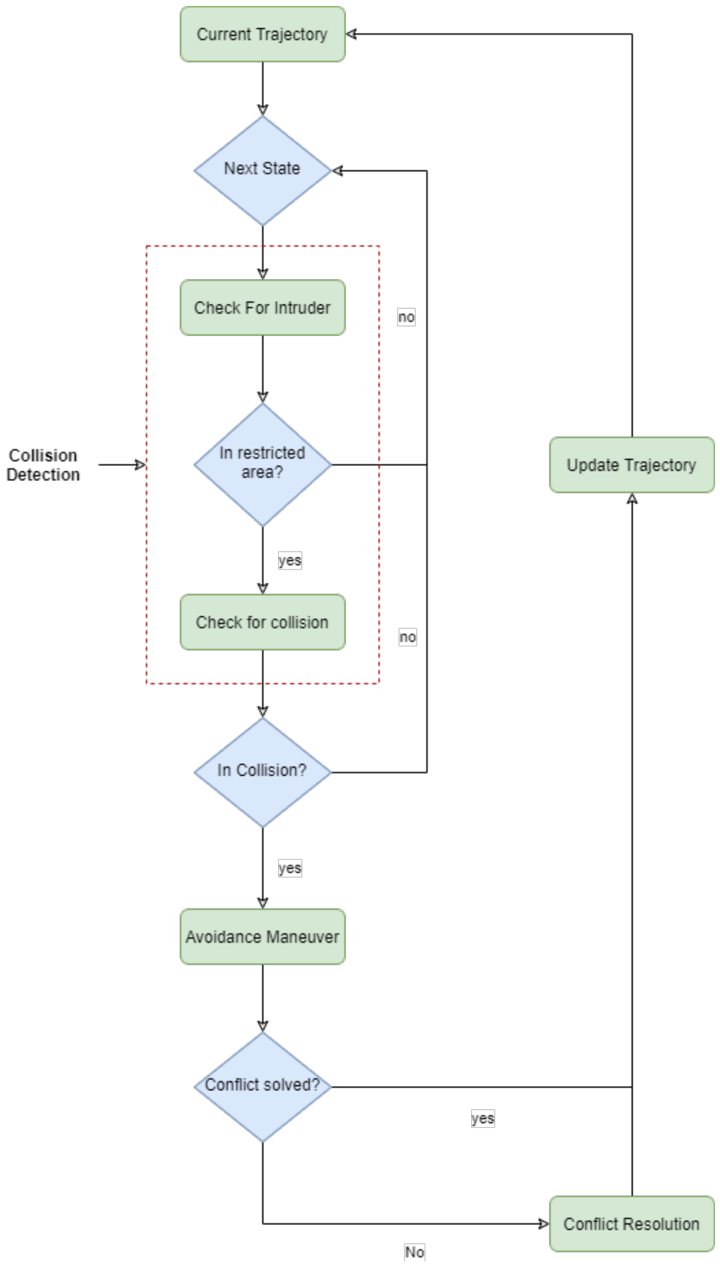


Figure 3.25: Online Real-Time Path Replanning proposed solution. This represents a rough overall description with each component being described in detail throughout this chapter

The replanning component is activated immediately after the quadrotor takes flight. A **Collision Detection** routine happens at each time step k when the next state in the trajectory s_i is given to the quadrotor. This routine checks for the presence of intruders in the environment and updates in the process the configuration space C_i which is now time dependant. The update process is made according to their present and perceived future locations (the method will be detailed in depth). A collision query results from between the affected states in the trajectory by the changes in the configuration space. The most imminent collision is treated with the highest priority and avoidance maneuvers in the form of a check and repair strategy are triggered. It is based on the adaptation of the previous fast trajectory generation method to deal with sudden changes in the dynamic environment. If the avoidance is successful, the trajectory is updated and the replanning process is reinitiated at the next time step. The replanning problem can be adapted from the formulations presented at the beginning of the chapter:

$$h : [0, 1] \rightarrow (C_{free})_{t_i} \tag{3.32}$$

where the free space C_{free} must reflect the dynamic changes $C_{dif} = C_{t_i} - C_{t_{i-1}}$ in the environment up to a determined time step i.

This replanning routine developed works under the assumption that the intruder is:

- **non-adversarial** by not actively pursuing the autonomous agent nor engaging in similar separate avoidance maneuvers as if it were an autonomous agent;
- **non-cooperative** since their operation does not fall under the same set of rules and therefore we do not know their future behaviour or state in the future;
- **detectable** within the detection ranges of equipped onboard sensors;

These characteristics are important to narrow the scope of the work since in the particular case of active adversarial agents, the avoidance protocols would require a much more robust approach.

Collision Detection

Intruder detection and tracking has to be done in order for the framework to function as a whole despite being another entire field of study in autonomous navigation. A careful investigation was made to better outline the real-world capabilities of current detection and tracking systems so as to validate the framework under those conditions. It is shown in [86] that current state of the art algorithms can be employed onboard UAVs equipped with relatively small sized cameras and achieve accurate real-time detection and tracking of dynamic obstacles. The paper outlined the specific case of detection and tracking of bicycles in a cycling race. The authors demonstrate that it is possible to achieve real-time accurate performance as long as the framerate of the camera sensors is kept at a reasonable value. This allows for real-time collision checking with dynamic obstacles. The results from the several tested algorithms were obtained using lightweight embedded processing devices (such as the **Jetson TX2**) capable of being attached to small quadrotors. With this in mind, the detection and tracking process is simulated in the framework by a periodic access to the position, velocity and overall dimensions of intruders when they are within line of sight of the sensing system. This happens at each time step k when the next state in the trajectory s_i is given to the quadrotor. As mentioned the dynamic environment update implies that we have partial knowledge of the state of intruders up to a certain time instance in the future. Obviously, since we do not have direct access to this, it is necessary to have a substantiated guess of those parameters that can lead to an accurate depiction of the future motion. For that reason, at any given instance in time t_i, the state of each intruder Obs_i is defined over the next k time steps by extending the current motion from the last known location. Formally this can be thought of as:

$$Obs_{i+k} = [p_{obs}(t_i) + k \cdot v_{obs}(t_i), v_{obs}(t_i)] \tag{3.33}$$

This prediction implies that the velocity at that time $v_{obs}(t_i)$ remains the same for the defined time horizon, which must be as short as possible for this assumption to remain accurate. In instances where the relative velocity $v_{rel}(t_i) = v_{agent}(t_i) - v_{obs}(t_i)$ has high values, it might not be possible to detect a collision in a feasible time frame. Safety margins were presented before in figure 3.16 and similarly to its use for generating trajectories, they are also employed when checking for potential collisions with intruders. Due to the high dynamic nature of intruders, the safety margins for this process are increased by 15% of their previous maximum value to provide further robustness. From this, a collision query is obtained and ordered according to the highest priority (closest in time) collision. At the point of collision, the octomap is updated according to algorithm 2 as well as the locations immediately following this point considering the current course of motion of the intruder.

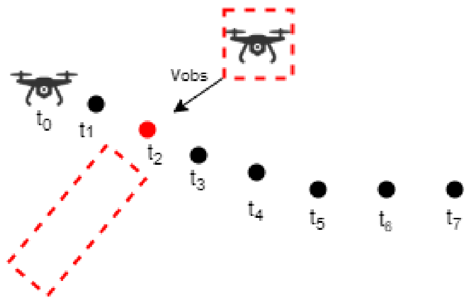
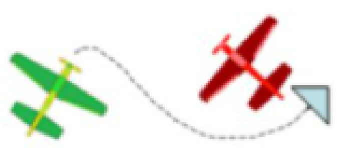


Figure 3.26: Visual representation of collision detection situation. A state in the agent's trajectory is in collision course with the intruder. The internal map representation is updated by extending the motion of the intruder after the hypothetical collision.

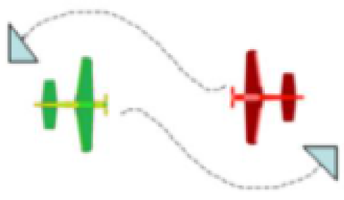
This is done in order to prevent the replanning algorithm from generating trajectories that cross the future motion of the intruder as it will be seen next.

Avoidance Maneuvers

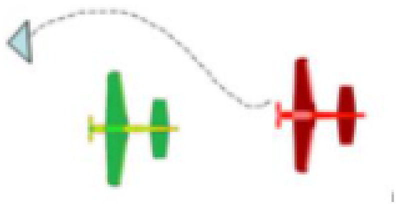
In airspace, avoidance maneuvers between aircrafts must follow a set of universal *Rules of the Air* which can be visualized below.



(a) Converging



(b) Approaching Head-on



(c) Overtaking

Figure 3.27: Rules of the air for collision avoidance distinct scenarios.

There are three distinct cases illustrated:

- In the first case, when **converging** the aircraft with the other on the right shall give way to the other;
- The second case is the **head-on** scenario, where each aircraft should turn to the right;
- The third case consists of an **overtaking** maneuver by a faster aircraft. This maneuver should be done by the right side of the slower aircraft.

These rules were designed with the intent to be applicable to cooperative avoidance where the intruder is a cooperative aircraft. Since we are treating the intruder as non-cooperative, these rules are not applicable and the autonomous agent must at seek an alternative trajectory in case of collision.

The first step of the check and repair approach was already presented and consists of identifying the possible collisions and updating the dynamic map already detailed. A similar trajectory generation method is employed. However, we want to decrease as possible the computation time of the trajectory while retaining smoothness and a relative short length. The latter was achieved by sampling a short path from start to goal using the Informed RRT*. The benefits behind this method cannot be harnessed here as shown in the pseudo-optimality results, where they only happen for sampling periods. Instead, we use the non-optimal RRT-Connect coupled with a subsequent path shortening recursive method. The changes to the basic RRT algorithm can be summed up below.

In this algorithm, two trees are initiated, one from the starting configuration q_{start} and another from the goal configuration q_{goal}. At each step, one of the trees is randomly expanded towards a new sampled

Algorithm 8: Pseudo algorithm for RRT-Connect

```
1  function RRTGrow ($q_{start}, q_{goal}$);
   Input   : start configuration and goal configuration
   Output: tree structure $G = (V, E)$ containing list of of vertex and edges
2  $V_a \leftarrow \{q_{start}\}$;
3  $V_b \leftarrow \{q_{goal}\}$;
4  $E_a \leftarrow \phi$;
5  $E_b \leftarrow \phi$;
6  for iteration $= 1, 2, ...N$ do
7      $q_{rand} \leftarrow$ SampleFree($i$);
8      if !extend($V_a, q_{rand}$) $= Trapped$ then
9          if !connect($V_b, q_{new}$) $= Reached$ then
10             return PATH($V_a, V_b$)
11         end
12     end
13     swap($V_a, V_b$);
14 end
15 return $G = (V, E)$
```

configuration (line 7). The function that takes care of this corresponds to lines 6 trough 11 from algorithm 4. When that is successful, the other tree attempts to connect to the new configuration q_{new} resulting from the previous expansion. The connect function in line 8 attempts to join both trees when in line of sight of each other. Until that is reached, the trees alternate with each others behaviour so they can both randomly explore the search space. This cycle ends when the connection between them is made and a continuous path is formed. However, this path may contain unnecessary vertexes which are removed to make it more direct. This final pruned path is then returned. After obtaining the path, a recursive method to shorten its length is applied. The method can be summed up by the following pseudo algorithm.

Algorithm 9: Pseudo algorithm for RRT-Connect

```
1  function ShortcutPath ($q_{start}, q_{goal}$);
   Input   : Path
   Output: Shortened Path
2  $path_{prev} \leftarrow G(V, E)$;
3  $path_{now} \leftarrow \phi$;
4  while $!timeoutReached$ do
5      $path_{prev} \leftarrow path_{now}$;
6      $path_{now} \leftarrow$ shortcutPath($path_{prev}, maxSteps, maxEmptySteps, rangeRatio, snapToVertex$);
7      if length($path_{now}$) $>$ length($path_{prev}$) $\cdot \gamma$ then
8          $rangeRatio \leftarrow$ switch($rangeRatio$);
9      end
10 end
11 return $path_{now}$
```

The shortening algorithm works by iteratively trying to reduce the size of the current path (while maintaining its validity) by attempting connections between randomly sampled points along path segments and not only the available vertexes from the original path. This connection is done through the use of the function **shortcutPath**, available in the **OMPL** framework. Besides the path to shorten, this function has a set of parameters such as $maxSteps$ and $maxEmptySteps$ which together control the number of attempts to shortcut the path (these are kept at default value). The $rangeRatio$ influences the distance between attempted connections and $snapToVertex$ which determines how close the sampled points are to the vertexes in a segment. Both these parameters are randomly chosen to prevent bias. The iterative process consists of multiple applications of this function to the path while randomizing the $rangeRatio$ value whenever the previous shortening produces a path whose length only represents a small improvement over the previous path (controlled by γ which is set to 1.1). The cycle ends when a timeout is reached. The process can be visualized below.

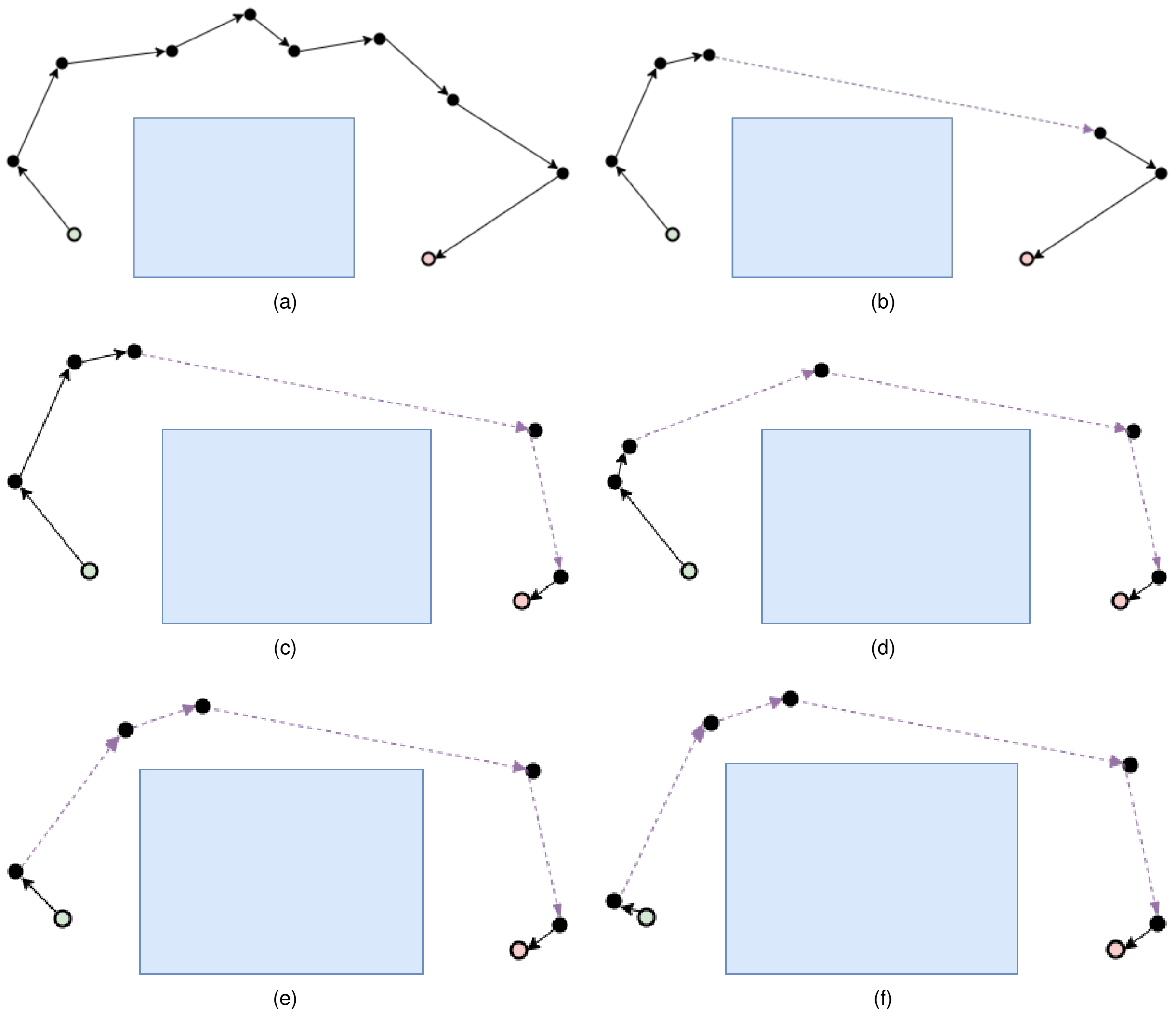


Figure 3.28: Path shortening visualization (a) through (f).

Using the RRT-Connect coupled with the path shortening method in the same environments presented for the Informed RRT* we obtained the following results.

Timeout	0.05s	0.1s	0.5s	1s	5s
RRT-Connect	33%	40%	33%	35%	41%
RRT-Connect + Path Shortening	74%	76%	77%	78%	78%
Informed RRT*	- %	74%	94%	99%	99%

Table 3.5: Pseudo-Optimality for each method with timeout variation (maze environment)

Timeout	0.05s	0.1s	0.5s	1s	5s
RRT-Connect	29%	36%	33%	34%	31%
RRT-Connect + Path Shortening	75%	76%	76%	77%	77%
Informed RRT*	-%	35%	61%	99%	99%

Table 3.6: Pseudo-Optimality for each method with timeout variation (Constrained Obstacle Course)

Timeout	0.05s	0.1s	0.5s	1s	5s
RRT-Connect	40%	38%	46%	37%	39%
RRT-Connect + Path Shortening	78%	78%	79%	80%	80%
Informed RRT*	-%	36%	59%	95%	99%

Table 3.7: Pseudo-Optimality for each method with timeout variation (Relaxed Obstacle Course)

In all scenarios, the minimum timeout set allows for a reasonable pseudo-optimality for the joint method compared with the just the RRT-Connect (Informed RRT* is not able to reach a solution in that time). Increasing the timeout does not wield significant improvements since the path shortening as presently constructed has theoretical limitations on improving the path. Nonetheless by applying this, it shows a good tradeoff between computation time and path length.

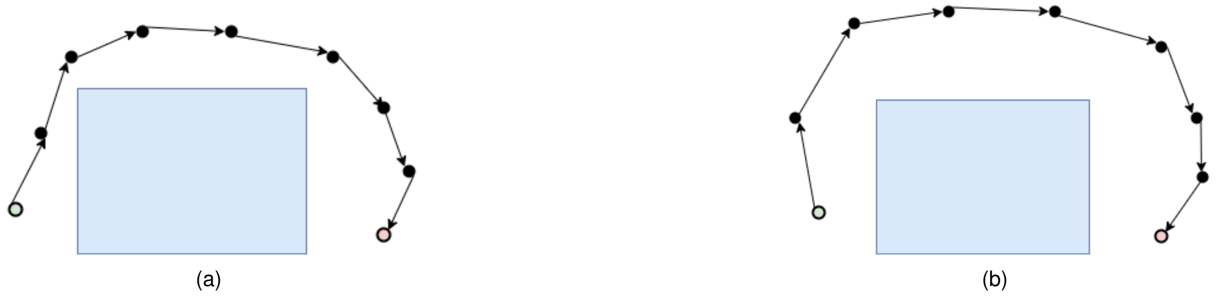


Figure 3.29: (a) path using normal sampling. (b) path using maximum clearance sampling.

There is one additional tool that was employed during path generation to strengthen dynamic obstacle avoidance. Ideally, we want to place the vertexes as far away from the dynamic obstacles as possible for safety purposes. Besides, the safety margins already mentioned, OMPL offers this as a functionality called **MaximizeMinimumClearance** in which the sampling of states is done with a bias towards states that offer higher clearance towards obstacles. This is employed in both sampling and path shortening.

Similarly to the offline trajectory generation, this path can now be transformed into a collision-free trajectory by using the optimization described before.

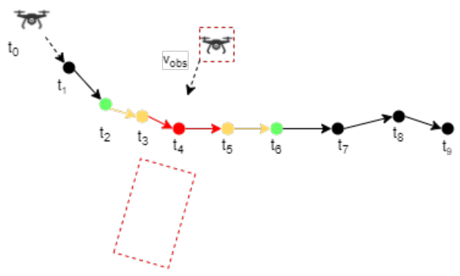


Figure 3.30: Visual representation of potential collision situation between an autonomous agent and an intruder (bounded by a red box) both shown as UAVs. States (in red) in the agent's trajectory are considered to be in collision course with the intruder, while states near the collision are signaled in orange. Green states represent exit points and possible re-entrance points. The red rectangular dotted area represents the region the autonomous agent can not cross and as stated is updated into the octomap.

Starting from a similar situation as outlined in figure 3.28, there are two distinct avoidance strategies that can be employed as shown below.

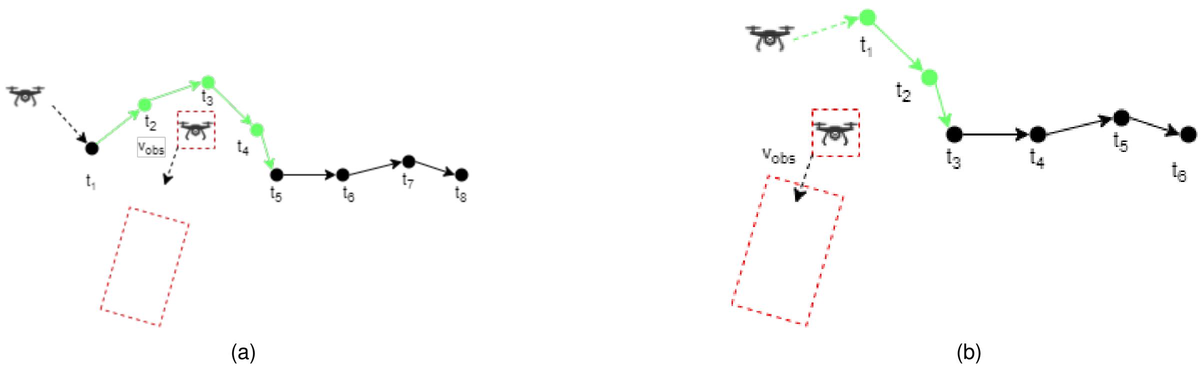


Figure 3.31: Progression of avoidance maneuver . The transitioning trajectory in green and the feasible portion of the previous trajectory in black.

The first one consists of reusing part of the original trajectory not affected by the collision. To do this we first define an exit point and a re-entrance point in the current trajectory and then generate a collision-free path between these points. These points are chosen within a reasonable range from the affected states in the collision similarly to the division illustrated in figure 3.32. This division is done to leverage the combination of the fast continuous update of the environment with the fast reactive trajectory generation instead of imposing dynamic obstacle constraints in the optimization method.

After the path is found, a trajectory is then generated through the optimization method. In order to have a smooth trajectory, additional constraints need to be specified on the exit and re-entrance points according to the values of the derivatives in the original trajectory. At the exit point at we have to guarantee $[v = v_{exit}, a = a_{exit}, j = j_{exit}, sn = sn_{exit}]$ and at re-entrance $[v = v_{re-entrance}, a = a_{re-entrance}, j = j_{re-entrance}, sn = sn_{re-entrance}]$. Besides this change in the optimization method, adjusting the parameters provided within the implementation such as the number of iterations which is lowered as well as increasing the time penalty are appropriate steps to take in order to prioritize a faster solution. The resulting trajectory consists of the junction between the newly generated trajectory and the remaining feasible portion of the previous trajectory. This is the preferential method that is first tried in any collision situation since we only have to readjust a small portion of the trajectory resulting in a very low computational effort since these transitioning trajectories are usually obtained from small paths.

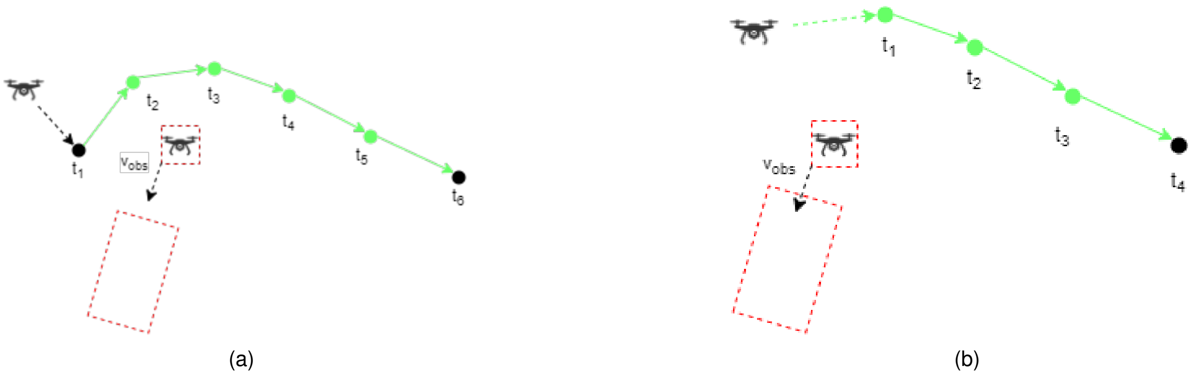


Figure 3.32: (a) path using normal sampling. (b) path using maximum clearance sampling.

In the event that the first method fails to provide a solution by exceeding a defined timeout, another attempt to correct the collision-bound trajectory is made. This time an exit point is chosen the same way as before but the trajectory from this point on is fully discarded and a new trajectory is generated originating at the exit point and ending at the same final state as defined in the original mission objective. As added constraints only the derivatives at the exit point need to be imposed in the newly generated trajectory. The resulting trajectory consists of this plus the remaining of the previous trajectory. In both approaches, there is a guarantee of a collision free trajectory for an immediate time horizon. However, the purely reactive nature of this approach might lead to another instance of collision after the correction of the trajectory. This is where the method has a clear drawback whose mitigation is made possible by the fast re-activeness of the method. Subsequent corrections are applied as long as there is a potential collision in sight. To better illustrate the application of the full framework with the avoidance maneuvers, several simulations were outlined.

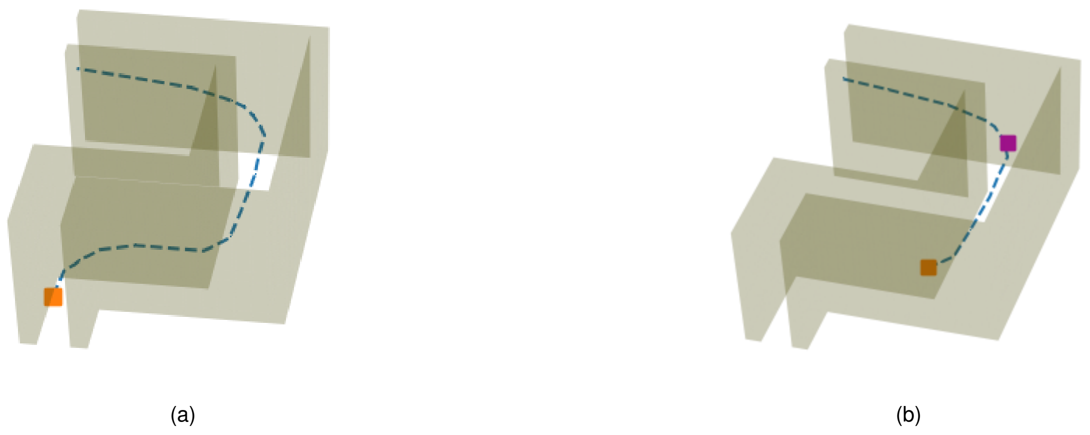
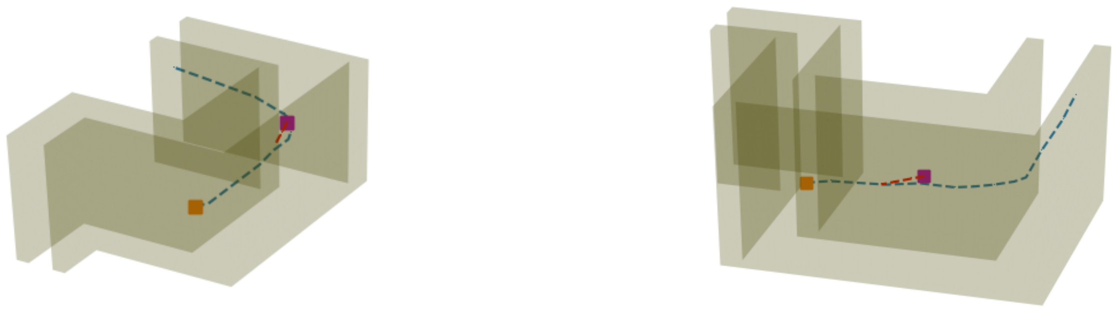


Figure 3.33: (a) Offline trajectory generated. (b) Detection of intruder.

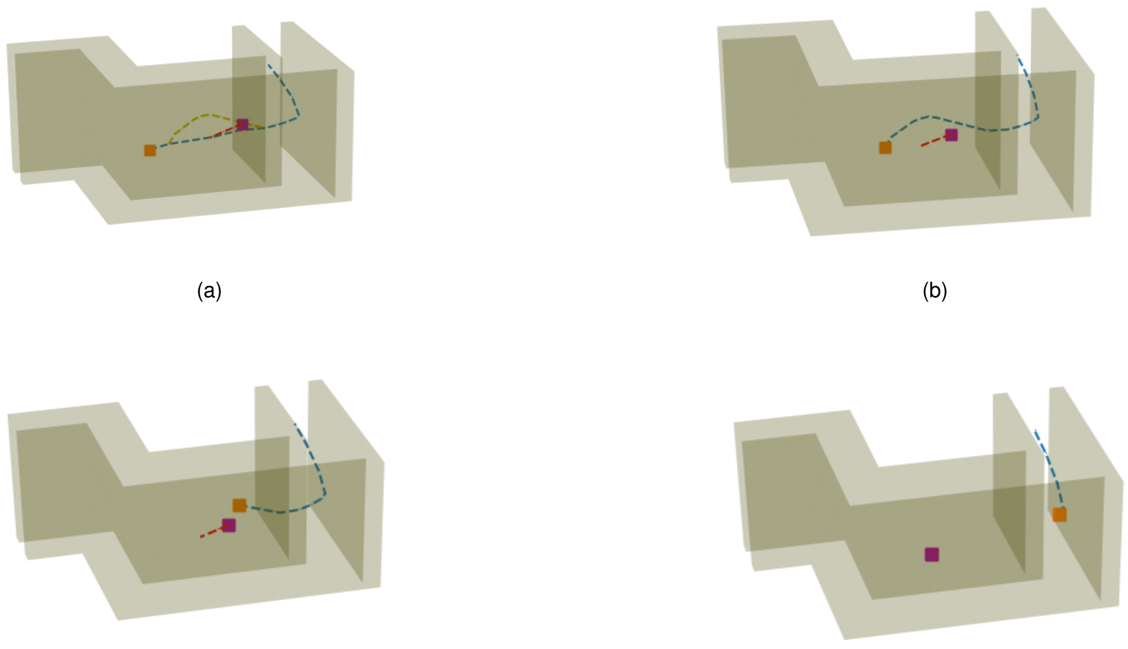
In the first figure (3.35-a), an offline trajectory is generated and the quadrotor starts tracking it. The next image (3.35-b) shows a detected intruder by the quadrotor at the moment it first falls into line of sight.



(a) (b)

Figure 3.34: (a) Estimated future motion of intruder. (b) Potential collision detected.

The quadrotor starts tracking the position and velocity of the intruder and based on that estimates (as explained) the future location of the intruder. At the moment a potential collision is detected avoidance maneuvers are employed.

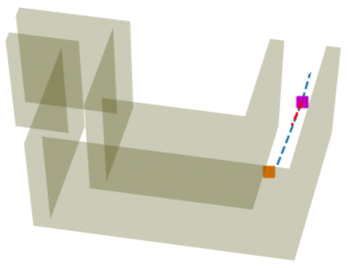


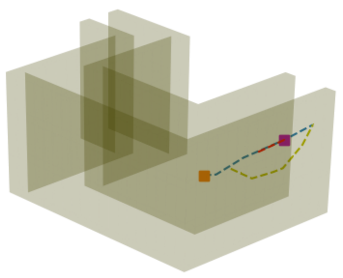
(a) (b)

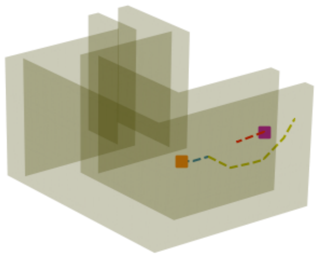
(c) (d)

Figure 3.35: (a) Transitioning trajectory generated. (b)-(c)-(d) Tracking of the new trajectory.

Using the first avoidance maneuver approach a transitioning trajectory is successfully created between a state in the original trajectory prior to the collision and another state after the collision. The resulting trajectory allows the quadrotor to avoid the intruder and it continues its new trajectory towards the destination.


(a)


(b)


(c)

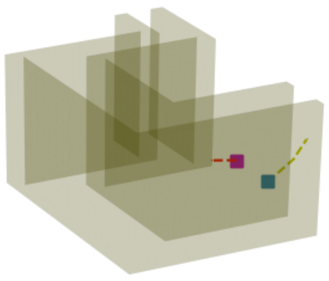
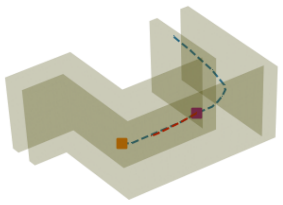

(d)

Figure 3.36: (a) New intruder detected. (b)-(c)-(d) New trajectory generated and tracking.

At another instance in the trajectory a new intruder is detected and similarly to before, a collision is detected. However, now there is no possible states before and after the collision to generate a transitioning trajectory and after failing to apply the first method, the second approach ensues by discarding the previous trajectory which is located after the chosen exit state. From there, a new trajectory towards the destination is generated. The figures shown are only one instance of comparable simulations tested where the relative velocity between the quadrotor and the intruder was kept within a reasonable range. The maximum velocity of the intruder was of $4m/s$ and the avoidance success rate hovered at around 80% meaning it was possible to avoid the dynamic obstacle when applying either of the two maneuvers. When dealing with intruders with higher velocity values than this, the success rate drops significantly being bottle-necked by the duration of the optimization algorithm despite remaining low. An example is shown below.


(a)

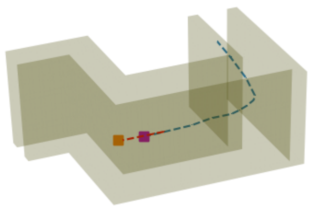

(b)

Figure 3.37: (a) an intruder is approaching at a high perceived velocity resulting in a potential collision. (b) In the time the intruder reaches the collision point, the quadrotor is not able to generate and escaping trajectory.

While the framework is specifically targeted towards application in known environments where dynamic obstacles are the only unknown variables, the idea of exploring unknown environments is also an appealing area. To explore that possibility the framework was adapted to work in unknown environments by applying the same concepts behind dynamic obstacle avoidance. The same sensing capabilities used to detect dynamic obstacles are used to simulate world building as the quadrotor navigates the environment. At each new update in the environment configuration, the trajectory is checked for feasibility and according to the outcome it might need to be repaired in a similar way as before. A simulation illustrating these concepts is presented below.

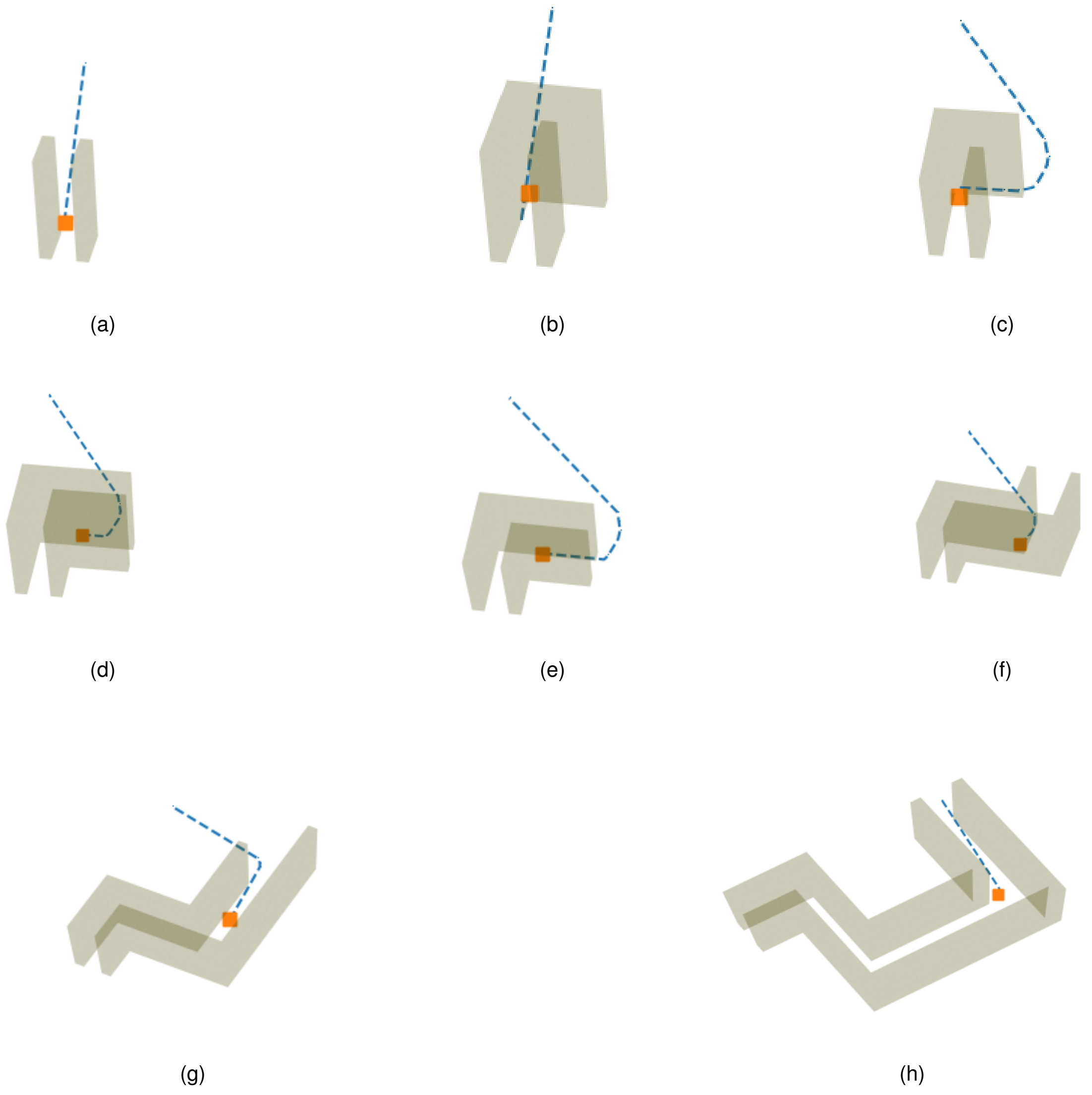


Figure 3.38: (a) through (h) : navigation in unknown environment. At first in (a) a trajectory is generated towards the destination but as the quadrotor moves in it, the sensing systems capture new information on the surroundings thus updating the configuration space. Due to this in (b) the trajectory becomes unfeasible and it is regrown as shown in (c). This process continues until the quadrotor reaches the goal.

Chapter 4

Physics Simulation

The next step in aiding the testing and validation of the developed framework is to deploy it in robust, flexible and accurate computer simulations where we have a high-fidelity environment and quadrotor dynamics modelling that the previous simulations were not able to reproduce since they reduced the quadrotor to a point in the configuration space perfectly capable of following the intended trajectory. This set of simulations is called **Software-In-The-Loop (SITL)** and allows for an accurate depiction of a real-world testing without the need for actual hardware. This is specially advantageous because it completely eliminates hazardous situations that could come up when dealing with the inherent unpredictable nature of autonomous navigation. In this work, a combination of **ROS** and the **PX4 firmware** (which serves as the UAV's autopilot and flight controller) with **Gazebo** acting as the simulation environment. The interaction between these components can be summed up below but it will be detailed further on. The communication between ROS and PX4 is done through the use of the **MAVROS** package. It is also common to deploy a Ground Control Station (GCS) in autonomous missions with pre-determined flight routes which is not the case here so it will be overlooked.

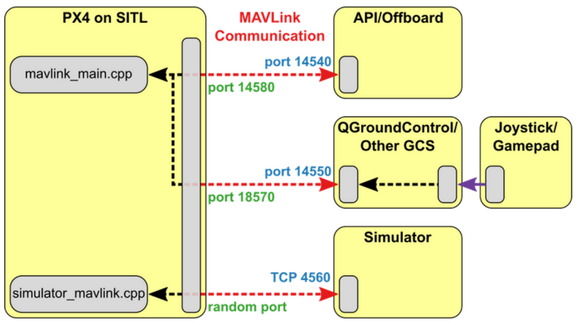


Figure 4.1: Visual representation of SITL simulation components. PX4 communicates with the simulator (e.g. Gazebo) to receive sensor data from the simulated world and send motor and actuator values. It communicates with the Ground Control Station (GCS) and an Offboard API (e.g. ROS) to send telemetry from the simulated environment and receive commands. Adapted from [87].

4.1 Gazebo

Gazebo [88] is high-fidelity open-source physics simulator developed for quadrotors and any other robots at the University of California. It uses the Ogre3D visualisation tool and allows the use of four distinct physics engines though the default is the Open Dynamics Engine(ODE). Compatibility and integration with PX4 and the ability to easily create world models in Gazebo made this the natural choice. The representation of any model in Gazebo is made through a special type of file called SDF (Simulation Description Format). This file is written in XML code formatting. Some basic concepts regarding world modelling in Gazebo are detailed in Appendix A.

4.2 PX4 SITL + ROS

The Robot Operating System (ROS) is an open-source software framework for robotic and drone development and has been featured extensively in many programs since it is development in 2009 [89]. It operates according to a publisher/subscriber messaging service where a global server called **ROSMaster** is established and clients (nodes) can connect to it. The communication between nodes and between these and the master node (ROSMaster) is done through dedicated channels (topics) that each node can publish to if they want to broadcast information or subscribe to in the case of receiving it. The main advantage of ROS as a robotics tool lies in the easy exchange of information such as sensor readings, position from GPS between multiple clients.

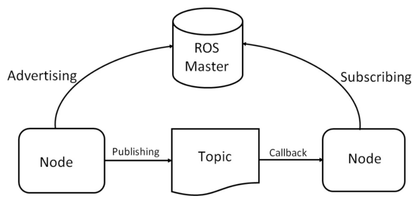


Figure 4.2: Generic ROS model. ROS Master is responsible for the management and registration services of nodes within the model. A node is an executable file in the system. Each node can communicate with other nodes by publishing or subscribing in topics which are effectively communication channels. Taken directly from [89].

PX4 is an open source autopilot (used to control the trajectory of a robot) flight stack focused on embedding communication and control of ground and underwater vehicles and aircraft such as multicopters, fixed wing aicrafts and VTOLs. By doing so, PX4 allows developers to deal only with high level functions such as planning and perception which are illustrated in Appendix B. One of the advantages of PX4 is that it can be deployed in computers and not only in embedding systems (such as a real flight controller board) making it useful for developers. The former is called SITL simulation while the latter is a Hardware In The Loop simulation. It also offers flexible and powerful flight modes and safety features. PX4 SITL communication with Gazebo is done through the **MAVLink** protocol consisting of a lightweight messaging protocol designed specifically for the drone ecosystem. This protocol works under a publish/subscribe and point-to-point design pattern where data is streamed in topics while configuration sub-protocols such as the mission protocol or parameter protocol are point-to-point with re-transmission. To communicate with ROS, an additional ”bridge” is used called **MAVROS** which deals with the middleware message translation between PX4 SITL on running on a MAVLink protocol to ROS.

4.3 Simulation Overview

To further expand on the concepts presented before, a high-level diagram of the overall simulation architecture is presented below.

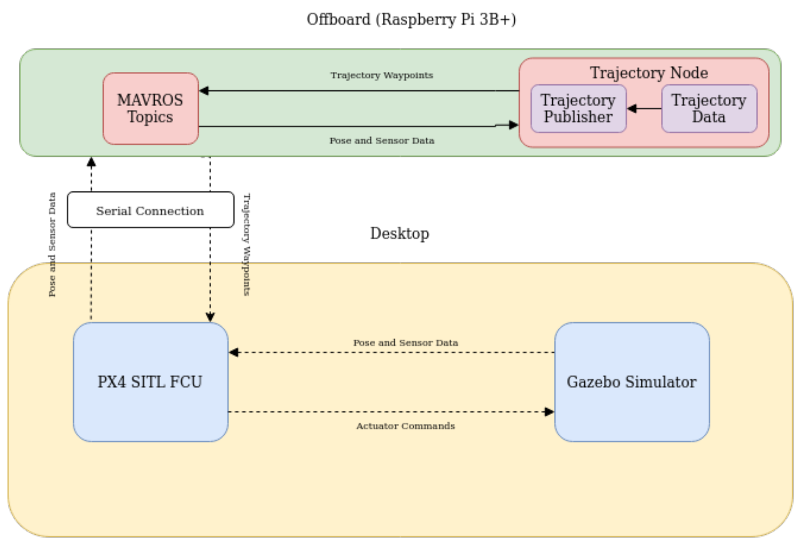


Figure 4.3: Simulation architecture diagram.

Three major components are outlined: Gazebo simulator, PX4 SITL firmware and the MAVROS/Trajectory nodes. One of the main goals of this thesis was to implement autonomous capabilities onboard a UAV so with that in mind the separation between components was made in the previous figure. The **Trajectory Node** which is comprised of the algorithms from the framework developed in chapter 3 as well as some necessary adjustments implemented are running on a Raspberry Pi 3B+ acting as a companion computer. This small single-board computer is easily carried even by small quadrotors so the choice here was natural. On the other hand we have a desktop, where the PX4 code is running in SITL mode (simulating the flight control unit, FCU, of the quadrotor) while communicating directly with Gazebo simulation environment. The companion computer and the desktop are connected via serial (using a USB-to-TTL cable). The Raspberry Pi is declared as the ROSMASTER with the **Trajectory Node** and **MAVROS Node**. The former is comprised of the algorithms developed in chapter 3 and publishes trajectory data to the latter which converts it into the appropriate MAVLink messages and streams it via serial connection to the simulated FCU on the desktop. The FCU receives this, decodes and turns them into the necessary command inputs (through the simulated controllers) which are then streamed via MAVLink to the simulated quadrotor in Gazebo. There is also an exchange of information regarding the quadrotors state back to the **Trajectory Node**. To better understand the functioning of the developed simulation, a standard example will be explained next.

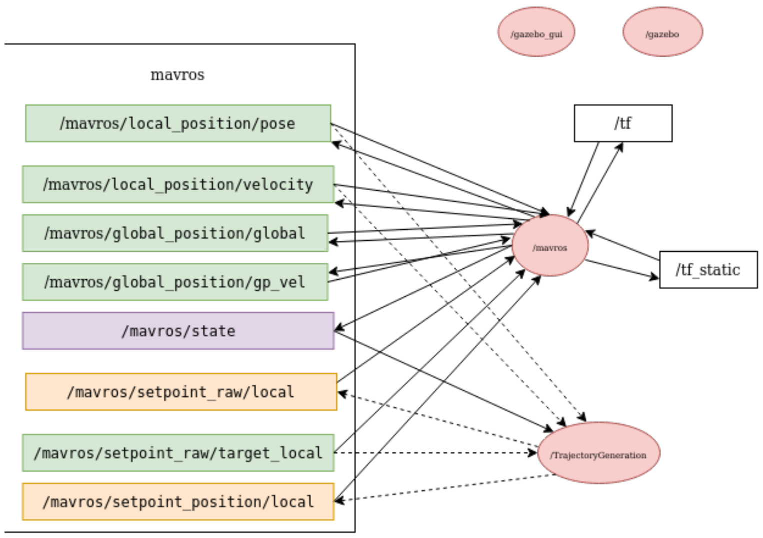

/gazebo_gui
/gazebo
mavros
/mavros/local_position/pose
/mavros/local_position/velocity
/mavros/global_position/global
/mavros/global_position/gp_vel
/mavros/state
/mavros/setpoint_raw/local
/mavros/setpoint_raw/target_local
/mavros/setpoint_position/local
/tf
/mavros
/tf_static
/TrajectoryGeneration

Figure 4.4: ROS graph of the simulation, showing the active nodes and topics.

(Since PX4 code is running in SITL mode and communicating directly with the gazebo physics, running ”behind the scenes”, the corresponding node does not show any connections to the others)

During a simulation, PX4 SITL and Gazebo are both launched on the desktop as well as the ROS nodes on the Raspberry Pi. To visualize the simulated quadrotor as well as the environment the graphical interface of Gazebo is also launched. On the Raspberry Pi side, the Trajectory node creates the necessary callbacks to the relevant topics: $/mavros/state$ holds the current state of the quadrotor; $/mavros/local\_position/pose$ and $/mavros/local\_position/velocity$ store information on the current position and velocity of the quadrotor; $/mavros/setpoint\_raw/local$ allows the quadrotor to publish relevant trajectory information and $/mavros/setpoint\_raw/target\_local$ is a sanity check topic which allows to verify if the information on the previous topic was published as intended. After initiating the callbacks, the node awaits for connection with the desktop by checking the serial connection periodically and once that connection is made it has to verify if the FCU from PX4 SITL is up and running. Before doing this, the publishing rate must be set to a value higher than $2Hz$ (meaning we should publish values at least twice each second). The value chosen and the reasoning behind will be discussed below. By periodically checking on the state of the quadrotor through $/mavros/state$, after a connection it is a good pratice to publish on a topic to check on the connection. Here, we publish on $/mavros/setpoint\_position/local$ which takes 3D coordinates (x, y, z). The quadrotor state is set to **Offboard** mode through $/mavros/state$. This mode is necessary since it allows to control robot movement and attitude by streaming data to the appropriate topics. The quadrotor is then armed through the same topic which activates the motors use. After this, we check on the current quadrotor position given by $/mavros/local\_position/pose$ and according to the value of the start state set in the trajectory generation algorithm, we either drive the quadrotor to this position by publishing it through the previously mentioned topic or if it is already at the position nothing is done. Once we publish the start state $(x_{start}, y_{start}, z_{start})$ on that topic, PX4 SITL controller drives the quadrotor to that position using arbitrary velocity and acceleration. Once the quadrotor reaches that position, the trajectory generation method is launched as before and a trajectory is generated containing several individual states. Before, each state in the trajectory was defined as containing a defined position, velocity, acceleration, jerk and snap. However, these states present discontinuites between the derivatives despite the trajectory generation method used, reducing this problem by employing piece-wise polynomials of higher degrees. If these states were feed directly to the quadrotor, it would lead to discontinuities in acceleration since a real quadrotor would not be able to change direction abruptly. To absorb the discontinuites some sort of feedback or feedforward controller must be used to drive the quadrotor current state to the desired state. PX4 firmware provides a feedforward cascade position and velocity controller which is detailed in Appendix C consisting of a proportional position controller (P) and a proportional-integral-derivative controller (PID). By setting a position and velocity from each state to $/mavros/setpoint\_raw/local$, PX4 uses the internal controller and the fact that quadrotor dynamics are differentially flat to produce the necessary thrust F and attitude Ψ to drive the quadrotor to the desired position at the desired velocity (according to arbitrary yaw angles ψ), thus effectively following the given trajectory. To that effect the position and velocity at each state in the quadrotor trajectory is then published as a setpoint to this topic at a rate equal to the defined sampling interval of the trajectory optimization method which was defined as $10Hz$. However, this value was lowered to $5Hz$ since in practice trajectory tracking performs better. This is due to the position and velocity controller which tends to react more accurately when setpoints are given more sparsely. According to PX4 documentation, there is also a possibility to define setpoints using position, velocity and also acceleration since an acceleration controller is also present. While attempting to feed the setpoints as such, the quadrotor exhibited erratic behaviour which was also corroborated by consulting several open discussions on similar attempts on the PX4 forums. $/mavros/setpoint\_raw/target\_local$ provides loopback information on the setpoints previously sent and it is a safety check to guarantee that the setpoints are being correctly sent. $/mavros/local\_position/pose$ is mainly used to see how well the trajectory tracking is being done throughout the simulation. The two remaining shown topics $/mavros/global\_position\_position/global$ and $/mavros/global\_position\_position/gp\_vel$ are used by PX4 to translate the setpoints from the local frame to the world frame. Also in the controller side, the necessary translations between inertial and body frames are also taken care of. The rest of the trajectory node works under the same assumptions as before, which means that dynamic obstacles are simulated in the same manner and they are not represented physically in Gazebo. In the next section, the results from several simulations are shown and analysed.

4.4 Results

In the simulations, the quadrotor used was the already mentioned 3DR Iris which is one of the standard aircrafts already modelled in PX4. The simulations are divided according to the evolution of the framework as described in chapter 3, testing first without avoidance capabilities and then inserting them. They are spread across different constructed Gazebo maps. Despite, using the less powerful Raspberry Pi to run the trajectory generation algorithm, the decrease in performance was lower than expected and it sat at a mean increase in computational time of around 20% when compared to the execution time on a desktop. These simulations are tailored to show the stability and to evaluate the effectiveness of the developed algorithms. This is done by analysing the deviation error between the commanded position p_{com} that is fed to PX4 controller and the resulting position p_{res} that is exhibited by the quadrotor. This is an example of metrics from trajectory tracking. The position error (which we want to keep as low and fluctuation free as possible) allows us to check on the potential real-world feasibility of both the trajectory generation method and the obstacle avoidance maneuvers followed. Formally it can be written as:

$$Error = |p_{com} - p_{res}| \tag{4.1}$$

Trajectory Generation

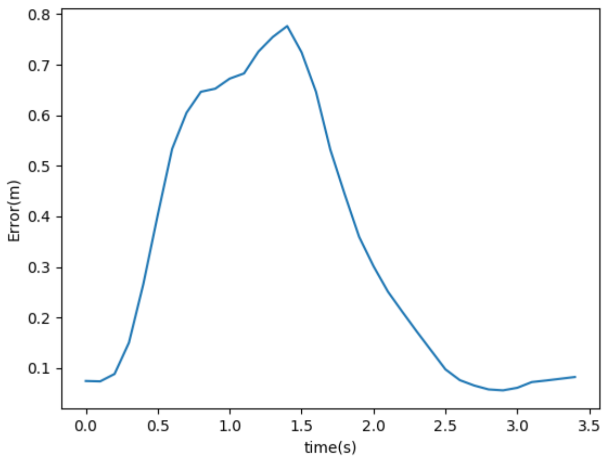


Figure 4.5: Position error evolution with time.

The first test corresponds to a simulation in which a simple maneuver of overcoming a wall (figure A.2 in Appendix A) between two points, one from each side. Both velocity and acceleration were set the highest values ($v = 9m/s$ and $a = 9m/s^2$) in which the trajectory generation algorithm can potentially operate according to the paper where the method was first introduced. After setting both values without first tuning the position and velocity controller from the default values, the quadrotor struggled to keep up with the position and velocity commands which lead to high position error translated in erratic behaviour such as hitting the wall or spinning out of control. After adjusting the controller parameters, the best results were achieved in figure 4.5 where position error reaches a maximum of $0.8m$, an acceptable error specially in spaced outdoor environments. There are only small oscillations and overall the control is smooth. This maximum error is verified when the quadrotor after ascending upwards straight to an altitude above the wall, it suddenly accelerates and this steep change leads to an overshoot in position. The tuned values were kept for the following simulations. The noisy behaviour in the position error at the end of each simulation is attributed to the fact that once the destination is reached, the quadrotor is instructed to hover around that point.

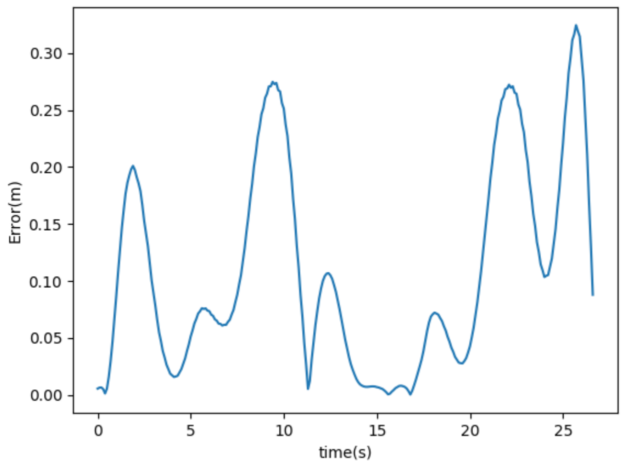


Figure 4.6: Position error evolution with time.

The next set of simulations were performed in the environment pictured in figure A.3 in Appendix A and they correspond to the navigation between the beginning position before and the goal position after the maze of obstacles. The tuned values correspond to an ideal situation as presented before and while the presence of more obstacles in this situation does not invalidate them, it might increase the tracking error if we maintain such a high acceleration. To prevent having to further adjust the values, an alternative is found by lowering the maximum acceleration to $6m/s^2$. The main benefit behind this is to prevent overshooting from happening when going around narrow paths between obstacles with a high acceleration. By doing this, the maximum position error (in figure 4.6) is kept under $0.3m$ which is evidence of a good tracking. The overshooting verified tends to happen primarily due to changes in direction when turning around obstacles in the first environment as explained before. Nonetheless this is expected behaviour which could be further mitigated by lowering the maximum acceleration but this is a cost that can put in risk some UAV applications requiring minimum trajectory times. Tuning the trajectory optimization parameters such as the time penalty to reduce abrupt changes in velocity could also be explored.

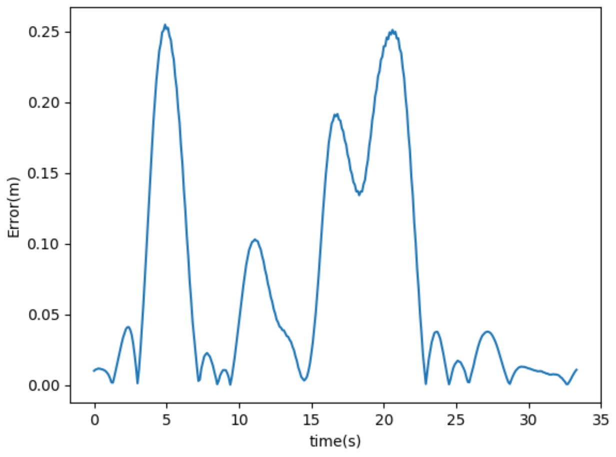


Figure 4.7: Position error evolution with time.

Under the same acceleration constraints as the previous simulations, another set of simulations were performed in the environment illustrated in figure A.4 in the Appendix A. The results obtained in figure 4.7 lead to analogous conclusions as before: overshooting is small but always present as expected and the tracking error is kept around acceptable values. We can also see that the overall error is smaller than in the previous simulations which is due to the more free space to navigate in the environment.

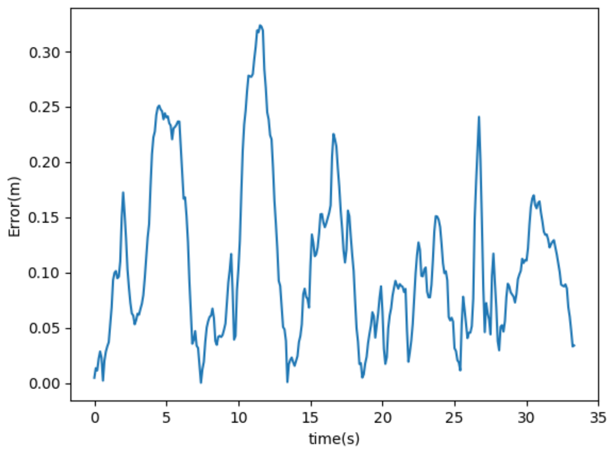


Figure 4.8: Position error evolution with time.

The last set of simulations were made on a highly space constrained environment (figure A.5 in Appendix A). The idea was to see if it was possible to navigate safely between the start and goal positions under the maximum acceleration set before. Despite several attempts at tuning the controller parameters to allow for an acceptable trajectory tracking, due to the constrained space and the inherent tracking phenomenons (such as overshooting), most simulations resulted in unacceptable deviations from the intended trajectory resulting in collisions with obstacles in the environment. Only by lowering acceleration to $2m/s^2$ it was possible to achieve an acceptable position error. However, this leads to a slowly moving quadrotor, thus crippling the inherent advantages of this type of aircrafts. This is obviously an outlier situation since in most practical applications UAVs have enough airspace to move freely in but it is always a good practice to perform stress testing simulations such as these.

Obstacle Avoidance Response

To test the tracking capabilities while performing avoidance maneuvers similar simulations to before were designed. All of them were performed on the environment shown in figure A.4 in Appendix A under the same velocity and acceleration conditions ($9m/s$ and $6m/s^2$, respectively) described before. The choice of this particular map is due to the fact that despite containing several static obstacles in it, there is still enough space allowed for a quadrotor to perform avoidance maneuvers when compared for example with the maze environment where such operations would be severely hindered.

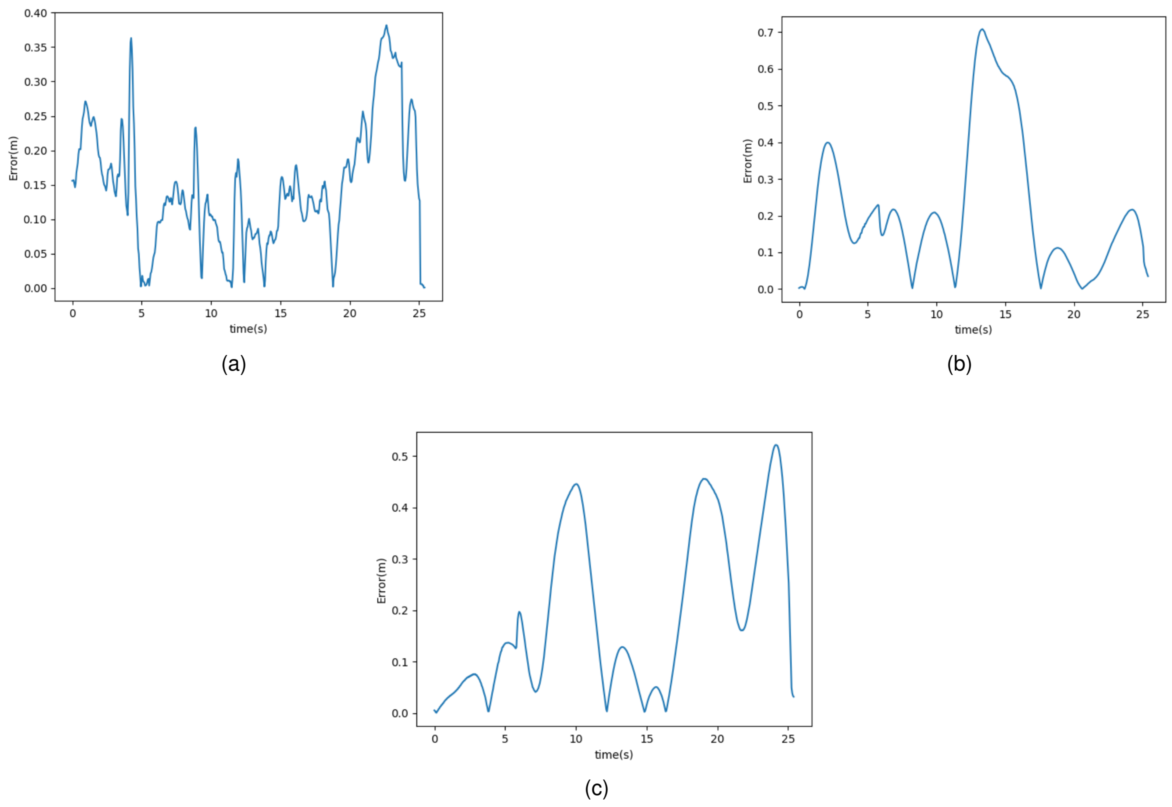


Figure 4.9: Position error evolution with time for three distinct simulations (a), (b) and (c) on the same environment.

The maximum peaks of position error in all three graphs correspond to situations where avoidance maneuvers were taken and they exhibit a similar behaviour as the situations mentioned before when turning corners or around obstacles. However, these peaks tend to be of a higher magnitude since it is common that the transitioning trajectories generated in avoidance maneuvers are more aggressive with more sudden changes in direction and therefore more steep control inputs. Nonetheless in all cases, when limiting the velocity of intruders to reasonable values (lower than $4m/s$), the maximum error was under $0.7m$ which falls within reasonable values for UAV applications. The same logic behind decreasing the maximum acceleration of intruders can also be used to improve the tracking error since less aggressive maneuvers would be needed.

4.5 Remarks

This chapter effectively ended the work done in this thesis with appropriate physics simulations to validate the developed framework. The reasons behind the use of ROS, Gazebo and PX4 SITL in the simulation environment was carefully presented as well as a general overview of the interaction with particular depth given to some of the components (it would not be within the scope of this work to fully explain the PX4 SITL simulation environment). Several simulations were designed to corroborate the application of each of the framework capabilities and the results fell in line with the expected outcome, showing that it is possible to follow the generated smooth trajectories with a high degree of accuracy.

The use of a Raspberry Pi to accommodate the trajectory algorithms was also studied showing that these algorithms are capable of running onboard a small UAV.

Chapter 5

Conclusions

The original goal of this thesis was set as the ”development of a framework that empowers an UAV with the capability of going from a start to a destination while simultaneously avoiding static and dynamic obstacles during its course”. Special emphasis was put on generating flyable smooth trajectories that require as least of a computational effort as possible.

This lead to a two step approach to deal with the problem: an Offline/Pre-Flight path planning and Online/Real Time Path Replanning.

- The first component consists of a path generation method, the Informed RRT*, which generates a pseudo-optimal path which is then fed to an optimization method capable of generating a trajectory. The optimization method focuses on minimizing the total snap (fourth derivative of position) along the trajectory which translates to a high degree of smoothness and low control effort;
- The second component enters into effect when the quadrotor first starts to fly and it is main application is to deal with dynamic obstacles (intruders) that might put the aircraft in risk. Avoidance of these intruders is based on reactive maneuvers where transitioning trajectories are generated between safe points in the original trajectory that allow the quadrotor to circumvent intruders or entire new trajectories from escape points. The trajectory is generated in a similar way as in the first component but now a faster path generation method is used, the RRT-Connect, coupled with a path shortening iterative process to lower path length. The optimization method is tuned to prioritize a faster generation while ensuring smoothness is maintained.

Environment modelling was an important topic and the decision fell on the Octomap representation. To test the capabilities of the developed framework, several non-physics simulations were carried out which showed the effectiveness of the approach under a set of drawn out conditions which serve the main purpose of narrowing the scope of the work done here.

The next step to further validate the capabilities of the framework were physics simulations using a combination of Ros, Gazebo and PX4 SITL. To guarantee the real-time application in small quadrotors, a Raspberry Pi was used to run the trajectory algorithms. Results showed a good performance even considering the computational limitations of the Raspberry Pi and the use of standard position and velocity controllers to track the generated trajectories.

Unfortunately the next logical step of performing HIL simulations was not performed, where instead of simulating quadrotor control through PX4 SITL, an actual flight controller unit is used. This process would greatly benefit the developed work as it would allow the framework to safely be employed on a real quadrotor for further testing. Several difficulties were faced during the development of this thesis derived from personal inexperience in the several theoretical fields involved in the thesis but also poorly devised planning. In addition to this, technical difficulties related to bugs from some of the software used such as the Octomap library contributed to the mentioned shortcomings. The framework was developed specifically to fit a small quadrotor with autonomous capabilities, it would be beneficial to further expand this to larger quadrotors and other multirotor aircraft. While it is a relatively recent field, neural networks could prove to be an interesting way of dealing with autonomous navigation in the future either by fully applying them to the problem or by using some of its benefits to complement path and trajectory generation algorithms. In order to improve tracking performance, a deeper dive in control theory could be done as well as integrating it into the current PX4 code. Some of the capabilities presented require the detection and tracking of intruders, which is something to consider for real time applications since they employ computationally expensive algorithms and further constrain the already low computational capacity of small onboard computers. Integration of these capabilities in the real world testing of the framework could also be explored.

Bibliography

[1] Airbus. Rethinking urban air mobility. *https://www.airbus.com/newsroom/stories/rethinking-urban-air-mobility.html*, June 2017.

[2] C. Weller. Drones could replace $127 billion worth of human labor. *https://www.businessinsider.com/drones-could-replace-127-billion-of-human-labor-2016-5*, May 2018.

[3] D. Vellante and D. Floyer. A new era of innovation: Moore's law is not dead and ai is ready to explode. *https://siliconangle.com/2021/04/10/new-era-innovation-moores-law-not-dead-ai-ready-explode*, Apr. 2021.

[4] A. Palmer. Amazon wins faa approval for prime air drone delivery fleet. *www.https://www.cnbc.com/2020/08/31/amazon-prime-now-drone-delivery-fleet-gets-faa-approval.html*, Aug. 2020.

[5] G. L. Dyndal, T. A. Berntsen, and S. Redse-Johansen. Autonomous military drones: no longer science fiction. *https://www.nato.int/docu/review/articles/2017/07/28/autonomous-military-drones-no-longer-science-fiction/index.html*, July 2017.

[6] C. Yinka-Banjo and O. Ajayi. Sky-farmers: Applications of unmanned aerial vehicles (uav) in agriculture. In G. Dekoulis, editor, *Autonomous Vehicles*, chapter 6. IntechOpen, Rijeka, 2020. doi::10.5772/intechopen.89488.

[7] World cities report 2016: Urbanization and development - emerging futures. Technical report, UN-Habitat, 2016.

[8] A. Drones. Autonomous drones regulation developments. *https://www.azurdrones.com/autonomous-drone-regulation/*, Dec. 2019.

[9] L. L. Gomes, L. Leal, T. R. Oliveira, J. P. V. S. Cunha, and T. C. Revoredo. Unmanned quadcopter control using a motion capture system. *IEEE Latin America Transactions*, 14(8):3606–3613, Dec. 2016. doi::10.1109/TLA.2016.7786340.

[10] B. Zhao, X. Chen, X. Zhao, J. Jiang, and J. Wei . Real-time uav autonomous localization based on smartphone sensors. *Sensors*, 18:4161, Nov. 2018. doi::10.3390/s18124161.

[11] B. Kakillioglu, A. Ren, Y. Wang, and S. Velipasalar. 3d capsule networks for object classification with weight pruning. *IEEE Access*, PP:1–1, Feb. 2020. doi::10.1109/ACCESS.2020.2971950.

[12] *OctoMap: A Probabilistic, Flexible, and Compact 3D Map Representation for Robotic Systems*, volume 2, Jan. 2010.

[13] H. Mei, Y. Tian, and L. Zu. A hybrid ant colony optimization algorithm for path planning of robot in dynamic environment. *Int J Inform Technol*, 12, Jan. 2006.

[14] *Genetic Algorithm Applied in UAV's Path Planning*, 2020. doi::10.1109/CEC48606.2020.9185909.

[15] P. Yao, W. Honglun, and Z. Su. Real-time path planning of unmanned aerial vehicle for target tracking and obstacle avoidance in complex dynamic environment. *Aerospace Science and Technology*, 47, Oct. 2015. doi::10.1016/j.ast.2015.09.037.

[16] R. Allen and M. Pavone. A real-time framework for kinodynamic planning in dynamic environments with application to quadrotor obstacle avoidance. *Robotics and Autonomous Systems*, 115, Dec. 2018. doi::10.1016/j.robot.2018.11.017.

[17] Payal, Akashdeep, and C. Raman Singh. *A Summarization of Collision Avoidance Techniques for Autonomous Navigation of UAV*, page 393. Springer International Publishing, 2020.

[18] X. Zhao, P. Sun, Z. Xu, H. Min, and H. Yu. Fusion of 3d lidar and camera data for object detection in autonomous vehicle applications. *IEEE Sensors Journal*, PP:1–1, Jan. 2020. doi::10.1109/JSEN.2020.2966034.

[19] D. Balemans, S. Vanneste, J. de Hoog, S. Mercelis, and P. Hellinckx. *LiDAR and Camera Sensor Fusion for 2D and 3D Object Detection*, pages 798–807. Springer International Publishing, Jan. 2020.

[20] D. De Silva, J. Roche, and A. Kondoz. Robust fusion of lidar and wide-angle camera data for autonomous mobile robots. *Sensors*, 18:2730, Aug. 2018. doi::10.3390/s18082730.

[21] A. Thibbotuwawa, G. Bocewicz, P. Nielsen, and B. Zbigniew. Planning deliveries with uav routing under weather forecast and energy consumption constraints. *IFAC-PapersOnLine*, 52(13):2730, Aug. 2019. doi::0.1016/j.ifacol.2019.11.231.

[22] G. Thanellas, V. Moulianitis, and N. Aspragathos. A spatially wind aware quadcopter (uav) path planning approach. *IFAC-PapersOnLine*, 52:283–288, Jan. 2019. doi::10.1016/j.ifacol.2019.08.084.

[23] M. Selecky, P. Váňa, M. Rollo, and T. Meiser. Wind corrections in flight path planning. *International Journal of Advanced Robotic Systems*, 10:1, May 2013. doi::10.5772/56455.

[24] K. Bollino and L. Lewis. Collision-free multi-uav optimal path planning and cooperative control for tactical applications. Aug. 2008. doi::10.2514/6.2008-7134.

[25] *Distributed Path Planning for Controlling a Fleet of UAVs : Application to a Team of Quadrotors*, July 2017. doi::10.1016/j.ifacol.2017.08.1908.

[26] F. Causa, G. Fasano, and M. Grassi. Multi-uav path planning for autonomous missions in mixed gnss coverage scenarios. *Sensors*, 18:4188, Nov. 2018. doi::10.3390/s18124188.

[27] S. D. Sciences and Technologies. *https://www.dji.com/pt/mavic-mini*.

[28] D. Mellinger and V. Kumar. Minimum snap trajectory generation and control for quadrotors. In *2011 IEEE International Conference on Robotics and Automation*, pages 2520–2525, 2011. doi::10.1109/ICRA.2011.5980409.

[29] F. Remondino. From point cloud to surface: The modeling and visualization problem. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 34, mar 2004. doi::10.3929/ethz-a-004655782.

[30] R. Triebel, P. Pfaff, and W. Burgard. Multi-level surface maps for outdoor terrain mapping and loop closing. pages 2276–2282, oct 2006. doi::10.1109/IROS.2006.282632.

[31] H. Freeman and D. J. Meagher. Octrees: A data structure for solid-object modeling. In H. Freeman and G. G. Pieroni, editors, *Computer Architectures for Spatially Distributed Data*, pages 249–259. Springer Berlin Heidelberg, 1985.

[32] A. Ravankar, A. Ravankar, Y. Kobayashi, and T. Emaru. Autonomous mapping and exploration of uav using low cost sensors. volume 4, page 5753, Nov. 2018. doi::10.3390/ecsa-5-05753.

[33] J.-D. Fossel, D. Hennes, D. Claes, S. Alers, and K. Tuyls. Octoslam: A 3d mapping approach to situational awareness of unmanned aerial vehicles. May 2013. doi::10.1109/ICUAS.2013.6564688.

[34] J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, USA, 1991. ISBN 0792391292.

[35] H. Alturbeh and J. F. Whidborne. Visual flight rules-based collision avoidance systems for uav flying in civil aerospace, 2020. doi::10.3390/robotics9010009.

[36] K. Yang. An efficient spline-based rrt path planner for non-holonomic robots in cluttered environments. In *2013 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 288–297, 2013. doi::10.1109/ICUAS.2013.6564701.

[37] R. Bohlin and L. E. Kavraki. Path planning using lazy prm. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, volume 1, pages 521–528, 2000. doi::10.1109/ROBOT.2000.844107.

[38] R. Bohlin and L. Kavraki. A lazy probabilistic roadmap planner for single query path planning. Sept. 2000.

[39] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. doi::10.1109/TSSC.1968.300136.

[40] M. Likhachev, D. Ferguson, G. Gordon, A. T. Stentz, and S. Thrun. Anytime dynamic a*: An anytime, replanning algorithm. In *Proceedings of 15th International Conference on Automated Planning and Scheduling (ICAPS '05)*, pages 262 – 271, Jun 2005.

[41] S. M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. 1998.

[42] J. J. Kuffner and S. M. LaValle. Rrt-connect: An efficient approach to single-query path planning. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, volume 2, pages 995–1001, 2000. doi::10.1109/ROBOT.2000.844730.

[43] S. Karaman and E. Frazzoli. Sampling-based algorithms for optimal motion planning. *CoRR*, 2011.

[44] S. Karaman and E. Frazzoli. Sampling-based algorithms for optimal motion planning, 2011.

[45] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot. Informed rrt*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic. *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, sep 2014. doi::10.1109/iros.2014.6942976.

[46] *UAV Path Planning System Based on 3D Informed RRT* for Dynamic Obstacle Avoidance*, Dec. 2018. doi::10.1109/ROBIO.2018.8665162.

[47] J. Gammell, S. Srinivasa, and T. Barfoot. Informed rrt*: Optimal incremental path planning focused through an admissible ellipsoidal heuristic. *IEEE International Conference on Intelligent Robots and Systems*, Apr. 2014. doi::10.1109/IROS.2014.6942976.

[48] *RT-RRT*: a real-time path planning algorithm based on RRT**, Nov. 2015. doi::10.1145/2822013.2822036.

[49] *Distributed Optimization by Ant Colonies*, Jan. 1991.

[50] B. Benhala, A. Ali, A. Mechaqrane, B. Benlahbib, F. Abdi, E. Abarkan, and M. Fakhfakh. Sizing of current conveyors by means of an ant colony optimization technique. pages 1 – 6, May 2011. doi::10.1109/ICMCS.2011.5945669.

[51] S. Carabaza, E. Besada, J. Lopez-Orozco, and J. de la Cruz. Ant colony optimization for multi-uav minimum time search in uncertain domains. *Applied Soft Computing*, 62, Sept. 2017. doi::10.1016/j.asoc.2017.09.009.

[52] U. Cekmez, M. Ozsiginan, and O. Sahingoz. Multi colony ant optimization for uav path planning with obstacle avoidance. pages 47–52, June 2016. doi::10.1109/ICUAS.2016.7502621.

[53] C. Jin, S. Jang, X. Sun, J. Li, and R. Christenson. Damage detection of a highway bridge under severe temperature changes using extended kalman filter trained neural network. *Journal of Civil Structural Health Monitoring*, 6:545–560, July 2016. doi::10.1007/s13349-016-0173-8.

[54] W. Luo, Q. Tang, C. Fu, and P. Eberhard. *Deep-Sarsa Based Multi-UAV Path Planning and Obstacle Avoidance in a Dynamic Environment*, pages 102–111. 06 2018. ISBN 978-3-319-93817-2. doi::10.1007/978-3-319-93818-9$_10$.

[55] R. Padhy, S. Verma, S. Ahmad, S. Choudhury, and P. Sa. Deep neural network for autonomous uav navigation in indoor corridor environments. *Procedia Computer Science*, 133:643–650, Jan. 2018. doi::10.1016/j.procs.2018.07.099.

[56] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. In *Proceedings. 1985 IEEE International Conference on Robotics and Automation*, volume 2, pages 500–505, 1985. doi::10.1109/ROBOT.1985.1087247.

[57] G. Fedele, L. D'Alfonso, F. Chiaravalloti, and G. D'Aquila. Obstacles avoidance based on switching potential functions. *Journal of Intelligent Robotic Systems*, 90, June 2018. doi::10.1007/s10846-017-0687-2.

[58] A. Budiyanto, A. Cahyadi, T. Adji, and O. Wahyunggoro. Uav obstacle avoidance using potential field under dynamic environment. pages 187–192, Aug. 2015. doi::10.1109/ICCEREC.2015.7337041.

[59] F. Haro and M. Torres-Torriti. A comparison of path planning algorithms for omnidirectional robots in dynamic environments. Oct. 2006. doi::10.1109/LARS.2006.334319.

[60] Y. Guo, X. Liu, W. Zhang, L. Xuhang, and Y. Yue. Obstacle avoidance planning for quadrotor uav based on improved adaptive artificial potential field. pages 2598–2603, Nov. 2019. doi::10.1109/CAC48633.2019.8996746.

[61] M. Li, h. Bai, and N. Krishnamurthi. A markov decision process for the interaction between autonomous collision avoidance and delayed pilot commands. *IFAC-PapersOnLine*, 51:378–383, Jan. 2019. doi::10.1016/j.ifacol.2019.01.012.

[62] X. Han, J. Wang, J. Xue, and Q. Zhang. Intelligent decision-making for 3-dimensional dynamic obstacle avoidance of uav based on deep reinforcement learning. In *2019 11th International Conference on Wireless Communications and Signal Processing (WCSP)*, pages 1–6, 2019. doi::10.1109/WCSP.2019.8928110.

[63] M. Arnold, R. Negenborn, G. Andersson, and B. De Schutter. Multi-area predictive control for combined electricity and natural gas systems. *2009 European Control Conference, ECC 2009*, Mar. 2015.

[64] M. Hwangbo and T. Kanade. Maneuver-based autonomous navigation of a small fixed-wing uav. In *2013 IEEE International Conference on Robotics and Automation*, pages 3961–3968, 2013. doi::10.1109/ICRA.2013.6631135.

[65] H.-K. Ahn, O. Cheong, J. Matousek, and A. Vigneron. Reachability by paths of bounded curvature in convex polygons. pages 251–259, May 2000. doi::10.1145/336154.336211.

[66] Y. Lin and S. Saripalli. Path planning using 3d dubins curve for unmanned aerial vehicles. pages 296–304, May 2014. ISBN 978-1-4799-2376-2. doi::10.1109/ICUAS.2014.6842268.

[67] D. Wilkie, J. van den Berg, and D. Manocha. Generalized velocity obstacles. pages 5573–5578, Dec. 2009. doi::10.1109/IROS.2009.5354175.

[68] Y. Jenie, E.-J. Van Kampen, C. De Visser, and Q. Chu. Velocity obstacle method for non-cooperative autonomous collision avoidance system for uavs. Jan. 2014. doi::10.2514/6.2014-1472.

[69] M. Choi, A. Rubenecia, T. Shon, and H. H. Choi. Velocity obstacle based 3d collision avoidance scheme for low-cost micro uavs. *Sustainability*, 9:1174, July 2017. doi::10.3390/su9071174.

[70] Y. Jenie. Three-dimensional velocity obstacle method for uav deconflicting maneuvers. Jan. 2015. doi::10.2514/6.2015-0592.

[71] B. Lindqvist, S. S. Mansouri, A. Agha-mohammadi, and G. Nikolakopoulos. Nonlinear mpc for collision avoidance and control of uavs with dynamic obstacles. *IEEE Robotics and Automation Letters*, 5(4): 6001–6008, 2020. doi::10.1109/LRA.2020.3010730.

[72] M. Castillo-Lopez, S. A. Sajadi-Alamdari, J. L. Sanchez-Lopez, M. A. Olivares-Mendez, and H. Voos. Model predictive control for aerial collision avoidance in dynamic environments. In *2018 26th Mediterranean Conference on Control and Automation (MED)*, pages 1–6, 2018. doi::10.1109/MED.2018.8442967.

[73] M. Radmanesh, M. Kumar, A. Nemati, and M. Sarim. Dynamic optimal uav trajectory planning in the national airspace system via mixed integer linear programming. *Proceedings of the Institution of Mechanical Engineers Part G Journal of Aerospace Engineering*, 230, Oct. 2015. doi::10.1177/0954410015609361.

[74] Zhe Zhang, Jianxun Li, and Jun Wang. Sequential convex programming for nonlinear optimal control problem in uav path planning. In *2017 American Control Conference (ACC)*, pages 1966–1971, 2017. doi::10.23919/ACC.2017.7963240.

[75] Z. Zhang, J. Wang, and J. Li. Lossless convexification of nonconvex minlp on the uav path-planning problem. *Optimal Control Applications and Methods*, 39, Dec. 2017. doi::10.1002/oca.2380.

[76] X. Zhang, A. Liniger, and F. Borrelli. Optimization-based collision avoidance. *IEEE Transactions on Control Systems Technology*, pages 1–12, 2020. doi::10.1109/TCST.2019.2949540.

[77] C. Richter, A. Bry, and N. Roy. *Polynomial Trajectory Planning for Aggressive Quadrotor Flight in Dense Indoor Environments*, pages 649–666. Apr. 2016. ISBN 978-3-319-28870-3. doi::10.1007/978-3-319-28872-7$_37$.

[78] F. Furrer, M. Burri, M. Achtelik, and R. Siegwart. Robot operating system (ros). *Studies Comp.Intelligence Volume Number:625*, The Complete Reference (Volume 1)(978-3-319-26052-5): Chapter 23, 2016. ISBN:978-3-319-26052-5.

[79] J. Pan, S. Chitta, and D. Manocha. Fcl: A general purpose library for collision and proximity queries. Jan. 2012.

[80] M. Moll, J. Lee, and M. Sherman. flexible-collision-library. https://https://github.com/flexible-collision-library, 2014.

[81] I. A. Şucan, M. Moll, and L. E. Kavraki. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, December 2012. doi::10.1109/MRA.2012.2205651.

[82] M. Burri, H. Oleynikova, M. W. Achtelik, and R. Siegwart. Real-time visual-inertial mapping, re-localization and planning onboard mavs in unknown environments. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1872–1878, 2015. doi::10.1109/IROS.2015.7353622.

[83] A. Markus, B. Michael, O. Helen, B. Rik, and P. Marija. mav_trajectory_generation. https://github.com/ethz-asl/mav_trajectory_generation, 2017.

[84] M. Powell. The bobyqa algorithm for bound constrained optimization without derivatives. *Technical Report, Department of Applied Mathematics and Theoretical Physics*, Jan. 2009.

[85] M. Mueller, M. Hehn, and R. D'Andrea. A computationally efficient motion primitive for quadrocopter trajectory generation. *IEEE Transactions on Robotics*, 31:1–17, Oct. 2015. doi::10.1109/TRO.2015.2479878.

[86] P. Nousi, I. Mademlis, I. Karakostas, A. Tefas, and I. Pitas. Embedded uav real-time visual object detection and tracking. 08 2019. doi::10.1109/RCAR47638.2019.9043931.

[87] L. Meier, D. Honegger, and M. Pollefeys. Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6235–6240, 2015. doi::10.1109/ICRA.2015.7140074.

[88] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154 vol.3, 2004. doi::10.1109/IROS.2004.1389727.

[89] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng. Ros: an open-source robot operating system. volume 3, Jan. 2009.

Appendix A

Gazebo Simulation Environment

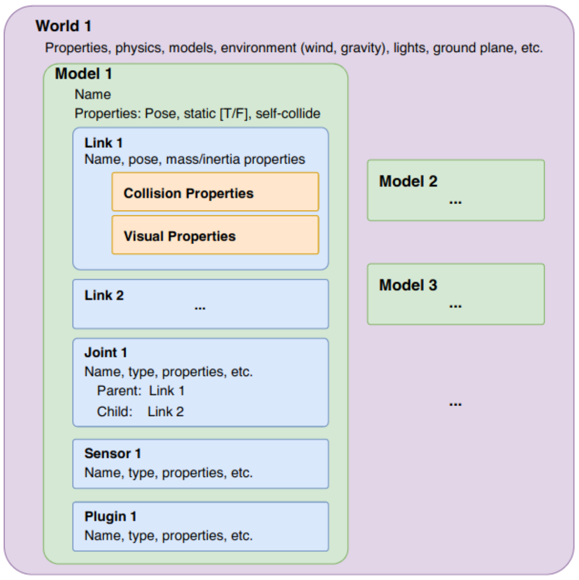

World 1
Properties, physics, models, environment (wind, gravity), lights, ground plane, etc.
Model 1
Name
Properties: Pose, static [T/F], self-collide
Link 1
Name, pose, mass/inertia properties
Collision Properties
Visual Properties
Link 2
...
Joint 1
Name, type, properties, etc.
Parent: Link 1
Child: Link 2
Sensor 1
Name, type, properties, etc.
Plugin 1
Name, type, properties, etc.
Model 2
...
Model 3
...
...

Figure A.1: Gazebo SDF high level hierarchy for creating basic worlds and models. Adapted from [88].

A world file can be defined to represent the generic properties of an environment such as gravity, wind speed and other static/dynamic properties. In each world several Gazebo models can be created. These are defined by .sdf files. These models are internally composed of sub-categories such as links, joints, sensors or plugins. Links are individual elements of the model such as the individual rotors of an UAV that are simulated physically. The .sdf file contains information on the mass, inertia matrix, visual, and collision properties of each link so that they can interact dynamically with the environment. Joints restrict the motion between links (one example could be the hinges that constrain the rotors to the airframe). Plugins can be used to command velocities or accelerations to the models simulated components thus allowing for interaction with the simulation model or world. In this work, the PX4 SITL plugin is used to allow the PX4 flight stack code to run in combination with the simulation. Sensor behaviour models are also simulated. The interaction between all of the above provides realistic dynamics environment for simulating multi-rotor vehicles. The different constructed Gazebo maps used in the simulations are shown below.

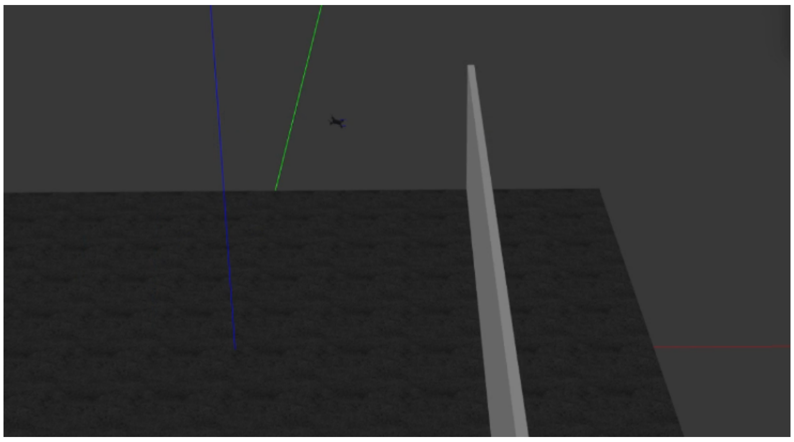


Figure A.2: Wall in Gazebo

This is a simple wall with $4m$ of height. Both length and width are not relevant to the nature of the simulations where this environment is used.

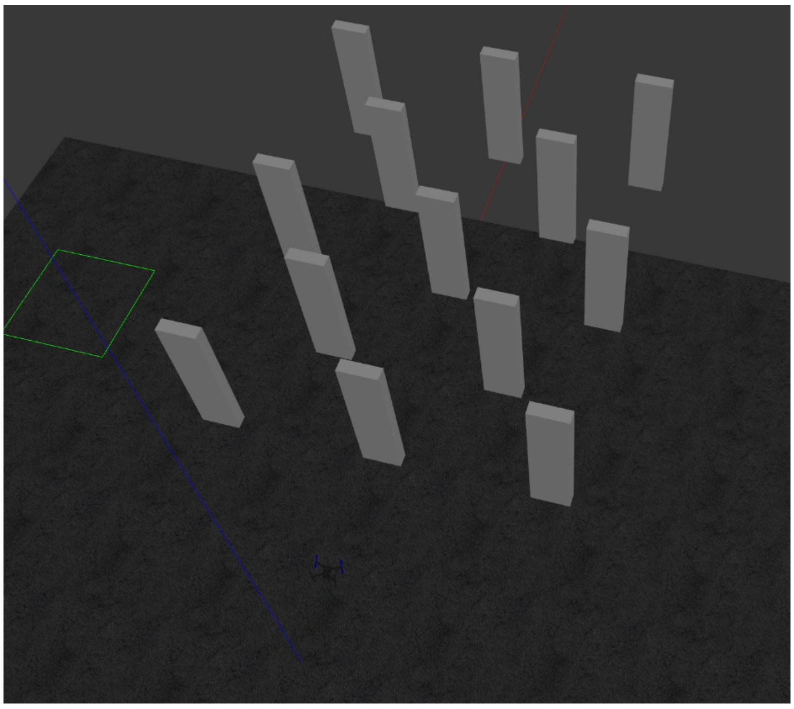


Figure A.3: Maze course in Gazebo

This environment is composed of several smaller parallelepipedal obstacles with equal dimensions of $0.5m - 1m - 3m$ (width, length and height) disposed in a symmetrical pattern on the gazebo grid with at least $2m$ all around spacing between them.

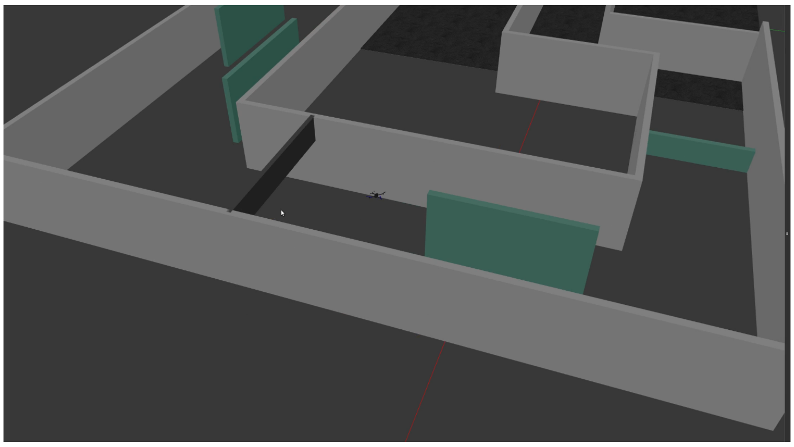


Figure A.4: Obstacle course in Gazebo

This environment is composed of spaced corridors filled with different obstacles at defined points with at least $2m$ of spacing all around.

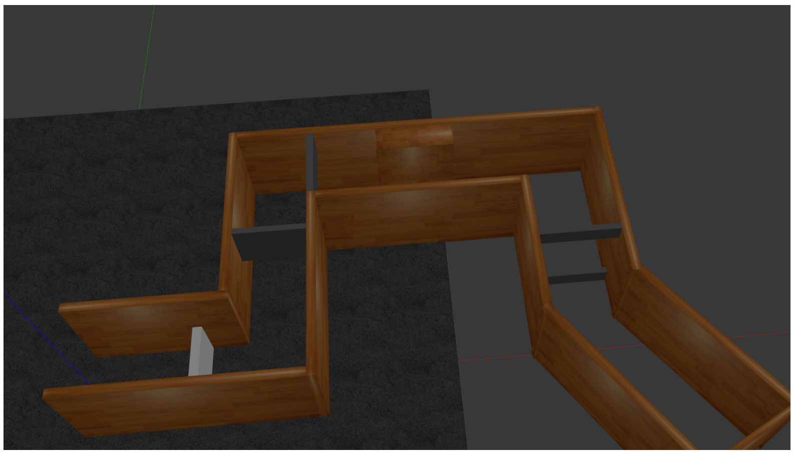


Figure A.5: Constrained obstacle course in Gazebo

This environment is a more spaced constrained version similar to the previous one. Spacing is now reduced to only $1m$.

Appendix B

PX4 Architecture

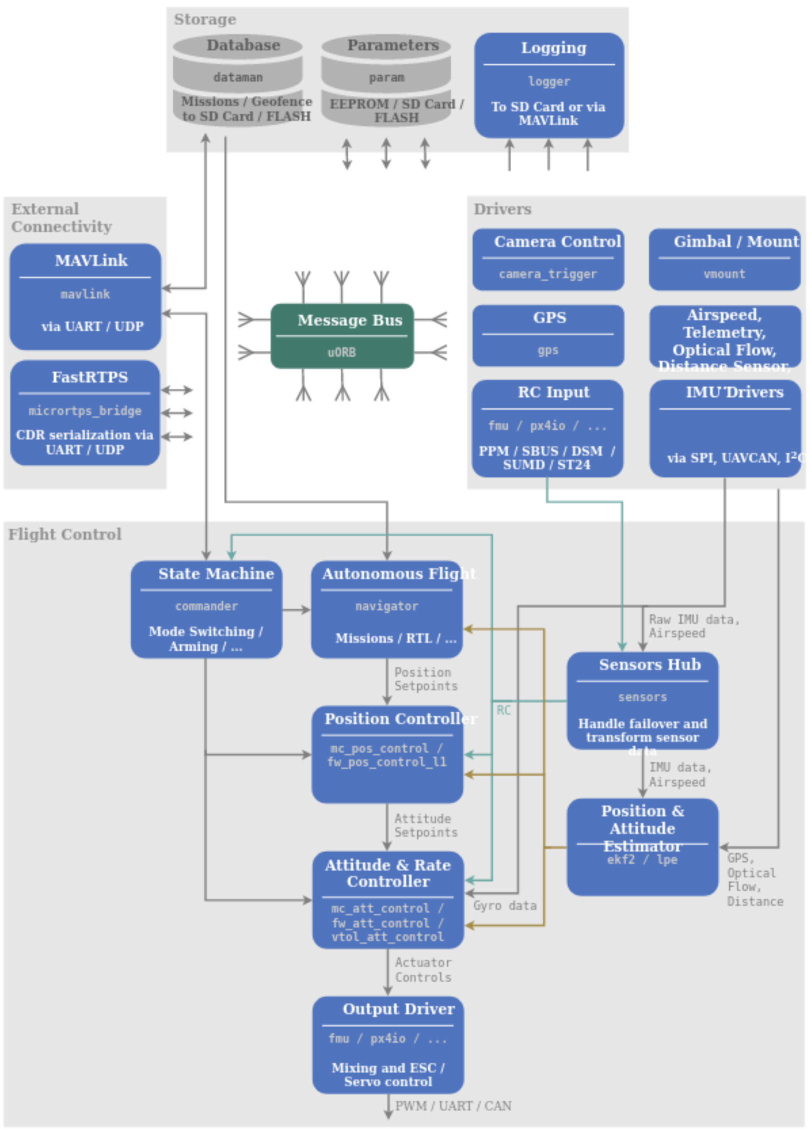


Figure B.1: High-Level Software Architecture of PX4. Taken directly from [87].

PX4 is made up of two layers: the flight stack which consists of the guidance, navigation and control for autonomous drones and the middleware which deals with the communication between systems.

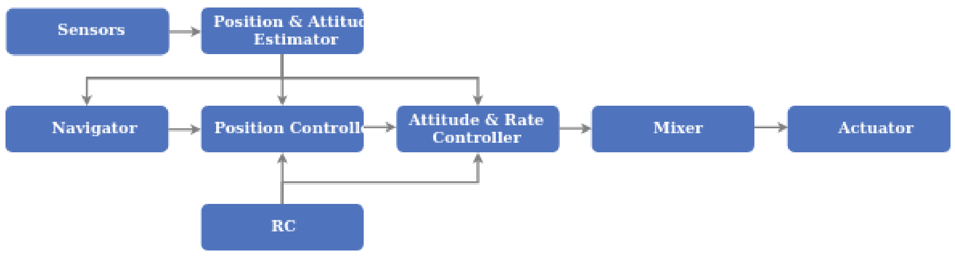
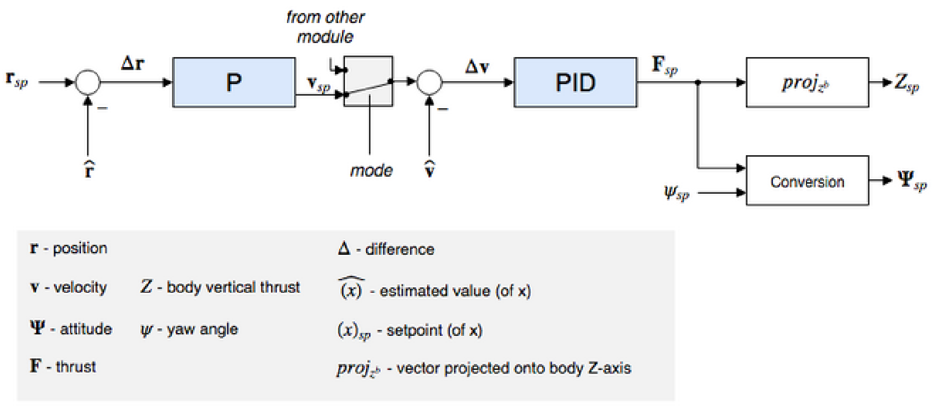


Figure B.2: Flight Stack architecture. Taken directly from [87].

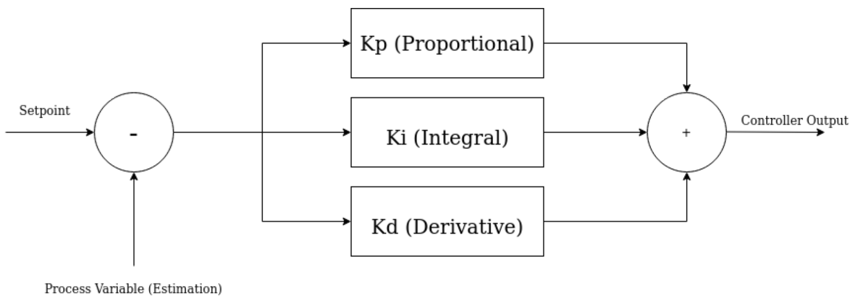
The estimator takes sensor measurements and computes a vehicle state from them such as computing the ground speed from IMU sensor data. The controller takes a setpoint and adjusts the value of the process variable to match it. The position controller is particularly relevant here since a desired position is feed onto the controller which produces the necessary attitude and thrust to attain the position. The mixer takes force inputs and converts them to motor commands. This conversion is specific to vehicle individual dynamics and motor arrangements.

Appendix C

PX4 Controller



(a)



(b)

Figure C.1: (a) Combined Position and Velocity Controller Diagram. The desired velocity v_{sp} is used as feedforward term, it is added to the output of the position controller and the result is used as the input to the velocity controller. Taken directly from[87]. (b) PID controller structure.

The structure of a standard PID controller can also be seen above. The proportional gain P is used to minimize the tracking error and the higher the value, the quicker the response is but at a cost of higher oscillations. The derivative gain D controls overshooting and higher values lead to amplified noise. The integral gain (I) keeps track of the error and increases its value when there is an accumulation of error through time. A good tradeoff between these values is expected.

Appendix D

Taversal Node Hierarchy

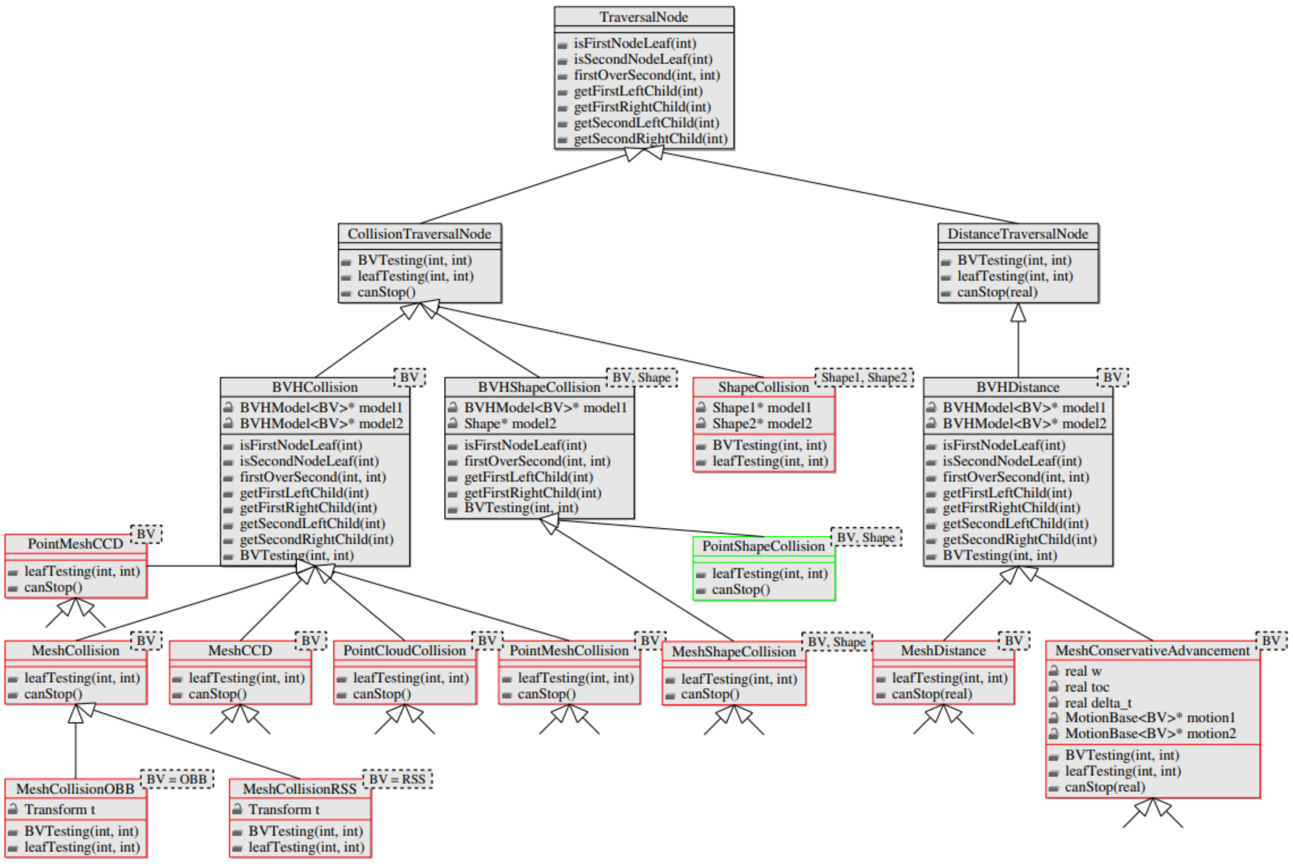


Figure D.1: Hierarchy for traversal node. *CollisionTraversalNode* corresponds to the collision checking routine while *DistanceTraversalNode* performs distance computation between two objects. Each of this subclasses is comprised of the functions necessary to perform these operations according to the requirements of each type of bounding volume involved (for example, *BVHCollision* performs the collision checking between two bounding volumes while *ShapeCollision* performs the same operation for two basic shapes. Taken directly from [79].