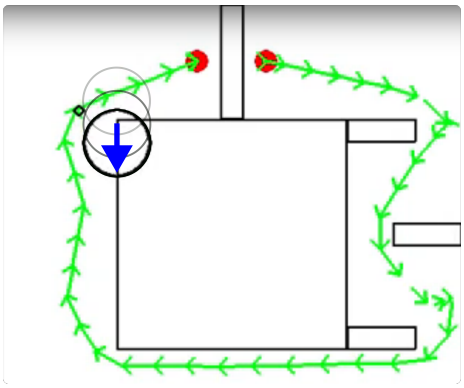
TÉCNICO LISBOA



Real-Time Trajectory Planning for UAVs in Environments with Moving Obstacles

António Luís Gomes Ramalho

Thesis to obtain the Master of Science Degree in

Aerospace Engineering

Supervisors: Prof. Afzal Suleman
Prof. Rodrigo Martins de Matos Ventura

Examination Committee

Chairperson: Prof. José Fernando Alves da Silva
Supervisor: Prof. Rodrigo Martins de Matos Ventura
Member of the Committee: Prof. Paulo Jorge Coelho Ramalho Oliveira

April 2020

”If I have seen further it is by standing on the shoulders of Giants.”
Isaac Newton

”To my mother, my father, my brother and Bea”

Acknowledgments

I would like to thank Professor Afzal Suleman for the opportunity to work in such a nice place in Canada and for the financial support. I would also like to thank him for the possibility to work in projects with direct contact to the industry, particularly working in an ambitious innovation project with Boeing, which was both exciting and enriching.

I would also like to thank Professor Rodrigo Ventura for all the highly productive meetings from which he would always provide me with some key information and advice that enabled this work to progress.

I must, in addition, thank my parents for not only the financial support but also for all the talks and the company provided, by my brother as well, making me feel at home despite the distance.

I would also like to show my gratitude for my colleagues in this work, Pablo and Luis, the ”AI team” that would every day be a company in the, sometimes scary, research world. A special thanks to Luis for making my English readable and for patiently helping me with one of the most annoying software I know: ROS.

Thanks also to my friends, which I missed deeply during my time in Canada, especially during the boring nights.

Finally, I thank Bea, for making this journey a happy one, full of adventures and good times.

Agradecimentos

Gostaria de agradecer ao Professor Afzal Suleman pela oportunidade de trabalhar em Victoria e pelo financiamento. Gostaria também de agradecer pela possibilidade de trabalhar em projectos em contacto directo com a indústria, particularmente pela oportunidade de trabalhar num ambicioso e inovador projecto com a Boeing, foi entusiasmante e enriquecedor.

Gostaria também de agradecer ao Professor Rodrigo Ventura, pelas reuniões altamente produtivas, nas quais o aconselhamento era sempre assertivo e preciosos para o progresso do meu trabalho.

Quero também agradecer aos meus pais, não só pelo financiamento, mas também por todas as chamadas e companhia que providenciaram, juntamente com o meu irmão, que sempre me fizeram sentir em casa apesar da distância.

Expresso também a minha gratidão aos meus colegas neste trabalho: Pablo e Luís, a equipa deste projecto que me ajudou a enfrentar o mundo, por vezes assustador, da investigação. Um agradecimento especial ao Luís por tornar o meu Inglês legível e por me ajudar pacientemente com um dos *softwares* mais frustrantes que eu conheço: o ROS.

Agradeço também aos meus amigos pela companhia e aos amigos que ficaram em Portugal, de quem senti uma enorme falta, especialmente a enfrentar noites aborrecidas.

Finalmente agradeço à Bea, por fazer esta jornada mais feliz e repleta de aventuras e bons momentos.

Resumo

Neste trabalho é apresentado um algoritmo de planeamento de trajectória em tempo real. O algoritmo desenvolvido é capaz de gerar rapidamente trajectórias que evitam obstáculos inesperados, tanto estáticos como em movimento, em tempo real. Este algoritmo tem a capacidade de gerar trajectórias óptimas em termos de custos operacionais, que são dados por uma combinação do tempo de trajectória com uma estimação da energia consumida. O algoritmo foi integrado com um sistema de TCAS simplificado, mostrando a capacidade de um algoritmo de planeamento autónomo utilizar um sistema inicialmente projectado para aeronaves com pilotos humanos. A solução é baseada numa Rapidly-exploring Random Tree (RRT) modificada e num optimizador de trajectória. A RRT modificada permite gerar trajectórias que respeitam um raio mínimo de curvatura. Foi ainda desenvolvido um processo que permite diminuir de forma rápida o comprimento das trajectórias geradas pela RRT. O optimizador de trajectória foi desenhado de forma a utilizar um número reduzido de variáveis de decisão. As trajectórias geradas por este algoritmo são formadas por uma sequência de *splines* de segunda ordem. Múltiplas simulações utilizando uma dinâmica aproximada foram realizadas de forma a avaliar as capacidades do algoritmo em tempo real. Foram também executadas simulações no simulador *Gazebo* de forma a validar a capacidade de um multi-rotor real seguir trajectórias agressivas geradas pelos algoritmos.

Palavras-chave: planeamento online, planeamento de trajectória, RRT, optimização de trajectória, TCAS, custos operacionais, multi-rotor

Abstract

In the present work a real-time trajectory planning algorithm for multi-rotor is developed. The algorithm is capable of avoiding both static and moving unexpected obstacles, in real time. The trajectory-planner is capable of generating optimal trajectories regarding operational costs, which are given by a combination of the estimated energy consumption and total trajectory time. The algorithm was integrated with a simplified TCAS system, showing the capability of the autonomous path planner to make use of a system originally created for manned aircrafts. The solution is based on a modified Rapidly-exploring Random Tree (RRT) algorithm and a trajectory optimization algorithm. The modified RRT algorithm allows to compute trajectories respecting a minimum curvature radius. An enhancement process was developed to quickly improve the length of the trajectory generated by the RRT. The trajectory optimization algorithm was developed aiming for using a small number of design variables, to improve real-time performance. The trajectory is defined by a series of 2nd degree splines. Multiple basic simulations were performed to evaluate the real-time capabilities of the algorithms. A simulation in the physics engine Gazebo was also analysed to validate the capability of a realistic multi-rotor to follow aggressive trajectories generated by these algorithms.

Keywords: online planning, trajectory planning, RRT, trajectory-optimization, TCAS, operational cost, multi-rotor

Contents

Acknowledgments v
Agradecimentos vii
Resumo ix
Abstract xi
List of Tables xvii
List of Figures xix
Nomenclature xxv
Glossary xxix

1 Introduction **1**
1.1 Context and Motivation 1
1.2 Online air-based path planning 2
1.3 Sense and Avoid Systems for UAVs 3
1.4 Objectives 4
1.5 Outline of the approach 4
1.5.1 Trajectory planner 5
1.5.2 Example 5
1.6 Contributions 7

2 Background **9**
2.1 Path-planning 9
2.2 Trajectory-Planning 11
2.3 Mathematical Optimization 11
2.3.1 IPOPT 12
2.3.2 Trajectory Optimization 14
2.4 Real time path planning for UAVs - Literature review 15
2.4.1 Literature review 15
2.5 Differential flatness 16
2.6 Multi-rotor-rotor dynamics 18
2.7 Quad-rotor differential flatness 19
2.8 Approximated dynamics 20

2.9 Traffic Collision Avoidance System 20

3 Modified RRT **23**

3.1 Basic RRT 23
3.2 Proposed algorithm 24
3.2.1 Geometrical considerations 26
3.2.2 Integration in the classic RRT algorithm 28
3.3 Enhancement step 31
3.3.1 Algorithm 31
3.3.2 Changing environments 32
3.4 Results 33

4 Trajectory Optimization **37**

4.1 Nomenclature 38
4.2 Problem formulation 38
4.3 Cost function 39
4.4 Kinematics 40
4.5 Maximum speed 41
4.6 Maximum acceleration 41
4.7 Obstacle clearance 42
4.7.1 Moving obstacles 44
4.7.2 Variable bounds 44
4.8 Tested formulations 45
4.9 Results 46
4.9.1 Complete computation 46
4.9.2 Comparison between formulations 48
4.10 Parameter tunning 50
4.10.1 Level Flight 51
4.10.2 Conflict resolution 51
4.10.3 Tuning the energy cost 52

5 Conceptual architecture **55**

5.1 Real-time trajectory-planner behaviour 56
5.1.1 Initial Computation 57
5.1.2 Regular Optimization 58
5.1.3 Trajectory Fix 58
5.1.4 Partial Regrow 59
5.1.5 Decision Maker 59
5.2 Integration with TCAS 60
5.2.1 Implemented simplified TCAS 60

6 Simulation **61**
6.1 Simplified Simulation . . . 61
6.1.1 Real-time avoidance . . . 61
6.1.2 Simulating an unknown environment . . . 66
6.1.3 Random maps . . . 66
6.1.4 Multi-Aircraft . . . 67
6.1.5 TCAS simulation . . . 69
6.2 Physics simulation . . . 70
6.2.1 RotorS Gazebo . . . 70
6.2.2 Controller . . . 70
6.2.3 Testing trajectory tracking . . . 71
6.3 TCAS system testing . . . 76
6.3.1 Results . . . 76

7 Conclusions **79**

References **81**

A Cost function, constraints and derivatives **85**
A.1 Cost function . . . 85
A.1.1 Time component . . . 85
A.1.2 Energy consumption . . . 85
A.2 Kinematics . . . 87
A.3 Maximum speed . . . 88
A.4 Maximum acceleration . . . 89
A.5 Obstacle clearance . . . 90
A.5.1 Spheres . . . 90
A.5.2 Sataic cuboids alligned with referential . . . 91

List of Tables

2.1 Traditional optimization problem representation. 12
2.2 Table defining the thresholds for TA and RA for different Sensitivity Levels (SL) [33]. . . . 22

3.1 Results of the application of the RRT algorithm and enhancement step to environment 1 . 33
3.2 Results of the application of the RRT algorithm and enhancement step to environment 2 . 33

6.1 Simulation run identifiers 72
6.2 Changed TCAS parameters for testing purposes. These parameters are explained in Section 2.9. 76

List of Figures

1.1 Part of the trajectory fixed (orange) in an initial trajectory computed by an RRT 5
1.2 Trajectory is optimized while UAV flies 5
1.3 New part of the trajectory is fixed and process re-starts 6
1.4 Unknown obstacle detected in the trajectory 6
1.5 Trajectory optimizer adjusts the trajectory avoiding the obstacle 6
1.6 Unknown complex obstacle detected in the trajectory 6
1.7 The maximum number of failures is reached and a trajectory is regrown around the obstacle 6

2.1 Seven degree of freedom robotic arm, taken directly from [15] 10
2.2 On the left a pointwise robot (red) in a 2D map. On the right the configuration space (defined by the x and y coordinates of the center of the robot), free configuration space is represented as yellow. 10
2.3 Timeline of the ITOMP solution taken directly from [29] 16
2.4 Scheme of the car system 17
2.5 Scheme of the topview of an hexacopter 19
2.6 Inertial frame (subscript W) and body frame (subscript B) of a multi-rotor. Taken directly from [30] 19
2.7 Symbology regarding intruder aircrafts in TCAS display. Taken directly from [33]. 22
2.8 Graphics showing the range threshold for a given closing rate (on the left) and the altitude separation threshold for a given altitude closing rate (on the right). The yellow line represents the TA threshold and the red line the RA threshold. 22

3.1 Initially only the start configuration is added to the tree 24
3.2 A random configuration is sampled. 24
3.3 A new configuration is created that expands the tree towards the random configuration. . 24
3.4 The new configuration and a new edge are added to the tree. 24
3.5 In some cases the expansion step fails, in those cases the new configuration is rejected . 25
3.6 The process continues until the goal configuration is added to the tree. 25
3.7 Vertices and edges 25
3.8 The green points and arrows represent the robot's position and speed respectively. The black points represent the vertices of the RRT. 25

3.9 The new vertex is in such a position that it is not possible to expand the tree directly towards it, the tree is expanded than using a maximum curvature step 26
3.10 Representation of the vectors $\mathbf{u}$ and $\mathbf{v}$ in blue 27
3.11 Circles describing the unreachable regions for a robot (in green) 27
3.12 Reachable regions in from a certain vertex (bounded by circles in blue), some limit possible robot positions are represented as green circles 28
3.13 Portion of an environment with an obstacle 30
3.14 Goal position falls into an unreachable region. Goal position represented as red, vertexes on the tree as black, vertex nearest to goal as green and the corresponding unreachable region as blue. 31
3.15 Consecutive steps in the enhancement algorithm. The blue circle identifies the *JumpBaseVertex* and the red circle identifies the *FurthestReachableVertex*. A green path represents a doable connection and a red path an impossible connection. 32
3.16 Environment 1 33
3.17 Environment 2 33
3.18 Example of the trajectory before (on the left) and after (on the right) the trajectory enhancement 35
3.19 Example of the trajectory before (on the left) and after (on the right) the trajectory enhancement 35
3.20 Example of the trajectory before (on the left) and after (on the right) the trajectory enhancement 35
3.21 Example of the trajectory before (on the left) and after (on the right) the trajectory enhancement 35

4.1 Intermediate points (blue) in trajectory segment (black) between consecutive waypoints (green arrows) for $M = 4$. 43
4.2 Trajectory segment between two waypoints (green arrows) crosses an obstacle (black line) without any of the waypoints being closer than d_{safe} to the obstacle. 44
4.3 Top view of the tested map 47
4.4 Tested map with textures 47
4.5 Comparison between the two optimizers, the computation time is represented in seconds. 47
4.6 Initial trajectory, to be optimized 48
4.7 Final (locally optimal) trajectory. 48
4.8 Computational time until local-minima is reached for both problems, using 100 runs for each. 49
4.9 Computational time until local-minima is reached for *Problem 2*, for different numbers of intermediate points per trajectory segment, averaged over 10 runs. 49

4.10 Trajectory before (top) and after (bottom) exchanging waypoints (green arrows) for intermediate collision checking points (blue circles). In this illustration the number of intermediate points per trajectory segment chosen was 2 (M=3). 49
4.11 Computational time until local-minima is reached for *Problem 2*, for different numbers of intermediate points per trajectory segment (replacing waypoints), averaged over 10 runs. 50
4.12 Locally optimal trajectory when 1 out of 2 waypoints are replaced by intermediate points. . 50
4.13 Locally optimal trajectory when 2 out of 3 waypoints are replaced by intermediate points. . 50
4.14 Locally optimal trajectory when 4 out of 5 waypoints are replaced by intermediate points. . 50
4.15 Trajectories for increasing values of k_h (from top to bottom) to force following a certain altitude. The results are shown for two possible first iterations, one that goes above the obstacle (trajectories the left) and another that goes below (trajectories the left). 52
4.16 Trajectory before (top) and after (bottom) k_c is set from 0 to a positive value. 52
4.17 Locally-ptimal trajectories for increasing energy cost (from left to right). Smaller arrows represent lower speeds. 53

5.1 Architecture of the software implementation 56
5.2 State machine describing the real-time trajectory planner. Red, blue and green states use methods from the **RRT**, **Optimizer** and **Feasibility Tester** scripts respectively. 57
5.3 Initial computation state. It Uses both the RRT and Optimizer methods. 57
5.4 *Regular Optimization* state. Uses **Optimizer** script. 58
5.5 Portion of trajectory that is optimized in the *Regular Optimization* state is represented as a series of blue arrows. The first and last blue arrows are treated as a local start and a local goal (these waypoints are fixed during the optimization). 58
5.6 *Trajectory Fix* state. Uses **Optimizer** script. 58
5.7 Portion of trajectory (blue) that is optimized in the state *Trajectory Fix*. The first and last blue arrows are treated as a local start and a local goal (these way-points are fixed during the optimization. 58
5.8 *Partial Regrow* state. Uses RRT methods. 59
5.9 Portion of trajectory (red) that regrown in state *Partial Regrow*. The first and last red arrows are treated as a local start and a local goal for the RRT algorithm. 59
5.10 Flowchart representing the functionality of the *Decision Maker* state. 60

6.1 RRT is grown 62
6.2 Initial trajectory is computed 62
6.3 The algorithm optimizes part of the trajectory ahead of the aircraft, it is visible that the optimized trajectory segment is smoother than the rest 62
6.4 Obstacle is detected 62
6.5 Trajectory optimizer adjusts the trajectory avoiding the obstacle 62
6.6 Obstacle is detected 62

6.7 Trajectory optimizer gets trapped in an unfeasible local minima, signalized by the red dashed ellipse (between the sphere and the cuboid obstacles). 62
6.8 RRT algorithm regrows the critical part of the trajectory 63
6.9 A feasible trajectory is found 63
6.10 Obstacle is detected critically close, trajectory optimizer would not have time to react in real time 63
6.11 All references are turned off to prevent the UAV from colliding. Previous trajectory is discarded 63
6.12 RRT algorithm regrows a new trajectory from the UAV position to the goal. 63
6.13 Before unexpected obstacle appears 63
6.14 Unexpected obstacle appears (top left of the figure), moving from the top to the bottom of the figure. 63
6.15 Trajectory is adjusted and aircrafts ”waits” for the obstacle to pass before proceeding. . . 63
6.16 Before the last moving obstacle appearance. 64
6.17 New obstacle appears, moving from the top to the bottom of the figure 64
6.18 Trajectory is adjusted. 64
6.19 The obstacle is avoided. 64
6.20 Trajectory while intruder flies to the right 64
6.21 Trajectory changes as intruder changes direction 64
6.22 Intruder is avoided 64
6.23 Initially algorithm plans to go below the intruder 64
6.24 Algorithm plans to go above the intruder 64
6.25 Intruder is avoided 64
6.26 Intruder is detected. As the intruder descends the algorithm does not detect a trajectory conflict 65
6.27 The trajectory is adjusted as the intruder starts to climb 65
6.28 Intruder starts to increase the climb rate. Trajectory is further adjusted. 65
6.29 Intruder starts to climb with such a rate that it becomes (apparently) impossible to go above it. Trajectory is adjusted to go under the intruder. 65
6.30 Intruder starts to descend 65
6.31 Algorithm (one second later) is informed that intruder is descending and tries to avoid it from above 65
6.32 Avoidance is successful 66
6.33 Complete map 66
6.34 Initially only two obstacles are visible, none of them interferes with the trajectory 66
6.35 An obstacle is detected and the trajectory is adjusted. 66
6.36 New obstacle is detected and the trajectory adjusted 66
6.37 Another obstacle is detected and the trajectory is adjusted 66
6.38 The final obstacle from the map is detected, it does not, however interfere with the trajectory. 66

6.39 Random unknown map. 67
6.40 Final executed trajectory. 67
6.41 Time, in seconds, taken for the agent to reach the goal position, on the unknown random maps. 67
6.42 Map with the starting and goal positions for aircrafts number 1 and number 2 68
6.43 Initial trajectories, computed simultaneously, in collision route. Both aircraft communicate their trajectories (scenario 1). 68
6.44 The algorithm, in an independent process for each aircraft, enables the generation of collision free trajectories (scenario 1). 68
6.45 The aircrafts avoid each other (scenario 1). 68
6.46 Initial trajectories, computed simultaneously, in collision route. Aircraft 2 is invisible for aircraft 1 (scenario 2). 68
6.47 The algorithm running for aircraft 2 plans a trajectory that avoids aircraft 1. Aircraft 1 plans a trajectory without considering the other aircraft (scenario 2). 68
6.48 The aircrafts avoid each other (scenario 2). 68
6.49 Both aircrafts are in collision route 69
6.50 TCAS systems determine resolutions and the trajectory-planner adapts the trajectories . 69
6.51 TCAS systems determine that the conflict is cleared and both aircrafts fly once again towards the goal. 69
6.52 Controller structure, taken directly from [46]. x_d represents the desired position speed and acceleration, $\mathbf{b}_{3_d}$ represents the axis perpendicular to the rotors' plane and $\mathbf{b}_{1_d}$ is related to the UAV heading. 71
6.53 Map visualization on the Gazebo simulator environment 71
6.54 Scheme of the top view of the Map. 71
6.55 Trajectory taken in run 1 (Max. acceleration = 6m/s^2). 72
6.56 Trajectory taken in run 2 (Max. acceleration = 6m/s^2). 72
6.57 Trajectory taken in run 3 (Max. acceleration = 10m/s^2). 72
6.58 Trajectory taken in run 4 (Max. acceleration = 10m/s^2). 72
6.59 Position error along the simulation time in run 1 (Max. acceleration = 6m/s^2). 72
6.60 Position error along the simulation time in run 2 (Max. acceleration = 6m/s^2). 72
6.61 Position error along the simulation time in run 3 (Max. acceleration = 10m/s^2). 73
6.62 Position error along the simulation time in run 4 (Max. acceleration = 10m/s^2). 73
6.63 Position error distribution in run 1. 73
6.64 Position error distribution in run 2. 73
6.65 Position error distribution in run 3. 73
6.66 Position error distribution in run 4. 73
6.67 Vehicle assuming an attitude to accelerate from the initial hovering position 74
6.68 Vehicle assuming an attitude to perform a curve trajectory segment 74
6.69 Simulated multi-rotor. 76

6.70 Intruder aircraft is detected. 77
6.71 TCAS declares proximate aircraft. 77
6.72 Resolution is provided by the TCAS. 77
6.73 Clear of conflict. 77
6.74 Trajectory of the aircrafts (altitude in the vertical axis and horizontal coordinate in the horizontal axis). 77
6.75 Altitude of one of the aircrafts over time with advisory markings. 78
6.76 Climb rate of one of the aircrafts over time with advisory markings. 78

.

Nomenclature

Greek symbols

λ	Lagrangian multipliers of equality constraint.
λ_i	Lagrangian multiplier of the ith equality constraint.
μ	Barrier parameter.
ϕ, θ, ψ	Roll, pitch and yaw angles.
$\phi_\mu(x)$	Barrier function.
σ	Robot state vector.
υ	Lagrangian multipliers of bound constraint.
υ_i	Lagrangian multiplier of the ith bound constraint.

Roman symbols

$c(x)$	Set of functions defining the equality constraints of an optimization problem (e.g. c(x)=0)
$d(x)$	Set of functions defining the inequality constraints of an optimization problem ($d_l < d(x) < d_u$)
$d_j(x)$	Function associated with the jth inequality constraint.
d_{lj}	Lower bound for the function associated with the jth inequality constraint.
d_{uj}	Upper bound for the function associated with the jth inequality constraint.
$f(x)$	Cost function associated with an optimization problem.
$\mathcal{I}$	Set of indexes of bounded design variables.
p, q, r	Angular velocities around the x_b, y_b and z_b axis of a multi-rotor.
s_j	Slack variable of the jth inequality constraint
u	Control input vector of a system.
w	Flat output vector of a differentially flat system.
x	Design variable vector

x_i ith design variable.

Subscripts

b Body fixed frame.

i, j, k, l Computational indexes.

w Fixed to the inertial frame.

ref Reference

Superscripts

T Transpose.

Glossary

RA Resolution Advisory is a resolution, provided as a suggested climb rate by the TCAS system, to solve conflicts with cooperative aircrafts.

RRT Rapidly exploring Random Trees is a family of sampling based planning algorithms.

SAA Sense and Avoid systems are capable of detecting and tracking intruders and avoid them.

TA Traffic Advisory is an audio signal provided by the TCAS to make the pilot aware of proximate traffic.

TCAS Traffic Collision Avoidance System is a certified airborne collision avoidance system that allows cooperative avoidance in manned aircrafts.

UAV Unmanned Air Vehicle is a powered aerial vehicle that does not carry an human operator.

Chapter 1

Introduction

In this chapter a description of the potential of autonomous Unmanned Air Vehicles is provided as a motivation for the current work. Then two of the main areas that are of interest to this work are introduced: online air-based path planning and sense and avoid systems. This chapter ends with the presentation of the objectives and outline of the present work, followed by the contribution.

1.1 Context and Motivation

With the constant development of Unmanned Air Vehicles (UAV) there has been an interest in using their potential for diverse applications. This potential could be further explored if autonomous UAV operation was possible, without the need for a human pilot. For a UAV to be operational without a human pilot it would have to be empowered with Artificial Intelligence (AI) that would allow it to deal with complex problems. Nowadays, the application of AI aiming for autonomous UAV operation is an interesting research topic. The following challenge is to proof these UAVs as safe and capable in order for them to be allowed to perform autonomously the desired applications. There is, however, a gap between the state of the art and these desired capabilities. For example, there has been an effort to integrate these vehicles in the non-segregated airspace [1]. To accomplish such, it must be ensured that UAVs satisfy certain safety authority guidelines and regulations [2]. However, there are not established regulations for unmanned vehicles, current work aiming to make UAVs capable of flying in non-segregated airspace derive these requirements from the UAVs based on existing regulations for manned aircraft.

Autonomous UAV operation require a diversity of capabilities. To enable the absence of a human pilot, the system requires some capabilities that are in part provided by humans, namely environment perception, motion planning and trajectory execution. In the terrain automobile industry, there has been many advances in the past years. This evolution was partially pushed by technological advancements in areas such as information technology, data analysis, computer vision, etc. [3].

The scenario for autonomous automobiles is however very different from others. For UAVs this problem is usually different. For such vehicles the environment is usually not as populated as for the autonomous cars. For a UAV the obstacles are usually other aircrafts and static ground obstacles. Due

to the smaller environment population and the greater freedom of movement of the intruder aircrafts it is usually defined a safe distance to these intruders much greater than the intruder size, making these algorithms more robust towards sensor uncertainty. On the other hand, for small UAVs, having a powerful computer on-board is not feasible due to the limited payload of these vehicles. A simple approach is then desirable to solve this problem because of computational limitations mentioned.

One of the efforts made in this sense is to provide Sense and Avoid (SAA) capabilities for these unmanned air vehicles, in such a way that the respective requirements for manned aircrafts are met. There has been, also, a general effort to move from centralized systems into networked and/or distributed ones. With the increasing complexity it seems advantageous to divide functionality into simpler components that cooperate among them [4]. It would then be desirable to create a distributed system. Usually SAA systems for UAVs can be divided in three major components: the aircraft and systems on-board, the ground station and communication links [4]. Some literature in SAA systems for UAVs present systems where processing and resolution decisions are taken partially or totally on the ground station can be found [4] [5]. However, a system based only on the aircraft would be desirable to make the aircraft behave like an independent component in a major system.

Autonomous systems are however complex, requiring a vast set of capabilities. This work will be focused in only a portion of that: the generation of collision free trajectories in real-time and the integration of the trajectory-planning algorithms with existing collision avoidance protocols for manned aircrafts.

1.2 Online air-based path planning

In robotics, path planning in unknown environments is a subject studied for many years [6]. In path planning problems for UAVs most of the existing solutions, to deal with unknown obstacles, require a dedicated ground station [4] [5] . This is due to the low computational power available on on-board computers in small UAVs. On-board path planning for small UAVs has been proposed in [7], using a field programmable gate array (FPGA) chip. In this work, a solution was created in which genetic algorithms are used to compute a path plan based on a provided environment and set of start and goal configuration. In [8], an online path planning algorithm for cooperative aircrafts is developed and implemented in relatively powerful on-board computers. In some works by Ioannis K. Nikolos et al. [9] [10], an evolutionary algorithm is developed which allows online path planning in unknown environments but this work considered static environments and it was never implemented in a vehicle.

Recently, in the end of 2018, Marco Pavone et al. [11] developed an online path planning algorithm that was shown to be able to compute trajectories in real-time in partially unknown environments with moving obstacles. The authors claim it was the first experimented algorithm, for multi-rotors, with such capabilities. The algorithms were, however, ran in a ground station. In another interesting work online path planning was accomplished with the environment being acquired by a depth camera [12].

The existing work regarding real-time path planning for multi-rotors in partially unknown environments with moving obstacles is not abundant and several ways of approaching the problem can be explored.

1.3 Sense and Avoid Systems for UAVs

An effective SAA system needs to supply two services in accordance with agreements reached at the *Sense and Avoid Workshops* where US FAA and Defense Agency experts discussed a number of fundamental issues [13]. They are a self-separation service that would act before a collision avoidance maneuver is needed, and a collision avoidance service to protect a small collision zone and usually achieved by an aggressive maneuver.

To achieve these services, the following list of sub-functions is required as it is described in [4]:

1. **Detect** any of various types of hazard, such as traffic, terrain or weather. At this step, it is merely an indication that something is there.

2. **Track** the motion of the detected object. This requires gaining sufficient confidence that the detection is valid, and making a determination of its position and trajectory.

3. **Evaluate** each tracked object, first to decide if its track may be predicted with sufficient confidence and second to test the track against criteria that would indicate that a SAA maneuver is needed. The confidence test would consider the uncertainty of the position and trajectory. The uncertainty could be greatest when a track is started, and again whenever a new maneuver is first detected. A series of measurements may be required to narrow the uncertainty about the new or changed trajectory. Also, when a turn is perceived, there is uncertainty about how great a heading change will result.

4. **Prioritize** the tracked objects based on their track parameters and the tests performed during the evaluation step. In some implementations, this may help to deal with limited SAA system capacity, while in others prioritization might be combined with the evaluation or declaration steps. Prioritization can consider criteria for the declaration decision that may vary with type of hazard or the context of the encounter (e.g., within a controlled traffic pattern).

5. **Declare** that the paths of own aircraft and the tracked object and the available avoidance time have reached a decision point that does indeed require maneuvering to begin. Separate declarations would be needed for self-separation and collision avoidance maneuvers.

6. **Determine** the specific maneuver, based on the particular geometry of the encounter, the maneuver capabilities and preferences for own aircraft, and all relevant constraints (e.g., airspace rules or the other aircraft's maneuver).

7. **Command** own aircraft to perform the chosen maneuver. Depending upon the implementation of the SAA, this might require communicating the commanded maneuver to the aircraft, or if the maneuver determination was performed on-board, merely internal communication among the aircraft's sub-systems.

8. **Execute** the commanded maneuver.

The main goal of this thesis is to provide a unmanned air vehicle the capability to go from a starting point to a goal in a dynamic, three-dimensional unknown environment. The Sense and Avoid capabilities in this work will be provided both by the path planner and a cooperative avoidance system inspired in an existing one for manned aircrafts.

The path planner approach, by generating collision free-paths, is not suitable for cooperative avoidance. When avoiding cooperative intruders it is highly desirable to follow an avoidance protocol. This ruled approach avoids situations where both aircrafts take, simultaneously, avoidance trajectories that lead to a new collision point. For this reason the path-planning algorithms developed in this work were integrated with existing collision avoidance systems. A simplified implementation of the Traffic Collision Avoidance System (TCAS) was integrated with the planning algorithms. A brief explanation on the TCAS system is provided in section 2.9. It will be described, further on in Section 4.10, how to adapt the algorithms developed in this work to empower them with features that makes them suitable for the present airspace.

1.4 Objectives

The main goal is to create an algorithm capable of planning aggressive trajectories (and for that the UAV dynamics must be considered) in partially unknown environments with moving obstacles. The algorithms should be able to quickly react to new detected obstacles (including moving obstacles) and safely avoid them by quickly adjusting the trajectory without having the need to stop the UAV for such. The trajectory-planner should also be capable of generating locally-optimal paths and be suitable for cooperative avoidance.

1.5 Outline of the approach

In this work it is proposed a real-time trajectory-planning algorithm for UAVs. The algorithm is designed for environments with both static and moving obstacles.

In the field of trajectory planning it is also desirable to compute optimal trajectories. Optimality will be defined in terms of mission costs (a combination between mission time and fuel/energy consumption). The algorithm will have anytime capabilities: it is possible to quickly generate a sub-optimal trajectory and then optimize it for a given period of time. A locally-optimal trajectory is computed if enough computation time is used. The trajectory will also be optimized while the UAV flies in order to improve the quality of the trajectories if they are not locally-optimal to begin with.

To enable path planning in real time a simplified dynamic model for multi-rotors will be used. In order to validate that the computed trajectories are suitable for aggressive multi-rotor maneuvering a simulation will be performed and the position error of the multi-rotor, relatively to the provided references, will be stored through the simulation and analysed afterwards.

The algorithms make use of trajectory optimization techniques. The optimization is performed by gradient based solvers that incrementally improve the quality of the trajectory. The first iteration (initial

trajectory) is computed using a modified RRT algorithm.

Further on the algorithms will be upgraded with other features, such as the capability of generating leveled trajectories at desired altitudes and follow desired climb rates.

The algorithms will finally be integrated with a simplified implementation of the TCAS system. This integration illustrates the capability of the proposed solution to respect this collision avoidance system (TCAS) originally designed for manned aircrafts.

1.5.1 Trajectory planner

The proposed solution consists of an incremental optimization approach. The core idea is based on continuously optimizing the trajectory while the UAV executes it. This, however, requires some additional logistics:

- It is required to define at each time which part of the trajectory should be optimized.
- It is required to have a first iteration for the trajectory optimizer.
- It is required to have a tool that prevents the optimizer to get trapped in unfeasible local minima.

It is required to define at each time which part of the trajectory should be optimized because the state on the trajectory that the UAV is executing in a given moment cannot be changed at that same moment and also, it makes no sense to optimize a part of the trajectory that was already executed. A practical implementation of this solution has an additional problem, the optimizer becomes significantly slower as the number of design variables is increased. For this reason, optimizing the entire trajectory is not always feasible, it is necessary to define a part of the trajectory that should be optimized at the time.

1.5.2 Example

A series of figures, Fig 1.1 - 1.7, will now be presented to exemplify this concept.

The selection of part of the trajectory that should be optimized at a given time will now be described. If the UAV is following the trajectory reference corresponding to the current time t_C, and the time used for an optimization increment is t_i, then the optimization is performed for the trajectory portion between $t_C + t_i$ and $t_C + t_i + t_n$ where t_n is a chosen parameter for a normal optimization scenario. The trajectory between t_C and $t_C + t_i$ is fixed once the optimization increment result will only be available at $t_C + t_i$. Once the optimization increment is finished, the current time t_C is updated and the process restarts (Fig 1.1 - 1.3).

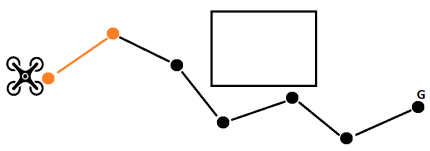


Figure 1.1: Part of the trajectory fixed (orange) in an initial trajectory computed by an RRT

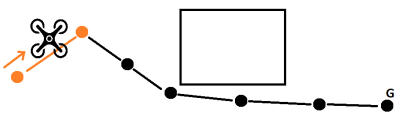


Figure 1.2: Trajectory is optimized while UAV flies

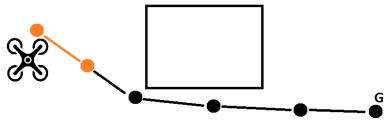


Figure 1.3: New part of the trajectory is fixed and process re-starts

In the case that there is some non-feasible portion in the trajectory (a part of the trajectory exceeds the maximum allowed speed or acceleration or there are collisions with obstacles) the trajectory is only optimized around the unfeasible portion (Fig 1.4 - 1.5).

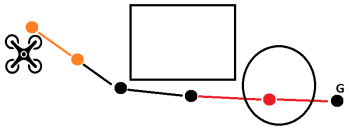


Figure 1.4: Unknown obstacle detected in the trajectory

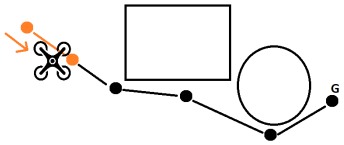


Figure 1.5: Trajectory optimizer adjusts the trajectory avoiding the obstacle

If the optimizer cannot make the trajectory feasible in a chosen, limited, number of increments (in this work it will be called the maximum number of failures MAX_f) the RRT algorithm regrows a trajectory around the unfeasible portion (Fig 1.6 - 1.7). This might happen if the trajectory falls into an unfeasible local minima.

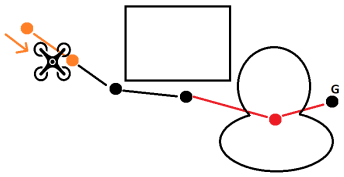


Figure 1.6: Unknown complex obstacle detected in the trajectory

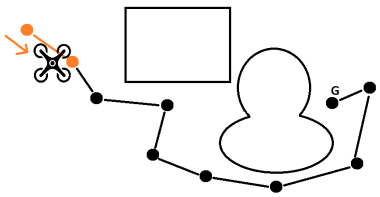


Figure 1.7: The maximum number of failures is reached and a trajectory is regrown around the obstacle

Finally, in the case that the non-feasible part of the trajectory corresponds to a time interval that is very close to the current time t_C, a different approach should be taken. In this scenario, the optimization increment might not be performed fast enough to avoid a collision. If this is the case, the UAV enters an ”emergency mode” which consists in stopping to follow the previous trajectory and recomputing the trajectory from scratch.

Concerning the first iteration given to the optimizer, it will be used a modified RRT. This RRT will also be used to regrow critical parts of the trajectory whenever it gets trapped in unfeasible local minima. The trajectory is computed from the UAV to the goal at the beginning of the mission and every time the robot discards the old trajectory when entering the ”emergency mode”.

1.6 Contributions

This thesis presents the following contributions:

- A real-time trajectory planning algorithm for multi-rotors in environments with moving obstacles.
- Integration of a simplified implementation of the TCAS system with the real-time trajectory planner.
- Simulation and evaluation of the algorithm performance.

To the best of the author's knowledge this work presents the following novelties:

- A curvature constrained RRT algorithm with an enhancement step for environments with moving obstacles.
- Analytical expressions for the distance to moving spheres with varying radius and the respective derivatives with respect to position and time.
- The integration of the TCAS system with an online trajectory-planner for multi-rotors.

The work was disseminated in the following ways:

- Presentation of the work: "Real-Time Path Planning for UAVs Using Improved RRT and Iterative Trajectory Optimization" in the "International Conference on Robotics and Robot Intelligence" in Vancouver, 2019. The work was produced in cooperation with Luis Romeiro, Professor Rodrigo Ventura, and Professor Afzal Suleman.
- The production of the articles: "Building the Bridge Between Autonomous and Manned Aircraft" and "Online Planner for Multi-Rotors based on Modified RRT and Trajectory-Optimization", in co-operation with Professor Rodrigo Ventura and Professor Afzal Suleman.

Chapter 2

Background

A brief overview of the relevant theoretical background for this work will be made in this chapter. The overview will cover the different types and categories of path-planning and present the definition of trajectory planning. Subsequently, it will be discussed mathematical optimization, and in more detail, the optimizer IPOPT, followed by a brief introduction on trajectory optimization.

The following topics are deeply related with the present work. A literature review on real-time path planning for UAVs is presented and special attention will be given to two works. Then the concept of differential flatness is presented before discussing the dynamics of the quad-rotor, followed by a section on why is the quad-rotor a differentially flat system. Finally, an approximation of the dynamics of the quad-rotor, which is often used in the literature associated with trajectory planning, is presented.

2.1 Path-planning

Automated planning is a relevant branch of artificial intelligence. It can be defined as the automated generation of a series of actions for an agent to take in order to accomplish a certain goal, which is essential in intelligent autonomous systems [6]. In order to keep this overview concise, the scope of this section will be narrowed to cover only path-planning applied to robots.

The configuration space $\mathcal{C}$ of a robot is generated by all the possible configurations a robot can acquire. The concept of configuration is described in [14] as a set of independent parameters that describe the position of every point in a body. For example, if a robot is a free point in a two-dimensional space, the configuration space is two dimensional and represents all the possible locations of the point on the map. If the robot is a mechanical arm with 7 degrees of freedom, as the one in Figure 2.1, given by 7 joints, the configuration space has 7 dimensions and corresponds to the robot configurations generated by any combination of the rotations of each joint.

Path planning for robots consists of generating a feasible path (series of configurations) that enables the robot to go from a start configuration to a goal configuration. Path planning algorithms are usually based on configuration space representations, for example, regular/occupancy grids, vertex graphs and Voronoi diagrams [6]. All the configurations in a path must lie in the free configuration space $\mathcal{C}_{free}$, which

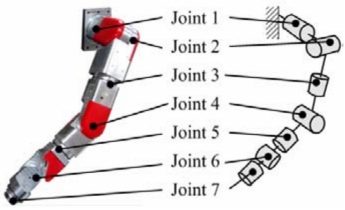


Figure 2.1: Seven degree of freedom robotic arm, taken directly from [15]

can be defined as the space formed by all the configurations that the robot can assume without being in collision with any obstacle.

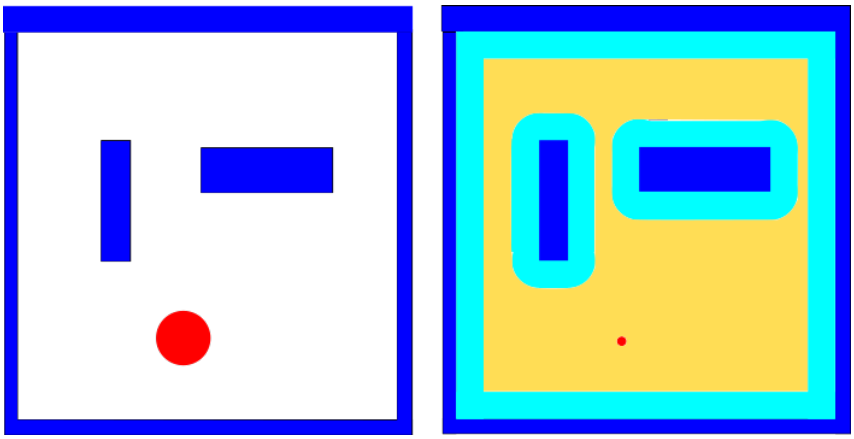


Figure 2.2: On the left a pointwise robot (red) in a 2D map. On the right the configuration space (defined by the x and y coordinates of the center of the robot), free configuration space is represented as yellow.

The path planning algorithm choice is related to the chosen configuration space representation. A review on path-planning algorithms was performed by Yang et. al. [16]. In this work, path planning algorithms are separated into five distinct categories:

- **Sampling based algorithms** - These sort of algorithms work by randomly sampling configurations (usually referred to as nodes) from a continuous configuration space. These nodes are used to create a path. These algorithms can fall into two categories: active or passive. Active algorithms, such as the Rapidly Exploring Random Trees (RRTs) and its variations, sample a series of nodes from the configuration space and, using these nodes, create a path. Passive sampling-based algorithms, such as the Probabilistic Road Map (PRM), sample the configuration space, creating a graph that contains the start and the goal, in order to obtain a path. However, these algorithms rely on a search algorithm to compute a path from the generated graph.
- **Node based optimal algorithms** - These algorithms explore the configuration space through a previously created decomposed graph. It is then possible to find optimal paths within the used graph. Some popular algorithms in this category are the A* and Dijkstra's algorithm, which are also described often as discrete optimal planning, road map or search algorithms.
- **Mathematic model based algorithms** - In this case, the environment and the system dynamics are described mathematically. A cost function is defined and a series of bounds are created. After

modelling the problem mathematically, it is possible to find a locally-optimal solution. In such a solution, any small disturbances in the final trajectory will increase its cost or violate the problem constraints.

- **Bio-inspired algorithms** - These algorithms fall into two categories: Neural Networks and Evolutionary algorithms (Genetic Algorithm GA is an example), which mimic biological behaviour to solve problems.

- **Multi-fusion based algorithms** - In [16] this category is used to describe combinations of algorithms, such as the required combination of Probabilistic Road Maps with a search algorithm.

Path-planning algorithms can be further categorized according to their capabilities [17].

- An algorithm has **anytime** capabilities if it is capable of producing a sub-optimal trajectory at first and then improve the solution while there is computational time available.

- An algorithm is considered **dynamic** if it is able to adapt the computed path in changing environments (partially unknown).

- An **online** planner interleaves planning and execution. These algorithms adapt the plan as new information in the agent and/or environment is available. The online capability is related to the dynamic capability described before. To be able to have an acceptable online performance an algorithm must be able to quickly adapt/recompute a trajectory and, for this reason, the computational time is of extreme relevance for these algorithms.

The term *online* will be used, in this work, interchangeably with *real-time*.

2.2 Trajectory-Planning

The terms path and trajectory are often used interchangeably. Formally, however, these terms are distinct. While path refers to, as mentioned before, all the consecutive configurations that a robot has to assume in order to go from a start to a goal configuration, in a trajectory there has to be a time moment associated to each of the configurations that define the trajectory. In a trajectory the speed and acceleration can be obtained simply by derivation [18].

2.3 Mathematical Optimization

Mathematical optimization or mathematical programming has been a studied problem for centuries. Bradley et al. [19] describe mathematical programming, in a management related perspective, as an optimal allocation of limited resources under a set of constraints imposed by the nature of the problem.

The problems approached in mathematical optimization will now be described in a more general way. The concept behind an optimization problem is simple: to determine a series of quantities that minimize

a cost function while respecting a series of constraints [1]. Some very basic nomenclature will now be presented.

- The quantities to be determined are called **design variables**.
- The final goal is to minimize a **cost function** that is a function of the design variables.
- In real problems the design variables cannot usually assume any combination of values, there are **constraints** to be respected which are formulated as equalities or inequalities.

Formally, these problems are presented in the following form:

minimize:	$f(x)$
subjected to:	$c(x) = 0$
	$d(x) \geq 0$

Table 2.1: Traditional optimization problem representation.

where x is the vector of design variables, $f(x)$ the cost function and $c(x) = 0$ and $d(x) \geq 0$ are series of equalities and inequalities, respectively, that represent the problem constraints. Reinforcing this idea, the cost function and the constraints ($f(x)$, $c(x)$ and $d(x)$) are characteristics of the problem and define the problem. The problem is solved when a combination of design variables $x_{solution}$ is found such that $f(x)$ (the cost function) assumes its minimum possible value while respecting $c(x) = 0$ and $d(x) \geq 0$ (the constraints of the problem).

There are two types of solutions for the optimization problem: global solutions and local solutions. Intuitively $x_{solution}$ is a global solution if it originates the minimum possible value of the cost function while respecting the constraints. On the other hand $x_{solution}$ is a local solution if it is associated with a minimum value of a cost function within a certain neighborhood while respecting the problem constraints.

There are multiple methods for solving this sort of problems, they will not, however, be reviewed in the present work. As stated in [20] most of the methods used for trajectory optimization are based on Newton's method for finding the solution to a numerical problem. There are then multiple possible formulations for considering the problems equalities and inequalities. Some popular formulations for optimization problems are unconstrained optimization, sequential quadratic programming and barrier methods [20]. There are also techniques which are not based on Newton's method, such as genetic algorithms and particle swarm optimization.

2.3.1 IPOPT

In the present work, the used solver is an implementation of an interior point with a filter line-search method called IPOPT [21]. The algorithm is complex and uses a primal-dual barrier method, originally developed by Andreas Wächter in [22], inspired in previous work by Fiacco et al. [23].

In sequential quadratic programming (SQP) there is the need to identify the active bound constraints. Barrier problems avoid this problem by replacing the constraints for logarithmic terms in the cost function. Thus being said, barrier methods solve optimization problems of the type:

[1] This explanation is inspired in the website from IBM: http://ibmdecisionoptimization.github.io/docplex-doc/mp.html

$$\begin{aligned} \text{min:} \quad & f(x) \\ \text{s.t.:} \quad & c(x) = 0 \\ & x_i > 0 \qquad , i \in \mathcal{I} \end{aligned} \tag{2.1}$$

where $\mathcal{I}$ defines the set of indexes of bounded variables. By converting the problem into the following one, stated in Equation 2.2, called a *barrier problem*:

$$\begin{aligned} \text{min:} \quad & \phi_\mu(x) = f(x) - \mu \textstyle\sum_{i \in \mathcal{I}} ln(x_i) \\ \text{s.t.:} \quad & c(x) = 0 \end{aligned} \tag{2.2}$$

where μ is the barrier parameter, which is greater than 0. This formulation imposes that $x_i > 0$ for $i \in \mathcal{I}$ because, as x_i tends to 0, the value of $\phi_\mu(x)$, the barrier function, tends to infinite. The formulation in Equation 2.2 matches the problem defined in Equation 2.1 for $\mu \to 0$. For $\mu > 0$, it becomes clear that the solution never lies in the boundary of the problem, instead it always lies in the interior of the desired design variable region. For this reason, the barrier methods are also called interior point methods. In these methods, multiple sub-problems (barrier problems) are solved for decreasing values of μ. These optimization sub-problems are converted into a set of equations to be solved. Before presenting those equations, the reader must notice that the derivatives $\frac{\partial \mu \sum_{i \in \mathcal{I}} ln(x_i)}{\partial x_i}$ are $\frac{\mu}{x_i}$. The terms $\frac{\mu}{x_i}$ will be replaced for υ_i and an extra constraint will be added to ensure that $\upsilon_i = \frac{\mu}{x_i}$, these new variables are named as Lagrangian multipliers for the bound constraints in [21]. The optimization problem is then formulated as the following series of equations:

$$\begin{aligned} \nabla f(x) + A(x)\lambda - \upsilon = 0 \\ c(x) = 0 \\ x_i \upsilon_i = \mu \qquad\qquad i \in \mathcal{I} \end{aligned} \tag{2.3}$$

where $A(x)$ is the transpose of the Jacobian of $c(x)$, λ stands for the vector of Lagrangian multipliers for the equality constraints and υ is a vector in which each entry υ_i is equal to zero except for $i \in \mathcal{I}$ (in those cases $\upsilon_i = \frac{\mu}{x_i}$ as stated before). The system is then solved for x, υ and λ. The step sizes for each of these variables at the kth iteration: Δ_x^k, Δ_υ^k and Δ_λ^k, are determined using Newton's method. However, this is not performed directly in the entire equation system. Instead, the system is separated and Δ_x^k and Δ_λ^k are calculated separately from Δ_υ^k, to improve the computational time. After calculating the search direction, the line-search filter determines the step size taken in that direction. More details can be found in [21].

IPOPT allows solving general dual-barrier problems of the type:

$$\begin{aligned} \text{min:} \quad & f(x) \\ \text{s.t.:} \quad & c(x) = 0 \\ & x_{il} < x_i < x_{iu} \qquad , i \in \mathcal{I} \\ & d_l < d(x) < d_u \end{aligned} \tag{2.4}$$

where x_{il} and x_{iu} are lower and upper boundaries for x_i. d_l and $d_j(x)$ are lower and upper boundaries

for the inequality functions $d(x)$. In order to convert the typical barrier problem stated in 2.1 into the more general form shown in 2.4, it is required to, instead of using the variables $v_i = \frac{\mu}{x}$, create the variables $v_{il} = \frac{\mu}{x_i - x_{il}}$ and $v_{iu} = \frac{\mu}{x_{iu} - x_i}$. Doing this, the equalities forcing these new definitions of v_{il} and v_{iu} become $v_{il} * (x - x_{il}) = \mu$ and $v_{iu} * (x_{iu} - x_i) = \mu$ respectively.

In order to explain how IPOPT deals with inequalities defined by general functions $d(x)$, it is defined an isolated inequality:

$$d_{jl} < d_j(x) < d_{ju} \tag{2.5}$$

where $d_j(x)$ represents the jth inequality function and d_{jl} and d_{ju} represent the respective lower and upper bounds. These inequalities are changed to a set of equalities and inequalities shown in Equation 2.6 by introducing slack variables s_j:

$$\begin{aligned} d_j(x) - s_j &= 0 \\ d_{jl} < s_j &< d_{ju} \end{aligned} \tag{2.6}$$

The equalities are treated then as regular equalities. The inequalities $d_{jl} < s_j < d_{ju}$ are then imposed in the same way as the inequalities $x_{il} < x_i < x_{iu}$.

2.3.2 Trajectory Optimization

Optimization techniques can be used to compute trajectories. However, it is required to state an optimization problem in such a way that its solution can be used to compute a trajectory. In [20], an overview of trajectory optimization techniques and applications is performed. Besides giving an overview in numerical methods for optimization, in [20] a series of traditional formulations of trajectory planning problems as optimization problems is given. Usually, in these problems, the system dynamics is imposed using equality constraints such as:

$$\dot{\sigma} = f(\sigma, u) \tag{2.7}$$

where σ represents the system state and u the control inputs. Nevertheless, these equalities must be formulated in a discrete-time domain. To achieve this, the derivatives $\dot{\sigma}$ are usually taken using finite differentiation.

Trajectory optimization has been applied also to multi-rotors. In [11], optimization is only used to compute splines joining way-points in a previously computed trajectory. On other works, such as [12, 24], trajectory optimization is used to completely compute trajectories. In [24], the trajectories are discretized into a series of way-points, unlike in [12] where trajectories are described as a series of high order splines, which parameters are optimized.

To sum up, in order to perform trajectory optimization it is required to choose a finite set of parameters to describe the trajectory, such as a set of way-points or a sequence of control inputs, and then formulate the problem in such a way that the system dynamic and actuation limits of the problem are respected.

Finally, a cost function to be minimized must be chosen to minimize.

2.4 Real time path planning for UAVs - Literature review

2.4.1 Literature review

Online path planning in unknown environments for UAVs was accomplished before in [25], but it ignored the vehicle dynamics. Therefore, the UAV would have to fly slowly in order to perform the computed trajectories. The most commonly used methods for real-time path planning are, arguably, sample-based methods. For example, basic RRTs allow a fast search for a trajectory, however the solution is sub-optimal.

For accomplishing a proper path-planning, it is desirable that the algorithm considers the kinodynamics of the problem. Attempts have been made to apply RRT*, an asymptotically optimal version of the RRT, for this problem.In [26], a kinodynamic RRT* was proposed, however in this work, the times required to compute a trajectory were in many cases greater than 100 seconds, making it unsuitable to use for online path-planning problems.

An interesting approach for path-planning problems is trajectory optimization. It has been used for many years with diverse applications, for example to define missions of orbit transfer and also launching space vehicles, like it is referred in [20]. In recent years,there has been a great amount of work regarding applying these algorithms to robotics. One of the most influential works was done by Matt Zucker et al. [27], where motion-planning for articulated robots was performed using optimization techniques. It discretized the trajectory as a series of configurations, and then the algorithm would minimize the distance to obstacles and maximize the smoothness of the trajectory. However, the trajectory time would have to be pre-determined. Despite having clear limitations, in a prior work, A. Richards et al. [24] addressed motion planning for UAVs. More recently, in 2016, Helen Oleynikova et al. [12] developed an algorithm for UAV path planning in unknown static environments.

Generally, trajectory optimization algorithms require a first iteration. This can be computed in many different ways, for example, it is possible to use a straight line as a first trajectory, even if this one is not obstacle free [28]. In the present work, it is presented a first iteration given by a modified RRT algorithm, which not only guarantees that the local-minimum is feasible as it also provides a trajectory close to satisfying the kinodynamic constraints of the problem.

Two different works [11, 29] are of particular interest for the current research.

In [11] Marco Pavone et al. developed an algorithm which they claim to be the first successful real-time path planner for UAVs in unknown dynamic environments physically tested. This work is very recent (very end of 2018). The processing, however, was made on a ground station and communicated with the UAV. The authors claim that the algorithms can run in real-time in an on-board computer if their code is converted from MATLAB to C++. This work combines a series of interesting features such as the use of machine learning to allow an online performance of a kinodynamic, asymptotically optimal sample-based planner, in this case, the Fast Marching Tree star (FMT*). In [11], the authors propose the usage

of three control input terms for the quadcopter: a feed forward, a feed backward and a reactive term. The feed forward and feed backward terms simply compel the quadcopter to follow the computed path (the control references and the feedback of the error, as in traditional controllers). The reactive term, like in a potential field approach, simulates a force that ”pushes” the quadcopter away from the obstacles. This last control term is, as mentioned by the authors, not essential, however, it has led to improvements in the performance of the solution.

In [29], the authors explore the fact that trajectory optimization techniques allow the computed trajectory to be continuously improved while the robot executes it. Figure 2.3 shows the timeline of the solution, directly taken from [29], where the steps 2,3, etc. are similar to step 1.

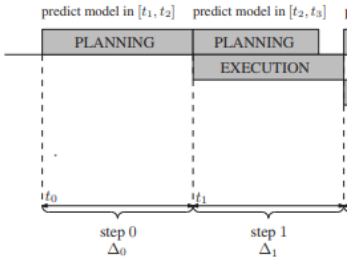


Figure 2.3: Timeline of the ITOMP solution taken directly from [29]

It was experimented in simulations for high degree of freedom robots and showed to be successful in achieving real-time path planning in unknown environments. It was also confirmed that this planner able to deal with moving obstacles, however, it would fail to provide a collision-free trajectory quickly enough in some scenarios with moving obstacles. To deal with the static obstacles the authors threat the environment as a voxel map. The robot is treated as a series of overlapping spheres. For moving obstacles the algorithm does not have information on the distance to those moving obstacles. Instead, it only checks the distance to those moving obstacles. The moving obstacles trajectory is predicted during a small period of time. The safety distance to be kept to those obstacles is increased along the time. The increasing safety distance is used to deal with the uncertainty on the trajectory of the moving obstacles. The algorithms only allow the robot to execute the trajectory if it is safe within a defined period of time.

2.5 Differential flatness

Many systems are under-actuated, meaning that the systems cannot be commanded to follow arbitrary trajectories in the configuration space. This happens for example when the input space dimension is smaller than the output space (configuration space) dimension, in this case, the systems are described as trivially under-actuated. One example of a trivially under-actuated system is a quad-rotor, with only 4 inputs (speeds of the 4 rotors) and a 12-dimensional configuration space [30].

The concept of differential flatness is now described. A system is considered flat if it is possible to find a set of outputs, which can be computed from the robot state, and a set of finite derivatives of the input (equal, in number, to the number of inputs), called *flat outputs*, such that it is possible to describe

all the possible trajectories of the system directly from the flat outputs and their derivatives [31]. In a formal way, in a system:

$$\dot{\sigma} = f(\sigma, u) \qquad , \sigma \in \mathbb{R}^m, u \in \mathbb{R}^n$$

where σ represents the system state and u, the control input, is differentially flat if it is possible to find the flat outputs w:

$$w = g\left(\sigma, u, \dot{u}, \ddot{u}, ..., u^{(l)}\right) \qquad , u \in \mathbb{R}^n$$

Such that it is possible to describe the inputs and outputs as functions of the flat outputs and its derivatives:

$$\sigma = \sigma\left(w, \dot{w}, ..., w^{(l)}\right) \qquad u = u\left(w, \dot{w}, ..., w^{(l)}\right)$$

where $u^{(l)}$ and $w^{(l)}$ represent the lth derivative of u and w respectively.

Note that usually in literature, the state vector is usually described by x, however, in the present work it will be used σ once x will be frequently used for addressing the set of design variables in an optimization problem.

One example of a differentially flat system is the basic model of a car. Let x and y be the coordinates of the point between the rear wheels of the vehicle. θ is the orientation of the vehicle, determining the angle between the symmetry axis of the vehicle (pointing forward) and the x axis. The control inputs of the system will be the speed of the middle point between the rear wheels of the vehicle v and the angle between the symmetry axis of the vehicle and the front wheels' direction δ. Let the distance between the rear wheel axis and the front wheel axis be b.

$$\sigma = [x, y, \theta] \quad u = [v, \delta]$$

The system scheme is shown in Figure 2.4

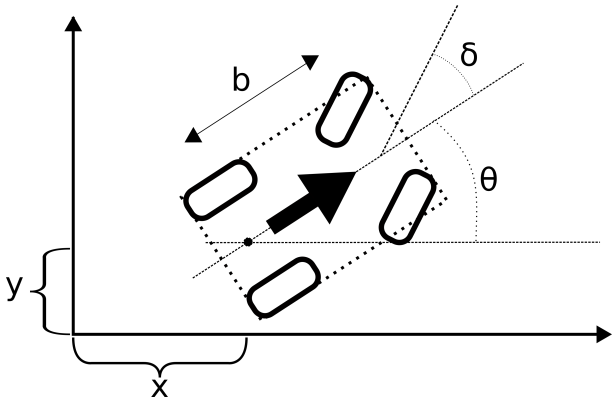


Figure 2.4: Scheme of the car system

In this system it is possible to use x and y as the variables forming the flat outputs.

$$w = [x, y]$$

From this variables and its derivatives it is possible to compute the state of the car and the control inputs, as shown in Equations 2.8 and 2.9:

$$\sigma = [x, \quad y, \quad arctan2\,(\dot{y}, \dot{x})] \tag{2.8}$$

$$u = \left[\sqrt{\dot{x}^2 + \dot{y}^2}, \quad atan2\left(-\ddot{x}\frac{\dot{y}}{\dot{x}^2 + \dot{y}^2} + \ddot{y}\frac{\dot{x}}{\dot{x}^2 + \dot{y}^2}, \frac{\sqrt{\dot{x}^2 + \dot{y}^2}}{b}\right)\right] \tag{2.9}$$

2.6 Multi-rotor-rotor dynamics

It is now introduced the concepts of inertial frame and body frame. The inertial frame is, as the name suggests, a frame that moves in a constant speed (or static), where the z_w axis is aligned with the vertical, pointing upwards. The body frame is fixed to the multi-rotor and the z_b is perpendicular to the rotor plane and points up when the multi-rotor is hovering. A representation of these frames is shown in Fig 2.6.

As stated in [30], the UAV state can be written as the position of its center of mass (x, y and z), the speed of the center of mass ($\dot{x}$, $\dot{y}$ and $\dot{z}$), the roll pitch and yaw angles (ϕ,θ and ψ) and the angular velocities around the body axis x_b, y_b and z_b (p, q and r respectively).

$$\sigma = [x, y, z, \phi, \theta, \psi, \dot{x}, \dot{y}, \dot{z}, p, q, r]^T \tag{2.10}$$

The inputs are considered to be the resulting force along the z_b component of the body frame of the vehicle (T) and the moments along x_b, y_b and z_b of the vehicle (M_x, M_y and M_z). This thrust and moments are assumed to be given linearly from the square of each rotor speed (ω_1^2, ... , ω_4^2). It is also assumed that the rotor speeds can be directly controlled. Let $u = [T, M_x, M_y, M_z]^T$ and $\omega^2 = [\omega_1^2$, ... , $\omega_4^2]^T$. For multi-rotors with more rotors, the control input u will be the same (thrust along the axis perpendicular to the rotors plane z_b and moments applied to the center of mass), once a series of rotors that provide thrust in the same direction are only capable of generating force along that same direction. To map from the squared rotation speed of n rotors ω^2 into control inputs u, as described before, it is used a matrix usually called allocation matrix A $4 \times n$, such as in Equation 2.11.

$$u = A[\omega_1^2, ..., \omega_n^2]^T \tag{2.11}$$

For example, for the hexa-copter shown in Figure 2.5, the allocation matrix A that converts the squared rotation speed of the six rotors $[\omega_1^2, ..., \omega_6^2]^T$ into the control input vector u composed by the

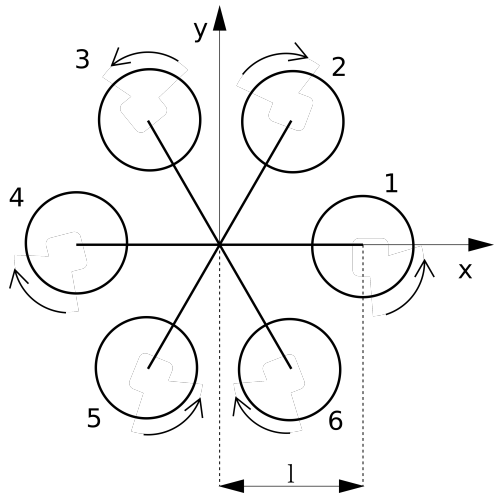


Figure 2.5: Scheme of the topview of an hexacopter

thrust along the z_b axis and the moments around the body axis $u = [T, M_x, M_y, M_z]^T$ is:

$$A = \begin{bmatrix} C_T & C_T & C_T & C_T & C_T & C_T \\ 0 & -lC_T\frac{\sqrt{3}}{2} & -lC_T\frac{\sqrt{3}}{2} & 0 & lC_T\frac{\sqrt{3}}{2} & lC_T\frac{\sqrt{3}}{2} \\ lC_T & l\frac{C_T}{2} & -l\frac{C_T}{2} & -lC_T & -l\frac{C_T}{2} & l\frac{C_T}{2} \\ -C_M & C_M & -C_M & C_M & -C_M & C_M \end{bmatrix} \tag{2.12}$$

Where the thrust and momentum along the axis of a rotor are given by $T = C_T\omega^2$ and $M = C_M\omega^2$.

It is clear that for more than 4 rotors the matrix does not have an inverse. For that reason, when it is desired to map u into $[\omega_1^2, ..., \omega_n^2]$, with $n > 4$, it is necessary to add some extra constraints to the system, making it possible to directly obtain $[\omega_1^2, ..., \omega_n^2]$ from u.

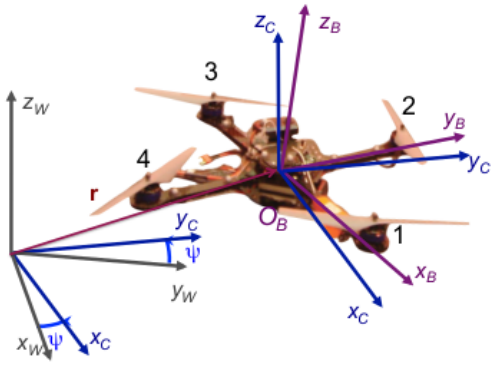


Figure 2.6: Inertial frame (subscript W) and body frame (subscript B) of a multi-rotor. Taken directly from [30]

2.7 Quad-rotor differential flatness

In [30], it is proven that a quad-rotor is a differentially flat system. This implies that it is possible to compute the control inputs and the UAV configuration from a trajectory defined in the flat output space.

The flat outputs chosen in this work were the vehicles center of mass coordinates in the inertial frame (x, y and z) and the yaw angle (ψ).

$$w = [x, y, z, \psi]^T \tag{2.13}$$

The authors provide expressions for computing the quad-rotor state σ and inputs u from the flat output variables and their derivatives up to the fourth derivative. This means that it is possible for the under-actuated quad-rotor to follow the computed trajectory in the flat output space (position and yaw) as long as the inputs are not saturated.

2.8 Approximated dynamics

In this work, like in some others [11, 24], a first approximation is made, considering that the acceleration of the vehicle can be directly controlled (ignoring attitude information). In such approximated system, the UAV can be described as a point with a mass, and its state considers only the position of the center of mass and the speed. This approximation, however, leads to generated trajectories that cannot be followed by the real quad-rotor. Discontinuities on the acceleration direction would translate into discontinuities in the vehicle attitude (the z_b axis of the body frame must be aligned with the thrust at all times). If it is desired to make use of the differential flatness propriety, computing directly the control inputs from the flat output variables (from a trajectory in 3-dimensional space with freedom to control the yaw angle), it is required to have information on the flat output variables up to the 4^{th} derivative (snap) with respect to time. This can be achieved by smoothing the trajectory, as it is made in [11].

In the present work, the computed trajectories only provide information up to the 2^{nd} derivative of the position. It is then desirable to have a suitable controller, once the inputs cannot be directly taken from the computed trajectory.

2.9 Traffic Collision Avoidance System

The Traffic Alert and Collision Avoidance System (TCAS) is an on-board conflict detection and resolution system which alerts pilots to the presence of nearby aircraft that pose a mid-air collision threat and issues conflict resolution advisories, [32]. TCAS is a complex safety-critical system in the area of air traffic management. TCAS is able to operate independently of the ground-based air traffic control (ATC) system.

TCAS systems provide different degrees of alerting regarding air-born collision. To do so, these systems rely on the communication between the beacon transponders on-board of the aircrafts.

The TCAS system provides two types of alerts to the pilot: Traffic Advisories (TA) and Resolution Advisories (RA). A traffic advisory (TA) is a sound message that alerts the pilot for proximate cooperative traffic, with possible risk of collision. Resolution advisories (RAs) are sound messages stating actions for pilots to take in order to avoid collision by assuring a vertical separation between aircrafts (climb,

descend, level...).

A TCAS display assists the pilot visualising proximate aircrafts. The intruder aircraft are represented using different color and shape codes accordingly to the alert level associated with themselves. TCAS displays often include a color-coded climb rate display. In this display pilots can visualize the recommended climb rate by the TCAS (in case of resolution advisory) to avoid collision.

In case of resolution advisory the TCAS systems on board of the conflicting aircraft coordinate the computed resolutions to avoid collision (for example avoiding that both aircrafts start climbing towards a new collision point). The TCAS systems will then select complementary resolutions.

Regarding the display symbology, both colour and shape are used to assist the pilot in interpreting the displayed information. Own-aircraft is depicted as a white or cyan airplane-like symbol. Other aircraft are depicted using geometric symbols, depending on their threat status, as follows [33]:

- An unfilled diamond, shown in either white or cyan, but not the same colour as own-aircraft symbol, is used to depict non-threat traffic (labelled as “Other”);
- A filled diamond, shown in either white or cyan, but not the same colour as own-aircraft symbol, is used to depict Proximate Traffic (non-threat traffic that is within 6 nmi and ± 1200 ft from own-aircraft);
- A filled amber or yellow circle is used to display intruders that have caused a traffic advisory to be issued;
- A filled red square is used to display intruders that have caused an RA to be issued.

The standardized symbology is shown in Figure 2.7.

In order to declare and resolve conflicts, the TCAS system tracks the intruders range and altitude. Based on the range measurements, the TCAS computes the time to the closest point of approach (CPA), which will be described in this section as the range tau τ_R. τ_R is calculated simply by dividing the range by the closure rate. Analogy, the vertical tau τ_V is the time for altitude-crossing and is calculated by dividing the altitude difference by the vertical closing rate. There are defined thresholds for issuing traffic and resolution advisories [33]. If both τ_R and τ_V are below these thresholds a traffic advisory or resolution advisory will be issued. For ranges smaller than a defined range called DMOD the TCAS behaves like τ_R is below the corresponding threshold. Analogously, if the altitude separation is smaller than a certain threshold called ZTHR the TCAS behaves like τ_V is below the corresponding threshold. These thresholds are defined by the sensitivity level, which is determined by the altitude of the aircraft. Table 2.2 shows these thresholds for each sensitivity level and the altitudes corresponding to each sensitivity level:

To illustrate this alert system, Figure 2.8 is presented. If an intruder aircraft set of parameters falls below the yellow line in both graphics then a traffic alert is issued. Similarly, if an intruder aircraft set of parameters falls below the red line in both graphics then a resolution advisory is issued. The slope of the yellow line and red line is the TA tau and the RA tau respectively.

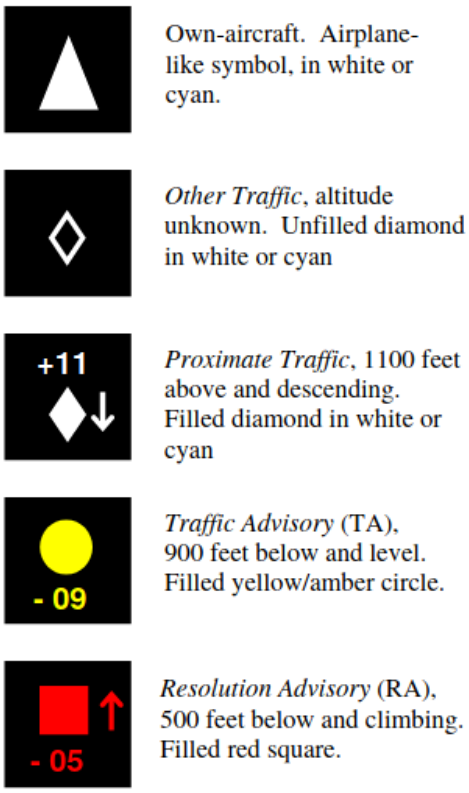


Figure 2.7: Symbology regarding intruder aircrafts in TCAS display. Taken directly from [33].

Own Altitude (feet)	SL	τ (seconds)		DMOD (nmi)		ZTHR (feet)	
		TA	RA	TA	RA	TA	RA
$<$1000	2	20	N/A	0.30	N/A	850	N/A
1000-2350	3	25	15	0.33	0.20	850	600
2350–5000	4	30	20	0.48	0.35	850	600
5000–10000	5	40	25	0.75	0.55	850	600
10000–20000	6	45	30	1.00	0.80	850	600
20000–42000	7	48	35	1.30	1.10	850	700
$>$ 42000	7	48	35	1.30	1.10	1200	800

Table 2.2: Table defining the thresholds for TA and RA for different Sensitivity Levels (SL) [33].

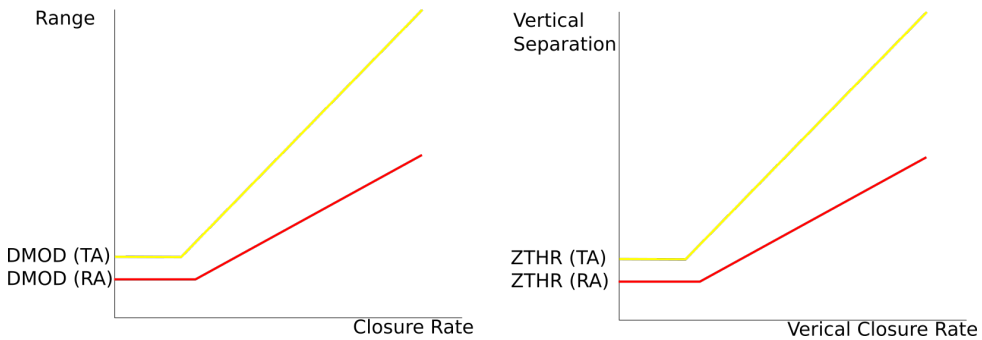


Figure 2.8: Graphics showing the range threshold for a given closing rate (on the left) and the altitude separation threshold for a given altitude closing rate (on the right). The yellow line represents the TA threshold and the red line the RA threshold.

Chapter 3

Modified RRT

There is a vast literature on different RRT algorithms and the possible applications for each. This work, however, will fail to make an overview of these algorithms, to keep it concise. Instead, a brief explanation of the basic RRT algorithms is presented, followed by the description of the proposed algorithm. Finally an enhancement method is presented, which allows to improve the solution of the proposed algorithm.

3.1 Basic RRT

Rapidly-exploring Random Trees (RRTs) were introduced by LaValle as a family of randomized planners ([34], chapter 5). LaValle introduced RRT algorithms in a general form that allow planning trajectories for high degree of freedom systems. The original algorithm consisted in expanding a tree of states be making incremental expansions from those configurations to randomly sampled new states. Each state in the tree is described as a vertex, each connection between states is described as an edge.

In the year 2000, Kuffner and LaValle [35] proposed an RRT algorithm that outputs a path from a start to a goal configuration called RRT-connect, this variation uses two trees. It will be explained how the original algorithm, using a single tree, can be used to compute a path from a start to a goal configuration while biasing the growth of the algorithm, as it is described in [36].

Initially an empty tree is created, and the start configuration is added to that tree. Then the main cycle begins. In this cycle configurations are randomly sampled. This sampling is biased in such a way that the goal configuration has a higher probability of being sampled. Then the closest configuration (vertex) of the tree, relative to the random configuration, is selected. If the random configuration is further away than a determined distance ϵ to the nearest configuration, a new configuration is created between the nearest and the random configurations, at a distance ϵ from the nearest vertex. Otherwise, if the random configuration is at a distance smaller than or equal to ϵ to the nearest vertex, the random configuration becomes the new vertex. If it is possible this new vertex is added to the tree, and also an edge connecting the new vertex to the previously determined nearest vertex. The process continues until the goal configuration is added to the tree. The pseudo-code for this process is shown in Algorithm 1.

```
tree = []
tree.addVertex(Qstart)
while Qgoal not in tree do
    Qrand = randomizeConfiguration()
    Qnear = nearest(tree, Qrand)
    Qnew = expand(Qnear, Qrand)
    if possibleExpansion(Qnear, Qnew, Obstacles) then
        tree.addVertex(Qnew)
        tree.addEdge(Qnew)
    end
end
return tree
```

Algorithm 1: Pseudo-code for the basic RRT algorithm

Figures 3.1-3.6 illustrate the steps taken by the RRT algorithm.

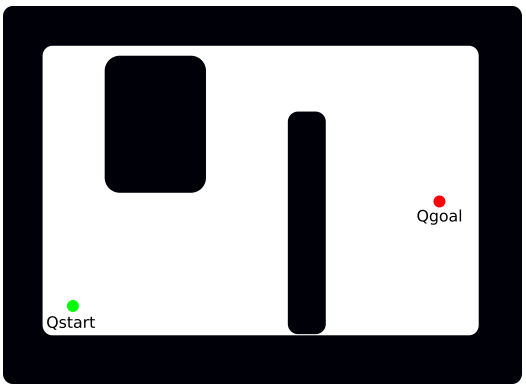


Figure 3.1: Initially only the start configuration is added to the tree

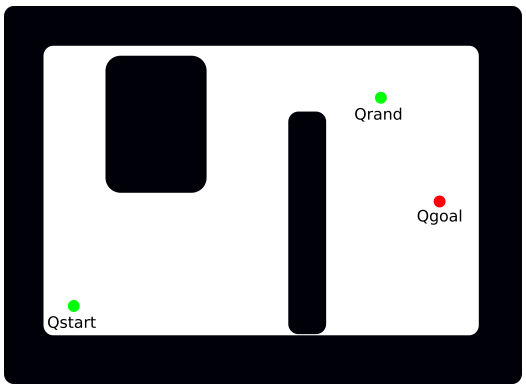


Figure 3.2: A random configuration is sampled.

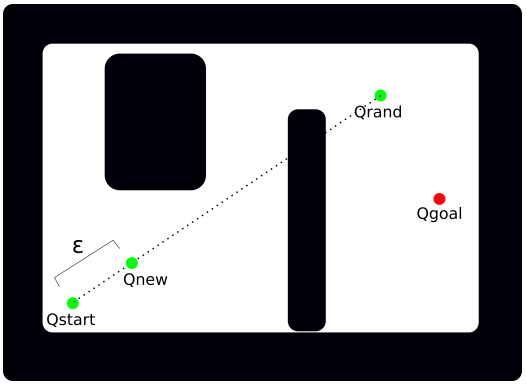


Figure 3.3: A new configuration is created that expands the tree towards the random configuration.

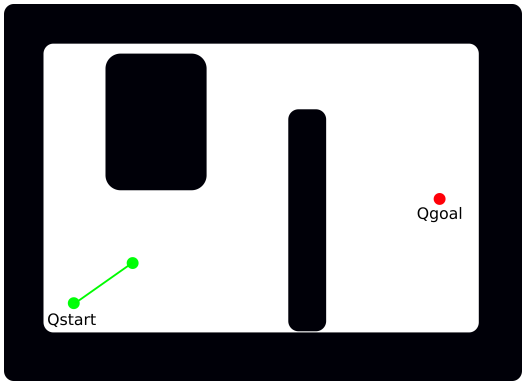


Figure 3.4: The new configuration and a new edge are added to the tree.

3.2 Proposed algorithm

Kinodynamic RRT* has a high computational cost that prevents it from being used in real time applications [26], as it was discussed in Section 2.4.1. Therefor, an alternative is proposed. An RRT which computes paths with a maximum curvature limit is presented. Unlike what is done in some literature, which consists in using the systems' dynamics for limiting the curvature of the paths [37], the problem is

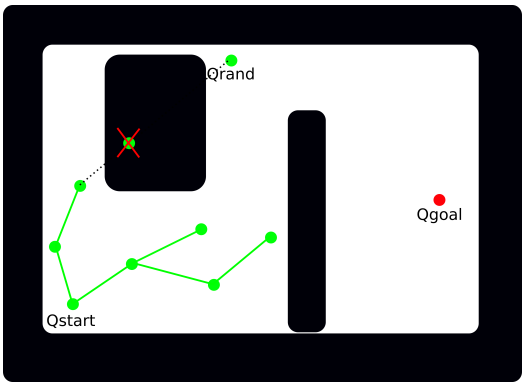


Figure 3.5: In some cases the expansion step fails, in those cases the new configuration is rejected

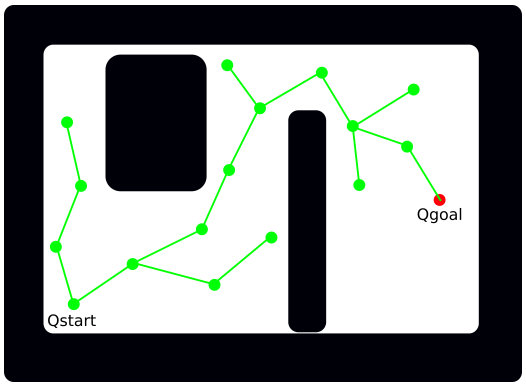


Figure 3.6: The process continues until the goal configuration is added to the tree.

approached directly. In the present work, geometrical constraints are applied as the tree is grown. In [38] the maximum curvature problem is also approached directly, however the paths generated between two vertices are arcs of circumference. In the proposed method, on the other hand, this paths are composed by a segment with maximum curvature and another with a straight line, generating paths with smaller length.

In this work the term vertexes describe, as usual in other works about RRTs, the vertices of the tree. The term edge corresponds to a line segment that connects two vertices. The edges are straight lines.

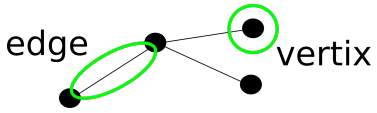


Figure 3.7: Vertices and edges

However, in this work, unlike what is done in most of the literature, the sequence of way-points that describe the trajectory is not defined by the vertices of the tree. After the goal position is added to the tree, the trajectory is computed by using, as way-points, the middle points between consecutive vertices. The speed associated to each waypoint has the direction of the line that joins consecutive vertices in the tree, as shown in Figure 3.8.

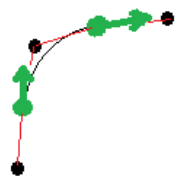


Figure 3.8: The green points and arrows represent the robot's position and speed respectively. The black points represent the vertices of the RRT.

3.2.1 Geometrical considerations

Acceptable angle between edges

It will now be defined the maximum allowed angle between consecutive edges as a function of the robot minimum curvature radius. Let d be the edge length, R_{min} be the robot minimum curvature radius and α be the angle (maximum angle) between consecutive edges. The angle α will be given in rad as:

$$\alpha = \pi - 2arctan\left(\frac{R_{min}}{d/2}\right) \tag{3.1}$$

It is obvious that the angle α can only take values between 0 and π, in this range the function $\cos(\alpha)$ is always decreasing, therefore limiting a maximum angle between two edges is the same as limiting a minimum $\cos(\alpha)$. Let now $\mathbf{e}_1$ represent one edge (position of vertex k minus position of vertex k-1) and $\mathbf{e}_2$ represent its consecutive edge (position of vertex k+1 minus position of vertex k). We can state that an edge $\mathbf{e}_2$ is acceptable after an edge $\mathbf{e}_1$ if and only if:

$$\frac{\mathbf{e}_1 \cdot \mathbf{e}_2}{\|\mathbf{e}_1\|\|\mathbf{e}_2\|} \geq \cos(\alpha_{MAX}) \tag{3.2}$$

Maximum curvature step

Sometimes a new vertex is created in such a position that it is not possible to create an edge from the previous vertex that is aligned with the new vertex (when the condition in equation 3.2 is not respected), such as in Figure 3.9.

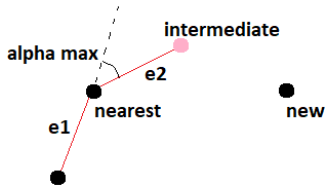


Figure 3.9: The new vertex is in such a position that it is not possible to expand the tree directly towards it, the tree is expanded than using a maximum curvature step

It will now be described how the tree is expanded in such a scenario. The tree is expanded here towards an intermediate vertex using a maximum curvature step (maximum angle between consecutive edges). The position of this intermediate vertex is calculated as follows: The vector $\mathbf{u}$ is computed, it is the projection of the position of the new vertex relatively to the nearest vertex on the direction of the edge $\mathbf{e}_1$.

$$\mathbf{u} = ((\mathbf{q}_{new} - \mathbf{q}_{near}) \cdot \mathbf{e}_1)\frac{\mathbf{e}_1}{\|\mathbf{e}_1\|^2} \tag{3.3}$$

The vector $\mathbf{v}$ is also computed, it is the projection of the position of the new vertex relatively to the nearest vertex on the hyperplane normal to $\mathbf{e}_1$ that contains the nearest vertex. This computation is

simply given by:

$$\mathbf{v} = (\mathbf{q_{new}} - \mathbf{q_{near}}) - \mathbf{u} \tag{3.4}$$

An example of the vectors $\mathbf{u}$ and $\mathbf{v}$ is represented in Figure 3.10.

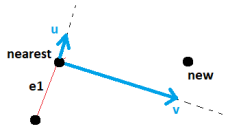


Figure 3.10: Representation of the vectors $\mathbf{u}$ and $\mathbf{v}$ in blue

The maximum curvature step can now be defined, let $\mathbf{e}_2$ denote this step (such as in figure 3.9) and let d denote the length of the edges:

$$\mathbf{e}_2 = d\left(\cos(\alpha_{MAX})\frac{\mathbf{e}_1}{\|\mathbf{e}_1\|} + \sin(\alpha_{MAX})\frac{\mathbf{v}}{\|\mathbf{v}\|}\right) \tag{3.5}$$

Unreachable regions

Finally there is also the need to define reachable regions. In this method of making maximum curvature turns until the robot is aligned with the new vertex there are some unreachable regions from each vertex, figure 3.11 is shown for an easier understanding of this concept:

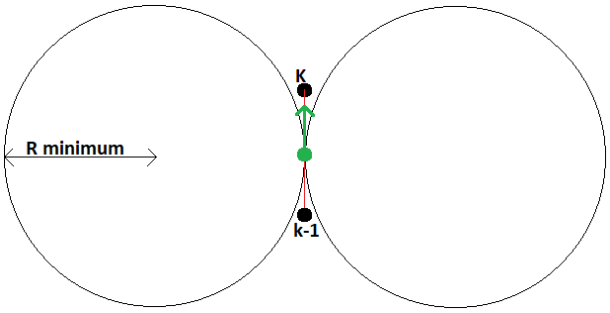


Figure 3.11: Circles describing the unreachable regions for a robot (in green)

This region is the reunion of two circles in the 2D plane and a horn torus in the 3D space. The region for the acceptable new vertices from the vertex k (in figure 3.11) is however greater, like it is shown in figure 3.12:

The unreachable region becomes now a spindle torus in three-dimensional space. The radius R_2 is given by $\sqrt{R_{min}^2 + \left(\frac{d}{2}\right)^2}$ .We can easily define a condition that dictates if a new vertex is or not acceptable. Taking into account the figure 3.10 and recalling the concepts of the $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{e}_1$ vectors defined before. A new vertex is reachable from a near vertex if and only if:

$$\left(\mathbf{u}\frac{\mathbf{e}_1}{\|\mathbf{e}_1\|} + \frac{d}{2}\right)^2 + (\|\mathbf{v}\| - R_{min})^2 \geq R_2^2 \tag{3.6}$$

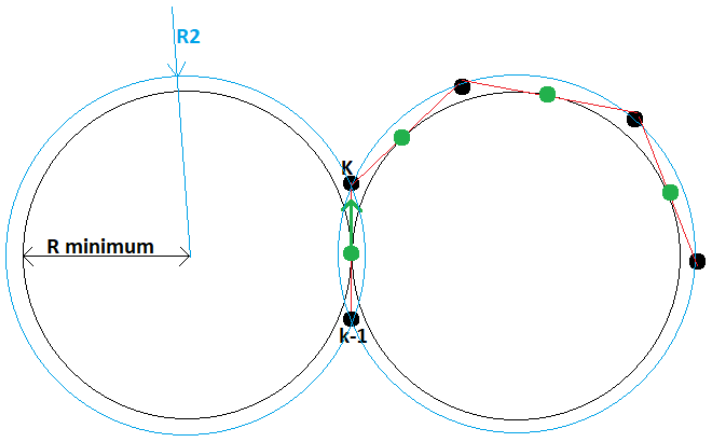


Figure 3.12: Reachable regions in from a certain vertex (bounded by circles in blue), some limit possible robot positions are represented as green circles

Where $\mathbf{u}$ and $\mathbf{v}$ are the vectors previously defined in Equations 3.3 and 3.4 respectively, and d is the length of the edges in the tree.

3.2.2 Integration in the classic RRT algorithm

The basic RRT, shown before in Algorithm 1, was changed in such a way that for a given $Qrand$ the tree is repeatedly expanded towards that $Qrand$ configuration until that same configuration is reached or until the expansion fails. The algorithm pseudo-code is presented in Algorithm 2:

```
tree = []
tree.addVertex(Qstart)
while Qgoal not in tree do
    Qrand = randomizeConfiguration()
    expand(tree, Qrand, Obstacles)
end
return tree
```

Algorithm 2: Modified RRT algorithm

The modifications proposed to the algorithm are now described:

New Expand function

The greatest modifications are in this function, the pseudo-code is presented in Algorithm 3:

The first while cycle is now explained (the second is trivial). In the beginning of the method $Qnear$ is set as the nearest point in the tree to the vertex Q. $e1$ is the edge that leads to $Qnear$. d is the length of an edge. Inside the cycle if the distance between Q and $Qnear$ is smaller than d the expansion fails. If it is possible to make an expansion directly towards Q from $Qnear$ (line 14) than $aligned = True$ the step is simply a vector with direction $Q - Qnear$ and euclidean norm d. If the it is not possible to to make a direct expansion than the step is a maximum curvature one (explained further). It is than tried to add a new point $Qinter = Qnear + step$ to the tree, if it is possible (considering the obstacles in the environment) to connect $Qnear$ to $Qinter$ (line 20) than $Qinter$ is added to the tree, $Qinter$ parent is set

```
Function expand(tree, Q, Obstacles
    Qnear = getNearest(tree, Q)
    e1 = Qnear-Qnear.getParent()
    totalExpansions = 0
    aligned = False
    if isReachable(e1, Qnear, Q) == False then
        return False
    end
    //while expansions are not directly towards Q
    while True do
        if —Q-Qnear —¡d then
            return False
        end
        if isAlignable(e1, Qnear, Q) == True then
            aligned = True
            step = straightForwardStep(Qnear, Q)
        else
            step = maximumCurvatureStep(e1, Qnear, Q)
        end
        Qinter = Qnear + step
        if tryConnect(Qnear, Qinter) == True then
            totalExpansions = totalExpansions+1
            tree.add(Qinter)
            Qinter.parent = Qnear
            e1 = Qinter-Qnear
            Qnear = Qinter
        else
            pruneTree(tree,totalExpansions)
            return False
        end
        if aligned == True then
            break
        end
    end
    //once the expansions are towards Q
    //the tree is successively expanded until
    //Q is reached or the expansion fails
    while Q != Qnear do
        step = straightForwardStep(e1, Qnear, Q)
        Qinter = Qnear + step
        if tryConnect(Qnear, Qinter) == True then
            totalExpansions = totalExpansions+1
            tree.add(Qinter)
            Qinter.parent = Qnear
            e1 = Qinter-Qnear
            Qnear = Qinter
        else
            pruneTree(tree,totalExpansions)
            return False
        end
    end
end
```

Algorithm 3: New expand function

to be $Qnear$, $e1$ is now the edge that leads to Qinter and $Qnear$ becomes $Qinter$, allowing this process to be repeated until it is possible to expand the tree directly towards Q (until aligned == true).

isReachable(e1, Qnear, Q)

Verifies if the condition described in equation 3.6 is satisfied.

isAligned(e1, Qnear, Q

Verifies if the condition in equation 3.2 is satisfied.

straightFowardStep(Qnear, Q)

Simply returns $\mathbf{e}_2 = d\frac{\mathbf{Q}-\mathbf{Qnear}}{\|\mathbf{Q}-\mathbf{Qnear}\|}$

maximumCurvatureStep(e1, Qnear, Q)

Simply returns the vector $\mathbf{e}_2$ as computed in equation 3.5

pruneTree(tree, totalExpansions)

The added curvature constraint arises new challenges for the RRT. An example is now shown in figure 3.13:

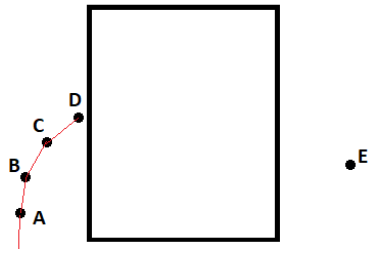


Figure 3.13: Portion of an environment with an obstacle

In this situation, an algorithm without the prune function tries to expand the tree towards vertex E. The nearest point of the tree was vertex A. The vertices B, C and D were added to the tree in the expansion, the expansion failed trying to add a vertex after D. This is a very frequent (and unpleasant) scenario if the prune function is not used. Any expansion towards a vertex Q that has D as the nearest vertex will fail. These failures will occur quite often: once D is a leaf vertex is will have a very large Voronoi region.

Therefore, every time an expansion is not successful because of the presence of an obstacle the tree is pruned. The pruning consists of deleting half (rounded up) of the vertices added to the tree while trying to reach the vertex Q. In this case, vertices C and D would be deleted.

Goal Obstacle

Without this feature it could be observed that the presented algorithm would often fail. The failures were due to the fact that the goal vertex was often in an unreachable region of the closest vertex in the tree. This was quite common, if the tree was successively expanded towards a random vertex behind the

goal region then there would be a series of aligned vertices that would cross near the goal. Afterwards, when trying to expand towards the goal one of those vertices would be selected, as it is known the closest point in a line segment r to a point X is the intersection of the perpendicular hyperplane to r that contains X. Consequently, the goal would often fall into the unreachable region, as illustrated in figure 3.14.

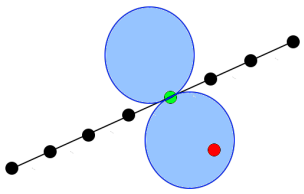


Figure 3.14: Goal position falls into an unreachable region. Goal position represented as red, vertexes on the tree as black, vertex nearest to goal as green and the corresponding unreachable region as blue.

The adopted solution to this problem is now presented. Whenever the tree is being expanded towards a random vertex a **goal Obstacle** is activated. This consists in a sphere with a radius of 2 times the minimum curvature radius. This prevents the algorithm from adding vertices to the tree that might place the goal position in an unreachable region.

3.3 Enhancement step

As it is known, trajectories computed by the RRTs might be very sub-optimal. Optimal RRT related algorithms require often great computational times like it was mentioned before. An enhancement step is now proposed. This enhancement step allows the solution computed by the smooth RRT to be enhanced in a very significant way without requiring much computational time. The method that allows this will now be studied.

3.3.1 Algorithm

The step consists in trying, from each vertex in the RRT, to reach directly another vertex in the trajectory as close to the goal as possible. A scheme is now presented in order to demonstrate this concept in Figure 3.15:

First it is desirable to define two terms. *JumpBaseVertex* is the vertex from which the direct expansions will be attempted (outlined by a blue circle in figure 3.15). *FurthestReachableVertex* is the furthest vertex in the trajectory to which a direct expansion was possible (outlined by a red circle in figure 3.15).

In this algorithm an expansion directly from the *JumpBaseVertex* is attempted for the vertex after the previous *FurthestReachableVertex*. If the expansion is possible than the trajectory with both these vertices connected is stored and the *FurthestReachableVertex* is updated. If the expansion fails than it is tried to expand towards the remaining vertices on the trajectory, if one of them succeeds than *FurthestReachableVertex* is updated. After every vertex is tested the *JumpBaseVertex* is changed to the next vertex on the trajectory. Sometimes the *JumpBaseVertex* reaches the *FurthestReachableVertex*

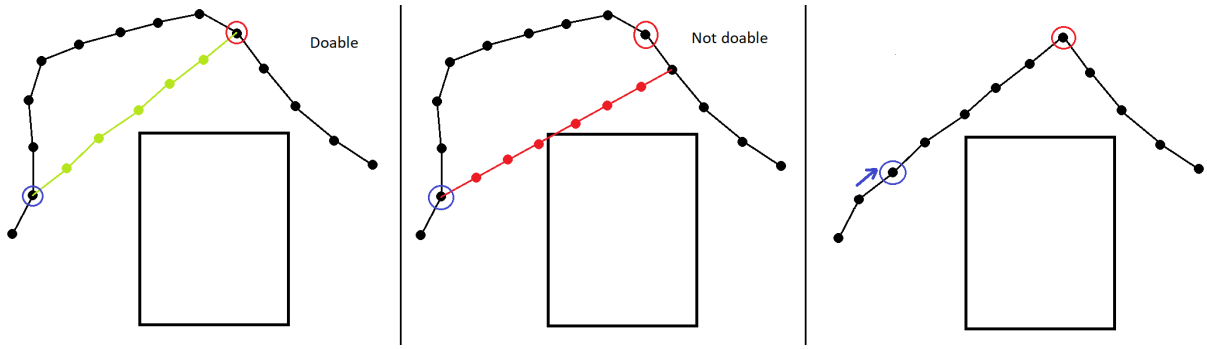


Figure 3.15: Consecutive steps in the enhancement algorithm. The blue circle identifies the *JumpBaseVertex* and the red circle identifies the *FurthestReachableVertex*. A green path represents a doable connection and a red path an impossible connection.

once expansions to vertexes closer than an expansion step fail in this algorithm. To prevent this every time *JumpBaseVertex* reaches the *FurthestReachableVertex* the *FurthestReachableVertex* is set as the next vertex. The process continues until the goal is stored as the *FurthestReachableVertex*. The pseudo-code for this is presented in Algorithmn 4.

```
JumpBase=start
FurthestReachable=start.next()
tryingToReach=FurthestReachable.next()
while FurthestReachable != goal do
    if expand(JumpBase, tryingToReach, Obstacles ) then
        FurthestReachable=tryingToReach
        tryingToReach=tryingToReach.next()
    else
        if tryingToReach!=goal then
            tryingToReach=tryingToReach.next()
        else
            JumpBase=JumpBase.next()
            if(JumpBase==FurthestReachable)
            FurthestReachable=FurthestReachable.next()
            tryingToReach=FurthestReachable.next()
        end
    end
end
```

Algorithm 4: Algorithm of the enhancement step.

3.3.2 Changing environments

This algorithm was also empowered with the capability to deal with moving obstacles. For this purpose it is necessary to add information about the time correspondent to each vertex of the tree, to allow checking whether each vertex is in collision with an obstacle at the corresponding moment in time.

Therefore, the starting time has to be propagated through the tree. To implement such, the time for each new vertex of the tree was computed at the moment it was added to the tree. The time corresponding to the newly added vertex is simply given by the time of its parent vertex plus the distance to the parent vertex divided by the UAV speed. In this work the term *parent vertex* is used to refer to the

existing vertex of the tree to which the new vertex is connected.

$$t_{new} = t_{parent} + \frac{distance_{parent-new}}{speed}$$

3.4 Results

The constrained curvature RRT was grown in 2 different environments, the start vertex is the red circle on the left and the goal is the red circle on the right (Fig. 3.16 - 3.17).

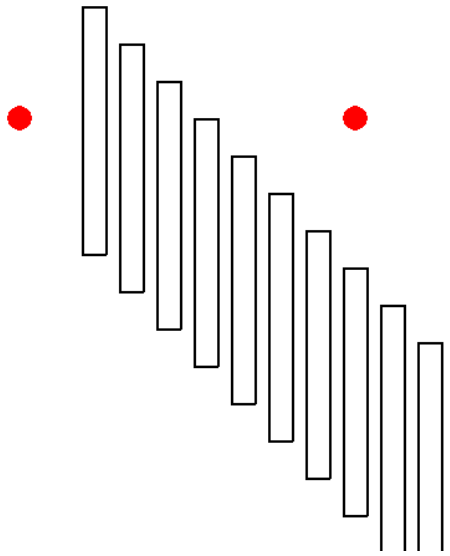


Figure 3.16: Environment 1

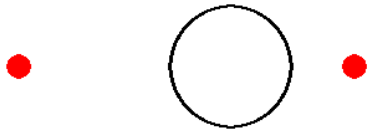


Figure 3.17: Environment 2

The algorithms for enhancing the trajectory were performed after growing the RRT 100 times in each environment. An **unrealistic cost** is defined as the minimum distance of a trajectory between start and goal if there were no curvature limitations, the pseudo-sub-optimality measure is then given by $\frac{cost-unrealisticcost}{unrealisticcost} \cdot 100\%$. The results are now presented:

Environment 1	time (s)	total distance	pseudo-sub-optimality
First trajectory	0.028	474	38.6%
Trajectory enhancement	0.026	376	9.64%
Final	0.054	376	9.64%

Table 3.1: Results of the application of the RRT algorithm and enhancement step to environment 1

Environment 2	time (s)	total distance	pseudo-sub-optimality
First trajectory	0.004	435	50%
Trajectory enhancement	0.013	308	6.21%
Final	0.017	308	6.21%

Table 3.2: Results of the application of the RRT algorithm and enhancement step to environment 2

In the first environment there is a narrow passage above the first obstacle which is hard for the algorithm (due to the constrained curvature) to overcome. For that reason, the algorithm takes more

time to compute a path than in the second environment. The enhancement step takes approximately as long as the initial trajectory computation for the environment 1 and almost 3 times more for the second. The enhanced trajectories have a length close to the unrealistic cost, and even closer to the locally-optimal cost for a curvature constrained path. Figures 3.18-3.21 show some examples of the computed trajectories before and after the enhancement step. This enhancement step allows to quickly improve the length of the trajectory without the need to sample new random configurations, like it is done in the RRT*.

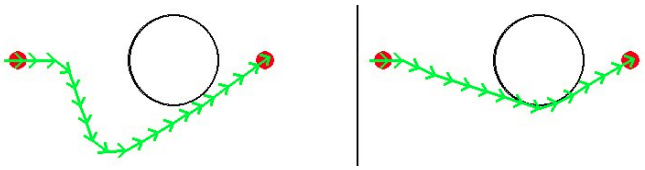


Figure 3.18: Example of the trajectory before (on the left) and after (on the right) the trajectory enhancement

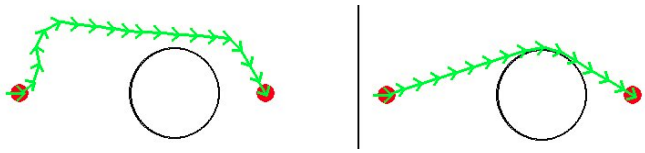


Figure 3.19: Example of the trajectory before (on the left) and after (on the right) the trajectory enhancement

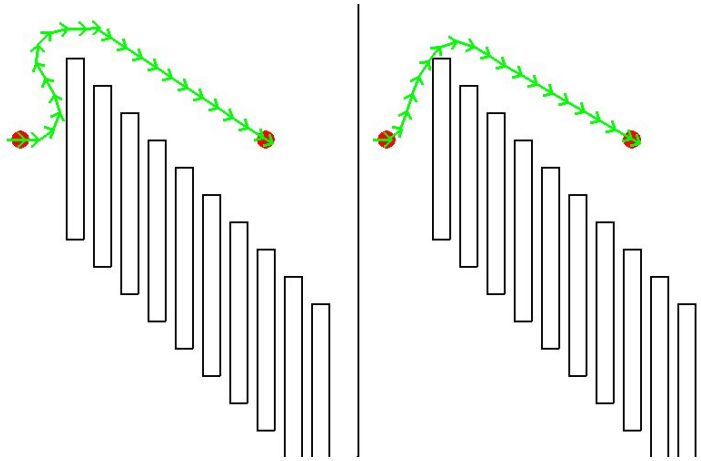


Figure 3.20: Example of the trajectory before (on the left) and after (on the right) the trajectory enhancement

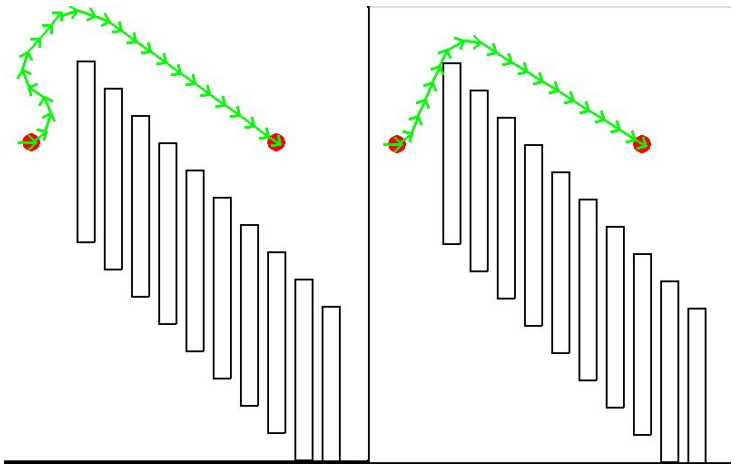


Figure 3.21: Example of the trajectory before (on the left) and after (on the right) the trajectory enhancement

Chapter 4

Trajectory Optimization

Optimization problems consist in minimizing a cost function $f(x)$ while respecting a series of constraints, as it has been described in section 2.3. These constraints can be equities $c(x) = 0$ or inequities $d(x) \geq 0$. In 2.3 the solver used is described, and it is a dual-barrier method. The problem will be formalized in the classical form:

$$\begin{aligned} \textbf{minimize:} \quad & f(x) \\ \textbf{subjected to:} \quad & c(x) = 0 \\ & d(x) \geq 0 \end{aligned}$$

There are multiple ways of solving this sort of problems, however, an exhaustive review on these methods will not be made in this work. In general optimization methods require an initial value for the design variable vector x_0. It will now be addressed how to formalize the trajectory-optimization problem as a conventional optimization problem. The approximations and assumptions made for this problem are:

- The multi-rotor is approximated to a point
- The attitude is, at this point, ignored
- It can move in any direction
- It has limited acceleration
- It has limited speed
- The multi-rotor state σ is defined by the position and speed of its center of mass.

The planning will be performed in a three-dimensional environment. To make use of the differential-flatness property of the multi-rotors it would be necessary to assure that the control inputs of the vehicle were not saturated, by evaluating the derivatives, up to the fourth derivative of position [31]. In the present work, however, for the sake of simplicity and computational performance, the computed trajectories are only differentiable up to the second derivative of the position. Therefore, the limitation on the speed and acceleration are used as heuristics to avoid saturating the control inputs of the multi-rotor.

4.1 Nomenclature

It will be now introduced some nomenclature that will be used in this section. Part of this nomenclature is already defined in Chapter 2. $[\sigma_0, ..., \sigma_{N-1}]$ is the vector with all the N robot states along the trajectory, excluding the start and the goal states. σ_s and σ_g are the start and goal states respectively and these will not be subjected to optimization. Also $\sigma_i = [\mathbf{p}_i, \mathbf{v}_i]$, where $\mathbf{p}_i$ and $\mathbf{v}_i$ corresponds to the position and speed vector of the robot in a given moment. This means that $\sigma_i \in \mathbb{R}^4$ for bi-dimensional environment and $\sigma_i \in \mathbb{R}^6$ for three-dimensional environment (e.g. UAV). Let also $p_{i,j}$ represent the j^{th} component of the robot position in the i^{th} state and $v_{i,j}$ represent the j component of the robot speed in the i^{th} state. For example, $p_{i,z}$ is the robot position along the z axis in the i^{th} state and $v_{i,y}$ represents the robot speed along the y axis in the i^{th} state. Basically: $\sigma_i = [\mathbf{p}_i, \mathbf{v}_i] = [p_{i,x}, p_{i,y}, p_{i,z}, v_{i,x}, v_{i,y}, v_{i,z}]$ (in 3 dimensional space).

It will be assumed that the robot acceleration $\mathbf{a}$ between two consecutive states is constant. Each state σ_i represents the state of the robot at the time $t = t_s + (i+1)\Delta t$ where t_s is the start time. Therefore the total trajectory time is given simply by $t_{total} = (N+1)\Delta t$ (one time step from start to σ_0, N-1 time steps between the N states and one time step between σ_{N-1} and the goal).

Considering this, the variables subjected to optimization will be $x = [\Delta t, \sigma_0, ..., \sigma_{N-1}]$. If it is written in the form of a vector of scalars (in 3 dimensional space):

$$x = [\Delta t, p_{0,x}, p_{0,y}, p_{0,z}, v_{0,x}, v_{0,y}, v_{0,z}, ..., p_{N-1,x}, p_{N-1,y}, p_{N-1,z}, v_{N-1,x}, v_{N-1,y}, v_{N-1,z}]$$

This will be the design variables of the problem, it is essential for the good comprehension of the following topics.

4.2 Problem formulation

It is possible to describe a path-planning problem as an optimization problem (trajectory optimization). This can be done by using, as design variables, the state of the system in different moments of the trajectory. It is also required to determine a cost function $f(x)$ (which will be minimized) and a series of constraints that must be respected, these constraints might be equalities or inequalities. This approach is called *direct transcription* or *direct collocation* [20].

It will now be presented the cost function, equalities and inequalities chosen for our problem. The gradients and Hessian matrices of these constraints should also be computed once optimization algorithms benefit from having analytic expressions for them, however, these calculations will not be presented. The cost function chosen is the total trajectory time. The constraints will be formulated in order to make the computed trajectory respect the kinodynamic constraints of the presented problem namely: the kinematics, maximum speed and maximum acceleration. There will also be formulated constraints to assure obstacle clearance.

4.3 Cost function

The cost function chosen is a linear combination between the trajectory time and energy consumption. The scalar weights k_T and k_F allow tuning the relevance given to each of these costs. Such cost functions are suitable, for example, for minimizing the mission-related costs. The cost function is then:

$$f(x) = k_T f_T(x) + k_F f_F(x) \tag{4.1}$$

The time component $f_T(x)$ is the total trajectory time, which is given as $(N+1)\Delta t$ where N represents the number of states subjected to optimization.

To model the energy consumption $f_F(x)$ for the multi-rotor the work by Marins et al. [39] will be analysed. In this work the authors estimate the energy consumption of a multi-rotor based on its dynamics and the physical principles of superposition, using a close form expression. The estimation errors obtained were less than 1% concerning experimental results. The energy consumption is divided into three components: related to hovering, related to acceleration/deceleration and related to the work performed by drag forces.

The power consumption related to hovering is simply given by a constant hovering power multiplied by the flight time. To include such cost in the cost function it is possible to simply use the time-related component, replacing k_T in Equation 4.1 by $k_T + k_F P_0$, where k_F is the scalar multiplied by the energy cost and P_0 is the hovering power. It is important to refer that accordingly to [39] the hovering power represents more than 90% of the power consumption in the tested small multi-rotor in a mission where the ”cruising speed” was 8 meters per second.

The power related to the acceleration/deceleration can be calculated as the variation of kinetic energy. In [39] it is assumed that the modulus of this variation is proportional to the energy consumption (decelerating also consumes energy). For the present work, it is also included the variations on gravitational potential energy, for this reason, this component of the energy consumption will be estimated as the modulus of the variation of the mechanical energy of the quad-copter. In [39] it is shown that this component of the energy consumption is the less relevant to determine the optimal speed for a straight-line path. Equation 4.2 is used to determine the energy consumption due to the variation of mechanical energy in a trajectory segment.

$$\begin{aligned} E &= \|E_M(i+1) - E_M(i)\| \\ E_M(i) &= \tfrac{1}{2} m \|\mathbf{v}_i\|^2 + mgp_{i,z} \end{aligned} \tag{4.2}$$

In this equation, E represents the consumed energy, $E_M(i)$ represents the mechanical energy at state σ_i, m represents the UAV mass and g represents the gravitational acceleration. $\mathbf{v}_i$ represents the speed of the UAV at state σ_i and $p_{i,z}$ the position of the UAV along the z axis.

Finally, the energy consumption related to the work done by drag forces is approximated in [39], for straight-line paths, as if the speed of the path was constant and equal to the ”cruise speed”, disregarding the accelerating and decelerating stages. The drag force was calculated as being a constant (vehicle dependent) multiplied by the squared speed. Aiming for a better result, and since the speed changes

linearly between waypoints, it is possible to compute the work done by the drag force between two consecutive waypoints as:

$$E = \int_{t_i}^{t_{i+1}} const \left\| v(t) \right\|^3 dt \quad \Leftrightarrow \quad E = \int_{t_i}^{t_{i+1}} const \left\| \mathbf{v}_i + (\mathbf{v}_{i+1} - \mathbf{v}_i) \frac{t - t_i}{t_{i+1} - t_i} \right\|^3 dt \tag{4.3}$$

The integral presented in Equation 4.3 has a closed-form solution. If the speeds in the consecutive waypoints were co-linear this equation would have a simple solution. However, that is not the general case. For the general case there is, indeed, a closed expression solution for the integral in Equation 4.3, however, the expression is quite complex, and so are its partial derivatives. For that reason, an assumption like the one made in [39] was made. It is assumed that the work done by the drag forces is proportional to the distance between waypoints times the average speed raised to the power of 2. In other words, it is assumed that the multi-rotor performs a straight-line trajectory at a constant speed between waypoints. The resulting expression for this component of the energy consumption in a trajectory segment is then:

$$E = \frac{1}{2} C_D \rho A_{ef} \| \mathbf{p}_{i+1} - \mathbf{p}_i \| \frac{\| \mathbf{v}_{i+1} + \mathbf{v}_i \|^2}{2} \tag{4.4}$$

In Equation 4.4, E represents the energy consumption in a trajectory segment, C_D the drag coefficient, ρ the air density and A_{ef} the UAV effective area.

The energy consumption is then given by the combination of these components, computed for every trajectory segment between waypoints (including start and goal).

4.4 Kinematics

The algorithm must respect the kinematics of the problem. It is assumed that the acceleration between to consecutive states σ_i and σ_{i+1} is constant. It is also assumed that the time elapsed for the robot to move from one state to another is Δt. With these assumptions, the position $\mathbf{p}_{i+1}$ should be given by:

$$\mathbf{p}_{i+1} = \mathbf{p}_i + \frac{\mathbf{v}_i + \mathbf{v}_{i+1}}{2} \Delta t \tag{4.5}$$

Writing this as an equity constraint in the form $c_K(x) = 0$:

$$\begin{aligned} \mathbf{p}_{i+1} - \mathbf{p}_i - \frac{\mathbf{v}_{i+1} + \mathbf{v}_i}{2} \Delta t = 0 \Leftrightarrow \\ \Leftrightarrow c_K(x) = \mathbf{p}_{i+1} - \mathbf{p}_i - \frac{\mathbf{v}_{i+1} + \mathbf{v}_{i+1}}{\Delta t} \end{aligned} \tag{4.6}$$

Let $\mathbf{x_s}$ and $\mathbf{x_g}$ be the start and goal state respectively. It is now possible to write the scalar equations $c_K(i,j)(x) = 0$ that represent the kinematic constraints between state σ_i and σ_{i+1} for the j^{th} component of the position/speed. $c_K(s,j)(x) = 0$ and $c_K(g,j)(x) = 0$ represent the constraints for the segments between start and the σ_0, and between σ_{N-1} and the goal, respectively. Then:

$$c_{K(s,j)} = p_{0,j} - p_{s,j} - \frac{v_{0,j} + v_{s,j}}{2} \Delta t \tag{4.7a}$$

$$c_{K(i,j)} = p_{i+1,j} - p_{i,j} - \frac{v_{i+1,j} + v_{i,j}}{2}\Delta t \tag{4.7b}$$

$$c_{K(g,j)} = p_{g,j} - p_{N-1,j} - \frac{v_{g,j} + v_{N-1,j}}{2}\Delta t \tag{4.7c}$$

Equation 4.7b applies for $i \in \{0, \cdots, N-2\}$ and $j \in \{x, y, z\}$. Equations 4.7a and 4.7c apply for $j \in \{x, y, z\}$.

Another possible formulation for this constraint is to take the norm or the squared norm of the vectorial constraint in Equation 4.6. In this way a smaller number of constraints is required, leading to a smaller number of Lagrange multipliers. This would lead to the inversion of smaller matrices while optimizing, however, the Jacobean and Hessian of the constraint would be denser matrices. In the case that the squared norm of the vectorial constraint is taken it leads to small values in the derivatives of the constraint when the constraint is close to being satisfied, which might be a problem. It would be interesting to test these formulations in future work.

4.5 Maximum speed

It is now required to write the maximum speed constraint in the form: $d_S(x) \geq 0$. The maximum speed can be written as:

$$\|\mathbf{v}_i\| \leq v_{MAX} \tag{4.8}$$

Re-writing this constraint in the desired form we have, for each state σ_i:

$$d_{S(i)} = v_{MAX} - \|\mathbf{v}_i\| \geq 0 \qquad , i \in \{1, \cdots, N-1\} \tag{4.9}$$

This constraint can also be written in a form that is more efficient computationally while assuring derivatives different from 0 in the boundary. This new inequality is obtained by squaring both sides of the inequality in Equation 4.8 (both sides of Equation 4.8 are positive at all times).

$$d_{S(i)} = v_{MAX}^2 - \|\mathbf{v}_i\|^2 \geq 0 \qquad , i \in \{1, \cdots, N-1\} \tag{4.10}$$

In this form, the Hessian matrix for the constraint is not as dense because the cross derivatives $\frac{\partial^2 s_{S(i)}}{\partial \mathbf{v}_{i,x} \partial \mathbf{v}_{i,y}}$, $\frac{\partial^2 s_{S(i)}}{\partial \mathbf{v}_{i,x} \partial \mathbf{v}_{i,z}}$ and $\frac{\partial^2 s_{S(i)}}{\partial \mathbf{v}_{i,y} \partial \mathbf{v}_{i,z}}$ become zero.

4.6 Maximum acceleration

Once again, it is convenient to state that we assume that the UAV has a constant acceleration between two consecutive states, this acceleration is given by $\mathbf{a_i} = \frac{\mathbf{v}_{i+1} - \mathbf{v}_i}{\Delta t}$. It is intuitive that the maximum acceleration constraint can be written as:

$$\|\mathbf{a_i}\| \leq a_{MAX} \Leftrightarrow \frac{\|\mathbf{v}_{i+1} - \mathbf{v}_i\|}{\Delta t} \leq a_{MAX}$$

To keep the derivatives simple, the form chosen for this inequity to be written was:

$$\|\mathbf{v}_{i+1} - \mathbf{v}_i\| \leq a_{MAX}\Delta t \tag{4.11}$$

The constraint should have the form $d_A(x) \geq 0$,

$$d_A(x) = a_{MAX}\Delta t - \|\mathbf{v}_{i+1} - \mathbf{v}_i\| \geq 0 \tag{4.12}$$

This constraint is now written in the form of a series of scalar constraints:

$$d_{A(s)} = a_{MAX}\Delta t - \|\mathbf{v}_0 - \mathbf{v}_s\| \tag{4.13a}$$

$$d_{A(i)} = a_{MAX}\Delta t - \|\mathbf{v}_{i+1} - \mathbf{v}_i\| \tag{4.13b}$$

$$d_{A(g)} = a_{MAX}\Delta t - \|\mathbf{v}_g + \mathbf{v}_{N-1}\| \tag{4.13c}$$

Equation 4.13b applies for $i \in \{0, \cdots, N-2\}$.

These equations can also be formulated in a different, more efficient manner. This can be done by squaring both sides of the inequality stated in equation 4.11. The resulting scalar constraints are:

$$d_{A(s)} = a_{MAX}^2\Delta t^2 - \|\mathbf{v}_0 - \mathbf{v}_s\|^2 \tag{4.14a}$$

$$d_{A(i)} = a_{MAX}^2\Delta t^2 - \|\mathbf{v}_{i+1} - \mathbf{v}_i\|^2 \tag{4.14b}$$

$$d_{A(g)} = a_{MAX}^2\Delta t^2 - \|\mathbf{v}_g + \mathbf{v}_{N-1}\|^2 \tag{4.14c}$$

This formulation leads to a less dense Hessian matrix of the constraints.

4.7 Obstacle clearance

It is now required to define the signed distance of a point to a convex obstacle. Let this signed distance $s(\mathcal{O}_k, \mathbf{p}_i)$ represent the distance from a point $\mathbf{p}_i$ to the closest point on the surface of a convex obstacle $\mathcal{O}_k$, *this distance is negative if the point* $\mathbf{p}_i$ *is inside the obstacle* $\mathcal{O}_k$ *and 0 if the point* $\mathbf{p}_i$ *lies on the obstacle* $\mathcal{O}_k$ *boundary*. The closest point on the surface of a convex obstacle $\mathcal{O}_k$ to a point $\mathbf{p}_i$ will be called $\mathbf{o}_{(k,i)}$. It can now be given analytic expressions for the signed distance:

$$s(\mathcal{O}_k, \mathbf{p}_i) = \begin{cases} -\|\mathbf{p}_i - \mathbf{o}_{(k,i)}\| & \text{, if } \mathbf{p}_i \text{ is inside } \mathcal{O}_k \\ \|\mathbf{p}_i - \mathbf{o}_{(k,i)}\| & \text{, otherwise} \end{cases} \tag{4.15}$$

The inequity that assures that the UAV is at least d_{safe} away from any obstacle is $s(\mathcal{O}_k, \mathbf{p}_i) \geq d_{safe}$. Writing the constraint in the desired form, let K represent the number of obstacles, then:

$$d_{O(i,k)} = s(\mathcal{O}_k, \mathbf{p}_i) - d_{safe} \geq 0 \tag{4.16}$$

Equation 4.16 applies for $i \in \{1, \cdots, N-1\}$ and $k \in \{0, \cdots, K-1\}$. This formulation for the obstacle clearance constraints is formulated based on [28]. The signed distance is, however, at the moment of writing, only available for spheres and cuboids in 3-dimensional environments and circles and rectangles in 2-dimensional environments. For enabling the computation of the derivatives and Hessian of the signed distance to a cuboid it was necessary to treat 3 different situations: when the closest point in the cuboid lies on a face, an edge or a vertex. The algorithm also supports moving spheres (and circles) to enable the algorithms to avoid other aircraft and moving obstacles. It can be generalized for convex obstacles using, as in [28], the Gilbert Johnson Keerthi algorithm [40] combined with the Expanding Polytope Algorithm [41].

In this formulation, the collision is only checked in the waypoints, which limits the spacing between them. To enable the usage of bigger trajectory segments, which reduce the number of waypoints and increases the algorithm performance, new constraints were imposed. These new constraints assure that a defined number of equally spaced intermediate points between waypoints is not close to any obstacle. It will be used $q = \mathcal{P}(\sigma_i, \sigma_{i+1}, m, M)$ to describe the intermediate point corresponding to the point placed in $t = t_i + \Delta t m/M$ of the trajectory segment between σ_i and σ_{i+1}. $M-1$ represents the number of intermediate points per segment, and $m \in \{1, ..., M-1\}$. Figure 4.1 shows the distribution of these intermediate points between two waypoints when M=4.

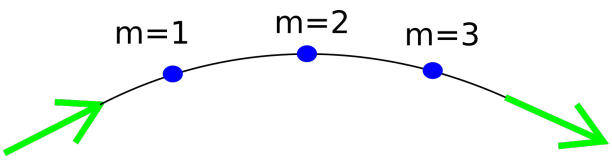


Figure 4.1: Intermediate points (blue) in trajectory segment (black) between consecutive waypoints (green arrows) for $M = 4$.

The new set of constraints is:

$$d_{OI(s,k,m)} = s(\mathcal{O}_k, \mathcal{P}(\sigma_s, \sigma_0, m, M)) - d_{safe} \tag{4.17a}$$

$$d_{OI(i,k,m)} = s(\mathcal{O}_k, \mathcal{P}(\sigma_i, \sigma_{i+1}, m, M)) - d_{safe}, \quad i \in \{0, ..., N-2\} \tag{4.17b}$$

$$d_{OI(g,k,m)} = s(\mathcal{O}_k, \mathcal{P}(\sigma_{N-1}, \sigma_g, m, M)) - d_{safe} \tag{4.17c}$$

The Equations 4.17 apply for $m \in \{1, ..., M\}$ and for $k \in \{0, ..., K-1\}$. Note that both Equations 4.17 and 4.16 form a total of $(N+1)MK$ constraints. This number is given by the number of trajectory segments times the number of checking points per segment (M-1 intermediate points plus waypoint =

M) times the number of obstacles. Given that most of these constraints are relaxed at most times, this formulation might not be the best. Using, instead of constraints, cost fields for imposing the obstacle clearance, as it is done in [28], might improve the computation time and should be tested in the future.

4.7.1 Moving obstacles

The algorithm was also expanded to deal with moving spheres and circles. For these moving spheres the signed distance can be computed using Equation 4.18:

$$d = \|\mathbf{p}_i - \mathbf{c}_o(t)\| - r(t) \tag{4.18}$$

where $\mathbf{c}_o(t)$ represents the position of the center of the sphere along time and $r(t)$ the radius of the sphere, both dependent on the time t. The time corresponding to the i^{th} waypoint is given by $t = t_{start} + (i+1)\Delta t$. In the implementation level, to allow to incorporate moving obstacles, it was also required to take the derivatives of the signed distance with respect to Δt.

There are considered two types of moving obstacles: the ones with known trajectories and the ones with unknown trajectories. For intruders with known trajectories, the safety distance that the algorithm keeps to these intruders is the same as the one kept for static obstacles. For intruders with unknown trajectories, the algorithm assumes that these intruders will fly at a constant speed and creates trajectories that keep a safety distance k_d times greater than the one kept for static obstacles and intruders with known trajectories. This increased safety distance is used to cope with the unpredictability of intruders with unknown trajectories.

4.7.2 Variable bounds

IPOPT allows to bound the design variables with both upper and lower bounds. For this problem, however, only one variable is bounded: the time step. The time step is imposed to be:

$$\Delta t \leq 2d_{safe}/v_{MAX} \tag{4.19}$$

This constraint assures that the trajectory does not ”jump” over obstacles like it is presented in Figure 4.2. If Equation 4.19 is not respected than the spacing between two consecutive waypoints is enough to have one waypoint in each side of an obstacle without any of them being too close to it.

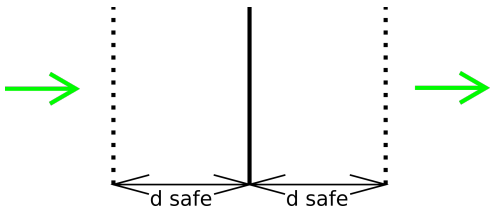


Figure 4.2: Trajectory segment between two waypoints (green arrows) crosses an obstacle (black line) without any of the waypoints being closer than d_{safe} to the obstacle.

However, if the obstacle clearance is also assured in intermediate points, as described in Section 4.7, Equation 4.19 becomes:

$$\Delta t \leq 2Md_{safe}/v_{MAX} \tag{4.20}$$

Where $M-1$ is the number of intermediate points for each trajectory segment. That being said, it is possible to increase the spacing between waypoints by introducing intermediate collision checking points.

4.8 Tested formulations

The maximum speed and maximum acceleration constraints are written in [24] as a series of linear constraints. Although the problem is different (in [24] the problem is bi-dimensional), this and other formulations of the constraints could be tested in the future.

Another interesting formulation for constraints would be to merge the maximum speed and maximum acceleration constraints by defining a "maximum thrust" constraint and write the thrust accordingly to the drag and inertial forces actuating in the UAV, this possibility, however, was not explored.

Two problems were implemented. In the first the energy costs are not considered, the speed constraints are formulated accordingly to Equation 4.9, the maximum acceleration constraints are formulated accordingly to 4.13 and the obstacle clearance is only checked at the waypoints (Equation 4.16). A simplified intuitive statement of this problem, which will be called *Problem 1*, is presented in Equation 4.21:

$$\begin{array}{lll} min_x & (N-1)\Delta t & \text{(trajectory time)} \\ s.t. & \mathbf{p}_{i+1} = \mathbf{p}_i + \frac{\mathbf{v}_{i+1}+\mathbf{v}_i}{2}\Delta t & \text{(kinematic constraints)} \\ & \|\mathbf{v}_i\| \leq v_{MAX} & \text{(maximum speed)} \\ & \|\mathbf{v}_{i+1} - \mathbf{v}_i\| \leq a_{MAX}\Delta t & \text{(maximum acceleration)} \\ & s(\mathcal{O}_k, \mathbf{p}_i) \geq d_{safe} & \text{(obstacle clearance)} \end{array} \tag{4.21}$$

where the design variables x are the position and speed at each waypoint, $\mathbf{p}_i$ and $\mathbf{v}_i$, and the time step between waypoints Δt. In the second problem the energy costs are considered, the speed constraints are formulated accordingly to Equation 4.10, the maximum acceleration constraints are formulated accordingly to 4.14 and the obstacle clearance is checked both at the waypoints (Equation 4.16) and at the intermediate points (Equations 4.1).*Problem 2* is shown in a simplified way in Equation 4.22.

$$
\begin{aligned}
min_x \quad & k_T f_T(x) + k_F f_F(x) && \text{(trajectory cost)} \\
s.t. \quad & \mathbf{p}_{i+1} = \mathbf{p}_i + \tfrac{\mathbf{v}_{i+1}+\mathbf{v}_{i+1}}{2}\Delta t && \text{(kinematic constraints)} \\
& \|\mathbf{v}_i\|^2 \leq v_{MAX}^2 && \text{(maximum speed, squared)} \\
& \|\mathbf{v}_{i+1} - \mathbf{v}_i\|^2 \leq a_{MAX}^2 \Delta t^2 && \text{(maximum acceleration, squared)} \\
& s(\mathcal{O}_k, \mathbf{p}_i) \geq d_{safe} && \text{(obstacle clearance, at waypoints)} \\
& s(\mathcal{O}_k, \mathcal{P}(\sigma_i, \sigma_{i+1}, m, M))) \geq d_{safe} && \text{(obstacle clearance, at intermediate points)}
\end{aligned}
\tag{4.22}
$$

The Hessian matrix was computed for the cost function and the constraints. The algorithm theoretically benefits from the usage of an analytical Hessian, however, if it is not provided one, IPOPT uses a Hessian approximation. After some testing, it becomes clear that for short optimization periods the usage of the analytical Hessian makes the algorithm slower. This might be because the algorithm computes, before it starts to optimize, an argument list for the function that returns the Hessian matrix. For this reason, in both problems, the Hessian matrix is approximated by IPOPT.

4.9 Results

4.9.1 Complete computation

This problem (cost function and constraints) was implemented in Python 2.7. The solver used for this problem was, as mentioned before, an interior point optimization approach, developed by A. Wächter and L. T. Biegler called IPOPT [21]. It was integrated on the remaining code using the wrapper *pyipopt* [1] . Another solver was also tested on this problem. This solver was used through a python wrapper, available in the *scipy* module, for an implementation of the Sequential Least SQuares Programming (SLSQP) Optimization subroutine originally implemented by Dieter Kraft [42]. Both optimizers were tested in a 3-dimensional scenario with 8 obstacles, the scenario was a Gazebo world, presented in Fig. 4.3 and 4.4. Gazebo is a physics simulator which will be described in Section 6.2.1. *Problem 1* was used in these tests, for both optimizers. The path planning problem was formulated as a 2-dimensional problem, where the UAV was constrained to flight 1.5 meters above the ground. The experiences were performed in a desktop with an AMD Ryzen 5 2600X Six-Core Processor 3.60GHz, 16GB installed RAM and an NVIDIA Quadro P2000. The operative system used was Ubuntu 18.04.

The ”RRT time” corresponds to the time for the execution of both the RRT algorithm and the enhancement step. The optimization time corresponds to the time that the optimizer takes to optimize the trajectory. The results presented in Figure 4.5 are the average times over 50 trials.

[1] pyipopt github, 2018. https://github.com/xuy/pyipopt

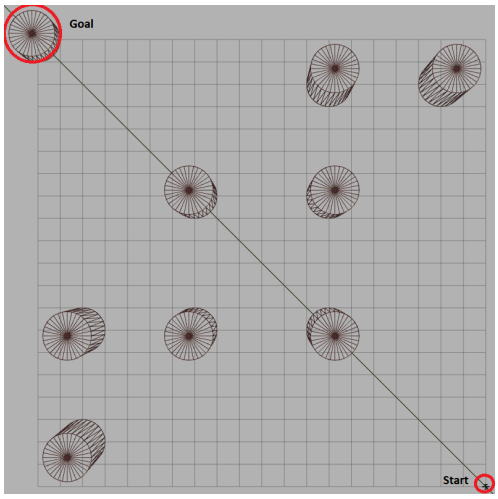


Figure 4.3: Top view of the tested map

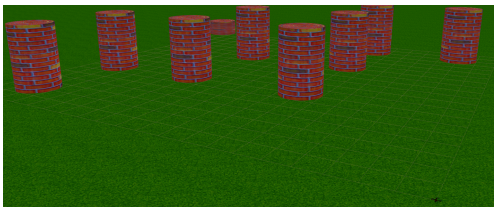


Figure 4.4: Tested map with textures

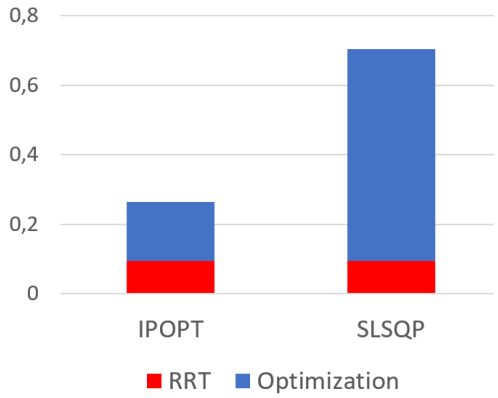


Figure 4.5: Comparison between the two optimizers, the computation time is represented in seconds.

It can be observed that the IPOPT optimizer outperforms the SLSQP. With IPOPT and the proposed RRT algorithm it is possible to obtain a locally optimal trajectory in less than 300 milliseconds. For a micro-processor, these times will increase. However, there is no need for the trajectory to be fully optimized in a single time-interval, the framework can continuously optimize the trajectory while the UAV flies.

4.9.2 Comparison between formulations

It will now be compared the performance of IPOPT generating a locally-optimal trajectory using *Problem 1* and *Problem 2* (described in Equation 4.21 and 4.22 respectively). For both problems, the same initial iteration (computed by an RRT) was given, shown in Figure 4.6. In the first iteration the safety distance to 10 and the speed to 10 s^{-1} (the distance units are defined by the pygame display). The time-step between waypoints Δt was set to 2 seconds (maximum allowed for these maximum speed and safety distance). All the simulations in this section were computed in a laptop using an Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz, 16Gb ram. The problem cost function, constraints and all the derivatives were implemented in Python 2.7.

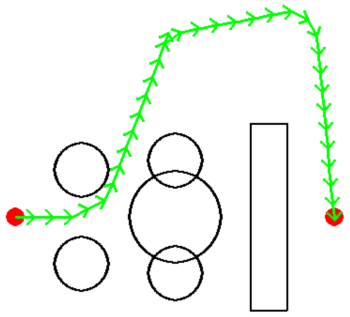


Figure 4.6: Initial trajectory, to be optimized

The trajectory was optimized until the local-minima was achieved 10 times, for both problems. The locally optimal trajectory is shown in Figure 4.7

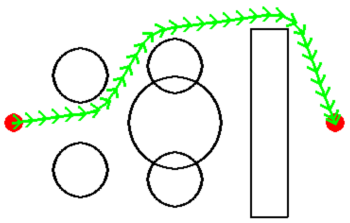


Figure 4.7: Final (locally optimal) trajectory.

The computational times are shown in Figure 4.8:

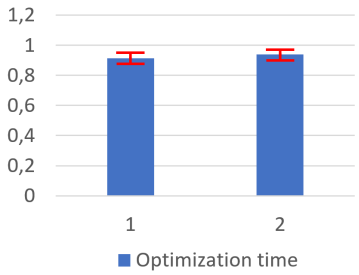


Figure 4.8: Computational time until local-minima is reached for both problems, using 100 runs for each.

The results suggest that there is no significant difference in computational performance between both problems. Using the same initial iteration and different number of intermediate checking points, the time for achieving a local minimum, for *Problem 2*, is shown in Figure 4.9 :

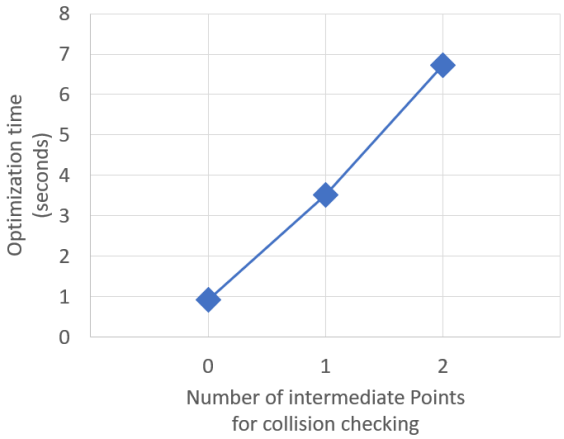


Figure 4.9: Computational time until local-minima is reached for *Problem 2*, for different numbers of intermediate points per trajectory segment, averaged over 10 runs.

As expected the computational time increases with the increasing number of collision checking points. However, when intermediate collision checking points are introduced, the time-step between waypoints can be increased. From the initial trajectory shown in Figure 4.6, $M-1$ for each M waypoints were deleted before the optimization starts, to make use of the advantage of having intermediate points. $M-1$ represents the number of intermediate points per segment. This can be seen as replacing waypoints for intermediate collision checking points, as shown in Figure 4.10.

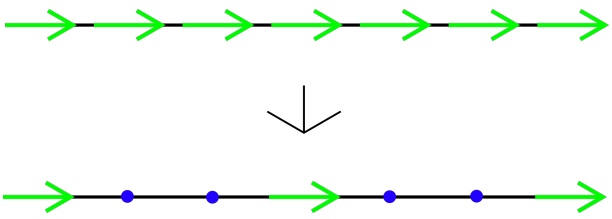


Figure 4.10: Trajectory before (top) and after (bottom) exchanging waypoints (green arrows) for intermediate collision checking points (blue circles). In this illustration the number of intermediate points per trajectory segment chosen was 2 (M=3).

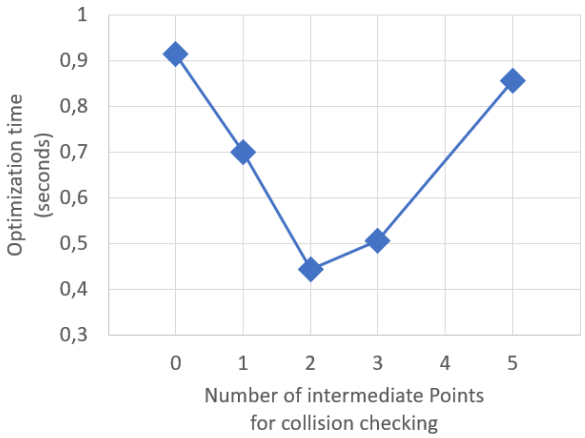


Figure 4.11: Computational time until local-minima is reached for *Problem 2*, for different numbers of intermediate points per trajectory segment (replacing waypoints), averaged over 10 runs.

Figure 4.11 show the time for computing a locally optimal trajectory when replacing waypoints for collision checking points:

From Figure 4.11 it is possible to observe that the computational time reduces when one or two intermediate points are used for collision checking. However, when the number of waypoints replaced by intermediate points is further increased the computational time starts to increase. This phenomenon might be because, when removing waypoints from the trajectory, the kinematic constraints are no longer satisfied and, for that reason, the algorithm takes longer to compute a locally optimal trajectory, given the fact that it also needs to ”fix” the trajectory. The results from these optimizations are shown in Figures 4.12 - 4.14.

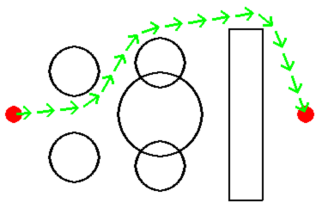


Figure 4.12: Locally optimal trajectory when 1 out of 2 waypoints are replaced by intermediate points.

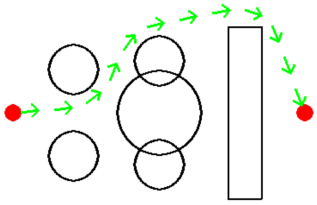


Figure 4.13: Locally optimal trajectory when 2 out of 3 waypoints are replaced by intermediate points.

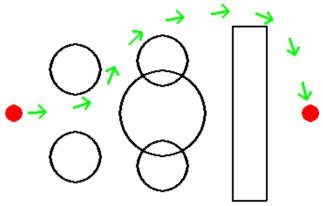


Figure 4.14: Locally optimal trajectory when 4 out of 5 waypoints are replaced by intermediate points.

4.10 Parameter tunning

Usually, in path planning, it is desired to achieve an optimal trajectory in terms of smoothness, time, energy consumption, among others. However, this approach is not suitable for some scenarios where there are guidelines to be followed. Imagine, for example, a driverless car trying to go from point A to point B in a city. If its trajectory planning algorithm only tries to minimize the driving time and is only constrained by the vehicle kinodymamics, then the car will likely drive on the wrong side of the road and

use inappropriate lanes.

For this reason, in some applications, other concerns must be taken into account. To do so the optimization problem must be re-formulated. In the present work, the guidelines are considered by introducing terms in the cost function, multiplied by constants. The guidelines could also be provided as constraints. They are added as penalizations so that the algorithm prioritizes the other constraints (collision avoidance for example). To understand this choice the urban driving example will be used once again. Imagine a narrow road with two lanes, the vehicles should drive on the right lane. Two vehicles go through that road in opposite directions, one of the vehicles loses control and goes to the other lane, towards the second vehicle. To avoid a collision the second vehicle must occupy the wrong lane or get off the road. For such scenarios, the optimization problem should be designed in such a way that it chooses to violate the guidelines over leading to a collision. To do so, in this work, it was chosen to add the guidelines as penalizations and the most critical restrictions as constraints.

To do so, the algorithms behaviours can be deliberately biased. In the present work, it was made by introducing terms in the cost function, multiplied by constants. By tuning the value of these constants it is possible to adjust the bias. Two examples will be given, in one of them the algorithm will be biased to generate trajectories at a defined constant altitude, in the other, it will be biased to follow TCAS like avoidance resolution: by following the desired climb rate in the initial part of the trajectory. This bias is not added as constraints so that the algorithm does not violate other constraints to satisfy the new ones.

4.10.1 Level Flight

For following the desired altitude, the error between the desired altitude z_{ref} and the altitude at each state was considered as a cost, the modified cost function is:

$$f_{new}(x) = f(x) + k_h \sum_{i=0}^{N-1} \|p_{i,z} - z_{ref}\| \tag{4.23}$$

Where $f(x)$ is the old cost function and k_h the constant that allows tuning the ”strength” of the altitude suggestion. Figure 4.15 shows how the trajectory evolves for different values of ”k_h, for two different initial iterations.

From Figure 4.15, it can be observed that it is possible to ”suggest” an altitude level for the trajectory without compromising any of the problem constraints.

4.10.2 Conflict resolution

The TCAS system, as it was discussed in Section 2.9, indicates climb rates for the pilot to follow, to avoid conflict. The term added to the cost function penalizes errors between the climbing speed and the desired climbing speed vz_{ref} for the first half of the waypoints. If this penalization considers all the waypoints than it will not change the trajectory because the average climbing speed is only defined by the start and goal altitudes and the trajectory time. The modified cost function is:

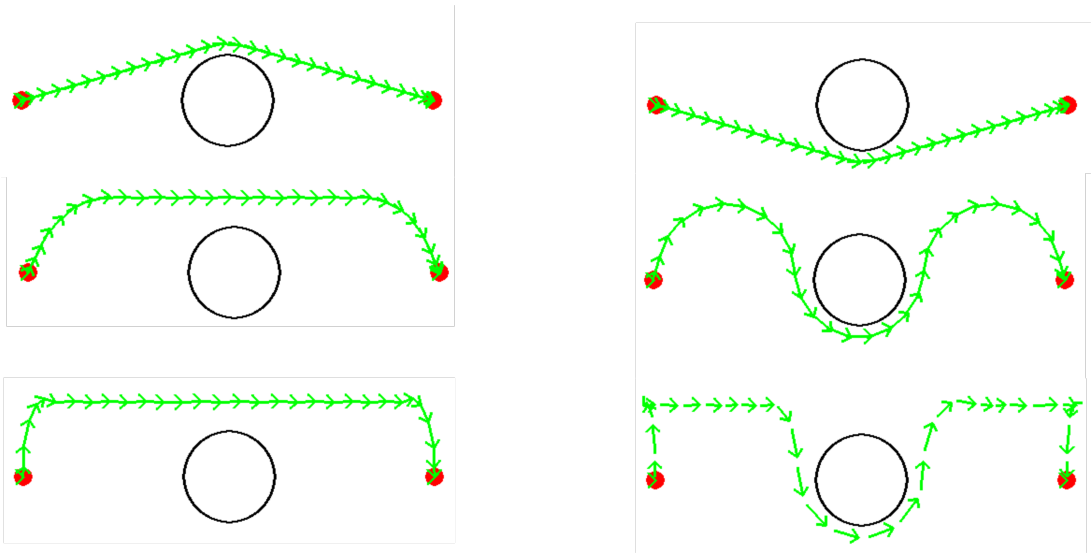


Figure 4.15: Trajectories for increasing values of k_h (from top to bottom) to force following a certain altitude. The results are shown for two possible first iterations, one that goes above the obstacle (trajectories the left) and another that goes below (trajectories the left).

$$f_{new}(x) = f(x) + k_c \sum_{i=0}^{round((N-1)/2)} \|v_{i,z} - vz_{ref}\| \tag{4.24}$$

Where $f(x)$ is the old cost function and k_c the constant that allows tuning the "strength" of the climb rate suggestion. Figure 4.16 shows how this modification allows changing the trajectory.

Figure 4.16: Trajectory before (top) and after (bottom) k_c is set from 0 to a positive value.

4.10.3 Tuning the energy cost

It is also interesting to visualize how the locally-optimal trajectory differs for different energy costs. In Figure 4.17 it is shown 3 different locally-optimal trajectories for the same first iteration and different cost values attributed to the energy consumption. From left to right the energy cost is increased relative to the trajectory time cost. As expected, the trajectory becomes slower. It is also possible to observe that the trajectory becomes shorter and less smooth.

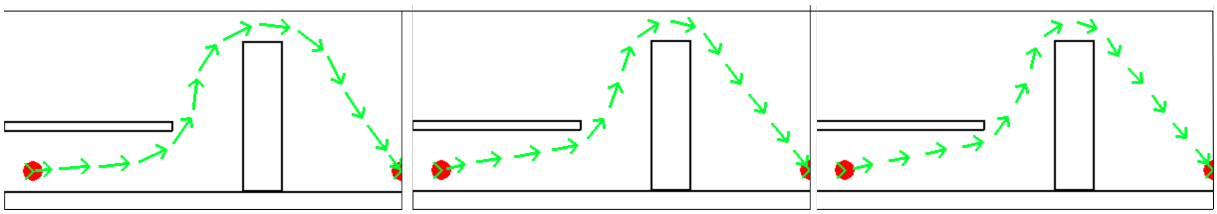


Figure 4.17: Locally-ptimal trajectories for increasing energy cost (from left to right). Smaller arrows represent lower speeds.

Chapter 5

Conceptual architecture

The software architecture for the real-time trajectory planner is now presented. The real-time trajectory planner launches a thread that continuously updates the trajectory and provides references (acceleration, speed and position) for the UAV to follow. It requires access to three methods. **Get time** that should provide a value for the current time when called, in seconds, **Get UAV State** that should return the UAV position and state when called and **Get Map** that should return a representation of the map when called.

To achieve this functionality, it is proposed the following architecture formed by 6 scripts:

- Types and Operations
- RRT
- Optimizer
- Feasibility checker
- Dynamic Avoidance Problem
- Real time trajectory planner

The scheme of the architecture is represented in figure 5.1.

The python script, **Types and Operations** contains the types declaration (Point, State and Obstacle) and also useful operations such as tensorial operations. There are 2 **Types and Operations** scripts implemented: 2D and 3D, which correspond to the types declarations and useful operations in two-dimensional and three-dimensional spaces respectively. All the remaining scripts described depend upon these ones.

The script **RRT** contains the implementation of the modified RRT algorithm proposed.

The script **Optimizer** contains the implementation of the trajectory optimization algorithm. There are already implemented two **Optimizer** scripts. Both of them use as solver an implementation of interior point optimization (IPOPT) [21] and each of them implements one of the formulations (formulation 1 and formulation 2) described in Section 4.8.

The script **Feasibility Checker** contains methods for determining if a computed trajectory is feasible for a given UAV. It also provides methods for finding in which part of the trajectory the failures occur and

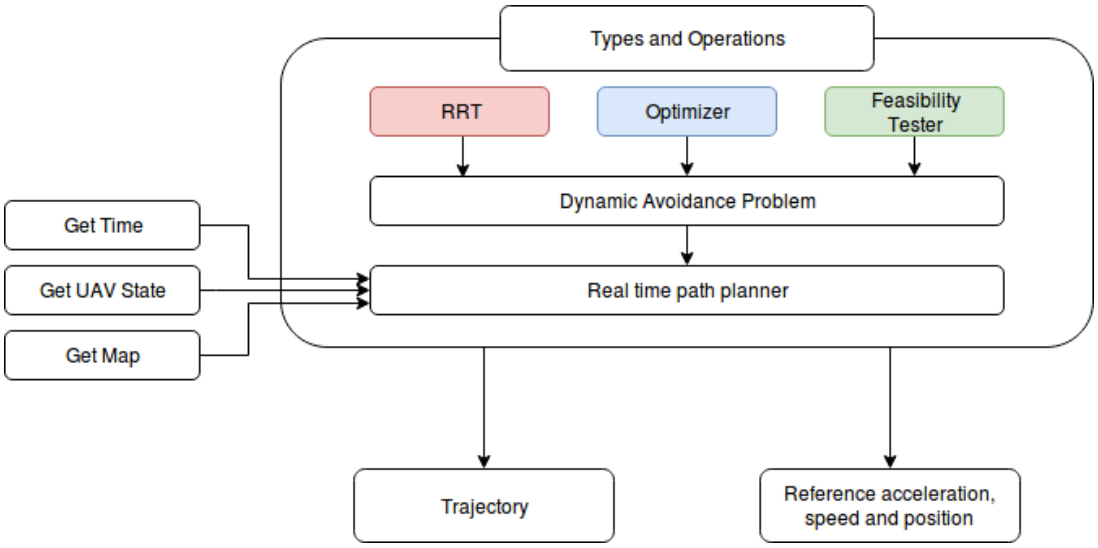


Figure 5.1: Architecture of the software implementation

what is the origin of such failures. These can be related to violations of the kinematics of the problem, maximum speed or maximum acceleration exceeded or parts of the trajectory that are too close to an obstacle.

The **Dynamic Avoidance Problem Script** provides an abstraction for the **RRT**, **Optimizer** and **Fedasibility Tester** scripts. It enables a centralized and simple way to call the implemented methods.

As mentioned before, the **Real Time Path Planner** launches a thread that continuously updates the trajectory and provides references (acceleration, speed and position) for the UAV to follow. It requires access to 3 methods. **Get time** , **Get UAV State** and **Get Map**.

5.1 Real-time trajectory-planner behaviour

This implementation can be more intuitively described using a simplified state machine diagram, as the one shown in Figure 5.2.

It will now be explained, with further detail, the behaviour of this script. This explanation will contribute to a deeper understanding of the developed trajectory planning solution. Each of the states, except the *Decision Maker*, have an associated timeout.

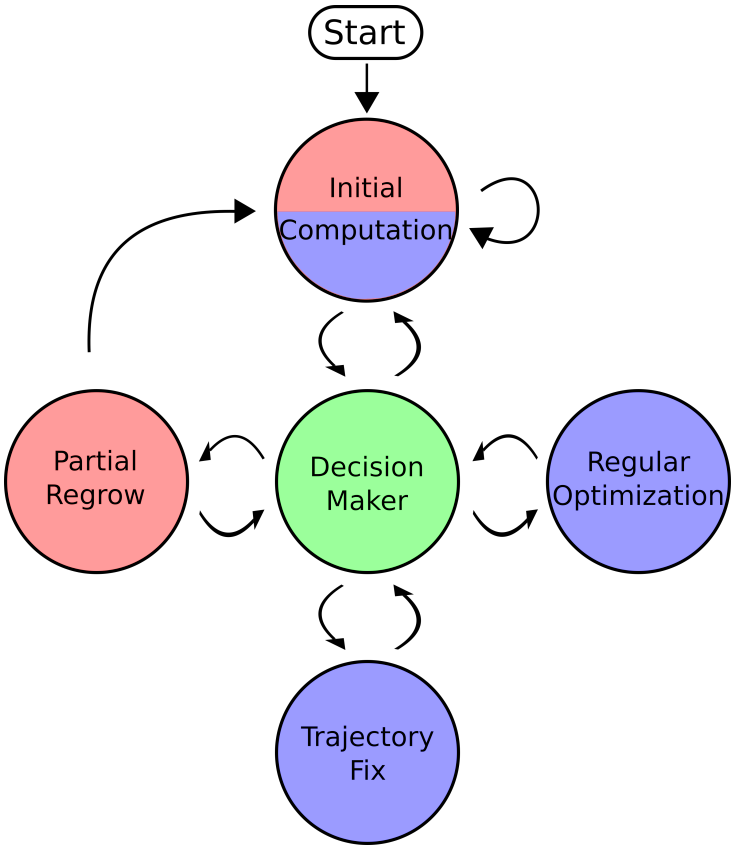


Figure 5.2: State machine describing the real-time trajectory planner. Red, blue and green states use methods from the **RRT**, **Optimizer** and **Feasibility Tester** scripts respectively.

5.1.1 Initial Computation

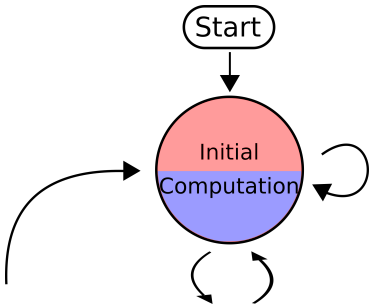


Figure 5.3: Initial computation state. It Uses both the RRT and Optimizer methods.

When the algorithm starts (and in some other situations) it enters the *Initial Computation* state (Figure 5.3). When this state is entered the UAV position is used as the starting state and if there is a trajectory, it is cleared. Then, a trajectory is computed using an RRT. If the RRT fails to compute a trajectory in a given time (*initialRRTTimeOut*) this state is re-started. If, on the other hand, the RRT can successfully compute a trajectory, a defined portion of that same trajectory is optimized for a defined time (*initialOptimizationTime*). This is where the anytime capability of the algorithm is evident, the *initialOptimizationTime* parameter controls the equilibrium between computational time and trajectory

quality.

5.1.2 Regular Optimization

In this state (Figure 5.4) a portion of the trajectory ahead of the UAV is optimized for a period of time defined by the parameter *regularOptimizationTime*. When this state is entered the algorithm chooses a portion of the trajectory between the point *regularOptimizationTime* ahead of the current time and another point ahead defined by the parameter *regularOptimizationPortion*. The portion of the trajectory which is optimized is shown in Figure 5.5. This choice assures that when the optimization is complete and the trajectory is updated, the UAV is not executing the updated part of the trajectory.

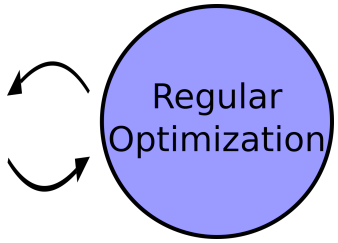


Figure 5.4: *Regular Optimization* state. Uses **Optimizer** script.

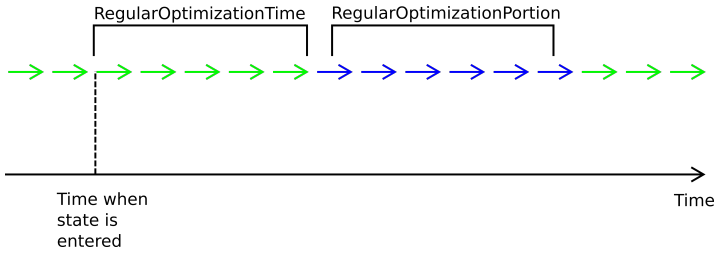


Figure 5.5: Portion of trajectory that is optimized in the *Regular Optimization* state is represented as a series of blue arrows. The first and last blue arrows are treated as a local start and a local goal (these waypoints are fixed during the optimization).
.

5.1.3 Trajectory Fix

The state *Trajectory Fix* is called when there is an unfeasible portion of the trajectory. It optimizes a portion of the trajectory for a certain time (*trajectoryFixTime*) part of the trajectory is controlled by the parameters *beforeFix* and *afterFix*. The trajectory is optimized between the time of the first violation minus *beforeFix* and the time of the last violation plus *afterFix*. Figure 5.7 illustrates this portion:

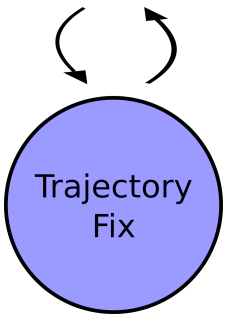


Figure 5.6: *Trajectory Fix* state. Uses **Optimizer** script.

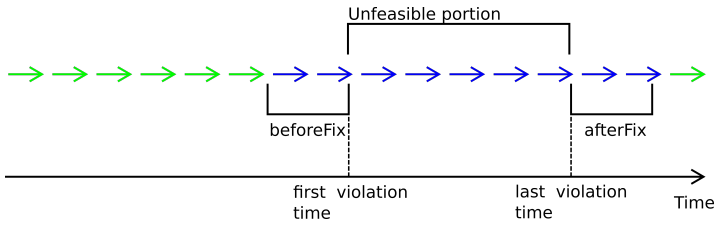


Figure 5.7: Portion of trajectory (blue) that is optimized in the state *Trajectory Fix*. The first and last blue arrows are treated as a local start and a local goal (these way-points are fixed during the optimization.

5.1.4 Partial Regrow

This state is called when the planner is trapped in an unfeasible local minima. Unlike the other states, where the trajectory is segmented based on time, this method separates the trajectory based on distance. It regrows the trajectory between the state that is at a distance *beforeRegrow* before the first violation index until *afterRegrow* after the last violation index, as shown in Figure 5.9. The time-out associated with this state is called *regrowTimeOut* and, if it is not possible to compute a trajectory within this time, the algorithm jumps to the *Initial Computation* state.

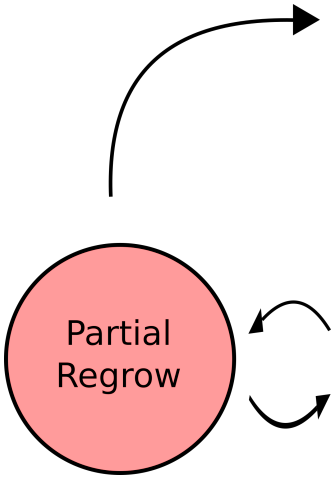


Figure 5.8: *Partial Regrow* state. Uses RRT methods.

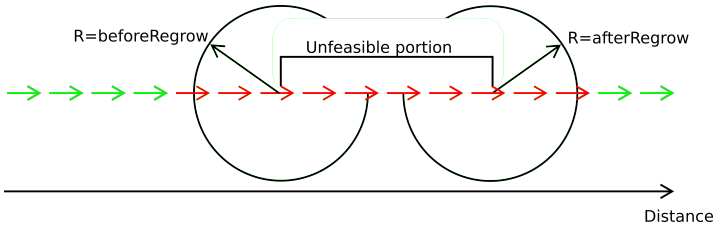


Figure 5.9: Portion of trajectory (red) that regrown in state *Partial Regrow*. The first and last red arrows are treated as a local start and a local goal for the RRT algorithm.

5.1.5 Decision Maker

The process behind the *Decision Maker* defines the logic of this algorithm. It implements the idea for the real-time trajectory-planning exposed in Chapter **??**. Its working principles will now be explained.

This state does not have an associated timeout. Instead, when the algorithm enters this state it is quickly set to another state. Basically, this state is responsible for determining the future state of the algorithm. A parameter called *failCount* is set to 0 after the initial computation. In this state, it is checked if there are any violations of the constraints in the trajectory (bad kinematics, maximum speed exceeded, maximum acceleration exceeded or too close to an obstacle). If there is not any violation, then *failCount* is set to 0 and the algorithm jumps to the *Regular Optimization* state. If there is a failure, it is checked if the first violation time is closer than a defined *criticalClearence*. If it is then the UAV is commanded to stop, *failCount* is set to zero and the algorithm jumps to the *Initial Computation* state. This represents an emergency stop of the UAV, to deal with critical situations. If the first violation time is after the defined *criticalClearence* than the *failCount* is incremented. If the *failCount* reaches a defined maximum (*maximumFailCount*), the *failCount* is set to 0 and the algorithm jumps to the *Partial Regrow* state. Otherwise the algorithm jumps to the *Trajectory Fix* state. To represent this idea in a more comprehensive way a flow-chart is presented in Figure 5.10.

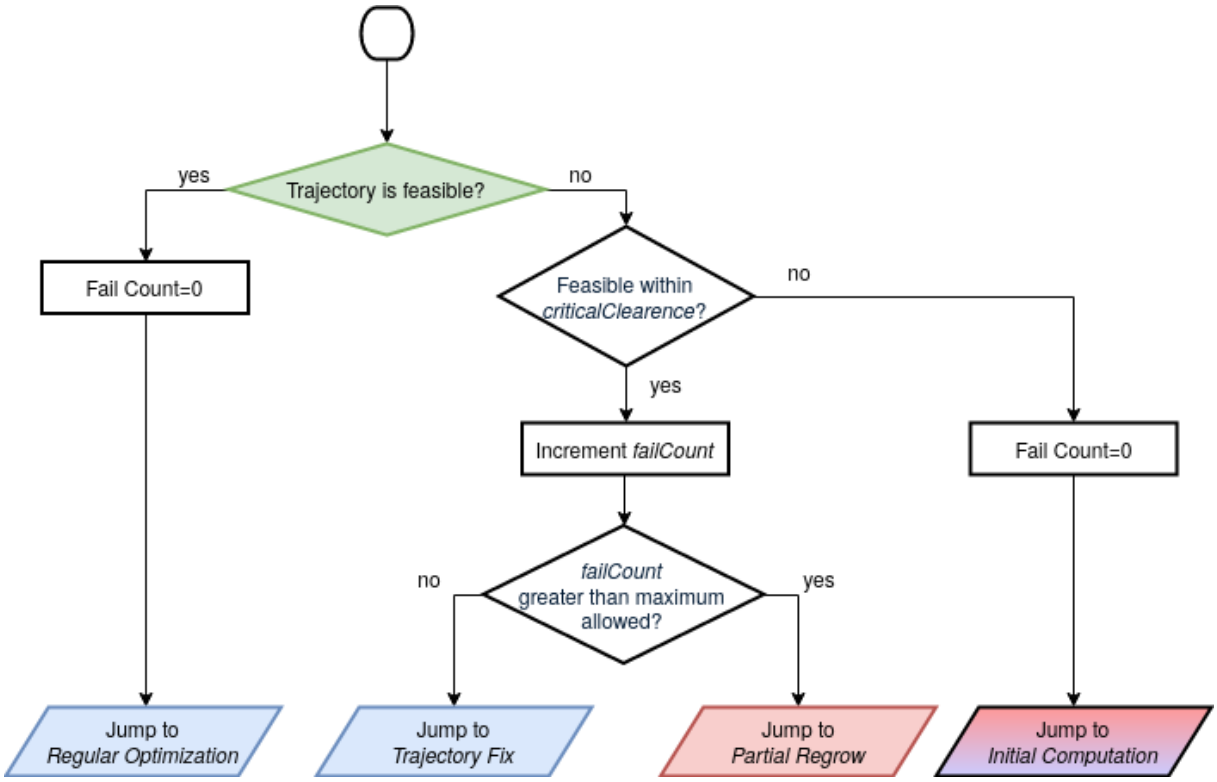


Figure 5.10: Flowchart representing the functionality of the *Decision Maker* state.

5.2 Integration with TCAS

5.2.1 Implemented simplified TCAS

A simplified TCAS system was developed. This system, as a normal TCAS system, tracks intruder aircrafts' range, altitude and bearing. A class *TCAS* and a class *transponder* were created. Instances of the class *TCAS* have one parameter that corresponds to a pointer to a *transponder* instance. When the TCAS system acquires an intruder aircraft, a pointer to the intruders transponder is stored. A method was developed in the trajectory planner that enables the command of climb rates (the execution behind this is described in Section 4.10.2). A pointer for that same method is provided to the *TCAS* class, enabling the TCAS thread to provide resolutions (resolution advisories or RA) to the planner.

Chapter 6

Simulation

In this chapter several simulations, and their respective results, are presented. It will be explained how a simplified simulation framework was developed to evaluate the performance of the algorithm in real-time. Then several results are presented showing the capability of the algorithm to adapt to changes in the environment, deal with other aircraft and follow resolutions provided by the TCAS system.

To validate the usage of the algorithms for real-vehicles, realistic physics simulations were performed using Gazebo. The used plugin and controller are briefly explained. A simulation is analysed to evaluate the capability of the simulated multi-rotor to follow the desired trajectories. Simulation of both the planning algorithms and the TCAS is also performed and analysed.

6.1 Simplified Simulation

The simplified model of the UAV used in this testes consisted of a point with a given mass to which the control inputs were applied as forces. A simple controller was developed to transform the reference acceleration, speed and position into force inputs for the model.

6.1.1 Real-time avoidance

The following figures (Fig. 6.1-6.12) show the results for real-time testing of the algorithms in a complex indoor environment with unexpected obstacles appearing in the map as the model follows the trajectory.

Real-time avoidance was also performed using moving obstacles. For avoiding non-cooperative intruders the algorithm assumes that these intruders travel at a constant speed. Once the trajectories of these intruders might be unpredictable, the safety distance that the algorithm considers for these intruders is k_d times the one considered for static obstacles and cooperative intruders. The algorithm was also empowered to be able to simulate sensor delay.

Several simulations were performed to validate the capability of avoiding moving intruders. In Figure 6.13-6.19 the aircraft flies through a map of known obstacles. It was possible in this simulation to manually insert moving obstacles. These moving obstacles start in a desired position at the top of the map and they move (at constant speed) towards the bottom of the map. These moving obstacles are

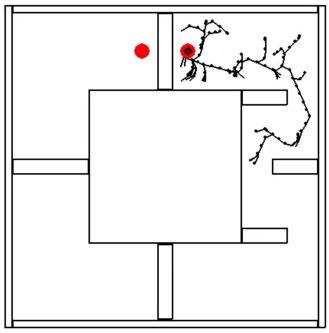


Figure 6.1: RRT is grown

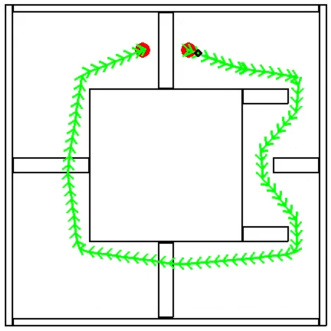


Figure 6.2: Initial trajectory is computed

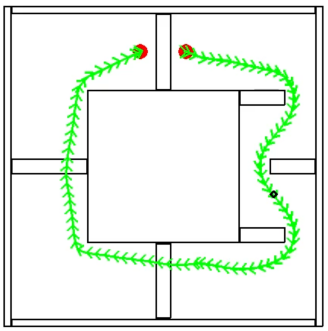


Figure 6.3: The algorithm optimizes part of the trajectory ahead of the aircraft, it is visible that the optimized trajectory segment is smoother than the rest

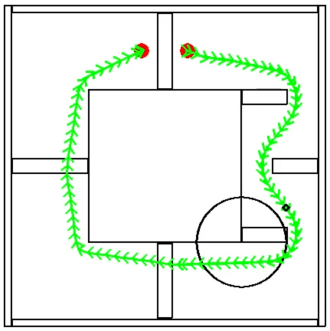


Figure 6.4: Obstacle is detected

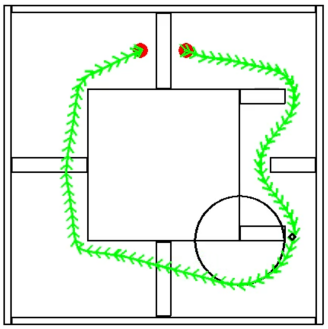


Figure 6.5: Trajectory optimizer adjusts the trajectory avoiding the obstacle

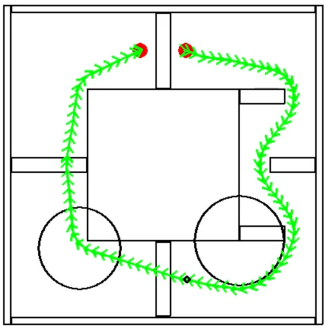


Figure 6.6: Obstacle is detected

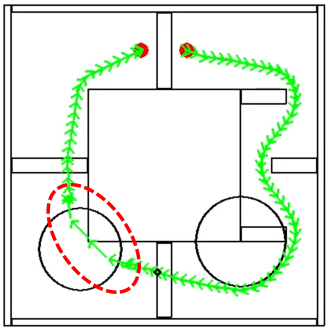


Figure 6.7: Trajectory optimizer gets trapped in an unfeasible local minima, signalized by the red dashed ellipse (between the sphere and the cuboid obstacles).

represented as circles. As it is possible to observe the algorithm was capable of avoiding these obstacles as well.

There is also the need for simulating intruders that do not move with constant speed. For that, the trajectory of an aircraft moving in a cluttered map was stored. It was then possible to manually insert an intruder that would follow this trajectory, unknown to the algorithm. The algorithm only knows the position and speed of the intruder with one second of delay, to simulate sensor processing time. Two

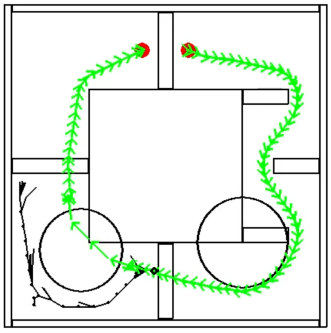


Figure 6.8: RRT algorithm regrows the critical part of the trajectory

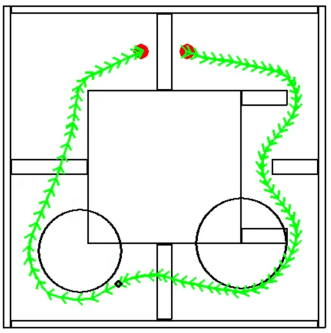


Figure 6.9: A feasible trajectory is found

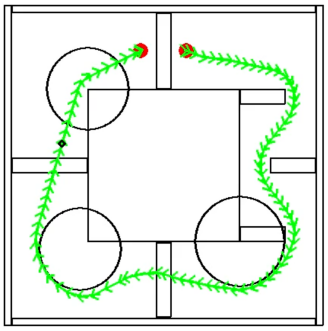


Figure 6.10: Obstacle is detected critically close, trajectory optimizer would not have time to react in real time

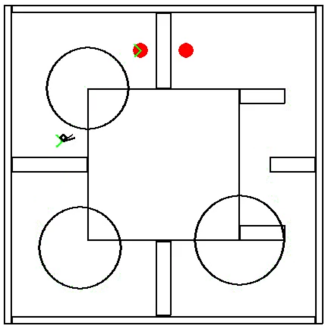


Figure 6.11: All references are turned off to prevent the UAV from colliding. Previous trajectory is discarded

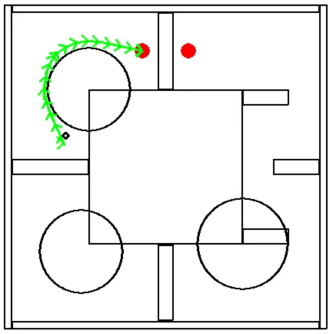


Figure 6.12: RRT algorithm regrows a new trajectory from the UAV position to the goal.

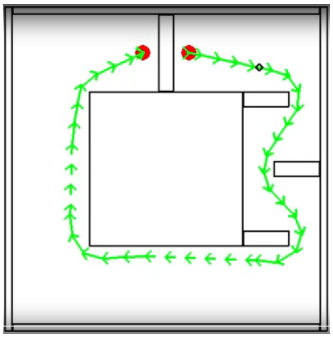


Figure 6.13: Before unexpected obstacle appears

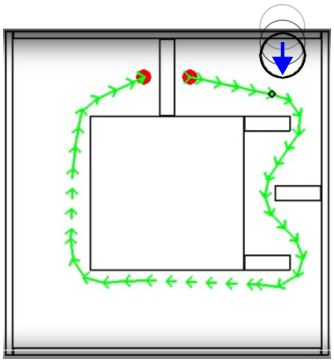


Figure 6.14: Unexpected obstacle appears (top left of the figure), moving from the top to the bottom of the figure.

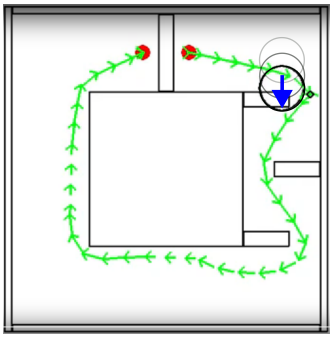


Figure 6.15: Trajectory is adjusted and aircrafts ”waits” for the obstacle to pass before proceeding.

runs are now shown, in the first one (Figures 6.20-6.22) the intruder is launched later than in the second (Figures 6.23-6.25):

To test the limits of the algorithm a 2D simulation was run with an intruder traveling in a zig-zag trajectory towards the aircraft using the produced algorithm. This represents a very challenging situation since the intruder trajectory is constantly changing direction, making it difficult for the algorithm to determine if the aircraft should go above or below the intruder. This simulation was run 10 times and the algorithm failed 4 times (collision happened). One successful simulation is presented in Figures 6.26-6.32. This

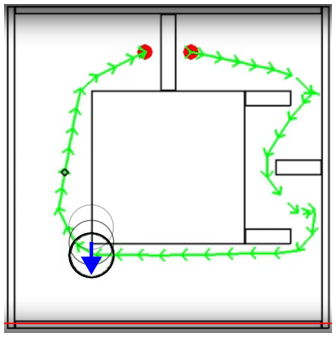


Figure 6.16: Before the last moving obstacle appearance.

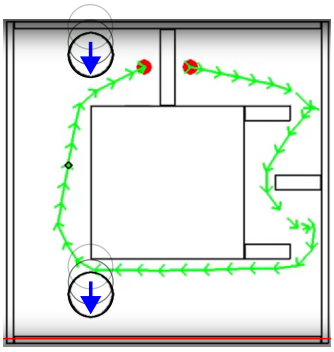


Figure 6.17: New obstacle appears, moving from the top to the bottom of the figure

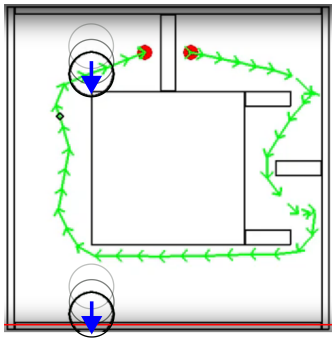


Figure 6.18: Trajectory is adjusted.

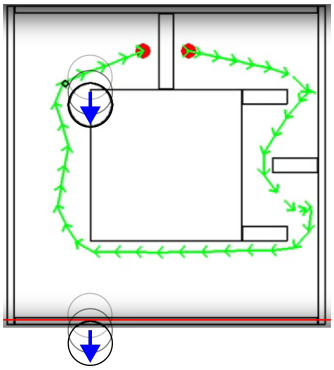


Figure 6.19: The obstacle is avoided.

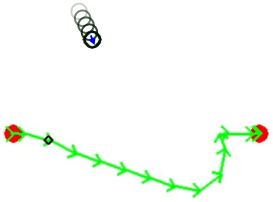


Figure 6.20: Trajectory while intruder flies to the right

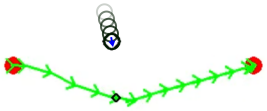


Figure 6.21: Trajectory changes as intruder changes direction

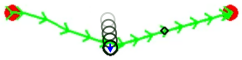


Figure 6.22: Intruder is avoided

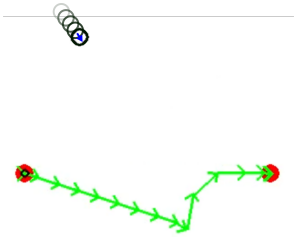


Figure 6.23: Initially algorithm plans to go below the intruder

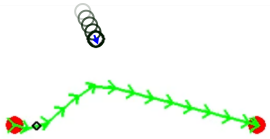


Figure 6.24: Algorithm plans to go above the intruder

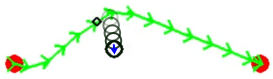


Figure 6.25: Intruder is avoided

poor performance in this situation arises mainly from two factors. First, the ideal avoidance trajectory for the obstacle, when it is climbing/descending goes under/above the obstacle, once the obstacle is constantly changing direction the computed trajectories quickly become unsuitable for the avoidance.

Secondly, the simulated sensor delay originates a significant error in the prediction of the position of the obstacle, once this prediction is based on a constant speed assumption.

Figure 6.26: Intruder is detected. As the intruder descends the algorithm does not detect a trajectory conflict

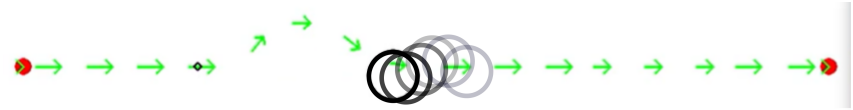


Figure 6.27: The trajectory is adjusted as the intruder starts to climb

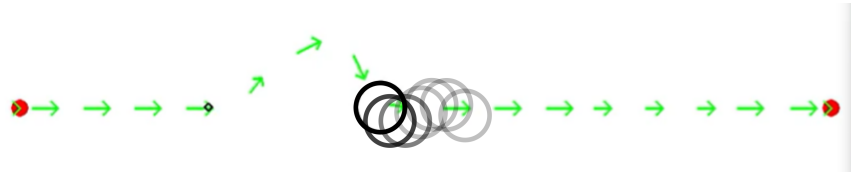


Figure 6.28: Intruder starts to increase the climb rate. Trajectory is further adjusted.

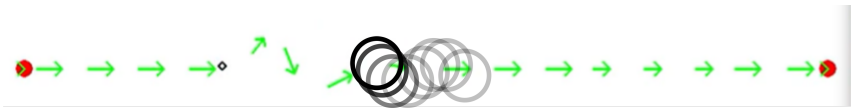


Figure 6.29: Intruder starts to climb with such a rate that it becomes (apparently) impossible to go above it. Trajectory is adjusted to go under the intruder.

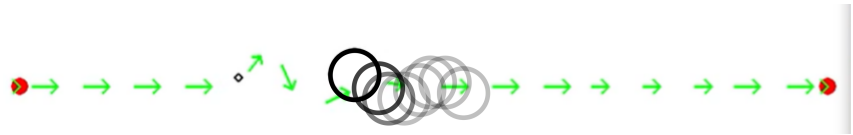


Figure 6.30: Intruder starts to descend

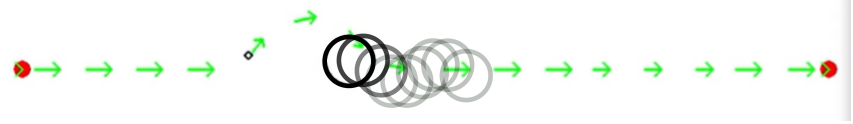


Figure 6.31: Algorithm (one second later) is informed that intruder is descending and tries to avoid it from above

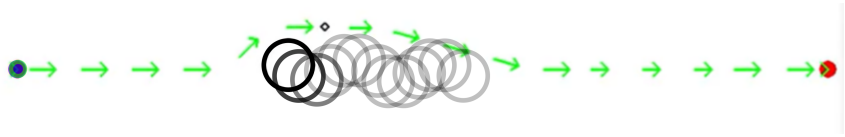


Figure 6.32: Avoidance is successful

6.1.2 Simulating an unknown environment

To simulate an unknown environment the algorithm was used to compute the trajectories in an environment, without knowing the position or number of obstacles in that environment. It was then assumed that the algorithm would acquire information on an obstacle when the aircraft was at a distance to the obstacle smaller than a certain detection range. The evolution of one of these simulations is now shown in Figures 6.33-6.38.

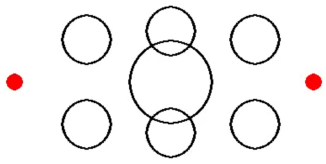


Figure 6.33: Complete map

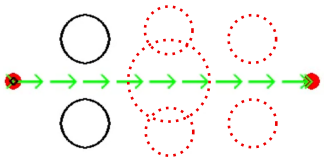


Figure 6.34: Initially only two obstacles are visible, none of them interferes with the trajectory

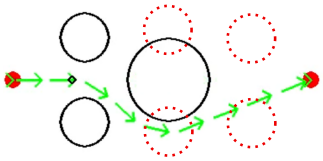


Figure 6.35: An obstacle is detected and the trajectory is adjusted.

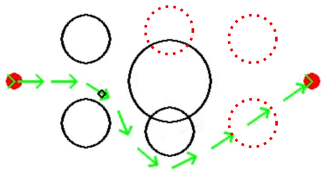


Figure 6.36: New obstacle is detected and the trajectory adjusted

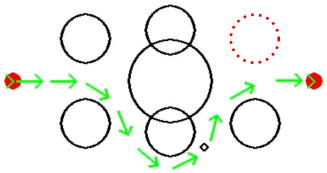


Figure 6.37: Another obstacle is detected and the trajectory is adjusted

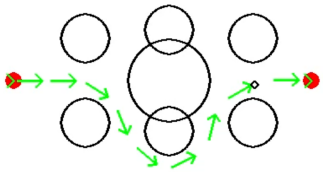


Figure 6.38: The final obstacle from the map is detected, it does not, however interfere with the trajectory.

6.1.3 Random maps

For preforming a statistical analysis of the performance of the algorithm several simulations were run in random maps, Figures 6.39 and 6.40 show one of these random maps and the computed trajectory through it, respectively. Notice that the trajectory is sub-optimal once the algorithm does not know the map beforehand.

Each map has 20 randomly generated circles, with a radius between 10 and 40 (display units). The distance from start to goal was 370 display units. Figure 6.41 shows the time taken to reach the goal.

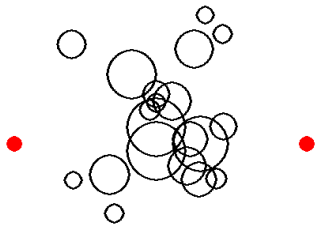


Figure 6.39: Random unknown map.

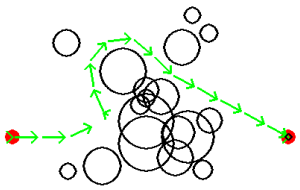


Figure 6.40: Final executed trajectory.

The algorithm was not able to compute a trajectory once, in that case, the aircraft stopped at a distance smaller than the safe distance, relative to an obstacle. For that reason, the RRT algorithm could not output any safe trajectory.

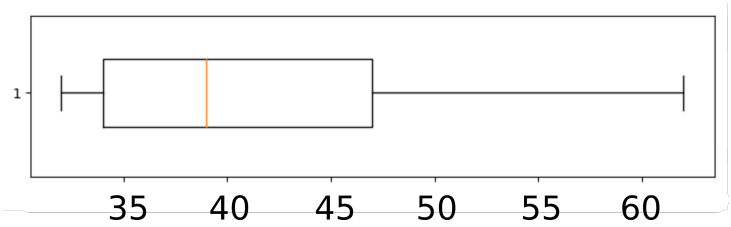


Figure 6.41: Time, in seconds, taken for the agent to reach the goal position, on the unknown random maps.

6.1.4 Multi-Aircraft

In a realistic scenario, several aircraft share the same air-space. In a future where urban transportation is also accomplished using aircraft, the algorithms must enable safe circulation. Once this work aims for a decentralized approach it is expected that algorithms running independently in several aircraft can provide collision-free paths, without a centralized entity making a decision.

The decentralized traffic avoidance was tested on the map in Figure 6.42, and two different scenarios where analysed:

- Both aircraft know the other aircraft position and planned trajectory. (Fig. 6.43-6.45)
- Aircraft number 2 knows aircraft number 1 position and planned trajectory. Aircraft number 2 is invisible to aircraft no 1. (Fig. 6.46-6.48)

The second scenario represents the case in which one of the aircraft (aircraft number 1) has priority and therefore it is not compelled to change its path to avoid the lower priority aircraft (aircraft number 2).

The algorithms are constantly replanning the trajectories over time, therefore knowing an aircraft X trajectory at a given moment is only temporarily useful for aircraft Y, once aircraft X trajectory will be re-planned.

The experiment was conducted 10 times for each of the scenarios and both aircrafts managed to arrive at the destination without any collisions in all trials.

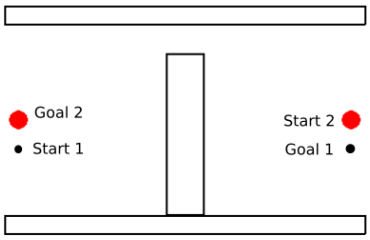


Figure 6.42: Map with the starting and goal positions for aircrafts number 1 and number 2

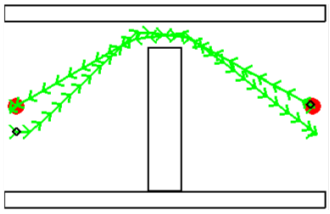


Figure 6.43: Initial trajectories, computed simultaneously, in collision route. Both aircraft communicate their trajectories (scenario 1).

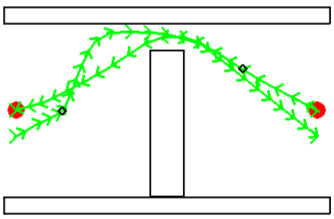


Figure 6.44: The algorithm, in an independent process for each aircraft, enables the generation of collision free trajectories (scenario 1).

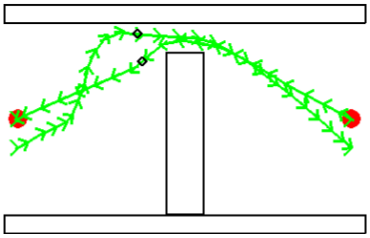


Figure 6.45: The aircrafts avoid each other (scenario 1).

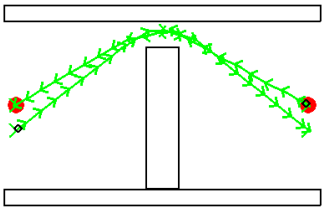


Figure 6.46: Initial trajectories, computed simultaneously, in collision route. Aircraft 2 is invisible for aircraft 1 (scenario 2).

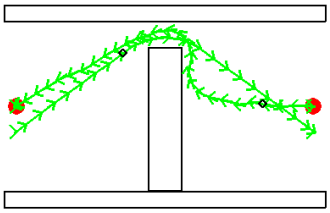


Figure 6.47: The algorithm running for aircraft 2 plans a trajectory that avoids aircraft 1. Aircraft 1 plans a trajectory without considering the other aircraft (scenario 2).

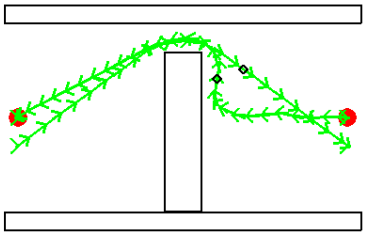


Figure 6.48: The aircrafts avoid each other (scenario 2).

6.1.5 TCAS simulation

The previous results are motivating from the perspective of testing the planner in demanding, changing environments. However, for cooperative systems, the approach taken must be different. Airborne avoidance of cooperative aircraft will ultimately be done by integrating existing collision avoidance systems based on protocols, specifically the TCAS.

For the TCAS testing, an environment without obstacles was used. One of the aircraft follows a trajectory from point A to point B, the other from point B to point A. The aircraft are in collision route. The aircraft do not have any information on the other aircraft, except for the data traditionally provided by the mode-S transponders for the TCAS system (range, altitude and bearing, the bearing is not used in this simulation).

In this test, as illustrated in Figures 6.49-6.51 and as expected, the TCAS systems identified the possible conflict and provided resolutions. One of the aircraft was commanded to climb and the other to descend. The real-time path planner followed these resolutions successfully until the TCAS system declared clearance of conflict.

Figure 6.49: Both aircrafts are in collision route

Figure 6.50: TCAS systems determine resolutions and the trajectory-planner adapts the trajectories

Figure 6.51: TCAS systems determine that the conflict is cleared and both aircrafts fly once again towards the goal.

6.2 Physics simulation

The previous algorithms were developed taking as assumptions that the UAV could be described as a body subjected to forces as inputs. The resulting trajectories are a sequence of constant acceleration segments. Such a formulation of the problem leads to the existence of discontinuities in the acceleration. A physical multi-rotor cannot, however, provide discontinuities in the acceleration, even if it is considered that the rotor velocities can change instantaneously. For this reason, some works use polynomials with higher degrees, to provide continuity of the acceleration and jerk (3rd derivative of the position) [11, 12, 43]. This is associated with the fact that the multi-rotor can only produce thrust in the axis perpendicular to the rotors' plane, meaning that discontinuities in the acceleration and jerk would correspond to discontinuities in the multi-rotor attitude and angular rate respectively. In [12] and [43] the authors use 11th and 9th order polynomials for computing the trajectories, respectively. In [11] the authors use a spline that assures continuity up to the 4th derivative of the position for smoothing the trajectory.

The multi-rotor dynamics will not be deeply discussed in this work, for keeping it concise. However, there is a need to verify the capability of a realistic multi-rotor to follow the computed trajectories, once the simplified model used to validate the real-time capability of the algorithms does not rigorously approximate a real UAV.

6.2.1 RotorS Gazebo

The Robotics Operative System (ROS) [1] supports a physics engine named Gazebo [44]. A plugin for Gazebo, RotorS, was developed in the Autonomous Systems Lab of ETH Zurich university [45]. This plugin includes multi-rotor models and example controllers, that are used in the current work. This open-source simulator also has the advantage of enabling the simulation of a variety of sensors and allow a posterior straight forward implementation in real multi-rotors.

6.2.2 Controller

The controller was adapted from the one described in the work by Lee et al. [46]. This controller has an inner and an outer loop. The outer loop is the position controller, it is responsible for transforming a reference speed and a reference position into a desired resulting thrust vector. The inner loop is responsible for tilting the quad-rotor to such an attitude that the desired thrust vector is produced. The outer loop determines what should be the acceleration of the quad-rotor and the inner loop is responsible for making the vehicle acquire that same acceleration. Fig. 6.52 presents a scheme of that same controller.

The desired acceleration is then computed using the expressions presented in Equation 6.1.

$$\mathbf{a}_d = \mathbf{a}_{ref} + (k_v(\mathbf{v}_{ref} - \mathbf{v}) + k_p(\mathbf{p}_{ref} - \mathbf{p}))\frac{1}{m} - \mathbf{g} \tag{6.1}$$

[1] ROS (robot operating system), documentation. accessed in 8th July, 2019. URL http:wiki. ros.org.

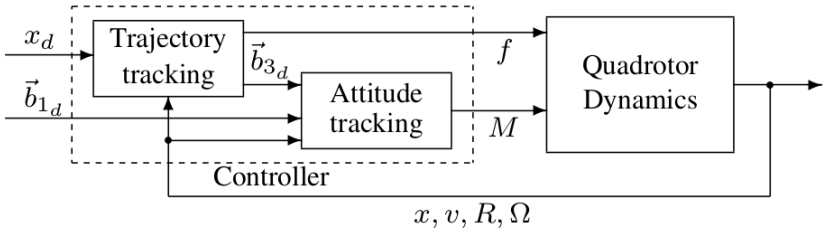


Figure 6.52: Controller structure, taken directly from [46]. x_d represents the desired position speed and acceleration, $\mathbf{b}_{3_d}$ represents the axis perpendicular to the rotors' plane and $\mathbf{b}_{1_d}$ is related to the UAV heading.

In equation 6.1 $\mathbf{a}_d$ represents the desired acceleration vector. $\mathbf{a}_{ref}$, $\mathbf{v}_{ref}$ and $\mathbf{p}_{ref}$ represent the reference acceleration, speed and position respectively, $\mathbf{v}$, $\mathbf{p}$ represent the vehicle measured speed and position, m represents the vehicle mass and finally $\mathbf{g}$ represents the gravitational acceleration.

6.2.3 Testing trajectory tracking

Map

It was chosen a simple map for validation of the algorithm. The map consisted of a gazebo model of a fast-food chain restaurant. Fig. 6.53 and 6.54 represent the simulation scenario and the top view scheme respectively.

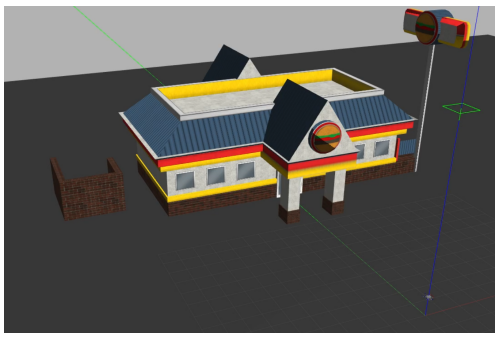


Figure 6.53: Map visualization on the Gazebo simulator environment

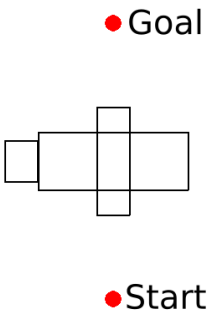


Figure 6.54: Scheme of the top view of the Map.

Results

The results of four runs are now presented. The runs differ in maximum acceleration allowed and the side of the building taken to arrive at the goal position. In all the runs the minimum distance allowed between the UAV and an obstacle was set to 3 meters and the maximum speed allowed for the UAV to travel was 15 m/s (54 km/h). The run identifiers are defined in table 6.1.

For each run, it is now presented the trajectory taken by the vehicle in Fig. 6.55-6.58.

Run Identifier	Maximum Speed (m/s)	Maximum Acceleration (m/s^2)	Side taken
1	15	6	Left
2	15	6	Right
3	15	10	Left
4	15	10	Right

Table 6.1: Simulation run identifiers

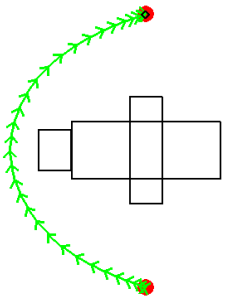


Figure 6.55: Trajectory taken in run 1 (Max. acceleration = 6m/s^2).

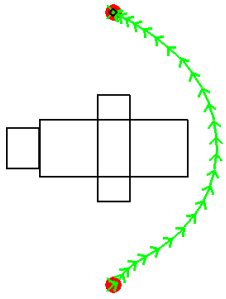


Figure 6.56: Trajectory taken in run 2 (Max. acceleration = 6m/s^2).

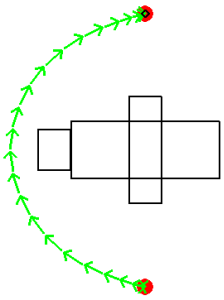


Figure 6.57: Trajectory taken in run 3 (Max. acceleration = 10m/s^2).

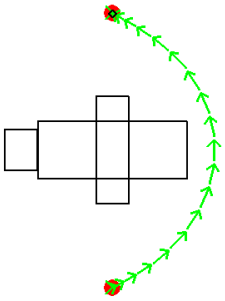


Figure 6.58: Trajectory taken in run 4 (Max. acceleration = 10m/s^2).

The norm of the difference between the aircraft position and the reference position provided by the algorithm was stored over time for every time an update was received on the aircraft position. The evolution of the position error along the time is now presented for the four runs in Fig. 6.59-6.62.

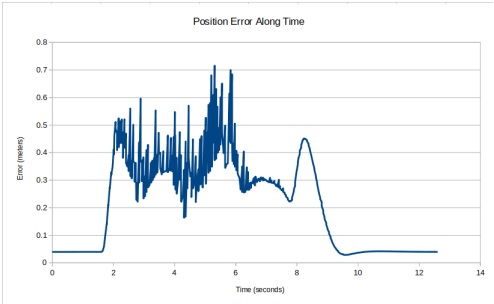


Figure 6.59: Position error along the simulation time in run 1 (Max. acceleration = 6m/s^2).

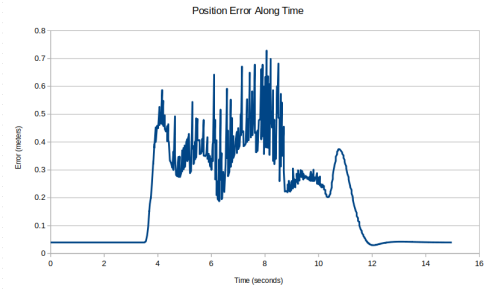


Figure 6.60: Position error along the simulation time in run 2 (Max. acceleration = 6m/s^2).

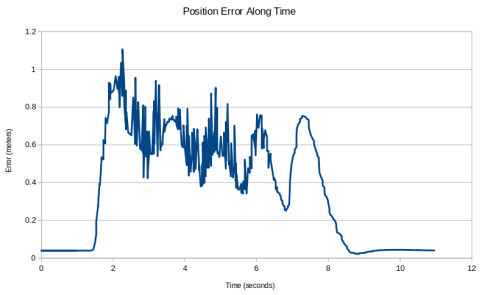


Figure 6.61: Position error along the simulation time in run 3 (Max. acceleration = 10m/s^2).

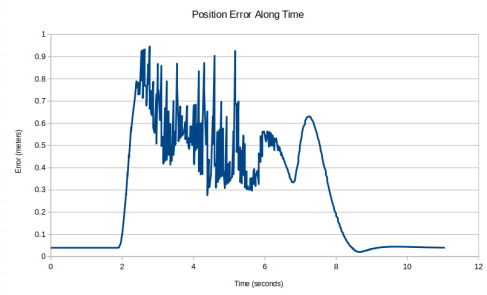


Figure 6.62: Position error along the simulation time in run 4 (Max. acceleration = 10m/s^2).

It is possible to observe in Fig. 6.59-6.62 that the position error is low and stabilized while the UAV is hovering, before and after executing the trajectory. The error does not tend to zero because the controller expression does not contain an integral term. During the trajectory execution, the position error rises but it is never greater than one meter except for brief moments in run 3. As expected the error is greater when the maximum acceleration allowed is greater (for runs 3 and 4).

It is now presented also the error distributions (Fig. 6.63-6.66). The graphs were created by counting the number of position readings that verified a position error within a 0.02 meter interval (for example, the number of readings corresponding to the error 0.12 corresponds to the number of readings in which the error falls between 0.12 and 0.14 meters). The same conclusions obtained for Fig.6.59-6.62 can be drawn.

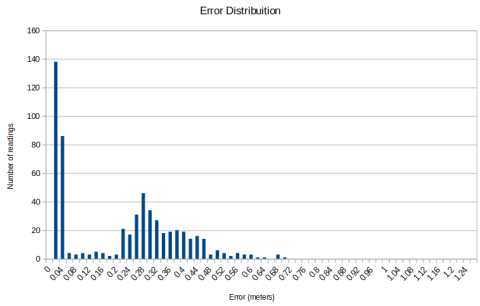


Figure 6.63: Position error distribution in run 1.

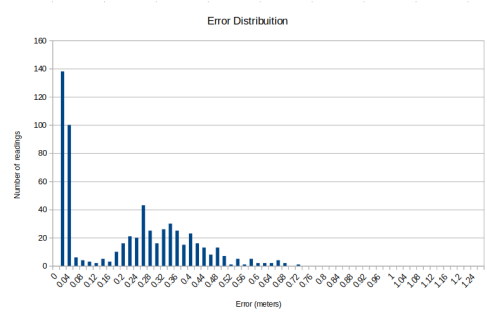


Figure 6.64: Position error distribution in run 2.

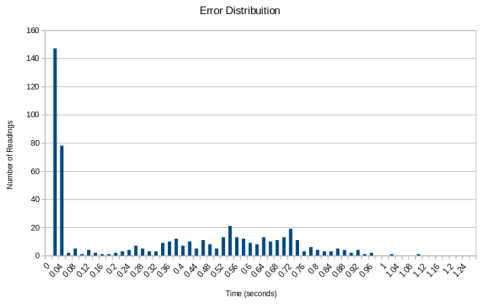


Figure 6.65: Position error distribution in run 3.

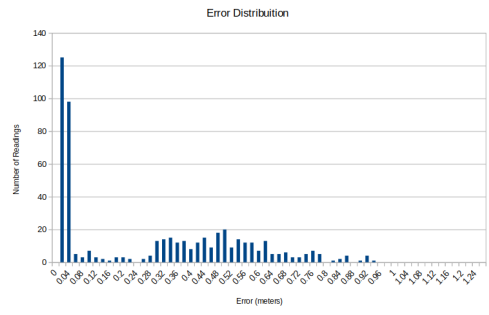


Figure 6.66: Position error distribution in run 4.

The presented results suggest that the algorithms are suitable for computing aggressive trajectories.

Figure 6.67: Vehicle assuming an attitude to accelerate from the initial hovering position

Figure 6.68: Vehicle assuming an attitude to perform a curve trajectory segment

The discontinuities in the acceleration do not lead to unacceptable errors for many applications, once the position error is always smaller than 1.2 meters. Besides, it can be observed that for trajectories with smaller acceleration the position error is also smaller, as expected. By limiting the maximum speed and acceleration it is possible to reduce the position error, for applications that require so.

6.3 TCAS system testing

A simulation was performed to validate the correct function of the TCAS system and its proper integration with the remaining solution. Some parameters were changed on the traditional TCAS system to obtain a less conservative behavior. These changes are shown in Table 6.2.

	Tested Solution	Conventional TCAS
RA tau (s)	10-15	15-35
TA tau (s)	20-30	20-48
RA DMOD (m)	80-100	370-2037
TA DMOD (m)	160-200	556-2408
RA ZTHR (m)	15-20	182-244
TA ZTHR (m)	30-40	259-336
Interrogation rate (Hz)	1	0.1-1

Table 6.2: Changed TCAS parameters for testing purposes. These parameters are explained in Section 2.9.

These changes were performed once the algorithms showed being able to quickly follow the desired resolutions provided by the TCAS. Similarly, to the test performed with the simplified dynamics (Section 6.1.5), in the present test two aircraft are set to travel in collision route. The aircraft only share the information that is usually available to the TCAS system, from the mode-S transponders (range, bearing and altitude). The simulated interrogation rate was 1Hz, the same as for commercial aircraft [33]. The aircraft were now simulated in Gazebo to enable a test with realistic multi-rotor dynamics. Figure 6.69 shows the simulated aircraft.

A virtual display was developed to emulate the TCAS display usually available for human pilots. This display allows a better understanding of the TCAS system state at each moment. The evolution of the display is shown in Figures 6.70-6.73.

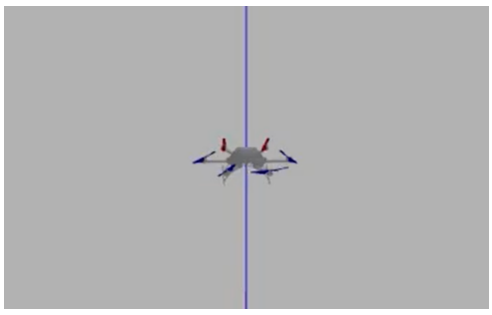


Figure 6.69: Simulated multi-rotor.

6.3.1 Results

Several measures were stored along the time to allow an analysis of the simulation. Figure 6.74 presents the altitude of each of the aircraft (vertical axis) corresponding to their horizontal position (horizontal

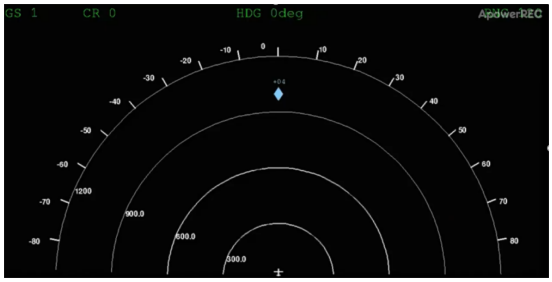


Figure 6.70: Intruder aircraft is detected.

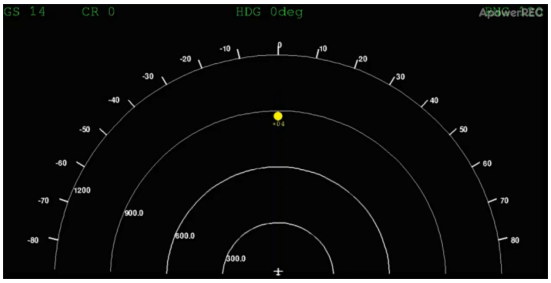


Figure 6.71: TCAS declares proximate aircraft.

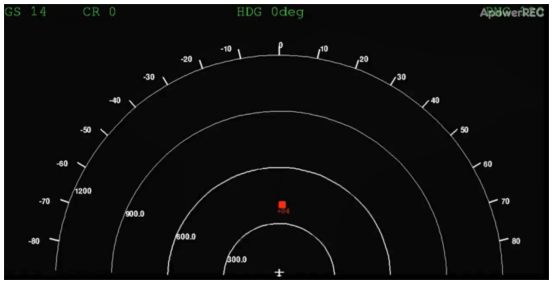


Figure 6.72: Resolution is provided by the TCAS.

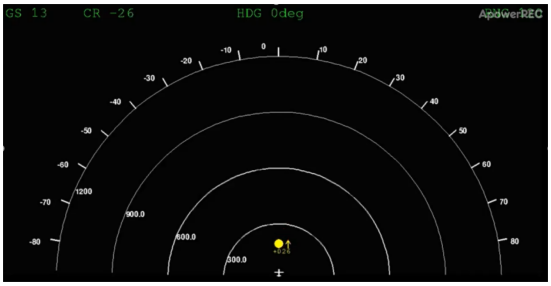


Figure 6.73: Clear of conflict.

axis) along the simulation. It is possible to see in Figure 6.74 that the conflict was cleared by the TCAS systems. The TCAS system corresponding to the aircraft that departed from the right (orange in the graphic in Figure 6.74) resolved the conflict by determining a certain climb rate. The TCAS system corresponding to the other aircraft (blue) determined a descend rate to solve the conflict. It is now presented the altitude and climb rate of one of the aircraft (orange in the previous graphic) along the time in Figures 6.75 and 6.76. In these graphics, the vertical lines represent the moments for which the TCAS system declared the traffic alert (TA), the resolution advisory (RA) and the conflict clearance, from left to right respectively.

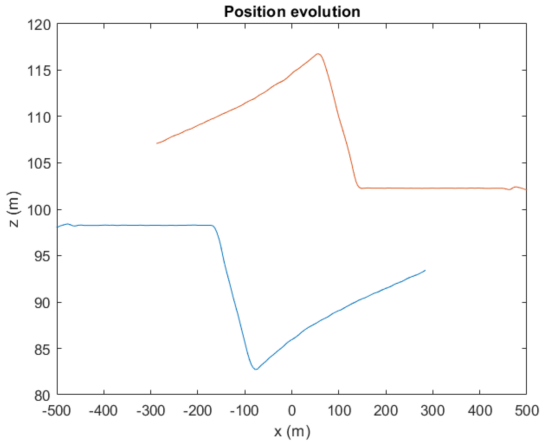


Figure 6.74: Trajectory of the aircrafts (altitude in the vertical axis and horizontal coordinate in the horizontal axis).

It is clear from Figures 6.75 and 6.76 that the aircraft starts climbing after the resolution advisory. There is however a certain delay of approximately 3 seconds, due to the processing time of the optimizer.

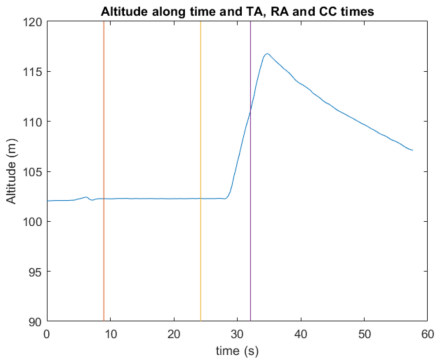


Figure 6.75: Altitude of one of the aircrafts over time with advisory markings.

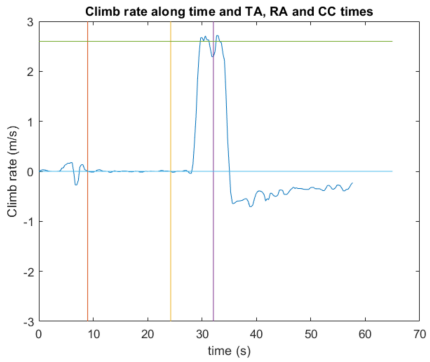


Figure 6.76: Climb rate of one of the aircrafts over time with advisory markings.

In Figure 6.76 it is also possible to see the desired climb rate (green horizontal line) that is correctly maintained by the aircraft during the avoidance.

Chapter 7

Conclusions

In this work, it was aimed to create an online planning algorithm, to allow multi-rotors to fly safely and efficiently in diverse changing environments.

In this work a real-time trajectory-planning algorithm was developed, using a modified RRT algorithm and a trajectory optimization algorithm. An enhancement step was also implemented, which allows a quick improvement of the trajectory generated by the RRT. The trajectory optimization algorithm allows generating locally-optimal trajectories, once it considers a cost function that is a combination of energy consumption and trajectory time. The developed algorithm was further integrated with a feature that stops the multi-rotor in critical scenarios until a feasible trajectory is found. A simplified testing framework was developed to access the capabilities of the developed algorithms. The real-time path-planner showed to be capable of creating collision-free trajectories in partially unknown environments by quickly adjusting the trajectory when new obstacles are detected. To validate that the algorithm is fit for a variety of maps, the planner was tested in random maps, where the obstacles are initially unknown and the information about their position is revealed only within a certain range. This algorithm is also capable of avoiding unexpected moving obstacles, considering also sensor delay.

Besides these dynamic capabilities, the developed trajectory planner allows selecting the amount of computational time spent in an initial optimization. This gives the algorithm anytime capabilities, allowing the user to choose between computational time and trajectory cost.

To validate that a real-multi-rotor is capable of following the computed trajectories, a simulation in Gazebo was performed. In this simulation, the virtual UAV was capable of following the trajectory while keeping a position error smaller than one meter at all times, in an aggressive manoeuvre.

The developed algorithms were then adapt to being more suitable for non-segregated air-space. This integration includes the possibility of computing trajectories that promote the flight at a certain flight level and the integration with a simplified TCAS system. The algorithm showed to be able to follow the resolutions provided by the TCAS system. This interesting feature shows how the algorithms can follow protocols designed for human pilots.

The algorithms at the time are only capable of considering spheres and cuboids as obstacles. For future work, an extension of the algorithm will be made to provide a better abstraction regarding the

environment representation. This extension will allow the planner to function correctly using external environment representations, which can be derived from sensor data, such as LIDAR data. It would also be interesting to test different formulations for the optimization problem, in an attempt to further improve the computational speed of the algorithm. Regarding the implementation, migrating the algorithms to C++ might significantly improve the real-time performance of the planner, opening the possibility to run the code in an on-board processor.

Further testing will be performed on physical vehicles. The algorithm will be tested on multi-rotors and, afterward, in fixed-wing aircraft. The planner can be extended to fixed-wing aircraft by, for instance, limiting the minimum speed (stall speed) and the climb rate of the aircraft.

References

[1] M. Correa, J. C. Jr., M. A. Rossi, and J. R. A. Jr. Improving the resilience of UAV in nonsegregated airspace using multiagent paradigm. In *Second Brazilian Conference on Critical Embedded Systems*, 2012.

[2] S. Ramasamy, R. Sabatini, and A. Gardi. Avionics sensor fusion for small size unmanned aircraft sense-and-avoid. In *2014 IEEE Metrology for Aerospace (MetroAeroSpace)*, pages 271–276, May 2014.

[3] K. Bimbraw. Autonomous cars: Past, present and future. *12th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, July 2015.

[4] P. Angelov. *Sense and Avoid in UAS*. Wiley, 1^{st} edition, 2012. ISBN:978-0-470-97975-4.

[5] S. Ramasamy, R. Sabatini, A. Gardi, and J. Liu. LIDAR obstacle warning and avoidance system for unmanned aerial vehicle sense-and-avoid. *Aerospace Science and Technology*, 2016.

[6] N. Sariff and N. Buniyamin. An overview of autonomous mobile robot path planning algorithms. Number 4, pages 183 – 188, 07 2006. ISBN 978-1-4244-0526-8. doi: 10.1109/SCORED.2006.4339335.

[7] J. Kok, L. F. Gonzalez, and N. Kelson. Fpga implementation of an evolutionary algorithm for autonomous unmanned aerial vehicle on-board path planning. *IEEE Transactions on Evolutionary Computation, VOL. 17*, pages 272 – 281, April 2013.

[8] J. TISDALE, Z. KIM, and J. K. HEDRICK. Autonomous UAV path planning and estimation. *IEEE Robotics & Automation Magazine*, pages 35–42, June 2009.

[9] I. K. Nikolos, K. P. Valavanis, I. Senior Member, N. C. Tsourveloudis, and A. N. Kostaras. Evolutionary algorithm based offline/online path planner for uav navigation. *IEEE Transactions on Systems, Man, and Cybernetics — PART B: cybernetics, VOL. 33, NO. 6*, pages 818–912, December 2003.

[10] X. Zhang, J. Chen, B. Xin, and H. Fang. Online path planning for uav using an improved differential evolution algorithm. In *Proceedings of the 18th World Congress The International Federation of Automatic Control Milano (Italy)*, pages 6349–6354, August 2011.

[11] 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, 12 2018. doi: 10.1016/j.robot.2018.11.017.

[12] H. Oleynikova, M. Burri, Z. Taylor, J. Nieto, R. Siegwart, and E. Galceran. Continuous-time trajectory optimization for online UAV replanning. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, October 2016.

[13] Ellis Chernoff et al. *Comprehensive set of recommendations for sUAS regulatory development*. Small Unmaned Aircraft System Aviation Rulemaking Committee, Washington, DC, April 2009.

[14] Lozano-Perez. Spatial planning: A configuration space approach. *IEEE Transactions on Computers*, C-32(2):108–120, Feb 1983. doi: 10.1109/TC.1983.1676196.

[15] C.-N. Cho, J.-H. Kim, S.-D. Lee, and J.-B. Song. Collision detection and reaction on 7 dof service robot arm using residual observer. *Journal of Mechanical Science and Technology*, 26, 04 2012. doi: 10.1007/s12206-012-0230-0.

[16] L. Yang, J. Qi, D. Song, J. Xiao, J. Han, and Y. Xia. Survey of robot 3d path planning algorithms. *Journal of Control Science and Engineering*, 2016:1–22, 01 2016. doi: 10.1155/2016/7426913.

[17] D. Ferguson, M. Likhachev, and A. Stentz. A guide to heuristic-based path planning. *Proceedings of the international workshop on planning under uncertainty for autonomous systems, international conference on automated planning and scheduling (ICAPS)*, pages 9–18, 2005.

[18] B. Siciliano, L. Sciavicco, V. Luigi, and G. Oriolo. Trajectory planning. In *Robotics: Modelling, Planning and Control*, chapter 4. Springer Science and Business Media, 01 2011.

[19] S. Bradley, A. Hax, and T. Magnanti. *Applied Mathematical Programming*, chapter 1. Addison-Wesley Publishing Company, 1977. ISBN 9780201004649. URL `https://books.google.pt/books?id=MSWdWv3Gn5cC`.

[20] J. T. Betts. A survey of numerical methods for trajectory optimization. *Journal of Guidance, Control, and Dynamics*, 21, August 1998.

[21] A. Wächter and L. T. Biegler. On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming, mathematical programming, 2006.

[22] A. Wächter. *An Interior Point Algorithm for Large-Scale Nonlinear Optimization with Applications in Process Engineering*. PhD thesis, Carnegie Mellon University, 2002.

[23] A. V. Fiacco and G. P. McCormick. *Nonlinear programming: sequential unconstrained minimization techniques*, volume 4. Society for Industrial and Applied Mathematics, 1990.

[24] A. Richards and J. P. How. Aircraft trajectory planning with collision avoidance using mixed integer linear programming. *Proceedings of the American Control Conference*, pages 1936–1941, May 2002.

[25] S. Grzonka, G. Grisetti, and W. Burgard. A fully autonomous indoor quadrotor. *IEEE Transactions on Robotics*, 28:90–100, 02 2012.

[26] D. J. Webb and J. van den Berg. Kinodynamic RRT*: Optimal motion planning for systems with linear differential constraints, 2012.

[27] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa. CHOMP: Gradient optimization techniques for efficient motion planning. In *2009 IEEE International Conference on Robotics and Automation*, pages 489–494, May 2009. doi: 10.1109/ROBOT.2009.5152817.

[28] J. Schulman, Y. Duan, J. Ho, A. Lee, I. Awwal, H. Bradlow, J. Pan, S. Patil, K. Goldberg, and P. Abbeel. Motion planning with sequential convex optimization and convex collision checking. *The International Journal of Robotics Research*, 33:1251–1270, 08 2014. doi: 10.1177/0278364914528132.

[29] C. Park, J. Pan, and D. Manocha. ITOMP: Incremental trajectory optimization for real-time replanning in dynamic environments. *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling*, page 207–215, 2012.

[30] 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, May 2011. doi: 10.1109/ICRA.2011.5980409.

[31] J. von Löwis and J. Rudolph. Real-time trajectory generation for flat systems with constraints. In A. Zinober and D. Owens, editors, *Nonlinear and Adaptive Control*, pages 385–394, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-45802-9.

[32] C. Livadas, J. Lygeros, and N. A. Lynch. High-level modeling and analysis of tcas. In *Proceedings 20th IEEE Real-Time Systems Symposium*, pages 115–125, Dec 1999. doi: 10.1109/REAL.1999.818833.

[33] F. A. Administration. *Introduction to TCAS II*. Version 7.1. U.S. Department of transportation, February 2011.

[34] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Available at http://planning.cs.uiuc.edu/.

[35] 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*, volume 2, pages 995–1001 vol.2, April 2000. doi: 10.1109/ROBOT.2000.844730.

[36] S. M. LaValle and J. James J. Kuffner. Randomized kinodynamic planning. *The International Journal of Robotics Research*, 20(5):378–400, 2001. doi: 10.1177/02783640122067453. URL `https://doi.org/10.1177/02783640122067453`.

[37] K. Schilling, R. Hess, and F. Kempf. Trajectory planning for car-like robots using rapidly exploring random trees *. *3rd IFAC Symposium on Telematics Applications*, 2013.

[38] T. Siauw, J. Adam, M. Cunha, I.-C. Hsu, P. Abbeel, J. Pouliot, K. Goldberg, A. Garg, and S. Patil. An algorithm for computing customized 3d printed implants with curvature constrained channels for enhancing intracavitary brachytherapy radiation delivery. *IEEE International Conference on Automation Science and Engineering (CASE)*, 2013.

[39] J. L. Marins, T. M. Cabreira, K. S. Kappel, and P. R. Ferreira. A closed-form energy model for multi-rotors based on the dynamic of the movement. In *2018 VIII Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 256–261, Nov 2018. doi: 10.1109/SBESC.2018.00047.

[40] G. E., J. D., and K. S. A fast procedure for computing the distance between complex objects in three dimensional space. *IEEE Journal of Robotics and Automation*, 4(2):193–203, April 1988.

[41] V. den Bergen. Proximity queries and penetration depth computation on 3d game objects. *Proceedings of the game developers conference (GDC)*, page 20–23, 2007.

[42] D. Kraft. *A Software Package for Sequential Quadratic Programming*. Deutsche Forschungs- und Versuchsanstalt fur Luft- und Raumfahrt Koln: Forschungsbericht. Wiss. Berichtswesen d. DFVLR, 1988.

[43] C. Richter, A. Bry, and N. Roy. Polynomial trajectory planning for aggressive quadrotor flight in dense indoor environments. *Proceedings of the International Symposium on Robotics Research (ISRR)*, 2013.

[44] 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)*, volume 3, pages 2149–2154 vol.3, Sep. 2004. doi: 10.1109/IROS.2004.1389727.

[45] F. Furrer, M. Burri, M. Achtelik, and R. Siegwart. *Robot Operating System (ROS): The Complete Reference (Volume 1)*, chapter RotorS—A Modular Gazebo MAV Simulator Framework, pages 595–625. Springer International Publishing, Cham, 2016. ISBN 978-3-319-26054-9. doi: 10.1007/978-3-319-26054-9_23. URL `http://dx.doi.org/10.1007/978-3-319-26054-9_23`.

[46] T. Lee, M. Leok, , and N. H. McClamroch. Geometric tracking control of a quadrotor UAV on SE(3). *49th IEEE Conference on Decision and Control*, pages 5420–5425, December 2010.

Appendix A

Cost function, constraints and derivatives

In this appendix the partial derivatives of the cost function and the constraints are presented. It is required to compute these derivatives if it is desired to solve the optimization problem using classic gradient based solvers.

A.1 Cost function

A.1.1 Time component

Time cost corresponds to the total trajectory time:

$$f(x) = (N+1)\Delta t \tag{A.1}$$

Where N represent the number of states subjected to optimization. The gradient of this function is 0 for every entrance except for:

$$\frac{\delta f}{\delta \Delta T} = N + 1 \tag{A.2}$$

A.1.2 Energy consumption

Energy consumption relative to the work produced by the thrust forces is given by:

$$E_i = \begin{cases} \frac{m}{2}(\|\mathbf{v}_{i+1}\|^2 - \|\mathbf{v}_i\|^2) + mg(p_{i,z} - p_{i+1,z}) & \text{, if } \|\mathbf{v}_{i+1}\|^2 + 2gp_{i+1,z} > \|\mathbf{v}_i\|^2 + 2gp_{i,z} \\ 0 & \text{, otherwise} \end{cases} \tag{A.3}$$

In equation A.3 m represents the mass of the vehicle and g the gravitic acceleration. Notice that sometimes these expressions are wrong once $\mathbf{F}(t) \cdot \mathbf{v}(t)$ can change signal between two states, however

this will be ignored. The term M_i will now refer to the mechanical energy associated to the ith state:

$$M_i = 2\left(\frac{\|\mathbf{v}_{i+1}\|^2}{2} + gp_{i+1,z}\right)$$

Equation A.3 can now be re-written as:

$$E_i = \begin{cases} M_{i+1} - M_i & \text{, if } M_{i+1} > M_i \\ 0 & \text{, otherwise} \end{cases} \tag{A.4}$$

The total energy consumption will be given by:

$$E = E_s + \sum_{i=0}^{N-2} E_i + E_g$$

Where E_s represents the energy spent between the start state and the state 0, E_i represents the energy spent between state i and state i+1, and E_g represents the energy spent between the N-1 state and the goal state.

For the derivatives:

$$\frac{\partial E}{\partial v_{0,j}} = \begin{cases} mv_{0,j} & \text{,if } M_0 > M_1 \text{ and } M_0 > M_s \\ -mv_{0,j} & \text{,if } M_0 < M_1 \text{ and } M_0 < M_s \\ 0 & \text{, otherwise} \end{cases} \tag{A.5a}$$

$$\frac{\partial E}{\partial p_{0,z}} = \begin{cases} mg & \text{,if } M_0 > M_1 \text{ and } M_0 > M_s \\ -mg & \text{,if } M_0 < M_1 \text{ and } M_0 < M_s \\ 0 & \text{, otherwise} \end{cases} \tag{A.5b}$$

$$\frac{\partial E}{\partial v_{i,j}} = \begin{cases} mv_{i,j} & \text{,if } M_i > M_{i+1} \text{ and } M_i > M_{i-1} \\ -mv_{i,j} & \text{,if } M_i < M_{i+1} \text{ and } M_i < M_{i-1} \\ 0 & \text{, otherwise} \end{cases} \tag{A.5c}$$

$$\frac{\partial E}{\partial p_{i,z}} = \begin{cases} mg & \text{,if } M_i > M_{i+1} \text{ and } M_i > M_{i-1} \\ -mg & \text{,if } M_i < M_{i+1} \text{ and } M_i < M_{i-1} \\ 0 & \text{, otherwise} \end{cases} \tag{A.5d}$$

$$\frac{\partial E}{\partial v_{N-1,j}} = \begin{cases} mv_{N-1,j} & \text{,if } M_{N-1} > M_g \text{ and } M_{N-1} > M_{N-2} \\ -mv_{N-1,j} & \text{,if } M_{N-1} < M_g \text{ and } M_{N-1} < M_{N-2} \\ 0 & \text{, otherwise} \end{cases} \tag{A.5e}$$

$$\frac{\partial E}{\partial p_{N-1,z}} = \begin{cases} mg & \text{,if } M_{N-1} > M_g \text{ and } M_{N-1} > M_{N-2} \\ -mg & \text{,if } M_{N-1} < M_g \text{ and } M_{N-1} < M_{N-2} \\ 0 & \text{, otherwise} \end{cases} \tag{A.5f}$$

The energy consumption due to the drag forces is given by:

$$E = -\int_{t_i}^{t_{i+1}} \mathbf{D}(t) \cdot \|v(t)\| dt \Leftrightarrow$$

the problem was simplified, it was assumed that the aircraft travels in a straight line at constant speed between waypoints, the energy consumption due to drag is now given by:

$$E_D = k\|\mathbf{p}_{i+1} - \mathbf{p}_i\| \left\| \frac{\mathbf{v}_{i+1} + \mathbf{v}_i}{2} \right\|^2$$

The derivatives can now be taken:

$$\frac{\partial E_D}{\partial p_{0,j}} = \frac{p_{0,j} - p_{1,j}}{\|\mathbf{p}_1 - \mathbf{p}_0\|} + \frac{p_{0,j} - p_{s,j}}{\|\mathbf{p}_0 - \mathbf{p}_s\|} \tag{A.6a}$$

$$\frac{\partial E_D}{\partial p_{i,j}} = \frac{p_{i,j} - p_{i+1,j}}{\|\mathbf{p}_{i+1} - \mathbf{p}_i\|} + \frac{p_{i,j} - p_{i-1,j}}{\|\mathbf{p}_i - \mathbf{p}_{i-1}\|} \tag{A.6b}$$

$$\frac{\partial E_D}{\partial p_{N-1,j}} = \frac{p_{N-1,j} - p_{g,j}}{\|\mathbf{p}_g - \mathbf{p}_{N-1}\|} + \frac{p_{N-1,j} - p_{N-2,j}}{\|\mathbf{p}_{N-1} - \mathbf{p}_{N-2}\|} \tag{A.6c}$$

$$\frac{\partial E_D}{\partial v_{0,j}} = 2(v_{0,j} - v_{1,j}) + 2(v_{0,j} - v_{s,j}) \tag{A.6d}$$

$$\frac{\partial E_D}{\partial v_{i,j}} = 2(v_{i,j} - v_{i+1,j}) + 2(v_{i,j} - v_{i-1,j}) \tag{A.6e}$$

$$\frac{\partial E_D}{\partial v_{N-1,j}} = 2(v_{N-1,j} - v_{g,j}) + 2(v_{N-1,j} - v_{N-2,j}) \tag{A.6f}$$

Equations A.6b and A.6e apply for $i \in \{1, ..., N-2\}$. Equations A.6 apply for $j \in \{x, y, z\}$.

A.2 Kinematics

The position $\mathbf{p}_{i+1}$ should be given by:

$$\mathbf{p}_{i+1} = \mathbf{p}_i + \frac{\mathbf{v}_i + \mathbf{v}_{i+1}}{2} \Delta t$$

Writing this as an equity constraint in the form $f_K(x) = 0$ we have:

$$\mathbf{p}_{i+1} - \mathbf{p_i} - \frac{\mathbf{v}_{i+1} + \mathbf{v}_i}{2} \Delta t = 0 \Leftrightarrow f_K(x) = \mathbf{p}_{i+1} - \mathbf{p}_i - \frac{\mathbf{v}_i + \mathbf{v}_{i+1}}{\Delta t} \tag{A.7}$$

The kinematic constraints can be now written as scalar equations:

$$f_{K(s,j)} = p_{0,j} - p_{s,j} - \frac{v_{0,j} + v_{s,j}}{2} \Delta t \qquad , j \in \{x, y, z\} \tag{A.8a}$$

$$f_{K(i,j)} = p_{i+1,j} - p_{i,j} - \frac{v_{i+1,j} + v_{i,j}}{2} \Delta t \qquad , j \in \{x, y, z\} \quad i \in \{0, \cdots, N-2\} \tag{A.8b}$$

$$f_{K(g,j)} = p_{g,j} - p_{N-1,j} - \frac{v_{g,j} + v_{N-1,j}}{2}\Delta t \qquad , j \in \{x, y, z\} \tag{A.8c}$$

The constraints in equation A.8 represent a total of $3(N+1)$ constraints. It will now be presented the gradient of each of this constraints.

$$\frac{\delta f_{K(s,j)}}{\delta \Delta t} = -\frac{v_{0,j} + v_{s,j}}{2} \qquad , j \in \{x, y, z\} \tag{A.9aa}$$

$$\frac{\delta f_{K(s,j)}}{\delta p_{0,j}} = 1 \qquad , j \in \{x, y, z\} \tag{A.9ab}$$

$$\frac{\delta f_{K(s,j)}}{\delta v_{0,j}} = -\frac{\Delta t}{2} \qquad , j \in \{x, y, z\} \tag{A.9ac}$$

$$\frac{\delta f_{K(i,j)}}{\delta \Delta t} = -\frac{v_{i+1,j} + v_{i,j}}{2} \qquad , j \in \{x, y, z\} \quad i \in \{0, \cdots, N-2\} \tag{A.9ba}$$

$$\frac{\delta f_{K(i,j)}}{\delta p_{i,j}} = -1 \qquad , j \in \{x, y, z\} \quad i \in \{0, \cdots, N-2\} \tag{A.9bb}$$

$$\frac{\delta f_{K(i,j)}}{\delta p_{i+1,j}} = 1 \qquad , j \in \{x, y, z\} \quad i \in \{0, \cdots, N-2\} \tag{A.9bc}$$

$$\frac{\delta f_{K(i,j)}}{\delta v_{i,j}} = -\frac{\Delta t}{2} \qquad , j \in \{x, y, z\} \quad i \in \{0, \cdots, N-2\} \tag{A.9bd}$$

$$\frac{\delta f_{K(i,j)}}{\delta v_{i+1,j}} = -\frac{\Delta t}{2} \qquad , j \in \{x, y, z\} \quad i \in \{0, \cdots, N-2\} \tag{A.9be}$$

$$\frac{\delta f_{K(g,j)}}{\delta \Delta t} = -\frac{v_{g,j} + v_{N-1,j}}{2} \qquad , j \in \{x, y, z\} \tag{A.9ca}$$

$$\frac{\delta f_{K(g,j)}}{\delta p_{N-1,j}} = -1 \qquad , j \in \{x, y, z\} \tag{A.9cb}$$

$$\frac{\delta f_{K(g,j)}}{\delta v_{N-1,j}} = -\frac{\Delta t}{2} \qquad , j \in \{x, y, z\} \tag{A.9cc}$$

A.3 Maximum speed

It is now required to write the maximum speed constraint in the form: $f_S(x) \geq 0$. The maximum speed can be written as:

$$\|\mathbf{v}_i\| \leq v_{MAx}$$

Re-writting this constraint in the desired form we have:

$$f_{S(i)} = v_{MAX} - \|\mathbf{v_i}\| \geq 0 \qquad , i \in \{1, \cdots, N-1\} \tag{A.4}$$

Taking the derivative of this constraints:

$$\frac{\delta f_{S(i)}}{\delta v_{(i,j)}} = -\frac{v_{(i,j)}}{\|\mathbf{v_i}\|} \qquad ,j \in \{x,y,z\} \quad i \in \{1,\cdots,N-1\} \tag{A.5}$$

Another formulation for the maximum speed constraints is:

$$f_{S(i)} = v_{MAX}^2 - \|\mathbf{v_i}\|^2 \geq 0 \qquad ,i \in \{1,\cdots,N-1\} \tag{A.6}$$

Taking the partial derivatives for this constraint:

$$\frac{\delta f_{S(i)}}{\delta v_{(i,j)}} = -2v_{(i,j)} \qquad ,j \in \{x,y,z\} \quad i \in \{1,\cdots,N-1\} \tag{A.7}$$

A.4 Maximum acceleration

The acceleration between consecutive states is given by $\mathbf{a_i} = \frac{\mathbf{v}_{i+1}-\mathbf{v}_i}{\Delta t}$. It is intuitive that the maximum acceleration constraint can be written as:

$$\|\mathbf{a_i}\| \leq a_{MAX} \Leftrightarrow \frac{\|\mathbf{v}_{i+1}-\mathbf{v}_i\|}{\Delta t} \leq a_{MAX}$$

The constraint should have the form $f_A(x) \geq 0$, to keep the derivatives simple, the form chosen for this inequity to be written was:

$$f_A(x) = a_{MAX}\Delta t - \|\mathbf{v}_{i+1}-\mathbf{v}_i\| \geq 0 \tag{A.8}$$

This constraint is now written in the form of a series of scalar constraints:

$$f_{A(s)} = a_{MAX}\Delta t - \|\mathbf{v_0}-\mathbf{v_s}\| \tag{A.9a}$$

$$f_{A(i,j)} = a_{MAX}\Delta t - \|\mathbf{v_{i+1}}-\mathbf{v_i}\| \qquad ,i \in \{1,\cdots,N-2\} \tag{A.9b}$$

$$f_{A(g,j)} = a_{MAX}\Delta t - \|\mathbf{v_g}+\mathbf{v_{N-1}}\| \tag{A.9c}$$

Taking the partial derivatives of these constraints:

$$\frac{\partial f_{A(s)}}{\partial \Delta t} = a_{MAX} \tag{A.10aa}$$

$$\frac{\partial f_{A(s)}}{\partial v_{0,j}} = -\frac{v_{0,j}-v_{s,j}}{\|\mathbf{v}_{0,j}-\mathbf{v}_{s,j}\|} \qquad ,j \in \{x,y,z\} \tag{A.10ab}$$

$$\frac{\partial f_{A(i)}}{\partial \Delta t} = a_{MAX}, \qquad i \in \{1,\cdots,N-2\} \tag{A.10ba}$$

$$\frac{\partial f_{A(i)}}{\partial v_{i,j}} = \frac{v_{i+1,j}-v_{i,j}}{\|\mathbf{v}_{i+1,j}-\mathbf{v}_{i,j}\|} \qquad ,j \in \{x,y,z\} \quad i \in \{1,\cdots,N-2\} \tag{A.10bb}$$

$$\frac{\partial f_{A(i)}}{\partial v_{i+1,j}} = -\frac{v_{i+1,j}-v_{i,j}}{\|\mathbf{v}_{i+1,j}-\mathbf{v}_{i,j}\|} \qquad ,j \in \{x,y,z\} \quad i \in \{1,\cdots,N-2\} \tag{A.10bc}$$

$$\frac{\partial f_{A(g)}}{\partial \Delta t} = a_{MAX} \tag{A.10ca}$$

$$\frac{\partial f_{A(g)}}{\partial v_{N-1,j}} = \frac{v_{g,j} - v_{N-1,j}}{\|\mathbf{v}_{g,j} - \mathbf{v}_{N-1,j}\|} \qquad , j \in \{x, y, z\} \tag{A.10cb}$$

Alternatively, the maximum acceleration constraints can be re-written by squaring both sides of the original inequality.

$$f_{A(s)} = a_{MAX}^2 \Delta t^2 - \|\mathbf{v_0} - \mathbf{v_s}\|^2 \tag{A.4a}$$

$$f_{A(i,j)} = a_{MAX}^2 \Delta t^2 - \|\mathbf{v_{i+1}} - \mathbf{v_i}\|^2 \qquad , i \in \{1, \cdots, N-2\} \tag{A.4b}$$

$$f_{A(g,j)} = a_{MAX} \Delta t^2 - \|\mathbf{v_g} + \mathbf{v_{N-1}}\|^2 \tag{A.4c}$$

The partial derivatives for this alternative formulation:

$$\frac{\partial f_{A(s)}}{\partial \Delta t} = 2a_{MAX}^2 \Delta t \tag{A.5aa}$$

$$\frac{\partial f_{A(s)}}{\partial v_{0,j}} = -2(v_{0,j} - v_{s,j}) \qquad , j \in \{x, y, z\} \tag{A.5ab}$$

$$\frac{\partial f_{A(i)}}{\partial \Delta t} = 2a_{MAX}^2 \Delta t, \qquad i \in \{1, \cdots, N-2\} \tag{A.5ba}$$

$$\frac{\partial f_{A(i)}}{\partial v_{i,j}} = 2(v_{i+1,j} - v_{i,j}) \qquad , j \in \{x, y, z\} \quad i \in \{1, \cdots, N-2\} \tag{A.5bb}$$

$$\frac{\partial f_{A(i)}}{\partial v_{i+1,j}} = -2(v_{i+1,j} - v_{i,j}) \qquad , j \in \{x, y, z\} \quad i \in \{1, \cdots, N-2\} \tag{A.5bc}$$

$$\frac{\partial f_{A(g)}}{\partial \Delta t} = 2a_{MAX}^2 \Delta t \tag{A.5ca}$$

$$\frac{\partial f_{A(g)}}{\partial v_{N-1,j}} = 2(v_{g,j} - v_{N-1,j}) \qquad , j \in \{x, y, z\} \tag{A.5cb}$$

A.5 Obstacle clearance

The inequality that assures that the UAV is at least d_{safe} away from any obstacle is $s(\mathcal{O}_k, \mathbf{p}_i) \geq d_{safe}$, writing the constraint in the desired form, let K represent the number of obstacles, then:

$$f_{O(i,k)} = s(\mathcal{O}_k, \mathbf{p}_i) - d_{safe} \geq 0 \qquad , i \in \{1, \cdots, N-1\} \quad k \in \{1, \cdots, K-1\} \tag{A.4}$$

A.5.1 Spheres

For a sphere the signed distance is given by:

$$s(\mathcal{O}_k, \mathbf{p}_i) = \|\mathbf{p}_i - \mathbf{c_k}(t)\| - r_k(t) \tag{A.5}$$

Where $\mathbf{c_k}(t)$ represents the center of the sphere $\mathcal{O}_k$ and $r_k(t)$ its radius. Taking the derivatives we obtain:

$$\frac{\partial f_{O(i,k)}}{\partial p_(i,j)} = \frac{p_{(i,j)} - c_{(k,j)}}{\|\mathbf{p_i} - \mathbf{c}_k\|} \tag{A.6a}$$

$$\frac{\partial f_{O(i,k)}}{\partial \Delta t} = -(i+1)\mathbf{v}_{ck} \cdot \frac{\mathbf{p}_i - \mathbf{c}_k}{\|\mathbf{p_i} - \mathbf{c}_k\|} - v_{rk} \tag{A.6b}$$

Where $\mathbf{v}_{ck}$ corresponds to the speed of the center of the sphere $\mathcal{O}_k$ and v_{rk} is $\frac{dr_k}{dt}$. The reader should keep in mind that i corresponds to the index of the state in the trajectory.

A.5.2 Sataic cuboids alligned with referential

It is desirable to have the signed distance implemented for this type of obstacles in an efficient way. It will now be described how this can be done. If $\mathbf{p_i}$ is a generic point and $\mathcal{O}_k$ a cuboid with its edges aligned with the world referential then let $\mathbf{o}_{k,i}$ be the closest point the boundary of the obstacle $\mathcal{O}_k$ to the point $\mathbf{p_i}$ and $\mathbf{o'}_{k,i}$ be the closest point the **inside** the obstacle $\mathcal{O}_k$ to the point $\mathbf{p_i}$. Let also $p_{(i,j)}$, $o_{(k,i,j)}$ and $o'_{(k,i,j)}$ be the *j*th component of the points $\mathbf{p}_i$, $\mathbf{o}_{(k,i)}$ and $\mathbf{o'}_{(k,i)}$ respectively. Let also x_{min}, y_{min}, z_{min}, x_{max}, y_{max} and z_{max} define the cuboid $\mathcal{O}_k$ (as being the minimum and maximum components x,y,z for a point belonging to the obstacle).

It is than straight forward to define the point $\mathbf{o'}_{(k,i)}$. (notice that if $\mathbf{p}_i$ is inside $\mathcal{O}_k$ than $\mathbf{o'}_{(k,i,j)} = \mathbf{p}_i$)

$$o'_{(k,i,j)} = \begin{cases} j_{min}, & if \quad p_{(i,j)} < j_{min} \\ p_{(i,j)}, & if \quad j_{min} < p_{(i,j)} < j_{max} \\ j_{max}, & if \quad p_{(i,j)} > j_{max} \end{cases}, \qquad j = \{x, y, z\} \tag{A.7}$$

If the point is outside the obstacle then the closest point in the cuboid can fall in:

- A vertex, if $j_{min} < p_{(i,j)} < j_{max}$ does not occur for any component.
- An edge, if $j_{min} < p_{(i,j)} < j_{max}$ occurs for only one component.
- A face if $j_{min} < p_{(i,j)} < j_{max}$ occurs for two components.

If $j_{min} < p_{(i,j)} < j_{max}$ is true for the tree components (x,y,z) then the point is inside the obstacle. The derivatives of the signed distance can be taken in the following way:

- If the closes point lie on a vertex then the derivatives are taken assuming a static closest point in the obstacle boundary.
- If the closest point lie on an edge the previous gradient is projected in the plane orthogonal to that edge.
- If the closest point lie on a face the previous gradient is projected in direction orthogonal to that face.