Single and multi-agent motion planning for multirotors at high speeds

Charbel Toumieh

université PARIS-SACLAY

Single and multi-agent motion planning for multirotors at high speeds

Planification de mouvement mono et multi-agents pour multirotors à haute vitesse

Thèse de doctorat de l'université Paris-Saclay

École doctorale n°580 : sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat: Robotique
Graduate School : Informatique et sciences du numérique, Référent : Faculté des sciences d'Orsay

Thèse préparée dans le Laboratoire interdisciplinaire des sciences du numérique (Université Paris-Saclay, CNRS), sous la direction de Alain LAMBERT, Maître de Conférences, Université Paris-Saclay

Thèse soutenue à Paris-Saclay, le 27 Septembre 2022, par

Charbel TOUMIEH

Composition du jury

Samia Bouchafa-Bruneau Professeure des Universités, IBISC, Univ Evry, Université Paris-Saclay	Présidente
David Filliat Professeur des Universités, U2IS - ENSTA Paris	Rapporteur & Examinateur
Tarek Hamel Professeur des Universités, I3S-CNRS, Institut Universitaire de France, Université Côte d'Azur	Rapporteur & Examinateur
Alain Lambert Maître de Conférences, LISN - Université Paris-Saclay	Directeur de thèse

THESE DE DOCTORAT
NNT : 2022UPASG072

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre: Planification de mouvement mono et multi-agents pour multirotors à haute vitesse
Mots clés: robotique autonome, planification de mouvement, multirotors, couloirs sûrs

Résumé: La navigation autonome des drones aériens a de nombreuses applications réelles qui peuvent rendre certaines tâches plus rapides et plus efficaces, telles que la recherche et le sauvetage. L'approche principale consiste à diviser le problème de la navigation autonome en sous-problèmes et à essayer de les résoudre de manière optimale. Ces sous-problèmes sont généralement considérés comme étant la perception (localisation et cartographie), la planification et le contrôle. Dans ce travail, nous abordons certains des sous-problèmes qui constituent des goulots d'étranglement du vol rapide et agile de la navigation autonome par drone. Nous mettons l'accent sur l'adaptation de nos algorithmes aux systèmes embarqués à faible puissance de calcul. Notre travail se scinde en 4 parties. La première partie concerne un nouvel algorithme de planification hors ligne, en environnment cartographié et statique, qui bat toutes les méthodes de l'état de l'art en termes de génération de trajectoire optimale en temps pour les multirotors. La deuxième partie traite de la cartographie et étudie les limites de l'utilisation d'un GPU pour transformer une carte de nuages de points générés par des capteurs en une grille de voxels. L'accent est mis sur la génération de la grille de voxels dans le temps de calcul le plus court possible pour la rendre adaptée aux systèmes embarqués à faible puissance de calcul. La troisième partie aborde, en partant d'une grille de voxel, le problème de la génération de couloirs sûrs qui sont utilisés dans les méthodes de planification de l'état de l'art pour planifier des trajectoires sûres et réalisables. Dans notre travail sur les couloirs sûrs, nous améliorons l'état de l'art en termes de sécurité, tout en restant dans les contraintes strictes des systèmes à faible puissance de calcul. La quatrième et dernière partie utilise nos travaux sur les couloirs sûrs et propose un nouveau cadre de planification améliorant l'état de l'art de la planification multirotor dans un environnement statique/dynamique pour la planification mono/multi-agent.

Title: Single and multi-agent motion planning for multirotors at high speeds
Keywords: autonomous robotics, motion planning, multirotors, safe corridors

Abstract: Autonomous navigation of aerial drones has many real-world applications that can make some tasks faster and more efficient, such as search and rescue. The main approach is to divide the problem of autonomous navigation into subproblems and try to solve them optimally. These subproblems are usually considered to be perception (localization and mapping), planning and control. In this work, we address some of these subproblems that are bottlenecks of fast and agile flight of autonomous drone navigation. We focus on making our algorithms suitable for low compute embedded systems. Our work can be divided into 4 parts. The first part presents a new offline planning algorithm in a mapped and static environment that beats all state-of-the-art methods in terms of time optimal trajectory generation for quadrotors. The second part addresses mapping and studies the limits of using a GPU to transform the point-cloud output of sensors into a voxel grid. The focus is on generating the voxel grid in the lowest computation time possible to make it suitable for low compute embedded systems. The third part (using voxel grids) tackles the problem of generating Safe Corridors that are used in state-of-the-art planning methods to plan safe and feasible trajectories. In our work on Safe Corridors we improve on the state-of-the-art in terms of safety, while remaining within the hard constraints of low compute systems. The fourth and final part uses our work on Safe Corridors and presents a new planning framework to improve on the state-of-the-art of multirotor planning in a static/dynamic environment for single/multi-agent planning.

ACKNOWLEDGMENTS

I would like to thank my PhD supervisor *Alain LAMBERT* who has been more than just a supervisor during my years as a PhD candidate. He has helped me navigate hard problems in research and personal life alike. I am extremely thankful for the opportunity he granted me by accepting me as his student, and the doors that his decision opened for me in my life as a researcher.

I would also like to thank my family for their emotional and financial support during my PhD. My mother, father, brother and sister have been an infinite source of unconditional empathy and love despite the dire economic and social conditions that my country (Lebanon) is going through.

In addition, I would like to thank the *Laboratoire Interdisciplinaire des Sciences du Numérique* (LISN) that welcomed me and helped me achieve my PhD. It provided the necessary hardware and software licenses, as well as an environment for fruitful and ambitious discussions.

Finally, I would like to thank the doctoral school *Sciences and Technologies of Information and Communication* (STIC) of University Paris-Saclay for funding my research and allowing me to work on the subjects I am most passionate about.

REMERCIEMENTS

Je tiens à remercier mon directeur de thèse *Alain LAMBERT* qui a été plus qu'un simple encadrant durant mes années de doctorat. Il m'a aidé à surmonter des problèmes difficiles dans la recherche et dans la vie personnelle. Je lui suis extrêmement reconnaissant de l'opportunité qu'il m'a offerte en m'acceptant comme doctorant et des portes que sa décision m'a ouvertes dans ma vie de chercheur.

Je tiens également à remercier ma famille pour leur soutien affectif et financier durant mon doctorat. Ma mère, mon père, mon frère et ma sœur ont été une source infinie d'empathie et d'amour inconditionnels malgré les conditions économiques et sociales désastreuses que traverse mon pays (le Liban).

Par ailleurs, je tiens à remercier le *Laboratoire Interdisciplinaire des Sciences du Numérique* (LISN) qui m'a accueilli et m'a aidé à réaliser ma thèse. Il a fourni le matériel et les licenses logicielles nécessaires, ainsi qu'un environnement pour des discussions fructueuses et ambitieuses.

Enfin, je tiens à remercier l'école doctorale *Sciences et Technologies de l'Information et de la Communication* (STIC) de l'Université Paris-Saclay pour avoir financé mes recherches et m'avoir permis de travailler sur les sujets qui me passionnent le plus.

Contents

ACKNOWLEDGMENTS 3

REMERCIEMENTS 5

1 INTRODUCTION 11
- 1.1 Motivation . . . 12
- 1.2 Context . . . 13
  - 1.2.1 Perception . . . 13
  - 1.2.2 Planning . . . 14
  - 1.2.3 Control . . . 16
- 1.3 Related Work . . . 17
  - 1.3.1 Drone racing . . . 17
  - 1.3.2 Voxel grids . . . 18
  - 1.3.3 Safe Corridors . . . 19
  - 1.3.4 Single-agent planning . . . 21
  - 1.3.5 Multi-agent planning . . . 23

2 MULTIROTOR MODEL 27
- 2.1 Base Model . . . 27
- 2.2 Controlled Model . . . 30
- 2.3 Aerodynamic Drag . . . 31

3 DRONE RACING 33
- 3.1 Introduction . . . 33
  - 3.1.1 Related work . . . 33
  - 3.1.2 Contribution . . . 35
- 3.2 MAV Model . . . 35
- 3.3 Optimal Control Problem . . . 37
  - 3.3.1 Continuous formulation . . . 37
  - 3.3.2 Obstacles/Path constraints . . . 38
  - 3.3.3 Discrete formulation . . . 38
- 3.4 Algorithm . . . 40
  - 3.4.1 Finding initial node index with the distance heuristic . . . 40
  - 3.4.2 Discrete gradient descent . . . 42
- 3.5 Simulation Results . . . 43
  - 3.5.1 Controller design . . . 43
  - 3.5.2 Trajectory generation parameters . . . 46
  - 3.5.3 Results and comparisons . . . 46
  - 3.5.4 Tracking performance . . . 47

3.5.5 Game of Drones competition - Microsoft, Neurips 2019 . . . 47
3.6 Limitations and Challenges . . . 48
3.7 Conclusion and Future Works . . . 49

4 VOXEL GRIDS **51**
4.1 Introduction . . . 51
4.1.1 Related work . . . 51
4.1.2 Contribution . . . 52
4.2 Nomenclature . . . 52
4.3 GPU Architecture . . . 52
4.4 The Method . . . 55
4.4.1 Voxel grid setup . . . 55
4.4.2 Populate occupied voxels . . . 57
4.4.3 Ray-trace to free voxels . . . 58
4.4.4 Update the local voxel grid . . . 61
4.4.5 Shift the local voxel grid . . . 61
4.5 Simulation Results . . . 62
4.5.1 Populate occupied voxels - simulation . . . 62
4.5.2 Ray-tracing to free voxels - simulation . . . 64
4.5.3 Update the local voxel grid - simulation . . . 67
4.5.4 Comparison with mit-acl-mapping . . . 67
4.6 Conclusion . . . 68

5 SAFE CORRIDORS **69**
5.1 Introduction . . . 69
5.1.1 Related work . . . 70
5.1.2 Contribution . . . 72
5.2 Environment and Definitions . . . 72
5.3 Convex Polyhedron Generation . . . 75
5.3.1 Overview . . . 75
5.3.2 Expanding a border . . . 79
5.3.3 Adapting the new border candidate and the corners . . . 80
5.4 The Improved Method . . . 80
5.4.1 Polyhedra volume heuristic . . . 84
5.4.2 Newly initialized corners . . . 85
5.5 Generating Safe Corridors . . . 86
5.6 Simulation Results . . . 87
5.6.1 Simulation setup . . . 87
5.6.2 Simulation results . . . 89
5.6.3 Case study . . . 90
5.7 Conclusion . . . 91

6 SINGLE-AGENT PLANNING - STATIC ENVIRONMENT 95
6.1 Introduction . . . 95
6.1.1 Related work . . . 95
6.1.2 Contribution . . . 96
6.2 MAV Model . . . 97
6.3 The Method . . . 99
6.3.1 Generating a global path . . . 101
6.3.2 Creating a Safe Corridor around the global path . . . 102
6.3.3 Generating a safe local reference trajectory . . . 103
6.3.4 Solving the MIQP/MPC problem . . . 104
6.4 Simulation Results . . . 106
6.4.1 Controller design . . . 107
6.4.2 Voxel grid generation . . . 107
6.4.3 Planner parameters . . . 107
6.4.4 Comparison with the state-of-the-art . . . 108
6.5 Conclusion . . . 111

7 SINGLE-AGENT PLANNING - DYNAMIC ENVIRONMENT 113
7.1 Introduction . . . 113
7.1.1 Related work . . . 113
7.1.2 Contribution . . . 114
7.2 Agent Model . . . 114
7.3 The Planner . . . 115
7.3.1 Generating the Temporal Safe Corridor . . . 115
7.3.2 Generating the reference trajectory . . . 116
7.3.3 Solving the MIQP/MPC problem . . . 118
7.4 Limitations . . . 120
7.5 Simulation . . . 120
7.5.1 Simulation environment . . . 120
7.5.2 Planner parameters . . . 121
7.5.3 Simulation results . . . 121
7.6 Conclusion . . . 122

8 MULTI-AGENT PLANNING - STATIC ENVIRONMENT 125
8.1 INTRODUCTION . . . 125
8.1.1 Problem statement . . . 125
8.1.2 Related work . . . 126
8.1.3 Contribution . . . 127
8.2 Assumptions . . . 127
8.3 Agent Model and Definitions . . . 128
8.4 The planner . . . 129
8.4.1 Generating a global path . . . 129
8.4.2 Creating a Safe Corridor around the global path . . . 130

8.4.3 Generating a time-aware Safe Corridor . . . 130
8.4.4 Generating a local reference trajectory . . . 133
8.4.5 Solving the MIQP/MPC problem . . . 136
8.4.6 Communication between agents . . . 138
8.5 Simulation Results . . . 139
8.5.1 Planner parameters . . . 139
8.5.2 Comparison with the state-of-the-art . . . 140
8.6 Conclusions and Future Works . . . 143

9 MULTI-AGENT EXPLORATION - STATIC ENVIRONMENT 145
9.1 Introduction . . . 145
9.1.1 Related work . . . 145
9.1.2 Contribution . . . 146
9.1.3 Assumptions . . . 146
9.2 Agent Model . . . 147
9.3 The Framework . . . 147
9.3.1 Local module - run on each agent . . . 148
9.3.2 Global module - run on a central hub . . . 149
9.4 Simulation . . . 151
9.4.1 Mapping parameters . . . 152
9.4.2 Planner parameters . . . 152
9.4.3 Controller parameters . . . 154
9.4.4 Simulation results . . . 154
9.5 Conclusion . . . 155

10 CONCLUSION 157
10.1 Work Summary . . . 157
10.2 Challenges and Future Directions . . . 158
10.2.1 Dynamic environments . . . 158
10.2.2 Stereo/Multiview matching . . . 159
10.2.3 Going through Narrow Gaps . . . 160
10.2.4 Machine Learning . . . 160

1 - INTRODUCTION

In this manuscript, we present our work which tackles some bottlenecks in high speed autonomous flight of multirotors. It is divided into 4 main parts: **Introduction** (chapter 1 - page 11), a preliminary on the **Multirotor Model** used in our thesis (chapter 2 - page 27), **Contributions** (chapters 3 - page 33 to 9 - page 145) and **Conclusion** (chapter 10 - page 157). In the Introduction chapter, we start by stating the motivation behind this work, then we give a global overview of the autonomous navigation literature of multirotors in the Context subsection. Finally, in the Related Works subsection, we present the related state-of-the-art works and indicate how our work improves on them.

The works presented in this manuscript have produced the following papers:

Journal

- C. Toumieh, A. Lambert, *Near Time-Optimal Trajectory Generation for Multirotors using Numerical Optimization and Safe Corridor*, Journal of Intelligent & Robotic Systems - Springer, vol. 105, no. 1, pp. 1-10, 2022, https://doi.org/10.1007/s10846-022-01625-0 [153]

- C. Toumieh, A. Lambert, *Voxel-Grid Based Convex Decomposition of 3D Space for Safe Corridor Generation*, Journal of Intelligent & Robotic Systems - Springer, vol. 105, no. 4, pp. 1-13, 2022, https://doi.org/10.1007/s10846-022-01708-y [149]

- C. Toumieh, A. Lambert, *Multi-Agent Planning using Decentralized Model Predictive Control and Time-Aware Safe Corridors*, IEEE Robotics and Automation Letters (RAL), vol. 7, no. 4, pp. 11110-11117, Oct. 2022, https://doi.org/10.1109/LRA.2022.3196777 [152]

Conference

- C. Toumieh, A. Lambert, *High-Speed Planning in Unknown Environments for Multirotors Considering Drag*, IEEE International Conference on Robotics and Automation (ICRA), 2021, pp. 7844–7850. https://doi.org/10.1109/ICRA48506.2021.9560773 [148]

Arxiv

- C. Toumieh, A. Lambert, *GPU accelerated voxel grid generation for fast mav exploration* [147]

- C. Toumieh, A. Lambert, *Shape-Aware Safe Corridors Generation for Robot Motion Planning* [154]
- C. Toumieh, A. Lambert, *MACE: Multi-Agent Autonomous Collaborative Exploration of Unknown Environments* [151]
- C. Toumieh, A. Lambert, *Multirotor Planning in Dynamic Environments using Temporal Safe Corridors* [150]

In addition, our work on drone racing allowed us to win an international competition on autonomous drone racing organized at NeurIPS 2019 by Microsoft, Stanford and ETH Zurich [104]. We won the tier 1 part of the competition which focused on planning and control.

1.1 . Motivation

The use of autonomous drones for commercial and industrial applications is gaining track, with more than 320'000 registered commercial drones in the US, [1] and a global drone service market expected to grow from $4.4B in 2018 to more than $60B by 2025 as per Business Insider [109], and to over $80B as per Forbes [4]. Multirotors are some of the most maneuverable and agile aerial drones [76] [156]. Their use have disrupted many industries such as cinematography, warehouse inventory tracking, search and rescue, agriculture, and remote sensing. The ability to use them to their full potential by exploiting their full dynamics and agility in combination with robust autonomy is of tremendous benefits for some applications such as agriculture and cinematography, and of crucial importance in some others such as search and rescue.

In the past few years, autonomous navigation for multirotors have been developed and offered in multiple consumer products [32] [139], however, they still underperform compared to human pilots in terms of speed, agility, and robustness. With drone racing becoming more and more popular [80], the difference between autonomous navigation and human piloting capabilities became more apparent. Furthermore, there are economically and socially beneficial applications such as cinematography, warehouse inventory tracking, package delivery, infrastructure inspection, and search and rescue that require a certain level of robust and fast autonomy that the state-of-the-art lacks. This thesis presents novel planning and mapping methods for multirotors to improve on the state-of-the-art of autonomous navigation for aerial drones, multirotors in particular.

1.2 . Context

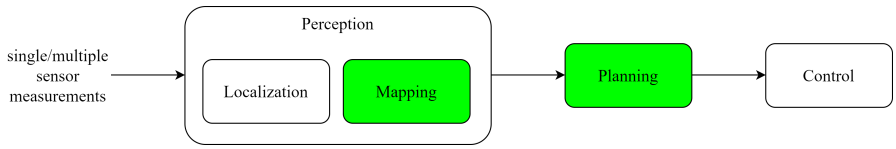


Figure 1.1: Classical autonomy pipeline of an autonomous drone. The modules to which we contribute are shown in green.

Our work tackles some of the bottlenecks in safety, robustness and speed in autonomous flight, reducing the gap between human pilots and autonomous multirotors, and in some cases, surpassing human performance. Barring an end-to-end machine learning approach for autonomous navigation, the main approach is to divide the autonomous navigation problem into subproblems and try to solve them each optimally. These subproblems are classically considered to be perception (localization and mapping), planning and control (Figure 1.1). We improve on the state-of-the-art in the mapping and planning sections for single/multi-agent planning.

1.2.1 . Perception

The autonomy pipeline starts with reading sensor measurements. There are many types of sensors that are used in autonomous systems such as lidars, cameras and radars [166]. The most common combination for aerial drones is cameras with inertial measurement units (IMU) [75] [139], as they are lightweight and do not considerably affect the drone's weight and agility. The outputs of these sensors are fused in the perception module to provide an estimation of the position of the drone while also mapping the surroundings (generally represented as a pointcloud). The fusion of these two steps is called simultaneous localization and mapping (SLAM) and has been the subject of many research works recently [21] [73] [77] [176]. However, in high speed flight situations, the localization step of the perception module lacked accuracy in comparison with low speed flight, which pushed the research community to create many datasets and localization/SLAM competitions that motivate and push researchers to improve in this area [30] [6] [159]. These competitions helped with the emergence of Visual Inertial Odometry (VIO) algorithms that are more and more accurate and robust at high speeds [50] [124] [125] [107] [123].

Furthermore, since the most convenient sensor to mount on a drone is a camera, using it to generate a dense pointcloud representation of the environment (mapping) is the common approach. Often, multiple cameras are mounted and techniques such as stereo-matching [143] [140] [162] [34] [140] and multi-view stereo [160] [165] [160] [168] are used to transform 2D images into 3D pointclouds. The use of multiple cameras adds significant robustness compared with

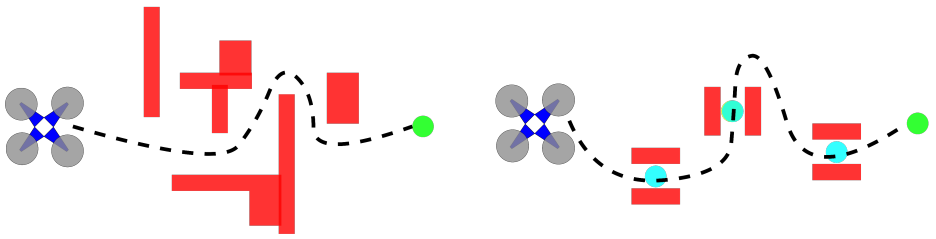
monocular approaches, hence its use on commercial autonomous drones [139] [32]. These approaches use machine learning techniques since they outperform classical depth estimation methods.

After generating a pointcloud representation of the environment, the mapping submodule is also responsible for transforming this representation into one that is useful for the planning module. The most simplistic and efficient of these representations are voxel grids [62] [134]. Other common representations are signed distance fields [117] [55], kd-trees [133] and octrees [64] [33]. When the environment is dynamic i.e. we have moving obstacles/agents, some other representations and methods are used to encode the variation of the environment over time and predict its state at future times such as dynamic occupancy grids [66] [63] [137]. These dynamic/temporal representations are then used by the planning module to avoid both static and dynamic obstacles.

1.2.2 . Planning

In general, there are two planning tasks that the state-of-the-art tackles (Figures 1.2a, 1.2b). The first is navigation to a given goal while exploring the environment and avoiding obstacles [146] [170] [88] [87]. This approach is useful in search and rescue [136] and exploration applications [31]. The second is navigating to a goal while being forced to go through intermediate waypoints or "wayspaces" (volumes in which the trajectory must pass) [101] [41]. This approach is useful in drone racing [106], and delivery and transportation applications [126]. To each of these tasks there can be single or multiple metrics that are optimized at the same time such as trajectory time and smoothness. For most applications, both trajectory time and smoothness are optimized for, putting different weights on each metric depending on the application. In some particular applications such as drone racing, we only care about time optimality.

Furthermore, these planning approaches can be divided in two categories: continuous-time polynomial trajectories (Figure 1.2d) and discrete-time trajectories (Figure 1.2c). Continuous-time polynomial trajectories [98] [108] [101] use the multirotors's differential-flatness property to achieve high computational efficiency. However, constraining the states of the multirotor to be a polynomial function can lead to suboptimal trajectories, especially when optimizing for trajectory time. Time discretized trajectories, on the other hand, use dynamic discretization techniques such as Euler or Runge-Kutta to divide the trajectory into discrete points and then solve for these points while minimizing some cost function [49] [59] [7]. Some methods use continuous polynomials as a motion primitive in a search based planning scheme [88] [90]. However, searching the state-space that is discretized in time can result in high computational costs due to the curse of dimensionality [90]. This is why most methods search in space only for a feasible path (disregarding



(a) A.1: goal with exploration

(b) A.2: goal with waypoints

(c) B.1: discrete trajectory

(d) B.2: continuous trajectory

(e) C.1: single-agent planning

(f) C.2: multi-agent planning

(g) D.1: static environment

(h) D.2: dynamic environment

Figure 1.2: We show the different categories of planning.

time), then add time using optimization based methods [146] [88] [169].

Planning methods can also be divided into 2 additional categories: single-agent planning methods (Figure 1.2e) and multi-agent planning methods (Figure 1.2f). In single-agent planning, we only have control over one multirotor [146] [88] [170], whereas in multi-agent planning we assume control over multiple multirotors [144] [120]. Multi-agent planning can further be divided into 2 categories: centralized planning where there is a central system that computes the trajectories of all agents, then sends each trajectory to the corresponding agent to execute [87] [120]; and decentralized planning where each agent computes its own trajectory using information sent by other nearby agents [144] [96]. Most recent works on multi-agent planning fall in the decentralized planning category due to its ease of implementation and low computation time with respect to centralized planning.

Finally, planning methods make assumptions about the environment: some are designed to deal with static environments [146] [170] [101] (Figure 1.2g) where all obstacles are static, whereas others can deal with dynamic environments [144] [90] (Figure 1.2h) that contain both static and dynamic obstacles. Naturally, dealing with dynamic environments is considerably harder since it usually also requires to track and predict the moving obstacles trajectory, in addition to choosing an appropriate environment representation for planning.

1.2.3 . Control

Multirotors are underactuated and have nonlinear dynamics. While they are amongst the most maneuverable and agile robots, they remain highly unstable systems in need of feedback control. An ideal control system should be able to utilize the multirotor dynamics to its full ability while remaining within the actuation limits of the rotors. Furthermore, it is preferable if the control system is able to adapt to external disturbances and changes in the dynamics/model of the multirotor with minimal latency.

For some time, classical methods such as cascaded control schemes [61] [35], geometric control [81] [102] and linear quadratic regulators [127] [128] [40] were good control options due to their extremely low computation time, even if their performance was not optimal. However, recent advances in optimization solvers [44] [157] [43] have made Nonlinear Model Predictive Control (NMPC) solvers computationally efficient, while delivering optimal performance in multirotor control [71]. Furthermore, it is able to perform well when the multirotor is subject to outside disturbances and changes in its dynamics/model [57]. This has made NMPC a popular choice for multirotor control in general, and for autonomous drone racing in particular [75].

1.3 . Related Work

In the past few years, the autonomy and agility of multirotors have been the focus of many research works that tried to push the state-of-the-art in terms of speed and agility [60] [100] [51] [122] [139] [102] [93] [105] [135] [170] [83] [118] [8].

Furthermore, many competitions have been organized in order to push researchers to improve on the state-of-the-art of fast and agile flight and approach/beat human performance. The autonomous drone racing series that took place in the recent NeurIPS and IROS conferences [106] [75] [97] as well as the AlphaPilot challenge [52] [42] are some of the most notable examples of such competitions.

In this section, we provide an introduction to the subjects that we contributed to (mapping and planning) as well as a summary of the corresponding state-of-the-art. We also briefly indicate the improvements that our works bring to the state-of-the-art.

1.3.1 . Drone racing

The primary goal of drone racing is to navigate through waypoints/wayspaces inside a given number of gates as fast as possible (Figure 1.2b). In order to perform well, one needs to complete two subtasks: robust navigation through the gates without crashing and a fast laptime. The first subtask requires accurate and precise localization of the drone with respect to the gate as well as (re-)planning, which was tackled in many works with different approaches such as [75] [39] [82] [68] [74]. For the second subtask, many state-of-the-art methods have been proposed (both offline and real-time planning methods) each having its advantages and drawbacks, which we will explore in this section.

Many polynomial methods have been proposed by the state-of-the-art which primarily rely on the differential flatness of quadrotors [129] [101] [101] [102]. Differential flatness allows expressing all states and inputs of the quadrotor in terms of its position and yaw angle (flat outputs), and their derivatives. This allows them to simplify the planning problem by transforming the quadrotor dynamics to an integrator model. These methods generally use waypoints and connect them with polynomial functions. However, constraining the states of the multirotor (position, velocity, ...) to a polynomial function leads to suboptimal results. Furthermore, including nonlinear dynamics such as drag forces in polynomial approaches is not trivial which is why all polynomial methods do not account for these forces which are significant at high speeds.

Other methods discretize the state-space and use graph search to find the best

combination of motion primitives that minimize a certain cost that includes time [87] [90]. However, much like polynomial methods, this approach does not include nonlinear dynamics and the optimality of the solution depends on the granularity of the discretization.

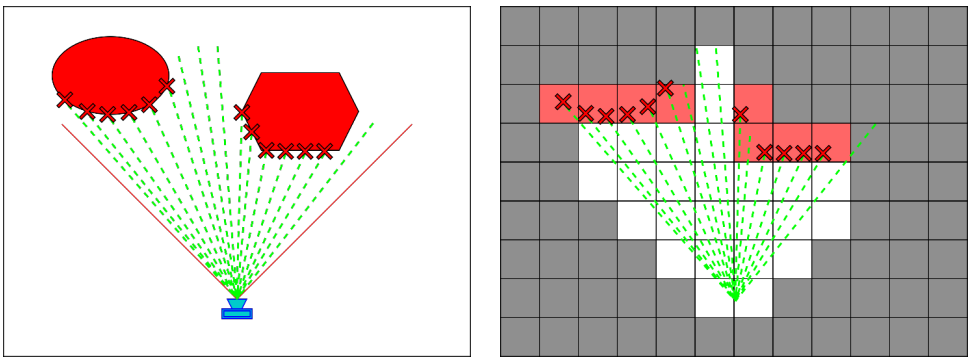
Finally, optimization based methods that discretize the whole trajectory and formulate the problem in a Model Predictive Control fashion [41] have the advantage of seamlessly including nonlinear dynamics that affect the multirotor such as gravity and drag forces. However, the work done in [41], while better than the rest of the literature, does not account for obstacle constraints. In addition, they only consider linear drag which is not a complete representation of drag forces which are quadratic in high speed flows [38].

In our work presented in chapter 3 (page 33), we tackle the problem of offline time optimal trajectory generation for multirotors in the context of drone racing. The planning framework that we created takes into account static obstacle constraints as well as nonlinear constraints pertaining to the multirotor dynamics. Prior to our work, no state-of-the-art method included those constraints, mainly due to the complexity and difficulties they add to the planning problem. This resulted in a considerable performance gap between us and the state-of-the-art. This was apparent in the drone racing competition *Game of Drones* [104] (organized by Microsoft, Stanford and ETH at *NeurIPS 2019*) that we participated in and won with a large margin between us and second place.

1.3.2 . Voxel grids

There are many environment representations that are used for planning such as signed distance fields [117] [55], octrees [64] [33], and voxel grids [134]. These methods transform the pointcloud output of autonomous navigation sensors such as lidars and IR sensors into an efficient intermediate representation that can be used for planning. Among these representations, voxel grids are the most simplistic and minimal in terms of information they provide, which results in a very low computation time. This made them popular among many state-of-the-art planning methods [146] [134].

A voxel grid is a regular grid composed of cells of equal dimensions. These cells are squares in 2D and cubes in 3D. Each cell is considered *occupied* if it has non empty intersection with an obstacle, *free* if it does not have an intersection, and *unknown* if the voxel hasn't been seen yet. After a sensor outputs a pointcloud by scanning for obstacles in its field of view, the *occupied* value is assigned to all the cells that contain any point of the pointcloud. Furthermore, cells that are in the field of view of the sensor that do do not contain any obstacle point and aren't occluded by an obstacle, are assigned the value of *free*. The cells are freed using



(a) Sensor measurement with obstacle points shown as red crosses.

(b) voxel grid corresponding to the sensor measurement.

Figure 1.3: Voxel grid generation using pointlcoud output of a sensor.

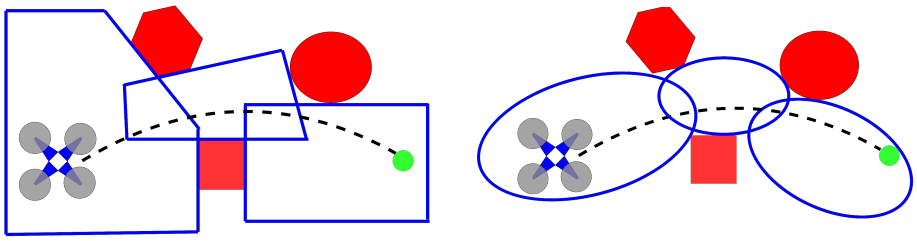
raytracing algorithms [19] [3]. All remaining cells that are outside the sensors' field of view are unchanged, so if they were *unknown* before the last measurement, they remain unknown (Figure 1.3).

In our work on voxel grids in chapter 4 (page 51), we investigated the use of a Graphical Processing Unit (GPU) to generate a voxel grid from a pointcloud, and compared it to a CPU-only implementation. We showed that the use of GPU accelerated the voxel grid generation process and reduced computation time especially in the raytracing section of the algorithm. The reduction in computation time allows to reduce the latency in the planning pipeline and hence react faster to changes in the environment. This increases the maximum speed a multirotor can achieve without risking crashing.

1.3.3 . Safe Corridors

Safe Corridors are a series of overlapping convex shapes that cover only free space in the environment (Figure 1.4). Generally, these convex shapes are polyhedra [88] [29] or ellipses/spheres [47]. They are used in state-of-the-art planning algorithms as an intermediate and simplified representation of the environment that allows to generate safe and feasible trajectories [88] [146].

In general, Safe Corridors are generated around a given path between the initial position and the goal position. This path can be generated using different optimal pathfinding methods such as A* [132] or Jumping Point Search (JPS) [58]. JPS preserves the optimality of A* while potentially reducing the computation time by an order of magnitude, which made it the most popular choice [88] [146]. Once the path is found, one can use segments or sampled points of the path to generate convex shapes around them [88].



(a) Safe Corridor composed of polyhedra.

(b) Safe Corridor composed of ellipses.

Figure 1.4: We show Safe Corridor examples composed of polyhedra/ellipses that cover only free space.

The quality of a Safe Corridor can be judged according to multiple metrics. One metric is the average number of convex shapes required to go from the initial to the goal position. Having a lower number of convex shapes means each convex shape traverses on average more of the path to the goal. This means that a trajectory constrained inside a convex shape can travel more distance/have higher velocity. Another metric is the average number of constraints per convex shape. Every convex shape will impose constraints on the trajectory to be inside it. These constraints are generally formulated as inequalities in an optimization problem [146], and the more inequalities there are, the higher the solving time. Thus, the number of constraints per convex shape affects the computation time and the total latency of the planning pipeline. Another important metric is the volume covered by the Safe Corridor. A bigger volume allows more room for the planning multirotor to maneuver in, and consequently leads to higher velocity trajectories.

In our work on Safe Corridors (chapter 5 - page 69), we present a novel voxel grid based Safe Corridor generation algorithm to generate Safe Corridors around a given path in a voxel grid representation of the environment. Our work outperforms the state-of-the-art in terms of safety while remaining within the constraints of low compute embedded systems. It also outperforms the state-of-the-art in terms of average number of polyhedra in the Safe Corridor (lower than the state-of-the-art) and average number of constraints per polyhedron (lower than the state-of-the-art). Our work is then extended by another work to improve the quality of the Safe Corridor by sensing its surroundings to know the best Safe Corridor fit (shape-aware approach).

1.3.4 . Single-agent planning

In this section we will address real-time single-agent planning for multirotors (unlike drone racing in section 1.3.1 - page 17 where the planning is done offline). The presented planning paradigms can be used for multiple applications such as exploration, infrastructure inspection and search and rescue. We will divide single-agent multirotor planning in 2 sections: planning in static environments and planning in dynamic environments.

Static environment

The majority of the literature makes the assumption of a static environment since it is considerably easier to deal with than dynamic environments where the dynamic obstacles have to be tracked and their future positions predicted. We will present an overview of this literature in this section.

Many methods presented in the literature use the differential flatness property of quadrotors [101]. In general, these methods minimize the squared euclidean norm of a derivative of the position in order to generate smooth trajectories [101], [20], [129]. However, none of these methods take into account nonlinear dynamics such as drag forces, and they are not particularly designed to efficiently deal with a constantly changing environment in applications such as exploration.

Other planning methods use closed-form solutions/motion primitives and transform the problem into graph search in the state space [87], [89], [90], [169]. These methods do not account for nonlinearities in the dynamics and are computationally expensive especially when generating complex maneuvers around obstacles. This makes them unsuitable for real time planning in complex environments.

There are methods that take into account obstacles while solving for the optimal trajectory. They use Euclidean Signed Distance Fields (ESDF) that transform the environment into a voxel grid whose voxels contain the distance to the closest obstacle [116], [117], [55], [45]. The resulting optimization problem of these methods is nonconvex and can lead to local minima and subpar trajectory quality compared with search based methods.

Another set of methods takes Safe Corridors (presented in section 1.3.3 - page 19) and optimizes for trajectories inside them [88], [161], [47], [78], [131], [146]. These methods escape the local minima problem of planning methods that rely solely on optimization by using Safe Corridors (since Safe Corridors always connect the initial position to the goal). None of the state-of-the-art methods account for drag forces which become significant at high speeds and can lead to unfeasible trajectories.

In our work on single-agent planning in static environments (chapter 6 - page 95), we created an algorithm for single-agent multirotor planning in a static environment that takes into account drag forces to further guarantee feasibility at high speeds. The planning framework uses our work on Safe Corridors, and is more computationally efficient than similar state-of-the-art methods [146] while having better feasibility guarantees.

Dynamic environment

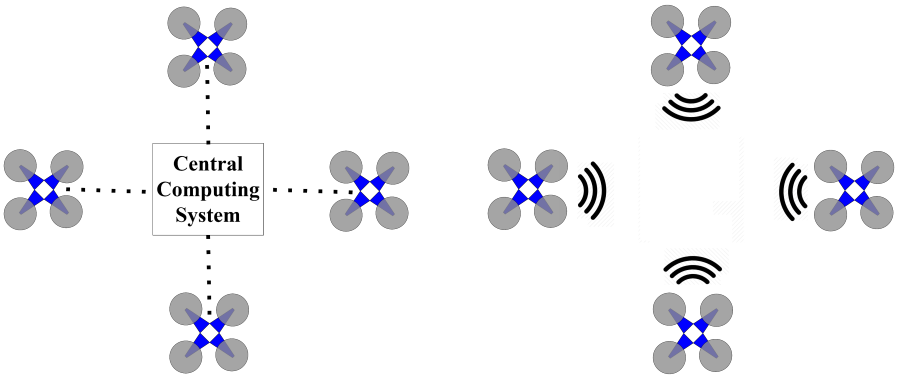
The literature on single-agent planning in dynamic environments is not as rich as the literature on static environments. However, recent works present different methods to deal with dynamic environments, each with its own approach and its own representation of dynamic environments.

A search based planning method have been proposed in [90]. The authors represent dynamic obstacles as linear velocity polyhedra, and use a voxel grid representation for static obstacles. Static obstacles collision check is done by sampling the generated trajectory and seeing if the sampled points are in an occupied voxel. Dynamic obstacles collision check is done by using their polyhedral representation and the polynomial property of the trajectory. However, much like in static environment planning, methods based on graph search in the state space lead to high computation times which makes them unsuitable for real-time applications.

Another method [85] models static and dynamic obstacles as ellipses, and adds them as constraints in a non-convex Model Predictive Control formulation. The generated trajectory is a series of discrete points that are outside all the ellipses representing the obstacles. However, decomposing all the environment into a series of ellipses is not trivial and can lead to high overhead computation time. Furthermore, if the environment is complex and contains a relatively high number of ellipses, the solving time of the optimization problem which now contains a large number of constraints becomes intractable for real-time applications.

Finally, a combination of a search based method and an optimization based method is presented in [144]. The result of the search based method is given as an initial solution for the optimization based method since the optimization problem is non-convex. The authors assume that all obstacles are given as a series of overlapping convex polyhedra, which, much like modeling obstacles as ellipses, is not trivial to generate and adds overhead to the planning pipeline.

In our work on single-agent planning in dynamic environments, we created an algorithm for single-agent planning in a dynamic environment. The presented approach is novel and presents some advantages over other state-of-the-art planning



(a) Centralized planning (b) Decentralized planning

Figure 1.5: We show the 2 types of multi-agent planning: centralized where the agents communicate with a Central Computing System that computes the trajectories of all agents; and decentralized where each agent computes its own trajectory and broadcasts it to nearby agents.

methods such as the environment representation that it uses for planning (temporal voxel grids). This representation of dynamic obstacles have gained traction especially in the autonomous driving domain [63], [66], [137].

1.3.5 . Multi-agent planning

Multi-agent planning is of tremendous benefits for applications such as exploration, search and rescue and agriculture. Thus, it has been the focus of many research works in the literature. Many approaches have been proposed, but in general multi-agent planning can be divided in 2 categories: centralized and decentralized planning (Figure 1.5). In centralized planning, all the trajectories of the agents are computed on a single computing entity/system that then sends each trajectory to the corresponding agent. In decentralized planning, all agents compute their trajectories on-board while also taking into account the trajectories generated by nearby agents. Decentralized planning methods have been the more popular approach recently due to their convenient implementation and low computation time.

Search based methods have been proposed for both centralized and decentralized planning in a static or dynamic environment [90]. However, much like the cases of single-agent planning, discretizing the state space and searching for a feasible path using motion primitives is computationally expensive, which makes this solution unsuitable for real-world applications.

Other methods model other planning agents and obstacles as spheres or ellipses

and include them as contraints in a Model Predictive Control (MPC) formulation [2], [70], [175], [85]. However, modeling the entire environment as a union of ellipses is not trivial and can lead to high solving times when the environment is complex.

Buffered voronoi cells have also been proposed for multi-agent collision avoidance [174], but they do not take into account static obstacles. Other approaches use seperating hyperplanes to avoid collision between agents and model other obstacles as ellipsoids to be added as constraints in a decentralized MPC formulation [96]. Much like other works relying on an ellipsoid decomposition of static obstacles, this method suffers from the fact that there is no trivial method to do such decompositions and the authors rely on human assisted decomposition instead.

Some methods have proposed the use of Safe Corridors [120], where the Safe Corridor is transformed into a Relative Safe Corridor that allows to mitigate inter-agent collisions. The generated trajectory is optimized sequentially in a centralized manner to generate collision free trajectories for all agents. While this method is computationally efficient for a low number of agents, it suffers the drawbacks of centralized planning methods which include a computation time that increases significantly every time we add a planning agent. Furthermore, the proposed method generates highly suboptimal trajectories for some agents in terms of trajectory length and time.

Finally, some methods use a combination of a search based approach and an optimization based approach in a decentralized manner [144]. They model all agents and static obstacles as a convex polyhedra and each agent takes the last generated trajectories of other agents to make sure it doesn't collide with them when generating its own trajectory. The problem is formulated as a nonconvex optimization problem that requires a good initial guess, which is why the authors use a search based method with motion primitives to find an initial good guess. This method is relatively computationally expensive since there is a search based part of the planning method that discretizes the time and the space which means the search is done in 4 dimensions. Furthermore, much like ellipsoid representation of the environments, generating a polyhedral representation is not trivial and adds considerable overhead to the planning pipeline.

In our work on multi-agent planning (chapter 8 - page 125), we created a framework for multi-agent planning in a static environment. Our work introduced a new concept which we call time-aware Safe Corridors which is inspired by Relative Safe Corridors in [120]. This new concept allows to avoid inter-agent collisions as well as static obstacles by adding the time-aware Safe Corridor constraints in a decentralized MPC formulation. The framework improves on the state-of-the-art in

multiple metrics such as traveled distance, trajectory smoothness and computation time. This work was extended/used in a subsequent project to explore a given volume in a static environment using multiple multirotors (chapter 9 - page 145).

2 - MULTIROTOR MODEL

In this section we will present the multirotor dynamics that we will be using throughout our works. We will use the quadorotor as an example of the multirotor. However, the final equations of motions are also applicable for drones with more than 4 rotors since they can be obtained using a similar derivation.

2.1 . Base Model

We derive the equations of motion of the quadrotor using the equations of motions of a rigid body. All vector and matrix symbols are in bold. The drone is composed of a transverse structure with a brushless motor (M_i) at the end of each end whose distance to the center is *l*. The structure is usually made of a lightweight material. Motors M_1 and M_3 rotate counterclockwise, while M_2 and M_4 rotate clockwise (Figure 2.1).

In order to represent the position and the attitude of the quadrotor, it is necessary to define a world frame $W = (\boldsymbol{x}_W, \boldsymbol{y}_W, \boldsymbol{z}_W)$ and a body frame $B = (\boldsymbol{x}_B, \boldsymbol{y}_B, \boldsymbol{z}_B)$. The origin of the local frame can be fixed any place on the surface of the Earth, whereas the origin of the body frame is the center of gravity of the quadrotor (Figure 2.1). The axes $\boldsymbol{x}_W$ and $\boldsymbol{y}_W$ point respectively towards the north and the east of the Earth, whereas the axis $\boldsymbol{z}_W$ is colinear with the vector of gravity but points in the opposite direction. The axis of the body frame $\boldsymbol{x}_B$ is between the M_1 et M_2 rotors, and the axis $\boldsymbol{y}_B$ is between M_2 et M_3. The axis $\boldsymbol{z}_B$ of the body frame can be obtain by a vector product between $\boldsymbol{x}_B$ and $\boldsymbol{y}_B$.

We define the following position and velocity vectors in the world frame:

$$\boldsymbol{p}_W = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad \text{et} \quad \boldsymbol{v}_W = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix}, \tag{2.1}$$

We define the attitude vector that represents the rotations around the axes of the world frame in the ZYX order :

$$\boldsymbol{\eta} = \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} \tag{2.2}$$

We define the rotation matrix that rotates a point in the body frame to the world frame, and consequently encodes the attitude of the quadrotor. It is formed by taking the unit vectors of the body frame expressed in the world frame, and

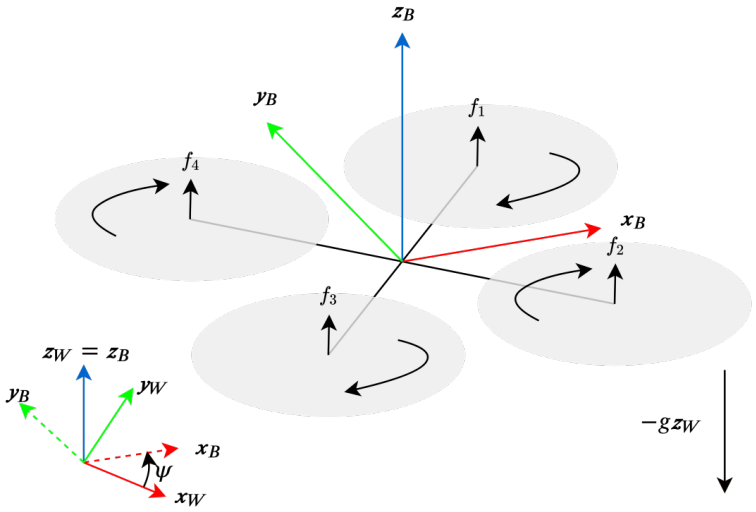


Figure 2.1: We show the world and body frames as well as the vector forces of each quadrotor. We also show the gravity direction and the rotation direction of the rotors.

then concatenating them horizontally:

$$\boldsymbol{R}_{WB} = [x_B \quad y_B \quad z_B]^T \tag{2.3}$$

We define the angular velocities in the body frame around x_B, y_B and z_B respectively:

$$\boldsymbol{\omega}_B = \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{2.4}$$

Furthermore, the rotation of the motor induces effects on the dynamics of the drone. We define k_f and k_τ as a constant thrust coefficient and a constant moment coefficient respectively that depend on the propeller used. The effects on the drone are a pushing force

$$f_i = k_f {\omega_i}^2, \quad i = 1, 2, 3, 4 \tag{2.5}$$

and a moment

$$\tau_i = k_\tau {\omega_i}^2, \quad i = 1, 2, 3, 4 \tag{2.6}$$

The sum of all the rotor forces gives a total thrust in the $\boldsymbol{z}_B$ direction of:

$$u = f_1 + f_2 + f_3 + f_4, \tag{2.7}$$

a roll torque (in the $\boldsymbol{x}_B$ axis)

$$\tau_x = l(f_1 - f_2 - f_3 + f_4), \tag{2.8}$$

a pitch torque (in the $\boldsymbol{y}_B$ axis)

$$\tau_y = l(f_1 + f_2 - f_3 - f_4), \tag{2.9}$$

and a yaw torque (in the $\boldsymbol{z}_B$ direction)

$$\tau_z = \tau_1 - \tau_2 + \tau_3 - \tau_4. \tag{2.10}$$

in the total system. Consequently, the quadrotor tilts in the positive roll direction when $\omega_2 + \omega_3 > \omega_1 + \omega_4$, in the positive pitch direction when $\omega_1 + \omega_2 > \omega_3 + \omega_4$, and in the positive yaw direction when $\omega_1 + \omega_3 > \omega_2 + \omega_4$.

The equations (2.7), (2.8), (2.9) and (2.10) can be written in the following matrix form:

$$\begin{bmatrix} u \\ \tau_x \\ \tau_y \\ \tau_z \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ k_a & -k_a & -k_a & k_a \\ k_a & k_a & -k_a & -k_a \\ k_c & -k_c & k_c & -kc \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{bmatrix}, \tag{2.11}$$

where $k_c = \frac{k_\tau}{k_f}$ et $k_a = l$

The modeling of the quadrotor dynamics is done using the equations of a rigid body, which are

$$m\dot{\boldsymbol{v}}_W = \boldsymbol{R}_{WB}\boldsymbol{F}_B + \boldsymbol{G}_W \tag{2.12}$$

$$\boldsymbol{I}\dot{\boldsymbol{\omega}}_B + \boldsymbol{\omega}_B \times \boldsymbol{I}\boldsymbol{\omega}_B = \boldsymbol{\tau}_B \tag{2.13}$$

where m the mass of the quadrotor, g the gravity and:

$$\boldsymbol{F}_B = \begin{bmatrix} 0 \\ 0 \\ u \end{bmatrix} \tag{2.14}$$

$$\boldsymbol{G}_W = \begin{bmatrix} 0 \\ 0 \\ -m.g \end{bmatrix} \tag{2.15}$$

$$\boldsymbol{\tau}_B = \begin{bmatrix} \tau_x \\ \tau_y \\ \tau_z \end{bmatrix} \tag{2.16}$$

$$\boldsymbol{I} = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \tag{2.17}$$

In conclusion the full dynamics of the quadrotor can be written as the following (we drop the subscript indicating the frame of reference in the rest of our work):

$$\dot{\boldsymbol{p}} = \boldsymbol{v} \tag{2.18}$$
$$\dot{\boldsymbol{v}} = \frac{u}{m}\boldsymbol{z}_B - g\boldsymbol{z}_W \tag{2.19}$$
$$\dot{\boldsymbol{R}} = \boldsymbol{R}\hat{\boldsymbol{\omega}} \tag{2.20}$$
$$\boldsymbol{I}\dot{\boldsymbol{\omega}} + \boldsymbol{\omega} \times \boldsymbol{I}\boldsymbol{\omega} = \boldsymbol{\tau} \tag{2.21}$$

where $\hat{\boldsymbol{\omega}}$ is the skew symmetric matrix of $\boldsymbol{\omega}$.

2.2 . Controlled Model

Using the cascaded approach presented in [15], we assume that the vehicle attitude $\boldsymbol{\eta}$ is controlled by low level attitude controller. We also assume that the controller has a low response time and allows for direct control of the attitude's rate of change $\dot{\boldsymbol{\eta}}$.

Furthermore, we assume that we can control directly the thrust magnitude u of the quadrotor since the motor dynamics are typically very fast [71].

We use the following state and control vectors:

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{p} \\ \boldsymbol{v} \\ \phi \\ \theta \\ \psi \end{bmatrix} \tag{2.22}$$

$$\boldsymbol{u} = \begin{bmatrix} c_{cmd} \\ \dot{\phi}_{cmd} \\ \dot{\theta}_{cmd} \\ \dot{\psi}_{cmd} \end{bmatrix} \tag{2.23}$$

The full dynamics are:

$$\dot{\boldsymbol{p}} = \boldsymbol{v} \tag{2.24}$$
$$\dot{\boldsymbol{v}} = \frac{c_{cmd}}{m}\boldsymbol{z}_B - g\boldsymbol{z}_W \tag{2.25}$$
$$\dot{\phi} = \dot{\phi}_{cmd} \tag{2.26}$$
$$\dot{\theta} = \dot{\theta}_{cmd} \tag{2.27}$$
$$\dot{\psi} = \dot{\psi}_{cmd} \tag{2.28}$$

These dynamics can also be obtained for a multirotor with an arbitrary number of rotors using the same approach presented in this chapter as demonstrated in [71].

2.3 . Aerodynamic Drag

The previously derived model is sufficient when navigating at relatively low speeds where aerodynamic drag is negligible. However in order to plan and control in high speeds, it is imperative to include the drag forces which increase as the speed increase.

Some works model the drag forces as linear with respect to the velocity of the quadrotor [119] [41] [35]. However, in this thesis we model the drag forces as quadratic with respect to the velocity since it is a more realistic representation of the drag that affects the quadrotor in a turbulent flow [38].

The final model which is the starting point of every work presented in this paper becomes:

$$\dot{\boldsymbol{p}} = \boldsymbol{v} \tag{2.29}$$

$$\dot{\boldsymbol{v}} = -g\boldsymbol{z}_W + \frac{c_{cmd}}{m}\boldsymbol{z}_B - \boldsymbol{R}\boldsymbol{D}\boldsymbol{R}^T\boldsymbol{v}||\boldsymbol{v}||_2 \tag{2.30}$$

$$\dot{\phi} = \dot{\phi}_{cmd} \tag{2.31}$$

$$\dot{\theta} = \dot{\theta}_{cmd} \tag{2.32}$$

$$\dot{\psi} = \dot{\psi}_{cmd} \tag{2.33}$$

Where the drag force is:

$$\boldsymbol{F}_{drag} = \boldsymbol{R}\boldsymbol{D}\boldsymbol{R}^T\boldsymbol{v}||\boldsymbol{v}||_2 \tag{2.34}$$

And the drag matrix $\boldsymbol{D}$ is diagonal and encodes the drag coefficients of the multirotor in the directions of its body frame $(\boldsymbol{x}_B, \boldsymbol{y}_B, \boldsymbol{z}_B)$.

3 - DRONE RACING

Trajectory generation is a fundamental problem for every type of robot. In most applications, the robots should reach their goals in the minimum time possible. Trajectory smoothness can also be optimized when trying to reach a given goal in addition to the time. Time-optimal and near time-optimal trajectory generation allows us to solve this type of problems. The generation of such trajectories for multirotors has gained traction with new applications in transport, delivery and search and rescue missions, as well as other applications in sports and entertainment such as drone racing.

The current state-of-the-art of multirotor trajectory generation is heavily based on polynomial methods and most methods choose a conservative approach when limiting the velocity or acceleration as a way to account for nonlinearities and guarantee feasibility, which limits time optimality and trajectory speed. We overcome this limitation by proposing a new formulation for multirotors trajectory generation that takes into account nonlinearities such as gravity and aerodynamic drag, It allows us to provide more time-optimal solutions then the state-of-the-art. We present an algorithm that uses our new formulation for near time-optimal trajectory generation for multirotors subject to obstacles/path constraints. We validate our approach using a state of the art simulator and compare it with other time-optimal trajectory generation methods.

3.1 . Introduction

Autonomous Micro-Aerial-Vehicals (MAVs) are gaining a lot of traction recently because of their agility and potential to complete tasks faster then humans while reducing the risks and costs. Researchers have previously used MAVs for infrastructure inspection [12], exploration tasks in unknown environment [13] and search and rescue missions [115]. Moreover, autonomous drone racing is gaining a growing attention recently with the introduction of competitions with substantial prizes [91], [104] with the aim to improve the state of the art. For that reason, many photo-realistic simulators have been developed to facilitate algorithm development and testing [138], [52]. In many of the aforementioned applications, planning time-optimal trajectories is crucial, which is the main motivation behind this work.

3.1.1 . Related work

In [61], the time-optimal trajectory is found by using the solution of the Hamiltonian minimum. In this approach, no obstacle constraints can be set. Furthermore, the maximum acceleration constraint is decoupled which leads to sub-optimal

solutions.

In [129] and [101] the authors use polynomials to generate minimum snap trajectories then find the time-optimal trajectory by adding time to the cost function and using gradient descent. The limitations of this approach is that the trajectories orders are fixed before the time optimization and may not be the ones that give the best time-optimal trajectory. Furthermore, in [129] feasibility is checked using a geometric controller that generates the applied thrusts, and checking if the thrusts hit the maximum allowed limit. This approach suffers from the fact that this controller does not have optimality guarantees (local or global) such as MPC.

In [87], the authors use linear quadratic minimum time control for motion primitive based planning. They however use the infinite norm of the acceleration to guarantee feasibility which leads to sub-optimal solutions.

In [46], the authors generate minimum-time piecewise polynomial trajectories for quadrotor flights. This suffers from the fact that the trajectory is constrained to be a polynomial which limits optimality. Furthermore, they limit the acceleration without including the effect of gravity which leads to sub-optimal solutions.

In [20], they use a polynomial method for trajectory generation, which limits optimality, since the states of the quadrotor are constrained to a polynomial function. They have soft constraints on the UAV states that can be violated.

In [11], the authors propose an analytical time and energy optimal trajectory generation method for MAVs. However, optimality is only given for exclusively horizontal movement and waypoints that have either direct line of sight or lie within an orthogonal environment.

In [141], the authors presented a strategy to address the minimum-time problem for quadrotors in constrained environments by generating a frame path, and expressing the quadrotor dynamics in a new set of coordinates (transverse coordinates) with respect to that path. However, they didn't consider drag forces in the quadrotor's model.

In all the previously cited works, drag forces are not included in the planning framework, which would naturally limit the maximum velocity the robot can attain. They instead use hard constraints which results in sub-optimal or unfeasible solutions. Furthermore, only [61] accounts for the gravity when limiting the acceleration but decouples the directions which also leads to sub-optimal solutions. In addition, all polynomial methods constrain the states of the robot to polynomial functions and require to specify waypoints which can limit the solution's optimality

when the waypoints are not specifically chosen to optimize trajectory time.

A recent work [41] was inspired by the work presented in this chapter (3) and presented a planning method that accounts for nonlinearities such as drag forces as well as gravity. However, the proposed framework takes into account only linear drag and does not account for quadratic drag (which is a more realistic representation of the drone dynamics as explained in chapter 2 - page 27). Furthermore, it does not take obstacles and path constraints in consideration and does not optimize the quadrotor's attitude to minimize drag.

3.1.2 . Contribution

We propose to overcome the aforementioned limitations of state-of-the-art planning methods by redesigning the problem and by proposing a new algorithm. There is no planning framework that accounts for all nonlinearities (drag and gravity) while also accounting for obstacle constraints and not constraining the states to a polynomial function. The main contributions of our work are:

- A novel formulation of the multirotor model for trajectory generation (section 3.2 - page 35) that takes into account all nonlinearites i.e. gravity and drag forces. The nonlinearities have not been considered by state-of-the-art works with the exception of [41] that takes into account linear drag rather than quadratic drag (quadratic drag is a more realistic representation of the real dynamics of the multirotor).

- A novel algorithm for near time-optimal trajectory generation for multirotors under obstacles/path constraints (section 3.4 - page 40). The algorithm uses a new heuristic that allows to transform the optimization problem from a Mixed Integer Nonlinear Program (MINLP) to an Nonlinear Program (NLP), and result in better convergence/solutions.

We also present simulation results[1] performed in Microsoft's AirSim simulator [138] that includes a state of the art physics engine that accurately simulates quadrotor dynamics and aerodynamic drag.

3.2 . MAV Model

In this section we will present the model of the multirotor that we use for trajectory generation/optimization. The derivation is mainly taken from [69]. We assume a low level controller that allows for controlling the attitude and thrust as presented in chapter 2 - page 27. We use the nomenclature defined in table 3.1.

[1] https://youtu.be/RIjiDV2woLU

Table 3.1: Nomenclature

g	gravity
m	multirotor mass
$\boldsymbol{p}$	position vector x, y, z in world frame
$\boldsymbol{v}$	velocity vector v_x, v_y, v_z in world frame
$\boldsymbol{a}$	modified acceleration vector
$\boldsymbol{j}$	control vector j_x, j_y, j_z in the world frame
$\boldsymbol{z}_W$	world frame z
$\boldsymbol{z}_B$	body frame z expressed in the world frame
$\boldsymbol{e}_3$	unit vector $[0\ 0\ 1]^T$
$\boldsymbol{R}$	rotation matrix from body to world frame
$\boldsymbol{D}$	drag matrix
ϕ	roll angle
θ	pitch angle
ψ	yaw angle
c_{cmd}	total thrust command
$\|\|.\|\|_2$	euclidean norm
d_0	drag coefficient in $\boldsymbol{x}_B$ and $\boldsymbol{y}_B$
d_1	drag coefficient in $\boldsymbol{z}_B$
$\boldsymbol{I_3}$	identity matrix of size 3

The equations of motion are:

$$\dot{\boldsymbol{p}} = \boldsymbol{v} \tag{3.1}$$

$$\dot{\boldsymbol{v}} = -g\boldsymbol{z}_W + \frac{c_{cmd}}{m}\boldsymbol{z}_B - \boldsymbol{R}\boldsymbol{D}\boldsymbol{R}^T\boldsymbol{v}||\boldsymbol{v}||_2 \tag{3.2}$$

$$\dot{\phi} = \dot{\phi}_{cmd} \tag{3.3}$$

$$\dot{\theta} = \dot{\theta}_{cmd} \tag{3.4}$$

$$\dot{\psi} = \dot{\psi}_{cmd} \tag{3.5}$$

Where the drag force is:

$$F_{drag} = \boldsymbol{R}\boldsymbol{D}\boldsymbol{R}^T\boldsymbol{v}||\boldsymbol{v}||_2 \tag{3.6}$$

We assume that the mutirotor is symmetrical around $\boldsymbol{z}_B$ and the drag coefficient is the same in $\boldsymbol{x}_B$ and $\boldsymbol{y}_B$. This gives us the following drag matrix:

$$\boldsymbol{D} = \begin{bmatrix} d_0 & 0 & 0 \\ 0 & d_0 & 0 \\ 0 & 0 & d_1 \end{bmatrix} \tag{3.7}$$

We then decompose $\boldsymbol{D}$ into the following using $\tilde{d_0} = d_1 - d_0$:

$$\begin{aligned}
\boldsymbol{D} &= d_0 \cdot (\boldsymbol{I_3} - \boldsymbol{e}_3 \cdot \boldsymbol{e}_3^T) + d_1 \cdot (\boldsymbol{e}_3 \cdot \boldsymbol{e}_3^T) - d_0 \cdot \boldsymbol{e}_3 \cdot \boldsymbol{e}_3^T + d_0 \cdot \boldsymbol{e}_3 \cdot \boldsymbol{e}_3^T && (3.8)\\
&= d_0 \cdot \boldsymbol{I_3} + (d_1 - d_0) \cdot \boldsymbol{e}_3 \cdot \boldsymbol{e}_3^T && (3.9)\\
&= d_0 \cdot \boldsymbol{I_3} + \tilde{d_0} \cdot \boldsymbol{e}_3 \cdot \boldsymbol{e}_3^T && (3.10)
\end{aligned}$$

We then replace the value of $\boldsymbol{D}$ in the equation 3.3:

$$\begin{aligned}
\dot{\boldsymbol{v}} &= -g\boldsymbol{z}_W + \frac{c_{cmd}}{m}\boldsymbol{z}_B - \boldsymbol{R}\boldsymbol{D}\boldsymbol{R}^T\boldsymbol{v}||\boldsymbol{v}||_2 && (3.11)\\
&= -g\boldsymbol{e}_3 + \frac{c_{cmd}}{m}\boldsymbol{R}\boldsymbol{e}_3 - \boldsymbol{R}(d_0 \cdot \boldsymbol{I_3} + \tilde{d_0} \cdot \boldsymbol{e}_3 \cdot \boldsymbol{e}_3^T)\boldsymbol{R}^T\boldsymbol{v}||\boldsymbol{v}||_2 && (3.12)\\
&= (\frac{c_{cmd}}{m} - C)\boldsymbol{R}\boldsymbol{e}_3 - g\boldsymbol{e}_3 - d_0\boldsymbol{v}||\boldsymbol{v}||_2 && (3.13)\\
&= \tilde{T}\boldsymbol{R}\boldsymbol{e}_3 - g\boldsymbol{e}_3 - d_0\boldsymbol{v}||\boldsymbol{v}||_2 && (3.14)
\end{aligned}$$

With:

$$\begin{aligned}
C &= \tilde{d_0} \cdot \boldsymbol{e}_3^T\boldsymbol{R}^T\boldsymbol{v}||\boldsymbol{v}||_2 && (3.15)\\
\tilde{T} &= \frac{c_{cmd}}{m} - C && (3.16)
\end{aligned}$$

Finally, we pose $\boldsymbol{a} = \tilde{T}\boldsymbol{R}\boldsymbol{e}_3 - g\boldsymbol{e}_3$ and we take as control input for the system $\boldsymbol{j}$ the derivative of $\boldsymbol{a}$ (not to be confused with the jerk of the system). The final system becomes:

$$\begin{aligned}
\dot{\boldsymbol{p}} &= \boldsymbol{v} && (3.17)\\
\dot{\boldsymbol{v}} &= \boldsymbol{a} - d_0\boldsymbol{v}||\boldsymbol{v}||_2 && (3.18)\\
\dot{\boldsymbol{a}} &= \boldsymbol{j} && (3.19)
\end{aligned}$$

3.3 . Optimal Control Problem

In this section, we will present the optimization problem that we use for trajectory generation. We will first introduce the continuous version of the time optimal control problem (section 3.3.1 - page 37) without accounting for obstacles. Then we will explain how to add obstacle constraints (section 3.3.2 - page 38). Finally we will derive the discrete version of the optimization problem that account for obstacles and that we solve to generate the near time-optimal trajectory.

3.3.1 . Continuous formulation

With $\boldsymbol{x} = [\boldsymbol{p}\ \boldsymbol{v}\ \boldsymbol{a}]^T$, $\boldsymbol{u} = \boldsymbol{j}$, $f(\boldsymbol{x}(t), \boldsymbol{u}(t))$ defined by Eq. (3.17), (3.18) and (3.19), T the total time of the trajectory, the Optimal Control Problem (OCP) is the following:

$$\underset{\boldsymbol{x}(.),\boldsymbol{u}(.),T}{\text{minimize}} \quad T \tag{3.20}$$

$$\text{subject to} \quad \dot{\boldsymbol{x}}(t) = f(\boldsymbol{x}(t), \boldsymbol{u}(t)), \quad \boldsymbol{x}(0) = \boldsymbol{x}_0 \tag{3.21}$$

$$\boldsymbol{x}(T) \in \boldsymbol{X}_T \tag{3.22}$$

$$\sqrt{a_x(t)^2 + a_y(t)^2 + (a_z(t) + g)^2} \leq a_{max} \tag{3.23}$$

$$a_z(t) \geq a_{z,min} \tag{3.24}$$

$$||\boldsymbol{u}(t)||_2 \leq j_{max} \tag{3.25}$$

The terminal state constraint $\boldsymbol{X}_T$ is a convex space. The upper limit a_{max} is determined from the maximum thrust of the rotors $c_{cmd,max}$ and the multirotor's mass m, and j_{max} is approximated from the rotors dynamics. We assume that the rotors provide thrust in only one direction:

$$a_{max} = \frac{c_{cmd,max}}{m} \tag{3.26}$$

$$a_{z,min} = -g \tag{3.27}$$

3.3.2 . Obstacles/Path constraints

The continuous formulation presented in the previous section assumes no obstacles are in the environment which is almost never the case. Thus, we need to add to the optimization problem some constraints that allow the generated trajectory to avoid obstacles. Obstacles can be be avoided by forming multiple safe convex spaces that are overlapping. This can be done by decomposing the space into overlapping polyhedra along a path [88]. However for the purpose of this work we will generate the overlapping polyhedra manually since the maps are known beforehand.

3.3.3 . Discrete formulation

The continuous optimization presented in section 3.3.1 (page - 37) is then discretized using Euler or Runge-Kutta 4th order. Note that We use the discretization step $h = \frac{T}{N}$ where N is the number of discretization steps. This means that the discretization step is dependent on a free optimization variable and hence the constraints of the dynamics will be nonlinear. A Nonlinear Program (NLP) is an optimization problem where some of the constraints or the objective function are nonlinear. Hence we have a NLP formulation. The Safe Corridor constraints which are used to avoid obstacles add binary integer variables to the optimization formulation. Hence, the final optimization problem is a Mixed Integer Nonlinear Program (MINLP). We add a small jerk (derivative of the acceleration) cost (the cost becomes Linear Quadratic Minimum Time - equation 3.28) along the path to make the convergence faster and the trajectory smoother.

$$\underset{\boldsymbol{x}_k, \boldsymbol{u}_k, b_{kp}, T}{\text{minimize}} \quad T + \frac{\epsilon}{N} \sum_{i=0}^{N-1} ||\boldsymbol{u}_k||_2^2 \tag{3.28}$$

subject to

$$\tilde{\boldsymbol{x}}_{k+1} = \tilde{f}_d(\tilde{\boldsymbol{x}}_k, \boldsymbol{u}_k), \quad k = 0 : N-1 \tag{3.29}$$

$$\boldsymbol{x}_0 = \boldsymbol{X}_0 \tag{3.30}$$

$$\boldsymbol{x}_N \in \boldsymbol{X}_T \tag{3.31}$$

$$\sqrt{a_{x,k}^2 + a_{y,k}^2 + (a_{z,k} + g)^2} \leq a_{max} \tag{3.32}$$

$$a_{z,k} \geq a_{z,min} \tag{3.33}$$

$$||\boldsymbol{u}_k||_2 \leq j_{max} \tag{3.34}$$

$$b_{kp} = 1 \implies \boldsymbol{A}_p \boldsymbol{p}_k \leq \boldsymbol{c}_p \tag{3.35}$$

$$\sum_{p=0}^{P-1} b_{kp} \geq 1 \tag{3.36}$$

$$b_{kp} \in \{0, 1\} \tag{3.37}$$

The jerk cost is multiplied by a small value ϵ in order to prioritize T and have a near time-optimal solution. It is also divided by N so that the number of points doesn't influence the cost. We choose the jerk cost such that it constitutes less than 5% of the total cost.

Note that we assume that we already know the model of the multirotor and the model constraints are added directly to the optimization problem which is fed to an off-the-shelf solver to solve. The model dynamics are included in the optimization formulation as discrete constraints (equations (3.29) to (3.34)).

We assume that the Safe Corridor that covers the free space is composed of P polyhedra with overlapping space between every two consecutive polyhedra. The P polyhedra are described by $\{(\boldsymbol{A}_p, \boldsymbol{c}_p)\}$, $p = 0 : P-1$ (Eqn. (3.35)). We introduce binary variables b_{kp} (P variables for each $\boldsymbol{x}_k$, $k = 0 : N-1$). We force all the points to be in at least one of the polyhedra with the constraint $\sum_{p=0}^{P-1} b_{kp} \geq 1$.

The constraints for obstacle avoidance (Safe Corridor - equations 3.35 to 3.37) cannot be directly included in the optimization problem because most MINLP solvers (IPOPT [158], BONMIN [17], KNITRO [112]) struggle to solve this problem and all fail to find a feasible solution from a cold start if $P > 5$. It is thus necessary to construct a new approach to solve this problem. Consequently, we designed an algorithm/heuristic to alleviate the problem and turn it from MINLP to NLP (which still includes the model constraints and dynamics).

3.4 . Algorithm

In this section we present a novel algorithm to solve the MINLP described in the previous section by turning it into an Nonlinear Program (NLP). Each discretized state $\boldsymbol{x_k}$ is called a node.

If the position of node k, $\boldsymbol{p}_k$, is in the overlapping space of two consecutive polyhedra $\{(\boldsymbol{A}_p, \boldsymbol{c}_p)\}$ and $\{(\boldsymbol{A}_{p+1}, \boldsymbol{c}_{p+1})\}$, this implies that all previous nodes from 0 to k will be in the polyhedra from 0 to p.

It follows that if $\boldsymbol{p}_{k_1}$ is in the overlapping space of two consecutive polyhedra $\{(\boldsymbol{A}_{p_1}, \boldsymbol{c}_{p_1})\}$ and $\{(\boldsymbol{A}_{p_1+1}, \boldsymbol{c}_{p_1+1})\}$, and $\boldsymbol{p}_{k_2}$ is in the overlapping space of two consecutive polyhedra $\{(\boldsymbol{A}_{p_2}, \boldsymbol{c}_{p_2})\}$ and $\{(\boldsymbol{A}_{p_2+1}, \boldsymbol{c}_{p_2+1})\}$, with $k_2 > k_1$ and $p_2 > p_1$, this implies that all nodes from k_1 to k_2 will be in the polyhedra from $p_1 + 1$ to p_2.

This means that by finding and fixing the optimal $P - 1$ nodes that belong in the $P - 1$ overlapped spaces of the polyhedra, we can deduce to which polyhedra all the other nodes belong i.e. b_{kp}. This allows us to move away from the MINLP formulation to an NLP formulation with the following algorithm.

We first start with two polyhedra, each adding N_{inc} nodes to the total number of nodes in the trajectory (without the initial node). We then optimize the trajectory. We then iteratively add one polyhedra (and N_{inc} nodes to the trajectory) and optimize the trajectory until we reach the terminal space. Optimizing the trajectory every time we add a polyhedra is composed of two steps:

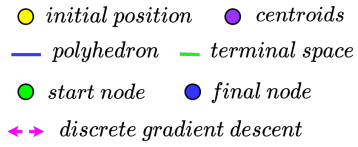
1. Find the initial node index in the overlapping space using the distance heuristic.

2. Use discrete gradient descent on the nodes in all the overlapping spaces between the polyhedra.

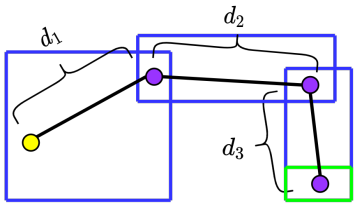
At the end of every iteration/optimization, we find the optimal nodes (or the indexes of the nodes) who should be in the overlapping spaces between the polyhedra as well as the optimal trajectory up to the last considered polyhedron. The terminal space at every iteration is the overlapping space between the last added polyhedra and the polyhedra to be added in the next iteration, until we reach the actual terminal space (Fig. 3.1).

3.4.1 . Finding initial node index with the distance heuristic

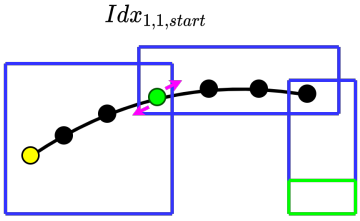
In this section, we present a heuristic that produces a initial guess for the node index that should be in the overlapping space between 2 consecutive polyhedra after adding a new polyhedron to the optimization problem. At every iteration,



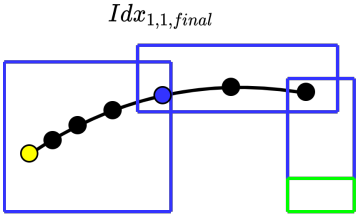
(a) legend



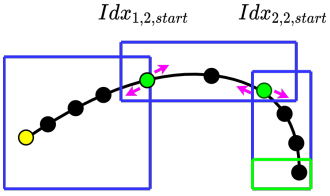
(b) centroids and distances used by the heuristic



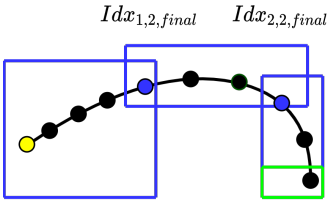
(c) Iteration 1: the index of the node in the overlapping space is fixed using the distance heuristic $Idx_{1,1,start} = N_{tot,1}\frac{d_1}{d_1+d_2} = 3$, with $N_{tot,1} = 1 + 2N_{inc} = 7$.



(d) Iteration 1: after running the discrete gradient descent on the node we end up with the optimal index $Idx_{1,1,final} = 4$.



(e) Iteration 2: $Idx_{2,2,start} = N_{tot,2}\frac{d_1+d_2}{d_1+d_2+d_3} = 6$, with $N_{tot,2} = 1 + 3N_{inc} = 10$, and $Idx_{1,2,start} = Idx_{1,1,final} * Idx_{2,2,start}/N_{tot,1}$.



(f) Iteration 2: after running the discrete gradient descent on both nodes until we converge to a local minimum, we obtain $Idx_{1,2,final}$ and $Idx_{2,2,final}$ as well as the optimal trajectory.

Figure 3.1: We show an example of our algorithm that solves the MINLP. It is running on a safe corridor composed of 3 polyhedra (we use rectangles for simplification) with the terminal space being a subspace of the last polyhedron. Every time we add a polyhedron, we also add $N_{inc} = 3$ nodes to the total nodes in the trajectory.

when we add a polyhedron, we first estimate the fraction of the total trajectory time that will be spent in this polyhedron. This fraction of time is the same as the fraction of nodes (from the total nodes) that will be inside the added polyhedron. In order to estimate the time, we use a distance heuristic.

We first find the centroids of all the overlapping spaces between the polyhedra (which are also polyhedra). We then calculate the distance between each 2 consecutive centroids. The sum of all distances is denoted d_{tot}. The distance between to centroids i and $i+1$ is denoted d_{i+1} (Fig. 3.1b). The estimated time spent in the polyhedra is thus calculated with the following heuristic $T_i = \frac{Td_i}{d_{tot}}$. This means that the estimated index of the node is $Idx_{i,i,start} = \frac{N_{tot,i}(d_{tot}-d_i)}{d_{tot}}$ with $N_{tot,i}$ the total number of nodes at iteration i. The first subscript in $Idx_{i,i,start}$ indicates the index of the overlapping space, the second the total number of overlapping spaces, and the third whether the index is obtained through the heuristic ($start$) or is the final index after the optimization ($final$). Since the index should be an integer, $Idx_{i,i,start}$ is rounded to the closest integer. This heuristic assumes that we traverse the distances with constant velocity.

Once we find the fraction of time T_i, we determine the remaining number of nodes that will be in the previous polyhedra $N_{rem} = Idx_{i,i,start}$, which is the same as the index of the node in the last overlapping space. We then recalculate the indexes of the nodes in the previous overlapping spaces $Idx_{j,i,start}$, $0 < j < i$, by using the results of the last optimization. The indexes are fixed such that the fraction of time spent in the each of the previous polyhedra is the same as the one that resulted from the last optimization (Fig. 3.1e).

3.4.2 . Discrete gradient descent

Fixing the indexes of the nodes in the overlapping spaces, allows us to fix all the binary variables b_{kp} in the MINLP described in section 3.2 (page 35), which turns it into an NLP.

After finding the start index of the node in the last overlapping space using the heuristic, and adjusting the other node indexes, we run discrete gradient descent on all the nodes in the overlapping spaces to find a locally optimal trajectory.

Running the discrete gradient descent on a node in an overlapping space consists of the following: we increase and decrease the index of the node and solve the new resulting NLPs, then see which direction minimizes the cost function. We then move in this direction, increasing/decreasing the node index until we reach a local minimum.

We first run the discrete gradient descent on the node in the last overlapping

space, while fixing all the node indexes in the other overlapping spaces according to the last optimization. After finding a local minimum, we then run discrete gradient descent on the second to last overlapping space while fixing all the others to find the local minimum. We continue this process of running gradient descent on nodes in overlapping spaces, moving from the last to the first overlapping space. After a full back-sweep i.e. running discrete gradient descent on all the nodes in all the overlapping spaces from last to first, we check whether there has been a change in the optimal node index in any of the overlapping spaces. If yes, we redo a full back-sweep of discrete gradient descent and check again. If no change occurs in the indexes i.e. all nodes are in a local minimum, the algorithm has converged.

3.5 . Simulation Results

We test our algorithm in the context of drone racing. For this purpose we adapt the algorithm presented in this work to become more specific for this use case. The gates are considered to be overlapping spaces (or a subspace of the overlapping spaces) to force the trajectory to pass through them. In some cases, the space between the gates is free (no obstacle constraints), and we don't need to constrain the trajectory inside a polyhedron, but only find the optimal node index inside the gate space (in the optimization this means the polyhedron is considered to be the euclidean space $\mathbb{R}^3$).

We use the *Building99_Hard* racing map from [104] (Fig. 3.3). This map has obstacle constraints between some of the gates which require us to create a safe corridor of overlapping polyhedra.

3.5.1 . Controller design

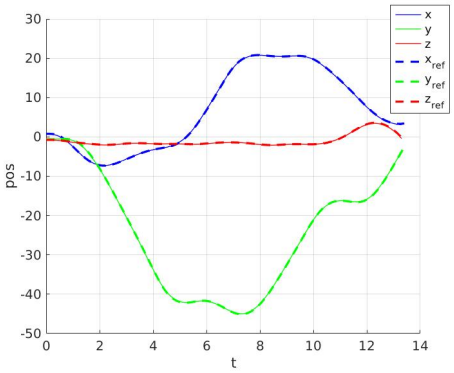
We control our quadrotor using a nonlinear MPC [71], with the ACADOS toolkit [157]. The MPC minimizes the cost function:

$$\begin{aligned} J = \int_{t=0}^{T} ||\boldsymbol{x}(t) - \boldsymbol{x}_{ref}(t)||^2_{\boldsymbol{Q}_x} + ||\boldsymbol{u}(t) - \boldsymbol{u}_{ref}(t)||^2_{\boldsymbol{R}_u} dt \\ + ||\boldsymbol{x}(T) - \boldsymbol{x}_{ref}(T)||^2_{\boldsymbol{P}} \end{aligned} \tag{3.38}$$

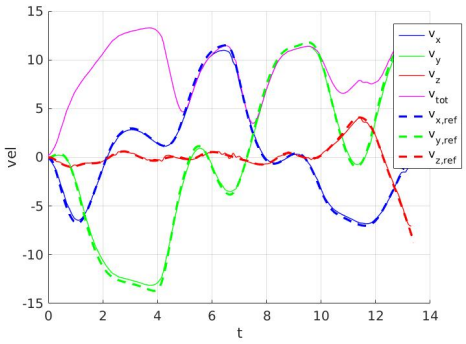
We use the model described in section 3.2 (page 35) with $\boldsymbol{u} = [c_{cmd}\ \dot{\phi}_{cmd}\ \dot{\theta}_{cmd}\ \dot{\psi}_{cmd}]^T$ and $\boldsymbol{x} = [\boldsymbol{p}\ \boldsymbol{v}\ \phi\ \theta\ \psi]^T$. The sampling time is $h = 0.05s$ and the horizon $N_h = 15$ which gives $T = 0.75s$. The weights are:

$$\boldsymbol{P} = \boldsymbol{Q}_x = diag(10, 10, 10, 0.1, 0.1, 0.1, 0, 0, 0.01) \tag{3.39}$$

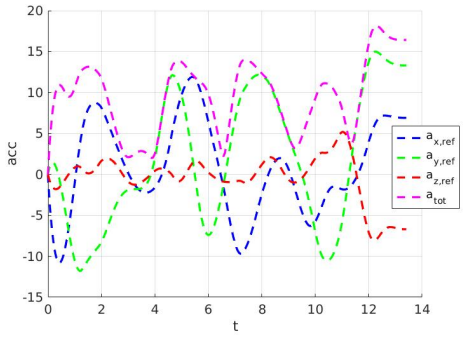
$$\boldsymbol{R}_u = diag(0, 0.05, 0.05, 0.05) \tag{3.40}$$



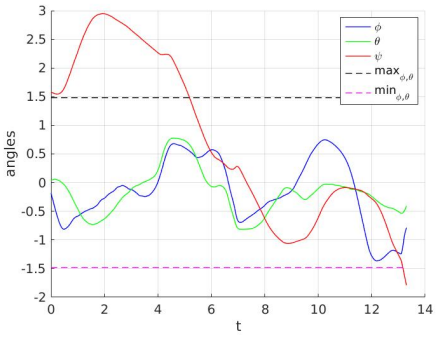
(a) real and reference position



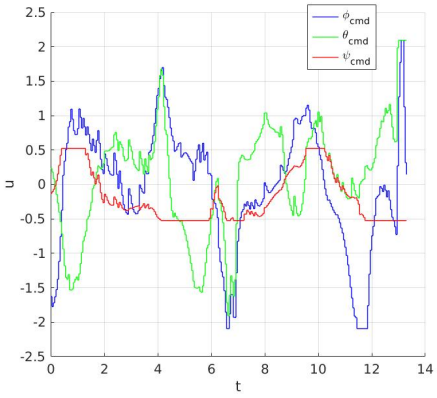
(b) real and reference velocity



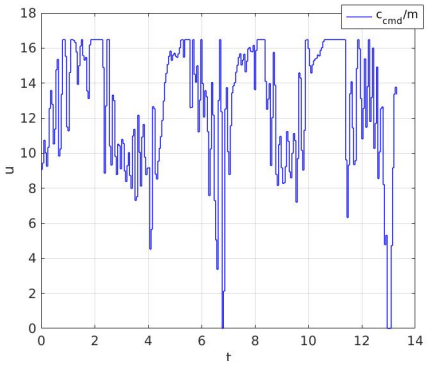
(c) reference acceleration



(d) roll, pitch and yaw



(e) roll, pitch and yaw rate commands



(f) acceleration command

Figure 3.2: Results of the simulation on *Building99_Hard*. We show the different states that result from the controller tracking as well as the reference states.

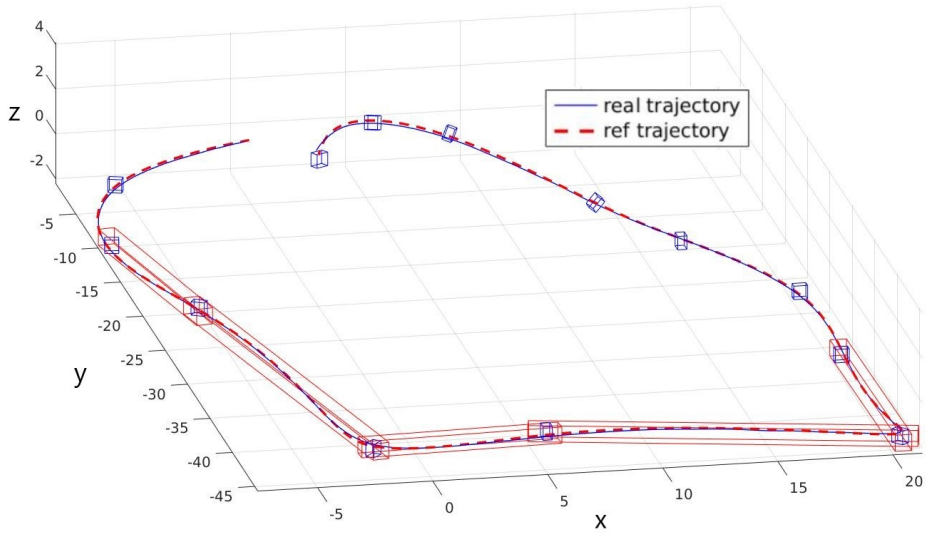


Figure 3.3: Reference and real 3d trajectory tracked by the quadrotor in Airsim on the map *Building99_Hard*. The gate constraints are the blue boxes and the obstacle constraints are the red boxes forming the safe corridor. Some gates do note require a safe corridor between them as there are no obstacles. The safe corridor is generated manually.

All parameters are set by approximation/experimentation and may not be optimal. They provide however some measurement to the feasibility of the generated trajectory. We limit $|\phi| \leq 85\ deg$, $\theta \leq 85\ deg$, $|\dot{\phi}_{cmd}| \leq 120\ deg/s$, $|\dot{\theta}_{cmd}| \leq 120\ deg/s$ and $|\dot{\psi}_{cmd}| \leq 60\ deg/s$.

3.5.2 . Trajectory generation parameters

In all generated trajectories, the solver we used for the NLP was KNITRO (interior/direct algorithm) with the CasADi interface [5]. We choose $a_{max} = 16.48 m/s^2$ which is the maximum acceleration the thrusts can achieve. We also choose $a_{z,min} = -g$, and $j_{max} = 60 m/s^3$. We don't put any constraints on the velocity since the drag forces will determine the maximum velocity we can reach.

We also choose the average number of nodes per polyhedron $N_{inc} = 20$ and the jerk cost $\epsilon = 0.005$. Naturally, we can decrease ϵ to get more time-optimal trajectories, however convergence would take more time, and beyond a certain order of magnitude of ϵ the improvement becomes negligible (less than 5%). Finally, we choose the drag matrix $\boldsymbol{D} = diag(0.01, 0.01, 0.14)$ (identified from the simulation).

3.5.3 . Results and comparisons

Table 3.2: Comparison between polynomial methods and our method on the track *Building99_Hard*. We denote in red the parameters that lead to unfeasible trajectories, and in bold the overall best result.

	Mellinger [101]	Burri [20]	Richter [129]	Our method
$a_{max} = 7\ m/s^2$ $v_{max} = 7\ m/s$	$49\ s$	$38.5\ s$	$47.5\ s$	-
$a_{max} = 17\ m/s^2$ $v_{max} = 30\ m/s$	$27\ s$	$16\ s$	$28\ s$	-
$a_{max} = 16.48\ m/s^2$	-	-	-	**13.5 s**

In Fig. 3.2 we show the reference states generated by our method as well as the real states tracked by the controller. In Fig. 3.3 we show the reference trajectory generated by our method as well as the real trajectory of the quadrotor. The maximum reached velocity is $13.5\ m/s$ and its average is $9\ m/s$. The maximum reference acceleration is $18\ m/s^2$ (table 3.2) and the average acceleration command is $12.72\ m/s^2$. Note that the reference acceleration shown is the acceleration vector defined in section 3.2 - page 35 (does not include drag forces) and **not** the derivative of the velocity (which includes the quadrotor's thrust, the gravity and the drag forces).

Note that the maximum acceleration applied on the quadrotor without adding the drag forces can reach $\sqrt{a_{max}^2 + g^2} = 19.18\ m/s^2$ which is the case where the

quadrotor is at full thrust with a roll/pitch at $90\ deg$.

In table 3.2 we show how our method performs compared to state-of-the-art polynomial methods. Our method outperforms these methods even when we allow them to break the dynamic constraints of the quadrotor. Note that the polynomial methods which we compare with have been designed for real time use while our method was designed for offline trajectory generation. However, the comparison serves to show the limitations of constraining a trajectory to a polynomial form and neglecting nonlinearities such as gravity and drag forces.

3.5.4 . Tracking performance

We evaluate the trajectory feasibility by evaluating the tracking performance of the controller. Note that an unfeasible trajectory can be tracked to within a certain margin of error by an ideal controller; however, the margin of error becomes larger as the trajectory becomes more unfeasible. We show in Tab. 3.3 the root mean squared errors (RMSE) $\overline{e}$ and absolute maximum errors e_{max} in all directions and in total.

Table 3.3: Tracking errors along all directions on *Building99_Hard* map. The values shown are in centimeters.

$\overline{e_x}$	$\overline{e_y}$	$\overline{e_z}$	$\overline{e_{tot}}$	$e_{x,max}$	$e_{y,max}$	$e_{z,max}$	$e_{tot,max}$
5.9	8.1	4.6	10.9	13.42	16.2	14.2	19.7

The total RMSE is $10.9\ cm$ and the absolute maximum error is $19.7\ cm$. Many factors contribute to tracking errors including but not limited to controller design, controller latency, and trajectory feasibility. Tracking performance can thus be increased by optimizing the controller parameters, or by reducing the bounds on agility and speed such as acceleration or jerk.

The errors shown indicate that our generated trajectory is feasible and safe if we were to inflate the obstacles by the maximum tracking error (without any further tuning of controller parameters or agility bounds).

3.5.5 . Game of Drones competition - Microsoft, Neurips 2019

The method presented in this work was used by our team (*Dédale*) in the *Game of Drones* autonomous drone racing competition (Tier 1) organized by Microsoft, Stanford and University of Zurich at NeurIPS 2019. The competition had 117 registered teams with 16 unique entries on the qualification leaderboard [97]. Our team won the Tier 1 competition which focused on planning and control. We

traversed the race track in $39.78\ s$, well ahead second place ($53.49\ s$)[2]. A plethora of methods was used by the competing teams, which allows for a fair evaluation of other state-of-the-art time-optimal planning methods.

The method was also used in the validation and qualification phase[3], beating the competition by a considerable margin on all maps (in total 5), which have different types and degrees of difficulties. We chose to present in this work the map that showcases the strength of our method: tight corridors where all the mentioned state-of-the-art approaches would fail to generate safe trajectories [41], or perform really poorly (polynomial methods).

3.6 . Limitations and Challenges

The presented approach is currently not suitable for real-time planning. The trajectory generation can take a few seconds for 2-3 gates, and up to 40 minutes for the map *Building99_Hard* which has 13 gates.

Furthermore, in real world experiments, the multirotor is subject to external disturbances, uncertainties and noises that may render the trajectory unfeasible. For example a strong gust of wind may deviate the drone from its path making it hard to recover and track the remainder of the trajectory which already uses the full dynamics of the drone. One way to remedy this problem is to reduce the dynamical limits so that we don't fully use the quadrotor's dynamics. This would allow the controller some margin to correct for the disturbances without having to locally replan. Another way to remedy the problem would be to use an exploration-exploitation based adaptive control law [155] that would deal with uncertainties and disturbances in an effective way.

Finally, the non-convex nature of the problem can present convergence challenges (bad local minima). This problem is present in all non-convex planning approaches [41]. This was partly remedied by the iterative nature of our algorithm as well as the heuristic we use, which provides a good initial guess for the optimization.

3.7 . Conclusion and Future Works

[2] https://microsoft.github.io/AirSim-NeurIPS2019-Drone-Racing/leaderboard_final.html

[3] https://microsoft.github.io/AirSim-NeurIPS2019-Drone-Racing/leaderboard.html

In this work we presented a new multirotor formulation (MINLP) that take into account nonlinearities (gravity and drag forces) as well as path constraints (overlapping polyhedra) for near time-optimal trajectory generation. We then proposed a new algorithm and heuristic to turn the MINLP into an NLP and solve it locally. We also presented simulation results that showcase how our algorithm performs in a drone racing scenario and evaluated the feasibility of the generated trajectories.

So far we have been able to cold start our optimization and rely on KNITRO with an interior/direct algorithm to find the optimal solution. We plan to investigate multiple initialization schemes for faster convergence with a warm start. Furthermore, we plan on testing with parallel computing for the discrete gradient descent (presented in section 3.4 - page 40) to better investigate the real-time capabilities of our algorithm.

4 - VOXEL GRIDS

Regular grids are a commonly used discretized representation of the environment. Their elementary cells are usually chosen to be cubes (also called voxels). Voxel grids are a minimal and efficient environment representation that is used for robot motion planning in numerous tasks. Many state-of-the-art planning algorithms use voxel grids composed of free, occupied and unknown voxels. In this work we propose a new GPU accelerated algorithm for partitioning the space into a voxel grid with occupied, free and unknown voxels. The proposed approach is low latency and suitable for high speed navigation.

This work is used in our following works in trajectory planning and obstacle avoidance since it is an essential part in the autonomy pipeline (perception and mapping) that precedes the planning and control stages.

4.1 . Introduction

Many sensors (RGB-D cameras, stereo-matching ...) output dense point-clouds as measurements and need to be processed and turned into an environment model/representation for motion planning. Fast 3D environment modelling is crucial for real-time high speed motion planning and exploration.

Many state-of-the art techniques use voxel grids for planning [134] [146] [145]. Some of them use the grid for Safe Corridor generation [88] while others use it for direct collision checking. They need the grid to be partitioned into occupied, free and unknown voxels. The fast generation of such grids is the main objective of this work.

We will first present the related work as well as our main contributions. Then, we will outline the different steps of the method and show the simulation results. Finally, we will compare it with **mit-acl-mapping**[1] used in [134]. Our method is implemented on CPU and GPU and we will discuss the performance difference throughout the chapter.

4.1.1 . Related work

Numerous 3D space representations exist such as signed distance fields [117] [55], octrees [64] [33], and voxel grids [134].

[1] https://gitlab.com/mit-acl/lab/acl-mapping

Signed distance fields grids [117] [55] include in every voxel information about the distance and gradient against obstacles and is essential for gradient-based planning methods. The advantage of voxel grids over signed distance fields is processing time, since they don't need to calculate the distance of every voxel to the closest obstacle.

Octrees are a tree data structure where each node has eight children. Octree based representations such as Octomap [64] are an efficient probabilistic 3D representation of the environment. The advantage of voxel grids over octrees is the constant voxel access (read/write) time.

Due to the reasons stated above, we choose voxel grids as the environment representation. Many methods exist for the generation of voxel grids. In [134] authors use [19] to trace every pixel of the image and free the traversed voxels between the camera center and the pixel depth. This approach leads to a high computational cost and become intractable for medium/high resolution RGB-D cameras.

In [62], the authors use General Purpose Graphics Processing Unit (GPGPU) to populate the voxel grid. The authors raycast every measurement point in a Brensenham fashion [19] and update the voxels according to the sensor model.

4.1.2 . Contribution

In this work we propose a new implementation of a voxelization algorithm on the GPU that tries to minimize computation time as much as possible. It takes as input a dense point cloud and outputs a local voxel grid centered at the robot with occupied, free, and unknown cells. However we don't use a probabilistic approach as this reduces performance of the GPU in our implementation (requires atomic operations). We use the GPU since it has the potential to speed up parallelizable tasks of voxel grid generation such as ray-tracing.

4.2 . Nomenclature

We define the various abbreviations and variables used throughout this chapter in table 4.1. All distances and coordinates are in meters except for voxel coordinates which are integers.

4.3 . GPU Architecture

In this section we briefly describe the GPU architecture/nomenclature and how it executes instructions. This will help make sense of the performance benchmarks

Table 4.1: Nomenclature

loc_grid	local voxel grid
ms_grid	measurement voxel grid
$grid\_size_x$	grid size in the x direction
$grid\_size_y$	grid size in the y direction
$grid\_size_z$	grid size in the z direction
vox_size	voxel cube side length
fov_x	camera field of view in the x direction
fov_y	camera field of view in the y direction
$depth$	IR depth sensor range
vox_depth	the $depth$ in number of voxels
vox_width	voxels covered by fov_x at $depth$
vox_height	voxels covered by fov_y at $depth$
x_i	voxel x coordinate in the grid frame
y_i	voxel y coordinate in the grid frame
z_i	voxel z coordinate in the grid frame
$\boldsymbol{T}_c^w$	transform from camera to world frame
$\boldsymbol{T}_w^v$	transform from world to grid frame
$\boldsymbol{p}_o^c$	obstacle point in the camera frame
$\boldsymbol{p}_o^v$	obstacle point in the voxel grid frame
$\boldsymbol{p}_c^v$	camera position in the voxel grid frame
vox_inf	number of voxels to inflate
ray_dir	direction of the ray to trace
ray_start	starting point of the ray to trace
max_dist	maximum ray traversal distance

in the simulation section. We will describe NVIDIA's Turing architecture which our GPU (RTX 2060) is built on [113].

The basic building block of a GPU is the Streaming Multiprocessor or SM. Each SM in the Turing architecture contains 64 cores (2 groups of 32 cores) and has its own L1 shared memory. In every group of 32 cores, all cores execute the same instructions simultaneously, but with different data (Single Instruction Multiple Threads - SIMT). Thus we get 32 threads (called a warp) all doing the same thing at the same time.

The warps (groups of 32 threads) are grouped into blocks and every block can be executed on one SM only. A group of blocks is called a grid.

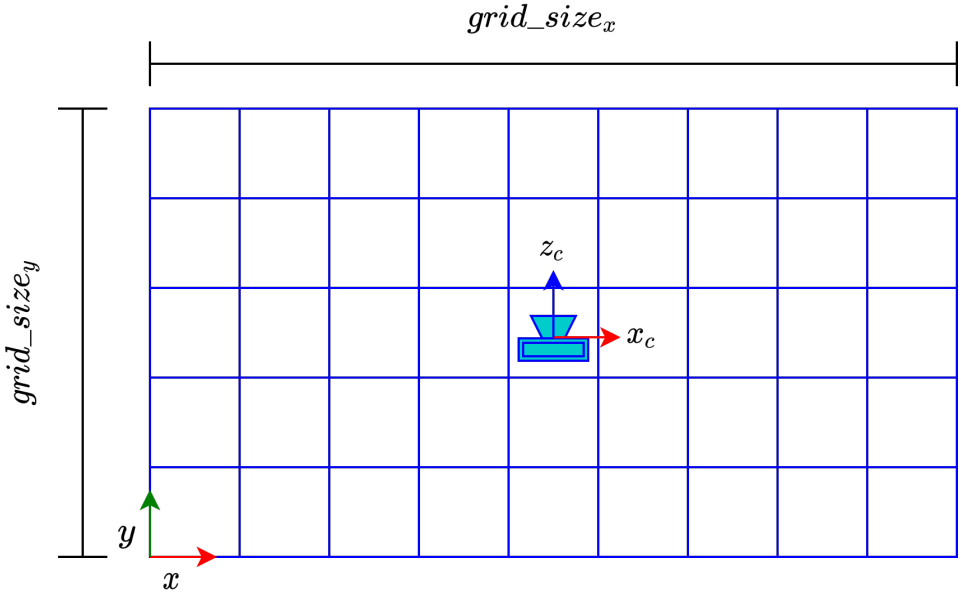


Figure 4.1: We show a 2D example of the grid with the origin set such as the robot/camera position is at the center of the grid.

Throughout the chapter we will measure the efficiency of our implementations through active threads per warp (how many of the 32 cores are active - the higher the better), and the number of warp cycles per executed instruction (the lower the better).

Active threads per warp also depend on predication when we have **if** - **else** conditional statements. These statements can cause branching when a thread meets the **if** condition while another thread meets the **else** condition. The first thread will execute the statement under the *if* condition while the other will be waiting. Once the execution of the *if* statement is done, the second thread starts executing the *else* statement while the first thread is now waiting. This means there is idle/wasted execution time. With predication, the GPU evaluates both sides of the conditional statement and then discards one of the results, based on the value of the boolean branch condition in each thread. Predication is in general effective for small branches.

The GPU code is written in CUDA [110].

4.4 . The Method

The objective is to generate a voxel grid representation of the world around the robot. The method takes a dense point cloud (stereo matching) or a depth image (IR depth sensor) in the camera frame, and populates a voxel grid with occupied, free and unknown cells.

The method consists into updating a sliding local voxel grid centered around the robot's position with a measurement grid that represents the latest measurement. It is divided into 5 steps (functional blocks):

1. Voxel grid setup
2. Populate occupied voxels.
3. Ray-trace to free voxels in camera field of view.
4. Update the local voxel grid with the measurement voxel grid.
5. Shift the local grid to be centered at the robot's position.

A diagram showing a global view of the execution pipeline of steps 1-4 is shown in Fig. 4.2.

In the voxel grid setup, we initialize the local voxel grid and measurement voxel grid. The local grid is only initialized once at the start of the algorithm and is updated at every measurement with the measurement voxel grid that is reset before each measurement. We then set the voxels that contain obstacles to occupied in the measurement voxel grid (populate occupied voxels).

An efficient ray-tracing method is used in the third step (Ray-trace to free voxels in camera field of view). Instead of tracing each pixel we leverage the voxel grid structure to reduce considerably the number of rays to trace by using ray bundling which accelerates the overall processing time. Ray bundling is inspired by [167] and [117].

We then update the local voxel grid with the measurement one which we populated with the occupied voxels and freed its voxels with ray-tracing. Finally we shift the local grid so that it is always centered at the robot position as the robot moves. We assume a regular voxel grid, but the method can be generalized for irregular grids. Each step as well as its GPU implementation are discussed in what follows.

4.4.1 . Voxel grid setup

We have 2 grids: the local voxel grid that is the representation of all previous measurements, and the measurement voxel grid that is the representation of the latest measurement. Both grids are of the same size. It is possible to use only

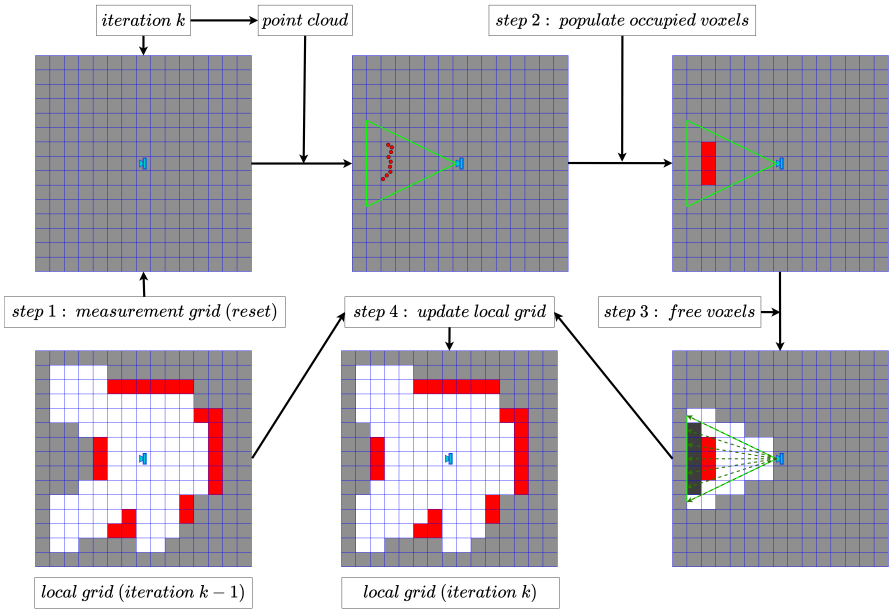


Figure 4.2: We show the execution pipeline at iteration k of steps 1-4 after receiving a measurement (point cloud). In the presented case, an dynamic obstacle moves away from the camera while the robot stays static. **Free** voxels are **white**, **occupied** are **red**, **unknown** are **grey**, **unknown and traced** are **black**. The camera field of view is shown in green and the point cloud as red circles.

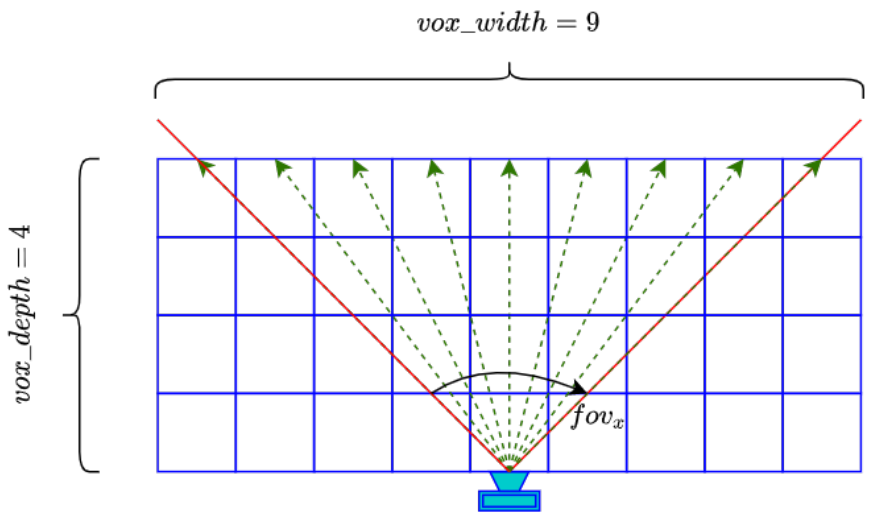


Figure 4.3: We show a 2D example of the rays (in **green**) we trace using our method. The field of view of the camera is limited by the **red** lines.

one grid (the local grid). However, this would require clearing all the voxels in the camera field of view (using ray-tracing) before applying steps (2-4). Otherwise, if we have an obstacle moving away from the sensor, the previously occupied voxels that are now free due to the obstacle's movement will remain occupied. We found empirically that this would result in a higher computation time then using 2 grids and merging them together (due to the high computation cost of ray-tracing).

The origin of the local voxel grid is initialized such as the initial robot/camera position is at the center of the grid. The orientation is the same as the ENU (East North Up) frame (Fig. 4.1). It is then shifted as the robot moves. All voxels are initialized as **unknown**, and will be updated when the measurement voxel grid is merged with the local voxel grid. The voxels of the measurement voxel grid are set to **unknown** before every single measurement. They are then changed by steps 2 and 3.

4.4.2 . Populate occupied voxels

In this step, we set all the voxels in the measurement grid that contain obstacle points to occupied using Alg. 1.

This is done by first transforming the points from the camera frame to the voxel grid's origin (line 2), then dividing the coordinates by the voxel size to get the integer coordinates of the occupied voxel (lines 3-5). For every measurement/camera pose, the transformation matrix is the same: $\boldsymbol{T}_c^v = \boldsymbol{T}_w^v \boldsymbol{T}_c^w$. It is calculated on the CPU and passed as an argument to the GPU implementation.

Algorithm 1 Populate occupied voxels

1: **function** AddOcc($\boldsymbol{T}_c^v, vox\_size, \boldsymbol{p}_o^c, ms\_grid, vox\_inf$)
2: $\quad \boldsymbol{p}_o^v = \boldsymbol{T}_c^v \boldsymbol{p}_o^c$
3: $\quad x_i \leftarrow \boldsymbol{p}_{o,x}^v / vox\_size$
4: $\quad y_i \leftarrow \boldsymbol{p}_{o,y}^v / vox\_size$
5: $\quad z_i \leftarrow \boldsymbol{p}_{o,z}^v / vox\_size$
6: $\quad$ **for** $i = x_i - vox\_inf$ **to** $x_i + vox\_inf$ **do**
7: $\quad\quad$ **for** $j = y_i - vox\_inf$: **to** $y_i + vox\_inf$ **do**
8: $\quad\quad\quad$ **for** $k = z_i - vox\_inf$ **to** $z_i + vox\_inf$ **do**
9: $\quad\quad\quad\quad ms\_grid[i, j, k] \leftarrow occupied$
10: $\quad\quad\quad$ **end for**
11: $\quad\quad$ **end for**
12: $\quad$ **end for**
13: **end function**

Furthermore, certain planning methods require inflating the obstacles by a number of voxels vox_inf. This can be implemented in this step (lines 6-9) where we set the voxels to the **occupied** value.

On the GPU, the frame transformation is done in parallel as well as setting the corresponding voxels to "occupied". If multiple threads in a warp want to concurrently set the same voxel to "occupied", only one of them succeeds while the others are discarded [113].

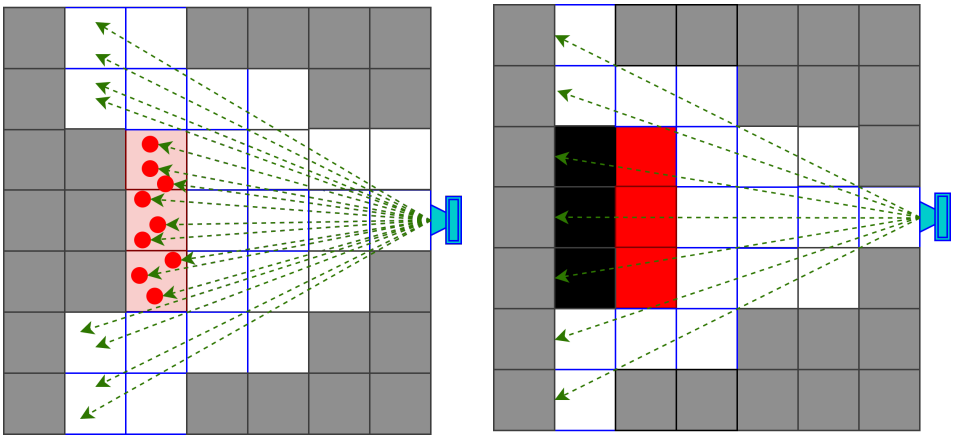
4.4.3 . Ray-trace to free voxels

In this step we free all the voxels in the measurement grid between the center of the camera and the occupied voxel since if any of them contained an obstacle, it would have been detected by the dense pointcloud representation.

Instead of tracing every point that is in a dense pointcloud, or every depth pixel in a depth image, we adopt another approach that significantly decreases the number of rays to trace (Alg. 2).

First we determine the $depth$ which we want to clear (e.g. range of the IR depth sensor) and calculate the number of voxels vox_depth covered by it. This is done by dividing the $depth$ by the voxel size vox_size.

Then, we determine $vox\_width = 2 * \tan(\frac{fov_x}{2}) * vox\_depth + 1$ with fov_x the field/angle of view of the camera in the x direction (Fig. 4.3).



(a) Simple ray-tracing (b) Our ray-tracing

Figure 4.4: We show the **points** of a dense pointcloud in **red**, the **unknown** voxels in **grey**, **occupied** voxels in **red**, **free** voxels in **white** and **unknown and traced** in **black**. The **unknown and traced** voxels will be used to update the local grid unlike the **unknown** voxels. Our method (b) traces less rays with a close end result to a simple ray-tracing method (a) where every pixel of the camera/measurement of the lidar is traced. Instead of tracing every pixel/measurement, we trace up to a given voxel in the grid. When we hit an occupied voxel, we continue tracing but set all subsequent voxels to **unknown and traced**. This speeds up computation time.

Algorithm 2 Ray-trace to free voxels

1: **function** RayTrace($ms\_grid, \boldsymbol{T}_c^v, x_i, y_i, z_i, vox\_size$)
2: $\quad \boldsymbol{p}_c^v = \boldsymbol{T}_c^v[0\ 0\ 0\ 1]^t$
3: $\quad ray\_start \leftarrow \boldsymbol{p}_c^v$
4: $\quad ray\_dir \leftarrow \boldsymbol{T}_c^v[x_i\ y_i\ z_i]^t$
5: $\quad max\_dist \leftarrow vox\_size\sqrt{x_i^2 + y_i^2 + z_i^2}$
6: $\quad traversed\_dist \leftarrow 0$
7: $\quad vox\_val \leftarrow free$
8: $\quad$ **while** $traversed\_dist < max\_dist$ **do**
9: $\quad\quad$ move by a voxel using [3] and set x_i, y_i, z_i to new traced voxel
10: $\quad\quad$ **if** $loc\_grid[x_i, y_i, z_i] == occupied$ **then**
11: $\quad\quad\quad vox\_val \leftarrow unknown\ and\ traced$
12: $\quad\quad$ **else**
13: $\quad\quad\quad ms\_grid[x_i, y_i, z_i] \leftarrow vox\_val$
14: $\quad\quad$ **end if**
15: $\quad\quad traversed\_dist \leftarrow vox\_size\sqrt{x_i^2 + y_i^2 + z_i^2}$
16: $\quad$ **end while**
17: **end function**

We also determine $vox\_height = 2 * \tan(\frac{fov_y}{2}) * vox\_depth + 1$ with fov_y the field/angle of view angle of the camera in the y direction. Note that vox_size, vox_width and vox_height are rounded to the closest integer.

Then we execute Algo. 2. Using [3], we trace all the rays who start from the camera center (lines 2-3), with each ray having as direction one of the vectors going from the camera center to the voxels (line 4) whose integer coordinates are:

$$z_i = vox\_depth$$
$$-\frac{(vox\_height - 1)}{2} \leq y_i \leq \frac{(vox\_height - 1)}{2}$$
$$-\frac{(vox\_width - 1)}{2} \leq x_i \leq \frac{(vox\_width - 1)}{2}$$

We subtract 1 from vox_height and vox_width since by construction they are odd numbers (Fig. 4.3). Since the camera frame is not always aligned with the voxel grid frame, we have to transform the ray direction vector from the camera to the voxel grid origin frame (line 4).

Each ray is stopped when it traverses a predefined distance (e.g. $depth$ distance or $vox\_size\sqrt{x_i^2 + y_i^2 + z_i^2}$ - line 5) Fig. 4.4. The voxels that the ray traverses before hitting an **occupied** voxel are set to free. All the voxels traced after hitting the occupied voxel are set to **unknown and traced** so they are

differentiated from the **unknown** voxels outside the sensor field of view during the update of the local grid (lines 8-14). This is useful in dynamic environments where a previously free/occupied voxel can become occluded by a moving obstacle.

This ray-tracing method assumes that all the obstacles in the camera field of view and within a certain depth are detected, which is ensured with RGB-D cameras or stereo-matching using state-of-the-art methods [143].

On the GPU, we may have concurrent writes by some threads. This results in an undefined behavior on the GPU, as some voxels may have different values written to them concurrently by different rays. This only affects voxels on the borders of a region occluded by an obstacle, where one traversing ray may clear it while another may set it as *free unknown and traced* (which happens in Fig. 4.4. This may not affect the overall performance depending on the application.

4.4.4 . Update the local voxel grid

After populating the measurement voxel grid with occupied voxels and ray-tracing to free voxels, the local voxel grid is updated with the **free**, **occupied**, and **unknown and traced** voxels, i.e. the previous values are replaced with the values of the **free**, **occupied**, and **unknown and traced** voxels of the measurement voxel grid (Alg. 3, line 2-4).

The grids are 1 dimensional arrays with the index $idx = x_i + y_i \times grid\_size_x + z_i \times grid\_size_x \times grid\_size_y$.

On the GPU the update is done in parallel resulting in a significant speed up. The speed up is due to being able to simultaneously access different memory addresses of the same array.

Algorithm 3 Update local grid with measurement grid

1: **function** UpdateGrid(loc_grid, ms_grid, idx)
2: **if** $ms\_grid[idx] \neq unknown$ **then**
3: $loc\_grid[idx] \leftarrow ms\_grid[idx]$
4: **end if**
5: **end function**

4.4.5 . Shift the local voxel grid

Every time the robot moves by a voxel or more, the local map is shifted to be centered at the new robot position and the new voxels resulting from that shift are initialized as **unknown**. This allows to always have the most relevant/close obstacle information to the robot. The shift is done in voxel units so we can simply

use/copy voxels from the old local voxel grid (before shifting).

The shift is done after the local map is updated. This way it doesn't add any unnecessary latency. This step is solely done on the CPU and can be run on a single core/thread which doesn't affect the global computation pipeline (it takes a few milliseconds depending on the size of the local grid).

4.5 . Simulation Results

We run the simulation on the following hardware setup: for the CPU we use Intel Core i7-9750H up to 4.50 GHz, and for the GPU we use NVIDIA's GeForce RTX 2060 up to 1.62 GHz.

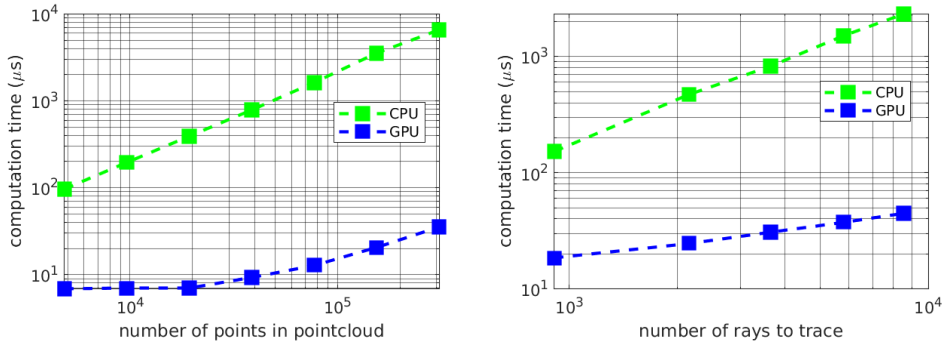
We simulate the robot using mit-acl-gazebo[2]. The sensor is an RGB-D camera that outputs a depth image with a resolution of 320×240. The maximum $depth$ is 6.5 meters, $fov_x = 85\,\text{deg}$ and $fov_y = 101\,\text{deg}$. The chosen size of the voxel grid is $(grid\_size_x = 15\ m, grid\_size_y = 15\ m, grid\_size_z = 3\ m)$ and the voxel size $vox\_size = 0.15\ m$. The robot size is $rob\_rad = 0.3\ m$ which implies $vox\_inf = \dfrac{rob\_rad}{vox\_size} = 2$. This results in $200\,000$ voxels, and $vox\_height \times vox\_width \approx 8\,500$ rays to trace. These are the standard parameters that we use for the simulation time comparisons unless specified otherwise.

We compare the performance of every step run on the CPU with its GPU version. We analyse the efficiency of our GPU implementation in terms of warp state statistics and warp cycles per execution instruction. The occupancy metric of the GPU is not studied as it depends on other factors than the efficiency of the algorithm such as the number of blocks and threads resulting from the pointcloud size/number of rays to trace/number of voxels in a grid. We choose 128 threads per block (4 warps).

4.5.1 . Populate occupied voxels - simulation

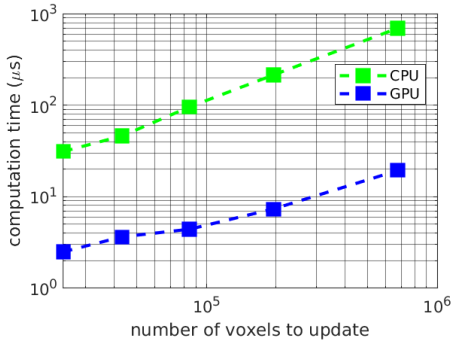
We compare the computation time of the CPU and GPU implementation of the functional block "populate occupied voxels" (section 4.4.2 - page 57) as the number of obstacle points increases (Fig. 4.5a). The scale of the x and y axes is logarithmic. The GPU computation time starts quasi-constant then increases linearly as the number of points surpasses $20\,000$. This is due to the GPU not being fully occupied before reaching the inflexion point. The CPU computation time scales linearly with the number of points in the pointcloud. This is due to the

[2] https://gitlab.com/mit-acl/lab/acl-gazebo



(a) A comparison between the CPU and GPU implementation of step 2 (populating occupied voxels) of our method.

(b) A comparison between the CPU and GPU implementation of step 3 (ray-tracing to free voxels) of our method.



(c) A comparison between the CPU and GPU implementation of step 4 (merging the measurement grid with the local grid) of our method.

Figure 4.5: A comparison between the CPU and GPU implementation of steps 2-4. The CPU implementation is sequential and runs on a single core.

fact that the time complexity of processing one point is constant.

The GPU computation time slope in the linear segment is smaller then that of the CPU. This is expected due to the SIMT architecture and the efficiency of our implementation.

The GPU implementation outperforms the CPU implementation $14\times$ for $5\,000$ points and the difference grows bigger as the number of obstacle points increases. For $300\,000$ point the difference in performance is $185\times$. The difference in performance increases so much due to the fact that the GPU is not fully occupied at $5\,000$ points. Additionally, it is due to the fact that the slope of the computation time of GPU is smaller than that of the CPU once the GPU becomes fully occupied (Fig. 4.5a).

GPU performance

The average active threads per warp (with predication) is $32/32$, and the average not predicated off threads per warp is $29.65/32$. The average warp cycles per executed instruction (which defines the latency between two consecutive instructions) is 11.85. This performance shows that our implementation for this step is efficient on the GPU.

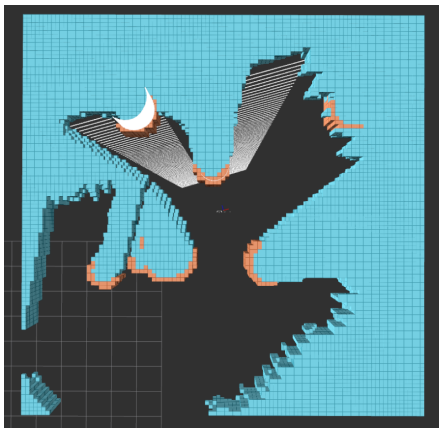
Table 4.2: CPU and GPU average computation time comparison for the different functional blocks using the standard parameters (detailed in section 4.5 - page 62) during an exploration task. The voxel grid setup is not included (negligible time) and the local grid shifting is not included (done only on the CPU after the local map is updated - doesn't affect latency).

	Populate occupied voxels	Ray-trace to free voxels	Update local grid with measurement grid	CUDA memory operations	Total
CPU	$0.1\ ms$	$2.2\ ms$	$0.2\ ms$	-	$2.5\ ms$
GPU	$0.007\ ms$	$0.044\ ms$	$0.007\ ms$	$0.712\ ms$	$0.77\ ms$
Improvement	$14.2\times$	$50\times$	$28.5\times$	-	$3.3\times$

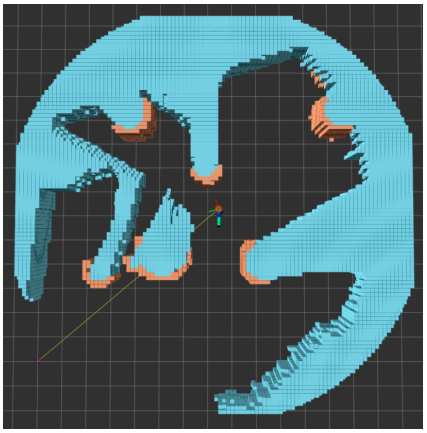
4.5.2 . Ray-tracing to free voxels - simulation

We compare the computation time of the CPU and GPU implementation of the functional block "ray-tracing to free voxels" as the number of rays to trace increases (Fig. 4.5b). The scale of the x and y axes is logarithmic.

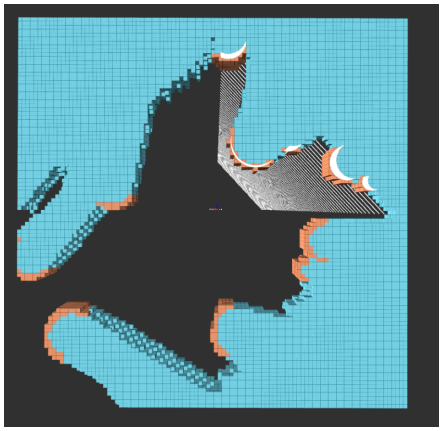
Both CPU and GPU computation times scale linearly. This is expected as the computation time of a single ray-tracing function is limited by max_dist and thus has a constant upper bound. The slope of the GPU computation time is also



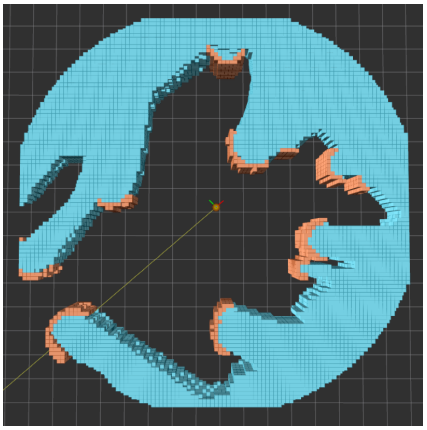
(a) case 1: our method



(b) case 1: mit-acl-mapping



(c) case 2: our method



(d) case 2: mit-acl-mapping

Figure 4.6: We show the voxel grid generated by our method and mit-acl-mapping during a MAV exploration task using [146]. The occupied voxels are shown in **orange**, the unknown voxels in **blue**, and the free voxels are transparent. The two methods deliver close results.

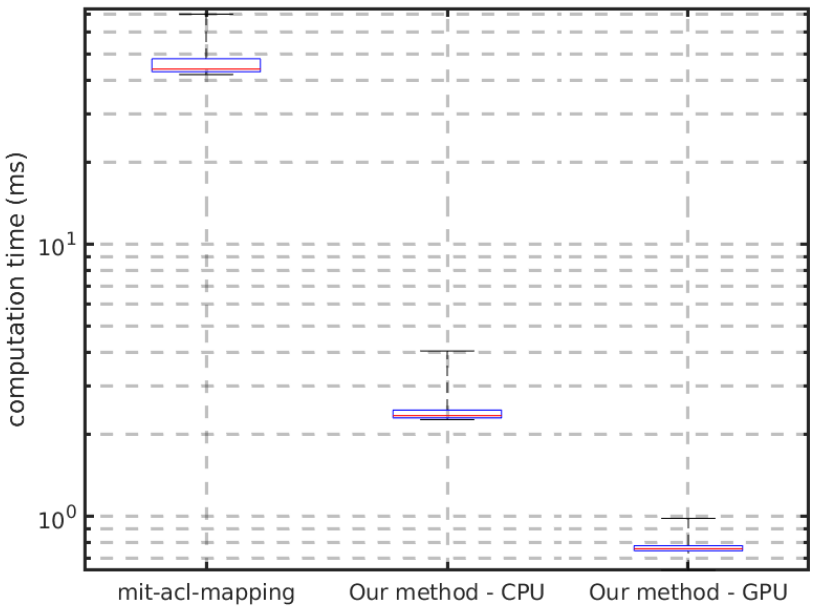


Figure 4.7: A comparison between mit-acl-mapping [134], the CPU and GPU implementation of our method. The red line represents the median. The lower and upper bounds of the box represent the 25th and 75th percentile respectively, and the lower and upper whiskers represent the minimum and maximum respectively.

smaller then that of the CPU in this case. The GPU implementation outperforms the CPU implementation $8\times$ for $1\,000$ rays and the difference grows bigger as the number of rays to trace increases. For $8\,500$ rays the difference in performance is $52\times$.

GPU performance

The average active threads per warp (with predication) is $24.12/32$, and the average not predicated off threads per warp is $20.57/32$. The reason is that we use the method described in [3] for ray tracing, which includes conditional if-else statements that cause branching. Branching in general degrades GPU performance and leads to less active threads per warp and wasted cycles. The average warp cycles per executed instruction is 11.87.

4.5.3 . Update the local voxel grid - simulation

We compare the computation time of the CPU and GPU implementation of the functional block "update the local voxel grid" as the number of voxels in the grid increases (Fig. 4.5c). The scale of the x and y axes is logarithmic.

Both CPU and GPU computation times scale linearly. This is expected as the computation time is constant (read/write access). The slope of the GPU computation time is also smaller then that of the CPU in this case because of the concurrent memory access operations that the GPU is capable of. The GPU implementation outperforms the CPU implementation $12.5\times$ for $24\,000$ voxels and the difference grows bigger as the number of voxels increases. For $675\,000$ voxels the difference in performance is $38\times$.

GPU performance

The average active threads per warp (with predication) is $29.81/32$, and the average not predicated off threads per warp is $29.18/32$. It increase slightly as the number of voxels in a grid increases. The active warps is not $32/32$ due to the *if* statement that verifies that a voxel of the measurement voxel grid is not **unknown** before merging it with the local voxel grid. The average warp cycles per executed instruction is 46.75.

4.5.4 . Comparison with mit-acl-mapping

The proposed method vastly outperforms mit-acl-mapping in terms of computation time (Fig. 4.7) while delivering similar results (Fig. 4.6). The reported computation times are for the setup explained at the beginning of Section 4.5 (page 62), while doing an exploration task using [146]. We show the distribution of the computation time (Fig. 4.7) as a boxplot. The CPU implementation of our method takes on average $2.5\ ms$ and is $18\times$ faster then mit-acl-mapping. The GPU implementation takes on average $0.77\ ms$ and is $3.3\times$ faster than the CPU implementation. Note that in addition to the kernels' execution time (steps 2,3 and 4), the GPU time includes allocating (*cudaMalloc*) and freeing (*cudaFree*) memory on the device (GPU), setting this memory (*cudaMemset*) and transferring data from the host (CPU) to the device and from the device to the host (*cudaMemcpy*). On average, *cudaMalloc* takes $308\ \mu s$, *cudaMemcpy* takes $286\ \mu s$, *cudaFree* takes $97\ \mu s$, *cudaMemset* takes $21\ \mu s$ and the kernels (launch + execution) take $58\ \mu s$ (Table 4.2).

The results (Fig. 4.6) show that our method delivers close results to mit-acl-mapping but more conservative (less voxels are freed) which is expected since our ray is stopped when it hits an occupied voxel whereas mit-acl-mapping uses [19] to trace every pixel of the image (Fig. 4.4a). The method used by mit-acl-mapping

cannot be implemented on a GPU as is, and the ray-tracing method used [19] can miss voxels that the ray passes through, unlike [3] which we use.

4.6 . Conclusion

We presented a novel method for the generation of voxel grids with occupied, free and unknown voxels. The method is efficient while sacrificing little accuracy. Some undefined behavior may occur on the borders of the occupied voxels, but its effects depend on the application and are not considerable in the case of multirotor autonomous navigation. We compared our method to the state-of-the-art and implemented a GPU version of it which resulted in a considerable speed up. The CPU and GPU implementations outperform the state-of-the-art in computation time while delivering similar quality.

5 - SAFE CORRIDORS

Recently, robot planning methods based on Safe Corridors showed promising results in fast navigation for micro-aerial vehicles. Safe Corridors are a series of overlapping convex shapes that cover only obstacle-free space in the environment. They allow to plan safe and dynamically feasible trajecotires. However, state-of-the art methods for Safe Corridor generation either generate "unsafe" Safe Corridors [88], or are not generic enough [67]. In this work we propose a new algorithm for decomposing 3D space into overlapping convex polyhedra based on a voxel-grid representation of the 3D space.

The presented method consists of starting with a seed voxel (around which we want to find the largest convex polyhedron), and expanding it in all directions until we no longer can according to the rules that we define in this chapter. The entity that we expand is what we call a convex grid (group of voxel cells), in which we can inscribe a convex polyhedron. As this convex grid is expanded, the inscribed polyhedron is changed and also puts limits as to how the convex grid can be expanded at the next iteration. The path around which we want to find a Safe Corridor is sampled into points that serve as seeds for each convex grid/polyhedron of the Safe Corridor.

In addition, the presented method creates a connectivity graph between polyhedra of a given Safe Corridor that allows to know which polyhedra intersect with each other. The connectivity graph can be used in planning methods to reduce computation time.

We compared the proposed method with the state-of-the-art in simulation and showed that it generates *Safer* Corridors with guarantees that no intersection exists between the Safe Corridor and the real world obstacles, while staying within real-time constraints. We also showed that our method outperforms the state-of-the-art in terms of number of constraints per polyhedron and number of polyhedra per Safe Corridor, which translates into faster computation time in the planning/optimization phase.

5.1 . Introduction

Safe Corridors are a series of overlapping convex shapes. The convexity of the shapes allows existing solvers to efficiently use them for robot planning. The trajectory is constrained to be inside the SC thus guaranteeing collision-free flight. It is thus of great interest to improve the quality of SC generators in terms of safety

and computation time.

The quality of an SC generator can be judged using multiple criteria. In this work we use the following criteria:

- **Genericness**: it represents the ability to create SCs in an environment that contains obstacles of arbitrary shapes without the requirement for a large amount of polyhedra. A good feature of an SC generator is the ability to generate polyhedra whose faces can potentially have different orthogonal vectors that adapt to the shape of obstacles in the environment. One measure of genericness is the total number of polyhedra in the SC.

- **Volume**: the volume covered by the SC gives more room for the planning robot to plan in, which results in faster or/and smoother trajectories (depending on the cost function used in the optimization stage of planning).

- **Computation time**: whether the SC generation is real-time on computationally constrained embedded systems.

- **Number of Constraints**: since in the majority of planning cases the Safe Corridor is used in an optimization framework, the number of constraints per polyhedron affects the number of inequalities added to the optimization and hence, the solving time.

5.1.1 . Related work

Motion planning

Many methods already use Safe Corridors (SC) for planning for autonomous mobile robots and static obstacle avoidance: humanoids [10], quadrotors [88] [146] and ground robots [146].

There are other approaches for robot planning such as [87], [89], [90], [169] which use motion primitives to change the planning problem into a graph search. This type of approach is computationally expensive especially when generating complex maneuvers around obstacles.

Other methods use Euclidean Signed Distance Fields (ESDF) that represent the 3D space as voxels encoding the distance to the nearest obstacle [116], [117], [55], [45]. The resulting planning problem is non convex, and can result in local minima problems.

Recently Safe Corridor based planning methods [146] were presented and compared to other state-of-the-art methods and have proven to be more performing in the combined metric of computation time, trajectory velocity and trajectory

smoothness.

Safe Corridors

Many methods in the literature exist for the convex decomposition of free space for robot planning.

The method proposed in [67] uses an OctoMap [64] and creates a series of overlapping axes-aligned cubes. While this method is generally computationally efficient and suitable for low compute systems, it is non-generic: it only performs well when obstacles are rectangular parallelopipeds (same issue applies when using cuboids as convex shapes instead of cubes, even if they increase the genericness of the method).

In [47], the authors use a pointcloud representation of the environment and decompose free space around a path into overlapping spheres. This approach is non-generic and can result in a high number of spheres in a relatively low complexity environment (such as a narrow straight corridor).

In [29], the authors search for both an ellipsoid and a set of hyperplanes (convex polyhedron) which separate it from the obstacles (represented as convex polyhedra). The ellipsoid/polyhedron is found by solving a nonconvex optimization problem that maximizes the ellipsoid's volume. This method is generic: there is no limitations to the generated convex shape in the sense that it is a cube or a sphere. However it suffers from a high computation time that is unsuitable for high speed real-time applications.

Finally, in [88], the authors use a pointcloud representation of the environment. The point cloud is downsampled before it is fed to the algorithm [146]. Otherwise, the computation time becomes intractable for real-time applications. The method consists in inflating an ellipsoid around a seed point/line until it hits an obstacle point. Then a tangent plane to the ellipsoid is generated at the obstacle point (a hyperplane), and all obstacle points that are on the side of the hyperplane that doesn't contain the seed are removed. Then, the authors of [88] inflate the ellipsoid further until it hits one of the remaining obstacle points, generate a new hyperplane and remove obstacle points in the same manner. This process is repeated until no obstacle points remain in the pointcloud. The collection of hyperplanes form a convex polyhedron. However, this approach can create unsafe *Safe* Corridors that penetrate the obstacles between the downsampled points. This method is the best performer amongst the state-of-the-art since it is the only one that is generic and real-time. We will compare our algorithm to it throughout the chapter.

5.1.2 . Contribution

The main contribution of this work is a new method for Safe corridor generation that is real-time and generic, and doesn't suffer from the problems that the only real-time generic state-of-the-art method suffers from [88]. It guarantees that the generated Safe Corridor will have an empty intersection with the occupied voxels of a voxel grid. Furthermore, the generated polyhedra will have a lower number of faces than [88], which results in a faster optimization time for planning methods (lower number of inequality constraints). The resulting Safe Corridor can be used by planning methods such as [146] and the method presented in chapter 6 (page 95) to generate feasible and safe trajectories for robot planning.

This contribution was later improved to create a new framework that will also be presented in this chapter in section 5.4 (page 80). The main additions of the improved method are the following:

- The addition of new conditions for expanding a border which makes the new framework have a better decomposition of the free space.
- The creation of a connectivity graph using the convex grid/polyhedron duality that allows to know which polyhedra intersect with each others.

5.2 . Environment and Definitions

The environment is represented as a voxel grid with occupied (obstacles) and free cells. Voxel grids can be efficiently generated from pointclouds as shown in the Voxel Grids section 4 (page 51). In this environment, we want to generate a series of overlapping polyhedra (Safe Corridor) that connect a starting position to a goal position. Every polyhedron we generate can have a minimum of 6 faces (cube) and a maximum of 18 faces (octadecahedron, see Fig. 5.1). It is limited by the occupied cells and covers only free cells. Every polyhedron is generated by iteratively expanding a convex grid (around a cell/seed) inside which we fit the convex polyhedron. The polyhedron doesn't have to be regular as shown in Fig. 5.1.

The convex grid is a group of adjacent cells in which we can fit a convex polyhedron that passes through all the borders of the grid. The borders of the convex grid are defined as the cells with the maximum/minimum coordinates in every direction (x, y and z). At the beginning of the algorithm the grid is only the seed voxel. As the algorithm progresses, the grid expands through its borders and affects/is affected by the inscribed convex polyhedron (Fig. 5.2).

We fix the direction of 6 sides (red in Fig. 5.1) of the polyhedron (2 in the x direction, 2 in the y direction and 2 in the z direction), and leave the direction of

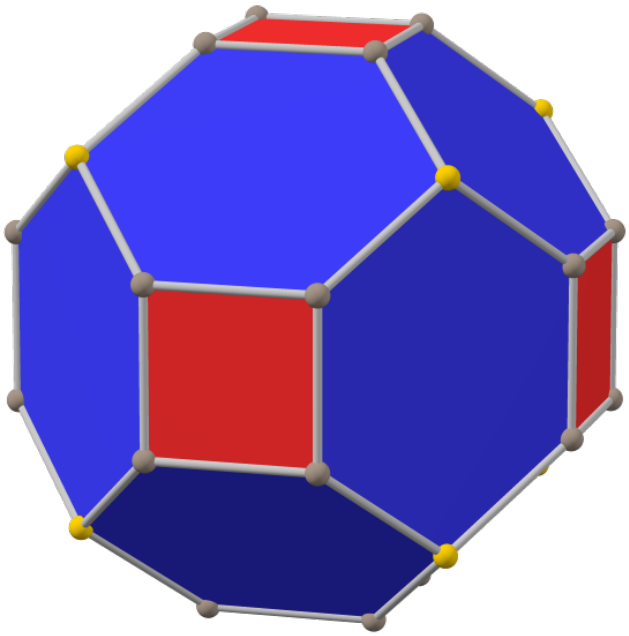
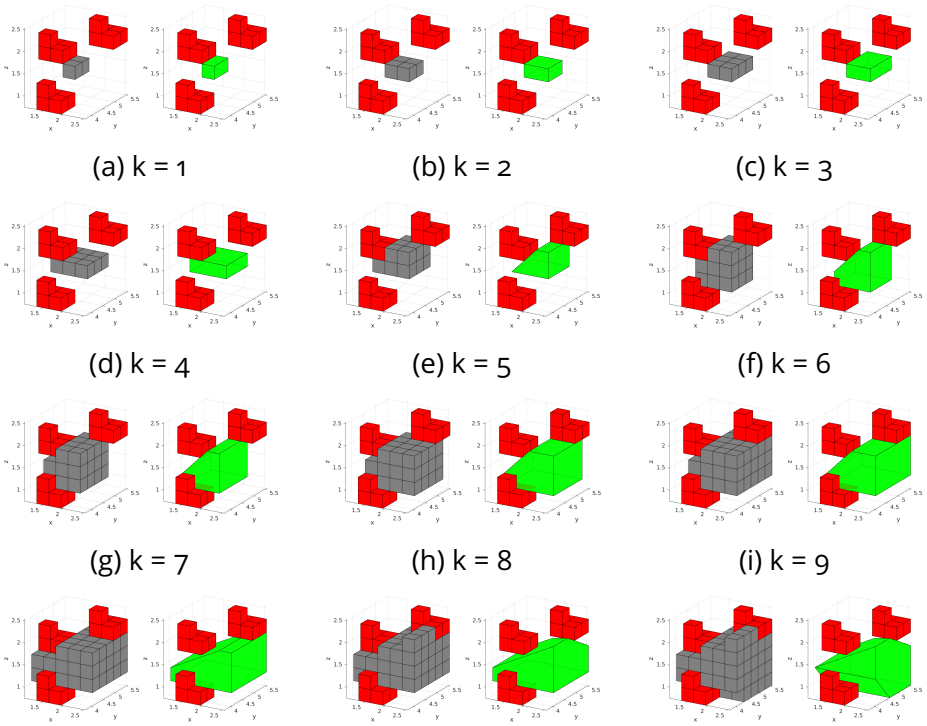


Figure 5.1: A convex octadecahedron [28]. The **side** faces are in **red** and the **corners** in **blue**. The direction of the 6 faces is fixed. The 12 corners' **position**, **slope** and **direction** are fixed as the convex grid expands.



(a) k = 1 (b) k = 2 (c) k = 3

(d) k = 4 (e) k = 5 (f) k = 6

(g) k = 7 (h) k = 8 (i) k = 9

(j) k = 10 (k) k = 11 (l) k = 12

Figure 5.2: We show the evolution of the **convex grid** (in **grey**) and the inscribed octadecahedron (in **green**) as the algorithm progresses. The **obstacles** are the **red** voxels.

the other 12 corners (blue in Fig. 5.1) to be fixed by the expanding convex grid. The **sides** have a single property: **position**. Since we know the orthogonal vector of the side, we only need to determine the point (**position**) through which it passes.

The **corners** (Fig. 5.3) have the following properties to determine:

1. **position:** a point through which it passes.
2. **slope:** the slope of the corner described by number of cells (Fig. 5.3 for more details). Initialized to 0 (corner doesn't exist).
3. **direction:** the direction of the slope. This indicates to which **side** the corner is "leaning" i.e. the side with which the **corner** forms the bigger inside angle (angle inside the polyhedron). This can take any value from 0 to 5. Initialized to -1.
4. **fixed:** whether the slope has been fixed or can still change as the **convex grid** expands. Initialized to *false*.
5. **steps:** this variable indicates how many cells of the corner exist on the side of the border that is in the opposite direction to the direction of the corner. Initialized to 0.

An instance of the corner where each property has a unique value is called a **state** of the corner. A **corner** adjacent to a **side** is a **corner** that has at least one intersection segment with the corresponding **side**. Each side has 4 adjacent **corners** (Fig. 5.1).

5.3 . Convex Polyhedron Generation

5.3.1 . Overview

The method consists in finding a convex polyhedron - an octadecahedron in the extreme case - (Fig. 5.1) inside an expanding convex grid. In this section we will define the *base* rules on which we will improve in the *Improved method* (see section 5.4 - page 80).

We first start with a seed (one voxel) around which we want to find a convex polyhedron. We then expand it in 6 directions (positive and negative x, y and z directions). The order of the cyclic expansion is: $-y$, x, y, $-x$, z, $-z$ (Fig. 5.2). Every time we expand in a direction we increment the expansion counter k. The algorithm is stopped when the expansion counter reaches a predefined limit, or when we cannot expand in any direction.

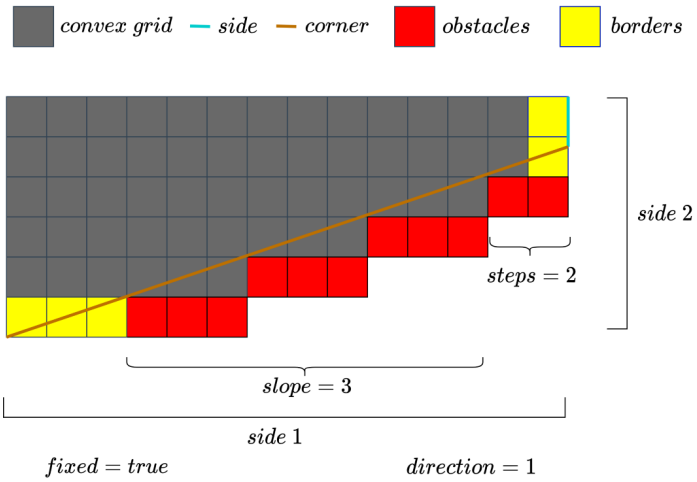
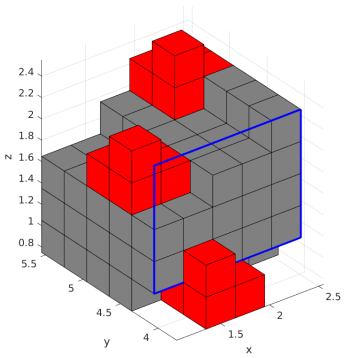
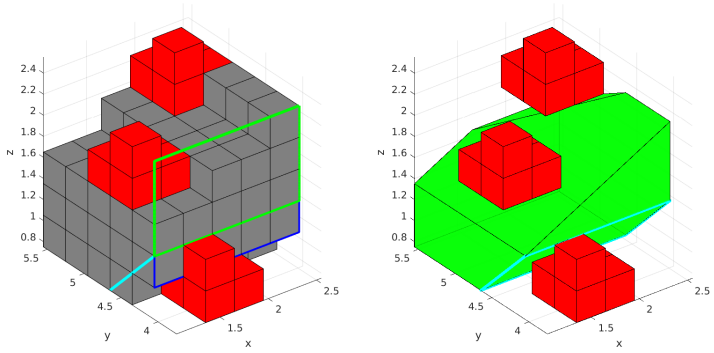


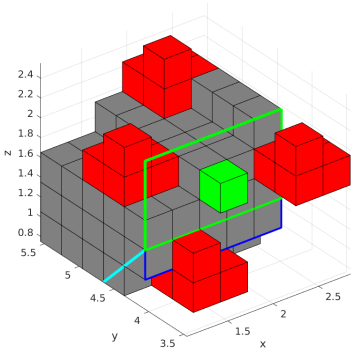
Figure 5.3: We show an example of a corner. The **slope** is 3 as the obstacles form a stair of 3 cells as a step. Since the slope is leaning to side 1 (the corner has a bigger inside angle (angle inside the polyhedron) with side 1), the **direction** is set to 1. Only 2 cells have been traversed on the opposite side (side 2), this means the **steps** is 2. The **slope** is **fixed** by the obstacles and will not change as we expand the borders.



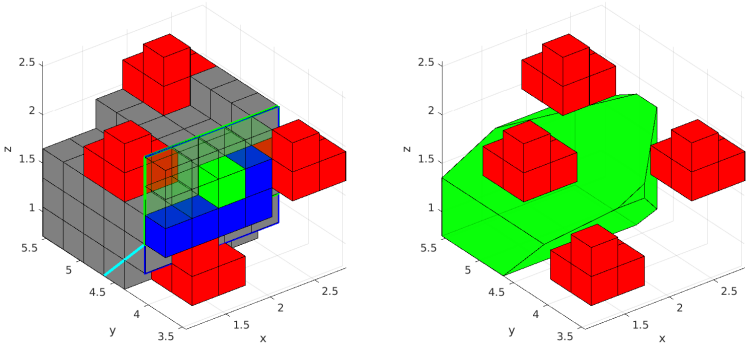
(a) **step = 1:** limits in **blue** fixed by the maximum coordinates of the border cells.



(b) **step = 2:** the **blue** limits are then modified to the **green** limits to take into account adjacent **corners**. We show in **cyan** the borders of the adjacent **corner** of the inscribed octadecahedron.

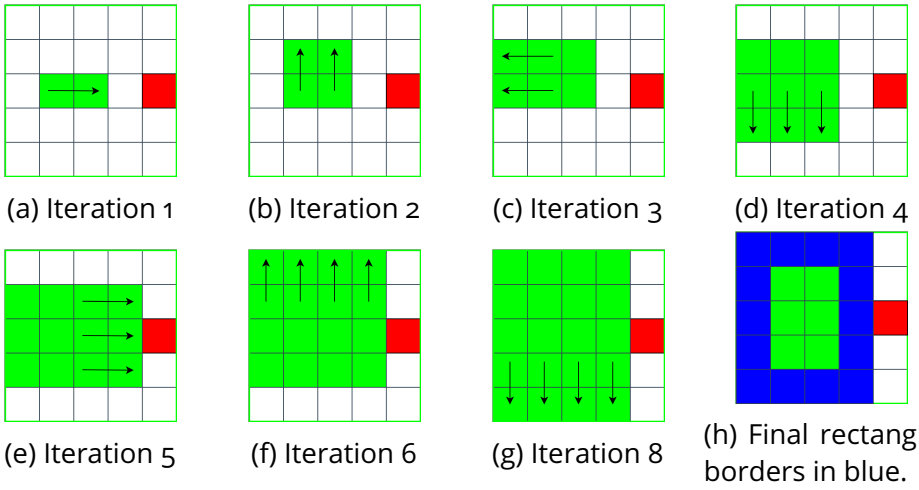


(c) **step = 3:** we choose a seed voxel (in **green**) that is in the **green** limits. The seed is chosen by expanding the border cells in the expansion direction ($-y$) and checking whether the new voxel is inside the **green limits** or occupied (coincides with a **red** voxel). The seed is chosen to be close to the center of the green limits.



(d) **step = 4:** we expand the seed to find the largest rectangle inside the **green** limits while taking into account the obstacles (in **red**). The **blue** voxels denote the **rectangle borders**. The expansion method is shown in Fig. 5.5. We added an obstacle for the sake of demonstration. The transparent voxels of the rectangle are not in the new **border candidate** (**green** and **blue** voxels) as they don't have any **grey** voxel (belonging to the **convex grid**) behind them. We show the new inscribed octadecahedron that results from this expansion once the new **border candidate** is validated Sect 5.3.3.

Figure 5.4: We show the method for determining the new **border candidate** when expanding in the $-y$ direction. These candidates will then need to be validated before modifying the adjacent **corners** and generating the resulting inscribed octadecahedron.



(a) Iteration 1 (b) Iteration 2 (c) Iteration 3 (d) Iteration 4

(e) Iteration 5 (f) Iteration 6 (g) Iteration 8 (h) Final rectangle borders in blue.

Figure 5.5: We show the expansion of a seed chosen during the expansion of a border of the convex grid. The seed is expanded into a rectangle that is within the **green** limits and that is limited by the obstacles (in **red**). We don't show Iteration 7 as the rectangle is not expanded at this iteration due to the **green** limit.

5.3.2 . Expanding a border

We have 6 borders (one in every direction), each containing cells of the **convex grid** that have the maximum coordinate in the expansion direction. At each iteration, we only consider one of them for expansion according to the cyclic expansion rule defined in 5.3.1.

We first determine the limits of the new **border candidate** cells. They are set using multiple steps that include the state of the adjacent **corners** to the border **side** (Fig. 5.4). In the first step we fix the **blue** limits. These limits are fixed by the maximum coordinates of the border cells i.e. they represent the minimum rectangle that includes all the border cells as shown in Fig. 5.4a.

In the second step, the **blue** limits are modified using the adjacent **corners** to generate the **green** limits. Each adjacent **corner** affects the side of the **blue** limits/rectangle that is adjacent to it by shrinking it a distance equal to the **corner**'s slope. An illustration can be seen in Fig. 5.4b where the adjacent corner in **cyan** (highlighted in the convex grid as well as the inscribed polyhedron) intersects the **blue** limits to generate the **green** limits. This intersection reduces the adjacent **blue** limit by a distance equal to the slope of the adjacent **corner**, which in this case is 1 voxel.

In step 3, we choose a seed voxel (shown in **green** in Fig. 5.4c) inside the **green**

limits by expanding the border in the expansion direction and checking whether the expanded voxel is inside the **green** limits or occupied. The seed is chosen to be close to the center of the **green** limits.

In step 4, we expand the seed found in step 3 to find the largest rectangle inside the **green** limits while taking into account obstacles as shown in Fig. 5.5. The **blue** voxels shown in Fig. 5.4d represent the newly formed **rectangle borders** These voxels (in **blue** in Fig. 5.4) constitute the **rectangle borders**, and they will be used to validate the new **border candidate** and modify the adjacent **corners** accordingly. The transparent voxels are not in the new **border candidate** (which consists of the **green** and **blue** voxels) since they don't have any **grey voxels** (belonging to the **convex grid**) behind them.

5.3.3 . Adapting the new border candidate and the corners

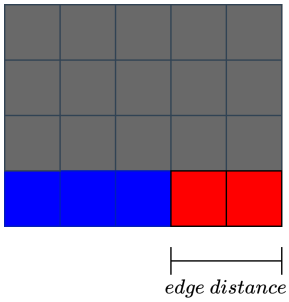
After finding the new **border candidate**, we have to check whether it is valid and to modify the 4 adjacent **corners** accordingly. We will give an example for one adjacent **corner** but the same principle is applied to the other 3 adjacent **corners**.

We first determine the side of the **rectangular borders** that affects a given adjacent **corner** i.e. the cells of the rectangle that are the closest to the other adjacent **side** of the adjacent **corner**. If this side exists (i.e. the voxels aren't transparent Fig. 5.4), we calculate the distance of this side to the corresponding **blue** border limit in number of voxels (**edge distance** Fig. 5.6d). Then depending on the value of that distance and the state of the adjacent **corner** we have multiple cases. All of these cases are explored in an exhaustive way and explained with example illustrations for **border candidates** in Fig. 5.6. Note that the example illustrations are only an example of the case which is defined in the subtitle of the figure (there are configurations that are different than the illustrations and that obey the conditions of each case). Also note that no configuration exists that doesn't obey one of the presented cases (they are exhaustive).

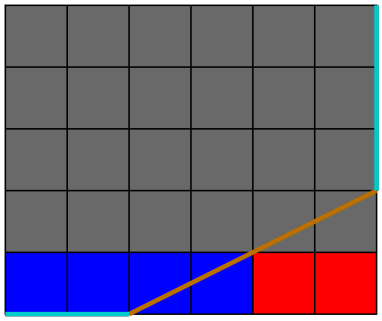
After going through all the cases for the 4 adjacent **corners**, if one of them leads to an invalid border, the new **border candidate** is discarded. Otherwise, the modified **corners** are saved, and the new **border candidate** is added to the **convex grid**.

5.4 . The Improved Method

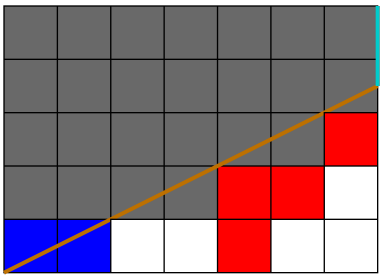
The improved method adds additional heuristics and rules to the *base* rules presented in section 5.3 (page 75) in order to make the decomposition of better quality in terms of adapting to the obstacles shape.



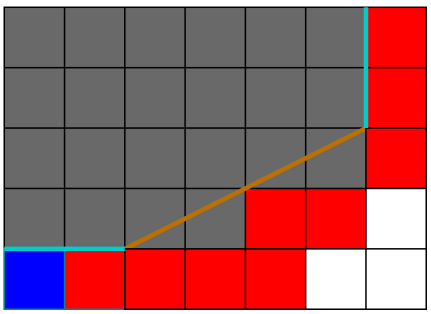
(a) We show a cross section of a convex grid and the **edge distance** (=2) between a side of the **rectangular border** (**blue voxels**) and the **blue** limits shown in Fig. 5.4.



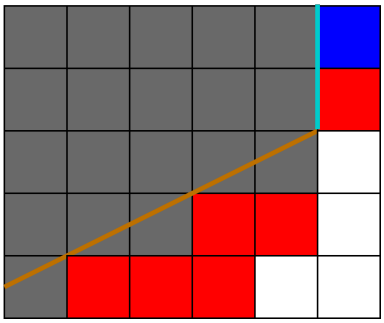
(b) **case 1:** The **corner** doesn't exist yet, it is initialized with the **slope** and **steps** equal to the **edge distance**. If the **edge distance** is bigger then 1, the **direction** is set to the expanded **side**. The **border candidate** is **valid**.



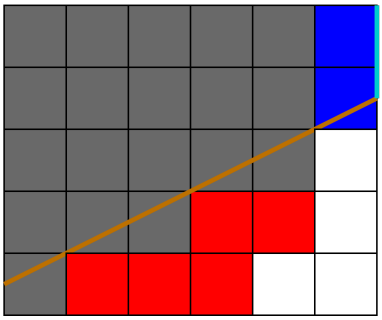
(c) **case 2:** The **corner** is **fixed** and the **direction** is the same as the expanded **side** or -1. If **edge distance** = slope, the **border candidate** is **valid**.



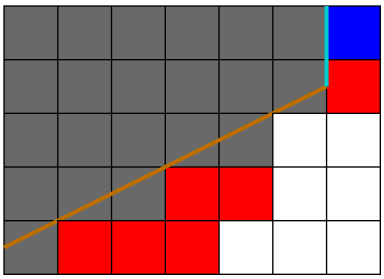
(d) **case 3:** The **corner** is **fixed** and the **direction** is the same as the expanded **side** or -1. If **edge distance** $>$ slope, the **border candidate** is **not valid**.



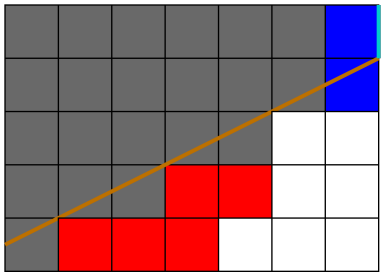
(e) **case 4:** The **corner** is **fixed** and the **direction** is different then the expanded **side**. If **steps** = **slope** and the **edge distance** $>$ 1, the **border candidate** is **not valid**.



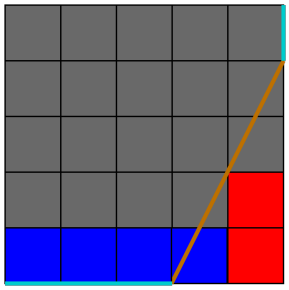
(f) **case 5:** The **corner** is **fixed** and the **direction** is different then the expanded **side**. If **steps** = **slope** and the **edge distance** = 1, the **border candidate** is **valid** and **steps** = 1.



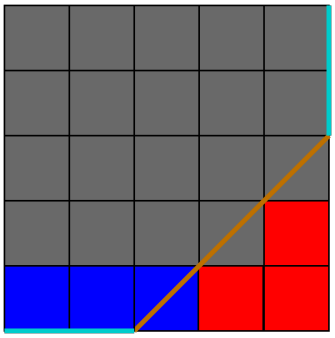
(g) **case 6:** The **corner** is **fixed** and the **direction** is different then the expanded **side**. If **steps** $<$ **slope** and the **edge distance** $>$ 0, the **border candidate** is **not valid**.



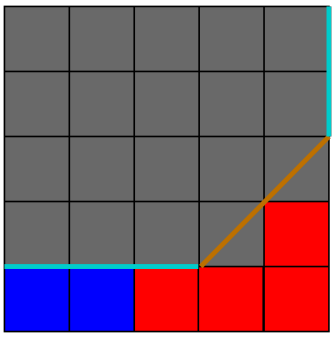
(h) **case 7:** The **corner** is **fixed** and the **direction** is different then the expanded **side**. If **steps** $<$ **slope** and the **edge distance** = 0, the **border candidate** is **valid** and **steps** is incremented by 1.



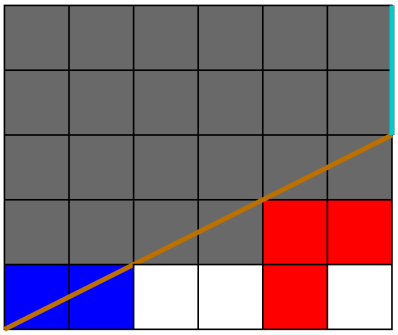
(i) **case 8:** The **corner** is **not fixed** and the **direction** is -1. If **edge distance** = 0, the **border candidate** is **valid** and **steps** and **slope** are incremented by 1. The **direction** is set to the adjacent **side** of the **corner** that is **not expanded**.



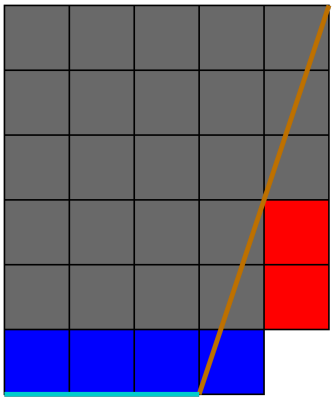
(j) **case 9:** The **corner** is **not fixed** and the **direction** is -1. If **edge distance** = 1, the **border candidate** is **valid** and the **corner** is **fixed**.



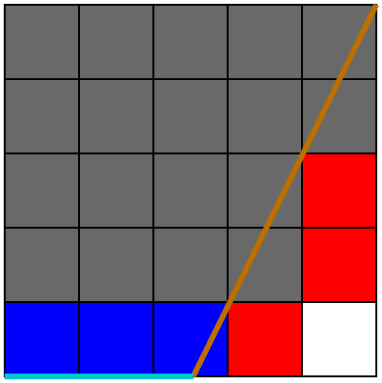
(k) **case 10:** The **corner** is **not fixed** and the **direction** is -1. If **edge distance** $>$ 1, the **border candidate** is **not valid**.



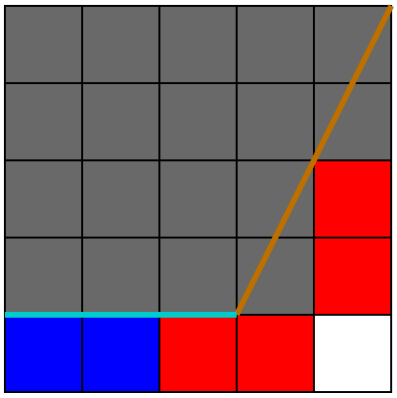
(l) **case 11:** The **corner** is **not fixed** and the **direction** is equal to the expanded side. The **border candidate** is **valid**, the **slope** is set to **edge distance** and the **corner** is **fixed**.



(m) **case 12:** The **corner** is **not fixed** and the **direction** is different then the expanded **side** or -1. If **edge distance** = 0, the **border candidate** is **valid** and the **slope** and **steps** are incremented by 1.

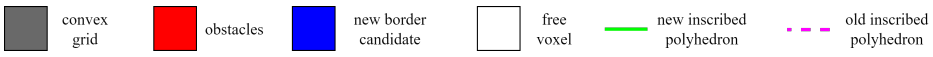


(n) **case 13:** The **corner** is **not fixed** and the **direction** is different then the expanded **side** or -1. If **edge distance** = 1, the **border candidate** is **valid**, **steps** is set to 1 and the **corner** is **fixed**.

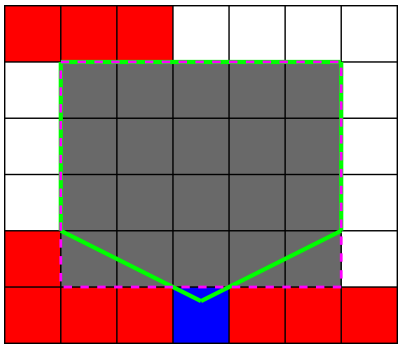


(o) **case 14:** The **corner** is **not fixed** and the **direction** is different then the expanded **side** or -1. If **edge distance** > 1, the **border candidate** is **not valid**.

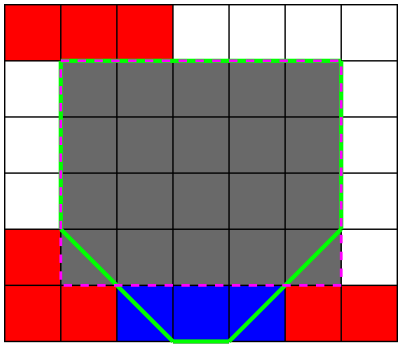
Figure 5.6: We present all the cases that we may encounter when modifying a **corner** after generating a new **border candidate**.



(a) Legend



(b) Invalid expansion



(c) Valid expansion

Figure 5.7: We show a 2D example of the heuristic that invalidates a border expansion. The heuristic that we use is that the number of the new border candidate should be bigger then the border that we expand in terms of number of voxels. In the first case 5.7b, the new border candidate that results from the expansion has 1 voxel whereas the number of voxels that we expanded was 5. In this case the expansion is invalid according to the heuristic. One can see that if we validate it, the new inscribed polyhedron (in green) in the convex grid covers less space then the old inscribed polyhedron (in dashed purple). A valid expansion according to the heuristic is shown in 5.7c, where the new border candidate has 3 voxels (more than half the voxels expanded - in this case 2.5). One can see that the new inscribed polyhedron covers more space than the old one.

5.4.1 . Polyhedra volume heuristic

We take the rules of expansion defined in section 5.3 (page 75) and add the following condition: when we find a new border candidate, we only validate it if the number of voxels in it is bigger then half the number of voxels in the border we are expanding. This heuristic generally avoids making the volume covered by the inscribed polyhedron smaller after the expansion. An expansion example can be seen in Fig. 5.7.

The rules defined in section 5.3 (page 75) in addition to this condition is what we will refer to as *legacy* rules in the remainder of this section to validate/invalidate an expansion.

5.4.2 . Newly initialized corners

When a new border candidate requires the initialization of a new corner (see section 5.3 - page 75 for more details on when corners are initialized), we have 2 cases. The first case is when the direction of the corner is fixed (which we will denote **C1**), and the other case is when it is not fixed (which will we denote **C2**) - see Fig. 5.8. Each case requires additional checks and rules to the *legacy* rules (applied to each newly initialized corner) before we validate the new border candidate.

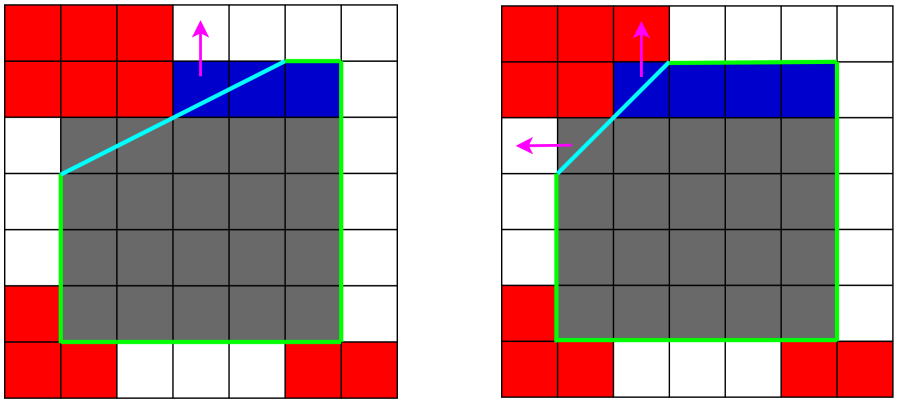
First check:

First we check the following condition for each of the newly initialized corners: we expand all the voxels of the new border candidate that are adjacent to the newly initialized corner in the same direction as the new border candidate. If all of the newly expanded voxels are empty and we are in the case **C1** then the expansion is invalid (Fig. 5.8a). If they are empty and we are in **C2** then we expand the voxels in the initial expansion border that are adjacent to the newly initialized corner. If the newly expanded voxels are also empty then the expansion of the new border candidate is not valid (Fig. 5.8b).

Second check:

If we have newly initialized corners, and after the first check if the expansion is still valid, we proceed to the second check which is done for every newly initialized corner: We first expand the new border candidate in the same expansion direction that gave us the new border candidate using the *legacy* rules (we call this expansion **E1**). This time the expansion is done by assuming that all the newly initialized corners do not exist, or in other words the cells of the convex grid that are on the side adjacent to the newly initialized corner do not exist 5.9b. The new expansion will now modify all corners accordingly.

If the new expansion is valid according to the *legacy* rules, and if we are in the case **C1**, we check if the slope of the corner that we obtain by the new expansion is smaller then the one obtained by the new border candidate 5.9c. If that is the case, the expansion is invalid. If we are in case **C2**, we also expand the other border adjacent to the newly initialized corner while assuming the newly initialized corner does not exist yet i.e. we discard the voxels of the new border candidate from the convex grid (we call this expansion **E2** - Fig. 5.10d). If both **E1** and **E2** result in the newly initialized corner having 0 slope, the expansion is invalid.



(a) Case **C1**: direction fixed (b) Case **C2**: direction unknown

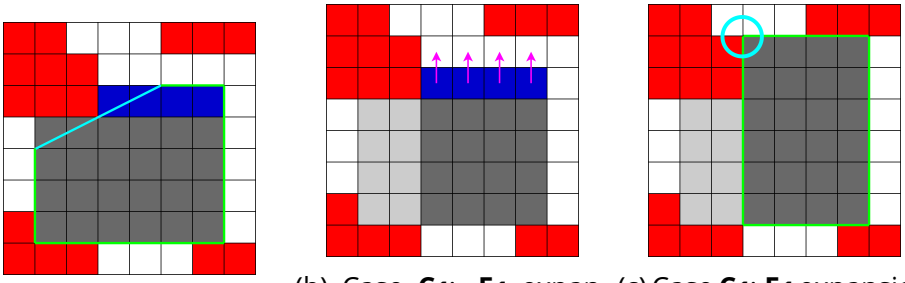
Figure 5.8: We show a 2D example of the 2 cases that are treated by our method. In the first cast 5.8a the corner of the new inscribed polyhedron (shown in cyan) has a fixed direction according to the *legacy* rules, whereas in case 2 5.8b the corner direction is not determined yet. We also show with the magenta arrows the expansion of the voxels adjacent to corner in each case. These voxels are used to determine whether the expansion is valid in each case. In the case shown if 5.8a, the expansion is not valid, whereas in 5.8b the expansion is valid because expanding one of the voxels results in an occupied voxel.

Note that these conditions invalidate expansions. In all other cases the expansion is validated (unless invalidated by *legacy* rules) and the inscribed polyhedron is modified accordingly.

The ability of the proposed improved method to sense its surroundings before making a decision on whether to expand in a direction or not is what gives it a shape-aware characteristic.

5.5 . Generating Safe Corridors

We use the method described above for generating convex polyhedra around a seed voxel to generate multiple convex polyhedra around a given path. Like [88], we first find a valid path between the staring point and the goal point using Jump Point Search (JPS) [58] which requires an occupancy grid. We additionally use the Distance Map Planner (DMP) [130] to push the path away from obstacles which results in a better seed decomposition. Note that JPS/DMP do not necessarily generate the best path in terms of obstacle clearance. Other path planners can also be used to find a valid path between these two points. The resulting path can also be used by our method to generate a Safe Corridor as described below.



(a) Case **C1**: initial case (b) Case **C1**: **E1** expansion - step 1. (c) Case **C1**: **E1** expansion - step 2.

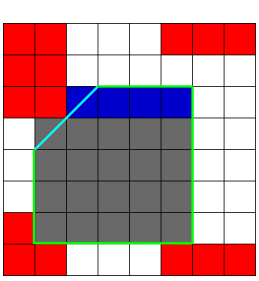
Figure 5.9: We show a 2D example of the case **C1** and the corresponding expansions (**E1**). We expand in the same direction that we expanded in to get the new border candidate 5.9b. This is done by ignoring the cells of the convex gird that are on the side adjacent to the newly initialized corner (depicted in light gray), which is also the same as saying the newly initialized corner does not exist. Once the new convex grid and the corresponding inscribed polyhedron are obtained 5.9c, we check for the newly initialized corner in the new inscribed polyhedron (encircled in cyan). If it does not exist (which is the case here), the border expansion that resulted in the new border candidate is not valid.

We use an iterative decomposition method to determine the seeds of the overlapping polyhedra. The first polyhedron is seeded with the starting position, then we find the intersection between the path and the polyhedron and seed the second polyhedron at this intersection. If the intersection results in the same seed as the last polyhedron, we move along the global path by the voxel resolution and choose a new seed. We continue this strategy until we reach the goal position.

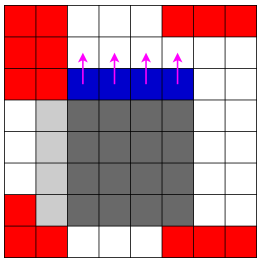
5.6 . Simulation Results

5.6.1 . Simulation setup

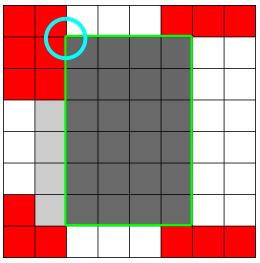
The simulation is done on an Intel® Core™ i7-9750H (base 2.60GHz, up to 4.50 GHz). All coordinates and distances are in meters. The environment is of size $50 \times 12 \times 12$ and is represented as a voxel grid of voxel size 0.3. We generate 400 obstacles of size between 0.9 and 1.5 (using a uniform distribution) i.e contained in a cube (**obstacle cube**) whose side length is between 3 and 5 voxels. The shape of each obstacle is randomized by setting to occupied with a probability of 0.5 the voxels inside the **obstacle cube**. The position of the obstacles is generated following a uniform distribution.



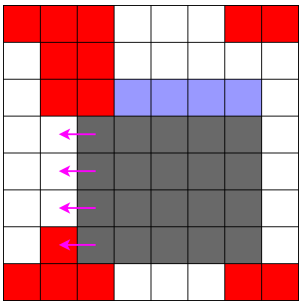
(a) Case **C2**: initial case



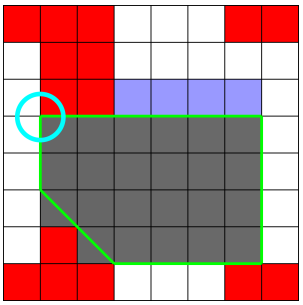
(b) Case **C2**: **E1** expansion - step 1.



(c) Case **C2**: **E1** expansion - step 2.



(d) Case **C2**: **E2** expansion - step 1.



(e) Case **C2**: **E2** expansion - step 2.

Figure 5.10: We show a 2D example of the case **C2** and the corresponding expansions (**E1** and **E2**). The expansion **E1** is done as shown in Fig. 5.9. The expansion **E2** is done by ignoring the new border candidate (depicted in light blue) i.e. assuming the newly initialized corner does not exist, and expanding in the adjacent direction of the newly initialized corner. The new expansion gives us a new value for the newly initialized corner. If by both expansions the newly initialized corner does not exist (which is encircled in cyan in 5.10c and 5.10e), the expansion is invalidated.

The starting point is $(x = 3, y = 6, z = 6)$ and the goal is $(x = 47, y = 6, z = 6)$. The JPS and DMP combined take on average $6\ ms$. We optimize Liu et al. [88] by using the voxel grid to get the obstacle points (centers of the occupied voxels) that are within a bounding box of the seed. We set this box to $4 \times 4 \times 4$. In order to cover the same volume, we set the number of border expansions $n = 36$ (6 border expansions to do a full rotation and expand by the voxel size in every direction - Fig. 5.2).

5.6.2 . Simulation results

We compare our *base* method (section 5.3 - page 75) and our *improved* method (section 5.4 - page 80) with the method proposed in [88] in finding a Safe Corridor (SC) between the same starting and goal points in 10 randomly generated environments. The comparison metrics are volume covered, number of constraints per polyhedron, number of polyhedra per SC (genericness), computation time and SC safety. We show the mean, max, and standard deviation values of the relevant metrics in Tab. 5.1.

Table 5.1: Comparison between Liu et al. [88], *base* method and the *improved* method proposed in this chapter on 10 randomly generated maps of size $50 \times 12 \times 12$ and with 400 obstacles. The **mean / max / standard deviation** of every metric is shown. The difference in performance between the *base* and the *improved* method is shown for the **mean** and **max** values. The better performer between the 3 methods is shown in bold.

	Volume (m^3)	Constr/Poly	Poly/SC	Comp. time (μs)	Safe
Liu et al. [88]	**482** / **519** / 29.5	15.2 / 26 / 3.2	27.5 / 33 / 2.8	**107** / **222** / 37.7	no
Base method	414 / 456 / 29.8	10.9 / 16 / 1.7	27.8 / **31** / 2.1	187 / 349 / 54	**yes**
Improved method	399 / 446 / 35	**7** / **12** / 1	**27.3** / **31** / 2	191 / 337 / 52	**yes**
Difference (%)	-3.6/-2.4	-35.7/-25	-1.8/0	+2/-3.4	-

Our *base* methods covers a lower volume then [88] (on average 13.9% less), but has a lower number of constraints per generated polyhedron (on average 30% less) which allows faster optimization times while planning. The number of polyhedra per generated SC is similar between our *base* method and [88] with a difference less than 3%. Our *base* method takes on average 81.1% more time than [88], however it remains largely within real time constraints and its share of the total planning time remains negligible as shown in chapter 6 (page 95). Finally, our *base* method is safe (in contrast to Liu et al. [88]) and guarantees no intersection exists between the real world obstacles and the generated SC. To better illustrate this case, we create a smaller environment with less obstacles as follows.

Our *improved* method is similar to our *base* method in terms of volume covered (on average 3.6% smaller), number of polyhedra per SC (on average 1.8% smaller),

computation time (on average 2% bigger), and safety (both equally safe). However, our *improved* method significantly outperforms our base method in terms of number of constraint per polyhedron (on average 35.7% smaller) which results in a lower number of constraints for the trajectory generation optimization problem. This reduction in the number of constraints results in a reduction in computation time.

5.6.3 . Case study

While the aforementioned performance metrics provide some insight into the comparative performance of these methods, we show in Fig. 5.11 an overhead view of the decomposition of a particular environment to better emphasize the differences. The shape of the obstacles shown in Fig. 5.11 is visible only in the x and y directions. The grid limits are $[0, 35]$ in the x direction, $[0, 10]$ in the y direction and $[0, 5]$ in the z direction. The voxel size is chosen $voxel\_size = 0.3$.

The starting point is $(x = 2, y = 5, z = 1.5)$ and goal $(x = 32, y = 5, z = 1.5)$. The JPS an DMP combined take $4\ ms$. We set the bounding box of Liu et al. [88] to $4 \times 4 \times 4$ and the number of border expansions of our *base* and *improved* method to $n = 36$. Since the voxel size is 0.3 and we expand by the voxel size in all directions every 6 iterations, we get a bounding box (cube) of side length $2 * (\frac{n}{6} * 0.3 + \frac{0.3}{2}) = 3.9$. The obstacle points shown are the centers of the occupied voxels in a voxel grid. The overhead view shows a horizontal slice (in the $x - y$ plane) of the obstacles i.e. the shown obstacle points are repeated at every voxel level in the z direction to create 3 dimensional obstacles.

In Fig.5.11 we show that the results of [88] and both our methods are somewhat similar in the way they efficiently cover the free space around a path using overlapping convex shapes. However, using [88] with a downsampled point cloud can result in a decomposition failure where the polyhedron penetrates the space between the points (encircled in **dark blue** in Fig. 5.11a). This is due to the nature of the algorithm they use (as described in 5.1.1) and to the fact that we use the voxel grid to downsample the pointcloud to accelerate the convex decomposition as done in [146]. This problem does not exist in our method. Since many methods can push the planning agent to the extremities of a polyhedra because of high speed or because of other planning agents restricting the planning space, guaranteeing that the Safe Corridor has no intersection with real world obstacles is crucial to avoid crashes.

In Fig. 5.11b and Fig. 5.11c we compare our *improved* method directly with our *base* method. We encircled in green and yellow the areas of interest that showcase the difference/similarity between the 2 methods.

First we encircled in green a part of the Safe Corridor where our method performs clearly better due to the fact that it is shape-aware and it can sense the surroundings before creating a polyhedron. In the case of the *base* method (Fig. 5.11b), one can see that in this tight corner (encircled in green), there are still free spaces that are not covered by the polyhedra of the Safe Corridor. Whereas in our case, the decomposition covers the free space perfectly, due to the fact that the algorithm recognizes that there is a sharp corner and doesn't generate a polyhedron with faces not parallel to the x and y directions.

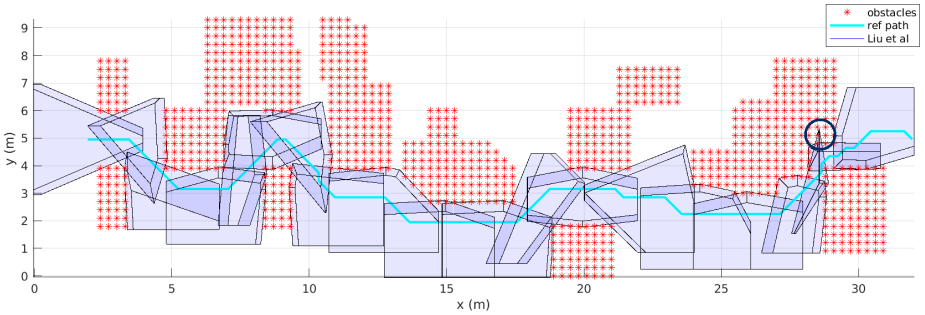
In the area encircled in yellow, we show our *base* method and our *improved* method perform in the presence of obstacles that have faces that are not parallel to the x or y directions i.e. *sloped* faces. The ability to cover free space around obstacles of arbitrary *slopes*/shapes is a measure of genericness of the method. Both methods succeed in capturing the slope in the polyhedron close to it. This is to show that while our *improved* method performs well in environments with rectangular shaped obstacles, it also performs just as well as *base* method when the obstacles have *sloped* faces.

After finding the Safe Corridor, any method can be used to generate an optimal feasible trajectory such as the method presented in chapter 6 (page 95) or [88], [146].

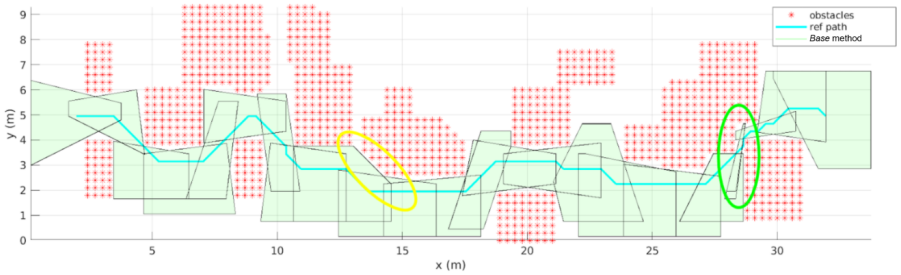
5.7 . Conclusion

In this work, we presented a novel method for the decomposition of free 3D space into overlapping convex polyhedra (that we call Safe Corridors or SC). The method leverages a voxel grid representation of the environment to ensure that no intersection exists between obstacles and the future SC. We introduced the concept of an expanding convex grid in which we inscribe a convex polyhedron that changes as the convex grid is expanded. This method is then used to create a SC by sampling a reference path into points and generating convex polyhedra around these points.

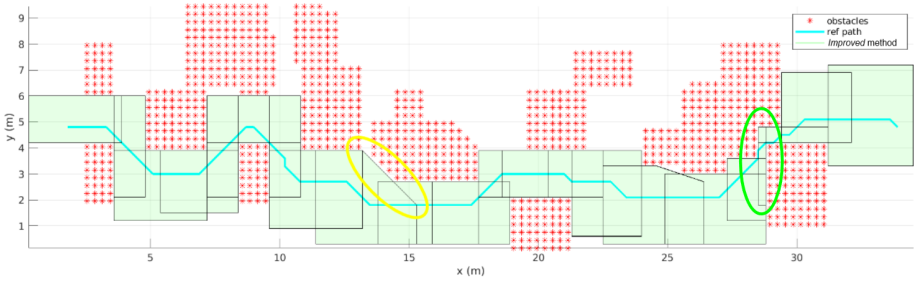
We compared our method (the *base* and *improved* versions) to the state-of-the-art [88] in terms of volume covered, number of constraints per polyhedron, number of polyhedra per SC, computation time and SC safety. Our method is outperformed by the state-of-the-art in terms of volume covered and computation time (while remaining largely within real time constraints), and are roughly similar in terms of number of polyhedra per SC. However, it outperform the state-of-the-art in terms of number of constraints per polyhedron. Finally, in terms of SC



(a) Liu et al's method [88]



(b) *Base* method



(c) *Improved* method

Figure 5.11: An overhead view of the Safe Corridor generated around a path using Liu et al. [88], our *base* method and our *improved* method. The place where a polyhedron generated by Liu et al.'s method penetrates the space between the obstacle points of the downsampled point cloud is encircled in **dark blue**. We encircled in green a place where our *improved* method generates a higher quality decomposition then the *base* method. We encircled in yellow a place where a slope in the obstacles is captured by our *improved* method and our *base* method to show that the improvement shown in the green ellipse doesn't hinder the genericness of our *improved* method in covering space efficiently around sloped obstacles (encircled in yellow).

safety, our method is able to guarantee that no intersection exists between the SC and the real world obstacles which is crucial to avoid crashes during navigation. This is not the case for [88] which can generate SCs that intersect with real world obstacles. We also showed how our *improved* version of the method generates SCs with significantly less constraints per polyhedron compared to our *base* version of the method, which translates into lower trajectory optimization time.

6 - SINGLE-AGENT PLANNING - STATIC ENVIRONMENT

In this work, we propose a new planning scheme for high-speed flight in an unknown static environment while taking into account drag forces. Drag forces become non-negligible at high speeds and may lead to unfeasible trajectories. The method leverages a new Mixed-Integer Quadratic Program/Model Predictive Control formulation that allows to easily account for drag forces. This formulation makes use of a state-of-the-art Safe Corridors generator to guarantee safety. It uses state-of-the-art mapping algorithms and solvers to achieve a higher computational efficiency than similar state-of-the-art methods. To the best of our knowledge, our method is the first high-speed planner that generates safe trajectories while accounting for drag. The proposed method is tested in simulation and compared to similar state-of-the-art methods for planning in unknown environments in terms of quality and computation time.

6.1 . Introduction

High-speed navigation in complex environments has numerous applications [26] such as infrastructure inspection [12], exploration [13], search and rescue [115] and cinematography [18]. In these applications, high-speed planning is favorable: in comparison with low-speed planning, high-speed planning allows to cover a larger distance on a single battery charge.

The motivation behind this work is to create a planner for high-speed flight (mean: $3.5\ m/s$, max: $6\ m/s$) in unknown environments while taking into account drag forces and the limitations of embedded computing.

The chapter is organized as follows: we will first present briefly the current state-of-the-art methods for multirotor planning and state the main contributions of our work. We will then describe the multirotor model used for planning and how it accounts for drag forces. The planning method is then presented with each step explained in a separate subsection. Finally, the simulation results are shown and comparisons with the state-of-the-art are done.

6.1.1 . Related work

Most of the current state-of-the-art planning methods rely on the differential flatness property of quadrotors [101]. Differential flatness allows expressing all

states and inputs of the quadrotor in terms of its position and yaw angle (flat outputs), and their derivatives. This allows them to simplify the planning problem by transforming the quadrotor dynamics to an integrator model. To generate smooth trajectory, they minimize the squared euclidean norm of a derivative of the position [101], [20], [129]. Some of these methods take into account static obstacles when solving the optimization problem, while others account for them after solving it. None of these methods take into account drag or the unknown aspect of environments.

Other planning methods use motion primitives or closed-form solutions to transform the planning problem into a graph search in the state space [87], [89], [90], [169]. These methods usually require a computationally expensive search in order to be able to generate complex maneuvers around obstacles.

There are methods that take into account obstacles when solving for the optimal trajectory by using Euclidean Signed Distance Fields (ESDF) that transform the 3D space into voxels encoding the distance to the nearest obstacle [116], [117], [55], [45]. These methods generate problems that are non convex, and lead to local minima problems. Others use successive convexification to solve the non-convex problem that results from including obstacles in the problem formulation [99]. These methods rely heavily on a good initialization and may have some convergence difficulties while handling complex environments.

In [86], obstacles must be decomposed into overlapping convex shapes which is non trivial when dealing with cluttered environments. Another set of methods rely on a convex decomposition of the free space: in [88], [161], [47], [78], [131], [146], polynomials or Bézier curves were used, and drag forces are not accounted for.

6.1.2 . Contribution

While some control methods account for drag [69], none of the aforementioned planning methods include drag forces in the planning framework. Our work takes inspiration from previous planners that use Safe Corridors, notably [88] and [146]. The main contributions of our work are:

- The integration of our state-of-the-art method for Safe Corridor generation (presented in section 5 - page 69) in a new planning framework.
- A novel planning algorithm that takes into account drag forces, and is significantly more computationally efficient than similar state-of-the-art methods.

We validate the feasibility of our generated trajectories by testing them in the simulation engine Airsim [138]. The physics engine of Airsim is state-of-the-art

and takes into account drag forces.

6.2 . MAV Model

Table 6.1: Nomenclature

g	gravity
m	multirotor mass
$\boldsymbol{p}$	position vector x, y, z in the world frame
$\boldsymbol{v}$	velocity vector v_x, v_y, v_z in the world frame
$\boldsymbol{a}$	acceleration vector from thrust and gravity in the world frame
$\boldsymbol{j}$	jerk vector j_x, j_y, j_z in the world frame
$\boldsymbol{z}_W$	world frame z
$\boldsymbol{z}_B$	body frame z
$\boldsymbol{R}$	rotation matrix from body to world frame
$\boldsymbol{D}$	quadratic drag matrix
ϕ	roll angle
θ	pitch angle
ψ	yaw angle
c_{cmd}	total thrust command

We assume a low-level controller that allows for controlling the attitude and thrust. We use the nomenclature defined in Table 6.1. The equations of motion are (Figure 6.1):

$$\dot{\boldsymbol{p}} = \boldsymbol{v} \tag{6.1}$$

$$\dot{\boldsymbol{v}} = -g\boldsymbol{z}_W + \frac{c_{cmd}}{m}\boldsymbol{z}_B - \boldsymbol{R}\boldsymbol{D}\boldsymbol{R}'\boldsymbol{v}||\boldsymbol{v}||_2 \tag{6.2}$$

$$\dot{\phi} = \dot{\phi}_{cmd} \tag{6.3}$$

$$\dot{\theta} = \dot{\theta}_{cmd} \tag{6.4}$$

$$\dot{\psi} = \dot{\psi}_{cmd} \tag{6.5}$$

We reformulate the five equations of motion to get the following:

$$\begin{aligned} \dot{\boldsymbol{p}} &= \boldsymbol{v} \\ \dot{\boldsymbol{v}} &= \boldsymbol{a} - \boldsymbol{D}_{lin\_max}\boldsymbol{v} \\ \dot{\boldsymbol{a}} &= \boldsymbol{j} \end{aligned} \tag{6.6}$$

$\boldsymbol{D}_{lin\_max}$ a diagonal matrix representing the maximum possible linear drag coefficient in all directions (identified offline). We replace the quadratic drag force

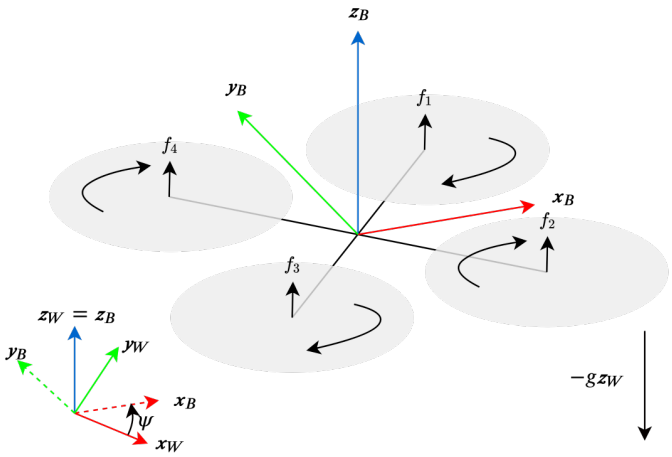


Figure 6.1: Schematics of the multirotor model with the used coordinate systems.

model with a linear worst-case scenario model at any given direction, because this will allow us to account for drag while maintaining the linearity of the dynamics. The linearity of the model allows us to solve the MPC problem using solvers that are very efficient for linear constraints. External forces (such as disturbances identified online) can also be added to the model to further guarantee feasibility, without affecting its linearity (and thus computation time). The online estimation of such forces can be done using [111].

Drag coefficients offline identification

We first identify the z component of the quadratic drag matrix ($\boldsymbol{D}_{3,3}$) by moving at full thrust $c_{cmd,max}$ in the $\boldsymbol{z}_W$ direction ($v_x = v_y = 0$). The velocity approaches a maximum constant value $v_{max,z}$ asymptotically. $\boldsymbol{D}_{3,3}$ is then calculated using equation (6.2) at the steady state ($\dot{v}_z = 0$): $\boldsymbol{D}_{3,3} = (-g + \frac{c_{cmd,max}}{m})/v_{max,z}^2$. Note that in this experiment the rotation matrix $\boldsymbol{R}$ is the identity matrix and $\boldsymbol{z}_W = \boldsymbol{z}_B$. Then we identify $\boldsymbol{D}_{2,2}$ by moving at full thrust in the y direction ($v_z = v_x = 0$). In this case we need to tilt only around the x axis of the body frame such that a fraction of $c_{cmd,max}$ compensates the gravity ($\dot{v}_z = 0$), and the remaining force of $c_{cmd,max}$ is in the y direction. The velocity approaches a maximum constant value $v_{max,y}$ asymptotically. By using equation (6.2), we get 1 equation ($\dot{v}_y = 0$) with the only unknown $\boldsymbol{D}_{2,2}$. We identify $\boldsymbol{D}_{1,1}$ similarly to $\boldsymbol{D}_{2,2}$. Finally, the elements on the diagonal of $\boldsymbol{D}_{lin\_max}$ are all chosen equal to: $\frac{\max(\boldsymbol{D}_{1,1},\boldsymbol{D}_{2,2},\boldsymbol{D}_{3,3})}{\min(v_{max,x},v_{max,y},v_{max,z})}$.

6.3 . The Method

The planner takes as input the position of the multirotor, a voxel grid partitioning space into free, occupied, and unknown voxels (centered at the robot position), and a goal in 3D space. It then proceeds to plan/explore by finding a global path and optimizing it locally in an iterative fashion until it reaches the goal.

The method is divided into 4 steps:

1. Generating a global path.
2. Creating a Safe Corridor around the global path.
3. Generating a safe local reference trajectory.
4. Solving the Mixed-Integer Quadratic Program (MIQP)/Model Predictive Control (MPC) problem.

All these steps are run at constant rates in a loop (Fig. 6.2). In the first step, we generate a global path to the goal. At every iteration, this step takes as input the current position of the robot and the goal. The second step generates a Safe Corridor using the last generated global path ($last$ in Fig. 6.2), and the Safe Corridor and MIQP/MPC solution generated in the previous iteration ($k-1$). The third step generates a local reference trajectory using the last generated global path, the Safe Corridor of the current iteration (k), and the MIQP/MPC solution and local reference solution generated in the previous iteration ($k-1$). The fourth step generates the optimal and safe trajectory using the Safe Corridor and local reference trajectory of the current iteration (k). The final trajectory is finally fed to the controller for execution.

Steps 2-4 are run at the same frequency ($10\mathrm{Hz}$) whereas step 1 can be run at an equal or lower frequency than $10\mathrm{Hz}$. The controller can be run at an equal or greater frequency than $10\mathrm{Hz}$. Our method has the following properties:

- **Resolution Completeness:** we guarantee that we will find a path from point A to point B as long as there exists a path and the local grid map is big enough and fine enough. This property is inherited from Jumping Point Search (JPS) [58] (used to find a global path). Assuming no external disturbances affect the dynamics of the multirotor, we can simply accelerate at the beginning of the first segment of the path and stop at the end of the segment. Then we turn to the angle of the next path segment and execute it until we reach its end (where we stop and turn to face the next segment). In this manner we can prove that the resolution completeness is inherited by the dynamically generated trajectory, assuming no external disturbances.
- **Feasibility:** we guarantee feasibility by accounting for drag and limiting input constraints to the quadrotor's dynamical limits.

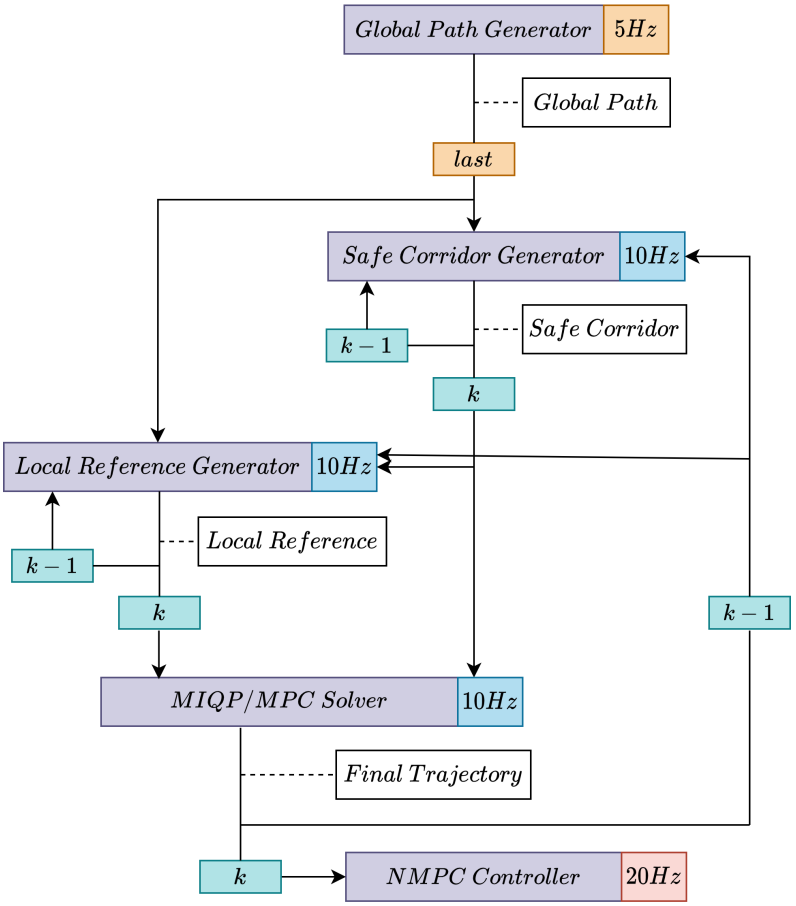


Figure 6.2: We show the global pipeline of our planning framework at iteration k. The last generated global path is always used at every iteration, and some steps use results generated at step $k-1$.

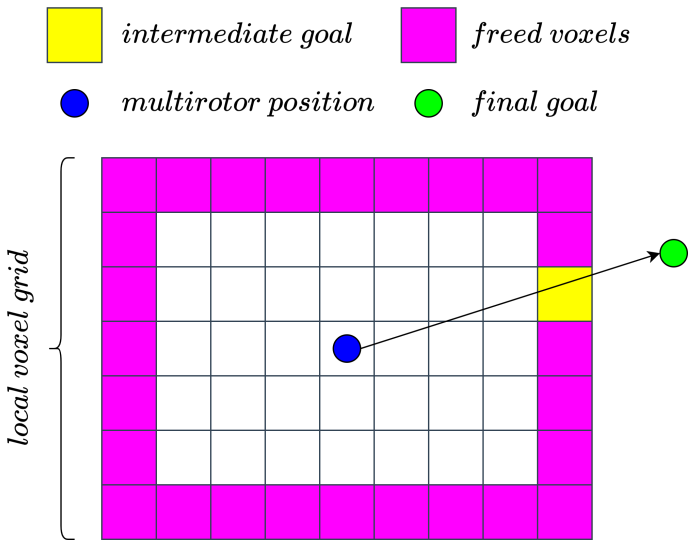


Figure 6.3: Choosing the intermediate goal when the final goal is outside the local voxel grid.

- **Smoothness and Local Optimality:** we get optimality in the sense of minimizing the jerk on a given time horizon.
- **Safety:** we guarantee the trajectory is collision-free through Safe Corridors and by using the Distance Map Planner (DMP) [130] on top of JPS, which pushes the trajectory away from obstacles (assuming perfect localization and control).

6.3.1 . Generating a global path

In this step, we will generate a global path from the robot position to the goal using a local voxel gird generated by [147] using a point cloud.

The local voxel grid spans a given volume. If the goal is outside this volume, we find the intersection between the borders of the grid and the line from the center of the multirotor to the goal (Fig. 6.3). This will determine the intermediate goal to which we plan. We free the intermediate goal voxel as well as all voxels on the border of the voxel grid to guarantee that we will find a path to the intermediate goal (if there exists a path).

We use Jumping Point Search (JPS) [58] to find a feasible path between the position of the multirotor and the goal (or intermediate goal). JPS is a shortest path algorithm that preserves A*'s optimality, while potentially lowering the computation time by an order of magnitude. We then use the Distance Map Planner (DMP) [130], to generate a safer path. It uses the artificial potential field to push away the path from the obstacles (Fig. 6.4). Pushing away the trajectory

from obstacles not only provides a safety margin for disturbances/uncertainties, but also allows for better vision/coverage when turning corners i.e. to see/cover more space behind a corner before turning it. This increases the overall trajectory speed.

The DMP adds time to the computation but as long as the map size is small enough or the voxel size is big enough i.e. the total number of voxels in the grid is small enough, the combined computation time of both JPS and DMP is lower than the MIQP/MPC solving time.

Note that other methods for path planning such as Rapidly exploring Randomized Tree (RRT*) [72] or Probabilistic Roadmap (PRM) [16] can be used to find the global path. However, we chose JPS since it guarantees optimality in finite time (in contrast to RRT* and PRM), and performs better than RRT* and PRM in terms of computation time.

6.3.2 . Creating a Safe Corridor around the global path

After finding a path, we decompose the space around the path into overlapping convex polyhedra (Safe Corridor). Many methods exist in the literature to create Safe Corridors [88]. We use the algorithm described in section 5 (page 69) as it provides better safety guarantees and a lower solving time for the MIQP/MPC. It takes a voxel grid with occupied, free, and unknown voxels, and a path around which we would like to find a Safe Corridor. It returns a series of overlapping polyhedra covering only the free space (Fig. 6.4).

At the first planning iteration, we find the convex polyhedron around the voxel containing the starting position of the global path (seed voxel). Then we find the intersection between the global path and the convex polyhedron, and find an additional polyhedron with the voxel containing the intersection as its seed. Sometimes, the voxel containing the intersection is the same seed voxel as the one used for the last polyhedron (which would result in a duplication of the same polyhedron). In this case, we move further along the global path to find the next closest voxel outside the last polyhedron. We then use this voxel as a seed for the next polyhedron. This algorithm is repeated until we reach the maximum number of polyhedra P_{hor} (polyhedra horizon).

At the next planning iterations, we first determine the minimum number of the polyhedra generated at the previous iteration (P_{min}) that contain the trajectory generated by the MIQP/MPC at the previous iteration, with preference given to the newest generated polyhedra. We then generate $P_{rem} = P_{hor} - P_{min}$ polyhedra using the aforementioned algorithm and add them to the P_{min} polyhedra. These polyhedra will be used by the MIQP/MPC to generate a safe trajectory at the current iteration.

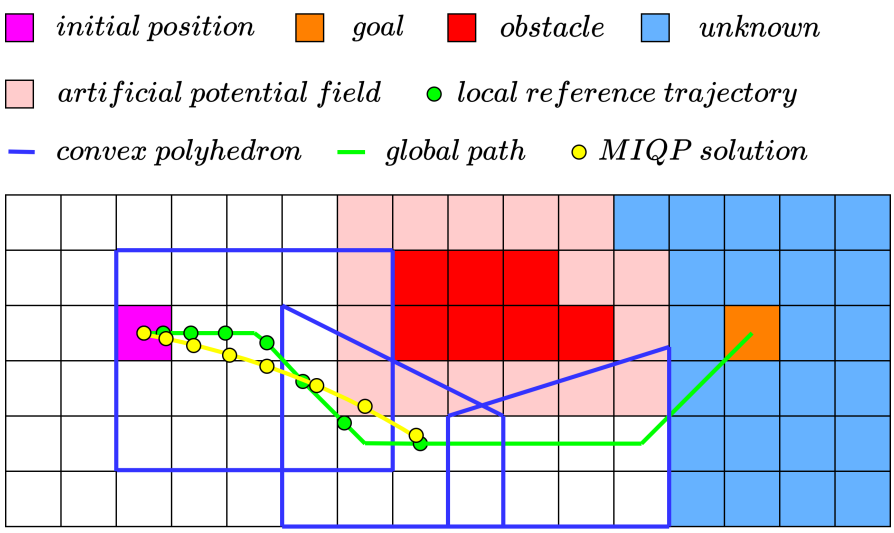


Figure 6.4: We show the first planning iteration of our algorithm. The global path is generated using JPS and DMP. DMP uses the artificial potential field (pink) to push the path away from the obstacles. The convex polyhedra (blue) span only the free space. A local reference trajectory is sampled from the global pushed path (green circles). It is then given with the overlapping convex polyhedra to the MIQP solver to generate a feasible trajectory (yellow circles).

6.3.3 . Generating a safe local reference trajectory

In this step, we use the global path and the Safe Corridor to generate a reference trajectory for the MIQP/MPC. We denote N the number of discretization steps and h the time step used in the MIQP/MPC.

We first determine the initial direction of the global path (first two nodes of the DMP), then we project the velocity vector of the robot along this direction (dot product). This will determine the initial sampling velocity: if the projection is negative (the multirotor is moving in the opposite direction to the global path direction), we set the initial sampling velocity to 0, otherwise it is kept as is. We then move along the global path for a time h (time step) with the initial velocity, and the found point is set as the first local reference point (closest green point to the initial position in Fig. 6.4). The sampling velocity is then updated such as $v_{samp,k+1} = \min(v_{samp,max}, v_{samp,k} + h \times a_{samp})$ with a_{samp} the acceleration of the sampling (chosen close to the maximum quadrotor acceleration), and $v_{samp,max}$ the maximum velocity of the sampling (chosen close to the maximum quadrotor velocity, which is limited by drag). $v_{samp,max}$ is set as an upper bound for the sampling velocity to keep the sampled trajectory close to feasibility. The next point is sampled starting from the last reference point and with the updated

sampling velocity. If a generated reference point is outside the Safe Corridor, the last reference point (which is inside the Safe Corridor) is duplicated and takes its place. We generate N reference points (the green points in Fig. 6.4).

At each iteration, after solving the MIQP/MPC, we check if the final state is close enough from the previous final reference point (within $thresh\_dist$). If yes, we regenerate a new local reference trajectory using the aforementioned algorithm starting from $x_{1,ref}$ (the closest green point to the initial position in Fig. 6.4). If no, wait another time step h to give the MPC enough time to reach the final reference point $x_{N,ref}$.

6.3.4 . Solving the MIQP/MPC problem

In this step, we generate a safe trajectory to be executed by the multirotor's controller using the safe local reference trajectory generated in section 6.3.3 (page 103) and the Safe Corridor generated in section 6.3.2 (page 102). We minimize the error to the reference trajectory as well as the jerk norm, which results in a smooth version of the local reference trajectory (Fig. 6.4).

At every iteration, the initial state $\boldsymbol{x}_0$ of the MPC is set to the first state $\boldsymbol{x}_1$ of the last generated trajectory. The terminal velocity $\boldsymbol{v}_N$ is set to $\mathbf{0}$ to make sure that the multirotor has a safe trajectory to execute in case subsequent MIQP optimizations fail to find a solution. We can also set $\boldsymbol{a}_N$ to $\mathbf{0}$, but this results in slower trajectories.

In case the solver fails to find a solution at a given iteration or the computation time exceeds the time step h, we skip the iteration (the solution is discarded), and at the next iteration, we solve the MIQP/MPC with the initial state $\boldsymbol{x}_2$ instead of $\boldsymbol{x}_1$. In case this also fails, we keep offsetting the initial position (which may reach $\boldsymbol{x}_N$ in the worst-case) until the solver converges to a solution within the time step h.

Dynamics

The state vector $\boldsymbol{x}$ is composed of the position $\boldsymbol{p}$, velocity $\boldsymbol{v}$ and acceleration $\boldsymbol{a}$, $\boldsymbol{x} = [\boldsymbol{p}\ \boldsymbol{v}\ \boldsymbol{a}]^T$; the control input is the jerk $\boldsymbol{u} = \boldsymbol{j}$; the dynamics $f(\boldsymbol{x}(t), \boldsymbol{u}(t))$ are defined by Eq. (6.6). The model is discretized using Euler or Runge-Kutta 4$^{\text{th}}$ order (RK4) to obtain the discrete dynamics $\boldsymbol{x}_{k+1} = f_d(\boldsymbol{x}_k, \boldsymbol{u}_k)$. RK4 is a more precise approximation of the continuous dynamics since the error in a single step h is $O(h^4)$, whereas Euler is $O(h^2)$. However, we choose the Euler method as it results in faster solving times without any noticeable difference in the generated trajectory.

State bounds

The acceleration resulting from the thrust and gravity, and the jerk obey the following equations at every discrete point k:

$$\sqrt{a_{x,k}^2 + a_{y,k}^2 + (a_{z,k} + g)^2} \leq a_{max} \tag{6.7}$$

$$a_{z,k} \geq a_{z,min} \tag{6.8}$$

$$\sqrt{j_{x,k}^2 + j_{y,k}^2 + j_{z,k}^2} \leq j_{max} \tag{6.9}$$

The velocity is limited by the drag forces. The bounds a_{max}, $a_{z,min}$ $(= -g)$ and j_{max} are imposed by the dynamical constraints. These constraints are quadratic. However, since we want them to be linear (faster solving time), each direction is decoupled and they are transformed into the following conservative constraints:

$$|a_{x,k}| \leq a_{x,max}, \quad |a_{y,k}| \leq a_{y,max} \tag{6.10}$$

$$a_{z,k} \leq a_{z,max}, \quad a_{z,k} \geq a_{z,min} \tag{6.11}$$

$$|j_{x,k}| \leq j_{x,max}, \quad |j_{y,k}| \leq j_{y,min}, \quad |j_{z,k}| \leq j_{z,max} \tag{6.12}$$

The decoupling of the dynamics results in not using the full dynamics of the multirotor. However we privilege computation time and feasibility over extremely high agility (which is not the case in the time optimal trajectory generation method presented in section 3 (page 33) where we prioritize agility). We choose the bounds of each direction such that:

$$\sqrt{a_{x,max}^2 + a_{y,max}^2 + (a_{z,max} + g)^2} \leq a_{max} \tag{6.13}$$

$$\sqrt{j_{x,max}^2 + j_{y,max}^2 + j_{z,max}^2} \leq j_{max} \tag{6.14}$$

Static obstacle avoidance

This is achieved by forcing every two consecutive discrete points (and thus the segment formed by them) to be in one of the overlapping polyhedra. Let's assume we have P overlapping polyhedra. They are described by $\{(\boldsymbol{A}_p, \boldsymbol{c}_p)\}$, $p = 0 : P-1$. The constraint that the discrete position $\boldsymbol{p}_k$ is in a polyhedron p is described by $\boldsymbol{A}_p.\boldsymbol{p}_k \leq \boldsymbol{c}_p$. We introduce binary variables b_{kp} (P variables for each $\boldsymbol{x}_k$, $k = 0 : N-1$) that indicate that $\boldsymbol{p}_k$ and $\boldsymbol{p}_{k+1}$ are in the polyhedron p. We force all the segments to be in at least one of the polyhedra with the constraint $\sum_{p=0}^{P-1} b_{kp} \geq 1$.

Typically, the number of polyhedra considered for optimization P_{hor} is 2 to avoid high solving times.

Formulation

We formulate our MPC under the following Mixed-Integer Quadratic Program (MIQP) formulation. The reference trajectory $x_{k,ref}$ is generated as described in section 6.3.3 (page 103). $\boldsymbol{R}_x$, $\boldsymbol{R}_N$ and $\boldsymbol{R}_u$ are the weight matrix for the discrete state errors without the final state, the weight matrix for the final discrete state error (terminal state), and the weight matrix for the input, respectively.

$$
\begin{aligned}
\underset{\boldsymbol{x}_k, \boldsymbol{u}_k}{\text{minimize}} \quad & \sum_{k=0}^{N-1} (||\boldsymbol{x}_k - \boldsymbol{x}_{k,ref}||^2_{\boldsymbol{R}_x} + ||\boldsymbol{u}_k||^2_{\boldsymbol{R}_u}) \\
& + ||\boldsymbol{x}_N - \boldsymbol{x}_{N,ref}||^2_{\boldsymbol{R}_N} && (6.15) \\
\text{subject to} \quad & \boldsymbol{x}_{k+1} = f_d(\boldsymbol{x}_k, \boldsymbol{u}_k), \quad k = 0 : N-1 && (6.16) \\
& \boldsymbol{x}_0 = \boldsymbol{X}_0 && (6.17) \\
& \boldsymbol{v}_N = \boldsymbol{0} && (6.18) \\
& |a_{x,k}| \leq a_{x,max} && (6.19) \\
& |a_{y,k}| \leq a_{y,max}, \quad a_{z,k} \leq a_{z,max} && (6.20) \\
& a_{z,k} \geq a_{z,min}, \quad |j_{x,k}| \leq j_{x,max} && (6.21) \\
& |j_{y,k}| \leq j_{y,min}, \quad |j_{z,k}| \leq j_{z,max} && (6.22) \\
& b_{kp} = 1 \implies \boldsymbol{A}_p \boldsymbol{p}_k \leq \boldsymbol{c}_p, \boldsymbol{A}_p \boldsymbol{p}_{k+1} \leq \boldsymbol{c}_p && (6.23) \\
& \sum_{p=0}^{P_{hor}-1} b_{kp} \geq 1 && (6.24) \\
& b_{kp} \in \{0, 1\} && (6.25)
\end{aligned}
$$

The MIQP is solved using the Gurobi solver [53].

6.4 . Simulation Results

The simulation is done using a quadrotor in Airsim [138], a photo-realistic state-of-the-art simulator. The state of the quadrotor is known. The obstacles are cylinders with a radius of $0.35\ m$ and span a $50\ m \times 50\ m$ area with a density of $0.1\ obs/m^2$. They are generated randomly, following a uniform distribution. The Gurobi solver is set to use one thread only as this resulted in faster computation times during our simulations. All testing is done on the Intel Core i7-9750H up to 4.50 GHz CPU, and for the GPU we use NVIDIA's GeForce RTX 2060 up to 1.62 GHz.

6.4.1 . Controller design

We control our quadrotor using a nonlinear MPC [71], with the **acados** toolkit [157]. The MPC minimizes the cost function:

$$J = \int_{t=0}^{T} ||\boldsymbol{x}(t) - \boldsymbol{x}_{ref}(t)||^2_{\boldsymbol{Q}_x} + ||\boldsymbol{u}(t) - \boldsymbol{u}_{ref}(t)||^2_{\boldsymbol{R}_u} dt + ||\boldsymbol{x}(T) - \boldsymbol{x}_{ref}(T)||^2_{\boldsymbol{P}} \tag{6.26}$$

We use the model described in section 6.2 (page 97) with $\boldsymbol{u} = [c_{cmd}\ \dot{\phi}_{cmd}\ \dot{\theta}_{cmd}\ \dot{\psi}_{cmd}]^T$ and $\boldsymbol{x} = [\boldsymbol{p}\ \boldsymbol{v}\ \phi\ \theta\ \psi]^T$. The sampling time is $h = 0.05s$ and the horizon $N_h = 15$ which gives a time horizon $T = 0.75s$. The weights are:

$$\boldsymbol{P} = \boldsymbol{Q}_x = diag(15, 15, 15, 0.01, 0.01, 0.01, 0, 0, 1) \tag{6.27}$$

$$\boldsymbol{R}_u = diag(0.05, 0.1, 0.1, 0.1) \tag{6.28}$$

All parameters are set by approximation/experimentation and may not be optimal. They provide however some measurement to the feasibility of the generated trajectory. We limit $|\phi| \leq 85\ deg$, $\theta \leq 85\ deg$, $|\dot{\phi}_{cmd}| \leq 120\ deg/s$, $|\dot{\theta}_{cmd}| \leq 120\ deg/s$ and $|\dot{\psi}_{cmd}| \leq 60\ deg/s$. These limits are determined by the physical dynamical limits of the multirotor.

Since the planning frequency is 10Hz and the control frequency is 20Hz, we interpolate linearly the reference trajectory generated by the planner (which has a time step of $100\ ms$) to get the reference trajectory of the controller (which has a time step of $50\ ms$). This means that a reference point will be added in the middle of every 2 consecutive reference points generated by the MIQP solver.

6.4.2 . Voxel grid generation

We use a lidar to have omnidirectional coverage, which also can be provided by omnidirectional stereo cameras (with stereo matching). The point cloud is transformed into a voxel grid with occupied, free and unknown voxels using the state-of-the-art GPU accelerated voxelization algorithm [147]. We choose a voxel size (side length) of $0.3\ m$ and a grid size of $(size_x = 16\ m, size_y = 16\ m, size_z = 3\ m)$. We inflate the obstacles by one voxel for the convex decomposition and 2 voxels for the JPS global pathfinding. The inflation is done on the GPU using [147].

6.4.3 . Planner parameters

We choose the following parameters: $N = 9$, $h = 100\ ms$, $a_{x,max} = a_{y,max} = 0.7g$, $a_{z,max} = 0.4g$, $a_{z,min} = -g$, $j_{x,max} = j_{y,max} = j_{z,max} = 15\ m/s^2$, $v_{max,samp} = 6m/s$, $a_{samp} = 7m/s^2$ $\boldsymbol{D}_{lin\_max} = diag(1, 1, 1)$, $thresh\_dist = 0.35\ m$, $P_{hor} = 2$. The weight matrices are diagonal:
$\boldsymbol{R}_x = \boldsymbol{R}_N = diag(100, 100, 100, 0, 0, 0, 0, 0, 0)$ and $\boldsymbol{R}_u = diag(0.01, 0.01, 0.01)$.
Some of these parameters are identified from the quadrotor's model in Airsim.

The DMP planner pushes the JPS path $0.6\ m$ away from obstacles (when possible). The planning frequency for the global path planning is 5Hz and that of the safe local trajectory generation, Safe Corridor generation, and MIQP/MPC optimization is 10Hz. However, due to the low computation time (maximum computation time of $19.7\ ms$), it can be increased up to 50Hz. Increasing the planning frequency allows to react faster to changes in the environment resulting in faster trajectories.

6.4.4 . Comparison with the state-of-the-art

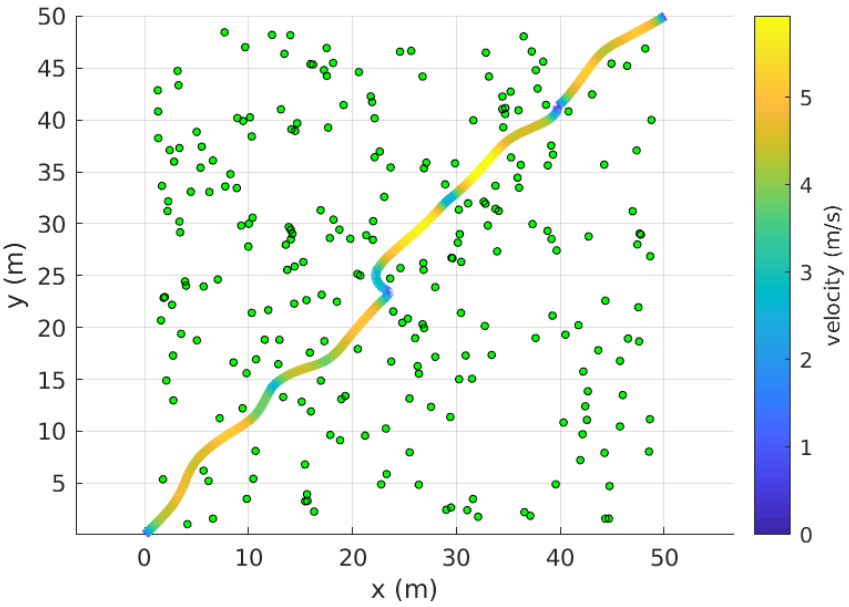
Table 6.2: Comparison between FASTER [146] and our planner on 10 randomly generated maps of size $50\ m \times 50\ m$ and with obstacle density $0.1\ obst/m^2$. The better performer is shown in bold. We show the **mean/max** values of each metric.

	Success	Flight distance	Flight velocity	Comp. time
FASTER [146]	10/10	**81.2/93 m**	**3.55/6.14 m/s**	20.2/102.3 ms
Our planner	10/10	83.2/92.5 m	3.45/5.95 m/s	**6/19.8 ms**
Difference	-	+2.5/-0.5 %	-3/-3 %	-336/-516 %

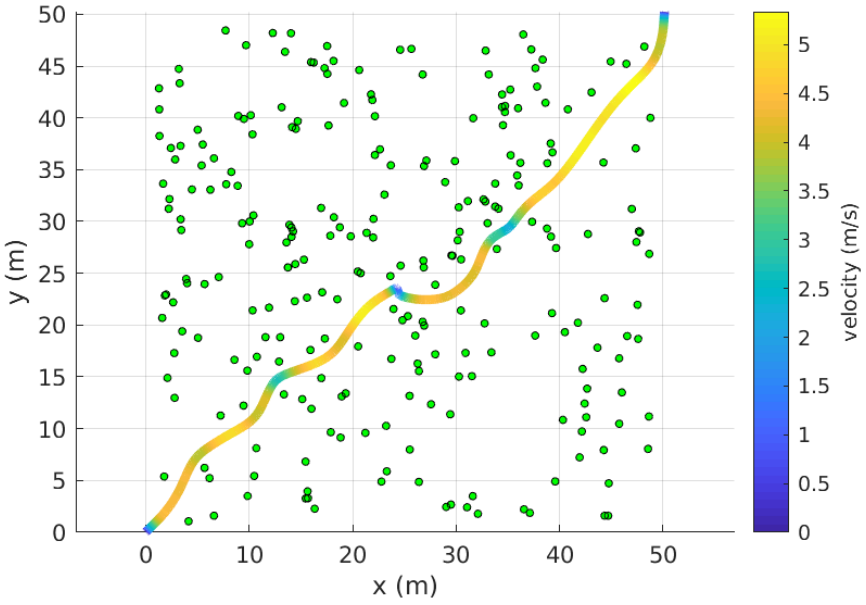
We compare our planner with the state-of-the-art FASTER planner [146]. It was fine-tuned to get a very good performance in terms of controller tracking and computation time.

The comparison is done using 10 randomly generated maps of obstacles which span a $50\ m \times 50\ m$ area with a density of $0.1\ obs/m^2$. We show the trajectories generated by both methods on one of the maps in Fig. 6.5 and the overall results over all 10 maps in Table 6.2. Both methods are similar in terms of flight distance and flight velocity with a slight advantage to FASTER, which outperforms our method by 2.5% in mean flight distance and 3% in mean flight velocity. However, in terms of computation time, our method far outperforms FASTER, with an advantage of 336% for the mean computation time and 516% for the maximum computation time. This renders our method much more suitable for low compute embedded systems, which was one of the main objectives of this work.

In Fig. 6.6 we show the breakdown of the computation time of our planner. The generation of overlapping convex polyhedra i.e. Safe Corridor takes on average $0.15\ ms$, has a median of $0.178\ ms$ and a max of $0.607\ ms$. The generation of a global path through JPS and DMP takes on average $2.07\ ms$, has a median of $1.948\ ms$ and a max of $7\ ms$. The MIQP/MPC solver takes on average $3.86\ ms$, has a median of $3.68\ ms$ and a max of $18.2\ ms$. The total planner takes on average $6\ ms$, has a median of $5.6\ ms$ and a max of $19.8\ ms$. Since the global path planning is run at 5Hz, its mean contribution to the total planning time (run



(a) FASTER [146]



(b) Our planner

Figure 6.5: The trajectories and velocity of the quadrotor generated by FASTER [146] and our planner on the same map.

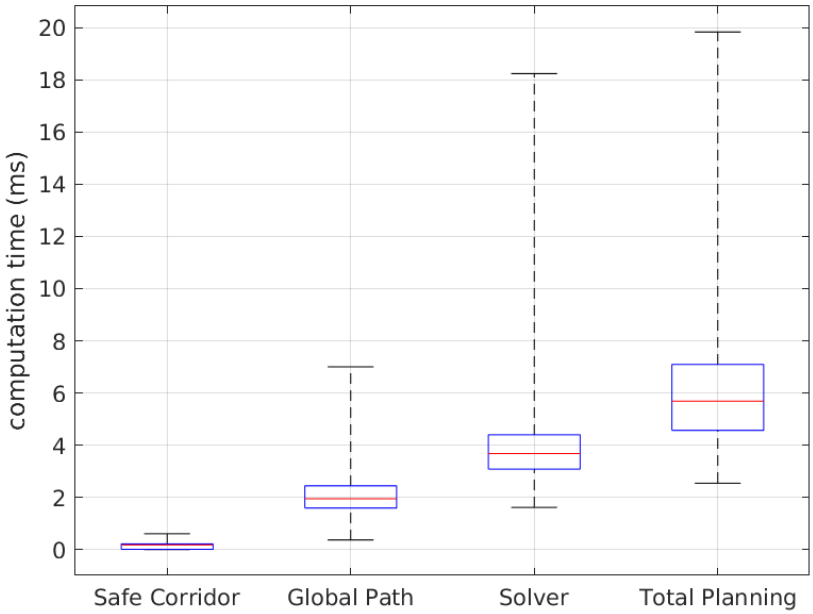


Figure 6.6: Boxplot of the computation times of the different functional blocks of the planner as well as the total computation time of the planner. The red segment represents the median. The lower and upper bounds of the box represent the 25^{th} and 75^{th} percentile respectively, and the lower and upper whiskers represent the minimum and maximum respectively.

at 10Hz) is halved. Setting up the Gurobi MIQP/MPC solver at every planning iteration takes on average $1\ ms$. The generation of a safe local trajectory (step 3) takes negligible time.

During our 10 simulated flights, the worst-case computation time for the Safe Corridor generation, global pathfinding, and MIQP/MPC solver don't happen at the same time, otherwise, the maximum total planning time would be the sum of the 3 maximum computation times + solver setup time. If we assume the worst-case scenario i.e. worst-case computation time in each step simultaneously (which can be seen in Fig. 6.6), the maximum total planning time would become $26.8\ ms$. This means that we can run our planner at 30Hz in the worst-case scenario.

If we take an embedded system like the NVIDIA Jetson Nano [114] whose CPU runs at 1.43GHz i.e. $3.14\times$ slower then the turbo-boosted clock of our CPU (4.5GHz), the theoretical worst case computation time would be $3.14 \times 26.8 = 82.4\ ms$. This means that the results produced in this work can be reproduced on a low-cost embedded system since the worst-case computation time is lower than the planning period which is $100\ ms$ (since the planner runs at 10Hz).

6.5 . Conclusion

In this work, we presented a novel method for high-speed planning in unknown environments that leverages the state-of-the-art in mapping (voxelization) and convex decomposition of free space to produce similar results to the state-of-the-art at a fraction of the computation cost. This makes it suitable for low-powered/low-cost embedded systems. The proposed planning method is, to the best of our knowledge, the first in its class to account for drag forces which adds guarantees to the feasibility of the trajectory and allows for better exploitation of the full dynamics of the multirotor. We also tested our planner in a state-of-the-art simulation environment (Airsim) and compared it with a similar state-of-the-art method. Our method showed similar results while being significantly more computationally efficient.

7 - SINGLE-AGENT PLANNING - DYNAMIC ENVIRONMENT

in this work, we propose a new method for multirotor planning in dynamic environments. the environment is represented as a Temporal Occupancy Grid which gives the current as well as the future/predicted state of all the obstacles. the method builds on our previous works in safe corridor generation and multirotor planning to avoid moving and static obstacles. it first generates a global path to the goal that doesn't take into account the dynamic aspect of the environment. we then use temporal safe corridors to generate safe spaces that the robot can be in at discrete instants in the future. finally we use the temporal safe corridors in an optimization formulation that accounts for the multirotor dynamics as well as all the obstacles to generate the trajectory that will be executed by the multirotor's controller. we show the performance of our method in simulations.

7.1 . Introduction

Multirotor planning in dynamics environments has many real world industrial, humanitarian and military applications. That's why recent research efforts have been focused on providing solutions or solution elements to this challenging problem.

A static environment has been the main assumption of multiple multirotor planning methods [12], [13], [20], [146], [47], [46], [88], [101]. It is the goal of this work to present a new multirotor planner for dynamic environments.

7.1.1 . Related work

Some works in the literature have addressed the problem of multirotor planning in dynamic environments using different approaches. Some consider only cooperative dynamic agents i.e. multi-agent planning, while assuming that the rest of the environment is static [120], [2], [70], [175], [174]. We will only discuss the works done where the dynamic obstacles are considered non-cooperative since this is the case that we treat in this work.

In [90], the authors propose a search-based method to avoid collision with all kind of obstacles (dynamic obstacles, planning agents and static obstacles). The state-space is discretized in space and time, then a graph search is used to find the set of trajectories that avoid all obstacles and take the robot closer to the goal.

However, search-based methods in general result in a high computation time when planning complex maneuvers due to the curse of dimensionality. This renders them non suitable for real-time embedded applications.

In [85], the authors model all obstacles (static and dynamic) as ellipses and include them in a non convex model predictive control formulation that keeps the planned discrete points outside the obstacle ellipses. While this approach may be suitable for a specific type of dynamic obstacles, decomposing the whole environment (including static obstacles) into ellipses is not trivial. One can imagine a tree with multiple branches, each branch also branching out into multiple branches. How to determine the number of ellipses that represent the tree or which points belong to the same ellipse? One would need to compromise between a heavy computational cost or an extremely inefficient representation. Furthermore, decomposing a complex environment into ellipses might result in an inefficient representation or a large number of ellipses, which when added as constraints to the planner increase computation time.

Finally, in [144] the authors present a method for avoiding all types of obstacles (static and dynamic). They use a combination of an optimization method and a search-based method, where the output of the search-based method is given as an initial guess for the optimization method. This choice is made due to the fact that the optimization problem is non-convex and requires a good initial guess. This method represents obstacles as polyhedron. This representation is not trivial to generate from sensor measurements (camera images or lidar pointclouds), and can incur a heavy computational burden i.e. higher computation time for the planner.

7.1.2 . Contribution

The main contribution of this work is a new planning framework for dynamic environments. The method takes as input a Temporal Occupancy Grid i.e. a series of occupancy grids that represent all obstacle positions at discrete points in finite future time. We use the Temporal Occupancy Grid to generate Temporal Safe Corridors which we then use to generate a trajectory that avoids all obstacles in a Model Predictive Control (MPC) fashion.

7.2 . Agent Model

We use the same simplified/linearized model presented in chapter 6 (page 95) for the planning agent (Tab. 7.1). The jerk j is the input of the system:

Table 7.1: Nomenclature

$\boldsymbol{p}$	position vector x, y, z in the world frame
$\boldsymbol{v}$	velocity vector v_x, v_y, v_z in the world frame
$\boldsymbol{a}$	acceleration vector from thrust and gravity in the world frame
$\boldsymbol{j}$	jerk vector j_x, j_y, j_z in the world frame
$\boldsymbol{D}_{lin\_max}$	linear drag matrix

$$\begin{aligned} \dot{\boldsymbol{p}} &= \boldsymbol{v} \\ \dot{\boldsymbol{v}} &= \boldsymbol{a} - \boldsymbol{D}_{lin\_max}\boldsymbol{v} \\ \dot{\boldsymbol{a}} &= \boldsymbol{j} \end{aligned} \tag{7.1}$$

7.3 . The Planner

The planner take as input a Temporal Occupancy Grid and then generates a trajectory that avoids all obstacles (static and dynamic) within the time horizon of the Temporal Occupancy Grid. The planner is divided into 3 steps:

1. Generating the Temporal Safe Corridor (TSC): in this step we take the Temporal Occupancy Grid and use it to generate a TSC.

2. Generating the reference trajectory: in this step we generate a global path from the position of the agent to the goal using the occupancy grid that represents all obstacles at the current instant. Then we use the path to generate a reference trajectory for the next step.

3. Generating the safe trajectory: in this step we use the TSC and the reference trajectory in a Model Predictive Control (MPC)/ Mixed-Integer Quadratic Program (MIQP) formulation to generate a safe and locally optimal trajectory.

7.3.1 . Generating the Temporal Safe Corridor

In this section we will discuss the generation and state-of-the-art of Temporal Occupancy Grids (TOG) as well as the generation of Temporal Safe Corridors (TSC).

Temporal Occupancy Grids

An occupancy grid partitions the space into regular cubes (voxels or cells) that are either occupied or free for the agent to move through. Temporal Occupancy Grids

(TOG) are a series of occupancy grids that represents the state of the environment at discrete points in time for a given time horizon. Obstacles moving through time are captured by the change in the voxels that are occupied/free as shown in Fig. 7.1. This representation of the environment is showing promise especially in the autonomous driving domain [63], [66], [137].

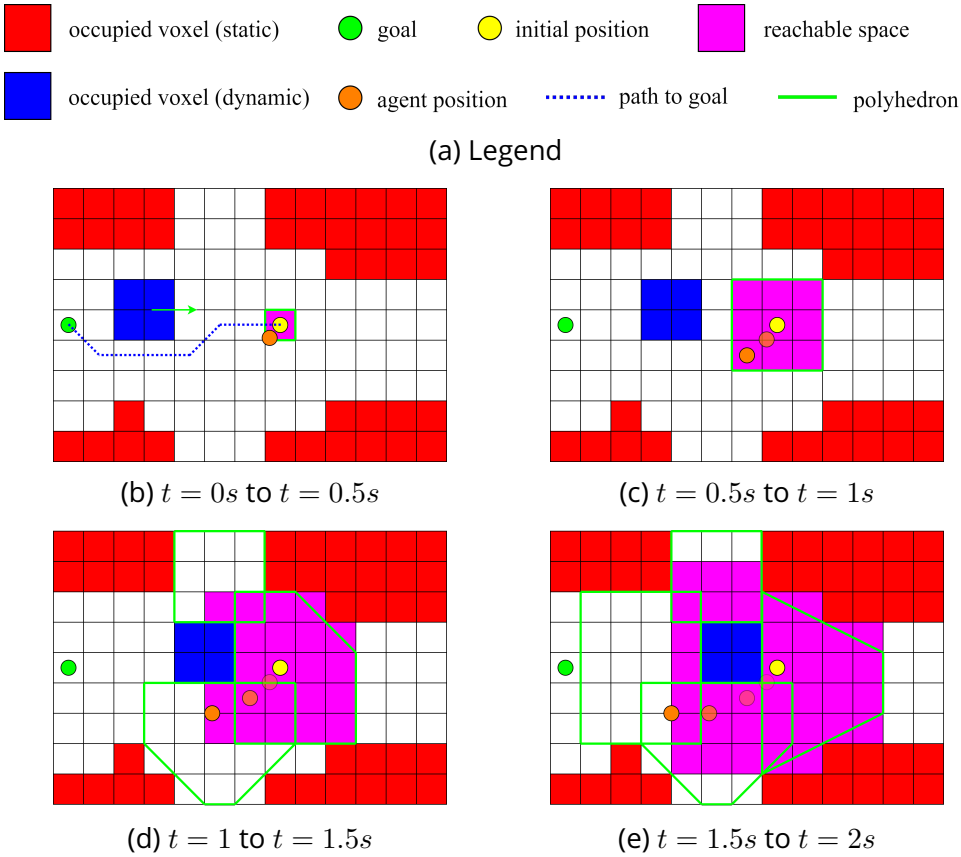
Temporal Safe Corridor

For each occupancy grid of the TOG, we generate a series of polyhedra that covers free space that the agent can be in at that instant in time. The free space that we cover is the intersection between the reachable space and the free space in the occupancy grid. The reachable space depends on the agent dynamics and grows bigger as we step forward in time as shown in Fig. 7.1. It allows to generate polyhedra that only cover it, reducing the total number of polyhedra used in the optimization problem i.e. reducing computation time. The reachable space is generated using a simplified approach: we use the maximum velocity that the agent can have an multiply it by the time step ($0.5\ s$ in the example shown in Fig. 7.1) to obtain the maximum distance (which we denote d_{reach}) that the agent can traverse in that time duration. The reachable space in voxels corresponds to the voxels that cover d_{reach} in all directions (a square with side length d_{reach}).

7.3.2 . Generating the reference trajectory

After generating the TSC, we generate a reference path that goes from the position of the agent to the goal. We use the first occupancy grid in the TOG (which corresponds to Fig. 7.1b) to generate the path using Jumping Point Search (JPS) [58] and Distance Map Planner (DMP) [130]. JPS finds the shortest path to the goal while DMP pushes the path from the obstacles when possible to create a clearance margin. Note that this path does not take into account the moving aspect of the dynamic obstacles and considers them as static.

The path is then used to generate a local reference trajectory for the MIQP/MPC. At each iteration we sample N points from the global trajectory to be used as reference for the N discrete positions of the MIQP/MPC. At every iteration l, these points are sampled using a starting point $\boldsymbol{x}^l_{0,ref}$. At the first iteration this starting point is the position of the planning agent. From the starting position, we move along the global path at the sampling speed v_{samp} (user input) for a time duration h, where h is the time step of the MIQP/MPC. The point at which we arrive is the second reference point $\boldsymbol{x}^l_{1,ref}$. We continue sampling in the same fashion until we reach $\boldsymbol{x}^l_{N,ref}$. At the subsequent iterations, we use the optimal trajectory generated by the last step (MIQP/MPC) of the previous iteration: we check if the final state is close enough from the previous final reference point (within $thresh\_dist$). If yes, the local reference trajectory is generated with the above-mentioned algo-



(a) Legend

(b) $t = 0s$ to $t = 0.5s$

(c) $t = 0.5s$ to $t = 1s$

(d) $t = 1$ to $t = 1.5s$

(e) $t = 1.5s$ to $t = 2s$

Figure 7.1: We show an example of a Temporal Occupancy Grid as well as a Temporal Safe Corridor over a duration of 2 seconds and a time step of $0.5s$. This means that the TOG is composed of 4 occupancy grids and the TSC is composed of 4 Safe Corridors. Each occupancy grid represents the occupied voxels during the whole duration of $0.5s$ i.e. all the position that the obstacle occupy during that duration. We show the occupied static obstacles in **red** and the occupied dynamic voxels in **blue**. The direction of movement of the obstacle is shown with a green arrow in Fig. 7.1b. The initial position of the agent is shown as a **yellow** circle, and the current position is shown in **orange** (with previous positions being more transparent to indicate a position in the past). The goal that we want to reach is shown in **green**. The reachable space is shown in **magenta**. The polyhedra composing the Safe Corridors are shown in **green**. They are generated using the method described in chapter 5 (page 69). The path generated in section 7.3.2 (page 116) is shown in **dotted blue**. At each time step, the Safe corridor is generated to cover the reachable space. When planning, the position of the agent at each time step is constrained to be inside one of the polyhedra of the corresponding Safe Corridor.

rithm starting from $\boldsymbol{x}^l_{0,ref} = \boldsymbol{x}^{l-1}_{1,ref}$. If no, the local reference trajectory generated at the previous iteration is used for the current iteration i.e. $\boldsymbol{x}^l_{0,ref} = \boldsymbol{x}^{l-1}_{0,ref}$.

7.3.3 . Solving the MIQP/MPC problem

In this step, a collision-free optimal trajectory is generated. It uses the local reference trajectory generated in section 7.3.2 (page 116) and the Temporal Safe Corridor generated in section 7.3.1 (page - 115). The error between the agent position and the local reference trajectory as well as the norm of the jerk are minimized. The resulting optimal trajectory gets us closer to the last sampled point (Fig. 7.1).

At every iteration l, the initial state $\boldsymbol{x}^l_0$ of the MPC is set to the second state $\boldsymbol{x}^{l-1}_1$ of the last generated trajectory (except for the first iteration where the initial state is the initial robot position). The terminal velocity $\boldsymbol{v}_N$ is set to $\mathbf{0}$.

In case the solver fails to find a solution at a given iteration or the computation time exceeds the time step h, we skip the iteration (the solution is discarded), and at the next iteration, we solve the MIQP/MPC with the initial state $\boldsymbol{x}_2$ instead of $\boldsymbol{x}_1$. In case this also fails, we keep offsetting the initial position (which may reach $\boldsymbol{x}_N$ in the worst case).

Dynamics

With the state $\boldsymbol{x} = [\boldsymbol{p}\ \boldsymbol{v}\ \boldsymbol{a}]^T$, the control $\boldsymbol{u} = \boldsymbol{j}$, and the dynamics $f(\boldsymbol{x}(t), \boldsymbol{u}(t))$ defined by Eq. (7.1), the model is discretized using Euler or Runge-Kutta 4$^{\text{th}}$ order (RK4) to obtain the discrete dynamics $\boldsymbol{x}_{k+1} = f_d(\boldsymbol{x}_k, \boldsymbol{u}_k)$. We choose the Euler method as it results in faster solving times (even though RK4 is more accurate). With a discretization step of h, the discretized dynamics become:

$$
\begin{aligned}
\boldsymbol{p}_{k+1} &= \boldsymbol{p}_k + h\boldsymbol{v_k} \\
\boldsymbol{v}_{k+1} &= \boldsymbol{v}_k + h(\boldsymbol{a}_k - \boldsymbol{D}_{lin\_max}\boldsymbol{v_k}) \\
\boldsymbol{a}_{k+1} &= a_k + h\boldsymbol{j_k} \\
\boldsymbol{x}_k &= [\boldsymbol{p}_k\ \boldsymbol{v}_k\ \boldsymbol{a}_k]^T \\
\boldsymbol{u}_k &= \boldsymbol{j}_k
\end{aligned}
\tag{7.2}
$$

State bounds

The agent velocity is limited by the drag forces. The maximum bounds on the acceleration and the jerk in each direction are determined by the dynamics of the agent. We define $a_{x,max}$ and $a_{y,max}$ as the maximum L1 norm of the acceleration in the directions x and y. We define $a_{z,max}$ and $a_{z,min}$ as the maximum and minimum values respectively of the acceleration in the z direction. These values

are deduced directly from the maximum thrust that a multirotor can generate.

Finally, we define $j_{x,max}$, $j_{y,max}$ and $j_{z,max}$ as the maximum L1 norm of the jerk in the directions x, y and z respectively. These values represent the limits on the rotational dynamics of a multirotor.

Collision avoidance

This is achieved by forcing every discrete point k to be in one of the polyhedra of the corresponding SC of the TSC (SC_k). Let's assume we have P_k polyhedra in SC_k. They are described by $\{(\boldsymbol{A}_{kp}, \boldsymbol{c}_{kp})\}$, $p = 0 : P - 1$. The constraint that the discrete position $\boldsymbol{p}_k$ is in a polyhedron p is described by $\boldsymbol{A}_{kp}.\boldsymbol{p}_k \leq \boldsymbol{c}_{kp}$. We introduce binary variables b_{kp} (P_k variables for each $\boldsymbol{x}_k$, $k = 0 : N - 1$) that indicate that $\boldsymbol{p}_k$ are in the polyhedron p. We force all the points to be in at least one of the polyhedra with the constraint $\sum_{p=0}^{P_k-1} b_{kp} \geq 1$.

Formulation

We formulate our MPC under the following Mixed-Integer Quadratic Program (MIQP) formulation. We remove the superscript l (which indicates the number of the iteration) from the reference and state variables for simplification.

$$
\begin{aligned}
\underset{\boldsymbol{x}_k, \boldsymbol{u}_k}{\text{minimize}} \quad & \sum_{k=0}^{N-1} (||\boldsymbol{x}_k - \boldsymbol{x}_{k,ref}||^2_{\boldsymbol{R}_x} + ||\boldsymbol{u}_k||^2_{\boldsymbol{R}_u}) \\
& + ||\boldsymbol{x}_N - \boldsymbol{x}_{N,ref}||^2_{\boldsymbol{R}_N} && (7.3) \\
\text{subject to} \quad & \boldsymbol{x}_{k+1} = f_d(\boldsymbol{x}_k, \boldsymbol{u}_k), \quad k = 0 : N - 1 && (7.4) \\
& \boldsymbol{x}_0 = \boldsymbol{X}_0 && (7.5) \\
& \boldsymbol{v}_N = \boldsymbol{0} && (7.6) \\
& |a_{x,k}| \leq a_{x,max} && (7.7) \\
& |a_{y,k}| \leq a_{y,max}, \quad a_{z,k} \leq a_{z,max} && (7.8) \\
& a_{z,k} \geq a_{z,min}, \quad |j_{x,k}| \leq j_{x,max} && (7.9) \\
& |j_{y,k}| \leq j_{y,min}, \quad |j_{z,k}| \leq j_{z,max} && (7.10) \\
& b_{kp} = 1 \implies \boldsymbol{A}_{kp}\boldsymbol{p}_k \leq \boldsymbol{c}_{kp} && (7.11) \\
& \sum_{p=0}^{P_k-1} b_{kp} \geq 1 && (7.12) \\
& b_{kp} \in \{0, 1\} && (7.13)
\end{aligned}
$$

The reference trajectory $\boldsymbol{x}_{k,ref}$ is generated as described in section 7.3.2 - page 116. $\boldsymbol{R}_x$, $\boldsymbol{R}_N$ and $\boldsymbol{R}_u$ are the weight matrix for the discrete state errors without the final state, the weight matrix for the final discrete state error (terminal state), and the weight matrix for the input, respectively.

This optimization problem is solved at every planning iteration to generate an optimal trajectory with respect to its cost function. The MIQP is solved using the Gurobi solver [53].

Success	Distance (m)	Velocity (m/s)	Flight time (s)	Comp. time (ms)	Jerk cost ($10^3 m/s^3$)
5/5	51 / 53 / 1.8	2.7 / 4.3 / 1.3	18 / 20 / 0.9	37 / 93 / 13	1/ 1.6 / 0.5

Table 7.2: Results on 5 randomly generated maps of size $50\ m \times 12\ m \times 12\ m$ and with 250 dynamic obstacles/250 static obstacles. We show the **mean / max / standard deviation** of each metric.

7.4 . Limitations

Our method constrains each point to be in a polyhedron, unlike the planning method presented in chapter 6 (page 95) which constrains the whole segment to be in a polyhedron. Constraining the whole segment to be in a polyhedron allows to guarantee safety and force the trajectory to pass through the intersection of 2 consecutive polyhedra. This adds an additional constraint that may render the problem dynamically infeasible. In fact, we opted for constraining points instead of segments because during our testing, the solver was not finding solutions/converging in some cases when we constrained the whole segment to be in a polyhedra instead of just the point. This results in our method requiring the obstacles be further inflated to avoid collision when the segment between 2 points in 2 different polyhedra passes through an obstacle.

Furthermore, the method we presented is based on MPC and this implies that the length of the horizon will affect its performance: the method may fail to find a solution when the dynamics of the agent do not allow it to avoid the moving obstacles using only the chosen time horizon. Increasing the time horizon would increase trajectory quality and reduce the amount of times it fails to find a solution, but this comes at a heavy computational cost.

7.5 . Simulation

7.5.1 . Simulation environment

The simulation is done in a $50\ m \times 12\ m \times 12\ m$ environment that is represented in a voxel grid of voxel size 0.3. It contains 200 static obstacles and 200

dynamic obstacles that can fit inside a cube of side length $1.5\ m$ i.e. 5 voxels. Each voxel inside the obstacle cube is occupied with a probability of 0.1. Then the obstacles are inflated by a voxel to account for the drone radius. The dynamic obstacles oscillate at a random frequency between $\pi/4\ rad/s$ and $\pi/7\ rad/s$ along a line whose position, direction and length (between $0\ m$ and $5\ m$) are generated randomly (following a uniform distribution). The Gurobi solver is set to use one thread only as this resulted in faster computation times during our simulations. All testing is done on the Intel Core i7-9750H up to 4.50 GHz CPU.

7.5.2 . Planner parameters

We choose the following parameters: $N = 7$, $h = 100\ ms$, $g = 9.81\ m/s^2$, $a_{x,max} = a_{y,max} = 2 * g$, $a_{z,max} = g$, $a_{z,min} = -g$, $j_{x,max} = j_{y,max} = j_{z,max} = 90\ m/s^2$, $v_{max} = 4\ m/s$, $\boldsymbol{D}_{lin\_max} = diag(1, 1, 1)$, $thresh\_dist = 0.4\ m$. The weight matrices are diagonal: $\boldsymbol{R}_x = diag(5, 5, 5, 0, 0, 0, 0, 0, 0)$, $\boldsymbol{R}_N = diag(50, 50, 50, 0, 0, 0, 0, 0, 0)$ and $\boldsymbol{R}_u = diag(0.005, 0.005, 0.005)$. The drone radius is $d_{rad} = 0.2\ m$.

The DMP planner pushes the JPS path $0.4\ m$ away from obstacles (when possible). The path finding step is run only once at the beginning of the planning since we know the environment beforehand, which means the path finding step has no contribution to the total computation time. The planning time horizon of our planner ($N * h = 0.7\ s$) cannot be too long because of computation time, nor too short because the agent has to stop at the end of the trajectory and so if it is too short, the velocity will be low. We find a compromise experimentally.

7.5.3 . Simulation results

The agent goes from the start position of $(1, 6, 6)$ to $(49, 6, 6)$ while avoiding both static and dynamic obstacles. We show in Tab. 7.2 the results of 5 randomly generated environments. The agent succeeds in avoiding all obstacles and reaching the goal in all 5 simulations. The mean flight distance is $51\ m$, the mean flight velocity is $2.75\ m/s$ and the mean flight time is $18.5\ s$. The mean computation time is $37\ ms$ and the max is $93\ ms$. This means that even in the worst case, the computation time is smaller than the time step of the MPC (which is $100\ ms$). In case the computation time is higher at a given iteration, we discard the generated solution and plan the next iteration starting from the second position of the trajectory i.e. we assume the robot continued on its trajectory for a 2 steps without replanning.

We also run another simulation with only dynamic obstacles (for better visibility and performance assessment) with the same parameters as before but we change their positions to be at the same height as the agent and the oscillating direction

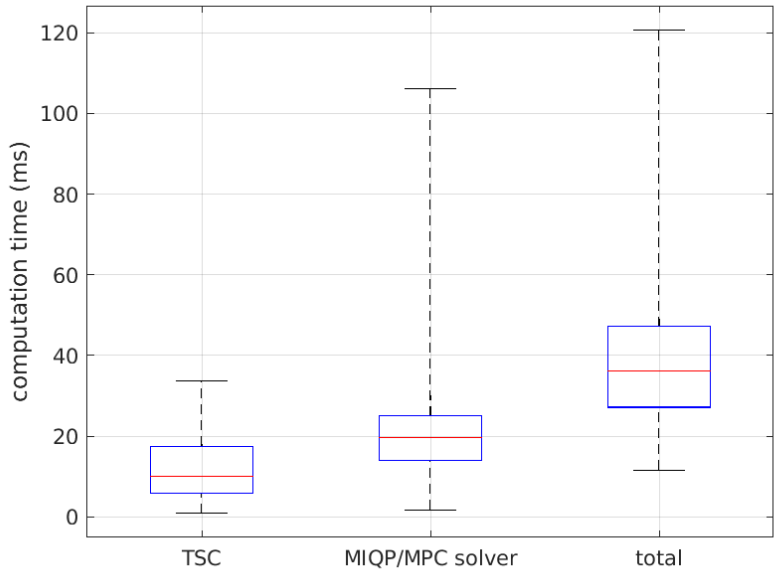


Figure 7.2: Box plots of the computation time for the Safe Corridor generation step, the solver optimization step and the total computation time which includes both steps. The red segment represents the median. The lower and upper bounds of the box represent the 25^{th} and 75^{th} percentile respectively, and the lower and upper whiskers represent the minimum and maximum respectively.

to be along the y axis. The agent still has the same starting position but the goal is now $(99, 6, 6)$ (the environment has doubled in size in the x direction). We also change $N = 8$ which increases the time horizon, and consequentially the computation time. The computation times for the Temporal Safe Corridor generation, the MIQP/MPC solver and the total computation time are shown in Fig. 7.2. In this case there were a few instances where the computation time exceeded the time step. The performance of the agent is shown in the video https://youtu.be/2ha8Huqi_qI.

7.6 . Conclusion

In this work, we explored a novel planning methods for multirotors in dynamic environments. The obstacles are represented as Temporal Occupancy Grids (TOG) that encode the positions of all obstacles (static and dynamic) at discrete instants in future time. The method consists in generating a Temporal Safe Corridor (a novel concept which we defined in this chapter), a local reference trajectory, and finally using the TSC and the reference trajectory in an MIQP/MPC formulation to

generate a safe trajectory that avoids all types of obstacles (static and dynamic). The method is tested in simulations and the limitations and challenges are presented.

8 - MULTI-AGENT PLANNING - STATIC ENVIRONMENT

Multi-agent planning has numerous real-world applications such as autonomous agriculture, search and rescue, and infrastructure inspection. It is thus of tremendous benefit to improve on the state-of-the-art in multiple metrics to make autonomous multi-agent systems more efficient and a more viable real-world solution. In this work, we propose a new framework for multi-agent planning in a static environment that improves upon the existing state-of-the-art in multiple areas such as computation time, trajectory smoothness, and feasibility.

The proposed method first generates a global path that only avoids static obstacles and then generates a Safe Corridor around it. It then extends the notion of Safe Corridors and makes them time-aware in order to account for the future positions of other agents. The time-aware Safe Corridor, along with a local reference trajectory generated from the global path, are used in an MIQP/MPC formulation that generates a collision-free optimal trajectory.

The proposed framework is real-time, decentralized, and synchronous. It is compared to another state-of-the-art method in simulations. It is compared to 2 recent state-of-the-art methods in simulations. It outperforms both methods in robustness as well as feasibility and computation time.

video: https://youtu.be/eyHYvF9iO0s

8.1 . INTRODUCTION

8.1.1 . Problem statement

There are multiple real world applications (exploration, search and rescue ...) for multi-agent planning that render research and advances in computation time and planning performance worthwhile.

A plethora of works have extensively addressed single agent planning in a static environment [12], [13], [20], [146], [47], [46], [88], [101]. However, the extension of these methods to multi-agent planning presents new challenges since the agents have to avoid each other as well as the static obstacles.

It is the purpose of this work to present a new method for multi-agent planning in a static environment that is also suitable for low compute embedded systems.

8.1.2 . Related work

Some works in the literature have addressed the problem of multi-agent planning with collision avoidance by modeling the agent and other obstacles as ellipsoids or spheres [2], [70], [175], [85]. They add the collision constraints in a Model Predictive Control (MPC) formulation to guarantee a minimum distance between the agents and the obstacles. However, partitioning static obstacles into ellipsoids is not trivial and can be inefficient for complex environments (see section 7.1 - page 113). Furthermore, the computation time increases with the number of added ellipsoids and may render the planning not suitable for low compute systems in complex and dense environments.

In [65], the authors present a centralized multi-agent planning framework that uses time-aware Safe Corridors. The method consists in 3 steps: roadmap generation, discrete planning and continuous refinement. The authors mention that the last step can be decentralized. The computation time, however, is not suitable for online high-speed planning, and the Safe Corridor generation method that they use requires a polyhedral representation of the environment which is not trivial to generate (see section 7.1 - page 113).

In [174], the authors use Buffered Voronoi Cells for multi-agent collision avoidance but do not address static obstacles. Others [96] use separating hyperplanes to avoid inter-agent collisions and model static obstacles as ellipsoid constraints in a decentralized MPC formulation. However, the authors of [96] do not address the complexity of decomposing static obstacles into ellipsoids using an automated algorithm, and use human assisted decomposition instead.

Search-based methods [90] have been proposed to avoid all type of obstacles (other agents, dynamic obstacles and static obstacles). However, they suffer from the curse of dimensionality which renders maneuvers highly computationally expensive and not suitable for real-time, low compute systems.

In [120], the authors find a global path and a Safe Corridor around it. Then the Safe Corridor is transformed into a Relative Safe Corridor to avoid inter-agent collisions. The trajectory is then sequentially optimized in a centralized way to find safe and collision free trajectories for all agents. This method is computationally efficient for a small number of agents, but can generate highly sub-optimal and long trajectories for some agents as shown in [120].

In [144] the authors present a new asynchronous multi-agent planning framework for avoiding other planning agents, dynamic obstacles (also called non-cooperative agents in some works) and static obstacles. They use a combination of a search-based method and an optimization method at every local planning it-

eration, where the result of the search-based method is used as an initialization for the optimization method. This choice is justified by the fact that the optimization method is non-convex and needs a good initial guess. The method keeps planning locally in asynchronous iterations until the agent reaches the goal.

Finally, in [121], the authors present an online distributed trajectory generation method for quadrotor swarm using time-aware Safe Corridors (called Linear Safe Corridors in [121]). The presented method uses an Octomap representation of the environment [64]. The Safe Corridor used by the planner is composed of only one polyhedron which leads to conservative/slow trajectories. The method plans local trajectories in an iterative fashion until the agent reaches the goal.

8.1.3 . Contribution

The main contribution of our paper is a novel decentralized and synchronous planning framework that is low compute and takes into account static obstacles and other planning agents. The proposed method takes inspiration mainly from [148] with many steps in common. Our global path finding, local reference generation, Safe Corridor generation, and the MIQP formulation is inspired by [148]. The temporal Safe Corridor generation is inspired by many works such as [65] [120]. The method is compared to 2 recent works [144] [121] in simulations in terms of computation time, and trajectory safety and performance.

8.2 . Assumptions

The following assumptions are made for each agent:

- Perfect control: the agent is able to follow the generated trajectory perfectly.
- Perfect localization: we assume the agent knows perfectly its position with respect to a fixed world frame.
- Peer to peer communication between all agents/no routing involved. The communication is used to transmit the planned trajectory of each agent to all the other nearby agents. We assume no communication loss between agents, but take into account communication delay.
- The communication delay must not exceed the difference between the planning period (time step of the MPC defined in section 6.3.4) and the computation time. We detail this assumption in section 8.4.6 and present a trivial mechanism to relax this hard constraint.

Previous works have done these hypotheses: [144] have the same hypotheses as presented but with the addition of the hypothesis that no communication delay

exists between agents.

The hypotheses of perfect control and perfect localization can be discarded by adding a security margin (inflating each agent's collision radius) to account for uncertainties in control and localization. More optimal solutions can be implemented to take into account control and localization uncertainties, but we leave them for future works as they are beyond the scope of this work.

8.3 . Agent Model and Definitions

Table 8.1: Nomenclature

$\boldsymbol{p}$	position vector x, y, z in the world frame
$\boldsymbol{v}$	velocity vector v_x, v_y, v_z in the world frame
$\boldsymbol{a}$	acceleration vector from thrust and gravity in the world frame
$\boldsymbol{j}$	jerk vector j_x, j_y, j_z in the world frame
$\boldsymbol{D}_{lin\_max}$	linear drag matrix
M	number of planning agents
N	number of discretization steps in the MPC
h	time step of the MPC
d_{rad}	drone radius

Each agent is modeled as the following linear model with the jerk j as input to the system as in [148] (Table 8.1):

$$\begin{aligned} \dot{\boldsymbol{p}} &= \boldsymbol{v} \\ \dot{\boldsymbol{v}} &= \boldsymbol{a} - \boldsymbol{D}_{lin\_max} \cdot \boldsymbol{v} \\ \dot{\boldsymbol{a}} &= \boldsymbol{j} \end{aligned} \tag{8.1}$$

$\boldsymbol{D}_{lin\_max}$ is a diagonal matrix that represents the maximum linear drag coefficient in all directions. This matrix is identified offline as shown in section 6 (page 95). The quadratic drag model is replaced with a linear worst-case scenario model (to ensure feasibility and safety). Many off-the-shelf solvers are very efficient for linear constraints, which is why we use a linear model. We suppose we have M planning agent each starting at a given position and they want to reach a given goal. The environment contains static obstacles, and the objective for each agent is to avoid them as well as other planning agents.

8.4 . The planner

Our decentralized planner is run concurrently on multiple agents whose clocks are synchronized. It is divided in 2 stages (Fig. 8.1). The first stage takes the position of the agent, a voxel grid partitioning space into free, and occupied voxels, and a goal in 3D space as input. It is run only once at the beginning of the planning and consists of two steps:

1. Generating a global path (section 8.4.1 - page 129).

2. Generating a Safe Corridor (section 8.4.2 - page 130)

In the first step, we generate a global path to the goal. The path is obtained by deforming an optimal/shortest distance path to push it away from obstacles. This improves the performance of the planned trajectory. The path serves to avoid static obstacles and does not consider other agents. In the second step we generate a series of overlapping convex polyhedra (Safe Corridor) around the global path while also not considering other agents.

The second stage takes the global path and Safe Corridor generated in the first stage, and the generated optimal trajectories of all the other agents at the previous iteration as input. It is run in a loop at a constant rate and consists of 3 steps:

1. Generating a time-aware Safe Corridor (section 8.4.3 - page 130).

2. Generating a local reference trajectory (section 8.4.4 - page 133).

3. Solving the Mixed-Integer Quadratic Program (MIQP)/Model Predictive Control (MPC) problem to generate a locally optimal trajectory (section 8.4.5 - page 136).

In the first step, we generate a time-aware Safe Corridor using the Safe Corridor and the generated trajectories of all the other agents at the previous iteration. This ensures no collisions happen with other agents (due to the hyperplanes that separate them as shown in section 8.4.3 - page 130). In the second step, a local reference trajectory is generated. In the third step, we formulate a convex MIQP/MPC that takes the local reference trajectory and the time-aware Safe Corridor, and generates an optimal trajectory that doesn't collide with obstacles or other agents. This trajectory can then be sent to the agent's controller to execute.

8.4.1 . Generating a global path

In this step, we take a voxel grid partitioning the space into free and occupied voxels and generate the shortest path from the starting voxel position of the robot to the goal voxel (which we call the global path). The global path is generated as presented in section 6.3.1 (page 101).

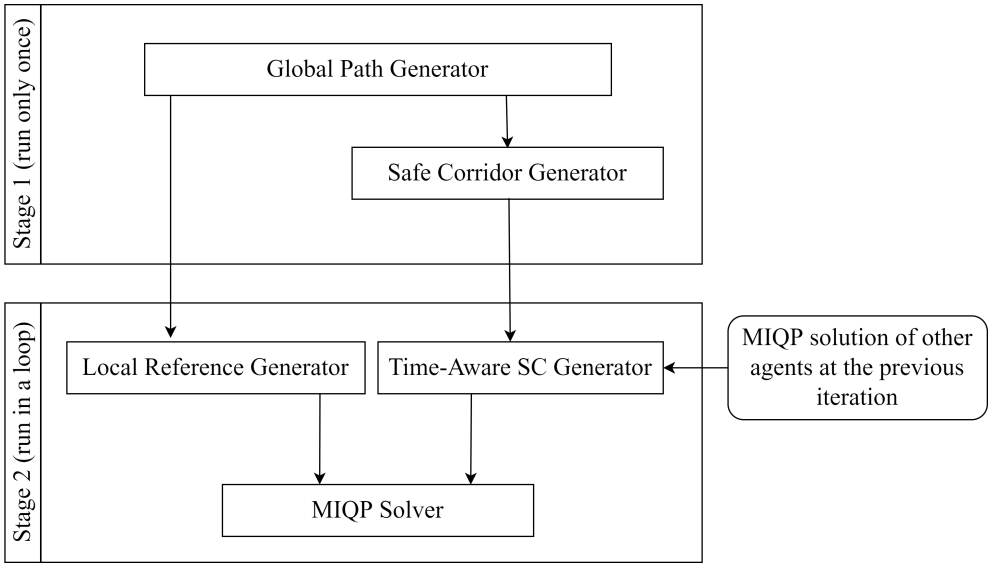


Figure 8.1: We show the global pipeline of the planning framework of a single planning agent. It is decomposed into two stages. Stage 1 is run only once at the beginning of the planning whereas stage 2 is run in a loop.

8.4.2 . Creating a Safe Corridor around the global path

After the global path is found, the free space around the path is decomposed into overlapping convex polyhedra the same manner as presented in section 6.3.2 (page 102).

The computation time in this step depends on the maximum size of the polyhedron that we want to generate: the more free space we want each polyhedron to cover, the more computation time is required.

8.4.3 . Generating a time-aware Safe Corridor

In this step, we use the generated Safe Corridor (SC) and the trajectories generated at the previous iteration of all the planning agents in order to generate a time-aware SC that accounts for other moving agents. We will use an simple illustrative example as we explain the algorithm (Fig. 8.2). SC decompose free space into overlapping convex polyhedra. Our algorithm that generates the time-aware SC will modify the polyhedron in which the agent exists to take into account other planning agents. Similarly, we modify a number of subsequent polyhedra that contain the further discrete positions of the trajectory.

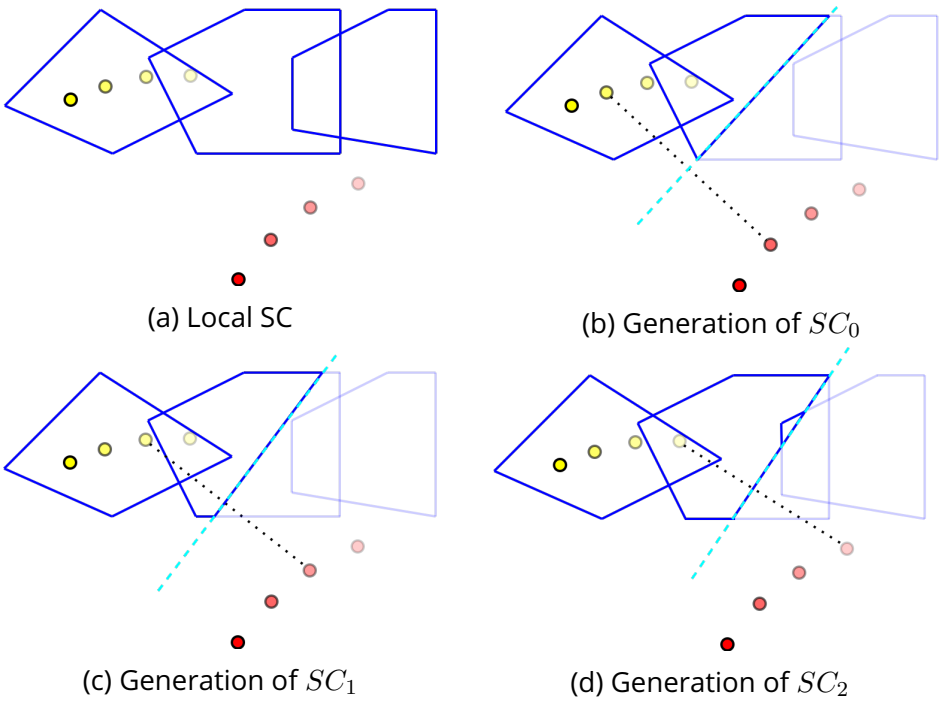
We first find the earliest polyhedra in the SC that contains the current planning agent position. We then use it with the subsequent $P_{hor} - 1$ polyhedra (with P_{hor} the polyhedra horizon, i.e., the number of polyhedra we consider for the op-

timization) as a local SC (Fig. 8.2a - where $P_{hor} = 3$). The local SC is then modified into N truncated local SCs, each denoted by SC_i (with N the number of discretization steps of the MPC). This series of truncated local SCs is what we call time-aware SC, and its generation is described in details in what follows.

Then, we take the generated optimal trajectories of the planning agent as well as all the other agents at the previous iteration. These trajectories are a series of $N + 1$ discrete positions (Fig. 8.2a - $N = 3$). The second discrete position is where the agents will be at the beginning of this iteration (since the trajectories are generated at the previous iteration). For each discrete position $\boldsymbol{p}_{plan,i}$ of the planning agent, a hyperplane is generated with each discrete position $\boldsymbol{p}_{j,i}$ of the other agents, with j a number between 1 and the number of other agents. This means the number of generated hyperplanes for each $\boldsymbol{p}_{plan,i}$ is equal to the number of other agents (Fig. 8.2 - $M = 1$, the hyperplanes are shown in cyan). The index i is between 1 and N and represents the index of the corresponding discrete position i.e. the position of the agent at time $i * h$ with h the discretization step (i is also equal to $k + 1$ with k defined in section 8.4.5 (page 136) and equation (8.9)). Each hyperplane has a normal vector $\boldsymbol{n}_{hyp}$ going from the position of the planning agent to the position of the other agent that we want to avoid using the hyperplane: $\boldsymbol{n}_{hyp} = \boldsymbol{p}_{j,i} - \boldsymbol{p}_{plan,i}$. It passes through the point $\boldsymbol{P}_{hyp}$, which corresponds to the midpoint between the position of the planning agent and the position of the other agent, offset by the drone radius d_{rad} in the direction of the planning agent: $\boldsymbol{P}_{hyp} = \frac{\boldsymbol{p}_{j,i}+\boldsymbol{p}_{plan,i}}{2} - d_{rad}\frac{\boldsymbol{n}_{hyp}}{||\boldsymbol{n}_{hyp}||_2}$. The corresponding constraint is $\boldsymbol{n}_{hyp}.(\boldsymbol{p}_{agent} - \boldsymbol{P}_{hyp}) \leq 0$ where $\boldsymbol{p}_{agent}$ is the discrete position that should satisfy the constraint. This constraint makes sure that the minimum distance between the planning agent and the other agent is $2d_{rad}$ since the hyperplane of the other agent is also offset by d_{rad} in its direction. All the hyperplanes generated using $\boldsymbol{p}_{plan,i}$ and all the other agents are added to the SC to generate SC_{i-1} (Fig. 8.2 - we show the generation of 3 SC_i). Each SC_i will serve as a local SC for the $i - 1$ and i discrete positions of the planning agent in the current iteration (i.e. both discrete positions will be constrained to to be inside SC_{i-1}). In Fig. 8.2, this means that, in the current planning iteration, $\boldsymbol{p}_0$ and $\boldsymbol{p}_1$ will be constrained to SC_0, $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ will be constrained to SC_1, $\boldsymbol{p}_2$ and $\boldsymbol{p}_3$ will be constrained to SC_2, $\boldsymbol{p}_3$ and $\boldsymbol{p}_4$ will be constrained to SC_2 (the last SC_i is used twice).

Dealing with stalemates

In some cases, 2 agents can be in a stalemate when they are moving one towards the other along the direction of $\boldsymbol{n}_{hyp}$. They end up both being stuck (stationary) at the borders of the hyperplanes without moving due to the fact that the optimizer of every agent can't find a moving/better trajectory to get closer to the goal (Fig.



(a) Local SC

(b) Generation of SC_0

(c) Generation of SC_1

(d) Generation of SC_2

Figure 8.2: We show how a time-aware SC is generated from a local SC. The position of the planning agent is shown as the yellow circles, and the other agent that we want to avoid is shown as the red circles. The positions of the agents become more transparent as we move in time.

8.4a). There is a wide literature on dealing with stalemates such as the right-hand rule [96] or priority planning [87]. To deal with this issue, we use a similar approach to the right-hand rule. We modify the normal of the hyperplane of every local SC of each agent to make it tilted (Fig. 8.4b). We define the normalized plane normal $\boldsymbol{n}_{hyp,norm}$, the *right* vector $\boldsymbol{r}$ that is the cross product between $\boldsymbol{n}_{hyp,norm}$ and $\boldsymbol{z}_W$ plus the cross product between $\boldsymbol{n}_{hyp,norm}$ and $\boldsymbol{y}_W$, a perturbation m, and a user chosen coefficient c that defines how tilted the final normal vector of the hyperplane $\boldsymbol{n}_{hyp,final}$ is with respect to the initial vector $\boldsymbol{n}_{hyp}$:

$$\boldsymbol{n}_{hyp,norm} = \frac{\boldsymbol{n}_{hyp}}{||\boldsymbol{n}_{hyp}||_2} \tag{8.2}$$

$$\boldsymbol{z}_W = [0,0,1]^T, \quad \boldsymbol{y}_W = [0,1,0]^T \tag{8.3}$$

$$\boldsymbol{r} = \boldsymbol{n}_{hyp,norm} \times \boldsymbol{z}_W + \boldsymbol{n}_{hyp,norm} \times \boldsymbol{y}_W \tag{8.4}$$

$$\boldsymbol{n}_{pert} = (c+m).\frac{\boldsymbol{r}}{||\boldsymbol{r}||_2} + c.\boldsymbol{z}_W \tag{8.5}$$

$$\boldsymbol{n}_{hyp,final} = \boldsymbol{n}_{pert} + \boldsymbol{n}_{hyp,norm} \tag{8.6}$$

Typically, we choose $c = 0.1$. The perturbation m changes values between 0 and 0.05 at every planning iteration in a continuous fashion (the change between two consecutive iterations is relatively small). The idea is to continuously perturb $\boldsymbol{n}_{hyp,norm}$ with a vector $\boldsymbol{n}_{pert}$ that slightly and continuously changes direction every iteration to avoid stalemates.

8.4.4 . Generating a local reference trajectory

In this step, the global path is used to generate a local reference trajectory for the MIQP/MPC. At each iteration we sample N points from the global trajectory to be used as reference for the N discrete positions of the MIQP/MPC. At every iteration l, these points are sampled using a starting point $\boldsymbol{x}^l_{0,ref}$. At the first iteration this starting point is the position of the planning agent. From the starting position, we move along the global path at the sampling speed v_{samp} (user input) for a time duration h, where h is the time step of the MIQP/MPC. The point at which we arrive is the second reference point $\boldsymbol{x}^l_{1,ref}$. We continue sampling in the same fashion until we reach $\boldsymbol{x}^l_{N,ref}$. At the subsequent iterations, we use the optimal trajectory generated by the last step (MIQP/MPC) of the previous iteration: we check if the final state is close enough from the previous final reference point or the first state is close enough from the first reference point (within $thresh\_dist$). If yes, the local reference trajectory is generated with the above-mentioned algorithm starting from $\boldsymbol{x}^l_{0,ref} = \boldsymbol{x}^{l-1}_{1,ref}$. If no, the local reference trajectory generated at the previous iteration is used for the current iteration i.e. $\boldsymbol{x}^l_{0,ref} = \boldsymbol{x}^{l-1}_{0,ref}$.

8.4.5 . Solving the MIQP/MPC problem

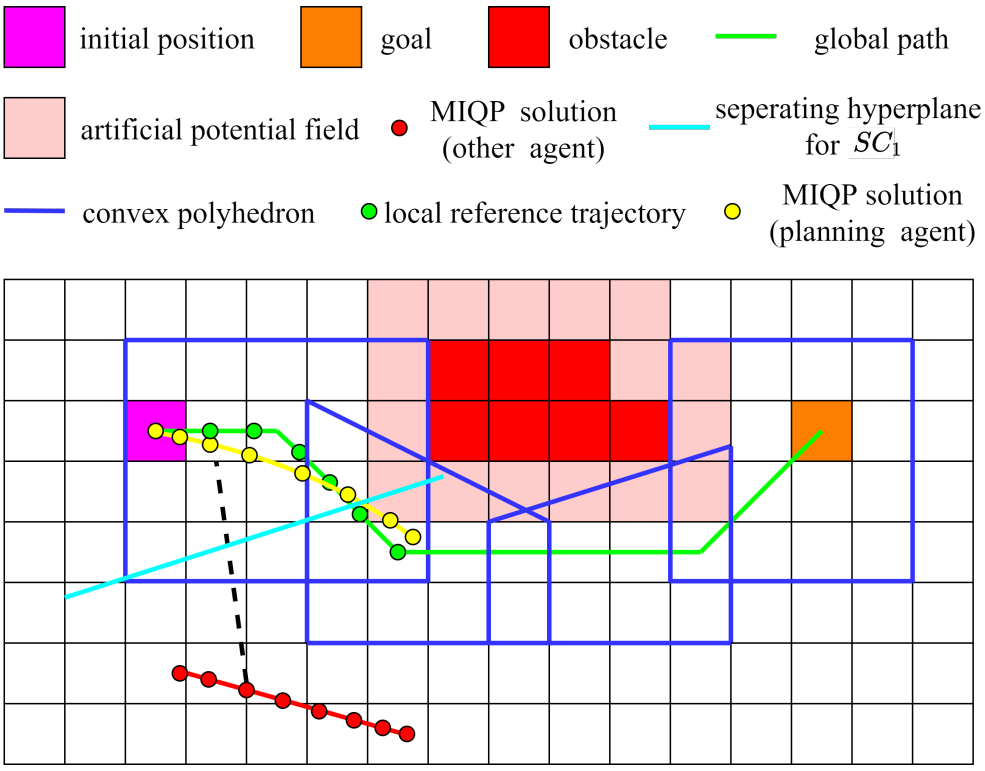
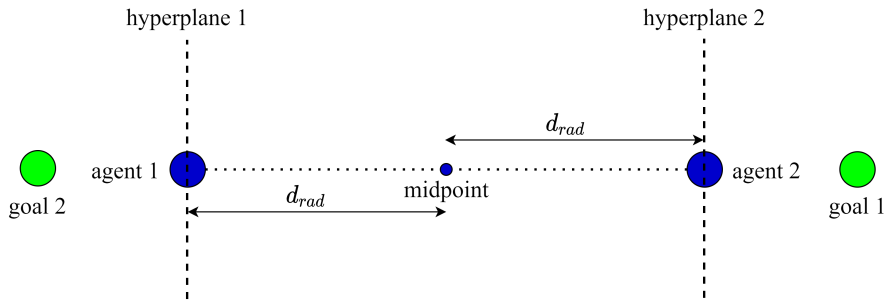
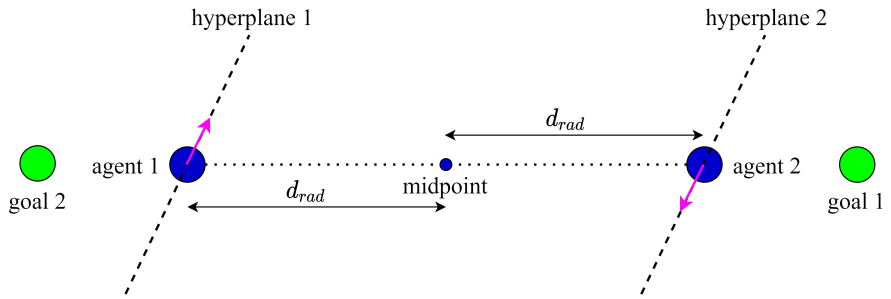


Figure 8.3: We show a planning iteration of our algorithm in a simplified 2D case. The global path is generated using JPS and DMP. DMP uses the artificial potential field (pink) to push the path away from the obstacles (in this case by a distance of 1 voxel). The convex polyhedra (blue) span only the free space. A local reference trajectory is sampled from the global path (green circles). We show the separating hyperplane (cyan) generated between the planning agent and another moving agent using the third predicted position. This hyperplane constraint is added to the convex polyhedron constraints to create SC_1 which will be used as constraint for the second and third positions in the next planning iteration. This procedure is done for all positions to create all SC_i (time-aware SC). It is then given at the next iteration with the local reference trajectory to the MIQP solver to generate a feasible trajectory (yellow circles).



(a) Stalemate case



(b) Breaking the stalemate

Figure 8.4: The stalemate happens when each agent wants to go to its goal on the other side of its own hyperplane, and there is no movement in any direction that makes it close to the goal. By changing the hyperplanes' normal, the agents can now move closer to their goals by moving along the magenta arrows shown in Fig. 8.4b. This would break the stalemate.

In this step, a collision-free optimal trajectory is generated. It uses the local reference trajectory generated in section 8.4.4 (page 133) and the time-aware Safe Corridor generated in section 8.4.2 (page 130). The error between the agent position and the local reference trajectory as well as the norm of the jerk are minimized. The resulting optimal trajectory is a smoothed version of the local reference trajectory (Fig. 8.3).

At every iteration l, the initial state $\boldsymbol{x}_0^l$ of the MPC is set to the second state $\boldsymbol{x}_1^{l-1}$ of the last generated trajectory (except for the first iteration where the initial state is the initial robot position). The terminal velocity $\boldsymbol{v}_N$ as well as the terminal acceleration $\boldsymbol{a}_N$ are set to $\mathbf{0}$ to make sure that the agent has a safe trajectory to execute in case subsequent MIQP optimizations fail to find a solution.

In case the solver fails to find a solution at a given iteration or the computation time exceeds the time step h, we skip the iteration (the solution is discarded), and at the next iteration, we solve the MIQP/MPC with the initial state $\boldsymbol{x}_2$ instead of $\boldsymbol{x}_1$. In case this also fails, we keep offsetting the initial position (which may reach $\boldsymbol{x}_N$ in the worst case). Every time the solver fails, no trajectory is sent to the other agents, and the other agents assume that the planning agent will follow the last successfully generated trajectory (which is what actually happens).

Dynamics

With $\boldsymbol{x} = [\boldsymbol{p}\ \boldsymbol{v}\ \boldsymbol{a}]^T$, $\boldsymbol{u} = \boldsymbol{j}$, $f(\boldsymbol{x}(t), \boldsymbol{u}(t))$ defined by Eq. 8.1, the model is discretized using Euler or Runge-Kutta 4$^{\text{th}}$ order to obtain the discrete dynamics $\boldsymbol{x}_{k+1} = f_d(\boldsymbol{x}_k, \boldsymbol{u}_k)$. We choose the Euler method as it results in faster solving times. With a discretization step of h, the discretized dynamics become:

$$\begin{aligned}
\boldsymbol{p}_{k+1} &= \boldsymbol{p}_k + h\boldsymbol{v_k} \\
\boldsymbol{v}_{k+1} &= \boldsymbol{v}_k + h(\boldsymbol{a}_k - \boldsymbol{D}_{lin_max}\boldsymbol{v_k}) \\
\boldsymbol{a}_{k+1} &= a_k + h\boldsymbol{j_k} \\
\boldsymbol{x}_k &= [\boldsymbol{p}_k\ \boldsymbol{v}_k\ \boldsymbol{a}_k]^T \\
\boldsymbol{u}_k &= \boldsymbol{j}_k
\end{aligned} \tag{8.7}$$

These dynamics are added as equality constraints in the formulation of the MIQP/MPC to make the generated trajectory adhere to the agent's dynamics.

State bounds

The agent velocity is limited by the drag forces. The maximum bounds on the acceleration and the jerk in each direction are determined by the dynamics of the

agent. We define $a_{x,max}$ and $a_{y,max}$ as the maximum L1 norm of the acceleration in the directions x and y. We define $a_{z,max}$ and $a_{z,min}$ as the maximum and minimum values respectively of the acceleration in the z direction. These values are deduced directly from the maximum thrust that a multirotor can generate.

Finally, we define $j_{x,max}$, $j_{y,max}$ and $j_{z,max}$ as the maximum L1 norm of the jerk in the directions x, y and z respectively. These values represent the limits on the rotational dynamics of a multirotor.

Collision avoidance

This is achieved by forcing every two consecutive discrete points k and $k+1$ (and thus the segment formed by them) to be in one of the polyhedra of a given local SC (SC_k) that belongs to the agent's time-aware SC. Since each SC_k is formed by adding constraints to the SC that take into account the position of all other agents at the discrete position k in the future, this allows the planning agent to plan in the predicted free space at time $h * k$ in the future. Let's assume we have P overlapping polyhedra in SC_k. They are described by $\{(\boldsymbol{A}_{kp}, \boldsymbol{c}_{kp})\}$, $p = 0 : P-1$. The constraint that the discrete position $\boldsymbol{p}_k$ is in a polyhedron p is described by $\boldsymbol{A}_{kp}.\boldsymbol{p}_k \leq \boldsymbol{c}_{kp}$. We introduce binary variables b_{kp} (P variables for each $\boldsymbol{x}_k$, $k = 0 : N-1$) that indicate that $\boldsymbol{p}_k$ and $\boldsymbol{p}_{k+1}$ are in the polyhedron p. We force all the segments to be in at least one of the polyhedra with the constraint $\sum_{p=0}^{P-1} b_{kp} \geq 1$. This means that there must exist an overlapping region between all polyhedra and that at least one discrete point must belong to this overlapping region when crossing from one polyhedron to another.

Typically, the number of polyhedra considered for optimization P_{hor} is 3 to avoid high solving times. We start with the polyhedra that contains the current position of the agent and take 2 other subsequent polyhedra for the optimization.

Formulation

We formulate our MPC under the following Mixed-Integer Quadratic Program (MIQP) formulation. We remove the superscript l (which indicates the number of the iteration) from the reference and state variables for simplification.

$$
\begin{aligned}
\underset{\boldsymbol{x}_k, \boldsymbol{u}_k}{\text{minimize}} \quad & \sum_{k=0}^{N} (||\boldsymbol{x}_k - \boldsymbol{x}_{k,ref}||^2_{\boldsymbol{R}_x} + ||\boldsymbol{u}_k||^2_{\boldsymbol{R}_u}) \\
& + ||\boldsymbol{x}_N - \boldsymbol{x}_{N,ref}||^2_{\boldsymbol{R}_N} && (8.8) \\
\text{subject to} \quad & \boldsymbol{x}_{k+1} = f_d(\boldsymbol{x}_k, \boldsymbol{u}_k), \quad k = 0 : N-1 && (8.9) \\
& \boldsymbol{x}_0 = \boldsymbol{X}_0 && (8.10) \\
& \boldsymbol{v}_N = \boldsymbol{0} && (8.11) \\
& \boldsymbol{a}_N = \boldsymbol{0} && (8.12) \\
& |a_{x,k}| \leq a_{x,max} && (8.13) \\
& |a_{y,k}| \leq a_{y,max}, \quad a_{z,k} \leq a_{z,max} && (8.14) \\
& a_{z,k} \geq a_{z,min}, \quad |j_{x,k}| \leq j_{x,max} && (8.15) \\
& |j_{y,k}| \leq j_{y,min}, \quad |j_{z,k}| \leq j_{z,max} && (8.16) \\
& b_{kp} = 1 \implies \begin{cases} \boldsymbol{A}_{kp} \boldsymbol{p}_k \leq \boldsymbol{c}_{kp} \\ \boldsymbol{A}_{kp} \boldsymbol{p}_{k+1} \leq \boldsymbol{c}_{kp} \end{cases} && (8.17) \\
& \sum_{p=0}^{P_{hor}-1} b_{kp} \geq 1 && (8.18) \\
& b_{kp} \in \{0, 1\} && (8.19)
\end{aligned}
$$

The reference trajectory $\boldsymbol{x}_{k,ref}$ is generated as described in section 8.4.4 (page 133). $\boldsymbol{R}_x$, $\boldsymbol{R}_N$ and $\boldsymbol{R}_u$ are the weight matrix for the discrete state errors without the final state, the weight matrix for the final discrete state error (terminal state), and the weight matrix for the input, respectively.

This optimization problem is solved at every planning iteration to generate an optimal trajectory with respect to its cost function. The MIQP is solved using the Gurobi solver [53].

8.4.6 . Communication between agents

At every planning iteration, each agent needs all the generated trajectories of the other agents at the previous iteration in order to be able to plan safely. This means that at every iteration, every agent needs to broadcast his current planned trajectory to all the other agents as soon as the last step in the planning framework is done, and the trajectory needs to reach all the other agents before the next planning iteration begins i.e. after h time from the current iteration. If the computation time of the generated trajectory at the current iteration is t_{comp}, this leaves $h - t_{comp}$ time for the trajectory to reach all the other agents before the next planning iteration begins for all the agents. It is thus crucial to have a low computation time to allow for communication latency. In order to account for the delay in a more active manner, one can simply estimate a fixed communication

delay t_d, and assume that we failed to produce a trajectory (which we treat in section 6.3.4) (page 104) when $t_d + t_{comp} > h$.

Note that practically, not all agents need to transmit their trajectories to all other planning agents, but only those who are in proximity (within a certain distance that takes into account each agent's dynamics). This would allow to remove constraints from the optimization formulation and reduce computation time. This can be achieved by preventing the routing of packets between agents and by having a signal that is weak enough to not pollute the communication between far away agents. In this case, it is possible to estimate t_d by taking into account the communication protocol and the maximum density of the agents.

8.5 . Simulation Results

All testing is done on the Intel Core i7-9750H up to 4.50 GHz CPU. The code is written in C++ and we use MATLAB for illustration, and MATLAB and ROS/Gazebo for animation in the video. The agents are in a symmetrical/circular configuration and swap positions (Fig. 8.5). Using asymmetrical configurations results in slightly better performance metrics for all methods, which is why we only present the hardest configuration (symmetrical). The **jerk cost** (which is a measurement of trajectory smoothness) is defined as $J_{\text{cost}} = \int_{t_{\text{ini}}}^{t_{\text{fin}}} ||\boldsymbol{j}(t)||^2 \mathrm{d}t$ where t_{ini} and t_{fin} are the initial and final time of the trajectory. We consider a simulation not successful when one or more drones fail to reach their goal position or the safety distance ($2 \cdot d_{rad}$) between the drones was violated at any point in time. For the **computation time** metric, we do not include stage 1 (global path finding and SC generation) of our planning method which is run only once at the beginning of the planning. Its value depends on the size of the environment as well as the obstacles inside it. We indicate the computation time of stage 1 in the simulations when comparing with different environments.

8.5.1 . Planner parameters

We choose the following parameters: $N = 7$, $h = 100\ ms$, $g = 9.81\ m/s^2$, $a_{x,max} = a_{y,max} = 2 * g$, $a_{z,max} = g$, $a_{z,min} = -g$, $j_{x,max} = j_{y,max} = j_{z,max} = 40\ m/s^2$, $v_{max,samp} = 3.5\ m/s$, $\boldsymbol{D}_{lin\_max} = diag(1,1,1)$, $thresh\_dist = 0.4\ m$, $P_{hor} = 3$. The weight matrices are diagonal:
$\boldsymbol{R}_x = diag(200, 200, 200, 0, 0, 0, 0, 0, 0)$, $\boldsymbol{R}_N = diag(100, 100, 100, 0, 0, 0, 0, 0, 0)$ and $\boldsymbol{R}_u = diag(0.01, 0.01, 0.01)$. All dynamical limits are set to the same values for all planners that we compare with [144] [121] (when possible).

The voxel grid used for path finding and Safe Corridor generation has a voxel size of $0.2\ m$. The DMP planner pushes the JPS path $0.4\ m$ away from obstacles (when possible). The planning frequency for stage 2 is 10Hz (since the time step

is $h = 100\ ms$).

The planning time horizon of our planner ($N \cdot h = 0.7\ s$) cannot be too long because of computation time, nor too short because the agent has to stop at the end of the trajectory and so if it is too short, the velocity will be low. We find a compromise experimentally.

8.5.2 . Comparison with the state-of-the-art

Table 8.2: Comparison between MADER [144] and our planner on 10 randomly generated maps of size $20\ m \times 20\ m \times 5\ m$ and with obstacle density $0.25\ obst/m^2$ for 4 agents. We show the **mean / max / standard deviation** of each metric. We also compute the difference in performance for the **mean** and **max** values. The better performer is shown in bold.

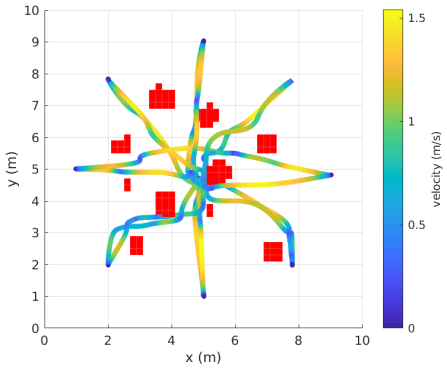
	Success	Flight distance (m)	Flight velocity (m/s)	Flight time (s)
MADER [144]	8/10	21.27 / 24.2 / 0.98	**2.77** / **4.3** / 1.1	**7.56** / 12.3 / 1.31
Our planner	**10/10**	**20.9** / **21.5** / 0.23	2.22 / 3.52 / 1.23	9.29 / **10** / 0.29
Difference (%)	-	-1.7 / -11.1	-20 / -18.1	+22.9 / -18.2

	Computation time (ms)	Jerk cost ($10^3 m/s^3$)
MADER [144]	72 / 233 / 54.6	6.91 / 13.9 / 3.37
Our planner	**4.6** / **26.6** / 1.9	**2.02** / **2.78** / 0.41
Difference (%)	-93.6 / -88.5	-70.77 / -80

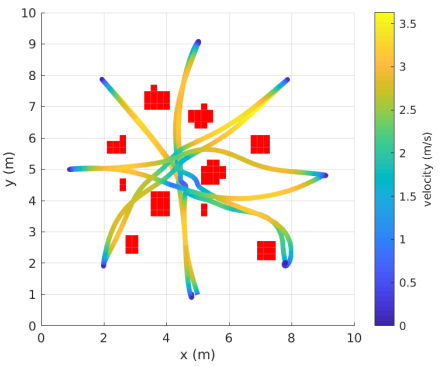
Table 8.3: Comparison between Park et al. [120] and our planner on 10 randomly generated maps of size $10\ m \times 10\ m \times 3\ m$ and with 10 randomly generated obstacles for 4, 8 and 16 agents. We show the **mean / max / standard deviation** of each metric. The better performer is shown in bold.

#		Success	Flight distance (m)	Flight velocity (m/s)
4	Park [121]	8/10	8.4 / 9 / 0.24	0.71 / 1.5 / 0.47
	Our planner	**10/10**	8.4 / **8.8** / 0.13	**1.58** / **3.3** / 1.3
8	Park [121]	8/10	8.9 / 9.94 / 0.54	0.71 / 1.5 / 0.41
	Our planner	**10/10**	**8.74** / **9.2** / 0.21	**1.56** / **3.65** / 1.21
16	Park [121]	7/10	9 / 10 / 0.43	0.68 / 1.5 / 0.41
	Our planner	**10/10**	**8.7** / **9.7** / 0.23	**1.45** / **3.63** / 1.15

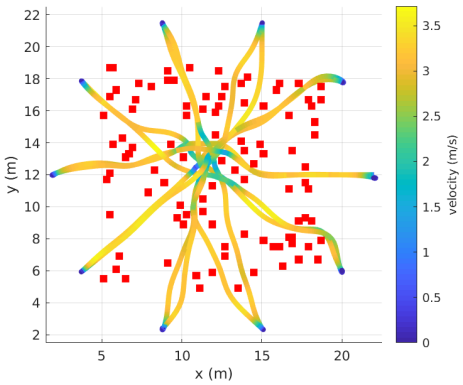
#		Flight time (s)	Computation time (ms)	Jerk cost ($10^3 m/s^3$)
4	Park [121]	11.7 / 17.2 / 1.74	7.2 / 30.5 / 2.3	**0.14** / **0.2** / 0.03
	Our planner	**5.32** / **5.7** / 0.26	**4.1** / **24.4** / 1.8	1.4 / 2.32 / 0.44
8	Park [121]	12.4 / 16.4 / 1.37	8.2 / 40.5 / 2.3	**0.16** / **0.25** / 0.04
	Our planner	**5.48** / **6.3** / 0.28	**5.7** / **38.2** / 3.5	1.8 / 2.5 / 0.4
16	Park [121]	13 / 16.2 / 1.4	9 / **33.5** / 2.2	**0.16** / **0.33** / 0.05
	Our planner	**5.9** / **7** / 0.4	**8.9** / 98 / 7	2.13 / 4.2 / 0.5



(a) Park et al. [121] - 8 agents in a *small environment*



(b) Our planner - 8 agents in a *small environment*



(c) Our planner - 10 agents in a *big environment*

Figure 8.5: The trajectories and velocity of the agents (swapping positions) generated by Park et al. [121] and our planner on the same map (an overhead view). We show the obstacles inflated by the drone radius using a voxel grid. Note that due to the perspective of the overhead view, the trajectories may appear as colliding with the obstacles, which is not the case (both methods generate collision free trajectories).

MADER

We compare our planner with the state-of-the-art MADER planner [144]. The simulation is done in a $25\ m \times 25\ m \times 5\ m$ environment that contains 100 obstacles of dimensions $0.2\ m \times 0.2\ m \times 3\ m$ (*big environment*). Their positions are generated randomly, following a uniform distribution (as shown if Fig. 8.5c). The drone radius is $d_{rad} = 0.2\ m$. This planner requires that all obstacles be modeled as convex polyhedra. The transformation of a pointcloud representation of the environment (which is the output of lidar, IR sensors, stereo matching algorithms...) into a convex polyhedral one is not trivial and can add computational overhead to the planning framework. This overhead is not considered in our comparisons and we assume a polyhedral representation is already available.

The comparison is done using 10 randomly generated maps of the environment. We choose a small number of agents (4) for the comparison in order to better estimate a real-world experiment, since all computations are done on the hexa-core CPU (to better simulate each agent having its own cpu/core since MADER requires heavy computation). The results are shown in Tab. 8.2.

In terms of **flight time** and **flight velocity**, MADER outperforms our method on average by around 20%. This is due to the fact that MADER has no bounds on the jerk and the acceleration of the trajectory can change instantly. This can lead to unfeasible trajectories as it can break the rotational dynamics/limits of a multirotor. During our experiments, the maximum jerk norm of a MADER trajectory was $188\ m/s^3$, whereas our trajectories adhered to the dynamical limits of $40\ m/s^3$ at each axis, which results in a total norm of $70\ m/s^3$. This is also reflected in the **jerk cost** metric, where on average our trajectory is $3.5\times$ *smoother*. This is also due to the fact that we use a short time horizon to minimize computation time. In fact, if we modify the variables $N = 10$, $v_{samp,max} = 4.5\ m/s$, we will outperform MADER in terms of mean flight velocity and mean flight time by around 10% (while still maintaining a 100% success rate). However, this would result in an increase in mean computation time to $5.7\ ms$ ($12.8\times$ better than MADER) and max computation time to $46\ ms$ ($5\times$ better than MADER).

On the other hand, our planner is more **robust** (higher success rate), has a lower **flight distance**, and is on average $15\times$ more computationally efficient (stage 1 takes on average $4\ ms$). The reason for the **computation time** disparity is the fact that MADER uses a search-based step to generate a trajectory in the time space that avoids all types of obstacles (dynamic and static) up to a given time horizon. This step suffers from the curse of dimensionality. This is in contrast to our method which finds a path that avoids all static obstacles, and then deals with the other moving agents in the optimization phase, which results in much faster computation time for the search-based phase. During the second step of MADER

(optimization), the obstacles are represented as convex polyhedra, which can lead to a relatively large number of inequality constraints in complex environments, and consequently to high solving times.

We also ran simulations for 10 and 16 agents in the *big environment* using our method (Fig. 8.5c), which resulted in the same performance metrics as Tab. 8.2, but with an increase in the average **computation time** ($6.6\ ms$ for 10 agents and $9\ ms$ for 16 agents). We could not run simulations for MADER with 10 and 16 agents on our PC and provide a fair comparison due to the heavy computation it requires. However, one can observe from the results presented in [144] and in Tab. 8.2, 8.3 that the difference in performance remains similar even when increasing the number of agents (the authors of MADER [144] use Google Cloud with multiple instances to simulate each agent).

Park et al.

We compare with Park et al. [121] using up to 16 agents (Tab. 8.3) since it is significantly more computationally efficient than MADER [144]. The comparison is done in 10 environments of size $10\ m \times 10\ m \times 3\ m$ that contain 10 obstacles (*small environment* - Fig. 8.5a). The obstacles are generated randomly as described in [121] . The drone radius is $d_{rad} = 0.15\ m$. This planner requires that all obstacles be modeled as an Octomap [64] (a representation which uses octrees for multiple map resolutions). However, their methods can also be used with voxel grids (like our method) which is trivial and efficient to generate from a point cloud (takes on average $2\ ms$ per pointcloud measurement to generate).

Our method is more **robust** (higher success rate) and has a slightly lower **flight distance**. It also outperforms Park et al. [121] by more than 100% in terms of **flight velocity** and **flight time**. The computation time of our method is on average lower than [121] (stage 1 takes on average $1\ ms$), but the maximum computation time is higher in the case of 16 agents. Finally, Park et al. [121] has a lower **jerk cost** due to the fact that it does not use the full dynamics of the agent and generates very conservative trajectories (despite the fact that, like MADER, there is no limit on the jerk). This, however, does not result in smoother trajectories as shown in Fig. 8.5a, 8.5b, due to the fact that the velocity of the trajectory generated by [121] is much lower.

8.6 . Conclusions and Future Works

In this paper, we presented a novel decentralized, synchronous and real-time method for multi-agent planning. The proposed method uses time-aware Safe Corridors in a Model Predictive Control/Mixed Integer Quadratic Program for-

mulation. Our method is very computationally efficient and is well suited for low compute embedded systems. We compared our planner to 2 recent state-of-the-art methods using different environments and different agent sizes. We showed that our planner is more reliable (higher success rate), more computationally efficient, and generates smoother trajectories with better feasibility guarantees.

9 - MULTI-AGENT EXPLORATION - STATIC ENVIRONMENT

In this paper, we propose a new framework for multi-agent collaborative exploration of unknown environments. The proposed method combines state-of-the-art algorithms in mapping, safe corridor generation and multi-agent planning. It first takes a volume that we want to explore, then proceeds to give the multiple agents different goals in order to explore a voxel grid of that volume. The exploration ends when all voxels are discovered as free or occupied, or there is no path found for the remaining undiscovered voxels. The state-of-the-art planning algorithm uses time-aware Safe Corridors to guarantee intra-agent collision safety as well safety from static obstacles. The presented approach is tested in a state of the art simulator for up to 4 agents.

video: https://youtu.be/v7P7HpBRY50

9.1 . Introduction

Multi-agent exploration has numerous real world applications such as search and rescue, infrastructure inspection and cave exploration. An exploration framework that is computationally efficient and safe is thus tremendously beneficial. It is the purpose of this paper to present a new method for multi-agent exploration that is suitable for low compute embedded systems.

9.1.1 . Related work

In this section we will discuss works in single agent planning and how it relates to our multi-agent planning approach. Also we will discuss the state-of-the-art of multi-agent planning and exploration: their advantages and their shortcomings.

Single-agent planning

Many works address single quadrotor planning. Recent state-of-the-art contributions use Safe Corridors to account for unknown/unexplored part of the environment and guarantee trajectory safety [146]. Safe Corridors are a series of overlapping convex polyhedra that cover only free space. In our work in chapter 6 (page 95), we use Safe Corridors in a MPC/MIQP formulation that is computationally efficient and accounts for drag forces. Our work is based on the same approach with modifications to account for other moving agents.

Multi-agent planning

While some works have addressed multi-agent planning [70], [175], [85], [174], [96], [88], only few works address the problem of low compute and safe multi-agent autonomous exploration [173] while avoiding static obstacles and other exploring agents. In [172] the authors present an autonomous multi-agent planning solution using only on-board resources. The presented approach does not address full volume exploration and focuses only on intra-agent collision avoidance as well as static obstacle collision avoidance.

Exploration

Single quadrotor exploration and frontier analysis/selection (i.e. which side of the map to explore) has been extensively studied in the literature. Recent work [171] has shown significant improvement over other methods such as [27], [163], [14]. The proposed method uses a frontier information structure (FIS) that is maintained incrementally. It is provided to the exploration planner that plans exploration motions in three sequential steps: it first finds global tours that are efficient i.e. a set of efficient movements to cover all existent frontiers; then it selects a local set of optimal viewpoints; finally it generates minimum-time local trajectories. However, it is unclear how to scale this approach to multi-agent planning and whether the computational efficiency would be maintained. It is why we inspired our goal selection method from the Classical method [163] for frontier/goal selection, which scales well for multiple agents i.e. is efficient in terms of computation time.

9.1.2 . Contribution

The main contribution of our paper is a novel exploration framework that is based on a decentralized and synchronous planning method. The planning method takes inspiration mainly from our work in chapter 6 (page 95) for single quadrotor planning and static obstacle avoidance, our work in chapter 8 (page 125) for intra-agent collision avoidance, and [163] for goal selection for each agent. The framework also uses our work on voxel grid generation in chapter 4 (page 51) and Safe Corridor generation in chapter 5 (page 69), which makes it low compute and suitable for embedded systems. The planning framework is tested in simulation using the state-of-the-art Airsim [138] simulator.

9.1.3 . Assumptions

The following assumptions are made:

- The position of each agent is known (communicated between agents) within a certain range of uncertainty. In the simulation, the position of each agent is

known perfectly, but our framework can account for uncertainties by inflating each agents collision radius.

- All agents can communicate with each other within a certain delay. The communication only needs to happen when agents are close to each other.

- All agents can communicate with a central hub within a certain delay. The central hub will be tasked with merging each agent's map with a global map and sending goals to each agent. It can be mobile or fixed, or one of the agents, as long as all agents can communicate with it and it has a certain amount of compute power.

9.2 . Agent Model

Table 9.1: Nomenclature

$\boldsymbol{p}$	position vector x, y, z in the world frame
$\boldsymbol{v}$	velocity vector v_x, v_y, v_z in the world frame
$\boldsymbol{a}$	acceleration vector from thrust and gravity in the world frame
$\boldsymbol{j}$	jerk vector j_x, j_y, j_z in the world frame
$\boldsymbol{D}_{lin\_max}$	linear drag matrix

We use the same simplified/linearized model presented in chapter 6 (page 95) for each agent (Tab. 9.1). The jerk j is the input of the system:

$$\begin{aligned} \dot{\boldsymbol{p}} &= \boldsymbol{v} \\ \dot{\boldsymbol{v}} &= \boldsymbol{a} - \boldsymbol{D}_{lin\_max}\boldsymbol{v} \\ \dot{\boldsymbol{a}} &= \boldsymbol{j} \end{aligned} \tag{9.1}$$

9.3 . The Framework

The framework takes a volume that we want to explore, then proceeds to give all the agents different goals until all the volume is explored. The volume is represented as a voxel grid, and it is considered fully explored when all the voxels are either free or occupied, or no agent can find a path to the remaining unknown voxels. The framework is divided into 2 modules (Figure 9.1):

1. A local module that is run on each agent and that contains mapping, planning and control submodules.

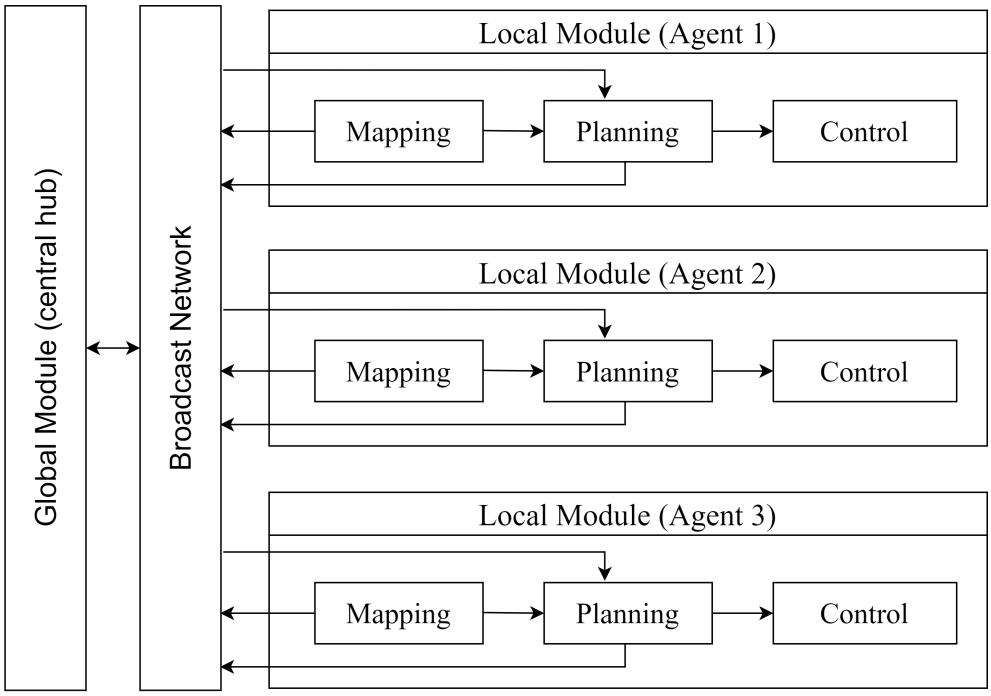


Figure 9.1: We show the global pipeline of the exploration framework with 3 agents. The broadcast network communicates the local maps of each agent to the global module, the goals computed by the global module to the agents, and generated trajectories of all the agents to each agent.

2. A global module that merges all the measurements of all the agents into a single global voxel grid. This module then sends goals to each agent to explore the volume.

Each of these modules as well as each submodule will be detailed separately in this section.

9.3.1 . Local module - run on each agent

The local module is run on each agent an consists of 3 submodules: mapping, planning and control.

Mapping

Each agent is equipped with an omnidirectional lidar that allows is to see in all directions. The output of the lidar is a pointcloud that is transformed into a voxel grid using a GPU accelerated method (described in chapter 4 - page 51). We

remove the points associated to other agents appearing in the lidar scan. This is done by removing all the points of the scan within a certain distance from the center of other agents (which we know due to the communication of trajectories between agents). The local voxel grid moves along with the agent such that the agent's position is always in the center of the grid. Each voxel grid is sent to the global module to be merged into the global voxel grid.

Planning

Each agent plans locally to reach the goal sent to it by the global module. The planning framework is the same as the one presented in chapter 6 (page 95) with modifications made to the Safe Corridor part to become time-aware as presented in chapter 8 (page 125). This allows to avoid collision with other agents as well as static obstacles.

The planning framework is divided into the following steps that are executed sequentially and at a constant rate (Figure 9.2):

1. Generate a global path that avoids static obstacles (section 6.3.1 - page 101).
2. Generate a Safe Corridor (section 6.3.2 - page 102).
3. Generate a time-aware Safe Corridor to avoid collision with other agents (section 8.4.3 - page 130).
4. Generate a local reference trajectory using the global path and the the Safe Corridor (section 8.4.4 - page 133).
5. Solve the MPC/MIQP problem to generate a trajectory close to the local reference trajectory and within the constraints of the time-aware Safe Corridor to guarantee safety (section 8.4.5 - page 136).

Control

Each agent is controlled using a nonlinear MPC [71], with the **acados** toolkit [157]. The controller is the same as the one presented in section 6.4.1 on page 107.

9.3.2 . Global module - run on a central hub

This module takes a volume that we want to explore, merges the measurements/maps of all the agents, and sends a goal to each agent at a constant rate until that volume is fully explored. The volume is represented as a voxel grid and is considered fully explored when all the voxels are either free or occupied, or no agent can find a path to the remaining unknown voxels.

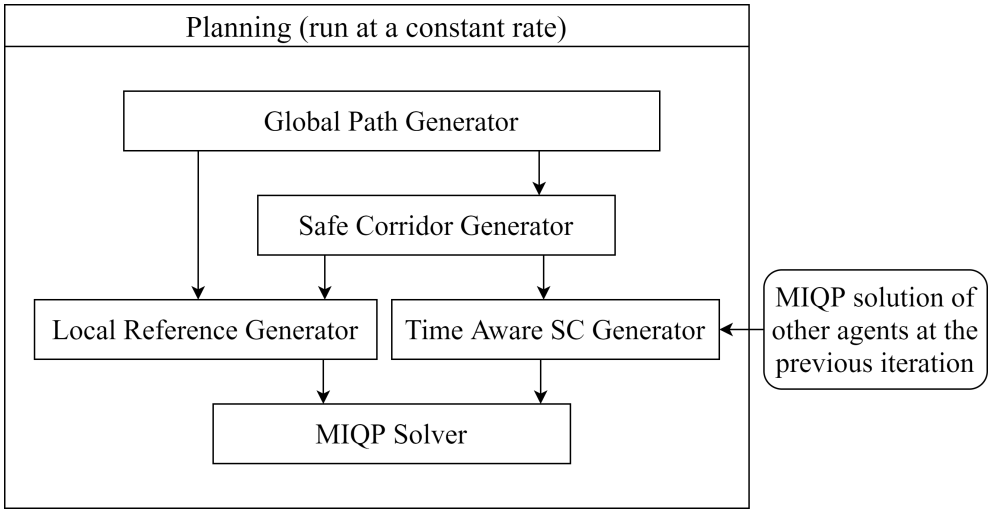


Figure 9.2: We show the global pipeline of the planning framework of a single planning agent.

Merging maps from all agents

Each agent has its local voxel grid that is used for local obstacle avoidance and navigation. When this map is transmitted to the global module, we are able to efficiently merge it into the global map if the voxels of the local and global maps overlap with each others i.e. the vector formed by the origin of the local map and the origin of the global map has its x, y and z components as multiples of the voxel size. This can be achieved during the initialization phase, by setting the initial origin of all maps (local and global) to $(0, 0, 0)$. As presented in chapter 4 (page 51), the origin of the local voxel grid will be always offset in all directions by a multiple of the voxel size to keep the robot at its center.

We only update a voxel of the global grid if its corresponding voxel in the local voxel grid is free or occupied. In the case of static environments, we can add an additional condition to only update a voxel grid in a global grid if its not occupied (and its corresponding voxel in the local voxel grid is free or occupied). The uncertainties in the environment/measurements can be encoded in the voxel grid by setting voxels that we don't know their value to unknown voxels and by using filtering methods.

Computing goals for agents

After updating the global map with the local voxel grids of all the agents, we proceed to compute goals for each agent using [163]. We define a neighbour voxel as a voxel that is reachable by moving at most 1 voxel unit in each direction (i.e. positive and negative x, y and z directions) from the current voxel. The goals are computed by doing the following steps sequentially (Figure 9.3):

1. Find the border voxels: this is done by going over all the voxels of the global map. If a voxel is free and has a neighbour voxel that is unknown, then it is designated as a border voxel.

2. Cluster the border voxels: we form clusters out of the border voxels using the following rule: if two border voxel are neighbors, they belong to the same cluster.

3. Compute cluster centroid: we compute the centroid of each cluster by averaging the positions of all the voxels that belong to that cluster.

4. Compute potential goals: we compute potential goals by finding the voxel border in a cluster that is the closest to its centroid. If multiple voxels have the same closest distance, we chose one randomly.

5. Compute goals: the finals goals are computed by first choosing the closest potential goal to the first agent, and removing it from the potential goals' list so that two agents don't get the same goal. From the remaining goals, we chose the closest one to the second agent and remove it from the list. We continue in this fashion until all the agents have goals, or no potential goals exist anymore.

9.4 . Simulation

The simulation is done in a $30\ m \times 30\ m \times 3\ m$ environment that contains 90 cylinder obstacles of radius $0.35\ m$ and height $3\ m$. Their positions are generated randomly, following a uniform distribution. The environment is generated in Airsim [138]. The Gurobi solver is set to use one thread only as this resulted in faster computation times during our simulations. All testing is done on the Intel Core i7-9750H up to 4.50 GHz CPU and NVIDIA's GeForce RTX 2060 up to 1.62 GHz. We simulate up to 4 agents due to the limited computing power to run the Airsim simulation, mapping/planning/control for each agent and the global module. The agents communicate between themselves using a ROS [142] topic. They also communicate with the global module using a ROS topic.

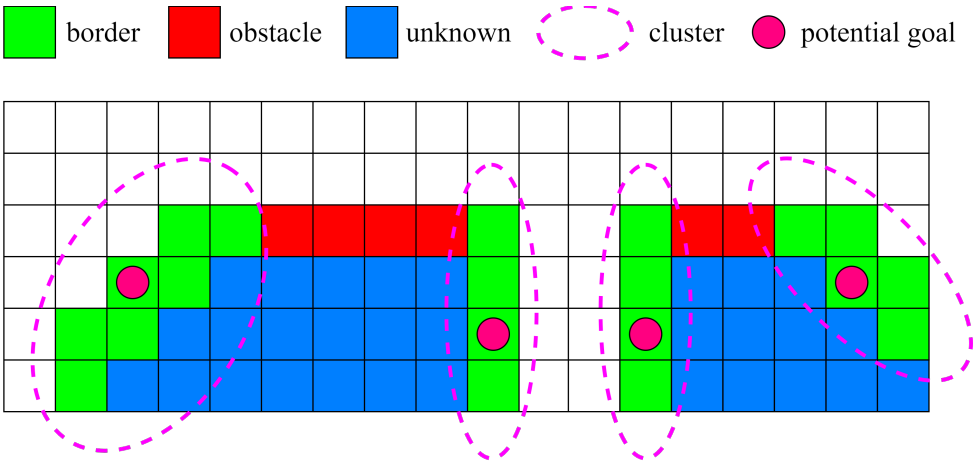


Figure 9.3: We show an example of the borders, clusters and computed potential goals of a voxel grid.

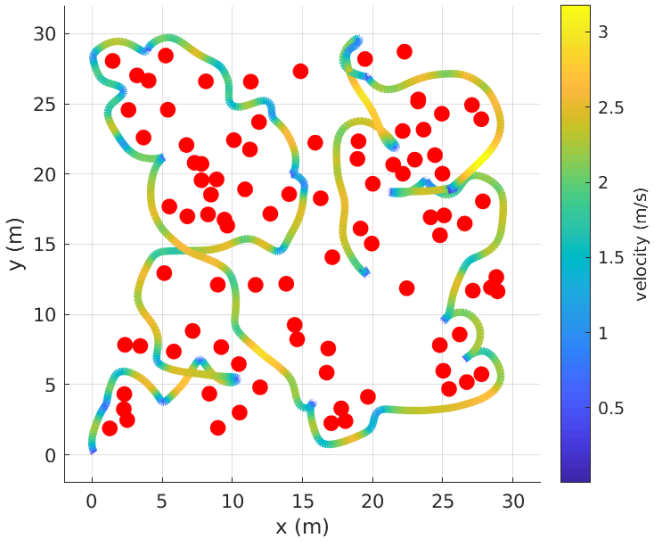
9.4.1 . Mapping parameters

The local map of each agent is of size $20\ m \times 30\ m \times 3\ m$. The global map size is $30\ m \times 30\ m \times 3\ m$. The voxel size used for both maps is $0.3\ m$. The update rate for the local map is 10 Hz (lidar measurement frequency) and the update rate of the global map is 5 Hz. Every time the global map is updated, a new set of goals is computed and sent to each agent. The local map update takes less than $1\ ms$ due to the GPU accelerated method we use (described in chapter 5 - page 69) and the global map update and goal computation takes less than $3\ ms$.

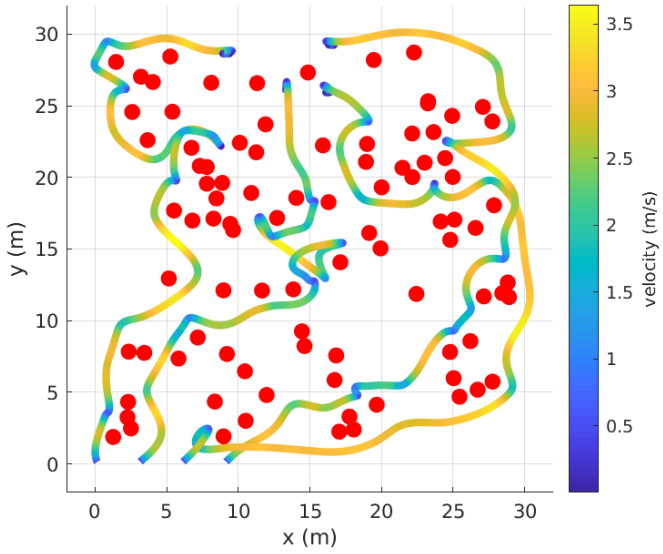
9.4.2 . Planner parameters

We choose the following parameters: $N = 12$, $h = 100\ ms$, $a_{samp} = a_{x,max} = a_{y,max} = a_{z,max} = 0.7 * g$, $a_{z,min} = -g$, $j_{x,max} = j_{y,max} = j_{z,max} = 8\ m/s^2$, $v_{samp} = 3.5\ m/s$, $\boldsymbol{D}_{lin\_max} = diag(1, 1, 1)$, $thresh\_dist = 0.4\ m$, $P_{hor} = 2$. The weight matrices are diagonal: $\boldsymbol{R}_x = diag(200, 200, 200, 0, 0, 0, 0, 0, 0)$, $\boldsymbol{R}_N = diag(100, 100, 100, 0, 0, 0, 0, 0, 0)$ and $\boldsymbol{R}_u = diag(0.01, 0.01, 0.01)$. The drone radius is $d_{rad} = 0.3\ m$. The DMP planner pushes the JPS path $0.6\ m$ (2 voxels) away from obstacles (when possible). The planning frequency is 10 Hz. The planning requires synchronized clocks between the agents. This is satisfied as all the agents use the same CPU clock during the simulation. In a real world situation, the clocks can be synchronized at the beginning of the exploration since the drift is minimal (1 microsecond every second) for the duration of an exploration task. Other solutions include using methods for synchronizing the clocks during the exploration such as [54], [9].

9.4.3 . Controller parameters



(a) Exploration with 1 agent



(b) Exploration with 4 agents

Figure 9.4: The trajectories and velocity of 1 exploring agent and 4 exploring agents on the same map. The initial position of agents are offset by $3\ m$ from each others in the x direction in the case of 4 agents

The frequency of the MPC controller is 20 Hz. The weights of the controller are:

$$\boldsymbol{P} = \boldsymbol{Q}_x = diag(15, 15, 15, 0.01, 0.01, 0.01, 0, 0, 1) \tag{9.2}$$

$$\boldsymbol{R}_u = diag(0.05, 0.1, 0.1, 0.1) \tag{9.3}$$

All parameters are set by approximation/experimentation and may not be optimal. We limit $|\phi| \leq 85\ deg$, $\theta \leq 85\ deg$, $|\dot{\phi}_{cmd}| \leq 120\ deg/s$, $|\dot{\theta}_{cmd}| \leq 120\ deg/s$ and $|\dot{\psi}_{cmd}| \leq 60\ deg/s$.

9.4.4 . Simulation results

Table 9.2: Comparison between 1,2,3 and 4 agents using our framework on 5 randomly generated maps of size $30\ m \times 30\ m \times 3\ m$ and with obstacle density $0.1\ obst/m^2$. We show the **mean / max / standard deviation** per agent of each metric, except for the Safety ratio.

#	Distance (m)	Velocity (m/s)	Exploration time (s)	Comp. time (ms)	Safety ratio
1	239 / 263 / 20.5	1.74 / 3.4 / 0.7	139 / 158 / 15.6	9 / 45 / 3.7	-
2	129 / 137 / 7.72	1.8 / 3.47 / 0.74	78.2 / 83.3 / 4.67	12 / 49.8 / 4.7	1.12
3	81.2 / 110 / 13.4	1.9 / 3.62 / 0.81	49.6 / 60.2 / 6.36	13 / 61 / 5.2	1.03
4	73 / 86.8 / 9.2	2.04 / 3.9 / 0.88	44.9 / 50 / 4.27	14.5 / 64 / 5.3	1.13

We show the results of doing a series of 5 explorations on 5 randomly generated maps using 1,2,3 and 4 agents in Tab. 9.2. We also show the performance of 1 agent and 4 agents on the same map (Fig. 9.4). The starting position of the first agent is $(0, 0, 0)\ m$. When using multiple agents, the starting position of agents are offset from each others $3\ m$ in the x direction (Fig. 9.4b). All agents stop moving when the exploration is done i.e. all voxels are discovered as free or occupied, or there is no path found for the remaining undiscovered voxels.

The average flight distance per agent decreases in a way that is approximately inversely proportional to the number of agents used. The average flight velocity slightly increases since an agent has less chances of being *torn* between two close goals as other agents explore goals that are close to it. The average exploration time which corresponds to the time required to explore the full volume also decreases in an approximately inversely proportional way to the number of agents used. The average computation time increases as we use more agents due to the fact that there are more constraints that are added to each agent's MIQP/MPC that correspond to the separating hyperplanes with other agents. Note that the computation time can be decreased by lowering the number of discretization steps N. Since the maximum computation time is $64\ ms$, this would leave $100 - 64 = 36\ ms$ to communicate the trajectory to other agents using a broadcasting network (in our case a ROS topic). This means that we can modify N to lower the computation

time enough to satisfy communication constraints. Finally we show the safety ratio which corresponds to the ratio: (minimum distance between any two agents at any time during the exploration)/(minimum collision distance). If it is smaller than 1, this means 2 agents collided at a given time during the exploration. The planner satisfies safety constraints when using any number of agents.

9.5 . Conclusion

In this paper, we presented a novel framework for multi-agent autonomous collaborative exploration of unknown environments. The method uses recent advances in Safe Corridor generation, high-speed mapping and multi-agent planning to guarantee safety and computational efficiency during the exploration task. The framework is tested in a state-of-the-art simulator using up to 4 agents.

In the future, we plan to implement the exploration framework in a fully autonomous setting. This requires to add one more submodule in the local module for localization/odometry. This new module would detect nearby agents and exchange the features that it sees with nearby agents (and nearby agents would do the same) so that all agents can localize relative to each other. This submodule would also account for localization uncertainty in the planning submodule to guarantee safety. We also plan on comparing with other exploration methods, which we did not do due to the lack of time in the PhD.

10 - CONCLUSION

10.1 . Work Summary

The objective of this thesis was to improve on some of the bottlenecks in autonomous multirotor navigation from mapping to planning. We first introduced the scope of our work as well as the related works in the **Introduction** section (Chapter 1 - page 11). Then we presented the **Multirotor Model** that was used in our work (Chapter 2 - page 27). The **Contributions** were then presented in Chapters 3 to 9. Finally, we present in this Chapter a **Summary** of the work done as well as the current **Challenges and Future Directions** of autonomous navigation of multirotors.

The main contributions of our works are the following:

- **Chapter 3 - page 33:** A novel multirotor model formulation that takes into account drag forces as well as a novel near time-optimal planning method that uses the aforementioned model. The proposed planning framework also takes into account obstacles by using Safe Corridors. This combination of features (accounting for obstacles and drag forces) was not present in state-of-the-art methods. Our work was used in a drone racing competition [104] and won first place in the qualification and final phases by a large margin (30% decrease in flight time over second place).
- **Chapter 4 - page 51:** An investigation into the use of GPU in the fast and efficient generation of voxel grids for fast navigation. We compared the performance with a typical CPU implementation and showed that it performs $3\times$ better in terms of computation time. We also showed how much each part of the generation benefited from the heavy GPU parallelization, with some parts benefiting from a $50\times$ speedup.
- **Chapter 5 - page 69:** A novel Safe Corridor generation method that is *Safer* than other state-of-the-art methods while also being generic and computationally efficient. The method uses a voxel grid representation of the environment which contributes to its safety attribute (in contrast to a down-sampled pointcloud representation). The method introduces the concept of a *convex grid*: a grid that we inflate around a seed voxel such that the inscribed polyhedron inside it remains convex. The method is then improved upon to make it sense the shape of the obstacle close to it and expand the convex grid accordingly (shape-aware approach).
- **Chapter 6 - page 95:** A novel planning framework for high speed planning of multirotors in unknown environments. This work uses our previous work on voxel grid (chapter 4 - page 51) and Safe Corridor generation (chapter 5

- page 69). The method takes into account drag forces for better feasibility guarantees than the state-of-the-art. Our method performs similarly in terms of flight speed, smoothness, and flight time with respect to the best state-of-the-art method. However, it performs significantly better in terms of computation time (on average 3× lower).

- **Chapter 7 - page 113:** A novel planning framework for multirotors in a dynamic and unknown environment. The method uses a temporal voxel grid as a representation of the environment evolving through a finite horizon of time. We used our work on Safe Corridors to create temporal Safe Corridors that represent the free space that a multirotor can occupy at a given future instant in time. We evaluated the novel approach and showed that it is computationally efficient and generates smooth and fast trajectories.

- **Chapter 8 - page 125:** A novel multi-agent planning framework for multirotors that also avoids static obstacles. It uses a voxel grid representation of the environment. It extends our work on Safe Corridors to make them time-aware and account for other planning agents in addition to the static obstacles. Our work is compared to 2 recent state-of-the-art multi-agent planning method. It outperforms them in terms of computation time, trajectory length and smoothness while generating high speed trajectories.

- **Chapter 9 - page 145:** A novel multi-agent collaborative exploration framework. The framework uses all our previous work on voxel grid generation, Safe Corridor generation and multi-agent planning. We showed how our algorithm performs while exploring a given volume in an unknown environment with respect to the number of exploring agents. The metrics we used were exploration time, traveled distance, trajectory velocity and computation time. We also evaluated the safety of the trajectories in terms of intra-agent collision.

10.2 . Challenges and Future Directions

Despite the immense work that has been done on vehicle autonomy in general, and multirotor autonomy in particular, there are still multiple significant challenges that autonomy algorithms have to solve before reaching the level of robustness and reliability that is required in some real-world applications such as search and rescue and package delivery.

10.2.1 . Dynamic environments

The most glaring challenge is how to deal with dynamic obstacles. Solving this challenge is of tremendous benefits since this would allow autonomous systems to navigate through multiple environments robustly such as urban environments

and forests. This has pushed the research community to organize competitions on dynamic obstacle avoidance [22] to further motivate researchers to work on this issue. This challenge have sub-challenges whose solutions are not clear: how to model obstacles (dynamic and static), and how to choose the appropriate planning method to use with the chosen obstacle model.

Multiple obstacle model/representations have been presented in the state-of-the-art such as representing all obstacles (static and dynamic) as ellispoids [85]. Another one is to represent all obstacles as convex polyhedra [144]. Both of these representations are not trivial to generate, so this may be an area where the state-of-the-art can improve. Another representation would be temporal occupancy/voxel grids where the position of all obstacles at future discrete times are represented by occupancy grids [63].

Many planning approaches exist that can be paired with a chosen obstacle model such as search based methods [90], optimization based methods [85], and methods that use both approaches [144]. So far no method has been able to provide a computationally efficient way to model and track all obstacles accurately and robustly with a computationally efficient way to generate a trajectory that avoids all obstacles robustly. Since these requirements are essential for some real-world applications that can have huge economical and societal benefits, we think it is a good research vector to pursue in the future.

10.2.2 . Stereo/Multiview matching

Multirotors have hard constraints on the weight they can carry since it affects heavily the distance they can cover per one charge of battery. For this reason, lightweight sensors such as cameras are favored in comparison with more heavy sensors such as lidars even if they require more computational overhead to generate a pointcloud representation of the environment that can be used for obstacle avoidance.

Since methods such as stereo matching or multiview matching are the only solution to extract 3D pointclouds from 2D images, they are used heavily in commercial and industrial drone solutions [139] [32]. However, stereo/multiview matching present their own challenges, where one finds himself often trading accuracy for computation speed. A look at the Kitti low-res two-view leaderboard [103] [48] which evaluates stereo matching algorithm as well as their computation times, shows that the best stereo matching algorithms [25] [24] require a level of compute/computation time that is currently intractable for low compute/lightweight drones. However some real-world applications do not require extreme accuracy of obstacle positions, and some methods [143] can provide that accuracy with a relatively low computation time (20 ms on NVIDIA TITAN V GPU). Still, the current state-of-the-art performs badly around thin objects such as wires and thin

branches, which leads to restrictions on where current commercial drones can be used [139] [32].

10.2.3 . Going through Narrow Gaps

One challenge of multirotor autonomous navigation is going through narrow gaps. These gaps may be tilted and multirotors can only pass through them by tilting to the exact angle of the gap. Going through narrow gaps has multiple applications especially in search and rescue where multirotors have to sometimes go through dense forests or ruins of buildings.

One solution to go through narrow gaps involves using morphing quadrotors such as the folding drone presented in [37]. The presented drone has adaptive morphology that allows it to *squeeze* its body by pulling the rotors closer to the main body. While this approach allows the drone to pass through gaps it otherwise couldn't pass through, there are still some tilted narrow gaps that require it to use its full dynamics and tilt to pass through, something that is not addressed by the authors.

The other set of solutions involves using the whole drone dynamics to pass through a narrow tilted window with the drone tilted exactly at the same angle of the window. Some have explored solutions designed specifically and only for going through a tilted window [92] [36] [84]. However these solutions cannot be used to navigate complex environments and avoid obstacles which makes their usefulness limited in exploration/search and rescue tasks.

Other approaches solve the problem of traversing tilted and narrow gaps with a general planning framework that can also be used to navigate and explore environments. Some use search-based motion planning which is computationally intractable for low compute systems [89]. Others use Safe Corridors which are not trivial to generate in narrow gaps and can result in high computational overhead [164]. We believe that a general planning framework that can take into account the drone dynamics and shape, and is able to explore complex environments while also being able to go through tilted and narrow gaps is of huge benefit for some real-world applications. This makes it an interesting and important research vector.

10.2.4 . Machine Learning

Machine learning approaches have been been recently proposed to solve the challenges that classical approaches have yet to overcome such as dynamic obstacle avoidance [56], stereo/multiview matching [143] [25] and going through narrow gaps [84]. Other works use machine learning as an end-to-end solution for drone acrobatics [76] and exploration [94] [95].

In all of the machine learning approaches that generate trajectories/commands

to be executed by the multirotor [84] [94] [95], a common criticism is that of safety. One cannot have guarantees that the generated trajectory will be safe and feasible since deep neural nets operate like a black box. One solution would be to add an algorithm that checks for the safety and feasibility of the generated trajectory/command but this would add computational overhead, and is not an elegant solution.

On the other hand, machine learning approaches that involve the perception module, especially stereo/multiview matching [143], semantic segmentation [79], and affordance segmentation [23], are the only viable approach since they outperform classical methods by a huge margin. We believe that investigating machine learning approaches to make stereo/multiview matching more efficient and accurate is of tremendous benefits since this problem is a significant bottleneck in high speed flight.

Bibliography

[1] F. A. Administration. Drones by the numbers, 2022. https://www.faa.gov/uas/resources/by_the_numbers/, accessed 2022-02-14.

[2] J. Alonso-Mora, T. Naegeli, R. Siegwart, and P. Beardsley. Collision avoidance for aerial vehicles in multi-agent scenarios. *Autonomous Robots*, 39 (1):101–121, 2015.

[3] J. Amanatides, A. Woo, et al. A fast voxel traversal algorithm for ray tracing. In *Eurographics*, volume 87, pages 3–10, 1987.

[4] G. Anandayuvaraj. Drones: The future of business? https://www.forbes.com/sites/forbesbusinesscouncil/2020/06/08/drones-the-future-of-business/, accessed 2022-02-14.

[5] J. A. E. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl. CasADi – A software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, In Press, 2018.

[6] A. Antonini, W. Guerra, V. Murali, T. Sayre-McCord, and S. Karaman. The blackbird uav dataset. *The International Journal of Robotics Research*, 39 (10-11):1346–1364, 2020.

[7] F. Augugliaro, A. P. Schoellig, and R. D'Andrea. Generation of collision-free trajectories for a quadrocopter fleet: A sequential convex programming approach. In *2012 IEEE/RSJ international conference on Intelligent Robots and Systems*, pages 1917–1922. IEEE, 2012.

[8] T. Baca, M. Petrlik, M. Vrba, V. Spurny, R. Penicka, D. Hert, and M. Saska. The mrs uav system: Pushing the frontiers of reproducible research, real-world deployment, and education with autonomous unmanned aerial vehicles. *Journal of Intelligent & Robotic Systems*, 102(1), Apr 2021. ISSN 1573-0409. doi: 10.1007/s10846-021-01383-5. URL http://dx.doi.org/10.1007/s10846-021-01383-5.

[9] S. A. Bakura, A. Lambert, and T. Nowak. Clock synchronization with adaptive weight factor for mobile networks. In *28th Mediterranean Conference on Control and Automation, MED 2020, Saint-Raphaël, France, September 15-18, 2020*, pages 1033–1038. IEEE, 2020. doi: 10.1109/MED48518.2020.9182870. URL https://doi.org/10.1109/MED48518.2020.9182870.

[10] N. Banerjee, X. Long, R. Du, F. Polido, S. Feng, C. G. Atkeson, M. Gennert, and T. Padir. Human-supervised control of the atlas humanoid robot for traversing doors. In *2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids)*, pages 722–729, 2015.

[11] M. Beul and S. Behnke. Analytical time-optimal trajectory generation and control for multirotors. In *2016 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 87–96. IEEE, 2016.

[12] A. Bircher, M. Kamel, K. Alexis, H. Oleynikova, and R. Siegwart. Receding horizon path planning for 3d exploration and surface inspection. *Autonomous Robots*, pages 1–16, 2016.

[13] A. Bircher, M. Kamel, K. Alexis, H. Oleynikova, and R. Siegwart. Receding horizon "next-best-view" planner for 3d exploration. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1462–1468, 2016. doi: 10.1109/ICRA.2016.7487281.

[14] A. Bircher, M. Kamel, K. Alexis, H. Oleynikova, and R. Siegwart. Receding horizon" next-best-view" planner for 3d exploration. In *2016 IEEE international conference on robotics and automation (ICRA)*, pages 1462–1468. IEEE, 2016.

[15] M. Blösch, S. Weiss, D. Scaramuzza, and R. Siegwart. Vision based mav navigation in unknown and unstructured environments. In *2010 IEEE International Conference on Robotics and Automation*, pages 21–28. IEEE, 2010.

[16] R. Bohlin and L. E. Kavraki. Path planning using lazy prm. In *Proceedings 2000 ICRA. Millennium conference. IEEE international conference on robotics and automation. Symposia proceedings (Cat. No. 00CH37065)*, volume 1, pages 521–528. IEEE, 2000.

[17] P. Bonami and J. Lee. Bonmin user's manual. *Numer Math*, 4:1–32, 2007.

[18] R. Bonatti, Y. Zhang, S. Choudhury, W. Wang, and S. Scherer. Autonomous drone cinematographer: Using artistic principles to create smooth, safe, occlusion-free trajectories for aerial filming. *arXiv preprint arXiv:1808.09563*, 2018.

[19] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems journal*, 4(1):25–30, 1965.

[20] M. Burri, H. Oleynikova, M. W. Achtelik, and R. Siegwart. Real-time visual-inertial mapping, re-localization and planning onboard mavs in unknown

environments. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1872–1878. IEEE, 2015.

[21] C. Campos, R. Elvira, J. J. G. Rodríguez, J. M. Montiel, and J. D. Tardós. Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam. *IEEE Transactions on Robotics*, 37(6):1874–1890, 2021.

[22] L. Carlone, M. Ryll, D. Scaramuzza, and A. Loquercio. Perception and action in dynamic environments - ieee-icra 2021 workshop. https://uzh-rpg.github.io/PADE-ICRA2021/, accessed 2022-03-13.

[23] H. Caselles-Dupré, M. Garcia-Ortiz, and D. Filliat. Are standard object segmentation models sufficient for learning affordance segmentation? *arXiv preprint arXiv:2107.02095*, 2021.

[24] X. Cheng, P. Wang, and R. Yang. Learning depth with convolutional spatial propagation network. *IEEE Transactions on Pattern Analysis and Machine Intelligence(T-PAMI)*, pages 1–1, 2019. ISSN 1939-3539. doi: 10.1109/TPAMI.2019.2947374.

[25] X. Cheng, Y. Zhong, M. Harandi, Y. Dai, X. Chang, H. Li, T. Drummond, and Z. Ge. Hierarchical neural architecture search for deep stereo matching. *Advances in Neural Information Processing Systems*, 33, 2020.

[26] S. Chung, A. A. Paranjape, P. Dames, S. Shen, and V. Kumar. A survey on aerial swarm robotics. *IEEE Transactions on Robotics*, 34(4):837–855, 2018.

[27] T. Cieslewski, E. Kaufmann, and D. Scaramuzza. Rapid exploration with multi-rotors: A frontier selection method for high speed flight. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2135–2142. IEEE, 2017.

[28] W. Commons. Polyhedron chamfered 6 edeq, 2018. URL https://upload.wikimedia.org/wikipedia/commons/9/94/Polyhedron_chamfered_6_edeq.png.

[29] R. Deits and R. Tedrake. Computing large convex regions of obstacle-free space through semidefinite programming. In *Algorithmic foundations of robotics XI*, pages 109–124. Springer, 2015.

[30] J. Delmerico, T. Cieslewski, H. Rebecq, M. Faessler, and D. Scaramuzza. Are we ready for autonomous drone racing? the UZH-FPV drone racing dataset. In *IEEE Int. Conf. Robot. Autom. (ICRA)*, 2019.

[31] A. Devo, J. Mao, G. Costante, and G. Loianno. Autonomous single-image drone exploration with deep reinforcement learning and mixed reality. *IEEE Robotics and Automation Letters*, 2022.

[32] DJI. Dji. https://www.dji.com/, accessed 2022-02-14.

[33] D. Duberg and P. Jensfelt. Ufomap: An efficient probabilistic 3d mapping framework that embraces the unknown. *arXiv preprint arXiv:2003.04749*, 2020.

[34] S. Duggal, S. Wang, W.-C. Ma, R. Hu, and R. Urtasun. Deeppruner: Learning efficient stereo matching via differentiable patchmatch. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4384–4393, 2019.

[35] M. Faessler, A. Franchi, and D. Scaramuzza. Differential flatness of quadrotor dynamics subject to rotor drag for accurate tracking of high-speed trajectories. *IEEE Robotics and Automation Letters*, 3(2):620–626, 2017.

[36] D. Falanga, E. Mueggler, M. Faessler, and D. Scaramuzza. Aggressive quadrotor flight through narrow gaps with onboard sensing and computing using active vision. In *2017 IEEE international conference on robotics and automation (ICRA)*, pages 5774–5781. IEEE, 2017.

[37] D. Falanga, K. Kleber, S. Mintchev, D. Floreano, and D. Scaramuzza. The foldable drone: A morphing quadrotor that can squeeze and fly. *IEEE Robotics and Automation Letters*, 4(2):209–216, 2018.

[38] G. Falkovich. *Fluid mechanics: A short course for physicists*. Cambridge University Press, 2011.

[39] M. Fehr, T. Schneider, and R. Siegwart. Visual-inertial teach and repeat powered by google tango. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1–9. IEEE, 2018.

[40] P. Foehn and D. Scaramuzza. Onboard state dependent lqr for agile quadrotors. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6566–6572. IEEE, 2018.

[41] P. Foehn, A. Romero, and D. Scaramuzza. Time-optimal planning for quadrotor waypoint flight. *Science Robotics*, 6(56):eabh1221, 2021.

[42] P. Foehn, D. Brescianini, E. Kaufmann, T. Cieslewski, M. Gehrig, M. Muglikar, and D. Scaramuzza. Alphapilot: Autonomous drone racing. *Autonomous Robots*, 46(1):307–320, 2022.

[43] G. Frison and M. Diehl. Hpipm: a high-performance quadratic programming framework for model predictive control. *IFAC-PapersOnLine*, 53(2):6563–6569, 2020.

[44] G. Frison, H. H. B. Sørensen, B. Dammann, and J. B. Jørgensen. High-performance small-scale solvers for linear model predictive control. In *2014 European Control Conference (ECC)*, pages 128–133. IEEE, 2014.

[45] F. Gao, W. Wu, Y. Lin, and S. Shen. Online safe trajectory generation for quadrotors using fast marching method and bernstein basis polynomial. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 344–351. IEEE, 2018.

[46] F. Gao, W. Wu, J. Pan, B. Zhou, and S. Shen. Optimal time allocation for quadrotor trajectory generation. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4715–4722. IEEE, 2018.

[47] F. Gao, W. Wu, W. Gao, and S. Shen. Flying on point clouds: Online trajectory generation and autonomous navigation for quadrotors in cluttered environments. *Journal of Field Robotics*, 36(4):710–733, 2019.

[48] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun. Kitti 2015 low-res two-view leaderboard. http://www.cvlibs.net/datasets/kitti/eval_scene_flow.php?benchmark=stereo, accessed 2022-03-13.

[49] M. Geisert and N. Mansard. Trajectory generation for quadrotor based systems using numerical optimal control. In *2016 IEEE international conference on robotics and automation (ICRA)*, pages 2958–2964. IEEE, 2016.

[50] P. Geneva, K. Eckenhoff, W. Lee, Y. Yang, and G. Huang. Openvins: A research platform for visual-inertial estimation. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4666–4672. IEEE, 2020.

[51] W. Giernacki, M. Skwierczyński, W. Witwicki, P. Wroński, and P. Kozierski. Crazyflie 2.0 quadrotor as a platform for research and education in robotics and control engineering. In *2017 22nd International Conference on Methods and Models in Automation and Robotics (MMAR)*, pages 37–42, 2017. doi: 10.1109/MMAR.2017.8046794.

[52] W. Guerra, E. Tal, V. Murali, G. Ryou, and S. Karaman. Flightgoggles: Photorealistic sensor simulation for perception-driven robotics using photogrammetry and virtual reality. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6941–6948, 2019. doi: 10.1109/IROS40897.2019.8968116.

[53] L. Gurobi Optimization. Gurobi optimizer reference manual, 2020. URL http://www.gurobi.com.

[54] C. Han, T. Nowak, and A. Lambert. Pulse synchronization for vehicular networks. In *2018 IEEE Intelligent Vehicles Symposium, IV 2018, Changshu, Suzhou, China, June 26-30, 2018*, pages 1125–1130. IEEE, 2018. doi: 10.1109/IVS.2018.8500375. URL https://doi.org/10.1109/IVS.2018.8500375.

[55] L. Han, F. Gao, B. Zhou, and S. Shen. Fiesta: Fast incremental euclidean distance fields for online motion planning of aerial robots. *arXiv preprint arXiv:1903.02144*, 2019.

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

[57] D. Hanover, P. Foehn, S. Sun, E. Kaufmann, and D. Scaramuzza. Performance, precision, and payloads: Adaptive nonlinear mpc for quadrotors. *IEEE Robotics and Automation Letters*, 7(2):690–697, 2021.

[58] D. D. Harabor and A. Grastien. Online graph pruning for pathfinding on grid maps. In *Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.

[59] C. R. Hargraves and S. W. Paris. Direct trajectory optimization using nonlinear programming and collocation. *Journal of guidance, control, and dynamics*, 10(4):338–342, 1987.

[60] G. Hattenberger, M. Bronz, and M. Gorraz. Using the Paparazzi UAV System for Scientific Research. In *IMAV 2014, International Micro Air Vehicle Conference and Competition 2014*, pages pp 247–252, Delft, Netherlands, Aug. 2014. doi: 10.4233/uuid:b38fbdb7-e6bd-440d-93be-f7dd1457be60. URL https://hal-enac.archives-ouvertes.fr/hal-01059642.

[61] M. Hehn and R. D'Andrea. Quadrocopter trajectory generation and control. *IFAC proceedings Volumes*, 44(1):1485–1491, 2011.

[62] A. Hermann, F. Drews, J. Bauer, S. Klemm, A. Roennau, and R. Dillmann. Unified gpu voxel collision detection for mobile manipulation planning. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4154–4160. IEEE, 2014.

[63] S. Hoermann, M. Bach, and K. Dietmayer. Dynamic occupancy grid prediction for urban autonomous driving: A deep learning approach with fully

automatic labeling. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2056–2063. IEEE, 2018.

[64] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard. Octomap: An efficient probabilistic 3d mapping framework based on octrees. *Autonomous robots*, 34(3):189–206, 2013.

[65] W. Hönig, J. A. Preiss, T. K. S. Kumar, G. S. Sukhatme, and N. Ayanian. Trajectory planning for quadrotor swarms. *IEEE Transactions on Robotics*, 34(4):856–869, 2018. doi: 10.1109/TRO.2018.2853613.

[66] A. Jain, S. Casas, R. Liao, Y. Xiong, S. Feng, S. Segal, and R. Urtasun. Discrete residual flow for probabilistic pedestrian behavior prediction. In *Conference on Robot Learning*, pages 407–419. PMLR, 2020.

[67] Jing Chen, Tianbo Liu, and Shaojie Shen. Online generation of collision-free trajectories for quadrotor flight in unknown cluttered environments. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1476–1483, 2016.

[68] S. Jung, S. Hwang, H. Shin, and D. H. Shim. Perception, guidance, and navigation for indoor autonomous drone racing using deep learning. *IEEE Robotics and Automation Letters*, 3(3):2539–2544, 2018.

[69] J.-M. Kai, G. Allibert, M.-D. Hua, and T. Hamel. Nonlinear feedback control of quadrotors exploiting first-order drag effects. *IFAC-PapersOnLine*, 50(1): 8189–8195, 2017.

[70] M. Kamel, J. Alonso-Mora, R. Siegwart, and J. Nieto. Robust collision avoidance for multiple micro aerial vehicles using nonlinear model predictive control. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 236–243. IEEE, 2017.

[71] M. Kamel, M. Burri, and R. Siegwart. Linear vs nonlinear mpc for trajectory tracking applied to rotary wing micro aerial vehicles. *IFAC-PapersOnLine*, 50(1):3463–3469, 2017.

[72] S. Karaman, M. R. Walter, A. Perez, E. Frazzoli, and S. Teller. Anytime motion planning using the rrt. In *2011 IEEE International Conference on Robotics and Automation*, pages 1478–1483. IEEE, 2011.

[73] P. Karkus, S. Cai, and D. Hsu. Differentiable slam-net: Learning particle slam for visual navigation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2815–2825, 2021.

[74] E. Kaufmann, A. Loquercio, R. Ranftl, A. Dosovitskiy, V. Koltun, and D. Scaramuzza. Deep drone racing: Learning agile flight in dynamic environments. In *Conference on Robot Learning*, pages 133–145. PMLR, 2018.

[75] E. Kaufmann, M. Gehrig, P. Foehn, R. Ranftl, A. Dosovitskiy, V. Koltun, and D. Scaramuzza. Beauty and the beast: Optimal methods meet learning for drone racing. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 690–696, 2019. doi: 10.1109/ICRA.2019.8793631.

[76] E. Kaufmann, A. Loquercio, R. Ranftl, M. Müller, V. Koltun, and D. Scaramuzza. Deep drone acrobatics. *arXiv preprint arXiv:2006.05768*, 2020.

[77] M. Kegeleirs, G. Grisetti, and M. Birattari. Swarm slam: Challenges and perspectives. *Frontiers in Robotics and AI*, 8:23, 2021.

[78] S.-p. Lai, M.-l. Lan, Y.-x. Li, and B. M. Chen. Safe navigation of quadrotors with jerk limited trajectory. *Frontiers of Information Technology & Electronic Engineering*, 20(1):107–119, 2019.

[79] F. Lateef and Y. Ruichek. Survey on semantic segmentation using deep learning techniques. *Neurocomputing*, 338:321–348, 2019.

[80] D. R. League. World's fastest drone: Drone racing league. https://www.youtube.com/watch?v=dPpQWIA59Wo, accessed 2022-02-14.

[81] T. Lee, M. Leok, and N. H. McClamroch. Geometric tracking control of a quadrotor uav on se (3). In *49th IEEE conference on decision and control (CDC)*, pages 5420–5425. IEEE, 2010.

[82] S. Li, E. van der Horst, P. Duernay, C. De Wagter, and G. de Croon. Visual model-predictive localization for computationally efficient autonomous racing of a 72-gram drone. arxiv 2019. *arXiv preprint arXiv:1905.10110*, 2019.

[83] S. Li, M. M. Ozo, C. De Wagter, and G. C. de Croon. Autonomous drone race: A computationally efficient vision-based navigation and control strategy. *Robotics and Autonomous Systems*, 133:103621, 2020.

[84] J. Lin, L. Wang, F. Gao, S. Shen, and F. Zhang. Flying through a narrow gap using neural network: an end-to-end planning and control approach. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3526–3533. IEEE, 2019.

[85] J. Lin, H. Zhu, and J. Alonso-Mora. Robust vision-based obstacle avoidance for micro aerial vehicles in dynamic environments. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2682–2688, 2020. doi: 10.1109/ICRA40945.2020.9197481.

[86] C. Liu, C.-Y. Lin, and M. Tomizuka. The convex feasible set algorithm for real time optimization in motion planning. *SIAM Journal on Control and optimization*, 56(4):2712–2733, 2018.

[87] S. Liu, N. Atanasov, K. Mohta, and V. Kumar. Search-based motion planning for quadrotors using linear quadratic minimum time control. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2872–2879. IEEE, 2017.

[88] S. Liu, M. Watterson, K. Mohta, K. Sun, S. Bhattacharya, C. J. Taylor, and V. Kumar. Planning dynamically feasible trajectories for quadrotors using safe flight corridors in 3-d complex environments. *IEEE Robotics and Automation Letters*, 2(3):1688–1695, 2017.

[89] S. Liu, K. Mohta, N. Atanasov, and V. Kumar. Search-based motion planning for aggressive flight in se (3). *IEEE Robotics and Automation Letters*, 3(3): 2439–2446, 2018.

[90] S. Liu, K. Mohta, N. Atanasov, and V. Kumar. Towards search-based motion planning for micro aerial vehicles. *arXiv preprint arXiv:1810.03071*, 2018.

[91] Lockheed-Martin. Alphapilot – lockheed martin ai drone racing innovation challenge. https://www.herox.com/alphapilot, accessed 2019-09-14.

[92] G. Loianno, C. Brunner, G. McGrath, and V. Kumar. Estimation, control, and planning for aggressive flight with a small quadrotor with a single camera and imu. *IEEE Robotics and Automation Letters*, 2(2):404–411, 2016.

[93] G. Loianno, C. Brunner, G. McGrath, and V. Kumar. Estimation, control, and planning for aggressive flight with a small quadrotor with a single camera and imu. *IEEE Robotics and Automation Letters*, 2(2):404–411, 2017. doi: 10.1109/LRA.2016.2633290.

[94] A. Loquercio, A. I. Maqueda, C. R. del Blanco, and D. Scaramuzza. Dronet: Learning to fly by driving. *IEEE Robotics and Automation Letters*, 3(2): 1088–1095, 2018. doi: 10.1109/LRA.2018.2795643.

[95] A. Loquercio, E. Kaufmann, R. Ranftl, M. Müller, V. Koltun, and D. Scaramuzza. Learning high-speed flight in the wild. *Science Robotics*, 6(59): eabg5810, 2021.

[96] C. E. Luis, M. Vukosavljev, and A. P. Schoellig. Online trajectory generation with distributed model predictive control for multi-robot motion planning. *IEEE Robotics and Automation Letters*, 5(2):604–611, 2020. doi: 10.1109/LRA.2020.2964159.

[97] R. Madaan, N. Gyde, S. Vemprala, M. Brown, K. Nagami, T. Taubner, E. Cristofalo, D. Scaramuzza, M. Schwager, and A. Kapoor. Airsim drone racing lab. In H. J. Escalante and R. Hadsell, editors, *Proceedings of the NeurIPS 2019 Competition and Demonstration Track*, volume 123 of *Proceedings of Machine Learning Research*, pages 177–191. PMLR, 08–14 Dec 2020. URL https://proceedings.mlr.press/v123/madaan20a.html.

[98] R. Mahony, V. Kumar, and P. Corke. Multirotor aerial vehicles: Modeling, estimation, and control of quadrotor. *IEEE Robotics Automation Magazine*, 19(3):20–32, 2012. doi: 10.1109/MRA.2012.2206474.

[99] Y. Mao, M. Szmuk, X. Xu, and B. Acikmese. Successive convexification: A superlinearly convergent algorithm for non-convex optimal control problems. *arXiv preprint arXiv:1804.06539*, 2018.

[100] L. Meier, P. Tanskanen, L. Heng, G. H. Lee, F. Fraundorfer, and M. Pollefeys. Pixhawk: A micro aerial vehicle design for autonomous flight using onboard computer vision. *Auton. Robots*, 33(1–2):21–39, aug 2012. ISSN 0929-5593. doi: 10.1007/s10514-012-9281-4. URL https://doi.org/10.1007/s10514-012-9281-4.

[101] 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. IEEE, 2011.

[102] D. Mellinger, N. Michael, and V. Kumar. Trajectory generation and control for precise aggressive maneuvers with quadrotors. *The International Journal of Robotics Research*, 31(5):664–674, 2012.

[103] M. Menze and A. Geiger. Object scene flow for autonomous vehicles. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[104] Microsoft-Airsim. Game of drones competition. https://github.com/microsoft/AirSim-NeurIPS2019-Drone-Racing, accessed 2019-09-14.

[105] K. Mohta, M. Watterson, Y. Mulgaonkar, S. Liu, C. Qu, A. Makineni, K. Saulnier, K. Sun, A. Zhu, J. Delmerico, K. Karydis, N. Atanasov, G. Loianno, D. Scaramuzza, K. Daniilidis, C. J. Taylor, and V. Kumar. Fast, autonomous flight in gps-denied and cluttered environments. *Journal of Field Robotics*, 35(1):101–120, 2018. doi: https://doi.org/10.1002/rob.21774. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21774.

[106] H. Moon, J. Martinez-Carranza, T. Cieslewski, M. Faessler, D. Falanga, A. Simovic, D. Scaramuzza, S. Li, M. Ozo, C. De Wagter, G. Croon,

S. Hwang, S. Jung, H. Shim, H. Kim, M. Park, T.-C. Au, and s.-j. Kim. Challenges and implemented technologies used in autonomous drone racing. *Intelligent Service Robotics*, 12, 04 2019. doi: 10.1007/s11370-018-00271-6.

[107] A. I. Mourikis, S. I. Roumeliotis, et al. A multi-state constraint kalman filter for vision-aided inertial navigation. In *ICRA*, volume 2, page 6, 2007.

[108] M. W. Mueller, M. Hehn, and R. D'Andrea. A computationally efficient algorithm for state-to-state quadrocopter trajectory generation and feasibility verification. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3480–3486, 2013. doi: 10.1109/IROS.2013.6696852.

[109] Newswire. Global drone service market report 2019: Market is expected to grow from usd 4.4 billion in 2018 to usd 63.6 billion by 2025, at a cagr of 55.9 https://markets.businessinsider.com/news/stocks/global-drone-service-market-report-2019-market-is-expected-to-grow-from-usd-4-4-billion-in-2018-to-usd-63-6-billion-by-2025-at-a-cagr-of-55-9-1028147695, accessed 2022-02-14.

[110] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[111] B. Nisar, P. Foehn, D. Falanga, and D. Scaramuzza. Vimo: Simultaneous visual inertial model-based odometry and force estimation. *IEEE Robotics and Automation Letters*, 4(3):2785–2792, 2019. doi: 10.1109/LRA.2019.2918689.

[112] J. Nocedal. Knitro: An integrated package for nonlinear optimization. In *Large-Scale Nonlinear Optimization*, pages 35–60. Springer, 2006.

[113] NVIDIA. Cuda c programming guide, . https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#global-memory-3-0, accessed 2020-07-30.

[114] NVIDIA. Nvidia jetson nano, . https://developer.nvidia.com/embedded/jetson-nano-developer-kit, accessed 2020-05-09.

[115] P. Oettershagen, T. Stastny, T. Mantel, A. Melzer, K. Rudin, P. Gohl, G. Agamennoni, K. Alexis, and R. Siegwart. Long-endurance sensing and mapping using a hand-launchable solar-powered uav. *Field and Service Robotics*, pages 441–454, 2016.

[116] H. Oleynikova, M. Burri, Z. Taylor, J. Nieto, R. Siegwart, and E. Galceran. Continuous-time trajectory optimization for online uav replanning.

In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5332–5339. IEEE, 2016.

[117] H. Oleynikova, Z. Taylor, M. Fehr, R. Siegwart, and J. Nieto. Voxblox: Incremental 3d euclidean signed distance fields for on-board mav planning. In *2017 Ieee/rsj International Conference on Intelligent Robots and Systems (iros)*, pages 1366–1373. IEEE, 2017.

[118] H. Oleynikova, C. Lanegger, Z. Taylor, M. Pantic, A. Millane, R. Siegwart, and J. Nieto. An open-source system for vision-based micro-aerial vehicle mapping, planning, and flight in cluttered environments. *Journal of Field Robotics*, 37(4):642–666, Apr 2020. ISSN 1556-4967. doi: 10.1002/rob.21950. URL http://dx.doi.org/10.1002/rob.21950.

[119] S. Omari, M.-D. Hua, G. Ducard, and T. Hamel. Nonlinear control of vtol uavs incorporating flapping dynamics. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2419–2425. IEEE, 2013.

[120] J. Park, J. Kim, I. Jang, and H. J. Kim. Efficient multi-agent trajectory planning with feasibility guarantee using relative bernstein polynomial. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 434–440. IEEE, 2020.

[121] J. Park, D. Kim, G. C. Kim, D. Oh, and H. J. Kim. Online distributed trajectory planning for quadrotor swarm with feasibility guarantee using linear safe corridor. *IEEE Robotics and Automation Letters*, 7(2):4869–4876, 2022. doi: 10.1109/LRA.2022.3152702.

[122] Parrot. Parrot anafi ai. https://www.parrot.com/en/drones/anafi-ai, accessed 2022-02-14.

[123] T. Qin, P. Li, and S. Shen. Vins-mono: A robust and versatile monocular visual-inertial state estimator. *IEEE Transactions on Robotics*, 34(4):1004–1020, 2018.

[124] X. Qiu, H. Zhang, W. Fu, C. Zhao, and Y. Jin. Monocular visual-inertial odometry with an unbiased linear system model and robust feature tracking front-end. *Sensors*, 19(8):1941, 2019.

[125] X. Qiu, H. Zhang, and W. Fu. Lightweight hybrid visual-inertial odometry with closed-form zero velocity update. *Chinese Journal of Aeronautics*, 2020.

[126] S. Rajendran and S. Srinivas. Air taxi service for urban mobility: A critical review of recent developments, future challenges, and opportunities. *Transportation research part E: logistics and transportation review*, 143:102090, 2020.

[127] P. Reist and R. Tedrake. Simulation-based lqr-trees with input and state constraints. In *2010 IEEE International Conference on Robotics and Automation*, pages 5504–5510. IEEE, 2010.

[128] E. Reyes-Valeria, R. Enriquez-Caldera, S. Camacho-Lara, and J. Guichard. Lqr control for a quadrotor using unit quaternions: Modeling and simulation. In *CONIELECOMP 2013, 23rd International Conference on Electronics, Communications and Computing*, pages 172–178. IEEE, 2013.

[129] C. Richter, A. Bry, and N. Roy. Polynomial trajectory planning for aggressive quadrotor flight in dense indoor environments. In *Robotics Research*, pages 649–666. Springer, 2016.

[130] K. Robotics. Mrsl jump point search planning library v1.1. https://github.com/KumarRobotics/jps3d, accessed 2020-05-09.

[131] G. Rousseau, C. S. Maniu, S. Tebbani, M. Babel, and N. Martin. Minimum-time b-spline trajectories with corridor constraints. application to cinematographic quadrotor flight plans. *Control Engineering Practice*, 89:190–203, 2019.

[132] S. Russell and P. Norvig. Artificial intelligence: a modern approach. 2002.

[133] R. B. Rusu and S. Cousins. 3d is here: Point cloud library (pcl). In *2011 IEEE international conference on robotics and automation*, pages 1–4. IEEE, 2011.

[134] M. Ryll, J. Ware, J. Carter, and N. Roy. Efficient trajectory planning for high speed flight in unknown environments. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 732–738. IEEE, 2019.

[135] I. Sa, M. Kamel, M. Burri, M. Bloesch, R. Khanna, M. Popović, J. Nieto, and R. Siegwart. Build your own visual-inertial drone: A cost-effective and open-source autonomous drone. *IEEE Robotics Automation Magazine*, 25 (1):89–103, 2018. doi: 10.1109/MRA.2017.2771326.

[136] D. C. Schedl, I. Kurmi, and O. Bimber. An autonomous drone for search and rescue in forests using airborne optical sectioning. *Science Robotics*, 6 (55):eabg1188, 2021.

[137] M. Schreiber, V. Belagiannis, C. Gläser, and K. Dietmayer. Motion estimation in occupancy grid maps in stationary settings using recurrent neural networks. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 8587–8593. IEEE, 2020.

[138] S. Shah, D. Dey, C. Lovett, and A. Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, 2017. URL https://arxiv.org/abs/1705.05065.

[139] Skydio. Skydio. https://www.skydio.com/, accessed 2022-02-14.

[140] X. Song, G. Yang, X. Zhu, H. Zhou, Z. Wang, and J. Shi. Adastereo: A simple and efficient approach for adaptive stereo matching. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10328–10337, 2021.

[141] S. Spedicato and G. Notarstefano. Minimum-time trajectory generation for quadrotors in constrained environments. *IEEE Transactions on Control Systems Technology*, 26(4):1335–1344, 2017.

[142] Stanford Artificial Intelligence Laboratory et al. Robotic operating system. URL https://www.ros.org.

[143] V. Tankovich, C. Hane, Y. Zhang, A. Kowdle, S. Fanello, and S. Bouaziz. Hitnet: Hierarchical iterative tile refinement network for real-time stereo matching. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14362–14372, 2021.

[144] J. Tordesillas and J. P. How. Mader: Trajectory planner in multiagent and dynamic environments. *IEEE Transactions on Robotics*, pages 1–14, 2021. doi: 10.1109/TRO.2021.3080235.

[145] J. Tordesillas, B. T. Lopez, J. Carter, J. Ware, and J. P. How. Real-time planning with multi-fidelity models for agile flights in unknown environments. *arXiv preprint arXiv:1810.01035*, 2018.

[146] J. Tordesillas, B. T. Lopez, and J. P. How. Faster: Fast and safe trajectory planner for flights in unknown environments. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1934–1940, 2019. doi: 10.1109/IROS40897.2019.8968021.

[147] C. Toumieh and A. Lambert. Gpu accelerated voxel grid generation for fast mav exploration. 2021. doi: 10.48550/ARXIV.2112.13169. URL https://arxiv.org/abs/2112.13169.

[148] C. Toumieh and A. Lambert. High-speed planning in unknown environments for multirotors considering drag. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7844–7850, 2021. doi: 10.1109/ICRA48506.2021.9560773.

[149] C. Toumieh and A. Lambert. Voxel-grid based convex decomposition of 3d space for safe corridor generation. *Journal of Intelligent & Robotic Systems*, 105(4):1–13, 2022.

[150] C. Toumieh and A. Lambert. Multirotor planning in dynamic environments using temporal safe corridors, 2022. URL https://arxiv.org/abs/2208.06950.

[151] C. Toumieh and A. Lambert. Mace: Multi-agent autonomous collaborative exploration of unknown environments, 2022. URL https://arxiv.org/abs/2208.06949.

[152] C. Toumieh and A. Lambert. Decentralized multi-agent planning using model predictive control and time-aware safe corridors. *IEEE Robotics and Automation Letters*, pages 1–8, 2022. doi: 10.1109/LRA.2022.3196777.

[153] C. Toumieh and A. Lambert. Near time-optimal trajectory generation for multirotors using numerical optimization and safe corridors. *Journal of Intelligent & Robotic Systems*, 105(1):1–10, 2022.

[154] C. Toumieh and A. Lambert. Shape-aware safe corridors generation using voxel grids, 2022.

[155] O. Tutsoy, D. E. Barkana, and K. Balikci. A novel exploration-exploitation-based adaptive law for intelligent model-free control approaches. *IEEE Transactions on Cybernetics*, pages 1–9, 2021. doi: 10.1109/TCYB.2021.3091680.

[156] J. Verbeke and J. D. Schutter. Experimental maneuverability and agility quantification for rotary unmanned aerial vehicle. *International Journal of Micro Air Vehicles*, 10(1):3–11, 2018.

[157] R. Verschueren, G. Frison, D. Kouzoupis, N. van Duijkeren, A. Zanelli, R. Quirynen, and M. Diehl. Towards a modular software package for embedded optimization. In *Proceedings of the IFAC Conference on Nonlinear Model Predictive Control (NMPC)*, 2018.

[158] A. Wächter and L. Biegler. Ipopt-an interior point optimizer, 2009.

[159] W. Wang, D. Zhu, X. Wang, Y. Hu, Y. Qiu, C. Wang, Y. Hu, A. Kapoor, and S. Scherer. Tartanair: A dataset to push the limits of visual slam. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4909–4916. IEEE, 2020.

[160] X. Wang, C. Wang, B. Liu, X. Zhou, L. Zhang, J. Zheng, and X. Bai. Multi-view stereo in the deep learning era: A comprehensive revfiew. *Displays*, 70: 102102, 2021.

[161] M. Watterson, S. Liu, K. Sun, T. Smith, and V. Kumar. Trajectory optimization on manifolds with applications to so (3) and r3xs2. In *Robotics: Science and Systems*, 2018.

[162] H. Xu and J. Zhang. Aanet: Adaptive aggregation network for efficient stereo matching. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1959–1968, 2020.

[163] B. Yamauchi. A frontier-based approach for autonomous exploration. In *Proceedings 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA'97.'Towards New Computational Principles for Robotics and Automation'*, pages 146–151. IEEE, 1997.

[164] S. Yang, B. He, Z. Wang, C. Xu, and F. Gao. Whole-body real-time motion planning for multicopters. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9197–9203, 2021. doi: 10.1109/ICRA48506.2021.9561526.

[165] Z. Yang, Z. Ren, Q. Shan, and Q. Huang. Mvs2d: Efficient multi-view stereo via attention-driven 2d convolutions. *arXiv preprint arXiv:2104.13325*, 2021.

[166] D. J. Yeong, G. Velasco-Hernandez, J. Barry, J. Walsh, et al. Sensor and sensor fusion technology in autonomous vehicles: A review. *Sensors*, 21(6): 2140, 2021.

[167] M. Yguel, O. Aycard, and C. Laugier. Efficient gpu-based construction of occupancy girds using several laser range-finders. *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 105–110, 2006.

[168] Z. Yu and S. Gao. Fast-mvsnet: Sparse-to-dense multi-view stereo with learned propagation and gauss-newton refinement. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1949–1958, 2020.

[169] B. Zhou, F. Gao, L. Wang, C. Liu, and S. Shen. Robust and efficient quadrotor trajectory generation for fast autonomous flight. *IEEE Robotics and Automation Letters*, 4(4):3529–3536, 2019.

[170] B. Zhou, J. Pan, F. Gao, and S. Shen. Raptor: Robust and perception-aware trajectory replanning for quadrotor fast flight. *IEEE Transactions on Robotics*, 37(6):1992–2009, 2021.

[171] B. Zhou, Y. Zhang, X. Chen, and S. Shen. Fuel: Fast uav exploration using incremental frontier structure and hierarchical planning. *IEEE Robotics and Automation Letters*, 6(2):779–786, 2021. doi: 10.1109/LRA.2021.3051563.

[172] X. Zhou, J. Zhu, H. Zhou, C. Xu, and F. Gao. Ego-swarm: A fully autonomous and decentralized quadrotor swarm system in cluttered environments, 2021.

[173] X. Zhou, X. Wen, Z. Wang, Y. Gao, H. Li, Q. Wang, T. Yang, H. Lu, Y. Cao, C. Xu, et al. Swarm of micro flying robots in the wild. *Science Robotics*, 7(66):eabm5954, 2022.

[174] H. Zhu and J. Alonso-Mora. B-uavc: Buffered uncertainty-aware voronoi cells for probabilistic multi-robot collision avoidance. In *2019 International Symposium on Multi-Robot and Multi-Agent Systems (MRS)*, pages 162–168, 2019. doi: 10.1109/MRS.2019.8901092.

[175] H. Zhu and J. Alonso-Mora. Chance-constrained collision avoidance for mavs in dynamic environments. *IEEE Robotics and Automation Letters*, 4 (2):776–783, 2019.

[176] D. Zou, P. Tan, and W. Yu. Collaborative visual slam for multiple agents: A brief survey. *Virtual Reality & Intelligent Hardware*, 1(5):461–482, 2019.